对象存储

(Object-Oriented Storage,OOS)

Python SDK 开发者指南 V6.0

中国电信股份有限公司

云计算分公司

目录

1. 对象存储（OOS）

对象存储（Object-Oriented Storage，OOS）是中国电信为客户提供的一种海量、弹性、廉价、高可用的存储服务。客户只需花极少的钱就可以获得一个几乎无限的存储空间，可以随时根据需要调整对资源的占用，并只需为真正使用的资源付费。

OOS 提供了基于 Web 门户和基于 HTTP REST 接口两种访问方式，用户可以在任何地方通过互联网对数据进行管理和访问。OOS 提供的 REST 接口与 Amazon S3 兼容，因此基于 OOS 的业务可以非常轻松的与 Amazon S3 对接。

2. 安装和使用

2.1 先决条件

您需要先获得以下项，然后才能使用 OOS Python SDK:

- 一个 OOS 账户

2.2 获取 OOS 凭证

AccessKeyId 和 SecretKey 是您访问 OOS 的密钥，OOS 会通过它来验证您的资源请求，请妥善保管。关于 AccessKeyId 和 SecretKey 的介绍，请参见开发者文档。

3. 安装和配置

3.1 下载 Python SDK

从官方渠道下载 OOS-Python-SDK 压缩包，解压后执行标准 python 包安装命令

```
python setup.py install
```

如果是在 python3 环境下使用，请确保已安装 urllib3。其他相关的组件有 jmespath、python-dateutil、futures，在 setup.py 中都有包含。

安装完成后在 examples 目录下有 examples.py 的详细示例，各个接口在示例中均有方法对应，请参照使用。

3.2 引入工程

在您的代码中使用 import 引入 OSS Python SDK 的包，并初始化连接。

```
importoos
from ooscore.client import Config


ENDPOINT = 'http://oos-cn.ctyunapi.cn'
IAM_ENDPOINT = 'http://oos-cn-iam.ctyunapi.cn'
ACCESS_KEY = 'fe49633483061d5b93e3'
SECRET_KEY = 'eba77e4fcdd33dbd76b4175e7e37d3136e503afd'

#SIGNATURE_VERSION = 's3'
SIGNATURE_VERSION = 'v4'
config = Config(signature_version=SIGNATURE_VERSION)
client = oos.client('s3', use_ssl=False, endpoint_url=ENDPOINT,
access_key_id=ACCESS_KEY, secret_access_key=SECRET_KEY,
config=config)
iam_client = oos.client('sts', use_ssl=False, endpoint_url=IAM_ENDPOINT,
access_key_id=ACCESS_KEY, secret_access_key=SECRET_KEY,
config=config)
```

4. 代码示例

4.1 Service 操作

4.1.1 Get Service (List Bucket)

对于做 Get 请求的服务，返回请求者拥有的所有 Bucket，其中“/”表示根目录。

```
# Service: 4.1.1 List Bucket
@handle_error
def list_bucket_example():
    response = client.list_buckets()
    pretty_print(response)
for bucket in response['Buckets']:
print('Bucket: {0}'.format(bucket['Name']))


    pretty_print(response)
```

4.1.2 Get Regions

获取资源池中的索引位置和数据位置列表。

```
# Bucket: 4.1.2 Get Regions
@handle_error
def get_regions_example():
print('Get Regions')
    response = client.get_regions()
    pretty_print(response)
```

4.2 Bucket 操作

4.2.1 Put/Delete Bucket

Put 操作用来创建一个新的 bucket。只有在 OOS 中注册的用户才能创建一个新的 bucket，匿名请求无效，创建 bucket 的用户将是 bucket 的拥有者。

Bucket 的命名方式中并不是支持所有的字符，具体请参见 bucket 命名规范。

```
# Bucket: 4.2.1 Put Bucket
@handle_error
def put_bucket_example():
print('Creating OOS bucket')
    response = client.create_bucket(
ACL='public-read-write',
Bucket=BUCKET,
CreateBucketConfiguration={
'MetadataLocationConstraint':
              {'Location': 'ChengDu'},
'DataLocationConstraint': {
'Type': 'Specified',
'LocationList': ['ChengDu'],
'ScheduleStrategy': 'Allowed'}
       }
    )
    pretty_print(response)


# Bucket: 4.2.5 Delete bucket
# 删除 Bucket 之前必须要清空
@handle_error
def delete_bucket_example():
print('Delete Bucket')
    response = client.delete_bucket(
Bucket=BUCKET
 )
    pretty_print(response)
```

4.2.2 GetBucket Location

这个 Get 操作用来获取 bucket 的索引位置和数据位置，只有 bucket 的所有者才能执行此操作。

```
# Bucket: 4.2.2 Get Bucket location
@handle_error
def get_bucket_location():
print('Get Bucket location')
    response = client.get_bucket_location(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.3 GET Bucket (List Objects)

这个 Get 操作返回 bucket 中部分或者全部（最多 1000）的 object 信息。用户可以在请求元素中设置选择条件来获取 bucket 中的 object 的子集。

```
# Bucket: 4.2.4 Get Bucket (List Objects)
@handle_error
def get_bucket_example():
print('List Objects')
    response = client.list_objects(
Bucket=BUCKET,
Marker='test',
MaxKeys=10,
    )
    pretty_print(response)
```

4.2.4 Get Bucket ACL

这个 Get 操作用来获取 bucket 的 ACL 信息，用户必须对改 bucket 有读权限。

```
# Bucket: 4.2.5 Delete bucket
# 删除 Bucket 之前必须要清空
@handle_error
def delete_bucket_example():
print('Delete Bucket')
    response = client.delete_bucket(
Bucket=BUCKET
    )
```

4.2.5 Put/Get/Delete Bucket Policy

如果 bucket 已经存在了 Policy，此操作会替换原有 Policy。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。如果 bucket 没有 policy，Get 时返回 404，NoSuchPolicy 错误。

```
# Bucket: 4.2.6 Put Bucket Policy
@handle_error
def put_bucket_policy_example():
print('Put Bucket policy')
    policy = {
"Version": "2012-10-17",
"Id": "aaaa-bbbb-cccc-dddd",
"Statement": [
          {
"Effect": "Allow",
"Sid": "1",
"Principal": {
"AWS": "*"
},
"Action": ["s3:*"],
"Resource": "arn:aws:s3:::test-jf/*"
}
       ]
    }
    response = client.put_bucket_policy(
Bucket=BUCKET,
Policy=json.dumps(policy)
    )
    pretty_print(response)


# Bucket: 4.2.7 Get Bucket Policy
@handle_error
def get_bucket_policy_example():
print('Get Bucket policy')
    response = client.get_bucket_policy(
Bucket=BUCKET
    )
    pretty_print(response)
    pretty_print(json.loads(response['Policy']))


# Bucket: 4.2.8 Delete bucket Policy
@handle_error
def delete_bucket_policy_example():
print('Delete Bucket policy')
    response = client.delete_bucket_policy(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.6 Put/Get/Delete Bucket WebSite

如果 bucket 已经存在了 website，此操作会替换原有 website。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。

WebSite 功能可以让用户将静态网站存放到 OOS 上。对于已经设置了 WebSite 的 Bucket，当用户访问 http://bucketName.oos-website-cn.oos.ctyunapi.cn 时，会跳转到用户指定的主页，当出现 4**错误时，会跳转到用户指定的出错页面。

```
# Bucket: 4.2.9 Put Bucket WebSite
@handle_error
def put_bucket_website_example():
print('Put Bucket Website')
    response = client.put_bucket_website(
Bucket=BUCKET,
WebsiteConfiguration={
'ErrorDocument': {
'Key': 'error1.html'
},
'IndexDocument': {
'Suffix': 'index1.html'
}
        }
    )
    pretty_print(response)


# Bucket: 4.2.10 Get Bucket WebSite
@handle_error
def get_bucket_website_example():
print('Get Bucket Website')
    response = client.get_bucket_website(
Bucket=BUCKET
    )
    pretty_print(response)


# Bucket: 4.2.11 Delete Bucket WebSite
@handle_error
def delete_bucket_website_example():
print('Delete Bucket Website')
    response = client.delete_bucket_website(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.7 Put/Get/Delete Bucket LifeCycle

存储在 OOS 中的对象有时需要有生命周期。比如，用户可能上传了一些周期性的日志文件到 bucket 中，一段时间后，用户可能不需要这些日志对象了，可以使用对象到期功能来指定 bucket 中对象的生命周期。

当对象的生命周期结束时，OOS 会异步删除他们。生命周期中配置的到期时间和实际删除时间之间可能会有一段延迟。对象到期后，用户将不会再为到期的对象付费。

用户可以通过在 bucket 中配置生命周期，来为对象设置到期时间。一个生命周期的配置中最多可以包含 100 条规则。每个规则指定了对象的前缀和生命周期，生命周期是指从对象创建开始到被删除之前的天数。生命周期的值必须是个正整数。OOS 通过将对象的创建时间加上生命周期时间来计算到期时间，并且将时间近似到下一天的 GMT 零点时间。比如，一个对象是 GMT 2016 年 1 月 15 日 10:30 分创建的，生命周期是 3 天，那么对象的到期时间是 GMT 2016 年 1 月 19 日 00:00。当重写一个对象时，OOS 将以最后更新时间为准，来重新计算到期时间。

当用户为 bucket 设置了生命周期时，这些规则将同时应用于已有对象和之后新创建的对象。比如，用户今天增加了一个生命周期的配置，指定某些前缀的对象 30 天后过期，那么 OOS 将会把满足条件的 30 天前创建的对象都加入到待删除队列中。

用户可以使用 GET, HEAD API 来查询对象的到期时间，这些接口会通过响应头来返回对象的到期时间信息。

Put Bucket Lifecycle 接口用于设置 bucket 的生命周期，如果生命周期的配置已经存在，将会被替换。用户可以通过设置生命周期，来让 OOS 删除过期的对象。

```
# Bucket: 4.2.16 Put Bucket Lifecycle
@handle_error
def put_bucket_lifecycle_example():
print('Put Bucket Lifecycle')
# 10 天后到期，用 Date 方式演示 python sdk 中整点时间处理
d = datetime.date.today() + datetime.timedelta(days=10)
    expire_date = datetime.datetime.combine(d, datetime.datetime.min.time())
    response = client.put_bucket_lifecycle(
Bucket=BUCKET,
LifecycleConfiguration={
'Rules': [
             {
'Expiration': {
'Date': expire_date
                 },
'ID': 'lifecycle-123',
'Prefix': 'test',
'Status': 'Enabled'
}
          ]
       }
    )
    pretty_print(response)


# Bucket: 4.2.17 Get Bucket Lifecycle
@handle_error
def get_bucket_lifecycle_example():
print('Get Bucket Lifecycle')
    response = client.get_bucket_lifecycle(
Bucket=BUCKET
    )
    pretty_print(response)


# Bucket: 4.2.18 Delete Bucket Lifecycle
@handle_error
def delete_bucket_lifecycle_example():
print('Delete Bucket Lifecycle')
    response = client.delete_bucket_lifecycle(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.8 Put/Get Bucket logging

如果 bucket 已经存在了 logging，Put 操作会替换原有 logging。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。日志名称：TargetPrefixGMTYYYY-mm-DD-HH-MM-SS-唯一字符串。其中 YYYY，mm，DD，HH，MM，SS 分别代表日志发送时的年，月，日，小时，分钟，秒。TargetPrefix 是用户在启动 Bucket 日志时配置的。唯一字符串部分没有实际含义，可以被忽略。

系统不会删除旧的日志文件。用户可以自己删除以前的日志文件，可以在 List Object 时指定 prefix 参数，挑选出旧的日志文件，然后删除。

当客户端的请求到达后，日志记录不会被立刻推送到 TargetBucket 中，会延迟一段时间。

```
# Bucket: 4.2.13 Put Bucket Logging
@handle_error
def put_bucket_logging_example():
print('Put Bucket Logging')
    response = client.put_bucket_logging(
Bucket=BUCKET,
BucketLoggingStatus={
'LoggingEnabled': {
'TargetBucket': '{0}-logging'.format(BUCKET),
'TargetPrefix': '{0}_log'.format(BUCKET)
            }
        }
    )
    pretty_print(response)


# Bucket: 4.2.14 Get Bucket Logging
@handle_error
def get_bucket_logging_example():
print('Get Bucket Logging')
    response = client.get_bucket_logging(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.9 Head Bucket

此操作用于判断 bucket 是否存在，而且用户是否有权限访问。如果 bucket 存在，而且用户有权限访问时，此操作返回 200 OK。否则，返回 404 不存在，或者 403 没有权限。

```
# Bucket: 4.2.15 Head Bucket
@handle_error
def head_bucket_example():
print('Head Bucket')
    response = client.head_bucket(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.10 Put/Get Bucket Accelerate

在 PUT 操作的 url 中加上 accelerate，可以进行添加或修改 CDN IP 白名单的操作。如果 bucket 已经配置了 CDN 加速，此操作会替换原有配置。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。一个 bucket 最多配置 5 个 IP 白名单地址段。

在 GET 操作的 url 中加上 accelerate，可以获得指定 bucket 的 cdn 配置信息。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。如果 bucket 没有配置过 CDN 加速，那么将不会返回状态信息。

```
# Bucket: 4.2.19 Put Bucket accelerate
@handle_error
def put_bucket_accelerate_example():
print('Put Bucket accelerate')
 response = client.put_bucket_accelerate(
Bucket=BUCKET,
AccelerateConfiguration={
'Status': 'Enabled',
'IPWhiteLists': ['36.111.88.0/24', '114.80.1.136']
       }
   )
   pretty_print(response)


# Bucket: 4.2.20 Get Bucket accelerate
@handle_error
def get_bucket_accelerate_example():
print('Get Bucket accelerate')
   response = client.get_bucket_accelerate(
Bucket=BUCKET
   )
   pretty_print(response)
```

4.2.11 Put/Get/Delete Bucket Cors

跨域资源共享（Cross-Origin Resource Sharing，CORS）定义了客户端 Web 应用程序在一个域中与另一个域中的资源进行交互的方式，是浏览器出于安全考虑而设置的一个限制，即同源策略。例如，当来自于 A 网站的页面中的 JavaScript 代码希望访问 B 网站的时候，浏览器会拒绝该访问，因为 A、B 两个网站是属于不同的域。

通过 CORS ，客户可以构建丰富的客户端 Web 应用程序，同时可以选择性地允许跨域访问 OOS 资源。

```
# Bucket: 4.2.21 Put Bucket cors
@handle_error
def put_bucket_cors_example():
print('Put Bucket cors')
    response = client.put_bucket_cors(
Bucket=BUCKET,
CORSConfiguration={
'CORSRules': [
                {
'AllowedOrigins': ['http://www.example1.com'],
'AllowedMethods': ['PUT', 'POST', 'DELETE'],
'AllowedHeaders': ['*']
                },
                {
'AllowedOrigins': ['http://www.example2.com'],
'AllowedMethods': ['PUT', 'POST'],
'AllowedHeaders': ['*']
                }
            ]
        }
    )
    pretty_print(response)


# Bucket: 4.2.22 Get Bucket cors
@handle_error
def get_bucket_cors_example():
print('Get Bucket cors')
    response = client.get_bucket_cors(
Bucket=BUCKET
    )
    pretty_print(response)


# Bucket: 4.2.23 Delete Bucket cors
@handle_error
def delete_bucket_cors_example():
print('Delete Bucket cors')
    response = client.delete_bucket_cors(
Bucket=BUCKET
    )
    pretty_print(response)
```

4.3 Object 操作

4.3.1 Put Object

Put 操作用来向指定 bucket 中添加一个对象，要求发送请求者对该 bucket 有写权限，用户必须添加完整的对象。

```
# Object: 4.3.1 Put Object
# 上传本地文件到 OOS bucket
@handle_error
def put_object_example():
print('Uploading OOS object')
with open(UPLOAD_FILE, 'rb') as data:
       client.put_object(
DataLocation='type=Specified,location=ChengDu,scheduleStrategy=Allowed',
Bucket=BUCKET,
Key=KEY,
Body=data,
StorageClass='EC_2_1'  # STANDARD | REDUCED_REDUNDANCY | EC_N_M
)
print('Done')
```

4.3.2 Get Object

GET 操作用来检索在 OOS 中的对象信息，执行 GET 操作，用户必须对 object 所在的 bucket 有读权限。如果 bucket 是 public read 的权限，匿名用户也可以通过非授权的方式进行读操作。

```
# Object: 4.3.2 Get Object
# 下载 OOS bucket 文件到本地，注意设置读取 response 流块大小，减轻内存负载
@handle_error
def get_object_example():
print('Downloading OOS object')
    response = client.get_object(
Bucket=BUCKET,
Key=KEY
    )
    pretty_print(response)
    body = response['Body']
with open(DOWNLOAD_FILE_PATH, 'wb') as fd:
for chunk in iter(lambda: body.read(4096), b''):
            fd.write(chunk)
print('Done')
```

4.3.3 Delete Object

Delete 操作用来删除在 OOS 中的对象，执行 GET 操作，用户必须对 object 所在的 bucket 有写权限。Delete 操作支持一次性批量删除多个对象。批量删除 Object 功能支持用一次请求删除一个 bucket 中的多个 object。如果你知道你想删除的 object 名字，此功能可以批量删除这些 object，而不用发送多个单独的删除请求。

对于每个 Object，OOS 都会返回删除的结果，成功或者失败。注意，如果请求中的 object 不存在，那么 OOS 也会返回删除成功。

批量删除功能支持两种格式的响应，全面信息和简明信息。默认情况下，OOS 在响应中会显示全面信息，即包含每个 object 的删除结果。在简明信息模式下，OOS 只返回删除出错的 object 的结果。对于成功删除的 object，在响应中将不返回任何信息。

```
# Object: 4.3.3 Delete Object
@handle_error
def delete_object_example():
print('Delete object')
    response = client.delete_object(
Bucket=BUCKET,
Key=KEY
    )
    pretty_print(response)
```

4.3.4 Put Object – Copy

通过 PUT 操作创建一个存储在 OOS 里的对象的拷贝。PUT 操作类似于执行一个 GET 然后在执行一次 PUT。增加请求头，x-awz-copy-source，使用 PUT 操作将源对象存入指定 bucket。要执行拷贝请求，用户需要对源对象有读权限，对目标 bucket 有写权限。

```
# Object: 4.3.4 Put Object-Copy
@handle_error
def put_object_copy_example():
print('Copy object')
    response = client.copy_object(
Bucket=BUCKET,
CopySource={'Bucket': BUCKET, 'Key': KEY},
Key='{0}-Copy'.format(KEY),
DataLocation='type=Specified,location=ChengDu,scheduleStrategy=Allowed'
)
    pretty_print(response)
```

4.3.5 Initial Multipart Object

本接口初始化一个分片上传（Multipart Upload）操作，并返回一个上传 ID，此 ID 用来将此次分片上传操作中上传的所有片段合并成一个对象。用户在执行每一次子上传请求（见 Upload Part）时都应该指定该 ID。用户也可以在表示整个分片上传完成的最后一个请求中指定该 ID。或者在用户放弃该分片上传操作时指定该 ID。

```
@handle_error
def multipart_upload_example():
print('Multipart Upload')
    upload = client.create_multipart_upload(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
DataLocation='type=Specified,location=ChengDu,scheduleStrategy=Allowed'
)
    pretty_print(upload)
    MultipartUpload = {
'Parts': []
    }
with open(UPLOAD_FILE, 'rb') as fd:
        num = 0
for chunk in iter(lambda: fd.read(1024 * 1024), b''):
            num += 1
part = client.upload_part(
Body=chunk,
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
PartNumber=num,
UploadId=upload['UploadId']
            )
          MultipartUpload['Parts'].append({
'PartNumber': num,
'ETag': part['ETag']
            })
            pretty_print(part)
print('List Parts')
    list = client.list_parts(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
MaxParts=10,
UploadId=upload['UploadId']
    )
    pretty_print(list)
    complete = client.complete_multipart_upload(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
UploadId=upload['UploadId'],
MultipartUpload=MultipartUpload
    )
    pretty_print(complete)
print('Done')
```

4.3.6 Upload Part

该接口用于实现分片上传操作中片段的上传。

在上传任何一个分片之前，必须执行 Initial Multipart Upload 操作来初始化分片上传操作，初始化成功后，OOS 会返回一个上传 ID，这是一个唯一的标识，用户必须在调用 Upload Part 接口时加入该 ID。

分片号 PartNumber 可以唯一标识一个片段并且定义该分片在对象中的位置，范围从 1 到 10000。如果用户用之前上传过的片段的分片号来上传新的分片，之前的分片将会被覆盖。

除了最后一个分片外，所有分片的大小都应该不小于 5M，最后一个分片的大小不受限制。

为了确保数据不会由于网络传输而毁坏，需要在每个分片上传请求中指定 Content-MD5 头，OOS 通过提供的 Content-MD5 值来检查数据的完整性，如果不匹配，则会返回一个错误信息。

代码参见 4.3.5

4.3.7 Complete Multipart Upload

该接口通过合并之前的上传片段来完成一次分片上传过程。

用户首先初始化分片上传过程，然后通过 Upload Part 接口上传所有分片。在成功将一次分片上传过程的所有相关片段上传之后，调用这个接口来结束分片上传过程。当收到这个请求的时候，OOS 会以分片号升序排列的方式将所有片段依次拼接来创建一个新的对象。在这个 Complete Multipart Upload 请求中，用户需要提供一个片段列表。同时，必须确保这个片段列表中的所有片段必须是已经上传完成的，Complete Multipart Upload 操作会将片段列表中提供的片段拼接起来。对片段列表中的每个片段，需要提供该片段上传完成时返回的 ETag 头的值和对应的分片号。

代码参见 4.3.5

4.3.8 Abort Multipart Upload

该接口用于终止一次分片上传操作。分片上传操作被终止后，用户不能再通过上传 ID 上传其它片段，之前已上传完成的片段所占用的存储空间将被释放。如果此时任何片段正在上传，该上传过程可能会也可能不会成功。所以，为了释放所有片段所占用的存储空间，可能需要多次终止分片上传操作。

```
# Object: 4.3.8 Abort Multipart Upload
@handle_error
def abort_multipart_example():
print('Abort Multipart')
    upload = client.create_multipart_upload(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
DataLocation='type=Specified,location=ChengDu,scheduleStrategy=Allowed'
)
    pretty_print(upload)

    response = client.abort_multipart_upload(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
UploadId=upload['UploadId']
    )
    pretty_print(response)
```

4.3.9 List Part

该操作用于列出一次分片上传过程中已经上传完成的所有片段。

该操作必须包含一个通过 Initial Multipart Upload 操作获取的上传 ID。该请求最多返回 1000 个上传片段信息，默认返回的片段数是 1000。用户可以通过指定 max-parts 参数来指定一次请求返回的片段数。如果用户的分片上传过程超过 1000 个片段，响应中的 IsTruncated 字段的值则被设置成 true，并且指定一个 NextPartNumberMarker 元素。用户可以在下一个连续的 List Part 请求中加入 part-number-marker 参数，并把它的值设置成上一个请求返回的 NextPartNumberMarker 值。

代码参见 4.3.5

4.3.10 Copy Part

可以将已经存在的 object 作为分段上传的片段，拷贝生成一个新的片段。需要指定请求头 x-amz-copy-source 来定义拷贝源。如果想拷贝源 object 中的一部分，可以加请求头 x-amz-copy-source-range

```
# Object: 4.3.10 Copy Part
@handle_error
def copy_part_example():
print('Copy Part')
    upload = client.create_multipart_upload(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY
    )
    pretty_print(upload)

    part = client.upload_part_copy(
Bucket=BUCKET,
CopySource={'Bucket': BUCKET, 'Key': KEY},
# CopySourceRange='bytes=0-9',
Key=MULTIPART_UPLOAD_KEY,
PartNumber=1,
UploadId=upload['UploadId']
    )
    pretty_print(part)

    MultipartUpload = {
'Parts': [
            {'PartNumber': 1,
'ETag': part['CopyPartResult']['ETag']}
        ]
    }

    complete = client.complete_multipart_upload(
Bucket=BUCKET,
Key=MULTIPART_UPLOAD_KEY,
UploadId=upload['UploadId'],
MultipartUpload=MultipartUpload
    )
    pretty_print(complete)
print('Done')
```

4.3.11 Delete Multiple Objects

批量删除 Object 功能支持用一个 HTTP 请求删除一个 bucket 中的多个 object。如果你知道你想删除的 object 名字，此功能可以批量删除这些 object，而不用发送多个单独的删除请求。

```
# Object: 4.3.12 Delete Multiple Objects
@handle_error
def delete_multiple_example():
print('Delete Multiple Objects')
    response = client.delete_objects(
Bucket=BUCKET,
Delete={
'Objects': [
                {
'Key': 'test'
},
                {
'Key': 'test-Copy'
}
            ]
        }
    )
    pretty_print(response)
```

4.3.12 生成共享链接

对于私有或只读 Bucket，可以通过生成 Object 的共享链接的方式，将 Object 分享给其他人，同时可以在链接中设置限速以对下载速度进行控制。在 SDK 中调用 AmazonS3 中的 generatePresignedUrl(String bucketName, String key, Date expiration)方法，可以生成共享链接 URL。参数分别是 Bucket 名称，Object 名称和过

期时间，如果过期时间传 Null 的话，默认的过期时间是 15 分钟。超出过期时间后，共享链接失效，不能再通过链接下载 Object。

```
# Object 4.3.16 Generate Shared link
@handle_error
def generate_shared_link_example():
print('Generate Shared link')
    shared_link = client.generate_presigned_url(
ClientMethod='get_object',
Params={
'Bucket': BUCKET,
'Key': KEY
        }
    )
print(shared_link)
```

4.3.13 Head Object

Head 操作用于获取对象的元数据信息，而不返回数据本身。当只希望获取对象的属性信息时，可以使用此操作。

```
# Object 4.3.17 Head Object
@handle_error
def head_object_example():
print('Head Object')
    response = client.head_object(
Bucket=BUCKET,
Key=KEY
    )
    pretty_print(response)
```

4.4 AccessKey 操作

4.4.1 Create AccessKey

创建一对普通的 AccessKey 和 SecretKey，默认的状态是 Active。只有主 key 才能执行此操作。

为保证账户的安全，SecretKey 只在创建的时候会被显示。请把 key 保存起来，比如保存到一个文本文件中。如果 SecretKey 丢失了，你可以删除 AccessKey，并创建一对新的 key。默认情况下，每个账号最多创建 10 个 AccessKey。

```
# AccessKey: 4.4.1 Create Access Key
@handle_error
def create_access_key_example():
print('Create access key')
    data = {
"Action": "CreateAccessKey"
}
    body = encode_params(data)
    response = iam_client.create_access_key(
Body=body
    )
    pretty_print(response)
```

4.4.2 Delete AccessKey

删除一对普通的 AccessKey 和 SecretKey。只有主 key 才能执行此操作。

```
# AccessKey: 4.4.2 Delete Access Key
@handle_error
def delete_access_key_example():
print('Delete access key')
    data = {
"Action": "DeleteAccessKey",
"AccessKeyId": "10c2758180defd92e034"
}
    body = encode_params(data)
    response = iam_client.delete_access_key(
Body=body
    )
    pretty_print(response)
```

4.4.3 Update AccessKey

更新普通的 AccessKey 的状态，或将普通 key 设置成为主 key，反之亦然。只有主 key 才能执行此操作。

```
# AccessKey: 4.4.3 Update Access Key
@handle_error
def update_access_key_example():
print('Update access key')
    data = {
"Action": "UpdateAccessKey",
"AccessKeyId": "10c2758180defd92e034",
"Status": "Inactive",  # Active or Inactive
"IsPrimary": False
}
    body = encode_params(data)
    response = iam_client.update_access_key(
Body=body
    )
    pretty_print(response)
```

4.4.4 List AccessKey

列出账号下的主 ke 和普通 key。只有主 key 才能执行此操作。可以通过 MaxItems 参数指定返回的结果数量，默认返回 100 个 key。可以通过 Marker 参数设置返回的起始位置，该参数可以从前一次请求的响应体中获得。为保证账号安全，list 操作时，不会返回 SecretKey。

```
# AccessKey: 4.4.4 List Access Key
@handle_error
def list_access_key_example():
print('List access key')
    data = {
'Action':'ListAccessKey',
'MaxItems': 10
}
    body = encode_params(data)
    response = iam_client.list_access_key(
Body=body
    )
    pretty_print(response)
```