



2019 级

《物联网数据存储与管理》课程

实 验 报 告

姓 名 张佳辉

学 号 U201916341

班 号 校交 1902

日 期 2022.04.10

目 录

一、实验目的.....	1
二、实验背景.....	1
三、实验环境.....	1
四、实验内容.....	2
4.1 对象存储技术实践.....	2
4.2 对象存储性能分析.....	2
五、实验过程.....	3
5.1 搭建对象存储服务端.....	3
5.2 搭建对象存储客户端并测试对象存储功能.....	4
5.4 对象存储性能分析.....	5
六、实验总结.....	8
参考文献.....	9

一、实验目的

1. 熟悉对象存储技术；
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。

二、实验背景

随着移动互联网的兴起而蓬勃发展，我们日常生活产生的海量数据带来了存储挑战。

首先毋庸置疑，我们的数据量在当今这个时代可以说是非常庞大的，互联网的数据量已经从 TB 级跃升至 PB、EB 乃至 ZB 级，如今依旧成几何级数不断上升对存储和处理带来挑战。

其次，数据内容多样性带来的数据异构，使得数据结构越来越复杂，不能很好地存储并且管理这些数据。

因此，需要一个高可用、可扩展的海量数据存储系统来满足物联网存储需求。对象存储（Object Storage Service, OSS）应运而生，也叫基于对象的存储，是一种解决和处理离散单元的方法，可提供基于分布式系统之上的对象形式的数据存储服务。对象存储和我们经常接触到的块和文件系统等存储形态不同，它提供 RESTful API 数据读写接口及丰富的 SDK 接口，并且常以网络服务的形式提供数据的访问。简单理解，对象存储类似酒店的代客泊车。顾客（前端应用）把车钥匙交给服务生，换来一张收据（对象的标识符）。顾客不用关心车（数据）具体停在哪个车位，这样省事儿、省时间。

三、实验环境

CPU: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz

内存: 16.0 GB

硬盘: SSD 1T

操作系统: Window11

Nodejs: v16.13.2

四、实验内容

根据实验给出的教程，选择对象存储服务端、对象存储客户端、对象存储评测工具。

4.1 对象存储技术实践

1. 搭建对象存储服务端，选择使用 Minio。
2. 搭建对象存储客户端，选择使用 Minio Client。
3. 测试对象存储基本功能是否正常。

4.2 对象存储性能分析

1. 搭建对象评测工具环境，选择 S3 Bench。
2. 调整对象存储评测参数，包括客户端数量、对象大小，观察数据存储性能。
3. 编写 Nodejs 代码，处理结果。
4. 整理和分析不同参数下存储性能数据。

五、实验过程

5.1 搭建对象存储服务端

在 minio 官网下载 windows 版本地 minio.exe 程序，下载好之后，通过 run-minio.cmd 这个 window 命令脚本来启动服务。

```
C:\WINDOWS\system32\cmd.exe
Finished loading IAM sub-system (took 0.0s of 0.0s to load data).
API: http://192.168.1.32:9000 http://127.0.0.1:9000
RootUser: hust
RootPass: hust_obs

Console: http://192.168.1.32:9090 http://127.0.0.1:9090
RootUser: hust
RootPass: hust_obs

Command-line: https://docs.min.io/docs/minio-client-quickstart-guide
$ mc.exe alias set myminio http://192.168.1.32:9000 hust hust_obs

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

图 1 运行 minio 服务端

我们可以看到，minio 服务已经成功启动，接着我们访问 <http://localhost:9090/login>，可以看到 minio 自带的管理界面，输入用户名和密码，即可进入页面。

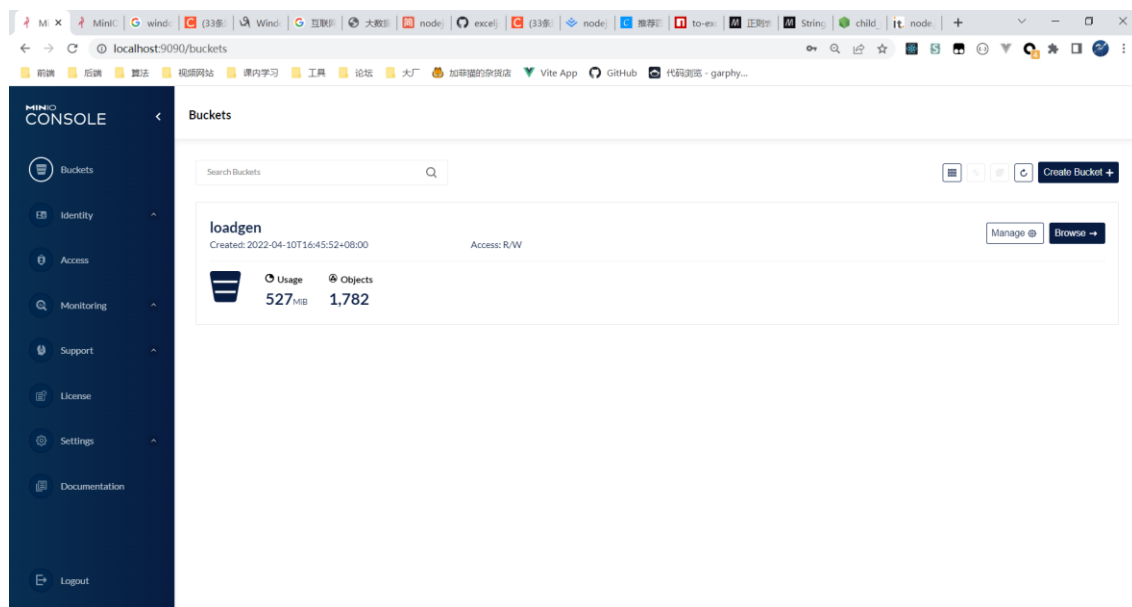
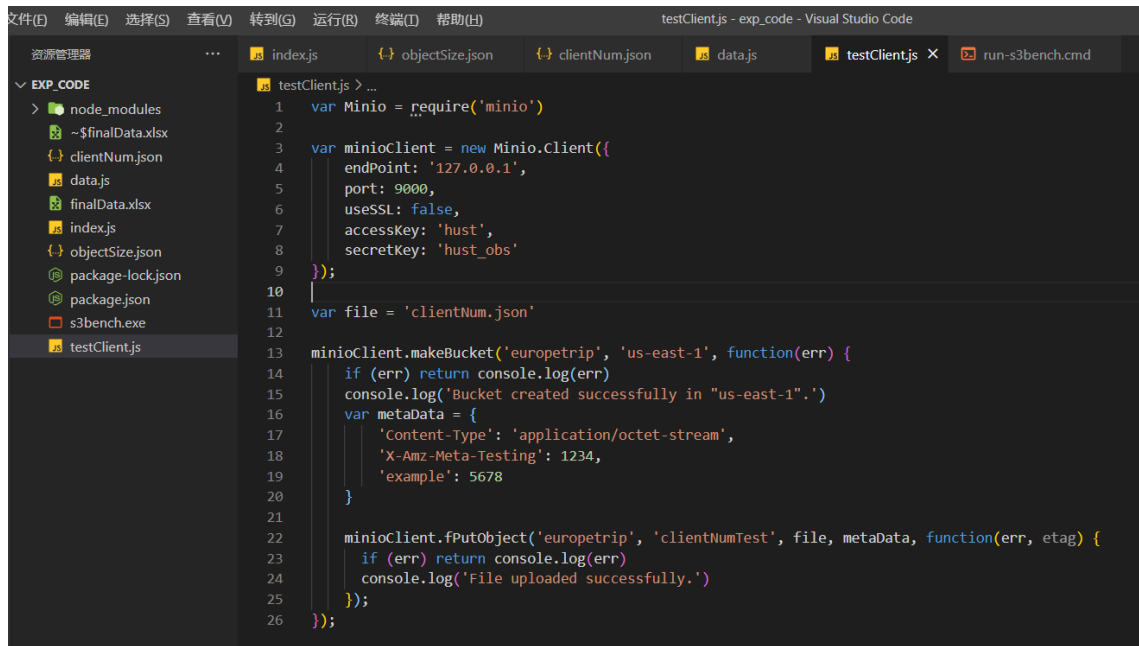


图 2 minio 管理界面

5.2 搭建对象存储客户端并测试对象存储功能

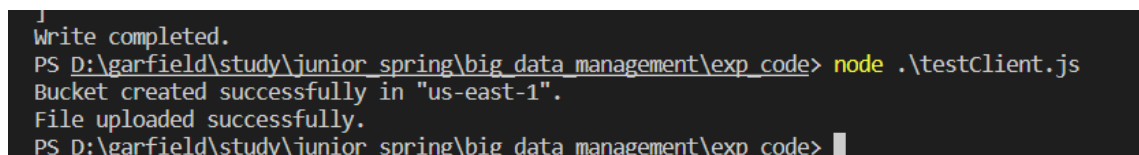
我们通过 nodejs 环境编写代码，作为 client，具体操作如下：

1. 下载 minio.js 的 npm 包
2. 连接 server
3. 创建一个 bucket
4. 上传一个文件
5. 执行 js 文件： node testClient.js



```
1 var Minio = require('minio')
2
3 var minioClient = new Minio.Client({
4   endPoint: '127.0.0.1',
5   port: 9000,
6   useSSL: false,
7   accessKey: 'hust',
8   secretKey: 'hust_obs'
9 });
10
11 var file = 'clientNum.json'
12
13 minioClient.makeBucket('europetrip', 'us-east-1', function(err) {
14   if (err) return console.log(err)
15   console.log('Bucket created successfully in "us-east-1".')
16   var metaData = {
17     'Content-Type': 'application/octet-stream',
18     'X-Amz-Meta-Testing': 1234,
19     'example': 5678
20   }
21
22   minioClient.fPutObject('europetrip', 'clientNumTest', file, metaData, function(err, etag) {
23     if (err) return console.log(err)
24     console.log('File uploaded successfully.')
25   });
26 });
```

图 2 nodejs 端的 minio client



```
Write completed.
PS D:\garfield\study\junior spring\big data management\exp_code> node .\testClient.js
Bucket created successfully in "us-east-1".
File uploaded successfully.
PS D:\garfield\study\junior spring\big data management\exp code>
```

图 4 运行结果

从终端提示我们可以看到，bucket 也创建完毕，file 也上传成功，我们到 minio 的管理后台去查看结果。

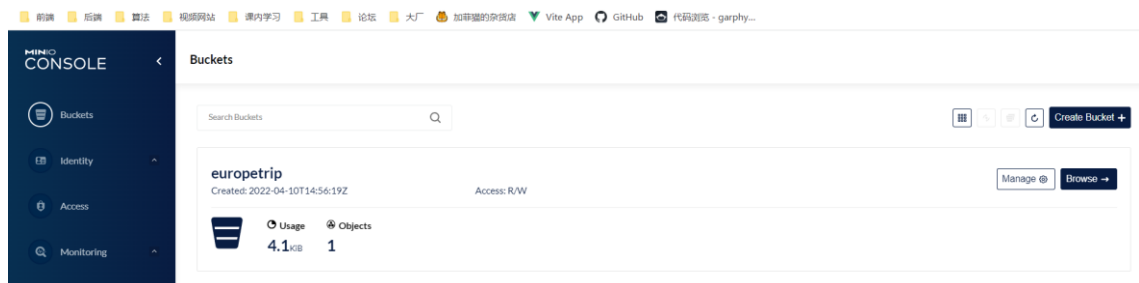


图 5 bucket 成功创建

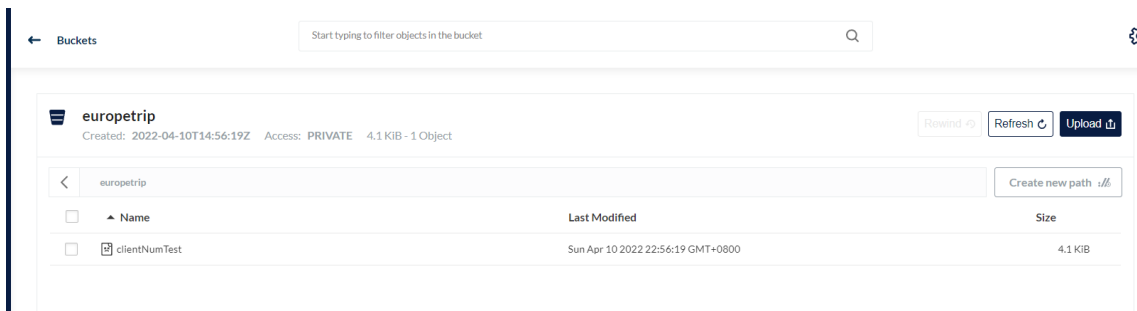


图 6 文件成功上传

5.4 对象存储性能分析

为了分析 minio 对象存储的性能，我选择安装 S3Bench（windows exe 文件），然后通过 run-s3bench.cmd 脚本来运行。

```
11 s3bench.exe ^
12     -accessKey=hust ^
13     -accessSecret=hust_obs ^
14     -bucket=loadgen ^
15     -endpoint=http://127.0.0.1:9000 ^
16     -numClients=8 ^
17     -numSamples=256 ^
18     -objectNamePrefix=loadgen ^
19     -objectSize=1024
20 pause
```

图 7 run-s3bench.cmd

通过阅读脚本内容，我们发现运行这个脚本需要一个名为 loadgen 的 bucket，因此我们先去 minio 后台创建，再运行，结果如下图

```
C:\WINDOWS\system32\cmd.exe
objectSize:      0.0010 MB
numClients:      8
numSamples:      256
verbose:         %!d(bool=false)

Results Summary for Write Operation(s)
Total Transferred: 0.250 MB
Total Throughput:  0.17 MB/s
Total Duration:    1.430 s
Number of Errors:  0
-----
Write times Max:      0.115 s
Write times 99th %ile: 0.094 s
Write times 90th %ile: 0.071 s
Write times 75th %ile: 0.062 s
Write times 50th %ile: 0.044 s
Write times 25th %ile: 0.029 s
Write times Min:      0.011 s

Results Summary for Read Operation(s)
Total Transferred: 0.250 MB
Total Throughput:  5.32 MB/s
Total Duration:    0.047 s
Number of Errors:  0
-----
Read times Max:       0.003 s
Read times 99th %ile: 0.003 s
Read times 90th %ile: 0.002 s
Read times 75th %ile: 0.002 s
Read times 50th %ile: 0.001 s
Read times 25th %ile: 0.001 s
Read times Min:       0.001 s

Cleaning up 256 objects...
Deleting a batch of 256 objects in range {0, 255}... Succeeded
Successfully deleted 256/256 objects in 225.822ms
```

图 8 s3bench 运行结果

为了测评 minio 的性能，我们当然要从不同的参数入手，这里我主要考虑了客户端数量以及对象大小这两个因素，而指标我主要参考 throughput 和 duration。当然作为程序员，肯定不能手动一条条执行指令并复制结果，这里我还是采用通过 nodejs 脚本编写代码来作为辅助工具。详细的处理代码可以参考 exp_code 文件夹下的 index.js 和 data.js。index 主要负责是批量跑命令以及数据的初步处理。Data.js 负责再将数据处理一下写入 excel。部分处理的数据如下图：


```
[
  {
    "base": { "objectSize": "0.0010", "numClients": "1", "numSamples": "1024" },
    "write": {
      "TotalTransferred": "1.000",
      "TotalThroughput": "0.15",
      "TotalDuration": "6.868",
      "NumberOfErrors": "0"
    },
    "read": {
      "TotalTransferred": "1.000",
      "TotalThroughput": "1.07",
      "TotalDuration": "0.935",
      "NumberOfErrors": "0"
    }
  },
  {
    "base": { "objectSize": "0.0010", "numClients": "2", "numSamples": "1024" },
    "write": {
      "TotalTransferred": "1.000",
      "TotalThroughput": "0.29",
      "TotalDuration": "3.415",
      "NumberOfErrors": "0"
    },
    "read": {
      "TotalTransferred": "1.000",
      "TotalThroughput": "2.14",
      "TotalDuration": "0.467",
      "NumberOfErrors": "0"
    }
  }
]
```

图 9 处理之后的数据

为了使结果更加清晰，我选择将数据写入 excel，结果如下

(1) 对象大小对性能的影响

变量为 objectSize, numClient = 8, numSamples = 1024

	A	B	C	D	E	F	G
1	object_size	W_TotalTransferred	W_TotalThroughput	W_TotalDuration	R_TotalTransferred	R_TotalThroughput	R_TotalDuration
2	0.0010	1.000	0.20	4.981	1.000	4.15	0.241
3	0.0020	2.000	0.36	5.535	2.000	7.92	0.253
4	0.0039	4.000	0.75	5.353	4.000	16.53	0.242
5	0.0078	8.000	1.51	5.303	8.000	30.99	0.258
6	0.0156	16.000	3.31	4.830	16.000	62.96	0.254
7	0.0312	32.000	5.92	5.410	32.000	130.26	0.246
8	0.0625	64.000	11.34	5.644	64.000	251.43	0.255
9	0.1250	128.000	21.85	5.859	128.000	476.14	0.269
10	0.2500	256.000	38.30	6.685	256.000	798.31	0.321
11	0.5000	512.000	71.72	7.139	512.000	1250.38	0.409
12	0.9766	1000.000	101.86	9.817	1000.000	1702.93	0.587
13							

根据 excel 分析，当对象大小从 0.001MB 逐渐增至 1MB，可知对象大小越大，吞吐率越大，消耗的总时间也越长。

(2) 并发客户端数对性能的影响

变量为 clientNum, objectSize = 1024, numSamples = 1024

client_num					
	A	B	C	D	E
1	client_num	W_TotalThroughput	W_TotalDuration	R_TotalThroughput	R_TotalDuration
2	1	0.15	6.868	1.07	0.935
3	2	0.29	3.415	2.14	0.467
4	4	0.37	2.690	3.48	0.288
5	8	0.18	5.494	4.67	0.214
6	16	0.16	6.273	3.89	0.257
7	32	0.14	7.044	3.56	0.281
8	64	0.14	7.109	3.05	0.328
9	128	0.14	6.945	2.32	0.430
10	256	0.13	7.589	1.48	0.676
11	512	0.13	7.601	1.11	0.905
12	1024	0.13	7.938	0.86	1.162
13					

根据 excel 分析, 当并发数量从 1 逐渐增至 1024, 吞吐量和总耗时均呈现先上升后下降的情况。

根据以上测试, 可知 I/O 延迟主要的影响因素:

①对象大小, 对象大小越大, 延迟越大。是因为对象越大需要从磁盘读取的数据越大, 耗时越长。

②并发数, 并发客户端越多, 延迟越大。是因为过多的连接请求产生拥塞, 需要排队处理请求, 而超过负荷的请求会造成请求失败, 数据丢失。

六、实验总结

本次实验是一次比较新颖的实验, 通过实验接触了面向对象存储这一新兴的存储技术, 虽然实验内容不算难, 但是能够从中学习到较多的技能。

在配置服务端、客户端和评测工具时, 老师给的脚本节省了很多学习的时间, 而且直接阅读脚本内容, 也能进一步理解是怎么配置的, 具体涉及到哪些参数, 参数具体作用是什么。由于 S3 Bench 没有批量测试和输出结果文件功能, 所以通过自己写代码实现了批量测试和终端输出的重定向。

参考文献

- [1] 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.
- [2] ARNOLD J. OpenStack Swift[M]. O' Reilly Media, 2014.
- [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] Dean J, Barroso L A. Association for Computing Machinery, 2013. The Tail at Scale[J]. Commun. ACM, 2013, 56(2): 74 - 80.
- [5] Delimitrou C, Kozyrakis C. Association for Computing Machinery, 2018. Amdahl's Law for Tail Latency[J]. Commun. ACM, 2018, 61(8): 65 - 72.