阿里云开放存储服务

ALIYUN OPEN STORAGE SERVICE

JAVA SDK 参考手册

(RELEASE 2.0.0)

2014.11.13

目录

1. 前言

1.1. 简介

本文档主要介绍 OSS Java 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 Access Key ID 和 Access Key Secret。文中的 ID 指的是 Access Key ID，KEY 指的是 Access Key Secret。如果您还没有开通或者还不了解 OSS，请登录 http://www.aliyun.com/product/oss 获取更多的帮助。

1.2. 版本更迭

此文档针对 OSS Java SDK 2.0.0 版本，此版本中变更了之前版本的依赖，使用之前版本 SDK 的程序在使用此版本 Java SDK 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.*。举例如下：

之前使用的 Java SDK 版本

```
import com.aliyun.openservices.ClientConfiguration;
import com.aliyun.openservices.ClientException;
import com.aliyun.openservices.ServiceException;
import com.aliyun.openservices.oss.OSSClient;
import com.aliyun.openservices.oss.OSSErrorCode;
import com.aliyun.openservices.oss.OSSException;
```

使用 2.0.0 Java SDK 版本

```
import com.aliyun.oss.ClientConfiguration;
import com.aliyun.oss.ClientException;
import com.aliyun.oss.ServiceException;
import com.aliyun.oss.OSSClient;
import com.aliyun.oss.OSSErrorCode;
import com.aliyun.oss.OSSException;
```

2. 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。

2.1. 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>";

        // 初始化一个 OSSClient
        OSSClient client = new OSSClient(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，为访问 OSS 做签名验证。关于 OSSClient 的详细介绍，参见 OSSClient 。

2.2. 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(accessKeyId, accessKeySecret);

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

关于 Bucket 的命名规范，参见 Bucket 命名规范。

2.3. 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(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。

Put Object 请求处理成功后，OSS 会将收到文件的 MD5 值放在返回结果的 ETag 中。用户可以根据 ETag 检验上传的文件与本地的是否一致。关于 Object 的命名规范，参见 Object 命名规范 。关于上传 Object 更详细的信息，参见 上传 Object 。

2.4. 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(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。

2.5. 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(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。

3. 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。

3.1. 新建 OSSClient

新建一个 OSSClient 很简单，如下面代码所示：

```
String key = "<key>";
String secret = "<secret>";
OSSClient client = new OSSClient(key, secret);
```

上面的方式使用默认域名作为 OSS 的服务地址，如果您想自己指定域名，可以传入 endpoint 参数来指定。

```
String key = "<key>";
String secret = "<secret>";
String endpoint = "http://oss.aliyuncs.com";
OSSClient client = new OSSClient(endpoint, accessKeyId, accessKeySecret);
```

3.2. 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	
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50
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

4. 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。

4.1. 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

4.2. 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。

4.3. 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());
}
```

4.4. 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");
}
```

4.5. 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。

5. 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。

5.1. 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

5.2. 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。

Put Object 请求处理成功后，OSS 会将收到文件的 MD5 值放在返回结果的 ETag 中。用户可以根据 ETag 检验上传的文件与本地的是否一致。

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 。

用户自定义元数据

OSS 支持用户自定义元数据来对 Object 进行描述。比如：

```
// 设置自定义元数据 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”的元数据。当用户下载此 Object 的时候，此元数据也可以一并得到。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”

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 Object 的分块上传 这一章中做介绍。

5.3. 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 false，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。

扩展参数

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。

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名返回在 CommonPrefixes 部分，子文件夹下递归的文件和文件夹不被显示.

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。

5.4. 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 以及自定义的元数据。

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);
```

5.5. 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);
}
```

5.6. 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 信信息

6. 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。
- 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。

下面我们将一步步介绍怎样实现 Multipart Upload

6.1. 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(bucketName, key);
InitiateMultipartUploadResult initiateMultipartUploadResult =
lient.initiateMultipartUpload(initiateMultipartUploadRequest);
// 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，强烈推荐用户在收到 OSS 的返回请求后，用该 MD5 值验证上传数据的正确性。
- 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 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(), partETags);

// 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。

6.2. 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);
```

6.3. 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。

6.4. 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。

7. 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。

7.1. 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：1、每个 bucket 最多只能使用 10 条规则。2、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。

7.2. 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());
      }
}
```

结果为一个 CORSRule 类的 list。遍历以后进行解析便可。
需要注意的是只有 bucket 的拥有者才能获取 CORS 规则，否则会出现错误码：AccessDenied。

7.3. 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。

8. 生成签名 URL

如果您想把自己的资源发放给第三方用户访问，但是又不想开放 Bucket 的读权限，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。

8.1. 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。

8.2. 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。

8.3. 添加用户自定义参数（UserMetadata）

如果您想使用签名的 URL 上传 Object，并指定 UserMetadata 等参数，可以这样做：

```
// 创建请求
GeneratePresignedUrlRequest generatePresignedUrlRequest = new GeneratePresignedUrlRequest(bucketName, key);

// HttpMethod 为 PUT
generatePresignedUrlRequest.setMethod(HttpMethod.PUT);

// 添加 UserMetadata
generatePresignedUrlRequest.addUserMetadata("key", "value");

// 生成签名的 URL
URL url = client.generatePresignedUrl(bucketName, key, expiration);
```

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 UserMetadata 的信息。

关于如何在 Http 请求中设置 UserMetadata 等参数，可以参考 OSS REST API 文档 中的相关内容。

9. 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 。

9.1. 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。

9.2. 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 InvalidBucketName	Access Key ID 不存在 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

错误码	描述
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