天翼云

对象存储

(Object-Oriented Storage,OOS)

OOS Python SDK 开发者指南 V6

天翼云科技有限公司

目录

1 前言

对象存储（Object-Oriented Storage，OOS）为客户提供一种海量、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。目前 OOS 提供以下四种服务：

- **对象存储（OOS）**：OOS 的服务为对象存储的核心服务，主要包括 bucket 操作管理以及 bucket 的对象存储操作管理，为 OOS 基础服务。
- **统计分析（Management API）**：统计分析指用户可以查询指定 bucket 的使用情况、指定数据域的使用情况。用户可以根据统计分析数据，采取对应的措施，用户可以通过管理控制台或 SDK 获取统计数据。
- **操作跟踪（CloudTrail）**：操作跟踪用于记录 OOS 账户的管理事件，并将产生的跟踪日志保存到指定的 OOS 存储桶中。记录的信息包括用户的身份，请求时间，源 IP 地址，请求参数以及服务返回的响应元素等。
- **访问控制（IAM）**：IAM（Identity and Access Management）是 OOS 为用户提供的用户身份管理与访问控制服务，您可以使用 IAM 创建、管理用户账号，并对这些账号进行权限分配，方便资源管理。

2 使用条件

2.1 先决条件

用户需要具备以下条件才能够使用 OOS SDK Python 版本：

- 一个 OOS 账户。
- 已经安装 Python 3.3 或以上版本。
- 已获取 AccessKeyId 和 SecretKey。
- 熟悉各接口的参数和响应参数的使用方法，详见《OOS 开发者文档》。

2.2 下载及安装

从官方渠道下载 oos-python-sdk-6.5.3.zip 压缩包，放到相应位置后并解压。“python-sdk-6.5.3”目录中“examples”为 sdk 的使用示例代码。

在“python-sdk-6.5.3”目录下执行标准 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。

由于操作跟踪（CloudTrail）服务使用了第三方的 requests 包，请安装 requests 包。

```
# pip install requests
```

3 SDK 使用设置

3.1 基本设置

使用 sdk 访问 OOS 的服务，需要设置正确的 AccessKeyId、SecretKey 和服务端 Endpoint，所有的服务可以使用同一 key 凭证来进行访问，但不同的服务需要使用不同的 endpoint 进行访问。OOS Endpoint 请参见 **Endpoint 列表**。

访问 OOS 的服务示例代码如下：

```
try:
    _config = Config(endpoint_url=ENDPOINT, signature_version=SIGNATURE_VERSION,
                     s3={'payload_signing_enabled': True})
    _iam_config = Config(endpoint_url=IAM_ENDPOINT,
```

```
signature_version=SIGNATURE_VERSION,
                        s3={'payload_signing_enabled': True})

except Exception as ex:
    print(traceback.format_exc())
    print(ex)
    exit(-1)
```

不同的服务需要使用不同的 Endpoint 进行访问，上例进行了对象存储服务（OOS）和访问控制服务（IAM）服务的两个配置设置。

IAM 服务时，设置该服务对应的 Endpoint，代码为：

```
endpoint_url=IAM_ENDPOINT
```

参数说明

参数	描述	是否必须
SERVICE_NAME	服务名，取值为： ● OOS 服务或统计分析服务：s3。 ● 操作跟踪服务：cloudtrail。 ● IAM 服务：sts。	否
ACCESS_KEY	AccessKey。	是
SECRET_KEY	SecretAccessKey。	是
IAM_ENDPOINT	OOS 域名如 https://oos-cn-iam.ctyunapi.cn	是
SIGNATURE_VERSION	计算签名采用的算法版本。 取值： ● s3v4' ● s3'	是
USE_SSL	是否使用 https 取值： ● True ● False	是

REGION_NAME	地域名，取值为 cn。	否、参照该服务的示例代码

3.2 创建 Client 对象

在使用各服务前，请创建该服务对应的 Client 对象，如 OOS 服务创建 OOS Client 的代码如下：

```
try:
_config = Config(endpoint_url=ENDPOINT,
signature_version=SIGNATURE_VERSION,
                    s3={'payload_signing_enabled': True})
    client = oos.client(service_name=SERVICE_NAME, use_ssl=False, endpoint_url=ENDPOINT,
api_version=API_VERSION,
                        access_key_id=ACCESS_KEY, secret_access_key=SECRET_KEY,
                        config=_config)
except Exception as ex:
    print(traceback.format_exc())
    print(ex)
    exit(-1)
```

使用统计分析服务则使用如下代码创建统计分析的服务对象：

```
try:
    config = Config()
    config.signature_version = SIGNATURE_VERSION
    credentials = Credentials(ACCESS_KEY, SECRET_KEY)
    client = ManagementClient(credentials, config, MANAGEMENT_ENDPOINT, False)
except Exception as ex:
    print(traceback.format_exc())
    print(ex)
    exit(-1)
```

使用操作跟踪服务则使用如下代码创建操作跟踪的服务对象：

```
try:
    config = Config()
    config.signature_version = SIGNATURE_VERSION
    credentials = Credentials(ACCESS_KEY, SECRET_KEY)
    trail_client = TrailClient(credentials, config, TRAIL_ENDPOINT, False)
```

```
except Exception as ex:
    print(traceback.format_exc())
    print(ex)
    exit(-1)
```

使用 IAM 服务则使用如下代码创建 IAM 的服务对象：

```
try:
    config = Config(endpoint_url=IAM_ENDPOINT,
                    region_name=REGION_NAME,
                    signature_version=SIGNATURE_VERSION, api_version=API_VERSION,
                    s3={'payload_signing_enabled': True})
    iam_client = oos.client(SERVICE_NAME,
                            api_version=API_VERSION,
                            use_ssl=False, endpoint_url=IAM_ENDPOINT,
                            access_key_id=ACCESS_KEY, secret_access_key=SECRET_KEY,
                            config=config,
                            verify=False)
except Exception as ex:
    print(traceback.format_exc())
    print(ex)
    exit(-1)
```

4 OOS 服务代码示例

OOS 的服务代码示例是 **python-sdk-6.5.3\examples** 目录的 **oos_example.py** 文件。

4.1 关于 Service 的操作

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。

该 API 只对验证用户有效，匿名用户不能执行该操作。

- 示例代码

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

4.1.2 GET Regions

获取资源池中的索引位置和数据位置列表。本接口只对使用对象存储网络的 endpoint 开放。

- 示例代码

```
def get_regions_example():
    response = client.get_regions()
    pretty_print(response)
```

4.2 关于 Bucket 的操作

4.2.1 PUT Bucket

此操作用来创建一个新的 bucket。Bucket 的命名方式中并不是支持所有的字符，具体请参见《OOS 开发者文档》Bucket 命名规范。

- 示例代码

当使用对象存储网络但不是仅有一个可用 location 的时候，使用下面的示例代码创建 Bucket，设置 location 值前，需要先调用 getRegions 接口，获得可用的 location：

```
def put_bucket_example():
    response = client.create_bucket(
        ACL='private',
        Bucket=BUCKET,
        CreateBucketConfiguration={
            'MetadataLocationConstraint':
               {'Location': 'QingDao'},
            'DataLocationConstraint': {
                'Type': 'Specified',
                'LocationList': ['QingDao'],
                'ScheduleStrategy': 'Allowed'}
        }
    )
    print(response['Location'])
    pretty_print(response)
```

4.2.2 GET Bucket Location

此操作用来获取 bucket 的索引位置和数据位置，只有根用户和拥有 GET Bucket Location 权限的子用户才能执行此操作。

- 示例代码

```
def get_bucket_location_example():
    response = client.get_bucket_location(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.3 GET Bucket ACL

此操作用来获取 Bucket ACL 信息，只有拥有 GET Bucket ACL 权限的用户才可以执行此操作。

示例代码

```
def get_bucket_acl_example():
    response = client.get_bucket_acl(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.4 GET Bucket (List Objects)

此操作用来返回 Bucket 中部分或者全部（每次最多 1000）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。

要执行该操作，需要对操作的 Bucket 拥有读权限。

- **示例代码**

```
def get_bucket_example():
    response = client.list_objects(
        Bucket=BUCKET,
        MaxKeys=10,
    )
    pretty_print(response)
```

4.2.5 DELETE Bucket

该操作用来删除 bucket，但要求被删除 bucket 中无 Object，即该 Bucket 中的所有 Object 都已被删除。

- **示例代码**

```
def delete_bucket_example():
    response = client.delete_bucket(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.6 PUT Bucket Policy

在 PUT 操作的 url 中加上 Policy，可以进行添加或修改 Policy 的操作。如果 Bucket 已经存在了 Policy，此操作会替换原有 Policy。只有根用户和拥有 PUT Bucket Policy 权限的用户才能执行此操作，否则会返回 403 AccessDenied 错误。

注意：如果 Bucket 的属性为私有或者公共读，使用该接口配置允许任何用户可以向该 Bucket 写对象的策略时，请联系天翼云客服评估审核后开通。

- **示例代码**

```
def put_bucket_policy_example():
    policy = {
        "Version": "2012-10-17",
        "Id": "http referer policy example",
        "Statement": [
            {
                "Effect": "Allow",
                "Sid": "1",
                "Principal": {
                    "AWS": "*"
                },
                "Action": ["s3:*"],
                "Resource": "arn:aws:s3:::" + BUCKET + "/*"
            }
        ]
    }
    response = client.put_bucket_policy(
        Bucket=BUCKET,
        Policy=json.dumps(policy)
    )
    pretty_print(response)
```

4.2.7 GET Bucket Policy

在 GET 操作的 url 中加上 policy，可以获得指定 Bucket 的 Policy。只有根用户和拥有 GET Bucket Policy 权限的用户才能执行此操作。

- **示例代码**

```
def get_bucket_policy_example():
    response = client.get_bucket_policy(
        Bucket=BUCKET
    )
    pretty_print(response)
    pretty_print(json.loads(response['Policy']))
```

4.2.8 DELETE Bucket Policy

在 DELETE 操作的 url 中加上 Policy，可以删除指定 Bucket 的 Policy。只有根用户和拥有 DELETE Bucket Policy 权限的子用户才能执行此操作。

- **示例代码**

```
def delete_bucket_policy_example():
    response = client.delete_bucket_policy(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.9 PUT Bucket WebSite

此操作用来配置网站托管属性。如果 Bucket 已经存在了 website，此操作会替换原有 website。只有根用户和拥有 PUT Bucket WebSite 权限的子用户才能执行此操作。

注意：

- OOS 自有网站托管域名不支持 HTTPS 访问，用户自定义域名支持 HTTPS 访问。
- 如果配置静态网站托管后，当匿名用户直接访问 Bucket 的域名，会将静态网站文件下载到本地。如果要实现访问静态网站时是预览网站内容，而非下载静态网站文件，静态网站域名须是 Bucket 绑定的已备案自定义域名，为 Bucket 绑定自定义域名请联系天翼云客服申请。
- 设置 Bucket 的网络配置请求消息体的上限是 10KiB。
- 尽量避免目标 Bucket 名中带有“.”，否则通过 HTTPS 访问时可能出现客户端校验证书出错。

网站托管配置步骤如下：

1）创建一个公共读属性的对象容器（Bucket）。

2）向天翼云客服提交工单，申请客户自定义域名添加白名单。

3）在域名管理中添加别名。

- 如果不使用 CDN 加速，将 Bucket 的 CNAME Record Value（*bucketname*.oos-website-cn.oos-*xx*.ctyunapi.cn）作为别名添加到域名管理系统中。
- 如果使用 CDN 加速，将 CDN 厂商提供的别名添加到域名管理系统中，然后在 CDN 回源地址中配置 OOS 侧的 CNAME Record Value，并将回源 host 配置为您的自定义域名（如 yourdomain.com）。

说明：创建 Bucket 时显示的 Endpoint 为 oos-cn.ctyunapi.cn，该 Endpoint 是针对整个对象存储网络的域名，该域名在解析时，会根据用户地理位置的不同解析到不同的资源池地址。如果创建 Bucket 时有多个数据域，系统默认选取创建时第一个有效数据位置作为 CNAME Record Value（*bucketname*.oos-website-cn.oos-*xx*.ctyunapi.cn）。CNAME Record Value 可以通过控制台 Bucket 属性中的**网站**查看。如果创建 Bucket 时，只有一个数据域可用，则在 Bucket 区域中展示的 CNAME Record Value 为 *bucketname*.oos-website-cn.oos-cn.ctyunapi.cn。所以如果使用静态网站托管，建议您根据 Bucket 区域属性中的数据位置，选择您想使用的数据位置的 CNAME Record Value 作为域名管理系统中的别名。例如您创建 Bucket 时有效数据位置为沈阳、兰州、成都、贵阳，则 Bucket 中展示的 CNAME Record Value 为 *bucketname*.oos-website-cn.oos-lnsy.ctyunapi.cn，您可以将 *bucketname*.oos-website-cn.oos-lnsy.ctyunapi.cn 作为别名，也可以将兰州、成都或者贵阳为域名的 CNAME Record Value 作为您的别名。

4) 上传文件

将网站的所有文件（html、CSS、js、图片等）上传到之前创建的 Bucket 中， 注意要保持文件之间的相对路径。

5) 配置 Bucket 网站属性：可以通过控制台或者调用本接口配置。

- **示例代码**

```
def put_bucket_website_example():
    response = client.put_bucket_website(
        Bucket=BUCKET,
        WebsiteConfiguration={
            'ErrorDocument': {
                'Key': 'error1.html'
```

```
            },
            'IndexDocument': {
                'Suffix': 'index1.html'
            }
        }
    )
    pretty_print(response)
```

4.2.10 GET Bucket WebSite

在 GET 操作的 url 中加上 website，可以获得指定 Bucket 的 website。

- 示例代码

```
def get_bucket_website_example():
    response = client.get_bucket_website(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.11 DELETE Bucket WebSite

在 DELETE 操作的 url 中加上 website，可以删除指定 bucket 的 website。只有根用户和拥有 DELETE Bucket WebSite 权限的子用户才能执行此操作。

- 示例代码

```
def delete_bucket_website_example():
    response = client.delete_bucket_website(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.12 List Multipart Uploads

该接口用于列出所有已经通过 Initiate Multipart Upload 请求初始化，但未完成或未终止的分片上传过程。

- 示例代码

```
def list_multipart_uploads_example():
    response = client.list_multipart_uploads(
        Bucket=BUCKET,
        MaxUploads=100
    )
    pretty_print(response)
```

4.2.13 PUT Bucket Logging

此操作用来添加/修改/删除 logging 的操作。如果 Bucket 已经存在了 logging，此操作会替换原有 logging。只有根用户和拥有 PUT Bucket Logging 权限的子用户才能执行此操作。

- 示例代码

```
def put_bucket_logging_example():
    response = client.put_bucket_logging(
        Bucket=BUCKET,
        BucketLoggingStatus={
            'LoggingEnabled': {
                'TargetBucket': '{0}-logging'.format(BUCKET),
                'TargetPrefix': '{0}_log'.format(BUCKET)
            }
        }
    )
    pretty_print(response)
```

4.2.14 GET Bucket Logging

此操作用来获得指定 Bucket 的 logging。只有根用户和拥有 GET Bucket Logging 权限的子用户才能执行此操作。

- 示例代码

```
def get_bucket_logging_example():
    response = client.get_bucket_logging(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.15 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。

- 示例代码

```
def head_bucket_example():
    response = client.head_bucket(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.16 PUT Bucket Lifecycle

此操作用来设置 Bucket 生命周期规则。只有根用户和具有 PUT Bucket Lifecycle 权限的子用户才能执行此操作。

生命周期是指对象从更新开始到被删除/转换存储类型之前的天数。

示例代码

```
def put_bucket_lifecycle_example():
    # 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',
                    'Prefix': 'test',
                    'Status': 'Enabled'
                }
            ]
        }
    )
    pretty_print(response)
```

4.2.17 GET Bucket Lifecycle

此接口用于返回配置的 Bucket 生命周期。

- **示例代码**

```
def get_bucket_lifecycle_example():
    response = client.get_bucket_lifecycle(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.18 DELETE Bucket Lifecycle

此操作用于删除配置的 Bucket 生命周期，OOS 将会删除指定 Bucket 的所有生命周期配置规则。用户的对象将永远不会到期，OOS 也不会再自动删除对象。只有根用户和拥有 DELETE Bucket Lifecycle 权限的子用户才能执行此操作。

- 示例代码

```
def delete_bucket_lifecycle_example():
    response = client.delete_bucket_lifecycle(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.19 PUT Bucket CORS

此操作用来设置 Bucket 的跨域资源共享(Cross-Origin Resource Sharing，CORS)。浏览器限制脚本内发起跨源 HTTP 请求，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。通过配置 CORS，可以解决不同域相互访问的问题，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。

- 示例代码

```
def put_bucket_cors_example():
    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)
```

4.2.20 GET Bucket CORS

返回 Bucket 的跨域配置信息。只有根用户和拥有 GET Bucket CORS 权限的子用户才能执行此操作。

- 示例代码

```
def get_bucket_cors_example():
    response = client.get_bucket_cors(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.21 DELETE Bucket CORS

删除 Bucket 的跨域配置信息。只有根用户和拥有 DELETE Bucket CORS 权限的子用户才能执行此操作。

- 示例代码

```
def delete_bucket_cors_example():
    response = client.delete_bucket_cors(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.2.22 Put Bucket Object Lock

使用此操作可以开启合规保留功能，开启后将对 Bucket 中所有对象生效。只有根用户和有权限的子用户才可以进行此操作，匿名用户不能进行此操作。

开启 Bucket 合规保留功能后，任何用户（包括根用户）都不能对此 Bucket 内处于合规保留期的对象进行修改和删除。

可以重复调用此接口：

- 如果已经开启合规保留策略：设置合规保留时长大于或等于上次设置的时长，才能生效。如果使用 Years 和 Days 两种方式设置合规保留时长，年与天的换算关系为：1 年等于 365 天。
- 如果未开启合规保留策略：设置合规保留时长可以大于、等于或小于上次设置的时长。

注意：

- 合规保留一旦开启，不能关闭，不能缩短合规保留时长，但可以延长合规保留时长。
- 合规保留的时间精确到秒，例如对 Bucket A 设置合规保留时长为 10 天，对象 A 属于 Bucket A，A1 的最后更新时间为 2019-3-1 12:00:00，该文件会在 2019-3-11 12:00:01 过合规保留期。
- 任何用户（包括根用户）都不能修改、覆盖、删除处于合规保留期的对象。
- 处于合规保留期的对象，无法通过调用 API、控制台修改对象的存储类型，只能通过生命周期修改存储类型。
- 处于合规保留期的对象，如果设置了生命周期规则，则修改存储类型的生命周期规则可以生效，设置删除操作的生命周期规则待对象过了合规保留期后才能生效。

- **示例代码**

```
def put_bucket_object_lock_example():
    response = client.put_bucket_object_lock(
        Bucket=BUCKET,
        ObjectLockConfiguration={
            'ObjectLockEnabled': 'Disabled',
            'Rule': {
                'DefaultRetention': {
                    'Mode': 'COMPLIANCE',
                    'Days': 1
                }
            }
        }
    )
    pretty_print(response)
```

4.2.23 GET Bucket Object Lock

使用此操作可以获取 Bucket 合规保留的配置信息。只有根用户和有权限的子用户才可以进行此操作。

- **示例代码**

```
def get_bucket_object_lock_example():
    response = client.get_bucket_object_lock(
        Bucket=BUCKET
    )
```

```
    pretty_print(response)
```

4.2.24 DELETE Bucket Object Lock

使用此操作可以删除未启用的合规保留配置信息。只有根用户和有权限的子用户才可以进行此操作。

- 示例代码

```
def delete_bucket_object_lock_example():
    response = client.delete_bucket_object_lock(
        Bucket=BUCKET
    )
    pretty_print(response)
```

4.3 关于 Object 的操作

4.3.1 PUT Object

此操作用来向指定 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。

说明：对象名称不能包含 ASCII 码为 0 的字符（NUL）。

- 示例代码

```
def put_object_example():
    with open(UPLOAD_FILE, 'rb') as data:
        response = client.put_object(
            DataLocation='type=Specified,location=QingDao,scheduleStrategy=Allowed',
            Bucket=BUCKET,
            Key=KEY,
            Body=data,
            StorageClass='STANDARD',
            ContentType='application/octet-stream'
        )
    pretty_print(response)
```

4.3.2 GET Object

此操作用来检索在 OOS 中的对象信息，执行此操作，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。

- 示例代码

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

4.3.3 DELETE Object

此操作用来删除指定的对象，要求用户要对对象所在的 bucket 拥有写权限。

- 示例代码

```
def delete_object_example():
    response = client.delete_object(
        Bucket=BUCKET,
        Key=KEY
    )
    pretty_print(response)
```

4.3.4 PUT Object - Copy

此操作用来创建一个存储在 OOS 里的对象拷贝。类似于执行一个 GET，然后再执行一次 PUT。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。

注意：当 OOS 接收到请求或者正在执行拷贝操作时，拷贝操作可能会返回失败。如果在拷贝操作开始之前出现异常，OOS 返回标准的错误信息。如果在拷贝操作过程中出现异常，由于 200 OK 状态码是先返回的，这意味着 200 OK 响应体可能包含成功或错误。请在客户端应用程序中解析响应体的内容并进行适当处理。

- 示例代码

```
def put_object_copy_example():
  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 Upload

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。

- 示例代码

```
def multipart_upload_example():
    upload = client.create_multipart_upload(
    Bucket=BUCKET,
    Key=MULTIPART_UPLOAD_KEY,
    DataLocation='type=Specified,location=ChengDu,scheduleStrategy=Allowed'
    )
    pretty_print(upload)
```

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。

- **示例代码**

响应中包含 Etag 头，用户需要在最后发送完成分片上传过程请求的时候包含该 Etag 值。

```
    MultipartUpload = {
        'Part': []
    }
    with open(UPLOAD_FILE, 'rb') as fd:
        num = 0
        for chunk in iter(lambda: fd.read(5 * 1024 * 1024), b''):
            num += 1
            part = client.upload_part(
                Body=chunk,
                Bucket=BUCKET,
                Key=MULTIPART_UPLOAD_KEY,
                PartNumber=num,
                UploadId=upload['UploadId']
            )
            MultipartUpload['Part'].append({
```

```
            'PartNumber': num,
            'ETag': part['ETag']
        })
        pretty_print(part)
```

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。

OOS 提供了不合并片段也可以读取 Object 内容的功能。在没有调用 Complete Multipart Upload 接口合并片段时，也可以通过调用 Get Object 接口来获取文件内容，OOS 会根据最近一次创建的 uploadId，以分片号升序的方式顺序读取片段内容，返回给客户端。

- **示例代码**

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

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。

- 示例代码

```
def abort_multipart_example():
    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。

- 示例代码

```
    list = client.list_parts(
        Bucket=BUCKET,
        Key=MULTIPART_UPLOAD_KEY,
        MaxParts=10,
        UploadId=upload['UploadId']
    )
    pretty_print(list)
```

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。

- 示例代码

```
def copy_part_example():
    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},
        Key=MULTIPART_UPLOAD_KEY,
        PartNumber=1,
        UploadId=upload['UploadId']
    )
    pretty_print(part)

    MultipartUpload = {
        'Part': [
            {'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)
```

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。

批量删除请求包含一个不超过 1000 个 Object 的 XML 列表。在这个 xml 中，需要指定要删除的 object 的名字。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。

- **示例代码**

```
def delete_multiple_example():
    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。

- **示例代码**

```
def generate_shared_link_example():
    shared_link = client.generate_presigned_url(
        ClientMethod='get_object',
        Params={
            'Bucket': BUCKET,
            'Key': KEY
        },
        ExpiresIn=3600  # 过期时间
    )
    print(shared_link)
```

4.3.13 HEAD Object

此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。

- 示例代码

```
def head_object_example():
    response = client.head_object(
        Bucket=BUCKET,
        Key=KEY,
    )
    pretty_print(response)
```

5 统计分析服务代码示例

统计分析（Statistics API）的服务代码示例见 **python-sdk-6.5.3\examples\management_example.py** 文件。

注意：要设置成统计分析（Statistics API）服务的 Endpoint。

设置 endpoint 的示例代码为：

```
MANAGEMENT_ENDPOINT = 'https://oos-cn-mg.ctyunapi.cn'
```

5.1 GetCapacity

此操作用来查询用户的容量。

- 示例代码

```
def get_capacity_example():
    request = GetCapacityRequest()
    request.BeginDate = "2023-07-20"
    request.EndDate = "2023-07-21"
    request.Bucket = BUCKET
    request.Freq = "byDay"
    result = client.get_capacity(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

5.2 GetDeleteCapacity

此操作用来查询用户删除的容量。

- 示例代码

```
def get_delete_capacity_example():
    request = GetDeleteCapacityRequest()
    request.BeginDate = "2023-07-20"
    request.EndDate = "2023-07-21"
    request.Bucket = BUCKET
    request.Freq = "byDay"
    result = client.get_delete_capacity(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

5.3 GetTraffics

此操作用来查询用户的流量。

- 示例代码

```
def get_traffics_example():
    request = GetTrafficsRequest()
    request.BeginDate = "2023-07-20"
    request.EndDate = "2023-07-21"
    request.Bucket = BUCKET
    request.Freq = "byDay"
    request.InOutType = "all"
    request.InternetType = "all"
    request.TrafficsType = "all"
    result = client.get_traffics(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

5.4 GetRequests

此操作用来查询用户的请求次数。

- 示例代码

```
def get_requests_example():
request = GetRequestsRequest()
    request.BeginDate = "2023-07-20"
    request.EndDate = "2023-07-21"
    request.Bucket = BUCKET
    request.Freq = "byHour"
    request.InternetType = "all"
    request.RequestsType = "all"
    result = client.get_requests(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

5.5 GetReturnCode

此操作用来查询用户请求返回码次数。

- 示例代码

```
def get_return_code_example():
    request = GetReturnCodeRequest()
    request.BeginDate = "2023-07-20"
```

```
    request.EndDate = "2023-07-21"
    request.Bucket = BUCKET
    request.Freq = "byHour"
    request.InternetType = "all"
    request.RequestsType = "all"
    request.ResponseType = "all"
    result = client.get_return_code(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

5.6 GetConcurrentConnection

此操作用来查询用户的并发连接数。

- 示例代码

```
def get_concurrent_connection_example():
    request = GetConcurrentConnectionRequest()
    request.BeginDate = "2023-07-20"
    request.EndDate = "2023-07-20"
    request.Bucket = BUCKET
    request.Freq = "by5min"
    request.InternetType = "all"
    result = client.get_concurrent_connection(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

5.7 GetUsage

此操作用来查询用户 Bucket 的使用情况。本接口已不推荐使用，请使用相对应的其他的接口。

- 示例代码

```
def get_usage_example():
    request = GetUsageRequest()
    request.BeginDate = "2023-07-20"
    request.EndDate = "2023-07-21"
    request.Bucket = BUCKET
    request.Freq = "byDay"
    result = client.get_usage(request)
    print("region count", result.RegionCount)
    i = 0
    while i < result.RegionCount:
```

```
        data_cnt = result.get_region(i).get_data_cnt()
        j = 0
        while j < data_cnt:
            print("upload region,data,", i, j,
                  result.get_region(i).get_data(j).Upload)
            j = j + 1
        i = i + 1
    print(result.get_header().to_string())
    print(result.to_string())
```

5.8 GetBandwidth

此操作用来查询用户的已用带宽。本接口已不推荐使用，请使用相对应的其他的接口。

- 示例代码

```
def get_bandwidth_example():
    request = GetBandwidthRequest()
    request.BeginDate = "2023-07-20-00-00"
    request.EndDate = "2023-07-21-10-10"
    request.Bucket = BUCKET
    result = client.get_bandwidth(request)
    print("region count", result.RegionCount)
    i = 0
    while i < result.RegionCount:
        data_cnt = result.get_region(i).get_data_cnt()
        j = 0
        while j < data_cnt:
            print("i,j,UploadBW,", i, j,
                  result.get_region(i).get_data(j).UploadBW)
            j = j + 1
        i = i + 1
    print(result.get_header().to_string())
    print(result.to_string())
```

6 操作跟踪服务代码示例

操作跟踪（CloudTrail API）的服务代码示例见 **python-sdk-6.5.3\examples\cloudtrail_example.py** 文件。

注意：要设置操作跟踪（CloudTrail API）服务的 Endpoint。

设置 endpoint 的示例代码为：

```
OOS_ENDPOINT_TRAIL = 'https://oos-cn-cloudtrail.ctyunapi.cn'
```

6.1 CreateTrail

此操作用来创建一个跟踪，并将跟踪日志保存到指定的 OOS Bucket。

注意：

- 一个账户最多创建 10 个跟踪。
- 使用 CreateTrail 创建跟踪后，跟踪默认是关闭状态，需要调用 StartLogging 开启跟踪。

- **示例代码**

```
def create_trail_example():
    request = CreateTrailRequest()
    request.name = REQ_NAME
    request.s3BucketName = BUCKET
    request.s3KeyPrefix = KRY_PREFIX
    result = trail_client.create_trail(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.2 DeleteTrail

此操作用来删除指定的跟踪。

- **示例代码**

```
def delete_trail_example():
    request = DeleteTrailRequest()
    request.name = REQ_NAME
    result = trail_client.delete_trail(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.3 DescribeTrails

此操作用来获取操作跟踪的设置信息。

- 示例代码

```
def describe_trail_example():
    request = DescribeTrailsRequest()
    request.name_list = [REQ_NAME]
    result = trail_client.describe_trail(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.4 GetTrailStatus

此操作用来获取跟踪状态信息。

- 示例代码

```
def get_trail_status_example():
    request = GetTrailStatusRequest()
    request.name = REQ_NAME
    result = trail_client.get_trail_status(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.5 PutEventSelectors

此操作用来配置跟踪的管理事件筛选器。

默认情况下，未设置管理事件筛选器的跟踪，会记录所有的管理事件，每个跟踪最多创建 1 个管理事件筛选器。

- 示例代码

```
def put_event_selectors_example():
    request = PutEventSelectorsRequest()
    request.trail_name = REQ_NAME
    selectors_list = []
    selector = EventSelect()
    selector.read_write_type = "All"
    selectors_list.append(selector)
    request.selector_list = selectors_list
    result = trail_client.put_event_selectors(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.6 GetEventSelectors

此操作用来查看管理事件筛选器的设置信息。

- 示例代码

```
def get_event_selectors_example():
    request = GetEventSelectorsRequest()
    request.trail_name = REQ_NAME
    result = trail_client.get_event_selectors(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.7 UpdateTrail

此操作用来更新跟踪设置参数，包括跟踪日志数据保存的 OOS Bucket、跟踪日志前缀等。

说明：

- 更新跟踪时，无需暂停跟踪。
- 如果 S3BucketName 和 S3KeyPrefix 都未指定，该操作不起作用，但是不报错，返回 200 OK。

- 示例代码

```
def update_trail_example():
    request = UpdateTrailRequest()
    request.name = REQ_NAME
    request.s3BucketName = BUCKET
    request.s3KeyPrefix = KRY_PREFIX
    result = trail_client.update_trail(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.8 StartLogging

此操作用来开启跟踪。

说明：使用 CreateTrail 创建跟踪后，跟踪默认是关闭状态，需要调用 StartLogging 开启跟踪。

- 示例代码

```
def start_logging_example():
    request = StartLoggingRequest()
    request.name = REQ_NAME
```

```
    result = trail_client.start_logging(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.9 StopLogging

此操作用来停止跟踪功能。

- 示例代码

```
def stop_logging_example():
    request = StopLoggingRequest()
    request.name = REQ_NAME
    result = trail_client.stop_logging(request)
    print(result.get_header().to_string())
    print(result.to_string())
```

6.10 LookupEvents

此操作用来查看账户中的管理事件。用户可以查看近 6 个月内发生的管理事件。

- 示例代码

```
def lookup_events_example():
    request = LookupEventsRequest()
    request.start_time = str(int((round(time.time() - 5 * 24 * 60 * 60) * 1000)))
    request.end_time = str(int(round(time.time() * 1000)))
    lookup_attribute_list = []
    lookup_attribute = LookupAttribute()
    lookup_attribute.attribute_key = ""  # 指定查询管理事件的属性 Key。
    lookup_attribute.attribute_value = ""  # 指定 AttributeKey 对应的值。
    lookup_attribute_list.append(lookup_attribute)
    request.lookup_attribute_list = lookup_attribute_list
    request.max_results = str(20)
    result = trail_client.lookup_events(request)
    print(result.get_header().to_string())
    print(result.get_event(0).to_string())
```

7 IAM 服务代码示例

IAM 的服务代码示例见 **python-sdk-6.5.3\examples\iam_example.py** 文件。

注意：要设置成访问控制（IAM）服务的 Endpoint。

设置 endpoint 的示例代码为：

```
IAM_ENDPOINT = 'https://oos-cn-iam.ctyunapi.cn/'
```

7.1 用户管理接口

7.1.1 CreateUser

此操作用来创建新的 IAM 用户。

- 示例代码

```
def create_user_example():
    data = {
        "Action": "CreateUser",
        "UserName": IAM_USER_NAME,
        "Tags.member.1.Key": "Key1",
        "Tags.member.1.Value": "Value1"
    }
    body = encode_params(data)
    response = iam_client.create_user(
        Body=body
    )
    pretty_print(response)
```

7.1.2 GetUser

此操作用来获取 IAM 用户信息。

- 示例代码

```
def get_user_example():
    data = {
        "Action": "GetUser",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.get_user(
        Body=body
    )
```

```
    pretty_print(response)
```

7.1.3 ListUsers

此操作用来列出 IAM 用户。如果账户中没有创建 IAM 用户，则返回空列表。

- 示例代码

```
def list_users_example():
    data = {
        "Action": "ListUsers",
        "MaxItems": 1000,
    }
    body = encode_params(data)
    response = iam_client.list_users(
        Body=body
    )
    pretty_print(response)
```

7.1.4 DeleteUser

此操作用来删除指定的 IAM 用户。

- 示例代码

```
def delete_user_example():
    data = {
        "Action": "DeleteUser",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.delete_user(
        Body=body
    )
    pretty_print(response)
```

7.1.5 TagUser

此操作用来为 IAM 用户添加标签。

说明：

- 可以同时添加一个或多个标签，一个 IAM 用户最多能添加 10 个标签。
- 如果 Tags.member.Key 已经存在，其值则会被新添加的 value 覆盖。

- 示例代码

```
def tag_user_example():
```

```
    data = {
        "Action": "TagUser",
        "Tags.member.1.Key": "Key1",
        "Tags.member.1.Value": "Value1",
        "Tags.member.2.Key": "Key2",
        "Tags.member.2.Value": "Value2",
        "Tags.member.3.Key": "Key3",
        "Tags.member.3.Value": "Value3",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.tag_user(
        Body=body
    )
    pretty_print(response)
```

7.1.6 UntagUser

此操作用来删除用户的指定标签。

说明：如果删除的标签不存在时，返回状态 200（成功）。

- **示例代码**

```
def untag_user_example():
    data = {
        "Action": "UntagUser",
        "TagKeys.member.1": "Key1",
        "TagKeys.member.2": "Key2",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.untag_user(
        Body=body
    )
    pretty_print(response)
```

7.1.7 ListUserTags

此操作用来列出指定 IAM 用户的标签。

- **示例代码**

```
def list_user_tags_example():
    data = {
        "Action": "ListUserTags",
```

```
        "MaxItems": 1000,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.list_user_tags(
        Body=body
    )
    pretty_print(response)
```

7.1.8 ListGroupsForUser

此操作用来列出指定 IAM 用户所属的 IAM 用户组。

- 示例代码

```
def list_groups_for_user_example():
    data = {
        "Action": "ListGroupsForUser",
        "MaxItems": 1000,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.list_groups_for_user(
        Body=body
    )
    pretty_print(response)
```

7.1.9 CreateAccessKey

此操作用来为指定的 IAM 用户创建新的 AccessKey。

说明：

- 新密钥的默认状态是 Active。
- 如果未指明 IAM 用户名，则为请求者创建新的 AccessKey。

- 示例代码

```
def create_access_key_example():
    data = {
        "Action": "CreateAccessKey",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.create_access_key(
        Body=body
    )
```

```
    pretty_print(response)
```

7.1.10 ListAccessKeys

此操作用来返回指定 IAM 用户的 AK 的详细信息。

说明：

- 如果指定用户没有 AK，则操作返回空列表。
- 如果未指定 IAM 用户，则返回请求者的 AK。

- 示例代码

```
def list_access_key_example():
    data = {
        'Action': 'ListAccessKeys',
        "MaxItems": 100,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.list_access_keys(
        Body=body
    )
    pretty_print(response)
```

7.1.11 GetAccessKeyLastUsed

此操作用来查询指定密钥最后一次使用的时间及服务名称。

- 示例代码

```
def get_access_key_last_used_example():
    data = {
        "Action": "GetAccessKeyLastUsed",
        "Version": API_VERSION,
        "AccessKeyId": ACCESS_KEY
    }
    response = iam_client.get_access_key_last_used(
        Body=encode_params(data)
    )
    pretty_print(response)
```

7.1.12 UpdateAccessKey

此操作用来更新指定访问密钥（AK）的状态，从 Active 到 Inactive，或者从 Inactive 到 Active。

说明：

- 如果请求中未携带 IAM 用户名，则更新请求者的密钥状态。
- 此操作可以管理根用户的密钥。

- **示例代码**

```
def update_access_key_example():
    data = {
        "Action": "UpdateAccessKey",
        "AccessKeyId": ACCESS_KEY,
        "Status": "Active",  # Active or Inactive
    }
    body = encode_params(data)
    response = iam_client.update_access_key(
        Body=body
    )
    pretty_print(response)
```

7.1.13 DeleteAccessKey

此操作用来删除指定 IAM 用户关联的 AccessKey。

说明：

- 如果未指定用户名，IAM 将根据签名请求的 OOS AccessKeyID 确定用户名。
- 此操作也可以用来删除根用户的 AccessKey。

- **示例代码**

```
def delete_access_key_example():
    data = {
        "Action": "DeleteAccessKey",
        "UserName": IAM_USER_NAME,
        "AccessKeyId": "your_accessKeyId"
# 建议此处不要写根用户 ak，根用户 ak 不能被随意删除；子用户未分配权限，无权限删除根用户 ak
    }
    body = encode_params(data)
    response = iam_client.delete_access_key(
        Body=body
    )
    pretty_print(response)
```

7.1.14 GetSessionToken

OOS 为用户提供临时授权访问。此操作用来获取临时访问密钥。根用户和子用户默认拥有调用此接口的权限。如果配置了禁止子用户调用该接口的 IAM 策略，该策略不会生效。

- 示例代码

```
def get_session_token_example():
    data = {
        "Action": "GetSessionToken",
        "Version": API_VERSION,
        "DurationSeconds": "1800"
    }
    body = encode_params(data)
    response = iam_client.get_session_token(
        Body=body
    )
    pretty_print(response)
    print(response['GetSessionTokenResult']['Credentials']['SessionToken'       ])
    print(response['GetSessionTokenResult']['Credentials']['AccessKeyId'])
    print(response['GetSessionTokenResult']['Credentials']['SecretAccessKey'])
```

7.1.15 CreateLoginProfile

此操作用来为指定 IAM 用户创建控制台的登录密码。

- 示例代码

```
def create_login_profile_example():
    data = {
        "Action": "CreateLoginProfile",
        "Password": "your_password",  # 指定用户登录控制台的密码
        "PasswordResetRequired": "false",  # IAM 用户使用初始密码登录控制台后是否需要重置密码
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.create_login_profile(
        Body=body
    )
    pretty_print(response)
```

7.1.16 GetLoginProfile

此操作用来获取 IAM 用户控制台登录密码创建的时间、用户首次登录后是否需要修改密码。

- 示例代码

```
def get_login_profile_example():
    data = {
        "Action": "GetLoginProfile",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.get_login_profile(
        Body=body
    )
    pretty_print(response)
```

7.1.17 UpdateLoginProfile

此操作用来更改指定 IAM 用户控制台的登录密码。

- 示例代码

```
def update_login_profile_example():
    data = {
        "Action": "UpdateLoginProfile",
        "Password": "your_password",  # 指定用户登录控制台的密码
        "PasswordResetRequired": "false",  # IAM 用户使用初始密码登录控制台后是否需要重置密码
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.update_login_profile(
        Body=body
    )
    pretty_print(response)
```

7.1.18 DeleteLoginProfile

此操作用来删除指定 IAM 用户控制台的登录密码。执行此操作后，指定用户将不能通过控制台进行 OOS 服务。

- 示例代码

```
def delete_login_profile_example():
    data = {
        "Action": "DeleteLoginProfile",
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.delete_login_profile(
```

```
        Body=body
    )
    pretty_print(response)
```

7.1.19 ChangePassword

此操作用来修改 IAM 用户自身的控制台登录密码。

注意：此操作仅能修改用户自身的控制台登录密码。

- **示例代码**

```
def change_password_example():
    data = {
        "Action": "ChangePassword",
        "NewPassword": "your_newPassword",
        "OldPassword": "your_oldPassword",
    }
    body = encode_params(data)
    response = iam_client.change_password(
        Body=body
    )
    pretty_print(response)
```

7.1.20 CreateVirtualMFADevice

此操作用来创建虚拟 MFA 设备。创建虚拟 MFA 后，可以使用 EnableMFADevice 启用虚拟 MFA 设备，并将该虚拟 MFA 设备与指定的 IAM 用户关联。

注意：QR 代码和 Base32 字符串中包含的 MFA 设备密钥，就像您的密码一样，应被妥善保管和处理。

- **示例代码**

```
def create_virtual_mfa_device_example():
    data = {
        "Action": "CreateVirtualMFADevice",
        "VirtualMFADeviceName": "your_virtualMFADeviceName"  # 虚拟 MFA 设备的名称。
    }
    body = encode_params(data)
    response = iam_client.create_virtual_mfa_device(
        Body=body
    )
    pretty_print(response)
```

7.1.21 EnableMFADevice

此操作用来启用指定的虚拟 MFA 设备，并将该虚拟 MFA 设备与指定的 IAM 用户关联。

- 示例代码

```
def enable_mfa_device_example():
    data = {
        "Action": "EnableMFADevice",
        "AuthenticationCode1": "your_code1",  # 虚拟 MFA 设备发出的验证码，为六位数字验证码。
        "AuthenticationCode2": "your_code2",
# 虚拟 MFA 设备发出的下一个紧邻 AuthenticationCode1 的六位数字验证码。
        "SerialNumber": SERIAL_NUMBER,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.enable_mfa_device(
        Body=body
    )
    pretty_print(response)
```

7.1.22 ListVirtualMFADevices

此操作用来按分配状态列出 OOS 账户中定义的虚拟 MFA 设备。如果未指定分配状态，则操作将返回所有虚拟 MFA 设备的列表。

- 示例代码

```
def list_virtual_mfa_devices_example():
    data = {
        "Action": "ListVirtualMFADevices",
        "AssignmentStatus": "Any",  # 指定需要列出的虚拟 MFA 设备状态。
        "MaxItems": "100"
    }
    body = encode_params(data)
    response = iam_client.list_virtual_mfa_devices(
        Body=body
    )
    pretty_print(response)
```

7.1.23 ListMFADevices

此操作用来列出 IAM 用户的虚拟 MFA 设备。

说明：如果请求中没有指定用户，则会列出请求用户自己名下的虚拟 MFA 设备。

- 示例代码

```
def list_mfa_devices_example():
    data = {
        "Action": "ListMFADevices",
        "UserName": IAM_USER_NAME,
        "MaxItems": "100"
    }
    body = encode_params(data)
    response = iam_client.list_mfa_devices(
        Body=body
    )
    pretty_print(response)
```

7.1.24 DeactivateMFADevice

此操作用来终止使用指定的 MFA 设备，并与用户解除关联。

- 示例代码

```
def deactivate_mfa_device_example():
    data = {
        "Action": "DeactivateMFADevice",
        "SerialNumber": SERIAL_NUMBER,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.deactivate_mfa_device(
        Body=body
    )
    pretty_print(response)
```

7.1.25 DeleteVirtualMFADevice

此操作用来删除指定的虚拟 MFA 设备。

- 示例代码

```
def delete_virtual_mfa_device_example():
    data = {
        "Action": "DeleteVirtualMFADevice",
        "SerialNumber": SERIAL_NUMBER
    }
    body = encode_params(data)
    response = iam_client.delete_virtual_mfa_device(
        Body=body
    )
    pretty_print(response)
```

7.2 用户组管理接口

7.2.1 CreateGroup

此操作用来创建新的 IAM 用户组。

- 示例代码

```
def create_group_example():
    data = {
        "Action": "CreateGroup",
        "GroupName": IAM_GROUP_NAME
    }
    body = encode_params(data)
    response = iam_client.create_group(
        Body=body
    )
    pretty_print(response)
    print("HTTPStatusCode" + str(response['ResponseMetadata']['HTTPStatusCode'])
    )
    print("GroupName" + response['CreateGroupResult']['Group']['GroupName']
    )
```

7.2.2 GetGroup

此操作用来获取指定 IAM 用户组及组内 IAM 用户列表。

- 示例代码

```
def get_group_example():
    data = {
        "Action": "GetGroup",
        "GroupName": IAM_GROUP_NAME,
        "MaxItems": "1000"
    }
    body = encode_params(data)
    response = iam_client.get_group(
        Body=body
    )
    pretty_print(response)
```

7.2.3 AddUserToGroup

此操作用来将指定的 IAM 用户加入到指定的 IAM 用户组，每次只能将一个用户加入到指定用户组。

说明：多次执行此操作将加同一个用户加入同一个组，不会报错。

- **示例代码**

```
def add_user_to_group_example():
    data = {
        "Action": "AddUserToGroup",
        "GroupName": IAM_GROUP_NAME,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.add_user_to_group(
        Body=body
    )
    pretty_print(response)
```

7.2.4 RemoveUserFromGroup

此操作用来将指定用户从指定用户组移除。

说明：如要移除的用户存在，但不在指定的用户组，不会报错，返回 200；如果要移除的用户不存在，会报 404 NoSuchEntity 错误。

- **示例代码**

```
def remove_user_from_group_example():
    data = {
        "Action": "RemoveUserFromGroup",
        "GroupName": IAM_GROUP_NAME,
        "UserName": IAM_USER_NAME
    }
    body = encode_params(data)
    response = iam_client.remove_user_from_group(
        Body=body
    )
    pretty_print(response)
```

7.2.5 ListGroups

此操作用来列出所有的 IAM 用户组。

- **示例代码**

```
def list_group_example():
    data = {
        "Action": "ListGroups",
        "MaxItems": 1000
```

```
    }
    body = encode_params(data)
    response = iam_client.list_groups(
        Body=body
    )
    pretty_print(response)
```

7.2.6 DeleteGroup

此操作用来删除指定的 IAM 用户组。

- 示例代码

```
def delete_group_example():
    data = {
        "Action": "DeleteGroup",
        "GroupName": IAM_GROUP_NAME
    }
    body = encode_params(data)
    response = iam_client.delete_group(
        Body=body
    )
    pretty_print(response)
```

7.3 策略管理接口

7.3.1 CreatePolicy

此操作用来为账户创建策略。如果策略名已存在，再创建同一名称的策略，后创建的策略会将已存在的同名策略覆盖。

- 示例代码

```
def create_policy_example():
    policy_document = '''{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "*",
        "Resource": "*"
        }]
    }'''
    data = {
        "Action": "CreatePolicy",
        "PolicyName": "your_policyName",  # 策略名称，策略名必须唯一。
        "Description": "The first policy",  # 策略描述。
        "PolicyDocument": policy_document
    }
    body = encode_params(data)
    response = iam_client.create_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.2 GetPolicy

此操作用来获取策略相关信息。

- 示例代码

```
def get_policy_example():
    data = {
        "Action": "GetPolicy",
        "PolicyArn": POLICY_ARN
    }
    body = encode_params(data)
    response = iam_client.get_policy(
        Body=body
```

```
    )
    pretty_print(response)
```

7.3.3 ListPolicies

此操作用来列出账户下所有的策略。

- 示例代码

```
def list_policies_example():
    data = {
        "Action": "ListPolicies",
        "OnlyAttached": "true",  # 用于标识是否只显示已关联 IAM 用户或 IAM 用户组的策略。
        "MaxItems": "10"
    }
    body = encode_params(data)
    response = iam_client.list_policies(
        Body=body
    )
    pretty_print(response)
```

7.3.4 ListEntitiesForPolicy

此操作用来列出指定策略所附加的所有 IAM 用户或 IAM 用户组。

您可以使用可选 EntityFilter 参数将结果限制为特定类型的实体（用户或组）。

- 示例代码

```
def list_entities_for_policy_example():
    data = {
        "Action": "ListEntitiesForPolicy",
        "PolicyArn": POLICY_ARN,
        "MaxItems": "10"
    }
    body = encode_params(data)
    response = iam_client.list_entities_for_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.5 DeletePolicy

此操作用来删除指定的策略。

注意：在删除策略前，要确保该策略没有附加到任何 IAM 用户或 IAM 用户组，可以按照下列步骤进行解除关联 IAM 用户和 IAM 用户组：

1. 使用 ListEntitiesForPolicy 操作查看关联的 IAM 用户和用户组。
2. 使用 DetachUserPolicy 解除策略关联的 IAM 用户，使用 DetachGroupPolicy 解除策略关联的 IAM 用户组。

- **示例代码**

```
def delete_policy_example():
    data = {
        "Action": "DeletePolicy",
        "PolicyArn": POLICY_ARN
    }
    body = encode_params(data)
    response = iam_client.delete_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.6 AttachUserPolicy

此操作用来将指定的策略与指定的 IAM 用户关联。

- **示例代码**

```
def attach_user_policy_example():
    data = {
        "Action": "AttachUserPolicy",
        "UserName": IAM_USER_NAME,
        "PolicyArn": POLICY_ARN
    }
    body = encode_params(data)
    response = iam_client.attach_user_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.7 ListAttachedUserPolicies

此操作用来列出与指定用户关联的策略。

- **示例代码**

```
def list_attached_user_policies_example():
    data = {
        "Action": "ListAttachedUserPolicies",
        "UserName": IAM_USER_NAME,
        "MaxItems": "10"
```

```
    }
    body = encode_params(data)
    response = iam_client.list_attached_user_policies(
        Body=body
    )
    pretty_print(response)
```

7.3.8 DetachUserPolicy

此操作用来解除指定用户关联的指定策略。

- 示例代码

```
def detach_user_policy_example():
    data = {
        "Action": "DetachUserPolicy",
        "UserName": IAM_USER_NAME,
        "PolicyArn": POLICY_ARN
    }
    body = encode_params(data)
    response = iam_client.detach_user_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.9 AttachGroupPolicy

此操作用来将指定的策略与指定的 IAM 用户组关联。

- 示例代码

```
def attach_group_policy_example():
    data = {
        "Action": "AttachGroupPolicy",
        "GroupName": IAM_GROUP_NAME,
        "PolicyArn": POLICY_ARN
    }
    body = encode_params(data)
    response = iam_client.attach_group_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.10 ListAttachedGroupPolicies

此操作用来列出与指定 IAM 用户组关联的策略。

- 示例代码

```
def list_attached_group_policies_example():
    data = {
        "Action": "ListAttachedGroupPolicies",
        "GroupName": IAM_GROUP_NAME,
        "MaxItems": "10"
    }
    body = encode_params(data)
    response = iam_client.list_attached_group_policies(
        Body=body
    )
    pretty_print(response)
```

7.3.11 DetachGroupPolicy

此操作用来解除指定 IAM 用户组关联的指定策略。

- 示例代码

```
def detach_group_policy_example():
    data = {
        "Action": "DetachGroupPolicy",
        "GroupName": IAM_GROUP_NAME,
        "PolicyArn": POLICY_ARN
    }
    body = encode_params(data)
    response = iam_client.detach_group_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.12 UpdateAccountPasswordPolicy

此操作用来更新账户的密码规则设置。

- 示例代码

```
def update_account_password_policy_example():
    data = {
        "Action": "UpdateAccountPasswordPolicy",
        "AllowUsersToChangePassword": "true",
        "HardExpiry": "true",
        "MaxPasswordAge": "0",
        "MinimumPasswordLength": "8",
        "PasswordReusePrevention": "3",
```

```
        "RequireLowercaseCharacters": "false",
        "RequireNumbers": "false",
        "RequireSymbols": "true",
        "RequireUppercaseCharacters": "false"
    }
    body = encode_params(data)
    response = iam_client.update_account_password_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.13 GetAccountPasswordPolicy

此操作用来获取账户的密码策略。

- 示例代码

```
def get_account_password_policy_example():
    data = {
        "Action": "GetAccountPasswordPolicy",
        "Version": API_VERSION
    }
    body = encode_params(data)
    response = iam_client.get_account_password_policy(
        Body=body
    )
    pretty_print(response)
```

7.3.14 DeleteAccountPasswordPolicy

此操作用来将账户的密码规则恢复到默认密码规则。

- 示例代码

```
def delete_account_password_policy_example():
    data = {
        "Action": "DeleteAccountPasswordPolicy",
        "Version": API_VERSION
    }
    body = encode_params(data)
    response = iam_client.delete_account_password_policy(
        Body=body
    )
    pretty_print(response)
```

7.4 服务数量查询

7.4.1 GetAccountSummary

此操作用来获取账户中的实体数量和服务限制信息。

- 示例代码

```
def get_account_summary_example():
    data = {
        "Action": "GetAccountSummary",
        "Version": API_VERSION
    }
    body = encode_params(data)
    response = iam_client.get_account_summary(
        Body=body
    )
    pretty_print(response)
```

8 附录

8.1 Endpoint 列表

对象存储网络和香港节点的 Endpoint 不同。

对象存储网络中的各个地区，使用统一的 OOS API、统计、操作跟踪和 IAM API 的 Endpoint。对象存储网络 Endpoint 列表如下：

- OOS API Endpoint: oos-cn.ctyunapi.cn，支持 HTTP 和 HTTPS。
- 统计 API Endpoint: oos-cn-mg.ctyunapi.cn，支持 HTTP 和 HTTPS。
- 操作跟踪 API Endpoint：oos-cn-cloudtrail.ctyunapi.cn，支持 HTTPS。
- IAM Endpoint: oos-cn-iam.ctyunapi.cn，支持 HTTPS。

说明：对于对象存储网络中的 OOS API，如果您的数据存储在某个资源池，建议您直接使用该资源池的 Endpoint。Endpoint 列表如下（Endpoint 列表仅为资源池 Endpoint 访问信息描述，与资源状态无关联）：

地区	OOS API Endpoint
郑州	oos-hazz.ctyunapi.cn
沈阳	oos-lnsy.ctyunapi.cn
四川成都	oos-sccd.ctyunapi.cn
乌鲁木齐	oos-xjwlmq.ctyunapi.cn
甘肃兰州	oos-gslz.ctyunapi.cn
山东青岛	oos-sdqd.ctyunapi.cn
贵州贵阳	oos-gzgy.ctyunapi.cn
湖北武汉	oos-hbwh.ctyunapi.cn
西藏拉萨	oos-xzls.ctyunapi.cn
安徽芜湖	oos-ahwh.ctyunapi.cn
广东深圳	oos-gdsz.ctyunapi.cn
江苏苏州	oos-jssz.ctyunapi.cn
上海 2	oos-sh2.ctyunapi.cn

香港节点分为**香港精品网络**和**香港普通网络**两种方式，精品网和普通网 OOS API 的 Endpoint 不同，统计、操作跟踪和 IAM API 的 Endpoint 相同：

- 香港精品网 OOS API Endpoint: oos-cnhk-hqnet.ctyunapi.cn，香港普通网 OOS API Endpoint：oos-cnhk-nqnet.ctyunapi.cn。支持 HTTP 和 HTTPS。
- 统计 API Endpoint: oos-cnhk-mg.ctyunapi.cn，支持 HTTP 和 HTTPS。
- 操作跟踪 API Endpoint：oos-cnhk-cloudtrail.ctyunapi.cn，支持 HTTPS。
- IAM Endpoint：oos-cnhk-iam.ctyunapi.cn，支持 HTTPS。