对象存储

(Object-Oriented Storage,OOS)

Android SDK 开发者指南 V1.0

中国电信股份有限公司

云计算分公司

目录

1. 简介

本文档主要介绍 OOS Android SDK 的安装和使用。本文档假设您已经开通了对象存储 OOS 服务，并创建了 AccessKeyId 和 SecretKey。OOS API 服务端地址请参见 OOS 开发者文档。

2. Android 开发工具包的设置选项

要开始使用 OOS Mobile SDK for Android，您可以设置该开发工具包并开始构建一个新项目，也可以将该开发工具包与现有项目集成。

2.1. 先决条件

您需要先获得以下项，然后才能使用 OOS Mobile SDK for Android:

1. 一个 OOS 账户
2. Android 2.3.3 (API 级别 10) 或更高版本 (有关 Android 平台的更多信息，请参阅 Android 开发人员)
3. Android Studio 或 适用于 Eclipse 的 Android 开发工具

2.2. 使用开发工具包（Android SDK)

满足先决条件后，您需要执行以下操作来开始使用该开发工具包:

1. 获取 OOS Mobile SDK for Android。
2. 在 AndroidManifest.xml 文件中设置权限。
3. 获取 OOS 凭证。

2.2.1. 获取 OOS Mobile SDK for Android

要获取 JAR 文件，请下载开发工具包。该开发工具包存储在名为

oos-android-sdk-#-#-# 的压缩文件中，其中 #-#-# 表示版本号。

1、 如果使用的是 Android Studio:

在“Project”视图中，将 oos-android-sdk-#-#-#-core.jar 以及您的项目将使用的 oos-android-sdk-#-#-#-s3.jar 文件拖动到 apps/libs 文件夹中。它们将自动包含在生成路径中。然后，将您的项目与 Gradle 文件同步。

2、 如果使用的是 Eclipse:

将 oos-android-sdk-#-#-#-core.jar 文件以及您的项目使用的

oos-android-sdk-#-#-#-s3.jar 文件拖动到 libs 文件夹中。它们将自动包含在生成路径中。

2.2.2. 在您的清单中设置权限

将以下权限策添加到您的 AndroidManifest.xml

```
<uses-permission android:name="android.permission.INTERNET" />
```

2.2.3. 获取 OOS 凭证

AccessKeyId 和 SecretKey 是您访问 OOS 的密钥，OOS 会通过它来验证您的资源请求，请妥善保管。您可以在对象存储的控制台—账户管理—API 密钥管理页面中查看到 AccessKeyId 和 SecretKey。密钥分为主密钥和普通密钥两种类型，每个用户可以拥有多个主密钥和普通密钥，两者的区别是：

1. 主密钥用于生成普通密钥，每个账户必须至少拥有一个主密钥。
2. 密钥可以被禁止使用，或者启用。当账户的主密钥只剩下一个时，该密钥不能被禁用或者删除。
3. 用户可以将普通密钥设置成为主密钥。
4. 普通密钥不能创建，删除，修改 bucket 属性。
5. 普通密钥只能操作以自己 AccessKey 开头的 Object，包括创建，删除，下载 Object 等操作。

   例如：普通 AccessKey 为 e67057e798af03040565，那么该 AccessKey 只能创建以 e67057e798af03040565 开头的 Object 名。
6. 普通密钥可以 list objects，但是参数 prefix 必须以 AccessKey 开头，即普通密钥只能 list 以自己的 AccessKey 开头的 Object。

在使用 SDK 访问 AccessKey 相关的 API 时，需要 setEndpointIAM 的 Endpoint，具体 IAM Endpoint 列表请参见开发者文档。

3. 配置 SDK

3.1. 使用 SDK 访问服务器

首先，创建 Client 对象，用于向服务器端发送请求，Andorid 代码如下：

```
AmazonS3 client= new AmazonS3Client(new
PropertiesCredentials(S3Sample.class.getResourceAsStream("AwsCreden
tials.properties")));
```

其中 AwsCredentials.properties 用于存储 accessKey 和 secretKey(此信息可以在控制台—账号管理—API 密钥管理中查看到)，例如：

```
accessKey =test
secretKey =test
```

http://oos.ctyunapi.cn 是 OOS 服务器的地址。

然后就可以调用 SDK 中的方法，对 Service,Bucket,Object 进行 PUT, GET, DELETE, HEAD 等操作。

3.2. SDK FAQ

1. 如何设置请求的重试功能

在创建 AmazonS3Client 时，可以配置客户端发送请求的最大重试次数，当服务端返回 5XX 错误，或连接超时等错误信息时，AmazonS3Client 会自动重发请求。配置方法如下：

```
    ClientConfiguration cc = newClientConfiguration();
    //这里设置重试两次，如果不设置的话，默认重试三次
    cc.setMaxErrorRetry(2);
    AmazonS3Client oosclient = newAmazonS3Client(new AWSCredentials()
{
    public String getAWSAccessKeyId() {
        return"yourAccessKey";
    }
    public String getAWSSecretKey() {
        return"yourSecretKey";
    }}, cc);
```

2. 如何设置超时时间

在创建 AmazonS3Client 时，可以配置客户端到服务器端的连接超时和 socket 超时时间，代码示例如下：

```
ClientConfiguration cc = newClientConfiguration();
cc.setConnectionTimeout(30*1000);//设置连接的超时时间，单位毫秒
cc.setSocketTimeout(30*1000) ;//设置 socket 超时时间，单位毫秒
AmazonS3Client oosclient = newAmazonS3Client(new AWSCredentials() {
public String getAWSAccessKeyId() {
    return"yourAccessKey";
}
public String getAWSSecretKey() {
    return"yourSecretKey";
}}, cc);
```

3. 如何设置 https 的 endpoint

通过 https 访问 OOS 服务的代码示例如下：

```
  System.setProperty("com.amazonaws.sdk.disableCertChecking",
"true");
  oosclient.setEndpoint("https://oos.ctyunapi.cn");//设置通过 https 方
式访问 oos 服务
```

4. 代码示例

4.1. OOSCredentials.properties

用于存储用户名和密码，例如：

```
  accessKey = yourAccessKey
  secretKey = yourSecretKey
```

4.2. Bucket 操作

4.2.1. Put Bucket

Put 操作用来创建一个新的 bucket。只有在 OOS 中注册的用户才能创建一个 新的 bucket，匿名请求无效，创建 bucket 的用户将是 bucket 的拥有者。

Bucket 的命名方式中并不是支持所有的字符，OOS 中 bucket 的 name 长度为

63 个字符以内，只支持小写字母数字，点(.)，以及横杠(-)。

```
    try {
          client.createBucket(bucketName);
       } catch (AmazonServiceException ase) {
          System.out.println("Caught an AmazonServiceException, which means
your request made it to OOS, " + "but was rejected with an error response
for some reason.");
          System.out.println("Error Message: " + ase.getErrorCode());
          System.out.println("Error Code:      " + ase.getErrorCode());
          System.out.println("Request ID:     " + ase.getRequestId());
       } catch (AmazonClientException ce) {
          System.out.println("Caught an ClientException, which means the
client encountered " + "a serious internal problem while trying to
communicate with OOS, " + "such as not being able to access the network.");
          System.out.println("Error Message: " + ce.getMessage());
       }
```

4.2.2. Set/Get Bucket ACL

这个 Get 操作用来获取 bucket 的 ACL 信息，用户必须对改 bucket 有读权限。

```
try {
CreateBucketRequest createBucketRequest = new.
                CreateBucketRequest(bucketName);
          //设置创建 bucket 的 ACL(Access Control List)
createBucketRequest.setCannedAcl(CannedAccessControlList.PublicRead);
client.createBucket(createBucketRequest);
GetBucketAclRequest getBucketAclRequest = new GetBucketAclRequest(
                bucketName);
AccessControlList acl = client.getBucketAcl(getBucketAclRequest);
for (Grant grant : acl.getGrants())
    assertEquals(grant.getPermission(), Permission.Read);
} catch (AmazonServiceException ase) {
          System.out.println("Caught an AmazonServiceException, which
means your request made it to OOS, " + "but was rejected with an error
response for some reason.");
          System.out.println("Error Message: " + ase.getErrorCode());
          System.out.println("Error Code:      " + ase.getErrorCode());
          System.out.println("Request ID:     " + ase.getRequestId());
} catch (AmazonClientException ce) {
          System.out.println("Caught an ClientException, which means the
client encountered " + "a serious internal problem while trying to
communicate with OOS, "
+ "such as not being able to access the network.");
          System.out.println("Error Message: " + ce.getMessage());
        }
```

4.2.3. Put/Get Bucket Policy

如果 bucket 已经存在了 Policy，此操作会替换原有 Policy。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。如果 bucket 没有 policy，Get 时返回 404，NoSuchPolicy 错误。

```
public void PutGetPolicyBucket() throws AmazonServiceException,
        AmazonClientException, IOException {
    Policy policy = newAPolicy(bucketName);
    client.setBucketPolicy(bucketName, policy.toJson());
    System.out.println(policy.toJson());
    BucketPolicy bucketPolicy = client.getBucketPolicy(bucketName);
    System.out.println(bucketPolicy.getPolicyText());
}

private Policy newAPolicy(String bucketName) {
    Policy policy = new Policy();
    policy.setId("1");
    Collection<Statement> statements = new ArrayList<Statement>();
    Statement s = new Statement(Effect.Allow);
    Collection<Action> actions = new ArrayList<Action>();
    Collection<Principal> principals = new ArrayList<Principal>();
    Collection<Resource> resources = new ArrayList<Resource>();
    List<Condition> conditions = new ArrayList<Condition>();
    actions.add(S3Actions.AllS3Actions);
    principals.add(Principal.AllUsers);
    resources.add(new Resource("arn:aws:s3:::" + bucketName + "/*"));
    conditions.add(new StringCondition(StringComparisonType.StringLike,
            "aws:Referer", "http://yourwebsitename.com/*"));
    s.setActions(actions);
    s.setPrincipals(principals);
    s.setResources(resources);
    s.setConditions(conditions);
    statements.add(s);
    policy.setStatements(statements);
    return policy;
  }
```

4.2.4. Put/Get Bucket WebSite

如果 bucket 已经存在了 website，此操作会替换原有 website。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。

WebSite 功能可以让用户将静态网站存放到 OOS 上。对于已经设置了 WebSite 的 Bucket，当用户访问 http://bucketName.oos-website-cn.oos.ctyunapi.cn 时，会跳转到用户指定的主页，当出现 4**错误时，会跳转到用户指定的出错页面。

```
    public void putGetDeleteWebSite() throws AmazonServiceException,
            AmazonClientException, IOException {
        client.createBucket(bucketName);
        BucketWebsiteConfiguration configuration = new
BucketWebsiteConfiguration("index.html", "error.html");
        client.setBucketWebsiteConfiguration(bucketName, configuration);
        BucketWebsiteConfiguration website = client
                .getBucketWebsiteConfiguration(bucketName);
        System.out.println(website.getIndexDocumentSuffix());
        System.out.println(website.getErrorDocument());
        client.deleteBucketWebsiteConfiguration(bucketName);
    }
```

4.2.5. Put/Get Bucket Trigger

Bucket trigger 即一个向异地资源池同步的触发器。

当客户端向本地资源池的 bucket 上传对象时，OOS 可以根据配置的触发器策略，自动将对象同步到异地资源池中。一个 bucket 可以 配置多个触发器，但只能有一个是默认的触发器。如果客户端要使用非默认的触发器上传对象，需要在 put object 时，加上请求头 x-ctyun-trigger，值是指定的 TriggerName。

只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。

```
    public BucketTest() {
        client = Common.getClient();
    }
    public void putGetBucketTrigger()
            throws AmazonServiceException, AmazonClientException {
        CtyunBucketTriggerConfiguration config = new
CtyunBucketTriggerConfiguration();
        CtyunTriggerMetadata metadata = new CtyunTriggerMetadata();
        // Trigger 的名称，是字母或数字，不能包含特殊符号，最多 20 个字符。
        metadata.setTriggerName("your trigger name");
        metadata.setDefault(false);// 是否是默认的 trigger
        CtyunRemoteSite remote = new CtyunRemoteSite();
        remote.setRemoteAK("your remote AK");// 异地资源池的 AccessKey
        remote.setRemoteSK("your remote SK");// 异地资源池的secretKey
        remote.setRemoteBucketName("remote bucket");// 异地资源池的 bucketName
        remote.setRemoteEndPoint("remote endpoint");// 异地资源池的 endpoint
        // CtyunRemoteSite 异地资源池的相关配置，可以配置向多个异 地资源池同步数据，
        // 每个异地资源池对应一 个 RemoteSite。如果不配置 RemoteSite，即 不复制到其他
资源池。
        List<CtyunRemoteSite> remoteSiteList = new
ArrayList<CtyunRemoteSite>();
        remoteSiteList.add(remote);
        metadata.setRemoteSites(remoteSiteList);
        List<CtyunTriggerMetadata> triggerMetadata = new
ArrayList<CtyunTriggerMetadata>();
        triggerMetadata.add(metadata);
        config.setTriggers(triggerMetadata);
        CtyunSetBucketTriggerConfigurationRequest trigger = new
CtyunSetBucketTriggerConfigurationRequest(
                "bucketName", config);
        client.ctyunSetBucketTrigger(trigger);
        config = client.ctyunGetBucketTrigger("bucketName");
    }
```

4.2.6. Put/Get Bucket LifeCycle

存储在 OOS 中的对象有时需要有生命周期。比如，用户可能上传了一些周期性的日志文件到 bucket 中，一段时间后，用户可能不需要这些日志对象了。 之前，用户需要

自己手动删除这些不用的对象，现在可以使用对象到期功能来指定 bucket 中对象的生命周期。

当对象的生命周期结束时，OOS 会异步删除他们。生命周期中配置的到期时间和实际删除时间之间可能会有一段延迟。对象到期后，用户将不会再为到期的对象付费。

用户可以通过在 bucket 中配置生命周期，来为对象设置到期时间。一个生 命周期的配置中最多可以包含 100 条规则。每个规则指定了对象的前缀和生命周期，生命周期是指从对象创建开始到被删除之前的天数。生命周期的值必须是个正整数。OOS 通过将对象的创建时间加上生命周期时间来计算到期时间，并且 将时间近似到下一天的 GMT 零点时间。比如，一个对象是 GMT 2016 年 1 月 15 日 10:30 分创建的，生命周期是 3 天，那么对象的到期时间是 GMT 2016 年 1 月 19 日 00:00。当重写一个对象时，OOS 将以最后更新时间为准，来重新计算到期时间。

当用户为 bucket 设置了生命周期时，这些规则将同时应用于已有对象和之后新创建的对象。比如，用户今天增加了一个生命周期的配置，指定某些前缀的 对象 30 天后过期，那么 OOS 将会把满足条件的 30 天前创建的对象都加入到待删除队列中。

用户可以使用 GET, HEAD API 来查询对象的到期时间，这些接口会通过响应头来返回对象的到期时间信息。

用户可以使用 OOS access logs 来查询对象是何时被删除的。OOS 删除到期 对象后，会在 access logs 中记录一条日志，操作项是"OOS.EXPIRE.OBJECT"。

Put Bucket Lifecycle 接口用于设置 bucket 的生命周期，如果生命周期的配置已经存在，将会被替换。用户可以通过设置生命周期，来让 OOS 删除过期的对象。

```
    public void putGetBucketLifeCycle()
            throws AmazonServiceException, AmazonClientException {
        BucketLifecycleConfiguration config = new
BucketLifecycleConfiguration();
        BucketLifecycleConfiguration.Rule rule = new
BucketLifecycleConfiguration.Rule();
        rule.setPrefix("object prefix");// 指明要使用规则的对象前缀，最长 1024
个字符
        // status 如果是 Enabled，那么规则立即生效。如果是 Disabled，那么规则不会
生效。
        rule.setStatus(BucketLifecycleConfiguration.ENABLED);
        // 以天数来描述生命周期，值是正整数； 如果使用 setExpirationDate(Date
data)方法
        // 生成时间早于此时间的对象将被认为是过期对象 ;Days 和 Date 二选一
        rule.setExpirationInDays(30);
        //最多包含 100 个规则
        List<BucketLifecycleConfiguration.Rule> rules = new
ArrayList<BucketLifecycleConfiguration.Rule>(100);
        rules.add(rule);
        config.withRules(rules);
        client.setBucketLifecycleConfiguration("bucketName", config);
        BucketLifecycleConfiguration getConfig =
client.getBucketLifecycleConfiguration("bucketName");
        System.out.println(getConfig.toString());
      }
```

4.2.7. Put/Get Bucket logging

如果 bucket 已经存在了 logging，put 操作会替换原有 logging。只有 bucket 的 owner 才能执行此操作，否则会返回 403 AccessDenied 错误。

日志名称： TargetPrefixGMTYYYY-mm-DD-HH-MM-SS-唯一字符串

其中 YYYY，mm，DD，HH，MM，SS 分别代表日志发送时的年，月，日， 小时，分钟，秒。TargetPrefix 是用户在启动 Bucket 日志时配置的。唯一字符串部分没有实际含义，可以被忽略。

系统不会删除旧的日志文件。用户可以自己删除以前的日志文件，可以在 List Object 时指定 prefix 参数，挑选出旧的日志文件，然后删除。

当客户端的请求到达后，日志记录不会被立刻推送到 TargetBucket 中，会延 迟一段时间。

```
public void putGetBucketLogging() throws AmazonClientException,
AmazonServiceException{
        BucketLoggingConfiguration config = new
BucketLoggingConfiguration();
        //指定要保存 log 的 bucket，OOS 会向此 bucket 存储日志。可以设置任意一个你拥有
的 bucket 作为 TargetBucket，包括启动日志的 bucket 本身。你也可以设置将多个 bucket
的日志存放到一个 TargetBucket 中，在这种 情况下，你需要为每个源 bucket 设置不同 的
TargetPrefix，以便不同 bucket 的 log 可 以被区分出来。
        config.setDestinationBucketName("destinationBucketName");
        //生成的 log 文件将以此为前缀命名
        config.setLogFilePrefix("logFilePrefix");
        //在异地互备过程中，目标资源池的 log 会保 存到此 bucket 中，oos 会向此 bucket
存储日 志
config.setCtyunTriggerDestinationBucketName("triggerDestBucketName");
        //在异地互备过程中，目标资源池的 log 文件 将以此为前缀命名
config.setCtyunTriggerDestinationLogFilePrefix("triggerDestinationLogFilePr
efix");
        //在异地互备过程中，源资源池的 log 会保存 到此 bucket 中，oos 会向此 bucket 存
储日志
        config.setCtyunTriggerSourceBucketName("triggerSourceBucketName");
        //在异地互备过程中，源资源池的 log 文件将 以此为前缀命名
config.setCtyunTriggerSourceLogFilePrefix("triggerSourceLogFilePrefix");
        SetBucketLoggingConfigurationRequest request = new
SetBucketLoggingConfigurationRequest("bucketName",config);
        client.setBucketLoggingConfiguration(request);
        BucketLoggingConfiguration configuration =.
        client.getBucketLoggingConfiguration("bucketName");
        System.out.println(configuration.toString());
      }
```

4.3. Object 操作

4.3.1. Put/Get/Delete Object

Put 操作用来向指定 bucket 中添加一个对象，要求发送请求者对该 bucket 有写权限，用户必须添加完整的对象。

GET 操作用来检索在 OOS 中的对象信息，执行 GET 操作，用户必须对 object 所在的 bucket 有读权限。如果 bucket 是 public read 的权限，匿名用户也可以通过非授权的方式进行读操作。

```
   public void putGetObject() throws AmazonClientException,
AmazonServiceException,IOException {
       File sourceFile =new File("sourceFilePath");
       PutObjectRequest putObjectRequest = new PutObjectRequest(bucketName,
              objectName, sourceFile);
       ObjectMetadata metadata = new ObjectMetadata();
       //按照请求/回应的方式用来定义缓存行为
       metadata.setCacheControl("no-cache");
       //指出对象的描述性的信息
       metadata.setContentDisposition("attachment; filename=testing.txt");
       //用字节的方式定义对象的大小 必填项
       metadata.setContentLength(sourceFile.length());
       metadata.setContentType("text/plain");
       //用户的元数据，当用户检索时，它将会和对象一起被存储并返回。
       //PUT 请求头大小限制为 8KB。在 PUT 请求头中， 用户定义的元数据大小限制为 2KB。
       metadata.addUserMetadata("test", "user metadata");
       putObjectRequest.setMetadata(metadata);
       client.putObject(putObjectRequest);
       GetObjectRequest getObjectRequest = new GetObjectRequest(bucketName,
              objectName);
       S3Object object = client.getObject(getObjectRequest);
       System.out.println(object.getKey());
       client.deleteObject(bucketName, objectName);
     }
```

4.3.2. Initial Multipart Upload

本接口初始化一个分片上传(Multipart Upload)操作，并返回一个上传 ID， 此 ID 用来将此次分片上传操作中上传的所有片段合并成一个对象。用户在执行每一次子上传请求(见 Upload Part)时都应该指定该 ID。用户也可以在表示整 个分片上传完成的最后一个请求中指定该 ID。或者在用户放弃该分片上传操作时指定该 ID。

```
        InitiateMultipartUploadRequest request = new
InitiateMultipartUploadRequest(
                bucketName, objectName);
        InitiateMultipartUploadResult result =
client.initiateMultipartUpload(request);
        //分片上传唯一标识 ID
        String uploadId = result.getUploadId();
```

4.3.3. Upload Part

该接口用于实现分片上传操作中片段的上传。

在上传任何一个分片之前，必须执行 Initial Multipart Upload 操作来初始化 分片上传操作，初始化成功后，OOS 会返回一个上传 ID，这是一个唯一的标识， 用户必须在调用 Upload Part 接口时加入该 ID。

分片号 PartNumber 可以唯一标识一个片段并且定义该分片在对象中的位置， 范围从 1 到 10000。如果用户用之前上传过的片段的分片号来上传新的分片，之前的分片将会被覆盖。

除了最后一个分片外，所有分片的大小都应该不小于 5M，最后一个分片的大小不受限制。

为了确保数据不会由于网络传输而毁坏，需要在每个分片上传请求中指定 Content-MD5 头，OOS 通过提供的 Content-MD5 值来检查数据的完整性，如果不匹配，则会返回一个错误信息。

代码片：

```
UploadPartRequest uploadRequest = new UploadPartRequest();
//待上传源文件
uploadRequest.setFile(file);
//part number 此分片与其他分片的相对位置编号。 范围从 1 到 10000
uploadRequest.setPartNumber(1);
uploadRequest.setUploadId(uploadId);
uploadRequest.setBucketName(bucketName);
uploadRequest.setKey(objectName);
//此分片的大小。除了最后一个分片外，所有分片的大小都应该不小于 5M
uploadRequest.setPartSize(partSize);
UploadPartResult uploadResult1 = client.uploadPart(uploadRequest);
uploadRequest.setFile(file);
uploadRequest.setPartNumber(2);
uploadRequest.setUploadId(uploadId);
uploadRequest.setBucketName(bucketName);
uploadRequest.setKey(objectName);
uploadRequest.setPartSize(file.length() - partSize);
//本分片在待上传文件的起始位置，如果不指定，将从文件头开始取。
uploadRequest.setFileOffset(partSize);
UploadPartResult uploadResult2 = client.uploadPart(uploadRequest);
```

4.3.4. Complete Multipart Upload

该接口通过合并之前的上传片段来完成一次分片上传过程。

用户首先初始化分片上传过程，然后通过 Upload Part 接口上传所有分片。 在成功将一次分片上传过程的所有相关片段上传之后，调用这个接口来结束分片上传过程。当收到这个请求的时候，OOS 会以分片号升序排列的方式将所有片 段依次拼接来创建一个新的对象。在这个 Complete Multipart Upload 请求中，用户需要提供一个片段列表。同时，必须确保这个片段列表中的所有片段必须是已经上传完成的，Complete Multipart Upload 操作会将片段列表中提供的片段拼接起来。对片段列表中的每个片段，需要提供该片段上传完成时返回的 ETag 头的值和对应的分单击此处输入文字。片号。

处理一次 Complete Multipart Upload 请求可能需要花费几分钟时间。OOS 在 处理这个请求之前会发送一个值为 200 响应头。在处理这个请求的过程中，OOS 每隔一段时间就会发送一个空格字符来防止连接超时。因为一个请求在初始的 200 响应已经发出之后仍可能失败，用户需要检查响应内容以判断请求是否成功。

注意，如果 Complete Multipart Upload 请求失败，应用程序应该尝试重新发送该请求。

由于 Complete Multipart Upload 请求可能需要花费几分钟时间，所以 OOS 提供了不合并片段也可以读取 Object 内容的功能。在没有调用 Complete Multipart Upload 接口合并片段时，也可以通过调用 Get Object 接口来获取文件内容，OOS 会根据最近一次创建的 uploadId，以分片号升序的方式顺序读取片段内容，返回给客户端。但此时不能返回整个文件的 ETag 值。

代码片：

```
        List<PartETag> parts = new ArrayList<PartETag>();
        //封装分片号和每个分片上传后OOS返回的etag值。 分片号应升序排序
        PartETag e = new PartETag(1, uploadResult1.getPartETag().getETag());
        parts.add(e);
        e = new PartETag(2, uploadResult2.getPartETag().getETag());
        parts.add(e);
        CompleteMultipartUploadRequest request4 = new
CompleteMultipartUploadRequest(
                bucketName, objectName, uploadId, parts);
        client.completeMultipartUpload(request4);
```

4.3.5. 分片上传完整代码示例

```
public void multipartUpload() throws AmazonClientException,
AmazonServiceException,IOException {
        File file = new File("file path");
        //每片大小设置为5M
        long partSize = 5 * 1024 * 1024;
        InitiateMultipartUploadRequest request = new
InitiateMultipartUploadRequest(bucketName, objectName);
        InitiateMultipartUploadResult result =
client.initiateMultipartUpload(request);
        //分片上传唯一标识 ID
        String uploadId = result.getUploadId();
        UploadPartRequest uploadRequest = new UploadPartRequest();
        //待上传源文件
        uploadRequest.setFile(file);
        //part number 此分片与其他分片的相对位置编号。 范围从 1 到 10000
        uploadRequest.setPartNumber(1);
        uploadRequest.setUploadId(uploadId);
        uploadRequest.setBucketName(bucketName);
        uploadRequest.setKey(objectName);
        //此分片的大小。除了最后一个分片外，所有分片的大小都应该不小于 5M
        uploadRequest.setPartSize(partSize);
        UploadPartResult uploadResult1 = client.uploadPart(uploadRequest);
        uploadRequest.setFile(file);
        uploadRequest.setPartNumber(2);
        uploadRequest.setUploadId(uploadId);
        uploadRequest.setBucketName(bucketName);
        uploadRequest.setKey(objectName);
        uploadRequest.setPartSize(file.length() - partSize);
        //本分片在待上传文件的起始位置，如果不指定，将从文件头开始取。
        uploadRequest.setFileOffset(partSize);
        UploadPartResult uploadResult2 = client.uploadPart(uploadRequest);
        List<PartETag> parts = new ArrayList<PartETag>();
        //封装分片号和每个分片上传后OOS返回的etag值。 分片号应升序排序
        PartETag partEtag = new PartETag(1,
uploadResult1.getPartETag().getETag());
        parts.add(partEtag);
        partEtag = new PartETag(2, uploadResult2.getPartETag().getETag());
        parts.add(partEtag);
        CompleteMultipartUploadRequest completeMulPartUploadrequest = new
CompleteMultipartUploadRequest(bucketName, objectName, uploadId, parts);
        client.completeMultipartUpload(completeMulPartUploadrequest);
        S3Object object = client.getObject(bucketName, objectName);
        System.out.println(object.getKey());
    }
```

4.3.6. Abort Multipart Upload

该接口用于终止一次分片上传操作。分片上传操作被终止后，用户不能再通过上传 ID 上传其它片段，之前已上传完成的片段所占用的存储空间将被释放。 如果此时任何片段正在上传，该上传过程也可能不会成功。因此，为了释放所有片段所占用的存储空间，可能需要多次终止分片上传操作。

代码片：

```
AbortMultipartUploadRequest abortMultipartUploadRequest = new
     AbortMultipartUploadRequest("bucketName","objectName",uploadId);
client.abortMultipartUpload(abortMultipartUploadRequest);
```

4.3.7. Delete Multiple Objects

批量删除 Object 功能支持用一次请求删除一个 bucket 中的多个 object。 如果你知道你想删除的 object 名字，此功能可以批量删除这些 object，而不用发 送多个单独的删除请求。

对于每个 Object，OOS 都会返回删除的结果，成功或者失败。注意，如果请求中的 object 不存在，那么 OOS 也会返回删除成功。

批量删除功能支持两种格式的响应，全面信息和简明信息。默认情况下， OOS 在响应中会显示全面信息，即包含每个 object 的删除结果。在简明信息模式下，OOS 只返回删除出错的 object 的结果。对于成功删除的 object，在响应中将不返回任何信息。

```
public void deleteObjects() throws AmazonClientException,
AmazonServiceException{
       DeleteObjectsRequest deleteObjectsRequest = new
DeleteObjectsRequest("bucketName");
       //使用简明信息模式来返回响应，当使用此元素时，需要指定true。
       deleteObjectsRequest.setQuiet(true);
       List<KeyVersion> keys = new
ArrayList<DeleteObjectsRequest.KeyVersion>();
       keys.add(new KeyVersion("objectName1"));
       keys.add(new KeyVersion("objectName2"));
       deleteObjectsRequest.setKeys(keys);
       //在简明信息模式下，OOS 只返回删除出错的 object 的结果
       DeleteObjectsResult result =
client.deleteObjects(deleteObjectsRequest);
       for(DeletedObject object : result.getDeletedObjects()) {
           System.out.println(object.getKey());
       }
     }
```

4.4. AccessKey 操作

创建一对普通的 AccessKey 和 SecretKey，默认的状态是 Active。只有主 key 才能执行此操作。

为保证账户的安全，SecretKey 只在创建的时候会被显示。请把 key 保存起来，比如保存到一个文本文件中。如果 SecretKey 丢失了，你可以删除 AccessKey， 并创建一对新的 key。默认情况下，每个账号最多创建 10 个 AccessKey。

```
public void createDeleteAKSK() {

       CreateAccessKeyResult createResult = client.ctyunCreateAccessKey();

       System.out.println(createResult.getAccessKey().getUserName());

       System.out.println(createResult.getAccessKey().getAccessKeyId());

       System.out.println(createResult.getAccessKey().getSecretAccessKey());

       //默认生成的是 Active 状态

       System.out.println(createResult.getAccessKey().getStatus());

       DeleteAccessKeyRequest delete = new DeleteAccessKeyRequest();

       delete.setAccessKeyId(createResult.getAccessKey().getAccessKeyId());

       client.ctyunDeleteAccessKey(delete);

     }
```