对象存储OSS

SDK手册

阿里云
aliyun.com

SDK手册

Java-SDK

前言

前言

SDK下载

Java SDK开发包(2015-11-24) 版本号 2.0.7：java_sdk_20151124.zip

版本迭代详情参考这里

简介

本文档主要介绍OSS Java SDK的安装和使用，针对于JAVA SDK版本2.0.7。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。

安装

安装

直接在Eclipse中使用JAR包

步骤如下：

- 在官方网站下载 Open Service Java SDK 。
- 解压文件。
- 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你的工程文件夹中。
- 在Eclipse右键工程 -> Properties -> Java Build Path -> Add JARs 。
- 选择你拷贝的所有JAR文件。
- 经过上面几步之后，你就可以在工程中使用OSS JAVA SDK了。

在Maven工程中使用SDK

在Maven工程中使用JAVA SDK十分简单，只要在在pom.xml文件中加入依赖就可以了。以2.0.6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>2.0.6</version>
</dependency>
```

快速入门

快速入门

在这一章里，您将学到如何用OSS Java 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:

```
import com.aliyun.oss.OSSClient;

public class Sample {

    public static void main(String[] args) {
        String accessKeyId = "<key>";
        String accessKeySecret = "<secret>";
        // 以杭州为例
        String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";

        // 初始化一个OSSClient
        OSSClient client = new OSSClient(endpoint,accessKeyId, accessKeySecret);

        // 下面是一些调用代码...
        ...
    }
}
```

在上面代码中，变量 accessKeyId 与 accessKeySecret 是由系统分配给用户的，称为ID对，用于标识用户，可能属于您的阿里云账号或者RAM账号为访问OSS做签名验证。关于OSSClient的详细介绍，参见 OSSClient。

Step-2. 新建bucket

Bucket是OSS全局命名空间，相当于数据的容器，可以存储若干Object。您可以按照下面的代码新建一个Bucket：

```
public void createBucket(String bucketName) {

    // 初始化OSSClient
    OSSClient client = new OSSClient(endpoint,accessKeyId, accessKeySecret);

    // 新建一个Bucket
    client.createBucket(bucketName);
}
```

关于Bucket的命名规范，参见Bucket中的命名规范。

Step-3. 上传Object

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一个Object的上传：

```
public void putObject(String bucketName, String key, String filePath) throws FileNotFoundException {

    // 初始化OSSClient
    OSSClient client = new OSSClient(endpoint,accessKeyId, accessKeySecret);

    // 获取指定文件的输入流
    File file = new File(filePath);
    InputStream content = new FileInputStream(file);

    // 创建上传Object的Metadata
    ObjectMetadata meta = new ObjectMetadata();

    // 必须设置ContentLength
    meta.setContentLength(file.length());

    // 上传Object.
    PutObjectResult result = client.putObject(bucketName, key, content, meta);

    // 打印ETag
    System.out.println(result.getETag());
}
```

Object通过InputStream的形式上传到OSS中。在上面的例子里我们可以看出，每个上传的Object，都需要指定和此Object关联的ObjectMetadata。ObjectMetaData是用户对该object的描述，由一系列name-value对组成；其中ContentLength是必须设置的，以便SDK可以正确识别上传Object的大小。为了保证上传文件服务器端与本地一致，用户可以设置ContentMD5，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。关于Object的命名规范，参见Object中的命名规范 。关于上传Object更详细的信息，参见Object中的上传Object 。

Step-4. 列出所有Object

当您完成一系列上传后，可能需要查看某个Bucket中有哪些Object，可以通过下面的程序实现：

```
public void listObjects(String bucketName) {

    // 初始化OSSClient
    OSSClient client = new OSSClient(endpoint,accessKeyId, accessKeySecret);

    // 获取指定bucket下的所有Object信息
    ObjectListing listing = client.listObjects(bucketName);

    // 遍历所有Object
    for (OSSObjectSummary objectSummary : listing.getObjectSummaries()) {
        System.out.println(objectSummary.getKey());
    }
}
```

listObjects方法会返回ObjectListing对象，ObjectListing对象包含了此次listObject请求的返回结果。其中我们可以通过ObjetListing中的getObjectSummaries方法获取所有Object的描述信息。

Step-5. 获取指定Object

您可以参考下面的代码简单地实现一个Object的获取：

```
public void getObject(String bucketName, String key) throws IOException {

    // 初始化OSSClient
    OSSClient client = new OSSClient(endpoint,accessKeyId, accessKeySecret);
    // 获取Object，返回结果为OSSObject对象
    OSSObject object = client.getObject(bucketName, key);
    // 获取Object的输入流
    InputStream objectContent = object.getObjectContent();
    // 处理Object
    ...
    // 关闭流
    objectContent.close();
}
```

当调用OSSClient的getObject方法时，会返回一个OSSObject的对象，此对象包含了Object的各种信息。通过OSSObject的getObjectContent方法，可以获取返回的Object的输入流，通过读取此输入流获取此Object的内容，在用完之后关闭这个流。

OSSClient

OSSClient

OSSClient是OSS服务的Java客户端，它为调用者提供了一系列的方法，用于和OSS服务进行交互。

新建OSSClient

新建一个OSSClient很简单，如下面代码所示：

```
String key = "<key>";
String secret = "<secret>";
// 以杭州为例
String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";

OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret);
```

关键需要传入AccessKey以及需要访问bucket的endpoint。如果想使用HTTPS协议，endpoint以https://开头即可。

配置OSSClient

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。

使用代理

下面的代码让客户端使用代理访问OSS服务：

```
// 创建ClientConfiguration实例
ClientConfiguration conf = new ClientConfiguration();

// 配置代理为本地8080端口
conf.setProxyHost("127.0.0.1");
conf.setProxyPort(8080);

// 创建OSS客户端
client = new OSSClient(endpoint, accessKeySecret, accessKeySecret, conf);
```

上面代码使得客户端的所有操作都会使用127.0.0.1地址的8080端口做代理执行。对于有用户验证的代理，可以配置用户名和密码：

```
// 创建ClientConfiguration实例
ClientConfiguration conf = new ClientConfiguration();

// 配置代理为本地8080端口
conf.setProxyHost("127.0.0.1");
conf.setProxyPort(8080);

//设置用户名和密码
conf.setProxyUsername("username");
conf.setProxyPassword("password");
```

设置网络参数

我们可以用ClientConfiguration设置一些网络参数：

```
ClientConfiguration conf = new ClientConfiguration();

// 设置HTTP最大连接数为10
conf.setMaxConnections(10);

// 设置TCP连接超时为5000毫秒
conf.setConnectionTimeout(5000);

// 设置最大的重试次数为3
conf.setMaxErrorRetry(3);

// 设置Socket传输数据超时的时间为2000毫秒
conf.setSocketTimeout(2000);
```

通过ClientConfiguration可以设置的参数有：

参数	描述
UserAgent	用户代理，指HTTP的User-Agent头。默认为”aliyun-sdk-java”
ProxyHost	代理服务器主机地址
ProxyPort	代理服务器端口
ProxyUsername	代理服务器验证的用户名

ProxyPassword	代理服务器验证的密码
ProxyDomain	访问NTLM验证的代理服务器的Windows域名
ProxyWorkstation	NTLM代理服务器的Windows工作站名称
MaxConnections	允许打开的最大HTTP连接数。默认为1024
SocketTimeout	打开连接传输数据的超时时间（单位：毫秒）。默认为50000毫秒
ConnectionTimeout	建立连接的超时时间（单位：毫秒）。默认为50000毫秒
MaxErrorRetry	可重试的请求失败后最大的重试次数。默认为3次

Bucket

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。一个应用，例如图片分享网站，可以对应一个或多个Bucket。一个用户最多可创建10个Bucket，但每个bucket中存放的object的数量没有限制，存储容量每个buckte最高支持2PB。

命名规范

Bucket的命名有以下规范：

- 只能包括小写字母，数字，短横线（-）
- 必须以小写字母或者数字开头
- 长度必须在3-63字节之间

新建Bucket

如下代码可以新建一个Bucket：

```
String bucketName = "my-bucket-name";

// 初始化OSSClient
OSSClient client = ...;

// 新建一个Bucket
client.createBucket(bucketName);
```

由于Bucket的名字是全局唯一的，所以尽量保证您的 bucketName 不与别人重复。

列出用户所有的Bucket

下面代码可以列出用户所有的Bucket：

```
// 获取用户的Bucket列表
List<Bucket> buckets = client.listBuckets();
```

```
// 遍历Bucket
for (Bucket bucket : buckets) {
    System.out.println(bucket.getName());
}
```

使用CNAME进行访问

当用户将自己的域名CNAME指向自己的一个bucket的域名后，用户可以使用自己的域名来访问OSS：

```
// 比如你的域名"http://cname.com"CNAME指向你的bucket域名"mybucket.oss-cn-hangzhou.aliyuncs.com"
OSSClient client = new OSSClient("http://cname.com/", /* accessKeyId */, /* accessKeySecret */);

PutObjectResult result = client.putObject("mybucket", /* key */, /* input */, /* metadata */);
```

用户只需要在创建OSSClinet类实例时，将原本填入该bucket的endpoint更换成CNAME后的域名即可。同时需要注意的是，使用该OSSClinet实例的后续操作中，bucket的名称只能填成被指向的bucket名称。

> 如果是专有云用户，希望使用非aliyuncs.com结尾的域名访问OSS。可以通过ClientConfiguration中setCnameExcludeList来设置endpoint来避免使用cname方式访问OSS。

判断Bucket是否存在

判断Bucket是否存在可以使用以下代码：

```
String bucketName = "your-bucket-name";

// 获取Bucket的存在信息
boolean exists = client.doesBucketExist(bucketName);

// 输出结果
if (exists) {
    System.out.println("Bucket exists");
} else {
    System.out.println("Bucket not exists");
}
```

设置Bucket ACL

设置Bucket ACL可以使用以下代码：

```
String bucketName = "your-bucket-name";

//以设置为私有权限举例
client.setBucketAcl(bucketName,CannedAccessControlList.Private);
```

获取Bucket ACL

获取Bucket ACL可以使用以下代码：

```
String bucketName = "your-bucket-name";
AccessControlList accessControlList = client.getBucketAcl(bucketName);
```

```
//可以打印出来看结果,也可以从控制台确认
System.out.println(accessControlList.toString());
```

获取Bucket地址

获取Bucket地址可以使用以下代码：

```
String bucketName = "your-bucket-name";

// 获取 bucket 地址
String location = client.getBucketLocation(bucketName);
System.out.println(location);
```

删除Bucket

下面代码删除了一个Bucket

```
String bucketName = "your-bucket-name";

// 删除Bucket
client.deleteBucket(bucketName)
```

需要注意的是，如果Bucket不为空（Bucket中有Object），则Bucket无法删除，必须删除Bucket中的所有Object后，Bucket才能成功删除。

Object

Object

在OSS中，用户操作的基本数据单元是Object。单个Object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object大小不能超过48.8TB。Object包含key、meta和data。其中，key是Object的名字；meta是用户对该object的描述，由一系列name-value对组成；data是Object的数据。

命名规范

Object的命名规范如下：

- 使用UTF-8编码
- 长度必须在1-1023字节之间
- 不能以“/”或者“\”字符开头
- 不能含有“\r”或者“\n”的换行符

上传Object

最简单的上传

代码如下：

```
public void putObject(String bucketName, String key, String filePath) throws FileNotFoundException {
    // 初始化OSSClient
    OSSClient client = ...;
    // 获取指定文件的输入流
    File file = new File(filePath);
    InputStream content = new FileInputStream(file);
    // 创建上传Object的Metadata
    ObjectMetadata meta = new ObjectMetadata();
    // 必须设置ContentLength
    meta.setContentLength(file.length());

    // 上传Object.
    PutObjectResult result = client.putObject(bucketName, key, content, meta);

    // 打印ETag
    System.out.println(result.getETag());
}
```

Object通过InputStream的形式上传到OSS中。在上面的例子里我们可以看出，每上传一个Object，都需要指定和Object关联的ObjectMetadata。ObjectMetaData是用户对该object的描述，由一系列name-value对组成；其中ContentLength是必须设置的，以便SDK可以正确识别上传Object的大小。 为了保证上传文件服务器端与本地一致，用户可以设置ContentMD5，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。

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Object来存储。但给用户提供了创建模拟文件夹的方式，如下代码：

```
String bucketName = "your-bucket-name";
//要创建的文件夹名称,在满足Object命名规则的情况下以"/"结尾
String objectName = "folder_name/";
OSSClient client = new OSSClient(OSS_ENDPOINT, ACCESS_ID, ACCESS_KEY);
ObjectMetadata objectMeta = new ObjectMetadata();
/*这里的size为0,注意OSS本身没有文件夹的概念,这里创建的文件夹本质上是一个size为0的Object,dataStream仍然可以有数据
 */
byte[] buffer = new byte[0];
ByteArrayInputStream in = new ByteArrayInputStream(buffer);
objectMeta.setContentLength(0);
try {
    client.putObject(bucketName, objectName, in, objectMeta);
} finally {
    in.close();
}
```

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照样可以上传下载,只是控制台会对以"/"结尾的Object以文件夹的方式展示。所以用户可以使用上述方式来实现创建模拟文件夹。而对文件夹的访问可以参看文件夹模拟功能

设定Object的ACL

OSS不仅提供了Bucket的ACL，也提供了Object的ACL设定。

Object的ACL | 权限值 | 中文名称 | 权限对访问者的限制 | |----------|----------|----------| | public-read-write | 公共读写 | 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。 | | public-read | 公共读，私有写 | 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。 | | private | 私有读写 | 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。 | | default | 默认权限 | 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 |

设置Object ACL注意事项

- 如果没有设置Object的权限，即Object的ACL为default，Object的权限和Bucket权限一致。
- 如果设置了Object的权限，Object的权限大于Bucket权限。举个例子，如果设置了Object的权限是public-read，无论Bucket是什么权限，该Object都可以被身份验证访问和匿名访问。

下面代码为Object设置了ACL:

```
// 初始化OSSClient
OSSClient client = ...;
private static final CannedAccessControlList[] ACLS = {
    CannedAccessControlList.Private,
    CannedAccessControlList.PublicRead,
    CannedAccessControlList.PublicReadWrite,
    CannedAccessControlList.Default
};

// 上传文件
final String key = "normal-set-object-acl";
final long inputStreamLength = 128 * 1024; //128KB
InputStream instream = genFixedLengthInputStream(inputStreamLength);
client.putObject(bucketName, key, instream, null);

//设置Object ACL
for (CannedAccessControlList acl : ACLS) {
    client.setObjectAcl(bucketName, key, acl);
}
```

获取Object ACL

下面代码读取Object设置的ACL，和设置时候的ACL权限一样

```
// 初始化OSSClient
OSSClient client = ...;

//读取Object ACL
    ObjectAcl returnedAcl = client.getObjectAcl(bucketName, key);
    System.out.println(returnedAcl.getPermission().toString());
}
```

设定Object的Http Header

OSS服务允许用户自定义Object的Http Header。下面代码为Object设置了过期时间：

```
// 初始化OSSClient
OSSClient client = ...;

// 初始化上传输入流
InputStream content = ...;

// 创建上传Object的Metadata
ObjectMetadata meta = new ObjectMetadata();

// 设置ContentLength为1000
meta.setContentLength(1000);

// 设置1小时后过期
Date expire = new Date(new Date().getTime() + 3600 * 1000);
meta.setExpirationTime(expire);
client.putObject(bucketName, key, content, meta);
```

Java SDK支持的Http Header有四种，分别为：Cache-Control 、 Content-Disposition 、Content-Encoding 、 Expires 。它们的相关介绍见 RFC2616 。

用户自定义元信息

OSS支持用户自定义元信息来对Object进行描述。比如：

```
// 设置自定义元信息name的值为my-data
meta.addUserMetadata("name", "my-data");

// 上传object
client.putObject(bucketName, key, content, meta);
```

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Object的时候，此元信息也可以一并得到。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2KB。

NOTE：user meta的名称大小写不敏感，比如用户上传object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”

使用Chunked编码上传

当无法确认上传内容的长度时（比如SocketStream作为上传的数据源,边接受边上传，直至Socket关闭为止），需要采用chunked编码。

chunked采用以下方式编码：

```
Chunked-Body = *chunk
"0" CRLF
footer
CRLF
chunk = chunk-size [chunk-ext] CRLF
chunk-data CRLF

hex-no-zero = <HEX excluding "0">
```

```
chunk-size = hex-no-zero * HEX
chunk-ext = *( ";" chunk-ext-name [ "=" chunk-ext-value ] )
chunk-ext-name = token
chunk-ext-val = token | quoted-string
chunk-data = chunk-size(OCTET)

footer = *entity-header
```

编码使用若干个Chunk组成，由一个标明长度为0的chunk结束，每个Chunk有两部分组成，第一部分是该Chunk的长度和长度单位（一般不写），第二部分就是指定长度的内容，每个部分用CRLF隔开。在最后一个长度为0的Chunk中的内容是称为footer的内容，是一些没有写的头部内容。

putobject chunked编码 当显式设置ObjectMetadata实例中的ContentLength属性时，采用普通方式上传（由Content-Length请求头决定请求body的长度）；反之，采用chunked编码方式上传。

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);

FileInputStream fin = new FileInputStream(new File(filePath));
// 如果不设置content-length, 默认为chunked编码。
PutObjectResult result = client.putObject(bucketName, key, fin, new ObjectMetadata());
```

追加上传

OSS允许用户通过追加上传（Append Object）的方式在一个Object后面直接追加内容。前提是该Object类型为Appendable。通过Append Object操作创建的Object类型为Appendable Object，而通过Put Object方式上传是Normal Object。

```
// 创建OSSClient实例
OSSClient client = new OSSClient(ENDPOINT, ACCESS_ID, ACCESS_KEY);

// 发起首次追加Object请求，注意首次追加需要设置追加位置为0
final String fileToAppend = "<file to append at first time>";
AppendObjectRequest appendObjectRequest = new AppendObjectRequest(bucketName, key, new File(fileToAppend));

// 设置content-type，注意设置object meta只能在使用Append创建Object设置
ObjectMetadata meta = new ObjectMetadata();
meta.setContentType("image/jpeg");
appendObjectRequest.setMetadata(meta);

// 设置追加位置为0,发送追加Object请求
appendObjectRequest.setPosition(0L);
AppendObjectResult appendObjectResult = client.appendObject(appendObjectRequest);

// 发起第二次追加Object请求，追加位置为第一次追加后的Object长度
final String fileToAppend2 = "<file to append at second time>";
appendObjectRequest = new AppendObjectRequest(bucketName, key, new File(fileToAppend2));

// 设置追加位置为前一次追加文件的大小,发送追加Object请求
appendObjectRequest.setPosition(appendObjectResult.getNextPosition());
appendObjectResult = client.appendObject(appendObjectRequest);

OSSObject o = client.getObject(bucketName, key);
```

```
// 当前该Object的大小为两次追加文件的大小总和
System.out.println(o.getObjectMetadata().getContentLength());
// 下一个追加位置为前两次追加文件的大小总和
System.err.println(appendObjectResult.getNextPosition().longValue());
```

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，如上述代码。另一种是通过下面要讲到的getObjectMetadata获取该Object的当前长度。

> 对于Object meta的设置只在使用追加上传创建Object时设置。后续如果需要更改该Object meta，可以通过下面要讲到的copy object接口实现（源和目的为同一个Object）。

分块上传

OSS允许用户将一个Object分成多个请求上传到后台服务器中，关于分块上传的内容，将在MultipartUpload中Object的分块上传 这一章中做介绍。

列出Bucket中的Object

列出Object

```
public void listObjects(String bucketName) {

    // 初始化OSSClient
    OSSClient client = ...;

    // 获取指定bucket下的所有Object信息
    ObjectListing listing = client.listObjects(bucketName);

    // 遍历所有Object
    for (OSSObjectSummary objectSummary : listing.getObjectSummaries()) {
        System.out.println(objectSummary.getKey());
    }
}
```

listObjects方法会返回 ObjectListing 对象，ObjectListing 对象包含了此次listObject请求的返回结果。其中我们可以通过 ObjetListing 中的 getObjectSummaries 方法获取所有Object的描述信息（List<OSSObjectSummary>）。

> NOTE：默认情况下，如果Bucket中的Object数量大于100，则只会返回100个Object， 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Object数目，可以修改 MaxKeys 参数，或者使用 Marker 参数分次读取。

拓展参数

通常，我们可以通过设置ListObjectsRequest的参数来完成更强大的功能。比如：

```
// 构造ListObjectsRequest请求
```

```
ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);

// 设置参数
listObjectsRequest.setDelimiter("/");
listObjectsRequest.setMarker("123");
...

ObjectListing listing = client.listObjects(listObjectsRequest);
```

上面代码中我们调用了 listObjects 的一个重载方法，通过传入 ListObjectsRequest 来完成请求。通过 ListObjectsRequest 中的参数设置我们可以完成很多扩展的功能。下表列出了 ListObjectsRequest 中可以设置的参数名称和作用：

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。

如果需要遍历所有的object，而object数量大于1000，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object的key作为本次迭代中的Marker即可。

文件夹功能模拟

我们可以通过 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Bucket内所有文件

当我们需要获取Bucket下的所有文件时，可以这样写：

```
// 构造ListObjectsRequest请求
ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);

// List Objects
ObjectListing listing = client.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下所有的文件：

```
// 构造ListObjectsRequest请求
ListObjectsRequest listObjectsRequest = new ListObjectsRequest(bucketName);

// 递归列出fun目录下的所有文件
listObjectsRequest.setPrefix("fun/");

ObjectListing listing = client.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 = client.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/
```

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。

获取Object

简单的读取Object

我们可以通过以下代码将Object读取到一个流中：

```
public void getObject(String bucketName, String key) throws IOException {

    // 初始化OSSClient
    OSSClient client = ...;

    // 获取Object，返回结果为OSSObject对象
    OSSObject object = client.getObject(bucketName, key);

    // 获取ObjectMeta
    ObjectMetadata meta = object.getObjectMetadata();

    // 获取Object的输入流
    InputStream objectContent = object.getObjectContent();
```

```
    // 处理Object
    ...

    // 关闭流
    objectContent.close();
}
```

OSSObject 包含了Object的各种信息，包含Object所在的Bucket、Object的名称、Metadata以及一个输入流。我们可以通过操作输入流将Object的内容读取到文件或者内存中。而ObjectMetadata包含了Object上传时定义的，ETag，Http Header以及自定义的元信息。

通过GetObjectRequest获取Object

为了实现更多的功能，我们可以通过使用 GetObjectRequest 来获取Object。

```
// 初始化OSSClient
OSSClient client = ...;

// 新建GetObjectRequest
GetObjectRequest getObjectRequest = new GetObjectRequest(bucketName, key);

// 获取0~100字节范围内的数据
getObjectRequest.setRange(0, 100);

// 获取Object，返回结果为OSSObject对象
OSSObject object = client.getObject(getObjectRequest);
```

我们通过 getObjectRequest 的 setRange 方法设置了返回的Object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。
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object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Object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。

修改 ResponseHeaderOverrides ， 它提供了一系列的可修改参数，可以自定义OSS的返回Header，如下表所示：

参数	说明

ContentType	OSS返回请求的content-type头
ContentLanguage	OSS返回请求的content-language头
Expires	OSS返回请求的expires头
CacheControl	OSS返回请求的cache-control头
ContentDisposition	OSS返回请求的content-disposition头
ContentEncoding	OSS返回请求的content-encoding头

直接下载Object到文件

我们可以通过下面的代码直接将Object下载到指定文件：

```
// 新建GetObjectRequest
GetObjectRequest getObjectRequest = new GetObjectRequest(bucketName, key);

// 下载Object到文件
ObjectMetadata objectMetadata = client.getObject(getObjectRequest, new File("/path/to/file"));
```

当使用上面方法将Object直接下载到文件时，方法返回ObjectMetadata对象。

只获取ObjectMetadata

通过 getObjectMetadata 方法可以只获取ObjectMetadata而不获取Object的实体。代码如下：

```
ObjectMetadata objectMetadata = client.getObjectMetadata(bucketName, key);
```

删除Object

下面代码删除了一个Object:

```
public void deleteObject(String bucketName, String key) {
    // 初始化OSSClient
    OSSClient client = ...;

    // 删除Object
    client.deleteObject(bucketName, key);
}
```

拷贝Object

拷贝一个Object

通过 copyObject 方法我们可以拷贝一个Object，代码如下：

```
public void copyObject(String srcBucketName, String srcKey, String destBucketName, String destKey) {
    // 初始化OSSClient
    OSSClient client = ...;

    // 拷贝Object
    CopyObjectResult result = client.copyObject(srcBucketName, srcKey, destBucketName, destKey);
```

```
    // 打印结果
    System.out.println("ETag: " + result.getETag() + " LastModified: " + result.getLastModified());
}
```

copyObject 方法返回一个 CopyObjectResult 对象，对象中包含了新Object的ETag和修改时间。使用该方法copy的object必须小于1G，否则会报错。若object大于1G，使用后文的Upload Part Copy

通过CopyObjectRequest拷贝Object

也可以通过 CopyObjectRequest 实现Object的拷贝：

```
// 初始化OSSClient
OSSClient client = ...;

// 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 = client.copyObject(copyObjectRequest);

System.out.println("ETag: " + result.getETag() + " LastModified: " + result.getLastModified());
```

CopyObjectRequest 允许用户修改目的Object的ObjectMeta，同时也提供 ModifiedSinceConstraint ，UnmodifiedSinceConstraint ，MatchingETagConstraints ， NonmatchingEtagConstraints 四个参数的设定， 用法与 GetObjectRequest 的参数相似，参见 GetObjectRequest的可设置参数。

NOTE：可以通过拷贝操作来实现修改已有 Object 的 meta 信息。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

POST 方式上传文件

对于后端服务关键是需要为前端提供policy以及Signature这两个表单域。

生成POST Policy

Post请求的policy表单域用于验证请求的合法性。 poicy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。详细的policy josn字串生成规则查看API文档Post Object中Post Policy。

比如配置如下policy字串：

```
{"expiration":"2015-02-25T14:25:46.000Z",
"conditions":[
    {"bucket":"oss-test2"},
    ["eq","$key","user/eric/${filename}"],
    ["starts-with","$key","user/eric"],
    ["starts-with","$x-oss-meta-tag","dummy_etag"],
```

```
    ["content-length-range",1,1024]
    ]
}
```

可以通过下面的代码生成上述json字串：

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);

Date expiration = DateUtil.parseIso8601Date("2015-02-25T14:25:46.000Z");
PolicyConditions policyConds = new PolicyConditions();
policyConds.addConditionItem("bucket", bucketName);
// 添加精确匹配条件项 “$”必须紧接大括号
policyConds.addConditionItem(MatchMode.Exact, PolicyConditions.COND_KEY, "user/eric/\\${filename}");
// 添加前缀匹配条件项
policyConds.addConditionItem(MatchMode.StartWith, PolicyConditions.COND_KEY, "user/eric");
policyConds.addConditionItem(MatchMode.StartWith, "x-oss-meta-tag", "dummy_etag");
// 添加范围匹配条件项
policyConds.addConditionItem(PolicyConditions.COND_CONTENT_LENGTH_RANGE, 1, 1024);

// 生成Post Policy字符串
String postPolicy = client.generatePostPolicy(expiration, policyConds);
System.out.println(postPolicy);

// 计算policy Base64编码
byte[] binaryData = postPolicy.getBytes("utf-8");
String encodedPolicy = BinaryUtil.toBase64String(binaryData);
System.out.println(encodedPolicy);
```

在生成policy字串之后，注意的是POST表单中的policy需要进行Base64编码。

生成Post Signature

同时，为了完成POST上传，需要生成Post Signature来验证请求合法性。可以参考下面代码。

```
//传入Post Policy原json字串，生成postSignature
String postSignature = client.calculatePostSignature(postPolicy);
System.out.println(postSignature);
```

MultipartUpload

Multipart Upload

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。

下面我们将一步步介绍怎样实现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块上传事件：

```
String bucketName = "your-bucket-name";
String key = "your-key";

// 初始化OSSClient
OSSClient client = ...;
// 开始Multipart Upload
InitiateMultipartUploadRequest initiateMultipartUploadRequest = new InitiateMultipartUploadRequest(bucketNam
e, key);
InitiateMultipartUploadResult initiateMultipartUploadResult = client.initiateMultipartUpload(initiateMultipartUpload
Request);
// 打印UploadId
System.out.println("UploadId: " + initiateMultipartUploadResult.getUploadId());
```

我们用 InitiateMultipartUploadRequest 来指定上传Object的名字和所属Bucket。在 InitiateMultipartUploadRequest 中，您也可以设置 ObjectMetadata ，但是不必指定其中的 ContentLength 。 initiateMultipartUpload 的返回结果中含有 UploadId ，它是区分分块上传事件的唯一标识，在后面的操作中，我们将用到它。

接下来我们可以有两种方式来传分块，一种是使用Upload Part从本地上传，另外一种是通过Upload Part copy从bucket中的object复制得到。

Upload Part本地上传

接着，我们把本地文件分块上传。 假设有一个文件，本地路径为 /path/to/file.zip 由于文件比较大，我们将其分块传输到OSS中。

```
// 设置每块为 5M
final int partSize = 1024 * 1024 * 5;
File partFile = new File("/path/to/file.zip");
// 计算分块数目
int partCount = (int) (partFile.length() / partSize);
if (partFile.length() % partSize != 0){
    partCount++;
}
// 新建一个List保存每个分块上传后的ETag和PartNumber
List<PartETag> partETags = new ArrayList<PartETag>();
for(int i = 0; i < partCount; i++){
    // 获取文件流
    FileInputStream fis = new FileInputStream(partFile);
    // 跳到每个分块的开头
    long skipBytes = partSize * i;
    fis.skip(skipBytes);
    // 计算每个分块的大小
    long size = partSize < partFile.length() - skipBytes ?
            partSize : partFile.length() - skipBytes;
    // 创建UploadPartRequest，上传分块
    UploadPartRequest uploadPartRequest = new UploadPartRequest();
    uploadPartRequest.setBucketName(bucketName);
```

```
    uploadPartRequest.setKey(key);
    uploadPartRequest.setUploadId(initiateMultipartUploadResult.getUploadId());
    uploadPartRequest.setInputStream(fis);
    uploadPartRequest.setPartSize(size);
    uploadPartRequest.setPartNumber(i + 1);
    UploadPartResult uploadPartResult = client.uploadPart(uploadPartRequest);
    // 将返回的PartETag保存到List中。
    partETags.add(uploadPartResult.getPartETag());
    // 关闭文件
    fis.close();
}
```

上面程序的核心是调用 uploadPart 方法来上传每一个分块，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。
- 为了保证数据在网络传输过程中不出现错误，用户上传是可以设置Content-MD5，OSS会计算上传数据的MD5值与用户上传的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，在后续完成分块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。

Upload Part本地chunked上传

对于分块上传也支持Chunked编码上传

```
File file = new File(filePath);
// 设置每块为 5M
final int partSize = 5 * 1024 * 1024;
int fileSize = (int) file.length();
// 计算分块数目
final int partCount = (file.length() % partSize != 0) ? (fileSize / partSize + 1) : (fileSize / partSize);
List<PartETag> partETags = new ArrayList<PartETag>();

for (int i = 0; i < partCount; i++) {
    InputStream fin = new BufferedInputStream(new FileInputStream(file));
    fin.skip(i * partSize);
    int size = (i + 1 == partCount) ? (fileSize - i * partSize) : partSize;

    UploadPartRequest req = new UploadPartRequest();
    req.setBucketName(bucketName);
    req.setKey(key);
    req.setPartNumber(i + 1);
    req.setPartSize(size);
    req.setUploadId(uploadId);
    req.setInputStream(fin);
    req.setUseChunkEncoding(true);  // 使用chunked编码

    UploadPartResult result = client.uploadPart(req);
```

```
    partETags.add(result.getPartETag());

    fin.close();
}
```

在UploadPartRequest这个类的实例里面，通过设置setUseChunkEncoding(true)来使得上传为chunked编码。

Upload Part Copy拷贝上传

Upload Part Copy 通过从一个已经存在的object中拷贝数据来上传一个object。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。

```
ObjectMetadata objectMetadata = client.getObjectMetadata(sourceBucketName,sourceKey);

long partSize = 1024 * 1024 * 100;
// 得到被拷贝object大小
long contentLength = objectMetadata.getContentLength();

// 计算分块数目
int partCount = (int) (contentLength / partSize);
if (contentLength % partSize != 0) {
    partCount++;
}
System.out.println("total part count:" + partCount);
List<PartETag> partETags = new ArrayList<PartETag>();

long startTime = System.currentTimeMillis();
for (int i = 0; i < partCount; i++) {
    System.out.println("now begin to copy part:" + (i+1));
    long skipBytes = partSize * i;
// 计算每个分块的大小
    long size = partSize < contentLength - skipBytes ? partSize : contentLength - skipBytes;
// 创建UploadPartCopyRequest，上传分块
    UploadPartCopyRequest uploadPartCopyRequest = new UploadPartCopyRequest();
    uploadPartCopyRequest.setSourceKey("/" + sourceBucketName + "/" + sourceKey);
    uploadPartCopyRequest.setBucketName(targetBucketName);
    uploadPartCopyRequest.setKey(targetKey);
    uploadPartCopyRequest.setUploadId(uploadId);
    uploadPartCopyRequest.setPartSize(size);
    uploadPartCopyRequest.setBeginIndex(skipBytes);
    uploadPartCopyRequest.setPartNumber(i + 1);
    UploadPartCopyResult uploadPartCopyResult = client.uploadPartCopy(uploadPartCopyRequest);
// 将返回的PartETag保存到List中。
     partETags.add(uploadPartCopyResult.getPartETag());
     System.out.println("now end to copy part:" + (i+1));
}
```

以上程序调用uploadPartCopy 方法来拷贝每一个分块。与UploadPart要求基本一致，需要通过setBeginIndex来定位到此次上传块开头所对应的位置，同时需要通过setSourceKey来指定copy的object

完成分块上传

完成分块上传代码如下：

```
CompleteMultipartUploadRequest completeMultipartUploadRequest =
        new CompleteMultipartUploadRequest(bucketName,key, initiateMultipartUploadResult.getUploadId(), partETa
gs);

// 完成分块上传
CompleteMultipartUploadResult completeMultipartUploadResult =
        client.completeMultipartUpload(completeMultipartUploadRequest);

// 打印Object的ETag
System.out.println(completeMultipartUploadResult.getETag());
```

上面代码中的 partETags 就是进行分块上传中保存的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Object。

取消分块上传事件

我们可以用 abortMultipartUpload 方法取消分块上传。

```
AbortMultipartUploadRequest abortMultipartUploadRequest =
            new AbortMultipartUploadRequest(bucketName, key, uploadId);

// 取消分块上传
client.abortMultipartUpload(abortMultipartUploadRequest);
```

获取Bucket内所有分块上传事件

我们可以用 listMultipartUploads 方法获取Bucket内所有上传事件。

```
// 获取Bucket内所有上传事件
ListMultipartUploadsRequest listMultipartUploadsRequest=new ListMultipartUploadsRequest(bucketName);
MultipartUploadListing listing = client.listMultipartUploads(listMultipartUploadsRequest);

// 遍历所有上传事件
for (MultipartUpload multipartUpload : listing.getMultipartUploads()) {
    System.out.println("Key: " + multipartUpload.getKey() + " UploadId: " + multipartUpload.getUploadId());
}
```

NOTE：默认情况下，如果Bucket中的分块上传事件的数量大于1000，则只会返回1000个Object， 且返回结果中 IsTruncated 为 false，反回 NextKeyMarker 和 NextUploadMarker 作为下此读取的起点。若想增大返回分块上传事件数目，可以修改 MaxUploads 参数，或者使用 KeyMarker 以及 UploadIdMarker 参数分次读取。

获取所有已上传的块信息

我们可以用 listParts 方法获取某个上传事件所有已上传的块。

```
ListPartsRequest listPartsRequest = new ListPartsRequest(bucketName, key, uploadId);

// 获取上传的所有Part信息
PartListing partListing = client.listParts(listPartsRequest);

// 遍历所有Part
for (PartSummary part : partListing.getParts()) {
```

```
    System.out.println("PartNumber: " + part.getPartNumber() + " ETag: " + part.getETag());
}
```

NOTE：默认情况下，如果Bucket中的分块上传事件的数量大于1000，则只会返回1000个Object， 且返回结果中 IsTruncated 为 false，并返回 NextPartNumberMarker 作为下此读取的起点。若想增大返回分块上传事件数目，可以修改 MaxParts 参数，或者使用 PartNumberMarker 参数分次读取。

跨域资源共享

跨域资源共享（CORS）

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。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);
ArrayList<CORSRule> putCorsRules = new ArrayList<CORSRule>();
//CORS规则的容器,每个bucket最多允许10条规则

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);
```

其中需要特别注意的是：

- 每个bucket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bucket的CORS规则，通过getBucketCORSRules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Bucket对应的CORS并清空所有规则。

```
// 清空bucket的CORS规则
oss.deleteBucketCORSRules(bucketName);
```

同样的，只有bucket的拥有者才能删除规则。

防盗链

防盗链设置

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：

```
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);
```

Referer参数支持通配符“*”和“？”，更多详细的规则配置可以参考用户手册OSS防盗链

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);
```

生命周期管理

生命周期管理（Lifecycle）

OSS提供Object生命周期管理来为用户管理对象。用户可以为某个Bucket定义生命周期配置，来为该Bucket的Object定义各种规则。目前，用户可以通过规则来删除相匹配的Object。每条规则都由如下几个部分组成：

- Object名称前缀，只有匹配该前缀的Object才适用这个规则

- 操作，用户希望对匹配的Object所执行的操作。
- 日期或天数，用户期望在特定日期或者是在Object最后修改时间后多少天执行指定的操作。

设置Lifecycle

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>
  <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object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object，Date则表示到指定的绝对时间之后就删除object(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。

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);

SetBucketLifecycleRequest req = new SetBucketLifecycleRequest(bucketName);
// 添加Lifecycle规则
req.AddLifecycleRule(new LifecycleRule("delete obsoleted files", "obsoleted/", RuleStatus.Enabled, 3));
req.AddLifecycleRule(new LifecycleRule("delete temporary files", "temporary/", RuleStatus.Enabled,
        DateUtil.parseIso8601Date("2022-10-12T00:00:00.000Z")));
// 设置Bucket Lifecycle
client.setBucketLifecycle(req);
```

可以通过下面的代码获取上述lifecycle规则。

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);

// 获取上述Bucket Lifecycle
List<LifecycleRule> rules = client.getBucketLifecycle(bucketName);
```

```
Assert.assertEquals(rules.size(), 2);

System.out.println("Rule1: ");
LifecycleRule r1 = rules.get(0);
System.out.println("ID: " + r1.getId());
System.out.println("Prefix: " + r1.getPrefix());
System.out.println("Status: " + r1.getStatus().toString());
System.out.println("ExpirationDays: " + r1.getExpriationDays());
System.out.println();

System.out.println("Rule2: ");
LifecycleRule r2 = rules.get(1);
System.out.println("ID: " + r2.getId());
System.out.println("Prefix: " + r2.getPrefix());
System.out.println("Status: " + r2.getStatus().toString());
System.out.println("ExpirationTime: " + DateUtil.formatIso8601Date(r2.getExpirationTime()));
```

可以通过下面的代码清空bucket中lifecycle规则。

```
OSSClient client = new OSSClient(endpoint, accessId, accessKey);

client.deleteBucketLifecycle(bucketName);
```

授权访问

授权访问

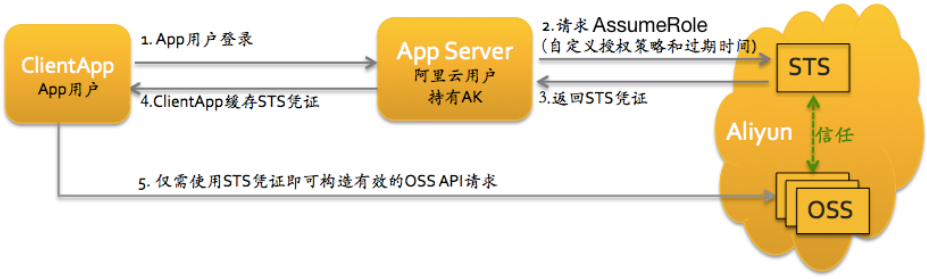
使用STS服务临时授权

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。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);
```

一个简单的示例

Server端收到一个授权请求，某一客户端用户baymax希望能够对Bucket：MyAppStorage下user/baymax/路径中的文件拥有写操作。

首先在通过STS SDK在Sever端的生成STS临时凭证代码如下：

```
import com.aliyuncs.DefaultAcsClient;
import com.aliyuncs.exceptions.ClientException;
import com.aliyuncs.http.MethodType;
import com.aliyuncs.http.ProtocolType;
import com.aliyuncs.profile.DefaultProfile;
import com.aliyuncs.profile.IClientProfile;
import com.aliyuncs.sts.model.v20150401.AssumeRoleRequest;
import com.aliyuncs.sts.model.v20150401.AssumeRoleResponse;

public class StsServiceSample {
  public static final String REGION_CN_HANGZHOU = "cn-hangzhou";
```

```
  public static final String STS_API_VERSION = "2015-04-01";
  public static final String STS_VERSION = "1";

  static AssumeRoleResponse assumeRole(String accessKeyId, String accessKeySecret,
                                       String roleArn, String roleSessionName, String policy, ProtocolType protocolType) throws Cl
ientException {
    try {
      IClientProfile profile = DefaultProfile.getProfile(REGION_CN_HANGZHOU, accessKeyId, accessKeySecret);
      DefaultAcsClient client = new DefaultAcsClient(profile);

      final AssumeRoleRequest request = new AssumeRoleRequest();
      request.setVersion(STS_API_VERSION);
      request.setMethod(MethodType.POST);
      request.setProtocol(protocolType);

      request.setRoleArn(roleArn);
      request.setRoleSessionName(roleSessionName);
      request.setPolicy(policy);

      final AssumeRoleResponse response = client.getAcsResponse(request);

      return response;
    } catch (ClientException e) {
      throw e;
    }
  }
  public static void main(String[] args) {
    String accessKeyId = "o************F";
    String accessKeySecret = "y******************U";
    String roleArn = "acs:ram::145883****900618:role/ossadminrole";
    String roleSessionName = "alice";
    String policy = "{\n" +
            "    \"Version\": \"1\", \n" +
            "    \"Statement\": [\n" +
            "        {\n" +
            "            \"Action\": [\n" +
            "                \"oss:GetBucket\", \n" +
            "                \"oss:GetObject\" \n" +
            "            ], \n" +
            "            \"Resource\": [\n" +
            "                \"acs:oss:*:177530****529849:mybucket\", \n" +
            "                \"acs:oss:*:177530****529849:mybucket/*\" \n" +
            "            ], \n" +
            "            \"Effect\": \"Allow\"\n" +
            "        }\n" +
            "    ]\n" +
            "}";
    ProtocolType protocolType = ProtocolType.HTTPS;

    try {
      final AssumeRoleResponse response = assumeRole(accessKeyId, accessKeySecret,
              roleArn, roleSessionName, policy, protocolType);

      System.out.println("Expiration: " + response.getCredentials().getExpiration());
      System.out.println("AccessKeyId: " + response.getCredentials().getAccessKeyId());
      System.out.println("AccessKeySecret: " + response.getCredentials().getAccessKeySecret());
```

```
      System.out.println("Security Token: " + response.getCredentials().getSecurityToken());
    } catch (ClientException e) {
      System.out.println("Failed to get a federation token.");
      System.out.println("Error code: " + e.getErrCode());
      System.out.println("Error message: " + e.getErrMsg());
    }

  }
}
```

客户端拿到上述服务端生成STS临时凭证，使用该凭证完成一次数据上传：

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.util.List;

import com.aliyun.oss.ClientException;
import com.aliyun.oss.OSSClient;
import com.aliyun.oss.OSSErrorCode;
import com.aliyun.oss.OSSException;
import com.aliyun.oss.ServiceException;
import com.aliyun.oss.model.CannedAccessControlList;
import com.aliyun.oss.model.GetObjectRequest;
import com.aliyun.oss.model.OSSObjectSummary;
import com.aliyun.oss.model.ObjectListing;
import com.aliyun.oss.model.ObjectMetadata;

public class OSSObjectSample {


    public static void main(String[] args) throws Exception {
      String bucketName = "MyAppStorage";
      String filepath= "/user/baymax/";
        String accessKeyId = "<your STS AccessKeyId>";
        String accessKeySecret = "<your STS AccessKeySecret>";
        String securityToken = "<your STS securityToken>";
          String endpoint = "http://oss-cn-hangzhou.aliyuncs.com";
      String uploadFilePath = "d:/temp/photo.jpg";

      OSSClient client = new OSSClient(endpoint,accessKeyId,accessKeySecret,securityToken);
      File file = new File(uploadFilePath);
      InputStream content = new FileInputStream(file);
      // 创建上传Object的Metadata
      ObjectMetadata meta = new ObjectMetadata();
      // 必须设置ContentLength
      meta.setContentLength(file.length());
      String key=filepath+"photo.jpg";
      PutObjectResult result = client.putObject(bucketName, key, content, meta);
    }
}
```

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 GeneratePresignedUrlRequest(bucketName, ke
y);

// 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,  “” ,  “” );
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, accessId, accessKey);
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,  “” ,  “” );
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);
```

异常

异常

OSS Java 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
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

Python-SDK

前言

前言

SDK下载

Python SDK开发包(2015-11-20) 版本 0.4.2：OSS_Python_API_20151120.zip

版本迭代详情参考这里

简介

本文档主要介绍OSS Python SDK的安装和使用，针对于0.4.2。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。

安装

安装

下面介绍一个完整的过程，示范如何使用Python在Windows平台和Linux平台上进行OSS bucket和object的基本操作。

环境要求

下面介绍一个完整的过程，示范如何使用Python在Windows平台和Linux平台上进行OSS bucket和object的基本操作。

安装好Python后：

- Linux shell环境下输入python并回车，来查看Python的版本。如下所示：

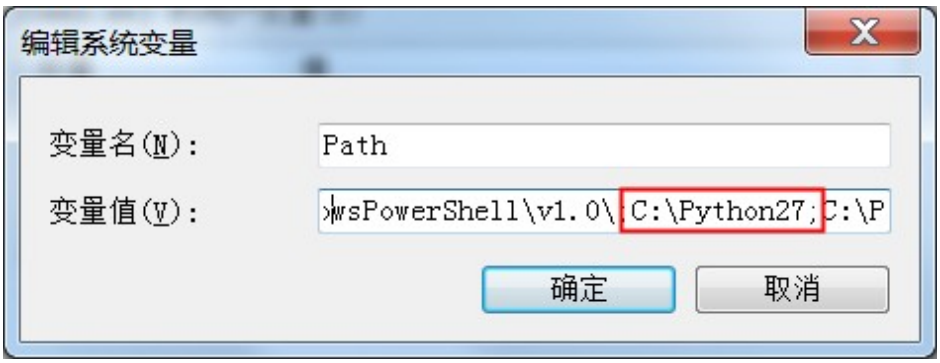
```
Python 2.5.4 (r254:67916, Mar 10 2010, 22:43:17)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-46)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Windows在cmd环境下输入python并回车，来查看Python的版本。如下所示：

```
C:\Documents and Settings\Administrator>python
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

以上说明python安装成功。

- 异常情况，如Windows在cmd环境下输入python并回车后，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:



如果没有安装Python，可以从python官网获取Python的安装包。网站有详细的安装说明来指导用户如何安装和使用Python。

安装和验证SDK

下面介绍在Windows平台和Linux平台上如何安装Python SDK，以及如何验证SDK是否安装成功。

安装SDK

1. 打开浏览器，输入oss.aliyun.com。
2. 在“产品帮助”-“开发者资源”这个位置找到Python SDK链接。
3. 点击链接并选择保存SDK安装包。
4. 下载后可以得到类似OSS_Python_API_xxxxxxxx.tar.gz的安装包。
5. 进入安装包所在的目录，将tar.gz的包进行解压缩。Linux可以使用tar -zxvf OSS_Python_API_xxxxxxxx.tar.gz命令解压缩。Windows可以使用7-Zip等解压缩工具。
6. 解压缩后得到的文件及目录如下

```
README
OSS_Python_SDK.pdf
setup.py
```

```
osscmd
oss/
oss_api.py
oss_util.py
oss_xml_handler.py
oss_sample.py
```

7. 两个平台的SDK安装

- 假设SDK已经解压缩到Windows平台的D盘，安装的日志如下：

```
D:\>cd  OSS_Python_API_20130712
D:\ OSS_Python_API_20130712 >python setup.py install
running install
running build
running build_py
creating build
creating build\lib
creating build\lib\oss
copying oss\oss_api.py -> build\lib\oss
copying oss\oss_sample.py -> build\lib\oss
copying oss\oss_util.py -> build\lib\oss
copying oss\oss_xml_handler.py -> build\lib\oss
copying oss\pkg_info.py -> build\lib\oss
copying oss\__init__.py -> build\lib\oss
running install_lib
creating C:\Python27\Lib\site-packages\oss
copying build\lib\oss\oss_api.py -> C:\Python27\Lib\site-packages\oss
copying build\lib\oss\oss_sample.py -> C:\Python27\Lib\site-packages\oss
copying build\lib\oss\oss_util.py -> C:\Python27\Lib\site-packages\oss
copying build\lib\oss\oss_xml_handler.py -> C:\Python27\Lib\site-packages\oss
copying build\lib\oss\pkg_info.py -> C:\Python27\Lib\site-packages\oss
copying build\lib\oss\__init__.py -> C:\Python27\Lib\site-packages\oss
byte-compiling C:\Python27\Lib\site-packages\oss\oss_api.py to oss_api.pyc
byte-compiling C:\Python27\Lib\site-packages\oss\oss_sample.py to oss_sample.pyc
byte-compiling C:\Python27\Lib\site-packages\oss\oss_util.py to oss_util.pyc
byte-compiling C:\Python27\Lib\site-packages\oss\oss_xml_handler.py to oss_xml_handler.pyc
byte-compiling C:\Python27\Lib\site-packages\oss\pkg_info.py to pkg_info.pyc
byte-compiling C:\Python27\Lib\site-packages\oss\__init__.py to __init__.pyc
running install_egg_info
Writing C:\Python27\Lib\site-packages\oss-0.1.3-py2.7.egg-info
```

- 假设SDK已经解压缩到Linux平台的oss目录，安装的日志如下：

```
[oss@oss python]$ sudo python setup.py install
Password:
running install
running bdist_egg
running egg_info
writing oss.egg-info/PKG-INFO
writing top-level names to oss.egg-info/top_level.txt
writing dependency_links to oss.egg-info/dependency_links.txt
writing manifest file 'oss.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
```

```
running build_py
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/oss
copying build/lib/oss/oss_util.py -> build/bdist.linux-x86_64/egg/oss
copying build/lib/oss/__init__.py -> build/bdist.linux-x86_64/egg/oss
copying build/lib/oss/pkg_info.py -> build/bdist.linux-x86_64/egg/oss
copying build/lib/oss/oss_xml_handler.py -> build/bdist.linux-x86_64/egg/oss
copying build/lib/oss/oss_sample.py -> build/bdist.linux-x86_64/egg/oss
copying build/lib/oss/oss_api.py -> build/bdist.linux-x86_64/egg/oss
byte-compiling build/bdist.linux-x86_64/egg/oss/oss_util.py to oss_util.pyc
byte-compiling build/bdist.linux-x86_64/egg/oss/__init__.py to __init__.pyc
byte-compiling build/bdist.linux-x86_64/egg/oss/pkg_info.py to pkg_info.pyc
byte-compiling build/bdist.linux-x86_64/egg/oss/oss_xml_handler.py to oss_xml_handler.pyc
byte-compiling build/bdist.linux-x86_64/egg/oss/oss_sample.py to oss_sample.pyc
byte-compiling build/bdist.linux-x86_64/egg/oss/oss_api.py to oss_api.pyc
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying oss.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying oss.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying oss.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying oss.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
zip_safe flag not set; analyzing archive contents...
oss.oss_sample: module references __file__
creating 'dist/oss-0.1.3-py2.5.egg' and adding 'build/bdist.linux-x86_64/egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing oss-0.1.3-py2.5.egg
removing '/usr/ali/lib/python2.5/site-packages/oss-0.1.3-py2.5.egg' (and everything under it)
creating /usr/ali/lib/python2.5/site-packages/oss-0.1.3-py2.5.egg
Extracting oss-0.1.3-py2.5.egg to /usr/ali/lib/python2.5/site-packages
oss 0.1.3 is already the active version in easy-install.pth
Installed /usr/ali/lib/python2.5/site-packages/oss-0.1.3-py2.5.egg
Processing dependencies for oss==0.1.3
Finished processing dependencies for oss==0.1.3
```

8. SDK的卸载 不同的机器和平台，SDK安装的目录不一样。卸载需要进入到SDK在安装的目录，将相关的安装目录删除即可。例如上一步所示的/usr/ali/lib/python2.5/site-packages/oss-0.1.3-py2.5.egg目录。 也可以通过下面的命令，从当前版本Python的site-packages 目录中找到oss相关的安装目录删除来卸载。

```
>>> import sys
>>> print sys.path
['','/usr/ali/lib/python2.5/site-packages/httplib2-0.7.7-py2.5.egg', '/usr/ali/lib/python2.5/site-
packages/oss-0.1.3-
py2.5.egg','/usr/ali/lib/python25.zip','/usr/ali/lib/python2.5', '/usr/ali/lib/python2.5/plat-
linux2','/usr/ali/lib/python2.5/lib-tk','/usr/ali/lib/python2.5/lib-dynload', '/usr/ali/lib/python2.5/site-
packages']
```

快速入门

快速入门

在这一章里，你将学到如何用OSS Python SDK完成一些基本的操作。

Step-1.初始化一个OssAPI

SDK的OSS操作通过OssAPI类完成的，下面代码创建一个OssAPI对象:

```
from oss.oss_api import *
#以杭州为例
endpoint=” oss-cn-hangzhou.aliyuncs.com”
accessKeyId, accessKeySecret=” your id” ,” your secret”
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
```

在上面代码中，变量 accessKeyId 与 accessKeySecret 是由系统分配给用户的，称为ID对，用于标识用户，为访问OSS做签名验证。关于OssAPI类的详细介绍，参见OssAPI。

Step-2. 新建bucket

Bucket是OSS全局命名空间，相当于数据的容器，可以存储若干Object。您可以按照下面的代码新建一个Bucket：

```
from oss.oss_api import *
#以杭州为例
endpoint=” oss-cn-hangzhou.aliyuncs.com”
accessKeyId, accessKeySecret=” your id” ,” your secret”
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#设置bucket权限为私有
res = oss.create_bucket(bucket,"private")
print "%s\n%s" % (res.status, res.read())
```

调用create_bucket方法返回一个HTTP Response类。 关于Bucket的命名规范，参见Bucket中的【命名规范】。

Step-3. 上传Object

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一个Object的上传：

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
res = oss.put_object_from_file(bucket, object, "test.txt")
print "%s\n%s" % (res.status, res.getheaders())
```

关于Object的命名规范，参见Object中的命名规范 。关于上传Object更详细的信息，参见Object一节【上传Object】。

Step-4. 列出所有Object

当您完成一系列上传后，可能需要查看某个Bucket中有哪些Object，可以通过下面的程序实现：

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
res = oss.get_bucket("bucketname")
print "%s\n%s" % (res.status, res.read())
```

更多灵活的参数配置，可以参考Object中【列出Bucket中object】

Step-5. 获取指定Object

您可以参考下面的代码简单地实现一个Object的获取：

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#直接读取object到本地文件
res = oss.get_object_to_file(bucket, object, "/filepath/test.txt")
print "%s\n%s" % (res.status, res.getheaders())
```

可以从响应的头部读取object的meta信息。

OssAPI

OssAPI

OssAPI类中提供了一些操作bucket和object的方法。用户可以调用这些方法来操作OSS。 OssAPI中提供的接口大多直接返回HTTP Response。关于HTTP Response的定义请参见python官方文档中的说明。 示例中用res来表示一个HTTP Response的实例。res.status表示Python SDK向OSS发送HTTP请求后，OSS HTTP Server的响应状态码。具体各种状态码的定义见OSS API文档。res.getheaders()表示OSS HTTP Server响应的Headers。res.read()表示HTTP响应的Body，有些情况下Body是无内容的。

初始化OssAPI

```
from oss.oss_api import *
#以杭州为例
endpoint=” oss-cn-hangzhou.aliyuncs.com”
accessKeyId, accessKeySecret=” your id”,” your secret”
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
```

其中endpoint需填为待操作bucket所在region的对应值，有关endpoint的详细说明参考这里。

配置OssAPI

如果想对OssAPI的一些细节参数进行配置，可以通过下面的一些方法进行配置。

方法	描述	默认值
set_timeout	设置超时时间	10
set_debug	设置debug模式	True
set_retry_times	设置错误重试次数	3
set_send_buf_size	设置发送数据buffer的大小	8192
set_recv_buf_size	设置接收数据buffer的大小	1024*1024*10

Bucket

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。一个应用，例如图片分享网站，可以对应一个或多个Bucket。一个用户最多可创建10个Bucket，但每个bucket中存放的object的数量没有限制，存储容量每个buckte最高支持2PB。

命名规范

Bucket的命名有以下规范：

- 只能包括小写字母，数字，短横线（-）
- 必须以小写字母或者数字开头
- 长度必须在3-63字节之间

新建Bucket

```
from oss.oss_api import *
#以杭州为例
endpoint=" oss-cn-hangzhou.aliyuncs.com"
accessKeyId, accessKeySecret=" your id"," your secret"
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

res = oss.create_bucket("bucketname","private") #设置bucket权限为私有
print "%s\n%s" % (res.status, res.read())
```

创建bucket主要需要输入bucket名字与ACL。由于Bucket的名字是全局唯一的，所以尽量保证您的 bucket名字不与别人重复。而ACL目前取值为private，public-read， public-read-write中的一种，分别表示私有，公共读，公共读写。详细权限内容请参考OSS访问控制

列出用户所有的Bucket

可以通过get_service或者list_all_my_buckets两个方法实现，两个方法是等价的。

```
from oss.oss_api import *
from oss import oss_xml_handler
#以杭州为例
endpoint=" oss-cn-hangzhou.aliyuncs.com"
accessKeyId, accessKeySecret=" your id"," your secret"
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

res = oss.list_all_my_buckets()
buckets_xml=oss_xml_handler.GetServiceXml(res.read())
for bucket_info in buckets_xml.bucket_list:
    print "----------------------------------------"
    print "Location:"+bucket_info.location
    print "Name:"+bucket_info.name
```

```
print "CreationDate:"+bucket_info.creation_date
```

所有的bucket信息以xml格式存在于HTTP Response的Body中，通过oss_xml_handler中的GetServiceXml类来进行解析。

使用CNAME进行访问

当用户将自己的域名CNAME指向自己的一个bucket的域名后，用户可以使用自己的域名来访问OSS：

```
from oss.oss_api import *
#将endpoint填为自己绑定的域名
endpoint=" cname.com"
accessKeyId, accessKeySecret=" your id"," your secret"
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#上传内容到CNAME指向bucket
res=oss.put_object_from_string( "bucketname"," objectname"," conetnet" )
```

用户只需要在创建OssAPI类实例时，将原本填入该bucket的endpoint更换成CNAME后的域名即可。同时需要注意的是，使用该OssAPI实例的后续操作中，只能操作CNAME指向的bucket。

设置Bucket ACL

通过create_bucket或者put_bucket方法，如果该bucket存在并且属于请求发起者，那么请求中的ACL设置会覆盖原有的，从而达到设置Bucket ACL的目的。

```
from oss.oss_api import *
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

#设置该bucket已存在，设定该bucket acl为public-read
res = oss.create_bucket(bucket,"public-read")
print "%s\n%s" % (res.status, res.read())
```

ACL目前取值为private，public-read， public-read-write中的一种，分别表示私有，公共读，公共读写。详细权限内容请参考OSS访问控制

获取Bucket ACL

通过get_bucket_acl方法可以获得该Bucket的ACL

```
from oss.oss_api import *
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#获取bucket acl
res= oss.get_bucket_acl(bucket)
acl_xml=oss_xml_handler.GetBucketAclXml(res.read())
print acl_xml.grant
```

acl信息以xml格式存在HTTP Response的Body中，通过oss_xml_handler中的GetBucketAclXml类来进行解析需要解析xml才能得到想要的结果。

获取Bucket地址

通过get_bucket_location可以获得该Bucket的location

```
from oss.oss_api import *
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#获取bucket 地址
res= oss.get_bucket_acl(bucket)
location_xml=oss_xml_handler.GetBucketLocationXml(res.read())
print location_xml.location
```

Location信息以xml格式存在于HTTP Response的Body中，通过oss_xml_handler中的GetBucketLocationXml类进行解析。

删除Bucket

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

res = oss. delete_bucket(bucket)
print "%s\n%s" % (res.status, res.read())
```

需要注意的是，如果Bucket不为空（Bucket中有Object，或者有分块上传的碎片），则Bucket无法删除，必须删除Bucket中的所有Object以及碎片后，Bucket才能成功删除。

Object

Object

在OSS中，用户操作的基本数据单元是Object。单个Object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object大小不能超过48.8TB。Object包含key、meta和data。其中，key是Object的名字；meta是用户对该object的描述，由一系列name-value对组成；data是Object的数据。

命名规范

Object的命名规范如下：

- 使用UTF-8编码
- 长度必须在1-1023字节之间
- 不能以“/”或者“\”字符开头
- 不能含有“\r”或者“\n”的换行符

上传Object

简单上传

下面两种方式都可以实现object上传。

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#直接上传字符串
```

```
res=oss.put_object_from_string(bucket,object,"string content")
print "%s\n%s" % (res.status, res.read())


from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#上传本地文件
res=oss.put_object_from_file("bucketname","objectname","test.txt")
print "%s\n%s" % (res.status, res.read())
```

使用该方法上传最大文件不能超过5G。如果超过可以使用MultipartUpload上传。

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Object来存储。但给用户提供了创建模拟文件夹的方式，如下代码：

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#上传空字符，并让object以“/”结尾
res=oss.put_object_from_string("bucketname","folder_name/","")
print "%s\n%s" % (res.status, res.read())
```

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照样可以上传下载,只是控制台会对以"/"结尾的Object以文件夹的方式展示。所以用户可以使用上述方式来实现创建模拟文件夹。而对文件夹的访问可以参看下文中【文件夹模拟功能】

设定Object的Http Header

OSS服务允许用户自定义Object的Http Header。下面代码为Object设置了过期时间：

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#设置expiretime
header={ “Expires” :” Fri, 28 Feb 2012 05:38:42 GMT” }
#上传本地文件，携带header
res=oss.put_object_from_file(bucket,object,"test.txt",headers= header)
print "%s\n%s" % (res.status, res.read())
```

通过传入dic类型的参数headers，来设定上传的头部。

可以设置的Http Header有：Cache-Control、Content-Disposition、Content-Encoding、Expires、Content-MD5。它们的相关介绍见RFC2616。建议用户上传时携带Content-MD5值，OSS会计算body的Content-MD5并检查一致性。

设置User Meta

OSS支持用户自定义Meta信息对Object进行描述。比如：

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#设置user-meta
headers={ “x-oss-meta-user” :” baymax” }
#上传本地文件
```

```
res=oss.put_object_from_file(bucket, object,"test.txt",headers= headers)
print "%s\n%s" % (res.status, res.read())
```

user meta是以“x-oss-meta”为前缀的头信息，与上传http header一样，通过dic类型的headers参数传入。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2KB。

> NOTE：user meta的名称大小写不敏感，比如用户上传object时，定义名字为“x-oss-meta-USER”的meta，在表头中存储的参数为：“x-oss-meta-user”。

追加上传

OSS允许用户通过追加上传（Append Object）的方式在一个Object后面直接追加内容。前提是该Object类型为Appendable。通过Append Object操作创建的Object类型为Appendable Object，而通过Put Object方式上传是Normal Object。

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

#创建一个appendable object
position = 0
res = oss.append_object_from_file(bucket, object, position, localfile)

#通过Head Object来获取追加上传的position
res = oss.head_object(bucket, object)
if res.status / 100 == 2:
    header_map = convert_header2map(res.getheaders())
    if header_map.has_key('x-oss-next-append-position'):
        position = header_map['x-oss-next-append-position']

res = oss.append_object_from_file(bucket, object, position, localfile)

#通过追加上传的返回值获取下次追加position
if res.status / 100 == 2:
    header_map = convert_header2map(res.getheaders())
    if header_map.has_key('x-oss-next-append-position'):
        position = header_map['x-oss-next-append-position']
res = oss.append_object_from_file(bucket, object, position, localfile)
```

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。（应该设置的position信息存放在x-oss-next-append-position头中）

> 对于Object meta的设置只在使用追加上传创建Object时设置。后续如果需要更改该Object meta，可以通过下面要讲到的copy object接口实现（源和目的为同一个Object）。

分块上传

OSS允许用户将一个Object分成多个请求上传到后台服务器中，关于分块上传的内容，将在MultipartUpload

中Object的分块上传这一章中做介绍。

列出Bucket中的Object

列出Object

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

#列出bucket内object
res= oss.list_bucket(bucket)
objects_xml=oss_xml_handler.GetBucketXml(res.read())
print objects_xml.show()
```

Object信息以xml格式存在于HTTP Response的Body中，需要通过oss_xml_handler.GetBucketXml类来解析结果。

拓展参数

我们可以使用更多参数完成更多复杂的功能。比如：

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

#列出bucket中以”pic”为前缀，以“/”为结尾的共同前缀CommonPrefixes
res= oss.list_bucket(bucket,prefix=”pic”,delimiter=”/”)
objects_xml=oss_xml_handler.GetBucketXml(res.read())
print objects_xml.show()
```

上述代码列出了bucket中以”pic”为前缀，以“/”为结尾的共同前缀CommonPrefixes。比如“pic-people/”。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。

文件夹功能模拟

我们可以通过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Bucket内文件

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

#列出bucket内object
res= oss.list_bucket(bucket)
objects_xml=oss_xml_handler.GetBucketXml(res.read())
#格式化输出结果
print "Objects:"
for object_info in objects_xml.content_list:
    print object_info.key

print "CommonPrefixs:"
for prefix in objects_xml.prefix_list:
    print prefix


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

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)

#列出bucket中”fun/”目录下所有文件
res= oss.list_bucket(“bucketname”,prefix=”fun/”)
objects_xml=oss_xml_handler.GetBucketXml(res.read())
#格式化输出结果
print "Objects:"
for object_info in objects_xml.content_list:
    print object_info.key

print "CommonPrefixs:"
for prefix in objects_xml.prefix_list:
    print prefix


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

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#列出bucket中“fun/”目录下文件以及子目录
res= oss.list_bucket(“bucketname”,prefix=”fun/”,delimiter=”/”)
objects_xml=oss_xml_handler.GetBucketXml(res.read())
#格式化输出结果
print "Objects:"
for object_info in objects_xml.content_list:
    print object_info.key

print "CommonPrefixs:"
for prefix in objects_xml.prefix_list:
    print prefix


Objects:
fun/test.jpg

CommonPrefixs:
fun/movie/
```

返回的结果中， Objects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。

获取Object

读取Object

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#读取文件
res=oss.get_object("bucketname","object")
print "%s\n%s\n%s " % (res.status, res.read(),res.getheaders())
```

Object内容在返回的HTTP Response中body内，可以通过res.read()方法获取。Object的头信息可以通过res.getheaders()获得，包括ETag，Http Header以及自定义的元信息。 同时也可以在请求头加入下列http头来实现更细致地操作。

参数	说明
Range	指定文件传输的范围。
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。 否则抛出304 Not Modified异常。
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。否则抛出412 precondition failed异常
If-Match	传入一组ETag，如果传入期望的ETag和object的ETag匹配， 则正常传输文件。否则抛出412 precondition failed异常

If-None-Match	传入一组ETag，如果传入的ETag值和Object的ETag不匹配，则 正常传输文件。否则抛出304 Not Modified异常。

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#设置请求header，读取20-100字节内容
headers={"Range":"bytes=20-100"}
res=oss.get_object("bucketname","object",headers)
print "%s\n%s\n%s" % (res.status, res.read(),res.getheaders())
```

我们通过设置Range 方法并根据返回的Object的范围，可以用此功能实现文件的分段下载和断点续传。

直接下载Object到文件

我们可以通过下面的代码直接将Object下载到指定文件：

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#下载object到“file_path”
res=oss.get_object_to_file("bucketname","object","file_path")
print "%s\n%s\n%s" % (res.status, res.read(),res.getheaders())
```

只获取ObjectMetadata

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#获取object头信息
res=oss.head_object("bucketname","object")
print "%s\n%s" % (res.status,res.getheaders())
```

通过head_object可以获得object的头信息，即包含objectmeta。

删除Object

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#删除object
res=oss.delete_object("bucketname","object")
print "%s\n%s" % (res.status,res.read())


from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#批量删除3个object
objectlist=[ “object1”,” object2”,” object3”,]
res=oss.batch_delete_objects ("bucketname", objectlist)
print "Is success?"
print res
```

其中batch_delete_objects()返回的是bool值，表示删除是否成功。

拷贝Object

在同一个region中，用户可以对有操作权限的object进行复制操作。

拷贝一个Object

通过 copyObject 方法我们可以拷贝一个Object，代码如下：

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#拷贝object
res=oss. copy_object(“source_bucket”,” source_object”,” target_bucket”,” target_object”):
print "%s\n%s" % (res.status,res.getheaders())
```

需要注意的是源bucket与目的bucket必须属于同一region。

同时允许用户修改目的Object的ObjectMeta ，也能对通过携带下列头部完成更细致的操作。

请求头	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）。 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）。 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。 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）。 默认值：无
x-oss-metadata-directive	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。 默认值：COPY 有效值：COPY、REPLACE

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#如果源etag与提供etag相等，执行拷贝操作
headers={“x-oss-copy-source-if-match”:” 5B3C1A2E053D763E1B002CC607C5A0FE”}
res=oss. copy_object(“source_bucket”,” source_object”,” target_bucket”,” target_object”,headers):
print "%s\n%s" % (res.status,res.getheaders())
```

修改Object Meta

可以通过拷贝操作来实现修改已有 Object 的 meta 信息。如果拷贝操作的源Object地址和目标Object地址相同，都会直接替换源Object的meta信息。

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#设置object的Content-Type
headers={ “Content-Type” :” image/japeg” }
res=oss. copy_object( “bucketname” ,” object” ,” bucketname” ,” object” ,headers)
print "%s\n%s" % (res.status,res.getheaders())
```

MultipartUpload

Multipart Upload

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。

下面我们将一步步介绍怎样实现Multipart Upload。

分步完成Multipart Upload

初始化

初始化一个分块上传事件

```
from oss.oss_api import *
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#初始化一个分块上传事件
res=oss.init_multi_upload("bucketname","mutipartobject")
body = res.read()
if res.status == 200:
    #使用GetInitUploadIdXml解析xml，获得upload id
    h = oss_xml_handler.GetInitUploadIdXml(body)
    upload_id = h.upload_id
else:
    err = ErrorXml(body)
    raise Exception("%s, %s" %(res.status, err.msg))
```

Upload id以xml格式保存在返回的HTTP Response中body内，使用提供的解析方法就能获得upload id。UploadId ，它是区分分块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

接着，把本地文件分块上传。 假设有一个文件，本地路径为 /path/to/file.zip 由于文件比较大，将其分块传输到OSS中。

```
from oss.oss_api import *
from oss import oss_util
from oss import oss_xml_handler

filename=" /path/to/file.zip"
max_part_num=20
filename = oss_util.convert_utf8(filename)
#将文件进行切分，每块信息保存在part_msg_list中
part_msg_list = oss_util.split_large_file(filename, object, max_part_num)
uploaded_part_map = oss_util.get_part_map(oss, bucket, object, upload_id)
for part in part_msg_list:
#提取每块信息
   part_number = str(part[0])
   partsize = part[3]
   offset = part[4]
   res = oss.upload_part_from_file_given_pos(bucket, object, filename, offset, partsize, upload_id, part_number)
#保存etag
   etag = res.getheader("etag")
   if etag:
      uploaded_part_map[part_number] = etag
```

上面程序的核心是调用 uploadPart 方法来上传每一个分块，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。
- 为了保证数据在网络传输过程中不出现错误，强烈推荐用户在上传part时携带meta：content-md5，在OSS收到数据后，用该MD5值验证上传数据的正确性，如果出现不一致会返回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。

完成分块上传

```
# 将oss_util.split_large_file()生成的part_msg_list转化成xml格式
part_msg_xml = oss_util .create_part_xml(part_msg_list)
res = oss.complete_upload(bucket,objectname, upload_id, part_msg_xml,)
```

取消分块上传事件

```
from oss.oss_api import *
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#使用upload id取消分块上传事件
oss.cancel_upload(bucketname，objectname, upload_id)
```

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。

获取Bucket内所有分块上传事件

```
from oss.oss_api import *
from oss import oss_util
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
```

```
#获得所有分块上传事件信息
res=oss.get_all_multipart_uploads(bucket)
#解析结果
res_xml= oss_xml_handler. GetMultipartUploadsXml(res.read())
for upload in res_xml.content_list
    print "----------------------------------------"
    print "key:" +upload.key
    print "uploaded:" +upload.upload_id
```

所有上传事件信息以xml格式存储在返回的http response的body中，可以通过GetMultipartUploadsXml这个类进行解析。

获取所有已上传的块信息

```
from oss.oss_api import *
from oss import oss_util
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#获得该分块上传事件所有的分块信息

res=oss.get_all_parts(bucket,object,upload_id)
#解析结果
res_xml= oss_xml_handler. GetPartsXml (res.read())
for part in res_xml.content_list
    print "----------------------------------------"
    print "partnumber:" +part. part_number
    print "lastmodified:" + part. last_modified
    print "etag:" + part. etag
    print "size:" + part. size
```

所有分块信息以xml格式存储在返回的http response的body中，可以通过GetPartsXml这个类进行解析。

跨域资源共享

跨域资源共享（CORS）

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put_cors()方法将指定的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。具体的规则主要通过上传一个xml格式字符串来进行参数设置。代码如下：

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#传入设置cors规则xml字符串
res=oss.put_cors(bucket,cors_xml)
print res.status()
```

关于设置cors的xml格式可以参考OSS API文档。

获取CORS规则

我们可以参考bucket的CORS规则，通过get_cors()这个方法获取。代码如下：

```
from oss.oss_api import *
from oss import oss_xml_handler

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#获取cors规则信息
res=oss.get_cors(bucket)
通过提供的CorsXml类来进行解析
cors_xml=oss_xml_handler.CorsXml(res.read())
for rule in cors_xml.rule_list:
    print "----------------"
    rule.show()
```

读取的CORS规则信息以xml格式保存在响应的body里面，通过read()方法读取出来，用提供的CorsXml类来进行解析。

删除CORS规则

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#清空bucket设置的cors规则
res=oss.delete_cors(bucket)
print res.status()
```

用于关闭指定Bucket对应的CORS并清空所有规则。

生命周期管理

生命周期管理（lifecycle）

OSS提供Object生命周期管理来为用户管理对象。用户可以为某个Bucket定义生命周期配置，来为该Bucket的Object定义各种规则。目前，用户可以通过规则来删除相匹配的Object。每条规则都由如下几个部分组成：

- Object名称前缀，只有匹配该前缀的Object才适用这个规则
- 操作，用户希望对匹配的Object所执行的操作。
- 日期或天数，用户期望在特定日期或者是在Object最后修改时间后多少天执行指定的操作。

设置Lifecycle

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>
  <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object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object，Date则表示到指定的绝对时间之后就删除object(绝对时间服从ISO8601的格式)。

通过put_lifecycle()方法将指定的bucket上设定一个lifecycle规则，如果原规则存在则覆盖原规则。具体的规则主要通过上传一个xml格式字符串来进行参数设置。代码如下：

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#传入设置lifecycle规则xml字符串
res=oss.put_lifecycle(bucket,lifecycle_xml)
print res.status()
```

关于设置lifecycle的xml格式可以参考api文档。

获取Lifecycle规则

我们可以参考bucket的lifecycle规则，通过get_ lifecycle ()这个方法获取。代码如下：

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#获取cors规则信息
res=oss.get_lifecycle(bucket)
#以xml格式保存在响应的body里
print res.read()
```

读取的lifecycle规则信息以xml格式保存在响应的body里面，通过read()方法读取出来。

删除Lifecycle规则

用于关闭指定Bucket对应的lifecycle并清空所有规则。

```
from oss.oss_api import *

oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#清空bucket设置的lifecycle规则
res=oss.delete_lifecycle (bucket)
print res.status()
```

错误响应

错误响应

错误响应处理

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。当返回状态为非2XX时，错误信息以xml格式保存在响应的body里面。可以参考下面代码的处理：

```
from oss.oss_api import *
from oss import oss_xml_handler
oss = OssAPI(endpoint, accessKeyId, accessKeySecret)
#以获得object列表为例
res= oss.list_bucket(bucket)
if res.status/100 !=  2:
    #解析xml格式的错误信息
    err_msg=oss_xml_handler.ErrorXml(res.read())
    print 'Code:'+err_msg.code
    print 'Message:'+err_msg.msg
    print 'Request id:'+err_msg.request_id
else :
    objects_xml=oss_xml_handler.GetBucketXml(res.read())
    objects_xml.show()
```

其中:

- 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。

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
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

Android-SDK

前言

前言

SDK下载

- Android SDK开发包(2015-12-03) 版本号 2.0.1：aliyun_OSS_Android_SDK_20151206

注：2.0.1版本为大版本更新，不向前兼容1.4.0版本及以下。

- github地址：点击查看
- sample地址：点击查看

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。

版本迭代详情参考**点击查看**

简介

本文档主要介绍OSS Android SDK的安装和使用，针对于Android SDK版本2.0.1。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。

安装

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-2.0.1.jar、okhttp-2.6.0.jar、okio-1.6.0.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>2.0.1</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.READ_PHONE_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>
```

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需要传入回调函数，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
```

考虑到简洁性，本文档中只有部分重要接口会列出同步、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，实际上两种方式都是支持的，用法大同小异，不再赘述。

初始化

初始化设置

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服务在各个区域的域名地址，目前支持两种形式

Endpoint类型	解释
OSS域名	在官方网站可以查询的内外网Endpoint
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS域名 在官方网站可以查询到的内外网Endpoint：

- Endpoint和部署区域有关系，所以不同Bucket可能拥有不同的Endpoint。
- 您可以登陆 阿里云OSS控制台，点击特定bucket，比如bucket-1。
- 在<Bucket概览>-<OSS域名>区域可以看到两个域名，一个是OSS外网域名：bucket-1.oss-cn-hangzhou.aliyuncs.com， 一个是内网域名：bucket-1.oss-cn-hangzhou-internal.aliyuncs.com。
- 那么此bucket的外网Endpoint就是oss-cn-hangzhou.aliyuncs.com，内网Endpoint是oss-cn-hangzhou-internal.aliyuncs.com。

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);
```

更多鉴权方式参考：访问控制

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);
```

更多鉴权方式参考：访问控制

设置网络参数

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);
```

快速入门

快速入门

以下演示了上传、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。鉴权细节详见后面的访问控制章节。

```
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后面的`访问控制`章节
OSSCredentialProvider credentialProvider = new OSSPlainTextAKSKCredentialProvider("<accessKeyId>", "<accessKeySecret>");

OSS oss = new OSSClient(getApplicationContext(), endpoint, credentialProvider);
```

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>", "<uploadFilePath>");

// 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<PutObjectRequest, PutObjectResult>() {
    @Override
    public void onSuccess(PutObjectRequest request, PutObjectResult result) {
        Log.d("PutObject", "UploadSuccess");
    }

    @Override
    public void onFailure(PutObjectRequest 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(); // 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.asyncGetObejct(get, new OSSCompletedCallback<GetObjectRequest, 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 serviceExceptio
n) {
        // 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
```

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);
```

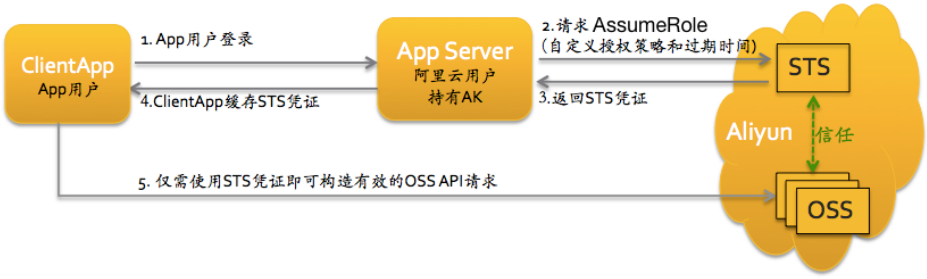
STS鉴权模式

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。

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

使用这种模式授权需要先开通阿里云RAM服务:RAM

STS使用手册：https://docs.aliyun.com/#/pub/ram/sts-sdk/sts_java_sdk&preface

OSS授权策略配置：https://docs.aliyun.com/#/pub/oss/product-documentation/acl&policy-configure

使用

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，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，如图示：

```
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+oJl3XSlbJ7StKpQ...."}
```

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_CHARSET_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。

签名算法参考：http://help.aliyun.com/document_detail/oss/api-reference/access-control/signature-header.html

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);
```

上传文件

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>", "<uploadFilePath>");

// 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());
}
```

注意，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>", "<uploadFilePath>");

// 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<PutObjectRequest, 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, ServiceException serviceExceptio
n) {
        // 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());
}
```

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, uploadFilePath);

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) {
        // 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());
        }
    }
```

```
});
```

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。

具体说明参考：Callback

代码示例：

```
PutObjectRequest put = new PutObjectRequest(testBucket, testObject, uploadFilePath);

ObjectMetadata metadata = new ObjectMetadata();
metadata.setContentType("application/octet-stream");

put.setMetadata(metadata);

put.setCallbackParam(new HashMap<String, String>() {
    {
        put("callbackUrl", "110.75.82.106/mbaas/callback");
        put("callbackBody", "test");
    }
});

// put.setCallbackVars(new HashMap<String, String>() {
//     {
//         put("x:var1", "value1");
//         put("x:var2", "value2");
//     }
// });


// 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<PutObjectRequest, 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, ServiceException serviceExceptio
n) {
        // 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());
        }
    }
});
```

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。

特别注意： 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>", "<uploadFil
ePath>");
// 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 OSSCompletedCallback<ResumableUplo
adRequest, ResumableUploadResult>() {
    @Override
    public void onSuccess(ResumableUploadRequest request, ResumableUploadResult result) {
        Log.d("resumableUpload", "success!");
    }
```

```
    @Override
    public void onFailure(ResumableUploadRequest request, ClientException clientExcepion, ServiceException service
Exception) {
        // 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());
        }
    }
});

// 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>", "<uploadFil
ePath>", 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 OSSCompletedCallback<ResumableUplo
adRequest, ResumableUploadResult>() {
    @Override
    public void onSuccess(ResumableUploadRequest request, ResumableUploadResult result) {
        Log.d("resumableUpload", "success!");
    }

    @Override
    public void onFailure(ResumableUploadRequest request, ClientException clientExcepion, ServiceException service
Exception) {
        // 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());
        }
    }
});

// resumableTask.waitUntilFinished();
```

下载文件

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，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.asyncGetObejct(get, new OSSCompletedCallback<GetObjectRequest, 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, ServiceException serviceExceptio
n) {
        // 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());
        }
    }
});
```

指定范围下载

您可以在下载文件时指定一段范围，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.asyncGetObejct(get, new OSSCompletedCallback<GetObjectRequest, 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, ServiceException serviceExceptio
n) {
        // 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<HeadObjectRequest, HeadObjectRe
sult>() {
    @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 serviceExcept
ion) {
        // 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();
```

授权访问

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>");
```

分片上传

分片上传

下面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, currentIn
dex);
    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>", "<objectK
ey>", 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。

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。

管理文件

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 serviceExcepti
on) {
        // 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 OSSCompletedCallback<CopyObjectReq
uest, CopyObjectResult>() {
    @Override
    public void onSuccess(CopyObjectRequest request, CopyObjectResult result) {
        Log.d("copyObject", "copy success!");
    }

    @Override
    public void onFailure(CopyObjectRequest request, ClientException clientExcepion, ServiceException serviceExcept
ion) {
        // 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());
        }
    }
});
```

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, Dele
teObjectResult>() {
    @Override
    public void onSuccess(DeleteObjectRequest request, DeleteObjectResult result) {
        Log.d("asyncCopyAndDelObject", "success!");
    }

    @Override
```

```
    public void onFailure(DeleteObjectRequest request, ClientException clientExcepion, ServiceException serviceExce
ption) {
        // 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<HeadObjectRequest, HeadObjectRe
sult>() {
    @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 serviceExcept
ion) {
        // 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();
```

异常响应

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
SignatureDoesNotMatch	签名错误
TooManyBuckets	用户的Bucket数目超过限制

iOS-SDK

前言

前言

SDK下载

- iOS SDK开发包(2015-11-20) 版本号 2.1.1：aliyun_OSS_iOS_SDK_20151120.zip
- github地址：https://github.com/aliyun/aliyun-oss-ios-sdk
- pod依赖： pod 'AliyunOSSiOS', '~> 2.1.1'
- demo地址： https://github.com/alibaba/alicloud-ios-demo

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。

版本迭代详情参考这里

简介

本文档主要介绍OSS iOS SDK的安装和使用，针对于iOS SDK版本2.1.1。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。

安装

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，推荐参考官方文档: 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>
```

注意，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。

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];

...
```

初始化

初始化设置

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服务在各个区域的域名地址，目前支持两种形式

Endpoint类型	解释
OSS域名	在官方网站可以查询的内外网Endpoint
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS域名 在官方网站可以查询到的内外网Endpoint：

- Endpoint和部署区域有关系，所以不同Bucket可能拥有不同的Endpoint。
- 您可以登陆 阿里云OSS控制台，点击特定bucket，比如bucket-1。
- 在<Bucket概览>-<OSS域名>区域可以看到两个域名，一个是OSS外网域名：bucket-1.oss-cn-hangzhou.aliyuncs.com， 一个是内网域名：bucket-1.oss-cn-hangzhou-internal.aliyuncs.com。
- 那么此bucket的外网Endpoint就是oss-cn-hangzhou.aliyuncs.com，内网Endpoint是oss-cn-hangzhou-internal.aliyuncs.com。

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];
```

更多鉴权方式参考：访问控制

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];
```

更多鉴权方式参考：访问控制

设置网络参数

也可以在初始化的时候设置详细的ClientConfiguration：

```
NSString *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
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 clientConfiguration:conf];
```

快速入门

快速入门

以下演示了上传、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。鉴权细节详见后面的访问控制章节。

```
NSString *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
id<OSSCredentialProvider> credential = [[OSSPlainTextAKSKPairCredentialProvider alloc] initWithPlainTextAccessKey:@"<your accessKeyId>"
                                                                                                          secretKey:@"<your accessKeySecret>"];

client = [[OSSClient alloc] initWithEndpoint:endpoint credentialProvider:credential];
```

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 totalBytesExpectedToSend) {
    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 totalBytesExpectedToWrite) {
    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];
```

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

```
// 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];
```

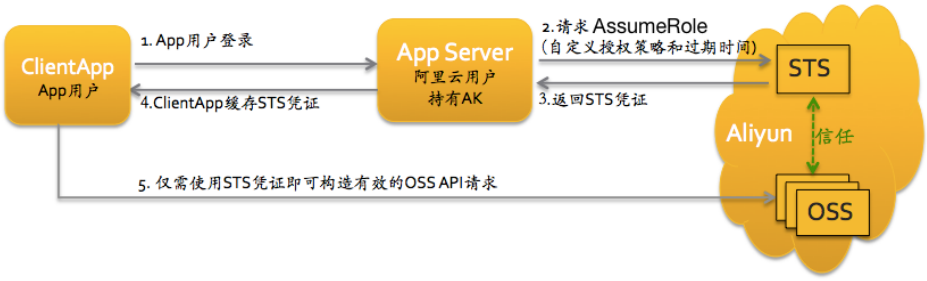
STS鉴权模式

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：



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。

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

使用这种模式授权需要先开通阿里云RAM服务:RAM

STS使用手册：https://docs.aliyun.com/#/pub/ram/sts-sdk/sts_java_sdk&preface

OSS授权策略配置：https://docs.aliyun.com/#/pub/oss/product-documentation/acl&policy-configure

使用

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，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，如图示：

```
id<OSSCredentialProvider> credential = [[OSSFederationCredentialProvider alloc] initWithFederationTokenGetter:^
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+oJl3XSlbJ7StKpQ.... "}
```

那么，您可以这么实现一个OSSFederationCredentialProvider实例：

```
id<OSSCredentialProvider> credential2 = [[OSSFederationCredentialProvider alloc] initWithFederationTokenGetter:
^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];
```

上传文件

上传Object

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 totalBytesExpectedToSend) {
    // 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];
```

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 totalBytesExpectedToSend) {
    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。

具体说明参考：Callback

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 totalBytesExpectedToSend) {
    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;
}];
```

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。

这个功能依赖OSS的分块上传接口实现，它不会在本地保存任何信息。在上传大文件前，您需要调用分块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的获取方式参考后面的 分块上传 章节。

特别注意： 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块上传实现的，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 == OSSClientErrorCodeCann
otResumeUpload) {
            // 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];
```

下载文件

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];
// required
request.bucketName = @"<bucketName>";
request.objectKey = @"<objectKey>";
```

```
//optional
request.downloadProgress = ^(int64_t bytesWritten, int64_t totalBytesWritten, int64_t totalBytesExpectedToWrite) {
    // 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];
```

指定范围下载

您可以在下载文件时指定一段范围，如：

```
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 totalBytesExpectedToWrite) {
    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;
}];
```

授权访问

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);
}
```

分块上传

分块上传

下面演示通过分块上传文件的整个流程：

初始化分块上传

```
__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;
}
```

上传分块

```
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.ab
soluteString error:nil] fileSize];
        [partInfos addObject:[OSSPartInfo partInfoWithPartNum:i eTag:result.eTag size:fileSize]];
    } else {
        NSLog(@"upload part error: %@", uploadPartTask.error);
        return;
    }
}
```

完成分块上传

```
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];
```

删除分块上传事件

```
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;
}
```

罗列分块

```
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;
}];
```

管理文件

管理Object

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;
}];
```

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
}];
```

删除Object

```
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

```
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;
}];
```

管理Bucket

Bucket管理

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;
}];
```

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;
}];
```

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;
}];
```

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;
}];
```

兼容旧版本

兼容旧版本

当前版本SDK对旧版本SDK进行了完全的重构，变成RESTful风格的调用方式，逻辑更清晰，也贴合OSS的其他SDK使用方式。对于已经使用了旧版本对象存取风格的用户，旧有接口将完全不再提供，建议迁移到新版本SDK的用法。但同时SDK也提供了一些妥协的接口，便于迁移。

需要额外引入:

```
#import <AliyunOSSiOS/OSSCompat.h>
```

上传文件

```
NSDictionary * objectMeta = @{@"x-oss-meta-name1": @"value1"};
OSSTaskHandler * tk = [client uploadFile:@<filepath>"
                      withContentType:@"application/octet-stream"
                       withObjectMeta:objectMeta
                        toBucketName:@"<bucketName>"
                         toObjectKey:@"<objectKey>"
                         onCompleted:^(BOOL isSuccess, NSError * error) {
                              if (error) {
                                  NSLog(@"upload object failed, erorr: %@", error);
                              } else {
                                  NSLog(@"upload object success!");
                              }
                        } onProgress:^(float progress) {
                            NSLog(@"progress: %f", progress);
                        }];

// [tk cancel];
```

或者从NSData * 上传:

```
[client uploadData:(NSData *)
  withContentType:@"application/octect-stream"
   withObjectMeta:nil
     toBucketName:@"<bucketName>"
      toObjectKey:@"<objectKey>"
      onCompleted:^(BOOL isSuccess, NSError * error) {
          // code
      } onProgress:^(float progress) {
          // code
      }];
```

下载文件

```
OSSTaskHandler * tk = [client downloadToDataFromBucket:@"<bucketName>"
                                             objectKey:@"<objectkey>"
                                           onCompleted:^(NSData * data, NSError * error) {
                                               if (error) {
                                                   NSLog(@"download object failed, erorr: %@", error);
                                               } else {
                                                   NSLog(@"download object success, data length: %ld", [data length]);
                                               }
                                           } onProgress:^(float progress) {
                                               NSLog(@"progress: %f", progress);
                                           }];

// [tk cancel];
```

或者直接下载到文件:

```
[client downloadToFileFromBucket:@"<bucketName>"
                       objectKey:@"<objectKey>"
                          toFile:@"<filepath>"
                     onCompleted:^(BOOL isSuccess, NSError * error) {
                         // code
                     } onProgress:^(float progress) {
                         // code
                     }];
```

断点上传

如果一次任务失败或者被取消，下次调用这个接口，只要文件、bucketName和objectKey保持和之前一致，上传就会继续从上次失败的地方开始：

```
OSSTaskHandler * tk = [client resumableUploadFile:@"<filepath>"
                                  withContentType:@"application/octet-stream"
                                   withObjectMeta:nil
                                     toBucketName:@"<bucketName>"
                                      toObjectKey:@"<objectKey>"
                                      onCompleted:^(BOOL isSuccess, NSError * error) {
                                          if (error) {
                                              NSLog(@"resumable upload object failed, erorr: %@", error);
                                          } else {
                                              NSLog(@"resumable upload object success!");
                                          }
```

```
                    } onProgress:^(float progress) {
                        NSLog(@"progress: %f", progress);
                    }];
//[tk cancel];
```

这个接口是为了兼容旧版本提供的，封装了较多细节，现在，更建议您通过分块上传的multipartUploadInit/uploadPart/listParts/completeMultipartUpload/abortMultipartUpload这几个接口，来实现您的断点续传。

异常响应

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
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

.NET-SDK

前言

前言

简介

- 此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2.1.0。
- 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Access Key管理创建Access Key。

兼容性

对于2.0.× 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：Aliyun.OpenServices.OpenStorageService变更为Aliyun.OSS

版本

当前最新版本：2.1.0

新增内容

- .NET Framework 2.0和.NET Framework 3.5支持
- 大文件拷贝接口：CopyBigObject
- 大文件上传接口：UploadBigObject
- 文件meta修改接口：ModifyObjectMeta

修改内容

- 提升SDK健壮性
- ContentType支持大多数MIME种类（226种）
- 补齐SDK API速查文档中的缺失注释
- 删除某些类中已经废弃的ObjectMetaData属性，请使用类中对应的的ObjectMetadata属性

安装

SDK安装

版本依赖

- 适用于.NET 2.0 及以上版本
- 适用于Visual Studio 2010及以上版本

版本迭代

- 版本迭代详情参考这里

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，成功后添加到项目引用中。

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

- 从这里下载SDK包，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。

注意：

- 上面的四种安装方式，选择一种安装即可。

初始化

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域名	在官方网站可以查询的内外网Endpoint
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS域名

在官方网站可以查询到的内外网Endpoint：

- Endpoint和部署区域有关系，所以不同Bucket可能拥有不同的Endpoint。
- 您可以登陆 阿里云OSS控制台，点击特定bucket，比如bucket-1。
- 在<Bucket概览>-<OSS域名>区域可以看到两个域名，一个是OSS外网域名：bucket-1.oss-cn-hangzhou.aliyuncs.com， 一个是内网域名：bucket-1.oss-cn-hangzhou-internal.aliyuncs.com。
- 那么此bucket的外网Endpoint就是oss-cn-hangzhou.aliyuncs.com，内网Endpoint是oss-cn-hangzhou-internal.aliyuncs.com。

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：

- 注册阿里云帐号
- 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);
```

注意：

- 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
```

快速入门

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);
}
```

提示：

- 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 。
- 关于OssClient的详细介绍，参见 初始化。

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);
    }
}
```

提示：

- 关于存储空间的命名规范，参见管理Bucket中的命名规范。

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);
    }
}
```

注意：

- 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。
- 关于文件的命名规范和其他更详细的信息，参见上传文件 。

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);
    }
}
```

提示：

- 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, _accessKeySecret);

        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);
            }
        }
    }
}
```

管理Bucket

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；

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);
    }
}
```

提示：

- 完整代码路径:samples/Samples/CreateBucketSample.cs

重要：

- 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.O
wner);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("List bucket failed. {0}", ex.Message);
    }
}
```

提示：

- 完整代码路径:samples/Samples/ListBucketSample.cs

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);
    }
}
```

提示：

- 完整代码路径:samples/Samples/CNameSample.cs

提示：

- 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);
    }
}
```

提示：

- 完整代码路径:samples/Samples/DoseBucketExistSample.cs

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);
    }
}
```

提示：

- 完整代码路径:samples/Samples/SetBucketAclSample.cs

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);
    }
}
```

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);
    }
}
```

提示：

- 完整代码路径:samples/Samples/CreateBucketSample.cs

重要：

- 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。

上传文件

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);
}
```

提示：

- 完整代码路径：samples/Samples/PutObjectSample.cs

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);
}
```

提示：

- 完整代码路径：samples/Samples/PutObjectSample.cs

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 eTag;
    using (var fs = File.Open(fileToUpload, FileMode.Open))
    {
        eTag = OssUtils.ComputeContentMd5(fs, fs.Length);
    }

    var objectMeta = new ObjectMetadata {ETag = eTag};
    client.PutObject(bucketName, key, fileToUpload, objectMeta);
    Console.WriteLine("Put object succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Put object failed, {0}", ex.Message);
}
```

提示：

- 完整代码路径：samples/Samples/GetObjectSample.cs

注意：

- 为了保证SDK发送的数据和OSS服务端接收到的数据一样，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);
}
```

提示：

- 完整代码路径：samples/Samples/PutObjectSample.cs
- .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);
}
```

提示：

- 完整代码路径：samples/Samples/PutObjectSample.cs
- 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。

重要：

- 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”

注意：

- 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。

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：

```
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);
}
```

提示：

- 完整代码路径：samples/Samples/PutObjectSample.cs
- 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。
- 而对文件夹的访问可以参看文件夹功能模拟

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();
    }
```

```
}
```

提示：

- 完整代码路径：samples/Samples/PutObjectSample.cs

注意：

- 使用异步上传时您需要实现自己的回调处理函数

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);
}
```

提示：

- 完整代码路径：samples/Samples/MultipartUploadSample.cs
- 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);
}
```

提示：

- 完整代码路径：samples/Samples/MultipartUploadSample.cs

注意：

- 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);
}
```

提示：

- 完整代码路径：samples/Samples/MultipartUploadSample.cs

注意：

- 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);
}
```

提示：

- 完整代码路径：samples/Samples/MultipartUploadSample.cs

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);
}
```

提示：

- 完整代码路径：samples/Samples/MultipartUploadSample.cs

重要：

- 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， 反回 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);
}
```

提示：

- 完整代码路径：samples/Samples/MultipartUploadSample.cs
- 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。

下载文件

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);
    }
```

```
}
```

提示：

- 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);
    }
}
```

提示：

- 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。
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);
    }
}
```

管理文件

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);
    }
}
```

注意：

- 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);
        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);
    }
}
```

注意：

- 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);
    }
}
```

提示：可以设置的参数名称和作用

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。

注意：

- 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/
```

提示：

- 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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

注意：

- 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);
    }
}
```

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 uplo
adId)
{
    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);
}
```

提示：

- 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);
    }
}
```

注意：

- 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。

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);
    }
}
```

授权访问

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 = new DateTime().AddHours(1)
}
var uri = client.GeneratePresignedUri(req);
```

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：

```
// 生成PUT方法的URL
var req = new GeneratePresignedUriRequest(bucketName, key, SignHttpMethod.Put);
{
    Expiration = new DateTime().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);
```

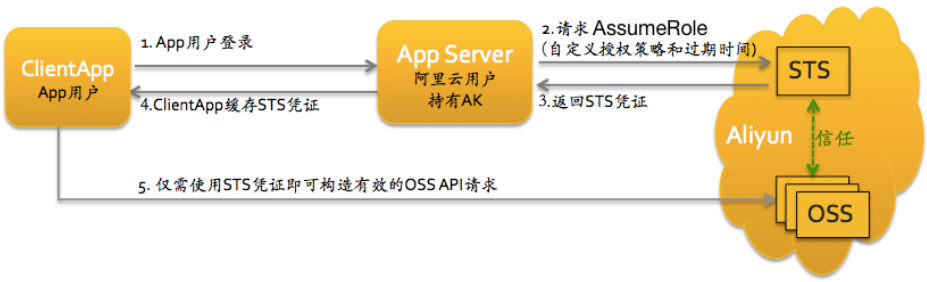
使用STS服务临时授权

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：



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);
```

生命周期管理

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Lifecycle

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>
  <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)。

可以通过下面的代码，设置上述lifecycle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);
```

注意

- 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。

可以通过下面的代码获取上述lifecycle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));
}
```

可以通过下面的代码清空存储空间中lifecycle规则。

```
using Aliyun.OSS;

// 初始化OssClient
var client = new OssClient(endpoint, accessKeyId, accessKeySecret);

var LifecycleRequest = new SetBucketLifecycleRequest(bucketName);
client.SetBucketLifecycle(LifecycleRequest);
```

跨域资源共享

跨域资源共享设置

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);
```

注意：

- 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);
}
```

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);
```

防盗链

防盗链设置

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);
```

注意：

- Referer参数支持通配符“*”和“？”，更多详细的规则配置可以参考用户手册OSS防盗链

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");
}
```

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);
```

异常

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
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

PHP-SDK

安装

SDK安装

- github地址：https://github.com/aliyun/aliyun-oss-php-sdk
- ChangeLog：https://github.com/aliyun/aliyun-oss-php-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。

旧版本文档

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。

安装

主要有三种途径获取SDK：

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。composer安装完成后，在您的PHP代 码中引入依赖即可：

```
require_once __DIR__ . '/vendor/autoload.php';
```

您也可以直接下载已经打包好的 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';
```

下载SDK包安装， aliyun-oss-php-sdk-2.0.1.zip ，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';
```

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）

提示：

- Ubuntu下可以使用apt-get包管理器安装php的cURL扩展 sudo apt-get install php5-curl

运行samples

1. 解压下载到的sdk包
2. 修改samples目录中的Config.php文件
    i. 修改 OSS_ACCESS_ID, 您从OSS获得的AccessKeyId
    ii. 修改 OSS_ACCESS_KEY, 您从OSS获得的AccessKeySecret
    iii. 修改 OSS_ENDPOINT, 您选定的OSS数据中心访问域名，如 oss-cn-hangzhou.aliyuncs.com
    iv. 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
3. 到samples目录中执行 php RunAll.php， 也可以单个运行某个Sample文件

初始化

初始化设置

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服务在各个区域的域名地址，目前支持两种形式

Endpoint类型	解释
OSS域名	在官方网站可以查询的内外网Endpoint
用户自定义域名	用户自定义域名，且CNAME指向OSS域名

OSS域名

在官方网站可以查询到的内外网Endpoint：

- Endpoint和部署区域有关系，所以不同Bucket可能拥有不同的Endpoint。
- 您可以登陆 阿里云OSS控制台，点击特定bucket，比如bucket-1。
- 在<Bucket概览>-<OSS域名>区域可以看到两个域名，一个是OSS外网域名：bucket-1.oss-cn-hangzhou.aliyuncs.com， 一个是内网域名：bucket-1.oss-cn-hangzhou-internal.aliyuncs.com。
- 那么此bucket的外网Endpoint就是oss-cn-hangzhou.aliyuncs.com，内网Endpoint是oss-cn-hangzhou-internal.aliyuncs.com。

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：

- 注册阿里云帐号
- 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
$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oss-cn-hangzhou.aliyuncs.com>";
try {
    $ossClient = new OssClient($accessKeyId, $accessKeySecret, $endpoint);
} catch (OssException $e) {
    print $e->getMessage();
}
```

OSS目前所有的节点列表见：OSS节点列表

使用自定义域名(CNAME)新建OssClient

```
<?php
$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);
} catch (OssException $e) {
```

```
    print $e->getMessage();
}
```

注意: **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);
$ossClient->setConnectTimeout(10);
```

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

快速入门

快速入门

在上一节初始化中，我们介绍了如何初始化OssClient客户端，这一章将使用这个客户端完成一些基本的操作。

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：

```
<?php
$bucket = "<您使用的存储空间名称，注意命名规范>";
try {
    $ossClient->createBucket($bucket);
} catch (OssException $e) {
    print $e->getMessage();
}
```

上传文件

文件是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
$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 {
    $ossClient->putObject($bucket, $object, $content);
} catch (OssException $e) {
    print $e->getMessage();
}
```

返回结果处理

OssClient提供的接口返回返回数据分为两种：

- Put，Delete类接口，接口返回null，如果没有OssException，即可认为操作成功
- Get，List类接口，接口返回对应的数据，如果没有OssException，即可认为操作成功

举个例子：

```
<?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

管理存储空间

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。

存储空间的命名有以下规范：

- 只能包括小写字母，数字，短横线（-）
- 必须以小写字母或者数字开头
- 长度必须在3-63字节之间

> 提示：
>
> - 以下场景的完整代码路径:samples/Bucket.php

新建存储空间

以下代码可以新建一个存储空间：

```php
<?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_PUBLIC_READ_WRITE);
    } catch (OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

重要：

- 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：

```php
<?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");
    }
}
```

设置存储空间访问权限(ACL)

设置存储空间访问权限可以使用以下代码：

```
<?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)

获取存储空间访问权限可以使用以下代码：

```
<?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);
}
```

删除存储空间

下面代码删除了一个存储空间

```
<?php
```

```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");
}
```

重要：

- 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后，存储空间才能成功删除。

上传文件

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, 使用分片上传方式文件大小不能超过48.8TB。

提示：

- SDK提供了多种上传方式，这里其中两种：简单上传和分片上传，sample代码的位置如下：
- 简单文件上传：samples/Object.php
- 分片文件上传: samples/MultipartUpload.php

上传指定字符串

下面的代码实现了上传指定字符串中的内容到文件中:

```php
<?php
/**
 * 简单上传,上传指定变量的内存值作为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__);
    $options = array(OssClient::OSS_HEADERS => array(
        'x-oss-meta-self-define-title2' => '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");
}
```

重要：

- user meta信息可以通过在OSS_HEADERS中增加x-oss-meta为前缀的key和相应的值设置
- user meta的key大小写不敏感，在表头中存储的参数为：x-oss-meta-name, 所以读取时读取名字为x-oss-meta-name的参数即可。

注意：

- 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

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");
}
```

重要：

- user meta信息可以通过在OSS_HEADERS中增加x-oss-meta为前缀的key和相应的值设置
- user meta的名称大小写不敏感，在表头中存储的参数为：'x-oss-meta-name',所以读取时读取名字为x-oss-meta-name的参数即可。

注意：

- 为了保证SDK发送的数据和OSS服务端接收到的数据一样，可以在$options中增加OssClient::OSS_CHECK_MD5 => true，这样服务端就会使用Md5值做校验
- 使用Md5校验时，性能会有所损失

注意：

- 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

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。

通过分片上传上传本地文件

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__;
    $options = array();
    try{
        $ossClient->multiuploadFile($bucket, $object, $file, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

通过分片上传上传本地目录

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");
}
```

通过原始接口执行分片上传操作（灵活）

分片上传(MultipartUpload)一般的流程如下:

1. 初始化一个分片上传任务（InitiateMultipartUpload）
2. 逐个或并行上传分片（UploadPart）
3. 完成上传（CompleteMultipartUpload）

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");
```

```
}
```

注意：

- 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
- 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，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。

查看当前正在进行的分片上传列表

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
<?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。

下载文件

下载文件

提示：

- 文件下载可以参考sample程序：samples/Object.php

下载文件到本地文件

以下代码可以下载文件到本地：

```php
<?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/upload-test-object-name.txt";
    $localfile = "upload-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");
}
```

下载文件到本地变量

以下代码可以下载文件到本地变量：

```
<?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";
    $options = array();
    try{
        $content = $ossClient->getObject($bucket, $object, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

管理文件

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。

提示：

- 以下场景的完整代码路径:samples/Object.php

列出存储空间中的文件

下面代码可以获取存储空间的文件列表:

```
<?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
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

删除一个文件

通过下面代码可以删除一个文件：

```
<?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");
}
```

删除多个文件

通过下面代码可以批量删除多个文件：

```
<?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");
}
```

拷贝一个文件

通过下面代码可以拷贝一个文件：

```
<?php
/**
 * 拷贝object
 * 当目的object和源object完全相同时，表示修改object的meta信息
 *
 * @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';
    $options = array();
    try{
        $ossClient->copyObject($from_bucket, $from_object, $to_bucket, $to_object, $options);
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

修改文件元信息(Object Meta)

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' => '2012-10-01 08:00:00',
            'Content-Disposition' => 'attachment; filename="xxxxxx"',
        ),
    );
    try{
        $ossClient->copyObject($fromBucket, $fromObject, $toBucket, $toObject, $copyOptions);
```

```
    } catch(OssException $e) {
        printf(__FUNCTION__ . ": FAILED\n");
        printf($e->getMessage() . "\n");
        return;
    }
    print(__FUNCTION__ . ": OK" . "\n");
}
```

获取文件的文件元信息(Object Meta)

通过下面代码，可以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");
    }
}
```

创建一个虚拟目录

通过下面代码可以创建一个虚拟目录：

```
<?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");
}
```

判断一个文件是否存在

通过下面代码可以判断一个文件是否存在：

```
<?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);
}
```

授权访问

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。

> 提示：
>
> - 以下场景的完整代码路径:samples/Signature.php

使用私有的下载链接

代码如下：

```
<?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; // 这个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_response_body(), $request-
>get_response_code());
    if ($res->isOK()) {
        print(__FUNCTION__ . ": OK" . "\n");
    } else {
        print(__FUNCTION__ . ": FAILED" . "\n");
    };
}
```

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。更多信息

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);
```

静态网站托管

静态网站托管

提示：

- 以下场景的完整代码路径:samples/BucketWebsiteManager.php

设置静态网站托管

通过下面代码可以设置静态网站托管：

```
<?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

通过下面代码可以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

下面代码可以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");
}
```

生命周期管理

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。*

提示：

- 以下场景的完整代码路径:samples/BucketLifecycle.php

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

可以通过下面的代码，设置上述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_DAY
S, 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_DAT
E, '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

可以通过下面的代码获取上述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

可以通过下面的代码清空存储空间中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");
}
```

设置访问日志

设置访问日志（Logging）

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

更多关于访问日志的内容请参考 Bucket访问日志

开启Bucket日志

通过下面代码可以开启Bucket日志

```php
<?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

通过下面代码可以查看Bucket日志配置：

```php
<?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

通过下面代码可以删除Bucket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");
}
```

跨域资源共享

跨域资源共享设置

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。

提示：

- 以下场景的完整代码路径:samples/BucketCors.php

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。代码如下：

```
<?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");
}
```

注意：

- 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

我们可以参考存储空间的CORS规则，通过GetBucketCORSRules方法。代码如下：

```
<?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

用于关闭指定存储空间对应的CORS并清空所有规则。

```
<?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");
}
```

防盗链

防盗链设置

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。

提示：

- 以下场景的完整代码路径:samples/BucketReferer.php

设置Referer白名单

通过下面代码设置Referer白名单：

```
<?php
```

```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");
}
```

注意：

- Referer参数支持通配符“*”和“？”，更多详细的规则配置可以参考产品文档OSS防盗链

获取Referer白名单

通过下面代码可以获取Referer白名单：

```php
<?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");
}
```

Ruby-SDK

安装

SDK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
```

> 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

注意：

- 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"。**注意：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。

   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。

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。

安装brew，在终端输入以下命令：

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

安装ruby，在终端输入以下命令：

```
brew install ruby
exec $SHELL -l
```

安装SDK，在终端输入以下命令：

```
gem install aliyun-sdk
```

在终端输入以下命令验证SDK安装成功：

```
irb
> require 'aliyun/oss'
=> true
```

快速开始

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
```

注： 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
```

注：

1. Bucket的命名规范请查看创建Bucket
2. 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
=> ...
```

了解更多

- 管理Bucket

- 上传文件
- 下载文件
- 管理文件
- Rails应用
- 自定义域名绑定
- 使用STS访问
- 设置访问权限
- 管理生命周期
- 设置访问日志
- 静态网站托管
- 设置防盗链
- 设置跨域资源共享
- 异常

管理Bucket

管理存储空间（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')
```

注意：

- Bucket的命名规范请查看创建Bucket
- 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')
```

注意：

- 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
```

上传文件

上传文件（Object）

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}"
}
```

注意：

- 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')
```

注意：

- 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。更多有关上传回调的内容请参考 OSS上传回调

目前OSS支持上传回调的接口只有put_object和resumable_upload。

```
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.put_object('files/hello', file: '/tmp/x', callback: callback)
rescue Aliyun::OSS::CallbackError => e
  puts "Callback failed: #{e.message}"
end
```

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
```

需要特别注意的是：

1. callback的url**不能**包含query string，而应该在:query参数中指定
2. 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。
3. 详细的例子可以参考callback.rb
4. 接受回调的server可以参考callback_server.rb

下载文件

下载文件（Object）

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}"
}
```

注意：

- 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)
```

管理文件

管理文件（Object）

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。公共前缀的概念请参考 列出Object

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：

```ruby
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。

注意：

- 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
```

Rails应用

与Rails集成

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::OSS::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 &lt; 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 &lt; 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。请参考管理文件

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：

1. 总是在第一个显示'../'指向上级目录

2. 对于Common prefix，显示为目录
3. 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，参考 下载文件

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。

自定义域名绑定

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。 绑定自定义域名请参考自定义域名绑定

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')
```

注意：

- 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）

使用STS访问

使用STS访问

OSS可以通过阿里云STS服务，临时进行授权访问。更多有关STS的内容请参考：阿里云STS 使用STS时请按以下步骤进行：

1. 在官网控制台创建子账号，参考OSS STS
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，参考OSS STS
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('&lt;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。

设置访问权限

设置访问权限（ACL）

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：

1. 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')
```

更多关于访问权限控制的内容请参考 访问控制

管理生命周期

管理生命周期（Lifecycle）

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：
  i. 指定距文件最后修改时间N天过期
  ii. 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

更多关于生命周期的内容请参考 文件生命周期

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 = []
```

设置访问日志

设置访问日志（Logging）

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

更多关于访问日志的内容请参考 Bucket访问日志

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)
```

静态网站托管

托管静态网站（Website）

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)
```

设置防盗链

设置防盗链（Referer）

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。更多OSS防盗链请参考： OSS防盗链

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: [])
```

设置跨域资源共享

设置跨域资源共享（CORS）

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。更多关于跨域资源共享的内容请参考 OSS跨域资源共享

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 = []
```

异常

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

OSS中常见的错误信息请参考 OSS错误响应

C-SDK

前言

前言

SDK下载

C SDK开发包（2015-11-12）版本0.0.7

- Linux:aliyun_OSS_C_SDK_v0.0.7.tar.gz
- Windows:oss_c_sdk_windows_v0.0.7.zip

版本迭代详情参考这里

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。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。

安装

安装

安装步骤

步骤如下：

1. 从官方网站下载Object Storage Service C SDK
2. 解压文件
3. 将解压后文件夹中的文件使用make命令生成liboss_c_sdk.a静态库
4. 将liboss_c_sdk.a静态库拷贝到您的工程第三方库中
5. 经过上面几步之后，您就可以在工程中使用OSS C SDK了

OSS C SDK采用autoconf和automake方式编译，使用 cURL 进行网络相关操作。因为需要用 HMAC 进行数字签名做授权，所以依赖了 OpenSSL 库。另外，OSS C SDK使用apr库作为底层数据结构，使用libxml2解析请求返回的xml，OSS C SDK并没有带上这几个外部库，因此在使用 OSS C SDK 之前需要先确认您的当前开发环境中是否已经安装了这所需的外部库，并且已经将它们的头文件目录和库文件目录都加入到了项目工程的设置。如何安装这些第三方库不在本文讨论范围，请自行查阅相关文档。

嵌入式环境下，安装第三方库时建议使用-Os优化选项。另外，可以参考以下链接减小第三方库的占用空间：Curl安装时优化选项参考http://www.cokco.cn/thread-11777-1-1.html， libxml2安装时优化选项参考http://curl.haxx.se/docs/install.html。

如果在项目构建过程中出现环境相关的编译错误和链接错误，请确认这些选项是否都已经正确配置，以及所依赖的库是否都已经正确的安装。

一键安装

- 一键化安装oss c sdk，轻松搞定第三方库依赖，参考这里

快速入门

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[])
{
    //aos_http_io_initialize first
    if (aos_http_io_initialize("oss_test", 0) != AOSE_OK) {
        exit(1);
    }
```

```
    //use OSS C SDK api to access OSS
    ...

    aos_http_io_deinitialize();
    return 0;
}
```

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。
aos_http_io_deinitialize清理OSS C SDK运行环境。

Step-2.初始化一个oss_request_options

OSS C SDK的OSS操作通过oss_request_options_t结构体完成的，下面代码创建一个oss_request_options对象:

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
char *oss_endpoint = "<oss endpoint>"

aos_pool_create(&p, NULL);

//init_ oss_request_options
oss_request_options = oss_request_options_create(p);
oss_request_options->config = oss_config_create(options->pool);
aos_str_set(&options->config->host, oss_endpoint);
options->config->port = oss_port;
aos_str_set(&options->config->id, "<your_access_key_id>");
aos_str_set(&options->config->key, "<your_access_key_secret>");
options->config->is_oss_domain = is_oss_domain;
options->ctl = aos_http_controller_create(options->pool, 0);
```

在上面代码中，变量 access_key_id与 access_key_secret 是由系统分配给用户的，称为ID对，用于标识用户，为访问OSS做签名验证。关于oss_request_options的详细介绍，参见oss_request_options。

Step-3. 新建bucket

Bucket是OSS全局命名空间，相当于数据的容器，可以存储若干Object。您可以按照下面的代码新建一个Bucket：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
oss_acl_e oss_acl = OSS_ACL_PRIVATE;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
```

```
s = oss_create_bucket(options, &bucket, oss_acl, &resp_headers);
aos_pool_destroy(p);
```

关于Bucket的命名规范，参见Bucket中的命名规范。

Step-4. 上传Object

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一个Object的上传：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
char *data = "<your object content>";
aos_list_t buffer;
aos_buf_t *content;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&object, object_name);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 1);
apr_table_set(headers, "x-oss-meta-author", "oss"); //object user meta

//read object content into buffer
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);

aos_pool_destroy(p);
```

关于Object的命名规范，参见Object中的命名规范 。关于上传Object更详细的信息，参见Object中的上传Object 。

Step-5. 列出所有Object

当您完成一系列上传后，可能需要查看某个Bucket中有哪些Object，可以通过下面的程序实现：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";
```

```
oss_list_object_params_t *params;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);

params = oss_create_list_object_params(p);
params->max_ret = 10;
aos_str_set(&params->prefix, "object_prefix");
aos_str_set(&params-> delimiter, "object_ delimiter");
aos_str_set(&params-> marker, "object_marker");

s = oss_list_object(options, &bucket, params, &resp_headers);;

aos_pool_destroy(p);
```

更多灵活的参数配置，可以参考Object中【列出Bucket中object】

Step-6. 获取指定Object

您可以参考下面的代码简单地实现一个Object的获取：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
aos_list_t buffer;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&object, object_name);

aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 0);

//get object content into buffer
aos_list_init(&buffer);
s = oss_get_object_to_buffer(options, &bucket, &object, headers, &buffer, &resp_headers);

aos_pool_destroy(p);
```

可以从响应的头部读取object的user meta信息。

配置初始化

oss_request_options

使用OSS C SDK进行OSS操作时需要初始化oss_request_options，其中config 变量用于存储访问OSS的基本信息，比如OSS域名、OSS端口号、用户的access_key_id 、用户的 access_key_secret 。is_oss_domain变量指定访问OSS是否使用三级域名（对于cname和ip访问，is_oss_domain设置为0）。ctl变量初始化OSS C SDK访问OSS的控制信息，通过设置ctl变量可以对访问OSS进行流控。

初始化oss_request_options

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
char *oss_endpoint = "<oss endpoint>";

aos_pool_create(&p, NULL);

//init_ oss_request_options
options = oss_request_options_create(p);
options ->config = oss_config_create(options->pool);
aos_str_set(&options->config->host,  oss_endpoint);
options->config->port=oss_port;
aos_str_set(&options->config->id, "<your_access_key_id>");
aos_str_set(&options->config->key, "<your_access_key_secret>");
options->config->is_oss_domain = is_oss_domain;
options->ctl = aos_http_controller_create(options->pool, 0);
```

其中oss_endpoint需填为待操作bucket所在region的对应值。

配置oss_request_options

如果想设置oss c sdk底层libcurl通信时的一些参数，可以对oss_request_options中的ctl成员变量进行设置。

Bucket

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。一个应用，例如图片分享网站，可以对应一个或多个Bucket。一个用户最多可创建10个Bucket，但每个Bucket中存放的Object的数量和大小总和没有限制，用户不需要考虑数据的可扩展性。

命名规范

Bucket的命名有以下规范：

- 只能包括小写字母，数字，短横线（-）
- 必须以小写字母或者数字开头
- 长度必须在3-63字节之间

新建Bucket

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
oss_acl_e oss_acl = OSS_ACL_PRIVATE;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
s = oss_create_bucket(options, &bucket, oss_acl, &resp_headers);

aos_pool_destroy(p);
```

创建bucket主要需要输入bucket名字与ACL。由于Bucket的名字是全局唯一的，所以请保证您的 bucket名字不与别人重复。而ACL目前取值为private，public-read， public-read-write中的一种，分别表示私有，公共读，公共读写。详细权限内容请参考OSS访问控制。如果Bucket已经存在且属于操作者，则可以覆盖该Bucket的ACL设置。

获取Bucket ACL

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";
aos_string_t oss_acl;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
s = oss_get_bucket_acl(options, &bucket,& oss_acl, &resp_headers);

aos_pool_destroy(p);
```

删除Bucket

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
```

```
char *bucket_name = "<your bucket name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
s = oss_delete_bucket (options, &bucket, &resp_headers);

aos_pool_destroy(p);
```

需要注意的是，如果Bucket不为空（Bucket中有Object，或者有分块上传的碎片），则Bucket无法删除，必须删除Bucket中的所有Object以及碎片后，Bucket才能成功删除。

Object

Object

在OSS中，用户操作的基本数据单元是Object。单个Object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object大小不能超过48.8TB。Object包含key、meta和data。其中，key是Object的名字；meta是用户对该object的描述，由一系列name-value对组成；data是Object的数据。

命名规范

Object的命名规范如下：

- 使用UTF-8编码
- 长度必须在1-1023字节之间
- 不能以“/”或者“\”字符开头
- 不能含有“\r”或者“\n”的换行符

上传Object

简单上传

从内存中上传数据到OSS

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
char *data = "<your object content>";
```

```
aos_list_t buffer;
aos_buf_t *content;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&object, object_name);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 0);

//read object content into buffer
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);
aos_pool_destroy(p);
```

上传本地文件到OSS

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
char *data = "<your object content>";
char *filename ="<your local filename>";
aos_string_t local_file;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&object, object_name);
aos_str_set(&local_file,filename);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 0);

s = oss_put_object_from_file (options, &bucket, &object, &local_file, headers, &resp_headers);

aos_pool_destroy(p);
```

使用该方法上传最大文件不能超过5G。如果超过可以使用mutipartupload上传。

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Object来存储。但给用户提供了创建模拟文件夹的方式，如下代码：

```
aos_pool_t *p;
```

```
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "folder_name/";
char *data = "";
aos_list_t buffer;
aos_buf_t *content;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
headers = aos_table_make(p, 0);

//read object content into buffer
...

s = oss_put_object_from_buffer (options, &bucket, &object, &buffer, headers, &resp_headers);

aos_pool_destroy(p);
```

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照样可以上传下载,只是控制台会对以"/"结尾的Object以文件夹的方式展示。所以用户可以使用上述方式来实现创建模拟文件夹。而对文件夹的访问可以参看文件夹模拟功能

设定Object的Http Header

OSS服务允许用户自定义Object的Http Header。下面代码为Object设置了过期时间：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>"
char *data = "<your object content>";
aos_list_t buffer;
aos_buf_t *content;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
```

```
aos_str_set(&object, object_name);

//set http header
headers = aos_table_make(p, 1);
apr_table_set(headers, "Expires", "Fri, 28 Feb 2012 05:38:42 GMT");

//read object content into buffer
...

aos_pool_destroy(p);
```

可以设置Http Header有：Cache-Control 、 Content-Disposition 、Content-Encoding 、 Expires 。它们的相关介绍见 RFC2616 。

设置User Meta

OSS支持用户自定义Meta信息对Object进行描述。比如：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>"
char *data = "<your object content>";
aos_list_t buffer;
aos_buf_t *content;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
headers = aos_table_make(p, 1);
//set user meta
apr_table_set(headers, "x-oss-meta-author", "oss");

//read object content into buffer
...

aos_pool_destroy(p);
```

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8KB。

> NOTE：user meta的名称大小写不敏感，比如用户上传object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”

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。

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *headers1;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
char *data = "<your object content>";
aos_list_t buffer;
aos_buf_t *content;
int64_t position = 0;
char *next_append_position;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&object, object_name);
aos_str_set(&bucket, bucket_name);
//get append position
headers = aos_table_make(p, 0);
s = oss_head_object(options, &bucket, &object, headers, &resp_headers);
if(s->code == 200) {
    next_append_position = (char*)(apr_table_get(resp_headers, "x-oss-next-append-position"));
    position = atoi(next_append_position);
}

//append object
headers1 = 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, headers1, &resp_headers);

aos_pool_destroy(p);


os_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
```

```
char *data = "<your object content>";
char *filename ="<your local filename>";
aos_string_t append_file;
int64_t position = 0;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...
aos_str_set(&object, object_name);
aos_str_set(&append_file,filename);
aos_str_set(&bucket, bucket_name);
headers = aos_table_make(p, 0);

s = oss_append_object_from_file (options, &bucket, &object, position, &append_file, headers, &resp_headers);

aos_pool_destroy(p);
```

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. 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。
5. List Objects请求的响应XML中，会把Appendable Object的Type设为Appendable。

分块上传

OSS允许用户将一个Object分成多个请求上传到后台服务器中，关于分块上传的内容，将在MultipartUpload中Object的分块上传 这一章中做介绍。

列出Bucket中的Object

Object信息存在于params的object_list中，可以aos_list_for_each_entry查看每个object的详细信息。
NOTE：默认情况下，如果Bucket中的Object数量大于100，则只会返回100个Object， 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Object数目，可以修改MaxKeys 参数，或者使用 Marker 参数分次读取。

拓展参数

通常，我们可以通过设置ListObjectsRequest的参数来完成更强大的功能。比如：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名，可通过is_oss_domain函数初始化
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";
```

```
oss_list_object_params_t *params;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);

params = oss_create_list_object_params(p);
params->max_ret = 10;
aos_str_set(&params->prefix, "pic");
aos_str_set(&params-> delimiter, "/");
aos_str_set(&params-> marker, "");

s = oss_list_object(options, &bucket, params, &resp_headers);;

aos_pool_destroy(p);
```

上述代码列出了bucket中以”pic”为前缀，以“/”为结尾的共同前缀CommonPrefixes。比如“pic-people/”。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。

获取Object

读取Object

我们可以通过操作输入流将Object的内容读取到文件或者内存中。Object的头信息可以通过resp_headers获得，包含了Object上传时定义的，ETag，Http Header以及自定义的元信息。

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。
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object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Object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。

我们通过设置Range 方法并根据返回的Object的范围，可以用此功能实现文件的分段下载和断点续传。

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
aos_list_t buffer;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
headers = aos_table_make(p, 1);

//设置range，读取文件的指定范围
  apr_table_set(headers, "Range", " bytes=20-100");

//get object content into buffer
aos_list_init(&buffer);
s = oss_get_object_to_buffer(options, &bucket, &object, headers, &buffer, &resp_headers);

aos_pool_destroy(p);
```

直接下载Object到文件 我们可以通过下面的代码直接将Object下载到指定文件：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
int object_name_len;
char *download_filename
aos_string_t download_file;
```

```
aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
aos_str_set(&download_file, download_filename);
headers = aos_table_make(p, 0);

s = oss_get_object_to_file (options, &bucket, &object, headers, &download_file,  &resp_headers);

aos_pool_destroy(p);
```

删除Object

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名，可通过is_oss_domain函数初始化
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char * object_name =  "<your object name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);

s = oss_delete_object (options, &bucket, &object, ,  &resp_headers);

aos_pool_destroy(p);
```

拷贝Object

在同一个region中，用户可以对有操作权限的object进行复制操作。同时提醒用户，如果拷贝文件超过1G，建议使用Upload Part Copy方式进行拷贝。

拷贝一个Object

通过 copyObject 方法我们可以拷贝一个Object，代码如下：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t * headers;
aos_table_t *resp_headers;
aos_string_t source_bucket;
aos_string_t source_object;
aos_string_t dest_bucket;
aos_string_t dest_object;
```

```
char *source_bucket_name =  "<your bucket name>";
char *source_object_name =  "<your object name>";
char *dest_bucket_name =  "<your bucket name>";
char *dest_object_name =  "<your object name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&source_bucket, 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);

s = oss_copy_object (options, &source_bucket, &source_object, &dest_bucket, &dest_object ,headers,  &resp_headers);

aos_pool_destroy(p);
```

需要注意的是源bucket与目的bucket必须属于同一region。

MultipartUpload

Multipart Upload

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。

分步完成Multipart Upload

初始化

初始化一个分块上传事件

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t * headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char * object_name =  "<your object name>";
aos_string_t upload_id;
```

```
aos_pool_create(&p, NULL);
// init_ oss_request_options
...
aos_str_set(&bucket, bucket_name);
aos_str_set(&object,object_name);

headers = aos_table_make(p, 0);
s = oss_init_multipart_upload (options, &bucket, &object, headers, &upload_id, &resp_headers);

aos_pool_destroy(p);
```

返回结果中含有 uplolad_id ，它是区分分块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

接着，把本地文件分块上传。 假设有一个文件，本地路径为 /path/to/file.zip 由于文件比较大，将其分块传输到OSS中。

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char * object_name =  "<your object name>";
aos_string_t upload_id;
char *filename = "<local filename>";
oss_upload_file_t *upload_file;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);

upload_file = oss_create_upload_file(p);
aos_str_set(&upload_file->filename, filename);
upload_file->file_pos = 0;
upload_file->file_last = 200 * 1024; //200KB
s = oss_upload_part_from_file (options, &bucket, &object, &upload_id, part_num,  upload_file, &resp_headers);

aos_pool_destroy(p);
```

上面程序的核心是调用oss_upload_part_from_file 方法来上传每一个分块，但是要注意以下几点：

- oss_upload_part_from_file方法要求除最后一个Part以外，其他的Part大小都要大于100KB。
- 为了保证数据在网络传输过程中不出现错误，强烈推荐用户在上传part时携带meta：content-md5，在OSS收到数据后，用该MD5值验证上传数据的正确性，如果出现不一致会返回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。

获取所有已上传的块信息

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name = "<your bucket name>";
char *object_name = "<your object name>";
aos_string_t upload_id;
oss_list_upload_part_params_t *params;
int max_ret = 1000;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);

params = params = oss_create_list_upload_part_params(p);
params->max_ret = max_ret;

s = oss_list_upload_part (options, &bucket, &object, &upload_id, params, &resp_headers);;

aos_pool_destroy(p);
```

所有分块信息存储在oss_list_upload_part_params_t的part_list中，可通过aos_list_for_each_entry对part_list进行遍历。

完成分块上传

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char * object_name =  "<your object name>";
aos_string_t upload_id;
oss_list_upload_part_params_t *params;
oss_list_part_content_t *part_content;
oss_complete_part_content_t *complete_content;
aos_list_t complete_part_list;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);
```

```
//获取所有已上传的分块信息
...

//构造complete的分块列表
aos_list_for_each_entry(part_content, &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);
}

s = oss_complete_multipart_upload (options, &bucket, &object, &upload_id, &complete_part_list,  &resp_headers);

aos_pool_destroy(p);
```

取消分块上传事件

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char * object_name =  "<your object name>";
aos_string_t upload_id;

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);
aos_str_set(&object, object_name);

s = oss_abort_multipart_upload (options, &bucket, &object, &upload_id, &resp_headers);

aos_pool_destroy(p);
```

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。

multipart upload copy复制文件

Upload Part Copy 通过从一个已经存在的object中拷贝数据来上传一个object。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。

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_string_t upload_id;
oss_list_upload_part_params_t *list_upload_part_params;
oss_upload_part_copy_params_t *upload_part_copy_params1;
aos_table_t *headers;
```

```
aos_table_t *resp_headers;
aos_table_t *list_part_resp_headers;
aos_list_t complete_part_list;
oss_list_part_content_t *part_content;
oss_complete_part_content_t *complete_content;
aos_table_t *complete_resp_headers;
aos_status_t *s;
int part1 = 1;
char *source_bucket_name = "<your bucket name>";
char *dest_bucket_name = "<your bucket name>";
char *source_object_name = "<your source object name>";
char *dest_object_name = "<your dest object name>";
aos_string_t dest_bucket;
aos_string_t dest_object;
int64_t range_start1 = 0;
int64_t range_end1 = 6000000;//not less than 5MB

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

//init mulitipart upload
//upload part copy part 1
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;
headers = aos_table_make(p, 0);
s = oss_upload_part_copy(options, upload_part_copy_params1, headers, &resp_headers);

// continue upload part copy like part 1
...

//list part
list_upload_part_params = oss_create_list_upload_part_params(p);
list_upload_part_params->max_ret = 1000;
aos_list_init(&complete_part_list);

aos_str_set(&dest_bucket, dest_bucket_name);
aos_str_set(&dest_object, dest_object_name);
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);
}

//complete multipart upload
```

```
s = oss_complete_multipart_upload(options, &dest_bucket, &dest_object, &upload_id, &complete_part_list, &complete_resp_headers);

aos_pool_destroy(p);
```

生命周期管理

生命周期管理（Lifecycle）

OSS提供Object生命周期管理来为用户管理对象。用户可以为某个Bucket定义生命周期配置，来为该Bucket的Object定义各种规则。目前，用户可以通过规则来删除相匹配的Object。每条规则都由如下几个部分组成：

- Object名称前缀，只有匹配该前缀的Object才适用这个规则
- 操作，用户希望对匹配的Object所执行的操作。
- 日期或天数，用户期望在特定日期或者是在Object最后修改时间后多少天执行指定的操作。

设置Lifecycle

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>
```

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object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object，Date则表示到指定的绝对时间之后就删除object(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;
int is_oss_domain = 1;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";
aos_list_t lifecycle_rule_list;
oss_lifecycle_rule_content_t *rule_content;

aos_pool_create(&p, NULL);
```

```
// init_ oss_request_options
...

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, "delete obsoleted files ");
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);

aos_pool_destroy(p);
```

可以通过下面的代码获取上述lifecycle规则。

```
aos_pool_t *p;
int is_oss_domain = 1;
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
char *bucket_name =  “<your bucket name>”;
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);
// init_ oss_request_options
...

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);
```

可以通过下面的代码清空bucket中lifecycle规则。

```
aos_pool_t *p;
int is_oss_domain = 1;
oss_request_options_t *options;
```

```
aos_status_t *s;
aos_table_t *resp_headers;
aos_string_t bucket;
char *bucket_name = "<your bucket name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

aos_str_set(&bucket, bucket_name);

s = oss_delete_bucket_lifecycle(options, &bucket, &resp_headers);

aos_pool_destroy(p);
```

授权访问

授权访问

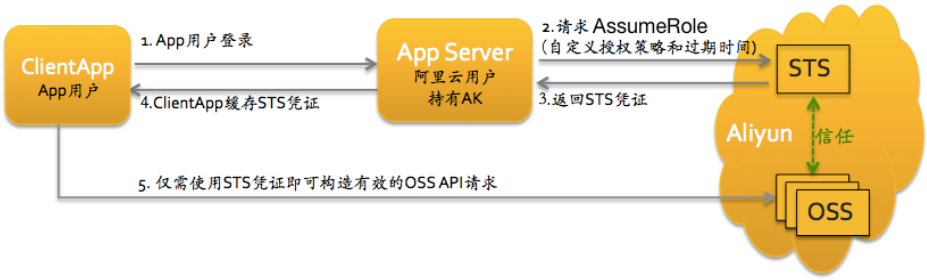
使用STS服务临时授权

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。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oss_request_options。以上传Object为例：

```
aos_pool_t *p;
int is_oss_domain = 1;//是否使用三级域名
oss_request_options_t *options;
aos_status_t *s;
aos_table_t *headers;
aos_table_t *resp_headers;
aos_string_t bucket;
aos_string_t object;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>"
char *data = "<your object content>";
aos_list_t buffer;
aos_buf_t *content;
char oss_endpoint = "oss endpoint";

aos_pool_create(&p, NULL);

// init_oss_request_options using sts_token
options = oss_request_options_create(p);
options->config = oss_config_create(options->pool);
aos_str_set(&options->config->host,  oss_endpoint);
options->config->port = oss_port;
aos_str_set(&options->config->id, "<tmp_access_key_id>");
aos_str_set(&options->config->key, "<tmp_access_key_secret>");
aos_str_set(&options->config->sts_token, "<sts_token>");
options->config->is_oss_domain = is_oss_domain;
options->ctl = aos_http_controller_create(options->pool, 0);

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);

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，您可以指定URL过期的时间，从而限制用户长时间访问。

生成签名url

生成get object请求url签名。

```
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;
char *url_str;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

req = aos_http_request_create(p);
req->method = HTTP_GET;
url_str = gen_test_signed_url(options, bucket_name, object_name, expire_time, req);

aos_pool_destroy(p);
```

生成put object请求url签名：

```
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;
char *url_str;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

req = aos_http_request_create(p);
req->method = HTTP_PUT;
url_str = gen_test_signed_url(options,bucket_name, object_name, expire_time, req);

aos_pool_destroy(p);
```

使用签名URL发送请求

使用URL签名的方式getobject

```
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 *url_str;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
aos_string_t url;
aos_string_t download_file;
char *filename = "<local filename>"

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

headers = aos_table_make(p, 0);
req = aos_http_request_create(p);
req->method = HTTP_GET;
url_str = gen_test_signed_url(options,bucket_name, object_name, expire_time, req);
aos_str_set(&url, url_str);
aos_str_set(&download_file, filename);
s = oss_get_object_to_file_by_url(options, &url, headers,&download_file, &resp_headers);

aos_pool_destroy(p);
```

使用URL签名的方式putobject

```
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 *url_str;
char *bucket_name =  "<your bucket name>";
char *object_name =  "<your object name>";
aos_string_t url;
aos_string_t local_file;
char *filename = "<local filename>"

aos_pool_create(&p, NULL);
// init_ oss_request_options
...

headers = aos_table_make(p, 0);
req = aos_http_request_create(p);
req->method = HTTP_PUT;
url_str = gen_test_signed_url(options,bucket_name, object_name, expire_time, req);
aos_str_set(&url, url_str);
aos_str_set(&local_file, filename);
s = oss_put_object_from_file_by_url(options, &url, &local_file, headers, &resp_headers);

aos_pool_destroy(p);
```

错误响应

错误响应

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
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

常见问题

常见问题分享

一键化安装OSS C SDK，轻松搞定第三方库依赖：

点击查看

OSS C SDK在嵌入式环境下如何编译：

点击查看

OSS C SDK如何设置程序日志的输出：

点击查看

OSS C SDK如何设置通信时和CURL相关的一些参数：

点击查看

OSS C SDK利用CURL提供的Callback实现上传：

点击查看

OSS C SDK利用CURL提供的Callback实现数据下载：

点击查看

SDK开发包下载

OSS SDK开发包

OSS SDK 开发包

语言	版本号	下载详情
Java SDK	2.0.7	点击这里
Python SDK	0.4.2	点击这里
Android SDK	1.4.0	点击这里
iOS SDK	2.1.1	点击这里
.NET SDK	2.0.0	点击这里
PHP SDK	-	点击这里

C SDK	-	点击这里

Java SDK开发包

Java SDK 开发包

Java SDK文档

点击查看

Java SDK开发包(2015-11-24) 版本号 2.0.7

Java SDK下载地址 ：java_sdk_20151124.zip

更新日志:

1. 修复SetObjectAcl存在的bug
2. 去掉log4j依赖，采用commons-logging

Java SDK开发包(2015-09-22) 版本号 2.0.6

更新日志:

1. 添加setObjectAcl/getObjectAcl接口，用于设置、获取指定Object的ACL；
2. 添加ClientConfiguration.setLogLevel接口设置全局日志级别，默认将404错误码日志级别设置为INFO；
3. 添加ClientConfiguration.setSLDEnabled接口设置是否开启二级域名的访问方式；
4. 添加ClientConfiguration.setSupportCname接口设置是否支持Cname作为Endpoint；
5. 输出Invalid Response详细错误信息、完善OssException/ClientException错误信息；
6. 修复设置Bucket默认ACL为private的bug；
7. 修复upload part支持chunked编码的bug，并将输入流包装为可重试输入流；
8. 添加webp至mime types列表。

Java SDK开发包(2015-07-10) 版本号 2.0.5

Java SDK下载地址 ：java_sdk_20150710.zip

更新日志:

- 添加Append Object支持；
- 添加HTTPS支持；
- DeleteObject、ListObjects接口添加encoding-type参数，允许指定Object名称的编码方式；
- 去除Expires响应头日期格式的强制检查，修复无法解析非GMT日期格式的Expires响应头。

Java SDK开发包(2015-05-29) 版本号 2.0.4

Java SDK下载地址 ：java_sdk_20150529.zip

更新日志:

- 添加STS支持；
- 新增测试Demo Jar，具体使用方式可参考demo文件夹下的Readme.txt；
- 修改CreateBucket逻辑，允许创建Bucket同时指定Bucket ACL；
- 修改bucket名称检查规则，允许操作带有下划线且已存在的Bucket，但不允许创建带有下划线的Bucket；
- 优化HTTP连接池，提升并发处理能力。

Java SDK开发包(2015-04-24) 版本号 2.0.3

Java SDK下载地址 ：java_sdk_20150424.zip

更新日志:

- 新增deleteObjects接口，批量删除Object列表；
- 新增doesObjectExist接口，判断指定Bucket下的Object是否存在；
- 新增switchCredentials接口，可用于更换已有OSSClient实例的Credentials；
- 新增带有CredentialsProvider（Credentials提供者）的OSSClient构造函数，可对CredentialsProvider进行扩展定制；
- 修复copyObject接口中Source Key包含特殊字符加号时出现的bug；
- 调整OSSException/ClientException异常信息的显示格式；

Java SDK开发包(2015-03-23) 版本号 2.0.2

Java SDK下载地址 ：java_sdk_20150323.zip

更新日志:

- 新增GeneratePostPolicy/calculatePostSignature接口，用于生成Post请求的Policy字符串及Post签名
- 支持Bucket Lifecycle
- 新增基于URL签名的Put/GetObject重载接口
- 支持PutObject、UploadPart以Chunked编码的方式上传数据
- 修复若干Bug

Java SDK开发包(2015-01-15) 版本号 2.0.1

Java SDK下载地址 ：java_sdk_20150115.zip

更新日志:

- 支持Cname，允许用户指定哪些是保留域名
- 生成URI时支持用户指定ContentType及ContentMD5
- CopyObject请求不支持Server端加密的问题
- 更改UserAgent的格式
- 扩充Location常量 - 增加部分Samples

Java SDK开发包(2014-11-13)

Java SDK下载地址 ：java_sdk_20141113.zip

新增内容： Upload Part Copy 功能 OSS Java SDK源码 示例代码

重要提示： 2.0.0 版本OSS Java SDK移除了2.0.0之前版本中OTS相关代码，调整了包结构，加入了OSS SDK源码和示例代码。 使用2.0.0之前版本开发的程序在使用2.0.0版本需要修改引用的包名称。 包名称 com.aliyun.openservices.* 与 com.aliyun.openservices.oss.* 更换为com.aliyun.oss.*。

Python SDK开发包

Python SDK 开发包

Python SDK文档

点击查看

Python SDK开发包(2015-11-20) 版本 0.4.2

更新日志:

- osscmd增加appendfromfile接口支持自动追加文件内容到指定object
- API中添加apengd(追加上传相关接口)

Python SDK开发包下载地址:OSS_Python_API_20151120.zip

Python cmd工具使用说明：点击查看

Python SDK开发包(2015-08-11) 版本 0.4.1

更新日志:

- osscmd支持通过对响应的xml编码接受对包括控制字符的list和delete object。

Python SDK开发包下载地址:OSS_Python_API_20150811.zip

Python cmd工具使用说明：点击查看

Python SDK开发包(2015-07-07) 版本 0.4.0

更新日志:

- osscmd中支持STS功能

Python SDK开发包下载地址:OSS_Python_API_20150707.zip

Python cmd工具使用说明：点击查看

Python SDK开发包(2015-06-24) 版本 0.3.9

更新日志:

- osscmd中添加copylargefile命令，支持大文件复制

Python SDK开发包下载地址:OSS_Python_API_20150624.zip

Python cmd工具使用说明：点击查看

Python SDK开发包(2015-04-13) 版本 0.3.8

更新日志:

- osscmd中修复multiupload指定max_part_num不生效的问题
- oss_api增加对upload_part指定part的md5校验

Python SDK开发包下载地址:oss_python_sdk_20150413.zip

Python cmd工具使用说明：点击查看

Python SDK开发包(2015-01-29) 版本 0.3.7

更新日志:

- oss_api中增加了lifecycle和referer相关的接口。
- osscmd中增加了lifecycle和referer相关的命令。
- osscmd中修复了指定upload_id不生效的问题。

Python SDK开发包下载地址:oss_python_sdk_20150129.zip

Python cmd工具使用说明：点击查看

Python SDK开发包(2014-12-31) 版本 0.3.6

更新日志:

- osscmd的uploadfromdir命令增加了check_point功能，使用--check_point选项设置。
- osscmd的deleteallobject命令增加--force功能，强制删除所有文件。
- osscmd的multipart命令和uploadfromdir、downloadtodir命令增加--thread_num选项，可以调整线程数
- oss_api中增加了根据文件名生成Content-Type的功能。
- osscmd的downloadtodir命令增加--temp_dir选项，支持将下载的文件临时存在指定目录下。
- osscmd增加--check_md5选项，能对上传的文件进行md5检查。

Python SDK开发包下载地址:oss_python_sdk_20141231.zip

快速入门可参考SDK中的README文件

Python SDK 开发包（2014-05-09）

更新日志:

- 修复oss_util中logger初始化错误的bug。
- 优化在某些情况下oss_api中multi_upload_file上传接口，减少网络异常情况下重传的次数。

Python SDK开发包下载地址：oss_python_sdk_20140509.zip

Android SDK开发包

Android SDK 开发包

Android SDK文档（针对2.0.1版本）

点击查看

Android SDK开发包(2015-12-04) 版本号2.0.1

开发包下载地址：aliyun_OSS_Android_SDK_20151206

更新日志：

2.0.1版本为大版本更新，不向前兼容1.4.0版本及以下。

1. 接口重构，更易用更多特性；
2. GetObject返回输入流，由用户自行处理数据；
3. 终端时间不准确时，自动同步服务器时间；

代码库地址：https://github.com/aliyun/aliyun-oss-android-sdk

maven坐标：

```
<dependency>
   <groupId>com.aliyun.dpa</groupId>
   <artifactId>oss-android-sdk</artifactId>
   <version>2.0.1</version>
</dependency>
```

Android SDK开发包(2015-09-13) 版本号1.4.0

开发包下载地址：OSS_Android_SDK_201500913

文档下载：**点击下载**

更新日志：

1. 替换网络库为OKHTTP；
2. SDK发布到公网maven；
3. demo移到Github维护: https://github.com/alibaba/dpa-demo-android；

maven坐标：

```
<dependency>
  <groupId>com.aliyun.dpa</groupId>
  <artifactId>oss-android-sdk</artifactId>
  <version>1.4.0</version>
</dependency>
```

Android SDK开发包(2015-07-31) 版本号1.3.0

开发包下载地址：OSS_Android_SDK_20150731

更新日志：

1. 开放独立的分块上传接口；
2. 增加OSSData直接从输入流上传的接口；
3. 修复STS模式下初始化时多获取了一次token的问题；
4. 修复断点下载主动取消后流没有读尽影响连接复用的问题；

Android SDK开发包(2015-07-01) 版本号1.2.0

开发包下载地址：OSS_Android_SDK_20150701

更新日志：

1. 使用httpdns解析域名，防止域名被劫持；
2. 优化连接复用，增加并发请求场景的稳定性；

Android SDK开发包(2015-06-02) 版本号1.1.0

开发包下载地址：OSS_Android_SDK_20150602

更新日志：

1. 支持STS鉴权方式；
2. ListObjectsInBucket的结果加上commonPrefixs；
3. 增加获取文件的输入流方法；

Android SDK开发包(2015-04-07) 版本号1.0.0

开发包下载地址：OSS_Android_SDK_20150407

> 需要注意，本次OSS Android SDK正式集成入阿里云OneSDK，为了风格统一，本次更新，SDK包名和若干接口名有所变动，细节请参考文档。

更新日志：

1. 支持ListObjectsInBucket；
2. 支持断点下载；
3. 支持全局的网络参数设置和断点续传的一些配置项；
4. 更改包命名风格与阿里云OneSDK统一：包名由：com.aliyun.mbaas.oss.*更新为：com.alibaba.sdk.android.oss.*；

Android SDK文档（针对0.3.0）

点击下载

Android SDK开发包(2015-01-23) 版本号0.3.0

开发包下载地址：OSS_Android_SDK_20150123

更新日志:

1. 完善对cname和CDN加速域名的支持
2. 增加定制基准时间接口
3. 异步操作时，token在子线程中生成
4. 检测HTTP异常响应是否来自OSS Server

Android SDK开发包(2014-12-20) 版本号0.2.2

开发包下载地址：OSS_Android_SDK_20141220

ChangeList：

1. 修复0.2.1版本中异步上传接口中objectKey参数错误传入BucketName的bug

Android SDK开发包(2014-12-17) 版本号0.2.1

开发包下载地址：OSS_Android_SDK_20141217

更新日志:

1. 增加OSSBucket类，可以针对单个Bucket进行域名、权限、加签设置；
2. 增加权限设置，可以指定某个Bucket的访问权限；
3. 在异步任务的进度回调和异常回调增加参数objectKey；
4. 可以为一个OSSObject生成访问URL，方便授权第三方URL访问；
5. 所有异步上传、下载任务都可以中途取消；
6. 修复断点上传接口设置Content-type无效的bug；

Android SDK开发包(2014-11-26) 版本号0.0.1

开发包下载地址：OSS_Android_SDK_20141126

iOS SDK开发包

iOS SDK 开发包

iOS SDK文档（针对2.1.1）

点击查看

iOS SDK开发包(2015-11-20) 版本号2.1.1

- iOS SDK开发包(2015-11-20) 版本号 2.1.1：aliyun_OSS_iOS_SDK_20151120.zip
- github地址：https://github.com/aliyun/aliyun-oss-ios-sdk
- pod依赖： pod 'AliyunOSSiOS', '~> 2.1.1'
- demo地址： https://github.com/alibaba/alicloud-ios-demo

更新日志：

1. putObject支持servercallback
2. 重试策略调整

iOS SDK开发包(2015-10-20) 版本号2.0.2

此次更新iOS SDK进行完全重构，不再兼容旧版本，旧版本处于只维护不更新状态，建议尽快迁移到新版本。新版本SDK要求iOS 7.0+，采用RESTFul风格接口，代码开源，用Pod管理依赖。

- 开发包下载地址：OSS_iOS_SDK_20151020
- github地址：https://github.com/aliyun/aliyun-oss-ios-sdk
- pod依赖: pod 'AliyunOSSiOS', :git => 'https://github.com/aliyun/AliyunOSSiOS.git'
- demo地址: https://github.com/alibaba/alicloud-ios-demo

更新日志：

1. 支持与RESTFul Api一致的参数设置
2. 支持后台传输服务
3. 支持下载时的分段回调，方便实现视频边下边播功能

iOS SDK文档（针对1.3.0）

点击下载

iOS SDK开发包(2015-08-05) 版本号1.3.0

开发包下载地址：OSS_iOS_SDK_20150805

更新日志：

1. 开放独立的分块上传接口
2. STS鉴权模式下，SDK自主管理token的生命周期，失效后才会重新获取
3. 修复并发情况下上传任务可能无法取消的问题
4. 上传、下载的异步接口返回handler，通过handler取消任务
5. 增加完整demo
6. 支持servercallback功能

iOS SDK开发包(2015-06-30) 版本号1.2.0

开发包下载地址：OSS_iOS_SDK_20150630

更新日志：

1. 使用httpdns解析域名，防止域名被劫持

iOS SDK开发包(2015-06-09) 版本号1.1.0

开发包下载地址：OSS_iOS_SDK_20150609

更新日志：

1. 将sdk的使用方式由原来的.a文件更改为framework
2. 增加对sts的支持
3. 修复个别情况下回调失效的bug

iOS SDK开发包(2015-04-07) 版本号1.0.0

开发包下载地址：OSS_iOS_SDK_20150407

> 注意：本次OSS iOS SDK正式集成入阿里云OneSDK，为了风格统一，本次更新，SDK包名和若干接口名有所变动，细节请参考SDK文档

更新日志：

1. 增加list Objects功能
2. 指定范围下载功能支持指定文件结尾
3. 增加使用OSS SDK新的使用方式

iOS SDK文档（针对0.1.2）

点击下载

iOS SDK开发包(2015-03-04) 版本号0.1.2

开发包下载地址：OSS_iOS_SDK_20150304

更新日志:

1. 支持上传中文字符命名的object key
2. 修复断点续传功能bug

iOS SDK开发包(2015-01-20) 版本号0.1.1

开发包下载地址：OSS_iOS_SDK_20150120

更新日志:

1. 将开发包提供方式由原来的framework变更为static library

2. 开发包同时提供真机、模拟器、真机和模拟器通用的三种static library
3. 添加bucket设置指向绑定的Cname域名接口

iOS SDK开发包(2014-12-22) 版本号0.1.0

开发包下载地址：OSS_iOS_SDK_20141222

PHP SDK开发包

PHP SDK 开发包

PHP SDK 开发包(2015-08-19)

下载地址 ：oss_php_sdk_20150819

更新日志

- 修复了在有response-content-disposition等HTTP Header的时候，下载签名不对的问题。
- 新增了转换响应body的设置，目前支持xml,array,json三种格式。默认为xml
- 新增copy_upload_part方法
- 支持sts
- 调整签名url中$options参数的位置
- fix read_dir循环遍历的问题
- 增加了referer和lifecycle相关的接口。增加了upload by file和multipart upload时候的content-md5检查的选项。
- 增加了init_multipart_upload 直接获取string类型的upload
- 调整了batch_upload_file函数的返回值，从原来的空，变成boolean，成功为true，失败为false。
- 调整了sdk.class.php 中工具函数的位置，放置在util/oss_util.class.php中，如果需要引用，需要增加OSSUtil::， 并引用该文件。

修复的Bug：

- 修复Copy object的过程中无法修改header的问题。
- 修复upload part时候自定义upload的语法错误。
- 修复上传的时候，office2007文件的mimetype无法设置正确的问题。
- 修复batch_upload_file时候，遇到空目录会超时退出的问题。

PHP SDK 开发包(2014-06-25)

下载地址 ：oss_php_sdk_20140625 新增功能：

- 加入设置 CORS 功能

PHP SDK V1 开发包（2013-06-25）

PHP SDK开发包（2012-10-10）

此版本主要按新API发布内容，修改生成域名规则

PHP SDK开发包（2012-08-17）

此版本主要解决fix get_sign_url无法设置Expires的问题

PHP SDK开发包（2012-06-12）

更新日志:

1. 修复了设置hostname的bug
2. 优化了内部的Exception处理
3. 支持三级域名，如bucket.storage.aliyun.com
4. 优化了demo程序，使得看起来更加的简洁

C SDK开发包

C SDK 开发包

OSS C SDK文档

点击查看

OSS C SDK开发包（2015-11-12）版本0.0.7

下载地址：

- Linux:aliyun_OSS_C_SDK_v0.0.7.tar.gz
- Windows:oss_c_sdk_windows_v0.0.7.zip

更新日志：

1. OSS C SDK修复sts_token超过http header最大限制的问题

OSS C SDK开发包（2015-10-29）版本0.0.6

下载地址：

- Linux:aliyun_OSS_C_SDK_v0.0.6.tar.gz
- Windows:oss_c_sdk_windows_v0.0.6.zip

更新日志：

1. OSS C SDK签名时请求头支持x-oss-date，允许用户指定签名时间，解决系统时间偏差导致签名出错的问题
2. OSS C SDK支持CNAME方式访问OSS，CNAME方式请求时指定is_oss_domain值为0
3. 新增OSS C SDK demo,提供简单的接口调用示例，方便用户快速入门
4. OSS C SDK sample示例中去除对utf8第三方库的依赖

OSS C SDK开发包（2015-09-14）版本0.0.5

下载地址：

- Linux:aliyun_OSS_C_SDK_v0.0.5.tar.gz
- Windows:aliyun_OSS_C_SDK_windows_v0.0.5.rar

更新日志：

1. 调整签名时获取GMT时间的方法
2. 调整req_id的处理方式，改为从aos_status_t放回状态中直接获取

OSS C SDK开发包（2015-08-17）版本0.0.4

下载地址：

- Linux:aliyun_OSS_C_SDK_v0.0.4.tar.gz
- Windows:aliyun_OSS_C_SDK_windows_v0.0.4.rar

更新日志：

1. 支持keeplive长连接
2. 支持lifecycle设置

OSS C SDK开发包（2015-07-08）版本0.0.3

下载地址：

- Linux:aliyun_OSS_C_SDK_v0.0.3.tar.gz
- Windows:aliyun_OSS_C_SDK_windows_v0.0.3.rar

更新日志：

1. 增加oss_append_object_from_buffer接口，支持追加上传buffer中的内容到object
2. 增加oss_append_object_from_file接口，支持追加上传文件中的内容到object

OSS C SDK开发包（2015-06-10）版本0.0.2

更新日志：

1. 增加oss_upload_part_copy接口，支持Upload Part Copy方式拷贝
2. 增加使用sts服务临时授权方式访问OSS

OSS C SDK开发包（2015-05-28）版本0.0.1

更新日志：

1. 增加oss_create_bucket接口，创建oss bucket
2. 增加oss_delete_bucket接口，删除oss bucket
3. 增加oss_get_bucket_acl接口，获取oss bucket的acl
4. 增加oss_list_object接口，列出oss bucket中的object
5. 增加oss_put_object_from_buffer接口，上传buffer中的内容到object
6. 增加oss_put_object_from_file接口，上传文件中的内容到object
7. 增加oss_get_object_to_buffer接口，获取object的内容到buffer

8. 增加oss_get_object_to_file接口，获取object的内容到文件
9. 增加oss_head_object接口，获取object的user meta信息
10. 增加oss_delete_object接口，删除object
11. 增加oss_copy_object接口，拷贝objet
12. 增加oss_init_multipart_upload接口，初始化multipart upload
13. 增加oss_upload_part_from_buffer接口，上传buffer中的内容到块中
14. 增加oss_upload_part_from_file接口，上传文件中的内容到块
15. 增加oss_list_upload_part接口，获取所有已上传的块信息
16. 增加oss_complete_multipart_upload接口，完成分块上传
17. 增加oss_abort_multipart_upload接口，取消分块上传事件
18. 增加oss_list_multipart_upload接口，获取bucket内所有分块上传事件
19. 增加oss_gen_signed_url接口，生成一个签名的URL
20. 增加oss_put_object_from_buffer_by_url接口，使用url签名的方式上传buffer中的内容到object
21. 增加oss_put_object_from_file_by_url接口，使用url签名的方式上传文件中的内容到object
22. 增加oss_get_object_to_buffer_by_ur接口，使用url签名的方式获取object的内容到buffer
23. 增加oss_get_object_to_file_by_url接口，使用url签名的方式获取object的内容到文件中
24. 增加oss_head_object_by_url接口，使用url签名的方式获取object的user meta信息

.NET SDK开发包

.NET SDK 开发包

.NET SDK文档

点击查看

.NET SDK 开发包 2.1.0（2015-11-28）

下载地址：aliyun_dotnet_sdk_2.1.0

环境要求：

- .NET Framework 2.0及以上版本
- 必须注册有Aliyun.com用户账户

程序集：Aliyun.OSS.dll

版本号：2.1.0

包结构：

- bin
  - Aliyun.OSS.dll .NET程序集文件
  - Aliyun.OSS.pdb 调试和项目状态信息文件
  - Aliyun.OSS.xml 程序集注释文档
- doc
  - Aliyun.OSS.chm 帮助文档

    - src
        - SDK源代码
- sample
    - Sample源代码

更新日志:

2015/11/28

- .NET Framework 2.0和.NET Framework 3.5支持
- 大文件拷贝接口：CopyBigObject
- 大文件上传接口：UploadBigObject
- 文件meta修改接口：ModifyObjectMeta
- 提升SDK健壮性
- ContentType支持大多数MIME种类（226种）
- 补齐SDK API速查文档中的缺失注释
- 删除某些类中已经废弃的ObjectMetaData属性，请使用类中对应的的ObjectMetadata属性

.NET SDK 开发包（2015-11-18）

下载地址：aliyun_dotnet_sdk_2.0.0

更新日志:

- 自动根据对象key和上传文件名后缀判断ContentType
- ListObjects，ListMultipartUploads，DeleteObjects接口默认增加EncodingType参数
- UploadPart接口新增内容md5校验
- 新增部分示例程序
- 精简命名空间
- 合并重复目录
- 统一目录名称
- 删除重复的测试项目
- 修改CNAME支持形式
- 当存储空间或者文件不存在时，DoesObjectExist不再抛出异常，而是返回false

.NET SDK 开发包（2015-05-28）

下载地址：aliyun_dotnet_sdk_20150528

更新日志:

- 添加Bucket Lifecycle支持，可添加、删除Lifecycle规则；
- 添加DoesBucketExist、DoesObjectExist接口，用于判断Bucket/Object的存在性；
- 添加SwitchCredentials，可在运行期更换用户账号信息；
- 添加ICredentialsProvider接口类，通过实现该类提供自定义的Credentials生成策略。
- 添加GeneratePostPolicy接口，用于生成Post Policy；
- 添加异步化接口（支持Put/Get/List/Copy/PartCopy等异步操作）；
- 添加STS支持；

- 添加自定义时间校准功能，可通过Client配置项SetCustomEpochTicks接口进行设置；
- 添加Chunked编码传输支持，上传时可以不指定Content-Length项；
- 修复设置Bucket CORS的Expose Header属性，但取出的结果为空的bug；
- 修复ListObjects请求返回的CommonPrefixs包含多个Prefix时，SDK只能取到第一个Prefix的bug；
- 修复当出现OSS相关异常时，解析出的RequestId、HostId为null的bug；
- 修复CopyObject/CopyPart接口中source key包含中文，出现编码错误的bug;

.NET SDK 开发包（2015-01-15）

下载地址：aliyun_dotnet_sdk_20150115

更新日志:

- 移除OTS分支，程序集命名更改为Aliyun.OSS.dll
- .NET Framework版本升至4.0及以上
- OSS: 添加Copy Part、Delete Objects、Bucket Referer List等接口
- OSS: 添加ListBuckets分页功能
- OSS: 添加CNAME支持
- OSS: 修复Put/GetObject流中断问题
- OSS: 增加Samples

.NET SDK 开发包（2014-06-26）

下载地址：aliyun_dotnet_sdk_20140626

Open Services SDK for.NET 包含了OSS和OTS的SDK。.NETSDK采用了与 Java SDK统一的接口设计，并结合C#语言特点适当地改进。（最新版本已支持OSS的分块上传操作）

环境要求：

- .NETFramework 3.5 SP1或以上版
- 必须注册有Aliyun.com用户账户，并开通相关服务（OSS、OTS）。

更新日志： 2014/06/26- OSS:

- 加入cors功能。

2013/09/02- OSS:

- 修复了某些情况下无法抛出正确的异常的Bug。
- 优化了SDK的性能。

2013/06/04- OSS:

- 将默认OSS服务访问方式修改为三级域名方式。

2013/05/20- OTS:

- 将默认的OTS服务地址更新为：http://ots.aliyuncs.com
- 新加入对Mono的支持。

- 修复了SDK中的几处Bug，使其运行更稳定。

2013/04/10- OSS：

- 添加了Object分块上传（Multipart Upload）功能。
- 添加了Copy Object功能。
- 添加了生成预签名URL的功能。
- 分离出IOss接口，并由OssClient继承此接口。

2012/10/10- OSS:

- 将默认的OSS服务地址更新为：http://oss.aliyuncs.com

2012/09/05- OSS:

- 解决ListObjects时Prefix等参数无效的问题。

2012/06/15- OSS：

- 首次加入对OSS的支持。包含了OSS Bucket、ACL、Object的创建、修改、读取、删除等基本操作。
- OTS：
- OTSClient.GetRowsByOffset支持反向读取。
- 加入对特定请求错误的自动处理机制。
- 增加HTML格式的帮助文件。

2012/05/16- OTS

- Client.GetRowsByRange支持反向读取。

2012/03/16- OTS

- 访问接口，包括对表、表组的创建、修改和删除等操作，对数据的插入、修改、删除和查询等操作。
- 访问的客户端设置，如果代理设置、HTTP连接属性设置等。
- 统一的结构化异常处理。

NodeJs SDK开发包

OSS NodeJs SDK

OSS NodeJs SDK

github链接地址如下:NodeJs SDK