对象存储实验报告

胡悦晨 机械科学与工程学院 D202180317

目录

实验一：系统搭建

服务端

下载 minio，通过官网下载 Windows 版本的 minio.exe，并将其放入 run-minio.cmd 所属 obs-tutorial-master 文件夹下，双击 run-minio.cmd 启动命令提示符窗口,运行客户端。

```
API: http://115.156.249.111:9000  http://127.0.0.1:9000
RootUser: hust
RootPass: hust_obs

Console: http://115.156.249.111:9090 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://115.156.249.111:9000 hust hust_obs

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

客户端：

下载客户端 mc.exe 到同一个文件夹中，打开一个 cmd 窗口，输入 mc.exe alias set myminio http://115.156.249.111:9000 hust hust_obs，也即启动服务器的端后的页面所给出的命令，执行成功后截图如下：

```
D:\minos\obs-tutorial-master>mc.exe alias set myminio http://115.156.249.111:9000 hust hust_obs
mc.exe: Configuration written to `C:\Users\gaogui\mc\config.json`. Please update your access credentials.
mc.exe: Successfully created `C:\Users\gaogui\mc\share`.
mc.exe: Initialized share uploads `C:\Users\gaogui\mc\share\uploads.json` file.
mc.exe: Initialized share downloads `C:\Users\gaogui\mc\share\downloads.json` file.
Added `myminio` successfully.
```

基于 minio 实现 CRUD 4 项基本操作

创建桶：使用在命令符窗口使用命令 mc mb D:/minos/picture，在 D:/minos 文件夹下创建桶 picture。

```
D:\minos\obs-tutorial-master>mc mb D:/minos/picture
Bucket created successfully `D:/minos/picture`.
```

上传本地文件：使用命令 mc mb D:/minos/test001.jfif D:/minos/picture，将本地文件夹 D:/minos 中的 test001.jfif 上传到创建的桶 picture 中。

```
D:\minos\obs-tutorial-master>mc cp D:/minos/test001.jfif D:/minos/picture
...st001.jfif:  143.29 KiB / 143.29 KiB [============================] 16.39 MiB/s 0s
```

列出桶中的所有对象：使用命令 mc ls D:/minos/picture 查看当前桶中的所有对象。

```
D:\minos\obs-tutorial-master>mc ls D:/minos/picture
[2022-01-07 10:56:56 CST] 143KiB test001.jfif
```

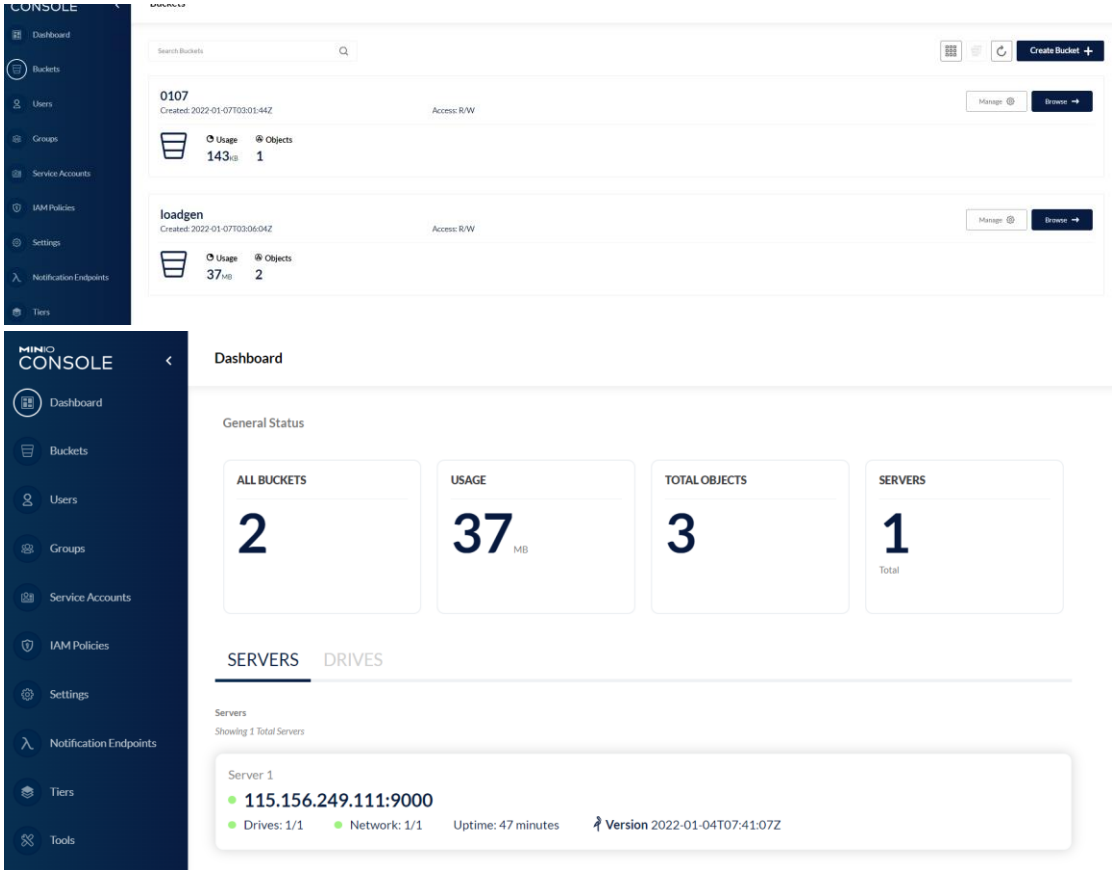
删除桶和对象：使用命令 mc rm D:/minos/picture，将桶和其中的对象全部删除。

```
D:\minos\obs-tutorial-master>mc rm D:/minos/picture
Removing `D:/minos/picture\`.
```

实验二：性能测试

选择 S3 Bench 测评工具，下载 s3bench.exe 文件并将其放入与 run-minio.cmd 同一文件夹 obs-tutorial-master 中。

进入 minio console 页面，创建项目名称为 loadgen 的桶



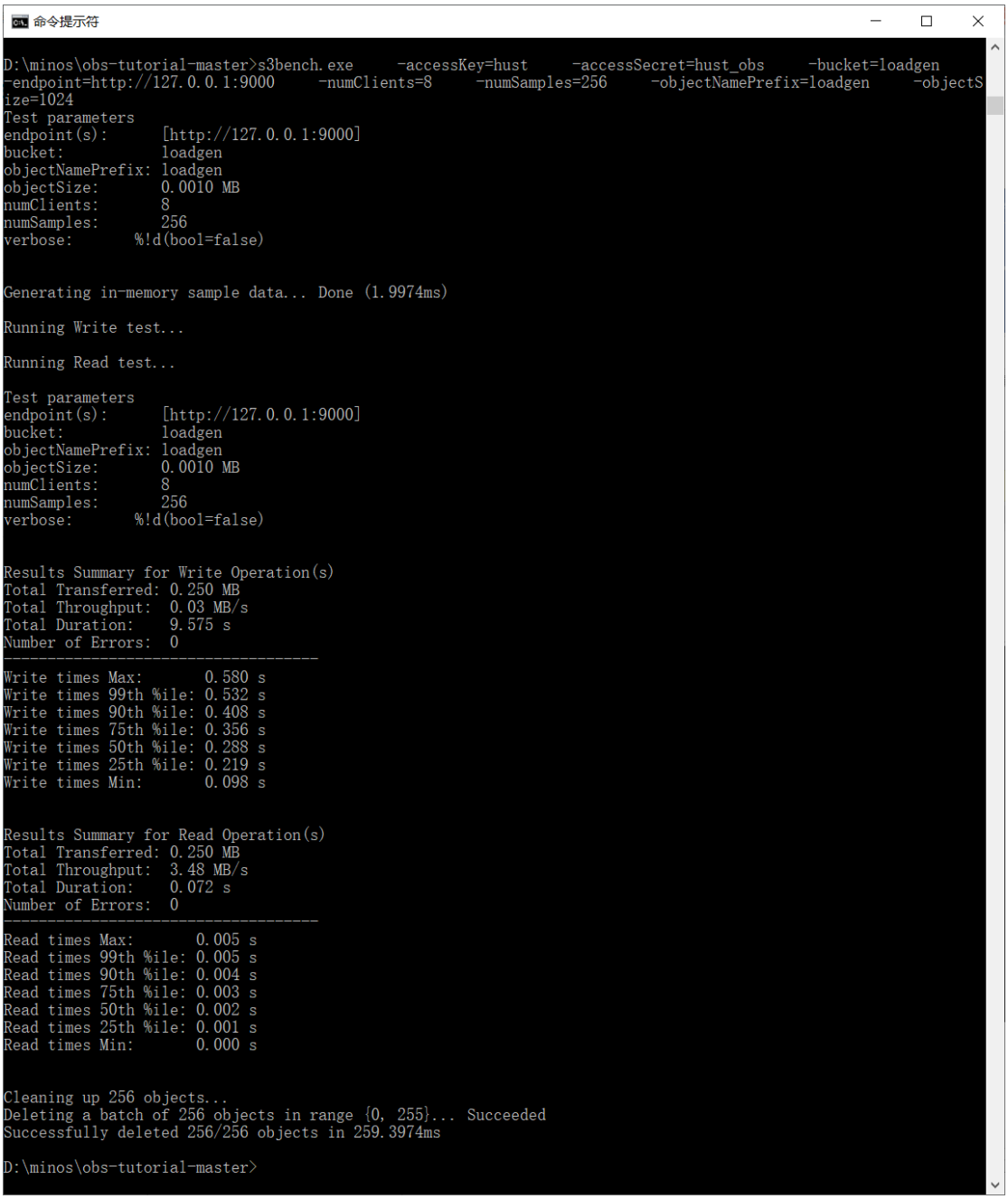
在 obs-tutorial-master 文件夹下通过 s3bench 向 minio 发送请求

命令行范例为

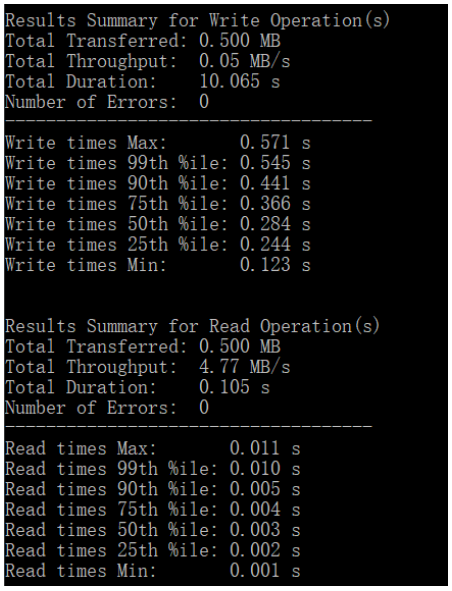
```
s3bench \
  -accessKey=hust -accessSecret=hust_obs \
  -endpoint=http://127.0.0.1:9000 \
```

```
    -bucket=loadgen -objectNamePrefix=loadgen \
    -numClients=8 -numSamples=256 -objectSize=1024
```

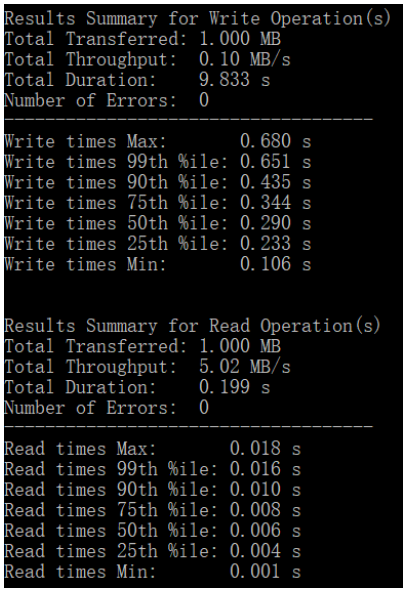
得到对象数量为 256，对象大小为 1024 负载下指标、延迟的分布。



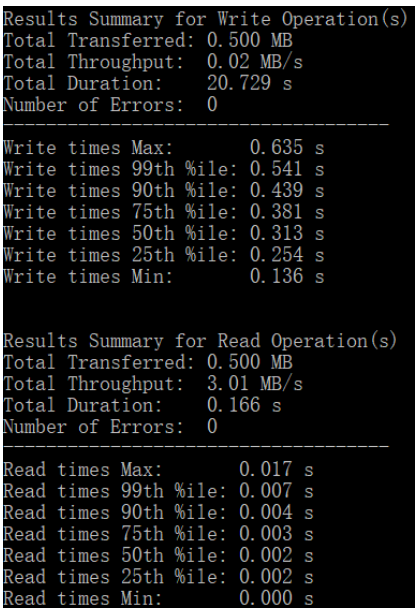
更改负载，修改对象数量和大小再进行实验，得到的实验数据如下：



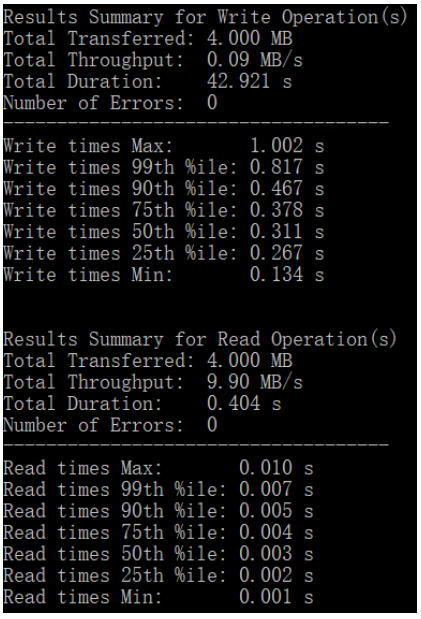
256*2048



256*4096



512*1024



1024*4096

观测以上数据可知，无论负载大小为多少，其按请求时间排序第 99%以上的请求占用的时间永远是最多的，某些负载下其占用时间最多请求的开销甚至为最短的 10 倍；同时，仅观测第 99%以上的请求占用的时间可以发现，随着负载的加重，其传输时间的开销也增大，系统的尾延迟现象越明显。

实验三：观测尾延迟

借助 Python 的 Jupyter Notebook Tutorial 平台，我们可以得到一组请求占用时间的具体数值，从而能绘制系统的尾延迟图像。在 Jupyter Notebook Tutorial 平台运行 obs-tutorial 中的脚本，得到的数据结果和图像如下所示：

表 1

latency							
1	81.7983	**26**	55.089	**52**	81.2366	**77**	54.199
2	76.0581	**27**	54.672	**53**	50.0617	**78**	55.1617
3	394.966	**28**	58.3143	**54**	65.4829	**79**	65.6307
4	203.358	**29**	65.7687	**55**	76.4959	**80**	64.7421
5	92.0756	**30**	48.6937	**56**	77.4121	**81**	68.6569
6	55.8422	**31**	53.6506	**57**	89.7205	**82**	50.0188
7	64.5833	**32**	43.4248	**58**	146.57	**83**	65.727
8	55.7456	**33**	55.342	**59**	60.4327	**84**	55.3596
9	42.8979	**34**	53.2553	**60**	56.5758	**85**	65.8329
10	98.8808	**35**	63.658	**61**	65.2933	**86**	53.678
11	55.053	**36**	166.767	**62**	64.8386	**87**	53.817
12	42.7687	**37**	55.5141	**63**	75.7935	**88**	55.1579
13	43.412	**38**	63.8025	**64**	54.224	**89**	66.1891
14	107.388	**39**	55.1441	**65**	66.8998	**90**	55.1484
15	85.819	**40**	52.4056	**66**	54.1937	**91**	65.7041
16	132.971	**41**	43.6327	**67**	76.9384	**92**	71.991
17	59.8075	**42**	55.7842	**68**	66.2868	**93**	47.9863
18	77.0197	**43**	64.0554	**69**	64.4343	**94**	98.7296
19	87.043	**44**	44.122	**70**	64.3744	**95**	76.1094
20	74.8019	**45**	59.1109	**71**	56.7436	**96**	70.6527
21	71.6884	**46**	48.2039	**72**	65.5391	**97**	59.4969
22	71.2416	**47**	76.0691	**73**	53.3304	**98**	56.051
23	325.628	**48**	55.2316	**74**	54.7693	**99**	64.8513
24	59.2425	**49**	65.345	**75**	65.1824	**100**	73.7875
25	64.0876	**50**	65.8514	**76**	66.047	**101**	45.3067

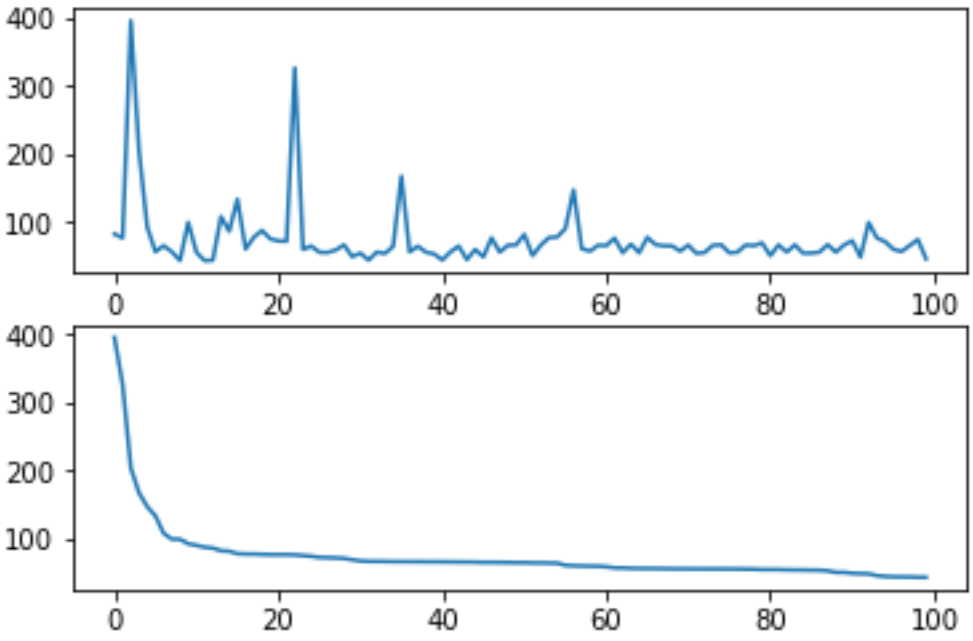


图 1

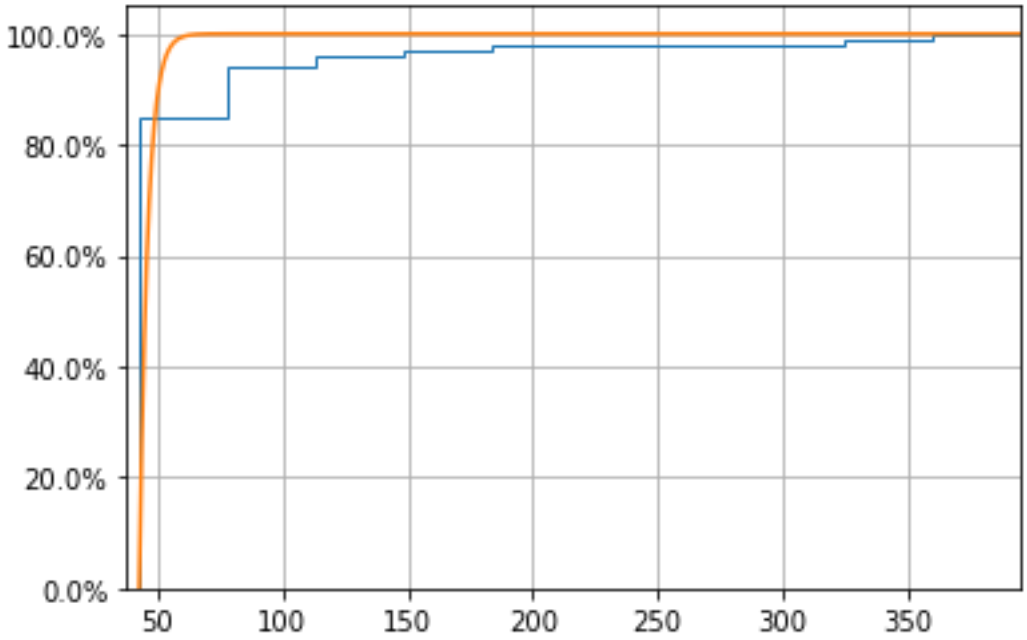


图 2

表 1 列出来 100 个请求对应的延迟，图 1 为表 1 数据的折线图，图 2 表示单个请求延迟的时间占总响应时间的大小。从图中可以看出序号 2、3 的请求时间高达 395ms 和 203ms，而系统 82%左右请求的时间都在 50ms 以下，剩余请求的开销远超过其他请求，因此该系统存在严重的尾延迟现象。