天翼云

对象存储

(Object-Oriented Storage,OOS)

Go 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）是为客户提供一种海量、弹性、廉价、高可用的存储服务。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，也可以通过 OOS 提供的 SDK 来调用 OOS 服务。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。

2 使用条件

2.1 先决条件

用户需要具备以下条件，然后才能够使用 OOS Go SDK 版本：安装 go 1.19 及以上版本。

2.2 获取访问凭证

AccessKeyId 和 SecretKey 是用户访问 OOS 的密钥，OOS 会通过它来验证用户的资源请求，请妥善保管。关于 AccessKeyId 和 SecretKey 的介绍，请阅读《OOS 开发者文档》。

2.3 开发者文档

完整的通用的开发者文档的下载地址为《OOS 开发者文档》，开发者在 SDK 前请务必首先阅读《OOS 开发者文档》，以便掌握各请求参数和响应参数的使用方法。

2.4 下载及安装

从官方渠道下载压缩包 **oos-go-sdk-6.2.0.zip**，放到相应位置后并解压。解压后包含代码压缩包和《OOS GO 开发者指南》两个文件。文件“oos-go-sdk-6.2.0.zip”为 sdk 源码，解压后目录“oos”的 go 文件为 sdk 的实现目录，目录“sample”下的文件为 sdk 的使用示例代码。根目录下 sample.go 为 main 文件，可以直接测试运行 sample 里的代码。

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 key 凭证和服务端地址 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 的服务端地址请参见《OOS 开发者文档》。

以下代码示例用于进行 key 和 endpoint 的设置：

```
/*
Sample code's env configuration. You need to specify them with the actual configuration
if you want to run sample code
*/
```

```
const (
    endpoint      string = "http://oos-cn.ctyunapi.cn"
    iamEndpoint   string = "https://oos-cn-iam.ctyunapi.cn"
    accessKey     string = "ak" // your access key
    secretKey     string = "sk" //   your secret key
    bucketName    string = "bucketName"
    userName string = "UserName"

    // The object name in the sample code
    objectKey          string = "your-object"
    objectKeyMultipart string = "your-multipart-object"

    // The local files to run sample code.
    localFile          string = "localFile"
    localFileMultipart string = "localFileMultipart"
)
```

参数说明

参数	描述	是否必须
accessKey	AccessKey。	是
secretKey	SecretAccessKey。	是
endpoint	OOS 域名如 https://oos-cn.ctyunapi.cn。	是
iamEndpoint	IAM 域名。	访问 iam 服务时必填
bucketName	Bucket 名称。	是
userName	用户名称。	是
objectKey	对象名称	是
objectKeyMultipart	分段对象名称。	否
localFile	要上传到 oos 的本地文件绝对路径.	否
localFileMultipart	要分段上传的本地文件绝对路径.	否

3.2 初始化 client

以下示例为创建 Bucket 相关操作的 client：

```
clientOptionV4 := oos.V4Signature(true) // 使用v4签名
//使用payload方式签名，只有v4签名有效。
isEnableSha256 := oos.EnableSha256ForPayload(true)
 //如果是iam接口，则只需将endpoint参数更换为iamEndpoint，注意：iam操作仅支持v4签名方式
client, err := oos.New(endpoint, accessKey, secretKey, clientOptionV4, isEnableSha256)
if err != nil {
    HandleError(err)
}
```

以下示例为创建 object 操作的 client(在创建 Bucket 操作的 client 基础上加了一步)：

```
clientOptionV4 := oos.V4Signature(true)   // 使用v4签名
//使用payload方式签名，只有v4签名有效。
isEnableSha256 := oos.EnableSha256ForPayload(false)
client, err := oos.New(endpoint, accessKey, secretKey, clientOptionV4, isEnableSha256)
if err != nil {
    HandleError(err)
}
// Get bucket 来调用object相关所有接口
bucket, err := client.Bucket(bucketName)
if err != nil {
    return nil, err
}
```

4 OOS 服务代码示例

说明：OOS 的服务代码示例全部在 sample 目录下。

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。

注意：只有验证用户可以执行该操作，匿名用户不能执行该操作。

- 示例代码

```
lbr, err := client.ListBuckets()
if err != nil {
    HandleError(err)
}

for _, bucket := range lbr.Buckets {
    fmt.Printf("list bucket name :%s CreateData:%s Owner ID:%s\n", bucket.Name,
bucket.CreationDate.String(),
        lbr.Owner.ID)
}
```

4.1.2 GET Regions

获取资源池中的索引位置和数据位置列表。

注意：香港节点不支持此接口。

- **示例代码**

```
ret, err := client.GetRegions()
if err != nil {
    HandleError(err)
}
```

4.2 关于 Bucket 的操作

4.2.1 PUT Bucket

PUT Bucket 操作可以用来：

- 创建一个新的 Bucket；
- 对已有 Bucket 的 Bucket ACL 进行修改；
- 对数据位置进行修改，但是不能修改索引位置（香港节点不支持）。

只有根用户和具有 PUT Bucket 权限的子用户才能创建 Bucket。

Bucket 的命名规范如下：

- Bucket 名称必须全局唯一；
- Bucket 名称长度介于 3 到 63 字节之间；
- Bucket 名称只能由小写字母、数字、短横线（-）和点（.）组成；
- Bucket 名称可以由一个或者多个小节组成，小节之间用点（.）隔开，各个小节需要：
  - 只能包含小写字母、数字和短横线（-）；
  - 必须以小写字母或者数字开始；
  - 必须以小写字母或者数字结束。
- Bucket 名称不能是一组或多组“数字.数字”的组合（如 192.162.0.1）；
- Bucket 名称中不能包含双点（..）、横线点（-.）和点横线（.-）；
- 不允许使用非法敏感字符，例如暴恐涉政相关信息等。

- **示例代码 1**

  当创建索引域为成都，数据域为本地时，使用下面的示例代码创建 Bucket：

```
confLocal, err := oos.BuildCreateBucketConfigLocal("ChengDu")
if err != nil {
    HandleError(err)
}
err = client.CreateBucket(bucketNameTest, confLocal)
if err != nil {
    HandleError(err)
}
```

- **示例代码 2**

  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：

```
locationist := make([]string, 0) //设置datalocation
locationist = append(locationist, "ChengDu")
locationist = append(locationist, "GuiYang")
//datalocation 为成都贵阳，metalocation 为成都，代码示例如下。
conf, err := oos.BuildCreateBucketConfigSpecified("ChengDu", locationist, true)
if err != nil {
    HandleError(err)
}

err = client.CreateBucket(bucketName, conf, oos.ACL(oos.ACLPublicRead))
if err != nil {
    HandleError(err)
}
```

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。

注意：香港节点不支持此接口。

- 示例代码

```
ret, err := client.GetBucketLocation(bucketName)
if err != nil {
    HandleError(err)
}
```

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

```
// Get bucket ACL
gbar, err := client.GetBucketACL(bucketName)
if err != nil {
    HandleError(err)
}
```

4.2.4 Get Bucket(List Objects)

此操作用来返回 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。

- **示例代码**

```
lor, err := bucket.ListObjects()
if err != nil {
    HandleError(err)
}
```

4.2.5 DELETE Bucket

此操作用来删除 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

```
err = client.DeleteBucket(bucketName)
if err != nil {
    HandleError(err)
}
```

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。

- 示例代码

```
text := string("{\"Version\":\"2012-10-17\",\"Id\":\"http referer policy
example\",\"Statement\":[{\"Sid\":\"Allow get requests referred by www.mysite.com \",
\"Effect\":\"Allow\",\"Principal\":{ \"AWS\": [\"*\"] }, \"Action\":\"s3:*\",
\"Resource\":\"arn:aws:s3:::bucketName/*\",
\"Condition\":{\"StringLike\":{\"aws:Referer\":[\"http://www.mysite.com/*\"]} }} ]}")

err = client.SetBucketPolicy(bucketName, text)
if err != nil {
    HandleError(err)
}
```

4.2.7 GET Bucket Policy

此操作用来返回指定 Bucket 的 Policy。只有根用户和拥有 GET Bucket Policy 权限的子用户才能执行此操作，否则会返回 403 AccessDenied 错误。如果 Bucket 没有 Policy，返回 404，NoSuchPolicy 错误。

- 示例代码

```
text, err = client.GetBucketPolicy(bucketName)
if err != nil {
    HandleError(err)
}
```

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，否则会返回 403 AccessDenied 错误。如果 Bucket 配置了 Policy，删除成功，返回 200 0K。如果 Bucket 没有配置 Policy，返回 204 NoContent。

- 示例代码

```
err = client.DeleteBucketPolicy(bucketName)
if err != nil {
    HandleError(err)
}
```

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。如果您需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，您需要为桶绑定已通过备案的自定义域名，请联系天翼云客服申请绑定自定义域名。
- 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。

- **示例代码**

```
config = oos.WebsiteConfiguration{}
config.IndexDocument.Suffix = "index.html"
config.ErrorDocument.Key = "error.html"
condition := oos.Condition{HttpErrorCodeReturnedEquals: "403", KeyPrefixEquals: "docs"}
redirect := oos.Redirect{HostName: "www.example.com", Protocol: "http",
ReplaceKeyPrefixWith: "documents/"}
rule := oos.RoutingRule{Condition: &condition, Redirect: &redirect}
config.RoutingRules = []oos.RoutingRule{rule}
err = client.SetBucketWebsite(bucketName, config)
if err != nil {
    HandleError(err)
}
```

4.2.10 GET Bucket Website

此操作用来获得指定 Bucket 的 website。只有根用户和拥有 GET Bucket WebSite 权限的子用户才能执行此操作，否则会返回 403 AccessDenied 错误。

- 示例代码

```
webSite, err := client.GetBucketWebsite(bucketName)
if err != nil {
    HandleError(err)
}
fmt.Println("website info: \n IndexDocument:" + webSite.IndexDocument.Suffix + "\n
ErrorDocument:" + webSite.ErrorDocument.Key)
for _, s := range *webSite.RoutingRules {
    fmt.Println(s.Redirect)
    fmt.Println(s.Condition)
}
```

4.2.11 DELETE Bucket Website

此操作用来删除指定 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，否则会返回 403 AccessDenied 错误。如果 Bucket 没有设置 Website 或者 Website 删除成功，返回 200 OK。

- 示例代码

```
err = client.DeleteBucketWebsite(bucketName)
if err != nil {
    HandleError(err)
}
```

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。

响应中最多返回 1000 个分片上传过程的信息，它既是响应能返回的最大分片上传过程数目，也是请求的默认值。用户也可以通过设置 max-uploads 参数来限制响应中的分片上传过程数目。如果当前的分片上传过程数超出了这个值，则响应中会包含一个值为 true 的 IsTruncated 元素。如果用户要列出多于这个值的分片上传过程信息，则需要继续调用 List Multipart Uploads 请求，并在请求中设置 *key-marker* 和 *upload-id-marker* 参数。

在响应体中，分片上传过程的信息通过 key 来排序。如果用户的应用程序中启动了多个使用同一 key 对象开头的分片上传过程，那么响应体中分片上传过程首先是通过 key 来排序，在相同 key 的分片上传内部则是按上传启动的起始时间的升序来进行排列。

- **示例代码**

```
lmurs, err := bucket.ListMultipartUploads()
if err != nil {
    HandleError(err)
}
```

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，否则会返回 403 AccessDenied 错误。

- 示例代码

```
// Create target bucket to store the logging files.
var targetBucketName = "testbucket"

// Case 1: Set the logging for the object prefixed with "prefix-1" and save their access logs to the target bucket
err := client.SetBucketLogging(bucketName, targetBucketName, "prefix-1", true)
if err != nil {
    HandleError(err)
}
```

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，否则会返回 403 AccessDenied 错误。

- 示例代码

```
logInfo, err := client.GetBucketLogging(bucketName)
if err != nil {
    HandleError(err)
}
fmt.Printf("Bucket Logging: %s \r\n", logInfo.LoggingEnabled.TargetBucket)
```

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。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。

- 示例代码

```
exist, errHead := client.HeadBucket(bucketNameTest)
if errHead != nil {
    HandleError(errHead)
} else if exist {
    fmt.Println("bucket " + bucketNameTest + " exist")
}
```

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/转换存储类型的时间。

通过设置存储桶的生命周期规则，可以：

- 删除与生命周期规则匹配的对象。当对象的生命周期到期时，OOS 会异步删除它们。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。但对象到期被删除后，用户将不需要为到期的对象付费。OOS 删除到期对象后，会在 Bucket log 中记录一条日志，操作项是"OOS.EXPIRE.OBJECT"。
- 将与生命周期规则匹配的对象由标准存储转换为低频访问存储。OOS 转换存储类型为低频访问存储后，会在 access logs 中记录一条日志，操作项是"OOS.TRANSITION_SIA.OBJECT"。

- **示例代码**

```
/* Case 1: Set the lifecycle. The rule ID is id1 and the applied objects' prefix is one
and expired time is 12/18/2022 */
var rule1 = oos.BuildLifecycleExpirRuleByDate("id1", "prefix-test/", false, 2022, 12,
18)
var rules = []oos.LifecycleRule{rule1}
err := client.SetBucketLifecycle(bucketName, rules)
if err != nil {
    HandleError(err)
}

/* Case 2: Set the lifecycle, The rule ID is id2 and the applied objects' prefix is two
and the expired time is three days after the object created. */
var rule2 = oos.BuildLifecycleExpirRuleByDays("id2", "two", true, 3)
rules = []oos.LifecycleRule{rule2}
err = client.SetBucketLifecycle(bucketName, rules)
if err != nil {
    HandleError(err)
}
```

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

```
gbl, err := client.GetBucketLifecycle(bucketName)
if err != nil {
    HandleError(err)
}
```

4.2.18 DELETE Buckete 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

```
err = client.DeleteBucketLifecycle(bucketName)
if err != nil {
    HandleError(err)
}
```

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**

```
rule := oos.CORSRule{}
rule.AllowedOrigin = []string{"http://www.example1.com", "http://ctyun.cn"}
rule.AllowedMethod = []string{"PUT", "POST", "DELETE"}
rule.AllowedHeader = []string{"*", "x-amz-*"}
rule.ExposeHeader = []string{"JavaScript XMLHttpRequest"}
rule.MaxAgeSeconds = 1000
corsRules := []oos.CORSRule{rule}

err = client.SetBucketCors(bucketName, corsRules)
if err != nil {
    HandleError(err)
}
```

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，否则会返回 403 AccessDenied 错误。

- 示例代码

```
cors, err := client.GetBucketCors(bucketName)
if err != nil {
    HandleError(err)
}
```

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，否则会返回 403 AccessDenied 错误。

- 如果 Bucket 配置了 CORS，删除成功，返回 200 0K。
- 如果 Bucket 没有配置 CORS，返回 204 NoContent。

- **示例代码**

```
err = client.DeleteBucketCors(bucketName)
if err != nil {
    HandleError(err)
}
```

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。

- 示例代码

```
bucketObjectLock := oos.BuildBucketObjectLockByDays(string(oos.BucketModeCompliance),
10000, false)
err := client.SetBucketObjectLock(bucketName, bucketObjectLock)
if err != nil {
    HandleError(err)
}

bucketObjectLock = oos.BuildBucketObjectLockByYears(string(oos.BucketModeCompliance),
10, false)
err = client.SetBucketObjectLock(bucketName, bucketObjectLock)
if err != nil {
    HandleError(err)
}
```

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

```
out, err := client.GetBucketObjectLock(bucketName)
if err != nil {
    HandleError(err)
}
```

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

```
err = client.DeleteBucketObjectLock(bucketName)
if err != nil {
    HandleError(err)
}
```

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

```
var val = "红豆生南国，春来发几枝。愿君多采撷，此物最相思。"

// Case 1: Upload an object from a string
objectName := "test.txt"
err = bucket.PutObject(objectName, strings.NewReader(val))
if err != nil {
    HandleError(err)
}

// Case 2: Upload an object whose value is a byte[]
err = bucket.PutObject("test_byte", bytes.NewReader([]byte(val)))
if err != nil {
    HandleError(err)
}

// Case 3: Upload the local file with file handle, user should open the file at first.
fd, err := os.Open(localFile)
if err != nil {
    HandleError(err)
}
defer fd.Close()

err = bucket.PutObject("test_file_fd", fd)
if err != nil {
    HandleError(err)
}

// Case 4: Upload an object with local file name, user need not open the file.
err = bucket.PutObjectFromFile("test_file_path", localFile)
if err != nil {
    HandleError(err)
```

```
}

// Case 5: Upload an object with specified properties,
PutObject/PutObjectFromFile/UploadFile also support this feature.
options := []oos.Option{
    oos.Expires(futureDate),
    oos.Meta("mykey", "myval"),
    oos.StorageClass("STANDARD"),
    oos.Connection("close"),
    oos.ContentType("jpeg"),
    // oos.ObjectDataLocation(oos.BuildObjectLocalDataLocation(false)),
    oos.ObjectDataLocation(oos.BuildObjectSpecifiedDataLocation("ChengDu", false)),
}

err = bucket.PutObject("test_meta", strings.NewReader(val), options...)
if err != nil {
    HandleError(err)
}
```

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

```
result, err := bucket.DoGetObject(&oos.GetObjectRequest{ObjectKey: objectKey},
[]oos.Option{})
if err != nil {
    HandleError(err)
}

fmt.Println(result.Response.Headers.Get("x-ctyun-metadata-location"))
fmt.Println(result.Response.Headers.Get("x-ctyun-data-location"))

// Case 1: Download the object into ReadCloser(). The body needs to be closed
var body io.ReadCloser
body, err = bucket.GetObject(objectKey)
if err != nil {
    HandleError(err)
}
data, err := ioutil.ReadAll(body)
body.Close()
if err != nil {
    HandleError(err)
}
fmt.Printf("read data len:%d\n", len(data))

// Case 2: Download the object to local file with file name specified
err = bucket.GetObjectToFile(objectKey, "mynewfile-2.jpg")
if err != nil {
    HandleError(err)
}
```

4.3.3 DELETE Object

此操作用来移除指定的对象，用户要对对象所在的 Bucket 拥有写权限才可以执行此操作。

- 示例代码

```
err = bucket.DeleteObject(objectKey)
if err != nil {
    return err
}
```

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

```
var descObjectKey = "descobject"
_, err = bucket.CopyObject(objectKey, descObjectKey)
if err != nil {
    HandleError(err)
}
```

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）时都必须指定该 ID。用户也可以在表示整个分片上传完成的合并分片的请求中指定该 ID。或者在用户放弃该分片上传操作时指定该 ID。

- 示例代码

```
imur, err := bucket.InitiateMultipartUpload(objectKeyMultipart,
oos.ObjectDataLocation(oos.BuildObjectSpecifiedDataLocation("ChengDu", true)))
if err != nil {
    HandleError(err)
}
```

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，所有分片的都不小于 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。

```
var partSize int64 = 5 * 1024 * 1024
chunks, err := oos.SplitFileByPartSize(localFileMultipart, partSize)
if err != nil {
    HandleError(err)
}
uploadParts := make([]oos.UploadPart, 0)
for index, chunk := range chunks {
    data := readFile(localFileMultipart, chunk.Offset, chunk.Size)
    uploadPart, err := bucket.UploadPart(imur, data, chunk.Size, index+1)
    if err != nil {
        HandleError(err)
    }
    uploadParts = append(uploadParts, uploadPart)
}
```

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**

```
compleRet, err := bucket.CompleteMultipartUpload(imur, uploadParts)
if err != nil {
    HandleError(err)
}
```

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

```
imur, err := bucket.InitiateMultipartUpload(objectKeyMultipart,
oos.ObjectDataLocation(oos.BuildObjectSpecifiedDataLocation("ChengDu", true)))
if err != nil {
    HandleError(err)
}
err = bucket.AbortMultipartUpload(imur)
if err != nil {
    HandleError(err)
}
```

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**

```
lmurs, err := bucket.ListMultipartUploads()
if err != nil {
    HandleError(err)
}

// list part test
upload := lmurs.Uploads[0]
imurTemp := oos.InitiateMultipartUploadResult{Bucket: bucket.BucketName,
    Key: upload.Key, UploadID: upload.UploadID}
lpr, err := bucket.ListUploadedParts(imurTemp)
if err != nil {
    HandleError(err)
}
```

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。如果请求头中带 x-amz-copy-source-range 参数，可以拷贝源 Object 中的一部分，但源对象必须大于 5GiB。PartNumber 为新对象分片号，UploadId 为新对象的分片上传 ID。

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 Copy Part 接口时加入该 ID。

- **示例代码**

```
imur, err = bucket.InitiateMultipartUpload(objectKeyMultipart,
oos.StorageClass(oos.StorageClassStandard))
if err != nil {
    HandleError(err)
}
var startPosition int64 = 0
partSize = 5 * 1024 * 1024
partNumber := 1
bucket.UploadPartCopy(imur, bucketName, objectKey, startPosition, partSize, partNumber)
```

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。

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。

最后，批量删除功能必须使用 Content-MD5 请求头，OOS 使用此头来保证请求体在传输过程中没有被修改。

- **示例代码**

```
delRes, err := bucket.DeleteObjects([]string{objectKey + "one", objectKey + "two"},
oos.DeleteObjectsQuiet(false))
if err != nil {
    HandleError(err)
}
```

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。
生成共享链接示例：

- 示例代码

```
signedURL, err := bucket.SignURL(objectName, oos.HTTPGet, 3600)
if err != nil {
    HandleError(err)
}
```

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

```
meta, err := bucket.HeadObject(objectKey)
if err != nil {
    HandleError(err)
}
```