

**2016** 级

《物联网数据存储与管理》课程

**实 验 报 告**

**姓 名 潘翔**

**学 号 U201614898**

**班 号 物联网1601班**

**日 期 2019.06.05**

目 录

[1 实验目的 1](#_Toc1878831403)

[2 实验背景 1](#_Toc2130381245)

[2.1 Account Server 1](#_Toc95616885)

[2.2 Container 1](#_Toc207840775)

[2.3 Object Server 2](#_Toc50496592)

[3 实验环境 3](#_Toc1844169683)

[3.1 系统环境. 3](#_Toc292610644)

[3.2 Minio-Server 3](#_Toc260605816)

[3.3 Minio-python API 5](#_Toc726435959)

[3.4 AWS-CLI 7](#_Toc1473737511)

[3.5 s3bench 8](#_Toc1023474301)

[3.6 CosBench测试 9](#_Toc211391287)

[3.7 OpenStack Swift CLI 9](#_Toc1540537129)

[3.8 OpenStack Swift Python-API 11](#_Toc1432152261)

[4 实验内容 12](#_Toc693700198)

[4.1 对象存储技术实践 12](#_Toc1716703651)

[4.2 对象存储性能分析 12](#_Toc92682917)

[5 实验过程 12](#_Toc1326679459)

[5.1 环境配置 12](#_Toc717760465)

[5.2 基于Cosbench进行单变量测试 13](#_Toc995864827)

[5.3 进行HDD/SSD性能比较 17](#_Toc2015513702)

[6 实验总结 20](#_Toc50125663)

[参考文献 20](#_Toc738377691)

# 实验目的

1. 熟悉对象存储技术，代表性系统及其特性；
2. 实践对象存储系统，部署实验环境，进行初步测试；
3. 基于对象存储系统，架设实际应用，示范主要功能。

# 实验背景

OpenStack Object Storage是一个高度可用，分布式，最终一致的对象blob存储。可以使用Object Storage API创建，修改和获取对象和元数据，该API是作为一组Representational State Transfer（REST）Web服务实现的。

## Account Server

表示层次结构的顶级

服务提供商会创建帐户，并拥有该帐户中的所有资源。该帐户定义容器的命名空间。容器在两个不同的帐户中可能具有相同的名称。

在OpenStack环境中，帐户与项目或租户同义。

## Container

定义对象的命名空间。两个不同容器中具有相同名称的对象表示两个不同的对象。可以在帐户中创建任意数量的容器。

除了包含对象之外，还可以使用容器通过使用访问控制列表（ACL）来控制对对象的访问。不能使用单个对象存储ACL。

在容器级别配置和控制许多其他功能，例如对象版本控制。

可以在一个请求中批量删除多达10,000个容器。

可以在具有云提供商的预定义名称和定义的容器上设置存储策略。

## Object Server

存储数据内容，例如文档，图像等。还可以使用对象存储自定义元数据。

使用Object Storage API，

存储无限数量的对象。每个对象可以大到5 GB，这是默认值。可以配置最大对象大小。

1. 使用大对象创建上载和存储任何大小的对象。
2. 使用跨源资源共享来管理对象安全性。
3. 使用内容编码元数据压缩文件。
4. 使用内容处置元数据覆盖对象的浏览器行为。
5. 安排删除对象。
6. 在单个请求中批量删除最多10,000个对象。
7. 自动提取存档文件。
8. 生成一个URL，该URL提供对对象的时间限制GET访问。
9. 使用表单POST中间件从浏览器直接将对象上传到Object Storage系统。
10. 创建指向其他对象的符号链接。

# 实验环境

## 系统环境.

OS:Manjaro 18.0.4 Illyria

Kernel: x86\_64 Linux 5.0.18-1-MANJARO

Shell: zsh 5.7.1

Docker:18.09.6-ce

Swift: Docker Version(Swift All In One)

Cosbench:4.1/4.2

Storage:HDD/SSD

## Minio-Server

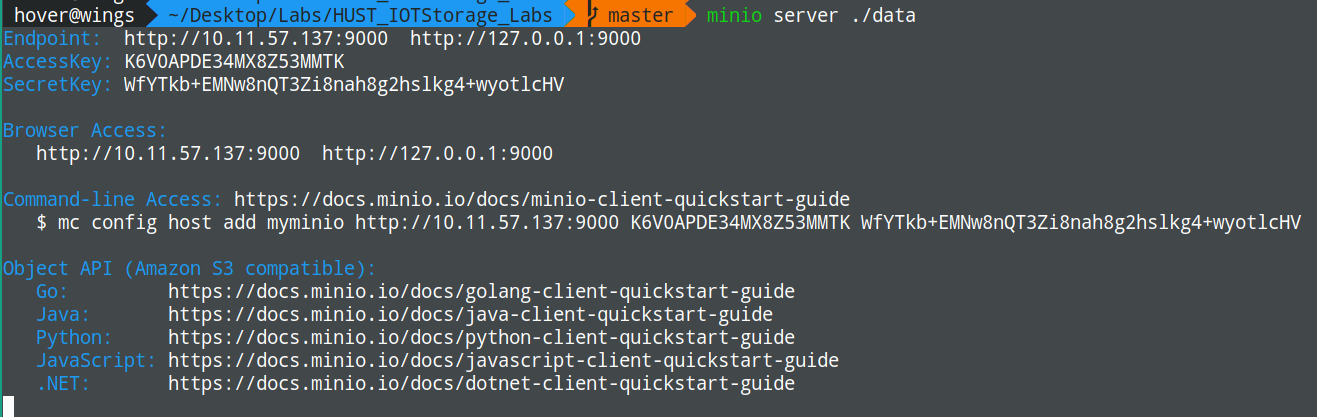


图3.2 Minio-Server启动验证



图3.3 Minio Host添加

此时在浏览器打开127.0.0.1:9000(endpoint)，在打开的minio browser，输入自己的AccessKey和SerectKey登陆。

此时可以选择右下角红色标记随意添加bucket和上传文件，方便地实现类似网盘的功能。

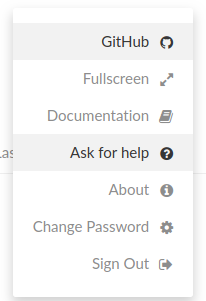


图3.4 账户管理相关选项

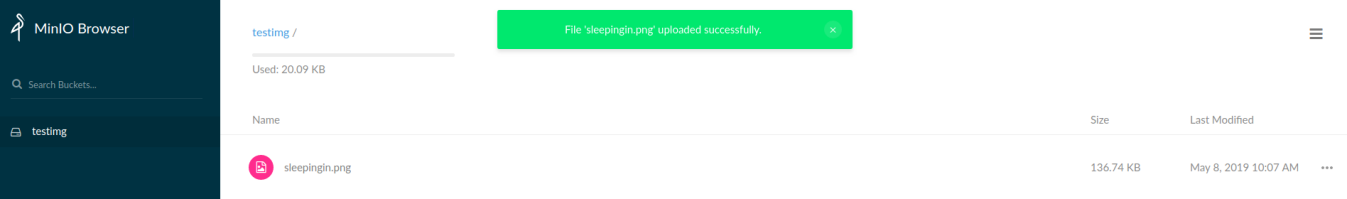


图3.5 Bucket/file测试图

注意，bucket有相应的命名限制，只能采用小写字母，而不允许下划线和相应大写字母的出现。

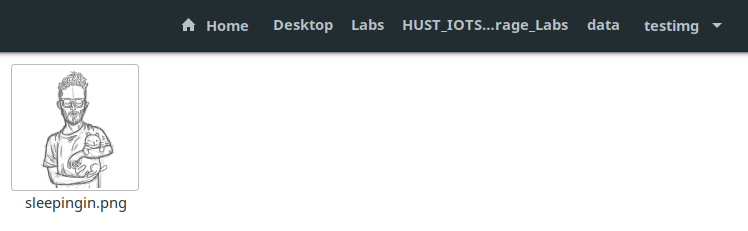


图3.6 本地文件验证

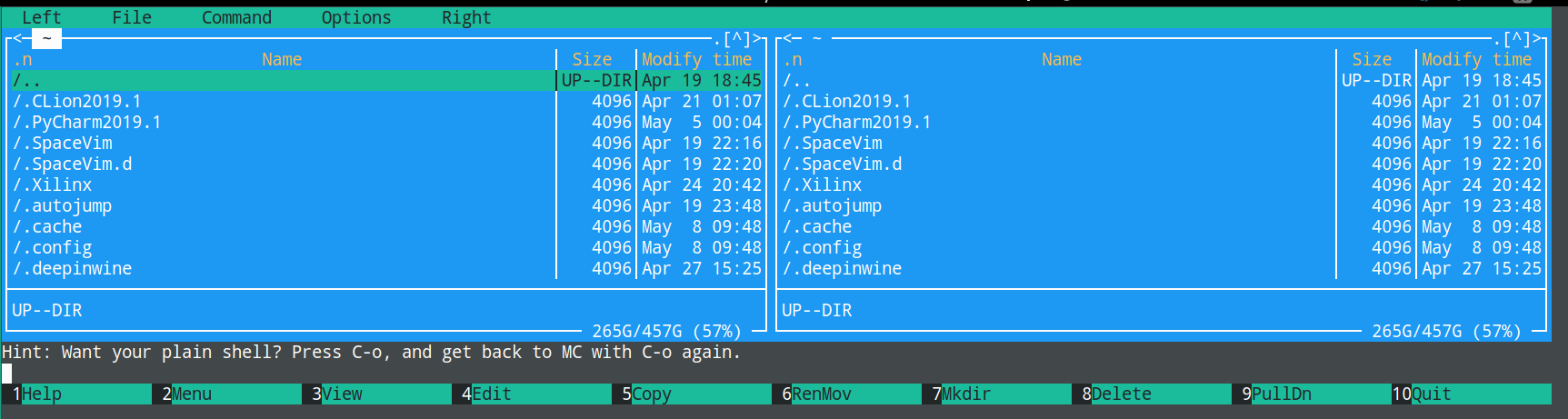


图3.7 MC界面

## Minio-python API

使用python-tutorial 进行API编程测试，使用官方测试仓库，代码如下

代码3.1 Minio-python API

from minio import Minio

from minio.error import ResponseError

from minio.error import (ResponseError, BucketAlreadyOwnedByYou,

                         BucketAlreadyExists)

minioClient = Minio('play.min.io:9000',

                  access\_key='Q3AM3UQ867SPQQA43P2F',

                  secret\_key='zuf+tfteSlswRu7BJ86wekitnifILbZam1KYY3TG',

                  secure=True)

# Make a bucket with the make\_bucket API call.

**try**:

       minioClient.make\_bucket("maylogs", location="us-east-1")

except BucketAlreadyOwnedByYou as err:

       pass

except BucketAlreadyExists as err:

       pass

except ResponseError as err:

**raise**

**else**:

        # Put an object 'pumaserver\_debug.log' with contents from 'pumaserver\_debug.log'.

**try**:

               minioClient.fput\_object('maylogs', 'pumaserver\_debug.log', '/tmp/pumaserver\_debug.log')

        except ResponseError as err:

               print(err)

运行结果如下

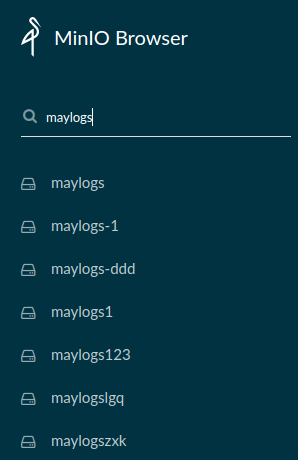


图3.8 Minio-python 测试结果

## AWS-CLI

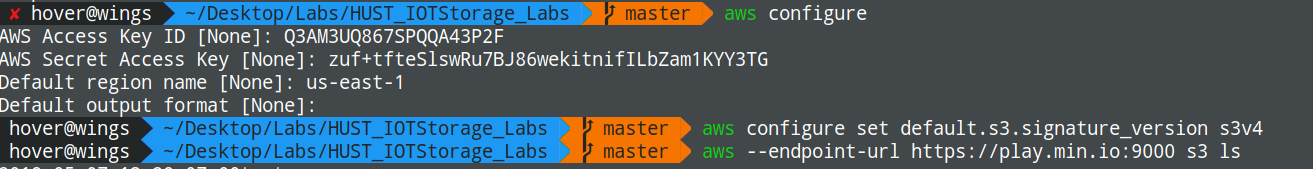


图3.9 使用官方测试AWS-CLI测试

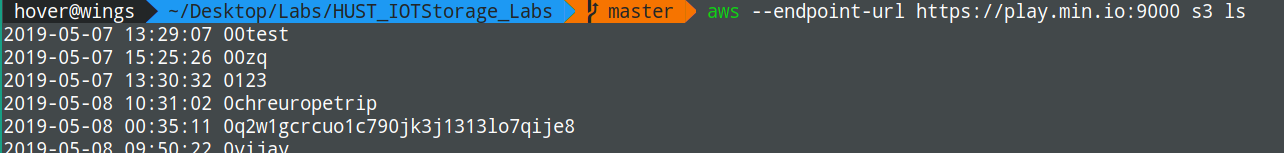


图3.10 此存储下的bucket



图3.11 创建bucket

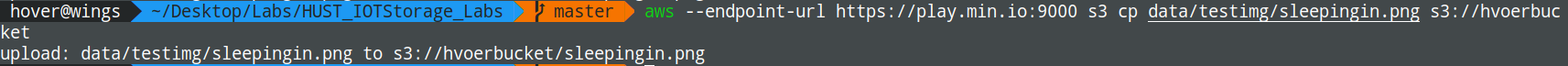


图3.12 上载IMG测试



图3.13 删除IMG测试

## s3bench

使用如下命令进行测试，其中参数可自行调节

go run ./s3bench.go -accessKey=KEY -accessSecret=SECRET -bucket=loadgen -endpoint=http://endpoint1:80,http://endpoint2:80 -numClients=2 -numSamples=10 -objectNamePrefix=loadgen -objectSize=1024

#!/bin/bash

**for** i in {1..10};**do**

     ~/go/bin/s3-benchmark -a 4VFY6G1OGS0VU5ELE6FL -s e4fWg5uC9BSfrrL6zmuWV4x42MQpPtPjW2Isq3gW -u http://127.0.0.1:9000 -b wasabi-benchmark \

    -d 3 -t i -z 1K | tail -1 -al > result.csv

done

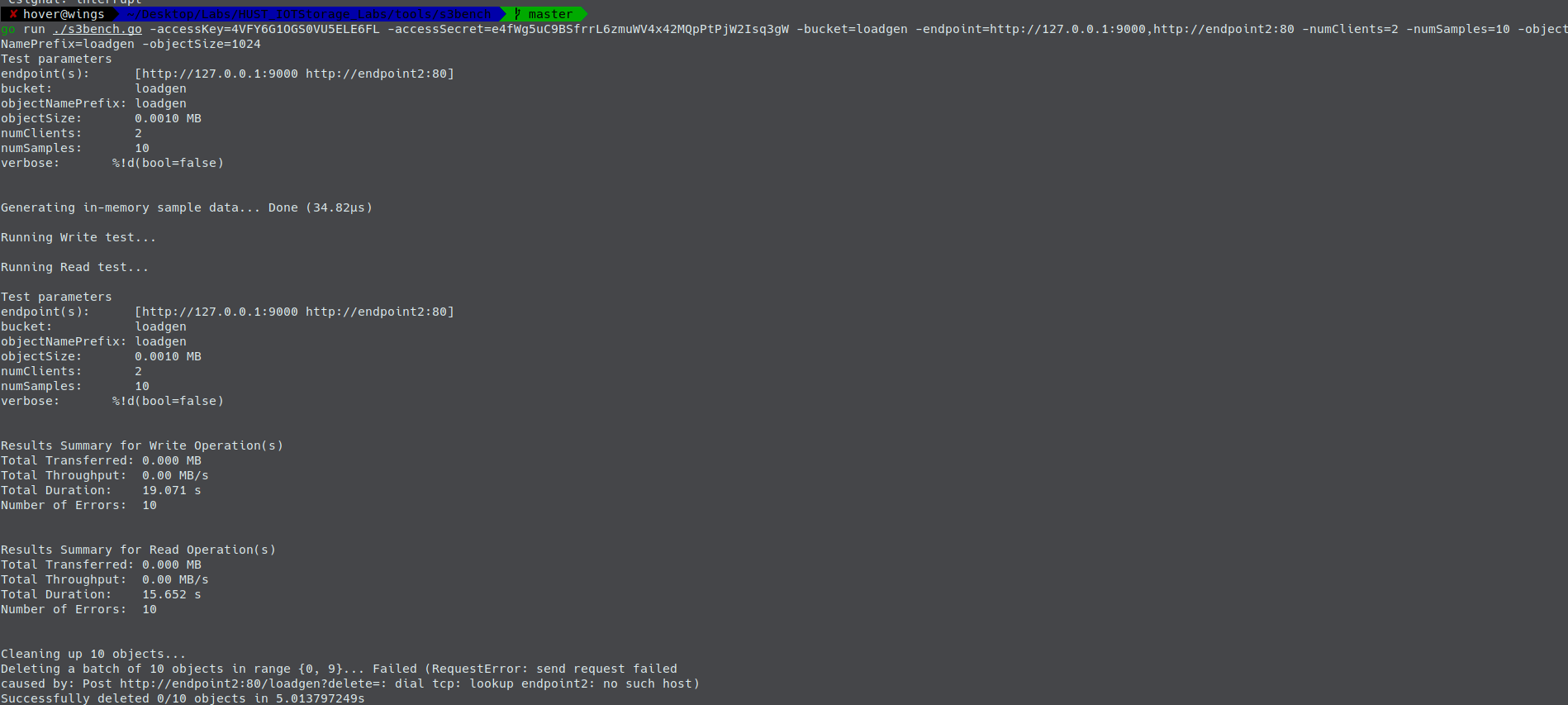


图3.14 s3bench测试

## CosBench测试

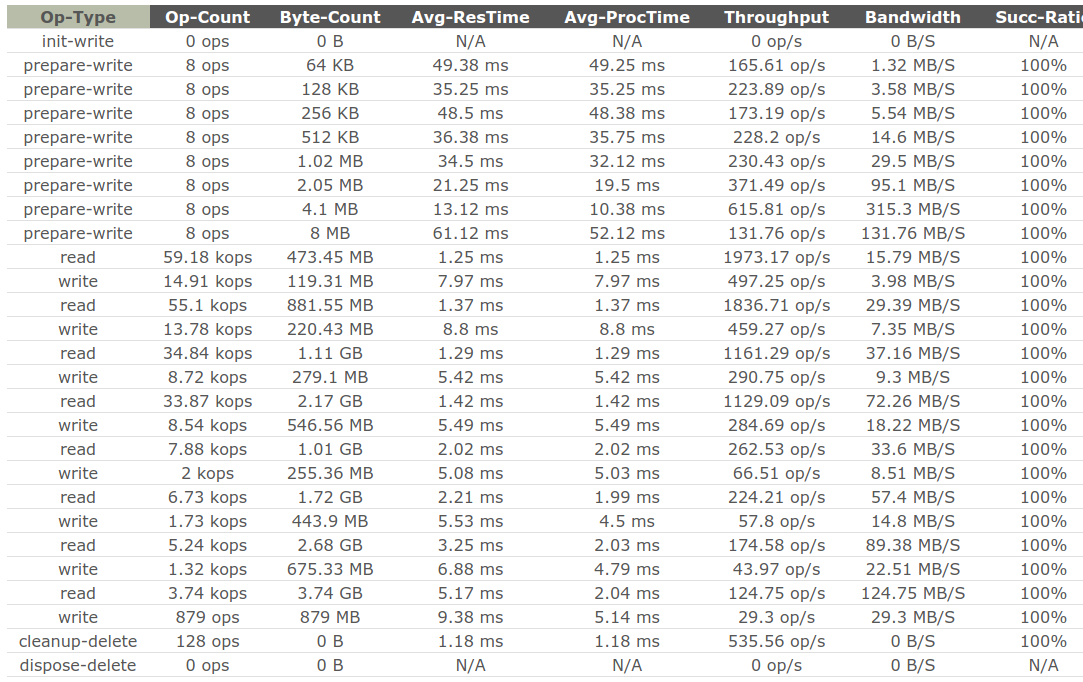


图3.15 cosbench测试

## OpenStack Swift CLI

使用OpenStack Swift进行相关测试

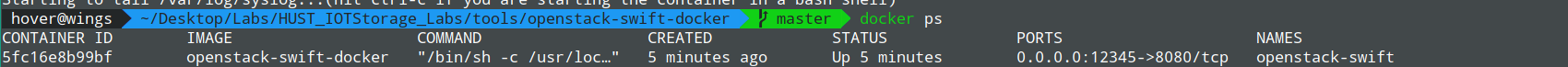


图3.16 Swift Docker启动

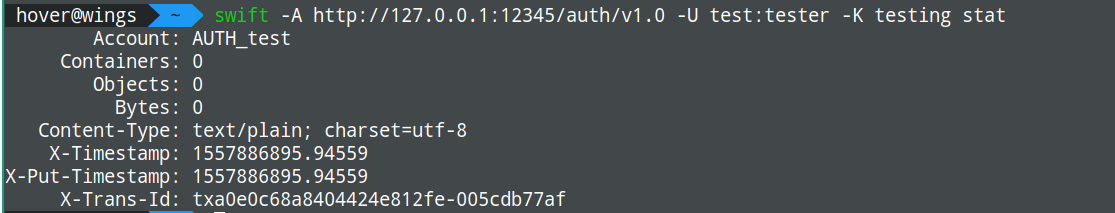


图3.17 Swift Docker启动检测

设置系统环境变量，避免每次命令行需要输入，由于采用V1.0验证，具有和V2.0不同的环境变量配置，需要注意。

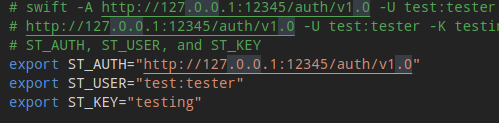


图3.18 SwiftAUTH环境变量设置

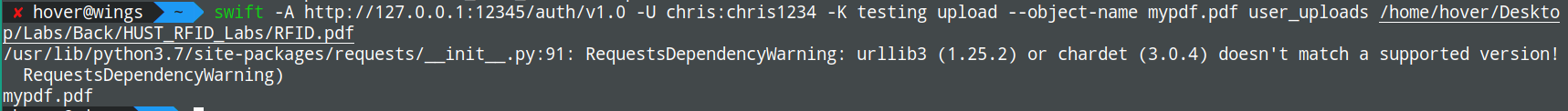


图3.19 Swift上传测试

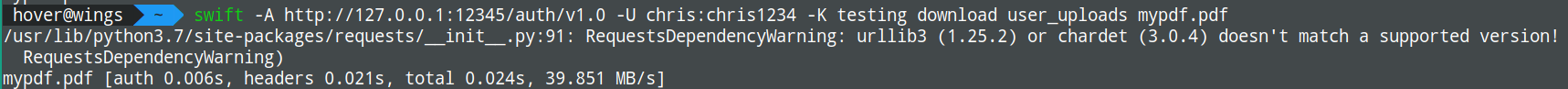


图3.20 Swift下载测试

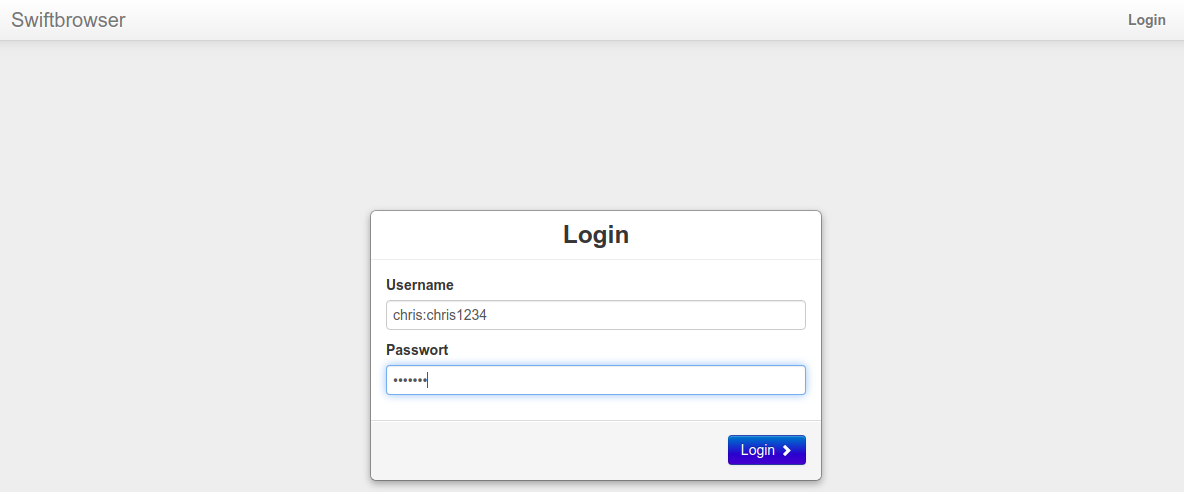


图3.21 Swift GUI-Client 测试

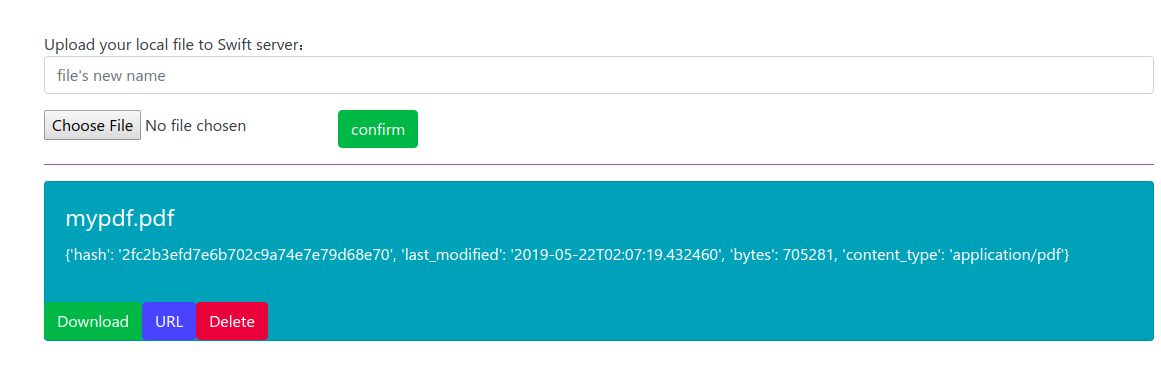


图3.22 Swift GUI测试

## OpenStack Swift Python-API

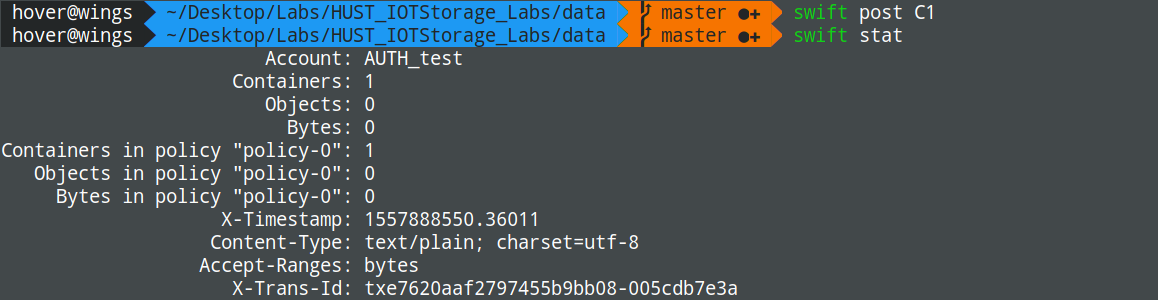


图3.23 Swift创建Container

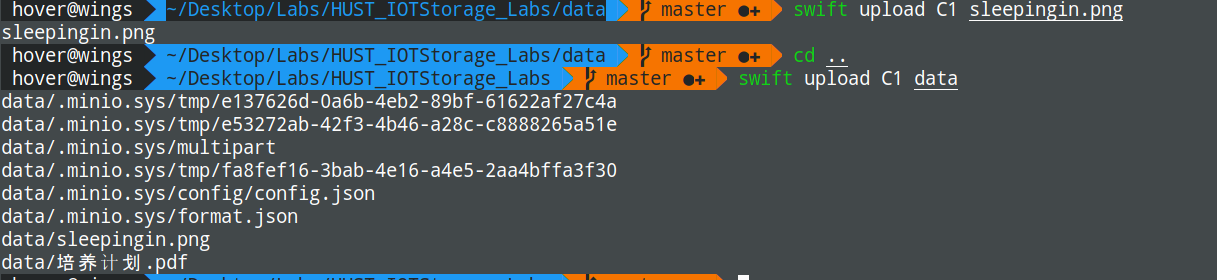


图3.24 Swift上传文件和文件夹测试

# 实验内容

## [对象存储技术实践](#_Toc509412099)

1. 采用Docker进行环境相关配置
2. 相关API测试以及测试程序编写
3. 测试数据下载分析

## 对[象存储性能分析](#_Toc509412100)

1. 对于不同情况下的测试结果进行绘图
2. 测试结果分析

# 实验过程

## 环境配置

1. 安装软件包

由于基于Arch系统优良的包管理和AUR，所需要的包和依赖可以一键安装。

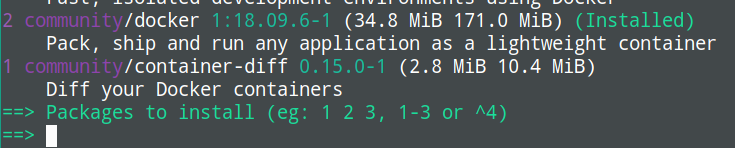


图5.1 yay安装相关包

1. 启动服务程序



图5.2 启动Docker服务程序

由于Docker服务程序的性能占用，故采用start，若希望开机启动，可以使用enable

1. 运行相应的Docker环境
   1. Swift

采用swift all in one 进行启动，由于arch系的支持原因，Open Stack系列并未支持swift，导致无官方包和相应AUR。

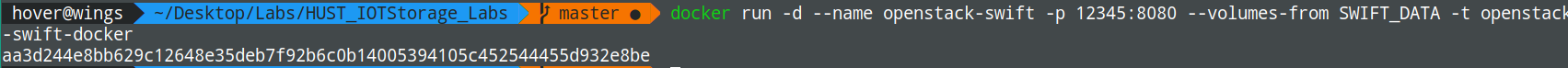


图5.3 swift docker run

* 1. Cosbench

由于Cosbench存在较严重的问题，不同的docker进行了尝试，也尝试对于docker内的问题进行修复。

其中：

* + 1. Cosbench启动失败

在不同的环境上均遇到了此问题，推测JAVA Version相关，docker内出现符号链接错误。

* + 1. 特定的workload启动失败

在8线程的时候出现失败，本地为四核八线程CPU，但docker中详细的设定可能会有差异，推测与docker有关。

## 基于Cosbench进行单变量测试

### Worker数量影响

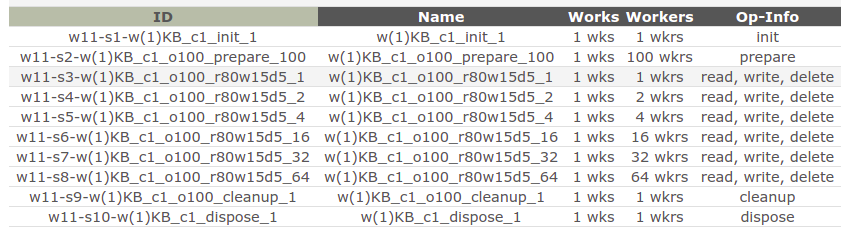


图5.4 workload配置-Workers变量

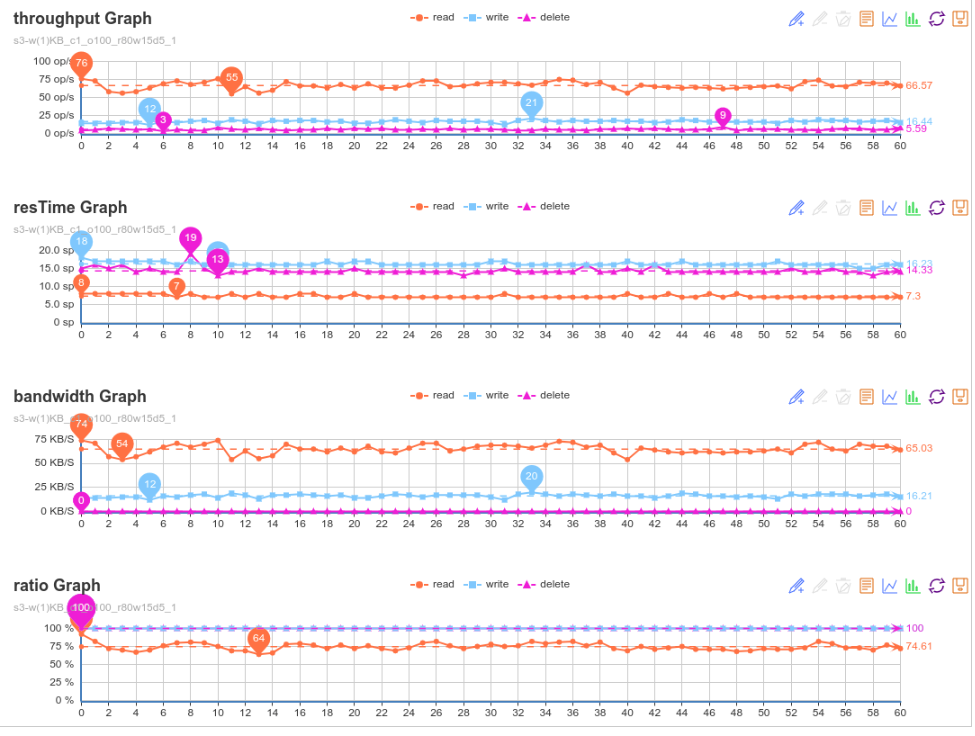


图5.5 workers变量改变结果

可以看到，在workers数量提升之后，整个曲线更加的平滑，同时各项指标有了相应的提升，读写吞吐率和带宽等都有明显增加，但是限于硬件的支持，多worker的上限与CPU线程数有关，并发度不可能无限提高，同时对于不同大小的文件，需要考虑创建/销毁worker，以及相应的调度开销。

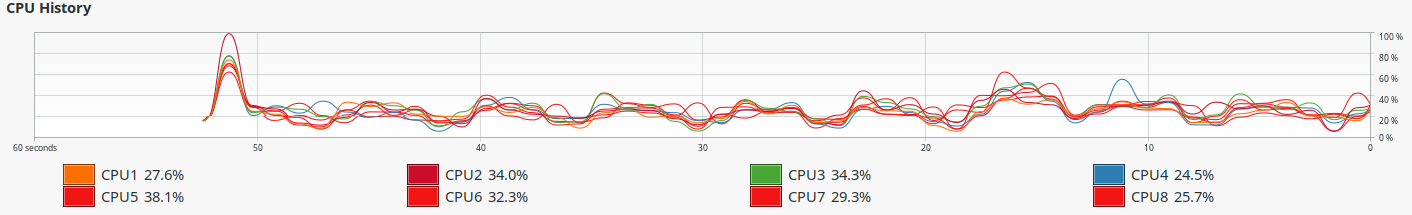


图5.6 CPU利用率曲线

此时可以看到，相对于通常的一核有难，七核围观的情况，核心之间的负载均衡做的较好。

### OBJ大小测试

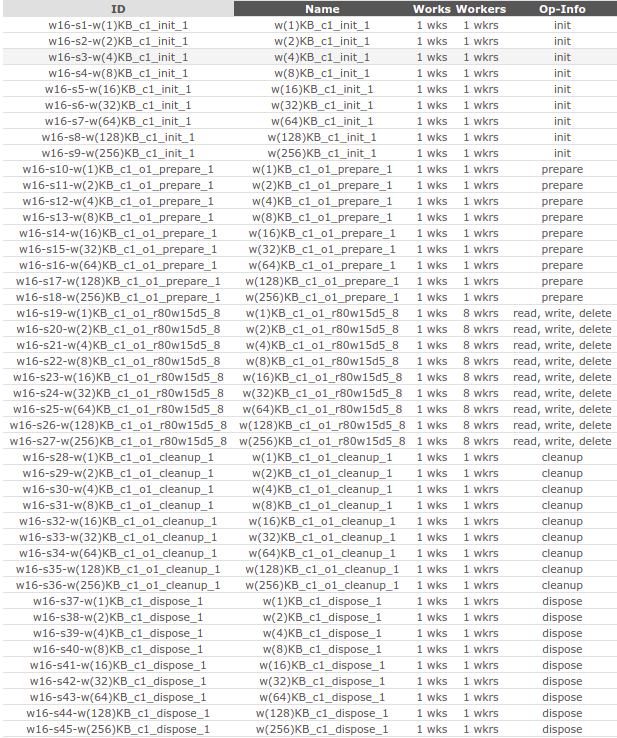


图5.7 OBJ Size测试

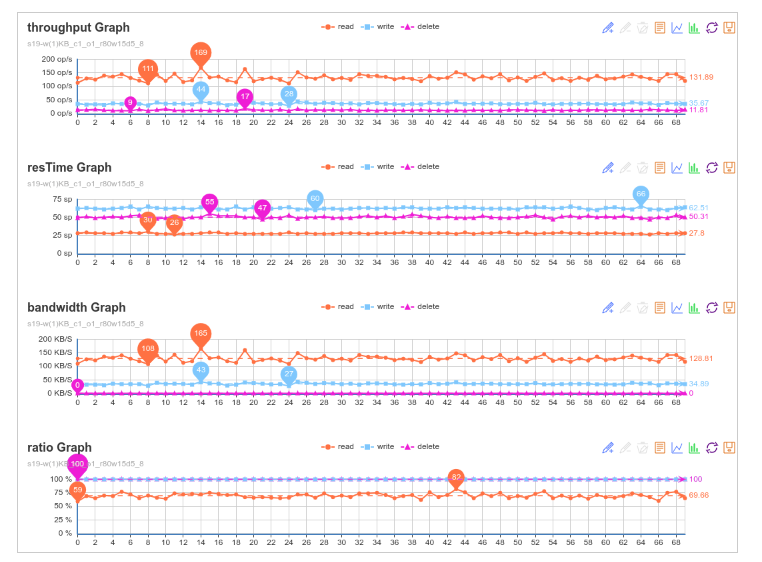


图5.8 OBJ Size测试图

当对象大小逐渐增大时，各项性能变优，判断OBS对于大文件的支持较好，基于此特性，在进行对象传输的过程中，存在相应的封装，对于minio等较为轻型的OBS，未提供或者封装较为简单，很多情况下传输的为裸文件。而Swift存在相应封装，对于小obj传输，其寻址，读取开销，封装开销严重影响吞吐率和带宽。同时，此时并未考虑相应的加解密开销，如果指定相应的加解密算法，差距将进一步拉大。

### Container数量测试

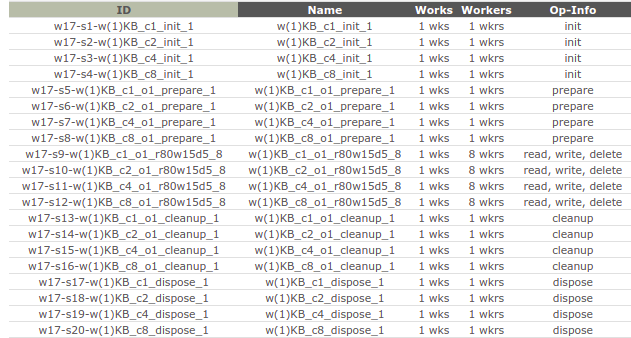


图5.9 Container Number测试

Container Number增加时，各项数据变化不明显，由于Container之间可以做冗余或者多节点存储，同步开销将成为主要的部分，此处查阅文档得知，Swift支持定期同步，且可以设置同步授权，此时，Container数量将会影响到多节点同步。

## 进行HDD/SSD性能比较

由于本机采用 SSD，而NAS之上采用HDD,在实际过程中，对于非时间敏感型数据的读取通常采用HDD存储，Swift docker形式封装不易指定相应的存储位置，采用Minio在不同分的地方启动服务，探究HDD/SSD的性能差异。

### SSD测试

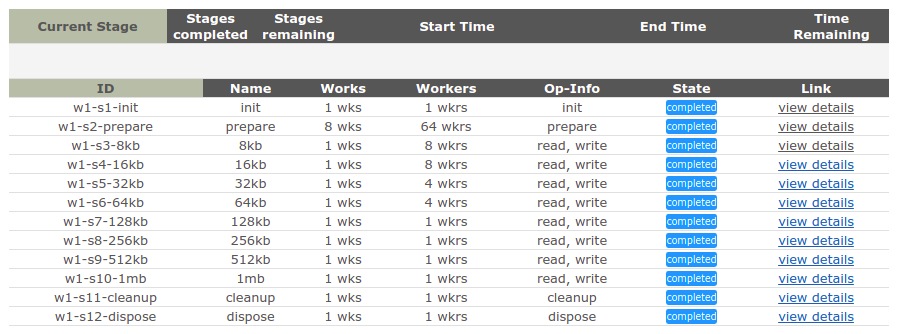


图5.10 测试配置

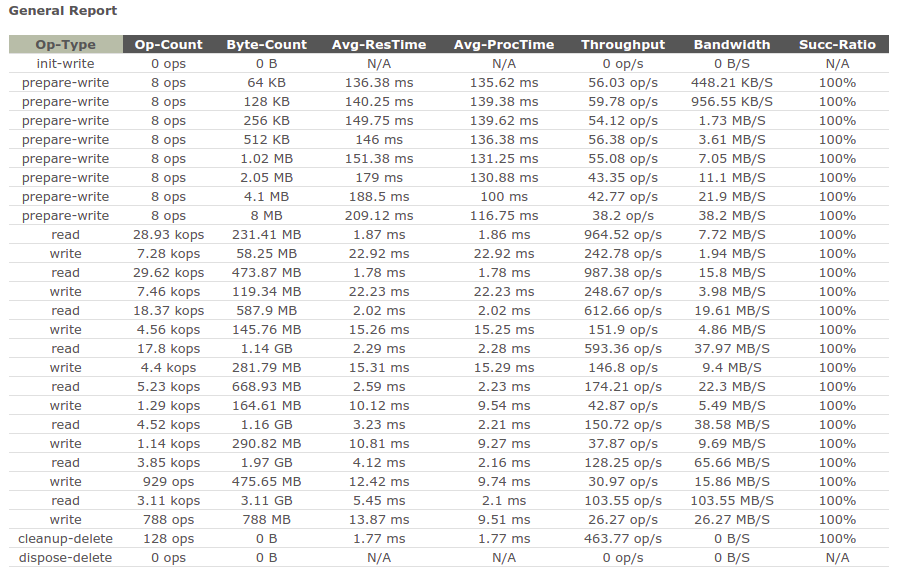
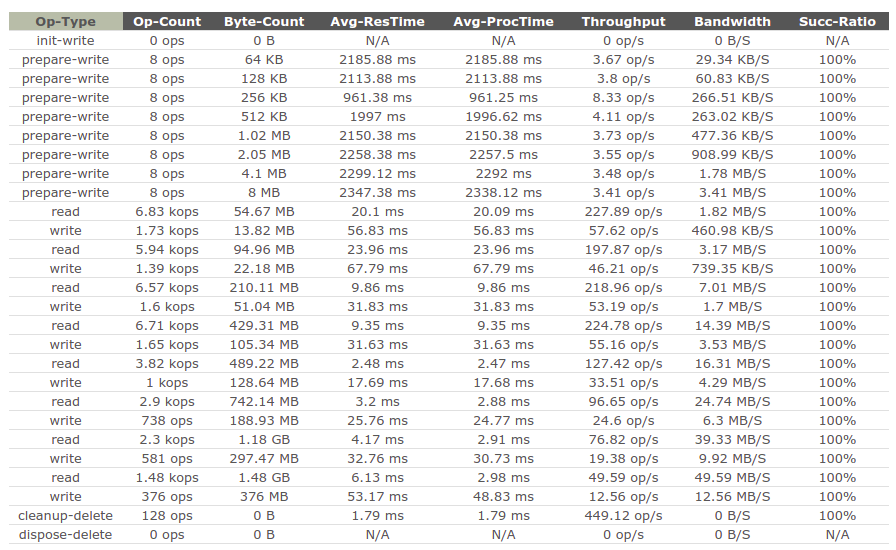


图5.11 SSD测试结果

### HDD测试



由上表可以看出，在读取方面，约有五倍的性能提升，在写方面约5倍的性能提升，同时在准备的时间大大缩短。而在较大文件的读取时，退化到约2倍，可以看到在顺序读取时SSD的性能优势不如随机读取时明显，同时worker数量对于SSD的性能提升远远大于HDD,限于HDD的寻道操作，并行提升并不明显，更适用于顺序读取。

# 实验总结

在实验过程中，环境配置可以说相对来说比较简单，但是某些玄学BUG可能无明显报错信息，转而转向Docker，由于所有的包括服务端和测试端均在Docker中启动，存在部分Automated Dcoker入口写死，需要自己手动更改Docker文件进行相应的编译，在此过程中提高了Docker的熟练程度。

实验过程中也体会到了实际工业环境与Toy之间的区别，对于swift，通常采用Obj Server,Container Server,Account Server三个服务器进行分别管理，处理调度分配，实际工作执行。而Docker封装all in one则仅仅使用端口号进行区分，实际上存在Container Server,Account Server等多个docker以进行配合，且单机模式无法判断带宽影响，同hadoop等分布式平台类似，大多数实验环境基于本地虚拟机，在云服务平台上，大多采用的是硬件虚拟化之后再使用多个docker模拟集群，与实际的物理存储环境可能存在差异。

测试方面，由于常规使用本地NAS做存储，使用花生壳做内网穿透，故相应的测试也在本地进行，并进行了HDD/SSD进行相应的比较。

最后，感谢实验过程中提供帮助的老师和同学以及所参阅资料的提供者。

# 参考文献

1. ARNOLD J. OpenStack Swift[M]. O’Reilly Media, 2014.
2. ZHENG Q, CHEN H, WANG Y等. COSBench: A Benchmark Tool for Cloud Object Storage Services[C]//2012 IEEE Fifth International Conference on Cloud Computing. 2012: 998–999.
3. WEIL S A, BRANDT S A, MILLER E L等. Ceph: A Scalable, High-performance Distributed File System[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2006: 307–320.
4. Swift Document.URL:[https://docs.openstack.org/swift](https://docs.openstack.org/swift/latest/container.html)