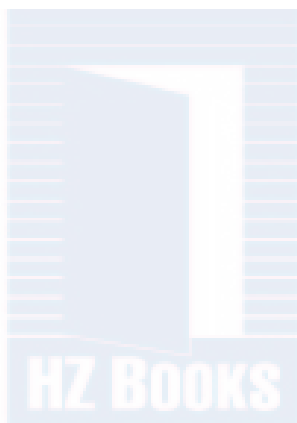


# 初识 Ceph



## 1.1 Ceph 概述

### 1. Ceph 简介

从 2004 年提交第一行代码开始到现在，Ceph 已经是一个有着十年之久的分布式存储系统软件，目前 Ceph 已经发展为开源存储界的当红明星，当然这与它的设计思想以及 OpenStack 的推动有关。

“Ceph is a unified, distributed storage system designed for excellent performance, reliability and scalability.” 这句话说出了 Ceph 的特性，它是可靠的、可扩展的、统一的、分布式的存储系统。Ceph 可以同时提供对象存储 RADOSGW（Reliable、Autonomic、Distributed、Object Storage Gateway）、块存储 RBD（Rados Block Device）、文件系统存储 Ceph FS（Ceph Filesystem）3 种功能，以此来满足不同的应用需求。

Ceph 消除了对系统单一中心节点的依赖，从而实现了真正的无中心结构的设计思想，这也是其他分布式存储系统所不能比的。通过后续章节内容的介绍，你可以看到，Ceph 几乎所有优秀特性的实现，都与其核心设计思想有关。

OpenStack 是目前最为流行的开源云平台软件。Ceph 的飞速发展离不开 OpenStack 的带动。目前而言，Ceph 已经成为 OpenStack 的标配开源存储方案之一，其实际应用主要涉及块存储和对象存储，并且开始向文件系统领域扩展。这一部分的相关情况，在后续章节中也将进行介绍。

### 2. Ceph 的发展

Ceph 是加州大学 Santa Cruz 分校的 Sage Weil（DreamHost 的联合创始人）专为博士论文设计的新一代自由软件分布式文件系统。

2004 年，Ceph 项目开始，提交了第一行代码。

2006 年，OSDI 学术会议上，Sage 发表了介绍 Ceph 的论文，并在该篇论文的末尾提供了 Ceph 项目的下载链接。

2010 年，Linus Torvalds 将 Ceph Client 合并到内核 2.6.34 中，使 Linux 与 Ceph 磨合度更高。

2012 年，拥抱 OpenStack，进入 Cinder 项目，成为重要的存储驱动。

2014 年，Ceph 正赶上 OpenStack 大热，受到各大厂商的“待见”，吸引来自不同厂商越来越多的开发者加入，Intel、SanDisk 等公司都参与其中，同时 Inktank 公司被 Red Hat 公司 1.75 亿美元收购。

2015 年，Red Hat 宣布成立 Ceph 顾问委员会，成员包括 Canonical、CERN、Cisco、Fujitsu、Intel、SanDisk 和 SUSE。Ceph 顾问委员会将负责 Ceph 软件定义存储项目的广泛议题，目标是使 Ceph 成为云存储系统。

2016 年，OpenStack 社区调查报告公布，Ceph 仍为存储首选，这已经是 Ceph 第 5 次位居调查的首位了。

### 3. Ceph 应用场景

Ceph 可以提供对象存储、块设备存储和文件系统服务，其对象存储可以对接网盘（owncloud）应用业务等；其块设备存储可以对接（IaaS），当前主流的 IaaS 云平台软件，

例如 OpenStack、CloudStack、Zstack、Eucalyptus 等以及 KVM 等，本书后续章节中将介绍 OpenStack、CloudStack、Zstack 和 KVM 的对接；其文件系统文件尚不成熟，官方不建议在生产环境下使用。

4. Ceph 生态系统

Ceph 作为开源项目，其遵循 LGPL 协议，使用 C++ 语言开发，目前 Ceph 已经成为最广泛的全球开源软件定义存储项目，拥有得到众多 IT 厂商支持的协同开发模式。目前 Ceph 社区有超过 40 个公司的上百名开发者持续贡献代码，平均每星期的代码 commits 超过 150 个，每个版本通常在 2 000 个 commits 左右，代码增减行数在 10 万行以上。在过去的几个版本发布中，贡献者的数量和参与公司明显增加，如图 1-1 所示。



图 1-1 部分厂商和软件

5. Ceph 用户群

Ceph 成为了开源存储的当红明星，国内外已经拥有众多用户群体，下面简单说一下 Ceph 的用户群。

(1) 国外用户群

1) CERN：CERN IT 部门在 2013 年年中开始就运行了一个单一集群超过 10 000 个 VM 和 100 000 个 CPU Cores 的云平台，主要用来做物理数据分析。这个集群后端 Ceph 包括 3PB 的原始容量，在云平台中作为 1000 多个 Cinder 卷和 1500 多个 Glance 镜像的存储

池。在 2015 年开始测试单一 30 PB 的块存储 RBD 集群。

2) DreamHost : DreamHost 从 2012 年开始运行基于 Ceph RADOSGW 的大规模对象存储集群, 单一集群在 3PB 以下, 大约由不到 10 机房集群组成, 直接为客户提供对象存储服务。

3) Yahoo Flick : Yahoo Flick 自 2013 年开始逐渐试用 Ceph 对象存储替换原有的商业存储, 目前大约由 10 机房构成, 每个机房在 1PB ~ 2PB, 存储了大约 2 500 亿个对象。

4) 大学用户: 奥地利的因斯布鲁克大学、法国的洛林大学等。

### (2) 国内用户群

1) 以 OpenStack 为核心的云厂商: 例如 UnitedStack、Awcloud 等国内云计算厂商。

2) Ceph 产品厂商: SanDisk、XSKY、H3C、杉岩数据、SUSE 和 Bigtera 等 Ceph 厂商。

3) 互联网企业: 腾讯、京东、新浪微博、乐视、完美世界、平安科技、联想、唯品会、福彩网和魅族等国内互联网企业。

## 6. 社区项目开发迭代

目前 Ceph 社区采用每半年一个版本发布的方式来进行特性和功能的开发, 每个版本发布需要经历设计、开发、新功能冻结, 持续若干个版本的 Bug 修复周期后正式发布下一个稳定版本。其发布方式跟 OpenStack 差不多, 也是每半年发布一个新版本。

Ceph 会维护多个稳定版本来保证持续的 Bug 修复, 以此来保证用户的存储安全, 同时社区会有一个发布稳定版本的团队来维护已发布的版本, 每个涉及之前版本的 Bug 都会被该团队移植回稳定版本, 并且经过完整 QA 测试后发布下一个稳定版本。

代码提交都需要经过单元测试, 模块维护者审核, 并通过 QA 测试子集后才能合并到主线。社区维护一个较大规模的测试集群来保证代码质量, 丰富的测试案例和错误注入机制保证了项目的稳定可靠。

## 7. Ceph 版本

Ceph 正处于持续开发中并且迅速提升。2012 年 7 月 3 日, Sage 发布了 Ceph 第一个 LTS 版本: Argonaut。从那时起, 陆续又发布了 9 个新版本。Ceph 版本被分为 LTS (长期

稳定版) 以及开发版本, Ceph 每隔一段时间就会发布一个长期稳定版。Ceph 版本具体信息见表 1-1。欲了解更多信息, 请访问 <https://Ceph.com/category/releases/>。

表 1-1 Ceph 版本信息

Ceph 版本名称	Ceph 版本号	发布时间
Argonaut	V0.48 (LTS)	2012.6.3
Bobtail	V0.56 (LTS)	2013.1.1
Cuttlefish	V0.61	2013.5.7
Dumpling	V0.67 (LTS)	2013.8.14
Emperor	V0.72	2013.11.9
Firefly	V0.80 (LTS)	2014.3.7
Giant	V0.87.1	2015.2.26
Hammer	V0.94 (LTS)	2015.4.7
Infernalis	V9.0.0	2015.5.5
Jewel	V10.0.0	2015.11
Jewel	V10.2.0	2016.3

1.2 Ceph 的功能组件

Ceph 提供了 RADOS、OSD、MON、LIBRADOS、RBD、RGW 和 Ceph FS 等功能组件, 但其底层仍然使用 RADOS 存储来支撑上层的那些组件, 如图 1-2 所示。

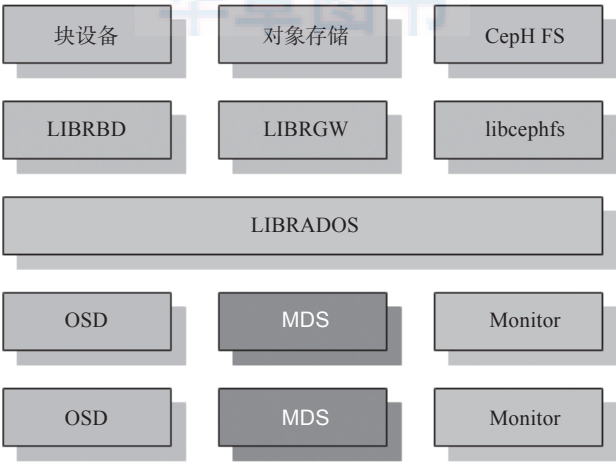


图 1-2 Ceph 功能组件的整体架构

下面分为两部分来讲述 Ceph 的功能组件。

### (1) Ceph 核心组件

在 Ceph 存储中，包含了几个重要的核心组件，分别是 Ceph OSD、Ceph Monitor 和 Ceph MDS。一个 Ceph 的存储集群至少需要一个 Ceph Monitor 和至少两个 Ceph 的 OSD。运行 Ceph 文件系统的客户端时，Ceph 的元数据服务器（MDS）是必不可少的。下面来详细介绍一下各个核心组件。

- ❑ Ceph OSD：全称是 Object Storage Device，主要功能包括存储数据，处理数据的复制、恢复、回补、平衡数据分布，并将一些相关数据提供给 Ceph Monitor，例如 Ceph OSD 心跳等。一个 Ceph 的存储集群，至少需要两个 Ceph OSD 来实现 active + clean 健康状态和有效的保存数据的双副本（默认情况下是双副本，可以调整）。注意：每一个 Disk、分区都可以成为一个 OSD。
- ❑ Ceph Monitor：Ceph 的监控器，主要功能是维护整个集群健康状态，提供一致性的决策，包含了 Monitor map、OSD map、PG（Placement Group）map 和 CRUSH map。
- ❑ Ceph MDS：全称是 Ceph Metadata Server，主要保存的是 Ceph 文件系统（File System）的元数据（metadata）。温馨提示：Ceph 的块存储和 Ceph 的对象存储都不需要 Ceph MDS。Ceph MDS 为基于 POSIX 文件系统的用户提供了一些基础命令，例如 ls、find 等命令。

### (2) Ceph 功能特性

Ceph 可以同时提供对象存储 RADOSGW（Reliable、Autonomic、Distributed、Object Storage Gateway）、块存储 RBD（Rados Block Device）、文件系统存储 Ceph FS（Ceph File System）3 种功能，由此产生了对应的实际场景，本节简单介绍如下。

RADOSGW 功能特性基于 LIBRADOS 之上，提供当前流行的 RESTful 协议的网关，并且兼容 S3 和 Swift 接口，作为对象存储，可以对接网盘类应用以及 HLS 流媒体应用等。

RBD（Rados Block Device）功能特性也是基于 LIBRADOS 之上，通过 LIBRBD 创建一个块设备，通过 QEMU/KVM 附加到 VM 上，作为传统的块设备来用。目前 OpenStack、CloudStack 等都是采用这种方式来为 VM 提供块设备，同时也支持快照、COW（Copy On Write）等功能。

Ceph FS（Ceph File System）功能特性是基于 RADOS 来实现分布式的文件系统，引入了 MDS（Metadata Server），主要为兼容 POSIX 文件系统提供元数据。一般都是当做文件系统来挂载。

后面章节会对这几种特性以及对应的实际场景做详细的介绍和分析。

### 1.3 Ceph 架构和设计思想

#### 1. Ceph 架构

Ceph 底层核心是 RADOS。Ceph 架构图如图 1-3 所示。

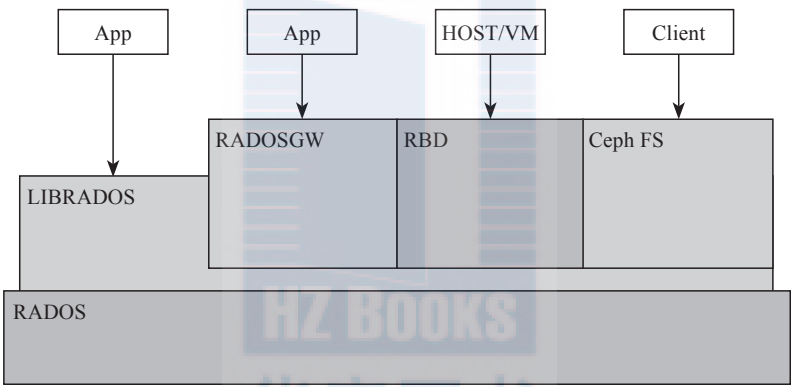


图 1-3 Ceph 架构图

- ❑ RADOS：RADOS 具备自我修复等特性，提供了一个可靠、自动、智能的分布式存储。
- ❑ LIBRADOS：LIBRADOS 库允许应用程序直接访问，支持 C/C++、Java 和 Python 等语言。
- ❑ RADOSGW：RADOSGW 是一套基于当前流行的 RESTful 协议的网关，并且兼容 S3 和 Swift。
- ❑ RBD：RBD 通过 Linux 内核（Kernel）客户端和 QEMU/KVM 驱动，来提供一个完全分布式的块设备。
- ❑ Ceph FS：Ceph FS 通过 Linux 内核（Kernel）客户端结合 FUSE，来提供一个兼容 POSIX 的文件系统。

具体的 RADOS 细节以及 RADOS 的灵魂 CRUSH (Controlled Replication Under Scalable Hashing, 可扩展哈希算法的可控复制) 算法, 这两个知识点会在后面的第 2、3 章详细介绍和分析。

## 2. Ceph 设计思想

Ceph 是一个典型的起源于学术研究课题的开源项目。虽然学术研究生涯对于 Sage 而言只是其光辉事迹的短短一篇, 但毕竟还是有几篇学术论文可供参考的。可以根据 Sage 的几篇论文分析 Ceph 的设计思想。

理解 Ceph 的设计思想, 首先还是要了解 Sage 设计 Ceph 时所针对的应用场景, 换句话说, Sage 当初做 Ceph 的初衷的什么?

事实上, Ceph 最初针对的应用场景, 就是大规模的、分布式的存储系统。所谓“大规模”和“分布式”, 至少是能够承载 PB 级别的数据和成千上万的存储节点组成的存储集群。

如今云计算、大数据在中国发展得如火如荼, PB 容量单位早已经进入国内企业存储采购单, DT 时代即将来临。Ceph 项目起源于 2004 年, 那是一个商用处理器以单核为主流, 常见硬盘容量只有几十 GB 的年代。当时 SSD 也没有大规模商用, 正因如此, Ceph 之前版本对 SSD 的支持不是很好, 发挥不了 SSD 的性能。如今 Ceph 高性能面临的挑战正是这些历史原因, 目前社区和业界正在逐步解决这些性能上的限制。

在 Sage 的思想中, 我们首先说一下 Ceph 的技术特性, 总体表现在集群可靠性、集群扩展性、数据安全性、接口统一性 4 个方面。

- ❑ **集群可靠性:** 所谓“可靠性”, 首先从用户角度来说数据是第一位的, 要尽可能保证数据不会丢失。其次, 就是数据写入过程中的可靠性, 在用户将数据写入 Ceph 存储系统的过程中, 不会因为意外情况出现而造成数据丢失。最后, 就是降低不可控物理因素的可靠性, 避免因机器断电等不可控物理因素而产生的数据丢失。
- ❑ **集群可扩展性:** 这里的“可扩展”概念是广义的, 既包括系统规模和存储容量的可扩展, 也包括随着系统节点数增加的聚合数据访问带宽的线性扩展。



- ❑ **数据安全性**：所谓“数据安全性”，首先要保证由于服务器死机或者是偶然停电等自然因素的产生，数据不会丢失，并且支持数据自动恢复，自动重平衡等。总体而言，这一特性既保证了系统的高度可靠和数据绝对安全，又保证了在系统规模扩大之后，其运维难度仍能保持在一个相对较低的水平。
- ❑ **接口统一性**：所谓“接口统一”，本书开头就说到了 Ceph 可以同时支持 3 种存储，即块存储、对象存储和文件存储。Ceph 支持市面上所有流行的存储类型。

根据上述技术特性以及 Sage 的论文，我们来分析一下 Ceph 的设计思路，概述为两点：**充分发挥存储本身计算能力和去除所有的中心点。**

- ❑ **充分发挥存储设备自身的计算能力**：其实就是采用廉价的设备和具有计算能力的设备（最简单的例子就是普通的服务器）作为存储系统的存储节点。Sage 认为当前阶段只是将这些服务器当做功能简单的存储节点，从而产生资源过度浪费（如同虚拟化的思想一样，都是为了避免资源浪费）。而如果充分发挥节点上的计算能力，则可以实现前面提出的技术特性。这一点成为了 Ceph 系统的核心思想。
- ❑ **去除所有的中心点**：搞 IT 的最忌讳的就是单点故障，如果系统中出现中心点，一方面会引入单点故障，另一方面也必然面临着当系统规模扩大时的可扩展性和性能瓶颈。除此之外，如果中心点出现在数据访问的关键路径上，也必然导致数据访问的延迟增大。虽然在大多数存储软件实践中，单点故障点和性能瓶颈的问题可以通过为中心点增加 HA 或备份加以缓解，但 Ceph 系统最终采用 Crush、Hash 环等方法更彻底地解决了这个问题。很显然 Sage 的眼光和设想还是很超前的。

## 1.4 Ceph 快速安装

在 Ceph 官网上提供了两种安装方式：快速安装和手动安装。快速安装采用 Ceph-Deploy 工具来部署；手动安装采用官方教程一步一步来安装部署 Ceph 集群，过于烦琐但有助于加深印象，如同手动部署 OpenStack 一样。但是，建议新手和初学者采用第一种方式快速部署并且测试，下面会介绍如何使用 Ceph-Deploy 工具来快速部署 Ceph 集群。

### 1.4.1 Ubuntu/Debian 安装

本节将介绍如何使用 Ceph-Deploy 工具来快速部署 Ceph 集群，开始之前先普及一下 Ceph-Deploy 工具的知识。Ceph-Deploy 工具通过 SSH 方式连接到各节点服务器上，通过执行一系列脚本来完成 Ceph 集群部署。Ceph-Deploy 简单易用同时也是 Ceph 官网推荐的默认安装工具。本节先来讲下在 Ubuntu/Debian 系统下如何快速安装 Ceph 集群。

#### 1) 配置 Ceph APT 源。

```
root@localhost~# echo deb http://ceph.com/debian-{ceph-stable-release}/  
$(lsb_release -sc) main | sudo tee /etc/apt/sources.list.d/ceph.list
```

#### 2) 添加 APT 源 key。

```
root@localhost:~# wget -q -O - 'https://ceph.com/git/?p=ceph.git;a=blob_  
plain;f=keys/release.asc' | sudo apt-key add -
```

#### 3) 更新源并且安装 ceph-deploy。

```
root@localhost:~# sudo apt-get update &&sudo apt-get install ceph-deploy -y
```

#### 4) 配置各个节点 hosts 文件。

```
root@localhost:~# cat /etc/hosts  
192.168.1.2  node1  
192.168.1.3  node2  
192.168.1.4  node3
```

5) 配置各节点 SSH 无密码登录，这就是本节开始时讲到的 Ceph-Deploy 工具要用过 SSH 方式连接到各节点服务器，来安装部署集群。输完 ssh-keygen 命令之后，在命令行会输出以下内容。

```
root@localhost:~# ssh-keygen  
Generating public/private key pair.  
Enter file in which to save the key (/ceph-client/.ssh/id_rsa):  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /ceph-client/.ssh/id_rsa.  
Your public key has been saved in /ceph-client/.ssh/id_rsa.pub
```

#### 6) 复制 key 到各节点。

```
root@localhost:~# ssh-copy-id node1
```

```
root@localhost:~# ssh-copy-id node2
root@localhost:~# ssh-copy-id node3
```

7) 在执行 `ceph-deploy` 的过程中会生成一些配置文件，建议创建一个目录，例如 `my-cluster`。

```
root@localhost:~# mkdir my-cluster
root@localhost:~# cd my-cluster
```

8) 创建集群 (Cluster)，部署新的 monitor 节点。

```
root@localhost:~# ceph-deploy new {initial-monitor-node(s)}
```

例如：

```
root@localhost:~# ceph-deploy new node1
```

9) 配置 `Ceph.conf` 配置文件，示例文件是默认的，可以根据自己情况进行相应调整和添加。具体优化情况本书后面会介绍。

```
[global]
fsid = 67d997c9-dc13-4edf-a35f-76fd693aa118
mon_initial_members = node1, node2
mon_host = 192.168.1.2, 192.168.1.3
auth_cluster_required = cephx
auth_service_required = cephx
auth_client_required = cephx
filestore_xattr_use_omap = true
<!-- 以上部分都是 ceph-deploy 默认生成的 -->
public network = {ip-address}/{netmask}
cluster network={ip-address}/{netmask}
<!-- 以上两个网络是新增部分，默认只是添加 public network，一般生产都是定义两个网络，集群网络和数据网络分开 -->
[osd]
.....
[mon]
.....
```

这里配置文件不再过多叙述。

10) 安装 Ceph 到各节点。

```
root@localhost:~# ceph-deploy install {ceph-node} [{ceph-node} ...]
```

例如：

## 12 ❖ Ceph 分布式存储实战

```
root@localhost:~# ceph-deploy install node1 node2 node3
```

11) 获取密钥 key, 会在 my-cluster 目录下生成几个 key。

```
root@localhost:~# ceph-deploy mon create-initial
```

12) 初始化磁盘。

```
root@localhost:~# ceph-deploy disk zap {osd-server-name}:{disk-name}
```

例如:

```
root@localhost:~# ceph-deploy disk zap node1:sdb
```

13) 准备 OSD。

```
root@localhost:~# ceph-deploy osd prepare {node-name}:{data-disk}[:{journal-disk}]
```

例如:

```
root@localhost:~# ceph-deploy osd prepare node1:sdb1:sdc
```

14) 激活 OSD。

```
root@localhost:~# ceph-deploy osd activate {node-name}:{data-disk-partition}[:{journal-disk-partition}]
```

例如:

```
root@localhost:~# ceph-deploy osd activate node1:sdb1:sdc
```

15) 分发 key。

```
root@localhost:~# ceph-deploy admin {admin-node} {ceph-node}
```

例如:


```
root@localhost:~# ceph-deploy admin node1 node2 node3
```

16) 给 admin key 赋权限。

```
root@localhost:~# sudo chmod +r /etc/ceph/ceph.client.admin.keyring
```

17) 查看集群健康状态, 如果是 active+clean 状态就是正常的。

```
root@localhost:~# ceph health
```

 提示 安装 Ceph 前提条件如下。

- ① 时间要求很高，建议在部署 Ceph 集群的时候提前配置好 NTP 服务器。
- ② 对网络要求一般，因为 Ceph 源在外国有时候会被屏蔽，解决办法多尝试机器或者代理。

## 1.4.2 RHEL/CentOS 安装

本节主要讲一下在 RHEL/CentOS 系统下如何快速安装 Ceph 集群。

1) 配置 Ceph YUM 源。

```
root@localhost:~# vim /etc/yum.repos.d/ceph.repo
[ceph-noarch]
name=Cephnoarch packages
baseurl=http://ceph.com/rpm-{ceph-release}/{distro}/noarch
enabled=1
gpgcheck=1
type=rpm-md
gpgkey=https://ceph.com/git/?p=ceph.git;a=blob_plain;f=keys/release.asc
```

2) 更新源并且安装 ceph-deploy。

```
root@localhost:~# yum update &&yum install ceph-deploy -y
```

3) 配置各个节点 hosts 文件。

```
root@localhost:~# cat /etc/hosts
192.168.1.2  node1
192.168.1.3  node2
192.168.1.4  node3
```

4) 配置各节点 SSH 无密码登录，通过 SSH 方式连接到各节点服务器，以安装部署集群。输入 ssh-keygen 命令，在命令行会输出以下内容。

```
root@localhost:~# ssh-keygen
Generating public/private key pair.
Enter file in which to save the key (/ceph-client/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /ceph-client/.ssh/id_rsa.
Your public key has been saved in /ceph-client/.ssh/id_rsa.pub
```

5) 拷贝 key 到各节点。

```
root@localhost:~# ssh-copy-id node1
root@localhost:~# ssh-copy-id node2
root@localhost:~# ssh-copy-id node3
```

6) 在执行 ceph-deploy 的过程中会生成一些配置文件，建议创建一个目录，例如 my-cluster。

```
root@localhost:~# mkdir my-cluster
root@localhost:~# cd my-cluster
```

7) 创建集群 (Cluster)，部署新的 monitor 节点。

```
root@localhost:~# ceph-deploy new {initial-monitor-node(s)}
```

例如：

```
root@localhost:~# ceph-deploy new node1
```

8) 配置 Ceph.conf 配置文件，示例文件是默认的，可以根据自己情况进行相应调整和添加。具体优化情况本书后面会介绍。

```
[global]
fsid = 67d997c9-dc13-4edf-a35f-76fd693aa118
mon_initial_members = node1, node2
mon_host = 192.168.1.2, 192.168.1.3
auth_cluster_required = cephx
auth_service_required = cephx
auth_client_required = cephx
filestore_xattr_use_omap = true
<!------- 以上部分都是 ceph-deploy 默认生成的 ----->
public network = {ip-address}/{netmask}
cluster network={ip-addesss}/{netmask}
<!------- 以上两个网络是新增部分，默认只是添加 public network，一般生产都是定义两个网络，集
群网络和数据网络分开 ----->
[osd]
.....
[mon]
.....
```

这里配置文件不再过多叙述。

9) 安装 Ceph 到各节点。

```
root@localhost:~# ceph-deploy install {ceph-node} [{ceph-node} ...]
```

例如：

```
root@localhost:~# ceph-deploy install node1 node2 node3
```

10) 获取密钥 key，会在 my-cluster 目录下生成几个 key。

```
root@localhost:~# ceph-deploy mon create-initial
```

11) 初始化磁盘。

```
root@localhost:~# ceph-deploy disk zap {osd-server-name}:{disk-name}
```

例如：

```
root@localhost:~# ceph-deploy disk zap node1:sdb
```

12) 准备 OSD。

```
root@localhost:~# ceph-deploy osd prepare {node-name}:{data-disk}[:{journal-disk}]
```

例如：

```
root@localhost:~# ceph-deploy osd prepare node1:sdb1:sdc
```

13) 激活 OSD。

```
root@localhost:~# ceph-deploy osd activate {node-name}:{data-disk-partition}
[:{journal-disk-partition}]
```

例如：

```
root@localhost:~# ceph-deploy osd activate node1:sdb1:sdc
```

14) 分发 key。

```
root@localhost:~# ceph-deploy admin {admin-node} {ceph-node}
```

例如：

```
root@localhost:~# ceph-deploy admin node1 node2 node3
```

15) 给 admin key 赋权限。

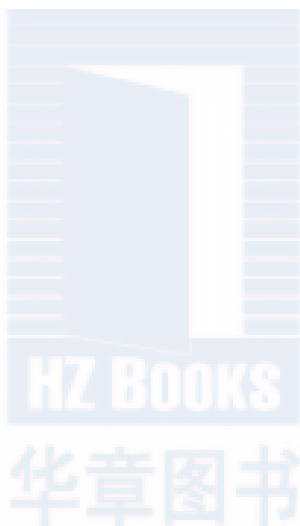
```
root@localhost:~# sudo chmod +r /etc/ceph/ceph.client.admin.keyring
```

16) 查看集群健康状态，如果是 active + clean 状态就是正常的。

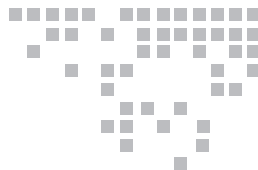
```
root@localhost:~# ceph health
```

## 1.5 本章小结

本章主要从 Ceph 的历史背景、发展事件、Ceph 的架构组件、功能特性以及 Ceph 的设计思想方面介绍了 Ceph，让大家对 Ceph 有一个全新的认识，以便后面更深入地了解 Ceph。







## 存储基石 RADOS

分布式对象存储系统 RADOS 是 Ceph 最为关键的技术，它是一个支持海量存储对象的分布式对象存储系统。RADOS 层本身就是一个完整的对象存储系统，事实上，所有存储在 Ceph 系统中的用户数据最终都是由这一层来存储的。而 Ceph 的高可靠、高可扩展、高性能、高自动化等特性，本质上也是由这一层所提供的。因此，理解 RADOS 是理解 Ceph 的基础与关键。

Ceph 的设计哲学如下。

- ❑ 每个组件必须可扩展。
- ❑ 不存在单点故障。
- ❑ 解决方案必须是基于软件的。
- ❑ 可摆脱专属硬件的束缚即可运行在常规硬件上。
- ❑ 推崇自我管理。

由第 1 章的讲解可以知道，Ceph 包含以下组件。

- ❑ 分布式对象存储系统 RADOS 库，即 LIBRADOS。
- ❑ 基于 LIBRADOS 实现的兼容 Swift 和 S3 的存储网关系统 RADOSGW。
- ❑ 基于 LIBRADOS 实现的块设备驱动 RBD。

- ❑ 兼容 POSIX 的分布式文件 Ceph FS。
- ❑ 最底层的分布式对象存储系统 RADOS。

## 2.1 Ceph 功能模块与 RADOS

Ceph 中的这些组件与 RADOS 有什么关系呢，笔者手绘了一张简单的 Ceph 架构图，我们结合图 2-1 来分析这些组件与 RADOS 的关系。

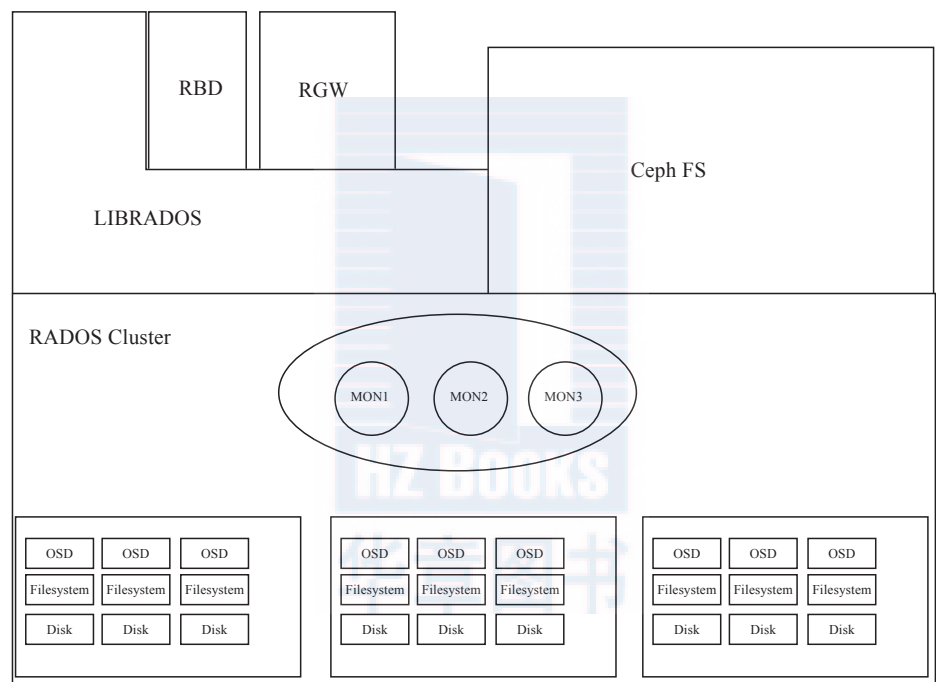


图 2-1 Ceph 架构图

Ceph 存储系统的逻辑层次结构大致划分为 4 部分：基础存储系统 RADOS、基于 RADOS 实现的 Ceph FS，基于 RADOS 的 LIBRADOS 层应用接口、基于 LIBRADOS 的应用接口 RBD、RADOSGW。Ceph 架构（见图 1-1）我们在第 1 章有过初步的了解，这里详细看一下各个模块的功能，以此了解 RADOS 在整个 Ceph 起到的作用。

### （1）基础存储系统 RADOS

RADOS 这一层本身就是一个完整的对象存储系统，事实上，所有存储在 Ceph 系统

中的用户数据最终都是由这一层来存储的。Ceph 的很多优秀特性本质上也是借由这一层设计提供。理解 RADOS 是理解 Ceph 的基础与关键。物理上，RADOS 由大量的存储设备节点组成，每个节点拥有自己的硬件资源（CPU、内存、硬盘、网络），并运行着操作系统和文件系统。本书后续章节将对 RADOS 进行深入介绍。

## （2）基础库 LIBRADOS

LIBRADOS 层的功能是对 RADOS 进行抽象和封装，并向上层提供 API，以便直接基于 RADOS 进行应用开发。需要指明的是，RADOS 是一个对象存储系统，因此，LIBRADOS 实现的 API 是针对对象存储功能的。RADOS 采用 C++ 开发，所提供的原生 LIBRADOS API 包括 C 和 C++ 两种。物理上，LIBRADOS 和基于其上开发的应用位于同一台机器，因而也被称为本地 API。应用调用本机上的 LIBRADOS API，再由后者通过 socket 与 RADOS 集群中的节点通信并完成各种操作。

## （3）上层应用接口

Ceph 上层应用接口涵盖了 RADOSGW（RADOS Gateway）、RBD（Reliable Block Device）和 Ceph FS（Ceph File System），其中，RADOSGW 和 RBD 是在 LIBRADOS 库的基础上提供抽象层次更高、更便于应用或客户端使用的上层接口。

其中，RADOSGW 是一个提供与 Amazon S3 和 Swift 兼容的 RESTful API 的网关，以供相应的对象存储应用开发使用。RADOSGW 提供的 API 抽象层次更高，但在类 S3 或 Swift LIBRADOS 的管理比便捷，因此，开发者应针对自己的需求选择使用。RBD 则提供了一个标准的块设备接口，常用于在虚拟化的场景下为虚拟机创建 volume。目前，Red Hat 已经将 RBD 驱动集成在 KVM/QEMU 中，以提高虚拟机访问性能。

## （4）应用层

应用层就是不同场景下对于 Ceph 各个应用接口的各种应用方式，例如基于 LIBRADOS 直接开发的对象存储应用，基于 RADOSGW 开发的对象存储应用，基于 RBD 实现的云主机硬盘等。

下面就来看看 RADOS 的架构。

## 2.2 RADOS 架构

RADOS 系统主要由两个部分组成，如图 2-2 所示。

1) OSD：由数目可变的大规模 OSD（Object Storage Devices）组成的集群，负责存储所有的 Objects 数据。

2) Monitor：由少量 Monitors 组成的强耦合、小规模集群，负责管理 Cluster Map。其中，Cluster Map 是整个 RADOS 系统的关键数据结构，管理集群中的所有成员、关系和属性等信息以及数据的分发。

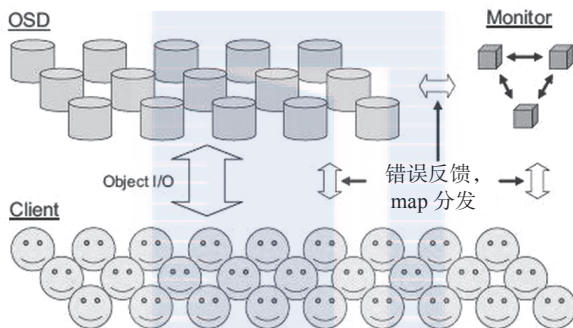


图 2-2 RADOS 系统架构图示

对于 RADOS 系统，节点组织管理和数据分发策略均由内部的 Mon 全权负责，因此，从 Client 角度设计相对比较简单，它给应用提供存储接口。

### 2.2.1 Monitor 介绍

正如其名，Ceph Monitor 是负责监视整个群集的运行状况的，这些信息都是由维护集群成员的守护程序来提供的，如各个节点之间的状态、集群配置信息。Ceph monitor map 包括 OSD Map、PG Map、MDS Map 和 CRUSH 等，这些 Map 被统称为集群 Map。

1) Monitor Map。Monitor Map 包括有关 monitor 节点端到端的信息，其中包括 Ceph 集群 ID，监控主机名和 IP 地址和端口号，它还存储了当前版本信息以及最新更改信息，可以通过以下命令查看 monitor map。

```
#ceph mon dump
```

2) OSD Map。OSD Map 包括一些常用的信息，如集群 ID，创建 OSD Map 的版本信息和最后修改信息，以及 pool 相关信息，pool 的名字、pool 的 ID、类型，副本数目以及 PGP，还包括 OSD 信息，如数量、状态、权重、最新的清洁间隔和 OSD 主机信息。可以通过执行以下命令查看集群的 OSD Map。

```
#ceph osd dump
```

3) PG Map。PG Map 包括当前 PG 版本、时间戳、最新的 OSD Map 的版本信息、空间使用比例，以及接近占满比例信息，同时，也包括每个 PG ID、对象数目、状态、OSD 的状态以及深度清理的详细信息，可以通过以下命令来查看 PG Map。

```
#ceph pg dump
```

4) CRUSH Map。CRUSH Map 包括集群存储设备信息，故障域层次结构和存储数据时定义失败域规则信息；可以通过以下命令查看 CRUSH Map。

```
#ceph osd crush dump
```

5) MDS Map。MDS Map 包括存储当前 MDS Map 的版本信息、创建当前 Map 的信息、修改时间、数据和元数据 POOL ID、集群 MDS 数目和 MDS 状态，可通过以下命令查看集群 MDS Map 信息。

```
#ceph mds dump
```

Ceph 的 MON 服务利用 Paxos 的实例，把每个映射图存储为一个文件。Ceph Monitor 并未为客户提供数据存储服务，而是为 Ceph 集群维护着各类 Map，并服务更新群集映射到客户机以及其他集群节点。客户端和其他群集节点定期检查并更新于 Monitor 的集群 Map 最新的副本。

Ceph Monitor 是个轻量级的守护进程，通常情况下并不需要大量的系统资源，低成本、入门级的 CPU，以及千兆网卡即可满足大多数的场景；与此同时，Monitor 节点需要有足够的磁盘空间来存储集群日志，健康集群产生几 MB 到 GB 的日志；然而，如果存储的需求增加时，打开低等级的日志信息的话，可能需要几个 GB 的磁盘空间来存储日志。

一个典型的 Ceph 集群包含多个 Monitor 节点。一个多 Monitor 的 Ceph 的架构通过法定人数来选择 leader，并在提供一致分布式决策时使用 Paxos 算法集群。在 Ceph 集群中有多个 Monitor 时，集群的 Monitor 应该是奇数；最起码的要求是一台监视器节点，这里推荐

Monitor 个数是 3。由于 Monitor 工作在法定人数，一半以上的总监视器节点应该总是可用的，以应对死机等极端情况，这是 Monitor 节点为  $N$  ( $N>0$ ) 个且  $N$  为奇数的原因。所有集群 Monitor 节点，其中一个节点为 Leader。如果 Leader Monitor 节点处于不可用状态，其他监视器节点有资格成为 Leader。生产集群必须至少有  $N/2$  个监控节点提供高可用性。

### 2.2.2 Ceph OSD 简介

Ceph OSD 是 Ceph 存储集群最重要的组件，Ceph OSD 将数据以对象的形式存储到集群中每个节点的物理磁盘上，完成存储用户数据的工作绝大多数都是由 OSD daemon 进程来实现的。

Ceph 集群一般情况都包含多个 OSD，对于任何读写操作请求，Client 端从 Ceph Monitor 获取 Cluster Map 之后，Client 将直接与 OSD 进行 I/O 操作的交互，而不再需要 Ceph Monitor 干预。这使得数据读写过程更为迅速，因为这些操作过程不像其他存储系统，它没有其他额外的层级数据处理。

Ceph 的核心功能特性包括高可靠、自动平衡、自动恢复和一致性。对于 Ceph OSD 而言，基于配置的副本数，Ceph 提供通过分布在多节点上的副本来实现，使得 Ceph 具有高可用性以及容错性。在 OSD 中的每个对象都有一个主副本，若干个从副本，这些副本默认情况下是分布在不同节点上的，这就是 Ceph 作为分布式存储系统的集中体现。每个 OSD 都可能作为某些对象的主 OSD，与此同时，它也可能作为某些对象的从 OSD，从 OSD 受到主 OSD 的控制，然而，从 OSD 在某些情况也可能成为主 OSD。在磁盘故障时，Ceph OSD Daemon 的智能对等机制将协同其他 OSD 执行恢复操作。在此期间，存储对象副本的从 OSD 将被提升为主 OSD，与此同时，新的从副本将重新生成，这样就保证了 Ceph 的可靠和一致。

Ceph OSD 架构实现由物理磁盘驱动器、在其之上的 Linux 文件系统以及 Ceph OSD 服务组成。对 Ceph OSD Daemon 而言，Linux 文件系统显性地支持了其扩展属性；这些文件系统的扩展属性提供了关于对象状态、快照、元数据内部信息；而访问 Ceph OSD Daemon 的 ACL 则有助于数据管理，如图 2-3 所示。

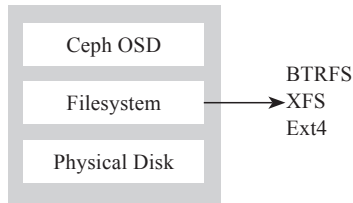


图 2-3 OSD 数据存储图

Ceph OSD 操作必须在一个有效的 Linux 分区的物理磁盘驱动器上，Linux 分区可以是 BTRFS、XFS 或者 EXT4 分区，文件系统是对性能基准测试的主要标准之一，下面来逐一了解。

1) BTRFS：在 BTRFS 文件系统的 OSD 相比于 XFS 和 EXT4 提供了最好的性能。BTRFS 的主要优点有以下 4 点。

- ❑ 扩展性 (scalability)：BTRFS 最重要的设计目标是应对大型机器对文件系统的扩展性要求。Extent、B-Tree 和动态 inode 创建等特性保证了 BTRFS 在大型机器上仍有卓越的表现，其整体性能不会随着系统容量的增加而降低。
- ❑ 数据一致性 (data integrity)：当系统面临不可预料的硬件故障时，BTRFS 采用 COW 事务技术来保证文件系统的一致性。BTRFS 还支持校验和，避免了 silent corrupt (未知错误) 的出现。而传统文件系统无法做到这一点。
- ❑ 多设备管理相关的特性：BTRFS 支持创建快照 (snapshot) 和克隆 (clone)。BTRFS 还能够方便地管理多个物理设备，使得传统的卷管理软件变得多余。
- ❑ 结合 Ceph，BTRFS 中的诸多优点中的快照，Journal of Parallel (并行日志) 等优势在 Ceph 中表现得尤为突出，不幸的是，BTRFS 还未能到达生产环境要求的健壮要求。暂不推荐用于 Ceph 集群的生产使用。

2) XFS：一种高性能的日志文件系统，XFS 特别擅长处理大文件，同时提供平滑的数据传输。目前 CentOS 7 也将 XFS+LVM 作为默认的文件系统。XFS 的主要优点如下。

- ❑ 分配组：XFS 文件系统内部被分为多个“分配组”，它们是文件系统中等长线性存储区。每个分配组各自管理自己的 inode 和剩余空间。文件和文件夹可以跨越分配组。这一机制为 XFS 提供了可伸缩性和并行特性——多个线程和进程可以同时在一个文件系统上执行 I/O 操作。这种由分配组带来的内部分区机制在一个文件系统跨越多个物理设备时特别有用，使得优化对下级存储部件的吞吐量利用率成为可能。
- ❑ 条带化分配：在条带化 RAID 阵列上创建 XFS 文件系统时，可以指定一个“条带化数据单元”。这可以保证数据分配、inode 分配，以及内部日志被对齐到该条带单元上，以此最大化吞吐量。
- ❑ 基于 Extent 的分配方式：XFS 文件系统中的文件用到的块由变长 Extent 管理，每



一个 Extent 描述了一个或多个连续的块。对那些把文件所有块都单独列出来的文件系统来说，Extent 大幅缩短了列表。

有些文件系统用一个或多个面向块的位图管理空间分配——在 XFS 中，这种结构被由一对 B+ 树组成的、面向 Extent 的结构替代了；每个文件系统分配组（AG）包含这样的一个结构。其中，一个 B+ 树用于索引未被使用的 Extent 的长度，另一个索引这些 Extent 的起始块。这种双索引策略使得文件系统在定位剩余空间中的 Extent 时十分高效。

❑ 扩展属性：XFS 通过实现扩展文件属性给文件提供了多个数据流，使文件可以被附加多个名 / 值对。文件名是一个最大长度为 256 字节的、以 NULL 字符结尾的可打印字符串，其他的关联值则可包含多达 64KB 的二进制数据。这些数据被进一步分入两个名字空间中，分别为 root 和 user。保存在 root 名字空间中的扩展属性只能被超级用户修改，保存在 user 名字空间中的可以被任何对该文件拥有写权限的用户修改。扩展属性可以被添加到任意一种 XFS inode 上，包括符号链接、设备节点和目录等。可以使用 attr 命令行程序操作这些扩展属性。xfsdump 和 xfsrestore 工具在进行备份和恢复时会一同操作扩展属性，而其他的大多数备份系统则会忽略扩展属性。

❑ XFS 作为一款可靠、成熟的，并且非常稳定的文件系统，基于分配组、条带化分配、基于 Extent 的分配方式、扩展属性等优势非常契合 Ceph OSD 服务的需求。美中不足的是，XFS 不能很好地处理 Ceph 写入过程的 journal 问题。

3) Ext4：第四代扩展文件系统，是 Linux 系统下的日志文件系统，是 Ext3 文件系统的后继版本。其主要特征如下。

❑ 大型文件系统：Ext4 文件系统可支持最高 1 Exbibyte 的分区与最大 16 Tebibyte 的文件。

❑ Extents：Ext4 引进了 Extent 文件存储方式，以替换 Ext2/3 使用的块映射（block mapping）方式。Extent 指的是一连串连续实体块，这种方式可以增加大型文件的效率并减少分裂文件。

❑ 日志校验和：Ext4 使用校验和特性来提高文件系统可靠性，因为日志是磁盘上被读取最频繁的部分之一。

❑ 快速文件系统检查：Ext4 将未使用的区块标记在 inode 当中，这样可以使诸如



e2fsck 之类的工具在磁盘检查时将这些区块完全跳过，而节约大量的文件系统检查的时间。这个特性已经在 2.6.24 版本的 Linux 内核中实现。

Ceph OSD 把底层文件系统的扩展属性用于表示各种形式的内部对象状态和元数据。XATTR 是以 key/value 形式来存储 xattr\_name 和 xattr\_value，并因此提供更多的标记对象元数据信息的方法。Ext4 文件系统提供不足以满足 XATTR，由于 XATTR 上存储的字节数的限制能力，从而使 Ext4 文件系统不那么受欢迎。然而，BTRFS 和 XFS 有一个比较大的限制 XATTR。

Ceph 使用日志文件系统，如增加了 BTRFS 和 XFS 的 OSD。在提交数据到后备存储器之前，Ceph 首先将数据写入称为一个单独的存储区，该区域被称为 journal，这是缓冲器分区在相同或单独磁盘作为 OSD，一个单独的 SSD 磁盘或分区，甚至一个文件文件系统。在这种机制下，Ceph 任何写入首先是日志，然后是后备存储，如图 2-4 所示。

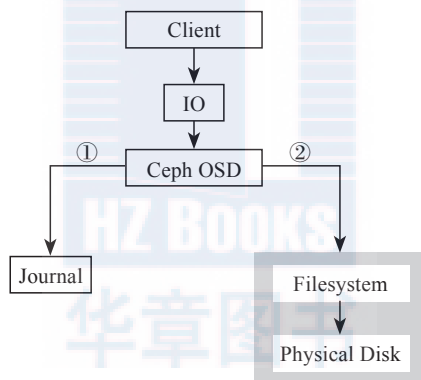


图 2-4 IO 流向图

journal 持续到后备存储同步，每隔 5s。默认情况下，10GB 是该 journal 的常用的大小，但 journal 空间越大越好。Ceph 使用 journal 综合考虑了存储速度和数据的一致性。journal 允许 Ceph OSD 功能很快做小的写操作；一个随机写入首先写入在上一个连续类型的 journal，然后刷新到文件系统。这给了文件系统足够的时间来合并写入磁盘。使用 SSD 盘作为 journal 盘能获得相对较好的性能。在这种情况下，所有的客户端写操作都写入到超高速 SSD 日志，然后刷新到磁盘。所以，一般情况下，使用 SSD 作为 OSD 的 journal 可以有效缓冲突发负载。

与传统的分布式数据存储不同，RADOS 最大的特点如下。

① 将文件映射到 Object 后，利用 Cluster Map 通过 CRUSH 计算而不是查找表方式定位文件数据到存储设备中的位置。优化了传统的文件到块的映射和 BlockMap 管理。

② RADOS 充分利用了 OSD 的智能特点，将部分任务授权给 OSD，最大程度地实现可扩展。

## 2.3 RADOS 与 LIBRADOS

LIBRADOS 模块是客户端用来访问 RADOS 对象存储设备的。Ceph 存储集群提供了消息传递层协议，用于客户端与 Ceph Monitor 与 OSD 交互，LIBRADOS 以库形式为 Ceph Client 提供了这个功能，LIBRADOS 就是操作 RADOS 对象存储的接口。所有 Ceph 客户端可以用 LIBRADOS 或 LIBRADOS 里封装的相同功能和对象存储交互，LIBRBD 和 LIBCEPHFS 就利用了此功能。你可以用 LIBRADOS 直接和 Ceph 交互（如与 Ceph 兼容的应用程序、Ceph 接口等）。下面是简单描述的步骤。

第 1 步：获取 LIBRADOS。

第 2 步：配置集群句柄。

第 3 步：创建 IO 上下文。

第 4 步：关闭连接。

LIBRADOS 架构图，如图 2-5 所示。

先根据配置文件调用 LIBRADOS 创建一个 RADOS，接下来为这个 RADOS 创建一个 radosclient，radosclient 包含 3 个主要模块（finisher、Messenger、Objector）。再根据 pool 创建对应的 iocx，在 iocx 中能够找到 radosclient。再调用 OSD 对生成对应 OSD 请求，与 OSD 进行通信响应请求。

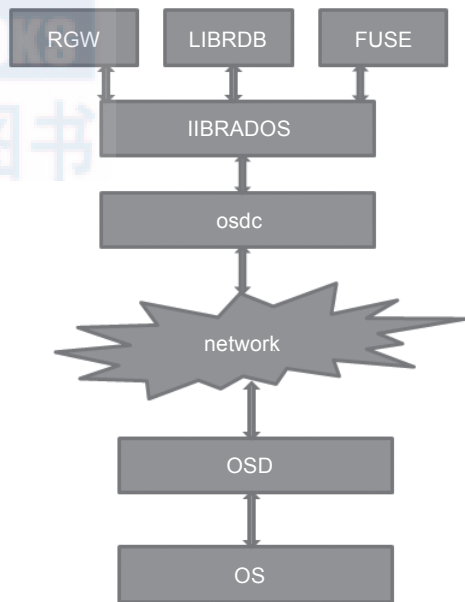


图 2-5 LIBRADOS 架构图

下面分别介绍 LIBRADOS 的 C 语言、Java 语言和 Python 语言示例。

## 1. LIBRADOS C 语言示例

下面是 LIBRADOS C 语言示例。

```
#include <stdio.h>
#include <string.h>
#include <rados/librados.h>
int main (int argc, char argv**)
{
    /* 声明集群句柄以及所需参数 */
    rados_t cluster;
    char cluster_name[] = "ceph";           // 集群名称
    char user_name[] = "client.admin";      // 指定访问集群的用户, 这里用 admin
    uint64_t flags;
    rados_ioctx_t io;                      // rados 上下文句柄
    char *poolname = "data";               // 目标 pool 名
    char read_res[100];
    char xattr[] = "en_US";
    /* 指定参数初始化句柄 */
    int err;
    err = rados_create2(&cluster, cluster_name, user_name, flags);

    if (err < 0) {
        fprintf(stderr, "%s: Couldn't create the cluster handle!
            %s\n", argv[0], strerror(-err));
        exit(EXIT_FAILURE);
    } else {
        printf("\nCreated a cluster handle.\n");
    }
    /* 读取配置文件用来配置句柄 */
    err = rados_conf_read_file(cluster, "/etc/ceph/ceph.conf");
    if (err < 0) {
        fprintf(stderr, "%s: cannot read config file: %s\n", argv[0],
            strerror(-err));
        exit(EXIT_FAILURE);
    } else {
        printf("\nRead the config file.\n");
    }

    /* 分解参数 */
    err = rados_conf_parse_argv(cluster, argc, argv);
    if (err < 0) {
        fprintf(stderr, "%s: cannot parse command line arguments:
            %s\n", argv[0], strerror(-err));
```

```

        exit(EXIT_FAILURE);
    } else {
        printf("\nRead the command line arguments.\n");
    }
    /* 连接集群 */
    err = rados_connect(cluster);
    if (err < 0) {
        fprintf(stderr, "%s: cannot connect to cluster: %s\n",
            argv[0], strerror(-err));
        exit(EXIT_FAILURE);
    } else {
        printf("\nConnected to the cluster.\n");
    }
    // 创建 rados 句柄上下文
    err = rados_ioctx_create(cluster, poolname, &io);
    if (err < 0) {
        fprintf(stderr, "%s: cannot open rados pool %s: %s\n",
            argv[0], poolname, strerror(-err));
        rados_shutdown(cluster);
        exit(EXIT_FAILURE);
    } else {
        printf("\nCreated I/O context.\n");
    }

    // 写对象
    err = rados_write(io, "hw", "Hello World!", 12, 0);
    if (err < 0) {
        fprintf(stderr, "%s: Cannot write object \"hw\" to pool %s: %s\n",
            argv[0], poolname, strerror(-err));
        rados_ioctx_destroy(io);
        rados_shutdown(cluster);
        exit(1);
    } else {
        printf("\nWrote \"Hello World\" to object \"hw\".\n");
    }
    // 设置对象属性
    err = rados_setxattr(io, "hw", "lang", xattr, 5);
    if (err < 0) {
        fprintf(stderr, "%s: Cannot write xattr to pool %s: %s\n",
            argv[0], poolname, strerror(-err));
        rados_ioctx_destroy(io);
        rados_shutdown(cluster);
        exit(1);
    } else {
        printf("\nWrote \"en_US\" to xattr \"lang\" for object \"hw\".\n");
    }
    rados_completion_t comp;
    // 确认异步 rados 句柄成功创建

```

```

err = rados_aio_create_completion(NULL, NULL, NULL, &comp);
if (err < 0) {
    fprintf(stderr, "%s: Could not create aio completion: %s\n",
            argv[0], strerror(-err));
    rados_ioctx_destroy(io);
    rados_shutdown(cluster);
    exit(1);
} else {
    printf("\nCreated AIO completion.\n");
}

/* Next, read data using rados_aio_read. */
// 异步读对象
err = rados_aio_read(io, "hw", comp, read_res, 12, 0);
if (err < 0) {
    fprintf(stderr, "%s: Cannot read object. %s %s\n", argv[0],
            poolname, strerror(-err));
    rados_ioctx_destroy(io);
    rados_shutdown(cluster);
    exit(1);
} else {
    printf("\nRead object \"hw\". The contents are:\n %s \n", read_res);
}
// 等待对象上的操作完成
rados_wait_for_complete(comp);

// 释放 complete 句柄
rados_aio_release(comp);
char xattr_res[100];
// 获取对象
err = rados_getxattr(io, "hw", "lang", xattr_res, 5);
if (err < 0) {
    fprintf(stderr, "%s: Cannot read xattr. %s %s\n", argv[0],
            poolname, strerror(-err));
    rados_ioctx_destroy(io);
    rados_shutdown(cluster);
    exit(1);
} else {
    printf("\nRead xattr \"lang\" for object \"hw\". The contents
            are:\n %s \n", xattr_res);
}
// 移除对象属性
err = rados_rmxattr(io, "hw", "lang");
if (err < 0) {
    fprintf(stderr, "%s: Cannot remove xattr. %s %s\n", argv[0],
            poolname, strerror(-err));
    rados_ioctx_destroy(io);
    rados_shutdown(cluster);
}

```

```

        exit(1);
    } else {
        printf("\nRemoved xattr \"lang\" for object \"hw\".\n");
    }
    // 删除对象
    err = rados_remove(io, "hw");
    if (err < 0) {
        fprintf(stderr, "%s: Cannot remove object. %s %s\n", argv[0],
            poolname, strerror(-err));
        rados_ioctx_destroy(io);
        rados_shutdown(cluster);
        exit(1);
    } else {
        printf("\nRemoved object \"hw\".\n");
    }
    rados_ioctx_destroy(io);           // 销毁 io 上下文
    rados_shutdown(cluster);          // 销毁句柄
}

```

## 2. LIBRADOS Java 语言示例

下面是 LIBRADOS Java 语言示例。

```

import com.ceph.rados.Rados;
import com.ceph.rados.RadosException;
import java.io.File;
public class CephClient {
    public static void main (String args[]){
        try {
            // 获取句柄
            Rados cluster = new Rados("admin");
            System.out.println("Created cluster handle.");

            File f = new File("/etc/ceph/ceph.conf");
            // 读取配置文件
            cluster.confReadFile(f);
            System.out.println("Read the configuration file.");
            // 连接集群
            cluster.connect();
            System.out.println("Connected to the cluster.");

        } catch (RadosException e) {
            System.out.println(e.getMessage()+"："+e.getReturnValue());
        }
    }
}

```

### 3. LIBRADOS Python 语言示例

下面是 LIBRADOS Python 语言示例。

```
#!/usr/bin/python
#encoding=utf-8
import rados, sys
cluster = rados.Rados(conffile='/etc/ceph/ceph.conf')    # 获取句柄
print "\n\nI/O Context and Object Operations"
print "=====
print "\nCreating a context for the 'data' pool"
if not cluster.pool_exists('data'):
    raise RuntimeError('No data pool exists')
ioctx = cluster.open_ioctx('data')                    # 获取 pool 的 io 上下文句柄
print "\nWriting object 'hw' with contents 'Hello World!' to pool 'data'."
ioctx.write("hw", "Hello World!")                    # 写入对象
print "Writing XATTR 'lang' with value 'en_US' to object 'hw'"
ioctx.set_xattr("hw", "lang", "en_US")                # 设置对象的属性
print "\nWriting object 'bm' with contents 'Bonjour tout le monde!' to pool 'data'."
ioctx.write("bm", "Bonjour tout le monde!")
print "Writing XATTR 'lang' with value 'fr_FR' to object 'bm'"
ioctx.set_xattr("bm", "lang", "fr_FR")
print "\nContents of object 'hw'\n-----"
print ioctx.read("hw")                                # 读取对象
print "\n\nGetting XATTR 'lang' from object 'hw'"
print ioctx.get_xattr("hw", "lang")                    # 读取对象属性
print "\nContents of object 'bm'\n-----"
print ioctx.read("bm")
    print "\nClosing the connection."
ioctx.close()                                          # 关闭 io 上下文

print "Shutting down the handle."
cluster.shutdown()                                    # 销毁句柄
```

## 2.4 本章小结

本章从宏观角度剖析了 Ceph 架构，将 Ceph 架构分为基础存储系统 RADOS、基于 RADOS 实现的 CEPHFS，基于 RADOS 的 LIBRADOS 层应用接口、基于 LIBRADOS 的应用接口 RBD、RADOSGW，其中着重讲解了 RADOS 的组成部分 MON、OSD 及其功能，最后解析 LIBRADOS API 的基本用法。

## 智能分布 CRUSH

### 3.1 引言

数据分布是分布式存储系统的一个重要部分，数据分布算法至少要考虑以下 3 个因素。

- 1) 故障域隔离。同份数据的不同副本分布在不同的故障域，降低数据损坏的风险。
- 2) 负载均衡。数据能够均匀地分布在磁盘容量不等的存储节点，避免部分节点空闲，部分节点超载，从而影响系统性能。
- 3) 控制节点加入离开时引起的数据迁移量。当节点离开时，最优的数据迁移是只有离线节点上的数据被迁移到其他节点，而正常工作的节点的数据不会发生迁移。

对象存储中一致性 Hash 和 Ceph 的 CRUSH 算法是使用比较多的数据分布算法。在 Amazon 的 Dynamo 键值存储系统中采用一致性 Hash 算法，并且对它做了很多优化。OpenStack 的 Swift 对象存储系统也使用了一致性 Hash 算法。

CRUSH (Controlled Replication Under Scalable Hashing) 是一种基于伪随机控制数据分布、复制的算法。Ceph 是为大规模分布式存储系统 (PB 级的数据和成百上千台存储设备) 而设计的，在大规模的存储系统里，必须考虑数据的平衡分布和负载 (提高资源利用



率)、最大化系统的性能,以及系统的扩展和硬件容错等。CRUSH 就是为解决以上问题而设计的。在 Ceph 集群里,CRUSH 只需要一个简洁而层次清晰的设备描述,包括存储集群和副本放置策略,就可以有效地把数据对象映射到存储设备上,且这个过程是完全分布式的,在集群系统中的任何一方都可以独立计算任何对象的位置;另外,大型系统存储结构是动态变化的(存储节点的扩展或者扩容、硬件故障等),CRUSH 能够处理存储设备的变更(添加或删除),并最小化由于存储设备的变更而导致的数据迁移。

## 3.2 CRUSH 基本原理

众所周知,存储设备具有吞吐量限制,它影响读写性能和可扩展性能。所以,存储系统通常都支持条带化以增加存储系统的吞吐量并提升性能,数据条带化最常见的方式是做 RAID。与 Ceph 的条带化最相似的是 RAID 0 或者是“带区卷”。Ceph 条带化提供了类似于 RAID 0 的吞吐量,N 路 RAID 镜像的可靠性以及更快速的恢复能力。

在磁盘阵列中,数据是以条带(stripe)的方式贯穿在磁盘阵列所有硬盘中的。这种数据的分配方式可以弥补 OS 读取数据量跟不上的不足。

1) 将条带单元(stripe unit)从阵列的第一个硬盘到最后一个硬盘收集起来,就可以称为条带(stripe)。有的时候,条带单元也被称为交错深度。在光纤技术中,一个条带单元被叫作段。

2) 数据在阵列中的硬盘上是以条带的形式分布的,条带化是指数据在阵列中所有硬盘中的存储过程。文件中的数据被分割成小块的数据段在阵列中的硬盘上顺序的存储,这个最小数据块就叫作条带单元。

决定 Ceph 条带化数据的 3 个因素。

- ❑ 对象大小:处于分布式集群中的对象拥有一个最大可配置的尺寸(例如,2MB、4MB 等),对象大小应该足够大以适应大量的条带单元。
- ❑ 条带宽度:条带有一个可以配置的单元大小,Ceph Client 端将数据写入对象分成相同大小的条带单元,除了最后一个条带之外;每个条带宽度,应该是对象大小的一小部分,这样使得一个对象可以包含多个条带单元。
- ❑ 条带总量:Ceph 客户端写入一系列的条带单元到一系列的对象,这就决定了条带

的总量，这些对象被称为对象集，当 Ceph 客户端写入的对象集合中的最后一个对象之后，它将会返回到对象集合中的第一个对象处。

### 3.2.1 Object 与 PG

Ceph 条带化之后，将获得 N 个带有唯一 oid（即 object 的 id）。Object id 是进行线性映射生成的，即由 file 的元数据、Ceph 条带化产生的 Object 的序号连缀而成。此时 Object 需要映射到 PG 中，该映射包括两部分。

- 1) 由 Ceph 集群指定的静态 Hash 函数计算 Object 的 oid，获取到其 Hash 值。
- 2) 将该 Hash 值与 mask 进行与操作，从而获得 PG ID。

根据 RADOS 的设计，假定集群中设定的 PG 总数为 M（M 一般为 2 的整数幂），则 mask 的值为 M-1。由此，Hash 值计算之后，进行按位与操作是想从所有 PG 中近似均匀地随机选择。基于该原理以及概率论的相关原理，当用于数量庞大的 Object 以及 PG 时，获得到的 PG ID 是近似均匀的。

计算 PG 的 ID 示例如下。

- 1) Client 输入 pool ID 和对象 ID（如 pool= 'liverpool'，object-id= 'john'）。
- 2) CRUSH 获得对象 ID 并对其 Hash 运算。
- 3) CRUSH 计算 OSD 个数，Hash 取模获得 PG 的 ID（如 0x58）。
- 4) CRUSH 获得已命名 pool 的 ID（如 liverpool=4）。
- 5) CRUSH 预先考虑到 pool ID 相同的 PG ID（如 4.0x58）。

### 3.2.2 PG 与 OSD

由 PG 映射到数据存储的实际单元 OSD 中，该映射是由 CRUSH 算法来确定的，将 PG ID 作为该算法的输入，获得到包含 N 个 OSD 的集合，集合中第一个 OSD 被作为主 OSD，其他的 OSD 则依次作为从 OSD。N 为该 PG 所在 POOL 下的副本数目，在生产环境中 N 一般为 3；OSD 集合中的 OSD 将共同存储和维护该 PG 下的 Object。需要注意的是，CRUSH 算法的结果不是绝对不变的，而是受到其他因素的影响。其影响因素主要有以下两个。

一是**当前系统状态**。也就是上文逻辑结构中曾经提及的 Cluster Map（集群映射）。当系统中的 OSD 状态、数量发生变化时，Cluster Map 可能发生变化，而这种变化将会影响到 PG 与 OSD 之间的映射。

二是**存储策略配置**。这里的策略主要与安全相关。利用策略配置，系统管理员可以指定承载同一个 PG 的 3 个 OSD 分别位于数据中心的不同服务器乃至机架上，从而进一步改善存储的可靠性。

因此，只有在 Cluster Map 和存储策略都不发生变化时，PG 和 OSD 之间的映射关系才是固定不变的。在实际使用中，策略一经配置通常不会改变。而系统状态的改变或者是因为设备损坏，或者是因为存储集群规模扩大。好在 Ceph 本身提供了对于这种变化的自动化支持，因而，即便 PG 与 OSD 之间的映射关系发生了变化，并不会对应用造成困扰。事实上，Ceph 正是需要有目的的利用这种动态映射关系。正是利用了 CRUSH 的动态特性，Ceph 才可以将一个 PG 根据需要动态迁移到不同的 OSD 组合上，从而自动化地实现高可靠性、数据分布 re-blancing 等特性。

之所以在此次映射中使用 CRUSH 算法，而不是其他 Hash 算法，原因之一是 CRUSH 具有上述可配置特性，可以根据管理员的配置参数决定 OSD 的物理位置映射策略；另一方面是因为 CRUSH 具有特殊的“稳定性”，也就是当系统中加入新的 OSD 导致系统规模增大时，大部分 PG 与 OSD 之间的映射关系不会发生改变，只有少部分 PG 的映射关系会发生变化并引发数据迁移。这种可配置性和稳定性都不是普通 Hash 算法所能提供的。因此，CRUSH 算法的设计也是 Ceph 的核心内容之一。

### 3.2.3 PG 与 Pool

Ceph 存储系统支持“池”（Pool）的概念，这是存储对象的逻辑分区。

Ceph Client 端从 Ceph mon 端检索 Cluster Map，写入对象到 Pool。Pool 的副本数目，Crush 规则和 PG 数目决定了 Ceph 将数据存储的位置，如图 3-1 所示。

Pool 至少需要设定以下参数。

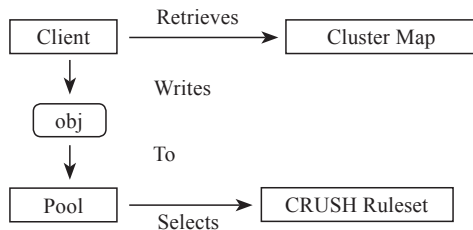


图 3-1 调用关系图

- ❑ 对象的所有权 / 访问权。
- ❑ PG 数目。
- ❑ 该 pool 使用的 CRUSH 规则。
- ❑ 对象副本的数目。

Object、PG、Pool、OSD 关系图，如图 3-2 所示。

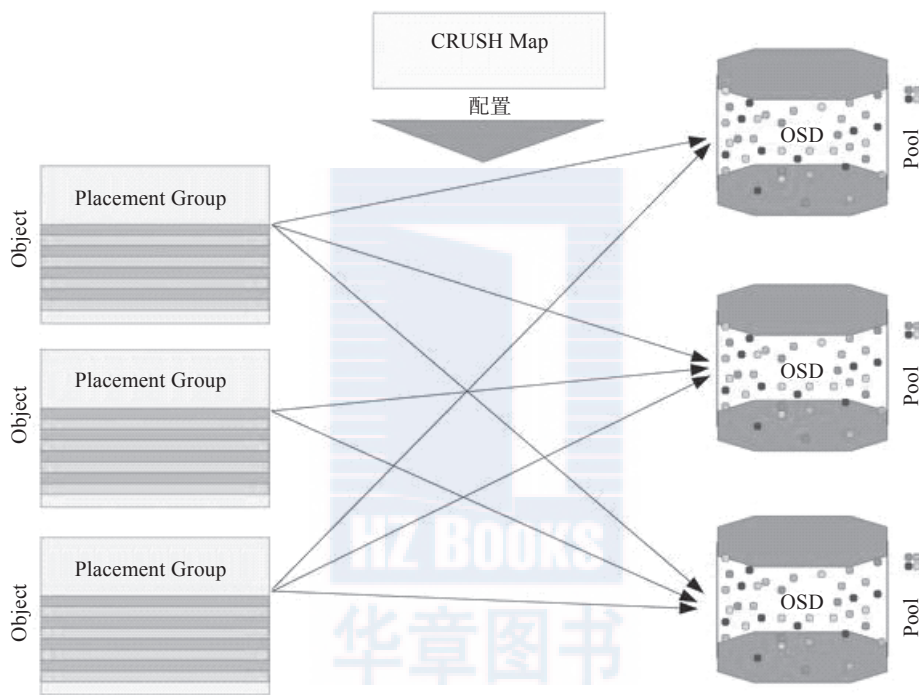


图 3-2 映射关系图

Pool 相关的操作如下。

#### 1) 创建 pool。

```
ceph osd pool create {pool-name} {pg-num} [{pgp-num}] [replicated] \
    [crush-ruleset-name] [expected-num-objects]
ceph osd pool create {pool-name} {pg-num} {pgp-num} erasure \
    [erasure-code-profile] [crush-ruleset-name] [expected_num_objects]
```

#### 2) 配置 pool 配额。

```
ceph osd pool set-quota {pool-name} [max_objects {obj-count}] [max_bytes {bytes}]
```

3) 删除 pool。

```
ceph osd pool delete {pool-name} [{pool-name} --yes-i-really-really-mean-it]
```

4) 重命名 pool。

```
ceph osd pool rename {current-pool-name} {new-pool-name}
```

5) 展示 pool 统计。

```
rados df
```

6) 给 pool 做快照。

```
ceph osd pool mksnap {pool-name} {snap-name}
```

7) 删除 pool 快照。

```
ceph osd pool rmsnap {pool-name} {snap-name}
```

8) 配置 pool 的相关参数。

```
ceph osd pool set {pool-name} {key} {value}
```

9) 获取 pool 参数的值。

```
ceph osd pool get {pool-name} {key}
```

10) 配置对象副本数目。

```
ceph osd pool set {poolname} size {num-replicas}
```

11) 获取对象副本数目。

```
ceph osd dump | grep 'replicated size'
```

### 3.3 CRUSH 关系分析

从本质上讲, CRUSH 算法是通过存储设备的权重来计算数据对象的分布的。在计算过程中, 通过 Cluster Map (集群映射)、Data Distribution Policy (数据分布策略) 和给出的一个随机数共同决定数据对象的最终位置。

## 1. Cluster Map

Cluster Map 记录所有可用的存储资源及相互之间的空间层次结构（集群中有多少个机架、机架上有多少服务器、每个机器上有多少磁盘等信息）。所谓的 Map，顾名思义，就是类似于我们生活中的地图。在 Ceph 存储里，数据的索引都是通过各种不同的 Map 来实现的。另一方面，Map 使得 Ceph 集群存储设备在物理层作了一层防护。例如，在多副本（常见的三副本）的结构上，通过设置合理的 Map（故障域设置为 Host 级），可以保证在某一服务器死机的情况下，有其他副本保留在正常的存储节点上，能够继续提供服务，实现存储的高可用。设置更高的故障域级别（如 Rack、Row 等）能保证整机柜或同一排机柜在掉电情况下数据的可用性和完整性。

### （1）Cluster Map 的分层结构

Cluster Map 由 Device 和 Bucket 构成。它们都有自己的 ID 和权重值，并且形成一个以 Device 为叶子节点、Bucket 为躯干的树状结构，如图 3-3 所示。

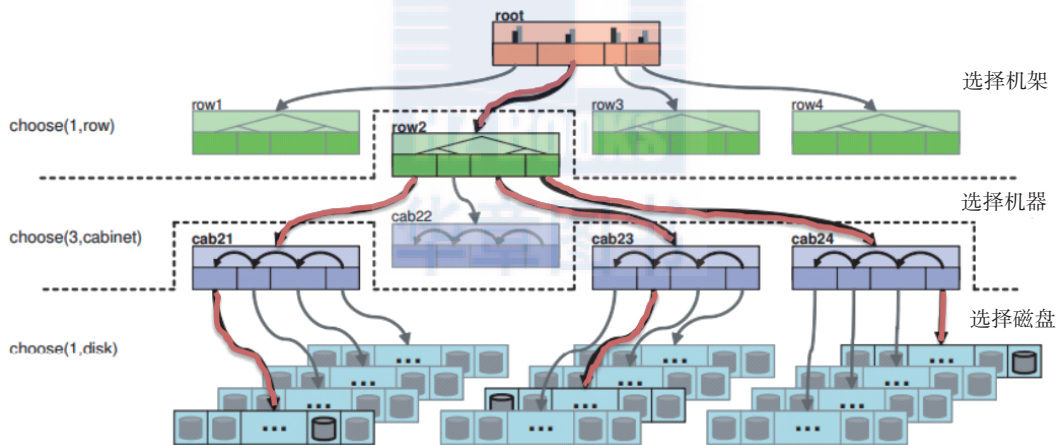


图 3-3 CRUSH 架构图

Bucket 拥有不同的类型，如 Host、Row、Rack、Room 等，通常我们默认把机架类型定义为 Rack，主机类型定义为 Host，数据中心（IDC 机房）定义为 Data Center。Bucket 的类型都是虚拟结构，可以根据自己的喜好设计合适的类型。Device 节点的权重值代表了存储设备的容量与性能。其中，磁盘容量是权重大小的关键因素。

OSD 的权重值越高，对应磁盘会被分配写入更多的数据。总体来看，数据会被均匀写入分布于群集所有磁盘，从而提高整体性能和可靠性。无论磁盘的规格容量，总能够均匀使用。

关于 OSD 权重值的大小值的配比，官方默认值设置为 1TB 容量的硬盘，对应权重值为 1。

可以在 `/etc/init.d/ceph` 原码里查看相关的内容。

```
...
363         get_conf osd_weight "" "osd crush initial weight"
364         defaultweight="$(df -P -k $osd_data/. | tail -1 | awk '{
print sprintf("%.2f", $2/1073741824) }')" ### 此处就是 ceph 默认权重的设置值
        get_conf osd_keyring "$osd_data/keyring" "keyring"
366         do_cmd_okfail "timeout 30 $BINDIR/ceph -c $conf --name=osd.$sid
--keyring=$osd_keyring osd crush create-or-move -- $sid ${osd_weight:-
${defaultweight:-1}} $osd_location"
...
```

## (2) 恢复与动态平衡

在默认设置下，当集群里有组件出现故障时（主要是 OSD，也可能是磁盘或者网络等），Ceph 会把 OSD 标记为 down，如果在 300s 内未能回复，集群就会开始进行恢复状态。这个“300s”可以通过“`mon osd down out interval`”配置选项修改等待时间。PG(Placement Groups) 是 Ceph 数据管理（包括复制、修复等动作）单元。当客户端把读写请求（对象单元）推送到 Ceph 时，通过 CRUSH 提供的 Hash 算法把对象映射到 PG。PG 在 CRUSH 策略的影响下，最终会被映射到 OSD 上。

## 2. Data Distribution Policy

Data Distribution Policy 由 Placement Rules 组成。Rule 决定了每个数据对象有多少个副本，这些副本存储的限制条件（比如 3 个副本放在不同的机架中）。一个典型的 rule 如下所示。

```
rule replicated_ruleset {    ##rule 名字
    ruleset 0                # rule 的 Id
    type replicated          ## 类型为副本模式，另外一种模式为纠删码 (EC)
    min_size 1               ## 如果存储池的副本数大于这个值，此 rule 不会应用
    max_size 10              ## 如要存储池的副本数大于这个值，此 rule 不会应用
```



```

step take default  ## 以 default root 为入口
step chooseleaf firstn 0 type host  ## 隔离域为 host 级，即不同副本在不同的主机上
step emit          ## 提交
}

```

根据实际的设备环境，可以定制出符合自己需求的 Rule，详见第 10 章。

### 3. CRUSH 中的伪随机

先来看一下数据映射到具体 OSD 的函数表达形式。

```
CRUSH(x)  ->  (osd1, osd2, osd3.....osdn)
```

CRUSH 使用了多参数的 Hash 函数在 Hash 之后，映射都是按既定规则选择的，这使得从 x 到 OSD 的集合是确定的和独立的。CRUSH 只使用 Cluster Map、Placement Rules、X。CRUSH 是伪随机算法，相似输入的结果之间没有相关性。关于伪随机的确认性和独立性，以下我们以一个实例来展示。

```

[root@host-192-168-0-16 ~]# ceph osd tree
ID WEIGHT  TYPE NAME                UP/DOWN REWEIGHT  PRIMARY-AFFINITY
-1 0.35999  root default
-3 0.09000  host host-192-168-0-17
 2 0.09000  osd.2                up 1.00000      1.00000
-4 0.09000  host host-192-168-0-18
 3 0.09000  osd.3                up 1.00000      1.00000
-2 0.17999  host host-192-168-0-16
 0 0.09000  osd.0                up 1.00000      1.00000
 1 0.09000  osd.1                up 1.00000      1.00000

```

以上是某个 Ceph 集群的存储的 CRUSH 结构。在此集群里我们手动创建一个 pool，并指定为 8 个 PG 和 PGP。

```

[root@host-192-168-0-16 ~]# ceph osd pool create crush 8 8
pool 'crush' created

```



**注意** PGP 是 PG 的逻辑承载体，是 CRUSH 算法不可缺少的部分。在 Ceph 集群里，增加 PG 数量，PG 到 OSD 的映射关系就会发生变化，但此时存储在 PG 里的数据并不会发生迁移，只有当 PGP 的数量也增加时，数据迁移才会真正开始。关于 PG 和 PGP 的关系，假如把 PG 比作参加宴会的人，那么 PGP 就是人坐的椅子，如果人员增加时，人的座位排序就会发生变化，只有增加椅子时，真正的座位排序变



更才会落实。因此，人和椅子的数量一般都保持一致。所以，在 Ceph 里，通常把 PGP 和 PG 设置成一致的。

可以查看 PG 映射 OSD 的集合，即 PG 具体分配到 OSD 的归属。

```
[root@host-192-168-0-16 ~]# ceph pg dump | grep ^22\. | awk '{print $1 "\t"
$17}' ## 22 表示 Pool id ##
dumped all in format plain
22.1      [1,2,3]
22.0      [1,2,3]
22.3      [3,1,2]
22.2      [3,2,0]
22.5      [1,3,2]
22.4      [3,1,2]
22.7      [2,1,3]
22.6      [3,0,2]
```

上面示例中，第一列是 PG 的编号（其中 22 表示的 Pool 的 ID），共计 8 个，第二列是 PG 映射到了具体的 OSD 集合。如“22.1 [1,2,3]”表示 PG 22.1 分别在 OSD1、OSD2 和 OSD3 分别存储副本。

在 Ceph 集群里，当有数据对象要写入集群时，需要进行两次映射。第一次从 object → PG，第二次是 PG → OSD set。每一次的映射都是与其他对象无相关的。以上充分体现了 CRUSH 的独立性（充分分散）和确定性（可确定的存储位置）。

### 3.4 本章小结

本章主要围绕 Ceph 的核心——CRUSH 展开，讲述了 CRUSH 的基本原理以及 CRUSH 的特性。读者可以了解 Ceph 基本要素的概念，重点应该把握 Ceph 的 Object、PG、Pool 等基本核心概念，熟悉读写流程以及 CRUSH 算法的组成、影响因素及选择流程。基于以上掌握的要点，为后面的深入学习打好坚实的基础。