# Ceph

## 之RADOS设计原理与实现

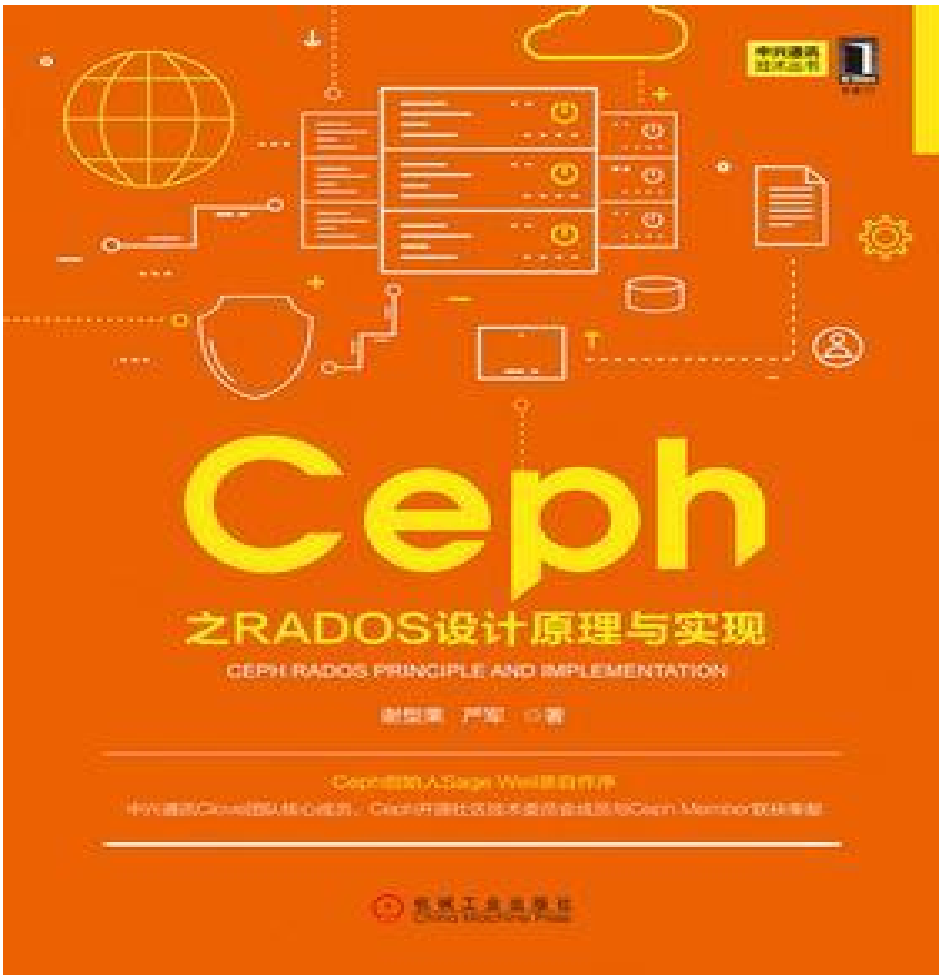
CEPH RADOS PRINCIPLE AND IMPLEMENTATION

谢型果 严军 著

Ceph创始人Sage Weil推荐序

中兴通讯Ceph团队核心成员，Ceph开源社区技术委员会成员与Open Monitor权威专家

机械工业出版社
China Machine Press


Ceph
之RADOS设计原理与实现
CEPH RADOS PRINCIPLE AND IMPLEMENTATION

# Ceph

## 之RADOS设计原理与实现

CEPH RADOS PRINCIPLE AND IMPLEMENTATION

谢型果 严军 著

Ceph创始人Sage Weil推荐作序

中兴通讯Ceph团队核心成员，Ceph中国社区技术委员会成员与Ceph Member联袂推荐

机械工业出版社
China Machine Press

# 版权信息

书名：Ceph之RADOS设计原理与实现
作者：谢型果,严军
排版：昷一
出版社：机械工业出版社
出版时间：2019-01-01
ISBN：9787111613893

# 序1

It has been my pleasure to see the growth in popularity of Ceph in China over the last several years.More than any other region，I see organizations recognizing the benefits of an open source，distributed，and scalable storage platform for file，block，and object workloads.Open source is a huge opportunity for organizations to work cooperatively to improve software and to benefit from each others'hard work.

The Chinese Ceph community has become an impressive force in the larger Ceph ecosystem.In March of 2018 the first Cephalocon conference took place in Beijing.The event spanned two days，included more than 60 speakers，and drew more than 1000 attendees.I was incredibly impressed with the amount of technical content that was presented by the local community，and with the scope of Ceph deployments in China.I look forward to returning soon for the next Cephalocon or Ceph Day event！

My primary goal is to grow and build the Ceph developer community in China.As more organizations rely on Ceph for their storage infrastructure，the pool of serious Ceph users grows，and every user presents a new opportunity to become involved in Ceph development.We face a number of challenges that make it hard for Chinese developers to participate in the upstream Ceph development community，including language，reliable and unrestricted internet access to common collaboration tools(including chat，video conferencing，and video

archives)，and time zones.However，I think these challenges can be overcome by building a community nexus that is centered in China，using what languages and tools are most convenient and natural，and by facilitating communication and collaboration with the broader community through experienced developers like Xie Xingguo.Books like this one that help new developers become familiar with Ceph are a critical part in this effort.

For the Ceph project I see a few key focus area for the next few years：(1)usability，(2)performance，(3)integration with emerging container platforms(like Kubernetes)，and(4)hybrid and multi-cloud capabilities.

Ceph has developed a reputation for being hard，and that has slowed adoption.This is a problem we need to fix—not only to make Ceph accessible to a broader set of users，but also to make the system manageable at large scales by small teams of operators.As storage demands grow，the scale of storage infrastructure will grow too，and making Ceph automate as much of its day to day operations as possible will become increasingly important.

The storage hardware landscape is also changing.Ceph is pure software，which means it can run of a broad range of systems and storage devices.However，as the industry continues to shift to solid state storage technologies like NAND flash and persistent memory technologies，the software will need to evolve and adapt to capture the performance of that hardware.A critical strategic effort is now underway to reimplement key parts of the Ceph data path using new software technologies，including SeaStar，SPDK，and DPDK.The success of this effort will depend on the participation and efforts of many community members.

The Kubernetes container platform is increasingly looking like it will dominate the next wave of IT infrastructure，and with any scale-out infrastructure platform，scale-out software-defined storage technologies

will be needed.The Ceph community is working hard to ensure that Ceph is the easiest and obvious choice for storage in this growing ecosystem through projects like Rook，but with any growing and evolving technology，the needs are changing all of the time.Sustained engagement with this community—and other emerging infrastructure projects—is needed to make sure we can meet the needs of users.

Finally，modern enterprises are increasingly deploying infrastructure across a range of different data centers，clouds，geographic regions，and regulatory regimes.Simply storing data reliably within a single cluster is no longer sufficient to solve businesses'storage problems.Federation capabilities，disaster recovery，and data management services that enable portability of applications—and their data—across clouds and data centers is necessary to provide users with the freedom from being locked-in to specific cloud providers or enterprise platforms.The next phase of evolution for Ceph will be in providing the underlying storage features that enable replication and migration of data sets across clusters and clouds for file，block，and especially object storage.Object storage in particular will be the preferred interface for the next generation of"cloud native"applications，and presents a new opportunity for Ceph to provide higher-level data services for managing the placement，replication，tiering，and migration of data across clouds in an automated，policy-driven way.

More than ever before I am excited about what is coming next for Ceph and the future of storage.I hope that you can be part of that journey！

Sage Weil，Ceph创始人

# 序2

随着分布式源代码托管网站GitHub风靡全球，开源正以燎原之势席卷整个软件世界，成为软件工程的新信条。

开源（软件活动）诞生了众多世界级的明星项目，其中最负盛名的莫过于Linux，它不仅在构建大型数据中心所必需的服务器操作系统领域占据统治地位，而且也是举世闻名的Android智能手机操作系统的核心。可以说Linux的出现在很大程度上加速了信息时代，特别是大数据时代的到来。另一个例子则是近年来闪电崛起、已成为云计算事实上标准的OpenStack。云计算的普及，在不断降低企业生产成本的同时也成倍提升着企业的生产效率。而OpenStack不但在私有云领域独领风骚，在公有云领域，全球有超过60个公有云也采用或者借鉴了它的技术架构。随着开源力量的进一步壮大，可以预见，这样的例子将会越来越多。

除了明星开源软件对于整个行业乃至整个世界产生的羊群效应，开源本身也代表了一种先进的软件开发与管理理念。

首先，开源代表开放、透明、高效、兼容并包与拒绝重复。由于借助的是全世界开发者（也包含最终用户）的力量，开源产品可以获得最广泛的需求收集、技术讨论与用户体验及反馈渠道，也得以最大程度地避免无效或者重复劳动。因此开源产品更新换代速度快、通用

性好并且质量可靠，同时由于开放源码，可以保证用户隐私不被窃取，这在“数据即价值”的大数据时代显得尤为可贵。

其次，开源社区人才辈出、大师云集。与大师同行，接受大师熏陶，与开源社区共成长，是快速提升自身的不二法门。

最后，开源社区的运作理念，例如提倡高度自动化、自管理、自组织等，其实是一种更高层次的敏捷。以开源社区的方式进行产品研发，无论对于产品本身，还是对于参与其中的每个人，无疑都会大有裨益。

中兴通讯深刻认同并重视开源的力量。多年来，我们持续投入并深耕于包括OpenStack、OpenNFV、Kubernetes、Ceph等在内的多个与云计算相关的全球性主流开源组织，极大地影响着，甚至引导着相关行业标准制定，成为推动行业变革、加速行业进化的关键性力量。作为中兴通讯开源领域的旗帜人物，本书作者之一谢型果是Ceph开源社区的技术委员会成员，他所带领的团队一直奋斗在社区最前沿，连续多个版本代码贡献数量位列前三。同时，中兴通讯基于Ceph的分布式存储产品商用规模也已经超过10 PB。以Ceph为代表，中兴通讯基于开源打造的系列产品经过理论与实践双重检验，真正做到了源于开源、高于开源，同时不忘回馈开源。

开源改变世界，我很荣幸能够见证这一历史时刻的到来并参与其中。

杨日　中兴通讯无线研究院副院长兼大数据研发中心主任

# 序3

随着ICT产业不断融合与发展，传统电信运营商开始逐步采用云计算、虚拟化、软件定义网络等IT领域新技术优化通信网络架构，通过网络功能虚拟化（NFV）等方式对软、硬件进行解耦，以提高资源利用率，实现网元快速部署与升级、降低维护和运营成本等目标，也为即将到来的5G移动通信提供更具弹性的基础设施平台。

作为传统CT领域的重要设备厂商，中兴通讯在当今开放、共享、数字经济转型的时代大背景下，始终重视并持续关注新技术的发展，积极推动ICT产业技术融合，致力于为客户提供更加优质的数字化与信息化服务。在寻求自身技术转型的过程中，中兴通讯十分注重云计算相关基础设施的建设。Ceph作为当今最先进的分布式统一存储平台之一，已成为私有云/NFVI事实上的标准——OpenStack的默认存储后端，也是中兴通讯重点关注和投入的方向。

《Ceph之RADOS设计原理与实现》是中兴通讯第二本原创性Ceph著作。中兴通讯Ceph团队从传统存储研发转型，近年来聚焦于以Ceph为代表的分布式存储领域，在与开源社区保持紧密协作的同时，高效支撑了公司国内外多个电信云商用局点的存储解决方案，具备较高的理论水平和丰富的实践经验。两位作者同为Ceph社区的核心贡献者（Ceph Member），长期工作在社区一线，无论是贡献数量还是质

量都在国内首屈一指。这次他们能够付出大量时间和精力将多年来的研究心得与实践经验编撰成册，非常难能可贵。

本书理论结合实际，深入浅出地介绍了Ceph系统的核心组件——RADOS的设计理念与实现方式，相信无论是从事Ceph研发还是运维工作的读者，都将从中获益匪浅。

陈新宇　中兴通讯电信云及核心网产品副总经理（主管研发）

# 前言

2018年3月，全球Cephers的盛会——Cephalocon APAC 2018在北京举行。来自RedHat、SUSE、Intel、中兴通讯、中国移动等Ceph生态联盟成员的1000多位Ceph开发者、使用者和爱好者共聚一堂，探讨Ceph的现状与未来，彰显了Ceph开源社区的蓬勃生机。

时光荏苒，自Ceph由Sage A.Weil在博士论文提出以来，十多年间，已经从一个默默无闻的学生作品成长为分布式存储领域最具活力与领导力的开源项目。据Ceph官方不完全统计，在世界范围内，目前已有超过100家公司（机构）研究与使用Ceph，其中不乏欧洲原子能研究组织（CERN）这样知名的全球性科研机构和Yahoo、阿里巴巴等著名的互联网公司。可见，作为分布式软件定义存储的标杆，Ceph领先的架构和设计理念已经深入人心。

Ceph的魅力源于其架构的前瞻性、可塑性和长期演进能力。事实上，在设计之初，Ceph被定位成一个纯粹的分布式文件系统，主要用于解决大型超级计算机之间如何通过联网的方式提供可扩展的文件存储服务。随着云计算、大数据和人工智能逐渐成为信息时代的主旋律，Ceph正不断拓展自身的触角，从取代Swift成为OpenStack首选存储后端进入公众视野，到完美适配以Amazon S3为代表的公有云接口，再到征战下一个没有硝烟的虚拟化（技术）高地——容器。时至今日，Ceph已然成为一个兼容块、文件、对象等各类经典/新兴存储协

议的超级统一存储平台。随着Ceph的加速进化，可以预见，我们将会看到越来越多的基于Ceph构建的自定义存储应用。

## 为什么写这本书

开源软件诞生的土壤决定了大部分开源软件从来就不是面向普通大众的，典型的如Linux，其无可视化界面的命令行操作方式和海量命令足以让90%的用户望而却步。Ceph作为一个出身于学院的开源作品也存在类似的缺点。此外，随着自身的不断演进和完善，Ceph已经从最初的分布式文件系统逐渐成长为一个全能的分布式统一存储平台，因此其复杂程度远远超过功能相对单一的传统存储系统。更糟的是，虽然社区有建议的编码规范，但是为了不挫伤贡献者的积极性，这些规范并未作为强制要求，因此随着贡献者数量的快速增长，Ceph代码本身也不可避免地趋于异构化。上述种种因素使得无论是使用还是开发Ceph都难度巨大，再加上语言和文化背景的差异，足以造成大量国内Ceph初级玩家难以逾越的鸿沟。

距我们创作《Ceph设计原理与实现》一书已经过去了两年。一方面，Ceph代码发生了巨大变化；另一方面，我们对Ceph的认知也有了较大提升。因此，我们两位负责研究RADOS组件的同事基于前作中的相关章节重新创作了本书。

与前作相比，本书更加专注于RADOS这个基础组件，而剥离了RBD、RGW、CephFS等具体存储应用和案例实战部分。这主要是基于以下考虑：

首先，RBD、RGW和CephFS与其承载的具体业务耦合度较高，例如RBD后续的重点工作是兼容iSCSI/FC传统块存储接口，而要彻底掌握RGW则必然要对以S3、Swift为代表的新兴对象存储协议簇有比较透彻的了解等，限于篇幅，很难单纯从Ceph的角度对这些组件做出比较完整和透彻的解读。

其次，由于时间仓促，加之不少章节均由不同的作者独立创作，因此前作中章节之间难免重复或者脱节，而本书则更加注重章节之间衔接与编排的合理性。此外，由于作者数量大幅减少，本书风格更加统一，相对而言读者可以获得更好的阅读体验。

再次，藉本次重新创作，我们进一步削弱了前作中相关章节与代码之间的耦合性，更加侧重于阐述设计理念。由于Ceph社区十分活跃，贡献者数量众多，每个版本代码都会发生翻天覆地的变化，因此，理解设计原理，以不变应万变，无疑比掌握某个特定版本的代码更为重要。

最后，需要再次强调的是，虽然本书部分章节源自《Ceph设计原理与实现》一书，但是基本上都进行了重新创作。重复录入这些章节不是简单的查漏补缺，而是进一步提炼与升华，它们是本书不可或缺的组成部分。事实上，与新增内容相比，重新创作这些章节花费了我们更多的时间与精力。

## 本书的读者对象

本书适合于对Ceph有一定了解，想更进一步参与到Ceph开源项目中来，并致力于后续为Ceph，特别是RADOS组件添砖加瓦的开发者或者高级开发者阅读。

此外，高级运维人员通过阅读本书也能够了解和掌握Ceph的核心设计理念及高级应用技巧，从而在日常运维工作中更加得心应手。

与《Ceph设计原理与实现》力求如实反映源码的实现细节不同，本书是Ceph（特别是RADOS组件）设计思想与基本理念的高度浓缩。有条件的读者可以将两本书对照阅读，相信可以有更大收获。

## 本书的主要内容

本书主要介绍Ceph的核心——RADOS。具体编排如下：

## 第1章　一生万物——RADOS导论

Ceph是集传统块、文件存储以及新兴对象存储于一身的超级分布式统一存储平台。

Ceph在架构上采用存储应用与存储服务完全分离的模式，并基于RADOS对外提供高性能和可轻松扩展的存储服务。理论上，基于RADOS及其派生的librados标准库可以开发任意类型的存储应用，典型的如Ceph当前的三大核心应用：RBD、RGW和CephFS。

作为全书的开始，本章旨在为读者建立一个RADOS的初步印象，主要介绍包括OSD、Monitor、存储池、PG、对象等在内的一众基本概念。

## 第2章　计算寻址之美与数据平衡之殇——CRUSH

CRUSH是Ceph两大核心设计之一。CRUSH良好的设计理念使其具有计算寻址、高并发和动态数据均衡、可定制的副本策略等基本特性，进而能够非常方便地实现诸如去中心化、有效抵御物理结构变化并保证性能随集群规模呈线性扩展、高可靠等高级特性，因而非常适合Ceph这类对可扩展性、性能和可靠性都有严苛要求的大型分布式存储系统。

CRUSH最大的痛点在于，在实际应用中，很容易出现由于CRUSH的先天缺陷导致PG分布不均，进而导致集群出现OSD之间数据分布失衡、集群整体空间利用率不高的问题，为此社区引入了包括reweight、weight-set、upmap、balancer在内的一系列手段加以改进。

## 第3章　集群的大脑——Monitor

Monitor是基于Paxos兼职议会算法构建的、具有分布式强一致性的小型集群，主要负责维护和传播集群表的权威副本。Monitor采用负荷分担的方式工作，因此，任何时刻、任意类型的客户端或者OSD都可以通过和集群中任意一个Monitor进行交互，以索取或者请求更新集群表。基于Paxos的分布式一致性算法可以保证所有Monitor的行为自始至终都是正确和自洽的。

### 第4章　存储的基石——OSD

对象存储起源于传统的NAS（例如NFS）和SAN存储，其基本思想是赋予底层物理存储设备（例如磁盘）一些CPU、内存资源等，使之成为一个抽象的对象存储设备（即OSD），能够独立完成一些低级别的文件系统操作（例如空间分配、磁盘I/O调度等），以实现客户端I/O操作（例如读、写）与系统调用（例如打开文件、关闭文件）之间的解耦。

与传统对象存储仅仅赋予OSD一些初级的“智能”不同，Ceph开创性地认为，这种“智能”可以被更进一步地用于执行故障恢复与数据自动平衡、提供完备的高性能本地对象存储服务等复杂任务上，从而使得基于OSD构建高可靠、高可扩展和高并发性能的大型分布式对象存储系统成为可能。

### 第5章　高性能本地对象存储引擎——BlueStore

BlueStore是默认的新一代高性能本地对象存储引擎。BlueStore在设计中充分考虑了对下一代全SSD以及全NVMe SSD闪存阵列的适配，增加了数据自校验、数据压缩等热点增值功能，面向PG提供高效、无差异和符合事务语义的本地对象存储服务。

### 第6章　移动的对象载体——PG

面向分布式的设计使得Ceph可以轻易管理拥有成百上千个节点、PB级以上存储容量的大规模集群。

通常情况下，对象大小是固定的。考虑到Ceph随机分布数据（对象）的特性，为了最大程度地实现负载均衡，不会将对象粒度设计得很大，因此即便一个普通规模的Ceph集群，也可以存储数以百万计的对象，这使得直接以对象为粒度进行资源和任务管理的代价过于昂贵。

简言之，PG是一些对象的集合。引入PG的优点在于：首先，集群中PG数量经过人工规划因而严格可控（反之，集群中对象的数量则时刻处于变化之中），这使得基于PG精确控制单个OSD乃至整个节点的资源消耗成为可能；其次，由于集群中PG数量远远小于对象数量，并且PG的数量和生命周期都相对稳定，因此以PG为单位进行数据同步或者迁移等，相较于直接以对象为单位而言，难度更小。

PG最引人注目之处在于其可以在OSD之间（根据CRUSH的实时计算结果）自由迁移，这是Ceph赖以实现自动数据恢复、自动数据平衡等高级特性的基础。

## 第7章　在线数据恢复——Recovery与Backfill

在线数据恢复是存储系统的重要研究课题之一。

与离线恢复不同，在线数据恢复的难点在于数据本身一直处于变化之中，同时在生产环境中一般都有兼顾数据可靠性和系统平稳运行的要求，因此如何合理地处理各种业务之间的冲突，恰当地分配各种业务之间的资源（例如CPU、内存、磁盘和网络带宽等），在尽可能提升数据恢复速度的同时，降低甚至完全避免数据恢复对正常业务造成干扰则显得至关重要。

按照能否依据日志进行恢复，Ceph将在线数据恢复细分为Recovery和Backfill两种方式。通常意义上，两者分别用于应对临时故障和永久故障，当然后者也常用于解决由于集群拓扑结构变化导致的数据和负载不均衡问题。

### 第8章　数据正确性与一致性的守护者——Scrub

Scrub是一种重要的辅助机制，用于守护集群数据的正确性与一致性。

实现上，Scrub主要依赖对象的有序性与信息摘要技术，前者使其可以不重复（从而高效）地遍历集群中的所有对象，后者则提供了一种快速检测数据正确性和一致性的手段。

与数据恢复类似，Scrub也分在线和离线两种方式。同样，由于数据本身一直处于变化之中，为了捕获数据错误和一致性问题，要求Scrub周期性地执行，同时为了能够完整地完成一次深度扫描，则要求Scrub基于合适的粒度、以合理的规则执行。

### 第9章　基于dmClock的分布式流控策略

dmClock是一种基于时间标签的分布式I/O调度算法。Ceph采用dmClock主要希望解决客户端业务与集群内部操作的I/O资源合理分配问题，但由于种种原因，这一部分研究进展比较缓慢。通过深入研究dmClock算法，我们在社区的基础上对其进行了改进和应用实践，增加了块设备卷粒度QoS与OSD粒度的数据恢复流量自适应控制的支持，并基于此进一步优化了整个集群的流控策略。

### 第10章　纠删码原理与实践

Ceph传统的3副本数据备份方式能够在取得高可靠性的前提下最小化客户端请求的响应时延，因而特别适合对可靠性和性能都有一定

要求的存储应用。这种目前使用最广泛的备份方式的缺点在于会大量占用额外的存储空间，因而导致集群的实际空间利用率不高。与之相反，纠删码以条带为单位，通过数学变换，将采用任意k+m备份策略所消耗的额外存储空间都成功控制在1倍以内，而代价是计算资源消耗变大和客户端请求响应时延变长，因而适合对时延不敏感的“冷数据”（例如备份数据）应用。

## 勘误与支持

《Ceph设计原理与实现》一书出版之后，我们收到了不少读者朋友的来信，指出了书中的错误和疏漏，在此对这些热心的读者朋友们一并表示感谢。

本书在前作的基础上，对相关章节进一步做了大幅修订，同时增加了大量新的章节。由于写作和认知水平有限，问题仍然在所难免，欢迎新老读者们通过以下电子邮箱对本书进行指正：

xie.xingguo@zte.com.cn

yan.jun8@zte.com.cn

## 致谢

Ceph官方社区的源代码是创作本书的原始素材，因此我们首先感谢Ceph官方社区和创始人Sage A.Weil先生。

其次，我们要感谢所在部门的主管领导谭芳，感谢他在日常工作中给予我们的关照，以及对于出版《Ceph设计原理与实现》与本书不遗余力的支持。

再次，感谢Clove团队，本书很大程度上是整个团队的智慧结晶。此外，我们要特别感谢宋维斌、韦巧苗、罗慕尧、朱尚忠、朱凯波等

几位同事，他（她）们通读了本书的初稿，提出了大量宝贵的修改意见，极大地增强了本书的专业性与可读性。

最后，感谢IT学院的闫林老师对于出版《Ceph设计原理与实现》及本书给予的鼓励和巨大帮助。

谢型果　严军

2018年8月

# 第1章
# 一生万物——RADOS导论

Ceph诞生于10年前，本来的意图是为大型超级计算机打造一个高可靠、高可扩展和高性能的分布式文件系统。在漫长的演进过程中，Ceph基于计算分布数据、完全去中心化的设计理念，经受住了时间考验，逐渐成长为大数据时代最具生命力与参考价值的开源统一存储系统。

Ceph的特点亦即其优点如下：

1）Ceph是软件定义存储。Ceph可以运行在几乎所有主流Linux发行版（例如CentOS和Ubuntu）和其他类UNIX操作系统（例如FreeBSD）之上。2016年，社区成功地将Ceph从x86架构移植到ARM架构，表明Ceph开始进军移动通信、低功耗等前沿领域。因此，Ceph的应用场景覆盖当前主流的软硬件平台。

2）Ceph是分布式存储。分布式基因使其可以轻易管理成百上千个节点、PB级及以上存储容量的大规模集群，基于计算寻址则让Ceph客户端可以直接与服务端的任意节点通信，从而避免因为存在访问热点而产生性能瓶颈。实际上，在没有网络传输限制的前提下，Ceph可以实现我们梦寐以求的、性能与集群规模呈线性扩展的优秀特性。

3）Ceph是统一存储。Ceph既支持传统的块、文件存储协议，例如SAN和NAS，也支持新兴的对象存储协议，例如S3和Swift，这使得Ceph在理论上可以满足当前一切主流的存储需求。此外，良好的架构设计使得Ceph可以轻易地拓展至需要存储的任何领域。

上述特点使得只要存在存储需求，Ceph就能找到用武之地。因此，诚如Ceph社区所言：Ceph是存储的未来！

为了尽可能地拓展“触角”，Ceph在架构上采用存储应用与存储服务完全分离的模式（即Client/Server模式），并基于RADOS（Reliable Autonomic Distributed Object Store）——一种可靠、高度自治的分布式对象存储系统，对外提供高性能和可轻松扩展的存储服务。

毋庸置疑，RADOS是Ceph的核心组件。理论上，基于RADOS及其派生的librados标准库可以开发任意类型的存储应用，典型的如Ceph的三大核心应用——RBD（Rados Block Device）、RGW（Rados GateWay）和CephFS，如图1-1所示。

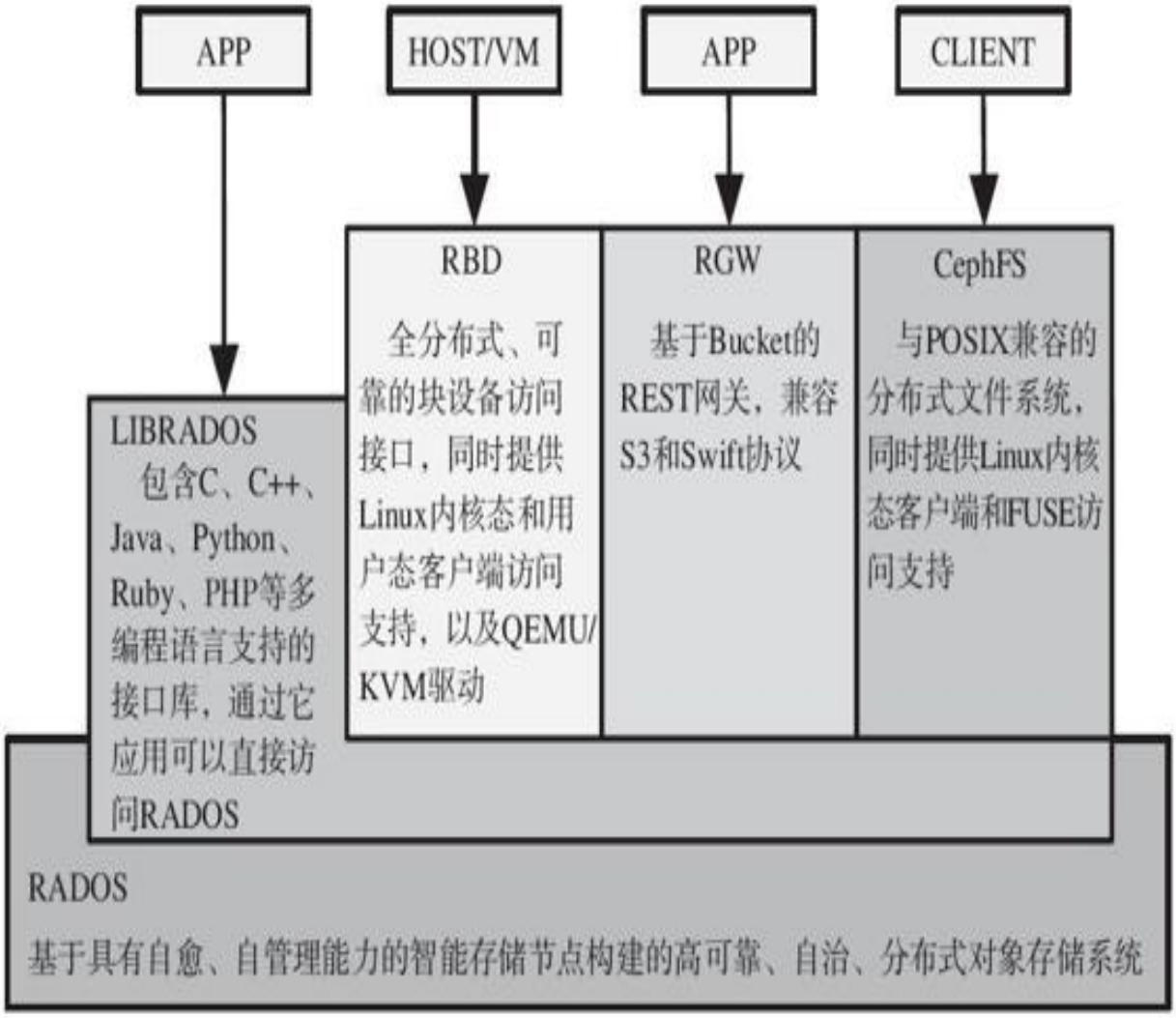



图1-1　Ceph的三大核心应用

作为全书的开始，本章简要介绍包括OSD、Monitor、存储池、PG、对象在内的基本概念，旨在为读者建立一个对RADOS的初步印象。但是作为一个分布式存储系统的核心组件，RADOS的庞大和复杂在所难免，因此本章的介绍不免存在挂一漏万之嫌。好在序幕刚刚开启，如果存在某些当前无法理解的细节，读者大可不必灰心，我们将在后续章节逐一解答。

## 1.1 RADOS概述

当存储容量达到PB级别或者更高规模时，对应的存储系统必然一直处于变化之中：随着存储需求的增长，会不断有新设备被加入进来；老设备因为故障而不断地被淘汰；任何时候都有大量数据写入、更新或者删除；持续不断的数据恢复，或者出于负载均衡而自动触发的数据迁移行为等。如果再考虑用户不断提升的数据可靠性和近乎无止境的性能诉求，那么任何基于中心控制器和查表法设计的传统存储都难免力不从心。

Ceph基于RADOS，可提供可靠、高性能和全分布式的存储服务。基于计算实时分布数据，允许自定义故障域，再加上任意类型的应用程序数据都被抽象为对象这个同时具备安全和强一致性语义的抽象数据类型，从而，RADOS可轻松地在大规模异构存储集群中实现动态数据与负载均衡。

简言之，一个RADOS集群由大量OSD（Object Store Device，对象存储设备）和少数几个Monitor组成。

OSD是个抽象概念，一般对应一个本地块设备（如一块磁盘或者一个RAID组等），在其工作周期内会占用一些CPU、内存、网络等物理资源，并依赖某个具体的本地对象存储引擎，来访问位于块设备上的数据。

基于高可靠设计的Monitor团体（quorum，本质上也是一个集群）则负责维护和分发集群的关键元数据，同时也是客户端与RADOS集群建立连接的桥梁——客户端通过咨询Monitor获得集群的关键元数据之后，就可以通过约定的方式（例如RBD、RGW、CephFS等）来访问RADOS集群，如图1-2所示。

为了去中心化、免除单点故障，RADOS使用一张紧凑的集群表对集群拓扑结构和数据映射规则进行描述。任何时刻，任何持有该表的合法客户端都可以独立地与位于任意位置的OSD直接通信。当集群拓扑结构发生变化时，RADOS确保这些变化能够及时地被Monitor捕获，并通过集群表高效传递至所有受影响的OSD和客户端，以保证对外提供不中断的业务访问。由于数据复制、故障检测和数据恢复都由每个OSD自动进行，因此即便存储容量上升至PB级别或者以上，系统也不会存在明显的调度和处理瓶颈。

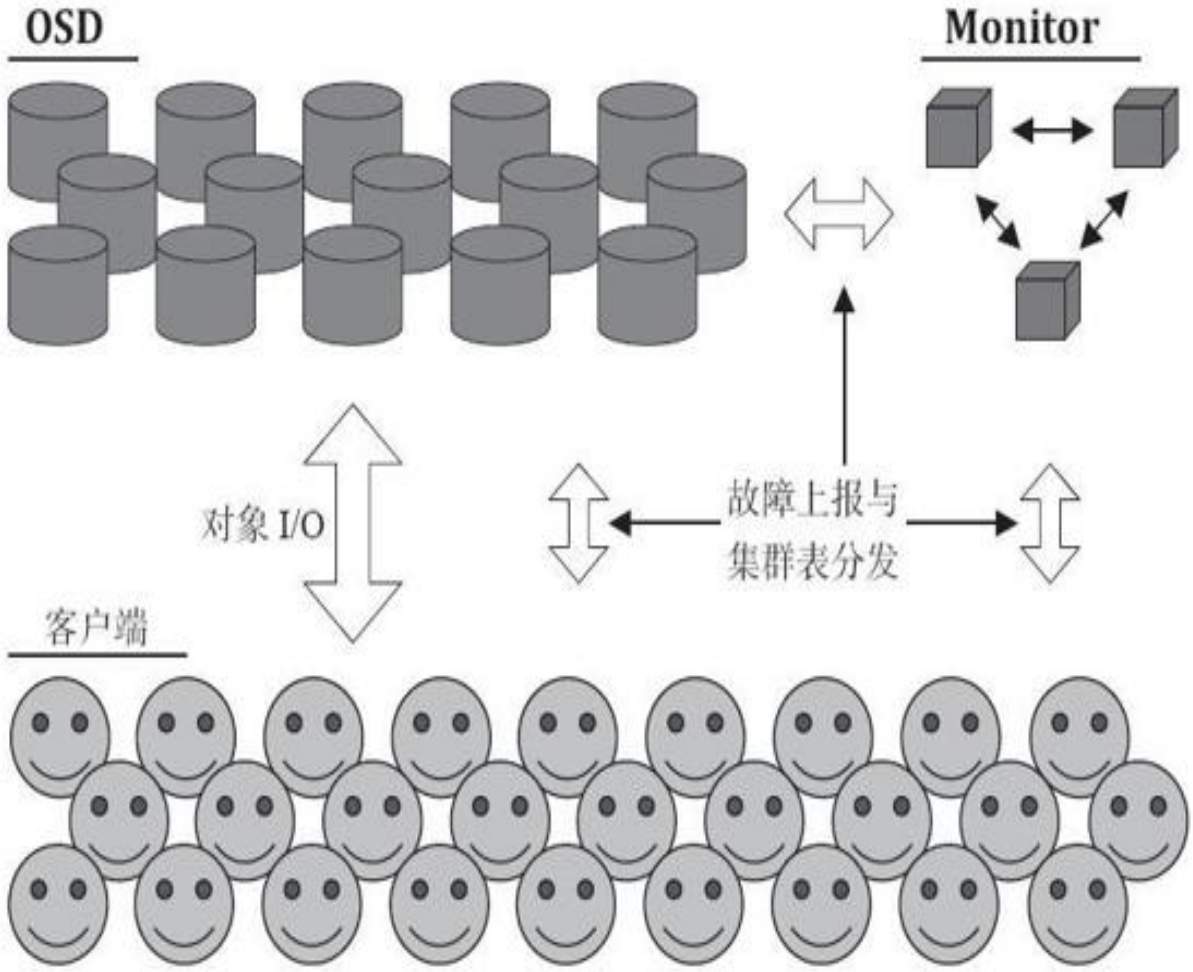



图1-2　一个典型的RADOS集群

RADOS取得高可扩展性的关键在于彻底抛弃了传统存储系统中诸如中心控制器、网关等概念，另辟蹊径地以基于可扩展哈希的受控副本分布算法——CRUSH（Controlled Replication Under Scalable Hashing）取而代之，作为衔接客户端和OSD的桥梁，使得客户端可以直接与OSD通信，从而得以彻底免除需要查表的烦琐操作。进一步地，由于CRUSH包含了获取集群当前数据分布形态所需的全部信息，所以OSD之间通过交互即可智能地完成诸如故障、扩容等引起的数据重新分布，而无须中心控制器进行指导和干预。集群表屏蔽了实际集群可能呈现的纷繁芜杂的细节，使得客户端可以将整个RADOS集群当作一个单一的“对象”来处理。

当然，在生产环境中，由于集群本身可能具有复杂的拓扑结构，以及随着时间的推移，不可避免地趋于异构化，为了最大化资源利用率，我们还必须对集群资源合理地分割和重组。

## 1.2 存储池与PG

为了实现存储资源按需配置，RADOS对集群中的OSD进行池化管理。资源池（对存储系统而言，具体则是存储池）实际上是个虚拟概念，表示一组约束条件，例如可以针对存储池设计一组CRUSH规则，限制其只能使用某些规格相同的OSD，或者尽量将所有数据副本分布在物理上隔离的、不同的故障域；也可以针对不同用途的存储池指定不同的副本策略，例如若存储池承载的存储应用对时延敏感，则采用多副本备份策略；反之，如果存储的是一些对时延不敏感的数据（例如备份数据），为提升空间利用率则采用纠删码备份策略；甚至可以为每个存储池分别指定独立的Scrub、压缩、校验策略等。

为了实现不同存储池之间的策略隔离，RADOS并不是将任何应用程序数据一步到位地写入OSD的本地存储设备，而是引入了一个中间结构，称为PG（Placement Group），执行两次映射，如图1-3所示。

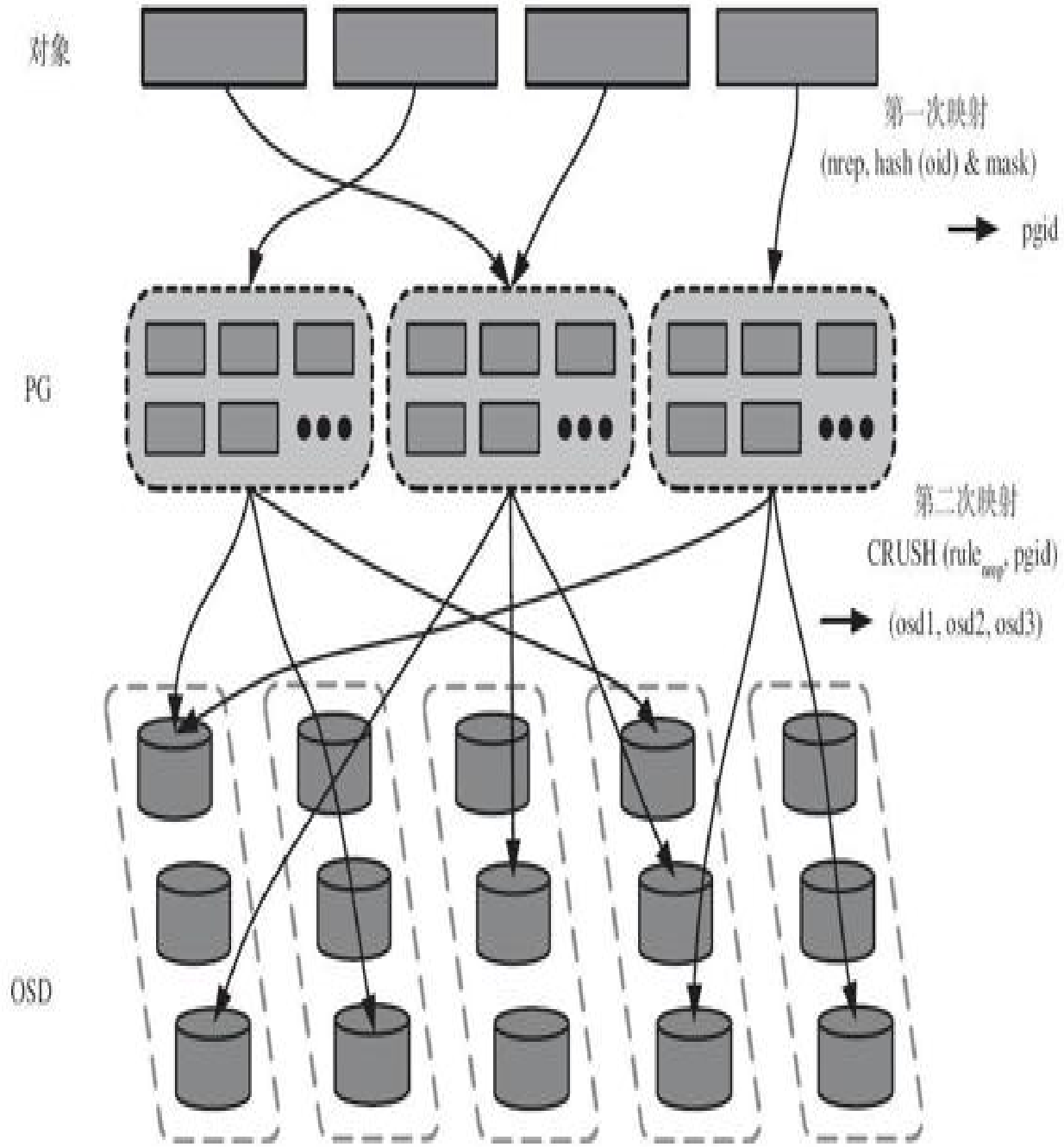



图1-3　应用程序数据通过两次映射到达OSD的本地存储设备

第一次映射是静态的，负责将任意类型的客户端数据按照固定大小进行切割、编号后，作为伪随机哈希函数输入，均匀映射至每个PG，以实现负载均衡策略；第二次映射实现PG到OSD的映射，仍然采用伪随机哈希函数（以保证PG在OSD之间分布的均匀性），但是其输

入除了全局唯一的PG身份标识之外，还引入了集群拓扑，并且使用CRUSH规则对映射过程进行调整，以帮助PG在不同OSD之间灵活迁移，进而实现数据可靠性、自动平衡等高级特性。最终，存储池以PG作为基本单位进行管理。

为了维持扁平寻址空间，实际上要求PG拥有一个全集群唯一的身份标识——PGID。由于集群所有存储池都由Monitor统一管理，所以Monitor可以为每个存储池分配一个集群内唯一的存储池标识。基于此，我们只需要为存储池中的每个PG再分配一个存储池内唯一的编号即可。假定某个存储池的标识为1，创建此存储池时指定了256个PG，那么容易理解对应的PGID可以写成1.0，1.1，…，1.255这样的形式。

## 1.3 对象演进与排序

与Linux“一切皆文件”的设计理念类似，Ceph将任意类型的客户端数据都抽象为对象这个小巧而精致的概念。

文件伴随文件系统一同诞生，是计算机科学中最基本和最经典的概念之一。为了增强（人机之间的）可交互性，一般而言文件对外支持使用基于UTF-8编码方式的字符串进行命名，作为其唯一身份标志。这意味着在最初的文件系统设计中，所有文件都共享一个扁平的全局命名空间（namespace）。

信息技术的飞速发展使得数据（信息）呈井喷式增长，因此，与传统文件系统不同，现代文件系统普遍采用面向管理海量数据（文件）的设计——例如XFS，这是一个64位文件系统，64位意味着XFS理论上可以存储和管理多达$2^{64}$个文件。而被誉为最后一个文件系统的ZFS则更进一步地采用面向128位寻址空间的设计（注：这表明单个ZFS文件系统可以管理的存储空间为$2^{128}$，但是因为ZFS仍然采用64位的inode设计，所以单个ZFS文件系统能够存储的最大文件数量仍然与XFS相同。）。正如ZFS的创造者所言：理论上，单个ZFS所能管理的存储容量“即便穷尽这个地球上的每粒沙子来制造存储设备也无法企及”。

出于查找效率考虑，面向管理海量文件而生的现代文件系统不可能继续采取全局唯一的扁平命名空间，而必须采用更高效的组织方式。参考人类社会族谱、各类社会组织机构等，一个自然的想法是转而使用树这种非线性数据结构对文件系统中所有文件进行描述：树中每个叶子节点表示一个文件，每个中间节点则起到了隔离和保护其辖下所有文件、特别是每个文件名作用域的作用，因此这些中间节点又被形象地称为文件夹（或者目录）。这样，即便使用数据结构中最常见的平衡二叉树来管理单个64位文件系统中的所有文件（注：指理论上限，即$2^{64}$。），树的高度（或者层级）也不超过64，也就是查找任何文件都可以经过至多64次比较得到。

与文件类似，Ceph也使用字符串对对象进行标识，使用命名空间对每个对象归属的作用域进行限制和隔离。除此之外，因为对象（必须通过PG间接地）归属于某个存储池，因此对象还必须记忆其归属的存储池标识。

作为文件系统中最经典的数据备份机制，快照和克隆对所有存储系统而言几乎都是必备功能。在将一切客户端数据对象化之后，很容易理解Ceph的快照和克隆功能，其实现粒度也应当是对象级的。因此，为了区分原始对象和由其产生的一众克隆对象，必须通过向对象中添加全局唯一的快照标识（snap-id）加以解决。进一步，在宣布支持纠删码这种数据备份机制后，Ceph的数据强一致性设计必然要求其支持对象粒度的数据回滚功能，为此还必须能够区分某个纠删码对象的当前版本和由其衍生的一众历史版本。类似地，这通过向对象中添加分片标识（shard-id）和对象版本号（generation）加以解决。

至此，我们已经得到了基于对象名、命名空间、对象归属的存储池标识、快照标识、分片标识和版本号区分集群中各种对象的方法，基于这些特征值构造的数据结构也被称为对象标识（object-id），其在集群内唯一定义一个Ceph对象。

事实上，无论是Ceph引以为傲的自动数据恢复和平衡功能，还是用于守护数据正确性与一致性的Scrub机制，都无不隐式地依赖于“可以通过某种手段不重复地遍历PG中所有对象”这个前提，这实际上要求能够对PG中的每个对象进行严格排序。一个比较直观的方法是简单地将对象标识中所有特征值按照一定顺序拼接起来，形成一个囊括对象所有特征信息的字符串。这样，直接通过字符串比较（容易理解，每个这种类型的字符串都唯一对应一个对象），我们即可实现对象排序。然而不幸的是，一些典型应用，例如RBD，为了保证对象的唯一性，一般都会倾向于使用较长的对象名（例如rbd_data.75b86e238e1f29.0000000000000050），因此如果直接采用上面这种方式，其效率实际上是十分低下的。

一个改进的方向是与存储池标识类似，要求每个对象也直接采用集群内唯一的数字标识。考虑到Ceph本身是个分布式存储系统，这个数字标识只能由客户端在创建每个对象时统一向Monitor申请。显然，这样做效率只会更低。那么，是否还有其他代价较小的方法能产生这个数字标识呢？答案是哈希。

哈希是一种在密码学中被广泛使用的非对称数学变换手段。大部分情况下，针对不同输入，哈希都能够得到不同输出，而使用不同输入得到相同输出的现象称为哈希冲突。产生哈希冲突的概率一是取决于算法本身，二是取决于输出长度。考虑到目前广泛使用的几种哈希算法都经过大量工程实践验证，其算法本身的质量毋庸置疑，因此采用典型哈希算法（例如MD5、SHA）产生冲突的概率主要受输出长度影响。举例来说，如果输出长度为16位，则原始输入经过哈希后至多产生65536种不同的结果（输出）；如果输出长度增加到32位，则将产生超过40亿种不同的结果。显然，在保证算法不变的前提下，增加输出长度可以不同程度地降低产生冲突的概率。当然，随着输出长度的增加，所需的存储空间也将相应增加，加之冲突的发生理论上不可避免，因此一般而言输出长度也不是越大越好。

考虑到Ceph当前的实现中，我们总是以PG为单位进行数据迁移、Scrub等，因此实际上仅需要能够针对单个PG中的全部对象进行排序即可。以标称容量为1TB的磁盘为例，按照每个OSD承载100个PG并且每个对象大小固定为4MB进行估算，平均每个PG存储的对象数量约为2621个。因此，在相当长的一段时间内（注：参考摩尔定律，假定磁盘容量每年翻一番，按照前面的估算，$2621\times2^9=1341952$，即要到9年之后，单个PG能够存储的对象数量才能达到百万级别。），我们认为每个PG能够存储的最大对象数量大约在百万至千万级别。进一步地，综合考虑内存消耗（这个哈希结果需要作为对象标识的一部分常驻内存）和比较效率（我们引入这个数值的初衷就是为了对对象排序进行加速），采用32位的哈希输出是一个比较合理的选择。

确定了哈希输出长度后，反过来我们需要决策哪些对象特征值适合充当哈希输入。最差的做法当然是直接使用对象的全部特征值。考虑到快照、纠删码数据回滚并不是常见操作，因此与之相关的快照标识、分片标识、版本号这类特征值在大部分情况下都是无效值，可以排除在外。进一步地，由于PG不能在存储池之间共享，即某个PG中的所有对象都只能归属于同一个存储池，所以在计算哈希时，存储池标识也是无效输入，同样需要予以剔除。

综上，我们认为使用命名空间加上对象名作为哈希输入是合适的。包含此32位哈希值之后的对象标识如表1-1所示。

表1-1　对象标识

| 特征值 | 含　义 |
| --- | --- |
| 对象名 | 任意长度的字符串，客户端必须保证基于命名空间＋对象名可以唯一定义（索引）存储池中的一个对象 |
| 键（可选） | 作用与对象名类似，目前暂未使用 |
| 命名空间（可选） | 任意长度的字符串，由客户端指定，客户端必须保证基于命名空间＋对象名可以唯一定义（索引）存储池中的一个对象 |
| 存储池标识 | 对象归属的存储池标识 |
| 哈希 | 使用对象名＋命名空间作为输入，通过字符哈希函数（默认为Robert Jenkin's hash function[注]）计算得到的32位哈希值 |
| 快照标识 | 从0开始编号，为64位无符号整型，以下3个值为预留：<br>• CEPH_SNAPDIR(-1)：标识隐藏的.snap目录对象，Luminous版本之后已经废弃<br>• CEPH_NOSNAP(-2)：表示原始对象（非克隆对象）<br>• CEPH_MAXSNAP(-3)：最大快照标识 |

（续）

| 特征值 | 含义 |
| --- | --- |
| 分片标识 | 仅纠删码存储池有效 |
| 版本号 | 对象版本号，仅纠删码存储池有效 |

注：http://burtleburtle.net/bob/hash/evahash.html

基于对象的特征值可以定义对象排序所依赖的比较算法如下：

·比较两个对象的分片标识

·比较两个对象的存储池标识

·比较两个对象的32位哈希值

·比较两个对象的命名空间

·比较两个对象的键

·比较两个对象的对象名

·比较两个对象的快照标识

·比较两个对象的版本号

·判定两个对象相等

进一步地，基于上述排序算法可以对PG中的所有对象执行快速排序，这是实现Backfill、Scrub等复杂功能的理论基础。

## 1.4 stable_mod与客户端寻址

Ceph通过C/S模式实现外部应用程序与RADOS集群之间的交互。任何时候，应用程序通过客户端访问集群时，首先由其客户端负责生成一个字符串形式的对象名，然后基于对象名计算得出一个32位哈希值。针对此哈希值，通过简单的数学运算，例如对存储池的PG数（pg_num）取模，可以得到一个PG在存储池内部的编号，加上对象本身已经记录了其归属存储池的唯一标识，最终可以找到负责承载该对象的PG。

假定标识为1的存储池中的某个对象，经过计算后32位哈希值为0x4979FA12，并且该存储池的pg_num为256，则由：

```
0x4979FA12 mod 256 = 18
```

我们知道此对象由存储池内编号为18的PG负责承载，并且其完整的PGID为1.18。

一般而言，将某个对象映射至PG时，我们并不会使用全部的32位哈希值，因此会出现不同对象被映射至同一个PG的现象。我们很容易验证该存储池内哈希值如下的其他对象，通过模运算同样会被映射至PGID为1.18的PG之上。

```
0x4979FB12 mod 256 = 18
0x4979FC12 mod 256 = 18
0x4979FD12 mod 256 = 18
…
```

可见，针对上面这个例子，我们仅仅使用了这个“全精度”32位哈希值的后8位。因此，如果pg_num可以写成$2^n$的形式（例如这里256可以写成$2^8$，即是2的整数次幂），则每个对象执行PG映射时，其32位哈希值中仅有低n位是有意义的。进一步地，我们很容易验证此时归属于同一个PG的对象，其32位哈希值中低n位都是相同的。基于此，我们将$2^n$-1称为pg_num的掩码，其中n为掩码的位数。

相反，如果pg_num不是2的整数次幂，即无法写成$2^n$的形式，仍然假定此时其最高位为n（从1开始计数），则此时普通的取模操作无法保证“归属于某个PG的所有对象的低n位都相同”这个特性。例如，假定pg_num为12，此时n=4，容易验证如下输入经过模运算后结果一致，但是它们只有低2位是相同的。

```
0x00 mod 12 = 0
0x0C mod 12 = 0
0x18 mod 12 = 0
0x24 mod 12 = 0
…
```

因此需要针对普通的取模操作加以改进，以保证针对不同输入，当计算结果相等时，维持“输入具有尽可能多的、相同的低位”这个相对有规律的特性。

一种改进的方案是使用掩码来替代取模操作。例如仍然假定pg_num对应的最高位为n，则其掩码为$2^n$-1，需要将某个对象映射至PG时，直接执行hash &($2^n$-1)即可。但是这个改进方案仍然存在一个潜在问题：如果pg_num不是2的整数次幂，那么直接采用这种方式进行映射

会产生空穴，即将某些对象映射到一些实际上并不存在的PG上，如图1-4所示。这里pg_num为12，n=4，可见执行hash &($2^4$-1)会产生0～15共计16种不同的结果。但是实际上编号为12～15的PG并不存在。


图1-4　使用掩码方式进行对象至PG映射

因此需要进一步对上述改进方案进行修正。由n为pg_num的最高位，必然有：

$$pg\_num \geq 2^{n-1}$$

即[0，$2^{n-1}$]内的PG必然都是存在的。于是可以通过hash &($2^{n-1}$-1)将那些实际上并不存在的PG重新映射到[0，$2^{n-1}$]区间，如图1-5所示。修正后的效果等同于将这些空穴通过平移的方式重定向到前半个对称区间。

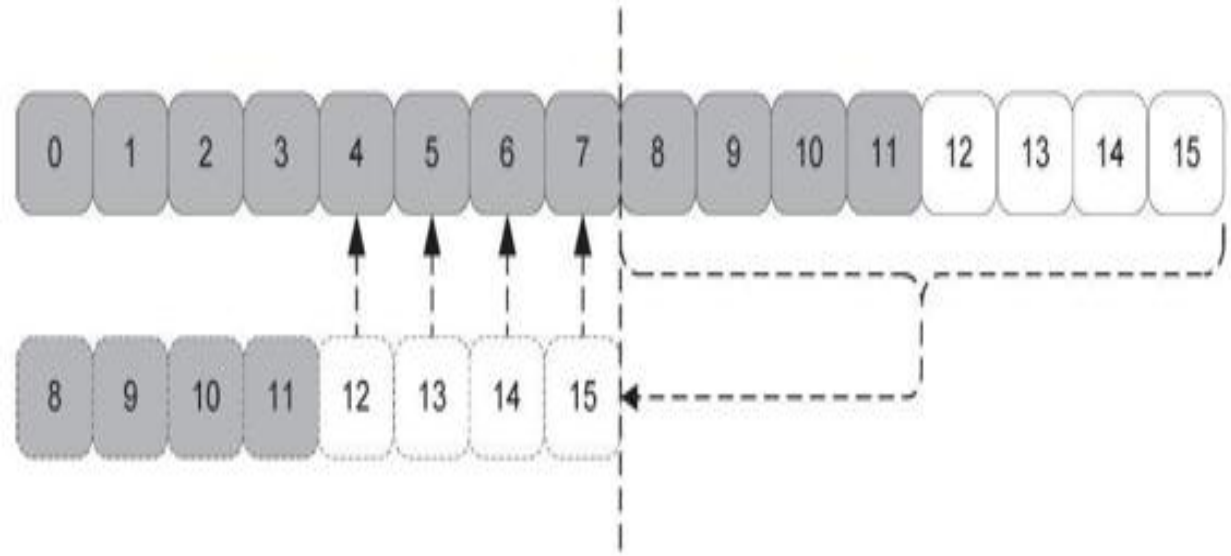


图1-5　使用修正后的掩码方式对图1-4中的映射结果进行校正

这样，退而求其次，如果pg_num不是2的整数次幂，我们只能保证相对每个PG而言，映射至该PG的所有对象，其哈希值低n-1位都是相同的。

改进后的映射方法称为stable_mod，其逻辑如下：

```
if ((hash & (2^n - 1)) < pg_num)
    return (hash & (2^n - 1));
else
    return (hash & (2^n-1-1));
```

仍然以pg_num等于12为例，可以验证：如果采用stable_mod方式，在保证结果相等的前提下，如下输入的低n-1=3位都是相同的。

```
0x05 stable_mod 12 = 5
0x0D stable_mod 12 = 5
0x15 stable_mod 12 = 5
0x1D stable_mod 12 = 5
…
```

综上，无论pg_num是否为2的整数次幂，采用stable_mod都可以产生一个相对有规律的对象到PG的映射结果，这是PG分裂的一个重要理论基础。

## 1.5 PG分裂与集群扩容

为了节省成本，初期规划的Ceph集群容量一般而言都会偏保守，无法应对随着时间推移而呈爆炸式增长的数据存储需求，因此集群扩容成为一种常见操作。

对Ceph而言，需要关注的一个问题是扩容前后负载在所有OSD之间的重新均衡，即如何能让新加入的OSD立即参与负荷分担，从而实现集群性能与规模呈线性扩展这一优雅特性。通常情况下，这是通过PG自动迁移和重平衡来实现的。然而遗憾的是，存储池中的PG数量并不会随着集群规模增大而自动增加，这在某些场景下往往会导致潜在的性能瓶颈。为此，Ceph提供了一种手动增加存储池中PG数量的机制。

当存储池中的PG数增加后，新的PG会被随机、均匀地映射至归属于该存储池的所有OSD之上。又由图1-3可知，执行对象至PG的映射时，此时作为stable_mod输入之一的pg_num已经发生了变化，所以会出现某些对象也被随之映射至新PG的现象。因而需要抢在客户端访问之前，将这部分对象转移至新创建的PG之中。

显然，为了避免造成长时间的业务停顿，我们应该尽可能减少对象移动。为此我们需要先研究对象在PG之间分布的特点，以便以最高效的方式产生新的PG和转移对象。

假定某个存储池老的pg_num为$2^4$，新的pg_num为$2^6$，以存储池中某个老PG（假定其PGID=Y.X，其中X=0b$X_3X_2X_1X_0$）为例，容易验证其中所有对象的哈希值都可以分成如图1-6所示的4种类型（按前面的分析，分裂前后，对象哈希值中只有低6位有效，图中只展示这6位）。

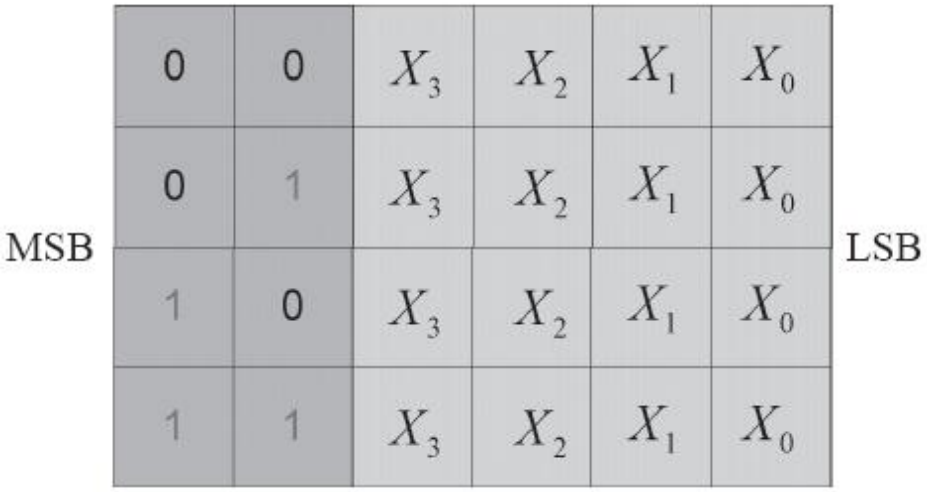



图1-6　4种类型对象哈希值

依次针对这4种类型的对象PG（Y.X）使用新的pg_num（$2^4$->$2^6$）再次执行stable_mod，结果如表1-2所示。

表1-2　使用新的pg_num重新执行stable_mod的结果

| 对象哈希值（32位，二进制） | 执行 stable_mod 结果（X=0b $X_5X_4X_3X_2X_1X_0$） |
|---|---|
| 0b???? ???? ???? ???? ???? ???? ??00 $X_5X_4X_3X_2X_1X_0$ | X |
| 0b???? ???? ???? ???? ???? ???? ??01 $X_5X_4X_3X_2X_1X_0$ | 1 x 16 + X |
| 0b???? ???? ???? ???? ???? ???? ??10 $X_5X_4X_3X_2X_1X_0$ | 2 x 16 + X |
| 0b???? ???? ???? ???? ???? ???? ??11 $X_5X_4X_3X_2X_1X_0$ | 3 x 16 + X |

可见，由于存储池的pg_num发生了变化，仅有第1种类型（$X_5X_4$=00）的对象使用stable_mod方法仍然能够映射至老的PG之上，其他3种类型的对象则并无实际PG对应。为此，我们需要创建3个新的PG来分别转移部分来自老PG中的对象。参考表1-2，容易验证这3个新的PG在存储池中的编号可以通过(m×16)+X(m=1，2，3)得到，同时由于（概率上）每种类型的对象数量相等，因此完成对象转移之后每个PG承载的数据仍然可以保持均衡。

针对所有老的PG重复上述过程，最终可以将存储池中的PG数量调整为原来的4倍。又因为在调整PG数量的过程中，我们总是基于老PG（称为祖先PG）产生新的孩子PG，并且新的孩子PG中最初的对象全部来源于老PG，所以这个过程被形象地称为PG分裂。

在FileStore的实现中，由于PG对应一个文件目录，其下的对象全部使用文件保存，所以出于索引效率和PG分裂考虑，FileStore对目录下的文件进行分层管理。采用stable_mod执行对象至PG的映射后，由于归属于同一个PG的所有对象，总是可以保证它们的哈希值至少低n-1位是相同的，所以一个自然的想法是使用对象哈希值逆序之后作为目录分层的依据。例如，假定某个对象的32位哈希值为0xA4CEE0D2，则其可能的一个目录层次为./2/D/0/E/E/C/4/A/0xA4CEE0D2，这样我们最终可以得到一个符合常规的、树形的目录结构。

仍然假定某个PG归属的存储池分裂之前共有256个PG（此时n=8，对应的掩码为$2^8$-1=255），PG对应的PGID为Y.D2，则某个时刻其一个可能的目录结构如下：

---

```
.
└── 2
    └── D
        ├── 0
        │   ├── 0x324140D2
        │   └── 0x78AF30D2
        ├── 1
```

```
│   └── 0x9898A1D2
├── 2
│   ├── 0x1578A2D2
│   └── 0x787AE2D2
├── 3
│   ├── 0x789713D2
│   └── 0x8976E3D2
├── 4
│   └── 0x123174D2
├── 5
│   └── 0xAA3CE5D2
├── 6
│   ├── 0x6873F6D2
│   └── 0xFABE36D2
├── 7
│   └── 0xA675F7D2
├── 8
│   ├── 0x9786A8D2
│   └── 0x990028D2
├── 9
│   └── 0x787329D2
├── A
│   └── 0x67812AD2
├── B
│   └── 0x7760FBD2
├── C
│   └── 0x798ADCD2
├── D
│   └── 0x787ABDD2
├── E
│   ├── 0x014ACED2
│   └── 0x123EEED2
└── F
    ├── 0x018BCFD2
    └── 0xBCC34FD2
```

---

如果分裂为4096个PG（此时n=12，对应的掩码为$2^{12}$-1=4095），则原来每个PG对应分裂成16个PG。仍以PGID为Y.D2的PG为例，通过简单计算可以得到这15个新增PG的PGID分别为：

---

```
Y.1D2
Y.2D2
…
```

```
Y.ED2
Y.FD2
```

---

可见这些新的PG对应的对象分别存储于老PG的如下目录之下。

---

```
./2/D/1/
./2/D/2/
…
./2/D/E/
./2/D/F/
```

---

此时可以不用移动对象（即文件），而是直接修改文件夹的归属，即可完成对象在新老PG之间的迁移。

引入PG分裂机制之后，如果仍然直接使用PGID作为CRUSH输入，据此计算新增孩子PG在OSD之间的映射结果，由于此时每个PG的PGID都不相同，必然触发大量新增孩子PG在OSD之间迁移。考虑到分裂之前PG在OSD之间的分布已经趋于均衡，更好的办法是让同一个祖先诞生的所有孩子PG与其保持相同的副本分布，这样当分裂完成之后，整个集群的PG分布仍然是均衡的。为此，每个存储池除了记录当前的PG数量之外，为了应对PG分裂，还需要额外记录分裂之前祖先PG的数量，后者称为PGP（Placement Group Placement）数量（pgp_num）。

最终，利用CRUSH执行PG映射时，我们不是直接使用PGID，而是转而使用其在存储池内的唯一编号先针对pgp_num执行stable_mod，再与存储池标识一起哈希之后作为CRUSH的特征输入，即可保证每个孩子PG与其祖先产生相同的CRUSH计算结果（因为此时祖先和孩子PG产生的CRUSH特征输入都相同），进而保证两者产生相同的副本分布。

Luminous将默认的本地对象存储引擎切换为BlueStore，此时同一个OSD下所有对象都共享一个扁平寻址空间，所以PG分裂时，甚至不存在上述对象在（新老）文件夹之间转移的过程，因而更加高效。

## 1.6 总结和展望

作为Ceph的服务端，RADOS集群负责提供可信赖、可扩展和高性能的分布式对象存储服务。串行化和安全性解耦的设计理念使得RADOS可在客户端对故障零感知的情况下仍然提供强一致性语义。

基于分布式高可靠机制构建的小型Monitor集群负责维护集群表的权威副本（master copy）。进一步地，通过集群表传播CRUSH，RADOS保证所有OSD和应用程序客户端都能清楚地了解集群当前的数据分布形态。一方面，这使得RADOS免受类似传统存储需要通过查表才能准确定位某个具体对象的困扰；另一方面，也使得RADOS能够在线处理诸如数据复制、一致性管理、故障恢复等复杂任务，同时仍然对外提供强一致性的读写服务。上述一切，再加上精心设计的、可扩展的故障检测机制和“涟漪式”集群表传播机制，使得在生产环境中构建超大规模的RADOS集群成为可能。

如前所述，当集群存储规模上升至PB级别或者以上，其拓扑结构必然是异构的。同时，由于集群中存在成千上万个本地存储设备，故障必然成为常态。以PG为粒度、高度并行的数据恢复机制使得RADOS无论是从短暂性的掉电，还是永久性的硬件损坏之中恢复都异常迅速和高效，从而大大降低了数据丢失风险。

“路漫漫其修远兮，吾将上下而求索。”新的征程已在前方展开，接下来，让我们首先接触一下Ceph两大核心设计之一的CRUSH，领略计算寻址之美。

第2章

# 计算寻址之美与数据平衡之殇——CRUSH

大部分存储系统将数据写入后端存储设备之后，很少会在设备之间再次移动数据。这就存在一个潜在问题：即使一个数据分布已经趋于完美均衡的系统，随着时间推移，新的空闲设备不断加入，老的故障设备不断退出，数据也会重新变得不均衡。这种情况在大型分布式存储系统中尤为常见。

一种可行的解决方案是将数据以足够小的粒度打散，然后完全随机地分布至所有存储设备。这样，从概率上而言，如果系统运行的时间足够长，所有设备的空间使用率将会趋于均衡。当新设备加入后，数据会随机地从不同的老设备迁移过来；当老设备因为故障退出后，其原有数据会随机迁出至其他正常设备。整个系统一直处于动态平衡之中，从而能够适应任何类型的负载和拓扑结构变化。进一步地，由于任何数据（可以是文件、块设备等）都被打散成多个碎片，然后写入不同的底层存储设备，从而使得在大型分布式存储系统中获得尽可能高的I/O并发和汇聚带宽成为可能。

一般而言，使用哈希函数可以达到上述目的。但是在实际应用中还需要解决两个问题：一是如果系统中存储设备数量发生变化，如何最小化数据迁移量，从而使得系统尽快恢复平衡；二是在大型（PB级及以上）分布式存储系统中，数据一般包含多个备份，如何合理地分布这些备份，从而尽可能地使得数据具有较高的可靠性。因此，需要对普通哈希函数加以扩展，使之能够解决上述问题，Ceph称之为CRUSH。

顾名思义，CRUSH是一种基于哈希的数据分布算法。以数据唯一标识符、集群当前的拓扑结构以及数据备份策略作为CRUSH输入，可以随时随地通过计算获取数据所在的底层存储设备（例如磁盘）位置并直接与其通信，从而避免查表操作，实现去中心化和高度并发。CRUSH同时支持多种数据备份策略，例如镜像、RAID及其衍生的纠删码等，并受控地将数据的多个备份映射到集群不同故障域中的底层存储设备之上，从而保证数据可靠性。上述特点使得CRUSH特别适用于对可扩展性、性能以及可靠性都有极高要求的大型分布式存储系统。

本章按照如下思路组织：首先，介绍CRUSH需要解决的问题，由此引出CRUSH最重要的基本选择算法——straw及其改进版本straw2；其次，完成基本算法分析后，介绍如何将其进一步拓展至具有复杂拓扑结构的真实集群，为了实现上述目标，首先介绍集群拓扑结构和数据分布规则的具体描述形式，然后以此为基础介绍CRUSH的完整实现；再次，生产环境中集群拓扑结构千变万化，CRUSH配置相对复杂，我们结合一些实际案例，分析如何针对CRUSH进行深度定制，以满足形式各异的数据分布需求；最后，由于CRUSH在执行多副本映射时存在一些缺陷，如果集群规模较小或者异构化，容易出现数据分布不均衡的问题，为此，我们针对一些可行的解决方案进行了探讨。

## 2.1 抽签算法

Ceph在设计之初被定位成用于管理大型分级存储网络的分布式文件系统。网络中的不同层级具有不同的故障容忍程度，因此也称为故障域。图2-1展示了一个具有3个层级（机架、主机、磁盘）的Ceph集群。

图2-1　具有3个层级的Ceph集群

在图2-1中，单个主机包含多个磁盘，每个机架包含多个主机，并采用独立的供电和网络交换系统，从而可以将整个集群以机架为单位划分为若干故障域。为了实现高可靠性，实际上要求数据的多个副本分布在不同机架的主机磁盘之上。因此，CRUSH首先应该是一种基于层级的深度优先遍历算法。此外，上述层级结构中，每个层级的结构特征也存在差异，一般而言，越处于顶层的其结构变化的可能性越小，反之越处于底层的则其结构变化越频繁。例如，大多数情况下一个Ceph集群自始至终只对应一个数据中心，但是主机或者磁盘数量随时间流逝则可能一直处于变化之中。因此，从这个角度而言，CRUSH还应该允许针对不同的层级按照其特点设置不同的选择算法，从而实现全局和动态最优。

在CRUSH的最初实现中，Sage一共设计了4种不同的基本选择算法，这些算法是实现其他更复杂算法的基础，它们各自的优缺点如表2-1所示。

表2-1　CRUSH基本选择算法对比

"添加元素"和"删除元素"通过对比同一场景每种算法产生的数据迁移量从而评判其好坏

| 算法 / 对比项 | unique | list | tree | straw |
|---|---|---|---|---|
| 时间复杂度 | O(1) | O(*N*) | O(log(*N*)) | O(*N*) |
| 添加元素 | 差 | 最好 | 好 | 最好 |
| 删除元素 | 差 | 差 | 好 | 最好 |

由表2-1可见，unique算法执行效率最高，但是抵御结构变化的能力最差；straw算法执行效率较低，但是抵御结构变化的能力最好；list和tree算法执行效率和抵御结构变化的能力介于unique与straw之间。如

果综合考虑呈爆炸式增长的存储空间需求（导致需要添加元素）、在大型分布式存储系统中某些部件故障是常态（导致需要删除元素），以及愈发严苛的数据可靠性需求（导致需要将数据副本存储在更高级别的故障域中，例如不同的数据中心），那么针对任何层级采用straw算法都是一个不错的选择。事实上，这也是CRUSH算法的现状，在大多数将Ceph用于生产环境的案例中，除了straw算法之外，其他3种算法基本上形同虚设，因此我们将重点放在分析straw算法上。

顾名思义，straw算法将所有元素比作吸管，针对指定输入，为每个元素随机地计算一个长度，最后从中选择长度最长的那个元素（吸管）作为输出。这个过程被形象地称为抽签（draw），元素的长度称为签长。

显然straw算法的关键在于如何计算签长。理论上，如果所有元素构成完全一致，那么只需要将指定输入和元素自身唯一编号作为哈希输入即可计算出对应元素的签长。因此，如果样本容量足够大，那么最终所有元素被选择的概率都是相等的，从而保证数据在不同元素之间均匀分布。然而实际上，前期规划得再好的集群，其包含的存储设备随着时间的推移也会逐渐趋于异构化，例如，由于批次不同而导致的磁盘容量差异。显然，通常情况下，我们不应该对所有设备一视同仁，而是需要在CRUSH算法中引入一个额外的参数，称为权重，来体现设备之间的差异，让权重大（对应容量大）的设备分担更多的数据，权重小（对应容量小）的设备分担更少的数据，从而使得数据在异构存储网络中也能合理地分布。

将上述理论应用于straw算法，则可以通过使用权重对签长的计算过程进行调整来实现，即我们总是倾向于让权重大的元素有较大的概率获得更大的签长，从而在每次抽签中更容易胜出。因此，引入权重之后straw算法的执行结果将取决于3个因素：固定输入、元素编号和元素权重。这其中，元素编号起的是随机种子的作用，所以针对固定输入，straw算法实际上只受元素权重的影响。进一步地，如果每个元素

的签长只与自身权重相关，则可以证明此时straw算法对于添加元素和删除元素的处理都是最优的。我们以添加元素为例进行论证。

1）假定当前集合中一共包含n个元素：

$(e_1，e_2，…，e_n)$

2）向集合中添加新元素$e_{n+1}$：

$(e_1，e_2，…，e_n，e_{n+1})$

3）针对任意输入x，加入$e_{n+1}$之前，分别计算每个元素签长并假定其中最大值为$d_{max}$：

$(d_1，d_2，…，d_n)$

4）因为新元素$e_{n+1}$的签长计算只与自身编号及自身权重相关，所以可以使用x独立计算其签长（同时其他元素的签长不受$e_{n+1}$加入的影响），假定为$d_{n+1}$；

5）又因为straw算法总是选择最大的签长作为最终结果，所以：

如果$d_{n+1}>d_{max}$，那么x将被重新映射至新元素$e_{n+1}$；反之，对x的已有映射结果无任何影响。

可见，添加一个元素，straw算法会随机地将一些原有元素中的数据重新映射至新加入的元素之中。同理，删除一个元素，straw算法会将该元素中全部数据随机地重新映射至其他元素之中。因此无论添加或者删除元素，都不会导致数据在除被添加或者删除之外的两个元素（即不相干的元素）之间进行迁移。

理论上straw算法是非常完美的，然而在straw算法实现整整8年之后，由于Ceph应用日益广泛，不断有用户向社区反馈，每次集群有新的OSD加入或者旧的OSD删除时，总会触发不相干的数据迁移，这迫使Sage对straw算法重新进行审视。原straw算法伪代码如下：

```
max_x = -1
max_item = -1
for each item:
    x = hash(input, r)
    x = x * item_straw
    if x > max_x:
        max_x = x
        max_item = item
return max_item
```

可见，算法执行的结果取决于每个元素根据输入（input）、随机因子（r）和item_straw计算得到的签长，而item_straw通过权重计算得到，伪代码如下：

```
reverse = rearrange all weights in reverse order
straw = -1
weight_diff_prev_total = 0
for each item:
    item_straw = straw * 0x10000
    weight_diff_prev = (reverse[current_item]-
reverse[prev_item])*items_remain
    weight_diff_prev_total += weight_diff_prev
    weight_diff_next = (reverse[next_item]-
reverse[current_item])*items_remain
    scale = weight_diff_prev_total/(weight_diff_prev_total +
weight_diff_next)
    straw *= pow(1 / scale, 1 / items_remain)
```

原straw算法中，将所有元素按其权重进行逆序排列后逐个计算每个元素的item_straw，计算过程中不断累积当前元素与前后元素的权重差值，以此作为计算下一个元素item_straw的基准。因此straw算法的最

终结果不但取决于每个元素的自身权重，而且也与集合中所有其他元素的权重强相关，从而导致每次有元素加入集合或者被从集合中删除时，都会引起不相干的数据迁移。

出于兼容性考虑，Sage重新设计了一种针对原有straw算法进行修正的新算法，称为straw2。

straw2计算每个元素的签长时仅使用自身权重，因此可以完美反映Sage的初衷（也因此避免了不相干的数据迁移），同时计算也更加简单。其伪代码如下：

```
max_x = -1
max_item = -1
for each item:
    x = hash(input, r)
    x = ln(x / 65536) / weight
    if x > max_x:
        max_x = x
        max_item = item
return max_item
```

分析straw2的伪代码可知，针对输入和随机因子执行哈希后，结果落在[0，65535]之间，因此x/65536必然小于1，对其取自然对数（ln(x/65536)）后结果为负值。进一步地，将其除以自身权重（weight）后，权重越大，结果越大（因为负得越少），从而体现出我们所期望的每个元素权重对于抽签结果的正反馈作用。

## 2.2 CRUSH算法详解

CRUSH基于上述基本选择算法完成数据映射，这个过程是受控的并且高度依赖于集群的拓扑描述——cluster map。不同的数据分布策略通过制定不同的placement rule来实现。placement rule实际上是一组包括最大副本数、故障（容灾）级别等在内的自定义约束条件，例如针对图2-1所示的集群，我们可以通过一条placement rule将互为镜像的3个数据副本（这也是Ceph的默认数据备份策略）分别写入位于不同机架的主机磁盘之上，以避免因为某个机架掉电而导致业务中断。

针对指定输入x，CRUSH将输出一个包含n个不同目标存储对象（例如磁盘）的集合。CRUSH的计算过程中仅仅使用x、cluster map和placement rule作为哈希函数输入，因此如果cluster map不发生变化（一般而言placement rule不会轻易变化），那么结果就是确定的；同时由于使用的哈希函数是伪随机的，所以CRUSH选择每个目标存储对象的概率相对独立（然而我们在后面将会看到，受控的副本策略改变了这种独立性），从而可以保证数据在整个集群之间均匀分布。

### 2.2.1 集群的层级化描述——cluster map

cluster map是Ceph集群拓扑结构的逻辑描述形式。考虑到生产环境中集群通常具有形如“数据中心→机架→主机→磁盘”（参考图2-1）这样的层级拓扑，所以cluster map使用树这种数据结构来实现：每个叶子节点都是真实的最小物理存储设备（例如磁盘），称为device；所有中间节点统称为bucket，每个bucket可以是一些devices的集合，也可以是低一级的buckets集合；根节点称为root，是整个集群的入口。每个节点绑定一种类型，表示其在集群中所处的层级，也可以作为故障域，除此之外还包含一个集群唯一的数字标识。根节点与中间节点的数字标识都是负值，只有叶子节点，也就是device才拥有非负的数字标识，表明其是承载数据的真实设备。节点的权重属性用于对CRUSH的选择过程进行调整，使得数据分布更加合理。父节点权重是其所有孩子节点的权重之和。

表2-2列举了cluster map中一些常见的节点（层级）类型。

表2-2　cluster map常见的节点类型

| 类型ID | 类型名称 | 类型ID | 类型名称 |
| --- | --- | --- | --- |
| 0 | osd | 6 | pod |
| 1 | host | 7 | room |
| 2 | chassis | 8 | datacenter |
| 3 | rack | 9 | region |
| 4 | row | 10 | root |
| 5 | pdu | | |

需要注意的是，这里并非强调每个Ceph集群都一定要划分为如表2-2所示的11个层级，并且每种层级类型的名称必须固定不变，而是层级可以根据实际情况进行裁剪，名称也可以按照自身习惯进行修改。假定所有磁盘规格一致（这样每个磁盘的权重一致），我们可以绘制图2-1所示集群的cluster map，如图2-2所示。

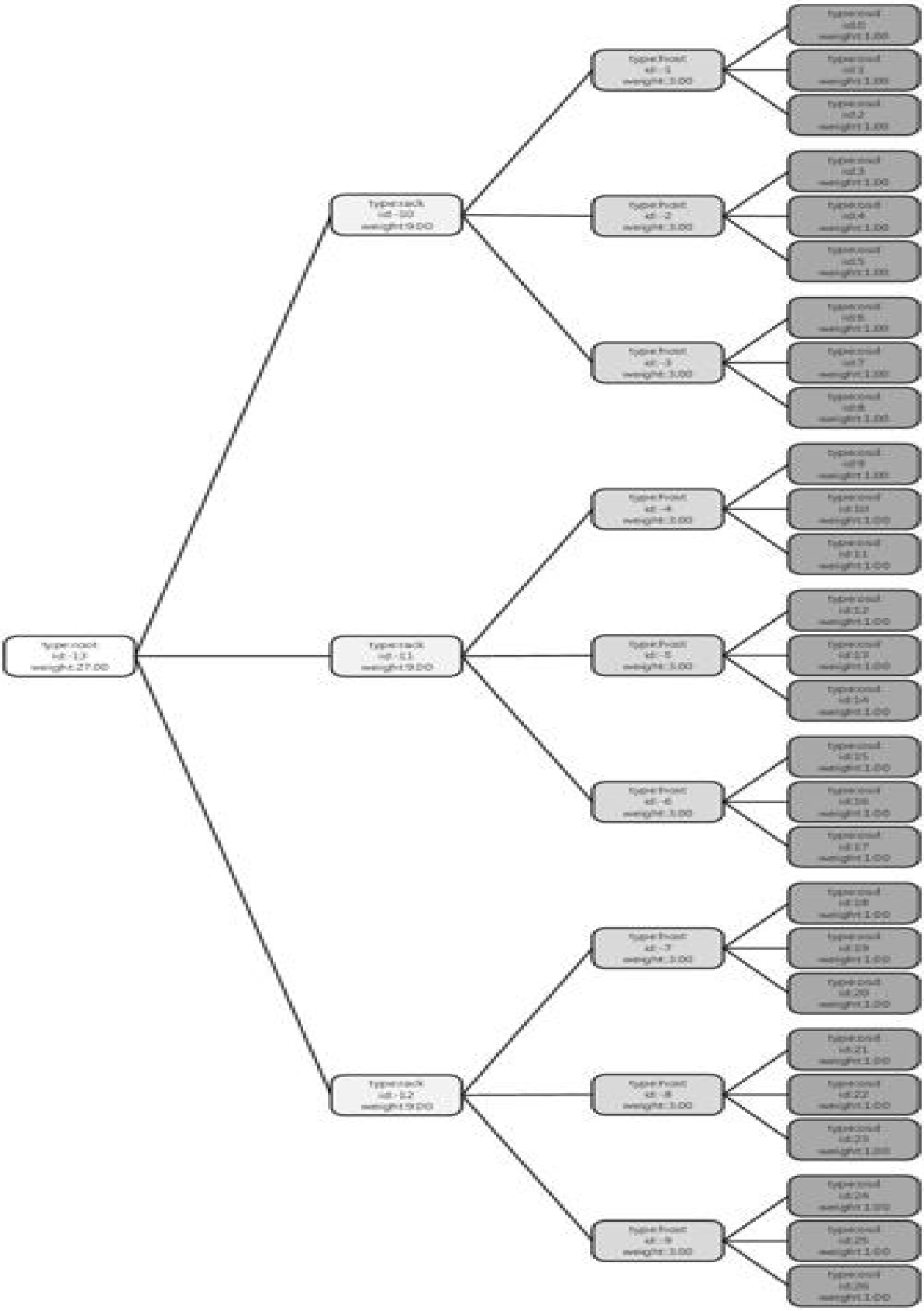

type:root
id:-13
weight:27.00
type:rack
id:-10
weight:9.00
type:rack
id:-11
weight:9.00
type:rack
id:-12
weight:9.00
type:host
id:-1
weight:3.00
type:host
id:-2
weight:3.00
type:host
id:-3
weight:3.00
type:host
id:-4
weight:3.00
type:host
id:-5
weight:3.00
type:host
id:-6
weight:3.00
type:host
id:-7
weight:3.00
type:host
id:-8
weight:3.00
type:host
id:-9
weight:3.00
type:osd
id:0
weight:1.00
type:osd
id:1
weight:1.00
type:osd
id:2
weight:1.00
type:osd
id:3
weight:1.00
type:osd
id:4
weight:1.00
type:osd
id:5
weight:1.00
type:osd
id:6
weight:1.00
type:osd
id:7
weight:1.00
type:osd
id:8
weight:1.00
type:osd
id:9
weight:1.00
type:osd
id:10
weight:1.00
type:osd
id:11
weight:1.00
type:osd
id:12
weight:1.00
type:osd
id:13
weight:1.00
type:osd
id:14
weight:1.00
type:osd
id:15
weight:1.00
type:osd
id:16
weight:1.00
type:osd
id:17
weight:1.00
type:osd
id:18
weight:1.00
type:osd
id:19
weight:1.00
type:osd
id:20
weight:1.00
type:osd
id:21
weight:1.00
type:osd
id:22
weight:1.00
type:osd
id:23
weight:1.00
type:osd
id:24
weight:1.00
type:osd
id:25
weight:1.00
type:osd
id:26
weight:1.00

图2-2　图2-1所示集群的cluster map描述

实现上，类似图2-2中这种树状的层级关系在cluster map中可以通过一张二维映射表建立。

```
<bucket, items>
```

树中每个节点都是一个bucket（device也被抽象成为一种bucket类型），每个bucket都只保存自身所有直接孩子的编号。当bucket类型为device（对应图2-2中的osd）时，容易知道，此时其对应的items列表为空，表明bucket实际上是叶子节点。

### 2.2.2 数据分布策略——placement rule

使用cluster map建立对应集群的拓扑结构描述之后，可以定义placement rule来完成数据映射。

每条placement rule可以包含多个操作，这些操作有以下3种类型。

1）take：从cluster map选择指定编号的bucket（即某个特定bucket），并以此作为后续步骤的输入。例如，系统默认的placement rule总是以cluster map中的root节点作为输入开始执行的。

2）select：从输入的bucket中随机选择指定类型和数量的条目（items）。Ceph当前支持两种备份策略，多副本和纠删码，相应的有两种select算法，firstn和indep。实现上两者都是采用深度优先遍历算法，并无显著不同，主要区别在于纠删码要求结果是有序的，因此，如果无法得到满足指定数量（例如4）的输出，那么firstn会返回形如[1，2，4]这样的结果，而indep会返回形如[1，2，CRUSH_ITEM_NONE，4]这样的结果，即indep总是返回要求数量的条目，如果对应的条目不存在（即选不出来），则使用空穴进行填充。select执行过程中，如果选中的条目故障、过载或者与其他之前已经被选中的条目冲突，都会触发select重新执行，因此需要指定最大尝试次数，防止select陷入死循环。

3）emit：输出选择结果。

可见，placement rule中真正起决定性作用的是select操作。

为了简化placement rule配置，select操作也支持故障域模式。以firstn为例，如果为故障域模式，那么firstn将返回指定数量的叶子设备，并保证这些叶子设备位于不同的、指定类型的故障域之下。因此，在故障域模式下，一条最简单的placement rule可以只包含如下3个操作：

```
take(root)
select(replicas, type)
emit(void)
```

上述select操作中的type为想要设置的故障域类型，例如设置为rack，则select将保证选出的所有副本都位于不同机架的主机磁盘之上；也可以设置为host，那么select只保证选出的所有副本都位于不同主机的磁盘之上。

图2-3以firstn为例，展示了select从指定的bucket当中查找指定数量条目的过程。

图2-3中几个关键处理步骤补充说明如下：

1）如何从bucket下选择一个条目（item）？

构建集群的cluster map时，通过分析每种类型的bucket特点，可以为其指定一种合适的选择算法（例如straw2），用于从对应的bucket中选择一个条目。因此从bucket选择条目的过程实际上就是执行设定的选择算法。这个选择算法的执行结果取决于两个因素：一是输入对象的特征标识符x，二是随机因子r（r实际上是哈希函数的种子）。因为x固定不变，所以如果选择失败，那么在后续重试的过程中需要对r进行调整，以尽可能输出不同的结果。目前r由待选择的副本编号和当前的尝试次数共同决定。

为了防止陷入死循环，需要对选择每个副本过程中的尝试次数进行限制，这个限制称为全局尝试次数（choose_total_tries）；同时由于在故障域模式下会产生递归调用，所以还需要限制产生递归调用时作为下一级输入的全局尝试次数。由于这个限制会导致递归调用时的全局尝试次数成倍增长，实现上采用一个布尔变量（chooseleaf_descend_once）进行控制，如果为真，则在产生递归调用时下一级被调用者至多重试一次，反之则下一级被调用者不进行重试，由调用者自身重试。为了降低冲突概率（如前，每次尽量使用不同的随机因子r可以减少冲突概率），也可以使用当前的重试次数（或者其$2^{N-1}$倍，这里的N由chooseleaf_vary_r参数决定）对递归调用时的随机因子r再次进行调整，这样产生递归调用时，其初始随机因子r将取决于待选择的副本编号和调用者传入的随机因子（称为parent_r）。

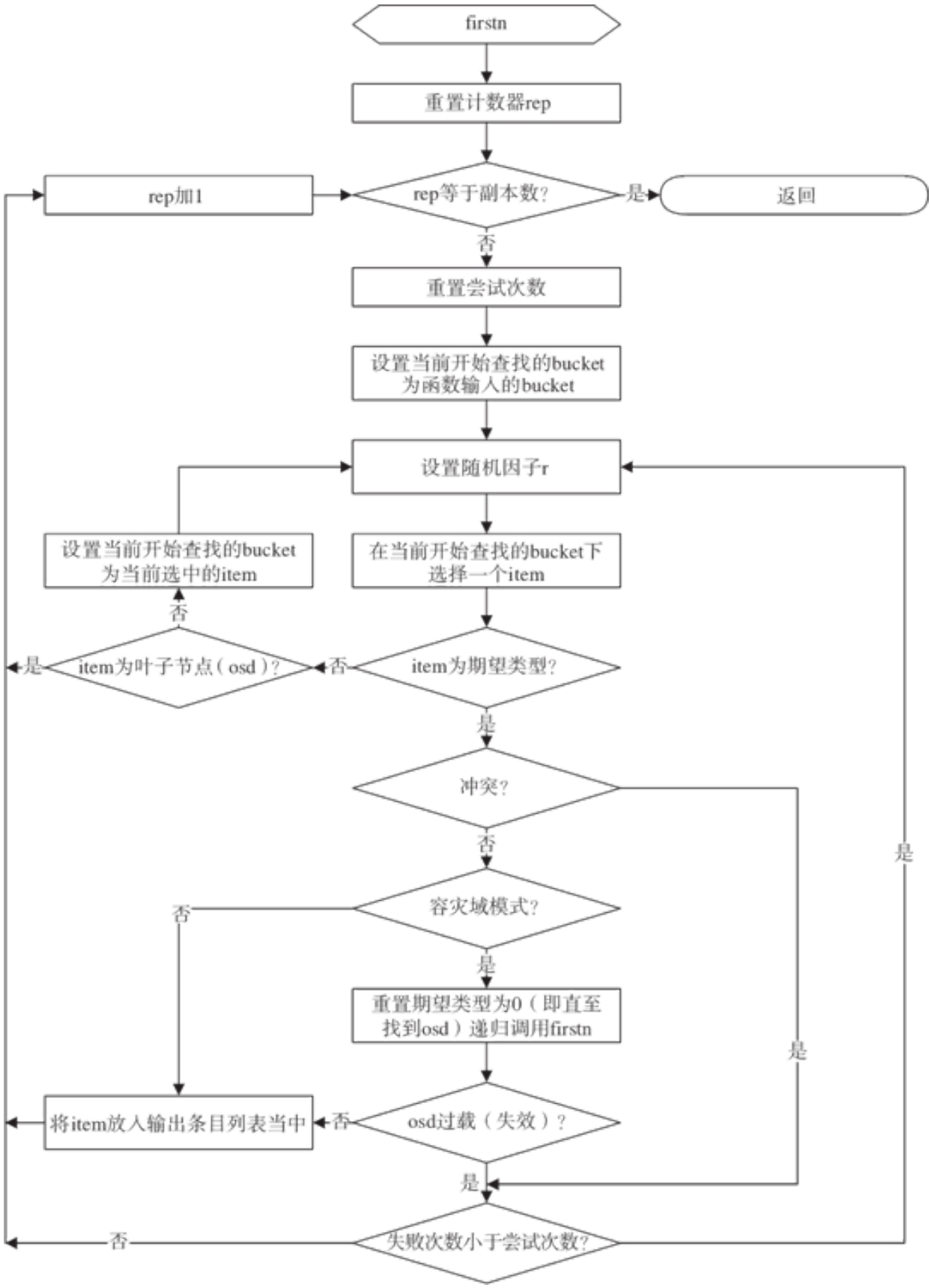

firstn
重置计数器rep
rep加1
rep等于副本数?
是
返回
否
重置尝试次数
设置当前开始查找的bucket
为函数输入的bucket
设置随机因子r
设置当前开始查找的bucket
为当前选中的item
在当前开始查找的bucket下
选择一个item
否
是
item为叶子节点(osd)?
否
item为期望类型?
是
冲突?
是
否
容灾域模式?
否
是
重置期望类型为0(即直至
找到osd)递归调用firstn
将item放入输出条目列表当中
否
osd过载(失效)?
是
是
否
失败次数小于尝试次数?
是

图2-3　基于firstn的select执行过程

值得一提的是，Jewel版本之前，故障域模式下作为递归调用时所使用的副本编号是固定的，例如调用者当前正在选择第2个副本，那么执行递归调用时的起始副本编号也将是2。按照上面的分析，副本编号会作为输入参数之一对递归调用时的初始随机因子r产生影响，有的用户反馈，这在OSD失效时会触发不必要的数据（PG）迁移，因此在Jewel版本之后，故障域模式下会对递归调用的起始副本独立编号（这个操作受chooseleaf_stable控制），以进一步降低两次调用之间的相关性。

由于选择的过程是执行深度优先遍历，在老的CRUSH实现中，如果对应集群的层次较多，并且在中间某个层次的bucket下由于冲突而选择条目失败，那么可以在当前的bucket下直接进行重试，而不用每次回归到初始输入的bucket之下重新开始重试，这样可以稍微提升算法的执行效率，此时同样需要对这个局部重试过程中的重试次数进行限制，称为局部重试次数（choose_local_retries）。此外，由于进入这种模式的直接原因是bucket自带选择算法冲突概率较高（即使用不同的r作为输入反复选中同一个条目），所以针对这种模式还设计了一种后备选择算法。这种后备选择算法的基本原理是：将对应bucket下的所有条目进行随机重排，只要输入x不变，那么随着r变化，算法会不停地记录前面已经被选择过的条目，并从本次选择的候选条目中排除，从而能够有效地降低冲突概率，保证最终能够成功选中一个不再冲突的条目。切换至后备选择算法需要冲突次数达到某个阈值，其主要由当前bucket的规模决定（原实现中要求冲突次数至少大于当前bucket下条目数的一半）。当然切换至后备选择算法时，也可以再次限制启用后备选择算法进行重试的次数（choose_local_fallback_retries）。上述过程因为对整个CRUSH的执行过程进行了大量人工干预从而严重损伤了CRUSH的伪随机性（即公平性），所以会导致严重的数据均衡问题，因此在Ceph的第一个正式发行版Argonaut之后即被废弃，不再建议启用。

表2-3汇总了如上分析的、所有影响CRUSH执行的可调参数（表中的默认值和最优值针对Jewel版本而言）。

表2-3　CRUSH可调参数

| 参数名称 | 默认值/最优值 | 说　明 |
| --- | --- | --- |
| choose_local_tries | 0/0 | 已废弃，不建议进行调整 |
| choose_local_fallback_tries | 0/0 | |

（续）

| 参数名称 | 默认值/最优值 | 说　明 |
| --- | --- | --- |
| choose_total_tries | 50/50 | 如果集群比较大，层次比较多，或者每个主机下的磁盘数量比较少，可能会导致CRUSH无法选出足够的OSD完成所有副本映射，此时可以通过设置更大的choose_total_tries加以解决 |
| chooseleaf_descend_once | 1/1 | 控制故障域模式下产生递归调用时的重试次数。<br>至多产生一次递归调用，递归时如果此标志位置位，至多重试一次。<br>不建议进行调整 |
| chooseleaf_vary_r | 1/1 | 不建议进行调整 |
| chooseleaf_stable | 1/1 | 不建议进行调整 |
| straw_calc_version | 1/1 | 用于对straw类型bucket下的条目权重计算过程进行校正。由于straw算法已经被废弃，所以本参数对于新建集群无影响 |

2）冲突

冲突指选中的条目已经存在于输出条目列表之中。

3）OSD过载（或失效）

虽然哈希以及由哈希派生出来的CRUSH算法从理论上能够保证数据在所有磁盘之间均匀分布，但是实际上，以下因素：

·集群规模较小，集群整体容量有限，导致集群PG总数有限，亦即CRUSH输入的样本容量不够。

·CRUSH本身的缺陷。CRUSH的基本选择算法中，以straw2为例，每次选择都是计算单个条目被选中的独立概率，但是CRUSH所要求的副本策略使得针对同一个输入、多个副本之间的选择变成了计算条件概率（我们需要保证副本位于不同故障域中的OSD之上），所以从原理上CRUSH就无法处理好多副本模式下的副本均匀分布问题。

都会导致在真实的Ceph集群、特别是异构集群中，出现大量磁盘数据分布悬殊（这里指每个磁盘已用空间所占的百分比）的情况，因此需要对CRUSH计算结果进行人工调整。这个调整同样是基于权重进行的，即针对每个叶子设备（OSD），除了由其基于容量计算得来的真实权重（weight）之外，Ceph还为其设置了一个额外的权重，称为reweight。算法正常选中一个OSD后，最后还将基于此reweight对该OSD进行一次过载测试，如果测试失败，则仍将拒绝选择该条目，这个过程如图2-4所示。

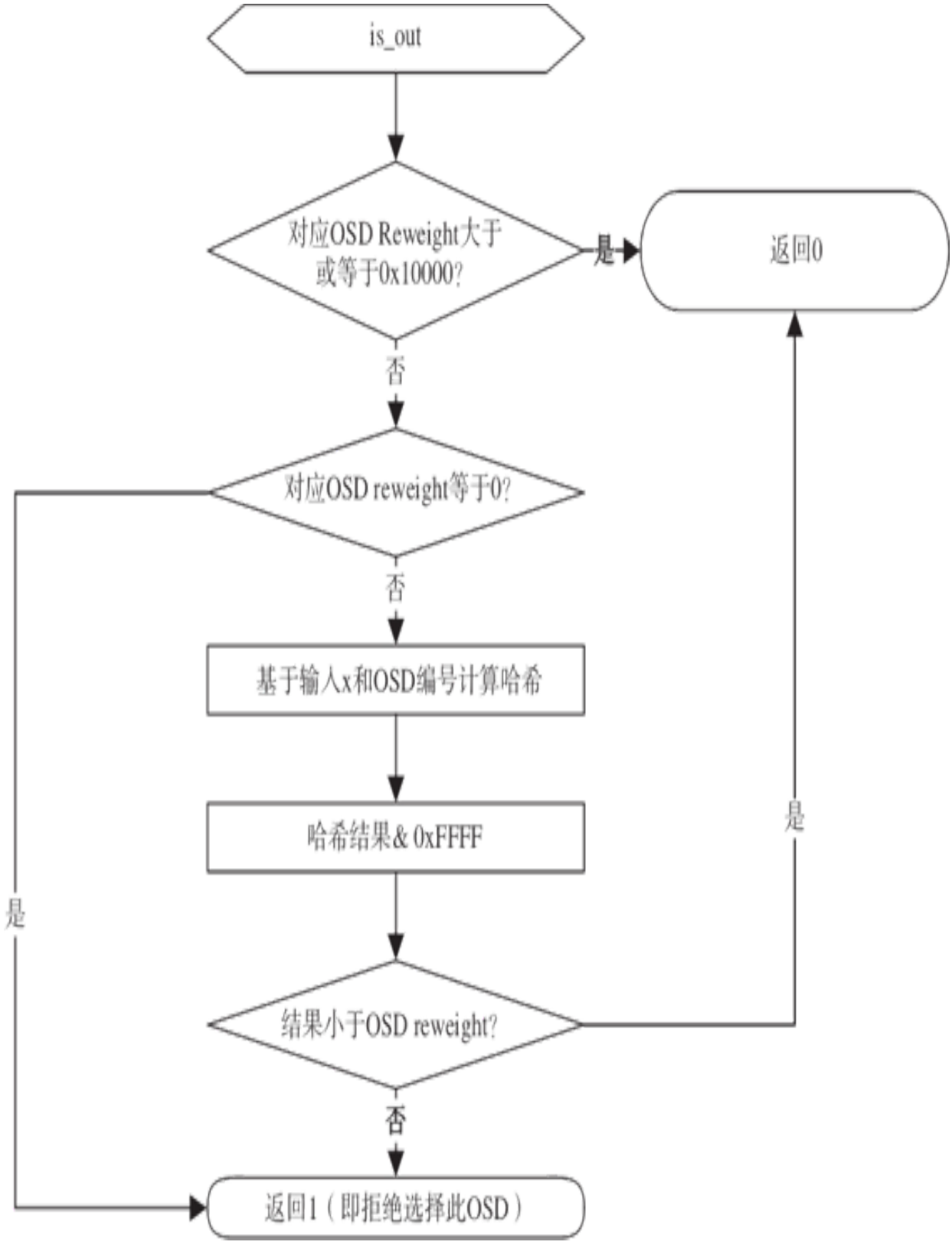

is_out
对应OSD Reweight大于
或等于0x10000?
是
返回0
否
对应OSD reweight等于0?
否
基于输入x和OSD编号计算哈希
哈希结果& 0xFFFF
结果小于OSD reweight?
是
是
否
返回1（即拒绝选择此OSD）

图2-4　过载测试

由图2-4可见，对应OSD的reweight调整得越高，那么通过测试的概率越高（例如手动设置某个OSD的reweight为0x10000，那么通过测试的概率是100%），反之则通过测试的概率越低。因此实际应用中，通过降低过载OSD或者（和）增加低负载OSD的reweight，都可以触发数据在OSD之间重新分布，从而使得数据分布更加合理。

引入过载测试的另一个好处在于可以对OSD暂时失效和OSD被永久删除的场景进行区分。区分这两者的意义在于：如果OSD暂时失效（例如对应的磁盘被拔出超过一定时间，Ceph会将其设置为out），可以通过将其reweight调整为0，从而利用过载测试将其从候选条目中淘汰，进而将其承载的数据迁移至其他OSD，这样后续该OSD正常回归时，将其reweight重新调整为0x10000即可将原来归属于该OSD的数据再次迁回，而迁回过程中只需要同步该OSD离线期间产生的新数据即可，亦即只需要进行增量同步；相反，如果是删除OSD，此时会同步将其从对应的bucket条目中删除，这样即便该OSD后续被重新添加回集群，由于其在cluster map中的唯一编号可能已经发生了变化，所以也可能承载与之前完全不同的数据。

初始时，Ceph将每个OSD的reweight都设置为0x10000，因此上述过载测试对CRUSH的最终选择结果不会产生任何影响。

## 2.3 调制CRUSH

按照Ceph的设计，任何涉及客户端进行对象寻址的场景都需要基于CRUSH进行计算，所以提升CRUSH的计算效率具有重要意义。调制CRUSH即针对表2-3中的参数进行调整，使得CRUSH正常工作的同时花费尽可能小的计算代价，以提升性能。需要注意的是，为了使得调整后的参数正常生效，需要保证客户端与RADOS集群的Ceph版本严格一致。

调制CRUSH最简单的办法是使用现成的、预先经过验证的模板（profile），命令如下：

```
ceph osd crush tunables {profile}
```

一些系统已经预先定义好的模板（截至Luminous版本）如表2-4所示。

表2-4　系统自定义的CRUSH可调参数模板

| 模板名称 | 说　明 |
| --- | --- |
| argonaut | 最初的CRUSH版本，支持后备选择算法（通过设置choose_local_tries和choose_local_fallback_tries启用），该算法已被废弃。<br>此外这个模板中choose_total_tries值为19，已经被证明在大部分用于生产环境的集群中都无法正常工作（无法选出足够的副本数） |
| bobtail | 将choose_total_tries设置为经过大量生产环境验证、更合理的50；引入chooseleaf_descend_once，对故障域模式下的重试次数进行控制 |
| firefly | 引入chooseleaf_vary_r，对故障域模式下，产生递归调用时作为下一级输入的随机因子r进行调整，减少冲突（失败）概率 |
| hammer | 增加straw2算法支持 |

（续）

| 模板名称 | 说　明 |
| --- | --- |
| jewel | 引入chooseleaf_stable，减少不相关数据迁移 |
| legacy | 同argonaut |
| optimal | 同Jewel |
| default | 同Jewel，这也是一个新集群创建时默认所启用的CRUSH可调参数 |

当然在一些特定的场景（例如集群比较大同时每个主机中的磁盘数量都比较少）中，可能使用上述模板仍然无法使得CRUSH正常工作，那么就需要手动进行参数调整。

### 2.3.1 编辑CRUSH map

为了方便将CRUSH的计算过程作为一个相对独立的整体进行管理（因为内核客户端也需要用到），Ceph将集群的cluster map和所有的placement rule合并成一张CRUSH map，因此基于CRUSH map可以独立实施数据备份及分布策略。一般而言，通过CLI（Command-Line Interface，命令行接口）即可方便地在线修改CRUSH的各项配置。当然也可以通过直接编辑CRUSH map实现，步骤如下：

#### （1）获取CRUSH map

大部分情况下集群创建成功后，对应的CRUSH map已经由系统自动生成，可以通过如下命令获取：

```
ceph osd getcrushmap -o {compiled-crushmap-filename}
```

上述命令将输出集群的CRUSH map至指定文件。当然出于测试或者其他目的，也可以手动创建CRUSH map，命令如下：

```
crushtool -o {compiled-crushmap-filename} --build --num_osds
Nlayer1...
```

其中，--num_osds{N}layer1...将N个OSD从0开始编号，然后在指定的层级之间平均分布，每个层级（layer）需要采用形如<name，algorithm，size>（其中size指每种类型的bucket下包含条目的个数）的三元组进行描述，并按照从低（靠近叶子节点）到高（靠近根节点）的顺序进行排列。例如，可以用如下命令生成图2-2所示集群的CRUSH map（osd->host->rack->root）：

```
crushtool -o mycrushmap --build --num_osds 27 host straw2 3 rack
straw2 3 \
    root uniform 0
```

需要注意的是，上述两种方式输出的CRUSH map都是经过编译的，需要经过反编译之后才能以文本的方式被编辑。

### （2）反编译CRUSH map

执行命令：

```
crushtool -d {compiled-crushmap-filename} -o {decompiled-
crushmap-filename}
```

即可将步骤（1）中输出的CRUSH map转化为可直接编辑的文本形式。例如：

```
crushtool -d mycrushmap -o mycrushmap.txt
```

### （3）编辑CRUSH map

得到反编译后的CRUSH map之后，可以直接以文本形式打开和编辑。例如可以直接修改表2-3中的所有可调参数：

```
vi mycrushmap.txt
# begin crush map
tunable choose_local_tries 0
tunable choose_local_fallback_tries 0
tunable choose_total_tries 50
tunable chooseleaf_descend_once 1
tunable chooseleaf_vary_r 1
tunable straw_calc_version 1
```

也可以修改placement rule：

```
vi mycrushmap.txt
# rules
rule replicated_ruleset {
    ruleset 0
    type replicated
    min_size 1
    max_size 10
    step take root
    step chooseleaf firstn 0 type host
    step emit
}
```

上述placement rule中各个选项含义如表2-5所示。

表2-5　CRUSH map中ruleset相关选项及其具体含义

| 选项名称 | | 含　义 |
| --- | --- | --- |
| ruleset | | 对应规则（集）的唯一编号。<br>不同的存储池可以使用不同的 ruleset。<br>引入 ruleset 的本意是：允许每个存储池通过（ruleset）来绑定多条 placement rule，以实现更加灵活的数据分布策略。但是由于 placement rule 自身就可以包含多个操作，所以 ruleset 并没有起到想象中的作用，在 Luminous 版本中已经被废弃（统一为 rule（即 placement rule）这个更加简单的概念） |
| type | | 副本策略，包含如下选项：<br>• replicated<br>• erasure |
| min_size | | 用于对选择副本数的范围进行约束 |
| max_size | | |
| step | take | 操作类型，前文已有介绍，这里仅对 step chooseleaf firstn 0 type host 补充说明如下：<br>• chooseleaf，故障域模式，可以替换为 choose，后者对应非故障域模式。<br>• firstn，两种选择算法之一，可以替换为 indep。<br>• 0，表示由具体的调用者指定输出的副本数，例如不同的存储池可以使用同一套 CRUSH 规则（拥有相同的备份策略），但是可以拥有不同的副本数。<br>• type，对应 chooseleaf 操作，指示输出必须是分布在由本选项指定类型的、不同的 bucket 之下的叶子节点；对应 choose 操作，指示输出类型 |
| | chooseleaf | |
| | emit | |

因此上述placement rule表示“采用多副本数据备份策略，副本必须位于不同主机的磁盘之上”。

### （4）编译CRUSH map

以文本形式编辑过的CRUSH map，相应地需要经过编译，才能被Ceph识别。执行如下命令：

```
crushtool -c {decompiled-crush-map-filename} -o {compiled-crush-map-filename}
```

### （5）模拟测试

在新的CRUSH map生效之前，可以先进行模拟测试，以验证对应的修改是否符合预期。例如，可以使用如下命令打印输入范围为[0，9]、副本数为3、采用编号为0的ruleset的映射结果：

```
crushtool -i mycrushmap --test --min-x 0 --max-x 9 --num-rep 3 --ruleset 0 \
    --show_mappings
CRUSH rule 0 x 0 [19,11,3]
CRUSH rule 0 x 1 [15,7,21]
CRUSH rule 0 x 2 [26,5,14]
CRUSH rule 0 x 3 [8,25,13]
CRUSH rule 0 x 4 [5,13,21]
CRUSH rule 0 x 5 [7,25,16]
CRUSH rule 0 x 6 [17,25,8]
CRUSH rule 0 x 7 [13,4,25]
CRUSH rule 0 x 8 [18,5,15]
CRUSH rule 0 x 9 [26,3,16]
```

也可以仅统计结果分布概况（这里输入变为[0，100000]）：

```
crushtool -i mycrushmap --test --min-x 0 --max-x 100000 --num-rep 3 \
        --ruleset 0 --show_utilization
rule 0 (replicated_ruleset), x = 0..100000, numrep = 3..3
rule 0 (replicated_ruleset) num_rep 3 result size == 3:
100001/100001
    device 0:               stored : 11243  expected : 11111.2
    device 1:               stored : 11064  expected : 11111.2
    device 2:               stored : 11270  expected : 11111.2
    device 3:               stored : 11154  expected : 11111.2
    device 4:               stored : 11050  expected : 11111.2
    device 5:               stored : 11211  expected : 11111.2
    device 6:               stored : 10848  expected : 11111.2
    device 7:               stored : 10958  expected : 11111.2
```

```
    device 8:              stored : 11203  expected : 11111.2
    device 9:              stored : 11031  expected : 11111.2
    device 10:             stored : 10997  expected : 11111.2
    device 11:             stored : 11165  expected : 11111.2
    device 12:             stored : 10993  expected : 11111.2
    device 13:             stored : 11188  expected : 11111.2
    device 14:             stored : 11150  expected : 11111.2
    device 15:             stored : 11222  expected : 11111.2
    device 16:             stored : 11152  expected : 11111.2
    device 17:             stored : 11103  expected : 11111.2
    device 18:             stored : 11044  expected : 11111.2
    device 19:             stored : 11056  expected : 11111.2
    device 20:             stored : 11023  expected : 11111.2
    device 21:             stored : 11514  expected : 11111.2
    device 22:             stored : 11026  expected : 11111.2
    device 23:             stored : 10888  expected : 11111.2
    device 24:             stored : 11025  expected : 11111.2
    device 25:             stored : 11069  expected : 11111.2
    device 26:             stored : 11356  expected : 11111.2
```

### （6）注入集群

新的CRUSH map验证充分后，可以重新注入集群，使之生效。执行如下命令：

```
ceph osd setcrushmap -i  {compiled-crushmap-filename}
```

## 2.3.2 定制CRUSH规则

在上一节中，介绍了编辑CRUSH map的一般方法。本节介绍如何对CRUSH规则进行灵活定制，以满足特定需求。我们仍以图2-1所示的集群为例进行说明。

最常见的场景是需要提升故障域，例如默认的故障域一般为host级别，可以将其提升为rack。修改对应的ruleset（当然也可以新建一条ruleset）如下：

```
vi mycrushmap.txt
# rules
rule replicated_ruleset {
    ruleset 0
    type replicated
    min_size 1
    max_size 10
    step take root
    step chooseleaf firstn 0 type rack
    step emit
}
```

测试结果如下：

```
crushtool -i mycrushmap --test --min-x 0 --max-x 9 --num-rep 3
\
    --ruleset 0 --show_mappings
CRUSH rule 0 x 0 [19,15,3]
```

```
CRUSH rule 0 x 1 [15,2,18]
CRUSH rule 0 x 2 [26,5,14]
CRUSH rule 0 x 3 [8,20,13]
CRUSH rule 0 x 4 [5,13,19]
CRUSH rule 0 x 5 [7,25,10]
CRUSH rule 0 x 6 [17,25,5]
CRUSH rule 0 x 7 [13,4,18]
CRUSH rule 0 x 8 [18,8,11]
CRUSH rule 0 x 9 [26,1,16]
```

---

可见，此时所有副本都位于不同rack的OSD之上。

也可以限制只选择某个特定rack（例如rack2）下的OSD。例如：

---

```
vi mycrushmap.txt
# rules
rule replicated_ruleset {
    ruleset 0
    type replicated
    min_size 1
    max_size 10
    step take rack2
    step chooseleaf firstn 0 type host
    step emit
}
```

---

测试结果如下：

---

```
crushtool -i mycrushmap --test --min-x 0 --max-x 9 --num-rep 3
\
    --ruleset 0 --show_mappings
CRUSH rule 0 x 0 [19,21,26]
CRUSH rule 0 x 1 [20,23,26]
CRUSH rule 0 x 2 [26,20,22]
CRUSH rule 0 x 3 [22,25,18]
CRUSH rule 0 x 4 [21,26,18]
CRUSH rule 0 x 5 [21,25,19]
CRUSH rule 0 x 6 [19,25,23]
CRUSH rule 0 x 7 [21,18,25]
CRUSH rule 0 x 8 [18,24,21]
```

---

可见，此时所有副本都被限制在rack2（包含OSD编号范围[18，26]）之下的OSD上。

另外需要注意的是，在CRUSH map中，除了OSD（叶子节点）之外，其他层级关系都是虚拟的，不管其有无实际的物理实体对应，这为灵活定制CRUSH提供了更大的便利。例如，下面这个极端的例子中，我们通过新建一个虚拟的host，以此为基础限制所有副本都必须分布在编号为0、9、18这3个特定的OSD上。

---

```
vi mycrushmap.txt
host virtualhost {
    id -14          # do not change unnecessarily
    # weight 3.000
    alg straw2
    hash 0  # rjenkins1
    item osd.0  weight 1.000
    item osd.9  weight 1.000
    item osd.18 weight 1.000
}

# rules
rule customized_ruleset {
    ruleset 1
    type replicated
    min_size 1
    max_size 10
    step take virtualhost
    step chooseleaf firstn 0 type osd
    step emit
}
```

---

实际测试结果如下：

---

```
crushtool -i mycrushmap --test --min-x 0 --max-x 9 --num-rep 1
\
    --ruleset 1 --show_mappings
CRUSH rule 0 x 0 [0]
CRUSH rule 0 x 1 [9]
CRUSH rule 0 x 2 [9]
```

```
CRUSH rule 0 x 3 [0]
CRUSH rule 0 x 4 [18]
CRUSH rule 0 x 5 [18]
CRUSH rule 0 x 6 [18]
CRUSH rule 0 x 7 [18]
CRUSH rule 0 x 8 [18]
CRUSH rule 0 x 9 [9]
```

---

更复杂的例子可以通过将上面这些例子进行适当的组合得到，这里不赘述。

## 2.4 数据重平衡

基于哈希随机分布数据的策略使得当Ceph集群中任意一个磁盘被写满时，整个集群就会被标记为满并阻止所有客户端后续写入。上述这种空间控制策略虽然看上去有些极端（例如将整个集群标记为满时，其整体空间使用率可能还不到70%），但却是这类存储系统的必要举措，因为我们无法预料客户端的写入最终会落到哪个磁盘之上。

在纠删码真正生产就绪之前，为了满足分布式系统的商用可靠性需求，Ceph当前最主流的数据备份策略还是3副本。受限于样本容量和CRUSH自身的缺陷，上述空间控制策略所带来的主要问题有：首先，在生产环境中，Ceph集群的空间利用率普遍不高，均值在23%左右，一些极端情况下可能更低，进而导致与传统存储相比，转投Ceph的空间成本居高不下；其次，参考经验数据，大部分本地文件系统在磁盘空间使用率超过80%时都会变得极其缓慢（例如ZFS在空间使用率超过80%并且碎片化比较严重时性能会下降一半左右），如果磁盘之间的空间使用率过于悬殊，此时Ceph出于强一致性考虑而设计的跨故障域多副本同步写策略又会使得系统的整体性能显著受木桶原理制约。

因此，为了解决上述问题，我们必须对集群数据分布进行调整，使得任意时刻集群中的所有OSD的空间使用率都尽可能地趋于一致。

一个容易忽略的事实是，虽然每个OSD的空间统计粒度可以细化到字节，但是为了使得所有OSD的空间使用率趋于均衡，我们无法以字节甚至对象为粒度，而只能以PG为粒度执行调整。这从原理上堵死了使得整个集群达到完美均衡状态的可能性。那么，退而求其次，我们能不能将每个OSD上的PG数量尽量调整至趋于均衡呢？在回答这个问题之前，我们首先需要研究为什么PG会在OSD之间分布不均。

由伪随机函数的特征，如果输入样本容量足够大，那么可以保证输出结果足够离散。换言之，如果将100万个对象随机映射到10个PG，那么有极大的概率可以使得每个PG上的对象数量偏差不超过1%；反之，如果样本容量很小，例如将100个PG随机映射到10个OSD，那么每个OSD上最终分布的PG数量则可能达到一个相对悬殊的状态。下面这个例子展示了一个小容量集群的PG分布情况：

```
ID CLASS WEIGHT REWEIGHT SIZE USE AVAIL %USE VAR PGS
0 ssd 1.00000 1.00000 10303M 1088M 9215M 10.57 1.00 11
1 ssd 1.00000 1.00000 10303M 1088M 9215M 10.57 1.00 10
2 ssd 1.00000 1.00000 10303M 1088M 9215M 10.57 1.00 7
3 ssd 1.00000 1.00000 10303M 1088M 9215M 10.57 1.00 6
4 ssd 1.00000 1.00000 10303M 1088M 9215M 10.57 1.00 8
5 ssd 1.00000 1.00000 10303M 1088M 9215M 10.57 1.00 6
    TOTAL 61823M 6533M 55290M 10.57
```

因此，如果整个集群的对象数量足够多，那么可以保证每个PG中的对象数量基本上是一致的；进一步地，如果保证每个OSD上的PG数量也一样多，那么理论上可以保证每个OSD分布的对象数量也趋于一致，从而保证每个OSD的空间使用率趋于一致。例如（说明：使用不同负载多次进行测试，结果比较接近，集群实际空间利用率在84%～92%浮动）：

```
ID CLASS WEIGHT REWEIGHT SIZE USE AVAIL %USE VAR PGS
0  ssd 0.54860 1.00000 561G 520G 42334M 92.64 1.00 123
1  ssd 0.54860 1.00000 561G 529G 33520M 94.17 1.01 123
```

```
2  ssd 0.54860 1.00000 561G 521G 41272M 92.83 1.00 123
3  ssd 0.54860 1.00000 561G 518G 44624M 92.24 0.99 123
4  ssd 0.54860 1.00000 561G 521G 40955M 92.88 1.00 124
5  ssd 0.54860 1.00000 561G 528G 34296M 94.04 1.01 123
6  ssd 0.54860 1.00000 561G 522G 40511M 92.96 1.00 123
7  ssd 0.54860 1.00000 561G 514G 47982M 91.66 0.99 123
8  ssd 0.54860 1.00000 561G 521G 41449M 92.79 1.00 123
9  ssd 0.54860 1.00000 561G 526G 36501M 93.65 1.01 123
10 ssd 0.54860 1.00000 561G 533G 28540M 95.04 1.02 123
11 ssd 0.54860 1.00000 561G 510G 52701M 90.84 0.98 124
12 ssd 0.54860 1.00000 561G 517G 45390M 92.11 0.99 123
13 ssd 0.54860 1.00000 561G 525G 37370M 93.50 1.01 123
14 ssd 0.54860 1.00000 561G 514G 48372M 91.59 0.99 123
15 ssd 0.54860 1.00000 561G 526G 36010M 93.74 1.01 123
    TOTAL 8988G 8352G 636G 92.92
```

---

为了让PG在每个OSD之间分布趋于均衡，一种办法是大幅度提升每个OSD上驻留的PG数量，然而实际上出于资源消耗和控制粒度的考虑，每个OSD上分布的PG数量不可能太多（例如Ceph推荐值是单个OSD承载100个PG），因此这个办法实际应用场景有限；另一种办法则是对PG映射至OSD的过程进行调整（或者说人工干预），这种调整又可以采用多种方式进行。我们接下来分别进行介绍。

### 2.4.1 reweight

CRUSH将PG映射至OSD的主要依据之一是每个OSD的权重（亦即理论或者期望的数据分布比重），也称为CRUSH weight，其反映了磁盘承载数据能力之间的物理差异，因此不便直接对其进行调整。

解决的办法是引入一个类似的概念，称为迁移权重（即上文中提到的reweight）。通过调整reweight的数值，可以使得一定数量的PG迁入或者迁出对应的OSD，从而使得OSD之间的PG数量趋于均衡。

注意到PG迁入和迁出的效果实际上是等同的，例如我们可以选择调低高负载（overload）OSD的reweight，使其PG迁出到其他低负载（underload）的OSD；也可以选择调高低负载OSD的reweight，使得某些PG从其他高负载的OSD迁入，因此实现上只需要进行单向调整即可。

通过reweight调整单个OSD中PG分布的过程如下：

首先查看集群的空间使用率统计。

```
ceph osd df tree
```

找到当前空间使用率较高的OSD，然后逐个执行：

```
ceph osd reweight {osd_num} {reweight}
```

上述命令中各个参数的含义如表2-6所示。

表2-6　reweight命令中的参数及其含义

| 参数 | 含　义 |
| --- | --- |
| osd_num | 必选，整型。<br>OSD 对应的数字 ID |
| reweight | 必选，浮点类型，取值范围：[0.1]。<br>待设置的 OSD 的 reweight。reweight 越小，将使得更多的数据从对应的 OSD 迁出 |

也可以批量调整，目前有两种模式，一种是按照OSD当前空间使用率（reweight-by-utilization），另一种是按照PG在OSD之间的分布（reweight-by-pg）。为了防止影响正常业务，可以先查看上述命令真正执行后，将会触发PG迁移情况的相关统计，以便规划执行调整的时机（以下都以reweight-by-utilization相关命令为例进行说明）。

```
ceph osd test-reweight-by-utilization {overload}{max_change} \
    {max_osds}{--no-increasing}
```

上述命令中各个参数的含义如表2-7所示。

表2-7　[test-]reweight-by-utilization命令中的参数及其含义

| 参数 | 含　义 |
| --- | --- |
| overload | 可选，整型，取值范围：≥100，默认值为120。<br>当且仅当某个OSD的空间使用率大于等于集群平均空间使用率的overload/100时，调整其reweight |
| max_change | 可选，浮点类型，取值范围：[0,1]，默认值为0.05。<br>每次调整reweight的最大幅度，即调整上限。实际每个OSD调整幅度取决于自身空间使用率与集群平均空间使用率的偏离程度，偏离越多，则调整幅度越大，反之则调整幅度越小 |
| max_osds | 可选，整型，默认值为4。<br>每次至多调整的OSD数目 |
| --no-increasing | 可选，如果携带，则不允许reweight上调（上调指将当前underload的OSD权重调大，让其分担更多的PG）；如果不携带，至多将OSD的reweight调整至1.0 |

例如：

```
ceph osd test-reweight-by-utilization 105 .2 4 --no-increasing
no change
moved 197 / 11016 (1.78831%)
avg 344.25
stddev 90.4592 -> 92.0923 (expected baseline 18.2618)
min osd.11 with 61 -> 60 pgs (0.177197 -> 0.174292 * mean)
max osd.31 with 424 -> 379 pgs (1.23166 -> 1.10094 * mean)

oload 105
max_change 0.2
max_change_osds 4
average 0.156612
overload 0.164443
osd.20 weight 1.000000 -> 0.844574
osd.27 weight 1.000000 -> 0.869125
osd.31 weight 1.000000 -> 0.890121
osd.13 weight 1.000000 -> 0.895248
```

由输出可见，本次如果真正执行reweight-by-utilization命令将导致：

·197个PG发生迁移。

·每个OSD承载的平均PG数目为344.25，执行本次调整后，标准方差将由90.4592变为92.0923。

·当前负载最轻的OSD为osd.11，只承载了61个PG，执行本次调整后，将承载60个PG。

·当前负载最重的OSD为osd.31，承载了424个PG，执行本次调整后，将减少到379个PG。

·执行本次调整后（命令可以重复执行），共计对osd.20、osd.27、osd.31、osd.13在内的4个OSD的reweight进行了调整。

输入以下命令将确认执行调整：

```
ceph osd reweight-by-utilization 105 .2 4 --no-increasing
```

因为调整reweight只是影响PG选择OSD的概率，这种调整方式最大的缺陷在于其具有极大的不确定性，无法实现PG在OSD之间迁移的精确控制，因此实际操作时可能需要根据经验反复进行多次，容易导致PG在OSD之间反复迁移，影响集群正常业务。同时，由于Ceph多副本策略影响了每个副本选择OSD的概率，如果副本数超过1，实际上使用唯一的CRUSH weight会存在先天性数据分布不均的缺陷，也很难通过调整reweight加以完美解决。

### 2.4.2 weight-set

考虑到存在多个副本的情况下，通过CRUSH选择不同位置的副本时，对应OSD呈现的概率都应该有所不同，也可以针对存储池中的每个OSD按照副本数设置一个权重组，称为weight-set。

因为Ceph支持共享存储池，所以每个OSD可以关联多个weight-set，每个weight-set所包含的元素数目和对应存储池的副本数相同，也就是说与CRUSH weight以及reweight直接和OSD绑定不同，weight-set是与存储池绑定的。至于如何合理地设置每个位置副本对应的权重，由于计算过程比较复杂，这里不做详述。

特别地，如果副本数为1，此时每个CRUSH bucket对应的weight-set将退化为一个一维数组，这在结构上和CRUSH weight完全等同。因此通过对CRUSH map编码时进行一些特殊的处理——例如将这个一维的weight-set直接替换掉对应bucket的CRUSH weight数组，则一些老版本的内核客户端也能够正确解析和理解CRUSH map。

上述这种特殊的、一维的weight-set也称为兼容模式的weight-set（compat weight-set）。当然，即便是多副本，也可以强制启用compat weight-set，其效果与直接调整reweight类似，但是由于weight-set是直接替换掉CRUSH weight进行数据映射，所以不存在多次数据迁移的问题。

两种模式的weight-set使用方法分别如下：

### （1）兼容模式

首先创建weight-set：

```
ceph osd crush weight-set create-compat
```

创建成功后，weight-set中每个OSD对应的权重会被自动初始化为其当前的CRUSH weight，可以通过下面的命令修改，以调整PG在OSD之间的分布。

```
ceph osd crush weight-set reweight-compat <item> <weight>
```

之后可以通过如下命令查看。

```
ceph osd crush dump
```

如果不再需要，则可以删除。

```
ceph osd crush weight-set rm-compat
```

### （2）非兼容模式

非兼容模式的weight-set需要与具体的存储池绑定，因此在创建weight-set时需要指定存储池名称。除此之外，非兼容模式下，weight-set本身又可以以两种形态存在：flat和positional。

在flat形态下，weight-set将退化为一个一维数组，其效果和兼容模式完全相同；而在positional形态下，则需要根据副本个数以及副本当前所处的位置（以3副本为例，CRUSH计算时分别称之为0号副本、1号副本和2号副本，编号既可以表示其在计算过程中的顺序，也可以表示副本所处的位置），为每个OSD指定一组权重。创建非兼容模式的weight-set命令如下：

```
ceph osd crush weight-set create <poolname> flat|positional
```

调整权重的命令与兼容模式类似，但是同样需要指定存储池名称。另外，如果是positional形态，则每个副本的权重都需要指定。例如（test为存储池名称，假定其为3副本）：

```
ceph osd crush weight-set create test positional
ceph osd crush weight-set reweight test osd.0 0.3 0.2 0.1
```

删除时同样需要指定存储池名称：

```
ceph osd crush weight-set rm <poolname>
```

引入weight-set的初衷是为了解决PG的多个副本在OSD之间完美分布的问题，然而实际上由于集群可能具有的复杂拓扑结构，随着副本数量上升，每个weight-set条目计算难度和计算量呈指数上升，导致其无法在生产环境中直接应用；而且，因为这种做法本质上仍然是在调整每次选择某个OSD的概率，与reweight类似，也存在无法精确控制效果的缺陷，导致其实际应用场景颇为受限。

### 2.4.3 upmap

为了解决由诸如永久性故障、扩容等因素可能导致正常业务长时间中断问题而引入的PG Temp机制，给了我们一个强烈提示：虽然PG到OSD之间的映射关系是直接通过CRUSH计算得到的，但是仍然可以通过PG Temp进行事后调整和校正。换言之，将PG Temp机制推广到更一般的场景，我们可以得到一种随心所欲调整PG分布的方法，称为upmap。

顾名思义，upmap作用在CRUSH完成选择之后，用于直接对CRUSH的选择结果进行调整。视调整粒度不同，当前有两种类型的upmap，一种针对CRUSH选择结果整体进行替换：

```
ceph osd pg-upmap <pgid><osdname (id|osd.id)> [<osdname (id|osd.id)>...]
```

另一种则只针对某个或者某些副本进行替换（注意：需要同时指定源和目的OSD，以实现将源OSD替换为目标OSD的功能）：

```
ceph osd pg-upmap-items <pgid><osdname (id|osd.id)> [<osdname (id|osd.id)>...]
```

如果不再需要（或者已经失效），则可以通过如下命令删除：

```
ceph osd rm-pg-upmap <pgid>
ceph osd rm-pg-upmap-items <pgid>
```

与前面两类方法不同，由于upmap可以针对CRUSH的选择结果进行精确校正，所以利用upmap很容易达成我们所梦寐以求的、PG在OSD之间完美分布的状态。然而遗憾的是，由于upmap直接作用于客户端寻址过程，所以如果客户端无法理解upmap（例如使用老版本的内核客户端），则无法启用upmap机制。

### 2.4.4 balancer

我们已经讨论了用于调整PG分布，进而调整集群数据分布、提升集群空间利用率和性能的一般性方法。如前所述，这3种方法除了各自的缺点之外，还有一个共同的缺点，即这种调整不是系统自发进行的，而是需要管理员手动干预，因而在实际的生产环境中使用起来非常不便。此外，由于PG具有自动迁移的特性，一旦集群发生故障、需要扩容等，那么之前手动调整的成果便会付之东流，又需要重复进行。因此，这类方法虽然看上去美好，但是却没有什么实用价值。

在Luminous版本中，Ceph为Mgr组件引入了一个balancer模块，用于对集群的数据分布自动进行调整。

balancer所依赖的工具就是前面提及的reweight、weight-set、upmap三种。这里的问题在于如何选择合适的时机进行调整以及如何评判每次调整的效果。

采用reweight方式执行人工调整时，我们的主要依据是集群每个OSD的空间使用率与平均值的偏离程度，亦即方差。然而直接套用方差作为调整依据存在一个潜在缺陷，我们以表2-8为例进行说明。

表2-8　处于不同时间点时，集群中各个OSD的空间使用率

| OSD | 空间使用率——情形 A | 空间使用率——情形 B |
| --- | --- | --- |
| 0 | 0.9 | 0.8 |
| 1 | 0.4 | 0.3 |
| 2 | 0.5 | 0.5 |

表2-8展示了同一个集群处于不同时间点时各个OSD空间使用率的分布情形，通过简单计算可知，上述两种情形的方差是相等的（均为0.374），但是实际上情形A要严峻得多（因为编号为0的OSD马上就要被写满了）。这说明我们在应用方差这种度量方法时，需要排除那些空间使用率低于均值的OSD的影响，即只用考虑那些过载的OSD。

基于这种修正后的方差计算公式我们可以建立评分系统：评分越高，说明集群空间使用率分布越不均衡，反之则说明集群空间使用率分布越趋于均衡。特别地，我们将评分为0的状态定义为集群空间使用率的完美均衡状态。

建立评分系统后，balancer可以针对集群当前空间使用率分布情况进行评估和打分，进而决策在什么时候启动调整。同时，因为可以实时生成一份调整后的集群表的拷贝，也可以复用这种评分系统对调整前后的效果预先进行评估，以判断是否能够执行调整以及调整到什么程度合适。

在生产环境中，除了考虑调整效果好坏之外，还需要考虑执行调整后对于集群正常业务的影响。一般而言，为了使得调整尽可能小地影响在线业务，可以采用单次微调结合周期性调整的方式进行。例如，我们可以每次只调整集群或者某个特定存储池1%的PG，每60s或者更长时间进行一次。特别地，如果前一次调整还没有完成则不要重复进行。

如果采用reweight/compat weight-set的调整方式，由于是定性调整，所以本质上是基于试探的方式进行。例如，针对系统当前所有过载的OSD进行排序，逐个将其权重调低一个比较合适的数值（称为调整步长），然后重新计算调整后的PG分布情况：如果评分下降，同时影响的PG数量也在合理范围之内，则确认执行调整；反之，如果评分上升，或者影响的PG数量太多，则说明步长设置得不合理，此时可以逐渐减小步长进行重试，直至满足条件为止。某些特殊场景下可能一直无法取得一个更加合理的数据分布（或者本身数据分布已经趋于完美均衡），则可以直接放弃本次调整，过一段时间之后再进行重试。如果采用upmap进行调整，由于本身可以针对PG的每个副本进行精确调整（即定量调整），则只需要考虑每次调整的PG数量在合理的范围之内即可。

balancer的调整默认针对整个集群进行，这种方式比较适合于整个集群都共享使用所有OSD的方式。当集群支持某些存储池独占式地使用OSD之后，为了防止针对某个特定存储池的调整也影响其他存储池，需要改造balancer，使之支持以存储池的粒度对PG进行调整。此外，除了周期性地对系统当前状态进行评估和调整之外，为了避免影响正常业务，也可以采用更加合理的调整方式，比如进一步规定能够进行调整的时间段、选择更加合理的时机进行调整等（例如在凌晨业务相对不那么繁忙的时间段执行调整，或者在集群扩容时顺便进行调整）。调整工具也可以依据集群的实际情况和用途进行切换，目前出于兼容性考虑，balancer默认使用的调整工具是compat weight-set，然而如果能够确认是全新创建的集群并且没有老版本的、内核态的客户端（即不存在兼容性问题），则完全可以考虑切换为upmap，调整效果更佳。

为了更好地理解balancer的工作流程，我们首先介绍几个与之相关的术语。

（1）plan

术语plan指balancer内部的一个优化任务。容易理解，集群的拓扑结构、空间负载情况一直处于变化之中，同时对集群PG的分布进行调整反过来又会对集群产生影响。因此，为了顺利执行调整，需要先创建一个优化任务，来记录集群当前的状态（包括PG分布情况、空间使用率等，其效果类似于快照），然后通过反复试探、对比，最终确定一个具体的优化方案，并记录在优化任务之中。

通常情况下plan的创建与执行是分离的，并没有时效限制。当然如果某个plan创建后长时间没有执行，对应的集群可能已经发生过剧烈变化，导致plan记录的信息和所选择的优化方案都已经过时，对应的plan也就失去了意义，此时可以简单删掉plan，重新创建然后再次执行。

（2）eval

术语eval指针对集群当前的数据分布状态进行评估，评估的对象可以是整个集群、某个存储池或者某个plan。

为了防止以偏概全，评估的依据或者说参考指标主要有3个，分别是PG数量、对象数量和字节数，实际得分为这三者得分的均值。

因为得分实际上为标准方差，得分越低，说明每个OSD的空间使用率与均值（期望值）差距越小。如果得分为0，则说明当前待评估对象的分布已经处于完美均衡状态。

（3）optimize

术语optimize指创建一个plan，粒度可以是整个集群、某个或者某些特定存储池。

（4）execute

术语execute指执行某个特定的plan。当某个plan被执行后（无论成功与否），即失去了价值，将会被balancer自动删除。

（5）mode

术语mode指balancer创建或者执行plan时，默认选择的工具和手段，当前有crush-compat、upmap和none。值得注意的是，none也用于间接关闭balancer。

了解相关术语之后，我们可以通过相关命令来详细了解balancer的工作流程。在正式启用balancer的自动平衡功能之前，可以先查看balancer自身状态。

```
ceph balancer status
```

其输出有如下几项。

·active：指示balancer当前是否启用了自动优化功能，该优化每60s执行一次，没有时间段限制，粒度为整个集群，默认为false（即关闭自动优化功能）。

·plans：指balancer当前已经创建，但是尚未执行的plan列表。

·mode：balancer执行调整的工具，当前有crush-compat、upmap和none三种，默认为none。注意，即使active为true，也必须选择一种有效的mode才能使能自动优化功能。

默认情况下，balancer不会启用自动优化功能，因此上述命令输出的plans一项为空，可以通过如下命令手动创建一个：

```
ceph balancer optimize <plan> {<pools> [<pools>...]}
```

上述命令可以带一个或者多个存储池名称，如果不带，则说明针对整个集群进行优化。创建好的plan将会出现在ceph balancer status命令的输出列表之中，可以通过如下命令直接执行：

```
ceph balancer execute <plan>
```

需要注意的是，该plan将在执行完成后自动被删除。如果不确定plan的执行效果，也可以先进行评估。

```
ceph balancer eval <plan> {<pools> [<pools>...]}
```

除了评估某个特定的plan之外，上述命令也可以针对整个集群、某个或者某些存储池当前的数据分布情况进行评估。如果plan效果不好，或者调整力度过大，可以直接删除（注意下面的命令将删除所有plan）。

```
ceph balancer reset
```

最后，可以通过如下两个命令来启用/停用balancer的自动优化功能：

```
ceph balancer on
ceph balancer off
```

在使能自动优化后，balancer在默认情况下将以60s为周期、以整个集群为粒度自动执行数据平衡。由于是在线数据平衡，如果集群正常业务对时延比较敏感，也可以通过重置自动优化执行的时间段、减小

每次调整力度等方式，来降低来自balancer的干扰。这可以通过修改如下配置项实现：

```
ceph config-key set mgr/balancer/begin_time 0000
ceph config-key set mgr/balancer/end_time 0600
ceph config-key set mgr/balancer/max_misplaced .01
```

调整上述配置项之后，自动优化将被限制仅在每天0～6点（注意是24小时制）之间进行，并且每次优化至多影响1%的PG。其他配置项可以查阅社区相关的文档获取，这里不赘述。

需要特别注意的是，在Mimic版本之前，balancer（以及驻留在Mgr中的其他插件）不支持这些配置项的动态加载，因此每次通过config-key命令修改了Mgr的任意配置项之后，都需要重启Mgr才能生效。

## 2.5 总结和展望

作为Ceph两大核心设计之一的CRUSH算法已经走过了10年时间。CRUSH良好的设计理念使其具有计算寻址、高并发和动态数据均衡、可定制的副本策略等基本特性，进而能够非常方便地实现诸如去中心化、有效抵御物理结构变化，并保证性能随集群规模呈线性扩展、高可靠等高级特性，因而非常适合Ceph这类对可扩展性、性能和可靠性都有严苛要求的大型分布式存储系统。

然而回到具体实现上，无论是list和tree算法先后被废弃，还是原创的straw算法被完全重写，无不证明CRUSH从来就不是完美的。也许CRUSH最为人诟病之处在于其引入扩展的副本策略支持之后所导致的数据不均衡问题，虽然从设计者的角度非常希望将这个问题交由CRUSH自身加以圆满解决，然而实现上由于Ceph所支持的集群可能具有复杂的拓扑结构，使得彻底解决这个问题困难重重。此外，在生产环境中，用户向社区抱怨在一些异构集群中CRUSH选不出足够副本数的声音从未停止，虽然可以通过针对CRUSH的一些参数进行调整加以解决（然而这从来就不是一件轻松的事情），但是相应地会以牺牲CRUSH的计算性能作为代价，因此也远远无法作为一个商业级成熟软件应有的解决方案。最后，基于计算寻址的设计使得无论何时针对CRUSH升级都要求客户端和RADOS集群同时进行，这通常会为内核客户端（krbd、kcephfs）类型的用户带来极大的困扰，同时由于Ceph

流控机制相对薄弱，升级过程中潜在的数据迁移问题有极大可能会影响到正常业务，进而成为在线升级方案中的隐患。

Luminous版本中balancer模块的引入让我们依稀看到了解决困扰社区已久的数据分布不均问题的曙光，然而其实际效果则仍有待广大用户检验。

第3章

# 集群的大脑——Monitor

在上一章中，我们系统介绍了CRUSH及相关的重要数据结构CRUSH map。实际上后者是作为集群表的一部分，由Monitor负责维护和进行传播的。

简言之，Monitor是基于Paxos兼职议会算法构建的、具有分布式强一致性的小型集群，主要负责维护和传播集群表的权威副本。Monitor采用负荷分担的方式工作，因此，任何时刻、任意类型的客户端或者OSD都可以通过和集群中任意一个Monitor进行交互，以索取或者请求更新集群表。基于Paxos的分布式一致性算法可以保证所有Monitor的行为自始至终都是正确和自洽的。

Paxos要求集群中超过半数Monitor处于活跃状态才能正常工作，以解决分布式系统中常见的脑裂问题。除此之外，在Ceph的工程实现中，由于（OSD）上下电、故障上报、故障切换等操作都由每个OSD独立与Monitor通过协商完成，并且每个操作都有可能需要多次修改集群表，为了防止一些临时性故障（例如网络拥塞）造成短时间内产生大量集群表更新和传播，进而对集群稳定性和性能造成冲击，Ceph对Paxos做了简化，将集群表的更新操作进行了串行化处理，即任意时刻

只允许由某个特定的Monitor统一发起集群表更新，并且一个时间段内所有的更新操作会被合并成一个单独的请求进行提交。

上面这个特殊的Monitor称为Leader，与之对应的其他Monitor则统称为Peon。Leader通过投票选举产生，一旦某个Monitor赢得超过半数的选票成为Leader，Ceph将保证在接下来的一个特定时间段内不会再有其他Monitor成为新的Leader，这个时间段被称为租期（Lease）。在正常运行过程中，Leader可以通过不断续租的方式来延长自身作为Leader的租期，但是一旦集群成员组成情况发生变化，例如Leader自身发生故障、新的Monitor加入或者老的Monitor退出等，整个集群就会重新触发Leader选举。

产生Leader之后，可以由其负责串行化集群表更新操作和发起集群表同步。在此之前，Leader需要先获取最新的集群表，这通过向集群成员表中除自身之外的每个成员发送Probe消息实现。顾名思义，Probe消息带有试探性质，每个收到Probe消息的成员必须在一个有限的时间间隔内（默认为2s）决定是否加入集群。如果最终确定加入，则通过Reply消息进行应答，并同步返回本地保存的集群表范围。一旦Leader确认有超过半数的成员进行了应答，并且成功地将集群表同步至最新版本之后，就可以向所有成员发布租期，开始其Leader生涯。

任意一个Peon检测到租期到期后Leader没有发起刷新（续租），就可以认为Leader已经异常，此时将触发集群重新进行Leader选举。反之，如果Leader发送续租请求后超时没有收到某个Peon的应答，则判定该Peon已经异常，同样将触发集群重新进行Leader选举。

每个租期内活跃的Monitor（包括Leader和Peon）都被自动赋予向客户端或者OSD分发集群表权威副本的权限，但是如果Peon收到集群表更新请求，则需要首先检查其是否为有效的更新请求。最终，所有经过确认的更新请求都被透传至Leader，由其负责串行化、执行合并、分配新的集群表版本号并在集群中发起同步。

基于2PC的同步机制和上述周期性的Leader租期刷新机制确保一旦租期内集群当前活跃的成员数目发生变化，那么对应的租期就会被收回，从而确保在此期间不会产生任何有效的集群表更新操作。因此，无论从集群中哪个成员发起更新，也无论集群成员如何变化，Monitor总是可以保证集群表的版本号单调递增。这有利于集群表的点对点传播，以及每个OSD或者客户端独立地完成集群表同步。

本章介绍Monitor的内部分工、集群表的具体构成，以及如何借助Monitor提供的CLI命令完成诸如增删OSD、存储池等集群管理操作。有关Paxos算法的实现细节，感兴趣的读者可以参考《The Part-Time Parliament》《Revisiting the Paxos Algorithm》《Paxos Made Simple》等文献，本书不再展开介绍。

## 3.1 集群表OSDMap

Monitor仅仅提供了一个基于Paxos实现分布式一致性的一般性框架，实际上Ceph存在各种不同类别的、需要依赖于Monitor进行“集中式”管理的数据，例如集群表、集群级别的统计和告警等，因此Monitor内部与之对应地衍生出了各种不同的类型，它们的具体职责如表3-1所示。

表3-1 Monitor类型及职责

| Monitor 类型 | 职 责 |
| --- | --- |
| AuthMonitor | 负责鉴权和授权，后者又包括密钥的分发与定时刷新等 |

（续）

| Monitor 类型 | 职　责 |
| --- | --- |
| HealthMonitor | 负责监控 Monitor 自身状态，产生相关告警，例如：<br>• 每个 Monitor 进程数据库驻留的磁盘，其空间使用率是否已经达到临界点（例如可用空间小于 30%，或者 Monitor 数据库大小超过 15GB 将会产生告警；进一步地，如果可用空间小于 5%，会导致 Monitor 进程直接退出）。<br>• Monitor 之间时钟是否同步（例如偏差超过 50ms 将会触发告警）。<br>• 是否有 Monitor 宕掉 |
| LogMonitor | 负责按照（管理员配置的）日志策略采集系统日志，并转发至指定的日志频道或者日志服务器 |
| MDSMonitor | 负责监测元数据服务器（MetaData Sever，MDS）的状态变化；处理文件系统（指 CephFS）相关的命令，例如创建文件系统、查看文件系统状态等 |
| OSDMonitor | 负责监控 OSD 状态，维护集群表 |
| PGMonitor | 负责收集 PG 相关的统计[⊖]，处理 PG 相关的命令等。事实上，在早期的实现中，PGMonitor 主要负责创建 PG，现在这一部分工作已被转移至 OSDMonitor 中。此外，随着 Mgr 组件的逐步完善，同时也为了给 Monitor 减负，与统计相关的工作也已转移至 Mgr 组件中 |

注1：也包括OSD级别的统计。

所有类型的Monitor中，OSDMonitor无疑最为重要，它负责守护集群表。

集群表主要由两部分组成：一是集群拓扑结构和用于计算寻址的CRUSH规则，由于两者结合得比较紧密，所以实现上将其纳入一张相对独立的CRUSH map统一进行管理；二是所有OSD的身份和状态信息。事实上，CRUSH map也是围绕OSD构建的，例如OSD是集群拓扑结构中的基本元素、CRUSH的输出是一系列OSD集合等，因此集群表也被称为OSDMap，其主要成员及其含义如表3-2所示。

表3-2　OSDMap主要成员及含义

| 成员 | 含　义 |
| --- | --- |
| blacklist | 客户端黑名单，加入黑名单的客户端无法访问集群 |
| created | 集群第一张 OSDMap 的产生时间，即集群创建时间 |
| crush | CRUSH map |
| epoch | 本 OSDMap 对应的版本号，总是单调递增 |
| erasure_code_profiles | 纠删码模板 |
| flags | 一些集群级别的标志位，用于实现某些特殊功能，例如：<br>• NEARFULL 和 FULL 用于监控集群的空间使用情况[6]；<br>• PAUSERD 和 PAUSEWR 用于禁止客户端读写集群数据；<br>• NOUP、NODOWN、NOIN 和 NOOUT 用于禁止 Monitor 更新 OSD 状态；<br>• NOSCRUB 和 NODEEPSCRUB 用于禁止集群自动执行 Scrub；<br>• NORECOVER 和 NOBACKFILL 用于禁止集群执行 Recovery 和 Backfill；<br>• NOREBALANCE 用于禁止集群自动进行数据重平衡 |

（续）

| 成员 | 含　义 |
| --- | --- |
| fsid | 集群 UUID |
| max_osd | 集群最大 OSD 个数。由于 OSD 从 0 开始编号，所以 max_osd 总是指向下一个尚未分配过的 OSD 编号。<br>此外，为了避免不必要的数据迁移，实现 OSD 无缝替换[⊖]，Monitor 并不会主动回收被删除的 OSD 编号，因此 max_osd 并不总是等于集群当前实际的 OSD 个数 |
| modified | 本 OSDMap 的生成时间 |
| osd_addrs | OSD 地址列表（Monitor 需要收集每个 OSD 包括公共地址、集群地址、心跳地址在内的多个通信地址） |
| osd_info | OSD 状态信息列表，主要包括[⊜]：<br>• down_at：最近一次 OSD 被标记为 Down 的 Epoch；<br>• lost_at：最近一次 OSD 被标记为 Lost 的 Epoch；<br>• up_from：最近一次 OSD 被标记为 Up 的 Epoch；<br>• up_thru：最近一次被标记为 Up 之后（即 up_from），OSD 在 OSDMap 中至少保持该 Up 状态至 up_thru 对应的 Epoch。<br>分析上述 4 个成员的含义我们可知，OSD 在 [up_from, up_thru] 期间一直处于 Up 状态；而最近一次 OSD 宕掉必然发生在 (up_thru, down_at] 之间 |
| osd_primary_affinity | OSD（承载 Primary[⊜]的）亲和性列表，数值越大，对应的 OSD 在执行 PG 映射过程中被选为 Primary 的概率越低 |
| osd_state | OSD 状态列表 |
| osd_uuid | OSD UUID 列表 |
| osd_weight | OSD reweight 列表 |
| osd_xinfo | OSD 扩展属性列表，例如每个 OSD 当前支持哪些特性 |
| pg_temp<br>primary_temp | 当前生效的 PG Temp 条目 |
| pg_upmap<br>pg_upmap_items | 当前生效的 upmap 条目 |
| pools | 集群已创建的存储池列表 |
| pool_name | 集群已创建的存储池名称列表 |
| pool_max | 集群最大存储池个数。由于存储池从 0 开始编号，所以 pool_max 总是指向下一个尚未分配过的存储池编号。<br>同 max_osd，由于 Monitor 从不对已经删除的存储池编号（也称为存储池标识）进行回收，所以 pool_max 并不总是等于集群当前实际的存储池个数 |

注：目前已被粒度更小的、存储池级别的标志代替。

注：由CRUSH相关章节我们知道，此过程中必须保持新老OSD的编号不变，否则必然会引起数据迁移。

注：OSD这些状态的具体含义我们将在后续章节介绍。

注：我们将在PG相关的章节介绍Primary、PG Temp等与PG相关的概念。

除了OSD相关的信息之外，由于Ceph以存储池（而不是OSD或者PG）的方式对外提供存储服务，所以OSDMap还详细记录了用户创建的所有存储池信息。存储池对应的管理结构如表3-3所示。

表3-3　存储池的管理结构

| 成员 | 含　义 |
| --- | --- |
| auid | 存储池拥有者标识 |
| crush_rule | 存储池关联的CRUSH规则 |
| erasure_code_profile | 存储池关联的纠删码模板，仅纠删码类型的存储池有效 |
| expected_num_objects | 存储池预计存储的对象数目。<br>由于FileStore采用多级目录对每个PG存储的对象（实际上是文件）进行管理，为了避免频繁的目录分级操作（也称为目录分裂）导致正常业务出现卡顿的情况，在每个PG存储的对象数量可以预期的前提下（这可以通过expected_num_objects / pg_num进行估算），FileStore可以在创建PG时就提前做好目录分级 |
| flags | 一些存储池级别的标志位，用于实现某些特殊功能，例如：<br>• EC_OVERWRITES：针对纠删码存储池，使能覆盖写；<br>• HASHPSPOOL：在早期Ceph版本的实现中，将PG通过CRUSH映射至OSD时，仅仅使用了每个PG的存储池内部编号，这导致隶属于不同存储池但是编号相同的PG的映射结果都是相同的[8]，因而会产生严重的PG分布不均衡问题。改进的方式是在执行CRUSH映射时，同时使用存储池编号和PG编号。这种方式可以在集群内唯一确定一个PG，因而可以保证每个PG的映射过程都相互独立，进而保证PG的分布相对均衡；<br>• NEARFULL、BACKFILLFULL、FULL和FULL_NO_QUOTA：前面3个标志表示当前存储池的空间使用率已经达到或者超过了一系列预先设定的阈值，其紧迫程度由低到高；FULL_NO_QUOTA表示用户为存储池设置了配额，当前存储池的逻辑已用空间（或者对象）已经达到或者超过配额；<br>• NODELETE、NOPGCHANGE和NOSIZECHANGE：分别表示不允许删除存储池、不允许通过PG分裂的方式改变存储池中的PG数目[9]、不允许修改存储池的size和min_size；<br>• NOSCRUB、NODEEPSCRUB：表示不允许存储池自动执行Scrub或者Deep-Scrub |
| min_size | size、min_size分别指存储池的最大副本数和最小副本数，其中最小副本数指示存储池对于灾难的最大容忍程度。<br>size由用户在创建存储池时指定，min_size则由系统根据size自动计算。<br>不同备份策略下的min_size计算方式如下：<br>• 多副本：min_size = (size + 1) / 2<br>• 纠删码：min_size = k（k的含义请参考纠删码的相关章节） |
| size | |

（续）

| 成员 | 含　义 |
| --- | --- |
| object_hash | 计算对象标识中32位哈希值所使用的哈希方法 |
| opts | 存储池相关的控制选项，例如自动Scrub间隔、后端对象存储引擎采用的压缩模式、压缩算法、启用压缩时期望的压缩率等 |
| pgp_num | PGP数目和PG数目 |
| pg_num | |
| quota_max_bytes | 存储池配额，当前有两种：一种限制客户端能够写入的字节数；另一种限制客户端能够（间接）创建的对象数。<br>需要注意的是，由于从每个OSD上报统计（包括空间、对象数目等）到Monitor汇总、检测并阻止超出配额的写入动作有滞后，所以如果存储池的配额设置得过小，那么会存在配额不能正确生效的情况（例如设置字节配额为GB甚至MB级别，那么客户端实际能够写入的字节数可能会大幅度超出配额）⊖ |
| quota_max_objects | |
| snaps | 存储池级别的快照信息 |
| type | 存储池类型，当前有两种，分别为多副本和纠删码 |

注：这里假定所有存储池共享使用同一条crush_rule。

注：自Mimic版本起，社区正在开发PG合并功能，参见https://github.com/ceph/ceph/pull/20469，这意味着后续也可以通过PG合并来减少存储池中的PG数量。

注：关于存储池相关的空间统计，我们将在下一章进行详细介绍。

由于OSDMap需要精确追踪集群中每个OSD的状态，如果集群规模比较大（显然此时OSD数量也多），那么OSDMap的内容会与之成比例

地增加，这导致OSDMap在编码之后变得十分臃肿，带来显著的传输负担。因此，通常情况下，除了第一张OSDMap以外，后续所有需要传递OSDMap的场景，都可以只传递前后两个OSDMap有差异（即被修改过）的部分。这种增量形式的OSDMap称为Incremental，其内容与OSDMap大同小异，这里不再赘述。

## 3.2 集群管理

由于Monitor可以直接修改OSDMap，如果需要针对集群执行一些管理操作，例如调整集群拓扑结构，创建或者删除存储池等，最快捷的方式是直接通过Monitor提供的CLI命令来完成。我们结合几个典型的应用场景举例进行说明。

### 3.2.1 OSD管理

添加、删除和替换OSD是三类常见的OSD管理操作，分别用于应对集群扩容、缩容和坏盘替换需求。

1.添加OSD

新建一个Ceph集群，或者已有集群需要扩容时，往往会涉及向集群中添加OSD（习惯上我们也称为部署OSD）。随着版本演进，Ceph不断推出新的工具，例如ceph-disk、ceph-volume等，来简化和加速OSD的部署过程，但是本质上仍然是通过这些工具向OSDMonitor发送一系列CLI命令来实现的。这些工具（内部的）详细工作流程如下（这里假定集群名称为“ceph”）：

首先，我们必须向Monitor申请一个集群唯一的OSD编号，作为OSD的身份标识：

```
ceph osd create [{uuid} [{osd-num}]]
```

接下来，我们需要为对应的OSD生成一个工作目录（其中osd-num为上一步返回的OSD编号）：

```
mkdir -p /var/lib/ceph/osd/ceph-{osd-num}
```

如果承载OSD的存储介质是裸设备，则进行格式化，然后将其挂载至上一步创建的工作目录下：

```
mkfs -t {fstype} /dev/{hdd}
mount -o user_xattr /dev/{hdd} /var/lib/ceph/osd/ceph-{osd-num}
```

然后初始化工作目录：

```
ceph-osd -i {osd-num} --mkfs --mkkey
```

注册鉴权密钥（OSD上电后需要先和AuthMonitor完成鉴权）：

```
ceph auth add osd.{osd-num} osd 'allow *' \
    mon 'allow profile osd' \
    mgr 'allow profile osd' \
    -i /var/lib/ceph/osd/ceph-{osd-num}/keyring
```

最后，将该OSD加入CRUSH map并移动至合适位置，这样OSD在上电之后就可以开始分担数据：

```
ceph osd crush add {osd-num} {weight} [{bucket-type}={bucket-
name} ...]
```

当然，如果集群拓扑比较简单，不存在主机以上的自定义层级，最后一步也可以不指定OSD在集群拓扑中的详细位置，这样后续OSD上电时，会自动进行调整。

所有操作成功之后，可以启动OSD：

```
systemctl start ceph-osd@{osd-num}
```

2.删除OSD

删除OSD是添加OSD的反向操作。首先需要停止OSD：

```
systemctl stop ceph-osd@{osd-num}
```

接下来将OSD从CRUSH map中移除：

```
ceph osd crush remove {osd-num}
```

删除鉴权密钥：

```
ceph auth rm osd.{osd-num}
```

注：Luminious版本之前的命令为：ceph auth del osd.{osd-num}

最后从OSDMap中移除OSD（主要是清理状态信息，包括通信地址在内的元数据等）：

```
ceph osd rm {osd-num}
```

如果是Luminous版本，也可以在停止OSD进程之后，直接通过如下一条命令删除OSD：

```
ceph osd purge {osd-num} --yes-i-really-mean-it
```

3.替换OSD

在生产环境中，如果出现由于磁盘损坏导致的OSD故障，为了避免资源浪费，一般需要进行替换而不是直接删除。原理上，替换操作可以通过先删除OSD再添加OSD这样的组合操作间接完成。但是由于修改了CRUSH map，会导致数据先从老的OSD迁出，然后再迁入新的OSD，即出现两次数据迁移，对性能和可靠性造成负面影响。因此Luminous版本对替换OSD操作做了专门优化，可以通过如下方式更加高效地完成。

替换之前如果老的OSD仍在工作，则需要先使其停止工作，之后执行：

```
ceph osd destroy {osd-num} --yes-i-really-mean-it
```

如果不是使用全新的磁盘进行替换，最好先对其进行格式化：

```
ceph-volume lvm zap /dev/sd{X}
```

最后创建并启动新的OSD即可（注意下面的osd-num需要替换为老的OSD编号）：

```
ceph-volume lvm create --osd-id {osd-num} --data /dev/sd{X}
```

### 3.2.2 存储池管理

存储池管理操作主要包括创建和删除两种。除此之外，还可以根据需要修改表3-3中存储池的（大部分）属性。

1.创建存储池

创建一个存储池的命令如下：

```
ceph osd pool create <poolname><pg_num> {<pgp_num>} \
    {replicated|erasure} {<erasure_code_profile>} \
    {<crush_rule_name>} {<expected_num_objects>}
```

上述命令中，除了存储池名称和PG数量必须指定之外，其余参数（其具体含义参见表3-3）可选。如果都不设置，则会创建一个使用默认CRUSH规则、3副本的存储池。

由于默认CRUSH规则会包含集群中所有OSD，这在生产环境中可能会导致一些潜在问题，例如无法满足用户对于冷热数据进行分级存储的需求，一旦某个或者某些OSD故障，则所有使用该规则的存储池都将受到不同程度的影响等。所以在创建存储池时，最好为其指定独立的CRUSH规则。

在大型异构集群中，为了降低成本，最常见的需求是为不同用途的存储池分配不同类型的设备，例如为承载数据库等时延敏感业务的

存储池分配高性能的存储设备（例如SSD），为承载备份数据的存储池分配性能一般的大容量存储设备（例如SATA HDD）等。当然，由于支持强一致性语义导致Ceph的分布式写严重受到木桶原理制约，所以为了提升性能（特别是写性能），任何时候限制同一个存储池尽量只使用规格相同、批次也相同的设备总是不错的选择。

参考CRUSH相关章节，上述需求固然可以通过直接修改CRUSH map予以满足，但问题是对普通用户而言难度太大，并且由于CRUSH map影响整个RADOS集群和所有客户端，在生产环境中直接操作CRUSH map风险也大。因此，在Luminous版本中，为了（更友好地）解决上述问题，Ceph为每个OSD根据其存储介质类型自动绑定了一个标签，例如HDD、SSD和NVMe SSD等。这样后续创建CRUSH规则时，通过（额外）指定标签，即可限制对应的CRUSH规则仅选择带有这种标签的设备。例如，下面这条命令创建了一条只选择SSD类型OSD的CRUSH规则（适用于多副本存储池）：

```
ceph osd crush rule create-replicated ssd_rule default host ssd
```

当然，如果理解这个标签仅仅是对OSD进行分类而不是将OSD与其后端存储介质类型强制绑定，也可以更进一步地通过标签来过滤出任何想要的OSD供存储池使用。

```
ceph osd crush rm-device-class all
ceph osd crush set-device-class foo 0 9 18
ceph osd crush rule create-replicated foo_rule default host foo
```

上述一系列命令最终将创建一条名为foo_rule的CRUSH规则，与之绑定的存储池将被限制只能使用编号为0、9和18的OSD，而不考虑其背后的存储介质类型究竟为何。

既然可以为OSD绑定标签，一个自然的拓展是为存储池也绑定标签，后者的意义在于，在生产环境中，一般而言，规模较大的Ceph集群都是多用途的（Ceph是统一存储平台），这意味着集群当中可能存在多个存储池，此时为不同类型的存储池绑定不同的标签可以很好地起到区分其用途的作用。例如，Ceph会自动将名为“backups”“images”“vms”“volumes”或者名称中包含了“rbd”的存储池打上“rbd”标签；将名称中包含了“.rgw”“.intent_log”“.log”“.usage”“.users”的存储池打上“rgw”标签等。当然，如果存储池没有自动绑定标签，或者为多用途，也可以通过如下命令来增加标签（如果需要绑定多个标签，则需要额外输入“--yes-i-really-mean-it”参数）：

```
ceph osd pool application enable <poolname> <app> {--yes-i-really-mean-it}
```

多余的标签可以通过如下方式删除：

```
ceph osd pool application disable <poolname> <app> {--yes-i-really-mean-it}
```

除了标签之外，Ceph进一步为每个存储池开放了存储少量自定义属性对的接口：

```
ceph osd pool application set <poolname> <app> <key> <value>
ceph osd pool application rm <poolname> <app> <key>
ceph osd pool application get {<poolname>} {<app>} {<key>}
```

需要注意的是，由于这些属性对都随存储池信息保存在OSDMap之中，为了避免用户写入大量属性对而导致OSDMap过分臃肿，Ceph

对每个存储池能够存储的自定义属性对都做了硬限制，例如键值对总数不能超过64、单个键或者值长度不超过128个字节等。

当存储池创建成功之后，Monitor将在后台触发PG创建。由于PG的创建是异步的，如果客户端在创建完存储池之后立即进行操作（例如写入数据），对应的请求有可能会被挂起，直至PG创建成功之后才能得到响应。

2.删除存储池

如果对应的存储池不再使用，可以通过如下命令删除，以回收其占用的存储空间：

```
ceph osd pool delete <poolname> <poolname> <--yes-i-really-really-mean-it>
```

与创建存储池类似，此时Monitor将在后台通知OSD进行PG删除。由于PG的删除也是异步的，因此对应存储池占用的存储空间不是被立即释放，而是需要一个过程。

特别地，出于安全考虑，Ceph为存储池删除操作增加了一种回收站机制：当该机制使能时，如果管理员发起存储池操作，Monitor不是真正删除，而是简单将其重命名，这样后续如果发现是误删，还可以通过反向重命名来找回原先的存储池。

3.修改存储池属性

表3-3中的大部分属性都可以通过如下命令进行修改（命令会自动提示存储池的哪些属性允许修改）：

```
ceph osd pool set <poolname> <key> {<value>}
```

可以通过如下命令确认修改是否生效：

```
ceph osd pool get <poolname> <key>
```

特别地，使用关键字all可以打印存储池的所有属性对：

```
ceph osd pool get <poolname> all
```

### 3.2.3 告警管理

除了对OSD和存储池进行管理之外，Monitor还需要实时监控集群的运行状态，针对异常情况产生告警，以便管理员识别和及时排除潜在的风险。

最常见和最重要的两类告警分别是OSD状态告警和PG状态告警，其中OSD状态告警主要包括：

·OSD宕掉

·OSD设置了某些特殊标志，例如NOUP/NODOWN/NOIN/NOOUT等

·OSD的空间使用率突破了某些阈值，同时会导致对应的存储池也产生相关告警

·OSD出现Slow Request等

PG状态告警主要包括：

·PG降级

·PG宕掉

·PG出现数据损坏

·PG状态未知等

这两类告警分别由OSDMonitor和PGMonitor负责监控和上报。

除此之外，还存在一些与配置项相关的告警（例如，与空间管理相关的配置项乱序、存在一些老版本的CRUSH参数）和Monitor自身相关的告警等。由于这些告警在生产环境中不太常见，这里不做进一步介绍。

## 3.3 总结和展望

Monitor负责为RADOS集群提供包括鉴权、授权、设备（OSD、主机等）管理、日志管理、告警管理等在内的一众集群管理服务。同时，通过与集群中所有OSD建立联系并周期性地交换和传播OSD状态信息，Monitor能够提供可信赖和可扩展的集群状态监控服务。为了避免单点故障，也为了避免在集群规模较大时存在明显的处理瓶颈，Monitor采用多活（负荷分担）方式进行工作，并基于Paxos算法保障自身的高可靠性和分布式一致性。

Monitor提供了丰富的CLI命令，通过这些CLI命令可以结合集群本身的特征对集群进行深度定制，从而最大化资源利用率，节省成本。但是由于命令数量众多，命令之间格式不太统一（例如针对常见的删除操作，不同的CLI命令可能分别使用delete、del、remove、rm等充当关键字），同时很多命令专业性和关联性都比较强。因此，对普通用户而言，记忆和使用这些命令可能存在较大的难度，这反过来会影响Ceph的使用与推广。

Monitor的告警系统整体上设计得比较简单，例如不支持告警定制。告警与系统本身的耦合性也比较强，例如正文提到的基础告警中，除了空间使用情况之外，Monitor并没有单独针对存储池状态的告警，一个可能的原因是，在Ceph的设计哲学中，存储池仅仅是个抽象

概念，而存储池的真实载体和基本管理单位其实是PG，因此Ceph认为提供PG级别的告警对于诊断和分析问题更有帮助。但是对于普通用户，特别是从传统存储转投Ceph的用户而言，由于PG是Ceph的内部概念，他们无法也无从去了解PG以及PG相关的告警，所以大多数情况下也就无法做出恰当的反馈和应对。事实上，由Ceph基于哈希随机分布数据的特点，只要任意一个PG不正常，那么对应的症状就会反馈至其归属的存储池这个整体。例如，存储池中任意一个PG宕掉，那么有可能存储池承载的所有业务都会发生阻塞。因此，从这个角度来看，将PG状态告警转化为对应存储池的状态告警可能更为合理，也更具通用性。

除此之外，由于需要周期性地收集和更新集群中所有OSD的状态，如果集群规模比较大，或者集群状态不稳定，采用Paxos这种重量级的分布式一致性协议会使得Monitor自身存在明显的处理瓶颈，这反过来又会影响集群对外业务的连续性和稳定性。为此，社区特地引入了一个全新的Mgr组件，用于分担统计、告警等不需要修改OSDMap的任务，来为Monitor减负。

第4章

# 存储的基石——OSD

对象存储起源于传统的NAS（例如NFS）和SAN存储，其基本思想是赋予底层物理存储设备（例如磁盘）一些CPU、内存资源等，使之成为一个抽象的对象存储设备（即OSD），能够独立完成一些低级别的文件系统操作（例如空间分配、磁盘I/O调度等），以实现客户端I/O操作（例如读、写）与系统调用（例如打开、关闭文件）之间的解耦。

和传统对象存储仅仅赋予OSD一些初级的“智能”不同，Ceph开创性地认为这种“智能”可以被更进一步地用于执行故障恢复与数据自动平衡、提供完备的高性能本地对象存储服务等复杂任务上，从而使得基于OSD构建高可靠、高可扩展和高并发性能的大型分布式对象存储系统成为可能。

OSD的“智能”主要体现在以下几个方面：

·资源定量控制—与传统对象存储不同，在Ceph中，每个OSD可以对其消耗的CPU、内存、网络带宽等资源进行精确控制。在基于通用服务器构建大规模存储集群的应用场景中，这种能力可以有效避免资源浪费，显著降低成本。

·实例化的对象存储—OSD进一步完善和实例化了对象这个传统存储中的抽象概念，定义了一整套完备和具有强一致性语义的对象操作接口（Application Programming Interface，API）。再加上客户端与服务端解耦并且允许客户端和OSD直接通信的设计，理论上基于对象可以开发任意类型的存储应用。

·自管理和自学习—每个OSD高度自治，数据复制、数据恢复、数据迁移等都由OSD自主进行而无须中心控制器干预；OSD之间互相监督，确保故障能够被及时捕获和上报至Monitor；通过OSD与客户端、以及OSD之间相互学习和点对点传播OSDMap，Ceph能够快速进行故障切换和恢复，并最大程度保证对外提供不中断的存储服务。

本章简要介绍OSD的组成、OSD的上电流程、基于OSD的故障检测和空间管理等，关于OSD“智能”的更多细节，我们将保留到PG相关的章节再进行详细介绍。

## 4.1 OSD概述

OSD是RADOS集群的基本存储单元，每个OSD皆可提供完备和具有强一致性语义的本地对象存储服务。

通常情况下，单个OSD一般仅用于管理单一的本地物理存储设备，并且在其活跃周期内，需要占用一定的CPU、内存和网络带宽资源，例如Ceph官网建议单个OSD的典型资源配置如表4-1所示。

表4-1 单个OSD建议的典型资源配置

| 资源类型 | 建议值 |
| --- | --- |
| CPU | 1个虚核 |
| 内存 | 与OSD管理的本地存储介质裸容量相关，计算方式为：裸容量每增加1TB，则内存相应地增加1GB；如果采用BlueStore作为后端对象存储引擎，则单个OSD的可用内存至少3GB |
| 网络 | 取决于存储节点承载的OSD个数；10盘位以上的存储服务器推荐公共和集群网络都采用万兆网络，并且集群网络配置最好优于公共网络配置；如果采用全SSD或者NVMe SSD阵列，则网络需求更高 |

在Luminous版本之前，OSD并不直接操作物理存储设备（习惯上称为裸设备），而是将其交给操作系统自带的本地文件系统进行管

理。因此，为了实现对象与文件之间的转义，也为了屏蔽不同本地文件系统之间的差异，OSD内部抽象出了一个ObjectStore层，由其负责实施具体的对象存储事务。ObjectStore当前有两种具体实现，分别是FileStore和BlueStore。顾名思义，FileStore需要在对象和文件之间来回转换，过程烦琐并且效率低下。从Luminous版本开始已经逐渐被直接以对象方式操作裸设备的BlueStore所取代。

除此之外，OSD还包含一些其他类型的数据成员（组件），它们和ObjectStore一起，帮助OSD提供完整并且越来越精细化的本地对象存储服务。

### 4.1.1 集群管理

为了和OSD直接通信，首先，客户端必须能够通过OSDMap获得诸如集群当前数据分布形态、OSD通信地址等在内的关键信息。因此，每个OSD上电时，必须向Monitor上报包括自身通信地址在内的元数据。此后，伴随着新的OSDMap在集群中传播，重新上电的OSD与客户端，以及其他OSD之间的通信可以逐渐恢复。

其次，由于RADOS集群不存在除Monitor之外的集中点，为了方便用户获取诸如集群裸容量、已用空间在内的一些概要信息，用于监控集群状态和排除潜在风险（例如集群存储容量即将耗尽时通知用户及时进行扩容），也必须通过所有OSD周期性地向Monitor上报统计数据来实现。

最后，无论是基于数据安全设计的鉴权和密钥定时刷新机制，还是OSD自身的数据恢复、重平衡流程等，都不可避免地涉及与Monitor交互。

因此，每个OSD驻留了一个Monitor的客户端组件（MonClient），负责与Monitor通信。专门引入MonClient（而不是采用一个通用通信组件）的必要性在于：在实践中，出于高可靠性考虑，集群中的Monitor往往不止一个，而是基于Paxos分布式一致性算法构建的小型集群。一些异常情况下，例如某个Monitor宕掉，可能导

致OSD需要重连。MonClient隐藏了这些复杂的实现细节，提供与Monitor通信的统一接口。

### 4.1.2 网络通信

网络通信组件——Messenger允许OSD采用各种流行的网络通信协议，例如TCP/IP、Infiniband、RDMA（Remote Direct Memory Access）等，与客户端或者其他OSD通信。

在设计上，Ceph将整个RADOS集群网络分为两个独立的平面，即公共网络平面和集群网络平面。公共网络，顾名思义，用于客户端与集群进行通信。由于客户端必须通过Monitor才能获得OSDMap，所以Monitor也必须暴露在公共网络之中。集群网络则用于OSD之间进行通信。RADOS集群的网络平面如图4-1所示。

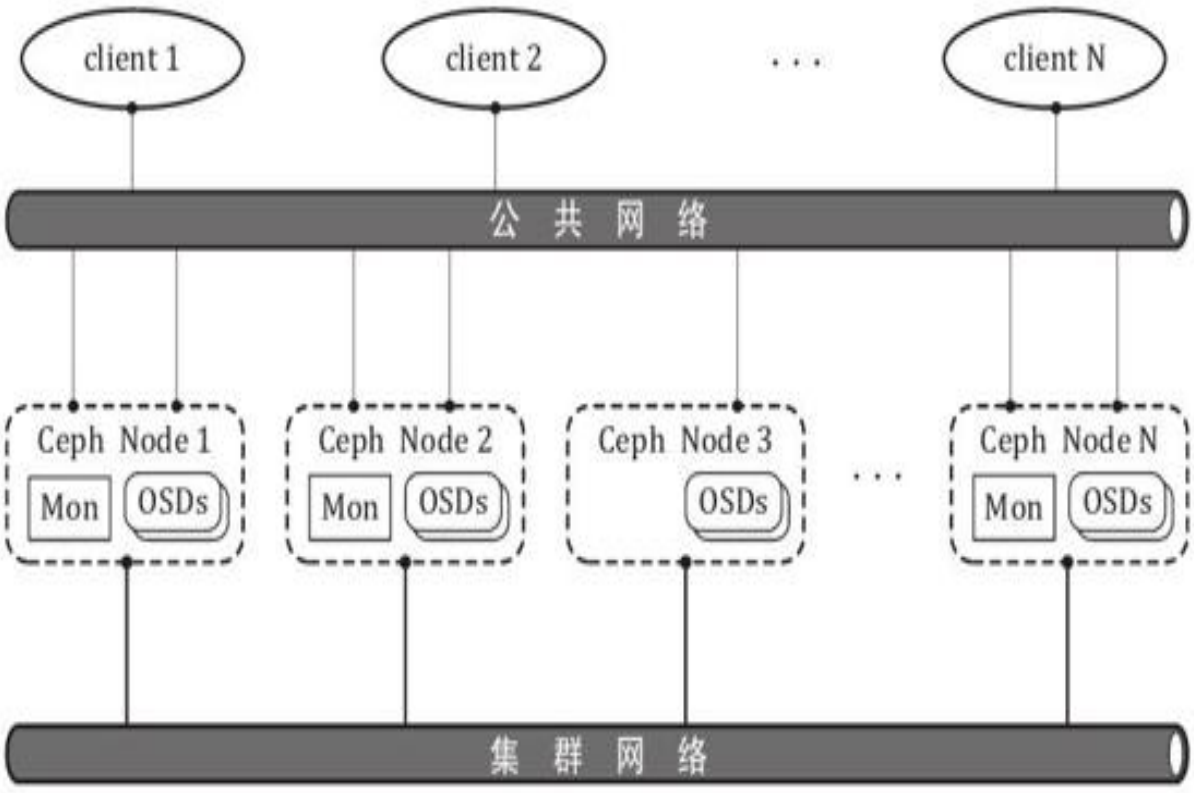



图4-1　RADOS集群的网络平面

原理上，OSD之间也可以复用公共网络进行通信。那为什么在设计上要将这两个网络平面分开呢？

首先，两者传输的流量并不对等。以典型的3副本为例，OSD每处理一次客户端的写操作，都要复制两份相同的数据到其他两个不同的OSD，所以此时每个OSD的输入（从客户端到这个OSD）和输出（从这个OSD到其他两个副本所在的OSD）流量之比为1：2。除此之外，由于集群内部还有大量诸如数据恢复、数据自动平衡等后台任务导致的、需要跨OSD进行传输的流量，因此总体上集群网络的负载要远远高于公共网络。

其次，两者承载的业务性质不同。如果强行将两者合二为一，由于后台任务大多是（磁盘和网络）流量密集型的，很可能会大量抢占

带宽，导致优先级最高的客户端业务所需的带宽得不到保障。相反，如果从物理上就将两者进行隔离，那么单纯从网络方面可以避免它们承载的业务相互之间产生干扰。

因此，实现上每个OSD都需要指定独立的公共网络地址和集群网络地址，并与对应的Messenger绑定。需要注意的是，虽然每个地址只绑定一个Messenger，但是每个Messenger却可以同时维护多条链路，例如多个客户端可以同时与某个OSD通过公共网络进行通信，每个OSD也可以通过集群网络和多个伙伴OSD进行通信等。在老的SimpleMessenger实现中，由于会为每条链路固定分配多达4个服务线程，因此在一些极端场景下可能会出现系统文件句柄耗尽的情况。

除此之外，每个OSD还有一些其他类型的Messenger，分别负责心跳检测和Cache-Tier相关的功能等，详情如表4-2所示。

表4-2　OSD中驻留的Messenger及其功能

| Messenger | 功　能 |
| --- | --- |
| public | 主要用于客户端与OSD之间进行通信 |
| cluster | 主要用于OSD之间进行通信 |
| heartbeat | 主要用于OSD之间的通信链路检测<br>由于心跳检测报文优先级最高，在设计上，为了防止其他类型的报文产生干扰，为heartbeat分配了独立的Messenger，即将heartbeat和其他任意类型的通信服务进行了隔离<br>此外，为了保证能够侦测到任意网络平面的故障，OSD之间同时通过公共网络和集群网络进行heartbeat |
| objecter | 主要用于实现Cache-Tier相关的、跨存储池之间的报文转发 |

### 4.1.3 公共服务

OSD内部驻留的公共服务组件主要有OSDService、各种类型的线程池、定时器等，下面分别予以介绍。

顾名思义，OSDService组件用于提供OSD级别的服务。由于OSD是PG的载体，所以OSDService的服务对象也主要是PG，可提供的服务如下：

·传播和发布OSDMap

·统一进行消息收发（包括与Monitor通信等）

·提供全局的本地对象存储引擎（ObjectStore）

·透传OSD本身的状态（例如OSD是否准备下电、OSD对应的本地存储容量使用情况等）

·针对一些PG级别的任务，例如Recovery、Scrub等，提供统一的资源分配和调度机制（方便实施任务优先级管理）

线程池则提供了一种供不同任务分时间片使用一些公共线程的通用方法。任何任务想要使用线程池中的线程资源时，必须先实例化线程池所支持的、某种抽象类型的工作队列，定义该任务的具体执行方

式（例如入队、出队方法，真正的处理函数等），然后再将工作队列与对应的线程池进行绑定。

特别地，为了防止产生死锁或者死循环，每种工作队列都要定义其任务执行过程中能够连续使用线程的最大时间片，如果超时，则将导致OSD进程进入休眠模式或者直接退出。

在Mimic版本之前，Ceph按照不同任务类型设计了多个线程池，例如专门用于处理Peering的线程池、专门用于删除PG的线程池、专门用于处理用户命令的线程池和处理所有其他任务的线程池。除了造成资源浪费之外（单个OSD占用的线程数过多一直为人诟病），这种处理方式最大的缺点在于弱化了这些任务之间的关联性，由于没有将所有任务都纳入统一的优先级度量体系，在OSD能够使用的资源严格受控的前提下，可能无法使得OSD的整体效率和性能达到最优。

意识到上面这个问题之后，Mimic版本对线程池进行了精简，仅保留了后面两个线程池，并将除用户命令之外的所有任务都放到同一个线程池中去进行处理。

最后，定时器用于处理OSD内部一些需要周期性执行的任务，例如心跳检测、心跳伙伴刷新、Scrub调度等。值得一提的是，在较老的版本中，每个OSD进程仅仅配备了一个定时器，并且该定时器工作时需要全程持有全局的osd_lock，因此在业务比较繁忙时容易造成定时器被osd_lock长时间阻塞，从而产生误检和误判。为此，后来（Jewel版本之后）社区将所有不需要osd_lock保护的任务转移至另一个定时器中进行。

## 4.2 OSD上电

前面我们介绍了OSD的基本概念，本节简要介绍OSD的上电流程。由于Ceph通过OSDMap维护和传播所有OSD的状态信息，所以OSD的上电过程中需要和Monitor进行（多次）交互。

与启动操作系统类似，OSD正式上电之前，也需要预先读取一些引导数据（bootstrap），用于身份校验等。容易理解，这些引导数据不能直接由OSD自身的ObjectStore接管。为此，我们创建OSD时，总是会预留少量存储空间（默认为100MB），并使用操作系统自带的本地文件系统（例如XFS）格式化，用于保存和访问这部分引导数据。这些引导数据的作用如表4-3所示。

表4-3　OSD相关的引导数据作用

| 引导数据类型 | 作　用 |
| --- | --- |
| magic | 校验数据，用于验证该OSD能否被当前的Ceph软件版本所识别和接管，也用于验证这些引导数据是否已经损坏 |
| whoami | OSD在集群中数字标识，参考CRUSH相关的章节，这是一个从0开始编号的整数，全集群唯一，可以直接作为OSD的身份标识 |
| ceph_fsid | 指示OSD归属集群的UUID |
| fsid | OSD自身的UUID |

此后，通过挂载ObjectStore，OSD可以读取自身保存的超级块。除了与引导数据中重叠的身份信息之外，超级块中还额外包含如下重要信息：

·当前磁盘数据支持的特性集

·OSD自身保存的OSDMap范围

·基于OSD管理的磁盘裸容量计算得到的权重等

如果身份验证失败（例如超级块中保存的whoami与引导数据中的不符），或者磁盘数据格式太老无法进行平滑升级，则终止启动过程；否则可以通过ObjectStore继续加载所有PG。

如果一切正常，OSD将初始化一个MonClient实例，通过它向Monitor发起鉴权并尝试获取需要定时刷新的会话密钥。通过鉴权并成功获取会话密钥之后，按照用户配置的策略，OSD可能还需要通知Monitor刷新自身在集群拓扑中的位置。

如果前一次下电之前，还有PG没有与OSD完成OSDMap同步，此时OSD可以通知这些PG继续执行同步。之后OSD将向Monitor发送一个

GetVersion消息，用于索取Monitor当前所拥有的OSDMap范围。

收到Monitor的回应消息之后，如果OSD检测到通过一次传输即可完成自身和Monitor之间的OSDMap同步（例如两者保存的最新OSDMap版本号没有相差太多）并且满足启动条件（例如集群没有设置NOUP标记），则可以直接向Monitor发送Boot消息。

Monitor收到Boot消息之后，如果判定对应的OSD具备正常启动条件，则将该OSD在下一个OSDMap标记为Up。等待下一个OSDMap正式生效后，Monitor再将所有缺失的OSDMap一次性地通过一个MOSDMap消息返回给对应的OSD。后者收到此消息后，首先通过ObjectStore将这些OSDMap固化到本地，然后将自身状态从booting修改为active，最后通知所有PG分批次、以异步的方式完成这些OSDMap同步。

至此，OSD上电流程即宣告完成。

## 4.3 故障检测

当规模上升至PB级别之后，集群中一般会包含大量OSD，此时OSD故障必然成为一种常态。作为保障对外业务不中断的重要前提，及时和完善的故障检测变得尤为重要。

为了区分故障类型，例如是临时性故障（例如主机重启）还是永久性故障（例如磁盘损坏），方便选择合理的数据恢复策略，Ceph为每个OSD赋予了4种外部可见的状态，如表4-4所示。

表4-4　OSD的4种状态

| 状态 | 含　义 |
| --- | --- |
| Up | 正常状态 |
| Down | 异常状态（对应临时性故障，不会触发PG迁移） |
| In | Out状态的对立面 |
| Out | Monitor如果检测到某个OSD处于Down状态超过一定时间（例如600s），会将其进一步地设置为Out（对应永久性故障）。参见CRUSH相关的章节，处于Out状态的OSD，其在CRUSH的选择过程中会被自然淘汰，这意味着这种类型的OSD无法再承载PG，它承载的数据（PG）会被及时地迁移至其他（状态为Up和In的）OSD |

如前所述，OSD故障必须记录到OSDMap中，才能伴随着OSDMap的传播被客户端和其他OSD感知并采取应对措施。因此，所有类型的故障最终都必须向Monitor汇报。

当前，Monitor一共可通过如下3种途径检测到OSD故障（或者下电）：

1）OSD自主上报，一般对应优雅下电。

2）投票方式，OSD之间通过周期性的心跳检测，监控彼此之间的状态，超过一定时间没有收到某个伙伴OSD的应答消息，则持续向Monitor上报该OSD失联消息（即投票）。满足如下条件之一，则Monitor会将候选OSD标记为Down：

·投票携带了Force/Immediate标识

·候选OSD累积的有效票数已经达到阈值（默认为2）

3）看门狗机制，每个OSD需要周期性（默认为300s）地向Monitor发送Beacon消息进行保活，如果Monitor在一段时间内（默认为900s）没有收到过某个OSD的任何Beacon消息，则将该OSD标记为Down。

上述3种检测方式中，1）比2）可靠，2）又比3）可靠，同时1）的效率和效果也是三者之中最佳的。因此，在任何情况下，只要可能，我们总是倾向于让OSD优雅下电。然而，如果OSD自身发生异常，则只能依赖方式2）和3），其中2）又是相对快捷的方式。

方式2）依赖于OSD之间如何通过心跳建立联系，即OSD如何选择有效的伙伴OSD。最粗糙的方式莫过于让每个OSD以广播的方式直接与所有其他OSD进行心跳检测。这种方式的弊端显而易见：一方面故障具有突发性，所以心跳检测需要周期性、不间断地进行；另一方面为了尽早捕获异常，我们倾向于使用比较短的心跳周期，如果一个集

群存在大量OSD，采用广播方式很容易引起心跳风暴。因此，每个OSD最好能够尽量选择一组完备、同时又尽可能小的伙伴OSD集合进行心跳检测，最终所有OSD之间形成一个类似蛛网的心跳检测网络，覆盖整个集群。

考虑到PG是OSD之间发生联系的桥梁，一个自然的想法是将每个OSD承载的PG关联的所有OSD都作为自己的心跳伙伴。同时，为了避免一些极端情况下某个OSD的伙伴OSD分布不够理想，还必须要按照一定的法则选择一些额外的OSD。完整的心跳伙伴选择过程简述如下：

1）选择每个PG的Up与Acting中的OSD。

2）选择在编号上与本OSD相邻的前一个和后一个状态为Up的OSD。

3）如果该OSD的心跳伙伴个数小于最小值（默认为10），则以本OSD的编号（即whoami）作为基准，依次选择集群中下一个状态为Up状态、编号相邻的OSD，直至达到最小值。

与为了提升数据的可靠性必须要指定更高级别的故障域类似，为了保证每个OSD的异常状态能够被尽可能多的伙伴OSD捕获并上报给Monitor，同时产生尽可能多的有效投票供Monitor以最快速度产生结论，伙伴OSD的选择原则上应该尽可能地覆盖集群拓扑中每个大的（例如故障域）分支。以一个包含3个主机的典配集群为例，每个OSD的伙伴OSD应该尽量包含来自其他两个主机的OSD，这样才能保证该主机下电后，其承载的OSD能够被Monitor迅速设置为Down状态。在生产实践中，为了提升速度，我们一般会采用并发的方式同时对多个主机进行部署，因此如果同一个主机上存在多个OSD，其编号一般都是不连续的，反之编号连续的OSD一般分布在不同的主机之上，这就解释了我们在选择心跳伙伴时，为什么总是会无条件地选择“在编号上与

本OSD相邻的前一个和后一个状态为Up的OSD”。然而遗憾的是，实际情况并非总是如此。

最后，需要注意的是，某个OSD被设置为Down状态只是简单触发其承载的PG通过Peering进行Primary切换（如果需要），以尽快恢复对外业务（此时PG工作在降级状态），只有当某个OSD处于Down状态足够长时间之后（例如超过600s），才有理由认为对应的OSD已经完全无法修复。此时出于数据可靠性考虑，Monitor会将其进一步地设置为Out，通知所有受波及的PG正式开始数据迁移。一些特殊情况下，例如整个主机、甚至整个机架的OSD都处于Down状态，或者整个集群大量（例如超过四分之一）OSD都处于Down状态，为了避免产生数据迁移风暴，Monitor会自动进行识别，仅仅转化合适数量并且满足约束条件的、处于Down状态的OSD为Out。

## 4.4 空间管理

reweight、weight-set、upmap以及能够自动进行集群空间使用率平衡的balancer等一系列机制的引入，无疑让我们看到了社区提升Ceph空间利用率、降低空间成本的决心，这反过来也足以说明空间管理之于存储系统、特别是分布式存储系统的重要性。

视紧迫程度不同，OSD将本地存储空间的使用情况分成4个等级，如表4-5所示。

表4-5　OSD空间使用率等级及其含义

（从上往下，紧迫程度由低到高）

| 等级名称 | 含　义 |
| --- | --- |
| Nearfull | 产生告警，除此之外无任何实质性影响 |
| Backfillfull | 产生告警，拒绝PG通过Backfill方式迁入或者继续迁入本OSD |
| Full | 产生告警，所有使用该OSD的存储池，对应客户端的写入操作将被有条件禁止[注] |
| Failsafefull | 产生告警，所有使用该OSD的存储池，对应客户端的写入操作将被无条件禁止 |

注：写入操作携带了CEPH_OSD_FLAG_FULL_FORCE和CEPH_OSD_FLAG_FULL_TRY标识的除外。

注：实际上没有Failsafefull这一告警类型，但是因为Failsafefull的门槛比Full高，此时必然已经产生了Full告警。

注：写入操作携带了CEPH_OSD_FLAG_FULL_TRY标识的除外。

每个OSD通过定时器周期性地检测自身的空间使用率和状态，并上报至Monitor。如果有OSD处于Full状态，那么Monitor会将对应的存储池也标记为Full，从而阻止该存储池承载的所有客户端后续的写操作。

引入Backfillfull这个等级的意义在于，有些数据迁移，例如数据恢复或者自动平衡过程中以Backfill方式进行的PG整体数据迁移，是集群内部自动触发的，它们并不受存储池的Full标记影响。

进一步地，如果Full设置得过高（默认为OSD管理的主要存储设备裸容量的95%），由于从OSD上报空间使用统计和状态，到Monitor真正将对应的存储池标记为Full并阻止客户端继续写入有滞后，所以仍然存在在此期间客户端继续产生大量写请求将OSD彻底写满的可能，此时Failsafefull可以作为防止产生OSD永久性地变成只读这类灾难性后果的最后一道屏障。

基于每个OSD上报的空间统计，Monitor可以汇总并计算集群以及各个存储池的空间分布情况。集群的比较简单，只需要按照空间统计类别逐个OSD进行累积即可。至于存储池，由于存在多个存储池共享使用全部或部分OSD的可能，情况比较复杂，我们接下来进行详细说明。

针对存储池，Ceph当前一共有如下3组空间度量指标：

·已用空间

·最大可用空间

·总空间（=已用空间+最大可用空间）

简言之，存储池的已用空间指其承载的所有应用程序数据占用的逻辑空间之和。逻辑空间意味着不需要考虑Ceph本身的备份策略、空间预留、元数据开销等，例如客户端累计写入了1MB数据，那么存储池的已用空间就统计为1MB，而实际上OSD为了存储这1MB数据可能分配了超过3MB的物理存储空间。

由上述分析可见，存储池的已用空间采用了与文件系统类似的统计方式，但是Ceph的实现又稍有不同，对于空间的使用，两者都有一个最小粒度要求，例如文件系统为4kB，而Ceph为一个对象，即默认情况下为4MB。由于Ceph的最小粒度比文件系统大了1024倍，所以Ceph的存储池已用空间统计更加粗糙，某些场景下（典型的如小块随机写）可能会存在统计结果和真实数据相差甚远的情况。

那么什么时候存储池的已用空间统计是准确的呢？容易理解，这要求每个对象都被真实数据写满，即不能存在稀疏写（即对象中不存在空穴），所以通常存在大量连续写或者重复多次随机写（直至每个对象都被写满）的场景下，存储池的已用空间统计会比较准确。

至于存储池的最大可用空间，其计算过程更加复杂，简要描述如下：

1）找到与存储池对应的CRUSH规则，遍历其关联的OSD，并分别计算每个OSD的CRUSH权重在其中所占的比重，例如该CRUSH规则关联了3个OSD，容量分别为1TB、2TB、3TB，那么最终各自的比重为1/6、2/6、3/6。

2）分别计算每个OSD的可用物理空间（具体由ObjectStore负责统计和上报），将其扣除预留空间（默认为OSD裸容量的5%），然后将

其除以步骤1）中计算得到的比重，最后再除以对应存储池的副本数，得到一个基于本OSD估算的、存储池的最大可用空间。

3）找出步骤2）中所有存储池最大可用空间中的最小值，作为该存储池的实时最大可用空间。

存储池的最大可用空间为什么采用上面这种“奇葩”的计算方式呢？这主要是基于以下考虑：

1）存储池能够使用哪些OSD，必须受对应CRUSH规则的约束。

2）由于不同存储池可以采用精简配置的方式来共享使用OSD的本地存储空间，这意味着这些存储池之间都是竞争关系，因此采用上述方式得到的可用空间，仅仅表明在计算完成的瞬间至多可供每个存储池使用的存储空间，这也是“最大”前缀的含义。

3）计算过程中，之所以选择逐个OSD进行反推，并选择其中的最小值，原因我们在前面已经提到过多次：由于Ceph随机分布数据的特点，只要存储池中任意一个OSD写满，对应的存储池就会被标记为Full并禁止客户端写入，因此一个存储池的最终空间使用率实际上只取决于池中空间使用率最高的那个OSD。

同样，由于Ceph随机分布数据的特点，随着数据持续写入，一旦某个采样周期内，隶属于该存储池的不同OSD已用空间变化差距过大，则可能导致前后两个采样周期内用于计算最大可用空间的基准OSD（参见步骤3））发生变化，进而导致存储池的最大可用空间统计发生跳变。我们以一个简单的例子进行说明：

1）假定某个2副本存储池仅包含两个OSD：OSD.0和OSD.1，裸容量都为1TB。

2）$t_1$时刻，两个OSD的已用空间都为0，此时：

存储池已用空间=0

存储池的最大可用空间=0.95TB

存储池总空间=0.95TB

3）$t_2$时刻，OSD.0已用空间为10%，OSD.1已用空间为20%，此时：

存储池已用空间=0.15TB

存储池的最大可用空间=((1-20%-5%)/0.5)/2=0.75TB

存储池总空间=0.15TB+0.75TB=0.90TB

可见，采用上面这种统计方式时，如果数据在OSD之间分布不是绝对均匀，那么必然会存在一部分空间泄漏——即既没有统计到存储池的已用空间中，也没有统计到存储池的最大可用空间中。例如，上面这个例子中，我们在计算最大可用空间时，以OSD.1作为参照物，假想OSD.0也使用了20%的物理容量（即OSD.0被平均了），导致与$t_1$时刻相比，$t_2$时刻的存储池总空间缩水了0.05TB。

综上，因为存储池的已用空间和最大可用空间计算过程各自存在缺陷，所以基于这两个数值相加得到的存储池总空间自然也不可能准确，并且这种偏差在OSD空间使用率不均衡的情况下更容易被放大。这就从另外一个角度解释了为何社区要不遗余力地解决数据分布不均衡的问题。

## 4.5 总结和展望

在Ceph超过10年的发展历程中，主流存储介质已经经历了从机械磁盘（HDD），到固态存储设备（SSD、NVMe SSD），再到今天非易失性内存设备（例如Optane）的变迁。引入OSD这个抽象概念的初衷在于屏蔽不同类型操作系统及底层驱动接口的实现细节，更好地适应底层存储介质的不断进化，从而使得基于Ceph可以为客户（包括其他依赖Ceph提供存储服务的下游开源项目，例如OpenStack等）提供长期、稳定和可信赖的存储服务。在数据即价值的信息时代，这种立足长远的愿景和持续演进的能力显得弥足珍贵。

尽管在最初的设计中，OSD已经被赋予了足够多的“智能”，例如基于OSD即可独立自主完成诸如数据恢复等难以想象的复杂任务（注意，这是在缺乏中心控制器的分布式系统中），但是历经10年打磨，OSD正在变得愈发聪明：

·对于各类资源的精确控制

·基于完备的对象语义构建无限存储应用的可能性

·不断进化的自我学习、自我管理能力等

可以预见，随着OSD（以及其他组件）的加速进化，在不远的未来，Ceph将变得越来越好用。

为了让Ceph在当下最火爆的容器领域也能占得一席之地，社区正在对OSD占用的资源做进一步的削减并实施更加严格的监控。例如，将所有元数据的内存管理结构，无论多寡，都纳入mempool进行管理；通过将默认的Messenger从Simple切换为Async、削减和合并线程池等方式来减少OSD对于线程（包括文件句柄）等系统资源的消耗。事实上，Ceph正在加速发展自身的容器项目，以实现单个OSD资源准“物理”级别的隔离，从而使得Ceph在未来可以完美适配任何类型的超融合存储应用场景。

完备的故障检测机制使得集群内部任意OSD故障都能被及时和准确地捕获，同时因为在OSD内部依托于PG实现类似于RAID2.0的细粒度数据迁移和数据恢复控制，所以故障切换与恢复可以高度并发，从而大大降低数据丢失风险，并（尽可能地）保证对外提供不中断的业务访问。

# 第5章
# 高效本地对象存储引擎——BlueStore

在上一章中，我们提到OSD依赖ObjectStore这个抽象类型的对象存储引擎管理不同类型存储介质，提供无差异、符合事务语义的本地对象存储服务。当前版本中，ObjectStore主要有FileStore和BlueStore两种实现方式。其中，FileStore由于仍然需要通过操作系统自带的本地文件系统间接管理磁盘，所以所有针对RADOS的对象操作，都需要预先转换为能够被本地文件系统识别、符合POSIX语义的文件操作，这个转换过程极其烦琐，因而效率低下。社区希望藉由全新打造的BlueStore来彻底解决FileStore的上述缺陷。

BlueStore在设计上充分考虑了对下一代全SSD闪存阵列的适配，例如使用RocksDB替代LevelDB来存储元数据。除此之外，BlueStore还有两个比较重要的特点：

·与FileStore不同，BlueStore由自身接管裸设备，从而可以绕过本地文件系统，不再需要执行对象与文件之间的转换，而是直接操作对象，这使得BlueStore的I/O路径大大缩短，I/O时延也随之得到改善。

·考虑到元数据的访问效率对性能有重要影响，BlueStore在设计上将元数据和用户数据严格分离，并支持分别存储至不同的设备。使用

更好的设备来存储BlueStore的元数据（例如使用HDD存储用户数据，使用NVMe SSD的某个分区来存储元数据）通常可以获得更好的性能。

本章按照如下形式组织：首先，我们回顾BlueStore需要重点解决的问题，以及期望具有的特性，由于BlueStore的设计理念与FileStore是如此不同，所以几乎所有磁盘数据结构都需要被重新设计；其次，BlueStore虽然绕过了本地文件系统，但是本地文件系统中诸如缓存、磁盘空间管理等技术对存储系统而言具有通用性，通过针对一些现有方案进行对比分析，我们期望找到一种高度契合BlueStore定位同时兼具优异性能的缓存/磁盘空间管理方案来与BlueStore进行适配；再次，数据库是BlueStore的心脏，RocksDB作为一种键值对数据库，具有高度可定制的特性，通过对RocksDB的一些组件进行替换并合理地进行功能裁剪，我们可以从RocksDB获得更加卓越的性能，从而使得BlueStore的心跳更加有力；最后，通过BlueStore上电、读写等几个关键流程的分析，我们简要探讨了BlueStore的内部实现，并据此给出部署BlueStore的步骤及注意事项。

## 5.1 设计原理

存储系统中，所有读操作都是同步的，即除非在缓存中命中，否则必须从磁盘读到指定的内容后才能向客户端返回。写操作则不一样，一般而言，出于效率考虑，所有写操作都会预先在内存中进行缓存，由文件系统进行合适的组织后，再批量写入磁盘。

理论上，数据写入缓存即可向客户端返回写入完成应答，但是由于内存数据在掉电后会丢失，因此出于数据可靠性考虑，我们无法这么做。一种可行的替代方案是将数据先写入相较普通磁盘而言性能更好并且掉电后数据不会丢失的中间设备，等待后续数据写入普通磁盘后再释放中间设备上的相应空间。这个写中间设备的过渡过程称为写日志，中间设备相应地称为日志设备，一般可由NVRAM或者SSD等高速固态存储设备充当。

引入日志设备后，数据在写入日志设备后即可向客户端应答写入完成。如果此时掉电，而数据尚未写入普通设备，那么系统重新上电后可以通过日志重放进行数据恢复。因此，引入日志（及高速日志设备）可以在不影响数据可靠性的前提下对写操作进行加速，包含日志的存储系统也被称为日志型存储系统。

日志型存储系统一个显而易见的缺点是引入了日志设备，需要消耗额外的硬件资源，但是由于日志空间可以循环使用，所以单个普通

磁盘不需要配备很大的日志空间，实现上也可以由多个普通磁盘共享使用一个大容量的高速日志设备。除此之外，由于同一份用户数据要先后写入日志设备和普通磁盘，所以日志型存储系统还存在“双写”的问题。特别地，如果日志设备和普通磁盘合一，由“双写”带来的性能损失则更加严重。

除了数据可靠性之外，存储系统还必须考虑数据一致性问题，即涉及数据修改的相关操作，要么全部完成，要么没有任何变化，而不能是介于此两者之间的某个状态（All or nothing）。符合上述语义的存储系统也被称为事务型存储系统，其最大的特点是所有写操作都符合事务的ACID语义。

BlueStore可以理解为一个事务型本地日志文件系统（实际上存储的是对象），由于其面向下一代全闪存阵列的设计，所以BlueStore在保证数据可靠性与一致性的前提下，需要尽可能地减小由于日志系统存在而引入的“双写”问题所带来的负面影响，以提升写性能（事实上，目前Ceph的主要商用备份策略依然是跨节点的多副本，所以Ceph的写性能一直为人诟病）。当前，全闪存阵列普遍使用SSD充当数据盘，为了起到写加速的作用，此时日志设备（与数据盘相对，以下简称日志盘）只能由性能更好的NVRAM或者NVMe SSD等高速固态存储设备充当。与传统阵列普遍使用机械磁盘充当数据盘、使用SSD充当日志盘不同，全闪存阵列后端固态存储介质的主要I/O时延开销不再是寻址时间，而是数据传输时间。因此，当一次写入的数据量超过一定规模时，写入日志盘的时延和直接写入数据盘的时延不再具有明显优势（参见表5-1），此时日志存在的必要性大大减弱。

一个可行的改进方案是使用增量日志，即针对大范围覆盖写，只在其前后非磁盘块大小对齐部分使用日志，其他部分因为不需要执行RMW，则可以直接执行重定向写。

表5-1　不同存储介质512kB顺序读写时延

机械磁盘（HitachiHUA722010CLA330）与普通SSD（S3500）读（写）时延相差约为64（162）倍
普通SSD（S3500）和NVMe SSD（P3600）读（写）时延相差约为2（4）倍

| 磁盘类型 | 512kB顺序读时延（ms） | 512kB顺序写时延（ms） |
| --- | --- | --- |
| HDD（HitachiHUA722010CLA330） | 515.32 | 473.37 |
| SSD（Intel S3500） | 8.38 | 2.91 |
| NVMe SSD（Intel P3600） | 3.36 | 0.74 |

为了更好地理解上述改进方案，首先介绍几个与之相关的术语：

·block-size：块大小。磁盘块大小指对磁盘执行操作的最小粒度（也称为原子粒度），例如，对机械磁盘而言，这个最小粒度一般为512字节，即一个扇区；SSD普遍使用更大的块大小，例如4kB。

·RMW：术语RMW（Read Modify Write）指当覆盖写（即改写已有内容）发生时，如果本次改写的内容不足一个磁盘块大小，则首先读取对应的块，然后将待修改的内容与读取的内容进行合并（从这个角度而言，也可以将M理解为Merge），最后将更新后的块重新写入原先位置。RMW引入了两个问题：一是额外的读惩罚；二是执行覆盖写的过程中，如果磁盘异常掉电，那么会存在潜在的数据损坏风险。

·COW：术语COW（Copy-On-Write）指当覆盖写发生时，不是直接更新磁盘对应位置的已有内容，而是重新在磁盘上分配一块新的空间，用于存放本次新写入的内容，这个过程也称为写时重定向。当新写完成、对应的地址指针更新之后，即可释放原有数据对应的磁盘空间。

COW理论上可以解决RMW引入的以下两个问题（但是自身也存在缺陷）：

1）COW机制破坏了数据在磁盘分布的物理连续性。经过多次COW后，前端任意大范围的顺序读后续都将变成随机读，由于读性能至关重要（例如读性能也会间接影响RMW性能），所以在机械磁盘作为主流存储介质的年代，采用COW机制的存储系统容易遭遇性能瓶颈。

2）COW需要频繁地执行空间重分配和地址指针重定向，最终将引入更多的元数据。由于任何操作必然涉及元数据，所以元数据是存储系统中当之无愧的热点数据。如果元数据过多，导致其无法常驻缓存，必然会导致存储系统性能大打折扣。从这个角度而言，为了提升性能，我们需要减少存储系统的元数据，特别是与空间管理相关的元数据（因为其与管理的空间大小成正比）。

了解上述基本概念后，理解BlueStore的写策略变得简单。简言之，BlueStore针对写请求综合运用了RMW和COW策略：任何一个写请求，根据磁盘块大小，将其切分为3个部分，即首尾非块大小对齐部分与中间块大小对齐部分，然后针对中间块对齐部分采用COW策略，首尾非块对齐部分采用RMW策略。考虑到RMW策略存在数据损坏风险，还需要针对这类操作引入日志，将对应的数据（执行过M操作之后的完整数据）成功写入日志盘之后再去真正执行覆盖写，覆盖写完成之后才能释放日志。

针对上层应用（主要是PG），BlueStore提供了读写两种类型的多线程API，这些API都是PG粒度的。由于读请求之间可以并发，而写请求则是排他的，PG内部使用读写锁来保证一致性（例如读写不发生乱序）。此外，所有读请求都是同步的，而写请求出于效率考虑一般需要设计成异步的，所以实现上还需要为每个PG设计一个队列，用于对所有归属于该PG的写请求进行保序。

这个队列称为OpSequencer，不同类型的ObjectStore实现略有不同。在BlueStore的实现中，OpSequencer主要包含两个FIFO（First In

First Out，即先进先出）队列，分别用于对写请求的不同阶段进行保序。如前所述，BlueStore将所有写请求划分为普通和带日志两种。带日志的写请求创建后，其生命周期分为两个阶段：写日志与覆盖写数据。只有当写日志完成之后，才可以开始覆盖写数据。由于后一阶段在实现上使用独立的线程池完成，所以需要一个额外的队列，用于将所有已经进入覆盖写阶段、带日志的写请求在线程池中再次进行排序。

所有写请求通过标准（由ObjectStore定义）的queue_transactions接口提交至BlueStore处理。顾名思义，通过queue_transactions提交的是多个事务，这些事务形成一个事务组，作为一个整体同样需要遵循ACID语义。当前受限于PG实现，一个事务组并不能真正操作同一个PG中的多个对象。

通过BlueStore的API来操作PG中的某个对象，首先需要找到对应的PG上下文和对象上下文。这两类上下文保存了关键的PG和对象元数据信息，使用数据库存储。我们先介绍它们的磁盘数据结构。

## 5.2 磁盘数据结构

相较FileStore而言，BlueStore绕过了本地文件系统，由自身直接接管裸设备，所以元数据类型要丰富得多，除了PG和对象对应的数据结构之外，还有大量与磁盘空间管理相关的数据结构。需要注意的是，在BlueStore中，每种类型的数据结构一般都有磁盘和内存两种格式，习惯上前者采用全部小写、以下划线“_”作为分隔符并且固定以字母t结尾的命名方式，而后者采用首字母大写的命名方式。

### 5.2.1 PG

PG对应的磁盘数据结构称为bluestore_cnode_t（简称cnode），目前仅包含一个字段，用于指示执行stable_mod时PG的掩码位数，如表5-2所示。

表5-2 bluestore_cnode_t

| 成员 | 含义 |
| --- | --- |
| bits | 指示对象（通过 stable_mod）映射至 PG 时，其 32 位全精度哈希值（从最低位开始）有多少位是有效的 |

### 5.2.2 对象

BlueStore中的对象非常类似于文件，例如每个对象拥有BlueStore实例内唯一的编号、采用精简配置、支持扩展属性等，因此对象的组织形式也效仿了文件，基于逻辑段（extent）进行。

参考文件系统，原理上，每个extent都可以写成形如{offset，length，data}这样的三元组形式，各成员含义如表5-3所示。

其中data是个抽象数据类型，主要用于从磁盘上索引对应逻辑段包含的数据。考虑到磁盘空间碎片化严重时，我们可能无法保证为每个逻辑上连续的段（即extent）分配物理上也连续的一段空间，即逻辑段与磁盘上的物理空间段应该是一对多的关系，因此data在设计上主要包含一些物理空间段的集合，每个段对应磁盘上的一块独立存储空间，BlueStore称之为bluestore_pextent_t（简称pextent），其成员含义如表5-4所示。

如前所述，出于访问效率考虑，磁盘的最小访问单元不可能设计为比特，而是块大小，所以pextent中的offset和length也必须是块大小对齐的。如果extent中的offset不是块大小对齐的，则上述物理空间强制对齐约束会使得extent的逻辑起始地址和对应的物理起始地址之间产生一个偏移；相应地，如果extent中的length不是块大小对齐的，则extent中的逻辑结束地址和对应的物理结束地址也可能产生一个偏移。后面我们将会看到，这两个偏移的存在将大大增加数据处理难度。

表5-3　extent抽象类型成员含义

| 成员 | 含　义 |
| --- | --- |
| offset | 对象内逻辑偏移，从 0 开始编址 |
| length | 逻辑段长度 |
| data | 逻辑段包含的数据，为抽象数据类型 |

表5-4　bluestore_pextent_t成员含义

| 成员 | 含　义 |
| --- | --- |
| offset | 磁盘上的物理偏移 |
| length | 长度 |

extent是对象内的基本数据管理单元，很多扩展功能，例如数据校验、数据压缩、对象之间的数据共享等，都是基于extent实现的，下面分别予以阐述。

1.数据校验

数据校验指对数据实施正确性检测。任何我们使用的计算机组件都不是完美的，都可能产生静默数据错误（Silent Data Corruption）。静默数据错误，顾名思义，是不能被计算机组件自身觉察的错误，因此其潜在危害性极大。单个组件静默数据错误出现概率极低，参考欧洲原子能研究组织（CERN）的研究报告，一般在$10^{-7}$水平，因而极易被忽略。但是由于每个组件（例如磁盘、RAID 5控制器、内存等）都可能产生，如果考虑长时间海量数据的场景，那么出现静默数据错误几乎是必然的。

检测静默数据错误的主要手段是引入校验和，即采用某种特定的校验算法，使用原始数据作为输入，得到固定长度输出，此输出即为校验和。之后将校验和与原始数据分开存储，后续读取数据时，分别读取原始数据与校验和，然后针对原始数据重新计算校验和。如果此校验和与之前保存的校验和相等，那么认为数据是正确的，反之则认为数据出错。

这里存在一个潜在问题：如果重新计算出来的校验和与之前保存的校验和不相等，那么到底是原始数据出错，还是保存的校验和出错了呢？BlueStore的解决方案比较简单，它直接将校验和单独使用数据库保存，借助于数据库的ACID特性，我们知道校验和总是可靠的，因此如果重新计算出来的校验和与数据库保存的校验和不一致，则可以断定是原始数据出错。

一种设计良好的校验算法应当具有极低的冲突概率（指使用不同输入得到相同输出的概率。显然冲突概率越低，说明校验算法发生误检的可能性越小，对应的校验算法越好），当然这也取决于输出长度。例如，针对同一种算法，输出64位校验和显然比输出32位校验和冲突的概率要低得多，因为前一种算法可能输出$2^{64}$种不同结果，而后一种算法只能输出$2^{32}$种不同结果。此外，冲突概率一般还与校验算法的执行效率相关。一般而言，更低的冲突概率总是对应更复杂的计算过程，此时整个校验算法的执行效率比较低。因此实际选择校验算法时，也不能一味追求低冲突概率，而是需要根据自身需求在这两者（冲突概率与执行效率）之间进行权衡。

当前主流的校验算法有CRC（Cyclic Redundancy Check，循环冗余校验）、xxhash（一种速度极快的非加密用途哈希算法）等，BlueStore默认都支持，可以根据配置项和自身需求进行选择。

2.数据压缩

数据压缩针对原始数据执行转换，以期得到长度更短的输出，目的是节省存储空间。与数据校验不同，数据压缩的转换过程必须是可逆的，即采用输出作为输入，我们反过来可以还原得到原始数据，后面这个过程称为数据解压缩。

常见的压缩算法大多是基于模式匹配的，因此一般而言，其过程迭代次数决定了压缩收益，这意味着对大多数压缩算法而言，更高的压缩比几乎总是对应更多的性能损耗（因为需要更多的迭代次数，即更长的压缩时间）。因此实际选用压缩算法时，同样需要考虑在这两者（压缩收益与执行效率）之间进行权衡。

采用数据压缩算法需要固化两个关键信息：一是选用的压缩算法；二是压缩后的数据长度。BlueStore使用压缩头保存这两个信息，如表5-5所示。

表5-5　bluestore_compression_header_t

| 成　　员 | 含　　义 |
| --- | --- |
| type | 压缩算法类型 |
| length | 数据压缩后的长度 |

需要注意的是，不同于其他元数据，压缩头是与压缩后的数据一起保存的，这主要是因为存储压缩头需要额外的磁盘空间，所以需要纳入压缩后的数据之中，据此一并计算压缩收益（即压缩率）。当压缩率小于某个阈值（目前为0.875，即要求至少压缩掉原始数据长度的1/8），即采用压缩算法获得的净收益太小，BlueStore将放弃本次压缩。

引入数据压缩的一个重大缺陷在于其对不完全覆盖写的负面影响。假定我们需要针对某个压缩后的extent执行不完全覆盖写，一般而

言有两种方案：一是先从磁盘上读出相应内容，解压缩，然后再执行正常的覆盖写流程；二是直接执行重定向写，即重新分配一个extent，且允许其与需要被覆盖写的extent存在部分重合内容。

容易理解，方案一的缺点在于严重影响写效率；方案二的缺点一是存在空间浪费，二是增加了元数据，使得读操作效率降低。由于压缩可以重复进行，即新分配的extent又可以再次执行压缩，如果采用方案二，执行多次不完全覆盖写后，对象中的部分内容可能被多个extent重复保存，更加浪费空间，有违引入压缩算法的初衷。

目前BlueStore基于方案二实现数据压缩策略，显然这远非一种完美的压缩策略，所以默认没有开启。

3.数据共享

数据共享主要指extent在不同对象之间的共享，一般由对象克隆操作引入。当某个extent的数据被多个对象共享时，需要使用一个中立结构来表明这些数据的共享信息。需要记录的共享信息主要包括共享数据的起始地址、数据长度和被共享的次数。因此，这个中立结构通常可以写成{offset，length，refs}这样的三元组形式。BlueStore称之为bluestore_shared_blob_t，其主要内容是一张基于extent的引用计数表。由于这张表在对象之间共享，所以需要独立于所有共享对象进行存储。

除了数据共享之外，还存在一种比较特殊的extent共享方式——磁盘空间共享。BlueStore由自身管理磁盘空间，所以可以自定义最小可分配空间。一般情况下，这个最小可分配空间与磁盘块大小相等，但是出于分配效率考虑，也可以将其提升为块大小的整数倍。例如，假定磁盘块大小为4kB、最小可分配空间为16kB，此时即使写入1字节的数据，也需要为之分配16kB的磁盘空间。但是实际上这已分配的16kB

空间中，只有一个4kB的块真正被使用，其他3个块均未被使用，因此存在被其他extent共享使用的可能。

至此，我们已经讨论了extent三元组中的data抽象数据类型目前所期望支持的特性，据此可以定义其磁盘数据结构如表5-6所示（BlueStore称之为blob）。

表5-6　bluestore_blob_t

| 成　员 | | 含　义 |
| --- | --- | --- |
| extents | | 磁盘上的物理段集合，单个 extent 的类型为 bluestore_pextent_t |
| flags | FLAG_MUTABLE | blob 对应数据可以被修改，例如没有被压缩 |
| | FLAG_COMPRESSED | blob 对应数据经过压缩 |
| | FLAG_CSUM | blob 对应数据经过校验 |
| | FLAG_HAS_UNUSED | blob 物理空间中包含未被使用的块 |
| unused | | blob 所有未被使用物理块的集合。以块大小为单位对整个 blob 的物理空间进行划分，使用单个比特标识每个区域的状态，如果置位，表明该区域包含用户数据；反之表明该区域不包含任何用户数据 |
| compressed_length_orig | | 压缩控制。<br>分别对应压缩前后的用户数据长度 |
| compressed_length | | |
| csum_type | | 校验控制。<br>csum_type 用于指定一种校验算法类型，例如 CRC32。<br>csum_chunk_order 用于指定计算校验和时，每次输入的原始数据块大小，例如，共计 16kB 的原始数据，csum_chunk_order 为 12，则每次输入 4kB（$2^{12}$）原始数据计算其校验和，共需 16kB/4kB = 4 次计算才能完成校验。值得注意的是，每次计算得到的校验和，其长度都是固定的，具体取决于所选择的校验算法，例如选择 CRC32，则每次输入 4kB 的原始数据，总是得到固定 4 字节的校验和。<br>csum_data 用于保存具体的校验和。<br>由于 BlueStore 使用数据库保存校验和，测试结果表明，键值对中，键或者值长度超过一定范围时，数据库的操作效率会显著降低。所以出于性能考虑，一般会限制单个 blob 所能保存数据的最大长度，进而限制产生的校验和长度。假定校验算法选择 CRC32，blob 允许保存的最大数据长度为 512kB，则每个 blob 至多产生 (512kB/4kB) × 4 = 512 字节的校验数据，从而避免显著降低数据库性能（注意：出于同样的原因，我们会将对象的 extent-map 切成若干个更小的集合进行存储，参考下文） |
| csum_chunk_order | | |
| csum_data | | |

相应的extent磁盘数据结构如表5-7所示。

表5-7　extent

| 成　员 | 含　义 |
| --- | --- |
| logical_offset | 逻辑段起始地址 |
| length | 逻辑段长度 |
| blob | 负责将逻辑段内的数据映射至磁盘 |
| blob_offset | 当logical_offset不是磁盘块大小自然对齐时，将对应逻辑段内的数据通过blob映射到磁盘物理段（或者集合中）会产生物理段内的偏移，这个偏移称为blob_offset；反之如果logical_offset天然块大小对齐，则blob_offset始终为0 |

表5-7中logical_offset、length和blob_offset几个成员的关系是逻辑段与物理段之间的关系，如图5-1所示。

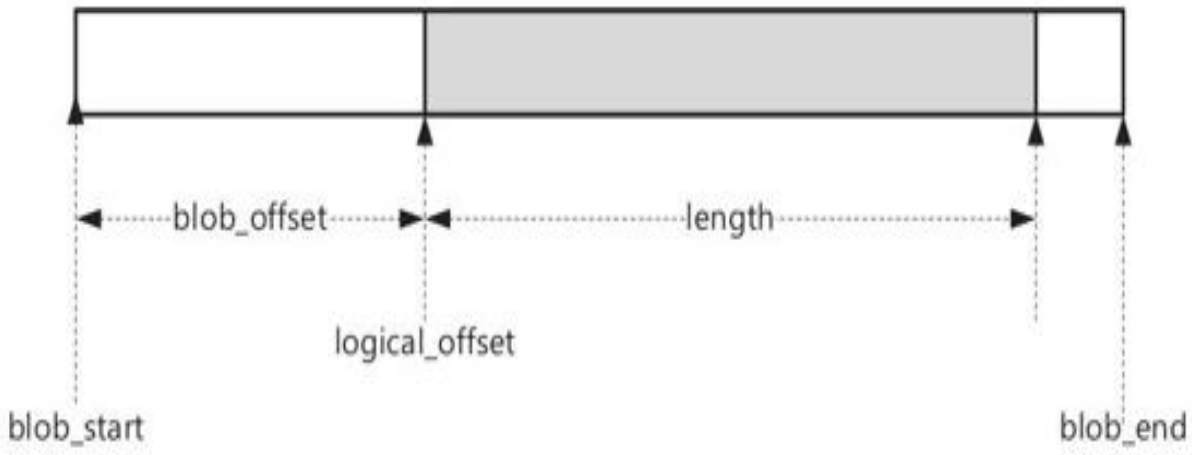



图5-1　extent中的逻辑段与物理段之间的关系

同一个对象的所有extent可以进一步组成一个extent-map，根据extent-map可以索引对象数据。需要注意的是，由于支持稀疏写，

extent之间可能不连续，即extent-map中可能存在空穴。此外，如果单个对象的extent数量过多（例如小块随机写的场景），或者磁盘碎片化严重等，都可能导致整个extent-map编码之后变得十分臃肿，从而严重影响数据库的性能。所以需要对超过一定大小的extent-map进行分片，并且将这些分片信息从其归属对象的元数据信息中独立出来，单独进行保存。这些分片信息称为shard_info，其关键数据成员如表5-8所示。

表5-8　shard_info

| 成员 | 含义 |
| --- | --- |
| offset | 分片对应的逻辑起始地址 |
| bytes | 分片编码后的长度 |

extent-map分片并不是一个十分精确的过程，我们只需要保证每个分片进行编码后，其长度落在一个指定的、相对宽松的范围内即可。因此实现时，总是先基于上一次的分片信息预先估算当前extent-map中每个extent编码后的平均大小，然后据此计算本次分片过程中每个新分片的逻辑起始地址和逻辑结束地址，作为后续真正将extent-map编码存盘时的依据。

由于不是基于精确计算（即不是先将每个extent编码之后，再确定分片信息），分片操作有时可能需要执行多次。此外，如果某个blob被两个相邻的extent共享，而这两个extent又恰好隶属于两个不同的分片范围，那么会因为该blob“跨越”（span）两个分片而无法确定其归属，此时BlueStore将优先针对该blob执行分裂操作，如果分裂失败（例如由于该blob被压缩过或者设置了FLAG_SHARED标志等），则将该blob加入spanning_blob_map，后续将整个spanning_blob_map与对象的其他元数据一并作为单条记录进行保存（BlueStore假定单个对象内这类blob的数量不是很多）。

针对extent-map进行分片的另一个好处在于可以实现对于extent的按需加载，即只有需要访问特定分片范围内的extent时，才从磁盘上读出对应的分片，通过解码还原出该分片内的所有extent至extent-map。同理，当归属于某个分片的extent被修改时，也只需要更新对应的分片即可。这样做一方面减少了常驻内存的元数据数量，另一方面也减少了操作对象上下文时波及的元数据数量，两者都有助于提高性能。

综上，我们可以定义extent-map的磁盘数据结构（实际上也包含部分内存数据结构，BlueStore未进行严格隔离）如表5-9所示。

表5-9　extent-map

| 成　员 | | 含　义 |
|---|---|---|
| extent_map | | 逻辑段集合 |
| shards | key | 将 extent-map 对应的分片写入数据库时，需要全局唯一的 key。这个 key 由对应的对象 key+ 分片的起始逻辑地址 + 固定的 "x" 后缀组成 |
| | offset | 分片的逻辑起始地址 |
| | shard_info | 参见表 5-8。<br>由于 extent-map 独立于对象存储，所以对象的磁盘结构中也需要相应地固化一些关键信息，用于从数据库中还原 extent-map |
| | loaded | 对象上下文从数据库加载时，我们并不需要同步加载完整的 extent-map，只有当对应范围内的 extent 被访问时，才从数据库加载包含该 extent 的分片。本标志位指示对应的分片已经从数据库中加载 |
| | dirty | 指示分片中的 extent 被修改过，后续需要对本分片重新编码、存盘 |
| spanning_blob_map | | 对象内所有跨越两个分片的 blob 集合 |

至此，我们已经了解了构建一个BlueStore对象（称为onode）数据部分所需的关键信息，它们之间的关系如图5-2所示。

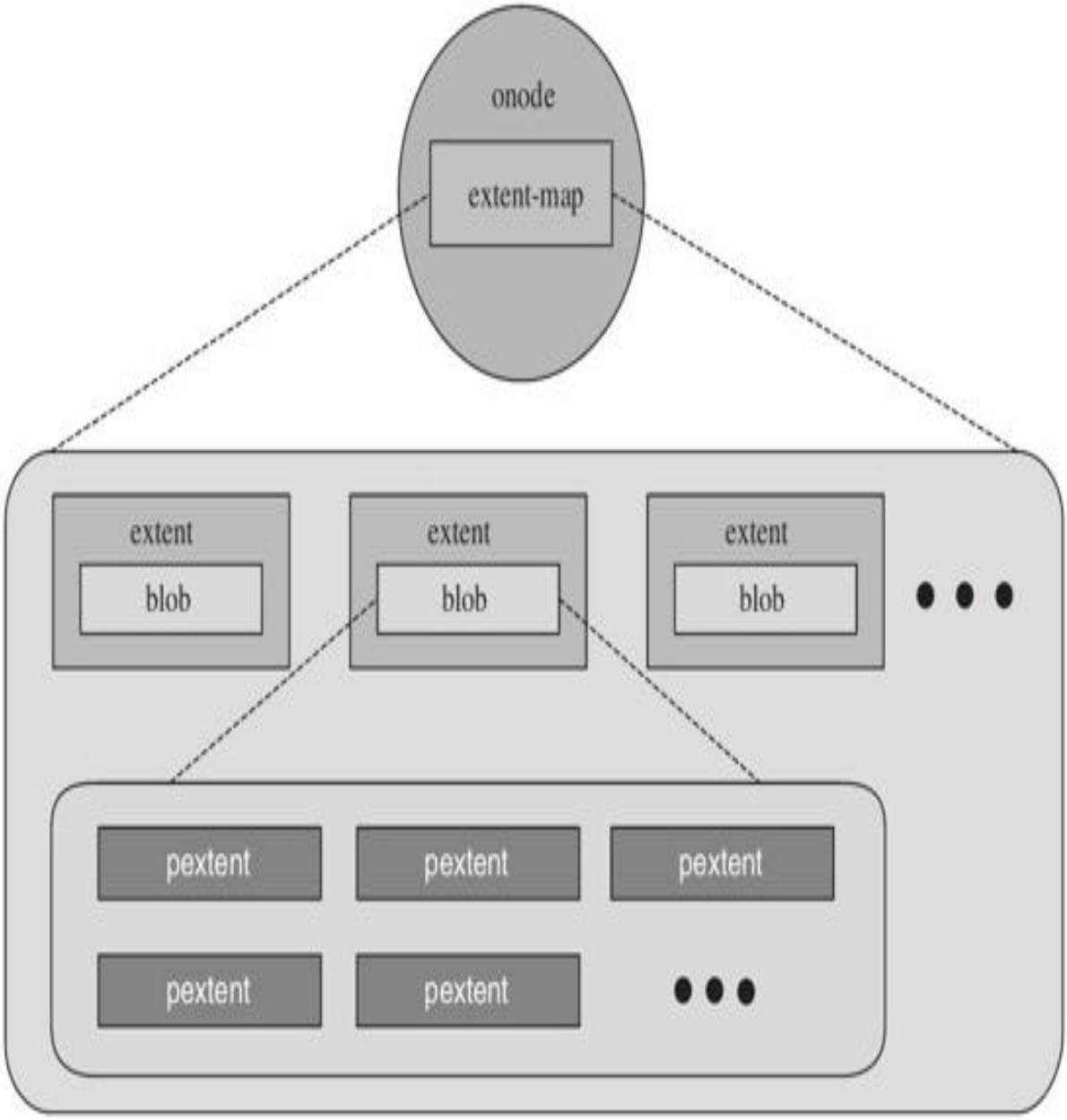



图5-2　onode内部结构（数据部分）

由图5-2可见，每个onode包含一张extent-map，extent-map包含若干extent，每个extent负责管理一个逻辑段内的数据并关联一个blob，blob

通过若干pextent最终将这些数据映射至磁盘。

与文件类似，对象除了可直接用于存储数据之外，也可用于存储自定义属性对。BlueStore将每个onode的存储空间划分为3部分，分别是：数据（如前所述，使用extent-map管理）、扩展属性和omap。数据及扩展属性与文件系统中文件的相关概念类似，omap存储的内容则与扩展属性十分类似，但是两者分别位于不同的地址空间。换言之，omap中的某个条目可以与扩展属性拥有相同的键但是不同的值（内容）。另外，一般的文件系统对于扩展属性长度都有限制（例如XFS限制单个扩展属性长度至多为4kB），但是使用omap则无此限制。从这个角度而言，也可以认为扩展属性只适用于保存少量小型属性对，而omap则适用于保存大量大型属性对。基于此，BlueStore中扩展属性与onode一并保存，而omap则分开保存。

完整的onode磁盘数据结构如表5-10所示。

表5-10　bluestore_onode_t

| 成　员 | | 含　义 |
| --- | --- | --- |
| nid | | 逻辑标识，单个 BlueStore 实例内唯一<br>nid 主要用来保证对象构建关联的 omap 索引时的唯一性 |
| size | | 对象大小 |
| attrs | | 扩展属性对 |
| flags | FLAG_OMAP | 对象关联的 omap 是否使用 |
| extent_map_shards | | 对象关联的 extent-map 的分片概要信息（参见表 5-8），用于从数据库中索引某个具体分片 |
| expected_object_size | | 客户端下发的访问提示信息，用于优化对象的读、写、压缩控制等策略 |
| expected_write_size | | |
| alloc_hint_flags | | |

需要注意的是，虽然FileStore和BlueStore操作对象的API相同，但是两者建立索引的方式却完全不同。老的FileStore直接将对象标识进行字符串转义后作为文件名存储（如果转义后的文件名仍然很长，例如超过255个字符，FileStore还需要再次执行哈希）。BlueStore同样需要对对象标识执行字符串转义，但是由于对象相关的元数据都使用数据库存储，所以转义后的字符串不是作为文件名而是作为数据库中的唯一索引。转义过程有两个关注点：一是在包含所有必要信息的前提下，需要尽可能地缩短转义后的字符串长度，这是因为转义后的字符串无论作为文件名还是数据库索引，长度之于性能都有至关重要的影响；二是对象标识中，既有诸如对象名、命名空间等不定长型的字符串成员，也有（32位）哈希、快照标识等定长整型成员，编码时需要区别对待，合理地界定每个成员的起始与结束位置。

## 5.3 缓存机制

### 5.3.1 概述

现代计算机系统中，为了适配不同组件（例如CPU、内存、磁盘等）之间处理数据速度之间的差异，一般都需要使用缓存。缓存最开始被集成在CPU内部，特指CPU高速缓存，如今概念已被扩展，常见的内存即是一种缓存，甚至磁盘内部也有缓存。一般而言，内存容量远大于CPU高速缓存容量，磁盘容量则远大于内存容量，因此无论是哪一种层次的缓存都面临一个同样的问题：当容量有限的缓存空闲页面全部用完之后，又有新页面需要被添加至缓存时，如何挑选并舍弃原有的部分页面，从而腾出空间放入新页面。解决这个问题的算法被称为缓存淘汰（或者替换）算法，典型的缓存淘汰算法有如下两种。

（1）LRU（Least Recently Used）

LRU算法总是淘汰缓存中当前保存的所有页面中最长时间未被使用过的页面。LRU算法的提出基于时间局部性原理：如果缓存中某个页面正在被访问，那么在近期内它很有可能再次被访问。基于LRU产生了许多变种，典型的如增强时钟算法。如果请求队列体现出了很好的时间局部特性，那么可以证明此时LRU是最优算法。此外，LRU算法容易实现，也是其优点之一。

### （2）LFU（Least Frequently Used）

LFU算法的提出基于另外一种页面访问模型SDD：它假定CPU读写主要存储设备中每个页面（对应磁盘，则以块为单位）的概率是独立的，并且整体上遵循一个固定的概率分布模型（例如正态分布）。符合SDD模型的系统中，有的页面被访问的频率很高，而有的相对较低，因此保留缓存中访问频率较高的页面可以提高命中率。基于SDD产生了LFU算法，它总是选择淘汰缓存中访问频率最低的页面。然而，历史访问频率较高的页面并不见得在将来很长一段时间内都会被持续访问，因此LFU算法一般也需要兼顾频率的时效性。

LRU和LFU算法只能在请求序列呈现明显的时间局部性或者空间局部性时取得较高的收益，因此单独而言都不是一种普适性算法。基于LRU和LFU算法产生了ARC（Adaptive Replacement Cache）算法。

ARC综合考虑了LRU与LFU算法的长处，同时使用两个队列对缓存中的页面进行管理：MRU（Most Recently Used）队列保留最近访问过的页面；MFU（Most Frequently Used）队列保留最近一段时间内至少被访问过两次的页面。ARC算法的关键之处在于两个队列的长度是可变的，会根据请求序列所呈现的特性自动进行调整，以取得在LRU算法和LFU算法之间的某种平衡。当系统中的请求序列呈现明显的时间局部性时，MFU队列长度为0，ARC退化为LRU算法；反之当系统中的请求序列呈现明显的空间局部性时，MRU队列长度为0，ARC退化为LFU算法。因此，无论请求序列呈现何种特性，ARC通过自身参数的调整，都能够始终保持良好的命中率。同时，由于这些参数的动态调整不需要人工干预，所以ARC是自适应的。ARC的缺点在于维护队列众多（MRU、MRF队列及其对应的影子队列，共计4个）、算法复杂，因此执行效率较低，在一些专有系统，特别是对于时延敏感的系统（例如数据库系统）中不是特别合适。

2Q（2Q基于LRU/2发展而来，后者本质上是一种LFU算法）是一种针对数据库、特别是关系型数据库系统优化的缓存淘汰算法。数据库系统由于需要保证每个操作的原子性，所以经常存在多个事务操作同一块热点数据的场景，因此针对数据库系统的缓存淘汰算法主要聚焦于如何识别多个并发事务之间的数据相关性。与ARC类似，2Q也使用了多个队列来管理整个缓存空间，分别称为A1in、A1out和Am。这些队列都是LRU队列，其中A1in和Am是真正的缓存队列，A1out则是A1in和Am的影子队列，即A1out只保存相关页面的管理结构而不保存真实数据。新页面一开始总是被加入A1in，当某个页面在A1in期间被频繁访问时，2Q认为这些访问是相关的，不会针对该页面执行任何热度提升操作，直至其被正常淘汰至A1out。当A1out中某个页面被再次命中时，2Q认为这些访问不再相关，此时执行页面热度提升，将其加入Am队列头部。Am队列中的页面再次被命中时，同样将其转移至Am头部进行页面热度提升。从Am中淘汰的页面也进入A1out。

分析2Q的原理可知，其关键在于使用A1in队列来识别真正的热点数据，即如果缓存中的同一个页面在一个较短的时间段内被连续索引，则将这些索引视作“相关的”，不进行重复统计。这个时间段称为“相关索引间隔”（Correlated Reference Period），在2Q的实现中取决于A1in队列的容量。同理，A1out的容量决定了一个页面被从A1in或者Am中淘汰时，其之前累计的访问热度还能持续多长时间，这个时间被称为“热度保留间隔”（Retained Information Period）。“相关索引间隔”和“热度保留间隔”是2Q判定页面热度的主要依据。在2Q的工程实现中，由于这两个参数的调整基于缓存容量自动进行，所以与ARC相比，2Q是极其高效的，特别适合于类似数据库系统这类时延敏感的应用。

综上，我们讨论了LRU和LFU两种基本缓存淘汰算法，以及由它们发展而来的ARC和2Q算法。实际上，任何算法都不是万能的，原因在于缓存和次级存储设备容量之间存在巨大差异，因此无法建立两者

之间的一一对应关系，而缓存淘汰算法的要义在于对请求序列进行预测，尽可能保留将来使用概率高的页面而淘汰将来使用概率低的页面。因此如果请求序列完全随机（即任何时刻访问次级存储设备中任意位置数据的概率都相等），那么任何算法都不可避免地存在误淘汰的可能，并且误淘汰的概率与所使用的缓存大小成反比。

### 5.3.2 实现

BlueStore目前使用了两种类型的缓存算法：LRU和2Q。如前所述，2Q非常类似于一个简化版本的ARC，区别在于只使用一个影子队列，用于保存从A1in和Am队列当中被淘汰的空页面。

参考Theodore和Dennis的测试结论，推荐A1in与Am队列的容量配比为1：1，A1out队列能够保存的空页面数量则为A1in和Am队列所能保存的页面之和。该推荐配比在所有测试场景下都可以取得一个比较高的命中率，因而具有普适性。

本节首先介绍Cache基类，然后介绍由Cache派生而来的TwoQCache类。至于LRUCache，因为比较简单并且可以视作TwoQCache的一种特例，则不做展开分析。

Cache的基本数据成员如表5-11所示。

表5-11 Cache

| 成员 | 含　义 |
| --- | --- |
| logger | 用于缓存命中率相关的统计 |
| lock | 互斥锁，缓存相关的所有操作几乎都需要在 lock 的保护下进行 |
| num_extents | 当前缓存的 extent 总数 |
| num_blobs | 当前缓存的 blob 总数 |

Cache既可用于缓存用户数据，也可用于缓存元数据。原理上，由于2Q特别适合于数据库应用，所以BlueStore使用2Q作为默认的缓存类型，应该尽量用来缓存元数据而不是用户数据。

BlueStore需要缓存的元数据主要有两种，Collection和Onode，分别对应PG上下文与对象上下文的内存管理结构。考虑到单个BlueStore实例所能管理的Collection数量有限（Ceph推荐每个OSD承载大约100个PG），而且Collection管理结构本身比较小巧，所以BlueStore将所有Collection设计成常驻内存。Onode则不同，一个BlueStore实例本身能够容纳的Onode仅受磁盘空间限制，这就决定了通常情况下Onode几乎不可能常驻内存，于是需要引入淘汰机制。因此Cache设计上主要面向两种类型的数据——用户数据和Onode。

与其他类型的元数据（例如OSDMap）类似，Onode也是直接采用LRU队列管理的。理论上可以直接将所有Onode在Cache中使用单个LRU队列管理，这样效率最高。但是由于每个Onode唯一归属于某个特定的Collection，所以还需要引入一个中间结构，来建立Onode与其归属Collection之间的联系，方便执行Collection级别的操作。例如，删除Collection时，不需要完整遍历Cache中所有Onode，来逐个检查其与被删除Collection之间的关系。这个中间结构称为OnodeSpace，其关键数据成员如表5-12所示。

表5-12　OnodeSpace

| 成员 | 含　义 |
| --- | --- |
| cache | 表明自身归属于哪个 Cache 实例㊂ |
| onode_map | 查找表 |

注：这说明每个BlueStore包含多个Cache实例。这主要是因为不同PG之间的客户端请求可以并发处理，为了提升性能，每个OSD相应地会设置多个PG工作队列（即op_shardedwq包括多个子队列），BlueStore中的Cache实例个数与之对应。

同理，针对用户数据，除了使用通用缓存队列管理之外，也需要额外建立它与上级管理结构之间的对应关系。由于Onode中Extent（对应extent的内存结构）是管理用户数据的基本单元，而Blob（对应Dlob的内存结构）真正负责执行用户数据与磁盘空间之间的映射，所以我们基于Blob引入一个中间结构——BufferSpace，作为连接Blob（用户数据）与Cache的桥梁，其关键数据成员如表5-13所示。

表5-13　BufferSpace

| 成员 | 含　义 |
| --- | --- |
| cache | 表明自身归属于哪个 Cache 实例 |
| buffer_map | 查找表 |
| writing | 包含脏数据的缓存队列（所有归属于同一个 blob 并且当前状态为 BUFFER_WRITING 的 Buffer 使用同一个 writing_list 管理） |

顾名思义，BufferSpace管理的基本单元是Buffer，每个Buffer负责管理Blob中的一段数据（注意，不一定是干净的数据）。Buffer关键数据成员如表5-14所示。

表5-14　Buffer

| 成　员 | | 含　义 |
|---|---|---|
| space | | 表明自身归属于哪个 BufferSpace 实例 |
| state | STATE_EMPTY | Buffer 当前没有保存任何数据。如果 Cache 类型为 2Q，表明对应的 Buffer 已经被淘汰，Buffer 目前位于 Alout 队列当中 |
| | STATE_CLEAN | Buffer 当前保存了干净数据（Buffer 中的数据没有被改写，并且与磁盘数据一致） |
| | STATE_WRITING | Buffer 当前保存了脏数据（Buffer 中的数据被改写，并且与磁盘数据不一致），Buffer 不存在于 Cache 中。<br>当对应的写操作完成，Buffer 状态会转化为 STATE_CLEAN，同时取决于 Buffer 的标志位，Buffer 会被加入 Cache 中或者删除 |
| cache_private | | 当 Buffer 存在于 Cache 中时，例如 2Q，用于将 Buffer 在 Cache 的不同队列之间进行调整，也表示 Buffer 当前位于哪个队列 |
| flags | FLAG_NOCACHE | 指示当前写入的数据不是热点数据，写操作完成后，对应的 Buffer 直接释放，不需要转入 Cache |
| offset | | 三元组，分别对应数据（在 extent 当中的）逻辑起始地址、逻辑长度和数据本身 |
| length | | |
| data | | |
| seq | | 对应写请求的序列号。写请求完成后，由回调函数按序列号批量处理对应的 Buffer，依据 Buffer 的 flags 写入 Cache 或者直接删除 |
| lru_item | | 用于将 Buffer 插入 Cache 的数据缓存队列 |
| state_item | | 用于将 Buffer 插入对应 BufferSpace 中的 writing_list |

参考表5-14，Buffer只有3种状态，因此对应的状态转换也比较简单，如图5-3所示。

所有Onode与Buffer最终都加入Cache，进行全局热度识别和应用淘汰策略。在BlueStore的2Q实现中，这两类数据分别应用了不同的淘汰策略：针对Onode采用LRU，针对Buffer才是真正的2Q。因此，TwoQCache内部实际上维护了4个队列（2Q本身只有A1in、Am和A1out三个队列），如表5-15所示。

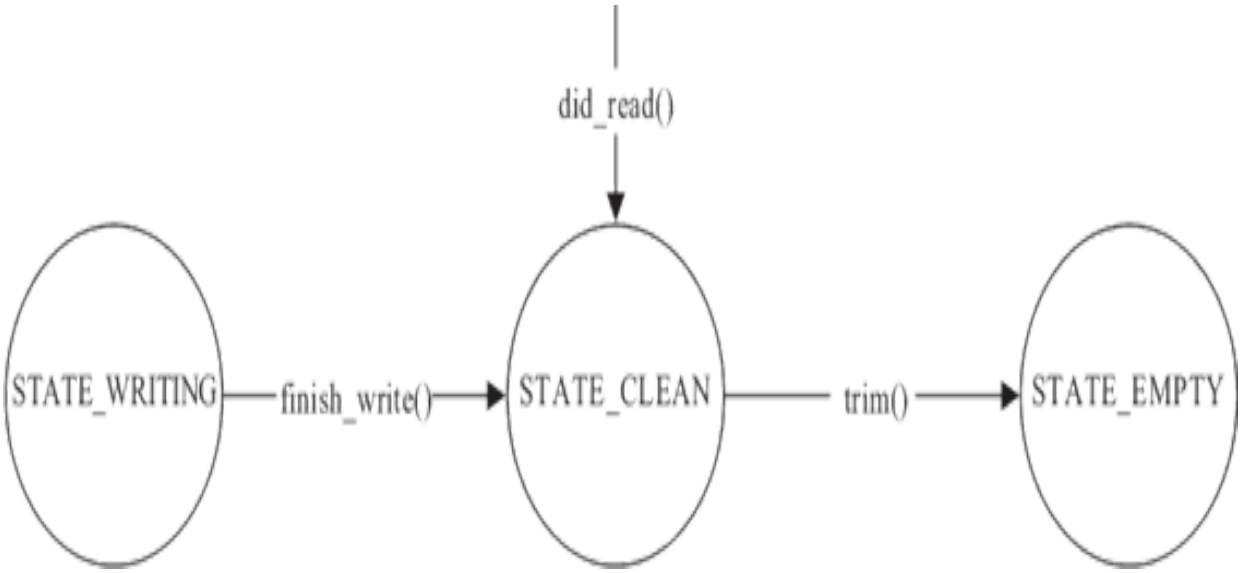



图5-3　Buffer状态转换

表5-15　TwoQCache

| 成员 | 含　义 |
| --- | --- |
| onode_lru | 全局Onode LRU队列 |
| warm_in | 参考2Q相关的理论分析，对应关系如下：<br>warm_in：A1in<br>buffer_hot：Am<br>warm_out：A1out |
| buffer_hot | |
| warm_out | |
| buffer_bytes | 全局缓存容量使用统计及每种队列容量使用统计，用于应用淘汰策略 |
| buffer_list_bytes | |

值得注意的是，虽然TwoQCache内部对Onode和Buffer分开进行管理，但是整个缓存空间却是全局共享的，因此，需要为两者指定一个合适的分割比例。由于用户数据缓存与操作系统以及磁盘缓存有所重叠，所以社区也有人提议将整个TwoQCache全部用作Onode缓存，不过实际效果仍然待验证。

## 5.4 磁盘空间管理

### 5.4.1 概述

块是磁盘进行数据操作的最小单位，任意时刻，磁盘中的每个块只能是“空闲”或“占用”两种状态之一，因而使得采用一个比特表示磁盘中一个块的状态成为可能。如果将磁盘空间按照块进行切分、编号，然后每个块使用唯一的一个比特表示其状态，那么所有比特最终会形成一个有序的比特流。通过这个比特流，可以以块为粒度，索引磁盘中任意（存储）空间的状态。上述这种磁盘空间管理方式称为位图。

容易理解，位图大小与其管理的磁盘空间成正比。例如，每管理1GB的磁盘空间，以块大小为扇区计，固定需要256kB的位图。使用位图进行磁盘空间管理的一个重大缺陷在于如果位图无法全部装入内存，那么其管理效率就会大打折扣。早期的计算机系统中，因为磁盘容量有限（当然内存容量也有限），使得本地文件系统直接采用位图管理磁盘空间成为可能。然而，随着存储系统逐步朝着专业化、高端化的方向发展，一个现代存储系统中内存与磁盘两种存储介质的容量可能达到一个相当悬殊的比例。例如，EMC的VMAXe宣称可支持的最大内存和磁盘容量分别为512GB和1.3PB（1：2662），这样整个系统的位图将超过300GB。显然，这种量级的位图如果作为一个整体，

其索引效率是极其低下的，因此在大容量、集中式的高端存储系统中不可能被直接采用。

一个改进的方向是使用段管理磁盘空间。每个段包含两个成员：offset和length，前者指示被管理磁盘空间的起始地址，后者指示其长度。假定offset与length都是64位无符号整数，这样单个段管理的磁盘空间范围为[0，$2^{64}$(16EB)]，而自身固定需要128=16B的存储空间。可见，如果每个段管理的磁盘空间足够大，那么使用段式磁盘空间管理可以取得极高的收益。反之，如果每个段固定只用于管理一个磁盘基本块，那么所耗费的存储空间将是位图的128倍。因此，段式磁盘空间管理相较于位图而言比较灵活，适用于上层存储应用对于磁盘空间申请范围比较宽泛的场景。

无论使用何种方式管理磁盘空间，当管理的磁盘空间足够大时，都需要考虑其索引效率，而索引效率一般与其对应的内存组织形式有关。例如，针对位图，假定某个位图包含n个比特，那么直接将这n个比特进行顺序排列或者树状排列（常用的树状排列形式为B-tree、AVL tree等二叉树形式），针对单个比特的查找操作时间复杂度分别为O(n)和O($\log_2 n$)（指二叉树）。当n足够大时，两种排列形式的索引效率可能相差几个数量级。

除此之外，由于上层应用对于磁盘空间的需求形式各异，采用不同的空间分配策略所取得的效果也会大相径庭。例如，针对段式管理，常见的空间分配策略有以下3种：

1）首次拟合法（First Fit）。首次拟合法总是查找所有段中第一个满足所需求空间大小的段进行分配后返回。

2）最佳拟合法（Best Fit）。最佳拟合法总是查找所有段中某个空间与所需求空间大小最接近的段进行分配后返回。

3）最差拟合法（Worst Fit）。最差拟合法总是查找所有段中空间最大的段，从中分配所需求的空间后返回。

上述3种空间分配策略各有优劣。一般而言，最佳拟合法适用于请求空间范围较为广泛的系统。因为按照最佳拟合法进行空间分配时，总是查找和分配管理空间与所请求空间大小最为匹配的段，从而使得整个系统中所有段管理的空间处于相差甚远的状态。相反，最差拟合法因为每次都从管理空间最大的段开始分配，从而使得整个系统中所有段管理的空间趋于一致，所以适用于请求空间范围较窄的系统。而首次拟合法的分配方式是随机的，因此它的适用场景介于两者之间，常见于事先无法对请求空间范围进行预测的通用系统。从分配效率来看，最佳拟合法和最差拟合法一般都需要针对所有段按其管理的空间大小进行排序，而首次拟合法一般按照每个段的起始地址自然排序。考虑随着时间推移，空间会逐步趋于碎片化，为了应对后续潜在的大块连续空间分配请求，也为了提升索引效率（索引效率和段的绝对数量强相关，减少段数量可以提升索引效率），需要将已归还的、物理上连续的段再次合并为一个独立的大段。首次拟合法在处理上述段合并过程中具有天然优势，因此其综合效率最高。

综上，一般需要综合考虑请求空间大小的分布规律、效率等因素来制定合适的空间分配策略。例如，一种常见的做法是在系统可用空间比较充裕时采用首次拟合法，以提升分配效率；当系统空间碎片化程度较高时，再切换至最佳拟合法，以减少空间碎片。

作为一种通用存储系统的本地存储系统，原理上BlueStore应该采用段式磁盘空间管理。但是，由于Ceph天然面向分布式设计的特性（这使得每个节点上内存和磁盘容量可以控制在一个较低水平，并且这种分而治之的思想使得每个节点的磁盘空间管理相对独立），加上BlueStore设计之初就被定位为面向SSD等具有比传统机械磁盘更大基

本块、更小标称容量的高速固态存储设备，所以BlueStore空间管理默认采用位图。

如前所述，磁盘所有基本块任意时刻只有“空闲”和“占用”两种状态，因此从这个角度而言磁盘空间管理也可以分为追踪所有空闲空间列表和追踪所有已分配空间列表两个部分。显然，这两部分是强相关的，例如从空闲空间列表中新分配空间需要同步将其加入已分配空间列表，表明其已被占用；反过来，从已分配空间列表中释放空间也需要同步将其加入空闲空间列表，供再次分配，这说明任意时刻我们都可以由一张列表推导出另一张列表。因此，BlueStore进行空间管理时，并不需要将两张列表全部存盘。考虑到每个onode已经详细记录了数据所对应的磁盘空间，并且我们一般在释放空间时执行合并操作（指将物理上连续的多个段合并为一个独立的大段，这样最终空闲空间列表中的条目相对较少），所以BlueStore选择将空闲空间列表存盘。系统上电时，通过加载空闲空间列表，最终可以在内存中还原出完整的已分配空间列表。

为了便于扩展，BlueStore使用FreelistManager与Allocator两种抽象数据类型来分别管理空闲空间列表和已分配空间列表，两者又各有段和位图两种具体实现形式。由于段形式比较简单，所以本书仅介绍两者的位图形式，分别是BitmapFreelistManager和BitmapAllocator。

## 5.4.2 BitmapFreelistManager

BitmapFreelistManager以块为粒度，将数量固定、物理上连续的多个块进一步组成一个段，从而将整个磁盘空间划分为若干连续的段进行管理。每个段以其在磁盘中的起始地址进行编号，可以得到一个BlueStore实例内唯一的索引，从而可以使用数据库固化BitmapFreelistManager中的所有段信息。如前所述，单个块的状态可以使用一个比特进行标记，因此每个段的值部分是一个长度固定的比特流，比特流中的某个比特置位，表明对应的块已经被分配，反之则表明对应的块处于空闲状态。此外，由于使用数据库存储段信息，需要合理调整BitmapFreelistManager中的段大小设置，过大（此时值长度变大）或者过小（此时键值对数量增加）都不利于充分发挥数据库的性能。

系统运行过程中，BitmapFreelistManager中的块需要频繁地在“空闲”和“占用”两种状态之间切换，这可以方便地通过表5-16所示的布尔运算实现。

表5-16　基于布尔运算实现单个块状态快速切换

| 原有状态 | 掩码 | XOR |
| --- | --- | --- |
| 0 | 1 | 0^1=1 |
| 1 | 1 | 1^1=0 |

由表5-16可见，固定使用“1”作为掩码，基于异或运算可以实现每个块在“空闲”（对应“0”）和“占用”（对应“1”）两种状态之间快速切换，因此，BitmapFreelistManager分配或者释放一段空间的运算逻辑是相同的，这是BitmapFreelistManager工作的理论依据。

### 5.4.3 BitmapAllocator

BitmapAllocator实现了一个块粒度的内存版本磁盘空间分配器。与Bitmap-FreelistManager不同，BitmapAllocator中的数据不需要使用数据库存盘，所以可以采用非扁平方式进行组织，以提升索引效率。实现上，BitmapAllocator中的块是以树状形式组织的，如图5-4所示。

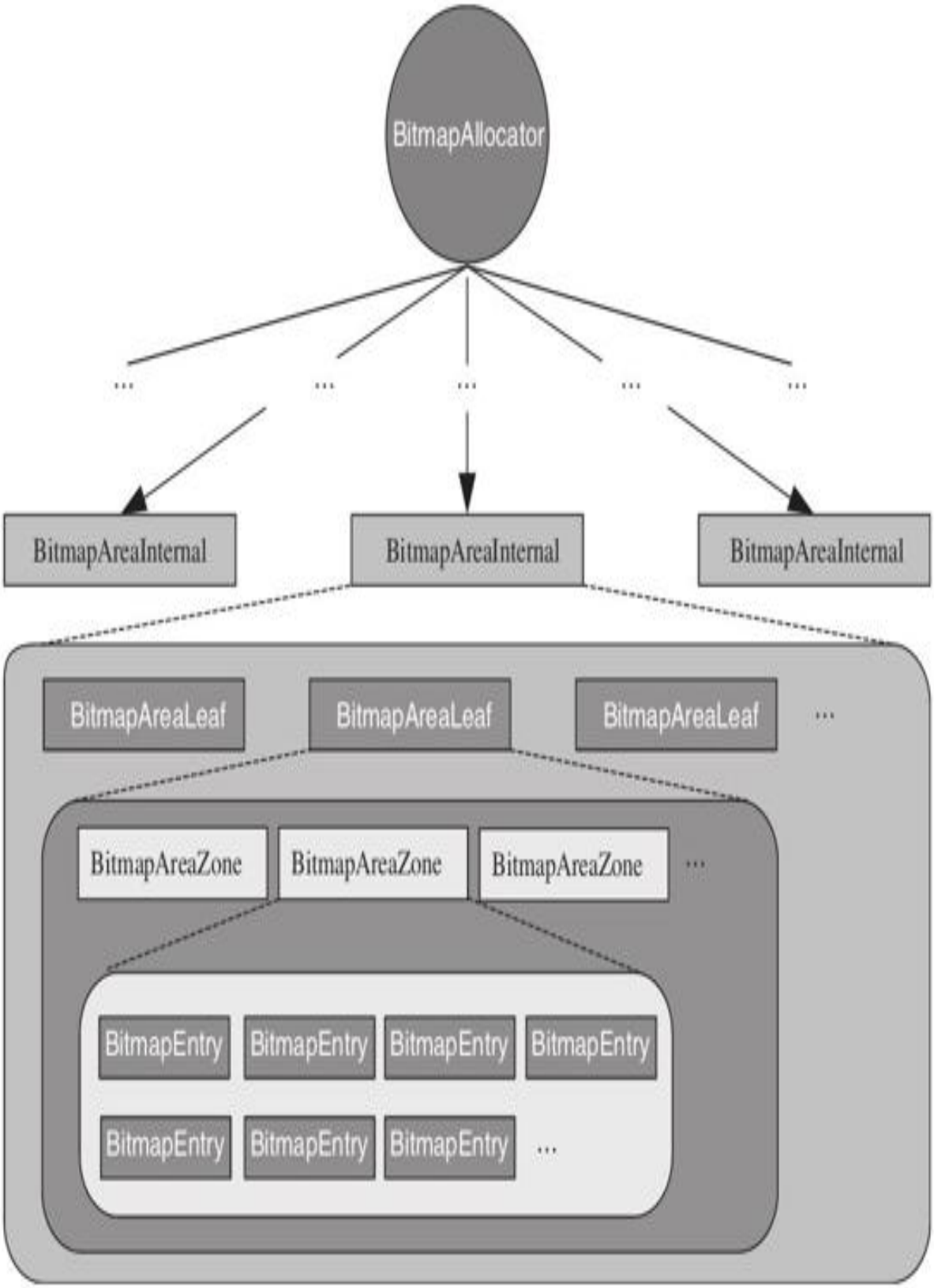

BitmapAllocator
BitmapAreaInternal
BitmapAreaInternal
BitmapAreaInternal
BitmapAreaLeaf
BitmapAreaLeaf
BitmapAreaLeaf
BitmapAreaZone
BitmapAreaZone
BitmapAreaZone
BitmapEntry
BitmapEntry
BitmapEntry
BitmapEntry
BitmapEntry
BitmapEntry
BitmapEntry

图5-4　BitmapAllocator的树状组织结构

BitmapEntry是整个BitmapAllocator中的最小可操作单位，例如，可以定义为64位无符号整数，这样单个BitmapEntry可以同时记录64个在物理上相邻的块的状态。一定数量的BitmapEntry可以进一步组成一个BitmapZone。BitmapZone是BitmapAllocator中单次最大可分配单位，其大小可配置。BitmapZone拥有独立的锁逻辑，所有API都被设计成原子的，因此不同BitmapZone之间的操作可以并发。如果BitmapAllocator管理的磁盘空间过大，为了提升索引效率，可以进一步引入一些中间逻辑结构，将所有BitmapZone再次进行树状组织。这个中间树状结构称为BitmapArea，其中叶子节点称为BitmapAreaLeaf，一个BitmapAreaLeaf叶子节点包含固定数量的BitmapZone；多个BitmapAreaLeaf叶子节点可以组成一个BitmapAreaInternal中间节点；多个BitmapAreaInternal中间节点可以再次作为更上一级BitmapAreaInternal中间节点的孩子节点，直至最终到达根节点——BitmapAllocator。

整个BitmapAllocator的拓扑结构是可配置的，主要受两个参数约束：一是BitmapZone大小，这个参数不但决定了BitmapAllocator单次可分配的最大连续空间，也决定了并发操作粒度；二是BitmapArea中单个（中间或叶子）节点的跨度（span-size），这个参数决定了BitmapArea的高度，进而决定了整个BitmapAllocator的索引效率。

为了减少空间碎片化，同时提升索引效率，BitmapAllocator也允许自定义最小可分配空间。顾名思义，最小可分配空间是BitmapAllocator分配空间的最小粒度，所有通过BitmapAllocator分配的空间必须是最小可分配空间的整数倍，相应地，这要求最小可分配空间必须配置为基本块大小的整数倍。实现时，BitmapAllocator总是以最小可分配空间为单位，直至分配到上层所需求的空间为止（例如最小可分配空间为16个基本块大小，则每找到16个连续的空闲比特视为完成一次分配；如果需求空间为256个基本块，则一共需要进行16次这样的分配），物理

上相邻的空间会在以段形式返回给上层应用时自动进行合并。可见，假定最小可分配空间与块大小相同（这是默认情况），BitmapAllocator执行空间分配时实际上是在逐位扫描，这种方式只在磁盘空间碎片化程度很高时被证明是行之有效的。反之，如果磁盘空间碎片化程度不高，同时又有大量空间需求比较大的分配请求，则不太适合继续采用上述逐位扫描的方式。一个改进的方向是使用大嘴法，即每次分配空间时，不是找到最小可分配空间即完成一次分配，而是找到尽可能多、同时为最小可分配空间整数倍的连续空间才算完成一次分配。

## 5.5 BlueFS

### 5.5.1 概述

诞生于2011年的LevelDB是基于Google的BigTable数据库系统发展而来的日志型键值对数据库（即非关系型数据库），专为存储海量（例如千万级别）键值对设计。

理论上，LevelDB中的键或者值只受存储容量限制，可以为任意长度的字节流，所有键值对严格按照键排序。LevelDB继承并发展了BigTable中LSM-Tree+SSTable的概念，将SSTable在磁盘上进行分级存储，以进一步提高性能，这也是LevelDB名称的由来。

然而随着SSD的逐渐普及，LevelDB使用单线程进行SSTable压缩以及利用mmap将SSTable读入内存的做法已经无法充分发挥SSD的性能。基于LevelDB诞生了RocksDB，后者致力于为新型高性能固态存储介质（例如SSD、NVRAM等）提供更加卓越的键值对型数据库访问性能。测试结论表明：LevelDB在空间利用率上更有优势，而RocksDB在绝大多数场景下性能更有优势。二者性能对比如表5-17所示。

表5-17　LevelDB与RocksDB的性能对比

| 测试条目 | LevelDB | RocksDB |
| --- | --- | --- |
| Write 100M values | 36m8.29s | 21m18.60s |
| DB Size | 2.7GB | 3.2GB |
| Query 100M values | 2m55.37s | 2m44.99s |
| Delete 50M values | 3m47.64s | 1m53.84s |
| Compaction | 3m59.87s | 3m20.27s |
| DB Size | 1.4GB | 1.6GB |
| Query 50M values | 12.12s | 13.59s |
| Write 50M values | 3m5.28s | 1m26.90s |
| DB Size | 673MB | 993MB |

RocksDB具有如下特点：

1）专为使用本地SSD设备作为存储后端且存储容量不超过几个TB的应用程序设计，是一种内嵌式的非分布式数据库。

2）适合于存储小型或者中型键值对；性能随键值对长度上升下降很快。

3）性能随CPU核数以及后端存储设备的I/O能力呈线性扩展。

除此之外，RocksDB还拥有诸多LevelDB所不具备的特性，典型如对于列簇（Column Families）、备份和还原点、Merge操作符的支持等。RocksDB采用C++进行开发，在设计上非常灵活，几乎所有组件都可以根据需要进行替换，其中包括用于固化SSTable和WAL（Write Ahead Log，指日志）的本地文件系统。

由于RocksDB设计理念与BlueStore高度一致，所以BlueStore默认使用RocksDB作为元数据存储引擎。同时，由于操作系统自带的本地文件系统（例如XFS、ext3、ext4、ZFS等）对RocksDB而言很多功能不是必须，为了进一步提升RocksDB的性能，需要对本地文件系统进行裁剪。当然更彻底的解决办法是为RocksDB量身定制一款本地文件系统。在此背景下，BlueFS应运而生。

BlueFS是个简易的用户态日志型文件系统，它恰到好处地实现了RocksDB::Env所定义的全部API。这些API主要用于固化RocksDB运行过程中产生的.sst（对应SSTable）和.log（对应WAL）文件。由于WAL非常影响RocksDB的性能，所以BlueFS设计上支持.sst和.log文件分开存储，以方便将.log文件单独保存在速度更快的固态存储设备（例如NVMe SSD或者NVRAM）之上。

这样，引入BlueFS后，BlueStore将所有存储空间从逻辑上分成了3个层次：

1）慢速（Slow）空间。这类空间主要用于存储对象数据，可由普通大容量机械磁盘提供，由BlueStore自行管理。

2）高速（DB）空间。这类空间主要用于存储BlueStore内部产生的元数据（例如onode），可由普通SSD提供，容量需求比1）小。由于BlueStore的元数据都交由RocksDB管理，而RocksDB最终通过BlueFS保存数据，所以这类空间由BlueFS直接管理。

3）超高速（WAL）空间。这类空间主要用于存储RocksDB内部产生的.log文件，可由NVMe SSD或NVRAM等时延相较普通SSD更小的设备充当，容量需求和2）相当（实际上还取决于RocksDB相关参数设置）。超高速空间也由BlueFS直接管理。

需要注意的是，上述高速、超高速空间的需求不是固定的，而是与慢速空间的使用情况紧密相关。如果这两类空间规划得较为保守，BlueFS也允许使用慢速空间进行数据转存。因此，在设计上，BlueStore将自身管理的部分慢速空间与BlueFS共享，并在运行过程中实时监控和动态调整，具体策略为：BlueStore通过自身周期性被唤醒的同步线程实时查询BlueFS的可用空间，如果BlueFS的可用空间在整个BlueStore可用空间中的占比过小，则新分配一定量的空间至BlueFS（如果BlueFS所有空间绝对数量不足设定的最小值，则一次性将其管理的空间总量追加至最小值）；反之如果BlueFS的可用空间在整个BlueStore可用空间中占比过大，则从BlueFS中回收一部分空间至自身。

综上，如果三类空间分别使用不同的设备管理，那么BlueFS中的可用空间一共有3种。对于.log文件以及自身产生的日志（BlueFS本身也是一种日志型本地文件系统），BlueFS总是优先选择使用WAL类型的设备空间；如果不存在或者WAL设备空间不足，则选择DB类型的设备空间；如果仍然不存在或者DB设备空间也不足，则选择Slow类型的设备空间。对于.sst文件，则优先使用DB类型的设备空间，如果不存在或者DB设备空间不足，则选择Slow类型的设备空间。

Slow类型的设备由BlueStore直接管理，所以与BlueFS共享的这部分空间也由BlueStore直接管理。BlueStore会将所有已经成功分配给BlueFS的空间段单独写入一个名为bluefs_extents的集合，并从自身的Allocator中扣除。每次更新bluefs_extents之后，BlueStore都会将其作为元数据存盘。这样，后续BlueFS上电时，通过BlueStore传递预先从数据库加载的bluefs_extents，即可由自身正确初始化这部分共享空间所对应的Allocator。

至于DB和WAL设备，由于单独由BlueFS管理并且对BlueStore不可见，所以在上电时由BlueFS自身负责初始化。这意味着通常情况下，

BlueFS在上电时会初始化3个Allocator实例，分别用于管理这3种类型的可用空间。

如前所述，对于磁盘空间的管理，一般固化空闲空间列表和已分配空间列表中的任意一种即可。BlueStore选择固化空闲空间列表，同时所有已分配空间信息也保存在每个onode之中，因此这两类信息实际上存在重复。但是理论上单个BlueStore实例能够保存的onode只受存储容量的限制，为了防止上电时从磁盘读取大量onode（从而延长上电时间），BlueStore需要额外固化一张空闲空间列表。

BlueFS则不同，一方面BlueFS存储的数据十分有限，其规模为BlueStore的千分之一至百分之一之间（常见的存储系统中元数据和数据比重一般都在这个范围之内）；另一方面RocksDB生成的.sst文件大小固定，并且从不进行修改，所以BlueFS中绝大部分磁盘空间需求都是比较统一和固定的。因此，基于上述两个因素，BlueFS既不保存空闲空间列表，也不保存已分配空间列表，而是通过上电时遍历所有文件的元数据信息来生成完整的已用空间列表，即Allocator。

## 5.5.2 磁盘数据结构

在上一节中，我们介绍了BlueFS的基本概念，本节介绍与之相关的磁盘数据结构。和其他通用文件系统类似，BlueFS也有文件和目录的概念，并且因为是日志型文件系统，所以BlueFS的磁盘数据主要包括文件、目录和日志3种。

传统文件系统普遍采用层级（树状）结构对目录和文件进行组织，这种组织方式在面向存储海量文件的设计中，因为具有较高的单点查找效率（以二叉树为例，单个查找操作的时间复杂度为$O(\log_2 n)$），并且可以以目录为单位对文件进行区域隔离，所以被实践证明是行之有效的。然而凡事皆有两面性，采用层级结构的文件系统其磁盘数据格式一般而言都比较复杂，因此如果存储的文件数量没有达到一定规模，其效率反而会比直接采用扁平结构低。

如前所述，BlueFS只用于存储单个BlueStore实例的元数据，所存储的文件规格比较统一（绝大多数为SSTable），并且数量十分有限，所以可以直接采用扁平结构进行组织。实现上，BlueFS使用两类表来追踪所有管理的文件及其目录层级关系，如图5-5所示。

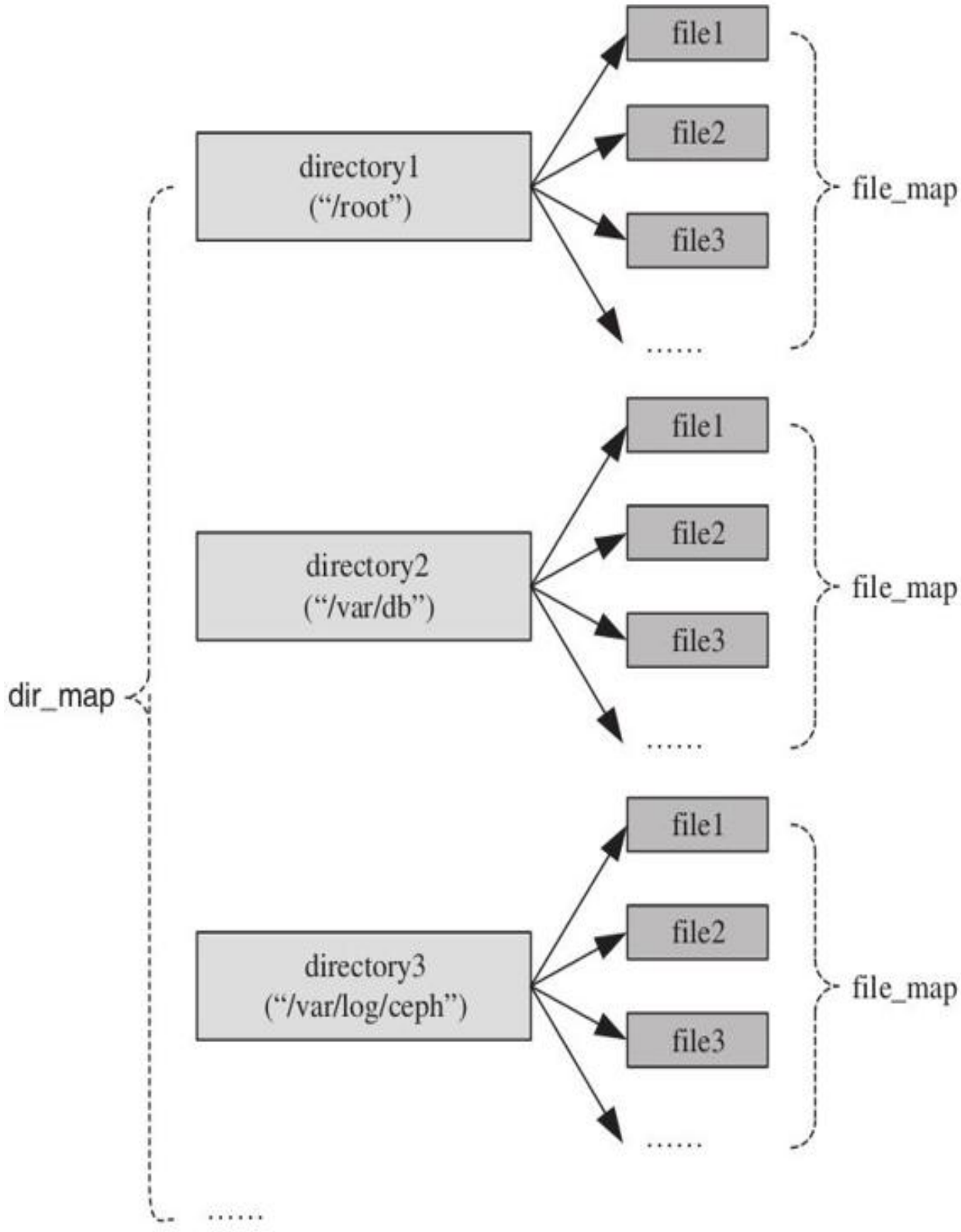



图5-5　BlueFS的文件组织形式

由图5-5可见，BlueFS定位某个具体文件一共需要经过两次查找：第一次通过dir_map找到文件所在的最底层文件夹（即目录）；第二次通过该文件夹下的file_map找到对应的文件。需要注意的是，由于是扁平组织，所以dir_map中每个条目描述的都是文件的绝对路径，即条目之间没有隶属关系。每个文件也采用一个类似inode的结构进行管理，BlueFS称之为bluefs_fnode_t（简称fnode）。因此，file_map建立的实际上是文件名与fnode之间的映射关系。fnode对应的磁盘数据结构如表5-18所示。

表5-18　bluefs_fnode_t

| 成员 | 含　义 |
| --- | --- |
| ino | 唯一标识一个 fnode |
| size | 文件大小 |

（续）

| 成员 | 含　义 |
| --- | --- |
| mtime | 文件上一次被修改的时间 |
| prefer_bdev | 存储该文件优先使用的块设备，例如 .log 文件或者 BlueFS 自身的日志文件将优先使用 WAL 设备 |
| extents | 磁盘上的物理段集合 |

原则上，fnode中的单个extent可以复用pextent（参见表5-4），但由于每个文件都可能使用来自多个不同块设备（WAL、DB和Slow）的空间，所以BlueFS的extent还需要额外记录其归属的块设备标识，如表5-19所示。

表5-19　bluefs_extent_t

| 成员 | 含　义 |
| --- | --- |
| bdev | 归属块设备标识 |
| offset | 归属块设备上的物理地址 |
| length | 空间长度 |

与其他日志型文件系统类似，BlueFS所有写请求也是基于日志的。每个写请求都会生成一个独立的日志事务，然后由RocksDB驱动批量进行提交。

以mkdir操作为例，BlueFS只是简单生成一条日志记录：

```
op_code: OP_DIR_CREATE
op_data: dir(string)
```

即可向RocksDB返回操作成功。

为了减少每次需要存储的日志量以提升效率，BlueFS采用增量日志模式。因此，随着时间的推移，BlueFS中的日志条目会逐渐增加，这一方面会造成不必要的空间浪费，尤其日志本身使用的是（宝贵的）WAL设备空间；另一方面会导致BlueFS在上电时需要经过长时间的日志重放，才能得到完整的dir_map与file_map，所以BlueFS需要定时对日志进行清理。

对日志进行清理的过程称为日志压缩，其主要逻辑是将当前最新的dir_map和所有file_map加入一个独立的日志事务并存盘。这样，下次上电时，BlueFS通过重放这个日志事务，即可还原出完整的dir_map与file_map。因此，这个日志事务可以作为后续增量日志事务的基准。为每个日志事务分配一个独一无二的序列号之后，所有序列号小于此基准序列号的日志事务都可以删除，从而释放日志空间。

在BlueFS开发初期，日志压缩是完全同步的。由于日志压缩需要独占式地访问dir_map和file_map，这会造成严重的写停顿（writestalls），导致客户端业务时延出现剧烈抖动，因此后来社区实现了日志压缩的一个改进版本。改进后的日志压缩除了生成内存基准日志事务这个步骤仍然同步之外（由于是内存操作，所以速度非常快），其他步骤都是异步的，因而可以大大减少由日志压缩引起的写停顿。

综上，我们定义日志事务的磁盘数据结构如表5-20所示。

表5-20　bluefs_transaction_t

| 成员 | 含　义 |
| --- | --- |
| uuid | 日志事务归属 BlueFS 实例对应的 uuid |
| seq | 全局唯一序列号 |
| op_bl | 编码后的日志事务条目，可以包含多条，每个条目由操作码和操作涉及的相关数据组成 |

由于上电时，我们总是先通过日志重放来获取BlueFS的所有元数据，所以还需要一个固定入口，用于索引日志（日志本身采用一个单独的文件进行保存）所对应的存储位置。这个入口称为超级块（SuperBlock，常见的本地文件系统都有类似的概念），BlueFS总是将其保存在DB设备的第二个4kB的存储空间，其结构如表5-21所示。

表5-21　bluefs_super_t

| 成员 | 含　义 | 成员 | 含　义 |
| --- | --- | --- | --- |
| uuid | BlueFS 关联的 uuid | block_size | DB/WAL 设备块大小，固定为 4kB |
| osd_uuid | BlueFS 关联的 OSD 对应的 uuid | log_fnode | 日志文件对应的 fnode |
| version | 超级块当前版本 | | |

BlueFS提供的API与传统文件系统并无二致，这里不赘述。最后，引入RocksDB和BlueFS之后的BlueStore整体架构图如图5-6所示。

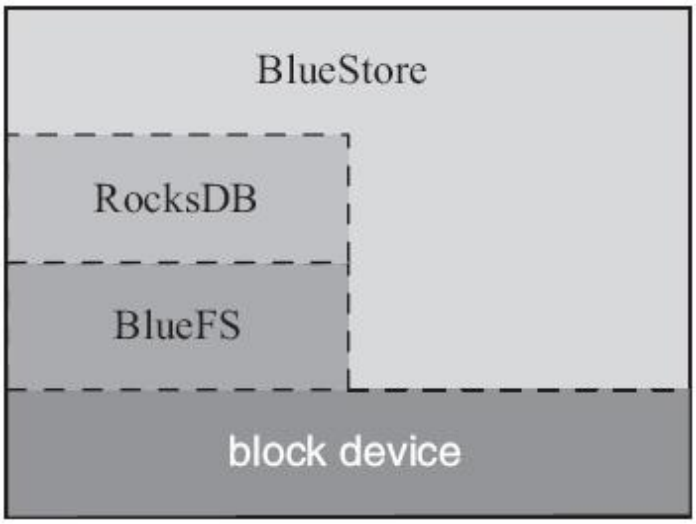



图5-6　BlueStore整体架构（使用RocksDB+BlueFS）

### 5.5.3 块设备

在上一节中，我们知道引入RocksDB+BlueFS后，BlueStore至多可以管理3种类型的存储空间——Slow、DB和WAL。这三类空间的容量和性能需求不尽相同，因此实际部署时可以分别由不同类型的块设备（Block Device）提供。

在Linux的设计中，一切皆文件，因此块设备也被内核当作文件进行管理。对块设备的访问最终通过相应的驱动程序来实现，并且只能以块（从512B到4kB不等）为粒度进行，这也是块设备名称的由来。不同块设备其传输速率除了取决于制造工艺外，还受限于所使用的总线（传输）标准，例如，采用传统SATA 3.0总线标准的块设备其理论传输速率上限为6Gbps，而如果采用PCIe 3.0X4总线标准则其理论传输速率上限高达32Gbps。SATA总线标准及其对应的AHCI接口其实是为高延时的机械磁盘设计的，但是目前依然为主流SSD所采用。随着SSD性能逐渐增强（摩尔定律指出：当价格不变时，集成电路上可容纳的元器件数目，约每隔18～24个月便会增加一倍，性能也将提升一倍），这些标准已经成为限制SSD发展的一大瓶颈。专为机械磁盘而设计的AHCI标准并不太适合低延时的SSD，于是NVMe应运而生。NVMe（Non-VolatileMemory express）与AHCI类似，都是一种逻辑设备接口标准。与传统SATA SSD相比，基于PCIe的NVMe SSD能够提供数十倍或更高的IOPS，同时平均时延下降至几十分之一或更低。

SPDK（Storage Performance Development Kit）是Intel专为高性能、可扩展、用户态的存储类应用程序开发的工具套。SPDK取得高性能的关键在于将所有必须的驱动程序移植到用户态，同时采用主动轮询来替代中断，从而避免内核上下文切换以及中断处理带来的额外开销。SPDK自带NVMe驱动，目前BlueStore基于SPDK实现了对NVMe SSD这类新型块设备的支持，从而在设计上支持使用NVMe SSD充当DB及WAL设备进行性能增强。这使得在未来，基于BlueStore构建全闪存的高性能Ceph存储集群成为可能。

## 5.6 实现原理

在前几节中，我们已经介绍了BlueStore一些基本设计理念及相关支撑组件，本节我们通过几个主要流程介绍BlueStore的具体实现，分别是mkfs、mount、read和write。

### 5.6.1 mkfs

mkfs主要固化一些用户指定的配置项至磁盘，这样后续BlueStore上电时，这些配置项将直接从磁盘读取，从而不受配置文件改变的影响（这也说明每个BlueStore实例的配置项可以是不同的）。之所以需要固化这些配置项，是由于不同的配置项对磁盘数据进行组织的方式不同（BlueStore的每一种组件，例如FreelistManager，都支持多种不同实现方式），如果前后两次上电使用不同的配置项访问磁盘数据有可能导致数据发生永久性损坏。一些需要在mkfs过程中写入磁盘的典型配置项及其含义如表5-22所示。

表5-22　需要通过mkfs写入磁盘的配置项

| 元数据类型 | 含　义 |
| --- | --- |
| os_type | ObjectStore 类型，目前主要有 FileStore 和 BlueStore 两种，两者对于磁盘数据的管理形式完全不同 |
| fsid | 唯一标识一个 BlueStore 实例 |
| freelist_type | 标识 FreelistManager 的类型<br>如前所述，由于 BlueStore 需要固化所有空闲空间列表至数据库，所以 FreelistManager 不能动态改变，否则上电时无法正常从数据库读取空闲空间信息。<br>基于 freelist_type 可以创建相应类型的 FreelistManager，然后由 FreelistManager 从数据库读取所有空闲空间信息，进而在内存中重建完整的空闲空间列表，即 Allocator（这意味着 Allocator 的具体类型在上电时是可以动态改变的） |
| kv_backend | 使用何种类型的数据库，目前有 LevelDB 与 RocksDB 可选 |
| bluefs | 如果使用 RocksDB 作为默认的 kv_backend，是否使用 BlueFS 替换 RocksDB 默认的本地文件系统 |

### 5.6.2 mount

OSD进程上电时，BlueStore通过mount操作完成正常上电前的检查和准备工作，其处理逻辑如图5-7所示。

由图5-7可见，mount操作主要包含以下几个步骤：

（1）校验ObjectStore类型

由于ObjectStore有多种实现方式，不同实现方式对于磁盘的管理方式不同，所以需要在mkfs时固化ObjectStore类型至磁盘，并在mount操作中执行校验，防止将磁盘数据写坏。

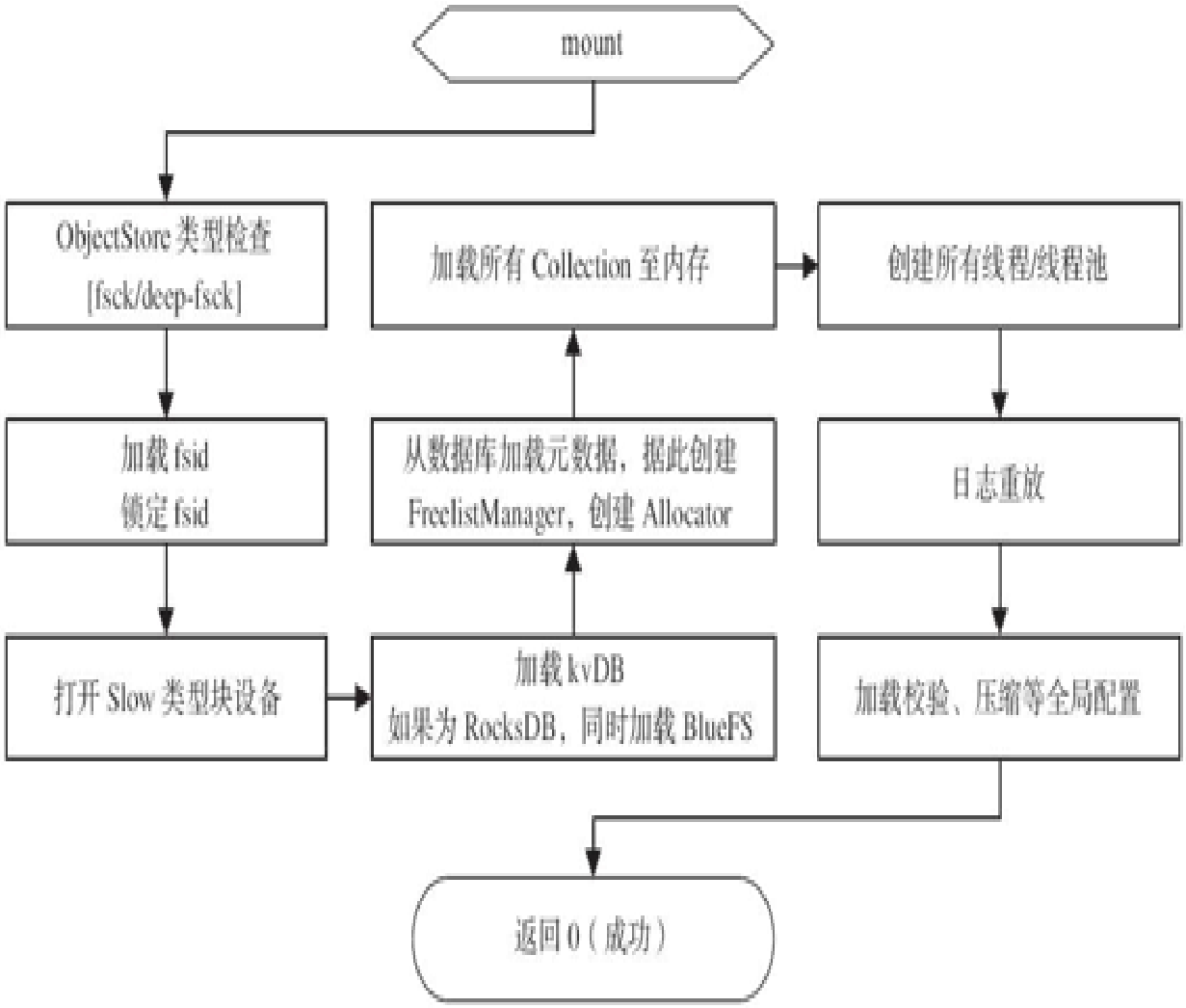



图5-7　mount处理逻辑

### （2）fsck或者deep-fsck

fsck扫描并校验所有元数据。如果打开deep选项，会进一步深度扫描并校验所有对象数据。如果BlueStore中对象数量比较多，那么在mount操作中执行fsck或者deep-fsck会大大延长OSD上电时间，因此也可以通过Ceph提供的工具在业务比较空闲的时间段手动执行。目前mount过程中的fsck选项默认是关闭的。

### （3）加载并锁定fsid

fsid唯一标识一个BlueStore实例。锁定fsid的目的是为了防止对应的磁盘被多个BlueStore实例同时打开和访问，从而引起数据一致性问题。

### （4）加载主设备

如前所述，主设备（即Slow设备）用来存储对象数据，一般可由大容量机械磁盘充当，由BlueStore直接管理。

### （5）加载数据库，读取元数据

表5-23列举了mount过程中需要从数据库读取的元数据。

表5-23　mount过程中需要读取的元数据

| 元数据类型 | 作　用 |
| --- | --- |
| nid_max | 每个对象拥有 BlueStore 实例内唯一的 nid，nid_max 用于标识当前 BlueStore 最小未分配的 nid，新建对象的 nid 总是从当前的 nid_max 开始分配 |
| blobid_max | 类似 nid，整个 BlueStore 实例内唯一。引入 blobid 的目的主要是实现 blob 在对象之间的共享，而共享信息（bluestore_shared_blob_t）需要独立于对象存储，因此需要一个全局唯一的标识 |
| freelist_type | 标识 FreelistManager 的类型 |
| min_alloc_size | 为了提升空间管理效率同时降低空间碎片化程度，BlueStore 也允许自行配置最小可分配空间（Minimal Allocable Size，MAS） |
| bluefs_extents | 从主设备分配给 BlueFS，供 BlueFS 使用的额外空间 |

### （6）加载Collection

如前所述，Collection数量有限，可以在上电过程中就全部预加载并常驻内存。

完成上述操作之后，mount随后会创建一些工作线程，比如用于数据同步的同步线程，用于执行回调函数的finisher线程，用于监控内存使用的mempool线程等。如果上次下电不是优雅下电，那么还可能需要通过日志重放来进行数据恢复。最后，设置了一些诸如校验算法、压缩算法之类的全局参数之后，mount操作全部完成，上层应用（例如PG）可以正常读写BlueStore中的数据。

### 5.6.3 read

read接口用于读取对象指定范围内的数据，目前BlueStore实现的read接口是同步的，其处理逻辑如图5-8所示。

参考图5-8，read逻辑比较简单，主要涉及查找Collection、查找Onode、读缓存和读磁盘4个步骤。

（1）查找Collection

由于BlueStore上电时已经通过mount操作预先将所有Collection加载至内存，并且单个BlueStore管理的PG数量有限，所以这个查找过程耗时相对后续操作几乎可以忽略不计。查找Collection成功之后，如果对应的Collection存在，将以阻塞的形式获取Collection内部读写锁中的读锁。针对同一个Collection，所有读请求（要求中间不能有写请求）可以并发。

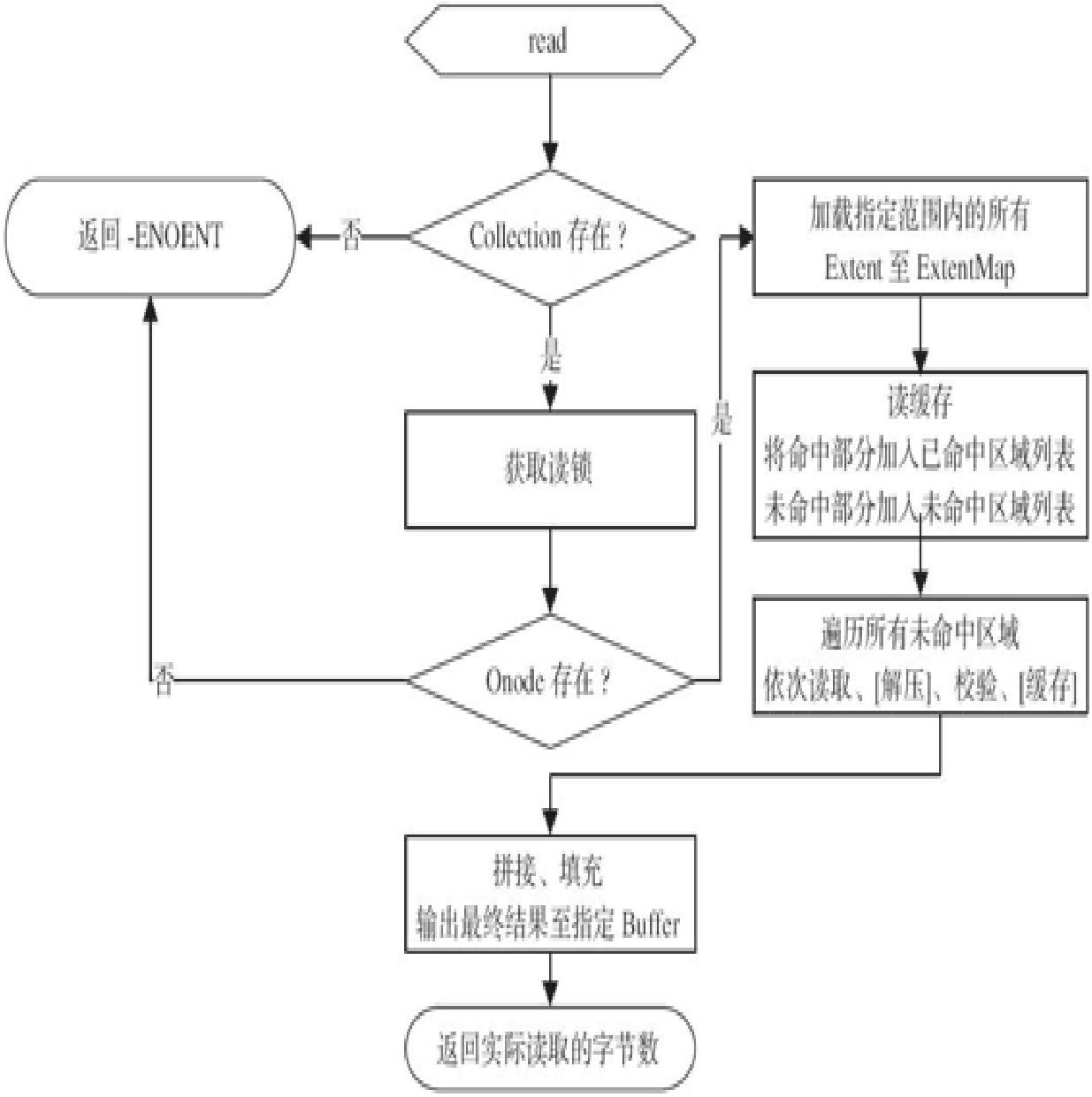



图5-8　read处理逻辑

### （2）查找Onode

如果Onode没有在缓存中命中，需要从数据库中加载。

为了防止从数据库中一次性读取大量数据用于（在内存中）重建Onode，造成前端线程长时间等待（读是同步的），这个过程并不会同

步加载Onode中用于索引数据的关键结构（例如extent-map）。

### （3）读缓存

如果Onode直接在Cache中命中，那么可能有部分数据已经存在于全局Cache之中，此时BlueStore会尝试先从Cache读取指定范围内的数据。读完Cache之后会产生已命中数据区域和未命中数据区域两张列表。

### （4）读磁盘

如果全部或者部分数据没有在Cache中命中，此时需要去磁盘读取。在步骤（3）中我们已经生成了完整的未命中数据区域列表，据此可以加载对应的Extent至内存，然后从磁盘对应位置去读取数据。根据配置，BlueStore可能对直接读到的数据执行校验以防止静默数据错误，同时如果数据经过压缩，还需要执行解压之后才能得到原始数据。最后，通过拼凑和填充（全0）的方式，我们可以得到指定范围内的完整数据，并返回给上层应用。

### 5.6.4 write

包括write在内的所有涉及数据修改的操作，都需要通过queue_transactions接口，以事务组的形式提交至BlueStore。queue_transactions处理逻辑如图5-9所示。

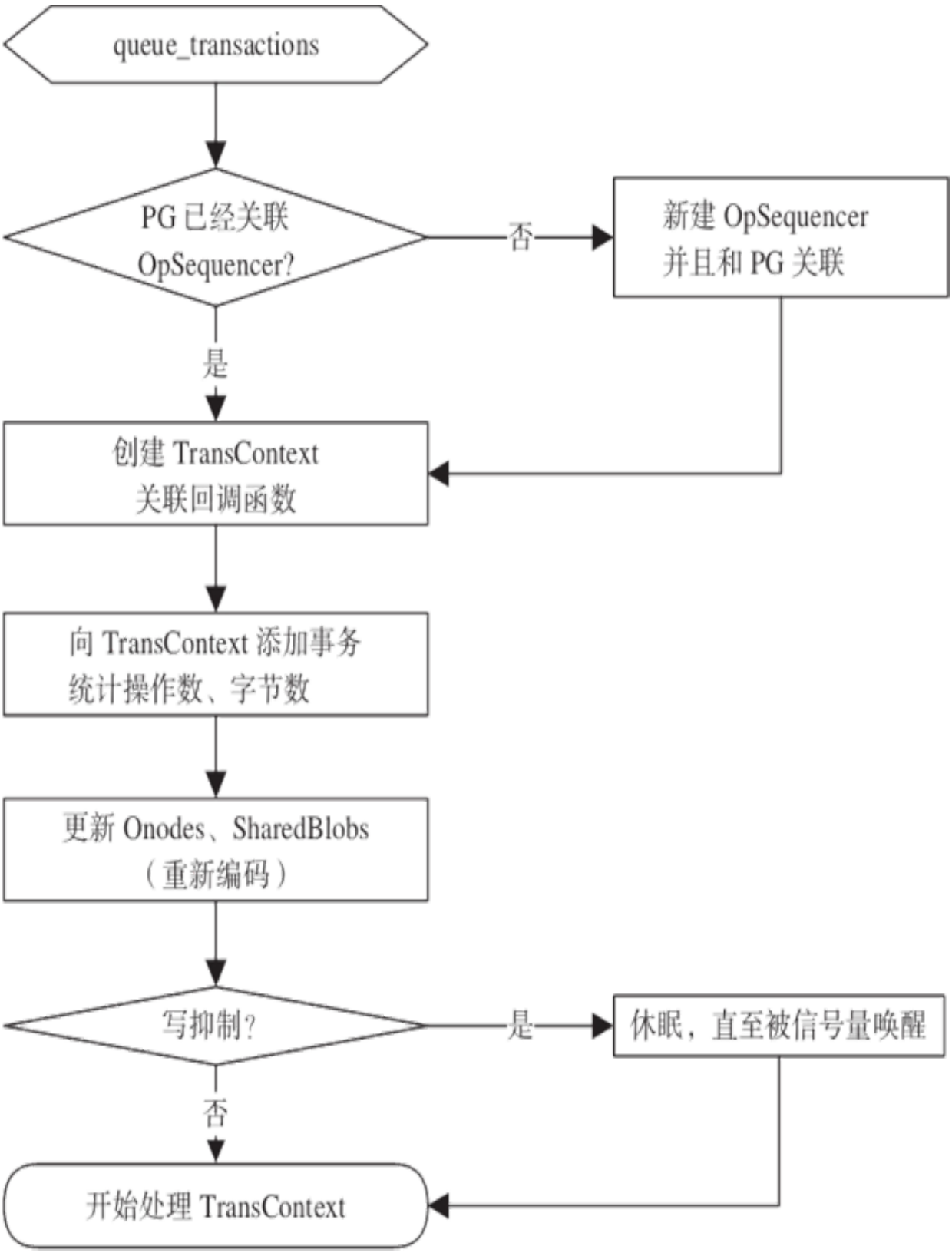



图5-9　通过queue_transactions向BlueStore提交事务

图5-9中，OpSequencer主要用于对同一个PG提交的多个事务组进行保序。针对同一个事务组中的多个操作，BlueStore会创建一个事务上下文——TransContext，将每个写操作顺序添加至TransContext，然后批量进行处理。所有写操作添加完成之后，由TransContext汇总操作数以及波及的字节数，提交至Throttle（顾名思义，这是BlueStore内部的一种流控机制），判断是否需要进行写抑制。如果没有过载，即不需要进行写抑制，则开始执行TransContext。

所有通过queue_transactions提交的事务组都是异步执行的，因此需要指定若干种类型的回调上下文，当相应的事务组执行到某个特定阶段后，通过执行对应的回调上下文来唤醒调用者执行相应操作。

常见的回调上下文共有两种：

（1）on_readable

由于（老的）FileStore默认需要启用日志，一般而言所有写请求都会先写入速度较快的日志盘。写日志阶段完成后，即可向调用者返回写日志完成应答，此时可以确保对应的写请求已经生效（掉电后不会丢失）。

（2）on_commit

也称为on_disk或者on_safe。顾名思义，指数据已经成功写入数据盘。

如果使用FileStore作为本地对象存储引擎，通常会建议使用HDD充当数据盘，使用SSD充当数据盘。这意味着写事务中，通常情况下写日志这个步骤比写数据速度要快。对一些非强一致性应用，此时可以通过on_readable直接告知其写请求已经处理完毕，从而起到性能加速的作用。

BlueStore则不同，由于BlueStore每个写请求只会产生少量WAL日志，并且WAL日志本身也作为元数据的一种直接与其他元数据一并使用数据库保存，所以BlueStore写日志操作要先于写数据操作完成。因此，通常情况下BlueStore会先执行on_commit，然后才执行on_readable。

将write操作加入TransContext的流程如图5-10所示。

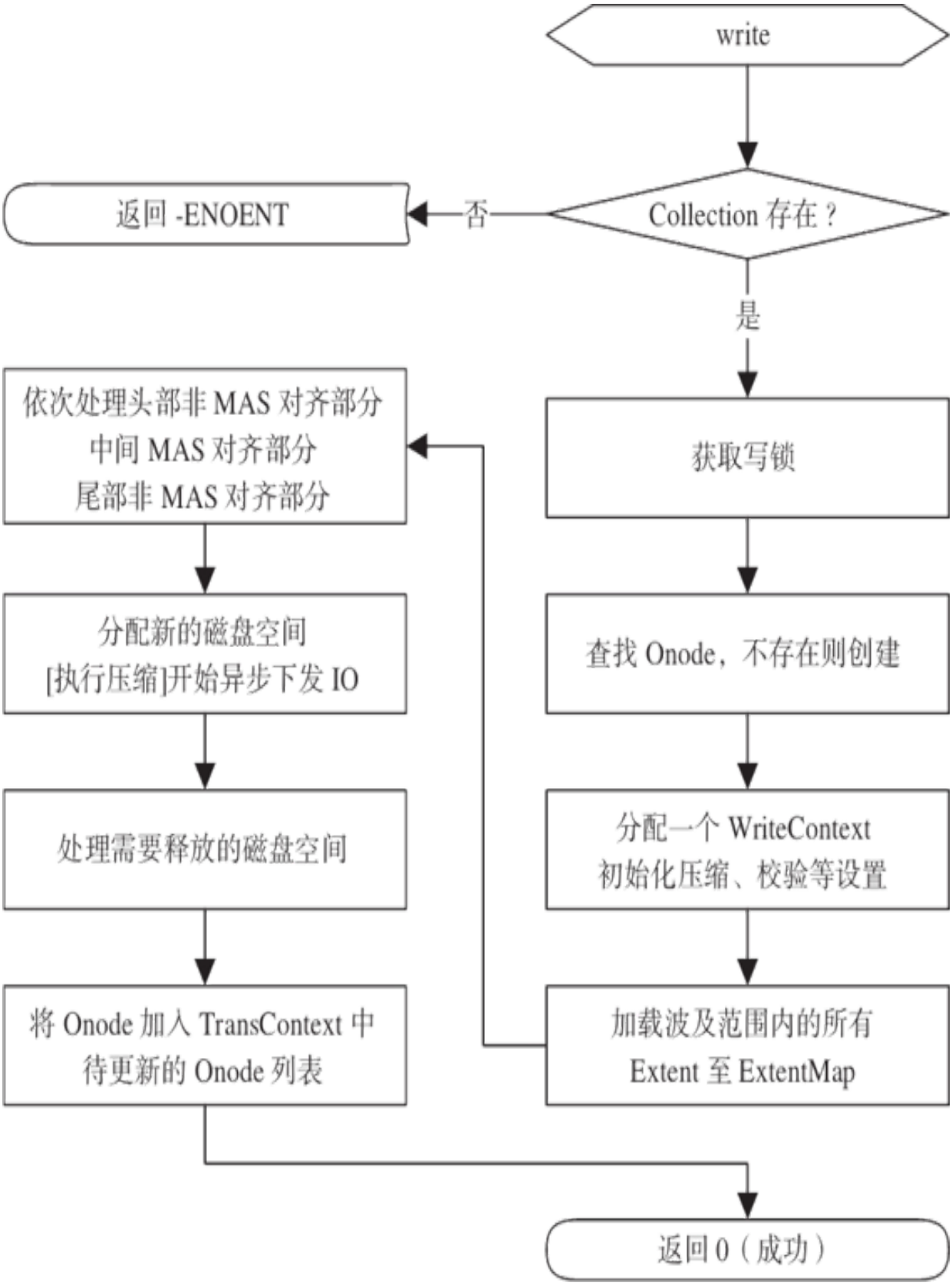

write
Collection 存在？
否
返回 -ENOENT
是
获取写锁
查找 Onode，不存在则创建
分配一个 WriteContext
初始化压缩、校验等设置
加载波及范围内的所有
Extent 至 ExtentMap
依次处理头部非 MAS 对齐部分
中间 MAS 对齐部分
尾部非 MAS 对齐部分
分配新的磁盘空间
[执行压缩]开始异步下发 IO
处理需要释放的磁盘空间
将 Onode 加入 TransContext 中
待更新的 Onode 列表
返回 0（成功）

图5-10　write操作

如果本次write操作范围内没有任何已有数据，则将对应的write操作称为新写，反之称为覆盖写。新写的处理逻辑相较覆盖写远为简单，按照写入对象内的逻辑地址范围是否是MAS对齐，可以将新写分为头尾非MAS对齐写和中间MAS对齐写。对应MAS对齐部分，其处理逻辑如图5-11所示。

注意图5-11中，对应一次写入数据量比较大的情况，之所以拆分为多个新的Extent进行处理，原因之前已经分析过了，主要是防止生成的校验和过大，影响其在数据库中的索引效率。

非MAS对齐写与MAS对齐写类似，区别在于：

·每次至多分配一个Extent。

·Extent中blob_offset不再为0。

·待写入的数据需要执行块对齐，无效部分使用全0填充，防止干净数据被污染。

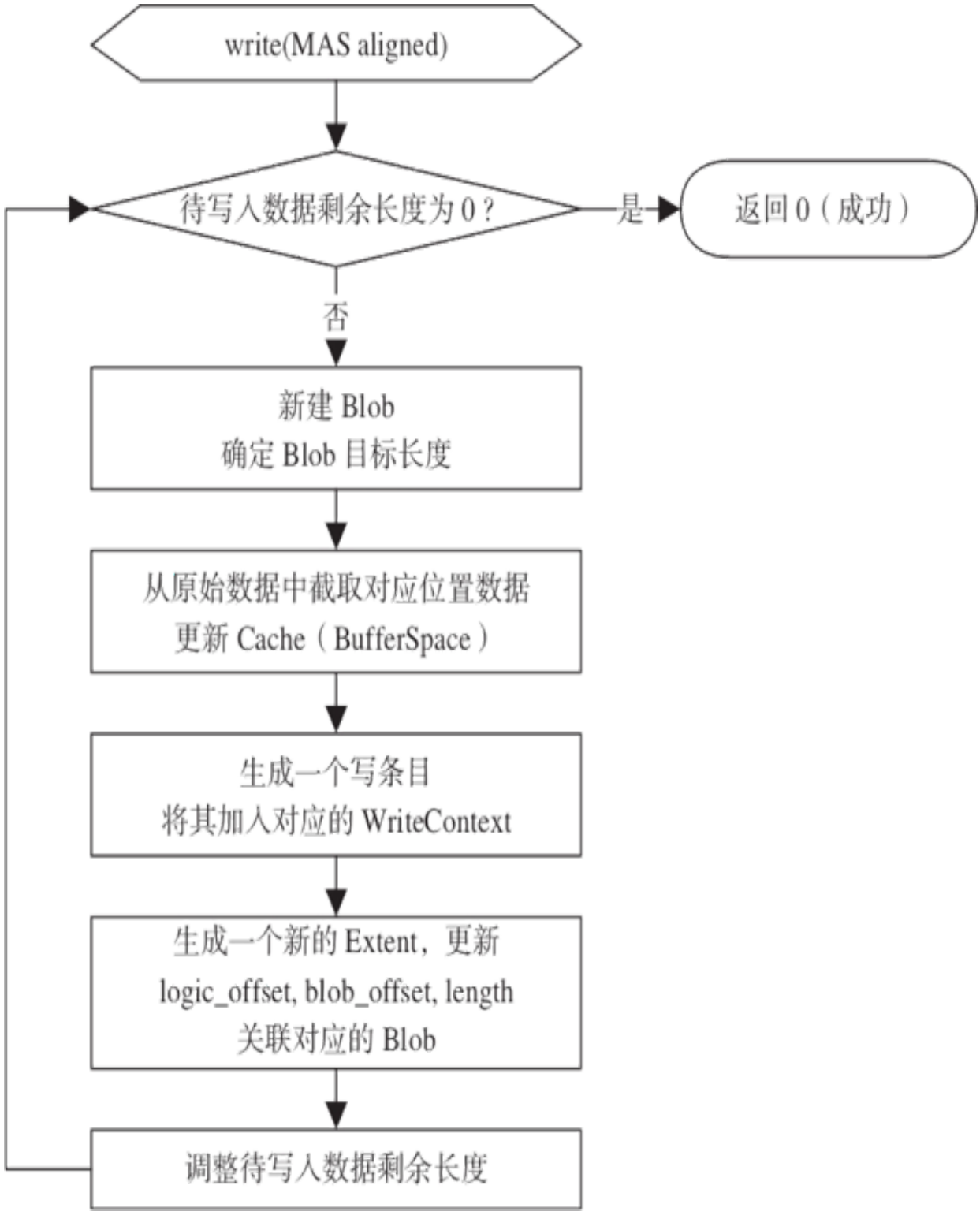



图5-11　MAS对齐写（新写）

对于覆盖写，如前所述，由于BlueStore对MAS对齐部分总是执行COW，其处理逻辑也与新写类似，不同之处在于此时需要找出所有波及范围内已经存在的Extent和它们占有的空间（当然也需要更新对应Onode的ExtentMap，比如移除老的Extent，然后加入新的Extent等），等待事务组同步完成之后再一并释放。

对于头尾非MAS对齐的覆盖写，情况则比较复杂，需要额外考虑以下几个因素：

### （1）能否直接跳过，执行COW

这种情况常见于对应的Extent被压缩过，此时执行RMW操作代价太大，所以直接采用COW策略。

### （2）能否直接复用已有Extent的unused块

如果设置的MAS大于基本块，那么Extent当中有可能产生以基本块为粒度的空穴，这些空穴会被BlueStore标记为unused。由于块是磁盘操作的原子单位，所以针对这些状态为unused的块进行操作不会对Extent中的其他内容造成影响，例如写的过程中掉电，由于之前这部分内容本身就是无效的，即便写入了新的垃圾数据也不会造成任何负面影响。

当然如果新写入的内容不足一个unused块，那么无效部分（显而易见，无效部分在块的头部和（或）尾部）仍然需要使用全0进行预填充。

### （3）是否需要执行WAL写

如果既不能执行COW，也不能复用已有Extent的unused块，那么此时RMW操作已经不可避免。为了防止写坏已有数据，此时需要使用WAL。

图5-12展示了WAL写常见的几种情况。这里假定MAS为基本块大小。如果MAS不是基本块大小，则任何WAL写都可以分为头部非基本块对齐部分、中间基本块对齐部分和尾部非基本块对齐部分，其中头、尾非基本块对齐部分的处理逻辑和图5-12类似。

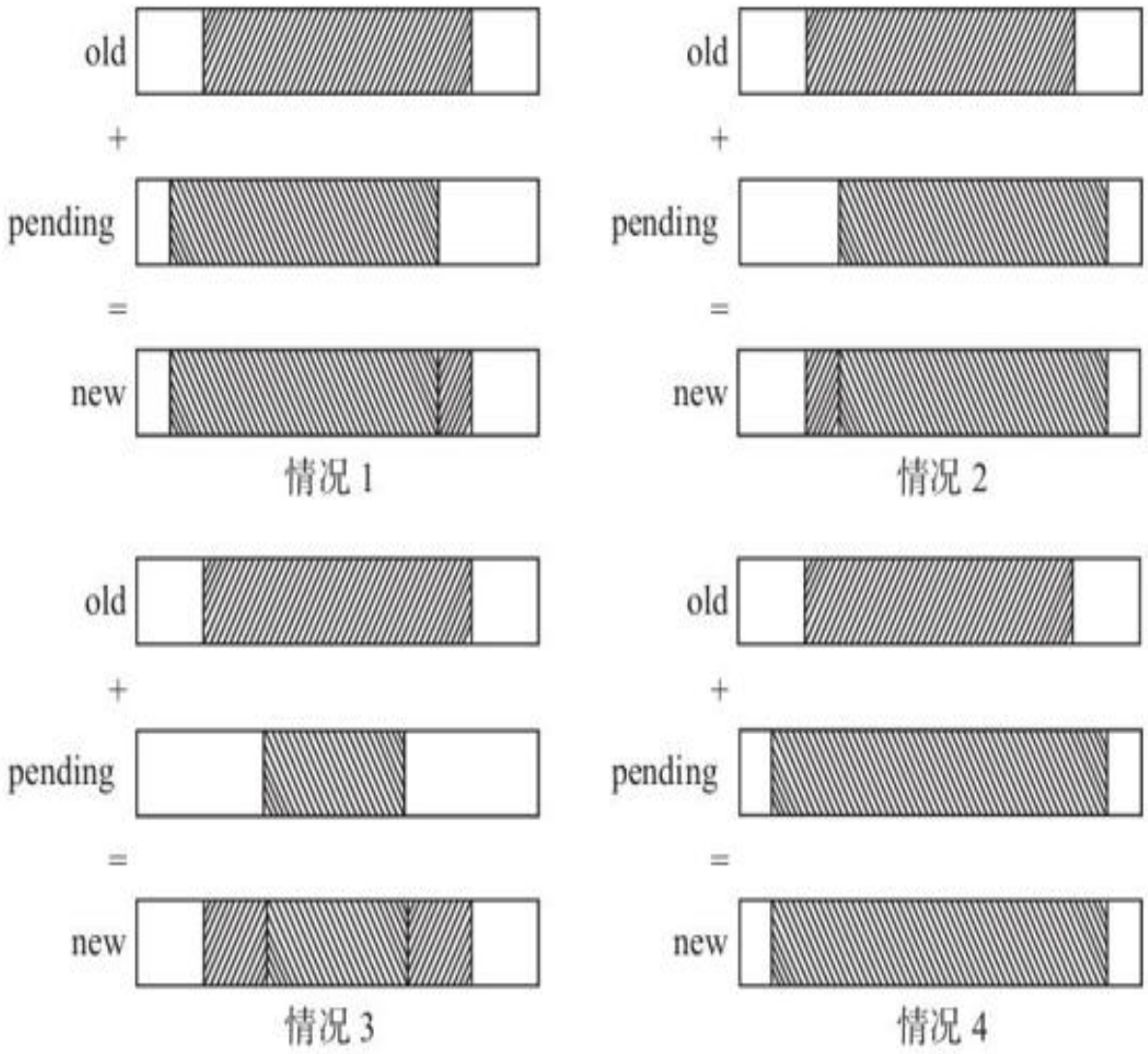



图5-12　WAL写的几种情况

由图5-12可见，WAL写首先需要依次执行补齐读、合并、填充（使用全0），得到一份新的、待写入原有位置的数据（大小为一个基

本块），然后基于此数据生成一个日志事务（除了数据之外，还需要指明需要针对哪个基本块进行覆盖写），并将其加入对应的TransContext。

如前所述，此日志事务与BlueStore中的其他元数据一样，也使用数据库保存。只有对应的日志事务成功写入数据库之后，才能开始对原有区域执行覆盖写，加上之前补齐读、合并、填充已经耗费了大量时间，所以WAL写效率实际上是非常低下的。

当所有写操作都被成功添加至TransContext时，就可以开始处理TransContext了。其处理过程可以分为如下3个阶段：

（1）等待所有在途的写I/O完成

这一部分写I/O主要是所有非WAL写通过COW产生的I/O，将被直接写入新分配的磁盘空间，因此可以在产生事务的过程中就开始同步执行。容易理解，如果此过程未完成即掉电，因为此时新的已分配空间和老的待释放空间尚未在数据库中更新，那么下次上电时不会产生任何影响。

（2）同步元数据至数据库

同一个事务组中受到波及而发生变化的元数据，包括已申请的空间和待释放的空间（如前所述，实际上只需要固化FreelistManager），都会通过一个事务同步提交至数据库。由数据库的ACID属性，我们知道这个阶段要么全部完成，要么没有任何影响，从而可以保证数据一致性。

（3）通过WAL对应的日志事务执行覆盖写（可选，没有WAL写则跳过）

至此阶段，对应的WAL日志事务已经写入数据库，可以通过直接重放WAL日志安全地执行覆盖写。等待覆盖写I/O完成之后，再次（在同步线程中）生成一个用于释放WAL日志事务条目的事务，并将其同步提交至数据库并等待其完成，最后将该事务组所有待释放磁盘空间加入Allocator（如前所述，这是常驻内存的可分配磁盘空间列表），供后续事务组使用。

由于queue_transactions是异步的，因此上述所有涉及等待磁盘I/O完成的操作，都通过注册回调函数实现。后端块设备通过执行回调函数，将对应的TransContext再次加入BlueStore的同步线程，可以继续处理TransContext，直至TransContext最终完成。

## 5.7 使用指南

至此，我们已经完整地介绍了BlueStore的基本设计思想和主要实现细节，本节介绍如何部署BlueStore。同时我们也列出了一些和BlueStore相关的配置参数，供高级用户进行性能调优时参考。

### 5.7.1 部署BlueStore

如前所述，BlueStore实现上非常灵活，一共可以支持Slow、DB和WAL三种类型的块设备，其中Slow设备直接用于保存对象数据，DB和WAL设备则用于保存RocksDB的数据和日志。

BlueStore虽然实现上要比FileStore复杂，但是两者的部署却是类似的，例如BlueStore中的DB、WAL设备与FileStore中的日志设备类似，也是通过符号链接的形式在上电时挂载至OSD的当前工作目录下的。每个OSD的主设备，除了用于直接存储对象数据之外，还需要预留少量空间用于保存OSD启动时的引导数据。因此如果使用BlueStore，每个OSD的主设备还要在部署时再次划分为容量一大一小两个分区，其中较小的分区使用本地文件系统格式化，用于保存引导数据，也作为OSD启动后的工作目录，较大的分区则以裸设备的形式由BlueStore直接接管，充当Slow设备。

因此，一个完整的基于BlueStore的OSD组成模型如图5-13所示。

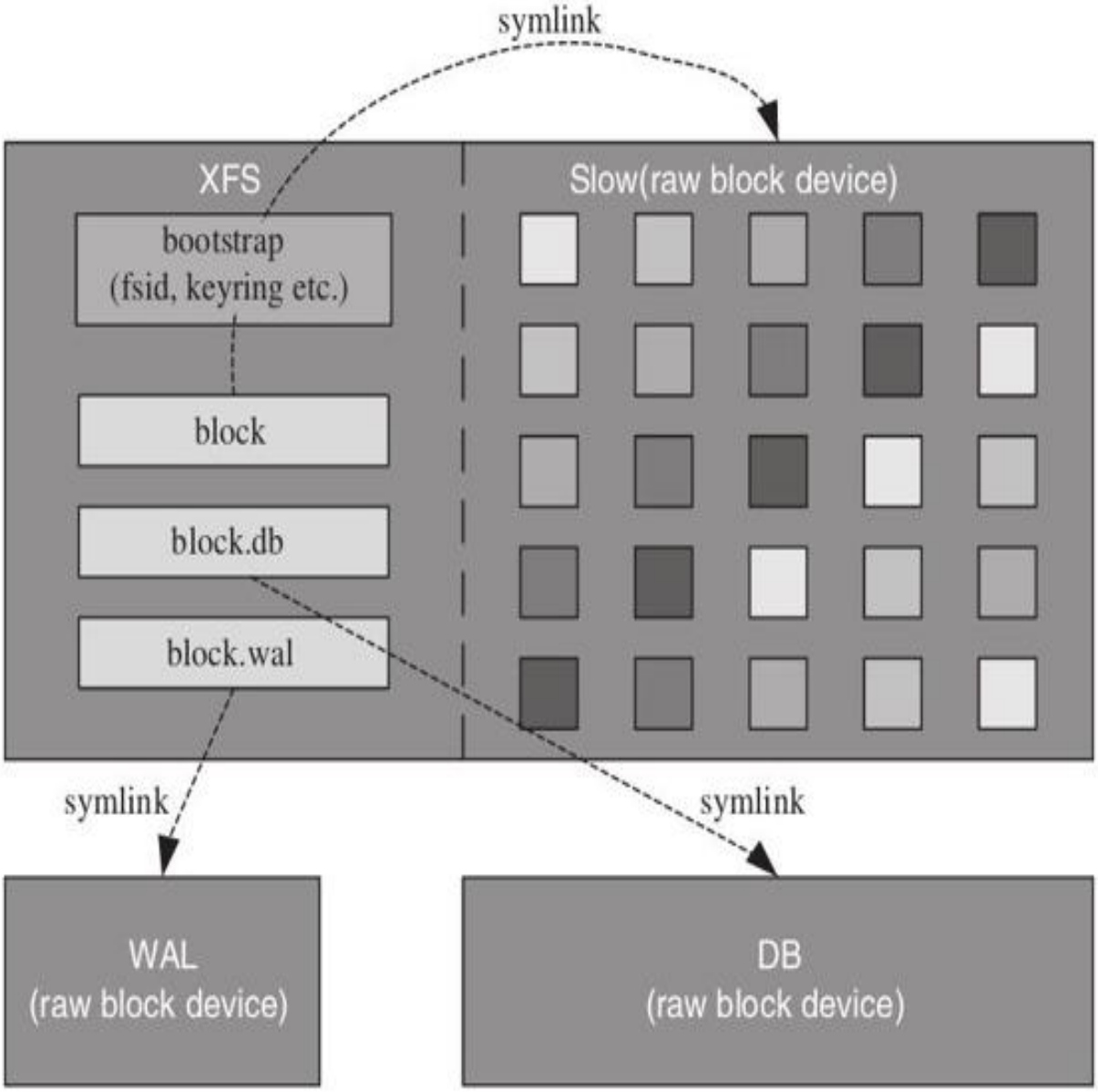



图5-13　基于BlueStore的OSD组成模型

如果所有分区或者块设备已经准备就绪，那么简单通过修改对应ceph.conf文件即可完成BlueStore部署。

---

```
[osd.0]
host = ceph-01
osd data = /var/lib/ceph/osd/ceph-0/
bluestore block path = /dev/disk/by-partlabel/osd-device-0-block
bluestore block db path = /dev/disk/by-partlabel/osd-device-0-db
```

```
bluestore block wal path = /dev/disk/by-partlabel/osd-device-0-
wal
```

这样后续BlueStore上电时，将首先在OSD当前工作目录下创建3个文件：

```
block
block.db
block.wal
```

然后通过读取ceph.conf配置文件，将这3个文件通过符号链接的形式指向对应分区或者块设备所在的真实路径。

当然也可以通过ceph-disk来自动完成BlueStore部署。为此，首先需要规划各个分区大小，同样可以通过ceph.conf文件指定，例如：

```
[global]
bluestore block db size = 67108864
bluestore block wal size = 134217728
bluestore block size = 5368709120
```

然后执行：

```
sudo ceph-disk prepare --bluestore /dev/sdb
```

上述命令将在sdb上创建4个分区，对应关系如下：

```
sudo sgdisk -p /dev/sdb
...
Number  Start (sector)    End (sector)  Size         Code  Name
    1            2048          206847   100.0 MiB   FFFF  ceph
data
    2          206848          337919   64.0 MiB    FFFF  ceph
```

```
block.db
    3           337920           600063   128.0 MiB   FFFF  ceph
block.wal
    4           600064         11085823   5.0 GiB     FFFF  ceph
block
```

当然也支持使用多个块设备，例如：

```
sudo ceph-disk prepare --bluestore /dev/sdb --block.db /dev/sdc
--block.wal /dev/sdc
```

上述命令将在sdb上创建两个分区，然后通过符号链接，将block指向sdb的第2个分区；将block.db指向sdc的第1个分区；将block.wal指向sdc的第2个分区。

或者更进一步地通过：

```
sudo ceph-disk prepare --bluestore /dev/sdb --block.db /dev/sdc1
\
    --block.wal /dev/sdd1
```

将block.db和block.wal分离，分别指向不同块设备sdc和sdd的第1个分区。

需要注意的是，DB和WAL设备的容量需要根据Slow设备的容量、读写速率以及RocksDB本身的参数综合进行规划，一般可以按照三者的裸容量之比——Slow：DB：WAL=100：1：1进行配置。

## 5.7.2 配置参数

本节按照类别汇总截至Luminous版本与BlueStore相关的配置参数，如表5-24～表5-31所示。

表5-24 BlueStore配置参数——部署

| 配置项 | 含 义 |
| --- | --- |
| bluestore_block_path | Slow 类型块设备所在路径 |
| bluestore_block_db_path | DB 类型块设备所在路径 |
| bluestore_block_wal_path | WAL 类型块设备所在路径 |
| bluestore_bluefs | 如果采用 RocksDB 作为默认的数据库引擎，是否使用 BlueFS 作为 RocksDB 的本地文件系统，默认为 true |
| bluestore_kvbackend | 采用的数据库引擎，默认为 rocksdb |
| bluestore_rocksdb_options | RocksDB 相关的配置选项[⊖] |

注：https://github.com/facebook/rocksdb/wiki

表5-25 BlueStore配置参数——extent-map

| 配置项 | 含　义 |
| --- | --- |
| bluestore_extent_map_inline_shard_prealloc_size | 如果整个 extent-map 编码后的长度比较小，那么可以直接将 extent-map 作为一个整体进行存储（即不进行分片），此时可以将编码后的 extent-map 直接在内存中进行缓存，本参数用于对相应的缓存进行预分配（亦即此类 extent-map 编码后的期望长度） |

（续）

| 配置项 | 含　义 |
| --- | --- |
| bluestore_extent_map_shard_max_size | 如果 extent-map 中任意一个分片编码后的实际长度不在此范围内，则触发重新分片 |
| bluestore_extent_map_shard_min_size | |
| bluestore_extent_map_shard_target_size | 单个 extent-map 分片编码后的期望长度 |
| bluestore_extent_map_shard_target_size_slop | 当某个 blob 跨越两个分片时，通过此系数对 bluestore_extent_map_shard_target_size 进行调整（以使得 blob 不再跨越两个分片） |

表5-26　BlueStore配置参数——Cache

| 配置项 | 含　义 |
| --- | --- |
| bluestore_2q_cache_kin_ratio | bluestore_2q_cache_kin_ratio用于控制A1in/Am队列容量比例；bluestore_2q_cache_kout_ratio用于间接控制“热度保留间隔” |
| bluestore_2q_cache_kout_ratio | |
| bluestore_cache_meta_ratio | 缓存中元数据所占比重 |
| bluestore_cache_size | 单个BlueStore实例配置Cache大小，默认为1GB |
| bluestore_cache_trim_interval | 针对Cache执行淘汰的时间间隔 |
| bluestore_cache_type | 缓存类型，包含如下选项：<br>• 2q（默认）<br>• lru |

表5-27　BlueStore配置参数——磁盘空间管理

| 配置项 | 含　义 |
| --- | --- |
| bluestore_allocator | Allocator 类型，包含如下选项：<br>• bitmap（默认）<br>• stupid（本质上是段） |
| bluestore_bitmapallocator_blocks_per_zone | bluestore_bitmapallocator_blocks_per_zone 限制单次可分配的最大连续空间<br>bluestore_bitmapallocator_span_size 限制内存空间分配树的高度 |
| bluestore_bitmapallocator_span_size | |
| bluestore_freelist_type | FreelistManager 类型，包含如下选项：<br>• bitmap（默认）<br>• extent |
| bluestore_freelist_blocks_per_key | 用于调整将空间段以键值对形式写入数据库时的值长度 |
| bluestore_min_alloc_size | 限制 Allocator 一次分配空间的上下限 bluestore_min_alloc_size 同时设置了 Allocator 分配空间的基本粒度，即 Allocator 分配的空间必须是 bluestore_min_alloc_size 的整数倍 |
| bluestore_max_alloc_size | |

表5-28　BlueStore配置参数——BlueFS

| 配置项 | 含　义 |
| --- | --- |
| bluefs_allocator | Allocator 类型，包含如下选项：<br>• bitmap（默认）<br>• stupid |
| bluefs_alloc_size | 最小可分配空间 |
| bluefs_max_prefetch | 如果执行预读，每次预读的最大字节数 |

表5-29　BlueStore配置参数——校验

| 配置项 | 含　义 |
| --- | --- |
| bluestore_csum_type | 校验算法，包含如下选项：<br>• none<br>• xxhash32<br>• xxhash64<br>• crc32c（默认）<br>• crc32c_16<br>• crc32c_8<br>不同校验算法效率不同。对同一种校验算法，最后的数字表明产生的校验数据长度，单位为比特。减少校验数据长度可以获得更好的数据库操作性能，但是会增加冲突概率 |

表5-30　BlueStore配置参数——压缩

| 配置项 | 含　义 |
| --- | --- |
| bluestore_compression_algorithm | 压缩算法，包含如下选项：<br>• zlib<br>• snappy（默认） |
| bluestore_compression_mode | 压缩策略，包含如下选项：<br>• force，强制执行压缩<br>• aggressive，除非前端写请求携带不压缩提示，否则执行压缩<br>• passive，除非前端写请求携带压缩提示，否则不执行压缩<br>• none，不执行压缩（默认） |
| bluestore_compression_max_blob_size | 如果允许对写入的数据进行压缩，BlueStore将基于客户端给出的不同写入提示设置合适的blob大小。<br>例如，对于顺序写，BlueStore会将blob的目标大小设置为bluestore_compression_max_blob_size；否则（采用较为保守的策略）将blob目标大小设置为bluestore_compression_min_blob_size |
| bluestore_compression_min_blob_size | |
| bluestore_compression_required_ratio | 针对数据执行压缩时，要求压缩后与压缩前数据所占用的目标磁盘空间比重必须小于此数值（即减小此数值要求压缩算法执行后取得更高的空间收益），否则放弃压缩 |

表5-31　BlueStore配置参数——事务

| 配置项 | 含　义 |
| --- | --- |
| bluestore_clone_cow | 针对 clone 操作，是否执行 COW 策略 |
| bluestore_max_bytes | 流量控制（写抑制）。<br>bluestore_max_ops 用于指定最大并发操作（同事务，一个事务组可能包含多个事务）数；bluestore_max_bytes 用于指定最大并发字节数。如果包含 WAL 写，那么还需要同时满足并发数不得超出 bluestore_wal_max_ops 和 bluestore_wal_max_bytes 的约束。违反上述任一约束条件将触发写抑制 |
| bluestore_max_ops | |
| bluestore_wal_max_bytes | |
| bluestore_wal_max_ops | |
| bluestore_wal_threads | BlueStore 使用线程池实现 WAL 写，这些参数用于调整线程池相关的配置。<br>bluestore_wal_threads 指定线程池中的服务线程个数。如果单个线程连续占用时间片超过 bluestore_wal_thread_timeout，那么 OSD 进程将休眠（最终将导致 OSD 被 Monitor 标记为 Down，从而触发 OSD 重启）；如果超过 bluestore_wal_thread_suicide_timeout，那么 OSD 进程将自杀 |
| bluestore_wal_thread_suicide_timeout | |
| bluestore_wal_thread_timeout | |

## 5.8 总结和展望

BlueStore于2015年年中引入，目标是替换已经服役超过10年的FileStore，作为新一代默认的ObjectStore引擎，提升FileStore一直为人诟病的写性能。

FileStore存在如下缺陷：

1）数据和元数据分离不彻底。

2）基于POSIX语义的目录层级结构使得针对PG中的对象进行顺序遍历非常困难，但这又是实现数据迁移（Backfill）、数据校验（Scrub）等关键功能所必需的。

3）强烈依赖本地文件系统，但是能够完美适配FileStore的则几乎没有；默认的XFS仍然过于重量级，很多功能对FileStore而言不是必需的。

4）日志叠加日志的设计使得写放大现象异常严重，制约写性能。

5）流控机制不完整导致IOPS和带宽抖动（FileStore自身无法控制本地文件系统的刷盘行为）。

6）频繁的syncfs系统调用导致居高不下的CPU利用率。

社区期望借BlueStore解决FileStore上述缺陷的同时带来至少2倍的写性能提升和同等读性能。此外，增加对于未来一些新型存储介质例如NVMe SSD以及诸如数据自校验、数据压缩等热点增值功能的支持也是必选项。基于当前版本的实测数据（测试工具为FIO），这个目标在基于NVMe SSD的全闪存阵列当中已经部分实现——512kB及以上块的顺序读写场景中实现了接近2倍的性能提升。但是BlueStore在主打机械磁盘的传统存储阵列中的表现仍然差强人意，小粒度随机读写性能与FileStore相比提升有限，性能调优依然任重而道远。

好在BlueStore设计得非常灵活，几乎所有关键组件都可定制，可以随时使用更好的方案进行替换，这为BlueStore后续取得长足进步奠定了良好的基础。

第6章

# 移动的对象载体——PG

在Ceph的设计与实现中，PG是最重要、最复杂的概念之一，原因如下：

1）在架构层次上，PG是连接客户端与ObjectStore的桥梁。它负责将所有来自客户端的请求转换为能够被ObjectStore理解的事务，并在OSD之间进行分发与同步。

2）PG是组成存储池的基本单位。存储池的各种特性，典型的如多副本和纠删码等截然不同的数据备份策略，最终都要依托PG实现。

3）Ceph允许用户自定义故障域的备份策略，使得通常情况下PG需要跨节点读写数据。众所周知，与节点内的数据一致性相比，跨节点的数据一致性（也称为分布式一致性）要复杂得多。

PG最引人注目之处在于，其可以在OSD之间（根据CRUSH的实时计算结果）自由迁移，这是Ceph赖以实现自动数据恢复、自动数据平衡等高级特性的基础。

本章介绍PG的基本概念和相关术语、客户端读写流程，以及PG如何通过Peering快速完成故障切换。

## 6.1 基本概念与术语

面向分布式的设计使得Ceph可以轻易地管理拥有成百上千个节点、PB级以上存储容量的大规模集群。

通常情况下，对象大小是固定的。考虑到Ceph随机分布数据（对象）的特性，为了最大程度地实现负载均衡，不会将对象粒度设计得很大。因此即便一个普通规模的集群，也可以存储数以百万计的对象，这使得直接以对象为粒度进行资源和任务管理代价过于昂贵。

简言之，PG是一些对象的集合。基于这些对象的对象名（以及诸如命名空间在内的其他特征值）生成的哈希值，针对对象归属存储池的pg_num执行stable_mod之后，可以得到PG在存储池内的唯一编号。

一方面，每个OSD承载的PG数量决定了其并发处理多个对象的能力；另一方面，过多的PG反过来又会消耗大量CPU、内存资源等，同时容易使得磁盘工作在超负荷状态。因此，为了发挥集群的最佳性能，在创建存储池时需要合理地设置pg_num。通常情况下，推荐将集群中每个OSD承载的PG数量控制在100左右。

值得注意的是，创建存储池时指定的PG数量实际上是逻辑数量。为了保证可靠性，Ceph会按照存储池绑定的备份策略将每个PG转化为

多个实例，由它们负责将对象的不同备份（对应多副本）或者分片（对应纠删码）写入不同的OSD。

上述过程是受控的，体现在两个方面：一是每个PG需要转化为多少个实例，由存储池的备份策略决定，例如多副本，则每个PG转化的实例数与存储池副本数相等；二是所有实例通过存储池关联的CRUSH规则受控地（但是仍然随机地）分布至位于不同故障域中的OSD上，以实现用户指定级别的数据隔离与保护策略。

仍以多副本存储池为例，由于正常情况下同一个PG所有实例保存的内容完全相同，原则上不需要对它们的身份加以区分，但是出于数据一致性考虑，仍然需要从中选出一个起主导作用的实例，称为Primary，由其作为集中点对所有任务（包括客户端读写）进行统筹。Primary之外的实例则统称为副本。

综上，引入PG的优点如下：

1）集群中PG数量经过人工规划因而严格可控（反之，集群中对象的数量则时刻处于变化之中），这使得基于PG精确控制单个OSD乃至整个节点的资源消耗成为可能。

2）由于集群中PG数量远远小于对象数量，并且PG的数量和生命周期都相对稳定，因此以PG为单位进行数据同步、迁移或者一致性校验等，相较于直接以对象为单位而言，难度更小。

实际上，即便引入PG将OSD资源、任务管理的粒度间接放大几个数量级，要在一个全分布式系统中实现数据强一致性语义的同时，保证其扩展性、可靠性和性能不受影响，仍是一项极具挑战性的工作。表6-1列举了本章以及后续章节需要用到的一些术语，它们是理解客户端读写、Peering等复杂流程的基础。

表6-1　常用术语

| 术　语 | 含　义 |
| --- | --- |
| Acting | 指一个有序的OSD集合，当前或者曾在某个Interval负责承载对应PG的副本。与Up的区别在于，Acting不是基于CRUSH计算出来的，而是按照一定的规则选出来的 |
| Authoritative History | 权威日志。权威日志是Peering判定副本间数据一致性的主要依据，通过副本之间交换Info并基于一定的规则从副本中选举产生。通过重放权威日志，可以使得PG内部就每个对象的应有状态（主要指版本号）再次达成一致 |

（续）

| 术　语 | 含　义 |
| --- | --- |
| Backfill | Backfill 与 Recovery 是 PG 目前使用的两种在线数据恢复（同步）手段。Backfill 指 Peering 完成后，如果基于权威日志无法对 Up 中的某些副本[⊖]实施增量同步（例如承载这些副本的 OSD 离线太久，或者由于全新 OSD 加入集群导致的副本整体迁移），则通过完全拷贝当前 Primary 中所有对象的方式进行全量同步 |
| Epoch | 通常指 OSDMap 的版本号，由 Monitor 负责生成，总是单调递增。<br>Epoch 变化意味着 OSDMap 发生了变化，需要通过一定的策略扩散至所有客户端和 OSD。为了避免 Epoch 变化过于剧烈导致用于传输 OSDMap 的网络流量显著增加，同时也导致 Epoch 消耗过快（Epoch 为无符号 32 位整数，假定每秒产生 13 个 Epoch，那么 10 年时间就会耗光），一个特定时间段内所有针对 OSDMap 的修改会被合并到同一个 Epoch 对应的 OSDMap |
| Eversion | 由 Epoch + Version 组成，其中 Version 是由当前 Primary 生成的序列号，总是单调递增，与 Epoch 一起唯一标识一次 PG 内的写操作 |
| Log | 日志（Log）基于 Eversion 顺序记录所有客户端写请求的概要信息，作为故障恢复时副本之间实施增量同步（即 Recovery）的依据 |
| Info | 指 PG 的基本元数据信息，主要包含如下几个重要字段：<br>last_update、log_tail 分别指向 Log 保存的最新、最老日志条目；<br>last_backfill（对象）指示上一次 Backfill 的进度；<br>history 以 Epoch 为单位进行度量，记录 PG 已经发生过的一些重要事件的时间节点，典型的如 last_epoch_started；<br>stat 包含所有 PG 级别的统计数据，例如对象数量、对象一共使用了多少存储空间等。<br>Peering 过程中，Primary 可以仅通过收集必要副本的 Info 选出权威日志，作为后续判定副本间数据一致性和选择 Acting 的依据 |
| Interval | 指一段 Epoch 编号连续的 OSDMap 变化序列，在此期间 PG 的 Up、Acting 等没有发生过变化。<br>Interval 与具体的 PG 绑定，这意味着即便针对同一个 OSDMap 变化序列，不同的 PG 也可能产生完全不同的 Interval 序列。<br>每个 Interval 的起始 Epoch 也称为 same_interval_since |
| Peering | 指（当前或者过去曾经）归属于同一个 PG 的所有副本就本 PG 所存储的全部对象及状态进行协商并最终达成一致的过程。<br>Peering 的主要依据是 Info 和 Log。此外，这里的达成一致，并不意味着 Peering 完成之后每个副本都实时拥有每个对象的权威版本。事实上，为了尽快恢复对外业务，一旦 Peering 完成并且满足一定条件，PG 就可以切换至 Active 状态继续接受客户端的读写请求，数据恢复可以在后台进行 |
| PGBackend | PGBackend 负责将客户端针对原始对象的操作转化为副本之间的分布式事务操作。<br>视存储池类型不同，当前存在两种类型的 PGBackend，分别是 ReplicatedBackend（对应多副本存储池）和 ECBackend（对应纠删码存储池） |
| PGID | PG 身份标识，由存储池标识 + PG 在存储池内的唯一编号组成[⊖] |

（续）

| 术　语 | 含　义 |
| --- | --- |
| PG Temp | 在 Peering 过程中，如果基于当前 OSDMap 计算得到的 Up 不合理（例如 Up 中的一些 OSD 新加入集群，根本没有 PG 的历史副本），为了降低业务中断风险，可以选择一些仍然相对完好的副本进行过渡。此时需要将这些副本（所在的 OSD）加入 PG Temp，并通知 Monitor 写入 OSDMap[⊖]。Peering 完成之后，将由位于 PG Temp 中的副本临时处理客户端发起的读写请求，直至 Up 中的副本全部恢复（完成同步） |
| PGPool | PG 关联存储池的概要信息。<br>如果是存储池快照模式，PGPool 中包含当前存储池所有的快照序列 |
| Primary | 分为 Up Primary 与 Acting Primary，对应 Up 和 Acting 中第一个 OSD 承载的副本。<br>容易理解 Primary 不是固定不变的，而是可以在不同的副本之间进行切换 |
| Pull/Push | 两种修复（同步）对象的方式，区别在于前者由 Primary 去其他副本拉取待修复对象的权威版本至本地完成修复，后者由 Primary 主动推送待修复对象的权威版本至目标副本（可以包含多个），由副本完成修复 |
| Recovery | 指基于日志针对 PG 进行数据恢复的过程，其最终目标是（结合 Backfill）将 PG 重新变为 Active + Clean 状态。<br>Recovery 是基于 Peering 的结果进行的，一旦 Peering 成功完成，并且 PG 存在降级对象[⊖]，Recovery 就可以切换至后台进行 |
| Replica | 通常情况下指 Up 与 Acting 中除 Primary 之外的成员 |
| Stray | 副本所在的 OSD 不是当前 Up 或者 Acting 的成员（但是过去某个或者某些 Interval 曾是）。<br>如果 PG 已经完成 Peering 并且处于 Active+Clean 状态，那么这个副本稍后将被删除；<br>如果 PG 尚未完成 Peering，这个副本仍有可能被转化为 Acting 中的成员 |
| Up | 指根据 CRUSH 计算出来的、有序的 OSD 集合，当前或者曾在某个 Interval 负责承载对应 PG 的副本。一般而言，Acting 和 Up 总是相同的，但是一些特殊情况下需要通过 PG Temp 来显式指定 Acting，这会导致 Up 与 Acting 不一致 |
| Watch/Notify | 一种用于客户端之间进行状态同步的机制 / 协议，基于一个众所周知的对象进行，例如对 RBD 而言，这个对象通常是 image 的 header 对象。<br>通过向指定对象发送 Watch 消息，客户端注册成为对象的一个观察者（Watcher），这些（Watch 相关的）信息将被固化至对象的基本属性之中，此后该客户端将收到所有向该对象发送的 Notify 消息，并通过 NotifyAck 进行应答。该 Watch 链路有超时限制，需要客户端周期性地发送 Ping 消息进行保活；超时后链路将被断开，客户端需要重连。<br>通过向指定对象发送 Notify 消息，客户端成为对象的一个通知者（Notifier）。在收到所有观察者响应（NotifyAck）或者超时之前，通知者（的后续行为）将被阻塞 |

注：如前所述，实际上Up是一些承载PG副本的OSD集合。为了简化起见，这里没有做严格意义上的区分，下同。

注：纠删码存储池中的PGID还需要加上其分片标识。

注：之所以需要通过PG Temp的方式修改OSDMap，是因为一旦发生了Primary切换，则一方面需要通知新的Primary重启Peering，另一方面需要通知所有客户端，让它们后续能够正确地将读写请求发送至新的Primary。

注：指存在于needs_recovery_map中的对象，needs_recovery_map含义我们将在Peering流程中进行介绍。

## 6.2 读写流程

处理来自客户端的读写请求是PG的基本功能，也是理解Peering等其他复杂流程的基础。读写请求以对象为单位进行，我们先介绍与之相关的术语和管理结构。

1.head对象、克隆对象和snapdir对象

head对象即原始对象。

引入快照机制之后，如果被改写的对象存在快照，为了支持快照回滚操作，一般需要先针对原始对象执行克隆，然后才真正改写原始对象。但是有两个特殊场景需要额外考虑——删除原始对象和重新创建原始对象。

在Luminous版本之前，如果原始对象被删除，但是仍然被有效的快照引用，PG会借助一个临时对象来保存与原始对象相关的历史信息，以便后续执行快照回滚。这个临时对象称为snapdir对象，顾名思义，它仅仅用于保存与原始对象历史快照及克隆相关的信息。同理，如果原始对象被重新创建并且关联的snapdir对象存在，则需要执行snapdir对象清理，并将其保存的快照及克隆等历史信息同步转移回原始对象。

显然，上述功能也可以简单地通过为对象增加一个标志位（FLAG_WHITEOUT）来实现，通过设置或者清除该标志位即可表明对象逻辑上是否存在。基于此，Luminous版本废弃了snapdir对象。

对象有两个关键属性（对客户端不可见），分别用于保存对象的基本信息和快照信息，通常称为OI（Object Info）和SS（Snap Set）属性。

2.object_info_t

它是对象OI属性的磁盘结构，保存对象除快照之外的元数据，具体成员如表6-2所示。

表6-2　object_info_t

| 成员 | | 含　义 |
| --- | --- | --- |
| alloc_hint_flags | | 客户端下发的访问模式提示，例如顺序读写、随机读写、仅执行追加写、是否建议压缩等 |
| data_digest | | 数据校验和 |
| expected_object_size | | 客户端下发的对象大小提示 |
| expected_write_size | | 客户端下发的（写）I/O大小提示 |
| flags | FLAG_DATA_DIGEST | 分别用于标识对象当前是否存在有效的数据校验和、是否使用了omap、是否存在有效的omap校验和 |
| | FLAG_OMAP | |
| | FLAG_OMAP_DIGEST | |
| last_reqid | | 上一次客户端修改本对象时，由其生成的唯一请求标识 |

（续）

| 成　员 | 含　义 |
| --- | --- |
| local_mtime⊖ | 上一次客户端修改本对象时，PG 响应此请求的本地时间 |
| mtime | 上一次客户端修改本对象时所携带的时间 |
| omap_digest | omap 校验和 |
| prior_version | 再上一次修改本对象时，PG 生成的版本号；如果为空，表明再上一次修改动作为创建或者克隆（即对象从无到有） |
| snaps | 对象关联的快照集，仅当对象为克隆对象时有效 |
| size | 对象大小 |
| soid | 对象唯一标识<br>如果 soid.snap 为 CEPH_NOSNAP，说明为 head 对象；<br>如果 soid.snap 为 CEPH_SNAPDIR，说明为 snapdir 对象（Luminous 版本之后已被废弃）；<br>否则为克隆对象，此时 soid.snap 为克隆对象关联的最新快照序列号 |
| user_version | 客户端可见的对象版本号 |
| version | 上一次修改本对象时，PG 生成的版本号 |
| watchers | 成功注册过 Watch 本对象的所有客户端信息 |

注：主要用于解决客户端和服务端时钟产生偏移时（特别是客户端时间超前于服务端），Cache Tier无法正常执行flush操作的故障。

3.ObjectState

object_info_t的内存版本，在其基础上增加了一个exists字段，用于指示对象逻辑上是否存在，如表6-3所示。

表6-3　ObjectState

| 成　　员 | 含　　义 |
| --- | --- |
| exists | 指示关联的对象是否存在 |
| oi(object_info_t) | OI 属性 |

4.SnapSet

对象SS属性的磁盘结构，保存对象快照及克隆信息，具体成员如表6-4所示。

表6-4　SnapSet

| 成　　员 | 含　　义 |
| --- | --- |
| clones | 对象关联的所有克隆对象 |
| clone_overlap | 每个克隆对象与前一个克隆对象（与它们在 clones 中的位置相对应）之间的重叠部分（即两次克隆操作之间对象没有被修改过的部分） |
| clone_size | 每个克隆对象大小 |
| head_exists | 指示当前 head 对象是否存在 |
| seq | 对象关联的最新快照序列号 |
| snaps | 对象关联的所有快照序列号（包含 seq） |

5.SnapSetContext

SnapSet的内存版本，主要增加了引用计数机制，便于SS属性在head对象与克隆对象之间共享，如表6-5所示。

表6-5　SnapSetContext

| 成　员 | 含　义 |
| --- | --- |
| exists | 指示 SnapSet 是否存在 |
| ref | 引用计数。SnapSetContext 可能被同一个对象的多个相关对象（head 对象、克隆对象）关联 |
| registered | 是否已经将 SnapSetContext 加入缓存 |
| snapset(SnapSet) | SS 属性 |

6.SnapContext

如果是客户端自定义快照模式（例如RBD，可以针对每个image单独执行快照操作），那么由客户端下发的请求自身会携带SnapContext，包含此客户端当前所有的快照信息；如果是存储池快照模式，那么PGPool关联的SnapContext会包含此存储池当前所有的快照信息。

SnapContext的具体成员如表6-6所示。

表6-6　SnapContext

| 成　员 | 含　义 |
| --- | --- |
| seq | 最新快照序列号 |
| snaps | 当前所有快照序列号（包含 seq 并且降序排列，即总有 snaps[0] == seq） |

7.ObjectContext

对象上下文保存了对象的OI与SS属性，此外，内部实现了一个属性缓存（主要用于缓存用户自定义属性对）和读写互斥锁机制，后者用于对来自客户端的请求进行保序。

对象上下文的具体成员如表6-7所示。

表6-7　ObjectContext

| 成　员 | 含　义 |
| --- | --- |
| attr_cache | 属性（指用户自定义属性）缓存 |
| obs(ObjectState) | 对象状态 |
| rwstate | 读写锁，用于对来自客户端的请求进行排队，以保证数据一致性，共有3种类型：RWREAD、RWWRITE和RWEXCL<br>与常见的读写锁实现逻辑不同，上述3种类型中，只有RWEXCL是真正的互斥锁，其他两种都可以被重复加锁<br>rwstate内部维护了一个等待队列（FIFO），如果加锁失败，对应的请求会进入该队列等待锁被释放后重试 |
| ssc(SnapSetContext) | SS上下文 |

8.Log

日志（Log）顺序记录了客户端写请求的概要消息，使用PG元数据对象的omap保存。单个日志条目结构如表6-8所示。

表6-8　日志条目（pg_log_entry_t）

| 成员 | | 含义 |
| --- | --- | --- |
| op | MODIFY | 除删除之外的写操作（注意：创建对象也会归结为 MODIFY 操作，此时 prior_version 为 0） |
| | CLONE | PG 当前并不支持由客户端直接发起克隆操作。这里的 CLONE 指 PG 在处理 head 对象时受快照影响内部产生的克隆（head 对象）操作 |
| | DELETE | 删除对象 |
| | LOST_REVERT | Peering 完成后，如果 PG 仍然存在丢失（unfound）或者无法恢复的对象，并且确认所有可能包含这些对象权威版本的 OSD 都已经被探测过，那么可以通过如下命令：<br>`ceph pg <pgid> mark_unfound_lost revert\|delete`<br>以回滚或者直接删除的方式对这些对象进行修复。<br>上述命令执行后将触发 PG 针对这些对象进行遍历，为每个对象生成一条新的日志：<br>1）对应 revert 命令，找出对象当前所能获得的最新版本，生成一条新的 LOST_REVERT 日志，并将日志中的 reverting_to 指向该版本。<br>2）对应 delete 命令，直接生成一条新的 LOST_DELETE 日志。<br>PG 批量提交日志后（此时命令执行完成），这些对象的修复可以在后台（按照更新后的日志）进行 |
| | LOST_DELETE | |
| soid | | 对象标识 |
| version(Eversion) | | version 指本次写生效之后对象的版本号。<br>prior_version 指本次写生效之前对象的版本号。<br>reverting_to 指使用 ceph pg <pgid> mark_unfound_lost revert 命令修复对象时，待回滚的版本号 |
| prior_version(Eversion) | | |
| reverting_to(Eversion) | | |
| reqid | | 客户端生成的唯一标识 |
| mtime | | 客户端产生此请求的本地时间 |
| user_version(Version) | | 对象版本号，对客户端可见 |

所有日志使用一个日志队列进行管理，该队列主要成员如表6-9所示。

表6-9　日志队列（pg_log_t）

| 成　员 | 含　义 |
| --- | --- |
| log | 日志队列，新的日志条目总是从队列尾部追加 |
| head | 指向最新日志条目（实时指向log尾部） |
| tail | 指向最老日志条目的前一个条目 |

9.OpContext

为方便起见，PG将所有来自客户端的请求和集群内部诸如执行数据恢复、Scrub等任务产生的请求都统称为op。

引入OpContext的意义如下：

1）单个op可能操作多个对象（指同一个对象的head对象、克隆对象），需要分别记录对应对象上下文的变化。

2）如果op涉及写操作，那么会产生一条或者多条新的日志。

3）如果op涉及异步操作，那么需要注册一个或者多个回调函数。

4）收集op相关的统计，例如读写次数、读写涉及的字节数等，并周期性地上报给Monitor，用于监控集群IOPS、带宽等。

OpContext的主要成员如表6-10所示。

表6-10　OpContext

| 成员 | 含　义 |
| --- | --- |
| at_version(Eversion) | 如果 op 包含写操作，那么 PG 将为 op 生成一个 PG 内唯一的序列号。该序列号单调递增，用于后续（例如 Peering）对本次写操作进行追踪和回溯 |
| clone_obc(ObjectContext) | 克隆对象上下文，仅在需要时加载 |
| log | op 产生的所有日志集合。<br>日志是基于对象的。单个 op 针对 head 对象的所有操作只会产生一条日志，但是快照机制的引入使得执行 op 过程中可能创建克隆对象，需要为其生成单独的日志。这解释了这里为什么是一个日志集合 |
| modified_ranges | 如果 op 包含写操作，记录对象本次被改写的数据范围 |
| new_obs(ObjectState) | op 执行之后[注]，新的对象状态和 SS 属性 |
| new_snapset(SnapSet) | |
| obc(ObjectContext) | 对象上下文 |
| obs(ObjectState) | op 执行之前，对象状态 |
| on_applied | 各类回调上下文，满足各自条件时执行[注] |
| on_committed | |
| on_finish | |
| on_success | |
| snapc(SnapContext) | 快照上下文，每次收到 op 时，基于 op 或者 PGPool 更新 |
| snapset(SnapSet) | op 执行之前，对象关联的 SS 属性 |

注：准确地说，是在Primary完成PG事务封装之后。

注：例如，on_applied一般在所有副本的本地事务都写入日志盘后执行；on_committed一般在所有副本的本地事务都写入数据盘后执行；on_success与on_finish通常在事务已经完成（也就是on_committed已经

执行）后执行；on_success一般可用于执行Watch/Notify相关的操作；on_finish通常用于删除OpContext。

10.RepGather

如果op包含写操作，通常情况下需要由Primary主导，在副本之间发起分布式写。RepGather用于（取代op）追踪该分布式写在副本之间的完成情况，其主要成员如表6-11所示。

表6-11　RepGather

| 成员 | 含　义 |
| --- | --- |
| all_applied | 所有副本已经将本地事务写入了日志盘 |
| all_committed | 所有副本已经将本地事务写入了数据盘 |
| on_applied | 同 OpContext 中的 on_applied |
| on_committed | 同 OpContext 中的 on_committed |
| on_finish | 同 OpContext 中的 on_finish |
| on_success | 同 OpContext 中的 on_success |
| rep_aborted | RepGather 被异常终止，例如 PG 收到新的 OSDMap 并且需要切换至一个新的 Interval |
| rep_done | RepGather 正常完成 |

了解上述预备知识之后，我们可以着手对读写流程进行分析，其大致分为如下几个阶段：

1）客户端基于对象标识中的32位哈希值，通过stable_mod找到存储池中承载该对象的PGID，然后使用该PGID作为CRUSH输入，找到对应PG当前Primary所在的OSD并发送读写请求。

2）OSD收到客户端发送的读写请求，将其封装为一个op，并基于其携带的PGID将其转发至对应的PG。

3）PG收到op后，完成一系列检查，所有条件均满足后，开始真正执行op。

4）如果op只包含读操作，那么直接执行同步读（对应多副本）或者异步读（对应纠删码），等待读操作完成后由Primary向客户端应答。

5）如果op包含写操作，则由Primary基于op生成一个针对原始对象操作的PG事务，然后将其提交至PGBackend，由后者按照备份策略转化为每个副本真正需要执行的本地事务，并进行分发。当Primary收到所有副本的写入完成应答之后，对应op执行完成，由Primary向客户端回应写入完成。

### 6.2.1 消息接收与分发

每个客户端的读写请求首先被OSD封装成一个op，然后按其携带的PGID投递至某个op_shardedwq队列进行处理。

顾名思义，op_shardedwq是一种工作队列（Work Queue），sharded关键字表明其内部可以存在多个队列。op_shardedwq最终关联osd_op_tp线程池，由池中线程真正对op进行处理。

通常情况下，osd_op_tp会包含多个服务线程，具体数量由系统按照不同的后端存储介质类型自动进行调整。假定op_shardedwq中实际工作队列数目为s，每个工作队列需要安排t个服务线程，则这s个工作队列与s*t个服务线程之间可以通过如下方式建立联系：

·将osd_op_tp中的所有服务线程依次从0开始编号：0，1，2，…，s*t-1。

·将op_shardedwq中的所有工作队列依次从0开始编号：0，1，2，…，s-1。

·将服务线程编号针对s取模，得到一个[0，1，2，…，s-1]内的结果，然后将其作为op_shardedwq中某个工作队列的索引。

如果op_shardedwq包含多个工作队列，为了保证负载均衡，所有op应该尽可能被均匀地派发至每个工作队列。由CRUSH相关章节的介绍

我们知道，PG（副本）被映射至OSD的过程随机，因此每个OSD承载的PG，它们的PGID也呈随机分布。简单将PGID针对队列个数取模，即可实现PG与某个op_shardedwq队列的间接绑定，并且此时每个队列绑定的PG个数也大致相同。采用同样的模运算，OSD即可准确将op投递至其归属PG所在的工作队列，进一步地，我们假定每个PG处理op的概率相等（通常情况下如此），则每个队列处理op的速率也大致相等，即每个队列的负载是基本均衡的。

除此之外，由于归属同一个PG的op都进入同一个工作队列排队，这样做一方面可以避免乱序，另一方面也便于实施优先级控制。

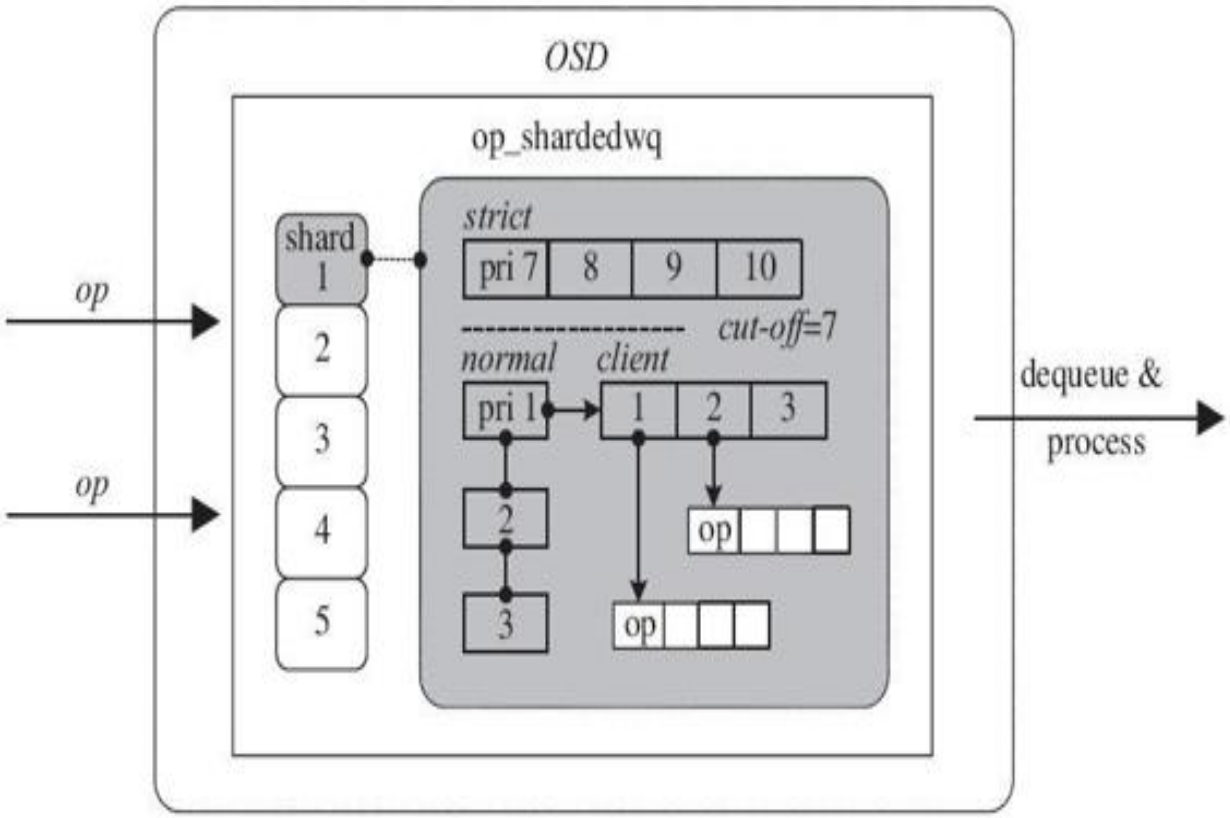



图6-1　op_shardedwq内部构造

pri表示优先级；strict表示严格优先级队列；normal表示普通优先级队列

如图6-1所示，op_shardedwq内部工作队列可以采用多种实现方式，默认为WeightedPriorityQueue（wpq），这是一种基于权重的优先级调度队列。队列内部首先按照优先级被划分为多个优先级队列，然后每个优先级队列按照op携带的客户端地址再次划分为若干会话子队列。

所有优先级队列按照其优先级与系统预先设定的阈值（图6-1中的cut-off）之间的关系，又可细分为严格优先级队列和普通优先级队列两种类型。所有优先级不小于该阈值的op进入严格优先级队列，反之则进入普通优先级队列。两种队列的入队方法相同：首先按照op携带的优先级，找到对应的优先级队列；然后再按op携带的客户端地址找到其归属的会话子队列；最后将op加入会话子队列的队尾。两者的区别在于出队方法。

针对所有严格优先级队列，总是严格从当前优先级最高的队列开始出队（这也是严格优先级队列名称的由来）。

针对所有普通优先级队列，则采用基于权重的Round-Robin调度方式进行出队：出队时，选择哪个普通优先级队列是完全随机的，但是选择结果与队列的优先级强相关。优先级越高，则被选中的概率越大；反之优先级低的队列被选中的概率也低。容易理解，采用这种出队方式的好处在于永远不会“饿死”来自某些低优先级客户端的op，同时能基于优先级（此时优先级充当了权重的角色）对op的处理速度进行定量控制。例如两个客户端的优先级分别为1：2，那么在此模式下理论上它们能够获得的IOPS（假定I/O大小相同）也为1：2。

由于每个优先级队列可能包含多个会话子队列，为了避免同一优先级下有来自不同客户端的op“饿死”，优先级队列内部维护了一个循环指向每个会话子队列的指针。op只能从指针当前指向的会话子队列队头出队（会话子队列是严格的FIFO队列）。出队之后，指针会从当前会话子队列自动指向下一个会话子队列。

如果同时存在两种类型的非空优先级队列，按照定义，任意严格优先级队列中的op，其优先级比所有位于普通优先级队列中的op都高，因此我们总是优先出队严格优先级队列中的op。只有所有严格优先级队列都为空时，才能继续出队普通优先级队列中的op。

当op成功地从op_shardedwq出队后，相应的服务线程会被唤醒，真正对其进行处理。整个处理流程是一个复杂的函数调用链，我们接下来依次介绍（默认忽略了Cache-Tier相关的处理）。

### 6.2.2 do_request

do_request作为PG处理op的第1步，主要完成一些全局（PG级别的）检查：

1）如果op携带的Epoch比PG当前持有的更新，那么需要等待PG完成OSDMap同步之后才能继续处理。

2）op能否被直接丢弃？例如满足如下条件之一，op将会被直接丢弃：

·op对应的客户端链路已经断开。

·收到op时，PG当前已经切换到一个更新的Interval（即PG此时的same_interval_since比op携带的Epoch要大，后续客户端会重发）。

·op在PG分裂之前发送（后续客户端会重发）。

3）PG自身状态，如果PG不是Active状态，op可能会被阻塞，需要等待PG变为Active状态之后才能被正常处理。

具体处理逻辑如图6-2所示。

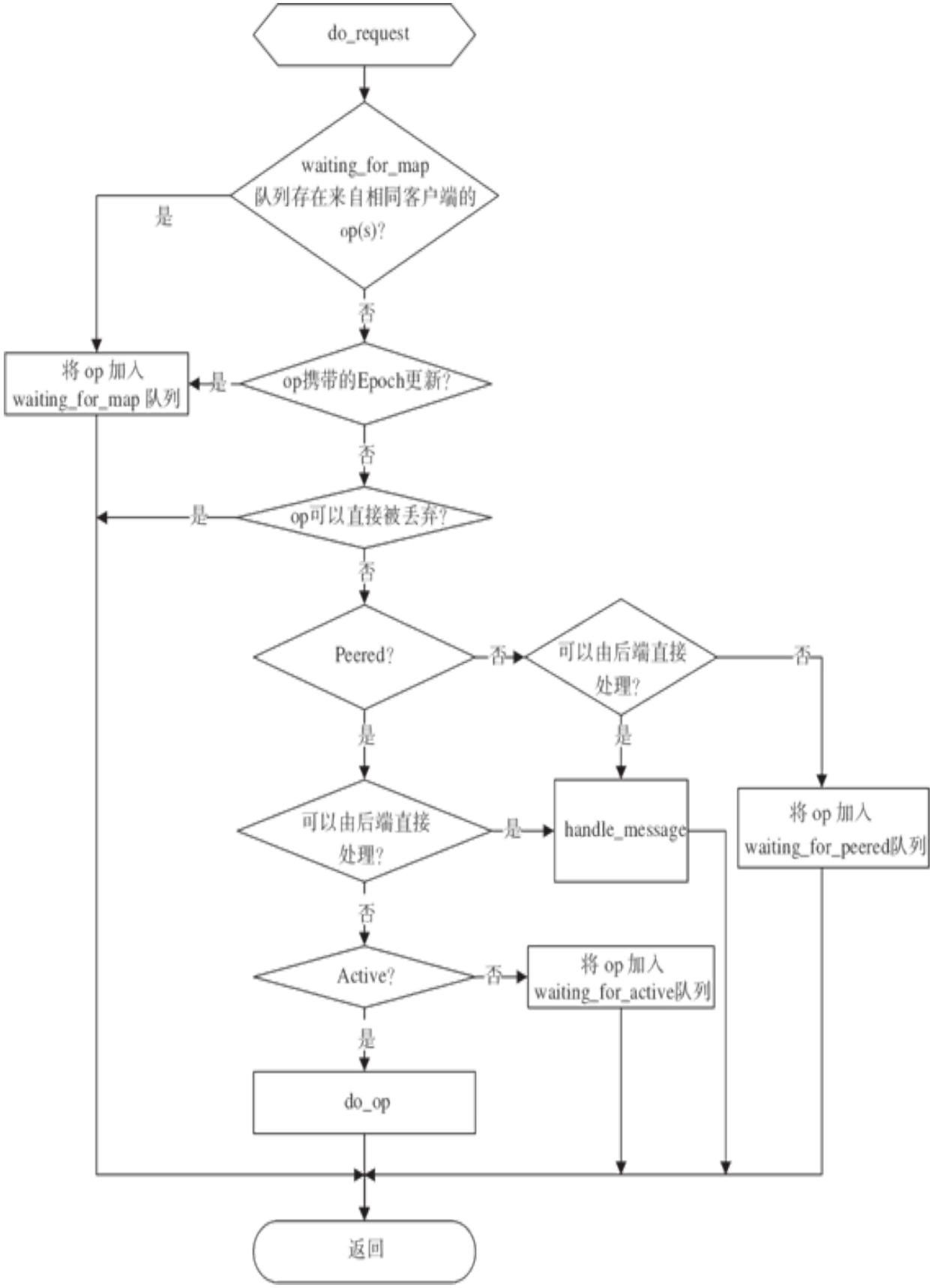

do_request
waiting_for_map
队列存在来自相同客户端的
op(s)?
是
否
将 op 加入
waiting_for_map 队列
是
op携带的Epoch更新?
否
是
op可以直接被丢弃?
否
Peered?
否
可以由后端直接
处理?
否
是
是
可以由后端直接
处理?
是
handle_message
将 op 加入
waiting_for_peered队列
否
Active?
否
将 op 加入
waiting_for_active队列
是
do_op
返回

图6-2　do_request处理逻辑

参考图6-2，op可能由于各种各样的原因需要被推迟处理，为此PG内部维护了多种op重试队列，它们的含义如表6-12所示。

表6-12　PG内部的op队列

| 队列名称 | 入队条件 |
| --- | --- |
| waiting_for_map | 收到op时，已经有来自同一个客户端的op存在于此队列之中；或者op携带的Epoch大于PG当前的Epoch |
| waiting_for_peered | PG不是Peered或者Active状态（例如PG处于Down或者Incomplete状态） |
| waiting_for_active | PG不是Active状态（例如PG处于Peered状态） |
| waiting_for_unreadable_object | op操作Primary上的降级对象或者对象不可读（例如纠删码的情况下，对象的一个或者多个分片损坏，并且无法通过解码恢复） |
| waiting_for_degraded_object | op包含写操作并且操作降级对象 |
| waiting_for_scrub | op操作的对象正在被Scrub。<br>注意：由于PG能够执行Scrub的前提是PG处于Active + Clean状态，所以PG总是最后检查此队列 |

为了保证op之间不会乱序，所有队列都被设计成FIFO队列并且队列之间严格有序。当某个限制条件被解除后，PG会触发相关队列的op出列，重新进入op_shardedwq排队，等候再次执行。

### 6.2.3 do_op

通过do_request校验之后，PG可以对op做进一步处理。特别地，如果确认是来自客户端的op，那么PG将通过do_op对其进行处理，处理逻辑如图6-3所示。

参考图6-3，do_op主要完成如下工作：

1）按照op携带的操作类型（单个op可以包含多个操作），初始化op中的各种标志位，例如：

·CEPH_OSD_RMW_FLAG_READ：说明op携带读操作。

·CEPH_OSD_RMW_FLAG_WRITE：说明op携带写操作。

·CEPH_OSD_RMW_FLAG_RWORDERED：说明需要对op进行读写保序。

2）完成op的合法性校验，出现下列情况之一，op将被认为不合法：

·PG不包含op所携带的对象。

·op携带了CEPH_OSD_FLAG_PARALLELEXEC标志，指示op之间可以并发执行。

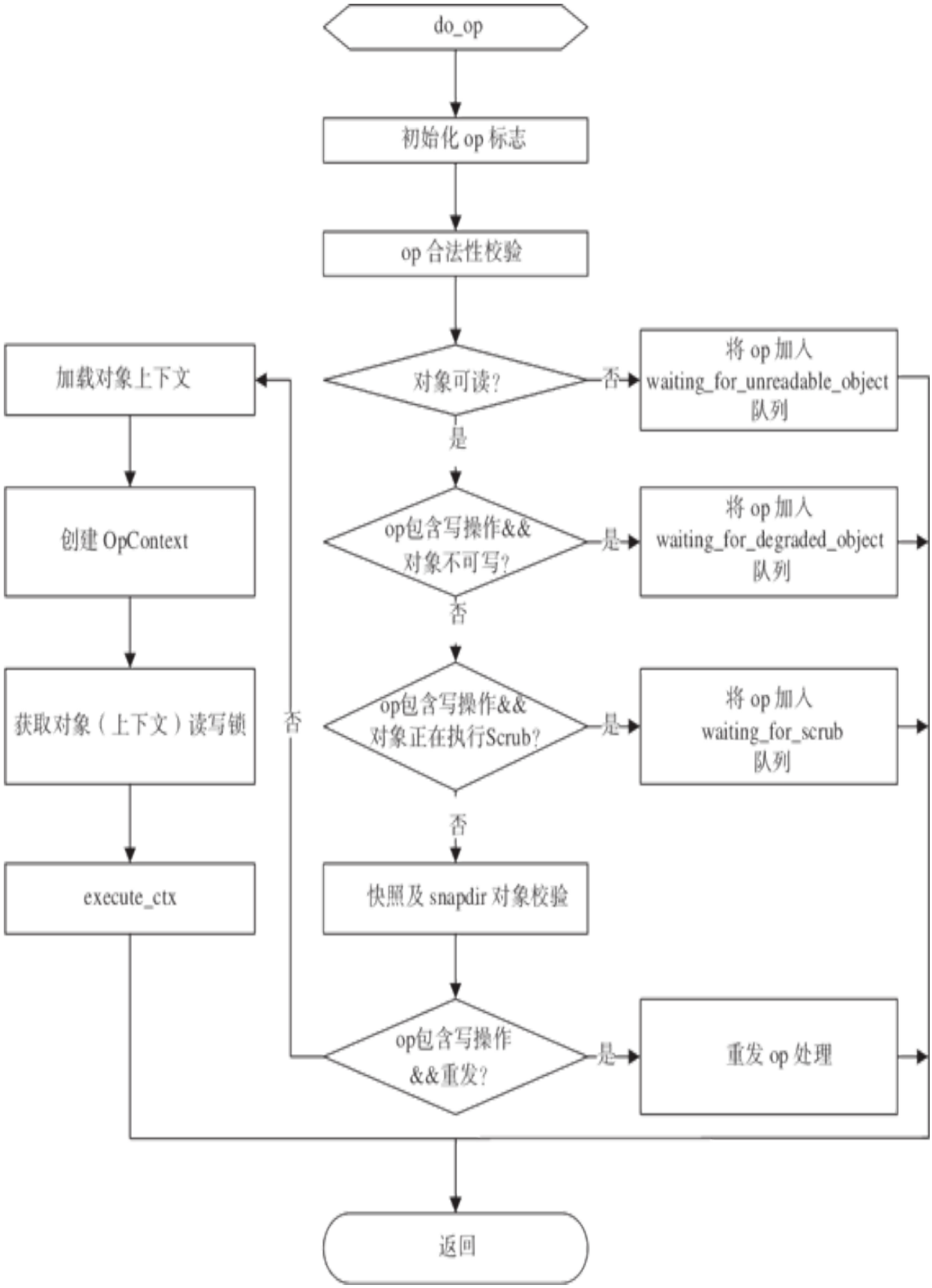

do_op
初始化 op 标志
op 合法性校验
对象可读?
否
将 op 加入 waiting_for_unreadable_object 队列
是
op包含写操作&& 对象不可写?
是
将 op 加入 waiting_for_degraded_object 队列
否
op包含写操作&& 对象正在执行Scrub?
是
将 op 加入 waiting_for_scrub 队列
否
快照及 snapdir 对象校验
op包含写操作 &&重发?
是
重发 op 处理
否
加载对象上下文
创建 OpContext
获取对象（上下文）读写锁
execute_ctx
返回

图6-3　do_op处理逻辑

·op携带了CEPH_OSD_FLAG_BALANCE[LOCALIZE]_READS标志，但是处理op的PG不是当前Up或者Acting中的成员。

·op对应的客户端未被赋予执行op所请求操作的权限（例如客户端仅被授予可读权限，但是op携带了写操作）。

·op携带对象名称、key或者命名空间长度超过最大限制。

·op对应的客户端被列入黑名单。

·op在存储池被标记为Full之前发送（PG通过比对op携带的Epoch与存储池标记为Full时的Epoch感知）并且没有携带CEPH_OSD_FLAG_FULL_TRY或者CEPH_OSD_FLAG_FULL_FORCE标志。

·op没有携带CEPH_OSD_FLAG_FULL_TRY标志并且OSD已经处于Failsafefull状态。

·op包含写操作并且企图访问快照对象。

·op包含写操作并且一次写入的数据量过大（例如超过90MB）。

3）检查op携带的对象是否不可读或者处于降级状态或者正在被Scrub，是则加入对应队列进行排队。

4）检查op是否为重发。

5）获取对象上下文（ObjectContext），创建OpContext对op进行追踪。

由于对象上下文保存了对象的关键属性，同时表明对应对象是否仍然存在，因此真正执行op之前，必须先查找对象上下文，具体如图6-

4所示。

与直接查找head对象相比，按照快照序列号查找快照对象（实际上会被映射为克隆对象）的难度在于一个克隆对象可以对应多个快照，因此首先需要根据快照序列号定位到某个特定克隆对象，然后通过分析其位于OI中的snaps属性才能进一步判定对应的快照序列号是否确实包含于克隆对象之中。

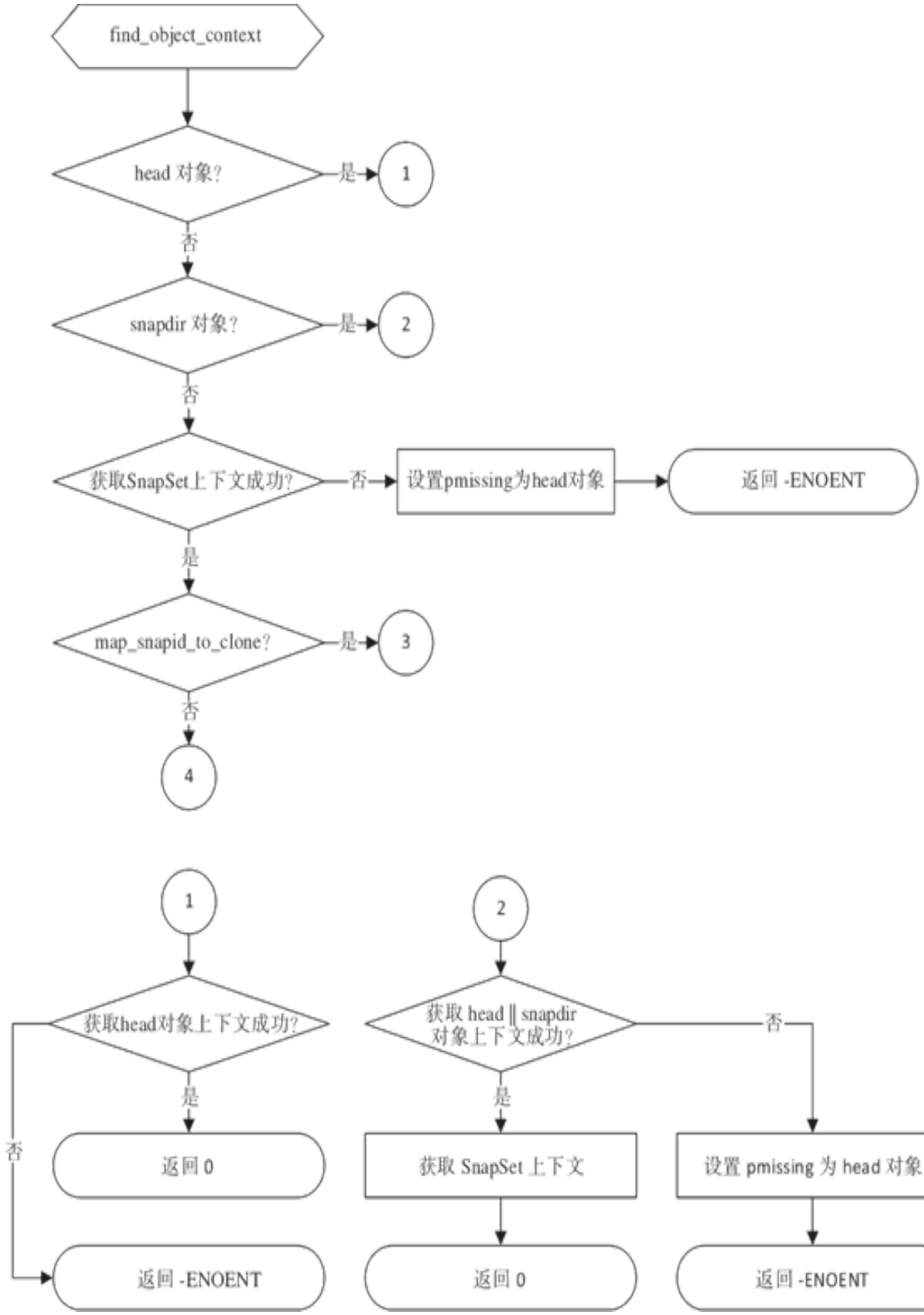

find_object_context
head 对象?
是
1
否
snapdir 对象?
是
2
否
获取SnapSet上下文成功?
否
设置pmissing为head对象
返回 -ENOENT
是
map_snapid_to_clone?
是
3
否
4
1
获取head对象上下文成功?
是
返回 0
否
返回 -ENOENT
2
获取 head ‖ snapdir
对象上下文成功?
是
获取 SnapSet 上下文
返回 0
否
设置 pmissing 为 head 对象
返回 -ENOENT

图6-4　查找对象上下文

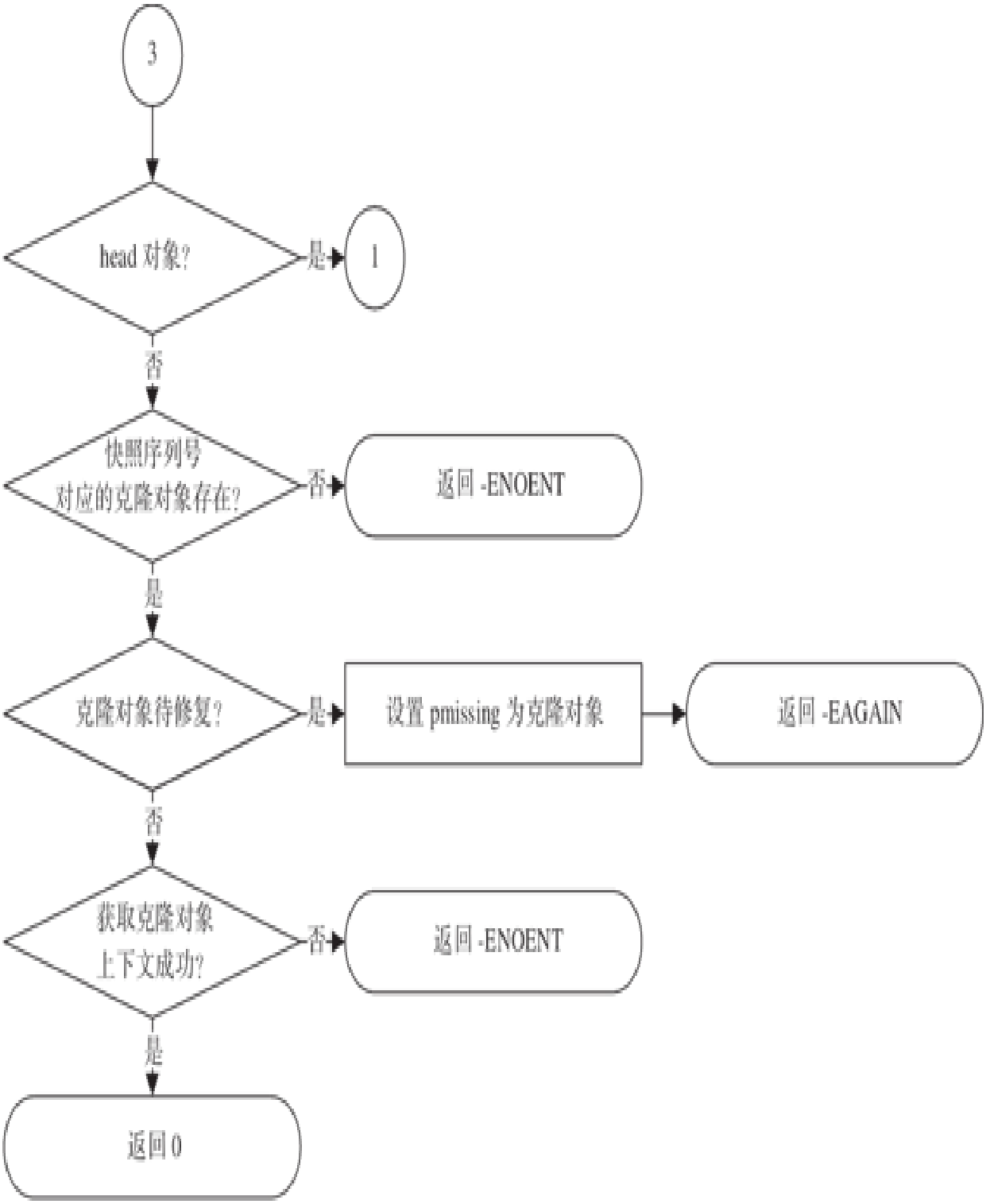



图6-4　（续）

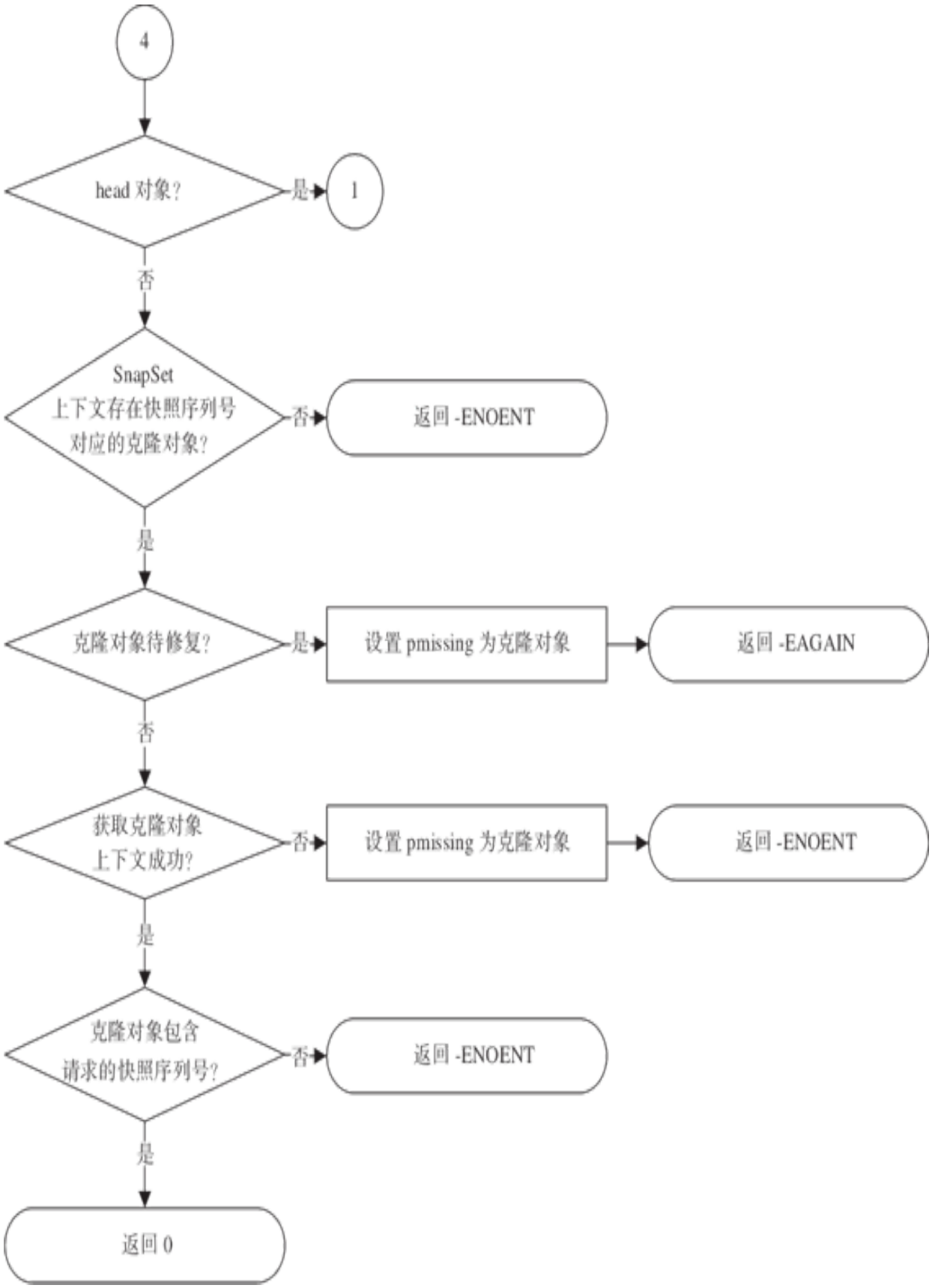

4
head 对象?
是
1
否
SnapSet
上下文存在快照序列号
对应的克隆对象?
否
返回 -ENOENT
是
克隆对象待修复?
是
设置 pmissing 为克隆对象
返回 -EAGAIN
否
获取克隆对象
上下文成功?
否
设置 pmissing 为克隆对象
返回 -ENOENT
是
克隆对象包含
请求的快照序列号?
否
返回 -ENOENT
是
返回 0

图6-4 （续）

通过快照序列号定位克隆对象的具体过程如下：由于对象SS属性中的clones集合保存了本head对象所关联的所有克隆对象，并且该集合按照每个克隆对象所对应的最新快照序列号进行升序排列，因此查找过程可以总结为：从clones集合中的第一个元素开始顺序遍历，直至找到第一个满足不小于指定快照序列号的克隆对象为止，如图6-5所示。


图6-5 通过快照序列号查找特定的克隆对象

clones[2](=8)对应的克隆对象实际上关联6、7、8三个快照，因此snapid=7包含其中

查找对象上下文最终可能产生0、-ENOENT和-EAGAIN三种结果。如果返回0，表明查找成功；如果返回-ENOENT，表明查找的对象不存在；如果返回-EAGAIN，表明查找的对象待修复，需要等待对象修复完成后重试op。

### 6.2.4 execute_ctx

成功获取对象上下文之后，可以通过execute_ctx正式执行OpContext，具体过程如图6-6所示。

参考图6-6，如果op包含写操作，execute_ctx将首先基于当前快照模式，更新OpContext中的快照上下文（SnapContext）；如果是自定义快照模式，直接基于op携带的快照信息更新；否则基于PGPool中的SnapContext更新。由于写请求是异步执行的（纠删码存储池的读请求也是异步执行的），在调用execute_ctx之前，可能还需要先获取对象上下文的读写锁，防止与之前尚未完成的写请求冲突，产生乱序。

execute_ctx中最关键的步骤是prepare_transaction。顾名思义，它完成事务准备。在BlueStore相关章节中，我们已经介绍过事务是用于保证数据一致性最常用的手段。与单个OSD由ObjectStore（例如BlueStore）这类事务型本地对象存储系统负责保证数据一致性不同，对存储池（或者说整个集群）而言，由于数据一般需要在多个节点备份，为了保证节点间的数据一致性（即分布式一致性），则必须依赖PG实现分布式的事务语义。因此，客户端针对原始对象的读写请求，首先将被转化为一个PG事务，用于保证分布式一致性；然后再由Primary统一转化为各个副本的本地事务（即ObjectStore事务），用于保证副本的本地一致性。

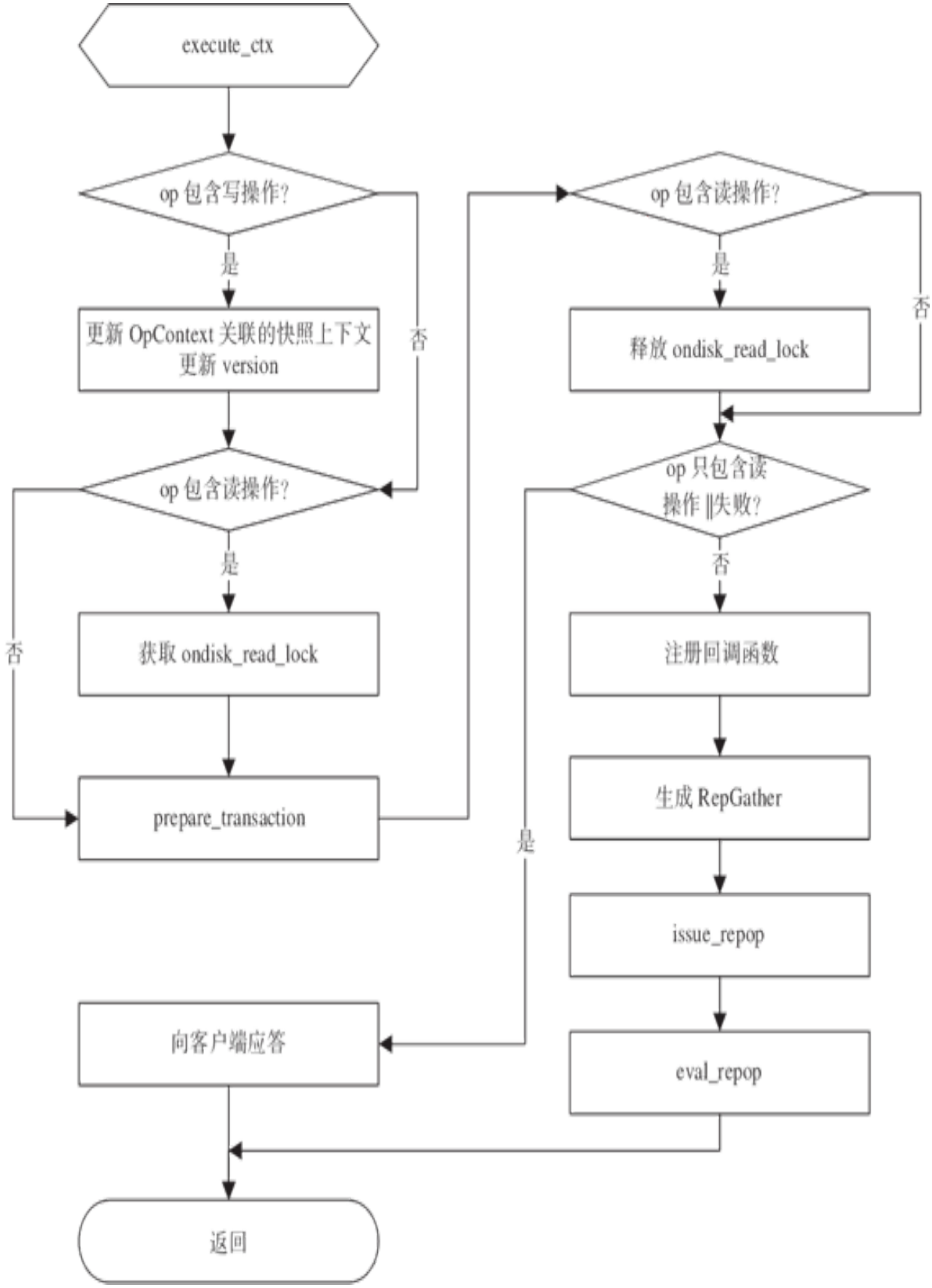

execute_ctx
op 包含写操作?
是
否
更新 OpContext 关联的快照上下文
更新 version
op 包含读操作?
是
否
获取 ondisk_read_lock
prepare_transaction
op 包含读操作?
是
否
释放 ondisk_read_lock
op 只包含读
操作 ||失败?
是
否
注册回调函数
生成 RepGather
issue_repop
eval_repop
向客户端应答
返回

图6-6　execute_ctx具体过程

由于涉及跨节点（OSD）协作，在被转化为副本的本地事务之后，仍由Primary牵头，将OpContext升级为RepGather，并通过issue_repop在副本之间进行分发。之后由每个副本的本地事务完成应答消息驱动，Primary不断通过eval_repop评估RepGather的当前进展，直至事务在副本之间的同步最终完成。

1.事务准备

针对多副本而言，由于每个副本保存的对象完全相同，所以由Primary生成的PG事务也可以作为每个副本的本地事务直接执行，即多副本模式下实际上并不需要区分PG事务和本地事务。引入纠删码之后，由于每个副本都只保存原始对象独一无二的分片，所以针对原始对象的操作与针对每个分片的操作必然存在差别，生成的事务也有所不同。

（1）PG事务

PG事务（PGTransaction）记录了一系列直接以原始对象（即从客户端视角，不考虑备份策略）为目标的写操作，后续可以为不同的PGBackend翻译成ObjectStore能够理解的本地事务并直接执行。

与本地事务不同，PG事务（以及相关的日志）主要用于保障副本之间的数据一致性，其公共接口如表6-13所示。

表6-13　PGTransaction公共接口

| 接口名称 | 含　义 |
| --- | --- |
| clone | 对象克隆 |
| clone_range | 范围克隆 |
| create | 创建新对象 |
| nop | 空操作，用于强制对象内的数据存盘 |
| omap_clear | 删除对象所有 omap 条目 |
| omap_rmkeys | 批量删除 omap 条目 |
| omap_setheader | 设置 omap header |
| omap_setkeys | 批量添加 omap 条目 |
| remove | 删除对象 |
| rename | 对象重命名。<br>注意：待重命名的对象必须是一个临时对象。<br>rename 操作的典型应用场景如：针对对象执行修复时，如果通过一次传输（Pull/Push）不能完成，则此时需要借助一个临时对象来进行中转，而不能直接针对原始对象操作，否则存在将原始对象写坏的风险。当全部数据传输完成之后，最终通过将原始对象删除，同时将临时对象重命名为原始对象来完成修复 |
| rmattr | 删除单个扩展属性 |
| setattr | 设置单个扩展属性 |
| setattrs | 批量设置扩展属性 |
| set_alloc_hint | 为对象设置一些特殊属性，例如客户端访问模式等提示信息 |

（续）

| 接口名称 | 含　义 |
| --- | --- |
| truncate | 截断对象[⊖] |
| write | 写 |
| zero | 使用全0填充对象指定范围内的数据 |

注：也支持truncate-up操作，效果等同于append，但是使用全0填充。

除了表6-13中这些接口之外，PGTransaction还特别提供了一个名为safe_create_traverse的接口，方便各类PGBackend安全地将其转化为每个副本的本地事务。之所以称之为safe，是因为如果涉及多对象操作，safe_create_traverse能够保证这些操作仍然以合理的顺序转化为本地事务中的操作，而不会发生乱序（例如先修改head对象，再创建克隆对象）。

每个op通过如下3个步骤被转化为一个完整的PG事务：do_osd_ops负责将op携带的每个操作转化为PGTransaction中的操作；make_writable负责处理快照相关的逻辑；finish_ctx则检查是否需要创建或者删除snapdir对象，生成日志，并刷新对象的OI与SS属性。

（2）do_osd_ops

图6-7展示了通过do_osd_ops将op携带的write操作转化为PGTransaction的过程。

由图6-7可见，op与PGTransaction中的操作并不总是一一对应的，例如这里单一的write操作可能转化为PGTransaction中的create+write（复合）操作。

### （3）make_writable

do_osd_ops执行完成之后，如果op针对head对象修改，那么需要通过make_writable来判定是否需要预先针对head对象执行克隆。如图6-8所示，判断head对象是否需要执行克隆的方式如下：取对象当前的SnapSet，与OpContext携带的实时SnapContext进行比较，如果SnapSet中最新的快照序列号比SnapContext中最新的快照序列号小，说明自上一次快照之后，又产生了新的快照，此时不能直接针对head对象进行修改，而是需要先执行克隆（默认执行全对象克隆）。如果SnapContext携带了多个新的快照序列（例如自某次快照产生后，很长时间内没有针对本对象执行过任何修改操作，而中间又多次执行了快照操作），那么所有比SnapSet中更新的快照序列号都将关联至同一个克隆对象（即后续针对这些快照执行回滚时，都将回滚至同一个克隆对象）。


do_osd_ops
(CEPH_OSD_OP_WRITE)
纠删码存储池相关的对齐校验
truncate_seq 有效?
是
对象存在?
否
是
否
offset + length 超过
对象大小?
生成 truncate 事务
更新对象大小为 truncate_size
是
否
调整 length
使得 offset + length 不超过对象大小
更新对象当前的
truncate_seq/size
offset 及 length 合法性校验
对象存在?
否
设置 new_obs 中 exists 属性为
true，生成 create 事务
是
length 为 0?
是
offset 大于对象大小?
否
生成 nop 事务
否
是
生成 write 事务
生成 truncate 事务
满对象写或者追加写?
否
是
清除数据校验和
计算并更新数据校验和
更新对象 modified_ranges
（和本次修改范围取并集）
更新对象大小
返回

图6-7　do_osd_ops(write)过程

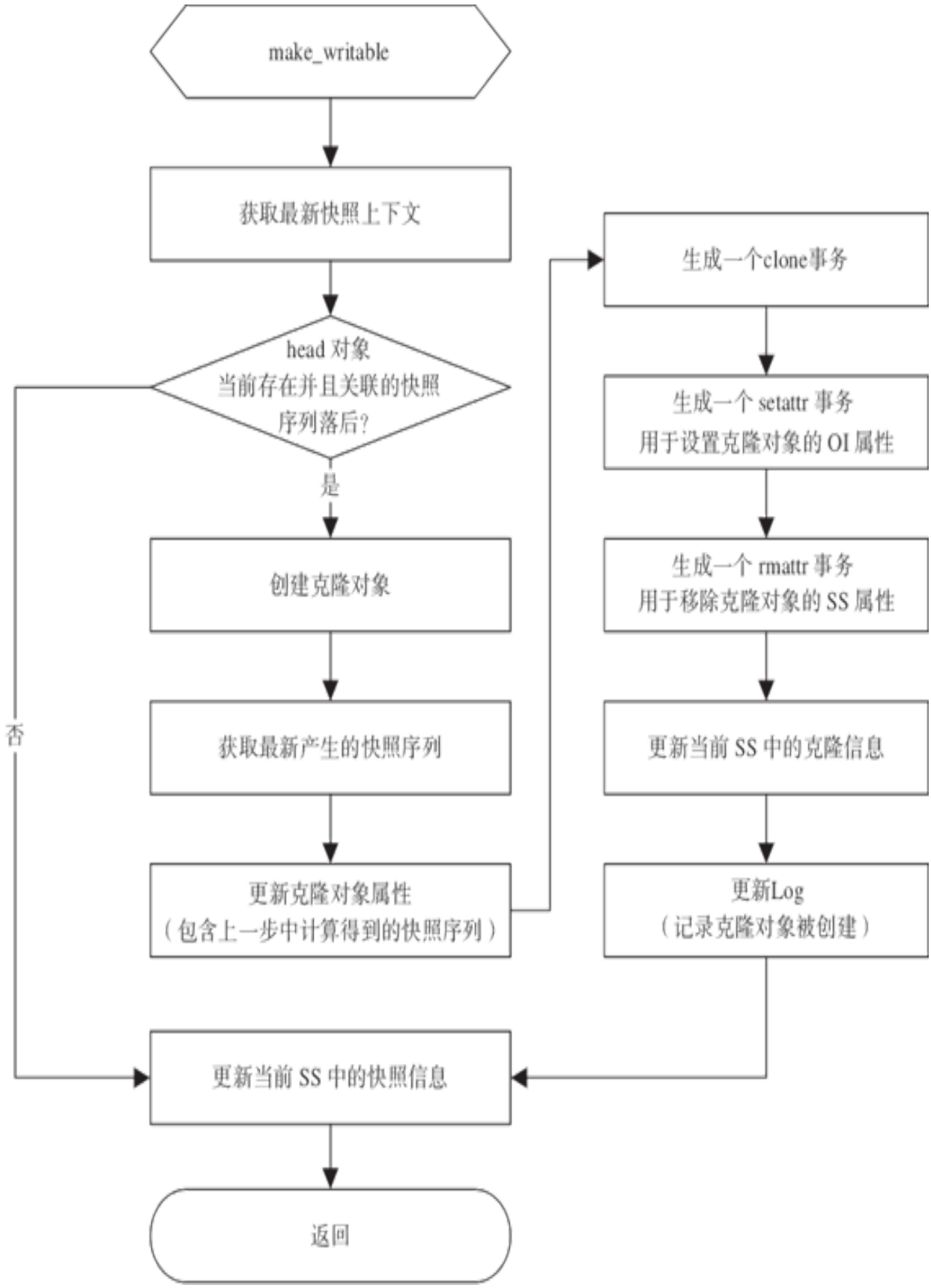

make_writable
获取最新快照上下文
head 对象
当前存在并且关联的快照
序列落后?
是
否
创建克隆对象
获取最新产生的快照序列
更新克隆对象属性
（包含上一步中计算得到的快照序列）
生成一个clone事务
生成一个 setattr 事务
用于设置克隆对象的 OI 属性
生成一个 rmattr 事务
用于移除克隆对象的 SS 属性
更新当前 SS 中的克隆信息
更新Log
（记录克隆对象被创建）
更新当前 SS 中的快照信息
返回

图6-8　make_writable流程

这里存在一种特殊情况：如果当前操作为删除head对象，并且该对象自创建之后没有经历过任何修改（即此时SnapSet为空），也需要对该head对象正常执行克隆后再删除。在Luminous版本之前，head对象将被真正删除并且后续将创建一个snapdir对象来转移（更新后的）SS信息。而Luminous版本则只需要简单设置head对象的FLAG_WHITEOUT标志位，来标识head对象逻辑上已经不存在即可。

创建克隆对象时，需要同步更新SS属性中的以下相关信息：

·在clones集合中记录当前克隆对象中的最新快照序列号。

·在clone_size集合中更新当前克隆对象大小，由于默认使用全对象克隆，所以克隆对象大小为执行克隆时head对象的实时大小。

·在clone_overlap集合中记录当前克隆对象与前一个克隆对象之间的重合部分，后续用于Recovery加速等。

如果确定需要执行克隆，则由make_writable为克隆对象生成一条独立的日志。

### （4）finish_ctx

顾名思义，finish_ctx完成事务准备阶段的收尾工作。

图6-9展示了finish_ctx的具体工作流程。由于PG事务的内存版本已经准备完毕，此时可以生成原始对象的操作日志，并安全地更新对象上下文中OI与SS属性。

2.事务分发与同步

完成（PG）事务准备之后，针对纯粹的读操作而言，如果是同步读，例如多副本，OpContext已经执行完毕，此时可以直接向客户端应答；如果是异步读，例如纠删码，则将op加入异步读队列，等待异步读完成之后再向客户端应答。

如果op包含了写操作，则由Primary负责执行副本间的本地事务分发与同步，这通过将op对应的OpContext转交给RepGather来完成。

参考表6-11，RepGather大部分内容直接来自于OpContext，主要区别在于RepGather增加了一些用于副本间进度同步的字段，例如all_applied和all_committed，分别表示Primary是否收到了所有副本将本地事务写入日志盘和数据盘的应答。

对多副本而言，由于每个副本保存的内容完全相同，所以将PG事务转化为每个副本本地事务的过程比较简单，多数情况下直接通过safe_create_traverse完成参数传递即可（PG事务与ObjectStore事务接口基本相同）。纠删码则比较复杂，特别地，当涉及覆盖写时，如果改写的部分不足一个完整条带（指写入的起始地址或者数据长度没有进行条带对齐），则需要执行RMW，因此期间会多次调用safe_create_traverse用于执行补齐读、重新生成完整条带和重新计算校验块等，最后才能真正转换为每个副本的本地事务。

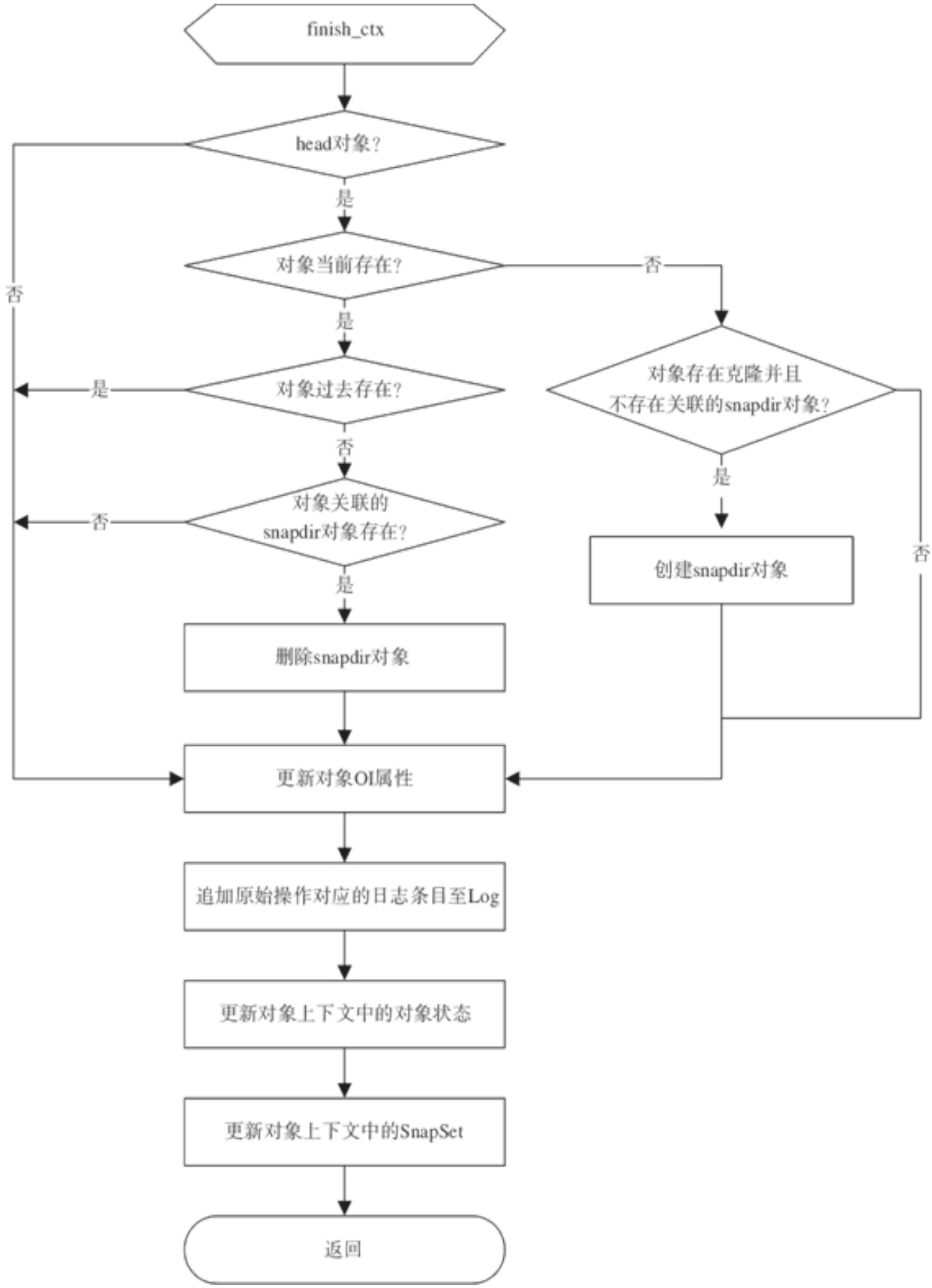

finish_ctx
head对象？
是
否
对象当前存在？
否
是
对象过去存在？
是
否
对象关联的
snapdir对象存在？
否
是
对象存在克隆并且
不存在关联的snapdir对象？
是
否
创建snapdir对象
删除snapdir对象
更新对象OI属性
追加原始操作对应的日志条目至Log
更新对象上下文中的对象状态
更新对象上下文中的SnapSet
返回

图6-9　finish_ctx流程

完成事务转换之后，仍由Primary负责分发，此后每个副本将异步执行各自的本地事务，为此需要由Primary预先向RepGather注册几类回调函数，用于指示RepGather最终完成时的动作，例如处理Watch/Notify、向客户端应答等。之后，随着每个副本向Primary回复写入日志盘完成应答（applied）与写入数据盘完成应答（committed），Primary不断对RepGather当前进展进行评估，直至全部副本完成，此时可以依次执行回调函数，并最终清理和释放RepGather。

## 6.3 状态迁移

前面我们介绍了客户端的读写流程，本节我们以Peering为例介绍PG如何通过状态迁移完成故障切换、数据恢复等复杂任务。

状态迁移常用于表征PG在不同任务之间进行了切换（例如，由Active+Clean变为Active+Clean+Scrubbing+Deep状态，说明PG当前正在或者即将执行Deep-Scrub）或者任务进度发生了变化（例如，由Active+Degraded变为Active+Clean状态，说明PG已经完成了所有降级对象的修复）。

上述状态是指能够为普通用户直接感知的外部状态。实现上，PG内部通过状态机来驱动PG在不同外部状态之间进行迁移，因此，相应地PG有内外两种状态。

我们首先介绍PG的外部状态，这是普通用户接触和了解PG最直观的途径，典型的如通过ceph–s命令可以实时追踪集群中所有PG外部状态的概况。一些常见的PG外部状态如表6-14所示。

表6-14　常见的PG外部状态

| 状态 | 含　义 |
| --- | --- |
| Activating | Peering 即将完成，正在等待所有副本同步并固化 Peering 结果（Info、Log 等） |
| Active | PG 可以正常处理来自客户端的读写请求 |
| Backfilling | 正在执行 Backfill |
| Backfill-toofull | 副本所在 OSD 空间不足（Backfillfull），Backfill 流程当前被挂起 |
| Backfill-wait | 等待 Backfill 资源预留完成 |
| Clean | PG 当前不存在降级对象，Acting 与 Up 内容一致，并且大小等于存储池副本数 |

（续）

| 状态 | 含　义 |
| --- | --- |
| Creating | PG 正在被创建 |
| Deep | PG 正在或者即将执行 Deep-Scrub |
| Degraded | PG 存在降级对象，或者 Acting 规模小于存储池副本数（但是不小于存储池最小副本数） |
| Down | Peering 过程中，PG 检测到某个不能被跳过的 Interval 中，当前仍然存活的副本不足以完成数据恢复 |
| Incomplete | Peering 过程中，由于无法选出权威日志或者通过 choose_acting 选出的 Acting 后续不足以完成数据恢复（例如针对纠删码，存活的副本数小于 k 值）等，导致 Peering 无法正常完成 |
| Inconsistent | PG 通过 Scrub 检测到某个或者某些对象在副本之间出现了不一致 |
| Peered | Peering 已经完成，但是当前 Acting 中副本个数小于存储池最小副本数 |
| Peering | PG 正在进行 Peering |
| Recovering | PG 正在后台按照 Peering 结果对降级对象进行修复 |
| Recovery-wait | 等待 Recovery 资源预留完成 |
| Remapped | Up 与 Acting 不一致 |
| Repair | 已启用 Scrub/Deep-Scrub 自动修复功能 |
| Scrubbing | PG 正在执行 Scrub |
| Stale | Monitor 检测到当前 Primary 所在的 OSD 宕掉并且后续没有发生切换，或者 Primary 超时未向 Monitor 上报 PG 相关的统计信息（例如出现临时性的网络拥塞） |
| Undersized | 当前 Acting 中副本个数小于存储池副本数（但是不小于存储池最小副本数） |

需要注意的是，表6-14中这些外部状态并不都是互斥的，某些时刻PG可能处于多个状态的叠加之中，例如Active+Clean表明PG一切正常，Active+Degraded+Recovering表明PG存在降级对象并且正在执行修复等。

### 6.3.1 状态机概述

PG外部状态的变化最终通过其内部状态机进行驱动。该状态机为一种基于事件驱动的有限状态机。集群拓扑或者状态的变化，例如OSD加入和删除、OSD宕掉和恢复、存储池创建和删除等，最终都会转化为状态机中的事件，驱动状态机在不同状态之间进行跳转。

该状态机主要状态如图6-10所示。

由图6-10可见，该状态机当前一共有4级状态。每一种状态又可以包含若干子状态，所有子状态中第一个状态为其默认状态。例如，该状态机初始化时，默认会进入Initial子状态；又比如，当该状态机从其他状态，如Reset状态，跳转至Started状态时，默认会进入Started/Start子状态等。

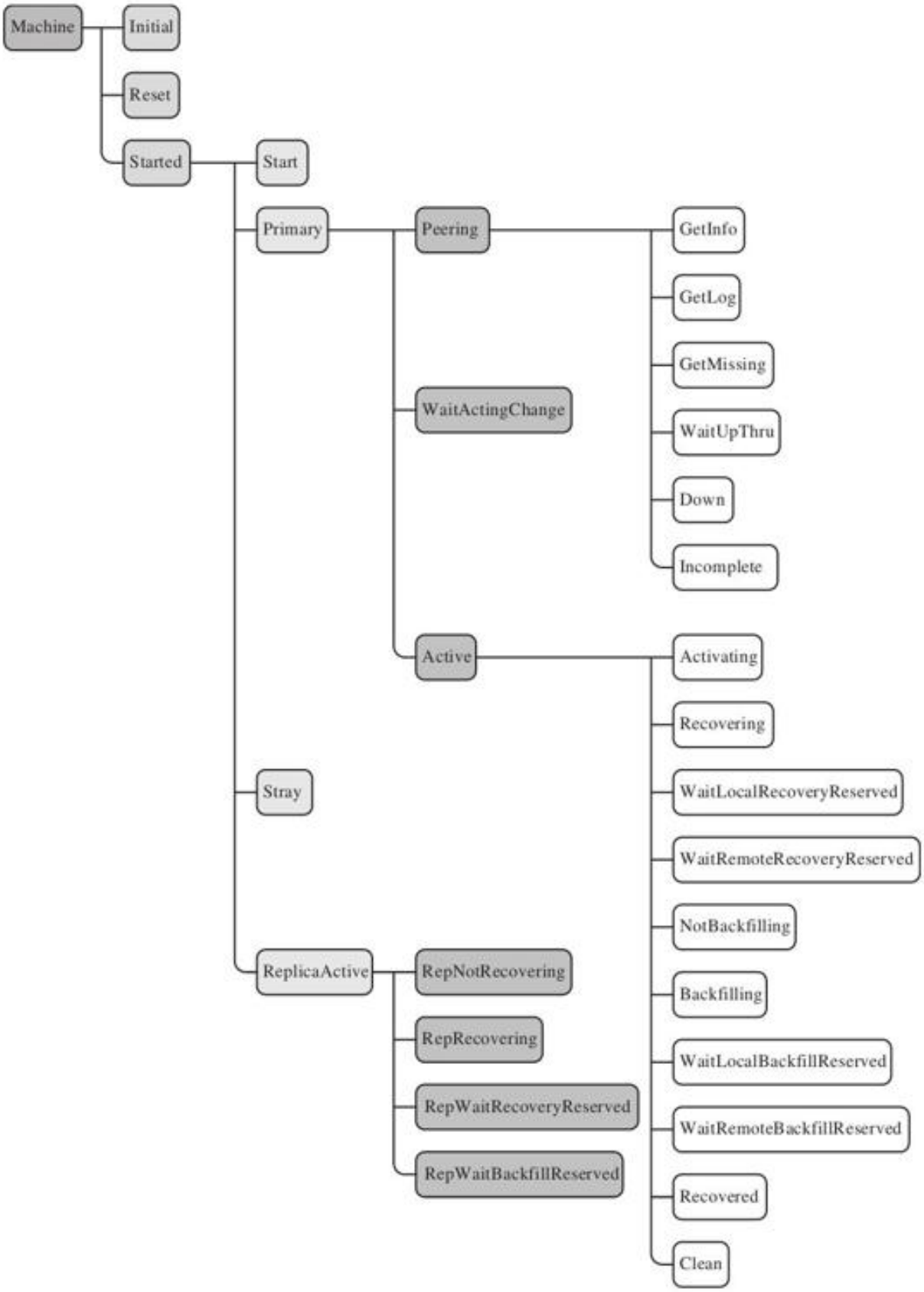

Machine
Initial
Reset
Started
Start
Primary
Peering
GetInfo
GetLog
GetMissing
WaitActingChange
WaitUpThru
Down
Incomplete
Active
Activating
Recovering
WaitLocalRecoveryReserved
WaitRemoteRecoveryReserved
NotBackfilling
Backfilling
WaitLocalBackfillReserved
WaitRemoteBackfillReserved
Recovered
Clean
Stray
ReplicaActive
RepNotRecovering
RepRecovering
RepWaitRecoveryReserved
RepWaitBackfillReserved

图6-10　PG状态机主要状态

与状态机相关的主要事件如表6-15所示。

表6-15　PG状态机主要事件

| 事件 | 含　义 |
| --- | --- |
| Activate | Peering 即将完成，通知相关副本保存 Peering 结果，准备激活 PG（使其变为 Active 状态） |
| ActMap | 激活 OSDMap |
| AdvMap | 同步 OSDMap |
| AllBackfillsReserved | 所有参与 Backfill 的副本完成资源预留，准备开始执行 Backfill |
| AllRemotesReserved | 所有参与 Recovery 的副本完成资源预留，准备开始执行 Recovery |
| AllReplicasActivated | 所有副本 Activate 操作完成 |
| AllReplicasRecovered | Recovery 完成，并且后续无须执行 Backfill |
| Backfilled | Backfill 完成 |
| BackfillTooFull | Backfill 副本所在的 OSD 空间不足（Backfillfull） |
| DoRecovery | 准备开始执行 Recovery，通知 Primary 发起资源预留 |
| GoClean | 通知 PG 跳转至 Clean 状态 |
| GotInfo | Peering 过程中，Primary 成功收到所有请求的 Info |
| Initialize | 创建 PG |
| Load | 加载 PG |
| LocalBackfillReserved | Backfill 过程中，Primary 本地预留资源成功 |
| LocalRecoveryReserved | Recovery 过程中，Primary 本地预留资源成功 |
| MInfoRec | PG 收到 Info |
| MLogRec | PG 收到 Log |
| MNotifyRec | PG 收到 Notify |
| NeedUpThru | 通知 OSD 进行 up_thru 更新 |
| NullEvt | 空事件 |
| RemoteBackfillReserved | 收到副本 Backfill 资源预留成功响应 |
| RemoteRecoveryReserved | 收到副本 Recovery 资源预留成功响应 |
| RemoteReservationRejected | Backfill 过程中某个副本资源预留失败，例如因为所在的 OSD 可用空间不足（Backfillfull） |
| RequestBackfill | 准备执行 Backfill |
| RequestRecovery | 准备执行 Recovery |

状态机的整体工作流程如图6-11所示。

接下来，我们结合一些常见场景来说明上述状态机是如何工作的。为叙述方便，我们约定形如“Started/Start”的复合状态指示状态机状态，形如“Active+Clean”的复合状态指示PG外部状态。大部分单独出现的状态，一般都用于表示PG的外部状态，具体含义可以参考表6-14。

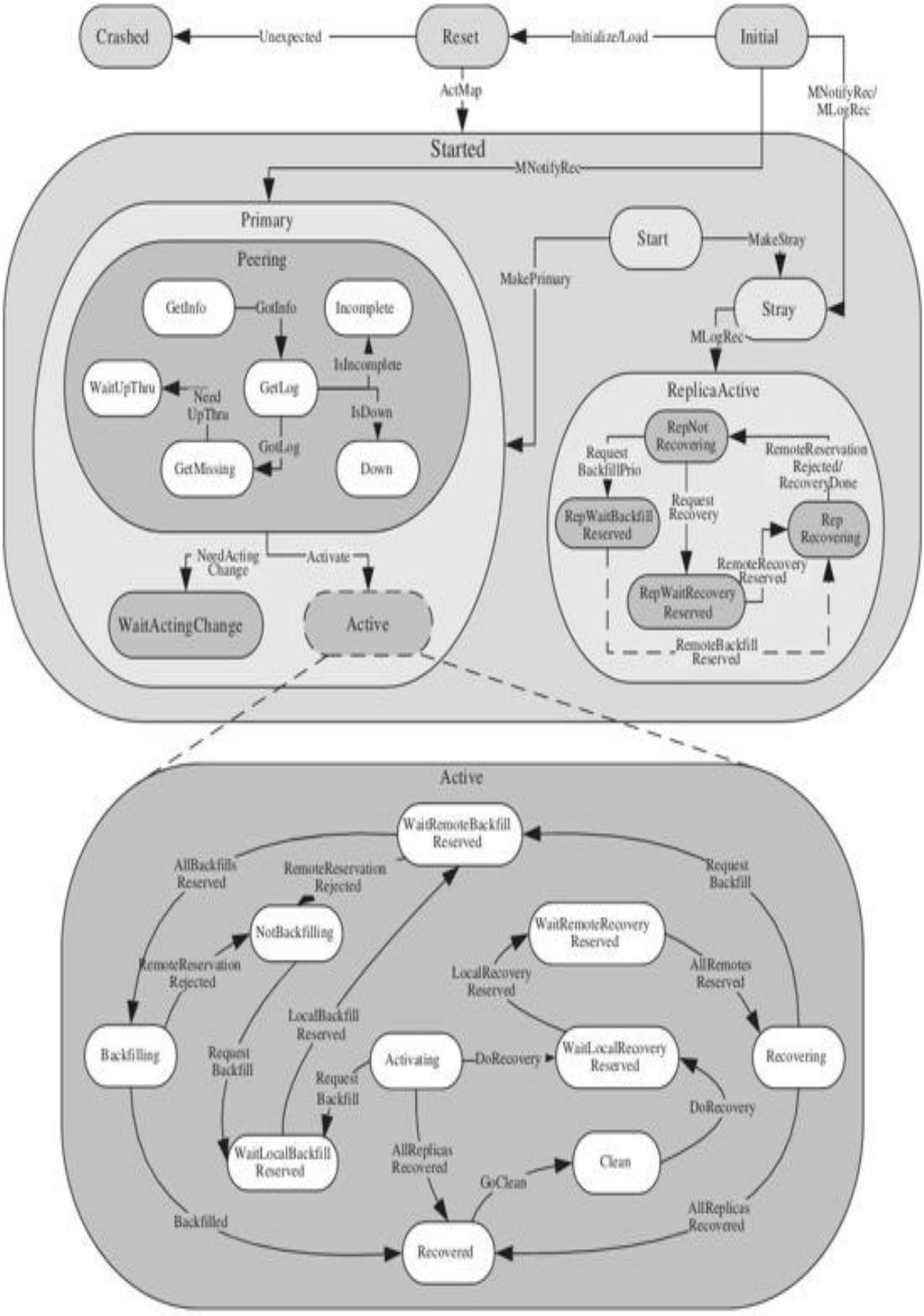

Crashed
Unexpected
Reset
Initialize/Load
Initial
ActMap
MNotifyRec/
MLogRec
Started
MNotifyRec
Primary
Peering
GetInfo
GotInfo
Incomplete
IsIncomplete
WaitUpThru
Need
UpThru
GetLog
IsDown
GetMissing
GotLog
Down
NeedActing
Change
Activate
WaitActingChange
Active
Start
MakeStray
MakePrimary
Stray
MLogRec
ReplicaActive
RepNot
Recovering
Request
BackfillPrio
RemoteReservation
Rejected/
RecoveryDone
RepWaitBackfill
Reserved
Request
Recovery
Rep
Recovering
RemoteRecovery
Reserved
RepWaitRecovery
Reserved
RemoteBackfill
Reserved
Active
WaitRemoteBackfill
Reserved
AllBackfills
Reserved
RemoteReservation
Rejected
Request
Backfill
NotBackfilling
WaitRemoteRecovery
Reserved
RemoteReservation
Rejected
LocalRecovery
Reserved
AllRemotes
Reserved
LocalBackfill
Reserved
Backfilling
Request
Backfill
Activating
DoRecovery
WaitLocalRecovery
Reserved
Recovering
Request
Backfill
DoRecovery
WaitLocalBackfill
Reserved
AllReplicas
Recovered
Clean
GoClean
Backfilled
Recovered
AllReplicas
Recovered

图6-11　PG状态机工作流程

### 6.3.2 创建PG

Monitor收到存储池创建命令之后，将异步地向池中每个OSD派发批量创建PG请求。

与读写流程类似，创建PG也由Primary主导进行，即Monitor仅需要向当前Primary所在的OSD下发PG创建请求即可，其他副本会在随后由Primary发起的Peering过程中自动被创建。

如图6-12所示，OSD在创建PG之前需要先确认PG关联的存储池是否仍然存在、Primary是否发生了变化、Up是否与Acting相同等。事实上，Monitor在向OSD发送PG创建请求时已经对上述信息进行过确认，之所以还要（再次）执行这类检查，主要是由于OSD收到请求时，整个集群拓扑结构可能又发生了变化（例如用户创建存储池之后马上删除），导致OSD当前持有的OSDMap可能比请求产生时的OSDMap更新。

一些极端情况下，例如由于网络振荡导致集群短时间内产生了大量OSDMap变化，为了防止PG在多个OSD上被重复创建（例如PG的Primary由A->B->C->A，即OSD.A收到此PG创建请求时，PG的Primary期间经过多次切换，最终又切回OSD.A），我们还需要对整个PG的（Interval）变化历史进行回溯。

回溯基于OSDMap变化列表进行。以PG创建时的OSDMap作为起点，以OSD当前持有的最新OSDMap作为终点，回溯最终将产生3个属性，分别称为：

·same_interval_since，标识PG最近一个Interval的起点。

·same_up_since，标识自该Epoch开始，PG当前的Up没有发生过变化。

·same_primary_since，标识自该Epoch开始，PG当前的Primary没有发生过变化。

因此，如果OSD检测到PG最新的same_primary_since大于请求中携带的Epoch，说明在此期间该PG的Primary曾经发生过切换，OSD需要放弃执行本次创建操作，将决策权交还给Monitor。

如果满足条件，OSD将向状态机投递一个NullEvt事件（顾名思义，该事件没有任何实际作用，仅仅用于初始化创建流程），正式开始创建PG。新创建的PG（为Primary）随后将加入OSD内部的peering_wq队列，通过Peering完成其他副本的创建。

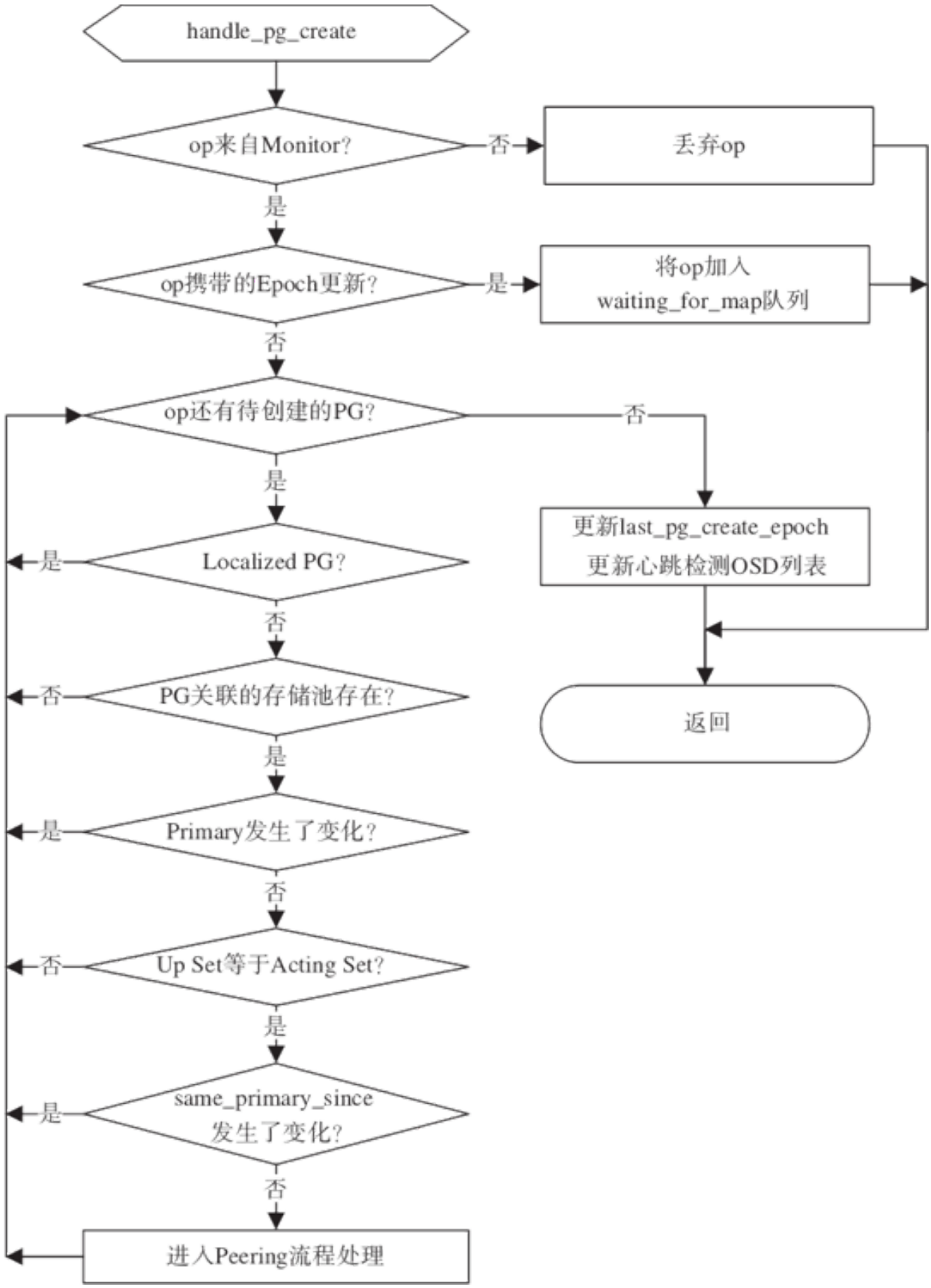

handle_pg_create
op来自Monitor?
否
丢弃op
是
op携带的Epoch更新?
是
将op加入
waiting_for_map队列
否
op还有待创建的PG?
否
是
更新last_pg_create_epoch
更新心跳检测OSD列表
是
Localized PG?
否
否
PG关联的存储池存在?
是
是
Primary发生了变化?
否
否
Up Set等于Acting Set?
是
是
same_primary_since
发生了变化?
否
返回
进入Peering流程处理

图6-12　OSD创建PG

### 6.3.3 Peering

与处理客户端发起的读写请求类似，在OSD内部，所有需要执行Peering的PG也会进入一个专门的peering_wq工作队列，该工作队列同样需要绑定一个线程池（peering_tp）为其提供具体的服务线程。

由于任意一个OSD故障都会导致大量PG需要同时执行Peering，为了缩短Peering时间（从而缩短故障切换时间），一方面Peering流程设计得非常轻量级，例如完成Peering仅需要在副本之间交换少量元数据（Info、Log等），另一方面peering_wq被设计成批处理队列，线程池中的线程对peering_wq中的PG进行处理时，可以一次出列和处理多个PG。

无论是存储池创建，还是OSD数量或者状态发生变化，最终只有反馈至OSDMap中，才能被PG感知。因此任何时候，当OSD收到新的OSDMap并完成同步之后，需要通知其承载的所有PG都进入peering_wq，各自完成OSDMap同步。

特别地，如果有大量OSDMap需要同步（例如OSD长时间下电后重新上电），为了防止某个PG过多地占用时间片导致其他PG的Peering得不到调度（PG之间的Peering进度可能不一致），每处理一定数量的OSDMap之后，对应PG会让出时间片，重新进入peering_wq等待下次继续同步。这样处理的原因有两个：一是由于Ceph随机分布数据，例如RBD中的一个image，一般情况下其数据最终会分布在其归属存储池的

所有PG之上，因此我们倾向于让所有PG都快速完成Peering，否则任意一个PG长时间卡在Peering阶段，有可能导致所有客户端（例如虚拟机）都无法正常工作；二是我们希望所有PG同步OSDMap的步调尽可能一致，以免OSD需要保存大量OSDMap，浪费存储空间。

每完成一张OSDMap同步，PG都会同步向状态机投递一个AdvMap事件，该事件携带新老两张OSDMap，供状态机判断是否需要重启Peering。如果需要重启，那么状态机将进入Reset状态，完成PG重启Peering之前的准备工作，例如更新Up、Acting，暂停Recovery、Scrub任务，丢弃尚未处理完成的op等。如果全部OSDMap同步完成（或者阶段性完成，Peering可以随时重启），则向状态机投递一个ActMap事件，正式启动Peering。

判断是否需要重启Peering的法则比较简单，只要检查本次OSDMap更新之后是否会触发PG进入一个新的Interval即可。由于PG支持对OSDMap批量进行同步，在正式启动Peering之前，PG在这些历史OSDMap变化序列中可能产生了一系列已经发生过的Interval，称为past_intervals。

1.past_intervals

单个Interval用于指示一个相对稳定的PG映射周期，在此期间，PG的Up与Acting没有发生过变化，关联存储池的副本数、最小副本数以及PG数量等也没有发生过变化，其磁盘数据结构如表6-16所示。

表6-16　pg_interval_t

| 成　员 | 含　义 |
| --- | --- |
| acting | Acting |
| first<br>last | Interval 的起始与终止 Epoch |
| maybe_went_rw | 用于标识本 Interval 内，PG 是否可能接受了客户端发起的读写请求（即 PG 曾经处于 Active 状态）<br>如果 maybe_went_rw 为 false，说明本 Interval 内不可能产生过由客户端发起的数据更新，因此后续执行 Peering 的过程中，可以安全地跳过（bypass） |
| primary | Acting 中的第一个 OSD，即 Acting Primary |
| up | Up |
| up_primary | Up 中的第一个 OSD，即 Up Primary |

由定义可见，Interval保存了针对自身后续执行Peering所需的全部（OSD）信息。因此，每次生成past_intervals之后，PG可以立即将past_intervals作为元数据之一固化至本地磁盘，防止PG持续引用一些较老的OSDMap，导致OSD需要存储大量OSDMap，浪费存储空间。

生成Interval的规则比较简单，每次从OSD获取当前待更新的OSDMap时，我们取其中PG关联存储池的副本数、最小副本数和PG数目，并基于CRUSH重新计算PG的Acting、ActingPrimary、Up、Up Primary，然后与前一个相邻的OSDMap进行比较。如果上述参数中任意一个发生变化，则说明老的Interval结束，新的Interval开始，此时可以完善老的Interval中诸如first、last、acting、primary、up、up_primary等信息。这里的难点在于如何设置maybe_went_rw。

maybe_went_rw用于标识对应Interval中，PG有无可能变为Active状态，从而接受客户端发起的读写请求。显然，如果该标志为true，为了

避免潜在的数据丢失风险，我们后续需要进一步探测此Interval中相关OSD进行确认，反之则可以直接跳过该Interval。

一种保守的策略是始终设置maybe_went_rw为true，但是相应地会带来额外的Peering时间开销。由于大部分情况下Peering是PG唯一不响应客户端请求的阶段，原则上我们应该尽最大可能缩短Peering时间。此外，触发OSDMap变化的原因种类繁多，由此生成的past_intervals中，也并非每个Interval都需要通过Peering去确认是否需要执行数据恢复。极端的例子如，某个Interval中，Acting小于存储池最小副本数，那么显然应该直接忽略该Interval。如果此时错误地设置maybe_went_rw为true，反而可能由于该Acting中某些OSD发生永久性故障导致Peering卡住。

考虑到每个Interval能够接受客户端读写请求的条件实际上十分苛刻（因此需要考虑的场景也就十分有限），合理的做法是尽最大可能找出那些可能发生过读写的场景。例如，某个Interval内，Acting Primary存活并且Acting不小于存储池最小副本数，那么几乎就可以断定该Interval内可能接受过客户端的读写请求。然而此时贸然设置maybe_went_rw为true仍然为时过早，我们通过举例进行说明。

为简单起见，假定某个集群只有编号为A、B的两个OSD，考虑如下场景：

1）Epoch 1，A和B正常在线；

2）Epoch 2，A和B同时掉电，但是由于Monitor检测OSD宕掉有滞后，所以仅A被标记为Down；

3）Epoch 3，Monitor检测到B宕掉，将B标记为Down；

4）Epoch 4，A重新上电并被Monitor标记为Up。

上述集群OSD状态变化过程可用OSDMap简记如下：

```
Epoch 1: A，B
Epoch 2: B
Epoch 3:
Epoch 4: A
```

当集群中某个PG在Epoch 4执行OSDMap同步时，上述OSDMap变化序列将被该PG分解为4个Interval（其中每个Interval仅包含一个Epoch，记为Interval(i)，i=1，2，3，4），并加入其past_intervals列表。

这里的问题在于，由于所有Interval都是PG事后根据CRUSH计算得到的，如果在Epoch 2，Monitor错误地将B标记为Up，那么经过计算我们仍然会（错误地）得到Interval(2)的Acting为B这个结论。由于此时Acting Primary存活（为B）并且Acting大小等于最小副本数（为1），按照前面的结论，我们认为期间可能由B正常完成了Peering，并接受了来自客户端的读写操作，从而设置Interval(2)的maybe_went_rw为true。因此，在随后进行的Peering过程中，Interval(2)无法被安全跳过，即我们必须等到B重新上线以确认到底有无客户端发起过数据更新，但这显然不是必须的。

为了解决上面这个问题（例如由于B永久性故障导致Peering卡住），我们规定PG在切换至新的Interval之后、成功完成Peering并重新开始接受客户端读写请求之前，必须先通知Monitor设置其归属OSD的up_thru。

针对上面这个例子，新的OSDMap变化序列为：

```
Epoch 1: A, B
Epoch 2: B up_thru[B] = 0
```

```
Epoch 3:
Epoch 4: A
```

---

由于Interval(2)期间B的up_thru一直为0，说明B未能成功完成Peering，因此可以安全地将Interval(2)的maybe_went_rw设置为false。

进一步地，考虑上面这个例子的另一种可能情形：

1）Epoch 1，A和B正常在线；

2）Epoch 2，A掉电并被Monitor标记为Down，PG正常完成了Peering；

3）Epoch 3，Monitor成功设置B的up_thru为2；

4）Epoch 4，Monitor检测到B宕掉，将B标记为Down；

5）Epoch 5，A重新上电并被Monitor标记为Up。

对应的OSDMap变化序列变为：

---

```
Epoch 1: A, B
Epoch 2: B up_thru[B] = 0
Epoch 3: B up_thru[B] = 2
Epoch 4:
Epoch 5: A
```

---

当PG在Epoch 5执行OSDMap同步时，上述OSDMap变化序列仍将被该PG分解为4个Interval，分别为Interval(1)、Interval(2～3)、Interval(4)、Interval(5)。

由于Interval(2～3)期间B成功地将自身的up_thru更新为2（不小于Interval(2～3)的起始Epoch），说明可能由B完成Peering并接受了客户

端的读写请求，此时必须将Interval(2～3)的maybe_went_rw设置为true。

这样，通过引入（与OSD状态相关的）up_thru，设置某个Interval的maybe_went_rw为true的条件变为：

·Acting Primary存在。

·Acting中的副本个数不小于存储池最小副本数。

·该Interval内，Acting Primary所在的OSD成功地更新了up_thru。

2.GetInfo

如果状态机在Reset状态下收到ActMap事件，则意味着可以正式启动Peering。通过向状态机发送MakePrimary事件，Primary将进入Started/Primary/Peering/GetInfo状态，开始收集所有副本的Info。其他副本则进入Started/Stray状态，等待Primary进一步确认身份。

针对Primary，由于此时执行Peering的主要依据past_intervals已经生成，可以据此收集本次需要参与Peering的全部副本信息。这些副本组成一个集合，称为PriorSet。

简言之，构建PriorSet的过程就是针对past_intervals中的Interval进行逆序遍历，找出所有我们感兴趣的Interval。那么哪些Interval是我们感兴趣的呢？

首先，Interval不能发生在当前Primary的last_epoch_started之前。

当Peering接近尾声时（PG变为Active之前），为了避免由于参与本次Peering的OSD故障等原因导致Peering前功尽弃，我们将在最后一步由Primary通知所有副本更新Info中的last_epoch_started，将其指向本

次Peering成功完成时的Epoch，并与日志以及Info中的其他信息一并固化至本地磁盘。

因此，如果Interval发生在Primary的last_epoch_started之前，说明其在上一次Peering中已经被成功处理过，无须再次进行处理。这也解释了为什么我们要针对past_intervals进行逆序遍历，因为一旦某个Interval发生在Primary的last_epoch_started之前，那么past_intervals中更早的Interval必然都可以忽略，此时可以直接结束遍历。

需要注意的是，Info中实际保存了两个last_epoch_started属性，一个位于Info属性之下，另一个则位于Info的history属性之下。前者由每个副本收到Primary发送的Peering完成通知之后更新，并与Peering结果一并存盘，后者则由Primary在收到所有副本Peering完成通知应答之后更新和存盘。由于Primary更新完history中的last_epoch_started属性之后会同步设置PG为Active状态，因此history中last_epoch_started也用于指示PG最近一次开始接受客户端读写请求时的Epoch。

其次，Interval的maybe_went_rw属性不能为false。

参考前面的分析，如果Interval的maybe_went_rw为false，那么可以直接跳过该Interval。

构建PriorSet的过程中，如果我们捕获到某个必须处理的Interval不足以完成Peering，例如Acting中所有副本目前都处于离线状态，或者Acting中目前剩余的在线副本不足以完成数据恢复，那么此时可以直接将PG设置为Down，终止Peering。反之，如果PriorSet构建成功，那么可以继续执行Peering。

由读写流程我们知道，日志保存了所有写请求的关键信息，因此只要能够从所有候选副本中选出一个拥有最完整日志（即权威日志）的副本，就可以以其作为基准，完成副本间的数据同步。

为了避免大量日志传输延长Peering时间，包括Primary在内的每个副本每次进行日志更新时，都会同步更新自身Info中与日志相关的last_update、log_tail等信息，并将Info与写请求一并存盘。因此，为了找出权威日志（及其所在的副本），Primary将首先向所有PriorSet中的副本发送Query消息，用于收集Info。

3.GetLog

当所有Info收集完毕之后，Primary通过向状态机发送一个GotInfo事件，跳转至Started/Primary/Peering/GetLog状态，准备执行日志同步。

在此之前，Primary需要先基于如下原则选取一份权威日志，作为同步的基准（以多副本为例）。

1）优先选取具有最新内容的日志（即Info中的last_update最大）。

2）如果存在多份满足条件1）的日志，优先选取保留了更多日志条目的日志（即Info中的log_tail最小）。

3）如果存在多份满足条件2）的日志，优先选取当前的Primary。

上述策略之所以可行，例如不必担心日志产生不连续性的原因在于，为了保证每个Interval切换之后能够正常发生客户端读写，PG必须在新的Interval内成功完成Peering，而Peering成功完成必然意味着PG至少已经将全部Acting中副本的日志记录同步到了最新。因此，实际操作时，我们可以进一步缩小权威日志的候选者范围，仅从最近一次成功完成过Peering的那些副本中选取。

如果选取权威日志失败，那么PG将向状态机发送一个IsIncomplete事件，跳转至Started/Primary/Peering/Incomplete状态，同时将自身状态设置为Incomplete。反之，PG可以开始基于权威日志选取后续需要（或

者能够）执行在线数据恢复的副本，这个过程称为choose_acting。我们仍以多副本为例进行说明。

理论上，由于我们强烈依赖CRUSH的伪随机性（这同样意味着公平性）在OSD间平均分配PG及其副本，进而在集群内实现负载均衡，一个显而易见的指导原则是尽量选取按照当前OSDMap计算得到的Up中的副本。然而，也正由于CRUSH选择Up的随机性，某些情况下，Up中的某些OSD或者因为没有PG的任何历史副本，或者因为所承载的副本内容过于落后，导致它们无法通过日志以增量的方式（即Recovery）同步，而只能通过拷贝其他正常PG全部内容的方式（即Backfill）来同步。特别地，如果上面这种情况发生在Primary（即Up Primary）上，将导致PG长时间无法接受客户端发起的读写请求，这显然不可接受。

一个变通的办法是尽可能选择一些还保留着相对完整内容的副本进行过渡，对应的OSD称为PG Temp（即这些OSD是副本的“临时”载体）。进一步地，我们通过在OSDMap中设置PG Temp，并显式替换CRUSH的计算结果（为区别起见，使用PG Temp替换后的Up称为Acting。当客户端需要向集群发送读写请求时，总是选择当前Acting中的第一个OSD（即Acting Primary所在的OSD）进行发送），即可在完成Peering之后，将客户端的读写请求重定向至Acting中的副本之上，从而避免产生长时间的业务中断。当Up中的副本在后台通过Backfill完成数据同步之后，可以通知Monitor取消PG Temp，此时Acting与Up将再次变得一致，客户端后续的读写业务也可以随之切回至Up中的副本。

上述Acting与Up不一致的现象，我们称为Remapped，它同时也是一种PG外部状态，表明当前Up中的某些副本需要或者正在通过Backfill进行修复。所有Up与Acting中的OSD一起组成一个新的集合，称为ActingBackfill。

引入PG Temp之后，考虑到我们的最终目的仍然是要将Acting切回Up，为了避免不必要的数据同步，我们需要针对PG Temp生效期间由

客户端产生的写请求进行特殊处理。

1）写入一个全新对象。容易理解，这是最简单的情况，可以同时写入ActingBackfill中的每个副本。

2）修改一个已有对象。如果该对象正在被ActingBackfill集合当中任意一个副本执行Backfill，则阻塞此请求。等待Backfill完成后，按如下方式处理：所有Acting中的副本和所有已经完成该对象Backfill的副本，正常执行事务；所有尚未完成该对象Backfill的副本，则直接执行一个空事务（因为这些副本上还不存在该对象），仅用于更新日志和统计信息。

至此，我们已经能够理解choose_acting名称的由来，它的目的是为PG选出一组副本来充当Acting。习惯上将choose_acting选出的结果称为want_acting，其具体选择过程如下：

1）选取Primary。如果当前Up中的Primary能够基于权威日志修复或者自身就是权威日志，则直接将其选为Primary；否则选择权威日志所在的副本作为Primary。将选中的Primary加入want_acting列表。

2）依次考虑Up中所有不在want_acting列表中的副本，如果其还能够基于权威日志修复，则加入want_acting列表，等待后续通过Recovery修复；反之，将其加入Backfill列表，等待后续通过Backfill修复。

3）如果当前want_acting列表大小等于存储池副本数，则终止；否则继续从当前Acting中依次选择能够基于日志修复的副本加入want_acting列表。

4）如果当前want_acting列表大小等于存储池副本数，则终止；否则继续从所有返回过Info的副本中选取能够基于日志修复的副本加入want_acting列表，直至当前want_acting大小等于存储池副本数。

如果choose_acting无法选出足够的副本完成数据恢复（例如针对纠删码而言，要求存活的副本数不小于k值才能进行数据恢复），那么PG将进入Incomplete状态；如果choose_acting选出来的want_acting与当前Acting不一致，说明需要借助PG Temp临时进行过渡，Primary将先向Monitor发送设置PG Temp请求，随后向状态机投递一个NeedActingChange事件，并进入Started/Primary/WaitActingChange状态，等待PG Temp在新的OSDMap生效后继续执行Peering。

自Jewel版本开始，为了进一步缩短Peering时间，Ceph引入了一种对PG Temp进行预填充的机制，称为Prime PG Temp，其主要设想在于：每次Monitor在新的OSDMap生效之后，同步计算基于当前OSDMap产生的所有PG映射结果，然后抢在下一个OSDMap生效之前，判定在此期间OSDMap发生的变化是否有可能会导致某些PG变为Remapped状态。如果可能，例如生成下一个OSDMap是由某些OSD宕掉触发，则基于新的OSDMap（尚未生效）重新计算PG映射结果，并与之前的计算结果相比较，找出那些即将受到影响的PG，预先填充它们的PG Temp，然后将这些PG Temp写入新的OSDMap使之一并生效，从而避免这些PG在后续Peering过程中再次向Monitor请求更新PG Temp。需要注意的是，这种预填充带有猜测性质，如果PG后续通过choose_acting选出来的PG Temp与之不符，仍然可以发送（新的）PG Temp要求Monitor修改。

当PG Temp生效之后或者Acting本身就与Up一致，Primary接下来将进行日志同步。如果Primary自身没有权威日志，那么需要通过发送Query消息去权威日志所在的副本去拉取。为了尽可能减少日志传输，Primary需要预先计算待拉取的日志条目范围，容易理解其终点对应权威日志的最新日志条目，因此仅需要确定起点，由于后续所有Acting中的副本都需要通过日志修复，为了防止修复某些副本时需要引用较老的日志条目而Primary当前又没有（比如这些副本最新日志版本号比Primary最老的日志版本号还小），而不得不从权威日志所在的副本重

新去获取，合理的做法是选择所有Acting副本中最小的last_update作为起点。

成功获取权威日志之后，Primary可以将其与本地日志进行合并，以生成本地权威日志。日志合并过程中主要考虑两点：一是权威日志中存在比Primary更老的日志条目。由于日志能够删除的前提是对应的写操作必须已经在（参与了该写操作的）副本之间完成了同步，所以这部分日志与Primary自身不存在一致性问题，无须特殊处理，直接追加在Primary本地日志的尾部即可；二是权威日志中存在比Primary更新的日志条目。原则上，仅需要将新的日志条目按照顺序依次追加到Primary本地日志的头部即可，但是两者存在重合的那部分日志条目则需要进行特殊处理。

假定Primary的本地日志和权威日志分别如表6-17和表6-18所示，对比两者最后一条日志，显然它们的Version相同但是Epoch不同（这种现象通常是由于PG的Interval发生了切换导致的），这意味着两条日志产生了分歧。那么此时应该如何处理这种分歧呢？

表6-17　Primary的本地日志

| Eversion | 操作类型 | 对象名 |
| --- | --- | --- |
| 10'4 | MODIFY | foo |
| 10'5 | MODIFY | foo |
| 10'6 | MODIFY | foo |

表6-18　权威日志

| Eversion | 操作类型 | 对象名 |
| --- | --- | --- |
| 10'4 | MODIFY | foo |
| 10'5 | MODIFY | foo |
| 11'6 | MODIFY | foo |

一种自然的想法是先将Primary本地与权威日志产生分歧的日志进行回退，即先解决分歧，然后再执行合并。考虑到这些分歧日志可能（多次）操作同一个对象，也可以先将Primary本地有分歧的日志取出，待与权威日志完成合并之后，再集中解决分歧，具体方法我们将在后面介绍。

与权威日志合并的过程中，如果Primary发现本地有对象需要修复，则将其加入自身的missing列表。完成合并之后（或者Primary自身就拥有权威日志），Primary通过向状态机发送一个GotLog事件，切换至Started/Primary/Peering/GetMissing状态，开始向所有仍然能够通过Recovery恢复的副本（即Acting中的副本）发送Query消息，以获取它们的日志。收到每个副本的日志之后，通过与本地日志对比（此时Primary已经完成了权威日志同步），Primary可以构建这些副本的missing列表，作为后续执行Recovery的依据。

4.GetMissing

每个副本的missing列表记录了所有需要通过Recovery进行修复的对象信息。以对象为单位，为了后续能够被正确修复，missing列表中的每个条目仅需要记录如下两个重要参数：

·need。need指对象需要被同步至此版本号，即对象的权威版本号。

·have。have指对象在归属副本（或者Primary）的本地版本号。

当Primary收到每个副本的本地日志后，可以通过日志合并的方式得到每个副本的missing列表。这一过程和前面我们提到的Primary本地日志与权威日志合并过程类似，也可能需要解决分歧日志。

为了解决分歧日志，我们首先将所有分歧日志按对象进行分类，即所有针对同一个对象操作的分歧日志都使用一个队列进行管理，然后逐个队列（对象）进行处理。我们假定针对同一个对象操作的一系列分歧日志中，最老的那条分歧日志生效之前对象的版本号为prior_version，则针对每个队列（对象）的处理都可以归结为以下5种情形之一：

1）本地存在比分歧日志条目更新的日志。如表6-17和表6-18所示的例子，此时仅包含一条分歧日志，其版本号为10’6，更新的日志条目（来自权威日志）版本号为11’6，两者操作同一个对象foo。为了防止两者执行完全不同的操作（例如一个改写foo的数据，另一个改写foo的omap），此时直接将本地对象删除，将其加入missing列表，同时设置其need为11’6，have为0（即本地没有）。

2）对象之前不存在（例如prior_version为0，或者最老的分歧日志类型为CLONE）。

此时直接删除对象。

3）对象当前位于missing列表之中（例如上一次Peering完成之后，Primary刚刚更新了自身的missing列表，但是其中的对象还没来得及修复，Acting中的某些OSD再次发生掉电）。如果此时have等于prior_version，说明所有分歧日志针对该对象的操作尚未生效（因此也就无须执行回滚），此时可以直接将对象从missing列表中移除；反之，则说明至少有部分分歧日志操作已经生效，此时需要将对象回滚至最老的分歧日志操作之前的版本，即prior_version。这通过修改对象在missing列表中的need为prior_version实现。

4）对象不在missing列表之中同时所有分歧日志都可以回滚。此时将所有分歧日志按照从新到老的顺序依次进行回滚。

5）对象不在missing列表之中并且至少存在一条分歧日志不可回滚。此时将本地对象直接删除，将其加入missing列表，同时设置其need为prior_version，have为0。

生成完整的missing列表之后，Primary将向状态机投递一个Activate事件，进入Started/Primary/Active/Activating状态，准备激活PG。

5.Activate

随着Peering接近尾声，在PG变为Active状态接受客户端读写请求之前，必须先固化本次Peering成果，防止因为系统掉电等意外因素导致前功尽弃。同时还必须收集后续用于执行在线数据恢复所依赖的关键信息。上述过程称为激活（Activate）PG。

（1）last_epoch_started

我们已经知道，last_epoch_started用于指示上一次Peering成功完成时的Epoch，但是由于Peering涉及在多个副本之间进行元数据和状态同步，所以同样存在进度不一致的可能。为此，我们设计了两个last_epoch_started，一个用于标识ActingBackfill中的副本，其本地Activate操作已经执行完成，直接作为其Info的子属性保存；另一个则由Primary检测到所有Activate操作都完成后，保存在自身Info的history属性下，并最终同步至所有ActingBackfill中的副本。

由于ActingBackfill成员已经全部确定，此时可以按照每个成员的实际情况，分别发送Info或者Log消息，将它们转化为Started/ReplicaActive状态，真正参与到后续客户端读写、Recovery或者Backfill流程中来。具体分为以下几种情形：

1）副本本身的日志已经与权威日志同步，则直接向其发送Info消息，通知其更新last_epoch_started即可。

2）副本需要从头开始Backfill，向其发送Log消息，重置其Info中的last_backfill为空对象。

3）副本可以通过Recovery修复，向其发送Log消息，并携带副本缺少的那部分日志条目。

收到Log消息的副本，如需重启Backfill，则初始化Backfill设置，否则需要通过日志合并在本地重新生成完整的日志和missing列表。

通过向状态机投递一个Activate事件，副本可以切换至Started/ReplicaActive状态，开始执行Activate操作，固化更新后的Info、日志（包含missing列表）等关键元数据至本地磁盘。

### （2）MissingLoc

至此，Primary已经得到了自身以及副本的missing列表，但是该列表仅仅记录了每个降级对象的权威版本号，Primary还必须清楚这些权威版本所在的确切位置才能执行修复。为此，PG引入了一种同时包含所有missing条目与每个降级对象权威版本位置信息的全局数据结构，称为MissingLoc。

MissingLoc内部可以细分为两张表：needs_recovery_map和missing_loc。顾名思义，它们分别保存了所有降级对象以及这些对象权威版本的位置信息。需要注意的是，降级对象的权威版本可能同时存在于多个副本上，因此missing_loc中记录的位置信息实际上是一些副本的集合。

由于Recovery必须由Primary主导，所以MissingLoc也必须由Primary来统一生成。对应生成needs_recovery_map和missing_loc的顺

序，生成完整的MissingLoc也分为两步：首先将所有副本的missing列表中的条目依次加入needs_recovery_map中；其次，以每个副本的Info和missing列表作为输入，针对needs_recovery_map逐个对象进行检查，以确定其权威版本的位置并填充至missing_loc。具体原则如下：

1）如果降级对象的权威版本号（即need）比副本的最新日志版本号（即last_update）还要大，说明副本不可能存在降级对象的权威版本，直接跳过。

2）如果副本需要执行或者继续执行Backfill，并且降级对象在上一个已经完成过Backfill的对象之后（通过Info中的last_backfill指示），说明该对象尚未被Backfill过，直接跳过。

3）降级对象也位于副本的missing列表之中，直接跳过。

4）否则认为副本存在降级对象的权威版本。

生成MissingLoc之后，如果needs_recovery_map不为空，即存在任何需要被Recovery的对象，则Primary会将自身设置为Degraded+Activating状态；进一步地，如果Primary检测到当前Acting小于存储池副本数，则同时设置Undersized状态。之后，Primary将通过ObjectStore固化本次Peering的结果，并等待其他副本完成Activate操作。

所有Activate操作完成之后，Primary将向状态机投递一个AllReplicasActivated事件来清除自身的Activating状态，同时检测此时Acting是否小于存储池最小副本数，如果小于，则将自身设置为Peered状态并终止后续处理；否则将自身设置为Active状态，同时将之前阻塞的、来自客户端的请求重新入列。

随着PG进入Active状态，Peering正式宣告完成，此后PG可以正常处理来自客户端的请求，Recovery或者Backfill可以切换至后台进行。

其实现细节我们将在下一章进行介绍。

## 6.4 总结和展望

在Ceph中，PG的定位或者说主要职责如下：

·作为存储池的基本组成单位，负责执行存储池绑定的副本策略。

·以OSD为单位，进行副本分布，将客户端针对原始对象的操作，转化为ObjectStore所能理解的本地事务操作，并保证副本之间的强一致性。

对商用存储系统而言，数据强一致性是必备特性。Ceph的分布式特性使得这种数据强一致性要进一步地在不同节点之间仍然保持，而PG能够“随心所欲”地在节点之间迁移使得要做到这一点难上加难，为此不得不在复杂度与效率之间进行折中。例如：

作为副本间数据一致性的主要依据，PG的日志系统设计得比较简单，并没有记录任何关于写操作本身的详细信息。因此一般而言，后续进行数据恢复时，只能简单地通过全对象拷贝完成，这在绝大部分情况下都会导致比按照详细日志执行增量恢复要多得多的节点间数据传输流量和本地磁盘读写流量。容易理解，这种做法一方面会严重影响正常业务，另一方面会导致数据恢复效率变低、时间变长，进而导致更高的数据损坏风险，因此无论如何都难以被称为一种优秀的数据恢复方案。

此外，PG Temp和Backfill机制的引入虽然降低了业务中断风险，但是当集群规模较小或者PG数量较少时，容易导致PG临时性地占用大量额外空间和产生大量额外读写流量。这一方面会使得Ceph本来就不高的空间利用率雪上加霜，另一方面也会反过来严重制约PG的整体数据恢复速度和效率。进一步地，由于PG可以在OSD之间进行自由迁移，如果集群状态发生振荡，容易导致PG在多个OSD之间频繁进行切换，残留大量“过时”副本，使得Peering过程中的数据一致性协商变得更加困难，Peering长时间不能收敛。

“无限风光在险峰”，征服PG这座Ceph中的珠穆朗玛峰无疑是众多Cephers心目中的终极理想。随着Ceph开发者队伍的日益壮大，特别是核心开发者的不断涌现，相信上述缺陷都将被一一攻克。

第7章

# 在线数据恢复——Recovery和Backfill

Ceph基于PG级别的日志保证副本之间的数据一致性。在实际部署时，出于可靠性考虑，一般会尽量将不同副本分布在预先设定的、位于不同故障域（例如主机）中的磁盘上。因此容易理解，这种级别的日志在实现上主要用于保证节点之间的数据一致性，即分布式一致性。

在BlueStore相关的章节中，我们已经提及，出于效率（例如并发性能）考虑，写请求一般被设计成异步的。当PG的多个副本从异常中恢复时，由于每个副本处理写请求的进度可能并不一致，所以会出现副本间数据不一致的现象，此时可以通过对比与分析各个副本保存的日志之间的差异，进行数据恢复。

由于PG能够保存的日志条目是有限的，按照能否依靠日志进行数据恢复，相应地存在两种不同的恢复方式，分别为Recovery和Backfill。其中，Recovery指对应副本能够通过日志进行恢复，也就是只需要修复该副本上与权威日志不同步的那部分对象（即降级对象）即可；而Backfill则指副本已经无法通过日志进行恢复（例如所在的OSD离线时间太久而期间产生了大量客户端发起的读写请求），或者

Ceph自身出于数据重平衡需要而执行的、以PG整体（全体对象）为目标的数据迁移过程。

显然Backfill比Recovery更加重量级，因此通常情况下通过Backfill恢复一个副本需要的时间比Recovery更长。为了尽可能缩短PG处于降级状态的时间，降低数据丢失风险，实现上我们总是优先选择尽可能多的副本参与到Recovery中来，等待Recovery完成之后，才开始着手进行Backfill恢复，等待Backfill完成之后，才最终将多余副本删除。

本章将详细介绍Recovery和Backfill两种在线数据恢复方式。由于Backfill本身依赖于Recovery，所以介绍顺序是先Recovery后Backfill。

## 7.1 Recovery

Peering完成之后，如果Primary检测到ActingBackfill中的任意一个副本（包括自身）还存在降级对象，那么可以通过日志来执行修复，这个过程称为Recovery。

### 7.1.1 资源预留

为了防止集群中大量PG同时执行Recovery从而严重影响正常业务，需要对Recovery进行约束，具体如表7-1所示。

表7-1 与Recovery相关的约束

| 配置项 | 含 义 |
| --- | --- |
| osd_max_backfills | 单个OSD允许同时执行Recovery或者Backfill的PG个数（包括Primary和副本）。<br>注意：虽然每个PG的Recovery和Backfill流程不能并发，但是不同PG的Recovery和Backfill流程可以并发 |
| osd_max_push_cost/osd_max_push_objects | 指示通过Push操作修复对象时，单个请求所能够携带的字节数/对象数 |
| osd_recovery_max_active | 每个OSD允许并发进行Recovery/Backfill的对象数 |

（续）

| 配置项 | 含　义 |
| --- | --- |
| osd_recovery_op_priority | 指示 recovery_op（Pull/Push）默认携带的优先级，这类 op 默认进入 op_shardedwq 处理，即需要与来自客户端的 op 竞争。因此，设置更低的 osd_recovery_op_priority 将使得 recovery_op 在与客户端 op 的竞争中处于劣势，从而抑制 Recovery，减小其对客户端业务的影响 |
| osd_recovery_sleep | 设置此参数可以使得 recovery_op 每次被处理之前，对应的服务线程先休眠一段指定的时间。<br>容易理解，这种方式可以显著延长 Recovery 任务的调度间隔，从而抑制 Recovery 产生的流量 |

在表7-1中，osd_max_backfills用于显式控制每个OSD能够同时进行Recovery的PG数。为了实现上述约束，在PG正式开始执行Recovery任务之前，必须先由Primary主导，发起资源预留。

为了申请Recovery资源，Primary必须先加入其归属OSD的全局异步资源预留队列。顾名思义，该队列对OSD现有资源（例如此处的Recovery资源）进行统筹，按PG的入队顺序进行分配。如果已经达到资源上限限制（例如当前可用资源数为0），则后续入队的所有资源预留请求都需要排队，等待有资源释放之后再重新申请；反之，则将资源直接分配给当前位于队列头部的PG（同时减少可用资源数），并执行该PG入队时指定的回调函数，以唤醒PG继续执行后续操作。

Primary本地资源预留成功后，需要进一步通知所有参与本次Recovery的副本远程进行资源预留，直至所有副本资源预留成功，之后才可以正式执行Recovery。

通常情况下，在生产环境中会要求集群的Recovery任务尽量不影响来自客户端的正常业务。由于集群总体IOPS和带宽有限，为了合理分配这些资源，实现上会让所有需要执行Recovery的PG也进入op_shardedwq工作队列，与来自客户端的请求一同参与竞争。这样，通过降低Recovery相关请求的权重，或者引入QoS对可用于Recovery的IOPS与带宽进行限制，我们可以有效地控制Recovery任务的资源消耗，避免对正常业务造成显著冲击。

### 7.1.2 对象修复

根据降级对象所处位置不同，Recovery一共有两种方式。

·Pull：指Primary自身存在降级对象，由Primary按照missing_loc选择合适的副本去拉取降级对象的权威版本至本地，然后完成修复的方式。

·Push：指Primary感知到一个或者多个副本当前存在降级对象，主动推送每个降级对象的权威版本至相应副本，然后再由副本本地完成修复的方式。

容易理解，为了修复副本，Primary必须先完成自我修复，即通常情况下总是先执行Pull操作，然后再执行Push操作（但是如果客户端正好需要改写某个只在副本上处于降级状态的对象，那么此时PG会强制通过Push的方式优先修复该对象，以避免长时间阻塞客户端的相关请求）。另一个必须这样处理的原因在于，客户端的请求都是由Primary统一处理的，为了及时响应客户端的请求，也必须优先恢复Primary的本地数据。

完成自我修复之后，Primary可以着手修复各个副本中的降级对象。因为在此前Peering过程中，Primary已经为每个副本生成了完整的missing列表，可以据此逐个副本完成修复。

无论采用Pull还是Push方式执行修复，其最终依据都是日志。参见表6-8，日志可以细分为以下几种类型：CLONE、DELETE、MODIFY、LOST_DELETE/LOST_REVERT等。由于日志仅仅记录了对象版本的变化历史，并没有更进一步地记录各类操作的详细信息，因此当前Recovery所采用的修复手段非常粗糙，一共只有如下两种（下面以多副本为例进行说明，纠删码修复方式略有不同）。

### （1）整对象拷贝

除DELETE、LOST_DELETE之外的操作，一律采用整对象拷贝的方式进行修复。

顾名思义，这种方式会完整地从源端拷贝包括数据、扩展属性、omap在内的全部对象内容至目的端。显然，如果目标对象内容比较多，例如包含大量的扩展属性或者omap条目，那么很难通过一次交互完成修复。此时可以通过在目的端新建一个临时对象，等对象内容全部拷贝完成后，再删除本地同名目标对象+重命名临时对象为目标对象（注意这两个操作要在同一个事务中完成）的方式加以过渡。

除了效率低下之外（例如即便只是简单修改了对象的某个扩展属性，也必须拷贝对象的全部内容），采用这种修复方式的另一个弊端在于，如果修复过程中副本所在的OSD反复上下线，那么可能导致OSD中残留大量临时对象，需要在OSD上电时予以清除。但是相应地，这会导致OSD上电时间变长，而更长的上电时间可能会产生更多的降级对象，进而导致更高的数据损坏风险。

### （2）删除对象

DELETE和LOST_DELETE操作，采用删除对象的方式进行修复。

这种修复方式本身比较简单和轻量级，但是在Luminous版本之前是（作为一种特殊的修复操作）直接放在Peering过程中同步进行处理的，因此如果PG存在大量待删除的对象，可能导致Peering需要较长时间才能完成。由于Peering过程中PG无法处理来自客户端的请求，所以上面这种处理方式增加了业务中断风险。因此，Luminous版本将删除对象作为一类正常的修复操作，转移至Recovery过程中与其他类型的修复操作统一进行处理。

出于可靠性考虑，每次Peering过程中，无论待Backfill副本数量有多少，Ceph总是会优先选出足够多（例如与存储池副本数相同）的副本参与到Recovery中来。因此，以典型的3副本备份方式为例，这意味着最差情况下可能同时有多达6个副本需要进行数据恢复。参考前面的分析，我们已经知道Recovery修复对象的主要方式是整对象拷贝（而Backfill本来就是），可以想见此时数据恢复效率有多低。

在Mimic版本之前，Recovery还存在另一个为人诟病之处：虽然Recovery是在后台执行的，但是如果客户端正好访问某个降级对象，则（出于可靠性考虑）还是需要先强制修复该对象之后，才能继续处理来自客户端的请求。具体又可以分为下面几种情况。

（1）客户端发起读请求，待访问对象在一个或者多个副本上处于降级状态

以多副本为例，由于每个副本保存的内容完全相同，只要Primary存在该对象的权威版本，那么对应的读请求就可以直接由Primary在本地完成。因此，对读请求而言，对象仅仅在副本上降级没有任何影响，但是如果在Primary上也处于降级状态，则需要等待Primary（通过Pull方式）完成该对象的修复之后才能继续处理。

为何Primary也会存在降级对象呢？出现这种现象的本质原因还是源于Ceph基于计算分布数据的设计理念，即副本分布在哪些OSD之

上、Primary又该由哪个副本来充当是根据CRUSH算出来的，只要按照CRUSH计算得到的Primary还能够通过日志修复，为了避免（需要通过PG Temp）来回切换从而增加Peering时间，Ceph就会尽可能地让其继续充当Primary，并在后台通过Recovery进行修复；反之，如果Primary必须执行Backfill，则通过PG Temp临时进行Primary切换，防止长时间阻塞客户端请求。

（2）客户端发起写请求，待访问对象在一个或者多个副本上处于降级状态

针对这种情况，要求必须修复该对象的所有降级副本之后才能继续处理该写请求。因此，仍以多副本为例，最差情况下可能需要完成两次独立的修复操作之后（即先修复Primary，再由Primary统一修复其他降级副本），才能最终响应客户端发起的写请求。

上述客户端访问降级对象、触发降级对象强制执行同步修复的做法容易导致客户端读写请求时延显著变长，进而导致集群中出现Slow Request告警。

由于Recovery已经保证有足够多的副本来固化客户端发起的写请求（即无须担心数据丢失），所以Backfill是完全异步的，基本可以做到不阻塞客户端的读写请求。当然，如果待读写的对象正好处于Backfill之中，则不可避免地仍然需要等待Backfill完成之后才能继续处理相关请求。不过由于客户端访问对象的随机性，加之PG中的对象数量一般比较多，所以这种冲突的概率较小，对客户端业务的影响也小。

而Recovery则不然，由于降级对象都是最近被客户端访问过的对象，由局部性原理我们知道，客户端后续（Recovery期间）有很大的概率会继续访问这些对象，所以Recovery容易对客户端业务产生比较大的负面影响。

### 7.1.3 增量Recovery和异步Recovery

综上，Recovery当前存在两大缺陷：一是修复手段单一并且粗糙（即整对象拷贝），占用大量磁盘带宽和网络带宽；二是如果客户端访问降级对象，需要（完全）修复这些降级对象之后才能继续处理客户端的请求，容易引起客户端业务卡顿。对此，我们可以针对性地进行改进，相应地有两种方法，分别是增量Recovery和异步Recovery。

#### （1）增量Recovery

容易理解，单个对象Recovery速度慢的原因在于会过度修复，即重复修复降级对象与其权威版本相比没有变化的部分。因此，为了加快Recovery速度，一个可行的办法是重新设计日志系统，使之能够回溯历史操作的详细信息，这样后续可以只修复降级对象与其权威版本之间的差异部分。

上述这种基于细化后的日志修复降级对象的方式称为增量Recovery，其能够提升Recovery速度的关键在于减少了待修复的数据量。当然，相应的代价是日志系统将变得复杂，修复难度也将显著增加。极端的例子比如针对一个对象反复执行多种不同操作，则后续想要基于为数众多的历史日志条目分析得出最佳修复方式绝非易事，反而不如直接进行整对象拷贝来得简单和高效。

#### （2）异步Recovery

考虑到PG本身可以在降级模式工作，也可以针对目前由客户端写请求强制触发降级对象（同步）执行修复的策略进行改造，即只需要保证满足存储池最小副本数约束的副本继续沿用当前策略即可，这样数据的可靠性仍然可以得到保障，而剩下的副本则可以采用类似Backfill的方式（但是仍然基于日志）异步在后台执行修复，不再阻塞来自客户端的写请求。

上述方案称为异步Recovery，其能够改善Recovery过程中客户端请求响应时延、提升IOPS的原因有以下两个：

1）由于写请求是异步的，大部分场景下每个等待执行Recovery的副本，其包含的降级对象都可能有所不同。因此任意一个这样的副本转而执行异步Recovery之后，与原方案相比，都将降低阻塞客户端读写请求的概率。副本之间降级对象差别越大，这种概率就越大，异步Recovery的效果也就越好。

2）原来执行（同步）Recovery的副本转而执行异步Recovery之后，如果客户端改写的对象仅在这些副本上降级，那么实际上等于变相执行了降级写。由于这部分写请求需要同步的副本数变少，时延自然得到改善。

然而遗憾的是，这些改善仅限于写请求。对于客户端发起的读请求，以多副本为例，由于其直接作用并终结于Primary，为了减小其被阻塞的概率，一种可行的办法是尽可能减少当前Primary上降级对象的数量，这可以通过改造Acting Primary的选举算法来实现。

通过Recovery完成所有降级对象修复之后，如果还有等待通过Backfill修复的副本，那么可以由Primary通知这些副本准备执行Backfill。

## 7.2 Backfill

与Recovery类似，如果有副本必须通过Backfill才能修复（这些副本构成一个集合，称为backfill_targets），则由Primary主导，先进行Backfill资源预留。资源预留成功之后，PG可以开始正式执行Backfill。

以多副本为例，理论上Backfill的最终效果是使得对应副本能够获得Primary当前状态下的一个完美拷贝。然而回到具体实现上，由于PG本身是个逻辑概念，其物理大小并没有限制（或者说只受存储空间限制），因此无法简单地作为一个整体进行原子拷贝（当然出于效率考虑我们也无法这么做）。考虑到对象是PG管理应用程序数据的基本单位，并且PG中所有对象都严格有序，因此Backfill也可以通过遍历和复制PG中的全部对象来实现。当然，为了避免Backfill过程中其他业务，特别是客户端业务对Backfill造成干扰进而导致数据不一致，还需要合理地处理Backfill与其他业务，特别是客户端业务之间的冲突。

为了追求效率，Backfill可以一次处理多个对象（但是数量仍然需要满足系统设定的约束条件）。因此，如果Backfill任务初次被调度，Primary必须先通过ObjectStore获得一个本次待执行Backfill对象的严格有序列表。

通过快速扫描（例如执行stat命令），Primary可以获得并记录这些对象的实时版本号，然后再通知所有参与本次Backfill的副本也同步扫

描，并将结果返回给自身进行汇总。Primary通过backfill_info（具体结构如表7-2所示）对Backfill的整体进度进行跟踪，peer_backfill_info（是一个backfill_info集合）则记录了每个Backfill副本的实时进度（后面我们可以看到副本之间的Backfill进度可能不一致）。

表7-2　backfill_info

| 术语 | 含　义 |
| --- | --- |
| begin | 本次扫描的起点 |
| end | 下次扫描的起点（注意：不是本次扫描的终点，即 objects 不包含 end 指向的那个对象） |
| objects | 本次扫描到的对象及其实时版本号 |
| version | 扫描发生时，PG 最新的日志版本号 |

需要注意的是，尽管每次扫描的起点一致，Primary自身收集的对象信息与每个副本收集的对象信息（包括对象数量、对象版本号等）却可能大相径庭。我们以一个简单的例子来进行说明：

1）假定PG有两个副本，当前分别分布在OSD.A（Primary）和OSD.B上。

2）$t_1$时刻，OSD.B宕掉，PG完成Peering之后处于降级状态，并正常处理客户端的读写请求。

3）$t_2$时刻，OSD.B被Monitor设置为Out，PG开始进行数据迁移。假定按照CRUSH的实时计算结果，两个副本此时分别分布在OSD.A和OSD.C上。容易理解，位于OSD.C上的（全新）副本将通过Backfill完成数据同步。

4）$t_3$时刻，位于OSD.C上的副本Backfill已经完成，OSD.C宕掉，OSD.B重新上线。假定按照CRUSH的实时计算结果，承载该PG的两个

副本（所在的OSD）再次切回OSD.A和OSD.B。由于OSD.B在$t_1$至$t_3$时刻之间一直不在线，所以容易理解其仍然保留了PG的一个副本（Primary在完成Backfill之后无法通知其及时进行删除）。又假定在此期间，客户端执行了大量读写操作（即产生了大量新日志，例如超过PG所能允许保留的最大日志条目），则位于OSD.B上的这个副本的内容必然已经过时，并且只能通过Backfill恢复。

针对上面这个例子，我们可知此时位于OSD.B上的这个副本，其包含的对象视图必然已经与Primary不同，并且可以细分为如下几种情形：

·缺少某些对象（Missing Object）

·存在某些多余对象（Redundant Object）

·对象存在，但是与Primary版本号不一致，即对象已经过时了（Obsolete Object）

·对象存在并且与Primary版本号一致（Normal Object）

由Ceph的一致性策略，我们知道Primary保存的是权威副本（注：这里特指Backfill的场景。其他场景，例如Recovery可能无法得出这个结论），因此相应的处理策略如下：

·拷贝对象

·删除对象

·修复对象

·跳过对象

除了删除对象之外，拷贝对象与修复对象本质上没有什么区别（例如我们总是可以在目的地先删除然后再“重新”创建对象，因为删除一个不存在的对象总是被允许的），可以直接复用Recovery的Push机制来完成。而删除对象因为代价较小，可以通过向每个副本发送单个消息来集中进行删除。这里的难点在于如何更新对应副本的统计信息。

原则上，驻留在每个副本Info中的stat字段应当如实反馈该副本的实时对象统计信息，例如，存在多少有效对象（指经过Primary确认的）、这些对象共计占用了多少逻辑空间等。对于每个即将从头开始Backfill的副本，Primary会在Peering完成之后将其Info中的stat清零。此后，随着Backfill推进，Primary会不断收集相关的统计信息并更新对应副本的stat。与上述同步对象的策略相对应，具体又可以分为如下几种情况：

·拷贝或者修复对象；等待拷贝或者修复完成后，再收集这些对象的统计信息并更新对应副本的stat。

·删除对象；不需要更新stat。

·跳过对象；由前面的分析，跳过的对象是正常对象，虽然本身不需要重新同步至副本，但是由于其相关的统计已经被Primary擦除，因此仍然需要重新计入副本的stat。

达到最大并发Backfill对象数目限制之后，Primary将结束本次调度。由于Backfill任务优先级较低，后续有可能一直被其他业务，特别是客户端业务抢占，这意味着Backfill两次调度的时间间隔基本不可预料。因此，在触发指定数量的对象执行Backfill之后，Primary还必须通过一个名为last_backfill_started的指针来记录和跟踪Backfill的实时进度，同时使用一个名为backfills_in_flight的集合，对所有在途的对象（指Missing Object或者Obsolete Object）进行追踪。

如果需要Backfill的对象数量比较多，为了避免类似重启等意外因素导致Backfill每次都从头开始，每个参与Backfill的副本还必须利用自身Info中的last_backfill字段，来记录和跟踪自身的Backfill进度，作为下次从异常中恢复时继续执行（如果能够）Backfill的依据。

为了防止两次调度之间，backfill_info当中残留的、等待Backfill的对象信息已经过时（例如在此期间客户端针对其中的某些对象又执行过写操作），当Backfill任务再次被调度时，必须由Primary确认是否需要重新扫描，判断规则如下：

·与上次扫描发生时Primary记录的最新日志版本号相比，当前最新日志版本号没有发生变化（例如系统处于空闲状态，无客户端访问），则无须重新扫描。

·上次扫描记录的最新日志版本号大于等于Primary当前保存的最老日志版本号，说明日志完整记录了两次扫描之间所有写请求信息，此时可以直接通过遍历日志获得相关对象的最新版本信息（如果有的话）。

·上次扫描记录的最新日志版本号比Primary当前保存的最老日志版本号还要小，说明两次扫描之间的某些写请求的日志已经丢失（被截断），此时只能由Primary重新发起扫描。

如果不需要扫描或者扫描已完成，Primary可以以last_backfill_started作为起点，再次触发新一批对象执行Backfill。如果之前某些在途的对象已经完成Backfill，Primary将更新相关统计，并通知对应副本也同步更新自身的Backfill进度和统计。

假定最大并发数为2，我们以一个backfill_targets规模为3的例子来说明上述Backfill过程，如图7-1所示。

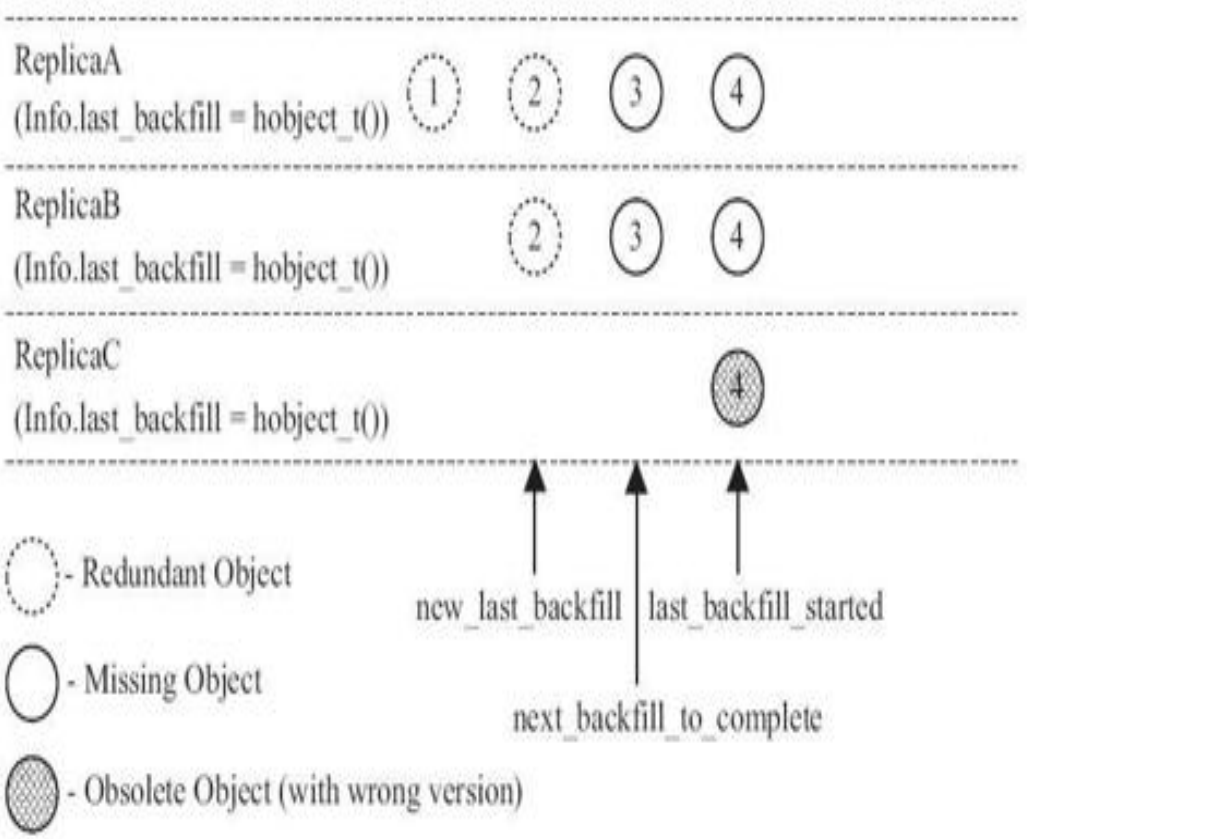



图7-1　3个副本（起点一致）参与Backfill——第1次调度

由图7-1可见，删除对象因为代价较小，并不受最大并发数限制，因此本次调度Primary实际上一次性处理了4个对象。

·Obj1：通知ReplicaA删除

·Obj2：通知ReplicaA和ReplicaB删除

·Obj3：通知ReplicaA和ReplicaB修复

·Obj4：通知ReplicaA、ReplicaB和ReplicaC修复

通知对应的副本删除冗余对象后，所有副本的last_backfill将被同步更新，指向Obj2。需要注意的是，上面这种情形中，虽然删除也是异步的，但是Primary并不需要等待对应的副本回应删除成功即可通知其更新Backfill进度。原因在于这两个消息的优先级相同，因此对应的副本必然先收到删除消息，成功地执行了对象删除之后才能更新自身的last_backfill（与stat）。反之，如果删除没有成功，那么对应的last_backfill必然也不会更新，因此无论如何都不会导致Backfill进度错乱。

仍然针对上面这个例子，当Backfill再次被调度和执行后，其结果如图7-2所示。

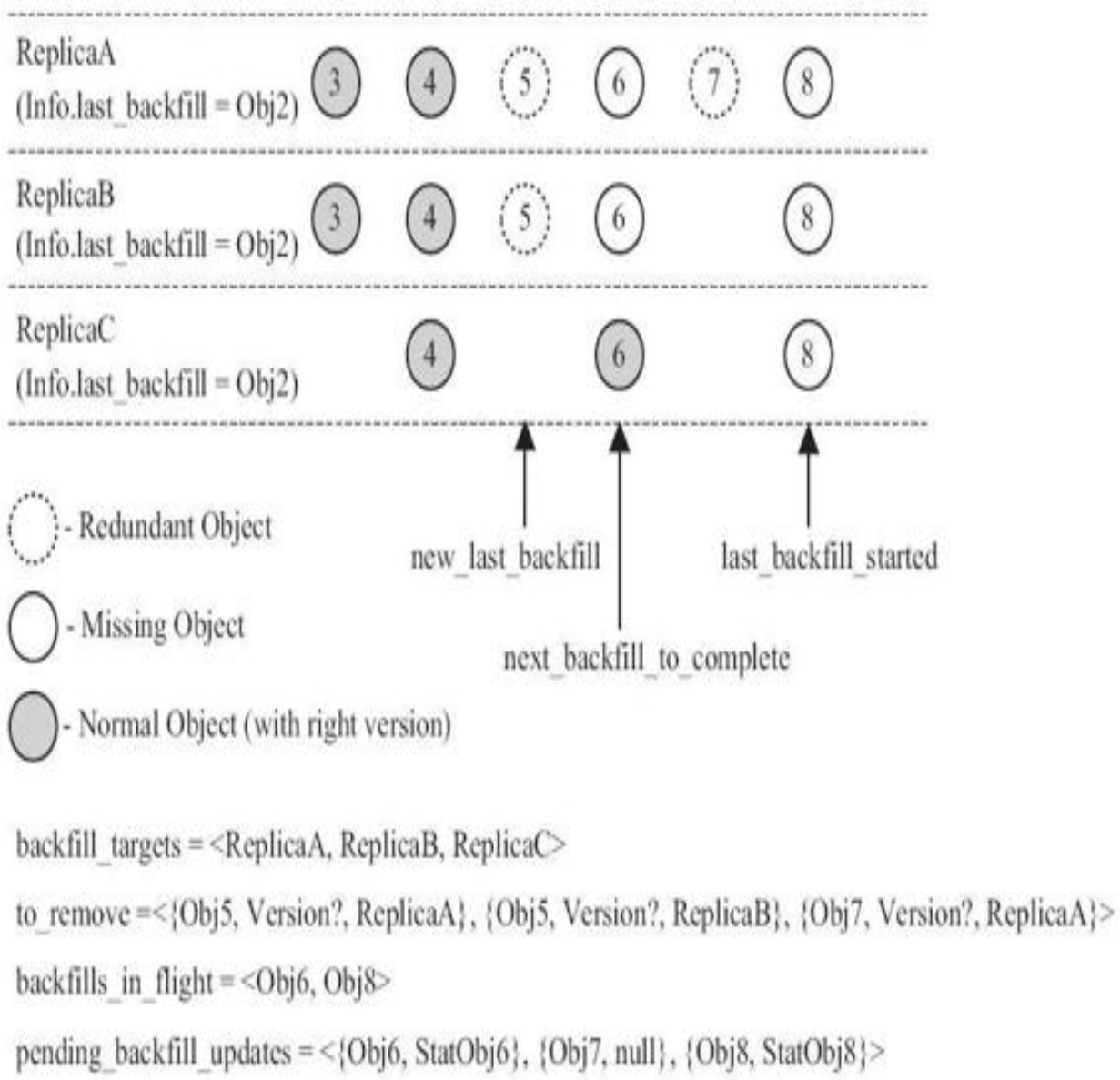



图7-2　三个副本（起点一致）参与Backfill——第2次调度

由图7-2可见，本轮调度结束后，所有副本的last_backfill将同步更新，指向Obj5。以此类推，随着Backfill不断被调度，last_backfill将不断指向更大的对象，直至指向max对象。此时Primary将向所有副本发送BACKFILL_FINISH消息，指示Backfill最终完成。

当然，在生产环境中，Backfill过程并不都是一帆风顺的，例如，假定某个PG当前Backfill尚未完成，由于集群拓扑结构变化导致PG的

Up又发生了变化（例如又加入了一个全新的OSD），那么此时就会出现两个或者多个副本Backfill进度参差不齐的情况。图7-3展示了一个相关的例子。

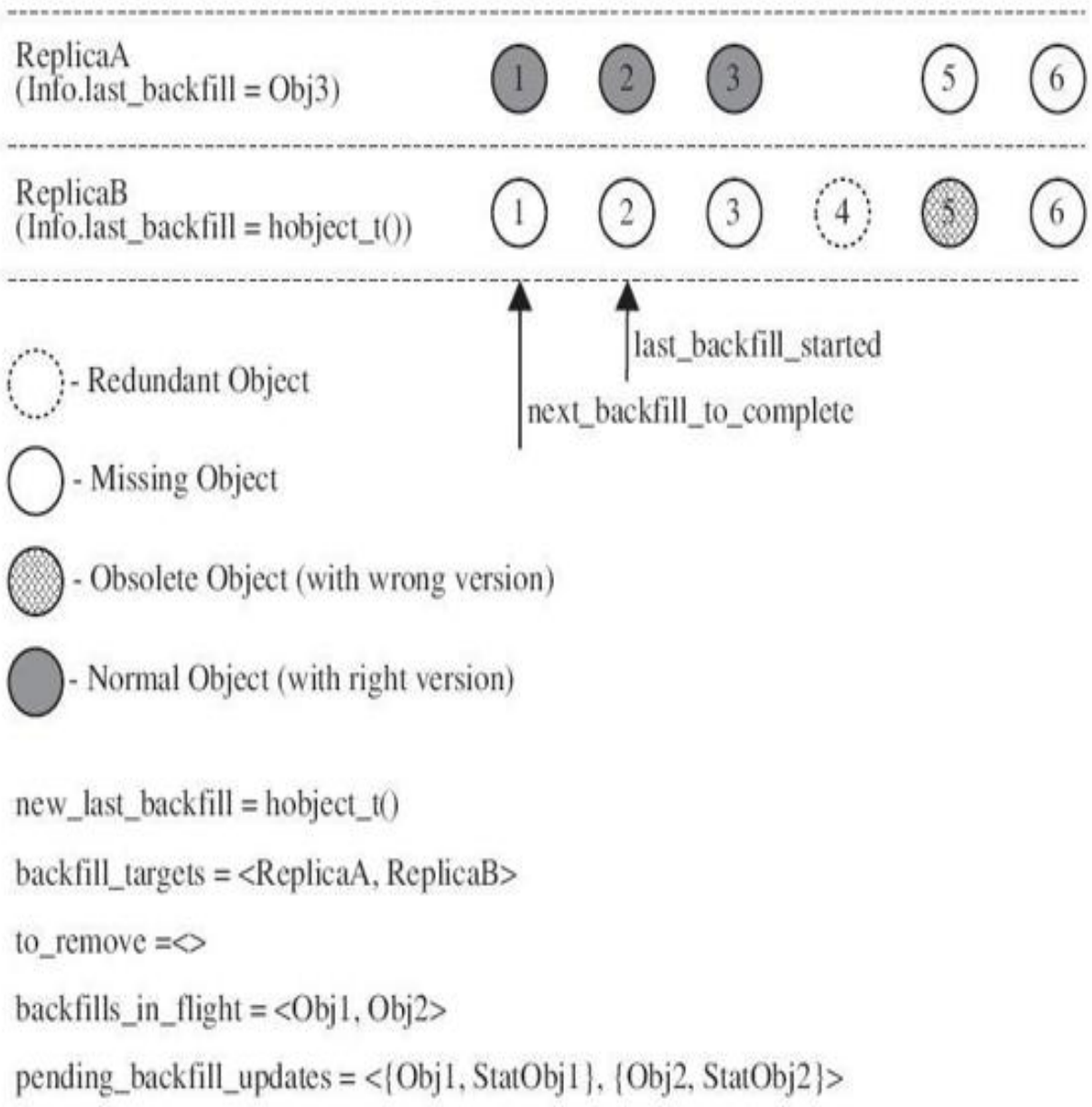



图7-3　两个副本（起点不一致）参与Backfill

由图7-3可见，前面3个对象只需要单独Backfill至ReplicaB中，而从Obj5开始，两个副本的Backfill进度将重新变得同步（即此后可以将一

个对象同时Push到两个副本之中完成修复）。因此，执行Backfill时，为了保证Backfill效率最高，每次重启Backfill之后，Primary总是从当前Backfill进度最落后的那个副本开始进行（对象）同步。

在一些特殊情况下，例如副本在Backfill过程中异常下线然后又重新上线，那么在随后的Peering过程中，Primary将决定其后续的恢复方式：如果离线时间不长，期间所有发生过的写请求，其对应的日志条目都可以从权威日志找到（没有被截断），那么可以在正常完成Recovery之后从上次的断点处继续Backfill，反之则必须完全重启Backfill。

在生产环境中，如果PG包含的对象数量比较多，同时客户端业务比较繁忙，那么该PG的Backfill过程有可能长时间无法完成，导致对应OSD上其他PG的Recovery或者Backfill都被阻塞（默认情况下我们限制每个OSD能同时进行Recovery或者Backfill的PG数为1）。由于Recovery的紧迫程度要高于Backfill，并且PG之间所存储数据的重要程度也有可能大相径庭，所以，相应的解决办法是允许Backfill中断和被抢占。幸运的是，由前面的分析我们知道，现有Backfill已经支持类似“断点续传”的机制，这为实现Backfill抢占功能奠定了坚实的理论基础。

Backfill完成之后，Primary将向Monitor发送一个新的PG Temp变更请求（此请求携带的PG Temp为空），用于将Acting重新调整为Up。再次经过Peering后PG将最终进入Active+Clean状态，此时可以删除不必要的副本，释放其占用的存储空间。

## 7.3 总结和展望

在线数据恢复是存储系统的重要课题之一。

与离线恢复不同，在线数据恢复的难点在于数据本身一直处于变化之中，同时在生产环境中一般都有兼顾数据可靠性和系统平稳运行的诉求。因此，如何合理地处理各种业务之间的冲突，恰当地分配各种业务之间的资源（例如CPU、内存、磁盘和网络带宽等），在尽可能提升数据恢复速度的同时，降低甚至完全避免数据恢复对正常业务造成干扰则显得至关重要。

按照能否依据日志执行恢复，Ceph将在线数据恢复细分为Recovery和Backfill两种方式。通常意义上，两者分别用于应对临时故障和永久故障，当然后者也常用于解决由于集群拓扑结构变化导致的数据不平衡问题。

理论上，更快的数据恢复速度必然消耗更多资源，从而不可避免地对其他业务，特别是正常业务造成更大干扰。因此，在可用于数据恢复的资源不变的前提下，为了提升数据恢复速度，必须减少待恢复的数据量；为了降低数据恢复对于正常业务的干扰，则必须合理地解决两者之间的冲突，保证冲突发生时，正常业务不被阻塞。相应地，Ceph通过引入增量Recovery与异步Recovery功能进行应对。

毫无疑问，增量Recovery和异步Recovery将是后续Ceph版本的亮点功能。然而遗憾的是，其本质都只是针对单个PG的数据恢复流程进行优化。回到生产环境中，即便集群出现少量OSD故障，往往也会导致成百上千个PG需要同时进行在线数据恢复，如何保证这些PG以一个最优的顺序进行调度，从而获得一个总体上的最短数据恢复时间则是更大的挑战。

从上述角度而言，在线数据恢复之于Ceph依然任重而道远。

第8章

# 数据正确性与一致性的守护者——Scrub

上一章我们讨论了Ceph的在线数据恢复策略，其主要依据是PG日志，具体实施过程中则假定所有通过ObjectStore读到的数据总是可靠的。然而，实际上任何我们使用的计算机组件都不是完美的，都有可能产生静默数据错误，因此即便通过Recovery或者Backfill完成了数据恢复，每个副本仍然可能存在本地数据错误，副本之间也仍然可能存在数据一致性问题。

由于静默数据错误的产生具有极大的偶然性和随机性，如果同一份数据在位于不同故障域下的磁盘上存有多个备份，那么可以认为多个备份同时出现静默数据错误的概率几乎为零。因此，在存在备份的前提下，如果能够准确检测和捕获静默数据错误，则可以利用副本之间的冗余性（通过复用Recovery机制）来进行修复。

简言之，Scrub是存储系统中用于保障数据正确性与一致性的一种机制，其核心要素包括两个，即检错和纠错。Scrub能够捕获数据错误（包括但不限于静默数据错误）的主要依据是校验和，即要求任何数据写入存储设备之前，先通过哈希计算出原始输入的信息摘要（即校验和），然后将两者分开进行存储。为了避免相互干扰，校验和一般

需要采用自身具有检错能力的存储系统（例如数据库）进行转存。这样，后续再次读取原始数据时，通过重新计算校验和，并与前一次存储的校验和进行比对（此时校验和总是可靠的），即可判定有无数据错误发生。

由于完整的一次Scrub过程涉及遍历系统中所有数据，所以Scrub是磁盘带宽密集型任务，这意味着Scrub进行时对正常业务（如有）必然会产生比较大的干扰。考虑到元数据非常重要同时数量也相对较少，因此在执行Scrub时，为了获得更快的执行速度同时减少对正常业务的干扰，也可以仅扫描元数据。当然，这种方式与面向所有数据的深度扫描（即Deep-Scrub）相比，能够捕获的错误有限，因此也被称为浅扫描。

本章介绍Scrub的基本概念、具体流程以及Luminous版本最新引入的Scrub抢占机制。

## 8.1 Scrub的指导思想

与数据恢复类似，Scrub也分在线和离线两种方式。商用存储系统作为重要的基础设施，一般都有年宕机时间不超过几个小时的要求，因此离线Scrub应用场景非常受限，本章主要讨论在线Scrub（如无特殊说明，下文中的Scrub均指在线Scrub）。

通常情况下，集群数据会一直处于变化之中，例如，任意时刻都可能存在对象的创建、删除或者修改操作。因此，为了捕获数据错误和一致性问题，Scrub必然需要周期性地执行。同时，为了能够完整地完成一次深度扫描，则要求Scrub基于合适的粒度、以合理的组织方式执行。

如何确定Scrub的粒度呢？由于RADOS的基本数据组织单元是对象，所以直觉上也许会认为以对象为粒度来执行Scrub是比较合理的选择，然而这会带来两个问题：首先，对象的生命周期并不固定，加之一个集群能够创建的对象数量理论上只受存储空间限制，因此如果集群中存在大量对象，逐个对象执行Scrub效率太低；其次，由于每个OSD诸如CPU、磁盘带宽、网络带宽等可用资源严格受限，以对象为粒度执行Scrub，相关资源的控制难度较大。与之相反，由于集群中PG数量相对固定，并且PG的生命周期也相对稳定，加之每个OSD的

可用资源都围绕PG进行组织，因此将Scrub任务交给PG，由PG按照各类任务的优先级统一进行调度才是更合理的选择。

除了对本地数据的正确性进行校验（前面已经提及，这依赖于校验和），Scrub也负责对副本之间的一致性（即分布式一致性）进行校验。

首先，也是最重要的，如果副本之间本身就存在数据不一致，例如PG正在进行数据恢复或者数据迁移等，那么显然无法正常进行Scrub，因此Scrub只能在PG处于Clean状态时进行。

其次，我们已经知道，在存储系统中，为了追求效率，写操作一般被设计为异步的。因此只要某个PG存在写操作，则该PG的多个副本之间存储的对象总是会略有差异。比如，某个对象在Primary上已经创建成功，但是在其他副本上尚未创建；又比如，某个对象在Primary上已经被改写，而在其他副本上仍然停留在前一个版本等。为了避免这种由写操作导致的临时性不一致对Scrub造成干扰，通常的做法是在Scrub进行时对所有波及的对象都加锁，即不允许客户端对它们执行写操作。

在最初的设计中，这种锁的粒度被设计为整个PG级别。容易想见，如果PG中的对象数量非常多，这种粒度的锁将导致整个PG的写请求长时间被阻塞。更糟糕的是，由于Ceph随机分布数据，每个PG都会或多或少地保存同一个存储池内所有应用程序所依赖的数据分片（即对象），所以这种粒度的锁可能导致所有应用程序（例如虚机）同时受到影响。

由于PG中的所有对象都可以严格排序，并且只要没有新的对象被创建、老的对象不消亡，那么这些对象在PG中的顺序就是固定的。因此，假定PG能够一直处于稳态（即没有对象创建或者删除操作），那

么Scrub就可以按照排序后的对象列表，每次选择若干对象，分批执行，直至Scrub最终完成。

与原始Scrub相比，上面这种新型的Scrub模式称为Chunky-Scrub。在此模式下，Scrub加锁的粒度可以根据客户自身需求进行量身定制，例如要求降低Scrub对正常业务的影响，则只需要减少每次执行Scrub的对象数量即可。当然，随着Scrub加锁粒度变小，PG执行一次完整Scrub的时间也将不可避免地变得更长。

然而实际上PG不可能一直处于稳态，也就是说在整个漫长的Scrub周期中，除了对PG整体加锁之外，我们无法阻止客户端（或者PG自身出于其他目的）创建和删除对象，那么这是不是意味着Chunky-Scrub不可能得出正确结论呢？答案当然是否定的，原因如下：

1）Scrub并非一种实时纠错机制，而是一种事后审计机制。换言之，在大部分情形下Scrub不一定能找出数据错误或者不一致，更不能找出系统中所有的数据错误与不一致（因为数据总是在变化之中）。因此每个Scrub周期多扫描或者漏扫描几个对象其实并无大碍，只要正常更新相关的统计即可（这个结论也适用于某些对象在刚被Scrub扫描完之后又被客户端修改的情形）。

2）由Scrub的指导原则我们知道，Chunky-Scrub能够实施的前提在于PG中所有对象都严格有序。因此，只要对象增减不会导致其他对象的相对顺序发生变化，则不会对Chunky-Scrub造成本质影响，而这显然是满足的。由于对象排序算法只用到了每个对象自身的特征值，而创建或者删除一个对象并不会造成其他对象标识的特征值发生变化，所以也就不会造成其他不相干对象的排序结果发生变化。更进一步地，如果创建或者删除的对象比Scrub本次扫描的最小对象还要小，那么只需要更新相关的、对象级别的统计即可，新创建的对象，其内容将在下次完整的Scrub任务中被重新扫描；反之如果创建或者删除的

对象比Scrub本次扫描的最大对象还要大，那么它们后续可以被本次Scrub任务正常扫描到。

解决了Scrub锁粒度问题之后，我们接下来讨论Scrub校验数据正确性依据——校验和。

在引入BlueStore这种新型ObjectStore之前，这个校验和是由Primary负责计算与更新的，主要涉及以下3种典型场景：

1）整对象写（包括新写与覆盖写）：每次基于写入的完整对象数据直接计算校验和即可。

2）追加写：为了避免需要读取前面已经写入的数据，一般可以采用CRC的方式计算校验和。此时可以基于前一次计算已经得到的校验和与本次追加写入的数据重新计算校验和。

3）部分覆盖写：此时需要先读取对象的已有数据，在内存中执行覆盖写，然后重新计算校验和。

在最后一种场景中，为了正常更新校验和，需要完整地读取对象的已有数据，代价巨大。因此实现时出于效率考虑将不再继续更新对象的校验和，同时已有的校验和（本次写入后将变得无效）也要随本次写入一并擦除。由于大部分业务都不可避免地涉及覆盖写，这意味着系统中的大部分对象实际上并没有PG级别的校验和，因此在执行Scrub时也就无法校验本地数据的正确性。

显然，这是Scrub的一个重大缺陷。在Ceph诞生10年之后，这个缺陷在Sage重新设计BlueStore时终于得到了应有的重视，解决的办法却是出人意料的简单：为了避免覆盖写场景下由于更新校验和而产生额外的读惩罚，BlueStore从不执行覆盖写操作。

除此之外，BlueStore更进一步地将本地数据正确性校验扩展到了Scrub机制之外，试想我们能否不依赖于Scrub，而是依赖于客户端自身的读请求自然地对数据执行校验呢？诚然，这是一个相当具有吸引力的设想，因为它从原理上可以替换掉Scrub的两大用途之一的数据正确性校验，实现的关键则在于能否继续沿用Scrub所采用的PG级别的校验和方案。

与Scrub主动去读取对象的全部内容不同，客户端的读行为存在不可预期性，因此类似PG采用“基于对象全部数据产生一个单独校验和”这种做法的困境在于，如果不是整对象读，那么这个校验和以及相应的校验机制就毫无用处。同样基于效率考虑，我们不可能为了验证数据的正确性，而在客户端仅仅读取对象的部分内容时转而要求BlueStore去读取对象的全部内容。因此沿用老的、PG级别的校验和方案无法顺利完成上述设想。

一个顺理成章的改进措施是，如果能够想办法使得校验和与原始数据长度一致并且一一对应，那么上述困境自然迎刃而解。当然如果校验和本身就与原始数据一样长，则其实际效果等同于完全拷贝，那么采用复杂的校验算法也就失去了意义。幸运的是，任何类型的存储介质，出于效率考虑，访问其数据必然存在一个最小粒度限制。因此，如果这个最小粒度足够大，那么很容易验证此时针对每个最小粒度的数据单独设置校验和仍然具有实践意义。例如，假定这个最小粒度为4kB，校验和为32位（即4个Byte），此时存储校验和所需的空间只是这个最小粒度的1/1024，并且无论客户端读取对象数据粒度如何，都不必因为需要执行数据校验而产生额外的读惩罚，存储系统最受关注的性能也因此得到了保障。

至此，我们已经完美解决了如何校验本地数据正确性的问题。在分布式存储系统中，通常情况下，由于数据还需要在位于多个不同故障域下的存储设备之间进行复制，所以类似节点间的数据传输错误、

数据不同步等异常同样不可避免。因此，除了本地副本的数据正确性之外，Scrub还需要进一步解决副本之间的数据一致性问题。

与本地数据正确性校验不同，面对数据的分布式一致性问题，我们不可避免地陷入了窘境：当副本之间的不一致真正发生时，我们如何让它们再次达成一致呢？参考Peering，可行的解决办法是引入一套仲裁机制，选出一个权威副本，然后再将其他副本与之进行比对和修正。那么权威副本又该参考怎样的标准去选取呢？仔细想想便会发现实际上很难有定论，例如我们简单规定将没有本地数据错误的副本作为权威副本，那么有可能出现两个副本都没有本地数据错误但是相互之间仍然数据不一致的情况。进一步地，在某些特殊场景下，即便我们没有一个完全正确的副本，我们仍然希望能够从中选出一个相对较好的副本作为权威副本，面对这类（显然非常合乎情理的）诉求同样可以举出很多反例使之无法达成。因此，总的来说，Scrub之于一致性，特别是分布式一致性，能起的作用非常有限。

如果单纯考虑效率，例如为了避免在副本之间传递大量数据直接进行比对，那么Scrub执行分布式一致性校验仍然可以采用校验和的方式，如果每个副本的校验和都一致，即可认为所有副本都是一致的。当然能这样做的前提是所有副本保存的内容都是完全相同的，因此容易预见这种方式对于纠删码存储池不适用。再者，即便针对多副本，由于覆盖写的缘故，对象多半已经没有PG级别的校验和。因此为了执行分布式一致性校验，即使BlueStore已经对所有读取的对象数据做了自校验，仍然需要Scrub在扫描对象时重新计算校验和。当然，针对一些本身已经比较简短的内容（例如对象的扩展属性），我们仍然可以直接将其在副本之间进行传输与比对，其代价相较采用校验和的方式而言更小，也更可靠。

## 8.2 Scrub流程详解

上一节我们讨论了Scrub的一些指导思想，本节我们讨论Scrub的具体实现，其整体上采用流水线设计，分为资源预留、范围界定、对象扫描、副本比对、统计更新与自动修复5个阶段。

除了数据正确性，Scrub还要校验数据一致性，所以Scrub必然要求对象的每个副本都参与其中。由于每个OSD承载的PG数量较多，而Scrub行为是系统以PG为粒度自动触发的，为了Scrub能够正常进行，必须对每个OSD能够同时执行Scrub的PG数量进行限制，否则有可能产生Scrub风暴，导致Scrub和其他业务都无法正常进行。这种限制最终通过跨OSD的全局资源预留机制来实现，接下来我们将详细介绍。

### 8.2.1 资源预留

与处理客户端I/O类似，考虑到Scrub过程中同一个PG的所有副本都必须参与其中，为了使得Scrub能够正常进行（例如不至于乱序），特别是方便捕获和处理Scrub与其他业务之间的冲突，我们必须选择Primary作为Scrub的发起者和协同者。因此，在Scrub真正开始之前，必须由Primary首先发起跨OSD之间的Scrub资源预留。主要过程如下：

1）Primary向自身所在的OSD申请Scrub资源预留，直至资源预留成功。

2）Primary通知所有副本进行远程资源预留，直至每个副本资源预留成功。

假定每个OSD能够同时执行Scrub的PG数为1，那么这里存在一个潜在的死锁问题，我们举例说明（以2副本为例）。

1）假定有OSD.A和OSD.B，某一时刻有PG.A（其Primary在OSD.A上，副本在OSD.B上）和PG.B（其Primary在OSD.B上，副本在OSD.A上）同时发起Scrub。

2）PG.A在本地请求资源预留成功，通知OSD.B上的副本进行远程资源预留；与此同时，PG.B在本地请求资源预留成功，通知OSD.A

上的副本进行远程资源预留。

在上面这个例子中，因为OSD.A和OSD.B上的资源都已经耗尽并且处于相互等待的状态，所以本质上是一种“死锁”状态。当然，解决的办法也比较简单：采用冲突退避的方式即可，即只要任意一个副本资源预留不成功，那么对应PG的Scrub就将被完全取消，之前已经预留成功的资源也需要全部释放，然后等待Primary发起重试。

当然，如果不做一些特殊处理，上面这种解决方案不免陷入另一个极端。我们继续以上面的例子为基础进行说明。

1）PG.A在本地请求资源预留成功，通知OSD.B上的副本进行远程资源预留；与此同时，PG.B在本地请求资源预留成功，通知OSD.A上的副本进行远程资源预留。

2）PG.A检测到远程资源预留失败，释放本地预留资源，同时取消本次Scrub；PG.B检测到远程资源预留失败，释放本地预留资源，同时取消本次Scrub。

3）下一时刻，OSD.A和OSD.B上又有两个PG需要同时执行Scrub……

可见，最终的结果同样是Scrub始终无法正常进行。通过简单分析我们可知，出现后面这种现象的根源在于OSD之间调度PG执行Scrub的时机：如果OSD之间同步调度，因为全局Scrub资源非常有限，很容易产生大量冲突和退避；反之如果每个PG执行Scrub的时机完全随机，并且能够做到在整个时间轴上（近似）均匀分布，则每个PG执行Scrub时所请求的资源基本上都能够一次性得到满足。

完成资源预留之后，PG可以正式开始执行Scrub，为此，需要首先确定Scrub的起点（注意：这是一个对象）。遗憾的是，通过前面的分析，虽然我们已经知道每个PG中的所有对象都严格有序，但是PG

自身却并没有记录这些对象的顺序信息。为此，我们只能要求并依赖于ObjectStore提供按照某种顺序（比如从小到大）遍历PG中所有对象的方法。

### 8.2.2 范围界定

在本书第1章我们已经介绍过，为了应对PG分裂，BlueStore为所有对象维护一个全局命名空间。在具体实现上，BlueStore将所有对象的元数据以键值对的形式统一存储在RocksDB数据库中。每个键都是一个长度不限的字符串，基于对象的完整标识（即包含了对象标识中的所有特征值，以保证绝不重复）生成。为了保证前后端排序结果的一致性，BlueStore使用对象标识中每个特征值生成键的顺序和第1章介绍的对象排序算法严格对应。因此，原理上BlueStore很容易满足有序遍历某个PG中所有对象的需求。

由于BlueStore采用全局命名空间，显然，如果没有传递PG对应的身份信息，BlueStore无法凭空找到某个PG中对象的起点（即最小的那个对象）。再者，由于对象标识并没有直接关联对象归属的PG信息，而是通过stable_mod间接引用，因此如果某个PG需要通知BlueStore查找当前归属自身最小的那个对象（也就是Scrub的起点），则首先应当构造并传递如下的一个对象标识。

·对象名：空

·命名空间：空

·存储池标识：PG归属的存储池标识

·32位哈希：X，其中X stable_mod PGID=PGID

·快照标识：0

·分片标识：0（表示无效的分片标识）

·版本号：0（表示无效的版本号）

这个对象标识表示对应PG理论上可以存储的最小对象，BlueStore可以据此查找第一个不小于该对象的真实对象，并返回给PG。

通过BlueStore得到一种顺序遍历PG中所有对象的方法之后，我们可以基于自定义的粒度执行Scrub。例如，使用默认配置，我们将限制一次扫描的对象数量是[5，25]。为什么这里一次扫描的对象数量是一个范围而不是一个固定的数值呢？原因在于，除了客户端创建的正常对象之外，PG还存在各种各样的“异常”对象。

首先，引入纠删码之后，因为需要支持回滚功能会产生备份对象。这类对象在Scrub过程中可以顺便校验其时效性，例如通过与PG内置的、需要保留的最小备份版本号对比，所有残留的、已经过时的备份对象可以趁此机会删除，自然也就不再需要继续参与到后续的Scrub过程中来。

其次，除了纠删码会产生备份对象，Ceph还会因为各种各样的原因产生一些临时对象。例如，Recovery过程中，如果某个对象内容太多，无法通过单次交互完成修复，那么需要在目的地先生成一个临时对象进行过渡，并在最后一步通过删除原始对象的过时或者错误版本（如有），再通过对象重命名将这个临时对象替换为目标对象以完成修复。显然这类对象并不需要进行Scrub，因此需要在Scrub时予以跳过。

需要注意的是，由于存储池默认从0开始编号，为了区分这类临时对象和正常对象，所有临时对象关联的存储池标识会被人为地设置成一个负数。按照对象排序算法，这意味着同一个PG中所有临时对象比任意一个正常对象都要“小”。另一方面，由于BlueStore基于字符串形式的对象标识执行排序，所以需要先将存储池标识转换为64位的无符号整数，然后再转成字符串形式。这会导致一个潜在问题：例如，-1转化为64位无符号整型数后，其对应的字符串表达形式为“FFFFFFFFFFFFFFFF”，因此在BlueStore的排序算法中，所有临时对象将排在所有正常对象之后（也就是所有临时对象都比任意一个正常对象要“大”），与前端排序算法冲突。

当然，解决上面这个问题的办法也比较简单，只需要采用类似二进制补码的方式，将已经强制转换为64位无符号类型的存储池标识的符号位再次翻转即可（亦即再加上0x8000000000000000ull）。这样，-1编码后将变成7FFFFFFFFFFFFFFF，而正常的存储池标识编码后的序列为8000000000000000、8000000000000001、8000000000000002等，可见，经过处理之后所有临时对象都比正常对象小。后续PG通过BlueStore查找Scrub的起点对象时，通过传入自身归属的存储池标识，所有临时对象都可以被有效地跳过，而不会对Scrub造成干扰。

最后，我们来讨论克隆对象。

快照功能的引入导致在针对快照对象修改时，可能会产生克隆对象。大部分场景下，克隆对象只携带了原始对象（即head对象）的部分信息，例如，一种设计良好的快照机制应该仅仅针对被修改的部分执行克隆，而不需要针对原始对象执行完整克隆。就这个角度而言，克隆对象必然要依附于原始对象而存在，这意味着在一次Scrub过程中，必须要将原始对象及其附属的克隆对象作为一个整体处理。因此，在执行Chunky-Scrub的过程中，我们需要对每次执行Scrub的对象

集进行筛选，使得归属于同一个原始对象的所有克隆对象，尽量在一个分片中与原始对象一起完成Scrub。

由于每个对象能够产生的克隆对象数量并不固定，当我们要求BlueStore每次返回固定数量的对象时，可能会出现克隆对象和其关联的原始对象被割裂的现象。例如，假定我们限制BlueStore每次固定返回25个对象（无论什么对象），那么某次查询时，BlueStore可能返回的对象数量序列为（6，7，8，4），括号中的数字为每个原始对象及其包含的克隆对象数量。假定最后一个对象共包含8个克隆对象，那么我们知道此时4这个数字仅仅表示原始对象产生的4个克隆对象，无法满足我们前面为Scrub设定的约束条件（即“每个原始对象和其克隆对象需要被同时处理”，也就是说此时最后一个数字应当为9）。因此，最后这4个对象需要在本次Scrub中予以剔除（推迟至下个分片与原始对象以及其他克隆对象一起进行Scrub），这意味着最终将仅有21个对象参与到本次Scrub中来。

### 8.2.3 对象扫描

确定好本次待扫描的对象集后，Primary可以通知所有副本（也包括自身）单独针对这些对象开始扫描，输出扫描报告并提交至Primary汇总。详细过程如下。

首先，针对输入对象集中的每个对象，每个副本依次执行以下步骤：

1）stat命令。如果不存在（这意味着副本之间就对象集中每个对象的存在性就可能产生不一致），则直接跳过（后续Primary会检测到该对象在这个副本上缺失，将复用Recovery机制进行修复）；如果出错，则记录错误类型并直接返回；否则继续往下处理；

2）记录对象大小，读取对象的所有扩展属性，如果没有开启深度扫描，则直接返回；否则继续往下处理；

3）开始执行深度扫描，依次读取对象的所有用户数据与omap，分别计算并记录各自的校验和，中途如果检测到数据错误（依赖于BlueStore，例如返回EIO），则记录错误类型并直接返回。需要注意的是，如果这个过程中Scrub捕获到对象存储的omap条目过多或者整个对象的omap消耗的存储空间过大，将在Scrub结束时触发相关告警。

其次，完成对象集中所有本地对象扫描之后，可以由每个副本自身预先做一下力所能及的修复，例如：

### （1）修复快照

因为诸如快照删除之类的管理命令都是异步完成的，所以副本中可能还残留了某些过时的克隆对象。可以通过如下方式找出这些克隆对象，并同步进行清理。

1）将所有对象使用一张基于C++标准实现的map进行存储。由于map是一类有序容器，又由于按照对象排序算法，我们知道每个原始对象比所有由其衍生的克隆对象都要大，因此如果这张map中同时包含某个原始对象及其关联的克隆对象，为了保证先找到原始对象，我们总是对map执行逆序遍历。

2）如果是原始对象，则尝试从其扩展属性中解析出SS属性。

3）如果是克隆对象，则首先，校验其原始对象是否存在，如果不存在，则产生一条告警；其次，检查其是否为原始对象的有效克隆，如果不是，则同样产生一条告警；再次，检查该克隆对象关联的快照标识集是否与原始对象记录的数值相等，如果不相等，同样产生告警，并以原始对象为准，同步进行修复。

### （2）修复对象标识

再次对map中的所有对象执行逆序遍历，尝试读取并解析对象扩展属性中的OI属性。如果找不到OI属性或者解析OI属性失败，则直接跳过；否则取出OI属性保存的对象标识，与由BlueStore直接通过查询数据库解码得到的对象标识进行比对，如果不一致，则认为OI属性保存的对象标识出错，并同步执行修复。

本地完成修复之后，每个副本可以将最终扫描结果返回给Primary，由其进行汇总和继续执行分布式一致性校验。

### 8.2.4 副本比对

当Primary完成所有副本扫描结果收集之后，可以通过副本间的信息比对，来执行副本间的数据一致性校验与修复。由于每个副本所包含的对象视图有可能不一致，因此，Primary首先需要基于所有副本返回的实时扫描结果，建立一个囊括所有对象的集合，称为MasterSet。

与Peering需要选举出一个权威日志类似，针对MasterSet中的每个对象，也需要由Primary选举出一个权威副本。选举规则如下：

·对应的副本没有stat错误、静默数据错误（如果读取对象内容时BlueStore返回EIO，说明有静默数据错误）、OI属性存在并且正确、对象大小正确等；进一步地，如果是原始对象，还要求其SS属性存在。

·如果同时存在满足上一条件的多个副本，则优先选择存在更多校验消息（数据校验和与omap校验和）的那个副本。

·如果同时存在满足上面两个条件的多个副本，则优先选择Primary对应的副本（否则随机选择）。

·反之，如果无法选出满足上述条件的权威副本，那么Scrub将记录对应对象的权威副本缺失，并更进一步地收集和记录错误发生时一些详细信息。

在选举每个对象的权威副本过程中，同时会执行副本之间的OI/SS属性一致性校验。其判断标准为：以第一个遇到的、本地校验通过的OI/SS属性作为基准，将后续其他副本的OI/SS属性与之进行对比，如果不一致，则认为后者的OI/SS属性出错。当然，由于缺乏理论支撑，这也远远不是一种什么完美的一致性校验方案。

在成功地选出对象的权威副本后，可以执行副本之间的一致性校验。其主要逻辑是，将每个副本依次与权威副本执行如下比较：

·如果副本没有这个对象，则将其加入missing列表，否则继续执行后续比较。

·数据校验和、omap校验和是否一致。如前所述，对象一共有两种类型的校验和：深度扫描过程中实时计算得到的校验和与客户端更新对象时由PG负责生成并同步写入OI属性中的校验和。为方便起见，我们将这两种校验和分别称为实时校验和与历史校验和。因此，这个比较又可以细分为两种情况，一是将副本的实时校验和与权威副本的实时校验和相比；二是如果副本的历史校验和存在，也将其与权威副本的实时校验和进行比较。

·对象大小是否一致。

·扩展属性是否一致。考虑到通常情况下扩展属性不多，可以直接将副本的所有扩展属性与权威副本逐一进行比对。如果副本有多余属性、缺少某些属性或者任一属性的内容不相等，都认为副本存在扩展属性错误。

·经过上述比较后，如果副本不存在任何与权威副本不一致的情况，则将其加入权威副本列表（authoritative）；否则将其加入不一致副本列表（inconsistent）。

·进一步地，完成对象的所有副本比对后，如果该对象没有缺失或者不一致的副本，并且OI属性没有保存校验和或者保存的校验和已经老化，那么可以趁此机会一并更新校验和。

依次完成MasterSet中所有对象的比对后，最终将产生两张错误对象列表，missing和inconsistent，它们分别记录了所有缺失和不一致的对象副本以及这些副本所在的位置；还有一张权威副本列表，它完整而正确地记录了PG应该存储的对象（以及当前那些权威副本所在的位置）。

基于权威副本可以重新执行PG粒度的统计，例如，本次扫描包含了多少个原始对象、多少个克隆对象、多少个omap对象，这些对象共计使用了多少逻辑空间等。同时也可以更进一步地进行克隆对象相关的校验，例如，原始对象的克隆对象数量是否正确、克隆对象排列是否乱序、克隆对象相关的统计等。当然，这些统计数据都是不完整的，需要随着Scrub进度不断更新，直至Scrub最终完成，才能与PG自身记录在Info中的历史统计信息进行对比与修正。

最后，如果产生了数据不一致（例如missing或者inconsistent不为空），并且使能了自动修复功能，那么可以利用权威副本列表对出错的对象执行修复。修复过程本质上是复用Recovery机制，我们已经进行过详细介绍，这里不赘述。

### 8.2.5 统计更新与自动修复

当PG中所有对象都完成了Scrub之后，可以将本次Scrub最终得到的统计数据与PG自身保存的历史数据进行比对。如果不一致，则以本次Scrub的结果为准，同步进行更新。同时，如果Scrub过程中捕获到任何错误，也需要通过本地对象存储进行固化，供管理员进行查询和回溯。

需要特别注意的是，由于Chunky-Scrub只阻塞指定范围内的对象被客户端进行改写，所以整个Scrub的生命周期内，如果客户端针对某些不在Chunky-Scrub范围内的对象执行了写操作（也包括创建和删除对象），那么其对应的统计，除了要正常地在PG的Info之中进行更新之外，也要在Scrub的全局统计结果中同步进行更新，否则必然会导致Scrub最终得出错误的统计结果。

## 8.3 Scrub抢占

至此，我们已经系统地介绍了Scrub的指导思想及实现细节，本节我们继续探讨Scrub的后续改进方向。

如果针对Ceph常见问题排个榜单，那么“臭名昭著”的Slow Request一定位列前茅。和访问降级对象类似，假定客户端正好需要访问某些位于Scrub范围中的对象，那么经验告诉我们此时很容易出现相关请求被挂起，最终因为长时间得不到调度而触发系统产生大量Slow Request告警的现象（Scrub只阻塞写请求）。

通过引入Chunky-Scrub、流控以及严格控制允许执行Scrub的时间段（或者控制Scrub的调度间隔）等方式，我们显著降低了Scrub阻塞其他业务的概率。然而，由于Scrub本身肩负着校验数据一致性的重任，我们不可能在生产环境中完全关闭Scrub，因此实际上Scrub与正常业务之间的冲突不可避免。当然，考虑到我们的最终目标并不是要彻底消除这种冲突，而是要避免产生Slow Request告警，因此需要先分析Scrub导致Slow Request告警的原因，才好对症下药。为此，我们先设定一些前置条件，然后举例进行说明。

·假定每个磁盘（对应一个OSD）的读/写带宽典型值为100Mbps，并且在混合读写模式下，这个带宽将下降至一半，为50Mbps。

·假定每次Chunky-Scrub扫描的对象数量固定为25个，并且全部为原始对象。

·假定每个对象存储的数据部分固定为4MB。

·假定某次Chunky-Scrub开始时，客户端正好要修改位于Scrub范围中的某个对象。

·假定当Scrub和客户端业务同时进行时，两者能够获得磁盘带宽比为2：8（因为来自客户端的I/O优先级较高）。

通过简单的计算，我们可以得到最差情况下某个客户端的I/O可能被阻塞的时间约为

25×4MB/(50Mbps×0.2)=10s

这个结果第一眼看上去或许不算太坏，然而由于客户端访问模式的不确定性，实际上大部分情况下磁盘在两种业务之间切换时所能达到的最大带宽会远低于50Mbps。如果更进一步地考虑系统调度开销以及诸如RBD、RGW这类应用还会向一些特殊对象中写入大量omap属性等种种复杂因素，则在一些极端情况下，客户端I/O被阻塞的时间只会更长，因此触发Slow Request告警也就不足为奇了。

为了解决上述问题，直觉上的办法或许是进一步缩小Chunky-Scrub窗口。例如，极端情况下可以将这个窗口缩小至[1，1]，也就是说我们每次只允许一个对象执行Scrub。遗憾的是，触发Slow Request的概率理论上依然存在（例如，想象我们正在Scrub一个包含了多达1GB omap属性对的对象），并且这种做法在大部分情况下也无法满足前面提及的、需要将一个对象及其产生的克隆对象尽量安排在同一个分片中执行Scrub的约束。因此，盲目缩小Scrub窗口并非一种理想的方案。

前面我们已经强调过，Scrub扮演的是事后审计的角色，也就是说其优先级要天然低于其他业务。因此，更合理的方案是在Scrub与其他业务（特别是客户端业务）产生冲突时，由Scrub进行退避。换言之，这要求正在执行中的Scrub任务能够随时被更高优先级的任务所打断和抢占，同时为了Scrub能够正常完成，还必须要求Scrub具有幂等性，也就是某个分片中的对象可以重复多次执行Scrub而不会对最终结果造成影响。幸运的是，Scrub天然具有幂等性，因此这里的关键在于执行抢占的时机和粒度。

首先，我们注意到，在单次Chunky-Scrub的执行过程中，每个环节的时间消耗非常悬殊，例如统计表明实际上99%的时间都消耗在了对象扫描阶段。如果在完成对象扫描之后再执行抢占，那么将导致本次Chunky-Scrub前功尽弃，这显然不是什么高性价比的做法。因此合理的抢占窗口应该是对象扫描之间。

其次，我们来考虑执行抢占的粒度。显然，如果对象扫描本身是个原子操作，执行抢占也就无从着手。因此，需要将对象扫描这个环节再次分解，拆成足够小的粒度，方便当检测到客户端业务与Scrub冲突时随时进行抢占。那么，多小的粒度是合理的呢？直觉上也许会认为是单个对象，但是考虑到单个对象也有可能保存大量数据或者omap条目（实际上在业务繁忙或者磁盘碎片化比较严重时，即便读取4MB数据也可能要花费很长时间），因此更合理的粒度是每次调度只扫描一定量的数据（例如512kB）或者omap条目（例如32个）。

最后，需要注意的是，PG需要记忆每次执行Scrub的起点。Scrub过程中如果发生了抢占，则Scrub再次被调度时将从起点重新开始执行。当然为了防止Scrub被无限抢占，也可以针对每次Scrub能够被抢占的次数进行限制，或者更进一步地，每次抢占发生时，逐渐缩小Chunky-Scrub扫描的对象范围以降低冲突概率。当然，后面这些都是一些锦上添花的做法，这里不再展开描述。

## 8.4 总结和展望

Scrub是一种重要的辅助机制，用于守护集群数据的正确性与一致性。

实现上，Scrub主要依赖对象的有序性与信息摘要技术，前者使其可以不重复（从而高效）地遍历集群中的所有对象，后者则提供了一种快速检测数据正确性和一致性的手段。

Scrub的局限性首先在于其并非一种实时监测机制，而是一种事后审计（抽查）机制，因此即便Scrub结果正常，也并不能说明集群数据完全正确和一致。其次，分布式一致性本身是软件工程中公认的难题，强分布式一致性（例如Paxos）容易导致系统的性能达不到生产环境要求，弱分布式一致性则意味着数据本身的可靠性得不到有效保障。因此，对分布式存储系统而言，如果想要兼顾性能和数据可靠性，往往意味着其一致性要求也介于两者之间。Scrub除了负责校验数据的正确性之外（实际上这项任务在Luminous版本已经转交给BlueStore实施），也是保障数据一致性的一种重要补充手段。然而，正如前面提到的，完备的分布式一致性校验机制必然要求其背后有强分布式一致性理论作为支撑，这反过来又会影响Scrub（和集群）的性能，因此在不过分损伤集群性能的前提下，实际上Scrub针对数据一致性能做的也比较有限。

此外，由于Scrub本身是磁盘（读）带宽密集型任务，在生产环境中完全放开Scrub相关限制的障碍在于，其会与客户端发起的正常业务产生严重冲突，影响集群对外性能。这固然可以通过对Scrub施加诸如执行时段、执行频率限制，甚至完全关闭Scrub等手段加以缓解或者解决，然而由于Scrub目前的重要性（例如Scrub可以有条件地自动修复某些受损的对象副本），真正的解决方案还得依赖于更加完备的冲突处理与流控机制。

第9章

# 基于dmClock的分布式流控策略

伴随着云计算、虚拟化技术的逐渐兴起，为了避免资源浪费，最大化资源利用率，传统计算机及其相关硬件资源，例如CPU、内存、网络以及存储介质等都被抽象为虚拟资源，纳入资源池进行池化管理，并以此为基础统一以虚拟机的形式对外提供可定制的计算和存储服务。虚拟机的出现对后端存储系统提出了更高的要求，存储系统需要在保证性能、可靠性等不受影响的前提下，对有限的存储空间和I/O资源更加合理地进行分配，以应对灵活多变的虚拟机应用场景。Ceph作为私有云事实上的标准——OpenStack的默认存储后端，同样存在I/O资源分配的问题。比如，生产环境中可能出现由于个别虚拟机侵占了整个集群的绝大部分I/O资源，导致其他虚拟机操作时延较大、用户体验不友好等问题；某些应用场景下需要为一些重要客户预留更多的I/O资源，以便提供更加优质的服务等。基于上述因素，Ceph社区正在积极地引入QoS（Quality of Service，服务质量）特性。QoS最早起源于网络通信，指一个通信网络能够利用各种基础技术，为不同类型的应用提供不同等级服务质量的能力。对存储系统而言，引入QoS旨在帮助系统更加合理地统筹有限的I/O资源，实现按需分配，对外提供更好的存储服务。

dmClock是一种用于分布式系统的I/O调度算法，由VMware发表的一篇学术论文中首次提出。它起源于mClock算法。dmClock是目前I/O调度方面公认比较优秀的算法，Ceph计划采用它来优化内部I/O调度策略并实现QoS特性。

本章分为以下几个部分：首先我们介绍社区对QoS特性的研究概况，mClock与dmClock算法的基本原理以及它们之间的差异；其次，我们结合实例来说明dmClock如何实现分布式系统的I/O资源调度；然后，我们基于自身实践介绍针对dmClock进行改进以及如何使其更好地服务于Ceph的思路；最后，总结dmClock当前存在的问题及后续的重点关注方向。

## 9.1 概述

如前所述，作为一种分布式存储系统，Ceph与传统存储系统存在诸多不同，其中最重要的一点是，传统存储系统一般而言存在集中控制点，而Ceph为了最大限度地追求可扩展性和高并发，允许客户端通过OSDMap直接与集群中的OSD通信。Ceph这种去中心化的架构设计使其QoS实现机制与传统存储系统存在显著区别：传统存储QoS实现位置首选中心节点，引入集中的QoS控制模块（虽然这样做更为简单）显然与Ceph去中心化的设计理念相悖，因此结合Ceph自身的特点，较为合理的做法是将QoS机制直接嵌入每个OSD中来实现。

早在Ceph刚被设计出来不久，Sage及其团队成员就在考虑QoS的支持问题，并于2007年提出了一个称为Bourbon框架的QoS解决方案。该框架在EBOFS（Extent and B-tree based Object FileSystem，基于Extent和B树的对象文件系统，也称为btrfs，是Ceph最初采用的本地文件系统）的基础上设计出一个可以感知QoS特性的Q-EBOFS系统。该方案的基本原理是，在btrfs原有的磁盘调度队列中，引入了一个客户端级别的新队列，采用基于权重的轮询机制，将不同客户端队列中的请求分发至磁盘调度队列，从而在单个OSD层面实现对不同客户端按其权重分配磁盘I/O资源的目标。然而，这种方式存在以下局限性，这可能也是其最终并未被Ceph采用的原因。

1）Ceph的组件采用了灵活的插件模式，例如存在多种本地对象存储引擎（FileStore、BlueStore等），btrfs只是传统FileStore所依赖的本地文件系统中的一种选择，而类似BlueStore这种新型引擎，甚至根本不再需要本地文件系统。因此，这种将QoS实现内置于本地文件系统的方案不具备通用性。

2）该方案主要用于不同权重客户端之间的I/O分配，无法解决客户端I/O资源预留以及最大I/O资源限制等问题。

随着Ceph开源社区的稳步发展，许多新的特性被逐步引入和完善，整个系统因而变得越来越复杂。所以，通过QoS功能保障集群的存储服务质量则显得越来越重要，特别是Ceph块设备接口被广泛应用于云计算领域之后，对块存储的QoS控制需求变得愈发紧迫，因此近几年社区正在计划基于dmClock算法开发QoS功能。

## 9.2 dmClock基本原理

dmClock算法是一种分布式的mClock算法。为了更好地理解dmClock，我们首先介绍mClock的基本概念和原理。

### 9.2.1 mClock

mClock是一种基于时间标签的I/O调度算法，适用于C/S模式的存储系统。这里的时间标签是指：客户端的每个I/O请求都会被打上一个时间戳，服务器依据时间戳的大小进行I/O调度，决定I/O请求被处理的时机与顺序。mClock支持I/O的预留（reservation）、权重（weight）和上限（limit）3个维度，通常也被称为QoS模板。其中，预留是指在服务器的处理能力之内，为客户端提供的最低I/O处理速率；权重是指多个客户端同时进行I/O访问时，服务器为每个客户端分配相应I/O资源的比重；上限是指服务器为某个客户端提供的最高I/O处理速率。

从逻辑结构上，mClock分为client和server两个部分。通常情况下，这两部分分别驻留在上述提及的客户端和服务器当中。其中，client负责下发客户端的QoS模板参数，并统计I/O请求的完成个数等信息；server是服务器中I/O调度的核心部分，主要为客户端的I/O请求计算时间标签，并基于此对服务器的I/O资源进行调度。一个典型的mClock应用模型如图9-1所示。

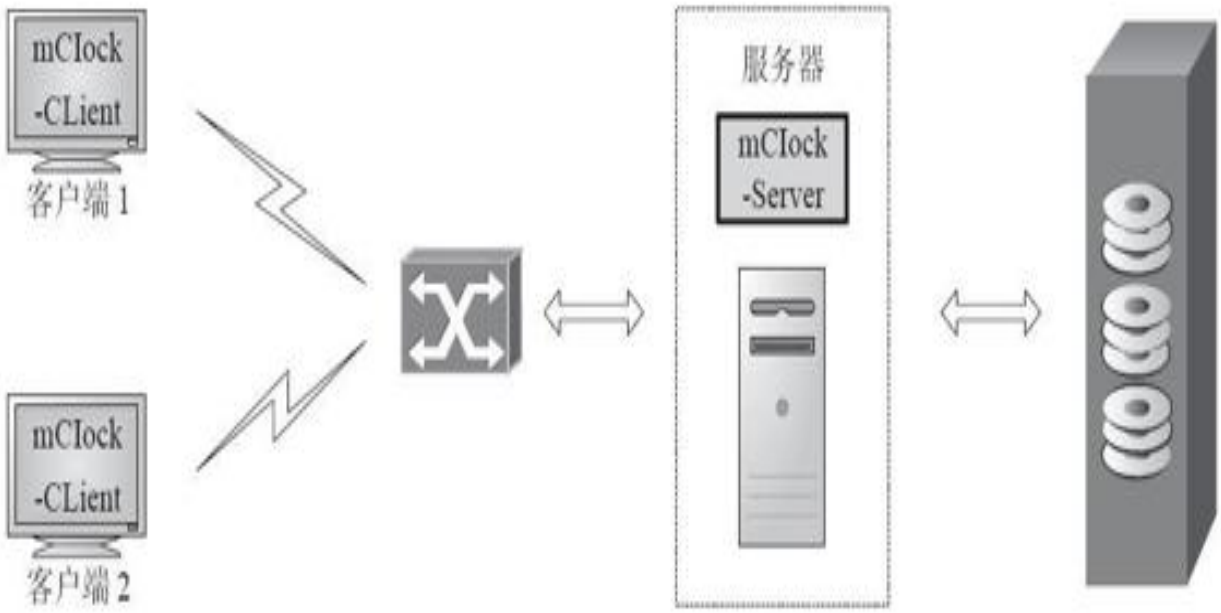



图9-1　mClock典型应用模型

mClock基本原理主要包含以下两个方面。

1）为客户端设置一套QoS模板，并在每个I/O请求中携带该QoS模板。

2）服务器计算每个I/O请求的时间标签，并分成两个不同的阶段处理I/O请求：首先进入基于预留时间标签的Constraint-Based阶段，处理所有预留时间标签满足条件的I/O请求；然后进入基于上限和权重时间标签的Weight-Based阶段，处理上限时间标签满足条件的I/O请求，如果有多个客户端同时满足条件，则依据权重时间标签的大小决定处理顺序，即权重时间标签较小的客户端I/O请求将被优先处理。

算法详解如下：

如果用[$r_i$，$w_i$，$l_i$]表示client i的QoS模板，用表示来自client i的第r个I/O请求的时间标签，则mClock算法伪代码可以表示如下：

```
Request Arrival(request r, time t, client ci)
begin
  if c_i was idle then
    /* Tag Adjustment */
    minWtag = minimum of all W tags;
    foreach active client c_j do
      W_j^r -= minWtag - t;
  done
  /* Tag Assignment */
```

```
    R_i^r = max {R_j^(r-1) + 1/r_i, t }   /* Reservation tag */
    L_i^r = max {L_j^(r-1) + 1/l_i, t }   /* Limit tag */
    W_i^r = max {W_j^(r-1) + 1/w_i, t }   /* Weight tag */
end
Schedule Request(time t)
begin
    Let E be the set of requests with R tag ≤ t
    if E not empty then
        /* Constraint-Based scheduling */
        Select I/O request with minimum R tag from E
    else
        /* Weight-Based scheduling */
        Let E' be the set of requests with L tag ≤ t
        if E' not empty then
            Select I/O request with minimum W tag from E'
            /* Assuming request belong to client c_k */
            Subtract 1/r_k from R tags of client c_k
end
```

通常情况下，以第1个I/O请求的到达时间作为初始基准时间标签，后续I/O请求的时间标签依据预设的QoS模板，对单位时间进行均匀切分（以1/r、1/w、1/l为步长）后计算而来。以某个client设置的QoS预留是100为例，我们期望服务器在1秒内至少处理来自该client的100个请

求，即每个请求在前一个请求的0.01s之后得到处理。如果某个I/O请求由步长计算的时间标签值比当前时间更小，则表明距服务器收到上一个请求已经过去了较长时间，这通常是由于客户端经过一段时间的空闲后再次发送请求而导致，此时应该重置该请求的时间标签为当前时间，并以此作为后续I/O请求的基准。

权重时间标签则是一种特殊的标签，只有当多个client同时存在时它才有意义。将一个client的权重值设置为100并不意味着服务器每秒需处理100个I/O请求，而是该client与其他client竞争时的依据（权重时间标签较小者被优先处理），所以权重时间标签是一个相对时间值，并不要求与真实时间一定有关联。但为了使多个client公平竞争，我们取首个I/O请求到达的真实时间作为基准时间标签，并在client数量或状态变化时对时间标签进行校准。当有新的client加入（或者client从空闲状态唤醒）时，其权重的基准时间标签与之前的client不同，而且之前的client运行过一段时间后，权重时间标签产生的漂移（与真实时间之差）通常会很大，导致新旧两类client在参与权重竞争时不在同一起跑线上，从而出现其中一类client一直竞争胜出，另一类却长时间得不到处理（如果未设置预留）的现象，并将一直持续到前者的时间标签值追上后者为止。为了解决这个问题，需要调整新旧client的权重时间标签，比如以新client的权重时间标签为准，将旧的client的权重时间标签都添加一个补偿值（例如上述伪代码中的：t-minWtag），反之亦可。

通过前面的介绍我们知道，mClock工作于动态轮转的两个阶段，并总是期望I/O请求尽量在Constraint-Based阶段被处理，以减少client之间的竞争。当某个client的一个I/O请求在Weight-Based阶段被处理后，其后的其他I/O请求的预留时间标签都需要进行调整，即减去一个时间标签步长1/r值，以填补在Weight-Based阶段被处理掉的I/O请求所带来的空缺，保持该client预留时间标签的正确性。相应的调整过程如图9-2所示。

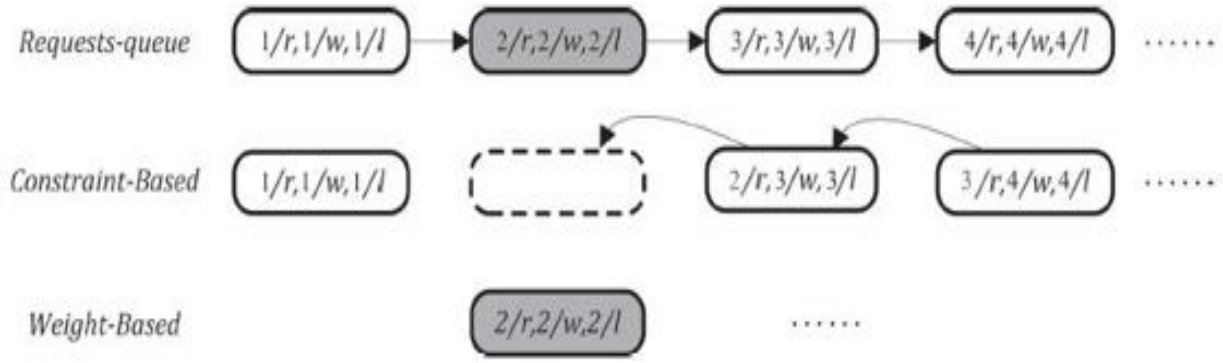



图9-2　预留时间标签值调整过程

当第2个请求在Weight-Based阶段被处理后，如果不对剩余的I/O请求进行预留时间标签进行调整，则第3个请求的原始预留时间标签（实际为$t_0$+3/r，其中$t_0$为基准时间标签，为了描述方便而简化为3/r）将更加难以满足被处理的条件（即，预留时间标签小于或等于当前时间），从而导致该client的I/O请求一直在Weight-Based阶段被处理，使得预留达不到预期效果。

### 9.2.2 dmClock

dmClock在mClock算法的基础上，针对分布式系统的特点做了相应修改。其在分布式系统的每个服务器上运行一个改进的mClock-server，在客户端驻留改进的mClock-client以统计各个服务器完成的I/O请求个数，并据此调整服务器处理I/O请求的速率，以期在总体上达到客户端的QoS限制目标。

与mClock的差别主要体现在以下几个方面：

1）分布式系统的各个服务器在回应I/O请求时，返回其是在哪个阶段（phase）被处理完成的（即Constraint-Based阶段或Weight-Based阶段）。

2）客户端统计各个服务器完成的I/O请求个数，在向某个服务器下发I/O请求时，携带自上次下发I/O请求之后，除目标服务器以外的其他服务器完成的I/O请求个数之和，即两次I/O下发之间的I/O请求完成数的增量，分别使用ρ（rho）和δ（delta）表示以上两个阶段的增量。

3）服务器计算I/O请求的时间标签时，不再根据步长均匀递增，而是使用ρ和δ作为调整因子，从而动态调整各个服务器处理该客户端I/O的频度，使得总体处理速率满足QoS设置的约束条件。

每个I/O请求的时间标签计算公式如下：

$$R_i^r = \max\left\{R_i^r \quad + \left( \quad {}_i^r + 1\right)/r_i,\ t\right\}$$

$$W_i^r = \max\left\{W_i^r \quad + \left( \quad {}_i^r + 1\right)/w_i,\ t\right\}$$

$$L_i^r = \max\left\{L_i^r \quad + \left( \quad {}_i^r + 1\right)/l_i,\ t\right\}$$

同样以之前的例子加以说明，假设某个client的预留值设置为100，其I/O请求由系统的多个服务器共同承担，各个服务器的I/O处理能力可能存在差异，通过ρ和δ计算I/O请求的时间标签值，以调整各个服务器处理I/O请求的个数，达到整体对外提供每秒处理100个I/O请求的效果。

## 9.3 dmClock算法实现

目前，dmClock算法的代码开发已经完成，并且合入了Ceph主干分支。从实现上来说，dmClock也可以分为client和server两个部分。其中，client部分主要提供两个接口，一个用于更新每个服务器完成的I/O请求个数，另一个用于获取上述统计数据。server部分是整个dmClock算法实现的核心部分，主要由一个两级映射队列组成，第1级为客户端组成的client队列，第2级为client对应的I/O请求子队列。每个I/O请求包含3个时间标签$<R_i$，$W_i$，$L_i>$，其中i表示其归属的client编号。server的队列结构如图9-3所示。

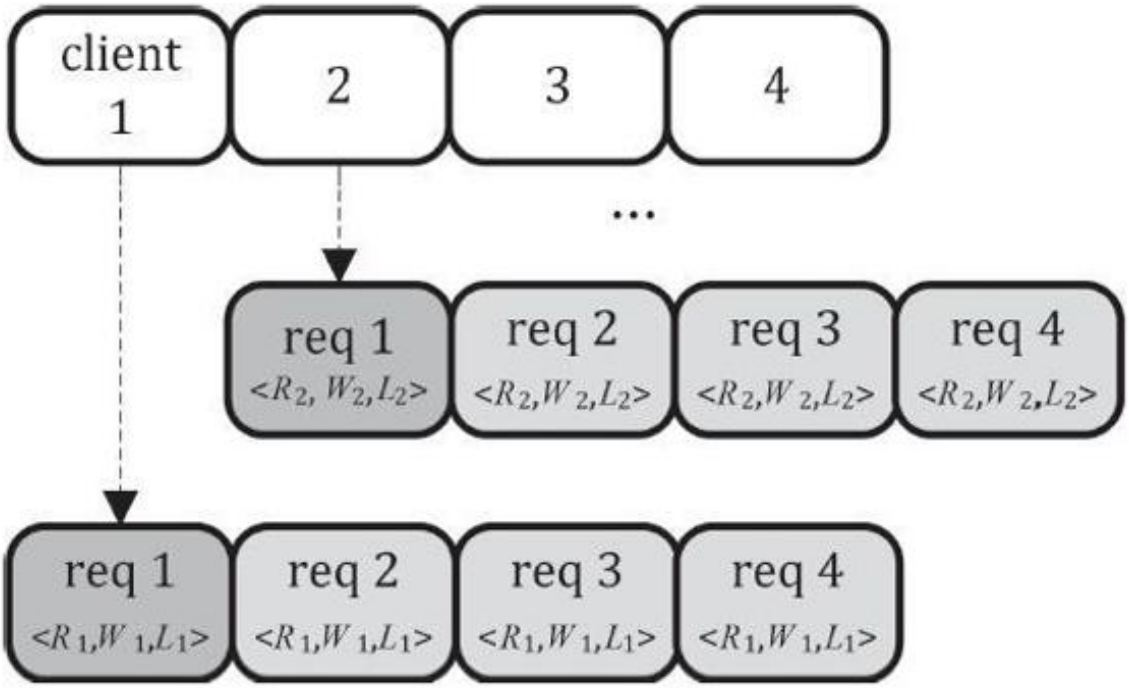



图9-3　dmClock-server队列结构

由于涉及大量动态排序操作，时间标签采用完全二叉树进行组织。在服务器中生成dmClock-server队列的同时，基于QoS的三个维度分别构建预留时间标签、权重时间标签和上限时间标签二叉树（为叙述方便，以下简称为预留二叉树、权重二叉树和上限二叉树）。每颗二叉树的节点为各个client及其对应的I/O请求子队列，节点在树中的位置依据其队首元素（即子队列当前的第一个元素）的时间标签进行确定，基本原则是父节点的时间标签小于子节点。5个节点的完全二叉树结构如图9-4所示。

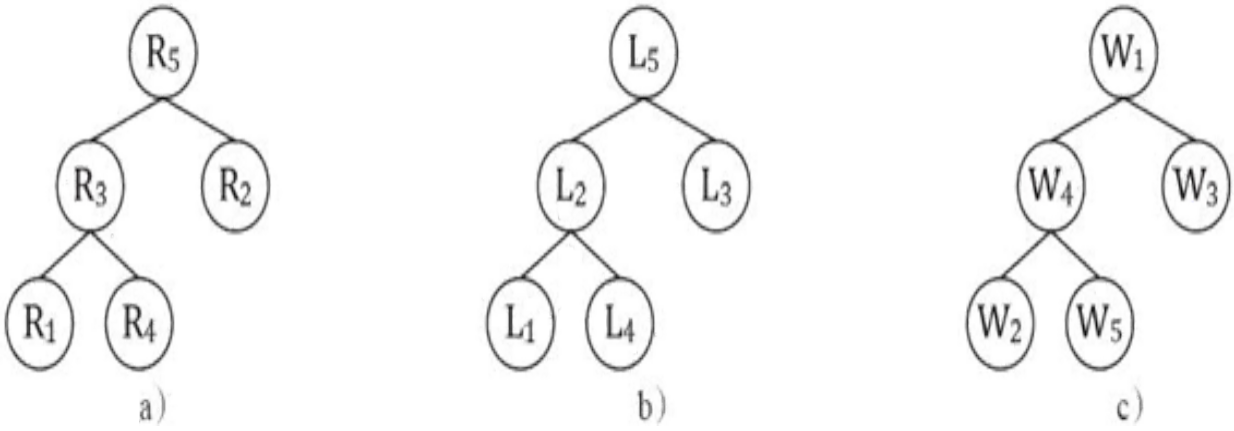



图9-4　时间标签完全二叉树结构

### 9.3.1 I/O请求入队

客户端的I/O请求发送至服务器后，首先进入client对应的子队列排队，同时服务器计算该请求的时间标签值，然后依据时间标签的大小调整其在二叉树中的位置。这个过程称为I/O请求入队。

具体来说，如果某个client的I/O请求子队列中还存在未处理的请求，则新的I/O请求直接挂入子队列尾部即可；如果client的子队列中不存在未处理的请求，或者是一个全新的client的请求（此时需要先创建相应的子队列），则除了将该I/O请求挂入其子队列外，还需要将client加入时间标签二叉树，并依据时间标签的大小将其调整至正确的位置（该I/O请求是client子队列的第一个元素）。各时间标签二叉树的调整规则如下。

·预留二叉树：以节点的队首元素的预留时间标签为依据，值较小的节点尽量调整至树的上层，反之调整至下层，最终根节点的预留时间标签最小。

·上限二叉树：用ready标记来表明每个I/O请求是否满足出队条件（上限时间标签是否小于或等于当前时间），默认为false，满足条件的请求ready标记为true，并将相应的节点往下层调整；反之则往上层调整，标记相同的节点依据时间标签的大小决定位置。

·权重二叉树：同样以ready标记来区分请求，与上限二叉树不同的是，队首元素的ready标记为true的节点往上层调整，ready标记为false的节点往下层调整；标记相同的节点则依据权重标签大小，值较小的置于上层，反之置于下层。

### 9.3.2 I/O请求出队

I/O请求出队是指在client队列中选择适当的client，从其I/O请求队列中出队一个元素的过程。通过前面原理部分的阐述我们知道，这个流程首先进入Constraint-Based阶段，取预留二叉树根节点的I/O请求队列，判断其队首元素的预留时间标签是否满足出队条件（小于或等于当前时间）。如果条件满足，则选取该根节点对应的client，从其请求队列的队首出队一个元素；否则进入Weight-Based阶段，从上限二叉树根节点开始，逐个判断队首元素的上限时间标签是否小于或等于当前时间，并设置满足条件的请求的ready标记为true，以决定其是否可参与随后的权重竞争。所谓权重竞争，是指对所有上限时间标签满足条件的client，依据其队首元素的权重时间标签大小，调整自身在权重标签二叉树中位置的过程，最终位于权重标签二叉树根节点的client竞争胜出。最后，从竞争胜出者的I/O请求队列的队首出队一个请求。I/O请求出队的流程如图9-5所示。

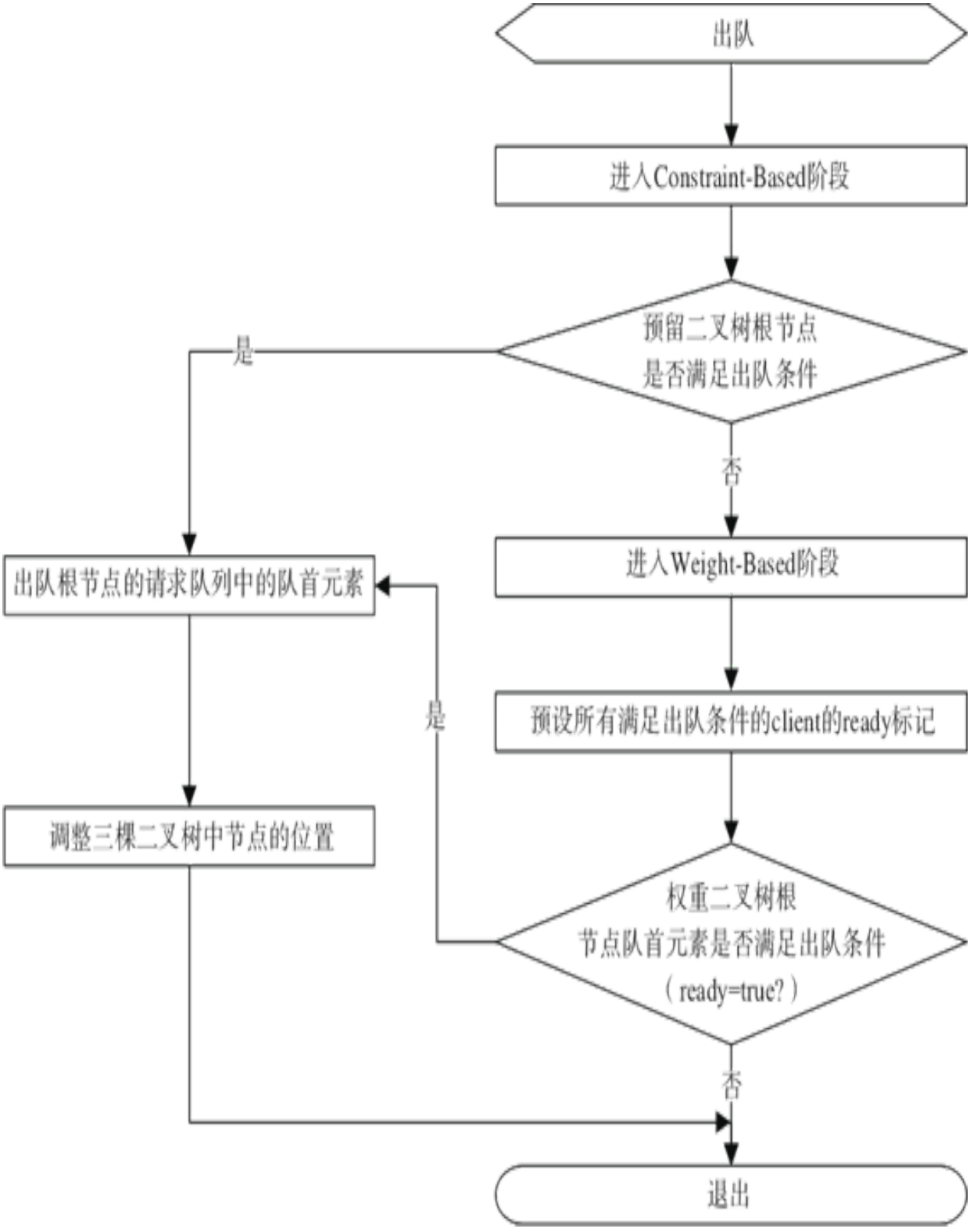



图9-5　I/O请求出队流程

三棵二叉树的节点并不是孤立的，而是指向相同的client及其子队列。任何一个client出队一个I/O请求后，由于队首元素改变而导致的时间标签变化，都可能影响该节点在三颗二叉树中的位置。因此，从根节点出队一个请求后，除了调整其在当前二叉树中的位置之外，还需同步调整对应节点在其他两棵树中的位置。位置调整包括两个方向：一个是上移（sift up），即以冒泡的方式，逐层往上尽量移动至根节点位置；一个是下移（sift down），选择较小的子节点，逐层往下尽量移动至最底层。每个新节点在加入三棵二叉树时，都会先加在树的尾部，再通过上移方式分别调整至恰当的位置。在请求出队的过程中，则通过上移或下移方式调整满足条件节点的位置。因此，节点在三棵树中的位置一直处于动态变化之中。

此外，需要特别注意的一点是，目前dmClock的实现中支持突破上限（allow limit break）模式，能在不满足上限或预留时间标签条件的情况下出队请求，这在某些对QoS上限没有严格要求的场景下，可以让服务器的I/O资源充分得到利用。不过大部分应用场景都有限制上限的要求，因而本书主要阐述关闭了该模式的情况。

## 9.3.3 实例分析

以某个系统中同时存在5个client为例，假定client的基准时间标签Ti都相同（为阐述方便，忽略其时间差），并且每个client的I/O请求队列中存在3个请求。约定[r，w，l]表示客户端的QoS模板，则每个I/O请求的时间标签可以根据公式：$T_i+n\times 1/Q_i$（其中，n=1，2，3）计算，由此得到的时间标签与$T_i$的差值如表9-1所示。

表9-1 client请求时间标签差值

| clients | QoS 模板 | Req 1 | | | Req 2 | | | Req 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | [10, 5, 20] | 0.1 | 0.2 | 0.05 | 0.2 | 0.4 | 0.1 | 0.3 | 0.6 | 0.15 |
| 2 | [20, 1, 60] | 0.05 | 1.0 | 0.017 | 0.1 | 2.0 | 0.034 | 0.15 | 3.0 | 0.051 |
| 3 | [40, 2, 50] | 0.025 | 0.5 | 0.02 | 0.05 | 1 | 0.04 | 0.075 | 1.5 | 0.06 |
| 4 | [30, 4, 40] | 0.033 | 0.25 | 0.025 | 0.066 | 0.5 | 0.05 | 0.099 | 0.75 | 0.075 |
| 5 | [50, 3, 90] | 0.02 | 0.33 | 0.011 | 0.04 | 0.66 | 0.022 | 0.06 | 0.99 | 0.033 |

以预留时间标签为例（$R_i$表示来自client i的首个请求），其二叉树的生成过程如图9-6所示。按照(a1)→(a2)→(a3)→(a4)的顺序，新加入的节点（图中用虚线框表示）都从树的尾部尝试上移，最后时间标签较小的$R_3$调整至根节点；继续加入节点如图(a5)→(a6)→(a7)，时间标签值

最小的新节点$R_5$，加入后经过连续两次上移，最终调整至最顶层根节点位置，从而形成了一棵按预留时间标签值排序的二叉树。权重标签二叉树与上限标签二叉树的生成过程与此类似，最终达到如图9-7中(a7)、(b7)及(c7)所示的状态。

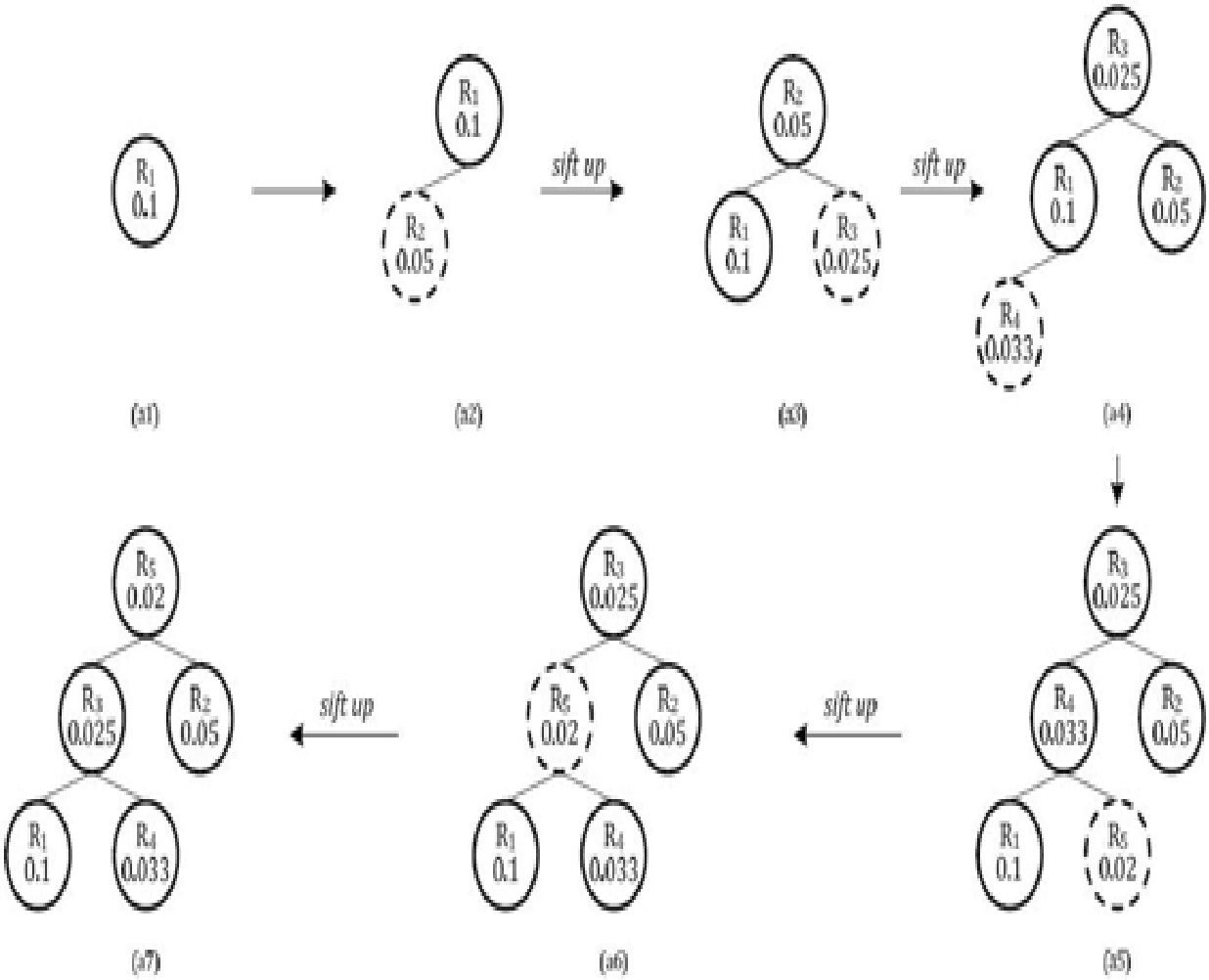



图9-6　预留标签二叉树生成过程

每个client后续的请求直接挂入队列尾部，不会改变标签二叉树的状态，直至有请求出队才触发变化。假设当前时刻（$T_i$+0.021s）需要出队一个请求，请求出队流程首先进入Constraint-Based阶段。用$Q_i^r$的形式表示来自client i的第r个请求的时间标签（其中，Q∈{R，W，L}），如图9-7所示，图(a7)→(a10)表示预留标签二叉树的状态变化，

图(b7)→(b9)以及图(c7)→(c9)分别表示出队请求对上限与权重标签二叉树造成的影响。首先，初始状态图(a7)的根节点满足出队条件（即$T_i+0.02<T_i+0.021$），因而从client 5出队其队首的请求；随后，client 5的第2个请求成为队首元素，相应地三棵二叉树中的时间标签都发生了变化，从而节点的位置可能都需要调整，如图(a8)、(b8)及(c8)中的虚线节点所示。在预留标签二叉树中，随着对应图(a8)→(a9)→(a10)的变化，根节点被调整至最底层，同时client 3成为新的根节点；上限标签二叉树如图(b8)→(b9)，$L_5^2$从根节结点下移至适当的位置；对于权重标签二叉树，相应节点的权重标签较大，不满足调整的条件，则继续保持在原位无须调整。

假设当前时间为$T_i+0.022s$，此时需再出队一个请求进行处理，由于预留标签二叉树的根节点不满足出队$T_i$条件（$R_3^1=T_i+0.025s$），则进入Weight-Based阶段。首先，根据上限标签二叉树查找出所有满足出队条件的client，将其队首元素的ready标记置为true，并将节点尽量下移，如图9-8中(b9)→(b12)左斜线节点所示；然后在权重标签二叉树中将其上移调整，如图(c9)→(c12)右斜线节点所示，调整完成后得到可优先出队请求的client 3。

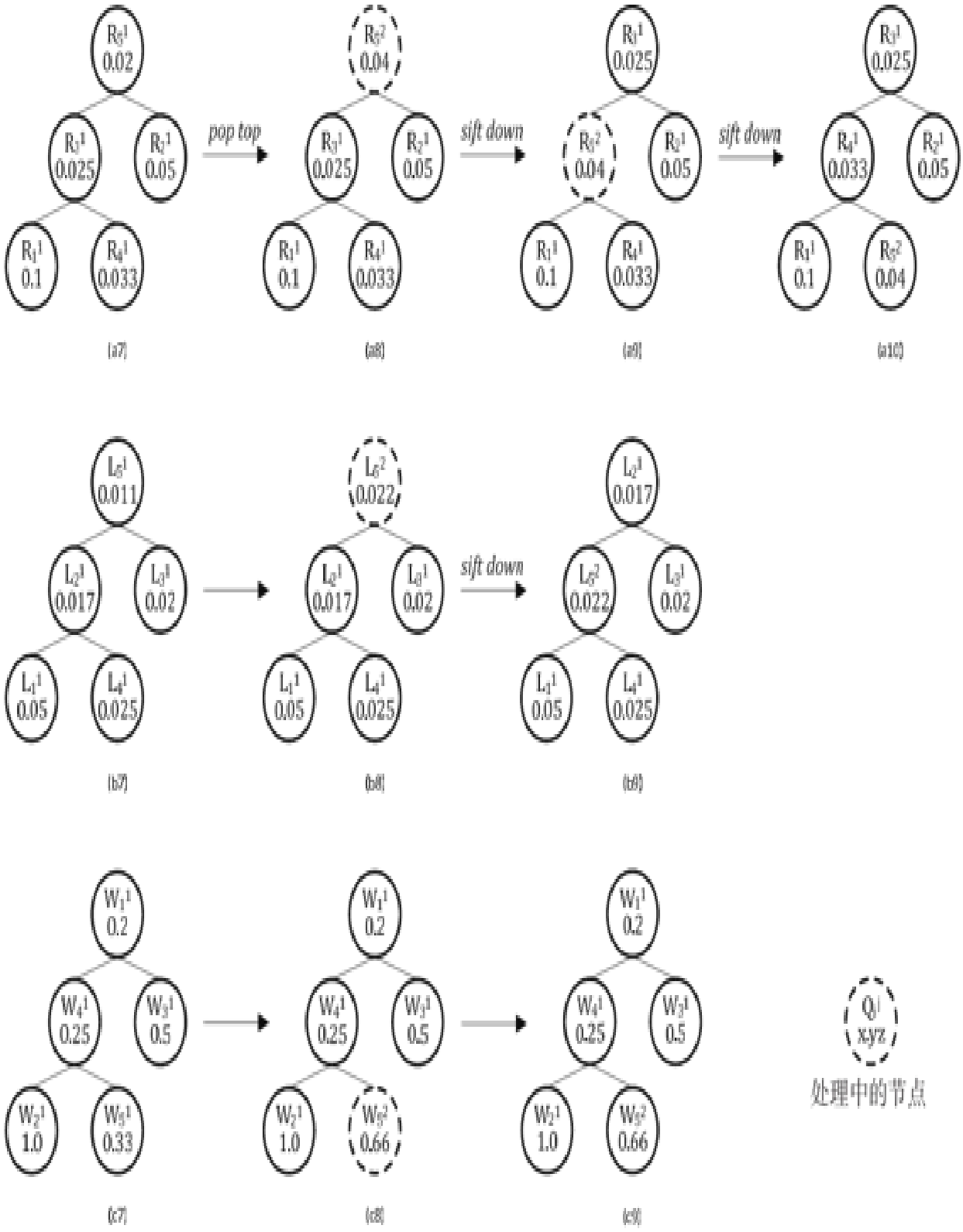



图9-7　Constraint-Based阶段请求出队过程

如图9-8(b9)→(b10)→(b11)→(b12)的查找过程，每次从根节点进行判断，满足条件则将其队首请求的ready标记置为true，并往下调整不再参与判断。根节点的下移调整，促使时间标签次小的节点成为新的根节点，逐个判断直至所有client都不满足条件为止。每个在上限标签二叉树中满足条件的节点，对应在权重二叉树中的节点的位置都需同步调整，如图9-8(c9)→(c10)→(c11)→(c12)所示，最终所有被标记的节点都上移调整至上层。

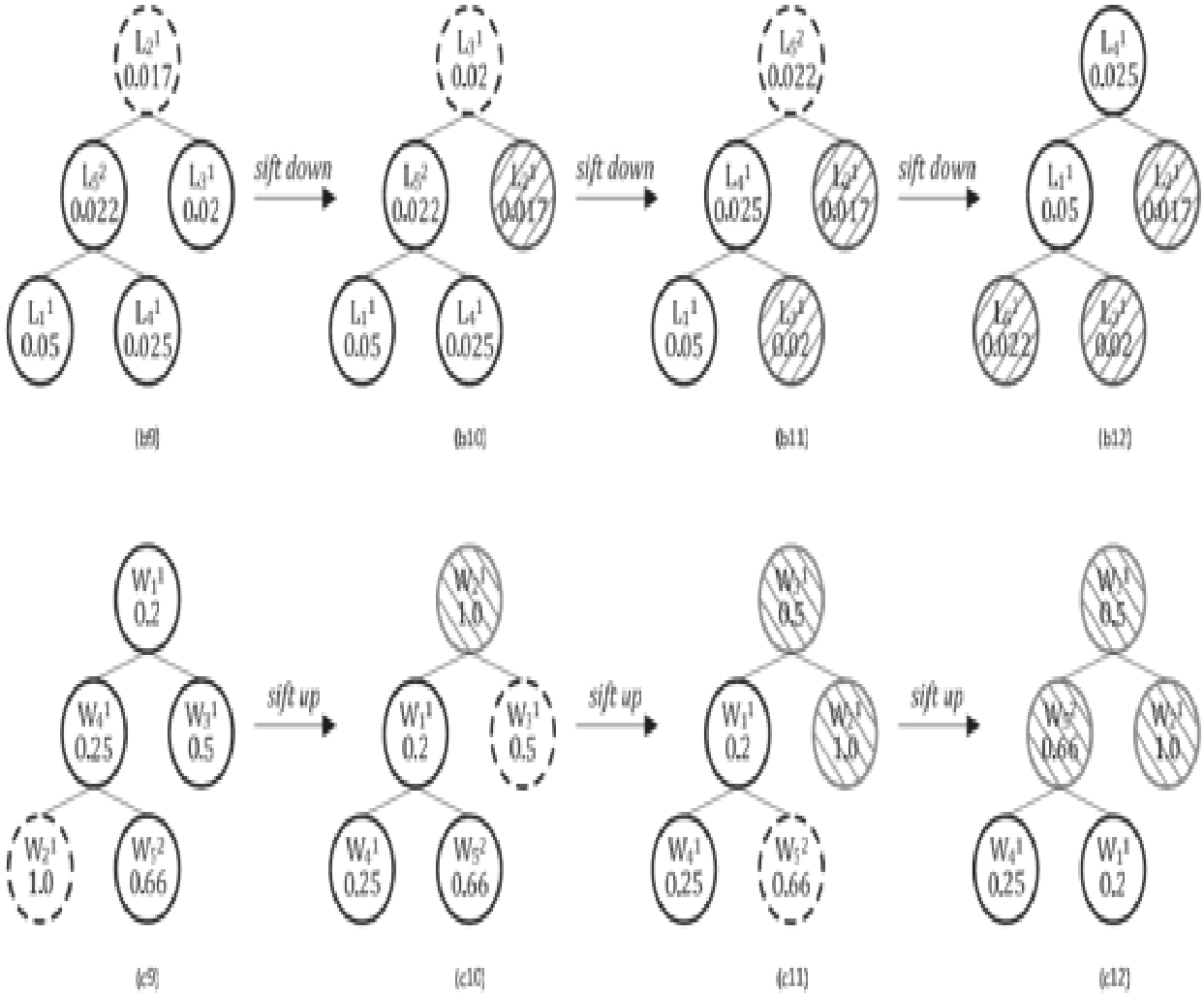



图9-8　权重标签二叉树调整过程

经过上述调整后，便可从权重标签二叉树根节点中出队元素，此处为client 3的第1个请求。整个出队过程及二叉树的状态变化如图9-9所

示。client 3的队首请求出队后，第2个请求便成为新的队首元素，但其ready为默认的false，不满足出队条件，因而需对该节点进行下移调整，如图9-9(c12)→(c13)→(c14)所示。同样，其对应在上限标签二叉树中的节点需要上移调整，以便参与下一次的出队条件判断，如图(b13)→(b14)所示。在Weight-Based阶段出队一个请求，也需要同步调整节点在预留标签二叉树中的位置，如图9-9(a10)→(a11)→(a12)所示，$R_3^2$被调整至最底层，而且这个过程会使得client 3的请求更难以满足Constraint-Based阶段的出队条件，影响预留效果（如之前原理部分所述）。为消除在Weight-Based阶段出队该请求所带来的影响，需要将client 3后续请求的预留时间标签都减去（reduce）一个时间步长$1/R_3$，然后将节点调整至适当的位置，如图9-9(a12)→(a13)所示。注意，此时(a13)中client 3的预留时间标签依然是0.025（即$R_3^2$），与调整之前(a10)中的预留时间标签$R_3^1$相等，相当于没有影响预留时间标签，这便是此次reduce操作的需要达到的目标。

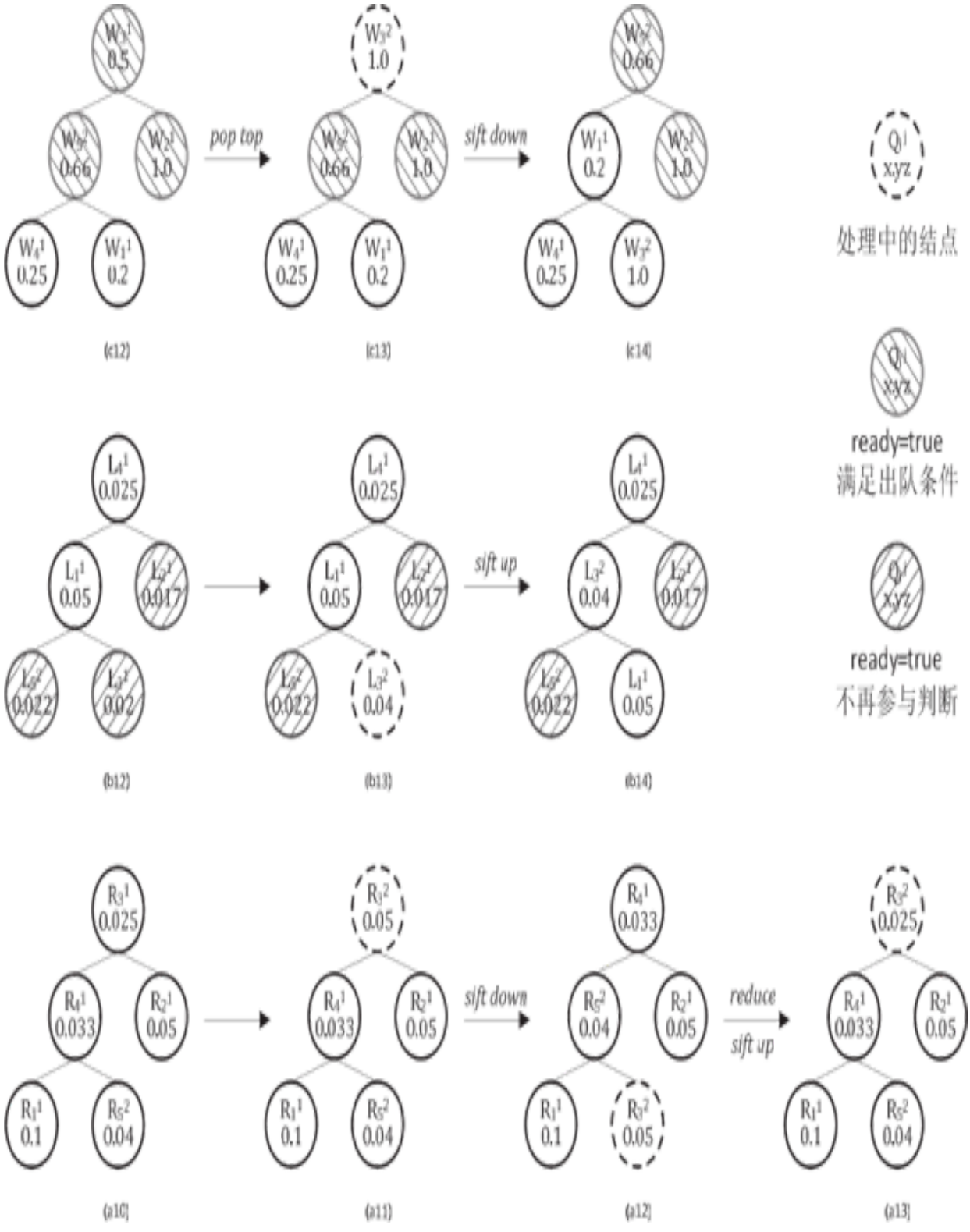



图9-9　Weight-Based阶段请求出队过程

## 9.4 在Ceph中的应用实践

### 9.4.1 client的界定

dmClock队列由多个client子队列组成，其I/O调度过程其实是一个如何选出正确的client的过程。对dmClock算法而言，client只是一个抽象概念，表示一个I/O资源可控的逻辑对象。具体到实际应用，其可以对应不同的独立实体，例如可以将虚拟机当作一个client。当dmClock应用到Ceph中时，我们希望它可以做得更多，主要体现在对client的不同界定上，这通常可以使用I/O类型进行区分。为此我们先介绍OSD内部的I/O类型。

参考PG相关的章节可知，OSD通过op_shardedwq队列统一处理来自客户端的I/O请求与系统内部产生的I/O操作（类型详见表9-2）。为了提高I/O处理并发度，op_shardedwq分为多个真实的子队列，每个子队列由多个线程处理，队列的类型、个数以及线程数都可以通过配置项进行调整。OSD支持多种不同类型的队列，主要包括优先级队列（prio）和基于权重的优先级队列（wpq）两种。Jewel版本之前默认使用prio队列，从Kraken版本以后默认使用wpq队列。无论使用哪一种队列，都要为每类I/O设置一个优先级，如表9-2所示。

表9-2　I/O类型和优先级

| 类型 | 默认优先级 | 说　明 |
| --- | --- | --- |
| clientop | 63 | 来自客户端的I/O请求 |
| subop | 可变 | OSD之间的各类I/O请求 |
| snaptrim | 3 | 在后台执行的快照异步删除操作产生的I/O |
| scrub | 3 | Scrub产生的I/O |
| recovery | 3 | Recovery或者Backfill产生的I/O |

dmClock加入到主干分支后，基于此增加了两个新的队列类型：mclock_opclass队列和mclock_client队列。其中，mclock_opclass将每种I/O类型当作一个client，所有来自集群外部的不同客户端的I/O请求都被认为是来自同一个client；而mclock_client将外部的每个客户端作为一个单独的client，与其他客户端以及集群内部产生的操作共同竞争系统I/O资源。由此可见，Ceph采用dmClock至少有两个目的：一是实现统一的I/O调度，解决外部客户端与集群内部I/O争用的问题；二是解决多客户端之间I/O资源的分配问题。

对分布式系统而言，QoS模板通常要求从客户端随I/O请求下发至服务器，这是由于每个客户端的QoS模板都可能不同，同时客户端还需要下发其他服务器完成的I/O请求数增量（即ρ和δ）等信息。但是对mclock_opclass队列而言，一方面，所有外部客户端都被看作同一个client；另一方面，snaptrim、scrub和recovery等不同类型的I/O都由OSD内部产生，并直接在本地进行出队处理，并不具备分布式特性（这也是该队列依然以mclock_为前缀命名的原因）。因此，启用mclock_opclass队列只需要在服务器（OSD）中为表9-2中每种类型的client分别设置QoS模板即可（如表9-3所示），而不用从客户端下发。

表9-3　client的QoS配置参数

| QoS 模板配置参数 | 默认值[1] | QoS 模板配置参数 | 默认值 |
|---|---|---|---|
| osd_op_queue_mclock_client_op_res | 1000.0 | osd_op_queue_mclock_snap_lim | 0.001[3] |
| osd_op_queue_mclock_client_op_wgt | 500.0 | osd_op_queue_mclock_recov_res | 0.0 |
| osd_op_queue_mclock_client_op_lim | 0.0[2] | osd_op_queue_mclock_recov_wgt | 1.0 |
| osd_op_queue_mclock_osd_subop_res | 1000.0 | osd_op_queue_mclock_recov_lim | 0.001[3] |
| osd_op_queue_mclock_osd_subop_wgt | 500.0 | osd_op_queue_mclock_scrub_res | 0.0 |
| osd_op_queue_mclock_osd_subop_lim | 0.0[2] | osd_op_queue_mclock_scrub_wgt | 1.0 |
| osd_op_queue_mclock_snap_res | 0.0 | osd_op_queue_mclock_scrub_lim | 0.001[3] |
| osd_op_queue_mclock_snap_wgt | 1.0 | | |

①以上默认值并不是最优配置，需要根据实际应用情况调整。

②limit设置为0表示对上限不作限制，加上预留及权重设置较大，表明该OSD绝大部分I/O资源都将用于处理客户端的请求，这是由于所有客户端共享这个QoS模板，可能会有大量I/O请求同时到来。

③之所以能将limit设置为0.001而不影响其正常工作，是由于当前allow limit break默认为true，也就是说在OSD空闲时可以突破limit的限制来处理相应的I/O请求，而不必等1000秒处理一个请求。

由于对外部客户端的界定一直未明确下来，mclock_client队列当前并未实现分布式特性，而是采用了与mclock_opclass队列类似的处理方式（即，在OSD中指定QoS模板）。不同之处在于，mclock_client队列的每个外部客户端都对应一个client，相应地存在一个独立的I/O请求队列。对于外部客户端的界定，社区希望找到一种通用的解决方案，但Ceph的应用场景非常广泛，支持块、文件以及对象多种存储接口，如何选择一个合适的粒度作为client，以灵活地满足各种不同应用的需求，并不是一件容易的事情。根据客户的需求，目前主要存在3种不同

的定义方式：一是以真实的客户端或者虚拟机作为client；二是以存储池为粒度，每个存储池作为一个独立的client；三是以每个块设备的存储卷（对应RBD的image）作为一个client。不过最终方案仍有待商榷。

我们结合实际情况与自身需求，基于社区dmClock的现状做了一些深入研究，成果体现在对dmClock算法实现的完善和改进以及其在Ceph的应用实践两个方面。其中应用实践又包括存储卷粒度的QoS实现和集群级别的流量控制策略，这两者都依赖于dmClock算法能够对带宽进行限制。因此我们首先介绍为dmClock算法增加带宽支持的方法。

### 9.4.2 支持带宽限制

dmClock是一种I/O调度算法，它主要关注I/O个数而非I/O大小。但是I/O大小对于一个I/O请求的影响不容忽视。例如，两个QoS模板相同的客户端同时运行，但客户端A都是1MB的大块I/O读写，而客户端B都是8kB的小块I/O读写，显然客户端A可能抢占客户端B的I/O带宽资源。为此，dmClock以8kB作为I/O块大小的基准，依据磁盘的平均寻道时间和磁盘的传输带宽，将大块I/O转换为8kB的倍数（计算公式请参考mClock算法论文）作为调整因子，作用在该I/O请求的时间标签上，以推迟该请求的处理时机，从而减少由于I/O大小悬殊带来的不利影响。然而，这种做法并不完美，依然存在如下一些问题：

1）无法实现对客户端I/O带宽的精确限制需求。上述对I/O大小进行转换是一种定性的计量方式，而且实际应用的I/O大小并不固定，这样显然无法实现I/O带宽精确限制。

2）影响原有QoS的准确性。比如某个客户端的QoS上限设置为100，当其以较大的I/O块大小（例如64kB）读写时，由于转换成8kB大小而调整了时间标签，使得每个I/O请求都被延迟处理，最终客户端得到的I/O上限必将低于100，这通常不是客户所期望的结果。

3）在dmClock算法刚提出的时候，机械硬盘是主流的存储介质，而随着SSD等高速设备逐渐普及，上述采用寻道时间的计算方式也不再适用。

基于上述种种因素，目前Ceph中的dmClock并未实现这种方式，也不支持对I/O带宽的限制。然而，作为一个完整的QoS控制策略，I/O带宽限制是一项必不可少的需求。特别是对块设备而言，顺序I/O通常会被客户端的操作系统合并，导致QoS的IOPS控制不准确。而I/O带宽的限制则不受此类合并操作的影响。

借用原有上限（limit）维度的基本原理，我们新增了一个QoS带宽（bandwidth）维度，以实现对I/O带宽进行精准限制的支持。参考dmClock原理部分的表述方法，用小写b表示客户端的带宽，大写B表示带宽时间标签，则来自client i的第r个I/O请求的带宽时间标签计算公式如下：

$$B_i^r = \max\left\{B_i^{r-1} + \left(\varphi_i^r + size\right)/b_i,\ t\right\}$$

其中，size表示当前I/O请求的大小，而$\varphi$与之前的ρ、δ含义类似，表示除目标服务器之外的其他服务器完成的I/O请求的大小之和。此外，关于Weight-Based阶段的I/O请求是否满足出队条件，原来只需要判断其上限时间标签是否小于等于当前时间即可，而此时则需要同时满足带宽时间标签小于等于当前时间的约束。

### 9.4.3 存储卷的QoS

前面我们介绍了client的多种可能界定方式，最终我们选择以块设备存储卷作为dmClock的一个client，主要是基于以下几点考虑：首先，虚拟机是一个OpenStack层面或者说上层应用的概念，如果以虚拟机为粒度，需要将client侧的实现置于虚拟机中，这意味着存储系统与上层应用强相关，显然不是我们所希望看到的；其次，以存储池为粒度又显得过于粗放，很多大型集群只有少数几个存储池，并且绝大部分卷共享同一个存储池，这种粒度的QoS显然没有太大的意义；最后，在许多客户的实际应用中，都有对卷粒度QoS功能的需求。因此，以存储卷作为client实现QoS功能，一方面可以与具体应用解耦，另一方面由于每个虚拟机都会关联一个或多个存储卷，对每个存储卷应用QoS可以间接作用于虚拟机，从而实现客户希望对每个虚拟机进行QoS控制的需求。

当然，对Ceph而言，以存储卷为粒度的QoS可能不是一种非常全面的做法，毕竟文件与对象存储不适用，但在实际应用中对块设备QoS的需求要远甚于其他两种存储接口。除此之外，对块设备QoS功能的实践，也可以为文件与对象存储的QoS设计提供参考和借鉴。因此，率先实现基于存储卷的QoS仍然具有积极意义。

由于Ceph是基于C/S访问模式的分布式存储系统，不同类型的应用需要通过不同类型的客户端与RADOS集群通信，为了实现存储卷的QoS，我们首先必须对相关的客户端（具体到存储卷，则是RBD

Client）和消息进行改造，使之能够传递基于dmClock实施QoS控制所必须的参数。考虑到实际应用中有动态调整QoS的需求（例如客户对购买的服务进行了升级），为了提供更好的用户体验，我们也实现了在线修改每个卷的QoS模板的功能。

1.模板与参数传递

为了实现存储卷的QoS，首先我们需要改造mclock_client队列，使其可以接受外部客户端（访问存储卷）下发的QoS模板，而不是使用上述默认的配置模板；然后改造访问存储卷的客户端，将QoS模板和请求完成统计信息随I/O请求下发下来。

模板与参数传递示意图如图9-10所示。其中，dmClock-server表示改造之后的mclock_client队列。rbd_header.<id>对象是存储卷（image）最重要的一个元数据对象，除了记录容量大小等基本信息外，还支持卷的私有配置参数以及用户自定义的元数据。QoS模板以自定义元数据的形式存储在rbd_header对象的omap中。

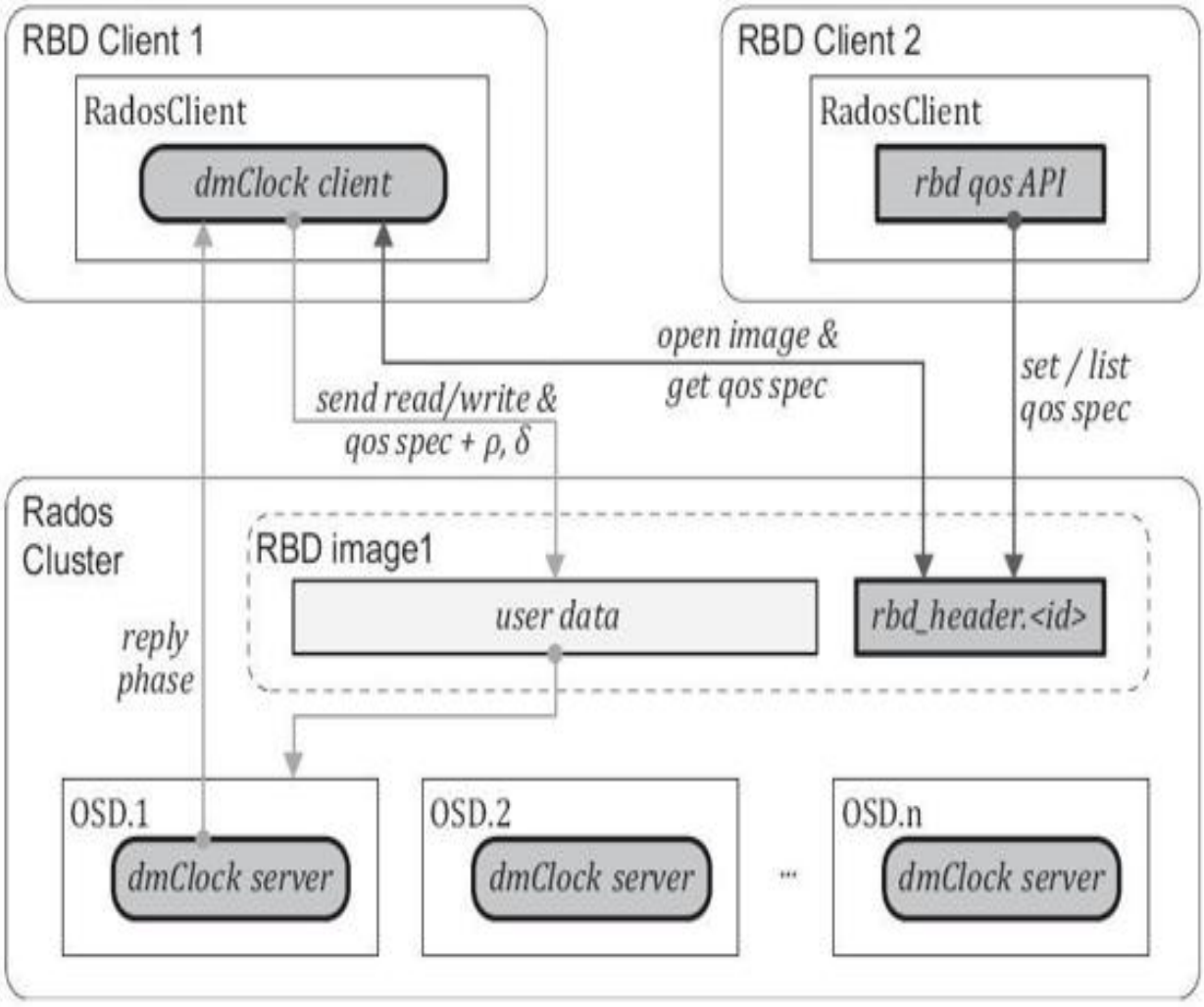



图9-10　QoS模板与参数传递示意图

在此，我们设计了一套RBD QoS API接口，用于设置或查询存储卷的QoS模板。通过API接口设置的模板参数被保存至存储卷的元数据中，当客户端打开卷的同时从元数据中读取QoS模板，以便发送读写I/O请求时携带该模板。为了支持QoS模板以及相关参数的传递，我们需要修改MOSDOp/MOSDOpReply消息，增加相关字段。当MOSDOp消息下发至OSD后，将被转换为op进行跟踪，该请求在进入dmClock-server队列时使用QoS模板以及ρ和δ值计算时间标签，并在随后出队时将phase信息保存至op之中。最后，在op被OSD处理完成后，我们将phase信息通过MOSDOpReply消息返回给客户端。收到phase信息后，

客户端将更新dmClock-client中对应服务器（此处为OSD）的ρ和δ值，以便积累两个请求之间所有服务器ρ和δ的增量，并在随后的I/O请求中应用和下发。

2.在线配置生效

当一个客户端正在使用某个存储卷时，需要感知元数据中QoS模板发生了变化，才能及时应用新的模板，这就是QoS模板的在线配置生效功能。通常访问一个RBD存储卷时，总是需要先打开一个RadosClient（顾名思义，RadosClient是一种用于与RADOS集群通信的通用客户端），包括调用API修改卷的QoS模板以及客户端读写存储卷两种访问方式莫不如此。但是两者打开的RadosClient通常不在同一个进程中，甚至不在同一个主机之上。因此需要一种消息同步机制通知正在使用该卷的客户端及时更新模板。而RADOS的Watch/Notify机制正好可以实现这个目标。

基于Watch/Notify机制实现的QoS模板在线配置生效过程如图9-11所示。其中，RBD Client 1为正在使用存储卷的客户端，RBD Client 2为尝试修改卷QoS模板的客户端。rbd_header.<id>对象除了保存QoS模板信息之外，也是该存储卷的客户端之间Watch/Notify通信的目标对象。此外，控制存储卷互斥访问的排他锁（exclusive-lock）信息也保存在该对象的扩展属性（xattr）当中，键名为lock.rbd_lock，值为持有该锁的客户端id以及ip等信息，该特性默认总是开启。

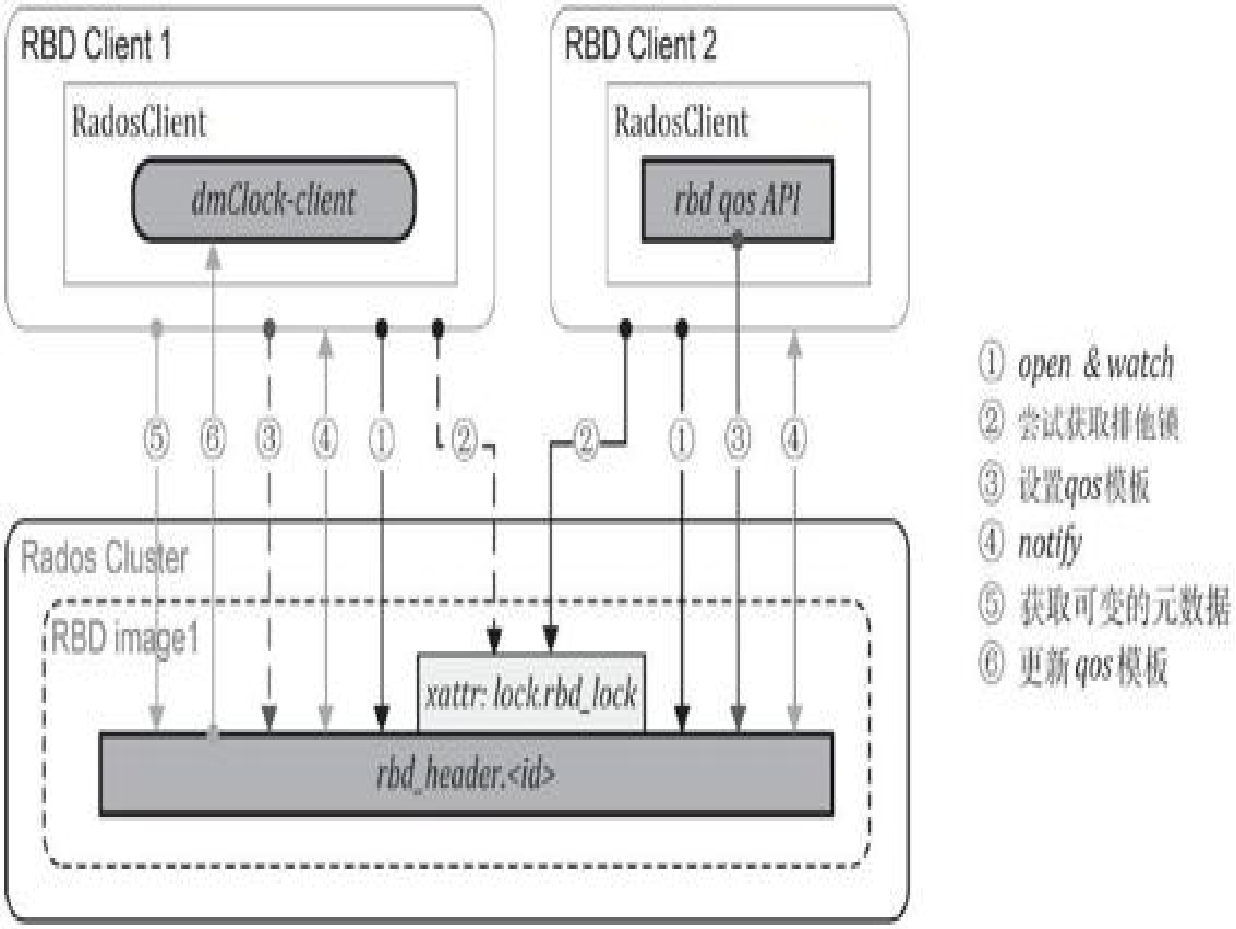



图9-11　在线配置生效示意图

为了对在线配置生效过程进行详细阐述，我们用形如1.①的格式表示Client 1的第①步，以此类推。首先，每个客户端以可写方式打开一个存储卷时，都会向rbd_header对象发送watch消息（如图9-11中1.①与2.①所示），以注册成为该卷的一个watcher，便于后续通过notify消息与其他客户端同步信息。对Client 1，如果需要对存储卷进行写操作，通常它会尝试获取排他锁，如图1.②所示（虚线表示其后可能被抢占）；当Client 2开始修改QoS模板时，首先尝试获取排他锁（2.②），如果获取成功则将新的QoS模板写入元数据（2.③，此处为rbd_header的omap）中，同时向rbd_header对象发送notify消息（2.④），通知其他watcher元数据已更新，其自身也将收到该notify消息（2.④）；Client 1收到了该notify消息（1.④），随后向rbd_header对象获取最新的可变元

数据（1.⑤，QoS模板属于可变元数据），并将新的QoS模板更新至dmClock-client中（1.⑥）。

如果2.②获取排他锁失败，Client 2将获知当前哪一个客户端持有锁，然后向其发送notify消息（2.④，注意其与前面notify消息的不同），该notify消息中包含了需要设置的新的QoS模板；Client 1收到该notify消息（1.④）后，将新的QoS模板写入元数据中（1.③），同时发送notify消息（1.④）通知包括自身在内的watcher元数据已发生变化；收到该notify消息（1.④）后，Client 1通过与上面相同的1.⑤及1.⑥步骤更新QoS模板，从而达到在线配置生效的目的。

此外，需要同步修改dmClock-server部分，在检测到client下发的QoS模板发生变化时，及时应用新的QoS模板，同时更新后续I/O请求的基准时间标签。

### 9.4.4 集群流控策略

集群后端（指内部的）I/O操作（即snaptrim、scrub和recovery）对前端（指外部客户端）的I/O请求影响较大。对snaptrim和scrub而言，可以通过设置相应的QoS模板降低执行频率，并结合现有的策略将其限定在前端I/O低发的时间段（例如0：00～6：00期间）在后台缓慢地执行。但是recovery类型（参见表9-2，可以由Recovery或者Backfill触发）的I/O则有所不同，如果纯粹由Backfill触发（例如对集群进行了扩容等），由于这种情况下都是整对象拷贝，如不加以限制可能会对前端业务造成极大影响。而如果是由Recovery触发，则表明集群此时处于降级状态，一方面我们总是期望Recovery能尽快完成，以降低数据丢失风险；另一方面在增量Recovery生产就绪之前，Recovery同样是以整对象为粒度的磁盘带宽密集型I/O操作，如果Recovery速度过快则必然会对前端I/O请求的时延造成严重影响，因此我们也需要针对可用于Recovery的磁盘带宽进行限制。

事实上，在Recovery过程中，对前端I/O性能造成重大影响的主要原因有以下几个（以多副本为例）：

1）Primary中存在降级对象，当前端I/O请求访问到这些对象时，必须通过Pull的方式先将其修复，才能响应前端I/O请求。

2）副本中存在降级对象，前端I/O对该对象的写请求将会被阻塞，直到Primary以Push的方式将其修复。

3）OSD正在修复降级对象，磁盘带宽被大量占用，无法及时响应前端I/O请求。

由前两个原因所导致的问题可以通过异步Recovery的方式缓解，但是如果由于Backfill或者最后一种情况，出现磁盘带宽被过度占用，导致前端I/O无法得到及时响应的问题，则必须通过QoS的带宽限制功能加以解决。

1.OSD数据恢复的流控

在第7章中我们已经详细介绍过OSD级别的数据恢复流控策略，例如通过调整osd_max_backfills和osd_recovery_sleep等参数，可以在一定程度上减少Recovery或者Backfill对前端I/O的影响。不过这只是一种定性控制，无法准确评估对于前端业务的影响。而通过调整mclock_client队列中recovery类型client的QoS模板的方式，虽然可以控制Primary触发数据恢复操作的频率，却无法限制副本侧OSD数据恢复的流量。因此，我们采用dmClock新增的带宽维度来精确控制每个OSD可用于数据恢复的流量，进而实现集群级别的定量流控策略。

简言之，可以将Primary的Pull/Push操作当成客户端的读/写请求，Primary所在的OSD作为dmClock-client触发数据恢复，副本所在的OSD作为dmClock-server响应Pull/Push请求。由PG分布的随机性可知，对整个集群而言，每个OSD既是客户端又是服务器。此外，由于只要限制副本侧OSD的整体数据恢复流量，而无须关注具体由哪个PG触发，与mclock_opclass队列中的客户端类似，所有作为客户端的OSD共享一套QoS模板即可。

由于OSD对应磁盘的物理带宽是固定的，数据恢复与前端I/O流量此消彼长，因而只需控制其中的数据恢复流量即可。为此，我们将数据恢复速率分成多个级别（例如，高、中、低），每个级别的流量都可以通过QoS模板精确调整；同时，在OSD中加入一个负载均衡器

（load_balancer），用于实时监测OSD中的数据恢复和前端I/O流量，并据此自动调整至合理的数据恢复等级。

2.集群流控自适应模式

对Ceph集群而言，通常要求支持前端业务优先、数据恢复优先两种流控模式。当集群工作在前端业务优先模式时，数据恢复速率将被严重抑制，而前端I/O性能几乎不受影响；相反，当选择数据恢复优先模式时，则无法保证前端业务性能，而后端数据恢复将以最快的速率完成。

显然上述两种模式各有各的适用场景，也各自存在缺陷。我们在这两种常见模式的基础上开发了第三种流控模式——自适应模式。顾名思义，自适应模式能够根据集群当前的I/O流量，自动调整前、后端I/O的处理速率，从而在对前端业务影响较小的情况下，尽快完成后端数据恢复而无须人工操作，因而是一种更为智能和实用的流控模式。为了实现自适应模式，我们需要一个信息集中点，定时收集所有PG状态及其包含的对象数量等信息，而恰好Mgr存在这样的机制。基于Mgr和上述单个OSD粒度的数据恢复流控，可以实现如图9-12所示的集群流控自适应模式。

由Ceph基于计算随机分布数据的特点，我们知道每个PG都保存了来自不同客户端的数据。因此，当集群中存在大量PG需要进行数据恢复时，我们总是倾向于使得所有PG都尽可能同时恢复正常，以免集群性能与可靠性受恢复时间最长的那个PG制约。因此，与Ceph自带的数据恢复流控策略相比，启用自适应模式首先必须放开可并发进行数据恢复的PG数目限制（osd_max_backfills）。

同样，由于对象被映射至PG的过程是随机的，每个PG实际存储的对象数目都可能不同，某些情况下甚至相差很大。当放开上述限制之后，虽然所有执行数据恢复PG的起点相同，但每个PG待恢复对象的数

量可能不同，致使包含这类对象少的PG数据恢复先于这类对象多的PG完成。随着时间推移，能够并发进行数据恢复的PG越来越少，数据恢复流量越来越低，从而延长整体数据恢复时间。为此，需要提高待恢复对象多的PG的并发对象数，使它们可以获得更高的数据恢复流量；同时减少那些待恢复对象少的PG的并发对象数，以降低它们的数据恢复流量（如图9-12中的①）。这样，通过实时监控与动态调整，我们最终能够使得所有PG都倾向于同时完成数据恢复，从而获得一个总体上的最短数据恢复时间，进而有效降低数据丢失风险。

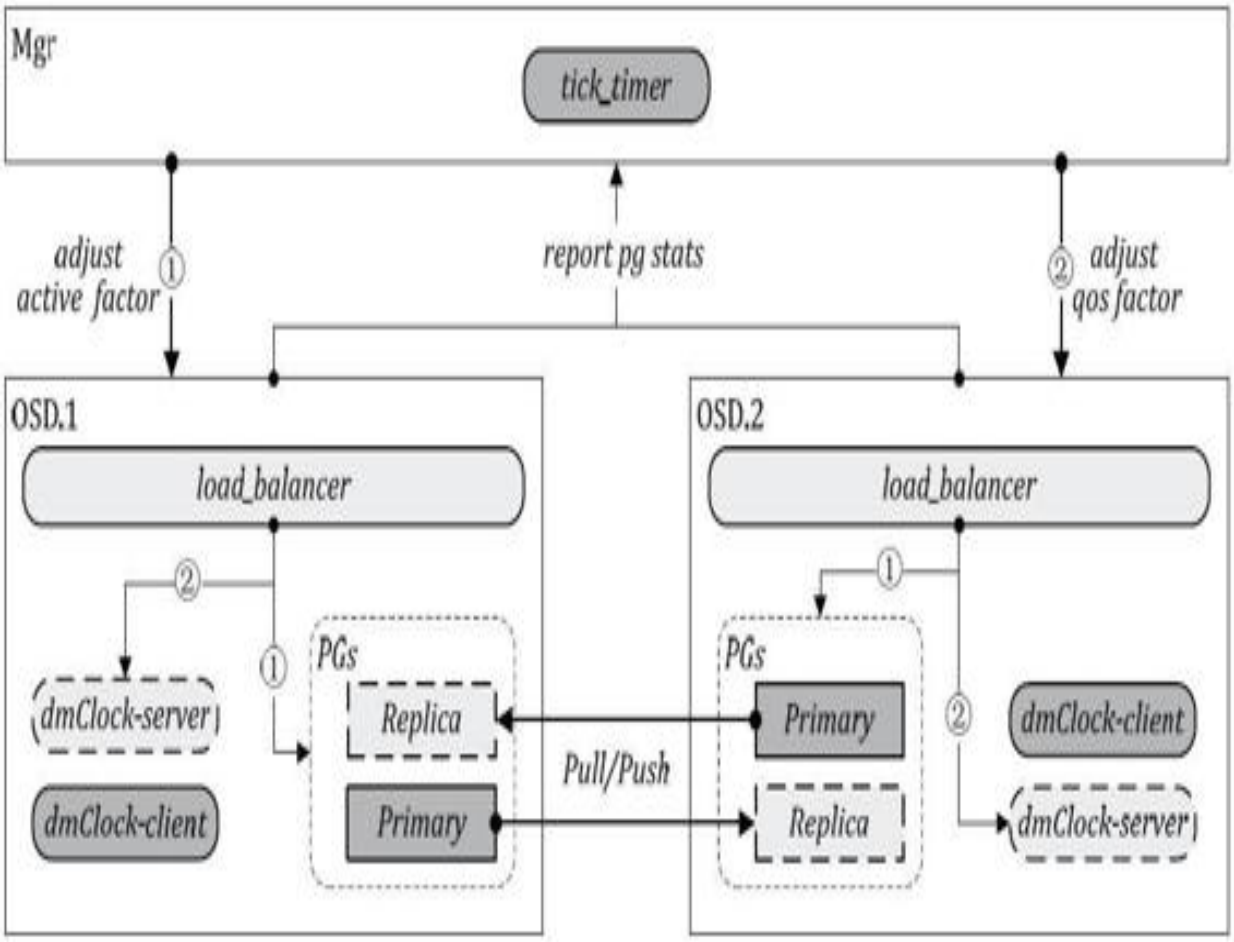



图9-12　集群流控自适应模式

上述流控策略同样适用于单纯由数据重平衡触发的数据迁移过程，即Backfill。由于我们总是倾向于让PG在OSD之间趋于均匀分布，

如果某个OSD待Backfill的对象数量多（之所以要Backfill这个OSD，理论上意味着这个OSD的PG较少），那么相应地其在整个集群中分摊的客户端流量也少；与之对应的待Backfill对象少的OSD分摊的客户端流量理论上也应该更多。这说明大多数情况下，当存在客户端流量时，按照统一尺度为每个OSD分配相同的Backfill带宽并不合理，而是需要在保持整个集群Backfill带宽恒定（需要根据经验数据确定）的前提下，尽可能地尝试为当前包含待Backfill对象数量多的OSD分配更大的Backfill带宽，同时减少包含待Backfill对象数量少的OSD的带宽（如图9-12中的②），进而保证最终所有OSD的Backfill流程都趋于同步完成。

综上，无论是何种原因触发的集群内部流量，通过上述集群流控模式改造我们总是可以自动在各种任务之间取得平衡，在最大程度降低客户端所受影响的同时，以最快速度恢复或者重新分布集群数据。

## 9.5 总结和展望

分布式存储系统的QoS是一个复杂的工程，我们与社区一起为之投入了多年时间，目前已经实现了dmClock算法部分。作为op_shardedwq队列的一种实现方式，这部分已经能够正常工作，但如何结合实际需求将其真正应用起来，还是我们努力的目标。本章介绍的一些探索和实践并不一定是最合理的方式，但能解决实际问题。我们也将继续致力于完善dmClock算法本身的实现，并将其更好地应用在Ceph当中。

关于dmClock的应用，目前还存在如下一些问题，这也是我们后续重点关注的方向。

（1）模板的合理设置

如何设置合理的QoS模板值，使得客户端能满足实际的应用场景需求，并且使系统的I/O资源利用率达到最高，或者内部各类I/O操作之间达到平衡和最优状态，这些都是后续需要重点考虑的问题。对于预留这个维度，需要保证所有client设置的预留之和处于系统的I/O处理能力之内，所以需要预先对系统的处理能力进行评估，对每个client的预留值进行合理的规划，并要考虑后续增加client时对预留的影响。另外，Ceph是个基于计算的分布式存储系统，不存在集中控制点，这对于我们期望通过权重控制来解决多个客户端对I/O资源的占用比重问

题是一个不利影响。由于Ceph客户端的I/O请求直接发往OSD，在极端情况下，当两个客户端分别与不同的OSD通信时，它们的权重起不到什么作用。社区已有相关的研究表明，当OSD的负荷不足时（指请求入队速率低于出队处理速率），不同权重的客户端所获得的系统I/O资源大致相等；只有当客户端请求足够分散，请求压力足够大，使得集群所有的OSD都达到超负荷时（即请求入队速率高于出队速率），权重的效果才能体现出来。这是由于负荷不足时，可以认为客户端的请求都已经得到了及时处理，不需要在队列中排队进行权重竞争；而超负荷时，请求都随机分布到了所有OSD中，每个OSD都通过权重分配I/O资源，从而整体上能够达到预设的权重效果。

### （2）突发I/O的处理

在原始的dmClock算法当中，考虑了对突发I/O的处理。这里的突发I/O是指某个客户端突然有很高的IOPS需求，即短时间内下发了很多I/O请求的情况。如果按照不考虑突发I/O情况的处理逻辑，由于客户端的QoS模板没有改变，系统将仍以自己的节奏处理这些I/O请求，并不能感知客户端的紧迫需求。为此，dmClock的做法是为每个client预先设置一个可调整的参数σ（sigma），当出现突发I/O的情况时，减少该client的权重时间标签值（调整其基准标签至$t-\sigma_i/w_i$），从而让其在权重竞争时更具优势。这样在不影响预留的情况下，可以短时间内在Weight-Based阶段响应该client的大量I/O请求，从而及时处理突发情况。但这里存在一个问题，即服务端如何判断客户端出现了突发I/O访问？目前的做法比较简单，当一个客户端一段时间未产生I/O请求，则将它的状态置为空闲状态；再次有I/O请求到来时变为活跃状态，服务端记录了客户端的状态；当检测到客户端状态从空闲变为活跃时，则认为客户端将出现突发I/O访问。然而不幸的是，这仅仅是突发I/O可能出现的情况之一。

# 第10章
# 纠删码原理与实践

不同类型的存储系统其组成形态和存储规模各异，然而无论何种类型的存储系统，它们都面临一个相同考验——由组件失效导致的数据丢失风险。组件失效的原因多种多样，小到磁盘扇区出现静默数据错误（机械磁盘由于电容老化或者电磁辐射可能导致某些比特出现误翻转，从而产生静默数据错误），大到某个主机甚至整个机柜异常掉电。有些组件本身具有容错机制，例如磁盘，实现上会针对每个扇区额外填充一些校验信息，这样即使少数几个比特出错，仍然可以基于校验信息自动对这些出错的比特进行修复，从而保证整个扇区的数据不致损坏。然而，上述容错机制在应对诸如磁盘消磁甚至遭到物理破坏这类极端情况时依然无能为力，因此还需要在系统层面设计额外的数据保护机制，防止数据因为个别组件完全失效而遭受不可逆转的损坏。

起源于通信系统的纠删码（Erasure Coding，EC）是目前在存储系统中广泛使用的一种数据保护机制。纠删码首先针对原始数据进行分片，然后基于分片进行编码以生成备份数据，最后将原始数据和备份数据分别写入不同的存储介质。数据恢复是编码的逆运算（为了能够进行数据恢复，要求编码采用的方法（函数）一定是可逆的），因此

也称为解码。纠删码的容错能力取决于生成备份数据的份数，这也是系统允许同时失效的最大存储介质数目。通常情况下，生成备份数据的份数越多，则对应的编码法则越复杂，同时也将消耗更多的额外存储空间。综合考虑编解码效率和存储空间成本，生产环境中很少会使用阶数（即备份数据份数）高于2的纠删码。

纠删码最简单的实现方式是完全复制，也称为镜像，即将数据不做任何处理同时写入多个不同的存储介质。这样，只要数据还有一个备份存活，就不必担心数据丢失，但代价是会消耗与阶数同等倍数的额外存储空间。为了降低存储空间成本，纠删码也发展出了其他更复杂的实现方式，典型的如高阶RAID（Redundant Array of Independent Disks，这里指RAID5或者RAID6）以及由RAID衍生而来、更具一般性的RS-RAID（Reed-Solomon RAID）。它们通过更加复杂的编解码策略建立原始数据和备份数据之间的映射关系，以增加计算成本作为代价，减少需要备份的数据量，从而降低存储空间成本。

本章按照如下形式组织：首先介绍在传统存储系统（SAN/NAS）中被广泛使用的数据保护机制RAID；以RAID技术为基础，通过分析其背后隐含的数学原理，我们可以推导出更具代表性的RAID表现形式——RS-RAID，以及由此发展而来的Jerasure标准库；得益于纠删码标准库的日益成熟和丰富，社区决定自Kraken版本起，增加对纠删码的一些高级特性，例如覆盖写（overwrites）的支持，我们将针对这些新增特性重点进行介绍，并借此分析后续将纠删码全面推广至生产环境时所面临的挑战。

## 10.1 RAID技术概述

数据存储在单个磁盘上，存在以下固有缺陷：

·访问速度慢。单一的I/O接口，无法实现并发。

·容量小。尽管（单个）磁盘容量不断提升，但是仍然无法满足呈爆炸式增长的数据存储需求。

·安全性差。容易成为一个单点故障点，安全性较差。

如果单个磁盘无法满足访问速度、容量、安全性等要求，那么可以使用多个磁盘组合起来提供存储服务。这些磁盘应该如何进行组合才能达到最优呢？RAID技术通过对可能的组合方式进行探索，做出了一些有益尝试。

我们已经知道，单个磁盘上的数据是以扇区作为基本单位进行存储和访问的。RAID抽象出一个类似于扇区的最小数据访问单位——条带（如图10-1所示）。这是一种虚拟化的设计方法，即RAID中的条带化，它并非将磁盘再次物理格式化为条或者带，而是想办法在多个磁盘之间建立一种逻辑映射关系。通过这种逻辑映射关系，RAID可以将多个物理磁盘抽象为一个容量更大、I/O并发程度更高的虚拟磁盘，以条带作为基本单位进行管理。

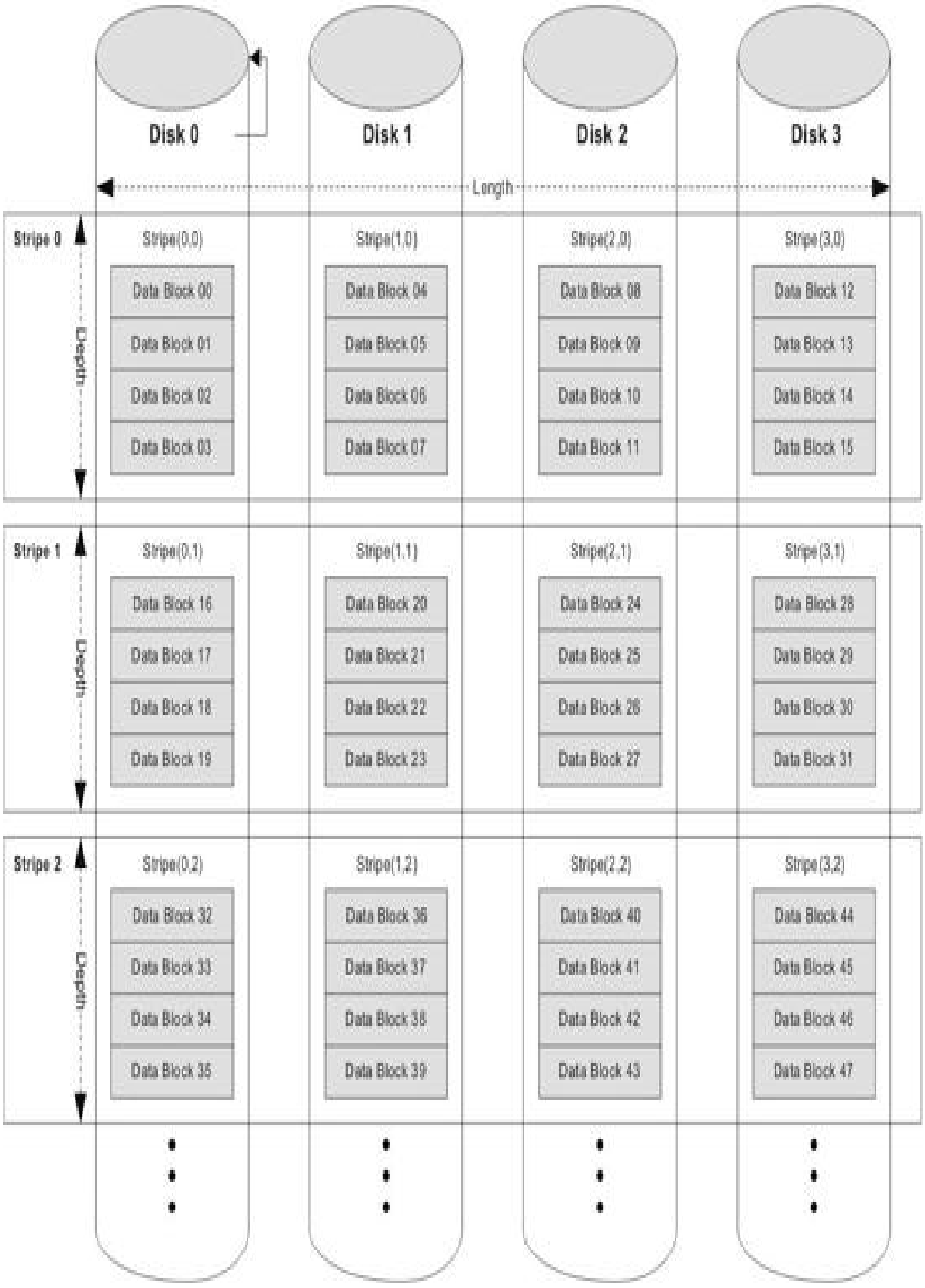

Disk 0
Disk 1
Disk 2
Disk 3
Length
Stripe 0
Stripe 1
Stripe 2
Depth
Depth
Depth
Stripe(0,0)
Stripe(1,0)
Stripe(2,0)
Stripe(3,0)
Data Block 00
Data Block 01
Data Block 02
Data Block 03
Data Block 04
Data Block 05
Data Block 06
Data Block 07
Data Block 08
Data Block 09
Data Block 10
Data Block 11
Data Block 12
Data Block 13
Data Block 14
Data Block 15
Stripe(0,1)
Stripe(1,1)
Stripe(2,1)
Stripe(3,1)
Data Block 16
Data Block 17
Data Block 18
Data Block 19
Data Block 20
Data Block 21
Data Block 22
Data Block 23
Data Block 24
Data Block 25
Data Block 26
Data Block 27
Data Block 28
Data Block 29
Data Block 30
Data Block 31
Stripe(0,2)
Stripe(1,2)
Stripe(2,2)
Stripe(3,2)
Data Block 32
Data Block 33
Data Block 34
Data Block 35
Data Block 36
Data Block 37
Data Block 38
Data Block 39
Data Block 40
Data Block 41
Data Block 42
Data Block 43
Data Block 44
Data Block 45
Data Block 46
Data Block 47

图10-1　条带

图中单个条带（Stripe）所跨过的磁盘个数，称为条带宽度（或者条带长度Length）；

单个条带在单个磁盘上分配的字节数，称为条带深度；

条带深度一般为磁盘基本块（指访问磁盘的最小粒度，例如扇区）的整数倍。

目前主流的RAID技术有如下几种基本模式。

### （1）RAID0

RAID0将多个磁盘（这里假定所有磁盘规格相同，下同）以条带为单位重新划分，形成一个地址空间在逻辑上连续的虚拟磁盘。其I/O能力以及可用存储空间是所有磁盘之和，但是不具备容错能力。一个典型的RAID0系统如图10-2所示。

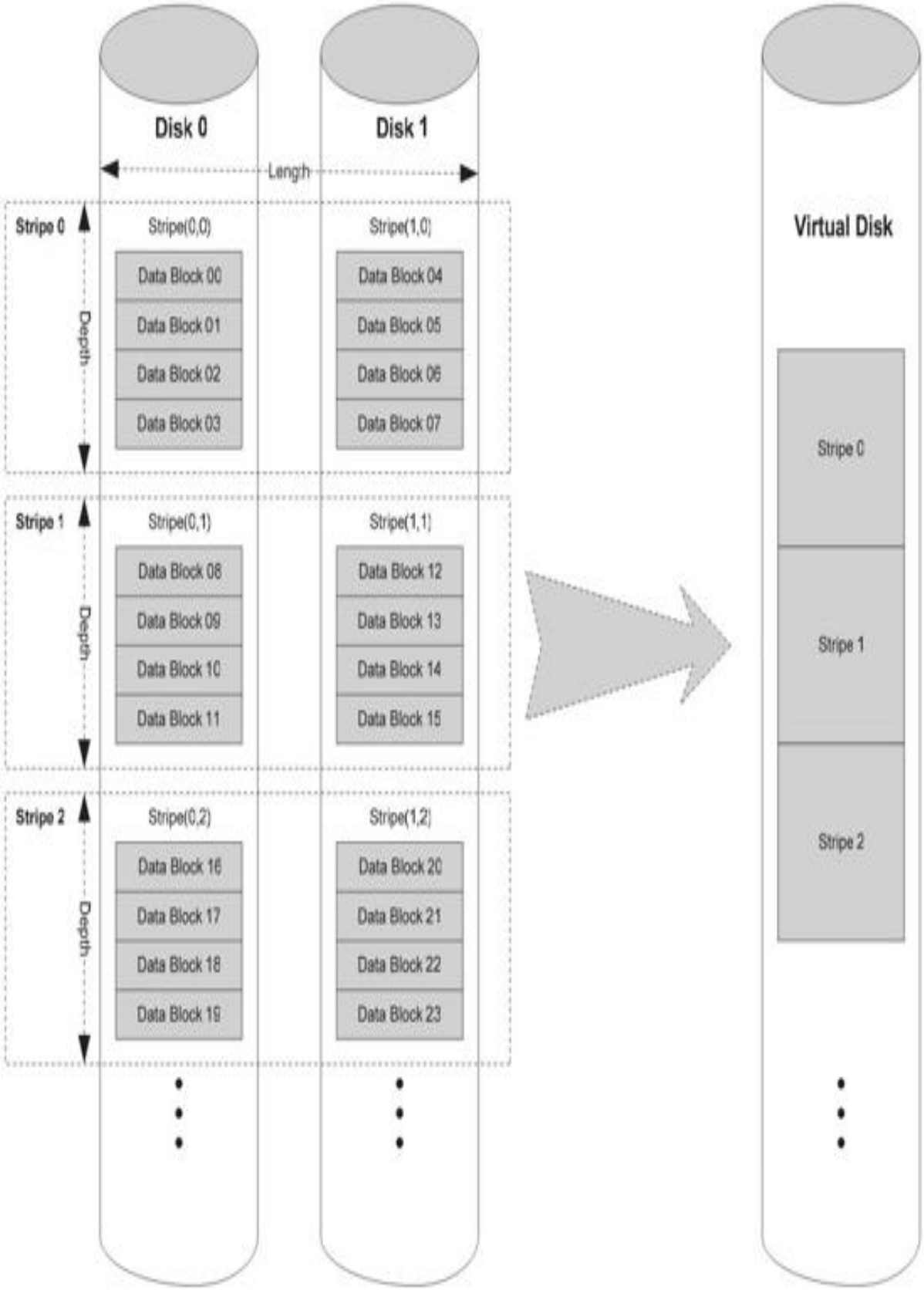

Disk 0
Disk 1
Length
Stripe 0
Stripe(0,0)
Stripe(1,0)
Data Block 00
Data Block 01
Data Block 02
Data Block 03
Data Block 04
Data Block 05
Data Block 06
Data Block 07
Depth
Stripe 1
Stripe(0,1)
Stripe(1,1)
Data Block 08
Data Block 09
Data Block 10
Data Block 11
Data Block 12
Data Block 13
Data Block 14
Data Block 15
Stripe 2
Stripe(0,2)
Stripe(1,2)
Data Block 16
Data Block 17
Data Block 18
Data Block 19
Data Block 20
Data Block 21
Data Block 22
Data Block 23
Virtual Disk
Stripe 0
Stripe 1
Stripe 2

图10-2　RAID0系统

### （2）RAID1

RAID1将同一份数据重复写入两个或多个磁盘，即每个磁盘存储的内容都是完全相同的，因此RAID1也称为镜像，它能够提供的I/O能力以及可用存储空间只有所有磁盘的1/N（N为镜像个数）。由于只要任意一个镜像存活，就可以确保数据不会丢失，因此RAID1的容错能力与镜像个数成正比。一个仅包含两个磁盘的RAID1系统如图10-3所示。

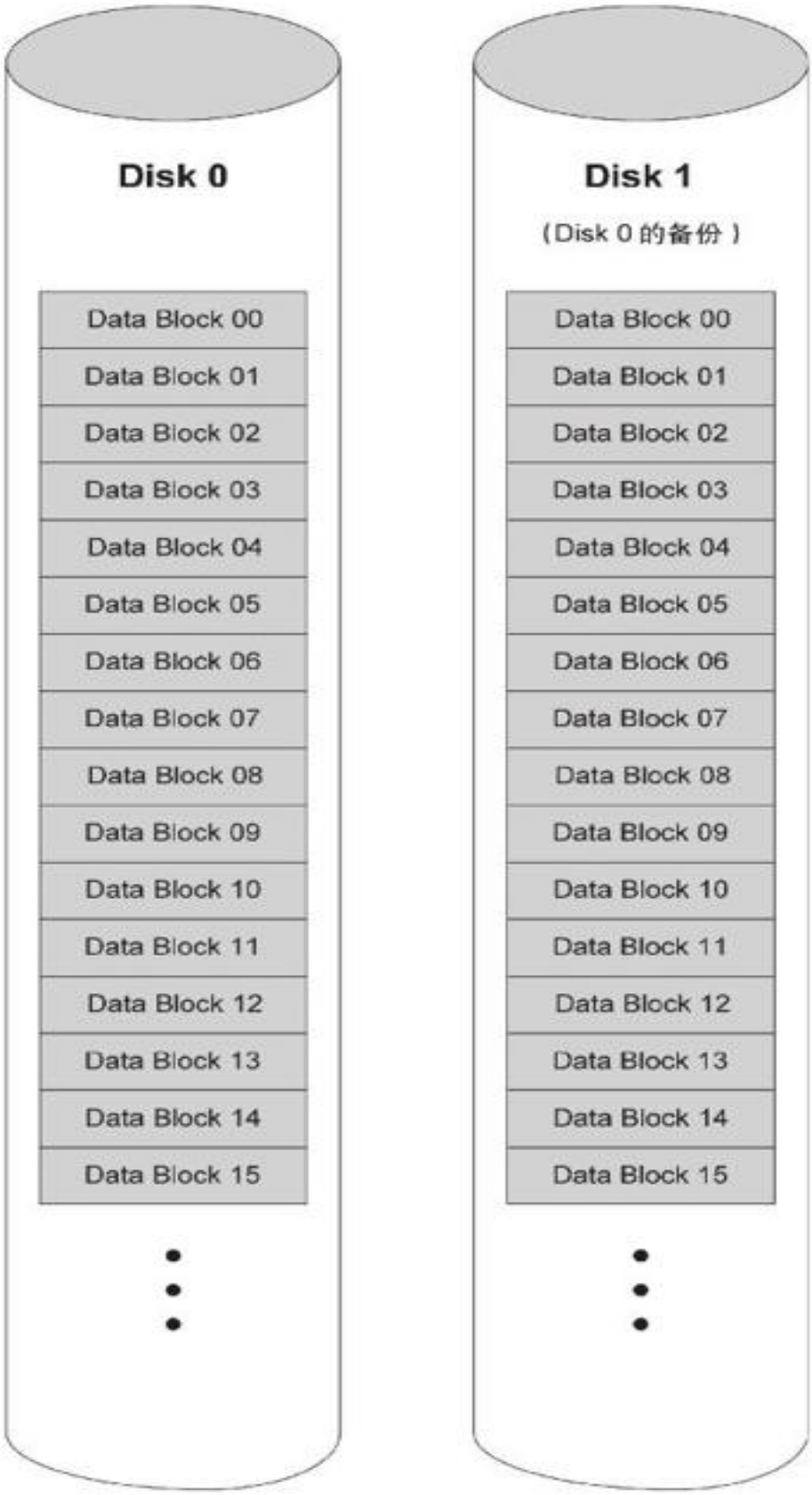

Disk 0
Data Block 00
Data Block 01
Data Block 02
Data Block 03
Data Block 04
Data Block 05
Data Block 06
Data Block 07
Data Block 08
Data Block 09
Data Block 10
Data Block 11
Data Block 12
Data Block 13
Data Block 14
Data Block 15
Disk 1
(Disk 0 的备份)
Data Block 00
Data Block 01
Data Block 02
Data Block 03
Data Block 04
Data Block 05
Data Block 06
Data Block 07
Data Block 08
Data Block 09
Data Block 10
Data Block 11
Data Block 12
Data Block 13
Data Block 14
Data Block 15

图10-3　RAID1系统

两个磁盘保存的内容完全一致，互为镜像

### （3）RAID5

RAID5以一个或多个基本块作为条带深度均分数据，同时基于异或运算生成一个与数据块同等大小的校验块。因此RAID5能够提供的I/O能力以及可用存储空间为所有磁盘的(N-1)/N，其中N为组成RAID5的磁盘个数，即RAID5的条带宽度。由异或运算性质，如果条带中任意一个数据块出错，都可以通过其他仍然正常的数据块与校验块执行异或操作进行数据恢复，因此RAID5具有一定的容错能力，至多允许一块磁盘异常。此外，由于正常情况下并不需要读取校验块，为了避免浪费读带宽，RAID5条带中校验块的位置不是固定不变的，而是不停地在不同磁盘之间跳动。图10-4展示了一个条带宽度为4、条带深度为4个基本块的RAID5系统。

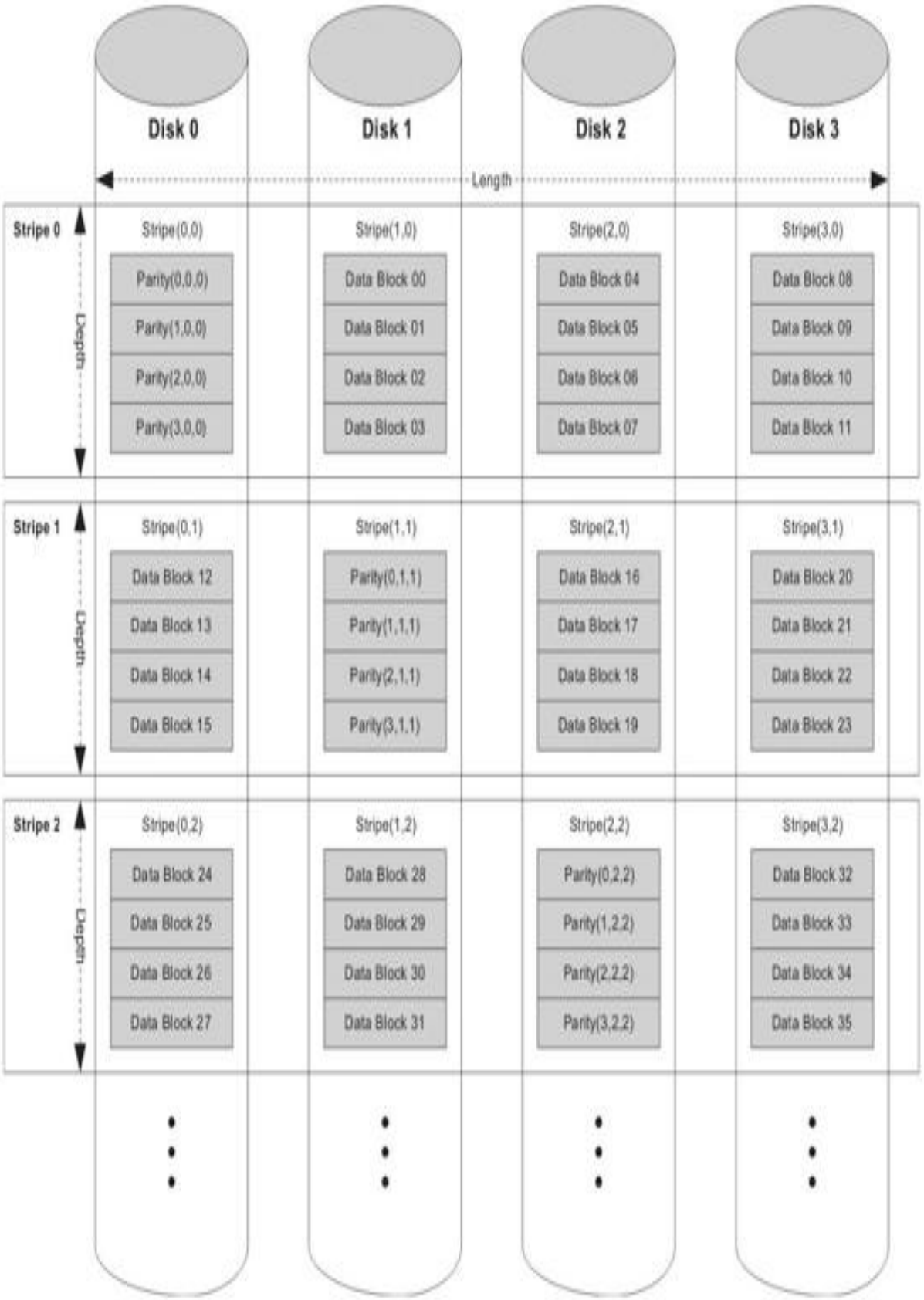

Disk 0
Disk 1
Disk 2
Disk 3
Length
Stripe 0
Depth
Stripe(0,0)
Parity(0,0,0)
Parity(1,0,0)
Parity(2,0,0)
Parity(3,0,0)
Stripe(1,0)
Data Block 00
Data Block 01
Data Block 02
Data Block 03
Stripe(2,0)
Data Block 04
Data Block 05
Data Block 06
Data Block 07
Stripe(3,0)
Data Block 08
Data Block 09
Data Block 10
Data Block 11
Stripe 1
Depth
Stripe(0,1)
Data Block 12
Data Block 13
Data Block 14
Data Block 15
Stripe(1,1)
Parity(0,1,1)
Parity(1,1,1)
Parity(2,1,1)
Parity(3,1,1)
Stripe(2,1)
Data Block 16
Data Block 17
Data Block 18
Data Block 19
Stripe(3,1)
Data Block 20
Data Block 21
Data Block 22
Data Block 23
Stripe 2
Depth
Stripe(0,2)
Data Block 24
Data Block 25
Data Block 26
Data Block 27
Stripe(1,2)
Data Block 28
Data Block 29
Data Block 30
Data Block 31
Stripe(2,2)
Parity(0,2,2)
Parity(1,2,2)
Parity(2,2,2)
Parity(3,2,2)
Stripe(3,2)
Data Block 32
Data Block 33
Data Block 34
Data Block 35

图10-4　4个磁盘构成的RAID5系统

常见的RAID5工程实现中，条带宽度和深度都是固定的（这样比较简单）。但是也有特例，例如ZFS自带的RAID5技术（ZFS称为RAIDZ）将条带宽度和深度设计成不固定的，而是会随着前端下发的I/O大小动态进行调整。这样做一方面可以避免空间浪费，另一方面也可以提升I/O并发能力。例如，假定条带宽度和深度固定，分别为5（即4个数据块+1个校验块）和4kB，那么容易验证这种形式的RAID5最适合处理大小固定为16kB的I/O；反之，如果I/O大小发生了变化，例如变为4kB，那么此时条带中将存在3个空穴，只能使用全0的数据填充，此时将浪费12kB的存储空间和3个磁盘的I/O能力。针对后面这种情况，RAIDZ会自动将条带深度调整为1kB，从而避免空间浪费，也最大程度地利用RAID组内磁盘的I/O并发能力。图10-5展示了一个RAIDZ磁盘条带分布的例子。

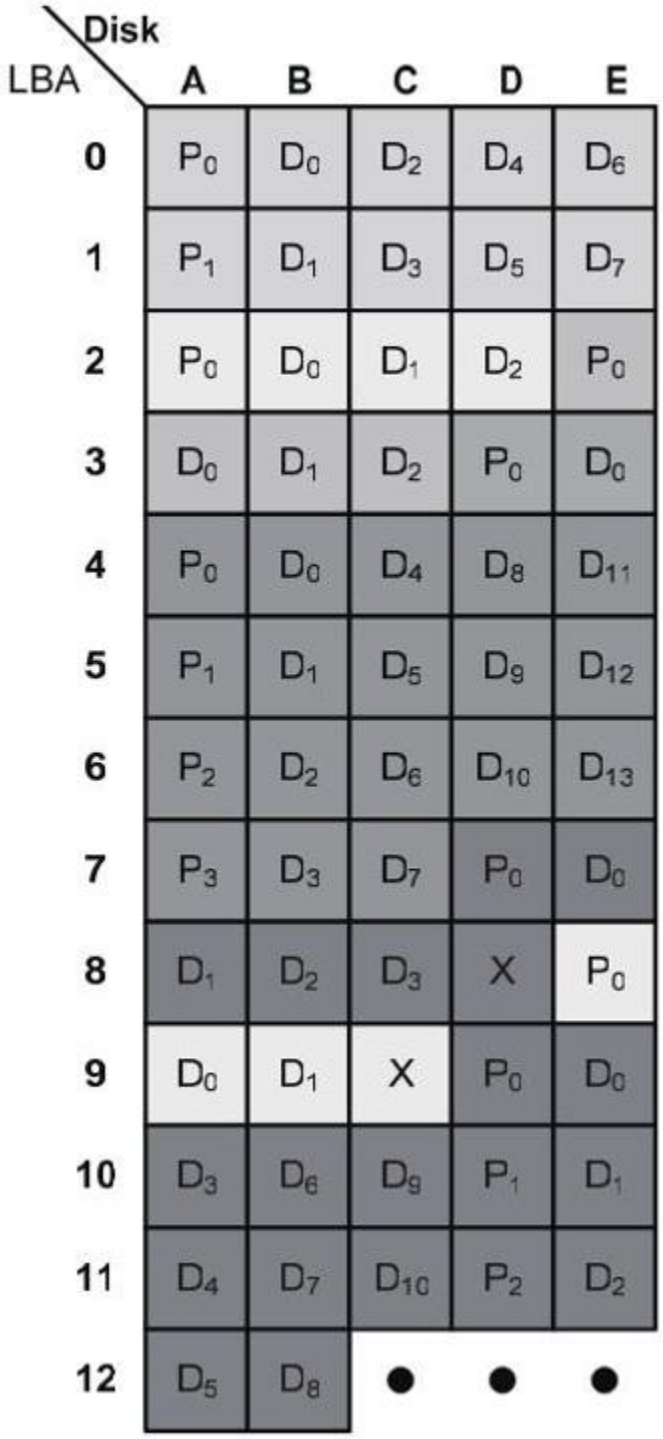



图10-5　RAIDZ条带在磁盘之间的分布

D（Data）和P（Parity）分别指条带中的数据块和校验块，由图可见，RAIDZ的条带宽度和深度都不固定。

### （4）RAID6

RAID6在RAID5的基础上进一步提升了容错能力，允许RAID阵列中同时有两块磁盘故障。原因在于RAID6为每个条带增加了一个校验块，即每个条带同时包含两个校验块，因此RAID6的空间和I/O利用率都比RAID5低，同时生成校验块的算法也更加复杂。

上述几种RAID形式中，RAID0的速度是最快的，可以将其与其他RAID形式组合，实现更高级的RAID形式。常见的组合如RAID10、RAID50等。

## 10.2 RS-RAID和Jerasure

我们已经了解了RAID的相关概念，本节我们分析高阶RAID（RAID5及以上）技术隐含的数学原理，并尝试将其推广至一般形式。

RAID5基于条带将数据均匀切分成多个数据块$d_i$(i=1，2，…，n)，然后尝试建立这些数据块之间的联系。例如使用普通加法：

$$d_1+d_2+d_3+\ldots+d_n=c$$

上式中，c为基于加法生成的校验块。这样，如果任意一个数据块损坏（此时等式中的某个$d_i$变为未知），都可以通过求解上述一元一次方程进行数据恢复。然而上面这个方法却不会在实际中应用，原因在于普通的加法容易产生进位，因此为了记录所有数据块之和，校验块占用的存储空间一般情况下要大于参与计算的任一数据块的存储空间。对计算机而言，最好的替代方案就是采用布尔运算（这是计算机得以发明的基础之一），具体要替代普通加法运算，则应该使用异或运算，这是因为：首先，异或是基于位的运算（简单并且是计算机最高效的运算方式）；其次，为了求解上述方程，需要对应运算的逆运算存在并且结果唯一，而异或运算的逆运算不但存在而且与正向运算完全相同；最后也最重要的是，异或运算不会产生进位，因此存储异或运算的结果并不需要比参与运算的任一对象更多的空间。

RAID6或者更高阶的RAID（例如ZFS实现了能够容忍3块磁盘同时故障的RAIDZ3）原理上和RAID5类似，实际上是利用条带中n个数据块通过编码得到m个校验块（m为允许同时故障的最大磁盘数目），如果出现磁盘故障，通过编码的逆向过程（称为解码）可以还原得到所有缺失的数据块，从而实现数据恢复。这个过程如图10-6所示。

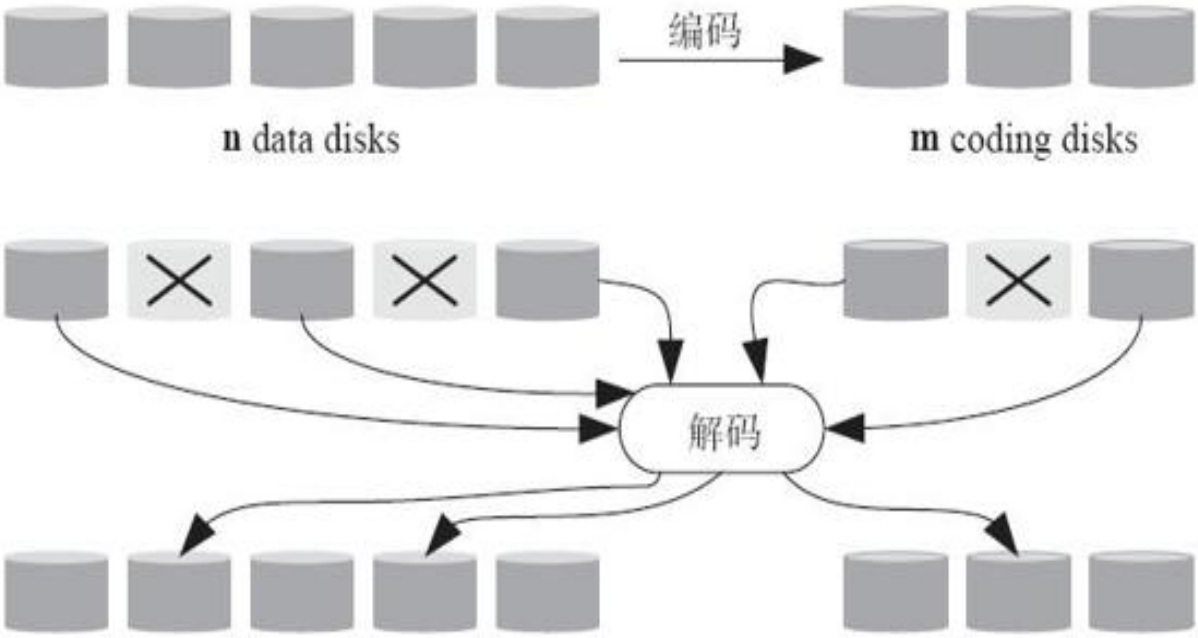



图10-6　高阶RAID实现图解

从数学上来说，上述问题可以转化为更一般的描述形式，即：如何基于n个可变输入，构造m个等式，使得对应的m元一次方程组有唯一解。Irving S.Reed和Gustave Solomon最早针对上述问题进行了研究，其成果是产生了一系列具有普遍意义、广泛应用于通信领域纠正信息传输过程中静默数据错误的Reed-Solomon codes。1997年，James S.Plank将Reed-Solomon codes引入存储系统，用于实现高阶RAID，对应的RAID技术也称为RS-RAID。

简言之，RS-RAID具体实现包含如下3个方面：

·基于范德蒙德矩阵计算校验和。

·基于高斯消元法进行数据恢复。

·基于伽罗华域执行编解码过程中所要求的算术运算。

我们接下来分别予以介绍。

### 10.2.1 计算校验和

假定当前有n个数据块，分别为$d_1$，$d_2$，…，$d_n$，定义$F_i$为所有数据块的组合方式，则根据$F_i$可以计算得到校验块$c_i$(i=1，2，…，m)。

$$c_i = F_i(d_1, d_2, \cdots, d_n) = \sum_{j=1}^{n} d_j f_{i,j}$$

换言之，如果我们使用向量D和C分别表示所有数据块和校验块的集合，$F_i$表示矩阵F中的每一行，则整个编码过程可以采用如下等式表示：

FD=C

因此，F也称为编码矩阵。RS-RAID使用m×n范德蒙德矩阵充当编码矩阵F，即有：$f_{i,j}=j^{i-1}$，于是上述等式变为：

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,1} & \cdots & f_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ f_{m,1} & f_{m,2} & \cdots & f_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & n \\ \vdots & \vdots & \vdots & \vdots \\ 1^{m-1} & 2^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

## 10.2.2 数据恢复

为了进行数据恢复，我们定义矩阵A和向量E，满足$A=\begin{bmatrix} I \\ F \end{bmatrix}$（I为单位矩阵）并且$E=\begin{bmatrix} D \\ C \end{bmatrix}$，于是有如下等式成立（AD=E）：

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & n \\ \vdots & \vdots & \vdots & \vdots \\ 1^{m-1} & 2^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

此时对于任意一个块（包括数据块和校验块），矩阵A和向量E都有一行与之对应，所以A也被称为分布矩阵。假定某个数据块失效，则相应地，我们将其对应的行从矩阵A与向量E中删除，于是得到一个新的等式：

A'=DE'

如果恰好有m个数据块失效（即向量D中存在m个未知数），那么可知，此时A'为一个n×n矩阵。因为F是范德蒙德矩阵，所以分布矩阵A的任意n行都是线性独立的，即A为非奇异矩阵。进而可知A'是可逆的，于是D中所有未知数都可以通过高斯消元法求解，即所有数据块都可以被修复。

### 10.2.3 算术运算

然而实际上，无法将上述理论直接应用于计算机来实现RS-RAID，这主要是因为数据恢复所依赖的高斯消元法需要用到除法运算，因此这个解虽然在实数域一定存在，但是由于计算机只能实现有限精度的除法运算，所以会导致该方法失效。另一个潜在的问题在于普通乘法运算容易产生进位，这使得存储校验块通常情况下要花费比任一数据块更多的空间，也违背了RS-RAID以消耗计算能力来降低存储空间成本的初衷。

考虑到计算机基于二进制，并且只能进行有限精度运算的特点，我们定义一个整型集合：

---

$$\{0, 1, 2, \ldots, 2^w - 1\}$$

---

显然，上述集合一共包含$2^w$个元素，例如当w=8时，该集合即对应计算机1字节所能存储的整数范围。同时，我们定义一套基于该集合的运算法则，该运算法则只包含加减乘除四则基本运算（高斯消元法只需要用到这4种基本运算），并且在集合内封闭。我们将满足上述约束条件的集合称为伽罗华域（Galois Field），也称为有限域（顾名思义，域（集合）中所包含的元素个数是有限的），其数学表达形式为$GF(2^w)$。

在RS-RAID的实现中，$GF(2^w)$中的加法运算非常简单，就是异或运算。因为加法和减法互为逆运算，而异或运算的逆运算为自身，所以$GF(2^w)$中的减法运算与加法运算相同，也是异或运算。以GF(2)={0，1}为例，容易验证（x表示未知数）：

0^x=0=>x=0^0=0

0^x=1=>x=1^0=1

1^x=0=>x=0^1=1

1^x=1=>x=1^1=0

可见，异或运算及其逆运算在GF(2)内是封闭的。

为了推导$GF(2^w)$中乘法运算的一般形式，我们首先定义基于GF(2)的多项式基本运算规则：

·多项式系数全部来自GF(2)（即只能取0或者1）。

·多项式中次数相同的项，可以基于GF(2)中的加法（即异或运算）进行合并。

例如假定：

r(x)=x+1

s(x)=x

则：

r(x)+s(x)=x+1+x=(1+1)x+1=(1^1)x+1=0+1=0^1=1

基于上述规则，我们定义多项式的模运算规则如下。

如果r(x)=q(x)t(x)+s(x)，其中：

s(x)和t(x)为任意多项式并且s(x)的次数小于q(x)

则：

r(x)mod q(x)=s(x)

例如假定：

$r(x)=x^2+x$

$q(x)=x^2+1$

因为：

$r(x)=(x^2+x)=(x^2+x)+1+1=x^2+1+x+1=q(x)+x+1$

所以：

r(x)mod q(x)=x+1

进一步地，如果多项式q(x)满足：

·次数为w

·所有项的系数都来自GF(2)

·不能被因式分解

那么可以基于q(x)生成$GF(2^w)$中的所有元素，方法如下：

1）前3个元素固定为0，1，x；

2）将前一个元素乘以x，然后针对q(x)取模；

3）重复步骤2），直至最终结果为1。

因此，满足上述条件的多项式q(x)也被称为$GF(2^w)$的生成多项式，记作：

$GF(2^w)=GF(2)[x]/q(x)$

常见的生成多项式如下：

w=4：$x^4+x+1$

w=8：$x^8+x^4+x^3+x^2+1$

w=16：$x^{16}+x^{12}+x^3+x+1$

w=32：$x^{32}+x^{22}+x^2+x+1$

w=64：$x^{64}+x^4+x^3+x+1$

例如，针对w=4，使用$q(x)=x^4+x+1$生成$GF(2^4)$中所有元素的过程如下：

0

1

x

$x^2=x\cdot x$

$x^3=x\cdot x^2$

$x^4=x\cdot x^3=x^4+x+1+x+1=q(x)+x+1=x+1$

$x^5=x\cdot x^4=x(x+1)=x^2+x$

$x^6=x\cdot x^5=x(x^2+x)=x^3+x^2$

$x^7=x\cdot x^6=x(x^3+x^2)=x^4+x+1+x^3+x+1=q(x)+x^3+x+1=x^3+x+1$

$x^8=x\cdot x^7=x(x^3+x+1)=x^4+x+1+x^2+1=q(x)+x^2+1=x^2+1$

$x^9=x\cdot x^8=x(x^2+1)=x^3+x$

$x^{10}=x\cdot x^9=x(x^3+x)=x^4+x+1+x^2+x+1=q(x)+x^2+x+1=x^2+x+1$

$x^{11}=x\cdot x^{10}=x(x^2+x+1)=x^3+x^2+x$

$x^{12}=x\cdot x^{11}=x(x^3+x^2+x)=q(x)+x^3+x^2+x+1=x^3+x^2+x+1$

$x^{13}=x\cdot x^{12}=x(x^3+x^2+x+1)=q(x)+x^3+x^2+1=x^3+x^2+1$

$x^{14}=x\cdot x^{13}=x(x^3+x^2+1)=x^4+x+1+x^3+1=q(x)+x^3+1=x^3+1$

$x^{15}=x\cdot x^{14}=x(x^3+1)=x^4+x+1+1=q(x)+1=1$

为了在RS-RAID中使用$GF(2^w)$，需要将$GF(2^w)$中的每个元素与一个w位的二进制数对应起来，为此只需要将上述多项式表示中每个$x^i(i\in(0，1，2，…，w-1))$项的系数和对应二进制数的第i个比特对应起来即可。表10-1汇总了$GF(2^4)$中元素的各种表示方式及其对应关系。

表10-1　$GF(2^4)$中所有元素及其不同表示方式

| 原生多项式 | 多项式表示 | 二进制表示 | 十进制表示 |
|---|---|---|---|
| 0 | 0 | 0000 | 0 |
| $x^0$ | 1 | 0001 | 1 |
| $x^1$ | $x$ | 0010 | 2 |
| $x^2$ | $x^2$ | 0100 | 4 |
| $x^3$ | $x^3$ | 1000 | 8 |
| $x^4$ | $x+1$ | 0011 | 3 |
| $x^5$ | $x^2+x$ | 0110 | 6 |
| $x^6$ | $x^3+x^2$ | 1100 | 12 |
| $x^7$ | $x^3+x+1$ | 1011 | 11 |
| $x^8$ | $x^2+1$ | 0101 | 5 |
| $x^9$ | $x^3+x$ | 1010 | 10 |
| $x^{10}$ | $x^2+x+1$ | 0111 | 7 |
| $x^{11}$ | $x^3+x^2+x$ | 1110 | 14 |
| $x^{12}$ | $x^3+x^2+x+1$ | 1111 | 15 |
| $x^{13}$ | $x^3+x^2+1$ | 1101 | 13 |
| $x^{14}$ | $x^3+1$ | 1001 | 9 |
| $x^{15}$ | 1 | 0001 | 1 |

据此，我们可以定义基于$GF(2^w)$的乘法运算规则：

1）将被乘数和乘数的二进制（十进制）表示转换为其对应的多项式表示。

2）执行多项式乘法，将得到的结果针对q(x)取模。

3）将步骤2）得到的结果转换为其对应的二进制（十进制）表示。

例如：

$3\times7=(x+1)\times(x^2+x+1)=x^3+x^2+x+x^2+x+1=x^3+1=9$

$13\times10=(x^3+x^2+1)\times(x^3+x)=(x^2+x+1)\times(x^4+x+1)+x^3+x+1=11$

或者更简单地，根据表10-1我们有：

$3\times7=x^4\times x^{10}=x^{14}=9$

$13\times10=x^{13}\times x^9=x^{22}=x^{15}\times x^7=x^7=11$

由对数运算性质：

$$log_xMN = log_xM + log_xN$$

$$log_x\frac{M}{N} = log_xM - log_xN$$

可见，引入对数运算可以将普通乘法、除法运算转化为加法、减法运算。因此，如果以gflog和gfilog分别表示GF($2^w$)中的对数运算和其逆运算（易见，GF($2^w$)中的元素不连续，所以gflog是一种离散对数），则其乘法运算和除法运算可以简化为（其中mod($2^w$-1)等价于对生成多项式q(x)取模）：

$$MN = gfilog\left(gflog(NN)\right) = gfilog\left(\left(gflog(M)+gflog(N)\right)\bmod\left(2^w-1\right)\right)$$

$$\frac{M}{N} = gfilog\left(gflog\left(\frac{M}{N}\right)\right) = gfilog\left(\left(gflog(M)-gflog(N)\right)\bmod\left(2^w-1\right)\right)$$

因此，如果能够预先计算出GF($2^w$)中所有元素的gflog和gfilog表，就可以基于这两张表快速执行域中任意两个元素的乘法和除法运算。注意，GF($2^w$)的每个元素都由原生多项式$x^i$(i(0，1，2，…，w-1))得来。又因为：

$$log_xx^i = i$$

所以：

$$gflog(x^i) = i$$

同时有：

$$gfilog(i) = x^i$$

例如，针对GF($2^4$)，结合表10-1我们有：

$$gflog(x^0) = gflog(1) = 0 \quad gfilog(0) = x^0 =$$

$$gflog(x^1) = gflog(2) = 1 \quad gfilog(1) = x^1 =$$

$$gflog(x^2) = gflog(4) = 2 \quad gfilog(2) = x^2 =$$

$$gflog(x^3) = gflog(8) = 3 \quad gfilog(3) = x^3 =$$

$$\cdots\cdots$$

表10-2展示了据此计算得到的完整gflog和gfilog表。

表10-2　GF($2^4$)对应的gflog和gfilog表

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $gflog[i]$ | - | 0 | 1 | 4 | 2 | 8 | 5 | 10 | 3 | 14 | 9 | 7 | 6 | 13 | 11 | 12 |
| $gfilog[i]$ | 1 | 2 | 4 | 8 | 3 | 6 | 12 | 11 | 5 | 10 | 7 | 14 | 15 | 13 | 9 | - |

根据表10-2我们可以方便地进行GF($2^4$)中任意两个元素的乘法和除法运算。例如：

$$3 \times 7 = gfilog(gflog(3)+gflog(7)) = gfilog(4+10) = gfilog(14) = 9$$

$$13 \times 10 = gfilog(gflog(13)+gflog(10)) = gfilog(13+9) = gfilog(7) = 11$$

$$3 \times 7 = gfilog(gflog(3)-gflog(7)) = gfilog(4-10) = gfilog(9) = 10$$

$$13 \times 10 = gfilog(gflog(13)-gflog(10)) = gfilog(13-9) = gfilog(4) = 3$$

至此，我们已经完整实现了基于$GF(2^w)$的四则运算，据此可以完成RS-RAID编解码过程中所需的全部算术运算。

## 10.2.4 缺陷与改进

上述基于范德蒙德矩阵和GF($2^w$)有限域的RS-RAID理论研究公开发表在1997年的A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems一文当中。6年之后，James发现了原文中的一个致命缺陷，即分布矩阵A在GF($2^w$)并不具备像其所宣称的那样具有“删除任意m行所得到n×n子矩阵仍然可逆”的特性，并通过Note Correction to the 1997 Tutorial on Reed-Solomon Coding一文进行了修正。修正后的分布矩阵（为了进行区分，称为分布矩阵B）总是可以通过如下的(n+m)×n范德蒙德矩阵经过有限步的初等变换，将前n行转化为单位矩阵之后得到。

$$\begin{bmatrix} 0^0(=0) & 0^1(=0) & \cdots & 0^{n-1}(=0) \\ 1^0 & 1^1 & \cdots & 1^{n-1} \\ 2^0 & 2^1 & \cdots & 2^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ (n+m-1)^0 & (n+m-1)^1 & \cdots & (n+m-1)^{n-1} \end{bmatrix}$$

可以证明，上述矩阵删除任意m行都是可逆的。又因为初等变换不改变矩阵的秩，所以由此得到的分布矩阵B仍然具有此特性。

然而，直接采用范德蒙德矩阵作为编码矩阵的代价在于编解码复杂度太高，尤其是当n和m比较大时，编解码时延一般无法满足生产环境的苛刻要求。1995年，Blomer等人在An XOR-Based Erasure-Resilient Coding Scheme一文中介绍了一种改进方案，其思路主要集中在两个方面：一是通过引入位矩阵（bit matrix），将编码过程中的乘法运算进一步转化为异或运算，因此生成校验块的速度更快（实际上这还取决于所选取的编码矩阵）；二是使用性质更好的柯西矩阵取代范德蒙德矩阵作为编码矩阵（可以证明，可用作编码矩阵的柯西矩阵在$GF(2^w)$中不是唯一的），因为范德蒙德矩阵求逆运算的时间复杂度为$O(n^3)$，而柯西矩阵求逆运算的时间复杂度仅为$O(n^2)$，所以采用柯西矩阵，理论上解码速度可以提升一个数量级。

2005年，James通过进一步研究发现，上述柯西矩阵的选择会对编码性能造成显著影响，“好”的柯西矩阵与“坏”的柯西矩阵导致的性能差异平均在10%之间，极端情况下可达83%。与此同时他也给出了生成一个“好”的柯西矩阵的一般性算法。上述研究结论发表在Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Storage Applications一文中，其中有优化前后详细的性能对比数据，这里不赘述。

### 10.2.5 Jerasure

2007年，James基于上述理论研究给出了RS-RAID的一个开源实现，称为Jerasure。

Jerasure所实现的纠删码是水平方式的，即需要同时使用k+m块不同的磁盘分别承载数据和校验数据（对应的磁盘分别称为数据盘和校验盘）。如果对应的存储系统能够容忍任意m块磁盘同时故障，那么称此时采用的纠删码是MDS（Maximum Distance Separable）类型的。

除上述两个参数之外，Jerasure库所实现的大部分纠删码还用到了一个额外参数w，称为字长，即一次编码的长度。例如假定w=8，则每次至多完成k字节原始数据的编码；w=16，则每次至多完成k×2字节原始数据的编码等。还有一些低阶纠删码，如RAID5，因为整个编解码过程中只用到异或运算，所以可以直接采用机器的天然字长（可以通过sizeof(long)得到）作为运算的基本单位。此时，每次编码的单位为包（packet），包大小（packetsize）为机器字长的整数倍即可。当然如果定义的包过大，导致编码时切分到某些数据盘上的数据不足一个包，由异或运算的性质，此时不足部分需要使用全0填充。表10-3汇总了Jerasure中的常见参数及其含义。

表10-3　Jerasure标准库中的常见参数及其含义

| 参数名称 | 含　义 |
| --- | --- |
| k | 数据盘个数 |
| m | 校验盘个数 |
| w | 字长 |
| packetsize | 包大小。<br>经过 James 验证，包大小（以及编码过程中使用的缓存大小）对编码性能有巨大影响[⊖]，需要根据具体业务场景进行测试和选择 |
| size | 每个盘待编码或者解码的字节数，必须是机器字长（sizeof(long)）的整数倍（例如，针对一个大文件进行编码，则此文件会被均匀切分成 $k$ 份，每份大小即为 size）。<br>如果使用位矩阵，要求 size 是 packetsize × $w$ 的整数倍。<br>不足此长度时，不足部分需要使用全 0 进行填充 |
| matrix | 编码矩阵。<br>所有元素只能从 $GF(2^w)$ 选取 |
| bitmatrix | 二进制编码矩阵，即将编码矩阵中每个元素都转化为对应的位矩阵表示。<br>矩阵中只包含 0 和 1 两种元素 |
| data ptrs | 二维数组。<br>包含 $k$ 个指针，每个指针指向长度为 size 的待编码数据 |
| coding ptrs | 二维数组。<br>包含 $m$ 个指针，每个指针指向长度为 size 的缓存空间，用于存放编码后的数据 |
| erasures | 一维数组，保存故障磁盘编号（磁盘编号范围为 0~ $k+m-1$）。<br>假定有 $e$ 个磁盘故障，则数组长度为 $e$+1，其中前 $e$ 个条目用于保存故障磁盘编号，最后一个元素满足 erasures[$e$] = = −1，用于标识数组结束 |
| erased | 一维数组，erasures 的另一种表达形式。<br>数组长度为 $k+m$，如果某个磁盘正常，则设置数组中对应元素的值为 0；否则设置为 1 |
| schedule | 二维数组，包含若干五元组，用于优化基于位矩阵的编码运算。<br>例如包含 $n$ 个五元组，则 schedule[n][0] == −1 标识数组结束 |
| cache | 三维数组，保存一系列缓存地址，用于对 RAID6 的解码过程进行优化 |
| row k ones | 布尔类型，用于对满足 $m>1$ 的解码过程进行优化。<br>解码时，如果对应编码矩阵第一行全部为 1 或者对应二进制编码矩阵前 $w$ 行构成 $k$ 个单位矩阵，则设置此标志有助于提升解码速度 |
| decoding matrix | 解码矩阵 |
| dm ids | 一维数组，用于指定仍然正常的磁盘编号，帮助生成解码矩阵 |

注：http://jerasure.org/jerasure-2.0/,11.1Judicious Selection of Buffer and Packet Sizes

值得一提的是，除了实现一般形式的RS-RAID，针对m=2，即RAID6这种业界目前使用最普遍同时也最具性价比的纠删码，Jerasure特别进行了两类优化：一类优化针对仍然采用范德蒙德作为编码矩阵的情形，由于此时编码矩阵中只包含1和2这两种元素，所以可以针对乘2运算进行优化，使其编码速度加快；另一类优化则是直接对编码矩阵进行改造，其结果是产生了一簇被称为最小密度RAID6（Minimal Density RAID6）的编码方法集。与柯西RS-RAID类似，最小密度RAID6也使用位矩阵作为编码矩阵，但是其构成元素（指转换为位矩阵之前的原始编码矩阵）没有必须来自于GF($2^w$)的限制，并且要求其中包含的非0元素尽可能少（反过来这意味着要求编码矩阵中的0尽可能多，即要求编码矩阵尽可能稀疏，这解释了这类编码方式为什么会被冠以“最小密度”的头衔），从而减少计算量，提升编解码性能。Jerasure目前支持3种类型的最小密度RAID6，分别是：

·Liberation，要求w必须是素数

·Blaum-Roth，要求w+1必须是素数

·Liber8tion，要求w必须等于8

三者的编解码效率相当，但是由于Liber8tion的w固定为1字节，所以相较Liberation和Blaum-Roth而言，有天然就是字节对齐的优势。

表10-4汇总了Jerasure目前所支持的编解码技术。

表10-4　Jerasure标准库（jerasure-2.0）所支持的编解码技术

| technique | 含　义 | technique | 含　义 |
| --- | --- | --- | --- |
| reed_sol_van | 基于范德蒙德矩阵的 RS-RAID | liberation | 最小密度 RAID6 |
| reed_sol_r6_op | 基于范德蒙德矩阵的 RAID6（优化） | blaum_roth | |
| cauchy_orig | 基于原生柯西矩阵的 RS-RAID | liber8tion | |
| cauchy_good | 基于最佳柯西矩阵的 RS-RAID | | |

## 10.3 纠删码在Ceph中的应用

Ceph支持多种类型的纠删码插件，例如Jerasure、ISA、LRC、SHEC等。

为了创建纠删码类型的存储池，首先需要指定纠删码的详细配置模板，该模板包含如表10-5所示的参数。

表10-5 erasure-code-profile

| 参数 | 说　明 |
| --- | --- |
| directory | 纠删码插件的加载路径，默认为/usr/lib/ceph/erasure-code |
| plugin | 用于指定所采用的纠删码插件，包含如下选项：<br>—jerasure（默认）<br>—lrc<br>—shec<br>—isa |
| key=value | 键值对，用于指定每种类型纠删码的具体配置参数，例如数据盘和校验盘个数、选用的编解码技术等。<br>不同类型的纠删码可以自定义键值对 |
| ruleset-failure-domain | 对应CRUSH模板中的故障域，例如为host，则要求纠删码的数据盘和校验盘分别位于不同主机之下 |
| --force | 如果携带，则覆盖集群中任何已经存在的同名纠删码模板 |

由于不同类型纠删码插件实现算法不尽相同，因此表10-5中的键值对部分（主要与具体算法相关）还需要根据官方文档进一步进行配置。例如Jerasure有如表10-6所示的可配置参数。

表10-6　Jerasure可配置参数（Jerasure-2.0）

| 参数 | 说　明 |
| --- | --- |
| k | 数据盘个数 |
| m | 校验盘个数 |
| technique | Jerasure当前所支持的编解码方式，默认为reed_sol_van |
| packetsize | 包大小，默认为2048，单位为字节 |

了解各个参数的含义之后，可以通过如下命令创建一个纠删码模板：

```
ceph osd erasure-code-profile set my-ec-profile plugin=jerasure
\
    k=4 m=2 technique=liber8tion ruleset-failure-domain=host
```

上述命令创建了一个采用liber8tion算法（注意：此时m必须为2）、故障域为主机级别（注意：因为k+m=6，此时必须有6台主机才能使得故障域配置正常生效）的纠删码模板。可以使用如下命令查看和确认：

```
ceph osd erasure-code-profile get my-ec-profile
k=4
m=2
packetsize=2048
plugin=jerasure
ruleset-failure-domain=host
ruleset-root=default
technique=liber8tion
w=8
```

最后，可以基于上述纠删码模板创建一个纠删码类型的存储池：

```
ceph osd pool create my-ec-pool 128 erasure my-ec-profile
```

纠删码存储池创建完成之后，客户端可以采用与多副本存储池相同的API来操作池中数据（对象），但是两者的实现细节存在较大差异，纠删码的读、写、数据恢复等流程要更加复杂。

### 10.3.1 术语

介绍Ceph的纠删码具体实现之前，先介绍几个与之相关的术语。

1.块（chunk）

将对象基于纠删码进行编码时，每次编码将产生若干大小相同的块（分为数据块和校验块，纠删码要求这些块是有序的，否则后续无法解码）。Ceph通过等量的PG副本将这些块分别存储至不同的OSD之中。每次编码时，序号相同的块总是由同一个副本负责存储。

2.条带（stripe）

如果对象太大导致无法一次完成编码，则可以分多次进行。每次编码的部分称为一个条带。同一个对象内的条带是有序的，按照生成条带的顺序从0开始编号。

3.分片（shard）

同一个对象中所有序号相同的块位于同一个副本上，它们组成对象的一个分片。分片的编号即块的序号。

块、条带以及分片之间的关系如图10-7所示。

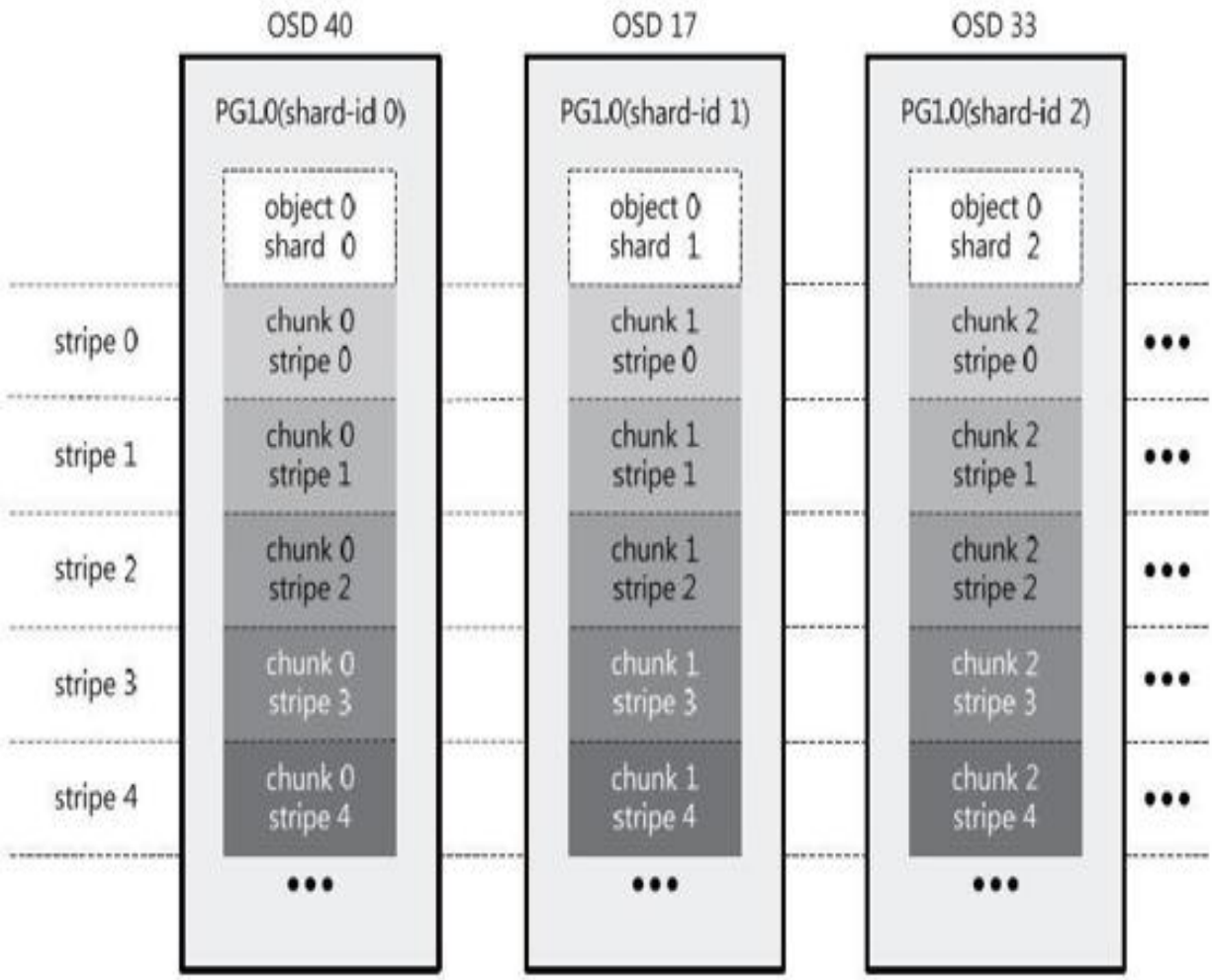



图10-7　块、条带、分片

4.k

条带中数据块的个数。

5.m

条带中校验块的个数。

在不考虑空间预留、元数据开销的情况下，纠删码的空间利用率可以通过k/(k+m)得到。例如k=9，m=3，则空间利用率为9/12=75%。可

见，在保持冗余度不变（即m不变，即不牺牲可靠性）的前提下，增大k值，可以有效提升纠删码存储池的空间利用率。

创建纠删码存储池时，每个PG会被转化为k+m个副本。与多副本除了Primary之外所有副本地位都相等不同，由于每个纠删码副本只保存每个对象的某个固定（序号的）分片，并且纠删码的解码过程要求输入的分片严格有序，所以需要对每个纠删码副本的身份进行严格区分，这通过向PGID中额外增加一个分片标识（shard-id）来实现。当然，出于数据一致性考虑，仍然需要从这k+m个副本中选出Primary，我们约定其总是由CRUSH返回的Up列表中第一个有效的OSD充当。

### 10.3.2 新写

了解上述基本概念之后，我们以读写过程为例来分析纠删码的具体实现。

首先，介绍最简单的情形，即向存储池中写入一个全新对象。为此，我们假定存在如下规格的纠删码存储池：k=3，m=2。图10-8展示了将一个名为NYAN的全新对象写入该存储池的过程。

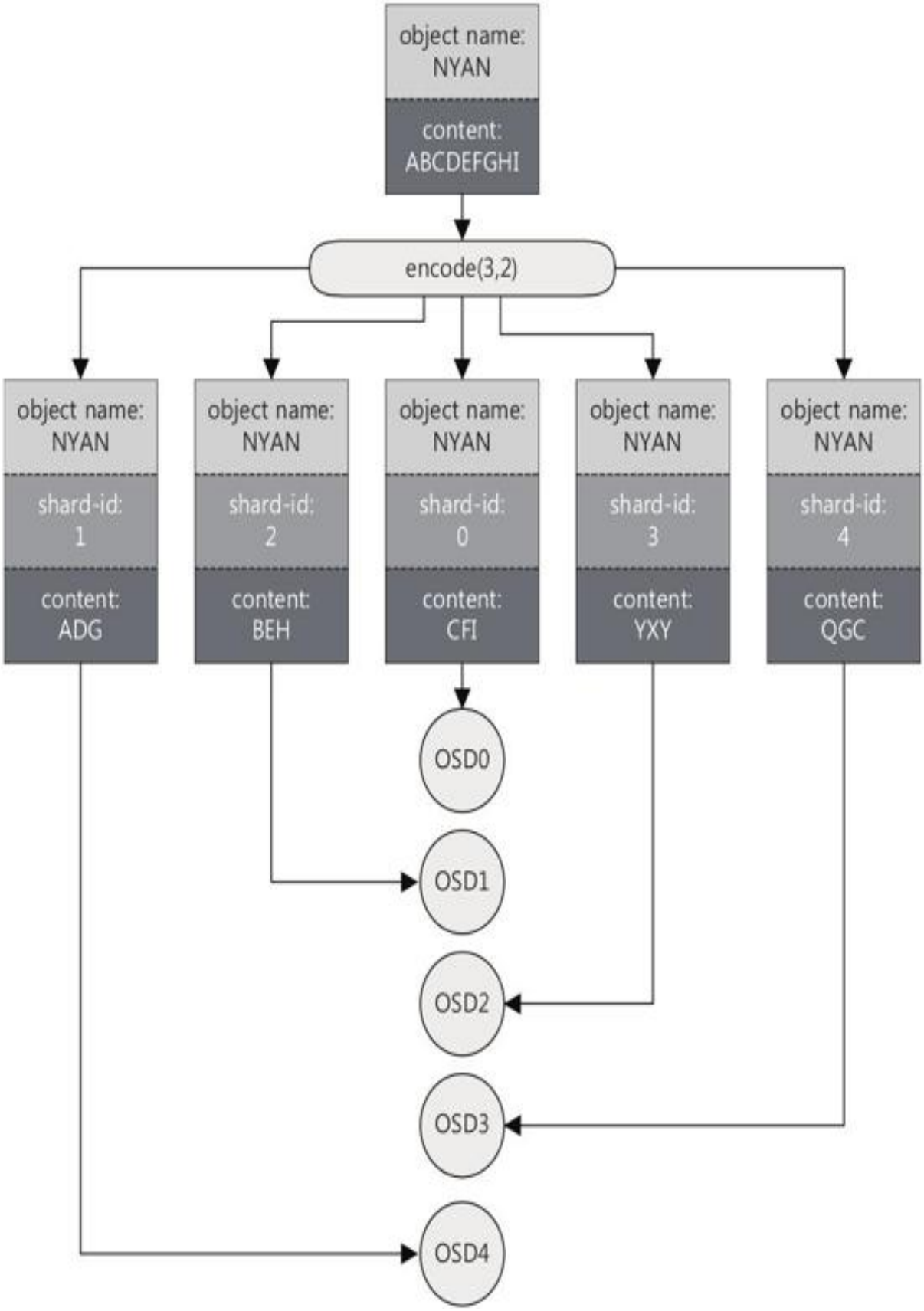

object name:
NYAN
content:
ABCDEFGHI
encode(3,2)
object name:
NYAN
shard-id:
1
content:
ADG
object name:
NYAN
shard-id:
2
content:
BEH
object name:
NYAN
shard-id:
0
content:
CFI
object name:
NYAN
shard-id:
3
content:
YXY
object name:
NYAN
shard-id:
4
content:
QGC
OSD0
OSD1
OSD2
OSD3
OSD4

图10-8　向纠删码存储池（k=3，m=2）写入一个全新对象

图10-8展示的是一种最简单的写请求，称为满条带写（Whole Stripe Write），即写入的数据范围正好覆盖对象的一个或多个完整条带。需要注意的是，将对象的每个条带切分为多个分片分别写入不同副本，以及后续从不同副本读取必需的分片还原出完整条带，都是由Primary独立完成的，其他副本并不感知。这意味着每个副本都认为自身保存的是一个完整而独立的对象。对象数据在逻辑上是连续的，总是以块（chunk）为单位，从0开始编址。仍以图10-8为例，假定块大小为1字节、条带大小为3字节（条带大小总是等于k*块大小），则5个OSD最终都向NYAN对象写入了3字节，它们在对象内的逻辑地址范围相同，都是[0，2]。

### 10.3.3 读

与满条带写类似，Primary收到客户端的读请求之后，首先将其携带的逻辑地址范围进行条带对齐，据此计算得到每个副本需要读取的数据范围，然后通知各个副本分别读取，最后再由Primary汇总并向客户端应答。

除此之外，针对读操作还有如下几个关键因素需要考虑。

#### （1）是否进行条带对齐

理论上读操作并不需要条带对齐，只要求每个副本能够读取指定范围的数据即可。例如针对图10-8，假定块大小变为1kB，待读取的数据范围为[512，2560)，则正常情况下的读请求序列为：

```
PG(shard-id = 0)：[512, 1024)
PG(shard-id = 1)：[0, 1024)
PG(shard-id = 2)：[0, 512)
```

假定PG从ObjectStore读取数据的最小粒度为一个扇区，可见此时不进行条带对齐可以使得开始和最后一个副本有概率少读部分数据。

然而遗憾的是，通常情况下并不会将条带设置得很大，这导致条带中的块（chunk）大小经常会小于磁盘的最小访问粒度，因此实际上

不进行条带对齐能够取得额外收益的情形非常有限。同时，如果PG处于降级状态（此时客户端仍然可以正常进行读写）或者Recovery触发的读操作，都有可能导致需要读取校验块执行解码后才能得到原始数据，而这个过程（即解码）必须以条带为单位进行。

综合考虑上述因素，在读操作开始之前，由Primary预先执行条带对齐是合适的，因此上面这个例子实际产生的读请求序列为：

```
PG(shard-id = 0)：[0, 1024)
PG(shard-id = 1)：[0, 1024)
PG(shard-id = 2)：[0, 1024)
```

### （2）是否需要对校验块的位置进行调整

如前所述，由于正常情况下并不需要读取校验块，所以RAID5需要对条带中校验块的位置进行动态调整，避免使用固定校验盘造成读带宽浪费。在Ceph的实现中，因为（正常情况下）副本并不会在OSD之间迁移，这意味着对同一个PG而言，其数据盘和校验盘的位置几乎总是固定的。为什么Ceph可以采用这种方式而不必担心浪费读带宽呢？原因在于，整个集群存在大量PG副本并且每个副本的分布是随机的，此时每个OSD既是一些PG的数据盘，也是另外一些PG的校验盘，因此即使不改变条带中校验块的位置，只要来自客户端的压力足够分散（因为随机分布数据，这个条件一般是满足的），便不会造成读带宽浪费。

### （3）是否仍然采用同步读

多副本模式下，由于每个副本保存的内容完全相同，并且由Primary负责保证数据一致性，因此所有读请求可以由Primary直接在本地完成，这个过程是同步的。纠删码则不然，由于每个对象的数据都

以分片的形式分布在多个副本之上，并且这些副本由位于不同故障域下的OSD承载，这导致客户端的单个读请求经常会被转化为跨节点的多个读请求。出于性能考虑，纠删码无法继续沿用多副本同步读的方式而必须采用异步读。

异步读虽然提高了读请求之间的并发度，有助于从整体上提升纠删码存储池的读带宽。但是由于涉及跨节点交互，所以原理上纠删码存储池的读操作响应时延要比多副本存储池高，这使得纠删码存储池不太适合时延敏感类的应用。

### （4）是否自动进行数据修复

纠删码的实现原理表明，当条带中损坏的块（包括数据块和校验块）个数不大于m时，总是可以通过解码还原出所有损坏的块。因此，如果处理客户端读请求时，Primary检测到条带中某些数据块产生了错误（例如静默数据错误），也可以（顺便）对这些坏块执行修复。这个过程称为自动修复（auto-repair）。

显然，启用自动修复的前提是OSD能够捕获数据错误。Luminous版本之前，由于FileStore不具备数据自校验功能，因此也就无法启用自动修复功能。Luminous及之后的版本，虽然FileStore的替代者BlueStore已经具备数据自校验功能，但是为了避免继续恶化纠删码的读性能（自动修复会进一步增加读请求的响应时延），自动修复是作为Scrub的一个附加功能实现的。

### 10.3.4 覆盖写

基于上述基本读写流程，我们可以着手来分析一种更复杂的情形——覆盖写。

顾名思义，覆盖写是针对对象的已有内容进行改写。对多副本而言，PG处理新写和覆盖写并无区别。纠删码则不然，通过引入条带的概念，纠删码将条带变成了更新对象数据的最小单位。因此如果覆盖写的起始或者结束地址没有进行条带对齐，那么对于不足一个完整条带的部分，其写入只能通过“读取完整条带→修改数据→基于新条带重新计算校验块→写入被修改过的数据块与校验块”这样的步骤来进行，即RMW。

和BlueStore中的RMW操作类似，RMW中的R，即读是为了与新的写入内容一起，拼凑出一个完整条带，以完成纠删码所要求的重新编码计算（目的是更新校验块），因此这个读也称为补齐读。容易理解，整个RMW操作中补齐读阶段最为耗时，因此为了提升覆盖写的性能，通常有两种思路：一是尽量减少RMW操作的数量；二是如果RMW操作不可避免，则需要尽量减少补齐读阶段读取的数据量。

引入写缓存是减少RMW操作数量的一种常见方法。仍以图10-8为例，假定块大小为1kB，条带大小为3kB，并且存在如下的覆盖写序列：

```
write1:[0,4096)
write2:[4096,8192)
write3:[8192,12288)
…
```

---

如果不做任何处理，那么上述序列中的每3个原始写请求都将被转化为2个满条带写和4个RMW写，如图10-9所示。

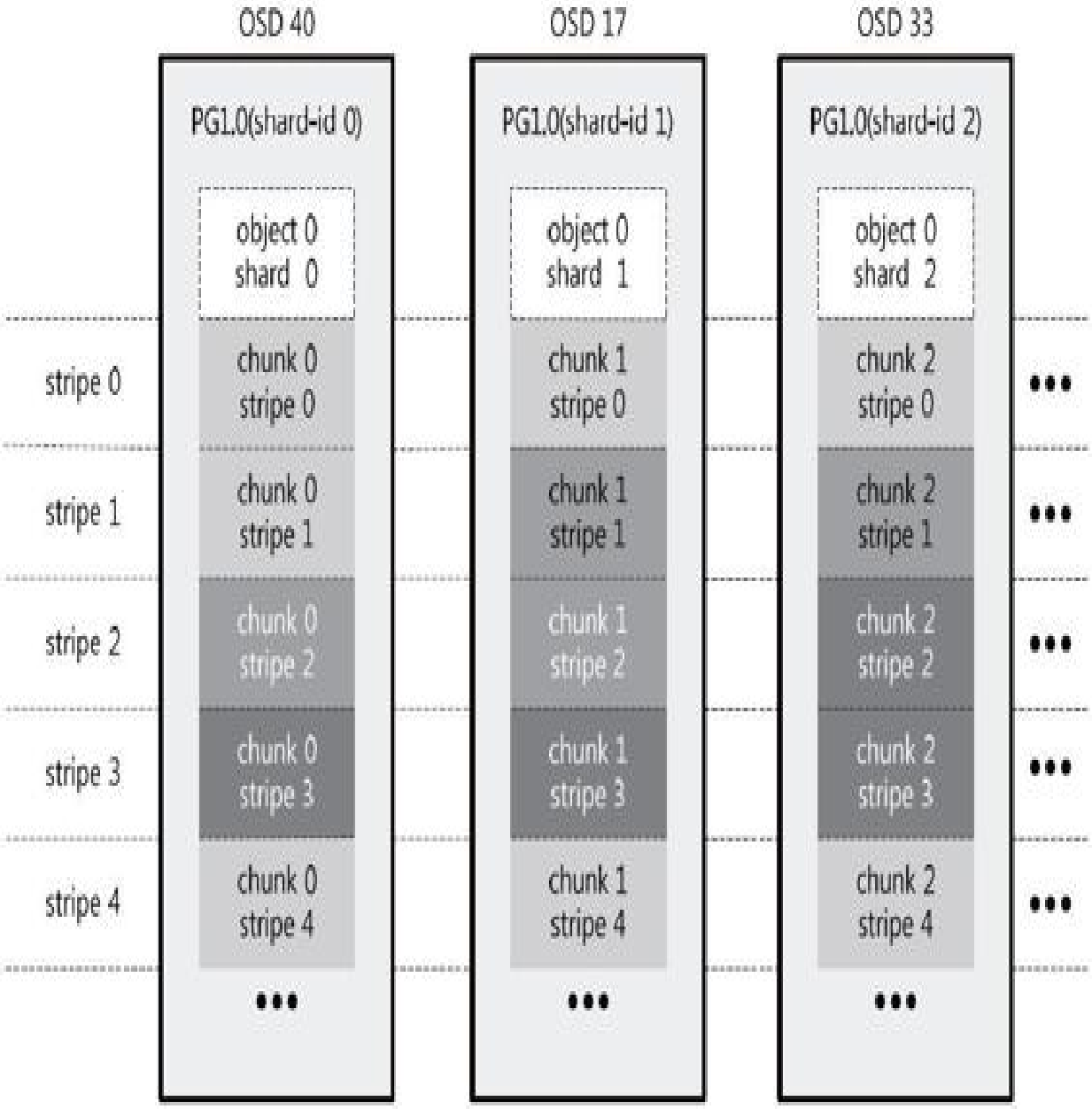


图10-9　覆盖写——引入缓存前

图中块大小为1kB，条带大小为3kB，这里忽略了校验块的处理

反之，如果引入缓存，通过对所有仍然驻留在缓存中的写请求执行合并，我们可以有效减少RMW操作的数量。例如，假定每个写请求在缓存中驻留的时间足够长，则上面这个例子中类似write1、write2、write3这样3个写请求最终会被聚合成为一个单独的写请求：

```
write:[0, 12288)
```

从而通过4次满条带写完成，如图10-10所示。

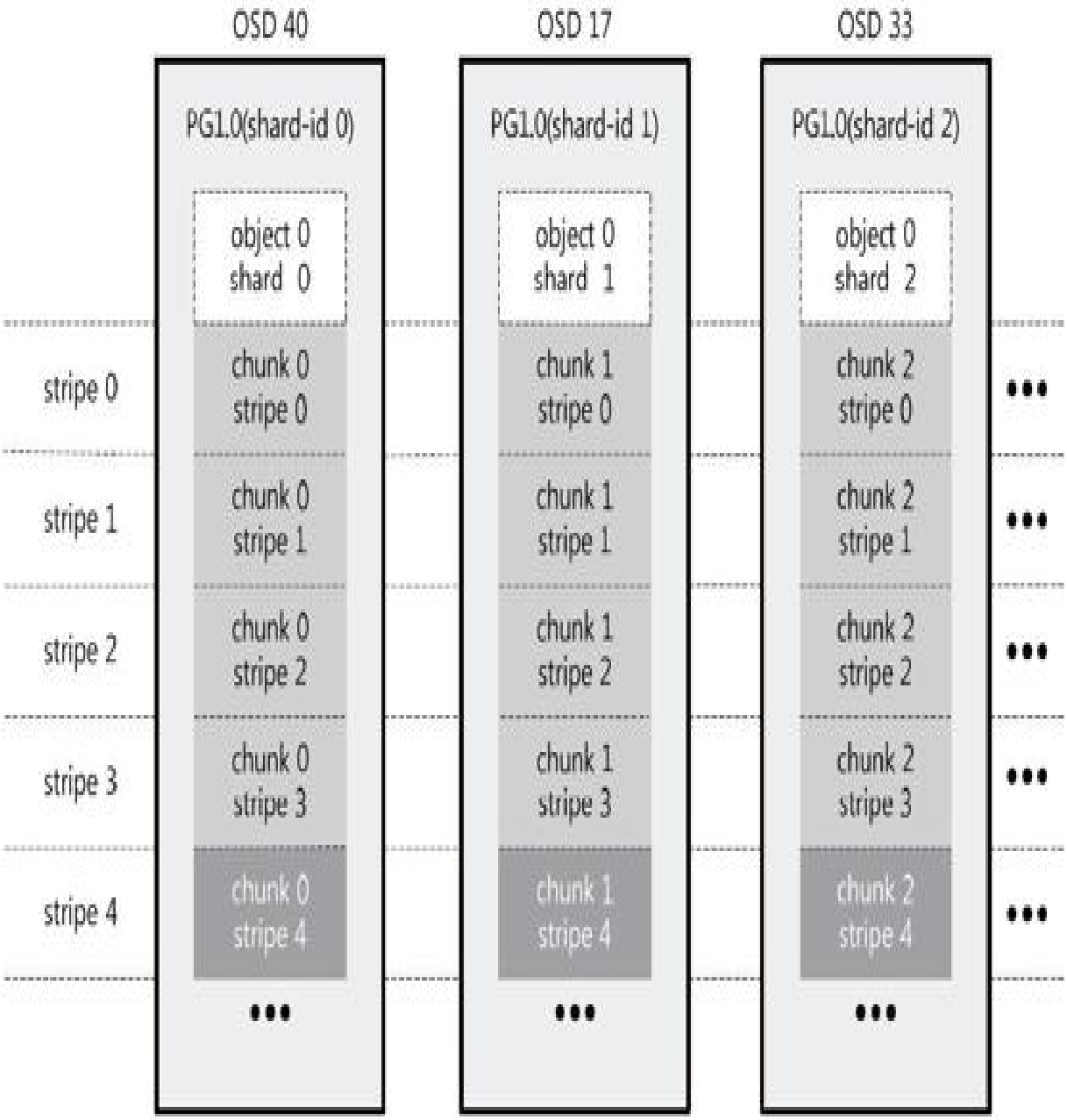



图10-10　覆盖写——引入缓存后

图中块大小为1kB，条带大小为3kB，这里忽略了校验块的处理。

由于不同对象的写请求无法进行合并，纠删码中的写缓存被设计成对象级别。引入写缓存的难度在于如何合理地控制将缓存中的数据批量同步至磁盘的时机。常见的思路是使用定时器驱动，或者基于缓

存容量设定合适的阈值（也称为水位）进行驱动。当然也可以综合使用这两种方法。

引入写缓存后，虽然通常情况下对提升覆盖写性能有所帮助，但是由于存在写请求合并，相应地会使得写请求的平均时延增加。

如果RMW操作不可避免（例如完全随机写，那么即便引入写缓存也无济于事），也需要尽最大可能减少补齐读的数据量。通常的做法是，基于被改写的数据范围预先计算出需要执行补齐读的块，而不是每次都执行满条带读。例如，对应如下的覆盖写请求（假定块大小为1kB，条带大小为3kB）：

```
write:[0, 2345)
```

显然条带中编号为0、1的两个块已经包含在写请求范围之内（后续将被新的内容完全覆盖），这种情况下仅需要针对编号为2的数据块执行补齐读。

### 10.3.5 日志

与多副本类似，纠删码也使用PG级别的日志来保证副本之间的数据一致性。例如，当系统掉电时，如果还有客户端的写请求没有处理完成，则下次上电时，或者通过日志将数据回滚至写之前的状态（对应尚未向客户端发送过写入完成应答的场景），或者通过日志重放继续执行（也称为前滚（rollforward），对应已经向客户端发送过写入完成应答的场景）。

对多副本而言，由于每个副本保存的内容完全相同，这意味着只要任意一个副本完成数据更新，后续就可以基于这个副本将其他副本的数据也同步至最新。所以基于日志执行数据恢复时，我们总是选择包含最新日志的那个副本，然后以此作为基准，通过日志前滚操作即可完成所有降级对象的修复。

纠删码则不然，当覆盖写发生时，由于每个副本保存的都是原始对象独一无二的内容（即分片），为了防止将原始对象写坏，每个副本在执行覆盖写之前必须先通过克隆操作进行备份。日志系统需要如实记录上述数据备份过程，以便系统从异常中恢复时，如果判定某些尚未向客户端应答的写请求无法通过前滚操作继续执行，则将其操作的对象回滚至前一个合适的版本（即备份）来确保一致性。

### 10.3.6 Scrub

针对纠删码存储池，Scrub有两种可选实现方案：

一种方案是由Primary负责生成与保存原始对象的校验和。执行Scrub时，由Primary负责读取原始对象的全部内容并重新计算校验和，再与之前保存的校验和进行比对。参考Scrub章节的分析，这个方案仅适用于客户端从不执行覆盖写的场景（否则需要读取对象的全部内容以重新生成校验和），因此应用场景受限。

另一种方案与多副本类似，即每个副本都只负责保证自身内容的正确性。由于缺乏参照物（多副本模式下每个副本保存的都是完整的原始对象，可以互为参照物），无法像多副本那样执行分布式一致性校验，此时可以通过重算与比较每个条带中的校验块来进行验证。

由于支持覆盖写，所以目前采用了后面这种方案，但是弱化了分布式一致性校验，例如，不会针对每个条带的数据执行校验。

## 10.4 总结和展望

作为一种早在20世纪90年代就已经在存储系统普及的数据保护技术，纠删码并非是一种新生事物。纠删码以其灵活多变的数据备份策略（理论上可以支持任意数量的备份）、较高的存储空间收益深受“廉价”存储系统的青睐，非常适合于存储大量对时延不敏感的“冷”数据（例如备份数据）。

Ceph最初的设计中，并未考虑支持纠删码。然而由于随机分布数据的特性，加上多副本本身居高不下的备份数据量，使得Ceph的存储空间利用率一直为人诟病。

自Emperor版本起，Ceph引入了一种全新的存储池类型——纠删码存储池。顾名思义，这种存储池使用纠删码替代多副本作为数据备份策略。社区希望借助纠删码能够以增加计算资源消耗为代价，换取更高存储空间收益的能力，进一步提升Ceph作为分布式统一存储解决方案的性价比。

然而事与愿违，这种被寄予厚望的新型存储池迟迟未能达到商用水准。究其原因，无外乎有以下几个：

1）相较多副本而言，纠删码实现更复杂。

纠删码以条带为单位，通过数学变换，将采用任意k+m备份策略所消耗的额外存储空间都成功地控制在1倍以内（与之相比，具有同等安全系数的多副本备份策略则要消耗m倍的额外存储空间），代价是大大增加了系统的复杂性。

对Ceph而言，与多副本类似，纠删码条带中的每个块都需要通过副本存储至位于不同故障域中的OSD来保证数据可靠性。这意味着在纠删码存储池中，为了获得同等的可靠性，通常情况下每个PG的副本数量要比多副本存储池多，从而对现行的数据同步与一致性机制提出了更大的挑战。

2）相较多副本而言，纠删码性能更差。

在存储系统中，读性能的表现至关重要。

采用C/S访问模式，使得Ceph的I/O路径相较传统存储而言本来就偏长，而其分布式天性进一步地使得任何客户端的读都会被纠删码转化为集群内多个跨节点的读，并且产生跨节点读的个数与k值成正比。跨节点的传输时延与协同消耗使得纠删码的读性能必然要比同等条件下的多副本差，再加上生产环境中为了取得更高的空间收益（对应更大的k值）和更高的安全系数（对应更大的m值，因为纠删码要求k≥m，所以m值增加的同时也意味着要求配置更高的k值）往往要求我们配置更大的k值和m值，这反过来又会使读性能变得更差。所以，一般而言纠删码的读性能（例如I/O平均时延）很少能满足生产环境的要求。

此外，纠删码在设计上使用条带作为数据更新的最小单位，出于编解码效率，我们一般不会将条带设计得很小，因此针对小块随机覆盖写的场景，纠删码非常容易产生大量RMW操作，这使得纠删码的综合写性能也比多副本要差。

也许纠删码最适合的应用场景是永远只进行追加写（或者删除），这也是早期社区开发者的思路。因此在纠删码存储池的早期版本中，一切覆盖写行为都是被禁止的，而是需要基于Cache-Tier技术，额外建立一个全SSD缓存池用于加速和过渡。

3）纠删码不支持omap。

Ceph通过omap为前端应用程序提供了一种存储海量或者超大型键值对的方法，例如RGW可以利用单个对象的omap记录多达10万条索引。与对象的数据部分不同，针对omap当前并没有很好的编码方法，所以纠删码存储池不支持访问对象的omap。这意味着类似RBD、RGW这类必须使用omap的典型存储应用，为了使用纠删码存储池，还必须额外创建一个多副本存储池。因此，与多副本存储池相比，纠删码存储池无论是使用还是维护都极为不便。

尽管前路漫漫，但是自Jewel之后，社区还是为纠删码存储池增加和完善了覆盖写支持，这是自引入纠删码以来社区迈出的具有非凡意义的一步，表明了社区后续坚定不移推出真正能够满足生产环境级别的纠删码解决方案的决心。

我们期待并将一起为之努力。