

2019 级

《大数据存储系统与管理》课程

# 实 验 报 告

姓 名 杨志军

学 号 U201916202

班 号 校交 1902 班

日 期 2022.04.14

# 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
4.3 对象存储服务性能优化.....	2
五、实验过程.....	3
5.1 对象存储技术实践.....	3
5.2 对象存储性能分析.....	6
5.3 对象存储服务性能优化.....	11
六、实验总结.....	16
参考文献.....	18

## 一、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，分析性能问题，架设应用实践。

## 二、实验背景

对象存储是一种将数据作为对象进行管理的计算机数据存储体系结构，它与其他存储结构不同，例如将数据作为文件层次结构管理的文件系统和将数据作为扇区和轨道内的块层次结构管理的块存储。每个对象通常包括数据本身、可变数量的元数据和全局唯一标识符。对象存储寻求实现其他存储架构无法实现的功能，例如可由应用程序直接编程的接口、可以跨越多个物理硬件实例的命名空间，以及对象级粒度的数据复制和数据分发等数据管理功能。

对象存储系统因其适用面广、使用方法简单而备受关注。但它在提供优秀服务的同时，也存在着一些问题，例如尾延迟现象。目前有关对象存储系统的性能优化方法仍在持续研究中。

## 三、实验环境

硬件环境	
CPU	Intel Core i7-8565U
内存	DDR4 2333M Hz 16.0GB
存储设备	NVMe KINGSTON SNVS500

软件环境	
Operation System	Windows 10
Python 内核	3.8.8
GoLang	1.18
boto3 (aws-sdk for Python)	1.20.9
aws-sdk-go (aws-sdk for GoLang)	1.43.34
go-echarts (echarts-sdk for GoLang)	2.2.5

系统组件	
Server	mock_s3
Client	boto3, aws-sdk-go
Test Program	my-s3bench 基于 aws-sdk-go 和 go-echarts 实现

## 四、实验内容

### 4.1 对象存储技术实践

实验一为系统搭建。根据个人情况选择适合自己的对象存储实验系统并完成搭建，并体验对象存储系统的功能服务。使用客户端发送 PUT 和 GET 请求，在服务端终端以及本地文件夹下观测服务端的响应结果。

由于我的系统环境上有配置好的 Python 环境，并且我个人也熟悉 Python 语言代码编写，因此我选择 `mock_s3` 作为我本次实验的服务端，选择 `boto3` API 作为客户端。

### 4.2 对象存储性能分析

实验二为性能观测。使用性能观测程序对实验一搭建的对象存储系统进行性能测试，并分析性能测试结果。

我尝试使用 `boto3` 编写自己的对象存储系统性能测试程序，但实现后发现 `boto3` API 的代码运行效率低，成为系统测试的瓶颈，无法很好地观测性能。因此我转而使用 Golang 语言中的 `aws-sdk-go` 作为我的测试 API，编写自己的测试程序 `my-s3bench`。我设置了多组“客户端数量、测试文件样例数量、测试文件大小”测试参数，对对象存储系统的读写性能进行测试，将结果以文本的方式初步呈现，并使用具体结果数据绘制成图表，一次测试对应一条曲线，多条曲线放置在一个平面上，方便对不同参数设置下系统的性能表现进行对比与分析。

### 4.3 对象存储服务性能优化

实验三为尾延迟挑战。在实验二的性能观测中可以发现对象存储系统存在尾延迟问题，实验三使用对冲请求或关联请求来应对尾延迟以优化对象存储系统，并观测使用了应对策略后的系统性能优化情况。

我通过修改 `my-s3bench` 中的代码，使测试程序可以模拟对冲请求或关联请求进行性能测试。我在“常规请求模式”、“对冲请求模式”、“关联请求模式”三种模式下对对象存储系统进行了性能测试，将测试结果进行对比分析，并总结归纳实验结论。

## 五、实验过程

### 5.1 对象存储技术实践

由于我的系统环境上有配置好的 Python 环境，并且我个人也熟悉 Python 语言代码编写，因此我选择 mock\_s3 作为我本次实验的服务端，选择 boto3 API 作为客户端。

- 基于 mock\_s3 的服务端搭建

mock\_s3 是一个用 Python 语言编写的简单的对象存储系统服务端，其代码精简而易懂，并且服务端的所有响应都会在终端输出显示出来，供用户参考，非常适合新手使用并学习。从 Github 上将代码克隆下来后，编写服务端配置启动脚本 run-mock-s3.cmd，运行脚本即可启动 mock\_s3 服务端。脚本程序的内容为 python mock\_s3/main.py --port 9090 --root .root，其含义是：运行 mock\_s3 的主函数，服务端监听本地端口 9090，并以“.root”文件夹作为存储系统的根目录，即客户端对服务端的所有文件读写操作都会被映射到“.root”文件夹下。服务端成功启动如图所示。

- boto3 环境安装

boto3 是亚马逊提供的 aws-sdk for Python，即基于 Python 语言的 aws 软件开发组件。和其他 aws-sdk 一样，boto3 提供了标准化的对一个对象存储系统进行读和写的操作 API。使用 boto3 之前，需要在本地安装 boto3 库。可以使用库管理工具 pip 来安装，安装命令为 pip install boto3。安装成功后，就可以通过编写并运行 Python 代码，把运行的 Python 脚本作为一个客户端，与服务端创建连接并取得对象存储服务。

- 客户端请求服务，服务端响应服务

搭建好服务端以及 boto3 后，我在 Jupyter Notebook 中使用 boto3 API 模拟客户端的操作。我使用以下代码配置客户端。

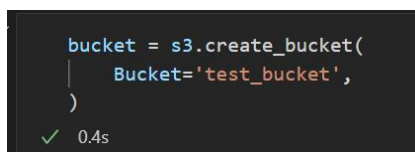
```
endpoint='http://127.0.0.1:9090'
access_key='hust'
secret_key='hust_obs'

s3=boto3.client(
    's3',
    aws_access_key_id=access_key,
    aws_secret_access_key=secret_key,
    use_ssl=False,
    region_name='cn',
    endpoint_url=endpoint,
)
```

图 5.1-1

需要注意的是，`s3=boto3.client(...)`函数并不会真的与服务端建立连接，而是类似于创建一个客户端配置 `config`，之后每次使用 `s3` 来请求服务端的服务都是用该配置即时建立一个连接并发送服务请求。因此，即使此处 `endpoint`, `access_key`, `secret_key` 参数设置错误，也不会有报错，而会在建立连接请求服务时报错。

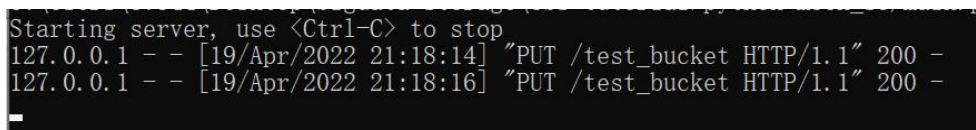
接下来我运行代码如下代码，请求在服务端中创建一个名为“test\_bucket”的存储桶。



```
bucket = s3.create_bucket(  
    Bucket='test_bucket',  
)  
✓ 0.4s
```

图 5.1-2

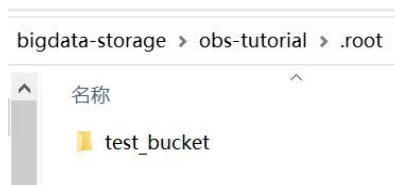
代码运行结束后，我们观察到 `mock_s3` 终端输出如下，证明服务端成功接收并响应了客户端的请求，创建了一个新桶。



```
Starting server, use <Ctrl-C> to stop  
127.0.0.1 - - [19/Apr/2022 21:18:14] "PUT /test_bucket HTTP/1.1" 200 -  
127.0.0.1 - - [19/Apr/2022 21:18:16] "PUT /test_bucket HTTP/1.1" 200 -  
-
```

图 5.1-3

我们可以在服务端的“`.root`”文件夹下找到新创建的“test\_bucket”文件夹，这就是服务端中名为“test\_bucket”的桶的存储位置。



```
bigdata-storage > obs-tutorial > .root  
名称  
test_bucket
```

图 5.1-4

我们使用 `boto3` 提供的 API 中的 `put_object` 来向服务端请求对象存储。我连续往服务端中 `put` 10 个对象，并输出每次 `put` 操作的运行时间。代码中我生成二进制串来模拟对象。代码内容和运行输出结果如下图所示。可以观察到，即使对象的大小很小（1024B），一次 `put` 操作仍然需要 1 秒左右的时间，API 效率极低。

```
for i in range(10):
    t1=time.time()
    response=s3.put_object(
        Bucket='test_bucket',
        Body=bytes(1024),
        Key='test'+str(i))
    t2=time.time()
    print(t2-t1)
```

[4] ✓ 10.2s

... 1.0510363578796387  
1.0201878547668457  
1.0317013263702393  
1.0159797668457031  
1.0441391468048096  
1.0168225765228271  
1.0192036628723145  
1.0191261768341064  
1.004087209701538  
1.020263910293579

图 5.1-5

在服务端终端可以观察到服务端对 10 次 put 操作的响应如下。

```
Starting server, use <Ctrl-C> to stop
127.0.0.1 - - [19/Apr/2022 21:18:14] "PUT /test_bucket HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:18:16] "PUT /test_bucket HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:25] "PUT /test_bucket/test0 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:26] "PUT /test_bucket/test1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:27] "PUT /test_bucket/test2 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:28] "PUT /test_bucket/test3 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:29] "PUT /test_bucket/test4 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:30] "PUT /test_bucket/test5 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:31] "PUT /test_bucket/test6 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:32] "PUT /test_bucket/test7 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:33] "PUT /test_bucket/test8 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:23:34] "PUT /test_bucket/test9 HTTP/1.1" 200 -
```

图 5.1-6

相应地，在".root/test\_bucket"中出现了被存储的对象，每个对象由元数据和对象内容组成。

bigdata-storage > obs-tutorial > .root > test_bucket >			
名称	修改日期	类型	
test0	2022/4/19 21:23	文件夹	
test1	2022/4/19 21:23	文件夹	
test2	2022/4/19 21:23	文件夹	
test3	2022/4/19 21:23	文件夹	
test4	2022/4/19 21:23	文件夹	
test5	2022/4/19 21:23	文件夹	
test6	2022/4/19 21:23	文件夹	
test7	2022/4/19 21:23	文件夹	
test8	2022/4/19 21:23	文件夹	
test9	2022/4/19 21:23	文件夹	

图 5.1-7

data-storage > obs-tutorial > .root > test_bucket > test5	
名称	修改日期
.mocks3_content	2022/4/19 21:23
.mocks3_metadata	2022/4/19 21:23

图 5.1-8

最后，我们尝试使用 boto3 发起 get 请求。代码内容和运行结果如下图所示。由图可知，boto3 提供的 get\_object 的运行时间很短，执行效率比 put\_object 高非常多。

```
for i in range(10):
    t1=time.time()
    response=s3.get_object(
        Bucket='test_bucket',
        Key='test'+str(i))
    t2=time.time()
    print(t2-t1)
```

✓ 0.7s

0.3657088279724121  
0.02522587776184082  
0.03466653823852539  
0.03289031982421875  
0.0306093692779541  
0.013472318649291992  
0.023671388626098633  
0.0326848030090332  
0.0319364070892334  
0.0316464900970459

图 5.1-9

服务端终端输出了对 get 请求的响应，如下图所示。

```
127.0.0.1 [19/Apr/2022 21:31:49] "PUT /test_bucket/test9 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test0 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test2 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test3 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test4 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test5 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test6 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test7 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test8 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 21:31:49] "GET /test_bucket/test9 HTTP/1.1" 200 -
```

图 5.1-10

至此，我们就完成了实验一的内容，即 mock\_s3 服务端、boto3 客户端的搭建，并执行 boto3 API 发送客户端的 PUT 和 GET 请求，在服务端终端以及本地文件夹下观测服务端的响应结果。

## 5.2 对象存储性能分析

5.1 中我们在本地搭建了对对象存储服务并安装了 boto3 sdk。在 5.2 中，我们将使用 aws-sdk 中提供的 PUT 和 GET API 来对服务端的 PUT 和 GET 性能进行观测，观测的指标包括吞吐率、服务时延等。为此，我们设计并实现一个测试程序。用户可以通过命令行来设置测试参数，包括服务端的 endpoint、bucket，单个对象大小、对象数量、同时与服务端连接的客户端数量等。测试程序将建立相应数量的客户端，发送相应数量、相应大小的对象 PUT 和 GET 请求，并记录服务响应延迟，测试完成后将对这些记录进行分析，在测试程序终端用文本输出测试结果。

起初，我使用 boto3 编写了完整的测试程序，但在运行程序时发现在 5.1 中提



到的 PUT API 执行效率太慢成为了测试的瓶颈，即使对象大小已经非常小，一个 PUT 操作仍至少要花费 1 秒，完全无法起到性能测试的作用。经分析，我认为这是由 Python 语言的执行效率低导致的。经查阅资料得知，GoLang 中也有一套 aws-sdk，并且其 API 执行效率非常高。因此，尽管我此前对 GoLang 一窍不通，我还是决定学习 GoLang 并使用 aws-sdk-go 来实现测试程序。

简单介绍一下 GoLang 实现测试程序的关键代码思路。一些简单的代码此处不再赘述，如解析命令行参数等。源代码文件已上传，详情请参见 [my-s3bench.go](https://github.com/awslabs/my-s3bench-go)。

由于 GoLang 中的通道机制可以让协程之间的通信实现起来非常简单，因此我使用通道机制以及“生产者——消费者”设计模式来提高测试程序的并行度。思路为：定义四个通道，分别为 putReqChan（存放 PUT 请求）、putRespChan（存放 PUT 响应）、getReqChan（存放 GET 请求）、getRespChan（存放 GET 响应）；有一个生产者协程专门负责根据当前测试模式产生请求，若当前测试模式为 Write Test，则产生参数设置的特定数量个 PUT 请求依次放入到 putReqChan 中，放置完成后关闭管道，若当前测试模式为 Read Test，则产生参数设置的特定数量个 GET 请求依次放入到 getReqChan 中，放置完成后关闭管道；客户端协程则是消费者协程，先不断地从 putReqChan 中取出一个 PUT 请求并发送，然后将收到的响应的延迟一起放入到 putRespChan 中，消费完 putReqChan 通道中的 PUT 请求后，开始消费 getReqChan 通道中的 GET 请求，发送请求，并将收到响应的延迟放入 getRespChan 中；由于生产者在放完所有请求后已经把通道关闭，因此消费者消费完通道内的所有请求后不会被阻塞，可以继续顺序执行；客户端协程的并发数量由参数设置中的 numOfClients 决定；putRespChan 和 getRespChan 两个通道内的响应延迟又由另外一个协程（主协程）来消费，这个协程的任务就是设置测试参数，启动 numOfClients 个客户端协程，启动请求生产者协程，在 Write Test 模式下取出 putRespChan 通道中的元素，对 Write 测试结果进行分析，在 Read Test 模式下取出 getRespChan 通道中的元素，对 Test 测试结果进行分析，分析结束后，会在打印输出测试结果分析报告。协程之间的通信如下图所示。

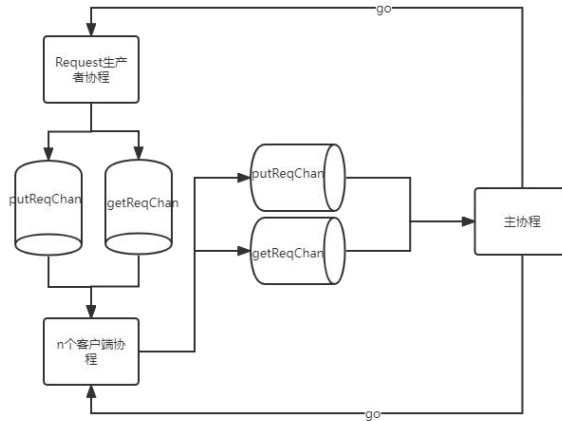


图 5.2-1 my-s3bench 中协程之间的通信

然而，测试程序以文本的形式输出测试结果报告并不能让用户很快地获取测试结果中的信息，因此我利用了 GoLang 中的 go-echarts 库将测试结果进行了可视化。当测试程序结束时，会自动生成可视化输出，并保存在本地的 Write.html 和 Read.html 文件中。

当测试参数 {numOfClients=8, numOfSamples=256, objectSize=1024} 时，测试程序输出的文本报告和可视化图表输出如下。由文本报告和可视化图表可知，大部分 PUT 和 GET 请求可以在较短时间内被响应，但是可能会有极小一部分的请求的响应延迟远远超出了正常的响应延迟。这种现象被称为“尾延迟”。

```

params:{ accessKey:hust secretKey:hust_obs bucket:loadgen endpoint:http://127.0.0.1:9090 numOfClients:8 numOfSample
s:256 objectNamePrefix:loadgen objectSize:1024 }Generating the sample data in memory whose size is 1024...
Done. Time cost : 0s
Setting up 8 clients...
Runnig Write Test...
Done.
Runnig Read Test...
Done.

Result Summary for Write Operations
Total Transferred:    0.250 MB
Total Throughput:     0.188 MB/s
Total Duration:       0.75115 s
Number of Errors: 0

-----
Write times Max :      0.04211 s
Write times 99%th:    0.03859 s
Write times 90%th:    0.03239 s
Write times 75%th:    0.02820 s
Write times 50%th:    0.02223 s
Write times 25%th:    0.01715 s
Write times Min :      0.01034 s

Result Summary for Read Operations
Total Transferred:    0.250 MB
Total Throughput:     0.141 MB/s
Total Duration:       0.56537 s
Number of Errors: 0

-----
Read times Max :      0.51976 s
Read times 99%th:    0.03175 s
Read times 90%th:    0.01863 s
Read times 75%th:    0.01527 s
Read times 50%th:    0.01207 s
Read times 25%th:    0.01096 s
Read times Min :      0.00844 s

The line chart of the test result is saved as Write.html
The line chart of the test result is saved as Read.html
  
```

图 5.2-2 {numOfClients=8, numOfSamples=256, objectSize=1024} 测试结果文本报告

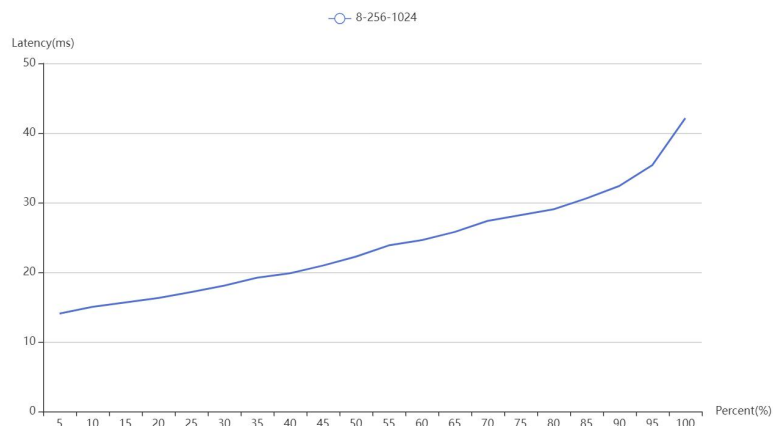


图 5.2-3 Writing Test 结果图表

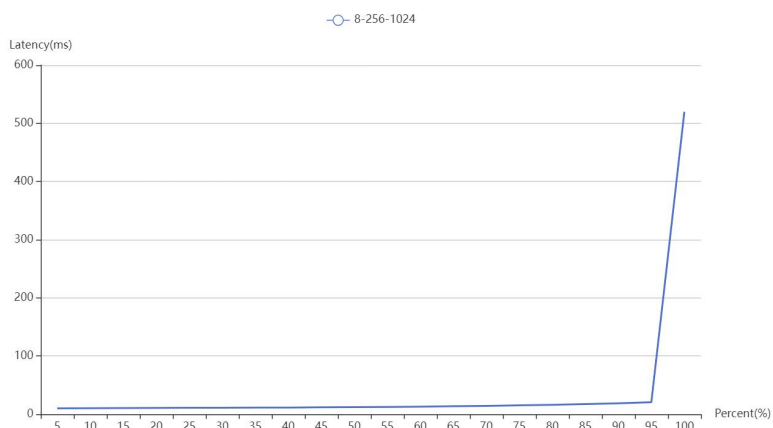


图 5.2-4 Reading Test 结果图表

为了更好地观测服务端在不同负载情况下的尾延迟现象，我们需要多次调整 {numOfClients, numOfSamples, objectSize} 参数，执行测试程序，观察输出的测试结果文本报告和可视化图表。人工地多次调整参数效率极低，因此我在测试程序的代码做了改动。我使测试程序可以支持用户在命令行参数中输入 numOfClients 数组, numOfSamples 数组, objectSize 数组，测试程序将用一个三重循环遍历三个数组中元素的所有组合情况，以每一组 {numOfClients, numOfSamples, objectSize} 参数来测试服务端性能，并输出测试结果文本报告，当所有测试结束时，将所有测试结果在一个图表上可视化展示出来，图表上的一条折线表示一次测试的结果，图例中的 “ %d-%d-%d ” 表示的是这条折线所代表的测试参数”numOfClients-numOfSamples-objectSize”。将所有折线放置在一个图表上的目的是方便用户观察与对比，若用户觉得折线数量太多影响观察，可以通过点击图

例来隐藏或展示一条折线。一次测试程序启动参数和生成的测试结果可视化图表如下所示。我给出的测试程序启动参数样例中，numOfClients=[1,4,8]，表示测试程序需要遍历 numOfClients 参数的值为 1,4,8 的测试，numOfSamples 和 objectSize 以此类推。因此，该测试程序启动参数样例会使测试程序总共进行  $3*3*2=18$  组测试。这也导致了测试结果文本报告非常长，不宜展示，但是我将它保存到了文本文件 **output.txt**，与可视化图表 **Write.html** 和 **Read.html** 一并上传到实验包中，若读者感兴趣可进行查阅。

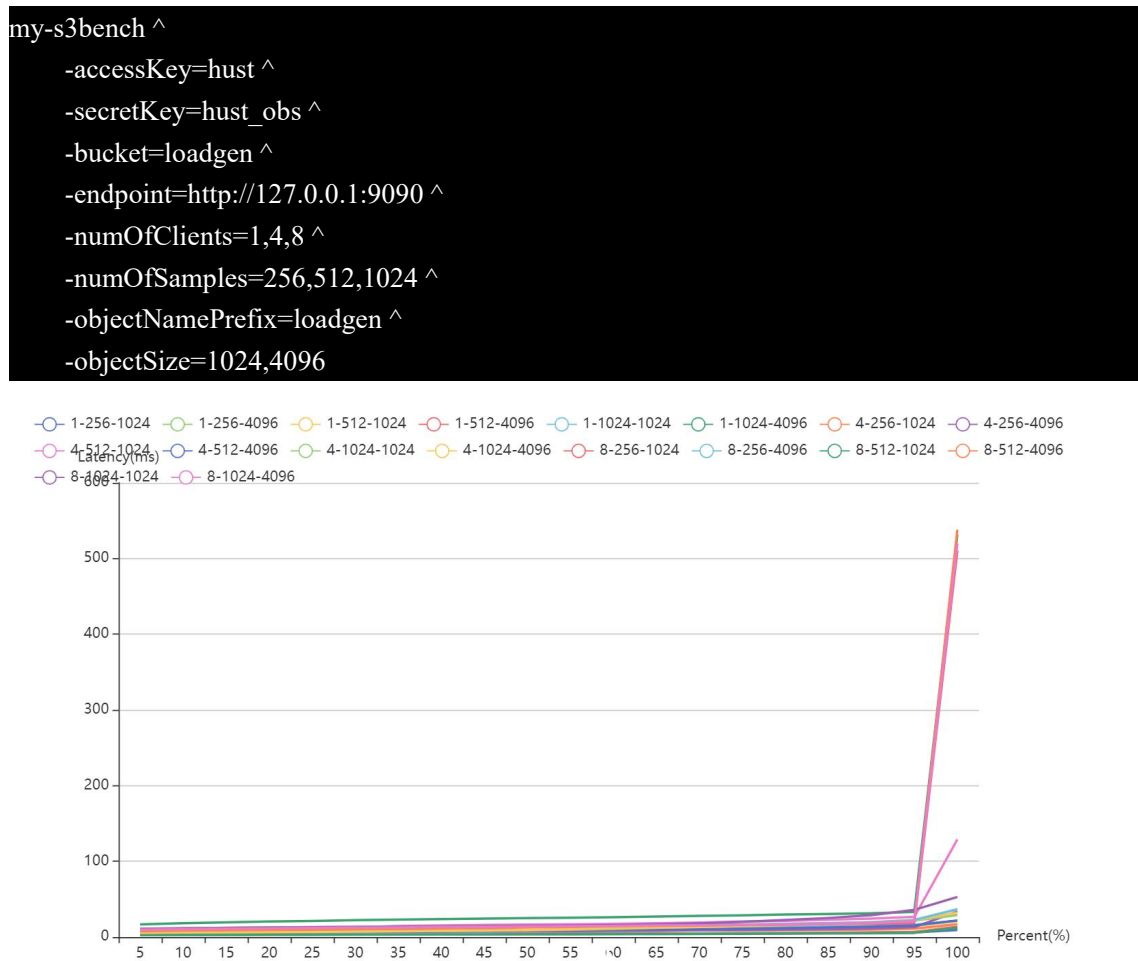


图 5.2-5 Writing Test 结果图表

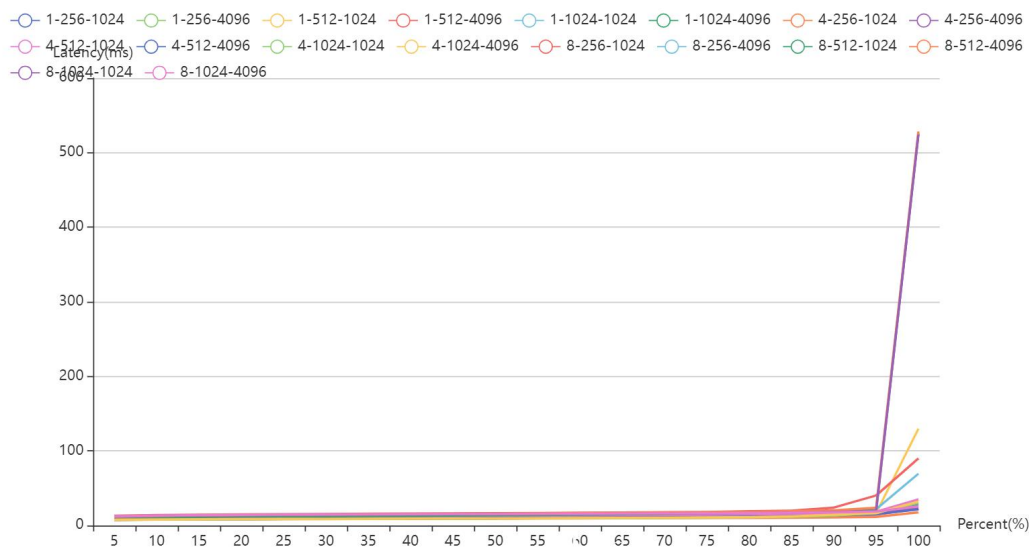


图 5.2-6 Reading Test 结果图表

从图表可以看出，在多组测试下都出现了尾延迟现象，部分组的尾延迟现象较严重，部分组的尾延迟现象较轻微。

至此，我们完成了实验二，即使用 `aws-sdk-go` 编写 s3 性能测试程序，将性能测试结果以文本报告和可视化图表的形势展现出来。并且，我更进一步地强化了性能测试程序的功能，即支持参数数组循环嵌套遍历测试。当输入的参数不是数组而是单个数时，测试程序的功能就退化到了一般功能，即只有一组参数，只进行一次测试。我将测试程序的代码 `my-s3bench.go`、测试程序可执行文件 `my-s3bench.exe`、测试程序启动脚本一并上传到实验包，感兴趣的读者可以参考。但是需要注意的是，运行测试程序的时候测试的组数是三个数组中元素个数的乘积，所以测试程序会进行导致大量的硬盘读写操作，请读者谨慎设置参数，注意保护自己的硬盘。

### 5.3 对象存储服务性能优化

在实验二中我们观察到，对象存储服务的响应存在尾延迟现象，即客户端在请求服务端服务的时候，会有极小的概率出现异常大的响应延迟。在特定事务背景下，若客户端对服务短延迟要求特别严格，那么这个小概率事件也是不容忽略的。因此，我们要直面尾延迟现象，并设法解决或缓解它。

目前广泛应用的两种尾延迟解决方案是对冲请求和关联请求。

对冲请求的想法是，若某一个请求未在客户端设置的时间阈值内被响应，则

客户端重新发送相同的请求，在多服务端的场景下，重新发送的请求可以被发送到与之前不同的服务端。换句话说，对冲请求试图通过“超时重传”的机制来解决尾延迟现象。

关联请求的想法是，客户端同时向多个服务器发送服务请求，当客户端收到一个请求的响应时，就取消其他请求。换句话说，关联请求试图通过“并行请求”的机制来提高服务响应容错率以解决尾延迟现象。

在实验三中，我修改实验二客户端协程 `clientRoutine` 的运行代码来模拟客户端支持对冲请求和关联请求的场景，具体实现方法如下。

我为客户端协程设置三个请求模式，模式 0 为常规模式，即实验二采取的单一请求发送的模式；模式 1 为对冲请求模式，在这个模式下客户端协程将模拟对冲请求；模式 2 为关联请求模式，在这个模式下客户端将模拟关联请求。用户可以在启动测试程序时通过命令行参数 `requestMode` 来设置请求模式，缺省值为 0。在对冲请求模式下，客户端协程先发送一个请求并记录响应时间  $t_1$ （与常规模式一样），若响应时间  $t_1$  超过了事先设置的请求超时阈值  $timeT$ ，则将这个请求重新发送一次并记录响应时间  $t_2$ ，则可以计算出在实际对冲请求下，本次请求的响应延迟为  $\min(t_1, timeT+t_2)$ ，将这个响应延迟作为结果送入响应通道。在关联请求模式下，客户端协程在发送第一个请求并记录响应时间  $t_1$  后，接着重新发送该请求并记录响应时间  $t_2$ ，则可以计算出在实际对冲请求下，本次请求的响应延迟为  $\min(t_1, t_2)$ ，将这个响应延迟作为结果送入响应通道。

在上述的实现中，有几点需要特殊说明。一，在实际的对冲请求和关联请求中，并不是像上述实现一样等前一个请求完成了再发送下一个请求，但由于我们使用的 `aws-sdk-go` 提供的 API 不支持异步的 GET 和 PUT 请求，所以我们只能用顺序请求的方式来实现，并通过后期计算来等效模拟实际对冲/关联请求场景下的响应延迟。二，实际应用中，服务端往往有多个，客户端重复发送的请求往往会发送到不同的服务端上，但是本次实验我们只有一个服务端，因此只能发送同一个服务端上。三，上述提到的对冲请求中客户端需要设置一个请求超时阈值  $timeT$ ，这个阈值需要能够让大部分第一次请求能够得到响应，只有极小部分的请求才能超过该阈值；我通过对实验二常规请求模式下生成的可视化图表中的每一条折线进行分析，发现大部分的请求可以在 30ms 内得到响应（读者可以打开图表进行查

验），因此我将 `timeT` 设置为 `30ms`；实际上，请求超时阈值 `timeT` 也可以使用动态维护的方法实现，参考 TCP 的超时重传阈值，受时间限制，作者没有尝试。

完成了代码实现后，我使用测试程序分别在对冲请求模式、关联请求模式下进行服务端性能测试，`{numOfClients, numOfSamples, objectSize}` 参数设置与实验二一致，采用数组循环嵌套遍历的方式来生成多组参数。

对冲请求测试结果如下，可以观察到，所有的请求都可以在 `60ms` 内得到响应，不存在实验二的结果中某个请求的响应实验达到将近 `600ms` 的现象，说明对冲请求很好地缓解了尾延迟现象。

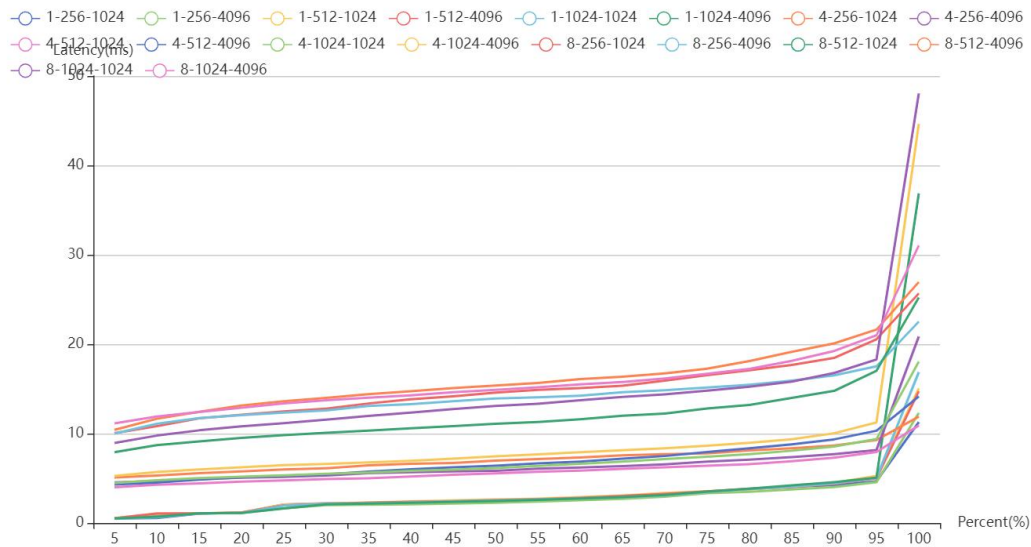


图 5.3-1 Writing Test 结果图表

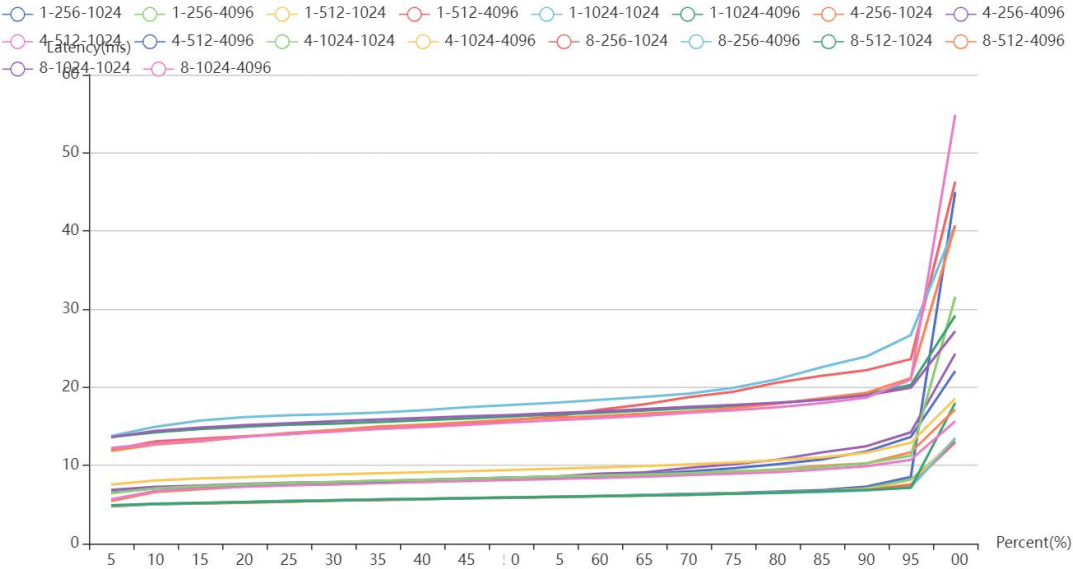


图 5.3-1 Reading Test 结果图表

关联请求测试结果如下。可以观察到，大部分请求的响应不存在尾延迟，仍然存在极小的一部分请求的响应有尾延迟，但相比于实验二的结果已有很大改善。我认为，这是因为关联请求会不计后果地同时发送多个请求，这可能会造成包括网络资源、服务端资源的浪费，造成网络拥塞、服务拥塞，尤其是在本次实验只有单一服务端的场景，服务拥塞现象会更明显。

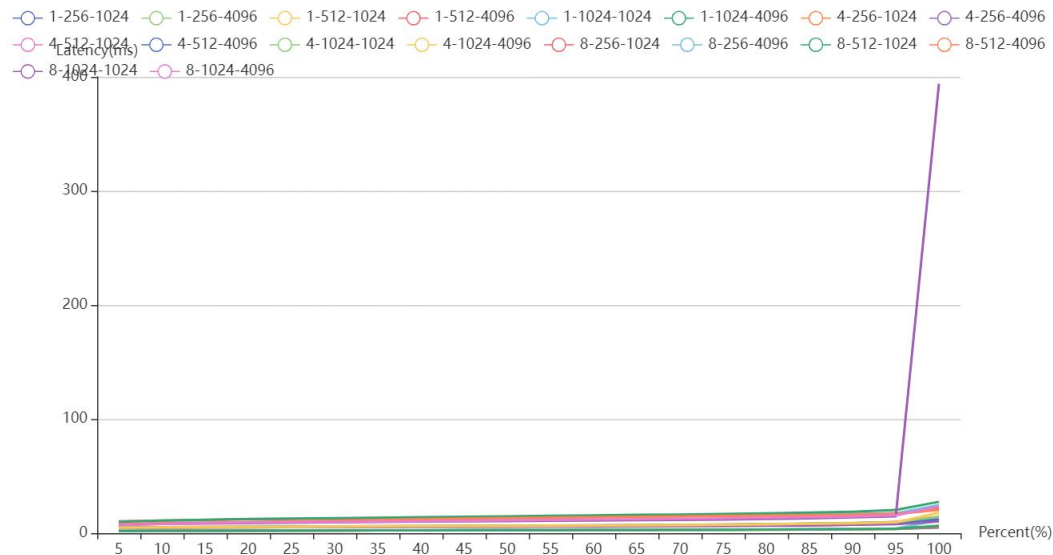


图 5.3-3 Writing Test 结果图表

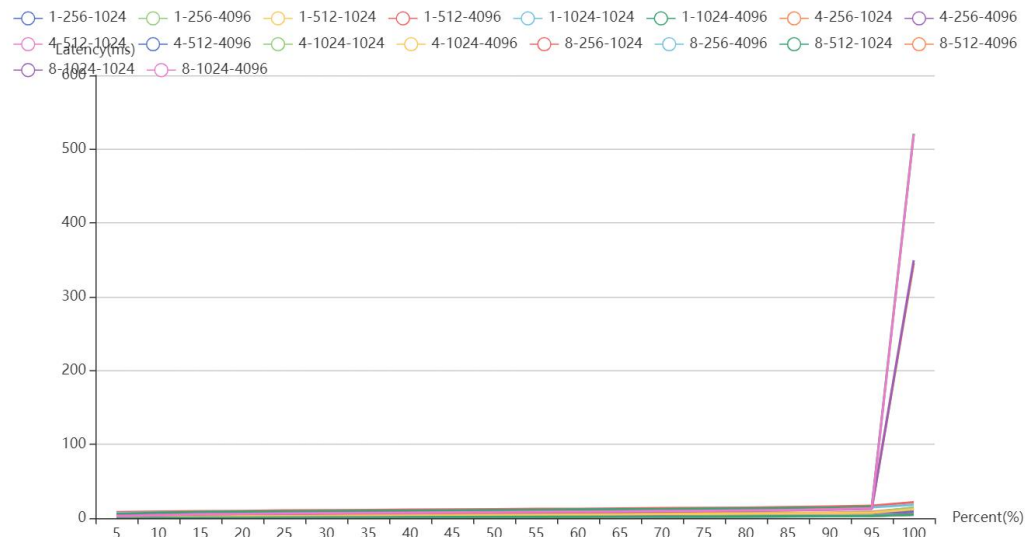


图 5.3-4 Reading Test 结果图表

点击图例隐藏有尾延迟的折线（Write 中有 1 条，Read 中有 4 条，灰色图例代表折线被隐藏）后，图表如下。可以观察到，使用了关联请求后，大部分的请求都能在 30ms 内被响应，这比对冲请求的 60ms 整整少了 30ms。这其实很好解释，因为对冲请求需要等响应超后才



能重新发送请求，而关联请求直接将同一个请求同时发送多份，节省了等待响应超时阈值的时间。但是关联请求的这种不计后果的做法也存在缺点，即上述提到的，多余的请求会造成包括网络资源、服务端资源的浪费，造成网络拥塞、服务拥塞，使服务端效率降低，正是因为这个原因，关联请求模式下的测试程序仍然检测到了尾延迟现象（虽然数量很少），而对冲请求模式下的测试程序就没有这种情况。

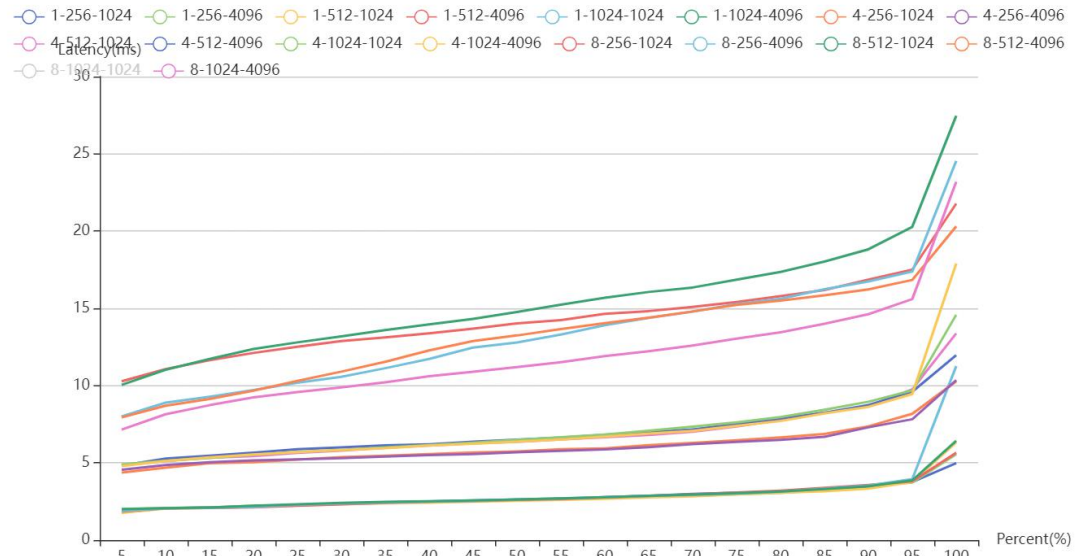


图 5.3-5 Writing Test 结果图表

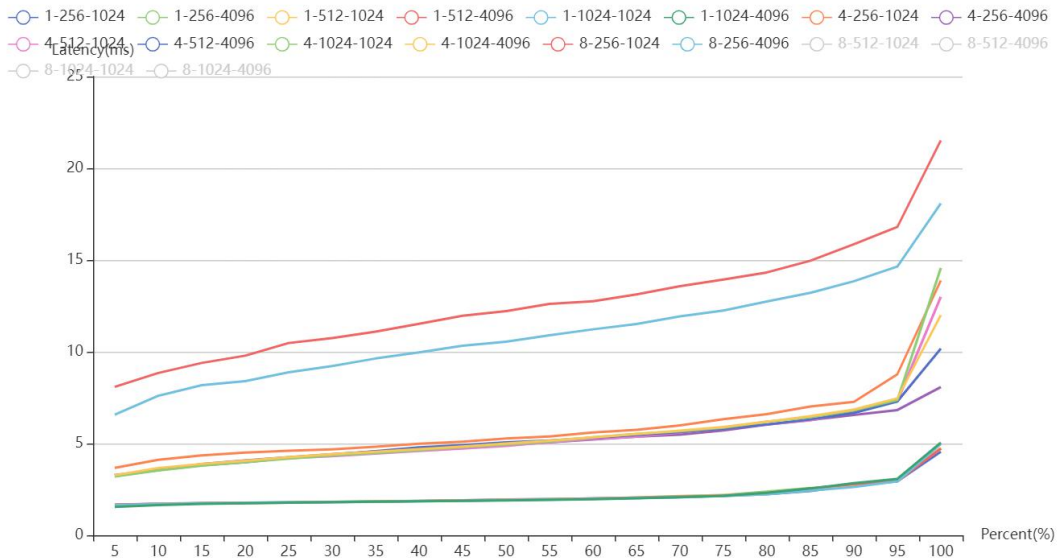


图 5.3-6 Reading Test 结果图表

综合上述分析，我们可以得出一个结论，在解决尾延迟现象时，对冲请求试图尽最大可能减少尾延迟发生的概率，而关联请求试图尽可能快地获得响应；在

服务资源充裕的情况下，关联请求的性能表现优于对冲请求；但当服务存在拥塞时，关联请求的性能表现可能出现巨大波动，存在尾延迟现象；而对冲请求则在各种服务环境下都能展现出稳定的性能表现。因此，我们可以构想出一种新的请求模式——“混合请求”，即将对冲请求与关联请求综合在一起，当服务资源充裕时，超时重传阈值 `timeT` 较小、甚至可以等于 0（`timeT=0` 时混合请求退化为关联请求），当出现服务拥塞时，超时重传阈值 `timeT` 增大，以对冲请求的方式来追求稳定的响应延迟。这实际上就是我们上述提到的动态维护超时响应阈值的对冲请求。

## 六、实验总结

在本次实验中，我动手搭建了对象存储服务端、客户端，并使用 `aws-sdk-go` 编写了一个 `s3` 测试程序——`my-s3bench`。并且 `my-s3bench` 支持数值参数以数组的方式传入，进行数组嵌套循环遍历，对不同数组元素的所有可能组合参数设置进行了测试，并能将测试结果以文本报告、可视化图表的形式输出以供用户参考，设计合理，功能强大。实验二中我使用测试程序对服务端的性能进行了测试，观察到了尾延迟现象。在实验三中，我使用对冲请求、关联请求来缓解尾延迟现象，并进一步对 `my-s3bench` 进行了升级，使它可以支持模拟对冲/关联请求来测试服务端性能（通过命令行参数 `requestMode` 来设置）；我使用升级后的测试程序观察对比了在“常规请求模式”、“对冲请求模式”、“关联请求模式”三种模式下的服务端性能表现，得出结论：对冲请求和关联请求都可以缓解尾延迟现象使服务端性能表现优于常规请求；在解决尾延迟现象时，对冲请求试图尽最大可能减少尾延迟发生的概率，而关联请求试图尽可能快地获得响应；在服务资源充裕的情况下，关联请求的性能表现优于对冲请求；但当服务存在拥塞时，关联请求的性能表现可能出现巨大波动，存在尾延迟现象；而对冲请求则在各种服务环境下都能展现出稳定的性能表现。

在本次实验中，我第一次通过实践感受到了对象存储的实现原理、优势、缺点，了解了它的性能表现、存在的问题、优化的手段。实验中关于尾延迟的解决方案也给了我很多启发，我会将这些解决方案积累到自己的“库”中，以防不时之需。在实验中，我还从零开始学习了 `GoLang` 语言，并最终实现了一个集自动化、

可视化为一体的我自认为优秀的测试程序。通过本次实验，我受益颇多。

## 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.