

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

作者简介

常涛

国内较早一批接触Ceph的先行者，具有多年分布式存储开发经验，曾在雅虎北京研发中心工作，后到京东、华为任职，目前在ZettaKit创业公司从事分布式存储、云计算相关技术研发。

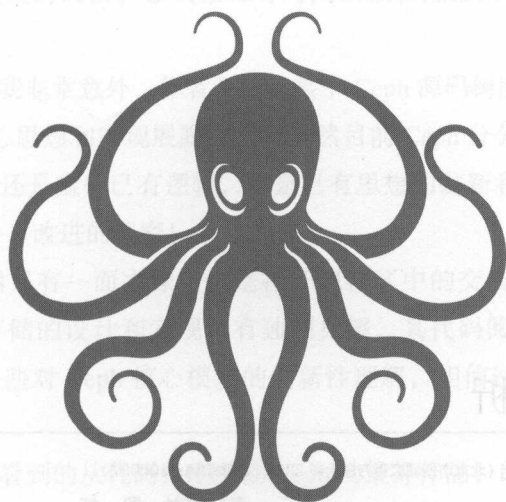


技术丛书

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

图书在版编目 (CIP) 数据

Ceph 源码分析 / 常涛编著. —北京: 机械工业出版社, 2016.10
(大数据技术丛书)

ISBN 978-7-111-55207-9

I. C… II. 常… III. 分布式文件系统 IV. TP316

中国版本图书馆 CIP 数据核字 (2016) 第 257284 号

Ceph 源码分析

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 吴 怡

责任校对: 殷 虹

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2016 年 11 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 16.75

书 号: ISBN 978-7-111-55207-9

定 价: 59.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

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 源码树枝解得恰到好处，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 巨无霸们组成的巨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 and 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, 介绍相关的操作实现流程。

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

目 录

序言
前言

第 1 章 Ceph 整体架构	1
1.1 Ceph 的发展历程	1
1.2 Ceph 的设计目标	2
1.3 Ceph 基本架构图	2
1.4 Ceph 客户端接口	3
1.4.1 RBD	4
1.4.2 CephFS	4
1.4.3 RadosGW	4
1.5 RADOS	6
1.5.1 Monitor	6
1.5.2 对象存储	7
1.5.3 pool 和 PG 的概念	7
1.5.4 对象寻址过程	8
1.5.5 数据读写过程	9
1.5.6 数据均衡	10
1.5.7 Peering	11
1.5.8 Recovery 和 Backfill	11

1.5.9 纠删码	11
1.5.10 快照和克隆	12
1.5.11 Cache Tier	12
1.5.12 Scrub	13
1.6 本章小结	13

第2章 Ceph 通用模块

2.1 Object	14
2.2 Buffer	16
2.2.1 buffer::raw	16
2.2.2 buffer::ptr	17
2.2.3 buffer::list	17
2.3 线程池	19
2.3.1 线程池的启动	20
2.3.2 工作队列	20
2.3.3 线程池的执行函数	21
2.3.4 超时检查	22
2.3.5 ShardedThreadPool	22
2.4 Finisher	23
2.5 Throttle	23
2.6 SafeTimer	24
2.7 本章小结	25

第3章 Ceph 网络通信

3.1 Ceph 网络通信框架	26
3.1.1 Message	27
3.1.2 Connection	29
3.1.3 Dispatcher	29
3.1.4 Messenger	29

3.1.5 网络连接的策略	30
3.1.6 网络模块的使用	30
3.2 Simple 实现	32
3.2.1 SimpleMessenger	33
3.2.2 Acceptor	33
3.2.3 DispatchQueue	33
3.2.4 Pipe	34
3.2.5 消息的发送	35
3.2.6 消息的接收	36
3.2.7 错误处理	37
3.3 本章小结	38
第 4 章 CRUSH 数据分布算法	39
4.1 数据分布算法的挑战	39
4.2 CRUSH 算法的原理	40
4.2.1 层级化的 Cluster Map	40
4.2.2 Placement Rules	42
4.2.3 Bucket 随机选择算法	46
4.3 代码实现分析	49
4.3.1 相关的数据结构	49
4.3.2 代码实现	50
4.4 对 CRUSH 算法的评价	52
4.5 本章小结	52
第 5 章 Ceph 客户端	53
5.1 Librados	53
5.1.1 RadosClient	54
5.1.2 IoCtxImpl	56
5.2 OSDC	56

5.2.1	ObjectOperation	56
5.2.2	op_target	57
5.2.3	Op	57
5.2.4	Striper	58
5.2.5	ObjectCacher	59
5.3	客户写操作分析	59
5.3.1	写操作消息封装	60
5.3.2	发送数据 op_submit	61
5.3.3	对象寻址 _calc_target	61
5.4	Cls	62
5.4.1	模块以及方法的注册	62
5.4.2	模块的方法执行	63
5.4.3	举例说明	64
5.5	Librbd	65
5.5.1	RBD 的相关的对象	65
5.5.2	RBD 元数据操作	66
5.5.3	RBD 数据操作	67
5.5.4	RBD 的快照和克隆	69
5.6	本章小结	71
第 6 章 Ceph 的数据读写		72
6.1	OSD 模块静态类图	72
6.2	相关数据结构	73
6.2.1	Pool	74
6.2.2	PG	75
6.2.3	OSDMap	75
6.2.4	OSDOp	77
6.2.5	Object_info_t	77
6.2.6	ObjectState	78

6.2.7 SnapSetContext	79
6.2.8 ObjectContext	79
6.2.9 Session	80
6.3 读写操作的序列图	81
6.4 读写流程代码分析	83
6.4.1 阶段 1: 接收请求	83
6.4.2 阶段 2: OSD 的 op_wq 处理	85
6.4.3 阶段 3: PGBackend 的处理	95
6.4.4 从副本的处理	95
6.4.5 主副本接收到从副本的应答	95
6.5 本章小结	96
第 7 章 本地对象存储	97
7.1 基本概念介绍	98
7.1.1 对象的元数据	98
7.1.2 事务和日志的基本概念	98
7.1.3 事务的封装	99
7.2 ObjectStore 对象存储接口	100
7.2.1 对外接口说明	101
7.2.2 ObjectStore 代码示例	101
7.3 日志的实现	102
7.3.1 Journal 对外接口	102
7.3.2 FileJournal	103
7.4 FileStore 的实现	109
7.4.1 日志的三种类型	110
7.4.2 JournalingObjectStore	111
7.4.3 Filestore 的更新操作	112
7.4.4 日志的应用	115
7.4.5 日志的同步	115

7.5 omap 的实现	116
7.5.1 omap 存储	117
7.5.2 omap 的克隆	118
7.5.3 部分代码实现分析	119
7.6 CollectionIndex	120
7.6.1 CollectIndex 接口	122
7.6.2 HashIndex	123
7.6.3 LFNIndex	124
7.7 本章小结	124

第 8 章 Ceph 纠删码

8.1 EC 的基本原理	125
8.2 EC 的不同插件	126
8.2.1 RS 编码	126
8.2.2 LRC 编码	126
8.2.3 SHEC 编码	128
8.2.4 EC 和副本的比较	129
8.3 Ceph 中 EC 的实现	129
8.3.1 Ceph 中 EC 的基本概念	129
8.3.2 EC 支持的写操作	130
8.3.3 EC 的回滚机制	131
8.4 EC 的源代码分析	132
8.4.1 EC 的写操作	132
8.4.2 EC 的 write_full	133
8.4.3 ECBackend	133
8.5 本章小结	133

第 9 章 Ceph 快照和克隆

9.1 基本概念	134
----------------	-----

9.1.1 快照和克隆	134
9.1.2 RDB 的快照和克隆比较	135
9.2 快照实现的核心数据结构	137
9.3 快照的工作原理	139
9.3.1 快照的创建	139
9.3.2 快照的写操作	139
9.3.3 快照的读操作	140
9.3.4 快照的回滚	141
9.3.5 快照的删除	141
9.4 快照读写操作源代码分析	141
9.4.1 快照的写操作	141
9.4.2 make_writeable 函数	142
9.4.3 快照的读操作	145
9.5 本章小结	146

第 10 章 Ceph Peering 机制

10.1 statechart 状态机	147
10.1.1 状态	147
10.1.2 事件	148
10.1.3 状态响应事件	148
10.1.4 状态机的定义	149
10.1.5 context 函数	150
10.1.6 事件的特殊处理	150
10.2 PG 状态机	151
10.3 PG 的创建过程	151
10.3.1 PG 在主 OSD 上的创建	151
10.3.2 PG 在从 OSD 上的创建	153
10.3.3 PG 的加载	154
10.4 PG 创建后状态机的状态转换	154

10.5 Ceph 的 Peering 过程分析	156
10.5.1 基本概念	156
10.5.2 PG 日志	159
10.5.3 Peering 的状态转换图	166
10.5.4 pg_info 数据结构	167
10.5.5 GetInfo	169
10.5.6 GetLog	176
10.5.7 GetMissing	181
10.5.8 Active 操作	183
10.5.9 副本端的状态转移	187
10.5.10 状态机异常处理	188
10.6 本章小结	188
第 11 章 Ceph 数据修复	189
11.1 资源预约	190
11.2 数据修复状态转换图	191
11.3 Recovery 过程	193
11.3.1 触发修复	193
11.3.2 ReplicatedPG	195
11.3.3 pgbackend	199
11.4 Backfill 过程	205
11.4.1 相关数据结构	205
11.4.2 Backfill 的具体实现	205
11.5 本章小结	210
第 12 章 Ceph 一致性检查	211
12.1 端到端的数据校验	211
12.2 Scrub 概念介绍	213
12.3 Scrub 的调度	213

12.3.1 相关数据结构	214
12.3.2 Scrub 的调度实现	214
12.4 Scrub 的执行	217
12.4.1 相关数据结构	217
12.4.2 Scrub 的控制流程	219
12.4.3 构建 ScrubMap	221
12.4.4 从副本处理	224
12.4.5 副本对比	225
12.4.6 结束 Scrub 过程	228
12.5 本章小结	228

第 13 章 Ceph 自动分层存储

13.1 自动分层存储技术	230
13.2 Ceph 分层存储架构和原理	231
13.3 Cache Tier 的模式	231
13.4 Cache Tier 的源码分析	234
13.4.1 pool 中的 Cache Tier 数据结构	234
13.4.2 HitSet	236
13.4.3 Cache Tier 的初始化	237
13.4.4 读写路径上的 Cache Tier 处理	238
13.4.5 cache 的 flush 和 evict 操作	245
13.5 本章小结	250

West 也相应成立了 Inktank 公司专注于 Ceph 的部署。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。

2017 年，Ceph 发布了第一个稳定版本。2018 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，社区平均每个月发布一个稳定版本。目前的最新版本为 10.2.1。

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。

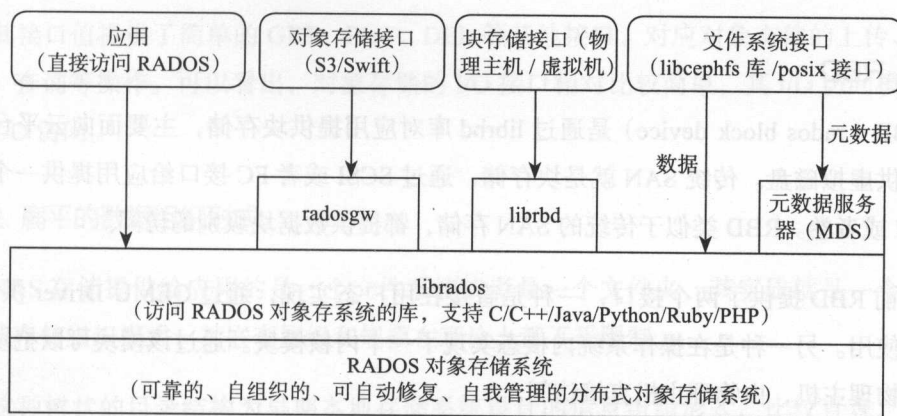


图 1-2 Ceph 基本架构图

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 librbid 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。

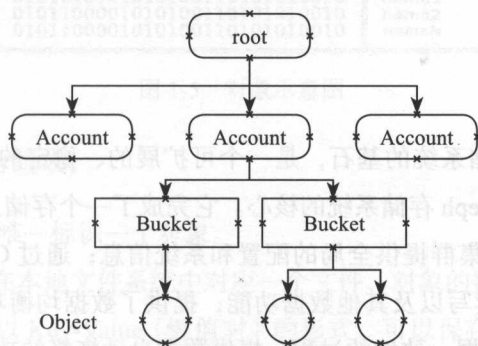


图 1-3 Amazon S3 的对象存储结构

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。

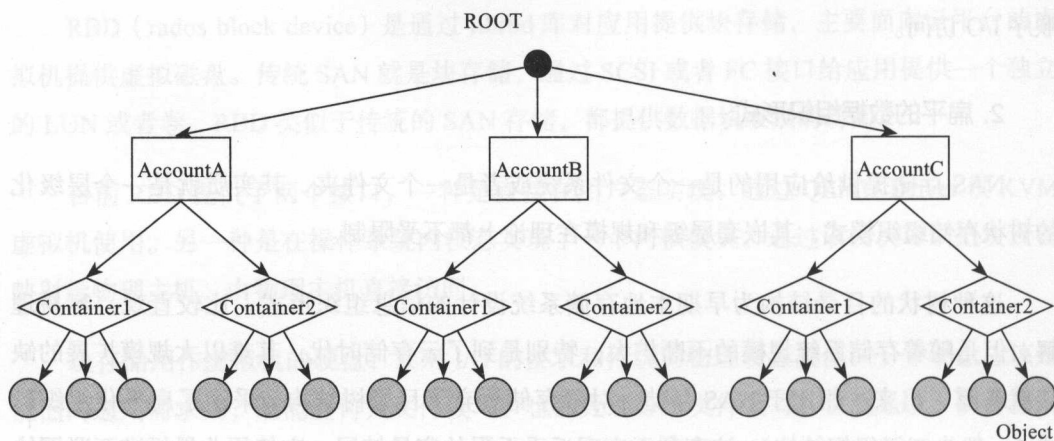


图 1-4 OpenStack Swift 对象存储结构

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。

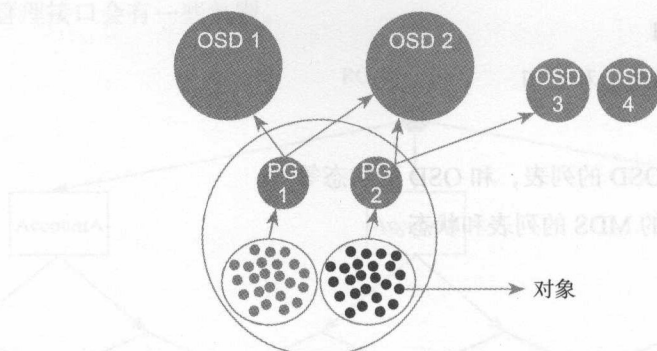


图 1-6 PG 的概念示意图

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。

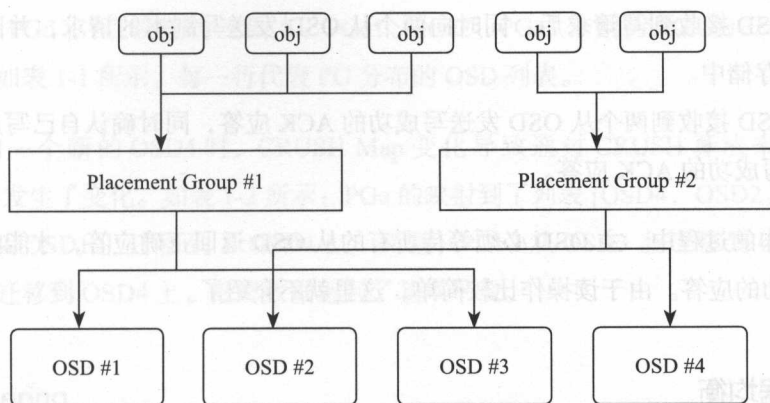


图 1-7 对象寻址过程

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：

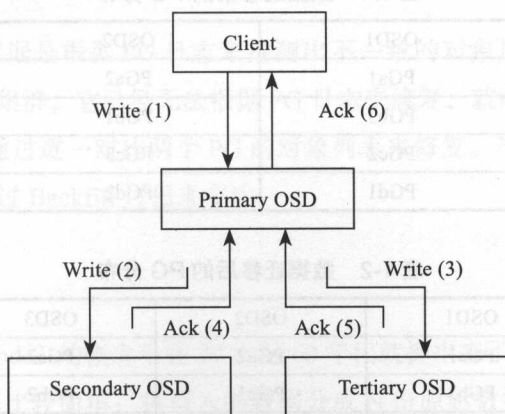


图 1-8 读写过程

- 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。

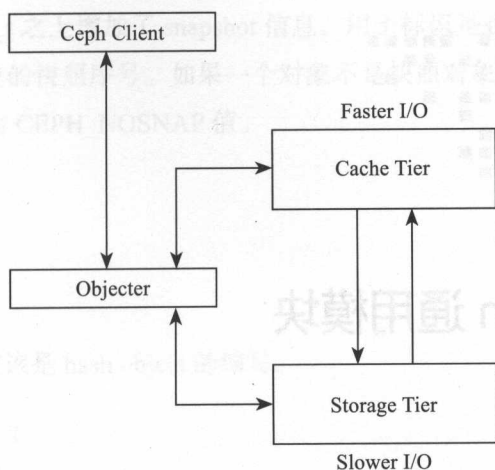


图 1-9 Cache Tier 结构图

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。

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，能够更好地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文件。在代码实现中，有 object、subject、hobject、ghobject 等不同的类。

结构 object_t 对应本地文件系统中的文件，name 就是对象名：

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

`subject_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 subject_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 `subject_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 ghobject_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。

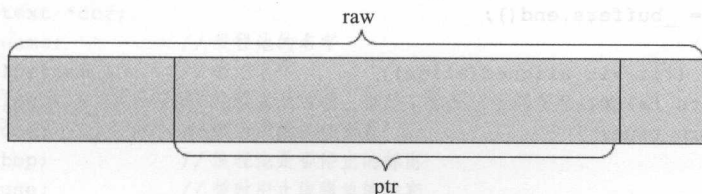


图 2-1 raw 和 ptr 示意图

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();
}

```

本PDF仅是样章,书籍版权归著者和出版社所有，未经允许不能在网上传播，如有需要，请尽量购买正版实体书，以表示对知识的尊重！实在有必要获取完整版本PDF，请联系QQ:2856202282,谢谢！

2856202282