实验一:系统搭建

配置 Python 环境

1. 下载 Minconda3 Windows 64-bit
2. 添加环境变量：在高级系统里找到环境变量->系统变量->在 path 添加 condabin 路径
3. 创建实验的 python 环境：conda create --name datacenter python=3.6

   关于 conda 和 pip 安装包的一些问题：

   1）用户环境 pip 安装，非 conda 环境下全局使用。

   2）conda 某个环境下 pip 安装，仅能在该虚拟环境下使用。

   3）用户环境 conda 安装，在 conda 共享包目录下存放，base 环境可以直接使用，用户环境不能使用，其它虚拟环境安装同样的包时，先在共享包目录下寻找。

   4）conda 某个环境下 conda 安装，首先会统一放到一个共享目录，然后复制到该环境的 site-packages 文件下。

服务端 Minio

1. 下载最新版 Minio：https://min.io/download
2. 将 minio.exe 移动到服务端的目录
3. 设置 Minio 环境变量，方便网页客户端的登录

(临时环境变量，当时 cmd 环境起作用)

```
set MINIO_ROOT_USER=hust
set MINIO_ROOT_PASSWORD=hust_obs
```

(用户环境变量，当前用户环境起作用)

```
setx MINIO_ROOT_USER hust
setx MINIO_ROOT_PASSWORD hust_obs
```

(系统环境变量，系统环境起作用，需要管理员权限执行命令)

```
setx /m MINIO_ROOT_USER hust
```

setx /m MINIO_ROOT_PASSWORD hust_obs

4. 启动 server 服务

minio -C ./ server ./root - console-address :9090

(-C 参数指定配置文件存储的路径，“./”表示当前路径，server 命令启动服务，后面跟着数据存放的目录，所有桶和对象都存储在当前目录的 root 的目录下)

5. 访问 minio 客户端，server 启动输出的任意一个网页 API 地址都可以访问

账号和密码在启动信息中显示，也就是之前设置的 hust，hust_obs

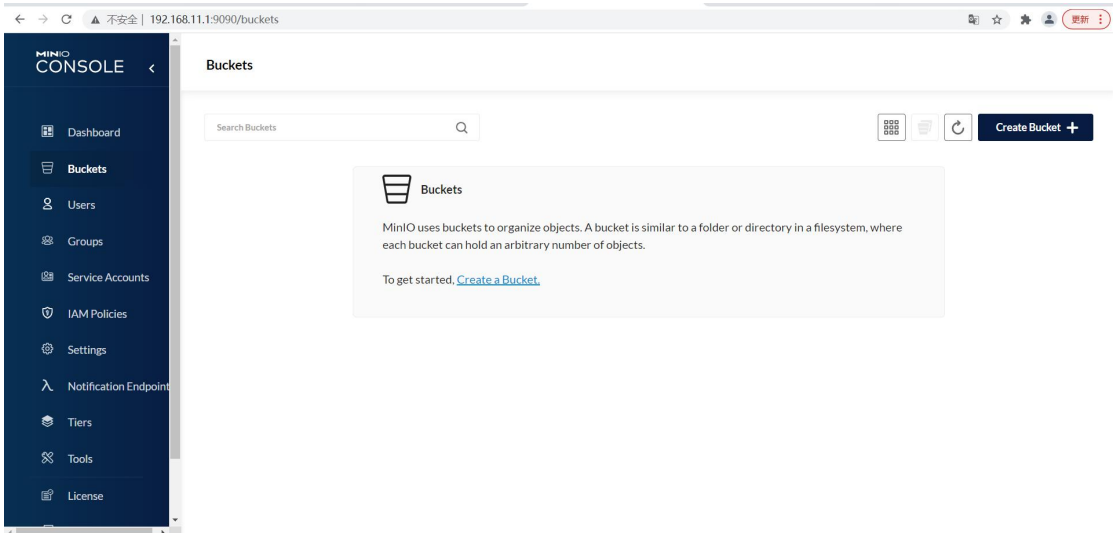
实验二：性能观测

评测工具

选择 S3 Bench，主要原因是操作笔记方便，命令简单，执行命令脚本：

```
s3bench.exe ^
    -accessKey=hust ^
    -accessSecret=hust_obs ^
    -bucket=loadgen ^
    -endpoint=http://127.0.0.1:9000 ^
    -numClients=8 ^
    -numSamples=256 ^
    -objectNamePrefix=loadgen ^
    -objectSize=1024
```

在执行脚本之前，要先创建我们的存储桶，不然会全部报错。创建桶的操作通过 Minio 客户端完成，在网站上访问客户端 API，输入用户密码登录，在控制台中 Buckets 界面创建。要注意我们创建的桶的名称要和脚本中桶的名称保持一致。



创建了桶后，我们就可以运行脚本了。

```
Running Write test...

Running Read test...

Test parameters
endpoint(s):      [http://127.0.0.1:9000]
bucket:           loadgen
objectNamePrefix: loadgen
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:    1.464 s
Number of Errors:  0
------------------------------------
Write times Max:       0.154 s
Write times 99th %ile: 0.144 s
Write times 90th %ile: 0.081 s
Write times 75th %ile: 0.061 s
Write times 50th %ile: 0.035 s
Write times 25th %ile: 0.022 s
Write times Min:       0.015 s


Results Summary for Read Operation(s)
Total Transferred: 0.250 MB
Total Throughput:  2.30 MB/s
Total Duration:    0.109 s
Number of Errors:  0
------------------------------------
Read times Max:       0.008 s
Read times 99th %ile: 0.006 s
Read times 90th %ile: 0.004 s
Read times 75th %ile: 0.004 s
Read times 50th %ile: 0.003 s
Read times 25th %ile: 0.003 s
Read times Min:       0.000 s


Cleaning up 256 objects...
Deleting a batch of 256 objects in range {0, 255}... Succeeded
Successfully deleted 256/256 objects in 728.2294ms
```

实验三：尾延迟挑战

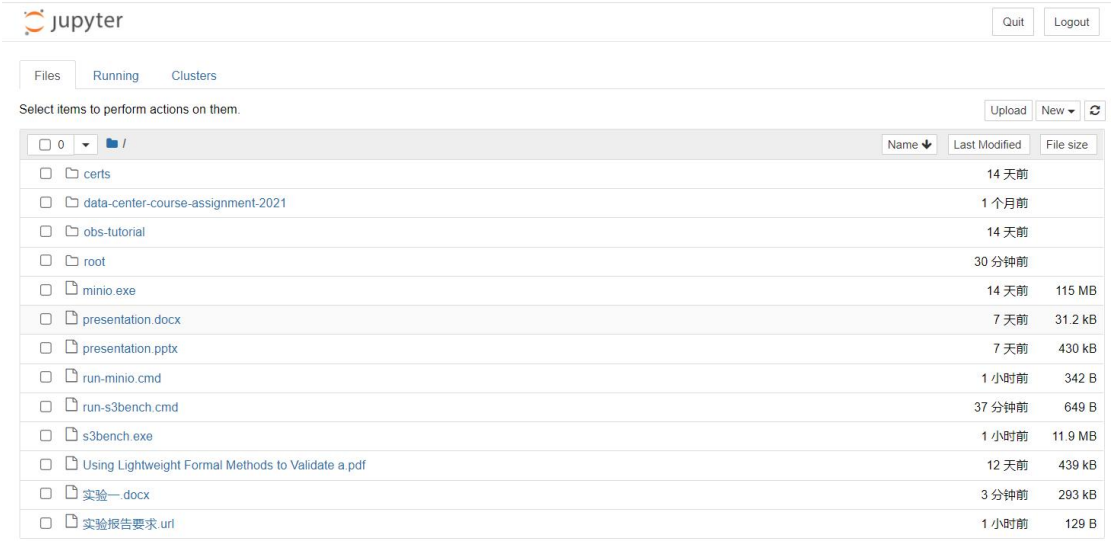
1. 工具介绍

Jupyter Notebook 是一个交互式笔记本，支持运行 40 多种编程语言。Jupyter Notebook 的本质是一个 Web 应用程序，便于创建和共享文学化程序文档，支持实时代码，数学方程，可视化和 markdown。它的主要用途是：数据清理和转换，数值模拟，统计建模，机器学习等，Jupyter Notebook 与 IPython 终端 共享同一个内核。

我们先进入实验的 python 环境，再通过 pip 命令下载 Jupyter Notebook 应用程序。代码如下：

```
C:\Users\11783\Desktop\数据中心技术>conda activate datacenter

(datacenter) C:\Users\11783\Desktop\数据中心技术>pip install jupyter
```

然后在命令行输入 jupyter notebook 打开应用程序。



在 Jupyter Notebook 上创建 python 文件时，可以选择内核环境，只有一个默认的 python3，没有用 conda 创建的实验环境。为了使用 datacenter 的 python 环境，我们还要安装一个 nb_conda 插件，然后重启 Jupyter Notebook 就可以切换到 conda 环境。但是要注意，nb_conda 和 python 版本是否冲突，实验证明 3.9 以上的 python 环境无法安装。由于我的系统环境是 3.10，conda base 环境是 3.9，都无法安装，所以我只能在 conda datacenter(python=3.6) 环境下安装 nb_conda，也只有在 datacenter 环境启动 Jupyter Notebook 才会显示 conda 虚拟环境。

2. 编写程序记录延迟

In [1]:
```python
import os
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
from boto3.session import Session
from tqdm import tqdm
import throttle

# 准备密钥
aws_access_key_id = 'hust'
aws_secret_access_key = 'hust_obs'

# 本地S3服务地址
local_s3 = 'http://192.168.48.1:9000'

# 建立会话
session = Session(aws_access_key_id = aws_access_key_id, aws_secret_access_key=aws_secret_access_key)

# 连接到服务
s3 = session.resource('s3', endpoint_url=local_s3)
```

In [2]:
```python
for bucket in s3.buckets.all():
    print('bucket name:%s' % bucket.name)
```

```
bucket name:loadgen
```

In [3]:
```python
bucket_name = 'test100objs' # bucket name中不能有下划线
if s3.Bucket(bucket_name) not in s3.buckets.all():
    s3.create_bucket(Bucket=bucket_name)
bucket = s3.Bucket(bucket_name)
for obj in bucket.objects.all(): # 若之前实验没有正常结束，则不为空
    print('obj name:%s' % obj.key)
```

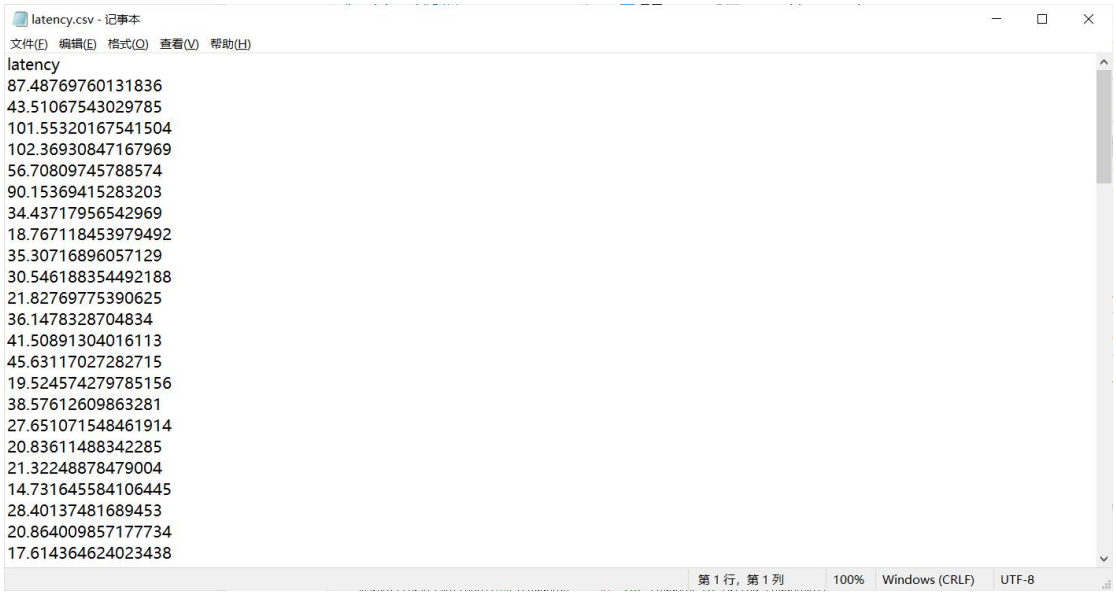
In [4]:
```python
# 初始化本地数据文件
local_file = "_test_4k.bin"
test_bytes = [0xFF for i in range(1024*4)] # 填充至所需大小

with open(local_file, "wb") as lf:
    lf.write(bytearray(test_bytes))

# 发起请求和计算系统停留时间
def request_timing(s3res, i): # 使用独立 session.resource 以保证线程安全
    obj_name = "testObj%08d"%(i,) # 所建对象名
    service_time = 0 # 系统滞留时间
    start = time.time()
    s3res.Object(bucket_name, obj_name).upload_file(local_file) # 将本地文件上传为对象
    end = time.time()
    system_time = end - start
    return system_time * 1000 # 换算为毫秒

# 按照请求到达率限制来执行和跟踪请求
def arrival_rate_max(s3res, i): # 不进行限速
    return request_timing(s3res, i)

@throttle.wrap(0.1, 2) # 100ms 内不超过2个请求， 下同.....
def arrival_rate_2(s3res, i):
    return request_timing(s3res, i)

@throttle.wrap(0.1, 4)
def arrival_rate_4(s3res, i):
    return request_timing(s3res, i)

@throttle.wrap(0.1, 8)
def arrival_rate_8(s3res, i):
    return request_timing(s3res, i)
```

In [5]:
```python
# 按照预设IAT发起请求
latency = []
failed_requests = []

with tqdm(desc="Accessing S3", total=100) as pbar:      # 进度条设置，合计执行 100 项上传任务（见 submit 部分），进度也设置为 100 步
    with ThreadPoolExecutor(max_workers=1) as executor: # 通过 max_workers 设置并发线程数
        futures = [
            executor.submit(
                arrival_rate_max,
                session.resource('s3', endpoint_url=local_s3), i) for i in range(100) # 为保证线程安全，应给每个任务申请一个新 resource
            ]
        for future in as_completed(futures):
            if future.exception():
                failed_requests.append(futures[futures.index(future)])
            else:
                latency.append(future.result()) # 正确完成的请求，采集延迟
            pbar.update(1)
```

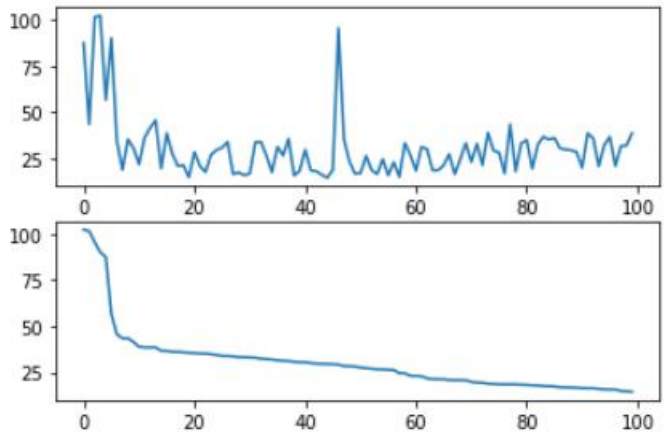
```
Accessing S3: 100%|██████████| 100/100 [00:03<00:00, 30.68it/s]
```

In [6]:
```python
# 清理实验环境
try:
    # 删除bucket下所有object
    bucket.objects.filter().delete()

    bucket.delete()
except botocore.exceptions.ClientError as e:
    print('error in bucket removal')

# 删除本地测试文件
os.remove(local_file)

# 记录延迟到CSV文件
with open("latency.csv", "w+") as tracefile:
    tracefile.write("latency\n")
    tracefile.writelines([str(la) + '\n' for la in latency])
with open("failed_request.csv", "w+") as tracefile:
    tracefile.write("failed_requests\n")
    tracefile.writelines([str(request) + '\n' for request in failed_requests])
```

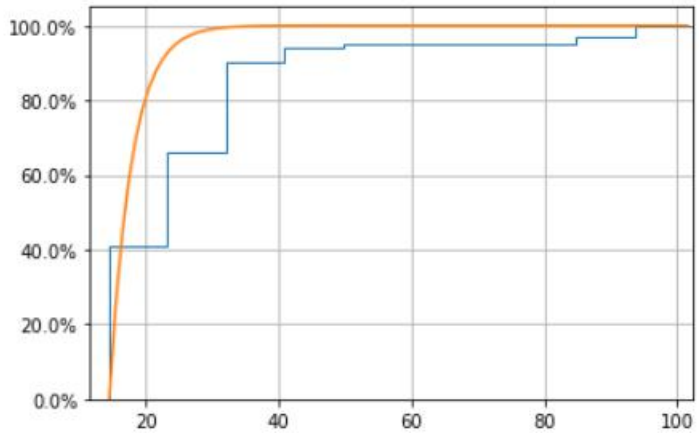
延迟结果和请求出错结果都存在本地的 csv 文件中，如图所示：



为了更明显地观察尾延迟现象，画出延迟的分布情况：



下面是排队论模型拟合曲线：



3. 尝试减轻尾延迟现象

刚开始打算启动一个新的线程来请求上传文件，写一个循环监测请求时间，如果延迟超过 50ms，则再启动一个线程重新发起请求，然而线程启动时间有一百毫秒以上，远远大于程序的最大延迟，所有不能用启动新线程的方法来消除尾延迟现象。于是就不知道怎么办了，既然不能启动新的线程，那么请求就只能是顺序执行，如果尾延迟的最大值要远远大于启动新线程耗费的时间，那么可以考虑重新启动一个线程去解决。