華中科技大學

数据中心技术实验报告

姓名 郑得华

学号 M202173785

目录

一. 实验背景介绍

1. 对象存储服务

对象存储是在文件存储的基础上发展而来的，对象存储抛弃了文件存储的命名空间、文件目录等结构。对象存储相比文件存储更加简洁，抛弃了命名空间、文件目录等结构，更加扁平化，在使用、扩展、维护方面更加符合大众化思想。操作主体由文件变为对象，对象的操作主要以 Put、Get 和 Delete 为主，十分类似 Java 的 HashMap，只不过对象存储中的对象不支持修改。总体来说，对象存储是为了克服文件存储的缺点，并发挥文件存储的优点而出现的。但对于存储内容而言，文件存储和对象存储并无本质区别，只是存储方式发生了变化。

2. OBS 的发展历程

对象存储的标准最初来自于学术科研的领域，美国的卡内基梅隆大学有个并行数据实验室，1995 年建立了一个叫 network attached secure disk 的项目，在这个项目中首次提出了对象存储的概念。随着项目不断往前发展，在更大范围里得到了认可。1999 年 SNIA（网络存储工业协会）成立了一个叫做 OSD（对象存储设备）的工作组，这个工作组发布了 ANSI 的 x3 T10 的标准，这就是对象存储的原型。

5 年后的 2004 年 SNIA 正式发布了 OSD1.0 的标准，第一波对象存储的技术潮流由此涌现，一个典型的从学术走向工业界的过程。在这第一波浪潮里，Oracle 推出了一个标准实现名为 Lusture，相信大学和科研机构的人员对此较为熟悉，它在科学计算等领域得到较多广泛应用，可谓第一代的知名对象存储。

而对象存储的的第二波技术热潮得力于亚马逊，2006 年 AWS 正式上线，S3 对象存储是其推出的第一个云服务，随着云计算的全球热潮，对象存储的知名度得到了更进一步的大发展，相信很多人是从 S3 这里第一次知晓了对象存储的存在。

第三次技术热潮从我看来和 OpenStack 的兴起有很大关系。 OpenStack 项目的最早只有两大组件 Nova 和 Swift，其中 NASA 贡献出来的 Nova 是解决虚拟化的问题，而另一个 Swift 项目就是由 Rackspace 公司贡献出来的对象存储。在 2012 前后，OpenStack 里面出现了一个明星项目 Ceph，相信到今天大家已经耳熟能详。这个统一存储项目刚开始吸引大家眼球的特点是它可以提供分布式的块存储，但没想到 8 年过去，使用最多最流行的反而是它的对象存储功能。而走在前列的专业对象存储公司 SwiftStack 公司也在今年出现了被 NVidia 收购的新闻，事情的发展真是难以预料。

回到国内来看，2018 年开始，对象存储开始出现了一波热潮：不仅市场上出现了非常多的对象存储公司，而且传统的 IT 大厂商也纷纷推出了自己的对象存储产品，同时大量的用户在不同的行业里开始采用对象存储方案，一直到 2021 年的今天，对象存储仍然很受关注。

二. 实验动机与学习基础

1. 实验动机和目的

(1) 熟悉性能指标：吞吐率、带宽、延迟
(2) 分析不同负载下的指标、延迟的分布
(3) 观测尾延迟现象
(4) 尝试对冲请求方案

2. 实验环境

配置运行对象存储系统 minio，兼容 AWS S3 协议;
配置 linux 系统下 go 开发环境编译 s3bench;
配置 linux 系统下 Rust 开发环境运行 s3-bench-rs;
配置 python 环境进行尾延迟的收集和可视化展示。

三. 技术实现方案

1. Minio

Minio 是个基于 Golang 编写的开源对象存储套件，基于 Apache License v2.0 开源协议，虽然轻量，却拥有着不错的性能。它兼容亚马逊 S3 云存储服务接口。可以很简单的和其他应用结合使用，例如 NodeJS、Redis、MySQL 等。MinIO 的应用场景除了可以作为私有云的对象存储服务来使用，也可以作为云对象存储的网关层，无缝对接 Amazon S3 或者 MicroSoft Azure

在本实验中我们使用 minio 提供对象存储服务，分别使用 LINUX 和 windows 版本测试对象存储服务的性能指标，主要包括吞吐率、带宽、延迟，并分析不同负载下的指标、延迟的分布，观测尾延迟现象。

(1) Windows 模式

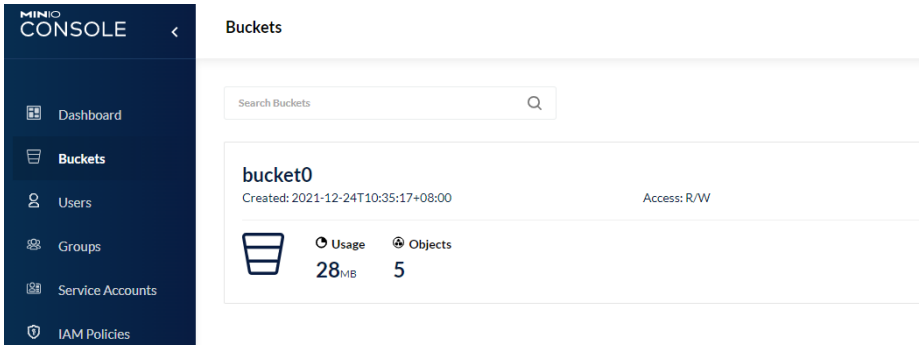
```
PS> setx MINIO_ROOT_USER minioadmin                                    设置访问用户名
PS> setx MINIO_ROOT_PASSWORD minioadmin                                 设置访问密码
PS> C:\minio.exe server F:\Data --console-address ":9001" 启动服务端
```

```
C:\Users\dward>minio.exe server F:\Data --console-address ":9001"
API: http://10.11.68.185:9000  http://127.0.0.1:9000
RootUser: minioadmin
RootPass: minioadmin

Console: http://10.11.68.185:9001 http://127.0.0.1:9001
RootUser: minioadmin
RootPass: minioadmin

Command-line: https://docs.min.io/docs/minio-client-quickstart-guide
   $ mc.exe alias set myminio http://10.11.68.185:9000 minioadmin minioadmin

Documentation: https://docs.min.io
```



(2)LINUX 模式

```
dpkg -i minio_20211220220716.0.0_amd64.deb
MINIO_ROOT_USER=admin  MINIO_ROOT_PASSWORD=password  minio  server /mnt/data --console-address ":9001"
```

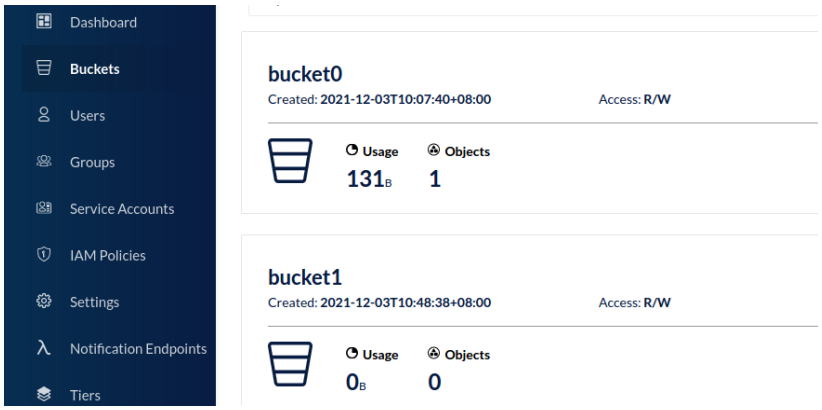
```
You are running an older version of MinIO released 3 weeks ago
Update: Run `mc admin update`

API: http://10.11.70.229:9000  http://172.17.0.1:9000  http://127.0.0.1:9000
RootUser: admin
RootPass: password

Console: http://10.11.70.229:9001 http://172.17.0.1:9001 http://127.0.0.1:9001
RootUser: admin
RootPass: password

Command-line: https://docs.min.io/docs/minio-client-quickstart-guide
   $ mc alias set myminio http://10.11.70.229:9000 admin password

Documentation: https://docs.min.io
```



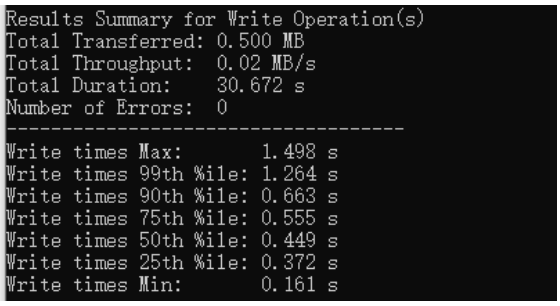
2. 运行 s3bench 测试相关性能指标

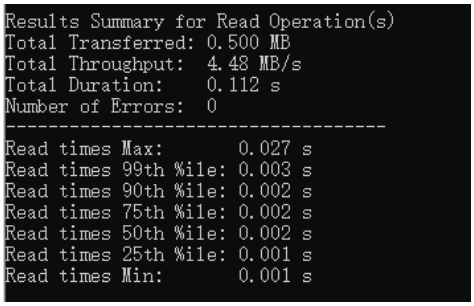
```
go get -u github.com/igneous-systems/s3bench
```

运行以上指令可以在 go 环境下编译 s3bench

```
s3bench \
  -accessKey= minioadmin -accessSecret=minioadmin \
  -endpoint=http://127.0.0.1:9000 \
  -bucket=loadgen -objectNamePrefix=loadgen \
  -numClients=10 -numSamples=100 -objectSize=1024
```

在打开 minio 服务端的基础上运行以上测试指令，可以得到如下结果：





四. 实验结果

在本实验中，我们主要测试的指标有：吞吐率 Throughput、延迟 Latency，以及环境参数：对象尺寸 object size、并发性、服务器数量。

1. 对象大小对测试指标的影响

这时我们固定 numClient=8,numSamples=256，单独改变对象的尺寸大小来观察性能指标测量值的变化。

	Operations			
	Write		Read	
测试指标	objectSize=1024	objectSize=20480	objectSize=1024	objectSize=20480
Total Transferred	0.250 MB	5.000MB	0.250 MB	5.000MB
Total Throughput	0.02 MB/s	0.36MB/s	4.29 MB/s	72.89MB/s
Total Duration	14.259s	13.788S	0.058s	0.069s
Number of Errors	0	0	0	0
times Max	0.856s	1.293s	0.004s	0.009s
times 99th %ile	0.808s	1.274s	0.004s	0.004s
times 90th %ile	0.686s	0.574s	0.002s	0.003s
times 75th %ile	0.534s	0.434s	0.002s	0.002s
times 50th %ile	0.410s	0.383s	0.002s	0.002s
times 25th %ile	0.339s	0.310s	0.001s	0.002s
times Mln %ile	0.168s	0.131s	0.001s	0.001s
Generating sample	objectSize=1024	4.9827ms	objectSize=20480	5.0129ms
Delete objects		504.382ms		491.6988ms

2. 客户端数量对测试指标的影响

这时我们固定 objectSize=1024,numSamples=256，单独改变客户端数量来观察性能指标测量值的变化。

	Operations			
	Write		Read	
测试指标	numClient =8	numClient =16	numClient =8	numClient =16
Total Transferred	0.250 MB	0.250 MB	0.250 MB	0.250 MB
Total Throughput	0.02 MB/s	0.01 MB/s	4.29 MB/s	3.62 MB/s
Total Duration	14.259s	17.333s	0.058s	0.069s
Number of Errors	0	0	0	0
times Max	0.856s	3.500s	0.004s	0.012s
times 99th %ile	0.808s	3.182s	0.004s	0.010s
times 90th %ile	0.686s	2.279s	0.002s	0.007s
times 75th %ile	0.534s	1.113s	0.002s	0.005s
times 50th %ile	0.410s	0.791s	0.002s	0.004s
times 25th %ile	0.339s	0.659s	0.001s	0.003s
times MIn %ile	0.168s	0.307s	0.001s	0.001s
Generating sample	numClient =8	4.9827ms	numClient =16	3.0161ms
Delete objects		504.382ms		681.9731ms

3. numSamples 对测试指标的影响

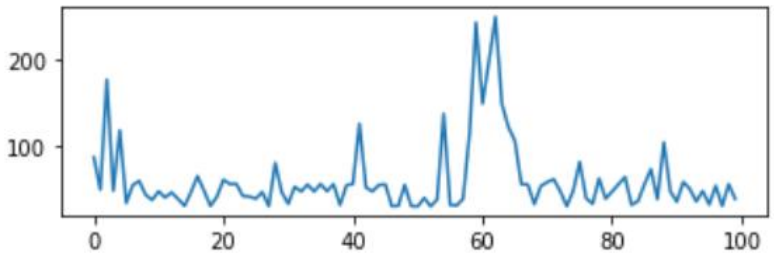
这时我们固定 objectSize=1024, numClient =8，单独改变 numSamples 大小来观察性能指标测量值的变化。

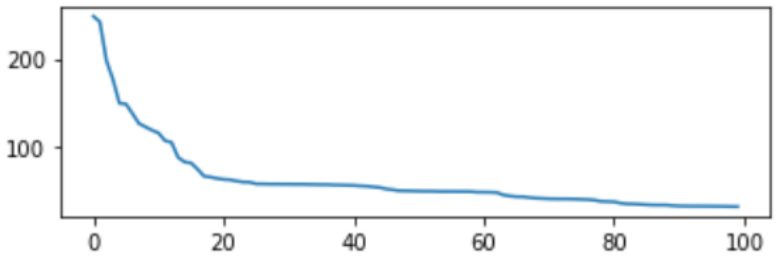
	Operations			
	Write		Read	
测试指标	numSamples=256	numSamples=512	numSamples=256	numSamples=512
Total Transferred	0.250 MB	0.500MB	0.250 MB	0.500MB
Total Throughput	0.02 MB/s	0.02 MB/s	4.29 MB/s	4.48 MB/s
Total Duration	14.259s	30.672s	0.058s	0.112s
Number of Errors	0	0	0	0
times Max	0.856s	1.498s	0.004s	0.027s
times 99th %ile	0.808s	1.264s	0.004s	0.003s
times 90th %ile	0.686s	0.663s	0.002s	0.002s
times 75th %ile	0.534s	0.555s	0.002s	0.002s
times 50th %ile	0.410s	0.449s	0.002s	0.002s
times 25th %ile	0.339s	0.372s	0.001s	0.001s
times MIn %ile	0.168s	0.161s	0.001s	0.001s
Generating sample	numSamples=256	4.9827ms	numSamples=512	2.9919ms
Delete objects		504.382ms		988.4894ms

4. 收集尾延迟并观察其分布

(1)延迟数据分布观察

在本实验中，按照不同的请求到达率设置不同的负载，首先发起请求并计算系统的停留时间，正确的完成请求后采集延迟，将采集到的延迟统一记录到文件中保存。并针对原始的延迟数据和排序后的延迟数据来观察延迟的分布情况，具体如以下两图所示：



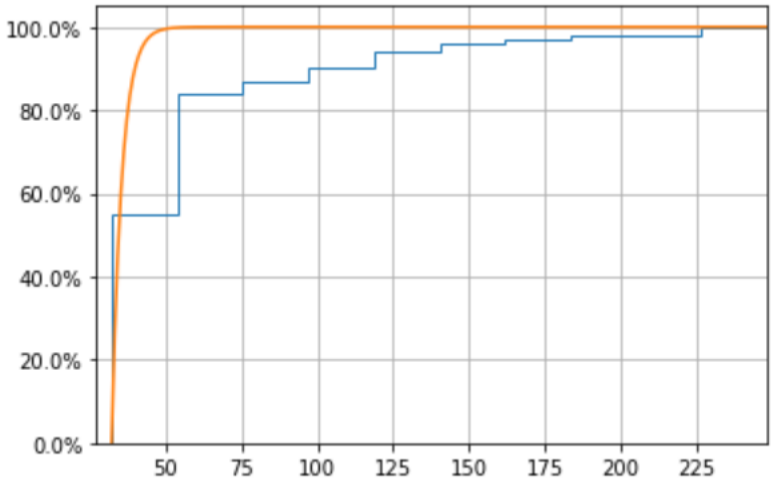


(2) 用排队论模型来拟合实测数据

排队论模型：

$$F(t) = 1 - e^{\alpha(\min(latency)-t)}$$

绘制结果如下图所示：



5. 基于 rusty-s3 和 tokio 进行 AWS S3 对象存储服务性能测试

运行项目运行 GET 和 PUT 测试，并输出运行时间和分析**尾延迟。**

(1)参数解释

time	每轮测试运行的时间，左边的是最小值，右边的是最大值，中间的是所有运行时间的最佳估计
change	相比上次测试的变化值（在该实验中可忽略）
outliers	离群值，表示该结果的值和其他结果相差较大
low severe	严重的低离群值
low mild	轻微的低离群值
high severe	严重的高离群值
high mild	轻微的高离群值

(2)Async GetObject

time	[19.668ms 20.717ms 21.895ms]
change	[-94.783% -89.789% -60.694%](p=0.01<0.05)
3	(3.00%) low mild
2	(2.00%) high mild
2	(2.00%) high severe

(3)Async PutObject

time	[19.229ms 19.827ms 20.523ms]
change	[-95.688% -92.127% -79.479%](p=0.01<0.05)
1	(1.00%) high mild
3	(3.00%) high severe

五． 实验总结

1. 改变负载参数对于测试指标的影响

在实验中我们分别改变 objectSize，numSamples 和 numClients 三个参数，维持其他参数不变，改变 objectSize,可知 transferred 参数成比例变化，总的吞吐量随着 objectSize 的增加有着明显的增加,而 duration 参数却变化不太明显，对于读写操作测试，最大延迟时间随 ObjectSize 的增加而有所提升。当我们只增加客户端的数量时，transferred 指标不变，总的吞吐量有所下降，延迟时间有着明显的增加,对于读写操作来说,各分位的延迟时间均有着明显的增加。同理。当我们只增加 numSamples 时，transferred 指标成比例变化，总的吞吐量几乎没有变化，延迟时间有着成倍增加的现象，对于读写操作，最大延迟时间有着明显的增加，其他分位的延迟时间变化并不明显。

2. 尾延迟的分布情况

经过实验 4 的数据可视化展示，大部分的延迟时间分布在中等区间，极大极小值占极少部分，总的延迟时间更加趋于中间值。

3. 基于 rusty-s3 和 tokio 的对象存储服务性能分析

根据测试数据可知，进行对象的 Get 操作时，最少运行时间为 19.688ms，3%的操作产生轻微的低离群值，2%的操作产生轻微的高离群值，另有 2%的操作产生严重的高离群值。进行对象的 Put 操作时，最少运行时间为 19.229ms，1%的操作产生轻微的高离群值，另有 3%的操作产生严重的高离群值。

4. 心得体会

这门课以及实验操作涉及到了本人的很多知识盲区，但是探索才有收获，通过学习老师给出的实验教程以及参考资料，逐渐掌握了对象存储技术的相关知识，对于学习数据中心技术等有了很好的补充。