

单位代码： 10293 密 级： _____

南京邮电大学

专业学位硕士学位论文



论文题目： OpenStack 和 Ceph 结合的云存储
设计与实现

学 号 1215012403

姓 名 周林

导 师 宋建新 教授

专业学位类别 工程硕士

类 型 全 日 制

专业（领域） 电子与通信工程

论文提交日期 二零一八年四月

The design and implement of cloud storage scheme combined with openStack and ceph

Thesis Submitted to Nanjing University of Posts and
Telecommunications for the Degree of
Master of Engineering



By

Zhou Lin

Supervisor: Prof. Song Jianxin

April 2018

南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京邮电大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生学号：_____ 研究生签名：_____ 日期：_____

南京邮电大学学位论文使用授权声明

本人承诺所呈交的学位论文不涉及任何国家秘密，本人及导师为本论文的涉密责任并列第一责任人。

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档；允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索；可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院办理。

非国家秘密类涉密学位论文在解密后适用本授权书。

研究生签名：_____ 导师签名：_____ 日期：_____

摘要

随着人们对信息存储的需求不断增加，对存储便捷性要求也越来越高，传统的存储方式已经不能满足这些需求。众所周知，传统的存储方式大部分只能依靠单个节点来存储信息，不仅成本高，而且效率低；若单节点的物理磁盘损坏，数据便会全部丢失，无法找回；假设某些企业用户后来发展壮大，信息储量远远超过了现在的容量，则需要人工扩容，其效率也是极其低下的，硬盘的使用率也很低，存在有的盘已经装满数据，有的盘还是空的情况，若企业想将数据迁移到别的地方也非常地困难，需要花费大量的人力物力来完成数据转移工作。应该使用什么样的方式来存储企业的数据，并对它们进行便捷的管理，是当前的企业面临的燃眉之急。实习所在的机敏科技软件有限公司（南京机敏科技软件有限公司是业内第一家 OpenStack 桌面云厂商，提供企业级稳定高效可靠的云计算服务，具有极简运维，多维度的安全防护，易扩容，易部署等特点）也面临同样的问题。

本文通过分析国内外研究现状并结合公司需求，从云存储系统方案设计、系统底层设计、系统功能接口设计和实现、系统部署和调试四个方面进行研究和实现；论文的主要工作如下：

首先，分析了云计算和 OpenStack 等相关技术，比较了 Ceph 和 Swift 的优缺点，提出了 Ceph 和 OpenStack 结合的两种方案，给出方案的设计架构，并比较了两种方案，分析了方案的可行性和复杂度，确定最终方案。

然后，是云存储系统底层的设计和部署，底层设计的方案包括服务器的使用、网络节点的选择、Monitor 节点和 OSD 节点的安排；底层部署则使用三台服务器新建 Ceph 的集群，需要给集群设置两个网络；包括公有网络和集群网络，使得集群能够健康的运行；

其次，根据用户的需求设计指定的接口，分析接口的逻辑，并在 Ceph 集群基础上实现接口的功能，界面则在 Horizon 中进行二次开发，系统可以实现数据存储、下载等功能；不仅对外可以提供对象存储服务，当数据过多时候还能够动态有效的进行扩容，使数据存储变得更高效、更便捷。

最后将系统部署到公司的服务器上，验证是否能够提供简单的对象存储功能，并对系统的性能进行测试。

关键词:OpenStack, Ceph, Swift, 云存储, 对象存储, Radosgw

Abstract

With the increasing demand for information storage, the demand for storage is more and more high, the traditional storage method can not meet these needs. It is well known that most of the traditional storage methods can only rely on single nodes to store information, not only high cost and low efficiency, but if the physical disk of single node is damaged, the data will be lost and can not be retrieved. Set some enterprise users to grow and grow, the information reserves are much more than the current capacity, the efficiency of artificial expansion is also extremely low, and the use of hard disk is very low; there may be some disks already full, some disks are still empty, if the enterprise wants to transfer the data to other places is also very difficult, it is necessary to spend a lot of manpower and material resources to complete the data transfer. Shift work. What kind of ways should be used to store data and manage them conveniently is the urgent need for the current enterprises. The intern's smart technology Software Co., Ltd. (Nanjing smart tech software Co., Ltd. is the first OpenStack desktop cloud manufacturer in the industry to provide firm and reliable and reliable cloud computing services, with the features of extremely simplified operation and maintenance, multidimensional security protection, easy expansion, easy deployment and so on) also face the same problems.

By analyzing the current research status at home and abroad and combining the needs of the company, this paper studies and implements the four aspects of the design of the cloud storage system, the design of the system bottom, the design and implementation of the system function interface, the deployment and debugging of the system. The main work of this paper is as follows:

First, it analyzes the related technologies such as cloud computing and OpenStack, compares the advantages and disadvantages of Ceph and Swift, puts forward two schemes combining Ceph and OpenStack, gives the design framework of the scheme, compares two schemes, analyzes the feasibility and complexity of the scheme, and determines the final scheme.

Then, it is the design and deployment of the bottom of the cloud storage system. The underlying design includes the use of the server, the selection of the network nodes, the Monitor node and the arrangement of the OSD nodes; the bottom deployment uses three servers to build the cluster of Ceph, and needs to set up two networks for the cluster, including the public network and the cluster network, so that the cluster can run healthily;

Secondly, the interface is designed according to the needs of the user, the logic of the interface

is analyzed, and the function of the interface is realized on the basis of the Ceph cluster. The interface is developed two times in the Horizon. The system can realize the functions of data storage, downloading and so on. Not only can we provide the storage and storage service of the object, but also can dynamically and effectively expand the data when the data is too much, so that the data storage can be stored. More efficient and more convenient.

Finally, the system is deployed to the company's server to verify whether a simple object storage function can be provided, and the reliability of the system is verified, and the system is highly reliable.

Key words: OpenStack, Ceph, Swift, Cloud storage, Object storage, Radosgw

目录

专用术语注释表	VII
第一章 绪论	1
1.1 项目背景及意义	1
1.2 国内外研究现状分析	2
1.2.1 国外研究现状	2
1.2.2 国内研究现状	3
1.3 本文的主要贡献和论文结构	3
第二章 OpenStack 与云存储系统需求分析	5
2.1 OpenStack 简介	5
2.1.1 OpenStack 架构	5
2.1.2 OpenStack 核心组件	6
2.1.3 OpenStack 主要技术	9
2.2 云存储系统功能需求分析	9
2.3 系统开发问题分析	10
2.4 本章小结	10
第三章 云存储的技术分析与 Ceph 的改进	11
3.1 云存储	11
3.1.1 云存储定义	11
3.1.2 云存储系统结构	12
3.2 对象存储	13
3.2.1 对象存储定义	13
3.2.2 对象存储原理	14
3.2.3 三种存储结构图	14
3.3 对象存储 Swift	15
3.3.1 Swift 设计背景	16
3.3.2 Swift 系统架构	16
3.3.3 Swift 核心组件	18
3.3.4 Swift 基本原理	19
3.3.5 Swift 主要特性	21
3.4 对象存储 Ceph	22
3.4.1 Ceph 设计背景	22
3.4.2 Ceph 设计思想	23
3.4.3 Ceph 总体结构	25
3.4.4 Ceph 工作原理	27
3.5 Ceph 和 Swift 比较	32
3.5.1 两者相同点	33
3.5.2 两者不同点	34
3.6 对象存储系统方案研究	34
3.6.1 系统设计方案一	34
3.6.2 系统方案设计二	35
3.6.3 系统方案比较	36

3.6.4 系统总体架构	36
3.7 本章小结.....	37
第四章 对象存储系统的设计与实现	39
4.1 接口设计相关技术介绍.....	39
4.1.1 接口开发框架	39
4.1.2 接口开发要求	40
4.1.3 接口设计思路	40
4.2 底层的设计.....	41
4.2.1 底层设计主要过程	41
4.2.2 底层设计逻辑结构图	42
4.3 Ceph 和 OpenStack 结合设计.....	43
4.3.1 概述	43
4.3.2 两者结合的系统架构设计	44
4.4 系统管理模块的设计.....	44
4.4.1 集群状态展示设计	44
4.4.2 OSD 操作设计	45
4.5 系统功能模块的设计.....	46
4.5.1 用户管理模块操作设计	46
4.5.2 桶 Buckets 的操作设计	47
4.5.3 Objects 的操作设计	48
4.6 系统底层实现.....	50
4.6.1 底层实现结构图	50
4.6.2 底层部署实现	51
4.7 Ceph 和 OpenStack 结合的实现	53
4.7.1 Ceph 作为 Cinder 后端的实现.....	53
4.7.2 Ceph 作为 Glance 后端的实现.....	55
4.8 系统管理模块的实现.....	55
4.8.1 集群状态展示功能的实现	56
4.8.2 OSD 添加/删除功能的实现	57
4.9 系统功能模块的实现.....	61
4.9.1 用户信息管理模块实现	61
4.9.2 Bucket 容器管理模块实现	63
4.9.3 Object 管理模块开发与实现.....	64
4.10 对磁盘利用率的改进.....	67
4.10.1 Ceph 权重简介	67
4.10.2 测试权重	68
4.10.3 改进方案	69
4.10.4 结果分析	69
4.11 本章小结.....	70
第五章 对象存储系统的部署和调试	71
5.1 测试环境部署.....	71
5.1.1 机敏云安装	71
5.1.2 Ceph 集群安装.....	72
5.2 环境模块调试.....	73
5.2.1 RGW 和 Nginx 服务器	73

5.2.2 参数配置	74
5.2.3 检测 S3 服务	74
5.2.4 RGW 的访问	75
5.2.5 Keystone 认证	76
5.3 系统管理模块展示	77
5.3.1 集群状态预览效果展示	77
5.3.2 OSD 添加效果展示	78
5.4 系统功能模块展示	79
5.4.1 用户管理模块效果展示	79
5.4.2 Bucket 容器效果展示	79
5.4.3 Object 操作效果展示	80
5.5 系统性能测试	81
5.5.1 验证系统高可靠性	81
5.5.2 验证系统的分布式特性	85
5.5.3 验证系统的高效平衡性	86
5.6 本章小结	87
第六章 总结与展望	88
参考文献	89
附录 1 攻读硕士学位期间撰写的论文	92
附录 2 程序代码清单	93
致谢	94

专用术语注释表

缩略词说明：

CAP	Consistency Availability Partition tolerance	一致性、可用性、分区容忍性
CIFS	Common Internet File System	通用 Internet 文件系统
DB	Database	数据库
FC	Fibre channel	光纤通道
FS	File System	文件系统
FTP	File Transfer Protocol	文件传输协议
IO	Input/output	输入输出
ISCSI	Internet Small Computer System Interface	互联网小型计算机系统接口
Libvirt	Libvirt	管理虚拟化平台的开源的 API
LVM	Logical Volume Manager	逻辑卷
MDS	metadata server daemon	元数据服务器守护进程
MDS	Metadata server	元数据设备
Mon	Monitor	Ceph 中监视节点
MQ	Message queue	消息队列，消息总线
NFS	Network File System	网络文件系统
OSD	Object Storage Daemon	对象存储守护进程
PG	Placement Group	Ceph 的逻辑存储单元
QEMU	QEMU	模拟处理器
RAID	Redundant Arrays of Independent Disks	磁盘阵列
RBD	Rados block device	块存储设备
RESTful	Representational State Transfer	一种软件架构风格、设计风格
RGW	Rados gateway	对象存储网关
SSH	Secure Shell	安全外壳协议

名词解释：

Ceph： 开源的对象存储系统。

Hypervisor： 一种运行在物理服务器和操作系统之间的中间层软件，虚拟化层。

雅虎 YCSB 框架： Yahoo! Cloud Serving Benchmark 是一款用来对云服务进行基础测试的工具。

OpenStack： 开源的云操作系统。

第一章 绪论

1.1 项目背景及意义

Google 公司早在 2006 年就已经提出了云计算的概念^[1]，现如今已发展了十二年，云计算已经与我们的生活息息相关了，特别是在教育方面^{[2][3]}；在如今的信息化大发展时代，云计算与我们的生活更加密不可分。当云计算成为热点之后，不仅信息的存储量飞速增长，用户对存储的便捷性、安全性的需求也越来越高。根据相关部门的统计，近几年的数据量急速增加，急需一种新的存储方式。然而在众多因素的限制下，如容量、价格和安全性等方面，以前的存储方式只依靠单个节点来存储信息，不仅成本高，效率低；而且若单节点的物理磁盘损坏，则数据全部丢失，很难再找回，资源利用率也很低下，可能存在有的盘已经装满数据，有的盘还是空的情况，而且存储后的数据迁移起来非常的艰难，工作量肯定是非常巨大的。特别是对很多数据大的企业来说，数据共享是一个很大的问题，管理起来也不方便^[4]，而且这些数据优势非结构化的，理所当然，要想保存这些数据成本肯定是很高的^[5]。

使用什么样的措施，才能有效的管理和应用存储的数据，降低数据的存储成本和提升数据保密性、安全性是企业需要解决的燃眉之急，是重中之重的问题。因此研究如何高效的存储，对机敏和其它很多公司来说都是很重要的，不仅可以带来经济利益，更能在同类公司崭露头角。

研究 Ceph 和 OpenStack 结合的难点在于：（1）模块之间融合问题，Swift 作为 OpenStack 原生模块，接口已经相当成熟，Ceph 的接口则完全不一样，需要重写（2）底层的设计、部署，模块之间版本、证书冲突等问题；（3）这个系统是基于 CephRGW 开发的^[6]，使用 RGW 来进行构造^[7]，可以用完备的对象存储服务；如何基于 RGW 开发接口，而且接口必须是要支持 Restful 标准的^[8]；（4）怎么结合 S3^[9]的命令行来验证用户的授权，使得授权用户可以使用 Amazon 的接口工具访问云存储系统服务。

目前公司还未给客户提供对象存储服务，又因为 Ceph 是可以实现统一存储的，所以将 OpenStack 和 Ceph 结合是一个很不错的趋势^[17]；公司只用到了 Ceph 的块存储功能和文件存储功能，其中也在研究怎么能够提高文件的吞吐效率等^[18]。所以本文主要的工作就是基于 CephRGW 开发对象存储服务，使用 Keystone 进行验证，客户能够便捷的管理自己的云存储系统。因此，构建一个具有高扩展性、高安全性、高容错率的对象存储系统，对于机敏公司的数据管理具有非常巨大的现实意义。

1.2 国内外研究现状分析

1.2.1 国外研究现状

国外对云存储的研究和开发起步都非常的早，早就有相应的产品了。自从 Google 公司开始提出云计算的概念以后，大部分的厂家都开始推出相关的服务了，很多都是耳熟能详的，下面就简单介绍几种知名的云存储服务：

(a) 亚马逊简单存储服务

亚马逊简单存储服务是日常生活中见到的最多也是全球覆盖最广的一种服务，它是亚马逊公司维护的基于云存储的简单存储服务，所有付费用户都可以便捷的在自己电脑上随时查看和下载保存的数据，不管用户是在哪个地方，只要能连接上互联网就可以访问。

亚马逊的 S3 存储服务能够让所有使用它的开发人员访问同一个快速并且廉价的基础设施，其具有很高的扩展性、高可靠性、高安全性^[10]。

(b) Google Cloud Storage

谷歌公司是全球知名的物联网公司，它旗下很多产品都很优秀，这个云计算的概念也是它提出来的，可以说是创始者，但并没有很早的推出产品，是在提出概念很久之后才有相应的服务提供^[11]。它是可以让付费用户通过接口进入谷歌云的。会员可以用很低的价格存储数据，它们会负责用户的数据安全。

(c) Dropbox

Dropbox 是一款非常老旧的存储服务提供商，大概在十年前就已经提供了云存储的服务；它主要的特性就是文件保护；所有认证的用户都会得到免费的 2G 空间，也有很多获得容量的方式。最重要一点就是 Dropbox 为 IOS 端和安卓端都推出了移动应用；只要能连上网，都可以随时查看用户的文件。

(d) iCloud

iCloud 是苹果公司专门为苹果用户专属的在线存储应用，只要使用 iPhone 的用户都非常的熟悉；可以同步用户的信息，照片等资料；但是免费的空间只有 5GB，若想升级则需要付费给苹果公司。iCloud 是苹果用户特有的，所以有一定局限性。

从上面的几款云存储产品来看，国外对云存储的研究已经非常的优秀；但是它们都是单独的对外提供存储服务；怎么将 OpenStack 和 Ceph 结合提供云存储服务的研究还很少，Ceph 可以提供统一的存储，它既可以为 OpenStack 服务，也可以对外提供对象存储服务。所以本文基于这个问题对 OpenStack 和 Ceph 的结合展开研究。

1.2.2 国内研究现状

云计算和云存储在国内起步较晚^[12]，最早起步的是阿里云，所以发展非常迅猛^[13]，政府机构，科研单位，院校及企业单位都极力发展云计算，各大 IT 企业也在提供多种多样的云存储服务，比较知名的有：阿里云存储、腾讯云存储、京东云存储、彩云等，实际上前几位大厂家已经瓜分了绝大部分的云存储服务市场。

这些年城市都向智能型城市发展，必然会促进新技术的腾飞。有的地区早在七年前就开始建设智慧城市了^[14]。智慧城市的建设是离不开大量数据，大量数据必然依靠云存储技术。

国内三大运营商也有各自的云存储服务。如中移动公司开发的跨平台的彩云服务^[15]。用户可以通过手机、PC 等多终端提交内容到云端备份，例如 139 邮箱的备份；以及飞信和彩云的融合。用户可以使用飞信号直接登陆彩云，然后使得手机端和电脑端的数据同步。联通则有“悦云”^[16]。中国电信推出的云存储服务名字叫“天翼云”。也可以实现手机备份，多终端同步等功能。

目前国内主流的互联网大企业都有自己的云存储服务，这些公司花费大量的资金，聘请高科技人才，争取走在科技的前沿，使得云存储的相关技术在国内得到了巨大的发展。机敏科技软件有限公司也在积极的为用户提供对象存储服务；这样的背景下，对象存储服务得到被重视，也是论文研究的主要目的。

1.3 本文的主要贡献和论文结构

云存储技术在如今信息大爆炸的时代，逐渐成为国内外的研究热点。通过对 Swift 的架构进行分析、对比，其架构、接口都存在需要改进的地方。针对 OpenStack 用户对对象存储的需求，本文分析了现有的研究成果，提出了一种 OpenStack 和 Ceph 结合的方案，该方案可以实现 OpenStack 虚拟机的大规模部署，且没有理论上限，并对外提供对象存储服务；本文对系统中的关键技术，如 OpenStack、Ceph、Swift 等原理进行研究。论文的主要工作如下：

(1) 针对企业的需求，分析了 Swift 架构的不足，开源项目 Ceph 的原理、存储过程，比较了 Ceph 与 Swift 两者的特点，并设计两种 OpenStack 和 Ceph 结合的方案；

(2) 底层的方案设计和部署，建立一个 Ceph 集群，验证集群的运行情况，并在集群基础上，部署 RGW 模块，并改进了 OSD 权重的设置，提高了磁盘利用率。

(3) 参数设计，使系统能够作为 OpenStack 的后端存储；对客户提供的对象存储功能。

(4) 基于 CephRGW 环境, 设计、实现用户指定的功能接口, 然后搭建了云存储系统, 通过部署使用, 验证系统达到了预期目标; 最后测试了系统的高可靠性、分布式特性及自动平衡特性。

本文的组织结构如下所示:

第一章对本项目的开发背景和意义进行阐述, 介绍了目前机敏公司对 Ceph 的使用情况, 接着又简单分析了国内外的一些云存储研发情况, 最后概括了论文的主要贡献和论文结构。

第二章叙述了 OpenStack 和云存储系统, 并对需要设计的系统进行了功能需求分析, 并讨论了 OpenStack 自带的 Swift 模块具有哪些缺点和 Ceph 的优势。

第三章主要是对云存储技术的分析, 详细介绍了云存储和对象存储, 并详细的分析了对象存储系统 Swift 和 Ceph, 并根据两者的差别设计方案, 确定系统的最终方案。

第四章介绍了系统的设计和实现。首先对系统整体架构进行设计, 给出设计方案, 包括底层、接口以及系统管理模块; 最后根据设计方案实现这些接口功能; 并改进了提高 Ceph 磁盘利用率的方法。

第五章主要是系统部署和调试。这一章主要讲述如何进行部署和调试该对象存储系统, 并展示了系统功能及运行效果图, 最后测试验证了系统的性能。

第六章是总结和展望。对论文进行了简要的总结, 分析论文的主要成果, 列出论文的不足之处, 并展望了下一步的工作。

第二章 OpenStack 与云存储系统需求分析

2.1 OpenStack 简介

OpenStack 是由全世界云计算开发人员共同贡献的一款开源软件，自问世以来，一直受到大家的青睐，可以通俗的理解成云操作系统，用它来管理云虚拟机的建立和销毁等。它就是云计算的一个框架，可以嫁接和管理各种架构。有很多学校通过 OpenStack 来部署实验室的环境^[19]，有的国家如印度，则由于安全问题等因素发展较为迟缓，对新技术摇摆不定^[20]，国内 OpenStack 的发展有很多的机遇，很多新型的公司崛起，如机敏、EasyStack 等。目前所在的机敏科技公司就是基于 OpenStack 的二次开发，根据客户的需求进行定制化开发。

2.1.1 OpenStack 架构

OpenStack 组件之间可以相互通信、相互协作，通过架构图来了解各个组件之间如何协同工作的，整体架构如图 2.1 所示：

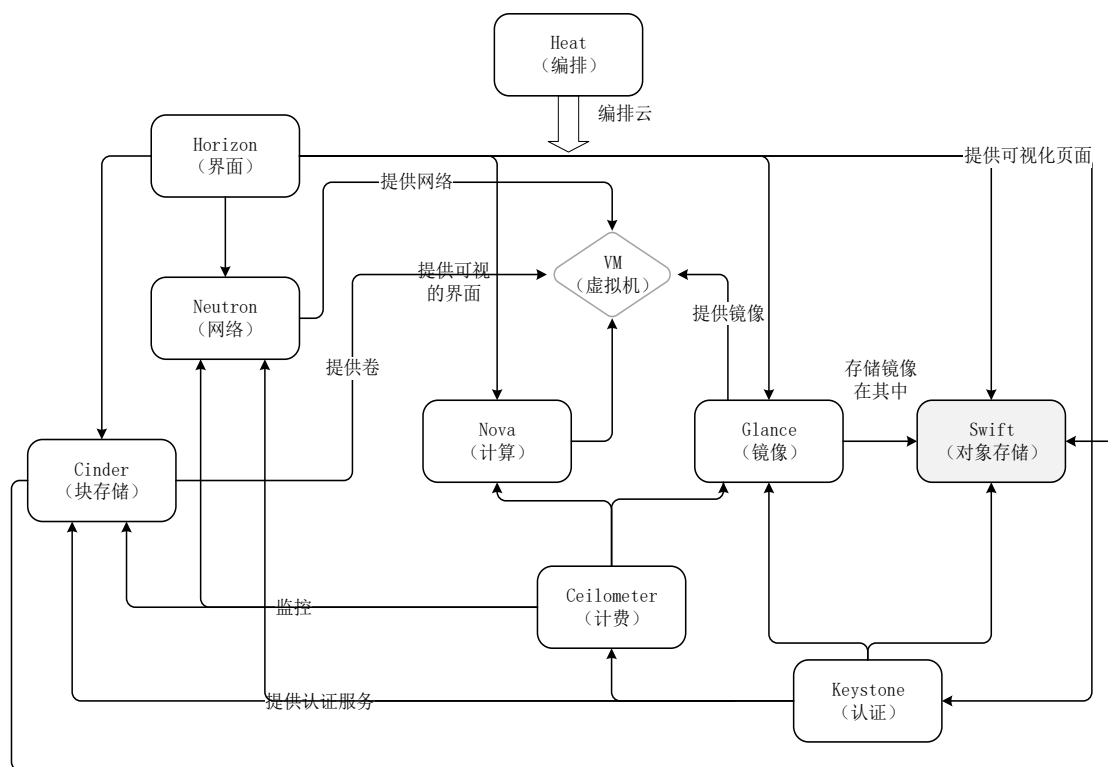


图 2.1 OpenStack 整体架构图

图中箭头都是表示提供服务的关系，如 Keystone 可以为 Swift、Glance、Cinder、Nertron 提供认证服务。OpenStack 中各个组件只负责一部分工作，下节将对其核心组件做简要介绍。

2.1.2 OpenStack 核心组件

本文重点是 CephRGW，所以对 Openstack 的核心组件只做简要的介绍，其他组件 Ceilometer（计费）、Heat（编排）不做介绍：

- Horizon（控制台），又名 Dashboard

Dashboard 就是为管理员和用户提供一个图形化接口，是基于 HTTP 协议，使用浏览器访问的，用户可以访问和管理的资源包括：计算资源、存储资源、网络资源等。Dashboard 具有很高的扩展性，可以定制开发，支持添加第三方的定制模块，就像机敏公司的和安贤公司（提供云安全服务）的整合，在 Dashboard 中不仅添加了计费、监控模块，还添加了安全管理模块。简要说就是用户直接操作的 Web 界面，是用户和底层交互的桥梁，可以根据用户的需求做定制化开发。机敏公司的主要产品机敏云就是根据用户的需求定制化开发，给用户需要提供他们需要的功能模块，模块结构如图 2.2 所示。

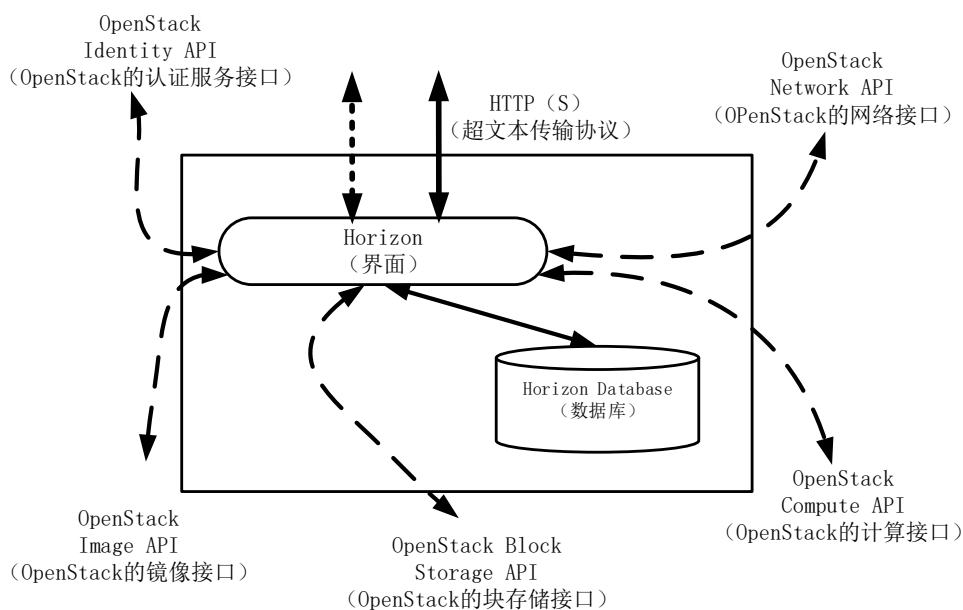


图 2.2 Horizon 示意图

- Nova（计算）

Nova 就是云计算环境中的核心控制器，就像电脑里的 CPU 一样，主要采用 Python 语言编写，是一个成熟的控制节点，它使用的虚拟化技术包括 KVM 和 XenServer，并结合自动化计算来对资源池进行操作，也就是说其负责创建，调度，销毁云主机。OpenStack 只是一个平台，它并不是提供计算资源，可以看作云的操作系统。Nova 的内部结构如图 2.3 所示。

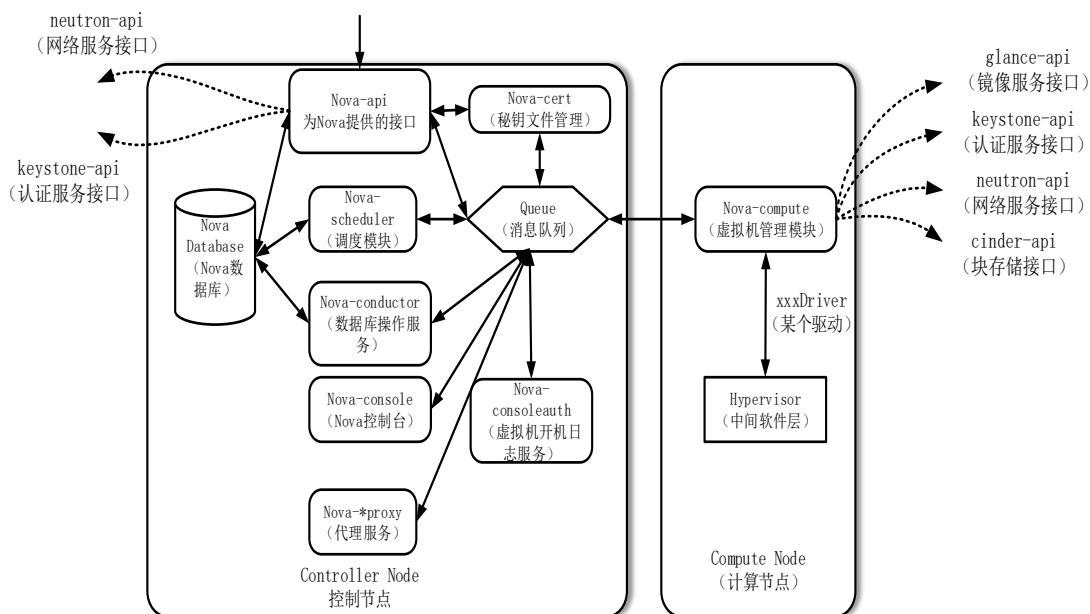


图 2.3 Nova 内部结构示意图

● Neutron (网络)

这部分是 OpenStack 里的网络服务，之前的项目名称叫 Quantum，后来改名为 Neutron。其主要负责云计算大环境下虚拟网络的功能。使得 OpenStack 可以更加灵活便捷的给租户划分物理网络，且在多租户的情况下，能够给每个租户都提供独立的网络环境。用户也可以自己创建网络，控制网络中的流量等，通过它可以人为控制服务器和设备连接到一个网络或连接到多个网络。

● Swift (对象存储)

就是 OpenStack 中的对象存储模块，因为 OpenStack 的存储服务一共有两种，一种是 Swift 提供的对象存储服务，另一种是 Cinder 提供的块存储服务（下面将介绍），Swift 是发展很久的一个对象存储开源项目，它是一个可扩展的存储系统，它将对象和文件分别存储在某一集群上的多台服务器节点上，各台服务器之间有着一定的联系，它必须完成副本的复制、更新，以保证数据的高可靠性、高安全性。而且在集群中可以添加服务器节点完成横向的扩容，它的每个 Zone 都是可复制的，其具有完全对称性。但是 Swift 的设计具有一定的缺陷，它的代理节点限制了流量上限，这是目前很多公司研究 Ceph 来取缔 Swift 的主要原因。Swift 的简要结构如图 2.4 所示。

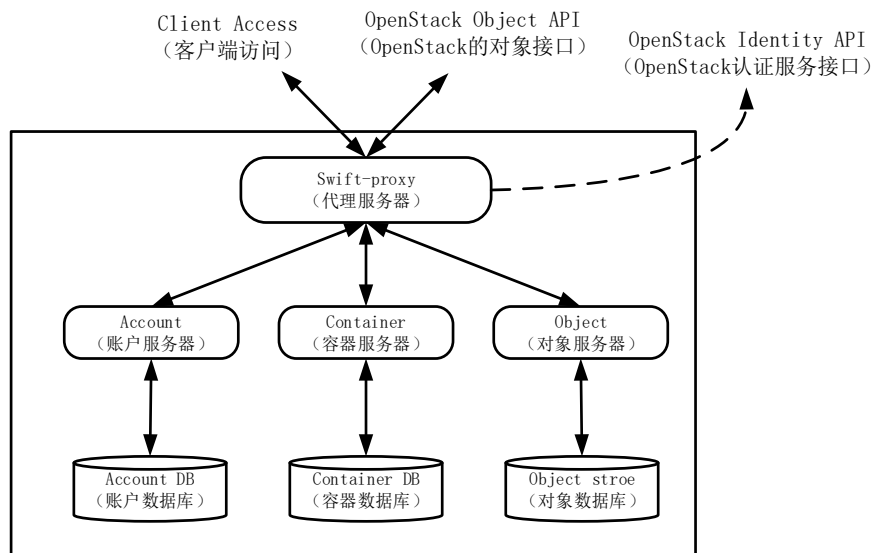


图 2.4 Swift 内部结构示意图

- Cinder (块存储)

Cinder 就是 OpenStack 的块存储模块，它的功能是云环境下提供块设备的创建，添加和卸载。可以简单的将块设备理解为一个磁盘，实际就是一个存储卷。块设备非常适用于性能要求很高的场景，例如数据库等。而且块设备具有一个很强的功能：快照功能，其可以实现存储卷的备份，可以利用快照功能进行数据恢复。简要的理解就是块存储就是给云主机提供附加的云盘。

- Glance (镜像)

提供镜像服务，装机时使用，就是系统镜像。

- Keystone (认证)

就是为各个组件提供认证服务，并进行授权，认证通过后有特定的服务列表记录用户的访问权限，通过查询权限列表来访问各个组件。Keystone 内部结构如图 2.5 所示。

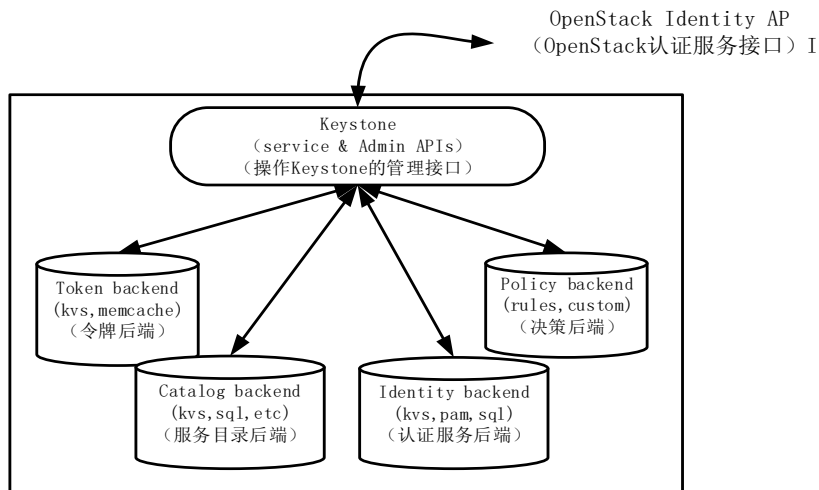


图 2.5 Keystone 内部结构示意图

2.1.3 OpenStack 主要技术

OpenStack 作为全球顶级的开源项目，它涉及到的技术非常广泛，如虚拟化技术、消息总线(MQ)技术等，虚拟化技术是云计算的基础，物理主机的硬件资源是被所有虚拟机共享的，如 CPU、内存、IO 硬件等。提供硬件资源的物理主机常被称为宿主机(Host)，虚拟机常被称为客户机(Guest)。而宿主机将自己的硬件资源虚拟化给客户机使用主要是通过 Hypervisor 技术实现的，它是所有虚拟化技术的核心。OpenStack 中涉及到两种通信，一个是项目之间通过 RESTful 进行通信；另一个则是项目内部不同的服务进程之间的通信，消息总线技术就是负责项目之间如何进行通信；

OpenStack 技术点有很多，所以基于 OpenStack 二次开发的大小型公司也如雨后春笋一般；因为它本身就具有很多的研究点，其中资源调度是一个关键的环节^[21]，涉及到很多的算法，例如怎么去调用虚拟机，怎么感知云计算中的容器位置^[22]等。国内外有很多关于 OpenStack 其他方面的研究，有的从优化部署架构^[23]，快速存储^[24]、信息安全和资源共享^[25]以及怎么去调试 OpenStack 的问题着手研究，这些都是很新型的问题，如使用状态图调试 OpenStack^[26]，不过目前大部分还是基于 FCFS 调度和随机路由算法来用于 OpenStack 的建模分析^[27]。

另外机敏公司目前正在着手做的虚拟机热迁移问题^[28]也是热门的研究方向，热迁移可以给用户带来很好的用户体验。本文着重讲怎么将 CephRGW 和 OpenStack 结合，主要就是利用 Keystone 的验证，和基于 Horizon 的二次开发。

2.2 云存储系统功能需求分析

为了满足用户对系统的使用需求，授权用户必须可以加密登录，对自己的数据进行管理；而且可以随时随地将文件上传到设计的云存储模块中。而且该系统必须有高可靠性；单点损坏的时候必须保证数据的完整性。

- 功能一：该系统可以实现用户加密登录；
- 功能二：该系统可以实现对用户数据的管理，用户可以在 Web 端访问该系统，对文件进行查看，上传，下载等；
- 功能三：该系统将数据存储在不同的节点上，并对数据进行备份；保证数据在单个节点损坏的时候不会丢失数据。
- 功能四：该系统可以自行的扩展存储设备；系统运行时，不需要关机，直接添加存储

设备。

2.3 系统开发问题分析

本文研究 Ceph 和 OpenStack 结合的难点在于：（1）模块之间融合问题，Swift 作为 OpenStack 原生模块，接口已经相当成熟，Ceph 的接口则完全不一样，需要重写（2）底层的设计、部署，模块之间版本、证书冲突等问题；（3）这个系统是基于 CephRGW 开发的，使用 RGW 来进行构造，如何基于 RGW 开发接口，而且接口必须是要支持 Restful 标准的；（4）怎么结合 S3 的命令行来验证用户的授权，使得授权用户可以使用 Amazon 的接口工具访问云存储系统服务。

2.4 本章小结

本章首先对 OpenStack 系统进行了概述，介绍了它的架构、核心组件以及主要技术；然后分别分析了本文开发的系统的功能需求，接着对该系统的开发问题进行了分析。

第三章 云存储的技术分析与 Ceph 的改进

3.1 云存储

3.1.1 云存储定义

云存储，顾名思义就是在用户在云端进行数据的存储和管理，这个概念比云计算的提出晚一点，是一种新兴的网络存储技术，从功能上来说就是指通过集群、网络和分布式文件系统等比较前沿的技术，将互联网中的大量存储设备使用某些方法聚集到一起共同工作，同时它们都能够很好的向用户提供数据的存储服务。

云存储就是在分布式处理(Distributed Computing)^[29]的基础上发展而来，简单的说就是一种使用户可以随时随地管理存储数据的云端存储系统，用户不需要关心数据存储位置以及数据维护等问题。

云存储应用实例如图 3.1 所示：

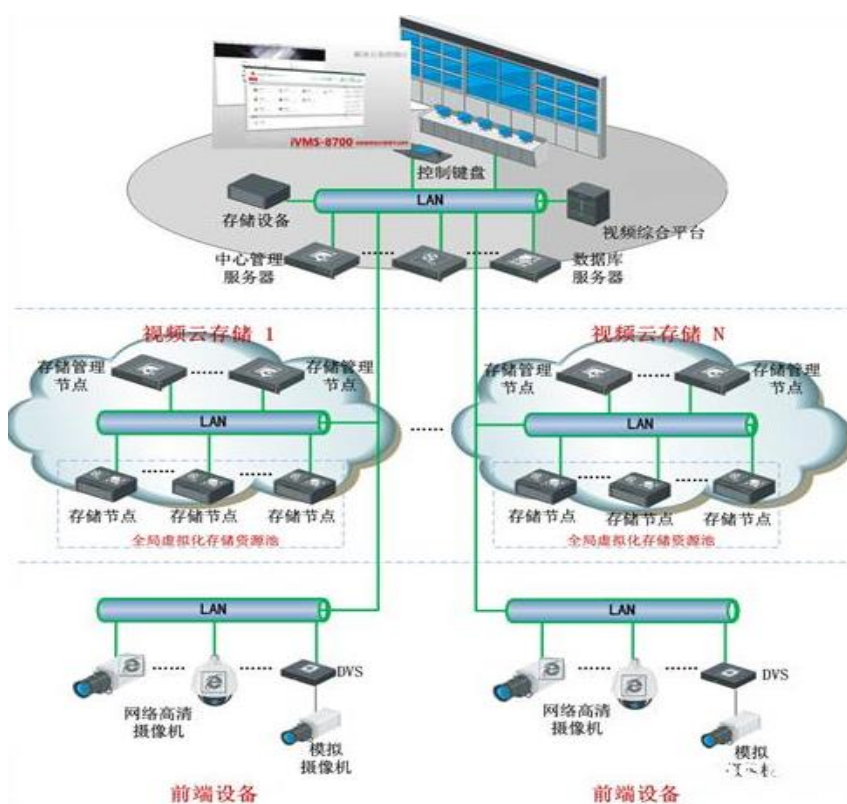


图 3.1 云存储结构示意图

图 3.1 中就是一个视频云存储的应用实例，图中云存储为上万个高清摄像机提供 PB 级的存储空间，不仅解决了用户因为硬盘损坏而导致视频流会丢帧的问题，还能有效解决录像丢

失的情况。云存储集的群系统还能够提供强大的处理能力，极大地提高了可靠性和稳定性以及容错率，它是目前云视频会议高效的保障。

3.1.2 云存储系统结构

传统的存储设备只是一个硬件，而云存储则不同，它不仅仅是一个硬件，而是一个由诸多部分组成的复杂系统，包括网络设备、存储设备、服务器、应用软件、API 接口等等。这里面核心的就是存储设备，它们通过应用软件可以对外提供存储服务。

云存储系统的结构模型主要由四层组成，如图 3.2 所示：

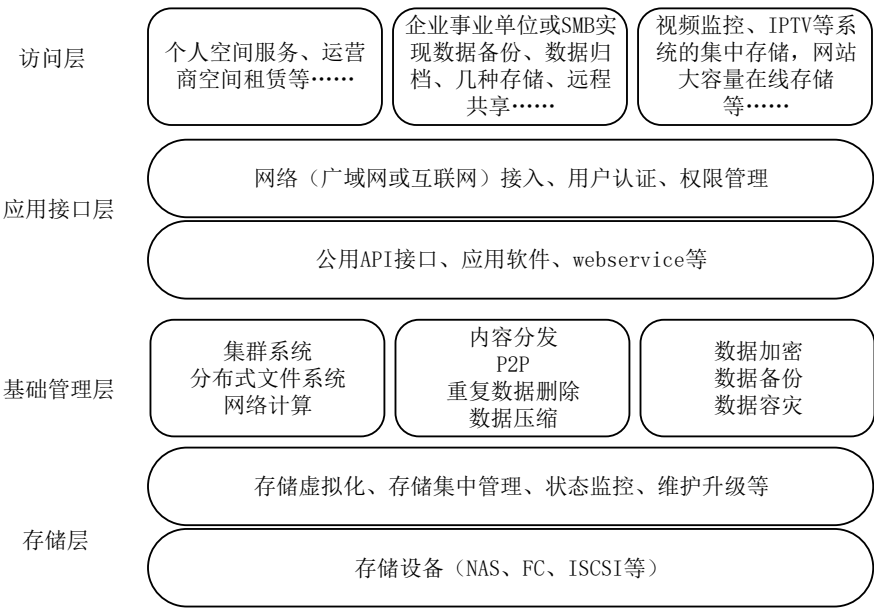


图 3.2 云存储结构模型图

下面对四层结构做简要介绍：

（1）存储层

存储层是最底层的部分，存储设备一般都是数量巨大且分布在不同的地方的服务器，彼此之间通过网络或者光纤通道网络来通信、共享数据。通常情况下在存储设备的上层中都会有一个用于调控这些设备的一个系统，它拥有许多的功能，如设备的虚拟化管理、多链路冗余管理以及对设备的状态进行监控和维护等。

（2）基础管理层

基础管理层不但是云存储系统的结构中最重要的一部分，而且也是这个系统中最困难的一部分。它通过很多互联网技术来实现所有设备一起工作的目标，使用多种备份技术，如设置副本数等，来保证用户数据的安全性。

（3）应用接口层

应用接口层则相对灵活，其可以定制开发，虽然 OpenStack 是开源项目，但是很多公司可以根据用户的需求定制开发接口，提供不同的服务。

（4）访问层

当某一个用户被授权之后，他可以通过一个提供的标准的接口来登录这个云存储服务系统。但是由于云存储技术的多样化，对于不同的运营单位来说，它们都提供了不同的云存储的访问方法和存储的方式。

云存储系统最终呈现给用户的就是访问层，只要向提供商付费成为会员，就可以通过该公司的接口来使用它们的服务，当然不同的公司肯定有多种多样的访问手段。

3.2 对象存储

3.2.1 对象存储定义

（一）对象存储定义

所谓对象存储就是基于对象的一种存储方式^[30]，也就是说存储的文件、视频等内容都会被视为一个个离散单元，而这些单元被称为对象，就是底层不关心用户存储的是什么，它只将其当作一个处理单元来对待。对象这个概念的提出就不再具有像文件一样的目录，它是一种扁平化的描述，没有层级之分，下面就会介绍 Ceph 里面的“桶”的概念，所有被分割的小对象都放在特定的桶中，这个“桶”概念其实就是一个命名空间。

对象存储系统是支持地域扩展的，所以共享数据时的一致性等问题是各个厂商需要考虑的重要问题。每个存储的数据都要满足 CAP 的三个要素：一致性、可用性和分区容忍性。

（二）对象存储数据迁移和访问

一般来说，企业所访问的介质无非就是主机、PC 机、移动端及应用等，对于不同的访问介质来说，对象存储的解决方案是截然不同的。但是倘若应用软件并不支持 HTTP 下 REST API 的访问方式，则需要以传统文件服务器协议的访问方式进行访问，需要添加一个网关来转换协议。当没有了文件存储系统中的 NFS 或者是 CIFS 给应用提供数据之后，面向对象存储系统则需要将位于磁盘上的数据块和文件（这里是指应用可以理解的文件）之间的抽象层给替换掉。目前的对象存储系统都是使用 API 来告诉上层应用如何存储和读取对象的标识。

总的来说，对于面向对象存储的操作，其本质并不会有所改变。大部分开源对象存储系统的操作也就是 POST, GET, PUT 和 DELETE 几种，假设需要上传海量的数据，则只要编

写一个脚本便可以实现。如果数据很大,则可能会不支持,Swift 就有文件大小的限制机制^[31],而 Ceph 没有文件大小的限制,不论多大的对象都会被切分成若干个小的 object,然后根据算法计算被存储的位置等相关信息。

3.2.2 对象存储原理

对象存储系统会将所存的文件封装成一个个小的对象,用户不可以直接打开或者修改所存储的文件,但是可以向 ftp 那样上传和下载文件。而且对象存储系统是没有层级的概念的,是一种扁平化存储方式。也就是只有一个“桶”的概念,不管存储的数据是什么、以及什么文件类型,它都直接会被分割成很多小的 object,然后放在桶里面,桶里面全部都是对象,没有优先级的概念,最大的特点就是对象名称就是一个地址,通过这个地址就可以查到对象存在哪边。假设对象访问权限被设置成“公开”,那么所有用户都可以通过网络来访问到该对象;而且它的拥有者还可以通过 REST API 的方式访问其中的对象。正因为这个特性,使得对象存储的使用场景大多是存储网站数据、移动 APP 或互联网应用的静态数据(如视频、图片、软件的安装包等)。通俗的理解就是:当我们走进超市购物时,大部分人做的第一件事就是存包。而且存包之后,用户并不需要记住包所存放的位置,只要保存好存包小票,也就是存包二维码;购物回来后直接扫描小票,存包的柜子就回自动打开。对象存储也是如此,无论你是把文字、图片和视频存储在云端,用户就得到一个类似二维码的 Key(钥匙);用户如果想要取回这些数据时,只需要把这钥匙交给云存储厂商即可拿到数据,对象存储就是这样的一个简单的过程。

目前分布式的存储架构已经是高性能存储的标准,其优点就是数据的自我恢复功能,但是对数千台服务器和数十亿个对象系统来说,修复副本就非常的困难,目前还没有较好的模型来解决这一问题^[32]。

3.2.3 三种存储结构图

如图 3.3 所示就是常见的几种存储类型,本文主要关注的就是对象存储类型,它们的结构图如下:

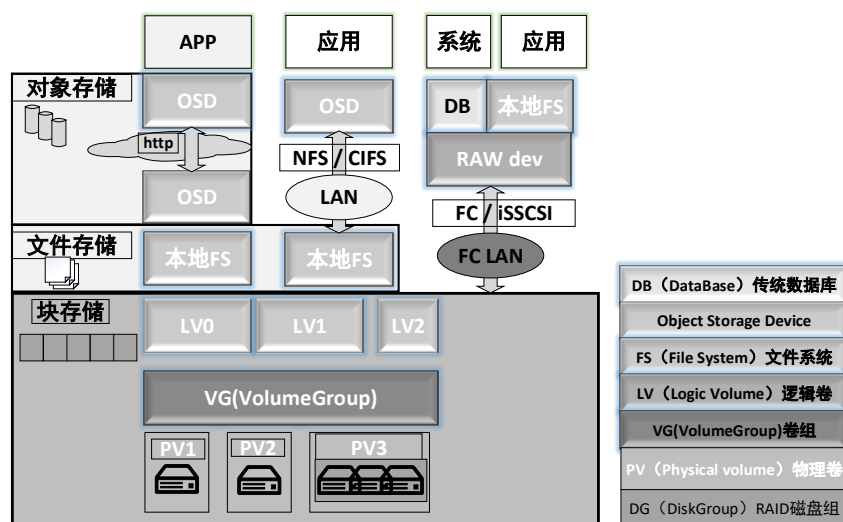


图 3.3 对象存储、块存储及文件存储结构图

块存储：典型的设备有磁盘阵列和硬盘，块存储就是将裸磁盘空间整个映射给主机使用的，这种方式有它的好处，因为通过 Raid(Redundant Arrays of Independent Disks) 和 LVM(Logical Volume Manager)等手段，对数据提供了保护，而且可以将多块廉价的设备组合起来，成为一个大容量的逻辑盘，但是它缺点也很明显，主机之间无法共享数据，在服务器不做集群的情况下，主机 A 的本地盘不能给主机 B 使用，无法共享数据。但是像数据库这类应用需要裸盘映射，此类应用适合使用块存储。

文件存储：典型的设备 FTP、NFS 服务器，文件存储可以克服块存储文件无法共享的问题。文件存储也有软硬一体化的设备，但是只要有一台笔记本或服务器，装上合适的操作系统，就可以架设 FTP 和 NFS 服务了。架设了这样服务之后的服务器，就是文件存储的一种了，只不过读写速度很慢。

对象存储：对象存储是最常用的方案，就是很多台式机内置大容量的硬盘，在装上对象存储软件，再用另外几台服务器作为管理节点，管理节点可以管理其他服务器对外提供读写访问的功能。对象存储克服了块存储和文件存储的缺点，又发扬了它们的优点。

总的来说，块存储读写快，不利于共享，文件存储读写慢，利于共享。对象存储则读写快，也利于共享。

3.3 对象存储 Swift

OpenStack 自带的 Swift 模块也是一个独立的开源项目，它提供对象存储服务，和 OpenStack 融合性好，本节将简要介绍 Swift 的主要特性、系统架构、核心组件和基本原理。

3.3.1 Swift 设计背景

Swift 很早就已经被开发出来了，但是发展和壮大是从它开源之后，当 OpenStack 还只是简单的几个组件（只有三个模块）时，Swift 就已经为 Nova 服务了。它可以支持多租户，也可以支持容器的读写。这些优点都使得它对互联网的数据存储有很大的帮助。因为互联网数据很多都是非结构化的，这正是 Swift 的强项。但是存储互联网的数据必须具有极高的安全性，Swift 中的用户数据安全问题既可以通过用户认证，Keystone 认证等方式，或者通过改进代理服务器来增强数据安全^[33]。其次互联网最重要的就是资源共享，Swift 除了通过 API 的方式共享数据，也可以通过其他方式来加强资源的共享^[34]。

3.3.2 Swift 系统架构

Swift 由三个服务器构成，分别是 Proxy Server（代理服务器）、Storage Server（存储服务器）和 Consistency Server（一致性服务器）^[36]。它的组成如图 3.4 所示，其中 Storage Node 存在两个服务，一个是 Storage 服务，另一个是 Consistency 服务。OpenStack 的 Keystone 服务已经替换了 Swift 中的 Auth 认证服务，其目的是为了让 OpenStack 不同项目之间能统一的认证管理。

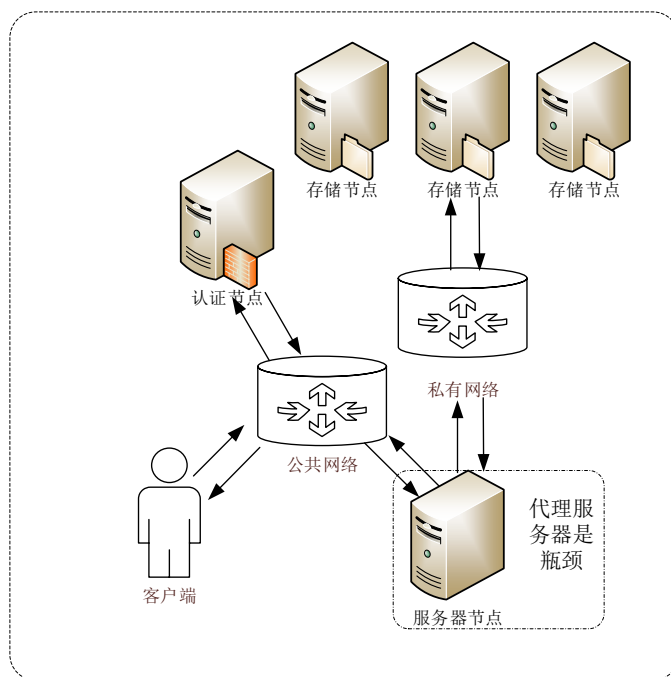


图 3.4 Swift 部署架构

上面简述的是 Swift 的系统架构，相对比较抽象，下面简单分析一个集群部署案例，通过

案例直观的了解各个组件功能。如图 3.5 所示是一种 Swift 集群部署方案：

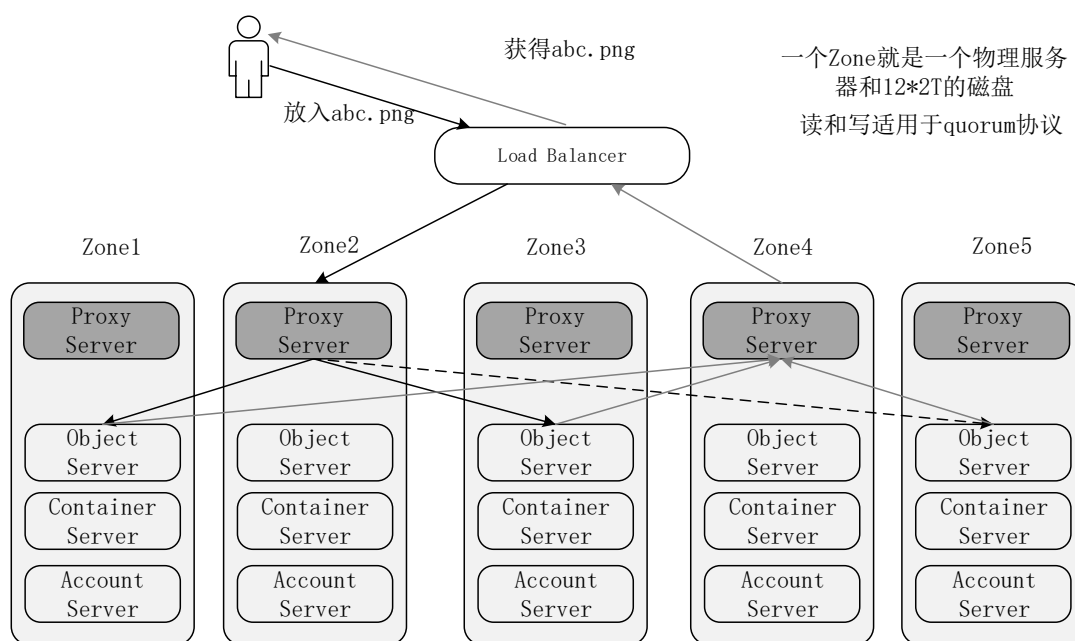


图 3.5 Swift 集群部署实例

这是某公司在测试环境中部署的 Swift 集群，集群又被分为 5 个 Zone，每个 Zone 则是一台物理存储服务器，每台服务器上是由 12 块 2TB 大小的 SATA 磁盘组成，只有操作系统所安装的盘才需要 RAID，其他盘只作为存储节点，是不需要 RAID 的。正如前面所提到的，Swift 是一种完全对称的系统架构，从上图的案例中可以得到很好的体现。图中每个服务器的角色是完全对等的，系统的配置也是完全一样的，都安装了 Swift 服务的软件包，主要有几大组件：Proxy Server、Container Server 和 Account Server 等。图 3.5 中演示了上传和下载文件的数据流过程。当用户将文件上传到集群时，PUT 请求通过负载均衡会随机的挑选一台代理服务器，然后将请求转发给刚选定的 Proxy Server，Proxy Server 会找到存储在本地的 Ring 文件，选择三个不同的 Zone 后端来存储该文件，同时会将这个文件发送到刚刚所选定的三个存储节点。这个过程就必须满足 NWR 策略。简单的说就是一个文件被存储成三份，存储成功的数目必须大于 $3/2$ ，也就是说至少有两份是写入成功的，才会给客户端返回一个存储成功的信息。同样的，在下载文件的时候，GET 请求也会通过负载均衡来随机的选择一个 Proxy Server，该服务器节点上的 Ring 文件是能够查询到此文件是被存储在哪三个节点上的。并同时去后端查询，而且必须有两个（或以上）存储节点返回可以提供该文件的信息时，代理服务器才可以去下载其中一个节点上的该文件。

3.3.3 Swift 核心组件

Swift 的组件很多，下面介绍几个它的核心组件：

(a) Proxy Server

顾名思义，Proxy Server 就是所谓的代理服务器，它是专门用来提供 Swift API 的服务器进程，它的主要功能是让 Swift 中的不同组件能正常的通信。当 Proxy Server 收到某个客户的请求之后，就会在 Ring 中搜索 Account、Container 或 Object 的位置，然后将这个请求转发到别的地方。不过就是由于代理服务器的设计限制了集群的整体吞吐量和伸缩性^[37]。

(b) Storage Server

Storage Server 是一种用于磁盘设备上的存储的服务。Swift 提供的是三种存储服务器：Account、Container 和 Object。其中 Container 服务器的主要功能是复制处理 object 的列表，但是它自身并不存储对象所对应的位置，它只存储在 Container 中存在的 object 对象的名称等信息，最后这些 Object 的信息都会存储在 Sqlite 数据库文件中。与此同时，Container Server 还会对 Object 或者 Container 和进行跟踪统计，例如记录 Object 的总数量和 Container 的使用状态等。

(c) Consistency Servers

存储数据到磁盘上或者是开放对外的 Rest-ful API 接口都是非常容易实现的事情，真正的难点在于怎么处理在这些过程中引发的故障。然而 Swift 中的 Consistency Servers 就很好的解决了这个难点，它的作用是排除由于数据丢失或者损毁，更甚是硬件带来的问题。它主要是由三个服务构成：Auditor、Updater 和 Replicator。Auditor 是在各个 Swift 服务器的后台里面 不停的监控并扫描磁盘，它的功能是检测对象、Container 以及账号的完整性。一旦发现检测到的数据是损坏的，那么该损坏的文件就被 Auditor 转移到专用的隔离区域，紧接着则需要 Replicator 来继续下面的拷贝工作了。Replicator 的主要功能是从其他地方拷贝一个完好的数据来代替该数据，图 3.6 给出了隔离对象的处理流程图。在实际过程中系统有可能会高负荷的运转或者是一些意料之外的特殊情况，此时，Container 或者账号中的数据就不会立即得到更新。如果数据没有成功的得到更新，那么这次的更新就会被立刻添加到本地文件系统的队列中，然后 Updaters 就会来着手处理这些未成功的数据更新了。在 Updaters 中，Account Updater 处理的是 Account 列表的更新，而 Container Updater 则是负责 Object 列表的更新。Replicator 还有一个重要的目的就是保证数据的存储位置是正确的，并具有合法的副本数，这样不管系统遭到什么样的破坏，都能够使得系统具有一致性。

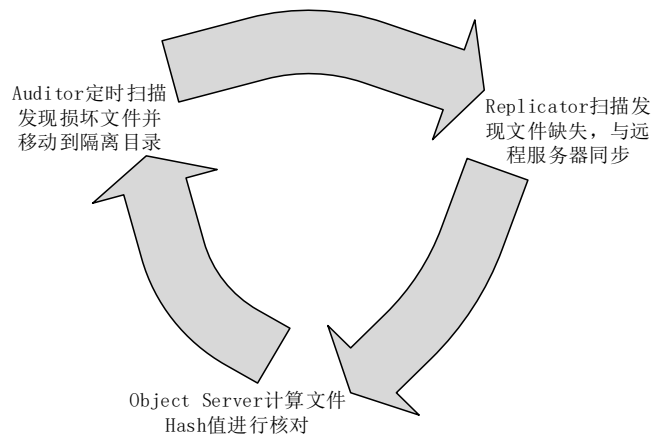


图 3.6 隔离对象的处理流程图

(d) Ring

Ring 则是 Swift 至关重要的组成部分，它存在的意义是记录某种映射关系，这种映射关系就是指被存储的对象和它所在的物理位置之间的一一对应关系。当需要查询 Account、Container、Object 的信息时，就必须查找集群中 Ring 的信息。Ring 使用 Zone、Device、Partition 和 Replica 来维护这些映射信息。Ring 中每个 Partition 在集群中都（默认）有 3 个 Replica。每个 Partition 的位置由 Ring 来维护，并存储在映射中。在数据平衡之后依然需要其他的算法来删除多余的数据，例如重复数据删除技术^[38]可以很大的减少存储开销。

3.3.4 Swift 基本原理

Swift 对象存储所涉及到的算法和基本的理论都是比较简单易懂的，其中最常见的就是以下几个。

(a) 一致性哈希算法

Swift 的基本就是通过一致性哈希算法来构建一个集群用于分布式对象的存储，该集群是冗余的，而且可扩展的。正因为 Swift 采用一致性哈希算法，所以当集群的 Node 数量发生改变的时候，尽可能的保持已经存在 Key 和 Node 的映射关系。该算法的思路可以简述为以下三个步骤：第一步，先计算得到各个节点所对应的哈希值，再将其根据这个哈希值放到一个区间范围为 $0 \sim 2^{32}$ 的圆环上面。第二步，存储对象的哈希值也是使用该计算方法得出，然后也将其放到步骤一中的那个圆环上。第三步，根据该数据的映射位置按照顺时针的方向寻找，找到最近的第一个节点，并将数据保存到这个节点上。假设遍历查找已经超过 2^{32} 这个区间，还没有找到节点，则会将数据放置到第一个节点上。例如在一个环形哈希空间中分布着 4 台

Node, 这时候, 如果需要添加一台新的 Node5, 可以先假设 Node5 的哈希值正好映射在 Node3 和 Node4 之间的某个位置, 那么这个情况下, 受到新添加 Node5 影响的对象是沿 Node5 逆时针遍历到 Node3 之间的所有对象, 因为它们本来的映射都是在 Node4 上的。其分布示意如图 3.7 所示。

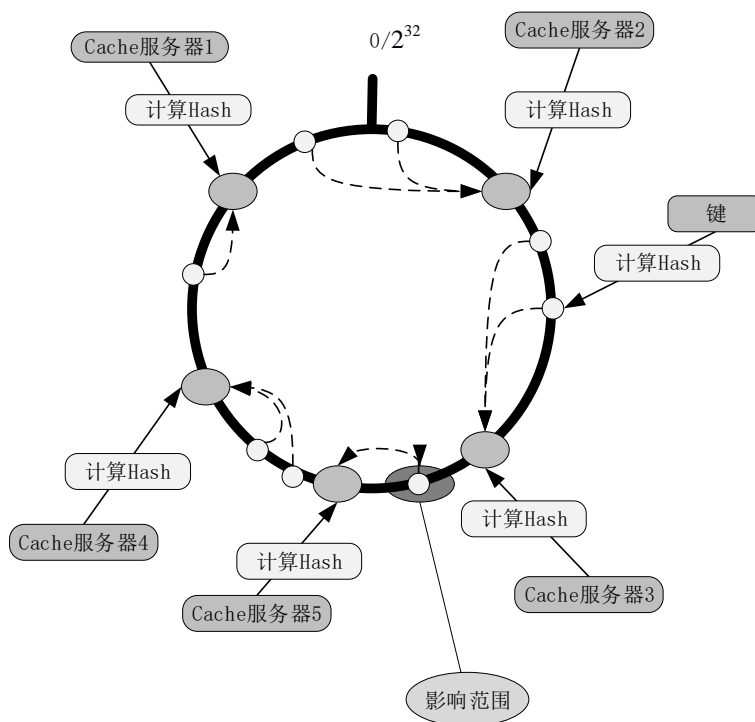


图 3.7 一致性哈希环的结构

(b) Replica

在常见的存储设备中, 可能会出现数据的存储在集群中某个节点上的数量有且仅有一份的情况, 那么当设备发生故障或者其他意外情况的时候, 尤其是一些人为的误操作, 都会造成很严重的后果, 因为此时保存的数据将会永久性的消失。针对此类情况, 就需要有一种方法来保证数据的安全性, 这个方法就是数据的冗余副本。而 **Replica** 就是 **Swift** 中作为副本的一种服务, 它的值一般默认为 3, 其理论依据就是 **NWR** 策略 (亦称为 **Quorum** 协议)。下面对该策略做一个简单的介绍, **NWR** 是一种用于控制一致性的策略, 这种策略在分布式存储系统中很常见, 众所周知, 亚马逊的 **Dynamo** 云储存系统就是使用了 **NWR** 策略来保证了数据的一致性。下面仔细介绍一下这个策略, 首先假设 **N** 代表的是同一份数据的 **Replica** 的份数, **W** 则是在更新一个数据对象的时候能够确保更新成功所需要的份数; **R** 指代的是当读取一个数据的时候所提供的 **Replica** 份数。在分布式的系统中, 为了避免数据发生永久性的一些错误, 数据是不能出现单点的情况, 即正常情况下 **Replica** 的数量不能为 1。因为一旦 **Replica** 为 1 的时候出现了错误, 将无法挽回。如果将 **N** 设置为 2, 一旦其中的一个节点发生了故障, 那

么就立刻变成了单点存储的情况了，所以 N 必须是大于 2 的存在。但并不是说 N 越高越好， N 数越大，系统的成功就越高，工业界通常会把 N 设置为 3。如常用的数据库 MySQL 中，它的 NWR 数值分别是 N 为 2， W 为 1， R 为 1，并没有满足这个策略。但是 Swift 的 N 是为 3， W 为 2， R 为 2 的，这就完全应和了 NWR 策略的说法，即可以说 Swift 是可靠的，不会出现单点故障的系统。

(c) Zone

Zone 的引入解决了在实际情况下的一个问题，即当所有的 Node 都存在于同一个机架上或者同一个机房当中，如果出现了停电、网络损坏或者其他问题的时候，那么用户就不能访问所有的 Node。Zone 是一种独特的机制，它可以对机器的物理位置进行隔离，以满足分区容忍性（CAP 理论中的 P）。Zone 的基本思想是把集群中的 Node 分配到每个 Zone 中。但是需要注意的是同一个 Partition 的 Replica 不能放在同一个 Node 上或同一个 Zone 内。最后，Zone 的大小并不是固定的，它完全可以根据业务需求和硬件条件进行决定，它可以是一块磁盘或者一台存储服务器等。

(d) Weight

Ring 中引入了权重的概念，即就是 Weight，Weight 解决了以后在添加存储能力更大的 Node 时的一个问题，方法则是给它分配到更多的 Partition。例如，在 2TB 容量的 Node 的 Partition 数为 1TB 的两倍的情况下，那么 Weight 的值在 2TB 下就可以设置为 200，而 1TB 的下就是 100。

3.3.5 Swift 主要特性

在 OpenStack 官方非常细致的介绍了它的基本特性，并且在 OpenStack 官网中陈列的 Swift 的几十个特性中最值的关注的几个特性如下所示。

(1) 极高的数据持久性

许多时候人们比较难区分数据的持久性和系统的可用性之间的区别，最简单的区分关系就是数据持久性一般可以认为是数据的可靠性，具体是指在系统中进行的数据存储，在某个时间节点数据丢失的可能性。举个例子，在上文中提到了亚马逊简单存储服务，它的数据持久性是 11 个 9，这个可以被理解是当在 S3 中存储了一万个（4 个 0）文件对象后，时间消逝到一千万（7 个 0）年之后，才可能会丢失存储文件中的一个文件。单纯在理论上讨论，如果 Swift 在 5 个 Zone、5*10 个存储节点的情况下，当数据的副本数是 3 的时候，那么数据的数据持久性就可以达到 10 个 9。

（2） 完全对称的系统架构

完全对称中的对称的意思是指在 Swift 中，各个节点是相当于完全的对等^[35]，这样在系统的维护成本方面得到了很大的降低。同时这也使 Swift 的扩展性得到了极大的扩展，一旦某个节点损坏，修复起来也比较简单。

（3） 无限的可扩展性

可扩展性中的扩展主要是两个方面，其一指的是存储的容量可以拥有无限扩展的能力，其二指的是 Swift 性能，如 QPS 或者吞吐量这些能力得到线性的提升。扩容在 Swift 中是非常简单，一般情况下，由于 Swift 的完全对称性，它的扩容就是将服务器进行增加即可。后续的工作则由系统来完成，如自动的数据迁移，但这有一个前提，就是各个节点的存储量必须有一个新的平衡。

（4） 简单、可依赖

简单不是常见意义上的简单，而是指架构的简单、代码的简洁和实现的可理解性，因为它并不存在大量晦涩难懂的分布式存储理论，取而代之的是一些基础的原则。可依赖性则是指当 Swift 在大量的测试和分析后，就可以很好的应用在最重要的一些存储业务上，这是因为 Swift 在出现问题的时候它可以通过自身来解决问题，即通过错误产生的日志来快速知道错误发生的位置，并自动的解决问题，所以这也是 Swift 一直很火的原因。

3.4 对象存储 Ceph

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 年的 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。由此，Ceph 开始广为人知。Ceph 是一种为优秀的性能、可靠性和可扩展性而设计的统一的、分布式文件系统。而第一个稳定的 Ceph 的版本发布是在 2012 年。

3.4.1 Ceph 设计背景

Ceph 这个系统具有很多优点，但从它的英文介绍^[39]中可以看出它是一个分布式的存储系统。它不仅具有极高的可靠性和扩展性，还能无限的扩容，是真正的无中心化存储。

它的英文介绍确实说出了 Ceph 的要义，可以将其作为理解 Ceph 设计思想的基础，也是了解其实现机制的出发点。这里面最主要也最重要的两个修饰词就是“统一的”和“分布式

的”。这两个概念是对存储系统来说非常重要的修饰词。顾名思义，“统一的”意思就是目前人们常用的存储方式 Ceph 都能够提供，它们分别是对象存储、块存储以及文件存储。这就大大的简化了系统的部署和运维，不需要使用 Cinder 来提供块存储、Swift 提供对象存储。而“分布式”在 Ceph 系统中就是指没有中心结构，没有理论上限，可以无限的扩展，在实践中，也已经被部署在成千上万台的服务器上。早在 2013 年的时候，Ceph 部署的集群就已经达到了存储容量为 3PB 的级别。

事实上，Ceph 开源已经很久了，只是让人感觉一直不温不火，从最初发布到慢慢的逐渐流行，Ceph 花了七年多的时间。本文之所以对 Ceph 和 OpenStack 的结合加以研究，主要从以下两个点来考虑：

第一个点：Ceph 相对 Swift 有很多优势，因为 Swift 是有代理节点的，所以这就造成了吞吐量受到很大的限制，而 Ceph 除了可以提供对象存储功能，还能给用户其他类型的存储服务。另外 Ceph 还可以自动化维护和无限扩容的。

这些优势都是源自于它先进的设计思想，简单的概括就是：对数据的读写操作无需像查目录一样查表，只需要根据 ID 号计算获取地址。就是 Ceph 的先进思想才使得它有非常高的可靠性和非常强的扩展性，没有理论上限，只要条件允许可以一直无限的扩展下去。只要深入的研究就会发现它几乎所有的优点都和它先进的设计思想密切相关的。

第二个点：因为 Ceph 在 OpenStack 的开源社区中受到了空前的关注。

现如今 OpenStack 肯定是最为火热的云操作系统，OpenStack 社区的活跃人数非常多，很多 Swift 的贡献者也开始关注 Ceph，结合两者的优势是势在必行的，只是 Swift 和 OpenStack 的融合度很高，这也是开发者们的一大顾虑。不过 Ceph 已经毋庸置疑的成为了最热的云存储方案。贡献者们越来越活跃，目前 Ceph 中还不够成熟的文件存储系统也将越来越稳定，这也会大大促进 Ceph 和 OpenStack 的融合。

3.4.2 Ceph 设计思想

3.4.2.1 Ceph 应用场景

可以从它的应用场景开始了解 Ceph 的原理和思想，初期它被设计出来的主要目的就是用来应对海量数据，不可避免的需要大规模部署集群，使得它设计初衷就是一个大规模、分布式的存储系统。

目前有专门针对这种大型的存储服务器测试框架，例如雅虎的 YCSB 框架等^[40]。

在 Ceph 初始人的理念中,要想设计这么庞大复杂的存储系统,必须是动态可扩展的,静态的系统终究会因为数据量的爆炸而崩溃,这种动态的特性可以被简述成以下三点:

(a) 第一点是存储系统规模必须是可以动态变化的,可以人为扩容且不影响现有的护具,不论什么巨大规模的系统都不是一天就能预想到的,肯定是随着业务量的增加慢慢积累的,这就意味着系统的规模是越来越大的,不存在最终规模。

(b) 第二点就是该系统中存储设备必须是可以随时更换的:对于一个由几千甚至几万个节点所组成的复杂系统,每个节点都有可能出现故障,所以替换节点也是必然会发生的情况。这就使得系统除了有极高的可靠性之外,还需要极强修复能力,因为数据损坏而更换存储设备的时候不会影响到别的数据,甚至不需要去切断电源。系统也会识别新加入的设备并重新均衡所存储的数据。如此系统的维护成本必将大大降低。

(c) 第三点就是该存储系统必须适应所存数据的频繁变化,因为这种系统的特性,它会经常被用来存储互联网中的数据,而互联网中数据可能是瞬息万变的,所以系统中的数据也会被不断的更新,这就对存储系统要求很高,这是设计之初不得不考虑的问题。

上面所说的这三点就对应了 Ceph 的关键特性。

3.4.2.2 Ceph 技术特性

对应刚刚分析的三点, Ceph 在设计的时候就必须满足的技术特性是:

(a) 高可靠性。顾名思义就是这个系统的安全性,是否可靠,存储在该系统中的数据是否容易丢失;当然也包含操作过程的可靠性,指的就是用户操作 Ceph 系统时候会不会造成数据的丢失。

(b) 高度自动化。自动化就是字面理解的意思,具体来说就是包括 Ceph 能够自动平衡所存储的数据,检测到某个 OSD 损坏的时候会自动的将其 Down 掉。这种高度自动化的好处不仅可以保证系统的可靠性,还可以保证在需要扩容的时候能够大大降低成本。

(c) 高可扩展性。高扩展性包括系统的规模、容量、接口等都可以扩展,高度可扩展就是指系统不仅支持规模的无上限的扩展,底层的接口也是多种多样的,很多应用都可以基于 Ceph 开发。

3.4.2.3 Ceph 设计思路

对应上面分析的三种特性,开发者对 Ceph 的设计思想可被简单总结成如下两个观点:

(a) Ceph 必须尽量利用物理设备本身的计算能力。就是摒弃传统的存储介质只能存储没有统计、检测数据的功能，就是因为其不具备计算能力，因此 Ceph 的创始人认为可以采用具有计算能力的设备作为存储介质，作为存储系统中的一个节点。如常见的服务器就可以作为节点来存储数据，这也是 Ceph 非常重要的设计理念。

(b) 完全的去中心化：不论系统的设计架构如何，只要出现了类似代理节点的东西就必然导致吞吐量会受到限制，当代理服务器上存储的数据过多导致崩溃时，可能所有需要经过这个代理服务器才能寻找的节点都将丢失数据，这就是中心节点的弊端。而且如果中心节点占了很打的网络资源，必然导致网络堵塞，对数据的操作也必然会有延迟，所以 Ceph 的设计必须是完全去中心化，采用新的算法来得到存储地址，不需要特定的节点来存储地址。

3.4.3 Ceph 总体结构

3.4.3.1 Ceph 层次结构

Ceph 存储系统的逻辑层次结构如图 3.8 所示：

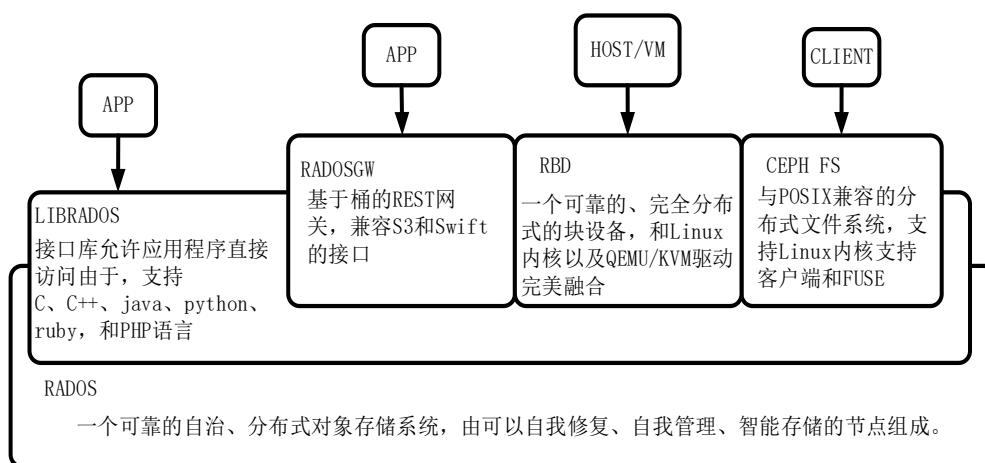


图 3.8 Ceph 存储系统的逻辑层次结构

本节将从下至上详细分析 Ceph 的四层结构^[41]：

(1) 基础存储系统 RADOS (Reliable, Autonomic, Distributed Object Store，即可靠的、自动化的、分布式的对象存储)

RADOS 作为 Ceph 的底层本身就是一个具有完整功能的对象存储系统。只不过因为功能需要，所以将底层的 RADOS 封装之后提供块存储以及文件存储的接口。如块存储就是直接

模拟了块存储的功能，底层还是用的对象存储。所以底层 RADOS 是整个系统的重中之重，也是整个 Ceph 的关键所在。

（2）基础库 librados

这层是对 RADOS 的进行了简单封装，然后将接口暴露给上一层使用，避免直接基于 RADOS 进行开发，可以简单的理解为将 RADOS 封装出一些接口，合并成一个库，然后提供给上层使用。

（3）高层应用接口

这层就属于二次封装了，暴露出的接口就有对象存储功能（RGW）、块存储（RBD）、文件存储（Ceph FS），这里的接口封装层次比 Librados 更高、更抽象，是可以直接提供给客户端使用的，开发者也可以根据自己的需要来选择使用哪种接口。虽然 RGW 也是提供的对象存储功能，但是没有 Librados 的功能强大；RBD 就是提供的块存储功能，Ceph FS 提过的是一个文件系统接口，目前 Ceph 的对象存储功能和块存储功能已经很完善了，本文主要讨论 CephRGW，RBD 和 CephFS 则不做过多介绍。

（4）应用层

这一层就是开发人员基于需求开发的很多应用，如有的网盘应用就是基于 RBD 开发的。基于 Ceph 暴露的接口可以开发很多不同的应用，有的对象存储应用是基于 Librados 进行开发的，或者基于 RGW 也可以开发对象存储应用等。

3.4.3.2 Ceph 整体结构

如下是 Ceph 集群的整体结构，其中包括内部网络和集群网络，这样集群网络可以从公共网络中缓解 OSD 复制和心跳的流量，避免堵塞。具体如图 3.9 所示。

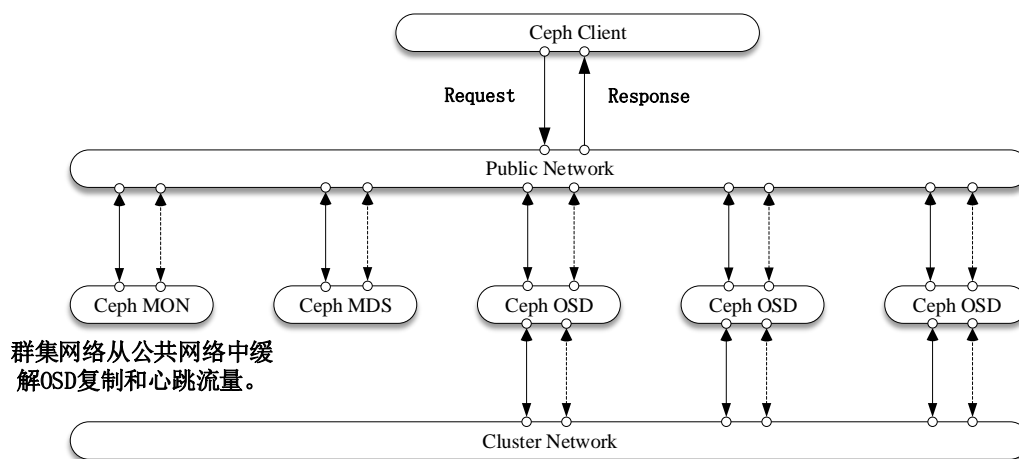


图 3.9 Ceph 集群整体结构

集群中网络规划不是绝对的，OSD 之间的副本通信也可以通过公共网络。集群网络只是为了防止网络拥堵而设计的。

3.4.4 Ceph 工作原理

Ceph 的工作原理非常复杂，不仅节点之间有密切的联系，还有副本之间心跳检测等；本节将介绍 Ceph 关键工作流程，包括对象怎么进行寻址、数据如何进行存取的简单流程，以及 Ceph 集群的维护。下面就从这四个方面来介绍 Ceph 的工作原理，分别是文件切分、寻址流程，数据操作流程，集群维护。

3.4.4.1 文件切分

一、IO 组成部分：

在介绍文件切分之前，先详细介绍一下读写 IO 的组成：

- Offset：在磁盘上的偏移量
- Length：文件的长度
- Data：需要读取的数据

二、对象名的产生

下面对 Ceph 中产生对象名的函数进行讲解，代码如图 3.10 所示：

```
unit64_t cur = offset;
unit64_t cur = len;
while (left > 0) {
    //layout into objects
    unit64_t blockno = cur / su; //which block
    //which horizontal stripe(Y)
    unit64_t stripeno = blockno / stripe_count;
    //which object in the object set (x)
    unit64_t stripepos = blockno % stripe_count;
    //which object set
    unit64_t objectsetno = stripeno / stripes_per_object;
    //object id
    unit64_t objectno = objectsetno * stripe_count + stripepos;

    //find oid, extent
    char buf(strlen(object_format) + 32);
    snprintf(buf, sizeof(buf), object_format, (long long unsigned)objectno);
    object_t oid = buf;
}
```

图 3.10 文件切分图

如图 3.10 所示；cur 就是读写 IO 的 offset，su 就是 4M 大小。cur/su，得到的 bockno 即为对象的序号，

stripe_count 为 1, stripes_per_object 为 1。最后算出 objectno，即为最终的序号。

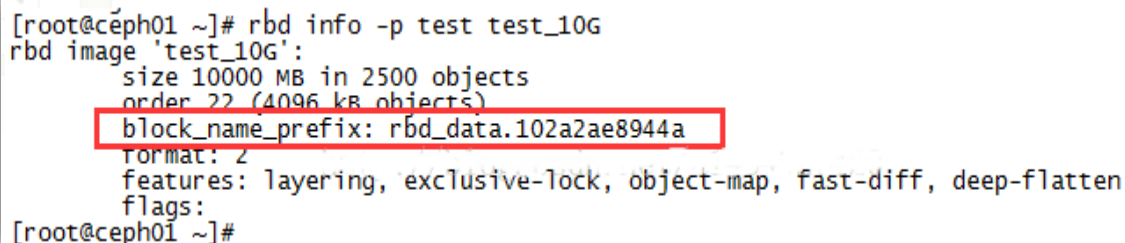
最后通过字符串拼接，把 block_name_prefix + objectno= oid。

三、对象名称的组成：

由图 3.11 所示，一个对象名是由五个部分组成，下面对其分别分析一下：

1、前缀部分 rbd\udata.102a2ae8944a

这是 rbd 镜像里块名称的前缀，可以用 rbd info 查看，通过他可以查找出对象和镜像的从属关系。

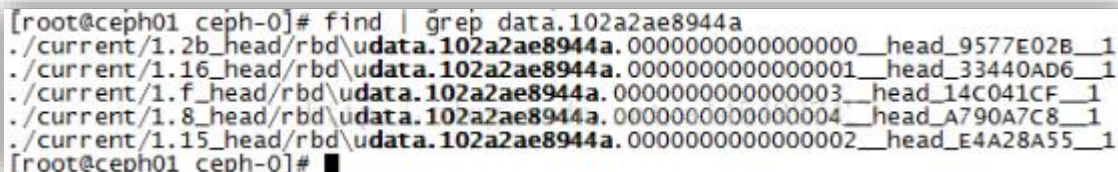


```
[root@ceph01 ~]# rbd info -p test test_10G
rbd image 'test_10G':
    size 10000 MB in 2500 objects
    order 22 (4096 kB objects)
    block_name_prefix: rbd_data.102a2ae8944a
    format: 2
    features: layering, exclusive-lock, object-map, fast-diff, deep-flatten
    flags:
[root@ceph01 ~]#
```

图 3.11 对象名称截图

2、对象的序号，每次通过读写 IO 的 offset 除以对象大小 4M，然后取整数。

假设文件大小为 4194035，计算 4194035/4096 最后结果为 1，那么这次 IO 就写在对象的序号为 0000000000000001 的对象上，从上面可以看出，对象是以 4M，按序切分的。假如写入一个 20M 大小 IO 写，产生的对象序号如图 3.12 所示：



```
[root@ceph01 ceph-0]# find | grep data.102a2ae8944a
./current/1.2b_head/rbd\udata.102a2ae8944a.0000000000000000__head_9577E02B__1
./current/1.16_head/rbd\udata.102a2ae8944a.0000000000000001__head_33440AD6__1
./current/1.f_head/rbd\udata.102a2ae8944a.0000000000000003__head_14C041CF__1
./current/1.8_head/rbd\udata.102a2ae8944a.0000000000000004__head_A790A7C8__1
./current/1.15_head/rbd\udata.102a2ae8944a.0000000000000002__head_E4A28A55__1
[root@ceph01 ceph-0]#
```

图 3.12 对象序号图

3、__head__

表示这个对象在 head 目录下

4、9577E02B

这个是对应的哈希值，这个是对应的哈希值，和文件夹的分层有关系。比如：对象数量

很多，一个目录容纳有点吃力，目录分层性能更好一些：

如图 3.13 所示：

```
rbid\udata.38576b8b4567.000000000000a97d_head_E59B3183__2 rbid\
rbid\udata.38576b8b4567.000000000000aa601_head_60EB4183__2 rbid\
rbid\udata.38576b8b4567.000000000000ab041_head_C8857183__2 rbid\
rbid\udata.38576b8b4567.000000000000ab782_head_092C0183__2 rbid\
rbid\udata.38576b8b4567.000000000000acdb0_head_1A986183__2 rbid\
rbid\udata.38576b8b4567.000000000000ad9fd_head_E2DBA183__2 rbid\
rbid\udata.38576b8b4567.000000000000aecc8_head_8965F183__2 rbid\
rbid\udata.38576b8b4567.000000000000b03d9_head_321A4183__2 rbid\
rbid\udata.38576b8b4567.000000000000b0548_head_77612183__2 rbid\
rbid\udata.38576b8b4567.000000000000b1ac0_head_41269183__2 rbid\
rbid\udata.38576b8b4567.000000000000b2fd9_head_B8660183__2 rbid\
rbid\udata.38576b8b4567.000000000000b5e47_head_0EA91183__2 rbid\
rbid\udata.38576b8b4567.000000000000b77ee_head_67F54183__2 rbid\
rbid\udata.38576b8b4567.000000000000bd264_head_FBFC4183__2 rbid\
[root@cephnode5 DIR_1]# pwd
/var/lib/ceph/osd/ceph-50/current/2.183_head/DIR_3/DIR_8/DIR_1
[root@cephnode5 DIR_1]#
```

图 3.13 目录分层图

从图 3.13 可以看出对象分层后，后缀为 183 的，正好在/DIR_3/DIR_8/DIR_1 目录下，而且正好反过来的。pg 下面对象的目录被分了 3 层。

5、__1

这里的_1 表示 pg 坐在的 pool 序号，也就是 pool id

上面详细介绍了文件的切分，已成的生成了 oid 了，这就每个 object 的唯一序号，接下来根据这个 id 通过映射寻址存储的 osd。

3.4.4.2 寻址流程

Ceph 系统中的寻址流程如图 3.14 所示：

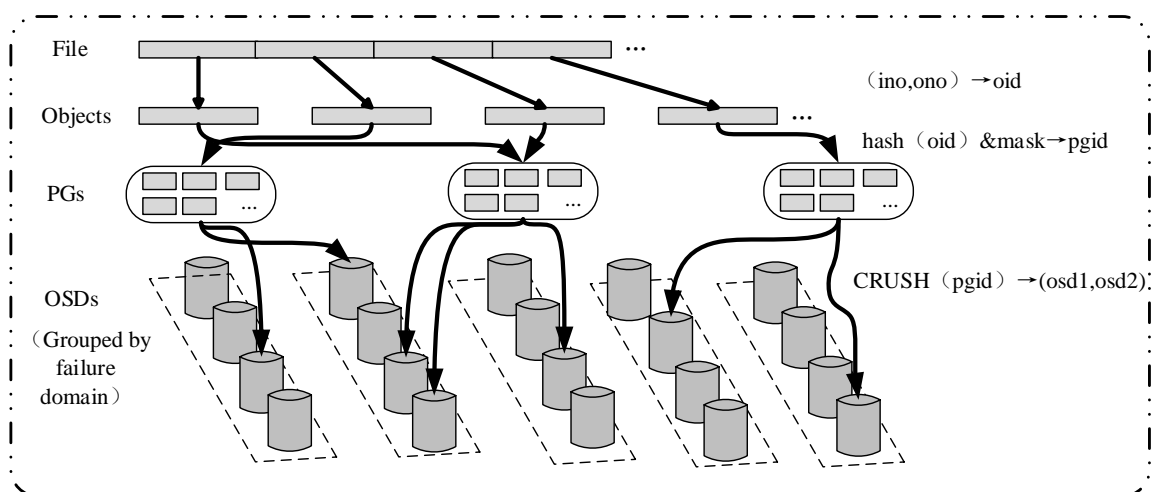


图 3.14 Ceph 系统中的寻址流程图

Ceph 的寻址流程如图 3.14 所示，为了更好的理解寻址流程。接下来先对图中的几个概念简要进行说明：

本节将寻址流程概括为三次映射：

1. 第一次映射是 File 到 object 的映射

完成这次映射，用户存储的文件对象就会被分割成很多小的 Object，这些对象是可以被 RADOS 直接处理的。

这里必须保证 ino 的唯一性，否则后续映射将发生错误。

2. Object -> PG 映射

在 file 被映射为一个或多个 object 之后，就需要将每个 object 独立地映射到一个 PG 中去。这个映射过程也很简单，如图中所示，其计算公式是：

$$\text{hash(oid)} \& \text{mask} \rightarrow \text{pgid}$$

由此可见，其计算由两步组成。第一步是使用 Ceph 系统指定的一个静态哈希函数将 oid 的哈希值计算出来，再将 oid 映射成为一个近似均匀分布的伪随机值。然后再将这个伪随机值和 mask（根据 RADOS 的设计，给定 PG 的总数为 m（m 应该为 2 的整数幂），则 mask 的值为 m-1。）按位进行相与运算，最终才得到的 PG 序号（pgid）。

3. PG -> OSD 映射

第三次映射至关重要，就是将作为 Object 的逻辑组织单元的 PG 映射到 OSD 上，OSD 是存储数据的实际存储单元。RADOS 是采用 CRUSH 的算法，只需将 pgid 代入其中，然后得到一组共 n 个 OSD。这 n 个 OSD 即共同负责存储和维护一个 PG 中的所有 object。

至此，整个映射过程就结束了。这里明显没有任何的全局性查表操作。只需算算就可以知道数据被存储在哪个位置。

3.4.4.2 数据操作流程

本节介绍 Ceph 集群对数据的操作流程，包括数据的查找和数据读写；如图 3.15 所示，假设待写入的 file 较小，无需切分，仅被映射为一个 object。其次，假定系统中一个 PG 被映射到 3 个 OSD 上。

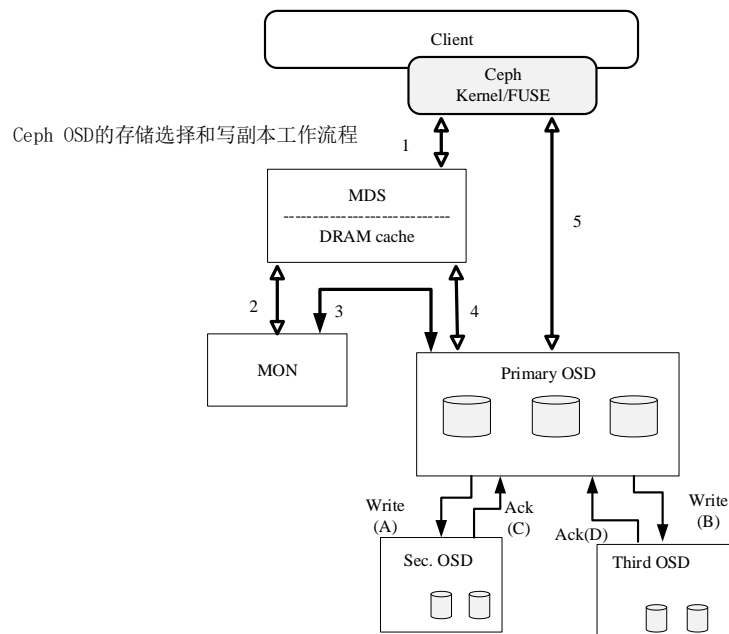


图 3.15 Ceph 数据操作流程

如图 3.15 所示，当某个客户端 client 发出需要向 Ceph 集群写入一个 file 请求时，首先需要在本地完成上节中所叙的寻址流程，将 file 变为一个 object，然后找出存储该 object 的一组三个 OSD。这三个 OSD 具有相互不同的序号，序号最靠前的那个 OSD 就是该组中的 Primary OSD，而后两个则依次是 Secondary OSD 和 Third OSD。

当找出三个 OSD 后，client 将直接与 Primary OSD 进行通信，发起写入操作（步骤 1）。当 Primary OSD 收到请求后，再分别向 Secondary OSD 和 Third OSD 发起写入操作（步骤 2、3）。当 Secondary OSD 和 Third OSD 完成各自的写入操作之后，会分别向 Primary OSD 发送确认信息（步骤 4、5）。当 Primary OSD 收到确认信号，确信其他两个 OSD 的写入完成后，则自己也完成数据写入，并向 client 发出信号，确认 object 写入操作已经完成（步骤 6）。

从上述流程可以看出，在正常情况下，client 可以独立完成 OSD 寻址操作，而不必依赖于其他系统模块。因此，大量的 client 可以同时和大量的 OSD 进行并行操作。同时，如果一个 file 被切分成多个 object，这多个 object 也可被并行发送至多个 OSD。

当用户需要读取数据，client 只需完成同样的寻址过程，并直接和 Primary OSD 进行通信。当前的 Ceph 设计中，被读取的数据都是由 Primary OSD 提供的。

3.4.4.3 集群工作原理

由若干个 Monitor 共同负责整个 Ceph 集群中所有 OSD 状态的发现与记录，并且会形成一个 Cluster Map 的 Master 版本，然后扩散到所有 OSD 和 Client 上。OSD 则使用 Cluster Map 进行数据的寻址。

Monitor 并不会主动的去轮询各个 OSD 的状态。恰恰相反，每个 OSD 都需要向 Monitor 上报自己的状态信息。常见的上报情况有两种：一种是新的 OSD 被加入到集群中了，第二种是某个 OSD 发现自身有异常。再接受到这些信息之后，Monitor 会更新 Cluster Map 并扩散出去。集群内部通信如图 3.16 所示：

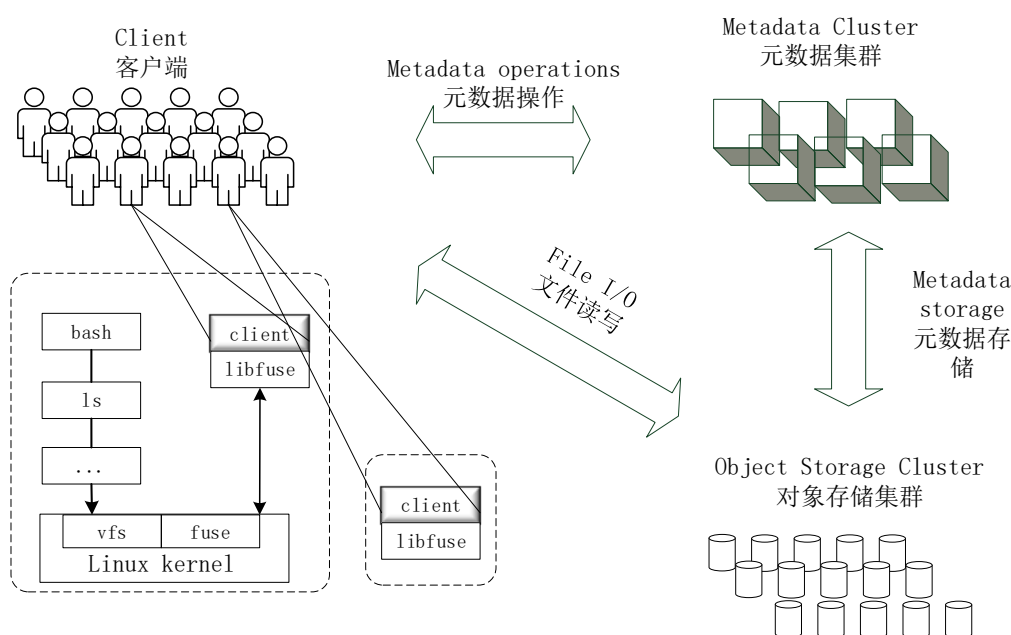


图 3.16 集群内部通信图

集群内部正确的通信是 OSD 和信息存储正常工作的基础，也是系统自动平衡的必要前提。

3.5 Ceph 和 Swift 比较

对于 Swift，厂商需要开发专有的 Swift API，这些 API 不仅和 Ceph 不同，和 Amazon S3 也不一样。

Swift 和 Ceph 在数据一致性的管理方面有所不同。Swift 的副本是异步写入，有可能导致不完整的更新，读取了一个错误的的数据。

Ceph 使用同步过程，需要在确认写入完成之前写入法定数量的副本。

Swift 安装的时候需要安装大量的依赖模块，然而 Ceph 的架构却是非常简洁。

如图 3.17 是 Swift 的结构图：

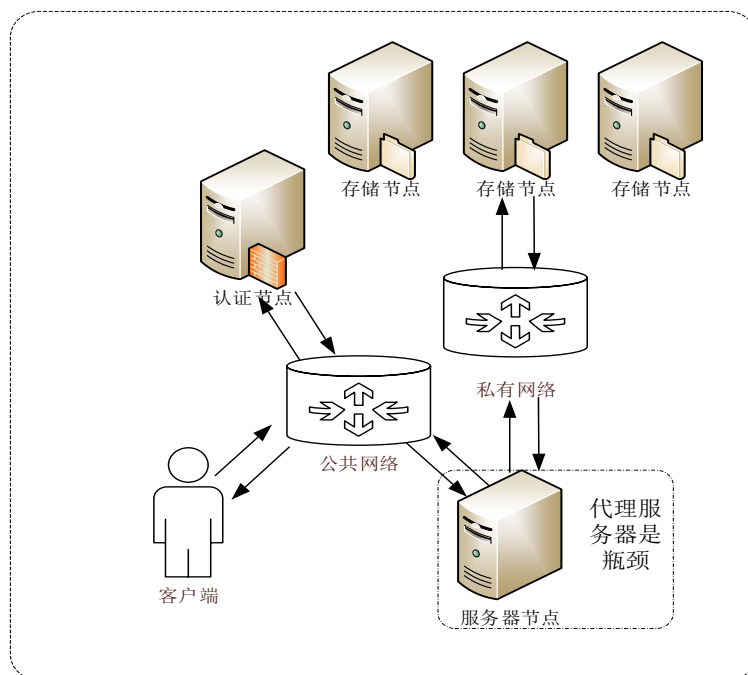


图 3.17 Swift 的结构图

从图 3.17 可以看出所有的流量必须要经过代理服务器节点，所以它成了 Swift 大规模的瓶颈。

Ceph 最主要的优势：

- (a) 模块简单。
- (b) 用更好的算法来处理数据备份，从这点上来说，Ceph 系统允许用户更加灵活地处理大规模所带来的问题。
- (c) Ceph 经过三次算法寻址，就可以准确找到 OSD 的位置。

3.5.1 两者相同点

Swift 和 Ceph 最主要的共同点就是它们都能提供对象存储服务，Swift 专一负责对象存储，这个方面比 Ceph 强一些。它最初就是一个独立的开源项目，它在 OpenStack 开始发展之初就出现了，它是 OpenStack 的核心项目，确实非常强大而且稳定。关键是，Swift 的架构设计导致在传输速度和延迟时间上都不太强。造成这个问题的主要原因就是 Swift 集群中所有进出的流量都要通过代理服务器。

3.5.2 两者不同点

两者的不同点有很多，下面介绍一下两者的不同点：

1. 开发语言：

Ceph 使用 C++ 语言开发，Swift 使用 Python 语言开发；速度性能上 C++ 明显要强于 Python 语言。

2. 支持的存储方式：

这是很重要的不同点，Ceph 支持统一存储，而 Swift 不支持块存储和文件存储；这就是 Ceph 的优势，它可以支持三种存储，使得用户不用维护多个模块。

3. 制定的标准方面：

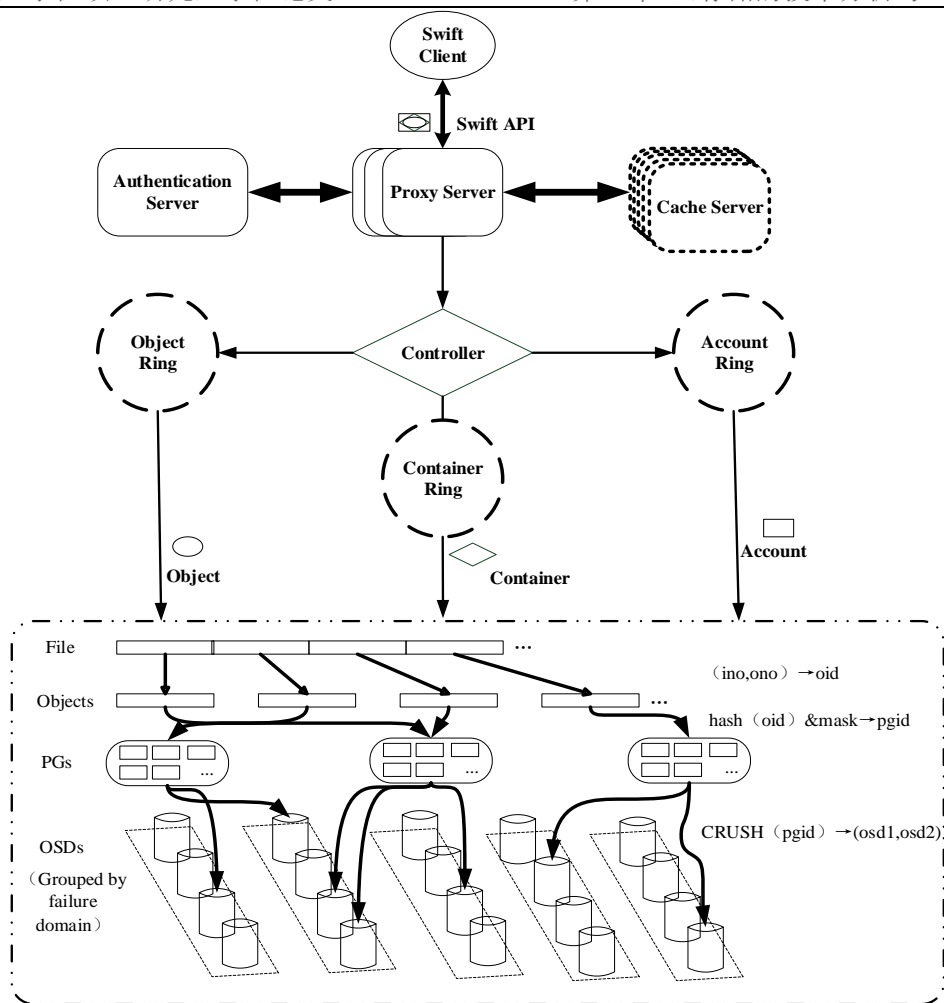
两者在设计之初就设置了很多标准，这些两者都不一样；如一致性、数据放置方法，读操作，写操作，安全性等^[42]。

3.6 对象存储系统方案研究

根据两者的不同，以及 Ceph 目前研究情况，本节提出两种设计方案。

3.6.1 系统设计方案一

Swift 一直作为 OpenStack 对象存储模块，具有丰富的接口，其它组件在使用 Swift 的对象存储功能时，模块之间不会出现兼容性问题；所以方案一保留 Swift 对其它组件的接口；底层使用更优秀的 Ceph 作为存储后端。就是上层的接口依然使用的是 Swift 接口，只是底层代理服务器节点和 Ceph 进行交互，主要研究 Ceph 怎么对接 Swift 的 API，这是一种思路，需要写接口实现。如图 3.18 是方案一的架构图：



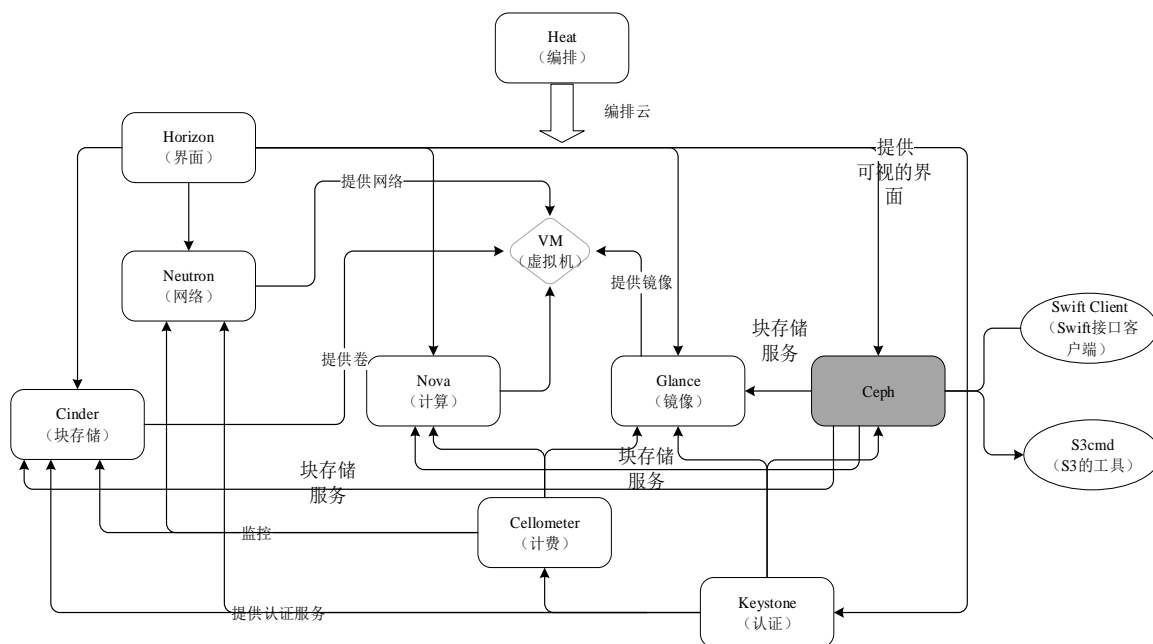
3.18 方案一 设计架构图

如图 3.18 所示，使用 Ceph 作为 Swift 的后端存储，用 Swift 的接口调用 Ceph；对象存储 RGW 兼容 S3^[43]和 Swift 的接口协议，Swift 打包管理，变成一个个对象，最后存储在 Ceph 的集群上，Ceph 底层不需要管谁来调用接口，这样就可以使用 Swift 和 OpenStack 的接口，可以无缝融合。

3.6.2 系统方案设计二

第二种方案是直接使用 Ceph 取代整个 Swift 模块，虽然 Swift 是 OpenStack 原生的模块，但是它有诸多缺点，而且除了对象存储功能，它并不能提供块存储和文件存储等。所以可以直接使用 Ceph 来取代 Swift 模块。

如图 3.18 是方案二的架构图：



3.19 方案二 设计架构图

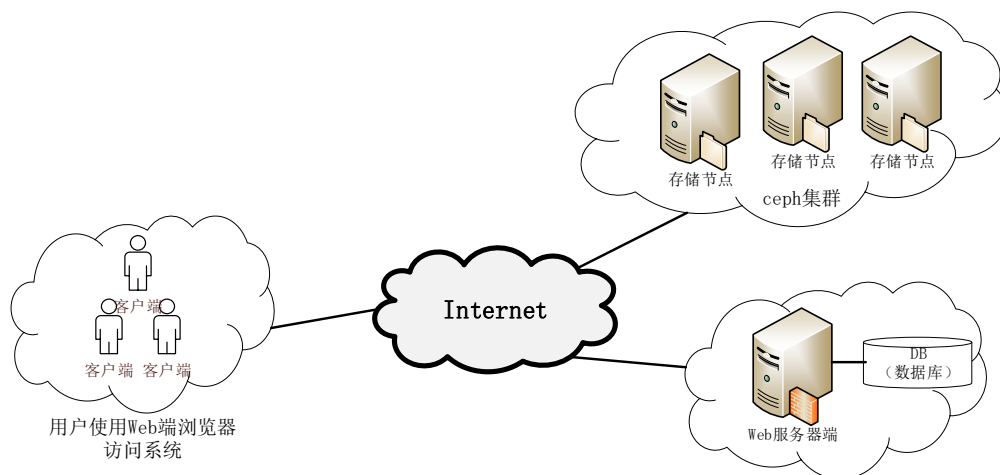
如图 3.19 所示，Ceph 完全取代了 Swift 模块，它给 Cinder、Nova、Glance 提供块存储，并对外提供对象存储服务；Nova 是计算模块，它使用的后端存储由 Ceph 提供；Neutron 则是网络模块，它负责各个组件之间的网络问题；Keystone 则给各个组件进行认证，Ceph 可为其记录认证信息；Nova 创建虚拟机 VM 的时候，由 Glance 提供镜像，并使用 Cinder 作为逻辑卷；Horizon 则是给各个组件提供可视化的界面。目前公司对 Ceph 的依赖仅仅是块存储，等本文的对象存储功能上线之后，会在新的版本中改进 Ceph 的性能，目前研发小组已经着手研究，如 Ceph 块的设计^[44]、多线程给 Ceph 带来的读取优化以及优化^[45]，硬件研发也会基于硬件优化 Ceph 的性能^[46]。

3.6.3 系统方案比较

由于方案一依然保留 Swift 的代理服务器节点；所以对于流量还是存在上限；而且需要同时维护两个组件，这使得维护开销非常大；而方案二则没有这个缺点，因为由于 Ceph 模块全部取代了 Swift 模块，而且 Ceph 能够提供统一的存储；这使得方案二比方案一优势明显；本文最终采用第二个方案设计云存储系统。

3.6.4 系统总体架构

最终本文采用方案二的架构图，本节主要对该系统的总体架构进行设计。图 3.20 是云存储系统的总体架构；



3.20 系统总体架构

如图所示，系统采用浏览器/服务器（Browser/Server，B/S）模式，使用浏览器访问 Web 服务器 Apache/Nginx，再将 Apache 服务器作为分布式集群客户端和 Ceph 集群进行交互，集群可以被无限扩展，网络提供客户端和服务端之间的连接，实现低成本、可扩展、有效的云存储系统。

3.7 本章小结

OpenStack 已经发展很多年，云存储的本质就是利用集群技术、网络技术、分布式技术等，为用户提供基于网络的数据存储和备份服务，客户可以访问和管理自己存储的东西。本文设计的对象存储系统主要面对企业客户，是需要对存储介质进行管理的，所以比常见的如百度网盘，多了一个 OSD 管理功能，企业客户可以自行的添加 OSD。总的来说其中涉及的技术非常多，本章主要介绍几个相关的技术以及概念、原理等。分别是云存储技术和概念、对象存储技术和概念、Ceph 及 Swift 介绍。然后对两者进行比较，给出设计方案并比较两者的优劣确定最终方案。最后根据最终方案来确定系统的总体架构。

第四章 对象存储系统的设计与实现

4.1 接口设计相关技术介绍

4.1.1 接口开发框架

本文采用 Django 框架来进行接口的开发, 经过几个版本的更新之后, 目前的 Django 框架已经非常强大, 设计思想很先进, 图 4.1 就是 horizon 二次开发的必要框架。

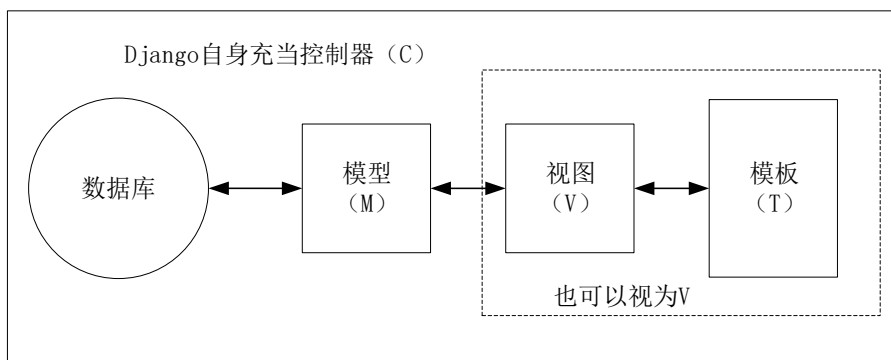


图 4.1 MVC 模式

毫无疑问, MVC^[47]是日常开发中经常用到的设计模式, 通俗的讲就是将应用程序分解成三个独立的部分:model(模型),view(视图),和 controller(控制器)。

在本文中, 采用一个典型的 MVC 模型, 即 Django 框架。Django 模式也被当作 MTV 模式, 是由于用户所用的输入, 交给控制器之后, 由 Django 框架自行处理, 因此去除了控制器这一模块的之后, Django 更专注于三大部分, 即模型, 模板以及视图。

V (View, 视图), 业务逻辑层。这一层其实是模型与模板之间的一个衔接层, 包括了模型的存取, 以及模板之间的逻辑关系

T (Template, 模板), 表现层。模板层主要处理的是显示页面以及在其他格式的文档中进行拓展, 与内容的表示表现相关。

M (Model, 模型), 数据存取层。顾名思义, 该层主要处理数据相关的操作, 例如数据的存取, 数据的验证, 数据的行为校验以及关系验证。

OpenStack 的各个组件都是可以独立开发的, 而且公司的开发都是使用 Django 框架, 如下图 4.2 所示, 每个 panel 都是一个单独的文件夹, 可以模仿 Overview 的文件夹方式来创建三个 panel。

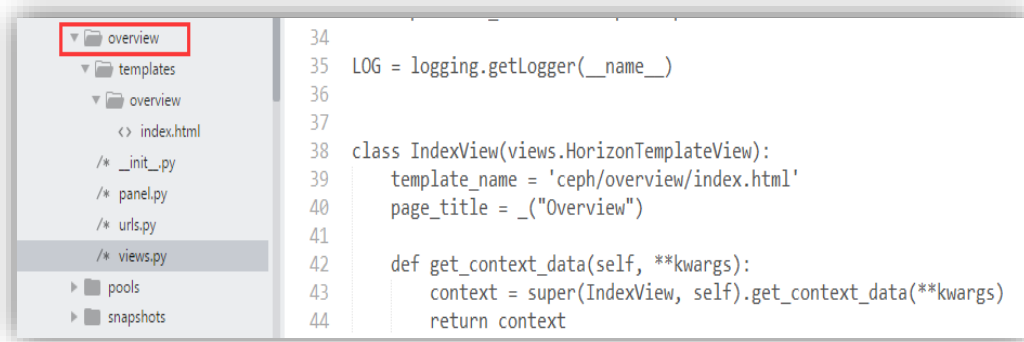


图 4.2 Overview 视图的 panel 结构

4.1.2 接口开发要求

为了能够给客户提供直接简单便捷的程序，需要做一些接口的设计。Ceph RGW 界面是 Dashboard 的一个 panel。本系统是基于 OpenStack 和 Ceph 的，底层是 Ceph 搭建的集群，但是目前操作 Ceph 都是通过 SSH 来发送命令到 Linux 系统，然后以命令来查询数据，倘若能开发一个接口工程，专门用来接收前端用户发来的请求。对接收到的参数做解析之后，告知 Ceph 集群中的 Monitor，让它来执行请求，然后再回写响应的信息，包括出错的信息。

所以接口必须满足以下几个要求：

第一：必须支持常规的前端操作请求，如 GET、POST 等。

第二：接口必须独立封装在 Horizon/dashboard 中，互相之间不能有干扰，是为了防止接口出错的情况出现。

第三：接口和 Ceph 集群之间使用 SSH 命令行进行沟通。

4.1.3 接口设计思路

因为三台服务器都是装的 Centos 系统，而且对于 Ceph 来说，查询集群状态都是通过发送 SSH 命令给集群来交互的，所以接口必须还是采用这种方式。

在开发接口的项目中，必须完成创建集群用户命令、添加 OSD 命令、查看集群运行状态命令等命令的封装。外部的应用程序则只需要用 URL 的方式发出请求即可，然后返回 json 格式的数据。

在本次开发的系统中，为了实现客户端与存储底层之间的交互，接口层的相关操作的主要功能就是通过对底层存储进行封装。接口层是应用层与底层的沟通桥梁，可以实现对象存

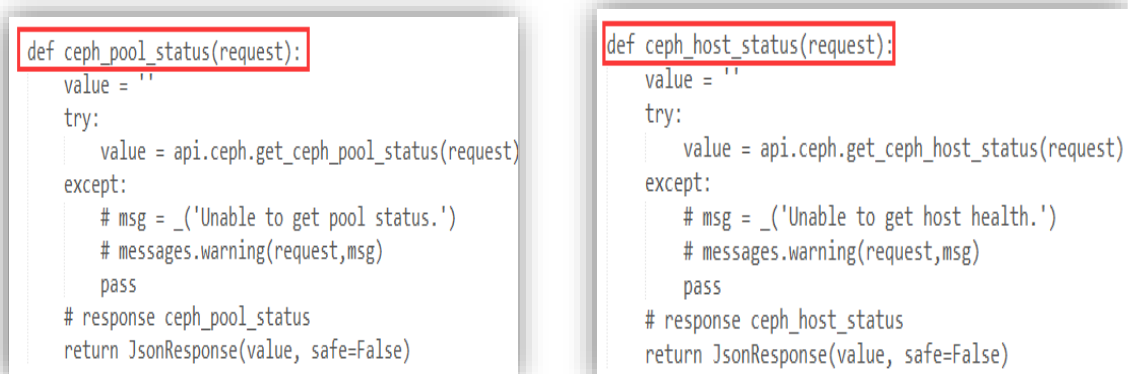
储的跨平台应用，同时具有开放端口给其它应用程序调用的功能。因此，其具有平台化，松耦合等功能。下面对接口层的开发实现进行详细介绍。

在本文设计的对象存储系统中对接口层进行封装的主要方法有：建立与删除用户，文件上传与下载，存储空间的管理等应用。接口对常用的应用进行了封装，并以 Restful API 的方式向应用层提供调用服务，而后以 json 方式进行结果的反馈。

Django 框架对接口层的 Restful API 进行了实现。其中 view.py 都是做控制和逻辑处理的，

Urls.py 里写的是 URL 配置，具有 Django 所支撑的网站目录的功能，这是该文件的本质模式，要为该 URL 模式提供调用的视图函数映射表。以此告诉 Django 不同 URL 调用不同的代码段。URL 的加载时从配置文件中开始的。

若开发 Overview 目录下展示 pool 的状态，在 index.html 默认显示的页面中定义一个 ajax，这个 ajax 就是定义的一个接口，它负责去拿数据。它不仅定义了 url，还定义了返回的数据为 json 格式等，并在 urls.py 中定义了接口，而函数的定义写在 view.py 中，或者将函数实体定义在 api 目录下的 Ceph 文件夹下，主要代码如图 4.3 所示：



```
def ceph_pool_status(request):
    value = ''
    try:
        value = api.ceph.get_ceph_pool_status(request)
    except:
        # msg = _('Unable to get pool status.')
        # messages.warning(request,msg)
        pass
    # response ceph_pool_status
    return JsonResponse(value, safe=False)
```

```
def ceph_host_status(request):
    value = ''
    try:
        value = api.ceph.get_ceph_host_status(request)
    except:
        # msg = _('Unable to get host health.')
        # messages.warning(request,msg)
        pass
    # response ceph_host_status
    return JsonResponse(value, safe=False)
```

图 4.3 接口代码示意图

4.2 底层的设计

4.2.1 底层设计主要过程

设计的对象存储的底层，主要就是以 Ceph 集群为基础，由于公司服务器数量的限制，本节只搭建了一个三节点的 Ceph 的集群。下面是本文对系统的底层设计：

(1) 网络设置：

将三个节点的 IP 分别设置为不同的 IP 地址，选取其中一个作为 Monitor 的节点。另外两

个服务器作为 OSD 节点。并需要设置两个网络，一个作为公共网络，一个作为集群网络。

(2) OSD 参数设置:

由于 OSD 之间的机制有限制，为了数据安全，一般的副本数被设置成 3，实验时不涉及到大量的数据，将 OSD 副本数改为 1。三台服务器之间必须能够通过 Ceph-deploy 进行 SSH 访问，能够以最小的基础措施安装 Ceph 集群。

(3) 节点权限设置:

节点之间必须能够通信，并且 Monitor 节点需要有权限来读取 config 文件，每个节点的用户必须有很高的权限，且需要安装 ntp 及 openssl-server 使主机能直接访问 OSD 节点，需要在主节点上登记节点的名称和 IP 地址。

(4) Monitor 初始化设置:

Monitor 的初始化必须保证三个节点之间正常通信，可以实现 OSD 的心跳检测；可以在 node1 上执行命令 Ceph-deploy install node1 node2 node3，安装完成之后，编辑 Ceph.conf 配置文件，然后执行 Monitor 的初始化。

(5) Ceph 网关设置:

Ceph 作为一个集群，和 Apache 或者 Nginx 服务器之间如何通信，这就涉及到网关的设置，需要 Ceph 网关接口来完成这项任务，Radosgw 其作用可以看成是 rados 和 HTTP 的结合，就是对 rados 进一步的封装。因此 radosgw 可以实现用户通过 HTTP 网关来访问 Ceph。

4.2.2 底层设计逻辑结构图

底层通过 S3 的接口与前端 Web 界面交互，实现为用户提供管理和操作对象存储系统的功能，底层的设计结构图如图 4.4 所示；

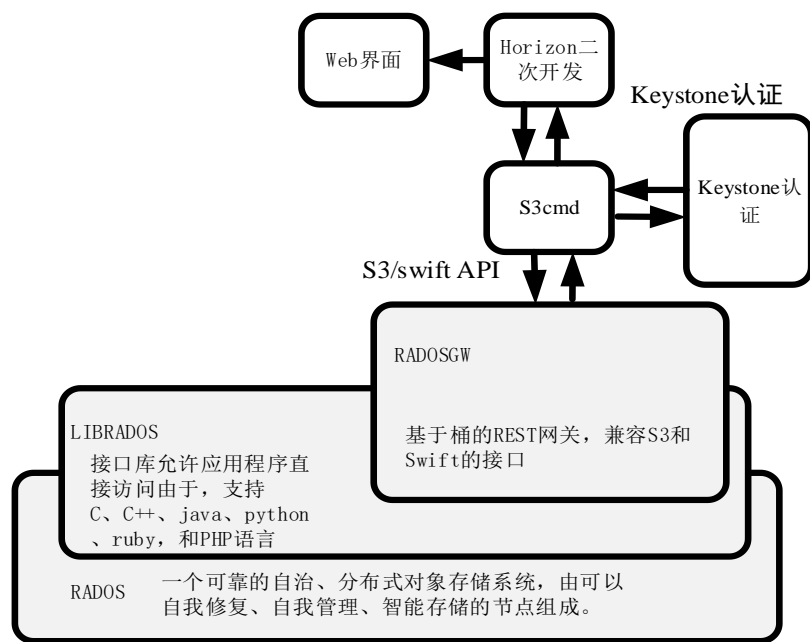


图 4.4 底层设计结构图

本文使用 S3 的接口来和 Radosgw 进行交互，使用 keystone 进行认证，通过 S3cmd 的工具将集群的信息返回到 Web 界面上。本文基于 Horizon 开发,将 Ceph 的信息返回到 Dashboard 上，用户可以直观的管理所存储的数据。

4.3 Ceph 和 OpenStack 结合设计

4.3.1 概述

因为 libvirt 配置了 librbd 的 QEMU 接口,通过它可以在 OpenStack 中使用 Ceph 块存储。Ceph 块存储是集群对象，这意味着它比独立的服务器有更好的性能。在 OpenStack 中使用 Ceph 块设备，必须首先安装 QEMU，libvirt 和 OpenStack，图 4.5 描述了 OpenStack 和 Ceph 技术层次结构：

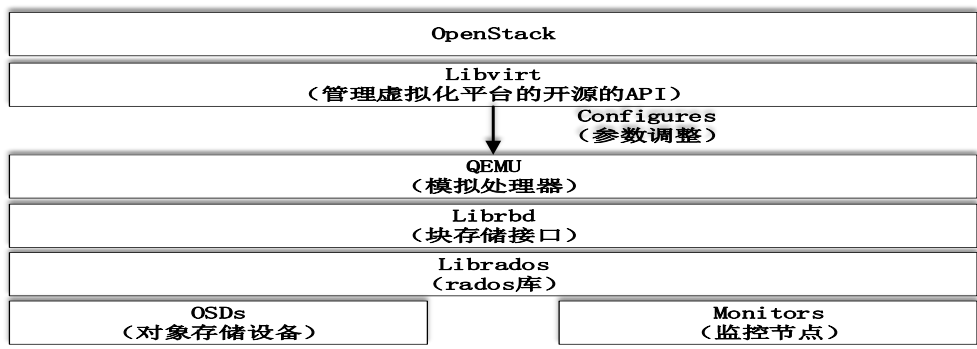


图 4.5 两者结合层次结构图

4.3.2 两者结合的系统架构设计

OpenStack 作为云操作系统，它的底层完全可以使用 Ceph 来提供存储，本文设计的系统就是基于 Ceph，对外提供对象存储服务，还对 OpenStack 的其他组件提供块存储；体现了 OpenStack 和 Ceph 的结合性；两者结合的系统架构如图 4.6 所示；

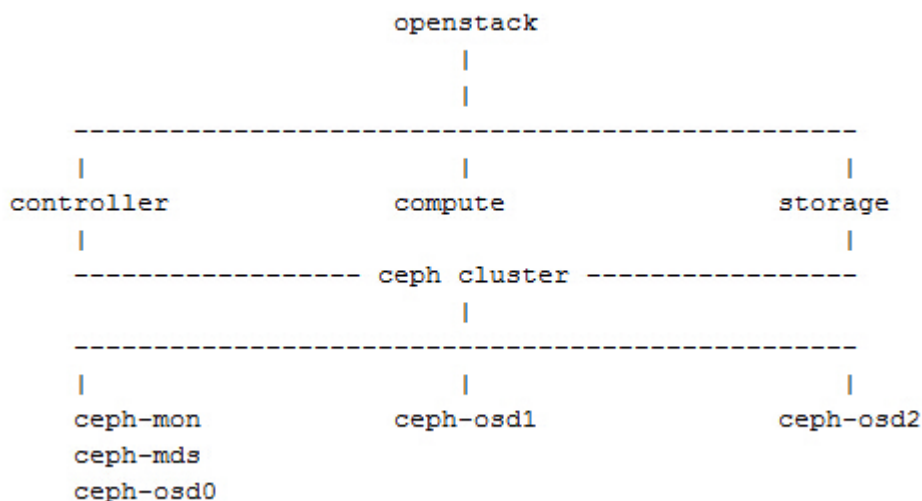


图 4.6 系统架构图

从图 4.6 中可以看出目前可以设置 Ceph 作为 Nova、Cinder 和 Glance 的存储后端。

4.4 系统管理模块的设计

4.4.1 集群状态展示设计

为了对系统当前的集群的所有状态进行实时的显示查看，因此本文设计了一个静态页面来供用户查看，在用户登录后，设计了个专用的按钮---云存储控制按钮。还可以看到目前 Ceph 集群的预览状态。Overview 页面展示对象存储的基本信息，展示出多少 OSD 还处于 up（或者 down）状态。在页面上设置了一个定时器来，当时间到零时，页面会自动刷新主机的存储和运行状态。展示流程图的基本设计如图 4.7 所示：

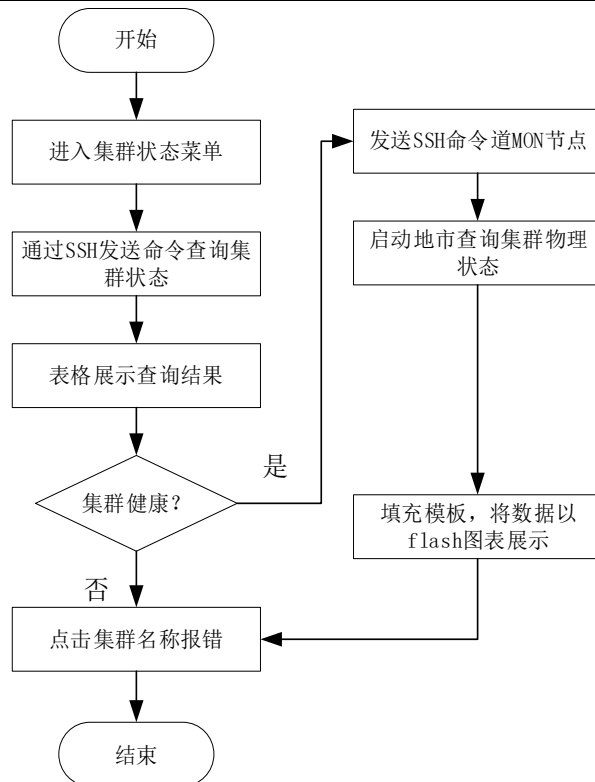


图 4.7 对象存储状态展示流程图

在管理集群的时候，一般常用 SSH 命令来进行控制，检测和控制 RGW 的一些状态。通信主要使用 json 格式，进行封装和解析，然后在前端界面上进行结果展示。

4.4.2 OSD 操作设计

对象存储核心就是 Mon 和 OSD，要想使得用户方便的使用对象存储功能，必须提供对 OSD 的操作，后期会继续添加功能，对 OSD 的状态进行操作，Up/Down 等。目前只做一个添加和删除 OSD 的功能：

主机添加/删除 OSD 操作流程：

- (1) 判断硬盘是否还有空余空间。
- (2) 是否有需要删除的 OSD。
- (3) 向集群 HOST 主机发送操作命令。
- (4) 执行相应的添加或者删除操作。

具体设计流程如图 4.8 所示：

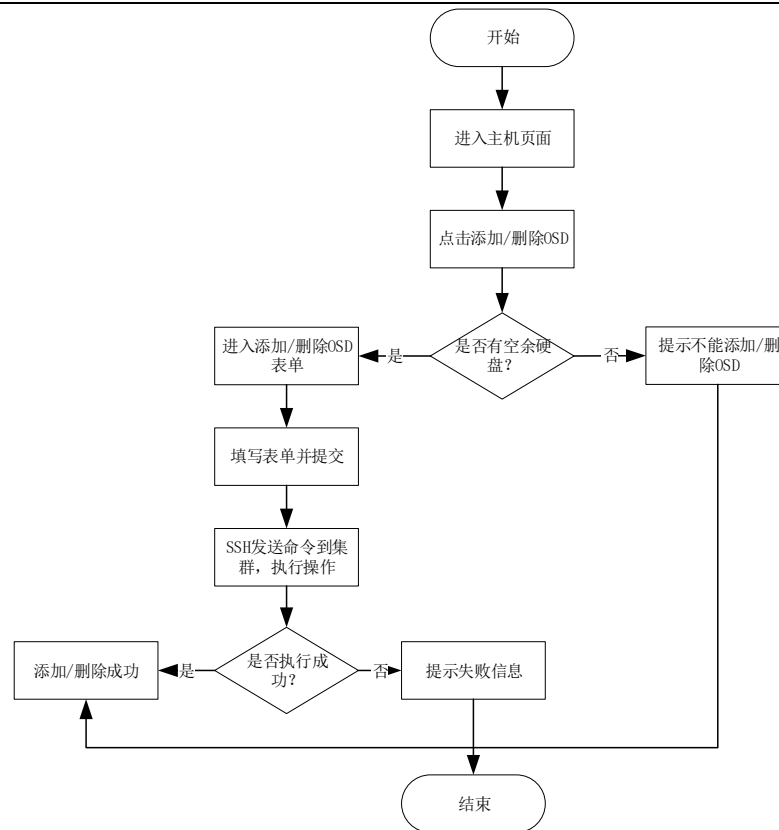


图 4.8 OSD 操作设计流程图

如图 4.8 所示，在设计 OSD 操作时，需要对磁盘进行判断，有空的磁盘才可以添加，若是删除 OSD 则不需要这部检查。

4.5 系统功能模块的设计

对象存储系统功能的设计思想是通过 Web 界面与用户进行一对一交互，在这个过程中，是通过 B/S 模式进行监控和管理系统的，因此本文选择的管理界面是以 Python Web 方式实现。MVC 方式进行架构，对象存储系统管理模块设计如下。

4.5.1 用户管理模块操作设计

这里所说的用户是指有登录云存储服务器并进行文件操作的权限的用户，本节在新建的池(pool)中，可以很清楚的看到几个默认的池，`.users.uid pool` 中都对应两个对象：`<user_name>` `<user_name>_buckets`，`rgw` 中 bucket 中的数据对象都是保存在 `.rgw.buckets` 中，其命名方式格式：`<bucket_id>_<key>`，`key` 是对象的名字，`bucket_id` (bucket marker) 对应 bucket 的元信息。可以根据 `rados` 自带的命令来查询这些被维护的数据。目前可以做到对已经注册的

用户信息进行展示。

展示用户列表流程图如图 4.9 所示：

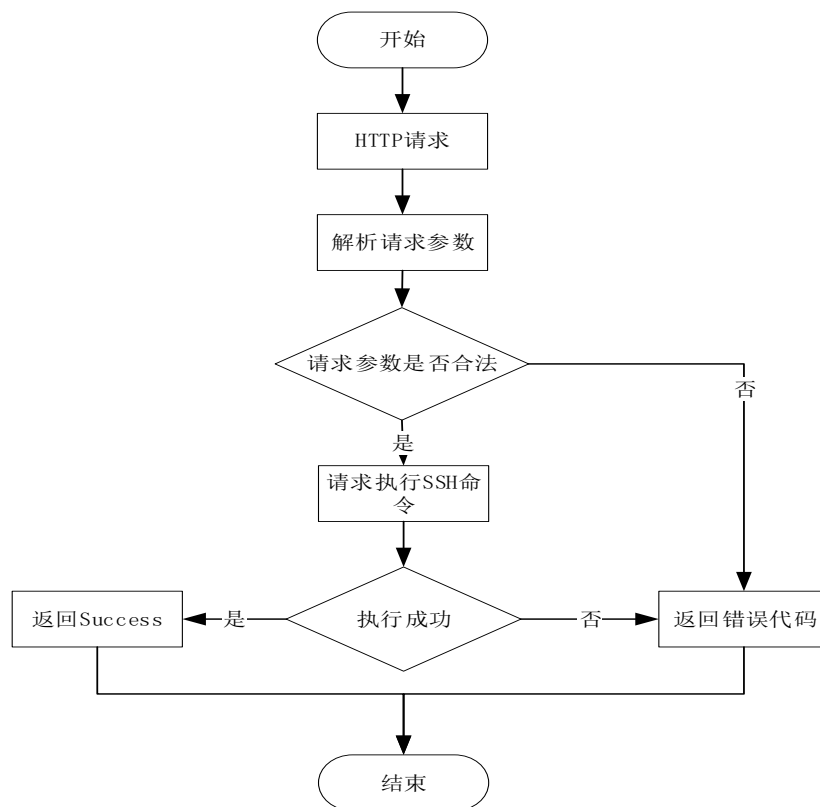


图 4.9 用户列表展示流程图

当 SSH 调用命令获取用户数据，以 json 的数据格式返回，目前对用户数据先不做处理。

4.5.2 桶 Buckets 的操作设计

对象存储中的桶可以看出，全局的命名空间是唯一的，当用户创建同名的 bucket 是无法成功的，只有在对象是属于某个 bucket 的时候，用户才被允许上传 object。因此桶也可以看作是用户的个人存储空间，创建桶和删除桶是最常见的操作。这里使用 S3 的接口，因为自带的接口太过繁琐。

4.5.2.1 创建 Buckets 功能设计

桶的创建流程如下：

- (1) 判断客户端输入的名称是否合法
- (2) 提交给用户
- (3) 通过 Amazon S3 的 API 来判断 Bucket 是否存在

(4) 判断用户是否有权限

必须完成上面的步骤,用户才可以新建一个 Bucket。它的设计逻辑流程图如图 4.10 所示:

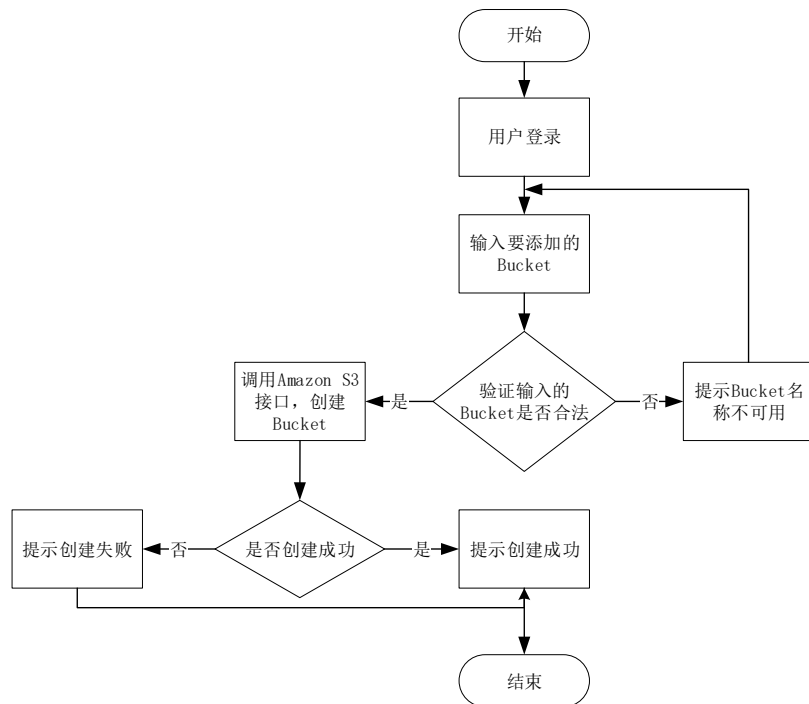


图 4.10 新建一个桶的流程图

设计新建桶操作时,对桶的名称也有限制,不仅仅需要唯一,长短也有限制。

4.5.3 Objects 的操作设计

4.5.3.1 上传 object 设计

Object 的基本设计思路就是把它当作云存储之中的基本单位,当用户将数据存储进去,在存储空间中的任何一个实体,无论是数据或者,都会被看成一个 Object,由 Key、data、以及 Metadata 组成的。对 Object 对象的操作由很多,上传下载,文件在线查看,重命名,移动,复制、删除等,本系统开发的功能主要就是 objects 的展示,以及上传和下载。上传 object 的设计流程如图 4.11 所示:

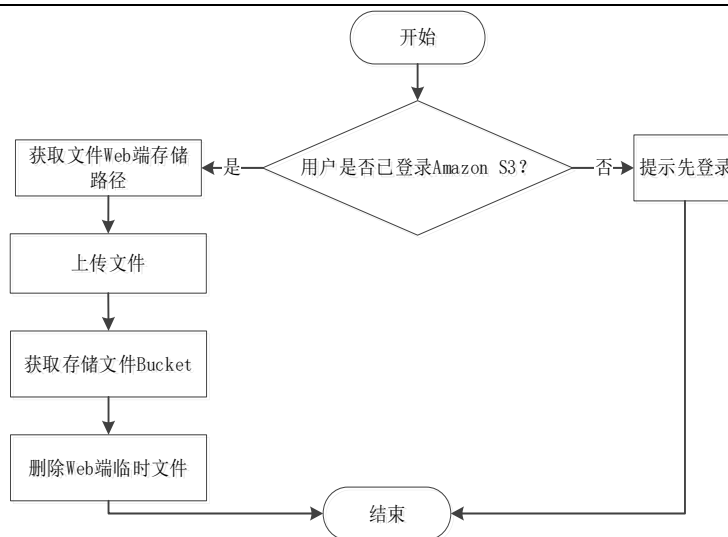


图 4.11 上传 object 对象设计图

4.5.3.2 查看 object 设计

查看 object 的设计和查看用户信息已经用户桶的流程是一致的；如图 4.12 是查看对象设计流程图：

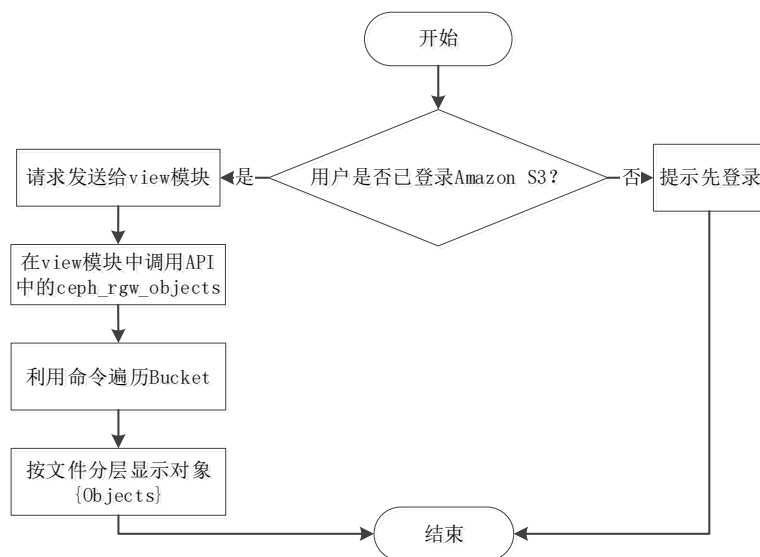


图 4.12 查看 object 对象设计图

4.5.3.2 下载 object 设计

下载 object 的设计，就是命令不同，逻辑还是一样的；它的设计逻辑如图 4.13 所示：

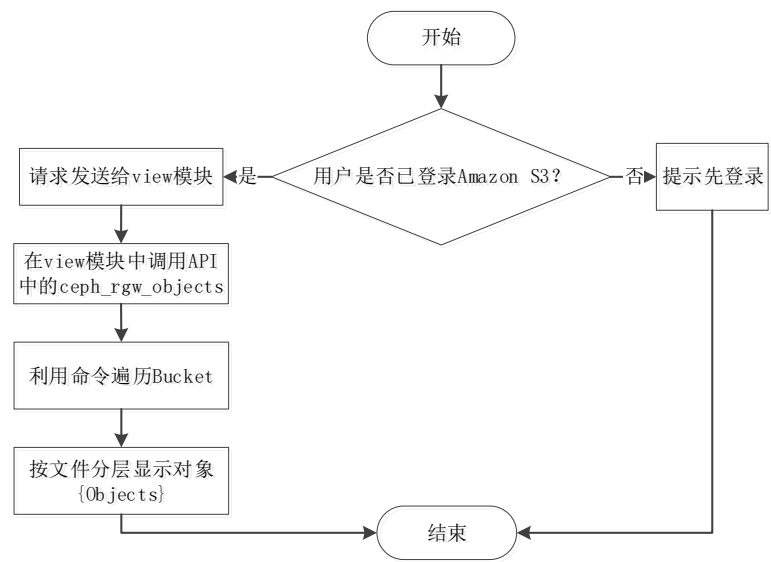


图 4.13 下载 object 对象设计图

本节对系统的功能做了设计，分别给出了设计流程图；前一节已经对系统的底层做了设计；下一节将详细阐述如何实现这些功能设计。

4.6 系统底层实现

4.6.1 底层实现结构图

如图 4.14 所示，底层实现需要三台 Centos 服务器、设置两个网络等，将在下一小节展开讨论底层的实现。

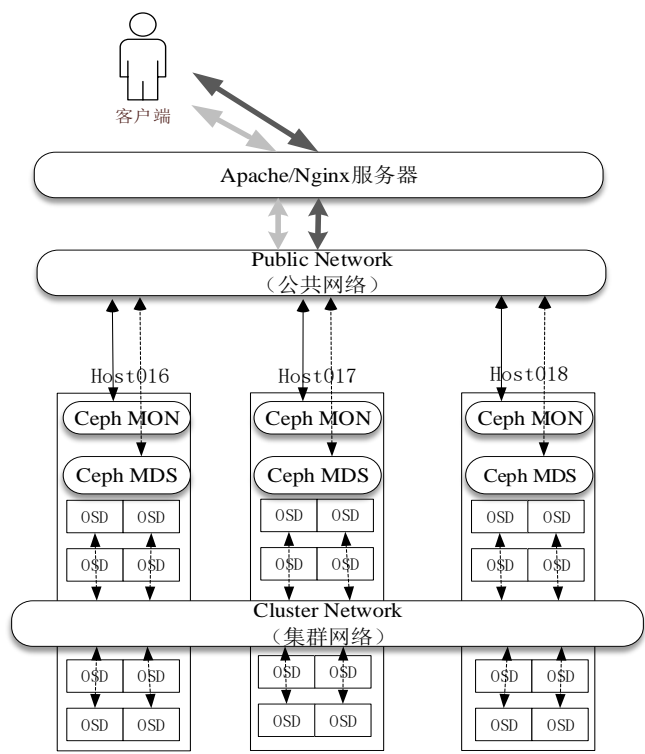


图 4.14 底层实现结构图

如图 4.14 所示就是底层的实现结构图；下一小节将根据这个结构图逐步实现底层环境搭建。

4.6.2 底层部署实现

本节从如下几个方面介绍系统底层的实现：

- 服务器环境实现：

选取公司三台服务器，每台服务器需要设置 IP 地址，分别设置为 192.222.1.16、192.222.1.17、192.222.1.18；Host016 需要使用机敏云产品；将 Host016 作为 Mon 的节点，Host017 和 Host018 的服务器作为 OSD 节点。出现如图 4.15 所示，表示机敏云部署成功：



图 4.15 机敏云部署成功图

- 网络设置：

三个服务器集群需要设置两个网络，一个作为公共网络，一个作为集群网络。目前的 OSD 数量还是很少的，可以将公共网络和集群网络设置成同一个；如图 4.16 所示，返回 True 则表示网络设置成功。

```
192.222.1.0/24
192.222.1.0/24
return: True
```

图 4.16 网络设置

- OSD 参数设置：

当 Monitor 节点安装成功之后，需要手动添加三个 osd，使用 ceph-disk 命令，并修改 ceph.conf 文件，可以先将副本设置成 1，以及添加最大名称限制，如图 4.17 所示：

```
[root@host005 ~]# ceph-disk prepare /var/lib/glance/osd1
[root@host005 ~]# ceph-disk prepare /var/lib/glance/osd2
[root@host005 ~]# ceph-disk prepare /var/lib/glance/osd3
```

```
● ceph-osd@1.service - Ceph object storage daemon
   Loaded: loaded (/usr/lib/systemd/system/ceph-osd@.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2017-11-08 11:44:50 CST; 8min ago
     Process: 26558 ExecStartPre=/usr/lib/ceph/ceph-osd-prestart.sh -
```

```
osd_pool_default_size = 1
auth_cluster_required = none
auth_service_required = none
auth_client_required = none
mon_osd_adjust_down_out_interval = false
mon_osd_adjust_heartbeat_grace = false
mon_osd_down_out_interval = 10800
osd_heartbeat_grace = 7
osd_heartbeat_interval = 2

osd_max_object_name_len = 256
osd_max_object_namespace_len = 64
osd_crush_chooseleaf_type = 0
```

图 4.17 OSD 参数设置图

- 节点权限设置:

每个节点行都需要创建用户，并赋予 root 权限。安装 ntp 及 openssl-server，以便主机节点直接访问 OSD 节点。然后修改主节点的 hosts 文件，将三个节点的 IP 地址和节点名称写入，如下图所示。配置 node1 主机节点上的 ssh，使用默认配置，并创建 config 文件，写入节点名称以及用户名。将 config 文件赋予 600 的权限。如图 4.18 所示：

```
[root@localhost ~]# vim /etc/hosts
```

```
192.222.1.16 node1
192.222.1.17 node2
192.222.1.18 node3
```

图 4.18 设置文件截图

- Monitor 初始化设置:

编辑 Ceph.conf 文件；Monitor 的设置必须保证三个节点之间正常通信，可以实现 OSD 的心跳检测；可以在 node1 上执行命令 Ceph-deploy install node1 node2 node3，安装完成之后，编辑 Ceph.conf 配置文件，然后执行 Monitor 的初始化。如图 4.19 所示：

```
[root@host005 ~]# storageinstaller-cli add_monitor host005 192.222.1.5
call method: add_monitor
host005
192.222.1.5
return: True
```

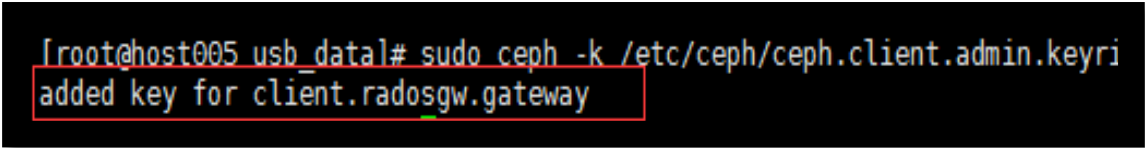
```
[root@localhost ~]# Ceph-deploy install node1 node2 node3
```

图 4.19 Monitor 设置部署图

- Ceph 网关设置:

将 Ceph-radosgw 和 Nginx 完成之后，创建 RGW 用户和 keyring，在服务器上创建 keyring，生成 Ceph-radosgw 服务对应的用户和 key，然后为用户添加访问权限导入 keyring 到集群中，

参数配置成功如图 4.20 所示：



```
[root@host005 usb_data]# sudo ceph -k /etc/ceph/ceph.client.admin.keyring
added key for client.radosgw.gateway
```

图 4.20 权限导入成功图

做完权限的导入，底层的环境就已经部署完成了。

4.7 Ceph 和 OpenStack 结合的实现

4.7.1 Ceph 作为 Cinder 后端的实现

在两者实现结合之前，已经完成了 Ceph 集群的实现，就是底层的实现；下面阐述如何将 Ceph 作为 Cinder 的后端，步骤如下：

- 创建存储池：

```
# ceph osd pool create volumes 128
# ceph osd pool create images 128
# ceph osd pool create vms 128
```

- 设置 Ceph 的客户端认证，在 OpenStack 节点执行如下命令：

```
# ceph auth get-or-create client.cinder mon 'allow r' osd 'allow class-read object_prefix
rbd_children, allow rwx pool=volumes, allow rwx pool=vms, allow rx pool=images'
# ceph auth get-or-create client.glance mon 'allow r' osd 'allow class-read object_prefix
rbd_children, allow rwx pool=images'
```

- 为 client.cinder、client.glance 添加密钥文件使节点改变属性。

```
# ceph auth get-or-create client.glance | ssh openstack sudo tee /etc/ceph/ceph.client.glance.keyring
# ssh openstack sudo chown glance:glance /etc/ceph/ceph.client.glance.keyring
# ceph auth get-or-create client.glance | ssh compute sudo tee /etc/ceph/ceph.client.glance.keyring
# ssh compute sudo chown nova:nova /etc/ceph/ceph.client.glance.keyring
# ceph auth get-or-create client.cinder | ssh compute sudo tee /etc/ceph/ceph.client.cinder.keyring
# ssh compute sudo chown nova:nova /etc/ceph/ceph.client.cinder.keyring
# ceph auth get-or-create client.cinder | ssh storage sudo tee /etc/ceph/ceph.client.cinder.keyring
# ssh storage sudo chown cinder:cinder /etc/ceph/ceph.client.cinder.keyring
```

运行 nova-compute 的节点 nova-compute 进程需要密钥文件。它们也存储 client.cinder 用户的密钥在 libvirt。libvirt 进程在 Cinder 中绑定块设备时需要用到它来访问集群。

- 在 nova-compute 节点创建一个临时的密钥副本：

```
# uuidgen
457eb676-33da-42ec-9a8c-9293d545c337
```

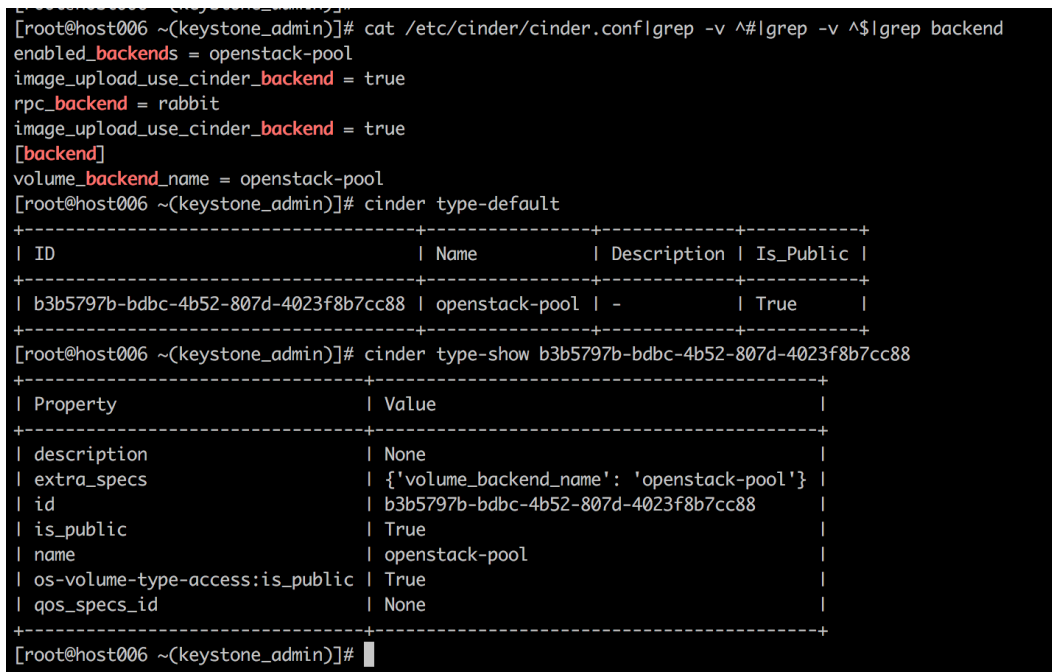
```
# cat > secret.xml <<EOF
<secret ephemeral='no' private='no'>
  <uuid>457eb676-33da-42ec-9a8c-9293d545c337</uuid>
  <usage type='ceph'>
    <name>client.cinder secret</name>
  </usage>
</secret>
EOF
# sudo virsh secret-define --file secret.xml
# sudo virsh secret-set-value --secret 457eb676-33da-42ec-9a8c-9293d545c337 --base64 $(cat
client.cinder.key) && rm client.cinder.key secret.xml
```

- 接下来更改 cinder 的配置文件：
- 在 openstack 节点和 storage 节点编辑 /etc/cinder/cinder.conf 配置文件并添加如下内容：

```
volume_driver = cinder.volume.drivers.rbd.RBDDriver
rbd_pool = volumes
rbd_ceph_conf = /etc/ceph/ceph.conf
rbd_flatten_volume_from_snapshot = false
rbd_max_clone_depth = 5
rbd_store_chunk_size = 4
rados_connect_timeout = -1
glance_api_version = 2
```

将 conf 文件保存，就已经完成了 Ceph 作为 Cinder 后端的参数配置。

可以用管道命令查看，如图 4.21 所示：



```
[root@host006 ~(keystone_admin)]# cat /etc/cinder/cinder.conf | grep -v ^# | grep -v ^$ | grep backend
enabled_backends = openstack-pool
image_upload_use_cinder_backend = true
rpc_backend = rabbit
image_upload_use_cinder_backend = true
[backend]
volume_backend_name = openstack-pool
[root@host006 ~(keystone_admin)]# cinder type-default
+-----+-----+-----+-----+
| ID | Name | Description | Is_Public |
+-----+-----+-----+-----+
| b3b5797b-bdbc-4b52-807d-4023f8b7cc88 | openstack-pool | - | True |
+-----+-----+-----+-----+
[root@host006 ~(keystone_admin)]# cinder type-show b3b5797b-bdbc-4b52-807d-4023f8b7cc88
+-----+-----+
| Property | Value |
+-----+-----+
| description | None |
| extra_specs | {'volume_backend_name': 'openstack-pool'} |
| id | b3b5797b-bdbc-4b52-807d-4023f8b7cc88 |
| is_public | True |
| name | openstack-pool |
| os-volume-type-access:is_public | True |
| qos_specs_id | None |
+-----+-----+
```

图 4.21 Cinder 后端测试图

由图 4.21 所示，此时已经实现了 Ceph 作为 Cinder 的存储后端。

4.7.2 Ceph 作为 Glance 后端的实现

将 Ceph 作为 Glance 后端的配置必须修改 glance 接口的配置文件，

- 编辑 /etc/glance/glance-api.conf 并添加如下内容：

```
[DEFAULT]
default_store = rbd
...
[glance_store]
stores = rbd
rbd_store_pool = images
rbd_store_user = glance
rbd_store_ceph_conf = /etc/ceph/ceph.conf
rbd_store_chunk_size = 8
```

- 如果需要精心的写复制功能，需要添加如下代码：

```
show_image_direct_url = True
```

此时已经实现了 Ceph 作为 Glance 的存储后端。

4.8 系统管理模块的实现

首先要知道 RGW 业务处理流程如下：

- http request --> apache 转 Fastcgi module（处理线程的模块），
- FastCgi module --> radosgw 通过 socket 请求实现，
- radosgw --> Ceph 集群 通过 socket 实现，调用 rados 接口。

可以把 Ceph RGW 简单理解成 Ceph 集群的一个客户端，用户通过它间接的访问 Ceph 集群，因此本文先在实验环境的 16 服务器上部署一个 RGW 节点。对象存储管理系统主要监控 Ceph 集群的运行状况，这里主要实现 OSD 的监控，从 mon 上面获取数据，在页面上展示，也可以根据用户的需要进行对象存储的管理，不论是添加 OSD 还是删除 OSD 都是很直接的操作，Ceph 本身是一个单独完整的对象存储，但是在和 OpenStack 对接后，可以提供比单独使用 Ceph 更全面更完整的平台管理。OpenStack 可以提供 keystone 的认证，本文所做的就是基于 RGW 对象存储对 horizon 进行二次开发。让机敏客户直观的使用对象存储，并进行管理。

CephRGW 对象存储管理系统有集群预览，实时图表，存储控制，OSD 控制，磁盘控制等性能状况。管理系统采用 Django 技术开发。开发模式是 MVC 模式。系统与底层 CephRGW 模块交互，采用接口层提供的 restful 交互，或者是 Python 与 Amazon S3 的命令操作交互。

4.8.1 集群状态展示功能的实现

集群状态的展示主要是通过客户端去执行 Ceph 的命令，查看集群与主机的运行状态以及各项相关的参数指标，并用正则表达式解析 Ceph 命令的执行结果，并将结果展示在 dashboard 上。为了实现实时的状态监控检测，已经设置了一个定时器，定时查看集群状态。系统采用图形化数据的方式进行了更直观的效果展示。具体实现步骤如下：

1. 对外界面上点击相应的请求均发至 api 目录中的对应 py 文件处理，在 api 中，对外暴露接口是 api/__init__.py 文件，根据 python 语言特性，用 __init__ 控制可见性，如下图所示，在 api/__init__.py 中添加 ceph 后端入口组件。如图 4.22 所示：

```
47 from openstack_dashboard.api import ceph
48 from openstack_dashboard.api import storageinstaller
49 from openstack_dashboard.api import idv
```

图 4.22 导入模块图

2. 在 api/ 中添加 ceph.py，主要用于的与 ceph 接口后端对接如图 4.23 所示：

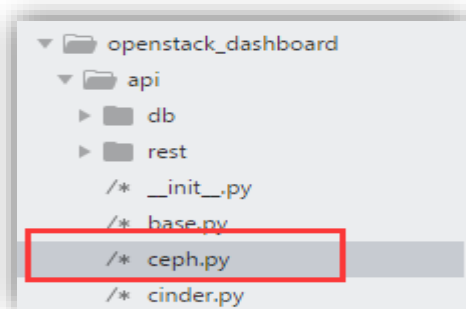


图 4.23 后端接口文件图

3. 要在 dashboards 视图添加 panel 时，在 dashboards/ceph/dashboard.py 中添加 panel 名称并注册，查看 admin/dashboard.py 文件，添加 ceph 的 panel，如图 4.24 所示：

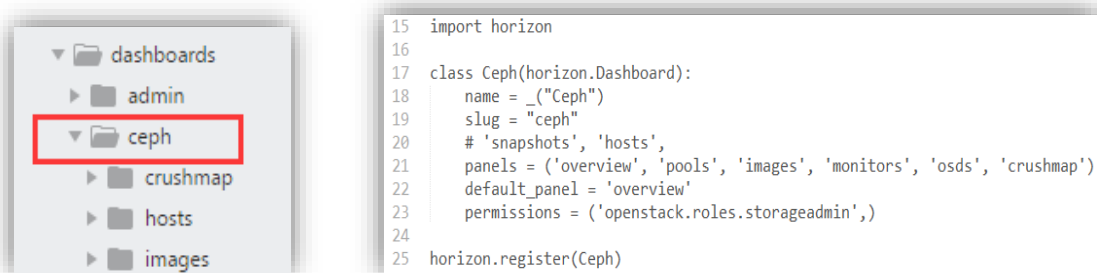


图 4.24 添加 ceph 的 panel 图

4. 之后在 dashboard 下建立 ceph 目录，UI 上各个 panel 之间像一个个插件，作为一个 Django 项目，实现时可以借鉴/拷贝其他 panel 的使用方式，可以发现目录结构和其他模块一致。
5. 其中 admin/ceph/panel.py 控制 panel 的显示，pool 的显示函数如图 4.25 所示：

```
def ceph_pool_status(request):
    value = ''
    try:
        value = api.ceph.get_ceph_pool_status(request)
    except:
        # msg = _('Unable to get pool status.')
        # messages.warning(request,msg)
        pass
    # response ceph_pool_status
    return JsonResponse(value, safe=False)
```

图 4.25 Pool 状态展示代码图

6. 对应的 url 为 horizon/dashboard /ceph /，调用 urls.py，接口如图 4.26 所示：

```
urlpatterns = patterns(
    VIEW_MOD,
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^ceph_health_status/$', views.ceph_health_status, name='ceph_health_status'),
    url(r'^ceph_warning_num/$', views.ceph_warning_num, name='ceph_warning_num'),
    url(r'^ceph_disk_status/$', views.ceph_disk_status, name='ceph_disk_status'),
    url(r'^ceph_pg_status/$', views.ceph_pg_status, name='ceph_pg_status'),
    url(r'^ceph_pool_status/$', views.ceph_pool_status, name='ceph_pool_status'),
    url(r'^ceph_host_status/$', views.ceph_host_status, name='ceph_host_status'),
    url(r'^ceph_mon_status/$', views.ceph_mon_status, name='ceph_mon_status'),
    url(r'^ceph_osd_status/$', views.ceph_osd_status, name='ceph_osd_status'),
    url(r'^ceph_mds_status/$', views.ceph_mds_status, name='ceph_mds_status'),
    url(r'^ceph_iops/$', views.ceph_iops, name='ceph_iops'),
)
```

图 4.26 函数接口定义图

8. 在 html 页面，panel 页面显示的数据包括，一段 js 将 stats 数据图形化显示，使用“tab_group.render”显示定义的 tab。

至此，集群展示功能的开发已经完成了。

4.8.2 OSD 添加/删除功能的实现

4.8.2.1 添加 OSD 功能实现

添加 OSD 的逻辑流程如下

在界面按钮上获取创建 OSD 请求，输入主机名，后端判断是否有新的盘，然后默认创建一个 down 的 osd，在根据封装的命令将 osd 激活；加入 crushmap 中。前面的步骤和上一小节一样；简要介绍下面的实现步骤：

- 1、在 dashboard/ceph/ 添加 osd 的 panel；在 dashboards 视图中添加 panel 时，在 dashboards/ceph/dashboard.py 中添加 panel 名称并注册；
- 2、编写添加 OSD 的表单函数，并调用 api 文件下的函数实现用户请求的提交；函数如图 4.27 所示：

```
class AddOsdForm(forms.SelfHandlingForm):
    host = forms.CharField(label=_("Host"),
                           max_length=8,
                           help_text=_("Name of the new osd."),
                           required=True)

    def handle(self, request, data):
        host = data['host']
        try:
            api.storageinstaller.add_osd(host)
        except:
            redirect = reverse('horizon:ceph:osds:index')
            messages.error(request, _("Failed to add osd %(host)s!") % dict(host=host))
            exceptions.handle(request, redirect=redirect)
            return False

        return True
```

图 4.27 添加 osd 表单函数图

- 3、前端发送添加 OSD 请求时，后端检测是否有空盘，并返回信息：
- 4、在 osd 的 panel 中编写 AddOsd 函数接口；并配置其 url；接口如图 4.28 所示：

```
class AddOsd(tables.LinkAction):
    name = "add_osd"
    verbose_name = _("Add Osd")
    url = "horizon:ceph:osds:add_osd"
    classes = ("ajax-modal", "btn-primary", "btn-launch",)
    icon = "plus"
```

图 4.28 函数接口图

- 5、在 osd/view.py 中定义函数实体。如图 4.29 所示：

```
def AddOsd(request, OSD_name):
    osd_info = request.POST
    host = Osd_name
    if mon_info:
        LOG.info("!---->publicIp:%s" % osd_info['publicIp'])
    try:
        res = api.storageinstaller.add_osd(host, mon_info['publicIp'])
        LOG.info("%s" % res)
    except:
        messages.error(request,
            _("Failed to add osd %s !") % dict(host=osd_info['publicIp']))

    next_url = reverse("horizon:ceph:osds:index")
    return http.HttpResponseRedirect(next_url)
```

图 4.29 函数定义图

- 6、编写判断 disk 的状态，如果还有空的磁盘就可以提示创建 osd，否则提示不能创建新的 osd；具体函数实现如图 4.30 所示：

```
def get_ceph_disk_status(request):
    client = RADOSClient(cluster_name)
    client.connect()
    outbuf = client.execute(prefix='pg dump', argdict={'format': 'json', 'dumpcontents': ['sum']})
    status = json.loads(outbuf)
    full_ratio = status['full_ratio']
    near_full_ratio = status['near_full_ratio']
    capacity = status['osd_stats_sum']
    total = capacity['kb']
    kb_used = capacity['kb_used']

    return {'used':str(kb_used/1024/1024), 'total':str(total/1024/1024), 'near_full':str(
        near_full_ratio), 'full':str(full_ratio)}
```

图 4.30 磁盘检测函数图

- 7、至此已经创建一个 down 的 osd，在根据封装的命令将 osd 激活；加入 crushmap 中。如图 4.31 所示：

```
def ActiveOSD(self):
    kill = lambda process: process.kill()
    cmd = ['ceph-disk', 'activate', 's%', '--format=json', '--conf='+self.conf]
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    my_timer = Timer(3, kill, [proc])
    stdout = ""
    try:
        my_timer.start()
        stdout, stderr = proc.communicate()
    finally:
        my_timer.cancel()
    if not stdout:
        raise
```

图 4.31 函数具体实现图

此时对 OSD 的添加功能就已经开发完成了。

4.8.2.2 删除 OSD 功能实现

删除 osd 的功能需要几条命令在集群执行；主要逻辑是：在 api 文件夹下的 ceph.py 中定义一个 delete_osd 函数，该函数接受一个 osd 的 id 号，ID 号是通过界面的复选框，传递给 filteraction 类获得，在 osd 目录下的 table 的这个类中是定义表格的类，就是前段表格的定义，然后添加 DeleteOsd 类。下面给出主要的实现步骤：

1. 在 ceph/osds/table 中添加删除 osd 函数的 url，如图 4.32 所示：

```
url = "horizon:ceph:osds:delete_osd"
classes = ("ajax-modal","", "btn-danger",)
icon = "minus"
```

图 4.32 url 定义图

2. 在 api/ 中的 ceph.py 里编写 delete_osd 函数，
3. 在 dashboard 下建立的 ceph 目录中，UI 上各个 panel 之间像一个个插件，作为一个 Django 项目，实现时可以模仿添加 osd 功能的 panel 使用方式，可以发现两者目录结构和模块基本一致。
4. 编写 delete_osd 接口，定义 url 以及函数名称，实现代码如图 4.33 所示：

```
class DeleteOsd(tables.LinkAction):
    name = "delete_osd"
    verbose_name = ("Delete OSD")
    url = "horizon:ceph:osds:delete_osd"
    classes = ("ajax-modal","", "btn-danger",)
    icon = "minus"
```

图 4.33 函数定义图

5. 将一个前端复选框选中的 osd 删除需要四个步骤，也就是说在集群的 Monitor 节点上需要执行四次操作才能完成删除操作。
6. 前端发来请求表单有 osd 的 id 好，系统需要将其 out 掉
代码 `ceph osd out osd.id`，这个才操作就是从集群中踢走这个 osd，
7. 当 osd 从集群中被提出的时候，它依然占有独立的线程，此时需要将这个线程停止；就是 stop 掉这个 osd 的进程，实现代码如下：`systemctl stop ceph-osd@osd.id`

8. 在 `osd` 的线程被停止掉之后, 将该 `osd` 从 `osd tree` 中移除, 实现代码: `ceph osd crush remove osd.osd.id` (`osd.id` 指的是 `osd` 的 `id` 号), 此时该 `osd` 已经不在 `osd tree` 中了, 接着需要执行 `ceph auth del osd.osd.id` 和 `ceph osd rm osd.osd.id`, 这时候 `osd` 才删除成功, 但是原来的数据和日志目录还在, 也就是 `osd` 存储的数据还在。
9. 此时系统将该 `osd` 所在的磁盘 `umount` (解挂载), 然后将磁盘进行擦除那么数据就会被完全删除了, 执行 `umount /osd.disk` (指的是 `osd` 所在的磁盘目录), 然后执行 `ceph-disk zap /osd.disk`。
10. 实现代码部分如图 4.34 所示:

```
def delete_osd(self):
    kill = lambda process: process.kill()
    cmd1 = ['ceph', 'osd', 'out', 's%', '--format=json', '--conf='+self.conf]
    cmd2 = ['systemctl', 'stop', 'ceph-osd@s%', '--format=json', '--conf='+self.conf]
    cmd3 = ['ceph', 'osd', 'crush', 'remove', 'osd.s%', '--format=json', '--conf='+self.conf]
    cmd4 = ['ceph', 'auth', 'del', 'osd.s%', '--format=json', '--conf='+self.conf]
    cmd5 = ['ceph', 'osd', 'rm', 's%', '--format=json', '--conf='+self.conf]
    proc = subprocess.Popen(cmd1, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    proc = subprocess.Popen(cmd2, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    proc = subprocess.Popen(cmd3, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    proc = subprocess.Popen(cmd4, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    proc = subprocess.Popen(cmd5, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    my_timer = Timer(3, kill, [proc])
    stdout = ""
    try:
        my_timer.start()
        stdout, stderr = proc.communicate()
    finally:
        my_timer.cancel()
    if not stdout:
        raise
```

图 4.34 删除 OSD 函数主要代码

详细的代码见附录, 在执行完这些步骤后, 前端请求删除的 `osd` 就被完全删除了。这里简单实现了删除 `osd` 的功能, 对 `osd` 的状态并没有检查, 操作具有一定的危险性。

4.9 系统功能模块的实现

4.9.1 用户信息管理模块实现

当 `radosgw` 安装完成之后, 系统会默认生成几个存储用户信息的 `pool`, 可以通过开发接口, 调用 `rados` 命令来实现用户信息的展示, 目前还没完成用户创建和删除功能的开发, 只是完成了用户信息的展示, 每个用户在 `.users.uid pool` 中都对应两个对象: `< user_name > < user_name >_buckets`, 可以通过命令 `rados -p .users.uid ls` 来查看用户, 以及 `.users` 中分别为用户的 `s3` 账号的 `access_key` 创建一个对象, 对象名是用 `access_key`, 可以查看 `.users` 来获取

用户的信息。

接着就在 Ceph.py 中定义函数(函数名),在 view.py 中定义函数的接口,这里使用的 MVC 模式,而后将命令查询到的信息以 json 格式返回到页面上主要代码及初期抓取用户数据截图如图 5.22 所示。

下面给出实现步骤:

- 1、 在 dashboard 文件夹下创建 RGWUser 文件夹,它们的开发模式和相同,在 RGWUser 文件下创建 template 和 RGWUser 文件夹,并创建 index.html 文件。
- 2、 在前端的 index.html 中定义 Users 变量;用来接收后端返回的数据;
- 3、 在 ceph/RGWUsers/panel.py 中定义 url,就是给函数实体的路径。
- 4、 要在 dashboards 视图添加 panel 时,在 dashboards/ceph/dashboard.py 中添加 panel 名称并注册,查看 ceph/dashboard.py 文件,添加 RGWUsers 的 panel
- 5、 根据创建的 panel,定义接口;如图 4.35 所示:

```
url(r'^ceph_mds_status/$', views.ceph_rgw_users, name='ceph_rgw_users'),
url(r'^ceph_mds_status/$', views.ceph_rgw_buckets, name='ceph_rgw_buckets'),
# url(r'^ceph_mds_status/$', views.ceph_rgw_objects, name='ceph_rgw_objects'),

url(r'^ceph_iops/$', views.ceph_iops, name='ceph_iops'),
```

图 4.35 函数 url 配置图

- 6、 根据前端请求的信息,封装命令查询当前授权用户,就是使用 S3 的命令行来执行操作查询当前的授权用户,并返回到前端 index 页面。主要代码实现如图 4.36 所示:

```
def ceph_rgw_users(request):
    value = ''
    try:
        value = api.ceph.get_cephrgw_user(request)
        # LOG.info("ceph_health_status = %s" % value)
    except:
        # msg = _('Unable to get cluster health.')
        # messages.warning(request,msg)
        pass
    # response health_status
    # LOG.info("ceph_health_status = %s" % value)
    return JsonResponse(value, safe=False)
```

图 4.36 接口定义图

- 7、 此时可以获得集群后端的用户数据，查到的数据都是以如下的格式返回数据：< user_name > < user_name >_buckets，用户数据草图如图 4.37 所示：



图 4.37 用户数据草图

此时，用户的数据以及能够被查询处理，剩余的工作就是 json 数据的解析和展示。

4.9.2 Bucket 容器管理模块实现

Bucket 是对象存储中的一个非常重要的概念。可以把 Bucket 理解成为一个具有足够的存储空间的硬盘，每个 Object 存放到对应的 Bucket 当中。因此，实现对象存储的核心工作是对 Bucket 的操作。

本文所述的系统进行的主要操作有：对所有 Bucket 进行展示，桶是被访问的资源，每个访问资源都具有对应的列表；实现步骤和查看用户信息一致，这里不再赘述；只是查询命令有所不同，通过 rados 的命令来查看当前用户创建的 Buckets；如图 4.38 所示：

```
[root@host016 ~]# rados -p default.rgw.users.uid listomapkeys foo.buckets
jimin-test
zhoulin
[root@host016 ~]#
```

图 4.38 rados 命令查看结果图

如图 4.38 就是通过命令来查询用户所建通的信息，接下来，就需要将命令封装到函数中去，并写查询接口供 Index 页面调用，并将查询到的数据以 json 格式返回，用正则表达式解析并展示出来。

创建桶要根据用户使用的 SDK 构建的创建请求进行创建。如果该用户已登录，则直接调用 Amazon S3 的 createBucket。示例代码如图 4.39 所示：



图 4.39 主要代码和效果图

用户已经登录，直接调用接口便可以查看所有的桶信息。主要代码如图 4.40 所示：

```
def ceph_rgw_getBuckets(self):
    kill = lambda process: process.kill()
    cmd = ['s3cmd', 'ls', '--format=json', '--conf='+self.conf]
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    my_timer = Timer(3, kill, [proc])
    stdout = ""
    try:
        my_timer.start()
        stdout, stderr = proc.communicate()
    finally:
        my_timer.cancel()
    if not stdout:
        raise
```

图 4.40 获取桶代码图

如图 4.40 所示，函数主要代码是 `cmd` 对 S3 命令的封装。

4.9.3 Object 管理模块开发与实现

4.9.3.1 上传 object 实现

功能描述：object 的上传，操作者必须拥有 WRITE 权限。

使用 Amazon S3 构建请求的过程如下：

- (a) 以 `AccessKey` 和 `SecretKey` 作为创建 `Amazon S3Client` 实例的参数进行实例创建
- (b) 创建准备上传的 `File` 文件对象
- (c) 调用 `AmazonS3Client` 的 `putObject` 方法进行对象上传。

本节给出实现该功能的实现示意如图 4.41：

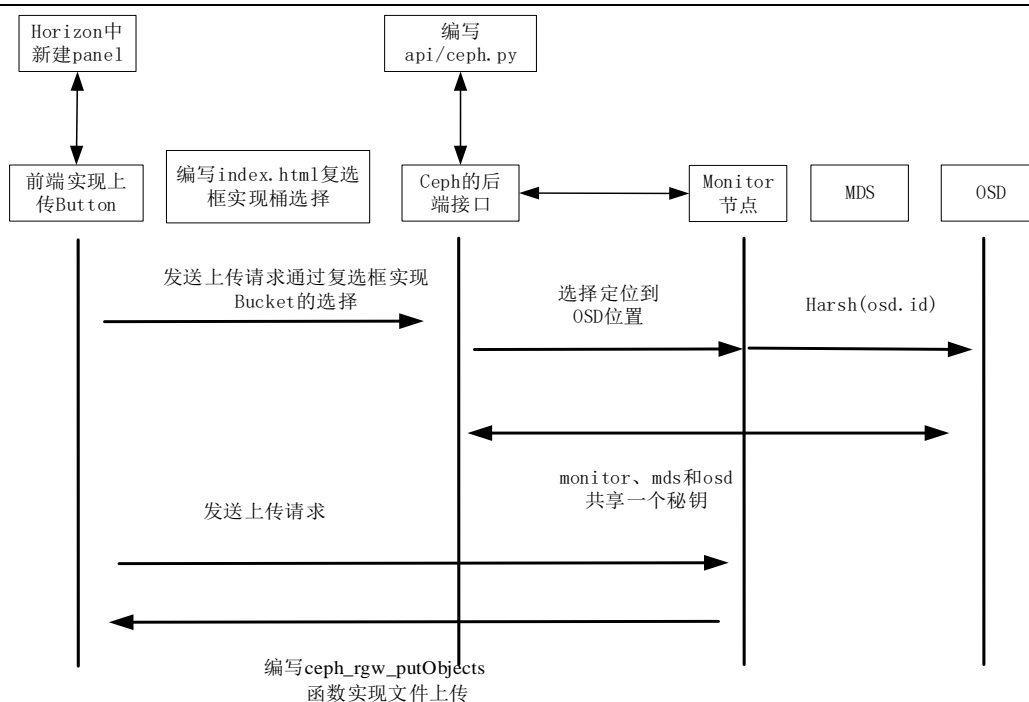


图 4.41 上传 Object 实现图

实现代码如图 4.42 所示：

```

def ceph_rgw_putObjects(self):
    kill = lambda process: process.kill()
    cmd = ['s3cmd', 'put', 's3//s%', '--format=json', '--conf='+self.conf]
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    my_timer = Timer(3, kill, [proc])
    stdout = ""
    try:
        my_timer.start()
        stdout, stderr = proc.communicate()
    finally:
        my_timer.cancel()
    if not stdout:
        raise

```

图 4.42 上传 object 函数代码图

至此已经完成了上传 object 功能的开发，其主要开发步骤就是撰写上传函数，以及前端表单实现对 Bucket 的选择。

4.9.3.2 查看 object 实现

要实现查看 object，就是根据给定的 Bucket 对该桶里的对象进行查看，操作者必须拥有对容器的 READ 权限。和查看用户信息设置方式一致，

使用 Amazon S3 构建请求的过程如下：

- (a) 以 AccessKey 和 SecretKey 作为参数进行 Amazon S3Client 实例的创建
- (b) 以容器名字作为参数并调用 Amazon S3Client 的 listObjects 方法，查看容器中所

有对象。

(c) 使用 `rados` 命令展示桶中的 `objects` 信息，具体内容如图 4.43 所示：

```
[root@host016 ~]# rados -p default.rgw.buckets.data ls
296893f4-7e57-4a51-9517-e802790f3790.64373.1_Word1-test.txt
296893f4-7e57-4a51-9517-e802790f3790.64373.1_Word2-test.txt
296893f4-7e57-4a51-9517-e802790f3790.64373.1_Word3-test.txt
296893f4-7e57-4a51-9517-e802790f3790.64373.1_Word-test.txt
296893f4-7e57-4a51-9517-e802790f3790.64373.1_Word0-test.txt
296893f4-7e57-4a51-9517-e802790f3790.64373.1_s3cmd-1.6.1.tar.gz
```

图 4.43 rados 查看 objects 的信息

下面给出实现步骤：

1. 在 `dashboard` 文件夹下创建 `RGWUser` 文件夹，它们的开发模式和相同，在 `RGWObjects` 文件下创建 `template` 和 `RGWObjects` 文件夹，并创建 `index.html` 文件。
2. 在前端的 `index.html` 中定义 `Objects` 变量：用来接收后端返回的数据；
3. 在 `ceph/RGWObjects/panel.py` 中定义 `url`，就是给函数实体的路径。
4. 要在 `dashboards` 视图中添加 `panel` 时，在 `dashboards/ceph/dashboard.py` 中添加 `panel` 名称并注册，查看 `ceph/dashboard.py` 文件，添加 `RGWObjects` 的 `panel`
5. 根据创建的 `panel`，定义接口；如图 4.44 所示：

```
url(r'^ceph_mds_status/$', views.ceph_rgw_users, name='ceph_rgw_users'),
url(r'^ceph_mds_status/$', views.ceph_rgw_buckets, name='ceph_rgw_buckets'),
# url(r'^ceph_mds_status/$', views.ceph_rgw_objects, name='ceph_rgw_objects'),

url(r'^ceph_iops/$', views.ceph_iops, name='ceph_iops'),
```

图 4.44 函数接口定义图

6. 根据前端请求的信息，封装命令查询 `Object`，就是使用 `S3` 的命令行来执行操作查询当前的授权用户，并返回到前端 `index` 页面。主要代码实现如图 4.45 所示：

```
def ceph_rgw_listObjects(self):
    kill = lambda process: process.kill()
    cmd = ['s3cmd', 'ls', 's3: //s%/' , '--format=json', '--conf='+self.conf]
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    my_timer = Timer(3, kill, [proc])
    stdout = ""
    try:
        my_timer.start()
        stdout, stderr = proc.communicate()
    finally:
        my_timer.cancel()
    if not stdout:
        raise
```

图 4.45 object 展示函数实现图

至此，对 Object 的查询功能开发完成。

4.9.3.3 下载 object 实现

通过指定的 Bucket 名称和 Object 名称下载一个 Object，操作者必须拥有 READ 的权限。对 Object 的下载功能的开发逻辑和上节一样，依然是新建 panel，并将该 panel 注册；在 index.html 中定义 listObj 变量，来接受后端返回的 json 数据；

使用 Amazon S3 构建请求的过程如下：

- (1) 以 AccessKey 和 SecretKey 作为参数创建 Amazon S3Client 实例
- (2) 以 Bucket 名、对象名称、保持的地址作为参数并调用 AmazonS3Clients 的 getObject 方法，下载指定对象。

现在和上传的功能很相似，都是需要定位到某个 Bucket，这里过程不再赘述，主要是实现的代码不同；Python 实现的关键代码（Ceph.py 中 Ceph_rgw_getObject 函数封装代码的截图）如图 4.46 所示：

```
def ceph_rgw_getObjects(self):
    kill = lambda process: process.kill()
    cmd = ['s3cmd', 'get', 's3://s%/s%.html', '--format=json', '--conf='+self.conf]
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    my_timer = Timer(3, kill, [proc])
    stdout = ""
    try:
        my_timer.start()
        stdout, stderr = proc.communicate()
    finally:
        my_timer.cancel()
    if not stdout:
        raise
```

图 4.46 获取 Objects 的主要代码

4.10 对磁盘利用率的改进

4.10.1 Ceph 权重简介

在 ceph 中各 osd 的 pg 数量是近似均匀的，可以认为各 pg 下的数据容量大致相等，因此从原理上来说保证各 osd pg 相等，则各 osd 上的磁盘使用量也差不多相同，但是由于算法做不到绝对均匀的原因某些 osd 上的 pg 数会相差比较大，这样某些 osd 上的空间使用量就会比较多。

权重就是影响 osd 存放 pg 的数目，给定一个脚本来使得 osd 自动设置权重，可以有效的使磁盘利用率变大。

4.10.2 测试权重

在 ceph 部署完成，各 pool 也创建完成后，可以通过命令调整 osd 的权重来调整 osd 上的 pg 数改变磁盘利用率

手动调整的权重的过程：

1.通过命令 `ceph -s` 或者 `ceph health` 检查 ceph 状态，有 osd near full

```
cluster bef6d01c-631b-4355-94fe-77d4eb1a6322
```

```
health HEALTH_WARN
```

```
4 near full osd(s)
```

2.ceph health detail 查看具体 near full 的 osd

```
osd.137 is near full at 86%
```

```
osd.169 is near full at 85%
```

```
osd.183 is near full at 86%
```

```
osd.199 is near full at 85%
```

3.通过 `ceph osd tree |grep osd.137` 查看具体的 osd 的 weight 值

```
137 1.63539 osd.137 up 1.00000 1.00000
```

```
169 1.63538 osd.169 up 0.89999 1.00000
```

4.调整 osd 的权重 `ceph osd crush reweight osd.134 1.5` 单盘 1.8TB，系统容量为 1.635，权重为 1.63539

$1.635 \times 0.86 = 1.406\text{TB}$ 那么从 1.63539 假设调到 1.5 那么 $1.406 \times 1.5 / 1.63539 = 1.29\text{TB}$ ， $1.29 / 1.635 = 0.789$

```
137 1.50000 1.00000 1674G 1379G 294G 82.40 1.13 258
```

5.ceph osd df|grep osd.x 查看 reweight 后的 osd 使用率

6.ceph -s 确认 near full 的 osd warning 消除

这是手工每次调整 osd 的权重步骤，下面给 osd 权重设置撰写一个脚本，使得 osd 能够自动合理的分配权重，提高磁盘利用率。

4.10.3 改进方案

因为怎么样，所以可以写脚本自动平衡权重，达到最高的磁盘利用率。需要统计各个 osd 上面的 pg 数目，使用如下脚本实现，如图 4.47 所示：

```
ceph pg dump|
grep '^23\.'|
awk -F ' ' '{print $1, $15}'|
awk -F "[ ]|[[ ]|[,]|[]" '{print $3, $4}'|
tr -s ' ' '\n'|
sort|
uniq -c|
sort -r|
awk '{printf("%s\n", $1)}'|
awk '{x[NR]=$0;s+=$0;n++;} END{a=s/n;for(i in x) {ss += (x[i]-a)^2} sd = sqrt(ss/n); print "SD = "sd}'
dumped all in format plain
SD = 8.71607
```

图 4.47 脚本代码截图

根据统计的 pg 数，可以计算调整后的 osd 上面的 pg 数的方差来判断效果，每次方差在减小就说明分别相对均匀。

编写脚本实现自动调整 reweight 权值；详细脚本见附录 2。

4.10.4 结果分析

下面本节对文件写入后，映射到 PG 上的文件数量，并加以分析，图 4.48 和 4.49 就是两次文件写入后映射到 PG 上的文件数量图。

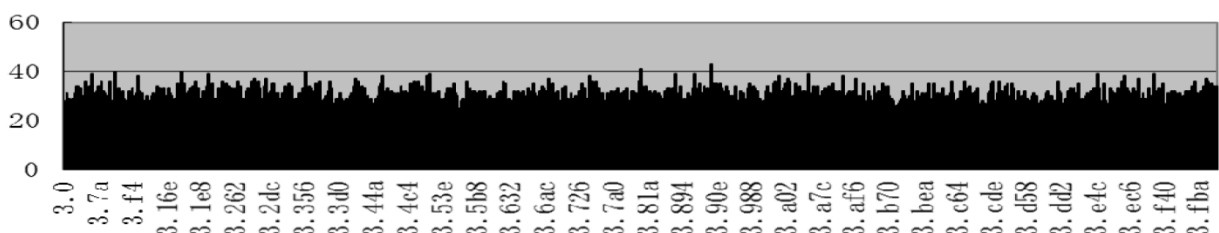


图 4.48 第一次文件写入后映射到 PG 上的文件数

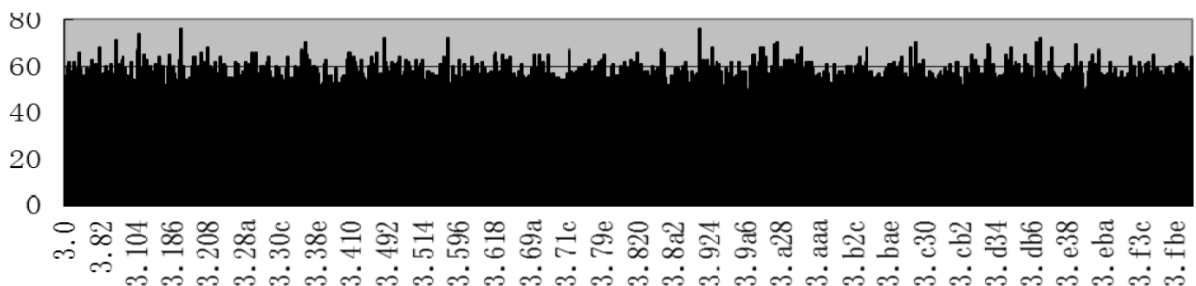


图 4.49 第二次文件写入后映射到 PG 上的文件数

从上面的两个图可以看出调整 OSD 的 reweight 值对 OSD 的分配影响明显。查看磁盘的使用率，峰值和均值差额占写入量的 0.065，可认为文件映射对 OSD 基本均匀。

4.11 本章小结

本章介绍了接口开发需要的框架、开发要求和设计的思路；非常详细的阐述了基于对象存储的系统的底层架构设计，以及对象存储系统管理模块和功能模块的设计；最后分别描述实现的过程；并附上主要代码截图。实现了包括 CephRGW 和 OpenStack 结合的系统各个子架构，对象存储用户展示模块，查看 Bucket 模块和查看 Object 模块。并提出对 OSD 权重的优化，初步分析效果明显。

第五章 对象存储系统的部署和调试

对象存储系统的搭建和调试分几个模块进行,先是底层 OpenStack 的安装,然后安装 Ceph 集群,集群必须是健康的状态;CephRGW 是一个独立的模块,将其部署成功之后,利用 Amazon 的命令行工具来实现访问,这里最重要的就是用户的 key 信息, S3 新建用户的时候都是随机生成一长串的 Id 和 Key,这是用户的身份证;将这个对象存储服务添加到 Keystone 的列表中,这样所有的组件都能来查询该服务,然后是接口层的开发,需要什么功能就必须在 Dashboard 中添加什么接口层代码;最后将接口层全部实现,包括集群状态展示一系列管理模块开发,和对象存储系统的开发。如图 5.1 所示是整个系统搭建和调试的流程图:

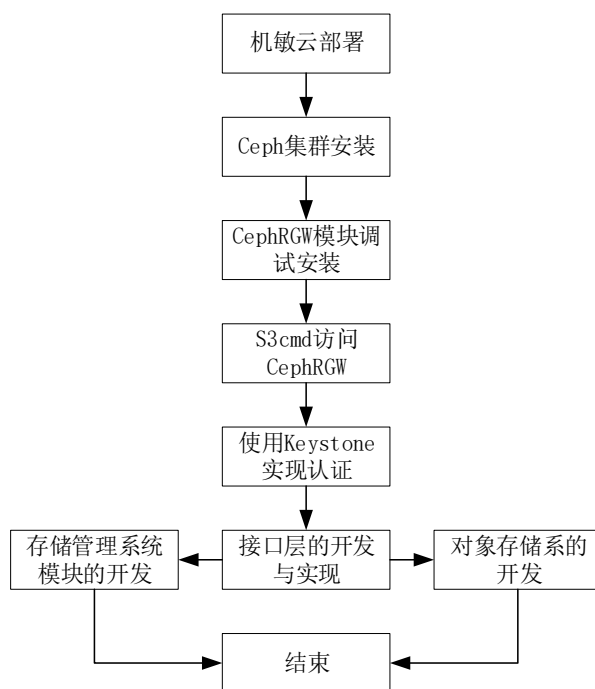


图 5.1 系统搭建流程图

5.1 测试环境部署

此时环境需要的硬件有三台服务器,并各自配有两个网卡和 2T 硬盘;如下开始来部署系统,显示演示底层的实现:

5.1.1 机敏云安装

因为公司产品机敏云已经将部署脚本及 O 版本镜像做到 U 盘里了,

第一步：先是对 Controller 节点的安装。

第二步：设置 glance（镜像存储）大小和磁盘位置，Nova 的本地存储会占用所选择的磁盘。

第三步：设置 ip 地址，子网掩码，和网关，ip 地址范围从 1 至 200，前面的网段一样，接下来就是等待服务器安装部署，完成后，浏览器输入刚刚设置的 ip 地址，如图 5.2 所示，表面环境还在部署中：



图 5.2 安装部署图

当浏览器提示输入用户名和密码的时候就直接登陆，进入 dashboard，可以创建云主机，至此，机敏云就安装完成了。

5.1.2 Ceph 集群安装

5.1.2.1 Monitor 节点安装

使用脚本安装 Ceph 集群，将外网和 OSD 之间的网络全部设置成 192.222.1.0/24，这样来安装单节点集群，之后安装 Monitor 节点，将 Host05 服务器作为 Monitor 节点，如图 5.3 所示，返回 True 的时候就表示集群和 Monitor 节点安装成功：

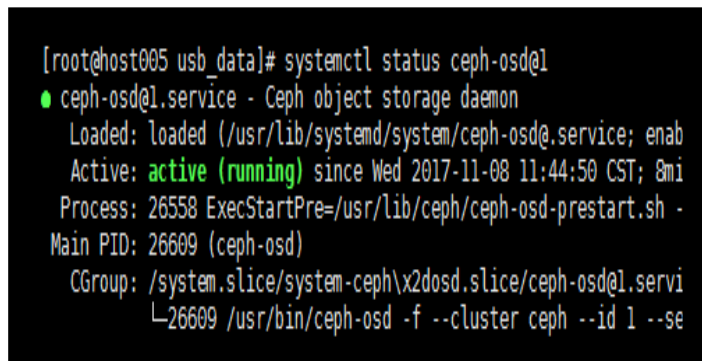
```
[root@host005 ~]# storageinstaller.  
call method: create_cluster  
192.222.1.0/24  
192.222.1.0/24  
return: True
```

```
[root@host005 ~]# storageinstall  
call method: add_monitor  
host005  
192.222.1.5  
return: True  
[root@host005 ~]#
```

图 5.3 集群和 Monitor 节点安装示意图

5.1.2.2 osd 的添加和激活

当 monitor 节点安装成功之后，需要手动添加三个 osd，使用 `ceph-disk` 命令，并修改 `ceph.conf` 文件，可以先将副本设置成 1，以及添加最大名称限制，如图 5.4 所示

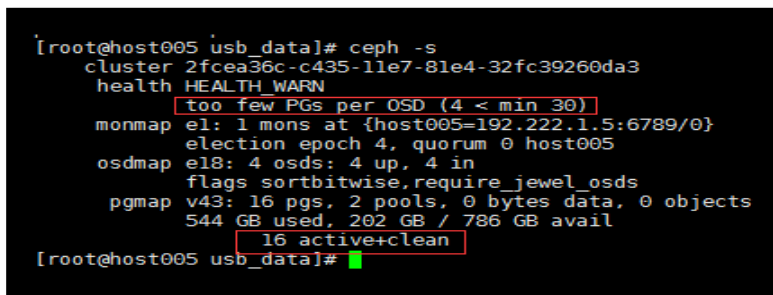


```
[root@host005 usb_data]# systemctl status ceph-osd@1
● ceph-osd@1.service - Ceph object storage daemon
   Loaded: loaded (/usr/lib/systemd/system/ceph-osd@1.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2017-11-08 11:44:50 CST; 8min ago
     Process: 26558 ExecStartPre=/usr/lib/ceph/ceph-osd-prestart.sh -s (code=exited, status=0/SUCCESS)
    Main PID: 26609 (ceph-osd)
      CGroup: /system.slice/system-ceph\x2dosd.slice/ceph-osd@1.service
              └─26609 /usr/bin/ceph-osd -f --cluster ceph --id 1 --setkeyring /etc/ceph/ceph.conf
```

图 5.4 osd 激活示意图

5.1.2.3 Pool 的创建

创建 pool，名称为 `zhoulin-pool`，设置 pg 数为 8，并查看 pools 的数目，此时集群的状态是警告状态，如图 5.5 所示：



```
[root@host005 usb_data]# ceph -s
cluster 2fcea36c-c435-11e7-81e4-32fc39260da3
health HEALTH_WARN
  too few PGs per OSD (4 < min 30)
monmap el: 1 mons at {host005=192.222.1.5:6789/0}
election epoch 4, quorum 0 host005
osdmap el8: 4 osds: 4 up, 4 in
flags sortbitwise,require_jewel_osds
pgmap v43: 16 pgs, 2 pools, 0 bytes data, 0 objects
544 GB used, 202 GB / 786 GB avail
16 active+clean
[root@host005 usb_data]#
```

图 5.5 pools 和集群示意图

5.2 环境模块调试

5.2.1 RGW 和 Nginx 服务器

可以把 Ceph RGW 简单理解成 Ceph 集群的一个客户端，用户通过它间接的访问 Ceph 集群，因此选择先在实验环境的 `host005` 服务器上部署一个 RGW 节点。

在节点上安装 `Ceph-radosgw` 和安装 `Nginx`，在服务器上安装 `nginx`，这里采用 `rpm` 方式安装，最后安装成功会显示 `Complete!` 如图 5.6 所示：

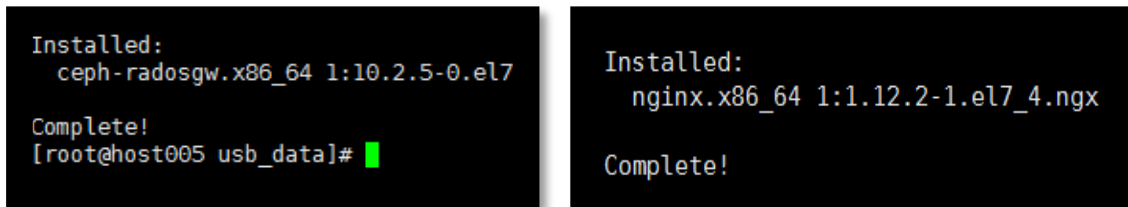


图 5.6 RGW 和 Nginx 安装成功示意图

Ceph-radosgw 和 Nginx 完成之后,需要创建 RGW 用户和 keyring,在服务器上创建 keyring,生成 Ceph-radosgw 服务对应的用户和 key,然后为用户添加访问权限导入 keyring 到集群中,导入成功后如图 5.7 所示:



图 5.7 RGW 导入 key 示意图

5.2.2 参数配置

在 host005 服务器的 /etc/Ceph.conf 上添加以下内容,完成 RGW 的配置,在 /etc/nginx/nginx.conf 中 http 段落中添加如下内容,如图 5.8 所示:

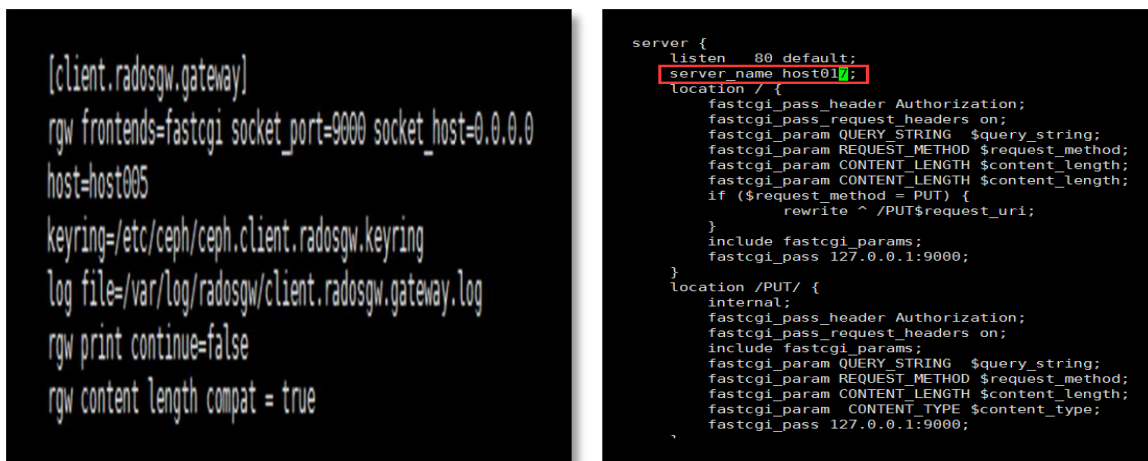


图 5.8 RGW 和 Nginx 配置示意图

5.2.3 检测 S3 服务

启动 RGW,查看 radosgw 是否正常,如图所示,一切正常,使用 curl 命令来访问 S3 服

务，如果出现下面的内容则表示服务是正常，效果如图 5.9 所示：

```
[root@host005 nginx]# curl host005
<?xml version="1.0" encoding="UTF-8"?><ListAllMyBucketsResult xmlns="http://
ets"></Buckets></ListAllMyBucketsResult>[root@host005 nginx]#
```

图 5.9 S3 服务示意图

创建 S3 用户，用户名为 zhoulin：

radosgw-admin 是 RGW 服务的命令行管理工具。已经知道了 RGW 兼容绝大部分 Amazon S3 API，下面先使用 radosgw-admin 来创建一个 S3 用户并记录下主要的 key 信息，如图 5.10 所示：

```
"keys": [
  {
    "user": "zhoulin",
    "access_key": "V3VN9RUJ6QPCAL09UIX2",
    "secret_key": "ujE1NdOpMvWlCDdTS38je9yfcZ5jcFGSBnuRzvJj"
  }
],
```

图 5.10 创建用户示意图

keys 这个信息非常重要，每次创建用户都会生成一个唯一的 key，用于下面配置客户端 s3cmd。

5.2.4 RGW 的访问

5.2.4.1 工具的配置

S3cmd 安装完成后，根据刚刚创建用户生成的 keys 等信息完成 S3cmd 的配置，主要的就是 Access key 和 Secret Key 两个信息，并生成了配置文件，需要将配置文件中主机名改成本文所用的 host005 主机，如图 5.11 所示：

```
gpg_passphrase = 
guess_mime_type = True
host_base = s3.amazonaws.com
host_bucket = %(bucket)s.s3.amazonaws.com
human_readable_sizes = False
invalidate_default_index_on_cf = False
invalidate_default_index_root_on_cf = True
invalidate_on_cf = False
```

```
gpg_passphrase = 
guess_mime_type = True
host_base = host005
host_bucket = %(bucket)s.host005
human_readable_sizes = False
invalidate_default_index_on_cf = False
invalidate_default_index_root_on_cf = True
invalidate_on_cf = False
```

图 5.11 S3cmd 客户端配置示意图

5.2.4.2 RGW 模块的使用

至此 RGW 已经安装完成, 使用 S3cmd 操作它时必须清楚 S3 的接口, 如下演示创建 Bucket、查看桶, 在新建 Bucket 的时候, 名字太短也会报错的, 当桶新建完成后, 可以使用 ls 命令查看所有的桶, 也可以使用 s3cmd rb s3://bucketname 来删除已建的桶如图 5.12 所示:

```
[root@host005 ceph]# s3cmd ls
2017-11-08 08:52 s3://My-test-bucket1
2017-11-08 08:52 s3://My-test-bucket2
2017-11-08 08:57 s3://Zhoulin-test-bucket
2017-11-08 08:57 s3://Zhoulin-test-bucket01
[root@host005 ceph]#

[root@host005 ceph]# s3cmd rb s3://Zhoulin-test-bucket
Bucket 's3://Zhoulin-test-bucket/' removed
[root@host005 ceph]# s3cmd ls
2017-11-08 08:52 s3://My-test-bucket1
2017-11-08 08:52 s3://My-test-bucket2
2017-11-08 08:57 s3://Zhoulin-test-bucket01
```

图 5.12 Bucket 操作示意图

接下来桶的操作完成之后, 使用 s3 客户端来操作 Object, 效果如图 5.13 所示:

```
[root@host005 ~]# s3cmd put testObject01.txt s3://Zhoulin-test-bucket01
WARNING: Module python-magic is not available. Guessing MIME types based on file extensions.
upload: 'testObject01.txt' -> 's3://Zhoulin-test-bucket01/testObject01.txt' [1 of 1]
16 of 16 100% in 0s 763.36 B/s done

[root@host005 ~]# s3cmd ls s3://Zhoulin-test-bucket01
2017-11-08 09:54 16 s3://Zhoulin-test-bucket01/testObject01.txt
```

图 5.13 Object 操作示意图

5.2.5 Keystone 认证

Keystone 自身作为对象存储服务的入口 (endpoint), 需要配置指向 Ceph 的网关。如图 5.14 所示:

```
openstack service create --name=swift \
    --description="Swift Service" \
    object-store

+-----+-----+
| Field | Value |
+-----+-----+
| description | Swift Service |
| enabled | True |
| id | 37c4c0e79571404cb4644201a4a6e5ee |
| name | swift |
| type | object-store |
+-----+-----+

openstack endpoint create --region RegionOne \
    --publicurl "http://radosgw.example.com:8080/swift/v1" \
    --adminurl "http://radosgw.example.com:8080/swift/v1" \
    --internalurl "http://radosgw.example.com:8080/swift/v1" \
    swift

+-----+-----+
| Field | Value |
+-----+-----+
| adminurl | http://radosgw.example.com:8080/swift/v1 |
| id | e4249d2b60e44743a67b5e5b38c18dd3 |
| internalurl | http://radosgw.example.com:8080/swift/v1 |
| publicurl | http://radosgw.example.com:8080/swift/v1 |
| region | RegionOne |
| service_id | 37c4c0e79571404cb4644201a4a6e5ee |
| service_name | swift |
| service_type | object-store |
+-----+-----+
```

图 5.14 Keystone 添加令牌截图

关键的是 URL 的管理。管理令牌是为管理请求在内部配置的令牌。

Ceph 对象的网关会定期查询重点列出撤销令牌。这些请求被编码和签名。另外，梯形图可以配置为提供自签名令牌，这些令牌也被编码和签名。网关需要能够解码和验证这些签名的消息，并且过程要求适当地设置网关。目前，Ceph 对象的网关将只有在编译与 NSS 可以执行程序。Ceph 对象配置网关工作重点也需要转换的 OpenSSL 证书，重点用于创建到 NSS 数据库格式的要求。

5.3 系统管理模块展示

5.3.1 集群状态预览效果展示

对象存储管理系统的作用是监测底层的运行环境，当前可以实现 Ceph 集群状态的检测，还有 RGW 用户和详细信息查看。

在浏览器中输入 192.222.1.16（16 服务器 IP），便可以访问，进入登录界面，如图 5.15 所示：



图 5.15 登录界面图

输入用户名和密码后，点击登录，进入查看和管理对象存储系统。

（1）集群状态展示

点击 overview，可以查看对象存储集群详情并监控。如图 5.16 所示：

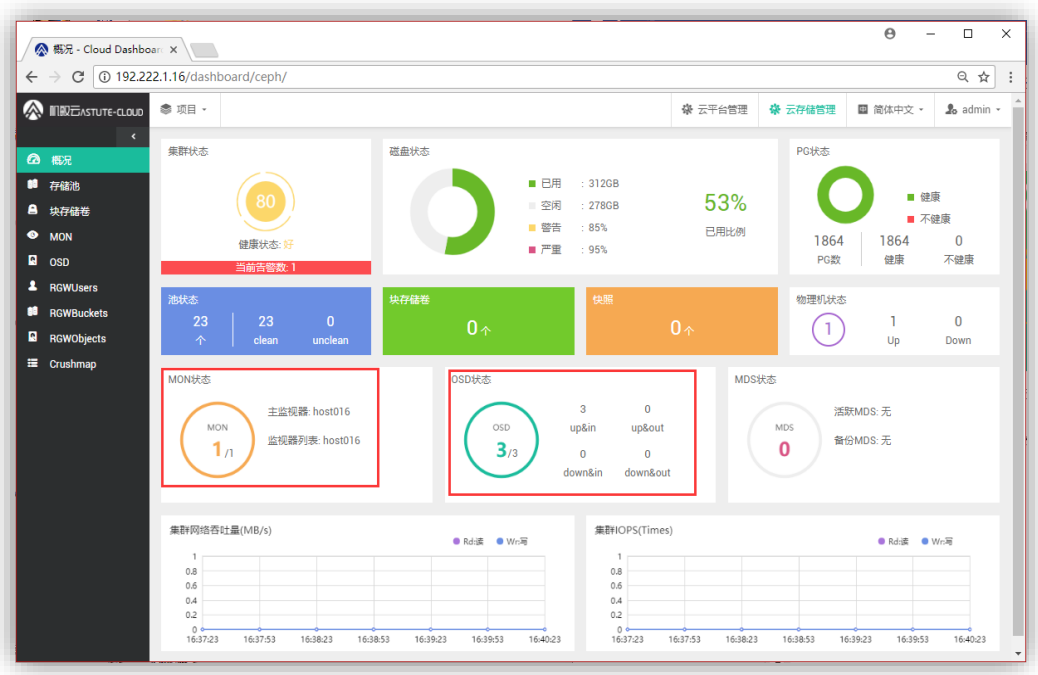


图 5.16 存储集群详情监控界面图

5.3.2 OSD 添加效果展示

(1) OSD 的添加

对 OSD 的状态有四种，分别为 in/out 和 up/down 的组合，本文并没有开发管理这四种状态的模块，只设计实现了添加/删除 OSD 的功能。如图 5.17 所示是添加 OSD 的功能截图：

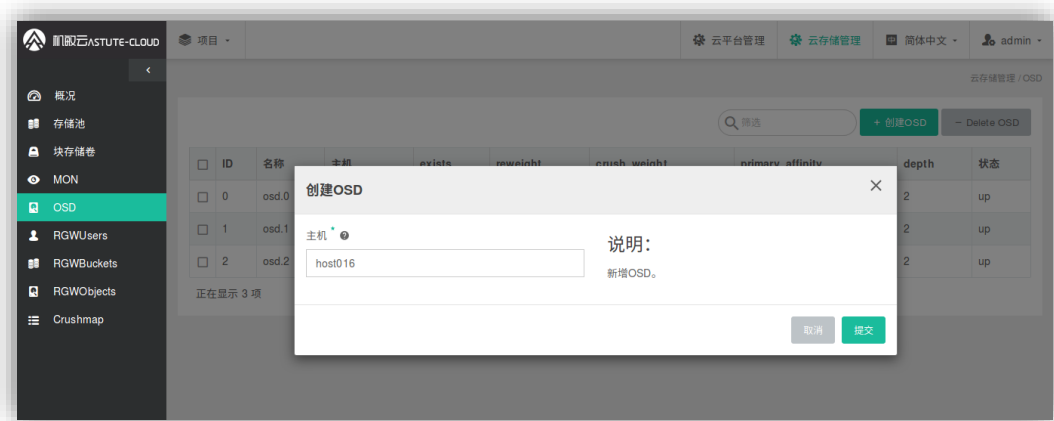


图 5.17 添加 OSD 截图

5.4 系统功能模块展示

对象存储功能模块是运用 CephRGW 的应用,为用户提供最直观、方便的对象存储应用。该模块不仅能够给用户提供最直接的文件存储操作,还包括可以提供文件的上传、下载等操作。对象存储系统具有很高的安全性,将人为失误导致的数据丢失率降到最低,用户可以放心的存储数据。

5.4.1 用户管理模块效果展示

用户登录之后,就可以查看当前 rgw 用户的列表,点击用户可以查看用户的详细信息。下面就是用户的管理列表,如图 5.18 所示:

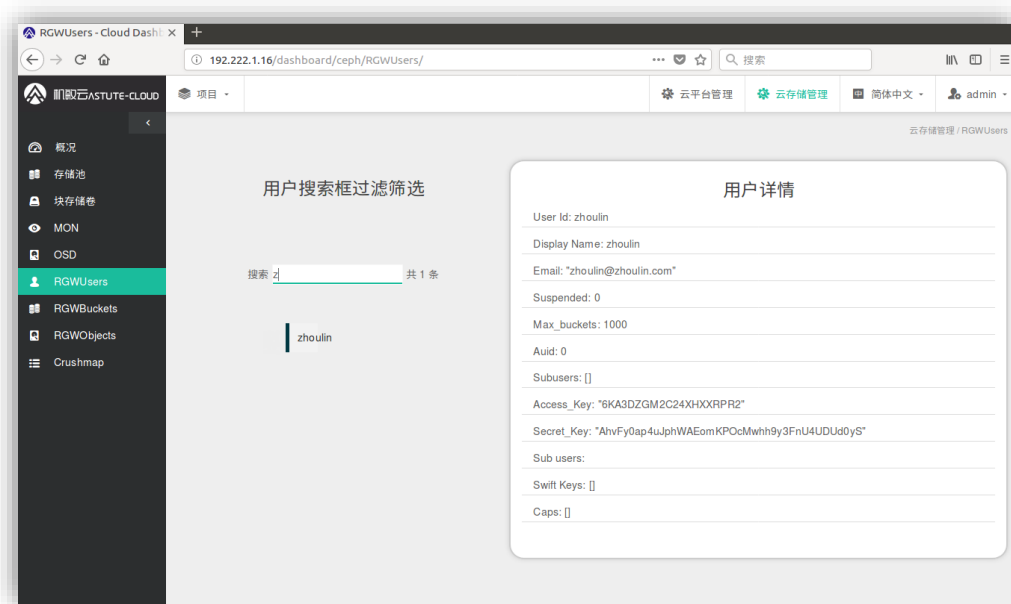


图 5.18 用户详情列表

5.4.2 Bucket 容器效果展示

每个用户都会创建不同的桶 (Buckets),可以点击用户名来查看该用户所创建的桶,如图 5.19 所示,输入 z, 就可以查看到 z 开头的用户,右侧会列出用户所创建的桶 (Buckets)。

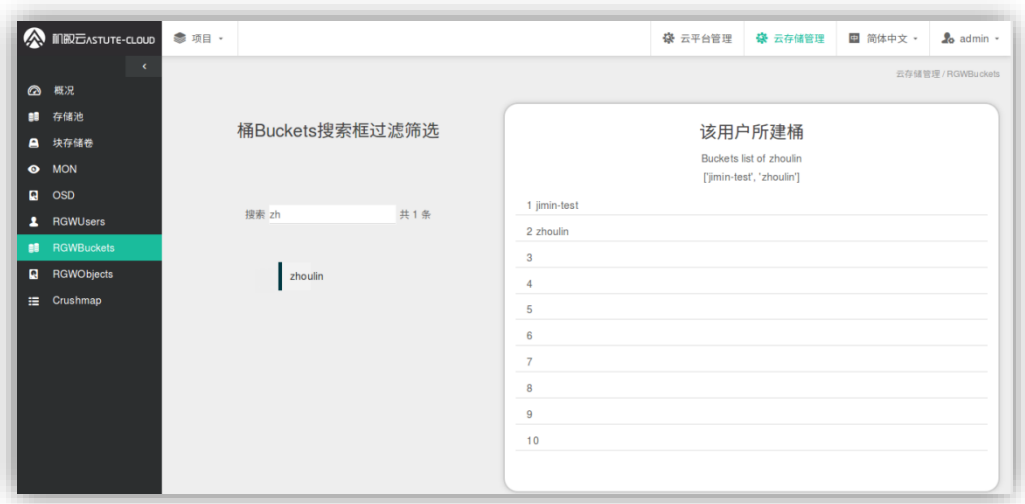


图 5.19 桶展示列表

5.4.3 Object 操作效果展示

系统所存储的文件主要就是用户需要放的 Objects，用户放到系统中，可以进行直观的管理，基于 web 界面进行操作，可以查看存储的对象列表，已经上传、下载文件。

需要点击某一个 Bucket，才能查看该 Bucket 下面的所有对象。如图 5.20 所示：

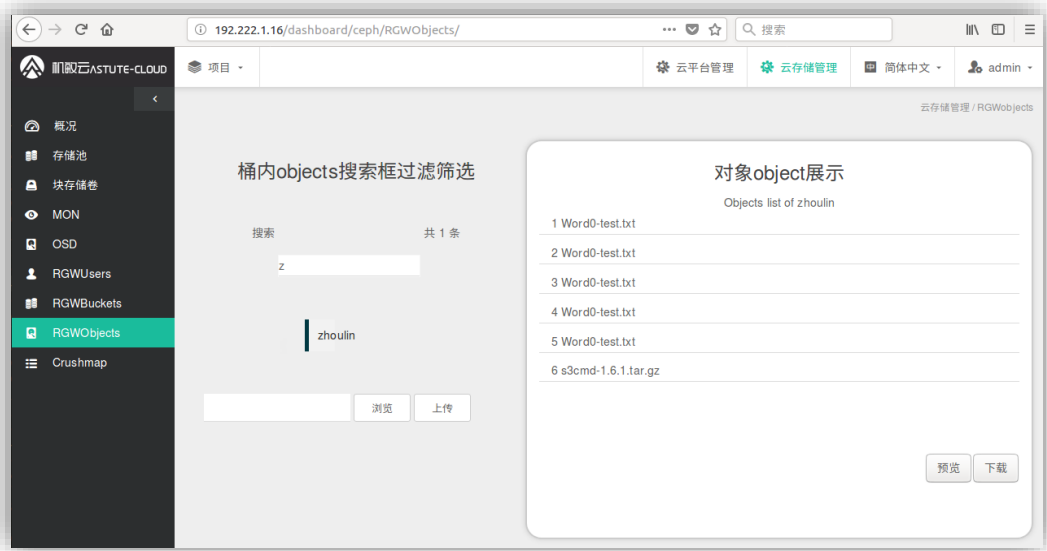


图 5.20 查看 objects 截图

系统支持文件上传功能和下载功能，界面分别如图 5.21 所示：



图 5.21 上传和下载功能示意图

下载的速度取决于当前的网速。

5.5 系统性能测试

5.5.1 验证系统高可靠性

本文设计的对象存储系统最大的特点和优势是 OpenStack 和 Ceph 的复合使用, 实现了取代 Swift 的目的, 从原理上来看是结合了云计算的四层结构模型, 设计与实现了一套高效的存储方案。

为了验证本系统的性能, 本文验证了上传和下载文件的功能, 以及在误删了一个 OSD 的情况下, 数据是否会丢失。

(a) 添加 OSD

在点击添加 OSD 的按钮, 则会跳出弹框, 需要输入 host016 主机名, 这样就可以自动根据 `ceph osd create` 按顺序添加 osd 进去。如图 5.22 所示:

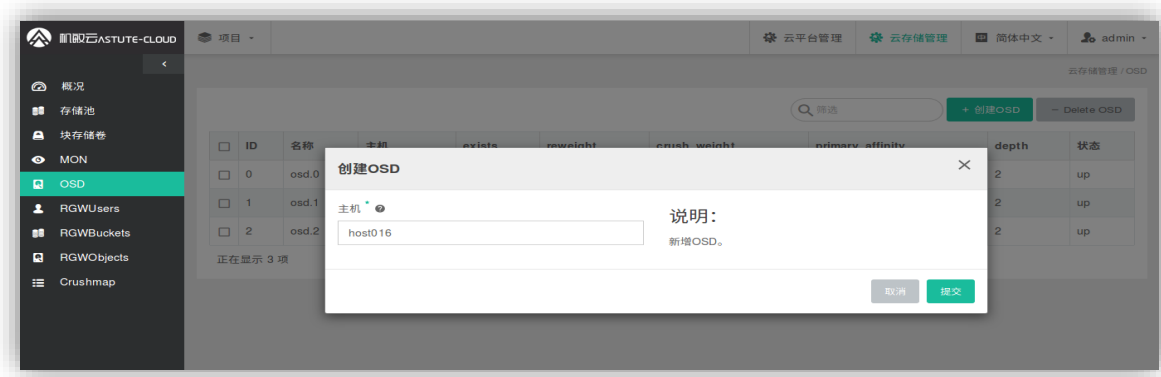


图 5.22 添加 OSD 图

添加成功之后的效果如图 5.23 所示：

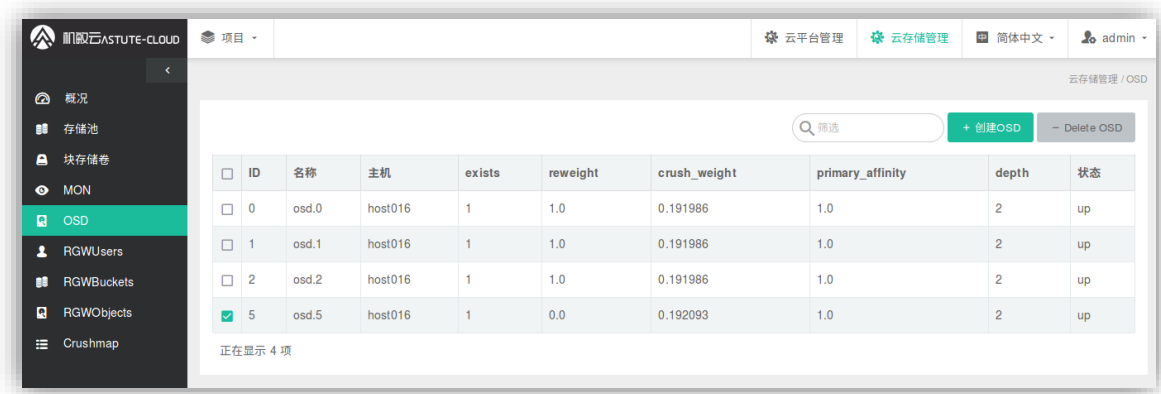


图 5.23 OSD 添加完成图

(b) 删除 OSD

删除之前的 OSD 有四个，选择点击 ID 为 5 的 osd，将其删除，如图 5.24 所示：

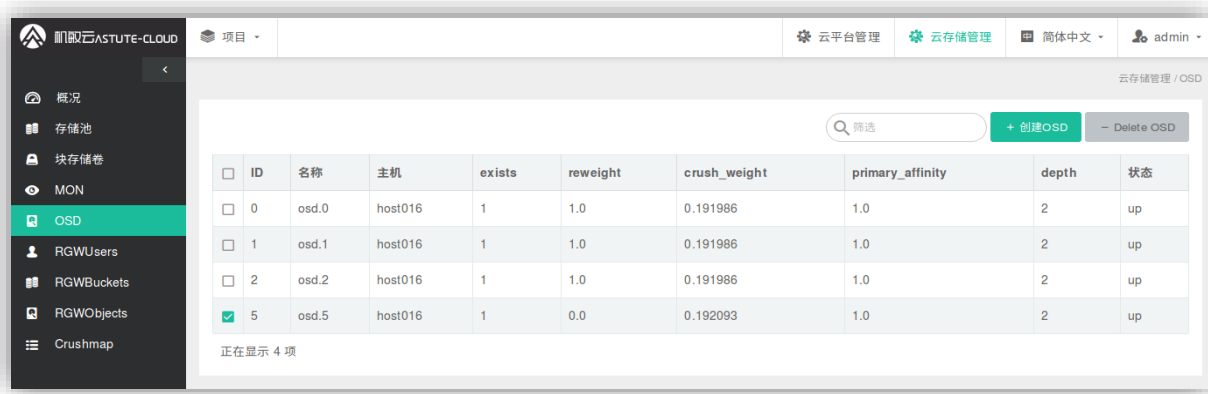


图 5.24 选择 OSD 图

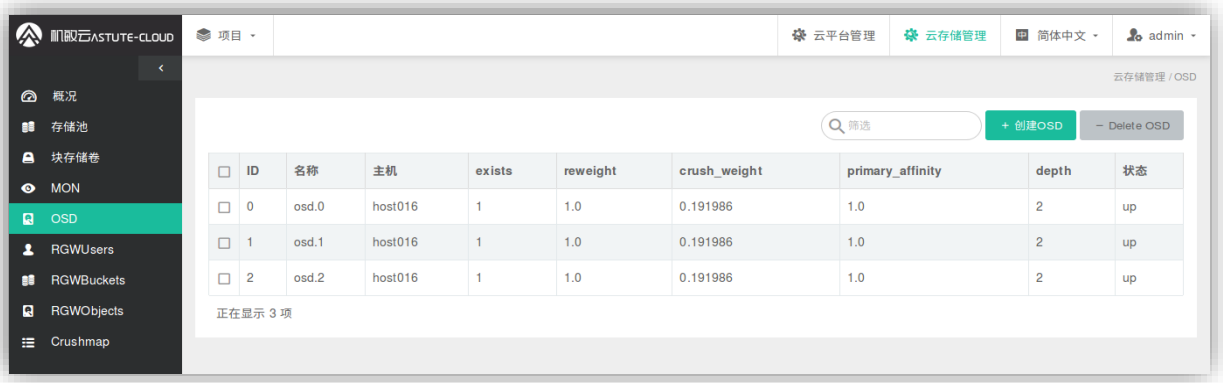


图 5.25 删除之后 OSD 图

(c) 上传 Object

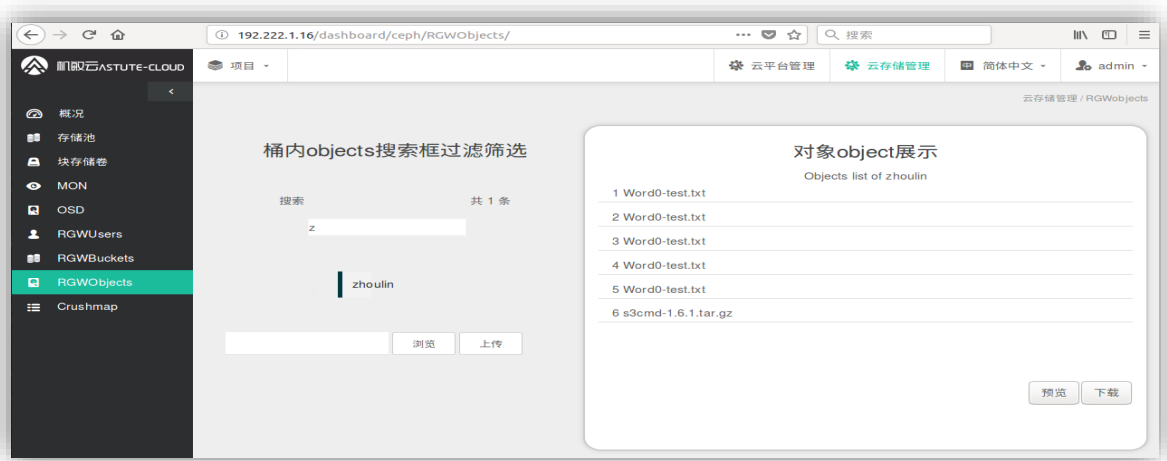


图 5.26 上传 object 之前图

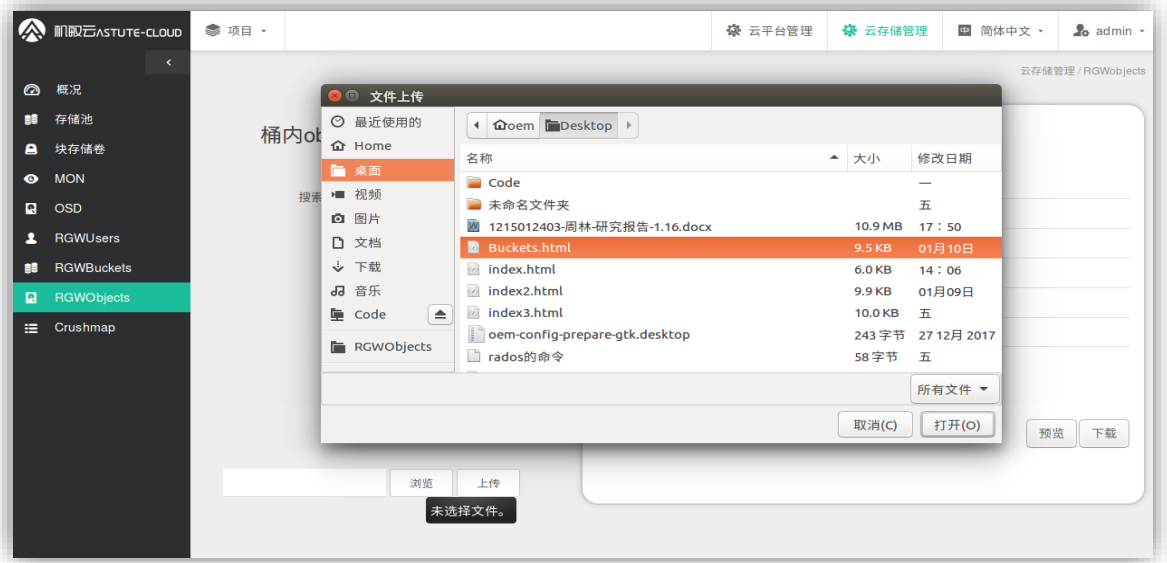


图 5.27 选择 object 图



图 5.28 完成 object 上传图

(d) 下载 Object

先选中 object，然后点击下载，就可以在默认路径（/root/temp）下找到下载的文件。



图 5.29 下载 object 截图

(e) OSD 破损测试

下面重要的是该系统的可靠性，实验一种人为破坏掉节点的情况：在删除了一个 OSD 之后，仍然能够找到存储的文件。如图 5.30 所示：

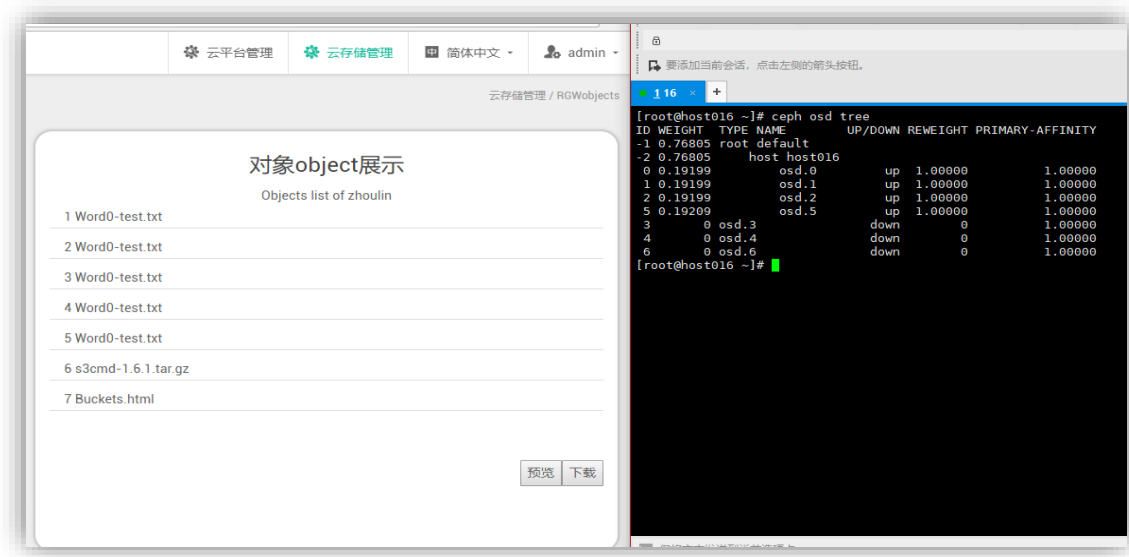


图 5.30 删除 osd 演示图

从以上的示例可以看出，本文设计的系统已经可以初步给客户提供对象存储服务，只要用户被认证，使用相同的 AccessKey 和 SecretKey 访问不同的主机都可以得到数据，Ceph 集群之间是协同工作的，存储对象不会被固定的存储在某个节点上。它们不唯一的分布在某个节点上，集群有一个内部网络，每个 osd 都会提供心跳数据向 monitor 汇报，所以能够为用户提供稳定的存储服务，而当集群中的主机增加或者删除 osd 的时候，集群内会进行存储空间再分配，这能够有效的使用硬盘。该存储集群具有高复用性，如果任何一台主机宕机并不会造成系统得崩溃。

5.5.2 验证系统的分布式特性

系统部署完成之后，本节测试两个 OSD 之间的带宽，可以侧面的反应出 OSD 是网络间分布式存在的，各个节点之间都是通过集群网络来进行互联。本节在 ceph1 上运行 iperf -s -p 6900，在 ceph2 上运行 iperf -c ceph -p 6900，重复多次，得到以下的数据（iperf 是一个网络性能测试工具）。如图 5.31 所示：

```
root@ceph2:~# iperf -c ceph1 -p 6900
-----
Client connecting to ceph1, TCP port 6900
TCP window size: 85.0 KByte (default)
-----
[ 3] local 192.168.56.103 port 41773 connected with 192.168.56.102 port 6900
[ ID] Interval      Transfer    Bandwidth
[ 3]  0.0-10.0 sec  1.25 GBytes  1.08 Gbits/sec
```

图 5.31 OSD 通信图

如上图所示，测试了两个节点之间的流量，说明两个 OSD（ceph1 和 ceph2）是通过网络连接，以此验证了系统的分布式特性。

5.5.3 验证系统的高效平衡性

本小节将继续验证系统的自动平衡性，zhoulin.txt，里面随便写了一些内容，可以用命令查看到文件分布在 OSD1 和 OSD2 上；并且可以在 OSD1 中查看到新建的文件，如图 5.32 所示：

```
root@ceph-node1:~# rados put my-object ./zhoulin.txt --pool=data
root@ceph-node1:~# rados -p data ls
my-object
root@ceph-node1:~# ceph osd map data my-object
osdmap e178 pool 'data' (8) object 'my-object' -> pg 8.c5034eb8 (8.0) -> up ([1,2], p1) acting ([1,2], p1)

root@ceph-node1:~# cat zhoulin.txt
this is zhoulin data
root@ceph-node1:~# ceph osd map data my-object
osdmap e188 pool 'data' (8) object 'my-object' -> pg 8.c5034eb8 (8.0) -> up ([1,2], p1) acting ([1,2], p1)
root@ceph-node1:~# cat /var/lib/ceph/osd/ceph-1/current/8.0_head/my-object_head_C5034EB8_8
this is zhoulin data
```

图 5.32 文件查看图

然后将 OSD2 所在的节点 down 掉。通过 ceph osd tree 可以查看，如图 5.33 所示：

```
root@ceph-node1:~# ceph osd tree
ID WEIGHT TYPE NAME UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.10211 root default
-2 0.02917 host ceph-node1
 0 0.01459 osd.0 up 1.00000 1.00000
 1 0.01459 osd.1 up 1.00000 1.00000
-3 0.04376 host ceph-node2
 2 0.01459 osd.2 down 0 1.00000
 3 0.01459 osd.3 down 0 1.00000
 6 0.01459 osd.6 down 0 1.00000
-4 0.02917 host ceph-node3
 4 0.01459 osd.4 up 1.00000 1.00000
 5 0.01459 osd.5 up 1.00000 1.00000
root@ceph-node1:~#
```

图 5.33 down 掉 OSD2 示意图

此时，再查看刚刚的文件已经分布到 OSD5 上面了，这就是系统的自动平衡性。如图 5.34 所示：

```
root@ceph-node1:~# ceph osd map data my-object
osdmap e190 pool 'data' (8) object 'my-object' -> pg 8.c5034eb8 (8.0) -> up ([1], p1) acting ([1], p1)
root@ceph-node1:~# ceph osd map data my-object
osdmap e190 pool 'data' (8) object 'my-object' -> pg 8.c5034eb8 (8.0) -> up ([1], p1) acting ([1], p1)
root@ceph-node1:~# ceph osd map data my-object
osdmap e206 pool 'data' (8) object 'my-object' -> pg 8.c5034eb8 (8.0) -> up ([1,5], p1) acting ([1,5], p1)
root@ceph-node1:~# ceph osd map data my-object
osdmap e206 pool 'data' (8) object 'my-object' -> pg 8.c5034eb8 (8.0) -> up ([1,5], p1) acting ([1,5], p1)
```

图 5.34 文件分布图

查看文件，如图 5.35 所示：

```
root@ceph-node3:~# cat /var/lib/ceph/osd/ceph-5/current/8.0_head/my-object_head_C5034EB8_8
this is zhoulin data
root@ceph-node3:~#
```

图 5.35 查看文件图

至此，就验证了系统的自动平衡性；主要过程就是 PG 的再平衡，如图 5.36 示：

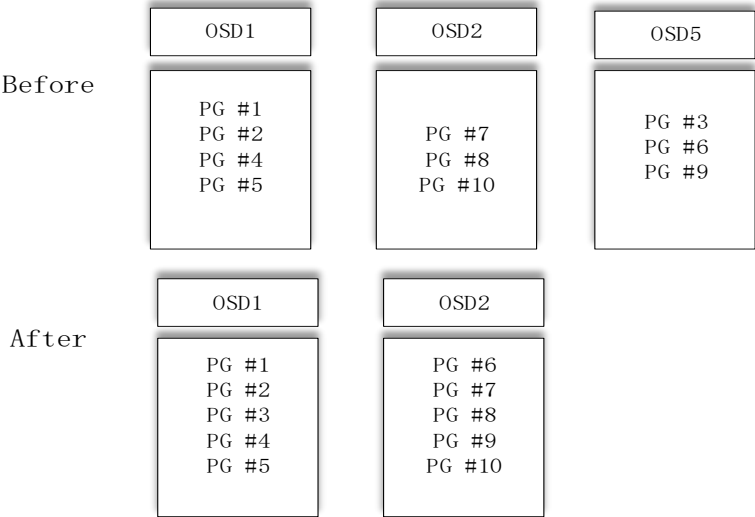


图 5.36 OSD 自动平衡示意图

5.6 本章小结

本章主要阐述的是对象存储系统的部署和调试，介绍了如何搭建集群环境，怎么添加 mon 节点和 osd 节点，然后又演示了系统的运行和相应的结果。最后测试了在随机 down 掉三个 osd 的情况下，数据的完整性。验证了本系统的可扩展性，稳定性及其可控性等优点。

第六章 总结与展望

本篇论文主要是基于机敏公司对 Ceph 的使用情况出发，因为机敏公司已经使用了 Ceph 的块存储，然后结合分析了云存储技术和用户的需求，以及当前 OpenStack 自带的对象存储模块，Swift 有哪些缺点和问题，并结合当前的 OpenStack 和 Ceph 等云存储相关技术，设计出了一个完整的对象存储系统。使用了 Django 框架，采用 MVC 的设计模式，设计了一套对象存储管理系统，付费用户可以非常方便的管理和查看系统的使用状况，这对公司机敏云的开发有很大的意义。

本文首先介绍了在目前大数据时代，信息量已经爆炸性增长，但是用户们不可避免的遇到了数据存储灵活性问题，而且存储的成本高居不下，用户所存的数据安全性、可靠性都很难得到保障。再结合机敏公司对 Ceph 块存储的使用，本文提出了一个基于 CephRGW 的云存储方案。先介绍了云计算的相关技术和原理，结合性能需求，给出了系统的架构设计，然后使用 Python 语言进行开发，实现了系统的基本功能，最后对系统的应用做了简要的阐述且做了破坏性测试，验证了系统的优越性。

本文的主要成果就是利用云计算技术结合 RGW 对象存储技术实现了数据在云端的管理。系统的界面设计参考了 inkscope 的界面。不足之处是，在资源的分享和搜索上还不能做到，更是没有 inkscope 的便捷性和丰富应用性，另外系统还将继续实现多终端查看，实现断点续传等功能，这个都需要进一步的研究和实现。云计算技术正在蓬勃的发展，在很多方向在不断的更新，如利用模糊逻辑来提供云计算的服务质量^[48]，基于对象监测的移动云计算^[49]，还有新的工作流调度方法^[50]，而且人工智能也和云计算相结合^[51]，在以后的版本中肯定需要更新迭代，不断的与新技术融合。

本系统的下一步工作是，为界面丰富更多的功能，考察多用户的并发使用。对系统的展望是，将来满足多用户的并发使用，大文件的断点续传，提供更丰富和个性化的存储应用，将其与 Docker、SDN、人工智能等新型技术结合，创造独一无二的机敏云产品。

参考文献

- [1] 罗军舟, 金嘉晖, 宋爱波, 等. 云计算:体系架构与关键技术[J]. 通信学报, 2011, 32 (7): 3-21.
- [2] 张怀南, 杨成. 我国云计算教育应用的研究综述[J]. 中国远程教育:综合版, 2013(1): 20-26.
- [3] 姜蕴莉. 云计算在高校中的应用分析[J]. 软件, 2014 (12): 120-122.
- [4] 刘谭兴. 分布式键值存储系统的设计与实现[D]. 西安: 西安电子科技大学, 2015.
- [5] 胡珊珊. 面向云存储的非结构化数据存储研究与应用[D]. 广州: 广东工业大学, 2014.
- [6] Weil S A, Brandt S A, Miller E L, *et al.* Ceph: a scalable, high-performance distributed file system[C]. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006:307-320.
- [7] Wang F, Nelson M, Oral S, *et al.* Performance and scalability evaluation of the Ceph parallel file system[C]. Parallel Data Storage Workshop, 2013:14-19.
- [8] 师绍秋. 面向云计算的 Restful 信息资源池的设计与实现[D]. 上海: 上海交通大学, 2014.
- [9] 宋顺, 姜莹. 基于 Amazon S3 兼容云存储平台的媒体文件存储与访问[J]. 信息通信技术, 2013 (1): 57-62.
- [10] 什么是 AWS 免费试用套餐? - Amazon Web Services. http://aws.amazon.com/cn/free/what-is-aws-free-usage-tier/?nc1=f_ls
- [11] 宋宁, 张志林, 张聪. 基于云存储服务的云编辑功能实现探析——以谷歌云端硬盘(Google Drive)与微软云端硬盘(One Drive)云存储服务为例[J]. 中国编辑, 2014 (6): 26-28.
- [12] 曾平. 国内云计算研究现状与未来[J]. 电脑与信息技术, 2014, 22 (1): 41-45.
- [13] 启明. 阿里云全球增速最快 华尔街分析师称赞阿里布局[N]. 世纪经济, 2015.8.21.
- [14] 张振刚, 张小娟. 广州智慧城市建设的现状、问题与对策[J]. 科技管理研究, 2015, v.35;No.338 (16): 87-93.
- [15] 周明. 云计算中的数据安全相关问题的研究[D]. 南京: 南京邮电大学, 2013.
- [16] 杨振贤. 基于云计算的安全数据存储研究与设计[J]. 信息安全与技术, 2011 (10): 13-15.
- [17] 王建宇. OpenStack 平台与 Ceph 统一存储的集成[J]. 中国管理信息化, 2016, 19(4): 168-168.
- [18] Wu C F, Hsu T C, Yang H, *et al.* File placement mechanisms for improving write throughputs of cloud storage services based on Ceph and HDFS[C]. IEEE International Conference on Applied System Innovation, 2017:1725-1728.
- [19] Sheela P S, Choudhary M. Deploying an OpenStack cloud computing framework for university campus[C]. International Conference on Computing, Communication and Automation, 2017:819-824.
- [20] Singh M, Gupta P K, Srivastava V M. Key challenges in implementing cloud computing in Indian healthcare industry[C]. IEEE Pattern Recognition Association of South Africa and Robotics and Mechatronics, 2017:162-167.
- [21] Lai Y T, Hsiao P W, Lee W T. Resource usage aware scheduling in OpenStack[C]. IEEE Region 10 Conference, 2017:668-672.
- [22] Zhao D F, Mohamed M, Ludwig H. Locality-aware Scheduling for Containers in Cloud Computing[C]. IEEE Transactions on Cloud Computing, 2018:1-1.
- [23] Bruschi R, Genovese G, Iera A, *et al.* OpenStack Extension for Fog-Powered Personal Services Deployment[C]. International Teletraffic Congress. 2017:19-23.
- [24] Biswas P, Patwa F, Sandhu R. Content Level Access Control for OpenStack Swift Storage[C]. ACM Conference on Data and Application Security and Privacy, 2015:123-126.
- [25] Zhang Y, Krishnan R, Sandhu R. Secure Information and Resource Sharing in Cloud Infrastructure as a Service[C]. ACM Conference on Data and Application Security and Privacy, 2014:81-90.
- [26] Xiang Y, Li H, Wang S, *et al.* Debugging OpenStack Problems Using a State Graph Approach[C]. ACM Sigops Asia-Pacific Workshop on Systems, 2016:13.

- [27] Kabiru Muhammad Maiyama, Demetres Kouvatso, Mohammed B, et al. Performance Modelling and Analysis of an OpenStack IaaS Cloud Computing Platform[C]. IEEE International Conference on Future Internet of Things and Cloud. IEEE Computer Society, 2017:198-205.
- [28] Kurhekar M, Kurhekar M, Kurhekar M. Physical to Virtual Migration of Ubuntu System on OpenStack Cloud[C]. ACM International Symposium on Women in Computing and Informatics, 2015:510-515.
- [29] Basnet S R, Chaulagain R S, Pandey S, et al. Distributed High Performance Computing in OpenStack Cloud over SDN Infrastructure[C]. IEEE International Conference on Smart Cloud. IEEE Computer Society, 2017:144-148.
- [30] Samundiswary S, Dongre N M. Object storage architecture in cloud for unstructured data[C]. International Conference on Inventive Systems and Control, 2017:1-6.
- [31] Rupprecht L, Zhang R, Owen B, et al. SwiftAnalytics: Optimizing Object Storage for Big Data Analytics[C]. IEEE International Conference on Cloud Engineering, 2017:245-251.
- [32] Carns P, Harms K, Jenkins J, et al. Impact of data placement on resilience in large-scale object storage systems[C]. IEEE Mass Storage Systems and Technologies, 2017:1-12.
- [33] Khorasani S A, Azmi R, Sabeti V. Secured data sharing using proxy Kerberos to improve Openstack Swift security[C]. IEEE International Conference on Computer and Knowledge Engineering, 2017:77-83.
- [34] Noor J, Islam A B M A A. iBuck: Reliable and secured image processing middleware for OpenStack Swift[C]. IEEE International Conference on Networking, Systems and Security, 2017:144-149.
- [35] Huo J, Qu H. Design and implementation of automatic defensive websites tamper-resistant based on OpenStack cloud system[C]. IEEE International Conference on Computer Science and Network Technology, 2016:280-284.
- [36] Huo J, Qu H, Wu L. Design and Implementation of Private Cloud Storage Platform Based on OpenStack[C]. IEEE International Conference on Smart City/socialcom/sustaincom, 2016:1098-1101.
- [37] Gughani S, Lu X, Panda D K. Swift-X: Accelerating OpenStack Swift with RDMA for Building an Efficient HPC Cloud[C]. IEEE International Symposium on Cluster, Cloud and Grid Computing, 2017:238-247.
- [38] Ma J W, Wang G, Liu X. DedupeSwift: Object-Oriented Storage System Based on Data Deduplication [C]. IEEE Trustcom/bigdatasec/ispa, 2017: 1069-1076.
- [39] Wei K, Yu L. Multi-level image software assembly technology based on OpenStack and Ceph [C]. IEEE Information Technology, Networking, Electronic and Automation Control Conference, 2016: 307-310.
- [40] Lee J, Song C, Kang K. Benchmarking Large-Scale Object Storage Servers[C], IEEE Computer Software and Applications Conference. 2016: 594-595.
- [41] Sha E H M, Liang Y, Jiang W, et al. Optimizing Data Placement of MapReduce on Ceph-Based Framework under Load-Balancing Constraint[C]. IEEE International Conference on Parallel and Distributed Systems, 2017:585-592.
- [42] Patil S R, Kulkarni V R, Jogdand R M, et al. A comparative review on Ceph and Swift open source cloud storage platform[C]. IEEE International Conference on Global Trends in Signal Processing, Information Computing and Communication, 2017:213-218.
- [43] Chen H M, Li C J, Ke B S. Designing A Simple Storage Services (S3) Compatible System Based on Ceph Software-Defined Storage System[C]. ACM International Conference on Multimedia Systems and Signal Processing Publisher, 2017:6-10.
- [44] Wang L, Wen Y. Design and Implementation of Ceph Block Device in Userspace for Container Scenarios[C]. IEEE International Symposium on Computer, Consumer and Control, 2016:383-386.
- [45] Ke Z, Ai H P. Optimization of Ceph Reads/Writes Based on Multi-threaded Algorithms[C]. IEEE International Conference on High PERFORMANCE Computing and Communications; IEEE International Conference on Smart City; IEEE International Conference on Data Science and Systems, 2017:719-725.
- [46] Shankar V, Lin R. Performance Study of Ceph Storage with Intel Cache Acceleration Software: Decoupling Hadoop MapReduce and HDFS over Ceph Storage[C]. IEEE International Conference on Cyber Security and

Cloud Computing, 2017:10-13.

- [47] Zhang S, Liu Z. Research on the construction and robustness testing of SaaS cloud computing data center based on the MVC design pattern[C]. International Conference on Inventive Systems and Control, 2017:1-4.
- [48] Mondal H S, Shekhar H, Hasan M T, et al. Improving quality of service in cloud computing architecture using fuzzy logic[C]. International Conference on Advances in Electrical Engineering, 2017:149-152.
- [49] Qi B R, Wu M F, Zhang L. A DNN-based object detection system on mobile cloud computing[C]. IEEE International Symposium on Communications and Information Technologies, 2017:1-6.
- [50] S. Phanikumar, Reddy G N. A novel method for scheduling workflows in cloud computing environment[C]. IEEE International Conference on Science Technology Engineering & Management, 2017:12-16.
- [51] Chen M, Herrera F, Hwang K. Cognitive Computing: Human-Centered Computing with Cognitive Intelligence on Clouds[C]. IEEE Access, 2018:1-1.

附录 1 攻读硕士学位期间撰写的论文

(1) Lin Zhou, Jianxin Song, Jinkun Yuan. Research and application of GPU pass-through based on KVM in desktop cloud, 2017 International Conference on Software, Data Processing and Information Technology (SDPIT 2017) Tianjin, China, 2017.5.24 (收录号: WOS:000414864800073)

附录 2 程序代码清单

开发的详细代码见 Git: <https://github.com/feeyman/CephRGW-.git>

分配权重脚本:

```
ceph pg dump | awk '
  /^pg_stat/ { col=1; while($col!="up") {col++}; col++ }
  /^[0-9a-f]+\.[0-9a-f]+/ { match($0, /^[0-9a-f]+/);
pool=substr($0, RSTART, RLENGTH); poollist[pool]=0;
  up=$col; i=0; RSTART=0; RLENGTH=0; delete osds;
  while(match(up,/[0-9]+/)>0) { osds[++i]=substr(up,RSTART,RLENGTH); up = substr(up,
RSTART+RLENGTH) }
  for(i in osds) {array[osds[i],pool]++; osdlist[osds[i]];}
}
ceph pg dump|grep '^23\.'
|awk -F ' ' '{print $1, $15}'
awk -F "[ ]|[.]" '{print $3, $4}'
tr -s ' '\n'
sort|
uniq -c|
sort -r|awk '{printf("%s\n", $1)}'
awk '{x[NR]=$0;s+=$0;n++} END {a=s/n;for(i in x) {ss += (x[i]-a)^2} sd = sqrt(ss/n); print "SD
="sd}'
dumped all in format plain
SD = 8.71607
```

致谢

大半年的实习生活转瞬即逝，刚去机敏科技报道的情景还历历在目，项目的开发以及论文的写作过程是辛苦的，充满了各种艰辛，然而一路走来，却也在辛苦中感受着收获的喜悦，在短暂的大半年实习生涯中，我得到了许多人的帮助和指导，在这里我对他们表示感谢。

首先要感谢我的导师宋建新教授，他知识渊博，做事认真严谨的态度，在我两年半的研究生生活中留下了深刻的影响。宋老师不仅对我们在学习和学生方面给与指导，而且在做事待人方面也为我们树立了一个良好的榜样。在这里衷心的祝愿宋老师身体健康、心想事成、家庭美满、工作顺利。

感谢机敏公司的众多同事；尤其感谢袁进坤导师，给了我很大的帮助，一开始对 OpenStack 相关的知识都是一头雾水，袁进坤导师给我列了详细的计划，给出方向，百忙之中还会关注我的进度，很多问题都会耐心的指导我，使我进步很大；也感谢王璐乐、杜俊勇和张成三位老师，他们分别在网络、存储、虚拟化三个领域都有十几年的经验，在我开发遇到问题的时候，不厌其烦的给我解答，有的时候被一个小问题困扰了半天，三位导师的指导点播真的很重要；也感谢张伟国在做 Django 开发的时候，给了我很大的帮助，帮我修改页面的一些参数。和你们在一起的日子短暂而幸福，是令人非常怀念的美好回忆。大家在生活和学习上的互相帮助和交流，促进了我的进步和成长。

感谢我的父母含辛茹苦的教育我，不论生活多么的辛苦，对我的学习和生活开支都毫不犹豫，一如既往的支持和关心我。

最后，再次感谢所有的师长、同学、亲人、好友，谢谢你们，在这里再次祝福你们。