



2020 级

《物联网数据存储与管理》课程

实 验 报 告

姓 名 杨晨

学 号 U202015596

班 号 CS2010

日 期 2023.05.29

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.....	3
4.1 对象存储技术实践.....	3
4.2 对象存储性能分析.....	3
五、实验过程.....	4
5.1 搭建对象存储服务端.....	4
5.2 搭建对象存储客户端并测试对象存储功能.....	5
5.4 对象存储性能分析.....	6
六、实验总结.....	9
参考文献.....	10

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

对象存储技术是一种广泛应用于计算机领域的数据存储方法。它是一种将数据以对象的形式存储在分布式系统中的技术。

对象存储技术与传统的文件系统和块存储技术有所不同。传统的文件系统以文件层次结构的形式组织数据，而块存储技术将数据切分成固定大小的块进行存储。而对象存储技术则将数据存储为独立的对象，每个对象包含数据本身以及与之关联的元数据（例如标识符、大小、创建日期等）。这些对象被存储在一个分布式的存储集群中，可以通过唯一的标识符来访问和检索。

对象存储技术具有以下几个主要特点：

1.无限扩展性：对象存储系统可以轻松扩展以适应大规模数据的存储需求。它可以在集群中添加更多的存储节点，实现横向扩展，从而提供高度可伸缩的存储解决方案。

2.数据冗余和容错：对象存储系统通过使用冗余数据副本和纠删码等技术来保护数据的完整性和可靠性。即使在存储节点故障或数据损坏的情况下，系统也可以恢复数据，并确保数据的持久性和可用性。

3.高性能访问：对象存储系统通过并行化和数据分布在多个节点上的方式实现高性能的数据访问。它可以支持大规模并发访问和高吞吐量的数据读写操作。

4.多样化的数据访问接口：对象存储技术支持多种数据访问接口，包括基于 RESTful API 的标准化接口（如 Amazon S3 和 OpenStack Swift）、文件访问接口（如 NFS 和 CIFS/SMB）以及块级访问接口（如 iSCSI 和 Fibre Channel）。这些接口使得开发人员和应用程序可以方便地与对象存储系统进行集成。

5.低成本：对象存储技术通常采用经济高效的硬件设备，并具备灵活的存储管理和自动化的数据迁移功能。这降低了存储成本，并提供了更高的存储效率。

对象存储技术在云存储、大数据分析、备份和灾难恢复等场景中得到了广泛应用。它为海量数据的存储和管理提供了可靠、高效和可扩展的解决

三、实验环境

CPU： Intel(R) Core(TM) i9-12900H CPU @ 2.40GHz

内存: 16.0 GB
硬盘: SSD 2T
操作系统: Window11
Nodejs: v16.13.2
对象存储服务端: Minio
对象存储客户端: Aws-cli/2.11.20
评测工具: S3bench

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，包括客如对象尺寸 `objectSize`、并发数 `numClients`、样本数 `numSamples`，测试吞吐率 `Throughput`、延迟 `Latency` 等指标，并记录结果制成表格。
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。

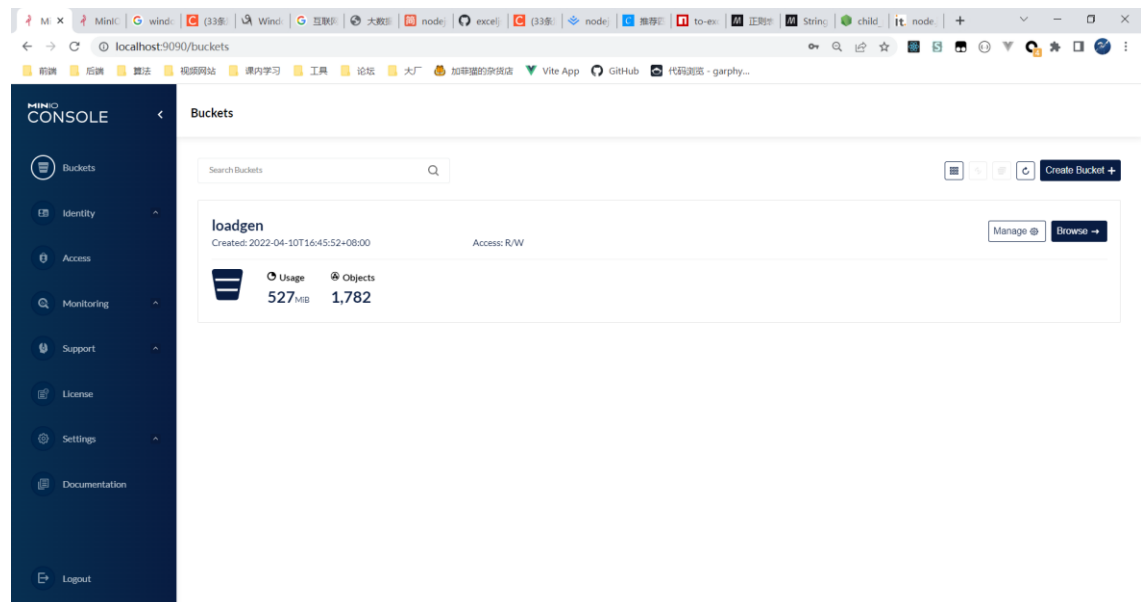
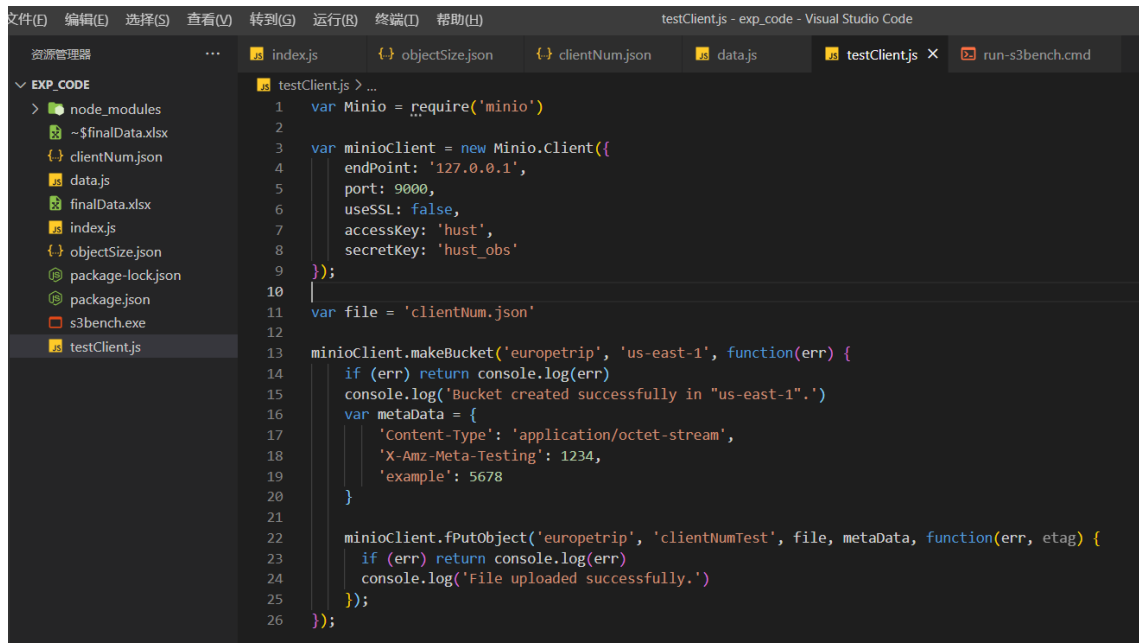


图 2 minio 管理界面

5.2 搭建对象存储客户端并测试对象存储功能

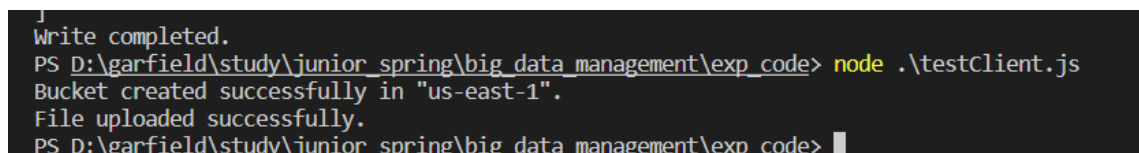
我们通过 nodejs 环境编写代码，作为 client，具体操作如下：

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.putObject('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。

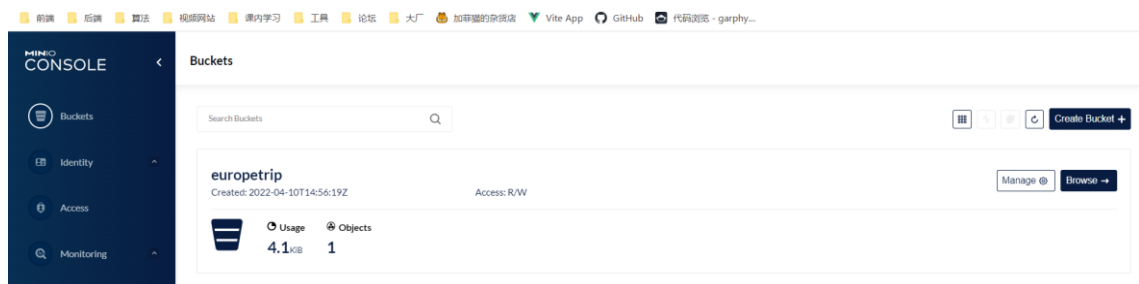


图 5 bucket 成功创建

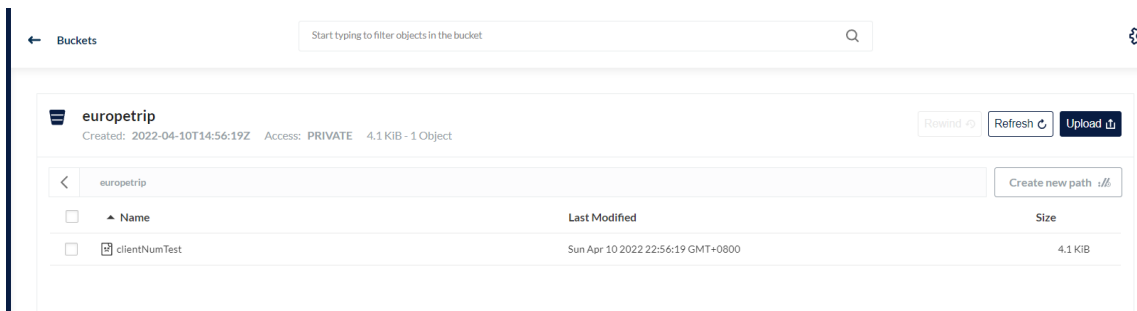


图 6 文件成功上传

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

	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

我使用了 MinIO 作为服务器, MC 作为客户端, 并使用 S3-bench 对不同影响因素下的存储性能进行了测试和观察。尽管我选择的测试数据量不大, 每组只有三个数据, 并且只完成了部分数据的测试, 所以无法得出非常准确的结论, 这有些遗憾。

然而, 这次实验对我来说还是有所收获的。通过这次实验, 我对对象存储技术有了更深入的了解, 同时也对相关软件有了一定的了解, 比如 MinIO 和 MC 的基本使用方法。由于我选择的是基本的软件, 并且直接在 Windows 上进行实验, 所以在实验过程中并没有遇到太多困难。但是, 无法尝试其他更高级的软件是一个小小的遗憾。数据存储应用非常广泛, 希望将来有机会能进一步接触更多的知识。

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.