阿里云
大数据轻量专有云

开发指南

产品版本：V2.0.0

文档版本：20180615

为了无法计算的价值 | 阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的”现状“、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含”阿里云”、Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	**禁止**： 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	**警告**： 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	**说明**： 您也可以通过按**Ctrl** + **A**选中全部文件。
>	多级菜单递进。	**设置** > **网络** > **设置网络类型**
粗体	表示按键、菜单、页面名称等UI元素。	单击 **确定**。
courier字体	命令。	执行 cd /d C:/windows 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	bae log list --instanceid *Instance_ID*
[]或者[a|b]	表示可选项，至多选择一个。	ipconfig *[-all|-t]*
{}或者{a|b}	表示必选项，至多选择一个。	swich *{stand | slave}*

目录

1 对象存储OSS

1.1 API参考

1.1.1 OSS API 文档简介

阿里云对象存储服务（Object Storage Service，简称OSS），是阿里云对外提供的海量、安全、低成本、高可靠的云存储服务。用户可以通过本文档提供的简单的REST接口，在任何时间、任何地点、任何互联网设备上进行上传和下载数据。基于OSS，用户可以搭建出各种多媒体分享网站、网盘、个人和企业数据备份等基于大规模数据的服务。

请确保在使用这些接口前，已充分了解了OSS产品说明、使用协议和收费方式。

1.1.2 API概览

关于Service操作

表 1-1: 关于Service操作

API	描述
GetService	得到该账户下所有Bucket

关于Bucket的操作

表 1-2: 关于Bucket的操作

API	描述
Put Bucket	创建Bucket
Put Bucket ACL	设置Bucket访问权限
Put Bucket Logging	开启Bucket日志
Put Bucket Website	设置Bucket为静态网站托管模式
Put Bucket Referer	设置Bucket的防盗链规则
Put Bucket Lifecycle	设置Bucket中Object的生命周期规则
Put Bucket Tagging	为Bucket添加一组标签
Get Bucket Acl	获得Bucket访问权限

API	描述
Get Bucket Location	获得Bucket所属的数据中心位置信息
Get Bucket Logging	查看Bucket的访问日志配置情况
Get Bucket Website	查看Bucket的静态网站托管状态
Get Bucket Referer	查看Bucket的防盗链规则
Get Bucket Lifecycle	查看Bucket中Object的生命周期规则
Get Bucket Tagging	获取某个Bucket关联的所有标签
Delete Bucket	删除Bucket
Delete Bucket Logging	关闭Bucket访问日志记录功能
Delete Bucket Website	关闭Bucket的静态网站托管模式
Delete Bucket Lifecycle	删除Bucket中Object的生命周期规则
DeleteBucketTagging	删除某个Bucket关联的所有标签
Get Bucket(List Object)	获得Bucket中所有Object的信息
Get Bucket Info	获取Bucket信息
List Bucket By Tagging	根据用户指定的Tag Key和Tag Value来List出符合条件的Bucket

关于Object的操作

表 1-3: 关于Object的操作

API	描述
Put Object	上传object
Copy Object	拷贝一个object成另外一个object
Get Object	获取Object
Append Object	以追加写的方式上传文件
Delete Object	删除Object
Delete Multiple Objects	删除多个Object
Head Object	获得Object的meta信息
Get Object Meta	获取某个Bucket下的某个Object的基本meta信息
Post Object	使用Post上传Object

API	描述
Put Object ACL	设置Object ACL
Get Object ACL	获取Object ACL信息
Put Symlink	创建符号链接
Get Symlink	获取符号链接

关于Multipart Upload的操作

表 1-4: 关于Multipart Upload的操作

API	描述
Initiate Multipart Uploade	初始化MultipartUpload事件
Upload Part	分块上传文件
Upload Part Copy	分块复制上传文件
Complete Multipart Upload	完成整个文件的Multipart Upload上传
Abort Multipart Upload	取消Multipart Upload事件
List Multipart Uploads	罗列出所有执行中的Multipart Upload事件
List Parts	罗列出指定Upload ID所属的所有已经上传成功Part

跨域资源共享(CORS)

表 1-5: 跨域资源共享(CORS)

API	描述
Put Bucket CORS	在指定Bucket设定一个CORS的规则
Get Bucket CORS	获取指定的Bucket目前的CORS规则
Delete Bucket CORS	关闭指定Bucket对应的CORS功能并清空所有规则
Option Object	跨域访问preflight请求

1.1.3 访问控制

1.1.3.1 用户签名验证

OSS通过使用AccessKeyId/ AccessKeySecret对称加密的方法来验证某个请求的发送者身份。

AccessKeyId用于标示用户，AccessKeySecret是用户用于加密签名字符串和OSS用来验证签名字

符串的密钥，其中AccessKeySecret必须保密，只有用户和OSS知道。AccessKey 根据所属账号的类型有所区分：

- 阿里云账户AccessKey：每个阿里云账号提供的AccessKey拥有对拥有的资源有完全的权限。
- RAM账户AccessKey：RAM账户由阿里云账号授权生成，所拥的AccessKey拥有对特定资源限定的操作权限。
- STS临时访问凭证：由阿里云账号或RAM账号生成，所拥的AccessKey在限定时间内拥有对特定资源限定的操作权限。过期权限收回。

当用户想以个人身份向OSS发送请求时，需要首先将发送的请求按照OSS指定的格式生成签名字符串；然后使用AccessKeySecret对签名字符串进行加密产生验证码。 OSS收到请求以后，会通过AccessKeyId找到对应的AccessKeySecret，以同样的方法提取签名字符串和验证码，如果计算出来的验证码和提供的一样即认为该请求是有效的；否则，OSS将拒绝处理这次请求，并返回HTTP 403错误。

1.1.3.2 在Header中包含签名

户可以在HTTP请求中增加 Authorization 的Header来包含签名（Signature）信息，表明这个消息已被授权。

Authorization字段计算的方法

```
Authorization = "OSS " + AccessKeyId + ":" + Signature
Signature = base64(hmac-sha1(AccessKeySecret,
 VERB + "\n"
 + Content-MD5 + "\n"
 + Content-Type + "\n"
 + Date + "\n"
 + CanonicalizedOSSHeaders
 + CanonicalizedResource))
```

- AccessKeySecret 表示签名所需的密钥
- VERB表示HTTP 请求的Method，主要有PUT，GET，POST，HEAD，DELETE等
- \n 表示换行符
- Content-MD5 表示请求内容数据的MD5值，对消息内容（不包括头部）计算MD5值获得128比特位数字，对该数字进行base64编码而得到。该请求头可用于消息合法性的检查（消息内容是否与发送时一致），如”eB5eJF1ptWaXm4bijSPyxw==”，也可以为空。详情参看*RFC2616 Content-MD5*
- Content-Type 表示请求内容的类型，如”application/octet-stream”，也可以为空
- Date 表示此次操作的时间，且必须为GMT格式，如”Sun, 22 Nov 2015 08:16:38 GMT”

- CanonicalizedOSSHeaders 表示以 **x-oss-** 为前缀的http header的字典序排列
- CanonicalizedResource 表示用户想要访问的OSS资源

其中，Date和CanonicalizedResource不能为空；如果请求中的Date时间和OSS服务器的时间差15分钟以上，OSS服务器将拒绝该服务，并返回HTTP 403错误。

构建CanonicalizedOSSHeaders的方法

所有以 **x-oss-** 为前缀的HTTP Header被称为CanonicalizedOSSHeaders。它的构建方法如下：

1. 将所有以 **x-oss-** 为前缀的HTTP请求头的名字转换成 **小写** 。如X-OSS-Meta-Name: TaoBao转换成x-oss-meta-name: TaoBao。
2. 如果请求是以STS获得的AccessKeyId和AccessKeySecret发送时，还需要将获得的security-token值，以 x-oss-security-token:security-token 的形式加入到签名字符串中。
3. 将上一步得到的所有HTTP请求头按照名字的字典序进行升序排列。
4. 删除请求头和内容之间分隔符两端出现的任何空格。如x-oss-meta-name: TaoBao转换成：x-oss-meta-name:TaoBao。
5. 将每一个头和内容用 \n 分隔符分隔拼成最后的CanonicalizedOSSHeaders。

说明：

- CanonicalizedOSSHeaders可以为空，无需添加最后的 \n。
- 如果只有一个，则如 x-oss-meta-a\n，注意最后的\n。
- 如果有多个，则如 x-oss-meta-a:a\nx-oss-meta-b:b\nx-oss-meta-c:c\n, 注意最后的\n。

构建CanonicalizedResource的方法

用户发送请求中想访问的OSS目标资源被称为CanonicalizedResource。它的构建方法如下：

1. 将CanonicalizedResource置成空字符串 ""；
2. 放入要访问的OSS资源 /BucketName/ObjectName
   （无ObjectName则CanonicalizedResource为/BucketName/，如果同时也没有BucketName则为/）
3. 如果请求的资源包括子资源(SubResource)，那么将所有的子资源按照字典序，从小到大排列并以 & 为分隔符生成子资源字符串。在CanonicalizedResource字符串尾添加 ？和子资源字符串。此时的CanonicalizedResource如：/BucketName/ObjectName?acl&uploadId=UploadId
4. 如果用户请求在指定了查询字符串(QueryString，也叫Http Request Parameters)，那么将这些查询字符串及其请求值按照字典序，从小到大排列，以 & 为分隔符，按参数添加

到CanonicalizedResource中。此时的CanonicalizedResource如：/BucketName/ObjectName?acl&response-content-type=ContentType&uploadId=UploadId。

说明：

- OSS目前支持的子资源(sub-resource)包括：acl，uploads，location，cors，logging，website，referer，lifecycle，delete，append，tagging，objectMeta，uploadId，partNumber，security-token，position，img，style，styleName，replication，replicationProgress，replicationLocation，cname，bucketInfo，comp，qos，live，status，vod，startTime，endTime，symlink，x-oss-process，response-content-type，response-content-language，response-expires，response-cache-control，response-content-disposition，response-content-encoding等。
- 子资源(sub-resource)有三种类型：
  - 资源标识，如子资源中的acl，append，uploadId，symlink等。
  - 指定返回Header字段，如 response-***。
  - 文件（Object）处理方式，如 x-oss-process，用于文件的处理方式。

计算签名头规则

1. 签名的字符串必须为 UTF-8 格式。含有中文字符的签名字符串必须先进行 UTF-8 编码，再与AccessKeySecret计算最终签名。
2. 签名的方法用*RFC 2104*中定义的HMAC-SHA1方法，其中Key为 AccessKeySecret 。
3. Content-Type 和 Content-MD5 在请求中不是必须的，但是如果请求需要签名验证，空值的话以换行符 \n 代替。
4. 在所有非HTTP标准定义的header中，只有以 x-oss- 开头的header，需要加入签名字符串；其他非HTTP标准header将被OSS忽略（如上例中的x-oss-magic是需要加入签名字符串的）。
5. 以 x-oss- 开头的header在签名验证前需要符合以下规范：
   - header的名字需要变成小写。
   - header按字典序自小到大排序。
   - 分割header name和value的冒号前后不能有空格。
   - 每个header之后都有一个换行符\n，如果没有header，CanonicalizedOSSHeaders就设置为空。

签名示例

假如AccessKeyId是4***7，AccessKeySecret是O***V

表 1-6: 签名示例

请求	签名字符串计算公式	签名字符串
PUT /nelson HTTP/1.0 Content-MD5: eB5eJF1ptWaXm4bijSPyxw== Content-Type: text/html Date: Thu, 17 Nov 2005 18:49:58 GMT Host: oss-example.regionid.example.com X-OSS-Meta-Author: ***@bar.com X-OSS-Magic: a***a	Signature = base64(hmac-sha1(AccessKeySecret,VERB + “\n” + Content-MD5 + “\n” + Content-Type + “\n” + Date + “\n” + CanonicalizedOSSHeaders+ CanonicalizedResource))	"PUT\n ***w==\n text/html\n Thu, 17 Nov 2005 18:49:58 GMT\n x-oss-magic:a***a\nx-oss-meta-author:***@bar.com\n/oss-example/nelson"

可用以下方法计算签名(Signature):

```
import base64
import hmac
import sha
h = hmac.new("O***V",
 "PUT\n***M=\ntext/html\nThu, 17 Nov 2005 18:49:58 GMT\nx-oss-magic:abracadabra\nx-oss-meta-author:***@bar.com\n/oss-example/nelson", sha)
Signature = base64.b64encode(h.digest())
print("Signature: %s" % Signature)
```

签名(Signature)计算结果应该为 2***A=，因为Authorization = “OSS” + AccessKeyId + “:” + Signature，所以最后Authorization为“OSS 4***7:2***A=”然后加上Authorization头来组成最后需要发送的消息：

```
PUT /nelson HTTP/1.0
Authorization:OSS 4***7:2***A=
Content-Md5: e***w==
Content-Type: text/html
Date: Thu, 17 Nov 2005 18:49:58 GMT
Host: oss-example.regionid.example.com
X-OSS-Meta-Author: ***@bar.com
X-OSS-Magic: a***a
```

细节分析

1. 如果传入的AccessKeyId不存在或inactive，返回403 Forbidden。错误码：InvalidAccessKeyId。

2. 若用户请求头中Authorization值的格式不对，返回400 Bad Request。错误码：InvalidArgument。
3. OSS所有的请求都必须使用HTTP 1.1协议规定的GMT时间格式。其中，日期的格式为：date1 = 2DIGIT SP month SP 4DIGIT; day month year (e.g., 02 Jun 1982)上述日期格式中，“天”所占位数都是“2 DIGIT”。因此，“Jun 2”、“2 Jun 1982”和“2-Jun-82”都是非法日期格式。
4. 如果签名验证的时候，头中没有传入Date或者格式不正确，返回403 Forbidden错误。错误码：AccessDenied。
5. 传入请求的时间必须在OSS服务器当前时间之后的15分钟以内，否则返回403 Forbidden。错误码：RequestTimeTooSkewed。
6. 如果AccessKeyId是active的，但OSS判断用户的请求发生签名错误，则返回403 Forbidden，并在返回给用户的response中告诉用户正确的用于验证加密的签名字符串。用户可以根据OSS的response来检查自己的签名字符串是否正确。返回示例：

```
<?xml version="1.0" ?>
<Error>
 <Code>
 SignatureDoesNotMatch
 </Code>
 <Message>
 The request signature we calculated does not match the signature you provided. Check your
key and signing method.
 </Message>
 <StringToSignBytes>
 47 45 54 0a 0a 0a 57 65 64 2c 20 31 31 20 4d 61 79 20 32 30 31 31 20 30 37 3a 35 39 3a 32
35 20 47 4d 54 0a 2f 75 73 72 65 61 6c 74 65 73 74 3f 61 63 6c
 </StringToSignBytes>
 <RequestId>
 1***2
 </RequestId>
 <HostId>
 regionid.example.com
 </HostId>
 <SignatureProvided>
 y***8=
 </SignatureProvided>
 <StringToSign>
 GET
Wed, 11 May 2011 07:59:25 GMT
/oss-example?acl
 </StringToSign>
 <OSSAccessKeyId>
 A***Q
 </OSSAccessKeyId>
</Error>
```

OSS SDK已经实现签名，用户使用OSS SDK不需要关注签名问题。如果您想了解具体语言的签名实现，请参考OSS SDK的代码。OSS SDK签名实现的文件如下表：

表 1-7: OSS SDK签名实现的文件

SDK	签名实现
Java SDK	*OSSRequestSigner.java*
Python SDK	*auth.py*
.Net SDK	*OssRequestSigner.cs*
PHP SDK	*OssClient.php*
C SDK	*oss_auth.c*
JavaScript SDK	*client.js*
Go SDK	*auth.go*
Ruby SDK	*util.rb*
iOS SDK	*OSSModel.m*
Android SDK	*OSSUtils.java*

当您自己实现签名，访问OSS报 **SignatureDoesNotMatch** 错误时，请使用 可视化签名工具 确认签名并排除错误。

常见问题

以消息内容为“123456789”来说，计算这个字符串的Content-MD5

正确的计算方式：

标准中定义的算法简单点说就是：

1. 先计算MD5加密的二进制数组（128位）。

2. 再对这个二进制进行base64编码（而不是对32位字符串编码）。

以Python为例子：

正确计算的代码为：

```
>>> import base64,hashlib
>>> hash = hashlib.md5()
>>> hash.update("0123456789")
>>> base64.b64encode(hash.digest())
'eB5eJF1ptWaXm4bijSPyxw=='
```

需要注意

正确的是：hash.digest()，计算出进制数组（128位）

```
>>> hash.digest()
'x\x1e^$]i\xb5f\x97\x9b\x86\xe2\x8d#\xf2\xc7'
```

常见错误是直接对计算出的32位字符串编码进行base64编码。

例如，错误的是：hash.hexdigest()，计算得到可见的32位字符串编码

```
>>> hash.hexdigest()
'781e5e245d69b566979b86e28d23f2c7'
```

错误的MD5值进行base64编码后的结果：

```
>>> base64.b64encode(hash.hexdigest())
'NzgxZTVlMjQ1ZDY5YjU2Njk3OWI4NmUyOGQyM2YyYzc='
```

1.1.3.3 在URL中包含签名

除了使用Authorization Head，用户还可以在URL中加入签名信息，这样用户就可以把该URL转给第三方实现授权访问。

实现方式

URL签名示例：

```
http://oss-example.regionid.example.com/oss-api.pdf?
AccessKeyId=AccessKeyId&Expires=1141889120&Signature=vjbyPxybdZaNmGa%
2ByT272YEAiv4%3D
```

URL签名，必须至少包含Signature，Expires，AccessKeyId三个参数。

- Expires 这个参数的值是一个UNIX时间（自UTC时间1970年1月1号开始的秒数，详见wiki），用于标识该URL的超时时间。如果OSS接收到这个URL请求的时候晚于签名中包含的Expires参数时，则返回请求超时的错误码。例如：当前时间是1141889060，开发者希望创建一个60秒后自动失效的URL，则可以设置Expires时间为1141889120。
- AccessKeyId 即密钥中的AccessKeyId。
- Signature 表示签名信息。所有的OSS支持的请求和各种Header参数，在URL中进行签名的算法和在Header中包含签名的算法基本一样。

```
Signature = urlencode(base64(hmac-sha1(AccessKeySecret,
 VERB + "\n"
 + CONTENT-MD5 + "\n"
 + CONTENT-TYPE + "\n"
 + EXPIRES + "\n"
 + CanonicalizedOSSHeaders
```

```
+ CanonicalizedResource)))
```

其中，与header中包含签名相比主要区别如下：

1. 通过URL包含签名时，之前的Date参数换成Expires参数。
2. 不支持同时在URL和Head中包含签名。
3. 如果传入的Signature，Expires，AccessKeyId出现不止一次，以第一次为准。
4. 请求先验证请求时间是否晚于Expires时间，然后再验证签名。
5. 将签名字符串放到url时，注意要对url进行urlencode。

- 临时用户URL签名时，需要携带 security-token，格式如下：

```
http://oss-example.regionid.example.com/oss-api.pdf?
AccessKeyId=AccessKeyId&Expires=1141889120&Signature=vjbyPxybdZaNmGa%
2ByT272YEAiv4%3D&security-token=SecurityToken
```

示例代码

URL中添加签名的python示例代码：

```
import base64
import hmac
import sha
import urllib
h = hmac.new("O***V",
 "GET\n\n\n1141889120\n/oss-example/oss-api.pdf",
 sha)
urllib.quote (base64.encodestring(h.digest()).strip())
```

说明：

- 上面为python的示例代码。
- OSS SDK中提供了提供URL签名的方法，使用方法请参看SDK文件中的授权访问章节。
- OSS SDK的URL签名实现，请参看下表。

表 1-8: OSS SDK的URL签名实现文件

SDK	URL签名方法	实现文件
Java SDK	OSSClient.generatePresignedUrl	*OSSClient.java*
Python SDK	Bucket.sign_url	*api.py*
.Net SDK	OssClient.GeneratePresignedUri	*OssClient.cs*
PHP SDK	OssClient.signUrl	*OssClient.php*
JavaScript SDK	signatureUrl	*object.js*

SDK	URL签名方法	实现文件
C SDK	oss_gen_signed_url	*oss_object.c*

细节分析

1. 使用在URL中签名的方式，会将你授权的数据在过期时间以内曝露在互联网上，请预先评估使用风险。
2. PUT和GET请求都支持在URL中签名。
3. 在URL中添加签名时，Signature，Expires，AccessKeyId顺序可以交换，但是如果Signature，Expires，AccessKeyId缺少其中的一个或者多个，返回403 Forbidden。错误码：AccessDenied。
4. 如果访问的当前时间晚于请求中设定的Expires时间，返回403 Forbidden。错误码：AccessDenied。
5. 如果Expires时间格式错误，返回403 Forbidden。错误码：AccessDenied。
6. 如果URL中包含参数Signature，Expires，AccessKeyId中的一个或者多个，并且Head中也包含签名消息，返回消息400 Bad Request。错误码：InvalidArgument。
7. 生成签名字符串时，除Date被替换成Expires参数外，仍然包含content-type、content-md5等上节中定义的Header。（请求中虽然仍然有Date这个请求头，但不需要将Date加入签名字符串中）。

1.1.3.4 临时授权访问

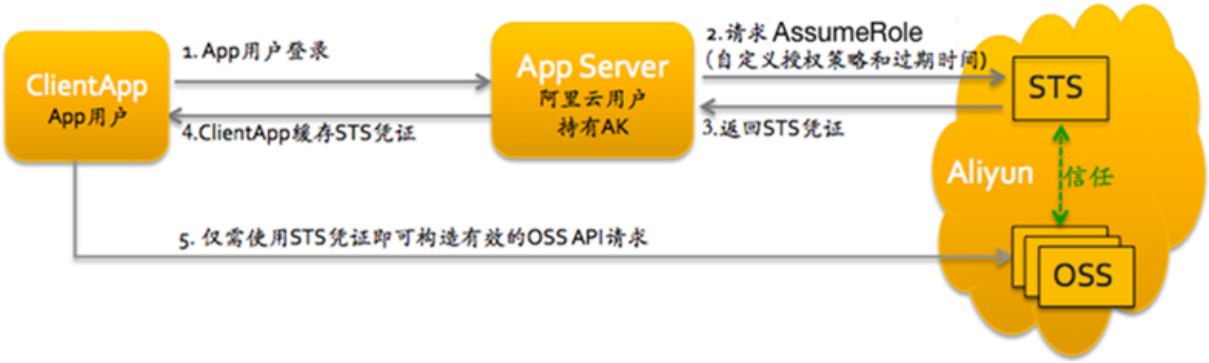
STS介绍

OSS可以通过阿里云STS服务，临时进行授权访问。阿里云STS（Security Token Service）是为云计算用户提供临时访问令牌的Web服务。通过STS，您可以为第三方应用或联邦用户（用户身份由您自己管理）颁发一个自定义时效和权限的访问凭证。第三方应用或联邦用户可以使用该访问凭证直接调用阿里云产品API，或者使用阿里云产品提供的SDK来访问云产品API。

- 您不需要透露您的长期密钥（AccessKey）给第三方应用，只需要生成一个访问令牌并将令牌交给第三方应用即可。这个令牌的访问权限及有效期限都可以由您自定义。
- 您不需要关心权限撤销问题，访问令牌过期后就自动失效。

以APP应用为例，交互流程如下图：

图 1-1: 交互流程



方案的详细描述如下：

1. App用户登录。App用户身份是客户自己管理。客户可以自定义身份管理系统，也可以使用外部Web账号或OpenID。对于每个有效的App用户来说，AppServer是可以确切地定义出每个App用户的最小访问权限。
2. AppServer请求STS服务获取一个安全令牌（SecurityToken）。在调用STS之前，AppServer需要确定App用户的最小访问权限（用Policy语法描述）以及授权的过期时间。然后通过调用STS的AssumeRole（扮演角色）接口来获取安全令牌。角色管理与使用相关内容请参考《RAM使用指南》中的角色管理。
3. STS返回给AppServer一个有效的访问凭证，包括一个安全令牌（SecurityToken）、临时访问密钥（AccessKeyId, AccessKeySecret）以及过期时间。
4. AppServer将访问凭证返回给ClientApp。ClientApp可以缓存这个凭证。当凭证失效时，ClientApp需要向AppServer申请新的有效访问凭证。比如，访问凭证有效期为1小时，那么ClientApp可以每30分钟向AppServer请求更新访问凭证。
5. ClientApp使用本地缓存的访问凭证去请求Aliyun Service API。云服务会感知STS访问凭证，并会依赖STS服务来验证访问凭证，并正确响应用户请求。

STS安全令牌详情，请参考《RAM使用指南》中的角色管理。关键是调用STS服务接口AssumeRole来获取有效访问凭证即可。也可以直接使用STS SDK来调用该方法。

使用STS凭证构造签名请求

用户的客户端端拿到STS临时凭证后，通过其中安全令牌（SecurityToken）以及临时访问密钥（AccessKeyId、AccessKeySecret）构建签名。授权访问签名的构建方式与使用直接使用根账号的AccessKey在Header中包含签名基本一致，关键注意两点：

- 用户使用的签名密钥为STS提供的临时访问密钥（AccessKeyId, AccessKeySecret）。

- 用户需要将安全令牌（SecurityToken）携带在请求header中或者以请求参数的形式放入URI中。这两种形式只能选择其一，如果都选择，oss会返回InvalidArgument错误。
  - 在header中包含头部x-oss-security-token：SecurityToken。计算签名CanonicalizedOSSHeaders时，将x-oss-security-token计算在内。
  - 在URL中携带参数**security-token=SecurityToken**。计算签名CanonicalizedResource时，将security-token当做一个sub-resource计算在内。

1.1.3.5 Bucket权限控制

OSS提供ACL（Access Control List）权限控制方法，OSS ACL提供Bucket级别的权限访问控制，Bucket目前有三种访问权限：public-read-write，public-read和private，它们的含义如下：

- public-read-write：任何人（包括匿名访问）都可以对该bucket中的object进行PUT，Get和Delete操作；所有这些操作产生的费用由该bucket的创建者承担，请慎用该权限。
- public-read：只有该bucket的创建者可以对该bucket内的Object进行写操作（包括Put和Delete Object）；任何人（包括匿名访问）可以对该bucket中的object进行读操作（Get Object）。
- private：只有该bucket的创建者可以对该bucket内的Object进行读写操作（包括Put、Delete和Get Object）；其他人无法访问该Bucket内的Object。

用户新创建一个新Bucket时，如果不指定Bucket权限，OSS会自动为该Bucket设置private权限。对于一个已经存在的Bucket，只有它的创建者可以通过OSS的 Put Bucket Acl接口修改该Bucket的权限。

1.1.4 公共HTTP头定义

公共请求头（Common Request Headers）

OSS的RESTful接口中使用了一些公共请求头。这些请求头可以被所有的OSS请求所使用，其详细定义如下：

表 1-9: 公共请求头

名称	描述
Authorization	用于验证请求合法性的认证信息。 类型：字符串 默认值：无 使用场景：非匿名请求
Content-Length	*RFC2616*中定义的HTTP请求内容长度。

名称	描述
	类型：字符串 默认值：无 使用场景：需要向OSS提交数据的请求
Content-Type	*RFC2616*中定义的HTTP请求内容类型。 类型：字符串 默认值：无 使用场景：需要向OSS提交数据的请求
Date	HTTP 1.1协议中规定的GMT时间，例如：Wed, 05 Sep. 2012 23:00:00 GMT 类型：字符串 默认值：无
Host	访问Host值，格式为：`<bucketname>.regionid.example.com`。 类型：字符串 默认值：无

公共响应头（Common Response Headers）

OSS的RESTful接口中使用了一些公共响应头。这些响应头可以被所有的OSS请求所使用，其详细定义如下：

表 1-10: 公共响应头

名称	描述
Content-Length	*RFC2616*中定义的HTTP请求内容长度。 类型：字符串 默认值：无 使用场景：需要向OSS提交数据的请求
Connection	标明客户端和OSS服务器之间的链接状态。 类型：枚举 有效值：open、close 默认值：无
Date	HTTP 1.1协议中规定的GMT时间，例如：Wed, 05 Sep. 2012 23:00:00 GMT 类型：字符串 默认值：无
ETag	ETag (entity tag) 在每个Object生成的时候被创建，用于标示一个Object的内容。对于Put Object请求创建的Object，ETag值是其内容的MD5值；对

名称	描述
	于其他方式创建的Object，ETag值是其内容的UUID。ETag值可以用于检查Object内容是否发生变化。 类型：字符串 默认值：无
Server	生成Response的服务器。 类型：字符串 默认值：AliyunOSS
x-oss-request-id	x-oss-request-id是由Aliyun OSS创建，并唯一标识这个response的UUID。如果在使用OSS服务时遇到问题，可以凭借该字段联系OSS工作人员，快速定位问题。 类型：字符串 默认值：无

1.1.5 关于Service的操作

1.1.5.1 GetService

对于服务地址作Get请求可以返回请求者拥有的所有Bucket，其中“/”表示根目录。

请求语法

```
GET / HTTP/1.1
Host: oss.example.com
Date: GMT Date
Authorization: SignatureValue
```

请求参数

GetService(ListBucket)时，可以通过prefix，marker和max-keys对list做限定，返回部分结果。

表 1-11: 请求参数

名称	描述
prefix	限定返回的bucket name必须以prefix作为前缀，可以不设定，不设定时不过滤前缀信息 数据类型：字符串 默认值：无
marker	设定结果从marker之后按字母排序的第一个开始返回，可以不设定，不设定时从头开始返回数据 类型：字符串

名称	描述
	默认值：无
max-keys	限定此次返回bucket的最大数，如果不设定，默认为100，max-keys取值不能大于1000 数据类型：字符串 默认值：100

响应元素(Response Elements)

表 1-12: 响应元素

名称	描述
ListAllMyBucketsResult	保存Get Service请求结果的容器。 类型：容器 子节点：Owner, Buckets 父节点：None
Prefix	本次查询结果的前缀，当bucket未全部返回时才有此节点。 类型：字符串 父节点：ListAllMyBucketsResult
Marker	标明这次GetService(ListBucket)的起点，当bucket未全部返回时才有此节点。 类型：字符串 父节点：ListAllMyBucketsResult
MaxKeys	响应请求内返回结果的最大数目，当bucket未全部返回时才有此节点。 类型：字符串 父节点：ListAllMyBucketsResult
IsTruncated	指明是否所有的结果都已经返回：“true”表示本次没有返回全部结果；“false”表示本次已经返回了全部结果。当bucket未全部返回时才有此节点。 类型：枚举字符串 有效值：true、false 父节点：ListAllMyBucketsResult
NextMarker	表示下一次GetService(ListBucket)可以以此为marker，将未返回的结果返回。当bucket未全部返回时才有此节点。 类型：字符串 父节点：ListAllMyBucketsResult

名称	描述
Owner	用于存放Bucket拥有者信息的容器。 类型：容器 父节点：ListAllMyBucketsResult
ID	Bucket拥有者的用户ID。 类型：字符串 父节点：ListAllMyBucketsResult.Owner
DisplayName	Bucket拥有者的名称 (目前和ID一致)。 类型：字符串 父节点：ListAllMyBucketsResult.Owner
Buckets	保存多个Bucket信息的容器。 类型：容器 子节点：Bucket 父节点：ListAllMyBucketsResult
Bucket	保存bucket信息的容器。 类型：容器 子节点：Name, CreationDate, Location 父节点：ListAllMyBucketsResult.Buckets
Name	Bucket名称。 类型：字符串 父节点：ListAllMyBucketsResult.Buckets.Bucket
CreateDate	Bucket创建时间类型：时间 (格式：yyyy-mm-ddThh:mm:ss.timezone, e.g., 2011-12-01T12:27:13.000Z) 父节点：ListAllMyBucketsResult.Buckets.Bucket
Location	Bucket所在的数据中心。 类型：字符串 父节点：ListAllMyBucketsResult.Buckets.Bucket
ExtranetEndpoint	Bucket访问的外网域名。 类型：字符串 父节点：ListAllMyBucketsResult.Buckets.Bucket
IntranetEndpoint	同区域ECS访问Bucket的内网域名。 类型：字符串 父节点：ListAllMyBucketsResult.Buckets.Bucket

细节分析

1. GetService这个API只对验证通过的用户有效。

2. 如果请求中没有用户验证信息（即匿名访问），返回403 Forbidden。错误码：AccessDenied。
3. 当所有的bucket都返回时，返回的xml中不包含Prefix、Marker、MaxKeys、IsTruncated、NextMarker节点，如果还有部分结果未返回，则增加上述节点，其中NextMarker用于继续查询时给marker赋值。

示例

请求示例 I

```
GET / HTTP/1.1
Date: Thu, 15 May 2014 11:18:32 GMT
Host: regionid.example.com
Authorization: OSS n***i:C***I=
```

返回示例 I

```
HTTP/1.1 200 OK
Date: Thu, 15 May 2014 11:18:32 GMT
Content-Type: application/xml
Content-Length: 556
Connection: keep-alive
Server: AliyunOSS
x-oss-request-id: 5***4
<?xml version="1.0" encoding="UTF-8"?>
<ListAllMyBucketsResult>
 <Owner>
 <ID>51264</ID>
 <DisplayName>51264</DisplayName>
 </Owner>
 <Buckets>
 <Bucket>
 <CreationDate>2015-12-17T18:12:43.000Z</CreationDate>
 <ExtranetEndpoint>regionid.example.com</ExtranetEndpoint>
 <IntranetEndpoint>regionid-internal.example.com</IntranetEndpoint>
 <Location>oss-cn-shanghai</Location>
 <Name>app-base-oss</Name>
 </Bucket>
 <Bucket>
 <CreationDate>2014-12-25T11:21:04.000Z</CreationDate>
 <ExtranetEndpoint>regionid.example.com</ExtranetEndpoint>
 <IntranetEndpoint>regionid-internal.example.com</IntranetEndpoint>
 <Location>regionid</Location>
 <Name>a***3</Name>
 </Bucket>
 </Buckets>
</ListAllMyBucketsResult>
```

请求示例II

```
GET /?prefix=xz02tphky6fjfiuc&max-keys=1 HTTP/1.1
Date: Thu, 15 May 2014 11:18:32 GMT
Host:  regionid.example.com
```

```
Authorization: OSS n***i:C***I=
```

返回示例Ⅱ

```
HTTP/1.1 200 OK
Date: Thu, 15 May 2014 11:18:32 GMT
Content-Type: application/xml
Content-Length: 545
Connection: keep-alive
Server: AliyunOSS
x-oss-request-id: 5***5
<?xml version="1.0" encoding="UTF-8"?>
<ListAllMyBucketsResult>
 <Prefix>xz02tphky6fjfiuc</Prefix>
 <Marker></Marker>
 <MaxKeys>1</MaxKeys>
 <IsTruncated>true</IsTruncated>
 <NextMarker>xz02tphky6fjfiuc0</NextMarker>
 <Owner>
 <ID>ut_test_put_bucket</ID>
 <DisplayName>ut_test_put_bucket</DisplayName>
 </Owner>
 <Buckets>
 <Bucket>
 <CreationDate>2014-05-15T11:18:32.000Z</CreationDate>
 <ExtranetEndpoint>regionid.example.com</ExtranetEndpoint>
 <IntranetEndpoint>oss-cn-hangzhou-internal.example.com</IntranetEndpoint>
 <Location>oss-cn-hangzhou</Location>
 <Name>x***0</Name>
 </Bucket>
 </Buckets>
</ListAllMyBucketsResult>
```

1.1.6 关于Bucket的操作

1.1.6.1 PutBucket

PutBucket用于创建Bucket（不支持匿名访问）。创建的Bucket所在的Region和发送请求的Endpoint所对应的Region一致。Bucket所在的数据中心确定后，该Bucket下的所有Object将一直存放在对应的地区。

请求语法

```
PUT / HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
x-oss-acl: Permission
Authorization: SignatureValue
```

细节分析

1. 可以Put请求中的 `x-oss-acl` 头来设置Bucket访问权限。目前Bucket有三种访问权限：public-read-write，public-read和private。

2. 如果请求的Bucket已经存在，并且请求者是所有者，返回200 OK成功。
3. 如果请求的Bucket已经存在，但是不是请求者所拥有的，返回409 Conflict。错误码：BucketAlreadyExists。
4. 如果想创建的Bucket不符合命名规范，返回400 Bad Request消息。错误码：InvalidBucketName。
5. 如果用户发起PUT Bucket请求的时候，没有传入用户验证信息，返回403 Forbidden消息。错误码：AccessDenied。
6. 如果PutBucket的时候发现已经超过bucket最大创建数时，**默认10个**，返回400 Bad Request消息。错误码：TooManyBuckets。
7. 创建的Bucket，如果没有指定访问权限，则默认使用 Private 权限。

示例

请求示例：

```
PUT / HTTP/1.1
Host: BucketName.regionid.example.com
Date: Fri, 24 Feb 2012 03:15:40 GMT
x-oss-acl: private
Authorization: OSS q***c:7***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 03:15:40 GMT
Location: /***
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.6.2 PutBucketACL

Put Bucket ACL接口用于修改Bucket访问权限。目前Bucket有三种访问权限：public-read-write，public-read和private。Put Bucket ACL操作通过Put请求中的“x-oss-acl”头来设置。这个操作只有该Bucket的创建者有权限执行。如果操作成功，则返回200；否则返回相应的错误码和提示信息。

请求语法

```
PUT /?acl HTTP/1.1
x-oss-acl: Permission
Host: BucketName.regionid.example.com
Date: GMT Date
```

```
Authorization: SignatureValue
```

细节分析

1. 如果bucket存在，发送时带的权限和已有权限不一样，并且请求发送者是bucket拥有者时。该请求不会改变bucket内容，但是会更新权限。
2. 如果用户发起Put Bucket请求的时候，没有传入用户验证信息，返回403 Forbidden消息。错误码：AccessDenied。
3. 如果请求中没有，“x-oss-acl”头，并且该bucket已存在，并属于该请求发起者，则维持原bucket权限不变。

示例

请求示例：

```
PUT /?acl HTTP/1.1
x-oss-acl: public-read
Host: BucketName.regionid.example.com
Date: Fri, 24 Feb 2012 03:21:12 GMT
Authorization: OSS q***c:K***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 03:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

如果该设置的权限不存在，示例400 Bad Request消息：

错误返回示例：

```
HTTP/1.1 400 Bad Request
x-oss-request-id: 5***9
Date: Fri, 24 Feb 2012 03:55:00 GMT
Content-Length: 309
Content-Type: text/xml; charset=UTF-8
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<Error>
  <Code>InvalidArgument</Code>
  <Message>no such bucket access control exists</Message>
  <RequestId>5***9</RequestId>
  <HostId>***-test.example.com</HostId>
  <ArgumentName>x-oss-acl</ArgumentName>
  <ArgumentValue>error-acl</ArgumentValue>
```

```
</Error>
```

1.1.6.3 PutBucketLogging

OSS提供Bucket访问日志的目的是为了方便bucket的拥有者理解和分析bucket的访问行为。OSS提供的Bucket访问日志不保证记录下每一条访问记录。

Bucket的拥有者可以为bucket开启访问日志记录功能。这个功能开启后，OSS将自动记录访问这个bucket请求的详细信息，并按照用户指定的规则，以小时为单位，将访问日志作为一个Object写入用户指定的bucket。OSS提供Bucket访问日志的目的是为了方便bucket的拥有者理解和分析bucket的访问行为。OSS提供的Bucket访问日志不保证记录下每一条访问记录。

请求语法

```
PUT /?logging HTTP/1.1
Date: GMT Date
Content-Length：ContentLength
Content-Type: application/xml
Authorization: SignatureValue
Host: BucketName.regionid.example.com
<?xml version="1.0" encoding="UTF-8"?>
<BucketLoggingStatus>
 <LoggingEnabled>
 <TargetBucket>TargetBucket</TargetBucket>
 <TargetPrefix>TargetPrefix</TargetPrefix>
 </LoggingEnabled>
</BucketLoggingStatus>
```

请求元素(Request Elements)

表 1-13: 请求元素

名称	描述	是否必需
BucketLoggingStatus	访问日志状态信息的容器类型：容器子元素：LoggingEnabled父元素：无	是
LoggingEnabled	访问日志信息的容器。这个元素在开启时需要，关闭时不需要。类型：容器子元素：TargetBucket, TargetPrefix父元素：BucketLoggingStatus	否
TargetBucket	指定存放访问日志的Bucket。类型：字符子元素：无父元素：BucketLoggingStatus.LoggingEnabled	在开启访问日志的时候必需
TargetPrefix	指定最终被保存的访问日志文件前缀。类型：字符子元素：None父元素：BucketLoggingStatus.LoggingEnabled	否

存储访问日志记录的object命名规则

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

命名规则中，TargetPrefix由用户指定；YYYY, mm, DD, HH, MM和SS分别是该Object被创建时的阿拉伯数字的年，月，日，小时，分钟和秒（注意位数）；UniqueString为OSS系统生成的字符串。一个实际的用于存储OSS访问日志的Object名称例子如下：

```
MyLog-oss-example-2012-09-10-04-00-00-0000
```

上例中，“MyLog-”是用户指定的Object前缀；“oss-example”是源bucket的名称；“2012-09-10-04-00-00”是该Object被创建时的北京时间；“0000” 是OSS系统生成的字符串。

LOG文件格式

表 1-14: LOG文件格式

名称	例子	含义
Remote IP	119.140.142.11	请求发起的IP地址（Proxy代理或用户防火墙可能会屏蔽该字段）
Reserved	-	保留字段
Reserved	-	保留字段
Time	[02/May/2012:00:00:04 +0800]	OSS收到请求的时间
Request-URI	“GET /aliyun-logo.png HTTP/1.1 ”	用户请求的URI(包括query-string)
HTTP Status	200	OSS返回的HTTP状态码
SentBytes	5576	用户从OSS下载的流量
RequestTime (ms)	71	完成本次请求的时间（毫秒）
Referer	http://www.aliyun.com/product/oss	请求的HTTP Referer
User-Agent	curl/7.15.5	HTTP的User-Agent头
HostName	oss-example.regionid.example.com	请求访问域名
Request ID	505B01695037C2AF032593A4	用于唯一标示该请求的UUID
LoggingFlag	true	是否开启了访问日志功能

名称	例子	含义
Requester Aliyun ID	1657136103983691	请求者的阿里云ID；匿名访问为“-”
Operation	GetObject	请求类型
Bucket	oss-example	请求访问的Bucket名字
Key	/aliyun-logo.png	用户请求的Key
ObjectSize	5576	Object大小
Server Cost Time (ms)	17	OSS服务器处理本次请求所花的时间（毫秒）
Error Code	NoSuchBucket	OSS返回的错误码
Request Length	302	用户请求的长度（Byte）
UserID	1657136103983691	Bucket拥有者ID
Delta DataSize	280	Bucket大小的变化量；若没有变化为“-”
Sync Request	-	是否是CDN回源请求；若不是为“-”
Reserved	-	保留字段

细节分析

1. 源Bucket和目标Bucket必须属于同一个用户。
2. 上面所示的请求语法中，“BucketName”表示要开启访问日志记录的bucket；“TargetBucket”表示访问日志记录要存入的bucket；“TargetPrefix”表示存储访问日志记录的object名字前缀，可以为空。
3. 源bucket和目标bucket可以是同一个Bucket，也可以是不同的Bucket；用户也可以将多个的源bucket的LOG都保存在同一个目标bucket内（建议指定不同的TargetPrefix）。
4. 当关闭一个Bucket的访问日志记录功能时，只要发送一个空的BucketLoggingStatus即可，具体方法可以参考下面的请求示例。
5. 所有PUT Bucket Logging请求必须带签名，即不支持匿名访问。
6. 如果PUT Bucket Logging请求发起者不是源bucket（请求示例中的BucketName）的拥有者，OSS返回403错误码；
7. 如果源bucket不存在，OSS返回错误码：NoSuchBucket。

8. 如果PUT Bucket Logging请求发起者不是目标bucket（请求示例中的TargetBucket）的拥有者，OSS返回403；如果目标bucket不存在，OSS返回错误码：InvalidTargetBucketForLogging。
9. 源Bucket和目标Bucket必须属于同一个数据中心，否则返回400错误，错误码为：InvalidTargetBucketForLogging。
10. PUT Bucket Logging请求中的XML不合法，返回错误码：MalformedXML。
11. 源bucket和目标bucket可以是同一个Bucket；用户也可以将不同的源bucket的LOG都保存在同一个目标bucket内（注意要指定不同的TargetPrefix）。
12. 源Bucket被删除时，对应的Logging规则也将被删除。
13. OSS以小时为单位生成bucket访问的Log文件，但并不表示这个小时的所有请求都记录在这个小时的LOG文件内，也有可能出现在上一个或者下一个LOG文件中。
14. OSS生成的Log文件命名规则中的“UniqueString”仅仅是OSS为其生成的UUID，用于唯一标识该文件。
15. OSS生成一个bucket访问的Log文件，算作一次PUT操作，并记录其占用的空间，但不会记录产生的流量。LOG生成后，用户可以按照普通的Object来操作这些LOG文件。
16. OSS会忽略掉所有以“x-”开头的query-string参数，但这个query-string会被记录在访问LOG中。如果你想从海量的访问日志中，标示一个特殊的请求，可以在URL中添加一个“x-”开头的query-string参数。如：

    http://oss-example.regionid.example.com/aliyun-logo.png

    http://oss-example.regionid.example.com/aliyun-logo.png?x-user=admin
    OSS处理上面两个请求，结果是一样的。但是在访问LOG中，你可以通过搜索“x-user=admin”，很方便地定位出经过标记的这个请求。
17. OSS的LOG中的任何一个字段，都可能出现“-”，用于表示未知数据或对于当前请求该字段无效。
18. 根据需求，OSS的LOG格式将来会在尾部添加一些字段，请开发者开发Log处理工具时考虑兼容性的问题。
19. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。

示例

开启bucket访问日志的请求示例：

```
PUT /?logging HTTP/1.1
Host: ***.regionid.example.com
Content-Length: 186
```

```
Date: Fri, 04 May 2012 03:21:12 GMT
Authorization: OSS q***c:K***=
<?xml version="1.0" encoding="UTF-8"?>
<BucketLoggingStatus>
<LoggingEnabled>
<TargetBucket>doc-log</TargetBucket>
<TargetPrefix>MyLog-</TargetPrefix>
</LoggingEnabled>
</BucketLoggingStatus>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 04 May 2012 03:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

关闭bucket访问日志的请求示例：

```
PUT /?logging HTTP/1.1
Host: ***.regionid.example.com
Content-Type: application/xml
Content-Length: 86
Date: Fri, 04 May 2012 04:21:12 GMT
Authorization: OSS q***c:K***=
<?xml version="1.0" encoding="UTF-8"?>
<BucketLoggingStatus>
</BucketLoggingStatus>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 04 May 2012 04:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.6.4 PutBucketWebsite

Put Bucket Website操作可以将一个bucket设置成静态网站托管模式。

请求语法

```
PUT /?website HTTP/1.1
Date: GMT Date
Content-Length：ContentLength
Content-Type: application/xml
Host: BucketName.regionid.example.com
Authorization: SignatureValue

<?xml version="1.0" encoding="UTF-8"?>
<WebsiteConfiguration>
   <IndexDocument>
      <Suffix>index.html</Suffix>
```

```
    </IndexDocument>
    <ErrorDocument>
        <Key>errorDocument.html</Key>
    </ErrorDocument>
</WebsiteConfiguration>
```

请求元素（Request Elements）

表 1-15: 请求元素

名称	描述	是否必须
ErrorDocument	子元素Key的父元素 类型：容器 父元素：WebsiteConfiguration	否
IndexDocument	子元素Suffix的父元素。 类型：容器 父元素：WebsiteConfiguration	是
Key	返回404错误时使用的文件名 类型：字符串 父元素：WebsiteConfiguration.ErrorDocument 有条件：当ErrorDocument设置时，必需	有条件
Suffix	返回目录URL时添加的索引文件名，不要为空，也不要包含"/"。例如索引文件设置为index.html，则访问：regionid.example.com/mybucket/mydir/这样请求的时候默认都相当于访问regionid.example.com/mybucket/index.html 类型：字符串 父元素：WebsiteConfiguration.IndexDocument	是
WebsiteConfiguration	请求的容器 类型：容器 父元素：无	是

细节分析

1. 所谓静态网站是指所有的网页都由静态内容构成，包括客户端执行的脚本，例如JavaScript；OSS不支持涉及到需要服务器端处理的内容，例如PHP，JSP，APS.NET等。
2. 如果你想使用自己的域名来访问基于bucket的静态网站，可以通过域名CNAME来实现。具体配置方法见3.4节：自定义域名绑定。
3. 用户将一个bucket设置成静态网站托管模式时，必须指定索引页面，错误页面则是可选的。

4. 用户将一个bucket设置成静态网站托管模式时，指定的索引页面和错误页面是该bucket内的一个object。
5. 在将一个bucket设置成静态网站托管模式后，对静态网站根域名的匿名访问，OSS将返回索引页面；对静态网站根域名的签名访问，OSS将返回Get Bucket结果。
6. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。

示例

请求示例：

```
PUT /?website HTTP/1.1
Host: oss-example.regionid.example.com
Content-Length: 209
Date: Fri, 04 May 2012 03:21:12 GMT
Authorization: OSS q***c:K***=

<?xml version="1.0" encoding="UTF-8"?>
<WebsiteConfiguration>
<IndexDocument>
<Suffix>index.html</Suffix>
</IndexDocument>
<ErrorDocument>
<Key>error.html</Key>
</ErrorDocument>
</WebsiteConfiguration>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 04 May 2012 03:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.6.5 PutBucketReferer

Put Bucket Referer操作可以设置一个bucket的referer访问白名单和是否允许referer字段为空的请求访问。Bucket Referer防盗链具体见*OSS防盗链*。

请求语法

```
PUT /?referer HTTP/1.1
Date: GMT Date
Content-Length：ContentLength
Content-Type: application/xml
Host: BucketName.oss.example.com
Authorization: SignatureValue

<?xml version="1.0" encoding="UTF-8"?>
```

```
<RefererConfiguration>
<AllowEmptyReferer>true</AllowEmptyReferer >
   <RefererList>
      <Referer> http://www.aliyun.com</Referer>
      <Referer> https://www.aliyun.com</Referer>
      <Referer> http://www.*.com</Referer>
      <Referer> https://www.?.aliyuncs.com</Referer>
   </RefererList>
</RefererConfiguration>
```

请求元素（Request Elements）

表 1-16: 请求元素

名称	描述	是否必需
RefererConfiguration	保存Referer配置内容的容器类型：容器子节点：AllowEmptyReferer节点、RefererList节点父节点：无	是
AllowEmptyReferer	指定是否允许referer字段为空的请求访问。类型：枚举字符串有效值：true或false 默认值：true 父节点：RefererConfiguration	是
RefererList	保存referer访问白名单的容器。类型：容器父节点：RefererConfiguration 子节点：Referer	是
Referer	指定一条referer访问白名单。类型：字符串父节点：RefererList	可选

细节分析

1. 只有Bucket的拥有者才能发起Put Bucket Referer请求，否则返回403 Forbidden消息。错误码：AccessDenied。
2. AllowEmptyReferer中指定的配置将替换之前的AllowEmptyReferer配置，该字段为必填项，系统中默认的AllowEmptyReferer配置为true。
3. 此操作将用RefererList中的白名单列表覆盖之前配置的白名单列表，当用户上传的RefererList为空时（不包含Referer请求元素），此操作会覆盖已配置的白名单列表，即删除之前配置的RefererList。
4. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。

示例

不包含Referer的请求示例：

```
PUT /?referer HTTP/1.1
Host: BucketName.oss.example.com
Content-Length: 247
Date: Fri, 04 May 2012 03:21:12 GMT
```

```
Authorization: OSS q***c:K***=

<?xml version="1.0" encoding="UTF-8"?>
<RefererConfiguration>
<AllowEmptyReferer>true</AllowEmptyReferer >
< RefererList />
</RefererConfiguration>
```

包含Referer的请求示例：

```
PUT /?referer HTTP/1.1
Host: BucketName.oss.example.com
Content-Length: 247
Date: Fri, 04 May 2012 03:21:12 GMT
Authorization: OSS q***c:K***=

<?xml version="1.0" encoding="UTF-8"?>
<RefererConfiguration>
<AllowEmptyReferer>true</AllowEmptyReferer >
< RefererList>
<Referer> http://www.aliyun.com</Referer>
<Referer> https://www.aliyun.com</Referer>
<Referer> http://www.*.com</Referer>
<Referer> https://www.?.aliyuncs.com</Referer>
</ RefererList>
</RefererConfiguration>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 04 May 2012 03:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.6.6 PutBucketLifecycle

Bucket的拥有者可以通过Put Bucket Lifecycle来设置Bucket的Lifecycle配置。Lifecycle开启后，OSS将按照配置，定期自动删除与Lifecycle规则相匹配的Object。

请求语法

```
PUT /?lifecycle HTTP/1.1
Date: GMT Date
Content-Length：ContentLength
Content-Type: application/xml
Authorization: SignatureValue
Host: BucketName.oss.example.com
<?xml version="1.0" encoding="UTF-8"?>
<LifecycleConfiguration>
 <Rule>
 <ID>RuleID</ID>
 <Prefix>Prefix</Prefix>
 <Status>Status</Status>
 <Expiration>
 <Days>Days</Days>
```

```
 </Expiration>
 <AbortMultipartUpload>
 <Days>Days</Days> </AbortMultipartUpload>
 </Rule>
</LifecycleConfiguration>
```

请求元素(Request Elements)

表 1-17: 请求元素

名称	描述	是否必需
CreatedBeforeDate	指定规则何时之前生效。日期必需服从ISO8601的格式，并且总是UTC的零点。例如：2002-10-11T00:00:00.000Z 类型：字符串 父节点：Expiration或者AbortMultipartUpload	Days和CreatedBeforeDate二选一
Days	指定规则在对象最后修改时间过后多少天生效。 类型：正整数 父节点：Expiration	Days和CreatedBeforeDate二选一
Expiration	指定Object规则的过期属性。 类型：容器 子节点：Days或CreatedBeforeDate 父节点：Rule	否
AbortMultipartUpload	指定未完成的Part规则的过期属性。 类型：容器 子节点：Days或CreatedBeforeDate 父节点：Rule	否
ID	规则唯一的ID。最多由255字节组成。当用户没有指定，或者该值为空时，OSS会为用户生成一个唯一值。 类型：字符串 子节点：无 父节点：Rule	否
LifecycleConfiguration	Lifecycle配置的容器，最多可容纳1000条规则。 类型：容器 子节点：Rule 父节点：无	是
Prefix	指定规则所适用的前缀。只有匹配前缀的对象才可能被该规则所影响。不可重叠。 类型：字符串	是

名称	描述	是否必需
	子节点：无 父节点：Rule	
Rule	表述一条规则 类型：容器 子节点：ID，Prefix，Status，Expiration 父节点：LifecycleConfiguration	是
Status	如果其值为Enabled，那么OSS会定期执行该规则；如果是Disabled，那么OSS会忽略该规则。 类型：字符串 父节点：Rule 有效值：Enabled，Disabled	是

细节分析

1. 只有Bucket的拥有者才能发起Put Bucket Lifecycle请求，否则返回403 Forbidden消息。错误码：AccessDenied。
2. 如果此前没有设置过Lifecycle，此操作会创建一个新的Lifecycle配置；否则，就覆写先前的配置。
3. 可以对Object设置过期时间，也可以对Part设置过期时间。这里的Part指的是以分片上传方式上传，但最后未提交的分片。

示例

请求示例：

```
PUT /?lifecycle HTTP/1.1
Host: BucketName.oss.example.com
Content-Length: 443
Date: Mon, 14 Apr 2014 01:08:38 GMT
Authorization: OSS q***c:K***=
<?xml version="1.0" encoding="UTF-8"?>
</LifecycleConfiguration>
 <Rule>
 <ID>delete objects and parts after one day</ID>
 <Prefix>logs/</Prefix>
 <Status>Enabled</Status>
 <Expiration>
 <Days>1</Days>
 </Expiration>
 <AbortMultipartUpload>
 <Days>1</Days>
 </AbortMultipartUpload>
 </Rule>
 <Rule>
 <ID>delete created before date</ID>
```

```
 <Prefix>backup/</Prefix>
 <Status>Enabled</Status>
 <Expiration>
 <CreatedBeforeDate>2014-10-11T00:00:00.000Z</CreatedBeforeDate>
 </Expiration>
 <AbortMultipartUpload>
 <CreatedBeforeDate>2014-10-11T00:00:00.000Z</CreatedBeforeDate>
 </AbortMultipartUpload>
 </Rule>
</LifecycleConfiguration>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Mon, 14 Apr 2014 01:17:10 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.6.7 PutBucketTagging

Put Bucket Tagging接口可以为Bucket添加一组标签。用户可以通过标签，对Bucket进行分类管理，如按标签查看账单信息、ListBucket时只显示带有指定标签的Bucket等。

请求语法

```
PUT /?tagging HTTP/1.1
Date: GMT Date
Content-Length：ContentLength
Content-Type: application/xml
Authorization: SignatureValue
Host: BucketName.regionid.example.com
<?xml version="1.0" encoding="UTF-8"?>
<Tagging>
 <TagSet>
 <Tag>
 <Key>Tag Name</Key>
 <Value>Tag Value</Value>
 </Tag>
 </TagSet>
</Tagging>
```

请求元素（Request Elements）

表 1-18: 请求元素

名称	描述	是否必需
Tagging	Bucket Tag的容器。 类型：容器 子节点：TagSet 父节点：无	是

名称	描述	是否必需
TagSet	保存一组Tag的容器，每个Bucket最多允许有10个标签。 类型：容器 子节点：Tag 父节点：Tagging	是
Tag	单条Tag的的容器。 类型：容器 子节点：Key或Value 父节点：TagSet	是
Key	标签的名称，对于同一个Bucket，标签名称唯一，大小写敏感，不可为空。 类型：字符串 子节点：无 父节点：Tag	是
Value	标签的值，可不唯一，大小写敏感，可为空。 类型：字符串 子节点：无 父节点：Tag	是

细节分析

1. 一个Bucket最多允许创建10个标签。如果用户创建的标签超过了限制，OSS返回Http 400 Bad Request消息，错误码：TooManyTags。
2. Bucket Tag的Key和Value允许的字符为除去ASCII码中十进制值小于32或等于127之外的所有UTF-8字符，大小写敏感。Tag Key允许最大长度为64，Tag Value允许最大长度为128。如果Bucket Tag的Key或Value不符合这些规则，OSS返回Http 400 Bad Request消息，错误码：InvalidArgument。
3. 对于同一个Bucket，创建的两个Tag Key不能重复，否则OSS会返回Http 400 Bad Request消息，错误码：RepeatedTags。
4. aliyun是OSS保留的标签前缀，Tag Key不能以aliyun这个前缀开头（大小写均不可），否则OSS返回Http 400 Bad Request消息，错误码：InvalidArgument。
5. Put Bucket Tagging是覆盖语义，即新Put的标签会完全覆盖已有的标签。

示例

请求示例

```
PUT /?tagging HTTP/1.1
```

```
Host: ***.regionid.example.com
Content-Type: application/xml
Content-Length: 86
Date: Thu, 23 Jun 2015 15:39:12 GMT
Authorization: OSS q***c:K***=
<Tagging>
 <TagSet>
 <Tag>
 <Key>Project</Key>
<Value>Project One</Value>
 </Tag>
 <Tag>
 <Key>User</Key>
 <Value>jsmith</Value>
 </Tag>
 </TagSet>
</Tagging>
```

返回示例

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Thu, 23 Jun 2015 15:39:12 GMT
Content-Length: 0
Connection: close
Server: AliyunOSS
```

1.1.6.8 GetBucket

Get Bucket操作可用来list Bucket中所有Object的信息。

请求语法

```
GET / HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

请求参数(Request Parameters)

GetBucket（ListObject）时，可以通过prefix，marker，delimiter和max-keys对list做限定，返回部分结果。另外，可以通过encoding-type对返回结果中的Delimiter、Marker、Prefix、NextMarker和Key这些元素进行编码。

表 1-19: 请求参数

名称	描述
delimiter	是一个用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素——CommonPrefixes。 数据类型：字符串 默认值：无

名称	描述
marker	设定结果从marker之后按字母排序的第一个开始返回。 数据类型：字符串 默认值：无
max-keys	限定此次返回object的最大数，如果不设定，默认为100，max-keys取值不能大于1000。 数据类型：字符串 默认值：100
prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix。 数据类型：字符串 默认值：无
encoding-type	指定对返回的内容进行编码，指定编码的类型。Delimiter、Marker、Prefix、NextMarker和Key使用UTF-8字符，但xml 1.0标准不支持解析一些控制字符，比如ascii值从0到10的字符。对于包含xml 1.0标准不支持的控制字符，可以通过指定encoding-type对返回的Delimiter、Marker、Prefix、NextMarker和Key进行编码。 数据类型：字符串 默认值：无 可选值：url

响应元素(Response Elements)

表 1-20: 响应元素

名称	描述
Contents	保存每个返回Object meta的容器。 类型：容器 父节点：ListBucketResult
CommonPrefixes	如果请求中指定了delimiter参数，则在OSS返回的响应中包含CommonPrefixes元素。该元素标明那些以delimiter结尾，并有共同前缀的object名称的集合。 类型：字符串 父节点：ListBucketResult
Delimiter	是一个用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素——CommonPrefixes。 类型：字符串

名称	描述
	父节点：ListBucketResult
EncodingType	指明返回结果中编码使用的类型。如果请求的参数中指定了encoding-type，那会对返回结果中的Delimiter、Marker、Prefix、NextMarker和Key这些元素进行编码。 类型：字符串 父节点：ListBucketResult
DisplayName	Object 拥有者的名字。 类型：字符串 父节点：ListBucketResult.Contents.Owner
ETag	ETag (entity tag) 在每个Object生成的时候被创建，用于标示一个Object的内容。对于Put Object请求创建的Object，ETag值是其内容的MD5值；对于其他方式创建的Object，ETag值是其内容的UUID。ETag值可以用于检查Object内容是否发生变化。不建议用户使用ETag来作为Object内容的MD5校验数据完整性。 类型：字符串 父节点：ListBucketResult.Contents
ID	Bucket拥有者的用户ID。 类型：字符串 父节点：ListBucketResult.Contents.Owner
IsTruncated	指明是否所有的结果都已经返回；“true”表示本次没有返回全部结果；“false”表示本次已经返回了全部结果。 类型：枚举字符串 有效值：true、false 父节点：ListBucketResult
Key	Object的Key. 类型：字符串 父节点：ListBucketResult.Contents
LastModified	Object最后被修改的时间。 类型：时间 父节点：ListBucketResult.Contents
ListBucketResult	保存Get Bucket请求结果的容器。 类型：容器 子节点：Name, Prefix, Marker, MaxKeys, Delimiter, IsTruncated, Nextmarker, Contents 父节点：None

名称	描述
Marker	标明这次Get Bucket（List Object）的起点。 类型：字符串 父节点：ListBucketResult
MaxKeys	响应请求内返回结果的最大数目。 类型：字符串 父节点：ListBucketResult
Name	Bucket名字 类型：字符串 父节点：ListBucketResult
Owner	保存Bucket拥有者信息的容器。 类型：容器 子节点：DisplayName, ID 父节点：ListBucketResult
Prefix	本次查询结果的开始前缀。 类型：字符串 父节点：ListBucketResult
Size	Object的字节数。 类型：字符串 父节点：ListBucketResult.Contents
StorageClass	Object的存储类型，目前只能是“Standard”类。 型：字符串 父节点：ListBucketResult.Contents

细节分析

1. Object中用户自定义的meta，在GetBucket请求时不会返回。
2. 如果访问的Bucket不存在，包括试图访问因为命名不规范无法创建的Bucket，返回404 Not Found错误，错误码：NoSuchBucket。
3. 如果没有访问该Bucket的权限，返回403 Forbidden错误，错误码：AccessDenied。
4. 如果因为max-keys的设定无法一次完成listing，返回结果会附加一个`<NextMarker>`，提示继续listing可以以此为marker。NextMarker中的值仍在list结果之中。
5. 在做条件查询时，即使marker实际在列表中不存在，返回也从符合marker字母排序的下一个开始打印。如果max-keys小于0或者大于1000，将返回400 Bad Request错误。错误码：InvalidArgument。

6. 若prefix，marker，delimiter参数不符合长度要求，返回400 Bad Request。错误码：InvalidArgument。
7. prefix，marker用来实现分页显示效果，参数的长度必须小于1024字节。
8. 如果把prefix设为某个文件夹名，就可以罗列以此prefix开头的文件，即该文件夹下递归的所有的文件和子文件夹。如果再把delimiter设置为 / 时，返回值就只罗列该文件夹下的文件，该文件夹下的子文件名返回在CommonPrefixes部分，子文件夹下递归的文件和文件夹不被显示。如一个bucket存在三个object : fun/test.jpg， fun/movie/001.avi， fun/movie/007.avi。若设定prefix为”fun/” ，则返回三个object；如果增加设定delimiter为“/” ，则返回文件”fun/test.jpg”和前缀”fun/movie/” ；即实现了文件夹的逻辑。

举例场景

在bucket“my_oss”内有4个object，名字分别为：

- oss.jpg
- fun/test.jpg
- fun/movie/001.avi
- fun/movie/007.avi

示例

请求示例：

```
GET / HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 08:43:27 GMT
Authorization: OSS q***c:B***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 08:43:27 GMT
Content-Type: application/xml
Content-Length: 1866
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
<Name>oss-example</Name>
<Prefix></Prefix>
<Marker></Marker>
<MaxKeys>100</MaxKeys>
<Delimiter></Delimiter>
   <IsTruncated>false</IsTruncated>
   <Contents>
```

```
        <Key>fun/movie/001.avi</Key>
        <LastModified>2012-02-24T08:43:07.000Z</LastModified>
        <ETag>&quot;5***E&quot;</ETag>
        <Type>Normal</Type>
        <Size>344606</Size>
        <StorageClass>Standard</StorageClass>
        <Owner>
            <ID>00220120222</ID>
            <DisplayName>user-example</DisplayName>
        </Owner>
    </Contents>
    <Contents>
        <Key>fun/movie/007.avi</Key>
        <LastModified>2012-02-24T08:43:27.000Z</LastModified>
        <ETag>&quot;5***E&quot;</ETag>
        <Type>Normal</Type>
        <Size>344606</Size>
        <StorageClass>Standard</StorageClass>
        <Owner>
            <ID>00220120222</ID>
            <DisplayName>user-example</DisplayName>
        </Owner>
    </Contents>
<Contents>
        <Key>fun/test.jpg</Key>
        <LastModified>2012-02-24T08:42:32.000Z</LastModified>
        <ETag>&quot;5***E&quot;</ETag>
        <Type>Normal</Type>
        <Size>344606</Size>
        <StorageClass>Standard</StorageClass>
        <Owner>
            <ID>0***2</ID>
            <DisplayName>***</DisplayName>
        </Owner>
    </Contents>
    <Contents>
        <Key>oss.jpg</Key>
        <LastModified>2012-02-24T06:07:48.000Z</LastModified>
        <ETag>&quot;5***E&quot;</ETag>
        <Type>Normal</Type>
        <Size>344606</Size>
        <StorageClass>Standard</StorageClass>
        <Owner>
            <ID>0***2</ID>
            <DisplayName>***</DisplayName>
        </Owner>
    </Contents>
</ListBucketResult>
```

请求示例(含Prefix参数)：

```
GET /?prefix=fun HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 08:43:27 GMT
Authorization: OSS q***c:B***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
```

```
Date: Fri, 24 Feb 2012 08:43:27 GMT
Content-Type: application/xml
Content-Length: 1464
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
<Name>***</Name>
<Prefix>fun</Prefix>
<Marker></Marker>
<MaxKeys>100</MaxKeys>
<Delimiter></Delimiter>
   <IsTruncated>false</IsTruncated>
   <Contents>
      <Key>fun/movie/001.avi</Key>
      <LastModified>2012-02-24T08:43:07.000Z</LastModified>
      <ETag>&quot;5***E&quot;</ETag>
      <Type>Normal</Type>
      <Size>344606</Size>
      <StorageClass>Standard</StorageClass>
      <Owner>
         <ID>0***2</ID>
         <DisplayName>***</DisplayName>
      </Owner>
   </Contents>
   <Contents>
      <Key>fun/movie/007.avi</Key>
      <LastModified>2012-02-24T08:43:27.000Z</LastModified>
      <ETag>&quot;5***E&quot;</ETag>
      <Type>Normal</Type>
      <Size>344606</Size>
      <StorageClass>Standard</StorageClass>
      <Owner>
         <ID>0***2</ID>
         <DisplayName>***</DisplayName>
      </Owner>
   </Contents>
   <Contents>
      <Key>fun/test.jpg</Key>
      <LastModified>2012-02-24T08:42:32.000Z</LastModified>
      <ETag>&quot;5***E&quot;</ETag>
      <Type>Normal</Type>
      <Size>344606</Size>
      <StorageClass>Standard</StorageClass>
      <Owner>
         <ID>0***2</ID>
         <DisplayName>***</DisplayName>
      </Owner>
   </Contents>
</ListBucketResult>
```

请求示例(含prefix和delimiter参数)：

```
GET /?prefix=fun/&delimiter=/ HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 08:43:27 GMT
```

```
Authorization: OSS q***c:D***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 08:43:27 GMT
Content-Type: application/xml
Content-Length: 712
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
<Name>oss-example</Name>
<Prefix>fun/</Prefix>
<Marker></Marker>
<MaxKeys>100</MaxKeys>
<Delimiter>/</Delimiter>
    <IsTruncated>false</IsTruncated>
    <Contents>
        <Key>fun/test.jpg</Key>
        <LastModified>2012-02-24T08:42:32.000Z</LastModified>
        <ETag>&quot;5***E&quot;</ETag>
        <Type>Normal</Type>
        <Size>344606</Size>
        <StorageClass>Standard</StorageClass>
        <Owner>
            <ID>0***2</ID>
            <DisplayName>***</DisplayName>
        </Owner>
    </Contents>
   <CommonPrefixes>
        <Prefix>fun/movie/</Prefix>
   </CommonPrefixes>
</ListBucketResult>
```

1.1.6.9 GetBucketAcl

Get Bucket ACL用来获取某个Bucket的访问权限。

请求语法

```
GET /?acl HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
```

```
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-21: 响应元素

名称	描述
AccessControlList	存储ACL信息的容器类型：容器父节点：AccessControlPolicy
AccessControlPolicy	保存Get Bucket ACL结果的容器类型：容器父节点：None
DisplayName	Bucket拥有者的名称。(目前和ID一致)类型：字符串父节点：AccessControlPolicy.Owner
Grant	Bucket的ACL权限。类型：枚举字符串有效值：private、public-read、public-read-write父节点：AccessControlPolicy.AccessControlList
ID	Bucket拥有者的用户ID类型：字符串父节点：AccessControlPolicy.Owner
Owner	保存Bucket拥有者信息的容器。类型：容器父节点：AccessControlPolicy

细节分析

只有Bucket的拥有者才能使用Get Bucket ACL这个接口。

示例

请求示例：

```
GET /?acl HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 04:11:23 GMT
Authorization: OSS q***c:C***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 04:11:23 GMT
Content-Length: 253
Content-Tupe: application/xml
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" ?>
<AccessControlPolicy>
    <Owner>
        <ID>0***2</ID>
        <DisplayName>***</DisplayName>
    </Owner>
```

```
    <AccessControlList>
        <Grant>public-read</Grant>
    </AccessControlList>
</AccessControlPolicy>
```

1.1.6.10 GetBucketLocation

Get Bucket Location用于查看Bucket所属的数据中心位置信息。

请求语法

```
GET /?location HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-22: 响应元素

名称	描述
LocationConstraint	Bucket所在的区域 类型：字符串 有效值： oss-cn-hangzhou、oss-cn-qingdao、oss-cn-beijing、oss-cn-hongkong、oss-cn-shenzhen、oss-cn-shanghai

细节分析

1. 只有Bucket的拥有者才能查看Bucket的Location信息，否则返回403 Forbidden错误，错误码：AccessDenied。
2. 目前LocationConstraint有效值：oss-cn-hangzhou，oss-cn-qingdao，oss-cn-beijing，oss-cn-hongkong，oss-cn-shenzhen，oss-cn-shanghai，oss-us-west-1，oss-us-east-1，oss-ap-southeast-1；分别对应杭州数据中心，青岛数据中心，北京数据中心、香港数据中心、深圳数据中心、上海数据中心、美国硅谷数据中心、美国弗吉尼亚数据中心和亚太（新加坡）数据中心。

示例

请求示例：

```
Get /?location HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 04 May 2012 05:31:04 GMT
```

```
Authorization: OSS q***c:c***=
```

已设置LOG规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Fri, 15 Mar 2013 05:31:04 GMT
Connection: keep-alive
Content-Length: 90
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<LocationConstraint xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >oss-cn-hangzhou</
LocationConstraint >
```

1.1.6.11 GetBucketInfo

Get Bucket Info操作用于查看bucket的相关信息。包括如下内容：

- 创建时间
- 外网访问Endpoint
- 内网访问Endpoint
- bucket的拥有者信息
- bucket的ACL（AccessControlList）

请求语法

```
GET /?bucketInfo HTTP/1.1
Host: BucketName.oss.example.com
Date: GMT Date
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-23: 响应元素

名称	描述
BucketInfo	保存Bucket信息内容的容器 类型：容器 子节点：Bucket节点 父节点：无
Bucket	保存Bucket具体信息的容器 类型：容器 父节点：BucketInfo节点
CreationDate	Bucket创建时间。时间格式 2013-07-31T10:56:21.000Z

名称	描述
	类型：时间 父节点：BucketInfo.Bucket
ExtranetEndpoint	Bucket访问的外网域名 类型：字符串 父节点：BucketInfo.Bucket
IntranetEndpoint	同区域ECS访问Bucket的内网域名 类型：字符串 父节点：BucketInfo.Bucket
Location	Bucket所在数据中心的区域 类型：字符串 父节点：BucketInfo.Bucket
Name	Bucket名字 类型：字符串 父节点：BucketInfo.Bucket
Owner	用于存放Bucket拥有者信息的容器。 类型：容器 父节点：BucketInfo.Bucket
ID	Bucket拥有者的用户ID。 类型：字符串 父节点：BucketInfo.Bucket.Owner
DisplayName	Bucket拥有者的名称 (目前和ID一致)。 类型：字符串 父节点：BucketInfo.Bucket.Owner
AccessControlList	存储ACL信息的容器 类型：容器 父节点：BucketInfo.Bucket
Grant	Bucket的ACL权限。 类型：枚举字符串 有效值：private、public-read、public-read-write父节点：BucketInfo.Bucket.AccessControlList

细节分析

1. 如果Bucket不存在，返回404错误。错误码：NoSuchBucket。

2. 只有Bucket的拥有者才能查看Bucket的信息，否则返回403 Forbidden错误，错误码：AccessDenied。
3. 请求可以从任何一个OSS的Endpoint发起。

示例

请求示例：

```
Get /?bucketInfo HTTP/1.1
Host: BucketName.oss.example.com
Date: Sat, 12 Sep 2015 07:51:28 GMT
Authorization: OSS q***c: B***=
```

成功获取Bucket信息的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Sat, 12 Sep 2015 07:51:28 GMT
Connection: keep-alive
Content-Length: 531
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<BucketInfo>
  <Bucket>
    <CreationDate>2013-07-31T10:56:21.000Z</CreationDate>
    <ExtranetEndpoint>regionid.example.com</ExtranetEndpoint>
    <IntranetEndpoint>regionid-internal.example.com</IntranetEndpoint>
    <Location>regionid</Location>
    <Name>oss-example</Name>
    <Owner>
      <DisplayName>***</DisplayName>
      <ID>2***3</ID>
    </Owner>
    <AccessControlList>
      <Grant>private</Grant>
    </AccessControlList>
  </Bucket>
</BucketInfo>
```

获取不存在的Bucket信息的返回示例：

```
HTTP/1.1 404
x-oss-request-id: 5***B
Date: Sat, 12 Sep 2015 07:51:28 GMT
Connection: keep-alive
Content-Length: 308
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<Error>
  <Code>NoSuchBucket</Code>
  <Message>The specified bucket does not exist.</Message>
  <RequestId>5***4</RequestId>
  <HostId>nosuchbucket.oss.example.com</HostId>
```

```
  <BucketName>nosuchbucket</BucketName>
</Error>
```

获取没有权限访问的Bucket信息的返回示例：

```
HTTP/1.1 403
x-oss-request-id: 5***C
Date: Sat, 12 Sep 2015 07:51:28 GMT
Connection: keep-alive
Content-Length: 209
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<Error>
  <Code>AccessDenied</Code>
  <Message>AccessDenied</Message>
  <RequestId>5***E</RequestId>
  <HostId>BucketName.oss.example.com</HostId>
</Error>
```

1.1.6.12 GetBucketLogging

Get Bucket Logging用于查看Bucket的访问日志配置情况。

请求语法

```
GET /?logging HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-24: 响应元素

名称	描述
BucketLoggingStatus	访问日志状态信息的容器 类型：容器 子元素：LoggingEnabled 父元素：无
LoggingEnabled	访问日志信息的容器。这个元素在开启时需要，关闭时不需要。 类型：容器 子元素：TargetBucket, TargetPrefix 父元素：BucketLoggingStatus
TargetBucket	指定存放访问日志的Bucket。 类型：字符 子元素：无 父元素：BucketLoggingStatus.LoggingEnabled

名称	描述
TargetPrefix	指定最终被保存的访问日志文件前缀。 类型：字符 子元素：无 父元素：BucketLoggingStatus.LoggingEnabled

细节分析

1. 如果Bucket不存在，返回404 no content错误。错误码：NoSuchBucket。
2. 只有Bucket的拥有者才能查看Bucket的访问日志配置情况，否则返回403 Forbidden错误，错误码：AccessDenied。
3. 如果源Bucket未设置Logging规则，OSS仍然返回一个XML消息体，但其中的BucketLoggingStatus元素为空。

示例

请求示例：

```
Get /?logging HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 04 May 2012 05:31:04 GMT
Authorization: OSS q***c:c***=
```

已设置LOG规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Fri, 04 May 2012 05:31:04 GMT
Connection: keep-alive
Content-Length: 210
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<BucketLoggingStatus xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
   <LoggingEnabled>
      <TargetBucket>mybucketlogs</TargetBucket>
      <TargetPrefix>mybucket-access_log/</TargetPrefix>
   </LoggingEnabled>
</BucketLoggingStatus>
```

未设置LOG规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Fri, 04 May 2012 05:31:04 GMT
Connection: keep-alive
Content-Length: 110
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
```

```
<BucketLoggingStatus xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
</BucketLoggingStatus>
```

1.1.6.13 GetBucketWebsite

Get Bucket Website操作用于查看bucket的静态网站托管状态。

请求语法

```
GET /?website HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-25: 响应元素

名称	描述
ErrorDocument	子元素Key的父元素 类型：容器 父元素：WebsiteConfiguration
IndexDocument	子元素Suffix的父元素 类型：容器 父元素：WebsiteConfiguration
Key	返回404错误时使用的文件名 类型：字符串 父元素：WebsiteConfiguration.ErrorDocument 有条件：当ErrorDocument设置时，必需
Suffix	返回目录URL时添加的索引文件名，不要为空，也不要包含"/"。例如索引文件设置为index.html，则访问：regionid.example.com/mybucket/mydir/这样请求的时候默认都相当于访问regionid.example.com/mybucket/index.html 类型：字符串 父元素：WebsiteConfiguration.IndexDocument
WebsiteConfiguration	请求的容器 类型：容器 父元素：无

细节分析

1. 如果Bucket不存在，返回404 no content错误。错误码：NoSuchBucket。

2. 只有Bucket的拥有者才能查看Bucket的静态网站托管状态，否则返回403 Forbidden错误，错误码：AccessDenied。
3. 如果源Bucket未设置静态网站托管功能，OSS会返回404错误，错误码为：NoSuchWebsiteConfiguration。

示例

请求示例：

```
Get /?website HTTP/1.1
Host: BucketName.regionid.example.com
Date: Thu, 13 Sep 2012 07:51:28 GMT
Authorization: OSS q***c:B***=
```

已设置LOG规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Thu, 13 Sep 2012 07:51:28 GMT
Connection: keep-alive
Content-Length: 218
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<WebsiteConfiguration xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
<IndexDocument>
<Suffix>index.html</Suffix>
   </IndexDocument>
   <ErrorDocument>
      <Key>error.html</Key>
   </ErrorDocument>
</WebsiteConfiguration>
```

未设置LOG规则的返回示例

```
HTTP/1.1 404
x-oss-request-id: 5***B
Date: Thu, 13 Sep 2012 07:56:46 GMT
Connection: keep-alive
Content-Length: 308
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<Error xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
   <Code>NoSuchWebsiteConfiguration</Code>
   <Message>The specified bucket does not have a website configuration.</Message>
   <BucketName>oss-example</BucketName>
   <RequestId>5***F</RequestId>
   <HostId>BucketName.regionid.example.com</HostId>
</Error>
```

1.1.6.14 GetBucketReferer

Get Bucket Referer操作用于查看bucket的Referer相关配置。

请求语法

```
GET /?referer HTTP/1.1
Host: BucketName.oss.example.com
Date: GMT Date
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-26: 响应元素

名称	描述
RefererConfiguration	保存Referer配置内容的容器 类型：容器 子节点：AllowEmptyReferer节点、RefererList节点 父节点：无
AllowEmptyReferer	指定是否允许referer字段为空的请求访问。 类型：枚举字符串 有效值：true 或 false 默认值：true 父节点：RefererConfiguration
RefererList	保存referer访问白名单的容器。 类型：容器 父节点：RefererConfiguration 子节点：Referer
Referer	指定一条referer访问白名单。 类型：字符串 父节点：RefererList

细节分析

1. 如果Bucket不存在，返回404错误。错误码：NoSuchBucket。
2. 只有Bucket的拥有者才能查看Bucket的Referer配置信息，否则返回403 Forbidden错误，错误码：AccessDenied。
3. 如果Bucket未进行Referer相关配置，OSS会返回默认的AllowEmptyReferer值和空的RefererList。

示例

请求示例：

```
Get /?referer HTTP/1.1
Host: BucketName.oss.example.com
Date: Thu, 13 Sep 2012 07:51:28 GMT
Authorization: OSS q***: B***=
```

已设置Referer规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Thu, 13 Sep 2012 07:51:28 GMT
Connection: keep-alive
Content-Length: 218
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<RefererConfiguration>
<AllowEmptyReferer>true</AllowEmptyReferer >
   <RefererList>
      <Referer> http://www.aliyun.com</Referer>
      <Referer> https://www.aliyun.com</Referer>
      <Referer> http://www.*.com</Referer>
      <Referer> https://www.?.aliyuncs.com</Referer>
   </RefererList>
</RefererConfiguration>
```

未设置Referer规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Thu, 13 Sep 2012 07:56:46 GMT
Connection: keep-alive
Content-Length: 308
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<RefererConfiguration>
<AllowEmptyReferer>true</AllowEmptyReferer >
< RefererList />
</RefererConfiguration>
```

1.1.6.15 GetBucketLifecycle

Get Bucket Lifecycle用于查看Bucket的Lifecycle配置。

```
GET /?lifecycle HTTP/1.1
Host: BucketName.oss.example.com
Date: GMT Date
```

```
Authorization: SignatureValue
```

细节分析

1. 只有Bucket的拥有者才能查看Bucket的Lifecycle配置，否则返回403 Forbidden错误，错误码：AccessDenied。
2. 如果Bucket或Lifecycle不存在，返回404 Not Found错误，错误码：NoSuchBucket或NoSuchLifecycle。

示例

请求示例：

```
Get /?lifecycle HTTP/1.1
Host: BucketName.oss.example.com
Date: Mon, 14 Apr 2014 01:17:29 GMT
Authorization: OSS q***:c***=
```

已设置Lifecycle的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***B
Date: Mon, 14 Apr 2014 01:17:29 GMT
Connection: keep-alive
Content-Length: 255
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<LifecycleConfiguration>
  <Rule>
    <ID>delete after one day</ID>
    <Prefix>logs/</Prefix>
    <Status>Enabled</Status>
    <Expiration>
      <Days>1</Days>
    </Expiration>
  </Rule>
</LifecycleConfiguration>
```

未设置Lifecycle的返回示例：

```
HTTP/1.1 404
x-oss-request-id: 5***B
Date: Mon, 14 Apr 2014 01:17:29 GMT
Connection: keep-alive
Content-Length: 278
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<Error>
  <BucketName>oss-example</BucketName>
  <Code>NoSuchLifecycle</Code>
  <Message>No Row found in Lifecycle Table.</Message>
  <RequestId>5***9</RequestId>
  <HostId> BucketName.oss.example.com</HostId>
```

```
</Error>
```

1.1.6.16 GetBucketTagging

Get Bucket Tagging接口用来获取某个Bucket关联的所有标签。

请求语法

```
GET /?tagging HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

请求元素（Request Elements）

表 1-27: 请求元素

名称	描述	是否必需
Tagging	Bucket Tag的容器。 类型：容器 子节点：TagSet 父节点：无	是
TagSet	保存一组Tag的容器。 类型：容器 子节点：Tag 父节点：Tagging	是
Tag	单条Tag的的容器。 类型：容器 子节点：Key或Value 父节点：TagSet	是
Key	标签的名称。 类型：字符串 子节点：无 父节点：Tag	是
Value	标签的值。 类型：字符串 子节点：无 父节点：Tag	是

细节分析

1. 如果请求的Bucket不存在，调用该接口时，OSS返回404 Not Found错误，错误码：NoSuchBucket。
2. 如果该Bucket没有关联任何标签，那么调用该接口时，OSS会返回空的TagSet。

示例

请求示例

```
GET /?tagging HTTP/1.1
Host: ***.regionid.example.com
Date: Thu, 23 Jun 2015 15:39:15 GMT
Authorization: OSS q***:C***=
```

返回示例

```
HTTP/1.1 200 OK
x-oss-request-id: 534B371674E88A4D8906008B
Date: Thu, 23 Jun 2015 15:39:15 GMT
Content-Length: 86
Content-Type: application/xml
Connection: close
Server: AliyunOSS
<?xml version="1.0" ?>
<Tagging>
 <TagSet>
 <Tag>
 <Key>Project</Key>
 <Value>Project One</Value>
 </Tag>
 <Tag>
 <Key>User</Key>
 <Value>jsmith</Value>
 </Tag>
 </TagSet>
</Tagging>
```

1.1.6.17 DeleteBucket

Delete Bucket用于删除某个Bucket。

请求语法

```
DELETE / HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. 如果Bucket不存在，返回404 no content错误。错误码：NoSuchBucket。

2. 为了防止误删除的发生，OSS不允许用户删除一个非空的Bucket。
3. 如果试图删除一个不为空的Bucket，返回409 Conflict错误，错误码：BucketNotEmpty。
4. 只有Bucket的拥有者才能删除这个Bucket。如果试图删除一个没有对应权限的Bucket，返回403 Forbidden错误。错误码：AccessDenied。

示例

请求示例：

```
DELETE / HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 05:31:04 GMT
Authorization: OSS q***:c***=
```

返回示例：

```
HTTP/1.1 204 No Content
x-oss-request-id: 534B371674E88A4D8906008B
Date: Fri, 24 Feb 2012 05:31:04 GMT
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
```

1.1.6.18 DeleteBucketLogging

Delete Bucket Logging操作用于关闭bucket访问日志记录功能。

请求语法

```
DELETE /?logging HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. 如果Bucket不存在，返回404 no content错误，错误码：NoSuchBucket。
2. 只有Bucket的拥有者才能关闭Bucket访问日志记录功能。如果试图操作一个不属于你的Bucket，OSS返回403 Forbidden错误，错误码：AccessDenied。
3. 如果目标Bucket并没有开启Logging功能，仍然返回HTTP状态码 204。

示例

请求示例

```
DELETE /?logging HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 05:35:24 GMT
```

```
Authorization: OSS q***c:6***=
```

返回示例

```
HTTP/1.1 204 No Content
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 05:35:24 GMT
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
```

1.1.6.19 DeleteBucketWebsite

Delete Bucket Website操作用于关闭bucket的静态网站托管模式。

请求语法

```
DELETE /?website HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. 如果Bucket不存在，返回404 no content错误，错误码：NoSuchBucket。
2. 只有Bucket的拥有者才能关闭Bucket的静态网站托管模式。如果试图操作一个不属于你的Bucket，OSS返回403 Forbidden错误，错误码：AccessDenied。

示例

请求示例：

```
DELETE /?website HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 05:45:34 GMT
Authorization: OSS q***:L***=
```

返回示例：

```
HTTP/1.1 204 No Content
x-oss-request-id: 5***B
Date: Fri, 24 Feb 2012 05:45:34 GMT
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
```

1.1.6.20 DeleteBucketLifecycle

通过Delete Bucket Lifecycle来删除指定Bucket的生命周期配置。

请求语法

```
DELETE /?lifecycle HTTP/1.1
Host: BucketName.oss.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. 本操作会删除指定Bucket的所有的生命周期规则。此后，该Bucket中不会有Object被自动删除。
2. 如果Bucket或Lifecycle不存在，返回404 Not Found错误，错误码：NoSuchBucket或NoSuchLifecycle。
3. 只有Bucket的拥有者才能删除Bucket的Lifecycle配置。如果试图操作一个不属于你的Bucket，OSS返回403 Forbidden错误，错误码：AccessDenied。

示例

请求示例：

```
DELETE /?lifecycle HTTP/1.1
Host: BucketName.oss.example.com
Date: Mon, 14 Apr 2014 01:17:35 GMT
Authorization: OSS q***:6***=
```

返回示例：

```
HTTP/1.1 204 No Content
x-oss-request-id: 5***B
Date: Mon, 14 Apr 2014 01:17:35 GMT
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
```

1.1.6.21 DeleteBucketTagging

Delete Bucket Tagging接口用来删除某个Bucket关联的所有标签。

请求语法

```
DELETE /?tagging HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. 如果Bucket不存在，OSS返回404 Not Found错误，错误码：NoSuchBucket。
2. 如果Bucket未关联任何标签，该接口仍返回204 No Content。

示例

请求示例

```
DELETE /?tagging HTTP/1.1
Host: ***.regionid.example.com
Date: Thu, 23 Jun 2015 15:39:18 GMT
Authorization: OSS q***c:C***=
```

返回示例

```
HTTP/1.1 204 No Content
x-oss-request-id: 5***B
Date: Thu, 23 Jun 2015 15:39:18 GMT
Connection: close
Content-Length: 0
Server: AliyunOSS
```

1.1.6.22 ListBucketByTagging

List Bucket By Tagging不会作为一个单独的接口开放出来，而是直接通过在GetService接口中增加查询参数来实现。可以根据用户指定的Tag Key和Tag Value来List出符合条件的Bucket。增加的两个查询参数如下：

表 1-28: 查询参数

名称	描述
tag-key	限定返回的bucket 必须带有tag-key指定的标签，可以不设定，不设定时不过滤标签信息。 数据类型：字符串 默认值：无
tag-value	限定返回的bucket必须带有tag-value指定的标签，必须与tag-key一起使用。可以不设定，不设定时，不过滤tag-value信息。 数据类型：字符串 默认值：无

示例

假设test_user一共有3个bucket：bucket1，bucket2，bucket3，其中bucket1和bucket2有Tag为key1:value1，bucket3有Tag为key3:value3。下面例子中List出tag-key=key1并且tag-value=value1的bucket：

请求示例

```
GET /tag-key=key1&tag-value=value1 HTTP/1.1
```

```
Date: Fri, 15 May 2015 11:18:32 GMT
Host: 10.97.188.37
Authorization: OSS n***:C***=
```

返回示例

```
HTTP/1.1 200 OK
Date: Fri, 15 May 2015 11:18:32 GMT
Content-Type: application/xml
Content-Length: 156
Connection: close
Server: AliyunOSS
x-oss-request-id: 5***4
<?xml version="1.0" encoding="UTF-8"?>
<ListAllMyBucketsResult>
 <Owner>
 <ID>***</ID>
 <DisplayName>***</DisplayName>
 </Owner>
 <Buckets>
 <Bucket>
 <Location>oss-cn-hangzhou</Location>
 <Name>bucket1</Name>
 <CreationDate>2015-05-15T11:18:32.000Z</CreationDate>
 </Bucket>
 <Bucket>
 <Location>oss-cn-hangzhou</Location>
 <Name>bucket2</Name>
 <CreationDate>2015-05-15T11:18:32.000Z</CreationDate>
 </Bucket>
 </Buckets>
</ListAllMyBucketsResult>
```

1.1.7 关于Object的操作

1.1.7.1 PutObject

Put Object用于上传文件。

请求语法

```
PUT /ObjectName HTTP/1.1
Content-Length：ContentLength
Content-Type: ContentType
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

请求Header

表 1-29: 请求Header

名称	描述
Cache-Control	指定该Object被下载时的网页的缓存行为；更详细描述请参照*RFC2616*。

名称	描述
	类型：字符串 默认值：无
Content-Disposition	指定该Object被下载时的名称；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-Encoding	指定该Object被下载时的内容编码格式；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-MD5	根据协议RFC 1864对消息内容（不包括头部）计算MD5值获得128比特位数字，对该数字进行base64编码为一个消息的Content-MD5值。该请求头可用于消息合法性的检查（消息内容是否与发送时一致）。虽然该请求头是可选项，OSS建议用户使用该请求头进行端到端检查。 类型：字符串 默认值：无 限制：无
Expires	过期时间；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无 **说明**： OSS不会对这个值进行限制和验证
x-oss-server-side-encryption	指定oss创建object时的服务器端加密编码算法。 类型：字符串 合法值：AES256
x-oss-object-acl	指定oss创建object时的访问权限。 类型：字符串 合法值：public-read，private，public-read-write

细节分析

1. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。
2. 如果请求头中的“Content-Length”值小于实际请求体（body）中传输的数据长度，OSS仍将成功创建文件；但Object大小只等于“Content-Length”中定义的大小，其他数据将被丢弃。

3. 如果试图添加的Object的同名文件已经存在，并且有访问权限。新添加的文件将覆盖原来的文件，成功返回200 OK。
4. 如果在PutObject的时候，携带以x-oss-meta-为前缀的参数，则视为user meta，比如x-oss-meta-location。一个Object可以有多个类似的参数，但所有的user meta总大小不能超过8k。
5. 如果Head中没有加入Content length参数，会返回411 Length Required错误。错误码：MissingContentLength。
6. 如果设定了长度，但是没有发送消息Body，或者发送的body大小小于给定大小，服务器会一直等待，直到time out，返回400 Bad Request消息。错误码：RequestTimeout。
7. 如果试图添加的Object所在的Bucket不存在，返回404 Not Found错误。错误码：NoSuchBucket。
8. 如果试图添加的Object所在的Bucket没有访问权限，返回403 Forbidden错误。错误码：AccessDenied。
9. 如果添加文件长度超过5G，返回错误消息400 Bad Request。错误码：InvalidArgument。
10. 如果传入的Object key长度大于1023字节，返回400 Bad Request。错误码：InvalidObjectName。
11. PUT一个Object的时候，OSS支持5个 HTTP *RFC2616*协议规定的Header 字段：Cache-Control、Expires、Content-Encoding、Content-Disposition、Content-Type。如果上传Object时设置了这些Header，则这个Object被下载时，相应的Header值会被自动设置成上传时的值。
12. 如果上传Object时指定了x-oss-server-side-encryption Header，则必须设置其值为AES256，否则会返回400和相应错误提示：InvalidEncryptionAlgorithmError。指定该Header后，在响应头中也会返回该Header，OSS会对上传的Object进行加密编码存储，当这个Object被下载时，响应头中会包含x-oss-server-side-encryption，值被设置成该Object的加密算法。

常见问题

Content-MD5计算方式错误以上传的内容为"123456789"来说，计算这个字符串的Content-MD5 正确的计算方式：标准中定义的算法简单点说就是： 1. 先计算MD5加密的二进制数组（128位）。 2. 再对这个二进制进行base64编码（而不是对32位字符串编码）。 以Python为例子： 正确计算的代码为：

```
>>> import base64,hashlib
>>> hash = hashlib.md5()
>>> hash.update("0123456789")
>>> base64.b64encode(hash.digest())
```

```
'eB5eJF1ptWaXm4bijSPyxw=='
```

需要注意正确的是：`hash.digest()`，计算出进制数组（128位） `>>> hash.digest() 'x\x1e^$]i\xb5f\x97\x9b\x86\xe2\x8d#\xf2\xc7'` 常见错误是直接对计算出的32位字符串编码进行base64编码。例如，错误的是：`hash.hexdigest()`，计算得到可见的32位字符串编码 `>>> hash.hexdigest() '781e5e245d69b566979b86e28d23f2c7'` 错误的MD5值进行base64编码后的结果： `>>> base64.b64encode(hash.hexdigest()) 'NzgxZTVlMjQ1ZDY5YjU2Njk3OWI4NmUyOGQyM2YyYzc='`

示例

请求示例：

```
PUT /oss.jpg HTTP/1.1
Host: ***.regionid.example.com
Cache-control: no-cache
Expires: Fri, 28 Feb 2012 05:38:42 GMT
Content-Encoding: utf-8
Content-Disposition: attachment;filename=oss_download.jpg
Date: Fri, 24 Feb 2012 06:03:28 GMT
Content-Type: image/jpg
Content-Length: 344606
Authorization: OSS q***:k***=

[344606 bytes of object data]
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
Date: Sat, 21 Nov 2015 18:52:34 GMT
Content-Length: 0
Connection: keep-alive
x-oss-request-id: 5***A
x-oss-bucket-version: 1***9
ETag: "A***2"
```

1.1.7.2 CopyObject

拷贝一个在OSS上已经存在的object成另外一个object，可以发送一个PUT请求给OSS，并在PUT请求头中添加元素“x-oss-copy-source”来指定拷贝源。OSS会自动判断出这是一个Copy操作，并直接在服务器端执行该操作。如果拷贝成功，则返回新的object信息给用户。该操作适用于拷贝小于1GB的文件，当拷贝一个大于1GB的文件时，必须使用Multipart Upload操作，具体见Upload Part Copy。

请求语法

```
PUT /DestObjectName HTTP/1.1
Host: DestBucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

```
x-oss-copy-source: /SourceBucketName/SourceObjectName
```

请求Header

表 1-30: 请求Header

名称	描述
x-oss-copy-source	复制源地址（必须有可读权限） 类型：字符串 默认值：无
x-oss-copy-source-if-match	如果源Object的ETag值和用户提供的ETag相等，则执行拷贝操作，并返回200；否则返回412 HTTP错误码（预处理失败）。 类型：字符串 默认值：无
x-oss-copy-source-if-none-match	如果源Object的ETag值和用户提供的ETag不相等，则执行拷贝操作，并返回200；否则返回304 HTTP错误码（预处理失败）。 类型：字符串 默认值：无
x-oss-copy-source-if-unmodified-since	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件，并返回200 OK；否则返回412 precondition failed错误。 类型：字符串 默认值：无
x-oss-copy-source-if-modified-since	如果源Object自从用户指定的时间以后被修改过，则执行拷贝操作；否则返回304 HTTP错误码（预处理失败）。 类型：字符串 默认值：无
x-oss-metadata-directive	有效值为COPY和REPLACE。如果该值设为COPY，则新的Object的meta都从源Object复制过来；如果设为REPLACE，则忽视所有源Object的meta值，而采用用户这次请求中指定的meta值；其他值则返回400 HTTP错误码。注意该值为COPY时，源Object的x-oss-server-side-encryption的meta值不会进行拷贝。 类型：字符串 默认值：COPY 有效值：COPY、REPLACE
x-oss-server-side-encryption	指定oss创建目标object时的服务器端熵编码加密算法 类型：字符串 有效值：AES256
x-oss-object-acl	指定oss创建object时的访问权限。

名称	描述
	类型：字符串 合法值：public-read，private，public-read-write

响应元素(Response Elements)

表 1-31: 响应元素

名称	描述
CopyObjectResult	Copy Object结果类型：字符串默认值：无
ETag	新Object的ETag值。类型：字符串父元素：CopyObjectResult
LastModified	新Object最后更新时间。类型：字符串父元素：CopyObjectResult

细节分析

1. 可以通过拷贝操作来实现修改已有Object的meta信息。
2. 如果拷贝操作的源Object地址和目标Object地址相同，则无论x-oss-metadata-directive为何值，都会直接替换源Object的meta信息。
3. OSS支持拷贝操作的四个预判断Header任意个同时出现，相应逻辑参见Get Object操作的细节分析。
4. 拷贝操作需要请求者对源Object有读权限。
5. 源Object和目标Object必须属于同一个数据中心，否则返回403 AccessDenied，错误信息为：Target object does not reside in the same data center as source object。
6. 拷贝操作的计费统计会对源Object所在的Bucket增加一次Get请求次数，并对目标Object所在的Bucket增加一次Put请求次数，以及相应的新增存储空间。
7. 拷贝操作涉及到的请求头，都是以“x-oss-”开头的，所以要加入签名字符串中。
8. 若在拷贝操作中指定了x-oss-server-side-encryption请求头，并且请求值合法（为AES256），则无论源Object是否进行过服务器端加密编码，拷贝之后的目标Object都会进行服务器端加密编码。并且拷贝操作的响应头中会包含x-oss-server-side-encryption，值被设置成目标Object的加密算法。在这个目标Object被下载时，响应头中也会包含x-oss-server-side-encryption，值被设置成该Object的加密算法；若拷贝操作中未指定x-oss-server-side-encryption请求头，则无论源Object是否进行过服务器端加密编码，拷贝之后的目标Object都是未进行过服务器端加密编码加密的数据。

9. 拷贝操作中x-oss-metadata-directive请求头为COPY（默认值）时，并不拷贝源Object的x-oss-server-side-encryption值，即目标Object是否进行服务器端加密编码只根据COPY操作是否指定了x-oss-server-side-encryption请求头来决定。
10. 若在拷贝操作中指定了x-oss-server-side-encryption请求头，并且请求值非AES256，则返回400和相应的错误提示：InvalidEncryptionAlgorithmError。
11. 如果拷贝的文件大小大于1GB，会返回400和错误提示：EntityTooLarge。
12. 该操作不能拷贝通过Append追加上传方式产生的object。
13. 如果文件类型为**符号链接**，只拷贝符号链接。

示例

请求示例：

```
PUT /copy_oss.jpg HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 07:18:48 GMT
x-oss-copy-source: /oss-example/oss.jpg
Authorization: OSS q***:g***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
Content-Type: application/xml
Content-Length: 193
Connection: keep-alive
Date: Fri, 24 Feb 2012 07:18:48 GMT
Server: AliyunOSS
<?xml version="1.0" encoding="UTF-8"?>
<CopyObjectResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
 <LastModified>Fri, 24 Feb 2012 07:18:48 GMT</LastModified>
 <ETag>"5***E"</ETag>
</CopyObjectResult>
```

1.1.7.3 GetObject

用于获取某个Object，此操作要求用户对该Object有读权限。

请求语法

```
GET /ObjectName HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

```
Range: bytes=ByteRange(可选)
```

请求参数(Request Parameters)

OSS支持用户在发送GET请求时，可以自定义OSS返回请求中的一些Header，前提条件用户发送的GET请求必须携带签名。这些Header包括：

表 1-32: 请求参数

名称	描述
response-content-type	设置OSS返回请求的content-type头 类型：字符串 默认值：无
response-content-language	设置OSS返回请求的content-language头 类型：字符串 默认值：无
response-expires	设置OSS返回请求的expires头 类型：字符串 默认值：无
response-cache-control	设置OSS返回请求的cache-control头 类型：字符串 默认值：无
response-content-disposition	设置OSS返回请求的content-disposition头 类型：字符串 默认值：无
response-content-encoding	设置OSS返回请求的content-encoding头 类型：字符串 默认值：无

请求Header

表 1-33: 请求Header

名称	描述
Range	指定文件传输的范围。如，设定 bytes=0-9，表示传送第0到第9这10个字符。 类型：字符串 默认值：无

名称	描述
If-Modified-Since	如果指定的时间早于实际修改时间，则正常传送文件，并返回200 OK；否则返回304 not modified 类型：字符串 默认值：无 时间格式：GMT时间，例如Fri, 13 Nov 2015 14:47:53 GMT
If-Unmodified-Since	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件，并返回200 OK；否则返回412 precondition failed错误 类型：字符串 默认值：无 时间格式：GMT时间，例如Fri, 13 Nov 2015 14:47:53 GMT
If-Match	如果传入期望的ETag和object的 ETag匹配，则正常传输文件，并返回200 OK；否则返回412 precondition failed错误 类型：字符串 默认值：无
If-None-Match	如果传入的ETag值和Object的ETag不匹配，则正常传输文件，并返回200 OK；否则返回304 Not Modified 类型：字符串 默认值：无

细节分析

1. GetObject通过range参数可以支持断点续传，对于比较大的Object建议使用该功能。
2. 如果在请求头中使用Range参数；则返回消息中会包含整个文件的长度和此次返回的范围，例如：Content-Range: bytes 0-9/44，表示整个文件长度为44，此次返回的范围为0-9。如果不符合范围规范，则传送整个文件，并且不在结果中提及Content-Range。
3. 如果“If-Modified-Since”元素中设定的时间不符合规范，直接返回文件，并返回200 OK。
4. If-Modified-Since和If-Unmodified-Since可以同时存在，If-Match和If-None-Match也可以同时存在。
5. 如果包含If-Unmodified-Since并且不符合或者包含If-Match并且不符合，返回412 precondition failed
6. 如果包含If-Modified-Since并且不符合或者包含If-None-Match并且不符合，返回304 Not Modified
7. 如果文件不存在返回404 Not Found错误。错误码：NoSuchKey。
8. OSS不支持在匿名访问的GET请求中，通过请求参数来自定义返回请求的header。

9. 在自定义OSS返回请求中的一些Header时，只有请求处理成功（即返回码为200时），OSS才会将请求的header设置成用户GET请求参数中指定的值。
10. 若该Object为进行服务器端熵编码加密存储的，则在GET请求时会自动解密返回给用户，并且在响应头中，会返回x-oss-server-side-encryption，其值表明该Object的服务器端加密算法。
11. 需要将返回内容进行 GZIP压缩传输的用户，需要在请求的Header中显示方式加入 Accept-Encoding:gzip，OSS会根据文件的Content-Type和文件大小，判断是否返回给用户经过GZIP 压缩的数据。如果采用了GZIP压缩则不会附带etag 信息。目前OSS支持GZIP压缩的Content-Type为HTML、Javascript、CSS、XML、RSS、Json，文件大小需不小于1k。
12. 如果文件类型为**符号链接**，返回目标文件的内容。并且， 响应头中Content-Length、ETag、Content-Md5 均为目标文件的元信息；Last-Modified是目标文件和符号链接的最大值；其他均为符号链接的元信息。
13. 如果文件类型为**符号链接**，并且目标文件不存在，返回404 Not Found错误。错误码：SymlinkTargetNotExist。
14. 如果文件类型为**符号链接**，并且目标文件类型是符号链接，返回400 Bad request错误。错误码：InvalidTargetType。

示例

请求示例：

```
GET /oss.jpg HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 06:38:30 GMT
Authorization: OSS q***:U***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 3***1
x-oss-object-type: Normal
Date: Fri, 24 Feb 2012 06:38:30 GMT
Last-Modified: Fri, 24 Feb 2012 06:07:48 GMT
ETag: "5***E "
Content-Type: image/jpg
Content-Length: 344606
Server: AliyunOSS
[344606 bytes of object data]
```

Range请求示例：

```
GET //oss.jpg HTTP/1.1
Host:***. regionid.example.com
Date: Fri, 28 Feb 2012 05:38:42 GMT
Range: bytes=100-900
```

```
Authorization: OSS q***:q***=
```

返回示例：

```
HTTP/1.1 206 Partial Content
x-oss-request-id: 2***9
x-oss-object-type: Normal
Date: Fri, 28 Feb 2012 05:38:42 GMT
Last-Modified: Fri, 24 Feb 2012 06:07:48 GMT
ETag: "5***E "
Accept-Ranges: bytes
Content-Range: bytes 100-900/344606
Content-Type: image/jpg
Content-Length: 801
Server: AliyunOSS
[801 bytes of object data]
```

自定义返回消息头的请求示例：

```
GET /oss.jpg?response-expires=Thu%2C%2001%20Feb%202012%2017%3A00%3A00%
20GMT& response-content-type=text&response-cache-control=No-cache&response-content-
disposition=attachment%253B%2520filename%253Dtesting.txt&response-content-encoding=
utf-8&response-content-language=%E4%B8%AD%E6%96%87 HTTP/1.1
Host: ***.regionid.example.com:
Date: Fri, 24 Feb 2012 06:09:48 GMT
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
x-oss-object-type: Normal
Date: Fri, 24 Feb 2012 06:09:48 GMT
Last-Modified: Fri, 24 Feb 2012 06:07:48 GMT
ETag: "5***E "
Content-Length: 344606
Connection: keep-alive
Content-disposition: attachment; filename:testing.txt
Content-language: 中文
Content-encoding: utf-8
Content-type: text
Cache-control: no-cache
Expires: Fri, 24 Feb 2012 17:00:00 GMT
Server: AliyunOSS
[344606 bytes of object data]
```

符号链接的请求示例：

```
GET /link-to-oss.jpg HTTP/1.1
Accept-Encoding: identity
Date: Tue, 08 Nov 2016 03:17:58 GMT
Host: ***.regionid.example.com
Authorization: OSS q***:q***=
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
```

```
Date: Tue, 08 Nov 2016 03:17:58 GMT
Content-Type: application/octet-stream
Content-Length: 20
Connection: keep-alive
x-oss-request-id: 5***D
Accept-Ranges: bytes
ETag: "8***7"
Last-Modified: Tue, 08 Nov 2016 03:17:58 GMT
x-oss-object-type: Symlink
Content-MD5: g***w==
```

1.1.7.4 AppendObject

Append Object以追加写的方式上传文件。通过Append Object操作创建的Object类型为Appendable Object，而通过Put Object上传的Object是Normal Object。

请求语法

```
POST /ObjectName?append&position=Position HTTP/1.1
Content-Length：ContentLength
Content-Type: ContentType
Host: BucketName.oss.example.com
Date: GMT Date
Authorization: SignatureValue
```

请求Header

表 1-34: 请求Header

名称	描述
Cache-Control	指定该Object被下载时的网页的缓存行为；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-Disposition	指定该Object被下载时的名称；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-Encoding	指定该Object被下载时的内容编码格式；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-MD5	根据协议RFC 1864对消息内容（不包括头部）计算MD5值获得128比特位数字，对该数字进行base64编码为一个消息的Content-MD5值。该请求头可用于消息合法性的检查（消息内容是否与发送时一致）。虽然该请求头是可选项，OSS建议用户使用该请求头进行端到端检查。 类型：字符串 默认值：无

名称	描述
	限制：无
Expires	过期时间；更详细描述请参照*RFC2616*。 类型：整数 默认值：无
x-oss-server-side-encryption	指定oss创建object时的服务器端加密编码算法。 类型：字符串 合法值：AES256
x-oss-object-acl	指定oss创建object时的访问权限。 类型：字符串 合法值：public-read，private，public-read-write

响应Header

表 1-35: 响应Header

名称	描述
x-oss-next-append-position	指明下一次请求应当提供的position。实际上就是当前Object长度。 当Append Object成功返回，或是因position和Object长度不匹配而引起的409错误时，会包含此header。 类型：64位整型
x-oss-hash-crc64ecma	表明Object的64位CRC值。该64位CRC根据*ECMA-182*标准计算得出。 类型：64位整型

和其他操作的关系

1. 不能对一个非Appendable Object进行Append Object操作。例如，已经存在一个同名Normal Object时，Append Object调用返回409，错误码ObjectNotAppendable。
2. 对一个已经存在的Appendable Object进行Put Object操作，那么该Appendable Object会被新的Object覆盖，类型变为Normal Object。
3. Head Object操作会返回x-oss-object-type，用于表明Object的类型。对于Appendable Object来说，该值为Appendable。对Appendable Object，Head Object也会返回上述的x-oss-next-append-position和x-oss-hash-crc64ecma。
4. Get Bucket（List Objects）请求的响应XML中，会把Appendable Object的Type设为Appendable

5. 不能使用Copy Object来拷贝一个Appendable Object，也不能改变它的服务器端加密的属性。可以使用Copy Object来改变用户自定义元信息。

细节分析

1. URL参数append和position均为CanonicalizedResource，需要包含在签名中。
2. URL的参数必须包含append，用来指定这是一个Append Object操作。
3. URL查询参数还必须包含position，其值指定从何处进行追加。首次追加操作的position必须为0，后续追加操作的position是Object的当前长度。例如，第一次Append Object请求指定position值为0，content-length是65536；那么，第二次Append Object需要指定position为65536。每次操作成功后，响应头部x-oss-next-append-position也会标明下一次追加的position。
4. 如果position的值和当前Object的长度不一致，OSS会返回409错误，错误码为PositionNotEqualToLength。发生上述错误时，用户可以通过响应头部x-oss-next-append-position来得到下一次position，并再次进行请求。
5. 当Position值为0时，如果没有同名Appendable Object，或者同名Appendable Object长度为0，该请求成功；其他情况均视为Position和Object长度不匹配的情形。
6. 当Position值为0，且没有同名Object存在，那么Append Object可以和Put Object请求一样，设置诸如x-oss-server-side-encryption之类的请求Header。这一点和Initiate Multipart Upload是一样的。如果在Position为0的请求时，加入了正确的x-oss-server-side-encryption头，那么后续的Append Object响应头部也会包含x-oss-server-side-encryption头，其值表明加密算法。后续如果需要更改meta，可以使用Copy Object请求。
7. 由于并发的关系，即使用户把position的值设为了x-oss-next-append-position，该请求依然可能因为PositionNotEqualToLength而失败。
8. Append Object产生的Object长度限制和Put Object一样。
9. 每次Append Object都会更新该Object的最后修改时间。
10. 在position值正确的情况下，对已存在的Appendable Object追加一个长度为0的内容，该操作不会改变Object的状态。

CRC64的计算方式

Appendable Object的CRC采用*ECMA-182*标准，和XZ的计算方式一样。用Boost CRC模块的方式来定义则有：

```
typedef boost::crc_optimal<64, 0x42F0E1EBA9EA3693ULL, 0xffffffffffffffffULL, 0xffffffffffffffffULL,
true, true> boost_ecma;

uint64_t do_boost_crc(const char* buffer, int length)
{
```

```
    boost_ecma crc;
    crc.process_bytes(buffer, length);
    return crc.checksum();
}
```

或是用Python crcmod的方式为：

```
do_crc64 = crcmod.mkCrcFun(0x142F0E1EBA9EA3693L, initCrc=0L, xorOut=0xffffffffffffffffL,
rev=True)

print do_crc64(“123456789”)
```

示例

请求示例：

```
POST /oss.jpg?append&position=0 HTTP/1.1
Host: BucketName.oss.example.com
Cache-control: no-cache
Expires: Wed, 08 Jul 2015 16:57:01 GMT
Content-Encoding: utf-8
Content-Disposition: attachment;filename=oss_download.jpg
Date: Wed, 08 Jul 2015 06:57:01 GMT
Content-Type: image/jpg
Content-Length: 1717
Authorization: OSS q***:k***=

[1717 bytes of object data]
```

返回示例：

```
HTTP/1.1 200 OK
Date: Wed, 08 Jul 2015 06:57:01 GMT
ETag: "0***0"
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
x-oss-hash-crc64ecma: 14741617095266562575
x-oss-next-append-position: 1717
x-oss-request-id: 5***1
```

1.1.7.5 DeleteObject

DeleteObject用于删除某个Object。

请求语法

```
DELETE /ObjectName HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. DeleteObject要求对该Object要有写权限。

2. 如果要删除的Object不存在，OSS也返回状态码204（ No Content ）。
3. 如果Bucket不存在，返回404 Not Found。
4. 如果文件类型为**符号链接**，只删除符号链接自身。

示例

请求示例：

```
DELETE /copy_oss.jpg HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 07:45:28 GMT
Authorization: OSS q***:z***=
```

返回示例：

```
HTTP/1.1 204 NoContent
x-oss-request-id: 5***1
Date: Fri, 24 Feb 2012 07:45:28 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.7.6 DeleteMultipleObjects

Delete Multiple Objects操作支持用户通过一个HTTP请求删除同一个Bucket中的多个Object。

Delete Multiple Objects操作支持一次请求内最多删除1000个Object，并提供两种返回模式：详细(verbose)模式和简单(quiet)模式：

- 详细模式：OSS返回的消息体中会包含每一个删除Object的结果。
- 简单模式：OSS返回的消息体中只包含删除过程中出错的Object结果；如果所有删除都成功的话，则没有消息体。

请求语法

```
POST /?delete HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Content-Length: ContentLength
Content-MD5: MD5Value
Authorization: SignatureValue

<?xml version="1.0" encoding="UTF-8"?>
<Delete>
  <Quiet>true</Quiet>
  <Object>
    <Key>key</Key>
  </Object>
...
```

```
</Delete>
```

请求参数(Request Parameters)

Delete Multiple Objects时，可以通过encoding-type对返回结果中的Key进行编码。

表 1-36: 请求参数

名称	描述
encoding-type	指定对返回的Key进行编码，目前支持url编码。Key使用UTF-8字符，但xml 1.0标准不支持解析一些控制字符，比如ascii值从0到10的字符。对于Key中包含xml 1.0标准不支持的控制字符，可以通过指定encoding-type对返回的Key进行编码。 数据类型：字符串 默认值：无 可选值：url

请求元素(Request Elements)

表 1-37: 请求元素

名称	描述
Delete	保存Delete Multiple Object请求的容器。 类型：容器 子节点：一个或多个Object元素，可选的Quiet元素 父节点： None
Key	被删除Object的名字。 类型：字符串 父节点：Object
Object	保存一个Object信息的容器。 类型：容器 子节点：key 父节点：Delete
Quiet	打开“简单”响应模式的开关。 类型：枚举字符串 有效值：true、false 默认值：false 父节点：Delete

响应元素(Response Elements)

表 1-38: 响应元素

名称	描述
Deleted	保存被成功删除的Object的容器。 类型：容器 子节点：Key 父节点：DeleteResult
DeleteResult	保存Delete Multiple Object请求结果的容器。类型：容器 子节点：Deleted 父节点：None
Key	OSS执行删除操作的Object名字。 类型：字符串 父节点：Deleted
EncodingType	指明返回结果中编码使用的类型。如果请求的参数中指定了encoding-type，那返回的结果会对Key进行编码。 类型：字符串 父节点：容器

细节分析

1. Delete Multiple Objects请求必须填Content-Length和Content-MD5字段。OSS会根据这些字段验证收到的消息体是正确的，之后才会执行删除操作。
2. 生成Content-MD5字段内容方法：首先将Delete Multiple Object请求内容经过MD5加密后得到一个128位字节数组；再将该字节数组用base64算法编码；最后得到的字符串即是Content-MD5字段内容。
3. Delete Multiple Objects请求默认是详细(verbose)模式。
4. 在Delete Multiple Objects请求中删除一个不存在的Object，仍然认为是成功的。
5. Delete Multiple Objects的消息体最大允许2MB的内容，超过2MB会返回MalformedXML错误码。
6. Delete Multiple Objects请求最多允许一次删除1000个Object；超过1000个Object会返回MalformedXML错误码。
7. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。

示例

请求示例I：

```
POST /?delete HTTP/1.1
Host: ***.regionid.example.com
Date: Wed, 29 Feb 2012 12:26:16 GMT
Content-Length:151
Content-MD5: o***==
Authorization: OSS q***:+z***=

<?xml version="1.0" encoding="UTF-8"?>

<Delete>
  <Quiet>false</Quiet>
  <Object>
    <Key>multipart.data</Key>
  </Object>
  <Object>
    <Key>test.jpg</Key>
  </Object>
  <Object>
    <Key>demo.jpg</Key>
  </Object>
</Delete>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 7***7
Date: Wed, 29 Feb 2012 12:26:16 GMT
Content-Length: 244
Content-Type: application/xml
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
<DeleteResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
    <Deleted>
       <Key>multipart.data</Key>
    </Deleted>
    <Deleted>
       <Key>test.jpg</Key>
    </Deleted>
    <Deleted>
       <Key>demo.jpg</Key>
    </Deleted>
</DeleteResult>
```

请求示例II：

```
POST /?delete HTTP/1.1
Host: ***.regionid.example.com
Date: Wed, 29 Feb 2012 12:33:45 GMT
Content-Length:151
Content-MD5: o***A==
Authorization: OSS q***c:W***=

<?xml version="1.0" encoding="UTF-8"?>
```

```
<Delete>
  <Quiet>true</Quiet>
  <Object>
    <Key>multipart.data</Key>
  </Object>
  <Object>
    <Key>test.jpg</Key>
  </Object>
  <Object>
    <Key>demo.jpg</Key>
  </Object>
</Delete>
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
Date: Wed, 29 Feb 2012 12:33:45 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.7.7 HeadObject

Head Object只返回某个Object的meta信息，不返回文件内容。

请求语法

```
HEAD /ObjectName HTTP/1.1
Host: BucketName/regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

请求Header

表 1-39: 请求Header

名称	描述
If-Modified-Since	如果指定的时间早于实际修改时间，则返回200 OK和Object Meta；否则返回304 not modified 类型：字符串默认值：无
If-Unmodified-Since	如果传入参数中的时间等于或者晚于文件实际修改时间，则返回200 OK和Object Meta；否则返回412 precondition failed错误类型：字符串 默认值：无
If-Match	如果传入期望的ETag和object的 ETag匹配，则返回200 OK和Object Meta；否则返回412 precondition failed错误类型：字符串 默认值：无
If-None-Match	如果传入的ETag值和Object的ETag不匹配，则返回200 OK和Object Meta；否则返回304 Not Modified 类型：字符串 默认值：无

细节分析

1. 不论正常返回200 OK还是非正常返回，Head Object都不返回消息体。
2. HeadObject支持在头中设定If-Modified-Since, If-Unmodified-Since, If-Match，If-None-Match的查询条件。具体规则请参见GetObject中对应的选项。如果没有修改，返回304 Not Modified。
3. 如果用户在PutObject的时候传入以x-oss-meta-为开头的user meta，比如x-oss-meta-location，返回消息时，返回这些user meta。
4. 如果文件不存在返回404 Not Found错误。
5. 若该Object为进行服务器端熵编码加密存储的，则在Head请求响应头中，会返回x-oss-server-side-encryption，其值表明该Object的服务器端加密算法。
6. 如果文件类型为**符号链接**， 响应头中Content-Length、ETag、Content-Md5 均为目标文件的元信息；Last-Modified是目标文件和符号链接的最大值；其他均为符号链接元信息。
7. 如果文件类型为**符号链接**，并且目标文件不存在，返回404 Not Found错误。错误码：SymlinkTargetNotExist。
8. 如果文件类型为**符号链接**，并且目标文件类型是符号链接，返回400 Bad request错误。错误码：InvalidTargetType。

示例

请求示例：

```
HEAD /oss.jpg HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 07:32:52 GMT
Authorization: OSS q***:J***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
x-oss-object-type: Normal
Date: Fri, 24 Feb 2012 07:32:52 GMT
Last-Modified: Fri, 24 Feb 2012 06:07:48 GMT
ETag: "f***8"
Content-Length: 344606
Content-Type: image/jpg
Connection: keep-alive
Server: AliyunOSS
```

1.1.7.8 GetObjectMeta

Get Object Meta用来获取某个Bucket下的某个Object的基本meta信息，包括该Object的ETag、Size（文件大小）、LastModified，并不返回其内容。

请求语法

```
GET /ObjectName?objectMeta HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

细节分析：

1. 无论正常返回还是非正常返回，Get Object Meta均不返回消息体。
2. Get Object Meta需包含objectMeta请求参数，否则表示Get Object请求。
3. 如果文件不存在返回404 Not Found错误。
4. Get Object Meta相比Head Object更轻量，仅返回指定Object的少量基本meta信息，包括该Object的ETag、Size（文件大小）、LastModified，其中Size由响应头Content-Length的数值表示。
5. 如果文件类型为**符号链接**，返回符号链接自身信息。

示例

请求示例：

```
GET /oss.jpg?objectMeta HTTP/1.1
Host: ***.regionid.example.com
Date: Wed, 29 Apr 2015 05:21:12 GMT
Authorization: OSS q***c:C***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
Date: Wed, 29 Apr 2015 05:21:12 GMT
ETag: "5***E"
Last-Modified: Fri, 24 Feb 2012 06:07:48 GMT
Content-Length: 344606
Connection: keep-alive
Server: AliyunOSS
```

1.1.7.9 PutObjectACL

Put Object ACL接口用于修改Object的访问权限。目前Object有三种访问权限：private, public-read , public-read-write。Put Object ACL操作通过Put请求中的“x-oss-object-acl”头来设置，这个操作只有Bucket Owner有权限执行。如果操作成功，则返回200；否则返回相应的错误码和提示信息。

请求语法

```
PUT /ObjectName?acl HTTP/1.1
x-oss-object-acl: Permission
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

Object ACL释义

表 1-40: Object ACL释义

名称	描述
private	该ACL表明某个Object是私有资源，即只有该Object的Owner拥有该Object的读写权限，其他的用户没有权限操作该Object
public-read	该ACL表明某个Object是公共读资源，即非Object Owner只有该Object的读权限，而Object Owner拥有该Object的读写权限
public-read-write	该ACL表明某个Object是公共读写资源，即所有用户拥有对该Object的读写权限
default	该ACL表明某个Object是遵循Bucket读写权限的资源，即Bucket是什么权限，Object就是什么权限

细节分析

1. Object的读操作包括：GetObject，HeadObject，CopyObject和UploadPartCopy中的对source object的读；Object的写操作包括：
   - PutObject
   - PostObject
   - AppendObject
   - DeleteObject
   - DeleteMultipleObjects
   - CompleteMultipartUpload
   - CopyObject对新的Object的写
2. x-oss-object-acl中权限的值必须在上述3种权限中。如果有不属于上述3种的权限，OSS返回400 Bad Request消息，错误码：InvalidArgument。

3. 用户不仅可以通过PutObjectACL接口来设置Object ACL，还可以在Object的写操作时，在请求头中带上x-oss-object-acl来设置Object ACL，效果与PutObjectA ACL等同。例如PutObject时在请求头中带上x-oss-object-acl可以在上传一个Object的同时设置某个Object的ACL。
4. 对某个Object没有读权限的用户读取某个Object时，OSS返回 403 Forbidden消息，错误码：AccessDenied，提示：You do not have read permission on this object.
5. 对某个Object没有写权限的用户写某个Object时，OSS返回 403 Forbidden消息，错误码：AccessDenied，提示：You do not have write permission on this object.
6. 只有Bucket Owner采用权限调用PutObjectACL来修改该Bucket下某个Object的ACL。非Bucket Owner调用PutObjectACL时，OSS返回 403 Forbidden消息，错误码：AccessDenied，提示：You do not have write acl permission on this object.
7. Object ACL优先级高于Bucket ACL。例如Bucket ACL是private的，而Object ACL是public-read-write的，则访问这个Object时，先判断Object的ACL，所以所有用户都拥有这个Object的访问权限，即使这个Bucket是private bucket。如果某个Object从来没设置过ACL，则访问权限遵循Bucket ACL。

示例

请求示例：

```
PUT /test-object?acl HTTP/1.1
x-oss-object-acl: public-read
Host: ***.regionid.example.com
Date: Wed, 29 Apr 2015 05:21:12 GMT
Authorization: OSS q***c:K***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
Date: Wed, 29 Apr 2015 05:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.7.10 GetObjectACL

Get Object ACL用来获取某个Bucket下的某个Object的访问权限。

请求语法

```
GET /ObjectName?acl HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
```

```
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-41: 响应元素

名称	描述
AccessControlList	存储ACL信息的容器 类型：容器 父节点：AccessControlPolicy
AccessControlPolicy	保存Get Object ACL结果的容器 类型：容器 父节点：None
DisplayName	Bucket拥有者的名称。(目前和ID一致) 类型：字符串 父节点：AccessControlPolicy.Owner
Grant	Object的ACL权限。 类型：枚举字符串 有效值：private，public-read，public-read-write 父节点：AccessControlPolicy.AccessControlList
ID	Bucket拥有者的用户ID 类型：字符串 父节点：AccessControlPolicy.Owner
Owner	保存Bucket拥有者信息的容器。 类型：容器 父节点：AccessControlPolicy

细节分析

1. 只有Bucket的拥有者才能使用GetObjectACL这个接口来获取该Bucket下某个Object的ACL，非Bucket Owner调用该接口时，返回403 Forbidden消息。错误码：AccessDenied，提示You do not have read acl permission on this object.
2. 如果从来没有对某个Object设置过ACL，则调用GetObjectACL时，OSS返回的ObjectACL会是default，表明该Object ACL遵循Bucket ACL。即：如果Bucket是private的，则该object也是private的；如果该object是public-read-write的，则该object也是public-read-write的。

示例

请求示例：

```
GET /test-object?acl HTTP/1.1
Host: ***.regionid.example.com
Date: Wed, 29 Apr 2015 05:21:12 GMT
Authorization: OSS q***c:C***=
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***1
Date: Wed, 29 Apr 2015 05:21:12 GMT
Content-Length: 253
Content-Tupe: application/xml
Connection: keep-alive
Server: AliyunOSS

<?xml version="1.0" ?>
<AccessControlPolicy>
   <Owner>
      <ID>0***2</ID>
      <DisplayName>0***2</DisplayName>
   </Owner>
   <AccessControlList>
      <Grant>public-read </Grant>
   </AccessControlList>
</AccessControlPolicy>
```

1.1.7.11 PostObject

Post Object使用HTML表单上传文件到指定bucket。Post作为Put的替代品，使得基于浏览器上传文件到bucket成为可能。Post Object的消息实体通过多重表单格式（multipart/form-data）编码，在Put Object操作中参数通过HTTP请求头传递，在Post操作中参数则作为消息实体中的表单域传递。

请求语法

```
POST / HTTP/1.1
Host: BucketName.regionid.example.com
User-Agent: browser_data
Content-Length：ContentLength
Content-Type: multipart/form-data; boundary=9431149156168
--9431149156168
Content-Disposition: form-data; name="key"
key
--9431149156168
Content-Disposition: form-data; name="success_action_redirect"
success_redirect
--9431149156168
Content-Disposition: form-data; name="Content-Disposition"
attachment;filename=oss_download.jpg
--9431149156168
Content-Disposition: form-data; name="x-oss-meta-uuid"
myuuid
--9431149156168
```

```
Content-Disposition: form-data; name="x-oss-meta-tag"
mytag
--9431149156168
Content-Disposition: form-data; name="OSSAccessKeyId"
access-key-id
--9431149156168
Content-Disposition: form-data; name="policy"
encoded_policy
--9431149156168
Content-Disposition: form-data; name="Signature"
signature
--9431149156168
Content-Disposition: form-data; name="file"; filename="MyFilename.jpg"
Content-Type: image/jpeg
file_content
--9431149156168
Content-Disposition: form-data; name="submit"
Upload to OSS
--9431149156168--
```

表单域

表 1-42: 表单域

名称	描述	必须
OSSAccessKeyId	Bucket 拥有者的Access Key Id。 类型：字符串 默认值：无 限制：当bucket非public-read-write或者提供了policy（或Signature）表单域时，必须提供该表单域。	有条件的
policy	policy规定了请求的表单域的合法性。不包含policy表单域的请求被认为是匿名请求，并且只能访问public-read-write的bucket。更详细描述请参照5.7.4.1 Post Policy。 类型：字符串 默认值：无 限制：当bucket非public-read-write或者提供了OSSAccessKeyId（或Signature）表单域时，必须提供该表单域。	有条件的
Signature	根据Access Key Secret和policy计算的签名信息，OSS验证该签名信息从而验证该Post请求的合法性。 类型：字符串 默认值：无 限制：当bucket非public-read-write或者提供了OSSAccessKeyId（或policy）表单域时，必须提供该表单域。	有条件的
Cache-Control, Content-Type, Content	REST请求头，更多的信息见Put Object。 类型：字符串 默认值：无	可选

名称	描述	必须
-Disposition, Content-Encoding, Expires		
file	文件或文本内容，必须是表单中的最后一个域。浏览器会自动根据文件类型来设置Content-Type，会覆盖用户的设置。 OSS一次只能上传一个文件。 类型：字符串 默认值：无	必须
key	上传文件的object名称。 如果需要使用用户上传的文件名称作为object名，使用${filename}变量。例如：如果用户上传了文件b.jpg，而key域的值设置为/user/a/${filename}，最终的object名为/user/a/b.jpg。 如果文件名包含路径，则去除文件名中的路径，例如用户上传了文件a/b/c/b.jpg，则取文件名为b.jpg，若key域的值设置为/user/a/${filename}，最终的object名为/user/a/b.jpg 类型：字符串 默认值：无	必须
success_action_redirect	上传成功后客户端跳转到的URL，如果未指定该表单域，返回结果由success_action_status表单域指定。如果上传失败，OSS返回错误码，并不进行跳转。 类型：字符串 默认值：无	可选
success_action_status	未指定success_action_redirect表单域时，该表单域指定了上传成功后返回给客户端的状态码。接受值为200, 201, 204（默认）。 如果该域的值为200或者204，OSS返回一个空文档和相应的状态码。如果该域的值设置为201，OSS返回一个XML文件和201状态码。 如果其值未设置或者设置成一个非法值，OSS返回一个空文档和204状态码。 类型：字符串 默认值：无	可选
x-oss-meta-*	用户指定的user meta值。 OSS不会检查或者使用该值。 类型：字符串 默认值：无	可选
x-oss-server-side-encryption	指定OSS创建object时的服务器端加密编码算法。 类型：字符串 合法值：AES256	可选

名称	描述	必须
x-oss-object-acl	指定oss创建object时的访问权限。 类型：字符串 合法值：public-read，private，public-read-write	可选
x-oss-security-token	若本次访问是使用STS临时授权方式，则需要指定该项为SecurityToken的值，同时OSSAccessKeyId需要使用与之配对的临时AccessKeyId，计算签名时，与使用普通AccessKeyId签名方式一致。 类型：字符串 默认值：无	可选

响应Header

表 1-43: 响应Header

名称	描述
x-oss-server-side-encryption	如果请求指定了x-oss-server-side-encryption熵编码，则响应Header中包含了该头部，指明了所使用的加密算法。类型：字符串

响应元素(Response Elements)

表 1-44: 响应元素

名称	描述
PostResponse	保持Post请求结果的容器。 类型：容器 子节点：Bucket, ETag, Key, Location
Bucket	Bucket名称。 类型：字符串 父节点：PostResponse
ETag	ETag (entity tag) 在每个Object生成的时候被创建，Post请求创建的Object，ETag值是该Object内容的uuid，可以用于检查该Object内容是否发生变化。 类型：字符串 父节点：PostResponse
Location	新创建Object的URL。 类型：字符串

名称	描述
	父节点：PostResponse

细节分析

1. 进行Post操作要求对bucket有写权限，如果bucket为public-read-write，可以不上传签名信息，否则要求对该操作进行签名验证。与Put操作不同，Post操作使用AccessKeySecret对policy进行签名计算出签名字符串作为Signature表单域的值，OSS会验证该值从而判断签名的合法性。
2. 无论bucket是否为public-read-write，一旦上传OSSAccessKeyId, policy, Signature表单域中的任意一个，则另两个表单域为必选项，缺失时OSS会返回错误码：InvalidArgument。
3. post操作提交表单编码必须为“multipart/form-data”，即header中Content-Type为

   ```
   multipart/form-data; boundary=xxxxxx
   ```

   这样的形式，boundary为边界字符串。
4. 提交表单的URL为bucket域名即可，不需要在URL中指定object。即请求行是

   ```
   POST / HTTP/1.1
   ```

   ，不能写成

   ```
   POST /ObjectName HTTP/1.1
   ```
5. policy规定了该次Post请求中表单域的合法值，OSS会根据policy判断请求的合法性，如果不合法会返回错误码：AccessDenied。在检查policy合法性时，policy中不涉及的表单域不进行检查。
6. 表单和policy必须使用UTF-8编码，policy为经过UTF-8编码和base64编码的JSON。
7. Post请求中可以包含额外的表单域，OSS会根据policy对这些表单域检查合法性。
8. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。
9. 如果POST请求中包含Header签名信息或URL签名信息，OSS不会对它们做检查。
10. 如果请求中携带以x-oss-meta-为前缀的表单域，则视为user meta，比如x-oss-meta-location。一个Object可以有多个类似的参数，但所有的user meta总大小不能超过8k。
11. Post请求的body总长度不允许超过5G。若文件长度过大，会返回错误码：EntityTooLarge。
12. 如果上传指定了x-oss-server-side-encryption Header请求域，则必须设置其值为AES256，否则会返回400和错误码：InvalidEncryptionAlgorithmError。指定该Header后，在响应头中也会返回

该Header，OSS会对上传的Object进行加密编码存储，当这个Object被下载时，响应头中会包含x-oss-server-side-encryption，值被设置成该Object的加密算法。

13.表单域为大小写不敏感的，但是表单域的值为大小写敏感的。

示例

请求示例：

```
POST / HTTP/1.1
Host: ***.regionid.example.com
Content-Length: 344606
Content-Type: multipart/form-data; boundary=9431149156168
--9431149156168
Content-Disposition: form-data; name="key"
/user/a/${filename}
--9431149156168
Content-Disposition: form-data; name="success_action_status"
200
--9431149156168
Content-Disposition: form-data; name="Content-Disposition"
content_disposition
--9431149156168
Content-Disposition: form-data; name="x-oss-meta-uuid"
uuid
--9431149156168
Content-Disposition: form-data; name="x-oss-meta-tag"
metadata
--9431149156168
Content-Disposition: form-data; name="OSSAccessKeyId"
44CF9590006BF252F707
--9431149156168
Content-Disposition: form-data; name="policy"
eyJleHBpcmF0aW9uIjoiMjAxMy0xMi0wMVQxMjowMDowMFoiLCJjb25kaXRpb25zIjpbWy
Jjb250ZW50LWxlbmd0aC1yYW5nZSIsIDAsIDEwNDg1NzYwXSx7ImJ1Y2tldCI6ImFoYWhh
In0sIHsiQSI6ICJhIn0seyJrZXkiOiAiQUJDIn1dfQ==
--9431149156168
Content-Disposition: form-data; name="Signature"
k***0+d***k=
--9431149156168
Content-Disposition: form-data; name="file"; filename="MyFilename.txt"
Content-Type: text/plain
abcdefg
--9431149156168
Content-Disposition: form-data; name="submit"
Upload to OSS
--9431149156168--
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 6***e
Date: Fri, 24 Feb 2014 06:03:28 GMT
ETag: 5***E
Connection: keep-alive
Content-Length: 0
```

```
Server: AliyunOSS
```

Post Policy

Post请求的policy表单域用于验证请求的合法性。 policy为一段经过UTF-8和base64编码的JSON文本，声明了Post请求必须满足的条件。虽然对于public-read-write的bucket上传时，post表单域为可选项，我们强烈建议使用该域来限制Post请求。

Policy示例

```
{ "expiration": "2014-12-01T12:00:00.000Z",
 "conditions": [
 {"bucket": "johnsmith" },
 ["starts-with", "$key", "user/eric/"]
 ]
}
```

Post policy中必须包含expiration和condtions。

Expiration

Expiration项指定了policy的过期时间，以ISO8601 GMT时间表示。例如”2014-12-01T12:00:00.000Z”指定了Post请求必须发生在2014年12月1日12点之前。

Conditions

Conditions是一个列表，可以用于指定Post请求的表单域的合法值。注意：表单域对应的值在检查policy之后进行扩展，因此，policy中设置的表单域的合法值应当对应于扩展之前的表单域的值。例如，如果设置key表单域为user/user1/${filename}，用户的文件名为a.txt，则Post policy应当设置成[“eq”，“$key”，“user/user1/${filename}”]，而不是[“eq”，“$key”，“$key”，“user/user1/a.txt”]。Policy中支持的conditions项见下表：

表 1-45: Conditions

名称	描述
content-length-range	上传文件的最小和最大允许大小。该condition支持contion-length-range匹配方式。
Cache-Control, Content-Type, Content-Disposition, Content-Encoding, Expires	HTTP请求头。该condition支持精确匹配和starts-with匹配方式。
key	上传文件的object名称。该condition支持精确匹配和starts-with匹配方式。
success_action_redirect	上传成功后的跳转URL地址。该condition支持精确匹配和starts-with匹配方式。

名称	描述
success_action_status	未指定success_action_redirect时，上传成功后的返回状态码。该condition支持精确匹配和starts-with匹配方式。
x-oss-meta-*	用户指定的user meta。该condition支持精确匹配和starts-with匹配方式。

如果Post请求中包含其他的表单域，可以将这些额外的表单域加入到policy的conditions中，conditions不涉及的表单域将不会进行合法性检查。

Conditions匹配方式

Conditions匹配方式	描述
精确匹配	表单域的值必须精确匹配conditions中声明的值。如指定key表单域的值必须为a：{“key”：“a”} 同样可以写为：[“eq”，“$key”，“a”]
Starts With	表单域的值必须以指定值开始。例如指定key的值必须以/user/user1开始：[“starts-with”，“$key”，“/user/user1”]
指定文件大小	指定所允许上传的文件最大大小和最小大小，例如允许的文件大小为1到10字节：[“content-length-range”，1, 10]

转义字符

于在 Post policy 中 $ 表示变量，所以如果要描述 $，需要使用转义字符\$。除此之外，JSON 将对一些字符进行转义。下图描述了 Post policy 的 JSON 中需要进行转义的字符。

表 1-46: 转义字符

转义字符	描述
\/	斜杠
\	反斜杠
\”	双引号
\$	美元符
\b	空格
\f	换页

转义字符	描述
\n	换行
\r	回车
\t	水平制表符
\uxxxx	Unicode 字符

Post Signature

对于验证的Post请求，HTML表单中必须包含policy和Signature信息。policy控制请求中那些值是允许的。计算Signature的具体流程为：

1. 创建一个 UTF-8 编码的 policy。
2. 将 policy 进行 base64 编码，其值即为 policy 表单域该填入的值，将该值作为将要签名的字符串。
3. 使用 AccessKeySecret 对要签名的字符串进行签名，签名方法与Head中签名的计算方法相同（将要签名的字符串替换为 policy 即可）。

1.1.7.12 PutSymlink

Put Symlink用于创建符号链接。

请求语法

```
PUT /ObjectName?symlink HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
x-oss-symlink-target: TargetObjectName
```

请求Header

表 1-47: 请求Header

名称	描述
x-oss-symlink-target	符号链接指向的目标文件。 类型：字符串 合法值：命名规范同Object

细节分析

1. TargetObjectName同ObjectName一样，需要URL encode。

2. 符号链接的目标文件类型不能为符号链接。
3. 创建符号链接
   - 不检查目标文件是否存在
   - 不检查目标文件类型是否合法
   - 不检查目标文件是否有权限访问

   以上检查，都推迟到GetObject等需要访问目标文件的API。
4. 如果试图添加的文件已经存在，并且有访问权限。新添加的文件将覆盖原来的文件，成功返回200 OK。
5. 如果在PutSymlink的时候，携带以x-oss-meta-为前缀的参数，则视为user meta，比如x-oss-meta-location。一个Object可以有多个类似的参数，但所有的user meta总大小不能超过8k。

示例

请求示例：

```
PUT /link-to-oss?symlink HTTP/1.1
Host: ***.regionid.example.com
Cache-control: no-cache
Content-Disposition: attachment;filename=oss_download.jpg
Date: Tue, 08 Nov 2016 02:00:25 GMT
Authorization: OSS q***:k***=
x-oss-symlink-target: oss.jpg
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
Date: Tue, 08 Nov 2016 02:00:25 GMT
Content-Length: 0
Connection: keep-alive
x-oss-request-id: 5***5
ETag: "0***6"
```

1.1.7.13 GetSymlink

用于获取符号链接，此操作要求用户对该符号链接有读权限。

请求语法

```
GET /ObjectName?symlink HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
```

```
Authorization: SignatureValue
```

响应Header

表 1-48: 响应Header

名称	描述
x-oss-symlink-target	符号链接指向的目标文件。 类型：字符串

细节分析

如果符号链接不存在返回404 Not Found错误。错误码：NoSuchKey

示例

请求示例：

```
GET /link-to-oss.jpg?symlink HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 06:38:30 GMT
Authorization: OSS q***:U***=
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
Date: Fri, 24 Feb 2012 06:38:30 GMT
Last-Modified: Fri, 24 Feb 2012 06:07:48 GMT
Content-Length: 0
Connection: keep-alive
x-oss-request-id: 5***A
x-oss-symlink-target: oss.jpg
ETag: "A***2"
```

1.1.8 关于MultipartUpload的操作

1.1.8.1 简介

除了通过PUT Object接口上传文件到OSS以外，OSS还提供了另外一种上传模式——Multipart Upload。用户可以在如下的应用场景内（但不仅限于此），使用Multipart Upload上传模式，如：

- 需要支持断点上传。
- 上传超过100MB大小的文件。
- 网络条件较差，和OSS的服务器之间的链接经常断开。
- 上传文件之前，无法确定上传文件的大小。

1.1.8.2 InitiateMultipartUpload

使用Multipart Upload模式传输数据前，必须先调用该接口来通知OSS初始化一个Multipart Upload事件。该接口会返回一个OSS服务器创建的全局唯一的Upload ID，用于标识本次Multipart Upload事件。用户可以根据这个ID来发起相关的操作，如中止Multipart Upload、查询Multipart Upload等。

请求语法

```
POST /ObjectName?uploads HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT date
Authorization: SignatureValue
```

请求参数(Request Parameters)

Initiate Multipart Upload时，可以通过encoding-type对返回结果中的Key进行编码。

表 1-49: 请求参数

名称	描述
encoding-type	指定对返回的Key进行编码，目前支持url编码。Key使用UTF-8字符，但xml 1.0标准不支持解析一些控制字符，比如ascii值从0到10的字符。对于Key中包含xml 1.0标准不支持的控制字符，可以通过指定encoding-type对返回的Key进行编码。 数据类型：字符串 默认值：无 可选值：url

表 1-50: 请求Header

名称	描述
Cache-Control	指定该Object被下载时的网页的缓存行为；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-Disposition	指定该Object被下载时的名称；更详细描述请参照*RFC2616*。 类型：字符串 默认值：无
Content-Encoding	指定该Object被下载时的内容编码格式；更详细描述请参照*RFC2616*。

名称	描述
	类型：字符串 默认值：无
Expires	过期时间（milliseconds）；更详细描述请参照*RFC2616*。 类型：整数 默认值：无
x-oss-server-side-encryption	指定上传该Object每个part时使用的服务器端加密编码算法，OSS会对上传的每个part采用服务器端加密编码进行存储。 类型：字符串 合法值：AES256

响应元素(Response Elements)

表 1-51: 响应元素

名称	描述
Bucket	初始化一个Multipart Upload事件的Bucket名称。 类型：字符串 父节点：InitiateMultipartUploadResult
InitiateMultipartUploadResult	保存Initiate Multipart Upload请求结果的容器。 类型：容器 子节点：Bucket, Key, UploadId 父节点：None
Key	初始化一个Multipart Upload事件的Object名称。 类型：字符串 父节点：InitiateMultipartUploadResult
UploadId	唯一标示此次Multipart Upload事件的ID。 类型：字符串 父节点：InitiateMultipartUploadResult
EncodingType	指明返回结果中编码使用的类型。如果请求的参数中指定了encoding-type，那返回的结果会对Key进行编码。 类型：字符串 父节点：容器

细节分析

1. 该操作计算认证签名的时候，需要加“?uploads”到CanonicalizedResource中。
2. 初始化Multipart Upload请求，支持如下标准的HTTP请求头：Cache-Control，Content-Disposition，Content-Encoding，Content-Type，Expires，以及以“x-oss-meta-”开头的用户自定义Headers。具体含义请参见PUT Object接口。
3. 初始化Multipart Upload请求，并不会影响已经存在的同名object。
4. 服务器收到初始化Multipart Upload请求后，会返回一个XML格式的请求体。该请求体内有三个元素：Bucket，Key和UploadID。请记录下其中的UploadID，以用于后续的Multipart相关操作。
5. 初始化Multipart Upload请求时，若设置了x-oss-server-side-encryption Header，则在响应头中会返回该Header，并且在上传的每个part时，服务端会自动对每个part进行熵编码加密存储，目前OSS服务器端只支持256位高级加密标准（AES256），指定其他值会返回400和相应的错误提示：InvalidEncryptionAlgorithmError；在上传每个part时不必再添加x-oss-server-side-encryption 请求头，若指定该请求头则OSS会返回400和相应的错误提示：InvalidArgument。

示例

请求示例：

```
POST /multipart.data?uploads HTTP/1.1
Host: ***.regionid.example.com
Date: Wed, 22 Feb 2012 08:32:21 GMT
Authorization: OSS q***4=
```

返回示例：

```
HTTP/1.1 200 OK
Content-Length: 230
Server: AliyunOSS
Connection: keep-alive
x-oss-request-id: 4***8
Date: Wed, 22 Feb 2012 08:32:21 GMT
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<InitiateMultipartUploadResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
   <Bucket> multipart_upload</Bucket>
   <Key>multipart.data</Key>
   <UploadId>0***D</UploadId>
</InitiateMultipartUploadResult>
```

1.1.8.3 UploadPart

初始化一个Multipart Upload之后，可以根据指定的Object名和Upload ID来分块（Part）上传数据。每一个上传的Part都有一个标识它的号码（part number，范围是1~10,000）。对于同一个Upload

ID，该号码不但唯一标识这一块数据，也标识了这块数据在整个文件内的相对位置。如果你用同一个part号码，上传了新的数据，那么OSS上已有的这个号码的Part数据将被覆盖。除了最后一块Part以外，其他的part最小为100KB；最后一块Part没有大小限制。

请求语法

```
PUT /ObjectName?partNumber=PartNumber&uploadId=UploadId HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Content-Length: Size
Authorization: SignatureValue
```

细节分析

1. 调用该接口上传Part数据前，必须调用Initiate Multipart Upload接口，获取一个OSS服务器颁发的Upload ID。
2. Multipart Upload要求除最后一个Part以外，其他的Part大小都要大于100KB。但是Upload Part接口并不会立即校验上传Part的大小（因为不知道是否为最后一块）；只有当Complete Multipart Upload的时候才会校验。
3. OSS会将服务器端收到Part数据的MD5值放在ETag头内返回给用户。
4. Part号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
5. 若调用Initiate Multipart Upload接口时，指定了x-oss-server-side-encryption请求头，则会对上传的Part进行加密编码，并在Upload Part响应头中返回x-oss-server-side-encryption头，其值表明该Part的服务器端加密算法，具体见Initiate Multipart Upload接口。6.为了保证数据在网络传输过程中不出现错误，用户发送请求时携带Content-MD5，OSS会计算上传数据的MD5与用户上传的MD5值比较，如果不一致返回InvalidDigest错误码。

示例

请求示例：

```
PUT /multipart.data?partNumber=1&uploadId=0***6 HTTP/1.1
Host: ***.regionid.example.com
Content-Length：6291456
Date: Wed, 22 Feb 2012 08:32:21 GMT
Authorization: OSS q***I=
[6291456 bytes data]
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
Connection: keep-alive
ETag: 7***9
x-oss-request-id: 3***a
```

```
Date: Wed, 22 Feb 2012 08:32:21 GMT
```

1.1.8.4 UploadPartCopy

Upload Part Copy通过从一个已存在的Object中拷贝数据来上传一个Part。通过在Upload Part请求的基础上增加一个Header:x-oss-copy-source来调用该接口。当拷贝一个大于1GB的文件时，必须使用Upload Part Copy的方式进行拷贝。如果想通过单个操作拷贝小于1GB的文件，可以参考Copy Object。

请求语法

```
PUT /ObjectName? partNumber=PartNumber&uploadId=UploadId HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Content-Length: Size
Authorization: SignatureValue
x-oss-copy-source: /SourceBucketName/SourceObjectName
x-oss-copy-source-range:bytes=first-last
```

请求Header

除了通用的请求Header，Upload Part Copy请求中通过下述Header指定拷贝的源Object地址和拷贝的范围。

表 1-52: 请求Header

名称	描述
x-oss-copy-source	复制源地址（必须有可读权限） 类型：字符串 默认值：无
x-oss-copy-source-range	源Object的拷贝范围。如，设定 bytes=0-9，表示传送第0到第9这10个字符。当拷贝整个源Object时不需要该请求Header。 类型：整型 默认值：无

下述请求Header作用于x-oss-copy-source指定的源Object。

表 1-53: 请求Header

名称	描述
x-oss-copy-source-if-match	如果源Object的ETAG值和用户提供的ETAG相等，则执行拷贝操作；否则返回412 HTTP错误码（预处理失败）。 类型：字符串

名称	描述
	默认值：无
x-oss-copy-source-if-none-match	如果源Object自从用户指定的时间以后就没有被修改过，则执行拷贝操作；否则返回412 HTTP错误码（预处理失败）。 类型：字符串 默认值：无
x-oss-copy-source-if-unmodified-since	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件，并返回200 OK；否则返回412 precondition failed错误。 类型：字符串 默认值：无
x-oss-copy-source-if-modified-since	如果源Object自从用户指定的时间以后被修改过，则执行拷贝操作；否则返回412 HTTP错误码（预处理失败）。 类型：字符串 默认值：无

响应元素(Response Elements)

表 1-54: 响应元素

名称	描述
x-oss-copy-source-if-match	如果源Object的ETAG值和用户提供的ETAG相等，则执行拷贝操作；否则返回412 HTTP错误码（预处理失败）。 类型：字符串 默认值：无
x-oss-copy-source-if-none-match	如果源Object自从用户指定的时间以后就没有被修改过，则执行拷贝操作；否则返回412 HTTP错误码（预处理失败）。 类型：字符串 默认值：无
x-oss-copy-source-if-unmodified-since	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件，并返回200 OK；否则返回412 precondition failed错误。 类型：字符串 默认值：无
x-oss-copy-source-if-modified-since	如果源Object自从用户指定的时间以后被修改过，则执行拷贝操作；否则返回412 HTTP错误码（预处理失败）。 类型：字符串 默认值：无

细节分析

1. 调用该接口上传Part数据前，必须调用Initiate Multipart Upload接口，获取一个OSS服务器颁发的Upload ID。
2. Multipart Upload要求除最后一个Part以外，其他的Part大小都要大于100KB。但是Upload Part接口并不会立即校验上传Part的大小（因为不知道是否为最后一块）；只有当Complete Multipart Upload的时候才会校验。
3. 不指定x-oss-copy-source-range请求头时，表示拷贝整个源Object。当指定该请求头时，则返回消息中会包含整个文件的长度和此次拷贝的范围，例如：Content-Range: bytes 0-9/44，表示整个文件长度为44，此次拷贝的范围为0-9。当指定的范围不符合范围规范时，则拷贝整个文件，并且不在结果中提及Content-Range。
4. 若调用Initiate Multipart Upload接口时，指定了x-oss-server-side-encryption请求头，则会对上传的Part进行加密编码，并在Upload Part响应头中返回x-oss-server-side-encryption头，其值表明该Part的服务器端加密算法，具体见Initiate Multipart Upload接口。
5. 该操作不能拷贝通过Append追加上传方式产生的object。

示例

请求示例：

```
PUT /multipart.data?partNumber=1&uploadId=0***6 HTTP/1.1
Host: ***.regionid.example.com
Content-Length：6291456
Date: Wed, 22 Feb 2012 08:32:21 GMT
Authorization: OSS q***l=
x-oss-copy-source: /oss-example/ src-object
x-oss-copy-source-range:bytes=100-6291756
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
Connection: keep-alive
x-oss-request-id: 3***a
Date: Thu, 17 Jul 2014 06:27:54 GMT'
<?xml version="1.0" encoding="UTF-8"?>
<CopyPartResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
 <LastModified>2014-07-17T06:27:54.000Z </LastModified>
 <ETag>"5***E"</ETag>
</CopyPartResult>
```

1.1.8.5 CompleteMultipartUpload

在将所有数据Part都上传完成后，必须调用Complete Multipart Upload API来完成整个文件的Multipart Upload。在执行该操作时，用户必须提供所有有效的数据Part的列表（包括part号码和

ETAG）；OSS收到用户提交的Part列表后，会逐一验证每个数据Part的有效性。当所有的数据Part验证通过后，OSS将把这些数据part组合成一个完整的Object。

请求语法

```
POST /ObjectName?uploadId=UploadId HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Content-Length: Size
Authorization: Signature

<CompleteMultipartUpload>
<Part>
<PartNumber>PartNumber</PartNumber>
<ETag>ETag</ETag>
</Part>
...
</CompleteMultipartUpload>
```

请求参数(Request Parameters)

Complete Multipart Upload时，可以通过encoding-type对返回结果中的Key进行编码。

表 1-55: 请求参数

名称	描述
encoding-type	指定对返回的Key进行编码，目前支持url编码。Key使用UTF-8字符，但xml 1.0标准不支持解析一些控制字符，比如ascii值从0到10的字符。对于Key中包含xml 1.0标准不支持的控制字符，可以通过指定encoding-type对返回的Key进行编码。 数据类型：字符串 默认值：无 可选值：url

请求元素(Request Elements)

表 1-56: 请求元素

名称	描述
CompleteMultipartUpload	保存Complete Multipart Upload请求内容的容器。 类型：容器 子节点：一个或多个Part元素 父节点：无
ETag	Part成功上传后，OSS返回的ETag值。 类型：字符串

名称	描述
	父节点：Part
Part	保存已经上传Part信息的容器。 类型：容器 子节点：ETag, PartNumber 父节点：InitiateMultipartUploadResult
PartNumber	Part数目。 类型：整数 父节点：Part

响应元素(Response Elements)

表 1-57: 响应元素

名称	描述
Bucket	Bucket名称。 类型：字符串 父节点：CompleteMultipartUploadResult
CompleteMultipartUploadResult	保存Complete Multipart Upload请求结果的容器。 类型：容器 子节点：Bucket, Key, ETag, Location 父节点：None
ETag	ETag (entity tag) 在每个Object生成的时候被创建，用于标示一个Object的内容。Complete Multipart Upload请求创建的Object，ETag值是其内容的UUID。ETag值可以用于检查Object内容是否发生变化。 类型：字符串 父节点：CompleteMultipartUploadResult
Location	新创建Object的URL。 类型：字符串 父节点：CompleteMultipartUploadResult
Key	新创建Object的名字。 类型：字符串 父节点：CompleteMultipartUploadResult
EncodingType	指明返回结果中编码使用的类型。如果请求的参数中指定了encoding-type，那返回的结果会对Key进行编码。 类型：字符串

名称	描述
	父节点：容器

细节分析

1. Complete Multipart Upload时，会确认除最后一块以外所有块的大小都大于100KB，并检查用户提交的Partlist中的每一个Part号码和Etag。所以在上传Part时，客户端除了需要记录Part号码外，还需要记录每次上传Part成功后，服务器返回的ETag值。
2. OSS处理Complete Multipart Upload请求时，会持续一定的时间。在这段时间内，如果客户端和OSS之间的链接断掉，OSS仍会继续将请求做完。
3. 用户提交的Part List中，Part号码可以是不连续的。例如第一块的Part号码是1；第二块的Part号码是5。
4. OSS处理Complete Multipart Upload请求成功后，该Upload ID就会变成无效。
5. 同一个Object可以同时拥有不同的Upload Id，当Complete一个Upload ID后，该Object的其他Upload ID不受影响。
6. 若调用Initiate Multipart Upload接口时，指定了x-oss-server-side-encryption请求头，则在Complete Multipart Upload的响应头中，会返回x-oss-server-side-encryption，其值表明该Object的服务器端加密算法。
7. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。

示例

请求示例：

```
POST /multipart.data? uploadId=0***4  HTTP/1.1
Host: ***.regionid.example.com
Content-Length: 1056
Date: Fri, 24 Feb 2012 10:19:18 GMT
Authorization: OSS q***0=

<CompleteMultipartUpload>
   <Part>
      <PartNumber>1</PartNumber>
      <ETag>"3***9"</ETag>
   </Part>
   <Part>
      <PartNumber>5</PartNumber>
      <ETag>"8***A"</ETag>
   </Part>
   <Part>
      <PartNumber>8</PartNumber>
      <ETag>"8***2"</ETag>
   </Part>
```

```
</CompleteMultipartUpload>
```

返回示例：

```
HTTP/1.1 200 OK
Server: AliyunOSS
Content-Length: 329
Content-Type: Application/xml
Connection: keep-alive
x-oss-request-id: 5***e
Date: Fri, 24 Feb 2012 10:19:18 GMT

<?xml version="1.0" encoding="UTF-8"?>
<CompleteMultipartUploadResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
   <Location>http://***.regionid.example.com /multipart.data</Location>
   <Bucket>oss-example</Bucket>
   <Key>multipart.data</Key>
   <ETag>B***D-3</ETag>
</CompleteMultipartUploadResult>
```

1.1.8.6 AbortMultipartUpload

该接口可以根据用户提供的Upload ID中止其对应的Multipart Upload事件。当一个Multipart Upload事件被中止后，就不能再使用这个Upload ID做任何操作，已经上传的Part数据也会被删除。

请求语法

```
DELETE /ObjectName?uploadId=UploadId HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: Signature
```

细节分析

1. 中止一个Multipart Upload事件时，如果其所属的某些Part仍然在上传，那么这次中止操作将无法删除这些Part。所以如果存在并发访问的情况，为了彻底释放OSS上的空间，需要调用几次Abort Multipart Upload接口。
2. 如果输入的Upload Id不存在，OSS会返回404错误，错误码为：NoSuchUpload。

示例

请求示例：

```
Delete /multipart.data?&uploadId=0***E  HTTP/1.1
Host: ***.regionid.example.com
Date: Wed, 22 Feb 2012 08:32:21 GMT
Authorization: OSS q***l=
```

返回示例：

```
HTTP/1.1 204
Server: AliyunOSS
```

```
Connection: keep-alive
x-oss-request-id: 0***d
Date: Wed, 22 Feb 2012 08:32:21 GMT
```

1.1.8.7 ListMultipartUploads

List Multipart Uploads可以罗列出所有执行中的Multipart Upload事件，即已经被初始化的Multipart Upload但是未被Complete或者Abort的Multipart Upload事件。OSS返回的罗列结果中最多会包含1000个Multipart Upload信息。如果想指定OSS返回罗列结果内Multipart Upload信息的数目，可以在请求中添加max-uploads参数。另外，OSS返回罗列结果中的IsTruncated元素标明是否还有其他的Multipart Upload。

请求语法

```
Get /?uploads HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: Signature
```

请求参数(Request Parameters)

表 1-58: 请求参数

名称	描述
delimiter	是一个用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素——CommonPrefixes。 类型：字符串
max-uploads	限定此次返回Multipart Uploads事件的最大数目，如果不设定，默认为1000，max-uploads取值不能大于1000。 类型：字符串
key-marker	与upload-id-marker参数一同使用来指定返回结果的起始位置。 l 如果upload-id-marker参数未设置，查询结果中包含：所有Object名字的字典序大于key-marker参数值的Multipart事件。 l 如果upload-id-marker参数被设置，查询结果中包含：所有Object名字的字典序大于key-marker参数值的Multipart事件和Object名字等于key-marker参数值，但是Upload ID比upload-id-marker参数值大的Multipart Uploads事件。 类型：字符串
prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix。 类型：字符串

名称	描述
upload-id-marker	与key-marker参数一同使用来指定返回结果的起始位置。 l 如果key-marker参数未设置，则OSS忽略upload-id-marker参数。 l 如果key-marker参数被设置，查询结果中包含：所有Object名字的字典序大于key-marker参数值的Multipart事件和Object名字等于key-marker参数值，但是Upload ID比upload-id-marker参数值大的Multipart Uploads事件。 类型：字符串
encoding-type	指定对返回的内容进行编码，指定编码的类型。Delimiter、KeyMarker、Prefix、NextKeyMarker和Key使用UTF-8字符，但xml 1.0标准不支持解析一些控制字符，比如ascii值从0到10的字符。对于包含xml 1.0标准不支持的控制字符，可以通过指定encoding-type对返回的Delimiter、KeyMarker、Prefix、NextKeyMarker和Key进行编码。 数据类型：字符串 默认值：无

响应元素(Response Elements)

表 1-59: 响应元素

名称	描述
ListMultipartUploadsResult	保存List Multipart Upload请求结果的容器。 类型：容器 子节点：Bucket, KeyMarker, UploadIdMarker, NextKeyMarker, NextUploadIdMarker, MasUploads, Delimiter, Prefix, CommonPrefixes, IsTruncated, Upload 父节点：None
Bucket	Bucket名称。 类型：字符串 父节点：ListMultipartUploadsResult
EncodingType	指明返回结果中编码使用的类型。如果请求的参数中指定了encoding-type，那返回的结果会对Delimiter、KeyMarker、Prefix、NextKeyMarker和Key这些元素进行编码。 类型：字符串 父节点：ListMultipartUploadsResult
KeyMarker	列表的起始Object位置。 类型：字符串 父节点：ListMultipartUploadsResult

名称	描述
UploadIdMarker	列表的起始UploadID位置。 类型：字符串 父节点：ListMultipartUploadsResult
NextKeyMarker	如果本次没有返回全部结果，响应请求中将包含NextKeyMarker元素，用于标明接下来请求的KeyMarker值。 类型：字符串 父节点：ListMultipartUploadsResult
NextUploadMarker	如果本次没有返回全部结果，响应请求中将包含NextUploadMarker元素，用于标明接下来请求的UploadMarker值。 类型：字符串 父节点：ListMultipartUploadsResult
MaxUploads	返回的最大Upload数目。 类型：整数 父节点：ListMultipartUploadsResult
IsTruncated	标明是否本次返回的Multipart Upload结果列表被截断。“true”表示本次没有返回全部结果；“false”表示本次已经返回了全部结果。 类型：枚举字符串 有效值：false、true 默认值：false 父节点：ListMultipartUploadsResult
Upload	保存Multipart Upload事件信息的容器。 类型：容器 子节点：Key, UploadId, Initiated 父节点：ListMultipartUploadsResult
Key	初始化Multipart Upload事件的Object名字。 类型：字符串 父节点：Upload
UploadId	Multipart Upload事件的ID。 类型：字符串 父节点：Upload
Initiated	Multipart Upload事件初始化的时间。 类型：日期 父节点：Upload

细节分析

1. “max-uploads”参数最大值为1000。
2. 在OSS的返回结果首先按照Object名字字典序升序排列；对于同一个Object，则按照时间序，升序排列。
3. 可以灵活地使用prefix参数对bucket内的object进行分组管理（类似与文件夹的功能）。
4. List Multipart Uploads请求支持5种请求参数： prefix，marker，delimiter，upload-id-marker和max-uploads。通过这些参数的组合，可以设定查询Multipart Uploads事件的规则，获得期望的查询结果。

示例

请求示例：

```
Get /?uploads  HTTP/1.1
Host:***.regionid.example.com
Date: Thu, 23 Feb 2012 06:14:27 GMT
Authorization: OSS q***Y=
```

返回示例：

```
HTTP/1.1 200
Server: AliyunOSS
Connection: keep-alive
Content-length: 1839
Content-type: application/xml
x-oss-request-id: 5***a
Date: Thu, 23 Feb 2012 06:14:27 GMT

<?xml version="1.0" encoding="UTF-8"?>
<ListMultipartUploadsResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
   <Bucket>oss-example</Bucket>
   <KeyMarker></KeyMarker>
   <UploadIdMarker></UploadIdMarker>
   <NextKeyMarker>oss.avi</NextKeyMarker>
   <NextUploadIdMarker>0***F</NextUploadIdMarker>
   <Delimiter></Delimiter>
   <Prefix></Prefix>
   <MaxUploads>1000</MaxUploads>
   <IsTruncated>false</IsTruncated>
   <Upload>
      <Key>multipart.data</Key>
      <UploadId>0***8</UploadId>
      <Initiated>2012-02-23T04:18:23.000Z</Initiated>
   </Upload>
   <Upload>
      <Key>multipart.data</Key>
      <UploadId>0***5</UploadId>
      <Initiated>2012-02-23T04:18:23.000Z</Initiated>
   </Upload>
   <Upload>
      <Key>oss.avi</Key>
      <UploadId>0***F</UploadId>
```

```
        <Initiated>2012-02-23T06:14:27.000Z</Initiated>
    </Upload>
</ListMultipartUploadsResult>
```

1.1.8.8 ListParts

List Parts命令可以罗列出指定Upload ID所属的所有已经上传成功Part。

请求语法

```
Get  /ObjectName?uploadId=UploadId HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: Signature
```

请求参数(Request Parameters)

表 1-60: 请求参数

名称	描述
uploadId	Multipart Upload事件的ID。 类型：字符串 默认值：无
max-parts	规定在OSS响应中的最大Part数目。 类型：整数 默认值：1,000
part-number-marker	指定List的起始位置，只有Part Number数目大于该参数的Part会被列出。 类型：整数 默认值：无
encoding-type	指定对返回的内容进行编码，指定编码的类型。Key使用UTF-8字符，但xml 1.0标准不支持解析一些控制字符，比如ascii值从0到10的字符。对于Key中包含xml 1.0标准不支持的控制字符，可以通过指定encoding-type对返回的Key进行编码。 数据类型：字符串 默认值：无 可选值：url

响应元素(Response Elements)

表 1-61: 响应元素

名称	描述
ListPartsResult	保存List Part请求结果的容器。 类型：容器 子节点：Bucket, Key, UploadId, PartNumberMarker, NextPartNumberMarker, MaxParts, IsTruncated, Part 父节点：无
Bucket	Bucket名称。 类型：字符串 父节点：ListPartsResult
EncodingType	指明对返回结果进行编码使用的类型。如果请求的参数中指定了encoding-type，那会对返回结果中的Key进行编码。 类型：字符串 父节点：ListPartsResult
Key	Object名称。 类型：字符串 父节点：ListPartsResult
UploadId	Upload事件ID。 类型：字符串 父节点：ListPartsResult
PartNumberMarker	本次List结果的Part Number起始位置。 类型：整数 父节点：ListPartsResult
NextPartNumberMarker	如果本次没有返回全部结果，响应请求中将包含NextPartNumberMarker元素，用于标明接下来请求的PartNumberMarker值。 类型：整数 父节点：ListPartsResult
MaxParts	返回请求中最大的Part数目。 类型：整数 父节点：ListPartsResult
IsTruncated	标明是否本次返回的List Part结果列表被截断。“true”表示本次没有返回全部结果；“false”表示本次已经返回了全部结果。 类型：枚举字符串

名称	描述
	有效值：true、false 父节点：ListPartsResult
Part	保存Part信息的容器。 类型：字符串 子节点：PartNumber，LastModified，ETag， Size 父节点：ListPartsResult
PartNumber	标示Part的数字。 类型：整数 父节点：ListPartsResult.Part
LastModified	Part上传的时间。 类型：日期 父节点：ListPartsResult.part
ETag	已上传Part内容的ETag。 类型：字符串 父节点：ListPartsResult.Part
Size	已上传Part大小。 类型：整数 父节点：ListPartsResult.Part

细节分析

1. List Parts支持max-parts和part-number-marker两种请求参数。
2. max-parts参数最大值为1000；默认值也为1000。
3. 在OSS的返回结果按照Part号码升序排列。
4. 由于网络传输可能出错，所以不推荐用List Part出来的结果（Part Number和ETag值）来生成最后Complete Multipart的Part列表。

示例

请求示例：

```
Get  /multipart.data?uploadId=0***5  HTTP/1.1
Host: ***.regionid.example.com
Date: Thu, 23 Feb 2012 07:13:28 GMT
Authorization: OSS q***8=
```

返回示例：

```
HTTP/1.1 200
Server: AliyunOSS
```

```
Connection: keep-alive
Content-length: 1221
Content-type: application/xml
x-oss-request-id: 1***c
Date: Thu, 23 Feb 2012 07:13:28 GMT

<?xml version="1.0" encoding="UTF-8"?>
<ListPartsResult xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
   <Bucket>multipart_upload</Bucket>
   <Key>multipart.data</Key>
   <UploadId>0***5</UploadId>
   <NextPartNumberMarker>5</NextPartNumberMarker>
   <MaxParts>1000</MaxParts>
   <IsTruncated>false</IsTruncated>
   <Part>
      <PartNumber>1</PartNumber>
      <LastModified>2012-02-23T07:01:34.000Z</LastModified>
      <ETag>&quot;3***9&quot;</ETag>
      <Size>6291456</Size>
   </Part>
   <Part>
      <PartNumber>2</PartNumber>
      <LastModified>2012-02-23T07:01:12.000Z</LastModified>
      <ETag>&quot;3***9&quot;</ETag>
      <Size>6291456</Size>
   </Part>
   <Part>
      <PartNumber>5</PartNumber>
      <LastModified>2012-02-23T07:02:03.000Z</LastModified>
      <ETag>&quot;7***9&quot;</ETag>
      <Size>1024</Size>
   </Part>
</ListPartsResult>
```

1.1.9 跨域资源共享

1.1.9.1 简介

跨域资源共享(CORS)允许WEB端的应用程序访问不属于本域的资源。OSS提供了CORS支持以方便利用OSS开发更灵活的WEB应用程序。OSS提供接口方便开发者控制跨域访问的各种权限。

1.1.9.2 PutBucketcors

Put Bucket cors操作将在指定的bucket上设定一个跨域资源共享(CORS)的规则，如果原规则存在则覆盖原规则。

请求语法

```
PUT /?cors HTTP/1.1
Date: GMT Date
Content-Length：ContentLength
Content-Type: application/xml
Host: BucketName.regionid.example.com
Authorization: SignatureValue

<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration>
```

```
    <CORSRule>
      <AllowedOrigin>the origin you want allow CORS request from</AllowedOrigin>
      <AllowedOrigin>...</AllowedOrigin>
      <AllowedMethod>HTTP method</AllowedMethod>
      <AllowedMethod>...</AllowedMethod>
        <AllowedHeader> headers that allowed browser to send</AllowedHeader>
          <AllowedHeader>...</AllowedHeader>
          <ExposeHeader> headers in response that can access from client app</ExposeHeader>
          <ExposeHeader>...</ExposeHeader>
          <MaxAgeSeconds>time to cache pre-fight response</MaxAgeSeconds>
    </CORSRule>
    <CORSRule>
      ...
    </CORSRule>
...
</CORSConfiguration >
```

请求元素（Request Elements）

表 1-62: 请求元素

名称	描述	是否必须
CORSRule	CORS规则的容器，每个bucket最多允许10条规则 类型：容器 父节点：CORSConfiguration	是
AllowedOrigin	指定允许的跨域请求的来源，允许使用多个元素来指定多个允许的来源。允许使用最多一个“*”通配符。如果指定为“*”则表示允许所有的来源的跨域请求。 类型：字符串 父节点：CORSRule	是
AllowedMethod	指定允许的跨域请求方法。 类型：枚举（GET,PUT,DELETE,POST,HEAD） 父节点：CORSRule	是
AllowedHeader	控制在OPTIONS预取指令中Access-Control-Request-Headers头中指定的header是否允许。在Access-Control-Request-Headers中指定的每个header都必须在AllowedHeader中有一条对应的项。允许使用最多一个“*”通配符 类型：字符串 父节点：CORSRule	否
ExposeHeader	指定允许用户从应用程序中访问的响应头（例如一个Javascript的XMLHttpRequest对象。）不允许使用“*”通配符。 类型：字符串 父节点：CORSRule	否

名称	描述	是否必须
MaxAgeSeconds	指定浏览器对特定资源的预取（OPTIONS）请求返回结果的缓存时间，单位为秒。一个CORSRule里面最多允许出现一个。 类型：整型 父节点：CORSRule	否
CORSConfiguration	Bucket的CORS规则容器 类型：容器 父节点：无	是

细节分析

1. 默认bucket是不开启CORS功能，所有的跨域请求的origin都不被允许。
2. 为了在应用程序中使用CORS功能，比如从一个www.a.com的网址通过浏览器的XMLHttpRequest功能来访问OSS，需要通过本接口手动上传CORS规则来开启。该规则由XML文档来描述。
3. 每个bucket的CORS设定是由多条CORS规则指定的，每个bucket最多允许10条规则，上传的XML文档最多允许16KB大小。
4. 当OSS收到一个跨域请求（或者OPTIONS请求），会读取bucket对应的CORS规则，然后进行相应的权限检查。OSS会依次检查每一条规则，使用第一条匹配的规则来允许请求并返回对应的header。如果所有规则都匹配失败则不附加任何CORS相关的header。
5. CORS规则匹配成功必须满足三个条件，首先，请求的Origin必须匹配一项AllowedOrigin项，其次，请求的方法（如GET，PUT等）或者OPTIONS请求的Access-Control-Request-Method头对应的方法必须匹配一项AllowedMethod项，最后，OPTIONS请求的Access-Control-Request-Headers头包含的每个header都必须匹配一项AllowedHeader项。
6. 如果用户上传了Content-MD5请求头，OSS会计算body的Content-MD5并检查一致性，如果不一致，将返回InvalidDigest错误码。

示例

添加bucket跨域访问请求规则示例：

```
PUT /?cors HTTP/1.1
Host: ***.regionid.example.com
Content-Length: 186
Date: Fri, 04 May 2012 03:21:12 GMT
Authorization: OSS q***A=

<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration>
   <CORSRule>
     <AllowedOrigin>*</AllowedOrigin>
     <AllowedMethod>PUT</AllowedMethod>
```

```
    <AllowedMethod>GET</AllowedMethod>
    <AllowedHeader>Authorization</AllowedHeader>
  </CORSRule>
  <CORSRule>
    <AllowedOrigin>http://www.a.com</AllowedOrigin>
    <AllowedOrigin>http://www.b.com</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedHeader> Authorization</AllowedHeader>
    <ExposeHeader>x-oss-test</ExposeHeader>
    <ExposeHeader>x-oss-test1</ExposeHeader>
    <MaxAgeSeconds>100</MaxAgeSeconds>
  </CORSRule>
</CORSConfiguration >
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***F
Date: Fri, 04 May 2012 03:21:12 GMT
Content-Length: 0
Connection: keep-alive
Server: AliyunOSS
```

1.1.9.3 GetBucketcors

Get Bucket cors操作用于获取指定的Bucket目前的CORS规则。

请求语法

```
GET /?cors HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

响应元素(Response Elements)

表 1-63: 响应元素

名称	描述
CORSRule	CORS规则的容器，每个bucket最多允许10条规则 类型：容器 父节点：CORSConfiguration
AllowedOrigin	指定允许的跨域请求的来源，允许使用多个元素来指定多个允许的来源。允许使用最多一个“*”通配符。如果指定为“*”则表示允许所有的来源的跨域请求。 类型：字符串 父节点：CORSRule
AllowedMethod	指定允许的跨域请求方法。 类型：枚举（GET,PUT,DELETE,POST,HEAD） 父节点：CORSRule

名称	描述
AllowedHeader	控制在OPTIONS预取指令中Access-Control-Request-Headers头中指定的header是否允许。在Access-Control-Request-Headers中指定的每个header都必须在AllowedHeader中有一条对应的项。允许使用最多一个“*”通配符 类型：字符串 父节点：CORSRule
ExposeHeader	指定允许用户从应用程序中访问的响应头（例如一个Javascript的XMLHttpRequest对象。不允许使用“*”通配符。 类型：字符串 父节点：CORSRule
MaxAgeSeconds	指定浏览器对特定资源的预取（OPTIONS）请求返回结果的缓存时间，单位为秒。一个CORSRule里面最多允许出现一个。 类型：整型 父节点：CORSRule
CORSConfiguration	Bucket的CORS规则容器 类型：容器 父节点：无

细节分析

1. 如果Bucket不存在，返回404 no content错误。错误码：NoSuchBucket。
2. 只有Bucket的拥有者才能获取CORS规则，否则返回403 Forbidden错误，错误码：AccessDenied。
3. 如果CORS规则不存在，返回404 Not Found错误，错误码NoSuchCORSConfiguration。

示例

请求示例：

```
Get /?cors HTTP/1.1
Host: ***.regionid.example.com
Date: Thu, 13 Sep 2012 07:51:28 GMT
Authorization: OSS q***w=
```

已设置CORS规则的返回示例：

```
HTTP/1.1 200
x-oss-request-id: 5***F
Date: Thu, 13 Sep 2012 07:51:28 GMT
Connection: keep-alive
Content-Length: 218
Server: AliyunOSS

<?xml version="1.0" encoding="UTF-8"?>
```

```
<CORSConfiguration>
  <CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
    <ExposeHeader>x-oss-test</ExposeHeader>
    <MaxAgeSeconds>100</MaxAgeSeconds>
  </CORSRule>
</CORSConfiguration>
```

1.1.9.4 DeleteBucketcors

Delete Bucket cors用于关闭指定Bucket对应的CORS功能并清空所有规则。

请求语法

```
DELETE /?cors HTTP/1.1
Host: BucketName.regionid.example.com
Date: GMT Date
Authorization: SignatureValue
```

细节分析

1. 如果Bucket不存在，返回404 no content错误，错误码：NoSuchBucket。
2. 只有Bucket的拥有者才能删除Bucket对应的CORS规则。如果试图操作一个不属于你的Bucket，OSS返回403 Forbidden错误，错误码：AccessDenied。

示例

请求示例：

```
DELETE /?cors HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 05:45:34 GMT
Authorization: OSS q***g=
```

返回示例：

```
HTTP/1.1 204 No Content
x-oss-request-id: 5***C
Date: Fri, 24 Feb 2012 05:45:34 GMT
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
```

1.1.9.5 OptionObject

浏览器在发送跨域请求之前会发送一个preflight请求（OPTIONS）并带上特定的来源域，HTTP方法和header信息等给OSS以决定是否发送真正的请求。 OSS可以通过Put Bucket cors接口来开启

Bucket的CORS支持，开启CORS功能之后，OSS在收到浏览器preflight请求时会根据设定的规则评估是否允许本次请求。如果不允许或者CORS功能没有开启，返回403 Forbidden。

请求语法

```
OPTIONS /ObjectName HTTP/1.1
Host: BucketName.regionid.example.com
Origin:Origin
Access-Control-Request-Method:HTTP method
Access-Control-Request-Headers:Request Headers
```

请求Header

表 1-64: 请求Header

名称	描述
Origin	请求来源域，用来标示跨域请求。 类型：字符串 默认值：无
Access-Control-Request-Method	表示在实际请求中将会用到的方法。 类型：字符串 默认值：无
Access-Control-Request-Headers	表示在实际请求中会用到的除了简单头部之外的headers。 类型：字符串 默认值：无

响应Header

表 1-65: 响应Header

名称	描述
Access-Control-Allow-Origin	请求中包含的Origin，如果不允许的话将不包含该头部。 类型：字符串
Access-Control-Allow-Methods	允许请求的HTTP方法，如果不允许该请求，则不包含该头部。 类型：字符串
Access-Control-Allow-Headers	允许请求携带的header的列表，如果请求中有不被允许的header，则不包含该头部，请求也将被拒绝。 类型：字符串
Access-Control-Expose-Headers	允许在客户端JavaScript程序中访问的headers的列表。 类型：字符串

名称	描述
Access-Control-Max-Age	允许浏览器缓存preflight结果的时间，单位为秒。 类型：整型

示例

请求示例：

```
OPTIONS /testobject HTTP/1.1
Host: ***.regionid.example.com
Date: Fri, 24 Feb 2012 05:45:34 GMT
Origin:http://www.example.com
Access-Control-Request-Method:PUT
Access-Control-Request-Headers:x-oss-test
```

返回示例：

```
HTTP/1.1 200 OK
x-oss-request-id: 5***C
Date: Fri, 24 Feb 2012 05:45:34 GMT
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Methods: PUT
Access-Control-Expose-Headers: x-oss-test
Connection: keep-alive
Content-Length: 0
Server: AliyunOSS
```

1.1.10 OSS错误响应

当用户访问OSS出现错误时，OSS会返回给用户相应的错误码和错误信息，便于用户定位问题，并做出适当的处理。

OSS的错误响应格式

当用户访问OSS出错时，OSS会返回给用户一个合适的3xx，4xx或者5xx的HTTP状态码；以及一个application/xml格式的消息体。

错误响应的消息体例子：

```
<?xml version="1.0" ?>
<Error xmlns=” http://doc.oss-cn-hangzhou.aliyuncs.com” >
 <Code>
 AccessDenied
 </Code>
 <Message>
 Query-string authentication requires the Signature, Expires and OSSAccessKeyId parameters
 </Message>
 <RequestId>
 1D842BC5425544BB
 </RequestId>
 <HostId>
```

```
 regionid.example.com
 </HostId>
</Error>
```

所有错误的消息体中都包括以下几个元素：

- Code：OSS返回给用户的错误码。
- Message：OSS给出的详细错误信息。
- RequestId：用于唯一标识该次请求的UUID；当你无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助。
- HostId：用于标识访问的OSS集群，与用户请求时使用的Host一致。

其他特殊的错误信息元素请参照每个请求的具体介绍。

OSS的错误码

OSS的错误码列表如下：

表 1-66: OSS的错误码

错误码	描述	HTTP状态码
AccessDenied	拒绝访问	403
BucketAlreadyExists	Bucket已经存在	409
BucketNotEmpty	Bucket不为空	409
EntityTooLarge	实体过大	400
EntityTooSmall	实体过小	400
FieldItemTooLong	Post请求中表单域过大	400
FilePartInterity	文件Part已改变	400
FilePartNotExist	文件Part不存在	400
FilePartStale	文件Part过时	400
IncorrectNumberOfFilesInPOSTRequest	Post请求中文件个数非法	400
InvalidArgument	参数格式错误	400
InvalidAccessKeyId	AccessKeyId不存在	403
InvalidBucketName	无效的Bucket名字	400
InvalidDigest	无效的摘要	400

错误码	描述	HTTP状态码
InvalidEncryptionAlgorithmError	指定的熵编码加密算法错误	400
InvalidObjectName	无效的Object名字	400
InvalidPart	无效的Part	400
InvalidPartOrder	无效的part顺序	400
InvalidPolicyDocument	无效的Policy文档	400
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket	400
InternalError	OSS内部发生错误	500
MalformedXML	XML格式非法	400
MalformedPOSTRequest	Post请求的body格式非法	400
MaxPOSTPreDataLengthExceededError	Post请求上传文件内容之外的body过大	400
MethodNotAllowed	不支持的方法	405
MissingArgument	缺少参数	411
MissingContentLength	缺少内容长度	411
NoSuchBucket	Bucket不存在	404
NoSuchKey	文件不存在	404
NoSuchUpload	Multipart Upload ID不存在	404
NotImplemented	无法处理的方法	400
PreconditionFailed	预处理错误	412
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟	403
RequestTimeout	请求超时	400
RequestIsNotMultiPartContent	Post请求content-type非法	400
SignatureDoesNotMatch	签名错误	403
TooManyBuckets	用户的Bucket数目超过限制	400

OSS不支持的操作

如果试图以OSS不支持的操作来访问某个资源，返回405 Method Not Allowed错误。

错误请求示例：

```
ABC /1.txt HTTP/1.1
Host: bucketname.regionid.example.com
Date: Thu, 11 Aug 2016 03:53:40 GMT
Authorization: signatureValue
```

返回示例：

```
HTTP/1.1 405 Method Not Allowed
Server: AliyunOSS
Date: Thu, 11 Aug 2016 03:53:44 GMT
Content-Type: application/xml
Content-Length: 338
Connection: keep-alive
x-oss-request-id: 57ABF6C8BC4D25D86CBA5ADE
Allow: GET DELETE HEAD PUT POST OPTIONS
<?xml version="1.0" encoding="UTF-8"?>
<Error>
 <Code>MethodNotAllowed</Code>
 <Message>The specified method is not allowed against this resource.</Message>
 <RequestId>57ABF6C8BC4D25D86CBA5ADE</RequestId>
 <HostId>bucketname.regionid.example.com</HostId>
 <Method>abc</Method>
 <ResourceType>Bucket</ResourceType>
</Error>
```

说明：

如果访问的资源是 /bucket/， ResourceType应该是bucket，如果访问的资源是 /bucket/object，ResourceType应该是object。

OSS操作支持但参数不支持的操作

如果在OSS合法的操作中，添加了OSS不支持的参数（例如在PUT的时候，加入If-Modified-Since参数），OSS会返回400 Bad Request错误。

错误请求示例：

```
PUT /abc.zip HTTP/1.1
Host: bucketname.regionid.example.com
Accept: */*
Date: Thu, 11 Aug 2016 01:44:50 GMT
If-Modified-Since: Thu, 11 Aug 2016 01:43:51 GMT
Content-Length: 363
```

返回示例：

```
HTTP/1.1 400 Bad Request
```

```
Server: AliyunOSS
Date: Thu, 11 Aug 2016 01:44:54 GMT
Content-Type: application/xml
Content-Length: 322
Connection: keep-alive
x-oss-request-id: 57ABD896CCB80C366955187E
x-oss-server-time: 0
<?xml version="1.0" encoding="UTF-8"?>
<Error>
 <Code>NotImplemented</Code>
 <Message>A header you provided implies functionality that is not implemented.</Message>
 <RequestId>57ABD896CCB80C366955187E</RequestId>
 <HostId>bucketname.regionid.example.com</HostId>
 <Header>If-Modified-Since</Header>
</Error>
```

1.1.11 S3 API兼容性说明

OSS提供了S3 API的兼容性，可以让您的数据从Amazon S3无缝迁移到阿里云OSS上。从Amazon S3迁移到OSS后，您仍然可以使用S3 API访问OSS，仅需要对S3的客户端应用进行如下改动：

1. 获取OSS主账号或子账号的AccessKeyId和AccessKeySecret，并在您使用的客户端和SDK中配置您申请的AccessKeyId与AccessKeySecret。
2. 设置客户端连接的endpoint为OSS endpoint。

迁移后用S3 API访问OSS

从S3迁移到OSS后，您使用S3 API访问OSS时，需要注意以下几点：

- Path style和virtual hosted style

  *Virtual hosted style*是指将Bucket放入host header的访问方式。OSS基于安全考虑，仅支持virtual hosted访问方式，所以在S3到OSS迁移后，客户端应用需要进行相应设置。部分S3工具默认使用Path style，也需要进行相应配置，否则可能导致OSS报错禁止访问。
- OSS对权限的定义与S3不完全一致，OSS仅支持S3中的private、public-read和public-read-write三种canned ACL模式。迁移后如有需要，可对权限进行相应调整。二者的主要区别可参考*表 1-67: OSS与S3的权限定义对比*。

表 1-67: OSS与S3的权限定义对比

对象	Amazon S3权限	Amazon S3	阿里云OSS
Bucket	READ	拥有bucket的list权限	对于bucket下的所有object，如果某object没有设置object权限，则该object可读

对象	Amazon S3权限	Amazon S3	阿里云OSS
	WRITE	Bucket下的Object可写入或覆盖	- 对于bucket下不存在的object，可写入 - 对于bucket下存在的object，如果该object没有设置object权限，则该object可被覆盖 - 可以initiate multipart upload
	READ_ACP	读取bucket ACL	读取bucket ACL，仅bucket owner和授权子账号拥有此权限
	WRITE_ACP	设置bucket ACL	设置bucket ACL，仅bucket owner和授权子账号拥有此权限
Object	READ	Object可读	Object可读
	WRITE	N/A	Object可以被覆盖
	READ_ACP	读取object ACL	读取object ACL，仅bucket owner和授权子账号拥有此权限
	WRITE_ACP	设置object ACL	设置object ACL，仅bucket owner和授权子账号拥有此权限

- 存储类型

  OSS支持标准（Standard）、低频访问（IA）和归档存储（Archive）三种存储类型，分别对应Amazon S3中的STANDARD、STANDARD_IA和GLACIER。

  与Amazon S3不同的是，OSS不支持在上传object时直接指定存储类型，object的存储类型由bucket的存储类型指定。OSS支持标准、低频访问和归档三种Bucket类型，Object的存储类型还可以通过lifecycle进行转换。

  归档存储类型的object在读取之前，要先使用Restore请求进行解冻操作。与S3不同，OSS会忽略S3 API中的解冻天数设置，解冻状态默认持续1天，用户可以延长到最多7天，之后，Object又回到初始时的冷冻状态。

- ETag

  - 对于PUT方式上传的object，OSS object的ETag与Amazon S3在大小写上有区别。OSS为大写，而S3为小写。如果客户端有关于ETag的校验，请忽略大小写。
  - 对于分片上传的object，OSS的ETag计算方式与S3不同。

与OSS兼容的S3 API

- Bucket操作：
  - Delete Bucket
  - Get Bucket（list objects）
  - Get Bucket ACL
  - Get Bucket lifecycle
  - Get Bucket location
  - Get Bucket logging
  - Head Bucket
  - Put Bucket
  - Put Bucket ACL
  - Put Bucket lifecycle
  - Put Bucket logging
- Object操作：
  - Delete Object
  - Delete Objects
  - Get Object
  - Get Object ACL
  - Head Object
  - Post Object
  - Put Object
  - Put Object Copy
  - Put Object ACL
- Multipart操作：
  - Abort Multipart Upload
  - Complete Multipart Upload

- Initiate Multipart Upload
- List Parts
- Upload Part
- Upload Part Copy

1.2 SDK参考

1.2.1 Java-SDK

1.2.1.1 前言

SDK 下载

- Java SDK 开发包最新版本 2.5.0：*aliyun_java_sdk_20170222.zip*
- github 地址：*https://github.com/aliyun/aliyun-oss-java-sdk*

简介

- OSS Java SDK 适用于 JDK 6 及以上版本；
- 本文档主要介绍 OSS Java SDK 的安装、使用及注意事项；
- 并且假设您已经开通了阿里云 OSS 服务，并创建了 AccessKeyId 和 AccessKeySecret。

兼容性

对于 2.×.× 系列 SDK：

- 接口：兼容
- 命名空间：兼容

对于 1.0.× 系列 SDK：

- 接口：兼容
- 命名空间：不兼容

  2.0.0 版本移除 1.0.x 版本中 Table Store 相关代码，调整包结构，将包名称 *com.aliyun.openservices.** 与 *com.aliyun.openservices.oss.** 更换为 *com.aliyun.oss.**。

1.2.1.2 安装

环境准备

适用于 JDK 6 及以上版本。

安装方式

方式一：在 Maven 项目中加入依赖项（推荐方式）

在 Maven 工程中使用 OSS Java SDK 只需在 pom.xml 中加入相应依赖即可。以 2.5.0 版本为例，在 dependencies 标签内加入如下内容：

```
<dependency>
    <groupId>com.aliyun.oss</groupId>
    <artifactId>aliyun-sdk-oss</artifactId>
    <version>2.5.0</version>
</dependency>
```

方式二：在 Eclipse 项目中导入 JAR 包

以 2.5.0 版本为例，步骤如下：

1. 下载 Java SDK 开发包版本号 *aliyun_java_sdk_20170222.zip*；
2. 解压该开发包；
3. 将解压后文件夹中的文件 *aliyun-sdk-oss-<versionId>.jar* 以及 lib 文件夹下的所有文件拷贝到您的项目中；
4. 在 Eclipse 中选择您的工程，右击**Properties** > **Java Build Path** > **Add JARs**；
5. 选中您在第三步拷贝的所有 JAR 文件。

经过以上几步，您就可以在 Eclipse 项目中使用 OSS Java SDK。

示例工程

OSS Java SDK 提供了基于 Maven、Ant 的示例工程，您可以在本地设备上编译运行示例工程。您也可以以示例工程为基础开发您的应用。

- Maven 示例工程：*aliyun-oss-java-sdk-demo-mvn.zip*
- Ant 示例工程：*aliyun-oss-java-sdk-demo-ant.zip*

说明：

- 编译运行前，请修改 *HelloOSS.java* 中 endpoint/accessKeyId/accessKeySecret/bucketName 为您的真实信息；
- 工程的编译运行方法，参看工程目录下 *README.md*。

示例程序

OSS Java SDK 提供丰富的示例程序，方便用户参考或直接使用。您可以通过以下两种方式获取示例程序：

- github 查看下载，OSS Java SDK *github* 下的 src/samples 为示例程序
- 下载 OSS Java SDK 开发包，如 *aliyun_java_sdk_20170222.zip*，解压后 aliyun_java_sdk_20170222/samples 为示例程序

示例包括以下内容：

示例文件	示例内容
GetStartedSample.java	展示了基本的上传、下载用法
SimpleGetObjectSample.java	展示了文件下载的用法
ListObjectsSample.java	展示了列举文件的用法
DeleteObjectsSample.java	展示了批量删除文件的用法
AppendObjectSample.java	展示了追加上传的用法
ObjectMetaSample.java	展示文件元信息的使用方法
CreateFolderSample.java	创建文件夹的用法，OSS文件夹的详细说明请参看文件夹
UploadSample.java	展示断点续传上传的用法
DownloadSample.java	展示了断点续传下载的用法
ImageSample.java	展示了图片服务的用法
PostObjectSample.java	展示了PostObject的用法，该实现不依赖于Java SDK
GetProgressSample.java	展示了上传、下载进度条的用法
CallbackSample.java	展示了上传回调的用法
CRCSample.java	展示了上传、下载CRC校验的用法
BucketOperationsSample.java	展示了存储空间配置的用法，包括权限、生命周期、日志、防盗链、CORS等
MultipartUploadSample.java	利用分片上传接口实现的并发上传，推荐直接使用断点续传上传（uploadFile）
ConcurrentGetObjectSample.java	利用范围下载实现的并发下载，推荐直接使用的断点续传下载（downloadFile）
UploadPartCopySample.java	展示了大文件分片复制的用法

1.2.1.3 初始化

OSSClient 是 OSS 服务的 Java 客户端，它为调用者提供一系列与 OSS 进行交互的接口，用于管理、操作存储空间（Bucket）和文件（Object）等 OSS 资源。使用 Java SDK 发起 OSS 请求，您需要初始化一个 OSSClient 实例，并根据需要修改 ClientConfiguration 的默认配置项。

确定 Endpoint

Endpoint 可以有以下几种形式：

示例	说明
http://oss-***.aliyuncs.com	以 HTTP 协议，公网访问杭州区域的 Bucket
https://oss-***.aliyuncs.com	以 HTTPS 协议，公网范围北京区域的 Bucket
http://my-***.com	以 HTTP 协议，通过用户自定义域名（CNAME）访问特定 Bucket

配置密钥

要接入阿里云 OSS，您需要拥有一个有效的 Access Key（包括 AccessKeyId 和 AccessKeySecret）用来进行签名认证。

获取 AccessKeyId 和 AccessKeySecret 之后，您便可以按照以下步骤进行初始化。

新建 OSSClient

使用 OSS 域名新建 OSSClient

新建一个 OSSClient 代码如下：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 使用访问OSS
// 关闭client
client.shutdown();
```

说明：

- 您的工程中可以有多个 OSSClient，也可以只有一个 OSSClient；
- OSSClient 可以并发使用；
- OSS 支持 https，当您的安全需求更高时，可以使用 https；

- OSSClient.shutdown 之后不能再使用。

使用自定义域名（CNAME）新建 OSSClient

下面的代码让客户端使用 CNAME 访问 OSS 服务。

```
String endpoint = "<yourEndpoint>";
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建ClientConfiguration实例，按照您的需要修改默认参数
ClientConfiguration conf = new ClientConfiguration();
// 开启支持CNAME选项
conf.setSupportCname(true);
// 创建OSSClient实例
OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret, conf);
// 使用访问OSS
// 关闭client
client.shutdown();
```

说明：

使用 CNAME 时无法使用 ListBuckets 接口。

使用 IP 新建 OSSClient

某些特殊情况（比如专有域）下，您需要 IP 地址做作为 endpoint。OSS Java sdk 2.1.2 及以后版本，会自动检测到 IP 地址，不需要再调用 setSLDEnabled 设置；OSS Java sdk 2.1.2 以前的版本，IP 地址初始化时需要设置 setSLDEnabled，代码如下：

```
// 请按照实际IP填写
String endpoint = "http://10.*.*.10";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建ClientConfiguration实例，按照您的需要修改默认参数
ClientConfiguration conf = new ClientConfiguration();
// 开启二级域名访问OSS，默认不开启
conf.setSLDEnabled(true)
// 创建OSSClient实例
OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret, conf);
// 使用访问OSS
// 关闭client
client.shutdown();
```

使用 STS 新建 OSSClient

使用 STS 新建一个 OSSClient 代码如下：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// 临时账号accessKeyId/accessKeySecret/securityToken
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
```

```
String securityToken = "<yourSecurityToken>";
// 创建OSSClient实例
OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret, securityToken);
// 使用访问OSS
// 关闭client
client.shutdown();
```

配置 OSSClient

如果您需要修改 OSSClient 的一些默认配置，请在构造 OSSClient 的时候传入 ClientConfiguration 实例。ClientConfiguration 是 OSSClient 的配置类，可配置代理、连接超时、最大连接数等参数。通过 ClientConfiguration 可以设置的参数见下表：

参数	描述	方法
MaxConnections	允许打开的最大 HTTP 连接数。默认为 1024	ClientConfiguration.setMaxConnections
SocketTimeout	Socket 层传输数据的超时时间（单位：毫秒）。默认为 50000 毫秒	ClientConfiguration.setSocketTimeout
ConnectionTimeout	建立连接的超时时间（单位：毫秒）。默认为 50000 毫秒	ClientConfiguration.setConnectionTimeout
ConnectionRequestTimeout	从连接池中获取连接的超时时间（单位：毫秒）。默认不超时	ClientConfiguration.setConnectionRequestTimeout
IdleConnectionTime	如果空闲时间超过此参数的设定值，则关闭连接（单位：毫秒）。默认为 60000 毫秒	ClientConfiguration.setIdleConnectionTime
MaxErrorRetry	请求失败后最大的重试次数。默认 3 次	ClientConfiguration.setMaxErrorRetry
SupportCname	是否支持 CNAME 作为 Endpoint，默认支持 CNAME	ClientConfiguration.setSupportCname
SLDEnabled	是否开启二级域名（Second Level Domain）的访问方式，默认不开启	ClientConfiguration.setSLDEnabled
Protocol	连接 OSS 所采用的协议（HTTP/HTTPS），默认为 HTTP	ClientConfiguration.setProtocol
UserAgent	用户代理，指 HTTP 的 User-Agent 头。默认为 aliyun-sdk-java	ClientConfiguration.setUserAgent

参数	描述	方法
ProxyHost	代理服务器主机地址	ClientConfiguration.setProxyHost
ProxyPort	代理服务器端口	ClientConfiguration.setProxyPort
ProxyUsername	代理服务器验证的用户名	ClientConfiguration.setProxyUsername
ProxyPassword	代理服务器验证的密码	ClientConfiguration.setProxyPassword

使用 ClientConfiguration 设置 OSSClient 参数代码如下：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建ClientConfiguration实例
ClientConfiguration conf = new ClientConfiguration();
// 设置OSSClient使用的最大连接数，默认1024
conf.setMaxConnections(200);
// 设置请求超时时间，默认50秒
conf.setSocketTimeout(10000);
// 设置失败请求重试次数，默认3次
conf.setMaxErrorRetry(5);
// 创建OSSClient实例
OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret, conf);
// 使用访问OSS
// 关闭OSSClient
client.shutdown();
```

1.2.1.4 快速入门

本节您将看到如何快速使用 OSS Java SDK，完成常见操作，如创建存储空间、上传文件、下载文件等。

初始化 OSSClient

向 OSS 发送任一 HTTP 请求之前，必须先创建一个 OSSClient 实例：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

// 使用访问OSS

// 关闭ossClient
```

```
ossClient.shutdown();
```

创建Bucket

存储空间（Bucket）是 OSS 全局命名空间，相当于数据的容器，可以存储若干文件（Object）。

以下代码展示如何新建一个 Bucket：

```
ossClient.createBucket("<bucketName>");
```

上传 Object

以下代码展示如何上传文件（object）至 OSS：

```
String content = "Hello OSS";
ossClient.putObject("<bucketName>", "<key>", new ByteArrayInputStream(content.getBytes()));
```

说明：

- Java SDK 通过 InputStream 上传 Object 至OSS。

下载Object

以下代码展示如何获取 Object 的文本内容：

```
OSSObject ossObject = ossClient.getObject("<bucketName>", "<key>");
InputStream content = ossObject.getObjectContent();
if (content != null) {
    BufferedReader reader = new BufferedReader(new InputStreamReader(content));
    while (true) {
        String line = reader.readLine();
        if (line == null) break;
        System.out.println("\n" + line);
    }
    content.close();
}
```

说明：

- 调用 OSSClient.GetObject 返回一个 OSSObject 实例，该实例包含文件内容及其元信息(meta)。
- 调用 OSSObject.GetObjectContent 获取文件输入流，可读取此输入流获取其内容，**用完之后关闭请这个流。**

列举 Object

当完成一系列上传 Object 操作后，可能需要查看 Bucket 下包含哪些 Object。以下代码展示如何列举指定 Bucket 下的 Object：

```
ObjectListing objectListing = ossClient.listObjects("<bucketName>");
for (OSSObjectSummary objectSummary : objectListing.getObjectSummaries()) {
    System.out.println(" - " + objectSummary.getKey() + "  " +
            "(size = " + objectSummary.getSize() + ")");
}
```

调用 OSSClient#listObjects 返回 ObjectListing 实例，该实例包含此次 listObject 请求的返回结果，可通过 ObjetListing#getObjectSummaries 获取所有 Object 的描述信息。

说明：

- 上面的代码默认列举 100 个 object。

删除 Object

以下代码展示如何删除指定 Object：

```
ossClient.deleteObject("<bucketName>", "<key>")
```

说明：

- OSS Java SDK **操作成功完成时**，**没有异常抛出**，**返回值有效**；**抛出异常说明操作失败**，**此时返回的数据无效**；
- 完整代码请参考：*GitHub*。

1.2.1.5 管理 Bucket

创建 Bucket

您可以使用 OSSClient.createBucket 创建 Bucket。如下代码展示如何新建一个 Bucket：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

// 创建bucket
String bucketName = "<your-oss-bucket-name>";
ossClient.createBucket(bucketName);
```

```
// 关闭client
ossClient.shutdown();
```

说明：

- Bucket的名字是**全局唯一的**，所以您需要保证Bucket名称不与别人重复。

上面代码创建的bucket，权限是**私有读写**。创建bucket时可以指定bucket权限，如下的示例代码：

```
CreateBucketRequest createBucketRequest= new CreateBucketRequest(bucketName);
// 设置bucket权限
createBucketRequest.setCannedACL(CannedAccessControlList.PublicRead);
ossClient.createBucket(createBucketRequest);
```

列举 Bucket

您可以使用 OSSClient.listBuckets 列举指定用户下的 Bucket。

简单列举

以下代码展示如何采用简单方式列举指定用户的 Bucket 列表：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

// 列举bucket
List<Bucket> buckets = ossClient.listBuckets();
for (Bucket bucket : buckets) {
    System.out.println(" - " + bucket.getName());
}

// 关闭client
ossClient.shutdown();
```

上面的代码最多只能返回 100 个 bucket，已经能满足用户的需求。列举 bucket 时，通过指定 prefix、marker、maxkeys 参数，可以过滤 bucket 列表，实现灵活的查询功能。

参数	作用
prefix	限定返回的 bucket name 必须以 prefix 作为前缀，可以不设定，不设定时不过滤前缀信息
marker	设定结果从 marker 之后按字母排序的第一个开始返回，可以不设定，不设定时从头开始返回

参数	作用
max keys	限定此次返回 bucket 的最大数，如果不设定，默认为 100，max-keys 取值不能大于 1000

指定前缀列举

```
ListBucketsRequest listBucketsRequest = new ListBucketsRequest();
listBucketsRequest.setPrefix("<yourBucketPrefix>");
for (Bucket bucket : ossClient.listBuckets()) {
    System.out.println(" - " + bucket.getName());
}
```

指定 max keys 列举

```
ListBucketsRequest listBucketsRequest = new ListBucketsRequest();
listBucketsRequest.setMaxKeys(500);
for (Bucket bucket : ossClient.listBuckets()) {
    System.out.println(" - " + bucket.getName());
}
```

删除 Bucket

您可以使用 OSSClient.deleteBucket 删除 Bucket。以下代码展示如何删除一个 Bucket：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

// 删除bucket
ossClient.deleteBucket("<bucketName>");

// 关闭client
ossClient.shutdown();
```

说明：

- 如果存储空间不为空（存储空间中有文件或者分片上传碎片），则存储空间无法删除；
- 必须先删除存储空间中的所有文件后，存储空间才能成功删除。

判断 Bucket 是否存在

您可以使用 OSSClient.doesBucketExist 接口判断该 Bucket 是否已存在。以下代码展示如何判断指定 Bucket 是否存在：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

boolean exists = ossClient.doesBucketExist("<bucketName>");

// 关闭client
ossClient.shutdown();
```

设置 Bucket ACL

Bucket 的 ACL 包含三类：Private（私有读写）、PublicRead（公共读私有写）、PublicReadWrite（公共读写）。您可以通过 OSSClient.setBucketAcl 设置 bucket 的权限。

权限	Java SDK 对应值
私有读写	CannedAccessControlList.Private
公共读私有写	CannedAccessControlList.PublicRead
公共读写	CannedAccessControlList.PublicReadWrite

以下代码展示如何设置 Bucket 的权限：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

// 设置bucket权限
ossClient.setBucketAcl("<bucketName>", CannedAccessControlList.Private);

// 关闭client
```

```
ossClient.shutdown();
```

获取 Bucket ACL

您可以通过 OSSClient.getBucketAcl 获取 bucket 的权限。以下代码展示如何获取 Bucket 的 ACL：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

AccessControlList acl = ossClient.getBucketAcl("<bucketName>");
// bucket权限
System.out.println(acl.toString());

// 关闭client
ossClient.shutdown();
```

获取 Bucket Location

Bucket Location 即 Bucket Region。

您可以通过 OSSClient.getBucketLocation 获取 bucket 的权限。以下代码展示如何获取 Bucket 的 Location：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

String location = ossClient.getBucketLocation("<bucketName>");
System.out.println(location);

// 关闭client
ossClient.shutdown();
```

获取 Bucket Info

Bucket 的 Info 包括 Location、CreationDate、Owner 及权限等信息。以下代码展示如何获取 Bucket 的 Info：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
```

```
String accessKeySecret = "<yourAccessKeySecret>";

// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);

BucketInfo info = ossClient.getBucketInfo("<bucketName>");
// Location
info.getBucket().getLocation();
// 创建日期
info.getBucket().getCreationDate();
// owner
info.getBucket().getOwner();
// 权限
info.getGrants();

// 关闭client
ossClient.shutdown();
```

1.2.1.6 上传文件

上传方式

在 OSS 中，用户操作的基本数据单元是文件（Object）。OSS Java SDK 提供了丰富的文件上传接口，可以通过以下方式上传文件：

- 流式上传
- 文件上传
- 追加上传
- 分片上传
- 断点续传上传

流式上传、文件上传、追加上传的文件（Object）最大不能超过 5 GB。当文件较大时，请使用分片上传，分片上传文件大小不能超过 48.8 TB。断点续传上传，支持并发、断点续传、自定义分片大小，断点续传是分片上传封装和加强。文件上传推荐使用断点续传。

简单上传

流式上传、文件上传称为简单上传。流式上传，使用 InputStream 作为 Object 数据源；文件上传使用本地文件作为 Object 数据源。

说明：

简单上传的完整代码请参考：*GitHub*

流式上传

您可以使用 OSSClient.putObject 上传您的数据流到 OSS。

上传字符串

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 上传字符串
String content = "Hello OSS";
ossClient.putObject("<yourBucketName>", "<yourKey>", new ByteArrayInputStream(content.
getBytes()));
// 关闭client
ossClient.shutdown();
```

上传 byte 数组

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId,accessKeySecret);
// 上传
byte[] content = "Hello OSS".getBytes();
ossClient.putObject("<yourBucketName>", "<yourKey>", new ByteArrayInputStream(content));
// 关闭client
ossClient.shutdown();
```

上传网络流

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 上传
InputStream inputStream = new URL("https://www.aliyun.com/").openStream();
ossClient.putObject("<yourBucketName>", "<yourKey>", inputStream);
// 关闭client
ossClient.shutdown();
```

上传文件流

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 上传文件流
```

```
InputStream inputStream = new FileInputStream("localFile");
ossClient.putObject("<yourBucketName>", "<yourKey>", inputStream);
// 关闭client
ossClient.shutdown();
```

上传本地文件

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 上传文件
ossClient.putObject("<yourBucketName>", "<yourKey>", new File("localFile"));
// 关闭client
ossClient.shutdown();
```

设置元信息

文件元信息（Object Meta），是对用户上传到 OSS 的文件的属性描述，分为两种：HTTP 标准属性（HTTP Headers）和 User Meta（用户自定义元信息）。 文件元信息可以在各种方式上传（流上传、文件上传、追加上传、分片上传、断点续传），或拷贝文件时进行设置。元信息的名称大小写不敏感。。

设定Http Header

OSS 允许用户自定义 Http header。Http header 请参考*RFC2616*。几个常用的 http header 说明如下：

名词	描述	默认值
Content-MD5	文件数据校验，设置了该值后 OSS 会启用文件内容 MD5 校验，把您提供的 MD5 与文件的 MD5 比较，不一致会抛出错误	无
Content-Type	文件的 MIME，定义文件的类型及网页编码，决定浏览器将以什么形式、什么编码读取文件。如果用户没有指定则根据 Key 或文件名的扩展名生成，如果没有扩展名则填默认值	application/octet-stream
Content-Disposition	指示 MINME 用户代理如何显示附加的文件，打开或下载，及文件名称	无
Content-Length	上传的文件的长度，超过流/文件的长度会截断，不足为实际值	流/文件时间长度

名词	描述	默认值
Expires	缓存过期时间，OSS 未使用，格式是格林威治时间（GMT）	无
Cache-Control	指定该 Object 被下载时的网页的缓存行为	无

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String content = "Hello OSS";
// 创建上传Object的Metadata
ObjectMetadata meta = new ObjectMetadata();
// 设置上传文件长度
meta.setContentLength(content.length());
// 设置上传MD5校验
String md5 = BinaryUtil.toBase64String(BinaryUtil.calculateMd5(content.getBytes()));
meta.setContentMD5(md5);
// 设置上传内容类型
meta.setContentType("text/plain");
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 上传文件
ossClient.putObject("<yourBucketName>", "<yourKey>", new ByteArrayInputStream(content.
getBytes()), meta);
// 关闭client
ossClient.shutdown();
```

说明：

- ObjectMetadata 提供常用的 HTTP Header 的设置，比如 Content-MD5、Content-Type 、Content-Length、Content-Disposition 、Content-Encoding 、Expires、x-oss-server-side-encryption 等；
- 使用 ObjectMetadata.setHeader(String key, Object value) 设置 Header。

用户自定义元信息

OSS 支持用户自定义 Object 的元信息，对 Object 进行描述。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String content = "Hello OSS";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 创建上传Object的Metadata
ObjectMetadata meta = new ObjectMetadata();
// 设置自定义元信息name的值为my-data
```

```
meta.addUserMetadata("property", "property-value");
// 上传文件
ossClient.putObject("<yourBucketName>", "<yourKey>", new ByteArrayInputStream(content.
getBytes()), meta);
// 关闭client
ossClient.shutdown();
```

说明：

- 在上面代码中，用户自定义了一个名称为“property”，值为“property-value”的元信息；
- 文件的元信息可以通过 OSSClient.getObjectMetadata 获取；
- 下载文件的时，文件的元信息也会同时下载；
- 一个文件可以有多个元信息，总大小不能超过 8 KB。

创建模拟文件夹

OSS 是没有文件夹这个概念的，所有元素都是以 Object 来存储。创建模拟文件夹本质上来说是创建了一个 size 为 0 的 Object。对于这个 Object 可以上传下载，只是控制台会对以“/”结尾的 Object 以文件夹的方式展示。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
final String keySuffixWithSlash = "parent_directory/";
ossClient.putObject("<bucketName>", keySuffixWithSlash, new ByteArrayInputStream(new
byte[0]));
// 关闭client
ossClient.shutdown();
```

说明：

- 创建模拟文件夹本质上来说是创建了一个名字以“/”结尾的文件。
- 对于这个文件照样可以上传下载，只是控制台会对以“/”结尾的文件以文件夹的方式展示。
- 多级目录创建最后一级即可，比如 dir1/dir2/dir3/，创建 dir1/dir2/dir3/ 即可，dir1/、dir1/dir2/ 不需要创建。

追加上传

简单上传，分片上传，断点续传上传，创建的Object都是Normal类型，这种Object在上传结束之后内容就是固定的，只能读取，不能修改。如果Object内容发生了改变，只能重新上传同名的Object

来覆盖之前的内容，这也是OSS和普通文件系统使用的一个重大区别。正因为这种特性，在很多应用场景下会很不方便，典型比如视频监控、视频直播领域等，视频数据在实时的不断产生。

OSS提供了用户通过追加上传（Append Object）的方式在一个Object后面直接追加内容的功能。通过这种方式操作的Object的类型为Appendable Object，而其他的方式上传的Object类型为Normal Object。每次追加上传的数据都能够即时可读。

您可以使用OSSClient.appendObject追加上传文件。

```
AppendObjectResult appendObject(AppendObjectRequest appendObjectRequest)
```

AppendObjectRequest可设置参数如下：

参数	作用	方法
BucketName	bucket名称	setBucketName(String bucketName)
Key	object名称	setKey(String key)
InputStream	待追加的内容，InputStream/File二选一	setInputStream(InputStream inputStream)
File	待追加的内容，InputStream/File二选一	setFile(File file)
ObjectMetadata	指定Object的元信息，第一次追加时有效	setMetadata(ObjectMetadata metadata)
Position	Object追加位置	setPosition(Long position)

AppendObjectResult的参数如下：

参数	含义	方法
nextPosition	指明下一次请求应当提供的position。实际上就是当前Object长度。	Long getNextPosition()
objectCRC64	Object的64位CRC值。该64位CRC根据*ECMA-182*标准计算得出	String getObjectCRC64()

说明：

追加上传的完整代码请参考：*GitHub*

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
```

```
String accessKeySecret = "<yourAccessKeySecret>";
String content = "Hello OSS";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
AppendObjectRequest appendObjectRequest = new AppendObjectRequest("<yourBucket
Name>",
        "<yourKey>", new ByteArrayInputStream(content.getBytes()));
// 第一次追加
appendObjectRequest.setPosition(0L);
AppendObjectResult appendObjectResult = ossClient.appendObject(appendObjectRequest);
// 第二次追加
appendObjectRequest.setPosition(appendObjectResult.getNextPosition());
appendObjectResult = ossClient.appendObject(appendObjectRequest);
// 第三次追加
appendObjectRequest.setPosition(appendObjectResult.getNextPosition());
appendObjectResult = ossClient.appendObject(appendObjectRequest);
// 关闭client
ossClient.shutdown();
```

说明：

- 追加上传的次数没有限制，文件大小上限为5GB。更大的文件请使用分片上传；
- 追加类型的文件(Append Object)暂时不支持copyObject操作。

断点续传上传

当上传大文件时，如果网络不稳定或者程序崩溃了，则整个上传就失败了。用户不得不重头再来，这样做不仅浪费资源，在网络不稳定的情况下，往往重试多次还是无法完成上传。通过OSSClient.uploadFile接口来实现断点续传上传，参数是**UploadFileRequest**，该请求有以下参数：

- bucket 存储空间名字，必选参数，通过构造方法设置
- key 上传到OSS的Object名字，必选参数，通过构造方法设置
- uploadFile 待上传的本地文件，必选参数，通过构造方法或setUploadFile设置
- partSize 分片大小，从100KB到5GB，单位是Byte，可选参数，默认100K，通过setPartSize设置
- taskNum 分片上传并发数，可选参数，默认为1，通过setTaskNum设置
- enableCheckpoint 上传是否开启断点续传，可选参数，默认断点续传功能关闭，通过setEnableCheckpoint设置
- checkpointFile 开启断点续传时，需要在本地记录分片上传结果，如果上传失败，下次不会再上传已经成功的分片，可选参数，默认与待上传的本地文件同目录，为*uploadFile.ucp*，可以通过setCheckpointFile设置
- objectMetadata，Object的元数据，可选参数，用户可以通过setObjectMetadata设置

- callback 上传成功后的回调，可选参数，用户可以通过setCallback设置。

其实现的原理是将要上传的文件分成若干个分片分别上传，最后所有分片都上传成功后，完成整个文件的上传。在上传的过程中会记录当前上传的进度信息（记录在checkpoint文件中），如果上传过程中某一分片上传失败，再次上传时会从checkpoint文件中记录的点继续上传。这要求再次调用时要指定与上次相同的checkpoint文件。上传完成后，checkpoint文件会被删除。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 设置断点续传请求
UploadFileRequest uploadFileRequest = new UploadFileRequest("<yourBucketName>", "<
yourKey>");
// 指定上传的本地文件
uploadFileRequest.setUploadFile("<yourLocalFile>");
// 指定上传并发线程数
uploadFileRequest.setTaskNum(5);
// 指定上传的分片大小
uploadFileRequest.setPartSize(1 * 1024 * 1024);
// 开启断点续传
uploadFileRequest.setEnableCheckpoint(true);
// 断点续传上传
ossClient.uploadFile(uploadFileRequest);
// 关闭client
ossClient.shutdown();
```

说明：

- 断点续传是分片上传的封装和加强，是用分片上传实现的；
- 文件较大或网络环境较差时，推荐使用分片上传；
- 断点续传支持指定ObjectMetadata，支持上传完成回调callback。

分片上传

对于大文件上传，可以切分成片上传。用户可以在如下的应用场景内（但不仅限于此），使用分片上传（Multipart Upload）模式：

- 需要支持断点上传。
- 上传超过100MB大小的文件。
- 网络条件较差，和OSS的服务器之间的链接经常断开。
- 上传文件之前，无法确定上传文件的大小。

分片上传（Multipart Upload）分为如下3个步骤：

- 初始化一个分片上传任务（InitiateMultipartUpload）
- 逐个或并行上传分片（UploadPart）
- 完成分片上传（CompleteMultipartUpload）或取消分片上传(AbortMultipartUpload)

分步完成Multipart Upload

说明：

分片上传的完整代码请参考：*GitHub*

初始化Multipart Upload

使用Multipart Upload模式传输数据前，必须先通知OSS初始化一个Multipart Upload事件。该操作会返回一个OSS服务器创建的全局唯一的Upload ID，用于标识本次Multipart Upload事件。用户可以根据这个ID来发起相关的操作，如中止Multipart Upload、查询Multipart Upload等。

调用OSSClient.initiateMultipartUpload初始化一个分片上传事件：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
String key = "yourKey";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
InitiateMultipartUploadRequest request = new InitiateMultipartUploadRequest(bucketName, key
);
InitiateMultipartUploadResult result = ossClient.initiateMultipartUpload(request);
String uploadId = result.getUploadId();
```

说明：

- 用InitiateMultipartUploadRequest指定上传文件的名字和所属存储空间（Bucket）；
- 在InitiateMultipartUploadRequest中，您也可以设置ObjectMeta；
- initiateMultipartUpload 的返回结果中含有UploadId，它是区分分片上传事件的唯一标识，在后面的操作中将用到它。

上传分片

初始化一个Multipart Upload之后，可以根据指定的Object名和Upload ID来分片（Part）上传数据。每一个上传的Part都有一个标识它的号码——分片号（part number，范围是1~10,000）。对于同一个Upload ID，该分片号不但唯一标识这一块数据，也标识了这块数据在整个文件内的相对位置。如果你用同一个分片号码，上传了新的数据，那么OSS上已有的这个分片的数据将被覆盖。除了最

后一块Part以外，其他的part最小为100KB；最后一块Part没有大小限制。每个分片不需要按顺序上传，甚至可以在不同进程、不同机器上上传，OSS会按照分片号排序组成大文件。

调用OSSClient.uploadPart上传分片：

```
List<PartETag> partETags = new ArrayList<PartETag>();
InputStream instream = new FileInputStream(new File("<localFile>"));
UploadPartRequest uploadPartRequest = new UploadPartRequest();
uploadPartRequest.setBucketName(bucketName);
uploadPartRequest.setKey(key);
uploadPartRequest.setUploadId(uploadId);
uploadPartRequest.setInputStream(instream);
// 设置分片大小，除最后一个分片外，其他分片要大于100KB
uploadPartRequest.setPartSize(100 * 1024);
// 设置分片号，范围是1~10000，
uploadPartRequest.setPartNumber(1);
UploadPartResult uploadPartResult = ossClient.uploadPart(uploadPartRequest);
partETags.add(uploadPartResult.getPartETag());
```

说明：

- UploadPart 方法要求除最后一个Part以外，其他的Part大小都要大于100KB。但是Upload Part接口并不会立即校验上传Part的大小（因为不知道是否为最后一块）；只有当Complete Multipart Upload的时候才会校验。
- OSS会将服务器端收到Part数据的MD5值放在ETag头内返回给用户。
- 为了保证数据在网络传输过程中不出现错误，SDK会自动设置Content-MD5，OSS会计算上传数据的MD5值与SDK计算的MD5值比较，如果不一致返回InvalidDigest错误码。
- Part号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
- 每次上传part时都要把流定位到此次上传块开头所对应的位置。
- 每次上传part之后，OSS的返回结果会包含一个 PartETag 对象，他是上传块的ETag与块编号（PartNumber）的组合，
- 在后续完成分片上传的步骤中会用到它，因此我们需要将其保存起来。一般来讲我们将这些PartETag 对象保存到List中。

完成分片上传

所有分片上传完成后，需要调用Complete Multipart Upload来完成整个文件的Multipart Upload。在执行该操作时，需要提供所有有效的分片列表（包括分片号和分片ETAG）；OSS收到提交的分片列表后，会逐一验证每个分片的有效性。当所有的数据Part验证通过后，OSS将把这些分片组合成一个完整的Object。

调用OSSClient.completeMultipartUpload完成分片上传：

```
Collections.sort(partETags, new Comparator<PartETag>() {
    @Override
    public int compare(PartETag p1, PartETag p2) {
        return p1.getPartNumber() - p2.getPartNumber();
    }
});
CompleteMultipartUploadRequest completeMultipartUploadRequest =
        new CompleteMultipartUploadRequest(bucketName, key, uploadId, partETags);
ossClient.completeMultipartUpload(completeMultipartUploadRequest);
```

说明：

- 上面代码中的 partETags 就是进行分片上传中保存的partETag的列表，它必须是按分片号升序排序；
- 分片可以是不连续的。

取消分片上传事件

该接口可以根据Upload ID中止对应的Multipart Upload事件。当一个Multipart Upload事件被中止后，就不能再使用这个Upload ID做任何操作，已经上传的Part数据也会被删除。

调用OSSClient.abortMultipartUpload取消分片上传事件：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 上传字符串
String content = "Hello OSS";
ossClient.putObject("<yourBucketName>", "<yourKey>", new ByteArrayInputStream(content.
getBytes()));
// 取消分片上传，其中uploadId来自于initiateMultipartUpload
AbortMultipartUploadRequest abortMultipartUploadRequest =
        new AbortMultipartUploadRequest("<yourBucketName>", "<yourKey>", "<uploadId>");
ossClient.abortMultipartUpload(abortMultipartUploadRequest);
// 关闭client
ossClient.shutdown();
```

获取已上传的分片

获取上传的分片可以罗列出指定Upload ID所属的所有已经上传成功的分片。可以调用

```
OSSClient.listParts
```

获取某个上传事件所有已上传分片。listParts的可设置的参数如下：

参数	作用	方法
UploadId	Upload Id，initiateMultipartUpload返回的结果获取。	ListPartsRequest.setUploadId(String uploadId)
MaxParts	OSS响应中的最大Part数目，即分页时每一页中Part数目。	ListPartsRequest.setMaxParts(int maxParts)
PartNumber Marker	指定List的起始位置，只有Part Number数目大于该参数的Part会被列出。	ListPartsRequest.setPartNumberMarker(Integer partNumberMarker)

简单列举

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 列举已上传的分片，其中uploadId来自于initiateMultipartUpload
ListPartsRequest listPartsRequest = new ListPartsRequest("<yourBucketName>", "<yourKey>",
 "<uploadId>");
PartListing partListing = ossClient.listParts(listPartsRequest);
for (PartSummary part : partListing.getParts()) {
    // 分片号，上传时候指定
    part.getPartNumber();
    // 分片数据大小
    part.getSize();
    // Part的ETag
    part.getETag();
    // Part的最后修改上传
    part.getLastModified();
}
// 关闭client
ossClient.shutdown();
```

说明：

- 默认情况下，如果存储空间中的分片上传事件的数量大于1000，则只会返回1000个Multipart Upload信息，且返回结果中 IsTruncated 为false，并返回 NextPartNumberMarker作为下此读取的起点。
- 如果没有一次性获取所有的上传分片，可以采用分页列举的方式。

获取所有已上传分片

默认情况下，listParts只能列举1000个分片，如果分片数量大于1000，列举所有分片请参考如下示例。

```
// endpoint以杭州为例，其他region请按实际情况填写
```

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 列举所有已上传的分片
PartListing partListing;
ListPartsRequest listPartsRequest = new ListPartsRequest("<yourBucketName>", "<yourKey>",
 "<uploadId>");
do {
    partListing = ossClient.listParts(listPartsRequest);
    for (PartSummary part : partListing.getParts()) {
        // 分片号，上传时候指定
        part.getPartNumber();
        // 分片数据大小
        part.getSize();
        // Part的ETag
        part.getETag();
        // Part的最后修改上传
        part.getLastModified();
    }
    listPartsRequest.setPartNumberMarker(partListing.getNextPartNumberMarker());
} while (partListing.isTruncated());
// 关闭client
ossClient.shutdown()
```

分页获取所有分片

默认情况下，listParts一次列举1000个分片，上面的

```
获取所有已上传分片
```

也是分页的一种特殊情况，每页1000个分片。如果需要指定每页分片的数量，即一次列举的分片数量，请参考以下代码。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 分页列举已上传的分片
PartListing partListing;
ListPartsRequest listPartsRequest = new ListPartsRequest("<yourBucketName>", "<yourKey>",
 "<uploadId>");
// 每页100个分片
listPartsRequest.setMaxParts(100);
do {
    partListing = ossClient.listParts(listPartsRequest);
    for (PartSummary part : partListing.getParts()) {
        // 分片号，上传时候指定
        part.getPartNumber();
        // 分片数据大小
        part.getSize();
        // Part的ETag
```

```
        part.getETag();
        // Part的最后修改上传
        part.getLastModified();
    }
    listPartsRequest.setPartNumberMarker(partListing.getNextPartNumberMarker());
} while (partListing.isTruncated());
// 关闭client
ossClient.shutdown();
```

获取存储空间内所有分片上传事件

列举分片上传事件可以罗列出所有执行中的分片上传事件，即已经初始化的尚未Complete或者Abort的分片上传事件。列举分片上传的可设置的参数如下：

参数	作用	方法
Prefix	限定返回的文件名(object)必须以Prefix作为前缀。注意使用Prefix查询时，返回的文件名(Object)中仍会包含Prefix。	ListMultipartUploadsRequest.setPrefix(String prefix)
Delimiter	用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素。	ListMultipartUploadsRequest.setDelimiter(String delimiter)
MaxUploads	限定此次返回分片上传事件的最大数目，默认为1000，MaxUploads取值不能大于1000。	ListMultipartUploadsRequest.setMaxUploads(Integer maxUploads)
KeyMarker	所有Object名字的字典序大于KeyMarker参数值的Multipart事件。可以与UploadIdMarker参数一同使用来指定返回结果的起始位置。	ListMultipartUploadsRequest.setKeyMarker(String keyMarker)
UploadIdMarker	与KeyMarker参数一同使用来指定返回结果的起始位置。如果KeyMarker参数未设置，则OSS忽略UploadIdMarker参数。 如果KeyMarker参数被设置，查询结果中包含：所有Object名字的字典序大于KeyMarker参数值和Object名字等于KeyMarker参数值且Upload ID比UploadIdMarker参数值大的分片上传事件。	ListMultipartUploadsRequest.setUploadIdMarker(String uploadIdMarker)

调用OSSClient.listMultipartUploads获取存储空间内分片上传事件。

简单列举分片上传事件

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 列举分片上传事件
ListMultipartUploadsRequest listMultipartUploadsRequest = new ListMultipartUploadsRequest(
bucketName);
MultipartUploadListing multipartUploadListing = ossClient.listMultipartUploads(listMultip
artUploadsRequest);
for (MultipartUpload multipartUpload : multipartUploadListing.getMultipartUploads()) {
    // Upload Id
    multipartUpload.getUploadId();
    // Key
    multipartUpload.getKey();
    // Date of initiate multipart upload
    multipartUpload.getInitiated();
}
// 关闭client
ossClient.shutdown();
```

说明：

- 默认情况下，如果存储空间中的分片上传事件的数量大于1000，则只会返回1000个文件，且返回结果中 IsTruncated 为 false，返回 NextKeyMarker 和 NextUploadIdMarker 作为下次读取的起点。
- 如果没有一次性获取所有的上传事件，可以采用分页列举的方式。

列举全部上传事件

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 列举分片上传事件
MultipartUploadListing multipartUploadListing;
ListMultipartUploadsRequest listMultipartUploadsRequest = new ListMultipartUploadsRequest(
bucketName);
do {
    multipartUploadListing = ossClient.listMultipartUploads(listMultipartUploadsRequest);
    for (MultipartUpload multipartUpload : multipartUploadListing.getMultipartUploads()) {
        // Upload Id
        multipartUpload.getUploadId();
        // Key
        multipartUpload.getKey();
        // Date of initiate multipart upload
```

```
        multipartUpload.getInitiated();
    }
    listMultipartUploadsRequest.setKeyMarker(multipartUploadListing.getNextKeyMarker());
    listMultipartUploadsRequest.setUploadIdMarker(multipartUploadListing.getNextUpl
oadIdMarker());
} while (multipartUploadListing.isTruncated());
// 关闭client
ossClient.shutdown();
```

分页列举全部上传事件

默认情况下，listMultipartUploads一次列举1000个上传事件，上面的\列举全部上传事件是分页的一种特殊情况，每页1000个事件。如果需要指定每页事件的数量，即一次列举的事件数量，请参考以下代码。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 列举分片上传事件
MultipartUploadListing multipartUploadListing;
ListMultipartUploadsRequest listMultipartUploadsRequest = new ListMultipartUploadsRequest(
bucketName);
// 每个页中事件数目
listMultipartUploadsRequest.setMaxUploads(50);
do {
    multipartUploadListing = ossClient.listMultipartUploads(listMultipartUploadsRequest);
    for (MultipartUpload multipartUpload : multipartUploadListing.getMultipartUploads()) {
        // Upload Id
        multipartUpload.getUploadId();
        // Key
        multipartUpload.getKey();
        // Date of initiate multipart upload
        multipartUpload.getInitiated();
    }
    listMultipartUploadsRequest.setKeyMarker(multipartUploadListing.getNextKeyMarker());
    listMultipartUploadsRequest.setUploadIdMarker(multipartUploadListing.getNextUpl
oadIdMarker());
} while (multipartUploadListing.isTruncated());
// 关闭client
ossClient.shutdown();
```

Post上传

OSS Java SDK目前暂不支持Post上传Object，示例提供了PostObject的示例程序，您可以直接使用或再示例基础上修改，请参考*PostObjectSample*。让您在修改或实现Post Object过程中有问题或疑问，请参考*JAVA模拟PostObject表单上传OSS*。

上传进度条

OSS Java sdk支持进度条功能，指示上传/下载的进度。下面的代码以OSSClient.putObject为例，说明进度条功能的使用。

说明：

上传进度条的完整代码请参考：*GitHub*

```
static class PutObjectProgressListener implements ProgressListener {
    private long bytesWritten = 0;
    private long totalBytes = -1;
    private boolean succeed = false;
    @Override
    public void progressChanged(ProgressEvent progressEvent) {
        long bytes = progressEvent.getBytes();
        ProgressEventType eventType = progressEvent.getEventType();
        switch (eventType) {
        case TRANSFER_STARTED_EVENT:
            System.out.println("Start to upload.....");
            break;
        case REQUEST_CONTENT_LENGTH_EVENT:
            this.totalBytes = bytes;
            System.out.println(this.totalBytes + " bytes in total will be uploaded to OSS");
            break;
        case REQUEST_BYTE_TRANSFER_EVENT:
            this.bytesWritten += bytes;
            if (this.totalBytes != -1) {
                int percent = (int)(this.bytesWritten * 100.0 / this.totalBytes);
                System.out.println(bytes + " bytes have been written at this time, upload progress: " +
percent + "%(" + this.bytesWritten + "/" + this.totalBytes + ")");
            } else {
                System.out.println(bytes + " bytes have been written at this time, upload ratio:
unknown" + "(" + this.bytesWritten + "/...)");
            }
            break;
        case TRANSFER_COMPLETED_EVENT:
            this.succeed = true;
            System.out.println("Succeed to upload, " + this.bytesWritten + " bytes have been
transferred in total");
            break;
        case TRANSFER_FAILED_EVENT:
            System.out.println("Failed to upload, " + this.bytesWritten + " bytes have been transferre
d");
            break;
        default:
            break;
        }
    }
    public boolean isSucceed() {
        return succeed;
    }
}
public static void main(String[] args) {
    String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
    String accessKeyId = "<accessKeyId>";
    String accessKeySecret = "<accessKeySecret>";
    String bucketName = "<bucketName>";
    String key = "object-get-progress-sample";
```

```
    OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
    try {
        // 带进度条的上传
        ossClient.putObject(new PutObjectRequest(bucketName, key, new FileInputStream("<
yourLocalFile>")).
                <PutObjectRequest>withProgressListener(new PutObjectProgressListener()));
    } catch (Exception e) {
        e.printStackTrace();
    }
    ossClient.shutdown();
}
```

说明：

- putObject/getObject/uploadPart都支持进度条功能；
- uploadFile/downloadFile不支持进度条功能。

上传回调

OSS在上传文件完成的时候可以提供回调（Callback）给应用服务器。用户只需要在发送给OSS的请求中携带相应的Callback参数，即能实现回调。现在支持CallBack的接口有：PutObject、PostObject、CompleteMultipartUpload。

上传回调的一种典型应用场景是与授权第三方上传同时使用，客户端在上传文件到OSS的时候指定到服务器端的回调，当客户端的上传任务在OSS执行完毕之后，OSS会向应用服务器端主动发起HTTP请求进行回调，这样服务器端就可以及时得到上传完成的通知从而可以完成诸如数据库修改等操作，当回调请求接收到服务器端的响应之后OSS才会将状态返回给客户端。

下面以PutObject为例说明上传回调的用法。

说明：

上传回调的完整代码请参考：*GitHub*

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 您的回调服务器地址，如http://oss-demo.aliyuncs.com或http://127.0.0.1:9090
String callbackUrl = "<yourCallbackServerUrl>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
String content = "Hello OSS";
PutObjectRequest putObjectRequest = new PutObjectRequest(bucketName, "key",
        new ByteArrayInputStream(content.getBytes()));
// 上传回调参数
Callback callback = new Callback();
callback.setCallbackUrl(callbackUrl);
```

```
callback.setCallbackHost("oss-cn-hangzhou.aliyuncs.com");
callback.setCallbackBody("{\\\"mimeType\\\":${mimeType},\\\"size\\\":${size}}");
callback.setCallbackBodyType(CallbackBodyType.JSON);
callback.addCallbackVar("x:var1", "value1");
callback.addCallbackVar("x:var2", "value2");
putObjectRequest.setCallback(callback);
PutObjectResult putObjectResult = ossClient.putObject(putObjectRequest);
// 读取上传回调返回的消息内容
byte[] buffer = new byte[1024];
putObjectResult.getCallbackResponseBody().read(buffer);
// 一定要close，否则会造成连接资源泄漏
putObjectResult.getCallbackResponseBody().close();
// 关闭client
ossClient.shutdown();
```

说明：

- **上传回调返回的消息体一定要close**，否则会造成连接资源泄漏。

1.2.1.7 下载文件

下载方式

OSS Java SDK 提供了丰富的文件下载接口，用户可以通过以下方式从 OSS 中下载文件：

- 流式下载
- 下载到本地文件
- 断点续传下载
- 范围下载

流式下载

在进行大文件下载时，往往不希望一次性处理全部内容，而是希望流式地处理，一次处理一部分内容。

说明：

流式下载的完整代码请参考：*GitHub*

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
OSSObject ossObject = ossClient.getObject(bucketName, "yourKey");
// 读Object内容
System.out.println("Object content:");
```

```
BufferedReader reader = new BufferedReader(new InputStreamReader(ossObject.getObjectC
ontent()));
while (true) {
    String line = reader.readLine();
    if (line == null) break;
    System.out.println("\n" + line);
}
reader.close();
// 关闭client
ossClient.shutdown();
```

说明：

- OSSObject实例包含文件所在的存储空间（Bucket）、文件的名称、Object Metadata以及一个输入流；
- 通过操作输入流将文件的内容读取到文件或者内存中。而Object Metadata包含ETag、HTTP Header及自定义的元信息；
- OSSClient.getObject获取的流一定要显示close，否则会造成资源泄露；

  ```
  OSSObject ossObject = ossClient.getObject(bucketName, key);
  ossObject.getObjectContent().close();
  ```

- 假如需要从OSS流式读取64KB的数据，请使用如下的方式多次读取，直到读取64KB或者文件结束：

  ```
  byte[] buf = new byte[1024];
  InputStream in = ossObject.getObjectContent();
  for (int n = 0; n != -1; ) {
    n = in.read(buf, 0, buf.length);
  }
  in.close();
  ```

  而不是如下的形式：

  ```
  byte[] buf = new byte[64 * 1024];
  InputStream in = ossObject.getObjectContent();
  in.read(buf, 0, buf.length);
  in.close();
  ```

  原因是，流式读取一次不一定能读到全部数据，详细说明请参考 *InputStream.read* 。

下载到本地文件

把Object的内容下载到指定的本地文件中。如果指定的本地文件不存在则会新建。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
```

```
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 下载object到文件
ossClient.getObject(new GetObjectRequest(bucketName, "<yourKey>"), new File("<yourLocalF
ile>"));
// 关闭client
ossClient.shutdown();
```

范围下载

如果OSS文件较大，并且只需要其中一部分数据，可以使用范围下载，下载指定范围的数据。如果指定的下载范围是0 - 100，则返回第0到第100个字节的数据，包括第100个，共101字节的数据，即[0, 100]。如果指定的范围无效，则传送整个文件。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
GetObjectRequest getObjectRequest = new GetObjectRequest(bucketName, "<yourKey>");
// 获取0~1000字节范围内的数据，包括0和1000，共1001个字节的数据
getObjectRequest.setRange(0, 1000);
// 范围下载
OSSObject ossObject = ossClient.getObject(getObjectRequest);
// 读取数据
byte[] buf = new byte[1024];
InputStream in = ossObject.getObjectContent();
for (int n = 0; n != -1; ) {
    n = in.read(buf, 0, buf.length);
}
// InputStream数据读完成后，一定要close，否则会造成连接泄漏
in.close();
// 关闭client
ossClient.shutdown();
```

说明：

- 如果指定的Range无效(比如开始位置、结束位置为负数，大于文件大小)，则会下载整个文件；
- 假如需要从OSS流式读取64KB的数据，请使用如下的方式多次读取，直到读取64KB或者文件结束：

```
byte[] buf = new byte[1024];
InputStream in = ossObject.getObjectContent();
for (int n = 0; n != -1; ) {
  n = in.read(buf, 0, buf.length);
}
```

```
in.close();
```

而不是如下的形式：

```
byte[] buf = new byte[64 * 1024];
InputStream in = ossObject.getObjectContent();
in.read(buf, 0, buf.length);
in.close();
```

原因是，流式读取一次不一定能读到全部数据，详细说明请参考 *InputStream.read* 。

断点续传下载

当下载大文件时，如果网络不稳定或者程序崩溃了，则整个下载就失败了。用户不得不重头再来，这样做不仅浪费资源，在网络不稳定的情况下，往往重试多次还是无法完成下载。通过OSSClient.downloadFile接口来实现断点续传分片下载，参数是**DownloadFileRequest**，该请求有以下参数：

- bucket 存储空间名字，必选参数，通过构造方法设置
- key 下载到OSS的Object名字，必选参数，通过构造方法设置
- downloadFile 本地文件，下载到该文件，可选参数，默认是key，通过构造方法或setDownloadFile设置
- partSize 分片大小，从1B到5GB，单位是Byte，可选参数，默认100K，通过setPartSize设置
- taskNum 分片下载并发数，可选参数，默认为1，通过setTaskNum设置
- enableCheckpoint 下载是否开启断点续传，可选参数，默认断点续传功能关闭，通过setEnableCheckpoint设置
- checkpointFile 开启断点续传时，需要在本地记录分片下载结果，如果下载失败，下次不会再下载已经成功的分片，可选参数，默认与downloadFile同目录，为*downloadFile.ucp*，可以通过setCheckpointFile设置

其实现的原理是将要下载的Object分成若干个分片分别下载，最后所有分片都下载成功后，完成整个文件的下载。在下载的过程中会记录当前下载的进度信息（记录在checkpoint文件中）和已下载的分片，如果下载过程中某一分片下载失败，再次下载时会从checkpoint文件中记录的点继续下载。这要求再次调用时要指定与上次相同的checkpoint文件。下载完成后，checkpoint文件会被删除。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
```

```
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 下载请求，10个任务并发下载，启动断点续传
DownloadFileRequest downloadFileRequest = new DownloadFileRequest("bucketName", "key
");
downloadFileRequest.setDownloadFile("downloadFile");
downloadFileRequest.setTaskNum(10);
downloadFileRequest.setEnableCheckpoint(true);
// 下载文件
DownloadFileResult downloadRes = ossClient.downloadFile(downloadFileRequest);
// 下载成功时，会返回文件的元信息
downloadRes.getObjectMetadata();
// 关闭client
ossClient.shutdown();
```

限定条件下载

下载文件时，可以指定一个或多个限定条件，满足限定条件时下载，不满足时报错，不下载文件。

可以使用的限定条件如下：

参数	说明
If-Modified-Since	如果指定的时间早于实际修改时间，则正常传送。否则返回错误。
If-Unmodified-Since	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件；否则返回错误。
If-Match	如果传入期望的ETag和object的 ETag匹配，则正常传输；否则返回错误。
If-None-Match	如果传入的ETag值和Object的ETag不匹配，则正常传输；否则返回错误。

说明：

- 如果If-Modified-Since设定的时间不符合规范，直接返回文件，并返回200 OK；
- If-Modified-Since和If-Unmodified-Since可以同时存在，If-Match和If-None-Match也可以同时存在；
- 如果包含If-Unmodified-Since并且不符合或者包含If-Match并且不符合，返回412 precondition failed；
- 如果包含If-Modified-Since并且不符合或者包含If-None-Match并且不符合，返回304 Not Modified。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
```

```
String accessKeySecret = "<yourAccessKeySecret>";
String bucketName = "<yourBucketName>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
GetObjectRequest request = new GetObjectRequest(bucketName, "<yourKey>");
request.setModifiedSinceConstraint(new Date());
// 下载object到文件
ossClient.getObject(request, new File("<yourLocalFile>"));
// 关闭client
ossClient.shutdown();
```

说明：

- ETag的值可以通过OSSClient.getObjectMetadata获取；
- OSSClient.getObject，OSSClient.downloadFile都支持限定条件。

下载进度条

OSS Java sdk支持进度条功能，指示上传/下载的进度。下面的代码以OSSClient.getObject为例，说明进度条功能的使用。

说明：

下载进度条的完整代码请参考：*GitHub*

```
static class GetObjectProgressListener implements ProgressListener {
        private long bytesRead = 0;
        private long totalBytes = -1;
        private boolean succeed = false;
        @Override
        public void progressChanged(ProgressEvent progressEvent) {
            long bytes = progressEvent.getBytes();
            ProgressEventType eventType = progressEvent.getEventType();
            switch (eventType) {
            case TRANSFER_STARTED_EVENT:
                System.out.println("Start to download.....");
                break;
            case RESPONSE_CONTENT_LENGTH_EVENT:
                this.totalBytes = bytes;
                System.out.println(this.totalBytes + " bytes in total will be downloaded to a local file");
                break;
            case RESPONSE_BYTE_TRANSFER_EVENT:
                this.bytesRead += bytes;
                if (this.totalBytes != -1) {
                    int percent = (int)(this.bytesRead * 100.0 / this.totalBytes);
                    System.out.println(bytes + " bytes have been read at this time, download progress:
 " +
                            percent + "%(" + this.bytesRead + "/" + this.totalBytes + ")");
                } else {
                    System.out.println(bytes + " bytes have been read at this time, download ratio:
unknown" +
                            "(" + this.bytesRead + "/...)");
                }
                break;
            case TRANSFER_COMPLETED_EVENT:
```

```
                this.succeed = true;
                System.out.println("Succeed to download, " + this.bytesRead + " bytes have been
transferred in total");
                break;
            case TRANSFER_FAILED_EVENT:
                System.out.println("Failed to download, " + this.bytesRead + " bytes have been
transferred");
                break;
            default:
                break;
            }
        }
        public boolean isSucceed() {
            return succeed;
        }
    }
public static void main(String[] args) {
    String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
    String accessKeyId = "<accessKeyId>";
    String accessKeySecret = "<accessKeySecret>";
    String bucketName = "<bucketName>";
    String key = "object-get-progress-sample";
    OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
    try {
            // 带进度条的下载
            client.getObject(new GetObjectRequest(bucketName, key).
                    <GetObjectRequest>withProgressListener(new GetObjectProgressListener()),
                    new File("<yourLocalFile>"));
    } catch (Exception e) {
        e.printStackTrace();
    }
    ossClient.shutdown();
}
```

说明：

- putObject/getObject/uploadPart都支持进度条功能；
- uploadFile/downloadFile不支持进度条功能。

1.2.1.8 管理文件

在OSS中，用户可以通过一系列的接口管理存储空间(Bucket)中的文件(Object)，比如SetObjectAcl，GetObjectAcl，ListObjects，DeleteObject，CopyObject，DoesObjectExist等。Object的名字又称为key或object key。

Object是否存在

通过OSSClient.doesObjectExist判断文件（object）是否存在。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
```

```
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// Object是否存在
boolean found = ossClient.doesObjectExist("<bucketName>", "<key>")
// 关闭client
ossClient.shutdown();
```

Object ACL

Object有四种访问权限：Default（默认），Private（私有读写），PublicRead（公共读私有写），PublicReadWrite（公共读写），含义如下：

权限	描述
默认	Objec是遵循Bucket的读写权限，即Bucket是什么权限，Object就是什么权限，Object的默认权限
私有读写	Object是私有资源，即只有该Object的Owner拥有该Object的读写权限，其他的用户没有权限操作该Object
公共读私有写	Object是公共读资源，即非Object Owner只有Object的读权限，而Object Owner拥有该Object的读写权限
公共读写	Object是公共读写资源，即所有用户拥有对该Object的读写权限

Object的权限优先级高于Bucket。例如Bucket是private的，而Object ACL是公共读写，则访问这个Object时，先判断Object的ACL，所有用户都拥有这个Object的访问权限，即使这个Bucket是private。如果某个Object从来没设置过ACL，则访问权限遵循Bucket ACL。

设置Object ACL

您可以通过OSSClient.setObjectAcl设置Object的权限。

权限	Java SDK对应值
私有读写	CannedAccessControlList.Private
公共读私有写	CannedAccessControlList.PublicRead
公共读写	CannedAccessControlList.PublicReadWrite

下面代码为Object设置ACL：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 设置Object权限
ossClient.setObjectAcl("<bucketName>", "<key>", CannedAccessControlList.PublicRead)
```

```
// 关闭client
ossClient.shutdown();
```

获取Object ACL

您可以通过OSSClient.getObjectAcl获取Object的权限。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
//读取Object ACL
ObjectAcl objectAcl = ossClient.getObjectAcl("<bucketName>", "<key>");
System.out.println(objectAcl.getPermission().toString());
// 关闭client
ossClient.shutdown();
```

获取文件元信息(Object Meta)

文件元信息(Object Meta)，是对用户上传到OSS的文件的属性描述，分为两种：HTTP标准属性（HTTP Headers）和User Meta（用户自定义元信息）。 文件元信息可以在各种方式上传或者拷贝文件时进行设置。

获取文件元信息可以使用OSSClient.getSimplifiedObjectMeta或OSSClient.getObjectMetadata。getSimplifiedObjectMeta只能获取文件的ETag、Size（文件大小）、 LastModified（最后修改时间）；getObjectMetadata能获取文件的全部元数据。**getSimplifiedObjectMeta更轻量、更快。**

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 获取文件的部分元信息
SimplifiedObjectMeta objectMeta = ossClient.getSimplifiedObjectMeta("<bucketName>", "<key
>");
System.out.println(objectMeta.getSize());
System.out.println(objectMeta.getETag());
System.out.println(objectMeta.getLastModified());
// 获取文件的全部元信息
ObjectMetadata metadata = ossClient.getObjectMetadata("<bucketName>", "<key>");
System.out.println(metadata.getContentType());
System.out.println(metadata.getLastModified());
System.out.println(metadata.getExpirationTime());
// 关闭client
```

```
ossClient.shutdown();
```

列出存储空间中的文件

可以通过OSSClient.listObjects列出bucket里的Objects。listObjects有三类参数格式：

- ObjectListing listObjects(String bucketName)
- ObjectListing listObjects(String bucketName, String prefix)
- ObjectListing listObjects(ListObjectsRequest listObjectsRequest)

前两类称为简单列举，最多返回100条object，参数prefix是指定返回Object的前缀。最后一类提供多种过滤功能，可以实现灵活的查询功能。

ObjectListing的参数如下：

参数	含义	方法
ObjectSummaries	限定返回的object meta。	List<OSSObjectSummary> getObjectSummaries()
Prefix	本次查询结果的开始前缀。	String getPrefix()
Delimiter	是一个用于对Object名字进行分组的字符。	String getDelimiter()
Marker	标明这次List Object的起点。	String getMarker()
MaxKeys	响应请求内返回结果的最大数目。	int getMaxKeys()
NextMarker	下一次List Object的起点。	String getNextMarker()
IsTruncated	指明是否所有的结果都已经返回。	boolean isTruncated()
CommonPrefixes	如果请求中指定了delimiter参数，则返回的包含CommonPrefixes元素。该元素标明以delimiter结尾，并有共同前缀的object的集合。	List<String> getCommonPrefixes()
EncodingType	指明返回结果中编码使用的类型。	String getEncodingType()

说明：

listObjects的完整代码请参考：*GitHub*

简单列举

列举出Bucket下的Object，最多100条object。

```
// endpoint以杭州为例，其他region请按实际情况填写
```

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 列举Object
ObjectListing objectListing = ossClient.listObjects("<bucketName>", "<KeyPrifex>");
List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
for (OSSObjectSummary s : sums) {
    System.out.println("\t" + s.getKey());
}
// 关闭client
ossClient.shutdown();
```

列举出Bucket下的指定前缀的Object，最多100条object。

```
ObjectListing objectListing = ossClient.listObjects("<bucketName>", "<KeyPrifex>");
List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
for (OSSObjectSummary s : sums) {
    System.out.println("\t" + s.getKey());
}
```

通过ListObjectsRequest列出文件

可以通过设置ListObjectsReques的参数实现各种灵活的查询功能。ListObjectsReques的可设置的参数如下：

参数	作用	方法
Prefix	限定返回的object key必须以prefix作为前缀。	setPrefix(String prefix)
Delimiter	是一个用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素—CommonPrefixes。	setDelimiter(String delimiter)
Marker	设定结果从marker之后按字母排序的第一个开始返回。	setMarker(String marker)
MaxKeys	限定此次返回object的最大数，如果不设定，默认为100，max-keys取值不能大于1000。	setMaxKeys(Integer maxKeys)
EncodingType	请求响应体中Object名称采用的编码方式，目前支持url。	setEncodingType(String encodingType)

指定最大返回条数

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
final int maxKeys = 200;
```

```
// 列举Object
ObjectListing objectListing = ossClient.listObjects(new ListObjectsRequest("<bucketName>").
withMaxKeys(maxKeys));
List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
for (OSSObjectSummary s : sums) {
    System.out.println("\t" + s.getKey());
}
// 关闭client
ossClient.shutdown();
```

返回指定前缀的object

最多返回100条。

```
final String keyPrefix = "<keyPrefix>"
ObjectListing objectListing = ossClient.listObjects(new ListObjectsRequest("<bucketName>").
withPrefix(keyPrefix));
List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
for (OSSObjectSummary s : sums) {
    System.out.println("\t" + s.getKey());
}
```

从指定Object后返回

不包括指定的Object，最多返回100条。

```
final String keyMarker = "<keyMarker>"
ObjectListing objectListing = ossClient.listObjects(new ListObjectsRequest("<bucketName>").
withMarker(keyMarker));
List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
for (OSSObjectSummary s : sums) {
    System.out.println("\t" + s.getKey());
}
```

分页获取所有Object

分页获取所有Object，每页maxKeys条Object。

```
final int maxKeys = 200;
String nextMarker = null;
do {
    objectListing = ossClient.listObjects(new ListObjectsRequest("<bucketName>").withMarker(
nextMarker).withMaxKeys(maxKeys));
    List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
    for (OSSObjectSummary s : sums) {
        System.out.println("\t" + s.getKey());
    }
    nextMarker = objectListing.getNextMarker();
} while (objectListing.isTruncated());
```

分页获取所有特定Object后的Object

分页获取所有特定Object后的Object，每页maxKeys条Object。

```
final int maxKeys = 200;
String nextMarker = "<nextMarker>";
```

```
do {
    objectListing = ossClient.listObjects(new ListObjectsRequest("<bucketName>").withMarker(
nextMarker).withMaxKeys(maxKeys));
    List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
    for (OSSObjectSummary s : sums) {
        System.out.println("\t" + s.getKey());
    }
    nextMarker = objectListing.getNextMarker();
} while (objectListing.isTruncated());
```

分页所有获取指定前缀的Object

分页所有获取指定前缀的Object，每页maxKeys条Object。

```
final int maxKeys = 200;
final String keyPrefix = "<keyPrefix>";
String nextMarker = "<nextMarker>";
do {
    objectListing = ossClient.listObjects(new ListObjectsRequest("<bucketName>").
            withPrefix(keyPrefix).withMarker(nextMarker).withMaxKeys(maxKeys));
    List<OSSObjectSummary> sums = objectListing.getObjectSummaries();
    for (OSSObjectSummary s : sums) {
        System.out.println("\t" + s.getKey());
    }
    nextMarker = objectListing.getNextMarker();
} while (objectListing.isTruncated());
```

指定Object名字编码

如果Object名字含有特殊字符，如**' " & < >** 、**中文** 等，需要进行编码传输。OSS目前支持**url** 编码。

```
final int maxKeys = 200;
final String keyPrefix = "<keyPrefix>";
String nextMarker = "<nextMarker>";
ObjectListing objectListing;
do {
    ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);
    listObjectsRequest.setPrefix(keyPrefix);
    listObjectsRequest.setMaxKeys(maxKeys);
    listObjectsRequest.setMarker(nextMarker);
    // 指定Object名称编码传输
    listObjectsRequest.setEncodingType("url");
    objectListing = ossClient.listObjects(listObjectsRequest);
    // Object解码
    for (OSSObjectSummary objectSummary: objectListing.getObjectSummaries()) {
        System.out.println("Key:" + URLDecoder.decode(objectSummary.getKey(), "UTF-8"));
    }
    // CommonPrefixe解码
    for (String commonPrefixes: objectListing.getCommonPrefixes()) {
        System.out.println("CommonPrefixes:" + URLDecoder.decode(commonPrefixes, "UTF-8
"));
    }
    // NextMarker解码
    if (objectListing.getNextMarker() != null) {
        nextMarker = URLDecoder.decode(objectListing.getNextMarker(), "UTF-8");
    }
```

```
} while (objectListing.isTruncated());
```

模拟文件夹功能

OSS是没有文件夹这个概念的，所有元素都是以Object来存储。创建模拟文件夹本质上来说是创建了一个size为0的Object。对于这个Object可以上传下载，只是控制台会对以“/”结尾的Object以文件夹的方式展示。

您可以通过 Delimiter 和 Prefix 参数的配合模拟出文件夹功能。Delimiter 和 Prefix 的组合效果是这样的：

- 如果把 Prefix 设为某个文件夹名，就可以罗列以此 Prefix 开头的文件，即该文件夹下递归的所有的文件和子文件夹（目录）。文件名在Contents中显示。
- 如果再把 Delimiter 设置为“/”时，返回值就只罗列该文件夹下的文件和子文件夹（目录），该文件夹下的子文件名（目录）返回在 CommonPrefixes 部分，子文件夹下递归的文件和文件夹不被显示。

说明：

- 创建文件的完整代码请参考：*GitHub*。

假设Bucket中有4个文件： oss.jpg ， fun/test.jpg ， fun/movie/001.avi ， fun/movie/007.avi ， “/” 作为文件夹的分隔符。下面的示例展示了如何模拟文件夹功能。

列出存储空间内所有文件

当我们需要获取存储空间下的所有文件时，可以这样写：

```
// 构造ListObjectsRequest请求
ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);
// 列出Object
ObjectListing listing = ossClient.listObjects(listObjectsRequest);
// 遍历所有Object
System.out.println("Objects:");
for (OSSObjectSummary objectSummary : listing.getObjectSummaries()) {
    System.out.println(objectSummary.getKey());
}
// 遍历所有CommonPrefix
System.out.println("CommonPrefixs:");
for (String commonPrefix : listing.getCommonPrefixes()) {
    System.out.println(commonPrefix);
}
```

输出：

```
Objects:
fun/movie/001.avi
fun/movie/007.avi
```

```
fun/test.jpg
oss.jpg
CommonPrefixs:
```

递归列出目录下所有文件

我们可以通过设置 Prefix 参数来获取某个目录（fun/）下所有的文件：

```
// 构造ListObjectsRequest请求
ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);
listObjectsRequest.setPrefix("fun/");
// 递归列出fun目录下的所有文件
ObjectListing listing = ossClient.listObjects(listObjectsRequest);
// 遍历所有Object
System.out.println("Objects:");
for (OSSObjectSummary objectSummary : listing.getObjectSummaries()) {
    System.out.println(objectSummary.getKey());
}
// 遍历所有CommonPrefix
System.out.println("\nCommonPrefixs:");
for (String commonPrefix : listing.getCommonPrefixes()) {
    System.out.println(commonPrefix);
}
```

输出：

```
Objects:
fun/movie/001.avi
fun/movie/007.avi
fun/test.jpg
CommonPrefixs:
```

列出目录下的文件和子目录

在 Prefix 和 Delimiter 结合的情况下，可以列出目录（fun/）下的文件和子目录：

```
// 构造ListObjectsRequest请求
ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);
// "/" 为文件夹的分隔符
listObjectsRequest.setDelimiter("/");
// 列出fun目录下的所有文件和文件夹
listObjectsRequest.setPrefix("fun/");
ObjectListing listing = ossClient.listObjects(listObjectsRequest);
// 遍历所有Object
System.out.println("Objects:");
for (OSSObjectSummary objectSummary : listing.getObjectSummaries()) {
    System.out.println(objectSummary.getKey());
}
// 遍历所有CommonPrefix
System.out.println("\nCommonPrefixs:");
for (String commonPrefix : listing.getCommonPrefixes()) {
    System.out.println(commonPrefix);
```

```
}
```

输出：

```
Objects:
fun/test.jpg
CommonPrefixs:
fun/movie/
```

说明：

- 返回的结果中， ObjectSummaries 的列表中给出的是fun目录下的文件。
- 而 CommonPrefixs 的列表中给出的是fun目录下的所有子文件夹。可以看出 fun/movie/001.avi ， fun/movie/007.avi 两个文件并没有被列出来，因为它们属于fun文件夹下的movie目录。

删除文件

删除单个文件

您可以通过OSSClient.deleteObject删除单个文件。

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 删除Object
ossClient.deleteObject("<bucketName>", "<key>");
// 关闭client
ossClient.shutdown();
```

删除多个文件

您可以通过OSSClient.deleteObjects批量删除文件。

```
public DeleteObjectsResult deleteObjects(DeleteObjectsRequest deleteObjectsRequest)
```

每次最多删除1000个Object，并提供两种返回模式：详细(verbose)模式和简单(quiet)模式：

- 详细模式：返回的成功删除Object的结果，即DeleteObjectsResult.getDeletedObjects，默认模式；
- 简单模式：返回的删除过程中出错的Object结果，即DeleteObjectsResult.getDeletedObjects。

DeleteObjectsRequest可设置参数如下：

参数	作用	方法
Keys	需要删除的Objects	setKeys(List<String>)
Quiet	返回模式，默认详细模式；true简单模式，false详细模式	setQuiet(boolean)
EncodingType	指定对返回的Key进行编码，目前支持url	setEncodingType(String)

DeleteObjectsResult的参数如下：

参数	含义	方法
deletedObjects	删除结果，详细模式时成功删除的objects，简单模式时删除失败的objects	List<String> getDeletedObjects()
EncodingType	deletedObjects中Key的进行编码，为空没有编码	getEncodingType()

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 删除Objects
List<String> keys = new ArrayList<String>();
keys.add("key0");
keys.add("key1");
keys.add("key2");
DeleteObjectsResult deleteObjectsResult = ossClient.deleteObjects(new DeleteObje
ctsRequest("<bucketName>").withKeys(keys));
List<String> deletedObjects = deleteObjectsResult.getDeletedObjects();
// 关闭client
ossClient.shutdown();
```

说明：

批量删除文件的完整代码请参考：*GitHub*

拷贝文件

在同一个区域（杭州，深圳，青岛等）中，可以将Object从一个Bucket复制到另外一个Bucket。您可以通过OSSClient.copyObject。

- CopyObjectResult copyObject(String sourceBucketName, String sourceKey, String destinatio nBucketName, String destinationKey)
- CopyObjectResult copyObject(CopyObjectRequest copyObjectRequest)

第一个方法指定源Bucket/Key和目标源Bucket/Key，目标object的内容和元数据与源object相同，称为简单拷贝。第二个方法允许指定目标文件的原因数据、允许指定拷贝的限制条件。如果拷贝操作的源Object地址和目标Object地址相同，则直接替换源Object的meta信息。

CopyObjectRequest可设置参数如下：

参数	作用	方法
sourceBucketName	源Object所在的Bucket的名称	setSourceBucketName(String sourceBucketName)
sourceKey	源Object的Key	setSourceKey(String sourceKey)
destinatio nBucketName	目标Object所在的Bucket的名称	setDestinationBucketName(String destinationBucketName)
destinationKey	目标Object的Key	setDestinationKey(String destinatio nKey)
newObjectMetadata	目标Object的元信息	setNewObjectMetadata(ObjectMeta data newObjectMetadata)
matchingET agConstraints	拷贝的限制条件，如果源Object的ETAG值和提供的ETAG相等，则执行拷贝操作；否则返回错误	setMatchingETagConstraints(List< String> matchingETagConstraints)
nonmatchin gEtagConstraints	拷贝的限制条件，如果源Object的ETAG值和用户提供的ETAG不相等，则执行拷贝操作；否则返回错误	setNonmatchingETagConstraints(List< String> nonmatchingEtagConstraints)
unmodified SinceConstraint	拷贝的限制条件，如果传入参数中的时间等于或者晚于文件实际修改时间，则正常拷贝；否则返回错误	setUnmodifiedSinceConstraint(Date unmodifiedSinceConstraint)
modifiedSi nceConstraint	拷贝的限制条件，如果源Object自从用户指定的时间以后被修改过，则执行拷贝操作；否则返回错误	setModifiedSinceConstraint(Date modifiedSinceConstraint)

CopyObjectRequest的参数如下：

参数	含义	方法
etag	OSS Object唯一性标志	String getETag()

参数	含义	方法
lastModified	Object最后修改时间	Date getLastModified()

说明：

- 用户需要有源Object的操作权限，否则会无法完成操作。
- 该操作不支持跨Region拷贝数据。比如：不支持将杭州Bucket里的Object拷贝到青岛。
- 该操作支持的最大Object大小为1GB。

简单拷贝

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 拷贝Object
CopyObjectResult result = ossClient.copyObject("<srcBucketName>", "<srcKey>", "<
destBucketName>", "<destKey>");
System.out.println("ETag: " + result.getETag() + " LastModified: " + result.getLastModified());
// 关闭client
ossClient.shutdown();
```

通过CopyObjectRequest拷贝

也可以通过 CopyObjectRequest 实现Object的拷贝：

```
// endpoint以杭州为例，其他region请按实际情况填写
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// accessKey请登录https://ak-console.aliyun.com/#/查看
String accessKeyId = "<yourAccessKeyId>";
String accessKeySecret = "<yourAccessKeySecret>";
// 创建OSSClient实例
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 创建CopyObjectRequest对象
CopyObjectRequest copyObjectRequest = new CopyObjectRequest(srcBucketName, srcKey,
destBucketName, destKey);
// 设置新的Metadata
ObjectMetadata meta = new ObjectMetadata();
meta.setContentType("text/html");
copyObjectRequest.setNewObjectMetadata(meta);
// 复制Object
CopyObjectResult result = ossClient.copyObject(copyObjectRequest);
System.out.println("ETag: " + result.getETag() + " LastModified: " + result.getLastModified());
// 关闭client
```

```
ossClient.shutdown();
```

拷贝大文件

CopyObject只能copy小于1GB的文件，大文件需要使用分片拷贝(Upload Part Copy)。分片拷贝分为三步：

- 初始化分片拷贝任务OSSClient.initiateMultipartUpload；
- 分片拷贝OSSClient.uploadPartCopy，除最后一个分片外，其他的分片大小都要大于100KB；
- 提交分片拷贝任务OSSClient.completeMultipartUpload。

说明：

分片拷贝的完整代码请参考：*GitHub*

```
String sourceBucketName = "<sourceBucketName>";
String sourceKey = "<sourceKey>";
String targetBucketName = "<targetBucketName>";
String targetKey = "<targetKey>";
// 得到被拷贝object大小
ObjectMetadata objectMetadata = ossClient.getObjectMetadata(sourceBucketName,
sourceKey);
long contentLength = objectMetadata.getContentLength();
// 分片大小，10MB
long partSize = 1024 * 1024 * 10;
// 计算分块数目
int partCount = (int) (contentLength / partSize);
if (contentLength % partSize != 0) {
    partCount++;
}
System.out.println("total part count:" + partCount);
// 初始化拷贝任务
InitiateMultipartUploadRequest initiateMultipartUploadRequest = new InitiateMultipartUpl
oadRequest(targetBucketName, targetKey);
InitiateMultipartUploadResult initiateMultipartUploadResult = ossClient.initiateMultipartUpload(
initiateMultipartUploadRequest);
String uploadId = initiateMultipartUploadResult.getUploadId();
// 分片拷贝
List<PartETag> partETags = new ArrayList<PartETag>();
for (int i = 0; i < partCount; i++) {
     // 计算每个分块的大小
    long skipBytes = partSize * i;
    long size = partSize < contentLength - skipBytes ? partSize : contentLength - skipBytes;
    // 创建UploadPartCopyRequest
    UploadPartCopyRequest uploadPartCopyRequest =
        new UploadPartCopyRequest(sourceBucketName, sourceKey, targetBucketName,
targetKey);
    uploadPartCopyRequest.setUploadId(uploadId);
    uploadPartCopyRequest.setPartSize(size);
    uploadPartCopyRequest.setBeginIndex(skipBytes);
    uploadPartCopyRequest.setPartNumber(i + 1);
    UploadPartCopyResult uploadPartCopyResult = ossClient.uploadPartCopy(uploadPart
CopyRequest);
    // 将返回的PartETag保存到List中
    partETags.add(uploadPartCopyResult.getPartETag());
```

```
}
// 提交分片拷贝任务
CompleteMultipartUploadRequest completeMultipartUploadRequest = new CompleteMu
ltipartUploadRequest(
                targetBucketName, targetKey, uploadId, partETags);
ossClient.completeMultipartUpload(completeMultipartUploadRequest);
```

说明：

- 分片拷贝时，可以指定目标object的元信息，通过InitiateMultipartUploadRequest指定；
- 分片拷贝也支持限定条件，通过UploadPartCopyRequest指定。

1.2.1.9 授权访问

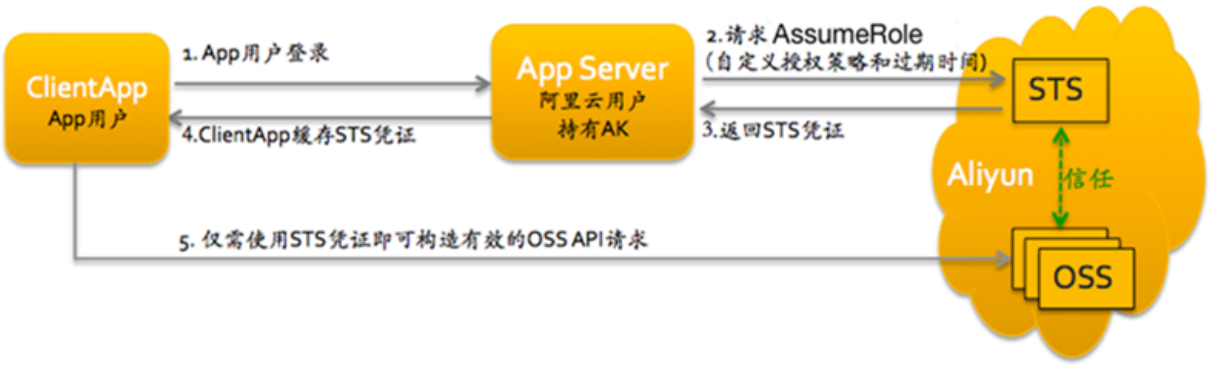
使用STS服务临时授权

介绍

OSS可以通过阿里云STS服务，临时进行授权访问。阿里云STS (Security Token Service) 是为云计算用户提供临时访问令牌的Web服务。通过STS，您可以为第三方应用或联邦用户（用户身份由您自己管理）颁发一个自定义时效和权限的访问凭证。第三方应用或联邦用户可以使用该访问凭证直接调用阿里云产品API，或者使用阿里云产品提供的SDK来访问云产品API。

- 您不需要透露您的长期密钥(AccessKey)给第三方应用，只需要生成一个访问令牌并将令牌交给第三方应用即可。这个令牌的访问权限及有效期限都可以由您自定义。
- 您不需要关心权限撤销问题，访问令牌过期后就自动失效。

以APP应用为例，交互流程如下图：



方案的详细描述如下：

1. App用户登录。App用户身份是客户自己管理。客户可以自定义身份管理系统，也可以使用外部Web账号或OpenID。对于每个有效的App用户来说，AppServer是可以确切地定义出每个App用户的最小访问权限。

2. AppServer请求STS服务获取一个安全令牌（SecurityToken）。在调用STS之前，AppServer需要确定App用户的最小访问权限（用Policy语法描述）以及授权的过期时间。 然后通过调用STS的AssumeRole（扮演角色）接口来获取安全令牌。
3. STS返回给AppServer一个有效的访问凭证，包括一个安全令牌（SecurityToken）、临时访问密钥（AccessKeyId, AccessKeySecret）以及过期时间。
4. AppServer将访问凭证返回给ClientApp。ClientApp可以缓存这个凭证。当凭证失效时，ClientApp需要向AppServer申请新的有效访问凭证。比如，访问凭证有效期为1小时，那么ClientApp可以每30分钟向AppServer请求更新访问凭证。
5. ClientApp使用本地缓存的访问凭证去请求Aliyun Service API。云服务会感知STS访问凭证，并会依赖STS服务来验证访问凭证，并正确响应用户请求。

关键是调用STS服务接口AssumeRole来获取有效访问凭证即可。也可以直接使用STS SDK来调用该方法。

使用STS凭证构造签名请求

用户的client端拿到STS临时凭证后，通过其中安全令牌(SecurityToken)以及临时访问密钥(AccessKeyId, AccessKeySecret)生成OSSClient。以上传Object为例：

```
String accessKeyId = "<accessKeyId>";
String accessKeySecret = "<accessKeySecret>";
String securityToken = "<securityToken>"
// 以杭州为例
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";

OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret, securityToken);
```

使用URL签名授权访问

生成签名URL

通过生成签名URL的形式提供给用户一个临时的访问URL。在生成URL时，您可以指定URL过期的时间，从而限制用户长时间访问。

生成一个签名的URL

代码如下：

```
String bucketName = "your-bucket-name";
String key = "your-object-key";

// 设置URL过期时间为1小时
Date expiration = new Date(new Date().getTime() + 3600 * 1000);

// 生成URL
```

```
URL url = client.generatePresignedUrl(bucketName, key, expiration);
```

生成的URL默认以GET方式访问，这样，用户可以直接通过浏览器访问相关内容。

生成其他Http方法的URL

如果您想允许用户临时进行其他操作（比如上传，删除Object），可能需要签名其他方法的URL，如下：

```
// 生成PUT方法的URL
URL url = client.generatePresignedUrl(bucketName, key, expiration, HttpMethod.PUT);
```

通过传入 HttpMethod.PUT 参数，用户可以使用生成的URL上传Object。

添加用户自定义参数（UserMetadata）

如果您想生成签名的URL来上传Object，并指定UserMetadata，Content-Type等头信息，可以这样做：

```
// 创建请求
GeneratePresignedUrlRequest generatePresignedUrlRequest = new GeneratePresignedUrl
Request(bucketName, key);

// HttpMethod为PUT
generatePresignedUrlRequest.setMethod(HttpMethod.PUT);

// 添加UserMetadata
generatePresignedUrlRequest.addUserMetadata("author", "baymax");

// 添加Content-Type
request.setContentType("application/octet-stream");

// 生成签名的URL
URL url = client.generatePresignedUrl(generatePresignedUrlRequest);
```

需要注意的是，上述过程只是生成了签名的URL，您仍需要在request header中添加meta的信息。可以参考下面的代码。

使用签名URL发送请求

现在java SDK支持put object和get object两种方式的URL签名请求。

使用URL签名的方式getobject

```
//服务器端生成url签名字串
OSSClient Server  = new OSSClient(endpoint, accessId, accessKey);
Date expiration = DateUtil.parseRfc822Date("Wed, 18 Mar 2015 14:20:00 GMT");
GeneratePresignedUrlRequest request = new GeneratePresignedUrlRequest(bucketName, key
, HttpMethod.GET);
//设置过期时间
request.setExpiration(expiration);
// 生成URL签名(HTTP GET请求)
URL signedUrl = Server .generatePresignedUrl(request);
```

```
System.out.println("signed url for getObject: " + signedUrl);

//客户端使用使用url签名字串发送请求
OSSClient client  = new OSSClient(endpoint, accessKeyId, accessKeySecret);
Map<String, String> customHeaders = new HashMap<String, String>();
// 添加GetObject请求头
customHeaders.put("Range", "bytes=100-1000");
OSSObject object = client.getObject(signedUrl,customHeaders);
```

使用URL签名的方式putobject

```
//服务器端生成url签名字串
OSSClient Server = new OSSClient(endpoint, accessKeyId, accessKeySecret);
Date expiration = DateUtil.parseRfc822Date("Wed, 18 Mar 2015 14:20:00 GMT");
GeneratePresignedUrlRequest request = new GeneratePresignedUrlRequest(bucketName, key
, HttpMethod.PUT);
//设置过期时间
request.setExpiration(expiration);
//设置Content-Type
request.setContentType("application/octet-stream");
// 添加user meta
request.addUserMetadata("author", "aliy");
// 生成URL签名(HTTP PUT请求)
URL signedUrl = Server.generatePresignedUrl(request);
System.out.println("signed url for putObject: " + signedUrl);

//客户端使用使用url签名字串发送请求
OSSClient client  = new OSSClient(endpoint, accessKeyId, accessKeySecret);
File f = new File(filePath);
FileInputStream fin = new FileInputStream(f);
// 添加PutObject请求头
Map<String, String> customHeaders = new HashMap<String, String>();
customHeaders.put("Content-Type", "application/octet-stream");
// 添加user meta
customHeaders.put("x-oss-meta-author", "aliy");
PutObjectResult result = client.putObject(signedUrl, fin, f.length(), customHeaders);
```

1.2.1.10 生命周期管理

生命周期规则

OSS允许用户对Bucket设置生命周期规则，以自动淘汰过期掉的文件，节省存储空间。针对不同前缀的文件，用户可以同时设置多条规则。一条规则包含：

- 规则ID，用于标识一条规则，不能重复
- 受影响的文件前缀，此规则只作用于符合前缀的文件
- 过期时间，有三种指定方式：
  1. 指定距文件最后修改时间N天过期
  2. 指定日期创建前的文件过期，之后的不过期

3. 指定在具体的某一天过期，即在那天之后符合前缀的文件将会过期，**而不论文件的最后修改时间**。不推荐使用。

- 是否生效

上面的过期规则对用户上传的文件有效。用户通过uploadPart上传的分片，也可以设置过期规则。Multipart的Lifecycle和文件的类似，过期时间支持1、2两种，不支持3，生效是以init Multipart upload的时间为准。

设置生命周期规则

通过OSSClient.setBucketLifecycle来设置生命周期规则：

```
SetBucketLifecycleRequest request = new SetBucketLifecycleRequest("bucketName");
// 最近修改3天后过期
request.AddLifecycleRule(new LifecycleRule(ruleId0, matchPrefix0, RuleStatus.Enabled, 3));
// 特定日期后过期
request.AddLifecycleRule(new LifecycleRule(ruleId1, matchPrefix1, RuleStatus.Enabled,
        DateUtil.parseIso8601Date("2022-10-12T00:00:00.000Z")));
// 特定日期前创建的文件过期
LifecycleRule rule = new LifecycleRule(ruleId4, matchPrefix4, RuleStatus.Enabled);
rule.setCreatedBeforeDate(DateUtil.parseIso8601Date("2022-10-12T00:00:00.000Z"));
request.AddLifecycleRule(rule);
// Multipart3天后过期
rule = new LifecycleRule(ruleId2, matchPrefix2, RuleStatus.Enabled);
LifecycleRule.AbortMultipartUpload abortMultipartUpload = rule.new AbortMultipartUpload();
abortMultipartUpload.setExpirationDays(3);
rule.setAbortMultipartUpload(abortMultipartUpload);
request.AddLifecycleRule(rule);
// 特定日期前的Multipart过期
rule = new LifecycleRule(ruleId3, matchPrefix3, RuleStatus.Enabled);
abortMultipartUpload = rule.new AbortMultipartUpload();
abortMultipartUpload.setCreatedBeforeDate(DateUtil.parseIso8601Date("2022-10-12T00:00:00
.000Z"));
rule.setAbortMultipartUpload(abortMultipartUpload);
request.AddLifecycleRule(rule);
ossClient.setBucketLifecycle(request);
```

查看生命周期规则

通过OSSClient.GetBucketLifecycle来查看生命周期规则：

```
List<LifecycleRule> rules = ossClient.getBucketLifecycle("bucketName");
for (LifecycleRule rule : rules) {
    System.out.println(rule.getId());
    System.out.println(rule.getPrefix());
    System.out.println(rule.getExpirationDays());
```

```
}
```

清空生命周期规则

通过OSSClient.DeleteBucketLifecycle设置来清空生命周期规则：

```
ossClient.deleteBucketLifecycle("bucketName");
```

1.2.1.11 跨域资源共享

跨域资源共享(CORS)允许web端的应用程序访问不属于本域的资源。OSS提供接口方便开发者控制跨域访问的权限。

设定CORS规则

通过setBucketCors 方法将指定的存储空间上设定一个跨域资源共享CORS的规则，如果原规则存在则覆盖原规则。具体的规则主要通过CORSRule类来进行参数设置。代码如下：

```
SetBucketCORSRequest request = new SetBucketCORSRequest();
request.setBucketName(bucketName);
//CORS规则的容器，每个bucket最多允许10条规则
ArrayList<CORSRule> putCorsRules = new ArrayList<CORSRule>();
CORSRule corRule = new CORSRule();
ArrayList<String> allowedOrigin = new ArrayList<String>();
//指定允许跨域请求的来源
allowedOrigin.add( "http://www.b.com");
ArrayList<String> allowedMethod = new ArrayList<String>();
//指定允许的跨域请求方法(GET/PUT/DELETE/POST/HEAD)
allowedMethod.add("GET");
ArrayList<String> allowedHeader = new ArrayList<String>();
//控制在OPTIONS预取指令中Access-Control-Request-Headers头中指定的header是否允许。
allowedHeader.add("x-oss-test");
ArrayList<String> exposedHeader = new ArrayList<String>();
//指定允许用户从应用程序中访问的响应头
exposedHeader.add("x-oss-test1");
corRule.setAllowedMethods(allowedMethod);
corRule.setAllowedOrigins(allowedOrigin);
corRule.setAllowedHeaders(allowedHeader);
corRule.setExposeHeaders(exposedHeader);
//指定浏览器对特定资源的预取(OPTIONS)请求返回结果的缓存时间，单位为秒。
corRule.setMaxAgeSeconds(10);
//最多允许10条规则
putCorsRules.add(corRule);
request.setCorsRules(putCorsRules);
oss.setBucketCORS(request);
```

说明：

- 每个存储空间最多只能使用10条规则。
- AllowedOrigins和AllowedMethods都能够最多支持一个*通配符。*表示对于所有的域来源或者操作都满足。

- 而AllowedHeaders和ExposeHeaders不支持通配符。

获取CORS规则

我们可以参考存储空间的CORS规则，通过GetBucketCors方法。代码如下：

```
ArrayList<CORSRule> corsRules;
//获得CORS规则列表
corsRules =  (ArrayList<CORSRule>) oss.getBucketCORSRules(bucketName);
for (CORSRule rule : corsRules) {
    for (String allowedOrigin1 : rule.getAllowedOrigins()) {
        //获得允许跨域请求源
        System.out.println(allowedOrigin1);
    }
    for (String allowedMethod1 : rule.getAllowedMethods()) {
        //获得允许跨域请求方法
        System.out.println(allowedMethod1);
    }
    if (rule.getAllowedHeaders().size() > 0){
        for (String allowedHeader1 : rule.getAllowedHeaders()) {
            //获得允许头部列表
            System.out.println(allowedHeader1);
        }
    }
    if (rule.getExposeHeaders().size() > 0) {
        for (String exposeHeader : rule.getExposeHeaders()) {
        //获得允许头部
        System.out.println(exposeHeader);
        }
    }
    if ( null != rule.getMaxAgeSeconds()) {
        System.out.println(rule.getMaxAgeSeconds());
    }
}
```

删除CORS规则

用于关闭指定存储空间对应的CORS并清空所有规则。

```
// 清空bucket的CORS规则
oss.deleteBucketCORSRules(bucketName);
```

1.2.1.12 设置访问日志

介绍

OSS允许用户对Bucket设置访问日志记录，设置之后对于Bucket的访问会被记录成日志，日志存储在OSS上由用户指定的Bucket中，文件的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中TargetPrefix由用户指定。日志规则由以下2项组成：

- target_bucket，存放日志文件的Bucket

- target_prefix，保存访问日志文件前缀

。

开启Bucket日志

通过OSSClient.setBucketLogging来开启日志功能：

```
SetBucketLoggingRequest request = new SetBucketLoggingRequest("sourceBucket");
request.setTargetBucket("targetBucket");
request.setTargetPrefix("targetPrefix");
ossClient.setBucketLogging(request);
```

查看Bucket日志设置

通过OSSClient.getBucketLogging来查看日志设置：

```
BucketLoggingResult result = ossClient.getBucketLogging("sourceBucket");
System.out.println(result.getTargetBucket());
System.out.println(result.getTargetPrefix());
```

关闭Bucket日志

通过OSSClient.setBucketLogging来关闭日志功能：

```
SetBucketLoggingRequest request = new SetBucketLoggingRequest("sourceBucket");
request.setTargetBucket(null);
request.setTargetPrefix(null);
ossClient.setBucketLogging(request);
```

1.2.1.13 静态网站托管

OSS 允许用户将自己的域名指向OSS服务的地址。这样用户访问他的网站的时候，实际上是在访问OSS的Bucket。对于网站，需要指定首页（index）和出错页（error）分别对应的Bucket中的文件名。

设置托管页面

通过OSSClient.setBucketWebsite来设置托管页面：

```
SetBucketWebsiteRequest request = new SetBucketWebsiteRequest("bucketName");
request.setIndexDocument("index.html");
request.setErrorDocument("error.html");
ossClient.setBucketWebsite(request);
```

查看托管页面

通过OSSClient.getBucketWebsite来查看托管页面：

```
BucketWebsiteResult result = ossClient.getBucketWebsite("bucketName");
System.out.println(result.getIndexDocument());
```

```
System.out.println(result.getErrorDocument());
```

清除托管页面

通过OSSClient.deleteBucketWebsite来清除托管页面：

```
ossClient.deleteBucketWebsite("bucketName");
```

1.2.1.14 防盗链

为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

设置Referer白名单

通过下面代码设置Referer白名单：

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);
List<String> refererList = new ArrayList<String>();
// 添加referer项
refererList.add("http://www.aliyun.com");
refererList.add("http://www.*.com");
refererList.add("http://www.*.aliyuncs.com");
// 允许referer字段为空，并设置Bucket Referer列表
BucketReferer br = new BucketReferer(true, refererList);
client.setBucketReferer(bucketName, br);
```

说明：

Referer参数支持通配符*和？。

获取Referer白名单

```
// 获取Bucket Referer列表
br = client.getBucketReferer(bucketName);
refererList = br.getRefererList();
for (String referer : refererList) {
    System.out.println(referer);
}
```

```
输出结果示例：
http://www.aliyun.com
http://www.*.com"
http://www.?.aliyuncs.com
```

清空Referer白名单

Referer白名单不能直接清空，只能通过重新设置来覆盖之前的规则。

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);
// 默认允许referer字段为空，且referer白名单为空。
BucketReferer br = new BucketReferer();
```

```
client.setBucketReferer(bucketName, br);
```

1.2.1.15 跨区域复制

跨区域复制是跨不同OSS数据中心的Bucket自动、异步地复制Object，它会将对源Bucket中的对象的改动（新建、覆盖、删除等）同步到目标Bucket。该功能能够很好的提供Bucket跨区域容灾或满足用户数据复制的需求。目标Bucket中的对象是源Bucket中对象的精确副本，它们具有相同的对象名、元数据以及内容（例如，创建时间、拥有者、用户定义的元数据、Object ACL、对象内容等）。

开启跨区域复制

通过OSSClient.addBucketReplication开启跨区域复制：

```
AddBucketReplicationRequest request = new AddBucketReplicationRequest("bucketName");
request.setReplicationRuleID("ruleId");
request.setTargetBucketName("targetBucketName");
request.setTargetBucketLocation("oss-cn-qingdao");
ossClient.addBucketReplication(request);
```

说明：

开启跨区域复制，默认会同步历史数据。如果不需要同步历史数据，使用AddBucketReplication Request.setEnableHistoricalObjectReplication(false)禁止历史数据同步。

查看跨区域复制

通过OSSClient.getBucketReplication查看bucket上开启的跨区域复制：

```
List<ReplicationRule> rules = ossClient.getBucketReplication("bucketName");
for (ReplicationRule rule : rules) {
    System.out.println(rule.getReplicationRuleID());
    System.out.println(rule.getTargetBucketLocation());
    System.out.println(rule.getTargetBucketName());
```

```
}
```

删除跨区域复制

通过OSSClient.deleteBucketReplication删除已开启的跨区域复制，删除后目标bucket和object依然存在：

```
ossClient.deleteBucketReplication("bucketName", "ruleId");
```

查看跨区域复制进度

复制进度分为历史数据同步进度、实时数据同步进度。历史数据的同步用百分比表示，如0.80表示完成了80%，仅对开启了历史数据同步的Bucket有效。实时数据同步用新写入数据的时间点表示，表示这个时间点之前的数据已同步完成。

通过OSSClient.deleteBucketReplication查看跨区域复制进度：

```
BucketReplicationProgress process = ossClient.getBucketReplicationProgress("bucketName", "
repRuleID");
System.out.println(process.getReplicationRuleID());
// 是否开启了历史数据同步
System.out.println(process.isEnableHistoricalObjectReplication());
// 历史数据同步进度
System.out.println(process.getHistoricalObjectProgress());
// 实时数据同步进度
System.out.println(process.getNewObjectProgress());
```

查看目标数据中心

通过OSSClient.getBucketReplicationLocation获取Bucket所在的数据中心可同步到的数据中心：

```
List<String> locations = ossClient.getBucketReplicationLocation("bucketName");
for (String loc : locations) {
    System.out.println(loc);
}
```

1.2.1.16 图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

图片处理基础功能

OSS图片处理提供以下功能：

- 获取图片信息

- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义
- 通过级联处理调用多个图片处理功能

图片处理使用

图片处理使用标准的 HTTP GET 请求来访问，所有的处理参数是编码在 URL 中的QueyString。

匿名访问

如果图片文件（Object）的访问权限是 **公共读** ，如下表所示的权限，则可以匿名访问图片服务。

Bucket权限	Object权限
公共读私有写（public-read）或公共读写（public-read-write）	默认（default）
任意权限	公共读私有写（public-read）或公共读写（public-read-write）

通过如下格式的三级域名匿名访问图片处理：

http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value

- bucket：用户的存储空间（bucket）名称
- endpoint：用户存储空间所在数据中心的访问域名
- object：用户上传在OSS上的图片文件
- image：图片处理保留标志符
- action：用户对图片做的操作，如缩放、裁剪、旋转等
- parame：用户对图片做的操作所对应的参数

例如：

http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100

自定义样式，使用如下格式的三级域名匿名访问图片处理：

http://bucket.<endpoint>/object?x-oss-process=x-oss-process=style/name

- style：用户自定义样式系统保留标志符

- name：自定义样式名称，即控制台定义样式的 **规则名**

例如：

http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=style/oss-pic-style-w-100

通过级联处理，可以对一张图片顺序实施多个操作，格式如下：

http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value/action,parame_value/...

例如：

http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100/rotate,90

图片服务也支持HTTPS访问，例如：

http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100

授权访问

对私有权限的文件（Object），如下表所示的权限，必须通过授权才能访问图片服务。

Bucket权限	Object权限
私有读写（private）	默认权限（default）
任意权限	私有读写（private）

生成带签名的图片处理的URL代码如下：

```
    String endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
    String accessKeyId = "<accessKeyId>";
    String accessKeySecret = "<accessKeySecret>";
    String bucketName = "<bucketName>";
    private static String key = "example.jpg";
    OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
    // 图片处理样式
    String style = "image/resize,m_fixed,w_100,h_100/rotate,90";
    // 过期时间10分钟
    Date expiration = new Date(new Date().getTime() + 1000 * 60 * 10 );
    GeneratePresignedUrlRequest req = new GeneratePresignedUrlRequest(bucketName, key,
 HttpMethod.GET);
    req.setExpiration(expiration);
    req.setProcess(style);
    URL signedUrl = ossClient.generatePresignedUrl(req);
    System.out.println(signedUrl);
```

说明：

- 授权访问支持 **自定义样式**、 **HTTPS**、**级联处理**
- 指定过期时间请使用 Date expiration = DateUtil.parseRfc822Date("Wed, 21 Dec 2022 14:20:00 GMT");

SDK访问

对于任意权限的图片文件，都可以直接使用 **SDK** 访问图片、进行处理。

说明：

- 图片处理的完整代码请参考：*GitHub*
- SDK处理图片文件支持 **自定义样式**、 **HTTPS**、**级联处理**

基础操作

图片处理的基础操作包括，获取图片信息、格式转换、缩放、裁剪、旋转、效果、水印等。

```
String endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
String accessKeyId = "<accessKeyId>";
String accessKeySecret = "<accessKeySecret>";
String bucketName = "<bucketName>";
String key = "example.jpg";
OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
// 缩放
String style = "image/resize,m_fixed,w_100,h_100";
GetObjectRequest request = new GetObjectRequest(bucketName, key);
request.setProcess(style);
ossClient.getObject(request, new File("example-resize.jpg"));
// 裁剪
style = "image/crop,w_100,h_100,x_100,y_100,r_1";
request = new GetObjectRequest(bucketName, key);
request.setProcess(style);
ossClient.getObject(request, new File("example-crop.jpg"));
// 旋转
style = "image/rotate,90";
request = new GetObjectRequest(bucketName, key);
request.setProcess(style);
ossClient.getObject(request, new File("example-rotate.jpg"));
// 锐化
style = "image/sharpen,100";
request = new GetObjectRequest(bucketName, key);
request.setProcess(style);
ossClient.getObject(request, new File("example-sharpen.jpg"));
// 水印
style = "image/watermark,text_SGVsbG8g5Zu-54mH5pyN5YqhIQ";
request = new GetObjectRequest(bucketName, key);
request.setProcess(style);
ossClient.getObject(request, new File("example-watermark.jpg"));
// 格式转换
style = "image/format,png";
request = new GetObjectRequest(bucketName, key);
request.setProcess(style);
ossClient.getObject(request, new File("example-format.png"));
// 图片信息
```

```
    style = "image/info";
    request = new GetObjectRequest(bucketName, key);
    request.setProcess(style);
    ossClient.getObject(request, new File("example-info.txt"));
    ossClient.shutdown();
```

自定义样式

```
    String endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
    String accessKeyId = "<accessKeyId>";
    String accessKeySecret = "<accessKeySecret>";
    String bucketName = "<bucketName>";
    String key = "example.jpg";
    OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
    // 自定义样式
    String style = "style/oss-pic-style-w-100";
    GetObjectRequest request = new GetObjectRequest(bucketName, key);
    request.setProcess(style);
    ossClient.getObject(request, new File("example-new.jpg"));
    ossClient.shutdown();
```

级联处理

```
    String endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
    String accessKeyId = "<accessKeyId>";
    String accessKeySecret = "<accessKeySecret>";
    String bucketName = "<bucketName>";
    private static String key = "example.jpg";
    OSSClient ossClient = new OSSClient(endpoint, accessKeyId, accessKeySecret);
    // 级联处理
    String style = "image/resize,m_fixed,w_100,h_100/rotate,90";
    GetObjectRequest request = new GetObjectRequest(bucketName, key);
    request.setProcess(style);
    ossClient.getObject(request, new File("example-new.jpg"));
    ossClient.shutdown();
```

图片处理工具

- 可视化图片处理工具*ImageStyleViever* ，可以直观的看到OSS图片处理的结果
- OSS图片处理的功能、使用演示 页面

1.2.1.17 异常处理

调用OSSClient类的相关接口时，**如果抛出异常**，**则表明操作失败**，**否则操作成功。抛出异常时**，**方法返回的数据无效。** OSS Java SDK包含两类异常，一类是服务器端异常OSSException，另一类是客户端异常ClientException，它们均继承自RuntimeException。

异常处理示例

```
try {
    // Do some operations with the instance here, such as put object...
    client.putObject(...);
} catch (OSSException oe) {
    System.out.println("Caught an OSSException, which means your request made it to OSS, "
```

```
            + "but was rejected with an error response for some reason.");
    System.out.println("Error Message: " + oe.getErrorCode());
    System.out.println("Error Code:       " + oe.getErrorCode());
    System.out.println("Request ID:      " + oe.getRequestId());
    System.out.println("Host ID:           " + oe.getHostId());
} catch (ClientException ce) {
    System.out.println("Caught an ClientException, which means the client encountered "
            + "a serious internal problem while trying to communicate with OSS, "
            + "such as not being able to access the network.");
    System.out.println("Error Message: " + ce.getMessage());
} finally {
    if (client != null) {
        client.shutdown();
    }
}
```

ClientException

ClientException表示客户端尝试向OSS发送请求以及数据传输时遇到的异常。例如，当发送请求时网络连接不可用时，则会抛出 ClientException；当上传文件时发生IO异常时，也会抛出ClientException。

OSSException

OSSException指服务器端错误，它来自于对服务器错误信息的解析，包含OSS会返回给用户相应的错误码和错误信息，便于用户定位问题，并做出适当的处理。

OSSException通常包含以下错误信息：

- Code： OSS返回给用户的错误码。
- Message： OSS提供的详细错误信息。
- RequestId：用于唯一标识该请求的UUID；当您无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助。
- HostId：用于标识访问的OSS集群，与用户请求时使用的Host一致。

OSS常见错误码

错误码	描述	HTTP状态码
AccessDenied	拒绝访问	403
BucketAlreadyExists	Bucket已经存在	409
BucketNotEmpty	Bucket不为空	409
EntityTooLarge	实体过大	400
EntityTooSmall	实体过小	400
FileGroupTooLarge	文件组过大	400

错误码	描述	HTTP状态码
FilePartNotExist	文件Part不存在	400
FilePartStale	文件Part过时	400
InvalidArgument	参数格式错误	400
InvalidAccessKeyId	AccessKeyId不存在	403
InvalidBucketName	无效的Bucket名字	400
InvalidDigest	无效的摘要	400
InvalidObjectName	无效的Object名字	400
InvalidPart	无效的Part	400
InvalidPartOrder	无效的part顺序	400
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket	400
InternalError	OSS内部发生错误	500
MalformedXML	XML格式非法	400
MethodNotAllowed	不支持的方法	405
MissingArgument	缺少参数	411
MissingContentLength	缺少内容长度	411
NoSuchBucket	Bucket不存在	404
NoSuchKey	文件不存在	404
NoSuchUpload	Multipart Upload ID不存在	404
NotImplemented	无法处理的方法	501
PreconditionFailed	预处理错误	412
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟	403
RequestTimeout	请求超时	400
SignatureDoesNotMatch	签名错误	403
InvalidEncryptionAlgorithmError	指定的熵编码加密算法错误	400

1.2.1.18 常见问题

包冲突

如果您在使用OSS Java SDK时，报如下或类似错误：

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/http/ssl/TrustStrategy
    at com.aliyun.oss.OSSClient.<init>(OSSClient.java:268)
    at com.aliyun.oss.OSSClient.<init>(OSSClient.java:193)
    at com.aliyun.oss.demo.HelloOSS.main(HelloOSS.java:77)
Caused by: java.lang.ClassNotFoundException: org.apache.http.ssl.TrustStrategy
    at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
    ... 3 more
```

或

```
Exception in thread "main" java.lang.NoSuchFieldError: INSTANCE
 at org.apache.http.impl.io.DefaultHttpRequestWriterFactory.<init>(DefaultHttpRequestWr
iterFactory.java:52)
 at org.apache.http.impl.io.DefaultHttpRequestWriterFactory.<init>(DefaultHttpRequestWr
iterFactory.java:56)
 at org.apache.http.impl.io.DefaultHttpRequestWriterFactory.<clinit>(DefaultHttpRequestWr
iterFactory.java:46)
 at org.apache.http.impl.conn.ManagedHttpClientConnectionFactory.<init>(ManagedHtt
pClientConnectionFactory.java:82)
 at org.apache.http.impl.conn.ManagedHttpClientConnectionFactory.<init>(ManagedHtt
pClientConnectionFactory.java:95)
 at org.apache.http.impl.conn.ManagedHttpClientConnectionFactory.<init>(ManagedHtt
pClientConnectionFactory.java:104)
 at org.apache.http.impl.conn.ManagedHttpClientConnectionFactory.<clinit>(ManagedHtt
pClientConnectionFactory.java:62)
 at org.apache.http.impl.conn.PoolingHttpClientConnectionManager$InternalConnectionFactory
.<init>(PoolingHttpClientConnectionManager.java:572)
 at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.<init>(PoolingHtt
pClientConnectionManager.java:174)
 at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.<init>(PoolingHtt
pClientConnectionManager.java:158)
 at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.<init>(PoolingHtt
pClientConnectionManager.java:149)
 at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.<init>(PoolingHtt
pClientConnectionManager.java:125)
 at com.aliyun.oss.common.comm.DefaultServiceClient.createHttpClientConnectionManager(
DefaultServiceClient.java:237)
 at com.aliyun.oss.common.comm.DefaultServiceClient.<init>(DefaultServiceClient.java:78)
 at com.aliyun.oss.OSSClient.<init>(OSSClient.java:268)
 at com.aliyun.oss.OSSClient.<init>(OSSClient.java:193)
 at OSSManagerImpl.upload(OSSManagerImpl.java:42)
 at OSSManagerImpl.main(OSSManagerImpl.java:63)
```

您的工程里可能有包冲突。原因是，OSS Java SDK使用了Apache httpclient 4.4.1，**您的工程使用了与Apache httpclient 4.4.1冲突的Apache httpclient或commons-httpclient。**请在您的工程目

录下执行“mvn dependency:tree”，查看工程使用的包及版本。如上述发生错误的工程里，使用了Apache httpclient 4.3：

```
[INFO] --- maven-dependency-plugin:2.2:tree (default-cli) @ maven-demo ---
[INFO] com.aliyun.oss:maven-demo:jar:0.1.1-SNAPSHOT
[INFO] +- junit:junit:jar:4.10:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.1:test
[INFO] +- org.apache.httpcomponents:httpclient:jar:4.3:compile
[INFO] |  +- org.apache.httpcomponents:httpcore:jar:4.3:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.1.3:compile
[INFO] |  \- commons-codec:commons-codec:jar:1.6:compile
[INFO] \- com.aliyun.oss:aliyun-sdk-oss:jar:2.2.1:compile
[INFO]    +- org.jdom:jdom:jar:1.1:compile
[INFO]    \- net.sf.json-lib:json-lib:jar:jdk15:2.4:compile
[INFO]       +- commons-beanutils:commons-beanutils:jar:1.8.0:compile
[INFO]       +- commons-collections:commons-collections:jar:3.2.1:compile
[INFO]       +- commons-lang:commons-lang:jar:2.5:compile
[INFO]       \- net.sf.ezmorph:ezmorph:jar:1.0.6:compile
```

包冲突有以下两种解决方法：

- 使用统一版本。如果您的工程里使用与Apache httpclient 4.4.1冲突的版本，请您也使用4.4.1版本。在pom.xml去掉其他版本的Apache httpclient依赖。如果您的工程使用了commons-httpclient也可能存在冲突，请去除commons-httpclient。
- 解除依赖冲突。如果您的工程依赖与多个第三方包，而第三方包又依赖不同版本的Apache httpclient，您的工程里会有依赖冲突，请使用exclusion解除。详细请参考*maven guides*。

OSS Java SDK依赖以下版本的包，冲突解决办法与httpclient类似，不再赘述。

```
[INFO] com.aliyun.oss:maven-demo:jar:0.1.1-SNAPSHOT
[INFO] +- junit:junit:jar:4.10:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.1:test
[INFO] \- com.aliyun.oss:aliyun-sdk-oss:jar:2.2.1:compile
[INFO]    +- org.apache.httpcomponents:httpclient:jar:4.4.1:compile
[INFO]    |  +- org.apache.httpcomponents:httpcore:jar:4.4.1:compile
[INFO]    |  +- commons-logging:commons-logging:jar:1.2:compile
[INFO]    |  \- commons-codec:commons-codec:jar:1.9:compile
[INFO]    +- org.jdom:jdom:jar:1.1:compile
[INFO]    \- net.sf.json-lib:json-lib:jar:jdk15:2.4:compile
[INFO]       +- commons-beanutils:commons-beanutils:jar:1.8.0:compile
[INFO]       +- commons-collections:commons-collections:jar:3.2.1:compile
[INFO]       +- commons-lang:commons-lang:jar:2.5:compile
[INFO]       \- net.sf.ezmorph:ezmorph:jar:1.0.6:compile
```

缺少包

编译/运行OSS Java SDK报如下错误：

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/http/auth/Credentials
    at com.aliyun.oss.OSSClient.<init>(OSSClient.java:268)
    at com.aliyun.oss.OSSClient.<init>(OSSClient.java:193)
    at com.aliyun.oss.demo.HelloOSS.main(HelloOSS.java:76)
```

```
Caused by: java.lang.ClassNotFoundException: org.apache.http.auth.Credentials
        at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
        ... 3 more
```

或

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/http/protocol/
HttpContext
        at com.aliyun.oss.OSSClient.<init>(OSSClient.java:268)
        at com.aliyun.oss.OSSClient.<init>(OSSClient.java:193)
        at com.aliyun.oss.demo.HelloOSS.main(HelloOSS.java:76)
Caused by: java.lang.ClassNotFoundException: org.apache.http.protocol.HttpContext
        at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
        ... 3 more
```

或

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/jdom/input/SAXBuilder
        at com.aliyun.oss.internal.ResponseParsers.getXmlRootElement(ResponseParsers.java:
645)
        at ... ...
        at com.aliyun.oss.OSSClient.doesBucketExist(OSSClient.java:471)
        at com.aliyun.oss.OSSClient.doesBucketExist(OSSClient.java:465)
        at com.aliyun.oss.demo.HelloOSS.main(HelloOSS.java:82)
Caused by: java.lang.ClassNotFoundException: org.jdom.input.SAXBuilder
        at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
        ... 11 more
```

等类似错误，说明您的工程**缺少OSS Java SDK编译或运行必须的包**。OSS Java SDK依赖下列包：

- aliyun-sdk-oss-2.2.1.jar
- hamcrest-core-1.1.jar
- jdom-1.1.jar
- commons-codec-1.9.jar
- httpclient-4.4.1.jar

- commons-logging-1.2.jar
- httpcore-4.4.1.jar
- log4j-1.2.15.jar

其中，log4j-1.2.15.jar是可选的，需要日志功能的时加入该包。其他包都是必不可少的。

解决办法：您的工程中在加入OSS Java SDK依赖的包，加入方法如下：

- 如果您的工程是Eclipse。请参考Java-SDK使用手册，**安装** > **方式二**：**在Eclipse项目中导入工程依赖的包**；
- 如果您的工程是Ant。请把OSS Java SDK依赖的包，放入工程lib目录；
- 如果您直接使用javac/java，请使用-classpath/-cp指定OSS Java SDK依赖的包路径，或把OSS Java SDK依赖的包放入classpath路经下。

连接超时

运行OSS Java SDK程序抛出如下异常：

```
com.aliyun.oss.ClientException: SocketException
    at com.aliyun.oss.common.utils.ExceptionFactory.createNetworkException(ExceptionFactory
.java:71)
    at com.aliyun.oss.common.comm.DefaultServiceClient.sendRequestCore(DefaultSer
viceClient.java:116)
    at com.aliyun.oss.common.comm.ServiceClient.sendRequestImpl(ServiceClient.java:121)
    at com.aliyun.oss.common.comm.ServiceClient.sendRequest(ServiceClient.java:67)
    at com.aliyun.oss.internal.OSSOperation.send(OSSOperation.java:92)
    at com.aliyun.oss.internal.OSSOperation.doOperation(OSSOperation.java:140)
    at com.aliyun.oss.internal.OSSOperation.doOperation(OSSOperation.java:111)
    at com.aliyun.oss.internal.OSSBucketOperation.getBucketInfo(OSSBucketOperation.java:
1152)
    at com.aliyun.oss.OSSClient.getBucketInfo(OSSClient.java:1220)
    at com.aliyun.oss.OSSClient.getBucketInfo(OSSClient.java:1214)
    at com.aliyun.oss.demo.HelloOSS.main(HelloOSS.java:94)
Caused by: org.apache.http.conn.HttpHostConnectException: Connect to oss-test.oss-cn-
hangzhou-internal.aliyuncs.com:80 [oss-test.oss-cn-hangzhou-internal.aliyuncs.com/10.84.135.
99] failed: Connection timed out: connect
    at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHtt
pClientConnectionOperator.java:151)
    at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHtt
pClientConnectionManager.java:353)
    at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:380)
    at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236)
    at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:184)
    at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)
    at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:184)
    at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:82)
    at com.aliyun.oss.common.comm.DefaultServiceClient.sendRequestCore(DefaultSer
viceClient.java:113)
```

```
    ... 9 more
```

原因是**endpoint错误或者网络不通**，如果不能直接找出错误。请使用OSSProbe工具检测，OSSProbe会给出可能的错误原因。

org.apache.http.NoHttpResponseException: The target server failed to respond

运行OSS Java SDK程序抛出如下异常：

```
com.aliyun.oss.ClientException: Unknown
        at com.aliyun.oss.common.utils.ExceptionFactory.createNetworkException(ExceptionFactory.java:68) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.common.comm.DefaultServiceClient.sendRequestCore(DefaultServiceClient.java:115) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.common.comm.ServiceClient.sendRequestImpl(ServiceClient.java:121) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.common.comm.ServiceClient.sendRequest(ServiceClient.java:67) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.internal.OSSOperation.send(OSSOperation.java:92) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.internal.OSSOperation.doOperation(OSSOperation.java:140) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.internal.OSSOperation.doOperation(OSSOperation.java:111) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.internal.OSSMultipartOperation.initiateMultipartUpload(OSSMultipartOperation.java:206) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.aliyun.oss.OSSClient.initiateMultipartUpload(OSSClient.java:765) ~[aliyun-sdk-oss-2.1.0.jar:na]
        at com.taobao.agoo.dump.client.OssTools.multipartUpload(OssTools.java:79) ~[agoo-dump-client-2.0.0-SNAPSHOT.jar:na]
        at com.taobao.agoo.dump.biz.manager.TaskExcutorManager$UploadTask.run(TaskExcutorManager.java:114) ~[agoo-dump-biz-2.0.0-SNAPSHOT.jar:na]
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471) [na:1.7.0_51]
        at java.util.concurrent.FutureTask.run(FutureTask.java:262) [na:1.7.0_51]
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145) [na:1.7.0_51]
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615) [na:1.7.0_51]
        at java.lang.Thread.run(Thread.java:744) [na:1.7.0_51]
Caused by: org.apache.http.NoHttpResponseException: The target server failed to respond
        at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:143) ~[httpclient-4.4.jar:4.4]
        at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:57) ~[httpclient-4.4.jar:4.4]
        at org.apache.http.impl.io.AbstractMessageParser.parse(AbstractMessageParser.java:261) ~[httpcore-4.4.jar:4.4]
        at org.apache.http.impl.DefaultBHttpClientConnection.receiveResponseHeader(DefaultBHttpClientConnection.java:165) ~[httpcore-4.4.jar:4.4]
        at org.apache.http.impl.conn.CPoolProxy.receiveResponseHeader(CPoolProxy.java:167) ~[httpclient-4.4.jar:4.4]
        at org.apache.http.protocol.HttpRequestExecutor.doReceiveResponse(HttpRequestExecutor.java:272) ~[httpcore-4.4.jar:4.4]
```

原因是：重用连接前没有检测连接是否有效，过期的连接重用会导致上述错误。该问题是Apache httpclient 4.4的bug，详见*HTTPCLIENT-1609*，4.4.1及以后版本修复。Java SDK 2.1.2前的版本使用的是Apache httpclient 4.4，存在上述问题；Java SDK 2.1.2及以后的版本，使用的是Apache httpclient 4.4.1，修复了该问题。**如果发现该问题请升级OSS Java SDK到2.1.2及以后版本。**

调用OSS Java SDK夯住

调用OSS Java SDK夯住(hang)，通过`jstack -l <pid>`查看堆栈，夯在如下的位置：

```
"main" prio=6 tid=0x000000000291e000 nid=0xc40 waiting on condition [0x0000000002dae000
]
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for  <0x00000007d85697f8> (a java.util.concurrent.locks.AbstractQu
euedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQu
eueuedSynchronizer.java:2043)
    at org.apache.http.pool.PoolEntryFuture.await(PoolEntryFuture.java:138)
    at org.apache.http.pool.AbstractConnPool.getPoolEntryBlocking(AbstractConnPool.java:306)
    at org.apache.http.pool.AbstractConnPool.access$000(AbstractConnPool.java:64)
    at org.apache.http.pool.AbstractConnPool$2.getPoolEntry(AbstractConnPool.java:192)
    at org.apache.http.pool.AbstractConnPool$2.getPoolEntry(AbstractConnPool.java:185)
    at org.apache.http.pool.PoolEntryFuture.get(PoolEntryFuture.java:107)
    at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.leaseConnection(
PoolingHttpClientConnectionManager.java:276)
    at org.apache.http.impl.conn.PoolingHttpClientConnectionManager$1.get(PoolingHtt
pClientConnectionManager.java:263)
    at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:190)
    at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:184)
    at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)
    at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:184)
    at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:82)
```

```
    at com.aliyun.oss.common.comm.DefaultServiceClient.sendRequestCore(DefaultSer
viceClient.java:113)
    at com.aliyun.oss.common.comm.ServiceClient.sendRequestImpl(ServiceClient.java:123)
    at com.aliyun.oss.common.comm.ServiceClient.sendRequest(ServiceClient.java:68)
    at com.aliyun.oss.internal.OSSOperation.send(OSSOperation.java:94)
    at com.aliyun.oss.internal.OSSOperation.doOperation(OSSOperation.java:146)
    at com.aliyun.oss.internal.OSSOperation.doOperation(OSSOperation.java:113)
    at com.aliyun.oss.internal.OSSObjectOperation.getObject(OSSObjectOperation.java:229)
    at com.aliyun.oss.OSSClient.getObject(OSSClient.java:629)
    at com.aliyun.oss.OSSClient.getObject(OSSClient.java:617)
    at samples.HelloOSS.main(HelloOSS.java:49)
```

原因：连接池中连接泄漏，可能是使用OSSClient.getObject，但是没有关闭。请检查您的程序，确保没有连接泄漏。关闭方法如下：

```
// 读取Object
OSSObject ossObject = ossClient.getObject(bucketName, key);
// 使用ObjectContent和ObjectMetadata
// ...
// 使用完毕后，一定要close
ossObject.getObjectContent().close();
```

ConnectionClosedException: Premature end of Content-Length delimited message body ...

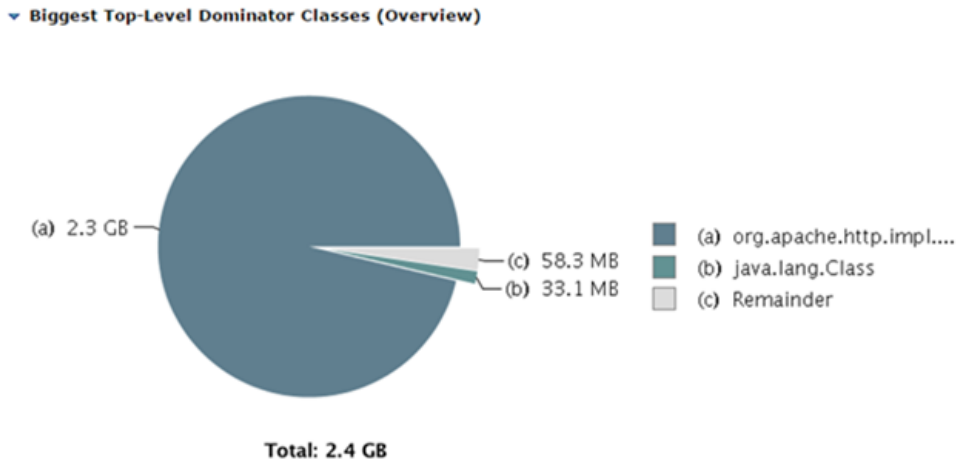
如果您在使用OSSClient.getObject时，报如下或类似错误：

```
Exception in thread "main" org.apache.http.ConnectionClosedException: Premature end of
Content-Length delimited message body (expected: 11990526; received: 202880
    at org.apache.http.impl.io.ContentLengthInputStream.read(ContentLengthInputStream.java:
180)
    at org.apache.http.impl.io.ContentLengthInputStream.read(ContentLengthInputStream.java:
200)
    at org.apache.http.impl.io.ContentLengthInputStream.close(ContentLengthInputStream.java:
103)
    at org.apache.http.impl.execchain.ResponseEntityProxy.streamClosed(ResponseEntityProxy
.java:128)
    at org.apache.http.conn.EofSensorInputStream.checkClose(EofSensorInputStream.java:228)
    at org.apache.http.conn.EofSensorInputStream.close(EofSensorInputStream.java:174)
    at java.io.FilterInputStream.close(FilterInputStream.java:181)
    at java.io.FilterInputStream.close(FilterInputStream.java:181)
    at com.aliyun.oss.event.ProgressInputStream.close(ProgressInputStream.java:147)
    at java.io.FilterInputStream.close(FilterInputStream.java:181)
    at samples.HelloOSS.main(HelloOSS.java:39)
```

原因：两次读取数据间隔时间超过1分钟。OSS会关闭超过1分钟没有发送/接收数据的连接。如果您的应用模式是，读取部分数据—处理数据—读取数据—处理数据，且处理数据的时间不固定。

内存泄露

调用OSS Java SDK的程序，运行一段时间(看业务量，几小时到几天不等)后内存爆掉。请使用内存分析工具，分析内存使用情况。推荐使用 *Eclipse Memory Analyzer(MAT)*，使用方法见 *使用 MAT 进行堆转储文件分析*。如果分析结果类似下图，PoolingHttpClientConnectionManager占96%的内存。



Biggest Top-Level Dominator Classes

Label	Number of Objects	Used Heap Size	Retained Heap Size	Retained Heap, %
org.apache.http.impl.conn.PoolingHttpClientConnectionManager	112,701	3,606,432	2,517,081,408	96.33%
java.lang.Class	6,562	60,792	34,679,768	1.33%
∑ Total: 2 entries	119,263	3,667,224	2,551,761,176	

原因：程序中可能多次new OSSClient，但是没有调用OSSClient.shutdown，造成对象泄漏。提示new OSSClient一定要和OSSClient.shutdown成对使用。

调用OSSClient.shutdown抛异常InterruptedException

调用OSSClient.shutdown抛如下异常：

```
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at com.aliyun.oss.common.comm.IdleConnectionReaper.run(IdleConnectionReaper:76)
```

原因：OSSClient后台线程IdleConnectionReaper，定时关闭闲置连接。IdleConnectionReaper线程在sleep时，调用shutdown，就会抛出上面的异常。OSS Java SDK v2.3.0已经修复该问题。您可以使用如下代码，忽略该异常：

```
try {
serviceClient.shutdown();
} catch(Exception e) {
}
```

1.2.2 Python-SDK

1.2.2.1 安装

相关资源

- *github*项目
- *SDK API*文档：所有的接口，以及类的细节

- *PyPi*主页
- 版本迭代

环境依赖

此版本的Python SDK适用于Python 2.6、2.7、3.3、3.4、3.5版本。首先请根据*python*官网的引导安装合适的Python版本。

安装好Python后：

- 在Linux Shell里验证Python版本：

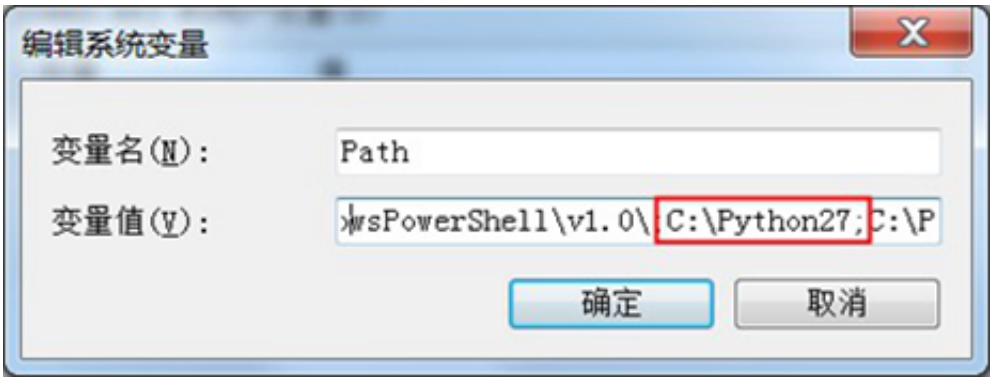
```
$ python -V
Python 2.7.10
```

上面的输出表明您已经成功安装了Python 2.7.10版本。

- Windows CMD环境下验证Python版本：

```
C:\> python -V
Python 2.7.10
```

上面的输出表明您已经成功安装了Python 2.7.10版本。如果提示“不是内部或外部命令”，请检查配置“环境变量”-“Path”，增加Python的安装路径。如图：



安装SDK

- 通过pip安装

```
pip install oss2
```

- 源码安装

从*github*下载相应版本的SDK包，解压后进入目录，确认目录下有setup.py这个文件：

```
python setup.py install
```

- 验证

首先命令行输入python并回车，在Python环境下检查SDK的版本：

```
>>> import oss2
>>> oss2.__version__
'2.0.0'
```

上面的输出表明您已经成功安装了特定版本的Python SDK（这里以版本2.0.0为例）。

卸载SDK

建议通过pip卸载：

```
pip uninstall oss2
```

示例程序

Python SDK的示例代码在examples目录下，地址是 *GitHub* 。示例包括以下内容：

示例文件	示例内容
object_basic.py	展示了文件相关的基本用法，包括上传、下载、罗列、删除等
object_extra.py	展示了文件相关的高级用法，如中文、设置用户自定义元信息、拷贝文件、追加上传等
upload.py	展示了文件上传的高级用法，如断点续传上传、分片上传等
download.py	展示了文件下载的用法，如下载文件、范围下载、断点续传下载等
object_check.py	展示了上传/下载时数据校验的用法，包括MD5和CRC
object_progress.py	展示了进度条功能的用法，包括上传进度条和下载进度条
object_callback.py	展示了上传回调的用法
object_post.py	展示了PostObject的用法，PostObject的实现不依赖于 Python SDK
sts.py	展示了*STS*的用法，包括角色扮演获取临时用户的密钥、使用临时用户的密钥访问OSS
live_channel.py	展示了Live Channel的用法
image.py	展示了图片服务的用法

示例文件	示例内容
bucket.py	展示了Bucket管理操作的用法，诸如创建、删除、列举Bucket等

历史版本

此版本的Python SDK相比于原先的0.4.2版本是不兼容的升级，并且**命令行工具osscmd也不随本次版本发布。**

1.2.2.2 快速入门

介绍

确认您已经理解OSS 基本概念，如Bucket、Object、Endpoint、AccessKeyId和AccessKeySecret等。

下面介绍如何使用OSS Python SDK来访问OSS服务，包括查看Bucket列表，上传文件，下载文件，查看文件列表等。 默认这些程序是写在一个脚本文件里，通过Python程序可以执行。并且，后面的例子可能会依赖于前面的例子。也可以把这些例子粘贴到 Python交互环境进行试验。

说明：
请不要用生产Bucket试验本文档中的例子。

查看Bucket列表

```
# -*- coding: utf-8 -*-

import oss2

auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
service = oss2.Service(auth, '您的Endpoint')

print([b.name for b in oss2.BucketIterator(service)])
```

上面代码中出现的类：

- oss2.Auth对象承载了用户的认证信息，即AccessKeyId和AccessKeySecret等；
- oss2.Service对象用于服务相关的操作，目前就是用来列举Bucket；
- oss2.BucketIterator对象是一个可以遍历用户Bucket信息的迭代器

新建bucket

在杭州区域新建一个私有Bucket：

```
bucket = oss2.Bucket(auth, 'http://oss-cn-hangzhou.aliyuncs.com', '您的bucket名')
bucket.create_bucket(oss2.models.BUCKET_ACL_PRIVATE)
```

其中oss2.Bucket对象用于上传、下载、删除对象，设置Bucket各种配置等。

上传文件

把本地文件local.txt上传到OSS，Object名为remote.txt：

```
bucket.put_object_from_file('remote.txt', 'local.txt')
```

下载文件

把OSS上的Object下载到本地文件：

```
bucket.get_object_to_file('remote.txt', 'local-backup.txt')
```

列举文件

列举Bucket下的10个文件：

```
from itertools import islice

for b in islice(oss2.ObjectIterator(bucket), 10):
    print(b.key)
```

其中oss2.ObjectIterator对象是一个迭代器，您可以像使用其他迭代器一样使用它。

删除文件

```
bucket.delete_object('remote.txt')
```

1.2.2.3 初始化

Python SDK几乎所有的操作都是通过oss2.Service、oss2.Bucket进行的。这里，我们会详细说明如何初始化上述两个类。

确定Endpoint

请先阅读开发人员指南中关于访问域名和自定义访问域名的部分，理解Endpoint相关的概念。

Endpoint可以有以下几种形式：

示例	说明
http://oss-cn-hangzhou.aliyuncs.com	以HTTP协议，公网访问杭州区域的Bucket
https://oss-cn-beijing.aliyuncs.com	以HTTPS协议，公网范围北京区域的Bucket
http://my-domain.com	以HTTP协议，通过用户自定义域名（CNAME）访问某个Bucket

关于CNAME需要注意的是：

- oss2.Service只支持非CNAME的Endpoint
- 自定义域名my-domain.com CNAME到形如 *http://<my-bucket>.oss-cn-hangzhou.aliyuncs.com* 这样的域名

下面的代码设置OSS的访问域名为Endpoint参数：

```
# -*- coding: utf-8 -*-
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
endpoint = 'http://oss-cn-hangzhou.aliyuncs.com' # 假设Bucket处于杭州区域
bucket = oss2.Bucket(auth, endpoint, '您的Bucket名')
```

也可以设置使用自定义域名：

```
# -*- coding: utf-8 -*-
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
cname = 'http://my-domain.com' # 假设您的域名为my-domain.com
bucket = oss2.Bucket(auth, cname, '您的Bucket名', is_cname=True)
```

设置连接超时

可以指定可选connect_time来设定连接超时时间，以秒为单位。下面的代码初始化一个oss2.Service对象，并把连接超时时间设为30秒（oss2.Bucket的初始化是类似的）：

```
# -*- coding: utf-8 -*-
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
```

```
service = oss2.Service(auth, 'http://oss-cn-hangzhou.aliyuncs.com', connect_timeout=30)
```

1.2.2.4 管理存储空间

存储空间（Bucket）是OSS上的命名空间，也是计费、权限控制、日志记录等高级功能的管理实体。

查看所有Bucket

通过oss2.BucketIterator可以遍历所有的Bucket：

```
# -*- coding: utf-8 -*-

import oss2

auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
service = oss2.Service(auth, '您的Endpoint')

print([b.name for b in oss2.BucketIterator(service)])
```

其中，oss2.Service是用来访问“OSS服务”相关的类，目前只是用来列举用户的Bucket。

创建Bucket

通过指定Endpoint和Bucket名，用户可以在指定的区域创建新的Bucket：

```
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
bucket.create_bucket()
```

比如，把Endpoint设为 http://oss-cn-beijing.aliyuncs.com，就可以在北京区域创建一个Bucket。 关于Endpoint、区域及其对应关系，以及Bucket的命名规范，请参考OSS 基本概念 。

创建时还可以指定Bucket的权限，如下面的代码创建一个公共可读的Bucket：

```
bucket.create_bucket(oss2.BUCKET_ACL_PUBLIC_READ)
```

删除Bucket

用下面的方法删除一个空的Bucket：

```
try:
    bucket.delete_bucket()
except oss2.exceptions.BucketNotEmpty:
    print('bucket is not empty.')
except oss2.exceptions.NoSuchBucket:
    print('bucket does not exist')
```

如果Bucket非空，即还有文件或进行中的分片上传，那么就无法删除，SDK会抛出BucketNotEmpty异常。如果，Bucket不存在，则抛出NoSuchBucket异常。

说明：

- 一旦Bucket被删除，Bucket名可能会被其他用户申请。
- 对于非空Bucket，可以通过边列举边删除（对于分片上传则是终止上传）的方法清空Bucket后，再删除。

查看Bucket访问权限

```
print(bucket.get_bucket_acl().acl)
```

设置Bucket访问权限

把Bucket的访问权限设为私有：

```
bucket.put_bucket_acl(oss2.BUCKET_ACL_PRIVATE)
```

1.2.2.5 上传文件

OSS有多种上传方式，不同的上传方式能够上传的数据大小也不一样。普通上传（PutObject）、追加上传（AppendObject）最多只能上传小于或等于5GB的文件；而分片上传每个分片可以达到5GB，合并后的文件能够达到48.8TB。首先介绍普通上传，我们会详细展示提供数据的各种方式，即方法中的 **data** 参数。其他上传接口有类似的data参数，不再赘述。

普通上传

通过 Bucket.put_object 方法，可以上传一个普通文件。

上传字符串

上传内存中的字符串：

```
# -*- coding: utf-8 -*-
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
bucket.put_object('remote.txt', 'content of object')
```

也可以指定上传的是bytes：

```
bucket.put_object('remote.txt', b'content of object')
```

或是指定为unicode：

```
bucket.put_object('remote.txt', u'content of object')
```

事实上，oss2.Bucket.put_object的第二个参数（参数名为data）可以接受两种类型的字符串：

- bytes：直接上传
- unicode：会自动转换为UTF-8编码的bytes进行上传

上传本地文件

```
with open('local.txt', 'rb') as fileobj:
    bucket.put_object('remote.txt', fileobj)
```

对比上传字符串的代码，注意到数据参数既可以是字符串，也可以是这里的文件对象（file object）。

说明：

必须以二进制的方式打开文件，因为内部实现需要知道文件包含的字节数。

Python SDK还提供了一个便捷的方法完成上面的工作：

```
bucket.put_object_from_file('remote.txt', 'local.txt')
```

上传网络流

```
import requests
input = requests.get('http://www.aliyun.com')
bucket.put_object('aliyun.txt', input)
```

requests.get返回的是一个可迭代对象（iterable），此时Python SDK会通过Chunked Encoding方式上传。

返回值

```
result = bucket.put_object('remote.txt', 'content of object')
print('http status: {0}'.format(result.status))
print('request_id: {0}'.format(result.request_id))
print('ETag: {0}'.format(result.etag))
print('date: {0}'.format(result.headers['date']))
```

每个OSS服务器返回的响应都有共同属性：

- status：HTTP返回码
- request_id：请求ID
- headers：HTTP响应头部

etag则是put_object返回值特有的属性。

说明：

请求ID唯一标识了一次请求，强烈建议把它作为程序日志的一部分。

小结

从上面的示例可以发现，Python SDK上传方法可以接受多种类型的输入源，这主要得益于第三方的requests库。小结一下，输入数据（data参数）可以有如下几种类型：

- bytes字符串
- unicode字符串：自动转换为UTF-8编码的bytes进行上传
- 文件对象（file object）：必须以二进制方式打开（如”rb”模式）
- 可迭代对象（iterable）：以Chunked Encoding的方式上传

说明：
对于文件对象，如果是可以seek和tell的，那么会从文件当前位置开始上传，直到文件结束。

断点续传

当需要上传的本地文件很大，或网络状况不够理想，往往会出现上传到中途就失败了。此时，如果对已经上传的数据重新上传，既浪费时间，又占用了网络资源。Python SDK提供了一个易用性接口 `oss2.resumable_upload` ，用于断点续传本地文件：

```
oss2.resumable_upload(bucket, 'remote.txt', 'local.txt')
```

其内部实现是当文件长度大于或等于可选参数 **multipart_threshold** 时，就进行分片上传。此时，会在HOME目录下建立.py-oss-upload目录，并把当前进度保存在其下的某个文件中。用户也可以通过可选参数store来指定保存进度的目录。

下面是一个完全定制化的例子：

```
oss2.resumable_upload(bucket, 'remote.txt', 'local.txt',
    store=oss2.ResumableStore(root='/tmp'),
    multipart_threshold=100*1024,
    part_size=100*1024,
    num_threads=4)
```

含义是

- **ResumableStore** 指定把进度保存到 */tmp/.py-oss-upload* 目录下
- **multipart_threshold** 指明只要文件长度不小于100KB就进行分片上传
- **part_size** 参数建议每片大小为100KB。如果文件太大，那么分片大小也可能会大于100KB
- **num_threads** 参数指定并发上传线程数为4

说明：

- 请把 **oss2.defaults.connection_pool_size** 设成大于或等于线程数。
- 要求 2.1.0 及以后版本。

分片上传

采用分片上传，用户可以对上传做更为精细的控制。这适用于诸如预先不知道文件大小、并发上传、自定义断点续传等场景。一次分片上传可以分为三个步骤：

- 初始化（Bucket.init_multipart_upload）：获得Upload ID
- 上传分片（Bucket.upload_part）：这一步可以并发进行
- 完成上传（Bucket.complete_multipart_upload）：合并分片，生成OSS文件

具体例子如下：

```
import os
from oss2 import SizedFileAdapter, determine_part_size
from oss2.models import PartInfo
key = 'remote.txt'
filename = 'local.txt'
total_size = os.path.getsize(filename)
part_size = determine_part_size(total_size, preferred_size=100 * 1024)
# 初始化分片
upload_id = bucket.init_multipart_upload(key).upload_id
parts = []
# 逐个上传分片
with open(filename, 'rb') as fileobj:
    part_number = 1
    offset = 0
    while offset < total_size:
        num_to_upload = min(part_size, total_size - offset)
        result = bucket.upload_part(key, upload_id, part_number,
                                    SizedFileAdapter(fileobj, num_to_upload))
        parts.append(PartInfo(part_number, result.etag))
        offset += num_to_upload
        part_number += 1
# 完成分片上传
bucket.complete_multipart_upload(key, upload_id, parts)
# 验证一下
with open(filename, 'rb') as fileobj:
    assert bucket.get_object(key).read() == fileobj.read()
```

其中：

- **determine_part_size**是一个用来确定分片大小的帮助函数。
- **SizedFileAdapter(fileobj, size)**会生成一个新的file object，起始偏移和原先一样，但最多只能读取size大小。

说明：

三个步骤中的对象名（key）必须一致；上传分片和完成上传中的Upload ID必须一致。

追加上传

可以通过Bucket.append_object方法进行追加上传：

```
result = bucket.append_object('append.txt', 0, 'content of first append')
bucket.append_object('append.txt', result.next_position, 'content of second append')
```

首次上传的偏移量（position参数）设为0。如果文件已经存在，且

- 不是可追加文件，则抛出ObjectNotAppendable异常；
- 是可追加文件，如果传入的偏移和文件当前长度不等，则抛出PositionNotEqualToLength异常。

如果不是首次上传，可以通过Bucket.head_object方法或上次追加返回值的next_position属性，得到偏移参数。

设置HTTP头

上传时，可以通过headers参数设置OSS支持的HTTP头部，如Content-Type：

```
bucket.put_object('a.json', '{"age": 1}', headers={'Content-Type': 'application/json; charset=utf-8
'})
```

设置自定义元信息

通过传入x-oss-meta-为前缀的HTTP头部，可以为文件设置自定义元信息：

```
bucket.put_object('story.txt', 'a novel', headers={'x-oss-meta-author': 'O. Henry'})
```

进度条

上传接口都提供了可选参数progress_callback，用来帮助实现进度条功能。下面的代码以Bucket.put_object为例，实现了一个简单的命令行下的进度显示功能（新开一个Python源文件）：

```
# -*- coding: utf-8 -*-
from __future__ import print_function
import os, sys
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
def percentage(consumed_bytes, total_bytes):
    if total_bytes:
        rate = int(100 * (float(consumed_bytes) / float(total_bytes)))
        print('\r{0}% '.format(rate), end='')
        sys.stdout.flush()
bucket.put_object('story.txt', 'a'*1024*1024, progress_callback=percentage)
```

说明：

- 当无法确定待上传的数据长度时，progress_callback的第二个参数（total_bytes）为None。

- 进度条的完整示例代码请参看 *GitHub* 。

上传回调

put_object、put_object_from_file、complete_multipart_upload提供提供了上传回调功能。下面的代码以Bucket.put_object为例，实现了一个简单的上传回调功能：

```
# -*- coding: utf-8 -*-
import json
import base64
import os
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
# 准备回调参数
callback_dict = {}
callback_dict['callbackUrl'] = 'http://oss-***.aliyuncs.com:23450'
callback_dict['callbackHost'] = 'oss-***.aliyuncs.com'
callback_dict['callbackBody'] = 'filename=${object}&size=${size}&mimeType=${mimeType}'
callback_dict['callbackBodyType'] = 'application/x-www-form-urlencoded'
# 回调参数是json格式，并且需要base64编码
callback_param = json.dumps(callback_dict).strip()
base64_callback_body = base64.b64encode(callback_param)
# 回调参数编码后放在header中传给oss
headers = {'x-oss-callback': base64_callback_body}
# 上传并回调
result = bucket.put_object('story.txt', 'a'*1024*1024, headers)
```

说明：

- 上传回调的详细说明请参看上传回调 。
- 上传回调的完整示例代码请参看 *GitHub* 。

1.2.2.6 下载文件

介绍

Python SDK提供了两个基本的下载接口：

- `Bucket.get_object` ：它的返回值是一个类文件对象（file-like object），同时也是一个可迭代对象（iterable）
- `Bucket.get_object_to_file` ：直接下载到本地文件

此外，还提供了一个易用性接口：

- `oss2.resumable_download`：帮助用户进行断点续传、并行下载。

流式下载

下面的例子一次性读取OSS文件，并打印出来：

```
# -*- coding: utf-8 -*-
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
remote_stream = bucket.get_object('remote.txt')
print(remote_stream.read())
```

既然是类文件对象，我们就可以方便的使用一些库函数，如下载到本地文件：

```
import shutil
remote_stream = bucket.get_object('remote.txt')
with open('local-backup.txt', 'wb') as local_fileobj:
    shutil.copyfileobj(remote_stream, local_fileobj)
```

由于返回值又是一个可迭代对象，所以可以把它流式的拷贝到另一个Object：

```
remote_stream = bucket.get_object('remote.txt')
bucket.put_object('remote-backup.txt', remote_stream)
```

下载到本地文件

下面的代码把OSS上的remote.txt文件，下载到当前目录下的local-backup.txt。

```
bucket.get_object_to_file('remote.txt', 'local-backup.txt')
```

指定下载范围

通过可选参数 **byte_range**，可以指定下载的范围。byte_range是一个tuple，表示范围的起止字节。下面的代码会下载**前100个字节**的数据：

```
remote_stream = bucket.get_object('remote.txt', byte_range=(0, 99))
```

说明：

byte_range表示的是字节偏移量的闭区间，字节偏移从0开始计。如(0, 99)表示从第0个字节到第99个字节（包含在内），共计100个字节的数据。

进度条

下载接口提供了可选参数 **progress_callback**，用来帮助实现进度条功能。下面的代码实现了一个简单的命令行下的进度显示功能（新建一个Python源文件）：

```
# -*- coding: utf-8 -*-
from __future__ import print_function
import os, sys
import oss2
```

```
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
def percentage(consumed_bytes, total_bytes):
    if total_bytes:
        rate = int(100 * (float(consumed_bytes) / float(total_bytes)))
        print('\r{0}% '.format(rate), end='')
        sys.stdout.flush()
bucket.get_object_to_file('remote.txt', 'local-backup.txt', progress_callback=percentage)
```

说明：

- 当待HTTP响应头部没有Content-Length头时，progress_callback的第二个参数（total_bytes）为None。
- 进度条的完整示例代码请参看 *GitHub* 。

断点续传

当需要下载的文件很大，或网络状况不够理想，往往下载到中途就失败了。如果下次重试，还需要重新下载，就会浪费时间和带宽。为此，Python SDK 提供了一个易用性接口 oss2.resumable_download 用于断点续传。

下面的代码把OSS文件remote.txt下载到本地当前目录，并重命名为local.txt。

```
oss2.resumable_download(bucket, 'remote.txt', 'local.txt')
```

断点续传的过程大致如下：

1. 在本地创建一个临时文件，文件名由原始文件名加上一个随机的后缀组成；
2. 通过指定HTTP请求的 Range 头，按照范围读取OSS文件，并写入到临时文件里相应的位置；
3. 下载完成之后，把临时文件重名为目标文件。

在上述过程中，断点信息，即已经下载的范围等信息，会保存在本地磁盘上。如果因为某种原因下载中断了，后续重试本次下载，就会读取断点信息，然后只下载缺失的部分。

下面是一个完全定制化的例子：

```
oss2.resumable_download(bucket, 'remote.txt', 'local.txt',
  store=oss2.ResumableDownloadStore(root='/tmp'),
  multiget_threshold=20*1024*1024,
  part_size=10*1024*1024,
  num_threads=3)
```

含义是

- **ResumableDownloadStore** 指定把断点信息保存到 */tmp/.py-oss-download* 目录下
- **multiget_threshold** 指明当文件长度不小于20MB时，就采用分范围下载

- **part_size** 建议每次下载10MB。如果文件太大，那么实际的值会大于指定值
- **num_threads** 指定并发下载线程数为3

使用该函数应注意如下细节：

- 对同样的源文件、目标文件，避免多个程序（线程）同时调用该函数。因为断点信息会在磁盘上互相覆盖，或临时文件名会冲突。
- 避免使用太小的范围（分片），即 **part_size** 参数不宜过小，建议大于或等于 **oss2.defaults.multiget_part_size** 。
- 如果目标文件已经存在，那么该函数会覆盖此文件。

说明：

- 请把 **oss2.defaults.connection_pool_size** 设成大于或等于线程数。
- 要求 2.1.0 及以后版本。

1.2.2.7 管理文件

通过Python SDK，用户可以罗列、删除、拷贝文件，也可以查看文件信息，更改文件元信息等。

罗列文件

Python SDK提供了一系列的迭代器，用于列举文件、分片上传等。

简单罗列

列举Bucket里的10个文件：

```
# -*- coding: utf-8 -*-
import oss2
from itertools import islice
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
for b in islice(oss2.ObjectIterator(bucket), 10):
    print(b.key)
```

按前缀罗列

只列举前缀为**img-**的所有文件：

```
for obj in oss2.ObjectIterator(bucket, prefix='img-'):
    print(obj.key)
```

模拟文件夹功能

OSS的存储空间（Bucket）本身是扁平结构的，并没有文件夹或目录的概念。用户可以通过在文件名里加入/来模拟文件夹。在列举的时候，则要设置delimiter参数（目录分隔符）为/，并通过是否为“公共前缀”来判断是否为文件夹。

罗列**根目录**下的所有内容：

```
for obj in oss2.ObjectIterator(bucket, delimiter='/'):
    if obj.is_prefix():  # 文件夹
        print('directory: ' + obj.key)
    else:                # 文件
        print('file: ' + obj.key)
```

说明：

模拟罗列文件夹这个操作比较低效，不建议使用。

判断文件是否存在

通过 object_exists 判断文件是否存在。返回值为 **true** 文件存在，为 **false** 文件不存在。

```
exist = bucket.object_exists('remote.txt')
if exist:
    print('object exist')
else:
    print('object not eixst')
```

删除文件

删除单个文件：

```
bucket.delete_object('remote.txt')
```

也可以删除多个文件（不能超过1000个）。下面的代码删除三个文件，同时打印成功删除的文件名：

```
result = bucket.batch_delete_objects(['a.txt', 'b.txt', 'c.txt'])
print('\n'.join(result.deleted_keys))
```

拷贝文件

把Bucket名为src-bucket下的source.txt拷贝到当前Bucket的target.txt文件。

```
bucket.copy_object('src-bucket', 'source.txt', 'target.txt')
```

拷贝大文件

当文件比较大时，建议使用分片拷贝的方式进行拷贝，可以避免因文件太大而超时。分片拷贝和分片上传类似，分成三步：

1. 初始化（Bucket.init_multipart_upload）：得到Upload ID
2. 拷贝分片（Bucket.upload_part_copy）：把源文件的一部分拷贝成目标文件的一个分片
3. 完成分片（Bucket.complete_multipart_copy）：完成分片拷贝，生成目标文件

请参考下面的示例：

```
from oss2.models import PartInfo
from oss2 import determine_part_size
src_key = 'remote.txt'
dst_key = 'remote-dst.txt'
bucket.put_object(src_key, 'a' * (1024 * 1024  + 100))
total_size = bucket.head_object(src_key).content_length
part_size = determine_part_size(total_size, preferred_size=100 * 1024)
# 初始化分片
upload_id = bucket.init_multipart_upload(dst_key).upload_id
parts = []
# 逐个分片拷贝
part_number = 1
offset = 0
while offset < total_size:
    num_to_upload = min(part_size, total_size - offset)
    byte_range = (offset, offset + num_to_upload - 1)
    result = bucket.upload_part_copy(bucket.bucket_name, src_key, byte_range,
                                     dst_key, upload_id, part_number)
    parts.append(PartInfo(part_number, result.etag))
    offset += num_to_upload
    part_number += 1
# 完成分片上传
bucket.complete_multipart_upload(dst_key, upload_id, parts)
```

更改文件元信息

更改用户自定义元信息：

```
bucket.update_object_meta('story.txt', {'x-oss-meta-author': 'O. Henry'})
bucket.update_object_meta('story.txt', {'x-oss-meta-price': '100 dollar'})
```

对于用户自定义元信息（x-oss-meta-为前缀的HTTP头部），每次调用都会覆盖以前的值。就上面的例子来说，第二次调用实际上是删除了x-oss-meta-author这个自定义元信息。

也可以更改Content-Type等信息：

```
bucket.update_object_meta('story.txt', {'Content-Type': 'text/plain'})
```

注意到这次调用不但修改了Content-Type，而且把原先设置的用户自定义元信息也给清除了。

查看文件访问权限

```
print(bucket.get_object_acl('story.txt').acl)
```

说明：

文件的访问权限有四种：default（默认）、private（私有读写）、public-read（公共读私有写）、public-read-write（公共读写）

设置文件访问权限

把文件的访问权限设为公共读私有写：

```
bucket.put_object_acl('story.txt', oss2.OBJECT_ACL_PUBLIC_READ)
```

说明：

文件的访问权限有四种：default（默认）、private（私有读写）、public-read（公共读私有写）、public-read-write（公共读写），分别对应：

- oss2.OBJECT_ACL_DEFAULT
- oss2.OBJECT_ACL_PRIVATE
- oss2.OBJECT_ACL_PUBLIC_READ
- oss2.OBJECT_ACL_PUBLIC_READ_WRITE

1.2.2.8 授权访问

使用私有链接下载

对于私有Bucket，可以生成私有链接（又称为“签名URL”）供用户访问：

```
# -*- coding: utf-8 -*-
import oss2
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
print(bucket.sign_url('GET', 'object-in-bucket.txt', 60))
```

上面的代码会打印出一个私有链接，可以把该链接分享给其他用户，直接用浏览器或者wget之类的工具下载。这个链接只在生成之时起60秒内有效。

使用STS服务临时授权

OSS用户可以通过阿里云STS服务（Security Token Service）进行临时授权访问。

使用STS时请按以下步骤进行：

1. 在官网控制台创建子账号。
2. 在官网控制台创建STS角色并赋予子账号扮演角色的权限。
3. 使用子账号的AccessKeyId/AccessKeySecret向STS申请临时token。
4. 使用临时token中的认证信息创建 `StsAuth` 类实例。

5. 使用 StsAuth 类实例初始化 Bucket 类实例。

下面我们给出一个完整的例子。首先安装官方的Python STS客户端：

```
$ pip install aliyun-python-sdk-sts
```

接下来，通过STS服务获取临时授权。下面代码中的 **end_point**、 **bucket_name**、**access_key_id**、**access_key_secret**、**role_arn** 需要用户根据实际情况，填写正确的值。并且我们假设，要扮演的角色是有上传文件权限的。

其中 **role_arn** 是角色的资源描述符。

```
# -*- coding: utf-8 -*-
from aliyunsdkcore import client
from aliyunsdksts.request.v20150401 import AssumeRoleRequest
import json
import oss2
endpoint = 'oss-cn-hangzhou.aliyuncs.com'
bucket_name = '<待访问的Bucket名>'
access_key_id = '<子账号AccessKeyId>'
access_key_secret = '<子账号AccessKeySecret>'
role_arn = '<角色的资源描述符>'
clt = client.AcsClient(access_key_id, access_key_secret, 'cn-hangzhou')
req = AssumeRoleRequest.AssumeRoleRequest()
# 为了简化讨论，这里没有设置Duration、Policy等，更多细节请参考RAM、STS的相关文档。
req.set_accept_format('json')  # 设置返回值格式为JSON
req.set_RoleArn(role_arn)
req.set_RoleSessionName('session-name')
body = clt.do_action(req)
# 为了简化讨论，没有做出错检查
token = json.loads(body)
# 初始化StsAuth实例
auth = oss2.StsAuth(token['Credentials']['AccessKeyId'],
                    token['Credentials']['AccessKeySecret'],
                    token['Credentials']['SecurityToken'])
# 初始化Bucket实例
bucket = oss2.Bucket(auth, endpoint, bucket_name)
# 上传一个字符串
bucket.put_object('object-name.txt', b'hello world')
```

说明：

- 临时token会在一段时间后过期，这就要在适当的时刻，重新获取token，并设置 **oss2.Bucket** 中的 **auth** 成员变量为新的 **StsAuth**。
- 要求 2.0.6 及以后版本。
- STS应用完整的示例代码请参看 *GitHub* 。

1.2.2.9 静态网站托管

用户可以通过Python SDK把自己的Bucket配置成静态网站托管模式。配置生效后，可以把OSS作为一个静态网站来进行访问，并且能够自动跳转到索引页和错误页面。

设置静态网站托管

下面的代码开启静态网站托管模式，并把索引页面设置为index.html，错误页面（404页面）设置为error.html：

```
# -*- coding: utf-8 -*-

import oss2
from oss2.models import BucketWebsite

auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')

bucket.put_bucket_website(BucketWebsite('index.html', 'error.html'))
```

获取静态网站托管配置

```
try:
    website = bucket.get_bucket_website()
    print('Index file is {0}, error file is {1}'.format(website.index_file, website.error_file))
except oss2.exceptions.NoSuchWebsite as e:
    print('Website is not configured, request_id={0}'.format(e.request_id))
```

注意到当静态网站托管模式没有开启时，get_bucket_website会抛出NoSuchWebsite异常。

关闭静态网站托管模式

```
bucket.delete_bucket_website()
```

1.2.2.10 生命周期管理

生命周期规则

用户可以为自己的Bucket设置生命周期规则，来管理器中的文件。目前，用户可以通过规则来删除相匹配的文件。每条规则都由如下几个部分组成：

- 规则ID，用于表示一条规则，不可以和别的规则重复
- Object名称前缀，只有匹配该前缀的Object才适用这个规则。前缀之间不能重叠，如 `/home` 和 `/home/user` 是不合法的。因为前者是后者的前缀
- 操作，用户希望对匹配的Object所执行的操作。
- 过期天数，指定距文件最后修改时间多少天之后删除
- 是否生效

说明：

生命周期涉及到删除用户数据，请务必仔细阅读相关文档，并在测试Bucket上进行试验后，再在生产Bucket中使用。

设置生命周期规则

以下示例设置了两条规则：

- 规则一：规则ID是**rule1**；前缀是**tests/**；状态是**启用**；过期天数是**356天**；
- 规则二：规则ID是**rule2**；前缀是**logging-**；状态是**关闭**；过期天数是**1天**；

```
# -*- coding: utf-8 -*-
import oss2
from oss2.models import LifecycleExpiration, LifecycleRule, BucketLifecycle
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
rule1 = LifecycleRule('rule1', 'tests/',
                      status=LifecycleRule.ENABLED,
                      expiration=LifecycleExpiration(days=356))
rule2 = LifecycleRule('rule2', 'logging-',
                      status=LifecycleRule.DISABLED,
                      expiration=LifecycleExpiration(days=1))
bucket.put_bucket_lifecycle(BucketLifecycle([rule1, rule2]))
```

获取生命周期规则

```
lifecycle = bucket.get_bucket_lifecycle()
for rule in lifecycle.rules:
    print('id={0}, prefix={1}, status={2}, days={3}, date={4}'
          .format(rule.id, rule.prefix, rule.status, rule.expiration.days, rule.expiration.date))
```

删除生命周期规则

删除Bucket所有的生命周期规则，即关闭生命周期功能：

```
bucket.delete_bucket_lifecycle()
# 再次获取就会抛出异常
try:
    lifecycle = bucket.get_bucket_lifecycle()
except oss2.exceptions.NoSuchLifecycle:
```

```
    print('lifecycle is not configured')
```

1.2.2.11 跨域资源共享

CORS允许web端的应用程序访问不属于本域的资源。OSS提供接口方便开发者控制跨域访问的权限。

设定CORS规则

下面的代码设置了一条CORS规则：

```
# -*- coding: utf-8 -*-
import oss2
from oss2.models import BucketCors, CorsRule
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
rule = CorsRule(allowed_origins=['*'],
                allowed_methods=['GET', 'HEAD'],
                allowed_headers=['*'],
                max_age_seconds=1000)
bucket.put_bucket_cors(BucketCors([rule]))
```

获取CORS规则

```
try:
    cors = bucket.get_bucket_cors()
except oss2.exceptions.NoSuchCors:
    print('cors is not set')
else:
    for rule in cors.rules:
        print('AllowedOrigins={0}'.format(rule.allowed_origins))
        print('AllowedMethods={0}'.format(rule.allowed_methods))
        print('AllowedHeaders={0}'.format(rule.allowed_headers))
        print('ExposeHeaders={0}'.format(rule.expose_headers))
        print('MaxAgeSeconds={0}'.format(rule.max_age_seconds))
```

删除CORS规则

```
bucket.delete_bucket_cors()
```

1.2.2.12 设置访问日志

访问日志简介

用户可以通过设置Bucket的访问日志配置，把对该Bucket的访问日志保存在指定的Bucket中，以供后续的分析。访问日志以文件的形式存在于指定的Bucket中，每小时会生成一个文本文件。文件名的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中**TargetPrefix**由用户在配置中指定。

日志配置由如下部分组成：

- TargetBucket：目标Bucket名，生成的日志文件会保存到这个Bucket中。
- TargetPrefix：日志文件名前缀，可以为空。

开启日志功能

下面的代码开启日志功能，且把日志保存在当前Bucket，日志文件名前缀为 **logging/** ：

```
# -*- coding: utf-8 -*-
import oss2
from oss2.models import BucketLogging
auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')
bucket.put_bucket_logging(BucketLogging(bucket.bucket_name, 'logging/'))
```

查看日志设置

```
logging = bucket.get_bucket_logging()
print('TargetBucket={0}, TargetPrefix={1}'.format(logging.target_bucket, logging.target_prefix))
```

关闭日志功能

```
bucket.delete_bucket_logging()
```

1.2.2.13 防盗链

OSS是按使用收费的服务，为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

设置防盗链

```
# -*- coding: utf-8 -*-

import oss2
from oss2.models import BucketReferer

auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')

bucket.put_bucket_referer(BucketReferer(True, ['http://aliyun.com', 'http://*.aliyuncs.com']))
```

上面的代码成功执行后，防盗链的配置如下：

- 第一个参数 **allow_empty_referer** 为**True**表示允许空的Referer；也可以根据实际需要，设成 **False**。
- Referer白名单为 *http://aliyun.com*，或能够通配*http://*.aliyuncs.com*。

获取防盗链设置

```
config = bucket.get_bucket_referer()
print('allow empty referer={0}, referers={1}'.format(config.allow_empty_referer, config.referers))
```

关闭防盗链

要关闭防盗链功能，只要设置成允许空Referer访问，以及清空Referer白名单。

```
bucket.put_bucket_referer(BucketReferer(True, []))
```

1.2.2.14 出错处理

介绍

在程序运行过程中，如果遇到错误，Python SDK会抛出相应的异常。 一共有三类异常：ClientError、RequestError和ServerError，它们都是OssError的子类。这些异常都在oss2.exceptions子模块中定义。

OssError一些重要的成员变量如下：

- status：int类型。对于ServerError就是HTTP状态码；对于另外两类异常，该值为固定值。
- request_id：str类型。对于ServerError就是OSS服务器返回请求ID；对于另外两类异常，该值为空字符串。
- code和message：str类型。就是OSS的错误响应格式里的Code和Message两个XML Tag中的文本。

ClientError

ClientError是因用户的输入有误引起的。比如，Bucket.batch_delete_objects当收到空的文件名列表时，就会抛出该异常。 ClientError对象的status值是固定的oss2.exceptions.OSS_CLIENT_ERROR_STATUS。

RequestError

当底层的HTTP库抛出异常时，Python SDK会将其转换为RequestError。这些异常对象的status值是固定的oss2.exceptions.OSS_REQUEST_ERROR_STATUS。

ServerError

当OSS服务器返回4xx或5xx的HTTP错误码时，Python SDK会将OSS Server的响应转换为ServerError。 为了方便使用，根据status和code，还派生出了一些子类：

异常类	对应的HTTP状态码	OSS错误码	备注
NotModified	304	空	没有修改
AccessDenied	403	AccessDenied	拒绝访问
NoSuchBucket	404	NoSuchBucket	Bucket不存在
NoSuchKey	404	NoSuchKey	文件名不存在
NoSuchUpload	404	NoSuchUpload	分片上传不存在
NoSuchWebsite	404	NoSuchWebsiteConfiguration	静态网站托管未配置
NoSuchLifecycle	404	NoSuchLifecycle	生命周期管理未配置
NoSuchCors	404	NoSuchCORSConfiguration	CORS未配置
BucketNotEmpty	409	BucketNotEmpty	Bucket非空
PositionNotEqualToLength	409	PositionNotEqualToLength	Append的位置和文件长度不相等
ObjectNotAppendable	409	ObjectNotAppendable	不是可追加文件

另外，所有404状态码的异常都是NotFound的子类；所有409状态码的异常都是Conflict的子类。

说明：

不是所有的OSS错误码都有对应的异常。目前只定义了比较常用的一些。

示例

下面的代码展示了下载一个文件时，如何处理文件名不存在的情形，并打印出HTTP状态码和请求ID：

```
# -*- coding: utf-8 -*-

import oss2

auth = oss2.Auth('您的AccessKeyId', '您的AccessKeySecret')
bucket = oss2.Bucket(auth, '您的Endpoint', '您的Bucket名')

try:
    stream = bucket.get_object('random-key.txt')
except oss2.exceptions.NoSuchKey as e:
```

```
    print('status={0}, request_id={1}'.format(e.status, e.request_id))
```

1.2.2.15 中文和时间

中文

为了讨论的便利，先对即将用到的名词进行界定和描述：

名词	描述
str	Python缺省的字符串类型。Python 2.x中是bytes类型；Python 3.x中是unicode类型
bytes	字节流，其长度就是字节数。如 **b'中文'** 的长度取决于编码，如果是UTF-8编码，则为6
unicode	unicode流，其长度是字符数，如 **u'中文'** 的长度是 2

输入、输出类型约定

Python SDK中有三类输入参数：

输入参数	建议类型	备注
OSS文件名	str	如果是bytes，则要求是UTF-8编码
本地文件名	str, unicode	如果是bytes，则要求是UTF-8编码
输入数据流	bytes	如Bucket.put_object的 **data** 参数

其中“本地文件名”指的是诸如Bucket.get_object_to_file里的本地文件名参数。

Python SDK还有两类输出：

输出	类型
解析XML得到的结果	str
下载内容	bytes

其中**解析XML得到的结果**指的是诸如Bucket.list_objects、Bucket.get_bucket_lifecycle等接口得到的结果中的字符串。

由于Python SDK默认认为bytes类型是经过UTF-8编码的，请**确保Python源文件也是UTF-8编码的**。

帮助函数

Python SDK提供了三个函数，帮助用户做类型转换：

函数	描述
to_bytes	把unicode类型转换为UTF-8编码的bytes；其他类型，则原值返回
to_unicode	把UTF-8编码的bytes转换为unicode；其他类型，则原值返回
to_string	Python 2.x中相当于to_bytes；Python 3.x相当于to_unicode

时间

Python SDK会把从服务器获得的时间戳字符串都转换为 *Unix Time*，即 自1970年1月1日UTC零点以来的秒数。比如Bucket.get_object结果中的last_modified就是一个int类型的Unix Time。

如果想得到datetime.datetime这样的类型，可以通过datetime.datetime.fromtimestamp()等方法转换。

1.2.2.16 图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处理服务。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

图片处理基础功能

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义
- 通过级联处理调用多个图片处理功能

图片处理使用

图片处理使用标准的 HTTP GET 请求来访问，所有的处理参数是编码在 URL 中的QueyString。

匿名访问

如果图片文件（Object）的访问权限是 公共读，如下表所示的权限，则可以匿名访问图片服务。

Bucket权限	Object权限
公共读私有写（public-read）或公共读写（public-read-write）	默认（default）
任意权限	公共读私有写（public-read）或 公共读写（public-read-write）

通过如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value
```

- bucket：用户的存储空间（bucket）名称
- endpoint：用户存储空间所在数据中心的访问域名
- object：用户上传在OSS上的图片文件
- image：图片处理保留标志符
- action：用户对图片做的操作，如缩放、裁剪、旋转等
- parame：用户对图片做的操作所对应的参数

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,
w_100
```

自定义样式，使用如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=x-oss-process=style/name
```

- style：用户自定义样式系统保留标志符
- name：自定义样式名称，即控制台定义样式的 **规则名**

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=style/oss-pic-style-w-100
```

通过级联处理，可以对一张图片顺序实施多个操作，格式如下：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value/action,parame_value/...
```

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100/rotate,90
```

图片服务也支持HTTPS访问，例如：

```
https://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100
```

授权访问

对私有权限的文件（Object），如下表所示的权限，必须通过授权才能访问图片服务。

Bucket权限	Object权限
私有读写（private）	默认权限（default）
任意权限	私有读写（private）

生成带签名的图片处理的URL代码如下：

```
# -*- coding: utf-8 -*-
import oss2
endpoint = '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
access_key_id = '<access_key_id>'
access_key_secret = '<access_key_secret>'
bucket_name = '<bucket_name>'
key = 'example.jpg'
# 创建Bucket对象，所有Object相关的接口都可以通过Bucket对象来进行
bucket = oss2.Bucket(oss2.Auth(access_key_id, access_key_secret), endpoint, bucket_name)
# 上传示例图片
bucket.put_object_from_file(key, 'example.jpg')
# 生成带签名的URL，过期时间10分钟
style = 'image/resize,m_fixed,w_100,h_100/rotate,90'
url = bucket.sign_url('GET', key, 10 * 60, params={'x-oss-process': style})
print(url)
```

说明：

- 授权访问支持 **自定义样式**、**HTTPS** 、**级联处理**。

- **sign_url** 过期时间单位是**秒**。

SDK访问

对于任意权限的图片文件，都可以直接使用 SDK 访问图片、进行处理。

说明：

- 图片处理的完整代码请参考：*GitHub*
- SDK处理图片文件支持 **自定义样式**、**HTTPS** 、**级联处理**。

基础操作

图片处理的基础操作包括，获取图片信息、格式转换、缩放、裁剪、旋转、效果、水印等。

```
# -*- coding: utf-8 -*-
import os
import oss2
endpoint = '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
access_key_id = '<access_key_id>'
access_key_secret = '<access_key_secret>'
bucket_name = '<bucket_name>'
key = 'example.jpg'
new_pic = 'example-new.jpg'
# 创建Bucket对象，所有Object相关的接口都可以通过Bucket对象来进行
bucket = oss2.Bucket(oss2.Auth(access_key_id, access_key_secret), endpoint, bucket_name)
# 上传示例图片
bucket.put_object_from_file(key, 'example.jpg')
# 图片缩放
style = 'image/resize,m_fixed,w_100,h_100'
bucket.get_object_to_file(key, new_pic, process=style)
# 图片裁剪
style = 'image/crop,w_100,h_100,x_100,y_100,r_1'
bucket.get_object_to_file(key, new_pic, process=style)
# 图片旋转
style = 'image/rotate,90'
bucket.get_object_to_file(key, new_pic, process=style)
# 图片锐化
style = 'image/sharpen,100'
bucket.get_object_to_file(key, new_pic, process=style)
# 图片加文字水印
style = 'image/watermark,text_SGVsbG8g5Zu-54mH5pyN5YqhIQ'
bucket.get_object_to_file(key, new_pic, process=style)
# 图片格式转换
style = 'image/format,png'
bucket.get_object_to_file(key, new_pic, process=style)
# 删除示例图片
bucket.delete_object(key)
# 清除本地文件
os.remove(new_pic)
```

说明：

get_object也支持图片处理功能。

自定义样式

```
# -*- coding: utf-8 -*-
import os
import oss2
endpoint = '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
access_key_id = '<access_key_id>'
access_key_secret = '<access_key_secret>'
bucket_name = '<bucket_name>'
key = 'example.jpg'
new_pic = 'example-new.jpg'
# 创建Bucket对象，所有Object相关的接口都可以通过Bucket对象来进行
bucket = oss2.Bucket(oss2.Auth(access_key_id, access_key_secret), endpoint, bucket_name)
# 上传示例图片
bucket.put_object_from_file(key, 'example.jpg')
# 自定义样式
style = 'style/oss-pic-style-w-100'
# 图片处理
bucket.get_object_to_file(key, new_pic, process=style)
# 删除示例图片
bucket.delete_object(key)
# 清除本地文件
os.remove(new_pic)
```

说明：

get_object也支持图片处理功能。

级联处理

```
# -*- coding: utf-8 -*-
import os
import oss2
endpoint = '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
access_key_id = '<access_key_id>'
access_key_secret = '<access_key_secret>'
bucket_name = '<bucket_name>'
key = 'example.jpg'
new_pic = 'example-new.jpg'
# 创建Bucket对象，所有Object相关的接口都可以通过Bucket对象来进行
bucket = oss2.Bucket(oss2.Auth(access_key_id, access_key_secret), endpoint, bucket_name)
# 上传示例图片
bucket.put_object_from_file(key, 'example.jpg')
# 级联处理
style = 'image/resize,m_fixed,w_100,h_100/rotate,90'
# 图片处理
bucket.get_object_to_file(key, new_pic, process=style)
# 删除示例图片
bucket.delete_object(key)
# 清除本地文件
os.remove(new_pic)
```

说明：

get_object也支持图片处理功能。

图片处理工具

- 可视化图片处理工具 *ImageStyleViever* ，可以直观的看到OSS图片处理的结果。
- OSS图片处理的功能、使用演示 页面。

1.2.3 Android-SDK

1.2.3.1 前言

本文档主要介绍OSS Android SDK的安装和使用。本文档假设您已经开通了阿里云OSS 服务，并创建了AccessKeyId 和AccessKeySecret。文中的ID 指的是AccessKeyId，KEY 指的是AccessKeySecret。如果您还没有开通或者还不了解OSS，请登录*OSS产品主页*获取更多的帮助。

环境要求

- Android系统版本：2.3及以上
- 必须注册有Aliyun.com用户账户，并开通OSS服务。

SDK下载

- Android SDK开发包(2016-09-15) 版本号 2.3.0：*aliyun_OSS_Android_SDK_20160915*
- github地址：点击查看
- sample地址：点击查看
- javadoc地址：点击查看

1.2.3.2 安装

直接引入jar包

当您下载了OSS Android SDK 的 zip 包后，进行以下步骤（对Android studio 或者 Eclipse 都适用）：

- 在官网下载 sdk 包
- 解压后得到 jar 包，目前包括 aliyun-oss-sdk-android-2.3.0.jar、okhttp-3.x.x.jar 和 okio-1.x.x.jar
- 将以上 3 个 jar 包导入 libs 目录

Maven依赖

```
<dependency>
    <groupId>com.aliyun.dpa</groupId>
    <artifactId>oss-android-sdk</artifactId>
```

```
    <version>2.3.0</version>
</dependency>
```

权限设置

以下是 OSS Android SDK 所需要的 Android 权限，请确保您的 AndroidManifest.xml 文件中已经配置了这些权限，否则，SDK 将无法正常工作。

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-permission>
```

混淆设置

在混淆配置中加入：

```
-keep class com.alibaba.sdk.android.oss.** { *; }
-dontwarn okio.**
-dontwarn org.apache.commons.codec.binary.**
```

对 SDK 中同步接口、异步接口的一些说明

考虑到移动端开发场景下不允许在UI线程执行网络请求的编程规范，SDK大多数接口都提供了同步、异步两种调用方式，同步接口调用后会阻塞等待结果返回，而异步接口需要在请求时传入回调函数，请求的执行结果将在回调中处理。

同步接口不能在UI线程调用。遇到异常时，将直接抛出ClientException或者ServiceException异常，前者指本地遇到的异常如网络异常、参数非法等；后者指OSS返回的服务异常，如鉴权失败、服务器错误等。

异步请求遇到异常时，异常会在回调函数中处理。

此外，调用异步接口时，函数会直接返回一个Task，Task可以取消、等待直到完成、或者直接获取结果。如：

```
OSSAsyncTask task = oss.asyncGetObejct(...);
task.cancel(); // 可以取消任务
task.waitUntilFinished(); // 等待直到任务完成
GetObjectResult result = task.getResult(); // 阻塞等待结果返回
```

接口支持同步和异步两种调用方式，考虑到简洁性，本文档中只有部分重要接口会同时提供同步、异步两种调用的示例，其他接口暂时以异步调用的示例为主。

1.2.3.3 初始化

OSSClient 是 OSS 服务的 Android 客户端，它为调用者提供了一系列的方法，可以用来操作，管理存储空间（bucket）和文件（object）等。在使用 SDK 发起对 OSS 的请求前，您需要初始化一个 OSSClient 实例，并对它进行一些必要设置。

确定 Endpoint

Endpoint 是阿里云 OSS 服务在各个区域的地址，目前支持以下两种形式：

Endpoint类型	解释
OSS区域地址	使用OSS Bucket所在区域地址
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS区域地址

使用OSS Bucket所在区域地址，Endpoint查询可以有下面两种方式：

- 查询Endpoint与区域对应关系详情。
- 您可以登录阿里云OSS控制台，进入Bucket概览页，Bucket域名的后缀部分：如 bucket-1.oss-***.aliyuncs.com 的 oss-***.aliyuncs.com 部分为该Bucket的外网Endpoint。

Cname

您可以将自己拥有的域名通过Cname绑定到某个存储空间（bucket）上，然后通过自己域名访问存储空间内的文件。、

比如您要将域名new-image.xxxxx.com绑定到深圳区域的名称为image的存储空间上：

您需要到您的域名xxxxx.com托管商那里设定一个新的域名解析，将 http://new-image.xxxxx.com 解析到 http://image.oss-cn-shenzhen.aliyuncs.com，类型为**CNAME**。

设置EndPoint和凭证

必须设置EndPoint和CredentialProvider：

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// 明文设置secret的方式建议只在测试时使用，更多鉴权模式请参考后面的`访问控制`章节
OSSCredentialProvider credentialProvider = new OSSPlainTextAKSKCredentialProvider("<
accessKeyId>", "<accessKeySecret>");
```

```
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

设置EndPoint为cname

如果您已经在bucket上绑定cname，将该cname直接设置到endPoint即可。如：

```
String endpoint = "http://new-image.xxxxx.com";
OSSCredentialProvider credentialProvider = new OSSPlainTextAKSKCredentialProvider("<
accessKeyId>", "<accessKeySecret>");
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

设置网络参数

您也可以在初始化的时候设置详细的ClientConfiguration：

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// 明文设置secret的方式建议只在测试时使用，更多鉴权模式请参考后面的访问控制章节
OSSCredentialProvider credentialProvider = new OSSPlainTextAKSKCredentialProvider("<
accessKeyId>", "<accessKeySecret>");
ClientConfiguration conf = new ClientConfiguration();
conf.setConnectionTimeout(15 * 1000); // 连接超时，默认15秒
conf.setSocketTimeout(15 * 1000); // socket超时，默认15秒
conf.setMaxConcurrentRequest(5); // 最大并发请求书，默认5个
conf.setMaxErrorRetry(2); // 失败后最大重试次数，默认2次
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider, conf);
```

1.2.3.4 快速入门

本节演示了上传、下载文件的基本流程。更多细节用法可以参考本工程的 test目录（点击查看）或者sample目录（点击查看）。

STEP-1. 初始化OSSClient

初始化主要完成Endpoint设置、鉴权方式设置、Client参数设置。其中，鉴权方式包含明文设置模式、自签名模式、STS鉴权模式。鉴权细节请参考访问控制章节。

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";

// 明文设置secret的方式建议只在测试时使用，更多鉴权模式请参考访问控制章节
OSSCredentialProvider credentialProvider = new
 OSSPlainTextAKSKCredentialProvider("<accessKeyId>", "<accessKeySecret>");

OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

通过OSSClient发起上传、下载请求是线程安全的，您可以并发执行多个任务。

STEP-2. 上传文件

这里假设您已经在控制台上拥有自己的Bucket，这里演示如何从把一个本地文件上传到OSS：

```
// 构造上传请求
```

```
PutObjectRequest put = new PutObjectRequest("<bucketName>", "<objectKey>", "<uploadFile
Path>");

// 异步上传时可以设置进度回调
put.setProgressCallback(new OSSProgressCallback<PutObjectRequest>() {
    @Override
    public void onProgress(PutObjectRequest request, long currentSize, long totalSize) {
        Log.d("PutObject", "currentSize: " + currentSize + " totalSize: " + totalSize);
    }
});

OSSAsyncTask task = oss.asyncPutObject(put, new OSSCompletedCallback<PutObjectR
equest, PutObjectResult>() {
    @Override
    public void onSuccess(PutObjectRequest request, PutObjectResult result) {
        Log.d("PutObject", "UploadSuccess");
    }

    @Override
    public void onFailure(PutObjectRequest request, ClientException clientExcepion, ServiceExc
eption serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});

// task.cancel(); // 可以取消任务

// task.waitUntilFinished(); // 可以等待直到任务完成
```

STEP-3. 下载指定文件

下载一个指定 object，返回数据的输入流，需要自行处理：

```
// 构造下载文件请求
GetObjectRequest get = new GetObjectRequest("<bucketName>", "<objectKey>");

OSSAsyncTask task = oss.asyncGetObject(get, new OSSCompletedCallback<GetObjectR
equest, GetObjectResult>() {
    @Override
    public void onSuccess(GetObjectRequest request, GetObjectResult result) {
        // 请求成功
        Log.d("Content-Length", "" + getResult.getContentLength());

        InputStream inputStream = result.getObjectContent();

        byte[] buffer = new byte[2048];
        int len;

        try {
```

```
            while ((len = inputStream.read(buffer)) != -1) {
                // 处理下载的数据
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onFailure(GetObjectRequest request, ClientException clientExcepion, ServiceException serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});

// task.cancel(); // 可以取消任务

// task.waitUntilFinished(); // 如果需要等待任务完成
```

1.2.3.5 访问控制

访问控制

移动终端是一个不受信任的环境，如果把AccessKeyId和AccessKeySecret直接保存在终端本地用来加签请求，存在极高的风险。为此，SDK提供了建议只在测试时使用的**明文设置模式**，和另外两种依赖于您的业务Server的鉴权模式：**STS鉴权模式**和**自签名模式**。

明文设置模式

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// 明文设置AccessKeyId/AccessKeySecret的方式建议只在测试时使用
OSSCredentialProvider credentialProvider = new OSSPlainTextAKSKCredentialProvider("<accessKeyId>", "<accessKeySecret>");
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

STS鉴权模式

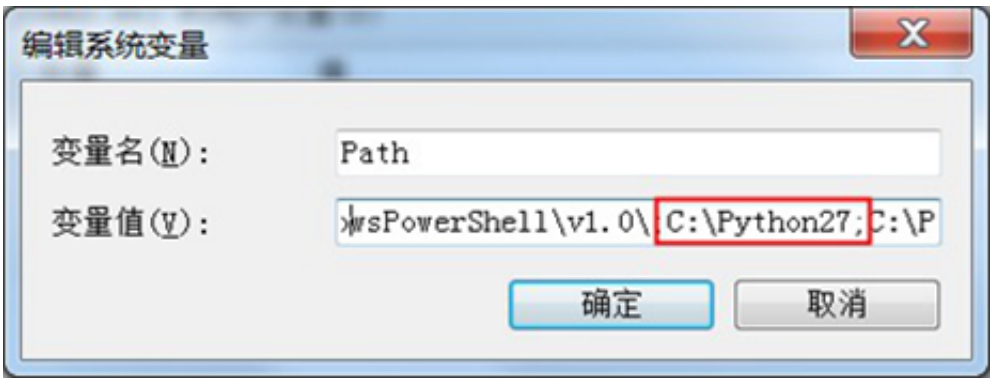
介绍

OSS可以通过阿里云STS服务，临时进行授权访问。阿里云STS (Security Token Service) 是为云计算用户提供临时访问令牌的Web服务。通过STS，您可以为第三方应用或联邦用户（用户身份由您自己管理）颁发一个自定义时效和权限的访问凭证，App端称为FederationToken。第三方应用或联

邦用户可以使用该访问凭证直接调用阿里云产品API，或者使用阿里云产品提供的SDK来访问云产品API。

- 您不需要透露您的长期密钥（AccessKey）给第三方应用，只需要生成一个访问令牌并将令牌交给第三方应用即可。这个令牌的访问权限及有效期限都可以由您自定义。
- 您不需要关心权限撤销问题，访问令牌过期后就自动失效。

以APP应用为例，交互流程如下图：



方案的详细描述如下：

1. App用户登录。App用户身份是客户自己管理。客户可以自定义身份管理系统，也可以使用外部Web账号或OpenID。对于每个有效的App用户来说，AppServer可以确切地定义出每个App用户的最小访问权限。
2. AppServer请求STS服务获取一个安全令牌（SecurityToken）。在调用STS之前，AppServer需要确定App用户的最小访问权限（用Policy语法描述）以及授权的过期时间。然后通过调用STS的AssumeRole（扮演角色）接口来获取安全令牌。角色管理与使用相关内容请参考《*RAM*使用指南》中的角色管理。
3. STS返回给AppServer一个有效的访问凭证，App端称为FederationToken，包括一个安全令牌（SecurityToken）、临时访问密钥（AccessKeyId, AccessKeySecret）以及过期时间。
4. AppServer将FederationToken返回给ClientApp。ClientApp可以缓存这个凭证。当凭证失效时，ClientApp需要向AppServer申请新的有效访问凭证。比如，访问凭证有效期为1小时，那么ClientApp可以每30分钟向AppServer请求更新访问凭证。
5. ClientApp使用本地缓存的FederationToken去请求Aliyun Service API。云服务会感知STS访问凭证，并会依赖STS服务来验证访问凭证，并正确响应用户请求。

直接设置StsToken

您可以在APP中，预先通过某种方式(如通过网络请求从您的业务Server上)获取一对StsToken，然后用它来初始化SDK。采取这种使用方式，您需要格外关注StsToken的过期时间，在StsToken即将过期时，需要您主动更新新的StsToken到SDK中。

初始化代码为：

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
OSSCredentialProvider credentialProvider = new OSSStsTokenCredentialProvider("<StsToken.
AccessKeyId>", "<StsToken.SecretKeyId>", "<StsToken.SecurityToken>");
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

在您判断到Token即将过期时，您可以重新构造新的OSSClient，也可以通过如下方式更新CredentialProvider：

```
oss.updateCredentialProvider(new OSSStsTokenCredentialProvider("<StsToken.AccessKeyId
>", "<StsToken.SecretKeyId>", "<StsToken.SecurityToken>"));
```

实现获取StsToken回调

如果您期望SDK能自动帮您管理Token的更新，那么，您需要告诉SDK如何获取Token。在SDK的应用中，您需要实现一个回调，这个回调通过您实现的方式去获取一个Federation Token（即StsToken），然后返回。SDK会利用这个Token来进行加签处理，并在需要更新时主动调用这个回调获取Token，如下所示：

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
OSSCredentialProvider credentialProvider = new OSSFederationCredentialProvider() {
    @Override
    public OSSFederationToken getFederationToken() {
        // 您需要在这里实现获取一个FederationToken，并构造成OSSFederationToken对象返回
        // 如果因为某种原因获取失败，可直接返回nil
        OSSFederationToken * token;
        // 下面是一些获取token的代码，比如从您的server获取
        ...
        return token;
    }
};
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

此外，如果您已经通过别的方式拿到token所需的各个字段，也可以在这个回调中直接返回。如果这么做的话，您需要自己处理token的更新，更新后重新设置该OSSClient实例的OSSCredentialProvider。

使用示例：

假设您搭建的server地址为： http://localhost:8080/distribute-token.json ，并假设访问这个地址，返回的数据如下：

```
{"accessKeyId":"STS.iA645eTOXEqP3cg3VeHf",
```

```
"accessKeySecret":"rV3VQrpFQ4BsyHSAvi5NVLpPIVffDJv4LojUBZCf",
"expiration":"2015-11-03T09:52:59Z",
"federatedUser":"335450541522398178:alice-001",
"requestId":"C0E01B94-332E-4582-87F9-B857C807EE52",
"securityToken":"CAES7QIIARKAAZPlqaN9ILiQZPS+JDkS/GSZN45RLx4YS/p3OgaUC+
oJI3XSIbJ7StKpQ...."}
```

那么，您可以这么实现一个 OSSFederationCredentialProvider 实例：

```
OSSCredentialProvider credetialProvider = new OSSFederationCredentialProvider() {
    @Override
    public OSSFederationToken getFederationToken() {
        try {
            URL stsUrl = new URL("http://localhost:8080/distribute-token.json");
            HttpURLConnection conn = (HttpURLConnection) stsUrl.openConnection();
            InputStream input = conn.getInputStream();
            String jsonText = IOUtils.readStreamAsString(input, OSSConstants.DEFAULT_CH
ARSET_NAME);
            JSONObject jsonObjs = new JSONObject(jsonText);
            String ak = jsonObjs.getString("accessKeyId");
            String sk = jsonObjs.getString("accessKeySecret");
            String token = jsonObjs.getString("securityToken");
            String expiration = jsonObjs.getString("expiration");
            return new OSSFederationToken(ak, sk, token, expiration);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
};
```

自签名模式

您可以把AccessKeyId/AccessKeySecret保存在您的业务server，然后在SDK实现回调，将需要加签的合并好的签名串POST到server，您在业务server对这个串按照OSS规定的签名算法签名之后，返回给该回调函数，再由回调返回。

content是已经根据请求各个参数拼接后的字符串，所以算法为：

```
signature = "OSS " + AccessKeyId + ":" + base64(hmac-sha1(AccessKeySecret, content))
```

如下：

```
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
credentialProvider = new OSSCustomSignerCredentialProvider() {
    @Override
    public String signContent(String content) {
        // 您需要在这里依照OSS规定的签名算法，实现加签一串字符内容，并把得到的签名传拼接
上AccessKeyId后返回
        // 一般实现是，将字符内容post到您的业务服务器，然后返回签名
        // 如果因为某种原因加签失败，描述error信息后，返回nil
        // 以下是用本地算法进行的演示
        return "OSS " + AccessKeyId + ":" + base64(hmac-sha1(AccessKeySecret, content));
    }
};
```

```
OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

说明：

无论是STS鉴权模式，还是自签名模式，您实现的回调函数，都需要保证调用时返回结果。所以，如果您在其中实现了向业务server获取token、signature的网络请求，建议调用网络库的同步接口。回调都是在SDK具体请求的时候，在请求的子线程中执行，所以不会阻塞主线程。

1.2.3.6 上传文件

简单上传本地文件

调用同步接口上传：

```
// 构造上传请求
PutObjectRequest put = new PutObjectRequest("<bucketName>", "<objectKey>", "<uploadFile
Path>");
// 文件元信息的设置是可选的
// ObjectMetadata metadata = new ObjectMetadata();
// metadata.setContentType("application/octet-stream"); // 设置content-type
// metadata.setContentMD5(BinaryUtil.calculateBase64Md5(uploadFilePath)); // 校验MD5
// put.setMetadata(metadata);
try {
    PutObjectResult putResult = oss.putObject(put);
    Log.d("PutObject", "UploadSuccess");
    Log.d("ETag", putResult.getETag());
    Log.d("RequestId", putResult.getRequestId());
} catch (ClientException e) {
    // 本地异常如网络异常等
    e.printStackTrace();
} catch (ServiceException e) {
    // 服务异常
    Log.e("RequestId", e.getRequestId());
    Log.e("ErrorCode", e.getErrorCode());
    Log.e("HostId", e.getHostId());
    Log.e("RawMessage", e.getRawMessage());
}
```

说明：

在Android中，不能在UI线程调用同步接口，只能在子线程调用，否则将出现异常。如果希望直接在UI线程中上传，请使用异步接口。

调用异步接口上传：

```
// 构造上传请求
PutObjectRequest put = new PutObjectRequest("<bucketName>", "<objectKey>", "<uploadFile
Path>");
// 异步上传时可以设置进度回调
put.setProgressCallback(new OSSProgressCallback<PutObjectRequest>() {
    @Override
    public void onProgress(PutObjectRequest request, long currentSize, long totalSize) {
        Log.d("PutObject", "currentSize: " + currentSize + " totalSize: " + totalSize);
```

```
    }
});
OSSAsyncTask task = oss.asyncPutObject(put, new OSSCompletedCallback<PutObjectR
equest, PutObjectResult>() {
    @Override
    public void onSuccess(PutObjectRequest request, PutObjectResult result) {
        Log.d("PutObject", "UploadSuccess");
        Log.d("ETag", result.getETag());
        Log.d("RequestId", result.getRequestId());
    }
    @Override
    public void onFailure(PutObjectRequest request, ClientException clientExcepion, ServiceExc
eption serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
// task.cancel(); // 可以取消任务
// task.waitUntilFinished(); // 可以等待任务完成
```

简单上传二进制byte[]数组

```
byte[] uploadData = new byte[100 * 1024];
new Random().nextBytes(uploadData);
// 构造上传请求
PutObjectRequest put = new PutObjectRequest(testBucket, testObject, uploadData);
try {
    PutObjectResult putResult = oss.putObject(put);
    Log.d("PutObject", "UploadSuccess");
    Log.d("ETag", putResult.getETag());
    Log.d("RequestId", putResult.getRequestId());
} catch (ClientException e) {
    // 本地异常如网络异常等
    e.printStackTrace();
} catch (ServiceException e) {
    // 服务异常
    Log.e("RequestId", e.getRequestId());
    Log.e("ErrorCode", e.getErrorCode());
    Log.e("HostId", e.getHostId());
    Log.e("RawMessage", e.getRawMessage());
}
```

上传到文件目录

OSS服务是没有文件夹这个概念的，所有元素都是以文件来存储，但给用户提供了创建模拟文件夹的方式。创建模拟文件夹本质上来说是创建了一个名字以/结尾的文件，对于这个文件照样可以上传下载，只是控制台会对以/结尾的文件以文件夹的方式展示。

如，在上传文件是，如果把ObjectKey写为 *"folder/subfolder/file"*，即是模拟了把文件上传到 *folder/subfolder/*下的 *file*文件。

说明：

路径默认是根目录，不需要以/开头。

上传Content-Type设置

Content-Type，在Web服务中定义文件的类型，决定以什么形式、什么编码读取这个文件。某些情况下，对于上传的文件需要设定Content-Type，否则文件不能以自己需求的形式和编码读取。

使用SDK上传文件时，如果不指定Content-Type，SDK会帮您根据后缀自动添加Content-Type。

```
// 构造上传请求
PutObjectRequest put = new PutObjectRequest("<bucketName>", "<objectKey>", "<uploadFile
Path>");
ObjectMetadata metadata = new ObjectMetadata();
// 指定Content-Type
metadata.setContentType("application/octet-stream");
// user自定义metadata
metadata.addUserMetadata("x-oss-meta-name1", "value1");
put.setMetadata(metadata);
OSSAsyncTask task = oss.asyncPutObject(put, new OSSCompletedCallback<PutObjectR
equest, PutObjectResult>() {
    ...
});
```

MD5校验设置

如果要校验上传到OSS的文件和本地文件是否一致，可以在上传文件时携带文件的Content-MD5值，OSS服务器会帮助用户进行MD5校验，只有在OSS服务器接收到的文件MD5值和Content-MD5一致时才可以上传成功，从而保证上传数据的一致性。

```
// 构造上传请求
PutObjectRequest put = new PutObjectRequest("<bucketName>", "<objectKey>", "<uploadFile
Path>");
ObjectMetadata metadata = new ObjectMetadata();
metadata.setContentType("application/octet-stream");
try {
    // 设置Md5以便校验
    metadata.setContentMD5(BinaryUtil.calculateBase64Md5("<uploadFilePath>")); // 如果是从
文件上传
    // metadata.setContentMD5(BinaryUtil.calculateBase64Md5(byte[])); // 如果是上传二进制数据
} catch (IOException e) {
    e.printStackTrace();
}
put.setMetadata(metadata);
OSSAsyncTask task = oss.asyncPutObject(put, new OSSCompletedCallback<PutObjectR
equest, PutObjectResult>() {
    ...
```

```
});
```

追加上传

Append Object以追加写的方式上传文件。通过Append Object操作创建的Object类型为Appendable Object，而通过Put Object上传的Object是Normal Object。

```
AppendObjectRequest append = new AppendObjectRequest(testBucket, testObject, uploadFile
Path);
ObjectMetadata metadata = new ObjectMetadata();
metadata.setContentType("application/octet-stream");
append.setMetadata(metadata);
// 设置追加位置
append.setPosition(0);
append.setProgressCallback(new OSSProgressCallback<AppendObjectRequest>() {
    @Override
    public void onProgress(AppendObjectRequest request, long currentSize, long totalSize) {
        Log.d("AppendObject", "currentSize: " + currentSize + " totalSize: " + totalSize);
    }
});
OSSAsyncTask task = oss.asyncAppendObject(append, new OSSCompletedCallback<
AppendObjectRequest, AppendObjectResult>() {
    @Override
    public void onSuccess(AppendObjectRequest request, AppendObjectResult result) {
        Log.d("AppendObject", "AppendSuccess");
        Log.d("NextPosition", "" + result.getNextPosition());
    }
    @Override
    public void onFailure(AppendObjectRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 异常处理
    }
});
```

用户使用Append方式上传，关键得对Position这个参数进行正确的设置。当用户创建一个Appendable Object时，追加位置设为0。当对Appendable Object进行内容追加时，追加位置设为Object当前长度。有两种方式获取该Object长度：一种是通过上传追加后的返回内容获取。另一种是通过head object获取文件长度。

上传后回调通知

客户端在上传Object时可以指定OSS服务端在处理完上传请求后，通知您的业务服务器，在该服务器确认接收了该回调后将回调的结果返回给客户端。因为加入了回调请求和响应的过程，相比简单上传，使用回调通知机制一般会导致客户端花费更多的等待时间。

代码示例：

```
PutObjectRequest put = new PutObjectRequest(testBucket, testObject, uploadFilePath);
put.setCallbackParam(new HashMap<String, String>() {
    {
        put("callbackUrl", "110.75.82.106/callback");
        put("callbackHost", "oss-cn-hangzhou.aliyuncs.com");
        put("callbackBodyType", "application/json");
```

```
        put("callbackBody", "{\"mimeType\":${mimeType},\"size\":${size}}");
    }
});
OSSAsyncTask task = oss.asyncPutObject(put, new OSSCompletedCallback<PutObjectR
equest, PutObjectResult>() {
    @Override
    public void onSuccess(PutObjectRequest request, PutObjectResult result) {
        Log.d("PutObject", "UploadSuccess");
        // 只有设置了servercallback，这个值才有数据
        String serverCallbackReturnJson = result.getServerCallbackReturnBody();
        Log.d("servercallback", serverCallbackReturnJson);
    }
    @Override
    public void onFailure(PutObjectRequest request, ClientException clientExcepion, ServiceExc
eption serviceException) {
        // 异常处理
    }
});
```

如果需要支持自定义参数，参考如下设置：

```
put.setCallbackParam(new HashMap<String, String>() {
    {
        put("callbackUrl", "http://182.92.192.125/leibin/notify.php");
        put("callbackHost", "oss-cn-hangzhou.aliyuncs.com");
        put("callbackBodyType", "application/json");
        put("callbackBody", "{\"object\":${object},\"size\":${size},\"my_var1\":${x:var1},\"my_var2\":
${x:var2}}");
    }
});
put.setCallbackVars(new HashMap<String, String>() {
    {
        put("x:var1", "value1");
        put("x:var2", "value2");
    }
});
```

断点续传

在无线网络下，上传比较大的文件持续时间长，可能会遇到因为网络条件差、用户切换网络等原因导致上传中途失败，整个文件需要重新上传。为此，SDK提供了断点上传功能。

断点上传暂时只支持上传本地文件。在上传前，可以指定断点记录的保存文件夹。若不进行此项设置，断点上传只在本次上传生效，某个分片因为网络原因等上传失败时会进行重试，避免整个大文件重新上传，节省重试时间和耗用流量。如果设置了断点记录的保存文件夹，那么，如果任务失败，在下次重新启动任务，上传同一文件到同一Bucket、Object时，将从断点记录处继续上传。

断点续传失败时，如果同一任务一直得不到续传，可能会在OSS上积累无用碎片。对这种情况，可以为Bucket设置lifeCycle规则，定时清理碎片。

断点续传的实现依赖 `InitMultipartUpload/UploadPart/ListParts/CompleteMultipartUpload/AbortMultipartUpload`，如果采用STS鉴权模式，请注意加上这些API所需的权限。

断点续传也支持上传后回调通知，用法和上述普通上传回调通知一致。

说明：

对于移动端来说，如果不是比较大的文件，不建议使用这种方式上传，因为断点续传是通过分片上传实现的，上传单个文件需要进行多次网络请求，效率不高。

不在本地持久保存断点记录的调用方式：

```
// 创建断点上传请求
ResumableUploadRequest request = new ResumableUploadRequest("<bucketName>", "<
objectKey>", "<uploadFilePath>");
// 设置上传过程回调
request.setProgressCallback(new OSSProgressCallback<ResumableUploadRequest>() {
    @Override
    public void onProgress(ResumableUploadRequest request, long currentSize, long totalSize)
 {
        Log.d("resumableUpload", "currentSize: " + currentSize + " totalSize: " + totalSize);
    }
});
// 异步调用断点上传
OSSAsyncTask resumableTask = oss.asyncResumableUpload(request, new OSSComplet
edCallback<ResumableUploadRequest, ResumableUploadResult>() {
    @Override
    public void onSuccess(ResumableUploadRequest request, ResumableUploadResult result) {
        Log.d("resumableUpload", "success!");
    }
    @Override
    public void onFailure(ResumableUploadRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 异常处理
    }
});
// resumableTask.waitUntilFinished(); // 可以等待直到任务完成
```

在本地持久保存断点记录的调用方式：

```
String recordDirectory = Environment.getExternalStorageDirectory().getAbsolutePath() + "/
oss_record/";
File recordDir = new File(recordDirectory);
// 要保证目录存在，如果不存在则主动创建
if (recordDir.exists()) {
    recordDir.mkdirs();
}
// 创建断点上传请求，参数中给出断点记录文件的保存位置，需是一个文件夹的绝对路径
ResumableUploadRequest request = new ResumableUploadRequest("<bucketName>", "<
objectKey>", "<uploadFilePath>", recordDirectory);
// 设置上传过程回调
request.setProgressCallback(new OSSProgressCallback<ResumableUploadRequest>() {
    @Override
    public void onProgress(ResumableUploadRequest request, long currentSize, long totalSize)
 {
        Log.d("resumableUpload", "currentSize: " + currentSize + " totalSize: " + totalSize);
    }
});
OSSAsyncTask resumableTask = oss.asyncResumableUpload(request, new OSSComplet
edCallback<ResumableUploadRequest, ResumableUploadResult>() {
```

```
    @Override
    public void onSuccess(ResumableUploadRequest request, ResumableUploadResult result) {
        Log.d("resumableUpload", "success!");
    }
    @Override
    public void onFailure(ResumableUploadRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 异常处理
    }
});
// resumableTask.waitUntilFinished();
```

1.2.3.7 下载文件

简单下载

下载指定文件，下载后将获得文件的输入流，此操作要求用户对该Object有读权限。

同步调用：

```
// 构造下载文件请求
GetObjectRequest get = new GetObjectRequest("<bucketName>", "<objectKey>");
try {
    // 同步执行下载请求，返回结果
    GetObjectResult getResult = oss.getObject(get);
    Log.d("Content-Length", "" + getResult.getContentLength());
    // 获取文件输入流
    InputStream inputStream = getResult.getObjectContent();
    byte[] buffer = new byte[2048];
    int len;
    while ((len = inputStream.read(buffer)) != -1) {
        // 处理下载的数据，比如图片展示或者写入文件等
    }
    // 下载后可以查看文件元信息
    ObjectMetadata metadata = getResult.getMetadata();
    Log.d("ContentType", metadata.getContentType());
} catch (ClientException e) {
    // 本地异常如网络异常等
    e.printStackTrace();
} catch (ServiceException e) {
    // 服务异常
    Log.e("RequestId", e.getRequestId());
    Log.e("ErrorCode", e.getErrorCode());
    Log.e("HostId", e.getHostId());
    Log.e("RawMessage", e.getRawMessage());
} catch (IOException e) {
    e.printStackTrace();
}
```

异步调用：

```
GetObjectRequest get = new GetObjectRequest("<bucketName>", "<objectKey>");
OSSAsyncTask task = oss.asyncGetObject(get, new OSSCompletedCallback<GetObjectR
equest, GetObjectResult>() {
    @Override
    public void onSuccess(GetObjectRequest request, GetObjectResult result) {
        // 请求成功
        InputStream inputStream = result.getObjectContent();
```

```
        byte[] buffer = new byte[2048];
        int len;
        try {
            while ((len = inputStream.read(buffer)) != -1) {
                // 处理下载的数据
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void onFailure(GetObjectRequest request, ClientException clientExcepion, ServiceExc
eption serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
```

图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处理服务。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义
- 通过级联处理调用多个图片处理功能

SDK中使用时，只需要在下载图片时，调用 request.setxOssProcess() 方法设置处理参数。示例：

```
// 构造图片下载请求
GetObjectRequest get = new GetObjectRequest("<bucketName>", "example.jpg");
// 图片处理
```

```
request.setxOssProcess("image/resize,m_fixed,w_100,h_100");
OSSAsyncTask task = oss.asyncGetObject(get, new OSSCompletedCallback<GetObjectR
equest, GetObjectResult>() {
    @Override
    public void onSuccess(GetObjectRequest request, GetObjectResult result) {
        // 请求成功
        InputStream inputStream = result.getObjectContent();
        byte[] buffer = new byte[2048];
        int len;
        try {
            while ((len = inputStream.read(buffer)) != -1) {
                // 处理下载的数据
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void onFailure(GetObjectRequest request, ClientException clientExcepion, ServiceExc
eption serviceException) {
        // 处理异常
    }
});
```

需要对图片进行其他处理，只要替换 request.setxOssProcess() 的参数就可以了。需要帮助，请参考：

- 可视化图片处理工具 *ImageStyleViever* ，可以直观的看到OSS图片处理的结果
- OSS图片处理的功能、使用演示 页面

指定范围下载

您可以在下载文件时指定一段范围，对于较大的Object适于使用此功能；如果在请求头中使用Range参数，则返回消息中会包含整个文件的长度和此次返回的范围。如：

```
GetObjectRequest get = new GetObjectRequest("<bucketName>", "<objectKey>");
// 设置范围
get.setRange(new Range(0, 99)); // 下载0到99字节共100个字节，文件范围从0开始计算
// get.setRange(new Range(100, Range.INFINITE)); // 下载从100个字节到结尾
OSSAsyncTask task = oss.asyncGetObject(get, new OSSCompletedCallback<GetObjectR
equest, GetObjectResult>() {
    @Override
    public void onSuccess(GetObjectRequest request, GetObjectResult result) {
        // 请求成功
        InputStream inputStream = result.getObjectContent();
        byte[] buffer = new byte[2048];
        int len;
        try {
            while ((len = inputStream.read(buffer)) != -1) {
                // 处理下载的数据
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @Override
```

```
    public void onFailure(GetObjectRequest request, ClientException clientExcepion, ServiceExc
eption serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
```

只获取文件元信息

通过headObject方法可以只获文件元信息而不获取文件的实体。代码如下：

```
// 创建同步获取文件元信息请求
HeadObjectRequest head = new HeadObjectRequest("<bucketName>", "<objectKey>");
OSSAsyncTask task = oss.asyncHeadObject(head, new OSSCompletedCallback<HeadObject
Request, HeadObjectResult>() {
    @Override
    public void onSuccess(HeadObjectRequest request, HeadObjectResult result) {
        Log.d("headObject", "object Size: " + result.getMetadata().getContentLength());
        Log.d("headObject", "object Content Type: " + result.getMetadata().getContentType());
    }
    @Override
    public void onFailure(HeadObjectRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
```

```
// task.waitUntilFinished();
```

1.2.3.8 授权访问

SDK支持签名出特定有效时长或者公开的URL，用于转给第三方实现授权访问。

签名私有资源的指定有效时长的访问URL

如果Bucket或Object不是公共可读的，那么需要调用以下接口，获得签名后的URL：

```
String url = oss.presignConstrainedObjectURL("<bucketName>", "<objectKey>", 30 * 60);
```

签名公开的访问URL

如果Bucket或Object是公共可读的，那么调用一下接口，获得可公开访问Object的URL：

```
String url = oss.presignPublicObjectURL("<bucketName>", "<objectKey>");
```

1.2.3.9 分片上传

本节演示通过分片上传文件的整个流程。

初始化分片上传

```
String uploadId;

InitiateMultipartUploadRequest init = new InitiateMultipartUploadRequest("<bucketName>", "<
objectKey>");
InitiateMultipartUploadResult initResult = oss.initMultipartUpload(init);

uploadId = initResult.getUploadId();
```

- 我们用InitiateMultipartUploadRequest来指定上传文件的名字和所属存储空间（Bucket）。
- 在InitiateMultipartUploadRequest中，您也可以设置ObjectMeta，但是不必指定其中的ContentLength 。
- initiateMultipartUpload 的返回结果中含有UploadId ，它是区分分片上传事件的唯一标识，在后面的操作中，我们将用到它。

上传分片

```
long partSize = 128 * 1024; // 设置分片大小

int currentIndex = 1; // 上传分片编号，从1开始

File uploadFile = new File("<uploadFilePath>"); // 需要分片上传的文件

InputStream input = new FileInputStream(uploadFile);
long fileLength = uploadFile.length();

long uploadedLength = 0;
List<PartETag> partETags = new ArrayList<PartETag>(); // 保存分片上传的结果
```

```
while (uploadedLength < fileLength) {

    int partLength = (int)Math.min(partSize, fileLength - uploadedLength);
    byte[] partData = IOUtils.readStreamAsBytesArray(input, partLength); // 按照分片大小读取文件的一段内容

    UploadPartRequest uploadPart = new UploadPartRequest("<bucketName>", "<objectKey>", uploadId, currentIndex);
    uploadPart.setPartContent(partData); // 设置分片内容
    UploadPartResult uploadPartResult = oss.uploadPart(uploadPart);
    partETags.add(new PartETag(currentIndex, uploadPartResult.getETag())); // 保存分片上传成功后的结果

    uploadedLength += partLength;
    currentIndex++;
}
```

上面程序的核心是调用UploadPart方法来上传每一个分片，但是要注意以下几点：

- UploadPart 方法要求除最后一个Part以外，其他的Part大小都要大于100KB。但是Upload Part接口并不会立即校验上传 Part的大小（因为不知道是否为最后一块）；只有当Complete Multipart Upload的时候才会校验。
- OSS会将服务器端收到Part数据的MD5值放在ETag头内返回给用户。
- Part号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
- 每次上传part时都要把流定位到此次上传片开头所对应的位置。
- 每次上传part之后，OSS的返回结果会包含一个分片的ETag，您需要将它和块编号组合成PartEtag，保存在list中，后续完成分片上传需要用到。

完成分片上传

```
CompleteMultipartUploadRequest complete = new CompleteMultipartUploadRequest("<bucketName>", "<objectKey>", uploadId, partETags);
CompleteMultipartUploadResult completeResult = oss.completeMultipartUpload(complete);

Log.d("multipartUpload", "multipart upload success! Location: " + completeResult.getLocation());
```

上面代码中的 partETags 就是进行分片上传中保存的partETag的列表，OSS收到用户提交的Part列表后，会逐一验证每个数据Part的有效性。 当所有的数据Part验证通过后，OSS会将这些part组合成一个完整的文件。

完成分片上传（设置ServerCallback）

```
CompleteMultipartUploadRequest complete = new CompleteMultipartUploadRequest("<bucketName>", "<objectKey>", uploadId, partETags);
CompleteMultipartUploadResult completeResult = oss.completeMultipartUpload(complete);
complete.setCallbackParam(new HashMap<String, String>() {
        {
            put("callbackUrl", "<server address>");
            put("callbackBody", "<test>");
```

```
            }
        });
Log.d("multipartUploadWithServerCallback", completeResult.getServerCallbackReturnBody());
```

完成分片上传请求可以设置Server Callback参数，请求完成后会向指定的Server Adress发送回调请求。

删除分片上传事件

我们可以用 abortMultipartUpload 方法取消分片上传。

```
AbortMultipartUploadRequest abort = new AbortMultipartUploadRequest("<bucketName>", "<
objectKey>", uploadId);
oss.abortMultipartUpload(abort); // 若无异常抛出说明删除成功
```

罗列分片

我们可以用 listParts 方法获取某个上传事件所有已上传的分片。

```
ListPartsRequest listParts = new ListPartsRequest("<bucketName>", "<objectKey>", uploadId);

ListPartsResult result = oss.listParts(listParts);

for (int i = 0; i < result.getParts().size(); i++) {
    Log.d("listParts", "partNum: " + result.getParts().get(i).getPartNumber());
    Log.d("listParts", "partEtag: " + result.getParts().get(i).getETag());
    Log.d("listParts", "lastModified: " + result.getParts().get(i).getLastModified());
    Log.d("listParts", "partSize: " + result.getParts().get(i).getSize());
}
```

- 默认情况下，如果存储空间中的分片上传事件的数量大于1000，则只会返回1000个Multipart Upload信息，且返回结果中 IsTruncated 为false，并返回 NextPartNumberMarker作为下此读取的起点。
- 若想增大返回分片上传事件数目，可以修改 MaxParts 参数，或者使用 PartNumberMarker 参数分次读取。

1.2.3.10 管理文件

罗列Bucket所有文件

```
 ListObjectsRequest listObjects = new ListObjectsRequest("<bucketName>");

// 设定前缀
listObjects.setPrefix("file");

// 设置成功、失败回调，发送异步罗列请求
OSSAsyncTask task = oss.asyncListObjects(listObjects, new OSSCompletedCallback<
ListObjectsRequest, ListObjectsResult>() {
    @Override
    public void onSuccess(ListObjectsRequest request, ListObjectsResult result) {
        Log.d("AyncListObjects", "Success!");
```

```
        for (int i = 0; i < result.getObjectSummaries().size(); i++) {
            Log.d("AyncListObjects", "object: " + result.getObjectSummaries().get(i).getKey() + " "
                    + result.getObjectSummaries().get(i).getETag() + " "
                    + result.getObjectSummaries().get(i).getLastModified());
        }
    }

    @Override
    public void onFailure(ListObjectsRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
task.waitUntilFinished();
```

上述代码列出了bucket中以”file”为前缀的所有文件。具体可以设置的参数名称和作用如下：

名称	作用
delimiter	用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素： CommonPrefixes。
marker	设定结果从marker之后按字母排序的第一个开始返回。
maxkeys	限定此次返回object的最大数，如果不设定，默认为100，maxkeys取值不能大于1000。
prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix。

检查文件是否存在

SDK提供了方便的同步接口检测某个指定Object是否存在OSS上：

```
try {
    if (oss.doesObjectExist("<bucketName>", "<objectKey>")) {
        Log.d("doesObjectExist", "object exist.");
    } else {
        Log.d("doesObjectExist", "object does not exist.");
    }
} catch (ClientException e) {
    // 本地异常如网络异常等
    e.printStackTrace();
} catch (ServiceException e) {
    // 服务异常
```

```
    Log.e("ErrorCode", e.getErrorCode());
    Log.e("RequestId", e.getRequestId());
    Log.e("HostId", e.getHostId());
    Log.e("RawMessage", e.getRawMessage());
}
```

复制文件

```
// 创建copy请求
CopyObjectRequest copyObjectRequest = new CopyObjectRequest("<srcBucketName>", "<
srcObjectKey>",
        "<destBucketName>", "<destObjectKey>");

// 可选设置copy文件元信息
// ObjectMetadata objectMetadata = new ObjectMetadata();
// objectMetadata.setContentType("application/octet-stream");
// copyObjectRequest.setNewObjectMetadata(objectMetadata);

// 异步copy
OSSAsyncTask copyTask = oss.asyncCopyObject(copyObjectRequest, new OSSComplet
edCallback<CopyObjectRequest, CopyObjectResult>() {
    @Override
    public void onSuccess(CopyObjectRequest request, CopyObjectResult result) {
            Log.d("copyObject", "copy success!");
    }

    @Override
    public void onFailure(CopyObjectRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
```

上述代码实现了CopyObject

说明：

- 源Object和目标Object必须属于同一个数据中心。
- 如果拷贝操作的源Object地址和目标Object地址相同，可以修改已有Object的meta信息。
- 拷贝文件大小不能超过1G，超过1G需使用Multipart Upload操作。

删除文件

```
// 创建删除请求
```

```
DeleteObjectRequest delete = new DeleteObjectRequest("<bucketName>", "<objectKey>");
// 异步删除
OSSAsyncTask deleteTask = oss.asyncDeleteObject(delete, new OSSCompletedCallback<
DeleteObjectRequest, DeleteObjectResult>() {
    @Override
    public void onSuccess(DeleteObjectRequest request, DeleteObjectResult result) {
        Log.d("asyncCopyAndDelObject", "success!");
    }

    @Override
    public void onFailure(DeleteObjectRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }

});
```

上述代码实现了删除Object。

说明：

DeleteObject要求对所在的Bucket有写权限。

只获取文件的元信息

下面的代码用于获取文件的元信息：

```
// 创建同步获取文件元信息请求
HeadObjectRequest head = new HeadObjectRequest("<bucketName>", "<objectKey>");

OSSAsyncTask task = oss.asyncHeadObject(head, new OSSCompletedCallback<HeadObject
Request, HeadObjectResult>() {
    @Override
    public void onSuccess(HeadObjectRequest request, HeadObjectResult result) {
        Log.d("headObject", "object Size: " + result.getMetadata().getContentLength()); // 获取文件
长度
        Log.d("headObject", "object Content Type: " + result.getMetadata().getContentType()); // 获
取文件类型
    }

    @Override
    public void onFailure(HeadObjectRequest request, ClientException clientExcepion,
ServiceException serviceException) {
        // 请求异常
        if (clientExcepion != null) {
            // 本地异常如网络异常等
            clientExcepion.printStackTrace();
```

```
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});

// task.waitUntilFinished();
```

1.2.3.11 管理Bucket

Bucket 是 OSS 上的命名空间，也是计费、权限控制、日志记录等高级功能的管理实体；Bucket 名称在整个 OSS 服务中具有全局唯一性，且不能修改；存储在 OSS 上的每个 Object 必须都包含在某个 Bucket 中。

新建Bucket

以下代码可以新建一个Bucket：

```
CreateBucketRequest createBucketRequest = new CreateBucketRequest("<bucketName>");
createBucketRequest.setBucketACL(CannedAccessControlList.PublicRead); // 指定Bucket的ACL权限
createBucketRequest.setLocationConstraint("oss-cn-hangzhou"); // 指定Bucket所在的数据中心
OSSAsyncTask createTask = oss.asyncCreateBucket(createBucketRequest, new OSSCompletedCallback<CreateBucketRequest, CreateBucketResult>() {
    @Override
    public void onSuccess(CreateBucketRequest request, CreateBucketResult result) {
        Log.d("locationConstraint", request.getLocationConstraint());
        }

    @Override
    public void onFailure(CreateBucketRequest request, ClientException clientException, ServiceException serviceException) {
        // 请求异常
        if (clientException != null) {
            // 本地异常如网络异常等
            clientException.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
```

上述代码在创建bucket时，指定了Bucket的ACL和所在的数据中心。

- 每个用户的Bucket数量不能超过10个。

- 每个Bucket的名字全局唯一，也就是说创建的Bucket不能和其他用户已经在使用的Bucket同名，否则会创建失败。
- 创建的时候可以选择Bucket ACL权限，如果不设置ACL，默认是private。
- 创建成功结果返回Bucket所在数据中心。

获取Bucket ACL权限

以下代码可以获取Bucket ACL：

```
GetBucketACLRequest getBucketACLRequest = new GetBucketACLRequest("<bucketName
>");
OSSAsyncTask getBucketAclTask = oss.asyncGetBucketACL(getBucketACLRequest, new
OSSCompletedCallback<GetBucketACLRequest, GetBucketACLResult>() {
    @Override
    public void onSuccess(GetBucketACLRequest request, GetBucketACLResult result) {
        Log.d("BucketAcl", result.getBucketACL());
        Log.d("Owner", result.getBucketOwner());
        Log.d("ID", result.getBucketOwnerID());
    }

    @Override
    public void onFailure(GetBucketACLRequest request, ClientException clientException,
ServiceException serviceException) {
        // 请求异常
        if (clientException != null) {
            // 本地异常如网络异常等
            clientException.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
    }
});
```

上述代码在获取Bucket的ACL权限。

- 目前Bucket有三种访问权限：public-read-write，public-read和private。
- 只有Bucket的拥有者才能使用Get Bucket ACL这个接口。
- 获取的结果中返回Bucket拥有者ID、拥有者名称（和ID保持一致）和权限。

删除Bucket

以下代码删除了一个Bucket：

```
DeleteBucketRequest deleteBucketRequest = new DeleteBucketRequest("<bucketName>");
OSSAsyncTask deleteBucketTask = oss.asyncDeleteBucket(deleteBucketRequest, new
OSSCompletedCallback<DeleteBucketRequest, DeleteBucketResult>() {
    @Override
    public void onSuccess(DeleteBucketRequest request, DeleteBucketResult result) {
```

```
        Log.d("DeleteBucket", "Success!");
    }

    @Override
    public void onFailure(DeleteBucketRequest request, ClientException clientException,
ServiceException serviceException) {
            // 请求异常
        if (clientException != null) {
            // 本地异常如网络异常等
            clientException.printStackTrace();
        }
        if (serviceException != null) {
            // 服务异常
            Log.e("ErrorCode", serviceException.getErrorCode());
            Log.e("RequestId", serviceException.getRequestId());
            Log.e("HostId", serviceException.getHostId());
            Log.e("RawMessage", serviceException.getRawMessage());
        }
});
```

说明：

- 为了防止误删除的发生，OSS不允许用户删除一个非空的Bucket。
- 只有Bucket的拥有者才能删除这个Bucket。

1.2.3.12 异常响应

OSS Android SDK 中有两种异常 ClientException 以及 ServiceException ，他们都是受检异常。

ClientException

ClientException指SDK内部出现的异常，比如参数错误，网络无法到达，主动取消等等。

ServiceException

OSSException指服务器端错误，它来自于对服务器错误信息的解析。OSSException一般有以下几个成员：

- Code：OSS返回给用户的错误码。
- Message：OSS给出的详细错误信息。
- RequestId：用于唯一标识该次请求的UUID；当您无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助。
- HostId：用于标识访问的OSS集群
- rawMessage：HTTP响应的原始Body文本

下面是OSS中常见的异常：

错误码	描述
AccessDenied	拒绝访问
BucketAlreadyExists	Bucket已经存在
BucketNotEmpty	Bucket不为空
EntityTooLarge	实体过大
EntityTooSmall	实体过小
FileGroupTooLarge	文件组过大
FilePartNotExist	文件Part不存在
FilePartStale	文件Part过时
InvalidArgument	参数格式错误
InvalidAccessKeyId	AccessKeyId不存在
InvalidBucketName	无效的Bucket名字
InvalidDigest	无效的摘要
InvalidObjectName	无效的Object名字
InvalidPart	无效的Part
InvalidPartOrder	无效的part顺序
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket
InternalError	OSS内部发生错误
MalformedXML	XML格式非法
MethodNotAllowed	不支持的方法
MissingArgument	缺少参数
MissingContentLength	缺少内容长度
NoSuchBucket	Bucket不存在
NoSuchKey	文件不存在
NoSuchUpload	Multipart Upload ID不存在
NotImplemented	无法处理的方法
PreconditionFailed	预处理错误
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟
RequestTimeout	请求超时

错误码	描述
SignatureDoesNotMatch	签名错误
TooManyBuckets	用户的Bucket数目超过限制

1.2.4 iOS-SDK

1.2.4.1 前言

简介

本文档主要介绍OSS iOS SDK的安装和使用。本文档假设您已经开通了阿里云OSS 服务，并创建了AccessKeyId 和AccessKeySecret。文中的ID 指的是AccessKeyId，KEY 指的是AccessKeySecret。如果您还没有开通或者还不了解OSS，请登录*OSS产品主页*获取更多的帮助。

环境要求

- iOS系统版本：iOS 7.0以上
- 必须注册有Aliyun.com用户账户，并开通OSS服务。

SDK下载

- iOS SDK开发包(2016-12-15) 版本号 2.6.0：*aliyun_OSS_iOS_SDK_20161215.zip*
- github地址：*https://github.com/aliyun/aliyun-oss-ios-sdk*
- pod依赖：`pod 'AliyunOSSiOS', '~> 2.6.0'`
- demo地址： *https://github.com/alibaba/alicloud-ios-demo*

1.2.4.2 安装

直接引入Framework

需要引入OSS iOS SDK framework。

在Xcode中，直接把framework拖入您对应的Target下即可，在弹出框勾选**Copy items if needed**。

Pod依赖

如果工程是通过pod管理依赖，那么在Podfile中加入以下依赖即可，不需要再导入framework：

```
pod 'AliyunOSSiOS'
```

Cocoapods是一个非常优秀的依赖管理工具，推荐参考文档：*CocoaPods安装和使用教程*。

直接引入Framework和Pod依赖，两种方式选其一即可。

工程中引入头文件

```
#import <AliyunOSSiOS/OSSService.h>
```

说明：

引入Framework后，需要在工程 Build Settings 的 Other Linker Flags 中加入 -ObjC。如果工程此前已经设置过 -force_load 选项，那么，需要加入 -force_load <framework path>/AliyunOSSiOS。

兼容IPv6-Only网络

OSS移动端SDK为了解决无线网络下域名解析容易遭到劫持的问题，已经引入了HTTPDNS进行域名解析，直接使用IP请求OSS服务端。在IPv6-Only的网络下，可能会遇到兼容性问题。而APP官方近期发布了关于IPv6-only网络环境兼容的APP审核要求，为此，SDK从2.5.0版本开始已经做了兼容性处理。在新版本中，除了-ObjC的设置，还需要引入两个系统库：

```
libresolv.tbd
SystemConfiguration.framework
```

关于苹果ATS政策

WWDC 2016开发者大会上，苹果宣布从2017年1月1日起，苹果App Store中的所有App都必须启用 App Transport Security（ATS) 安全功能。也就是说，所有的新提交 app 默认是不允许使用 NSAllowsArbitraryLoads 来绕过 ATS 限制的。我们最好保证 app 的所有网络请求都是 HTTPS 加密的，否则可能会在应用审核时遇到麻烦。

本SDK在 2.6.0 以上版本中对此做出支持，其中，SDK不会自行发出任何非HTTPS请求，同时，SDK支持 https:// 前缀的 Endpoint，只需要设置正确的HTTPS Endpoint，就能保证发出的网络请求都是符合要求的。

所以，用户需要注意：

- 设置 Endpoint时，需要使用 https:// 前缀的URL。
- 在实现加签、获取STSToken等回调时，需要确保自己不会发出非HTTPS 的请求。

对于OSSTask的一些说明

所有调用api的操作，都会立即获得一个OSSTask，如：

```
OSSTask * task = [client getObject:get];
```

可以为这个Task设置一个延续(continution)，以实现异步回调，如：

```
[task continueWithBlock: ^(OSSTask *task) {
    // do something
    ...
    return nil;
}];
```

也可以等待这个Task完成，以实现同步等待，如：

```
[task waitUntilFinished];
...
```

1.2.4.3 初始化

OSSClient是OSS服务的iOS客户端，它为调用者提供了一系列的方法，可以用来操作，管理存储空间（bucket）和文件（object）等。在使用SDK发起对OSS的请求前，您需要初始化一个OSSClient实例，并对它进行一些必要设置。

确定Endpoint

Endpoint是阿里云OSS服务在各个区域的地址，目前支持两种形式。

Endpoint类型	解释
OSS区域地址	使用OSS Bucket所在区域地址
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS区域地址

使用OSS Bucket所在区域地址，Endpoint查询可以有下面两种方式：

- 查询Endpoint与区域对应关系详情。
- 您可以登录阿里云OSS控制台，进入Bucket概览页，Bucket域名的后缀部分：如 bucket-1.oss-cn-hangzhou.aliyuncs.com 的 oss-cn-hangzhou.aliyuncs.com 部分为该Bucket的外网Endpoint。

Cname

您可以将自己拥有的域名通过Cname绑定到某个存储空间（bucket）上，然后通过自己域名访问存储空间内的文件。

比如您要将域名new-image.xxxxx.com绑定到深圳区域的名称为image的存储空间上：

您需要到您的域名xxxxx.com托管商那里设定一个新的域名解析，将 http://new-image.xxxxx.com 解析到 http://image.oss-cn-shenzhen.aliyuncs.com，类型为CNAME。

设置EndPoint和凭证

必须设置EndPoint和CredentialProvider：

```
NSString *endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
// 由阿里云颁发的AccessKeyId/AccessKeySecret构造一个CredentialProvider。
// 明文设置secret的方式建议只在测试时使用，更多鉴权模式请参考后面的访问控制章节。
id<OSSCredentialProvider> credential = [[OSSPlainTextAKSKPairCredentialProvider alloc]
initWithPlainTextAccessKey:@"<your accessKeyId>" secretKey:@"<your accessKeySecret>"];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

设置EndPoint为cname

如果您已经在bucket上绑定cname，将该cname直接设置到endPoint即可。如：

```
NSString *endpoint = "http://new-image.xxxxx.com";
id<OSSCredentialProvider> credential = [[OSSPlainTextAKSKPairCredentialProvider alloc]
initWithPlainTextAccessKey:@"<your accessKeyId>" secretKey:@"<your accessKeySecret>"];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

说明：

苹果要求支持ATS标准后，所有 Endpoint URL都必须为HTTPS URL，而 cname 域名暂不支持证书设置，所以暂时不能用cname设置 Endpoint。

设置网络参数

您也可以在初始化的时候设置详细的ClientConfiguration：

```
NSString *endpoint = "https://oss-cn-hangzhou.aliyuncs.com";
// 明文设置secret的方式建议只在测试时使用，更多鉴权模式请参考后面的访问控制章节
id<OSSCredentialProvider> credential = [[OSSPlainTextAKSKPairCredentialProvider alloc]
initWithPlainTextAccessKey:@"<your accessKeyId>" secretKey:@"<your accessKeySecret>"];
OSSClientConfiguration * conf = [OSSClientConfiguration new];
conf.maxRetryCount = 3; // 网络请求遇到异常失败后的重试次数
conf.timeoutIntervalForRequest = 30; // 网络请求的超时时间
conf.timeoutIntervalForResource = 24 * 60 * 60; // 允许资源传输的最长时间
```

```
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential clientConf
iguration:conf];
```

1.2.4.4 快速入门

本节演示了上传、下载文件的基本流程。更多细节用法可以参考本工程的 test资源：点击查看 或者demo示例：点击查看。

STEP-1. 初始化OSSClient

初始化主要完成Endpoint设置、鉴权方式设置、Client参数设置。其中，鉴权方式包含明文设置模式、自签名模式、STS鉴权模式。鉴权细节请参考 访问控制章节。

```
NSString *endpoint = "https://oss-cn-hangzhou.aliyuncs.com";
// 明文设置secret的方式建议只在测试时使用，更多鉴权模式请参考后面的`访问控制`章节
id<OSSCredentialProvider> credential = [[OSSPlainTextAKSKPairCredentialProvider alloc]
initWithPlainTextAccessKey:@"<your accessKeyId>"
                                                                    secretKey:@"<your
accessKeySecret>"];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

通过OSSClient发起上传、下载请求是线程安全的，您可以并发执行多个任务。

STEP-2. 上传文件

这里假设您已经在控制台上拥有自己的Bucket。SDK的所有操作，都会返回一个 OSSTask，您可以为这个task设置一个延续动作，等待其异步完成，也可以通过调用 waitUntilFinished 阻塞等待其完成。

```
OSSPutObjectRequest * put = [OSSPutObjectRequest new];
put.bucketName = @"<bucketName>";
put.objectKey = @"<objectKey>";
put.uploadingData = <NSData *>; // 直接上传NSData
put.uploadProgress = ^(int64_t bytesSent, int64_t totalByteSent, int64_t totalBytesExpectedTo
Send) {
    NSLog(@"%lld, %lld, %lld", bytesSent, totalByteSent, totalBytesExpectedToSend);
};
OSSTask * putTask = [client putObject:put];
[putTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"upload object success!");
    } else {
        NSLog(@"upload object failed, error: %@" , task.error);
    }
    return nil;
}];
// 可以等待任务完成
```

```
// [putTask waitUntilFinished];
```

STEP-3. 下载指定文件

下载一个指定 object 为 NSData：

```
OSSGetObjectRequest * request = [OSSGetObjectRequest new];
request.bucketName = @"<bucketName>";
request.objectKey = @"<objectKey>";
request.downloadProgress = ^(int64_t bytesWritten, int64_t totalBytesWritten, int64_t totalBytes
ExpectedToWrite) {
    NSLog(@"%lld, %lld, %lld", bytesWritten, totalBytesWritten, totalBytesExpectedToWrite);
};
OSSTask * getTask = [client getObject:request];
[getTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"download object success!");
        OSSGetObjectResult * getResult = task.result;
        NSLog(@"download result: %@", getResult.downloadedData);
    } else {
        NSLog(@"download object failed, error: %@" ,task.error);
    }
    return nil;
}];
// 如果需要阻塞等待任务完成
// [task waitUntilFinished];
```

1.2.4.5 访问控制

移动终端是一个不受信任的环境，如果把AccessKeyId和AccessKeySecret直接保存在终端本地用来加签请求，存在极高的风险。为此，SDK提供了建议只在测试时使用的**明文设置模式**，和另外两种依赖于您的业务Server的鉴权模式：**STS鉴权模式**和**自签名模式**。

明文设置模式

```
// 明文设置AccessKeyId/AccessKeySecret的方式建议只在测试时使用
id<OSSCredentialProvider> credential = [[OSSPlainTextAKSKPairCredentialProvider alloc]
initWithPlainTextAccessKey:@"<your accessKeyId>" secretKey:@"<your accessKeySecret>"];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

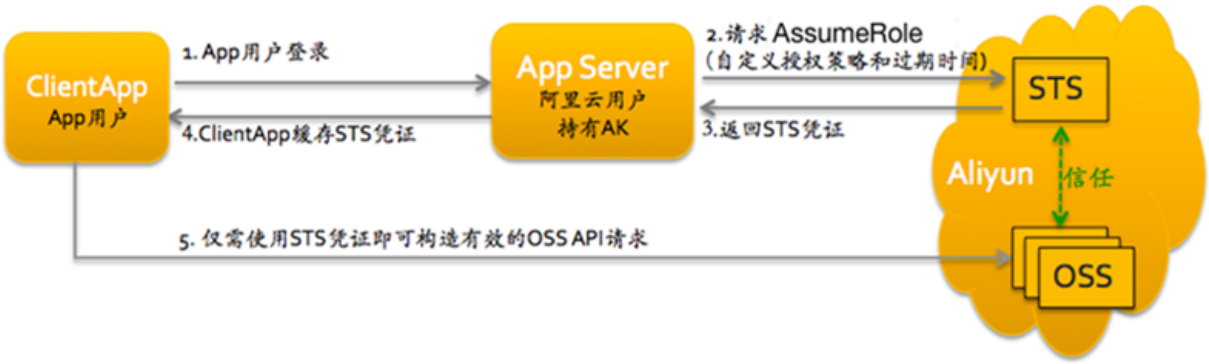
STS鉴权模式

介绍

OSS可以通过阿里云STS服务，临时进行授权访问。阿里云STS (Security Token Service) 是为云计算用户提供临时访问令牌的Web服务。通过STS，您可以为第三方应用或联邦用户（用户身份由您自己管理）颁发一个自定义时效和权限的访问凭证，App端称为FederationToken。第三方应用或联邦用户可以使用该访问凭证直接调用阿里云产品API，或者使用阿里云产品提供的SDK来访问云产品API。

- 您不需要透露您的长期密钥(AccessKey)给第三方应用，只需要生成一个访问令牌并将令牌交给第三方应用即可。这个令牌的访问权限及有效期限都可以由您自定义。
- 您不需要关心权限撤销问题，访问令牌过期后就自动失效。

以APP应用为例，交互流程如下图所示：



方案的详细描述如下：

1. App用户登录。App用户身份是客户自己管理。客户可以自定义身份管理系统，也可以使用外部Web账号或OpenID。对于每个有效的App用户来说，AppServer可以确切地定义出每个App用户的最小访问权限。
2. AppServer请求STS服务获取一个安全令牌(SecurityToken)。在调用STS之前，AppServer需要确定App用户的最小访问权限（用Policy语法描述）以及授权的过期时间。然后通过调用STS的AssumeRole(扮演角色)接口来获取安全令牌。角色管理与使用相关内容请参考《*RAM*使用指南》中的角色管理。
3. STS返回给AppServer一个有效的访问凭证，App端称为FederationToken，包括一个安全令牌(SecurityToken)、临时访问密钥(AccessKeyId, AccessKeySecret)以及过期时间。
4. AppServer将FederationToken返回给ClientApp。ClientApp可以缓存这个凭证。当凭证失效时，ClientApp需要向AppServer申请新的有效访问凭证。比如，访问凭证有效期为1小时，那么ClientApp可以每30分钟向AppServer请求更新访问凭证。
5. ClientApp使用本地缓存的FederationToken去请求Aliyun Service API。云服务会感知STS访问凭证，并会依赖STS服务来验证访问凭证，并正确响应用户请求。

直接设置StsToken

您可以在APP中，预先通过某种方式（如通过网络请求从您的业务Server上）获取一对StsToken，然后用它来初始化SDK。采取这种使用方式，您需要格外关注StsToken的过期时间，在StsToken即将过期时，需要您主动更新新的StsToken到SDK中。

初始化代码为：

```
id<OSSCredentialProvider> credential = [[OSSStsTokenCredentialProvider alloc] initWithAc
cessKeyId:@"<StsToken.AccessKeyId>" secretKeyId:@"<StsToken.SecretKeyId>" securityTo
ken:@"<StsToken.SecurityToken>"];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

在您判断到Token即将过期时，您可以重新构造新的OSSClient，也可以通过如下方式更新CredentialProvider:

```
client.credentialProvider = [[OSSStsTokenCredentialProvider alloc] initWithAccessKeyId
:@"<StsToken.AccessKeyId>" secretKeyId:@"<StsToken.SecretKeyId>" securityToken:@"<
StsToken.SecurityToken>"];
```

实现获取StsToken回调

如果您期望SDK能自动帮您管理Token的更新，那么，您需要告诉SDK如何获取Token。在SDK的应用中，您需要实现一个回调，这个回调通过您实现的方式去获取一个Federation Token（即StsToken），然后返回。SDK会利用这个Token来进行加签处理，并在需要更新时主动调用这个回调获取Token，如下所示：

```
id<OSSCredentialProvider> credential = [[OSSFederationCredentialProvider alloc] initWithFe
derationTokenGetter:^OSSFederationToken * {
    // 您需要在这里实现获取一个FederationToken，并构造成OSSFederationToken对象返回
    // 如果因为某种原因获取失败，可直接返回nil
    OSSFederationToken * token;
    // 下面是一些获取token的代码，比如从您的server获取
    ...
    return token;
}];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

此外，如果您已经通过别的方式拿到token所需的各个字段，也可以在这个回调中直接返回。如果这么做的话，您需要自己处理token的更新，更新后重新设置该OSSClient实例的OSSCredentialProvider。

使用示例：

假设您搭建的server地址为： http://localhost:8080/distribute-token.json ，并假设访问这个地址，返回的数据如下：

```
{"accessKeyId":"STS.iA645eTOXEqP3cg3VeHf",
"accessKeySecret":"rV3VQrpFQ4BsyHSAvi5NVLpPIVffDJv4LojUBZCf",
"expiration":"2015-11-03T09:52:59Z",
"federatedUser":"335450541522398178:alice-001",
"requestId":"C0E01B94-332E-4582-87F9-B857C807EE52",
```

```
"securityToken":"CAES7QIIARKAAZPlqaN9ILiQZPS+JDkS/GSZN45RLx4YS/p3OgaUC+
oJI3XSlbJ7StKpQ.... "}
```

那么，您可以这么实现一个 OSSFederationCredentialProvider 实例：

```
id<OSSCredentialProvider> credential2 = [[OSSFederationCredentialProvider alloc] initWithFe
derationTokenGetter:^OSSFederationToken * {
    // 构造请求访问您的业务server
    NSURL * url = [NSURL URLWithString:@"http://localhost:8080/distribute-token.json"];
    NSURLRequest * request = [NSURLRequest requestWithURL:url];
    OSSTaskCompletionSource * tcs = [OSSTaskCompletionSource taskCompletionSource];
    NSURLSession * session = [NSURLSession sharedSession];
    // 发送请求
    NSURLSessionTask * sessionTask = [session dataTaskWithRequest:request
                                      completionHandler:^(NSData *data, NSURLResponse *response
, NSError *error) {
                                          if (error) {
                                              [tcs setError:error];
                                              return;
                                          }
                                          [tcs setResult:data];
                                      }];
    [sessionTask resume];
    // 需要阻塞等待请求返回
    [tcs.task waitUntilFinished];
    // 解析结果
    if (tcs.task.error) {
        NSLog(@"get token error: %@", tcs.task.error);
        return nil;
    } else {
        // 返回数据是json格式，需要解析得到token的各个字段
        NSDictionary * object = [NSJSONSerialization JSONObjectWithData:tcs.task.result
                                                         options:kNilOptions
                                                           error:nil];
        OSSFederationToken * token = [OSSFederationToken new];
        token.tAccessKey = [object objectForKey:@"accessKeyId"];
        token.tSecretKey = [object objectForKey:@"accessKeySecret"];
        token.tToken = [object objectForKey:@"securityToken"];
        token.expirationTimeInGMTFormat = [object objectForKey:@"expiration"];
        NSLog(@"get token: %@", token);
        return token;
    }
}];
```

自签名模式

```
id<OSSCredentialProvider> credential = [[OSSCustomSignerCredentialProvider alloc]
initWithImplementedSigner:^NSString *(NSString *contentToSign, NSError *__autoreleasing *
error) {
    // 您需要在这里依照OSS规定的签名算法，实现加签一串字符内容，并把得到的签名传拼接上
AccessKeyId后返回
    // 一般实现是，将字符内容post到您的业务服务器，然后返回签名
    // 如果因为某种原因加签失败，描述error信息后，返回nil
    NSString *signature = [OSSUtil calBase64Sha1WithData:contentToSign withSecret:@"<your
accessKeySecret>"]; // 这里是用SDK内的工具函数进行本地加签，建议您通过业务server实现远
程加签
    if (signature != nil) {
        *error = nil;
    } else {
```

```
        *error = [NSError errorWithDomain:@"<your domain>" code:-1001 userInfo:@"<your error
 info>"];
        return nil;
    }
    return [NSString stringWithFormat:@"OSS %@:%@", @"<your accessKeyId>", signature];
}];
client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

说明：

无论是STS鉴权模式，还是自签名模式，您实现的回调函数，都需要保证调用时返回结果。所以，如果您在其中实现了向业务server获取token、signature的网络请求，建议调用网络库的同步接口，或进行异步到同步的转换。回调都是在SDK具体请求的时候，在请求的子线程中执行，所以不用担心阻塞主线程。

1.2.4.6 上传文件

简单上传

上传Object可以直接上传OSSData，或者通过NSURL上传一个文件：

```
OSSPutObjectRequest * put = [OSSPutObjectRequest new];
// 必填字段
put.bucketName = @"<bucketName>";
put.objectKey = @"<objectKey>";
put.uploadingFileURL = [NSURL fileURLWithPath:@"<filepath>"];
// put.uploadingData = <NSData *>; // 直接上传NSData
// 可选字段，可不设置
put.uploadProgress = ^(int64_t bytesSent, int64_t totalByteSent, int64_t totalBytesExpectedTo
Send) {
    // 当前上传段长度、当前已经上传总长度、一共需要上传的总长度
    NSLog(@"%lld, %lld, %lld", bytesSent, totalByteSent, totalBytesExpectedToSend);
};
// 以下可选字段的含义参考： https://docs.aliyun.com/#/pub/oss/api-reference/object&PutObject
// put.contentType = @"";
// put.contentMd5 = @"";
// put.contentEncoding = @"";
// put.contentDisposition = @"";
// put.objectMeta = [NSMutableDictionary dictionaryWithObjectsAndKeys:@"value1", @"x-oss-
meta-name1", nil]; // 可以在上传时设置元信息或者其他HTTP头部
OSSTask * putTask = [client putObject:put];
[putTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"upload object success!");
    } else {
        NSLog(@"upload object failed, error: %@" , task.error);
    }
    return nil;
}];
// [putTask waitUntilFinished];
```

```
// [put cancel];
```

上传到文件目录

OSS服务是没有文件夹这个概念的，所有元素都是以文件来存储，但给用户提供了创建模拟文件夹的方式。创建模拟文件夹本质上来说是创建了一个名字以/结尾的文件，对于这个文件照样可以上传下载，只是控制台会对以/结尾的文件以文件夹的方式展示。

如，在上传文件是，如果把ObjectKey写为 `"folder/subfolder/file"`，即是模拟了把文件上传到 *folder/subfolder/* 下的 *file* 文件。注意，路径默认是根目录，不需要以/开头。

上传时设置Content-Type和开启校验MD5

上传时可以显式指定ContentType，如果没有指定，SDK会根据文件名或者上传的ObjectKey自动判断。另外，上传Object时如果设置了Content-Md5，那么OSS会用之检查消息内容是否与发送时一致。SDK提供了方便的Base64和MD5计算方法。

```
OSSPutObjectRequest * put = [OSSPutObjectRequest new];
// 必填字段
put.bucketName = @"<bucketName>";
put.objectKey = @"<objectKey>";
put.uploadingFileURL = [NSURL fileURLWithPath:@"<filepath>"];
// put.uploadingData = <NSData *>; // 直接上传NSData
// 设置Content-Type，可选
put.contentType = @"application/octet-stream";
// 设置MD5校验，可选
put.contentMd5 = [OSSUtil base64Md5ForFilePath:@"<filePath>"]; // 如果是文件路径
// put.contentMd5 = [OSSUtil base64Md5ForData:<NSData *>]; // 如果是二进制数据
// 进度设置，可选
put.uploadProgress = ^(int64_t bytesSent, int64_t totalByteSent, int64_t totalBytesExpectedTo
Send) {
    // 当前上传段长度、当前已经上传总长度、一共需要上传的总长度
    NSLog(@"%lld, %lld, %lld", bytesSent, totalByteSent, totalBytesExpectedToSend);
};
OSSTask * putTask = [client putObject:put];
[putTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"upload object success!");
    } else {
        NSLog(@"upload object failed, error: %@" , task.error);
    }
    return nil;
}];
// [putTask waitUntilFinished];
// [put cancel];
```

追加上传

Append Object以追加写的方式上传文件。通过Append Object操作创建的Object类型为Appendable Object，而通过Put Object上传的Object是Normal Object。

```
OSSAppendObjectRequest * append = [OSSAppendObjectRequest new];
```

```
// 必填字段
append.bucketName = @"<bucketName>";
append.objectKey = @"<objectKey>";
append.appendPosition = 0; // 指定从何处进行追加
NSString * docDir = [self getDocumentDirectory];
append.uploadingFileURL = [NSURL fileURLWithPath:@"<filepath>"];
// 可选字段
append.uploadProgress = ^(int64_t bytesSent, int64_t totalByteSent, int64_t totalBytes
ExpectedToSend) {
    NSLog(@"%lld, %lld, %lld", bytesSent, totalByteSent, totalBytesExpectedToSend);
};
// 以下可选字段的含义参考：https://docs.aliyun.com/#/pub/oss/api-reference/object&
AppendObject
// append.contentType = @"";
// append.contentMd5 = @"";
// append.contentEncoding = @"";
// append.contentDisposition = @"";
OSSTask * appendTask = [client appendObject:append];
[appendTask continueWithBlock:^id(OSSTask *task) {
    NSLog(@"objectKey: %@", append.objectKey);
    if (!task.error) {
        NSLog(@"append object success!");
        OSSAppendObjectResult * result = task.result;
        NSString * etag = result.eTag;
        long nextPosition = result.xOssNextAppendPosition;
    } else {
        NSLog(@"append object failed, error: %@" , task.error);
    }
    return nil;
}];
```

上传后回调通知

客户端在上传Object时可以指定OSS服务端在处理完上传请求后，通知您的业务服务器，在该服务器确认接收了该回调后将回调的结果返回给客户端。因为加入了回调请求和响应的过程，相比简单上传，使用回调通知机制一般会导致客户端花费更多的等待时间。

代码示例：

```
OSSPutObjectRequest * request = [OSSPutObjectRequest new];
request.bucketName = @"<bucketName>";
request.objectKey = @"<objectKey>";
request.uploadingFileURL = [NSURL fileURLWithPath:@<filepath>"];
// 设置回调参数
request.callbackParam = @{
                          @"callbackUrl": @"<your server callback address>",
                          @"callbackBody": @"<your callback body>"
                          };
// 设置自定义变量
request.callbackVar = @{
                        @"<var1>": @"<value1>",
                        @"<var2>": @"<value2>"
                        };
request.uploadProgress = ^(int64_t bytesSent, int64_t totalByteSent, int64_t totalBytes
ExpectedToSend) {
    NSLog(@"%lld, %lld, %lld", bytesSent, totalByteSent, totalBytesExpectedToSend);
};
OSSTask * task = [client putObject:request];
```

```
[task continueWithBlock:^id(OSSTask *task) {
    if (task.error) {
        OSSLogError(@"%@", task.error);
    } else {
        OSSPutObjectResult * result = task.result;
        NSLog(@"Result - requestId: %@, headerFields: %@, servercallback: %@",
              result.requestId,
              result.httpResponseHeaderFields,
              result.serverReturnJsonString);
    }
    return nil;
}];
```

在无线网络下，上传比较大的文件持续时间长，可能会遇到因为网络条件差、用户切换网络等原因导致上传中途失败，整个文件需要重新上传。为此，SDK提供了断点上传功能。

这个功能依赖OSS的分片上传接口实现，它不会在本地保存任何信息。在上传大文件前，您需要调用分片上传的初始化接口获得UploadId，然后持有这个UploadId调用断点上传接口，将文件上传。如果上传异常中断，那么，持有同一个UploadId，继续调用这个接口上传该文件，上传会自动从上次中断的地方继续进行。

如果上传已经成功，UploadId会失效，如果继续拿着这个UploadId上传文件，会遇到Domain为OSSClientErrorDomain，Code为OSSClientErrorCodeCannotResumeUpload的NSError，这时，需要重新获取新的UploadId上传文件。

也就是说，您需要自行保存和管理与您文件对应的UploadId。UploadId的获取方式参考分片上传 章节。

断点续传失败时，如果同一任务一直得不到续传，可能会在OSS上积累无用碎片。对这种情况，可以为Bucket设置lifeCycle规则，定时清理碎片。

断点续传的实现依赖 InitMultipartUpload/UploadPart/ListParts/CompleteMultipartUpload/AbortMultipartUpload，如果采用STS鉴权模式，请注意加上这些API所需的权限。

断点上传同样支持上传后回调通知，用法和上述普通上传回调相同。

说明：

对于移动端来说，如果不是比较大的文件，不建议使用这种方式上传，因为断点续传是通过分片上传实现的，上传单个文件需要进行多次网络请求，效率不高。

```
__block NSString * uploadId = nil;
OSSInitMultipartUploadRequest * init = [OSSInitMultipartUploadRequest new];
init.bucketName = <bucketName>;
init.objectKey = <objectKey>;
// 以下可选字段的含义参考：https://docs.aliyun.com/#/pub/oss/api-reference/multipart-upload&InitiateMultipartUpload
// append.contentType = @"";
```

```
// append.contentMd5 = @"";
// append.contentEncoding = @"";
// append.contentDisposition = @"";
// init.objectMeta = [NSMutableDictionary dictionaryWithObjectsAndKeys:@"value1", @"x-oss-
meta-name1", nil];
// 先获取到用来标识整个上传事件的UploadId
OSSTask * task = [client multipartUploadInit:init];
[[task continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSInitMultipartUploadResult * result = task.result;
        uploadId = result.uploadId;
    } else {
        NSLog(@"init uploadid failed, error: %@", task.error);
    }
    return nil;
}] waitUntilFinished];
// 获得UploadId进行上传，如果任务失败并且可以续传，利用同一个UploadId可以上传同一文件
到同一个OSS上的存储对象
OSSResumableUploadRequest * resumableUpload = [OSSResumableUploadRequest new];
resumableUpload.bucketName = <bucketName>;
resumableUpload.objectKey = <objectKey>;
resumableUpload.uploadId = uploadId;
resumableUpload.partSize = 1024 * 1024;
resumableUpload.uploadProgress = ^(int64_t bytesSent, int64_t totalByteSent, int64_t
totalBytesExpectedToSend) {
    NSLog(@"%lld, %lld, %lld", bytesSent, totalByteSent, totalBytesExpectedToSend);
};
resumableUpload.uploadingFileURL = [NSURL fileURLWithPath:<your file path>];
OSSTask * resumeTask = [client resumableUpload:resumableUpload];
[resumeTask continueWithBlock:^id(OSSTask *task) {
    if (task.error) {
        NSLog(@"error: %@", task.error);
        if ([task.error.domain isEqualToString:OSSClientErrorDomain] && task.error.code ==
OSSClientErrorCodeCannotResumeUpload) {
            // 该任务无法续传，需要获取新的uploadId重新上传
        }
    } else {
        NSLog(@"Upload file success");
    }
    return nil;
}];
// [resumeTask waitUntilFinished];
// [resumableUpload cancel];
```

1.2.4.7 下载文件

简单下载

下载文件，可以指定下载为本地文件，或者下载为NSData：

```
OSSGetObjectRequest * request = [OSSGetObjectRequest new];
// 必填字段
request.bucketName = @"<bucketName>";
request.objectKey = @"<objectKey>";
// 可选字段
request.downloadProgress = ^(int64_t bytesWritten, int64_t totalBytesWritten, int64_t totalBytes
ExpectedToWrite) {
    // 当前下载段长度、当前已经下载总长度、一共需要下载的总长度
    NSLog(@"%lld, %lld, %lld", bytesWritten, totalBytesWritten, totalBytesExpectedToWrite);
```

```
};
// request.range = [[OSSRange alloc] initWithStart:0 withEnd:99]; // bytes=0-99，指定范围下载
// request.downloadToFileURL = [NSURL fileURLWithPath:@"<filepath>"]; // 如果需要直接下载到文件，需要指明目标文件地址
OSSTask * getTask = [client getObject:request];
[getTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"download object success!");
        OSSGetObjectResult * getResult = task.result;
        NSLog(@"download result: %@", getResult.dowloadedData);
    } else {
        NSLog(@"download object failed, error: %@" ,task.error);
    }
    return nil;
}];
// [getTask waitUntilFinished];
// [request cancel];
```

图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处理服务。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义
- 通过级联处理调用多个图片处理功能

SDK中使用时，只需要在下载图片时，为 request 设置 xOssProcess 属性。示例：

```
OSSGetObjectRequest * request = [OSSGetObjectRequest new];
request.bucketName = @"<bucketName>";
request.objectKey = @"example.jpg";
// 图片处理
request.xOssProcess = @"image/resize,m_lfit,w_100,h_100";
OSSTask * getTask = [client getObject:request];
[getTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"download image success!");
        OSSGetObjectResult * getResult = task.result;
        NSLog(@"download image data: %@", getResult.dowloadedData);
    } else {
        NSLog(@"download object failed, error: %@" ,task.error);
```

```
    }
    return nil;
}];
// [getTask waitUntilFinished];
// [request cancel];
```

需要对图片进行其他处理，只要替换 request.xOssProcess 的值就可以了。需要帮助，请参考：

- 可视化图片处理工具 *ImageStyleViever* ，可以直观的看到OSS图片处理的结果
- OSS图片处理的功能、使用演示 页面

流式下载

实际上，SDK没有提供stream类型的下载接口，但是提供了类似 NSURLSession 库的 didRecieve Data 的分段回调功能，下载时，每次得到一段数据，会回调这个函数进行通知。

说明：

如果设置了这个回调，下载的结果将不再包含实际数据。

```
OSSGetObjectRequest * request = [OSSGetObjectRequest new];
// required
request.bucketName = @"<bucketName>";
request.objectKey = @"<objectKey>";
// 分段回调函数
request.onRecieveData = ^(NSData * data) {
    NSLog(@"Recieve data, length: %ld", [data length]);
};
OSSTask * getTask = [client getObject:request];
[getTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"download object success!");
    } else {
        NSLog(@"download object failed, error: %@" ,task.error);
    }
    return nil;
}];
// [getTask waitUntilFinished];
// [request cancel];
```

指定范围下载

您可以在下载文件时指定一段范围，对于较大的Object适于使用此功能；如果在请求头中使用 Range参数，则返回消息中会包含整个文件的长度和此次返回的范围。

```
OSSGetObjectRequest * request = [OSSGetObjectRequest new];
request.bucketName = @"<bucketName>;
request.objectKey = @"<objectKey>";
request.range = [[OSSRange alloc] initWithStart:1 withEnd:99]; // bytes=1-99
// request.range = [[OSSRange alloc] initWithStart:-1 withEnd:99]; // bytes=-99
// request.range = [[OSSRange alloc] initWithStart:10 withEnd:-1]; // bytes=10-
request.downloadProgress = ^(int64_t bytesWritten, int64_t totalBytesWritten, int64_t totalBytes
ExpectedToWrite) {
    NSLog(@"%lld, %lld, %lld", bytesWritten, totalBytesWritten, totalBytesExpectedToWrite);
```

```
};
OSSTask * getTask = [client getObject:request];
[getTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"download object success!");
        OSSGetObjectResult * getResult = task.result;
        NSLog(@"download result: %@", getResult.dowloadedData);
    } else {
        NSLog(@"download object failed, error: %@" ,task.error);
    }
    return nil;
}];
// [getTask waitUntilFinished];
// [request cancel];
```

只获取文件元信息

通过headObject方法可以只获文件元信息而不获取文件的实体。代码如下：

```
OSSHeadObjectRequest * request = [OSSHeadObjectRequest new];
request.bucketName = @"<bucketName>;
request.objectKey = @"<objectKey>";
OSSTask * headTask = [client headObject:request];
[headTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"head object success!");
        OSSHeadObjectResult * result = task.result;
        NSLog(@"header fields: %@", result.httpResponseHeaderFields);
        for (NSString * key in result.objectMeta) {
            NSLog(@"ObjectMeta: %@ - %@", key, [result.objectMeta objectForKey:key]);
        }
    } else {
        NSLog(@"head object failed, error: %@" ,task.error);
    }
    return nil;
}];
```

1.2.4.8 授权访问

SDK支持签名出特定有效时长或者公开的URL，用于转给第三方实现授权访问。

签名私有资源的指定有效时长的访问URL

如果Bucket或Object不是公共可读的，那么需要调用以下接口，获得签名后的URL：

```
NSString * constrainURL = nil;

// sign constrain url
OSSTask * task = [client presignConstrainURLWithBucketName:@"<bucket name>"
                                          withObjectKey:@"<object key>"
                                     withExpirationInterval: 30 * 60];
if (!task.error) {
    constrainURL = task.result;
} else {
    NSLog(@"error: %@", task.error);
```

```
}
```

签名公开的访问URL

如果Bucket或Object是公共可读的，那么调用一下接口，获得可公开访问Object的URL：

```
NSString * publicURL = nil;

// sign public url
task = [client presignPublicURLWithBucketName:@"<bucket name>"
                                withObjectKey:@"<object key>"];
if (!task.error) {
    publicURL = task.result;
} else {
    NSLog(@"sign url error: %@", task.error);
}
```

1.2.4.9 分片上传

本节演示通过分片上传文件的整个流程。

初始化分片上传

```
__block NSString * uploadId = nil;
__block NSMutableArray * partInfos = [NSMutableArray new];

NSString * uploadToBucket = @"<bucketName>";
NSString * uploadObjectkey = @"<objectKey>";

OSSInitMultipartUploadRequest * init = [OSSInitMultipartUploadRequest new];
init.bucketName = uploadToBucket;
init.objectKey = uploadObjectkey;

// init.contentType = @"application/octet-stream";

OSSTask * initTask = [client multipartUploadInit:init];

[initTask waitUntilFinished];

if (!initTask.error) {
    OSSInitMultipartUploadResult * result = initTask.result;
    uploadId = result.uploadId;
} else {
    NSLog(@"multipart upload failed, error: %@", initTask.error);
    return;
}
```

- OSSInitMultipartUploadRequest指定上传文件的所属存储空间Bucket和文件名字。
- multipartUploadInit发挥的结果中包含UploadId，是区分分片上传的唯一标示，后面的操作中会用到。

上传分片

```
for (int i = 1; i <= 3; i++) {
    OSSUploadPartRequest * uploadPart = [OSSUploadPartRequest new];
    uploadPart.bucketName = uploadToBucket;
```

```
    uploadPart.objectkey = uploadObjectkey;
    uploadPart.uploadId = uploadId;
    uploadPart.partNumber = i; // part number start from 1

    uploadPart.uploadPartFileURL = [NSURL URLWithString:@"<filepath>"];
    // uploadPart.uploadPartData = <NSData *>;

    OSSTask * uploadPartTask = [client uploadPart:uploadPart];

    [uploadPartTask waitUntilFinished];

    if (!uploadPartTask.error) {
        OSSUploadPartResult * result = uploadPartTask.result;
        uint64_t fileSize = [[[NSFileManager defaultManager] attributesOfItemAtPath:uploadPart.
uploadPartFileURL.absoluteString error:nil] fileSize];
        [partInfos addObject:[OSSPartInfo partInfoWithPartNum:i eTag:result.eTag size:fileSize]];
    } else {
        NSLog(@"upload part error: %@", uploadPartTask.error);
        return;
    }
}
```

上述代码调用uploadPart来上传每一个分片，注意：

- 每一个分片上传请求需指定UploadId和PartNum。
- uploadPart要求除最后一个Part外，其他的Part大小都要大于100KB。但是Upload Part接口并不会立即校验上传。Part的大小（因为不知道是否为最后一块）；只有当Complete Multipart Upload的时候才会校验。
- Part号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
- 每次上传part时都要把流定位到此次上传片开头所对应的位置。
- 每次上传part之后，OSS的返回结果会包含一个分片的ETag，值为part数据的MD5值，您需要将它和块编号组合成PartEtag保存，后续完成分片上传需要用到。

完成分片上传

下面代码中的 partInfos就是进行分片上传中保存的partETag的列表，OSS收到用户提交的Part列表后，会逐一验证每个数据Part的有效性。当所有的数据Part验证通过后，OSS会将这些part组合成一个完整的文件。

```
OSSCompleteMultipartUploadRequest * complete = [OSSCompleteMultipartUploadRequest
new];
complete.bucketName = uploadToBucket;
complete.objectKey = uploadObjectkey;
complete.uploadId = uploadId;
complete.partInfos = partInfos;

OSSTask * completeTask = [client completeMultipartUpload:complete];

[[completeTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSCompleteMultipartUploadResult * result = task.result;
```

```
        // ...
    } else {
        // ...
    }
    return nil;
}] waitUntilFinished];
```

完成分片上传（设置ServerCallback）

完成分片上传请求可以设置Server Callback参数，请求完成后会向指定的Server Adress发送回调请求；可通过查看返回结果的result.serverReturnJsonString，查看servercallback结果。

```
OSSCompleteMultipartUploadRequest * complete = [OSSCompleteMultipartUploadRequest
new];
complete.bucketName = @"<bucketName>";
complete.objectKey = @"<objectKey>";
complete.uploadId = uploadId;
complete.partInfos = partInfos;
complete.callbackParam = @{
                        @"callbackUrl": @"<server address>",
                        @"callbackBody": @"<test>"
                        };
complete.callbackVar = @{
                       @"var1": @"value1",
                       @"var2": @"value2"
                       };
OSSTask * completeTask = [client completeMultipartUpload:complete];

[[completeTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSCompleteMultipartUploadResult * result = task.result;
        NSLog(@"server call back return : %@", result.serverReturnJsonString);
    } else {
        // ...
    }
    return nil;
}] waitUntilFinished];
```

删除分片上传事件

下面代码取消了对应UploadId的分片上传请求。

```
OSSAbortMultipartUploadRequest * abort = [OSSAbortMultipartUploadRequest new];
abort.bucketName = @"<bucketName>";
abort.objectKey = @"<objectKey>";
abort.uploadId = uploadId;

OSSTask * abortTask = [client abortMultipartUpload:abort];

[abortTask waitUntilFinished];

if (!abortTask.error) {
    OSSAbortMultipartUploadResult * result = abortTask.result;
    uploadId = result.uploadId;
} else {
    NSLog(@"multipart upload failed, error: %@", abortTask.error);
    return;
```

```
}
```

罗列分片

调用listParts方法获取某个上传事件所有已上传的分片。

```
OSSListPartsRequest * listParts = [OSSListPartsRequest new];
listParts.bucketName = @"<bucketName>";
listParts.objectKey = @"<objectkey>";
listParts.uploadId = @"<uploadid>";

OSSTask * listPartTask = [client listParts:listParts];

[listPartTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"list part result success!");
        OSSListPartsResult * listPartResult = task.result;
        for (NSDictionary * partInfo in listPartResult.parts) {
            NSLog(@"each part: %@", partInfo);
        }
    } else {
        NSLog(@"list part result error: %@", task.error);
    }
    return nil;
}];
```

说明：

默认情况下，如果存储空间中的分片上传事件的数量大于1000，则只会返回1000个Multipart Upload信息，且返回结果中 IsTruncated 为false，并返回 NextPartNumberMarker作为下此读取的起点。

1.2.4.10 管理文件

罗列Bucket所有Object

```
OSSGetBucketRequest * getBucket = [OSSGetBucketRequest new];
getBucket.bucketName = @"<bucketName>";
// 可选参数，具体含义参考：https://docs.aliyun.com/#/pub/oss/api-reference/bucket&GetBucket
// getBucket.marker = @"";
// getBucket.prefix = @"";
// getBucket.delimiter = @"";
OSSTask * getBucketTask = [client getBucket:getBucket];
[getBucketTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSGetBucketResult * result = task.result;
        NSLog(@"get bucket success!");
        for (NSDictionary * objectInfo in result.contents) {
            NSLog(@"list object: %@", objectInfo);
        }
    } else {
        NSLog(@"get bucket failed, error: %@", task.error);
    }
    return nil;
```

```
}];
```

罗列操作具体可设置的参数名称和作用如下：

名称	作用
delimiter	用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素： CommonPrefixes。
marker	设定结果从marker之后按字母排序的第一个开始返回。
maxkeys	限定此次返回object的最大数，如果不设定，默认为100，maxkeys取值不能大于1000。
prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix。

检查文件是否存在

SDK提供了同步接口检测某个指定Object是否在OSS上：

```
NSError * error = nil;
BOOL isExist = [client doesObjectExistInBucket:TEST_BUCKET withObjectKey:@"file1m"
withError:&error];
if (!error) {
    if(isExist) {
        NSLog(@"File exists.");
    } else {
        NSLog(@"File not exists.");
    }
} else {
    NSLog(@"Error!");
}
```

复制Object

```
OSSCopyObjectRequest * copy = [OSSCopyObjectRequest new];
copy.bucketName = @<bucketName>;
copy.objectKey = @"<objectKey>";
copy.sourceCopyFrom = [NSString stringWithFormat:@"/%@/%@", @"<bucketName>, @"<
objectKey_copyFrom>"];
OSSTask * task = [client copyObject:copy];
[task continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        // ...
    }
    return nil;s
}];
```

上述代码实现了CopyObject，注意：

- 源Object和目标Object必须属于同一个数据中心。
- 如果拷贝操作的源Object地址和目标Object地址相同，可以修改已有Object的meta信息。

- 拷贝文件大小不能超过1G，超过1G需使用Multipart Upload操作。

删除Object

下面代码实现了DeleteObject，要求对所在的Bucket有写权限。

```
OSSDeleteObjectRequest * delete = [OSSDeleteObjectRequest new];
delete.bucketName = @"<bucketName>";
delete.objectKey = @"<objectKey>";
OSSTask * deleteTask = [client deleteObject:delete];
[deleteTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        // ...
    }
    return nil;
}];
// [deleteTask waitUntilFinished];
```

只获取Object的Meta信息

下面代码用于获取Object的元信息：

```
OSSHeadObjectRequest * head = [OSSHeadObjectRequest new];
head.bucketName = @"<bucketName>";
head.objectKey = @"<objectKey>";
OSSTask * headTask = [client headObject:head];
[headTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSHeadObjectResult * headResult = task.result;
        NSLog(@"all response header: %@", headResult.httpResponseHeaderFields);
        // some object properties include the 'x-oss-meta-*'s
        NSLog(@"head object result: %@", headResult.objectMeta);
    } else {
        NSLog(@"head object error: %@", task.error);
    }
    return nil;
}];
```

1.2.4.11 管理Bucket

创建bucket

```
OSSCreateBucketRequest * create = [OSSCreateBucketRequest new];
create.bucketName = @"<bucketName>";
create.xOssACL = @"public-read";
create.location = @"oss-cn-hangzhou";
OSSTask * createTask = [client createBucket:create];
[createTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"create bucket success!");
    } else {
        NSLog(@"create bucket failed, error: %@", task.error);
    }
    return nil;
```

```
}];
```

上述代码在创建bucket时，指定了Bucket的ACL和所在的数据中心。

- 每个用户的Bucket数量不能超过10个。
- 每个Bucket的名字全局唯一，也就是说创建的Bucket不能和其他用户已经在使用的Bucket同名，否则会创建失败。
- 创建的时候可以选择Bucket ACL权限，如果不设置ACL，默认是private。
- 创建成功结果返回Bucket所在数据中心。

罗列所有bucket

```
OSSGetServiceRequest * getService = [OSSGetServiceRequest new];
OSSTask * getServiceTask = [client getService:getService];
[getServiceTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSGetServiceResult * result = task.result;
        NSLog(@"buckets: %@", result.buckets);
        NSLog(@"owner: %@, %@", result.ownerId, result.ownerDispName);
        [result.buckets enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger idx, BOOL *
_Nonnull stop) {
            NSDictionary * bucketInfo = obj;
            NSLog(@"BucketName: %@", [bucketInfo objectForKey:@"Name"]);
            NSLog(@"CreationDate: %@", [bucketInfo objectForKey:@"CreationDate"]);
            NSLog(@"Location: %@", [bucketInfo objectForKey:@"Location"]);
        }];
    }
    return nil;
}];
```

上处代码返回请求者拥有的所有Bucket。

匿名访问不支持该操作。

罗列bucket中的文件

```
OSSGetBucketRequest * getBucket = [OSSGetBucketRequest new];
getBucket.bucketName = @"<bucketName>";
// getBucket.marker = @"";
// getBucket.prefix = @"";
// getBucket.delimiter = @"";
OSSTask * getBucketTask = [client getBucket:getBucket];
[getBucketTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        OSSGetBucketResult * result = task.result;
        NSLog(@"get bucket success!");
        for (NSDictionary * objectInfo in result.contents) {
            NSLog(@"list object: %@", objectInfo);
        }
    } else {
        NSLog(@"get bucket failed, error: %@", task.error);
    }
    return nil;
```

```
}];
```

上述代码罗列了Bucket中的文件。

- 罗列操作必须具备访问该Bukcet的权限。
- 罗列时，可以通过prefix，marker，delimiter和max-keys对list做限定，返回部分结果。

删除bucket

```
OSSDeleteBucketRequest * delete = [OSSDeleteBucketRequest new];
delete.bucketName = @"<bucketName>";
OSSTask * deleteTask = [client deleteBucket:delete];
[deleteTask continueWithBlock:^id(OSSTask *task) {
    if (!task.error) {
        NSLog(@"delete bucket success!");
    } else {
        NSLog(@"delete bucket failed, error: %@", task.error);
    }
    return nil;
}];
```

上述代码删除了一个Bucket。

- 只有Bucket的拥有者才能删除这个Bucket。
- 为了防止误删除的发生，OSS不允许用户删除一个非空的Bucket。

1.2.4.12 异常响应

SDK中发生的异常分为两类：ClientError和ServerError。其中前者指的是参数错误、网络错误等，后者指OSS Server返回的异常响应。

Error类型	Error Domain	Code	UserInfo
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeNetworkingFailWithResponseCode0	连接异常
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeSignFailed	签名失败
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeFileCantWrite	文件无法写入
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeInvalidArgument	参数非法
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeNilUploadid	断点续传任务未获取到uploadId

Error类型	Error Domain	Code	UserInfo
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeTaskCancelled	任务被取消
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeNetworkError	网络异常
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeCannotResumeUpload	断点上传失败，无法继续上传
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeNetworkError	本地系统异常
ClientError	com.aliyun.oss.clientError	OSSClientErrorCodeNotKnown	未知异常
ServerError	com.aliyun.oss.serverError	(-1 * httpResponseCode)	解析响应XML得到的Dictionary

1.2.5 .NET-SDK

1.2.5.1 前言

简介

- OSS C# SDK适用的 .NET Framework 版本为2.0及其以上。
- 本文档主要介绍OSS C# SDK的安装和使用，针对于OSS C# SDK版本 2.4.0。
- 并且假设您已经开通了阿里云OSS服务，并创建了 **AccessKeyId** 和 **AccessKeySecret**。
- 如果您还没有开通或者还不了解阿里云OSS服务，请登录*OSS产品主页*了解。
- 如果还没有创建 **AccessKeyId** 和 **AccessKeySecret**，请到阿里云控制台上创建。

SDK下载

- 当前最新版本：*2.4.0*
- SDK包：*aliyun_dotnet_sdk_2.4.0*
- 源代码：*GitHub*

兼容性

对于2.x.x 系列SDK：

- 接口：兼容

- 命名空间：兼容

对于1.0.x 系列SDK：

- 接口：兼容
- 命名空间：不兼容，Aliyun.OpenServices.OpenStorageService变更为Aliyun.OSS

1.2.5.2 安装

SDK安装

版本依赖

Windows

- 适用于 .NET 2.0 及以上版本
- 适用于 Visual Studio 2010 及以上版本

Linux / Mac

- 适用于 Mono 3.12 及以上版本

版本迭代

版本迭代详情，请参考*Change Log*。

Windows环境安装

NuGet安装

- 如果您的Visual Studio没有安装NuGet，请先安装 *NuGet*。
- 安装好NuGet后，先在 Visual Studio 中新建或者打开已有的项目，然后选择**工具** > **NuGet程序包管理器** > **管理解决方案的NuGet程序包**。
- 搜索 **aliyun.oss.sdk**，在结果中找到 **Aliyun.OSS.SDK**，选择最新版本，点击**安装**，成功后添加到项目应用中。

GitHub安装

- 如果没有安装git，请先安装 *git*
- git clone：*https://github.com/aliyun/aliyun-oss-csharp-sdk.git*
- 下载好源码后，按照**项目引入方式安装**即可

DLL引用方式安装

- 下载SDK包 *aliyun_csharp_sdk_2_4_0.zip*，解压后bin目录包括了Aliyun.OSS.dll文件。

- 在Visual Studio的**解决方案资源管理器**中选择您的项目，然后右键**项目名称** > **引用**，在弹出的菜单中选择**添加引用**，在弹出**添加引用**对话框后，选择**浏览**，找到SDK包解压的目录，在bin目录下选中 *Aliyun.OSS.dll*文件，点击**确定**即可。

项目引入方式安装

- 如果是下载了SDK包或者从GitHub上下载了源码，希望源码安装，可以右键**解决方案**，在弹出的菜单中点击**添加** > **现有项目**。
- 在弹出的对话框中选择 *aliyun-oss-sdk.csproj*文件，点击打开。
- 接下来右键**您的项目** > **引用**，选择**添加引用**，在弹出的对话框选择**项目**选项卡后选中 *aliyun-oss-sdk* 项目，点击**确定**即可。

Unix / Mac环境安装

NuGet安装

- 先在 **Xamarin** 中新建或者打开已有的项目，然后选择 **工具** > **Add NuGet Packages**。
- 搜索 *aliyun.oss.sdk*，在结果中找到 *Aliyun.OSS.SDK*，选择最新版本，点击 **Add Package**，成功后添加到项目应用中。

GitHub安装

- 如果没有安装git，请先安装 *git*
- git clone：*https://github.com/aliyun/aliyun-oss-csharp-sdk.git*
- 下载好源码后，使用Xamarin打开，在Release模式下编译aliyun-oss-sdk项目，生成Aliyun.OSS.dll，然后通过DLL引用方式安装

DLL引用方式安装

- 从下载SDK包*aliyun_csharp_sdk_2_4_0.tar.gz*，解压后bin目录包括了Aliyun.OSS.dll文件。
- 在Xamarin的**解决方案**中选择您的项目，然后右键**项目名称** > **引用**，在弹出的菜单中选择 **Edit References**，在弹出**Edit References**对话框后，选择**.Net Assembly** > **浏览**，找到SDK包解压的目录，在bin目录下选中 *Aliyun.OSS.dll*文件，点击**Open**即可。

示例程序

OSS C# SDK提供丰富的示例程序，方便用户参考或直接使用。您可以从*GitHub*获取示例程序。示例程序包括以下内容：

示例文件	示例内容
PutObjectSample.cs	展示了Object上传的用法
AppendObjectSample.cs	展示了Object追加上传的用法
DoesObjectExistSample.cs	展示了判断Object是否存在的用法
DeleteObjectsSample.cs	展示了Object删除的用法
CopyObjectSample.cs	展示了Object复制的用法
ModifyObjectMetaSample.cs	展示了修改Object Meta的用法
MultipartUploadSample.cs	展示了分片上传的用法
ResumableSample.cs	展示了断点续传上传的用法
GetObjectSample.cs	展示了Object下载的用法
GetObjectByRangeSample.cs	展示了Object范围下载的用法
GetObjectAclSample.cs	展示了获取Object访问权限的用法
SetObjectAclSample.cs	展示了设置Object访问权限的用法
ListObjectsSample.cs	展示了列举Objects的用法
UrlSignatureSample.cs	展示了授权访问的用法
CNameSample.cs	展示使用CName访问OSS的用法
PostPolicySample.cs	展示了PostObject的用法
CreateBucketSample.cs	展示了创建Bucket的用法
DeleteBucketSample.cs	展示了删除Bucket的用法
DoesBucketExistSample.cs	展示了判断Bucket是否存在的用法
ListBucketsSample.cs	展示了列举Bucket的用法
SetBucketAclSample.cs	展示了设置Bucket的访问权限的用法
SetBucketLifecycleSample.cs	展示了设置Bucket中Objects生命周期的用法
SetBucketLoggingSample.cs	展示了设置Bucket访问日志的用法
SetBucketRefererSample.cs	展示了设置Bucket防盗链的用法
SetBucketWetbsiteSample.cs	展示了设置Bucket静态网站托管的用法
SetBucketCorsSample.cs	展示了设置Bucket跨域访问的用法
ImageProcessSample.cs	展示了图片处理的用法

1.2.5.3 初始化

OssClient是OSS服务的C#客户端，它为调用者提供了一系列的方法，可以用来操作，管理存储空间（Bucket）和文件（Object）等。

确定Endpoint

Endpoint是阿里云OSS服务在各个区域的域名地址，目前支持两种形式。

Endpoint类型	解释
OSS区域地址	使用OSS Bucket所在区域地址
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS区域地址

使用OSS Bucket所在区域地址，Endpoint查询可以通过以下两种方式：

- 查询Endpoint与区域对应关系详情。
- 您可以登录阿里云OSS控制台，进入Bucket概览页，Bucket域名的后缀部分：如 bucket-1.oss-cn-hangzhou.aliyuncs.com 的 oss-cn-hangzhou.aliyuncs.com 部分为该Bucket的外网Endpoint。

CNAME

您可以将自己拥有的域名通过CNAME绑定到某个存储空间（Bucket）上，然后通过自己域名访问存储空间内的文件。

比如您要将域名new-image.xxxxx.com绑定到深圳区域的名称为image的存储空间上：

您需要到您的域名xxxxx.com托管商那里设定一个新的域名解析，将 http://new-image.xxxxx.com 解析到 http://image.oss-cn-shenzhen.aliyuncs.com，类型为CNAME。

配置密钥

要接入阿里云OSS，您需要拥有一个有效的 Access Key（包括AccessKeyId和AccessKeySecret）用来进行签名认证。可以通过如下步骤获得：

- 注册阿里云账号
- 申请*AccessKey*

在获取到 AccessKeyId和 AccessKeySecret之后，您可以按照下面步骤进行初始化对接。

新建Client

使用OSS域名新建Client

新建一个OssClient很简单，如下面代码所示：

```
using Aliyun.OSS;
const string accessKeyId = "<your AccessKeyId>";
const string accessKeySecret = "<your AccessKeySecret>";
const string endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
/// <summary>
/// 由用户指定的OSS访问地址、阿里云颁发的AccessKeyId/AccessKeySecret构造一个新的
OssClient实例。
/// </summary>
/// <param name="endpoint">OSS的访问地址。</param>
/// <param name="accessKeyId">OSS的访问ID。</param>
/// <param name="accessKeySecret">OSS的访问密钥。</param>
var ossClient = new OssClient(endpoint, accessKeyId, accessKeySecret);
```

使用自定义域名（CNAME）新建Client

下面的代码让客户端使用CNAME访问OSS服务：

```
using Aliyun.OSS;
using Aliyun.OSS.Common;
// 创建ClientConfiguration实例
var conf = new ClientConfiguration();
// 配置使用Cname
conf.IsCname = true;
/// <summary>
/// 由用户指定的OSS访问地址、、阿里云颁发的AccessKeyId/AccessKeySecret、客户端配置
/// 构造一个新的OssClient实例。
/// </summary>
/// <param name="endpoint">OSS的访问地址。</param>
/// <param name="accessKeyId">OSS的访问ID。</param>
/// <param name="accessKeySecret">OSS的访问密钥。</param>
/// <param name="conf">客户端配置。</param>
var client = new OssClient(endpoint, accessKeyId, accessKeySecret, conf);
```

说明：

使用CNAME时，无法使用ListBuckets接口。

配置Client

如果您想配置OssClient的一些细节的参数，可以在构造OssClient的时候传入ClientConfiguration对象。ClientConfiguration是OSS服务的配置类，可以为客户端配置代理，最大连接数等参数。

设置网络参数

我们可以用ClientConfiguration设置一些网络参数：

```
conf.ConnectionLimit = 512;  //HttpWebRequest最大的并发连接数目
conf.MaxErrorRetry = 3;     //设置请求发生错误时最大的重试次数
```

```
conf.ConnectionTimeout = 300;  //设置连接超时时间
conf.SetCustomEpochTicks(customEpochTicks);        //设置自定义基准时间
```

1.2.5.4 快速入门

在这一章里，我们将学到如何用OSS .NET SDK完成一些基本的操作。

Step-1.初始化一个OssClient

SDK的OSS操作通过OssClient类完成的，下面代码创建一个OssClient对象：

```
using Aliyun.OSS;

/// <summary>
/// 由用户指定的OSS访问地址、阿里云颁发的AccessKeyId/AccessKeySecret构造一个新的
OssClient实例。
/// </summary>
/// <param name="endpoint">OSS的访问地址。</param>
/// <param name="accessKeyId">OSS的访问ID。</param>
/// <param name="accessKeySecret">OSS的访问密钥。</param>
public void CreateClient(string endpoint, string accessKeyId, string accessKeySecret)
{
    var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
}
```

说明：

- 在上面代码中，变量 accessKeyId 与 accessKeySecret 是由系统分配给用户的，用于标识用户，可能属于您的阿里云账号或者RAM账号。

Step-2. 新建存储空间

存储空间（Bucket）是OSS全局命名空间，相当于数据的容器，可以存储若干文件。您可以按照下面的代码新建一个存储空间：

```
using Aliyun.OSS;

/// <summary>
/// 在OSS中创建一个新的存储空间。
/// </summary>
/// <param name="bucketName">要创建的存储空间的名称</param>
public void CreateBucket(string bucketName)
{
    var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
    try
    {
        var bucket = client.CreateBucket(bucketName);

        Console.WriteLine("Create bucket succeeded.");

        Console.WriteLine("Name:{0}", bucket.Name);
    }
    catch (Exception ex)
    {
```

```
        Console.WriteLine("Create bucket failed, {0}", ex.Message);
    }
}
```

Step-3. 上传文件

文件（Object）是OSS中最基本的数据单元，用下面代码可以实现上传文件：

```
using Aliyun.OSS;

/// <summary>
/// 上传指定的文件到指定的OSS的存储空间
/// </summary>
/// <param name="bucketName">指定的存储空间名称</param>
/// <param name="key">文件的在OSS上保存的名称</param>
/// <param name="fileToUpload">指定上传文件的本地路径</param>
public void PutObject(string bucketName, string key, string fileToUpload)
{
    var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

    try
    {
        var result = client.PutObject(bucketName, key, fileToUpload);

        Console.WriteLine("Put object succeeded");
        Console.WriteLine("ETag:{0}", result.ETag);
    }
    catch (Exception es)
    {
        Console.WriteLine("Put object failed, {0}", ex.Message);
    }
}
```

说明：

- 每个上传的文件，都可以指定和此文件关联的ObjectMeta。
- ObjectMeta是用户对该文件的描述，由一系列key-value对组成；
- 为了保证上传文件服务器端与本地一致，用户可以设置Content-MD5，OSS会计算上传数据 的MD5值并与用户上传的MD5值比较，如果不一致返回InvalidDigest错误码。
- 计算出来的Content-MD5需要在上传时设置给ObjectMetadata的ETag。

Step-4. 列出所有文件

当您完成一系列上传后，可能需要查看某个存储空间中有哪些文件，可以通过下面的程序实现：

```
using Aliyun.OSS;

/// <summary>
/// 列出指定存储空间的文件列表
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void ListObjects(string bucketName)
{
```

```
    var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

    try
    {
        var listObjectsRequest = new ListObjectsRequest(bucketName);
        var result = client.ListObjects(listObjectsRequest);

        Console.WriteLine("List object succeeded");
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine(summary.Key);
        }
    }
    catch (Exception es)
    {
        Console.WriteLine("List object failed, {0}", ex.Message);
    }
}
```

Step-5. 获取指定文件

您可以参考下面的代码简单地实现一个文件的获取：

```
using Aliyun.OSS;

var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

/// <summary>
/// 从指定的OSS存储空间中获取指定的文件
/// </summary>
/// <param name="bucketName">要获取的文件所在的存储空间名称</param>
/// <param name="key">要获取的文件在OSS上的名称</param>
/// <param name="fileToDownload">本地存储下载文件的目录<param>
public void GetObject(string bucketName, string key, string fileToDownload)
{
    try
    {
        var object = client.GetObject(bucketName, key);

        //将从OSS读取到的文件写到本地
        using (var requestStream = object.Content)
        {
            byte[] buf = new byte[1024];
            using (var fs = File.Open(fileToDownload, FileMode.OpenOrCreate);
            {
                var len = 0;
                while ((len = requestStream.Read(buf, 0, 1024)) != 0)
                {
                    fs.Write(buf, 0, len);
                }
            }
        }
    }
    catch (Exception es)
    {
        Console.WriteLine("Get object failed, {0}", ex.Message);
    }
```

```
}
```

说明：

- 当调用OssClient的GetObject方法时，会返回一个OssObject的对象，此对象包含了文件的各种信息。
- 通过OssObject的GetObjectContent方法，可以获取返回的文件的输入流，通过读取此输入流获取此文件的内容，在用完之后关闭这个流。

Step-6. 删除指定文件

您可以参考下面的代码实现一个文件的删除：

```
using Aliyun.OSS;

/// <summary>
/// 删除指定的文件
/// </summary>
/// <param name="bucketName">文件所在存储空间的名称</param>
/// <param name="key">待删除的文件名称</param>
public void DeleteObject(string bucketName, string key)
{
    var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
    try
    {
        client.DeleteObject(bucketName, key);
        Console.WriteLine("Delete object succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Delete object failed, {0}", ex.Message);
    }
}
```

示例程序

下面是一个完整的程序，演示了创建存储空间，设置存储空间ACL，查询存储空间的ACL，上传文件，下载文件，查询文件列表，删除文件，删除存储空间等操作。

```
using System;
using System.Collections.Generic;
using Aliyun.OSS;

namespace TaoYe
{
    /// <summary>
    /// 快速入门示例程序
    /// </summary>
    public static class SimpleSamples
    {
        private const string _accessKeyId = "<your AccessKeyId>";
        private const string _accessKeySecret = "<your AccessKeySecret>";
        private const string _endpoint = "<valid host name>";
```

```
        private const string _bucketName = "<your bucket name>";
        private const string _key = "<your key>";
        private const string _fileToUpload = "<your local file path>";

        private static OssClient _client = new OssClient(_endpoint, _accessKeyId, _accessKey
Secret);

        public static void Main(string[] args)
        {
            CreateBucket();
            SetBucketAcl();
            GetBucketAcl();

            PutObject();
            ListObjects();
            GetObject();
            DeleteObject();

            // DeleteBucket();

            Console.WriteLine("Press any key to continue . . . ");
            Console.ReadKey(true);
        }

        /// <summary>
        /// 创建一个新的存储空间
        /// </summary>
        private static void CreateBucket()
        {
            try
            {
                var result = _client.CreateBucket(_bucketName);
                Console.WriteLine("创建存储空间{0}成功", result.Name);
            }
            catch (Exception ex)
            {
                Console.WriteLine("创建存储空间失败，原因：{0}", ex.Message);
            }
        }

        /// <summary>
        /// 上传一个新文件
        /// </summary>
        private static void PutObject()
        {
            try
            {
                _client.PutObject(_bucketName, _key, _fileToUpload);
                Console.WriteLine("上传文件成功");
            }
            catch (Exception ex)
            {
                Console.WriteLine("上传文件失败，原因：{0}", ex.Message);
            }
        }

        /// <summary>
        /// 列出存储空间内的所有文件
        /// </summary>
        private static void ListObjects()
        {
```

```
            try
            {
                var keys = new List<string>();
                ObjectListing result = null;
                string nextMarker = string.Empty;

                /// 由于ListObjects每次最多返回100个结果，所以，这里需要循环去获取，直到返回结
果中IsTruncated为false
                do
                {
                    var listObjectsRequest = new ListObjectsRequest(_bucketName)
                    {
                        Marker = nextMarker,
                        MaxKeys = 100
                    };
                    result = _client.ListObjects(listObjectsRequest);

                    foreach (var summary in result.ObjectSummaries)
                    {
                        keys.Add(summary.Key);
                    }

                    nextMarker = result.NextMarker;
                } while (result.IsTruncated);

                Console.WriteLine("列出存储空间中的文件");
                foreach (var key in keys)
                {
                    Console.WriteLine("文件名称：{0}", key);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("列出存储空间中的文件失败，原因： {0}", ex.Message);
            }

        }

        /// <summary>
        /// 下载文件
        /// </summary>
        private static void GetObject()
        {
            try
            {
                var result = _client.GetObject(_bucketName, _key);
                Console.WriteLine("下载的文件成功，名称是：{0}，长度：{1}", result.Key, result.
Metadata.ContentLength);
            }
            catch (Exception ex)
            {
                Console.WriteLine("下载文件失败，原因：{0}", ex.Message);
            }
        }

        /// <summary>
        /// 删除文件
        /// </summary>
        private static void DeleteObject()
        {
            try
```

```
            {
                _client.DeleteObject(_bucketName, _key);
                Console.WriteLine("删除文件成功");
            }
            catch (Exception ex)
            {
                Console.WriteLine("删除文件失败，原因： {0}", ex.Message);
            }
        }

        /// <summary>
        /// 获取存储空间ACL的值
        /// </summary>
        private static void GetBucketAcl()
        {
            try
            {
                var result = _client.GetBucketAcl(_bucketName);

                foreach (var grant in result.Grants)
                {
                    Console.WriteLine("获取存储空间权限成功，当前权限：{0}", grant.Permission.
ToString());
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("获取存储空间权限失败，原因： {0}", ex.Message);
            }
        }

        /// <summary>
        /// 设置存储空间ACL的值
        /// </summary>
        private static void SetBucketAcl()
        {
            try
            {
                _client.SetBucketAcl(_bucketName, CannedAccessControlList.PublicRead);
                Console.WriteLine("设置存储空间权限成功");
            }
            catch (Exception ex)
            {
                Console.WriteLine("设置存储空间权限失败，原因：{0}", ex.Message);
            }
        }

        /// <summary>
        /// 删除存储空间
        /// </summary>
        private static void DeleteBucket()
        {
            try
            {
                _client.DeleteBucket(_bucketName);
                Console.WriteLine("删除存储空间成功");
            }
            catch (Exception ex)
            {
                Console.WriteLine("删除存储空间失败", ex.Message);
            }
```

```
        }
    }
}
```

1.2.5.5 管理Bucket

存储空间（Bucket）是OSS上的命名空间，也是计费、权限控制、日志记录等高级功能的管理实体。

新建存储空间

如下代码可以新建一个存储空间：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 创建一个新的存储空间（Bucket）
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void CreateBucket(string bucketName)
{
    try
    {
        // 新建一个Bucket
        client.CreateBucket(bucketName);
        Console.WriteLine("Create bucket succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Create bucket failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

说明：

由于存储空间的名字是全局唯一的，所以必须保证您的BucketName不与别人重复。

列出用户所有的存储空间

下面代码可以列出用户所有的存储空间：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 列出账户下所有的存储空间信息
/// </summary>
public void ListBuckets()
{
```

```
    try
    {
        var buckets = client.ListBuckets();
        Console.WriteLine("List bucket succeeded");
        foreach (var bucket in buckets)
        {
            Console.WriteLine("Bucket name：{0}，Location：{1}，Owner：{2}", bucket.Name,
bucket.Location, bucket.Owner);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("List bucket failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

使用CNAME进行访问

当用户将自己的域名CNAME指向自己的一个存储空间的域名后，用户可以使用自己的域名来访问OSS：如果需要使用CNAME，需要将ClientConfigration中的IsCname设置为true。

```
using Aliyun.OSS;
using Aliyun.OSS.Common;
/// <summary>
/// 通过CNAME上传文件
/// </summary>
public void PutObjectByCname()
{
    try
    {
        // 比如你的域名"http://my-cname.com"CNAME指向你的存储空间域名"mybucket.oss-cn-
hangzhou.aliyuncs.com"
        // 创建ClientConfiguration实例
        var conf = new ClientConfiguration();
        // 配置使用Cname
        conf.IsCname = true;
        var client = new OssClient("http://my-cname.com/", accessKeyId, accessKeySecret, conf);
        var result = client.putObject("mybucket", key, fileToUpload);
        Console.WriteLine("Put object succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put object failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。

- 用户只需要在创建OssClinet类实例时，将原本填入该存储空间的endpoint更换成CNAME后的域名，且将ClientConfiguration的参数IsCname设置为true。
- 同时需要注意的是，使用该OssClient实例的后续操作中，存储空间的名称只能填成被指向的存储空间名称。

判断存储空间是否存在

判断存储空间是否存在可以使用以下代码：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 判断存储空间是否存在
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void DoesBucketExist(string bucketName)
{
    try
    {
        var exist = client.DoesBucketExist(bucketName);
        Console.WriteLine("Check object Exist succeeded");
        Console.WriteLine("exist ? {0}", exist);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Check object Exist failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

设置存储空间访问权限

设置存储空间访问权限可以使用以下代码：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 设置存储空间的访问权限
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void SetBucketAcl(string buckteName)
{
    try
    {
        // 指定Bucket ACL为公共读
        client.SetBucketAcl(bucketName, CannedAccessControlList.PublicRead);
        Console.WriteLine("Set bucket ACL succeeded");
    }
    catch (Exception ex)
```

```
    {
        Console.WriteLine("Set bucket ACL failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

获取存储空间访问权限

获取存储空间访问权限可以使用以下代码：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 获取存储空间的访问权限
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void GetBucketAcl(string bucketName)
{
    try
    {
        string bucketName = "your-bucket";
        var acl = client.GetBucketAcl(bucketName);
        Console.WriteLine("Get bucket ACL success");
        foreach (var grant in acl.Grants)
        {
            Console.WriteLine("获取存储空间权限成功，当前权限：{0}", grant.Permission.ToString
());
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Get bucket ACL failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

删除存储空间

下面代码删除了一个存储空间：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 删除存储空间
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void DeleteBucket(string bucketName)
{
```

```
    try
    {
        client.DeleteBucket(bucketName);
        Console.WriteLine("Delete bucket succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Delete bucket failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- 如果存储空间不为空（存储空间中有文件或者分片上传碎片），则存储空间无法删除。
- 必须先删除存储空间中的所有文件后，存储空间才能成功删除。

1.2.5.6 上传文件

在OSS中，用户操作的基本数据单元是文件（Object）。单个文件最大允许大小根据上传数据方式不同而不同，Put Object方式最大不能超过5GB, 使用multipart上传方式文件大小不能超过48.8TB。

简单的上传

上传指定字符串

```
using System.Text;
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret)
try
{
    string str = "a line of simple text";
    byte[] binaryData = Encoding.ASCII.GetBytes(str);
    MemoryStream requestContent = new MemoryStream(binaryData);
    client.PutObject(bucketName, key, requestContent);
    Console.WriteLine("Put object succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Put object failed, {0}", ex.Message);
}
```

说明：

完整代码参考：*GitHub*。

上传指定的本地文件

```
using Aliyun.OSS;

// 初始化OssClient
```

```
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
try
{
    string fileToUpload = "your local file path";
    client.PutObject(bucketName, key, fileToUpload);
    Console.WriteLine("Put object succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Put object failed, {0}", ex.Message);
}
```

说明：

完整代码参考：*GitHub*。

上传文件并且带MD5校验

```
using Aliyun.OSS;
using Aliyun.OSS.util;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
try
{
    string fileToUpload = "your local file path";
    string md5;
    using (var fs = File.Open(fileToUpload, FileMode.Open));
    {
        md5 = OssUtils.ComputeContentMd5(fs, fs.Length);
    }

    var objectMeta = new ObjectMetadata
    {
        ContentMd5 = md5
    };
    client.PutObject(bucketName, key, fileToUpload, objectMeta);
    Console.WriteLine("Put object succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Put object failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*。
- 为了保证SDK发送的数据和OSS服务端接收到的数据一致，可以在ObjectMeta中增加Content-Md5值，这样服务端就会使用这个MD5值做校验。
- 使用Md5时，性能会有所损失。

上传文件带Header

带标准Header

OSS服务允许用户自定义文件的Http Header。下面代码为文件设置了过期时间：

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    using (var fs = File.Open(fileToUpload, FileMode.Open))
    {
        var metadata = new ObjectMetadata();
        metadata.ContentLength = fs.Length;
        metadata.ExpirationTime = DateTime.Parse("2015-10-12T00:00:00.000Z");
        client.PutObject(bucketName, key, fs, metadata);
        Console.WriteLine("Put object succeeded");
    }
}
catch (Exception ex)
{
    Console.WriteLine("Put object failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*。
- .NET SDK支持的Http Header：Cache-Control 、 Content-Disposition 、Content-Encoding 、Expires、Content-Type等。
- 它们的相关介绍见 *RFC2616* 。

带自定义Header

OSS支持用户自定义元信息来对文件进行描述。比如：

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    using (var fs = File.Open(fileToUpload, FileMode.Open))
    {
        var metadata = new ObjectMetadata();
        metadata.UserMetadata.Add("name", "my-data");
        metadata.ContentLength = fs.Length;
        client.PutObject(bucketName, key, fs, metadata);
        Console.WriteLine("Put object succeeded");
    }
}
catch (Exception ex)
{
    Console.WriteLine("Put object failed, {0}", ex.Message);
```

```
}
```

说明：

- 完整代码参考：*GitHub*。
- 在上面代码中，用户自定义了一个名字为**name**，值为**my-data**的元信息。
- 当用户下载此文件的时候，此元信息也可以一并得到。
- 一个文件可以有多个类似的参数，但所有的user meta总大小不能超过2KB。
- 使用上述方法上传最大文件不能超过5G。如果超过可以使用MutipartUpload上传。
- user meta的名称大小写不敏感，比如用户上传文件时，定义名字为**name**的meta,在表头中存储的参数为：**x-oss-meta-name**，所以读取时读取名字为**name**的参数即可。
- 但如果存入参数为**name**，读取时使用**name**读取不到对应信息，会返回**null**。

创建模拟文件夹

OSS服务是没有文件夹这个概念的，所有元素都是以文件来存储。但给用户提供了创建模拟文件夹的方式，代码如下所示：

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    // 重要：这时候作为目录的key必须以斜线（ / ）结尾
    const string key = "yourfolder/";
    // 此时的目录是一个内容为空的文件
    using (var stream = new MemoryStream())
    {
        client.PutObject(bucketName, key, stream);
        Console.WriteLine("Create dir {0} succeeded", key);
    }
}
catch (Exception ex)
{
    Console.WriteLine("Create dir failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*。
- 创建模拟文件夹本质上来说是创建了一个空文件。
- 对于这个文件照样可以上传下载，只是控制台会对以/结尾的文件以文件夹的方式展示。
- 所以用户可以使用上述方式来实现创建模拟文件夹。

异步上传

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

static AutoResetEvent _event = new AutoResetEvent(false);

public static void AsyncPutObject()
{
    try
    {
        using (var fs = File.Open(fileToUpload, FileMode.Open))
        {
            var metadata = new ObjectMetadata();
            metadata.CacheControl = "No-Cache";
            metadata.ContentType = "text/html";
            client.BeginPutObject(bucketName, key, fs, metadata, PutObjectCallback, new string('a
', 8));

            _event.WaitOne();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put object failed, {0}", ex.Message);
    }
}

private static void PutObjectCallback(IAsyncResult ar)
{
    try
    {
        var result = client.EndPutObject(ar);
        Console.WriteLine("ETag:{0}", result.ETag);
        Console.WriteLine("User Parameter:{0}", ar.AsyncState as string);
        Console.WriteLine("Put object succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put object failed, {0}", ex.Message);
    }
    finally
    {
         _event.Set();
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- 使用异步上传时您需要实现自己的回调处理函数。

追加上传

追加写的方式上传文件。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

/// <summary>
/// 追加内容到指定的OSS文件中。
/// </summary>
/// <param name="bucketName">指定的存储空间名称。</param>
/// <param name="key">OSS上会被追加内容的文件名称。</param>
/// <param name="fileToUpload">指定被追加的文件的路径。</param>
public static void AppendObject(string bucketName, string key, string fileToUpload)
{
    //第一次追加文件的时候，文件可能已经存在，先获取文件的当前长度，如果不存在，position
为0
    long position = 0;
    try
    {
        var metadata = client.GetObjectMetadata(bucketName, key);
        position = metadata.ContentLength;
    }
    catch(Exception)  {}

    try
    {
        using (var fs = File.Open(fileToUpload, FileMode.Open))
        {
            var request = new AppendObjectRequest(bucketName, key)
            {
                ObjectMetadata = new ObjectMetadata(),
                Content = fs,
                Position = position
            };

            var result = client.AppendObject(request);
            // 设置下次追加文件时的position位置
            position = result.NextAppendPosition;

            Console.WriteLine("Append object succeeded, next append position:{0}", position);
        }

        // 再次追加文件，这时候的position值可以从上次的结果中获取到
        using (var fs = File.Open(fileToUpload, FileMode.Open))
        {
            var request = new AppendObjectRequest(bucketName, key)
            {
                ObjectMetadata = new ObjectMetadata(),
                Content = fs,
                Position = position
            };

            var result = client.AppendObject(request);
            position = result.NextAppendPosition;

            Console.WriteLine("Append object succeeded, next append position:{0}", position);
        }
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine("Append object failed, {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。

分片上传

除了通过PutObject接口上传文件到OSS以外，OSS还提供了另外一种上传模式：Multipart Upload。用户可以在如下的应用场景内（但不仅限于此），使用Multipart Upload上传模式，如：

- 需要支持断点上传。
- 上传超过100MB大小的文件。
- 网络条件较差，和OSS的服务器之间的链接经常断开。
- 上传文件之前，无法确定上传文件的大小。

下面我们将一步步学习怎样实现Multipart Upload。

分步完成Multipart Upload

初始化Multipart Upload

我们使用 initiateMultipartUpload 方法来初始化一个分片上传事件：

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    string bucketName = "your-bucket-name";
    string key = "your-key";

    // 开始Multipart Upload
    var request = new InitiateMultipartUploadRequest(bucketName, key);
    var result = client.InitiateMultipartUpload(request);

    // 打印UploadId
    Console.WriteLine("Init multi part upload succeeded");
    Console.WriteLine("Upload Id:{0}", result.UploadId);
}
catch (Exception ex)
{
    Console.WriteLine("Init multi part upload failed, {0}", ex.Message);
```

```
}
```

说明：

- 完整代码参考：*GitHub*。
- 我们用InitiateMultipartUploadRequest来指定上传文件的名字和所属存储空间（Bucket）。
- 在InitiateMultipartUploadRequest中，您也可以设置ObjectMeta，但是不必指定其中的ContentLength 。
- initiateMultipartUpload 的返回结果中含有UploadId ，它是区分分片上传事件的唯一标识，在后面的操作中，我们将用到它。

Upload Part本地上传

我们把本地文件分片上传。假设有一个文件，本地路径为 /path/to/file.zip 由于文件比较大，我们将其分片传输到OSS中。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

// 计算片个数
var fi = new FileInfo(fileToUpload);
var fileSize = fi.Length;
var partCount = fileSize / partSize;
if (fileSize % partSize != 0)
{
    partCount++;
}

// 开始分片上传
try
{
    var partETags = new List<PartETag>();
    using (var fs = File.Open(fileToUpload, FileMode.Open))
    {
        for (var i = 0; i < partCount; i++)
        {
            var skipBytes = (long)partSize * i;

            //定位到本次上传片应该开始的位置
            fs.Seek(skipBytes, 0);

            //计算本次上传的片大小，最后一片为剩余的数据大小，其余片都是part size大小。
            var size = (partSize < fileSize - skipBytes) ? partSize : (fileSize - skipBytes);
            var request = new UploadPartRequest(bucketName, objectName, uploadId)
            {
                InputStream = fs,
                PartSize = size,
                PartNumber = i + 1
            };
```

```
            //调用UploadPart接口执行上传功能，返回结果中包含了这个数据片的ETag值
            var result = _ossClient.UploadPart(request);
            partETags.Add(result.PartETag);
        }
        Console.WriteLine("Put multi part upload succeeded");
    }
}
catch (Exception ex)
{
    Console.WriteLine("Put multi part upload failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*。
- 上面程序的核心是调用UploadPart方法来上传每一个分片，但是要注意以下几点：
  - UploadPart 方法要求除最后一个Part以外，其他的Part大小都要大于100KB。但是Upload Part接口并不会立即校验上传 Part的大小（因为不知道是否为最后一片）；只有当Complete Multipart Upload的时候才会校验。
  - OSS会将服务器端收到Part数据的MD5值放在ETag头内返回给用户。
  - 为了保证数据在网络传输过程中不出现错误，SDK会自动设置Content-MD5，OSS会计算上传数据的MD5值与SDK计算的MD5值比较， 如果不一致返回InvalidDigest错误码。
  - Part号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
  - 每次上传part时都要把流定位到此次上传片开头所对应的位置。
  - 每次上传part之后，OSS的返回结果会包含一个 PartETag 对象，他是上传片的ETag与片编号（PartNumber）的组合，
  - 在后续完成分片上传的步骤中会用到它，因此我们需要将其保存起来。一般来讲我们将这些PartETag 对象保存到List中。

完成分片上传

完成分片上传代码如下：

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    var completeMultipartUploadRequest = new CompleteMultipartUploadRequest(bucketName,
key, uploadId);
    foreach (var partETag in partETags)
    {
        completeMultipartUploadRequest.PartETags.Add(partETag);
```

```
    }
    var result = client.CompleteMultipartUpload(completeMultipartUploadRequest);
    Console.WriteLine("complete multi part succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("complete multi part failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*
- 上面代码中的 partETags 就是进行分片上传中保存的partETag的列表，OSS收到用户提交的Part列表后，会逐一验证每个数据Part的有效性。
- 当所有的数据Part验证通过后，OSS会将这些part组合成一个完整的文件。

取消分片上传事件

我们可以用 AbortMultipartUpload 方法取消分片上传。

```
 using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    var request = new AbortMultipartUpload(bucketName, key, uploadId);
    client.AbortMultipartUpload(request);
    Console.WriteLine("Abort multi part succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Abort multi part failed, {0}", ex.Message);
}
```

说明：

完整代码参考：*GitHub*。

获取存储空间内所有分片上传事件

我们可以用 ListMultipartUploads 方法获取存储空间内所有上传事件。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    // 获取Bucket内所有上传事件
    var request = new ListMultipartUploadsRequest(bucketName);
```

```
    var multipartUploadListing = client.ListMultipartUploads(request);
    Console.WriteLine("List multi part succeeded");

    // 获取各事件信息
    var multipartUploads = multipartUploadListing.MultipartUploads;
    foreach (var mu : multipartUploads)
    {
        Console.WriteLine("Key:{0}, UploadId:{1}", mu.Key , mu.UploadId);
    }

    var commonPrefixes = multipartUploadListing.CommonPrefixes;
    foreach (var prefix : commonPrefixes)
    {
        Console.WriteLine("Prefix:{0}", prefix);
    }
}
catch (Exception ex)
{
    Console.WriteLine("List multi part uploads failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*。
- 默认情况下，如果存储空间中的分片上传事件的数量大于1000，则只会返回1000个文件，且返回结果中 IsTruncated 为 false，返回 NextKeyMarker 和 NextUploadIdMarker 作为下此读取的起点。
- 若想增大返回分片上传事件数目，可以修改 MaxUploads 参数，或者使用 KeyMarker 以及 UploadIdMarker 参数分次读取。

获取所有已上传的片信息

我们可以用 ListParts 方法获取某个上传事件所有已上传的片。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

try
{
    var listPartsRequest = new ListPartsRequest(bucketName, key, uploadId);
    var listPartsResult = client.ListParts(listPartsRequest);

    Console.WriteLine("List parts succeeded");

    // 遍历所有Part
    var parts = listPartsResult.Parts;
    foreach (var part : parts)
    {
        Console.WriteLine("partNumber:{0}, ETag:{1}, Size:{2}", part.PartNumber, part.ETag, part.
Size);
    }
}
```

```
catch (Exception ex)
{
    Console.WriteLine("List parts failed, {0}", ex.Message);
}
```

说明：

- 完整代码参考：*GitHub*。
- 默认情况下，如果存储空间中的分片上传事件的数量大于1000，则只会返回1000个Multipart Upload信息，且返回结果中 IsTruncated 为false，并返回 NextPartNumberMarker作为下此读取的起点。
- 若想增大返回分片上传事件数目，可以修改 MaxParts 参数，或者使用 PartNumberMarker 参数分次读取。

通过断点续传上传

除了支持分片上传外，还提供了断点续传功能，如果某次上传中断，下次可以从上次失败的位置开始上传，以便加快速度。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

/// <summary>
/// 断点续传
/// </summary>
/// <param name="bucketName">指定的存储空间名称。</param>
/// <param name="key">保存到OSS上的名称。</param>
/// <param name="fileToUpload">指定上传文件的路径。</param>
/// <param name="checkpointDir">保存断点续传中间状态文件的目录，如果指定了，则会具有断点续传功能，否则会重新上传</param>
public static void ResumableUploadObject(string bucketName, string key, string fileToUpload, string checkpointDir)
{
    try
    {
        client.ResumableUploadObject(bucketName, key, fileToUpload, null, checkpointDir);

        Console.WriteLine("Resumable upload object:{0} succeeded", key);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Resumable upload object failed, {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。

- checkpointDir目录中会保存断点续传的中间状态，用于失败后，下次继续上传时使用。
- 如果checkpointDir为null，断点续传功能不会生效，每次都会重新上传。

1.2.5.7 下载文件

简单的下载文件

我们可以通过以下代码将文件读取到一个流中：

```
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

/// <summary>
/// 从指定的OSS存储空间中获取指定的文件
/// </summary>
/// <param name="bucketName">要获取的文件所在的存储空间的名称</param>
/// <param name="key">要获取的文件的名称</param>
/// <param name="fileToDownload">文件保存的本地路径</param>
public void GetObject(string bucketName, string key, string fileToDownload)
{
    try
    {
        var object = client.GetObject(bucketName, key);
        using (var requestStream = object.Content)
        {
            byte[] buf = new byte[1024];
            var fs = File.Open(fileToDownload, FileMode.OpenOrCreate);
            var len = 0;
            while ((len = requestStream.Read(buf, 0, 1024)) != 0)
            {
                fs.Write(buf, 0, len);
            }
            fs.Close();
        }

        Console.WriteLine("Get object succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Get object failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- OssObject 包含了文件的各种信息，包含文件所在的存储空间（Bucket）、文件的名称、Metadata以及一个输入流。
- 我们可以通过操作输入流将文件的内容读取到文件或者内存中。而ObjectMeta包含了文件上传时定义的，ETag，Http Header以及自定义的元信息。

分段读取文件

为了实现更多的功能，我们可以通过使用 GetObjectRequest 来读取文件，比如分段读取：

```
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

public void GetObject(string bucketName, string key, string fileToDownload)
{
    try
    {
        var getObjectRequest = new GetObjectRequest(bucketName, key);

        //读取文件中的20到100个字符，包括第20和第100个字符
        getObjectRequest.SetRange(20, 100);

        var object = client.GetObject(getObjectRequest);

        // 将读到的数据写到fielToDownload文件中去
        using (var requestStream = object.Content)
        {
            byte[] buf = new byte[1024];
            var fs = File.Open(fileToDownload, FileMode.OpenOrCreate);
            var len = 0;
            while ((len = requestStream.Read(buf, 0, 1024)) != 0)
            {
                fs.Write(buf, 0, len);
            }
            fs.Close();
        }
        Console.WriteLine("Get object succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Get object failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*
- 我们通过GetObjectRequest的setRange方法设置了返回的文件的范围。
- 我们可以用此功能实现文件的分段下载和断点续传。
- GetObjectRequest可以设置以下参数：

参数	说明
Range	指定文件传输的范围。
ModifiedSinceConstraint	如果指定的时间早于实际修改时间，则正常传送文件。否则抛出304 Not Modified异常。

参数	说明
UnmodifiedSinceConstraint	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件。否则抛出412 precondition failed异常
MatchingETagConstraints	传入一组ETag，如果传入期望的ETag和文件的 ETag匹配，则正常传输文件。否则抛出412 precondition failed异常
NonmatchingEtagConstraints	传入一组ETag，如果传入的ETag值和文件的ETag不匹配，则正常传输文件。否则抛出304 Not Modified异常。
ResponseHeaderOverrides	自定义OSS返回请求中的一些Header。

只获取文件元信息

通过GetObjectMetadata方法可以只获取ObjectMeta而不获取文件的实体。代码如下：

```
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

/// <summary>
/// 获取文件的元信息。
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
/// <param name="key">文件在OSS上的名称</param>
public void GetObjectMetadata(string bucketName, string key)
{
    try
    {
        var metadata = client.GetObjectMetadata(bucketName, key);

        Console.WriteLine("Get object meta succeeded");
        Console.WriteLine("Content-Type:{0}", metadata.ContentType);
        Console.WriteLine("Cache-Control:{0}", metadata.CacheControl);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Get object meta failed. {0}", ex.Message);
    }
}
```

1.2.5.8 管理文件

在OSS中，用户可以通过一系列的接口管理存储空间(Bucket)中的文件(Object)，比如ListObjects，DeleteObject，CopyObject，DoesObjectExist等。

列出存储空间中的文件

简单列出文件

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
```

```
/// <summary>
/// 列出指定存储空间下的文件的摘要信息OssObjectSummary列表
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void ListObjects(string bucketName)
{
    try
    {
        var listObjectsRequest = new ListObjectsRequest(bucketName);
        var result = client.ListObjects(listObjectsRequest);
        Console.WriteLine("List objects succeeded");
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine("File name:{0}", summary.Key);
        }
    }
    catch (Exception ex)
    {
         Console.WriteLine("List objects failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- 默认情况下，如果存储空间中的文件数量大于100，则只会返回100个文件，且返回结果中IsTruncated 为 true，并返回 NextMarker 作为下此读取的起点。
- 若想增大返回文件数目，可以修改MaxKeys参数，或者使用Marker参数分次读取。

带前缀过滤的列出文件

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 列出指定存储空间下其Key以prefix为前缀的文件的摘要信息OssObjectSummary
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
/// <param name="prefix">限定返回的文件必须以此作为前缀</param>
public void ListObjects(string bucketName, string prefix)
{
    try
    {
        var listObjectsRequest = new ListObjectsRequest(bucketName)
        {
            Prefix = prefix
        };
        var result = client.ListObjects(listObjectsRequest);
        Console.WriteLine("List objects succeeded");
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine("File Name:{0}", summary.Key);
        }
    }
    catch (Exception ex)
    {
```

```
        Console.WriteLine("List objects failed. {0}", ex.Message);
    }
}
```

通过异步方式列出文件

```
using Aliyun.OSS;
// 初始化OssClient
static OssClient ossClient = new OssClient(endpoint, accessKeyId, accessKeySecret);
static AutoResetEvent _event = new AutoResetEvent(false);
public static void AsyncListObjects()
{
    try
    {
        var listObjectsRequest = new ListObjectsRequest(bucketName);
        ossClient.BeginListObjects(listObjectsRequest, ListObjectCallback, null);
        _event.WaitOne();
    }
    catch (Exception ex)
    {
        Console.WriteLine("List objects failed. {0}", ex.Message);
    }
}
private static void ListObjectCallback(IAsyncResult ar)
{
    try
    {
        var result = ossClient.EndListObjects(ar);
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine("文件名称：{0}", summary.Key);
        }
        _event.Set();
        Console.WriteLine("List objects succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("List objects failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- 上面的ListObjectCallback方法就是异步调用结束后执行的回调方法，如果使用异步类型的接口，都需要实现类似接口。

通过ListObjectsRequest列出文件

我们可以通过设置ListObjectsRequest的参数来完成更强大的功能。比如：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
```

```
/// 列出指定存储空间下的文件的摘要信息OssObjectSummary列表
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void ListObjects(string bucketName)
{
    try
    {
        var listObjectsRequest = new ListObjectsRequest(bucketName)
        {
            Delimiter = "/",
            Marker = "abc"
        };
        result = client.ListObjects(listObjectsRequest);
        Console.WriteLine("List objects succeeded");
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine("文件名称：{0}", summary.Key);
        }
    }
    catch (Exception ex)
    {
         Console.WriteLine("List objects failed. {0}", ex.Message);
    }
}
```

名称	作用
Delimiter	用于对文件名字进行分组的字符。所有名字包含指定的前缀且第一次出现Delimiter字符之间的文件作为一组元素：CommonPrefixes。
Marker	设定结果从Marker之后按字母排序的第一个开始返回。
MaxKeys	限定此次返回文件的最大数，如果不设定，默认为100，MaxKeys取值不能大于1000。
Prefix	限定返回的文件名称必须以Prefix作为前缀。注意使用Prefix查询时，返回的文件名中仍会包含Prefix。

说明：

- 完整代码参考：*GitHub*。
- 如果需要遍历所有的文件，而文件数量大于100，则需要进行多次迭代。每次迭代时，将上次迭代列取最后一个文件的key作为本次迭代中的Marker即可。

文件夹功能模拟

我们还可以通过 Delimiter 和 Prefix 参数的配合模拟出文件夹功能。Delimiter 和 Prefix 的组合效果是这样的：如果把 Prefix 设为某个文件夹名，就可以罗列以此 Prefix 开头的文件，即该文件夹下递归的所有的文件和子文件夹。如果再把 Delimiter 设置为“/”时，返回值就只罗列该文件夹下的文件，该文件夹下的子文件夹返回在 CommonPrefixes 部分，子文件夹下递归的文件和

文件夹不被显示。假设Bucket中有4个文件： oss.jpg ， fun/test.jpg ， fun/movie/001.avi ， fun/movie/007.avi ，我们把 / 符号作为文件夹的分隔符。

列出存储空间内所有文件

当我们需要获取存储空间下的所有文件时，可以这样写：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public void ListObject(string bucketName)
{
    try
    {
        ObjectListing result = null;
        string nextMarker = string.Empty;
        do
        {
            var listObjectsRequest = new ListObjectsRequest(bucketName)
            {
                Marker = nextMarker,
                MaxKeys = 100
            };
            result = client.ListObjects(listObjectsRequest);
            Console.WriteLine("File:");
            foreach (var summary in result.ObjectSummaries)
            {
                Console.WriteLine("Name:{0}", summary.Key);
            }
            nextMarker = result.NextMarker;
        } while (result.IsTruncated);
    }
    catch (Exception ex)
    {
        Console.WriteLine("List object failed. {0}", ex.Message);
    }
}
```

输出：

```
File:
Name:fun/movie/001.avi
Name:fun/movie/007.avi
Name:fun/test.jpg
Name:oss.jpg
```

递归列出目录下所有文件

我们可以通过设置 Prefix 参数来获取某个目录下所有的文件：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public void ListObject(string bucketName)
{
    try
    {
```

```
        var listObjectsRequest = new ListObjectsRequest(bucketName)
        {
            Prefix = "fun/"
        };
        result = client.ListObjects(listObjectsRequest);
        Console.WriteLine("List object succeeded");
        Console.WriteLine("File:");
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine("Name:{0}", summary.Key);
        }
        Console.WriteLine("Dir:");
        foreach (var prefix in result.CommonPrefixes)
        {
            Console.WriteLine("Name:{0}", prefix);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("List object failed. {0}", ex.Message);
    }
}
```

输出：

```
List object succeeded
File:
Name:fun/movie/001.avi
Name:fun/movie/007.avi
Name:fun/test.jpg
目录：
```

列出目录下的文件和子目录

在 Prefix 和 Delimiter 结合的情况下，可以列出目录下的文件和子目录：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public void ListObjects(string bucketName)
{
    try
    {
        var listObjectsRequest = new ListObjectsRequest(bucketName)
        {
            Prefix = "fun/",
            Delimiter = "/"
        };
        result = client.ListObjects(listObjectsRequest);
        Console.WriteLine("List object succeeded");
        Console.WriteLine("File:");
        foreach (var summary in result.ObjectSummaries)
        {
            Console.WriteLine("Name:{0}", summary.Key);
        }
        Console.WriteLine("Dir:");
        foreach (var prefix in result.CommonPrefixes)
        {
            Console.WriteLine("Name:{0}", prefix);
        }
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine("List object failed. {0}", ex.Message);
    }
}
```

输出：

```
List object success
File:
Name:fun/test.jpg
Dir:
Name:fun/movie/
```

说明：

- 返回的结果中， ObjectSummaries 的列表中给出的是fun目录下的文件。
- 而 CommonPrefixs 的列表中给出的是fun目录下的所有子文件夹。可以看出 fun/movie/001.avi ， fun/movie/007.avi 两个文件并没有被列出来，因为它们属于fun文件夹下的movie目录。

删除文件

删除一个文件

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 列出指定存储空间下的特定文件
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
/// <param name="key">文件的名称</param>
public void DeleteObject(string bucketName, string key)
{
    try
    {
        client.DeleteObject(bucketName, key);
        Console.WriteLine("Delete object succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Delete object failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

删除多个文件

```
using Aliyun.OSS;
```

```
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 删除指定存储空间中的所有文件
/// </summary>
/// <param name="bucketName">存储空间的名称</param>
public void DeleteObjects(string bucketName)
{
    try
    {
        var keys = new List<string>();
        var listResult = client.ListObjects(bucketName);
        foreach (var summary in listResult.ObjectSummaries)
        {
            keys.Add(summary.Key);
        }
        var request = new DeleteObjectsRequest(bucketName, keys, false);
        client.DeleteObjects(request);
        Console.WriteLine("Delete objects succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Delete objects failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

拷贝文件

在同一个区域（杭州，深圳，青岛等）中，用户可以对有操作权限的文件进行复制操作。

拷贝一个文件

通过 copyObject 方法我们可以拷贝一个文件，代码如下：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 拷贝文件
/// </summary>
/// <param name="sourceBucket">原文件所在存储空间的名称</param>
/// <param name="sourceKey">原文件的名称</param>
/// <param name="targetBucket">目标文件所在存储空间的名称</param>
/// <param name="targetKey">目标文件的名称</param>
public void CopyObect(string sourceBucket, string sourceKey, string targetBucket, string
targetKey)
{
    try
    {
        var metadata = new ObjectMetadata();
        metadata.AddHeader(Util.HttpHeaders.ContentType, "text/html");
        var req = new CopyObjectRequest(sourceBucket, sourceKey, targetBucket, targetKey)
        {
            NewObjectMetadata = metadata
```

```
        };
        var ret = client.CopyObject(req);
        Console.WriteLine("Copy object succeeded");
        Console.WriteLine("文件的ETag:{0}", ret.ETag);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Copy object failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- 使用该方法拷贝的文件必须小于1G，否则会报错。若文件大于1G，使用下面的Upload Part Copy。

拷贝大文件

初始化Multipart Upload

我们使用 initiateMultipartUpload 方法来初始化一个分片上传事件：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public void InitiateMultipartUpload(string bucketName, string key)
{
    try
    {
        // 开始Multipart Upload
        var request = new InitiateMultipartUploadRequest(bucketName, key);
        var result = client.InitiateMultipartUpload(request);
        // 打印UploadId
        Console.WriteLine("Init multipart upload succeeded");
        Console.WriteLine("Upload Id:{0}", result.UploadId);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Init multipart upload failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

Upload Part Copy拷贝上传

Upload Part Copy 通过从一个已经存在文件的中拷贝数据来上传一个新文件。当拷贝一个大于500MB的文件，建议使用Upload Part Copy的方式来进行拷贝。

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public void UploadPartCopy(string sourceBucket, string sourceKey, string targetBucket, string
targetKey, string uploadId)
{
    try
    {
        // 计算总共的分片个数
        var metadata = client.GetObjectMetadata(sourceBucket, sourceKey);
        var fileSize = metadata.ContentLength;
        var partCount = (int)fileSize / partSize;
        if (fileSize % partSize != 0)
        {
            partCount++;
        }
        // 开始分片拷贝
        var partETags = new List<PartETag>();
        for (var i = 0; i < partCount; i++)
        {
            var skipBytes = (long)partSize * i;
            var size = (partSize < fileSize - skipBytes) ? partSize : (fileSize - skipBytes);
            var request = new UploadPartCopyRequest(targetBucket, targetKey, sourceBucket,
sourceKey, uploadId)
            {
                PartSize = size,
                PartNumber = i + 1,
                BeginIndex = skipBytes
            };
            var result = client.UploadPartCopy(request);
            partETags.Add(result.PartETag);
        }
        Console.WriteLine("Upload part copy succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Upload part copy failed. {0}", ex.Message);
    }
```

说明：

- 完整代码参考：*GitHub*。
- 以上程序调用uploadPartCopy方法来拷贝每一个分片。
- 与UploadPart要求基本一致，需要通过BeginIndex来定位到此次上传片开头所对应的位置，同时需要通过SourceKey来指定拷贝的文件。

完成分片上传

完成分片上传代码如下：

```
using Aliyun.OSS;
```

```
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public void CompleteMultipartUpload(string bucketName, string key, string uploadId)
{
    try
    {
        var completeMultipartUploadRequest = new CompleteMultipartUploadRequest(
bucketName, key, uploadId);
        foreach (var partETag in partETags)
        {
            completeMultipartUploadRequest.PartETags.Add(partETag);
        }
        var result = client.CompleteMultipartUpload(completeMultipartUploadRequest);
        Console.WriteLine("Complete multipart upload succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Complete multipart upload failed. {0}", ex.Message);
    }
}
```

说明：

- 完整代码参考：*GitHub*。
- 上面代码中的 partETags 就是进行分片上传中保存的partETag的列表，OSS收到用户提交的Part列表后，会逐一验证每个数据Part的有效性。
- 当所有的数据Part验证通过后，OSS会将这些part组合成一个完整的文件。

通过断点续传拷贝

除了支持分片拷贝外，还提供了断点续传功能，如果某次拷贝中断，下次可以从上次失败的位置开始拷贝，以便加快速度。

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
public static void ResumableCopyObject(string sourceBucketName, string sourceKey,
                                       string destBucketName, string destKey)
{
    string checkpointDir = @"<your checkpoint dir>";
    try
    {
        var request = new CopyObjectRequest(sourceBucketName, sourceKey, destBucketName
, destKey);
        client.ResumableCopyObject(request, checkpointDir);
        Console.WriteLine("Resumable copy new object:{0} succeeded", request.DestinationKey);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Resumable copy new object failed, {0}", ex.Message);
    }
```

```
}
```

说明：

- 完整代码参考：*GitHub*。
- checkpointDir目录中会保存断点续传的中间状态，用于失败后，下次继续拷贝时使用。
- 如果checkpointDir为null，断点续传功能不会生效，每次都会重新拷贝。

修改文件Meta

可以通过ModifyObjectMeta操作来实现修改已有文件的 meta 信息。

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 修改文件的meta值
/// </summary>
/// <param name="bucketName">文件所在存储空间的名称</param>
/// <param name="key">文件的名称</param>
public void ModifyObjectMeta(string bucketName, string key)
{
    try
    {
        var meta = new ObjectMetadata();
        meta.ContentType = "application/octet-stream";
        client.ModifyObjectMeta(bucketName, key, meta);
        Console.WriteLine("Modify object meta succeeded");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Modify object meta failed. {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

设置文件权限

您可以通过 SetBucketAcl 设置文件权限。

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 设置文件权限
/// </summary>
/// <param name="bucketName">文件所在存储空间的名称</param>
/// <param name="key">文件的名称</param>
public void SetObjectAclSample(string bucketName, string key)
{
```

```
    try
    {
        client.SetObjectAcl(bucketName, key, CannedAccessControlList.PublicRead);
        Console.WriteLine("Set Object:{0} Acl succeeded ", key);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed with error info: {0}", ex.Message);
    }
}
```

说明：

- 文件的四种访问权限：private、public-read、public-read-write、default分别对应于CannedAccessControlList.Private、CannedAccessControlList.PublicRead、CannedAccessControlList.PublicReadWrite、CannedAccessControlList.Default。
- 完整代码参考：*GitHub*。

读取文件权限

您可以通过 GetBucketAcl 读取文件权限。

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
/// <summary>
/// 读取文件权限
/// </summary>
/// <param name="bucketName">文件所在存储空间的名称</param>
/// <param name="key">文件的名称</param>
public void SetObjectAclSample(string bucketName, string key)
{
    try
    {
        var result = client.GetObjectAcl(bucketName, key);
        Console.WriteLine("Get Object Acl succeeded,Id:{0} Acl:{1} succeeded",
                          result.Owner.Id,  result.ACL.ToString());
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed with error info: {0}", ex.Message);
    }
}
```

说明：

完整代码参考：*GitHub*。

1.2.5.9 授权访问

使用URL签名授权访问

生成签名URL

通过生成签名URL的形式提供给用户一个临时的访问URL。在生成URL时，您可以指定URL过期的时间，从而限制用户长时间访问。

生成一个签名的URL

代码如下：

```
var req = new GeneratePresignedUriRequest(bucketName, key, SignHttpMethod.Get)
{
    Expiration = DateTime.Now.AddHours(1)
};
var uri = client.GeneratePresignedUri(req);
```

说明：

生成的URL默认以GET方式访问，这样，用户可以直接通过浏览器访问相关内容。

生成其他Http方法的URL

如果您想允许用户临时进行其他操作（比如上传，删除文件），可能需要签名其他方法的URL，如下所示：

```
// 生成PUT方法的URL
var req = new GeneratePresignedUriRequest(bucketName, key, SignHttpMethod.Put)
{
    Expiration = DateTime.Now.AddHours(1),
    ContentType = "text/html"
};
var uri = client.GeneratePresignedUri(req);
```

使用签名URL发送请求

现在.Net SDK支持Put Object和Get Object两种方式的URL签名请求。

使用URL签名的方式Put Object

```
var generatePresignedUriRequest = new GeneratePresignedUriRequest(bucketName, key,
SignHttpMethod.Put);
var signedUrl = client.GeneratePresignedUri(generatePresignedUriRequest);
var result = client.PutObject(signedUrl, fileToUpload);
```

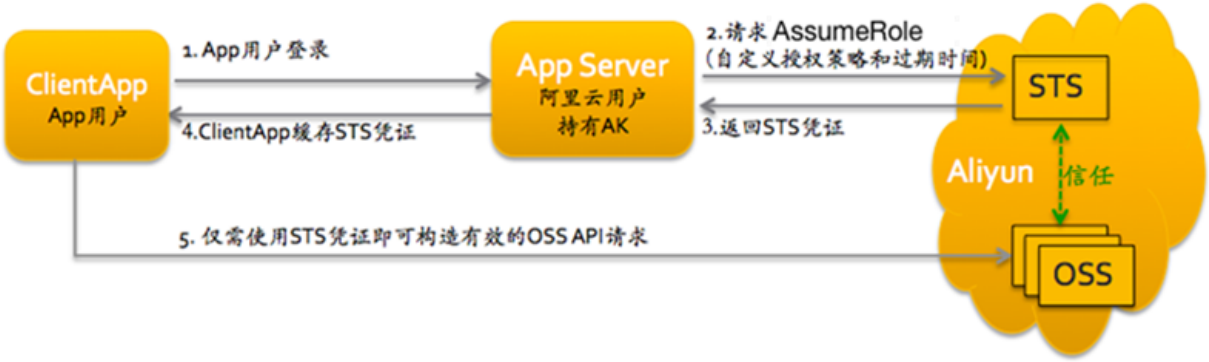
使用STS服务临时授权

介绍

OSS可以通过阿里云STS服务，临时进行授权访问。阿里云STS (Security Token Service) 是为云计算用户提供临时访问令牌的Web服务。通过STS，您可以为第三方应用或联邦用户（用户身份由您自己管理）颁发一个自定义时效和权限的访问凭证。第三方应用或联邦用户可以使用该访问凭证直接调用阿里云产品API，或者使用阿里云产品提供的SDK来访问云产品API。

- 您不需要透露您的长期密钥(AccessKey)给第三方应用，只需要生成一个访问令牌并将令牌交给第三方应用即可。这个令牌的访问权限及有效期限都可以由您自定义。
- 您不需要关心权限撤销问题，访问令牌过期后就自动失效。

以APP应用为例，交互流程如下图所示：



方案的详细描述如下：

1. App用户登录。App用户身份是客户自己管理。客户可以自定义身份管理系统，也可以使用外部Web账号或OpenID。对于每个有效的App用户来说，AppServer是可以确切地定义出每个App用户的最小访问权限。
2. AppServer请求STS服务获取一个安全令牌(SecurityToken)。在调用STS之前，AppServer需要确定App用户的最小访问权限（用Policy语法描述）以及授权的过期时间。然后通过调用STS的AssumeRole(扮演角色)接口来获取安全令牌。角色管理与使用相关内容请参考《*RAM*使用指南》中的角色管理。
3. STS返回给AppServer一个有效的访问凭证，包括一个安全令牌(SecurityToken)、临时访问密钥(AccessKeyId, AccessKeySecret)以及过期时间。
4. AppServer将访问凭证返回给ClientApp。ClientApp可以缓存这个凭证。当凭证失效时，ClientApp需要向AppServer申请新的有效访问凭证。比如，访问凭证有效期为1小时，那么ClientApp可以每30分钟向AppServer请求更新访问凭证。
5. ClientApp使用本地缓存的访问凭证去请求Aliyun Service API。云服务会感知STS访问凭证，并会依赖STS服务来验证访问凭证，并正确响应用户请求。

STS安全令牌详情，请参考《*RAM*使用指南》中的角色管理。关键是调用STS服务接口*AssumeRole*来获取有效访问凭证即可。也可以直接使用STS SDK来调用该方法，点击查看使用STS凭证构造签名请求

用户的client端拿到STS临时凭证后，通过其中安全令牌(SecurityToken)以及临时访问密钥(AccessKeyId, AccessKeySecret)生成OssClient。以上传文件为例：

```
string accessKeyId = "<accessKeyId>";
string accessKeySecret = "<accessKeySecret>";
string securityToken = "<securityToken>"
// 以杭州为例
string endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
var ossClient = new OssClient(endpoint, accessKeyId, accessKeySecret, securityToken);
```

1.2.5.10 生命周期管理

OSS提供文件生命周期管理来为用户管理对象。用户可以为某个存储空间定义生命周期配置，来为该存储空间的文件定义各种规则。 目前，用户可以通过规则来删除相匹配的文件。每条规则都由如下几个部分组成：

- 文件名称前缀，只有匹配该前缀的文件才适用这个规则
- 操作，用户希望对匹配的文件所执行的操作。
- 日期或天数，用户期望在特定日期或者是在文件最后修改时间后多少天执行指定的操作。

设置生命周期

生命周期的配置规则由一段xml表示。

```
<LifecycleConfiguration>
  <Rule>
    <ID>delete obsoleted files</ID>
    <Prefix>obsoleted/</Prefix>
    <Status>Enabled</Status>
    <Expiration>
      <Days>3</Days>
    </Expiration>
  </Rule>
  <Rule>
    <ID>delete temporary files</ID>
    <Prefix>temporary/</Prefix>
    <Status>Enabled</Status>
    <Expiration>
      <Date>2022-10-12T00:00:00.000Z</Date>
    </Expiration>
  </Rule>
</LifecycleConfiguration>
```

一个生命周期的Config里面可以包含多个Rule（最多1000个）。

各字段解释：

- ID字段是用来唯一表示本条规则。
- Prefix指定对存储空间下的符合特定前缀的文件使用规则，不能重叠。
- Status指定本条规则的状态，只有Enabled和Disabled，分别表示启用规则和禁用规则。
- Expiration节点里面的Days表示大于文件最后修改时间指定的天数就删除文件，Date则表示到指定的绝对时间之后就删除文件（绝对时间服从ISO8601的格式）。

可以通过下面的代码，设置上述生命周期规则。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

var setBucketLifecycleRequest = new SetBucketLifecycleRequest(bucketName);

// 创建第一条规则
LifecycleRule lcr1 = new LifecycleRule()
{
    ID = "delete obsoleted files",
    Prefix = "obsoleted/",
    Status = RuleStatus.Enabled,
    ExpriationDays = 3
};

//创建第二条规则
//当ExpirationTime指定使用Date时，请注意此时的含义是：从这个时刻开始一直生效
LifecycleRule lcr2 = new LifecycleRule()
{
     ID = "delete temporary files",
     Prefix = "temporary/",
     Status = RuleStatus.Enabled,
     ExpirationTime = DateTime.Parse("2022-10-12T00:00:00.000Z")
};
setBucketLifecycleRequest.AddLifecycleRule(lcr1);
setBucketLifecycleRequest.AddLifecycleRule(lcr2);

client.SetBucketLifecycle(setBucketLifecycleRequest);
```

说明：

- 完整代码参考：*GitHub*。
- 上面的规则一中ExpriationDays使用了Days，表示3天之后一直生效。
- 上面的规则二中ExpriationDays使用了Date，表示2022-10-12T00:00:00.000Z之后一直生效。除非明确清楚使用Date时的含义，否则请慎重使用。

获取生命周期

可以通过下面的代码获取上述生命周期规则。

```
using Aliyun.OSS;
```

```
var rules = client.GetBucketLifecycle(bucketName);
foreach (var rule in rules)
{
    Console.WriteLine("ID: {0}", rule.ID);
    Console.WriteLine("Prefix: {0}", rule.Prefix);
    Console.WriteLine("Status: {0}", rule.Status);

    if (rule.ExpriationDays.HasValue)
        Console.WriteLine("ExpirationDays: {0}", rule.ExpriationDays);
    if (rule.ExpirationTime.HasValue)
        Console.WriteLine("ExpirationTime: {0}", FormatIso8601Date(rule.ExpirationTime.Value
));
}
```

说明：

完整代码参考：*GitHub*。

清空生命周期

可以通过下面的代码清空存储空间中生命周期规则。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

var LifecycleRequest = new SetBucketLifecycleRequest(bucketName);
client.SetBucketLifecycle(LifecycleRequest);
```

1.2.5.11 设置访问日志

访问日志简介

用户可以通过设置Bucket的访问日志配置，把对该Bucket的访问日志保存在指定的Bucket中，以供后续的分析。访问日志以文件的形式存在于指定的Bucket中，每小时会生成一个文本文件。文件名的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中 TargetPrefix 由用户在配置中指定。

日志配置由如下部分组成：

- TargetBucket：目标Bucket名，生成的日志文件会保存到这个Bucket中。
- TargetPrefix：日志文件名前缀，可以为空。

开启日志功能

下面的代码开启日志功能，且把日志保存在 TargetBucket 中，日志文件名前缀为 logging-：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
// 开启访问日志
var request = new SetBucketLoggingRequest(bucketName, targetBucketName, "logging-");
client.SetBucketLogging(request);
```

说明：

- **bucketName** 和 **targetBucketName** 可以为相同Bucket。
- 完整代码请参考：*GitHub*。

查看日志设置

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
var result = client.GetBucketLogging(bucketName);
Console.WriteLine("Get bucket:{0} Logging, prefix:{1}, target bucket:{2}",
                  bucketName, result.TargetPrefix, result.TargetBucket);
```

说明：

完整代码请参考：*GitHub*。

关闭日志功能

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
client.DeleteBucketLogging(bucketName);
```

说明：

- 日志功能关闭后，已经生成的日志文件不会删除。
- 完整代码请参考：*GitHub*。

1.2.5.12 跨域资源共享

跨域资源共享(CORS)允许web端的应用程序访问不属于本域的资源。OSS提供接口方便开发者控制跨域访问的权限。

设定CORS规则

通过setBucketCors 方法将指定的存储空间上设定一个跨域资源共享CORS的规则，如果原规则存在则覆盖原规则。具体的规则主要通过CORSRule类来进行参数设置。代码如下：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
var req = new SetBucketCorsRequest(bucketName);
var r1 = new CORSRule();
//指定允许跨域请求的来源
r1.AddAllowedOrigin("http://www.a.com");
//指定允许的跨域请求方法(GET/PUT/DELETE/POST/HEAD)
r1.AddAllowedMethod("POST");
//控制在OPTIONS预取指令中Access-Control-Request-Headers头中指定的header是否允许。
r1.AddAllowedHeader("*");
//指定允许用户从应用程序中访问的响应头
r1.AddExposeHeader("x-oss-test");
req.AddCORSRule(r1);
client.SetBucketCors(req);
```

说明：

完整代码参考：*GitHub*。

说明：

- 每个存储空间最多只能使用10条规则。
- AllowedOrigins和AllowedMethods都能够最多支持一个 * 通配符。* 表示对于所有的域来源或者操作都满足。
- 而AllowedHeaders和ExposeHeaders不支持通配符。

获取CORS规则

我们可以参考存储空间的CORS规则，通过GetBucketCors方法。代码如下：

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
var rules = client.GetBucketCors(bucketName);
foreach (var rule in rules)
{
    Console.WriteLine("AllowedOrigins:{0}", rule.AllowedOrigins);
    Console.WriteLine("AllowedMethods:{0}", rule.AllowedMethods);
    Console.WriteLine("AllowedHeaders:{0}", rule.AllowedHeaders);
```

```
    Console.WriteLine("ExposeHeaders:{0}", rule.ExposeHeaders);
    Console.WriteLine("MaxAgeSeconds:{0}", rule.MaxAgeSeconds);
}
```

说明：

完整代码参考：*GitHub*。

删除CORS规则

用于关闭指定存储空间对应的CORS并清空所有规则。

```
using Aliyun.OSS;
// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
// 清空存储空间的CORS规则
client.DeleteBucketCors(bucketName);
```

说明：

完整代码参考：*GitHub*。

1.2.5.13 防盗链

OSS是按使用收费的服务，为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

设置Referer白名单

通过下面代码设置Referer白名单：

```
using Aliyun.OSS;
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
var refererList = new List<string>();
// 添加referer项
refererList.Add("http://www.aliyun.com");
refererList.Add("http://www.*.com");
refererList.Add("http://www.?.aliyuncs.com");
// 允许referer字段为空，并设置存储空间Referer列表
var request = new SetBucketRefererRequest(bucketName, refererList);
request. AllowEmptyReferer = true;
client.setBucketReferer(bucketName, br);
Console.WriteLine("设置存储空间{0}的referer白名单成功", bucketName);
```

说明：

完整代码参考：*GitHub*。

说明：

Referer参数支持通配符 * 和 ？。

获取Referer白名单

```
using Aliyun.OSS;
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
var rc = client.GetBucketReferer(bucketName);
Console.WriteLine("allow ? " + (rc.AllowEmptyReferer ? "yes" : "no"));
if (rc.RefererList.Referers != null)
{
    for (var i = 0; i < rc.RefererList.Referers.Length; i++)
        Console.WriteLine(rc.RefererList.Referers[i]);
}
else
{
    Console.WriteLine("Empty Referer List");
}
```

说明：

完整代码参考：*GitHub*。

清空Referer白名单

Referer白名单不能直接清空，只能通过重新设置来覆盖之前的规则。

```
using Aliyun.OSS;
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);
// 默认允许referer字段为空，且referer白名单为空。
var request = new SetBucketRefererRequest(bucketName);
client.SetBucketReferer(request);
Console.WriteLine("清空存储空间{0}的referer白名单成功", bucketName);
```

1.2.5.14 图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处理服务。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

图片处理基础功能

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义

- 通过级联处理调用多个图片处理功能

图片处理使用

图片处理使用标准的 HTTP GET 请求来访问，所有的处理参数是编码在 URL 中的QueyString。

匿名访问

如果图片文件（Object）的访问权限是 公共读，如下表所示的权限，则可以匿名访问图片服务。

Object权限	Object权限
公共读私有写（public-read）或 公共读写（public-read-write）	默认（default）
任意权限	公共读私有写（public-read）或 公共读写（public-read-write）

通过如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value
```

- bucket：用户的存储空间（bucket）名称
- endpoint：用户存储空间所在数据中心的访问域名
- object：用户上传在OSS上的图片文件
- image：图片处理保留标志符
- action：用户对图片做的操作，如缩放、裁剪、旋转等
- parame：用户对图片做的操作所对应的参数

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,
w_100
```

自定义样式，使用如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=x-oss-process=style/name
```

- style：用户自定义样式系统保留标志符
- name：自定义样式名称，即控制台定义样式的 规则名

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=style/oss-pic-style-w-100
```

通过级联处理，可以对一张图片顺序实施多个操作，格式如下：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value/action,parame_value/...
```

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100/rotate,90
```

图片服务也支持HTTPS访问，例如：

```
https://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100
```

授权访问

对私有权限的文件（Object），如下表所示的权限，必须通过授权才能访问图片服务。

Bucket权限	Object权限
私有读写（private）	默认权限（default）
任意权限	私有读写（private）

生成带签名的图片处理的URL代码如下：

```
public static void GenerateIamgeUri(string bucketName)
{
    try
    {
        var process = "image/resize,m_fixed,w_100,h_100";
        var req = new GeneratePresignedUriRequest(bucketName, key, SignHttpMethod.Get)
        {
            Expiration = DateTime.Now.AddSeconds(30 * 60),
            Process = process
        };
        // 产生带有签名的URI
        var uri = client.GeneratePresignedUri(req);
        Console.WriteLine("Generate Presigned Uri:{0} with process:{1} succeeded ", uri, process
);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed with error info: {0}", ex.Message);
    }
```

```
}
```

说明：

- 授权访问支持 **自定义样式**、**HTTPS**、**级联处理**
- **GeneratePresignedUri** 过期时间单位是**秒**
- 完整的代码请参看 *GitHub*

SDK访问

对于任意权限的图片文件，都可以直接使用 SDK 访问图片、进行处理。

说明：

- 图片处理的完整代码请参考：*GitHub*
- SDK处理图片文件支持**自定义样式**、**HTTPS**、**级联处理**

基础操作

图片处理的基础操作包括，获取图片信息、格式转换、缩放、裁剪、旋转、效果、水印等。

```
public static void ImageProcess(string bucketName)
{
    try
    {
        client.PutObject(bucketName, key, imageFile);
        // 图片缩放
        var process = "image/resize,m_fixed,w_100,h_100";
        var ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-resize.jpg", ossObject.Content);
        // 图片裁剪
        process = "image/crop,w_100,h_100,x_100,y_100,r_1";
        ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-crop.jpg", ossObject.Content);
        // 图片旋转
        process = "image/rotate,90";
        ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-rotate.jpg", ossObject.Content);
        // 图片锐化
        process = "image/sharpen,100";
        ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-sharpen.jpg", ossObject.Content);
        // 图片加水印
        process = "image/watermark,text_SGVsbG8g5Zu-54mH5pyN5YqhIQ";
        ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-watermark.jpg", ossObject.Content);
        // 图片格式转换
        process = "image/format,png";
        ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-format.png", ossObject.Content);
        // 图片信息
        process = "image/info";
```

```
        ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-info.txt", ossObject.Content);
        Console.WriteLine("Get Object:{0} with process:{1} succeeded ", key, process);
    }
    catch (OssException ex)
    {
        Console.WriteLine("Failed with error code: {0}; Error info: {1}. \nRequestID:{2}\tHostID:{3
}",
            ex.ErrorCode, ex.Message, ex.RequestId, ex.HostId);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed with error info: {0}", ex.Message);
    }
}
```

自定义样式

```
public static void ImageProcess(string bucketName)
{
    try
    {
        client.PutObject(bucketName, key, imageFile);
        // 自定义样式
        var process = "style/oss-pic-style-w-100";
        var ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-style.jpg", ossObject.Content);
    }
    catch (OssException ex)
    {
        Console.WriteLine("Failed with error code: {0}; Error info: {1}. \nRequestID:{2}\tHostID:{3
}",
            ex.ErrorCode, ex.Message, ex.RequestId, ex.HostId);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed with error info: {0}", ex.Message);
    }
}
```

级联处理

```
public static void ImageProcess(string bucketName)
{
    try
    {
        client.PutObject(bucketName, key, imageFile);
        // 级联处理
        var process = "image/resize,m_fixed,w_100,h_100/rotate,90";
        var ossObject = client.GetObject(new GetObjectRequest(bucketName, key, process));
        WriteToFile(dirToDownload + "/sample-style.jpg", ossObject.Content);
    }
    catch (OssException ex)
    {
        Console.WriteLine("Failed with error code: {0}; Error info: {1}. \nRequestID:{2}\tHostID:{3
}",
            ex.ErrorCode, ex.Message, ex.RequestId, ex.HostId);
    }
    catch (Exception ex)
    {
```

```
        Console.WriteLine("Failed with error info: {0}", ex.Message);
    }
}
```

图片处理工具

- 可视化图片处理工具 *ImageStyleViever* ，可以直观的看到OSS图片处理的结果
- OSS图片处理的功能、使用演示 页面

1.2.5.15 异常

OSS .NET SDK 中有两种异常 ClientException 以及 OSSException ，他们都继承自或者间接继承自 RuntimeException。

ClientException

ClientException指SDK内部出现的异常，比如未设置BucketName，网络无法到达等等。

OSSException

OSSException指服务器端错误，它来自于对服务器错误信息的解析。OSSException一般有以下几个成员：

- Code：OSS返回给用户的错误码。
- Message：OSS给出的详细错误信息。
- RequestId：用于唯一标识该次请求的UUID；当您无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助。
- HostId：用于标识访问的OSS集群（目前统一为oss.aliyuncs.com）

下面是OSS中常见的异常：

错误码	描述
AccessDenied	拒绝访问
BucketAlreadyExists	Bucket已经存在
BucketNotEmpty	Bucket不为空
EntityTooLarge	实体过大
EntityTooSmall	实体过小
FileGroupTooLarge	文件组过大
FilePartNotExist	文件Part不存在
FilePartStale	文件Part过时

错误码	描述
InvalidArgument	参数格式错误
InvalidAccessKeyId	AccessKeyId不存在
InvalidBucketName	无效的Bucket名字
InvalidDigest	无效的摘要
InvalidObjectName	无效的Object名字
InvalidPart	无效的Part
InvalidPartOrder	无效的part顺序
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket
InternalError	OSS内部发生错误
MalformedXML	XML格式非法
MethodNotAllowed	不支持的方法
MissingArgument	缺少参数
MissingContentLength	缺少内容长度
NoSuchBucket	Bucket不存在
NoSuchKey	文件不存在
NoSuchUpload	Multipart Upload ID不存在
NotImplemented	无法处理的方法
PreconditionFailed	预处理错误
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟
RequestTimeout	请求超时
SignatureDoesNotMatch	签名错误
TooManyBuckets	用户的Bucket数目超过限制

1.2.6 JavaScript-SDK

1.2.6.1 安装

- github地址：*https://github.com/ali-sdk/ali-oss*
- API文档：*https://github.com/ali-sdk/ali-oss#summary*
- ChangeLog：*https://github.com/ali-sdk/ali-oss/blob/master/History.md*

要求

- 开通阿里云OSS服务，并创建了AccessKeyId 和AccessKeySecret。
- 如果您还没有开通或者还不了解阿里云OSS服务，请登录*OSS产品主页*了解。
- 如果还没有创建AccessKeyId和AccessKeySecret，请到*阿里云Access Key管理*创建Access Key。

环境要求

OSS JavaScript SDK基于*Node.js*环境构建，通过*Browserify*和*Babel*产生适用于浏览器的代码。用户在浏览器中和Node.js环境中都可以使用OSS JavaScript SDK。

但是由于浏览器环境的特殊性，有一些功能无法使用：

- 流式上传：浏览器中无法设置chunked编码，用分片上传替代
- 操作本地文件：浏览器中不能直接操作本地文件系统，用签名URL的方式下载
- Bucket相关的操作（listBuckets/BucketACL/BucketLogging等），由于OSS暂时不支持Bucket相关的跨域请求，Bucket相关的操作可以在控制台进行

使用方式

OSS JavaScript SDK同时支持同步和异步的使用方式，参考这篇文章：

- 同步方式：类似协程的方式，基于*Generator Function*，使用 `co` 和 `yield`
- 异步方式：类似callback的方式，API接口返回*Promise*，使用 `.then()` 处理返回结果，使用 `.catch()` 处理错误

在同步方式中，使用 `new OSS()` 创建client，在异步方式中，使用 `new OSS.Wrapper()` 创建client。

下面分别举例，先上传一个文件，然后立即下载这个文件。

同步方式

```
// var client = new OSS(...);

var co = require('co');
co(function* () {
  var r1 = client.put('object', '/tmp/file');
  console.log('put success: %j', r1);
  var r2 = client.get('object');
  console.log('get success: %j', r2);
}).catch(function (err) {
  console.error('error: %j', err);
```

```
});
```

异步方式

```
// var client = new OSS.Wrapper(...);

client.put('object', '/tmp/file').then(function (r1) {
  console.log('put success: %j', r1);
  return client.get('object');
}).then(function (r2) {
  console.log('get success: %j', r2);
}).catch(function (err) {
  console.error('error: %j', err);
});
```

安装

在浏览器中使用

支持的浏览器：

- IE(>=10)和Edge
- 主流版本的Chrome/Firefox/Safari
- 主流版本的Android/iOS/WindowsPhone

在浏览器中使用只需要在网页标签中包含如下 `<script>` 标签：

```
<script src="http://gosspublic.alicdn.com/aliyun-oss-sdk-4.3.0.min.js"></script>
```

就可以在代码中使用 `OSS.Wrapper` 对象：

```
<script type="text/javascript">
  var client = new OSS.Wrapper({
    region: '<oss region>',
    accessKeyId: '<Your accessKeyId(STS)>',
    accessKeySecret: '<Your accessKeySecret(STS)>',
    stsToken: '<Your securityToken(STS)>',
    bucket: '<Your bucket name>'
  });
</script>
```

使用Node.js

支持的Node.js版本：

- 4.x
- 5.x

首先使用*npm*安装SDK的开发包：

```
npm install ali-oss
```

然后在你的程序中使用：

```
var OSS = require('ali-oss');
var client = new OSS({
  region: '<oss region>',
  accessKeyId: '<Your accessKeyId>',
  accessKeySecret: '<Your accessKeySecret>',
  bucket: '<Your bucket name>'
});
```

如果使用npm遇到网络问题，可以使用淘宝提供的npm镜像：*cnpm*

使用Bower

对于Web项目，你也可以通过*Bower*安装SDK：

```
bower install ali-oss
```

然后在网页中添加 `<script>` 标签：

```
<script src="bower_components/ali-oss/dist/aliyun-oss-sdk.min.js"></script>
```

1.2.6.2 快速开始-浏览器

下面介绍如何在浏览器中使用OSS JavaScript SDK来访问OSS服务，包括上传/下载文件和查看文件列表。

> **说明**：
> 为了简化，下面的介绍直接在网页中使用AccessKeyId和AccessKeySecret，这是不安全的做法。实际使用中应使用STS进行临时授权访问。

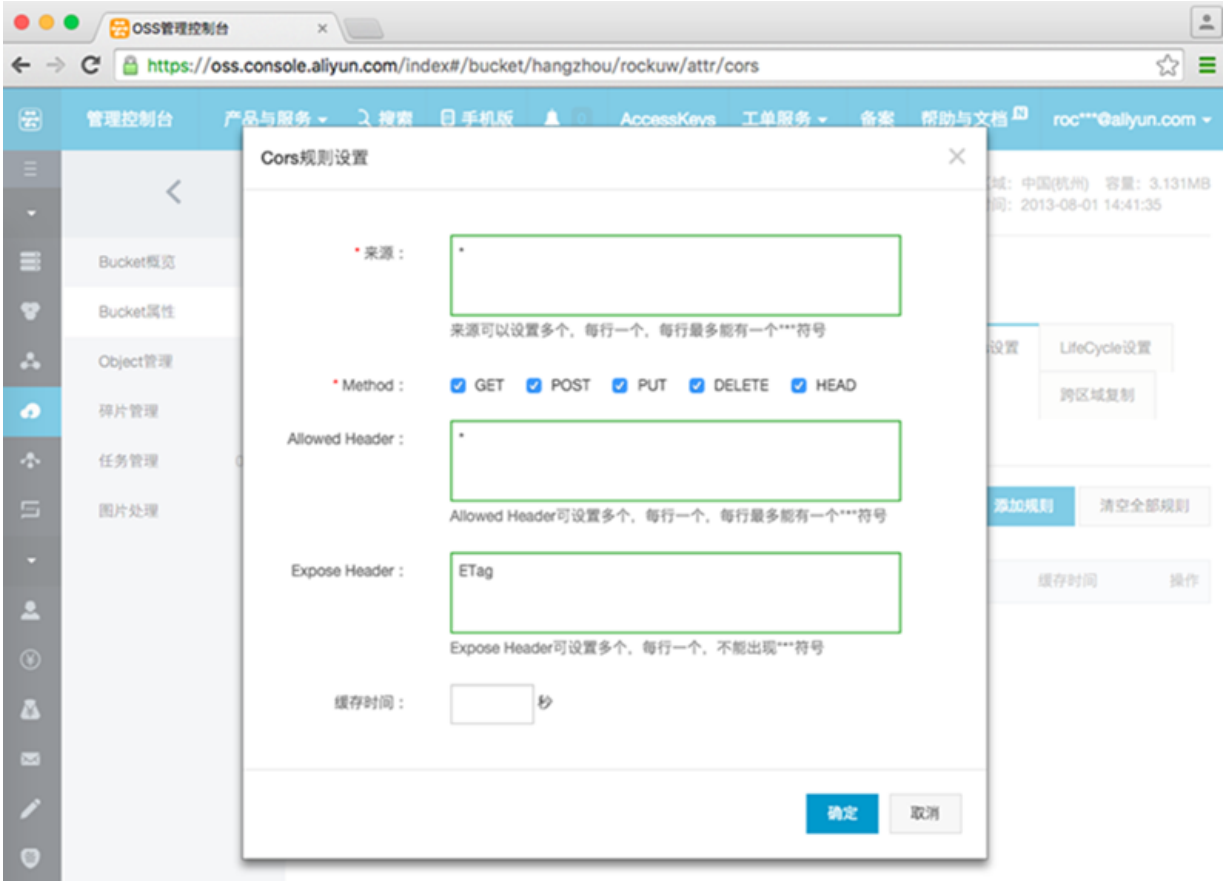
Bucket设置

从浏览器中直接访问OSS需要开通Bucket的CORS设置：

- 将allowed origins设置成 `*`
- 将allowed methods设置成 `PUT, GET, POST, DELETE, HEAD`
- 将allowed headers设置成 `*`
- 将expose headers设置成 `ETag`

> **说明**：

请将该条CORS规则设置成所有CORS规则的第一条。



使用SDK

目前浏览器中不能直接进行Bucket相关的操作（例如list buckets, get/set bucket logging/referer/website/cors）。但是可以进行Object相关的操作（例如上传/下载文件，查看文件列表等）。

包含SDK

首先在html文件的 <head> 中包含如下标签：

```
<script src="http://gosspublic.alicdn.com/aliyun-oss-sdk-4.4.4.min.js"></script>
```

通过new OSS.Wapper()来创建client，OSS.Wrapper提供了异步的接口，类似于callback的方式，在.then()中处理返回的结果，在.catch()中处理错误。

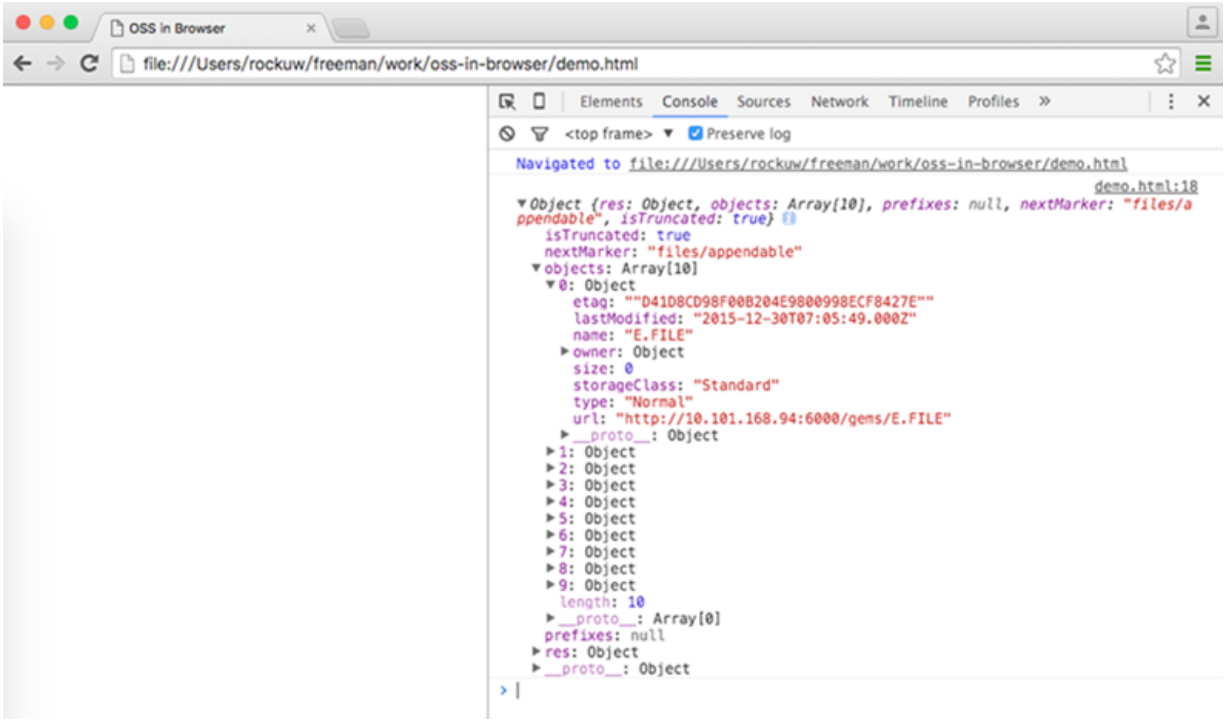
查看文件列表

```
<script type="text/javascript">
  var client = new OSS.Wrapper({
    region: '<oss region>',
    accessKeyId: '<Your accessKeyId>',
    accessKeySecret: '<Your accessKeySecret>',
    bucket: '<Your bucket name>'
  });
  client.list({
```

```
    'max-keys': 10
  }).then(function (result) {
    console.log(result);
  }).catch(function (err) {
    console.log(err);
  });
</script>
```

其中 **region** 参数是指您申请OSS服务时的区域，例如 **oss-cn-hangzhou**。

在浏览器中打开html文件，然后打开Chrome的**开发者控制台**，就可以看到list文件的结果了。

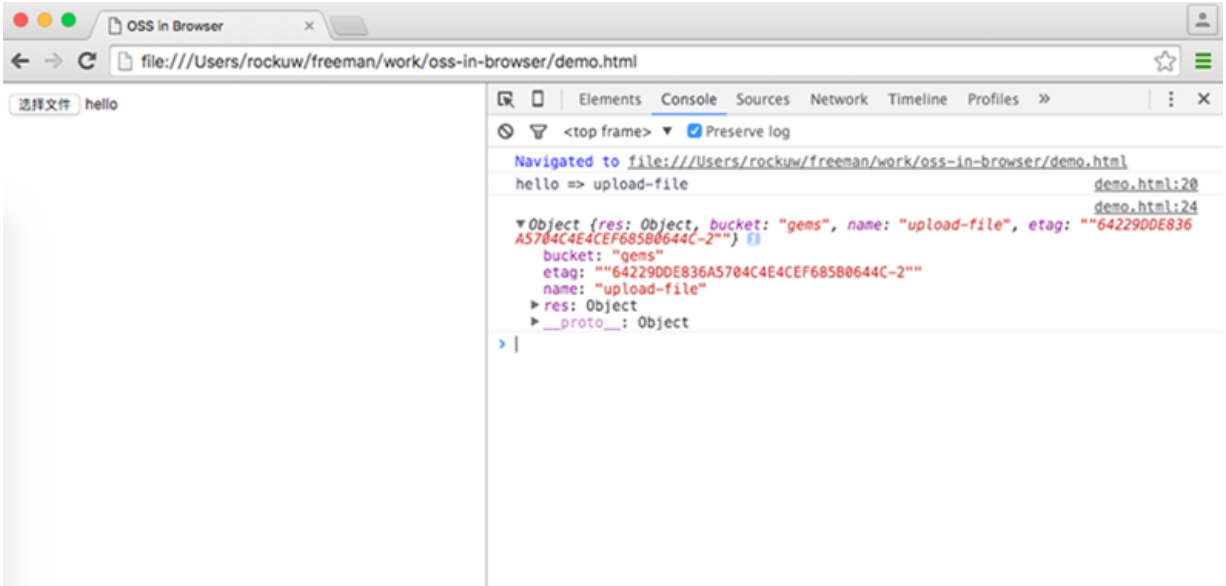


上传文件

下面使用 multipartUpload 接口来上传文件：

```
<body>
  <input type="file" id="file" />
  <script type="text/javascript">
    var client = new OSS.Wrapper({
      region: '<oss region>',
      accessKeyId: '<Your accessKeyId>',
      accessKeySecret: '<Your accessKeySecret>',
      bucket: '<Your bucket name>'
    });
    document.getElementById('file').addEventListener('change', function (e) {
      var file = e.target.files[0];
      var storeAs = 'upload-file';
      console.log(file.name + ' => ' + storeAs);
      client.multipartUpload(storeAs, file).then(function (result) {
        console.log(result);
      }).catch(function (err) {
        console.log(err);
      });
    });
```

```
  </script>
</body>
```

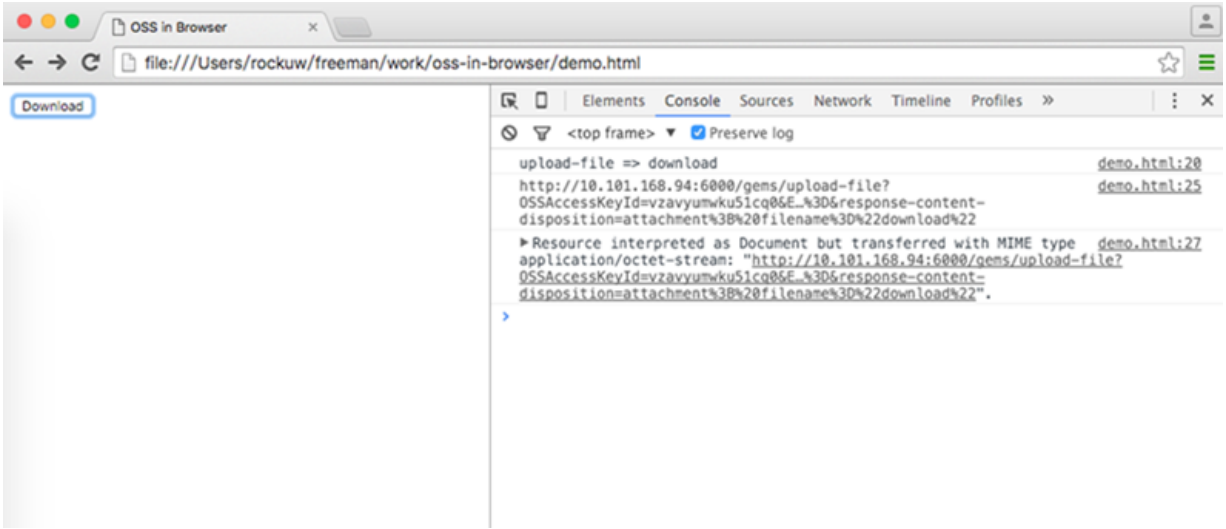


下载文件

在浏览器中不能直接操作文件，因此采用签名URL的方式来生成文件的下载链接，用户只需要点击链接就可以下载文件。

```
<body>
  <input type="button" id="download" value="Download" />
  <script type="text/javascript">
    var client = new OSS.Wrapper({
      region: '<oss region>',
      accessKeyId: '<Your accessKeyId>',
      accessKeySecret: '<Your accessKeySecret>',
      bucket: '<Your bucket name>'
    });
    document.getElementById('download').addEventListener('click', function (e) {
      var objectKey = 'upload-file';
      var saveAs = 'download';
      console.log(objectKey + ' => ' + saveAs);
      var result = client.signatureUrl(objectKey, {
        expires: 3600,
        response: {
          'content-disposition': 'attachment; filename="' + saveAs + '"'
        }
      });
      console.log(result);
      window.location = result;
    });
  </script>
```

```
</body>
```



1.2.6.3 快速开始-NodeJS

下面介绍如何在Node.js环境中使用OSS JavaScript SDK来访问OSS服务，包括查看Bucket列表，查看文件列表，上传/下载文件和删除文件。为了方便修改，下面的介绍会新建一个 app.js，下面的功能演示代码都写在这个文件中。

安装SDK

首先在工作目录安装ali-oss：

```
npm install ali-oss
```

使用同步方式

由于SDK基于*ES6*开发，使用了*Generator Function*使得用户能够方便地用同步的方式异步的代码，需要配合 co 使用。具体可参考这篇博客。使用同步方式还需要安装 co：

```
npm install co
```

使用异步方式

为了支持callback的使用方式，SDK同时也提供了异步的基于*Promise*的接口，使用上类似callback，具体可参考这篇博客。

下面的文档将以**同步的方式**为例。

初始化Client

创建一个文件 *app.js* 并写入下面的内容：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>'
});
```

其中 **region** 参数是指您申请OSS服务时的区域，例如 **oss-cn-hangzhou**。

如果所使用的endpoint不在上述列表中，可以通过以下参数指定endpoint：

- internal：配合 **region** 使用，如果指定 **internal** 为 **true**，则访问内网节点
- secure：配合 **region** 使用，如果指定了 **secure** 为 **true**，则使用HTTPS访问
- endpoint：例如 *http://oss-cn-hangzhou.aliyuncs.com*，如果指定了 **endpoint**，则 **region** 会被忽略，**endpoint** 可以指定HTTPS，也可以是IP形式
- cname：配合 **endpoint** 使用，如果指定了 **cname** 为 **true**，则将 **endpoint** 视为用户绑定的自定义域名
- bucket：如果未指定 **bucket**，则进行Object相关的操作时需要先调用 **useBucket** 接口（只需要调用一次）
- timeout：默认为60秒，指定访问OSS的API的超时时间

查看Bucket列表

在app.js末尾添加如下内容，使用listBuckets接口查看Bucket列表：

```
co(function* () {
  var result = yield client.listBuckets();
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

运行并查看结果：node app.js。

查看文件列表

修改app.js，使用list接口查看文件列表：

```
co(function* () {
  client.useBucket('Your bucket name');
  var result = yield client.list({
    'max-keys': 5
  });
```

```
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

使用node app.js运行并查看结果。

上传一个文件

修改app.js，使用put接口上传一个文件：

```
co(function* () {
  client.useBucket('Your bucket name');
  var result = yield client.put('object-key', 'local file');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

下载一个文件

修改app.js，使用get接口下载一个文件：

```
co(function* () {
  var result = yield client.get('object-key', 'local file');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

删除一个文件

修改app.js，使用delete接口下载一个文件：

```
co(function* () {
  var result = yield client.delete('object-key');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

1.2.6.4 管理Bucket

存储空间（Bucket）是OSS上的命名空间，也是计费、权限控制、日志记录等高级功能的管理实体。

查看所有Bucket

使用listBuckets接口列出当前用户下的所有Bucket，用户还可以指定**prefix**参数，列出Bucket名字为特定前缀的所有Bucket：

```
var co = require('co');
var OSS = require('ali-oss');
```

```
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>'
});
co(function* () {
  var result = yield client.listBuckets();
  console.log(result);
  var result = yield client.listBuckets({
    prefix: 'prefix'
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

创建Bucket

使用putBucket接口创建一个Bucket，用户需要指定Bucket的名字：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>'
});
co(function* () {
  var result = yield client.putBucket('bucket name');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

说明：

- 由于存储空间的名字是全局唯一的，所以必须保证您的Bucket名字不与别人的重复。

删除Bucket

使用deleteBucket接口删除一个Bucket，用户需要指定Bucket的名字：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>'
});
co(function* () {
  var result = yield client.deleteBucket('bucket name');
  console.log(result);
}).catch(function (err) {
  console.log(err);
```

```
});
```

说明：

- 如果该Bucket下还有文件存在，则需要先删除所有文件才能删除Bucket
- 如果该Bucket下还有未完成的上传请求，则需要通过listUploads和abortMultipartUpload先取消那些请求才能删除Bucket。

Bucket访问权限

用户可以设置Bucket的访问权限，允许或者禁止匿名用户对其内容进行读写。更多关于访问权限的内容请参考访问权限。

获取Bucket的访问权限（ACL）

通过getBucketACL查看Bucket的ACL：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>'
});
co(function* () {
  var result = yield client.getBucketACL('bucket name');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

设置Bucket的访问权限（ACL）

通过putBucketACL设置Bucket的ACL：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>'
});
co(function* () {
  var result = yield client.putBucketACL('bucket name', 'region', 'public-read');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

1.2.6.5 上传文件

用户可以通过以下方式向OSS中上传文件：

- 上传本地文件
- 流式上传
- 上传Buffer内容
- 分片上传
- 断点上传

上传本地文件

通过put接口来上传一个本地文件到OSS：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.put('object-key', 'local-file');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

流式上传

通过putStream接口来上传一个Stream中的内容，**stream**参数可以是任何实现了**Readable Stream**的对象，包含文件流，网络流等。当使用putStream接口时，SDK默认会发起一个chunked encoding的HTTP PUT请求。如果在**options**指定了**contentLength**参数，则不会使用chunked encoding。

```
var co = require('co');
var OSS = require('ali-oss');
var fs = require('fs');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  // use 'chunked encoding'
  var stream = fs.createReadStream('local-file');
  var result = yield client.putStream('object-key', stream);
  console.log(result);
  // don't use 'chunked encoding'
  var stream = fs.createReadStream('local-file');
  var size = fs.statSync('local-file').size;
  var result = yield client.putStream(
    'object-key', stream, {contentLength: size});
  console.log(result);
```

```
}).catch(function (err) {
  console.log(err);
});
```

上传Buffer内容

用户也可以通过put接口简单地将Buffer中的内容上传到OSS：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.put('object-key', new Buffer('hello world'));
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

分片上传

在需要上传的文件较大时，可以通过multipartUpload接口进行分片上传。分片上传的好处是将一个大请求分成多个小请求来执行，这样当其中一些请求失败后，不需要重新上传整个文件，而只需要上传失败的分片就可以了。一般对于大于100MB的文件，建议采用分片上传的方法。

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.multipartUpload('object-key', 'local-file', {
    progress: function* (p) {
      console.log('Progress: ' + p);
    }
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

上面的**progress**参数是一个进度回调函数，用于获取上传进度。**progress**可以是一个generator function（**function***），也可以是一个**thunk**：

```
var progress = function (p) {
  return function (done) {
    console.log(p);
    done();
```

```
  };
};
```

断点上传

分片上传提供**progress**参数允许用户传递一个进度回调，在回调中SDK将当前已经上传成功的比例和断点信息作为参数。为了实现断点上传，可以在上传过程中保存断点信息（checkpoint），发生错误后，再将已保存的checkpoint作为参数传递给**multipartUpload**，此时将从上次失败的地方继续上传。

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var checkpoint;
  // retry 5 times
  for (var i = 0; i < 5; i++) {
      var result = yield client.multipartUpload('object-key', 'local-file', {
        checkpoint: checkpoint,
        progress: function* (percentage, cpt) {
          checkpoint = cpt;
        }
      });
      console.log(result);
      break; // break if success
    } catch (err) {
      console.log(err);
    }
  }
}).catch(function (err) {
  console.log(err);
});
```

上面的代码只是将checkpoint保存在变量中，如果程序崩溃的话就丢失了，用户也可以将它保存在文件中，然后在程序重启后将checkpoint信息从文件中读取出来。

1.2.6.6 下载文件

用户可以通过以下方式从OSS中下载文件：

- 下载到本地文件
- 流式下载
- 下载到Buffer
- HTTP下载（浏览器下载）

下载到本地文件

通过get接口来下载Object到一个本地文件：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.get('object-key', 'local-file');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

流式下载

使用getStream来下载文件时，返回一个Readable Stream，用户可以流式地处理文件内容。

```
var co = require('co');
var OSS = require('ali-oss');
var fs = require('fs');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.getStream('object-key');
  console.log(result);
  var writeStream = fs.createWriteStream('local-file');
  result.stream.pipe(writeStream);
}).catch(function (err) {
  console.log(err);
});
```

下载Buffer

用户也可以通过get接口简单地将文件内容下载到Buffer中：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.get('object-key');
  console.log(result.content);
}).catch(function (err) {
  console.log(err);
```

```
});
```

HTTP下载

对于存放在OSS中的文件，在不用SDK的情况下用户也可以直接使用HTTP下载，这包括直接使用浏览器下载，或者使用wget、curl等命令行工具下载。这时文件的URL需要由SDK生成。使用signatureUrl方法生成可下载的HTTP地址，URL的有效时间默认为半个小时：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
var url = client.signatureUrl('object-key');
console.log(url);
var url = client.signatureUrl('object-key', {expires: 3600});
console.log(url);
// signed URL for PUT
var url = client.signatureUrl('object-key', {method: 'PUT'});
console.log(url);
```

1.2.6.7 管理文件

一个Bucket下可能有非常多的文件，SDK提供一系列的接口方便用户管理文件。

查看所有文件

通过list来列出当前Bucket下的所有文件。主要的参数如下：

- prefix 指定只列出符合特定前缀的文件
- marker 指定只列出文件名大于marker之后的文件
- delimiter 用于获取文件的公共前缀
- max-keys 用于指定最多返回的文件个数

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  // 不带任何参数，默认最多返回1000个文件
  var result = yield client.list();
  console.log(result);
  // 根据nextMarker继续列出文件
  if (result.isTruncated) {
    var result = yield client.list({
```

```
      marker: result.nextMarker
    });
  }
  // 列出前缀为'my-'的文件
  var result = yield client.list({
    prefix: 'my-'
  });
  console.log(result);
  // 列出前缀为'my-'且在'my-object'之后的文件
  var result = yield client.list({
    prefix: 'my-',
    marker: 'my-object'
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

模拟目录结构

OSS是基于对象的存储服务，没有目录的概念。存储在一个Bucket中所有文件都是通过文件的key唯一标识，并没有层级的结构。这种结构可以让OSS的存储非常高效，但是用户管理文件时希望能够像传统的文件系统一样把文件分门别类放到不同的“目录”下面。通过OSS提供的**公共前缀**的功能，也可以很方便地模拟目录结构。

假设Bucket中已有如下文件：

```
foo/x
foo/y
foo/bar/a
foo/bar/b
foo/hello/C/1
foo/hello/C/2
...
foo/hello/C/9999
```

接下来我们实现一个函数叫listDir，列出指定目录下的文件和子目录：

```
var co = require('co');
var OSS = require('ali-oss');
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
function* listDir(dir)
  var result = yield client.list({
    prefix: dir,
    delimiter: '/'
  });
  result.prefixes.forEach(function (subDir) {
    console.log('SubDir: %s', subDir);
  });
  result.objects.forEach(function (obj) {
    console.log(Object: %s', obj.name);
```

```
  });
end
```

运行结果如下：

```
> yield listDir('foo/')
=> SubDir: foo/bar/
   SubDir: foo/hello/
   Object: foo/x
   Object: foo/y
> yield listDir('foo/bar/')
=> Object: foo/bar/a
   Object: foo/bar/b
> yield listDir('foo/hello/C/')
=> Object: foo/hello/C/1
   Object: foo/hello/C/2
   ...
   Object: foo/hello/C/9999
```

文件元信息

向OSS上传文件时，除了文件内容，还可以指定文件的一些属性信息，称为**元信息**。这些信息在上传时与文件一起存储，在下载时与文件一起返回。

因为文件元信息在上传/下载时是附在HTTP Headers中， HTTP协议规定不能包含复杂字符。因此元信息**只能是简单的ASCII可见字符**，**不能包含换行。** 所有元信息的总大小不能超过8KB。

使用put，putStream和multipartUpload时都可以通过指定**meta**参数来指定文件的元信息：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.put('object-key', 'local-file', {
    meta: {
      year: 2016,
      people: 'mary'
    }
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

通过putMeta接口来更新文件元信息：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
```

```
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.putMeta('object-key', {
    meta: {
      year: 2015,
      people: 'mary'
    }
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

拷贝文件

使用copy拷贝一个文件。拷贝可以发生在下面两种情况：

- 同一个Bucket
- 两个不同Bucket，但是它们在同一个region，此时的源Object名字应为/bucket/object的形式

另外，拷贝时对文件元信息的处理有两种选择：

- 如果没有指定**meta**参数，则与源文件相同，即拷贝源文件的元信息
- 如果指定了**meta**参数，则使用新的元信息覆盖源文件的信息

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  // 两个Bucket之间拷贝
  var result = yield client.copy('to', '/from-bucket/from');
  console.log(result);
  // 拷贝元信息
  var result = yield client.copy('to', 'from');
  console.log(result);
  // 覆盖元信息
  var result = yield client.copy('to', 'from', {
    meta: {
      year: 2015,
      people: 'mary'
    }
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
```

```
});
```

删除文件

通过delete来删除某个文件：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.delete('object-key');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

批量删除文件

通过deleteMulti来删除一批文件，用户可以通过**quiet**参数来指定是否返回删除的结果：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>',
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: 'Your bucket name'
});
co(function* () {
  var result = yield client.deleteMulti(['obj-1', 'obj-2', 'obj-3']);
  console.log(result);
  var result = yield client.deleteMulti(['obj-1', 'obj-2', 'obj-3'], {
    quiet: true
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

1.2.6.8 自定义域名绑定

OSS支持用户将自定义的域名绑定到OSS服务上，这样能够支持用户无缝地将存储迁移到OSS上。例如用户的域名是my-domain.com，之前用户的所有图片资源都是形如http://img.my-domain.com/x.jpg的格式，用户将图片存储迁移到OSS之后，通过绑定自定义域名，仍可以使用原来的地址访问到图片：

- 开通OSS服务并创建Bucket

- 修改域名的DNS配置，增加一个CNAME记录，将img.my-domain.com指向OSS服务的endpoint（如my-bucket.oss-cn-hangzhou.aliyuncs.com）
- 在官网控制台将img.my-domain.com与创建的Bucket绑定
- 将图片上传到OSS的这个Bucket中

这样就可以通过原地址*http://img.my-domain.com/x.jpg*访问到存储在OSS上的图片。

在使用SDK时，也可以使用自定义域名作为endpoint，这时需要将**cname**参数设置为true，如下面的例子：

```
var co = require('co');
var OSS = require('ali-oss')

var client = new OSS({
  endpoint: '<Your endpoint>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  cname: true
});

client.useBucket('my-bucket')
```

说明：

使用CNAME时，无法使用list_buckets接口。（因为自定义域名已经绑定到某个特定的Bucket。）

1.2.6.9 使用STS访问

OSS可以通过阿里云STS服务，临时进行授权访问。

1. 在官网控制台创建子账号。
2. 在官网控制台创建STS角色并赋予子账号扮演角色的权限。
3. 使用子账号的AccessKeyId/AccessKeySecret向STS申请临时token
4. 使用临时token中的认证信息创建OSS的Client
5. 使用OSS的Client访问OSS服务

在使用STS访问OSS时，需要设置**stsToken**参数，如下面的例子所示：

```
var OSS = require('ali-oss');
var STS = OSS.STS;
var co = require('co');

var sts = new STS({
  accessKeyId: '<子账号的AccessKeyId>',
  accessKeySecret: '<子账号的AccessKeySecret>'
});

co(function* () {
```

```
  var token = yield sts.assumeRole(
    '<role-arn>', '<policy>', '<expiration>', '<session-name>');

  var client = new OSS({
    region: '<region>',
    accessKeyId: token.credentials.AccessKeyId,
    accessKeySecret: token.credentials.AccessKeySecret,
    stsToken: token.credentials.SecurityToken,
    bucket: '<bucket-name>'
  });
}).catch(function (err) {
  console.log(err);
});
```

在向STS申请临时token时，还可以指定自定义的STS Policy。这样申请的临时权限是**所扮演角色的权限与Policy指定的权限的交集**。下面的例子将通过指定 STS Policy申请对my-bucket的只读权限，并指定临时token的过期时间为15分钟：

```
var OSS = require('ali-oss');
var STS = OSS.STS;
var co = require('co');

var sts = new STS({
  accessKeyId: '<子账号的AccessKeyId>',
  accessKeySecret: '<子账号的AccessKeySecret>'
});

var policy = {
  "Statement": [
    {
      "Action": [
        "oss:Get*"
      ],
      "Effect": "Allow",
      "Resource": ["acs:oss:*:*:my-bucket/*"]
    }
  ],
  "Version": "1"
};

co(function* () {
  var token = yield sts.assumeRole(
    '<role-arn>', policy, 15 * 60, '<session-name>');

  var client = new OSS({
    region: '<region>',
    accessKeyId: token.credentials.AccessKeyId,
    accessKeySecret: token.credentials.AccessKeySecret,
    stsToken: token.credentials.SecurityToken,
    bucket: '<bucket-name>'
  });
}).catch(function (err) {
  console.log(err);
```

```
});
```

1.2.6.10 设置访问权限

OSS允许用户对Bucket和Object分别设置访问权限，方便用户控制自己的资源可以被如何访问。对于Bucket，有三种访问权限：

- public-read-write 允许匿名用户向该Bucket中创建/获取/删除Object
- public-read 允许匿名用户获取该Bucket中的Object
- private 不允许匿名访问，所有的访问都要经过签名

创建Bucket时，默认是private权限。之后用户可以通过putBucketACL来设置Bucket的权限，通过getBucketACL来获取Bucket的权限。

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.getBucketACL('bucket-name');
  console.log(result);
  var result = yield client.putBucketACL('bucket-name', 'region', 'acl');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

对于Object，有四种访问权限：

- default 继承所属的Bucket的访问权限，即与所属Bucket的权限值一样
- public-read-write 允许匿名用户读写该Object
- public-read 允许匿名用户读该Object
- private 不允许匿名访问，所有的访问都要经过签名

创建Object时，默认为default权限。之后用户可以通过putACL来设置Object的权限。

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.getACL('my-object');
  console.log(result.acl); // default
```

```
  yield client.putACL('my-object', 'public-read');
  var result = yield client.getACL('my-object');
  console.log(result.acl); // public-read
}).catch(function (err) {
  console.log(err);
});
```

说明：

- 如果设置了Object的权限（非default），则访问该Object时进行权限认证时会优先判断Object的权限，而Bucket的权限设置会被忽略。
- 允许匿名访问时（设置了public-read或者public-read-write权限），用户可以直接通过浏览器访问，例如：

  http://bucket-name.oss-cn-hangzhou.aliyuncs.com/object.jpg

1.2.6.11 管理生命周期

OSS允许用户对Bucket设置生命周期规则，以自动淘汰过期掉的文件，节省存储空间。用户可以同时设置多条规则，一条规则包含：

- 规则ID，用于标识一条规则，不能重复
- 受影响的文件前缀，此规则只作用于符合前缀的文件
- 过期时间，有两种指定方式：
  - 指定距文件最后修改时间N天过期
  - 指定在具体的某一天过期，即在那天之后符合前缀的文件将会过期，**而不论文件的最后修改时间**。不推荐使用。
- 是否生效

设置生命周期规则

通过putBucketLifecycle来设置生命周期规则：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.putBucketLifecycle('bucket-name', 'region', [
    {
      id: 'rule1',
      status: 'Enable',
```

```
      prefix: 'foo/',
      days: 3
    },
    {
      id: 'rule2',
      status: 'Disabled',
      date: '2022-10-11T00:00:00.000Z'
    }
  ]);
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

查看生命周期规则

通过getBucketLifecycle来查看生命周期规则：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.getBucketLifecycle('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

清空生命周期规则

通过deleteBucketLifecycle设置一个空的Rule数组来清空生命周期规则：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.deleteBucketLifecycle('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
```

```
});
```

1.2.6.12 设置访问日志

OSS允许用户对Bucket设置访问日志记录，设置之后对于Bucket的访问会被记录成日志，日志存储在OSS上由用户指定的Bucket中，文件的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中TargetPrefix由用户指定。

开启Bucket日志

通过putBucketLogging来开启日志功能：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.putBucketLogging('bucket-name', 'region', 'logs/');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

查看Bucket日志设置

通过getBucketLogging来查看日志设置：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.getBucketLogging('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

关闭Bucket日志

通过deleteBucketLogging来关闭日志功能：

```
var co = require('co');
```

```
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.deleteBucketLogging('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

1.2.6.13 静态网站托管

在自定义域名绑定中提到，OSS允许用户将自己的域名指向OSS服务的地址。这样用户访问他的网站的时候，实际上是在访问OSS的Bucket。对于网站，需要指定首页(index)和出错页(error)分别对应的Bucket中的文件名。

设置托管页面

通过putBucketWebsite来设置托管页面：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.putBucketLogging('bucket-name', 'region', {
    index: 'index.html',
    error: 'error.html'
  });
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

查看托管页面

通过getBucketWebsite来查看托管页面：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.getBucketLogging('bucket-name', 'region');
```

```
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

清除托管页面

通过deleteBucketWebsite来清除托管页面：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.deleteBucketLogging('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

1.2.6.14 设置防盗链

OSS是按使用收费的服务，为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

设置Referer白名单

通过putBucketReferer设置Referer白名单：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.putBucketReferer('bucket-name', 'region', true, [
    'my-domain.com',
    '*.example.com'
  ]);
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

查看Referer白名单

通过getBucketReferer设置Referer白名单：

```
var co = require('co');
```

```
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.getBucketReferer('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

清空Referer白名单

通过deleteBucketReferer设置清空Referer白名单：

```
var co = require('co');
var OSS = require('ali-oss')
var client = new OSS({
  region: '<Your region>'
  accessKeyId: '<Your AccessKeyId>',
  accessKeySecret: '<Your AccessKeySecret>',
  bucket: '<Your bucket name>'
});
co(function* () {
  var result = yield client.deleteBucketReferer('bucket-name', 'region');
  console.log(result);
}).catch(function (err) {
  console.log(err);
});
```

1.2.6.15 图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

图片处理基础功能

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义

- 通过级联处理调用多个图片处理功能

图片处理使用

图片处理使用标准的 HTTP GET 请求来访问，所有的处理参数是编码在 URL 中的QueyString。

匿名访问

如果图片文件（Object）的访问权限是公共读，如下表所示的权限，则可以匿名访问图片服务。

Bucket权限	Object权限
公共读私有写（public-read）或公共读写（public-read-write）	默认（default）
任意权限	公共读私有写（public-read）或 公共读写（public-read-write）

通过如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value
```

- bucket：用户的存储空间（bucket）名称
- endpoint：用户存储空间所在数据中心的访问域名
- object：用户上传在OSS上的图片文件
- image：图片处理保留标志符
- action：用户对图片做的操作，如缩放、裁剪、旋转等
- parame：用户对图片做的操作所对应的参数

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,
w_100
```

自定义样式，使用如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=x-oss-process=style/name
```

- style：用户自定义样式系统保留标志符
- name：自定义样式名称，即控制台定义样式的规则名

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=style/oss-pic-style-w-100
```

通过级联处理，可以对一张图片顺序实施多个操作，格式如下：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value/action,parame_value/...
```

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100/rotate,90
```

图片服务也支持HTTPS访问，例如：

```
https://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100
```

授权访问

对私有权限的文件（Object），如下表所示的权限，必须通过授权才能访问图片服务。

Bucket权限	Object权限
私有读写（private）	默认权限（default）
任意权限	私有读写（private）

生成带签名的图片处理的URL代码如下：

```
var OSS = require('ali-oss');
var client = new OSS({
  accessKeyId: '<accessKeyId>',
  accessKeySecret: '<accessKeySecret>',
  bucket: '<bucketName>',
  endpoint: '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
});
// 过期时间10分钟，图片处理式样"image/resize,w_300"
var signUrl = client.signatureUrl('example.jpg', {expires: 600, 'process' : 'image/resize,w_300'});
console.log("signUrl="+signUrl);
```

说明：

- 授权访问支持**自定义样式**、**HTTPS**、**级联处理**。
- 指定过期时间expires的单位是秒。

SDK访问

对于任意权限的图片文件，都可以直接使用 **SDK** 访问图片、进行处理。

说明：

SDK处理图片文件支持**自定义样式**、**HTTPS**、**级联处理**。

基础操作

图片处理的基础操作包括，获取图片信息、格式转换、缩放、裁剪、旋转、效果、水印等。

```
var OSS = require('ali-oss');
var co = require('co');
var client = new OSS({
  accessKeyId: '<accessKeyId>',
  accessKeySecret: '<accessKeySecret>',
  bucket: '<bucketName>',
  endpoint: '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
});
// 缩放
co(function* () {
  var result = yield client.get('example.jpg', './example-resize.jpg',
                          {process: 'image/resize,m_fixed,w_100,h_100'});
}).catch(function (err) {
  console.log(err);
});
// 裁剪
co(function* () {
  var result = yield client.get('example.jpg', './example-crop.jpg',
                          {process: 'image/crop,w_100,h_100,x_100,y_100,r_1'});
}).catch(function (err) {
  console.log(err);
});
// 旋转
co(function* () {
  var result = yield client.get('example.jpg', './example-rotate.jpg',
                          {process: 'image/rotate,90'});
}).catch(function (err) {
  console.log(err);
});
// 锐化
co(function* () {
  var result = yield client.get('example.jpg', './example-sharpen.jpg',
                          {process: 'image/sharpen,100'});
}).catch(function (err) {
  console.log(err);
});
// 水印
co(function* () {
  var result = yield client.get('example.jpg', './example-watermark.jpg',
                          {process: 'image/watermark,text_SGVsbG8g5Zu-54mH5pyN5YqhIQ'});
}).catch(function (err) {
  console.log(err);
});
// 格式转换
co(function* () {
  var result = yield client.get('example.jpg', './example-format.jpg',
                          {process: 'image/format,png'});
}).catch(function (err) {
```

```
  console.log(err);
});
// 图片信息
co(function* () {
  var result = yield client.get('example.jpg', './example-info.txt',
                                {process: 'image/info'});
}).catch(function (err) {
  console.log(err);
});
```

自定义样式

说明：

需要事先到oss控制台添加自定义式样example-sytle。

```
var OSS = require('ali-oss');
var co = require('co');
var client = new OSS({
  accessKeyId: '<accessKeyId>',
  accessKeySecret: '<accessKeySecret>',
  bucket: '<bucketName>',
  endpoint: '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
});
// 自定义样式
co(function* () {
  var result = yield client.get('example.jpg', './example-custom-style.jpg',
                                {process: 'style/example-sytle'"});
}).catch(function (err) {
  console.log(err);
});
```

级联处理

```
var OSS = require('ali-oss');
var co = require('co');
var client = new OSS({
  accessKeyId: '<accessKeyId>',
  accessKeySecret: '<accessKeySecret>',
  bucket: '<bucketName>',
  endpoint: '<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>'
});
// 级联处理
co(function* () {
  var result = yield client.get('example.jpg', './example-cascade.jpg',
                                {process: 'image/resize,m_fixed,w_100,h_100/rotate,90'});
}).catch(function (err) {
  console.log(err);
});
```

图片处理工具

- 可视化图片处理工具 *ImageStyleViever*，可以直观的看到OSS图片处理的结果。
- OSS图片处理的功能、使用演示页面。

1.2.6.16 异常

使用SDK时如果请求出错，会有相应的异常抛出。服务器端异常通常会包含以下信息：

- status: 出错请求的HTTP状态码
- code: OSS的错误码
- message: OSS的错误信息
- requestId: 标识该次请求的UUID；当您无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助。

调试

在Node.js中您可以通过设置DEBUG环境变量来开启调试：

```
DEBUG=ali-oss node app.js
```

在浏览器环境中您可以通过在console中设置localStorage.debug变量来开启调试：

```
localStorage.debug = 'ali-oss'
```

1.2.6.17 浏览器应用

浏览器支持

- IE(>=10)和Edge
- 主流版本的Chrome/Firefox/Safari
- 主流版本的Android/iOS/WindowsPhone

说明：

在IE中使用需要在引入sdk之前引入promise库：

```
<script src="https://www.promisejs.org/polyfills/promise-6.1.0.js"></script>
<script src="http://gosspublic.alicdn.com/aliyun-oss-sdk.min.js"></script>
```

设置

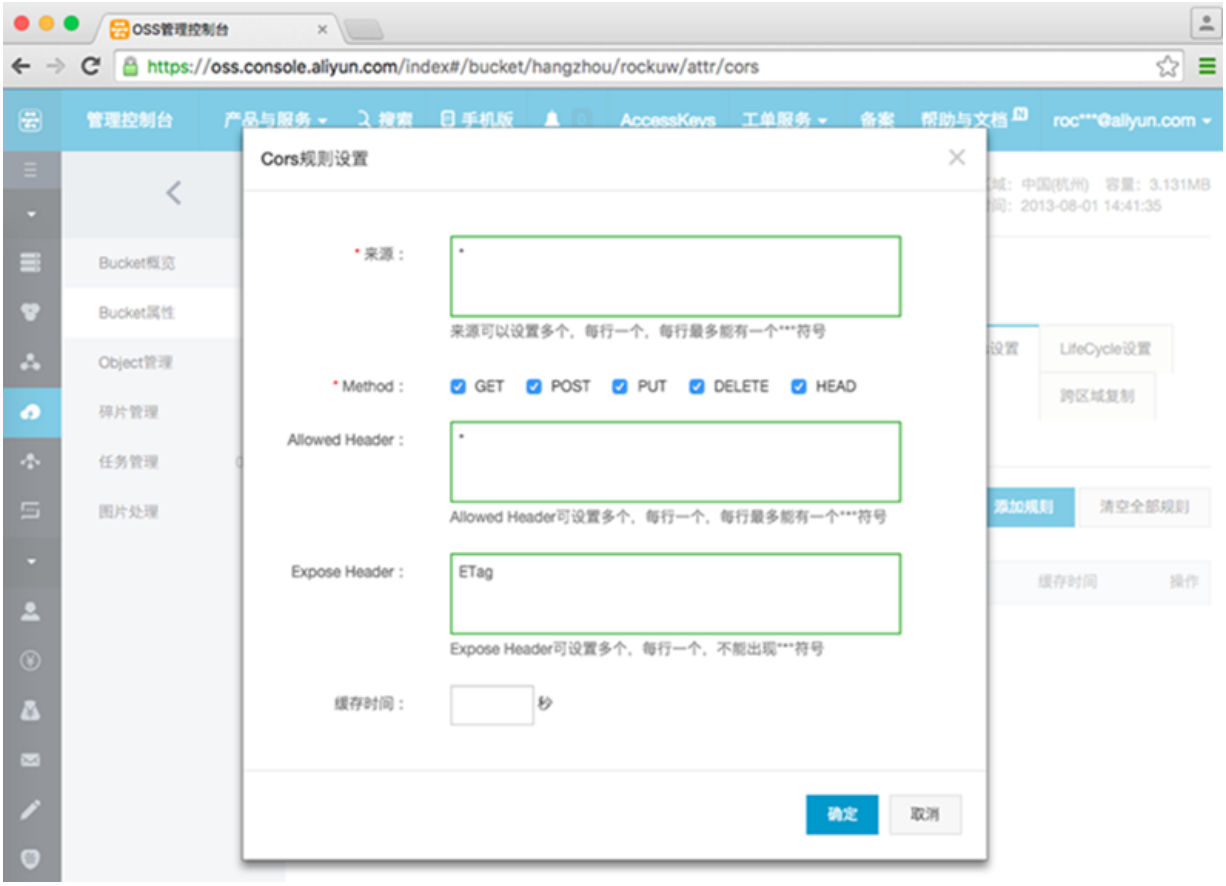
Bucket设置

从浏览器中直接访问OSS需要开通Bucket的CORS 设置：

- 将allowed origins设置成 *
- 将allowed methods设置成 PUT, GET, POST, DELETE, HEAD
- 将allowed headers设置成 *

- 将expose headers设置成 ETag

> **说明**：
> 请将该条CORS规则设置成所有CORS规则的第一条。



STS设置

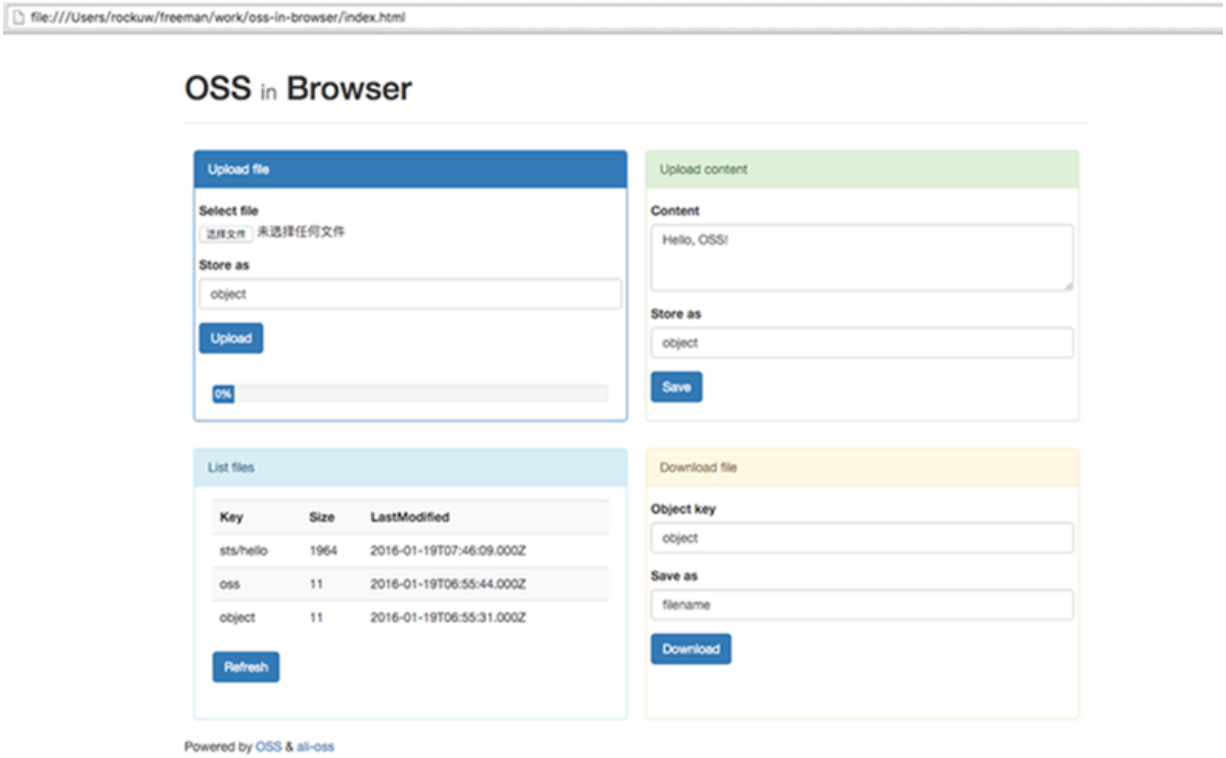
为了不在网页中暴露AccessKeyId和AccessKeySecret，我们采用STS进行临时授权。授权服务器由用户维护，只有认证的用户才能向授权服务器申请临时权限。

例子

下面我们将使用SDK开发一个网页应用，包含4个功能：

- 上传文件
- 上传文本
- 列出文件
- 下载文件

完整的代码可以在*oss-in-browser*找到。最终的效果如下：



下面的介绍将以**上传文件**为例。

1. 创建网页

对于**上传文件**的功能，在网页上需要4个控件：

- 一个文件选择框，用于选择需要上传的文件
- 一个文本框，用于输入上传到OSS的Object名字
- 一个按钮，用于开始上传
- 一个进度条，用于显示上传的进度

下面的代码创建了这4个控件，其中class="xxx"是为了使用*Bootstrap*样式，可以先忽略。

```
<div class="form-group">
  <label>Select file</label>
  <input type="file" id="file" />
</div>
<div class="form-group">
  <label>Store as</label>
  <input type="text" class="form-control" id="object-key-file" value="object" />
</div>
<div class="form-group">
  <input type="button" class="btn btn-primary" id="file-button" value="Upload" />
</div>
<div class="form-group">
  <div class="progress">
    <div id="progress-bar"
         class="progress-bar"
         role="progressbar"
```

```
        aria-valuenow="0"
        aria-valuemin="0"
        aria-valuemax="100" style="min-width: 2em;">
      0%
    </div>
  </div>
</div>
```

2. 添加样式

接下来为网页添加一些样式，让它更好看一些，这里我们用到了*Bootstrap*。在网页的**<head>**标签中加入样式：

```
<head>
  <title>OSS in Browser</title>
  <link rel="stylesheet" href="bootstrap.min.css" />
</head>
```

其中bootstrap.min.css可以在bootstrap的官网下载。

3. 添加脚本

到目前为止，网页是静态的，点击上面的按钮也不会有反应。接下来的重点是为它添加一些JavaScript脚本，让它能够上传文件/下载文件/列出文件。

首先在**<head>**标签中引入SDK的开发包：

```
<head>
  <title>OSS in Browser</title>
  <link rel="stylesheet" href="bootstrap.min.css" />
  <script type="text/javascript" src="http://gosspublic.alicdn.com/aliyun-oss-sdk-4.4.4.min.js"></
script>
  <script type="text/javascript" src="app.js"></script>
</head>
```

其中app.js是真正执行上传文件的代码，内容如下：

```
var appServer = 'http://localhost:3000';
var bucket = 'js-sdk-bucket-sts';
var region = 'oss-cn-hangzhou';
var urllib = OSS.urllib;
var OSS = OSS.Wrapper;
var STS = OSS.STS;
var applyTokenDo = function (func) {
  var url = appServer;
  return urllib.request(url, {
    method: 'GET'
  }).then(function (result) {
    var creds = JSON.parse(result.data);
    var client = new OSS({
      region: region,
      accessKeyId: creds.AccessKeyId,
      accessKeySecret: creds.AccessKeySecret,
      stsToken: creds.SecurityToken,
      bucket: bucket
    });
```

```
    return func(client);
  });
};
var progress = function (p) {
  return function (done) {
    var bar = document.getElementById('progress-bar');
    bar.style.width = Math.floor(p * 100) + '%';
    bar.innerHTML = Math.floor(p * 100) + '%';
    done();
  }
};
var uploadFile = function (client) {
  var file = document.getElementById('file').files[0];
  var key = document.getElementById('object-key-file').value.trim() || 'object';
  console.log(file.name + ' => ' + key);
  return client.multipartUpload(key, file, {
    progress: progress
  }).then(function (res) {
    console.log('upload success: %j', res);
    return listFiles(client);
  });
};
window.onload = function () {
  document.getElementById('file-button').onclick = function () {
    applyTokenDo(uploadFile);
  }
};
```

上传一个文件分为以下步骤：

- 向授权服务器申请临时权限。其中授权服务器是**网站开发者**构建的用于向**终端用户**提供临时授权的服务。开发者可以在授权时为不同的用户提供不同的权限。授权服务器可以参考这个例子，为了简便，例子中授权服务器直接向用户返回临时凭证。
- 使用临时密钥创建OSS Client。
- 通过multipartUpload上传选中的文件，并通过**progress**参数设置进度条。

4. 其他功能

上传文本内容、获取文件列表、下载文件等功能请参考代码示例：*OSS in Browser*。

1.2.7 PHP-SDK

1.2.7.1 安装

SDK

- GitHub地址：*https://github.com/aliyun/aliyun-oss-php-sdk*
- 版本迭代：*https://github.com/aliyun/aliyun-oss-php-sdk/releases*

环境要求

- PHP 5.3+，可通过php -v命令查看当前的PHP版本

- cURL 扩展，可通过php -m命令查看curl扩展是否已经安装好

说明：

- Ubuntu下可以使用apt-get包管理器安装php的cURL扩展 sudo apt-get install php-curl
- CentOS下可以使用yum包管理器安装php的cURL扩展 sudo yum install php-curl
- Windows下php、php-curl的安装方法请参见 *WIN下编译使用Aliyun OSS PHP SDK*
- PHP SDK暂时不支持PHP7

安装

composer方式

如果您通过composer管理您的项目依赖，可以在你的项目根目录运行：

```
composer require aliyuncs/oss-sdk-php
```

或者在你的composer.json中声明对Aliyun OSS SDK for PHP的依赖：

```
"require": {
    "aliyuncs/oss-sdk-php": "~2.0"
}
```

然后通过composer install安装依赖。完成后，目录结构应该像下面这样：

```
.
├── app.php
├── composer.json
├── composer.lock
└── vendor
```

其中app.php是用户的应用程序，*vendor/*目录下包含了所依赖的库，用户需要在app.php中引入依赖：

```
require_once __DIR__ . '/vendor/autoload.php';
```

说明：

- 如果您的项目中已经引用过autoload.php，则加入了SDK的依赖之后，不需要再引入autoload.php了。
- 如果使用composer出现网络错误，可以使用composer中国区的镜像源，方法是在命令行执行：composer config -g repositories.packagist composer http://packagist.phpcomposer.com

phar方式

使用phar单文件方式，在发布页面中，选择相应版本，下载已经打包好的phar文件，然后在你的代码中引入这个文件即可：

```
require_once '/path/to/oss-sdk-php.phar';
```

源码方式

使用SDK源码，在发布页面中，选择相应版本，下载已经打包好的zip文件，解压后的根目录中包含一个autoload.php文件，您需要在代码中引入这个文件：

```
require_once '/path/to/oss-sdk/autoload.php';
```

示例程序

通过下面的步骤运行示例程序：

- 解压下载到的sdk包
- 修改samples目录中的Config.php文件
  - 修改 **OSS_ACCESS_ID**，您从OSS获得的AccessKeyId
  - 修改 **OSS_ACCESS_KEY**，您从OSS获得的AccessKeySecret
  - 修改 **OSS_ENDPOINT**，您选定的OSS数据中心访问域名，如 http://oss-cn-hangzhou.aliyuncs.com
  - 修改 **OSS_TEST_BUCKET**，您要用来运行sample使用的bucket，sample 程序会在这个bucket中创建一些文件，注意不能用生产环境的bucket，以免污染用户数据
- 到samples目录中执行 php RunAll.php，也可以单个运行某个Sample文件

示例程序包括以下内容：

示例文件	示例内容
Object.php	展示了Object操作的用法，包括上传、下载、复制、删除、列举、元信息等
MultipartUpload.php	展示了大文件上传、分片上传的用法
Signature.php	展示了URL签名授权访问的用法
Callback.php	展示了上传回调的用法
Image.php	展示了图片处理的用法
LiveChannel.php	展示了LiveChannel的用法
Bucket.php	展示了Bucket管理操作的用法，包括创建、删除、列举、权限等

示例文件	示例内容
BucketLifecycle.php	展示了如何设置/读取/清除Bucket的生命周期
BucketLogging.php	展示了如何设置/读取/清除Bucket的日志
BucketReferer.php	展示了如何设置/读取/清除Bucket的防盗链
BucketWebsite.php	展示了如何设置/读取/清除Bucket的静态网站托管
BucketCors.php	展示了如何设置/读取/清除Bucket的跨域资源访问

旧版本

本版本相对于 **1.*.*** 版本是一个大版本升级，接口不再兼容，建议用户使用最新版本的SDK，如果您还是使用 **2.0.0** 版本以下的sdk，相应文档可以从此处下载。

1.2.7.2 初始化

OSS\OssClient 是SDK的客户端类，使用者可以通过OssClient提供的接口管理存储空间(Bucket)和文件(Object)等。

确定Endpoint

Endpoint是阿里云OSS服务在各个区域的地址，目前支持两种形式。

Endpoint类型	解释
OSS区域地址	使用OSS Bucket所在区域地址
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS区域地址

使用OSS Bucket所在区域地址，Endpoint查询可以有下面两种方式：

- 您可以登录阿里云OSS控制台，进入Bucket概览页，Bucket域名的后缀部分：如bucket-1.oss-cn-hangzhou.aliyuncs.com的oss-cn-hangzhou.aliyuncs.com部分为该Bucket的外网Endpoint。

CNAME

您可以将自己拥有的域名通过CNAME绑定到某个存储空间（Bucket）上，然后通过自己域名访问存储空间内的文件。

比如您要将域名new-image.xxxxx.com绑定到深圳区域的名称为image的存储空间上：

您需要到您的域名xxxxx.com托管商那里设定一个新的域名解析，将 http://new-image.xxxxx.com 解析到 http://image.oss-cn-shenzhen.aliyuncs.com，类型为CNAME

配置密钥

要接入阿里云OSS，您需要拥有一对有效的 AccessKey(包括AccessKeyId和AccessKeySecret)用来进行签名认证。可以通过如下步骤获得：

- 注册阿里云账号
- 申请*AccessKey*

在获取到 AccessKeyId和 AccessKeySecret之后，您可以按照下面步骤进行初始化。

新建OssClient

使用OSS域名新建OssClient

```
<?php
use OSS\OssClient;
use OSS\Core\OssException;
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您选定的OSS数据中心访问域名，例如http://oss-cn-hangzhou.aliyuncs.com>";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
} catch (OssException $e) {
    print $e->getMessage();
}
```

使用自定义域名(CNAME)新建OssClient

```
<?php
use OSS\OssClient;
use OSS\Core\OssException;
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您的绑定在某个Bucket上的自定义域名>";
try {
    $ossClient = new OssClient(
        $accessKeyId, $accessKeySecret, $endpoint, true /* use cname */);
} catch (OssException $e) {
    print $e->getMessage();
}
```

其中OssClient的构造函数中，第4个参数是含义是是否使用自定义域名，在使用CNAME的时候需要将它置成**true**。而如果使用的是OSS官方域名，则不需要填此项，或者填为**false**。

> **说明**：
> 使用自定义域名时，无法使用ListBuckets接口。

配置网络参数

我们可以用ClientConfiguration设置一些网络参数：

```
<?php
$ossClient->setTimeout(3600 /* seconds */);
$ossClient->setConnectTimeout(10 /* seconds */);
```

其中：

- setTimeout 设置请求超时时间，单位秒，默认是5184000秒，这里建议不要设置太小，如果上传文件很大，消耗的时间会比较长。
- setConnectTimeout 设置连接超时时间，单位秒，默认是10秒。

1.2.7.3 快速入门

请确认您已经熟悉 OSS的基本概念，如Bucket、Object、Endpoint、AccessKeyId和AccessKeySecret等。本节您将看到如何快速使用OSS PHP SDK，完成进行常见操作，如创建存储空间、上传文件、下载文件等。

常用类

类名	解释
OSS\OssClient	OSS客户端类，用户通过OssClient的实例调用接口
OSS\Core\OssException	OSS异常类，用户在使用的过程中，只需要注意这个异常

基本操作

创建存储空间

您可以按照下面的代码新建一个存储空间(Bucket)：

```
<?php
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您选定的OSS数据中心访问域名，例如http://oss-cn-hangzhou.aliyuncs.com>";
$bucket = "<您使用的存储空间名称，注意命名规范>";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
    $ossClient->createBucket($bucket);
} catch (OssException $e) {
    print $e->getMessage();
}
```

上传文件

文件(Object)是OSS中最基本的数据单元，您可以把它简单地理解为文件，用下面代码可以实现上传：

```
<?php
```

```
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您选定的OSS数据中心访问域名，例如http://oss-cn-hangzhou.aliyuncs.com>";
$bucket= " <您使用的Bucket名字，注意命名规范>";
$object = " <您使用的Object名字，注意命名规范>";
$content = "Hi, OSS.";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
    $ossClient->putObject($bucket, $object, $content);
} catch (OssException $e) {
    print $e->getMessage();
}
```

下载文件

上传文件后，您可以读取它的内容。以下代码获取Object的文本内容：

```
<?php
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您选定的OSS数据中心访问域名，例如http://oss-cn-hangzhou.aliyuncs.com>";
$bucket= " <您使用的Bucket名字，注意命名规范>";
$object = " <您使用的Object名字，注意命名规范>";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
    $content = $ossClient->getObject($bucket, $object);
    print("object content: " . $content);
} catch (OssException $e) {
    print $e->getMessage();
}
```

列举文件

当完成上传文件操作后，可能需要查看存储空间下包含哪些文件。以下代码展示如何列举存储空间下的文件：

```
<?php
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您选定的OSS数据中心访问域名，例如http://oss-cn-hangzhou.aliyuncs.com>";
$bucket= " <您使用的Bucket名字，注意命名规范>";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
    $listObjectInfo = $ossClient->listObjects($bucket);
    $objectList = $listObjectInfo->getObjectList();
    if (!empty($objectList)) {
        foreach ($objectList as $objectInfo) {
        print($objectInfo->getKey() . "\t" . $objectInfo->getSize() . "\t" . $objectInfo->getLastMod
ified() . "\n");
        }
    }
} catch (OssException $e) {
    print $e->getMessage();
```

```
}
```

说明：

- 上面的代码默认列举100个object。

删除文件

以下代码可以删除指定存储空间下指定的文件(Object)：

```
<?php
$accessKeyId = "<您从OSS获得的AccessKeyId>";
$accessKeySecret = "<您从OSS获得的AccessKeySecret>";
$endpoint = "<您选定的OSS数据中心访问域名，例如http://oss-cn-hangzhou.aliyuncs.com>";
$bucket= " <您使用的Bucket名字，注意命名规范>";
$object = " <您使用的Object名字，注意命名规范>";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
    $ossClient->deleteObject($bucket, $object);
} catch (OssException $e) {
    print $e->getMessage();
}
```

返回结果处理

OssClient提供的接口返回数据分为两类：

- Put，Delete类接口返回null，如果没有OssException，即可认为操作成功
- Get，List类接口返回对应的数据，如果没有OssException，即可认为操作成功

例如：

```
<?php
$bucketListInfo = $ossClient->listBuckets();
$bucketList = $bucketListInfo->getBucketList();
foreach($bucketList as $bucket) {
    print($bucket->getLocation() . "\t" . $bucket->getName() . "\t" . $bucket->getCreatedate() . "\n
");
}
```

上面代码中的`$bucketListInfo`的数据类型是**OSS\Model\BucketListInfo**。

1.2.7.4 管理存储空间

Bucket是OSS上的存储空间，也是计费、权限控制、日志记录等高级功能的管理实体。

说明：

以下场景的完整代码请参考： *GitHub*。

新建存储空间

您可以使用OssClient::createBucket新建存储空间：

```
<?php
/**
 * 创建一存储空间
 * acl 指的是bucket的访问控制权限，有三种，私有读写，公共读私有写，公共读写。
 * 私有读写就是只有bucket的拥有者或授权用户才有权限操作
 * 三种权限分别对应OSSClient::OSS_ACL_TYPE_PRIVATE，
 *             OssClient::OSS_ACL_TYPE_PUBLIC_READ,
 *             OssClient::OSS_ACL_TYPE_PUBLIC_READ_WRITE
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket 要创建的bucket名字
 * @return null
 */
function createBucket($ossClient, $bucket)
{
    try {
        $ossClient->createBucket($bucket, OssClient::OSS_ACL_TYPE_PRIVATE);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- Bucket的名字是**全局唯一的**，所以您需要保证Bucket名称不与别人重复。

判断存储空间是否存在

您可以使用OssClient::doesBucketExist判断存储空间是否存在：

```
/**
 *  判断Bucket是否存在
 *
 * @param OssClient $ossClient OssClient实例
 * @param string $bucket 存储空间名称
 */
function doesBucketExist($ossClient, $bucket)
{
    try {
        $res = $ossClient->doesBucketExist($bucket);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    if ($res === true) {
        print(__FUNCTION__ . ": OK" . "\n");
    } else {
        print(__FUNCTION__ . ": FAILED" . "\n");
```

```
    }
}
```

列出用户所有的存储空间

您可以使用OssClient::listBuckets列出用户所有的存储空间：

```
<?php
/**
 * 列出用户所有的Bucket
 *
 * @param OssClient $ossClient OssClient实例
 * @return null
 */
function listBuckets($ossClient)
{
    $bucketList = null;
    try{
        $bucketListInfo = $ossClient->listBuckets();
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    $bucketList = $bucketListInfo->getBucketList();
    foreach($bucketList as $bucket) {
        print($bucket->getLocation() . "\t" . $bucket->getName() . "\t" . $bucket->getCreatedate() .
 "\n");
    }
}
```

删除存储空间

您可以使用OssClient::deleteBucket删除存储空间：

```
<?php
/**
 * 删除存储空间
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket 待删除的存储空间名称
 * @return null
 */
function deleteBucket($ossClient, $bucket)
{
    try{
        $ossClient->deleteBucket($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 如果存储空间不为空（存储空间中有文件或者分片上传的分片），则存储空间无法删除。
- 必须先删除存储空间中的所有文件和分片后，存储空间才能成功删除。

设置存储空间访问权限(ACL)

除了在创建存储空间的时候能够对存储空间的 ACL 进行设置，也可以根据自己的业务需求对存储空间的ACL进行修改。这个操作只有该存储空间的创建者有权限执行。

存储空间有三种访问权限：

权限	PHP SDK对应值
私有读写	OssClient::OSS_ACL_TYPE_PRIVATE
公共读私有写	OssClient::OSS_ACL_TYPE_PUBLIC_READ
公共读写	OssClient::OSS_ACL_TYPE_PUBLIC_READ_WRITE

您可以使用OssClient::putBucketAcl设置存储空间的访问权限：

```
<?php
/**
 * 设置bucket的acl配置
 *
 * @param OssClient $ossClient OssClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putBucketAcl($ossClient, $bucket)
{
    $acl = OssClient::OSS_ACL_TYPE_PRIVATE;
    try {
        $ossClient->putBucketAcl($bucket, $acl);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

获取存储空间访问权限(ACL)

您可以使用OssClient::getBucketAcl获取存储空间的访问权限：

```
<?php
/**
 * 获取bucket的acl配置
 *
 * @param OssClient $ossClient OssClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getBucketAcl($ossClient, $bucket)
```

```
{
    try {
        $res = $ossClient->getBucketAcl($bucket);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    print('acl: ' . $res);
}
```

1.2.7.5 上传文件

在OSS中，用户操作的基本数据单元是文件（Object）。OSS PHP SDK提供了丰富的文件上传接口，可以通过以下方式上传文件：

- 字符串上传
- 文件上传
- 追加上传
- 分片上传

字符串上传、文件上传、追加上传的文件最大不能超过5GB。当文件较大时，请使用分片上传，分片上传文件大小不能超过48.8TB。

字符串上传

下面的代码实现了上传指定字符串中的内容到文件中：

```
<?php
/**
 * 上传字符串作为object的内容
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    $content = file_get_contents(__FILE__);
    try{
        $ossClient->putObject($bucket, $object, $content);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
```

```
}
```

上传本地文件

下面的代码实现了上传指定的本地文件到文件中：

```
<?php
/**
 * 上传指定的本地文件内容
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function uploadFile($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    $filePath = __FILE__;
    try{
        $ossClient->uploadFile($bucket, $object, $filePath);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

使用上述方法上传最大文件不能超过5G, 如果超过可以使用分片文件上传。

追加上传

简单上传，分片上传，断点续传上传，创建的Object都是Normal类型，这种Object在上传结束之后内容就是固定的，只能读取，不能修改。如果Object内容发生了改变，只能重新上传同名的Object来覆盖之前的内容，这也是OSS和普通文件系统使用的一个重大区别。正因为这种特性，在很多应用场景下会很不方便，典型比如视频监控、视频直播领域等，视频数据在实时的不断产生。

OSS提供了用户通过追加上传（Append Object）的方式在一个Object后面直接追加内容的功能。通过这种方式操作的Object的类型为Appendable Object，而其他的方式上传的Object类型为Normal Object。每次追加上传的数据都能够即时可读。

追加上传字符串

下面的代码实现了追加上传指定字符串到文件(Object)中：

```
<?php
/**
 * 字符串追加上传
 *
 * @param OssClient $ossClient OSSClient实例
```

```
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/append-test-object-name.txt";
    $content_array = array('Hello OSS', 'Hi OSS', 'OSS OK');
    try{
        $position = $this->ossClient->appendObject($this->bucket, $object, $content_array[0], 0);
        $position = $this->ossClient->appendObject($this->bucket, $object, $content_array[1], $
position);
        $position = $this->ossClient->appendObject($this->bucket, $object, $content_array[2], $
position);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

追加上传文件

下面的代码实现了追加上传本地文件(File)到OSS文件(Object)：

```
 /**
 * 文件追加上传
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/append-test-object-name.txt";
    try{
        $position = $this->ossClient->appendFile($this->bucket, $object, __FILE__, 0);
        $position = $this->ossClient->appendFile($this->bucket, $object, __FILE__, $position);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 追加上传的次数没有限制，文件大小上限为5GB。更大的文件请使用分片上传。
- 追加类型的文件(Append Object)暂时不支持copyObject操作。

分片上传

除了通过PutObject接口上传文件到OSS以外，OSS还提供了另外一种上传模式：Multipart Upload。用户可以在如下的应用场景内（但不仅限于此），使用Multipart Upload上传模式，如：

- 需要支持断点上传。
- 上传超过100MB大小的文件。
- 网络条件较差，和OSS的服务器之间的链接经常断开。
- 上传文件之前，无法确定上传文件的大小。

说明：

分片上传的完整代码请参见：*GitHub*。

分片上传本地文件

下面代码通过封装后的易用接口进行分片上传文件操作：

```
<?php
/**
 * 通过multipart上传文件
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function multiuploadFile($ossClient, $bucket)
{
    $object = "test/multipart-test.txt";
    $file = __FILE__;
    try{
        $ossClient->multiuploadFile($bucket, $object, $file);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ":  OK" . "\n");
}
```

分片上传本地目录

下面代码通过封装后的易用接口进行分片上传目录操作：

```
<?php
/**
 * 按照目录上传文件
 *
 * @param OssClient $ossClient OssClient
 * @param string $bucket 存储空间名称
 *
 */
function uploadDir($ossClient, $bucket) {
    $localDirectory = ".";
    $prefix = "samples/codes";
    try {
        $ossClient->uploadDir($bucket, $prefix, $localDirectory);
    }  catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
```

```
        printf($e->getMessage() . "\n");
        return;
    }
    printf(__FUNCTION__ . ": completeMultipartUpload OK\n");
}
```

原始接口分片上传

分片上传(MultipartUpload)一般的流程如下：

- 初始化一个分片上传任务（InitiateMultipartUpload）
- 逐个或并行上传分片（UploadPart）
- 完成上传（CompleteMultipartUpload）或取消分片上传(AbortMultipartUpload)

下面通过一个完整的示例说明了如何通过原始的api接口一步一步的进行分片上传操作，如果用户需要做断点续传等高级操作，可以参考下面代码：

```
<?php
/**
 * 使用基本的api分阶段进行分片上传
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @throws OssException
 */
function putObjectByRawApis($ossClient, $bucket)
{
    $object = "test/multipart-test.txt";
    /**
     *  step 1. 初始化一个分块上传事件，也就是初始化上传Multipart, 获取upload id
     */
    try{
        $uploadId = $ossClient->initiateMultipartUpload($bucket, $object);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": initiateMultipartUpload FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": initiateMultipartUpload OK" . "\n");
    /*
     * step 2. 上传分片
     */
    $partSize = 10 * 1024 * 1024;
    $uploadFile = __FILE__;
    $uploadFileSize = filesize($uploadFile);
    $pieces = $ossClient->generateMultiuploadParts($uploadFileSize, $partSize);
    $responseUploadPart = array();
    $uploadPosition = 0;
    $isCheckMd5 = true;
    foreach ($pieces as $i => $piece) {
        $fromPos = $uploadPosition + (integer)$piece[$ossClient::OSS_SEEK_TO];
        $toPos = (integer)$piece[$ossClient::OSS_LENGTH] + $fromPos - 1;
        $upOptions = array(
            $ossClient::OSS_FILE_UPLOAD => $uploadFile,
            $ossClient::OSS_PART_NUM => ($i + 1),
            $ossClient::OSS_SEEK_TO => $fromPos,
```

```
            $ossClient::OSS_LENGTH => $toPos - $fromPos + 1,
            $ossClient::OSS_CHECK_MD5 => $isCheckMd5,
        );
        if ($isCheckMd5) {
            $contentMd5 = OssUtil::getMd5SumForFile($uploadFile, $fromPos, $toPos);
            $upOptions[$ossClient::OSS_CONTENT_MD5] = $contentMd5;
        }
        //2. 将每一分片上传到OSS
        try {
            $responseUploadPart[] = $ossClient->uploadPart($bucket, $object, $uploadId, $
upOptions);
        } catch(OssException $e) {
            printf(__FUNCTION__ . ": initiateMultipartUpload, uploadPart - part#{$i} FAILED\n");
            printf($e->getMessage() . "\n");
            return;
        }
        printf(__FUNCTION__ . ": initiateMultipartUpload, uploadPart - part#{$i} OK\n");
    }
    $uploadParts = array();
    foreach ($responseUploadPart as $i => $eTag) {
        $uploadParts[] = array(
            'PartNumber' => ($i + 1),
            'ETag' => $eTag,
        );
    }
    /**
     * step 3. 完成上传
     */
    try {
        $ossClient->completeMultipartUpload($bucket, $object, $uploadId, $uploadParts);
    }  catch(OssException $e) {
        printf(__FUNCTION__ . ": completeMultipartUpload FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    printf(__FUNCTION__ . ": completeMultipartUpload OK\n");
}
```

说明：

- 上面程序一共分为三个步骤：1. initiate 2. uploadPart 3. complete
- 第二步UploadPart方法用来上传每一个分片，但是要注意以下几点：
- UploadPart 方法要求除最后一个Part以外，其他的Part大小都要大于或等于100KB。但是Upload Part接口并不会立即校验上传Part的大小（因为不知道是否为最后一块）；只有当Complete Multipart Upload的时候才会校验。
- OSS会将服务器端收到Part数据的MD5值在OssClient的UploadPart接口中返回；
- Part号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
- 每次上传Part时都要把流定位到此次上传块开头所对应的位置。

- 每次上传Part之后，OSS的返回结果会包含一个 PartETag 对象，它是上传块的ETag与块编号（PartNumber）的组合。在后续完成分片上传的步骤中会用到它，因此我们需要将其保存起来，然后在第三步complete的时候使用，具体操作参考上面代码。
- 分片上传任务初始化或上传部分分片后，可以使用abortMultipartUpload接口中止分片上传事件。当分片上传事件被中止后，就不能再使用这个Upload ID做任何操作，已经上传的分片数据也会被删除。

查看已上传的分片

查看上传的分片可以罗列出指定Upload ID所属的所有已经上传成功的分片。

通过下面代码，可以查看一次上传任务已上传的分片：

```
/**
 * 查看已上传的分片
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @param string $uploadId  upload Id
 * @return null
 */
function putObject($ossClient, $bucket, $uploadId)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try{
        $listPartsInfo = $ossClient->listParts($bucket, $object, $upload_id);
        foreach ($listPartsInfo->getListPart() as $partInfo) {
            print($partInfo->getPartNumber() . "\t" . $partInfo->getSize() . "\t" . $partInfo->getETag() .
 "\t" . $partInfo->getLastModified() . "\n");
        }
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

查看当前正在进行的分片上传任务

通过下面代码，可以得到当前正在进行中，还未完成的分片上传：

```
<?php
/**
 * 获取当前未完成的分片上传列表
 *
 * @param $ossClient OssClient
 * @param $bucket   string
 */
function listMultipartUploads($ossClient, $bucket) {
    $options = array(
                'delimiter' => '/',
                'max-uploads' => 100,
```

```
            'key-marker' => '',
            'prefix' => '',
            'upload-id-marker' => ''
        );
    try {
        $listMultipartUploadInfo = $ossClient->listMultipartUploads($bucket, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": listMultipartUploads FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    printf(__FUNCTION__ . ": listMultipartUploads OK\n");
    $listUploadInfo = $listMultipartUploadInfo->getUploads();
    var_dump($listUploadInfo);
}
```

上述例子中提到的$options参数说明：

key	说明
delimiter	是一个用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素——CommonPrefixes。
key-marker	与upload-id-marker参数一同使用来指定返回结果的起始位置。
max-uploads	限定此次返回Multipart Uploads事件的最大数目，如果不设定，默认为1000，max-uploads取值不能大于1000。
prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix。
upload-id-marker	与key-marker参数一同使用来指定返回结果的起始位置。

设置文件元信息(Object Meta)

文件元信息(Object Meta)，是对用户上传到OSS的文件的属性描述，分为两种：HTTP标准属性（HTTP Headers）和User Meta（用户自定义元信息）。 文件元信息可以在各种方式上传（字符串上传、文件上传、追加上传、分片上传），或拷贝文件时进行设置。元信息的名称大小写不敏感。

设定Http Header

OSS允许用户自定义Http header。Http header请参考 *RFC2616*。几个常用的http header说明如下：

名称	描述	默认值
Content-MD5	文件数据校验，设置了该值后OSS会启用文件内容MD5校验，把您提供的MD5与文件的MD5比较，不一致会抛出错误	无

名称	描述	默认值
Content-Type	文件的MIME，定义文件的类型及网页编码，决定浏览器将以什么形式、什么编码读取文件。如果用户没有指定则根据Key或文件名的扩展名生成，如果没有扩展名则填默认值	application/octet-stream
Content-Disposition	指示MINME用户代理如何显示附加的文件，打开或下载，及文件名称	无
Content-Length	上传的文件的长度，超过流/文件的长度会截断，不足为实际值	流/文件时间长度
Expires	缓存过期时间，OSS未使用，格式是格林威治时间（GMT）	无
Cache-Control	指定该Object被下载时的网页的缓存行为	无

下面的代码实现了上传时设置文件的Http Headers：

```
/**
 * 上传时设置文件的元数据
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    $content = file_get_contents(__FILE__);
    $options = array(
        OssClient::OSS_HEADERS => array(
            'Expires' => '2012-10-01 08:00:00',
            'Content-Disposition' => 'attachment; filename="xxxxxx"',
    ));
    try{
        $ossClient->putObject($bucket, $object, $content, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
```

```
}
```

说明：

- 通过设置文件的Content-Type，可以修改文件的类型。
- 通过设置文件的Content-Disposition，可以控制文件的下载行为。

用户自定义元信息

OSS支持用户自定义Object的元信息，对Object进行描述。

下面的代码实现了上传时设置文件的自定义元数据：

```
<?php
/**
 * 上传时设置文件的自定义元数据
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    $content = file_get_contents(__FILE__);
    $options = array(OssClient::OSS_HEADERS => array(
        'x-oss-meta-self-define-title' => 'user define meta info',
    ));
    try{
        $ossClient->putObject($bucket, $object, $content, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 文件的元信息可以通过OssClient::getObjectMeta获取。
- user meta的名称大小写不敏感，例如设置为：x-oss-meta-name，读取名字为**x-oss-meta-name**的参数即可。
- 一个文件可以有多个元信息，总大小不能超过8KB。

上传时使用MD5校验

为了确保PHP SDK发送的数据和OSS服务端接收到的数据一致，OSS支持MD5校验。文件上传时（字符串上传、文件上传、追加上传、分片上传）默认关闭MD5，如果您需要打开MD5校验，请上传文件的options设置。

下面的代码实现了上传时开启MD5校验：

```
<?php
/**
 * 上传时开启MD5校验
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    $options = array(OssClient::OSS_CHECK_MD5 => true);
    try{
        $ossClient->uploadFile($bucket, $object, __FILE__, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

使用Md5校验时，性能会有所损失。

1.2.7.6 下载文件

OSS PHP SDK提供了丰富的文件下载接口，用户可以通过以下方式从OSS下载文件(object)：

- 下载文件到本地文件
- 下载文件到内存
- 分段下载
- 条件下载

说明：

文件下载的完整代码请参见：*GitHub*。

下载文件到本地文件

以下代码可以下载OSS文件到本地文件：

```
<?php
```

```
/**
 * get_object_to_local_file
 *
 * 获取object
 * 将object下载到指定的文件
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getObjectToLocalFile($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/download-test-object-name.txt";
    $localfile = "download-test-object-name.txt";
    $options = array(
        OssClient::OSS_FILE_DOWNLOAD => $localfile,
    );
    try{
        $ossClient->getObject($bucket, $object, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK, please check localfile: 'upload-test-object-name.txt'" . "\n");
}
```

下载文件到内存

以下代码可以下载文件到内存：

```
<?php
/**
 * 获取object的内容
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try{
        $content = $ossClient->getObject($bucket, $object);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

范围下载

如果OSS文件较大，您只需要其中一部分数据，可以使用范围下载，下载指定范围的数据。如果指定的下载范围是0 - 100，则返回第0到第100个字节的数据，包括第100个，共101字节的数据，即[0，100]。如果指定的范围无效，则下载整个文件。

以下代码可以下载文件[0, 100]的内容到内存：

```
<?php
/**
 * 获取object的内容
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try{
        $options = array(OssClient::OSS_RANGE => '0-100');
        $content = $ossClient->getObject($bucket, $object ,  $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 范围下载的内容也可以下载的文件；
- 下载范围为闭区间[start, end]，包括两端的字节。

条件下载

下载文件时，可以指定一个或多个限定条件，满足限定条件时下载，不满足时报错，不下载文件。

可以使用的限定条件如下：

参数	说明	PHP SDK对应值
If-Modified-Since	如果指定的时间早于实际修改时间，则正常传送。否则返回错误。	OssClient::OSS_IF_MODIFIED_SINCE
If-Unmodified-Since	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件；否则返回错误。	OssClient::OSS_IF_UNMODIFIED_SINCE
If-Match	如果传入期望的ETag和object的ETag匹配，则正常传输；否则返回错误。	OssClient::OSS_IF_MATCH
If-None-Match	如果传入的ETag值和Object的ETag不匹配，则正常传输；否则返回错误。	OssClient::OSS_IF_NONE_MATCH

说明：

- 如果If-Modified-Since设定的时间不符合规范，直接返回文件，并返回200 OK；
- If-Modified-Since和If-Unmodified-Since可以同时存在，If-Match和If-None-Match也可以同时存在；
- 如果包含If-Unmodified-Since并且不符合或者包含If-Match并且不符合，返回412 precondition failed；
- 如果包含If-Modified-Since并且不符合或者包含If-None-Match并且不符合，返回304 Not Modified。

```
<?php
/**
 * 如果文件在指定的时间之后修改过，则下载文件
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try{
        $options = array(
            OssClient::OSS_HEADERS => array(
                OssClient::OSS_IF_MODIFIED_SINCE => "Fri, 13 Nov 2015 14:47:53 GMT"),
    );
        $content = $ossClient->getObject($bucket, $object ,  $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 文件的ETag值可以通过OssClient::getObjectMeta获取；
- 条件下载的内容也可以下载的文件。

1.2.7.7 管理文件

在OSS中，用户可以通过一系列的接口管理存储空间(Bucket)中的文件(Object)，比如ListObjects，DeleteObject，CopyObject，DoesObjectExist等。

说明：

以下场景的完整代码路径：*GitHub*。

列出存储空间中的文件

您可以使用OssClient::listObjects列出存储中间中的文件：

```
<?php
/**
 * 列出Bucket内所有目录和文件，根据返回的nextMarker循环调用listObjects接口得到所有文件和目录
 *
 * @param OssClient $ossClient OssClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function listAllObjects($ossClient, $bucket)
{
    //构造dir下的文件和虚拟目录
    for ($i = 0; $i < 100; $i += 1) {
        $ossClient->putObject($bucket, "dir/obj" . strval($i), "hi");
        $ossClient->createObjectDir($bucket, "dir/obj" . strval($i));
    }
    $prefix = 'dir/';
    $delimiter = '/';
    $nextMarker = '';
    $maxkeys = 30;
    while (true) {
        $options = array(
            'delimiter' => $delimiter,
            'prefix' => $prefix,
            'max-keys' => $maxkeys,
            'marker' => $nextMarker,
        );
        var_dump($options);
        try {
            $listObjectInfo = $ossClient->listObjects($bucket, $options);
        } catch (OssException $e) {
            printf(__FUNCTION__ . ": FAILED\n");
            printf($e->getMessage() . "\n");
            return;
        }
        // 得到nextMarker，从上一次listObjects读到的最后一个文件的下一个文件开始继续获取文件列表
        $nextMarker = $listObjectInfo->getNextMarker();
        $listObject = $listObjectInfo->getObjectList();
        $listPrefix = $listObjectInfo->getPrefixList();
        var_dump(count($listObject));
        var_dump(count($listPrefix));
        if ($nextMarker === '') {
            break;
        }
    }
}
```

上述例子中提到的$options参数说明：

key	说明
delimiter	用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素——CommonPrefixes

key	说明
prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix
max-keys	限定此次返回object的最大数，如果不设定，默认为100，max-keys取值不能大于1000
marker	设定结果从marker之后按字母排序的第一个开始返回

说明：

prefix/delimiter/marker/max-keys都是可选参数。

判断文件是否存在

您可以使用OssClient::doesObjectExist判断文件是否存在：

```
<?php
/**
 * 判断object是否存在
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket bucket名字
 * @return null
 */
function doesObjectExist($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try{
        $exist = $ossClient->doesObjectExist($bucket, $object);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    var_dump($exist);
}
```

创建虚拟目录

OSS是没有文件夹这个概念的，所有元素都是以Object来存储。创建文件夹本质上来说是创建了一个size为0的Object。对于这个Object可以上传下载，只是控制台会对以 / 结尾的Object以文件夹的方式展示。

```
<?php
/**
 * 创建虚拟目录
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
```

```
 */
function createObjectDir($ossClient, $bucket) {
    try{
        $ossClient->createObjectDir($bucket, "dir");
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

删除单个文件

您可以使用OssClient::deleteObject删除单个文件：

```
<?php
/**
 * 删除object
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket bucket名字
 * @return null
 */
function deleteObject($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try{
        $ossClient->deleteObject($bucket, $object);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

文件删除后将无法恢复。

删除多个文件

您可以使用OssClient::deleteObjects批量删除多个文件：

```
<?php
/**
 * 批量删除object
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket bucket名字
 * @return null
 */
function deleteObjects($ossClient, $bucket)
{
    $objects = array();
    $objects[] = "oss-php-sdk-test/upload-test-object-name.txt";
```

```
    $objects[] = "oss-php-sdk-test/upload-test-object-name.txt.copy";
    try{
        $ossClient->deleteObjects($bucket, $objects);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

拷贝文件

您可以使用OssClient::copyObject拷贝文件：

```
<?php
/**
 * 拷贝object
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket bucket名字
 * @return null
 */
function copyObject($ossClient, $bucket)
{
    $from_bucket = $bucket;
    $from_object = "oss-php-sdk-test/upload-test-object-name.txt";
    $to_bucket = $bucket;
    $to_object = $from_object . '.copy';
    try{
        $ossClient->copyObject($from_bucket, $from_object, $to_bucket, $to_object);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

OssClient::copyObject可以拷贝小于1GB的文件。拷贝一个大于1GB的文件时，请使用OssClient::uploadPartCopy。

修改文件元信息(Object Meta)

文件元信息(Object Meta)，是对用户上传到OSS的文件的属性描述，分为两种：HTTP标准属性（HTTP Headers）和User Meta（用户自定义元信息）。文件元信息可以在各种方式上传或者拷贝文件时进行设置。

HTTP标准属性，Cache-Control、Content-Disposition、Content-Encoding、Content-Language、Expires、Content-Length、Content-Type、Last-Modified等。

为了便于用户对Object进行更多描述。OSS中规定所有以x-oss-meta-为前缀的参数，则视为User Meta比如x-oss-meta-location。一个Object可以有多个类似的参数，但所有的User Meta总大小不能超过8k。这些User Meta信息会在下载文件/获取文件元数据时返回。

可以通过拷贝操作来实现修改已有文件元信息。如果拷贝操作的源文件地址和目标文件地址相同，都会直接替换源文件的文件元信息。

```
/**
 * 修改文件元信息
 * 利用copyObject接口的特性：当目的object和源object完全相同时，表示修改object的文件元信息
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function modifyMetaForObject($ossClient, $bucket)
{
    $fromBucket = $bucket;
    $fromObject = "oss-php-sdk-test/upload-test-object-name.txt";
    $toBucket = $bucket;
    $toObject = $fromObject;
    $copyOptions = array(
        OssClient::OSS_HEADERS => array(
            'Expires' => '2018-10-01 08:00:00',
            'Content-Disposition' => 'attachment; filename="xxxxxx"',
            'x-oss-meta-location' => 'location',
        ),
    );
    try{
        $ossClient->copyObject($fromBucket, $fromObject, $toBucket, $toObject, $copyOptions);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 通过修改文件的Content-Type，可以修改文件的类型。
- 通过修改文件的Content-Disposition，可以控制文件的下载行为。

获取文件的文件元信息(Object Meta)

您可以使用OssClient::getObjectMeta获取文件的文件元信息：

```
<?php
/**
 * 获取object meta, 也就是getObjectMeta接口
 *
 * @param OssClient $ossClient OssClient实例
```

```
 * @param string $bucket 存储空间名称
 * @return null
 */
function getObjectMeta($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try {
        $objectMeta = $ossClient->getObjectMeta($bucket, $object);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    if (isset($objectMeta[strtolower('Content-Disposition')]) &&
        'attachment; filename="xxxxxx"' === $objectMeta[strtolower('Content-Disposition')]
    ) {
        print(__FUNCTION__ . ": ObjectMeta checked OK" . "\n");
    } else {
        print(__FUNCTION__ . ": ObjectMeta checked FAILED" . "\n");
    }
}
```

设置文件访问权限

Object有四种访问权限：default（默认），private（私有读写），public-read（公共读私有写），public-read-write（公共读写），含义如下：

权限	描述	PHP SDK对应值
默认	Objec是遵循Bucket的读写权限，即Bucket是什么权限，Object就是什么权限，Object的默认权限	default
私有读写	Object是私有资源，即只有该Object的Owner拥有该Object的读写权限，其他的用户没有权限操作该Object	private
公共读私有写	Object是公共读资源，即非Object Owner只有Object的读权限，而Object Owner拥有该Object的读写权限	public-read
公共读写	Object是公共读写资源，即所有用户拥有对该Object的读写权限	public-read-write

Object的权限优先级高于Bucket。例如Bucket是private的，而Object ACL是公共读写，则访问这个Object时，先判断Object的ACL，所有用户都拥有这个Object的访问权限，即使这个Bucket是private。如果某个Object从来没设置过ACL，则访问权限遵循Bucket ACL。

您可以使用OssClient::putObjectAcl设置文件的访问权限：

```
<?php
/**
 * 获取文件范围权限
 *
 * @param OssClient $ossClient OssClient实例
```

```
 * @param string $bucket 存储空间名称
 * @return null
 */
function putObjectAcl($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    $acl = "public-read";
    try {
        $ossClient->putObjectAcl($bucket, $object, $acl);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

获取文件访问权限

您可以使用OssClient::getObjectAcl获取文件的访问权限：

```
<?php
/**
 * 获取object的访问权限
 *
 * @param OssClient $ossClient OssClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getObjectMeta($ossClient, $bucket)
{
    $object = "oss-php-sdk-test/upload-test-object-name.txt";
    try {
        $objectAcl = $ossClient->getObjectAcl($bucket, $object);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    var_dump($objectAcl);
}
```

1.2.7.8 授权访问

使用URL签名授权访问

通过生成签名URL的形式提供给用户一个临时的访问URL。在生成URL时，您可以指定URL过期的时间，从而限制用户长时间访问。

> **说明**：
> 以下场景的完整代码路径：*GitHub*。

使用私有的下载链接

生成GetObject的签名url示例如下：

```
<?php
/**
 * 生成GetObject的签名url,主要用于私有权限下的读访问控制
 *
 * @param $ossClient OssClient OSSClient实例
 * @param $bucket string bucket名称
 * @return null
 */
function getSignedUrlForGettingObject($ossClient, $bucket)
{
    $object = "test/test-signature-test-upload-and-download.txt";
    $timeout = 3600; // URL的有效期是3600秒
    try{
        $signedUrl = $ossClient->signUrl($bucket, $object, $timeout);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": signedUrl: " . $signedUrl. "\n");
    /**
     * 可以类似的代码来访问签名的URL，也可以输入到浏览器中去访问
     */
    $request = new RequestCore($signedUrl);
    $request->set_method('GET');
    $request->send_request();
    $res = new ResponseCore($request->get_response_header(), $request->get_respon
se_body(), $request->get_response_code());
    if ($res->isOK()) {
        print(__FUNCTION__ . ": OK" . "\n");
    } else {
        print(__FUNCTION__ . ": FAILED" . "\n");
    };
}
```

说明：

生成的URL默认以GET方式访问，这样，用户可以直接通过浏览器访问相关内容。

使用私有的上传链接

如果您想允许用户临时进行其他操作（比如上传，删除文件），可能需要签名其他方法的URL，如下：

```
<?php
/**
 * 生成PutObject的签名url,主要用于私有权限下的写访问控制
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket bucket名称
 * @return null
 * @throws OssException
 */
function getSignedUrlForPuttingObject($ossClient, $bucket)
{
```

```
    $object = "test/test-signature-test-upload-and-download.txt";
    $timeout = 3600;
    $options = NULL;
    try{
        $signedUrl = $ossClient->signUrl($bucket, $object, $timeout, "PUT");
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": signedUrl: " . $signedUrl. "\n");
    $content = file_get_contents(__FILE__);
    $request = new RequestCore($signedUrl);
    $request->set_method('PUT');
    $request->add_header('Content-Type', '');
    $request->add_header('Content-Length', strlen($content));
    $request->set_body($content);
    $request->send_request();
    $res = new ResponseCore($request->get_response_header(),
        $request->get_response_body(), $request->get_response_code());
    if ($res->isOK()) {
        print(__FUNCTION__ . ": OK" . "\n");
    } else {
        print(__FUNCTION__ . ": FAILED" . "\n");
    };
}
```

临时凭证(STS)上传和下载

介绍

OSS可以通过阿里云STS服务，临时进行授权访问。阿里云STS (Security Token Service) 是为云计算用户提供临时访问令牌的Web服务。通过STS，您可以为第三方应用或联邦用户（用户身份由您自己管理）颁发一个自定义时效和权限的访问凭证。STS更详细的解释请参考 *STS介绍*。

使用STS凭证创建OssClient

用户的client端拿到STS临时凭证后，通过其中安全令牌(SecurityToken)以及临时访问密钥(AccessKeyId, AccessKeySecret)生成OssClient。

通过下面代码可以使用STS临时凭证创建一个OssClient：

```
<?php
$accessKeyId = "<accessKeyId>";
$accessKeySecret = "<accessKeySecret>";
$securityToken = "<securityToken>";
$endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
```

```
$ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint, false, $securityToken);
```

1.2.7.9 静态网站托管

在自定义域名绑定中提到，OSS 允许用户将自己的域名指向OSS服务的地址。这样用户访问他的网站的时候，实 际上是在访问OSS的Bucket。对于网站，需要指定首页(index)和出错页(error) 分别对应的Bucket中的文件名。

说明：

以下场景的完整代码路径：*GitHub*。

设置静态网站托管

您可以通过OssClient::putBucketWebsite设置静态网站托管：

```
<?php
/**
 * 设置bucket的静态网站托管模式配置
 *
 * @param $ossClient OssClient
 * @param  $bucket string 存储空间名称
 * @return null
 */
function putBucketWebsite($ossClient, $bucket)
{
    $websiteConfig = new WebsiteConfig("index.html", "error.html");
    try {
        $ossClient->putBucketWebsite($bucket, $websiteConfig);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

获取静态网站托管配置

您可以通过OssClient::getBucketWebsite获取静态网站托管配置：

```
<?php
/**
 * 获取bucket的静态网站托管状态
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket bucket名字
 * @return null
 */
function getBucketWebsite($ossClient, $bucket) {
    $websiteConfig = null;
    try{
        $websiteConfig = $ossClient->getBucketWebsite($bucket);
```

```
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    print($websiteConfig->serializeToXml() . "\n");
}
```

删除静态网站托管配置

您可以通过OssClient::deleteBucketWebsite删除静态网站托管配置：

```
<?php
/**
 * 删除bucket的静态网站托管模式配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket bucket名字
 * @return null
 */
function deleteBucketWebsite($ossClient, $bucket) {
    try{
        $ossClient->deleteBucketWebsite($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

1.2.7.10 生命周期管理

OSS提供文件生命周期管理来为用户管理对象。用户可以为某个存储空间定义生命周期配置，来为该存储空间的文件定义各种规则。目前，用户可以通过规则来删除相匹配的文件。每条规则都由如下几个部分组成：

- 文件名称前缀，只有匹配该前缀的文件才适用这个规则
- 操作，用户希望对匹配的文件所执行的操作。
- 日期或天数，用户期望在特定日期或者是在文件最后修改时间后多少天执行指定的操作。这里不推荐用户直接使用日期作为文件生命周期管理的方式#使用这种方式#在指定日期之后符合前缀的文件会过期#而不论文件的最后修改时间。

说明：

以下场景的完整代码路径：*GitHub*。

Lifecycle规则说明

lifecycle的配置规则由一段xml表示。

```
<?xml version="1.0" encoding="utf-8"?>
<LifecycleConfiguration>
    <Rule>
        <ID>delete obsoleted files</ID>
        <Prefix>obsoleted/</Prefix>
        <Status>Enabled</Status>
        <Expiration><Days>3</Days></Expiration>
    </Rule>
    <Rule>
        <ID>delete temporary files</ID>
        <Prefix>temporary/</Prefix>
        <Status>Enabled</Status>
        <Expiration><Date>2022-10-12T00:00:00.000Z</Date></Expiration>
    </Rule>
</LifecycleConfiguration>
```

一个Lifecycle的Config里面可以包含多个Rule（最多1000个）。

各字段解释：

- ID字段是用来唯一表示本条规则。
- Prefix指定对存储空间下的符合特定前缀的文件使用规则，不能重叠。
- Status指定本条规则的状态，只有Enabled和Disabled，分别表示启用规则和禁用规则。
- Expiration节点里面的Days表示大于文件最后修改时间指定的天数就删除文件，Date则表示到指定的绝对时间之后就删除文件(绝对时间服从ISO8601的格式)。

设置Lifecycle规则

您可以通过OssClient::putBucketLifecycle设置存储空间的lifecycle规则：

```
<?php
/**
 * 设置bucket的生命周期配置
 *
 * @param OssClient $ossClient OssClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putBucketLifecycle($ossClient, $bucket)
{
    $lifecycleConfig = new LifecycleConfig();
    $actions = array();
    $actions[] = new LifecycleAction(OssClient::OSS_LIFECYCLE_EXPIRATION, OssClient::
OSS_LIFECYCLE_TIMING_DAYS, 3);
    $lifecycleRule = new LifecycleRule("delete obsoleted files", "obsoleted/", "Enabled", $actions
);
    $lifecycleConfig->addRule($lifecycleRule);
    $actions = array();
    $actions[] = new LifecycleAction(OssClient::OSS_LIFECYCLE_EXPIRATION, OssClient::
OSS_LIFECYCLE_TIMING_DATE, '2022-10-12T00:00:00.000Z');
```

```
    $lifecycleRule = new LifecycleRule("delete temporary files", "temporary/", "Enabled", $actions
);
    $lifecycleConfig->addRule($lifecycleRule);
    try {
        $ossClient->putBucketLifecycle($bucket, $lifecycleConfig);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

获取Lifecycle规则

您可以通过OssClient::getBucketLifecycle获取存储空间的lifecycle规则。

```
<?php
/**
 * 获取bucket的生命周期配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket bucket名字
 * @return null
 */
function getBucketLifecycle($ossClient, $bucket)
{
    $lifecycleConfig = null;
    try{
        $lifecycleConfig = $ossClient->getBucketLifecycle($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    print($lifecycleConfig->serializeToXml() . "\n");
}
```

删除Lifecycle规则

您可以通过OssClient::deleteBucketLifecycle清空存储空间中lifecycle规则。

```
<?php
/**
 * 删除bucket的生命周期配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket bucket名字
 * @return null
 */
function deleteBucketLifecycle($ossClient, $bucket)
{
    try{
        $ossClient->deleteBucketLifecycle($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
```

```
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

1.2.7.11 设置访问日志

OSS允许用户对Bucket设置访问日志记录，设置之后对于Bucket的访问会被记录成日志，日志存储在OSS上由用户指定的Bucket中，文件的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中TargetPrefix由用户指定。日志规则由以下3项组成：

- enable，是否开启
- target_bucket，存放日志文件的Bucket
- target_prefix，日志文件的前缀

说明：

以下场景的完整代码路径：*GitHub*。

开启Bucket日志

您可以通过OssClient::putBucketLogging开启存储空间的访问日志：

```
<?php
/**
 * 设置bucket的Logging配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function putBucketLogging($ossClient, $bucket)
{
    $option = array();
    //访问日志存放在本bucket下
    $targetBucket = $bucket;
    $targetPrefix = "access.log";
    try {
        $ossClient->putBucketLogging($bucket, $targetBucket, $targetPrefix, $option);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
```

```
}
```

查看Bucket日志设置

您可以通过OssClient::getBucketLogging查看存储空间的日志配置：

```
<?php
/**
 * 获取bucket的Logging配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function getBucketLogging($ossClient, $bucket)
{
    $loggingConfig = null;
    $options = array();
    try {
        $loggingConfig = $ossClient->getBucketLogging($bucket, $options);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    print($loggingConfig->serializeToXml() . "\n");
}
```

关闭Bucket日志

您可以通过OssClient::deleteBucketLogging删除存储空间的日志配置：

```
<?php
/**
 * 删除bucket的Logging配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string $bucket 存储空间名称
 * @return null
 */
function deleteBucketLogging($ossClient, $bucket)
{
    try {
        $ossClient->deleteBucketLogging($bucket);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
```

```
}
```

1.2.7.12 跨域资源共享

跨域资源共享(CORS)允许web端的应用程序访问不属于本域的资源。OSS提供接口方便开发者控制跨域访问的权限。

说明：

以下场景的完整代码路径：*GitHub*。

设定CORS规则

通过putBucketCors 方法将指定的存储空间上设定一个跨域资源共享CORS的规则，如果原规则存在则覆盖原规则。具体的规则主要通过CORSRule类来进行参数设置。

您可以通过OssClient::putBucketCors设置存储空间的CORS规则：

```
<?php
/**
 * 设置bucket的cors配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket 存储空间名称
 * @return null
 */
function putBucketCors($ossClient, $bucket)
{
    $corsConfig = new CorsConfig();
    $rule = new CorsRule();
    $rule->addAllowedHeader("x-oss-header");
    $rule->addAllowedOrigin("http://www.b.com");
    $rule->addAllowedMethod("POST");
    $rule->setMaxAgeSeconds(10);
    $corsConfig->addRule($rule);
    try{
        $ossClient->putBucketCors($bucket, $corsConfig);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

- 每个存储空间最多只能使用10条CorsRule。
- AllowedOrigins和AllowedMethods都能够最多支持一个 * 通配符。* 表示对于所有的域来源或者操作都满足。

- 而AllowedHeaders和ExposeHeaders不支持通配符。

获取CORS规则

您可以通过OssClient::getBucketCors获取存储空间上设置的CORS规则：

```
<?php
/**
 * 获取并打印bucket的cors配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket bucket名字
 * @return null
 */
function getBucketCors($ossClient, $bucket)
{
    $corsConfig = null;
    try{
        $corsConfig = $ossClient->getBucketCors($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    print($corsConfig->serializeToXml() . "\n");
}
```

删除CORS规则

您可以通过OssClient::deleteBucketCors关闭并清除存储空间上所有的CORS规则。

```
<?php
/**
 * 删除bucket的所有的cors配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket bucket名字
 * @return null
 */
function deleteBucketCors($ossClient, $bucket)
{
    try{
        $ossClient->deleteBucketCors($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

1.2.7.13 防盗链

OSS是按使用收费的服务，为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

说明：

以下场景的完整代码路径：*GitHub*。

设置Referer白名单

您可以通过OssClient::putBucketReferer设置Referer白名单：

```
<?php
/**
 * 设置存储空间的防盗链配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket 存储空间名称
 * @return null
 */
function putBucketReferer($ossClient, $bucket)
{
    $refererConfig = new RefererConfig();
    $refererConfig->setAllowEmptyReferer(true);
    $refererConfig->addReferer("www.aliiyun.com");
    $refererConfig->addReferer("www.aliiyuncs.com");
    try{
        $ossClient->putBucketReferer($bucket, $refererConfig);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

说明：

Referer参数支持通配符 * 和 ？。

获取Referer白名单

您可以通过OssClient::getBucketReferer获取Referer白名单：

```
<?php
/**
 * 获取bucket的防盗链配置
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket 存储空间名称
 * @return null
 */
function getBucketReferer($ossClient, $bucket)
{
    $refererConfig = null;
    try{
        $refererConfig = $ossClient->getBucketReferer($bucket);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
```

```
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
    print($refererConfig->serializeToXml() . "\n");
}
```

清空Referer白名单

Referer白名单不能直接清空，只能通过重新设置来覆盖之前的规则。

```
<?php
/**
 * 删除bucket的防盗链配置
 * Referer白名单不能直接清空，只能通过重新设置来覆盖之前的规则。
 *
 * @param OssClient $ossClient OSSClient实例
 * @param string    $bucket 存储空间名称
 * @return null
 */
function deleteBucketReferer($ossClient, $bucket)
{
    $refererConfig = new RefererConfig();
    try{
        $ossClient->putBucketReferer($bucket, $refererConfig);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

1.2.7.14 自定义域名绑定

OSS支持用户将自定义的域名(CNAME)绑定到OSS的Bucket上，这样能够支持用户无缝地将存储迁移到OSS上。例如用户的域名是my-domain.com，之前用户的所有图片资源都是形如 http://img.my-domain.com/x.jpg 的格式，用户将图片存储迁移到OSS之后，通过绑定自定义域名，仍可以使用原来的地址访问到图片。使用步骤如下：

1. 开通OSS服务并创建Bucket。
2. 修改域名的DNS配置，增加一个CNAME记录，将img.my-domain.com指向OSS服务的endpoint（如my-bucket.oss-cn-hangzhou.aliyuncs.com）。
3. 在官网控制台或者使用SDK将img.my-domain.com与创建的Bucket绑定。
4. 将图片上传到OSS的这个Bucket中。

这样就可以通过原地址 http://img.my-domain.com/x.jpg 访问到存储在OSS上的图片。

增加一个CNAME

通过addBucketCname接口为Bucket增加一个CNAME绑定：

```
<?php
use OSS\OssClient;
$client = new OssClient(
    '<Your AccessKeyId>',
    '<Your AccessKeySecret>',
    '<Your Endpoint>');
$client->addBucketCname('bucket name', 'img.my-domain.com');
```

删除一个CNAME

通过deleteBucketCname接口删除一个CNAME绑定：

```
<?php
use OSS\OssClient;
$client = new OssClient(
    '<Your AccessKeyId>',
    '<Your AccessKeySecret>',
    '<Your Endpoint>');
$client->deleteBucketCname('bucket name', 'img.my-domain.com');
```

获取已绑定的CNAME

通过getBucketCname接口获取Bucket已绑定的CNAME列表：

```
<?php
use OSS\OssClient;
$client = new OssClient(
    '<Your AccessKeyId>',
    '<Your AccessKeySecret>',
    '<Your Endpoint>');
$cnameConfig = $client->getBucketCname('bucket name', 'img.my-domain.com');
var_dump($cnameConfig);
```

1.2.7.15 图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处理服务。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

图片处理基础功能

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换

- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的**图片处理** > **样式管理**中定义
- 通过级联处理调用多个图片处理功能

图片处理使用

图片处理使用标准的 HTTP GET 请求来访问，所有的处理参数是编码在 URL 中的QueyString。

匿名访问

如果图片文件（Object）的访问权限是 公共读，如下表所示的权限，则可以匿名访问图片服务。

Bucket权限	Object权限
公共读私有写（public-read）或公共读写（public-read-write）	默认（default）
任意权限	公共读私有写（public-read）或 公共读写（public-read-write）

通过如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value
```

- bucket：用户的存储空间（bucket）名称
- endpoint：用户存储空间所在数据中心的访问域名
- object：用户上传在OSS上的图片文件
- image：图片处理保留标志符
- action：用户对图片做的操作，如缩放、裁剪、旋转等
- parame：用户对图片做的操作所对应的参数

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,
w_100
```

自定义样式，使用如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=x-oss-process=style/name
```

- style：用户自定义样式系统保留标志符

- name：自定义样式名称，即控制台定义样式的 规则名

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=style/oss-pic-style-w-100
```

通过级联处理，可以对一张图片顺序实施多个操作，格式如下：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value/action,parame_value/...
```

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100/rotate,90
```

图片服务也支持HTTPS访问，例如：

```
https://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100
```

授权访问

对私有权限的文件（Object），如下表所示的权限，必须通过授权才能访问图片服务。

Bucket权限	Object权限
私有读写（private）	默认权限（default）
任意权限	私有读写（private）

生成带签名的图片处理的URL代码如下：

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use OSS\OssClient;
$endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
$accessKeyId = "<access_key_id>";
$accessKeySecret = "<access_key_secret>";
$bucket = "<bucket_name>";
$object = "example.jpg";
$ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
/**
 * 生成一个带签名的可用于浏览器直接打开的url, URL的有效期是3600秒
 */
$timeout = 3600;
$options = array(
    OssClient::OSS_PROCESS => "image/resize,m_lfit,h_100,w_100" );
$signedUrl = $ossClient->signUrl($bucket, $object, $timeout, "GET", $options);
```

```
Common::println("rtmp url: \n" . $signedUrl);
```

说明：

- 授权访问支持**自定义样式**、**HTTPS**、**级联处理**。
- **signedUrl** 过期时间单位是**秒**。

SDK访问

对于任意权限的图片文件，都可以直接使用SDK访问图片、进行处理。

说明：

- 图片处理的完整代码请参考：*GitHub*。
- SDK处理图片文件支持**自定义样式**、**HTTPS**、**级联处理**。

基础操作

图片处理的基础操作包括，获取图片信息、格式转换、缩放、裁剪、旋转、效果、水印等。

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use OSS\OssClient;
$endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
$accessKeyId = "<access_key_id>";
$accessKeySecret = "<access_key_secret>";
$bucket = "<bucket_name>";
$object = "example.jpg";
$download_file = "download.jpg";
$ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
// 先把本地的example.jpg上传到指定$bucket, 命名为$object
$ossClient->uploadFile($bucket, $object, "example.jpg");
// 图片缩放
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/resize,m_fixed,h_100,w_100" );
$ossClient->getObject($bucket, $object, $options);
// 图片裁剪
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/crop,w_100,h_100,x_100,y_100,r_1" );
$ossClient->getObject($bucket, $object, $options);
// 图片旋转
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/rotate,90" );
$ossClient->getObject($bucket, $object, $options);
// 图片锐化
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/sharpen,100" );
$ossClient->getObject($bucket, $object, $options);
// 图片水印
```

```
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/watermark,text_SGVsbG8g5Zu-54mH5pyN5YqhIQ
" );
$ossClient->getObject($bucket, $object, $options);
// 图片格式转换
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/format,png" );
$ossClient->getObject($bucket, $object, $options);
// 获取图片信息
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => "image/info" );
$ossClient->getObject($bucket, $object, $options);
//最后删除上传的$object
$ossClient->deleteObject($bucket, $object);
```

自定义样式

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use OSS\OssClient;
$endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
$accessKeyId = "<access_key_id>";
$accessKeySecret = "<access_key_secret>";
$bucket = "<bucket_name>";
$object = "example.jpg";
$download_file = "download.jpg";
$ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
// 先把本地的example.jpg上传到指定$bucket, 命名为$object
$ossClient->uploadFile($bucket, $object, "example.jpg");
$style = "style/oss-pic-style-w-300";
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => $style);
$ossClient->getObject($bucket, $object, $options);
//最后删除上传的$object
$ossClient->deleteObject($bucket, $object);
```

级联处理

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use OSS\OssClient;
$endpoint = "<endpoint, 例如http://oss-cn-hangzhou.aliyuncs.com>";
$accessKeyId = "<access_key_id>";
$accessKeySecret = "<access_key_secret>";
$bucket = "<bucket_name>";
$object = "example.jpg";
$download_file = "download.jpg";
$ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint, false);
// 先把本地的example.jpg上传到指定$bucket, 命名为$object
$ossClient->uploadFile($bucket, $object, "example.jpg");
$style = "image/resize,m_fixed,w_100,h_100/rotate,90";
$options = array(
    OssClient::OSS_FILE_DOWNLOAD => $download_file,
    OssClient::OSS_PROCESS => $style);
$ossClient->getObject($bucket, $object, $options);
```

```
//最后删除上传的$object
$ossClient->deleteObject($bucket, $object);
```

图片处理工具

- 可视化图片处理工具 *ImageStyleViever*，可以直观的看到OSS图片处理的结果。
- OSS图片处理的功能、使用演示页面。

1.2.7.16 异常处理

调用OssClient类的相关接口时，**如果抛出异常，则表明操作失败，否则操作成功。抛出异常时，方法返回的数据无效。**OssClient类的接口异常时会抛出异常类 *OssException* 。

异常处理示例

```
try {
    $ossClient->createBucket($bucket);
} catch (OssException $e) {
    print("Exception:" . $e->getMessage() . "\n");
}
```

OssException

OssException指包括两类：

- 客户端异常，包括参数无效、文件不存在等错误。该类错误可以通过OssException::getMessage()获取错误信息。
- 服务器端异常，指OSS返回的错误，比如无权限、文件不存在等。可以通过OssException::getMessage()获取错误信息。该类异常还包含以下信息：
  - HTTPStatus: HTTP状态码，通过方法getHTTPStatus()获取。
  - ErrorCode： OSS返回给用户的错误码，通过方法getErrorCode()获取。
  - ErrorMessage： OSS提供的错误描述，通过方法getErrorMessage()获取。
  - RequestId： 用于唯一标识该请求的UUID；当您无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助。通过方法getRequestId()获取。
  - Details： OSS返回的错误信息原文。通过方法getDetails()获取。

OSS常见错误码

错误码	描述	HTTP状态码
AccessDenied	拒绝访问	403
BucketAlreadyExists	Bucket已经存在	409

错误码	描述	HTTP状态码
BucketNotEmpty	Bucket不为空	409
EntityTooLarge	实体过大	400
EntityTooSmall	实体过小	400
FileGroupTooLarge	文件组过大	400
FilePartNotExist	文件Part不存在	400
FilePartStale	文件Part过时	400
InvalidArgument	参数格式错误	400
InvalidAccessKeyId	AccessKeyId不存在	403
InvalidBucketName	无效的Bucket名字	400
InvalidDigest	无效的摘要	400
InvalidObjectName	无效的Object名字	400
InvalidPart	无效的Part	400
InvalidPartOrder	无效的part顺序	400
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket	400
InternalError	OSS内部发生错误	500
MalformedXML	XML格式非法	400
MethodNotAllowed	不支持的方法	405
MissingArgument	缺少参数	411
MissingContentLength	缺少内容长度	411
NoSuchBucket	Bucket不存在	404
NoSuchKey	文件不存在	404
NoSuchUpload	Multipart Upload ID不存在	404
NotImplemented	无法处理的方法	501
PreconditionFailed	预处理错误	412
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟	403
RequestTimeout	请求超时	400
SignatureDoesNotMatch	签名错误	403

错误码	描述	HTTP状态码
InvalidEncryptionAlgorithmError	指定的熵编码加密算法错误	400

1.2.8 C-SDK

1.2.8.1 前言

SDK下载

版本3.4.0（2017-02-22）：

- Linux SDK包：*aliyun_oss_c_sdk_v3.4.0.tar.gz*
- Windows SDK包：*aliyun_oss_c_sdk_v3.4.0.zip*
- 源代码：*GitHub*

兼容性

- 3.*.* 相对于 3.0.0兼容
- 3.0.0 相对于 2.*.* 的兼容性
  - Windows兼容
  - Linux接口兼容，链表(aos_list_t)遍历接口不兼容
    - aos_list_for_each_entry
    - aos_list_for_each_entry_reverse
    - aos_list_for_each_entry_safe
    - aos_list_for_each_entry_safe_reverse
- 2.1.0 相对于 2.0.0兼容
- 2.0.0 相对于 1.0.0，以下结构体和接口不兼容，其余都兼容
  - oss_request_options_t
  - oss_get_object_to_buffer
  - oss_get_object_to_file
  - oss_get_object_to_buffer_by_url
  - oss_get_object_to_file_by_url
  - oss_init_multipart_upload
  - oss_complete_multipart_upload
- 1.0.0 相对于 0.0.* 不兼容

简介

本文档主要介绍OSS C SDK的安装和使用。本文档假设您已经开通了阿里云OSS 服务，并创建了AccessKeyId 和AccessKeySecret。如果您还没有开通或者还不了解OSS，请登录*OSS产品主页*获取更多的帮助。

1.2.8.2 安装

SDK安装要求

- 开通阿里云OSS服务，并创建了AccessKeyId 和AccessKeySecret。
- 如果您还没有开通或者还不了解阿里云OSS服务，请登录*OSS产品主页*了解。
- 如果还没有创建AccessKeyId和AccessKeySecret，请到阿里云Access Key管理创建Access Key。

Linux

环境依赖

OSS C SDK使用curl进行网络操作，无论是作为客户端还是服务器端，都需要依赖curl。另外，OSS C SDK使用apr、apr-util库解决内存管理以及跨平台问题，使用minixml库解析请求返回的xml，OSS C SDK(Linux)并没有带上这几个外部库，您需要确认这些库已经安装，并且将它们的头文件目录和库文件目录都加入到了项目中。

安装第三方库

Ubuntu/Debian

- 安装CMake

  执行以下命令安装CMake。

  ```
  sudo apt-get install cmake
  ```

- 安装第三方库

  执行以下命令安装第三方库。

  ```
  sudo apt-get install libcurl4-openssl-dev libapr1-dev libaprutil1-dev libmxml-dev
  ```

RedHat/Aliyun/CentOS

- 安装CMake

执行以下命令安装CMake。

```
sudo yum install cmake
```

- 安装第三方库

  执行以下命令安装第三方库。

```
sudo yum install curl-devel apr-devel apr-util-devel
```

- minixml

  请下载系统对应的rpm包：

  - 64位 *mxml-2.10-1.x86_64.rpm*
  - 32位 *mxml-2.10-1.i386.rpm*

  执行以下命令安装minixml：

```
rpm -ivh mxml-2.10-1.x86_64.rpm
```

SuSE

- 安装CMake

  执行以下命令安装CMake。

```
zypper install cmake
```

- 安装第三方库

  执行以下命令安装第三方库。

```
zypper install libcurl-devel libapr1-devel libapr-util1-devel mxml-devel
```

其他Linux

- CMake (建议2.6.0及以上版本)

  请从这里下载，典型的安装方式如下：

```
./configure
make
make install
```

> **说明**：
> 执行./configure时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure —prefix=/your/install/path/。

- libcurl （建议 7.32.0 及以上版本）

  请从这里下载，并参考*libcurl 安装指南*安装。典型的安装方式如下：

  ```
  ./configure
  make
  make install
  ```

  说明：

  - 执行./configure时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure —prefix=/your/install/path/。
  - 如果要使用MEDIA-C-SDK，请确保./configure执行完后，最后一行的Protocols里面包含HTTPS，如果没有，请先安装openssl-devel等ssl开发包，然后重新安装libcurl。

- apr （建议 1.5.2 及以上版本）

  请从这里下载，典型的安装方式如下：

  ```
  ./configure
  make
  make install
  ```

  说明：

  执行./configure时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure —prefix=/your/install/path/。

- apr-util （建议 1.5.4 及以上版本）

  请从这里下载，安装时需要注意指定—with-apr选项，典型的安装方式如下：

  ```
  ./configure --with-apr=/your/apr/install/path
  make
  make install
  ```

  说明：

  - 执行./configure时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure —prefix=/your/install/path/
  - 需要通过—with-apr指定apr安装目录，如果apr安装到系统目录下需要指定—with-apr=/usr/local/apr/

- minixml （建议 2.8 及以上版本）

请从这里下载，典型的安装方式如下：

```
./configure
make
sudo make install
```

说明：

执行./configure时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure —prefix=/your/install/path/。

编译安装OSS C SDK

典型的编译命令如下：

```
cmake .
make
make install
```

说明：

- 执行cmake . 时默认会到/usr/local/下面去寻找curl，apr，apr-util，mxml的头文件和库文件。
- 默认编译是Debug类型，可以指定以下几种编译类型： Debug, Release, RelWithDebInfo和MinSizeRel，如果要使用release类型编译，则执行cmake -f CMakeLists.txt -DCMAKE_BUILD_TYPE=Release
- 如果您在安装curl，apr，apr-util，mxml时指定了安装目录，则需要在执行cmake时指定这些库的路径，
  - 比如：cmake -f CMakeLists.txt -DCURL_INCLUDE_DIR=/usr/local/include/curl/ -DCURL_LIBRARY=/usr/local/lib/libcurl.so -DAPR_INCLUDE_DIR=/usr/local/include/apr-1/ -DAPR_LIBRARY=/usr/local/lib/libapr-1.so -DAPR_UTIL_INCLUDE_DIR=/usr/local/apr/include/apr-1 -DAPR_UTIL_LIBRARY=/usr/local/apr/lib/libaprutil-1.so -DMINIXML_INCLUDE_DIR=/usr/local/include -DMINIXML_LIBRARY=/usr/local/lib/libmxml.so
- 如果要指定安装目录，则需要在cmake时增加： -DCMAKE_INSTALL_PREFIX=/your/install/path/usr/local/
- 如果执行cmake时报以下错误：Could not find apr-config/apr-1-config，原因是在默认路径里面找不到apr-1-config文件，这时候可以在执行cmake命令时，在最后面加上-DAPR_CONFIG_BIN=/path/to/bin/apr-1-config。如果报：Could not find apu-config/apu-1-config，则需要加上-DAPU_CONFIG_BIN=/path/to/bin/apu-1-config。

Linux示例工程

基于Aliyun OSS C SDK的示例工程 *aliyun-oss-c-sdk-demo.tar.gz* 。其中，示例oss-c-sdk-demo-specified-installation基于OSS C SDK及依赖的第三方库，都安装在*/home/your/oss/csdk*下；其他示例工程是基于OSS C SDK及依赖的第三方库，安装在默认目录下，即安装时不指定目录。

- oss-c-sdk-demo-centos Aliyun/CentOS下，基于默认安装路径的OSS C SDK示例工程
- oss-c-sdk-demo-debian Debian下，基于默认安装路径的OSS C SDK示例工程
- oss-c-sdk-demo-redhat RedHat下，基于默认安装路径的OSS C SDK示例工程
- oss-c-sdk-demo-suse SuSE下，基于默认安装路径的OSS C SDK示例工程
- oss-c-sdk-demo-ubuntu Ubuntu下，基于默认安装路径的OSS C SDK示例工程
- oss-c-sdk-demo-specified-installation Linux下，基于自定义安装目录的OSS C SDK示例工程

示例工程的详细说明，请参看 *Linux下使用Aliyun OSS C SDK* 。

解压后进入工程目录(oss-c-sdk-demo-xxx)，执行make，编译示例工程；执行make clean清理编译产生的文件；执行./main运行可执行程序。

说明：

- 示例代码中的OSS_ENDPOINT、ACCESS_KEY_ID、ACCESS_KEY_SECRET、BUCKET_NAME请更换成有效值；
- 如果OSS C SDK及依赖库的动态库不在系统目录下，执行时请使用LD_LIBRARY_PATH指定。

Windows

环境与依赖

- OSS C SDK(Windows)依赖的第三方库和Linux版本一致，分别是apr，apr-util，curl，mxml。
- SDK提供了Visual Studio 2008和Visual Studio 2010的项目工程，分别支持Visual Studio 2008，Visual Studio 2010及其以后版本。
- SDK提供了third_party，包含了需要用到的第三方库(apr, apr-util, curl, mxml)的头文件和库文件，用户可以使用这里的头文件和库文件编译，运行sample和test项目。

Visual Studio 2008版本

- 对应的项目文件为oss_c_sdk_2008.sln等包含2008的文件。
- OSS C SDK需要用到stdint.h头文件，Visual C++ 2008默认是没有此头文件，如果用户没有提前安装此头文件，可以将third_party/include中的stdint.h.bak重命名为stdint.h后使用。

- 从2.0.0版本才开始支持Visual Studio 2008，之前版本均不支持。

Visual Studio 2010及其以后版本

- 对应的项目文件为oss_c_sdk.sln等不包含数字的文件。
- Visual Studio 2010及其以后版本都包含了stdint.h头文件。
- 如果用户使用Visual Studio 2012及其以后版本打开时，会提示用户是否将项目升级成使用最新版的编译器和库，这里最好和用户自己的项目保持一致：如果用户的项目使用了最新版本的编译器和库，就选择升级，否则可以不升级。

Visual Studio示例工程

示例工程提供了基于Visual Studio 2008/2010/2012/2013/2015的工程，您可以用Visual Studio直接打开对应工程， 修改Endpoint/AccessKeyID/AccessKeySecret/BucketName等配置后，既可编译运行。您也可以在示例工程基础上开发您的项目。

- 示例工程：*aliyun-oss-c-sdk-sample.zip*
- GitHub地址：*GitHub*

说明：

- Windows环境下，**目前只支持x86**，**暂不支持x64**。
- 运行oss-c-sdk-sample前，请在Visual Studio做如下配置。在Sulution Explore中选择工程oss-c-sdk-sample，右击选择**Property**，在oss-c-sdk-sample Property Pages中配置**Configuration Properties** > **Debugging** > **Environment**为 **PATH=..\oss-c-sdk\lib\Release\;%PATH%** 。

使用Visual Studio编译OSS C SDK、运行示例程序的详细步骤及常见问题，请参考 *Windows下编译使用Aliyun OSS C SDK*。

示例工程

C SDK提供丰富的示例程序，方便用户参考或直接使用。您可以从*GitHub*获取示例程序。示例程序包括以下内容：

示例文件	示例内容
oss_put_object_sample	展示了文件上传的用法，包括从内存/文件上传、携带MD5校验上传、携带元数据上传、url签名上传等
oss_get_object_sample.c	展示了文件下载的用法，包括下载到内存、下载到文件、范围下载、URL签名下载等

示例文件	示例内容
oss_append_object_sample.c	展示了追加上传的用法，包括从内存追加上传、从文件追加上传
oss_multipart_upload_sample.c	展示了分片上传的用法，包括从内存分片上传、从文件分片上传、取消上传分片上传等
oss_resumable_sample.c	展示了并发断点续传上传的用法
oss_head_object_sample.c	展示了获取文件元信息的用法
oss_delete_object_sample.c	展示了删除文件的用法，包括删除单个文件、删除多个文件
oss_callback_sample.c	展示了上传回调的用法，包括上传回调、分片上传回调
oss_progress_sample.c	展示了进度条的用法，包括上传、追加上传、分片上传、下载等
oss_crc_sample.c	展示了上传、下载使用CRC校验的用法
oss_image_sample.c	展示了图片处理的用法

1.2.8.3 初始化

确定Endpoint

Endpoint是阿里云OSS服务在各个区域的地址，目前支持两种形式。

Endpoint类型	解释
OSS区域地址	使用OSS Bucket所在区域地址
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS区域地址

使用OSS Bucket所在区域地址，Endpoint查询可以有下面两种方式：

- 您可以登录阿里云OSS控制台，进入Bucket概览页，Bucket域名的后缀部分：如bucket-1.oss-cn-hangzhou.aliyuncs.com的oss-cn-hangzhou.aliyuncs.com部分为该Bucket的外网Endpoint。

CNAME

您可以将自己拥有的域名通过CNAME绑定到某个存储空间（Bucket）上，然后通过自己域名访问存储空间内的文件。

比如您要将域名new-image.xxxxx.com绑定到深圳区域的名称为image的存储空间上：

您需要到您的域名xxxxx.com托管商那里设定一个新的域名解析，将 *http://new-image.xxxxx.com* 解析到 *http://image.oss-cn-shenzhen.aliyuncs.com*，类型为CNAME。

配置密钥

要接入阿里云OSS，您需要拥有一对有效的 AccessKey(包括AccessKeyId和AccessKeySecret)用来进行签名认证。可以通过如下步骤获得：

- 注册阿里云账号
- 申请*AccessKey*

在获取到 AccessKeyId和 AccessKeySecret之后，您可以按照下面步骤进行初始化。

初始化请求选项

使用OSS C SDK时需要初始化请求选项(oss_request_options_t)，其中config用于存储访问OSS的基本信息，比如OSS域名、用户的AccessKeyId 、用户的AccessKeySecret。is_cname变量指定访问OSS是否使用CNAME。 ctl用于设置访问OSS时的控制信息。

```
void init_options(oss_request_options_t *options) {
    options->config = oss_config_create(options->pool);
    aos_str_set(&options->config->endpoint, "<您的Endpoint>");
    aos_str_set(&options->config->access_key_id, "<您的AccessKeyId>");
    aos_str_set(&options->config->access_key_secret, "<您的AccessKeySecret>");
    options->config->is_cname = 0;
    options->ctl = aos_http_controller_create(options->pool, 0);
}
int main() {
    aos_pool_t *p;
    oss_request_options_t *options;
    /* 全局变量初始化，应该放在程序启动时，其他所有逻辑之前 */
    if (aos_http_io_initialize(NULL, 0) != AOSE_OK) {
        return -1;
    }
    /* 初始化内存池和options */
    aos_pool_create(&p, NULL);
    options = oss_request_options_create(p);
    init_options(options);
    /* 逻辑代码 */
    /* 释放内存pool资源，包括了通过pool分配的内存，比如options等*/
    aos_pool_destroy(p);
    /* 释放全局资源，应该放在程序结束前 */
    aos_http_io_deinitialize();
    return 0;
}
```

说明：

- 如果想使用https，只需要设置endpoint时，前缀使用https://即可。
- 支持多线程，但是aos_http_io_initialize和aos_http_io_deinitialize只需要在主线程里面调用即可，其他线程不需要调用。

- 如果想设置oss c sdk底层libcurl通信时的一些参数，可以对请求选项中的ctl进行设置，实现控制诸如数据上传最低速度、连接超时时间、DNS缓存失效时间等目的。
- 您也可以通过ctl获得使用oss c sdk访问OSS的性能参数。以上传数据为例，用户在上传完数据后可以得到一系列的参数指标，比如开始上传数据的时间start_time，第一字节上传的时间first_byte_time，完成数据上传的时间finish_time。
- 关于如何设置请求选项，请详见*oss c sdk如何设置通信时和CURL相关的一些参数*。

1.2.8.4 快速入门

在这一章里，您将学到如何用OSS C SDK完成一些基本的操作。

Step-1.初始化OSS C SDK运行环境

OSS C SDK使用时首先需要初始化运行环境，使用结束前需要清理运行环境，下面代码演示初始化OSS C SDK运行环境：

```
int main(int argc, char *argv[])
{
    /* 程序入口处调用aos_http_io_initialize方法，这个方法内部会做一些全局资源的初始化，涉及网络，内存等部分 */
    if (aos_http_io_initialize(NULL, 0) != AOSE_OK) {
        exit(1);
    }

    /* 调用OSS SDK的接口上传或下载文件 */
    /* ... 用户逻辑代码，这里省略 */

    /* 程序结束前，调用aos_http_io_deinitialize方法释放之前分配的全局资源 */
    aos_http_io_deinitialize();
    return 0;
}
```

说明：

- aos_http_io_initialize初始化OSS C SDK运行环境，第一个参数可以用于个性化设置user agent的内容，用作后续统计。
- aos_http_io_deinitialize清理OSS C SDK运行环境。

Step-2.初始化请求选项

OSS C SDK的所有操作需要初始化请求选项，下面代码完成初始化请求选项：

```
    /* 等价于apr_pool_t，用于内存管理的内存池，实现代码在apr库中 */
    aos_pool_t *pool;
    oss_request_options_t *options;

    /* 重新创建一个新的内存池，第二个参数是NULL，表示没有继承自其他内存池 */
    aos_pool_create(&pool, NULL);
```

```
    /* 创建并初始化options，这个参数内部主要包括endpoint,access_key_id,acces_key_secret，
is_cname, curl参数等全局配置信息
     * options的内存是由pool分配的，后续释放掉pool后，options的内存也相当于释放掉了，不再
需要单独释放内存
     */
    options = oss_request_options_create(pool);
    options->config = oss_config_create(options->pool);

    /* aos_str_set是用char*类型的字符串初始化aos_string_t类型*/
    aos_str_set(&options->config->endpoint, "<您的Endpoint>");
    aos_str_set(&options->config->access_key_id, "<您的AccessKeyId>");
    aos_str_set(&options->config->access_key_secret, "<您的AccessKeySecret>");

    /* 是否使用了CNAME */
    options->config->is_cname = 0;

    /* 用于设置网络相关参数，比如超时时间等*/
    options->ctl = aos_http_controller_create(options->pool, 0);
```

Step-3. 新建存储空间(Bucket)

您可以按照下面的代码新建一个存储空间：

```
    aos_pool_t *p;
    oss_request_options_t *options;
    aos_status_t *s;
    aos_table_t *resp_headers;
    oss_acl_e oss_acl = OSS_ACL_PRIVATE;
    char *bucket_name = "<您的bucket名字>";
    aos_string_t bucket;

    aos_pool_create(&p, NULL);

    options = oss_request_options_create(p);
    init_options(options);

    /* 将char*类型数据赋值给aos_string_t类型的bucket */
    aos_str_set(&bucket, bucket_name);
    s = oss_create_bucket(options, &bucket, oss_acl, &resp_headers);

    /* 判断请求是否成功 */
    if (aos_status_is_ok(s)) {
        printf("create bucket succeeded\n");
    } else {
        printf("create bucket failed\n");
    }

    /* 执行完一个请求后，释放掉这个内存池，结果就是会释放掉这个请求过程中各个部分分配的
内存 */
    aos_pool_destroy(p);
```

说明：

- Bucket名字不能与OSS服务中其他用户已有的存储空间重复，所以你需要选择一个独特的存储空间名字以避免创建失败。

- oss_create_bucket的返回值是aos_status_t类型，包括了code(http code)，error_code，error_msg和req_id，req_id可以协助调查问题。其他接口，如果没有特殊说明外，也都是返回aos_status_t类型。

Step-4. 上传文件

文件是OSS中最基本的数据单元，用下面代码可以实现一个文件的上传：

```
    aos_pool_t *p;
    oss_request_options_t *options;
    aos_status_t *s;
    aos_table_t *headers;
    aos_table_t *resp_headers;
    char *bucket_name = "<您的bucket名字>";
    char *object_name = "<您的object名字>";
    aos_string_t bucket;
    aos_string_t object;
    char *data = "object content";
    aos_list_t buffer;
    aos_buf_t *content;

    aos_pool_create(&p, NULL);

    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);

    /* 初始化参数 */
    aos_str_set(&object, object_name);
    aos_str_set(&bucket, bucket_name);
    headers = aos_table_make(p, 0);

    /* 将char*类型的数据转换为oss_put_object_from_buffer接口需要的aos_list_t类型的 */
    aos_list_init(&buffer);
    content = aos_buf_pack(options->pool, data, strlen(data));
    aos_list_add_tail(&content->node, &buffer);

    /* 上传文件 */
    s = oss_put_object_from_buffer(options, &bucket, &object, &buffer, headers, &resp_headers
);

    /* 判断请求是否成功 */
    if (aos_status_is_ok(s)) {
        printf("put file succeeded\n");
    } else {
        printf("put file failed\n");
    }

    /* 释放资源 */
    aos_pool_destroy(p);
```

Step-5. 列出存储空间中的所有文件

当您完成一系列上传后，可能需要查看某个存储空间中有哪些文件，可以通过下面的程序实现：

```
    aos_pool_t *p;
```

```
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
oss_list_object_params_t *params;
oss_list_object_content_t *content;
int max_ret = 1000;
char *key;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
params = oss_create_list_object_params(p);
params->max_ret = max_ret;
aos_str_set(&params->prefix, "<prefix>");
aos_str_set(&params->delimiter, "<delimiter>");
aos_str_set(&params->marker, "<marker>");

s = oss_list_object(options, &bucket, params, &resp_headers);

/* 判断请求是否成功 */
if (aos_status_is_ok(s)) {
    printf("list file succeeded\n");
} else {
    printf("list file failed\n");
}

/* 获取每个文件的名称  */
aos_list_for_each_entry(content, &params->object_list, node) {
    key = apr_psprintf(p, "%.*s", content->key.len, content->key.data);
}

/* 释放资源 */
aos_pool_destroy(p);
```

Step-6. 下载指定文件

您可以参考下面的代码简单地实现下载指定文件：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
aos_list_t buffer;
aos_buf_t *content;
char *buf;
int64_t len = 0;
int64_t size = 0;
int64_t pos = 0;
```

```
aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&object, object_name);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 1);

/* 下载文件到buffer中 */
aos_list_init(&buffer);
s = oss_get_object_to_buffer(options, &bucket, &object, headers, &buffer, &resp_headers);

/* 判断请求是否成功 */
if (aos_status_is_ok(s)) {
    printf("get file succeeded\n");
} else {
    printf("get file failed\n");
}

/* 从buffer中将aos_list_t类型的数据转为char*类型的，并计算读到的文件总长度 */
len = aos_buf_list_len(&buffer);
buf = aos_pcalloc(p, len + 1);
buf[len] = '\0';
aos_list_for_each_entry(content, &buffer, node) {
    size = aos_buf_size(content);
        memcpy(buf + pos, content->pos, size);
        pos += size;
}

/* 释放资源 */
aos_pool_destroy(p);
```

1.2.8.5 管理Bucket

Bucket是OSS上的存储空间，也是计费、权限控制、日志记录等高级功能的管理实体；存储空间名称在整个OSS服务中具有全局唯一性，且不能修改；存储在OSS上的每个文件必须都包含在某个存储空间中。

新建存储空间

通过oss_create_bucket接口，可以实现创建一个存储空间，用户需要指定存储空间的名字：

```
aos_pool_t *p;
oss_request_options_t *options;
oss_acl_e oss_acl = OSS_ACL_PRIVATE;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
aos_table_t *resp_headers;
aos_status_t *s;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
```

```
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);

/* 创建存储空间 */
s = oss_create_bucket(options, &bucket, oss_acl, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("create bucket succeeded\n");
} else {
    printf("create bucket failed\n");
}

aos_pool_destroy(p);
```

说明：

- 由于存储空间的名字是全局唯一的，所以必须保证您的存储空间名字不与别人的重复。

删除存储空间

通过oss_delete_bucket接口，可以实现删除一个存储空间，用户需要指定存储空间的名字：

```
aos_pool_t *p;
oss_request_options_t *options;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
aos_table_t *resp_headers;
aos_status_t *s;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);

/* 删除存储空间 */
s = oss_delete_bucket (options, &bucket, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("delete bucket succeeded\n");
} else {
    printf("delete bucket failed\n");
}

aos_pool_destroy(p);
```

说明：

- 如果该存储空间下还有文件存在，则需要先删除所有文件才能删除存储空间。
- 如果该存储空间下还有未完成的上传请求，则需要通过 oss_list_multipart_upload 和 oss_abort_multipart_upload 先取消那些请求才能删除存储空间。

存储空间访问权限

用户可以设置存储空间的访问权限，允许或者禁止匿名用户对其内容进行读写。

获取存储空间的访问权限（ACL）

通过oss_get_bucket_acl接口，可以实现查看存储空间的ACL：

```
aos_pool_t *p;
oss_request_options_t *options;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
aos_table_t *resp_headers;
aos_status_t *s;
char *oss_acl;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);

/* 获取存储空间访问权限 */
s = oss_get_bucket_acl(options, &bucket,& oss_acl, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get bucket acl succeeded\n");
} else {
    printf("get bucket acl failed\n");
}

aos_pool_destroy(p);
```

设置存储空间的访问权限（ACL）

通过oss_create_bucket接口，可以实现设置存储空间的ACL：

```
aos_pool_t *p;
oss_request_options_t *options;
oss_acl_e oss_acl = OSS_ACL_PRIVATE;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
aos_table_t *resp_headers;
aos_status_t *s;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);

/* 设置存储空间访问权限 */
s = oss_put_bucket_acl(options, &bucket, oss_acl, &resp_headers);
```

```
if (aos_status_is_ok(s)) {
    printf("put bucket acl succeeded\n");
} else {
    printf("put bucket acl failed\n");
}

aos_pool_destroy(p);
```

说明：

- 操作者必须是Bucket的拥有者，否则不允许设置该存储空间的访问权限。
- 存储空间的访问权限oss_acl_e是一个枚举值，可选值包括：OSS_ACL_PRIVATE、OSS_ACL_PUBLIC_READ、OSS_ACL_PUBLIC_READ_WRITE。

1.2.8.6 上传文件

在OSS中，用户操作的基本数据单元是文件(Object)。单个文件的最大允许大小根据上传数据方式不同而不同，Put Object方式文件最大不能超过5GB，使用分片上传方式文件大小不能超过48.8TB。

OSS C SDK提供了丰富的文件上传接口，用户可以通过以下方式向OSS中上传文件：

- 简单上传
- 追加上传
- 断点续传
- 分片上传

简单上传

从内存中上传数据到OSS

通过oss_put_object_from_buffer接口，可以实现从内存中上传数据到OSS：

```
void put_object_from_buffer()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_table_t *headers = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    aos_list_t buffer;
    aos_buf_t *content = NULL;
    char *str = "test oss c sdk";
    aos_status_t *s = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    aos_str_set(&bucket, "<您的bucket名字>");
```

```
    aos_str_set(&object, "<您的object名字>");
    /* 初始化参数 */
    aos_list_init(&buffer);
    content = aos_buf_pack(options->pool, str, strlen(str));
    aos_list_add_tail(&content->node, &buffer);
    /* 上传文件 */
    s = oss_put_object_from_buffer(options, &bucket, &object,
                &buffer, headers, &resp_headers);
    /* 判断是否上传成功 */
    if (aos_status_is_ok(s)) {
        printf("put object from buffer succeeded\n");
    } else {
        printf("put object from buffer failed\n");
    }
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

说明：

完整代码参考：*GitHub*

上传本地文件到OSS

通过oss_put_object_from_file接口，并指定**fliepath**参数，可以实现上传一个本地文件到OSS：

```
void put_object_from_file()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_table_t *headers = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    char *filename = __FILE__;
    aos_status_t *s = NULL;
    aos_string_t file;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    headers = aos_table_make(options->pool, 1);
    apr_table_set(headers, OSS_CONTENT_TYPE, "image/jpeg");
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    aos_str_set(&file, filename);
    /* 上传文件 */
    s = oss_put_object_from_file(options, &bucket, &object, &file,
                            headers, &resp_headers);
    /* 判断是否上传成功 */
    if (aos_status_is_ok(s)) {
        printf("put object from file succeeded\n");
    } else {
        printf("put object from file failed\n");
    }
    /* 释放资源*/
    aos_pool_destroy(p);
```

```
}
```

说明：

- 使用该方式上传最大文件不能超过5G。如果超过可以使用分片上传。
- 完整代码参考：*GitHub*

追加上传

OSS支持可追加的文件类型，调用时需要指定文件追加的位置，对于新创建文件，这个位置是0；对于已经存在的文件，这个位置必须是追加前文件的长度。

- 文件不存在时，调用Append Object会创建一个可追加的文件
- 文件存在时，调用Append Object会向文件末尾追加内容

Append Object以追加写的方式上传文件，通过Append Object操作创建的文件类型为Appendable Object，而通过Put Object上传的文件类型是Normal Object。

从内存中追加数据到OSS

通过oss_append_object_from_buffer接口，可以实现从内存中上传数据到OSS：

```
void append_object_from_buffer()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    char *str = "test oss c sdk";
    aos_status_t *s = NULL;
    int64_t position = 0;
    aos_table_t *headers1 = NULL;
    aos_table_t *headers2 = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    aos_list_t buffer;
    aos_buf_t *content = NULL;
    char *next_append_position = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    headers1 = aos_table_make(p, 0);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    /* 获取起始追加位置 */
    s = oss_head_object(options, &bucket, &object, headers1, &resp_headers);
    if (aos_status_is_ok(s)) {
        next_append_position = (char*)(apr_table_get(resp_headers,
                    "x-oss-next-append-position"));
        position = atoi(next_append_position);
    }
    /* 追加文件 */
```

```
    headers2 = aos_table_make(p, 0);
    aos_list_init(&buffer);
    content = aos_buf_pack(p, str, strlen(str));
    aos_list_add_tail(&content->node, &buffer);
    s = oss_append_object_from_buffer(options, &bucket, &object,
            position, &buffer, headers2, &resp_headers);
    /* 判断是否追加成功 */
    if (aos_status_is_ok(s))
    {
        printf("append object from buffer succeeded\n");
    } else {
        printf("append object from buffer failed\n");
    }
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

说明：

完整代码参考：*GitHub*

从本地文件追加数据到OSS

通过oss_append_object_from_file接口，并指定**fliepath**参数， 可以实现将一个本地文件的数据追加到OSS：

```
void append_object_from_file()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_table_t *headers1 = NULL;
    aos_table_t *headers2 = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    char *filename = __FILE__;
    aos_status_t *s = NULL;
    aos_string_t file;
    int64_t position = 0;
    char *next_append_position = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    headers1 = aos_table_make(options->pool, 0);
    headers2 = aos_table_make(options->pool, 0);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    aos_str_set(&file, filename);
    /* 获取起始追加位置 */
    s = oss_head_object(options, &bucket, &object, headers1, &resp_headers);
    if(aos_status_is_ok(s)) {
        next_append_position = (char*)(apr_table_get(resp_headers,
                    "x-oss-next-append-position"));
        position = atoi(next_append_position);
    }
    /* 追加文件 */
```

```
    s = oss_append_object_from_file(options, &bucket, &object,
                                    position, &file, headers2, &resp_headers);
    /* 判断是否追加成功 */
    if (aos_status_is_ok(s)) {
        printf("append object from file succeeded\n");
    } else {
        printf("append object from file failed\n");
    }
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

说明：

- 不能对一个非Appendable Object进行Append Object操作。例如，已经存在一个同名Normal Object时，Append Object调用返回409，错误码ObjectNotAppendable。
- 对一个已经存在的Appendable Object进行Put Object操作，那么该Appendable Object会被新的Object覆盖，类型变为Normal Object。
- Head Object操作会返回x-oss-object-type，用于表明Object的类型。对于Appendable Object来说，该值为Appendable。对Appendable Object，Head Object也会返回上述的x-oss-next-append-position和x-oss-hash-crc64ecma。
- 不能使用Copy Object来拷贝一个Appendable Object，也不能改变它的服务器端加密的属性。可以使用Copy Object来改变用户自定义元信息。
- 完整代码参考：*GitHub*

断点续传

大文件上传时，网络不稳定或者其他异常发生，则整个上传就失败了。用户不得不重新上传，造成浪费资源；在网络不稳定的情况下，往往要重试多次。断点续传上传，支持并发上传、断点续传。断点续传通过 oss_resumable_clt_params_t 控制断点续传的行为，它有以下参数：

- part_size：分片大小，从100KB到5GB，单位是***byte***,
- thread_num：并发线程数，默认为***1***
- enable_checkpoint：是否开启断点续传，**默认不开启**
- checkpoint_path：*checkpoint* 文件的路径，默认放在上传文件目录下 *{upload_file_path}.cp*

断点续传实现的原理是，将要上传文件分成若干个分片分别上传，所有分片都上传成功后，完成整个文件的上传。在上传的过程中会记录当前上传的进度信息（记录在*checkpoint*文件中），如果上

传过程中某一分片上传失败，再次上传时会从*checkpoint*文件中记录的点继续上传。这要求再次调用时要指定与上次相同的*checkpoint*文件。上传完成后，*checkpoint*文件会被删除。

```
void resumable_upload()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_string_t filename;
    aos_status_t *s = NULL;
    int is_cname = 0;
    aos_table_t *headers = NULL;
    aos_table_t *resp_headers = NULL;
    aos_list_t resp_body;
    oss_request_options_t *options = NULL;
    oss_resumable_clt_params_t *clt_params;
    aos_pool_create(&p, NULL);
    options = oss_request_options_create(p);
    init_sample_request_options(options, is_cname);
    headers = aos_table_make(p, 0);
    aos_str_set(&bucket, BUCKET_NAME);
    aos_str_set(&object, "my_key.zip");
    aos_str_set(&filename, "local_big_file.zip");
    aos_list_init(&resp_body);
    // 断点续传
    clt_params = oss_create_resumable_clt_params_content(p, 1024 * 100, 3, AOS_TRUE,
NULL);
    s = oss_resumable_upload_file(options, &bucket, &object, &filename, headers, NULL,
        clt_params, NULL, &resp_headers, &resp_body);
    if (aos_status_is_ok(s)) {
        printf("upload succeeded\n");
    } else {
        printf("upload failed\n");
    }
    aos_pool_destroy(p);
}
```

说明：

完整代码参考：*GitHub*

分片上传

除了通过简单上传将文件上传到OSS以外，OSS还提供了另外一种上传模式：分片上传。用户可以在如下的应用场景内（但不仅限于此），使用分片上传模式，如：

- 需要支持断点上传。
- 上传超过100MB大小的文件。
- 网络条件较差，和OSS的服务器之间的连接经常断开。
- 需要流式地上传文件。上传文件之前，无法确定上传文件的大小。

易用接口完成分片上传

为了方便用户完成分片上传，下面代码通过封装后的易用接口oss_upload_file进行分片上传文件操作。

```
void test_upload_file()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    oss_request_options_t *options = NULL;
    aos_status_t *s = NULL;
    int part_size = 100 * 1024;
    aos_string_t upload_id;
    aos_string_t filepath;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    aos_str_null(&upload_id);
    aos_str_set(&filepath, __FILE__);
    /* 分片上传 */
    s = oss_upload_file(options, &bucket, &object, &upload_id, &filepath,
                        part_size, NULL);
    /* 判断是否上传成功 */
    if (aos_status_is_ok(s)) {
        printf("upload file succeeded\n");
    } else {
        printf("upload file failed\n");
    }
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

说明：

- 使用oss_upload_file进行分片上传文件操作，如果是一个新的分片上传操作，设置upload_id为NULL，可以通过aos_str_null(&upload_id)实现；如果是对一个已经存在的upload_id进行续传操作，设置upload_id为指定的upload_id_str，可以通过aos_str_set(&upload_id, upload_id_str)实现。
- 使用oss_upload_file进行分片上传文件操作时，需要指定分片大小， 分片大小要大于100KB。分片号码的范围是1~10000，如果在切分文件时，发现分片号码的范围超出这个范围，oss_upload_file将自动调整分片大小。
- 使用oss_upload_file进行分片上传文件操作，如果指定的upload_id已经存在，那么本次分片上传的分片大小要和指定upload_id的分片大小保持一致。
- 完整代码参考：*GitHub*

分步完成分片上传

分步完成分片上传的好处是灵活，一般的流程如下：

1. 初始化一个分片上传任务（oss_init_multipart_upload）
2. 逐个或并行上传分片（oss_upload_part_from_file）
3. 完成上传（oss_complete_multipart_upload）

初始化

初始化一个分片上传事件。

```
void init_multipart_upload()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_table_t *headers = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    aos_string_t upload_id;
    oss_upload_file_t *upload_file = NULL;
    aos_status_t *s = NULL;
    oss_list_upload_part_params_t *params = NULL;
    aos_list_t complete_part_list;
    oss_list_part_content_t *part_content = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    headers = aos_table_make(p, 1);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    /* 初始化分片上传，获取一个上传ID */
    s = oss_init_multipart_upload(options, &bucket, &object,
                                  &upload_id, headers, &resp_headers);
    /* 判断是否初始化分片上传成功 */
    if (aos_status_is_ok(s)) {
        printf("Init multipart upload succeeded, upload_id:%.*s\n",
               upload_id.len, upload_id.data);
    } else {
        printf("Init multipart upload failed, upload_id:%.*s\n",
               upload_id.len, upload_id.data);
    }
    /* 上传每一个分片，代码参考下一节，这里省略*/
    /* 完成分片上传，代码参考后面章节，这里省略*/
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

说明：

- 返回结果中含有uplolad_id ，它是区分分片上传事件的唯一标识，在后面的操作中，我们将用到它。

- 上述代码只是演示如何初始化分片上传，只是部分代码，完整的代码参考下面的GitHub链接。
- 完整代码参考：*GitHub*
- 1.0.0版本中的参数顺序是...headers, upload_id...，2.0.0版本中的参数顺序是...upload_id，headers...。

本地上传分片

接着，把本地文件分片上传。

```
void multipart_upload_file()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    int is_cname = 0;
    aos_table_t *headers = NULL;
    aos_table_t *complete_headers = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    aos_string_t upload_id;
    oss_upload_file_t *upload_file = NULL;
    aos_status_t *s = NULL;
    oss_list_upload_part_params_t *params = NULL;
    aos_list_t complete_part_list;
    oss_list_part_content_t *part_content = NULL;
    oss_complete_part_content_t *complete_part_content = NULL;
    int part_num1 = 1;
    int part_num2 = 2;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    headers = aos_table_make(p, 1);
    resp_headers = aos_table_make(options->pool, 5);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    /* 初始化分片上传，获取upload id，代码参考前面章节，这里省略*/
    /* 上传第一个分片 */
    upload_file = oss_create_upload_file(p);
    aos_str_set(&upload_file->filename, MULTIPART_UPLOAD_FILE_PATH);
    upload_file->file_pos = 0;
    upload_file->file_last = 200 * 1024; //200k
    s = oss_upload_part_from_file(options, &bucket, &object, &upload_id,
        part_num1, upload_file, &resp_headers);
    /* 判断是否上传分片成功 */
    if (aos_status_is_ok(s)) {
        printf("Multipart upload from file succeeded\n");
    } else {
        printf("Multipart upload from file failed\n");
    }
    /* 上传第二个分片 */
    upload_file->file_pos = 200 *1024;//remain content start pos
    upload_file->file_last = get_file_size(MULTIPART_UPLOAD_FILE_PATH);
    s = oss_upload_part_from_file(options, &bucket, &object, &upload_id,
        part_num2, upload_file, &resp_headers);
    /* 判断是否上传分片成功 */
```

```
    if (aos_status_is_ok(s)) {
        printf("Multipart upload from file succeeded\n");
    } else {
        printf("Multipart upload from file failed\n");
    }
    /* 完成分片上传，代码参考下一章节，这里省略*/
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

上面程序的核心是调用oss_upload_part_from_file接口来上传每一个分片。

说明：

- oss_upload_part_from_file接口要求除最后一个分片以外，其他的分片大小都要大于100KB。
- 分片号码的范围是1~10000。如果超出这个范围，OSS将返回InvalidArgument的错误码。
- 每次上传分片时都要把流定位到此次上传分片开头所对应的位置。
- 上述代码只是演示如何初始化分片上传，只是部分代码，完整的代码参考下面的GitHub链接。
- 完整代码参考：*GitHub*

获取已上传的分片，完成分片上传

```
void complete_multipart_upload()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_table_t *complete_headers = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    aos_string_t upload_id;
    aos_status_t *s = NULL;
    oss_list_upload_part_params_t *params = NULL;
    aos_list_t complete_part_list;
    oss_list_part_content_t *part_content = NULL;
    oss_complete_part_content_t *complete_part_content = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    headers = aos_table_make(p, 1);
    resp_headers = aos_table_make(options->pool, 5);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    /* 初始化分片上传，获取一个Upload Id，代码参考上面章节，这里省略*/
    /* 上传每个分片，代码参考上面章节，这里省略*/
    /* 获取已经上传的分片
     * 调用oss_complete_multipart_upload接口时需要每个分片的ETag值，这个值有两种获取途
径：
     * 第一种是上传每个分片的时候，返回结果里面会包含这个分片的ETag值，可以保存下来，等
后面使用；
     * 第二种是通过调用oss_list_upload_part接口获取已经上传的分片的ETag值。
```

```
     * 下面演示的是第二种方式
     */
    params = oss_create_list_upload_part_params(p);
    params->max_ret = 1000;
    aos_list_init(&complete_part_list);
    s = oss_list_upload_part(options, &bucket, &object, &upload_id,
                             params, &resp_headers);
    /* 判断是否获取分片列表成功 */
    if (aos_status_is_ok(s)) {
        printf("List multipart succeeded\n");
    } else {
        printf("List multipart failed\n");
    }
    aos_list_for_each_entry(part_content, &params->part_list, node) {
        complete_part_content = oss_create_complete_part_content(p);
        aos_str_set(&complete_part_content->part_number,
                    part_content->part_number.data);
        aos_str_set(&complete_part_content->etag, part_content->etag.data);
        aos_list_add_tail(&complete_part_content->node, &complete_part_list);
    }
    /* 完成分片上传 */
    s = oss_complete_multipart_upload(options, &bucket, &object, &upload_id,
            &complete_part_list, complete_headers, &resp_headers);
    /* 判断完成分片上传是否成功 */
    if (aos_status_is_ok(s)) {
        printf("Complete multipart upload from file succeeded, upload_id:%.*s\n",
                upload_id.len, upload_id.data);
    } else {
        printf("Complete multipart upload from file failed\n");
    }
    /* 释放资源 */
    aos_pool_destroy(p);
}
```

说明：

- 2.0.0相对于1.0.0版本，oss_complete_multipart_upload接口增加了headers参数，用来在完成时修改headers值
- 完整代码参考：*GitHub*

取消分片上传事件

```
void abort_multipart_upload()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    aos_table_t *headers = NULL;
    aos_table_t *resp_headers = NULL;
    oss_request_options_t *options = NULL;
    aos_string_t upload_id;
    aos_status_t *s = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
```

```
    headers = aos_table_make(p, 1);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    /* 初始化分片上传，获取一个Upload Id*/
    s = oss_init_multipart_upload(options, &bucket, &object,
                                  &upload_id, headers, &resp_headers);
    if (aos_status_is_ok(s)) {
        printf("Init multipart upload succeeded, upload_id:%.*s\n",
               upload_id.len, upload_id.data);
    } else {
        printf("Init multipart upload failed\n");
    }
    /* 取消这次分片上传 */
    s = oss_abort_multipart_upload(options, &bucket, &object, &upload_id,
                                   &resp_headers);
    /* 判断取消分片上传是否成功 */
    if (aos_status_is_ok(s)) {
        printf("Abort multipart upload succeeded, upload_id::%.*s\n",
               upload_id.len, upload_id.data);
    } else {
        printf("Abort multipart upload failed\n");
    }
    /* 释放资源 */
    aos_pool_destroy(p);
}
```

说明：

- 当一个分片上传事件被中止后，就不能再使用这个upload_id做任何操作，已经上传的分片数据也会被删除。
- 完整代码参考：*GitHub*

1.2.8.7 下载文件

OSS C SDK提供了丰富的文件下载接口，用户可以通过以下方式从OSS下载文件：

- 下载文件到内存
- 下载文件到本地文件
- 分段下载文件

下载文件到内存

通过oss_get_object_to_buffer接口，可以实现将文件下载到内存中：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *params;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
```

```
aos_string_t bucket;
aos_string_t object;
aos_list_t buffer;
aos_buf_t *content;
char *buf;
int64_t len = 0;
int64_t size = 0;
int64_t pos = 0;
aos_pool_create(&p, NULL);
/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);
/* 初始化参数 */
aos_str_set(&object, object_name);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 0);
params = aos_table_make(p, 0);
/* 下载文件 */
aos_list_init(&buffer);
s = oss_get_object_to_buffer(options, &bucket, &object, &buffer, headers, params, &
resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object succeeded\n");
    /* 将下载内容拷贝到buffer中*/
    len = aos_buf_list_len(&buffer);
    buf = aos_pcalloc(p, len + 1);
    buf[len] = '\0';
    aos_list_for_each_entry(content, &buffer, node) {
    size = aos_buf_size(content);
        memcpy(buf + pos, content->pos, size);
        pos += size;
    }
} else {
    printf("get object failed\n");
}
aos_pool_destroy(p);
```

说明：

- 2.0.0相对于1.0.0版本，oss_get_object_to_buffer接口增加了params参数，同时headers和params允许为NULL，1.0.0及其之前版本不支持为NULL
- oss_get_object_to_buffer_by_url和oss_get_object_to_file_by_url参数也增加了params参数
- 完整代码参考：*GitHub*

下载文件到本地文件

通过oss_get_object_to_file接口，可以实现将文件下载到指定文件：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *params;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
```

```
char *object_name = "<您的object名字>";
char *filepath = "<本地文件路径>";
aos_string_t bucket;
aos_string_t object;
aos_string_t file;
aos_pool_create(&p, NULL);
/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);
/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
aos_str_set(&file, filepath);
headers = aos_table_make(p, 0);
params = aos_table_make(p, 0);
/* 下载文件 */
s = oss_get_object_to_file(options, &bucket, &object, &file, headers, params, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object succeeded\n");
} else {
    printf("get object failed\n");
}
aos_pool_destroy(p);
```

说明：

- 2.0.0相对于1.0.0版本，oss_get_object_to_file接口增加了params参数，同时headerss和params允许为NULL，1.0.0及其之前版本不支持为NULL。
- 如果本地有同名文件 *filepath*，本地文件将会被覆盖。
- 完整代码参考：*GitHub*

分段下载文件

通过设置Range指定返回文件的传输范围，可以实现文件的分段下载。以下代码实现分段下载文件到内存中：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *params;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
aos_list_t buffer;
aos_pool_create(&p, NULL);
/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);
/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
```

```
params = NULL;
headers = aos_table_make(p, 1);
/* 设置Range，读取文件的指定范围，bytes=20-100包括第20和第100个字符 */
apr_table_set(headers, "Range", "bytes=20-100");
aos_list_init(&buffer);
/* 分片下载文件 */
s = oss_get_object_to_buffer(options, &bucket, &object, headers, params, &buffer, &
resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object succeeded\n");
} else {
    printf("get object failed\n");
}
aos_pool_destroy(p);
```

1.2.8.8 管理文件

查看所有文件

通过oss_list_object接口，可以列出当前存储空间下的所有文件。

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
oss_list_object_params_t *params;
oss_list_object_content_t *content;
int max_ret = 1000;
char *key;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&bucket, bucket_name);
params = oss_create_list_object_params(p);
params->max_ret = max_ret;
aos_str_set(&params->prefix, "<查看文件的前缀>");
aos_str_set(&params->delimiter, "<查看文件的分隔符>");
aos_str_set(&params->marker, "<查看文件的起点>");
s = oss_list_object(options, &bucket, params, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object succeeded\n");
    /* 下载文件 */
    aos_list_for_each_entry(content, &params->object_list, node) {
    key = apr_psprintf(p, "%.*s", content->key.len, content->key.data);
    }
} else {
    printf("get object failed\n");
}
aos_pool_destroy(p);
```

说明：

默认情况下，如果存储空间中的文件数量大于1000，则只会返回1000个文件，且返回结果中truncated为true，并返回next_marker作为下此读取的起点。若想增大返回文件数目，可以修改max_ret参数，或者使用marker参数分次读取。

创建模拟目录

OSS是基于对象的存储服务，没有目录的概念。创建模拟目录本质上来说是创建了一个size为0的文件。对于这个文件照样可以上传下载，只是控制台会对以 / 结尾的文件以文件夹的方式展示，所以用户可以使用上述方式来实现创建模拟目录。

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的文件夹名字/>";
aos_string_t bucket;
aos_string_t object;
char *data = "";
aos_list_t buffer;
aos_buf_t *content;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
headers = aos_table_make(p, 0);
aos_list_init(&buffer);
content = aos_buf_pack(options->pool, data, strlen(data));
aos_list_add_tail(&content->node, &buffer);
s = oss_put_object_from_buffer(options, &bucket, &object, &buffer, headers, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("create dir succeeded\n");
} else {
    printf("create dir failed\n");
}
aos_pool_destroy(p);
```

说明：

完整代码参考：*GitHub*

设定Object的Http Header

向OSS上传文件时，除了文件内容，还可以指定文件的一些属性信息，称为元信息。这些信息在上传时与文件一起存储，在下载时与文件一起返回。下面代码为文件设置了过期时间：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
```

```
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
char *data = "<object content>";
aos_list_t buffer;
aos_buf_t *content;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&object, object_name);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 0);
/* 设置http header */
headers = aos_table_make(p, 1);
apr_table_set(headers, "Expires", "Fri, 28 Feb 2012 05:38:42 GMT");
/* 读取数据到buffer中 */
aos_list_init(&buffer);
content = aos_buf_pack(options->pool, data, strlen(data));
aos_list_add_tail(&content->node, &buffer);
s = oss_put_object_from_buffer(options, &bucket, &object, &buffer, headers, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("put object succeeded\n");
} else {
    printf("put object failed\n");
}
aos_pool_destroy(p);
```

说明：

- 可以设置Http Header有：Cache-Control 、 Content-Disposition 、Content-Encoding、Expires 。它们的相关介绍见 *RFC2616* 。
- 完整代码参考：*GitHub*

设置User Meta

OSS支持用户自定义Meta信息对文件进行描述。比如：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
char *data = "<object content>";
aos_list_t buffer;
aos_buf_t *content;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
headers = aos_table_make(p, 1);
```

```
/* 设置用户自定义meta */
apr_table_set(headers, "x-oss-meta-author", "oss");
/* 上传数据到OSS中 */
aos_list_init(&buffer);
content = aos_buf_pack(options->pool, data, strlen(data));
aos_list_add_tail(&content->node, &buffer);
s = oss_put_object_from_buffer(options, &bucket, &object, &buffer, headers, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("put object succeeded\n");
} else {
    printf("put object failed\n");
}
aos_pool_destroy(p);
```

说明：

- 因为文件元信息在上传/下载时是附在HTTP Headers中， HTTP协议规定不能包含复杂字符。
- 一个文件可以有多个类似的参数，但所有的user meta总大小不能超过8KB。

获取文件的元数据

有时候，再向OSS上传文件后或者读取文件前需要获取文件的一些元数据，比如长度，文件类型等，这时候可以通过oss_object_head接口获取文件的元数据。下面代码获取文件的长度和文件类型：

```
void head_object()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    oss_request_options_t *options = NULL;
    aos_table_t *headers = NULL;
    aos_table_t *resp_headers = NULL;
    aos_status_t *s = NULL;
    char *content_length_str = NULL;
    char *object_type = NULL;
    int64_t content_length = 0;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    /* 初始化参数 */
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    headers = aos_table_make(p, 0);
    /* 获取文件元数据 */
    s = oss_head_object(options, &bucket, &object, headers, &resp_headers);
    if (aos_status_is_ok(s)) {
        /* 获取文件长度 */
        content_length_str = (char*)apr_table_get(resp_headers, OSS_CONTENT_LENGTH);
        if (content_length_str != NULL) {
            content_length = atoll(content_length_str);
        }
        /* 获取文件的类型 */
        object_type = (char*)apr_table_get(resp_headers, OSS_OBJECT_TYPE);
```

```
        printf("head object succeeded, object type:%s, content_length:%ld\n",
                object_type, content_length);
    } else {
        printf("head object failed\n");
    }
    aos_pool_destroy(p);
}
```

说明：

- oss_head_object接口不会去下载文件内容，只会读取文件的元数据，包括系统级别和用户自定义的元数据
- 完整代码参考：*GitHub*

拷贝文件

在同一个region中，用户可以对有操作权限的文件进行拷贝操作。以下代码通过oss_copy_object接口实现拷贝一个文件：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t * headers;
aos_table_t *resp_headers;
char *source_bucket_name = "<您的源bucket名字>";
char *source_object_name = "<您的源object名字>";
char *dest_bucket_name = "<您的目的bucket名字>";
char *dest_object_name = "<您的目的object名字>";
aos_string_t source_bucket;
aos_string_t source_object;
aos_string_t dest_bucket;
aos_string_t dest_object;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&source_bucket, source_bucket_name);
aos_str_set(&source_object, source_object_name);
aos_str_set(&dest_bucket, dest_bucket_name);
aos_str_set(&dest_object, dest_object_name);
headers = aos_table_make(p, 0);
/* 拷贝文件 */
s = oss_copy_object(options, &source_bucket, &source_object, &dest_bucket, &dest_object,
headers, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("copy object succeeded\n");
} else {
    printf("copy object failed\n");
}
aos_pool_destroy(p);
```

说明：

- 需要注意的是源bucket与目的bucket必须属于同一region。

- 如果拷贝文件超过1G，建议使用分片拷贝文件方式进行拷贝。

分片拷贝文件

当拷贝一个大于500MB的文件，建议通过oss_upload_part_copy接口来进行拷贝。以下代码实现分片拷贝一个文件：

```
aos_pool_t *p;
oss_request_options_t *options;
char *source_bucket_name = "<您的源bucket名字>";
char *source_object_name = "<您的源object名字>";
char *dest_bucket_name = "<您的目的bucket名字>";
char *dest_object_name = "<您的目的object名字>";
aos_string_t dest_bucket;
aos_string_t dest_object;
aos_string_t upload_id;
aos_table_t *init_headers;
aos_table_t *copy_headers;
aos_table_t *list_part_resp_headers;
aos_table_t *complete_resp_headers
aos_table_t *resp_headers;
aos_status_t *s;
oss_list_upload_part_params_t *list_upload_part_params;
oss_upload_part_copy_params_t *upload_part_copy_params1;
aos_list_t complete_part_list;
oss_list_part_content_t *part_content;
oss_complete_part_content_t *complete_content;
int part1 = 1;
int64_t range_start1 = 0;
int64_t range_end1 = 6000000;//not less than 5MB
int max_ret = 1000;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&dest_bucket, dest_bucket_name);
aos_str_set(&dest_object, dest_object_name);
init_headers = aos_table_make(p, 0);
s = oss_init_multipart_upload(options, &dest_bucket, &dest_object, init_headers, &upload_id, &
resp_headers);
/* 拷贝第一个分片数据 */
upload_part_copy_params1 = oss_create_upload_part_copy_params(p);
aos_str_set(&upload_part_copy_params1->source_bucket, source_bucket_name);
aos_str_set(&upload_part_copy_params1->source_object, source_object_name);
aos_str_set(&upload_part_copy_params1->dest_bucket, dest_bucket_name);
aos_str_set(&upload_part_copy_params1->dest_object, dest_object_name);
aos_str_set(&upload_part_copy_params1->upload_id, upload_id->data);
upload_part_copy_params1->part_num = part1;
upload_part_copy_params1->range_start = range_start1;
upload_part_copy_params1->range_end = range_end1;
copy_headers = aos_table_make(p, 0);
s = oss_upload_part_copy(options, upload_part_copy_params1, copy_headers, &resp_headers
);
if (aos_status_is_ok(s)) {
    printf("upload part copy succeeded\n");
} else {
    printf("upload part copy failed\n");
}
/* 继续拷贝剩余的分片，这里省略*/
...
```

```
/* 列出分片 */
list_upload_part_params = oss_create_list_upload_part_params(p);
list_upload_part_params->max_ret = max_ret;
aos_list_init(&complete_part_list);
s = oss_list_upload_part(options, &dest_bucket, &dest_object, &upload_id,
list_upload_part_params, &list_part_resp_headers);
aos_list_for_each_entry(part_content, &list_upload_part_params->part_list, node) {
   complete_content = oss_create_complete_part_content(p);
   aos_str_set(&complete_content->part_number, part_content->part_number.data);
   aos_str_set(&complete_content->etag, part_content->etag.data);
   aos_list_add_tail(&complete_content->node, &complete_part_list);
}
/* 完成分片拷贝 */
s = oss_complete_multipart_upload(options, &dest_bucket, &dest_object, &upload_id, &
complete_part_list, &complete_resp_headers);
if (aos_status_is_ok(s)) {
    printf("complete multipart upload succeeded\n");
} else {
    printf("complete multipart upload failed\n");
}
aos_pool_destroy(p);
```

删除文件

通过oss_delete_object接口，可以实现删除某个文件：

```
void delete_object()
{
    aos_pool_t *p = NULL;
    aos_string_t bucket;
    aos_string_t object;
    oss_request_options_t *options = NULL;
    aos_table_t *resp_headers = NULL;
    aos_status_t *s = NULL;
    aos_pool_create(&p, NULL);
    /* 创建并初始化options */
    options = oss_request_options_create(p);
    init_options(options);
    aos_str_set(&bucket, "<您的bucket名字>");
    aos_str_set(&object, "<您的object名字>");
    /* 删除文件 */
    s = oss_delete_object(options, &bucket, &object, &resp_headers);
    /* 判断是否删除成功 */
    if (aos_status_is_ok(s)) {
        printf("delete object succeed\n");
    } else {
        printf("delete object failed\n");
    }
    /* 释放资源*/
    aos_pool_destroy(p);
}
```

说明：

完整代码参考：*GitHub*

批量删除文件

通过oss_delete_objects接口，可以实现删除一批文件，用户可以通过**is_quiet**参数来指定是否返回删除的结果：

```
aos_pool_t *p;
aos_status_t *s;
aos_table_t *resp_headers;
oss_request_options_t *options;
char *bucket_name = "<您的bucket名字>";
char *object_name1 = "<您的object名字1>";
char *object_name2 = "<您的object名字2>";
aos_string_t bucket;
aos_string_t object1;
aos_string_t object2;
oss_object_key_t *content1;
oss_object_key_t *content2;
aos_list_t object_list;
aos_list_t deleted_object_list;
int is_quiet = 1;
options = oss_request_options_create(p);
init_options(options);
aos_str_set(&bucket, bucket_name);
aos_str_set(&object1, object_name1);
aos_str_set(&object2, object_name2);
/* 构建待删除文件列表 */
aos_list_init(&object_list);
aos_list_init(&deleted_object_list);
content1 = oss_create_oss_object_key(p);
aos_str_set(&content1->key, object_name1);
aos_list_add_tail(&content1->node, &object_list);
content2 = oss_create_oss_object_key(p);
aos_str_set(&content2->key, object_name2);
aos_list_add_tail(&content2->node, &object_list);
/* 删除多个文件 */
s = oss_delete_objects(options, &bucket, &object_list, is_quiet, &resp_headers, &deleted_ob
ject_list);
if (aos_status_is_ok(s)) {
    printf("delete objects succeeded\n");
} else {
    printf("delete objects failed\n");
}
aos_pool_destroy(p);
```

批量删除指定前缀的文件

通过oss_delete_objects_by_prefix接口，可以实现删除一批指定前缀的文件：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *prefix_str = "<删除文件的前缀>";
aos_string_t bucket;
aos_string_t prefix;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
```

```
init_options(options);
aos_str_set(&bucket, bucket_name);
aos_str_set(&prefix, prefix_str);
/* 删除满足特定前缀的文件*/
s = oss_delete_objects_by_prefix(options, &bucket, &prefix);
if (aos_status_is_ok(s)) {
    printf("delete objects by prefix succeeded\n");
} else {
    printf("delete objects by prefix failed\n");
}
aos_pool_destroy(p);
```

说明：

- 批量删除指定前缀的文件可以实现删除目录的功能，比如删除dir目录，可以通过设置prefix的值为 **dir/** 实现。
- 如果设置prefix的值为空字符串(“”)或者NULL，将会删除整个bucket，请谨慎使用。

1.2.8.9 授权访问

使用STS服务临时授权

OSS可以通过阿里云STS服务，临时进行授权访问。使用STS时请按以下步骤进行：

1. 在官网控制台创建子账号。
2. 在官网控制台创建STS角色并赋予子账号扮演角色的权限。
3. 使用子账号的AccessKeyId/AccessKeySecret向STS申请临时token
4. 使用临时token中的认证信息创建OSS的Client
5. 使用OSS的Client访问OSS服务

使用STS凭证构造签名请求

用户的client端拿到STS临时凭证后，通过其中安全令牌(SecurityToken)以及临时访问密钥(AccessKeyId, AccessKeySecret)生成oss_request_options。以上传文件为例：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
char *data = "<object content>";
aos_list_t buffer;
aos_buf_t *content;

aos_pool_create(&p, NULL);
```

```
// init_oss_request_options using sts_token
/* 创建并用STS token初始化options */
options = oss_request_options_create(p);
options->config = oss_config_create(options->pool);
aos_str_set(&options->config->endpoint, "<您的Endpoint>");
aos_str_set(&options->config->access_key_id, "<您的临时AccessKeyId>");
aos_str_set(&options->config->access_key_secret, "<您的临时AccessKeySecret>");
aos_str_set(&options->config->sts_token, "<您的sts_token>");
options->config->is_cname = 0;
options->ctl = aos_http_controller_create(options->pool, 0);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
headers = aos_table_make(p, 0);
aos_list_init(&buffer);
content = aos_buf_pack(options->pool, data, strlen(data));
aos_list_add_tail(&content->node, &buffer);

/* 上传文件 */
s = oss_put_object_from_buffer_s(options, &bucket, &object, &buffer, headers, &resp_headers
);
if (aos_status_is_ok(s)) {
    printf("put object succeeded\n");
} else {
    printf("put object failed\n");
}

aos_pool_destroy(p);
```

URL签名授权

可以通过生成签名URL的形式提供给用户一个临时的访问URL。在生成URL时，可以指定URL过期的时间，从而限制用户长时间访问。

生成签名url

通过oss_gen_signed_url接口生成请求url签名。

生成下载请求的url签名

```
aos_pool_t *p;
oss_request_options_t *options;
aos_http_request_t *req;
char *url_str;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
apr_time_t now;
int64_t expire_time;
int one_hour = 3600; /* 单位：秒*/

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
```

```
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
req = aos_http_request_create(p);
req->method = HTTP_GET;
now = apr_time_now(); //millisecond
expire_time = now / 1000000 + one_hour;

/* 生成签名url */
url_str = oss_gen_signed_url(options, &bucket, &object, expire_time, req);
printf("临时下载url:%s\n", url_str);

aos_pool_destroy(p);
```

生成上传文件请求的url签名

```
aos_pool_t *p;
oss_request_options_t *options;
aos_http_request_t *req;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
aos_string_t bucket;
aos_string_t object;
apr_time_t now;
int64_t expire_time;
int one_hour = 3600;
char *url_str = NULL;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
req = aos_http_request_create(p);
req->method = HTTP_PUT;
now = apr_time_now(); //millisecond
expire_time = now / 1000000 + one_hour;

/* 生成签名url */
url_str = oss_gen_signed_url((options, &bucket, &object, expire_time, req);
printf("临时上传url:%s\n", url_str);

aos_pool_destroy(p);
```

使用签名URL发送请求

使用URL签名的方式下载文件。

```
aos_pool_t *p;
oss_request_options_t *options;
aos_http_request_t *req;
aos_table_t *headers;
aos_table_t *resp_headers;
```

```
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
char *filepath = "<本地文件路径>";
aos_string_t bucket;
aos_string_t object;
aos_string_t file;
char *url_str;
apr_time_t now;
int64_t expire_time;
int one_hour = 3600;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
aos_str_set(&file, filepath);
headers = aos_table_make(p, 0);
req = aos_http_request_create(p);
req->method = HTTP_GET;
now = apr_time_now();  /* 单位：微秒 */
expire_time = now / 1000000 + one_hour;

/* 生成签名url */
url_str = oss_gen_signed_url(options, &bucket, &object, expire_time, req);

/* 使用签名url上传文件 */
s = oss_get_object_to_file_by_url(options, url_str, headers, &file, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object succeeded\n");
} else {
    printf("get object failed\n");
}
aos_pool_destroy(p);
```

使用URL签名的方式上传文件

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_http_request_t *req;
aos_table_t *headers;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
char *object_name = "<您的object名字>";
char *filepath = "<本地文件路径>";
aos_string_t bucket;
aos_string_t object;
aos_string_t file;
char *url_str;
apr_time_t now;
int64_t expire_time;
int one_hour = 3600;

aos_pool_create(&p, NULL);
```

```
/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 初始化参数 */
aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
aos_str_set(&file, filepath);
headers = aos_table_make(p, 0);
req = aos_http_request_create(p);
req->method = HTTP_PUT;
now = apr_time_now(); /* 单位：微秒*/
expire_time = now / 1000000 + one_hour;

/* 生成签名url */
url_str = oss_gen_signed_url(options, &bucket, &object, expire_time, req);

/* 使用签名url上传文件 */
s = oss_put_object_from_file_by_url(options, url_str, &file, headers, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("put objects by signed url succeeded\n");
} else {
    printf("put objects by signed url failed\n");
}

aos_pool_destroy(p);
```

1.2.8.10 生命周期管理

OSS允许用户对存储空间设置生命周期规则，以自动淘汰过期掉的文件，节省存储空间。

设置生命周期规则

通过oss_put_bucket_lifecycle接口，可以实现设置生命周期规则：

lifecycle的配置规则由一段xml表示。

```
<LifecycleConfiguration>
  <Rule>
    <ID>delete obsoleted files</ID>
    <Prefix>obsoleted/</Prefix>
    <Status>Enabled</Status>
    <Expiration>
      <Days>3</Days>
    </Expiration>
  </Rule>
</LifecycleConfiguration>
```

各字段解释：

- ID字段是用来唯一表示本条Rule（各个ID之间不能由包含关系，比如abc和abcd这样的）。
- Prefix指定对bucket下的符合特定前缀的文件使用规则。
- Status指定本条规则的状态，只有Enabled和Disabled，分别表示启用规则和禁用规则。

- Expiration节点里面的Days表示大于文件最后修改时间指定的天数就删除文件，Date则表示到指定的绝对时间之后就删除文件(绝对时间服从ISO8601的格式)。

可以通过下面的代码，设置上述lifecycle规则。

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
aos_list_t lifecycle_rule_list;
oss_lifecycle_rule_content_t *rule_content;
char *rule_name = "rule_name";

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 创建生命周期规则并设置给存储空间 */
aos_str_set(&bucket, bucket_name);
aos_list_init(&lifecycle_rule_list);
rule_content = oss_create_lifecycle_rule_content(p);
aos_str_set(&rule_content->id, rule_name);
aos_str_set(&rule_content->prefix, "obsoleted");
aos_str_set(&rule_content->status, "Enabled");
rule_content->days = 3;
aos_list_add_tail(&rule_content->node, &lifecycle_rule_list);
s = oss_put_bucket_lifecycle(options, &bucket, &lifecycle_rule_list, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("put bucket lifecycle succeeded\n");
} else {
    printf("put bucket lifecycle failed\n");
}

aos_pool_destroy(p);
```

查看生命周期规则

通过oss_get_bucket_lifecycle接口，可以实现查看生命周期规则：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;
aos_list_t lifecycle_rule_list;
oss_lifecycle_rule_content_t *rule_content;
char *rule_id;
char *prefix;
char *status;
int days = INT_MAX;
char* date = "";

aos_pool_create(&p, NULL);
```

```
/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 获取存储空间生命周期规则并打印 */
aos_str_set(&bucket, bucket_name);
aos_list_init(&lifecycle_rule_list);
s = oss_get_bucket_lifecycle(options, &bucket, &lifecycle_rule_list, &resp_headers);
aos_list_for_each_entry(rule_content, &lifecycle_rule_list, node) {
    rule_id = apr_psprintf(p, "%.*s", rule_content->id.len, rule_content->id.data);
    prefix = apr_psprintf(p, "%.*s", rule_content->prefix.len, rule_content->prefix.data);
    status = apr_psprintf(p,"%.*s", rule_content->status.len, rule_content->status.data);
    date = apr_psprintf(p, "%.*s", rule_content->date.len, rule_content->date.data);
    days = rule_content->days;
}

aos_pool_destroy(p);
```

清空生命周期规则

通过oss_delete_bucket_lifecycle接口，实现清空生命周期规则：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
char *bucket_name = "<您的bucket名字>";
aos_string_t bucket;

aos_pool_create(&p, NULL);

/* 创建并初始化options */
options = oss_request_options_create(p);
init_options(options);

/* 删除存储空间生命周期规则 */
aos_str_set(&bucket, bucket_name);
s = oss_delete_bucket_lifecycle(options, &bucket, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("delete bucket lifecycle succeeded\n");
} else {
    printf("delete bucket lifecycle failed\n");
}

aos_pool_destroy(p);
```

1.2.8.11 图片处理

OSS图片处理，是OSS对外提供的海量、安全、低成本、高可靠的图片处理服务。用户将原始图片上传保存到OSS，通过简单的 RESTful 接口，在任何时间、任何地点、任何互联网设备上对图片进行处理。图片处理提供图片处理接口，图片上传请使用上传接口。基于OSS图片处理，用户可以搭建自己的图片处理服务。

图片处理基础功能

OSS图片处理提供以下功能：

- 获取图片信息
- 图片格式转换
- 图片缩放、裁剪、旋转
- 图片效果
- 图片添加图片、文字、图文混合水印
- 自定义图片处理样式，在控制台的 **图片处理** > **样式管理** 中定义
- 通过级联处理调用多个图片处理功能

图片处理使用

图片处理使用标准的 HTTP GET 请求来访问，所有的处理参数是编码在 URL 中的QueyString。

匿名访问

如果图片文件（Object）的访问权限是 公共读，如下表所示的权限，则可以匿名访问图片服务。

Bucket权限	Object权限
公共读私有写（public-read）或公共读写（public-read-write）	默认（default）
任意权限	公共读私有写（public-read）或 公共读写（public-read-write）

通过如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value
```

- bucket：用户的存储空间（bucket）名称
- endpoint：用户存储空间所在数据中心的访问域名
- object：用户上传在OSS上的图片文件
- image：图片处理保留标志符
- action：用户对图片做的操作，如缩放、裁剪、旋转等
- parame：用户对图片做的操作所对应的参数

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100
```

自定义样式，使用如下格式的三级域名匿名访问图片处理：

```
http://bucket.<endpoint>/object?x-oss-process=x-oss-process=style/name
```

- style：用户自定义样式系统保留标志符
- name：自定义样式名称，即控制台定义样式的 规则名

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=style/oss-pic-style-w-100
```

通过级联处理，可以对一张图片顺序实施多个操作，格式如下：

```
http://bucket.<endpoint>/object?x-oss-process=image/action,parame_value/action,parame_value/...
```

例如：

```
http://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100/rotate,90
```

图片服务也支持HTTPS访问，例如：

```
https://image-demo.oss-cn-hangzhou.aliyuncs.com/example.jpg?x-oss-process=image/resize,w_100
```

授权访问

对私有权限的文件（Object），如下表所示的权限，必须通过授权才能访问图片服务。

Bucket权限	Object权限
私有读写（private）	默认权限（default）
任意权限	私有读写（private）

生成带签名的图片处理的URL代码如下：

```
aos_pool_t *p;
oss_request_options_t *options;
aos_http_request_t *req;
char *url_str;
aos_table_t *params = NULL;
aos_string_t bucket;
aos_string_t object;
```

```
apr_time_t now;
int64_t expire_time;
aos_pool_create(&p, NULL);
/* 创建并初始化options */
options = oss_request_options_create(p);
init_sample_request_options(options, AOS_FALSE);
/* 初始化参数 */
aos_str_set(&bucket, "<您的bucket名字>");
aos_str_set(&object, "<您的object名字>");
/* 图片处理 */
params = aos_table_make(p, 1);
apr_table_set(params, OSS_PROCESS, "image/resize,m_fixed,w_100,h_100");
req = aos_http_request_create(p);
req->method = HTTP_GET;
req->query_params = params;
/* 过期时间，单位秒 */
now = apr_time_now();
expire_time = now / 1000000 + 10 * 60;
/* 生成签名url */
url_str = oss_gen_signed_url(options, &bucket, &object, expire_time, req);
printf("url:%s\n", url_str);
aos_pool_destroy(p);
```

说明：

- 授权访问支持**自定义样式**、**HTTPS**、**级联处理**。
- **oss_gen_signed_url** 过期时间单位是**秒**。

SDK访问

对于任意权限的图片文件，都可以直接使用SDK 访问图片、进行处理。

说明：

- 图片处理的完整代码请参考：*GitHub*
- SDK处理图片文件支持**自定义样式**、**HTTPS**、**级联处理**。

基础操作

图片处理的基础操作包括，获取图片信息、格式转换、缩放、裁剪、旋转、效果、水印等。

```
aos_pool_t *p = NULL;
aos_string_t bucket;
aos_string_t object;
oss_request_options_t *options = NULL;
aos_table_t *headers = NULL;
aos_table_t *params = NULL;
aos_table_t *resp_headers = NULL;
aos_status_t *s = NULL;
aos_string_t filename;
char *style = NULL;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
```

```
init_sample_request_options(options, AOS_FALSE);
aos_str_set(&bucket, "<您的bucket名字>");
aos_str_set(&object, "<您的object名字>");
aos_str_set(&filename, "<处理后图片保存路径>");
params = aos_table_make(p, 1);
style = "image/resize,m_fixed,w_100,h_100";
apr_table_set(params, OSS_PROCESS, "image/resize,m_fixed,w_100,h_100");
/* 下载处理后的图片到本地文件 */
s = oss_get_object_to_file(options, &bucket, &object, headers,
                           params, &filename, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object to file succeeded\n");
} else {
    printf("get object to file failed\n");
}
aos_pool_destroy(p);
```

自定义样式

```
aos_pool_t *p = NULL;
aos_string_t bucket;
aos_string_t object;
oss_request_options_t *options = NULL;
aos_table_t *headers = NULL;
aos_table_t *params = NULL;
aos_table_t *resp_headers = NULL;
aos_status_t *s = NULL;
aos_string_t filename;
char *style = NULL;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_sample_request_options(options, AOS_FALSE);
aos_str_set(&bucket, "<您的bucket名字>");
aos_str_set(&object, "<您的object名字>");
aos_str_set(&filename, "<处理后图片保存路径>");
/* 自定义样式 */
style = "style/oss-pic-style-w-100";
params = aos_table_make(p, 1);
apr_table_set(params, OSS_PROCESS, "image/resize,m_fixed,w_100,h_100");
/* 下载处理后的图片到本地文件 */
s = oss_get_object_to_file(options, &bucket, &object, headers,
                           params, &filename, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object to file succeeded\n");
} else {
    printf("get object to file failed\n");
}
aos_pool_destroy(p);
```

级联处理

```
aos_pool_t *p = NULL;
aos_string_t bucket;
aos_string_t object;
oss_request_options_t *options = NULL;
aos_table_t *headers = NULL;
aos_table_t *params = NULL;
aos_table_t *resp_headers = NULL;
aos_status_t *s = NULL;
```

```
aos_string_t filename;
char *style = NULL;
aos_pool_create(&p, NULL);
options = oss_request_options_create(p);
init_sample_request_options(options, AOS_FALSE);
aos_str_set(&bucket, "<您的bucket名字>");
aos_str_set(&object, "<您的object名字>");
aos_str_set(&filename, "<处理后图片保存路径>");
/* 级联处理 */
style = "image/resize,m_fixed,w_100,h_100/rotate,90";
params = aos_table_make(p, 1);
apr_table_set(params, OSS_PROCESS, "image/resize,m_fixed,w_100,h_100");
/* 下载处理后的图片到本地文件 */
s = oss_get_object_to_file(options, &bucket, &object, headers,
                           params, &filename, &resp_headers);
if (aos_status_is_ok(s)) {
    printf("get object to file succeeded\n");
} else {
    printf("get object to file failed\n");
}
aos_pool_destroy(p);
```

图片处理工具

- 可视化图片处理工具 *ImageStyleViever* ，可以直观的看到OSS图片处理的结果。
- OSS图片处理的功能、使用演示页面。

1.2.8.12 错误处理

异常处理

使用OSS C SDK时如果请求出错，会有相应的错误信息在aos_status_s中输出。aos_status_s有以下几个属性：

- code: 出错请求的HTTP状态码，整形类型。
- error_code: OSS的错误码，字符串类型。
- error_msg: OSS的错误信息，字符串类型。
- req_id: 标识该次请求的UUID；当您无法解决问题时，可以凭这个req_id来请求OSS开发工程师的帮助，字符串类型。

超时处理

如何判断超时

- 如果返回的aos_status_t中的code为5XX，表示服务器内部错误，可以重试。
- 如果返回的aos_status_t中的code不等于2XX，且error_code为-992或者-995时，表示链接超时或请求超时，可以重试。

- 可以使用aos_status.h中的aos_should_retry(aos_status_t *)判断返回的错误码是否需要重试，如果返回1，表示需要重试。

如何设置超时时间

- 设置链接超时：options->ctl->options->connect_timeout，单位秒，默认是10秒。
- 设置DNS超时：options->ctl->options->dns_cache_timeout，单位秒，默认是60秒。
- 设置请求超时：
  - 通过设置options->ctl->options->speed_limit的值控制能容忍的最小速率，默认是1024，即1K。
  - 通过设置options->ctl->options->speed_time的值控制能容忍的最长时间，默认是15秒。
  - 如果使用了上述默认值，表示如果连续15秒的传输速率小于1K，则超时。

常见错误码

错误码	描述
AccessDenied	拒绝访问
BucketAlreadyExists	Bucket已经存在
BucketNotEmpty	Bucket不为空
EntityTooLarge	实体过大
EntityTooSmall	实体过小
FileGroupTooLarge	文件组过大
FilePartNotExist	文件Part不存在
FilePartStale	文件Part过时
InvalidArgument	参数格式错误
InvalidAccessKeyId	AccessKeyId不存在
InvalidBucketName	无效的Bucket名字
InvalidDigest	无效的摘要
InvalidObjectName	无效的Object名字
InvalidPart	无效的Part
InvalidPartOrder	无效的part顺序
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket

错误码	描述
InternalError	OSS内部发生错误
MalformedXML	XML格式非法
MethodNotAllowed	不支持的方法
MissingArgument	缺少参数
MissingContentLength	缺少内容长度
NoSuchBucket	Bucket不存在
NoSuchKey	文件不存在
NoSuchUpload	Multipart Upload ID不存在
NotImplemented	无法处理的方法
PreconditionFailed	预处理错误
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟
RequestTimeout	请求超时
SignatureDoesNotMatch	签名错误
TooManyBuckets	用户的Bucket数目超过限制

1.2.8.13 FAQ

OSS C SDK在嵌入式环境下如何编译OSS C SDK如何设置程序日志的输出OSS C SDK如何设置通信时和CURL相关的一些参数OSS C SDK利用CURL提供的Callback实现上传OSS C SDK利用CURL提供的Callback实现数据下载OSS C SDK(windows版本)如何上传和下载包含中文名的文件

1.2.9 Ruby-SDK

1.2.9.1 安装

- github地址：*https://github.com/aliyun/aliyun-oss-ruby-sdk*
- API文档地址：*http://www.rubydoc.info/gems/aliyun-sdk/*
- ChangeLog：*https://github.com/aliyun/aliyun-oss-ruby-sdk/blob/master/CHANGELOG.md*

要求

- 开通阿里云OSS服务，并创建了AccessKeyId 和AccessKeySecret。
- 如果您还没有开通或者还不了解阿里云OSS服务，请登录 *OSS产品主页*了解。

- 如果还没有创建AccessKeyId和AccessKeySecret，请到阿里云Access Key管理创建Access Key。

安装

直接用gem安装：

```
gem install aliyun-sdk
```

如果无法访问 *https://rubygems.org*，则可以使用淘宝的镜像源：

```
gem install aliyun-sdk --clear-sources --source https://ruby.taobao.org
```

或者通过*bundler*安装，首先在你的应用程序的 Gemfile中添加：

```
gem 'aliyun-sdk', '~> 0.3.0'
```

然后运行：

```
# 使用淘宝的镜像源，可选
bundle config mirror.https://rubygems.org https://ruby.taobao.org

bundle install
```

说明：

https://ruby.taobao.org 是完整的rubygems.org的镜像，自动和官方源同步。 不方便访问rubygems.org的用户可以使用此源。

依赖

- Ruby版本 >= 1.9.3
- 支持Ruby运行环境的Windows/Linux/OS X系统

说明：

- SDK依赖的一些gem是本地扩展的形式，因此需要安装ruby-dev以支持编译本地扩展的gem
- SDK依赖的处理XML的gem(nokogiri)要求环境中包含zlib库

Linux

Linux中以Ubuntu为例，安装上述依赖的方法：

```
sudo apt-get install ruby ruby-dev zlib1g-dev
```

各个Linux发行版都有自己的包管理工具，安装的方法类似。

Windows

1. 前往*Ruby Installer*下载RubyInstaller，双击安装，在安装时选中**Add Ruby executables to your PATH**。

   说明：
   请下载2.1及以下版本。2.2版本因为存在问题无法顺利安装。

2. 前往*Ruby Installer*下载DEVELOPMENT KIT，注意选择相应的版本。下载后是一个压缩包，在解压前首先在C盘根目录建立一个文件夹 *C:\RubyDev*，然后将压缩包解压到此文件夹。
3. 按 **Windows + R** 输入 cmd 后回车进入到命令行窗口，输入以下命令：

   ```
   cd C:\RubyDev
   ruby dk.rb init
   ruby dk.rb install
   ```

   如果最后一步install时显示“config.yml配置错误”，则用文本编辑器打开 *C:\RubyDev\config.yml*，将其内容改为：

   ```
   ---
   - C:/Ruby21
   ```

   保存config.yml然后继续 ruby dk.rb install。其中*Ruby21*是Ruby的安装目录。根据具体的名字填写。完成后关闭此命令行窗口。

4. 按 **Windows + R** 输入 cmd 后回车进入到命令行窗口，输入以下命令：

   ```
   gem install aliyun-sdk
   ```

   安装顺利完成后，输入 irb进入Ruby交互式命令行，输入 require 'aliyun/oss'，如果显示 **true** 则SDK已经顺利安装。

OS X

1. OS X系统默认已经安装了ruby，但是为了方便使用和管理，建议用户再安装一个开发用的版本。
2. 在终端输入xcode-select --install安装*Xcode command line tools*。如果安装失败，可选择手动下载安装（见下载的步骤）。
3. 从苹果开发者网站下载*Xcode command line tools*，需要用您的Apple ID登录后才能下载。注意选择与您的系统匹配的版本。下载完成后双击加载dmg文件，然后在打开的窗口中双击安装程序进行安装。在安装的过程中需要输入您的系统密码。

4. 安装brew，在终端输入以下命令：

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

5. 安装ruby，在终端输入以下命令：

```
brew install ruby
exec $SHELL -l
```

6. 安装SDK，在终端输入以下命令：

```
安装SDK，在终端输入以下命令：
```

7. 在终端输入以下命令验证SDK安装成功：

```
irb
> require 'aliyun/oss'
=> true
```

1.2.9.2 快速开始

下面介绍如何使用OSS Ruby SDK来访问OSS服务，包括查看Bucket列表，查看文件列表，上传/下载文件和删除文件。为了方便使用，下面的操作都是在Ruby的交互式命令行irb中进行。

初始化Client

在命令行中输入并回车：

```
irb
```

进入到Ruby的交互式命令行模式。接着通过require引入SDK的包：

```
> require 'aliyun/oss'
=> true
```

说明：

在接下来的演示中，**>** 符号后面的内容是用户输入的命令，**=>** 后面的内容是程序返回的内容。

接下来创建Client：

```
> client = Aliyun::OSS::Client.new(
>   endpoint: 'endpoint',
>   access_key_id: 'AccessKeyId',
>   access_key_secret: 'AccessKeySecret')
=> #<Aliyun::OSS::Client...
```

将其中的参数替换成您实际的endpoint，AccessKeyId和AccessKeySecret。

查看Bucket列表

通过以下命令查看Bucket列表：

```
> buckets = client.list_buckets
=> #<Enumerator...
> buckets.each { |b| puts b.name }
=> bucket-1
=> bucket-2
=> ...
```

如果Bucket列表为空，则可以用以下命令创建一个Bucket：

```
> client.create_bucket('my-bucket')
=> true
```

说明：

- Bucket名字不能与OSS服务中其他用户已有的Bucket重复，所以你需要选择一个独特的Bucket名字以避免创建失败。

查看文件列表

通过以下命令查看Bucket中的文件列表：

```
> bucket = client.get_bucket('my-bucket')
=> #<Aliyun::OSS::Bucket...
> objects = bucket.list_objects
=> #<Enumerator...
> objects.each { |obj| puts obj.key }
=> object-1
=> object-2
=> ...
```

上传一个文件

通过以下命令向Bucket中上传一个文件：

```
> bucket.put_object('my-object', :file => 'local-file')
=> true
```

其中*'local-file'*是需要上传的本地文件的路径。上传成功后，可以通过 `list_objects` 来查看：

```
> objects = bucket.list_objects
=> #<Enumerator...
> objects.each { |obj| puts obj.key }
=> my-object
```

```
=> ...
```

下载一个文件

通过以下命令从Bucket中下载一个文件：

```
> bucket.get_object('my-object', :file => 'local-file')
=> #<Aliyun::OSS::Object...
```

其中*'local-file'*是文件保存的路径。下载成功后，可以打开文件查看其内容。

删除一个文件

通过以下命令从Bucket中删除一个文件：

```
> bucket.delete_object('my-object')
=> true
```

删除文件后可以通过 `list_objects` 来查看文件确实已经被删除：

```
> objects = bucket.list_objects
=> #<Enumerator...
> objects.each { |obj| puts obj.key }
=> object-1
=> ...
```

1.2.9.3 管理Bucket

存储空间（Bucket）是OSS上的命名空间，也是计费、权限控制、日志记录等高级功能的管理实体。

查看所有Bucket

使用`Client#list_buckets`接口列出当前用户下的所有Bucket，用户还可以指定**:prefix**参数，列出Bucket名字为特定前缀的所有Bucket：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

buckets = client.list_buckets
buckets.each { |b| puts b.name }

buckets = client.list_buckets(:prefix => 'my-')
```

```
buckets.each { |b| puts b.name }
```

创建Bucket

使用Client#create_bucket接口创建一个Bucket，用户需要指定Bucket的名字：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

client.create_bucket('my-bucket')
```

说明：

- 由于存储空间的名字是全局唯一的，所以必须保证您的Bucket名字不与别人的重复。

删除Bucket

使用Client#delete_bucket接口删除一个Bucket，用户需要指定Bucket的名字：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

client.delete_bucket('my-bucket')
```

说明：

- 如果该Bucket下还有文件存在，则需要先删除所有文件才能删除Bucket。
- 如果该Bucket下还有未完成的上传请求，则需要通过list_uploads和abort_upload先取消那些请求才能删除Bucket。用法请参考 *API文档*。

查看Bucket是否存在

用户可以通过Client#bucket_exists?接口查看当前用户的某个Bucket是否存在：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')
```

```
puts client.bucket_exists?('my-bucket')
```

Bucket访问权限

用户可以设置Bucket的访问权限，允许或者禁止匿名用户对其内容进行读写。更多关于访问权限的内容请参考访问权限。

获取Bucket的访问权限（ACL）

通过Bucket#acl查看Bucket的ACL：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
puts bucket.acl
```

设置Bucket的访问权限（ACL）

通过Bucket#acl=设置Bucket的ACL：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.acl = Aliyun::OSS::ACL::PUBLIC_READ
puts bucket.acl
```

1.2.9.4 上传文件

OSS Ruby SDK提供了丰富的文件上传接口，用户可以通过以下方式向OSS中上传文件：

- 上传本地文件到OSS
- 流式上传
- 断点续传上传
- 追加上传
- 上传回调

上传本地文件

通过Bucket#put_object接口，并指定**:file**参数来上传一个本地文件到OSS：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
```

```
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.put_object('my-object', :file => 'local-file')
```

流式上传

在进行大文件上传时，往往不希望一次性处理全部的内容然后上传，而是希望流式地处理，一次上传一部分内容。甚至如果要上传的内容本身就来自网络，不能一次 获取，那只能流式地上传。通过Bucket#put_object接口，并指定**block**参数来将流式生成的内容上传到OSS：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.put_object('my-object') do |stream|
  100.times { |i| stream << i.to_s }
end
```

断点续传上传

当上传大文件时，如果网络不稳定或者程序崩溃了，则整个上传就失败了。用户不得不重头再来，这样做不仅浪费资源，在网络不稳定的情况下，往往重试多次 还是无法完成上传。通过Bucket#resumable_upload接口来实现断点续传上传。它有以下参数：

- key 上传到OSS的Object名字
- file 待上传的本地文件路径
- opts 可选项，主要包括：
  - :cpt_file：指定checkpoint文件的路径，如果不指定则默认为与本地文件同目录下的*file.cpt*，其中**file**是本地文件的名字
  - :disable_cpt：如果指定为true，则上传过程中不会记录上传进度，失败后也无法进行续传
  - :part_size：指定每个分片的大小，默认为4MB
  - &block：如果调用时候传递了block，则上传进度会交由block处理

详细的参数请参考*API文档*。

其实现的原理是将要上传的文件分成若干个分片分别上传，最后所有分片都上传成功后，完成整个文件的上传。在上传的过程中会记录当前上传的进度信息（记 录在checkpoint文件中），如果上传

过程中某一分片上传失败，再次上传时会从 checkpoint文件中记录的点继续上传。这要求再次调用时要指定与上次相同的 checkpoint文件。上传完成后，checkpoint文件会被删除。

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.resumable_upload('my-object', 'local-file') do |p|
  puts "Progress: #{p}"
end

bucket.resumable_upload(
  'my-object', 'local-file',
  :part_size => 100 * 1024, :cpt_file => '/tmp/x.cpt') { |p|
  puts "Progress: #{p}"
}
```

说明：

- SDK会将上传的中间状态信息记录在cpt文件中，所以要确保用户对cpt文件有写权限。
- cpt文件记录了上传的中间状态信息并自带了校验，用户不能去编辑它，如果cpt文件损坏则上传无法继续。整个上传完成后cpt文件会被删除。
- 如果上传过程中本地文件发生了改变，则上传会失败。

追加上传

OSS支持可追加的文件类型，通过Bucket#append_object来上传可追加的文件， 调用时需要指定文件追加的位置，对于新创建文件，这个位置是0；对于已经存在的文件，这个位置必须是追加前文件的长度。

- 文件不存在时，调用append_object会创建一个可追加的文件。
- 文件存在时，调用append_object会向文件末尾追加内容。

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
# 创建可追加的文件
bucket.append_object('my-object', 0) {}

# 向文件末尾追加内容
next_pos = bucket.append_object('my-object', 0) do |stream|
  100.times { |i| stream << i.to_s }
end
```

```
next_pos = bucket.append_object('my-object', next_pos, :file => 'local-file-1')
next_pos = bucket.append_object('my-object', next_pos, :file => 'local-file-2')
```

说明：

- 只能向可追加的文件（即通过append_object创建的文件）追加内容。
- 可追加的文件不能被拷贝。

上传回调

用户在上传文件时可以指定上传回调，这样在文件上传成功后OSS会向用户提供的服务器地址发起一个HTTP POST请求，相当于一个通知机制。用户可以在收 到回调的时候做相应的动作。

目前OSS支持上传回调的接口只有put_object和resumable_upload。

上面的例子使用put_object上传了一个文件，并指定了上传回调并将此次上传 的bucket和object信息添加在body中，应用服务器收到这个回调后，就知道这个文件已经成功上传到OSS了。

resumable_upload的使用方法类似：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

callback = Aliyun::OSS::Callback.new(
  url: 'http://10.101.168.94:1234/callback',
  query: {user: 'put_object'},
  body: 'bucket=${bucket}&object=${object}'
)

begin
  bucket.resumable_upload('files/hello', '/tmp/x', callback: callback)
rescue Aliyun::OSS::CallbackError => e
  puts "Callback failed: #{e.message}"
end
```

说明：

- callback的url**不能**包含query string，而应该在**:query**参数中指定。
- 可能出现文件上传成功，但是执行回调失败的情况，此时client会抛出 CallbackError，用户如果要忽略此错误，需要显式接住这个异常。
- 详细的例子可以参考*callback.rb*。
- 接受回调的server可以参考*callback_server.rb*。

1.2.9.5 下载文件

OSS Ruby SDK提供了丰富的文件下载接口，用户可以通过以下方式从OSS中下载文件：

- 下载到本地文件
- 流式下载
- 断点续传下载
- HTTP下载（浏览器下载）

下载到本地文件

通过Bucket#get_object接口，并指定**:file**参数来下载到一个本地文件到：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.get_object('my-object', :file => 'local-file')
```

流式下载

在进行大文件下载时，往往不希望一次性处理全部的内容，而是希望流式地处理，一次处理一部分内容。 通过Bucket#get_object接口，并指定**block**参数来流式处理下载的内容：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.get_object('my-object') do |chunk|
  # handle_data(chunk)
  puts "Got a chunk, size: #{chunk.size}."
end
```

断点续传下载

当下载大文件时，如果网络不稳定或者程序崩溃了，则整个下载就失败了。用户不得不重头再来，这样做不仅浪费资源，在网络不稳定的情况下，往往重试多次还是无法完成下载。 通过Bucket#resumable_download接口来实现断点续传下载。它有以下参数：

- key 要下载的Object名字
- file 下载到本地文件的路径
- opts 可选项，主要包括：

- :cpt_file：指定checkpoint文件的路径，如果不指定则默认为与本地文件同目录下的*file.cpt*，其中*file*是本地文件的名字
- :disable_cpt：如果指定为true，则下载过程中不会记录下载进度，失败后也无法进行续传
- :part_size：指定每个分片的大小，默认为10MB
- &block：如果调用时候传递了block，则下载进度会交由block处理

详细的参数请参考*API文档*。

其实现的原理是将要下载的Object分成若干个分片分别下载，最后所有分片都下载成功后，完成整个文件的下载。在下载的过程中会记录当前下载的进度信息（记录在checkpoint文件中）和已下载的分片（保存为*file.part.N*，其中 *file* 是下载的本地文件的名字），如果下载过程中某一分片下载失败，再次下载时会从checkpoint文件中记录的点继续下载。这要求再次调用时要指定与上次 相同的checkpoint文件。下载完成后，part文件和checkpoint文件都会被删除。

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.resumable_download('my-object', 'local-file') do |p|
  puts "Progress: #{p}"
end

bucket.resumable_download(
  'my-object', 'local-file',
  :part_size => 100 * 1024, :cpt_file => '/tmp/x.cpt') { |p|
  puts "Progress: #{p}"
}
```

说明：

- SDK会将下载的中间状态信息记录在cpt文件中，所以要确保用户对cpt文件有写权限。
- SDK会将已下载的分片保存在part文件中，所以要确保用户对file所在的目录有创建文件的权限。
- cpt文件记录了下载的中间状态信息并自带了校验，用户不能去编辑它，如果cpt文件损坏则下载无法继续。
- 如果下载过程中待下载的Object发生了改变（ETag改变），或者part文件丢失或被修改，则下载会报错。

HTTP下载

对于存放在OSS中的文件，在不用SDK的情况下用户也可以直接使用HTTP下载，这包括直接使用浏览器下载，或者使用wget、curl等命令行工具下载。这时文 件的URL需要由SDK生成。使用Bucket#object_url方法生成可下载的HTTP地址，它接受以下参数：

- key 待下载的Object的名字
- sign 是否生成带签名的URL，对于拥有public-read/public-read-write权限 的Object，不带签名的URL也可以访问；对于private权限的Object，则必须使用带签名的URL才能访问
- expiry URL的有效时间，默认为60s

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
# 生成URL，默认带签名，有效时间为60秒
puts bucket.object_url('my-object')
# http://my-bucket.oss-cn-hangzhou.aliyuncs.com/my-object?Expires=1448349966&
OSSAccessKeyId=5viOHfldyK6K72ht&Signature=aM2HpBLeMq1aec6JCd7BBAKYiwI%3D

# 不带签名的URL
puts bucket.object_url('my-object', false)
# http://my-bucket.oss-cn-hangzhou.aliyuncs.com/my-object

# 指定URL过期时间为1小时（3600秒）
puts bucket.object_url('my-object', true, 3600)
```

1.2.9.6 管理文件

一个Bucket下可能有非常多的文件，SDK提供一系列的接口方便用户管理文件。

查看所有文件

通过Bucket#list_objects来列出当前Bucket下的所有文件。主要的参数如下：

- :prefix：指定只列出符合特定前缀的文件
- :marker：指定只列出文件名大于marker之后的文件
- :delimiter：用于获取文件的公共前缀

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
# 列出所有文件
objects = bucket.list_objects
```

```
objects.each { |o| puts o.key }

# 列出前缀为'my-'的所有文件
objects = bucket.list_objects(:prefix => 'my-')
objects.each { |o| puts o.key }

# 列出前缀为'my-'且在'my-object'之后的所有文件
objects = bucket.list_objects(:prefix => 'my-', :marker => 'my-object')
objects.each { |o| puts o.key }
```

模拟目录结构

OSS是基于对象的存储服务，没有目录的概念。存储在一个Bucket中所有文件都 是通过文件的key唯一标识，并没有层级的结构。这种结构可以让OSS的存储非常高效，但是用户管理文件时希望能够像传统的文件系统一样把文件分门别类放到不同的**目录**下面。通过OSS提供的**公共前缀**的功能，也可以很方便地模拟目录 结构。

假设Bucket中已有如下文件：

```
foo/x
foo/y
foo/bar/a
foo/bar/b
foo/hello/C/1
foo/hello/C/2
...
foo/hello/C/9999
```

接下来我们实现一个函数叫`list_dir`，列出指定目录下的文件和子目录：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

def list_dir(dir)
  objects = bucket.list_objects(:prefix => dir, :delimiter => '/')
  objects.each do |obj|
    if obj.is_a?(OSS::Object) # object
      puts "Object: #{obj.key}"
    else # common prefix
      puts "SubDir: #{obj}"
    end
  end
end
```

运行结果如下：

```
> list_dir('foo/')
=> SubDir: foo/bar/
   SubDir: foo/hello/
   Object: foo/x
```

```
  Object: foo/y

> list_dir('foo/bar/')
=> Object: foo/bar/a
  Object: foo/bar/b

> list_dir('foo/hello/C/')
=> Object: foo/hello/C/1
  Object: foo/hello/C/2
  ...
  Object: foo/hello/C/9999
```

文件元信息

向OSS上传文件时，除了文件内容，还可以指定文件的一些属性信息，称为**元信息**。这些信息在上传时与文件一起存储，在下载时与文件一起返回。

在SDK中文件元信息用一个**Hash**表示，其他key和value都是**String**类型，并 且都**只能是简单的ASCII可见字符**，**不能包含换行。** 所有元信息的总大小不能超过8KB。

> **说明**：
> 因为文件元信息在上传/下载时是附在HTTP Headers中， HTTP协议规定不能包含复杂字符。

使用Bucket#put_object，Bucket#append_object和 Bucket#resumable_upload 时都可以通过指定**：metas** 参数来指定文件的元信息：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

bucket.put_object(
  'my-object-1',
  :file => 'local-file',
  :metas => {'year' => '2016', 'people' => 'mary'})

bucket.append_object(
  'my-object-2', 0,
  :file => 'local-file',
  :metas => {'year' => '2016', 'people' => 'mary'})

bucket.resumable_upload(
  'my-object',
  'local-file',
  :metas => {'year' => '2016', 'people' => 'mary'})
```

通过Bucket#update_object_metas命令来更新文件元信息：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
```

```
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

bucket.update_object_metas('my-object', {'year' => '2017'})
```

拷贝文件

使用Bucket#copy_object拷贝一个文件。拷贝时对文件元信息的处理有两种选 择，通过**:meta_directive**参数指定：

- 与源文件相同，即拷贝源文件的元信息
- 使用新的元信息覆盖源文件的信息

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

# 拷贝文件元信息
bucket.copy_object(
  'my-object', 'copy-object',
  :meta_directive => Aliyun::OSS::MetaDirective::COPY)

# 覆盖文件元信息
bucket.copy_object(
  'my-object', 'copy-object',
  :metas => {'year' => '2017'},
  :meta_directive => Aliyun::OSS::MetaDirective::REPLACE)
```

删除文件

通过Bucket#delete_object来删除某个文件：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

bucket.delete_object('my-object')
```

批量删除文件

通过Bucket#batch_delete_object来删除一批文件，用户可以通过**:quiet**参数来指定是否返回删除的结果：

```
require 'aliyun/oss'
```

```
client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

objs = ['my-object-1', 'my-object-2']
result = bucket.batch_delete_object(objs)
puts result #['my-object-1', 'my-object-2']，默认返回删除成功的文件

objs = ['my-object-3', 'my-object-4']
result = bucket.batch_delete_object(objs, :quiet => true)
puts result #[]，不返回删除的结果
```

1.2.9.7 Rails应用

在Rails应用中使用OSS Ruby SDK只需要在Gemfile中添加以下依赖：

```
gem 'aliyun-sdk', '~> 0.3.0
```

然后在使用OSS时引入依赖就可以了：

```
require 'aliyun/oss'
```

另外，SDK的rails/目录下提供一些方便用户使用的辅助代码。

下面我们将利用SDK来实现一个简单的OSS文件管理器（oss-manager），最终包含以下功能：

- 列出用户所有的Bucket
- 列出Bucket下所有的文件，按目录层级列出
- 上传文件
- 下载文件

1. 创建项目

先安装Rails，然后创建一个Rails应用，oss-manager：

```
gem install rails
rails new oss-manager
```

作为一个好的习惯，使用git管理项目代码：

```
cd oss-manager
git init
git add .
```

```
git commit -m "init project"
```

2. 添加SDK依赖

编辑oss-manager/Gemfile，向其中加入SDK的依赖：

```
gem 'aliyun-sdk', '~> 0.3.0'
```

然后在oss-manager/下执行：

```
bundle install
```

保存这一步所做的更改：

```
git add .
git commit -m "add aliyun-sdk dependency"
```

3. 初始化OSS Client

为了避免在项目中用到OSS Client的地方都要初始化，我们在项目中添加一个初始化文件，方便在项目中使用OSS Client：

```
# oss-manager/config/initializers/aliyun_oss_init.rb
require 'aliyun/oss'

module OSS
  def self.client
    unless @client
      Aliyun::Common::Logging.set_log_file('./log/oss_sdk.log')

      @client = Aliyun::OSS::Client.new(
        endpoint:
          Rails.application.secrets.aliyun_oss['endpoint'],
        access_key_id:
          Rails.application.secrets.aliyun_oss['access_key_id'],
        access_key_secret:
          Rails.application.secrets.aliyun_oss['access_key_secret']
      )
    end

    @client
  end
end
```

上面的代码在SDK的rails/目录下可以找到。这样初始化后，在项目中使用OSS Client就非常方便：

```
buckets = OSS.client.list_buckets
```

其中endpoint和AccessKeyId/AccessKeySecret保存在 *oss-manager/conf/secrets.yml*中，例如：

```
development:
  secret_key_base: xxxx
  aliyun_oss:
    endpoint: xxxx
```

```
  access_key_id: aaaa
  access_key_secret: bbbb
```

保存代码：

```
git add .
git commit -m "add aliyun-sdk initializer"
```

4. 实现List buckets功能

首先用rails生成管理Buckets的controller：

```
rails g controller buckets index
```

这样会在oss-manager中生成以下文件：

- app/controller/buckets_controller.rb Buckets相关的逻辑代码
- app/views/buckets/index.html.erb Buckets相关的展示代码
- app/helpers/buckets_helper.rb 一些辅助函数

首先编辑buckets_controller.rb，调用OSS Client，将list_buckets的结果存放在@*buckets*变量中：

```
class BucketsController < ApplicationController
  def index
    @buckets = OSS.client.list_buckets
  end
end
```

然后编辑views/buckets/index.html.erb，将Bucket列表展示出来：

```
<h1>Buckets</h1>
<table class="table table-striped">
  <tr>
    <th>Name</th>
    <th>Location</th>
    <th>CreationTime</th>
  </tr>

  <% @buckets.each do |bucket| %>
  <tr>
    <td><%= link_to bucket.name, bucket_objects_path(bucket.name) %></td>
    <td><%= bucket.location %></td>
    <td><%= bucket.creation_time.localtime.to_s %></td>
  </tr>
  <% end %>
</table>
```

其中bucket_objects_path是一个辅助函数，在 app/helpers/buckets_helper.rb中：

```
module BucketsHelper
  def bucket_objects_path(bucket_name)
    "/buckets/#{bucket_name}/objects"
  end
```

```
end
```

这样就完成了列出所有Bucket的功能。在运行之前，我们还需要配置Rails 的路由，使得我们在浏览器中输入的地址能够调用正确的逻辑。编辑`config/routes.rb`，增加一条：

```
resources :buckets do
  resources :objects
end
```

好了，在oss-manager/下输入`rails s`以启动rails server，然后在浏览器中输入 *http://localhost:3000/buckets/* 就能看到Bucket列表了。

最后保存一下代码：

```
git add .
git commit -m "add list buckets feature"
```

5. 实现List objects功能

首先生成一个管理Objects的controller：

```
rails g controller objects index
```

然后编辑app/controllers/objects_controller.rb:

```
class ObjectsController < ApplicationController
  def index
    @bucket_name = params[:bucket_id]
    @prefix = params[:prefix]
    @bucket = OSS.client.get_bucket(@bucket_name)
    @objects = @bucket.list_objects(:prefix => @prefix, :delimiter => '/')
  end
end
```

上面的代码首先从URL的参数中获取Bucket名字，为了只按目录层级显示，我们还需要一个前缀。然后调用OSS Client的list_objects接口获取文件列表。注意， 这里获取的是指定前缀下，并且以'/'为分界的文件。这样做是为也按目录层级列出文件。

接下来编辑app/views/objects/index.html.erb：

```
<h1>Objects in <%= @bucket_name %></h1>
<p> <%= link_to 'Upload file', new_object_path(@bucket_name, @prefix) %></p>
<table class="table table-striped">
  <tr>
    <th>Key</th>
    <th>Type</th>
    <th>Size</th>
    <th>LastModified</th>
  </tr>

  <tr>
    <td><%= link_to '../', with_prefix(upper_dir(@prefix)) %></td>
```

```
    <td>Directory</td>
    <td>N/A</td>
    <td>N/A</td>
  </tr>

  <% @objects.each do |object| %>
  <tr>
    <% if object.is_a?(Aliyun::OSS::Object) %>
    <td><%= link_to remove_prefix(object.key, @prefix),
          @bucket.object_url(object.key) %></td>
    <td><%= object.type %></td>
    <td><%= number_to_human_size(object.size) %></td>
    <td><%= object.last_modified.localtime.to_s %></td>
    <% else  %>
    <td><%= link_to remove_prefix(object, @prefix), with_prefix(object) %></td>
    <td>Directory</td>
    <td>N/A</td>
    <td>N/A</td>
    <% end  %>
  </tr>
  <% end %>
</table>
```

上面的代码将文件按目录结构显示，主要逻辑是：

- 总是在第一个显示'../'指向上级目录
- 对于Common prefix，显示为目录
- 对于Object，显示为文件

上面的代码中用到了 `with_prefix`、 `remove_prefix` 等一些辅助函数，它们定义在app/helpers/objects_helper.rb中：

```
module ObjectsHelper
  def with_prefix(prefix)
    "?prefix=#{prefix}"
  end

  def remove_prefix(key, prefix)
    key.sub(/^#{prefix}/, '')
  end

  def upper_dir(dir)
    dir.sub(/[^\/]+\/$/, '') if dir
  end

  def new_object_path(bucket_name, prefix = nil)
    "/buckets/#{bucket_name}/objects/new/#{with_prefix(prefix)}"
  end

  def objects_path(bucket_name, prefix = nil)
    "/buckets/#{bucket_name}/objects/#{with_prefix(prefix)}"
  end
```

```
end
```

完成之后运行rails s，然后在浏览器中输入地址 *http://localhost:3000/buckets/my-bucket/objects/*就可以查看文件列表了。

惯例保存代码：

```
git add .
git commit -m "add list objects feature"
```

6. 下载文件

注意到在上一步显示文件列表时，我们为每个文件也添加了一个链接：

```
<td><%= link_to remove_prefix(object.key, @prefix),
        @bucket.object_url(object.key) %></td>
```

其中Bucket#object_url是一个为文件生成临时URL的方法。

7. 上传文件

在Rails这种服务端应用中，用户上传文件有两种办法：

1. 用户先将文件上传到Rails的服务器上，服务器再将文件上传到OSS。这样做需要Rails服务器作为中转，文件多拷贝了一遍，不是很高效。
2. 服务器为用户生成表单和临时凭证，用户直接上传文件到OSS。

第一种方法比较简单，与普通的上传文件一样。下面我们用的是第二种方法：

首先在app/controllers/objects_controller.rb中增加一个#new方法，用于生成上传表单：

```
def new
  @bucket_name = params[:bucket_id]
  @prefix = params[:prefix]
  @bucket = OSS.client.get_bucket(@bucket_name)
  @options = {
    :prefix => @prefix,
    :redirect => 'http://localhost:3000/buckets/'
  }
end
```

然后编辑app/views/objects/new.html.erb：

```
<h2>Upload object</h2>

<%= upload_form(@bucket, @options) do %>
  <table class="table table-striped">
    <tr>
      <td><label>Bucket:</label></td>
      <td><%= @bucket.name  %></td>
    </tr>
    <tr>
```

```
      <td><label>Prefix:</label></td>
      <td><%= @prefix  %></td>
    </tr>

    <tr>
      <td><label>Select file:</label></td>
      <td><input type="file" name="file" style="display:inline" /></td>
    </tr>

    <tr>
      <td colspan="2">
        <input type="submit" class="btn btn-default" value="Upload" />
        <span>  </span>
        <%= link_to 'Back', objects_path(@bucket_name, @prefix) %>
      </td>
    </tr>
  </table>
<% end %>
```

其中upload_form是SDK提供的一个方便用户生成上传表单的辅助函数，在SDK 的代码rails/aliyun_oss_helper.rb中。用户需要将其拷贝到app/helpers/目录下。完成之后运行rails s，然后访问*http://localhost:3000/buckets/my-bucket/objects/new*就能够上传文件了。

最后记得保存代码：

```
git add .
git commit -m "add upload object feature"
```

8. 添加样式

为了让界面更好看一些，我们可以添加一点样式（CSS）。

首先下载bootstrap，解压后将 *bootstrap.min.css*拷贝到*app/assets/stylesheets/*下。

然后在修改app/views/layouts/application.html.erb，将yield一行改成：

```
  <div id="main">
    <%= yield %>
  </div>
```

这会为每个页面添加一个id为main的 ，然后修改app/assets/stylesheets/application.css，加入以下内容：

```
body {
    text-align: center;
}

div#main {
    text-align: left;
    width: 1024px;
    margin: 0 auto;
}
```

这会让网页的主体内容居中显示。通过添加简单的样式，我们的页面是不是更加赏心悦目了呢？

至此，一个简单的demo就完成了。完整的demo代码可以在 *OSS Ruby SDK Demo* 中找到。

1.2.9.8 自定义域名绑定

OSS支持用户将自定义的域名绑定到OSS服务上，这样能够支持用户无缝地将存储迁移到OSS上。例如用户的域名是my-domain.com，之前用户的所有图片资源都是形如http://img.my-domain.com/x.jpg的格式，用户将图片存储迁移到OSS之 后，通过绑定自定义域名，仍可以使用原来的地址访问到图片：

- 开通OSS服务并创建Bucket
- 将img.my-domain.com与创建的Bucket绑定
- 将图片上传到OSS的这个Bucket中
- 修改域名的DNS配置，增加一个CNAME记录，将img.my-domain.com指向OSS服务的endpoint（如my-bucket.oss-cn-hangzhou.aliyuncs.com）

这样就可以通过原地址http://img.my-domain.com/x.jpg访问到存储在OSS上的图片。

在使用SDK时，也可以使用自定义域名作为endpoint，这时需要将**:cname**参数设置为true，如下面的例子：

```
require 'aliyun/oss'

include Aliyun::OSS

client = Client.new(
  endpoint: 'ENDPOINT',
  access_key_id: 'ACCESS_KEY_ID',
  access_key_secret: 'ACCESS_KEY_SECRET',
  cname: true)

bucket = client.get_bucket('my-bucket')
```

说明：

使用CNAME时，无法使用list_buckets接口。（因为自定义域名已经绑定到某个特定的Bucket。）

1.2.9.9 使用STS访问

OSS可以通过阿里云STS服务，临时进行授权访问。

1. 在官网控制台创建子账号。
2. 在官网控制台创建STS角色并赋予子账号扮演角色的权限。
3. 使用子账号的AccessKeyId/AccessKeySecret向STS申请临时token
4. 使用临时token中的认证信息创建OSS的Client

5. 使用OSS的Client访问OSS服务

在使用STS访问OSS时，需要设置**:sts_token**参数，如下面的例子所示：

```
require 'aliyun/sts
require 'aliyun/oss'

sts = Aliyun::STS::Client.new(
  access_key_id: '<子账号的AccessKeyId>',
  access_key_secret: '<子账号的AccessKeySecret>')

token = sts.assume_role('<role-arn>', '<session-name>')

client = Aliyun::OSS::Client.new(
  endpoint: '<endpoint>',
  access_key_id: token.access_key_id,
  access_key_secret: token.access_key_secret,
  sts_token: token.security_token)

bucket = client.get_bucket('my-bucket')
```

在向STS申请临时token时，还可以指定自定义的STS Policy。这样申请的临时权限是**所扮演角色的权限与Policy指定的权限的交集**。下面的例子将通过指定 STS Policy申请对my-bucket的只读权限，并指定临时token的过期时间为15分钟：

```
require 'aliyun/sts
require 'aliyun/oss'

sts = Aliyun::STS::Client.new(
  access_key_id: '<子账号的AccessKeyId>',
  access_key_secret: '<子账号的AccessKeySecret>')

policy = Aliyun::STS::Policy.new
policy.allow(['oss:Get*'], ['acs:oss:*:*:my-bucket/*'])

token = sts.assume_role('<role arc>', '<session name>', policy, 15 * 60)

client = Aliyun::OSS::Client.new(
  endpoint: 'ENDPOINT',
  access_key_id: token.access_key_id,
  access_key_secret: token.access_key_secret,
  sts_token: token.security_token)

bucket = client.get_bucket('my-bucket')
```

更详细的用法和参数说明请参考*API文档*。

1.2.9.10 设置访问权限

OSS允许用户对Bucket和Object分别设置访问权限，方便用户控制自己的资源可以被如何访问。对于Bucket，有三种访问权限：

- public-read-write 允许匿名用户向该Bucket中创建/获取/删除Object

- public-read 允许匿名用户获取该Bucket中的Object
- private 不允许匿名访问，所有的访问都要经过签名

创建Bucket时，默认是private权限。之后用户可以通过bucket.acl=来设置 Bucket的权限。

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
puts bucket.acl
```

对于Object，有四种访问权限：

- default 继承所属的Bucket的访问权限，即与所属Bucket的权限值一样
- public-read-write 允许匿名用户读写该Object
- public-read 允许匿名用户读该Object
- private 不允许匿名访问，所有的访问都要经过签名

创建Object时，默认为default权限。之后用户可以通过 bucket.set_object_acl来设置Object的权限。

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')

acl = bucket.get_object_acl('my-object')
puts acl # default
bucket.set_object_acl('my-object', Aliyun::OSS::ACL::PUBLIC_READ)
acl = bucket.get_object_acl('my-object')
puts acl # public-read
```

需要注意的是：

- 如果设置了Object的权限（非default），则访问该Object时进行权限认证时会优先判断Object的权限，而Bucket的权限设置会被忽略。

- 允许匿名访问时（设置了public-read或者public-read-write权限），用户 可以直接通过浏览器访问，例如：

```
http://bucket-name.oss-cn-hangzhou.aliyuncs.com/object.jpg
```

- 访问具有public权限的Bucket/Object时，也可以通过创建匿名的Client来进行：

```
require 'aliyun/oss'

# 不填access_key_id和access_key_secret，将创建匿名Client，只能访问具有
# public权限的Bucket/Object
client = Aliyun::OSS::Client.new(endpoint: 'endpoint')
bucket = client.get_bucket('my-bucket')

bucket.get_object('my-object', :file => 'local_file')
```

1.2.9.11 管理生命周期

OSS允许用户对Bucket设置生命周期规则，以自动淘汰过期掉的文件，节省存储空间。用户可以同时设置多条规则，一条规则包含：

- 规则ID，用于标识一条规则，不能重复
- 受影响的文件前缀，此规则只作用于符合前缀的文件
- 过期时间，有两种指定方式：
  1. 指定距文件最后修改时间N天过期
  2. 指定在具体的某一天过期，即在那天之后符合前缀的文件将会过期，**而不论文件的最后修改时间**。不推荐使用。
- 是否生效

设置生命周期规则

通过`Bucket#lifecycle=`来设置生命周期规则：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket.lifecycle = [
  LifeCycleRule.new(
    :id => 'rule1', :enabled => true, :prefix => 'foo/', :expiry => 3),
  LifeCycleRule.new(
    :id => 'rule2', :enabled => false, :prefix => 'bar/', :expiry => Date.new(2016, 1, 1))
```

```
]
```

查看生命周期规则

通过Bucket#lifecycle来查看生命周期规则：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

rules = bucket.lifecycle
puts rules
```

清空生命周期规则

通过Bucket#lifecycle=设置一个空的Rule数组来清空生命周期规则：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket.lifecycle = []
```

1.2.9.12 设置访问日志

OSS允许用户对Bucket设置访问日志记录，设置之后对于Bucket的访问会被记录成日志，日志存储在OSS上由用户指定的Bucket中，文件的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中TargetPrefix由用户指定。日志规则由以下3项组成：

- enable，是否开启
- target_bucket，存放日志文件的Bucket
- target_prefix，日志文件的前缀

开启Bucket日志

通过Bucket#logging=来开启日志功能：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
```

```
bucket.logging = BucketLogging.new(
  enable: true, target_bucket: 'logging_bucket', target_prefix: 'my-log')
```

查看Bucket日志设置

通过Bucket#logging来查看日志设置：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
log = bucket.logging
puts log.to_s
```

关闭Bucket日志

通过Bucket#logging=来关闭日志功能：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.logging = BucketLogging.new(enable: false)
```

1.2.9.13 静态网站托管

在自定义域名绑定中提到，OSS 允许用户将自己的域名指向OSS服务的地址。这样用户访问他的网站的时候，实 际上是在访问OSS的Bucket。对于网站，需要指定首页(index)和出错页(error) 分别对应的Bucket中的文件名。

更多关于静态网站托管的内容请参考 OSS静态网站托管。

设置托管页面

通过Bucket#website=来设置托管页面：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
```

```
bucket.website = BucketWebsite.new(index: 'index.html', error: 'error.html')
```

查看托管页面

通过Bucket#website来查看托管页面：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
web = bucket.website
puts web.to_s
```

清除托管页面

通过Bucket#website=来清除托管页面：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.website = BucketWebsite.new(enable: false)
```

1.2.9.14 设置防盗链

OSS是按使用收费的服务，为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

设置Referer白名单

通过Bucket#referer=设置Referer白名单：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.referer = BucketReferer.new(
  allow_empty: true, whitelist: ['my-domain.com', '*.example.com'])
```

查看Referer白名单

通过Bucket#referer设置Referer白名单：

```
require 'aliyun/oss'
```

```
client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
ref = bucket.referer
puts ref.to_s
```

清空Referer白名单

通过`Bucket#referer=`设置清空Referer白名单：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.referer = BucketReferer.new(allow_empty: true, whitelist: [])
```

1.2.9.15 设置跨域资源共享

跨域资源共享(CORS)允许web端的应用程序访问不属于本域的资源。OSS提供接口方便开发者控制跨域访问的权限。

OSS的跨域共享设置由一条或多条CORS规则组成，每条CORS规则包含以下设置：

- allowed_origins，允许的跨域请求的来源，如www.my-domain.com, *
- allowed_methods，允许的跨域请求的HTTP方法(PUT/POST/GET/DELETE/HEAD)
- allowed_headers，在OPTIONS预取指令中允许的header，如x-oss-test, *
- expose_headers，允许用户从应用程序中访问的响应头
- max_age_seconds, 浏览器对特定资源的预取（OPTIONS）请求返回结果的缓存时间

设置CORS规则

通过`Bucket#cors=`设置CORS规则：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.cors = [
    CORSRule.new(
      :allowed_origins => ['aliyun.com', 'http://www.taobao.com'],
      :allowed_methods => ['PUT', 'POST', 'GET'],
      :allowed_headers => ['Authorization'],
      :expose_headers => ['x-oss-test'],
      :max_age_seconds => 100)
```

```
]
```

查看CORS规则

通过Bucket#cors查看CORS规则：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
cors = bucket.cors
puts cors.map(&:to_s)
```

清空CORS规则

通过Bucket#cors=清空CORS规则：

```
require 'aliyun/oss'

client = Aliyun::OSS::Client.new(
  endpoint: 'endpoint',
  access_key_id: 'AccessKeyId', access_key_secret: 'AccessKeySecret')

bucket = client.get_bucket('my-bucket')
bucket.cors = []
```

1.2.9.16 异常

使用SDK时如果请求出错，会有相应的异常抛出，同时在log（默认为程序运行目录下oss_sdk.log）中也会记录详细的出错信息。

OSS Ruby SDK中有两种异常：ClientError和ServerError，它们都是 RuntimeError的子类。

ClientError

ClientError指SDK内部出现的异常，比如参数设置错误或者断点上传/下载中出现的文件被修改的错误。

ServerError

ServerError指服务器端错误，它来自于对服务器错误信息的解析。ServerError 有以下几个属性：

- http_code: 出错请求的HTTP状态码
- error_code: OSS的错误码
- message: OSS的错误信息

- request_id: 标识该次请求的UUID；当您无法解决问题时，可以凭这个RequestId来请求OSS开发工程师的帮助

1.2.10 Go-SDK

1.2.10.1 安装

- github地址：*https://github.com/aliyun/aliyun-oss-go-sdk*
- API文档地址：*https://godoc.org/github.com/aliyun/aliyun-oss-go-sdk/oss*
- 版本迭代：*https://github.com/aliyun/aliyun-oss-go-sdk/releases*

要求

- 开通阿里云OSS服务，并创建了AccessKeyId 和AccessKeySecret。
- 如果您还没有开通或者还不了解阿里云OSS服务，请登录OSS产品主页了解。
- 如果还没有创建AccessKeyId和AccessKeySecret，请到阿里云Access Key管理创建Access Key。
- 您已经安装了Go编译运行环境。如果您未安装，请参考Go安装下载安装编译运行环境。下载安装Go编译运行环境。Go安装完毕后请正确设置GOPATH变量，如果您需要了解更多GOPATH，请执行命令go help gopath。

安装

GitHub安装

- 执行命令go get github.com/aliyun/aliyun-oss-go-sdk/oss获取远程代码包。
- 在您的代码中使用import "github.com/aliyun/aliyun-oss-go-sdk/oss"引入OSS Go SDK的包。
- 如果您需要查看示例程序，请到*https://github.com/aliyun/aliyun-oss-go-sdk/*下的sample目录。

说明：

- 使用go get命令安装过程中，界面不会打印提示，如果网络较差，需要一段时间，请耐心等待。如果安装过程中发生超时，请再次执行go get安装。
- 安装成功后，在您的安装路径下（即GOPATH变量中的第一个路径），会有Go Sdk的库*pkg/linux_amd64/github.com/aliyun/aliyun-oss-go-sdk/oss.a* (win在*pkg\windows_amd64\github.com\aliyun\aliyun-oss-go-sdk\oss.a*)，源文件在*src/github.com/aliyun/aliyun-oss-go-sdk*，如果没有请重新安装。

示例

运行示例

- 拷贝示例文件。到OSS Go SDK的安装路径（即GOPATH变量中的第一个路径），进入OSS Go SDK的代码目录*src\github.com\aliyun\aliyun-oss-go-sdk*，把其下的sample目录和sample.go复制到您的测试工程src目录下。
- 修改sample/config.go里的endpoint、AccessKeyId、AccessKeySecret、BucketName等配置。
- 请在您的工程目录下执行go run src/sample.go。

示例内容

Go SDK的示例代码在sample目录下，地址是 *GitHub* 。示例包括以下内容：

示例文件	示例内容
new_bucket.go	展示了如何初始化Client、Bucket
put_object.go	展示了简单上传、断点续传上传的用法
append_object.go	展示了追加上传的用法
get_object.go	展示了流式下载、范围下载、断点续传下载的用法
delete_object.go	展示了删除单个文件、批量删除文件的方法
copy_object.go	展示了文件拷贝、文件断点续传拷贝的用法
list_objects.go	展示了列举文件的用法，包括默认参数列举、指定参数列举
object_meta.go	展示了如何设置、读取文件元数据（Object Meta）
object_acl.go	展示了如何设置、读取文件权限（Object ACL）
cname_sample.go	展示了CNAME的用法
create_bucket.go	展示了如何创建存储空间
list_buckets.go	展示了列举存储空间的用法，包括默认参数列举、指定参数列举
bucket_acl.go	展示了如何读取/设置存储空间的权限（Bucket ACL）
bucket_referer.go	展示了如何设置/读取/清除存储空间的白名单（Bucket Referer）
bucket_logging.go	展示了如何设置/读取/清除存储空间的日志（Bucket Logging）

示例文件	示例内容
bucket_lifecycle.go	展示了如何设置/读取/清除存储空间中文件的生命周期（Bucket Lifecycle）
bucket_cors.go	展示了如何设置/读取/清除存储空间的跨域访问（Bucket CORS）

1.2.10.2 快速开始

下面介绍如何使用OSS Go SDK来访问OSS服务，包括查看Bucket列表，查看文件列表，上传/下载文件和删除文件。使用OSS Go SDK，需要引入oss包import github.com/aliyun/aliyun-oss-go-sdk/oss。

初始化Client

初始化Client，即创建Client：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}
```

说明：

- Endpoint的是OSS的访问域名，如杭州数据中的访问域名是*http://oss-cn-hangzhou.aliyuncs.com*。
- AccessKeyId和AccessKeySecret是OSS的访问密钥。。
- 您运行示例程序时，请将Endpoint，AccessKeyId和AccessKeySecret替换成您的实际配置。

查看Bucket列表

通过Client.ListBuckets查看Bucket列表：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

lsRes, err := client.ListBuckets()
if err != nil {
    // HandleError(err)
}

for _, bucket := range lsRes.Buckets {
    fmt.Println("bucket:", bucket.Name)
```

```
}
```

说明：

- Bucket名字不能与OSS服务中其他用户已有的Bucket重复，所以你需要选择一个独特的Bucket名字以避免创建失败。

获取Bucket

Bucket的操作有Client的方法完成，如创建/删除Bucket、设置/清除Bucket的权限/生命周期/防盗链等，Object的操作有Bucket的方法完成，如上传/下载/删除文件、设置Object的访问权限等。用户可以通过Client.Bucket获取指定Bucket的操作句柄。

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}
```

查看文件列表

通过Bucket.ListObjects查看Bucket中的文件列表：

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

lsRes, err := bucket.ListObjects()
if err != nil {
    // HandleError(err)
}

for _, object := range lsRes.Objects {
    fmt.Println("Object:", object.Key)
```

```
}
```

上传文件

通过Bucket.PutObjectFromFile上传文件：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.PutObjectFromFile("my-object", "LocalFile")
if err != nil {
    // HandleError(err)
}
```

其中*LocalFile*是需要上传的本地文件的路径。上传成功后，可以通过Bucket.ListObjects来查看。

下载文件

通过Bucket.GetObject下载文件：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.GetObjectToFile("my-object", "LocalFile")
if err != nil {
    // HandleError(err)
}
```

其中*LocalFile*是文件保存的路径。下载成功后，可以打开文件查看其内容。

删除文件

通过Bucket.DeleteObject从Bucket中删除文件：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
```

```
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.DeleteObject("my-object")
if err != nil {
    // HandleError(err)
}
```

删除文件后可以通过Bucket.ListObjects来查看文件确实已经被删除。

1.2.10.3 管理Bucket

存储空间（Bucket）是OSS上的命名空间，也是计费、权限控制、日志记录等高级功能的管理实体。

查看所有Bucket

使用Client.ListBuckets接口列出当前用户下的所有Bucket，用户还可以指定**Prefix**等参数，列出Bucket名字为特定前缀的所有Bucket：

说明：

ListBuckets的示例代码在sample/list_buckets.go。

```
import (
    "fmt"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

// 列出Bucket，默认100条。
lsRes, err := client.ListBuckets()
if err != nil {
    // HandleError(err)
}
fmt.Println("buckets:", lsRes.Buckets)

// 指定前缀筛选
lsRes, err = client.ListBuckets(oss.Prefix("my-bucket"))
if err != nil {
    // HandleError(err)
}
```

```
fmt.Println("buckets:", lsRes.Buckets)
```

创建Bucket

> **说明**：
>
> CreateBucket的示例代码在sample/create_bucket.go。

使用Client.CreateBucket接口创建一个Bucket，用户需要指定Bucket的名字：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.CreateBucket("my-bucket")
if err != nil {
    // HandleError(err)
}
```

创建Bucket时不指定权限，使用默认权限oss.ACLPrivate。创建时用户可以指定Bucket的权限：

```
err = client.CreateBucket("my-bucket", oss.ACL(oss.ACLPublicRead))
if err != nil {
    // HandleError(err)
}
```

> **说明**：
>
> - 由于存储空间的名字是全局唯一的，所以必须保证您的Bucket名字不与别人的重复。

删除Bucket

使用Client.DeleteBucket接口删除一个Bucket，用户需要指定Bucket的名字：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.DeleteBucket("my-bucket")
if err != nil {
    // HandleError(err)
}
```

> **说明**：
>
> - 如果该Bucket下还有文件存在，则需要先删除所有文件才能删除Bucket。

- 如果该Bucket下还有未完成的上传请求，则需要通过 Bucket.ListMultipartUploads 和 Bucket.AbortMultipartUpload 先取消那些请求才能删除Bucket。用法请参考 *API文档*。

查看Bucket是否存在

用户可以通过Client.IsBucketExist接口查看当前用户的某个Bucket是否存在：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

isExist, err := client.IsBucketExist("my-bucket")
if err != nil {
    // HandleError(err)
}
```

Bucket访问权限

用户可以设置Bucket的访问权限，允许或者禁止匿名用户对其内容进行读写。更多关于访问权限的内容请参考访问权限。

说明：

Bucket访问权限的示例代码sample/bucket_acl.go。

查看Bucket的访问权限

通过Client.GetBucketACL查看Bucket的ACL：

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

aclRes, err := client.GetBucketACL("my-bucket")
if err != nil {
    // HandleError(err)
}
fmt.Println("Bucket ACL:", aclRes.ACL)
```

设置Bucket的访问权限（ACL）

通过Client.SetBucketACL设置Bucket的ACL：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
```

```
if err != nil {
    // HandleError(err)
}

err = client.SetBucketACL("my-bucket", oss.ACLPublicRead)
if err != nil {
    // HandleError(err)
}
```

说明：

Bucket有三种权限：私有读写、公共读私有写、公共读写，分别对应Go sdk的常量ACLPrivate、ACLPublicRead、ACLPublicReadWrite。

1.2.10.4 上传文件

OSS Go SDK提供了丰富的文件上传接口，用户可以通过以下方式向OSS中上传文件：

- 简单上传PutObject，适合小文件
- 分片上传UploadFile，适合大文件
- 追加上传AppendObject

简单上传

从数据流(io.Reader)中读取数据上传

通过Bucket.PutObject完成简单上传。

说明：

简单上传的示例代码在sample/put_object.go。

字符串上传

```
import "strings"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.PutObject("my-object", strings.NewReader("MyObjectValue"))
if err != nil {
    // HandleError(err)
```

```
}
```

byte数组上传

```
import "bytes"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.PutObject("my-object", bytes.NewReader([]byte("MyObjectValue")))
if err != nil {
    // HandleError(err)
}
```

文件流上传

```
import "os"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

fd, err := os.Open("LocalFile")
if err != nil {
    // HandleError(err)
}
defer fd.Close()

err = bucket.PutObject("my-object", fd)
if err != nil {
    // HandleError(err)
}
```

根据本地文件名上传

通过Bucket.PutObjectFromFile可以上传指定的本地文件，把本地文件内容作为Object的值。

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}
```

```
bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.PutObjectFromFile("my-object", "LocalFile")
if err != nil {
    // HandleError(err)
}
```

说明：

使用上述方法上传最大文件不能超过5G。如果超过请使用分片上传。

上传时指定元信息

使用数据流上传文件时，用户可以指定一个或多个文件的元信息。元数据的名称大小写不敏感，比如用户上传文件时，定义名字为**name**的meta，使用Bucket.GetObjectDetailedMeta读取结果是：**X-Oss-Meta-Name**，比较/读取时请忽略大小写。

可以指定的元信息如下：

参数	说明
CacheControl	指定该Object被下载时的网页的缓存行为。
ContentDisposition	指定该Object被下载时的名称。
ContentEncoding	指定该Object被下载时的内容编码格式。
Expires	指定过期时间。用户自定义格式，建议使用http.TimeFormat格式。
ServerSideEncryption	指定oss创建object时的服务器端加密编码算法。合法值：AES256。
ObjectACL	指定oss创建object时的访问权限。
Meta	自定义参数，以**X-Oss-Meta-**为前缀的参数。

```
import (
    "strings"
    "time"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
```

```
}

expires := time.Date(2049, time.January, 10, 23, 0, 0, 0, time.UTC)
options := []oss.Option{
    oss.Expires(expires),
    oss.ObjectACL(oss.ACLPublicRead),
    oss.Meta("MyProp", "MyPropVal"),
}
err = bucket.PutObject("my-object", strings.NewReader("MyObjectValue"), options...)
if err != nil {
    // HandleError(err)
}
```

说明：

Bucket.PutObject、Bucket.PutObjectFromFile、Bucket.UploadFile、Bucket.UploadFile都支持上传时指定元数据。

创建模拟文件夹

OSS服务是没有文件夹这个概念的，所有元素都是以文件来存储。但给用户提供了创建模拟文件夹的方式，如下代码：

```
import "strings"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.PutObject("my-dir/", strings.NewReader(""))
if err != nil {
    // HandleError(err)
}
```

说明：

- 创建模拟文件夹本质上来说是创建了一个空文件。
- 对于这个文件照样可以上传下载，只是控制台会对以 / 结尾的文件以文件夹的方式展示。
- 所以用户可以使用上述方式来实现创建模拟文件夹。

追加上传

OSS支持可追加的文件类型，通过Bucket.AppendObject来上传可追加的文件， 调用时需要指定文件追加的位置，对于新创建文件，这个位置是0；对于已经存在的文件，这个位置必须是追加前文件的长度。

- 文件不存在时，调用AppendObject会创建一个可追加的文件；
- 文件存在时，调用AppendObject会向文件末尾追加内容。

说明：

追加上传的示例代码在sample/append_object.go。

```
import "strings"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

var nextPos int64 = 0
// 第一次追加的位置是0，返回值为下一次追加的位置
nextPos, err = bucket.AppendObject("my-object", strings.NewReader("YourObjectValue"),
nextPos)
if err != nil {
    // HandleError(err)
}

// 第二次追加
nextPos, err = bucket.AppendObject("my-object", strings.NewReader("YourObjectValue"),
nextPos)
if err != nil {
    // HandleError(err)
}

// 您还可以进行多次Append
```

说明：

- 只能向可追加的文件（即通过AppendObject创建的文件）追加内容。
- 可追加的文件不能被拷贝。

第一次追加时，即位置开始位置是0的追加，您可以指定文件(Object)的元信息；除了第一次追加，其他追加不能指定元信息。

```
    // 第一次追加指定元信息
    nextPos, err = bucket.AppendObject("my-object", strings.NewReader("YourObjectValue"), 0,
oss.Meta("MyProp", "MyPropVal"))
    if err != nil {
        // HandleError(err)
    }
```

分片上传

当上传大文件时，如果网络不稳定或者程序崩溃了，则整个上传就失败了。用户不得不重头再来，这样做不仅浪费资源，在网络不稳定的情况下，往往重试多次还是无法完成上传。 通过Bucket.UploadFile接口来实现断点续传上传。它有以下参数：

- objectKey 上传到OSS的Object名字
- filePath 待上传的本地文件路径
- partSize 分片上传大小，从100KB到5GB，单位是Byte
- options 可选项，主要包括：
  - Routines 指定分片上传的并发数，默认是1，及不使用并发上传
  - Checkpoint 指定上传是否开启断点续传，及checkpoint文件的路径。默认断点续传功能关闭，checkpoint文件的路径可以指定为空，为与本地文件同目录下的*file.cp*，其中*file*是本地文件的名字
  - 其他元信息，请参看上传时指定元信息

其实现的原理是将要上传的文件分成若干个分片分别上传，最后所有分片都上传成功后，完成整个文件的上传。在上传的过程中会记录当前上传的进度信息（记 录在checkpoint文件中），如果上传过程中某一分片上传失败，再次上传时会从 checkpoint文件中记录的点继续上传。这要求再次调用时要指定与上次相同的 checkpoint文件。上传完成后，checkpoint文件会被删除。

说明：

分片上传的示例代码在sample/put_object.go。

```
    import "github.com/aliyun/aliyun-oss-go-sdk/oss"

    client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
    if err != nil {
        // HandleError(err)
    }

    bucket, err := client.Bucket("my-bucket")
    if err != nil {
```

```
        // HandleError(err)
    }

    // 分片大小100K，3个协程并发上传分片，使用断点续传
    err = bucket.UploadFile("my-object", "LocalFile", 100*1024, oss.Routines(3), oss.Checkpoint
(true, ""))
    if err != nil {
        // HandleError(err)
    }
```

说明：

- SDK会将上传的中间状态信息记录在cp文件中，所以要确保用户对cp文件有写权限。
- cpt文件记录了上传的中间状态信息并自带了校验，用户不能去编辑它，如果cpt文件损坏则重新上传所有分片。整个上传完成后cpt文件会被删除。
- 如果上传过程中本地文件发生了改变，则重新上传所有分片。
- 指定断点续传checkpoint文件路径使用*oss.Checkpoint(true, "your-cp-file.cp")*。
- 使用*bucket.UploadFile(objectKey, localFile, 100*1024)*，默认不使用分片并发上传、不启动断点续传。

获取所有已上传的分片信息

您可以用Bucket.ListUploadedParts获取某个分片上传已上传的分片。

```
    import "fmt"
    import "github.com/aliyun/aliyun-oss-go-sdk/oss"

    client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
    if err != nil {
        // HandleError(err)
    }

    bucket, err := client.Bucket("my-bucket")
    if err != nil {
        // HandleError(err)
    }

    imur, err := bucket.InitiateMultipartUpload("my-object")
    if err != nil {
        // HandleError(err)
    }

    lsRes, err := bucket.ListUploadedParts(imur)
    if err != nil {
        // HandleError(err)
    }
    fmt.Println("Parts:", lsRes.UploadedParts)
```

获取所有分片上传的任务

通过Bucket.ListMultipartUploads来列出当前分片上传任务。主要的参数如下：

参数	说明
Delimiter	用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素。
MaxUploads	限定此次返回Multipart Uploads事件的最大数目，默认为1000，max-uploads取值不能大于1000。
KeyMarker	所有Object名字的字典序大于KeyMarker参数值的Multipart事件。
Prefix	限定返回的文件名(object)必须以Prefix作为前缀。注意使用Prefix查询时，返回的文件名(Object)中仍会包含Prefix。

使用默认参数

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

lsRes, err := bucket.ListMultipartUploads()
if err != nil {
    // HandleError(err)
}
fmt.Println("Uploads:", lsRes.Uploads)
```

指定前缀

```
lsRes, err := bucket.ListMultipartUploads(oss.Prefix("my-object-"))
if err != nil {
    // HandleError(err)
}
fmt.Println("Uploads:", lsRes.Uploads)
```

指定最多返回100条结果数据

```
lsRes, err := bucket.ListMultipartUploads(oss.MaxUploads(100))
if err != nil {
    // HandleError(err)
}
fmt.Println("Uploads:", lsRes.Uploads)
```

同时指定前缀和最大返回条数

```
lsRes, err := bucket.ListMultipartUploads(oss.Prefix("my-object-"), oss.MaxUploads(100))
if err != nil {
    // HandleError(err)
```

```
}
fmt.Println("Uploads:", lsRes.Uploads)
```

1.2.10.5 下载文件

OSS Go SDK提供了丰富的文件下载接口，用户可以通过以下方式从OSS中下载文件：

- 下载到流io.ReadCloser
- 下载到本地文件
- 分片下载

下载到流io.ReadCloser

说明：

下载的示例代码在sample/get_object.go。

下载文件到流

```
import (
    "fmt"
    "io/ioutil"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

body, err := bucket.GetObject("my-object")
if err != nil {
    // HandleError(err)
}
data, err := ioutil.ReadAll(body)
if err != nil {
    // HandleError(err)
}
body.Close()
fmt.Println("data:", string(data))
```

说明：

io.ReadCloser数据读取完毕后，需要调用Close关闭。

下载文件到缓存

```
import (
```

```
    "bytes"
    "io"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

body, err := bucket.GetObject("my-object")
if err != nil {
    // HandleError(err)
}
buf := new(bytes.Buffer)
io.Copy(buf, body)
body.Close()
```

下载文件到本地文件流

```
import (
        "io"
        "os"
        "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

body, err := bucket.GetObject("my-object")
if err != nil {
    // HandleError(err)
}
defer body.Close()

fd, err := os.OpenFile("LocalFile", os.O_WRONLY|os.O_CREATE, 0660)
if err != nil {
    // HandleError(err)
}
defer fd.Close()

io.Copy(fd, body)
```

下载到本地文件

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
```

```
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.GetObjectToFile("my-object", "LocalFile")
if err != nil {
    // HandleError(err)
}
```

分片下载

当下载大文件时，如果网络不稳定或者程序崩溃了，则整个下载就失败了。用户不得不重头再来，这样做不仅浪费资源，在网络不稳定的情况下，往往重试多次还是无法完成下载。 通过Bucket.DownloadFile接口来实现断点续传下载。它有以下参数：

- objectKey 要下载的Object名字
- filePath 下载到本地文件的路径
- partSize 下载分片大小，从1B到5GB，单位是Byte
- options 可选项，主要包括：
  - Routines 指定分片下载的并发数，默认是1，及不使用并发下载
  - Checkpoint 指定下载是否开启断点续传，及checkpoint文件的路径。默认断点续传功能关闭，checkpoint文件的路径可以指定为空，如果不指定则默认为与本地文件同目录下的*file.cpt*，其中*file*是本地文件的名字
  - 下载时限定条件，请参看指定限定条件下载

其实现的原理是将要下载的Object分成若干个分片分别下载，最后所有分片都下载成功后，完成整个文件的下载。在下载的过程中会记录当前下载的进度信息（记录在checkpoint文件中）和已下载的分片，如果下载过程中某一分片下载失败，再次下载时会从checkpoint文件中记录的点继续下载。这要求再次调用时要指定与上次 相同的checkpoint文件。下载完成后，checkpoint文件会被删除。

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
```

```
    }

    err = bucket.DownloadFile("my-object", "LocalFile", 100*1024, oss.Routines(3), oss.
Checkpoint(true, ""))
    if err != nil {
        // HandleError(err)
    }
```

说明：

- SDK会将下载的中间状态信息记录在cp文件中，所以要确保用户对cpt文件有写权限。
- cpt文件记录了下载的中间状态信息并自带了校验，用户不能去编辑它，如果cpt文件损坏则重新下载文件。
- 如果下载过程中待下载的Object发生了改变（ETag改变），或者part文件丢失或被修改，则重新下载文件。
- 指定断点续传checkpoint文件路径使用*oss.Checkpoint(true, "your-cp-file.cp")*。
- 使用*bucket.DownloadFile(objectKey, localFile, 100*1024)*，默认不使用分片并发下载、不启动断点续传。

指定限定条件下载

下载文件时，用户可以指定一个或多个限定条件，所有的限定条件都满足时下载，不满足时报错不下载文件。可以使用的限定条件如下：

参数	说明
IfModifiedSince	如果指定的时间早于实际修改时间，则正常传送。否则返回错误。
IfUnmodifiedSince	如果传入参数中的时间等于或者晚于文件实际修改时间，则正常传输文件；否则返回错误。
IfMatch	如果传入期望的ETag和object的 ETag匹配，则正常传输；否则返回错误。
IfNoneMatch	如果传入的ETag值和Object的ETag不匹配，则正常传输；否则返回错误。

```
    import "time"
    import "github.com/aliyun/aliyun-oss-go-sdk/oss"

    client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
    if err != nil {
        // HandleError(err)
    }

    bucket, err := client.Bucket("my-bucket")
```

```
if err != nil {
    // HandleError(err)
}

date := time.Date(2015, time.November, 10, 23, 0, 0, 0, time.UTC)

// 限定条件不满足，不下载文件
err = bucket.GetObjectToFile("my-object", "LocalFile", oss.IfModifiedSince(date))
if err == nil {
    // HandleError(err)
}

// 满足限定条件，下载文件
err = bucket.GetObjectToFile("my-object", "LocalFile", oss.IfUnmodifiedSince(date))
if err != nil {
    // HandleError(err)
}
```

说明：

- ETag的值可以通过Bucket.GetObjectDetailedMeta获取。
- Bucket.GetObject，Bucket.GetObjectToFile，Bucket.DownloadFile都支持限定条件。

文件压缩下载

文件可以压缩下载，目前支持GZIP压缩。Bucket.GetObject、Bucket.GetObjectToFile支持压缩功能。

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.GetObjectToFile("my-object.txt", "LocalFile.gzip", oss.AcceptEncoding("gzip"))
if err != nil {
    // HandleError(err)
}
```

1.2.10.6 管理文件

一个Bucket下可能有非常多的文件，SDK提供一系列的接口方便用户管理文件。

列出存储空间中的文件

通过Bucket.ListObjects来列出当前Bucket下的文件。主要的参数如下：

参数	说明
Delimiter	用于对Object名字进行分组的字符。所有名字包含指定的前缀且第一次出现delimiter字符之间的object作为一组元素CommonPrefixes
Prefix	限定返回的object key必须以prefix作为前缀。注意使用prefix查询时，返回的key中仍会包含prefix
MaxKeys	限定此次返回object的最大数，如果不设定，默认为100，max-keys取值不能大于1000
Marker	设定结果从marker之后按字母排序的第一个开始返回

说明：

ListObjects的示例代码在sample/list_objects.go。

使用默认参数获取存储空间的文件列表，默认返回100条Object

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

lsRes, err := bucket.ListObjects()
if err != nil {
    // HandleError(err)
}

fmt.Println("Objects:", lsRes.Objects)
```

指定最大返回数量，最多不能超过1000条

```
lsRes, err := bucket.ListObjects(oss.MaxKeys(200))
if err != nil {
    // HandleError(err)
}
fmt.Println("Objects:", lsRes.Objects)
```

返回指定前缀的Object，默认最多返回100条

```
lsRes, err := bucket.ListObjects(oss.Prefix("my-object-"))
if err != nil {
    // HandleError(err)
}
```

```
fmt.Println("Objects:", lsRes.Objects)
```

指定从某个Object(my-object-xx)后返回，默认最多100条

```
lsRes, err := bucket.ListObjects(oss.Marker("my-object-xx"))
if err != nil {
    // HandleError(err)
}
fmt.Println("Objects:", lsRes.Objects)
```

分页获取所有Object，每次返回200条

```
marker := oss.Marker("")
for {
    lsRes, err := bucket.ListObjects(oss.MaxKeys(200), marker)
    if err != nil {
        HandleError(err)
    }
    marker = oss.Marker(lsRes.NextMarker)

    fmt.Println("Objects:", lsRes.Objects)

    if !lsRes.IsTruncated {
        break
    }
}
```

分页所有获取从特定Object后的所有的Object，每次返回50条

```
marker = oss.Marker("my-object-xx")
for {
    lsRes, err := bucket.ListObjects(oss.MaxKeys(50), marker)
    if err != nil {
        // HandleError(err)
    }

    marker = oss.Marker(lsRes.NextMarker)

    fmt.Println("Objects:", lsRes.Objects)

    if !lsRes.IsTruncated {
        break
    }
}
```

分页所有获取指定前缀为的Object，每次返回80个。

```
prefix := oss.Prefix("my-object-")
marker := oss.Marker("")
for {
    lsRes, err := bucket.ListObjects(oss.MaxKeys(80), marker, prefix)
    if err != nil {
        // HandleError(err)
    }

    prefix = oss.Prefix(lsRes.Prefix)
    marker = oss.Marker(lsRes.NextMarker)
```

```
        fmt.Println("Objects:", lsRes.Objects)

        if !lsRes.IsTruncated {
            break
        }
    }
```

模拟目录结构

OSS是基于对象的存储服务，没有目录的概念。存储在一个Bucket中所有文件都 是通过文件的key唯一标识，并没有层级的结构。这种结构可以让OSS的存储非常高效，但是用户管理文件时希望能够像传统的文件系统一样把文件分门别类放到 不同的“目录”下面。通过OSS提供的“公共前缀”的功能，也可以很方便地模拟目录结构。

假设Bucket中已有如下文件：

```
foo/x
foo/y
foo/bar/a
foo/bar/b
foo/hello/C/1
foo/hello/C/2
```

通过ListObjects，列出指定目录下的文件和子目录：

```
    lsRes, err := bucket.ListObjects(oss.Prefix("foo/"), oss.Delimiter("/"))
    if err != nil {
        // HandleError(err)
    }
    fmt.Println("Objects:", lsRes.Objects,"SubDir:", lsRes.CommonPrefixes)
```

结果中lsRes.Objects为文件，包括foo/x、foo/y；lsRes.CommonPrefixes即子目录，包括foo/bar/、foo/hello/。

判断文件是否存在

通过Bucket.IsObjectExist来判断文件是否存在。

```
    import "github.com/aliyun/aliyun-oss-go-sdk/oss"

    client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
    if err != nil {
        // HandleError(err)
    }

    bucket, err := client.Bucket("my-bucket")
    if err != nil {
        // HandleError(err)
    }

    isExist, err := bucket.IsObjectExist("my-object")
    if err != nil {
        // HandleError(err)
```

```
}
```

删除单个文件

通过Bucket.DeleteObject来删除某个文件：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.DeleteObject("my-object")
if err != nil {
    // HandleError(err)
}
```

删除多个文件

通过Bucket.DeleteObjects来删除多个文件，用户可以通过**DeleteObjectsQuiet**参数来指定是否返回删除的结果。默认返回删除结果。

说明：

删除文件的示例代码在sample/delete_object.go。

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

// 默认返回删除成功的文件
delRes, err := bucket.DeleteObjects([]string{"my-object-1", "my-object-2"})
if err != nil {
    // HandleError(err)
}
fmt.Println("Deleted Objects:", delRes.DeletedObjects)

// 不返回删除的结果
_, err = bucket.DeleteObjects([]string{"my-object-3", "my-object-4"},
    oss.DeleteObjectsQuiet(true))
if err != nil {
    // HandleError(err)
```

```
}
```

说明：

- Bucket.DeleteObjects至少有一个ObjectKey，不能为空。
- Bucket.DeleteObjects使用的Go的xml包，该包实现了XML1.0标准，XML1.0不支持的特性请不要使用。

获取文件的元信息(Object Meta)

OSS上传/拷贝文件时，除了文件内容，还可以指定文件的一些属性信息，称为“元信息”。这些信息在上传时与文件一起存储，在下载时与文件一起返回。

在SDK中文件元信息用一个**Map**表示，其他key和value都是**string**类型，并 且都**只能是简单的ASCII可见字符**，**不能包含换行。** 所有元信息的总大小不能超过8KB。

说明：

- 因为文件元信息在上传/下载时是附在HTTP Headers中， HTTP协议规定不能包含复杂字符。
- 元数据的名称大小写不敏感，比较/读取时请忽略大小写。

使用Bucket.GetObjectDetailedMeta来获取Object的元信息。

说明：

元信息的示例代码在sample/object_meta.go。

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

props, err := bucket.GetObjectDetailedMeta("my-object")
if err != nil {
    // HandleError(err)
}
fmt.Println("Object Meta:", props)
```

说明：

Bucket.GetObjectMeta的结果中不包括Object的权限，获取Object权限通过Bucket.GetObjectACL。

修改文件元信息(Object Meta)

用户一次修改一条或多条元信息，可用元信息如下：

元信息	说明
CacheControl	指定新Object被下载时的网页的缓存行为。
ContentDisposition	指定新Object被下载时的名称。
ContentEncoding	指定新Object被下载时的内容编码格式。
Expires	指定新Object过期时间，建议使用GMT格式。
Meta	自定义参数，以**X-Oss-Meta-**为前缀的参数。

使用Bucket.SetObjectMeta来设置Object的元信息。

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

// 一次修改一条Meta
err = bucket.SetObjectMeta("my-object", oss.Meta("MyMeta", "MyMetaValue"))
if err != nil {
    // HandleError(err)
}

// 修改多条Meta
options := []oss.Option{
    oss.Meta("MyMeta", "MyMetaValue"),
    oss.Meta("MyObjectLocation", "HangZhou"),
}
err = bucket.SetObjectMeta("my-object", options...)
if err != nil {
    // HandleError(err)
}
```

拷贝文件

使用Bucket.CopyObject或Bucket.CopyObjectToBucket拷贝文件，前者是同一个Bucket内的文件拷贝，后者是Bucket之间的文件拷贝。

说明：

拷贝文件的示例代码在sample/copy_objects.go。

同一个Bucket内的文件拷贝

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

_, err = bucket.CopyObject("my-object", "descObjectKey")
if err != nil {
    // HandleError(err)
}
```

不同Bucket之间的文件拷贝

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

_, err = bucket.CopyObjectToBucket("my-object", "my-bucket-desc", "descObjectKey")
if err != nil {
    // HandleError(err)
}
```

拷贝时处理文件元信息

文件拷贝(CopyObject/CopyObjectToBucket)时对文件元信息的处理有两种选择，通过**MetadataDirective**参数指定：

- oss.MetaCopy 与源文件相同，即拷贝源文件的元信息
- oss.MetaReplace 使用新的元信息覆盖源文件的信息

默认值是oss.MetaCopy。

COPY时，MetadataDirective为MetaReplace时，用户可以指定新对象的如下的元信息：

元信息	说明
CacheControl	指定新Object被下载时的网页的缓存行为。
ContentDisposition	指定新Object被下载时的名称。
ContentEncoding	指定新Object被下载时的内容编码格式。
Expires	指定新Object过期时间（seconds）更详细描述请参照RFC2616。
ServerSideEncryption	指定OSS创建新Object时的服务器端加密编码算法。
ObjectACL	指定OSS创建新Object时的访问权限。
Meta	自定义参数，以**X-Oss-Meta-**为前缀的参数。

```
import (
    "time"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

expires := time.Date(2049, time.January, 10, 23, 0, 0, 0, time.UTC)
options := []oss.Option{
    oss.MetadataDirective(oss.MetaReplace),
    oss.Expires(expires),
    oss.ObjectACL(oss.ACLPublicRead),
    oss.Meta("MyMeta", "MyMetaValue")}

_, err = bucket.CopyObject("my-object", "descObjectKey", options...)
if err != nil {
    // HandleError(err)
}
```

限定拷贝条件

文件拷贝时可以设置限定条件，条件满足时拷贝，不满足时报错不拷贝。可以使用的限定条件如下：

参数	说明
CopySourceIfMatch	如果源Object的ETAG值和用户提供的ETAG相等，则执行拷贝操作；否则返回错误。

参数	说明
CopySourceIfNoneMatch	如果源Object的ETAG值和用户提供的ETAG不相等，则执行拷贝操作；否则返回错误。
CopySourceIfModifiedSince	如果传入参数中的时间等于或者晚于源文件实际修改时间，则正常拷贝；否则返回错误。
CopySourceIfUnmodifiedSince	如果源Object自从用户指定的时间以后被修改过，则执行拷贝操作；否则返回错误。

```
import (
    "fmt"
    "time"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

date := time.Date(2015, time.November, 10, 23, 0, 0, 0, time.UTC)
// 约束条件不满足，拷贝没有执行
_, err = bucket.CopyObject("my-object", "descObjectKey", oss.CopySourceIfModifiedSince(
date))
fmt.Println("CopyObjectError:", err)

// 约束条件满足，拷贝执行
_, err = bucket.CopyObject("my-object", "descObjectKey", oss.CopySourceIfUnmodifi
edSince(date))
if err != nil {
    // HandleError(err)
}
```

说明：

Bucket.CopyObject、Bucket.CopyObjectToBucket都支持拷贝时处理文件元信息、限定拷贝条件。

1.2.10.7 自定义域名绑定

OSS支持用户将自定义的域名绑定到OSS服务上，这样能够支持用户无缝地将存储迁移到OSS上。例如用户的域名是my-domain.com，之前用户的所有图片资源都是形如*ttp://img.my-domain.com/xx.jpg*的格式，用户将图片存储迁移到OSS之后，通过绑定自定义域名，仍可以使用原来的地址访问到图片：

- 开通OSS服务并创建Bucket
- 将img.my-domain.com与创建的Bucket绑定
- 将图片上传到OSS的这个Bucket中
- 修改域名的DNS配置，增加一个CNAME记录，将img.my-domain.com指向OSS服务 的endpoint（如my-bucket.oss-cn-hangzhou.aliyuncs.com）

这样就可以通过原地址*http://img.my-domain.com/x.jpg*访问到存储在OSS上的图片。

在使用SDK时，可以使用自定义域名作为endpoint，这时需要将**UseCname**参数设置为true，如下面的例子：

说明：

跨域资源共享的示例代码在sample/cname_sample.go。

```
import (
    "fmt"
    "io/ioutil"
    "strings"
    "github.com/aliyun/aliyun-oss-go-sdk/oss"
)

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret", oss.UseCname(true))
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

err = bucket.PutObject("my-object", strings.NewReader("MyObjectValue"))
if err != nil {
    // HandleError(err)
}

body, err := bucket.GetObject("my-object")
if err != nil {
    // HandleError(err)
}
data, err := ioutil.ReadAll(body)
if err != nil {
    // HandleError(err)
}
body.Close()
data = data  // 处理数据

lsRes, err := bucket.ListObjects()
if err != nil {
    // HandleError(err)
}
fmt.Println("Objects:", lsRes.Objects)
```

```
err = bucket.DeleteObject("my-object")
if err != nil {
    // HandleError(err)
}
```

说明：

使用Cname时，无法使用list_buckets接口。（因为自定义域名已经绑定到某个特定的Bucket。）

1.2.10.8 设置访问权限

OSS允许用户对Bucket和Object分别设置访问权限，方便用户控制自己的资源可以被如何访问。对于Bucket，有三种访问权限：

- public-read-write 允许匿名用户向该Bucket中创建/获取/删除Object
- public-read 允许匿名用户获取该Bucket中的Object
- private 不允许匿名访问，所有的访问都要经过签名

创建Bucket时，默认是private权限。之后用户可以通过Client.SetBucketACL来设置 Bucket的权限。上面三种权限分布对应Go SDK中的常量ACLPublicReadWrite、ACLPublicRead、ACLPrivate。

Bucket访问权限

说明：

Bucket访问权限设置的示例代码在sample/bucket_acl.go。

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

// 设置Bucket ACL
err = client.SetBucketACL("my-bucket", oss.ACLPublicRead)
if err != nil {
    // HandleError(err)
}

// 查看Bucket ACL
aclRes, err := client.GetBucketACL("my-bucket")
if err != nil {
    // HandleError(err)
}
```

```
fmt.Println("Bucket ACL:", aclRes.ACL)
```

Object访问权限

对于Object，有四种访问权限：

- default 继承所属的Bucket的访问权限，即与所属Bucket的权限值一样
- public-read-write 允许匿名用户读写该Object
- public-read 允许匿名用户读该Object
- private 不允许匿名访问，所有的访问都要经过签名

创建Object时，默认为default权限。之后用户可以通过Bucket.SetObjectACL来设置Object的权限。上面四种权限分布对应 Go SDK中的常量ACLDefault、ACLPublicReadWrite、ACLPublicRead、ACLPrivate。

说明：

Object访问权限设置的示例代码在sample/object_acl.go。

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

bucket, err := client.Bucket("my-bucket")
if err != nil {
    // HandleError(err)
}

// 设置Object的访问权限
err = bucket.SetObjectACL("my-object", oss.ACLPrivate)
if err != nil {
    // HandleError(err)
}

// 查看Object的访问权限
aclRes, err := bucket.GetObjectACL("my-object")
if err != nil {
    // HandleError(err)
}
fmt.Println("Object ACL:", aclRes.ACL)
```

说明：

- 如果设置了Object的权限（非default），则访问该Object时进行权限认证时会优先判断Object的权限，而Bucket的权限设置会被忽略。

- 允许匿名访问时（设置了public-read或者public-read-write权限），用户可以直接通过浏览器访问，例如：*http://bucket-name.oss-cn-hangzhou.aliyuncs.com/object.jpg*。

1.2.10.9 管理生命周期

OSS允许用户对Bucket设置生命周期规则，以自动淘汰过期掉的文件，节省存储空间。用户可以同时设置多条规则，一条规则包含：

- 规则ID，用于标识一条规则，不能重复
- 受影响的文件前缀，此规则只作用于符合前缀的文件
- 过期时间，有两种指定方式：
  1. 指定距文件最后修改时间N天过期
  2. 指定在具体的某一天过期，即在那天之后符合前缀的文件将会过期，**而不论文件的最后修改时间**。不推荐使用。
- 是否生效

说明：
管理生命周期的示例代码在sample/bucket_lifecycle.go。

设置生命周期规则

通过Client.SetBucketLifecycle来设置生命周期规则：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

// id:"rule1", enable:true, prefix:"foo/", expiry:Days 3
rule1 := oss.BuildLifecycleRuleByDays("rule1", "foo/", true, 3)
// id:"rule2", enable:false, prefix:"bar/", expiry:Date 2016/1/1
rule2 := oss.BuildLifecycleRuleByDate("rule2", "bar/", true, 2016, 1, 1)
rules := []oss.LifecycleRule{rule1, rule2}

err = client.SetBucketLifecycle("my-bucket", rules)
if err != nil {
    // HandleError(err)
}
```

查看生命周期规则

通过Client.GetBucketLifecycle来查看生命周期规则：

```
import "fmt"
```

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

lcRes, err := client.GetBucketLifecycle("my-bucket")
if err != nil {
    // HandleError(err)
}
fmt.Println("Lifecycle Rules:", lcRes.Rules)
```

清空生命周期规则

通过Client.DeleteBucketLifecycle设置一个空的Rule数组来清空生命周期规则：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.DeleteBucketLifecycle("my-bucket")
if err != nil {
    // HandleError(err)
}
```

1.2.10.10 设置访问日志

OSS允许用户对Bucket设置访问日志记录，设置之后对于Bucket的访问会被记录成日志，日志存储在OSS上由用户指定的Bucket中，文件的格式为：

```
<TargetPrefix><SourceBucket>-YYYY-mm-DD-HH-MM-SS-UniqueString
```

其中TargetPrefix由用户指定。日志规则由以下3项组成：

- enable，是否开启
- target_bucket，存放日志文件的Bucket
- target_prefix，指定最终被保存的访问日志文件前缀

说明：

Bucket访问权限设置的示例代码在sample/bucket_logging.go。

开启Bucket日志

通过Client.SetBucketLogging来开启日志功能：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"
```

```
client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

// target_bucket:"my-target-bucket", target_prefix:"my-object-", enable: true
err = client.SetBucketLogging("my-bucket", "my-target-bucket", "my-object-", true)
if err != nil {
    // HandleError(err)
}
```

查看Bucket日志设置

通过Client.GetBucketLogging来查看日志设置：

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

logRes, err := client.GetBucketLogging("my-bucket")
if err != nil {
    // HandleError(err)
}
fmt.Println("Target Bucket:", logRes.LoggingEnabled.TargetBucket,
            "Target Prefix:", logRes.LoggingEnabled.TargetPrefix)
```

关闭Bucket日志

通过Bucket.DeleteBucketLogging来关闭日志功能：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.DeleteBucketLogging("my-bucket")
if err != nil {
    // HandleError(err)
}
```

1.2.10.11 静态网站托管

在自定义域名绑定中提到，OSS 允许用户将自己的域名指向OSS服务的地址。这样用户访问他的网站的时候，实 际上是在访问OSS的Bucket。对于网站，需要指定首页(index)和出错页(error) 分别对应的Bucket中的文件名。

设置托管页面

通过Client.SetBucketWebsite来设置托管页面：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

// bucketName:"my-bucket", indexWebsite:"index.html", errorWebsite:"error.html"
err = client.SetBucketWebsite("my-bucket", "index.html", "error.html")
if err != nil {
    // HandleError(err)
}
```

查看托管页面

通过Client.GetBucketWebsite来查看托管页面：

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

wsRes, err := client.GetBucketWebsite("my-bucket")
if err != nil {
    // HandleError(err)
}

fmt.Println("indexWebsite:", wsRes.IndexDocument.Suffix,
            "errorWebsite:", wsRes.ErrorDocument.Key)
```

清除托管页面

通过Client.DeleteBucketWebsite来清除托管页面：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.DeleteBucketWebsite("my-bucket")
if err != nil {
    // HandleError(err)
```

```
}
```

1.2.10.12 设置防盗链

OSS是按使用收费的服务，为了防止用户在OSS上的数据被其他人盗链，OSS支持基于HTTP header中表头字段referer的防盗链方法。

说明：

设置防盗链的示例代码在sample/bucket_referer.go。

设置Referer白名单

通过Bucket.SetBucketReferer设置Referer白名单，该函数有三个参数：

- bucketName 存储空间名称。
- referers 访问白名单列表。一个bucket可以支持多个referer参数。eferer参数支持通配符 * 和 ?。
- allowEmptyReferer 指定是否允许referer字段为空的请求访问。默认配置为true。

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

referers := []string{"http://www.aliyun.com",
                     "http://www.???.aliyuncs.com",
                     "http://www.*.com"}
err = client.SetBucketReferer("my-bucket", referers, false)
if err != nil {
    // HandleError(err)
}
```

查看Referer白名单

通过Bucket.GetBucketReferer设置Referer白名单：

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

refRes, err := client.GetBucketReferer("my-bucket")
if err != nil {
    // HandleError(err)
}
fmt.Println("Referers:", refRes.RefererList,
```

```
            "AllowEmptyReferer:", refRes.AllowEmptyReferer)
```

清空Referer白名单

清空Referer白名单，即把白名单设置成空，allowEmptyReferer为true：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.SetBucketReferer("my-bucket", []string{}, true)
if err != nil {
    // HandleError(err)
}
```

1.2.10.13 设置跨域资源共享

跨域资源共享(CORS)允许web端的应用程序访问不属于本域的资源。OSS提供接口方便开发者控制跨域访问的权限。

OSS的跨域共享设置由一条或多条CORS规则组成，每条CORS规则包含以下设置：

- allowed_origins，允许的跨域请求的来源，如www.my-domain.com, *
- allowed_methods，允许的跨域请求的HTTP方法(PUT/POST/GET/DELETE/HEAD)
- allowed_headers，在OPTIONS预取指令中允许的header，如x-oss-test, *
- expose_headers，允许用户从应用程序中访问的响应头
- max_age_seconds, 浏览器对特定资源的预取（OPTIONS）请求返回结果的缓存时间

说明：

跨域资源共享的示例代码在sample/bucket_cors.go。

设置CORS规则

通过Client.SetBucketCORS设置CORS规则：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

rule1 := oss.CORSRule{
    AllowedOrigin: []string{"*"},
    AllowedMethod: []string{"PUT", "GET"},
    AllowedHeader: []string{},
    ExposeHeader:  []string{},
```

```
    MaxAgeSeconds: 200,
}

rule2 := oss.CORSRule{
    AllowedOrigin: []string{"http://www.a.com", "http://www.b.com"},
    AllowedMethod: []string{"POST"},
    AllowedHeader: []string{"Authorization"},
    ExposeHeader:  []string{"x-oss-test", "x-oss-test1"},
    MaxAgeSeconds: 100,
}

err = client.SetBucketCORS("my-bucket", []oss.CORSRule{rule1, rule2})
if err != nil {
    // HandleError(err)
}
```

查看CORS规则

通过Client.GetBucketCORS查看CORS规则：

```
import "fmt"
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

corsRes, err := client.GetBucketCORS("my-bucket")
if err != nil {
    // HandleError(err)
}
fmt.Println("Bucket CORS:", corsRes.CORSRules)
```

清空CORS规则

通过Client.DeleteBucketCORS清空CORS规则：

```
import "github.com/aliyun/aliyun-oss-go-sdk/oss"

client, err := oss.New("Endpoint", "AccessKeyId", "AccessKeySecret")
if err != nil {
    // HandleError(err)
}

err = client.DeleteBucketCORS("my-bucket")
if err != nil {
    // HandleError(err)
}
```

1.2.10.14 错误

Go中调用出错，会统一返回接口error，该接口定义如下：

```
type error interface {
    Error() string
```

```
}
```

其他的错误继承于该接口。比如HTTP错误请求返回的错误如下：

```
//net.Error
type Error interface {
    error
    Timeout() bool   // Is the error a timeout
    Temporary() bool // Is the error temporary
}
```

使用OSS Go SDK时如果请求出错，会有相应的error返回。HTTP请求、IO等错误会返回Go预定的错误。 OSS Server处理请求出错，返回如下的错误，该错误实现了error接口。

```
type ServiceError struct {
    Code       string    // OSS返回给用户的错误码
    Message    string    // OSS给出的详细错误信息
    RequestId  string    // 用于唯一标识该次请求的UUID
    HostId     string    // 用于标识访问的OSS集群
    StatusCode int       // HTTP状态码
}
```

如果OSS返回的HTTP状态码与预期不符返回如下错误，该错误也实现了error接口。

```
type UnexpectedStatusCodeError struct {
    allowed []int    // 预期OSS返回HTTP状态码
    got     int      // OSS实际返回HTTP状态码
}
```

OSS的错误码

OSS的错误码列表如下：

错误码	描述	HTTP状态码
AccessDenied	拒绝访问	403
BucketAlreadyExists	Bucket已经存在	409
BucketNotEmpty	Bucket不为空	409
EntityTooLarge	实体过大	400
EntityTooSmall	实体过小	400
FileGroupTooLarge	文件组过大	400
InvalidLinkName	Object Link与指向的Object同名	400
LinkPartNotExist	Object Link中指向的Object不存在	400

错误码	描述	HTTP状态码
ObjectLinkTooLarge	Object Link中Object个数过多	400
FieldItemTooLong	Post请求中表单域过大	400
FilePartInterity	文件Part已改变	400
FilePartNotExist	文件Part不存在	400
FilePartStale	文件Part过时	400
IncorrectNumberOfFilesInPOSTRequest	Post请求中文件个数非法	400
InvalidArgument	参数格式错误	400
InvalidAccessKeyId	AccessKeyId不存在	403
InvalidBucketName	无效的Bucket名字	400
InvalidDigest	无效的摘要	400
InvalidEncryptionAlgorithmError	指定的熵编码加密算法错误	400
InvalidObjectName	无效的Object名字	400
InvalidPart	无效的Part	400
InvalidPartOrder	无效的part顺序	400
InvalidPolicyDocument	无效的Policy文档	400
InvalidTargetBucketForLogging	Logging操作中有无效的目标bucket	400
InternalError	OSS内部发生错误	500
MalformedXML	XML格式非法	400
MalformedPOSTRequest	Post请求的body格式非法	400
MaxPOSTPreDataLengthExceededError	Post请求上传文件内容之外的body过大	400
MethodNotAllowed	不支持的方法	405
MissingArgument	缺少参数	411
MissingContentLength	缺少内容长度	411
NoSuchBucket	Bucket不存在	404
NoSuchKey	文件不存在	404

错误码	描述	HTTP状态码
NoSuchUpload	Multipart Upload ID不存在	404
NotImplemented	无法处理的方法	501
PreconditionFailed	预处理错误	412
RequestTimeTooSkewed	发起请求的时间和服务器时间超出15分钟	403
RequestTimeout	请求超时	400
RequestIsNotMultiPartContent	Post请求content-type非法	400
SignatureDoesNotMatch	签名错误	403
TooManyBuckets	用户的Bucket数目超过限制	400
InvalidEncryptionAlgorithmError	指定的熵编码加密算法错误	400

说明：

- 上表的错误码即OssServiceError.Code，HTTP状态码即OssServiceError.StatusCode。
- 如果试图以OSS不支持的操作来访问某个资源，返回405 Method Not Allowed错误。

1.2.11 Media-C-SDK

1.2.11.1 前言

简介

不少情况下，我们都需要将摄像头拍摄的视频快速存储到云端（OSS），但是我们也有一些因素要考虑：

- 设备上不能存储永久access key id和access key secret，因为可能泄露
- 设备上只允许上传或者下载，不允许删除、修改配置等管理权限
- 可以提供网页让用户去管理自己的视频
- 对设备的权限精准控制
- 对设备的权限存在有效期，不能让设备永久持有某种权限`
- 希望摄像机输出的音视频可以通过HLS协议直接被用户观看

针对以上考虑，我们推出了OSS MEDIA C SDK，构建于OSS C SDK版本之上，可以方便的解决上述问题，为音视频行业提供更完善易用的解决方案。

- 本文档主要介绍OSS MEDIA C SDK的安装和使用，适用于OSS MEDIA C SDK版本2.0.1。
- 并且假设您已经开通了阿里云OSS 服务，并创建了AccessKeyId 和AccessKeySecret。
- 如果您还没有开通或者还不了解阿里云OSS服务，请登录*OSS产品主页*了解。
- 如果还没有创建AccessKeyId和AccessKeySecret，请到阿里云Access Key管理创建Access Key。

SDK下载

Linux（2016-09-27）版本2.0.1

- SDK包： *aliyun-media-c-sdk-2.0.1.tar.gz*
- 源代码：*GitHub*

Windows

- 不支持

版本迭代详情参考这里

兼容性

对于1.x.x 系列SDK，由于OSS C SDK中list相关的接口发生变化后不兼容，其他接口兼容。

1.2.11.2 安装

版本依赖

Linux

OSS C SDK = 3.x.x

Windows

不支持

Linux环境安装

- 从这里下载*SDK*)包或者下载源代码，应该包括src，sample，test三个目录和CMakeList.txt文件。

安装到系统目录

- 如果OSS C SDK和其依赖都是安装在系统目录下(/usr/local/或/usr/)，且希望OSS MEDIA C SDK也安装到系统目录下，执行下列命令编译安装：

```
cmake .
make
```

```
make install
```

- 上面命令执行成功后，OSS MEDIA C SDK会自动安装到/usr/local/下面。

安装到自定义目录（依赖包安装到系统目录）

- 如果OSS C SDK和其依赖都是安装到系统目录下(/usr/local/或/usr/)，但希望OSS MEDIA C SDK安装到自定义目录，比如/home/user/aliyun/oss/install/，执行下列命令编译安装：

```
cmake . -DCMAKE_INSTALL_PREFIX=/home/user/aliyun/oss/install/usr/local/
make
make install
```

- 上面命令执行成功后，OSS MEDIA C SDK会自动安装到/home/user/aliyun/oss/install/usr/local/下面

安装到自定义目录（依赖包安装在自定义目录）

- 如果OSS C SDK或某些依赖包安装到了自定义目录，此时编译OSS MEDIA C SDK时默认是找不到这些包的头文件和库文件，所以需要在执行cmake时指定路径，比如已经将OSS C SDK安装到了/home/user/aliyun/oss/install/目录，则执行下列命令编译安装：

```
cmake . -DCMAKE_INSTALL_PREFIX=/home/user/aliyun/oss/install/usr/local/ -
DOSS_C_SDK_INCLUDE_DIR=/home/user/aliyun/oss/install/usr/local/include/ -
DOSS_C_SDK_LIBRARY=/home/user/aliyun/oss/install/usr/local/lib/liboss_c_sdk.so
make
make install
```

- 上面命令执行成功后，OSS MEDIA C SDK会自动安装到/home/user/aliyun/oss/install/usr/local/下面
- 其他依赖包相关参数名称：APR_UTIL_LIBRARY，APR_LIBRARY，CURL_LIBRARY，APR_INCLUDEDIRS，APU_INCLUDEDIRS，OSS_C_SDK_INCLUDE_DIR，CURL_INCLUDEDIRS等

仅编译安装客户端SDK

- 默认是同时编译安装客户端和服务端的sdk的，如果仅需要编译安装客户端的SDK，则执行下列命令编译安装：

```
cmake . -DONLY_BUILD_CLIENT=ON
make
make install
```

- 如果仅需要编译安装服务端，将ONLY_BUILD_CLIENT修改为ONLY_BUILD_SERVER即可。
- 只有同时编译客户端和服务端时才会编译测试用例

其他编译安装方式和问题

- 编译模式：目前支持四种，分别是Debug，Release，MinSizeRef，RelWithDebInfo，指定使用某种编译类型，使用参数-DCMAKE_BUILD_TYPE，比如使用Debug模式编译，则在cmake是增加参数-DCMAKE_BUILD_TYPE=Debug：cmake . -DCMAKE_BUILD_TYPE=Debug，默认是Release模式。
  - Debug：没有做任何代码优化，支持gdb，一般用来调试程序
  - Release：使用了更高级别的优化，一般适用于生产环境
  - MinSizeRef：生成最小大小的库文件，一般用于嵌入式环境
  - RelWithDebInfo：使用了更高级的优化，但附带了调试信息，一般也用于生产环境
- 执行cmake时出现”Targets may link only to libraries. CMake is dropping the item”的warnning，原因是指定的library路径不对，library路径应该指定到*.so，比如/path/to/xxx.so。
- 如果需要使用OSS C SDK的静态库，则在执行cmake时指定-DOSS_C_SDK_LIBRARY=/home/user/aliyun/oss/install/usr/local/lib/liboss_c_sdk_static.a即可。其他库类似。
- 执行cmake时出现”CMake Error: The following variables are used in this project, but they are set to NOTFOUND.”，原因是相应的库无法从默认路径中找到，需要用户指定，参考安装到自定义目录。

1.2.11.3 初始化

确定Endpoint

Endpoint是阿里云OSS服务在各个区域的域名地址，目前支持两种形式。

Endpoint类型	解释
OSS区域地址	使用OSS Bucket所在区域地址
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS区域地址

使用OSS Bucket所在区域地址，Endpoint查询可以有下面两种方式：

- 您可以登录阿里云OSS控制台，进入Bucket概览页，Bucket域名的后缀部分：如bucket-1.oss-cn-hangzhou.aliyuncs.com的oss-cn-hangzhou.aliyuncs.com部分为该Bucket的外网Endpoint。

配置密钥

要接入阿里云OSS，您需要拥有一个有效的 Access Key(包括AccessKeyId和AccessKeySecret)用来进行签名认证。可以通过如下步骤获得：

- 注册阿里云账号
- 申请*AccessKey*

使用RAM和STS服务

如果您的产品需要从设备端（比如手机、平板电脑、摄像机等）上传、下载文件，设备端应该只获取到临时授权，而非永久授权。这时候，需要用到RAM和STS服务。

1. 开通RAM服务。
2. 创建一个角色，可以给这个角色授予AliyunOSSFullAccess和AliyunSTSAssumeRoleAccess等权限。
3. 创建成功后，在角色详情里面有一个Arn，类似于acs:ram:xxxx:role/yyyy，这个就是role_arn，在后续获取临时token时需要使用。

1.2.11.4 客户端

OSS MEDIA C SDK分为客户端，服务端和HLS三部分，下面主要介绍客户端的相关操作，其他的操作请访问后面章节。

接口

客户端相关的操作接口都位于oss_media_file.h中，目前提供的接口有：

- oss_media_file_open
- oss_media_file_stat
- oss_media_file_tell
- oss_media_file_seek
- oss_media_file_read
- oss_media_file_write
- oss_media_file_close

下面会详细介绍各个接口的功能和注意事项。

基础结构体介绍

```
/**
 * OSS MEDIA FILE的元数据，包括文件长度，位置和类型
 */
typedef struct {
    int64_t length;
    int64_t pos;
    char    *type;
} oss_media_file_stat_t;
/**
```

```
 * OSS MEDIA FILE的属性信息
 */
typedef struct oss_media_file_s {
    void   *ipc;
    char   *endpoint;
    int8_t is_cname;
    char   *bucket_name;
    char   *object_key;
    char   *access_key_id;
    char   *access_key_secret;
    char   *token;
    char   *mode;
    oss_media_file_stat_t _stat;
    time_t expiration;
    auth_fn_t auth_func;
} oss_media_file_t;
```

说明：

- type，文件类型，Normal或者Appendable
- ipc，用于设置设备唯一标识，根据用户的需要可以在auth func中使用，如果用不到可以忽略。
- endpoint，比如oss-cn-hangzhou.aliyuncs.com。
- is_cname，是否使用了CNAME
- bucket_name，OSS上存储空间的名称
- object_key，OSS上文件的名称
- access_key_id，阿里云提供的访问控制的access key id，这里有两种使用方式，第一种，使用主账号或者子账号的永久access key id，此时后面的token需要设置为NULL，第二种使用通过get_token获取到的临时access key id。
- access_key_secret，阿里云提供的访问控制的access key secret，这里也有两种使用方式，第一种，使用主账号或者子账号的永久access key secret，此时后面的token需要设置为NULL，第二种使用通过get_token获取到的临时access key secret。access_key_id和access_key_secret必须同时使用同种方式。
- token，如果是使用主账号或者子账号的永久access key id和access key secret，这里需要设置为NULL。如果是终端设备上传下载资源，此时需要应用服务器给终端设备提供临时的访问权限，可以通过服务端的get_token获取到临时的access_key_id，access_key_secret和token。如果用户设置了token不为NULL，则会认为是使用了临时access key id，临时access key secrety和临时token，所以，如果不想使用临时token，请将其设置为NULL。
- expiration，授权失效的时间，首次授权后，后续超过失效时间后才会再次授权
- auth，授权函数，用户需要实现一个函数，在这个函数内部为host，bucket，token等赋值
- mode，读写模式

初始化

```
/**
 * @brief 初始化oss media
 * @note  在程序开始的时候应该首先调用此接口，初始化OSS MEDIA C SDK
 * @return:
 *     返回0时表示成功
 *     否则，表示出现了错误，可能导致失败的原因包括：内存不足，apr、curl版本太低等
 */
int oss_media_init(aos_log_level_e log_level);
```

说明：

示例代码参考：*GitHub*

销毁

```
/**
 * @brief 销毁oss meida
 * @note  在程序结束的时候应该最后调用此接口，销毁OSS MEDIA C SDK
 */
void oss_media_destroy();
```

说明：

示例代码参考：*GitHub*

打开文件

```
/**
 * @brief 打开一个oss文件
 * @param[in] bucket_name oss上存储文件的存储空间名称
 * @param[in] object_key oss上的文件名称
 * @param[in] mode:
 *     'r': 读模式
 *     'w': 覆盖写模式
 *     'a': 追加写模式
 *     notes: 不允许组合使用
 * @param[in] auth_func   授权函数，设置access_key_id/access_key_secret等
 * @return:
 *     返回非NULL时成功，否则失败
 */
oss_media_file_t* oss_media_file_open(char *bucket_name,
                                      char *object_key,
                                      char *mode,
                                      auth_fn_t auth_func);
```

说明：

- mode，支持只读、覆盖写、追加写三种模式，**不支持组合模式**。读模式，打开文件后，一次读取部分文件内容，完整内容一般需要多次读取；覆盖写，打开文件后，一次写入完整的文件内

容，**可以多次写入**，**但是最后一次写回覆盖的前面写入的内容**；追加写，打开文件后，可以多次追加写文件。

- 示例代码参考：*GitHub*

关闭文件

```
/**
 * @brief 关闭oss文件
 */
void oss_media_file_close(oss_media_file_t *file);
```

说明：

示例代码参考：*GitHub*

写文件

```
/**
 * @brief 写文件到oss上
 * @return:
 *     成功时返回写入的数据大小，返回-1表示写失败
 */
int64_t oss_media_file_write(oss_media_file_t *file, const void *buf, int64_t nbyte);
```

示例程序：

```
/* 授权函数 */
static void auth_func(oss_media_file_t *file) {
    file->endpoint = "your endpoint";
    file->is_cname = 0;
    file->access_key_id = "阿里云提供的access key id或者临时access key id";
    file->access_key_secret = "阿里云提供的access key secret或者临时access key secret";
    file->token = "通过get_token接口获取到得临时token";
    /* 本次授权的有效时间 */
    file->expiration = time(NULL) + 300;
}
static void write_file() {
    oss_clean(g_filename);
    int64_t write_size;
    oss_media_file_t *file;
    char *bucket_name = "<your bucket name>";
    char *key = "<your object key>";
    char *content = "aliyun oss media c sdk";
    /* 打开一个文件 */
    file = oss_media_file_open(bucket_name, key, "w", auth_func);
    if (!file) {
        printf("open media file failed\n");
        return;
    }
    /* 写文件 */
    write_size = oss_media_file_write(file, content, strlen(content));
    if (-1 != write_size) {
         printf("write %" PRId64 " bytes succeeded\n", write_size);
    } else {
```

```
        oss_media_file_close(file);
        printf("write failed\n");
        return;
    }
    /* 关闭文件，释放资源 */
    oss_media_file_close(file);
}
```

说明：

- 示例中，打开文件的模式为覆盖写("w")，如果多次写入，后一次写回覆盖前一次写；如果需要追加写，打开文件的模式请指定为追加写("a")。
- 示例代码参考：*GitHub*

读文件

```
/**
 * @brief 读取固定数目nbyte的数据
 * @note  buf的大小应该大于等于nbyte + 1
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：不是以读模式打开的文件，无法连接OSS，无权限读OSS等
 */
int64_t oss_media_file_read(oss_media_file_t *file, void *buf, int64_t nbyte);
```

示例程序：

```
    int ntotal, nread, nbuf = 16;
    char buf[nbuf];
    char *content;
    char *bucket_name = "<your bucket name>";
    char *key = "<your object key>";
    oss_media_file_t *file;
    /* 打开文件 */
    file = oss_media_file_open(bucket_name, key, "r", auth_func);
    if (!file) {
        printf("open media file failed\n");
        return;
    }
    /* 读文件 */
    content = malloc(stat.length + 1);
    ntotal = 0;
    while ((nread = oss_media_file_read(file, buf, nbuf)) > 0) {
        memcpy(content + ntotal, buf, nread);
        ntotal += nread;
    }
    content[ntotal] = '\0';
    /* 关闭文件 */
    oss_media_file_close(file);
    free(content);
    printf("oss media c sdk read object succeeded\n");
```

```
}
```

说明：

示例代码参考：*GitHub*

管理文件

OSS MEDIA FILE也支持tell，seek和stat操作。

```
/**
 * @brief 获取当前OSS MEDIA FILE的位置
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：不是以读模式打开的文件，无法连接OSS，无权限读OSS等
 */
int64_t oss_media_file_tell(oss_media_file_t *file);
/**
 * @brief 设置OSS MEDIA FILE的指针到指定位置
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：不是以读模式打开的文件，无法连接OSS，无权限读OSS等
 */
int64_t oss_media_file_seek(oss_media_file_t *file, int64_t offset);
/**
 * @brief 获取OSS MEDIA FILE的元数据
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限读OSS等
 */
int oss_media_file_stat(oss_media_file_t *file, oss_media_file_stat_t *stat);
```

示例程序：

```
static void seek_tell_stat_file() {
    int ntotal, nread, nbuf = 16;
    char buf[nbuf];
    char *bucket_name = "<your bucket name>";
    char *key = "<your object key>";
    oss_media_file_t *file;
    /* 打开文件 */
    file = oss_media_file_open(bucket_name, key, "r", auth_func);
    if (!file) {
        printf("open media file failed\n");
        return;
    }
    /* 获取文件meta信息 */
    oss_media_file_stat_t stat;
    if (0 != oss_media_file_stat(file, &stat)) {
        oss_media_file_close(file);
        oss_media_file_free(file);
        printf("stat media file[%s] failed\n", file->object_key);
        return;
    }
```

```
    printf("file [name=%s, length=%ld, type=%s]",
           file->object_key, stat.length, stat.type);
    /* tell */
    printf("file [position=%" PRId64 "]", oss_media_file_tell(file));
    /* seek */
    oss_media_file_seek(file, stat.length / 2);
    /* 关闭文件 */
    oss_media_file_close(file);
    printf("oss media c sdk seek object succeeded\n");
}
```

说明：

示例代码参考：*GitHub*

1.2.11.5 服务端

OSS MEDIA C SDK分为客户端，服务端和HLS三部分，下面主要介绍服务端的相关操作，其他的操作请访问后面章节。

接口

服务端相关的操作接口都位于oss_media.h中，目前提供的接口有：

- oss_media_create_bucket
- oss_media_delete_bucket
- oss_media_create_bucket_lifecycle
- oss_media_get_bucket_lifecycle
- oss_media_delete_bucket_lifecycle
- oss_media_delete_file
- oss_media_list_files
- oss_media_get_token
- oss_media_get_token_from_policy

下面会详细介绍各个接口的功能和注意事项。

基础结构体介绍

```
typedef struct oss_media_config_s {
    char *endpoint;
    int is_cname;
    char *access_key_id;
    char *access_key_secret;
    char *role_arn;
} oss_media_config_t;

typedef struct oss_media_files_s {
    char *path;
```

```
    char *marker;
    int  max_size;

    char *next_marker;
    int  size;
    char **file_names;
} oss_media_files_t;
```

说明：

- endpoint，比如oss-cn-hangzhou.aliyuncs.com。
- is_cname，是否使用了CNAME
- access_key_id，阿里云提供的访问控制的access key id
- access_key_secret，阿里云提供的访问控制的access key secret
- role_arn，阿里云访问控制中创建的角色的Arn，具体位置在：访问控制RAM控制台 -> 角色管理 -> 点击相应角色名称 -> 基本信息 -> Arn，格式类似于acs:ram::xxxxxx:role/yyyy。如果还没有角色，需要创建一个新的角色，并赋予AliyunOSSFullAccess和AliyunSTSAssumeRoleAccess等权限。
- marker，设定结果从marker之后按字母排序的第一个开始返回。
- max_size，设定返回的最大数，取值不能大于1000。
- next_marker，下次执行时开始的位置

初始化

```
/**
 *  @brief  初始化oss media
 *  @note   在程序开始的时候应该首先调用此接口，初始化OSS MEDIA C SDK
 *  @return:
 *      返回0时表示成功
 *      否则，表示出现了错误，可能导致失败的原因包括：内存不足，apr、curl版本太低等
 */
int oss_media_init();
```

说明：

示例代码参考：*GitHub*

销毁

```
/**
 *  @brief  销毁oss meida
 *  @note   在程序结束的时候应该最后调用此接口，销毁OSS MEDIA C SDK
 */
```

```
void oss_media_destroy();
```

说明：

示例代码参考：*GitHub*

创建存储空间

```
/**
 * @brief 创建新的存储空间
 * @paramp[in] oss_media_acl_t
 *    OSS_ACL_PRIVATE        私有读写
 *    OSS_ACL_PUBLIC_READ     公共读，私有写
 *    OSS_ACL_PUBLIC_READ_WRITE 公共读写
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限等
 */
int oss_media_create_bucket(oss_media_config_t *config, const char *bucket_name,
oss_media_acl_t acl);
```

示例程序：

```
static void init_media_config(oss_media_config_t *config) {
    config->endpoint = "your endpoint";
    config->access_key_id = "阿里云提供的access key id";
    config->access_key_secret = "阿里云提供的access key secret";
    config->role_arn = "阿里云访问控制RAM提供的role arn";
    config->is_cname = 0;
}

void create_bucket() {
    int ret;
    char *bucket_name;
    oss_media_config_t config;

    /* 初始化变量 */
    bucket_name = "<your bucket name>";
    init_media_config(&config);

    /* 创建存储空间 */
    ret = oss_media_create_bucket(&config, bucket_name, OSS_ACL_PRIVATE);

    if (0 == ret) {
        printf("create bucket[%s] succeeded.\n", bucket_name);
    } else {
        printf("create bucket[%s] failed.\n", bucket_name);
    }
}
```

说明：

示例代码参考：*GitHub*

删除存储空间

```
/*
 * @brief 删除存储空间
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限等
 */
int oss_media_delete_bucket(oss_media_config_t *config, const char *bucket_name);
```

示例程序：

```
void delete_bucket() {
    int ret;
    char *bucket_name;
    oss_media_config_t config;

    /* 初始化变量 */
    bucket_name = "<your bucket name>";
    init_media_config(&config);

    /* 删除存储空间 */
    ret = oss_media_delete_bucket(&config, bucket_name);

    if (0 == ret) {
        printf("delete bucket[%s] succeeded.\n", bucket_name);
    } else {
        printf("delete bucket[%s] failed.\n", bucket_name);
    }
}
```

说明：

示例代码参考：*GitHub*

创建存储生命周期规则

```
/**
 * @brief 创建存储空间生命周期规则
 * @note  这些规则可以控制文件的自动删除时间
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限等
 */
int oss_media_create_bucket_lifecycle(oss_media_config_t *config, const char *bucket_name,
oss_media_lifecycle_rules_t *rules);
```

示例程序：

```
void create_bucket_lifecycle() {
    int ret;
    char *bucket_name;
    oss_media_lifecycle_rules_t *rules;
    oss_media_config_t config;

    /* 初始化变量 */
```

```
    bucket_name = "<your bucket name>";
    init_media_config(&config);

    /* 创建生命周期规则 */
    rules = oss_media_create_lifecycle_rules(2);
    oss_media_lifecycle_rule_t rule1;
    rule1.name = "example-1";
    rule1.path = "/example/1";
    rule1.status = "Enabled";
    rule1.days = 1;
    oss_media_lifecycle_rule_t rule2;
    rule2.name = "example-2";
    rule2.path = "/example/2";
    rule2.status = "Disabled";
    rule2.days = 2;

    rules->rules[0] = &rule1;
    rules->rules[1] = &rule2;

    /* 设置存储空间的生命周期规则 */
    ret = oss_media_create_bucket_lifecycle(&config,
            bucket_name, rules);

    if (0 == ret) {
        printf("create bucket[%s] lifecycle succeeded.\n", bucket_name);
    } else {
        printf("create bucket[%s] lifecycle failed.\n", bucket_name);
    }

    /* 释放资源 */
    oss_media_free_lifecycle_rules(rules);
}
```

说明：

示例代码参考：*GitHub*

获取存储空间的生命周期规则

```
/**
 *  @brief  获取存储空间的生命周期规则
 *  @return:
 *      返回0时表示成功
 *      否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS等
 */
int oss_media_get_bucket_lifecycle(oss_media_config_t *config, const char *bucket_name,
oss_media_lifecycle_rules_t *rules);
```

示例程序：

```
void get_bucket_lifecycle() {
    int ret, i;
    char *bucket_name;
    oss_media_lifecycle_rules_t *rules;
    oss_media_config_t config;

    /* 初始化变量 */
    bucket_name = "<your bucket name>";
```

```
    init_media_config(&config);

    /* 获取生命周期规则 */
    rules = oss_media_create_lifecycle_rules(0);
    ret = oss_media_get_bucket_lifecycle(&config, bucket_name, rules);

    if (0 == ret) {
        printf("get bucket[%s] lifecycle succeeded.\n", bucket_name);
    } else {
        printf("get bucket[%s] lifecycle failed.\n", bucket_name);
    }

    for (i = 0; i < rules->size; i++) {
        printf(">>>> rule: [name:%s, path:%s, status=%s, days=%d]\n",
               rules->rules[i]->name, rules->rules[i]->path,
               rules->rules[i]->status, rules->rules[i]->days);
    }

    /* 释放资源 */
    oss_media_free_lifecycle_rules(rules);
}
```

说明：

示例代码参考：*GitHub*

删除存储空间的生命周期规则

```
/**
 *  @brief  删除存储空间的生命周期规则
 *  @return:
 *      返回0时表示成功
 *      否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限等
 */
int oss_media_delete_bucket_lifecycle(oss_media_config_t *config, const char *bucket_name);
```

示例程序：

```
void delete_bucket_lifecycle()
{
    int ret;
    char *bucket_name;
    oss_media_config_t config;

    /* 初始化变量 */
    bucket_name = "<your bucket name>";
    init_media_config(&config);

    /* 删除生命周期规则 */
    ret = oss_media_delete_bucket_lifecycle(&config, bucket_name);

    if (0 == ret) {
        printf("delete bucket[%s] lifecycle succeeded.\n", bucket_name);
    } else {
        printf("delete bucket[%s] lifecycle failed.\n", bucket_name);
    }
```

```
}
```

说明：

示例代码参考：*GitHub*

删除文件

```
/**
 * @brief 删除存储空间中特定的文件
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限等
 */
int oss_media_delete_file(oss_media_config_t *config, const char *bucket_name, const char *
key);
```

示例程序：

```
void delete_file() {
    int ret;
    oss_media_config_t config;
    char *file;
    char *bucket_name;

    /* 初始化变量 */
    file = "oss_media_file";
    bucket_name = "<your bucket name>";
    init_media_config(&config);

    /* 删除文件 */
    ret = oss_media_delete_file(&config, bucket_name, file);

    if (0 == ret) {
        printf("delete file[%s] succeeded.\n", file);
    } else {
        printf("delete file[%s] lifecycle failed.\n", file);
    }
}
```

说明：

示例代码参考：*GitHub*

列出文件

```
/**
 * @brief 列出特定存储空间内的文件
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接OSS，无权限等
 */
```

```
int oss_media_list_files(oss_media_config_t *config, const char *bucket_name, oss_media_
files_t *files);
```

示例程序：

```
void list_files() {
    int ret, i;
    char *bucket_name;
    oss_media_files_t *files;
    oss_media_config_t config;

    /* 初始化变量 */
    bucket_name = "<your bucket name>>";
    init_media_config(&config);

    files = oss_media_create_files();
    files->max_size = 50;

    /* 列举文件 */
    ret = oss_media_list_files(&config, bucket_name, files);

    if (0 == ret) {
        printf("list files succeeded.\n");
    } else {
        printf("list files lifecycle failed.\n");
    }

    for (i = 0; i < files->size; i++) {
        printf(">>>>file name: %s\n", files->file_names[i]);
    }

    /* 释放资源 */
    oss_media_free_files(files);
}
```

说明：

示例代码参考：*GitHub*

获取临时token

```
/*
 *  @brief  获取临时token
 *  @param[in]:
     mode:
 *      'r': 读模式
 *      'w': 覆盖写模式
 *      'a': 追加写模式
 *    expiration: 临时token的有效时间，最小15分钟，最长1小时，单位秒
 *  @return:
 *      返回0时表示成功
 *      否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接阿里云STS服务，没
有开通RAM、STS，没有创建Role，参数不合法等
 */
int oss_media_get_token(oss_media_config_t *config,
                        const char *bucket_name,
                        const char *path,
                        const char *mode,
```

```
                    int64_t expiration,
                    oss_media_token_t *token);
```

示例程序：

```
void get_token() {
    int ret;
    char *bucket_name;
    oss_media_token_t token;
    oss_media_config_t config;

    /* 初始化变量 */
    bucket_name = "<your bucket name>";
    init_media_config(&config);

    /* 获取临时token */
    ret = oss_media_get_token(&config, bucket_name, "/*", "rwa",
                              3600, &token);

    if (0 == ret) {
        printf("get token succeeded, access_key_id=%s, access_key_secret=%s, token=%s\n",
               token.tmpAccessKeyId, token.tmpAccessKeySecret, token.securityToken);
    } else {
        printf("get token failed.\n");
    }
}
```

说明：

示例代码参考：*GitHub*

通过自定义policy获取token

```
/**
 * @brief 通过指定自定义的policy获取token
 * @param[in]:
 *     expiration: 临时token的有效时间，最小15分钟，最长1小时，单位秒
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误，可能导致失败的原因包括：无法连接阿里云STS服务，没有开通RAM、STS，没有创建Role，参数不合法等
 */
int oss_media_get_token_from_policy(oss_media_config_t *config,
                                    const char *policy,
                                    int64_t expiration,
                                    oss_media_token_t *token);
```

示例程序：

```
void get_token_from_policy() {
    int ret;
    oss_media_token_t token;
    char *policy;
    oss_media_config_t config;

    /* 初始化变量 */
    init_media_config(&config);
```

```
    /* 设置自定义policy */
    policy = "{\"Version\":\"1\",\"Statement\":[{\"Effect\":\"Allow\", "
            "\"Action\":\"*\", \"Resource\":\"*\"}]}";

    /* 获取token */
    ret = oss_media_get_token_from_policy(&config, policy, 3600, &token);

    if (0 == ret) {
        printf("get token succeeded, access_key_id=%s, access_key_secret=%s, token=%s\n",
                token.tmpAccessKeyId, token.tmpAccessKeySecret, token.securityToken);
    } else {
        printf("get token failed.\n");
    }
}
```

说明：

示例代码参考：*GitHub*

1.2.11.6 HLS基础接口

OSS MEDIA C SDK 客户端部分支持将接收到的H.264、AAC格式封装为TS、M3U8格式，然后写到OSS上，用户通过对应的m3u8地址就可以欣赏视频音频了。

接口

HLS相关基础接口都位于oss_media_hls.h中，目前提供的接口有：

- oss_media_hls_open
- oss_media_hls_write_frame
- oss_media_hls_begin_m3u8
- oss_media_hls_write_m3u8
- oss_media_hls_end_m3u8
- oss_media_hls_flush
- oss_media_hls_close

下面详细介绍各个接口的功能和注意事项。

基础结构体介绍

```
/**
 *  OSS MEDIA HLS FRAME的元数据
 */
typedef struct oss_media_hls_frame_s {
    stream_type_t stream_type;
    frame_type_t frame_type;
    uint64_t pts;
    uint64_t dts;
    uint32_t continuity_counter;
```

```
    uint8_t  key:1;
    uint8_t *pos;
    uint8_t *end;
} oss_media_hls_frame_t;

/**
 *  OSS MEDIA HLS的描述信息
 */
typedef struct oss_media_hls_options_s {
    uint16_t video_pid;
    uint16_t audio_pid;
    uint32_t hls_delay_ms;
    uint8_t encrypt:1;
    char    key[OSS_MEDIA_HLS_ENCRYPT_KEY_SIZE];
    file_handler_fn_t handler_func;
    uint16_t pat_interval_frame_count;
} oss_media_hls_options_t;

/**
 *  OSS MEDIA HLS FILE的描述信息
 */
typedef struct oss_media_hls_file_s {
    oss_media_file_t *file;
    oss_media_hls_buf_t *buffer;
    oss_media_hls_options_t options;
    int64_t frame_count;
} oss_media_hls_file_t;
```

说明：

- stream_type，流类型，目前支持st_h264和st_aac两种
- frame_type，帧类型，目前支持ft_non_idr，ft_idr，ft_sei，ft_sps，ft_pps，ft_aud等
- pts，显示时间戳
- dts，解码时间戳
- continuity_counter，递增计数器，从0-15，起始值不一定取0，但必须是连续的
- key，是否是关键帧
- pos，当前帧数据的起始位置(含)
- end，当前帧数据的结束位置(不含)
- video_pid，视频的pid
- audio_pid，音频的pid
- hls_delay_ms，显示延迟毫秒数
- encrypt，是否使用AES-128加密，目前暂不支持
- key，使用加密时的秘钥，目前暂不支持
- handler_func，文件操作回调函数
- pat_interval_frame_count，隔多少帧插入一个pat，mpt表

打开HLS文件

```
/**
 * @brief 打开一个OSS HLS文件
 * @param[in] bucket_name oss上存储文件的存储空间名称
 * @param[in] object_key oss上的文件名称
 * @param[in] auth_func 授权函数，设置access_key_id/access_key_secret等
 * @return:
 *     返回非NULL时成功，否则失败
 */
oss_media_hls_file_t* oss_media_hls_open(char *bucket_name, char *object_key, auth_fn_t
auth_func);
```

说明：

示例代码参考：*GitHub*

关闭HLS文件

```
/**
 * @brief 关闭OSS HLS文件
 */
int oss_media_hls_close(oss_media_hls_file_t *file);
```

说明：

示例代码参考：*GitHub*

写HLS文件

```
/**
 * @brief 写H.264或者AAC的一帧数据到oss上
 * @param[in] frame        h.264或者aac格式的一帧数据
 * @param[out] file        hls file
 * @return:
 *     返回0时表示成功
 *     否则，表示出现了错误
 */
int oss_media_hls_write_frame(oss_media_hls_frame_t *frame, oss_media_hls_file_t *file);
```

示例程序：

```
static void write_frame(oss_media_hls_file_t *file) {
    oss_media_hls_frame_t frame;
    FILE    *file_h264;
    uint8_t *buf_h264;
    int     len_h264, i;
    int     cur_pos = -1;
    int     last_pos = -1;
    int     video_frame_rate = 30;
    int     max_size = 10 * 1024 * 1024;
    char    *h264_file_name = "/path/to/example.h264";

    /* 读取H.264文件 */
```

```
    buf_h264 = calloc(max_size, 1);
    file_h264 = fopen(h264_file_name, "r");
    len_h264 = fread(buf_h264, 1, max_size, file_h264);

    /* 初始化frame结构体 */
    frame.stream_type = st_h264;
    frame.pts = 0;
    frame.continuity_counter = 1;
    frame.key = 1;

    /* 遍历H.264的数据，抽取出每帧数据，然后写入oss */
    for (i = 0; i < len_h264; i++) {
        /*  判断当前位置是否下一帧数据的开头，也就是当前帧的结尾 */
        if ((buf_h264[i] & 0x0F)==0x00 && buf_h264[i+1]==0x00
            && buf_h264[i+2]==0x00 && buf_h264[i+3]==0x01)
        {
            cur_pos = i;
        }

    /* 如果获取到完整的一帧数据，就调用接口转为HLS格式后写入OSS */
        if (last_pos != -1 && cur_pos > last_pos) {
            frame.pts += 90000 / video_frame_rate;
            frame.dts = frame.pts;

            frame.pos = buf_h264 + last_pos;
            frame.end = buf_h264 + cur_pos;

            oss_media_hls_write_frame(&frame, file);
        }

        last_pos = cur_pos;
    }

    /* 关闭文件，释放资源 */
    fclose(file_h264);
    free(buf_h264);
}
```

说明：

- 示例代码参考：*GitHub*
- 如果H.264的数据中缺少Access Unit Delimiter NALs（00 00 00 01 09 xx），需要添加这个NAL，否则无法在ipad，iphone，safari上播放
- H.264的帧是通过0xX0，0x00，0x00，0x01分隔的；AAC的帧是通过0xFF，0x0X分隔的
- 当前帧为关键帧时，frame.key需要设置为1

写M3U8文件

```
/**
 *  @brief  写M3U8文件的头部数据
 *  @param[in]  max_duration  TS文件最长持续时间
 *  @param[in]  sequence      TS文件起始编号
 *  @param[out] file          m3u8 file
 *  @return:
```

```
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误
 */
void oss_media_hls_begin_m3u8(int32_t max_duration,
                              int32_t sequence,
                              oss_media_hls_file_t *file);

/**
 * @brief 写M3U8文件数据
 * @param[in] size          m3u8 item个数
 * @param[in] m3u8          m3u8 item的详细数据
 * @param[out] file         m3u8 file
 * @return:
 *     返回0时表示成功
 *     否则，返回-1时表示出现了错误
 */
int oss_media_hls_write_m3u8(int size,
                             oss_media_hls_m3u8_info_t m3u8[],
                             oss_media_hls_file_t *file);

/**
 * @brief 写M3U8文件的结束符等数据
 * @param[out] file         m3u8 file
 */
void oss_media_hls_end_m3u8(oss_media_hls_file_t *file);
```

示例程序：

```
static void write_m3u8() {
    char *bucket_name;
    char *key;
    oss_media_hls_file_t *file;

    bucket_name = "<your bucket name>";
    key = "<your m3u8 file name>";

    /* 打开一个HLS文件用来写M3U8格式的数据，文件名必须以.m3u8结尾 */
    file = oss_media_hls_open(bucket_name, key, auth_func);
    if (file == NULL) {
        printf("open m3u8 file[%s] failed.", key);
        return;
    }

    /* 构造3个ts格式文件的信息 */
    oss_media_hls_m3u8_info_t m3u8[3];
    m3u8[0].duration = 9;
    memcpy(m3u8[0].url, "video-0.ts", strlen("video-0.ts"));
    m3u8[1].duration = 10;
    memcpy(m3u8[1].url, "video-1.ts", strlen("video-1.ts"));

    /* 写入M3U8文件
    oss_media_hls_begin_m3u8(10, 0, file);
    oss_media_hls_write_m3u8(2, m3u8, file);
    oss_media_hls_end_m3u8(file);

    /* 关闭HLS文件 */
    oss_media_hls_close(file);

    printf("write m3u8 to oss file succeeded\n");
```

```
}
```

说明：

- 目前使用的M3U8版本是3
- 如果是录播，需要在结束的时候调用oss_media_hls_end_m3u8(file)接口写入结束符，否则可能无法播放；如果是直播，则不能调用此接口
- 示例代码参考：*GitHub*
- 可以通过示例程序观看效果
- Windows平台可以通过*VLC播放器*观看，iPhone，iPad，Mac等可以直接使用Safari观看

1.2.11.7 HLS封装接口

OSS MEDIA C SDK 客户端部分支持将接收到的H.264、AAC封装为TS、M3U8格式后写入OSS，除了基础接口外，还提供封装好的录播、直播接口。

接口

HLS相关封装接口都位于oss_media_hls_stream.h中，目前提供的接口有：

- oss_media_hls_stream_open
- oss_media_hls_stream_write
- oss_media_hls_stream_close

下面详细介绍各个接口的功能和注意事项。

基础结构体介绍

```
/**
 *  OSS MEDIA HLS STREAM OPTIONS的描述信息
 */
typedef struct oss_media_hls_stream_options_s {
    int8_t is_live;
    char *bucket_name;
    char *ts_name_prefix;
    char *m3u8_name;
    int32_t video_frame_rate;
    int32_t audio_sample_rate;
    int32_t hls_time;
    int32_t hls_list_size;
} oss_media_hls_stream_options_t;
/**
 *  OSS MEDIA HLS STREAM的描述信息
 */
typedef struct oss_media_hls_stream_s {
    const oss_media_hls_stream_options_t *options;
    oss_media_hls_file_t *ts_file;
    oss_media_hls_file_t *m3u8_file;
    oss_media_hls_frame_t *video_frame;
```

```
    oss_media_hls_frame_t *audio_frame;
    oss_media_hls_m3u8_info_t *m3u8_infos;
    int32_t ts_file_index;
    int64_t current_file_begin_pts;
    int32_t has_aud;
    aos_pool_t *pool;
} oss_media_hls_stream_t;
```

说明：

- is_live，是否是直播模式，直播模式的时候M3U8文件里面只最新的几个ts文件信息
- bucket_name，存储HLS文件的存储空间名称
- ts_name_prefix，TS文件名称的前缀
- m3u8_name，M3U8文件名称
- video_frame_rate，视频数据的帧率
- audio_sample_rate，音频数据的采样率
- hls_time，每个ts文件最大持续时间
- hls_list_size，直播模式时在M3U8文件中最多保留的ts文件个数

打开HLS stream文件

```
/**
 * @brief 打开一个oss hls文件
 * @param[in] auth_func 授权函数，设置access_key_id/access_key_secret等
 * @param[in] options 配置信息
 * @return:
 *     返回非NULL时成功，否则失败
 */
oss_media_hls_stream_t* oss_media_hls_stream_open(auth_fn_t auth_func,
                    const oss_media_hls_stream_options_t *options);
```

说明：

示例代码参考：*GitHub*

关闭HLS stream文件

```
/**
 * @brief 关闭HLS stream文件
 */
int oss_media_hls_stream_close(oss_media_hls_stream_t *stream);
```

说明：

示例代码参考：*GitHub*

写HLS stream文件

```
/**
 *  @brief  写入视频和音频数据
 *  @param[in]  video_buf 视频数据
 *  @param[in]  video_len 视频数据的长度，可以为0
 *  @param[in]  audio_buf 音频数据
 *  @param[in]  audio_len 音频数据的长度，可以为0
 *  @param[in]  stream    HLS stream
 *  @return:
 *      返回0时表示成功
 *      否则，表示出现了错误
 */
int oss_media_hls_stream_write(uint8_t *video_buf,
                               uint64_t video_len,
                               uint8_t *audio_buf,
                               uint64_t audio_len,
                               oss_media_hls_stream_t *stream);
```

示例程序：

```
static void write_video_audio_vod() {
    int ret;
    int max_size = 10 * 1024 * 1024;
    FILE *h264_file, *aac_file;
    uint8_t *h264_buf, *aac_buf;
    int h264_len, aac_len;
    oss_media_hls_stream_options_t options;
    oss_media_hls_stream_t *stream;
    /* 设置HLS stream的参数值 */
    options.is_live = 0;
    options.bucket_name = "<your buckete name>";
    options.ts_name_prefix = "vod/video_audio/test";
    options.m3u8_name = "vod/video_audio/vod.m3u8";
    options.video_frame_rate = 30;
    options.audio_sample_rate = 24000;
    options.hls_time = 5;
    /* 打开HLS stream */
    stream = oss_media_hls_stream_open(auth_func, &options);
    if (stream == NULL) {
        printf("open hls stream failed.\n");
        return;
    }
    /* 创建两个buffer用来存储音频和视频数据 */
    h264_buf = malloc(max_size);
    aac_buf = malloc(max_size);
    /* 读取一段视频数据和音频数据，然后调用接口写入OSS */
    {
        h264_file = fopen("/path/to/video/1.h264", "r");
        h264_len = fread(h264_buf, 1, max_size, h264_file);
        fclose(h264_file);
        aac_file = fopen("/path/to/audio/1.aac", "r");
        aac_len = fread(aac_buf, 1, max_size, aac_file);
        fclose(aac_file);
        ret = oss_media_hls_stream_write(h264_buf, h264_len,
                aac_buf, aac_len, stream);
        if (ret != 0) {
            printf("write vod stream failed.\n");
            return;
```

```
        }
    }
    /* 再读取一段视频数据和音频数据，然后调用接口写入OSS */
    {
        h264_file = fopen("/path/to/video/2.h264", "r");
        h264_len = fread(h264_buf, 1, max_size, h264_file);
        fclose(h264_file);
        aac_file = fopen("/path/to/audio/1.aac", "r");
        aac_len = fread(aac_buf, 1, max_size, aac_file);
        fclose(aac_file);
        ret = oss_media_hls_stream_write(h264_buf, h264_len,
                aac_buf, aac_len, stream);
        if (ret != 0) {
            printf("write vod stream failed.\n");
            return;
        }
    }
    /* 写完数据后，关闭HLS stream */
    ret = oss_media_hls_stream_close(stream);
    if (ret != 0) {
        printf("close vod stream failed.\n");
        return;
    }
    /* 释放资源 */
    free(h264_buf);
    free(aac_buf);
    printf("convert H.264 and aac to HLS vod succeeded\n");
}
```

说明：

- 目前的录播、直播接口都支持只有视频，只有音频，同时有音视频等。
- 示例代码参考：*GitHub*
- 目前的录播、直播接口比较初级，用户如果有高级需求，可以模拟这两个接口，使用基础接口自助实现高级定制功能。
- 可以通过示例程序观看效果

1.2.11.8 使用场景

在上两节分别介绍了客户端和服务端的相关操作，接下来我们介绍如何将客户端和服务端连接起来使用，如果您还没有阅读前两节，强烈建议先阅读前两节，然后再阅读本节。

视频监控

在前言里面，介绍了可以方便的用于网络摄像头等设备，这里，会详细介绍一下如何使用。

角色包括三个，网络摄像机，应用服务器，阿里云对象存储服务（OSS），网络摄像机内部使用OSS MEDIA C SDK的client部分，应用服务器内部使用OSS C MEDIA SDK的server部分，他们的流程如下：

说明：

- 当网络摄像机拍摄了一段视频，需要上传到OSS。
- 首先，网络摄像机向应用服务器发送网络请求：要求获取一个上传视频到OSS的授权。
- 应用服务器收到请求后，通过检查觉得可以给网络摄像机上传的权限，就通过OSS MEDIA C SDK的get_token接口，向阿里云请求一个在特定有效期有效的，只有上传权限的token。
- 阿里云接收到应用服务器的获取token请求后，通过检查用户的配置，如果允许，就颁发一个临时token（包括临时access key id，临时access key secret和临时sts token）：只有上传OSS的权限，且在特定时间内有效，然后发送给应用服务器。
- 应用服务器收到临时token后，转发给刚才要token的网络摄像机。
- 网络摄像机获取到token后，就可以通过OSS MEDIA C SDK client部分的oss_media_write接口上传视频文件到OSS了。

- 还可以在应用服务器上使用C SDK的server部分或者JAVA, C#，Go，Python，Php，Ruby等SDK实现一个HTTP服务，这样其他人就可以在网页上查看，管理各个视频文件。

示例代码

下面是一个简单模拟客户端和服务端操作的示例程序：

```
char* global_temp_access_key_id = NULL;
char* global_temp_access_key_secret = NULL;
char* global_temp_token = NULL;

/* 授权函数 */
static void auth_func(oss_media_file_t *file) {
    file->endpoint = "your endpoint";
    file->is_cname = 0;
    file->access_key_id = global_temp_access_key_id;
    file->access_key_secret = global_temp_access_key_secret;
    file->token = global_temp_token;

    /* 本次授权的有效时间  */
    file->expiration = time(NULL) + 300;
}

/* 模拟服务端发送token给客户端 */
static void send_token_to_client(oss_media_token_t token) {
    global_temp_access_key_id = token.tmpAccessKeyId;
    global_temp_access_key_secret = token.tmpAccessKeySecret;
    global_temp_token = token.securityToken;
}

void get_and_use_token() {
    oss_media_token_t token;

    /* 服务端逻辑：从阿里云获取到临时token后发送给客户端 */
    {
        int ret;
        char *policy = NULL;
        oss_media_config_t config;

        policy = "{\n"
                "\"Statement\": [\n"
                "{"
                "\"Action\": \"oss:*\",\n"
                "\"Effect\": \"Allow\",\n"
                "\"Resource\": \"*\"\n"
                "}\n"
                "],\n"
                "\"Version\": \"1\"\n"
                "}\n";

        init_media_config(&config);

    /* 从阿里云请求一个临时授权token */
        ret = oss_media_get_token_from_policy(&config, policy,
                17 * 60, &token);

    if (ret != 0) {
       printf ("Get token failed.");
       return;
```

```
    }

    /* 模拟将临时token发送给客户端 */
        send_token_to_client(token);
    }

    /* 客户端逻辑：从服务端获取到临时token后，使用临时token操作文件 */
    {
        int ret;
        int64_t write_size = 0;
        oss_media_file_t *file = NULL;
        char *content = NULL;
    char *bucket_name;
    char *object_key;
        oss_media_file_stat_t stat;

        content = "hello oss media file\n";
    bucket_name = "<your bucket name>";
    object_key = "key";

        /* 打开文件 */
        file = oss_media_file_open(bucket_name, object_key, "w", auth_func);
        if (file != NULL) {
        printf ("open file failed.");
        return;
    }

        /* 写文件 */
        write_size = oss_media_file_write(file, content, strlen(content));
        if (write_size != strlen(content)) {
        printf ("write file failed.");
        return;
    }

        /* 关闭文件释放资源 */
        oss_media_file_close(file);
    }
}
```

说明：

- policy可以从阿里云访问控制RAM中选择**角色管理**，然后点击某个角色，在基本信息下面的方框中获取。
- 如果不需要精准控制权限，可以使用更简单的oss_media_get_token接口，其中path参数可以是**/***，mode参数可以是 **rwa**。

1.2.11.9 常见问题

OSS MEDIA C SDK和OSS C SDK是啥关系？

OSS MEDIA C SDK依赖于OSS C SDK，OSS C MEDIA SDK中的上传，下载等功能是通过调用OSS C SDK的接口实现的。

OSS MEDIA C SDK是否支持windows？

目前还不支持。

是否支持追加写文件？

支持，调用oss_media_file_open时使用 **a** 模式，然后可以通过多次调用oss_media_file_write接口实现追加写。

什么是role arn？如何获取role arn？

role arn表示的是需要扮演角色的id，由阿里云访问控制RAM提供。可以前往访问控制RAM，选择**角色管理**，单击已经创建的角色名称，**基本信息** > **Arn**，值类似于：acs:ram::xxxx:role/yyyyy。如果还没有已创建的角色，需要在角色管理页面创建一个新的用户角色，并赋予AliyunSTSAssumeRoleAccess和其他相应角色。

如何运行sample？

修改sample/config.c文件，增加自己的access key id，access key secret，bucket等值，然后编译后，在bin目录下就会出现sample的可执行文件。

报错：error:a timeout was reached

检查一下host的值，是否是类似于oss-cn-hangzhou.aliyuncs.com的值。这个是C SDK的一个已知问题，会在后期版本修复。

运行sample时报错：error:Couldn't resolve host name 和[code=-990, message=HttploError]

修改sample/config.c文件，配置您自己的参数值，然后重新编译即可。测试也一样。

客户端和服务端的access key id，access key secret，token配置有啥不同和注意点？

- 服务端只需要配置access key id和access key secret，这两个值有两种来源：第一个是主账号的AccessKeys，第二个是主账号生成的子账号的AccessKeys。
- 客户端有两种配置方式，第一种是和服务端一样，只配置主账号或者子账号的access key id，access key secret，第二种是配置access key id，access key secret和token三个值，但这三个值都是服务端通过oss_media_get_token或者oss_media_get_token_from_policy获取到的临时AccessKey和token，有时间期限，超过有效期后，就不能再次使用。

执行sample获取token的时候出现以下错误：http_code=500, error_code=GetSTSTokenError, error_message=Internal Error

- 原因是安装的libcurl不支持HTTPS协议，导致无法访问sts服务。具体过程是机器上没有安装openssl-devel等ssl的开发包，在编译libcurl的时候找不到ssl，libcurl就自动禁止了HTTPS协议，导致编译出来的libcurl库不支持HTTPS，最终访问STS失败。
- 解决办法是先安装openssl-devel等ssl开发包，然后重新安装libcurl。在安装libcurl时，当执行完./configure后，检查最后一行的Protocols里包含了HTTPS，如果包含了，就说明正确了。

2 MaxCompute

2.1 SDK介绍

在本文档中，我们仅会对较为常用的MaxCompute核心接口做简短介绍，更多详细信息请参阅获取到的SDK中的SDK Java Doc。

包名	描述
odps-sdk-core	MaxCompute基础功能。 封装了基础的MaxCompute概念及其编程接口，包括odps、project、table等，tunnel相关的功能也在此包。
odps-sdk-core-internal	MaxCompute扩展功能。 封装了一些不常用的MaxCompute概念和操作，例如Event、XFlow等。
odps-sdk-commons	MaxCompute基础设施。 包含TableSchema、Column、Record、OdpsType等基础设施和一些util的封装。
odps-sdk-udf	MaxCompute UDF编程接口。
odps-sdk-mapred	MaxCompute MapReduce作业编程接口。

2.2 AliyunAccount

阿里云认证账号。输入参数为accessId及accessKey，是阿里云用户的身份标识和认证密钥。此类用来初始化MaxCompute。

2.3 Odps

MaxCompute SDK的入口，用户通过此类来获取项目空间下的所有对象集合，包括：Projects、Tables、Resources、Functions、Instances。

用户可以通过传入AliyunAccount实例来构造MaxCompute对象。程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
odps.setDefaultProject("my_project");
for (Table t : odps.tables()) {
....
}
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.4 Projects

MaxCompute DPS中所有项目空间的集合。

集合中的元素为Project 。程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps  odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
Project p = odps.projects().get("my_exists");
p.reload();
Map<String, String> properties = prj.getProperties();
...
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.5 Project

对项目空间信息的描述，可以通过Projects获取相应的项目空间。

2.6 SQLTask

用于运行、处理SQL任务的接口。可以通过run接口直接运行SQL。run接口返回Instance 实例，通过Instance获取SQL的运行状态及运行结果。

程序示例如下，仅供参考：

```
Account account = new AliyunAccount("my_access_id", "my_access_key"); Odps odps = new
Odps(account);
String  odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
Instance instance = SQLTask.run(odps, "my_project", "select ...");
String id = instance.getId();
instance.waitforsuccess();
Set<String> taskNames = instance.getTaskNames();
for (String name
taskNames) {
TaskSummary summary = instance.getTaskSummary(name);
String s = summary.getSummaryText();
}
Map<String, String> results = instance.getTaskResults();
Map<String, TaskStatus> taskStatus = instance.getTaskStatus();
for (Entry<String, TaskStatus> status : taskStatus.entrySet()) {
String result = results.get(status.getKey());
}
```

说明：

如果用户想创建表，需要通过SQLTask接口，而不是Table 接口。用户需要将创建表（CREATE TABLE）的语句传入SQLTask。MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.7 Instances

MaxCompute中所有实例（Instance）的集合。

集合中的元素为Instance 。程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
odps.setDefaultProject("my_project");
for (Instance i : odps.instances ()) {
....
}
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.8 Instance

对实例信息的描述，可以通过Instances获取相应的实例。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps (account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
Instance ins = odps.instances().get("instance id");
Date startTime = instance.getStartTime();
Date endTime = instance.getEndTime();
...
Status instanceStatus = instance.getStatus();
String instanceStatusStr = null;
if (instanceStatus == Status.TERMINATED) {
instanceStatusStr = TaskStatus.Status.SUCCESS.toString();
Map<String, TaskStatus> taskStatus = instance.getTaskStatus();
for (Entry<String, TaskStatus> status : taskStatus.entrySet()) {
if (status.getValue().getStatus() != TaskStatus.Status.SUCCESS) {
instanceStatusStr = status.getValue().getStatus().toString();
break;
}
}
} else {
instanceStatusStr = instanceStatus.toString();
}
...
TaskSummary summary = instance.getTaskSummary("instance name");
```

```
String s = summary.getSummaryText();
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.9 Tables

MaxCompute中所有表的集合。集合中的元素为Table。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
odps.setDefaultProject("my_project");
for (Table t : odps.tables()) {
....
}
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.10 Table

对表信息的描述，可以通过Tables获取相应的表。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
Table t = odps.tables().get("table name");
t.reload();
Partition part = t.getPartition(new PartitionSpec(tableSpec[1]));
part.reload();
...
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.11 Resources

MaxCompute中所有资源的集合。集合中的元素为Resource。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
odps.setDefaultProject("my_project");
for (Resource r : odps.resources()) {
....
}
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.12 Resource

对资源信息的描述，可以通过Resources获取相应的资源。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
Resource r = odps.resources().get("resource name");
r.reload();
if (r.getType() == Resource.Type.TABLE) { TableResource tr = new TableResource(r);
String tableSource = tr.getSourceTable().getProject() + "." + tr.getSourceTable().getName();
if (tr.getSourceTablePartition() != null) {
tableSource += " partition(" + tr.getSourceTablePartition().toString() + ")";
}
....
}
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

一个创建文件资源的示例：

```
String projectName = "my_porject";
String source = "my_local_file.txt";
File file = new File(source);
InputStream is = new FileInputStream(file);
FileResource resource = new FileResource();
String name = file.getName();
resource.setName(name);
```

```
odps.resources().create(projectName, resource, is);
```

一个创建表资源的示例：

```
TableResource resource = new TableResource(tableName, tablePrj, partitionSpec);
//resource.setName(INVALID_USER_TABLE);
resource.setName("table_resource_name");
odps.resources().update(projectName, resource);
```

2.13 Functions

MaxCompute中所有函数的集合。集合中的元素为Function。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
odps.setDefaultProject("my_project");
for (Function f : odps.functions()) {
....
}
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

2.14 Function

对函数信息的描述，可以通过Functions获取相应的函数。

程序示例如下：

```
Account account = new AliyunAccount("my_access_id", "my_access_key");
Odps odps = new Odps(account);
String odpsUrl = "<your odps endpoint>";
odps.setEndpoint(odpsUrl);
Function f = odps.functions().get("function name");
List<Resource> resources = f.getResources();
```

说明：

MaxCompute提供了两个服务地址供用户选择。详细说明请参考《*Tunnel SDK*简介》。

一个创建函数的示例：

```
String resources = "xxx:xxx";
String classType = "com.aliyun.odps.mapred.open.example.WordCount";
ArrayList<String> resourceList = new ArrayList<String>();
for (String r : resources.split(":")) {
resourceList.add(r);
}
```

```
Function func = new Function();
func.setName(name);
func.setClassType(classType);
func.setResources(resourceList);
odps.functions().create(projectName, func);
```

2.15 Spark Shell

Spark向集群提交任务时用到的常用接口之一。

执行如下命令，启动接口应用。

```
$cd $SPARK_HOME
-- 进入spark目录。
$bin/spark-shell --master yarn
-- 选择运行模式并启动应用。
```

简单应用示例如下：

```
sc.parallelize(0 to 100, 2).collect
sql("show tables").show
sql("select * from spark_user_data").show(200,100)
```

2.16 Spark R

Spark向集群提交任务时用到的常用接口之一。

执行如下命令，启动接口应用。

```
$mkdir -p /home/admin/R && unzip ./R/R.zip -d /home/admin/R/
-- 创建一个目录R并解压该目录下的R.zip。
$export PATH=/home/admin/R/bin/:$PATH
-- 设置环境变量。
$bin/sparkR --master yarn --archives ./R/R.zip
-- 选择运行模式并启动应用。
```

简单应用示例如下：

```
df <- as.DataFrame(faithful)
df
head(select(df, df$eruptions))
head(select(df, "eruptions"))
head(filter(df, df$waiting < 50))
results <- sql("FROM spark_user_data SELECT *")
head(results)
```

2.17 Spark SQL

Spark向集群提交任务时用到的常用接口之一。

执行如下命令，启动接口应用。

```
$cd $SPARK_HOME
```

```
-- 进入spark目录。
$bin/spark-sql --master yarn
-- 选择运行模式并启动应用。
```

简单应用示例如下：

```
show tables;
select * from spark_user_data limit 3;
quit;
```

2.18 Spark JDBC

Spark向集群提交任务时用到的常用接口之一。

执行如下命令，启动接口应用。

```
$sbin/stop-thriftserver.sh
-- 停止线程。
$sbin/start-thriftserver.sh
-- 重启线程。
$bin/beeline
-- 启动应用。
```

简单应用示例如下：

```
!connect jdbc:hive2://localhost:10000/odps_smoke_test
show tables;
select * from mr_input limit 3;
!quit
```

2.19 更多接口信息

在上述文档中，仅对较为常用的MaxCompute核心接口做了简短介绍，用户如需获取更加完整的接口信息，请参见如下的MaxCompute接口介绍链接。

*MaxCompute*完整接口信息。

3 DataWorks

3.1 前言

关于本文档

本文档全面系统地给出了阿里云DataWorks中各相关对象的API定义，通过完整的参数说明和使用示例，帮助开发人员了解阿里云大数据开发套件的基本模型的使用方法。

阅读对象

本文档可作为开发人员在对阿里云DataWorks的模型和架构有基本了解的情况下，进行二次开发和数据收集的参考。

文档约定

本文档遵循如下约定：

- **排版约定**；

  下表主要描述了本手册中常用的排版约定：

字体格式或标志	释义
粗体	所有标题和功能点均使用**加粗**字体表示。
【表管理】	表示大数据集成服务平台中的具体功能模块。
create table	表示代码示例，如create table s_member_cart。
斜体	命令行参数（代码示例必须由实际值进行替代的部分）采用斜体表示。
【注意】	表示需要读者注意的事项。
【提示】	配置、操作或使用此平台的技巧。
注意事项内容	表示需要读者注意的具体事项说明。

- **符号约定。**

  下表主要描述了本手册中常用的符号约定：

符号	描述	范例	释义
>	表示在平台中的菜单选项	文件>新建>项目	从**文件**菜单中选择**新建**，然后从**新建**子菜单中选择**项目**。

->	表示某流程的先后顺序	申请->审批->赋权	此申请流程需先进行**申请**，再进行**审批**，最后**赋权**。

3.2 基本术语

项目空间（Project）

项目空间是阿里云大数据集成服务平台最基本的组织对象，是用户管理表（Table）、资源（Resource）、自定义函数（UDF）、节点（Node）、权限等的基本单元。

3.3 产品简介

DataWorks是一种简单高效，图形化页面化构建和实现算法逻辑，实施数据处理的调试和定期运维的集成环境。

3.4 API概览

调度系统API

接口名称	接口功能
新建Flow API	在调度系统中创建Flow
更新Flow API	更新调度系统中的Flow信息
.....	

3.5 调用方式

3.5.1 请求结构

3.5.1.1 新建工作流任务

调用方式

```
POST http://baseapi.example.com/v2.0/flow/project/{projectIdentifier}
```

参数

```
{
    FlowCreateDto flowCreateDto;
}
```

FlowUpdateDto属性

```
 {
```

```
String flowName;
String flowDisplayName;
String owner;
String runType; //运行类型，取值: "0", "1", "2"
String scheduleExp; // 调度的cron表达式
Date startEffectDate; // 有效起始时间
Date endEffectDate; // 有效终止时间
String paraValue; // 运行参数，可被每个节点继承
String description; // 描述信息
Boolean isAuto;    //是否自动调度
NodeCreateDto rootNode;  //Flow的根节点
List<NodeCreateDto> includeNodes;//Flow包含的节点
List<String>  sameCycleParents;//依赖的正常周期Flow,格式:"不同 flow 名字用逗号分隔，若为不同project 下的 flow，格式为 ProjectName.FlowName"
String diffCycleDependentType;//跨周期依赖的依赖类型，0 - 不跨周期，1 - 用户自定义跨周期依赖， 2 - 跨一层子节点依赖， 3 - 自依赖
 List<String>  diffCycleParents;//跨周期依赖的Flow，格式:"不同 flow 名字用逗号分隔，若为不同 project 下的 flow，格式为 ProjectName.FlowName"
}
```

NodeCreateDto属性

```
 {
String nodeName; // 节点名称
String nodeDisplayName; // 节点展示名称
Integer type; // 执行代码类型
String runType;//运行类型
Integer dependentType; // 依赖类型，0 - 不跨周期，1 - 用户自定义跨周期依赖， 2 - 跨一层子节点依赖， 3 - 自依赖
String nodeCode;//节点对应的代码
String description; // 描述信息
String paraValue; // 参数信息
Date startEffectDate; // 允许调度的起始日期
Date endEffectDate; // 允许调度的终止日期
String owner; // 负责人账号
String source; // 来源
String connection; // 连接串
String scheduleExp; // 调度的cron表达式
List<String> parents;//上层节点
List<String> customDependentParents;//自定义依赖有父节点
}
```

返回结果

```
 {
String requestId;//请求的id
String returnCode;//0表示调用成功
String returnMessage;//返回执行的详细信息
Boolean returnValue; //执行成功返回true
```

```
}
```

3.5.1.2 更新工作流任务

调用方式

```
PUT  http://baseapi.example.com/v2.0/flow/project/{projectIdentifier}
```

参数

```
{
FlowUpdateDto flowUpdateDto;
}
```

FlowUpdateDto属性

```
{
String flowName;
String flowDisplayName;
String owner;
String runType; //运行类型，取值: "0", "1", "2"
String scheduleExp; // 调度的cron表达式
Date startEffectDate; // 有效起始时间
Date endEffectDate; // 有效终止时间
String paraValue; // 运行参数，可被每个节点继承
String description; // 描述信息
Boolean isAuto;    //是否自动调度
List<String>  sameCycleParents;//依赖的正常周期Flow,格式:"不同 flow 名字用逗号分隔，若为不同project 下的 flow，格式为 ProjectName.FlowName"
String diffCycleDependentType;//跨周期依赖的依赖类型，0 - 不跨周期，1 - 用户自定义跨周期依赖， 2 - 跨一层子节点依赖， 3 - 自依赖
List<String>  diffCycleParents;//跨周期依赖的Flow,格式:"不同 flow 名字用逗号分隔，若为不同project 下的 flow，格式为 ProjectName.FlowName"
}
```

返回结果

```
{
String requestId;//请求的id
String returnCode;//0表示调用成功
String returnMessage;//返回执行的详细信息
Boolean returnValue; //执行成功返回true
```

```
}
```

3.5.1.3 向上查询父节点

调用方式

```
GET  http://baseapi.example.com/v2.0/node/project/{projectIdentifier}/flow/{flowName}/node/{
nodeName}/parents
```

参数

```
{
String projectName; //url中的项目名
String flowName; //url中的flowName
String nodeName; //url中的节点名
Integer depth; //层数
}
```

返回结果

```
 {
String requestId;//请求的id
String returnCode;//0表示调用成功
String returnMessage;//返回执行的详细信息
List<NodeInfo> returnValue [{
    String nodeName; // 节点名称
    String nodeDisplayName; // 节点展示名称
    Integer runType; // 节点类型，参见RunType类
    Long fileId; // 代码编号
    Integer fileVersion; // 代码版本号
    String filePath; // 代码在压缩包的相对路径
    Integer nodeType; // 执行代码类型
    String description; // 描述信息
    String paraValue; // 参数信息
    Date startEffectDate; // 允许调度的起始日期
    Date endEffectDate; // 允许调度的终止日期
    String cronExpress; // cron表达式
    String owner; // 负责人账号
    String flowName;//流程名
    String projectName;//项目名
    Integer dependentType; // 依赖类型，0 - 不跨周期，1 - 用户自定义跨周期依赖， 2 - 跨一层
子节点依赖， 3 - 自依赖
    String source; // 来源
    String connection; // 连接串
    Date createTime; // 创建时间
    String createUser; // 创建人
    Date modifyTime; // 最新修改时间
    String modifyUser; // 最新修改人
    Integer priority; // 优先级
    String resourceGroupIdentifier; // 节点所属资源组标识
    Integer baselineId; // 节点所属基线编号
    Integer cycleType; // 周期类型，参见CycleType类
    List<NodeRelationInfo> parentRelations;//父节点信息
    List<NodeRelationInfo> childRelations;//子节点信息
```

```
},{},{}...]; //执行结果
}
```

3.5.1.4 向下查询子节点

调用方式

```
GET  http://baseapi.example.com/v2.0/node/project/{projectIdentifier}/flow/{flowName}/node/{
nodeName}/children
```

参数

```
 {
String projectName; //url中的项目名
String flowName; //url中的flowName
String nodeName; //url中的节点名
Integer depth; //层数
}
```

返回结果

```
 {
   String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      List<NodeInfo> returnValue [{
   String nodeName; // 节点名称
   String nodeDisplayName; // 节点展示名称
   Integer runType; // 节点类型，参见RunType类
   Long fileId; // 代码编号
   Integer fileVersion; // 代码版本号
   String filePath; // 代码在压缩包的相对路径
   Integer nodeType; // 执行代码类型
   String description; // 描述信息
   String paraValue; // 参数信息
   Date startEffectDate; // 允许调度的起始日期
   Date endEffectDate; // 允许调度的终止日期
   String cronExpress; // cron表达式
   String owner; // 负责人账号
   String flowName;//流程名
   String projectName;//项目名
   Integer dependentType; // 依赖类型，参见DependentType类
   String source; // 来源
   String connection; // 连接串
   Date createTime; //建时间
   String createUser; // 创建人
   Date modifyTime; // 最新修改时间
   String modifyUser; // 最新修改人
   Integer priority; // 优先级
   String resourceGroupIdentifier; // 节点所属资源组标识
   Integer baselineId; // 节点所属基线编号
   Integer cycleType; // 周期类型，参见CycleType类
   List<NodeRelationInfo> parentRelations;//父节点信息
   List<NodeRelationInfo> childRelations;//子节点信息
```

```
},{},{}...]; //执行结果
}
```

3.5.1.5 查询节点的代码

调用方式

```
GET  http://baseapi.example.com/v2.0/node/project/{projectIdentifier}/flow/{flowName}/node/{
nodeName}/code
```

参数

```
 {
String projectName; //url中的项目名
String flowName; //url中的flowName
String nodeName; //url中的节点名
 }
```

返回结果

```
 {
String requestId;//请求的id
String returnCode;//0表示调用成功
String returnMessage;//返回执行的详细信息
String returnValue; //执行结果,返回节点代码
}
```

3.5.1.6 批量更新节点

调用方式

```
GET  http://baseapi.example.com/v2.0/node/update/multiple/config
```

参数

```
{
List<ReplaceNodeConfigDto> replaceDtos;
}
```

ReplaceNodeConfigDto参数

```
{
String projectName;
String flowName;
String nodeName;
Integer baselineId;
String owner;
Integer resgroupId;
```

```
}
```

返回结果

```
 {
String requestId;//请求的id
String returnCode;//0表示调用成功
String returnMessage;//返回执行的详细信息
Boolean returnValue; //执行成功返回true
}
```

3.5.1.7 分页查询节点

调用方式

```
GET  http://baseapi.example.com/v2.0/node
```

参数

```
{
NodeSearchDto condition;
 }
```

NodeSearchDto参数

```
{
String runTypes; // 节点运行类型,多个以逗号分隔
String nodeTypes; // 节点类型,多个以逗号分隔
String projectName;//精确匹配
String flowName; //精确匹配
String owner; // 节点负责人
String modifyTime;//节点发布日期,格式yyyy-mm-dd
String searchText;//节点名称,模糊查询,nodename,displayname
Boolean includeRelation;//返回的结果中,是否包含依赖关系
 }
```

返回结果

```
{
String requestId;//请求的id
String returnCode;//0表示调用成功
String returnMessage;//返回执行的详细信息
List<NodeInfo> returnValue; /
Integer count; //返回结果数
```

```
}
```

3.5.1.8 根据节点名字查询节点

调用方式

```
GET  http://baseapi.example.com/v2.0/project/{projectIdentifier}/flow/{flowName}/node/{
nodeName}
```

返回结果

```
{
  String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    NodeInfo returnValue; //执行成功返回true
 }

NodeInfo
{
    String nodeName; // 节点名称
    String nodeDisplayName; // 节点展示名称
    Integer runType; // 节点类型，参见RunType类
    Long fileId; // 代码编号
    Integer fileVersion; // 代码版本号
    String filePath; // 代码在压缩包的相对路径
    Integer nodeType; // 执行代码类型
    String description; // 描述信息
    String paraValue; // 参数信息
    Date startEffectDate; // 允许调度的起始日期
    Date endEffectDate; // 允许调度的终止日期
    String cronExpress; // cron表达式
    String owner; // 负责人账号
    String flowName;//流程名
    String projectName;//项目名
    Integer dependentType; // 依赖类型，参见DependentType类
    String source; // 来源
    String connection; // 连接串
    Date createTime; // 创建时间
    String createUser; // 创建人
    Date modifyTime; // 最新修改时间
    String modifyUser; // 最新修改人
    Integer priority; // 优先级
    String resourceGroupIdentifier; // 节点所属资源组标识
    Integer baselineId; // 节点所属基线编号
    Integer cycleType; // 周期类型，参见CycleType类
    Integer projectId;// 项目ID
    List<NodeRelationInfo> parentRelations //父节点信息
    List<NodeRelationInfo> childRelations;//子节点信息
    String dqcDescription ;
    Integer dqcType ;
    Boolean isFlowType;//true表节点是Flow类型,false是节点类型
}

NodeRelationInfo
{
```

```
    String  projectName; //依赖的Node所属Flow的项目名
    String  flowName; //依赖的Node所属Flow名
    String  nodeName; //依赖的Node名
    Integer relationType; // 依赖方式，参见DependentType类,只有普通，自定义
    Integer projectId;// 项目ID
    }
```

3.5.1.9 新建节点

调用方式

```
POST  http://baseapi.example.com/v2.0/project/{projectIdentifier}/flow/{flowName}
```

参数

```
{
NodeCreateDto createDto
}
```

NodeCreateDto参数

```
{
String nodeName; // 节点名称
String nodeDisplayName; // 节点展示名称
Integer type; // 执行代码类型
String runType;//运行类型
Integer dependentType; // 依赖类型，参见DependentType类
String nodeCode;//节点对应的代码
String description; // 描述信息
String paraValue; // 参数信息
Date startEffectDate; // 允许调度的起始日期
Date endEffectDate; // 允许调度的终止日期
String owner; // 负责人账号
String source; // 来源
String connection; // 连接串
String scheduleExp; // 调度的cron表达式
List<String> parents;//上层节点
List<String> customDependentParents;//自定义依赖有父节点
}
```

返回结果

```
{
    String requestId;//请求的id
        String returnCode;//0表示调用成功
        String returnMessage;//返回执行的详细信息
        NodeInfo returnValue; //执行成功返回true
 }

NodeInfo
{
    String nodeName; // 节点名称
    String nodeDisplayName; // 节点展示名称
    Integer runType; // 节点类型，参见RunType类
```

```
    Long fileId; // 代码编号
    Integer fileVersion; // 代码版本号
    String filePath; // 代码在压缩包的相对路径
    Integer nodeType; // 执行代码类型
    String description; // 描述信息
    String paraValue; // 参数信息
    Date startEffectDate; // 允许调度的起始日期
    Date endEffectDate; // 允许调度的终止日期
    String cronExpress; // cron表达式
    String owner; // 负责人账号
    String flowName;//流程名
    String projectName;//项目名
    Integer dependentType; // 依赖类型，参见DependentType类
    String source; // 来源
    String connection; // 连接串
    Date createTime; // 创建时间
    String createUser; // 创建人
    Date modifyTime; // 最新修改时间
    String modifyUser; // 最新修改人
    Integer priority; // 优先级
    String resourceGroupIdentifier; // 节点所属资源组标识
    Integer baselineId; // 节点所属基线编号
    Integer cycleType; // 周期类型，参见CycleType类
    Integer projectId;// 项目ID
    List<NodeRelationInfo> parentRelations //父节点信息
    List<NodeRelationInfo> childRelations;//子节点信息
    String dqcDescription ;
    Integer dqcType ;
    Boolean isFlowType;//true表节点是Flow类型,false是节点类型
}

NodeRelationInfo
{
    String  projectName; //依赖的Node所属Flow的项目名
    String  flowName; //依赖的Node所属Flow名
    String  nodeName; //依赖的Node名
    Integer relationType; // 依赖方式，参见DependentType类,只有普通，自定义
    Integer projectId;// 项目ID
  }
```

3.5.1.10 更新节点

调用方式

```
PUT  http://baseapi.example.com/v2.0/node/project/{projectIdentifier}/flow/{flowName}
```

参数

```
{
   NodeUpdateDto updateDto;
}
```

NodeUpdateDto参数

```
{
```

```
    String nodeName; // 节点名称
    String nodeDisplayName; // 节点展示名称
    Integer type; // 执行代码类型
    String runType;//运行类型
    Integer dependentType; // 依赖类型，参见DependentType类
    String nodeCode;//节点对应的代码
    String description; // 描述信息
    String paraValue; // 参数信息
    Date startEffectDate; // 允许调度的起始日期
    Date endEffectDate; // 允许调度的终止日期
    String owner; // 负责人账号
    String source; // 来源
    String connection; // 连接串
    String scheduleExp; // 调度的cron表达式
    List<String> parents;//上层节点,格式"projectName.flowName.nodeName"
    List<String> customDependentParents;//自定义依赖有父节点
}
```

返回结果

```
{
    String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    Boolean returnValue; //执行成功返回true
}
```

3.5.1.11 删除节点

调用方式

```
DELETE  http://baseapi.example.com/v2.0/node//project/{projectIdentifier}/flow/{flowName}/
node/{nodeName}
```

返回结果

```
{
    String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    Boolean returnValue; //执行成功返回true
}
```

3.5.1.12 根据BaseID删除节点

调用方式

```
DELETE  http://baseapi.example.com/v2.0/node/project/{projectIdentifier}/baseid/{baseid}
```

返回结果

```
{
    String requestId;//请求的id
```

```
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    Boolean returnValue; //执行成功返回true
}
```

3.5.1.13 查询实例运行日志

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/log
```

参数

```
{
    Long nodeInsId; //url中的instanceId
}
```

返回结果

```
{
    String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    String returnValue; //执行结果返回日志
 }
```

3.5.1.14 查询实例历史运行日志

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns/{historyIntanceId}/historyLog
```

参数

```
{
    Long historyInsId; //url中的historyIntanceId
}
```

返回结果

```
{
    String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    String returnValue; //执行结果返回日志
```

```
}
```

3.5.1.15 更新任务调度配置进入当前实例

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/refresh
```

参数

```
{
   Long instanceId; //url中的instanceId
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
}
```

3.5.1.16 终止实例运行

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/kill
```

参数

```
{
   Long instanceId; //url中的instanceId
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
```

```
}
```

3.5.1.17 把节点实例设置成功,并唤醒下游实例

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/setSuccess
```

参数

```
{
   Long instanceId; //url中的instanceId
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
}
```

3.5.1.18 选择一批实例，从指定的实例开始，批量重跑

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/runMultiple
```

参数

```
{
    RunMultipleNodeInsDto runMultipleNodeInsDto;
}
```

RunMultipleNodeInsDto参数

```
{
    List<Long> includeNodeInsIds; //需要重跑的实例ID列表
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
```

```
}
```

3.5.1.19 重新运行节点实例

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/rerun
```

参数

```
{
   Long instanceId; //url中的instanceId
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
}
```

3.5.1.20 禁用节点实例

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/disable
```

参数

```
{
   Long instanceId; //url中的instanceId
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
```

```
}
```

3.5.1.21 恢复指定的暂停的节点实例

调用方式

```
PUT  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/enable
```

参数

```
{
   Long instanceId; //url中的instanceId
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue; //执行结果，成功返回true
}
```

3.5.1.22 分页查询节点实例

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns
```

参数

```
{
   NodeInsSearchDto condition;
}
```

NodeInsSearchDto参数

```
{
   String projectName;
   String flowName;
   String searchText; // 节点名称或displayname
   String instanceIds; // 多个实例ID,以逗号分隔
   Integer dagType; // DAG类型
   String statuses;//多状态
   String instanceTypes;//任务类型：0： 正常 1：一次性任务 2:表示暂停的节点实例 3：空跑

   Integer nodeType;//节点类型
   String owner; // 负责人，统一使用工号
   String bizdate; // 业务日期,格式为yyyy-MM-dd HH:mi:ss
   String bizBeginHour;//格式为yyyy-MM-dd HH:mi:ss
   String bizEndHour;//格式为yyyy-MM-dd HH:mi:ss
   String createTime;//格式为yyyy-MM-dd HH:mi:ss
```

```
   Long dagId;   //dag实例 ID
}
```

返回结果

```
{
      String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      List<NodeInsInfo> returnValue [{
   Long historyId; // 历史实例ID
   Long instanceId; // 任务实例ID
   Long dagId; // 任务实例所属的流程实例ID
   Integer instanceType; // 任务类型,参见InstanceType枚举
   Integer dagType; // DAG类型
   Integer status; // 任务状态
   Long opSeq; // 操作序列号
   Integer opCode; // 操作命令
   String owner; // 负责人，统一使用工号
   Date bizdate; // 业务日期
   Date gmtdate; // 处理日期
   Integer nodeType; // 执行代码类型，如odps_sql等
   Integer priority; // 优先级
   String paraValue; // 参数信息
   String projectName; // 任务实例所属应用ID
   Long relatedDagId; // 任务关联的工作流实例编号
   Date finishTime; // 任务结束时间
   Date beginWaitTimeTime; // 变成等待时间状态的时间
   Date beginWaitResTime; // 变成等待资源状态的时间
   Date beginRunningTime; // 变成运行中状态的时间
   Date createTime; // 创建时间
   String createUser; // 创建人
   Date modifyTime; // 最新修改时间
   String modifyUser; // 最新修改人
   Integer cycleType; // 周期类型
   Date cycleTime; // 周期时间
   Integer dependentType; // 依赖关系类型
   String nodeName; // 节点名称
   String flowName; // 根节点所属工作流定义
   Long inGroupId; // 该周期是当天的第几个周期
   String resourceGroupIdentifier; // 节点所属资源组标识
   Integer baselineId; // 任务所属基线编号
      },{},{}...]; //执行结果
      Integer count; //查询返回结果数
```

```
}
```

3.5.1.23 根据节点实例ID查询

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns/{instanceId}
```

返回结果

```
{
      String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      returnValue {
   Long historyId; // 历史实例ID
   Long instanceId; // 任务实例ID
   Long dagId; // 任务实例所属的流程实例ID
   Integer instanceType; // 任务类型,参见InstanceType枚举
   Integer dagType; // DAG类型
   Integer status; // 任务状态
   Long opSeq; // 操作序列号
   Integer opCode; // 操作命令
   String owner; // 负责人，统一使用工号
   Date bizdate; // 业务日期
   Date gmtdate; // 处理日期
   Integer nodeType; // 执行代码类型，如odps_sql等
   Integer priority; // 优先级
   String paraValue; // 参数信息
   String projectName; // 任务实例所属应用ID
   Long relatedDagId; // 任务关联的工作流实例编号
   Date finishTime; // 任务结束时间
   Date beginWaitTimeTime; // 变成等待时间状态的时间
   Date beginWaitResTime; // 变成等待资源状态的时间
   Date beginRunningTime; // 变成运行中状态的时间
   Date createTime; // 创建时间
   String createUser; // 创建人
   Date modifyTime; // 最新修改时间
   String modifyUser; // 最新修改人
   Integer cycleType; // 周期类型
   Date cycleTime; // 周期时间
   Integer dependentType; // 依赖关系类型
   String nodeName; // 节点名称
   String flowName; // 根节点所属工作流定义
   Long inGroupId; // 该周期是当天的第几个周期
   String resourceGroupIdentifier; // 节点所属资源组标识
   Integer baselineId; // 任务所属基线编号
}; //执行结果
      Integer count; //查询返回结果数
```

```
}
```

3.5.1.24 按层数查询父节点实例（包含工作流任务和节点）

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/parents?depth={depth}
```

返回结果

```
{
      String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      List<NodeInsInfo> returnValue [{
   Long historyId; // 历史实例ID
   Long instanceId; // 任务实例ID
   Long dagId; // 任务实例所属的流程实例ID
   Integer instanceType; // 任务类型,参见InstanceType枚举
   Integer dagType; // DAG类型
   Integer status; // 任务状态
   Long opSeq; // 操作序列号
   Integer opCode; // 操作命令
   String owner; // 负责人，统一使用工号
   Date bizdate; // 业务日期
   Date gmtdate; // 处理日期
   Integer nodeType; // 执行代码类型，如odps_sql等
   Integer priority; // 优先级
   String paraValue; // 参数信息
   String projectName; // 任务实例所属应用ID
   Long relatedDagId; // 任务关联的工作流实例编号
   Date finishTime; // 任务结束时间
   Date beginWaitTimeTime; // 变成等待时间状态的时间
   Date beginWaitResTime; // 变成等待资源状态的时间
   Date beginRunningTime; // 变成运行中状态的时间
   Date createTime; // 创建时间
   String createUser; // 创建人
   Date modifyTime; // 最新修改时间
   String modifyUser; // 最新修改人
   Integer cycleType; // 周期类型
   Date cycleTime; // 周期时间
   Integer dependentType; // 依赖关系类型
   String nodeName; // 节点名称
   String flowName; // 根节点所属工作流定义
   Long inGroupId; // 该周期是当天的第几个周期
   String resourceGroupIdentifier; // 节点所属资源组标识
   Integer baselineId; // 任务所属基线编号
      },{}..{}]; //执行结果
      Integer count; //查询返回结果数
```

```
}
```

3.5.1.25 按层数查询子节点实例（包含工作流任务和节点）

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/children?depth={depth}
```

返回结果

```
{
      String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      List<NodeInsInfo> returnValue [{
   Long historyId; // 历史实例ID
   Long instanceId; // 任务实例ID
   Long dagId; // 任务实例所属的流程实例ID
   Integer instanceType; // 任务类型,参见InstanceType枚举
   Integer dagType; // DAG类型
   Integer status; // 任务状态
   Long opSeq; // 操作序列号
   Integer opCode; // 操作命令
   String owner; // 负责人，统一使用工号
   Date bizdate; // 业务日期
   Date gmtdate; // 处理日期
   Integer nodeType; // 执行代码类型，如odps_sql等
   Integer priority; // 优先级
   String paraValue; // 参数信息
   String projectName; // 任务实例所属应用ID
   Long relatedDagId; // 任务关联的工作流实例编号
   Date finishTime; // 任务结束时间
   Date beginWaitTimeTime; // 变成等待时间状态的时间
   Date beginWaitResTime; // 变成等待资源状态的时间
   Date beginRunningTime; // 变成运行中状态的时间
   Date createTime; // 创建时间
   String createUser; // 创建人
   Date modifyTime; // 最新修改时间
   String modifyUser; // 最新修改人
   Integer cycleType; // 周期类型
   Date cycleTime; // 周期时间
   Integer dependentType; // 依赖关系类型
   String nodeName; // 节点名称
   String flowName; // 根节点所属工作流定义
   Long inGroupId; // 该周期是当天的第几个周期
   String resourceGroupIdentifier; // 节点所属资源组标识
   Integer baselineId; // 任务所属基线编号
      },{}..{}]; //执行结果
      Integer count; //查询返回结果数
```

```
}
```

3.5.1.26 查询节点的历史实例信息

调用方式

```
GET  http://baseapi.example.com/v2.0/nodeIns/{instanceId}/history?start={start}&limit={limit}
```

返回结果

```
{
      String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      List<NodeInsInfo> returnValue [{
   Long historyId; // 历史实例ID
   Long instanceId; // 任务实例ID
   Long dagId; // 任务实例所属的流程实例ID
   Integer instanceType; // 任务类型,参见InstanceType枚举
   Integer dagType; // DAG类型
   Integer status; // 任务状态
   Long opSeq; // 操作序列号
   Integer opCode; // 操作命令
   String owner; // 负责人，统一使用工号
   Date bizdate; // 业务日期
   Date gmtdate; // 处理日期
   Integer nodeType; // 执行代码类型，如odps_sql等
   Integer priority; // 优先级
   String paraValue; // 参数信息
   String projectName; // 任务实例所属应用ID
   Long relatedDagId; // 任务关联的工作流实例编号
   Date finishTime; // 任务结束时间
   Date beginWaitTimeTime; // 变成等待时间状态的时间
   Date beginWaitResTime; // 变成等待资源状态的时间
   Date beginRunningTime; // 变成运行中状态的时间
   Date createTime; // 创建时间
   String createUser; // 创建人
   Date modifyTime; // 最新修改时间
   String modifyUser; // 最新修改人
   Integer cycleType; // 周期类型
   Date cycleTime; // 周期时间
   Integer dependentType; // 依赖关系类型
   String nodeName; // 节点名称
   String flowName; // 根节点所属工作流定义
   Long inGroupId; // 该周期是当天的第几个周期
   String resourceGroupIdentifier; // 节点所属资源组标识
   Integer baselineId; // 任务所属基线编号
      },{}..{}]; //执行结果
      Integer count; //查询返回结果数
```

```
}
```

3.5.1.27 以补数据方式调起一个工作流任务中的一批节点

调用方式

```
POST  http://baseapi.example.com/v2.0/dag/single/flow/multiple/node/manual
```

参数

```
{
   CreateMultipleNodeInsDto dto;
}
```

CreateMultipleNodeInsDto属性

```
{
      String instanceName; // 工作流实例名称,非空
      String projectName;//项目名，非空
      String flowName; //工作流名称，非空
      Date startDayBizdate; //业务起始时间,天任务使用，非空
      Date endDayBizdate; //业务起始时间,天任务使用，非空
      String startHourBizdate; //小时任务的业务时间,配合startDayBizdate,endDayBizdate一起使
用
      String endHourBizdate; //小时任务的业务时间
      String startNodeName;//本批次调起的节点,从哪个节点开始运行
      List<NodeDto> nodeList;//批量调起的节点，非空
}
```

返回结果

```
{
      String requestId;//请求的id
      String returnCode;//0表示调用成功
      String returnMessage;//返回执行的详细信息
      List<Long> returnValue;//执行成功返回调起的节点id列表
}
```

3.5.1.28 以冒烟测试的方式调起一个工作流任务中的一个节点

调用方式

```
POST  http://baseapi.example.com/v2.0/dag/single/flow/multiple/node/smoke
```

参数

```
{
   SmokeNodeInsDto dto;
```

```
}
```

SmokeNodeInsDto属性

```
{
   String instanceName; // 工作流实例名称，非空
   String projectName;//项目名，非空
   String flowName;//工作流名称,startFlowId表示本批次从哪个Flow开始触发
   String nodeName; //节点名称
   Date bizdate; //业务起始时间,天任务使用
   String startHourBizdate; //小时任务的业务时间,配合startDayBizdate,endDayBizdate一起使用

   String endHourBizdate; //小时任务的业务时间
}
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Long returnValue;//执行成功返回调起的节点id
}
```

3.5.1.29 终止DAG进程

调用方式

```
POST  http://baseapi.example.com/v2.0/dag/{dagId}/kill
```

返回结果

```
{
   String requestId;//请求的id
   String returnCode;//0表示调用成功
   String returnMessage;//返回执行的详细信息
   Boolean returnValue；//成功返回true
}
```

3.5.1.30 提交ZIP包

调用方式

```
POST  http://baseapi.example.com/v2.0/submission/project/{projectIdentifier}
```

参数

```
 {
      InputStream is;        //zip 文件对应流
      FormDataContentDisposition fdcd;
```

```
}
```

返回结果

```
{
    String requestId;//请求的id，本接口为异步提交，需要根据请求id查询提交状态
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    Boolean returnValue;//添加成功返回true
}
```

3.5.1.31 查询ZIP包提交状态

调用方式

```
POST  http://baseapi.example.com/v2.0/submission/project/{projectIdentifier}/{requestId}
```

返回结果

```
{
    String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    AsyncSubmissionLogInfo returnValue [{
  String requestId;
  Date createTime;//接受提交请求时间
  String name;//生成的名称标识,格式为[调用方的系统名-backToCheckId-createTime]
  String status;//success：执行成功，fail:执行失败,running：正在执行
  String message;     }];
}
```

3.5.2 接口鉴权

Header

```
base_id(必填)
tenant_id(必填)
local_name(可选，默认：zh_CN)
```

安全

调用API的接口，都需要经过签名， 调用方需做以下操作。

1. 在url中增加2个固定字段 ：baseKey，timestamp。
2. 生成签名，加到http header中，名为signature。
   - 以GET方法为例：

     假设B提供了http接口为 /getapi，需要的query parameter分别为name，age。A原有的调用url应该是/getapi?name=fengyeng&age=20，接入token中心后， 在query parameter中增

加了两个参数，所以新的url应该是/getapi?name=fengyeng&age=20?baseKey=dsa134&timestamp=123456789， 随后利用新的url中的query parameter结合token生成signature， 再在header中增加一个变量signature:dsafadfsa141fdsaf1 。

- 以POST方法为例：

  假设B提供了http接口为 /postapi，A原有的调用url应该是/postapi，接入token中心后，在query parameter中增加了两个参数，所以新的url应该是/postapi? baseKey=dsa134&timestamp=123456789，随后利用新的url中的query parameter结合token生成signature，再在header中增加一个变量signature:dsafadfsa141fdsaf1。

3.5.3 返回值

返回结构为JSON格式，包含内容统一为requestId, requestCode, requestMessage, returnValue。如果接口调用失败，则根据requestCode/requestMessage可获知基本报错信息；根据requestId可在系统所有后台日志中追溯接口调用情况，获取详细的调用信息。如果接口调用成功则requestCode为0，如有返回值则作为returnValue返回。

```
{
    String requestId;//请求的id
    String returnCode;//0表示调用成功
    String returnMessage;//返回执行的详细信息
    Object returnValue;
}
```

4 大数据应用加速器

4.1 前言

概述

本文档全面系统地给出了阿里云大数据应用加速器DTBoost中各相关对象的API定义，通过完整的参数说明和使用示例，帮助开发人员了解阿里云数据应用加速器可编程接口的使用。

阅读对象

本文档可作为开发人员在对阿里云大数据应用加速器DTBoost的模型和架构有基本了解的情况下，进行二次开发和数据收集的参考手册。

4.2 产品简介

大数据应用加速器（DTBoost）是阿里云结合阿里巴巴自身大数据应用场景，经过多年总结抽象出的企业级大数据应用平台，它的目标如下所示。

- 让业务人员可以快速的理解数据，应用数据。
- 数据模型设计，重计算模型设计。
- 结构开放，快速支撑数据应用开发。
- 支撑企业内部共建共享，协同开发。

大数据应用加速器除了提供UI交互界面之外，还有一组强大API接口，可以方便的完成系统集成，大大降低应用开发成本。

4.3 API调用方式

4.3.1 接口访问

接口的访问地址为：http://{产品前缀}.example.com/${apipath}

以下*API列表*中列出的API地址，都是 ${apipath} 部分。

4.3.2 接口鉴权

接口的鉴权形式采用私有秘钥签名的形式。

签名算法

获取签名所需的信息。

$HTTP_METHOD

大写，如： POST，GET

```
$URL
```

服务器收到的请求path + querystring， querystring 中必须包含 timestamp 参数，签名校验会通过该字段检查签名的时效性。

说明：

- timestamp 为unix-timestamp * 1000 ，即毫秒单位。
- querystring需要urldecode。

```
$BODY
```

http请求的body部分。

计算签名

每个租户都有自己的一个 $secretToken，这个token可以在DTBoost的**控制台** > **个人中心**页面找到，如图 *4-1: 个人中心*所示。

图 4-1: 个人中心

签名的方式有两种：

- 带body的场景，如post请求

```
$HTTP_METHOD + '\n' +
$URL + '\n' +
```

```
md5($BODY)
```

- 不带body的场景，如get请求

```
$HTTP_METHOD + '\n' +
$URL + '\n' +
```

签名的hash算法为： hmac sha-256，secretkey 为前面提到的$secretToken。

Node.js 版本的签名代码

```
const crypto = require('crypto');
const urllib = require('urllib');
function hmacsha256(str, secret) {
  const hash = crypto.createHmac('sha256', secret);
  hash.update(str, 'utf8');
  return hash.digest('hex');
}
let token = '您的token信息'; // token框架会提供
let timestamp = Date.now();
let str =  'GET\n/otm/api/categories/download?category_id=10001&timestamp='+timestamp;
let signature = hmacsha256(str, token);
let options = {
  headers: {
    signature: signature
  }
};
let url = 'http://$domain/otm/api/categories/download?category_id=10001&timestamp=
1461502358436'
urllib.request(url, options, function (err, result) {
  if (err) {}
  ...
});
```

4.3.3 接口版本控制

当一个接口的 ${apipath} 部分设计了类似 /api/v1/xxxx 结构， 则说明接口存在版本区分，注意查看不同版本的接口说明，并选择合适的版本调用。

当接口路径不包含版本号时，说明该接口始终保持相同的调用方式。

4.3.4 返回格式

请求接口的时候，可能有如下httpCode返回：

- 504：请求超时，请联系阿里云技术支持获得帮助。
- 500：接口异常，请联系阿里云技术支持获得帮助。
- 403：没有权限，可能是资源访问权限受限，请先申请查询对象的访问权限。
- 200：接口请求成功。

当接口httpCode为**200** 的时候，接口返回为一个JSON格式的数据， 格式包含如下：

```
{
"code" :  "SUCCESS",   // {String} 请求返回状态码，字符串类型，大写字母
"data": [],        // {Array} 请求成功之后获得的返回数据
"message": null   // {null|String} 如有异常，返回执行异常时的详细信息
}
```

4.4 API列表

4.4.1 标签中心API

主要是标签数据更新回调接口。

调用方式

```
POST  http://dtboost.example.com/analyse/api/query
```

参数

```
 {
"tql": "select * from object",  // {String} 查询的tql
"extra": {
 "schema_code": "ads_test" , // {String} 云计算资源名称
 "entity": "user" // {String} 实体名称
},
"page_size":  10,  // {Int} [可选]，分页大小，默认不分页
"page_no": 2     // {Int} [可选]页码
 }
```

返回结果

```
 {
"code": "SUCCESS",   // {String} 返回code
"data": [{} ,  {} ,  {}],    // {Array} TQL查询返回的数据集合
"message": null    // {null|String} 如果异常，则返回详细的错误信息String，如果异常来自于计算引擎，请查阅计算引擎的错误码参考资料
}
```

说明：

Code 可以为：

- SUCCESS 查询成功。
- ERROR 查询失败，失败原因在 message中。

4.4.2 整合分析API

4.4.2.1 TQL查询接口

调用方式

POST http://dtboost.example.com/analyse/api/query

参数

```
 {
"tql": "select * from object",  // {String} 查询的tql
"extra": {
 "schema_code": "ads_test" , // {String} 云计算资源名称
 "entity": "user" // {String} 实体名称
},
"page_size":  10,  // {Int} [可选]，分页大小，默认不分页
"page_no": 2     // {Int} [可选]页码
 }
```

返回结果

```
 {
"code": "SUCCESS",   // {String} 返回code
"data": [{} ,  {} ,  {}],    // {Array} TQL查询返回的数据集合
"message": null    // {null|String} 如果异常，则返回详细的错误信息 String，如果异常来自于计算引擎，请查阅计算引擎的错误码参考资料
}
```

说明：

Code 可以为：

- SUCCESS 查询成功。
- ERROR 查询失败，失败原因在 message中。

4.4.2.2 Expr查询接口

调用方式

```
POST  http://dtboost.example.com/analyse/api/query/json
```

参数

```
{
"expr":  {},  // {Object} url中的项目名
"extra": {
 "schema_code": "ads_test", // {String} 云计算资源名称
 "entity": "user"            // {String} 实体名称
},
```

```
"page_size": 10   // {Int} [可选] 分页大小，默认不分页
"page_no": 2      // {Int} [可选] 第几页
}
```

返回结果

```
{
"code": "SUCCESS",        // {String} 返回code
"data": [{} ， {} ， {}],   // {Array} TQL查询返回的数据集合
"message":  null          // {null|String} 如异常，则返回详细的异常信息，如计算引擎异常，请查阅对应计算引擎的异常码手册
}
```

该接口返回数据和 TQL接口一致。

4.4.2.3 JQL查询接口

调用方式

```
POST  http://dtboost.example.com/analyse/api/query/json
```

参数

```
{
"json":  {}, //url中的项目名
"extra": {
 "schema_code": "ads_test", // {String} 云计算资源名称
 "entity": "user"           // {String} 实体名称
},
"page_size": 10,   // {Int} [可选] 分页大小，默认不分页
"page_no": 2       // {Int} [可选] 第几页
}
```

返回结果

```
{
"code": "SUCCESS", // {String} 返回code
"data": [{} ， {} ， {}], // {Array} TQL查询返回的数据集合
"message": null // {null|String} 如异常，则返回详细的异常信息，如异常由于计算引擎引擎，请查阅相应计算引擎的错误码手册
}
```

该接口返回数据和 TQL 接口一致。

4.4.2.4 TQL语法帮助

TQL（Tag Query Language）整合分析提供的基于OLT的标签查询语言，实现基于标签的分析查询。它对用户屏蔽了物理模型和物理存储，用户更容易进行分析查询。

TQL的语法定义如下：

```
SELECT [distinct] tag_expr_list FROM ol_reference [WHERE where_condition] [GROUP BY
tag_expr_list] [ORDER BY order_condition] [LIMIT num]
```

简单查询

- 查询某个用户的性别：

```
select user.gender from user where user.uid = xxx;
```

- 计算所有类目下的成交笔数：

```
select trade.cateid,count(*) as cnt from trade group by trade.cateid;
```

使用group by...having

```
select sum(user.age) as s, user.location from user group by location having s > 1000;
```

Inner Join

```
select user.age,count(*)  from user join trade on user.uid=trade.user.uid  where trade.cateid=
xxx;
```

Left Join/Right Join

```
select trade.cateid, user.age from user left join trade on (user.uid=trade.user.uid)   where trade.
date > '2015-09-07';
```

子查询

```
select
    a.uid, a. amt_pay,
b.collect_cnt
from (
 select
  sum(trade.amt_pay) as amt_pay,
  trade.user.uid as uid
 from
  trade
 group by trade.user.uid
) as a left join (
 select
  count(*) as collect_cnt,
  collect.user.uid as uid
 from
                 collect
 group by collect.date
```

```
) as b on (a.uid =b.uid);
```

Function

TQL 对于函数的支持依赖于所使用的执行引擎，即查询执行的数据库系统除此之外，TQL还支持如下函数：

```
* udfprofile 计算百分比sql
```

```
 select user.age_level, udf_profile(user.uid) from user group by        user.age_level;
```

```
udfsegment 分段函数
```

```
select udf_segment(user.age, '1:[0,18];2:(18,28);3:(28,50);4') as age_level from user;等价于sql
 select case when user.age >= 0 and user.age <= 18 then 1 when user.age > 18 and user.age
 <= 28 then 2 when user.age > 28 and user.age <= 50 then 3 else 4 end as age_level from user
;
```

4.4.2.5 Expr语法帮助

查询表达式的结构

表达式是一个JSON结构，共有6个元素：return，target，variable，filter，limit，order by（不区分大小写和先后顺序）。

```
{
   "RETURN":   //数组格式，必选项，申明查询的返回结果
   "TARGET":   //数组格式，必选项，申明分析的对象
   "VARIABLE": //可选，json 格式，子查询的逻辑在此表达，子查询返回的还是Object、Link
"FILTER":   //可选，字符串格式，过滤条件
"limit":    //可选，字符串格式， limit
"orderby"  //可选，字符串格式，排序
}
```

表达式可以嵌套

variable对应于SQL里的子查询，可以嵌套表达。

```
// 这个例子里，$object1、$object2 都来自嵌套的子查询
{
   "TARGET": ["object->$object1->$object2"]
   "RETURN": ["object.tag", "object1.tag1", "object2.tag2"]
   "VARIABLE": {
"object1":{
   "TARGET":["user"],
   "RETURN":["user.tag1"]
},
"object2":{
    "TARGET":["User2->$ojbect3"]  //object2再次嵌套object3
    "RETURN":["User2.tag2","object3.tag3"]
    "VARIABLE":{
        "object3":{
```

```
            "TARGET":["User3"]
            "RETURN":["User3.tag3"]
        }
    }
}
  }
}
```

RETURN：期望返回的结果

其中：

- RETURN里可以加别名。
- 子查询里RETURN的必须加别名。

```
{
  "TARGET":["User2->$ojbect3"]  //object2再次嵌套object3
  "RETURN":["User2.tag2","object3.tag3"] //object3的tag3来自 User3.tag3
  "VARIABLE":{
"object3":{
    "TARGET":["User3"]
          "RETURN":["User3.tag3 as tag3"]
}
   }
}
```

TARGET

TARGET分两类，共三种：object，link，子查询。

TARGET中申明对象的时候可以加过滤条件，取子集：

- 十二月份的成交：trade(month=201512)
- 女装类目的商品： auction(cat_name='女装')
- 上海的女性用户：user(gender='1' and city='shanghai')

TARGET之间的join关系：

- join的类型
  - inner join
  - outer join，且只有left outer join，如需right join，可以通过调整join的方向来转换。
- left join的方向
  - user->trade => user left join trade
  - trade->user => trade left join user
- join的连接点joinKey

- object/link和子查询join，必须指定joinKey。
- link和link之间的join，必须指定joinKey。
- 其他情况，由系统根据OTM的元数据自动决定joinKey。

Join Cases: :

- Join Case 1：Object join

  Object 根据外键做join，不需要指定joinKey，由系统根据otm元数据自动决定，前提是注册标签的时候指定了外键。

```
target:["auction-shop"]  // 商品join店铺，auction上有一个外键shop_id(商品所属的店铺)指向shop
target:["设备->供应商"],  // 设备join供应商，通过设备上的外键(供应商ID) 来和供应商来join  统计每个店铺的商品数量
{
  "target":["auction-shop"],
   "return":["shop.name", "count(auction.id) as aucs"]
}
```

- Join Case 2：Object join

  Link 根据外键做join，不需要指定joinKey，由系统根据otm元数据自动决定。

```
target:["auction-trade"] //商品join交易
target:["auction-trade->shop"] //商品join交易，交易join店铺 target:["auction-trade", "auction-shop"] //商品join交易，商品再join店铺
```

- Join Case 3：Object join

  子查询和子查询进行join，必须要指定 joinKey。

  分析对象为5月份成交大于500的店铺，期望返回结果为统计每个店铺的商品数量，auction 和shop1 这个子查询做join，子查询必须指定joinKey 。

```
{
  "target":["auction-$shop1[shop_id]"],
  "return":["shop1.shopname", "count(auction.id) as aucs"],
  "variable":{
"shop1":{
   "target":["shop-trade(month=201605)"],
   "return":[
 "shop.shop_id as shop_id",
 "shop.name as shopname",
 "sum(trade.gmv) as month5_total_gmv"
   ],
   "filter":"sum(trade.gmv)>500"
 }
  }
}
```

- Join Case 4: Link和子查询

Link和子查询之间的join，必须要指定joinKey

```
"target":["trade[user.uid,shop.shopid] -> $search1[user_id, shop_id]"]
```

相当于 TQL：

```
SELECE XX FROM trade FROM (search子查询) as search1 on(trade.user.uid=
search1.user_id     and trade.shop.shopid=search1.shopid )
```

- Join Case 5：子查询和子查询join

  子查询之间的join，必须要指定joinKey

```
"target":["$trade1[uid,shopid] -> $search1[user_id, sellerid]"]
```

  相当于 TQL：

```
SELECT XX FROM (trade子查询) as trade1 FROM (search子查询) as search1 on(
trade1.uid=search1.user_id     and trade1.shopid=search1.sellerid  )
```

- Join Case 6：Link和Link

  Link之间的join，必须要指定joinKey

```
"target":["trade[user.uid,shop.shopid] -> search[user.uid,shop.shopid]"]
```

VARIABLE详解

分析表达式通过variable来承载嵌套的逻辑，可以理解为SQL中的子查询。

- 子查询的return元素，必须加别名，如： user.user*id as user*id。
- target元素里，引用子查询的对象是必须要加$，如：$user1，其他地方引用子查询对象时不能加$。

```
{
   "TARGET":["$shop1"]   //子查询的对象名要加$,
   "RETURN": ["shop1.star", "shop1.id"]  //其他地方引用子查询时不需要加$,
   "FILTER":"shop1.star>1"     ,
   "ORDER BY":"shop1.star"     ,
   "VARIABLE":{
"shop1":{
   "RETURN":["shop.star","sum(trade.amt) as amt"]
   "TARGET":["shop-trade"]
}
   }
}
```

子查询的表达：

- 子查询Case 1：

分析对象：店铺的成交，按店铺星级统计平均成交金额。

子查询：在子查询里汇总店铺的总金额， 最外面求金额的平均值。

```
{
  "RETURN": ["shop.star", "avg(shop.amt) as avg_amt"]
  "TARGET":["$shop"] //子查询的对象要加
  "VARIABLE":{
      shop:{
        "RETURN":["shop.star","sum(trade.amt) as amt"]
         "TARGET":["shop-trade"]
     }
  }
}
```

等价的 TQL：

```
select
  shop.star,
  avg(shop.totoal_amt)
from (     //子查询，算出每个店铺的总金额
  select
shop.id,
shop.star,
sum(trade.amt) as amt
  from shop join trade
on(shop.shop_id=trade.shop.shop_id)
  group by shop.id,shop.star
) shop
group by shop.star
```

- 子查询Case 2：

  分析目标：在线商品数大于10 并且月成交大于10000 的店铺。

  分析结果：各个星级的店铺数。

  子查询：

  1. 店铺的在线商品数：店铺join商品。
  2. 店铺的当月成交金额：店铺join成交。

```
{
  "RETURN":["shop.star","count(shop.shopid) as shops"]
  "TARGET":[
"shop-$shop_join_auctions[shopid]->$shop_join_trade[shopid]"
  ]
  "VARIABLE":{
    "shop_join_auction":{
      "RETURN":[
        "shop.shopid as shopid",
        "shop.star as star",
        "count(auction.id) as items"
      ],
      "TARGET":["shop-auction(is_online='true')"]
```

```
    },
    "shop_join_trade":{
      "RETURN":[ "shop.shop_id as shopid", "sum(trade.amt) as total_amt" ], "TARGET":["
shop-trade"],
      "FILTER":"trade.month='201601'"
    }
  }
  "FILTER":"shop_join_auctions.items > 10 and shop_join_trade.total_amt > 10000"
}
```

等价的 TQL：

```
select
  shop.star,
  count(shop.id) as shops
from shop join(
  select
    shop.shop_id,
    count(auction.id) as items
  from shop join auction
  where
    auction.is_online="true"
  group by shop.shop_id
)shop_join_auc on (shop.shopid= shop_join_auc.shopid)
join(
  select
    shop.shopid,
    sum(trade.amt) as total_amt
  from shop join trade
  group by shop
) shop_join_trade on(shop.shopid= shop_join_trade.shopid)
where
  shop_join_auc.items>10 and
  shop_join_trade.total_amt>10000
group by star
```

4.5 附：术语与缩略语

4.5.1 基本术语

实体（Object）

即客观世界的一个对象，如人员、车辆、飞机、火车、酒店等。

关系（Link）

描述实体和实体之间发生的关联。如用户实体购买了商品实体，购买记录表就是一个 Link，它连接了用户实体和商品实体。

标签（Tag）

标签是实体的属性，如人员实体的标签可以有：性别、年龄等。

4.5.2 缩略词

TQL

标签查询语言（Tag Query Language）是分析接口支持的一种查询语言，基于实体、关系、标签的一种类似SQL的查询语法。其语法和SQL中的select语法非常相识，但不支持update、insert、delete等其他语法。

JQL

结构化标签查询语言（JSON Query Language）是分析接口支持查询输入语言，基于实体、关系、标签的描述，JSON格式的语法。其语法表达的含义和TQL相同。

5 关系网络分析

5.1 前言

概述

本文档详细探讨了阿里云 关系网络分析(Graph Analytics) 支持的 API 操作和相关的示例。

应用范围

使用本产品前，用户需满足必要的技能要求。推荐云产品开发工程师使用本文档。

5.2 简介

欢迎使用阿里云关系网络分析服务（Graph Analytics Service），简称（I+ Service）。用户可以使用本文档介绍的API对I+服务进行相关操作。

请确保在使用这些接口前，已充分了解了I+产品说明、使用协议和收费方式。

5.2.1 术语表

术语	全称	中文	说明
object		实体	
property		属性	
link		关系	
objectType		实体类型	对应后台管理配置的O编码，例O0001。
linkType		关系类型	对应后台管理配置的L编码，例L0001。
propertyId/propId		属性ID	对应实体与关系的属性编码例：O0001P001，L0001P001。
propertyType		属性查询类型	枚举： string_equel， 条件全词匹配。 string_like，条件模糊匹配。 number，数值查询。

术语	全称	中文	说明
			time，date，时间格式。 dict_select，全词匹配。
propertyList		基本属性条件列表	
derivated		间接关系条件列表	经过一定逻辑组装的属性条件，因编码暂未开放，现在可以使用后台默认配置。
statPropList		累计属性条件列表	经过sum，count计算的累计属性条件。
degree		度数	与某实体有直接联系的关系叫一度关系，两个点中经过一个间接点叫两度，经过两个间接点叫三度，以此类推。
direct/d		关系方向	关系的数据流向，如A给B打电话，则方向为A->B。

5.2.2 业务限制资源规格限制说明

在接口说明部分，凡出现对参数可选值、可用规格方面与官网上给出的资源规格限制发生矛盾时，均以官网上给出的值为准。

5.2.3 其他说明

关系网络中非常重要的过滤条件格式说明：

```
{
        "propertyId":"L0001P001",
        "propertyType":"string_equal",
        "values":[""]
    }
```

所有时间格式的输入包括date/time，都只支持10位时间戳格式，getTime()/1000。

场景一

propertyType=string_equal/string_like，values支持多个值输入，多个值为“或”。

场景二

propertyType=time/date/number，values为两个值的数组。例如：

- [10,20] 表示大于等于10且小于等于20。
- [10,10] 表示等于10。
- [10,*] 表示大于等于10。
- [*,20] 表示小于等于20。

场景三

propertyType=dict_select，values支持多个码值输入，多个值间关系为“或”。

5.3 更新历史

发布时间	更新	说明
2017-05-11	新增I+产品V1.8版本提供的API服务。	接口包括：实体查询服务接口、关联反查服务接口、默认关联反查服务接口、群集分析服务接口、共同邻居服务接口、路径分析服务接口。
2017-08-10	• 调整发布的接口。 • 增加鉴权。 • 增加流量控制。 • 统一了入参格式，增加接口参数说明。 • 增加OLP转义支持。	增加搜索及数据持久化相关的接口，修改了默认关联反查，关联反查，群集分析，共同邻居，路径分析，血缘分析接口。

5.4 API概览

分类列举各个接口。

5.4.1 搜索服务API

API	描述
/rest/search/queryNodes.json 详见5.实体查询服务	按条件查询节点信息。
/rest/search/queryLinks.json 详见6.关系查询服务	按条件查询关系信息。

5.4.2 关系网络服务API

API	描述
/rest/graph/getFastGraph.json 详见7.默认关联反查服务	一键多度查询：单个节点出发，扩展指定度数的关系网络路径。
/rest/graph/getGraph.json 详见8.关联反查服务	关联反查：单个或多个节点出发，获取节点关系网络路径。
/rest/graph/getGroupGraph.json 详见9.群集分析服务	群集分析：获取多个节点之间的直接关联的网络路径。
/rest/graph/commonLinkGraph.json 详见10.共同邻居服务	共同邻居：获取与输入的N个点同时存在关系的路径对象。
/rest/graph/pathAnalysisGraph.json 详见11.路径分析服务	路径分析：获取两个节点之间的最短网络路径或全路径，最多查询6度内的直接关系。
/rest/graph/kinshipGraph.json 详见12.血缘分析服务	血缘分析：多度关系的一种业务场景，查询一个点的同类关系多度扩展。

5.4.3 数据持久化服务API

API	描述
/rest/etl/persistNode.json 详见13.虚拟节点持久化服务	虚拟节点属性持久化。
/rest/etl/persistLink.json 详见14.添加关系持久化服务	增加关系持久化。

5.4.4 OLP转义API

API	描述
/rest/meta/queryOlpMetaConfig.json 详见15.查看OLP配置信息	查看OLP配置信息映射。
/rest/meta/updateOlpMetaConfig.json 详见16.添加关系持久化服务	更新OLP变量与业务变更的映射。

5.5 调用方式

5.5.1 请求结构

5.5.1.1 服务地址

环境	服务地址	备注
内部测试环境	---	无。
生产环境	---	根据实际部署的环境咨询PE。

5.5.1.2 通信协议

通讯协议采用HTTP协议。

5.5.1.3 请求方法

对 I+ API 接口调用是通过向 I+ API 的服务端地址发送HTTP POST请求，并按照接口说明在请求中加入相应请求参数来完成的；根据请求的处理情况，系统会返回处理结果。

5.5.1.4 请求参数

请求参数采用Json格式，application/json协议，requestBody请求。

5.5.1.5 字符编码

字符编码采用UTF-8编码。

5.5.2 公共参数

公共请求参数

I+提供的服务接口请求参数的结构为Json格式，放到请求的requestBody中，示例如下：

```
{
    "objectType": "O0003",
    "objectvalue": [
      "O0003P0004-371411371416495795"
    ]
    }
```

公共返回参数

I+提供的服务接口返回参数的结构为Json格式，Json结构中节点data内查询的返回的结果，elapsedTime表示接口的查询时间，noteMsg表示错误信息，success表示接口调用是否成功。

示例

```
{
```

```
  "data": {
    "groups": [],
    "linkCnt": 0,
    "linkDetails": {},
    "linkPropMaps": {},
    "linkProps": {},
    "links": [],
    "nodeCnt": 0,
    "nodePropMaps": {},
    },
    "nodesProps": {}
  },
  "elapsedTime": 59,
   "errorcode": 0,
  "noteMsg": "",
  "success": true
}
```

5.5.3 返回结果

成功结果

I+提供的服务接口返回参数的结构为Json格式，Json结构中success为true表示接口调用成功，成功结果保存在data中。

例如：

```
{
  "data": {
  },
  "elapsedTime": 167,
  "errorcode": 0,
  "noteMsg": "",
  "success": true
}
```

错误结果

I+提供的服务接口返回参数的结构为Json格式，Json结构中success为false表示接口调用失败，失败原因保存在noteMsg中。

例如：

```
{
  "data": {
  },
  "elapsedTime": 167,
  "noteMsg": "011005:用户未登录",
  "success": false
```

```
}
```

公共错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.5.4 接口限制

请求限制

避免计算资源紧张，保证I+系统稳定，每个SID同时接收请求N(10)个，超过的请求轮循等待10S，还没有请求释放，返回系统繁忙，请稍后再试。

5.5.5 鉴权

首次调用或session过期，返回011005错误码，表示没有权限

鉴权请求

```
post http://ip:7001/login.json
{"userNum":"xianyi","password":"e10adc3949ba59abbe56e057f20f883e"}
```

返回：

```
{
  "data": {
    "cookie": "995282b4-82f3-444d-ac6a-50030912e243".....
  },
  "elapsedTime": 167,
  "errorcode": 0,
  "noteMsg": "",
  "success": true
}
```

鉴权验证

后面所有请求，head中增加cookie:sid=995282b4-82f3-444d-ac6a-50030912e243。

5.6 实体查询服务

5.6.1 描述

实体查询API提供客户端通过I+接口，查询关系网络符合条件的实体节点，接口输入为查询类型及查询条件，比如通过姓名户籍查询身份证信息，通过车牌颜色和车牌号查询车辆信息等。

5.6.2 请求参数

名称	类型	是否必须	描述
objectType	String	是	实体节点的类型，跟I+后台配置的实体类型匹配，比如O0001为手机号。
propertyList	Array< PropertyFilter>	是	实体过滤条件。
propertyId	String	否	实体过滤属性的类型。
propertyType	String	否	实体过滤属性的类别。
values	Array<String>	否	实体过滤属性的值。

5.6.3 返回参数

名称	类型	描述
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。

5.6.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	

错误代码	描述	Http状态码	语义
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.6.5 示例

请求示例

```
{
   "objectType":"O0001",
   "propertyList":[
      {
         "propertyId":"O0001P001",
         "propertyType":"string_equal"
         "values":["321321321"]
      }
   ]
     }
```

返回示例

```
{
  "data": {
    "nodes": {
      "O0003P0004-893751893750806906": {
        "id": "O0003P0004-893751893750806906",
        "label": "张三",
        "type": "O0003",
        "virtual": false
      }
    },
    "nodesProps": {
     "O0003P0004-893751893750806906": {
      "O0003P0001":"张三",
      "O0003P0002":"男",
      "O0003P0003":"浙江省",
      "O0003P0004":"893751893750806906"
     }
    }
  },
  "elapsedTime": 50,
```

```
  "noteMsg": "",
  "success": true
      }
```

5.7 关系查询服务

5.7.1 描述

关系查询API提供客户端通过I+接口，查询关系网络符合条件的关系信息，接口输入为查询类型及查询条件，比如通过乘车时间和出发站等信息查询一次人与火车的关系等。

5.7.2 请求参数

名称	类型	是否必须	描述
linkType	String	是	关系节点的类型，跟I+后台配置的关系类型匹配，比如L0001为通话号。
propertyList	Array< PropertyFilter>	是	
propertyId	String	否	关系过滤属性的类型。
propertyType	String	否	关系过滤属性的类别。
values	Array<String>	否	关系过滤属性的值。

5.7.3 返回参数

名称	类型	描述
links	Array<Link>	关系边列表。
id	String	关系边ID。
source	String	关系边的源实体ID。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体ID。
targetType	String	关系边的目标实体类型，如O0004。
linkDetails	Array<String>	关系边包含的边明细记录id列表。

名称	类型	描述
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的ID。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.7.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	

错误代码	描述	Http状态码	语义
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.7.5 示例

请求示例

```
{
    "linkType":"L0001",
    "propertyList":[
        {
            "propertyId":"L0001P001",
            "propertyType":"string_equal",
            "values":[""]
        }
    ]
}
```

返回示例

```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
     } ,
     "L0003^49052e268251cf1b8252c407961e132a": {
      "L0003P0001":"",
      "L0003P0002":"",
     }
    },
    "links": [
```

```
      {
        "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
        ],
        "source": "O0003P0004-893751893750806906",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      },
      {
        "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^49052e268251cf1b8252c407961e132a"
        ],
        "source": "O0003P0004-371411371416495795",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      }
    ],
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```


5.8 默认关联反查服务

5.8.1 描述

默认关联反查API提供客户端可以通过I+接口查询，从一个实体节点出发，查询默认关系的关系网络路径。

- 默认关系通过I+后台配置，支持关系类型配置，关系过滤属性配置，目标节点属性。
- 如果默认关系没有配置，默认查询实体类型可以查询的所有一度直接关系。

5.8.2 请求参数

名称	类型	是否必须	描述
objectType	String	是	实体节点的类型，跟I+后台配置的实体类型匹配，比如O0001为手机号。
objectValue	String	是	实体的主键属性类型和主键值，主键属性类型与I+后台配置的实体主键类型一致，主键和主键值以-连接，比

名称	类型	是否必须	描述
			如：身份证-身份证号码，O0003P0004-371411371416495795。
degree	int	是	查询度数，支持1-5度，建议不超过5度。

5.8.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。
linkCnt	Integer	网络中关系边个数。
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。
targetType	String	关系边的目标实体类型，如O0004。

名称	类型	描述
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.8.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	

错误代码	描述	Http状态码	语义
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.8.5 示例

请求示例

```
{
  "workspaceId":"",
  "objectType":"O0001",
  "objectValue":"O0001P001",
  "degree":3,
  "defaultTimeCond":{
    "propertyId":"",
    "propertyType":"time",
    "values":[null,null]
    }
}
```

返回示例

```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
```

```
  } ,
  "L0003^49052e268251cf1b8252c407961e132a": {
   "L0003P0001":"",
   "L0003P0002":"",
  }
 },
 "links": [
  {
   "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
   "linkDetails": [
    "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
   ],
   "source": "O0003P0004-893751893750806906",
   "sourceType": "O0003",
   "target": "O0004P0002-HB1163",
   "targetType": "O0004"
  },
  {
   "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
   "linkDetails": [
    "L0003^49052e268251cf1b8252c407961e132a"
   ],
   "source": "O0003P0004-371411371416495795",
   "sourceType": "O0003",
   "target": "O0004P0002-HB1163",
   "targetType": "O0004"
  }
 ],
 "nodeCnt": 3,
 "nodes": {
  "O0003P0004-371411371416495795": {
   "id": "O0003P0004-371411371416495795",
   "label": "李四",
   "type": "O0003",
   "virtual": false
  },
  "O0003P0004-893751893750806906": {
   "id": "O0003P0004-893751893750806906",
   "label": "张三",
   "type": "O0003",
   "virtual": false
  },
  "O0004P0002-HB1163": {
   "id": "O0004P0002-HB1163",
   "label": "HB1163",
   "type": "O0004",
   "virtual": false
  }
 },
 "nodesProps": {
  "O0003P0004-371411371416495795": {
   "O0003P0001":"李四",
   "O0003P0002":"",
   "O0003P0003":"",
   "O0003P0004":"371411371416495795"
  },
  "O0003P0004-893751893750806906": {
   "O0003P0001":"张三",
   "O0003P0002":"",
   "O0003P0003":"",
   "O0003P0004":"893751893750806906"
  },
```

```
      "O0004P0002-HB1163": {
       "O0003P0001":"",
       "O0003P0002":"HB1163",
       "O0003P0003":""
      }
     }
   },
   "elapsedTime": 0,
   "noteMsg": "",
   "success": true
 }
```

5.9 关联反查服务

5.9.1 描述

关联反查API提供客户端可以通过I+接口查询，从一个或多个实体节点出发，查询指定关系的关系网络路径。

- 支持多个不同类型的节点出发。
- 支持查询多个关系。
- 支持定义关系属性的过滤条件。
- 支持定义目标节点属性的过滤条件。

5.9.2 请求参数

名称	类型	是否必须	描述
objects	String,Array<String>	是	起始节点列表,KV结构，K为实体类型，和I+后台配置相同，V为实体ID数组，参见请求示例。
links	Array<Link>	是	图查询的需要关系。
linkType	String	是	关系类型和I+后台配置相同。
direct	int	否	关系出入度，0:出度，1:入度，2:出入度，3:无向，不传默认无向/出入度（支持关联反查，群集分析，共同邻居）。

名称	类型	是否必须	描述
queryProps	Array<String>	否	指定想要查询的属性，默认查询所有属性。
propertyList	Array<PropertyFilter>	否	需要过滤关系的属性，详见示例。
propertyId	String	否	关系过滤属性的类型。
propertyType	String	否	关系过滤属性的查询类型。
values	Array<String>	否	详见其他说明。
derivated	Array<PropertyFilter>	否	同字类关系的过滤属性，详见示例，同类为后台管理配置的间接关系，不输入使用后面默认配置。
statPropList	Array<statPropFilter>	否	累计属性条件过滤，用于数值类型sum,count值过滤，属性后台配置，支持关联反查，群集分析，共同邻居。
propertyId	string	否	固定值，一般为linkType+T+基础属性ID(次数为P000)，例：L0001TP000。
propertyType	string	否	固定为number。
values	List<String>	否	同基础属性。
statType	int	否	统计条件类型，0按属性值/次数查询，1 按属性值/次数的排序号查询。
targetObjects	Array<ObjectFilter>	否	目标节点的属性过滤，详见示例。
objectType	String	是	节点类型和I+后台配置相同。

名称	类型	是否必须	描述
propertyList	Array<PropertyFilter>	否	需要目标节点的属性，详见示例。
propertyId	String	否	目标节点过滤属性的类型。
propertyType	String	否	目标节点过滤属性的类别。
values	Array<String>	否	目标节点过滤属性的值。

5.9.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。
linkCnt	Integer	网络中关系边个数。
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。

名称	类型	描述
targetType	String	关系边的目标实体类型，如O0004。
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.9.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	

错误代码	描述	Http状态码	语义
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.9.5 示例

请求示例

```
{
  "data": {
    "objects": {
      "O0003": [
        "O0003P0004-371411371416495795",
        "O0003P0004-893751893750806906"
      ],
      "O0004":[
       "O0004P0002-HB1163",
       "O0004P0002-HB1985",
       "O0004P0002-HB4999"
      ]
    },
    "links": [
      {
        "linkType": "L0003",
        "propertyList": [],
        "derivated": []
      },
      {
        "linkType": "L0004",
        "propertyList": [
          {
            "propertyId": "L0004CC002",
            "propertyType": "string_like",
            "values": [
              "11"
            ]
          }
        ],
        "derivated": [
          {
            "count": {
              "propertyType": "string_equal",
              "values": [
```

```
            "1"
          ]
        },
        "propIds": [
          "L0004CC002"
        ],
        "intervaltime": []
      }
    ]
  }
],
"targetObjects": [
  {
    "objectType": "O0003",
    "propertyList": [
      {
        "propertyId": "O0003P0002",
        "propertyType": "string_like",
        "values": [
          "22"
        ]
      }
    ]
  }
]
}
}
```

返回示例

```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
     } ,
     "L0003^49052e268251cf1b8252c407961e132a": {
      "L0003P0001":"",
      "L0003P0002":"",
     }
    },
    "links": [
      {
        "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
        ],
```

```
      "source": "O0003P0004-893751893750806906",
      "sourceType": "O0003",
      "target": "O0004P0002-HB1163",
      "targetType": "O0004"
    },
    {
      "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
      "linkDetails": [
        "L0003^49052e268251cf1b8252c407961e132a"
      ],
      "source": "O0003P0004-371411371416495795",
      "sourceType": "O0003",
      "target": "O0004P0002-HB1163",
      "targetType": "O0004"
    }
  ],
  "nodeCnt": 3,
  "nodes": {
    "O0003P0004-371411371416495795": {
      "id": "O0003P0004-371411371416495795",
      "label": "李四",
      "type": "O0003",
      "virtual": false
    },
    "O0003P0004-893751893750806906": {
      "id": "O0003P0004-893751893750806906",
      "label": "张三",
      "type": "O0003",
      "virtual": false
    },
    "O0004P0002-HB1163": {
      "id": "O0004P0002-HB1163",
      "label": "HB1163",
      "type": "O0004",
      "virtual": false
    }
  },
  "nodesProps": {
   "O0003P0004-371411371416495795": {
    "O0003P0001":"李四",
    "O0003P0002":"",
    "O0003P0003":"",
    "O0003P0004":"371411371416495795"
   },
   "O0003P0004-893751893750806906": {
    "O0003P0001":"张三",
    "O0003P0002":"",
    "O0003P0003":"",
    "O0003P0004":"893751893750806906"
   },
   "O0004P0002-HB1163": {
    "O0003P0001":"",
    "O0003P0002":"HB1163",
    "O0003P0003":""
   }
  }
},
"elapsedTime": 0,
"noteMsg": "",
"success": true
```

```
}
```

5.10 群集分析服务

5.10.1 描述

群集分析API提供客户端可以通过I+接口查询，从多个实体节点出发，查询实体之间指定关系的一度关系网络路径。

- 支持多个不同类型的节点出发。
- 支持查询多个关系。
- 支持定义关系的过滤条件。
- 目标节点在输入节点范围内。

5.10.2 请求参数

名称	类型	是否必须	描述
group	Array<group>	否	输入节点分组，用于合并节点的群集分析。
id	String	否	合并节点ID，值唯一就可以。
children	Array<String>	否	一个合并节点ID包含的节点。
objects	String,Array<String>	是	起始节点列表,KV结构，K为实体类型，和I+后台配置相同，V为实体ID数组，参见请求示例。
links	Array<Link>	是	图查询的需要关系。
linkType	String	是	关系类型和I+后台配置相同。
direct	int	否	关系出入度，0:出度，1:入度，2:出入度，3:无向，不传默认无向/出入度（支持关联反查，群集分析，共同邻居）。

名称	类型	是否必须	描述
queryProps	Array<String>	否	指定想要查询的属性，默认查询所有属性。
propertyList	Array<PropertyFilter>	否	需要过滤关系的属性，详见示例。
propertyId	String	否	关系过滤属性的类型。
propertyType	String	否	关系过滤属性的查询类型。
values	Array<String>	否	详见其他说明。
derivated	Array<PropertyFilter>	否	同字类关系的过滤属性，详见示例，同类为后台管理配置的间接关系，不输入使用后面默认配置。
statPropList	Array<statPropFilter>	否	累计属性条件过滤，用于数值类型sum,count值过滤，属性后台配置，支持关联反查，群集分析，共同邻居。
propertyId	string	否	固定值，一般为linkType+T+基础属性ID(次数为P000)，例：L0001TP000。
propertyType	string	否	固定为number。
values	List<String>	否	同基础属性。
statType	int	否	统计条件类型，0按属性值/次数查询，1 按属性值/次数的排序号查询。

5.10.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。

名称	类型	描述
linkCnt	Integer	网络中关系边个数。
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。
targetType	String	关系边的目标实体类型，如O0004。
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属

名称	类型	描述
		性值，比如"L0003P0001":"乘车时间"。

5.10.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.10.5 示例

请求示例

```
{
  "data": {
    "objects": {
      "O0003": [
        "O0003P0004-371411371416495795",
        "O0003P0004-893751893750806906"
      ],
      "O0004":[
       "O0004P0002-HB1163",
       "O0004P0002-HB1985",
       "O0004P0002-HB4999"
      ]
    },
    "links": [
      {
        "linkType": "L0003",
        "propertyList": [],
        "derivated": []
      },
      {
        "linkType": "L0004",
        "propertyList": [
          {
            "propertyId": "L0004CC002",
            "propertyType": "string_like",
            "values": [
              "11"
            ]
          }
        ],
        "derivated": [
          {
            "count": {
              "propertyType": "string_equal",
              "values": [
                "1"
              ]
            },
            "propIds": [
              "L0004CC002"
            ],
            "intervaltime": []
          }
        ]
      }
    ],
    "targetObjects": [
      {
        "objectType": "O0003",
        "propertyList": [
          {
            "propertyId": "O0003P0002",
            "propertyType": "string_like",
            "values": [
              "22"
            ]
          }
```

```
        ]
      }
    ]
  }
}
```

返回示例


```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
     } ,
     "L0003^49052e268251cf1b8252c407961e132a": {
      "L0003P0001":"",
      "L0003P0002":"",
     }
    },
    "links": [
      {
        "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
        ],
        "source": "O0003P0004-893751893750806906",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      },
      {
        "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^49052e268251cf1b8252c407961e132a"
        ],
        "source": "O0003P0004-371411371416495795",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      }
    ],
    "nodeCnt": 3,
    "nodes": {
      "O0003P0004-371411371416495795": {
        "id": "O0003P0004-371411371416495795",
        "label": "李四",
        "type": "O0003",
```

```
        "virtual": false
      },
      "O0003P0004-893751893750806906": {
        "id": "O0003P0004-893751893750806906",
        "label": "张三",
        "type": "O0003",
        "virtual": false
      },
      "O0004P0002-HB1163": {
        "id": "O0004P0002-HB1163",
        "label": "HB1163",
        "type": "O0004",
        "virtual": false
      }
    },
    "nodesProps": {
     "O0003P0004-371411371416495795": {
      "O0003P0001":"李四",
      "O0003P0002":"",
      "O0003P0003":"",
      "O0003P0004":"371411371416495795"
     },
     "O0003P0004-893751893750806906": {
      "O0003P0001":"张三",
      "O0003P0002":"",
      "O0003P0003":"",
      "O0003P0004":"893751893750806906"
     },
     "O0004P0002-HB1163": {
      "O0003P0001":"",
      "O0003P0002":"HB1163",
      "O0003P0003":""
     }
    }
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```


5.11 共同邻居服务

5.11.1 描述

关联反查API提供客户端可以通过I+接口查询，从多个实体节点出发，查询指定关系，与输入节点直接相关节点的关系网络路径。

- 支持多个不同类型的节点出发。
- 支持查询多个关系。
- 支持定义关系的过滤条件。
- 支持定义目标节点的过滤条件。
- 支持配置共同邻居个数的配置。

5.11.2 请求参数

名称	类型	是否必须	描述
objects	String,Array<String>	是	起始节点列表，KV结构，K为实体类型，和I+后台配置相同，V为实体ID数组，参见请求示例。
links	Array<Link>	是	图查询的需要关系。
linkType	String	是	关系类型和I+后台配置相同。
direct	int	否	关系出入度，0:出度，1:入度，2:出入度，3:无向，不传默认无向/出入度（支持关联反查，群集分析，共同邻居）。
queryProps	Array<String>	否	指定想要查询的属性，默认查询所有属性。
propertyList	Array<PropertyFilter>	否	需要过滤关系的属性，详见示例。
propertyId	String	否	关系过滤属性的类型。
propertyType	String	否	关系过滤属性的查询类型。
values	Array<String>	否	详见其他说明。
derivated	Array<PropertyFilter>	否	同字类关系的过滤属性，详见示例，同类为后台管理配置的间接关系，不输入使用后面默认配置。
statPropList	Array<statPropFilter>	否	累计属性条件过滤，用于数值类型sum,count值过滤，属性后台配置，支持关联反查，群集分析，共同邻居。

名称	类型	是否必须	描述
propertyId	string	否	固定值，一般为linkType+T+基础属性ID(次数为P000)，例：L0001TP000。
propertyType	string	否	固定为number。
values	List<String>	否	同基础属性。
statType	int	否	统计条件类型，0按属性值/次数查询，1 按属性值/次数的排序号查询。
targetObjects	Array<ObjectFilter>	否	目标节点的属性过滤，详见示例。
objectType	String	是	节点类型和I+后台配置相同。
propertyList	Array<PropertyFilter>	否	需要目标节点的属性，详见示例。
propertyId	String	否	目标节点过滤属性的类型。
propertyType	String	否	目标节点过滤属性的类别。
values	Array<String>	否	目标节点过滤属性的值。
degree	int	否	用于路径分析，共同邻居，血缘分析。

5.11.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。
linkCnt	Integer	网络中关系边个数。
nodes	Array<Node>	节点列表。
id	String	节点id。

名称	类型	描述
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。
targetType	String	关系边的目标实体类型，如O0004。
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.11.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.11.5 示例

请求示例

```
{
  "data": {
    "degree": "3",
    "objects": {
      "O0003": [
```

```
        "O0003P0004-371411371416495795",
        "O0003P0004-893751893750806906"
      ],
      "O0004":[
       "O0004P0002-HB1163",
       "O0004P0002-HB1985",
       "O0004P0002-HB4999"
      ]
    },
    "links": [
      {
        "linkType": "L0003",
        "propertyList": [],
        "derivated": []
      },
      {
        "linkType": "L0004",
        "propertyList": [
          {
            "propertyId": "L0004CC002",
            "propertyType": "string_like",
            "values": [
              "11"
            ]
          }
        ],
        "derivated": [
          {
            "count": {
              "propertyType": "string_equal",
              "values": [
                "1"
              ]
            },
            "propIds": [
              "L0004CC002"
            ],
            "intervaltime": []
          }
        ]
      }
    ],
    "targetObjects": [
      {
        "objectType": "O0003",
        "propertyList": [
          {
            "propertyId": "O0003P0002",
            "propertyType": "string_like",
            "values": [
              "22"
            ]
          }
        ]
      }
    ]
  }
```

```
}
```

返回示例


```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
     } ,
     "L0003^49052e268251cf1b8252c407961e132a": {
      "L0003P0001":"",
      "L0003P0002":"",
     }
    },
    "links": [
      {
        "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
        ],
        "source": "O0003P0004-893751893750806906",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      },
      {
        "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^49052e268251cf1b8252c407961e132a"
        ],
        "source": "O0003P0004-371411371416495795",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      }
    ],
    "nodeCnt": 3,
    "nodes": {
      "O0003P0004-371411371416495795": {
        "id": "O0003P0004-371411371416495795",
        "label": "李四",
        "type": "O0003",
        "virtual": false
      },
      "O0003P0004-893751893750806906": {
        "id": "O0003P0004-893751893750806906",
```

```
        "label": "张三",
        "type": "O0003",
        "virtual": false
      },
      "O0004P0002-HB1163": {
        "id": "O0004P0002-HB1163",
        "label": "HB1163",
        "type": "O0004",
        "virtual": false
      }
    },
    "nodesProps": {
     "O0003P0004-371411371416495795": {
      "O0003P0001":"李四",
      "O0003P0002":"",
      "O0003P0003":"",
      "O0003P0004":"371411371416495795"
     },
     "O0003P0004-893751893750806906": {
      "O0003P0001":"张三",
      "O0003P0002":"",
      "O0003P0003":"",
      "O0003P0004":"893751893750806906"
     },
     "O0004P0002-HB1163": {
      "O0003P0001":"",
      "O0003P0002":"HB1163",
      "O0003P0003":""
     }
    }
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```


5.12 路径分析服务

5.12.1 描述

关联反查API提供客户端可以通过I+接口查询，查询两个实体之间指定关系的一度或多度关系网络路径。

- 支持多个不同类型的节点出发。
- 支持查询多个关系。
- 支持定义关系的过滤条件。
- 支持定义目标节点的过滤条件。
- 支持设置路径分析的路径查询数量。
- 最多支持6度的关系拓展。
- 路径分析不支持间接关系。

5.12.2 请求参数

名称	类型	是否必须	描述
objects	String,Array<String>	是	起始节点列表,KV结构，K为实体类型，和I+后台配置相同，V为实体ID数组，参见请求示例。
links	Array<Link>	是	图查询的需要关系。
linkType	String	是	关系类型和I+后台配置相同。
direct	int	否	关系出入度，0:出度，1:入度，2:出入度，3:无向，不传默认无向/出入度（支持关联反查，群集分析，共同邻居）。
queryProps	Array<String>	否	指定想要查询的属性，默认查询所有属性。
propertyList	Array<PropertyFilter>	否	需要过滤关系的属性，详见示例。
propertyId	String	否	关系过滤属性的类型。
propertyType	String	否	关系过滤属性的查询类型。
values	Array<String>	否	详见其他说明。
degree	int	否	传值则返回指定度数内的所有路径，不传值默认查询最短路径。

5.12.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。
linkCnt	Integer	网络中关系边个数。

名称	类型	描述
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。
targetType	String	关系边的目标实体类型，如O0004。
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.12.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.12.5 示例

请求示例

```
{
  "data": {
    "type": "remote",
    "degree": "3",
    "objects": {
```

```
      "O0003": [
        "O0003P0004-371411371416495795",
        "O0003P0004-893751893750806906"
      ],
      "O0004":[
       "O0004P0002-HB1163",
       "O0004P0002-HB1985",
       "O0004P0002-HB4999"
      ]
    },
    "links": [
      {
        "linkType": "L0003",
        "propertyList": [],
        "derivated": []
      },
      {
        "linkType": "L0004",
        "propertyList": [
          {
            "propertyId": "L0004CC002",
            "propertyType": "string_like",
            "values": [
              "11"
            ]
          }
        ],
        "derivated": [
          {
            "count": {
              "propertyType": "string_equal",
              "values": [
                "1"
              ]
            },
            "propIds": [
              "L0004CC002"
            ],
            "intervaltime": []
          }
        ]
      }
    ],
    "targetObjects": [
      {
        "objectType": "O0003",
        "propertyList": [
          {
            "propertyId": "O0003P0002",
            "propertyType": "string_like",
            "values": [
              "22"
            ]
          }
        ]
      }
    ]
  }
```

```
}
```

返回示例


```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
     } ,
     "L0003^49052e268251cf1b8252c407961e132a": {
      "L0003P0001":"",
      "L0003P0002":"",
     }
    },
    "links": [
      {
        "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
        ],
        "source": "O0003P0004-893751893750806906",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      },
      {
        "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
        "linkDetails": [
          "L0003^49052e268251cf1b8252c407961e132a"
        ],
        "source": "O0003P0004-371411371416495795",
        "sourceType": "O0003",
        "target": "O0004P0002-HB1163",
        "targetType": "O0004"
      }
    ],
    "nodeCnt": 3,
    "nodes": {
      "O0003P0004-371411371416495795": {
        "id": "O0003P0004-371411371416495795",
        "label": "李四",
        "type": "O0003",
        "virtual": false
      },
      "O0003P0004-893751893750806906": {
        "id": "O0003P0004-893751893750806906",
```

```
        "label": "张三",
        "type": "O0003",
        "virtual": false
      },
      "O0004P0002-HB1163": {
        "id": "O0004P0002-HB1163",
        "label": "HB1163",
        "type": "O0004",
        "virtual": false
      }
    },
    "nodesProps": {
     "O0003P0004-371411371416495795": {
      "O0003P0001":"李四",
      "O0003P0002":"",
      "O0003P0003":"",
      "O0003P0004":"371411371416495795"
     },
     "O0003P0004-893751893750806906": {
      "O0003P0001":"张三",
      "O0003P0002":"",
      "O0003P0003":"",
      "O0003P0004":"893751893750806906"
     },
     "O0004P0002-HB1163": {
      "O0003P0001":"",
      "O0003P0002":"HB1163",
      "O0003P0003":""
     }
    }
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```


5.13 血缘分析服务

5.13.1 描述

血缘分析API提供客户端可以通过I+接口查询，特定业务关系的多度扩展。

- 支持单个实体。
- 特定关系是指在后台管理-业务参数-血缘关系配置的关系，一般如果血缘，同户号等。
- 不支持条件过滤。
- 支持设置查询度数，不超过5度。

5.13.2 请求参数

名称	类型	是否必须	描述
objects	String,Array<String>	是	起始节点列表,KV结构，K为实体类型，和

名称	类型	是否必须	描述
			I+后台配置相同，V为实体ID数组，参见请求示例。
degree	int	否	传值则返回指定度数内的所有路径，不传值默认查询最短路径。

5.13.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。
linkCnt	Integer	网络中关系边个数。
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。
targetType	String	关系边的目标实体类型，如O0004。

名称	类型	描述
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.13.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	

错误代码	描述	Http状态码	语义
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.13.5 示例

请求示例

```
{
    "degree": "3",
    "objects": {
      "O0003": [
        "O0003P0004-371411371416495795"
      ]
    }
}
```

返回示例

```
{
  "data": {
    "linkCnt": 2,
    "linkDetails": {
      "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "乘火车",
        "linkId": "L0003^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0003",
      },
      "L0003^49052e268251cf1b8252c407961e132a": {
       "label": "乘火车",
        "linkId": "L0003^49052e268251cf1b8252c407961e132a",
        "linkType": "L0003"
      }
    },
    "linkProps": {
     "L0003^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0003P0001":"",
      "L0003P0002":"",
     } ,
     "L0003^49052e268251cf1b8252c407961e132a": {
      "L0003P0001":"",
```

```
    "L0003P0002":"",
   }
  },
  "links": [
    {
      "id": "O0003#O0003P0004-893751893750806906^O0004#O0004P0002-HB1163",
      "linkDetails": [
        "L0003^27b44b00fcf663b8fdfad968f8a2eba9"
      ],
      "source": "O0003P0004-893751893750806906",
      "sourceType": "O0003",
      "target": "O0004P0002-HB1163",
      "targetType": "O0004"
    },
    {
      "id": "O0003#O0003P0004-371411371416495795^O0004#O0004P0002-HB1163",
      "linkDetails": [
        "L0003^49052e268251cf1b8252c407961e132a"
      ],
      "source": "O0003P0004-371411371416495795",
      "sourceType": "O0003",
      "target": "O0004P0002-HB1163",
      "targetType": "O0004"
    }
  ],
  "nodeCnt": 3,
  "nodes": {
    "O0003P0004-371411371416495795": {
      "id": "O0003P0004-371411371416495795",
      "label": "李四",
      "type": "O0003",
      "virtual": false
    },
    "O0003P0004-893751893750806906": {
      "id": "O0003P0004-893751893750806906",
      "label": "张三",
      "type": "O0003",
      "virtual": false
    },
    "O0004P0002-HB1163": {
      "id": "O0004P0002-HB1163",
      "label": "HB1163",
      "type": "O0004",
      "virtual": false
    }
  },
  "nodesProps": {
   "O0003P0004-371411371416495795": {
    "O0003P0001":"李四",
    "O0003P0002":"",
    "O0003P0003":"",
    "O0003P0004":"371411371416495795"
   },
   "O0003P0004-893751893750806906": {
    "O0003P0001":"张三",
    "O0003P0002":"",
    "O0003P0003":"",
    "O0003P0004":"893751893750806906"
   },
   "O0004P0002-HB1163": {
    "O0003P0001":"",
    "O0003P0002":"HB1163",
```

```
      "O0003P0003":""
     }
    }
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```

5.14 虚拟节点持久化服务

5.14.1 描述

虚拟节点持久化API提供客户端持久化一个系统数据库中不存在的点到数据库。

持久化系统不存在的用户添加的节点信息。

5.14.2 请求参数

名称	类型	是否必须	描述
objectType	String	是	实体类型，和I+后台配置相同。
objectValue	String	是	节点值，属性标识与主键属性值的组合。
properties	Array<Properties>	是	实体需要添加的属性。
propId	string	是	实体属性ID，与后台管理配置对应。
propValue	string	是	实体属性对应的值。

5.14.3 返回参数

名称	类型	描述
nodeCnt	Integer	网络中实体节点个数。
nodes	Array<Node>	节点列表。
id	String	节点id。
label	String	节点标签，跟I+后台配置的实体属性一致。
type	String	节点类型，如O0001。
virtual	Boolean	节点在网络中是否存在。

名称	类型	描述
nodesProps	Array<Property>	节点属性列表。
<Property>	<String,String>	节点属性KV值，K为节点类型，和I+后台配置的实体类型一致，V为节点属性，比如"O0003P0001":"张三"。

5.14.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信	200	

错误代码	描述	Http状态码	语义
	息在noteMsg中详细解释。		

5.14.5 示例

请求示例

```
{
   "objectType":"O0001",
   "objectValue":"O0001P0001-1323432543",
   "properties":[
      {
         "propId":"O0001P001",
         "propValue":"1323432543"
      },
      {
         "propId":"O0001P002",
         "propValue":"陈大"
      },
      {
         "propId":"O0001P003",
         "propValue":"C类"
      },
      {
         "propId":"O0001P006",
         "propValue":"中国联通"
      },
      {
         "propId":"O0001P007",
         "propValue":"1340005435"
      }
   ]
}
```

返回示例

```
{
  "data": {
    "nodeCnt": 1,
    "nodes": {
      " O0001P0001-1323432543": {
        "id": " O0001P0001-1323432543",
        "label": "陈大",
        "avatar": "/static/img/id.png",
        "type": "O0001",
        "virtual": false
      }
    },
    "nodesProps": {
     " O0001P0001-1323432543": {
      " O0001P0001":"1323432543",
      " O0001P002":"陈大",
      " O0001P003":"C类",
      " O0001P006":"中国联通" ,
          " O0001P007":"2017-01-01 00:00:01"
```

```
      }
    }
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```

5.15 添加关系持久化服务

5.15.1 描述

添加关系持久化API提供用户持久化两点之间的关系数据到数据库。

持久化系统不存在的，用户自己的关系数据。

5.15.2 请求参数

名称	类型	是否必须	描述
linkType	String,Array<String>	是	起始节点列表，KV结构，K为实体类型，和I+后台配置相同，V为实体ID数组，参见请求示例。
source	Object	是	关系的其中一个实体。
objectType	String	是	实体类型。
objectValue	int	否	实体值。
target	Object	是	关系的另外一个实体。
objectType	String	否	实体类型。
objectValue	String	否	实体值。
properties	Array<Properties>	是	实体需要添加的属性。
propId	string	是	实体属性ID，与后台管理配置对应。
propValue	string	是	实体属性对应的值。

5.15.3 返回参数

名称	类型	描述
linkCnt	Integer	网络中关系边个数。

名称	类型	描述
links	Array<Link>	关系边列表。
id	String	关系边id。
source	String	关系边的源实体id。
sourceType	String	关系边的源实体类型，如O0003。
target	String	关系边的目标实体id。
targetType	String	关系边的目标实体类型，如O0004。
linkDetails	Array<String>	关系边包含的边明细记录id列表。
linkDetails	Array<LinkDetail>	关系边明细列表。
label	String	关系边明细的label。
linkId	String	关系边明细的id。
linkType	String	关系边明细类型。
linkProps	Array<Property>	关系边明细属性列表。
<Property>	<String,String>	关系属性KV值，K为关系边属性类型，和I+后台配置的关系边属性类型一致，V为关系边属性值，比如"L0003P0001":"乘车时间"。

5.15.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信	200	

错误代码	描述	Http状态码	语义
	息在noteMsg中详细解释。		
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.15.5 示例

请求示例

```
{
    "linkType":"L00001",
    "source":{
        "objectType":"O0001",
        "objectValue":"O0001P001-13100000003"
    },
    "target":{
        "objectType":"O0001",
        "objectValue":"O0001P001-13100000004"
    },
    "properties":[
        {
            "propId":"L00001P013",
            "propValue":"13100000003"
        },
        {
            "propId":"L00001P015",
            "propValue":"13100000004"
        },
        {
            "propId":"L00001P012",
```

```
            "propValue":"14204320000"
        },
        {
            "propId":"L00001P022",
            "propValue":"1"
        },
        {
            "propId":"L00001P023",
            "propValue":"20"
        }
    ]
}
```


返回示例


```
{
  "data": {
    "linkCnt": 1,
    "linkDetails": {
      "L00001^27b44b00fcf663b8fdfad968f8a2eba9": {
        "label": "通话",
        "linkId": "L0001^27b44b00fcf663b8fdfad968f8a2eba9",
        "linkType": "L0001",
        "d": "1",
      }
    },
    "linkProps": {
     "L0001^27b44b00fcf663b8fdfad968f8a2eba9": {
      "L0001P0001":"",
      "L0001P0002":"",
     }
    },
    "links": [
      {
        "id": "O0001#O0001P0001-13100000003^O0001#O0001P0001-13100000004",
        "linkDetails": [
          "L0001^27b44b00fcf663b8fdfad968f8a2eba9"
        ],
        "source": "O0001P0001-13100000003",
        "sourceType": "O0001",
        "target": "O0001P0001-13100000004",
        "targetType": "O0001"
      }
    ]
    }
  },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
```

```
}
```

5.16 查看olpmeta配置映射

5.16.1 描述

查看I+ OLP模型的配置信息，用于其他系统用户配置业务变量映射。

5.16.2 请求参数

无。

5.16.3 返回参数

名称	类型	描述
objects	Array<ObjectDef>	所有实体定义。
objDefId	String	实体定义ID，如：O0001。
objName	String	实体名称。
bizVar	String	业务变量。
objProps	Array<propertyDef>	属性列表。
propDefId	String	属性定义ID，如：O0001P001。
propName	String	属性中文名。
bizVar	String	属性业务变量。
links	Array<LinkDef>	所有关系定义。
linkDefId	String	关系定义ID，如：L0001。
linkName	String	实体名称。
bizVar	String	业务变量。
linkProps	Array<propertyDef>	属性列表。
propDefId	String	属性定义ID，如：L0001P001。
propName	String	属性中文名。
bizVar	String	属性业务变量。

5.16.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.16.5 示例

返回示例

```
{
        "data": {
    "objects":[
        {
            "objDefId":"O0001",
```

```
            "objName":"手机",
            "bizVar":"",
            "objProps":[
                {
                    "propDefId":"O0001P001",
                    "propName":"手机号",
                    "bizVar":""
                },{
                    "propDefId":"O0001P002",
                    "propName":"运营商",
                    "bizVar":""
                }
            ]
        },

    ],
    "links":[
        {
            "linkDefId":"L00001",
            "linkName":"通话",
            "bizVar":"tell",
            "linkProps":[
                {
                    "propDefId":"L0001P001",
                    "propName":"手机号",
                    "bizVar":"phone"
                }
            ]
        }
    ]
          },
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```


5.17 更新olpmeta配置映射

5.17.1 描述

更新I+ OLP模型的配置信息，用于其他系统用户配置业务变量映射。

5.17.2 请求参数

名称	类型	描述
objects	Array<ObjectDef>	所有实体定义。
objDefId	String	实体定义ID，如：O0001。
objName	String	实体名称。
bizVar	String	业务变量。
objProps	Array<propertyDef>	属性列表。

名称	类型	描述
propDefId	String	属性定义ID，如：O0001P001。
propName	String	属性中文名。
bizVar	String	属性业务变量。
links	Array<LinkDef>	所有关系定义。
linkDefId	String	关系定义ID，如：L0001。
linkName	String	实体名称。
bizVar	String	业务变量。
linkProps	Array<propertyDef>	属性列表
propDefId	String	属性定义ID，如：L0001P001。
propName	String	属性中文名。
bizVar	String	属性业务变量。

5.17.3 返回参数

无。

5.17.4 错误码

错误代码	描述	Http状态码	语义
00XXXX	00开头的异常为系统通用异常，错误信息在noteMsg中详细解释。	200	
10XXXX	10开头的异常为鉴权认证异常，错误信息在noteMsg中详细解释。	200	
20XXXX	20开头的异常为用户及角色管理异常，错误信息在noteMsg中详细解释。	200	
30XXXX	30开头的异常为数据源配置异常，错误信息在noteMsg中详细解释。	200	

错误代码	描述	Http状态码	语义
40XXXX	40开头的异常为meta配置异常，错误信息在noteMsg中详细解释。	200	
50XXXX	50开头的异常为缓存刷新异常，错误信息在noteMsg中详细解释。	200	
60XXXX	60开头的异常为系统参数配置异常，错误信息在noteMsg中详细解释。	200	
70XXXX	70开头的异常为业务参数配置异常，错误信息在noteMsg中详细解释。	200	

5.17.5 示例

请求示例

```
{
    "objects":[
        {
            "objDefId":"O0001",
            "objName":"手机",
            "bizVar":"",
            "objProps":[
                {
                    "propDefId":"O0001P001",
                    "propName":"手机号",
                    "bizVar":""
                },{
                    "propDefId":"O0001P002",
                    "propName":"运营商",
                    "bizVar":""
                }
            ]
        },

    ],
    "links":[
        {
            "linkDefId":"L00001",
            "linkName":"通话",
            "bizVar":"tell",
            "linkProps":[
                {
                    "propDefId":"L0001P001",
                    "propName":"手机号",
```

```
                "bizVar":"phone"
            }
        ]
    }
  ]
}
```

返回示例

```
{
        "data": "OK"
  "elapsedTime": 0,
  "noteMsg": "",
  "success": true
}
```