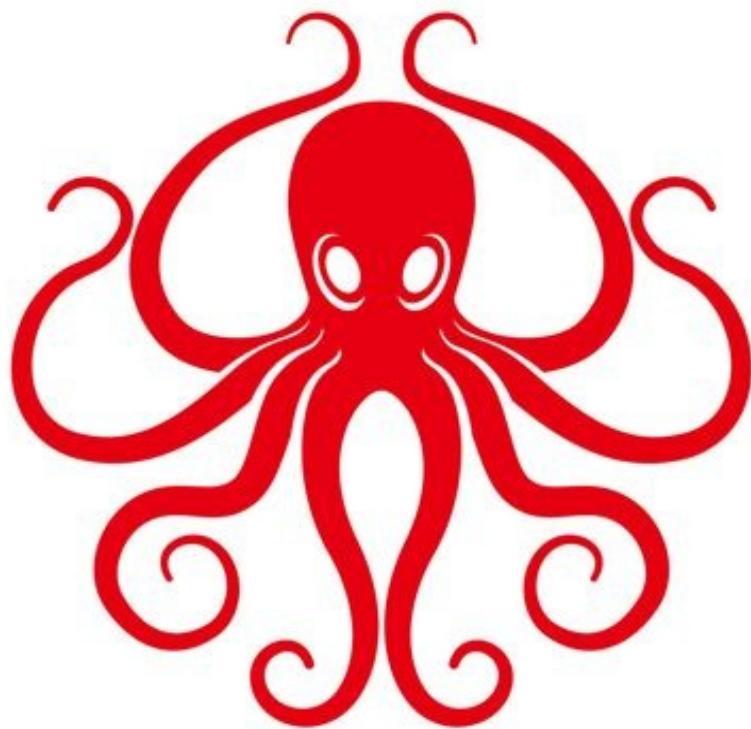




从源码角度全面解析Ceph的整体框架和各个模块的实现原理

包含作者多年开发经验，掌握分布式存储技术必备参考



The Source Code Analysis of Ceph

Ceph源码分析

常涛◎编著



机械工业出版社
China Machine Press

大数据技术丛书

Ceph源码分析

常涛 编著

ISBN : 978-7-111-55207-9

本书纸版由机械工业出版社于2016年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

序言

前言

第1章 Ceph整体架构

1.1 Ceph的发展历程

1.2 Ceph的设计目标

1.3 Ceph基本架构图

1.4 Ceph客户端接口

 1.4.1 RBD

 1.4.2 CephFS

 1.4.3 RadosGW

1.5 RADOS

 1.5.1 Monitor

 1.5.2 对象存储

 1.5.3 pool和PG的概念

 1.5.4 对象寻址过程

 1.5.5 数据读写过程

 1.5.6 数据均衡

 1.5.7 Peering

 1.5.8 Recovery和Backfill

 1.5.9 纠删码

1.5.10 快照和克隆

1.5.11 Cache Tier

1.5.12 Scrub

1.6 本章小结

第2章 Ceph通用模块

2.1 Object

2.2 Buffer

2.2.1 buffer::raw

2.2.2 buffer::ptr

2.2.3 buffer::list

2.3 线程池

2.3.1 线程池的启动

2.3.2 工作队列

2.3.3 线程池的执行函数

2.3.4 超时检查

2.3.5 ShardedThreadPool

2.4 Finisher

2.5 Throttle

2.6 SafeTimer

2.7 本章小结

第3章 Ceph网络通信

3.1 Ceph网络通信框架

3.1.1 Message

3.1.2 Connection

3.1.3 Dispatcher

3.1.4 Messenger

3.1.5 网络连接的策略

3.1.6 网络模块的使用

3.2 Simple实现

3.2.1 SimpleMessenger

3.2.2 Acceptor

3.2.3 DispatchQueue

3.2.4 Pipe

3.2.5 消息的发送

3.2.6 消息的接收

3.2.7 错误处理

3.3 本章小结

第4章 CRUSH数据分布算法

4.1 数据分布算法的挑战

4.2 CRUSH算法的原理

4.2.1 层级化的Cluster Map

4.2.2 Placement Rules

4.2.3 Bucket随机选择算法

4.3 代码实现分析

4.3.1 相关的数据结构

4.3.2 代码实现

4.4 对CRUSH算法的评价

4.5 本章小结

第5章 Ceph客户端

5.1 Librados

5.1.1 RadosClient

5.1.2 IoCtxImpl

5.2 OSDC

5.2.1 ObjectOperation

5.2.2 op_target

5.2.3 Op

5.2.4 Striper

5.2.5 ObjectCacher

5.3 客户写操作分析

5.3.1 写操作消息封装

5.3.2 发送数据op_submit

5.3.3 对象寻址_calc_target

5.4 Cls

5.4.1 模块以及方法的注册

5.4.2 模块的方法执行

5.4.3 举例说明

5.5 Librbd

5.5.1 RBD的相关的对象

5.5.2 RBD元数据操作

5.5.3 RBD数据操作

5.5.4 RBD的快照和克隆

5.6 本章小结

第6章 Ceph的数据读写

6.1 OSD模块静态类图

6.2 相关数据结构

6.2.1 Pool

6.2.2 PG

6.2.3 OSDMap

6.2.4 OSDOp

6.2.5 Object_info_t

6.2.6 ObjectState

6.2.7 SnapSetContext

6.2.8ObjectContext

6.2.9 Session

6.3 读写操作的序列图

6.4 读写流程代码分析

6.4.1 阶段1：接收请求

6.4.2 阶段2：OSD的op_wq处理

6.4.3 阶段3：PGBackend的处理

6.4.4 从副本的处理

6.4.5 主副本接收到从副本的应答

6.5 本章小结

第7章 本地对象存储

7.1 基本概念介绍

7.1.1 对象的元数据

7.1.2 事务和日志的基本概念

7.1.3 事务的封装

7.2 ObjectStore对象存储接口

7.2.1 对外接口说明

7.2.2 ObjectStore代码示例

7.3 日志的实现

7.3.1 Jouanal对外接口

7.3.2 FileJournal

7.4 FileStore的实现

7.4.1 日志的三种类型

7.4.2 JournalingObjectStore

7.4.3 Filestore的更新操作

7.4.4 日志的应用

7.4.5 日志的同步

7.5 omap的实现

7.5.1 omap存储

7.5.2 omap的克隆

7.5.3 部分代码实现分析

7.6 CollectionIndex

7.6.1 CollectIndex接口

7.6.2 HashIndex

7.6.3 LFNIndex

7.7 本章小结

第8章 Ceph纠删码

8.1 EC的基本原理

8.2 EC的不同插件

8.2.1 RS编码

8.2.2 LRC编码

8.2.3 SHEC编码

8.2.4 EC和副本的比较

8.3 Ceph中EC的实现

8.3.1 Ceph中EC的基本概念

8.3.2 EC支持的写操作

8.3.3 EC的回滚机制

8.4 EC的源代码分析

8.4.1 EC的写操作

8.4.2 EC的write_full

8.4.3 ECBackend

8.5 本章小结

第9章 Ceph快照和克隆

9.1 基本概念

9.1.1 快照和克隆

9.1.2 RBD的快照和克隆比较

9.2 快照实现的核心数据结构

9.3 快照的工作原理

9.3.1 快照的创建

9.3.2 快照的写操作

9.3.3 快照的读操作

9.3.4 快照的回滚

9.3.5 快照的删除

9.4 快照读写操作源代码分析

9.4.1 快照的写操作

9.4.2 make_writeable函数

9.4.3 快照的读操作

9.5 本章小结

第10章 Ceph Peering机制

10.1 statechart状态机

10.1.1 状态

10.1.2 事件

- 10.1.3 状态响应事件
- 10.1.4 状态机的定义
- 10.1.5 context函数
- 10.1.6 事件的特殊处理
- 10.2 PG状态机
- 10.3 PG的创建过程
 - 10.3.1 PG在主OSD上的创建
 - 10.3.2 PG在从OSD上的创建
 - 10.3.3 PG的加载
- 10.4 PG创建后状态机的状态转换
- 10.5 Ceph的Peering过程分析
 - 10.5.1 基本概念
 - 10.5.2 PG日志
 - 10.5.3 Peering的状态转换图
 - 10.5.4 pg_info数据结构
 - 10.5.5 GetInfo
 - 10.5.6 GetLog
 - 10.5.7 GetMissing
 - 10.5.8 Active操作
 - 10.5.9 副本端的状态转移
 - 10.5.10 状态机异常处理
- 10.6 本章小结

第11章 Ceph数据修复

11.1 资源预约

11.2 数据修复状态转换图

11.3 Recovery过程

 11.3.1 触发修复

 11.3.2 ReplicatedPG

 11.3.3 pgbackend

11.4 Backfill过程

 11.4.1 相关数据结构

 11.4.2 Backfill的具体实现

11.5 本章小结

第12章 Ceph一致性检查

12.1 端到端的数据校验

12.2 Scrub概念介绍

12.3 Scrub的调度

 12.3.1 相关数据结构

 12.3.2 Scrub的调度实现

12.4 Scrub的执行

 12.4.1 相关数据结构

 12.4.2 Scrub的控制流程

 12.4.3 构建ScrubMap

 12.4.4 从副本处理

12.4.5 副本对比

12.4.6 结束Scrub过程

12.5 本章小结

第13章 Ceph自动分层存储

13.1 自动分层存储技术

13.2 Ceph分层存储架构和原理

13.3 Cache Tier的模式

13.4 Cache Tier的源码分析

13.4.1 pool中的Cache Tier数据结构

13.4.2 HitSet

13.4.3 Cache Tier的初始化

13.4.4 读写路径上的Cache Tier处理

13.4.5 cache的flush和evict操作

13.5 本章小结

序言

自从2013年加入Ceph社区以来，我一直想写一本分析Ceph源码的书，但是两年多来提交了数万行的代码后，我渐渐放下了这个事情。Ceph每个月、每周都会发生巨大变化，我总是想让Ceph源码爱好者看到最新最棒的设计和实现，社区一线的模块维护和每周数十个代码提交集的阅读，让我很难有时间回顾和把握其他Ceph爱好者的疑问和需求点。

今天看到这本书让我非常意外，作者常涛把整个Ceph源码树肢解得恰到好处，如庖丁解牛般将Ceph的核心思想和实现展露出来。虽然目前Ceph分分钟都有新的变化，但无论是新的模块设计，还是重构已有逻辑，都是已有思想的翻新和延续，这些才是众多Ceph开发者能十年如一日改进的秘密！

我跟作者常涛虽然只有一面之缘，但是在开源社区中的交流已经足够成为彼此的相知。他对于分布式存储的设计和实现都有独到见解，其代码阅读和理解灵感更是超群。我在年前看到他一些对Ceph核心模块的创新性理解，相信这些都通过这本书展现出来了。

这本书是目前我所看到的从代码角度解读Ceph的最好作品，即使在全球范围内，都没有类似的书籍能够与之媲美。相信每个Ceph爱好者都能从这本书中找到自己心中某些疑问的解答途径。

作为Ceph社区的主要开发者，我也想在这里强调Ceph的魅力，希望每个读者都能充分感受到Ceph社区生机勃勃的态势。Ceph是开源世界中存储领域的一个里程碑！在过去很难想像，从IT巨无霸们组成的大存储壁垒中能够诞生一个真正被大量用户使用并投入生产环境的开源存储项目，而Ceph这个开源存储项目已经成为全球众多海量存储项目的主要选择。

众所周知，在过去十年里，IT技术领域中巨大的创新项目很多来自于开源世界，从垄断大数据的Hadoop、Spark，到风靡全球的Docker，都证明了开源力量推动了新技术的产生与发展。而再往以前看十年，从Unix到Linux，从Oracle到MySQL/PostgreSQL，从VMWare到KVM，开源世界从传统商业技术继承并给用户带来更多的选择。处于开源社区一线的我欣喜地看到，在IT基础设施领域，越来越多的创业公司从创立之初就以开源为基石，而越来越多的商业技术公司也受益于开源，大量的复杂商业软件基于开源分布式数据库、缓存存储、中间件构建。相信开源的Ceph也将成为IT创新的驱动力。正如Sage Weil在2016 Ceph Next会议上所说，Ceph将成为存储里的Linux！

王豪迈，XSKY公司CTO

2016年9月8日

前言

随着云计算技术的兴起和普及，云计算基石：分布式共享存储系统受到业界的重视。Ceph以其稳定、高可用、可扩展的特性，乘着开源云计算管理系统OpenStack的东风，迅速成为最热门的开源分布式存储系统。

Ceph作为一个开源的分布式存储系统，人人都可以免费获得其源代码，并能够安装部署，但是并不等于人人都能用起来，人人都能用好。用好一个开源分布式存储系统，首先要对其架构、功能原理等方面有比较好的了解，其次要有修复漏洞的能力。这些都是在采用开源分布式存储系统时所面临的挑战。

要用好Ceph，就必须深入了解和掌握Ceph源代码。Ceph源代码的实现被公认为比较复杂，阅读难度较大。阅读Ceph源代码，不但需要对C++语言以及boost库和STL库非常熟悉，还需要有分布式存储系统相关的基础知识以及对实现原理的深刻理解，最后还需要对Ceph框架和设计原理以及具体的实现细节有很好的把握。所以Ceph源代码的阅读是相当有挑战性的。

本着对Ceph源代码的浓厚兴趣以及实践工作的需要，需要对Ceph在源代码层级有比较深入的了解。当时笔者尽可能地搜索有关Ceph源代码的介绍，发现这方面的资料比较少，笔者只能自己对着Ceph源代码开始了比较艰辛的阅读之旅。在这个过程中，每一个小的进步都来之不易，理解一些实现细节，都需要对源代码进行反复地推敲和琢磨。自己在阅读的过程

中，特别希望有人能够帮助理清整体代码的思路，能够解答一下关键的实现细节。本书就是秉承这样一个简单的目标，希望指引和帮助广大Ceph爱好者更好地理解和掌握Ceph源代码。

本书面向热爱Ceph的开发者，想深入了解Ceph原理的高级运维人员，想基于Ceph做优化和定制的开发人员，以及想对社区提交代码的研究人员。官网上有比较详细的介绍Ceph安装部署以及操作相关的知识，希望阅读本书的人能够自己动手实践，对Ceph进一步了解。本书基于目前最新的Ceph 10.2.1版本进行分析。

本书着重介绍Ceph的整体框架和各个实现模块的实现原理，对核心源代码进行分析，包括一些关键的实现细节。存储系统的实现都是围绕数据以及对数据的操作来展开，只要理解核心的数据结构，以及数据结构的相关操作就可以大致了解核心的实现和功能。本书的写作思路是先介绍框架和原理，其次介绍相关的数据结构，最后基于数据结构，介绍相关操作的实现流程。

最后感谢一起工作过的同事们，同他们在Ceph技术上进行交流沟通并加以验证实践，使我受益匪浅。感谢机械工业出版社的编辑吴怡对本书出版所做的努力，以及不断提出的宝贵意见。感谢我的妻子孙盛南女士在我写作期间默默的付出，对本书的写作提供了坚强的后盾。

由于Ceph源代码比较多，也比较复杂，写作的时间比较紧，加上个人的水平有限，错误和疏漏在所难免，恳请读者批评指正。有任何的意见和

建议都可发送到我的邮箱changtao381@163.com，欢迎读者与我交流Ceph相关的任何问题。

常涛

2016年6月于北京

第1章 Ceph整体架构

本章从比较高的层次对Ceph的发展历史、Ceph的设计目标、整体架构进行简要介绍。其次介绍Ceph的三种对外接口：块存储、对象存储、文件存储。还介绍Ceph的存储基石RADOS系统的一些基本概念、各个模块组成和功能。最后介绍了对象的寻址过程和数据读写的原理，以及RADOS实现的数据服务等。

1.1 Ceph的发展历程

Ceph项目起源于其创始人Sage Weil在加州大学Santa Cruz分校攻读博士期间的研究课题。项目的起始时间为2004年，在2006年基于开源协议开源了Ceph的源代码。Sage Weil也相应成立了Inktank公司专注于Ceph的研发。在2014年5月，该公司被Red Hat收购。Ceph项目的发展历程如图1-1所示。

2012年，Ceph发布了第一个稳定版本。2014年10月，Ceph开发团队发布了Ceph的第七个稳定版本Giant。到目前为止，社区平均每三个月发布一个稳定版本，目前的最新版本为10.2.1。

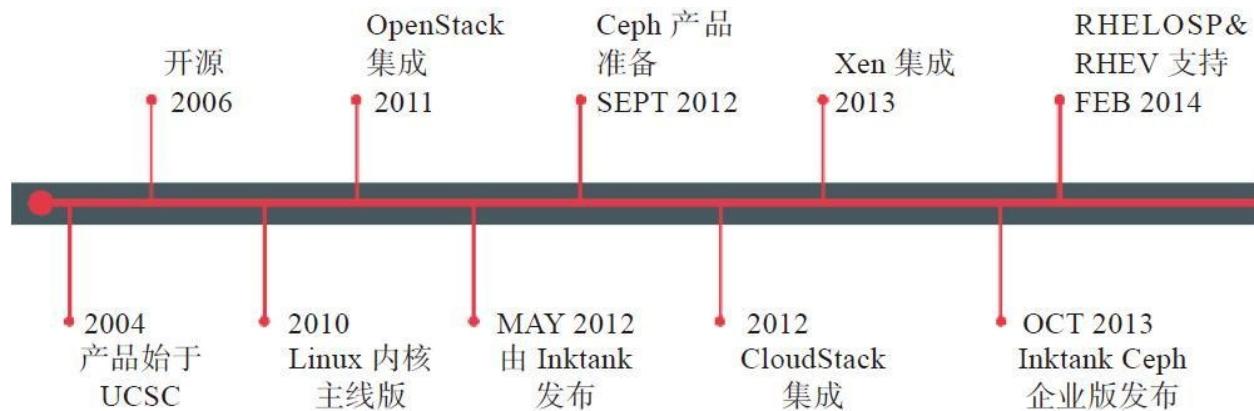


图1-1 Ceph的发展历程

1.2 Ceph的设计目标

Ceph的设计目标是采用商用硬件（Commodity Hardware）来构建大规模的、具有高可用性、高可扩展性、高性能的分布式存储系统。

商用硬件一般指标准的x86服务器，相对于专用硬件，性能和可靠性较差，但由于价格相对低廉，可以通过集群优势来发挥高性能，通过软件的设计解决高可用性和可扩展性。标准化的硬件可以极大地方便管理，且集群的灵活性可以应对多种应用场景。

系统的高可用性指的是系统某个部件失效后，系统依然可以提供正常服务的能力。一般用设备部件和数据的冗余来提高可用性。Ceph通过数据多副本、纠删码来提供数据的冗余。

高可扩展性是指系统可以灵活地应对集群的伸缩。一般指两个方面，一方面指集群的容量可以伸缩，集群可以任意地添加和删除存储节点和存储设备；另一方面指系统的性能随集群的增加而线性增加。

大规模集群环境下，要求Ceph存储系统的规模可以扩展到成千上万个节点。当集群规模达到一定程度时，系统在数据恢复、数据迁移、节点监测等方面会产生一系列富有挑战性的问题。

1.3 Ceph基本架构图

Ceph的整体架构由三个层次组成：最底层也是最核心的部分是RADOS对象存储系统。第二层是librados库层；最上层对应着Ceph不同形式的存储接口实现，架构如图1-2所示。

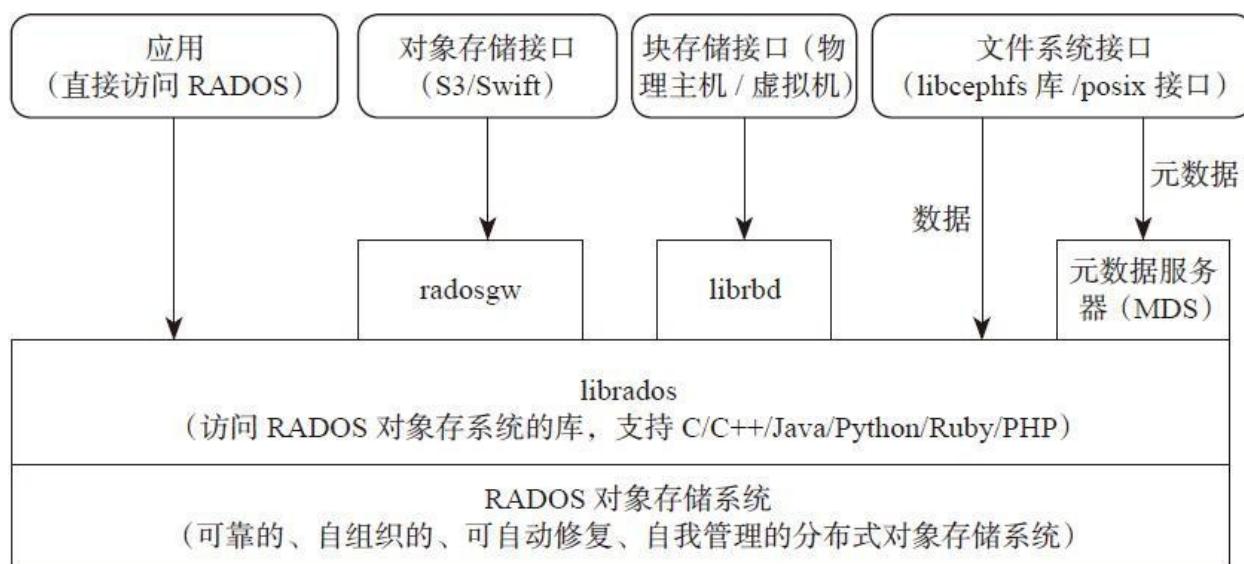


图1-2 Ceph基本架构图

Ceph的整体架构大致如下：

- 最底层基于RADOS (reliable, autonomous, distributed object store)，它是一个可靠的、自组织的、可自动修复、自我管理的分布式对象存储系统。其内部包括ceph-osd后台服务进程和ceph-mon监控进程。
- 中间层librados库用于本地或者远程通过网络访问RADOS对象存储系统。它支持多种语言，目前支持C/C++语言、Java、Python、Ruby和PHP语

言的接口。

·最上层面向应用提供3种不同的存储接口：

·块存储接口，通过librbd库提供了块存储访问接口。它可以为虚拟机提供虚拟磁盘，或者通过内核映射为物理主机提供磁盘空间。

·对象存储接口，目前提供了两种类型的API，一种是和AWS的S3接口兼容的API，另一种是和OpenStack的Swift对象接口兼容的API。

·文件系统接口，目前提供两种接口，一种是标准的posix接口，另一种通过libcephfs库提供文件系统访问接口。文件系统的元数据服务器MDS用于提供元数据访问。数据直接通过librados库访问。

1.4 Ceph客户端接口

Ceph的设计初衷是成为一个分布式文件系统，但随着云计算的大量应用，最终变成支持三种形式的存储：块存储、对象存储、文件系统，下面介绍它们之间的区别。

1.4.1 RBD

RBD (rados block device) 是通过librbd库对应用提供块存储，主要面向云平台的虚拟机提供虚拟磁盘。传统SAN就是块存储，通过SCSI或者FC接口给应用提供一个独立的LUN或者卷。RBD类似于传统的SAN存储，都提供数据块级别的访问。

目前RBD提供了两个接口，一种是直接在用户态实现，通过QEMU Driver供KVM虚拟机使用。另一种是在操作系统内核态实现了一个内核模块。通过该模块可以把块设备映射给物理主机，由物理主机直接访问。

块存储用作虚拟机的硬盘，其对I/O的要求和传统的物理硬盘类似。一个硬盘应该是能面向通用需求的，既能应付大文件读写，也能处理好小文件读写。也就是说，块存储既需要有较好的随机I/O，又要求有较好的顺序I/O，而且对延迟有比较严格的要求。

1.4.2 CephFS

CephFS通过在RADOS基础之上增加了MDS（Metadata Server）来提供文件存储。它提供了libcephfs库和标准的POSIX文件接口。CephFS类似于传统的NAS存储，通过NFS或者CIFS协议提供文件系统或者文件目录服务。

Ceph最初的设计为分布式文件系统，其通过动态子树的算法实现了多元数据服务器，但是由于实现复杂，目前还远远不能使用。目前可用于生产环境的是最新Jewel版本的CephFS为主从模式（Master-Slave）的元数据服务器。

1.4.3 RadosGW

RadosGW基于librados提供了和Amazon S3接口以及OpenStack Swift接口兼容的对象存储接口。可将其简单地理解为提供基本文件（或者对象）的上传和下载的需求，它有两个特点：

- 提供RESTful Web API接口。
- 它采用扁平的数据组织形式。

1. RESTful的存储接口

其接口值提供了简单的GET、PUT、DEL等其他接口，对应对象文件的上传、下载、删除、查询等操作。可以看出，对象存储的I/O接口相对比较简单，其I/O访问模型都是顺序I/O访问。

2. 扁平的数据组织形式

NAS存储提供给应用的是一个文件系统或者是一个文件夹，其实质就是一个层级化的树状存储组织模式，其嵌套层级和规模在理论上都不受限制。

这种树状的目录结构为早期本地存储系统设计的信息组织形式，比较直观，容易理解。但是随着存储系统规模的不断扩大，特别是到了云存储时代，其难以大规模扩展的缺点就暴露了出来。相比于NAS存储，对象存

储放弃了目录树结构，采用了扁平化组织形式（一般为三级组织结构），这有利于实现近乎无限的容量扩展。它使用业界标准互联网协议，更加符合面向云服务的存储、归档和备份需求。

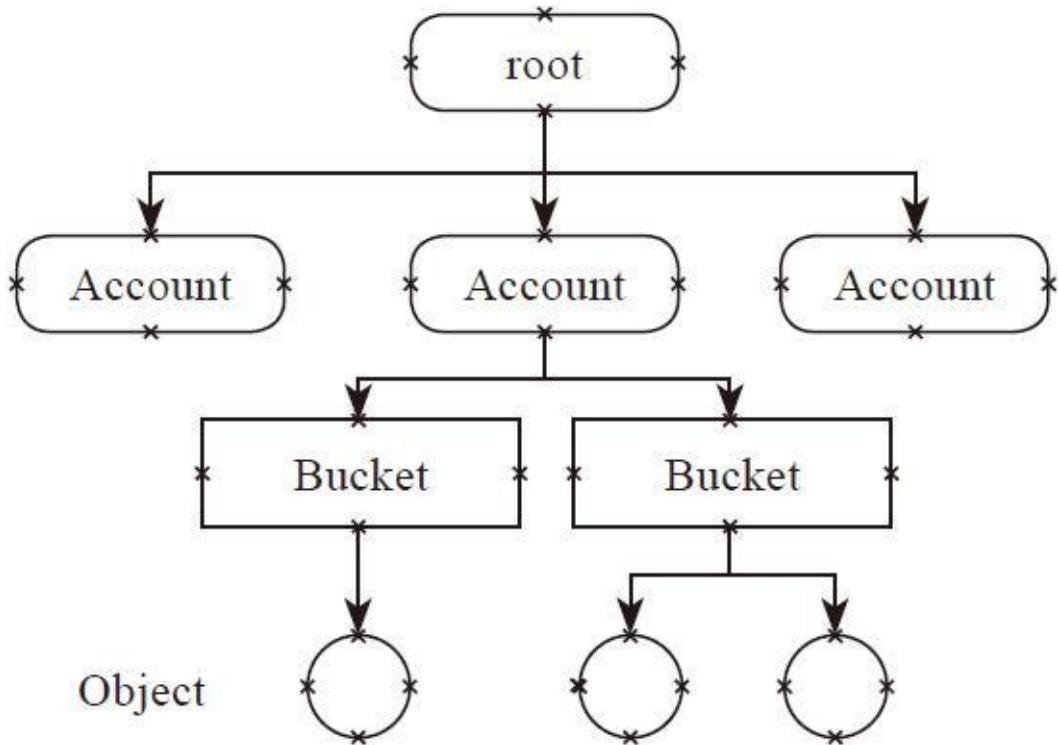


图1-3 Amazon S3的对象存储结构

由于Amazon在云存储领域的影响力，Amazon的S3接口已经成为事实上的对象存储的标准接口。如图1-3所示，其接口分三级存储：Account/Bucket/Object（账户/桶/对象）。一个Account可以看作一个用户（租户），其下可以包含若干个的Bucket，一个Bucket可以拥有若干对象，其数量在理论上都不受限制。

在云计算领域，OpenStack已经成为广泛采用的云计算管理系统，OpenStack的对象存储接口Swift也成为广泛采用的接口，如图1-4所示，其

也采用分三级存储：Account/Container/Object（账户/容器/对象），每层节点数均没有限制。可以看出，Swift接口和S3类似，Swift的Container对应S3的Bucket概念。Swift接口和S3接口没有太大的区别，但是一些管理接口会有一些差别。

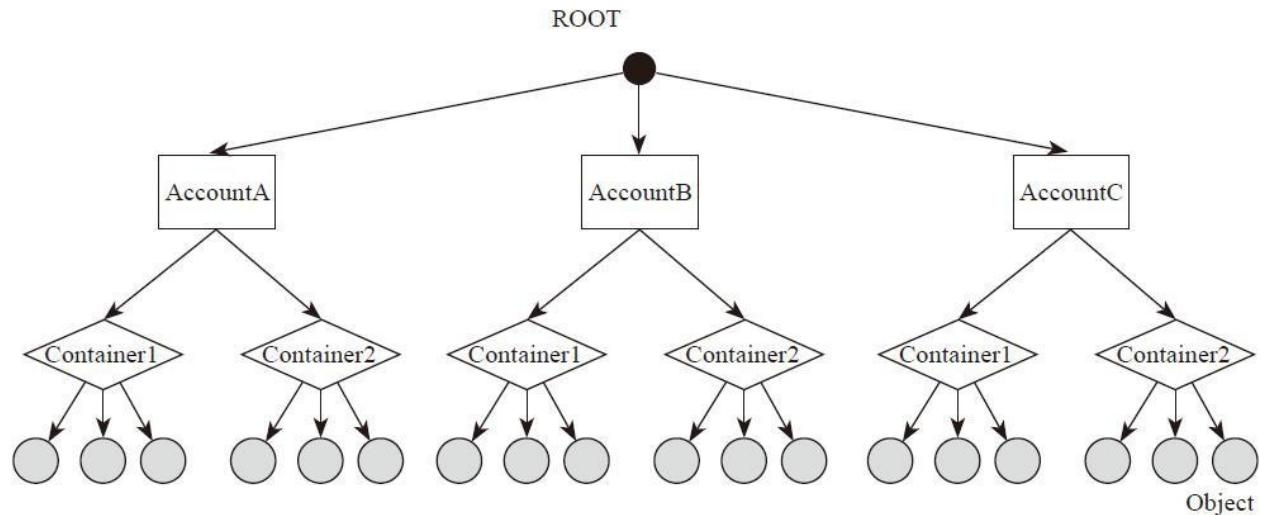


图1-4 OpenStack Swift对象存储结构

1.5 RADOS

RADOS是Ceph存储系统的基石，是一个可扩展的、稳定的、自我管理的、自我修复的对象存储系统，是Ceph存储系统的核心。它完成了一个存储系统的核心功能，包括：Monitor模块为整个存储集群提供全局的配置和系统信息；通过CRUSH算法实现对象的寻址过程；完成对象的读写以及其他数据功能；提供了数据均衡功能；通过Peering过程完成一个PG内存达成数据一致性的过程；提供数据自动恢复的功能；提供克隆和快照功能；实现了对象分层存储的功能；实现了数据一致性检查工具Scrub。下面分别对上述基本功能做简要的介绍。

1.5.1 Monitor

Monitor是一个独立部署的daemon进程。通过组成Monitor集群来保证自己的高可用。Monitor集群通过Paxos算法实现了自己数据的一致性。它提供了整个存储系统的节点信息等全局的配置信息。

Cluster Map保存了系统的全局信息，主要包括：

- Monitor Map
 - 包括集群的fsid
 - 所有Monitor的地址和端口
 - current epoch
- OSD Map：所有OSD的列表， 和OSD的状态等。
- MDS Map：所有的MDS的列表和状态。

1.5.2 对象存储

这里所说的对象是指RADOS对象，要和RadosGW的S3或者Swift接口的对象存储区分开来。对象是数据存储的基本单元，一般默认4MB大小。图1-5就是一个对象的示意图。

ID	Binary Data	Metadata
1234	0101010101010100110101010010 0101100001010100110101010010 0101100001010100110101010010	name1 value1 name2 value2 nameN valueN

图1-5 对象示意图

一个对象由三个部分组成：

- 对象标志（ID），唯一标识一个对象。
- 对象的数据，其在本地文件系统中对应一个文件，对象的数据就保存在文件中。
- 对象的元数据，以Key-Value（键值对）的形式，可以保存在文件对应的扩展属性中。由于本地文件系统的扩展属性能保存的数据量有限制，RADOS增加了另一种方式：以Leveldb等的本地KV存储系统来保存对象的元数据。

1.5.3 pool和PG的概念

pool是一个抽象的存储池。它规定了数据冗余的类型以及对应的副本分布策略。目前实现了两种pool类型：replicated类型和Erasure Code类型。一个pool由多个PG构成。

PG (placement group) 从名字可理解为一个放置策略组，它是对象的集合，该集合里的所有对象都具有相同的放置策略：对象的副本都分布在相同的OSD列表上。一个对象只能属于一个PG，一个PG对应于放置在其上的OSD列表。一个OSD上可以分布多个PG。

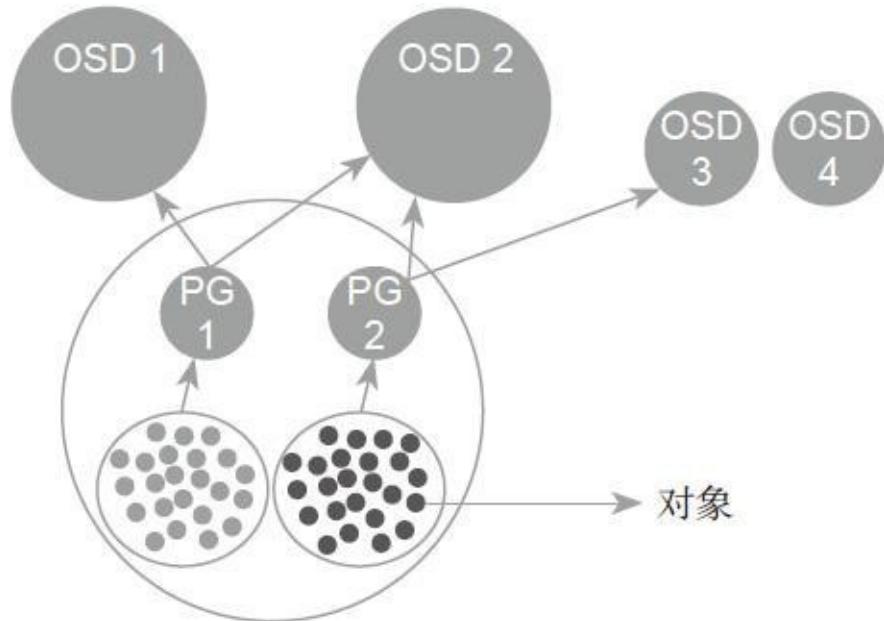


图1-6 PG的概念示意图

PG的概念如图1-6所示，其中：

- PG1和PG2都属于同一个pool， 所以都是副本类型， 并且都是两副本。
- PG1和PG2里都包含许多对象， PG1上的所有对象， 主从副本分布在OSD1和OSD2上， PG2上的所有对象的主从副本分布在OSD2和OSD3上。
- 一个对象只能属于一个PG， 一个PG包含多个对象。
- 一个PG的副本分布在对应的OSD列表中。在一个OSD上可以分布多个PG。示例中PG1和PG2的从副本都分布在OSD2上。

1.5.4 对象寻址过程

对象寻址过程指的是查找对象在集群中分布的位置信息，过程分为两步：

1) 对象到PG的映射。这个过程是静态hash映射（加入pg split后实际变成了动态hash映射方式），通过对object_id，计算出hash值，用该pool的PG的总数量pg_num对hash值取模，就可以获得该对象所在的PG的id号：

```
pg_id = hash( object_id ) % pg_num
```

2) PG到OSD列表映射。这是指PG上对象的副本如何分布在OSD上。它使用Ceph自己创新的CRUSH算法来实现，本质上是一个伪随机分布算法。

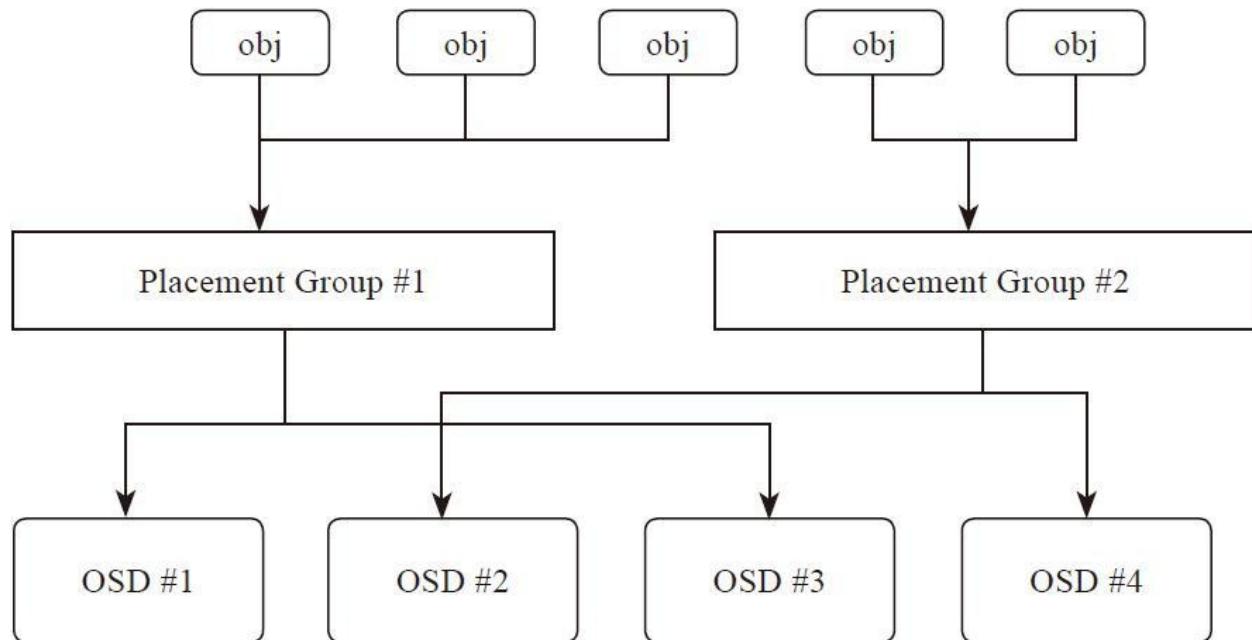


图1-7 对象寻址过程

如图1-7所示的对象寻址过程：

- 1) 通过hash取模后计算，前三个对象分布在PG1上，后两个对象分布在PG2上。
- 2) PG1通过CRUSH算法，计算出PG1分布在OSD1、OSD3上；PG2通过CRUSH算法分布在OSD2和OSD4上。

1.5.5 数据读写过程

Ceph的数据写操作如图1-8所示，其过程如下：

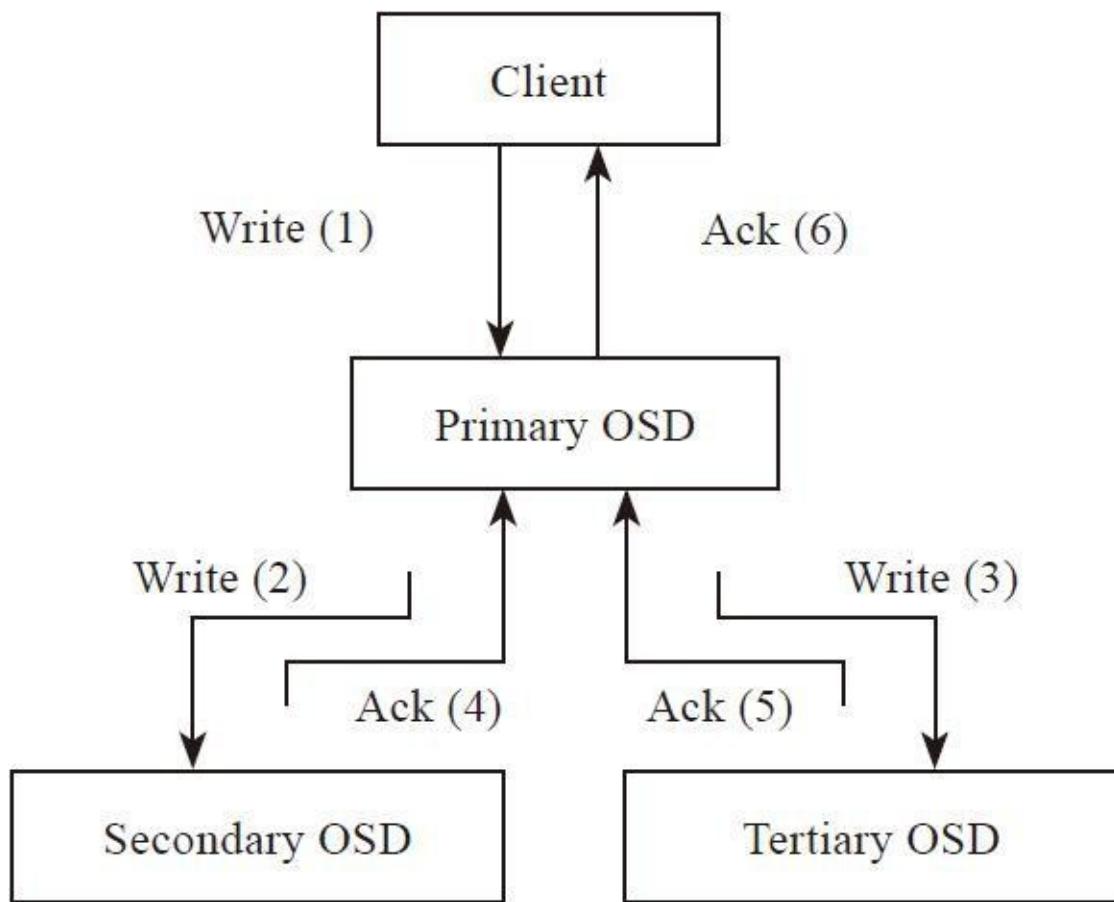


图1-8 读写过程

- 1) Client向该PG所在的主OSD发送写请求。
- 2) 主OSD接收到写请求后，同时向两个从OSD发送写副本的请求，并同时写入主OSD的本地存储中。
- 3) 主OSD接收到两个从OSD发送写成功的ACK应答，同时确认自己

写成功，就向客户端返回写成功的ACK应答。

在写操作的过程中，主OSD必须等待所有的从OSD返回正确应答，才能向客户端返回写操作成功的应答。由于读操作比较简单，这里就不介绍了。

1.5.6 数据均衡

当在集群中新添加一个OSD存储设备时，整个集群会发生数据的迁移，使得数据分布达到均衡。Ceph数据迁移的基本单位是PG，即数据迁移是将PG中的所有对象作为一个整体来迁移。

迁移触发的流程为：当新加入一个OSD时，会改变系统的CRUSH Map，从而引起对象寻址过程中的第二步，PG到OSD列表的映射发生了变化，从而引发数据的迁移。

举例来说明数据迁移过程，表1-1是数据迁移前的PG分布，表1-2是数据迁移后的PG分布。

表1-1 数据迁移前的PG分布

	OSD1	OSD2	OSD3
PGa	PGa1	PGa2	PGa3
PGb	PGb3	PGb1	PGb2
PGc	PGc2	PGc3	PGc1
PGd	PGd1	PGd2	PGd3

表1-2 数据迁移后的PG分布

	OSD1	OSD2	OSD3	OSD4
PGa	PGa1	PGa2	PGa3	PGa1
PGb	PGb3	PGb1	PGb2	PGb2
PGc	PGc2	PGc3	PGc1	PGc3
PGd	PGd1	PGd2	PGd3	

当前系统有3个OSD，分布为OSD1、OSD2、OSD3；系统有4个PG，

分布为PGa、 PGb、 PGc、 PGd； PG设置为三副本： PGa1、 PGa2、 PGa3分别为PGa的三个副本。 PG的所有分布如表1-1所示， 每一行代表PG分布的OSD列表。

当添加一个新的OSD4时， CRUSH Map变化导致通过CRUSH算法来计算PG到OSD的分布发生了变化。如表1-2所示： PGa的映射到了列表[OSD4, OSD2, OSD3]上， 导致PGa1从OSD1上迁移到了OSD4上。同理， PGb2从OSD3上迁移到OSD4， PGc3从OSD2上迁移到OSD4上， 最终数据达到了基本平衡。

1.5.7 Peering

当OSD启动，或者某个OSD失效时，该OSD上的主PG会发起一个Peering的过程。Ceph的Peering过程是指一个PG内的所有副本通过PG日志来达成数据一致的过程。当Peering完成之后，该PG就可以对外提供读写服务了。此时PG的某些对象可能处于数据不一致的状态，其被标记出来，需要恢复。在写操作的过程中，遇到处于不一致的数据对象需要恢复的话，则需要等待，系统优先恢复该对象后，才能继续完成写操作。

1.5.8 Recovery和Backfill

Ceph的Recovery过程是根据在Peering的过程中产生的、依据PG日志推算出的不一致对象列表来修复其他副本上的数据。

Recovery过程的依据是根据PG日志来推测出不一致的对象加以修复。当某个OSD长时间失效后重新加入集群，它已经无法根据PG日志来修复，就需要执行Backfill（回填）过程。Backfill过程是通过逐一对比两个PG的对象列表来修复。当新加入一个OSD产生了数据迁移，也需要通过Backfill过程来完成。

1.5.9 纠删码

纠删码（Erasure Code）的概念早在20世纪60年代就提出来了，最近几年被广泛应用在存储领域。它的原理比较简单：将写入的数据分成N份原始数据块，通过这N份原始数据块计算出M份效验数据块， $N+M$ 份数据块可以分别保存在不同的设备或者节点中。可以允许最多M个数据块失效，通过 $N+M$ 份中的任意N份数据，就还原出其他数据块。

目前Ceph对纠删码（EC）的支持还比较有限。RBD目前不能直接支持纠删码（EC）模式。其或者应用在对象存储radosgw中，或者作为Cache Tier的二层存储。其中的原因和具体实现都将在后面的章节详细介绍。

1.5.10 快照和克隆

快照（snapshot）就是一个存储设备在某一时刻的全部只读镜像。克隆（clone）是在某一时刻的全部可写镜像。快照和克隆的区别在于快照只能读，而克隆可写。

RADOS对象存储系统本身支持Copy-on-Write方式的快照机制。基于这个机制，Ceph可以实现两种类型的快照，一种是pool级别的快照，给整个pool中的所有对象统一做快照操作。另一种就是用户自己定义的快照实现，这需要客户端配合实现一些快照机制。RBD的快照实现就属于后者。

RBD的克隆实现是在基于RBD的快照基础上，在客户端librbd上实现了Copy-on-Write（cow）克隆机制。

1.5.11 Cache Tier

RADOS实现了以pool为基础的自动分层存储机制。它在第一层可以设置cache pool，其为高速存储设备（例如SSD设备）。第二层为data pool，使用大容量低速存储设备（如HDD设备）可以使用EC模式来降低存储空间。通过Cache Tier，可以提高关键数据或者热点数据的性能，同时降低存储开销。

Cache Tier的结构如图1-9所示，说明如下：

- Ceph Client对于Cache层是透明的。
- 类Objecter负责请求是发给Cache Tier层，还是发给Storage Tier层。
- Cache Tier层为高速I/O层，保存热点数据，或称为活跃的数据。
- Storage Tier层为慢速层，保存非活跃的数据。
- 在Cache Tier层和Storage Tier层之间，数据根据活跃度自动地迁移。

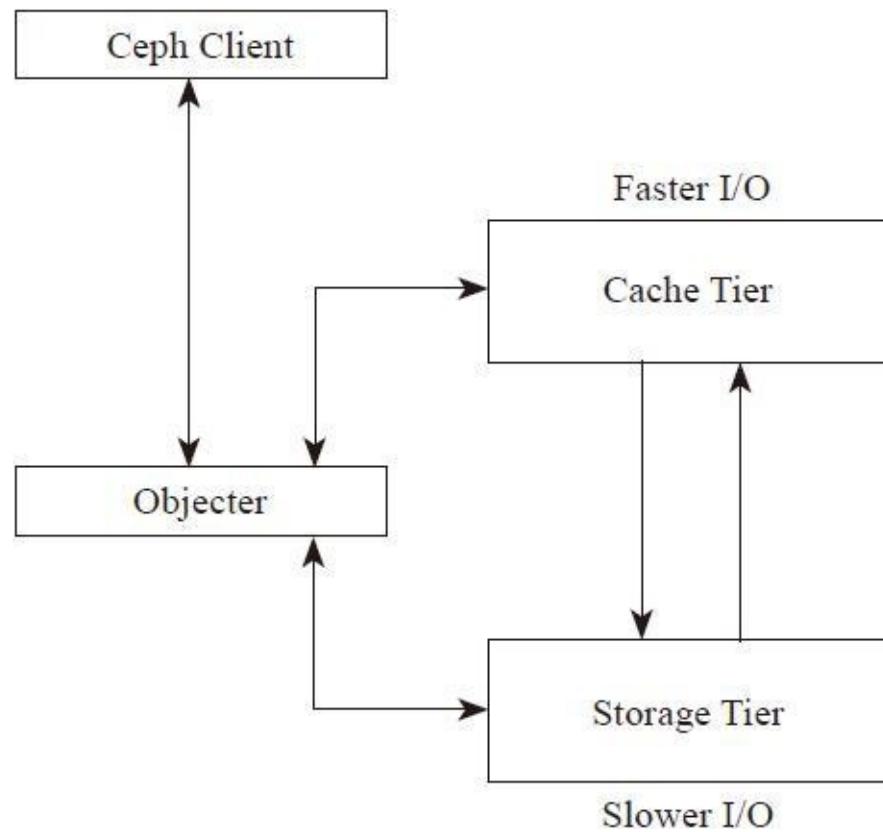


图1-9 Cache Tier结构图

1.5.12 Scrub

Scrub机制用于系统检查数据的一致性。它通过在后台定期（默认每天一次）扫描，比较一个PG内的对象分别在其他OSD上的各个副本的元数据和数据来检查是否一致。根据扫描的内容分为两种，第一种是只比较对象各个副本的元数据，它代价比较小，扫描比较高效，对系统影响比较小。另一种扫描称为deep scrub，它需要进一步比较副本的数据内容检查数据是否一致。

1.6 本章小结

本章介绍了Ceph的系统架构，通过本章，可以对Ceph的基本架构和各个模块的组件有了整体的了解，并对一些基本概念及读写的原理、各个数据功能模块有了大致了解。

本章介绍的Ceph客户端的基本概念，在第5章会详细介绍到。本章介绍的对象寻址的CRUSH算法将会在第4章详细介绍到。在本章介绍的本地对象存储将会在第7章详细介绍到。本章介绍的数据读写流程，将会在第6章详细介绍。本章介绍的纠删码将在第8章中详细介绍。本章介绍的快照和克隆将在第9章详细介绍。本章介绍的Ceph Peering的过程将会在第10章详细介绍。本章介绍的数据恢复和回填将会在第11章介绍到。本章介绍的Ceph Srcub机制将会在第12章详细介绍。本章介绍的Cache Tier将在第13章详细介绍。

第2章 Ceph通用模块

本章介绍Ceph源代码通用库中的一些比较关键而又比较复杂的数据结构。Object和Buffer相关的数据结构是普遍使用的。线程池ThreadPool可以提高消息处理的并发能力。Finisher提供了异步操作时来执行回调函数。Throttle在系统的各个模块各个环节都可以看到，它用来限制系统的请求，避免瞬时大量突发请求对系统的冲击。SafeTimer提供了定时器，为超时和定时任务等提供了相应的机制。理解这些数据结构，能够更好理解后面章节的相关内容。

2.1 Object

对象Object是默认为4MB大小的数据块。一个对象就对应本地文件系统中的一个文件。在代码实现中，有object、sobject、hobject、ghobject等不同的类。

结构object_t对应本地文件系统的一个文件，name就是对象名：

```
struct object_t {
    string name;
    .....
}
```

sobject_t在object_t之上增加了snapshot信息，用于标识是否是快照对象。数据成员snap为快照对象的对应的快照序号。如果一个对象不是快照对象（也就是head对象），那么snap字段就被设置为CEPH_NOSNAP值。

```
struct sobject_t {
    object_t oid;
    snapid_t snap;
    .....
}
```

hobject_t是名字应该是hash object的缩写。

```
struct hobject_t {
    object_t oid;
    snapid_t snap;
private:
    uint32_t hash;
    bool max;
    uint32_t nibblewise_key_cache;
    uint32_t hash_reverse_bits;
    .....
}
```

```
public:  
    int64_t pool;  
    string nspace;  
private:  
    string key;  
    .....  
}
```

其在sobject_t的基础上增加了一些字段：

- int64_t pool：所在的pool的id。
- string nspace：nspace一般为空，它用于标识特殊的对象。
- string key：对象的特殊标记。
- string hash：hash和key不能同时设置，hash值一般设置为就是pg的id值。

ghobject_t在对象hobject_t的基础上，添加了generation字段和shard_id字段，这个用于ErasureCode模式下的PG：

- shard_id用于标识对象所在的osd在EC类型的PG中的序号，对应EC来说，每个osd在PG中的序号在数据恢复时非常关键。如果是Replicate类型的PG，那么字段就设置为NO_SHARD (-1)，该字段对于replicate是没用。
 - generation用于记录对象的版本号。当PG为EC时，写操作需要区分写前后两个版本的object，写操作保存对象的上一个版本(generation)的对象，当EC写失败时，可以rollback到上一个版本。
-

```
struct gobject_t {
    hobject_t hobj;
    gen_t generation;
    shard_id_t shard_id;
    bool max;
public:
    static const gen_t NO_GEN = UINT64_MAX;
    .....
}
```

2.2 Buffer

Buffer就是一个命名空间，在这个命名空间下定义了Buffer相关的数据结构，这些数据结构在Ceph的源代码中广泛使用。下面介绍的buffer`::`raw类是基础类，其子类完成了Buffer数据空间的分配，buffer`::`ptr类实现了Buffer内部的一段数据，buffer`::`list封装了多个数据段。

2.2.1 buffer::raw

类buffer::raw是一个原始的数据Buffer，在其基础之上添加了长度、引用计数和额外的crc校验信息，结构如下：

```
class buffer::raw {
public:
    char *data;           //数据指针

    unsigned len;         //数据长度

    atomic_t nref;        //引用计数

    mutable RWLock crc_lock; //读写锁，保护

crc_map
map<pair<size_t, size_t>, pair<uint32_t, uint32_t> > crc_map;
//crc校验信息，第一个

pair为数据段的起始和结束 (
from,to), 第二个

pair是

crc32校验

码，

pair的第一字段为

base crc32校验码，第二个字段为加上数据段后计算出的

crc32校验码。

.....

}
```

下列类都继承了buffer::raw，实现了data对应内存空间的申请：

- 类raw_malloc实现了用malloc函数分配内存空间的功能。
- 类class buffer :: raw_mmap_pages实现了通过mmap来把内存匿名映射到进程的地址空间。
- 类class buffer :: raw_posix_aligned调用了函数posix_memalign来申请内存地址对齐的内存空间。
- 类class buffer :: raw_hack_aligned是在系统不支持内存对齐申请的情况下自己实现了内存地址的对齐。
- 类class buffer :: raw_pipe实现了pipe做为Buffer的内存空间。
- 类class buffer :: raw_char使用了C++的new操作符来申请内存空间。

2.2.2 buffer::ptr

类buffer::ptr就是对于buffer::raw的一个部分数据段。结构如下：

```
class CEPH_BUFFER_API ptr {
    raw *_raw;
    unsigned _off, _len;
    .....
}
```

ptr是raw里的一个任意的数据段，_off是在_raw里的偏移量，_len是ptr的长度。raw和ptr的示意图如图2-1所示。

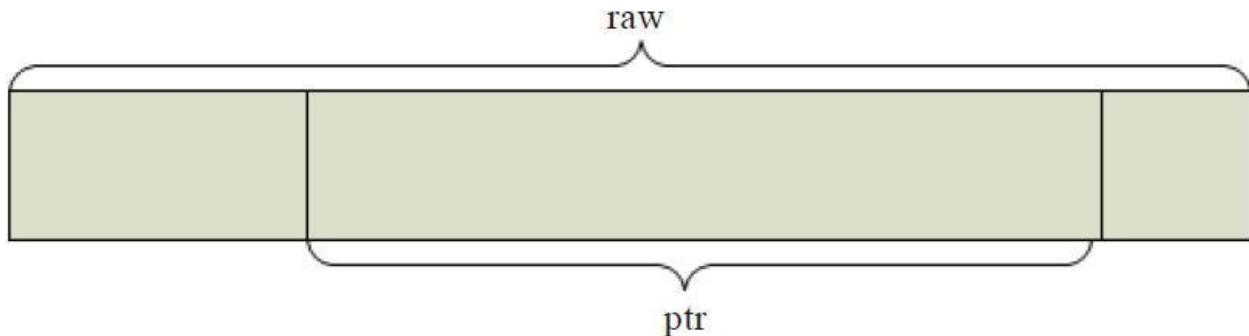


图2-1 raw和ptr示意图

2.2.3 buffer::list

类buffer::list是一个使用广泛的类，它是多个buffer::ptr的列表，也就是多个内存数据段的列表。结构如下：

```
class CEPH_BUFFER_API list {
    std::list<ptr> _buffers;      //所有的
    ptr
    unsigned _len;           //所有的
    ptr的数据总长度

    unsigned _memcpy_count;    //当调用函数
    rebuild用来内存对齐时，需要内存拷贝的数据量

    ptr append_buffer;        //当有小的数据就添加到这个
    buffer里

    mutable iterator last_p;   //访问
    list的迭代器

    .....
}
```

buffer::list的重要的操作如下所示。

- 添加一个ptr到list的头部：

```
void push_front(ptr& bp) {
    if (bp.length() == 0)
        return;
    _buffers.push_front(bp);
    _len += bp.length();
```

```
}
```

·添加一个raw到list头部中，先构造一个ptr，后添加list中：

```
void push_front(raw *r) {
    ptr bp(r);
    push_front(bp);
}
```

·判断内存是否以参数align对齐，每一个ptr都必须以align对齐：

```
bool buffer::list::is_aligned(unsigned align) const
{
    for (std::list<ptr>::const_iterator it = _buffers.begin();
        it != _buffers.end();
        ++it)
        if (!it->is_aligned(align))
            return false;
    return true;
}
```

·添加一个字符到list中，先查看append_buffer是否有足够的空间，如果没有，就新申请一个4KB大小的空间：

```
void buffer::list::append(char c)
{
    // 检查当前的
    append_buffer是否有足够的空间

    unsigned gap = append_buffer.unused_tail_length();
    if (!gap) {
        // 如果没有空间，就申请一个

        append_buffer!
        append_buffer = create_aligned(CEPH_BUFFER_APPEND_SIZE,
                                      CEPH_BUFFER_APPEND_SIZE);
        append_buffer.set_length(0);    //到目前为止，没有用到
    }
    append(append_buffer, append_buffer.append(c) - 1, 1);
    // 把该数据段添加到
```

```
append_buffer中
```

```
}
```

·内存对齐：有些情况下，需要内存地址对齐，例如当以directIO方式写入数据至磁盘时，需要内存地址按内存页面大小（page）对齐，也即buffer`:`list的内存地址都需按page对齐。函数rebuild用来完成对齐的功能。其实现的方法也比较简单，检查没有对齐的ptr，申请一块新对齐的内存，把数据拷贝过去，释放内存空间就可以了。

·buffer`:`list还集成了其他额外的一些功能：

·把数据写入文件或从文件读取数据的功能。

·计算数据的crc32校验。

2.3 线程池

线程池（ThreadPool）在分布式存储系统的实现中是必不可少的，在Ceph的代码中广泛用到。Ceph中线程池的实现也比较复杂，结构如下：

```
class ThreadPool : public md_config_obs_t {
    CephContext *cct;
    string name;           //线程池的名字

    string lockname;      //锁的名字

    Mutex _lock;          //线程互斥的锁，也是工作队列访问互斥的锁

    Cond _cond;           //锁对应的条件变量

    bool _stop;           //线程池是否停止的标志

    int _pause;           //暂时中止线程池的标志

    int _draining;
    Cond _wait_cond;
    int ioprio_class, ioprio_priority;
    vector<WorkQueue_*> work_queues;     //工作队列

    int last_work_queue;        //最后访问的工作队列

    set<WorkThread*> _threads;       //线程池中的工作线程

    list<WorkThread*> _old_threads;   //等待进
joined操作的线程

    int processing;
}
```

类ThreadPool里包涵一些比较重要的数据成员：

- 工作线程集合_threads。

- 等待Join操作的旧线程集合_old_threads。

- 工作队列集合，保存所有要处理的任务。一般情况下，一个工作队列对应一个类型的处理任务，一个线程池对应一个工作队列，专门用于处理该类型的任务。如果是后台任务，又不紧急，就可以将多个工作队列放置到一个线程池里，该线程池可以处理不同类型的任务。

线程池的实现主要包括：线程池的启动过程，线程池对应的工作队列的管理，线程池对应的执行函数如何执行任务。下面分别介绍这些实现，然后介绍一些Ceph线程池实现的超时检查功能，最后介绍ShardedThreadpool的实现原理。

2.3.1 线程池的启动

函数ThreadPool::start() 用来启动线程池，其在加锁的情况下，调用函数start_threads，该函数检查当前线程数，如果小于配置的线程池，就创建新的工作线程。

2.3.2 工作队列

工作队列（WorkQueue）定义了线程池要处理的任务。任务类型在模板参数中指定。在构造函数里，就把自己加入到线程池的工作队列集中：

```
template<class T>
class WorkQueue : public WorkQueue_ {
    ThreadPool *pool;
    WorkQueue(string n, time_t ti, time_t sti, ThreadPool* p) : WorkQueue_
        (n, ti, sti), pool(p) {
        pool->add_work_queue(this);
    }
    .....
}
```

WorkQueue实现了一部分功能：进队列和出队列，以及加锁，并用通过条件变量通知相应的处理线程：

```
bool queue(T *item) {
    pool->_lock.Lock();
    bool r = _enqueue(item);
    pool->_cond.SignalOne();
    pool->_lock.Unlock();
    return r;
}
void dequeue(T *item) {
    pool->_lock.Lock();
    _dequeue(item);
    pool->_lock.Unlock();
}
void clear() {
    pool->_lock.Lock();
    _clear();
    pool->_lock.Unlock();
}
```

还有一部分功能，需要使用者自己定义。需要自己定义实现保存任务的容器，添加和删除的方法，以及如何处理任务的方法：

```
virtual bool _enqueue(T *) = 0;
//从提交的任务中去除一个项

virtual void _dequeue(T *) = 0;
//去除一个项并返回原始指针

virtual T *_dequeue() = 0;
virtual void _process(T *t) { assert(0); }
virtual void _process(T *t, TPHandle &) {
    _process(t);
}
```

2.3.3 线程池的执行函数

函数worker为线程池的执行函数：

```
void ThreadPool::worker(WorkThread *wt)
```

其处理过程如下：

- 1) 首先检查_stop标志，确保线程池没有关闭。
- 2) 调用函数join_old_threads把旧的工作线程释放掉。检查如果线程数量大于配置的数量_num_threads，就把当前线程从线程集合中删除，并加入_old_threads队列中，并退出循环。
- 3) 如果线程池没有暂时中止，并且work_queues不为空，就从last_work_queue开始，遍历每一个工作队列，如果工作队列不为空，就取出一个item，调用工作队列的处理函数做处理。

2.3.4 超时检查

TPHandle是一个有意思的事情。每次线程函数执行时，都会设置一个grace超时时间，当线程执行超过该时间，就认为是unhealthy的状态。当执行时间超过suicide_grace时，OSD就会产生断言而导致自杀，代码如下：

```
struct heartbeat_handle_d {
    const std::string name;
    atomic_t timeout, suicide_timeout;
    time_t grace, suicide_grace;
    std::list<heartbeat_handle_d*>::iterator list_item;
}
class TPHandle {
    friend class ThreadPool;
    CephContext *cct;
    heartbeat_handle_d *hb; //心跳

    time_t grace;           //超时

    time_t suicide_grace;  //自杀的超时时间

}
```

结构heartbeat_handle_d记录了相关信息，并把该结构添加到HeartbeatMap的系统链表中保存。OSD会有一个定时器，定时检查是否超时。

2.3.5 ShardedThreadPool

这里简单介绍一个ShardedThreadPool。在之前的介绍中，ThreadPool实现的线程池，其每个线程都有机会处理工作队列的任意一个任务。这就会导致一个问题，如果任务之间有互斥性，那么正在处理该任务的两个线程有一个必须等待另一个处理完成后才能处理，从而导致线程的阻塞，性能下降。

例2-1 如表2-1所示，线程Thread1和Thread2分别正在处理Job1和Job2。

由于Job1和Job2的关联性，二者不能并发执行，只能顺序执行，二者之间用一个互斥锁来控制。如果Thread1先获得互斥锁就先执行，Thread2必须等待，直到Thread1执行完Job1后释放了该互斥锁，Thread2获得该互斥锁后才能执行Job2。显然，这种任务的调度方式应对这种不能完全并行的任务是有缺陷的。实际上Thread2可以去执行其他任务，比如Job5。Job1和Job2既然是顺序的，就都可以交给Thread1执行。

表2-1 ThreadPool的处理模型示例

线程\任务	Job1	Job2	Job3	Job4	Job5
Thread1	*				
Thread2		*			
Thread3				*	
Thread4			*		

因此，引入了Sharded ThreadPool进行管理。ShardedThreadPool对上述的任务调度方式做了改进，其在线程的执行函数里，添加了表示线程的thread_index：

```
void shardedthreadpool_worker(uint32_t thread_index);
```

具体如何实现Shard方式，还需要使用者自己去实现。其基本的思想就是：每个线程对应一个任务队列，所有需要顺序执行的任务都放在同一个线程的任务队列里，全部由该线程执行。

2.4 Finisher

类Finisher用来完成回调函数Context的执行，其内部有一个FinisherThread线程来用于执行Context回调函数：

```
class Finisher {  
    ....  
  
    vector<Context*> finisher_queue;  
    // 需要执行的  
  
    Context, 成功返回值为  
  
    0  
    list<pair<Context*,int> > finisher_queue_rval; //  
  
    // 需要执行的  
  
    Context, 返回值为  
  
    int类型的有效值  
  
    ....  
}
```

2.5 Throttle

类Throttle用来限制消费的资源数量（也常称为槽位“slot”），当请求的slot数量达到max值时，请求就会被阻塞，直到有新的槽位释放出来，代码如下：

```
class Throttle {
    CephContext *cct;
    const std::string name;
    PerfCounters *logger;
    ceph::atomic_t count, max;
    // count:当前占用的

    slot的数量

    // max:sloct数量的最大值

    Mutex lock;          //等待的锁

    list<Cond*> cond;  //等待的条件变量

    .....

}
```

函数get用于获取数量为c个slot，参数c默认为1，参数m默认为0，如果m不为默认的0值，就用m值重新设置slot的max值。如果成功获取数量为c个slot，就返回true，否则就阻塞等待。例如：

```
bool Throttle::get(int64_t c, int64_t m)
```

函数get_or_fail当获取不到数量为c个slot时，就直接返回false，不阻塞

等待：

```
bool Throttle::get_or_fail(int64_t c)
```

函数put用于释放数量为c个slot资源：

```
int64_t Throttle::put(int64_t c)
```

2.6 SafeTimer

类SafeTimer实现了定时器的功能，代码如下：

```
class SafeTimer
{
    CephContext *cct;
    Mutex& lock;
    Cond cond;
    bool safe_callbacks;      //是否是
    safe_callbacks
    SafeTimerThread *thread; //定时器执行线程

    std::multimap<utime_t, Context*> schedule;
                                //目标时间和定时任务执行函数

    Context
    std::map<Context*, std::multimap<utime_t, Context*>::iterator> events;
                                //定时任务

    <-->定时任务在
    schedule中的位置映射

    bool stopping;           //是否停止
}
```

添加定时任务的命令如下：

```
void SafeTimer::add_event_at(utime_t when, Context *callback)
```

取消定时任务的命令如下：

```
bool cancel_event(Context *callback);
```

定时任务的执行如下：

```
void SafeTimer::timer_thread()
```

本函数一次检查scheduler中的任务是否到期，其循环检查任务是否到期执行。任务在schedule中是按照时间升序排列的。首先检查，如果第一任务没有到时间，后面的任务就不用检查了，直接终止循环。如果第一任务到了定时时间，就调用callback函数执行，如果是safe_callbacks，就必须在获取lock的情况下执行Callback任务。

2.7 本章小结

本章介绍了src/common目录下的一些公共库中比较常见的类的实现。BufferList在数据读写、序列化中使用比较多，它的各种不同成员函数的使用方法需要读者自己进一步了解。对于ShardedThreadPool，本章只介绍了实现的原理，具体实现在不同的场景会有不同，需要读者面对具体的代码自己去分析。

第3章 Ceph网络通信

本章介绍Ceph网络通信模块，这是客户端和服务器通信的底层模块，用来在客户端和服务器之间接收和发送请求。其实现功能比较清晰，是一个相对较独立的模块，理解起来比较容易，所以首先介绍它。

3.1 Ceph网络通信框架

一个分布式存储系统需要一个稳定的底层网络通信模块，用于各节点之间的互联互通。对于一个网络通信系统，要求如下：

- 高性能。** 性能评价的两个指标：带宽和延迟。

- 稳定可靠。** 数据不丢包，在网络中断时，实现重连等异常处理。

网络通信模块的实现在源代码src/msg的目录下，其首先定义了一个网络通信的框架，三个子目录里分别对应：Simple、Async、XIO三种不同的实现方式。

Simple是比较简单的，目前比较稳定的实现，系统默认的用于生产环境的方式。它最大的特点是：每一个网络链接，都会创建两个的线程，一个专门用于接收，一个专门用于发送。这种模式实现比较简单，但是对于大规模的集群部署，大量的链接会产生大量的线程，会消耗CPU资源，影响性能。

Async模式使用了基于事件的I/O多路复用模式。这是目前网络通信中广泛采用的方式，但是在Ceph中，官方宣称这种方式还处于试验阶段，不够稳定，还不能用于生产环境。

XIO方式使用了开源的网络通信库accelio来实现。这种方式需要依赖第三方的库accelio稳定性，需要对accelio的使用方式以及代码实现都比较

熟悉。目前也处于试验阶段。特别注意的是，前两种方式只支持TCP/IP协议，XIO可以支持Infiniband网络。

在msg目录下定义了网络通信的抽象框架，它完成了通信接口和具体实现的分离。在其下分别有msg/simple子目录、msg/Async子目录、msg/xio子目录，分别对应三种不同的实现。

3.1.1 Message

类Message是所有消息的基类，任何要发送的消息，都要继承该类，格式如图3-1所示。

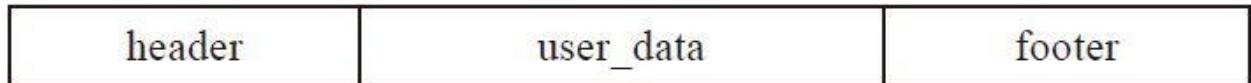


图3-1 消息发送格式

消息的结构如下：header是消息头，类似一个消息的信封(envelope)，user_data是用于要发送的实际数据，footer是一个消息的结束标记，如下所示：

```
class Message : public RefCountedObject {
    ceph_msg_header  header;           // 消息头

    ceph_msg_footer  footer;          // 消息尾

    // 用户数据

    bufferlist      payload;         // "front" unaligned blob
    bufferlist      middle;          // "middle" unaligned blob
    bufferlist      data;            // data payload (page-alignment will be preserved)
    // 消息相关的时间戳

    utime_t  recv_stamp;             // 开始接收数据的时间戳

    utime_t  dispatch_stamp;        // dispatch 的时间戳

    utime_t  throttle_stamp;        // 获取
    throttle 的
    slot 的时间戳
```

```
utime_t recv_complete_stamp; //接收完成的时间戳

ConnectionRef connection; //网络连接类

uint32_t magic; //消息的魔术字

bi::list_member_hook<> dispatch_q; //boost::intrusive需要的字段

}
```

下面分别介绍其中的重要参数。

ceph_msg_header为消息头，它定义了消息传输相关的元数据：

```
struct ceph_msg_header {
    __le64 seq; //当前
    session内消息的唯一序号

    __le64 tid; //消息的全局唯一的
    id
    __le16 type; //消息类型

    __le16 priority; //优先级

    __le16 version; //消息编码的版本

    __le32 front_len; // payload的长度

    __le32 middle_len; // middle的长度

    __le32 data_len; // data的长度

    __le16 data_off; //对象的数据偏移量

    struct ceph_entity_name src; //消息源

    //一些旧的代码，用于兼容，如果为零就忽略
}
```

```
    __le16 compat_version;
    __le16 reserved;
    __le32 crc;           //消息头的  
crc32c校验信息

} __attribute__ ((packed));
```

ceph_msg_footer为消息的尾部，附加了一些crc校验数据和消息结束标志：

```
struct ceph_msg_footer {
    __le32 front_crc, middle_crc, data_crc;
        //分别对应  
crc效验码

    __le64 sig;           //消息的  
64位

    signature
    __u8 flags;          //结束标志

} __attribute__ ((packed));
```

消息带的数据分别保存在payload、middle、data这三个bufferlist中。payload一般保存Ceph操作相关的元数据，middle目前没有使用到，data一般为读写的数据。

在源代码src/messages下定义了系统需要的相关消息，其都是Message类的子类。

3.1.2 Connection

类Connection对应端（port）对端的socket链接的封装。其最重要的接口是可以发送消息：

```
struct Connection : public RefCountedObject {
    mutable Mutex lock;           //锁保护

Connection的所有字段

    Messenger *msgr;
    RefCountedObject *priv;      //链接的私有数据

    int peer_type;               //链接的
peer类型

    entity_addr_t peer_addr;     //peer的地址

    utime_t last_keepalive, last_keepalive_ack;
    //最后一次发送

keepalive的时间和最后一次接收

keepalive的

ACK的时间

private:
    uint64_t features;          //一些
feature的标志位

public:
    bool failed;                //当值为
true时，该链接为

lossy链接已经失效了

    int rx_buffers_version;     //接收缓存区的版本
```

```
map<ceph_tid_t,pair<bufferlist,int> > rx_buffers; //接收缓冲区  
  
消息的标识  
ceph_tid --> (buffer, rx_buffers_version)的映射  
  
}
```

其最重要的功能就是发送消息的接口：

```
virtual int send_message(Message *m) = 0;
```

3.1.3 Dispatcher

类Dispatcher是消息分发的接口，其分发消息的接口为：

```
virtual bool ms_dispatch(Message *m) = 0;  
virtual void ms_fast_dispatch(Message *m);
```

Server端注册该Dispatcher类用于把接收到的Message请求分发给具体处理的应用层。Client端需要实现一个Dispatcher函数，用于处理收到的ACK应对消息。

3.1.4 Messenger

Messenger是整个网络抽象模块，定义了网络模块的基本API接口。网络模块对外提供的基本功能，就是能在节点之间发送和接受消息。

向一个节点发送消息的命令如下：

```
virtual int send_message(Message *m, const entity_inst_t& dest) = 0;
```

注册一个Dispatcher用来分发消息的命令如下：

```
void add_dispatcher_head(Dispatcher *d)
```

3.1.5 网络连接的策略

Policy定义了Messenger处理Connection的一些策略：

```
struct Policy {
    bool lossy;           //如果为
    true, 该当该连接出现错误时就删除

    bool server;          //如果为
    true, 为服务端，都是被动连接

    bool standby;         //如果为
    true, 该连接处于等待状态

    bool resetcheck;      //如果为
    true, 该连接出错后重连

    //该
    connection相关的流控操作

    Throttle *throttler_bytes;
    Throttle *throttler_messages;
    //本地端的一些

    feature标志

    uint64_t features_supported;
    //远程端需要的一些

    feature标志

    uint64_t features_required;
}
```

3.1.6 网络模块的使用

通过下面最基本的服务器和客户端的实例程序，了解如何调用网络通信模块提供的接口来完成收发请求消息的功能。

1.Server程序分析

Server程序源代码在test/simple_server.cc里，这里只展示有关网络部分的核心流程。

1) 调用Messenger的函数create创建一个Messenger的实例，配置选项g_conf->ms_type为配置的实现类型，目前有三种方式：simple、async、xio：

```
messenger = Messenger::create(g_ceph_context,
    g_conf->ms_type, entity_name_t::MON(-1),
    "simple_server",
    0 /* nonce */);
```

2) 设置Messenger的属性：

```
messenger->set_magic(MSG_MAGIC_TRACE_CTR);
messenger->set_default_policy(
    Messenger::Policy::stateless_server(CEPH_FEATURES_ALL, 0));
```

3) 对于Server，需要bind服务端地址：

```
r = messenger->bind(bind_addr);
if (r < 0)
```

```
    goto out;
common_init_finish(g_ceph_context);
```

4) 创建一个Dispatcher，并添加到Messenger：

```
dispatcher = new SimpleDispatcher(messenger);
messenger->add_dispatcher_head(dispatcher);
```

5) 启动Messenger：

```
messenger->start();
messenger->wait(); //本函数必须等
```

start完成才能调用

SimpleDispatcher函数里实现了ms_dispatch，用于把接收到的各种请求消息分发给相关的处理函数。

2.Client程序分析

源代码在test/simple_client.cc里，这里只展示有关网络部分的核心流程。

1) 调用Messenger的函数create创建一个Messenger的实例：

```
messenger = Messenger::create(g_ceph_context, g_conf->ms_type,
                           entity_name_t::MON(-1),
                           "client",
                           getpid());
```

2) 设置相关的策略：

```
messenger->set_magic(MSG_MAGIC_TRACE_CTR);
messenger->set_default_policy(Messenger::Policy::lossy_client(0, 0));
```

3) 创建Dispatcher类并添加，用于接收消息：

```
dispatcher = new SimpleDispatcher(messenger);
messenger->add_dispatcher_head(dispatcher);
dispatcher->set_active();
```

4) 启动消息：

```
r = messenger->start();
if (r < 0)
    goto out;
```

5) 下面开始发送请求，先获取目标Server的链接：

```
conn = messenger->get_connection(dest_server);
```

6) 通过Connection来发送请求消息。这里的消息发送方式都是异步发送，接收到请求消息的ACK应答消息后，将在Dispatcher的ms_dispatch或者ms_fast_dispatch处理函数里做相关的处理：

```
Message *m;
for (msg_ix = 0; msg_ix < n_msgs; ++msg_ix) {
    //如果需要，这里要添加实际的数据

    if (! n_dsize) {
        m = new MPing();
    } else {
        m = new_simple_ping_with_data("simple_client", n_dsize);
    }
    conn->send_message(m);
}
```

综上所述，通过Ceph的网络框架发送消息比较简单。在Server端，只需要创建一个Messenger实例，设置相应的策略并绑定服务端口，然后就设置一个Dispatcher来处理接收到的请求。在Client端，只需要创建一个Messenger实例，设置相关的策略和Dispatcher用于处理返回的应答消息。通过获取对应Server的connection来发送消息即可。

3.2 Simple实现

Simple在Ceph里实现比较早，目前也比较稳定，是在生产环境中使用的网络通信模块。如其名字所示，实现相对比较简单。下面具体分析一下，Simple如何实现Ceph网络通信框架的各个模块。

3.2.1 SimpleMessenger

类SimpleMessenger实现了Messenger接口。

```
class SimpleMessenger : public SimplePolicyMessenger {
    Acceptor accepter;      //用于接受客户端的链接请求

    DispatchQueue dispatch_queue; //接收到的请求的消息分发队列

    bool did_bind;      //是否绑定

    __u32 global_seq; //生成全局的消息

    seq
        ceph_spinlock_t global_seq_lock; //用于保护

    global_seq
        //地址-

    pipe映射

    ceph::unordered_map<entity_addr_t, Pipe*> rank_pipe;
    //正在处理的

    pipes
        set<Pipe*> accepting_pipes;
        //所有的

    pipes
        set<Pipe*>      pipes;
        //准备释放的

    pipes
        list<Pipe*>      pipe_reap_queue;
        //内部集群的协议版本

    int cluster_protocol;
}
```

3.2.2 Acceptor

类Acceptor用来在Server端监听端口，接收链接，它继承了Thread类，本身是一个线程，来不断地监听Server的端口：

```
class Acceptor : public Thread {
    SimpleMessenger *msgr;
    bool done;
    int listen_sd;    //监听的端口

    uint64_t nonce;
    .....

}
```

3.2.3 DispatchQueue

DispatchQueue类用于把接收到的请求保存在内部，通过其内部的线程，调用SimpleMessenger类注册的Dispatch类的处理函数来处理相应的消息：

```
class DispatchQueue {
    .....
    mutable Mutex lock;
    Cond cond;
    class QueueItem {
        int type;
        ConnectionRef con;
        MessageRef m;
        .....
    } ;

    PrioritizedQueue<QueueItem, uint64_t> mqueue;      //接收消息的优先队列

    set<pair<double, Message*> > marrival;
    //接收到的消息集合

    pair为

    (recv_time, message)
    map<Message *, set<pair<double, Message*> >::iterator> marrival_map;
    //消息-所在集合位置的映射

    .....

};
```

其内部的mqueue为优先级队列，用来保存消息，marrival保存了接收到的消息。marrival_map保存消息在集合中的位置。

函数DispatchQueue :: enqueue用来把接收到的消息添加到消息队列

中，函数DispatchQueue::entry为线程的处理函数，用于处理消息。

3.2.4 Pipe

类Pipe实现了PipeConnection的接口，它实现了两个端口之间的类似管道的功能。

对于每一个pipe，内部都有一个Reader和一个Writer线程，分别用来处理这个Pipe有关的消息接收和请求的发送。线程DelayedDelivery用于故障注入测试：

```
class Pipe : public RefCountedObject {
    class Reader : public Thread {
        ...
    } reader_thread;
    //接收线程，用于接收数据

    class Writer : public Thread {
        ...
    } writer_thread;
    //发送线程，用于发送数据

    SimpleMessenger *msgr;           // msgr的指针

    uint64_t conn_id;                //分配给
                                    Pipe自己唯一的
    id
    char *recv_buf;                 //接收缓存区

    int recv_max_prefetch;          //接收缓冲区一次预取的最大值

    int recv_ofs;                   //接收的偏移量

    int recv_len;                   //接收的长度
```

```

int sd;                                // pipe对应的

socked fd
struct iovec msgvec[IOV_MAX]; //发送消息的

iovec结构

int port;                               //连接端口

int peer_type;                          //连接对方的类型

entity_addr_t peer_addr;                //对方地址

Messenger::Policy policy;              //策略

Mutex pipe_lock;
int state;                             //当前链接的状态

atomic_t state_closed;                 //如果非

0, 那么状态为

STATE_CLOSED
PipeConnectionRef connection_state;    //PipeConnection的引用

utime_t backoff;                      // backoff的时间

map<int, list<Message*>> out_q;   //准备发送的消息优先队列

DispatchQueue *in_q;                  //接收消息的

DispatchQueue list<Message*> sent;   //要发送的消息

Cond cond;
bool send_keepalive;
bool send_keepalive_ack;
utime_t keepalive_ack_stamp;
bool halt_delivery;                  //如果

Pipe队列销毁, 停止增加

__u32 connect_seq, peer_global_seq;
uint64_t out_seq;                    //发送消息的序列号

uint64_t in_seq, in_seq_acked;       //接收到消息序号和

```

ACK的序号

}

3.2.5 消息的发送

1) 当发送一个消息时，首先要通过Messenger类，获取对应的Connection：

```
conn = messenger->get_connection(dest_server);
```

具体到SimpleMessenger的实现如下所示：

- a) 首先比较，如果dest.addr是my_inst.addr，就直接返回local_connection。
 - b) 调用函数_lookup_pipe在已经存在的Pipe中查找。如果找到，就直接返回pipeConnectionRef；否则调用函数connect_rank新创建一个Pipe，并加入到msgr的register_pipe里。
- 2) 当获得一个Connection之后，就可以调用Connection的发送函数来发送消息。

```
conn->send_message(m);
```

其最终调用了SimpleMessenger::submit_message函数：

- a) 如果Pipe不为空，并且状态不是Pipe::STATE_CLOSED状态，调用函数pipe->_send把发送的消息添加到out_q发送队列里，触发发送线程。

b) 如果Pipe为空，就调用connect_rank创建Pipe，并把消息添加到out_q发送队列中。

3) 发送线程writer把消息发送出去。通过步骤2，要发送的消息已经保存在相应Pipe的out_q队列里，并触发了发送线程。每个Pipe的Writer线程负责发送out_q的消息，其线程入口函数为Pipe::writer，实现功能：

- a) 调用函数_get_next_outgoing从out_q中获取消息。
- b) 调用函数write_message(header, footer, blist) 把消息的header、footer、数据blist发送出去。

3.2.6 消息的接收

1) 每个Pipe对应的线程Reader用于接收消息。入口函数为Pipe`::reader`, 其功能如下：

a) 判断当前的state, 如果为STATE_ACCEPTING, 就调用函数Pipe`::accept`来接收连接, 如果不是STATE_CLOSED, 并且不是STATE_CONNECTING状态, 就接收消息。

b) 先调用函数tcp_read来接收一个tag。

c) 根据tag, 来接收不同类型的消息如下所示：

- CEPH_MSGR_TAG_KEEPALIVE消息。

- CEPH_MSGR_TAG_KEEPALIVE2, 在CEPH_MSGR_TAG_KEEPALIVE的基础上, 添加了时间。

- CEPH_MSGR_TAG_KEEPALIVE2_ACK。

- CEPH_MSGR_TAG_ACK。

- CEPH_MSGR_TAG_MSG, 这里才是接收的消息。

- CEPH_MSGR_TAG_CLOSE。

d) 调用函数read_message来接收消息，当本函数返回后，就完成了接收消息。

2) 调用函数in_q->fast_preprocess (m) 预处理消息。

3) 调用函数in_q->can_fast_dispatch (m) , 如果可以进行 fast_dispatch, 就in_q->fast_dispatch (m) 处理。fast_dispatch并不把消息加入到mqueue里，而是直接调用msgr->ms_fast_dispatch函数，并最终调用注册的fast_dispatcher函数处理。

4) 如果不能fast_dispatch, 就调用函数in_q->enqueue (m, m->get_priority () , conn_id) 把接收到的消息加入到DispatchQueue的mqueue队列里，由DispatchQueue的分发线程调用ms_dispatch处理。

ms_fast_dispatch和ms_dispatch两种处理的区别在于：ms_dispatch是由 DispatchQueue的线程处理的，它是一个单线程；ms_fast_dispatch函数是由 Pipe的接收线程直接调用处理的，因此性能比前者要好。

3.2.7 错误处理

网络模块复杂的功能是如何处理网络错误。无论是接收还是发送，会出现各种异常错误，包括返回异常错误码，接收数据的magic验证不一致，接收的数据的效验验证不一致，等等。错误的原因主要是由于网络本身的错误（物理链路等），或者字节跳变引起的。

目前错误处理的方法比较简单，处理流程如下：

- 1) 关闭当前socket的连接。
- 2) 重新建立一个socket连接。
- 3) 重新发送没有接受到ACK应对的消息。

函数Pipe::fault用来处理错误：

- 1) 调用shutdown_socket关闭pipe的socket。
- 2) 调用函数requeue_sent把没有收到ACK的消息重新加入发送队列，当发送队列有请求时，发送线程会不断地尝试重新连接。

3.3 本章小结

本章介绍了Ceph的网络通信模块的框架，及目前生产环境中使用的Simple实现。它对每个链接都会有一个发送线程和接收线程用来处理发送和接收。实现的难点还在于网络链接出现错误时的各种错误处理。

第4章 CRUSH数据分布算法

本章介绍Ceph的数据分布算法CRUSH，它是一个相对比较独立的模块，和其他模块的耦合性比较少，功能比较清晰，比较容易理解。在客户端和服务器都有CRUSH的计算，了解它可以更好地理解后面的章节。

CRUSH算法解决了PG的副本如何分布在集群的OSD上的问题。本章首先介绍CRUSH算法的原理，并给出相应的示例，然后进一步分析其实现的一些核心代码。

4.1 数据分布算法的挑战

存储系统的数据分布算法要解决数据如何分布到集群中的各个节点和磁盘上，其面临如下的挑战：

·**数据分布和负载的均衡。** 首先是数据分布均衡，使数据能均匀地分别在各个节点和磁盘上。其次是负载均衡，使数据访问（读写等操作）的负载在各个节点和磁盘上的负载均衡。

·**灵活应对集群伸缩。** 系统可以方便地增加或者删除存储设备（包括节点和设备失效的处理）。当增加或者删除存储设备后，能自动实现数据的均衡，并且迁移的数据尽可能地少。

·**支持大规模集群。** 为了支持大规模的存储集群，就要求数据分布算法维护的元数据相对较小，并且计算量不能太大。随着集群规模的增加，数据分布算法的开销比较小。

在分布式存储系统中，数据分布算法对于分布式存储系统至关重要。目前有两种基本实现方法，一种是基于集中式的元数据查询的方式，如HDFS的实现：文件的分布信息（layout信息）是通过访问集中式元数据服务器获得；另一种是基于分布式算法以计算获得。例如一致性哈希算法（DHT）等。Ceph的数据分布算法CRUSH就属于后者。

4.2 CRUSH算法的原理

CRUSH算法的全称为：Controlled、Scalable、Decentralized Placement of Replicated Data，可控的、可扩展的、分布式的副本数据放置算法。

由第1章中介绍过的RADOS对象寻址过程可知，CRUSH算法解决PG如何映射到OSD列表中。其过程可以看成函数：

```
CRUSH(X) ->
(OSDi, OSDj, OSDk)
```

输入参数：

- X为要计算的PG的pg_id。
- Hierarchical Cluster Map为Ceph集群的拓扑结构。
- Placement Rules为选择策略。

输出一组可用的OSD列表。

下面将分别详细介绍Hierarchical Cluster Map的定义和组织方式。

Placement rules定义了副本选择的规则。最后介绍Bucket随机选择算法的实现。

4.2.1 层级化的Cluster Map

层级化的Cluster Map定义了OSD集群具有层级关系的静态拓扑结构。OSD的层级使得CRUSH算法在选择OSD时实现了机架感知（rack awareness）的能力，也就是通过规则定义，使得副本可以分布在不同的机架、不同的机房中，提供数据的安全性。

层级化的Cluster Map的一些基本概念如下：

- Device：最基本的存储设备，也就是OSD，一个OSD对应一个磁盘存储设备。
- bucket：设备的容器，可以递归的包含多个设备或者子类型的bucket。 bucket的类型：bucket可以有很多的类型，例如host就代表了一个节点，可以包含多个device。Rack就是机架，包含多个host等。在Ceph里默认的有root、datacenter、room、row、rack、host六个等级。用户也可以自己定义新的类型。每个device都设置了自己的权重，和自己的存储空间相关。bucket的权重就是子bucket（或者设备）的权重之和。

下列举例说明bucket的用法。

例4-1 Cluster Map的定义

```
host test1 { //类型
    host, 名字为
```

```

test1
  id -2          // bucket的

id, 一般为负值

# weight 3.000      //权重, 默认为子

item的权重之和

alg straw          // bucket随机选择的算法

hash 0            // bucket随机选择的算法使用的

hash函数, 这里

0代表使用

hash函数

jenkins1
  item osd.1 weight 1.000    // item1:

osd.1和权重值

  item osd.2 weight 1.000
  item osd.3 weight 1.000
}
host test2{
  id -3
  # weight 3.000
  alg straw
  hash 0
  item osd.3 weight 1.000
  item osd.4 weight 1.000
  item osd.5 weight 1.000
}
root default{        // root类型的

bucket, 名字为

default
  id -1          // id号

# weight 6.000
alg straw          //随机选择的算法

hash 0            // rjenkins1
item test1 weight 3.000
item test2 weight 3.000
}

```

根据上面Cluster Map的语法定义，图4-1给出了比较直观的层级化的树型结构。

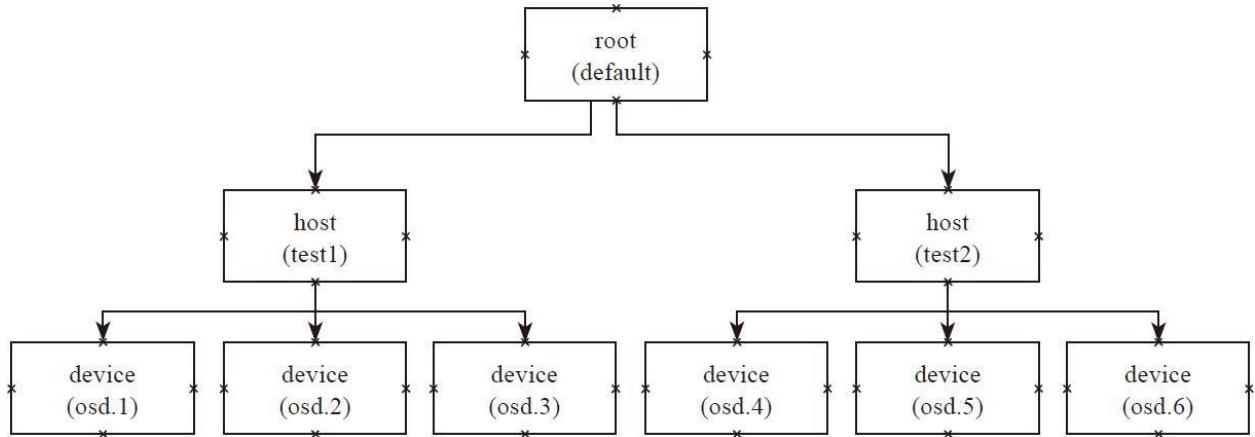


图4-1 Cluster Map示例图

在上面的Cluster Map定义中：

- 有一个root类型的bucket，名字为default。
- root下面有两个host类型的bucket，名字分别为test1和test2，其下分别有三个osd设备，每个device的权重都为1.000，说明它们的容量大小都相同。host的权重为子设备之和为3.000，它是自动计算的，不需要设置。
- Hash设置了使用的hash函数，值0代表使用rjenkins1函数。
- alg代表在该bucket里选择子item的算法。

4.2.2 Placement Rules

Cluster Map反映了存储系统层级的物理拓扑结构。Placement Rules决定了一个PG的对象副本如何选择的规则，通过这些可以自己设定规则，用户可以设定副本在集群中的分布。其定义格式如下：

```
tack(a)
choose
  choose firstn {num} type {bucket-type}
  chooseleaf firstn {num} type {bucket-type}.

  If {num} == 0, choose pool-num-replicas buckets (all available).
  If {num} > 0 && < pool-num-replicas, choose that many buckets.
  If {num} < 0, it means pool-num-replicas - {num}.
Emit
```

Placement Rules的执行流程如下：

- 1) take操作选择一个bucket，一般是root类型的bucket。
- 2) choose操作有不同的选择方式，其输入都是上一步的输出：
 - a) choose firstn深度优先选择出num个类型为bucket-type个的子bucket。
 - b) chooseleaf先选择出num个类型为bucket-type个子bucket，然后递归到页节点，选择一个OSD设备：
 - 如果num为0， num就为pool设置的副本数。

·如果num大于0， 小于pool的副本数， 那么就选择出num个。

·如果num小于0， 就选择出pool的副本数减去num的绝对值。

3) emit输出结果。

操作chooseleaf firstn{num}type{bucket-type}可以等同于两个操作：

a) choose firstn{num}type{bucket-type}

b) choose firstn 1 type osd

例4-2 Placement Rules：三个副本分布在三个Cabinet中。

如图4-2所示的Cluster Map：顶层是一个root bucket， 每个root下有四个row类型bucket。每个row下面有4个cabinet， 每个cabinet下有若干个OSD设备（图中有4个host， 每个host有若干个OSD设备， 但是在本crush map中并没有设置host这一级别的bucket， 而是直接把4个host上的所有OSD设备定义为一个cabinet）：

```
rule replicated_ruleset {
    ruleset 0                         // ruleset的编号
    id
    type replicated                   //类型
    replicated或者
    erasure code
    min_size 1                        //副本数最小值

    max_size 10                       //副本数最大值

    step take root                   //选择一个
```

root bucket, 做下一步的输入

```
step choose firstn 1 type row    //选择一个  
row, 同一排  
  
step choose firstn 3 type cabinet //选择三个  
cabinet, 三副本分别在不同的  
cabinet  
  step choose firstn 1 type osd   //在上一步输出的三个  
cabinet中, 分别选择一个  
  
osd  
  step emit  
}
```

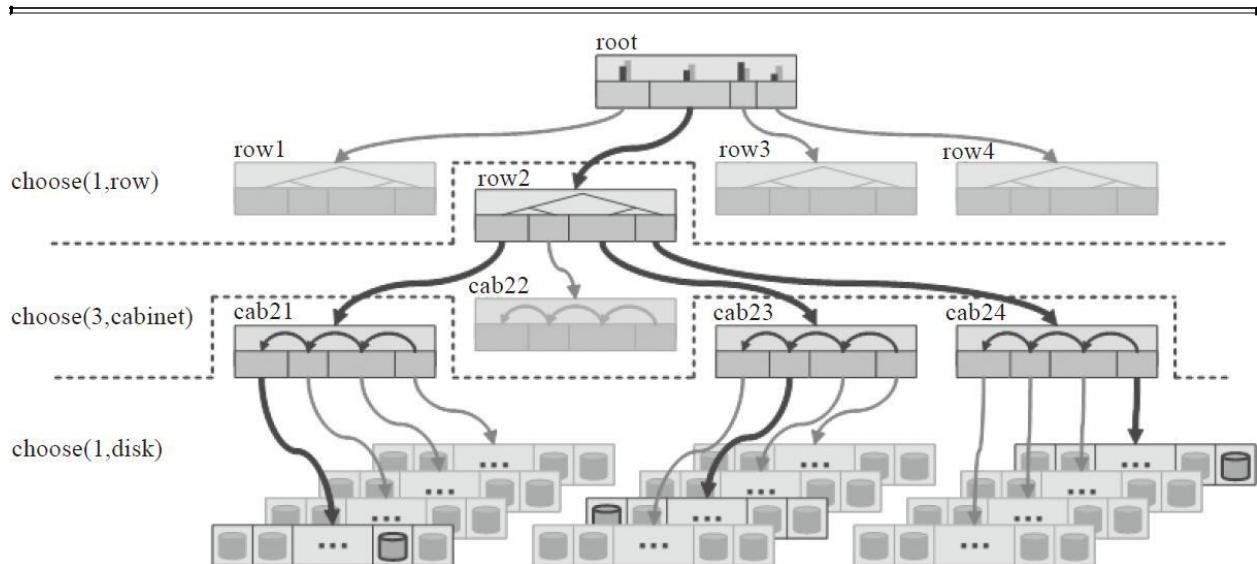


图4-2 例4-2 Cluster Map

根据上面的定义和图4-2的Cluster Map所示，选择算法的执行过程如下：

- 1) 选中root bucket作为下一个步骤的输入。
- 2) 从root类型的bucket中选择一个row类的子bucket, 其选择的算法在

root的定义中设置，一般设置为straw算法。

- 3) 从上一步的输出row中，选择三个cabinet，其选择的算法在row中定义。
- 4) 从上一步输出的三个cabinet中，分别选择一个OSD，并输出。

根据本rule sets，选择出三个OSD设备分布在一个row上的三个cabinet中。

例4-3 Placement Rules：主副本分布在SSD上，其他副本分布在HDD上。

如图4-3所示的Cluster Map：定义了两个root类型的bucket，一个命名为SSD的root类型的bucket，其OSD存储介质都是SSD盘。它包含两个host，每个host上的设备都是SSD磁盘；另一个是名为HDD的root类型的bucket，其OSD的存储介质都是HDD磁盘，它有两个host，每个host上的设备都是HDD磁盘。

```
rule ssd-primary {
    ruleset 5
    type replicated
    min_size 5
    max_size 10
    step take ssd           //选择
    ssd这个
    root bucket为输入

    step chooseleaf firstn 1 type host   //选择一个
    host，并递归选择叶子节点
```

```

osd
step emit //输出结果

step take hdd //选择
hdd这个

root bucket为输入

step chooseleaf firstn -1 type host
//选择总副本数减一个

host，并分别递归选择一个叶子节点

osd
step emit //输出结果

}

```

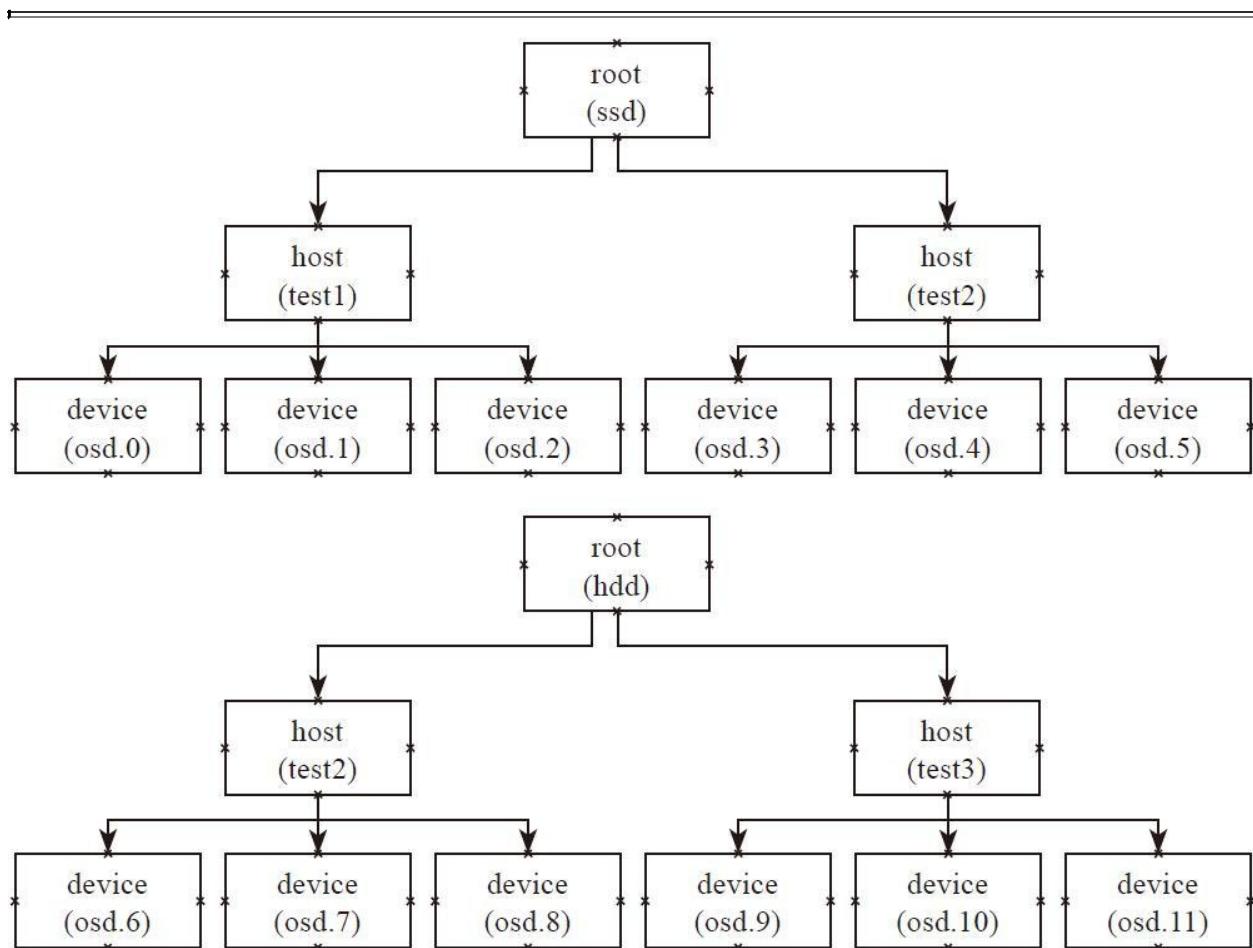


图4-3 例4-3 Cluster Map

根据图4-3所示的Cluster Map， 代码中的rulesets的执行过程如下：

- 1) 首先take操作选择ssd为root类型的bucket。
- 2) 在ssd的root中先选择一个host， 然后以该host为输入， 递归至叶子节点， 选择一个osd设备。
- 3) 输出选择的设备， 也就是ssd设备。
- 4) 选择hdd作为root的输入。
- 5) 选择2个host (副本数减一， 默认3副本)， 并分别递归选择一个OSD设备， 最终选择出两个hdd设备。
- 6) 输出最终的结果。

最终输出3个设备， 一个是SSD类型的磁盘， 另外两个是HDD磁盘。通过上述规则， 就可以把PG的主副本分布在SSD类型的OSD上， 其他副本分布在HDD类型的磁盘上。

4.2.3 Bucket随机选择算法

Bucket随机选择算法解决了如何从Bucket中选择出需要的子item问题。它定义了四种不同的Bucket选择算法，每种Bucket的选择算法基于不同的数据结构，采用不同伪随机选择函数。

在本节涉及的Hash函数，其参数分别为；

hash(x, r, i)

·x为要计算的PG的id。

·r为选择的副本序号。

·i为bucket的id号。

下面具体介绍Bucket的四种随机选择算法的过程，并介绍当选择算法出现冲突、失效或过载等特殊情况的处理。

1.Uniform Bucket

Uniform类型适用于每个item，具有相同权重，且item很少添加和删除，也就是item的数量比较固定。它用了伪随机排列算法。

2.List Bucket

List类型的Bucket中，其子item在内存中使用数据结构中的链表来保存，其所包含的item可以具有任意的权重。具体查找方法如下：

- 1) 从List Bucket的表头item开始查找，它先得到表头item的权重 W_h ，剩余链表中所有item的权重之和为 W_s 。
- 2) 根据本节提到的hash (x, r, i) 函数得到一个[0~1]的值 v ，假如这个值 v 在 $[0 \sim W_h/W_s]$ 之中，则选择表头item，并返回表头item的id值。
- 3) 否者继续遍历剩余的链表，继续递归选择。

通过上述介绍可知，List类型的Bucket查找复杂度是 $O(n)$ 。

3.Tree Bucket

Tree类型的Bucket其item的组织成树结构：每个item组成决策树的叶子节点。根节点和中间节点是虚节点，其权重等于左右子树的权重之和。由于item在叶子节点，所以每次选择只能走到叶子节点才能选择一个item出来。其具体查找方法如下：

- 1) 从该Tree bucket的root item（虚节点）开始遍历。
- 2) 它先得到节点的左子树的权重 W_l ，得到节点的权重 W_n ，然后根据哈希函数hash (x, r, i) 得到一个[0~1]的值 v ：
 - a) 如果值 v 在 $[0 \sim W_l/W_n]$ 之间，那么左子树中继续选择item。

- b) 否者在右子树中继续选择item。
- c) 继续遍历子树，直到到达叶子节点，叶子节点item为最终选出的一个结果。

由上述过程可知，Tree Bucket每次选择一个item都要遍历到子节点。其查找复杂度是 $O(\log n)$ 。

4. Straw Bucket

Straw类的Bucket为默认的选择算法。该Bucket中的item选中概率是相同的，其实现如下：

- 1) 函数 $f(w_i)$ 为和item的权重 w_i 相关的函数，决定了每个item被选中的概率。
- 2) 给每个item计算出一个长度，其计算公式为：

```
length = f(wi)*hash(x, r, i)
```

length值最大的item就是被选中的item。

5. Bucket选择算法的对比

如表4-1所示。

表4-1 Bucket选择算法对比

Bucket 选择算法	选择的速度	item 添加难易程度	item 删除难易程度
uniform	O(1)	poor	poor
list	O(n)	optimal	poor
tree	O(log n)	Good	Good
straw	O(n)	Better	Better
straw2	O(n)	optimal	optimal

算法straw比较容易应对item的添加和删除，为默认的Bucket选择算法。算法straw2是对算法straw的一些改进，可以减少数据的迁移数量。

6.冲突、失效或者过载

当通过上述Bucket选择算法选出一个OSD后，有可能出现冲突（重复选择），该OSD已经失效了，或者过载（负载过重）的情况，就需要重新选择一次。

根据上述算法分析，选择时都依赖哈希函数：

hash(x, r, i)

其中，x为PG的id，r为选择的副本数，i为Bucket的id号。当选择出现上述情况需要重新选择。上述各种Bucket选择算法都依赖hash函数。当重新选择时，把参数r顺序增加即可通过上述hash函数重新计算一个新的hash值。

例4-4 冲突选择过程

过程见表4-2。

表4-2 冲突选择过程示例

	r=0	r=1	r=2	r=3
pg 1.0	osd1	osd2	osd3	
pg 1.1	osd2	osd2	osd3	osd4

说明如下：

- 1) pg 1.0根据副本r分别等于0、1、2来计算hash (x, r, i)，处理OSD列表{osd1, osd2, osd3}。
- 2) pg 1.1根据同样的方法：在r等于0时选择出了osd2，在r等于1时又选择了osd2，产生了冲突。这时就用r分别等于2, 3来继续选择剩余的副本。最终pg1.1选择出的OSD列表为{osd2, osd3, osd4}.

4.3 代码实现分析

在介绍了CRUSH算法的原理之后，下面就分析CRUSH算法实现的关键数据结构，并对算法具体实现函数进行分析。

4.3.1 相关的数据结构

CRUSH算法相关的数据结构有crush_map结构、crush_bucket结构和crush_rule结构，下面将详细介绍。

1.crush_map

结构crush_map定义了静态的所有Cluster Map的bucket。bucket为动态申请的二维数组，保存了所有的bucket结构。rules定义了所有的crush_rule结构：

```
struct crush_map {
    struct crush_bucket **buckets;
    struct crush_rule **rules;
    .....
}
```

2.crush_bucket

结构crush_bucket用于保存Bucket相关的信息：

```
struct crush_bucket {
    __s32 id;           // bucket的
    id, 一般为负值
    __u16 type;         //类型, 如果是
    0, 就是
    OSD 设备
```

```
__u8 alg;           // bucket的选择算法

__u8 hash;          // bucket的
hash函数

__u32 weight;      // bucket的权重

__u32 size;         // bucket下的
item的数量

__s32 *items;       // 子
bucket在
crush_bucket结构

buckets数组的下标，这里特别要注意的是，其子
item的
crush_bucket结构体都统一保存在
crush map结构中的
buckets数组中
，这里只保存其在数组中的下标

//以下是随机排序选择算法的一些
Cache的参数

__u32 perm_x;      //要选择的
x
__u32 perm_n;      //排列的总的元素

__u32 *perm;        //排列组合的结果

};
```

3.crush_rule

结构crush_rule

```
struct crush_rule {
    __u32 len;                                //steps的数组的长度

    struct crush_rule_mask mask;      //ruleset相关的配置参数

    struct crush_rule_step steps[0]; //操作步

};

struct crush_rule_mask {
    __u8 ruleset;                            //ruleset的编号

    __u8 type;                               //类型

    __u8 min_size;                          //最新

    size
    __u8 max_size;                          //最大

    size
};

struct crush_rule_step {
    __u32 op;                                //step操作步的操作码

    __s32 arg1;                             //如果是

take, 参数就是选择的

bucket的

id号

//如果是

select, 就是选择的数量

    __s32 arg2;                             //如果是

select, 是选择的类型

};
```

4.3.2 代码实现

代码builder.c和builder.h文件里主要实现了如何构造crush_map数据结构。Crush.c和Crush.h文件里定义了crush_map相关的数据结构和destroy方法。文件CrushCompiler.h和CrushCompiler.cc为crush map的词法和语法分析相关处理。类CrushWarpper是对CRUSH的所有核心实现进行的封装。CRUSH算法的核心实现在mapper.c文件里。

1.crush_do_rule

函数crush_do_rule里完成了CRUSH算法的选择过程：

```
int crush_do_rule(const struct crush_map *map,           //crush map结构
                  int ruleno,                 //ruleset的号
                  int x,                      //输入，一般是
                  pg的
                  id
                  int *result,                //输出
                  osd列表
                  int result_max,             //输出
                  osd列表的数量
                  const __u32 *weight,        //所有
                  osd的权重，通过它来判断
                  osd是否
                  out
                  int weight_max,             //所有
```

osd的数量

```
int *scratch)
```

函数cursh_do_rule根据step的数量，循环调用相关的函数选择bucket。如果是深度优先，就调用函数crush_choose_firstn，如果是广度优先，就调用函数crush_choose_indep来选择。

2.crush_choose_firstn

函数调用crush_bucket_choose选择需要的副本数，并对选择出来的OSD做了相关的冲突检查，如果冲突或者失效或者过载，继续选择新的OSD。

3.bucket算法

函数crush_bucket_choose根据不同的类型bucket，选择不同的算法来实现从bucket中选出item，这里介绍最常用的straw算法：

```
static int bucket_straw_choose(struct crush_bucket_straw *bucket,
                                int x, int r)
```

函数bucket_straw_choose用于straw类型的bucket的选择，输入参数x为pgid，r为副本数，其具体实现如下：

1) 对每个item，计算hash值：

```
draw = crush_hash32_3(bucket->h.hash, x, bucket->h.items[i], r);
```

2) 获取低16位，并乘以权重相关的修正值：

```
draw &= 0xffff;
draw *= bucket->straws[i];
```

3) 选取draw值最大的item为选中的item。

由上可知，这种算法类似抽签，是一种伪随机选择算法。

4.4 对CRUSH算法的评价

通过以上分析，可以了解到CRUSH算法实质是一种可分层确定性伪随机选择算法，它是Ceph分布式文件系统的一个亮点和创新。

优点如下：

- 输入元数据（cluster map、placement rule）较少，可以应对大规模集群。
- 可以应对集群的扩容和缩容。
- 采用以概率为基础的统计上的均衡，在大规模集群中可以实现数据均衡。

目前存在的缺点如下：

- 在小规模集群中，会有一定的数据不均衡现象。
- 增加新设备时，导致旧设备之间也有数据的迁移。

4.5 本章小结

本章介绍了RADOS的数据分布算法CRUSH的原理。Hierarchical Cluster Map实质定义了存储集群的静态拓扑结构。Placement Rules开放了数据副本的选择规则，可以由用户自己定义和编辑。Bucket算法定义了从bucket选择一个item的算法。通过本章可以了解CRUSH算法的具体实现，它实质是一个可分层的伪随机分布选择算法，是Ceph的一个创新，但它并不完美，需要许多改进。

第5章 Ceph客户端

本章介绍Ceph的客户端实现。客户端是系统对外提供的功能接口，上层应用通过它来访问Ceph存储系统。本章首先介绍Librados和Osdc两个模块，通过它们可直接访问RADOS对象存储系统。其次介绍ClS扩展模块使用它们可方便地扩展现有的接口。最后介绍Librbd模块。由于Librados和Librbd的多数实现流程都比较类似，本章在介绍相关的数据结构后，只选取一些典型的操作流程介绍。

5.1 Librados

Librados是RADOS对象存储系统访问的接口库，它提供了pool的创建、删除、对象的创建、删除、读写等基本操作接口。架构如图5-1所示。

在最上层是类RadosClient，它是Librados的核心管理类，处理整个RADOS系统层面以及pool层面的管理。类IoctxImpl实现单个pool层的对象读写等操作。OSDC模块实现了请求的封装和通过网络模块发送请求的逻辑，其核心类Objecter完成对象的地址计算、消息的发送等工作。

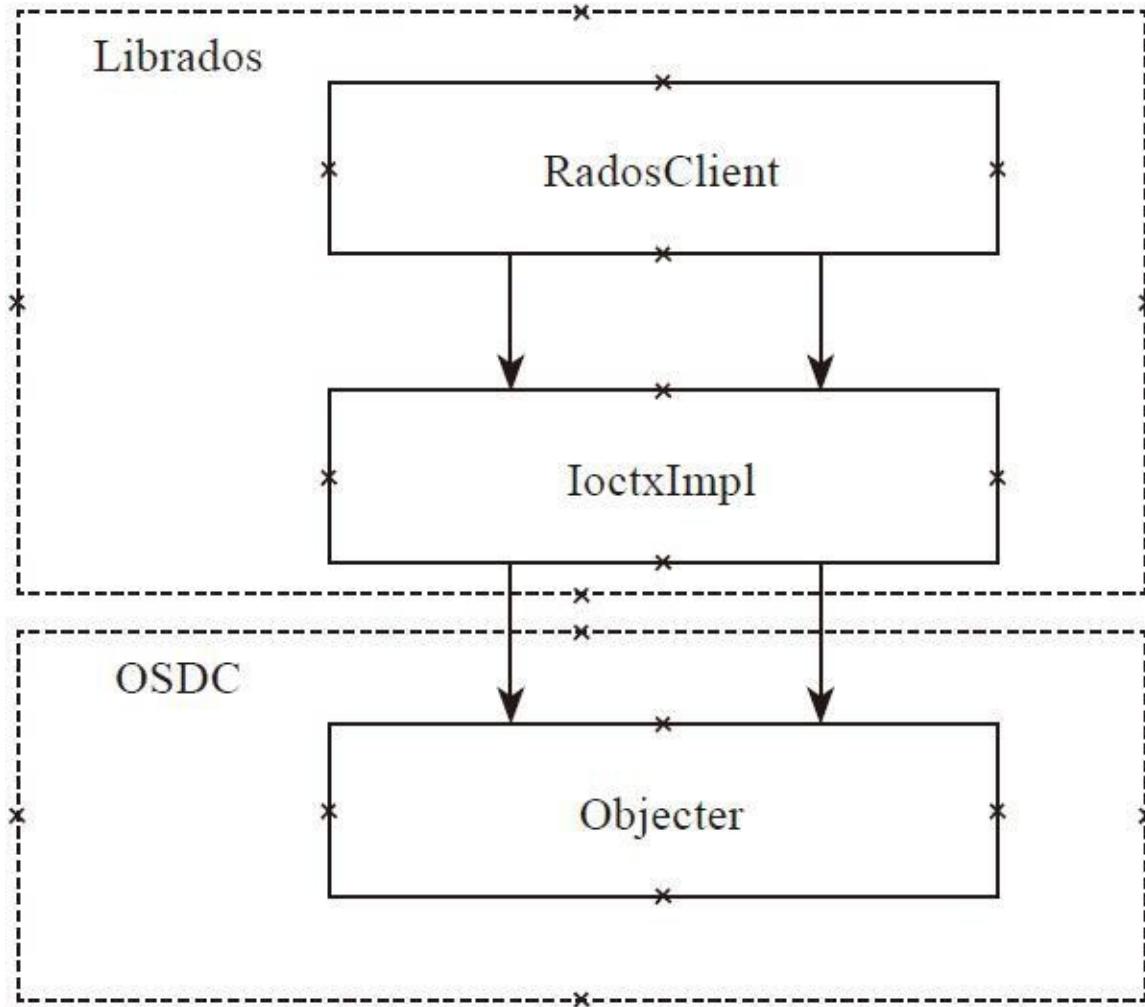


图5-1 Librados架构图

5.1.1 RadosClient

代码如下：

```
class librados::RadosClient : public Dispatcher
{
public:
    md_config_t *conf;           //配置文件

private:
    enum {
        DISCONNECTED,
        CONNECTING,
        CONNECTED,
    } state;                   //和

Monitor的网络连接状态

    MonClient monclient;        //Monitor客户端

    Messenger *messenger;      //网络消息接口

    uint64_t instance_id;       //rados客户端实例的

    id
    Objecter *objecter;        //objecter对象指针

    Mutex lock;
    Cond cond;
    SafeTimer timer;           //定时器

    int refcnt;                //引用计算

    Finisher finisher;         //用于执行回调函数的

finisher类

.....


}
```

通过RadosClient的成员函数，可以了解RadosClient的实现功能如下：

1) 网络连接。

Connect函数是RadosClient的初始化函数，完成了许多的初始化工作：

a) 调用函数monclient.build_initial_monmap，从配置文件里检查是否有初始的Monitor的地址信息。

b) 创建网络通信模块messenger，并设置相关的Policy信息。

c) 创建objecter对象并初始化。

d) 调用monclient.init () 函数初始化monclient。

e) Timer定时器初始化，Finisher对象初始化。

2) pool的同步和异步创建。

a) 函数pool_create同步创建pool。其实现过程为调用Objecter::create_pool函数，构造PoolOp操作，通过Monitor的客户端monc发送请求给Monitor创建一个pool，并同步等待请求的返回。

b) 函数pool_create_async异步创建。与同步方式的区别在于注册了回调函数，当创建成功后，执行回调函数通知完成。

3) pool的同步和异步删除。

函数`delete_pool`完成删除，函数`delete_pool_async`异步删除。其过程和pool的创建过程相同，向Monitor发送删除的请求。

4) 查找pool和列举pool。

函数`lookup_pool`用于查找pool，函数`pool_list`用于列出所有的pool。pool的相关的信息都保存在OsdMap信息当中。

5) 获取pool和系统的信息。

函数`get_pool_stats`用于获取pool的统计信息，函数`get_fs_stats`用于获取系统的统计信息。

6) 命令处理。

函数`mon_command`处理Monitor相关的命令，它调用函数`monclient.start_mon_command`把命令发送给Monitor处理。函数`osd_command`处理OSD相关的命令，它调用函数`objecter->osd_command`把命令发送给对应OSD处理。函数`pg_command`处理PG相关的命令，它调用函数`objecter->pg_command`把命令发送给该PG的主OSD来处理。

7) 创建IoCtxImpl对象。

函数`create_ioctx`创建一个pool相关的上下文信息`IoCtxImpl`对象。

5.1.2 IoCtxImpl

类IoCtxImpl是pool相关的上下文信息，一个pool对应一个IoCtxImpl对象，可以在该pool里创建，删除对象，完成对象数据读写等各种操作，包括同步和异步的实现。其处理过程都比较简单，而且过程类似：

- 1) 把请求封装成ObjectOperation类（该类定义在osdc/Objecter.h中）。
- 2) 然后再添加pool的地址信息，封装成Objecter::Op对象。
- 3) 调用函数objecter->op_submit发送给相应的OSD，如果是同步操作，就等待操作完成。如果是异步操作，就不用等待，直接返回。当操作完成后，调用相应的回调函数通知。

5.2 OSDC

OSDC是客户端比较底层的模块，其核心在于封装操作数据，计算对象的地址，发送请求和处理超时。

5.2.1 ObjectOperation

类ObjectOperation用于操作相关的参数统一封装在该类里，该类可以一次封装多个对象的操作：

```
struct ObjectOperation {
    vector<OSDOp> ops;           //多个操作

    int flags;                    //操作的标志

    int priority;                //优先级

    vector<bufferlist*> out_bl;   //每个操作对应的输出缓存区队列

    vector<Context*> out_handler; //每个操作对应的回调函数队列

    vector<int*> out_rval;        //每个操作对应的操作结果队列

}
```

类OSDOP封装对象的一个操作。结构体ceph_osd_op封装一个操作的操作码和相关的输入和输出参数：

```
struct OSDOp {
    ceph_osd_op op;             //各种操作码和操作参数

    sobject_t soid;             //操作对象

    bufferlist indata, outdata;  //输入和输出

    bufferlist
    int32_t rval;               //操作结果

}
```


5.2.2 op_target

结构op_target封装了对象所在的PG，以及PG对应的OSD列表等地址信息：

```
struct op_target_t {
    int flags;                                //标志

    object_t base_oid;                         //读取的对象

    object_locator_t base_oloc;                //对象的
                                                pool信息

    object_t target_oid;                       //最终读取的目标对象

    object_locator_t target_oloc;              //最终目标对象的
                                                pool信息
```

在这里由于

Cache tier的存在，导致产生最终读取的目标和
pool的不同。

}

5.2.3 Op

结构Op封装了完成一个操作的相关上下文信息，包括target地址信息、链接信息等：

```
struct Op : public RefCountedObject {
    OSDSession *session;           //OSD相关的Session信息

    int incarnation;              //引用次数

    op_target_t target;           //地址信息

    vector<OSDOp> ops;          //对应多个操作的封装

    snapid_t snapid;              //快照的id
    SnapContext snapc;            // pool层级的快照信息

    bufferlist *outbl;             //输出的bufferlist
    vector<bufferlist*> out_bl;   //每个操作对应的bufferlist

    bufferlist
    vector<Context*> out_handler; //每个操作对应的回调函数

    vector<int*> out_rval;        //每个操作对应的输出结果

}
```

5.2.4 Striper

对象有分片（stripe）时，类Striper用于完成对象分片数据的计算。数据结构ceph_file_layout用来保存文件或者image的分片信息：

```
struct ceph_file_layout {           //文件
    -> 对象的映射
    *
    uint32_t fl_stripe_unit;        //stripe的单位，必须是
    page_size的倍数

    uint32_t fl_stripe_count;       //stripe跨越的对象数

    uint32_t fl_object_size;        //对象的大小

    uint32_t fl_cas_hash;           //哈希值，当为
    0没有设置
    ;当为
    1=sha256的哈希值

    /* pg -> disk layout的映射
     */
    uint32_t fl_object_stripe_unit; //没有用到

    /*对象
     -> pg layout映射
     */
    uint32_t fl_pg_preferred;       //PG优先选择的主
    OSD
    uint32_t fl_pg_pool;           //pool的
    id
} __attribute__((packed));
```

对象ObjectExtent用来记录对象内的分片信息：

```
class ObjectExtent {
public:
    object_t      oid;           //对象的
    id
    uint64_t      objectno;      //分片序号
    uint64_t      offset;        //对象内的偏移
    uint64_t      length;        //长度
    uint64_t      truncate_size; //对象
    truncate的操作的
    size
    object_locator_t oloc;       //对象位置信息，例如在哪个
pool中，等等

    vector<pair<uint64_t,uint64_t> > buffer_extents; //Extents在
buffer中的偏移和长度，有可能多个
extents
}
void Striper::file_to_extents(
    CephContext *cct, const char *object_format,
    const ceph_file_layout *layout, //分片信息

    uint64_t offset, uint64_t len, //文件的偏移，长度

    uint64_t trunc_size,
    map<object_t,vector<ObjectExtent> >& object_extents, //分布到每个对象的数据段

    uint64_t buffer_offset) //在
buffer中的偏移量

}
```

函数file_to_extents完成了file到对象stripe后的映射。只有了解清楚了每

个概念，计算方法都比较简单。下面举例说明。

例5-1 file_to_extents示例

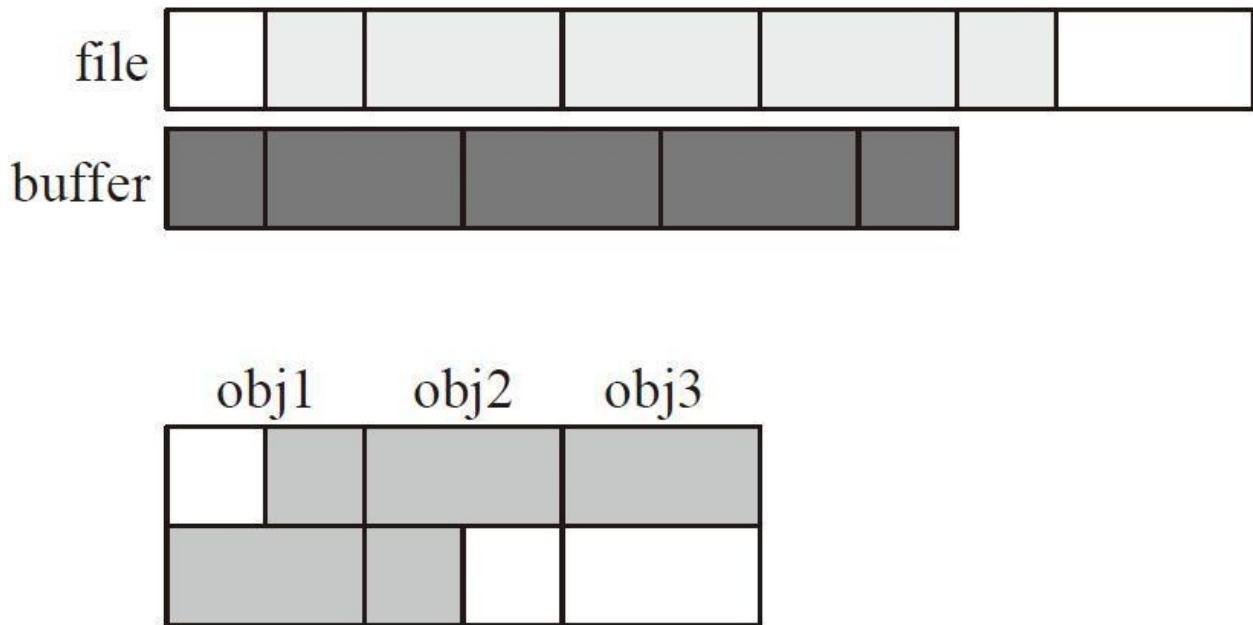


图5-2 file_to_extents示例

如图5-2所示，要计算的文件的offset为2KB，length为16KB。文件的分片信息，stripe为4KB，stripe_count为3，object_size为8KB。对象obj1对应的ObjectExtent为：

```
object_extents[obj1] =
{   oid  = "obj1",
    objectno = 0,
    offset = 2k,
    length = 6k,
    buffer_extents = { [0,2k], [6k,4k] }
}
```

其中， oid就是映射对象的id， objectno为stirpe对象的序号， offset为映射的数据段在对象内的起始偏移， length为对象内的长度。 buffer_extents为映射的数据在buffer内的偏移和长度。

5.2.5 ObjectCacher

类ObjectCacher提供了客户端的基于LRU算法对象数据缓存功能，其实比较简单，这里就不深入分析了。

5.3 客户写操作分析

以下代码是通过Librados库的接口写入数据到对象中的典型例程，对象的其他操作过程都类似：

```
rados_t cluster;
rados_ioctx_t ioctx;
rados_create(&cluster, NULL);
rados_conf_read_file(cluster, NULL);
rados_connect(cluster);
rados_ioctx_create(cluster, pool_name.c_str(), &ioctx);
rados_write(ioctx, "foo", buf, sizeof(buf), 0)
```

上述代码是C语言接口完成的，其流程如下：

- 1) 首先调用rados_create函数创建一个RadosClient对象，输出为类型rados_t，它是一个void类型的指针，通过librados::RadosClient对象的强制转换产生。第二个参数id为一个标识符，一般传入为NULL。
- 2) 调用函数rados_conf_read来读取配置文件。第二个参数为配置文件的路径，如果是NULL，就搜索默认的配置文件。
- 3) 调用rados_connect函数，它调用了RadosClient的connect函数，做相关的初始化工作。
- 4) 调用函数rados_ioctx_create，它调用RadosClient的create_ioctx函数，创建pool相关的IoCtxImpl类，其输出为类型rados_ioctx_t，它也是void类型的指针，由IoCtxImpl对象转换而来。

5) 调用函数rados_write函数，向该pool的名为“foo”的对象写入数据。
其调用IoCtxImpl类的write操作。

5.3.1 写操作消息封装

本函数完成具体的写操作：代码如下：

```
librados::IoCtxImpl::write(const object_t& oid, bufferlist& bl,  
                           size_t len, uint64_t off)
```

其实现过程如下：

- 1) 创建ObjectOperation对象，封装写操作的相关参数。
- 2) 调用函数operate完成处理。
 - a) 调用函数objecter->prepare_mutate_op把ObjectOperation类型的封装成Op类型，添加了object_locator_t相关的pool信息。
 - b) 调用objecter->op_submit把消息发送出去。
 - c) 等到操作完成。

5.3.2 发送数据op_submit

函数op_submit用来把封装好的操作Op通过网络发送出去。

函数_op_submit_with_budget用来处理Throttle相关的流量限制。如果osd_timeout大于0，就是设置定时器，当操作超时，就调用定时器回调函数op_cancel取消操作。

函数_op_submit完成了关键的地址寻址和发送工作，其处理过程如下：

- 1) 调用函数_calc_target来计算对象的目标OSD。
- 2) 调用函数_get_session获取目标OSD的链接，如果返回值为-EAGAIN，就升级为写锁，重新获取。
- 3) 检查当前的状态标志，如果当前是CEPH_OSDMAP_PAUSEWR或者OSD空间满，就暂时不发送请求，否则调用函数_prepare_osd_op准备请求的消息，调用函数_send_op发送出去。

5.3.3 对象寻址_calc_target

函数_calc_target用于完成对象到osd的寻址过程：

```
int Objecter::_calc_target(op_target_t *t, epoch_t *last_force_resend, bool any_
```

其处理过程如下：

- 1) 首先根据t->base_oloc.pool的pool信息， 获取pg_pool_t对象。
- 2) 检查如果强制重发， force_resend设置为true。
- 3) 检查cache tier， 如果是读操作， 并且有读缓存， 就设置target_oloc.pool为该pool的read_tier值；如果是写操作， 并且有写缓存， 就设置target_oloc.pool为该pool的write_tier值。
- 4) 调用函数osdmap->object_locator_to_pg获取目标对象所在的PG。
- 5) 调用函数osdmap->pg_to_up_acting_osds， 通过CRUSH算分， 获取该PG对应的OSD列表。
- 6) 如果是写操作， target的OSD就设置为主OSD；如果是读操作， 如果设置了CEPH OSD FLAG BALANCE READS标志， 就随机选择一个副本读取。如果设置了CEPH OSD FLAG LOCALIZE READS标志， 就尽可能选择本地副本读取。

5.4 Cls

Cls是Ceph的一个模块扩展，它允许用户自定义对象的操作接口和实现方法，为用户提供了一种比较方便的接口扩展方式。目前rbd和lock等模块都使用了这种机制。

5.4.1 模块以及方法的注册

类ClassHandler用来管理所有的扩展模块。函数register_class用来注册模块：

```
class ClassHandler{
    CephContext *cct;
    Mutex mutex;
    map<string, ClassData> classes;    //所有注册的模块：模块名-模块元数据信息

    ...
}
```

类ClassData描述了一个模块的相关元数据信息。它描述一个扩展模块的相关信息，包括模块名、模块相关的操作方法以及依赖的模块：

```
struct ClassData {
    enum Status {
        CLASS_UNKNOWN,           //初始未知状态

        CLASS_MISSING,           //缺失状态（动态链接库找不着）

        CLASS_MISSING_DEPS,      //依赖的模块缺失

        CLASS_INITIALIZING,      //正在初始化

        CLASS_OPEN,               //已经初始化（动态链接库以及加载成功）

    } status;                  //当前模块的加载状态

    string name;              //模块的名字

    ClassHandler *handler;    //管理模块的指针
```

```
void *handle;
map<string, ClassMethod> methods_map; //模块下所有注册的方法

map<string, ClassFilter> filters_map; //模块下所有注册过滤方法

set<ClassData *> dependencies; //本模块依赖的模块

set<ClassData *> missing_dependencies; //缺失的依赖模块

}
```

ClassMethod定义一个模块具体的方法名，以及函数类型：

```
struct ClassMethod {
    struct ClassHandler::ClassData *cls; //所属模块的
    ClassData的指针

    string name; //方法名

    int flags; //方法相关的标志

    cls_method_call_t func; //C类型函数指针

    cls_method_cxx_call_t cxx_func; //C++类型函数指针

}
```

在src/objclass/class_api.c里定义了一些辅助函数用来注册模块以及方法：

·注册一个模块如下：

```
int cls_register(const char *name, cls_handle_t *handle);
```

·注册一个模块的方法如下：

```
int cls_register_method(cls_handle_t hclass, const char *method,  
                      int flags, cls_method_call_t class_call, cls_method_  
                      handle_t *handle)
```

5.4.2 模块的方法执行

模块方法的执行在类ReplicatedPG的函数do_osd_ops里实现。执行方法对应的操作码为CEPH OSD OP_CALL值：

```
int ReplicatedPG::do_osd_ops(OpContext *ctx, vector<OSDOp>& ops){  
    ....  
  
    case CEPH OSD OP CALL:  
        //加载相关的模块  
  
        ClassHandler::ClassData *cls;  
        result = osd->class_handler->open_class(cname, &cls);  
        assert(result == 0);    //函数  
  
        init_op_flags()已经对结果做了验证  
  
        //根据方法名获取方法  
  
        ClassHandler::ClassMethod *method = cls->get_method(mname.c_str());  
        //执行方法  
  
        result = method->exec((cls_method_context_t)&ctx, indata, outdata);....  
}
```

5.4.3 举例说明

以Cls下的rbd扩展接口来说明模块的定义、注册和调用过程：

- 1) rbd的定义和注册。在cls_rbd.cc的函数里，注册了rbd模块，以及自定义的方法：

```
cls_register("rbd", &h_class);
cls_register_cxx_method(h_class, "create",
    CLS_METHOD_RD | CLS_METHOD_WR,
    create, &h_create); ...
```

- 2) cls_rbd_client.h和cls_rbd_client.cc里定义了客户端访问rbd函数，其调用ioctx → exec函数，封装成CEPH OSD_OP_CALL类型的ObjectOperation操作，发送给相应的osd处理。

- 3) cls_rbd.cc里定义了相应的函数在服务端的实现，其把输入参数从bufferlist中解析处理，调用相应的实现，把输出结果封装到输出bufferlist中。

```
int create(cls_method_context_t hctx, bufferlist *in,
           bufferlist *out)
{
    string object_prefix;
    uint64_t features, size;
    uint8_t order;
    try {
        bufferlist::iterator iter = in->begin();
        ::decode(size, iter);
        ::decode(order, iter);
        ::decode(features, iter);
        ::decode(object_prefix, iter);
```

```
    } catch (const buffer::error &err) {
        return -EINVAL;
        .....
    }

```

通过以上介绍，可以了解Cls的扩展模块：如何定义、注册一个新的模块，以及调用机制等。

5.5 Librbd

Librbd模块实现了RBD（rados block device）接口，其基于Librados实现了对RBD的基本操作。Librbd的架构如图5-3所示。

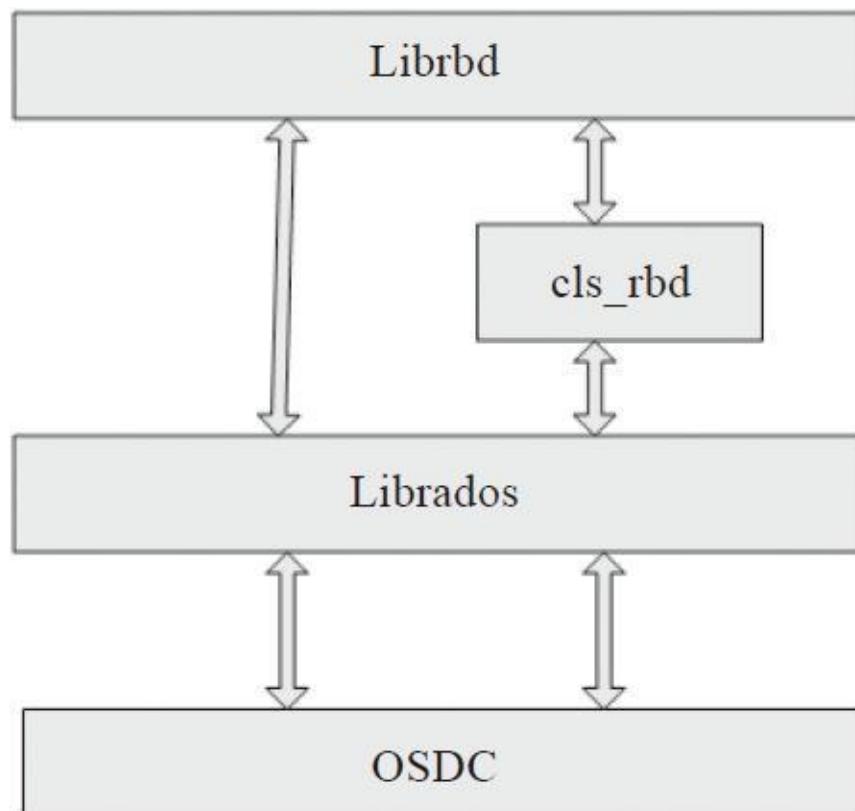


图5-3 Librbd的架构图

在最上层是Librbd层，模块cls_rbd是一个Cls扩展模块，实现了RBD的元数据相关的操作。RBD的数据访问直接通过该Librados来访问。在最底层是OSDC层完成数据的发送。

5.5.1 RBD的相关对象

RBD的相关对象如下所示。

·rbd_directory对象：该对象在每个pool里都存在，用来保存该pool下所有的RBD的目录信息。当创建RBD块设备时，会检查如果rbd_directory对象不存在，就会创建该对象。该对象的omap属性里保存所有的RBD设备的名字和id信息。如表5-1所示。

表5-1 rbd_directory的omap的kv属性

Key 值	Value 值
"name_" + rbd_name1	"id_" + rbd_id1
"name_" + rbd_name2	"id_" + rbd_id2
.....

omap的属性里，保存所有RBD的设备名和id信息，key值为“name”和设备名的拼接，value为“id_”和RBD的id的拼接。

·对于每一个RBD设备，其创建如下对应的对象（最新的版本v2）：

·rbd_id.<name>：称为RBD的id_obj对象，其名字保存了RBD的name信息，其对象的内容保存了该RBD的id信息。

·rbd_header.<id>：称为RBD的head_obj对象，其omap保存了RBD相关的元数据信息。

·rbd_object_map.<id>：保存了其对象和父image对象映射信息。

·数据对象（多个）。

```
rbd_data.<id>.00000000  
rbd_data.<id>.00000001.....
```

5.5.2 RBD元数据操作

结构ImageCtx用来处理一个Image的上下文相关信息。在Internal.cc和Internal.h定义了RBD的元数据相关的操作函数，其调用Cls下的RBD模块来完成RBD设备的创建、删除、快照等元数据操作。

下面通过分析RBD的创建过程来分析RBD的元数据操作过程，创建代码如下：

```
extern "C" int rbd_create4(rados_ioctx_t p, const char *name,
                           uint64_t size, rbd_image_options_t opts)
```

RBD的创建根据传入参数的不同，有4个函数入口，下面主要研究rbd_create4函数，其实现过程如下：

- 1) 调用函数librbd::create设置RBD相应的参数：
 - a) 设置RBD的format格式。
 - b) 设置RBD的feature信息。
 - c) 设置RBD的分片信息stripe_unit和stripe_count参数。
 - d) 设置RBD的order值，其决定了RBD对象size大小。
 - e) 设置bid为rados的instance_id，根据它来生成RBD的id值。

2) 如果是版本v2，就调用函数create_v2来继续创建：

- a) 获取id_obj的名字：rbd_id.<name>，调用函数io_ctx.create创建该对象。
- b) 和bid随机产生一个id，做为新的image的id值。
- c) 调用函数cls_client::set_id设置id，就是在对象id_obj的内容中写入id。
- d) 调用函数cls_client::dir_add_image把新创建的RBD加入rbd_directory目录中，也就是在对象rbd_directory的omap属性中加入该RBD的名字和id的键值对。
- e) 获取对象head_obj的名字，也就是rbd_header.<id>的格式。
- f) 调用函数cls_client::create_image创建image，其在head_obj对象中的omap属性中设置size、order、feature等RBD相关的元数据信息。
- g) 如果对象有stripe，就调用函数cls_client::set_stripe_unit_count设置stripe相关的参数。
- h) 如果有ObjectMap属性，就设置ObjectMap，如果RBD有mirror，就完成rbd_journal相关的设置。

通过RBD的创建过程，可以了解其元数据相关的操作，就是通过Cls的RBD模块设置相关的元数据信息。其他的元数据操作过程都类似。

5.5.3 RBD数据操作

每个ImageCtx中都有一个对象AioImageRequestWQ，它是一个工作队列，基于ThreadPool::PointerWQ来实现了异步请求的发送工作。

如图5-4所示：类AioObjectRequest是单个对象的异步请求。

AioObjectRead和AbstractAioObjectWrite分别为对象的异步读和异步写操作。对象的异步操作truncate、 write、 trim、 zero等操作都继承AbstractAioObjectWrite类。

如图5-5所示：类AioImageRequest是所有Image操作的基类，AioObjectRead实现了image的异步读操作的逻辑。AbstractAioImageWrite为异步写操作的抽象类。AioImageWrite， AioImageDiscard， AioImageFlush继承了AbstractAioImageWrite类，分别完成了异步写，异步丢弃某一段数据（Discard），异步数据Flush等操作。

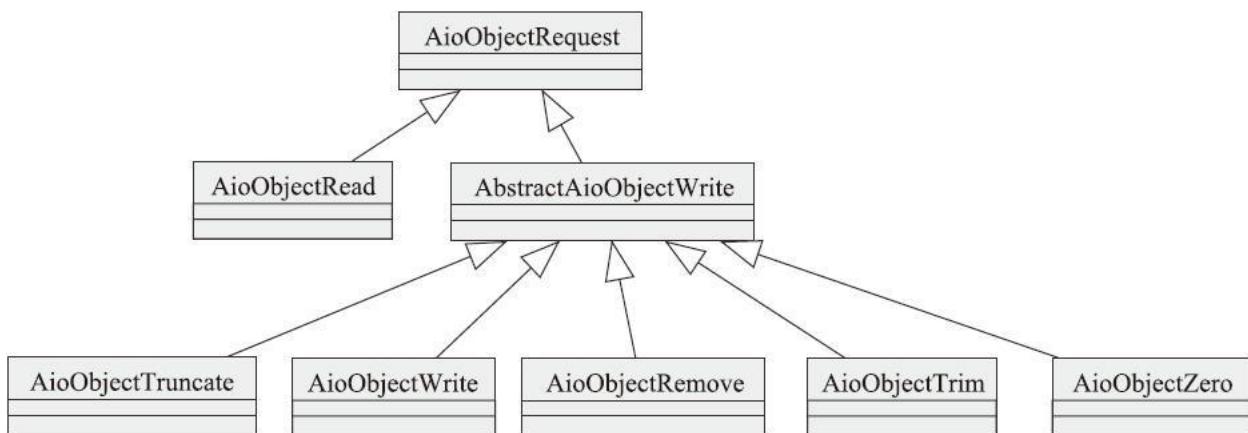


图5-4 对象请求类图

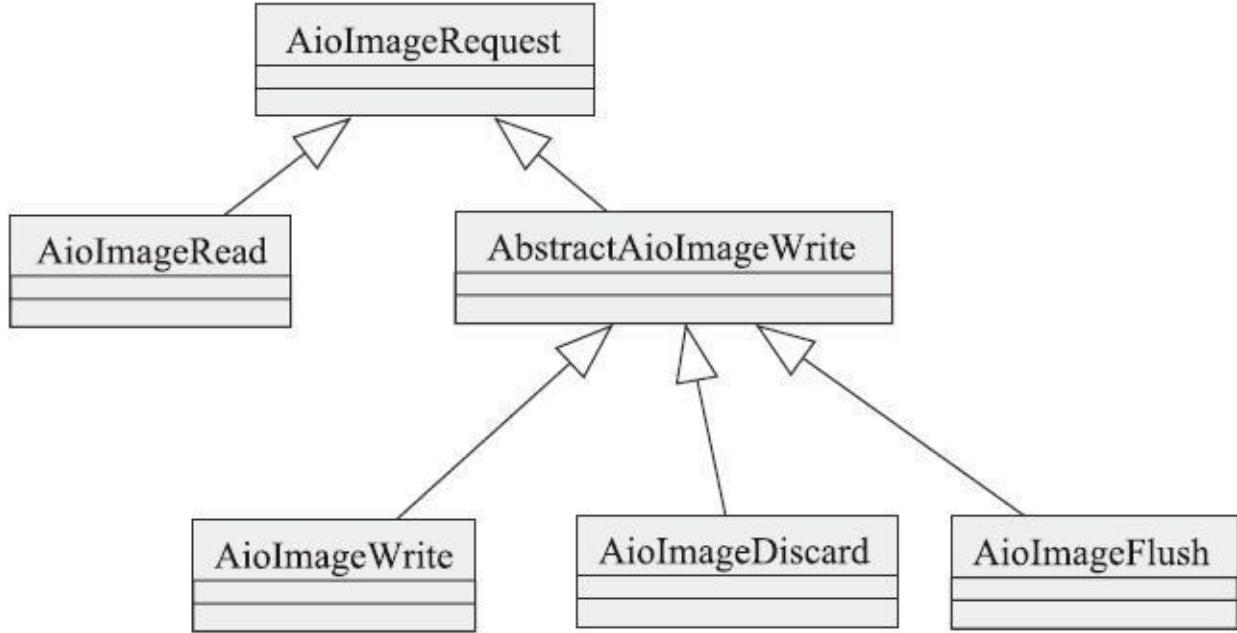


图5-5 Image请求类图

当一个AioImageRequest跨越多个对象时，每个对象就会产生一个AioObjectRequest请求，每个AioObjectRequest请求分别处理。

下面以rbd_aio_write为例，来介绍RBD的数据读写的流程：

```

extern "C" int rbd_aio_write(rbd_image_t image, uint64_t off, size_t len,
                           const char *buf, rbd_completion_t c)
  
```

其参数分别为：

- image：对象。
- off：image内的偏移。
- len：写操作的长度。

·buf：写操作的数据buf。

·异步操作的回调函数。

处理流程如下：

- 1) 调用ictx->aio_work_queue->aio_write函数来处理。
 - a) 如果是非阻塞IO，或者有rbd mirror，或者有阻塞的IO请求，就调用函数AioImageWrite对象，加入到AioImageRequestWQ的工作队列里。
 - b) 否则调用AioImageRequest :: aio_write，构建AioImageWrite对象，并调用req.send () 发送。
- 2) 加入AioImageRequestWQ队列的AioImageWrite请求，在线程池的处理函数AioImageRequestWQ :: process里，同样调用req.send () 函数来处理。
- 3) AioImageWrite类的send函数继承自AioImageRequest的send函数，其直接调用send_request函数，AioImageWrite的send_request函数继承来自AbstractAioImageWrite的send_request函数。
- 4) 在AbstractAioImageWrite的send_request函数里，调用了Striper :: file_to_extents函数，它计算该image的要写入的数据段映射到对象上的数据段。
- 5) 在函数send_object_requests里，针对每一个对象，构建

AioObjectWrite请求。如果有rbd journal，就把请求加入到aio_object_requests里；否则就调用AioObjectWrite的send函数处理。

- 6) 在AbstractAioObjectWrite::send函数里，调用AbstractAioObjectWrite::send_pre函数。该函数处理object_map相关的操作后；如果是写操，就调用AbstractAioObjectWrite::send_write函数。
- 7) 在函数AbstractAioObjectWrite::send_write里，区分了两种操作：如果已经确认该对象不存在，并且有父image，就handle_write_guard函数处理clone相关的copyup操作。否则就直接调用函数AbstractAioObjectWrite::send_write_op处理。
- 8) 在函数AbstractAioObjectWrite::send_write_op里，调用m_ictx->data_ctx.aio_operate函数，通过该rados层的操作函数，完成对象数据的写入。

5.5.4 RBD的快照和克隆

RBD的快照基于rados的snapshot的机制实现。RBD的快照在客户端的实现比较简单，其核心流程就是申请新的snap_seq，把snap_name和snap_id记录在RBD的元数据中。写操作的copy-on-write机制是依赖rados在OSD服务端实现的。

RBD的clone操作在客户端实现了RBD的copy-on-write机制。当对RBD的image发起读操作或者写操作时，如果对象不存在，就需要检查：如果有父image，则读取父image所对应的对象数据。

1.RBD的snapshot的创建

RBD创建snapshot的核心逻辑在SnapshotCreateRequest里处理。其流程如下：

- 1) 调用函数SnapshotCreateRequest<I>::send_suspend_aio来阻塞RBD的写操作。
- 2) 调用函数SnapshotCreateRequest<I>::send_allocate_snap_id向Monitor申请一个新的snap_seq序号。
- 3) 在函数SnapshotCreateRequest<I>::send_create_snap () 里，调用cls_client::snapshot_add把snap_name和snap_id添加到RBD的head_obj的

元数据中。

4) 调用函数ObjectMap::snapshot_add，构建object_map::SnapshotCreateRequest，做ObjectMap相关的处理。

2.RBD的CopyUp操作

当RBD在读写一个对象时，如果该对象不存在，并且有父image，就需要CopyUp操作，读取父image所对应的对象数据。

其对应的实现在函数void AbstractAioObjectWrite::send_write()里，如果对象不存在，并且有parent，就调用函数handle_write_guard处理：

```
if (!m_object_exist && has_parent()) {
    m_state = LIBRBD_AIO_WRITE_GUARD;
    handle_write_guard();
} else {
    send_write_op(true);
}
```

在函数void AbstractAioObjectWrite::handle_write_guard里，如果有parent，就调用函数send_copyup函数：

```
if (has_parent) {
    send_copyup();
} else {
    // parent may have disappeared -- send original write again
    ldout(m_ictx->cct, 20) << "should_complete(" << this
    << "): parent overlap now 0" << endl;
    send_write();
}
```

在函数AbstractAioObjectWrite::send_copyup里，构建CopyupRequest

请求， 来处理读取parent image对应的对象。

3.ObjectMap

在RBD里， 新添加了一个特性， 就是ObjectMap， 它添加了RBD的一个属性， 用Bit vector的形式来记录一个对象是否存在， 这样就可以极大地提供clone卷的读写性能。

5.6 本章小结

本章大致介绍了客户端的各个模块的功能以及核心典型操作的处理流程。由于Librados和Librbd的代码比较庞大，承载了所有功能的接口，故一些功能本章没有介绍到或者介绍得比较简略，但是客户端的代码有很大的类似性，通过了解典型流程，便不难理解其他代码。

第6章 Ceph的数据读写

本章介绍Ceph的服务端OSD（书中简称OSD模块或者OSD）的实现。其对应的源代码在src/osd目录下。OSD模块是Ceph服务进程的核心实现，它实现了服务端的核心功能。本章先介绍OSD模块静态类图相关数据结构，再着重介绍服务端数据的写入和读取流程。

6.1 OSD模块静态类图

OSD模块的静态类图如图6-1所示。

OSD模块的核心类及其之间的静态类图说明如下：

- 类OSD和类OSDService是核心类，处理一个osd节点层面的工作。在早期的版本中，OSD和OSDService是一个类。由于OSD的类承载了太多的功能，后面的版本中引入OSDService类，分担一部原OSD类的功能。
- 类PG处理PG相关状态维护以及实现PG层面的基本功能。其核心功能是用boost库的statechart状态机来实现的PG状态转换。
- 类ReplicatedPG继承了类PG，在其基础上实现了PG内的数据读写以及数据恢复相关的操作。
- 类PGBackend的主要功能是把数据以事务的形式同步到一个PG其他从OSD节点上。
- PGBackend的内部类PGTransaction就是同步的事务接口，其两个类型的实现分别对应RPGTransaction和ECTransaction两个子类。
- PGBackend两个子类ReplicatedBackend和ECBackend分别对应PG的两种类型的实现。
- 类SnapMapper额外保存对象和对象的快照信息，在对象的属性里保存

了相关的快照信息。这里保存的快照信息为冗余信息，用于数据效验。

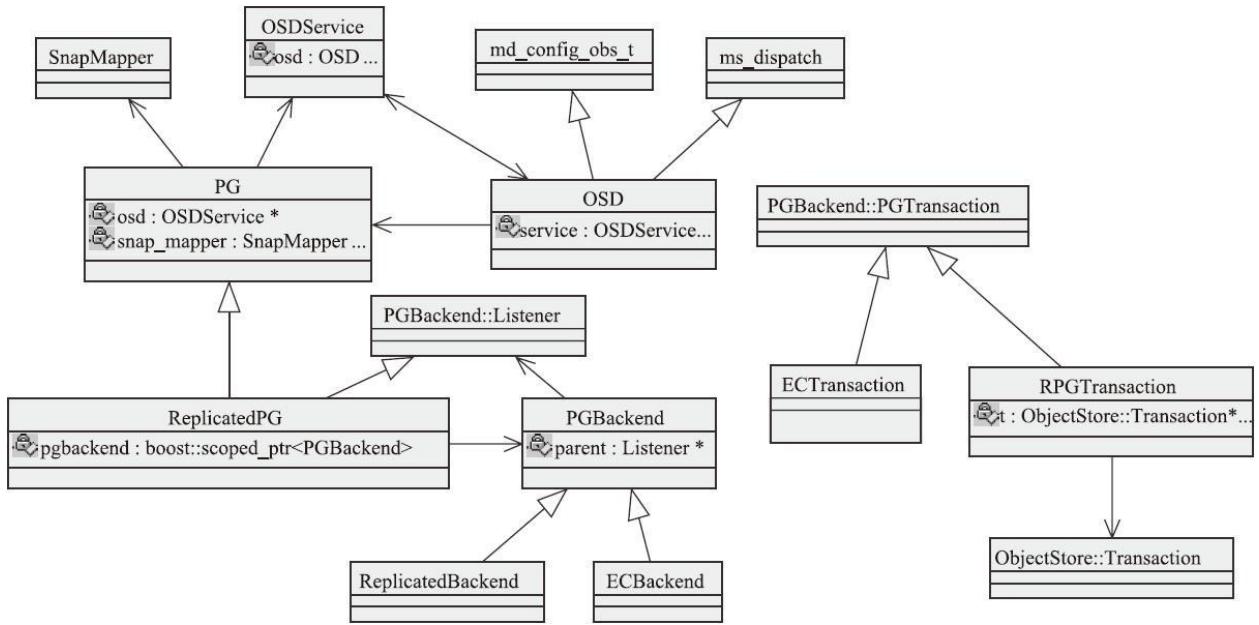


图6-1 OSD模块的静态类图

6.2 相关数据结构

下面将介绍OSD模块相关的一些核心的数据结构。从最高的逻辑层次为pool的概念，然后是PG的概念。其次是OSDMap记录了集群的所有的配置信息。数据结构OSDOp是一个操作上下文的封装。结构object_info_t保存了一个对象的元数据信息和访问信息。对象ObjectState是在object_info_t基础上添加了一些内存的状态信息。SnapSetContext和ObjectContext分别保存了快照和对象的上下文相关的信息。Session保存了一个端到端的链接相关的上下文信息。

6.2.1 Pool

Pool是整个集群层面定义的一个逻辑的存储池。对一个Pool可以设置相应的数据冗余类型，目前有副本和纠删码两种实现。数据结构pg_pool_t用于保存Pool的相关信息。Pool的数据结构如下：

```
struct pg_pool_t {
    enum {
        TYPE_REPLICATED = 1,      //副本
        //TYPE_RAID4 = 2,          //从来没有实现的
        raid4
        TYPE_ERASURE = 3,         //纠删码
    };
    uint64_t flags;             // pool相关的标志

    __u8 type;                 //类型
    __u8 size, min_size;       // pool的
size和
min_size, 也就是副本数和至少保证的副本数

    __u8 crush_ruleset;        // rule set的编号
    __u8 object_hash;          //对象映射的
hash函数

    __u32 pg_num, pgp_num;     // PG的数量,
pgp的值（用于
rule set的设置）

    string erasure_code_profile; //EC的配置信息
```

.....
}

·type：定义了Pool的类型，目前有replication和ErasureCode两种类型。

·size和min_size定义了Pool的冗余模式：

·如果是replication模式，size定义了副本数目，min_size为副本的最小数目。例如：如果size设置为3，副本数为3，min_size设置为1就只允许两副本损坏。

·如果是Erasure Code (M+N)，size是总的分片数M+N；min_size是实际数据的分片数M。

·crush_ruleset：Pool对应的crush规则号。

·erasure_code_profile：EC的配置方式。

·object_hash：通过对对象名映射到PG的hash函数。

·pg_num：Pool里PG的数量。

通过上面的介绍可以了解到，Pool根据类型不同，定义了两种模式，分别保存了两种模式相关的参数。此外，在结构pg_pool_t里还定义了Pool级别的快照相关的数据结构、Cache Tier相关的数据结构，以及其他一些统计信息。在介绍快照（参见第9章）和Cache Tier（参见第13章）时再详细介绍相关的字段。

6.2.2 PG

PG可以认为是一组对象的集合，该集合里的对象有共同特征：副本都分布在相同的OSD列表中。PG的数据结构如下：

```
struct pg_t {
    uint64_t m_pool;           //pg所在的pool
    uint32_t m_seed;          //pg的序号
    int32_t m_preferred;      //pg优先选择的主osd
};
```

结构体pg_t只是一个PG的静态描述信息。类PG及其子类ReplicatedPG都是和PG相关的处理。

```
struct spg_t {
    pg_t pgid;
    shard_id_t shard;
};
```

数据结构spg_t在pg_t的基础上，加了一个shard_id字段，代表了该PG所在的OSD在对应的OSD列表中的序号。

在Erasure Code模式下，该字段保存了每个分片的序号。该序号在EC的数据encode和decode过程中很关键；对于副本模式，该字段没有意义，都设置为shard_id_t := NO_SHARD值。

PG的分裂 当一个pool里的PG数量不够时，系统允许通过命令增加PG的数量，就会产生PG的分裂，使得一个PG分裂为2的幂次方个PG。PG的分裂后，新的PG和其父PG的OSD列表是一致的，其数据的移动也是本地数据的启动，开销比较小。

6.2.3 OSDMap

类OSDMap定义了Ceph整个集群的全局信息。它由Monitor实现管理，并以全量或者增量的方式向整个集群扩散。每一个epoch对应的OSDMap都需要持久化保存在meta下对应对象的omap属性中。

下面介绍OSDMap核心成员，内部类Incremental以增量的形式保存了OSDMap新增的信息，其内部成员和OSDMap类似，这里就不介绍了。

```
class OSDMap {
    //系统相关信息

    uuid_d fsid;                      //当前集群的
    fsid值

    epoch_t epoch;                     //当前集群的
    epoch值

    utime_t created, modified;         //创建修改的时间戳

    int32_t pool_max;                 //最大的
    pool数量

    uint32_t flags;                   //一些标志信息

    //OSD相关信息

    int num_osd;                      //OSD的总数量

    int num_up_osd;                   //处于
    up状态的
```

OSD的数量

```
int num_in_osd; //处于  
in状态的  
OSD的数量  
  
int32_t max_osd; //OSD的最大数目  
  
vector<uint8_t> osd_state; //OSD的状态  
  
ceph::shared_ptr<addrs_s> osd_addrs; //OSD的地址  
  
vector<__u32> osd_weight; //OSD的权重  
  
vector<osd_info_t> osd_info; //OSD的基本信息  
  
ceph::shared_ptr< vector<uuid_d> > osd_uuid; //OSD对应的  
uuid  
vector<osd_xinfo_t> osd_xinfo; //OSD的一些扩展信息  
  
//PG相关的信息  
  
ceph::shared_ptr< map<pg_t, vector<int32_t> > > pg_temp;  
// temp pg mapping (e.g. while we rebuild)  
ceph::shared_ptr< map<pg_t, int32_t > > primary_temp;  
// temp primary mapping (e.g. while we rebuild)  
ceph::shared_ptr< vector<__u32> > osd_primary_affinity;  
//< 16.16 fixed point, 0x10000 = baseline  
//pool相关的信息  
  
map<int64_t, pg_pool_t> pools; //pool的  
id到类  
pg_pool_t的映射  
  
map<int64_t, string> pool_name; //pool的  
id到  
pool的名字的映射  
  
map<string, map<string, string> > erasure_code_profiles; //pool的  
EC相关的信息
```

```
map<string,int64_t> name_pool;           //pool的名字到
pool的
id的映射

//Crush相关的信息

ceph::shared_ptr<CrushWrapper> crush;    //CRUSH算法

.....
}
```

通过OSDMap数据成员的了解，可以看到，OSDMap包含了四类信息：首先是集群的信息，其次是pool相关的信息，然后是临时PG相关的信息，最后就是所有OSD的状态信息。

6.2.4 OSDOp

类MOSDOp封装了一些基本操作相关的数据。

```
class MOSDOp : public Message {
    object_t oid;           //操作的对象

    object_locator_t oloc;  //对象的位置信息

    pg_t pgid;              //对象所在的
                            PG的

    id
    vector<OSDOp> ops;     //针对

    oid的多个操作集合

    private:
    //快照相关

    snapid_t snapid;//snapid,如果是
                     CEPH_NOSNAP,就是

    head对象；否则就是等于

    snap_seq
    snapid_t snap_seq;
    //如果是

    head对象

    ,就是最新的快照序号

    //如果是

    snap对象,就是

    snap对应的

    seq
    vector<snapid_t> snaps; //所有的

    snap列表
```

```
    uint64_t features;           //一些  
    feature的标志  
  
    osd_reqid_t reqid;          //请求的唯一  
    id标识
```

MOSDOp在其成员ops向量里分装了多个类型为OSDOp操作数据。MOSDOp封装的操作都是关于对象oid相关的操作，一个MOSDOp只封装针对同一个对象oid的操作。但是对于rados_clone_range这样的操作，需要有一个目标对象oid，还有一个源对象oid，那么源对象的oid就保存在结构OSDOp里。

数据结构OSDOp封装了一个OSD操作需要的数据和元数据：

```
struct OSDOp {  
    ceph_osd_op op;           //具体操作数据的封装  
  
    sobject_t soid;          //src oid，并不是  
    op操作的对象，而是源操作对象  
  
    //例如  
    rados_clone_range 需要目标  
    obj 和源  
    obj  
    bufferlist indata, outdata; //操作的输入输出的  
    data  
    int32_t rval;             //操作返回值  
  
    OSDOp() : rval(0) {  
        memset(&op, 0, sizeof(ceph_osd_op));  
    }  
}
```


6.2.5 Object_info_t

结构object_info_t保存了一个对象的元数据信息和访问信息。其做为对象的一个属性，持久化保存在对象xattr中，对应的key为 OI_ATTR (“_”)， value就是object_info_t的encode后的数据。

```
struct object_info_t {
    hobject_t soid;                                // 对应的对象

    eversion_t version, prior_version;   // 对象的当前版本, 前一个版本

    version_t user_version;                      // 用户操作的版本

    osd_reqid_t last_reqid;                     // 最后请求的请求

    id
    uint64_t size;                               // 对象的大小

    utime_t mtime;                             // 修改时间

    utime_t local_mtime;                       // 修改的本地时间

    typedef enum {
        FLAG_LOST      = 1<<0,
        FLAG_WHITEOUT  = 1<<1,                  // object logically does not exist
        FLAG_DIRTY     = 1<<2,
        // object has been modified since last flushed or undirtied
        FLAG OMAP       = 1 << 3,                // has (or may have) some/any omap data
        FLAG_DATA_DIGEST = 1 << 4,               // has data crc
        FLAG_OMAP_DIGEST = 1 << 5,              // has omap crc
        FLAG_CACHE_PIN = 1 << 6,                // pin the object in cache tier
        //
        FLAGUSES_TMAP = 1<<8,                 // deprecated; no longer used.
    } flag_t;
    flag_t flags;                                // 对象的一些标记

    .....
    vector<snapid_t> snaps;                     // clone对象的快照信息

    uint64_t truncate_seq, truncate_size;
```

```
//truncate操作的序号和  
size  
map<pair<uint64_t, entity_name_t>, watch_info_t> watchers;  
//watchers记录了客户端监控信息，一旦对象的状态发送变化，需要通知客户端  
  
__u32 data_digest; //< data crc32c  
__u32 omap_digest; //< omap crc32c  
//数据或者  
  
omap信息的  
crc32校验信息，可能有，也可能没有  
  
}
```

6.2.6 ObjectState

对象ObjectState是在object_info_t基础上添加了一个字段exists，用来标记对象是否存在。

```
struct ObjectState {
    object_info_t oi;
    bool exists;
    //the stored object exists (i.e., we will remember the object_info_t)
    ObjectState() : exists(false) {}
    ObjectState(const object_info_t &oi_, bool exists_)
        : oi(oi_), exists(exists_) {}
};
```

为什么要加一个额外的bool变量来标记呢？因为object_info_t可能是从缓存的attrs[OI_ATTR]中获取的，并不能确定对象是否存在。

6.2.7 SnapSetContext

SnapSetContext保存了快照的相关信息，即SnapSet的上下文信息。关于SnapSet的内容，可以参考快照相关的介绍：

```
struct SnapSetContext {
    hobject_t oid;      //对象

    int ref;           //本结构的引用计数

    bool registered;   //是否在
    SnapSet Cache中记录

    SnapSet snapset;  //SnapSet 对象快照相关的记录

    bool exists;       //snapset是否存在

    SnapSetContext(const hobject_t& o) :
        oid(o), ref(0), registered(false), exists(true) { }
};
```

6.2.8 ObjectContext

ObjectContext可以说是对象在内存中的一个管理类，保存了一个对象的上下文信息。

```
struct ObjectContext {
    ObjectState obs;           //主要是
    object_info_t,
    描述了对象的状态信息

    SnapSetContext *ssc;        //快照上下文信息
    , 如果没有快照就为空

    Context *destructor_callback; //析构函数的

    private:
        Mutex lock;
    public:
        Cond cond;
        int unstable_writes, readers, writers_waiting, readers_waiting;
        //正在写操作的数目，正在读操作的数目

        //等待写操作的数目，等待读操作的数目

        //如果该对象的写操作被阻塞去恢复另一个对象，设置这个属性

    ObjectContextRef blocked_by; //本对象被某个对象阻塞

    set<ObjectContextRef> blocking; //本对象阻塞的对象集合

    //任何在
    obs.oi.watchers中的
    watchers在
    watchers队列中或者在
    unconnected_watchers中
```

```

map<pair<uint64_t, entity_name_t>, WatchRef> watchers;
//属性的缓存

map<string, bufferlist> attr_cache;
list<OpRequestRef> waiters; //等待状态变化的

waiters
int count; //读或写的数目

struct RWState {
    enum State {
        RWNONE,
        RWREAD,
        RWWRITE,
        RWEXCL,
    };
    State state:4; //读写的状态

    //如果设置，获得锁后，重新执行

backfill操作

    bool recovery_read_marker:1;
    //如果设置，获得锁后重新加入

snaptrim队列中

    bool snaptrimmer_write_marker:1;
}
.....
}

```

下面两个字段比较难理解，进行一些补充说明：

·blocked_by记录了当前对象被其他对象阻塞，blocking记录了本对象阻塞其他对象的情况。当一个对象的写操作依赖其他对象时，就会出现这些情况。这一般对应一个操作涉及多个对象，比如copy操作。把对象obj1上的部分数据拷贝到对象obj2，如果源对象obj1处于missing状态，需要恢复，那么obj2对象就block了obj1对象。

· 内部类RWState通过定义了4种状态，实现了对对象的读写加锁。

6.2.9 Session

类Session是和Connection相关的一个类，用于保存Connection相关的上下文相关的信息。

```
struct Session : public RefCountedObject {
    EntityName entity_name;           //peer实例的名字

    OSDCap caps;
    int64_t auid;
    ConnectionRef con;               //相关的

    Connection
    WatchConState wstate;
    Mutex session_dispatch_lock;
    list<OpRequestRef> waiting_on_map;
    所有的

    OpRequest请求都先添加在这个队列里

    OSDMapRef osdmap;                // Map as of which waiting_for_pg is current
    map<spg_t, list<OpRequestRef>> waiting_for_pg;
    //当前需要更新

    Osdmap的

    pg和对应的请求

    Spinlock sent_epoch_lock;        //通过消息向外通知的

    epoch
    epoch_t last_sent_epoch;         //发送对端的

    epoch
    Spinlock received_map_lock;
    epoch_t received_map_epoch;      //最新的

    MOSDMap消息接收到的

    received_map_epoch
    Session(CephContext *cct) :
        RefCountedObject(cct),
        auid(-1), con(0),
        session_dispatch_lock("Session::session_dispatch_lock"),
        last_sent_epoch(0), received_map_epoch(0)
    {}
};
```

函数update_waiting_for_pg用于检查是否有最新的osdmap：

- 1) 如果该PG有分裂的PG，就把分裂出的新的PG以及对应的OpRequest加入到session的waiting_for_pg队列里。
- 2) 如果该PG不分裂，就并把PG和opRequest加入到waiting_for_pg队列里。

6.3 读写操作的序列图

写操作序列图如图6-2所示。

写操作分为三个阶段：

·**阶段一** 从函数ms_fast_dispatch到函数op_wq.queue函数为止，其处理过程都在网络模块的回调函中处理，主要检查当前OSD的状态，以及epoch是否一致。

·**阶段二** 这个阶段在工作队列op_wq中的线程池里处理，在类ReplicatedPG里，其完成对PG的状态、对象的状态的检查，并把请求封装成事务。

·**阶段三** 本阶段也是在工作队列op_wq中的线程池里处理，主要功能都在类ReplicatedBackend中实现。核心工作就是把封装好的事务通过网络分发到从副本上，最后调用本地FileStore的函数完成本地对象的数据写入。

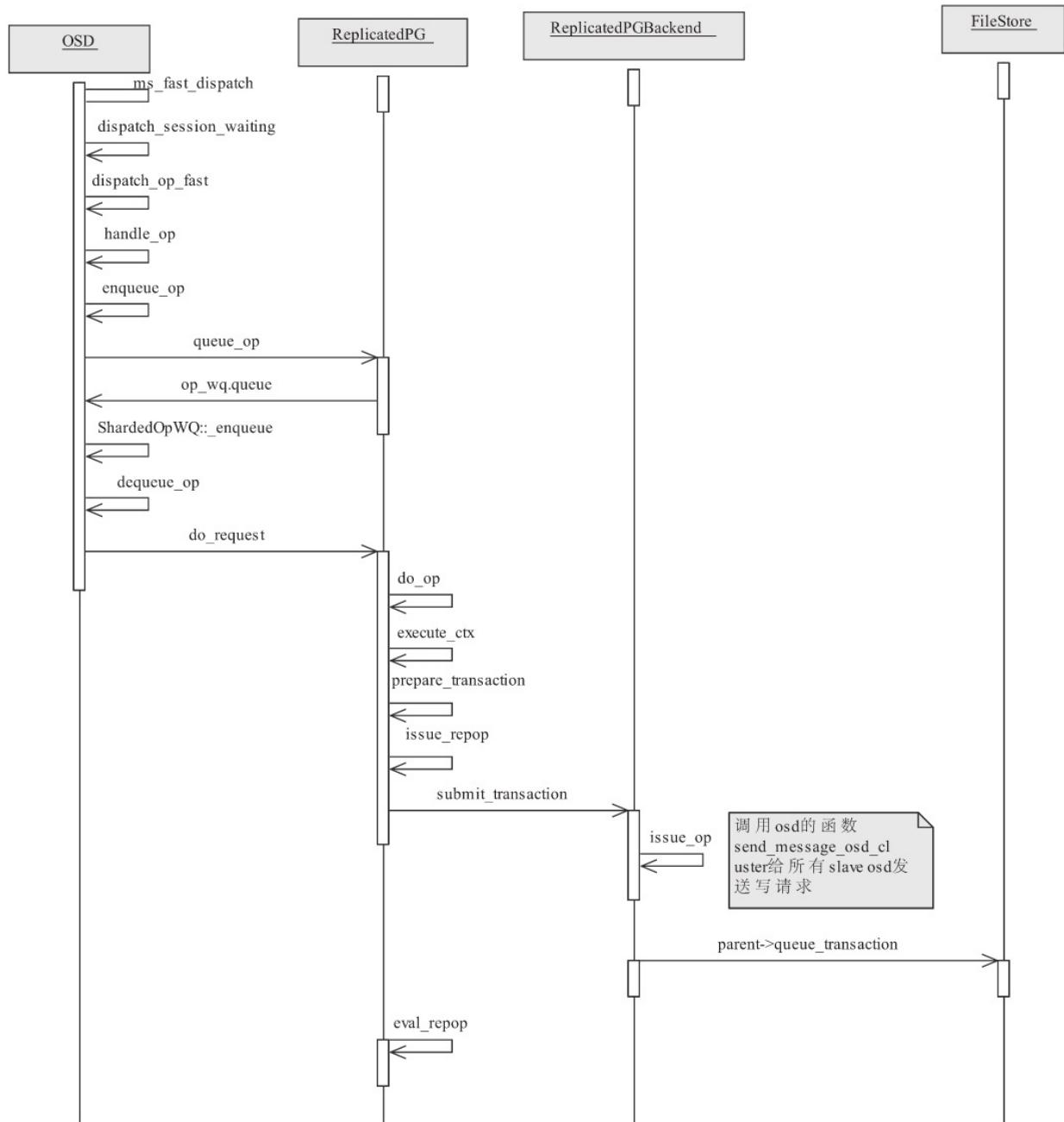


图6-2 OSD处理写操作的序列图

6.4 读写流程代码分析

在介绍了上述的数据结构和基本的流程之后，下面将从服务端接收到消息开始，分三个阶段具体分析读写的过程。

6.4.1 阶段1：接收请求

读写请求都是从OSD::ms_fast_dispatch开始，它是接收读写消息message的入口。下面从这里开始读写操作的分析。本阶段所有的函数是被网络模块的接收线程调用，所以理论上应该尽可能的简单，处理完成后交给后面的OSD模块的OpWQ工作队列来处理。

1. ms_fast_dispatch

```
void OSD::ms_fast_dispatch(Message *m)
```

函数ms_fast_dispatch为OSD注册了网络模块的回调函数，其被网络的接收线程调用，具体实现过程如下：

- 1) 首先检查service，如果已经停止了，就直接返回。
- 2) 调用函数op_tracker.create_request把Message消息转换为OpRequest类型，数据结构OpRequest包装了Message，并添加了一些其他信息。
- 3) 获取nextmap（也就是最新的osdmap）和session，类Session保存了一个Connection的相关信息。
- 4) 调用函数update_waiting_for_pg来更新session里保存的OSDMap信息。

- 5) 把请求加入waiting_on_map的列表里。
- 6) 调用函数dispatch_session_waiting处理，它循环调用函数dispatch_op_fast处理请求。
- 7) 如果session->waiting_on_map不为空，说明该session里还有等待osdmap的请求，把该session加入到session_waiting_for_map队列里。

2.dispatch_op_fast

```
bool OSD::dispatch_op_fast(OpRequestRef& op, OSDMapRef& osdmap)
```

该函数检查OSD目前的epoch是否最新：

- 1) 检查变量is_stopping如果true，就直接返回true值。
- 2) 调用函数op_required_epoch (op)，从OpRequest中获取msg带的epoch，进行比较，如果该值大于OSD最新的epoch，则调用函数osdmap_subscribe更新epoch，返回false值。
- 3) 否则，根据消息类型，调用相应的消息处理函数，本章只关注处理函数handle_op相关的流程。

3.handle_op

```
void OSD::handle_op(OpRequestRef& op, OSDMapRef& osdmap)
```

该函数处理OSD相关的操作，其处理流程如下：

- 1) 首先调用op_is_discardable，检查该OP是否可以丢弃。
- 2) 构建share_map结构体，获取client_session，从client_session获取last_sent_epoch，调用函数service.should_share_map来设置share_map.should_send标志，该函数用于检查是否需要通知对方更新epoch值。这里和dispatch_op_fast的处理区别是，上次是更新自己，这里是通知对方更新。需要注意是，client和OSD的epoch不一致，并不影响读写，只要epoch的变化不影响本次读写PG的OSD list变化。
- 3) 从消息里获取_pgid，从_pg_id里获取pool。
- 4) 调用函数osdmap->raw_pg_to_pg，最终调用pg_pool_t::raw_pg_to_pg函数，对PG做了调整。
- 5) 调用osdmap->get_primary_shard函数，获取该PG的主OSD。
- 6) 调用函数get_pg_or_queue_for_pg，通过pgid获取PG类指针。如果获取成功，就调用函数enqueue_op处理请求。
- 7) 如果PG类的指针没有获取成功，做一些错误检查：
 - a) seng_map为空，client需要重试。
 - b) 客户端的osdmap里没有当前的pool。

- c) 当前OSD的osdmap没有该pool， 或者当前OSD不是该PG的主OSD。

总结，这个函数主要检查了消息的源端epoch是否需要share，最主要的是获取读写请求相关的PG类后，下面就进入PG类的处理。

4.queue_op

```
void PG::queue_op(OpRequestRef& op)
```

该函数的实现如下：

- 1) 加map_lock锁，该锁保护waiting_for_map列表，判断waiting_for_map列表不为空，就把当前OP加入该列表，直接返回。
waiting_for_map列表不为空，说明有操作在等待osdmap的更新，说明当前osdmap不信任，不能继续当前的处理。
- 2) 函数op_must_wait_for_map判断当前的epch是否大于OP的epoch，如果是，则必须加入waiting_for_map等待，等待更新PG当前的epoch值。



这里的osdmap的epoch的判断，是一个PG层的epoch的判断。和前面的判断不在一个层次，这里是需要等待的。

- 3) 最终，把请求加入OSD的op_wq处理队列里。

总结，这个函数做在PG类里，做PG层面的相关检查，如果ok，就加入OSD的op_wq工作队列里继续处理。

6.4.2 阶段2：OSD的op_wq处理

op_wq是一个ShardedWQ类型的工作队列。以下操作都是在op_wq对应的线程池里调用做相应的处理。这里着重分析读写流程。

1.dequeue_op

```
void OSD::dequeue_op(PGRef pg, OpRequestRef op, ThreadPool::TPHandle &handle)
```

- 1) 检查如果op->send_map_update为true，也就是如果需要更新osdmap，就调用函数service.share_map更新源端的osdmap信息。在函数OSD::handle_op里，只在op->send_map_update里设置了是否需要share_map的标记，在这里才真正去发消息实现share操作。
- 2) 检查如果pg正在删除，就把本请求丢弃，直接返回。
- 3) 调用函数pg->do_request (op, handle) 处理请求。

总之，本函数主要实现了使请求源端更新osdmap的操作，接下来在PG里调do_request来处理。

2.do_request

本函数进入ReplicatedPG类来处理：

```
void ReplicatedPG::do_request( OpRequestRef& op, ThreadPool::TPHandle &handle)
```

处理过程如下：

- 1) 调用函数can_discard_request检查op是否可以直接丢弃掉。
- 2) 检查变量flushes_in_progress如果还有flush操作，把op加入waiting_for_peered队列里，直接返回。
- 3) 如果PG还没有peered，调用函数can_handle_while_inactive检查pgbackend能否处理该请求，如果可以，就调用pgbackend->handle_message处理；否则加入waiting_for_peered队列，等待PG完成peering后再处理。



注意

PG处于inactive状态，pgbackend只能处理MSG OSD PG PULL类型的消息。这种情况可能是：本OSD可能已经不在该PG的acting osd列表中，但是可能在上一阶段该PG的OSD列表中，所以PG可能含有有效的对象数据，这些对象数据可以被该PG当前的主OSD拉取以修复当前PG的数据。

- 4) 此时PG处于peered并且flushes_in_progress为0的状态下，检查pgbackend能否处理该请求。pgbackend可以处理数据恢复过程中的PULL和PUSH请求，以及主副本发的从副本的更新相关SUBOP类型的请求。
- 5) 如果是CEPH_MSG OSD_OP，检查该PG的状态，如果处于非active或者replay状态，则把请求添加到waiting_for_active等待队列。

- 6) 检查如果该pool是cache pool，而该操作没有带CEPH_FEATURE OSD CACHE-POOL的feature标志，返回EOPNOTSUPP错误码。
- 7) 根据消息的类型，调用相应的处理函数来处理。

本函数开始进入ReplicatedPG层面来处理，主要检查当前PG的状态是否正常，是否可以处理请求。

3.do_op

函数do_op数比较复杂，处理读写请求的状态检查和一些上下文的准备工作。其中大量的关于快照的处理，本章在遇到快照处理时只简单介绍一下。

```
void ReplicatedPG::do_op(OpRequestRef& op)
```

具体处理过程如下：

- 1) 调用函数m->finish_decode，把消息带的数据从bufferlist中解析出相关的字段。
- 2) 调用osd->osd->init_op_flags初始化op->rmw_flags，函数init_op_flags根据flag来设置rmw_flags标志。
- 3) 如果是读操作：

- a) 如果消息里带有CEPH_OSD_FLAG_BALANCE_READS（平衡读）或者CEPH_OSD_FLAG_LOCALIZE_READS（本地读）标志，表明主从副本都允许读。检查本OSD必须是该PG的primary或者replica之一。
 - b) 如果没有上述标志，读操作只能读取主副本，本OSD必须是该PG的主OSD。
- 4) 如果里面含有includes_pg_op操作，调用pg_op_must_wait检查该操作是否需要等待，如果需要等待，加入waiting_for_all_missing队列；如果不需等待，调用do_pg_op处理PG相关的操作。这里的PG操作，都是CEPH_OSD_OP_PGLS等类似的PG相关的操作，需要确保该PG上没有需要修复的对象，否则ls列出的对象就不准确。
- 5) 调用函数op_has_sufficient_caps检查是否有相关的操作权限。
- 6) 检查对象的名字是否超长。
- 7) 检查操作的客是户端是否在黑名单（blacklist）中。
- 8) 检查磁盘空间是否满。
- 9) 检查如果是写操作，并且是snap，返回-EINVAL，快照不允许写操作。如果写操作带的数据大于osd_max_write_size（如果设置了），直接返回-OSD_WRITETOOBIG错误。

可以看到，以上完成基本的与操作相关的参数检查。

10) 构建要访问对象的head对象（head对象和快照对象的概念可查看后面介绍快照的章节）。

11) 如果是顺序写，调用函数scrubber.write_blocked_by_scrub检查：如果head对象正在进行scrub操作，就加入waiting_for_active队列，等待scrub操作完成后继续本次请求的处理。

12) 检查head对象是否处于缺失状态（missing）需要恢复，调用函数wait_for_unreadable_object把当前请求加入相应的队列里等待恢复完成。

13) 如果是顺序写，检查head对象是否is_degraded_or_backfilling_object，也就是正在恢复状态，需要调用wait_for_degraded_object加入相应的队列等待。

14) 检查head对象的特殊情况：

a) 检查队列objects_blocked_on_degraded_snap里如果保存的head对象，就需要等待。该队列里保存的head对象在rollback到某个版本的快照时，该版本的snap对象处于缺失状态，必须等待该snap对象恢复，从而完成rollback操作。因此该队列的head对象目前处于缺失状态。

b) 队列objects_blocked_on_snap_promotion里的对象表示head对象 rollback到某个版本的快照时，该版本的快照对象在Cache pool层没有，需要到Data pool层获取。

如果head对象在上述的两个队列中，head对象都不能执行写操作，需

要等待获取快照对象，完成rollback后才能写入。

可知，以上10~14步骤是构建并检查head对象的状态是否正常。

15) 如果是顺序写操作，检查该对象是否在objects_blocked_on_cache_full队列中，该队列中的对象因Cache pool层空间满而阻塞写操作。



注意

当head对象被删除时，系统自动创建一个snapdir对象用来保存快照相关的信息。head对象和snapdir对象只能有一个存在，其都可以用来保存快照相关的信息。

16) 检查该对象的snapdir对象（如果存在）是否处于missing状态。

17) 检查snapdir对象是否可读，如果不能读，就调用函数wait_for_unreadable_object等待。

18) 如果是写操作，调用函数is_degraded_or_backfilling_object检查snapdir对象是否缺失。

19) 检查如果是CEPH_SNAPDIR类型的操作，只能是读操作。Snapdir对象只能读取。

20) 检查是否是客户端replay操作。

- 21) 构建对象oid, 这才是实际要操作的对象, 可能是snap对象也可能是head对象。
- 22) 调用函数检查maybe_await_blocked_snapset是否被block, 检查该对象缓存的ObjectContext如果设置为blocked状态, 该object有可能正在flush, 或者copy (由于Cache Tier), 暂时不能写, 需要等待。
- 23) 调用函数find_object_context获取object_context, 如果获取成功, 需要检查oid的状态。
- 24) 如果hit_set不为空, 就需要设置hit_set.hit_set和agent_state都是Cache tier的机制, hit_set记录cachepool中对象是否命中, 暂时不深入分析。
- 25) 如果agent_state不为空, 就调用函数agent_choose_mode设置agent的状态, 调用函数maybe_handle_cache来处理, 如果可以处理, 就返回。
- 26) 获取object_locator, 验证是否和msg里的相同。
- 27) 检查该对象是否被阻塞。
- 28) 获取src_abc, 也就是src_oid对应的ObjectContext :同样的方法, 对src_oid做各种状态检查, 然后调用find_object_context函数获取ObjectContext。
- 29) 如果是操作对象snapdir对象, 相关的操作就需要所有的clone对

象，获取clone对象的objectContext。对每一个clone对象，构建objectContext，并把它加入的src_obs中。

30) 创建opContext。

31) 调用execute_ctx (ctx) 。

总之，do_op主要检查相关对象的（head对象、snapdir对齐、src对象等）状态是否正常，并获取ObjectContext、OpContext相关的上下文信息。

4.get_object_context

本函数获取一个对象的ObjectContext信息。

```
ObjectContextRef ReplicatedPG::get_object_context(
    const hobject_t& soid, //soid要获取的对象
    bool can_create, //是否允许创建新的
    ObjectContext
        map<string, bufferlist> *attrs) //attrs对象的属性
```

关键是从属性OI_ATTR中获取object_info_t信息。具体过程如下：

- 1) 首先从LRU缓存object_contexts的中获取该对象的ObjectContext，如果获取成功，就直接返回结果。
- 2) 如果从LRU cache里没有查找到：

- a) 如果参数attrs值不为空，就从attrs里获取OI_ATTR的属性值。
- b) 否则调用函数pgbackend->objects_get_attr获取该对象的OI_ATTR属性值。如果获取失败，并且不允许创建，就直接返回ObjectContextRef () 的空值。
- 3) 如果成功获取OI_ATTR属性值，就从该属性值中decode后获取object_info_t的值。
- 4) 调用get_snapset_context获取SnapSetContext。
- 5) 调用相关函数设置obc相关的参数，并返回obc。

5.get_snapset_context

本函数获取对象的snapset_context结构，其过程和函数get_object_context类似。具体实现如下：

- 1) 首先从LRU缓存snapset_contexts获取该对象的snapset_context，如果成功，直接返回结果。
- 2) 如果不存在，并且can_create，就调用pgbackend->objects_get_attr函数获取SS_ATTR属性。只有head对象或者snapdir对象保存有SS_ATTR属性，如果head对象不存在，就获取snapdir对象的SS_ATTR属性值，根据获得的值，decode后获的SnapsetContext结构。

6.find_object_context

本函数查找对象的object_context，这里需要理解snapshot相关的知识。根据snap_seq正确与否获取相应的clone对象，然后获取相应的object_context。

```
int ReplicatedPG::find_object_context(const hobject_t& oid,           //要查找的对象
                                       ObjectContextRef *pobc,          //输出对象的
                                       ObjectContext
                                       bool can_create,                //是否需要创建
                                       bool map_snapid_to_clone,       //映射
                                       snapid到
                                       clone对象
                                       hobject_t *pmissing)             //如果对象不存在，返回缺失的对象
{
```

参数map_snapid_to_clone指该snap是否可以直接对应一个clone对象，也就是snap对象的snap_id在SnapSet的clones列表中。

- 1) 如果是head对象，就调用函数get_object_context获取head对象的ObjectContext，如果失败，设置head对象为pmissing对象，返回-ENOENT；如果成功，返回0。
- 2) 如果是snapdir对象，先获取head对象的ObjectContext，如果失败，继续获取snapdir对象的ObjectContext，如果失败，返回返回-ENOENT；如果成功，返回0。

3) 如果非map_snapid_to_clone并且该snap已经标记删除了，就直接返回-ENOENT， pmissing为空，意味着该对象确实不存在。

4) 调用函数get_snapset_context来获取SnapSetContext，如果不存在，设置pmissing为head对象，返回-ENOENT。

5) 如果是map_snapid_to_clone：

a) 如果oid.snap大于ssc->snapset.seq，说明该snap是最新做的快照，osd端还没有完成相关的信息更新，直接返回head对象object_context，如果head对象存在，就返回0，否则返回-ENOENT。

b) 否则，直接检查SnapSet的clones列表，如果没有，就直接返回-ENOENT。

c) 如果找到，检查对象如果处于missing，pmissing就设置为该clone对象，返回-EAGAIN。如果没有，就获取该clone对象的object_context。

6) 如果不是map_snapid_to_clone，就不能从snap_id直接获取clone对象，需要根据snaps和clones列表，计算snap_id对应的clone对象：

a) 如果oid.snap>ssc->snapset.seq，获取head对象的ObjectContext。

b) 计算oid.snap首次大于ssc->snapset.clones列表中的clone对象，就是oid对应的clone对象。

c) 检查该clone对象如果missing，设置pmissing为该clone对象，返回-

EAGAIN。

- d) 获取该clone对象的ObjectContext。
- e) 最后检查该clone对象是如果first和last之间，这是合理的情况，返回0；否则就是异常情况，返回-ENOENT。

本函数是获取实际对象的ObjectContext，如果不是head对象，就需要获取快照对象实际对应的clone对象的ObjectContext。

7.execute_ctx

在do_op函数里，做了大量的对象状态的检查和上下文相关信息的获取，本函数开始执行相关的操作。

```
void ReplicatedPG::execute_ctx(OpContext *ctx)
```

处理过程如下：

- 1) 首先在OpContext中创建一个新的事务，该事务为pgbackend定义的事务。

```
ctx->op_t = pgbackend->get_transaction()
```

- 2) 如果是写操作，更新ctx->snapc值。ctx->snapc值保存了该操作的客户端附带的快照相关信息：

- a) 如果是给整个pool的快照操作， 就设置ctx->snapc等于pool.snapc的值。
- b) 如果是用户特定快照（目前只有rbd实现）， ctx->snapc值就设置为消息带的相关信息：

```
ctx->snapc.seq = m->get_snap_seq();  
ctx->snapc.snapc = m->get_snaps();
```

- c) 如果设置了CEPH OSD FLAG ORDERSNAP标志， 客户端的snap_seq比服务端的小， 就直接返回-EOLDSNAPC错误码。
- 3) 如果是read操作， 该对象的ObjectContext加ondisk_read_lock锁；对于源对象， 无论读写操作， 都需要加ondisk_read_lock锁。



所谓的源对象，就是一个操作中带两个对象，比如copy操作，源对象会有读操作。

- 4) 调用函数prepare_transaction把相关的操作封装到ctx->op_t的事务中。如果是读操作，对于replicate类型，该函数直接调用pgbackend->objects_read_sync同步读取数据。如果是EC，就把请求加入pending_async_reads完成异步读取操作。
- 5) 解除操作3中加的相关的锁。

- 6) 如果是读操作，并且ctx->pending_async_reads为空，说明是同步读取，调用complete_read_ctx完成读取操作，给客户端返回应答消息。如果是异步读取，就调用函数ctx->start_async_reads完成异步读取。读操作到这里就结束。后续都是写操作的流程。
- 7) 调用calc_trim_to，计算需要trim的pg log的版本。
- 8) 调用函数issue_repop向各个副本发送同步操作请求。
- 9) 调用函数eval_repop，检查发向各个副本的同步操作是否已经reply成功，做相应的操作。

从上可以看出，execute_ctx操作把相关的操作打包成事务，并没有真正的对对象的数据做修改。

8.calc_trim_to

本函数用于计算是否应该将旧的pg log日志进行trim操作：

```
void ReplicatedPG::calc_trim_to()
```

处理过程如下：

- 1) 首先计算target值：target值为最少保留的日志条数，默认设置为配置项cct->_conf->osd_min_pg_log_entries的值。如果pg处于degraded，或者正在修复的状态，target值为cct->_conf->osd_max_pg_log_entries（默认

10000条)。

2) 变量min_last_complete_ondisk为本pg在本osd上的完成最后一条日志记录的版本。如果它不为空，且不等于pg_trim_to，当前pg log的size大于target值，就计算需要trim掉的日志的条数：

- a) num_to_trim为日志总数目减去target，如果它小于日志一次trim的最小值cct->conf->osd_pg_log_trim_min，就返回。
- b) 否则，从日志头开始计算最新的pg_trim_to版本。

9.prepare_transaction

本函数用于把相关的更新操作打包为事务，包括比较复杂的部分为对象的snapshot的处理：

```
int ReplicatedPG::prepare_transaction(OpContext *ctx)
```

处理过程如下：

- 1) 首先调用函数ctx->snapc.is_valid() 来验证SnapSet的有效性。
- 2) 调用函数do_osd_ops打包请求到ctx->op_t的transaction中。
- 3) 如果事务为空，或者没有修改操作，就直接返回result。
- 4) 检查磁盘空间是否满。

- 5) 如果该对象是head对象，就有相关快照对象COW机制的操作，需要调用函数make_writeable来完成，在关于快照的介绍中会详细介绍到。
- 6) 调用函数finish_ctx来完成后续处理，该函数主要完成了快照相关的处理。如果head对象存在，就删除snapdir对象；如果不存在，就创建snapdir对象，用来保存快照相关的信息。后文会进一步介绍。

10.issue_repop

```
void ReplicatedPG::issue_repop(RepGather *repop, OpContext *ctx)
```

本函数的处理过程如下：

- 1) 首先更新actingbackfill的osd对应的peer_info的相关信息：如果pinfo.last_update和pinfo.last_complete二者相等，说明该peer的状态处于clean状态，就同时更新二者，否则只更新pinfo.last_update值。
- 2) 对该对象的ObjectContext的ondisk_write_lock加写锁，如果有clone对象，对该clone对象的ObjectContext的ondisk_write_lock加写锁。如果snapset_obi不为空，也就是可能创建或者删除snapdir对象，对该ObjectContext的ondisk_write_lock加锁。
- 3) 如果pool是可以rollback的（也就是ErasureCode模式），检查pg log也应该支持rollback操作。
- 4) 分别设置三个回调Context，调用函数pgbackend->submit_transaction

来完成事务向从osd的发送。

本函数调用pgbackend的submit_transaction函数向从osd开始发送操作日志。

6.4.3 阶段3：PGBackend的处理

PGBackend为PG的更新操作增加了一层与PG类型相关的实现。对于Replicate类型的PG由类ReplicatedBackend实现。其核心处理过程是把封装好的事务分发到该PG对应的其他从OSD上；对于ErasureCode类型的PG由类ECBackend实现，其核心处理过程为主chunk向各个分片chunk分发数据的过程。下面着重介绍Replicate的处理方式。

ReplicatedBackend :: submit_transaction函数最终调用网络接口，把更新的请求发送给从OSD，其处理过程如下：

- 1) 首先构建InProgressOp请求记录。
- 2) 调用函数ReplicatedBackend :: issue_op把请求发送出去：对于该PG中的每一个从OSD：
 - a) 调用函数generate_subop生成MSG OSD REPOP类型的请求。
 - b) 调用函数get_parent () → send_message_osd_cluster把消息发送出去。
- 3) 最后调用parent → queue_transactions函数来完成自己，也就是该PG的主OSD上本地对象的数据修改。

6.4.4 从副本的处理

当PG的从副本OSD接收到MSG OSD REPOP类型的操作，也就是主副本发来的同步写的操作时，处理流程和上述流程都一样。在函数sub_op_modify_impl里，对本地存储应用相应的事务，完成本地对象的数据写入。

6.4.5 主副本接收到从副本的应答

当PG的主副本接收到从副本的应答消息MSG OSD REPOPREPLY时，处理流程和上述类似，不同之处在于，在函数ReplicatedPG::do_request里调用了函数Replicated-Backend::handle_message，在该函数里调用了ReplicatedBackend::sub_op_modify_reply函数处理该请求。

sub_op_modify_reply函数的处理过程如下：

- 1) 首先在in_progress_ops中查找到该请求。
- 2) 如果是ondisk的ACK，也就是事务已经应答，就在ip_op.waiting_for_commit删除该OSD，该事务已经应答，那么必定已经提交了，那么从ip_op.waiting_for_applied删除该OSD。
- 3) 如果只是事务提交到日志中的ACK，就从ip_op.waiting_for_applied删除。



注意

这里特别说明的是，从副本需要给主副本发送两次ACK，一次是事务提交到日志中，并没有应答到实际的对象数据中，一次是完成应答操作返回的ACK。

- 4) 最后检查，如果ip_op.waiting_for_applied为空，也就是所有从OSD

的请求都返回来了，并且ip_op.on_applied不为空，就调用该Context的complete函数。同样，检查ip_op.waiting_for_commit为空，就调用该Context的函数的complete函数。

下面看一下，in_progress_ops注册的回调函数。其回调函数是在ReplicatedPG::issue_repop函数调用里注册的。

```
Context *on_all_commit = new C OSD RepopCommit(this, repop);
Context *on_all_applied = new C OSD RepopApplied(this, repop);
Context *onapplied_sync = new C OSD OndiskWriteUnlock(
    repop->obc,
    repop->ctx->clone_obc,
    unlock_snapset_obc ? repop->ctx->snapset_obc : ObjectContextRef());
```

回调函数都最终调用了函数ReplicatedPG::eval_repop，其最终向client发送应答消息。这里强调的是，主副本必须等所有的处于up的OSD都返回成功的ACK应答消息，才向客户端返回请求成功的应答。

6.5 本章小结

本章介绍了OSD读写流程核心处理过程。通过本章的介绍，可以了解读写流程的主干的流程，并对一些核心概念和数据结构的处理做了介绍。当然读写流程是Ceph文件系统的核心流程，其实现细节比较复杂，还需要读者对照代码继续研究。目前在这方面的工作，许多都集中在提供Ceph的读写性能。其基本的方法更多的就是优化读写流程的关键路径，通过减少锁来提供并发，同时简化一些关键流程。

第7章 本地对象存储

本地对象存储模块完成了数据如何原子地写入磁盘，这就涉及事务和日志的概念。对象如何在本地文件系统中组织的代码实现在src/os中。本章将介绍在单个OSD上数据如何写入磁盘中。

目前有4种本地对象存储实现：

- FileStore：这是目前比较稳定，生产环境上使用的主流对象存储引擎，也是本章重点介绍的对象存储引擎。
- BlueStore：这是目前社区在实现的一个新版本，社区丢弃了本地文件系统，自己写了一个简单的，专门支持RADOS用户态的文件系统。
- KStore：这是以本地KV存储系统实现的对象存储，它基于RODOS的框架用来实现一个分布式的KV存储系统。
- Memstore：它把数据和元数据都保存在内存中，用来测试和验证使用。

KStore和Memstore两种存储引擎比较简单，这里就不介绍了。BlueStore社区还正在开发之中，这里也暂时不介绍。本章将详细介绍目前在生产环境中使用的FileStore存储的实现。

7.1 基本概念介绍

RADOS本地对象存储系统（也称为对象存储引擎）基于本地文件系统实现，目前默认的文件系统为XFS。一个对象包含数据和元数据两种数据。对应本地文件系统里，一个对象就是一个固定大小（默认4MB）的文件，其元数据保存在文件的扩展属性或者本地独立的KV存储系统中。

7.1.1 对象的元数据

对象的元数据就是用于对象描述信息的数据，它以简单的key-value（键值对）形式存在，在RADOS存储系统中有两种实现：xattrs和omap：

- xattrs保存在对象对应文件的扩展属性中，这要求支持对象存储的本地文件系统支持扩展属性。

- omap就是object map的简称，是一些键值对，保存在本地文件系统之外的独立的key-value存储系统中，例如leveldb、rockdb等。

有些本地文件系统可能不支持扩展属性，有些虽然也支持扩展属性但对key或者value占用空间的大小有限制，或者扩展属性占的总的空间大小有限制。对于leveldb等本地键值存储系统基本没有这样的限制，但是它的写性能优越于读性能。所以一般情况下，xattrs保存一些比较小而经常访问的信息。omap保存一些大而不是经常访问的数据。

7.1.2 事务和日志的基本概念

假设磁盘正在执行一个操作，此时由于发生磁盘错误，或者系统宕机，或者断电等其他原因，导致只有部分数据写入成功。这种情况就会出现磁盘上的数据有一部分是旧数据，部分是新写入的数据，使得磁盘数据不一致。

当一个操作要么全部成功，要么全部失败，不允许只有部分操作成功，就称这个操作具有原子性。引入事务和日志，就是为了实现操作的原子性，解决数据的不一致性问题。

引入日志后，数据写入变为两步：1) 先把要写入的数据全部封装成一个事务，其整体作为一条日志，先写入日志文件并持久化的磁盘，这个过程称为日志的提交（journal submit）。2) 然后再把数据写入对象文件中，这称为日志的应用（journal apply）。

当系统在日志提交的过程中出错，系统重启后，直接丢弃不完整的日志条目即可，该条日志对应的实际对象数据并没有修改，数据可以保持一致。当在日志应用的过程中出错，由于数据已经写入并回刷到日志盘中，系统重启后，重放（replay）日志，就可以保证新数据重新完整写入，保证了数据的一致性。

这个机制需要确保所有的更新操作都是幂等操作。所谓幂等操作，就

是数据的更新可以多次写入，不会产生任何副作用。对象存储的操作一般都具有幂等性。

在事务的提交过程中，一条日志记录可以对应一个事务。为了提高日志提交的性能，一般都允许多条事务并发提交，一条日志记录可以对应多个事务，批量提交。所以事务的提交过程，一般和日志的提交过程是一个概念。

日志有三个处理阶段，对应过程分别为：

- 日志提交（journal submit）：日志写入日志磁盘。
- 日志的应用（journal apply）：日志对应的修改更新到对象的磁盘文件中。这个修改不一定写入磁盘，可能缓存在本地文件系统的页缓存（page cache）中。
- 日志的同步（Journal sync或者journal commit）：当确定日志对应的修改操作已经刷回到磁盘中，就可以把相应的日志记录删除，释放出所占的日志空间。

7.1.3 事务的封装

ObjectStore的内部类Transaction用来实现相关的事务。它有两种封装形式，一种是use_tbl，事务把操作的元数据和数据都封装在bufferlist类型的tbl中：

```
bool use_tbl;      //标记是否使用  
tbl来  
encode/decode  
bufferlist tbl;
```

另一种是不使用tbl，把元数据操作以struct Op的结构体，封装在op_bl中，把操作相关的数据封装在dataz_bl中：

```
bufferlist data_zl;  
bufferlist op_zl;  
bufferptr op_ptr; //操作临时指针
```

由于结构体op里保存的是coll_id和object_id，所以需要保存coll_t到coll_id的映射关系，以及ghobject_t和object_id的映射关系。从这种存储方式就可以看出，这种方式和前一种的区别，是在数据封装上实现了一种压缩。当事务中多个操作有相同的对象时，只保存一次ghobject_t结构体，其他情况只保存index来索引。

```
map<coll_t, __le32> coll_index; //coll_t -> coll_id的映射关系
```

```
map<ghobject_t, __le32, ghobject_t::BitwiseComparator> object_index;
//ghobject_t -> object_id的映射关系

__le32 coll_id; //当前分配的
coll_id的最大值

__le32 object_id; //当前分配的
object_id的最大值
```

数据结构TransactionData记录了一个事务中有关操作的统计信息：

```
struct TransactionData {
    __le64 ops; //本事务中操作数目

    //以下记录了事务中的带的数据最大的操作的：

    __le32 largest_data_len; //最大的数据长度

    __le32 largest_data_off; //在对象中的偏移

    __le32 largest_data_off_in_tbl; //在
tbl中的偏移

    __le32 fadvise_flags; //一些标志

}
```

一个事务中，对应如下三类回调函数，分别在事务不同的处理阶段调用。当事务完成相应阶段工作后，就调用相应的回调函数来通知事件完成。注意每一类都可以注册多个回调函数：

```
list<Context *> on_commit;
list<Context *> on_applied;
```

```
list<Context *> on_applied_sync;
```

on_commit是事务提交完成之后调用的回调函数；on_applied_sync和on_applied都是事务应用完成之后的回调函数。前者是被同步调用执行，后者是在Finisher线程里异步调用执行。

7.2 ObjectStore对象存储接口

下面通过对对象存储的接口说明和代码示例，可以了解对象存储的基本功能及如何使用这些功能，从而对对象存储有一个概要了解。

7.2.1 对外接口说明

类ObjectStore是对象存储系统抽象操作接口。所有的对象存储引擎都有继承并实现它定义的接口。下面列出一些有代表性的函数接口。

·objectStore初始化相关的函数：

·mkfs 创建objectstore相关的系统信息。

·mount 加载objectsotre相关的系统信息。

·statfs 获取objectstore系统信息。

·获取属性已经collection相关的信息：

·getattr 获取对象的扩展属性xattr。

·omap_get 获取对象的omap信息。

·queue_transactions是所有ObjectStore更新操作的接口。更新相关的操作（例如创建一个对象，修改属性，写数据等）都是以事务的方式提交给ObjectStore，该函数被重载成各种不同的接口。其参数为：

·Sequencer*osr 用于保证在同一个Sequencer的操作是顺序的。

·list<Transaction*>&tls 要提交的事务，或者事务的列表。

·Context*onreadablehe和Context*onreadable_sync这个函数分别对应事务的on_apply和on_apply_sync，当事务apply完成之后，修改后的数据可以被读取了。

·Context*ondisk 这个回调函数就是事务进行on_commit后的回调函数。

7.2.2 ObjectStore代码示例

下面通过一段ObjectStore的测试代码，来展示ObjectStore的基本功能和接口使用：

- 1) 首先调用create方法创建一个ObjectStore实例。参数分别为配置选项CephContext，对象存储的类型：filestore。对象存储的目录名、日志文件名如下：

```
ObjectStore *store_ = ObjectStore::create(g_ceph_context,
                                         string("filestore"),
                                         string("store_test_temp_dir"),
                                         string("store_test_temp_journal"));
```

- 2) 创建并加载ObjectStore本地对象存储：

```
store_->mkfs();
store_->mount();
```

- 3) 创建了一个collection，并写数据到对象中。对象的任何写操作，都先调用事务的相关写操作。事务把相关操作的元数据和数据封装，然后调用store的apply_transaction来真正实现数据的更改。

```
ObjectStore::Sequencer osr("test");
coll_t cid;
ghobject_t hoid(hobject_t(sobject_t("Object 1", CEPH_NOSNAP)));
bufferlist bl;
bl.append("1234512345");
int r;
ObjectStore::Transaction t;      //创建一个事务
```

```
t.create_collection(cid, 0);      //创建一个  
collection  
t.write(cid, hoid, 0, bl.length(), bl);    //写对象数据  
  
r = store->apply_transaction(&osr, std::move(t));  
                                //事务的应用，真正实现数据的写入操作  
  
store->umount();  
r = store->mount();
```

7.3 日志的实现

在本地对象中，日志是实现操作一致性的机制。在介绍Filestore之前，首先需要了解Journal的机制。本节首先介绍Journal的对外接口，通过对对外提供的功能接口，可以了解日志的基本功能。然后详细介绍FileJournal的实现。

7.3.1 Jouanal对外接口

通过Ceph中的test_filejournal.cc的一段测试程序，可以比较清楚地了解Jouanal的外部接口，通过这些外部接口可以了解Journal的功能，以及如何使用Jouarnal：

```
FileJournal j(fsid, finisher, &sync_cond, path, directio, aio);
j.create();
j.make_writeable();
bufferlist bl;
bl.append("small");
j.submit_entry(1, bl, 0, new C_SafeCond(&wait_lock, &cond, &done));
wait();
j.close();
```

通过上述代码可以了解到，日志的基本使用方法如下所示：

- 1) 创建一个FileJournal类型的日志对象，其构造函数path为日志文件的路径。
- 2) 调用日志的create方法创建日志，并调用函数make_writeable设置为可写状态。
- 3) 在bufferlist中添加一段数据作为测试的日志数据，调用submit_entry提交一条日志。
- 4) 等待日志提交完成。
- 5) 关闭日志。

7.3.2 FileJournal

类FileJournal以文件（包括块设备文件看做特殊文件）做为日志，实现了Journal的接口。下面先介绍FileJournal的数据结构和日志的格式，然后介绍日志的三个阶段：日志的提交（journal submit）、日志的应用（journal apply）、日志的同步（journal sync或者commit、trim）及其具体的实现过程。

1.FileJournal数据结构

FileJournal数据结构如下：

```
struct write_item {
    uint64_t seq;           //日志的序号

    bufferlist bl;          //日志的内容

    uint32_t orig_len;       //日志的原始长度

    TrackedOpRef tracked_op; //操作的跟踪记录

    write_item(uint64_t s, bufferlist& b, int ol, TrackedOpRef opref) :
        seq(s), orig_len(ol), tracked_op(opref) {
        bl.claim(b, buffer::list::CLAIM_ALLOW_NONSHAREABLE);
    }
    write_item() : seq(0), orig_len(0) {}
};

list<write_item> writeq;      //保存

write_item的队列

Mutex writeq_lock;           //writeq相关的锁

Cond writeq_cond;            //writeq相关的条件变量
```

结构体write_item封装了一条提交的日志， seq为提交日志的序号， bl保存了提交的日志数据， 也就是提交的事务或者事务的集合序列化后的数据。Journal并不关心日志数据的内容， 它认为一条日志就是一段写入的数据， 只负责把数据正确写入日志盘。orig_len记录日志的原始长度， 由于日志字节对齐的需要， 最终写入的数据长度可能会变化。tracked_op用于跟踪记录一些统计信息。

所有提交的日志， 都先保存到writeq这个内部的队列中， writeq_lock和writeq_cond是writeq相应的锁和条件变量：

```
struct completion_item {
    uint64_t seq;          //日志的序号

    Context *finish;        //完成后的回调函数

    utime_t start;         //提交时间

    TrackedOpRef tracked_op;
    completion_item(uint64_t o, Context *c, utime_t s,
                    TrackedOpRef opref)
        : seq(o), finish(c), start(s), tracked_op(opref) {}
    completion_item() : seq(0), finish(0), start(0) {}
};

Mutex completions_lock;
list<completion_item> completions;
```

结构体completion_item记录准备提交的item， 保存在列表completions中。由锁completions_lock保护。

```
off64_t write_pos;           //日志的写的开始位置
```

```

off64_t read_pos; //日志读取的位置

uint64_t writing_seq; //本次正在写入的最大

seq
uint64_t last_committed_seq; //最后同步完成的

seq
bool write_stop; //写线程是否停止的标记

enum {
    FULL_NOTFULL = 0,
    FULL_FULL = 1,
} full_state; //日志的磁盘空间的状态

//日志头部

struct header_t {
    enum {
        FLAG_CRC = (1<<0),
    };
    uint64_t flags; //日志的一些标志

    uuid_d fsid; //文件系统的

    id
    __u32 block_size; //日志文件或者设备的块大小

    __u32 alignment; //对齐

    int64_t max_size; //日志的最大

    size
    int64_t start; //offset of first entry日志条目的起始地址

    uint64_t committed_up_to; //已经提交的日志的

    seq, 等同于

    journaled_seq
    uint64_t start_seq; //等同于日志的起始,
    last_committed_seq + 1
} header;

```

结构体header_t是日志文件的头信息，保存日志相关的全局信息。

```
deque<pair<uint64_t, off64_t> > journalq;
// seq -  
> end用于记录日志在日志文件中的结束偏移，用于日志删除时可以知道偏移  
  
uint64_t writing_seq; //当前正在写入的最大的  
seq  
bool must_write_header; //是否必须写入一个  
header，当日志同步完成后，就设置本标准，必须写入一个  
header，以持久化保存应日志同步而变化了的  
header结构  
  
Mutex write_lock;
//特别需要指出，以上数据结构都受  
write_lock的保护  
  
Mutex finisher_lock;
Cond finisher_cond;
uint64_t journaled_seq; //已经提交成功的日志的最大  
seq
```

2. 日志的格式

日志的格式如下：

```
entry_header_t + journal data + entry_header_t
```

每条日志数据的头部和尾部都添加了entry_header_t结构。此外，日志在每次同步完成的时候设置must_write_header为true时会强制插入一个日志头header_t的结构，用于持久化存储header中变化了的字段。日志的结构如图7-1所示。

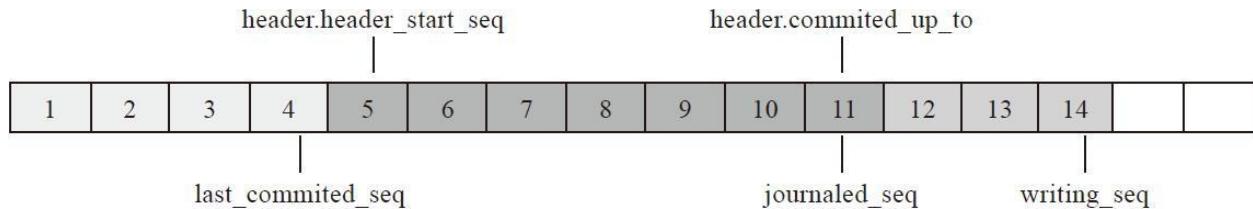


图7-1 日志的结构图

日志记录1到4为已经由日志同步时trim掉的日志（逻辑删除），
 last_committed_seq记录了最后一个trim的日志序号，当前值为4。日志记录5
 到11为已经提交成功日志。journaled_seq记录已经提交成功的日志最大
 序号，当前值为11。日志记录12到14为正在提交，还没有完成的日志。
 writing_seq记录正在提交的最大日志序号，当前值为14。

header_start_seq记录提交成功的第一条有效日志，当前显示为5，
 header.committed_up_to为提交成功的最后一条日志，当前值为11。

3.日志的提交

函数FileJournal::submit_entry用于提交日志。

```
submit_entry
FileJournal::submit_entry(uint64_t seq, bufferlist& e, uint32_t orig_len, Context
```

其处理过程如下：

- 1) 其分别加writeq_lock锁和completions_lock锁。
- 2) 然后构造completion_item和write_item两个数据结构体，并分别加
入到completions队列和writeq队列中。

3) 如果writeq队列为空，就通过writeq_cond.Signal() 触发写线程。
(如果writeq不为空，说明线线程一直在忙碌处理请求，不会睡眠，就没有必要再次触发了)。

write_thread线程是FileJournal内部的一个工作线程。该线程的处理函数write_thread_entry实现了不断地从writeq队列中取出write_item请求，并完成把日志数据写入日志磁盘的工作。

write_thread_entry函数调用prepare_multi_write函数，把多个write_item合并成到一个bufferlist中打包写入，可以提高日志写入磁盘的性能。配置项g_conf->journal_max_write_entries为一次写入的最大日志条目数量，配置项g_conf->journal_max_write_bytes为一次合并写入的最大字节数。

prepare_multi_write函数调用batch_pop_write函数，从writeq里一次获取所有的write_item，对每一个write_item调用函数prepare_single_write处理。

prepare_single_write函数实现了把日志数据封装成日志格式的数据：

- 1) 在数据的前端和后端都添加了一个entry_header_t的数据结构。
- 2) journalq记录了该条日志在日志文件（或者设备）中的结束偏移量，它在日志同步时，用于逻辑删除该条日志：

```
journalq.push_back(pair<uint64_t, off64_t>(seq, queue_pos));
```

- 3) 更新writing_seq的值为当前日志的seq，并更新当前日志写入的

queue_pos的值。当queue_pos达到日志的最大值时，需要从日志的头开始：

```
writing_seq = seq;
queue_pos += size;
if (queue_pos >= header.max_size)
queue_pos = queue_pos + get_top() - header.max_size;
```

当调用函数prepare_multi_write把日志封装成需要的格式后，就需要把数据写入日志磁盘。目前有两种实现方式：一种是同步写入；另一种是Linux系统提供的异步IO的方式。

同步写入方式直接调用do_write函数实现，该函数具体过程如下：

- 1) 首先判断是否需要写入日志的header头部：日志在每隔配置选项设定的g_conf->journal_write_header_frequency日志条数后或者must_write_header设置时插入一个header头。
- 2) 其次计算如果日志数据超过了日志文件最大长度，就需要把该条日志的数据分两次写入：尾端写入一部分，然后绕到头部写入一部分。
- 3) 如果不是directIO，就需要调用函数fsync把日志数据刷到磁盘上。
- 4) 加finisher_lock，更新journalized_seq的值。
- 5) 如果日志为不满的状态，并且plug_journal_completions没有设置，就调用函数queue_completions_thru来把completion_item的callback函数加入Finisher。最后删除该completion_item项。

异步IO的方式写入是日志写入的另一种方式，是使用了Linux操作系统内核自带的异步（aio）。Linux内核实现的aio，必须以directIO的方式写入。aio由于内核实现，数据不经过缓存直接落盘，性能要比同步的方式高很多。

异步IO相关的数据结构如下：

```
struct aio_info {      //一条
    aio的记录信息

    struct iocb iocb;    //提交的异步
    aio控制块

    bufferlist bl;      //aio的数据

    struct iovec *iov;   //异步
    aio方式的数据

    bool done;           //是否完成的标志

    uint64_t off, len;  //< these are for debug only
    uint64_t seq;        //aio的序号

};

io_context_t aio_ctx;      //异步
io的上下文

list<aio_info> aio_queue; //已经提交的
aio请求队列

Mutex aio_lock;           //aio队列的锁

Cond aio_cond;            //控制
```

inflight的

aio不能过多，否则需要等待

```
Cond write_finish_cond; //aio完成，触发完成线程去处理
```

```
int aio_num, aio_bytes; //正在进行的
```

aio的数量和数据量

异步IO的方式写日志，调用了do_aio_write函数，基本过程和do_write相似，只是写入的方式调用了write_aio_bh函数。

函数write_aio_bh实现了调用Linux的aio的API异步写入磁盘。实现过程如下：

1) 首先把数据转换成iov的格式。

2) 构造aio_info数据结构，该结构保存了aio的上下文信息，并把它加入aio_queue队列里。

3) 调用函数io_prep_pwritev和函数io_submit，异步提交请求：

```
io_prep_pwritev(&aio.iocb, fd, aio iov, n, pos)
int r = io_submit(aio_ctx, 1, &piocb);
```

4) 触发write_finish_thread检查是异步IO是否完成：

```
aio_lock.Lock();
write_finish_cond.Signal();
aio_lock.Unlock();
```

函数write_finish_thread_entry为线程write_finish_thread的执行函数，用
来检查aio是否完成。其具体实现如下：

- 1) 调用函数io_getevents来检查是否有aio事件完成的：

```
io_event event[16];
int r = io_getevents(aio_ctx, 1, 16, event, NULL);
```

- 2) 对每个完成的事件，获取相应的aio_info结构。检查写入的数据是
否是aio的数据长度，然后设置ai→done为true，标记该aio已经完成。
- 3) 调用函数check_aio_completion检查aio_queue队列的完成情况。由
于aio的并发提交，多条日志可能并发完成，为了保证日志是按照seq的顺
序完成，必须从头按顺序检查aio_queue的完成情况。
- 4) 更新journalized_seq的值，调用queue_completions_thru完成后续工
作，这和同步实现工作一样。

4. 日志的同步

日志的同步是在日志已经成功应用完成，对应的日志数据已经确保写
入磁盘的日志后，就删除相应的日志，释放出日志空间的过程。

函数committed_thru用来实现日志的同步。它只是完成了日志的同步功
能。日志同步时，必须确保日志应用完成，这个逻辑在Filestore里完成。

函数committed_thru具体实现如下：

- 1) 确保当前同步的seq比上次last_committed_seq大。
- 2) 用seq更新last_committed_seq的值。
- 3) 把journalq中小于等于seq的记录删除掉，这些记录已经无用了。
- 4) 删除相应的日志。这里只是逻辑上的删除，只是修改header.start指针就可以了。当journalq为空，就删除到设置write_pos位置。如果journalq不为空，就设置为队列的第一条记录的偏移值，这是最新日志的起始地址，也就是上一条日志的结尾。
- 5) 设置must_write_header为true，由于header更新了，日志必须把header写入磁盘。
- 6) 日志可能因日志空间满而阻塞写线程，此时由于释放出更多空间，就调用commit_cond.Signal () 唤醒写线程。

7.4 FileStore的实现

在介绍完本地对象存储的概念、对外接口和日志实现后，本节介绍FileStore的实现。首先看一下FileStore的静态类图，如图7-2所示。

FileStore的静态类图说明如下：

- ObjectStore是对象存储的接口。FileStore继承了JournalingObjectStore类。
- JournalingObjectStore实现了和日志之间的交互。
- FileStoreBackend定义了本地文件系统支持Filestore的一些接口，不同的文件系统会增加自己支持特性实现了不同的FileStoreBackend，例如BtrfsFileStoreBackend和XfsFilestoreBackend分别对应XFS和BTRFS两种文件系统的实现。
- Journal类定义了日志的抽象接口，FileJournal实现了以本地文件或者本地设备为存储的日志系统。

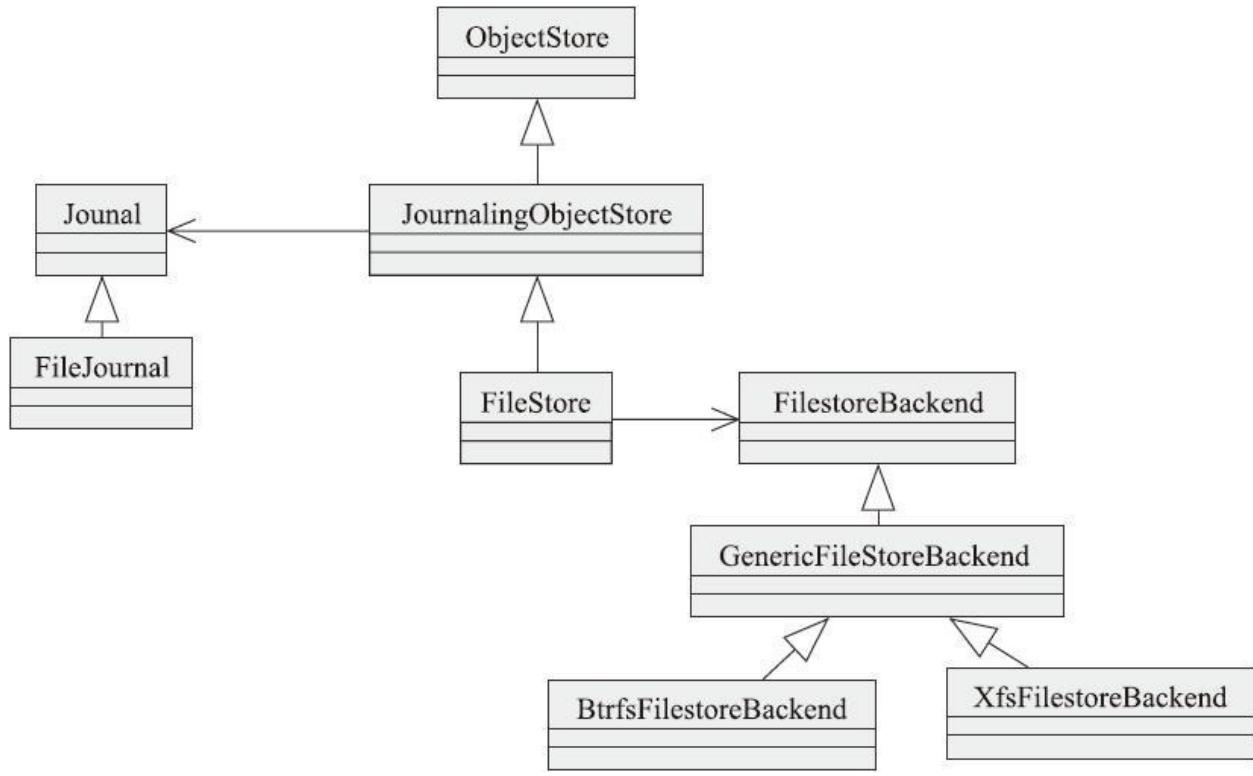


图7-2 `FileStore`的静态类图

7.4.1 日志的三种类型

在FileStore里，根据日志提交方式的不同，有三种类型的日志：

·journal writeahead：日志数据先提交并写入日志磁盘上，然后再完成日志的应用（更新实际对象数据）。这种方式适合XFS、EXT4等不支持快照的本地文件系统使用这种方式。

·journal parallel：日志提交到日志磁盘上和日志应用到实际对象中并行进行，没有先后顺序。这种方式适用于BTRFS和ZFS等实现了快照操作的文件系统。由于具有文件系统级别快照功能，当日志的应用过程出错，导致数据不一致的情况下，文件系统只需要回滚到上一次快照，并replay从上次快照开始的日志就可以了。显然，这种方式比writeahead方式性能更高。但是由于btrfs和zfs目前在Linux都不稳定，这种方式很少用。

·不使用日志的情况，数据直接写入磁盘后才返回客户端应答。这种方式目前FileStore也实现了，但是性能太差，一般不使用。

本节只介绍FileStore默认的方式：journal writeahead方式的日志实现。

7.4.2 JournalingObjectStore

类JournalingObjectStore实现了ObjectStore和FileJournal的一些交互管理。它通过类SubmitManager和类ApplyManager分别实现了日志提交和日志应用的管理。下面分别介绍这两个类。

1. SubmitManager

类SubmitManager实现了日志提交的管理，准确地说是日志序号的管理。函数op_submit_start给提交的日志分配一个序号。函数op_submit_finish在日志提交完成后验证：当前的op的seq等于上次提交的序号op_submit_finish加1。最后更新op_submit_finish的值。

```
class SubmitManager {
    Mutex lock;
    uint64_t op_seq;           // 日志最新的序号

    uint64_t op_submit_finish; // 日志上次提交的序号

}
```

2. ApplyManager

ApplyManager负责日志应用的相关处理：

```
class ApplyManager {
    Journal *&journal;
    Finisher &finisher;
    Mutex apply_lock;
    bool blocked;      // 日志应用是否阻塞，当前日志同步时需要
```

```
Cond blocked_cond;
int open_ops;      //正在进行日志

apply的

ops的数量

uint64_t max_applied_seq;    //日志应用完成的最大的

seq
Mutex com_lock;
map<version_t, vector<Context*> > commit_waiters;
//日志完成后的回调函数，用于没有日志的类型

uint64_t committing_seq,      //正在应用的日志的

seq
committed_seq;      //已经应用完成的

seq
}
```

函数ApplyManager :: op_apply_start在日志应用前调用，其实现如下操作：

1) 给加apply_lock锁。

2) 当blocked设置时，就等待，暂停日志的应用，这个后面日志同步时会说明。

3) 统计值open_ops加1。

函数ApplyManager :: op_apply_finish在日志应用完成后调用，其实现如下操作：

1) 加apply_lock锁。

- 2) 统计值open_ops自减。
- 3) 如果blocked为true，就调用blocked_cond.Signal（）发通知。
- 4) 更新max_applied_seq的值为应用完成的最大日志号。

函数ApplyManager::commit_start用于在日志同步时计算目前完成的最大日志seq。

ApplyManager::commit_started表示日志开始同步，commit_finish函数在日志同步结束时调用。

7.4.3 Filestore的更新操作

先介绍数据结构。结构Op代表了一个ObjectStore的操作的上下文信息
在FileStore里的封装：

```
struct Op {
    utime_t start;      //日志应用的开始时间

    uint64_t op;        //日志的

    seq
    list<Transaction*> tls;  //事务列表

    Context *onreadable,*onreadable_sync;//事务应用完成之后的回调函数和同步回调函数

    uint64_t ops, bytes;  //操作数目和字节数, 这里的

    ops指的是对象的基本操作例如

    create,

    write,

    delete等基本操作

    ,一个

    Op带多个

    Transaction, 可能带多个基本操作。

    TrackedOpRef osd_op;
};
```

类OpSequencer用于实现请求的顺序执行。在同一个Sequencer类的请
求，保证执行的顺序，包括日志commit的顺序和apply的顺序。一般情况
下，一个PG对应一个Sequencer类。所以一个PG里的操作都是顺序执行。

```

class OpSequencer{
    Mutex qlock; //本来所保护队列

    q, 在
    flush时,
    peek和
    degueue也被该锁保护

    list<Op*> q;           //操作序列

    list<uint64_t> jq;     //日志序号

    list<pair<uint64_t, Context*>> flush_commit_waiters;
    Cond cond;
public:
    Sequencer *parent;
    Mutex apply_lock;      //日志应用的互斥锁

    int id;
}

```

1.更新实现

FileStore的更新操作分两步：首先把操作封装成事务，以事务的形式整体提交日志并持久化到日志磁盘，然后再完成日志的应用，也就是修改实际对象的数据。

```

int FileStore::queue_transactions(Sequencer *posr, vector<Transaction>& tls,
                                  TrackedOpRef osd_op,
                                  ThreadPool::TPHandle *handle)

```

FileStore更新的入口函数为queue_transactions函数。参数posr用于确保执行的顺序；参数tls是Transaction的列表，一次可以提交多个事务；TrackedOpRef用于跟踪信息。

具体处理流程如下：

1) 首先调用collect_contexts函数把tls中所有事务的on_applied类型的回调函数收集在一个list<Context*>结构里，然后调用list_to_context函数把Context列表变成了一个Context类。这里实际就是包装成一个回调函数。on_commit和on_applied_sync都做了类似的工作。

2) 构建op对象。把相关的操作封装到op对象里：

```
Op *o = build_op(tls, onreadable, onreadable_sync, osd_op);
```

3) 调用函数prepare_entry把所有日志的数据封装到tbl里。

```
int orig_len = journal->prepare_entry(o->tls, &tbl);
```

4) 分配一个日志的序号，并设置在op结构里。

```
uint64_t op_num = submit_manager.op_submit_start();
o->op = op_num;
```

5) 如果日志是m_filestore_journal_parallel模式，就调用函数_op_journal_transactions开始提交日志。然后调用函数queue_op，它把请求添加到op_wq里同时开始日志的应用。

6) 如果日志是m_filestore_journal_writeahead模式，就把该op添加到osr中，调用_op_journal_transactions函数提交日志。

7) 调用函数submit_manager.op_submit_finish (op_num) , 通知submit_manager日志提交完成。

2.writeahead和parallel的处理方式的不同

函数_op_journal_transactions用于提交日志。当有日志并且日志可写时，就调用journal的submit_entry函数提交日志。当日志提交成功后，就会调用_op_journal_transactions注册的回调函数onjournal。

函数queue_op用于把操作添加的Filestore内部的线程池对应队列中OpWq中。OpWq的调用_do_op函数完成日志的应用，也就是完成实际的操作。

对于日志parallel方式，日志的提交和应用并发进行：

```
_op_journal_transactions(tbl, orig_len, o->op, ondisk, osd_op);
//加入到提交队列里

queue_op(osr, o);
```

对于writeahead方式，先调用函数_op_journal_transactions提交日志，其注册的回调函数onjournal为C_JournaledAhead：

```
osr->queue_journal(o->op);
_op_journal_transactions(tbl, orig_len, o->op,
new C_JournaledAhead(this, osr, o, ondisk),
osd_op);
```

在回调函数的C_JournaledAhead类里，其finish函数调用

`_journaled_ahead`, 该函数调用`queue_op (osr, o)` 把相应请求添加到`op_wq.queue (osr)` 工作队列中。工作队列的处理线程实现了日志`apply`操作，完成实际对象数据的修改。

从上可以看出，`writeahead`是先完成了日志的提交，然后才开始日志的应用。`Parallel`方式是同时完成日志的提交和应用。

如果日志是第三种类型，即没有日志的形式，就调用函数`apply_manager.add_waiter`把`Contex`添加到`commit_waiters`中。

7.4.4 日志的应用

操作队列OpWq用于完成日志的应用。处理函数_process调用_do_op函数来完成应用日志，实现真正的修改操作。

函数_do_op函数实现操作如下：

- 1) 首先调用osr->apply_lock.Lock () 加锁，通过该锁，实现了同一时刻，osr里只有一个操作在进行。
- 2) 调用op_apply_start，通知apply_manager类开始日志的应用。
- 3) 调用_do_transactions函数，用于完成日志的应用。它对每一个事务调用_do_transaction函数，解析事务，执行相应的操作。
- 4) 调用op_apply_finish (o->op) 通知apply_manager完成。

7.4.5 日志的同步

在FileStore内部会创建sync线程，用来定期同步日志，该线程的入口函数为：

```
void FileStore::sync_entry()
```

函数sync_entry定期执行同步操，处理过程如下：

- 1) 调用函数tp.pause ()，用来暂停FileStore的op_wq的线程池，等待正在应用的日志完成。
- 2) 然后调用fsync同步内存中的数据到数据盘，当同步完成后，就可以丢弃相应的日志，释放相应的日志空间。

7.5 omap的实现

omap的静态类图7-3所示。类ObjectMap定义了omap的抽象接口。类DBObjectMap实现了以KeyValueDB的本地存储实现的ObjectMap的接口。

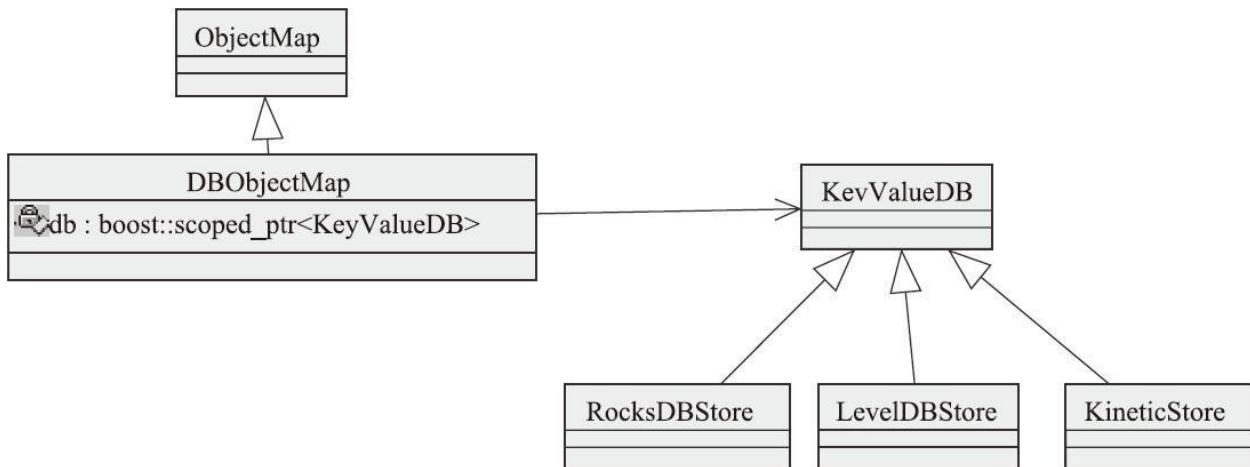


图7-3 omap静态类图

目前实现KeyValueDB的本地存储分别为：Facebook开源的levelDB存储，对应类LevelDBStore；Google开源的RocksDB存储，对应类RocksDBStore实现；KineticStore存储对应的类kineticStore。默认采用LevelDB实现。

表7-1是LevelDB是一个key-value的存储系统，它是一维的flat模式的KV存储。表7-2是对象存储，多个不同对象的多个不同属性的二维存储模式。

二者如何映射呢？由于对象的名字是全局唯一的，属性在对象内也是

唯一的，所以在LevelDB层面就可以用Object名字和属性的名字联合作为在LevelDB的key，这是能想到的比较直观的解决方式。

例如：object1的属性（key1， value1）的保存在LevelDB如下：

```
(object1_name + key1,    value1)
```

这种方法的一个缺点：当一个对象有多个KV值时，Object1的name多次作为key存储，由于Object的name一般比较长，这样存储方式浪费空间比较大。于是就提出了一种压缩的存储方法，也就是目前omap的存储方式。

表7-1 levelDB的flat的KV存储模式

Key	Value
Key1	Value1
Key2	Value2
.....
KeyN	Value N

表7-2 对象的omap存储模式

Object1	Key1	Value1
	Key2	Value2

Object2	Key1	Value1
	Key2	Value2

.....		

7.5.1 omap存储

目前omap的在leveldb中存储分两步：

- 1) 在LevelDB中，保存键值对：

```
key:  
HOBJECT_TO_SEQ + ghobject_key(oid)  
value:  
header
```

HOBJECT_TO_SEQ是固定的前缀标识字符串，函数ghobject_key获取对应的对象唯一的key字符串。

header保存对象在LevelDB中的唯一标识seq，以及支持快照的父对象的信息，同时保存了对象的collection和oid（这里冗余保存，因为藏key信息里就可以获得）。

```
struct _Header {  
    uint64_t seq;           //自己在  
    levlldb中的序号  
  
    uint64_t parent;        //父对象的序号  
  
    uint64_t num_children;  //子对象的数量  
  
    coll_t c;               //对象所在的  
    collection  
    ghobject_t oid;         //对象标识
```

```
SequencerPosition spos; //保存了当前的日志的序号（  
op_seq, 日志内多个事务的序号, 每个事务内操作的序号,  
};
```

2) 对象的属性保存以下格式的键值对：

```
Key :  
USER_PREFIX + header_key(header->seq) + XATTR_PREFIX + key  
Value :  
value(omap的值
```

综上所述，设置和获取对象的属性，需要两步：先根据对象的oid，构造键（HOBJECT_TO_SEQ+ghobject_key（oid）），获取header；根据对象的header中的seq，拼接在levelDB中的key值

（USER_PREFIX+header_key（header->seq）+XATTR_PREFIX+key），获取value值。

变量state用于保存KeyValueDB的全局状态，目前只有seq信息。

```
struct State {  
    __u8 v;           //版本  
    uint64_t seq;    //全局分配的  
    seq  
}state;
```

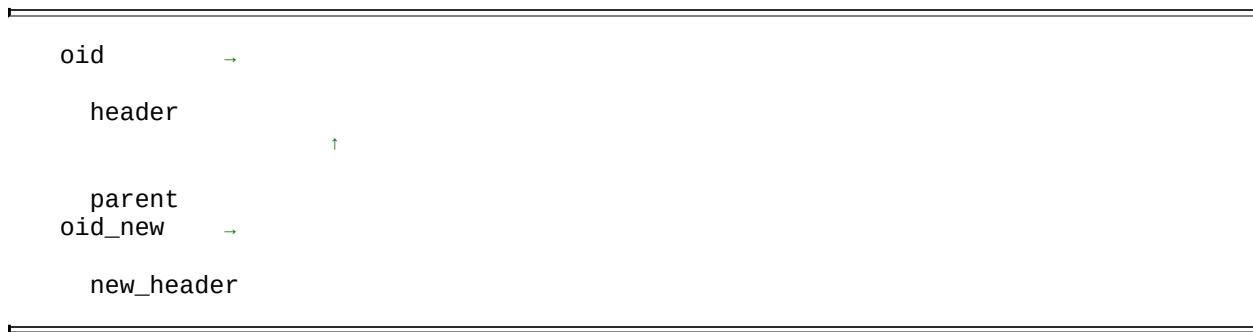
函数write_state用于每次分配seq后，把state信息写入LevelDB中，保

存：

```
SYS_PREFIX + GLOBAL_STATE_KEY -> state
```

7.5.2 omap的克隆

omap的克隆机制的实现如下所示：



- 当克隆一个新的对象old_new时，仅创建一个对应的new_header，并不是把该对象的所有属性都在leveldb中拷贝一遍，同时在new_header的parent字段保存header的seq号，从而建立了它们之间的父子联系。
- 当读取一个子对象的属性时，如果子对象不存在该属性，需要去父对象获取。

可以看出， omap的clone机制也实现了copy-on-write机制。

7.5.3 部分代码实现分析

1.SequencerPosition

类SequencerPosition用来验证操作的顺序性，当操作执行时，后面的操作序号必须大于之前的操作序号。

```
struct SequencerPosition {  
    uint64_t seq; // 日志的序号  
  
    uint32_t trans; // 一个日志内多个事务的序号  
  
    uint32_t op; // 一个事务内多个  
    op的序号  
}
```

2.lookup_create_map_header

本函数用于获取对象的header：

- 1) 首先调用函数_lookup_map_header查找对象的header：
 - a) 首先在caches里查找是否缓存。
 - b) 调用底层KeyValueDB查找header

```
int r = db->get(HOBJECT_TO_SEQ, map_header_key(oid), &out);
```

c) 如果查找成功，就返回header对象，如果不成功，返回一个新创建的header对象。

2) 调用函数_generate_new_header来设置Header的字段，并调用函数write_state写入全部变量state：

```
header->seq = state.seq++;
if (parent) {
    header->parent = parent->seq;
    header->spos = parent->spos;
}
header->num_children = 1;
header->oid = oid;
```

3) 调用函数set_map_header，把新的header设置到LevelDB中。

3.get_xattrs

本函数用于获取对象的属性，实现如下：

1) 首先获取对应的Header头部。

2) 调用db设置具体的数据：

```
Header header = lookup_map_header(h1, oid);
if (!header)
    return -ENOENT;
return db->get(xattr_prefix(header), to_get, out);
```

4.set_keys

本函数用于设置对象的属性，实现如下：

1) 获取KeyValueDB::Transaction的一个事务。

```
KeyValueDB::Transaction t = db->get_transaction();
```

2) 先获取对象的Header：

```
MapHeaderLock hl(this, oid);
Header header = lookup_create_map_header(hl, oid, t);
if (!header)
    return -EINVAL;
if (check_spos(oid, header, spos))
    return 0;
```

3) 先调用事务set函数设置属性：

```
t->set(user_prefix(header), set);
```

4) 调用db提交事务：

```
return db->submit_transaction(t)
```

7.6 CollectionIndex

Collection的概念对应到本地文件系统中就是一个目录，用于存储一个PG里的所有的对象。

一个collection对应本地文件系统的一个目录，一个PG对应于一个Collection，该PG的所有对象都保存在这个目录里，定义在类coll_t中：

```
class coll_t {
    enum type_t {
        TYPE_META = 0,
        TYPE_LEGACY_TEMP = 1, //这个类型不再使用
        TYPE_PG = 2,
        TYPE_PG_TEMP = 3,
    };
    type_t type ;      类型
    meta,
    pg,
    temp
    spg_t pgid;      对应的
    pgid
    uint64_t removal_seq; //这个字段不再使用，没有编码持久化存储
    string _str; //缓存的字符串
};
```

collection有三种不同的类型：TYPE_META类型表示这个PG里保存的是元数据（meta）相关的对象，TYPE_PG表示该collection保存的是PG相关的数据，TYPE_PG_TEMP保存临时对象。

当一个PG的对象数量比较多时，就会在一个目录里保存大量的文件。对于底层文件系统来说，如果一个目录里保存大量文件，当达到一定的程度后，性能会急剧下降。那么就需要一个collection里对应多个层级的子目录来存储大量文件，从而提高性能。

图7-4为CollectionIndex的静态类图。IndexManager类为管理CollectionIndex的实现。HashIndex实现了LFNIndex，LFNIndex实现了CollectionIndex接口。

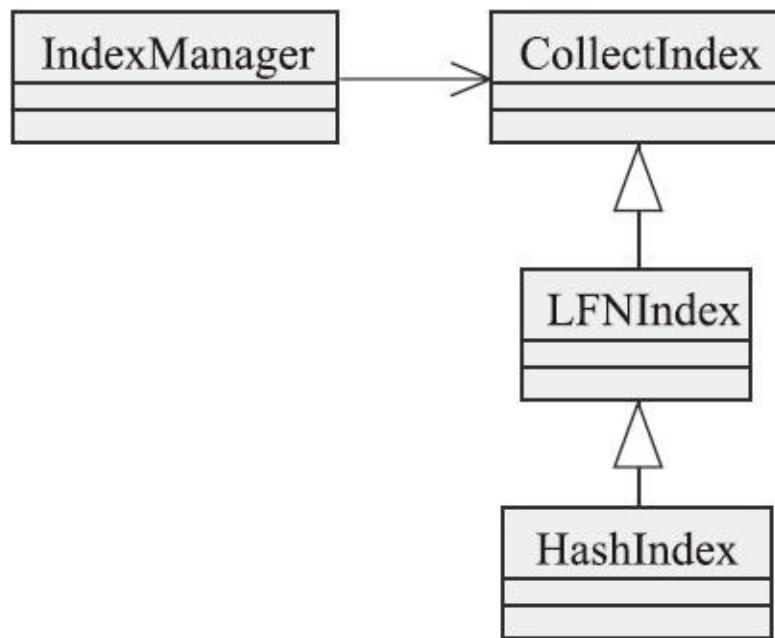


图7-4 CollectionIndex静态类图

7.6.1 CollectIndex接口

CollectionIndex使一个Collection里的对象保存在多层次子目录中。类CollectionIndex是对象在文件系统中多层目录存储的接口。

通过接口说明，就可以看到其能提供的功能：

·查找一个对象，返回对象对应文件的存储路径：

```
virtual int lookup(const ghobject_t &oid, //输入参数，要查找的对象  
IndexedPath *path, //输出参数，对象的路径  
int *hardlink  
) = 0;
```

·根据对象的路径，创建一个对象：

```
virtual int created(  
const ghobject_t &oid, //创建对象名  
const char *path //创建对象路径  
) = 0;
```

·删除一个对象：

```
virtual int unlink(  
const ghobject_t &oid //要删除的对象  
) = 0;
```

· 分裂目录，当上次目录里保存的文件或者子目录达到一定数量，就需要分裂成两个目录：

```
virtual int split(
    uint32_t match,           // 输入参数：符合本子目录的
    bit值

    uint32_t bits,            // 输入参数：要检查的
    bit
    CollectionIndex* dest   // 输入参数：目标
    index集合

) { assert(0); return 0; }
```

· 根据对象的hash值，按序列出对象：

```
virtual int collection_list_partial(
    const ghobject_t &start,  // 输入参数：对象的起始位置

    const ghobject_t &end,    // 输入参数：对象的结尾

    bool sort_bitwise,        // 输入参数：对象排序的顺序

    int max_count,            // 输入参数：最大对象数目

    vector<ghobject_t> *ls,   // 输出参数：对象列表

    ghobject_t *next          // 输出参数：下一次的对象起始

) = 0;
```

从上述接口介绍可知，CollectionIndex提供了对象到其对应文件保存的目录路径映射管理。

7.6.2 HashIndex

HashIndex是CollectionIndex的一个实现。HashIndex实现了用对象的Hash值做为对象存储的目录。

1.对象保存目录方式

以对象的HASH值为基准，从低位到高位十六进制的字符保存。

例如对象ghobject_t ("object", CEPH_NO_SNAP, 0xA4CEE0D2) 的Hash值是：0xA4CEE0D2，如果当前保存的目录层级是2级，那么该对象保存的目录为：

```
root/DIR_2/DIR_D
```

如果是3级目录，那么该对象保存的目录就是：

```
root/DIR_2/DIR_D/DIR_0
```

2.目录层级

如何确定当前保存目录的层级呢？何时创建一个新的子目录呢？当一个目录中的对象数目超过如下值：

```
abs(merge_threshold)) * 16 * split_multiplier
```

就重新创建一个子目录，原来的子目录要分裂为两个目录。其中 merge_threshold由配置选项g_ceph_context->_conf->filestore_merge_threshold设置。split_multiplier由g_conf->filestore_split_multiple设置。

一个目录保存对象的统计信息保存在目录的扩展属性中，数据结构 subdir_info_s定义了相关的属性；

```
struct subdir_info_s {
    uint64_t objs;           //该目录中对象的数目
    uint32_t subdirs;        //该目录中子目录的数目
    uint32_t hash_level;    //子目录的
    hash层级数
}
```

7.6.3 LFNIndex

HashIndex继承了LFNIndex接口。LFNIndex是Long File Name Index的缩写。从名称就可以知道，LFNIndex用来处理如下情况：当对象名太长，超过了本地文件系统支持的长度时，LFXIndex实现把超出的部分文件名保存到文件的扩展属性中。有可能保存到扩展属性的多个key-value存储对中。

7.7 本章小结

本章介绍本地对象存储的基本概念，以及ObjectStore对象存储接口，通过它可以了解对象存储如何调用。然后介绍了Journal的对外接口和Filejournal的实现，以及Filestore的更新操作。最后介绍了对象omap实现以及对象在本地文件系统中的组织方式。

目前对象存储是研究的热点。通过上述介绍可知，FileStore的日志方式的写操作都需要数据的两次写入：一次写日志，另一次写数据对象。对于S3接口的对象存储等应用场景，双写是没有必要的。社区推出了新的存储引擎BlueStore，其解决了某些场景下的避免双写，同时提高了性能。但目前BlueStore还不稳定，不能用于生产环境。

第8章 Ceph纠删码

本章介绍Ceph纠删码（Erasure Code, EC）的实现。首先介绍纠删码的原理，并对几种不同的编码原理进行分析，然后介绍纠删码的具体实现。其实现过程大部分逻辑和副本类似，这里着重介绍能完成数据回滚操作的不同实现点。

8.1 EC的基本原理

纠删码（EC）是最近一段时间存储领域（特别是在云存储领域）比较流行的数据冗余存储的方法，它的原理和传统的RAID类似，但是比RAID方式更灵活。

它将写入的数据分成N份原始数据，通过这N份原始数据计算出M份效验数据。把N+M份数据分别保存在不同的设备或者节点中，并通过N+M份中的任意N份数据块还原出所有数据块。

EC包含了编码和解码两个过程：将原始的N份数据计算出M份效验数据称为编码过程；通过这N+M份数据中的任意N份数据来还原出原始数据的过程称为解码过程。EC可以容忍M份数据失效，任意小于等于M份的数据失效能通过剩下的数据还原出原始数据。

目前一些主流的云存储厂商都采用EC编码方式。Google GFS II中采用了最基本的RS（6, 3）编码，Facebook的HDFS RAID的早期编码方式为RS（10, 4）编码。微软的云存储系统Azure使用了的LRC（12, 2, 2）编码。

8.2 EC的不同插件

Ceph支持以插件的形式来指定不同的EC编码方式。各种编码的不同点，实质就是在ErasureCode的三个指标之间折中的结果，这个三指标是：空间利用率、数据可靠性和恢复效率。

8.2.1 RS编码

目前应用最广泛的纠删码是ReedSolomon编码，简称RS码。这种编码在1960年出现了，过了很多年以后才提出了快速算法并广泛应用于各种通信系统中，直到近几年才逐渐应用于各种存储系统中。下面介绍RS编码的几个实现。

1.Jerasure

Jerasure是一个ErausreCode开源实现库，它实现了EC的RS编码。目前Ceph中默认的编码就是Jerasure方式。

2.ISA

ISA是Intel提供的一个EC库，只能运行在Intel CPU上，它利用了Intel处理器本地指令来加速EC的计算。

RS编码的不足之处在于：在N+K个数据块中有任意一块数据失效，都需要读取N块数据来恢复丢失数据。在数据恢复的过程中引起的网络开销比较大。因此，LRC编码和SHEC编码分别从不同的角度做了相关优化。

8.2.2 LRC编码

LRC编码的核心思想为：将校验块（parity block）分为全局校验块（global parity）和局部校验块（local reconstruction parity），从而减少恢复数据的网络开销。其目标在于解决当单个磁盘失效后恢复过程的网络开销。

LRC (M, G, L) 的三个参数分别为：

· M 是原始数据块的数量。

· G 为全局校验块的数量。

· L 为局部校验块的数量。

编码过程为：把数据分成 M 个同等大小的数据块，通过该 M 个数据块计算出 G 份全局效验数据块。然后把 M 个数据块平均分成 L 组，每组计算出一个本地数据效验块，这样共有 L 个局部数据校验块。

下面以Azure的LRC (12, 2, 2) 和Facebook的HDFS RAID的早期编码方式RS (10, 4) 为例来比较LRC和RS编码在恢复过程的开销。参见表8-1。

表8-1 LRC编码举例及其与RS的比较

LRC (12,2,2)	RS (12+4)
$D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_{10} D_{11} D_{12}$ $L_1 \quad \quad \quad L_2$ $G_1 \quad G_2$	$D_1 \sim D_{12} \quad P_1 \sim P_4$

表8-1所示对应LRC编码：总共有12个数据块，分别为 $D_1 \sim D_{12}$ 。有两个本地数据校验块 L_1 和 L_2 ， L_1 为通过第一组数据块 $D_1 \sim D_6$ 计算而得的本地效验数据块； L_2 为第二组数据块 $D_7 \sim D_{12}$ 计算而得的本地效验数据块。有2个全局数据效验块 G_1 和 G_2 ，它是通过所有数据块 $D_1 \sim D_{12}$ 计算而来。对应RS编码，数据块为 $D_1 \sim D_{12}$ ，计算出的效验块为 $P_1 \sim P_4$ 。

不同情况下的数据恢复开销：

- 如果数据块 $D_1 \sim D_{12}$ 只有一个数据块损坏， LRC只需要读取6个额外的数据块来恢复。而RS需要读取12个其他的数据块来修复。
- 如果 L_1 或者 L_2 其中一个数据块损坏， LRC需要读取6个数据块。如果 G_1 ， G_2 其中一个数据损坏， LRC仍需要读取12个数据块来修复。

最大允许失效的数据块：

·RS允许数据块和校验块中任意的小于等于4个数据的失效。

·而对于LRC：

·数据块中，只允许任意的小于等于2个数据块失效。

·允许所有的效验块 (G_1 ， G_2 ， L_1 ， L_2) 同时失效。

- 允许至多两个数据块和两个本地效验块同时失效。

综上分析可知：对于只有一个数据块失效，或者一个本地数据效验块失效的情况下，在恢复该数据块时，LRC比RS可以减少一半的磁盘IO和网络带宽。所以LRC重点在单个磁盘失效后恢复的优化。但是对于数据可靠性来说，通过最大允许失效的数据块个数的讨论可知，LRC会有一定的损失。

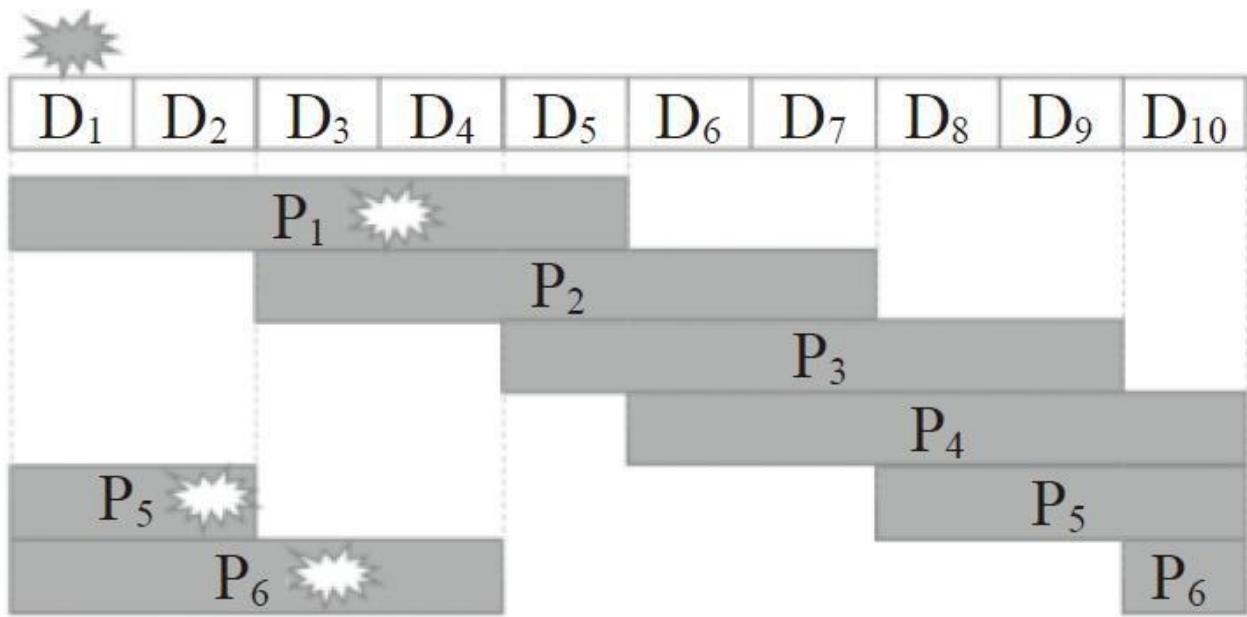
8.2.3 SHEC编码

SHEC编码方式为SHEC (K, M, L) , 其中K代表data chunk的数量, M代表parity chunk的数量, L代表计算parity chunk时需要的data chunk的数量。其最大允许失效的数据块为 : ML/K 。这样恢复失效的单个数据块只需要额外读取L个数据块。

下面以SHEC (10, 6, 5) 为例, 其最大允许失效的数据块为 :

$$M(6) * L(5) / K(10) = 3$$

如图8-1所示的, $D_1 \sim D_{10}$ 位数据块, P_1 为数据块 $D_1 \sim D_5$ 计算出的校验块。 P_2 为 $D_3 \sim D_7$ 计算出的校验块。其他校验块的计算如图所示。当一个数据块失效时, 只读取5个数据块就可以恢复。



SHEC(10, 6, 5)

图8-1 SHEC编码示意图

8.2.4 EC和副本的比较

在Ceph中可以设置一个pool为EC类型，并可以设置N和M的参数。副本和各种RC的编码比如表8-2所示。

表8-2 副本和各种RC的编码比较

	三副本	RS (10,4)	LRC (10,6,5)	SHEC (10,6,5)
数据容量开销	3X	1.4X	1.8X	1.6X
数据恢复开销 (单个数据块失效)	1X	10X	5X	5X
可靠性	高	中	中	中下

说明如下：

- 在三副本的情况下，恢复效率和可靠性都比较高，缺点就是数据容量开销比较大。
- 对于EC的RS编码，和三副本比较，数据开销显著降低，以恢复效率和可靠性为代价。
- LRC编码以数据容量开销略高的代价，换取了数据恢复开销的显著降低。
- SHEC编码用可靠性换代价，在LRC的基础上进一步降低了容量开销。

8.3 Ceph中EC的实现

8.3.1 Ceph中EC的基本概念

下面介绍一些EC的基本概念。注意，这里stripe的概念是指RADOS系统定义的，可能与其他系统的定义不同。

·chunk：一个数据块就叫data chunk，简称chunk，其大小为chunk_size设置的字节数。

·stripe：用来计算同一个校验块的一组数据块，称为data stripe，简称stripe，其大小为stripe_width，参与的数据块的数目为stripe_size，这几个概念的关系如下：

$$\text{stripe_width} = \text{chunk_size} \times \text{stripe_size}$$

如图8-2所示为一个EC (4+2) 示例：stripe_size为4，chunk_size的大小为1K，那么stripe_width的大小就为4K。在Ceph系统中，默认的stripe_width就为4K。

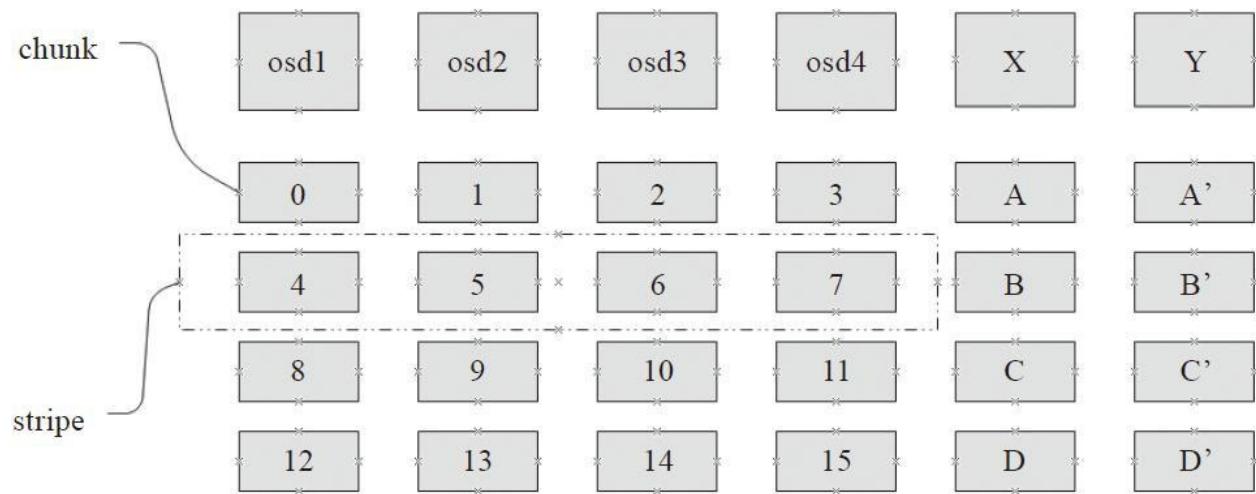


图8-2 EC的分片示意图

8.3.2 EC支持的写操作

目前Ceph的EC方式的写入操作是有一定限制的，其目前只支持如下操作：

- create object：创建对象。

- remove object：删除对象。

- write full：写整个对象。

- append write（stripe width aligned）：追加写入（限定追加操作的起始偏移以stripe_width对齐）。

目前Ceph只支持上述操作，而不支持overwrite操作，其主要有如下两个条件的限制：

- 由于编码和解码的过程都以stripe width整块数据计算。

- EC在特殊场景需要回滚的机制。

所以，目前EC只支持append写操作中，写操作的起始偏移offset以stripe_width对齐的情况，如果end不是以stripe_width对齐，就补0对齐即可。

目前不支持以下情况：

·情况1：append写操作，写操作的起始偏移offset没有以stripe_width对齐。

·情况2：overwrite写操作， offset和end都不以stripe_width对齐。

由于计算数据校验块需要读取整个stripe的数据块。所在情况1和情况2都需要读取该stripe缺失的数据块，来计算效验块。由于性能的原因，目前不支持。

·情况3：overwrite写操作，写操作的起始偏移offset和结束位置end都以stripe_width对齐。

情况3目前也不支持，其原因是由于EC的回滚的机制导致。下面将介绍EC的回滚机制。

8.3.3 EC的回滚机制

依据EC的原理可知，EC (N+M) 的写操作如果有小于等于M个OSD失效，不会导致数据丢失，数据可恢复。EC在理论上就最多只能容忍M个OSD失效。如果OSD失效的数量大于M，这种情况超出了理论设计的范畴，系统无法处理这种情况。可以说这是合理的。

但是对于所有的存储系统，必须应对一种特殊的情况：整个机房或者整个数据中心全局断电，系统重启后可恢复，并且数据不丢失。

当存储系统全局断电时，其数据的写入状态就有可能出现：小于N个磁盘的数据成功写入，而其他磁盘没有写成功的情况。

以图8-2所示的EC (4+2) 为例，假设写操作只有3个OSD写成功了，其他3个OSD没有来得及把数据写入磁盘。这种情况下，不但导致新数据写入失败，而且导致旧数据也无法读取成功。这就需要EC支持回滚机制，回滚到最后一次成功写入的旧数据版本。

Ceph目前支持的EC操作都是回滚比较容易实现的，实现机制如下：

- create object操作的回滚实现比较简单，删除该对象即可。
- 对于remove object操作，在执行时并不删除该对象，而是暂时保留该对象；如果需要回滚，就可以直接恢复。

·writeFull操作：暂时保留旧的对象，创建一个新的对象完成写操作。

当需要回滚时，恢复旧的数据对象。

·append操作：记录append时的size到PG日志中；当回滚时，对该对象做truncate操作即可。

8.4 EC的源代码分析

对应EC的上述三种更新操作，其本地回滚的信息都记录在对应的PG日志记录的mod_desc里：

```
struct pg_log_entry_t{
    .....
    ObjectModDesc mod_desc;
    .....
};
```

在函数ReplicatedPG :: do_osd_ops中实现操作的事务封装，下面着重分析一下EC的写操作和write_full操作的实现。

8.4.1 EC的写操作

操作步骤如下：

- 1) 首先验证如果是EC类型，写操作的offset必须以stripe_width对齐，否则不支持。

```
case CEPH_OSD_OP_WRITE:  
    if (pool.info.requires_aligned_append() &&  
        (op.extent.offset % pool.info.required_alignment() != 0)) {  
        result = -EOPNOTSUPP;  
        break;  
    }
```

- 2) 如果对象不存在，就在mod_desc中添加创建的信息，否则在mod_desc中添加old size的信息：

```
ctx->mod_desc.create();
```

否则就是追加写：

```
ctx->mod_desc.append(oi.size);
```

- 3) 最后把写操作添加到事务中：

```
if (pool.info.require_rollback())  
    t->append(soid, op.extent.offset, op.extent.length, osd_op.indata, op.flags);
```

8.4.2 EC的write_full

操作步骤如下：

- 1) 如果对象已经存在，调用函数ctx->mod_desc.rmobject，如果返回false，说明已经记录了信息，直接删除；如果返回true，就调用stash保存旧的对象数据，用来恢复：

```
case CEPH_OSD_OP_WRITEFULL:  
....  
  
if (obs.exists) {  
    if (ctx->mod_desc.rmobject(ctx->at_version.version)) {  
        t->stash(soid, ctx->at_version.version);  
    } else {  
        t->remove(soid);  
    }  
}
```

- 2) 在事务中写入数据：

```
t->append(soid, 0, op.extent.length, osd_op.indata, op.flags);
```

8.4.3 ECBackend

类ECBackend实现了EC的读写操作。ECUtil里定义了编码和解码的函数实现。ECTransaction定了EC的事务。相关的代码都比较清晰，这里就不详细介绍。

8.5 本章小结

本章介绍Ceph纠删码的编码原理，以及目前支持的操作和目前尚不支持其他操作的原因，并简单介绍了EC的代码实现。目前纠删码的研究是一个热点。它可以极大地提供存储利用率，降低存储成本。目前研究都在着力研究纠删码如何直接支持块存储，也就是随机overwrite操作的能力。

第9章 Ceph快照和克隆

本章介绍Ceph的高级数据功能：快照和克隆，它们在企业级的存储系统中是必不可少的。这里首先介绍Ceph中快照和克隆的基本概念，其次介绍快照实现相关的数据结构，然后介绍快照操作的原理，最后分析快照的读写操作的源代码实现。

9.1 基本概念

下面介绍快照和克隆的基本概念，以及二者之间的区别。

9.1.1 快照和克隆

快照是一个RBD在某一时刻全部数据的只读镜像。克隆是在某一时刻全部数据的可写镜像。快照和克隆都是某一时间点的镜像，区别在于快照只能读，而克隆可以写。

Ceph支持两种类型的快照：一种pool级别的快照（pool snap），是给pool整体做一个快照；另一种是用户管理的快照（self managed snap）。目前RBD快照的实现就属于后者。用户的写操作必须自己提供SnapContex信息。注意，这两种快照是互斥的，两种快照不能同时存在。也就是说，如果对pool整体做了快照操作，就不能对该pool中的RBD做快照操作。

无论是pool级别的快照，还是RBD的快照，其实现的基本原理都是相同的。都是基于对象的COW（copy-on-write）机制。Ceph可以完成秒级的快照操作和克隆操作。

这里需要特别指出的是，对象的clone操作指的是快照对应的克隆操作，是RADOS在OSD服务端实现的对象的拷贝。RBD的clone操作是RBD的客户端实现的RBD层面的克隆。它们俩不是一个概念，希望读者区分开来。

在具体的实现过程中，克隆依赖快照的实现，克隆是在一个快照的基础上实现了可写功能。

图9-1是快照和克隆的示意图，其生成过程如下所示：

- 1) 首先创建一个RBD块设备：rbd1。
- 2) 对该块设备rbd1创建一个快照snap1。
- 3) 调用rbd protect来保护该快照，则该快照就不能被删除。
- 4) 从该快照中克隆出一个新的image，其名字为clone1。

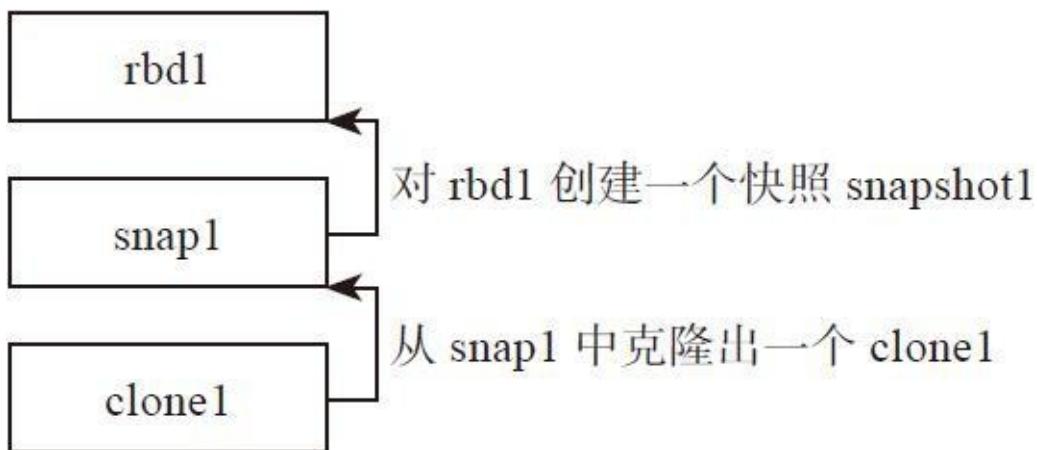


图9-1 快照和克隆示意图

一个image的数据对象和快照对象都在同一个pool中，每个image的对象和对应的快照对象都在相同OSD上的相同PG中。快照的对象拷贝都是在OSD本地进行。

9.1.2 RBD的快照和克隆比较

RBD的快照和克隆在实现层面完全不同。快照是RADOS支持的，基于OSD服务端的COW机制实现的。而RBD的克隆操作完全是RBD客户端实现的一种COW机制，对于OSD的Server端是无感知的。

怎么理解RBD的克隆操作是由RBD的客户端实现的？如图9-1所示的快照和克隆，对克隆的image的读写过程如下：

- 1) 当对克隆image，也就是clone1发起写操作时，客户端对应的OSD发送正常的写请求。
- 2) OSD返回给客户端应答，表明该OSD上对应的对象不存在。
- 3) 客户端要发读请求到给克隆image的父image，读取对应snap 1上的数据返回给客户端。
- 4) 客户端把该快照数据写入克隆image中。
- 5) 客户端给克隆image发送写操，写入实际要写入的数据。

由以上过程可知，克隆的拷贝操作是由客户端控制完成，OSD的Sever端配合完成普通的读写操作。

当克隆的层级比较多时，需要客户端不断递归到其父image上去读取对

应的快照对象，这会严重影响克隆的性能。

如图9-2所示，当读写clone2时，客户端首先给clone2发送写请求，如果对象不存在，就需要向clone1发送读请求，读取clone1对应的snap2的快照数据，并写入clone2，然后完成实际的写入操作。

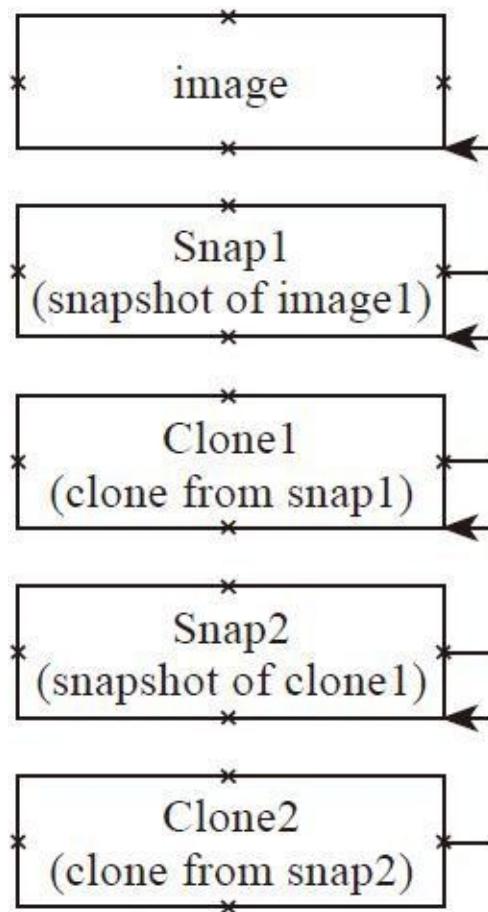


图9-2 多层级的克隆

如果clone1上的对象不存在，同样，客户端继续递归，不断给给父image发送读请求，读取snap1的快照数据，并写入clone1中。

如果层级过多就会影响克隆操作的性能。因此系统提供了RBD的

flattern操作，可直接把父image的快照对象拷贝给克隆image，这样以后就不需要去向父image查找对象数据了，从而提高了性能。

9.2 快照实现的核心数据结构

快照的核心数据结构如下：

- head对象：也就是对象的原始对象，该对象可以进行写操作。
- snap对象：对某个对象做快照后，通过cow机制copy出来的快照对象只能读，不能写。
- snap_seq或者seq：快照序号，每次做snapshot操作系统都分配一个相应快照序号，该快照序号在后面的写操作中发挥重要作用。
- snapdir对象：当head对象被删除后，仍然有snap和clone对象，系统自动创建一个snapdir对象，来保存SnapSet信息。head对象和snapdir对象只有一个存在，其属性都可以保存快照相关的信息。这主要用于文件系统的快照实现。

1.SnapContext

在文件librados/IoCtxImpl.h中定义了snap相关的数据结构：

```
struct SnapContext {  
    snapid_t seq;           //最新的快照序号  
  
    vector<snapid_t> snaps; //当前存在的快照序号，降序排队  
}
```

SnapContext数据结构用来在客户端（RBD端）保存snap相关的信息。

这个结构持久化存储在RBD的元数据中：

```
struct librados::IoCtxImpl {
    .....
    snapid_t snap_seq;
    ::SnapContext snapc;
    .....
}
```

其中：

·seq为最新的快照序号。

·snaps降序保存了该RBD的所有的快照序号。

数据结构IoCtxImpl里的snap_seq一般也称为快照的id（snap id）。当打开一个image时，如果打开的是一个卷的快照，那么snap_seq的值就是该snap对应的快照序号。否则snap_seq就为CEPH_NOSNAP（-2），来表示操作的不是卷的快照，而是卷自身。

2.SnapSet

数据结构SnapSet用于保存Server端（也就是OSD端）与快照相关的信息：

```
struct SnapSet {
    snapid_t seq;           //最新的快照序号

    bool head_exists;        //head对象是否存储
```

```
vector<snapid_t> snaps; //所有的快照序号列表（降序排列）

vector<snapid_t> clones; //所有的
clone对象序号列表（升序排列）

map<snapid_t, interval_set<uint64_t> > clone_overlap;
//和上次
clone对象之间
overlap的部分

map<snapid_t, uint64_t> clone_size;
//clone对象的
size
}
```

下面是其中一些数据字段介绍：

·seq保存最新的快照序号。

·head_exists保存head对象是否存在。

·snaps保存所有的快照序号。

·clones保存所有快照后的写操作需要clone的对象记录。

这里特别强调的是clones和snaps的区别。由于不是每次做快照操作后，都要拷贝对象。只当快照操作后有写操作，才会触发相关对象的clone操作复制出一份新的对象，该对象是clone出来的，其快照序号记录在clones队列中，称为clone对象。

·clone_overlap保存本次clone对象和上次clone对象（或者head对象）的

overlap的部分，也就是重叠的部分。clone操作后，每次写操作，都要维护这个信息。这个信息用于在数据恢复阶段对象恢复的优化。

- clone_size保存每次clone后的对象的size。

SnapSet数据结构持久化保存在head对象的xattr的扩展属性中：

- 在Head对象的xattr中保存key为snapset，value为SnapSet结构序列化后的值。

- 在snap对象的xattr中保存key为user.cephos.seq的snap_seq值。

9.3 快照的工作原理

9.3.1 快照的创建

RBD快照创建的基本步骤如下：

- 1) 向Monitor发送请求， 获取一个最新的快照序号snap_seq的值。
- 2) 把该次快照的snap_name和snap_seq的值保存到RBD的元数据中。

在RBD的元数据里保存了所有快照的名字和对应的snap_seq号，并不会触发OSD端的数据操作，所以非常快。

9.3.2 快照的写操作

当对一个image做了一次快照后，该image写入数据时，由于快照的存在需要启动copy-on-write（cow）机制。下面将介绍cow机制的具体实现。

客户端的每次写操作，消息中都必须带数据结构SnapContex信息，它包含了客户端认为的最新快照序号seq，以及该对象的所有快照序号snaps的列表。在OSD端，对象的Snap相关信息保存在SnapSet数据结构中，当有写操作发生时，处理过程按照如下规则进行。

规则1

如果写操作所带SnapContex的seq值小于SnapSet的seq值，也就是客户端最新的快照序号小于OSD端保存的最新的快照序号，那么直接返回-EOLDSNAP错误。

Ceph客户端始终保持最新的快照序号。如果客户端不是最新的快照序号，可能的情况是：在多个客户端的情形下，其他客户端有可能创建了快照，本客户端有可能没有获取到最新的快照序号。

Ceph有一套Watcher回调通知机制来实现快照序号的更新。如果其他客户端对一个卷做了快照，就会产生了一个最新的快照序号。OSD端接收到最新快照序号变化后，通知相应的连接客户端更新最新的快照序号。如果有客户端没有及时更新，也没有太大的问题，OSD端会返回客户端-

EOLDSNAP，客户端会主动更新为最新的快照序号，重新发起写操作。

规则2

如果head对象不存在，创建该对象并写入数据，用SnapContext相应的信息更新SnapSet的信息。

规则3

如果写操作所带SnapContex的seq值等于SnapSet的seq值，做正常的读写。

规则4

如果写操作所带SnapContex的seq值大于SnapSet的seq值：

- 1) 对当前head对象做copy操作，clone出一个新的快照对象，该快照对象的snap序号为最新的序号，并把clone操作记录在clones列表里，也就是把最新的快照序号加入到clones队列中。
- 2) 用SnapContex的seq和snaps值更新SnapSet的seq和snaps值。
- 3) 写入最新的数据到head对象中。

9.3.3 快照的读操作

快照读取数据时，输入参数为RBD的名字和快照的名字。RBD的客户端通过访问RBD的元数据，来获取快照对应的snap_id，也就是快照对应的snap_seq值。

在OSD端，获取head对象保存的SnapSet数据结构。然后根据SnapSet中的snaps和clones值来计算快照所对应的正确的快照对象。

9.3.4 快照的回滚

快照的回滚，就是把当前的head对象回滚到某个快照对象。具体操作如下：

- 1) 删除当前head对象的数据。
- 2) 拷贝相应的snap对象到head对象。

其源代码的实现在ReplicatedPG :: _rollback_to里。

9.3.5 快照的删除

删除快照时，直接删除rbd的元数据中保存的Snap相关快照信息，然后给Monitor发快照删除信息。Monitor随后给相应的OSD发送删除的快照序号，然后由OSD控制删除本地相应的快照对象。该快照是否被其他快照对象共享。

由上可知，Ceph的快照删除是延迟删除，并不是直接立即删除。

9.4 快照读写操作源代码分析

下面着重分析快照的写操作和读操作相关的源代码实现。

9.4.1 快照的写操作

在结构体OpContext的上下文中，保存了快照相关的信息：

```
struct OpContext {
    const SnapSet *snapset; //旧的
    SnapSet, 也就是
    OSD服务端保存的快照信息。
    ObjectState new_obs;
    SnapSet new_snapset; //新的
    SnapSet
    SnapContext snapc; //写操作带的，也就是客户端的
    SnapContext信息
};

}
```

在读写的关键流程中，有关快照的处理如下：

1) 在OSD写操作的流程中，在函数ReplicatedPG::execute_ctx中，把消息带的SnapContext信息保存在了OpContext的snapc中：

```
ctx->snapc.seq = m->get_snap_seq();
ctx->snapc.snap = m->get_snaps();
```

2) 在OpContext的构造函数里，用结构snapset字段初始化了结构new_snapset的相关字段。当前new_snapset保存的就是OSD服务端的快照信息：

```
if (obc->ssc) {
    new_snapset = obc->ssc->snapset;
    snapset = &obc->ssc->snapset;
}
```

3) 在函数ReplicatedPG`::prepare_transaction`里调用了函数ReplicatedPG`::make_writeable`来完成快照相关的操作。

9.4.2 make_writeable函数

函数make_writeable处理快照相关的写操作，其处理流程如下：

- 1) 首先判断，如果服务端的最新快照序号大于客户端的快照序号，就用服务端的快照信息更新客户端的快照信息：

```
if (ctx->new_snapset.seq > snapc.seq) {  
    snapc.seq = ctx->new_snapset.seq;  
    snapc.snap = ctx->new_snapset.snap;  
    dout(10) << " using newer snapc " << snapc << endl;  
}
```

在数据读写的流程中可知，在函数ReplicatedPG::execute_ctx里已经判断了：客户端的最新快照序号不能小于服务端的快照序号，否则就直接返回-EOLDSNAPC错误码客户端更新快照序号后重试。所以笔者认为这段代码不会进入，所以是无用的代码。

- 2) 调用函数filter_snapc把已经删除的快照过滤掉。
- 3) 如果head对象存在，并且snaps的size不为空（有快照），并且客户端的最新快照序号大于服务端的最新快照序号，在这种情况下要克隆对象，实现对象数据的拷贝了：
 - a) 构造clone对象coid，其coid.snap为最新的客户端seq值。
 - b) 计算snaps列表，也就是本次克隆对象对应的所有快照。

- c) 构造clone_abc，也就是克隆对象coid的ObjectContext。特别需要指出的是该克隆对象的object_info_t中的snaps信息，就是在上一步中计算出的snaps列表。
- d) 调用函数_make_clone实现对象的克隆操作。此时克隆操作都先封装在新创建的事务t中：

```
PGBackend::PGTransaction *t = pgbackend->get_transaction();
_make_clone(ctx, t, ctx->clone_abc, soid, coid, snap_oi);
t->append(ctx->op_t);
delete ctx->op_t;
ctx->op_t = t;
```

注意，之前的写操作封装在事务ctx->op_t中，把该事务追加到事务t的尾部，然后删除ctx->op_t事务，事务t赋值给ctx->op_t。这样在事务应用时，就是先做克隆操作，然后才完成写操作。

- 4) 最后把该克隆对象添加到ctx->new_snapset.clones中，并添加clone_size记录和clone_overlap记录。
- 5) 根据当前的写操作修改范围modified_ranges，来计算修改clone_overlap的记录，也就是当前head对象和上次克隆对象的重叠区域，该信息用来优化快照对象的恢复。
- 6) 更新服务端的快照信息为客户端的快照记录信息。

下面举例说明。

例9-1 快照写操作见表9-1。

表9-1 写操作示例

操作序列	操作	RBD 的元数据信息	OSD 端对应的对象
1	write data1 snapContext{ seq=0, snaps={ } }		SnapSet={ seq = 0, head_exists= true, snaps={}, clones={}, } obj1_head(data1)
2	create snapshot name=" snap1"	("snap1" ,1)	

(续)

操作序列	操作	RBD 的元数据信息	OSD 端对应的对象
3	write data2 snapContext{ seq=1, snaps={1} }		SnapSet={ seq = 1, head_exists= true, snaps={1}, clones={1}, } obj1_1(data1) obj1_head(data2)
4	create snapshot name=" snap3"	("snap1" ,1) ("snap3" ,3)	
5	create snapshot name=" snap6"	("snap1" ,1) ("snap3" ,3) ("snap6" ,6)	
6	write data3 snapContext{ seq=6, snaps={6,3,1} }		SnapSet={ seq = 6, head_exists= true, snaps={6,3,1}, clones={1,6}, } obj1_1(data1) obj1_6(data2) obj1_head(data3)

说明如下：

- 1) 在操作1里为第一次写操作，写入的数据为data1， SnapContext的初始seq为0， snaps列表为空。按规则2， OSD端创建对象并写入对象数据，用SnapContext的数据更新SnapSet中的数据。
- 2) 在操作2里，创建了该RBD一个快照，名字为snap1，并向Monitor申请分配一个快照序号，其值为1。在该卷的元数据里添加了快照的名字和对应的快照序号。
- 3) 操作3里，写入数据data2，写操作所带SnapContext中的seq值为1，snap列表为{1}。在OSD端处理，此时SnapContext的seq大于SnapSet的seq，操作按照规则4：
 - a) 更新SnapSet中的seq为1，snaps列表更新为{1}值。
 - b) 创建快照对象obj1_1，拷贝当前head对象的数据data1到快照对象obj1_1中（快照对象名字下划线后面为快照序号，Ceph目前快照对象的名字中含有快照序号）。此时快照对象obj1_1的数据为data1，并在clones中添加clone操作记录，clone列表的值为{1}。
 - c) 向head对象obj1_head中写入数据data2中。
- 4) 操作4和操作5连续做了两次快照操作，快照的名字分别为snap3和snap6，分配的快照序号分别为3，6（在Ceph里，快照序号是由Monitor分配的，全局唯一，所以单个RBD的快照序号不一定连续）。
- 5) 操作6写入数据data3，此时写操作所带SnapContext中的seq值为6，

snap值为{6, 3, 1}共三个快照。此时SnapSet的seq为1，操作按规则4处理过程如下：

- a) 更新SnapSet结构中的seq值为6, snap值为{6, 3, 1}。
- b) 创建快照对象obj1_6, 拷贝当前head对象的数据data2到快照对象obj1_6中，并把本次克隆操作记录添加到clone队列中。更新后的clone队列的值为{1, 6}。
- c) 向head对象obj1_head中写入数据data3。

9.4.3 快照的读操作

快照的读取操作核心在函数ReplicatedPG::find_object_context里实现，其原理是根据读对象的快照序号，查找实际对应的克隆对象的ObjectContext。基本的步骤如下：

- 1) 如果对象的快照序号oid.snap大于服务端的最新快照序号ssc->snapshot.seq，获取head对象就是该快照对应的实际数据对象。
- 2) 计算oid.snap首次大于ssc->snapshot.clones列表中的克隆对象，就是oid对应的克隆对象。

例如在例9-1中，最后的SnapSet为：

```
-----  
SnapSet = {  
    seq=6,  
    snaps={6, 3, 1},  
    clones={1, 6},  
    ...  
}  
-----
```

这时候读取seq为3的快照，由于seq为3的快照并没有写入数据，也就没有对应的克隆对象，通过计算可知，seq为3的快照和snap为1的快照对象数据是一样的，所以就读取obj1_1对象数据。

9.5 本章小结

Ceph的基于Copy-on-Write的机制实现了秒级别的快照，其高效率的核心原理在于做快照操作时不会直接拷贝数据，而是只做了快照的记录，当只有实际的写操作发生时，才实现对象的拷贝操作。实质就是把整个卷的拷贝操作开销分散到后续每次写操作过程中，这样就实现了快照操作只是增加了新的快照记录，所以快照操作可以在秒级实现。

第10章 Ceph Peering机制

本章介绍Ceph中比较复杂的模块：Peering机制。该过程保障PG内各个副本之间数据的一致性，并实现PG的各种状态的维护和转换。本章首先介绍boost库的statechart状态机基本知识，Ceph使用它来管理PG的状态转换。其次介绍PG的创建过程以及相应的状态机创建和初始化。然后详细介绍Peering机制三个具体的实现阶段：GetInfo、GetLog、GetMissing。

10.1 statechart状态机

Ceph在处理PG的状态转换时，使用了boost库提供的statechart状态机。因此先简单介绍一下statechart状态机的基本概念和涉及的相关知识，以便更好地理解Peering过程中PG的状态机转换流程。下面在举例时截取了PG状态机的部分代码。

10.1.1 状态

在statechart里，一个状态的定义方式有两种：

·没有子状态情况下的状态定义：

```
struct Reset : boost::statechart::state< Reset, RecoveryMachine >
```

这里定义了状态Reset，它需要继承boost::statechart::state类。该类的模板参数中，第一个参数为状态自己的名字Reset，第二个参数为该状态所属状态机的名字，表明Reset是状态机RecoveryMachine的一个状态。

·有子状态情况下的状态定义：

```
struct Started : boost::statechart::state< Started, RecoveryMachine, Start >
```

状态Started也是状态机RecoveryMachine的一个状态，模板参数中多一个参数Start，它是状态Started的默认初始子状态，其定义如下：

```
struct Start : boost::statechart::state< Start, Started >
```

这里定义的Start是状态Started的子状态。第一个模板参数是自己的名字，第二个模板参数是该子状态所属父状态的名字。

综上所述，一个状态，要么属于一个状态机，要么属于一个状态，成

为该状态的子状态。其定义的模板参数，第一个参数是自己，第二个参数是拥有者，第三个参数是它的起始子状态。

10.1.2 事件

状态能够接收并处理事件。事件可以改变状态，促使状态发生转移。

在boost库的statechart状态机中定义事件的方式如下所示：

```
struct QueryState : boost::statechart::event< QueryState >
```

QueryState为一个事件，需要继承boost：：statechart：：event类，模板参数为自己的名字。

10.1.3 状态响应事件

在一个状态内部，需要定义状态机处于当前状态时可以接受的事件以及如何处理这些事件的方法：

```
struct Initial : boost::statechart::state< Initial, RecoveryMachine >, NamedStat
typedef boost::mpl::list <
    boost::statechart::transition< Initialize, Reset >,
    boost::statechart::custom_reaction< Load >,
    boost::statechart::custom_reaction< NullEvt >,
    boost::statechart::transition< boost::statechart::event_base, Crashed >
> reactions;
boost::statechart::result react(const Load&);
boost::statechart::result react(const MNotifyRec&);
boost::statechart::result react(const MInfoRec&);
boost::statechart::result react(const MLogRec&);
    boost::statechart::result
        react(const boost::statechart::event_base&) {
    return discard_event();
}
}
```

上述代码列出了状态RecoveryMachine/Initial可以处理的事件列表和处理对应事件方法：

·通过boost::mpl::list定义该状态可以处理多个事件类型。在本例中可以处理Initialize、Load、NullEvt以及event_base事件。

·简单事件处理：

```
boost::statechart::transition< Initialize, Reset >
```

定义了状态Initial接收到事件Initialize后，无条件直接跳转到Reset状

态。

·用户自定义事件处理：当接收到事件后，需要根据一些条件来决定状态如何转移，这个逻辑需要用户自己定义实现：

```
boost::statechart::custom_reaction< Load >
```

custom_reaction定义了一个用户自定义的事件处理方法，必须有一个react的处理函数处理对应该事件。状态转移的逻辑需要用户自己在react函数里实现：

```
boost::statechart::result react(const Load&)
```

·NullEvt事件用户自定义处理，但是没有实现react函数来处理，最终事件匹配了boost::statechart::event_base事件，直接调用函数discard_event把事件丢弃掉。

10.1.4 状态机的定义

RecoveryMachine为定义的状态机，需要继承boost::statechart::state_machine类。

```
class RecoveryMachine : public boost::statechart::state_machine< RecoveryMachine, Initial >
```

模板参数第一个参数为自己的名字，第二个参数为状态机默认的初始状态Initial。

状态机的基本操作有两个：

- 状态机的初始化：

```
machine.initiate()
```

- 函数process_event用来向状态机投递事件，从而触发状态机接收并处理该事件：

```
machine.process_event(evt);
```

10.1.5 context函数

context是状态机的一个比较有用的函数，它可以获取当前状态的所有祖先状态的指针。通过它可以获取父状态以及祖先状态的一些内部参数和状态值。

例如状态Start是RecoveryMachine的一个状态，状态Started是Start状态的一个子状态，那么如果当前状态是Started，就可以通过该函数获取它的父状态Start的指针：

```
Start* parent = context<Start>()
```

同时也可以获取其祖先状态RecoveryMachine的指针：

```
RecoveryMachine* grandparent = context<RecoveryMachine>()
```

综上所述，context函数为获取当前状态的祖先状态上下文提供了一种方法。

10.1.6 事件的特殊处理

事件除了在状态转移列表中触发状态转移，或者进入用户自定义的状态处理函数，还可以有下列特殊的处理方式：

- 在用户自定义的函数里，可以直接调用函数transit来直接跳转到目标状态。例如：

```
transit<WaitRemoteBackfillReserved>()
```

可以直接跳转到状态WaitRemoteBackfillReserved。

- 在用户自定义的函数里，可以调用函数post_event直接产生相应的事件，并投递给状态机。

- 在用户自定义的函数里，调用函数discard_event可以直接丢弃事件，不做任何处理。

- 在用户自定义的函数里，调用函数forward_event可以把当前事件继续投递给状态机。

10.2 PG状态机

在类PG的内部定义了类RecoveryState，该类RecoveryState的内部定义了PG的状态机RecoveryMachine和它的各种状态。

在每个PG对象创建时，在构造函数里创建一个新的RecoveryState类的对象，并创建相应的RecoveryMachine类的对象，也就是创建了一个新的状态机。每个PG类对应一个独立的状态机来控制该PG的状态转换。

图10-1为PG状态机的总体状态转换图，相对比较复杂，在介绍相关的内容模块时再逐一详细介绍。

10.3 PG的创建过程

在PG的创建过程中完成了PG对应的状态机的创建和状态机的初始化操作。一个PG的创建过程会由其所在OSD在PG中担任的角色不同，创建的机制也不相同。

10.3.1 PG在主OSD上的创建

当创建一个Pool时，通过客户端命令行给Monitor发送创建Pool的命令，Monitor对该Pool的每一个PG对应的主OSD发送创建PG的请求。

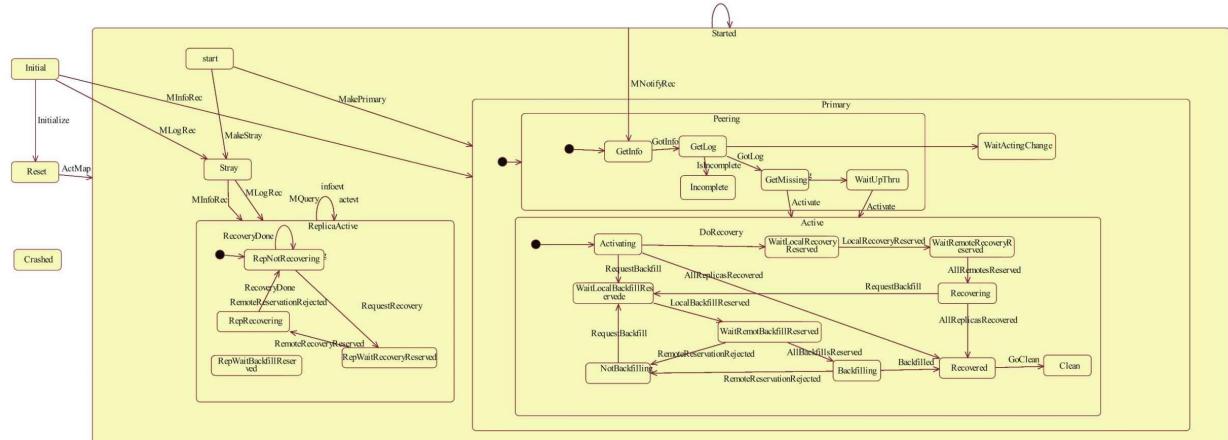


图10-1 PG状态机的总体状态转换图

函数OSD::handle_pg_create用于处理Monitor发送的创建PG的请求，其消息类型为MOSDPGCreate的数据结构：

```
struct MOSDPGCreate : public Message {
    version_t epoch;
    map<pg_t, pg_create_t> mkpg; //要创建的
    PG
    map<pg_t, utime_t> ctimes; //对应
    PG的创建时间
}
```

数据结构pg_create_t包含了一个PG创建相关的参数：

```
struct pg_create_t {
    epoch_t created; //创建了
    epoch pg
    pg_t parent; //如果
    parent不为空
    (if != pg_t()), 本
```

PG不是从
parent中分

裂出来的位

```
} __s32 split_bits;
```

函数handle_pg_create的处理过程如下：

- 1) 首先调用函数require_mon_peer确保是由Monitor发送的创建消息。
- 2) 调用函数require_same_or_newer_map检查epoch是否一致。如果对方的epoch比自己拥有的新，就更新自己的epoch；否则就直接拒绝该请求。
- 3) 对消息中mkpg列表里每一个PG，开始执行如下创建操作：
 - a) 检查该PG的参数split_bits，如果不为0，那么就是PG的分裂请求，这里不做处理；检查PG的preferred，如果设置了，就跳过，目前不支持；检查确认该pool存在；检查本OSD是该PG的主OSD；如果参数up不等于acting，说明该PG有temp_pg，至少确定该PG存在，直接跳过。
 - b) 调用函数_have_pg获取该PG对应的类。如果该PG已经存在，跳过。
 - c) 调用函数PG::: _create在本地对象存储中创建相应的collection。
 - d) 调用函数_create_lock_pg初始化PG。

- e) 调用函数pg->handle_create (&rctx) 给新创建PG状态机投递事件，
PG的状态发生相应的改变，后面会介绍。
- f) 所有修改操作都打包在事务rctx.transaction中，调用函数
dispatch_context将事务提交到本地对象存储中。

- 4) 调用函数maybe_update_heartbeat_peers来更新OSD的心跳列表。

10.3.2 PG在从OSD上的创建

Monitor并不会给PG的从OSD发送消息来创建该PG，而是由该主OSD上的PG在Peering过程中创建。主OSD给从OSD的PG状态机投递事件时，在函数handle_pg_peering_evt中，如果发现该PG不存在，才完成创建该PG。

函数handle_pg_peering_evt是处理Peering状态机事件的入口。该函数会查找相应的PG，如果该PG不存在，就创建该PG。该PG的状态机进入RecoveryMachine/Stray状态。

10.3.3 PG的加载

当OSD重启时，调用函数OSD::init()，该函数调用load_pgs函数加载已经存在的PG，其处理过程和创建PG的过程相似。

10.4 PG创建后状态机的状态转换

图10-2为PG总体状态转换图的简化版：状态Peering、Active、RelicaActive的内部状态没有添加进去。

通过该图可以了解PG的高层状态转换过程，如下所示：

- 1) 当PG创建后，同时在该类内部创建了一个属于该PG的RecoveryMachine类型的状态机，该状态机的初始化状态为默认初始化状态Initial。
- 2) 在PG创建后，调用函数pg->handle_create (&rctx) 来给状态机投递事件：

```
void PG::handle_create(RecoveryCtx *rctx){  
    dout(10) << "handle_create" << dend1;  
    Initialize evt;  
    recovery_state.handle_event(evt, rctx);  
    ActMap evt2;  
    recovery_state.handle_event(evt2, rctx);  
}
```

由以上代码可知：该函数首先向RecoveryMachine投递了Initialize类型的事件。由图10-2可知，状态机在RecoveyMachine/Initial状态接收到Initialize类型的事件后直接转移到Reset状态。其次，向RecoveryMachine投递了ActMap事件。

- 3) 状态Reset接收到ActMap事件，跳转到Started状态。

```

boost::statechart::result PG::RecoveryState::Reset::react(const ActMap&){.....
    return transit< Started >();
}

```

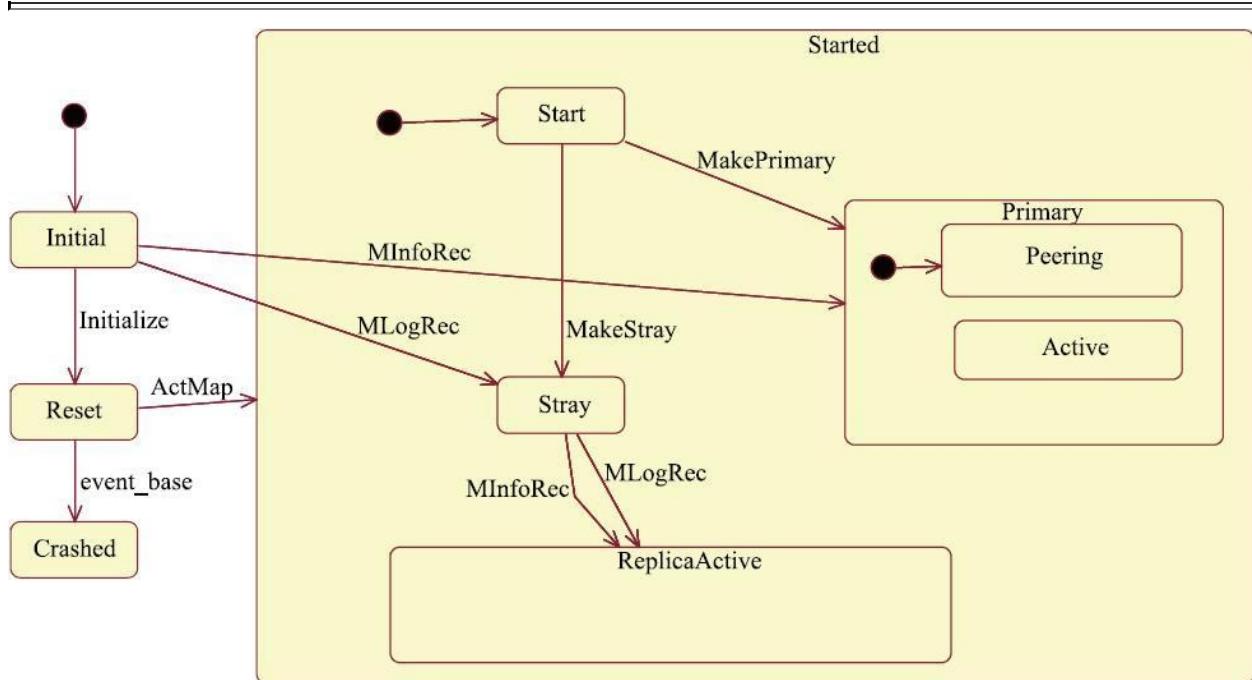


图10-2 PG总体状态图的简化版

在自定义的react函数里直接调用了transit函数跳转到Started状态。

4) 进入状态RecoveryMachine/Started后，就进入RecoveryMachine/Started的默认的子状态RecoveryMachine/Started/Start中：

```

PG::RecoveryState::Start::Start(my_context ctx)
: my_base(ctx),
  NamedState(context< RecoveryMachine >().pg->cct, "Start")
{
    context< RecoveryMachine >().log_enter(state_name);
    PG *pg = context< RecoveryMachine >().pg;
    if (pg->is_primary()) {
        dout(1) << "transitioning to Primary" << endl;
        post_event(MakePrimary());
    } else { //is_stray
        dout(1) << "transitioning to Stray" << endl;
        post_event(MakeStray());
    }
}

```

```
}
```

由以上代码可知，在Start状态的构造函数中，根据本OSD在该PG中担任的角色不同分别进行如下处理：

- 如果是主OSD，就调用函数post_event，抛出事件MakePrimary，进入主OSD的默认子状态Primary/Peering中。
- 如果是从OSD，就调用函数post_event，抛出事件MakeStray，进入Started/Stray状态。

对于一个PG的OSD处于Stray状态，是指该OSD上的PG副本目前状态不确定，但是可以响应主OSD的各种查询操作。它有两种可能：一种是最终转移到状态ReplicatActive，处于活跃状态，成为PG的一个副本。另一种可能的情况是：如果是数据迁移的源端，可能一直保持Stray状态，该OSD上的副本可能在数据迁移完成后，PG以及数据就都被删除了。

10.5 Ceph的Peering过程分析

在介绍了statechar状态机和PG的创建过程后，正式开始Peering过程介绍。Peering的过程使一个PG内的OSD达成一个一致状态。当主从副本达成一个一致的状态后，PG处于active状态，Peering过程的状态就结束了。但此时该PG的三个OSD的数据副本上的数据并非完全一致。

PG在如下两种情况下触发Peering过程。

- 当系统初始化时，OSD重新启动导致PG重新加载，或者PG新创建时，PG会发起一次Peering的过程。
- 当有OSD失效，OSD的增加或者删除等导致PG的acting set发生了变化，该PG就会重新发起一次Peering过程。

10.5.1 基本概念

1.acting set和up set

acting set是一个PG对应副本所在的OSD列表，该列表是有序的，列表中第一个OSD为主OSD。在通常情况下，up set和acting set列表完全相同。要理解它们的不同之处，需要理解下面介绍的“临时PG”概念。

2.临时PG

假设一个PG的acting set为[0, 1, 2]列表。此时如果osd0出现故障，导致CRUSH算法重新分配该PG的acting set为[3, 1, 2]。此时osd3为该PG的主OSD，但是osd3为新加入的OSD，并不能负担该PG上的读操作。所以PG向Monitor申请一个临时的PG，osd1为临时的主OSD，这时up set变为[1, 3, 2]，acting set依然为[3, 1, 2]，导致acting set和up set不同。当osd3完成Backfill过程之后，临时PG被取消，该PG的up set修复为acting set，此时acting set和up set都为[3, 1, 2]列表。

3.权威日志

权威日志（在代码里一般简写为olog）是一个PG的完整顺序且连续操作的日志记录。该日志将作为数据修复的依据。

4.up_thru

引入up_thru的概念是为了解决特殊情况：当两个以上的OSD处于down状态，但是Monitor在两次epoch中检测到了这种状态，从而导致Monitor认为它们是先后宕掉。后宕的OSD有可能产生数据的更新，导致需要等待该OSD的修复，否则有可能产生数据丢失。

例10-1 up_thru处理过程

下图为初始情况：

epoch	处于 up 的 OSD	
1	A	B
2		B
3		
4	A	

过程如下所示：

1) 在epoch1时，一个PG中有A、B两个OSD（两个副本）都处于up的状态。

2) 在epoch2时，Monitor检测到了A处于down状态，B仍然处于up状态。由于Monitor的检测可能滞后，实际可能有两种情况：

情况1：此时B其实也已经和A同时宕了，只是Monitor没有检测到。此时PG不可能完成PG的Peering过程，PG没有新数据写入。

情况2：此时B确实处于up状态，由于B上保持了完整的数据，PG可以

完成Peering过程并处于active的状态，可以接受新的数据写操作。

上述两种不同的情况，Monitor无法区分。

3) 在epoch3时，Monitor检测到B也宕了。

4) 在epoch4时，A恢复了up的状态后，该PG发起Peering过程，该PG是否允许完成Peering过程处于active状态，可以接受读写操作？

·如果在epoch2时，属于情况1：PG并没有数据更新，B上不会新写入数据，A上的数据保存完整，此时PG可以完成Peering过程从而处于active状态，接受写操作。

·如果在epoch2时，属于情况2：PG上有新数据更新到了osd B，此时osd A缺失一些数据，该PG不能完成Peering过程。

为了使Monitor能够区分上述两种情况，引入了up_thru的概念，up_thru记录了每个OSD完成Peering后的epoch值。其初始值设置为0。

在上述情况2，PG如果可以恢复为active状态，在Peering过程，须向Monitor发送消息，Monitor用数组up_thru[osd]来记录该OSD完成Peering后的epoch值。

当引入up_thru后，上述例子的处理过程如下：

epoch	处于 up 的 OSD		monitor up_thru
1	A	B	
2		B	up_thru[B]=0
3			
4	A		

情况1的处理流程如下：

- 1) 在epoch1时， up_thru[B]为0， 也就是说B在epoch为0时参与完成Peering。
- 2) 在epoch2时， Monitor检查到OSD A处于down状态， OSD B仍处于up状态（实际B已经处于down状态）， PG没有完成Peering过程， 不会向Monitor上报更新up_thru的值。
- 3) epoch3时， A和B两个OSD都宕了。
- 4) epoch4时， A恢复up状态， PG开始Peering过程， 发现up_thru[B]为0， 说明在epoch为2时没有更新操作， 该PG可以完成Peering过程， PG处于active状态。

情况2的处理如下所示：

epoch	处于 up 的 OSD		monitor up_thru
1	A	B	
2		B	up_thru[B]=0
3		B	up_thru[B]=2
4			
5	A		

情况2的处理流程如下：

- 1) 在epoch1时, up_thru[B]为0, 也就是说B在epoch为0时参与完成Peering过程。
- 2) 在epoch2时, Monitor检查到OSD A处于down状态, OSD B还处于up状态, 该PG完成了Peering过程, 向Monitor上报B的up_thru变为当前epoch的值为2, 此时PG可接受写操作请求。
- 3) 在epoch4时, A和B都宕了, B的up_thru为2。
- 4) 在epoch5时, A处于up状态, 开始Peering过程, 发现up_thru[B]为2, 说明在epoch为2时完成了Peering, 有可能有更新操作, 该PG需要等待B恢复。否则可能丢失B上更新的数据。

10.5.2 PG日志

PG日志（pg log）为一个PG内所有更新操作的记录（下文所指的日志，如不特别指出，都是指PG日志）。每个PG对应一个PG日志，它持久化保存在每个PG对应pgmeta_oid对象的omap属性中。

它有如下的特点：

- 记录一个PG内所有对象的更新操作元数据信息，并不记录操作的数据。
- 是一个完整的日志记录，版本号是顺序的且连续的。

1.pg_log_t

结构体pg_log_t在内存中保存了该PG的所有操作日志，以及相关的控制结构。

```
struct pg_log_t {
    eversion_t head;      //日志的头，记录最新的日志记录

    eversion_t tail;      //日志的尾，记录最旧的日志记录

    eversion_t can_rollback_to; //用于
                                EC，指示本地可以回滚的版本，可回滚的版本都大于版本
                                can_rollback_to的值

    eversion_t rollback_info_trimmed_to;
                //在
```

EC的实现中，本地保留了不同版本的数据。本数据段指示本

PG里可以删除掉的对象版本

```
list<pg_log_entry_t> log; //所有日志的列表  
.....  
}
```

需要注意的是， PG日志的记录是以整个PG为单位， 包括该PG内所有对象的修改记录。

2.pg_log_entry_t

结构体pg_log_entry_t记录了PG日志的单条记录， 其数据结构如下：

```
struct pg_log_entry_t {  
    __s32 op; //操作的类型  
  
    hobject_t soid; //操作的对象  
  
    eversion_t version; //本次操作的版本  
  
    prior_version; //前一个操作的版本  
  
    reverting_to; //本次操作回退的版本 (仅用于回滚操作)  
  
    ObjectModDesc mod_desc; //用于保存本地回滚的一些信息， 用于  
    EC模式下的回滚操作  
  
    bufferlist snaps; //克隆操作用于记录当前对象的  
    snap列表  
  
    osd_reqid_t reqid; //请求唯一标识 (   
    called + tid)
```

```
vector<pair<osd_reqid_t, version_t>> extra_reqids;
version_t user_version; //用户的版本

utime_t      mtime;      //这是用户本地时间

.....
}
```

3.IndexedLog

类IndexedLog继承了类pg_log_t，在其基础上添加了根据一个对象来检索日志的功能，以及其他相关的功能。

4.日志的写入

函数PG::add_log_entry添加pg_log_entry_t条目到PG日志中。同时更新了info.last_complete和info.last_update字段。

PGLog::write_log函数将日志写到对应的pgmeta_oid对象的kv存储中。在这里并没有直接写入磁盘，而是先把日志的修改添加到ObjectStore::Transaction类型的事务中，与数据操作组成一个事务整体提交磁盘。这样可以保证数据操作、日志更新及其pg info信息的更新都在一个事务中，都以原子方式提交到磁盘上。

5.日志的trim操作

函数trim用来删除不需要的旧日志。当日志的条目数大于min_log_entries时，需要进行trim操作。

```
void PGLog::trim(LogEntryHandler *handler,
eversion_t trim_to, pg_info_t &info)
```

6. 合并权威日志

函数merge_log用于把本地日志和权威日志合并：

```
void PGLog::merge_log(ObjectStore::Transaction& t,
                      pg_info_t &oinfo, pg_log_t &olog,
                      pg_shard_t fromosd,
                      pg_info_t &info, LogEntryHandler *rollbacker,
                      bool &dirty_info, bool &dirty_big_info)
```

其处理过程如下：

1) 本地日志和权威日志没有重叠的部分：在这种情况下就无法依据日志来修复，只能通过Backfill过程来完成修复。所以先确保权威日志和本地日志有重叠的部分：

```
assert(log.head >= olog.tail && olog.head >= log.tail);
```

2) 本地日志和权威日志有重叠部分的处理：

- 如果olog.tail小于log.tail，也就是权威日志的尾部比本地日志长。在这种情况下，只要把日志多出的部分添加到本地日志即可，它不影响missing对象集合。

- 本地日志的头部比权威日志的头部长，说明有多出来的divergent日志，调用函数rewind_divergent_log去处理。

·本地日志的头部比权威日志的头部短，说明有缺失的日志，其处理过程为：把缺失的日志添加到本地日志中，记录missing的对象，并删除多出来的日志记录。

下面举例说明函数merge_log的不同处理情况。

例10-2 函数merge_log应用举例

情况1：权威日志的尾部版本比本地日志的尾部小，如下所示：

log (本地日志)			obj10(1,6) modify log_tail	obj11(1,7) modify	obj13(1,8) modify log_head
olog (权威日志)	obj3(1,4) modify log_tail	obj4(1,5) modify	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head

本地log的log_tail为obj10 (1, 6)，权威日志olog的log_tail为obj3 (1, 4)。

日志合并的处理方式如下所示：

log (本地日志)	obj3 (1,4) modify log_tail	obj4(1,5) modify	obj10(1,6) modify log_tail	obj11(1,7) modify	obj13(1,8) modify
olog (权威日志)	obj3 (1,4) modify log_tail	obj4(1,5) modify	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify

把日志记录obj3 (1, 4)、obj4 (1, 5)添加到本地日志中，修改info.log_tail和log.tail指针即可。

情况2：本地日志的头部版本比权威日志长，如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head			
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head

权威日志的log_head为obj13 (1, 8) , 而本地日志的log_head为obj10 (1, 11) 。本地日志的log_head版本大于权威日志的log_head版本, 调用函数rewind_divergent_log来处理本地有分歧的日志。

在本例的具体处理过程为：把对象obj10、obj11、obj13加入missing列表中用于修复。最后删除多余的日志，如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head			
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
missing object: obj10 (1,6) obj11(1,7) obj13(1,8)						

本例比较简单，函数rewind_divergent_log会处理比较复杂的一些情况，后面会介绍到。

情况3：本地日志的头部版本比权威日志的头部短，如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head			

权威日志的log_head为obj10 (1, 11) , 而本地日志的log_head为

obj13 (1, 8) , 即本地日志的log_head版本小于权威日志的log_head版本。

其处理方式如下：把本地日志缺失的日志添加到本地，并计算本地缺失的对象。最后把缺失的对象添加到missing object列表中用于后续的修复，处理结果如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
missing object		obj13(1,9) obj11(1,10)				

7.处理副本日志

函数proc_replica_log用于处理其他副本节点发过来的和权威日志有分叉（divergent）的日志。其关键在于计算missing的对象列表，也就是需要修复的对象，如下所示：

```
void PGLog::proc_replica_log(ObjectStore::Transaction& t,
                           pg_info_t &oinfo, const pg_log_t &olog,
                           pg_missing_t& omissing,
                           pg_shard_t from) const
```

其参数都为远程节点的信息：

·oinfo：远程节点的pg_info_t。

·olog：远程节点的日志。

·omissing：远程节点的missing信息，为输出信息。

·from：来自远程节点。

函数proc_replica_log的具体处理过程如下：

- 1) 如果日志不重叠，就无法通过日志来修复，需要进行Backfill过程，直接返回。
- 2) 如果日志的head相同，说明没有分歧日志（divergent log），直接返回。
- 3) 下面处理的都是这种情况：日志有重叠并且日志的head不相同，需要处理分歧的日志：
 - 计算第一条分歧日志first_non_divergent，从本地日志后往前查找小于等于olog.head的日志记录。
 - 版本lu为分歧日志的边界。如果first_non_divergent没有找到，或者小于权威日志的log_tail，那么lu就设置为log_tail，否则就设置为first_non_divergent日志记录的版本。
 - 把所有的分歧日志都添加到divergent队列里。
 - 构建一个IndexedLog的对象folog，把所有没有分歧的日志添加到folog里。

- 调用函数`_merge_divergent_entries`处理分歧日志。
- 更新`oinfo`的`last_update`为`lu`版本。
- 如果有对象`missing`, 就设置`last_complete`为小于`first_missing`的版本。

函数`_merge_divergent_entries`处理所有的分歧日志, 首先把所有分歧日志的对象按照对象分类, 然后分别调用函数`_merge_object_divergent_entries`对每个分歧日志的对象进行处理。

函数`_merge_object_divergent_entries`用于处理单个对象的

- 1) 首先进行比较, 如果处理的对象`hoid`大于`info.last_backfill`, 说明该对象本来就不存在, 没有必要修复。



注意

这种情况一般发生在如下情景：该PG在上一次Peering操作成功后, PG还没有处理clean状态, 正在Backfill过程中, 就再次触发了Peering的过程。`info.last_backfill`为上次最后一个修复的对象。

在本PG完成Peering后就开始修复, 先完成Recovery操作, 然后会继续完成上次的Backfill操作, 所以没有必要在这里检查来修复。

- 2) 通过该对象的日志记录来检查版本是否一致。首先确保是同一个对

象，本次日志记录的版本prior_version等于上一条日志记录的version值。

3) 版本first_divergent_update为该对象的日志记录中第一个产生分歧的版本；版本last_divergent_update为最后一个产生分歧的版本；版本prior_version为第一个分歧产生的前一个版本，也就是应该存在的对象版本。布尔变量object_not_in_store用来标记该对象不缺失，且第一条分歧日志操作是删除操作。处理分歧日志的五种情况如下所示：

情况1：在没有分歧的日志里查找到该对象，但是已存在的对象的版本大于第一个分歧对象的版本。这种情况的出现，是由于在merge_log中产生权威日志时的日志更新，相应的处理已经做了，这里不做任何处理。

情况2：如果prior_version为eversion_t()，为对象的create操作或者是clone操作，那么这个对象就不需要修复。如果已经在missing记录中，就删除该missing记录。

情况3：如果该对象已经处于missing列表中，如下进行处理：

- 如果日志记录显示当前已经拥有的该对象版本have等于prior_version，说明对象不缺失，不需要修复，删除missing中的记录。
- 否则，修改需要修复的版本need为prior_version；如果prior_version小于等于info.log_tail时，这是不合理的，设置new_divergent_prior用于后续处理。

情况4：如果该对象的所有版本都可以回滚，直接通过本地回滚操作

就可以修复，不需要加入missing列表来修复。

情况5：如果不是所有的对象版本都可以回滚，删除相关的版本，把prior_version加入missing记录中用于修复。

10.5.3 Peering的状态转换图

由10.4节分析可知，主OSD上PG对应的状态机RecoveryMachine目前已经处于Started/Primary/Peering状态。从OSD上的PG对应的RecoveryMachine处于Started/Stray状态。本节总体介绍Peering过程的状态转换。

如图10-3所示为Peering状态转换图，其过程如下：

- 1) 当进入Primary/Peering状态后，就进入默认子状态GetInfo中。
- 2) 状态GetInfo接收事件GotInfo后，转移到GetLog状态中。
- 3) 如果状态GetLog接收到IsIncomplete事件后，跳转到Incomplete状态。
- 4) 状态GetLog接收到事件GotLog后，就转入GetMissing状态。
- 5) 状态GetMissing接收到事件Activate事件，转入状态active状态。

由上述Peering的状态转换过程可知，Peering过程基本分为如下三个步骤：

步骤1：GetInfo：PG的主OSD通过发送消息获取所有从OSD的pg_info信息。

步骤2：GetLog：根据各个副本获取的pg_info信息的比较，选择一个

拥有权威日志的OSD（auth_log_shard）。如果主OSD不是拥有权威日志的OSD，就从该OSD上拉取权威日志。主OSD完成拉取权威日志后也就拥有了权威日志。

步骤3：GetMissing：主OSD拉取其他从OSD的PG日志（或者部分获取，或者全部获取FULL_LOG）。通过与本地权威日志的对比，来计算该OSD上缺失的object信息，作为后续Recovery操作过程的依据。

最后通过Active操作激活主OSD，并发送notify通知消息，激活相应的从OSD。

下面介绍这三个主要步骤。

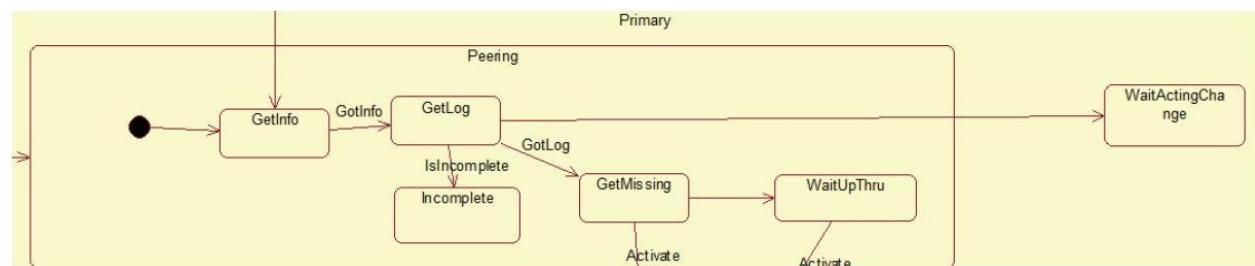


图10-3 Peering状态转换图

10.5.4 pg_info数据结构

数据结构pg_info_t保存了PG在OSD上的一些描述信息。该数据结构在Peering的整个过程，以及后续的数据修复中都发挥了重要作用，理解该数据结构的各个关节字段的含义可以更好地理解相关的过程。pg_info_t数据结构如下：

```
struct pg_info_t{
    spg_t pgid;                                //PG的
    id
    eversion_t last_update;                     //PG最后一次更新的版本

    eversion_t last_complete;
    epoch_t last_epoch_started;
    version_t last_user_version;               //最后更新的用户版本号，用于分层存。

    eversion_t log_tail;                        //日志的尾部版本

    hobject_t last_backfill;                   //上一次
    Backfill操作的对象指针。如果该
    OSD的
    Backfill操作没有完成，那么
    last_backfill 和
    last_complete之间的对象就丢失。

    interval_set<snapid_t> purged_snaps;   //PG要删除的
    snap集合

    pg_stat_t stats;                          //PG的统计信息

    pg_history_t history;                    //PG的历史信息
```

```
pg_hit_set_history_t hit_set;           //这是  
Cache Tier用的  
hit_set  
}
```

结构pg_history_t保存了PG的一些历史信息：

```
struct pg_history_t {  
epoch_t epoch_created;           //PG创建的  
epoch值  
  
epoch_t last_epoch_started;  
epoch_t last_epoch_clean;        //PG处于  
clean状态时的  
epoch值  
  
.....  
}
```

1.last_epoch_started介绍

last_epoch_started字段有两个地方出现，一个是pg_info结构里的last_epoch_started，代表最后一次Peering成功后的epoch值，是本地PG完成Peering后就设置的。另一个是pg_history_t结构里的last_epoch_started，是PG里所有的OSD都完成Peering后设置的epoch值。

2.last_complete和last_backfill的区别

在这里特别指出last_update和last_complete、last_backfill之间的区别。下面通过一个例子来讲解，同时也大概了解PG数据恢复的流程。在数

据恢复过程中先进行Recovery过程，再进行Backfill过程（可以参考第11章的详细介绍）。

情况1：在PG处于clean状态时，last_complete就等于last_update的值，并且等于PG日志中的head版本。它们都同步更新，此时没有区别。last_bacfill设置为MAX值。例如：下面的PG日志里有三条日志记录。此时last_update和last_complete以及pg_log.head都指向版本（1, 2）。由于没有缺失的对象，不需要恢复，last_backfill设置为MAX值。示例如下所示：

obj1 (1,0)	obj2 (1,1)	obj3 (1,2) last_update last_complete pg_log.head		
--------------	--------------	---	--	--

情况2：当该osd1发生异常之后，过一段时间后又重新恢复，当完成了Peering状态后的情况。此时该PG可以继续接受更新操作。例如：下面的灰色字体的日志记录为该osd1崩溃期间缺失的日志，obj7为新的写入的操作日志记录。last_update指向最新的更新版本（1, 7），last_complete依然指向版本（1, 2）。即last_update指的是最新的版本，last_complete指的是上次的更新版本。过程如下：

obj1 (1,0)	obj2 (1,1)	obj3 (1,2) last_complete	obj4(1,3)	obj3(1,4)	obj5(1,5)	obj6(1,6)	obj7(1,7) last_update pg_log.head
--------------	--------------	-------------------------------	-----------	-----------	-----------	-----------	---

last_complete为Recovery修复进程完成的指针。当该PG开始进行Recovery工作时，last_complete指针随着Recovery过程推进，它指向完成修复的版本。例如：当Recovery完成后last_complete指针指向最后一个修复的

对象的版本 (1, 6) , 如下所示 :

obj1 (1,0)	obj2 (1,1)	obj3 (1,2)	obj4(1,3)	obj3(1,4)	obj5(1,5)	obj6(1,6) last_complete	obj7(1,7) last_update pg_log.head
--------------	--------------	--------------	-----------	-----------	-----------	----------------------------	---

[last_backfill](#)为Backfill修复进程的指针。在Ceph Peering的过程中，该PG有osd2无法根据PG日志来恢复，就需要进行Backfill过程。last_backfill初始化为MIN对象，用来记录Backfill的修复进程中已修复的对象。例如：进行Backfill操作时，扫描本地对象（按照对象的hash值排序）。

last_backfill随修复的过程中不断推进。如果对象小于等于last_backfill，就是已经修复完成的对象。如果对象大于last_backfill且对象的版本小于last_complete，就是处于缺失还没有修复的对象。过程如下所示：

Backfill 对象的列表	MIN	obj1 (1,0)	obj2 (1,1)	obj3 (1,4)	obj4 (1,3)	obj5 (1,5)	obj6 (1,6)
	last_backfill						last_complete

当恢复完成之后，last_backfill设置为MAX值，表明恢复完成，设置last_complete等于last_update的值。

10.5.5 GetInfo

GetInfo过程获取该PG在其他OSD上的结构图pg_info_t信息（也称pg_info信息）。这里的其他OSD包括当前PG的活跃OSD，以及past interval期间该PG所有处于up状态的OSD。

由10.5.4节的介绍可知，当PG进入Primary/Peering状态后，就进入默认的子状态GetInfo里。其主要流程在构造函数里完成：

```
PG::RecoveryState::GetInfo::GetInfo(my_context ctx)
    : my_base(ctx),
    NamedState(context< RecoveryMachine >().pg->cct, "Started/Primary/Peering/GetI
{
    ...
}

}
```

在构造函数GetInfo里，完成了核心的功能，实现过程如下：

1) 调用函数generate_past_intervals计算past intervals的值：

```
pg->generate_past_intervals();
```

2) 调用函数build_prior构造获取pg_info_t信息的OSD列表：

```
pg->build_prior(prior_set);
```

3) 调用函数get_infos给参与的OSD发送获取请求：

```
get_infos();
```

由上述可知，GetInfo过程基本分三个步骤：计算past_interval的过程；通过调用函数build_prior来计算要获取pg_info信息的OSD列表；最后调用函数get_infos给相关的OSD发送消息来获取pg_info信息，并处理接收到的Ack应答。

1. 计算past_interval

函数past_interval是epoch的一个序列。在该序列内一个PG的acting set和up set不会变化。current_inteval是一个特殊的past_interval，它是当前最新的一个没有变化的序列。示例如下：

说明如下：

- 1) Ceph系统当前的epoch值为20，PG1.0的acting set和up set都为列表[0, 1, 2]。
- 2) osd3失效导致了osd map变化，epoch变为21。
- 3) osd5失效导致了osd map变化，epoch变为22。
- 4) osd6失效导致了osd map变化，epoch变为23。

上述三次epoch的变化都不会改变PG1.0的acting set和up set。

epoch	20	21	22	23	24	25	26
失效 OSD		osd3	osd5	osd6	osd2	osd12	osd13
PG 1.0	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,8]	[0,1,8]	[0,1,8]
	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,8]	[0,1,8]	[0,1,8]
					last_epoch_started	last_epoch_clean	
	past_interval				current_interval		

5) osd2失效导致了osd map变化， epoch变为24；此时导致PG1.0的acting set和up set变为[0, 1, 8]，若此时Peering过程成功完成，则last_epoch_started为24。

6) osd12失效导致了osd map变化， epoch变为25，此时如果PG1.0完成了Recovery操作，处于clean状态，last_epoch_clean就为25。

7) osd13失效导致了osd map变化， epoch变为26。

epoch序列[20, 21, 22, 23]就为PG1.0的一个past interval，epoch序列[24, 25, 26]就为PG1.0的current interval。

数据结构pg_interval_t用于保存past_interval的信息：

```

struct pg_interval_t {
    vector<int32_t> up, acting; //在本
interval阶段
PG处于
up和
acting状态的
OSD
    epoch_t first, last;           //起始和结束的
epoch
    bool maybe_went_rw;           //在这个阶段是否有数据读写操作

```

```
int32_t primary; //主  
OSD  
int32_t up_primary; //up状态的主  
OSD  
};
```

上例中， past_interval对象的p值为：

```
p = {  
    up = [0,1,2],  
    acting = [0,1,2],  
    first = 20,  
    last = 23,  
  
    maybe_went_rw = true,  
  
    primary = 0,  
    up_primay = 0,  
}
```

函数generate_past_intervals用于计算past_intervals的值，计算的结果保
存在PG中past_intervals的map结构里， map的key值为first epoch的值：

```
map<epoch_t, pg_interval_t> past_intervals;
```

具体计算过程如下：

- 1) 调用函数_calc_past_interval_range推测需要计算的past_interval的起
始epoch值 (start) 和结束epoch值 (end) 。如果返回false，说明不需要计
算past_interval，所有的past_interval已经计算好了。

2) 从start到end开始计算past_interval。过程为调用函数

check_new_interval比较两次epoch对应的osd map的变化。如果检查是一个新值，就创建一个新的past_interval对象。

```
bool PG::_calc_past_interval_range(epoch_t *start, epoch_t *end, epoch_t oldest_
```

函数_calc_past_interval_range用于计算past_interval的范围。参数
oldest_map为OSD的superblock里保存的最老osd map，输出为start和end，
分别为需要计算的past_interval的start和end值。具体实现过程如下。

计算end值如下所示：

1) 变量end为当前osd map的epoch值，如果
info.history.same_interval_since不为空，就设置为该值。该值表示和当前的
osd map的epoch值在同一个interval中。

```
if (info.history.same_interval_since) {  
    *end = info.history.same_interval_since;  
} else {  
    //当前
```

PG可能是新引入的，计算整个

range期间

```
interval  
*end = osdmap_ref->get_epoch();  
}
```

2) 查看past_intervals里已经计算的past_interval的第一个epoch，如果
已经比info.history.last_epoch_clean小，就不用计算了，直接返回false值。

否则设置end为其first值。

```
*end = past_intervals.begin()->first;
```

计算start值如下所示：

- 1) start设置为info.history.last_epoch_clean，从最后一次last_epoch_clean算起。
- 2) 当PG为新建时，从info.history.epoch_created开始计算。
- 3) oldest_map值为保存的最早osd map的值，如果start小于这个值，相关的osd map信息缺失，所以无法计算。

所以将start设置为三者的最大值：

```
*start = MAX(MAX(info.history.epoch_created,  
info.history.last_epoch_clean),  
oldest_map);
```

下面举例说明计算past_interval的过程。

例10-3 past_interval计算示例

4	5	6	7	8	9	10	11	12	13	14	15	16
past_interval 1				past_interval 2			past_interval 3			current_interval		

如上表所示：一个PG有4个interval。past_interval 1，开始epoch为4，结束的epoch为8；past_interval 2的epoch区间为(9, 11)；past_interval 3

的区间为 (12, 13) ; current_interval的区间为 (14, 16) 。最新的epoch为16, info.history.same_interval_since为14, 意指是从epoch14开始, 之后的epoch值和当前的epoch值在同一个interval内。info.history.last_epoch_clean为8, 就是说在epoch值为8时, 该PG处于clean状态。

计算start和end的方法如下：

1) start的值设置为info.history.last_epoch_clean值, 其值为8。

2) end值从14开始计算, 检查当前已经计算好的past_intervals的值。

past_interval的计算是从后往前计算。如果第一个past interval的first小于等于8, 也就是past_interval 1已经计算过了, 那么后面的past_interval 2和past_interval 3都已经计算过, 就直接退出。否则就继续查找没有计算过的past_interval的值。

2.构建OSD列表

函数build_prior根据past_intervals来计算probe_targets列表, 也就是要去获取pg_info的OSD列表。具体实现为：首先重新构造一个PriorSet对象，在PriorSet的构造函数中完成下列操作：

1) 把当前PG的acting set和up set中的OSD加入到probe列表中。

2) 检查每个past_intervals阶段：

a) 如果interval.last小于info.history.last_epoch_started, 这种情况下

`past_interval`就没有意义，直接跳过。

b) 如果该interval的act为空，就跳过。

c) 如果该interval没有rw操作，就跳过。

d) 对于当前interval的每一个处于acting状态的OSD进行检查：

·如果该OSD当前处于up状态，就加入到`up_now`列表中。同时加入到`probe`列表中，用于获取权威日志以及后续数据恢复。

·如果该OSD当前不是up状态，但是在该`past_interval`期间还处于up状态，就加入`up_now`列表中。

·否则加入`down`列表，该列表保存所有宕了的OSD。

·如果当前interval确实有宕的OSD，就调用函数`pcontdec`，也就是PG的`IsPGRecoverablePredicate`函数。该函数判断该PG在该interval期间是否可恢复。如果无法恢复，直接设置`pg_down`为true值。



注意

这里特别强调的是，要确保每个interval期间都可以进行修复。函数`IsPGRecoverable-Predicate`实际上是一个类的运算符重载。对于不同类型的PG有不同的实现。对于ReplicatedPG对应的实现类为`RPCRecPred`，其至少保证有一个处于up状态的OSD；对应ErasureCode (n+m) 类型的PG，至少

有n个处于up状态的OSD。

- 3) 如果prior.pg_down设置为true, 就直接设置PG为PG_STATE_DOWN状态。
- 4) 检查是否需要need_up_thru设置。
- 5) 用prior_set->probe设置probe_targets列表。

3. 获取pg_info信息

在上述过程中计算出了PG在past interval期间以及当前处于up状态的OSD列表, 下面就发送请求给OSD来获取pg_info信息：

```
void PG::RecoveryState::GetInfo::get_infos()
```

函数get_infos向prior_set的probe集合中的每个OSD发送pg_query_t::INFO消息, 来获取PG在该OSD上的pg_info信息。发送消息的过程调用RecoveryMachine类的send_query函数来进行：

```
context< RecoveryMachine >().send_query(  
    peer,  
    pg_query_t(pg_query_t::INFO,  
              it->shard, pg->pg_whoami.shard,  
              pg->info.history,  
              pg->get_osdmap()->get_epoch())  
    );  
    peer_info_requested.insert(peer);  
    pg->blocked_by.insert(peer.osd)
```

数据结构pg_notify_t定义了获取pg_info的ACK信息：

```
struct pg_notify_t {
    epoch_t query_epoch;      //查询时请求消息的
    epoch_t epoch_sent;        //发送时应对消息的
    epoch_t info;              // pg_info的信息
    shard_id_t to;             //目标
    OSD shard_id_t from;       //源
};
```

在主OSD收到pg_info的ACK消息后封装成MNotifyRec事件发送给该PG对应的状态机。在下列的事件处理函数中来处理MNotifyRec事件。

```
boost::statechart::result PG::RecoveryState::GetInfo::react(const MNotifyRec& in
```

具体处理过程如下：

- 1) 首先从peer_info_requested里删除该peer，同时从blocked_by队列里删除。
- 2) 调用函数PG::proc_replica_info来处理副本的pg_info消息：
 - a) 首先检查如果该OSD的pg_info信息，如果已经存在，并且last_update参数相同，则说明已经处理过，返回false值。否则保存该pg_info的值。
 - b) 调用函数has_been_up_since检查该OSD在send_epoch时已经处于up

状态。

- c) 确保自己是主OSD， 把该OSD的pg_info信息保存到peer_info数组，并加入might_have_unfound数组里。该数组里的OSD用于后续的数据恢复。
 - d) 调用函数unreg_next_scrub使该PG不在scrub操作的队列中。
 - e) 调用info.history.merge函数处理从OSD发过来的pg_info信息。处理方法是：更新为最新的字段， 设置dirty_info为true值。
 - f) 调用函数reg_next_scrub注册PG下一次的scrub的时间。
 - g) 如果该OSD既不在up数组中也不在acting数组中， 那就加入stray_set列表中。当PG处于clean状态时， 就会调用purge_strays函数删除stray状态的PG及其上的对象数据。
 - h) 如果是一个新的OSD， 就调用函数update_heartbeat_peers更新需要heartbeat的OSD列表。
- 3) 在变量old_start里保存了调用proc_replica_info前主OSD的pg->info.history.last_epoch_started， 如果该epoch值小于合并后的值， 说明该值被更新了， 从OSD上的epoch值比较新， 需要进行如下操作：
- a) 调用pg->build_prior重新构建prior_set对象。
 - b) 从peer_info_requested队列中去掉上次构建的prior_set中存在的

OSD，这里最新构建上次不存在的OSD列表。

- c) 调用get_infos函数重新发送查询peer_info请求。
- 4) 调用pg->apply_peer_features更新相关的features值。
- 5) 当peer_info_requested队列为空，并且prior_set不处于pg_down的状态时，说明收到所有OSD的peer_info并处理完成。
- 6) 最后检查past_interval阶段至少有一个OSD处于up状态且非incomplete状态；否则该PG无法恢复，标记状态为PG_STATE_DOWN并直接返回。
- 7) 最后完成处理，调用函数post_event (GotInfo ()) 抛出GetInfo事件进入状态机的下一个状态。

在GetInfo状态里直接定义了当前状态接收到GotInfo事件后，直接跳转到下一个状态GetLog里：

```
struct GetInfo : boost::statechart::state< GetInfo, Peering >, NamedState {
    typedef boost::mpl::list <
        boost::statechart::custom_reaction< QueryState >,
        boost::statechart::transition< GotInfo, GetLog >,
        boost::statechart::custom_reaction< MNotifyRec >
    > reactions;
}
```

10.5.6 GetLog

当PG的主OSD获取到所有从OSD（以及past interval期间的所有参与该PG且目前仍处于active状态的OSD）的pg_info信息后，就跳转到GetLog状态。

```
PG::RecoveryState::GetLog::GetLog(my_context ctx)
```

然后在GetLog的构造函数里做相应的处理，其具体处理过程分析如下：

- 1) 调用函数pg->choose_acting (auth_log_shard) 选出具有权威日志的OSD，并计算出acting_backfill和backfill_targets两个OSD列表。输出保存在auth_log_shard里。
- 2) 如果选择失败并且want_acting不为空，就抛出NeedActingChange事件，状态机转移到Primary/WaitActingChang状态，等待申请临时PG返回结果。如果want_acting为空，就抛出IsIncomplete事件，PG的状态机转移到Primay/Peering/Incomplete状态。表明失败，PG就处于Incomplete状态。
- 3) 如果auth_log_shard等于pg->pg_whoami的值，也就是选出的拥有权威日志的OSD为当前主OSD，直接抛出事件GotLog () 完成GetLog过程。
- 4) 如果pg->info.last_update小于权威OSD的log_tail，也就是本OSD的

日志和权威日志不重叠，那么本OSD无法恢复，抛出IsIncomplete事件。经过函数choose_acting的选择后，主OSD必须是可恢复的。如果主OSD不可恢复，必须申请一个临时PG，选择拥有权威日志的OSD为临时主OSD。

5) 如果自己不是权威日志的OSD，则需要去拥有权威日志的OSD上去拉取权威日志，并与本地合并。

[1.choose_acting](#)

函数choose_acting用来计算PG的acting_backfill和backfill_targets两个OSD列表。acting_backfill保存了当前PG的acting列表，包括需要进行Backfill操作的OSD列表；backfill_targets列表保存了需要进行Backfill的OSD列表。其处理过程如下：

- 1) 首先调用函数find_best_info来选举出一个拥有权威日志的OSD，保存在变量auth_log_shard里。
- 2) 如果没有选举出拥有权威日志的OSD，则进入如下流程：
 - a) 如果up不等于acting，申请临时PG，返回false值。
 - b) 否则确保want_acting列表为空，返回false值。
- 3) 计算是否是compat_mode模式，检查是，如果所有的OSD都支持纠删码，就设置compat_mode值为true。
- 4) 根据PG的不同类型，调用不同的函数，对应ReplicatedPG调用函数

calc_replicated_acting来计算PG的需要列表：

```
set<pg_shard_t> want_backfill, want_acting_backfill;
//want_backfill为该
PG需要进行
Backfill的
pg_shard
//want_acting_backfill包括进行
acting和
Backfill的
pg_shard
pg_shard_t want_primary; //主
OSD
vector<int> want;           //在
compat_mode模式下，和
want_acting_backfill相同
```

5) 下面就是对PG做的一些检查：

a) 计算num_want_acting数量， 检查如果小于min_size， 进行如下操作：

- 如果对于EC， 清空want_acting， 返回false值。
- 对于ReplicatePG， 如果该PG不允许小于min_size的恢复， 清空want_acting， 返回false值。

b) 调用IsPGRecoverablePredicate来判断PG现有的OSD列表是否可以恢复， 如果不能恢复， 清空want_acting， 返回false值。

- 6) 检查如果want不等于acting， 设置want_acting为want：
 - a) 如果wang_acting等于up， 申请empty为pg_temp的OSD列表。
 - b) 否则申请want为pg_temp的OSD列表。
 - 7) 最后设置PG的actingbackfill为want_acting_backfill， 设置backfill_targets为want_backfill，并检查backfill_targets里的pg_shard应该不在stray_set里面。
 - 8) 最终返回true值。
- 下面举例说明需要申请pg_temp的场景：
- 1) 当前PG1.0， 其acting列表和up列表都为[0, 1, 2]， PG处于clean状态。
 - 2) 此时， osd0崩溃， 导致该PG经过CRUSH算法重新获得acting和up列表都为[3, 1, 2]。
 - 3) 选择出拥有权威日志的osd为1， 经过calc_replicated_acting算法， want列表为[1, 3, 2]， acting_backfill为[1, 3, 2]， want_backfill为[3]。特别注意want列表第一个为主OSD， 如果up_primay无法恢复， 就选择权威日志的OSD为主OSD。
 - 4) want[1, 3, 2]不等于acting[3, 1, 2]时， 并且不等于up[3, 1, 2]， 需要向Monitor申请pg_temp为want。

5) 申请成功pg_temp以后, acting为[3, 1, 2], up为[1, 3, 2], osd1做为临时的主OSD, 处理读写请求。当该PG恢复处于clean状态, pg_temp取消, acting和up都恢复为[3, 1, 2]。

2.find_best_info

函数find_best_info用于选取一个拥有权威日志的OSD。根据last_epoch_clean到目前为止, 各个past interval期间参与该PG的所有目前还处于up状态的OSD上pg_info_t信息, 来选取一个拥有权威日志的OSD, 选择的优先顺序如下:

- 1) 具有最新的last_update的OSD。
- 2) 如果条件1相同, 选择日志更长的OSD。
- 3) 如果1, 2条件都相同, 选择当前的主OSD。

代码实现具体的过程如下:

- 1) 首先在所有OSD中计算max_last_epoch_started, 然后在拥有最大的last_epoch_started的OSD中计算min_last_update_acceptable的值。
- 2) 如果min_last_update_acceptable为eversion_t::max(), 返回infos.end(), 选取失败。
- 3) 根据以下条件选择一个OSD:

- a) 首先过滤掉last_update小于min_last_update_acceptable, 或者last_epoch_started小于max_last_epoch_started_found, 或者处于incomplete的OSD。
- b) 如果PG类型是EC, 选择最小的last_update ;如果PG类型是副本, 选择最大的last_update的OSD。
- c) 如果上述条件都相同, 选择long tail最小的, 也就是日志最长的OSD。
- d) 如果上述条件都相同, 选择当前的主OSD。

综上的选择过程可知：拥有权威日志的OSD特征如下：必须是非incomplete的OSD；必须有最大last_epoch_strated；last_update有可能是最大，但至少是min_last_update_acceptable，有可能是日志最长的OSD，有可能是主OSD。

[3.calc_replicated_acting](#)

本函数计算本PG相关的下列OSD列表：

- want_primary：主OSD, 如果它不是up_primary, 就需要申请pg_temp。
- backfill：需要进行Backfill操作的OSD。
- acting_backfill：所有进行acting和Backfill的OSD的集合。

·want和acting_backfill的OSD相同，前者类型是pg_shard_t，后者为int型。

具体处理过程如下：

- 1) 首先选择want_primary列表中的OSD：
 - a) 如果up_primary处于非incomplete状态，并且last_update大于等于权威日志的log_tail，说明up_primary的日志和权威日志有重叠，可通过日志记录恢复，优先选择up_primary为主OSD。
 - b) 否则选择auth_log_shard，也就是拥有权威日志的OSD为主OSD。
 - c) 把主OSD加入到want和acting_backfill列表中。
- 2) 函数的输入参数size为要选择的副本数，依次从up、acting、all_info里选择size个副本OSD：
 - a) 如果该OSD上的PG处于incomplete的状态，或者cur_info.last_update小于主OSD和auth_log_shard的最小值，则该PG副本无法通过日志修复，只能通过Backfill操作来修复。把该OSD分别加入backfill和acting_backfill集合中。
 - b) 否则就可以根据PG日志来恢复，只加入acting_backfill集和want列表中，不用加入到Backfill列表中。

4. 收到缺失的权威日志

如果主OSD不是拥有权威日志的OSD，就需要去拥有权威日志的OSD上拉取权威日志：

```
boost::statechart::result PG::RecoveryState::GetLog::react(const MLogRec& logevt
```

当收到权威日志后，封装成MLogRec类型事件。本函数就用于处理该事件。它首先确认是从auth_log_shard端发送的消息，然后抛出GotLog()事件：

```
boost::statechart::result PG::RecoveryState::GetLog::react(const GotLog&)
```

本函数捕获GetLog事件，处理过程如下：

- 1) 如果msg不为空，就调用函数proc_master_log合并自己缺失的权威日志，并更新自己pg_info相关的信息。从此，做为主OSD，也是拥有权威日志的OSD。
- 2) 调用函数pg->start_flush添加一个空操作。
- 3) 状态转移到GetMissing状态。

经过GetLog阶段的处理后，该PG的主OSD已经获取了权威日志，以及pg_info的权威信息。

10.5.7 GetMissing

GetMissing的处理过程为：首先，拉取各个从OSD上的有效日志。其次，用主OSD上的权威日志与各个从OSD的日志进行对比，从而计算出各从OSD上不一致的对象并保存在对应的pg_missing_t结构中，做为后续数据修复的依据。

主OSD的不一致的对象信息，已经在调用函数proc_master_log合并权威日志的过程中计算出来，所以这里只计算从OSD上的不一致的对象。

1. 拉取从副本上的日志

在GetMissing的构造函数里，通过对比主OSD上的权威pg_info信息，来获取从OSD上的日志信息。

```
PG::RecoveryState::GetMissing::GetMissing(my_context ctx)
```

其具体处理过程为遍历pg->actingbackfill的OSD列表，然后做如下的处理：

- 1) 不需要获取PG日志的情况：
 - a) 如果pi.is_empty () 为空，没有任何信息，需要Backfill过程来修复，不需要获取日志。

- b) pi.last_update小于pg->pg_log.get_tail () , 该OSD的pg_info记录中, last_update小于权威日志的尾部记录, 该OSD的日志和权威日志不重叠, 该OSD操作已经远远落后于权威OSD, 已经无法根据日志来修复, 需要Backfill过程来修复。
 - c) pi.last_backfill为hobject_t () , 说明在past interval期间, 该OSD标记需要Backfill操作, 实际并没开始Backfill的工作, 需要继续Backfill过程。
 - d) pi.last_update等于pi.last_complete, 说明该PG没有丢失的对象, 已经完成Recovery操作阶段, 并且pi.last_update等于pg->info.last_update, 说明日志和权威日志的最后更新一致, 说明该PG数据完整, 不需要恢复。
- 2) 获取日志的情况 : 当pi.last_update大于pg->info.log_tail, 该OSD的日志记录和权威日志记录重叠, 可以通过日志来修复。变量sine是从last_epoch_started开始的版本值 :
- a) 如果该PG的日志记录pi.log_tail小于等于版本值since, 那就发送消息pg_query_t : : LOG, 从since开始获取日志记录。
 - b) 如果该PG的日志记录pi.log_tail大于版本值since, 就发送消息pg_query_t : : FULLLOG来获取该OSD的全部日志记录。
- 3) 最后检查如果peer_missing_requested为空, 说明所有获取日志的请求返回并处理完成。如果需要pg->need_up_thru, 抛出

post_event (NeedUpThru ()) ;否则，直接调用
post_event (Activate (pg->get_osdmap () ->get_epoch ())) 进入
Activate状态。

下面举例说明获取日志的两种情况：

权威日志	(9, 10)	(10,0) sine	(10,1)	(10,2)	(10,3)	last_update
osd0 日志					log_tail			
osd1 日志	log_tail							

当前last_epoch_started的值为10, sine是last_epoch_started后的首个日志版本值当前需要恢复的有效日志是经过sine操作之后的日志，之前的日子已经没有用了。

对应osd0，其日志log_tail小于since，全部拷贝osd0上的日志；对应osd1，其日志log_tail大于since，只拷贝从sine开始的日志记录。

2. 收到从副本上的日志记录处理

当一个PG的主OSD接收到从OSD返回的获取日志ACK应答后，就把该消息封装成MLogRec事件。状态GetMissing接收到该事件后，在下列事件函数里处理该事件：

```
boost::statechart::result PG::RecoveryState::GetMissing::react(const MLogRec& lo
```

具体过程如下：

- 1) 调用proc_replica_log处理日志。通过日志的对比，获取该OSD上处于missing状态的对象列表。
- 2) 如果peer_missing_requested为空，即所有的获取日志请求返回并处理。如果需要pg->need_up_thru，抛出NeedUpThru () 事件。否则，直接调用函数post_event (Activate (pg->get_osdmap () ->get_epoch ())) 进入Activate状态。

函数proc_replica_log处理各个从OSD上发过来的日志。它通过比较该OSD的日志和本地权威日志，来计算该OSD上处于missing状态的对象列表。具体处理过程调用pg_log.proc_replica_log来处理日志，输出为omissing，也就是该OSD缺失的对象。

10.5.8 Active操作

由上述可知，如果GetMissing处理成功，就跳转到Activate状态。到本阶段为止，可以说Peering主要工作已经完成，但还需要后续的处理，激活各个副本，如下所示：

```
PG::RecoveryState::Active::Active(my_context ctx)
```

状态Activate的构成函数里处理过程如下：

- 1) 在构造函数里初始化了remote_shards_to_reserve_recovery和remote_shards_to_reserve_backfill，需要Recovery操作和Backfill操作的OSD。
- 2) 调用函数pg->start_flush来完成相关数据的flush工作。
- 3) 调用函数pg->activate完成最后的激活工作。

1. MissingLoc

类MissingLoc用来记录处于missing状态对象的位置，也就是缺失对象的正确版本分析在哪些OSD上。恢复时就去这些OSD上去拉取正确对象的数据：

```
class MissingLoc {  
map<hobject_t, pg_missing_t::item, hobject_t::BitwiseComparator> needs_recovery_m  
//缺失的对象
```

```

--> item (现在版本, 缺失的版本)

map<hobject_t, set<pg_shard_t>, hobject_t::BitwiseComparator > missing_loc;
//缺失的对象

---> 所在的

OSD集合

set<pg_shard_t> missing_loc_sources;
//所有缺失对象所在的

OSD集合

PG *pg;
set<pg_shard_t> empty_set;
public:
    boost::scoped_ptr<IsPGReadablePredicate> is_readable;
boost::scoped_ptr<IsPGRecoverablePredicate> is_recoverable;
}

```

下面介绍一些MissingLog处理函数，作用是添加相应的missing对象列表。其对应两个函数：add_active_missing函数和add_source_info函数。

add_active_missing函数用于把一个副本中的所有缺失对象添加到MissingLoc的needs_recovery_map结构里：

```
void add_active_missing(const pg_missing_t &missing)
```

add_source_info函数用于计算每个缺失对象是否在本OSD上：

```
PG::MissingLoc::add_source_info(pg_shard_t fromosd,
const pg_info_t &oinfo, const pg_missing_t &omissing,
bool sort_bitwise, ThreadPool::TPHandle* handle)
```

具体实现如下：

遍历needs_recovery_map里的所有对象，对每个对象做如下处理：

- 1) 如果oinfo.last_update<need (所需的缺失对象的版本) 大于 oinfo.last_update的值，就跳过。
- 2) 如果该PG正常的last_backfill指针小于MAX值，说明还处于Backfill 阶段，但是sort_bitwise不正确，跳过。
- 3) 如果该对象大于last_backfill，显然该对象不存在，跳过。
- 4) 如果该对象大于last_complete，说明该对象或者是上次Peering之后 缺失的对象，还没有来得及恢复；或者是新创建的对象。检查如果在 missing记录已存在，就是上次缺失的对象，直接跳过；否则就是新创建的对象，存在该OSD中。
- 5) 经过上述检查后，确认该对象在本OSD上，在missing_loc添加该对象的location为本OSD。

2. Activate状态

PG :: activate函数是Peering过程的最后一步，该函数完成以下功能：

- 更新一些pg_info的参数信息。
- 给replica发消息，激活副本PG。
- 计算MissingLoc，也就是缺失对象分布在哪些OSD上，用于后续的恢

复。

具体处理过程如下：

- 1) 如果需要客户回答，就把PG添加到replay_queue队列里。
- 2) 更新info.last_epoch_started变量， info.last_epoch_started指的是本OSD在完成目前Peering进程后的更新，而info.history.last_epoch_started是PG的所有的OSD都确认完成Peering的更新。
- 3) 更新一些相关的字段。
- 4) 注册C_PG_ActivateCommitted回调函数，该函数最终完成activate的工作。
- 5) 初始化snap_trimq快照相关的变量。
- 6) 设置info.last_complete指针：
 - 如果missing.num_missing () 等于0，表明处于clean状态。直接更新info.last_complete等于info.last_update，并调用pg_log.reset_recovery_pointers () 调整log的complete_to指针。
 - 否则，如果有需要恢复的对象，就调用函数pg_log.activate_not_complete (info)，设置info.last_complete为缺失的第一个对象的前一版本。

7) 以下都是主OSD的操作，给每个从OSD发送MOSDPGLog类型的消息，激活该PG的从OSD上的副本。分别对应三种不同处理：

·如果pi.last_update等于info.last_update，这种情况下，该OSD本身就是clean的，不需要给该OSD发送其他信息。添加到activator_map只发送pg_info来激活从OSD。其最终的执行在PeeringWQ的线程执行完状态机的事件处理后，在函数OSD::dispatch_context里调用OSD::do_infos函数实现。

·需要Backfill操作的OSD，发送pg_info，以及osd_min_pg_log_entries数量的PG日志。

·需要Recovery操作的OSD，发送pg_info，以及从缺失的日志。

8) 设置MissingLoc，也就是统计缺失的对象，以及缺失的对象所在的OSD，核心就是调用MissingLoc的add_source_info函数，见MissingLoc相关的分析。

9) 如果需要恢复，把该PG加入到osd->queue_for_recovery (this) 的恢复队列中。

10) 如果PG的size小于act set的size，也就是当前的OSD不够，就标记PG的状态为PG_STATE_DEGRADED和PG_STATE_UNDERSIZED状态，最后标记PG为PG_STATE_ACTIVATING状态。

3. 收到从OSD的MOSDPGLog的应对

当收到从OSD发送的MOSDPGLog的ACK消息后，触发MInfoRec事件，下面这个函数处理该事件：

```
boost::statechart::result PG::RecoveryState::Active::react(const MInfoRec& infoevt)
```

处理过程比较简单：检查该请求的源OSD在本PG的actingbacfill列表中，以等待列表中删除该OSD。最后检查，当收集到所有的从OSD发送的ACK，就调用函数all_activate_and_committed触发AllReplicasActivated事件。

对应主OSD在事务的回调函数C_PG_ActivateCommitted里实现，最终调用_activate_committed加入peer_activated集合里。

4.AllReplicasActivated

这个函数处理AllReplicasActivated事件：

```
boost::statechart::result PG::RecoveryState::Active::react(const AllReplicasActivated &evt)
```

当所有的replica处于activated状态时，进行如下处理：

- 1) 取消PG_STATE_ACTIVATING和PG_STATE_CREATING状态，如果该PG上acting状态的OSD数量大于等于Pool的min_size，设置该PG为PG_STATE_ACTIVE的状态；否则设置为PG_STATE_PEERED状态。

- 2) ReplicatedPG :: check_local 检查本地的stray对象是否都被删除。
- 3) 如果有读写请求在等待Peering操作，则把该请求添加到处理队列 pg->requeue_ops (pg->waiting_for_peered) 。
- 4) 调用函数ReplicatedPG :: on_activate，如果需要Recovery操作，触发DoRecovery事件，如果需要Backfill操作，触发RequestBackfill事件；否则触发AllReplicasRecovered事件。
- 5) 初始化Cache Tier需要的hit_set对象。
- 6) 初始化Cache Tier需要的agent对象。

10.5.9 副本端的状态转移

当创建PG后，根据不同的角色，如果是主OSD， PG对应的状态机就进入了Primary状态。如果不是主OSD， 就进入Stray状态。

1.Stray状态

Stray状态有两种情况。

情况1：只接收到PGINFO的处理：

```
boost::statechart::result PG::RecoveryState::Stray::react(const MInfoRec& infoev
```

从PG接收到主PG发送的MInfoRec事件，也就是接收到主OSD发送的pg_info信息。其判断如果当前pg->info.last_update大于infoevt.info.last_update，说明当前的日志有divergent的日志，调用函数rewind_divergent_log清理日志即可。最后抛出Activate (infoevt.info.last_epoch_started事件，进入ReplicaActive状态。

情况2：接收到MOSDPGLog消息：

```
boost::statechart::result PG::RecoveryState::Stray::react(const MLogRec& logevt)
```

当从PG接收到MLogRec事件，就对应着接收到主PG发送的MOSDPGLog消息，其通知从PG处于activate状态，具体处理过程如下：

- 1) 如果msg->info.last_backfill为hobject_t () , 需要Backfill操作的OSD。
- 2) 否则就是需要Recovery操作的OSD, 调用merge_log把主OSD发送过来的日志合并。
抛出Activate (logevt.msg->info.last_epoch_started) 事件, 使副本转移
到ReplicaActive状态。

2.ReplicaActive状态

ReplicaActive状态如下：

```
boost::statechart::result PG::RecoveryState::ReplicaActive::react(  
    const Activate& actevt)
```

当处于ReplicaActive状态, 接收到Activate事件, 就调用函数pg->activate, 在函数_activate_committed给主PG发送应答信息, 告诉自己处于activate状态, 设置PG为activate状态。

10.5.10 状态机异常处理

在上面的流程介绍中，只介绍了正常状态机的转换流程。Ceph之所以用状态机来实现PG的状态转换，就是可以实现任何异常情况下的处理。下面介绍当OSD失效时导致相关的PG重新进行Peering的机制。

当一个OSD失效，Monitor会通过heartbeat检测到，导致osd map发生了变化，Monitor会把最新的osd map推送给OSD，导致OSD上的受影响PG重新进行Peering操作。

具体的流程如下：

- 1) 在函数OSD`::handle_osd_map`处理osd map的变化，该函数调用`consume_map`，对每一个PG调用`pg->queue_null`，把PG加入到`peering_wq`中。
- 2) `peering_wq`的处理函数`process_peering_events`调用OSD`::advance_pg`函数，在该函数里调用PG`::handle_advance_map`给PG的状态机`RecoveryMachine`发送`AdvMap`事件：

```
boost::statechart::result PG::RecoveryState::Started::react(const AdvMap& advmap)
```

当处于`Started`状态，接收到`AdvMap`事件，调用函数`pg->should_restart_peering`检查，如果是`new_interval`，就跳转到`Reset`状态，重

新开始一次Peering过程。

10.6 本章小结

本章介绍了Ceph的Peering过程，其核心过程就是通过各个OSD上保存的PG日志选择出一个权威日志的OSD。以该OSD上的日志为基础，对比其他OSD上的日志记录，计算出各个OSD上缺失的对象信息。这样，PG就使各个OSD的数据达成了一致。

第11章 Ceph数据修复

当PG完成了Peering过程后，处于Active状态的PG就已经可以对外提供服务了。如果该PG的各个副本上有不一致的对象，就需要进行修复。Ceph的修复过程有两种：Recovery和Backfill。

Recovery是仅依据PG日志中的缺失记录来修复不一致的对象。Backfill是PG通过重新扫描所有的对象，对比发现缺失的对象，通过整体拷贝来修复。当一个OSD失效时间过长导致无法根据PG日志来修复，或者新加入的OSD导致数据迁移时，就会启动Backfill过程。

从第10章可知，PG完成Peering过程后，就处于activate状态，如果需要Recovery，就产生DoRecovery事件，触发修复操作。如果需要Backfill，就会产生RequestBackfill事件来触发Backfill操作。在PG的数据修复过程中，如果既有需要Recovery过程的OSD，又有需要Backfill过程的OSD，那么处理过程需要先进行Recovery过程的修复，再完成Backfill过程的修复。

本章介绍Ceph的数据修复的实现过程。首先介绍数据修复的资源预约的知识，然后通过介绍修复的状态转换图，大概了解整个数据修复的过程。最后分别详细介绍Recovery过程和Backfill过程的具体实现。

11.1 资源预约

在数据修复的过程中，为了控制一个OSD上正在修复的PG最大数目，需要资源预约，在主OSD上和从OSD上都需要预约。如果没有预约成功，需要阻塞等待。一个OSD能同时修复的最大PG数在配置选项osd_max_backfills中设置，默认值为1。

类AsyncReserver用来管理资源预约，其模板参数<T>为要预约的资源类型。该类实现了异步的资源预约。当成功完成资源预约后，就调用注册的回调函数通知调用方预约成功：

```
class AsyncReserver {
    unsigned max_allowed;      // 定义允许的最大资源数量，在这里指允许修复的
                                // PG的数量

    unsigned min_priority;     // 最小的优先级

    Finisher *f;
                                // 当预约成功后，用来执行的回调函数

    map<unsigned, list<pair<T, Context*> >> queues;
                                // 优先级到待预约资源链表的映射，

    pair<T, Context*> 定义预约的资源和注册的回调函数

    map<T, pair<unsigned, typename list<pair<T, Context*> >::iterator >> queue_poi
                                // queue链表中的位置指针

    set<T> in_progress;       // 预约成功，正在使用的资源

}
```

1. 资源预约

函数request_reservation用于预约资源：

```
void request_reservation(  
    T item, //输入参数：申请的资源  
    Context *on_reserved, //输入参数：申请成功后的回调函数
```

具体处理过程如下：

- 1) 把要请求的资源根据优先级添加到queue队列中，并在queue_pointers中添加其对应的位置指针：

```
queues[prio].push_back(make_pair(item, on_reserved));  
queue_pointers.insert(make_pair(item, make_pair(prio, --(queues[prio]).  
end())));
```

- 2) 调用函数do_queues用来检查queue中的所有资源预约申请：从优先级高的请求开始检查，如果还有配额并且其请求的优先级至少不小于最小优先级，就把资源授权给它。
- 3) 在queue队列中删除该资源预约请求，并在queue_pointers删除该资源的位置信息。把该资源添加到in_progress队列中，并把请求相应的回调函数添加到Finisher类中，使其执行该回调函数。最后通知预约成功。

2. 取消预约

函数cancel_reservation用于释放拥有的不再使用的资源：

```
void cancel_reservation(  
    T item  
        //输入参数：删除申请的资源  
)
```

具体处理过程如下：

- 1) 如果该资源还在queue队列中，就删除（这属于异常情况的处理）；否则在in_progress队列中删除该资源。
- 2) 调用do_queues函数把该资源重新授权给其他等待的请求。

11.2 数据修复状态转换图

如图11-1所示的是修复过程状态转换图。当PG进入Active状态后，就进入默认的子状态Activating。

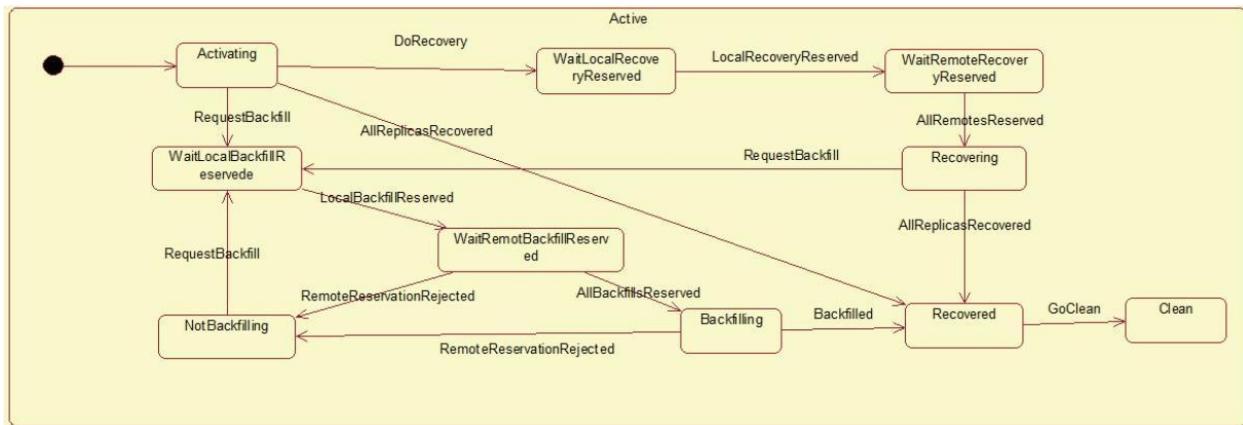


图11-1 修复过程状态转换图

数据修复的状态转换过程如下所示。

情况1：当进入Activating状态后，如果此时所有的副本都完整，不需要修复，其状态转移过程如下：

- 1) Activating状态接收到AllReplicatedRecovered事件，直接转换到Recovered状态。
- 2) Recovered状态接收到GoClean事件，整个PG转入Clean状态。

情况2：当进入Activating状态后，没有Recovery过程，只需要Backfill过程的情况：

- 1) Activating状态直接接收到RequestBackfill事件，进入WaitLocalBackfillReserved状态。
- 2) 当WaitLocalBackfillReserved状态接收到LocalBackfillReserved事件后，意味着本地资源预约成功，转入WaitRemoteBackfillReserved状态。
- 3) 所有副本资源预约成功后，主PG就会接收到AllBackfillsReserved事件，进入Backfilling状态，开始实际数据Backfill操作过程。
- 4) Backfilling状态接收Backfilled事件，标志Backfill过程完成，进入Recovered状态。
- 5) 异常处理：当在状态WaitRemoteBackfillReserved和Backfilling接收到Remote ReservationRejected事件时，表明资源预约失败，进入NotBackfilling状态，再次等待RequestBackfilling事件来重新发起Backfill过程。

情况3：当PG既需要Recovery过程，也可能需要Backfill过程时，PG先完成Recovery过程，再完成Backfill过程，特别强调这里的先后顺序。其具体过程如下：

- 1) Activating状态：在接收到DoRecovery事件后，转移到WaitLocalRecoveryReserved状态。
- 2) WaitLocalRecoveryReserved状态：在这个状态中完成本地资源的预约。当收到LocalRecoveryReserved事件后，标志着本地资源预约的完成，

转移到WaitRemote-RecoveryReserved状态。

3) WaitRemoteRecoveryReserved状态：在这个状态中完成远程资源的预约。当接收到AllRemotesReserved事件，标志着该PG在所有参与数据修复的从OSD上完成资源预约，进入Recovering状态。

4) Recovering状态：在这个状态中完成实际的数据修复工作。完成后把PG设置为PG_STATE_RECOVERING状态，并把PG添加到recovery_wq工作队列中，开始启动数据修复：

```
pg->state_clear(PG_STATE_RECOVERY_WAIT);
pg->state_set(PG_STATE_RECOVERING);
pg->osd->queue_for_recovery(pg);
```

5) 在Recovering状态完成Recovery工作后，如果需要Backfill工作，就接收RequestBackfill事件，转入Backfill流程。

6) 如果没有Backfill工作流程，直接接收AllReplicasRecovered事件，转入Recovered状态。

7) Recovered状态：到达本状态，意味着已经完成数据修复工作。当收到事件GoClean后，PG就进入clean状态。

11.3 Recovery过程

数据修复的依据是在Peering过程中产生的如下信息：

- 主副本上的缺失对象的信息保存在pg_log类的pg_missing_t结构中。
- 各从副本上的缺失对象信息保存在OSD对应的peer_missing中的pg_missing_t结构中。
- 缺失对象的位置信息保存在类MissingLoc中。

根据以上信息，就可以知道该PG里各个OSD缺失的对象信息，以及该缺失的对象目前在哪些OSD上有完整的信息。基于上面的信息，数据修复过程就相对比较清晰：

- 对于主OSD缺失的对象，随机选择一个拥有该对象的OSD，把数据拉取过来。
- 对于replica缺失的对象，从主副本上把缺失的对象数据推送到从副本上来完成数据的修复。
- 对于比较特殊的快照对象，在修复时加入了一些优化的方法。

11.3.1 触发修复

Recovery过程由PG的主OSD来触发并控制整个修复的过程。在修复的过程中，先修复主OSD上缺失（或者不一致）的对象，然后修复从OSD上缺失的对象。由数据修复状态转换过程可知，当PG处于Activate/Recovering状态后，该PG被加入到OSD的RecoveryWQ工作队列中。在recovery_wq里，其工作队列的线程池的处理函数调用do_recovery函数来执行实际的数据修复操作：

```
void OSD::do_recovery(PG *pg, ThreadPool::TPHandle &handle)
```

函数do_recovery由RecoveryWQ工作队列的线程池的线程执行。其输入的参数为要修复的PG，具体处理流程如下：

- 1) 配置选项osd_recovery_sleep设置了线程做一次修复后的休眠时间。如果设置了该值，每次线程开始先休眠相应的时间长度。该参数默认值为0，不需要休眠。
- 2) 加recovery_wq.lock () 锁，用来保护recovery_wq队列以及变量recovery_ops_active。计算可修复对象的max值，其值为允许修复的最大对象数osd_recovery_max_active减去正在修复的对象数recovery_ops_active，然后调用函数recovery_wq.unlock () 解锁。
- 3) 如果max小于等于0，即没有修复对象的配额，就把PG重新加入工

作队列recovery_wq中并返回；否则如果max大于0，调用pg->lock_suspend_timeout (handle) 重新设置线程超时时间。检查PG的状态，如果该PG处于正在被删除状态，或者既不处于peered状态，也不是主OSD，则直接退出。

- 4) 调用函数pg->start_recovery_ops修复，返回值more为还需要修复的对象数目。输出参数started为已经开始修复的对象数。
- 5) 如果more为0，也就是没有修复的对象了。但是pg->have_unfound () 不为0，还有unfound对象（即缺失的对象，目前不知道在哪个OSD上能找到完整的对象），调用函数discover_all_missing在might_have_unfound队列中的OSD上继续查找该对象，查找的方法就是给相关的OSD发送获取该OSD的pg_log的消息。
- 6) 如果rctx.query_map->empty () 为空，也就是没有找到其他OSD去获取pg_log来查找unfound对象，就结束该PG的recover操作，调用函数从recovery_wq._dequeue (pg) 删除PG。
- 7) 函数dispatch_context做收尾工作，在这里发送query_map的请求，把ctx.transaction的事务提交到本地对象存储中。

由上过程分析可知，do_recovery函数的核心功能是计算要修复对象的max值，然后调用函数start_recovery_ops来启动修复。

11.3.2 ReplicatedPG

类ReplicatedPG用于处理Replicate类型PG的相关修复操作。下面分析它用于修复的start_recovery_ops函数及其相关函数的具体实现。

1.start_recovery_ops

函数start_recovery_ops调用recovery_primary和recovery_replicas来修复该PG上对象的主副本和从副本。修复完成后，如果仍需要Backfill过程，则抛出相应事件触发PG状态机，开始Backfill的修复进程。

```
bool ReplicatedPG::start_recovery_ops(int max, RecoveryCtx *prctx,
                                         ThreadPool::TPHandle &handle, int *ops_started)
```

该函数具体处理过程如下：

- 1) 首先检查OSD，确保该OSD是PG的主OSD。如果PG已经处于PG_STATE_RECOVERING或者PG_STATE_BACKFILL的状态则退出。
- 2) 从pg_log获取missing对象，它保存了主OSD缺失的对象。参数num_missing为主OSD缺失的对象数目；num_unfound为该PG上缺失的对象却没有找到该对象其他正确副本所在的OSD；如果num_missing为0，说明主OSD不缺失对象，直接设置info.last_complete为最新版本info.last_update的值。

- 3) 如num_missing等于num_unfound，说明主OSD所缺失对象都为unfound类型的对象，先调用函数recover_replicas启动修复replica上的对象。
- 4) 如果started为0，也就是已经启动修复的对象数量为0，调用函数recover_primary修复主OSD上的对象。
- 5) 如果started仍然为0，且num_unfound有变化，再次启动recover_replicas修复副本。
- 6) 如果started不为零，设置work_in_progress的值为true。
- 7) 如果recovering队列为空，也就是没有正在进行Recovery操作的对象，状态为PG_STATE_BACKFILL，并且backfill_targets不为空，started小于max，missing.num_missing () 为0，的情况下：
 - a) 如果标志get_osdmap () ->test_flag (CEPH_OSDMAP_NOBACKFILL) 设置了，就推迟Backfill过程。
 - b) 如果标志CEPH_OSDMAP_NOREBALANCE设置了，且是degrade的状态，推迟Backfill过程。
 - c) 如果backfill_reserved没有设置，就抛出RequestBackfill事件给状态机，启动Backfill过程。

- d) 否则，调用函数recover_backfill开始Backfill过程。
- 8) 最后PG如果处于PG_STATE_RECOVERING状态，并且对象修复成功，就检查：如果需要Backfill过程，就向PG的状态机发送RequestBackfill事件；如果不需要Backfill过程，就抛出AllReplicasRecovered事件。
- 9) 否则，PG的状态就是PG_STATE_BACKFILL状态，清除该状态，抛出Backfilled事件。

[2.recover_primary](#)

函数recover_primary用来修复一个PG的主OSD上缺失的对象：

```
int ReplicatedPG::recover_primary(int max, ThreadPool::TPHandle &handle)
```

其处理过程如下：

- 1) 调用pgbackend->open_recovery_op返回一个PG类型相关的PGBackend :: Recovery Handle。对于ReplicatedPG对应的RPGHandle，内部有两个map，保存了Push和Pull操作的封装PushOp和PullOp：

```
struct RPGHandle : public PGBackend::RecoveryHandle {
    map<pg_shard_t, vector<PushOp> > pushes;
    map<pg_shard_t, vector<PullOp> > pulls;
};
```

- 2) last_requested为上次修复的指针，通过调用low_bound函数来获取还没有修复的对象。

3) 遍历每一个未被修复的对象：latest为日志记录中保存的该缺失对象的最后一條日志，soid为缺失的对象。如果latest不为空：

- a) 如果该日志记录是pg_log_entry_t :: CLONE类型，这里不做任何的特殊处理，直到成功获取snapshot相关的信息SnapSet后再处理。
- b) 如果该日志记录类型为pg_log_entry_t :: LOST_REVERT类型：该revert操作为数据不一致时，管理员通过命令行强行回退到指定版本，reverting_to记录了回退的版本号：

·如果item.have等于latest->reverting_to版本，也就是通过日志记录显示当前已经拥有回退的版本，那么就获取对象的ObjectContext，如果检查对象当前的版本obc->obs.oi.version等于latest->version，说明该回退操作完成。

·如果item.have等于latest->reverting_to，但是对象当前的版本obc->obs.oi.version不等于latest->version，说明没有执行回退操作，直接修改对象的版本号为latest->version即可。

·否则，需要拉取该reverting_to版本的对象，这里不做特殊的处理，只是检查所有OSD是否拥有该版本的对象，如果有就加入到missing_loc记录该版本的位置信息，由后续修复继续来完成。

- c) 如果该对象在recovering过程中，表明正在修复，或者其head对象正在修复，跳过，并计数增加skipped；否则调用函数recover_missing来修

复。

4) 调用函数pgbackend->run_recovery_op, 把PullOp或者PushOp封装的消息发送出去。

下面举例说明，当最后的日志记录类型为LOST_REVERT时的修复过程。

例11-1 日志修复过程。

PG日志的记录如下：每个单元代表一条日志记录，分别为对象的名字和版本以及操作，版本的格式为（epoch, version）。灰色的部分代表本OSD上缺失的日志记录，该日志记录是从权威日志记录中拷贝过来的，所以当前该日志记录是连续完整的。

obj2(1,3) modify	obj1(1,4) modify	obj2(1,5) modify	obj1(1,6) modify	obj1(1,7) modify	obj1(1,8) modify
------------------	------------------	------------------	------------------	------------------	------------------

情况1：正常情况的修复。

缺失的对象列表为[obj1, obj2]。当前修复对象为obj1。由日志记录可知：对象obj1被修改过三次，分别为版本6, 7, 8。当前拥有的obj1对象的版本have值为4，修复时只修复到最后修改的版本8即可。

情况2：最后一个操作为LOST_REVERT类型的操作。

obj2(1,3) modify	obj1(1,4) modify	obj2(1,5) modify	obj1(1,6) modify	obj1(1,7) modify	obj1(1,8) lost_revert_ version = 8 prior_version=7 reverting_to=4
------------------	------------------	------------------	------------------	------------------	---

对于要修复的对象obj1，最后一次操作为LOST_REVERT类型的操作，该操作当前版本version为8，修改前的版本prior_version为7，回退版本reverting_to为4。

在这种情况下，日志显示当前已经有版本4，检查对象obj1的实际版本，也就是object_info里保存的版本号：

- 1) 如果该值是8，说明最后一次revert操作成功，不需要做任何修复动作。
- 2) 如果该值是4，说明LOST_REVERT操作就没有执行。当然数据内容已经是版本4了，只需要修改object_info的版本为8即可。

如果回退的版本reverting_to不是版本4，而是版本6，那么最终还是需要把obj1的数据修复到版本6的数据。Ceph在这里的处理，仅仅是检查其他OSD缺失的对象中是否有版本6，如果有，就加入到missing_loc中，记录拥有该版本的OSD位置，待后续继续修复。

3.recover_missing

函数recovery_missing处理snap对象的修复。在修复snap对象时，必须首先修复head对象或者snapdir对象，获取SnapSet信息，然后才能修复快照

对象自己。

```
int ReplicatedPG::recover_missing(const hobject_t &soid, eversion_t v, int priority, PGBackend::RecoveryHandle *h)
```

具体实现如下：

- 1) 检查如果对象soid是unfound, 直接返回PULL_NONE值。暂时无法修复处于unfound的对象。
- 2) 如果修复的是snap对象：
 - a) 查看如果对应的head对象处于missing, 递归调用函数recover_missing先修复head对象。
 - b) 查看如果snapdir对象处于missing, 就递归调用函数recover_missing先修复snapdir对象。
- 3) 从head对象或者snapdir对象中获取head_obi信息。
- 4) 调用函数pgbackend->recover_object把要修复的操作信息封装到PullOp或者PushOp对象中，并添加到RecoveryHandle结构中。

11.3.3 pgbackend

pgbackend封装了不同类型的Pool的实现。ReplicatedBackend实现了replicate类型的PG相关的底层功能，ECbackend实现了Erasure code类型的PG相关的底层功能。

由11.3.2节的分析可知，需要调用pgbackend的recovery_object函数来实现修复对象的信息封装。这里只介绍基于副本的。

函数recovery_object实现pull操作，调用prepare_pull操作把请求封装成PullOp结构。如果是push操作，就调用start_pushes把请求封装成PushOp的操作。

1.pull操作

Prepare_pull函数把要拉取的object相关的操作信息打包成PullOp类信息，如下所示：

```
void ReplicatedBackend::prepare_pull(
    eversion_t v,           //要拉取对象的版本信息

    const hobject_t& soid,   //要拉取的对象

    ObjectContextRef headctx, //拉取对象的
    ObjectContext信息

    RPGHandle *h)           //封装后保存的
    RecoveryHandle
```

难点在于snap对象的修复处理过程。下面先介绍PullOp数据结构。

PullOp数据结构如下：

```
struct PullOp {
    hobject_t soid;                                //需要拉取的对象

    ObjectRecoveryInfo recovery_info;               //对象修复的信息

    ObjectRecoveryProgress recovery_progress;        //对象修复进度信息

}

struct ObjectRecoveryInfo {
    hobject_t soid;                                //修复的对象

    eversion_t version;                            //修复对象的版本

    uint64_t size;                                 //修复对象的大小

    object_info_t oi;                             //修复对象的

    object_info信息

    SnapSet ss;                                    //修复对象的快照信息

    interval_set<uint64_t> copy_subset;
    //对象需要拷贝的集合

    , 在修复快照对象时，需要从别的

    OSD拷贝到本地的对象的区段集合

    map<hobject_t, interval_set<uint64_t>> clone_subset;
    //clone对象修复时，需要从本地对象拷贝来修复的区间

}

struct ObjectRecoveryProgress {
    bool first;                                   //是否是首次修复操作

    uint64_t data_recovered_to;                  //数据已经修复的位置指针
```

```
    bool data_complete; //数据是否修复完成

    string omap_recovered_to; //omap已经修复的位置指针

    bool omap_complete; //omap是否修复完成

}
```

函数prepare_pull具体处理过程如下：

- 1) 通过调用函数get_parent () 来获取PG对象的指针。pgbackend的parent就是相应的PG对象。通过PG获取missing、 peer_missing、 missing_loc等信息。
- 2) 从soid对象对应的missing_loc的map中获取该soid对象所在的OSD集合。把该集合保存在shuffle这个向量中。调用random_shuffle操作对OSD列表随机排序，然后选择向量中首个OSD作为缺失对象来拉取源OSD的值。从这一步可知，当修复主OSD上的对象，而多个从OSD上有该对象时，随机选择其中一个源OSD来拉取。
- 3) 当选择了一个源shard之后，查看该shard对应的peer_missing来确保该OSD上不缺失该对象，即确实拥有该版本的对象。
- 4) 确定拉取对象的数据范围：

- a) 如果是head对象，直接拷贝对象的全部，在copy_subset加入区间(0, -1)，表示全部拷贝，最后设置size为-1：

```
recovery_info.copy_subset.insert(0, (uint64_t)-1);
```

```
recovery_info.size = ((uint64_t)-1);
```

b) 如果该对象是snap对象，确保head对象或者snapdir对象二者必须存在一个。如果headctx不为空，就可以获取SnapSetContext对象，它保存了snapshot相关的信息。调用函数calc_clone_subsets来计算需要拷贝的数据范围。

5) 设置PullOp的相关字段，并添加到RPGHandle中。

函数calc_clone_subsets用于修复快照对象。在介绍它之前，这里需要介绍SnapSet的数据结构和clone对象的overlap概念。

在SnapSet结构中，字段clone_overlap保存了clone对象和上一次clone对象的重叠的部分：

```
struct SnapSet {
    snapid_t seq;
    bool head_exists;
    vector<snapid_t> snaps;      //序号降序排列

    vector<snapid_t> clones;     //序号升序排列

    map<snapid_t, interval_set<uint64_t> > clone_overlap;
        //写操作导致的和最新的克隆对象重叠的部分

    map<snapid_t, uint64_t> clone_size;
};
```

下面通过一个示例来说明clone_overlap数据结构的概念。

[例11-2](#) clone_overlap数据结构如图11-2所示：

snap2	1	2	3	4	5	6	7	8
snap3	1	2	3	4	5	6	7	8

图11-2 clone_overlap示意图

snap3从snap2对象clone出来，并修改了区间3和4，其在对象中范围的offset和length为(4, 8)和(8, 12)。那么在SnapSet的clone_overlap中就记录：

```
clone_overlap[3] = { 4,8
,(8,12)}
```

函数calc_clone_subsets用于修复快照对象时，计算应该拷贝的数据区间。在修复快照对象时，并不是完全拷贝快照对象，这里用于优化的关键在于：快照对象之间是有数据重叠，数据重叠的部分可以通过已存在的本地快照对象的数据拷贝来修复；对于不能通过本地快照对象拷贝修复的部分，才需要从其他副本上拉取对应的数据。

函数calc_clone_subsets具体实现如下：

1) 首先获取该快照对象的size，把(0, size)加入到data_subset中：

```
data_subset.insert(0, size);
```

2) 向前查找(oldest snap)和当前快照相交的区间，直到找到一个不

缺失的快照对象，添加到clone_subsets中。这里找的不重叠区间，是从不缺失快照对象到当前修复的快照对象之间从没有修改过的区间，所以修复时，直接从已存在的快照对象拷贝所需区间数据即可。

- 3) 同理，向后查找（newset snap）和当前快照对象相重叠的对象，直到找到一个不缺失的对象，添加到clone_subsets中。
- 4) 去除掉所有重叠的区间，就是需要拉取的数据区间：

```
data_subset.subtract(cloning);
```

对于上述的算法，下面举例来说明。

例11-3 快照对象修复示例如图11-3所示：

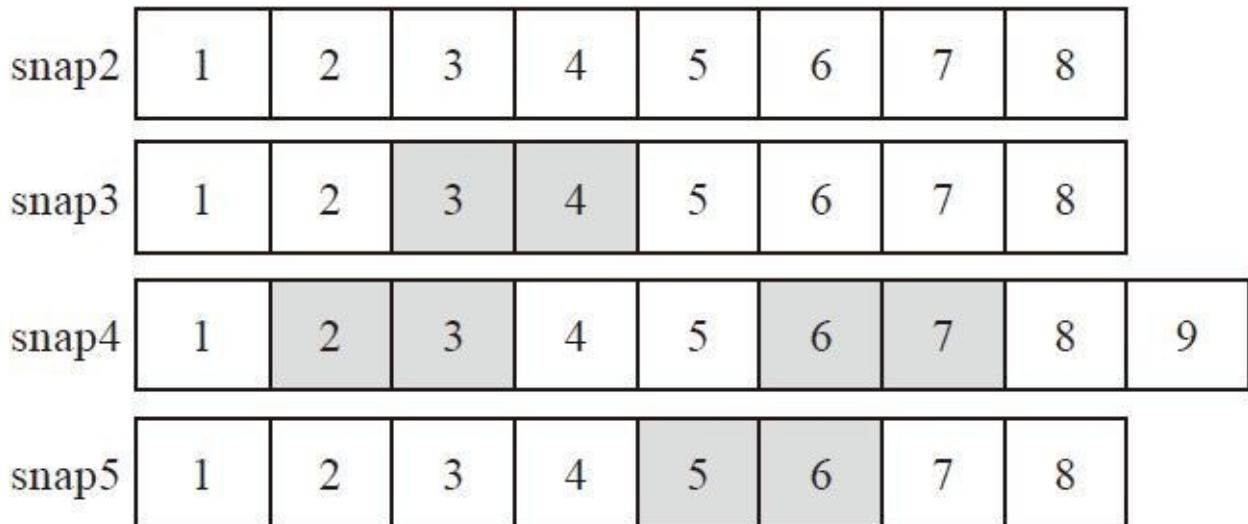


图11-3 快照对象修复计算示例图

要修复的对象为snap4，不同长度代表各个clone对象的size是不同的，

其中灰色的区间代表clone后修改的区间。snap2、snap3和snap5都是已经存在的非缺失对象。

算法处理流程如下：

- 1) 向前查找和snap4重叠的区间，直到遇到非缺失对象snap2为止。从snap4到snap2一直重叠的区间为1, 5, 8三个区间。因此，修复对象snap4时，修复1, 5, 8区间的数据，可以直接从已经存在的本地非缺失对象snap2拷贝即可。
- 2) 同理，向后查找和snap4重叠的区间，直到遇到非缺失对象snap5为止。snap5和snap4重叠的区间为1, 2, 3, 4, 7, 8六个区间。因此，修复对象snap4时，直接从本地对象snap4中拷贝区间1, 2, 3, 4, 7, 8即可。
- 3) 去除上述本地就可修复的区间，对象snap4只有区间6需要从其他OSD上拷贝数据来修复。

2.push操作

函数start_pushes获取actingbackfill的OSD列表，通过peer_missing查找缺失该对象的OSD，调用prep_push_to_replica打包PushOp请求。

函数prep_push_to_replica实现过程如下：

- 1) 如果需要push的对象是snap对象：检查如果head对象缺失，调用prep_push推送head对象；如果是headdir对象缺失，则调用prep_push推送

headdir对象。

2) 如果是snap对象，调用函数calc_clone_subsets来计算需要推送的快照对象的数据区间。

3) 如果是head对象，调用calc_head_subsets来计算需要推送的head对象的区间，其原理和计算快照对象类似，这里就不详细说明了。最后调用prep_push封装PushInfo信息，在函数build_push_op里读取要push的实际数据。

3.处理修复操作

函数run_recover_op调用send_pushed函数和send_pulls函数把请求发送给相关的OSD，这个流程比较简单。

当主OSD把对象推送给缺失该对象的从OSD后，从OSD需要调用函数handle_push来实现数据写入工作，从而来完成该对象的修复。同样，当主OSD给从OSD发起拉取对象的请求来修复自己缺失的对象时，需要调用函数handle_pulls来处理该请求的应对。

在函数ReplicatedBackend :: handle_push里处理handle_push的请求，主要调用submit_push_data函数来写入数据。

handle_pulls函数收到一个PullOp操作，返回PushOp操作，处理流程如下：

- 1) 首先调用store->stat函数，验证该对象是否存在，如果不存在，则调用函数prep_push_op_blank，直接返回空值。
- 2) 如果该对象存在，获取ObjectRecoveryInfo和ObjectRecoveryProgress结构。如果progress.first为true并且recovery_info.size为-1，说明是全拷贝修复：将recovery_info.size设置为实际对象的size，清空recovery_info.copy_subset，并把(0, size)区间添加到recovery_info.copy_subset.insert (0, st.st_size)的拷贝区间。
- 3) 调用函数build_push_op，构建PullOp结构。如果出错，调用prep_push_op_blank，直接返回空值。

函数build_push_on完成构建push的请求。具体处理如下：

- 1) 如果progress.first为true，就需要获取对象的元数据信息。通过store->omap_get_header获取omap的header信息，通过store->getattr获取对象的扩展属性信息，并验证oi.version是否为recovery_info.version；否则返回-EINVAL值。如果成功，new_progress.first设置为false。
- 2) 上一步只是获取了omap的header信息，并没有获取omap信息。这一步首先判断progress omap_complete是否完成，（初始化设置为false）如果没有完成，就迭代获取omap的(key, value)信息，并检查一次获取信息的大小不能超过cct->_conf->osd_recovery_max_chunk设置的值（默认为8MB）。特别需要注意的是，当该配置参数的值小于一个对象的size时，一个对象的修复需要多次数据的push操作。为了保证数据的完整一致性，

先把数据拷贝到PG的temp存储空间。当拷贝完成之后，再移动到该PG的实际空间中。

- 3) 开始拷贝数据：检查recovery_info.copy_subset, 也就是拷贝的区间。
- 4) 调用函数store->fiemap来确定有效数据的区间out_op->data_included 的值，通过store->read读取相应的数据到data里。
- 5) 设置PullOp的相关字段，并返回。

11.4 Backfill过程

当PG完成了Recovery过程之后，如果backfill_targets不为空，表明有需要Backfill过程的OSD，就需要启动Backfill的任务，来完成PG的全部修复。下面介绍Backfill过程相关的数据结构和具体处理过程。

11.4.1 相关数据结构

数据结构BackfillInterval用来记录每个peer上的Backfill过程。其字段说明如下：

- version：记录扫描对象列表时，当前PG对象更新的最新版本，一般为last_update，由于此时PG处于active状态，可能正在进行写操作。其用来检查从上次扫描到现在是否有对象写操作。如果有，完成写操作的对象在已扫描的对象列表中，进行Backfill操作时，该对象就需要更新为最新版本。
- objects：扫描到的准备进行Backfill操作的对象列表。
- begin：当前处理的对象。
- end：本次扫描对象的结束，用于作为下次扫描对象的开始：

```
struct BackfillInterval {
    // 一个
    peer 的
    backfill_interval 信息

    eversion_t version; // 扫描时的最新对象版本

    map<hobject_t, eversion_t, hobject_t::Comparator> objects;
    bool sort_bitwise;
    hobject_t begin;      // 当前处理的对象

    hobject_t end;        // 本次扫描对象的结束
```

}

11.4.2 Backfill的具体实现

函数recovery_backfill作为Backfill过程的核心函数，控制整个Backfill修复进程。其工作流程如下。

1) 初始设置。

在函数on_activate里设置了PG的属性值new_backfill为true，设置了last_backfill_started为earliest_backfill () 的值。该函数计算需要backfill的OSD中，peer_info信息里保存的last_backfill的最小值。

peer_backfill_info的map中保存各个需要Backfill的OSD所对应backfillInterval对象信息。首先初始化begin和end都为peer_info.last_backfill，由PG的Peering过程可知，在函数activate里，如果需要Backfill的OSD，设置该OSD的peer_info的last_backfill为hobject_t ()，也就是MIN对象。

backfills_in_flight保存了正在进行Backfill操作的对象，pending_backfill_updates保存了需要删除的对象。

2) 设置backfill_info.begin为last_backfill_started，调用函数update_range来更新需要进行Backfill操作的对象列表。

3) 根据各个peer_info的last_backfill对相应的backfillInterval信息进行

trim操作。根据last_backfill_started来更新backfill_info里相关字段。

- 4) 如果backfill_info.begin小于等于earliest_peer_backfill () , 说明需要继续扫描更多的对象, backfill_info重新设置, 这里特别注意的是, backfill_info的version字段也重新设置为 (0, 0) , 这会导致在随后调用的update_scan函数再调用scan_range函数来扫描对象。
- 5) 进行比较, 如果pbi.begin小于backfill_info.begin, 需要向各个OSD发送MOSDPGScan : : OP_SCAN_GET_DIGEST消息来获取该OSD目前拥有的对象列表。
- 6) 当获取所有OSD的对象列表后, 就对比当前主OSD的对象列表来进行修复。
- 7) check对象指针, 就是当前OSD中最小的需要进行Backfill操作的对象:
 - a) 检查check对象, 如果小于Backfill_info.begin, 就在各个需要Backfill操作的OSD上删除该对象, 加入到to_remove队列中。
 - b) 如果check对象大于或者等于backfill_info.begin, 检查拥有check对象的OSD, 如果版本不一致, 加入need_ver_targ中。如果版本相同, 就加入keep_ver_targs中。
 - c) 那些begin对象不是check对象的OSD, 如果pinfo.last_backfil小于backfill_info.begin, 那么, 该对象缺失, 加入missing_targs列表中。

d) 如果`pinfo.last_backfil`大于`backfill_info.begin`, 说明该OSD修复的进度已经超越当前的主OSD指示的修复进度, 加入`skip_targs`中。

8) 对于`keep_ver_targs`列表中的OSD, 不做任何操作。对于`need_ver_targs`和`missing_targs`中的OSD, 该对象需要加入到`to_push`中去修复。

9) 调用函数`send_remove_op`给OSD发送删除的消息来删除`to_remove`中的对象。

10) 调用函数`prep_backfill_object_push`把操作打包成`PushOp`, 调用函数`pgbackend->run_recovery_op`把请求发送出去。其流程和Recovery流程类似。

11) 最后用`new_last_backfill`更新各个OSD的`pg_info`的`last_backfill`值。如果`pinfo.last_backfill`为MAX, 说明backfill操作完成, 给该OSD发送`MOSDPGBackfill::OP_BACKFILL_FINISH`消息; 否则发送`MOSDPGBackfill::OP_BACKFILL_PROGRESS`来更新各个OSD上的`pg_info`的`last_backfill`字段。

下面举例说明。

例11-4 如图11-4所示, 该PG分布在5个OSD上 (也就是5个副本, 这里为了方便列出各种处理情况), 每一行上的对象列表都是相应OSD当前对应`backfillInterval`的扫描对象列表。`osd5`为主OSD, 是权威的对象列表,

其他OSD都对照主OSD上的对象列表来修复。

osd0	obj4(1,1) last_backfill peer_backfill_info[0].begin	obj5(1,4)	obj6(1,10)
osd1		obj5(1,3) last_backfill peer_backfill_info[1].begin	
osd2	obj4(1,1) last_backfill peer_backfill_info[2].begin	obj5(1,4)	
osd3		obj6(1,1) last_backfill peer_backfill_info[3].begin	obj7(1,8)
osd4		obj5(1,4) last_backfill peer_backfill_info[4].begin	obj6(1,10)
osd5 (主)		obj5(1,4) backfill_info.begin	obj6(1,10)
	last_backfill_started		

图11-4 backfill修复过程 (1)

下面举例来说明步骤7中的不同的修复方法：

- 1) 当前check对象指针为主OSD上保存的peer_backfill_info中begin的最小值。图中check对象应为obj4对象。
- 2) 比较check对象和主osd5上的backfill_info.begin对象，由于check小于obj5，所以obj4为多余的对象，所有拥有该check对象的OSD都必须删除该对象。故osd0和osd2上的obj4对象被删除，同时对应的begin指针前移。

osd0	obj4(1,1) last_backfill	obj5(1,4) peer_backfill_info[0].begin	obj6(1,10)
osd1		obj5(1,3) last_backfill peer_backfill_info[1].begin	
osd2	obj4(1,1) last_backfill	obj6(1,4) peer_backfill_info[2].begin	
osd3		obj6(1,1) last_backfill peer_backfill_info[3].begin	obj7(1,8)
osd4		obj5(1,4) last_backfill peer_backfill_info[4].begin	obj6(1,10)
osd5 (primary)		obj5(1,4) backfill_info.begin	obj6(1,10)
	last_backfill_started		

图11-5 backfill修复过程 (2)

3) 当前各个OSD的状态如图11-5所示：此时check对象为obj5， 比较check和backfill_info.begin的值：

a) 对于当前begin为check对象的osd0、 osd1、 osd4：

- 对于osd0和osd4， check对象和backfill_info.begin对象都是obj5， 且版本号都为 (1, 4)， 加入到keep_ver_targs列表中， 不需要修复。

- 对于osd1， 版本号不一致， 加入need_ver_targs列表中， 需要修复。

b) 对于当前begin不是check对象的osd2和osd3：

- 对于osd2， 其last_backfill小于backfill_info.begin， 显然对象obj5缺失， 加入missing_targs修复。

·对于osd3，其last_backfill大于backfill_info.begin，也就是说其已经修复到obj6了，obj5应该已经修复了，加入skip_targs跳过。

4) 步骤3处理完成后，设置last_backfill_started为当前的backfill_info.begin的值。backfill_info.begin指针前移，所有begin等于check对象的begin指针前移，重复以上步骤继续修复。

函数update_range调用函数scan_range更新BackfillInterval修复的对象列表，同时检查上次扫描对象列表中，如果有对象发生写操作，就更新该对象修复的版本。

具体实现步骤如下：

1) bi->version记录了扫描要修复的对象列表时PG最新更新的版本号，一般设置为last_update_applied或者info.last_update的值。初始化时，bi->version默认设置为(0, 0)，所以小于info.log_tail，就更新bi->version的设置，调用函数scan_range扫描对象。

2) 检查如果bi->version的值等于info.last_update，说明从上次扫描对象开始到当前时间，PG没有写操作，直接返回。

3) 如果bi->version的值小于info.last_update，说明PG有写操作，需要检查从bi->version到log_head这段日志中的对象：如果该对象有更新操作，修复时就修复最新的版本；如果该对象已经删除，就不需要修复，在修复队列中删除。

下面举例说明update_range的处理过程。

例11-5 update_range的处理过程

1) 日志记录如下图所示：

obj1(1,2) modify	Obj1(1,3) modify	obj2(1,4) modify	obj3(1,5) modify	obj4(1,6) modify			
---------------------	---------------------	---------------------	---------------------	---------------------	--	--	--

扫描列表为： bi->objects [obj1(1,3), obj2(1,4) obj3(1,5) obj4(1,6)]
(begin) (end)

BackfillInterval的扫描的对象列表：bi->begin为对象obj1 (1, 3) , bi->end为对象obj6 (1, 6) , 当前info.last_update为版本 (1, 6) , 所以bi->version设置为 (1, 6) 。由于本次扫描的对象列表不一定能修复完，只能等下次修复。

2) 日志记录如下图所示：

obj1(1,2) modify	Obj1(1,3) modify	obj2(1,4) modify	obj3(1,5) modify	obj4(1,6) modify	obj3(1,7) modify	obj4(1,8) delete	
---------------------	---------------------	---------------------	---------------------	---------------------	---------------------	---------------------	--

扫描列表为： bi->objects [obj1(1,3), obj2(1,4) obj3(1,5) obj4(1,6)]
(begin) (end)

第二次进入函数recover_backfill, 此时begin对象指向了obj2对象。说明上次只完成了对象obj1的修复。继续修复时，期间有对象发生更新操作：

a) 对象obj3有写操作，版本更新为 (1, 7) 。此时对象列表中要修复的对象obj3版本 (1, 5) , 需要更新为版本 (1, 7) 的值。

b) 对象obj4发送删除操作，不需要修复了，所以需要从对象列表中删除。

综上所述可知，Ceph的Backfill过程是扫描OSD上该PG的所有对象列表，和主OSD做对比，修复不存在的或者版本不一致的对象，同时删除多余的对像。

11.5 本章小结

本章介绍了Ceph的数据修复的过程，有两个过程：Recovery过程和Backfill过程。Recovery过程根据missing记录，先完成主副本的修复，然后完成从副本的修复。对于不能通过日志修复的OSD，Backfill过程通过扫描各个部分上的对象来全量修复。整个Ceph的数据修复过程比较清晰，比较复杂的副本可能就是涉及快照对象的修复处理。

目前这部分代码是Ceph最核心的代码，除非必要，都不会轻易修改。目前社区也提出了修复时的一种优化方法。就是在日志里记录修改的对象范围，这样在Recovery过程中不必拷贝整个对象来修复，只修复修改过的对象对应的范围即可，这样在某些情况下可以减少修复的数据量。

第12章 Ceph一致性检查

本章介绍Ceph的一致性检查工具Scrub机制。首先介绍数据校验的基本知识，其次介绍Scrub的基本概念，然后介绍Scrub的调度机制，最后介绍Scrub具体实现的源代码分析。

12.1 端到端的数据校验

在存储系统中可能会发生数据静默损坏（Silent Data Corruption），这种情况的发生大多是由于数据的某一位发生异常反转（Bit Error Rate）。

图12-1是一般存储系统的协议栈，数据损坏的情况会发生在系统的所有模块中：

- 硬件错误，例如内存、CPU、网卡等。
- 数据传输过程中的信噪干扰，例如SATA、FC等协议。
- 固件bug，例如RAID控制器、磁盘控制、网卡等。
- 软件bug，例如操作系统内核的bug，本地文件系统的bug，SCSI软件模块的bug等。

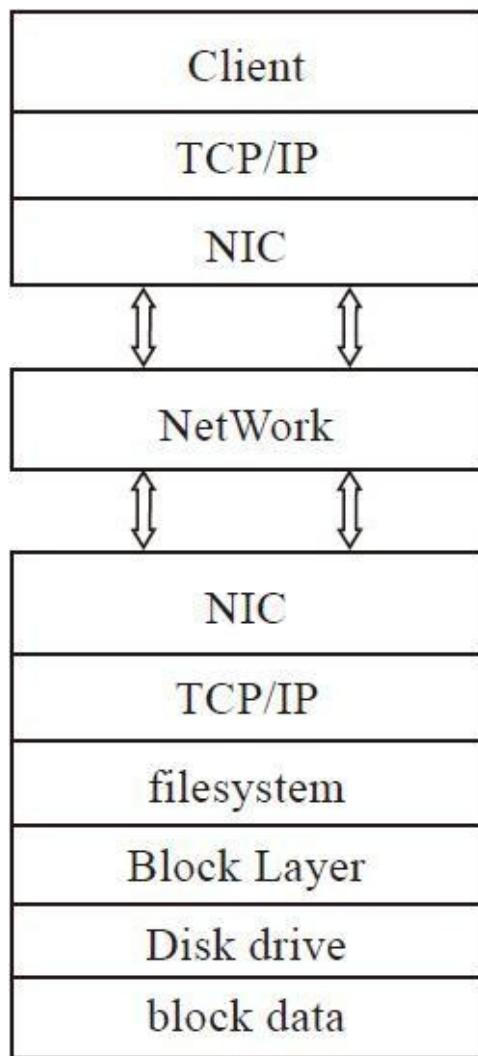


图12-1 一般存储系统的协议栈

在传统的高端磁盘阵列中，一般采用端到端的数据校验实现数据的一致性。所谓端到端的数据校验，指客户端（应用层）在写入数据时，为每个数据块都计算一个CRC校验信息，并将这个校验信息和数据块发送至磁盘（Disk）。磁盘在接收到数据包之后，会重新计算校验信息，并和接收到的校验信息作对比。如果不一致，那么就认为在整个I/O路径上存在错误，返回I/O操作失败；如果校验成功，就把数据校验信息和数据保存在磁盘上。同样，在数据读取时，客户端再获取数据块和从磁盘读取校验信息

时，也需要再次检查是否一致。

通过这种方法，应用层可以很明确地知道一次I/O请求的数据是否一致。如果操作成功，那么磁盘上的数据必然是正确的。

这种方式在不影响I/O性能或者影响比较小的情况下，可以提高数据读写的完整性。但这种方式也有一些缺点：

- 无法解决目的地址错误导致的数据损坏问题。

- 端到端的解决方案需要在整个I/O路径上附加校验信息。现在的I/O协议栈涉及的模块比较多，每个模块都附加这种校验信息实现起来比较困难。

由于这种实现方式对Ceph的I/O性能影响比较大，所以Ceph并没有实现端到端的数据校验，而是实现Ceph Scrub机制，采用一种通过在后台扫描的方案来解决Ceph的一致性检查。

12.2 Scrub概念介绍

Ceph在内部实现了数据一致性检查的一个工具：Ceph Scrub。其原理为：通过对比各个对象副本的数据和元数据，完成副本一致性检查。

这种方法的优点是在后台可以发现由于磁盘损坏而导致的数据不一致现象。缺点是发现的时机往往比较滞后。

Scrub按照扫描的内容分为两种方式：

- 一种叫Scrub，它仅仅通过对比对象各副本的元数据，来检查数据的一致性。由于只检查元数据，读取数据量和计算量都比较小，是一种比较轻度的检查。

- 另一种叫deep-scrub，它进一步检查对象的数据内容是否一致，实现了深度扫描，几乎要扫描磁盘上的所有数据并计算crc32校验值，因此比较耗时，占用系统资源更多。

Scrub按照扫描的方式分为两种：

- 在线扫描：不影响系统正常的业务。

- 离线扫描：需要整个系统暂停或者冻结。

Ceph的Scrub功能实现了在线检查，即不中断系统当前读写请求，客户端可以继续完成读写访问。整个系统并不会暂停，但是后台正在进行Scrub

的对象要被锁定暂时阻止访问，直到该对象完成Scrub操作后才能解锁允许访问。

12.3 Scrub的调度

Scrub的调度解决了一个PG何时启动Scrub扫描机制。主要有以下方式：

- 手动立即启动执行扫描。
- 在后台设置一定的时间间隔，按照间隔的时间来启动。比如默认时间为一天执行一次。
- 设置启动的时间段。一般设定一个系统负载比较轻的时间段来执行Scrub操作。

12.3.1 相关数据结构

在类PG里有下列与Scrub相关的数据结构：

```
mutex sched_scrub_lock;           //Scrub相关变量的保护锁

int scrubs_pending;               //资源预约已经成功，正等待

Scrub的

PG
int scrubs_active;               //正在进行

Scrub的

PG
set<ScrubJob> sched_scrub_pg;  //PG对应的所有

ScrubJob列表
```

结构体ScrubJob封装了一个PG的Scrub任务相关的参数：

```
struct ScrubJob {
    spg_t pgid;                  // Scrub对应的

    PG
    utime_t sched_time;          // Scrub任务的调度时间，如果当前负载比较高，或者当前的时间不在

    设定的

    Scrub工作时间段内，就会延迟调度

    utime_t deadline;            // 调度时间的上限，过了该时间必须进行

    Scrub操作，而不受系统负载和

    Scrub时间段的限制

};
```

12.3.2 Scrub的调度实现

在OSD的初始化函数OSD::init中，注册了一个定时任务：

```
tick_timer_without_osd_lock.add_event_after(cct->conf->osd_heartbeat_
interval, new C_Tick_WithoutOSDLock(this));
```

该定时任务每隔osd_heartbeat_interval时间段（默认为1秒），就会触发定时器的回调函数OSD::tick_without_osd_lock()，处理过程的函数调用关系图如图12-2所示。

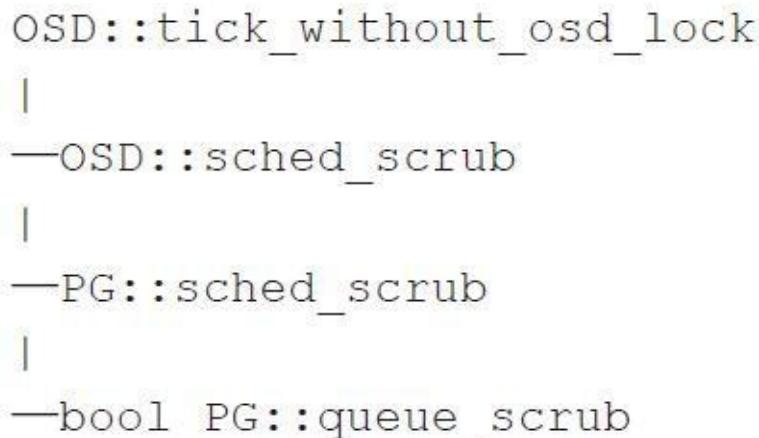


图12-2 Scrub的触发函数调用关系

以上函数实现了PG的Scrub调度工作。下面将介绍处理过程中关键的两个函数OSD::sched_scrub和PG::sched_scrub函数的实现。

1.OSD::sched_scrub函数

本函数用于控制一个PG的Scrub过程启动时机，具体过程如下：

- 1) 调用函数can_inc_scrubs_pending来检查是否有配额允许PG开始启动Scrub操作。变量scrubs_pending记录了已经完成资源预约正在等待Scrub的PG的数量，变量scrubs_active记录了正在进行Scrub检查的PG数量，其二者的数量之和不能超过系统配置参数cct->_conf->osd_max_scrubs的值。该值设置了同时允许Scrub的最大的PG数量。
- 2) 调用函数scrub_time_permit检查是否在允许的时间段内。如果cct->_conf->osd_scrub_begin_hour大于cct->_conf->osd_scrub_end_hour，当前时间必须在二者设定的时间范围之间才允许。如果cct->_conf->osd_scrub_begin_hour小于等于cct->_conf->osd_scrub_end_hour，当前时间在二者设定的时间范围之外才允许。
- 3) 调用函数scrub_load_below_threshold检查当前的系统负载是否允许。函数getloadavg获取最近1、5、15分钟的系统负载。
 - a) 如果最近1分钟的负载小于cct->_conf->osd_scrub_load_threshold的设定值，就允许执行。
 - b) 如果最近1分钟的负载小于daily_loadavg的值，并且最近1分钟负载小于最近15分钟的负载，就允许执行。
- 4) 获取第一个等待执行Scrub操作的ScrubJob列表，如果它的scrub.sched_time大于当前时间now值，说明时间不到，就跳过该PG先执行

下一个任务。

5) 获取该PG对应的PG对象，如果该PG的pgbackend支持Scrub，并且处于active状态：

- a) 如果scrub.deadline小于now值，也就是已经超过最后的期限，必须启动Scrub操作。
- b) 或者此时time_permit并且load_is_low，也就是时间和负载都允许。

在上述两种情况下，调用函数pg->sched_scrub () 来执行Scrub操作。

2.PG::sched_scrub函数

本函数实现了对执行Scrub任务时相关参数的设置，并完成了所需资源的预约。其处理过程如下：

- 1) 首先检查PG的状态，必须是主OSD，并且处于active和clean状态，并且没有正在进行Scrub操作。
- 2) 设置deep_scrub_interval值：如果该PG所在的pool选项中没有设置该值，就设置为系统配置参数cct->_conf->osd_deep_scrub_interval的值。
- 3) 检查是否启动deep_scrub，如果当前时间大于info.history.last_deep_scrub_stamp与deep_scrub_interval之和，就启动deep_scrub操作。

- 4) 如果scrubber.must_scrub的值为true，为用户手动强制启动deep_scrub操作。如果该值为false，则需系统自动以一定的概率来启动deep_scrub操作，具体实现就是：自动产生一个随机数，如果该随机数小于cct->_conf->osd_deep_scrub_randomize_ratio，就启动deep_scrub操作。
- 5) 决定最终是否启动deep_scrub，在步骤3) 和4) 中只要有一个设置好，就启动deep_scrub操作。
- 6) 如果osdmap或者pool中带有不支持deep_scrub的标记，就设置time_for_deep为false，不启动deep_scrub操作。
- 7) 如果osdmap或者pool中带有不支持Scrub的标记，并且也没有启动deep-scrub操作则返回并退出。
- 8) 如果cct->_conf->osd_scrub_auto_repair设置了自动修复，并且pgbackend也支持，而且是deep_scrub操作，则进行如下判断过程：
 - a) 如果用户设置了must_repair，或者must_scrub，或者must_deep_scrub，说明这次Scrub操作是用户触发的，系统尊重用户的選擇，不会自动设置scrubber.auto_repair的值为true。
 - b) 否则，系统就设置scrubber.auto_repair的值为true来自动进行修复。
- 9) Scrub过程和Recovery过程类似，都需要耗费系统大量资源，需要去PG所在的OSD上预约资源。如果scrubber.reserved的值为false，还没有完成资源的预约，需进行如下操作：

- a) 把自己加入到scrubber.reserved_peers中。
- b) 调用函数scrub_reserve_replicas， 向OSD发送CEPH OSD OP SCRUB RESERVE消息来预约资源。
- c) 如果scrubber.reserved_peers.size () 等于acting.size () ， 表明所有的从OSD资源预约成功， 把PG设置为PG STATE DEEP SCRUB状态。 调用函数queue_scrub把该PG加入到工作队列op_wq中， 触发Scrub任务开始执行。

12.4 Scrub的执行

Scrub的具体执行过程大致如下：通过比较对象各个OSD上副本的元数据和数据，来完成元数据和数据的校验。其核心处理流程在函数PG \cdot chunky_scrub里控制完成。

12.4.1 相关数据结构

Scrub操作相关的主要数据结构有两个，一个是Scrubber控制结构，它相当于是一次Scrub操作的上下文，控制一次PG的操作过程。另一个是ScurbMap保存需要比较的对象各个副本的元数据和数据的摘要信息。

1.Scrubber

结构体Scrubber用来控制一个PG的Scrub的过程：

```
struct Scrubber {
    // 元数据

    set<pg_shard_t> reserved_peers;           // 资源预约的
    shard
    bool reserved, reserve_failed;              // 是否预约资源，预约资源是否失败

    epoch_t epoch_start;                        // 开始
    Scrub 操作的

    epoch
    // common to both scrubs
    bool active;                             // Scrub是否开始

    bool queue_snap_trim;
    // 当

    PG 有

    snap_trim 操作时，如果检查
    Scrubber 处于
    active 状态，说明正在进行
    Scrub 操作，那么
    snap_trim 操作暂停，设置
```

queue_snap_trim的值为

true。当

PG完成

Scrub任务后，如果

queue_snap_trim的值为

true，就把

PG添加到相应的工作队列里，继续完成

snap_trim操作。

```
int waiting_on;                                //等待的副本计数

set<pg_shard_t> waiting_on_whom;              //等待的副本

int shallow_errors;                            //轻度扫描错误数

int deep_errors;                             //深度扫描错误数

int fixed;                                  //已经修复的对象数

ScrubMap primary_scrubmap;                  //主副本的

ScrubMap
map<pg_shard_t, ScrubMap> received_maps; //接收到的从副本的

ScrubMap
OpRequestRef active_rep_scrub;
utime_t scrub_reg_stamp;           // stamp we registered for
// flags to indicate explicitly requested scrubs (by admin)
bool must_scrub, must_deep_scrub, must_repair;
    bool auto_repair;             //是否自动修复

// Maps from objects with errors to missing/inconsistent peers
map<hobject_t, set<pg_shard_t>, hobject_t::BitwiseComparator> missing;
//扫描出的缺失对象

map<hobject_t, set<pg_shard_t>, hobject_t::BitwiseComparator> inconsistent;
//扫描出的不一致对象

// Map from object with errors to good peers
map<hobject_t, list<pair<ScrubMap::object, pg_shard_t> >, hobject_
t::BitwiseComparator> authoritative;
//如果所有副本对象中有不一致的对象，

authoritative记录了正确对象所在的
```

```
OSD
    // digest updates which we are waiting on
    int num_digest_updates_pending;      //等待更新

digest的对数目

    hobject_t start, end;                //扫描对象列表的开始和结尾

    eversion_t subset_last_update;       //扫描对象列表中最新的版本号

    bool deep;                          //是否为深度扫描

    uint32_t seed;                      //计算

crc32校验码的种子

    list<Context*> callbacks;
} scrubber;
```

2.ScrubMap

数据结构ScrubMap保存准备校验的对象以及相应的校验信息：

```
struct ScrubMap {
    struct object {
        map<string, bufferptr> attrs;          //对象的属性

        set<snapid_t> snapcolls;               //该对象所有的

snap序号

        uint64_t size;                         //对象的

size
        __u32 omap_digest;                    //omap的

crc32c校验码

        __u32 digest;                        //对象数据的

crc32c校验码

        uint32_t nlinks;                     //snap对象（
```

clone对象) 对应的

snap的数量

```
bool negative:1;  
bool digest_present:1; //是否计算了数据的校验码标志
```

```
bool omap_digest_present:1; //是否有
```

omap的校验码标志

```
bool read_error:1; //读对象的数据出错标志
```

```
bool stat_error:1; //调用
```

stat获取对象的元数据出错标志

```
};
```

```
....
```

```
map<hobject_t, object, hobject_t::BitwiseComparator> objects;  
//需要校验的对象 (
```

hobject) →校验信息 (

Object) 的映射

```
eversion_t valid_through;  
eversion_t incr_since;
```

```
....
```

```
};
```

内部类object用来保存对象需要校验的信息，包括以下5个方面：

- 对象的大小 (size)

- 对象的属性 (attrs)

- 对象omap的校验码 (digest)

- 对象数据的校验码（digest）
- 对象所有clone对象的快照序号

12.4.2 Scrub的控制流程

Scrub的任务由OSD的工作队列OpWq来完成，调用对应的处理函数pg->scrub (handle) 来执行。

PG::scrub函数最终调用PG::chunky_scrub函数来实现。PG::chunky_scrub函数控制了Scrub操作状态转换和核心处理过程。

具体过程分析如下所示：

1) Scurbber的初始状态为PG::Scrubber::INACTIVE，该状态的处理如下：

- a) 设置scrubber.epoch_start的值为info.history.same_interval_since。
- b) 设置scrubber.active的值为true。
- c) 设置状态scrubber.state的值为PG::Scrubber::NEW_CHUNK。
- d) 根据peer_features，设置scrubber.seed的类型，这个seed是计算crc32的初始化哈希值。

2) PG::Scrubber::NEW_CHUNK状态的处理如下：

- a) 调用get_pgbbackend () ->objects_list_partial函数从start对象开始扫描一组对象，一次扫描的对象数量在如下两个配置参数之间：cct->conf-

>osd_scrub_chunk_min (默认值为5) 和cct->_conf->osd_scrub_chunk_max (默认值为25) 。

b) 计算出对象的边界。相同的对象具有相同的哈希值。从列表后面开始查找哈希值不同的对象，从该地方划界。这样做的目的是把一个对象的所有相关对象（快照对象、回滚对象）划分在一次扫描校验过程中。

c) 调用函数_range_available_for_scrub检查列表中的对象，如果有被阻塞的对象，就设置done的值为true，退出PG本次的Scrub过程。

d) 根据pg_log计算start到end区间对象最大的更新版本号，这个最新版本号设置在scrubber_subset_last_update里。

e) 调用函数_request_scrub_map向所有副本发送消息，获取相应ScrubMap的校验信息。

f) 设置状态为PG::Scrubber::WAIT_PUSHES。

3) PG::Scrubber::WAIT_PUSHES状态的处理如下：

a) 如果active_pushes的值为0，设置状态为PG::Scrubber::WAIT_LAST_UPDATE，进入下一个状态处理。

b) 如果active_pushes不为0，说明该PG正在进行Recovery操作。设置done的值为true，直接结束。在进入chunky_scrub时，PG应该处于CLEAN状态，不可能有Recovery操作，这里的Recovery操作可能是上次进行

chunky_scrub操作后的修复操作。

4) PG_：Scrubber_：WAIT_LAST_UPDATE状态的处理如下：

a) 如果last_update_applied的值小于scrubber.subset_last_update的值，说明虽然已经把操作写入了日志，但是还没有应用到对象中。由于Scrub操作后面的步骤有对象的读操作，所以需要等待日志应用完成。设置done的值为true结束本次PG的Scrub过程。

b) 否则就设置状态为PG_：Scrubber_：BUILD_MAP。

5) PG_：Scrubber_：BUILD_MAP状态的处理如下：

a) 调用函数build_scrub_map_chunk构造主OSD上对象的ScrubMap结构。

b) 如果构造成功，计数scrubber.waiting_on的值减1，并从队列中删除scrubber.waiting_on_whom，则相应的状态设置为PG_：Scrubber_：WAIT_REPLICAS。

6) PG_：Scrubber_：WAIT_REPLICAS状态的处理如下：

a) 如果scrubber.waiting_on不为零，说明有replica请求没有回答，设置done的值为true，退出并等待。

b) 否则，进入PG_：Scrubber_：COMPARE_MAPS状态。

7) PG : : Scrubber : : COMPARE_MAPS状态的处理如下：

- a) 调用函数scrub_compare_maps比较各副本的校验信息。
- b) 将参数scrubber.start的值更新为scrubber.end。
- c) 调用函数requeue_ops, 把由于Scrub而阻塞的读写操作重新加入操作队列里执行。
- d) 状态设置为PG : : Scrubber : : WAIT_DIGEST_UPDATES。

8) PG : : Scrubber : : WAIT_DIGEST_UPDATES状态的处理如下：

- a) 如果有scrubber.num_digest_updates_pending等待, 等待更新数据的digest或者omap的digest。
- b) 如果scrubber.end小于hobject_t : : get_max (), 本PG还有没有Scrub操作完成的对象, 设置状态scrubber.state为PG : : Scrubber : : NEW_CHUNK, 继续把PG加入到osd->scrub_wq中。
- c) 否则, 设置状态为PG : : Scrubber : : FINISH值。

9) PG : : Scrubber : : FINISH状态的处理如下：

- a) 调用函数scrub_finish来设置相关的统计信息, 并触发修复不一致的对象。
- b) 设置状态为PG : : Scrubber : : INACTIVE。

12.4.3 构建ScrubMap

构建ScrubMap有多个函数实现，下面分别介绍。

1. build_scrub_map_chunk

函数build_scrub_map_chunk用于构建从start到end之间的所有对象的校验信息并保存在ScrubMap结构中。

```
int PG::build_scrub_map_chunk(
    ScrubMap &map,
    hobject_t start, hobject_t end,
    bool deep, uint32_t seed,
    ThreadPool::TPHandle &handle)
```

处理过程分析如下：

- 1) 设置map.valid_through的值为info.last_update。
- 2) 调用get_pgbbackend () ->objects_list_range函数列出所有的start和end范围内的对象，ls队列存放head和snap对象，rollback_obs队列存放用来回滚的ghobject_t对象。
- 3) 调用函数get_pgbbackend () ->be_scan_list扫描对象，构建ScrubMap结构。
- 4) 调用函数_scan_rollback_obs来检查回滚对象：如果对象的generation小于last_rollback_info_trimmed_to_applied值，就删除该对象。

5) 调用`_scan_snaps`来修复SnapMapper里保存的snap信息。

2.`_scan_snaps`

函数`_scan_snaps`扫描head对象保存的snap信息和SnapMapper中保存的该对象的snap信息是否一致。它以前者保存的对象snap信息为准，修复snapMapper中保存的对象snap信息。

```
void PG::_scan_snaps(ScrubMap &smap)
```

具体实现过程为：对于ScrubMap里的每一个对象循环做如下操作：

- 1) 如果对象的`hoid.snap`的值小于`CEPH_MAXSNAP`的值，那么该对象是snap对象，从`o.attrs[OI_ATTR]`里获取`object_info_t`信息。
- 2) 检查`oi`的`snaps`。如果`oi.snaps.empty ()`为0，设置`nlinks`等于1；如果`oi.snaps.size ()`为1，设置`nlinks`等于2；否则设置`nlinks`等于3。
- 3) 从`oi`获取`oi_snaps`，从`snap_mapper`获取`cur_snaps`，比较两个snap信息，以`oi`的信息为准：
 - a) 如果函数`snap_mapper.get_snaps (hoid, &cur_snaps)`的结果为`ENOENT`，就把信息该添加到`snap_mapper`里。
 - b) 如果信息不一致，先删除`snap_mapper`里不一致的对象，然后把该对象的snap信息添加到`snap_mapper`里。

3.be_scan_list

函数be_scan_list用于构建ScrubMap中对象的校验信息：

```
void PGBackend::be_scan_list(
    ScrubMap &map,
    const vector<hobject_t> &ls,
    bool deep,
    uint32_t seed,
    ThreadPool::TPHandle &handle)
```

具体处理过程就是循环扫描ls向量中的对象：

- 1) 调用store->stat获取对象的stat信息：
 - a) 如果获取成功，设置o.size的值等于st.st_size，并调用store->getattrs把对象的attr信息保存在o.attrs里。
 - b) 如果stat返回结果r为-ENOENT，就直接跳过该对象（该对象在本OSD上可能缺失，在后面比较结果时会检查出来）。
 - c) 如果stat返回结果r为-EIO，就设置o.stat_error的值为true。
- 2) 如果deep的值为true，调用函数be_deep_scrub进行深度扫描，获取对象的omap和data的digest信息。

4.be_deep_scrub

函数be_deep_scrub实现对象的深度扫描：

```
void ReplicatedBackend::be_deep_scrub(
```

```
const hobject_t &poid,           //深度扫描的对象  
uint32_t seed,                 //crc32的种子  
ScrubMap::object &o,           //保存对应的校验信息  
ThreadPool::TPHandle &handle)
```

实现过程分析如下：

- 1) 设置data和omap的bufferhash的初始值都为seed。
- 2) 循环调用函数store->read读取对象的数据，每次读取长度为配置参数cct->_conf->osd_deep_scrub_stride (512k)，并通过bufferhash计数crc32校验值。如果中间出错 (r==EIO)，就设置o.read_error的值为true。最后设置o.digest为计算出crc32的校验值，设置o.digest_present的值为true。
- 3) 调用函数store->omap_get_header获取header，迭代获取对象的omap的key-value值。计算header和key-value的digest信息，并设置在o.omap_digest中，标记o.omap_digest_present的值为true。

综上可知，通过函数be_scan_list来获取对象的元数据信息，通过be_deep_scrub函数获取对象的数据和omap的digest信息保存在ScrubMap结构中。

12.4.4 从副本处理

当从副本接收到主副本发送来的MOSDRepScrub类型消息，用于获取对象的校验信息时，就调用函数replica_scrub来完成。

函数replica_scrub具体实现如下：

- 1) 首先确保scrubber.active_rep_scrub不为空。
- 2) 检查如果msg->map_epoch的值小于info.history.same_interval_since的值就直接返回。在这里从副本直接丢弃掉过时的MOSDRepScrub请求。
- 3) 如果last_update_applied的值小于msg->scrub_to的值，也就是从副本上完成日志应用的操作落后主副本scrub操作的版本，必须等待它们一致。把当前的op操作保存在scrubber.active_rep_scrub中等待。
- 4) 如果active_pushes大于0，表明有Recovery操作正在进行，同样把当前的op操作保存在scrubber.active_rep_scrub中等待。
- 5) 否则就调用函数build_scrub_map_chunk来构建ScrubMap，并发送给主副本。

当等待的本地操作应用完成之后，在函数ReplicatedPG :: op_applied检查，如果scrubber.active_rep_scrub不为空，并且该操作的版本等于msg->scrub_to，就会把保存的op操作重新放入osd->op_wq请求队列，继续完成

该请求。

12.4.5 副本对比

当对象的主副本和从副本都完成了校验信息的构建，并保存在相应的结构ScrubMap中，下一步就是对比各个副本的校验信息来完成一致性检查。首先通过对对象自身的信息来选出一个权威的对象，然后用权威对象和其他对象做比较来检验。下面介绍用于比较的函数。

1.scrub_compare_maps

函数scrub_compare_maps实现比较不同的副本信息是否一致，处理过程如下：

- 1) 首先确保acting.size () 大于1，如果该PG只有一个OSD，则无法比较。
- 2) 把actingbackfill对应OSD的ScrubMap放置到maps中。
- 3) 调用函数be_compare_scrubmaps来比较各个副本的对象，并把对象完整的副本所在shard保存在authoritative中。
- 4) 调用_scrub函数继续比较snap之间对象的一致性。

2.be_compare_scrubmaps

函数be_compare_scrubmaps用来比较对象各个副本的一致性，其具体

处理过程分析如下：

- 1) 首先构建master set，也就是所有副本OSD上对象的并集。
- 2) 对master set中的每一个对象进行如下操作：
 - a) 调用函数be_select_auth_object选择出一个具有权威对象的副本auth，如果没有选择出权威对象，变量shallow_errors加一来记录这种错误。
 - b) 调用函数be_compare_scrub_objects，比较各个shard上的对象和权威对象：分别对data的digest、omap的omap_digest和属性attrs进行对比：
 - 如果结果为clean，表明该对象和权威对象的各项比较完全一致，就把该shard添加到auth_list列表中。
 - 如果结果不为clean，就把该对象添加到cur_inconsistent列表中，分别统计shallow_errors和deep_errors的值。
 - 如果该对象在该shard上不存在，添加到cur_missing列表中，统计shallow_errors值。
 - c) 检查该对象所有的比较结果：如果cur_missing不为空，就添加到missing队列；如果有cur_inconsistent对象，就添加到inconsistent对象里；如果该对象有不完整的副本，就把没有问题的记录放在authoritative中。
 - d) 如果权威对象object_info里记录的数据的digest和omap的omap_digest

和实际扫描出数据计算的结果不一致， update的模式就设置为FORCE， 强制修复。如果object_info里没有data的digest和omap的digest， 修复的模式update设置为MAYBE。

e) 最后检查， 如果update的模式为FORCE， 或者该对象存在的时间age大于配置参数g_conf->osd_deep_scrub_update_digest_min_age的值， 就加入missing_digest列表中。

[3.be_select_auth_object](#)

函数be_select_auth_object用于在各个OSD上的副本对象中， 选择出一个权威的对象：auth_obj对象。其原理是根据自身所带的冗余信息来验证自己是否完整， 具体流程如下：

- 1) 首先确认该对象的read_error和stat_error没有设置， 也就在获取对象的数据和元数据的过程中没有出错， 否则就直接跳过。
- 2) 确认获取的属性OI_ATTR值不为空， 并将数据结构object_info_t正确解码， 就设置当前的对象为auth_obj对象。
- 3) 验证保存在object_info_t中的size值和扫描获取对象的size值是否一致， 如果不一致， 就继续查找一个更好的auth_obj对象。
- 4) 如果是replicated类型的PG， 验证在object_info_t里保存的data和omap的digest值是否和扫描过程计算出的值一致。如果不一致， 就继续查找一个更好的auth_obj对象。

5) 如上述都一致，直接终止循环，当前已经找到满意的auth_obj对象。

由上述的选择过程可知，选中一个权威对象的条件如下：

·步骤1），2）两点条件是基础，对象的数据和属性能正确读取。

·步骤3），4）是利用了object_info_t中保存的对象size，以及omap和data的digest的冗余信息。用这些信息和对象扫描读取的数据计算出的信息比较来验证。

4._scrub

函数_scrub检查对象和快照对象之间的一致性：

```
void ReplicatedPG::_scrub(ScrubMap &scrubmap,
                           const map<hobject_t, pair<uint32_t, uint32_t> > &missing_digest)
```

如果pool有cache pool层，那么就允许拷贝对象有不一致的状态，因为有些对象可能还存在于cache pool中没有被刷回。通过函数pool.info.allow_incomplete_clones () 来确定上述情况。

其实代码比较复杂，下面通过举例来说明其实现基本过程。

例12-1 _scrub函数实现过程举例，如表12-1所示。

表12-1 _scrub函数实现过程

步骤	对象列表	希望的对象
1	obj1 snap1	head/snapdir, unexpected obj1 snap 1
2	obj2 head	head/snapdir, head ok [snapset clones 6 4 2 1]
3	obj2 snap7	obj2 snap 6, unexpected obj2 snap 7
4	obj2 snap6	obj2 snap 6, match
5	obj2 snap4	obj2 snap 4, match
6	obj3 head	obj2 snap 2 (expected), obj2 snap 1 (expected), head ok [snapset clones 3 1]
7	obj3 snap3	obj3 snap 3 match
8	obj3 snap1	obj3 snap 1 match
9	obj4 snapdir	head/snapdir, snapdir ok [snapset clones 4]
10	EOL	obj4 snap 4, (expected)

_scrb实现过程说明如下：

- 1) 对象obj1 snap1为快照对象，就应该有head或者snapdir对象。但是该快照对象没有对应的head或者snapdir对象，那么该对象被标记为unexpected对象。
- 2) 对象obj2 head是一个head对象，这是预期的对象。通过head对象，获取snapset的clones列表为[6, 4, 2, 1]。
- 3) 检查对象obj2 snap7并没有在对象obj2的snapset的clones列表中，为异常对象。
- 4) 对象obj2的快照对象snap6，在snapset的clones列表中。
- 5) 对象obj2的快照对象snap4，在snapset的clones列表中。

6) 当遇到obj3 head对象时，预期的对象obj2中应该还有快照对象snap2和snap1为缺失的对象。继续获取snap3的snapset的clones值为列表[3, 1]。

7) 对象obj3的快照对象snap3和预期对象一致

8) 对象obj3的快照对象snap1和预期对象一致。

9) 对象obj4的snapdir对象，符合预期。获取该对象的snapset的clones值为列表[4]。

10) 扫描的对象列表结束了，但是预期对象为obj4的快照对象snap4，对象obj4 snap4缺失。

目前对于excepted的对象和unexcepted的对象，都只是在日志中标记出来，并不做进一步的处理。

最后对于那些其他都正确但对象的digest不正确的数据对象，也就是missing_digest中需要更新digest的对象，发送更新digest请求。

12.4.6 结束Scrub过程

scrub_finish函数用于结束Scrub过程，其处理过程如下：

- 1) 设置了相关PG的状态和统计信息。
- 2) 调用函数scrub_process_inconsistent用于修复scrubber里标记的missing和inconsistent对象，其最终调用repair_object函数。它只是在peer_missing中标记对象缺失。
- 3) 最后触发DoRecovery事件发送给PG的状态机，发起实际的对象修复操作。

12.5 本章小结

本章介绍了Scrub的基本原理，及Scrub过程的调度机制，然后介绍了校验信息的构建和比较验证的具体流程。

最后，总结一下Ceph一致性检查Scrub功能实现的关键点：

- 如果做Scrub操作检查的对象范围start和end之间有对象正在进行写操作，就退出本次Scrub进程；如果已经开始启动了Scrub，那么在start和end之间的对象写操作需要等待Scrub操作的完成。
- 检查就是对比主从副本的元数据（size、 attrs、 omap）和数据是否一致。权威对象的选择是根据对象自己保持的信息（object info）和读取对象的信息是否一致来选举出。然后用权威对象对比其他对象副本来检查。

第13章 Ceph自动分层存储

本章介绍Ceph的Cache Tier模块，它实现了的自动分层存储技术。本章首先介绍自动分层存储的概念和原理，其次介绍Ceph的自动分层存储Cache Tier的读写模式，最后对Cache Tier的源代码进行详细的分析。

13.1 自动分层存储技术

分层存储在“信息生命周期管理”的基础上对数据进一步分层。这个概念的提出基于这样一个事实：在数据的不同生命周期里，其访问的频度是截然不同的，即使是处于同一生命周期的不同类型的数据，访问频率也会不同。

自动分层存储技术目标在于：把用户访问频率高的数据放置在高性能、小容量的存储介质中，把大量低频访问的数据放置在大容量、性能相对较低的存储介质中。它为用户提供的价值在于：在提高热点数据存储性能的同时，降低了存储成本。首先，数据可以自由安全地迁移到更低层的存储介质中，这样可以节约存储成本；其次热点数据可以自动的从低层迁移到高层存储层，提高访问热点数据的性能。

自动分层存储技术实现的核心点在于：

- 数据访问行为的追踪、统计与分析：持续追踪与统计每个数据块的存取频率，并通过定期分析，识别出存取频率高的“热”区块，与存取频率低的“冷”区块。

- 数据迁移：以存取频率为基础，定期执行数据迁移，将热点数据块迁移到高速存储层，把较不活跃的冷数据块迁移到低速存储层。

传统的磁盘阵列存储产品的自动分层存储技术中，数据访问行为统计

和分析的粒度是以数据块为单位的。在Ceph中，数据访问行为统计、分析以及数据在各存储层之间迁移的粒度是以对象（默认为4MB）为基本单位的。

13.2 Ceph分层存储架构和原理

在Ceph里，Cache Tier模块在pool层设置。可以设置一个pool为另一个pool的cache层。在这里，做缓存层的pool称为cache pool（或者cache tier、hot pool等）；相对低性能的存储层称为data pool（或者base pool、cold pool等）。

图13-1是Cache Tier的存储架构图。在client层的librados对于Cache Tier是透明的，它对Cache Tier是无感知的。Objecter实现了Cache Tier相关的逻辑处理。Cache Tier层为高速存储层，热点数据都保存在Cache Tier层，Data Tier层为慢速设备存储层，所有数据最终都会保存在Data Tier层。

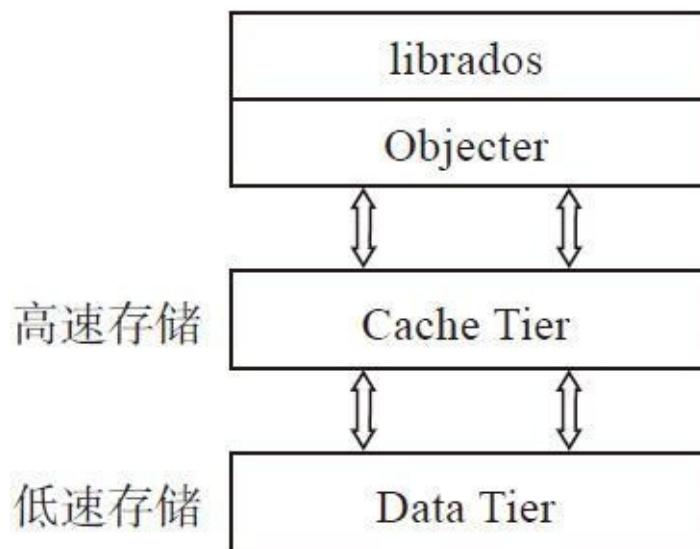


图13-1 Cache Tier存储架构图

13.3 Cache Tier的模式

在结构cache_mode_t的枚举类型里， 定义cache pool有几种模式：

·write back

·read forward

·read proxy

·write proxy

·read only

下面将详细介绍各种模式的不同应用场景。

1.write back模式

在write back模式下， 读写请求都直接发给cache pool， 这种模式适合于大量修改数据的应用场景（例如图片视频编辑、 事务处理OLTP类应用）。

在write back模式下， 读操作时对象在缓存层不存在（cache miss）时， 有read forward和read proxy两种特殊模式进行处理。写操作在缓存不命中的情况下有writeproxy的特殊模式进行处理。

2.read forward模式

当进行read操作时，对象不在cache pool中，就直接返回，client直接向data pool发送请求，数据直接返回给client。

如图13-2所示，当客户端向cache pool发起读请求时，出现cache miss状态，就返回redirect信息给client，client根据返回的redirect信息，再次直接向base pool层发起读请求，并返回需要的数据。

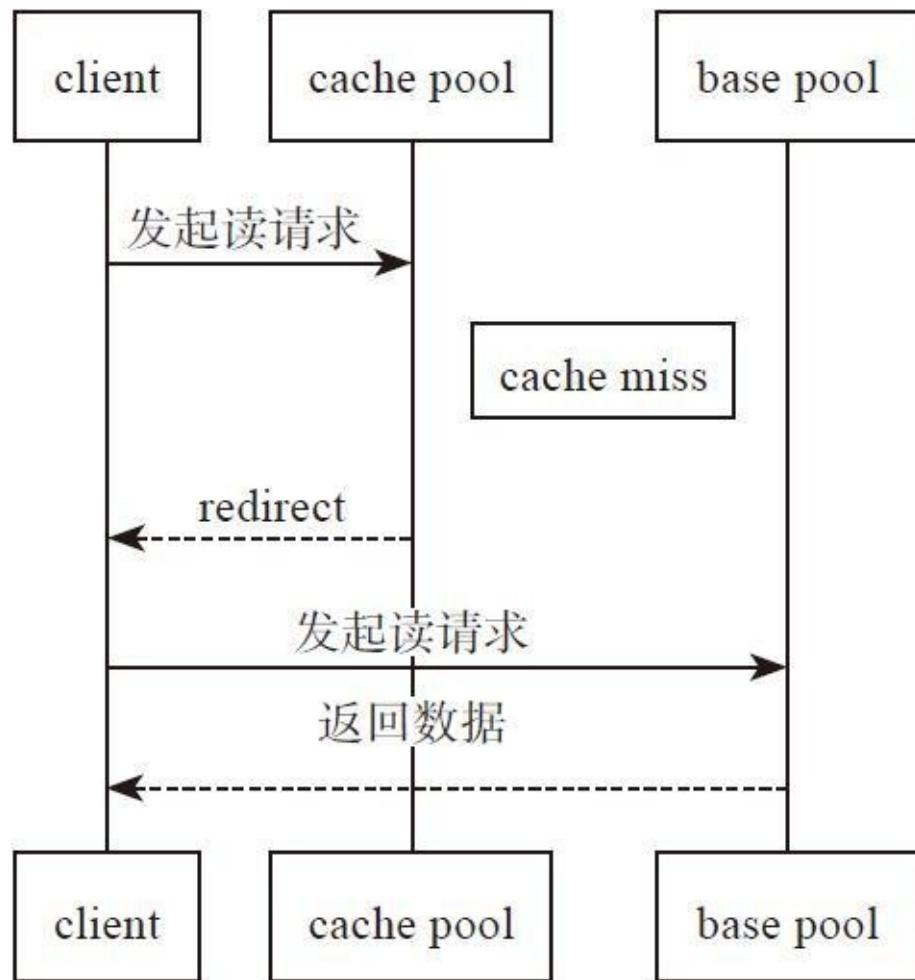


图13-2 read forward模式

3.Read proxy模式

当进行read操作时，对象不在cache pool中，cache pool层向data pool发送请求，返回给cache pool层，cache pool层再返回给client数据。

如图13-3所示，当client向cache pool发起读请求，该对象在cache pool处于cache miss状态，cache pool层向base pool层发起读请求，返回数据给cache pool层，cache pool层再将数据返回给client。

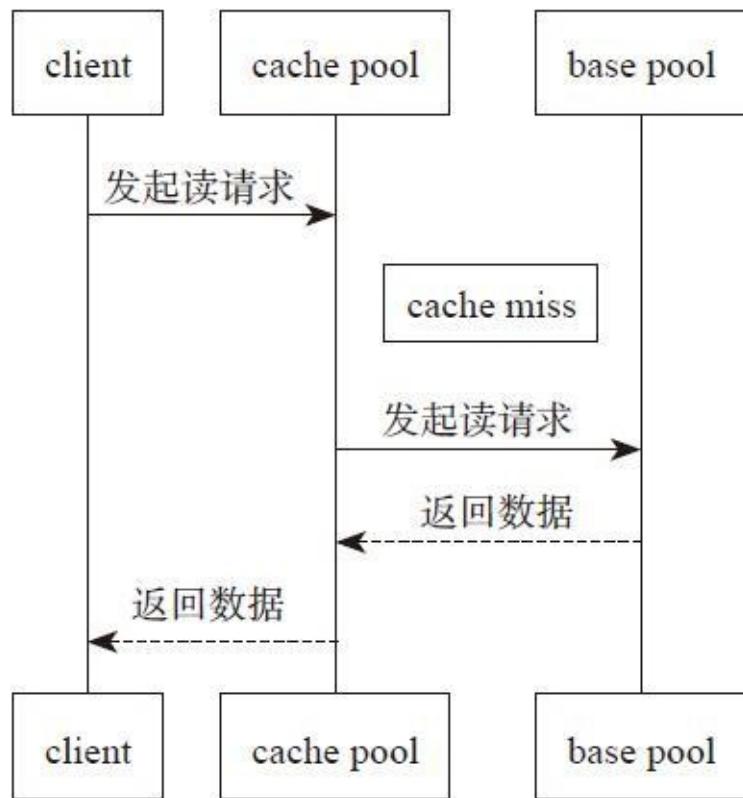


图13-3 read proxy模式

4.write proxy模式

图13-4是write proxy模式示意图，当客户端先cache pool发起写操作时，处于cache miss的状态，cache pool并不等该对象从base pool中加载到

cache pool中，然后写入，而是直接发请求给base pool，直接把数据写入base pool层。

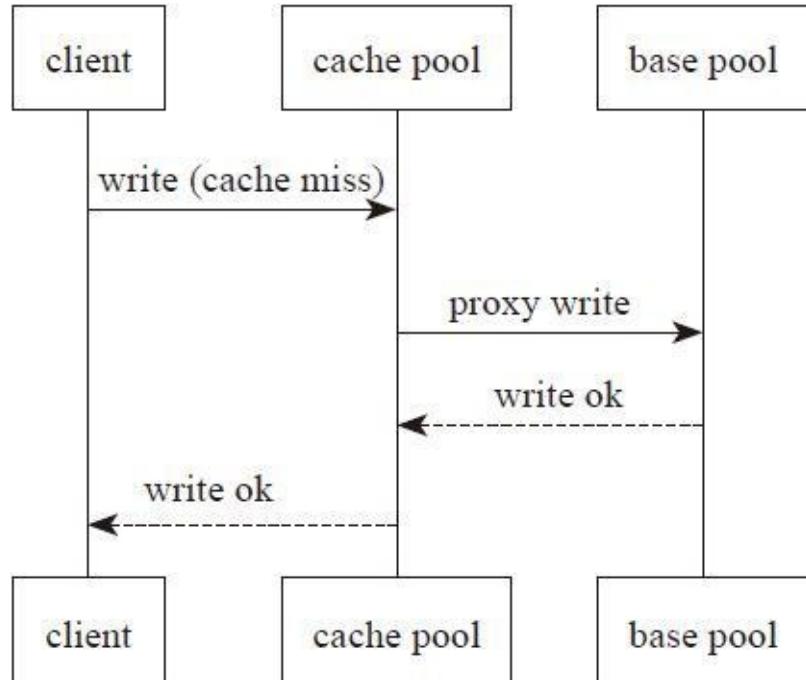


图13-4 write proxy模式

write proxy模式中，当cache pool处于cache miss状态时，直接写入base pool，降低了写操作的延迟，提高了写操作的性能。

5.read only模式

read only模式也称为：write-around或者read cache。读请求直接发送给cache pool，写请求并不经过cache pool，而是直接发送给base pool。

这种方式下，cache pool设置为单副本，极大地减少缓存空间的占用。当cache pool层失效时，也不会有数据丢失。这种模式比较适合数据一

次写入多次读取的应用场景。例如图片、视频、音频等的读写操作。

13.4 Cache Tier的源码分析

Cache Tier的代码分布在Ceph源代码的各个模块，其核心在对象的数据读写路径上。下面就着重分析相关的数据结构，并介绍其如何影响对象读写路径的。

13.4.1 pool中的Cache Tier数据结构

数据结构pg_pool_t里有Cache Tier相关数据字段，如下所示：

```
set<uint64_t> tiers;
int64_t tier_of;
```

这两个字段用来设置pool的属性：

- 如果当前pool是一个cache pool，那么tier_of记录了该cache pool的base pool层。
- 如果当前pool是base pool，那么tiers就记录该base pool的cache pool层，一个base pool可以设置多个cache pool层。

使用如下命令来设置Cache Tier的两个存储层之间的关系：

```
ceph osd tier add {data_pool} {cache_pool}
```

pool的信息都由Monitor来持久化存储。该命令行程序发送请求给Monitor，然后由Monitor相关pool设置上述属性值：

```
cache_mode_t cache_mode; // cache的模式
```

cache_mode设置了cache的模式，可以通过下面命令设置：

```
ceph osd tier cache-mode {cachepool} {cache-mode}
    int64_t read_tier;
    int64_t write_tier;
```

read_tier和write_tier分别设置base pool的读缓存层和写缓存层。根据Ceph不同的Cache Tier模式来设置。如果是write_back模式，那么该cache pool既是read层，又是write层。如果只是read only模式，那么设置的cache pool只是read层，并不是write层。

根据模式分别设置read tier和write tier字段的命令如下：

```
ceph osd tier set-overlay {data_pool} {cache_pool}
    uint64_t target_max_bytes;
    uint64_t target_max_objects;
```

字段target_max_bytes设置了cache pool的最大的字节数。
target_max_objects设置了cache pool的最大对象数量。

```
uint32_t cache_target_dirty_ratio_micro;
uint32_t cache_target_dirty_high_ratio_micro;
uint32_t cache_target_full_ratio_micro;
```

上面这三项分别为：

- 目标脏数据率：当脏数据率达到这个值时，后台agent开始flush数据
- 高目标脏数据率：当脏数据率达到这个值时，后台agent开始高速flush数据
- 数据满的比率：当数据达到这个比率就认为cache处于满的状态

字段cache_min_flush_age设置了一个对象在cache中被返回base tier的最长时间。字段cache_min_evict_age设置了一个对象在cache中被evict操作的最小操作时间。

hit_set_params是hitset相关的参数。字段hit_set_period用来设置每过该段时间，系统要重新产生一个新的hit_set对象来记录对象的缓存统计信息。hit_set_count用来记录系统保存最近的多少个hit_set记录。
use_gmt_hitset表示hitset archive对象的命名规则。

13.4.2 HitSet

类HitSet用来跟踪和统计对象的访问行为。目前仅记录了对象是否在缓存中。它定义了一个缓存查找的抽象接口。目前的三种实现方式分别为：ExplicitObjectHitSet、ExplicitHashHitSet、BloomHitSet。下面分别介绍

1. ExplicitObjectHitSet

类ExplicitObjectHitSet用一个基于hobject的set来记录对象的命中：

```
ceph::unordered_set<hobject_t> hits;
```

这种方式实现简单，直观。但是缺点也很明显，要在内存中缓存数据结构hobject，其占用内存空间比较大。

可以粗略地统计一下，在假设对象的名字占40个字节，一个hobject大约占100字节那么大：

400G 的 SSD 盘(卡) 的占用空间	$400G / 4M \times 100 = 10M$
4T 的 SSD 盘(卡) 的占用空间	$4T / 4M \times 100 = 100M$

2. ExplicitHashHitSet

类ExplicitHashHitSet基于对象32位哈希值的set来记录的对象命中：

```
ceph::unordered_set<uint32_t> hits;
```

每个对象占用4字节的内存空间，占用内存空间如下：

400G 的 SSD 盘 (卡) 的占用空间	$400\text{G} / 4\text{M} \times 4 = 400\text{k}$
4T 的 SSD 盘 (卡) 的占用空间	$4\text{T} / 4\text{M} \times 4 = 4\text{M}$

这种方式占用空间相对较少。但缺点也比较明显：当知道一个对象的哈希值，去寻找对应的对象时，需要扫描所有的对象，计算对象的哈希值来对比查找。

3.BloomHitSet

类BloomHitSet采用压缩了的Bloom filter的方式来记录对象是否在缓存中。其原理实现在这里就不详细介绍了，它进一步减少了占用的内存空间。

特别需要注意的是：在模式CACHEMODE_NONE（没有cache tier）和模式read only模式下，是不需要HitSet来记录缓存命中的。只有在以下模式write back、read forward、read_proxy中才需要HitSet来记录缓存命中。

13.4.3 Cache Tier的初始化

Cache Tier初始化有两个入口，如下所示：

- on_active：如果该pool已经设置为cache pool，在该cache pool的所有PG处于active状态后初始化。
- on_pool_change：当该pool的所有PG都已经处于active状态后，才设置该pool为cache pool，那么就等待Monitor通知osd map相关信息的变化，在on_pool_change函数里初始化。

1.hit_set_setup

函数hit_set_setup用来创建并初始化HitSet对象：

```
void ReplicatedPG::hit_set_setup()
```

具体实现如下：

- 1) 首先检查如果PG既不处于active状态，也不处于primary状态，就调用函数hit_set_clear清除hit_set相关的记录，直接返回不做任何设置。
- 2) 如果参数pool.info.hit_set_count为零，或者参数pool.info.hit_set_period为零，或者pool.info.hit_set_params参数为HitSet::TYPE_NONE，就调用hit_set_clear清空hit_set的记录，并调用函数

hit_set_remove_all删除所有的HitSet对象。

- 3) 调用函数hit_set_create根据类型创建相应的HitSet类。
- 4) 调用函数hit_set_apply_log来添加从PG日志中获取的新对象记录，并添加到HitSet中。

2.agent_setup

函数agent_setup完成agent相关的初始化工作：

```
void ReplicatedPG::agent_setup()
```

具体实现如下：

- 1) 首先检查各种参数，如果PG处于下列情况之一：

- 不处于active状态。
- 非primary。
- 模式是pg_pool_t::CACHEMODE_NONE。
- pool.info.tier_of没有设置。
- cache pool不存在。

调用函数agent_clear，停止agent线程，清除相关设置。

- 2) 如果agent_state为空，就重新设置agent_state类，并设置相关的初始化参数。
- 3) 调用函数agent_choose_mode设置flush_mode模式和evict_mode模式，并把PG添加到agent相应的工作队列中。

13.4.4 读写路径上的Cache Tier处理

在OSD的正常读写路径上，如果该pool有Cache Tier设置，处理逻辑就发生了变化。如下所示：

```
void ReplicatedPG::do_op(OpRequestRef& op){
{
    .....
    bool in_hit_set = false;
    if (hit_set) {
        if (obc.get()) {
            if (obc->obs.oi.soid != hobject_t() && hit_set->contains(obc->obs.oi.soid))
                in_hit_set = true;
        } else {
            if (missing_oid != hobject_t() && hit_set->contains(missing_oid))
                in_hit_set = true;
        }
        if (!op->hitset_inserted) {
            hit_set->insert(oid);
            op->hitset_inserted = true;
            if (hit_set->is_full() ||
                hit_set_start_stamp + pool.info.hit_set_period <= m->get_recv_stamp())
                hit_set_persist();
        }
    }
    if (agent_state) {
        if (agent_choose_mode(false, op))
            return;
    }
    if (maybe_handle_cache(op,
                           write_ordered,
                           obc,
                           r,
                           missing_oid,
                           false,
                           in_hit_set))
        return;
    .....
}
```

这里增加了相关的处理逻辑：

- 1) 首先根据对象的上下文信息obc获取的情况，分两种情况调用函数

hit_set->contains检查该对象是否命中，如果对象没有在hit_set中就调用hit_set->insert把该对象插入hit_set中。

- 2) 如果hit_set已经满了，或者时间达到了pool.info.hit_set_period的值，就持久化保存该hit_set对象到磁盘上，创建一个新的hit_set对象。
- 3) 调用函数agent_choose_mode对相应的PG设置flush模式和evict模式。
- 4) 调用函数maybe_handle_cache来处理有关cache的读写请求。

下面分别介绍相应函数。

1.agent_choose_mode

函数agent_choose_mode用来计算一个PG的flush_mode和evict_mode的参数值：

```
bool ReplicatedPG::agent_choose_mode(bool restart, OpRequestRef op)
```

如果该函数返回值为true，就表明该请求Op被重新加入请求队列里（由于EvictMode为Full），其他情况都返回false值。

这个函数实现过程如下：

- 1) 如果该agent_state处于agent_state->delaying状态，就返回flase值。

- 2) 如果info.stats.stats_invalid为true， 表明当前的统计信息无效。 agent 模式的计算依赖这些统计信息，在这种情况下暂时跳过，暂不计算 flush_mode和evict_mode的值。
- 3) 计算divisor值，也就是cache_pool里的PG数量。
- 4) 计算unflushable，也就是不能刷回的对象，并去掉HitSet相关的对象。
- 5) 获取base_pool并检查如果base_pool不支持omap，就去掉所有需要 omap支持的对象。
- 6) 计算num_user_objects，其值为统计的对象数减去unflushable对象数。
- 7) 计算num_user_bytes，用统计信息的字节数减去unflushable_bytes的字节数。
- 8) 计算脏对象的数目num_dirty的值，如果base_pool不支持omap，就去掉带omap的对象。
- 9) 计算dirty_micro和full_micro，分别是脏数据的比率和数据满的比率，单位为百万分之一：
 - a) 如果设置了pool.info.target_max_bytes，就按照字节计算。首先计算每个对象的平均大小avg_size：

·dirty_micro=脏对象数目×每个对象的平均大小/每个PG的平均字节数
×1000000

·full_micro=用户对象数目×每个对象平均大小/每个PG的平均字节数
×1000000

b) 如果设置了pool.info.target_max_objects, 就按照对象数目来计算：

·dirty_objects_micro=脏对象数目/每个PG的平均对象数目×1000000

·full_objects_micro=用户对象数目/每个PG的平均对象数目×1000000

最终选择两种计算方式中最大的一个为最终结果。

10) 获取flush_target和flush_high_target参数。如果是restart, 或者当前的flush_mode为IDLE, 就用flush_slop对二者增加, 否则就减少（通过flush_slop对比率做修正）。

11) 根据脏数据的比例, 设置flush_mode：

·如果dirty_micro大于flush_high_target, 设置flush_mode为
TierAgentState :: FLUSH_MODE_HIGH。

·如果dirty-micro大于flush_target, 就设置flush_mode为
TierAgentState :: FLUSH_MODE_LOW。

12) 获取evict_target的值, 用evict_slop做一点比例的修正。

- 13) 如果full_micro大于1000000, evict_mode就设置为EVICT_MODE_FULL模式, evict_effort设置为最大1000000。
- 14) 如果full_mircor大于evict_target, 就设置evict_mode为EVICT_MODE_SOME类型。这时候需要计算evict_effort的值, evict_effort为PG在agent工作队列里的优先级 :
- over=full-evict_target
 - span=1-evict_target
 - evict_effort=over/span, 转换成单位为百万分之一
 - 通过g_conf->osd_agent_quantize_effort的修正, 使得evict_effort的级别不会太多。

例13-1 举例说明evict_effort

例如 : evict_target : 60%

full_micro : 80%

over=80%-60%

span=1-60%

$$\text{evict_effort} = (80\% - 60\%) / (1 - 60\%) = 50\%$$

本例子里为了便于理解，都使用了单位百分之一。

15) 设置新计算的flush_mode值，并更新相关的统计信息。

16) 设置新计算的evict_mode值，如果evict_mode是由EVICT_MODE_FULL变为其他类型，并且PG的状态为is_active()，就需要把当前的op以及因cache full而等待的操作都重新加入请求队列，设置返回值为true。

17) 根据模式，做相应的处理：

- a) 如果idle，就调用函数agent_disable_pg把该PG从agent_queue中删除。
- b) 如果是restart，或者之前是idle，就调用函数agent_enable_pg把该PG重新加入agent_queue处理队列中。
- c) 如果之前已经在队列中，就调用函数agent_adjust_pg来调整evict_effort，也就是调整在队列中的优先级。

2.maybe_handle_cache_detail

函数maybe_handle_cache处理有关cache的逻辑，调用函数maybe_handle_cache_detail来完成，处理流程如下：

- 1) 首先检查op的请求标志里如果有CEPH_OSD_FLAG_IGNORE_CACHE标志就直接返回。如果

pool.info.cache_mode为pg_pool_t::CACHEMODE_NONE也直接返回。

- 2) 如果可以获取obc，并且是write_ordered，该对象被阻塞，就返回cache_result_t::NOOP值。
- 3) 如果该对象确实不存在，就返回cache_result_t::NOOP值。
- 4) 如果可以获得obc，并且对象存在，则认为缓存命中，则直接返回cache_result_t::NOOP值。
- 5) 检查如果操作里设置了op->need_skip_handle_cache ()，就返回cache_result_t::NOOP值。
- 6) 如果是CACHEMODE_WRITEBACK模式（下列情况都是缓存没有命中，cache pool中没有需要的对象）：

情况1：当前agent_state的evict_mode为EVICT_MODE_FULL，说明当前的cache已经接近满了：

- a) 如果是只读操作：

```
if (!op->may_write() && !op->may_cache() && !write_ordered && !must_promote)
```

如果可以can_proxy_read，就调用函数do_proxy_read读取。否则调用do_cache_redirect函数返回给客户端redirect信息，客户端再去访问base pool，这种方式就是forward方式读取。

b) 如果是write操作，就调用block_write_on_full_cache阻塞当前的操作。

情况2：不是EVICT_MODE_FULL模式：

a) 如果hit_set为空：说明在这种模式下不需要hitset（可能使read only模式），调用promote_object函数去base pool读取该对象。

b) 如果hit_set不为空，并且op->may_write () || op->may_cache ()，也就是写操作：

·如果允许can_proxy_write，就调用do_proxy_write来以proxy的方式写入，否则调用promote_object操作，返回cache_result_t：：

BLOCKED_PROMOTE

·如果是以proxy_write方式写入，检查是否还需要调用函数promote_object操作。

c) 否则，也就是read操作

·如果允许proxy_read，就调用函数do_proxy_read以代理的方式来读取。

·检查如果需要promote，就调用函数maybe_promote来检查。

·如果既没有promote，又没有proxy_read，就调用do_cache_redirect来读取。

- 7) 如果是CACHEMODE_FORWARD模式，就直接调用函数do_cache_redirect，以forward模式读取。
- 8) 如果是CACHEMODE_READONLY模式：如果是读操作，就调用promote_object函数从base pool读取；如果是写操作，就以forward的方式写。
- 9) 如果是CACHEMODE_READFORWARD模式
 - 如果是write操作，evict的模式为TierAgentState :: EVICT_MODE_FULL，就阻塞，否则就调用函数promote_object来读取该对象，然后写入。
 - 如果是读，就以forward模式读取。

如果是CACHEMODE_READPROXY模式，对于写，和CACHEMODE_READ-FORWARD处理相同；对于读，就以proxy模式读取。

下面介绍上述处理过程中的，处理代理读、代理写、从数据层加载对象到缓存层的具体处理函数。

函数do_proxy_read给base pool发送请求消息，来读取cache pool没有的对象数据。函数finish_proxy_read完成了prox_read的回调函数，其对结果做检查，并清理等待列表。通过函数complete_read_ctx给client端发送读操作

的返回消息。特别注意的是，函数并没有把对象的数据写入cache pool，所
以后续还需要调用函数promote_object从base pool读取该对象数据，然后写
入cache pool中。

函数do_proxy_write直接写数据到base pool中。函数finish_proxy_write
为do_proxy_write的回调函数，完成给客户端的发送ACK应对。同样，
cache pool中并没有该数据对象，还需要后续调用promote_object函数把对
象从base pool中读到cache pool中。

函数promote_object完成base pool读取相关的对象并将其写入cache pool
中，处理过程如下：

- 1) 首先检查scrubber.write_blocked_by_scrub是否处于block状态。
- 2) 构造PromoteCallback回调函数，然后调用函数start_copy拷贝数
据。

函数start_copy用于执行copy操作拷贝数据，其处理过程如下：

- 1) 检查如果copy_ops里有该对象，就调用函数cancel_copy取消正在进
行的copy操作。
- 2) 构造CopyOp操作，并添加到copy_ops的map中。
- 3) 调用obc->start_block ()，把该obc设置为blocked为true的状态。
- 4) 调用函数_copy_some做相应的操作。

函数_copy_some执行copy操作做部分数据量的拷贝，它是通过调用osdc的读操作来完成，其处理过程如下：

- 1) 根据cop->flags设置各种flags标志。
- 2) 构建C_GatherBuilder，它是一个Context的回调函数。
- 3) 调用函数cop->cursor.is_initial () 判断是否为第一次操作，并且cop->mirror_snapset为ture，就需要调用list_snaps获取snapSet的信息，并把结果保存在cop->results.snapset中，保存tid到cop->objecter_tid2中。
- 4) 如果cop->results.user_version被设置了（在第一次操作中设置），在以后的操作中，需要执行assert_version操作。
- 5) 如果该pool不支持omap (ec)，在copyget_flags标志中，设置CEPH OSD COPY GET FLAG NOTSUPP OMAP标志。
- 6) 调用op.copy_get操作，设置相关的操作，并调用函数op.set_last_op_flags设置最后一个op的flags标志。
- 7) 构造C_Copyfrom回调函数，调用函数gather.set_finisher设置gather的finisher函数。
- 8) 调用函数osd->objecter->read把操作发送出去。
- 9) 把tid设置为fin->tid，调用函数gather.activate () 使回调函数激活。

函数process_copy_chunk为Copyfrom的回调函数，完成了数据的部分写入。函数finish_promote为PromoteCallback的回调函数，完成后续promote操作。

13.4.5 cache的flush和evict操作

cache pool空间不够时，需要选择一些脏对象回刷到数据层（即flush操作），并将一些clean对象从缓存层剔除（即evict操作），以释放更多的缓存空间。这两种操作都是在后台完成的。flush操作和evict操作算法的好坏，决定了Cache Tier的缓存命中率。

1.OSDService中Agent相关的数据结构

在类OSDService中，定义了一个AgentThread的后台线程，用于完成两个任务：一是把dirty对象从cache pool层适时地回刷到base pool层，二是从cache pool层剔除掉一些不经常访问的clean对象。

```
Class OSDService{
    .....
    Mutex agent_lock; //agent线程锁，保护以下所有的数据结构

    Cond agent_cond; //线程相应的条件变量

    map<uint64_t, set<PGRef> > agent_queue;
    //所有回刷或者剔除所需的

    PG集合，根据

    PG集合的优先级，保存在不同的

    map中

    set<PGRef>::iterator agent_queue_pos;
    //当前在扫描的

    PG集合的一个位置，只有

    agent_valid_iterator为

    true时，这个指针才有效，
```

否则从集合的起始处扫描

```
bool agent_valid_iterator;
int agent_ops; // 所有正在进行的回刷和剔除操作

int flush_mode_high_count; //如果回刷模式为
HIGH模式，该值就增加

set<hobject_t, hobject_t::BitwiseComparator> agent_oids;
//所有正在进行的
agent的操作（回刷或剔除）的对象

bool agent_active; //agent是否有效

//agent线程

struct AgentThread : public Thread {
    OSDService *osd;
    explicit AgentThread(OSDService *o) : osd(o) {}
    void *entry() {
        osd->agent_entry();
        return NULL;
    }
} agent_thread;
bool agent_stop_flag; //agent 停止的标志

Mutex agent_timer_lock;
SafeTimer agent_timer;
//agent相关的定时器，该定时器用于：当扫描一个
```

PG对象时，该对象既没有剔除操作，也没有回刷

操作，就停止

PG的扫描，把该

PG加入到定时器，

5s后继续

```
} .....
```

2.数据结构TierAgentState

在ReplicatedPG的内部，变量agent_state用来保存PG相关的agent信息。

```
struct TierAgentState {
    hobject_t position;      //PG内扫描的对象位置

    int started;   // PG里所有对象扫描完成后，所发起的所有
                    agent操作数目。当
                    PG扫描完成

                    所有的对象，如果没有
                    agent操作，就需要延迟一段时间

    hobject_t start;  //本次扫描起始位置

    bool delaying;    //是否延迟

    //历史的统计信息

    pow2_hist_t temp_hist;
    int hist_age;
    map<time_t, HitSetRef> hit_set_map; //HitSet的历史记录

    //最近处于
    clean的对象

    list<hobject_t> recent_clean;
}
```

3.agent_entry

agent_entry是agent_thread的入口函数，它在后台完成flush操作和evict操作，具体处理流程如下：

- 1) 加agent_lock锁。该锁保护OSDService里所有和agent相关的字段。
- 2) 如果agent_stop_flag为true，就将agent_lock解锁并退出，否则继续以下操作。
 - 3) 扫描agent_queue队列，如果agent_queue为空，就在条件变量agent_cond上等待。
 - 4) 从队列agent_queue中取出级别最高的PG的集合top（始终从级别最高的取），如果top集合为空，就把该集合从agent_queue中删除，并使agent_valid_iterator集合内的PG指针设为false，使其无效。
 - 5) 检查如果正在进行的agent操作数agent_ops，如果该值大于所配置最大允许的agent操作的数量g_conf->osd_agent_max_ops，或者非agent_active，就等待。
 - 6) 如果agent_valid_iterator有效，就从agent_queue_pos处获取该PG set中的一个PG；否则就从该set的头开始，获取相应的PG指针。
 - 7) 获取agent操作的最大数目max值，以及agent_flush_quota的值。
 - 8) 调用函数pg->agent_work，正常情况返回true值。如果它的返回值为false，该PG处于delay状态，需要加入agent_timer定时器，在配置项g_conf->osd_agent_delay_time设定的秒数后，调用agent_choose_mode重新设置模式。

4.agent_work

agent_work完成一个PG内的evict操和flush操作， 主要的流程如下：

- 1) 调用函数lock () 对PG加锁， agent_state数据结构受它保护。
- 2) 检查如果agent_state为空， 或者agent_state->is_idle () ， 就解锁返回true值。
- 3) 调用函数agent_load_hit_sets加载hit_set的历史对象到内存中。
- 4) 调用函数pgbackend->objects_list_partial来扫描本PG的对象， 从agent_state->position开始位置扫描， 结果保存在对象向量ls中。最小扫描一个对象， 最大扫描数是配置参数
osd_pool_default_cache_max_evict_check_size设置的对象个数。
- 5) 对扫描到的ls中的对象， 做如下检查：
 - 跳过hitset对象。
 - 跳过degraded对象。
 - 跳过missing对象。
 - 跳过object context不存在的对象。
 - 跳过对象的obs不存储。

·跳过正在进行Scrub操作的对象。

·跳过已经被阻塞的对象。

·跳过有正在读写请求的对象。

·如果base_pool不支持omap，就跳过有omap的对象。

·如果agent_state->evict_mode !=TierAgentState::

EVICT_MODE_IDLE，调用agent_maybe_evict函数来剔除掉一些对象。

·如果当前的flush_mode不是IDLE状态，就调用agent_maybe_flush函数来回刷一些对象。

·如果started大于agent_max，也就是已经发起的agent操作数目大于最大允许的agent_max数目，就停止并退出。设置next的指针，也就是下次开始扫描的对象的起始位置。

6) 更新agent_state->hist_age的值，如果agent_state->hist_age大于g_conf->osd_agent_hist_halfife，就重置为0，并调用函数agent_state->temp_hist.decay () 把直方图的统计的对象减半。

7) 比较扫描的指针，如果PG的对象扫描了一圈后，total_started值为0，也就是没有agent操作（flush操作或者evict操作），就设置need_delay值为true，需要延迟一段时间。

8) 调用函数hit_set_in_memory_trim在内存中对一些旧的hitset对象做

trim操作。

9) 如果需要延迟，就调用函数agent_delay把该PG从agent_queue中删除。

10) 调用agent_choose_mode重新计算agent的evict和flush的模式值。

5.agent_maybe_evict

函数agent_maybe_evict完成一个对象的evict操作，处理过程分析如下：

1) 首先检查对象的状态：

- 如果不是after_flush，就需要确保对象处于clean状态，否则直接返回false值。

- 如果该对象还有watcher，说明还有客户端在使用该对象，返回false值。

- 对象如果处于blocked状态，返回false值。

- 如果对象处于cache_pined状态，返回false值。

- 验证对象的clone信息是否一致，如果不一致，返回false值。

2) 如果evict_mode为EVICT_MODE_SOME模式：

·需要确保该对象处于clean的时间大于pool.info.cache_min_evict_age的长度。

·调用函数agent_estimate_temp计算该对象的热度值temp。

·调用函数get_position_micro计算该对象的temp值在对象的统计直方图的位置值emp_upper。

·如果1000000减去temp_upper的值大于等于agent_state->evict_effort, 该对象就不删除，直接返回false值。

3) 如果evict_mode为EVICT_MODE_FULL模式，就调用函数_delete_oid删除该对象，最后调用函数simple_opc_submit发起实践的删除请求。

6.agent_maybe_flush

函数agent_maybe_flush完成一个对象的flush操作，处理过程如下：

- 1) 检查如果对象不脏，就返回false值。
- 2) 检查如果该对象处于cache_pinned，就返回false值。
- 3) 如果当前evict_mode不是evict_mode_full状态，就检查该对象处于dirty的时间是否超过pool.info.cache_min_flush_age值。
- 4) 检查对象是否处于activate状态。

5) 调用函数start_flush来完成对象的回刷操作。

7.start_flush

函数start_flush完成实际的flush操作，具体处理过程如下：

1) 首先调用obc->ssc->snapset.get_filtered把已经删除的snap对象过滤掉。

2) 检查比当前clone对象版本更早的克隆对象：

·如果还有版本更早的克隆对象处于missing状态，就返回-ENOENT错误码。

·如果还有版本更早的克隆对象，获取该对象的older_obi值，并且标记该对象处于dirty状态，直接返回EBUSY；如果该对象的obi不存在，就默认认为clean状态。

例如：clones[1, 4, 5, 8]，当前snap为5，就必须确保[1, 4]处于clean状态。

3) 如果blocking设置为true，就设置对象处于blocked状态。

4) 检查如果该对象在flush_ops中，也就是该对象已经在flush中，根据不同的情况分别处理。

5) 最后处理克隆对象的flush操作。

13.5 本章小结

本章介绍了Ceph的Cache Tier的原理和架构，及其各种模式，以及因引入Cache Tier而引起Ceph的读写路径上不同的处理。最后介绍了Cache Tier后台的flush操作和evict操作的处理过程。

Ceph的Cache Tier功能目前在对象的访问频率和热点统计上的实现都比较简单，有待后续实现更好的基于自学习的Cache算法。