对象存储

(Object-Oriented Storage,OOS)

C SDK 开发者指南 V1.0

中国电信股份有限公司

云计算分公司

1. 简介

本文档主要介绍 OOS C 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。

本 SDK 使用符合 C89 标准的 C 语言实现。由于 C 语言的普适性，原则上此 SDK 可以跨所有主流平台，不仅可以直接在 C 和 C++ 的工程中使用，也可以用于与 C 语言交互性较好的语言中，例如 C# (使用 P/Invoke 交互)、Java (使用 JNI 交互)、Lua 等。

本开发指南假设开发者使用的开发语言是 C/C++，C-SDK 以动态库的方式提供，开发者可以直接将 SDK 的动态库链接到自己的工程中。

2. 先决条件

2.1. 前提

您需要先获得以下项，然后才能使用 OOS C SDK

1. 一个 OOS 账户
2. 主流的 Linux 操作系统，并安装 gcc 编译器。

2.2. 如何获取 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. 安装

OOS C SDK 使用 curl 进行网络操作，无论是作为客户端还是服务器端，都需要依赖 curl。OSS C SDK 使用 apr/apr-util 库解决内存管理以及跨平台问题，使用 minixml 库解析请求返回的 xml。OOS C SDK 并没有带上这三个外部库，您需要确认这三个库已经安装，并且将它们的头文件目录和库文件目录都加入到了项目中。

3.1. 第三方库下载以及安装

libcurl（建议 7.32.0 及以上版本）

请从这里下载，并参考 libcurl 安装指南安装。典型的安装方式如下：

```
./configure
   make
```

注意：

执行./configure 时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure --prefix=/your/install/path/

apr （建议 1.5.2 及以上版本）

请从这里下载，典型的安装方式如下：

```
./configure
make
make install
```

注意：

执行./configure 时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure --prefix=/your/install/path/

apr-util （建议 1.5.4 及以上版本）

请从这里下载，安装时需要注意指定--with-apr 选项，典型的安装方式如下：

```
./configure --with-apr=/your/apr/install/path
make
make install
```

注意：

- 执行./configure 时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure --prefix=/your/install/path/

- 需要通过--with-apr 指定 apr 安装目录，如果 apr 安装到系统目录下需要指定 --with-apr=/usr/local/apr/

minixml （建议 2.8 及以上版本）

请从这里下载，典型的安装方式如下:

```
./configure
make
make install
```

注意：

- 执行./configure 时默认是配置安装目录为/usr/local/，如果需要指定安装目录，请使用 ./configure --prefix=/your/install/path/

3.2. OOS C SDK 的安装

手动解压并拷贝动态库。命令如下：

```
tar -xzvf oos-c-sdk.tar.gz
cd oos-c-sdk/
cp -ar include/oos /usr/local/include
cp lib/liboos.so /usr/lib64/
```

4. 代码示例

4.1. 接口函数调用流程

每个接口调用需要完成同样的准备工作：

1. 初始化内存池 oos_initialize
2. 创建内存池 oos_pool_create

3. 创建 options 并设置 AccessKey、SecretKey 和 HostName。

4. 调用对应接口

5. 释放内初池

具体实例代码片：

```
    int main()
    {
        // 初始化内存池
        oos_initialize();
        oos_pool_t *p = NULL;
        request_options_t *options = NULL;
        oos_status_t *status = NULL;
        // 创建内存池
        oos_pool_create(&p, NULL);
        options = (request_options_t *)oos_pcalloc(p,
sizeof(request_options_t));
        options->pool = p;
        // 设置 options 信息
        oos_str_set(&options->access_key, ACCESS_KEY);
        oos_str_set(&options->secret_key, SECRET_KEY);
        oos_str_set(&options->endpoint, HOST_NAME);
        // 调用接口
        put__trigger_test(options, status);
        // 释放内存池
        oos_pool_destroy(p);
        oos_terminate();
        return 0;
    }
```

4.2. Bucket 操作

4.2.1. Put/Delete 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(-)。

代码片：

```
    void create_bucket_test(request_options_t *options,oos_status_t *status){
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status =
oos_create_bucket(options,"test",BUCKET_TYPE_PRIVATE,&resp_headers);
        printf("service code %d\n", status->code);
    }
    void delete_bucket_test(request_options_t *options,oos_status_t *status){
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status = oos_delete_bucket(options,"test",&resp_headers);
        printf("service code %d\n", status->code);
    }
```

4.2.2. Put/Get Bucket Policy

如果 bucket 已经设置了 Policy，Put 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。

代码片：

```
    void put_bucket_policy_test(request_options_t *options,oos_status_t
*status){
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        char text[] = "{\"Version\":\"2012-10-17\",\"Id\":\"http referer policy
example\",\"Statement\":[{\"Sid\":\"Allow get requests referred by
www.mysite.com \", \"Effect\":\"Allow\",\"Principal\":{ \"AWS\": [\"*\"] },
\"Action\":\"s3:*\", \"Resource\":\"arn:aws:s3:::example-bucket/*\",
\"Condition\":{\"StringLike\":{\"aws:Referer\":[\"http://www.mysite.com/*\"
]} }} ]}";
        status = oos_put_bucket_policy(options,
BUCKET_NAME,text,&resp_headers);
        printf("service code %d\n", status->code);
    }

    void get_bucket_policy_test(request_options_t *options,oos_status_t
*status){
        char* policy_text= oos_pcalloc(options->pool,2048);
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status =
oos_get_bucket_policy(options,"test",policy_text,&resp_headers);
        printf("policy text %s\n", policy_text);
        printf("service code %d\n", status->code);
    }
```

4.2.3. 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 4XX 错误时，会跳转到用户指定的出错页面。

代码片：

```
    void put_bucket_website_test(request_options_t *options,oos_status_t
*status){
        website_configuration_t* configweb;
        configweb = oos_pcalloc(options->pool,sizeof(*configweb));
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        oos_str_set(&configweb->suffix,"suffix.index.html");
        oos_str_set(&configweb->key,"key.index.html");
        status =
oos_put_bucket_website(options,BUCKET_NAME,configweb,&resp_headers);
        printf("service code %d\n", status->code);
    }

    void get_bucket_website_test(request_options_t *options,oos_status_t
*status){
        website_configuration_t* configweb;
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status = oos_get_bucket_website(options,BUCKET_NAME,&configweb,
&resp_headers);
        printf("service code %d\n", status->code);
        printf("website prefix %s\n", configweb->suffix.data);
    }
```

4.2.4. 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。

代码片：

```
    void put_bucket_trigger_test(request_options_t *options,oos_status_t
*status){
       trigger_configuration_t* config;
       config = oos_pcalloc(options->pool,sizeof(*config));

       oos_str_set(&config->triggerName,"trigger name");
       oos_str_set(&config->isDefault,"false");
       oos_str_set(&config->remontEndPoint,"remontEndPoint");
       oos_str_set(&config->replicaMode,"replicaMode");
       oos_str_set(&config->remoteBucketName,"remoteBucketName");
       oos_str_set(&config->remoteAK,"remoteAK");
       oos_str_set(&config->remoteSK,"remoteSK");
       trigger_configuration_t configs[1];
       memcpy((char *)&configs[0],(char *)config,sizeof(*config));
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       status =
oos_put_bucket_trigger(options,BUCKET_NAME,configs,1,&resp_headers);
       printf("service code %d\n", status->code);
    }

    void get_bucket_trigger_test(request_options_t *options,oos_status_t
*status){
       trigger_configuration_t* config ;
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       status =
oos_get_bucket_trigger(options,BUCKET_NAME,&config,&resp_headers);
       printf("service code %d\n", status->code);
       struct list_head *pos;
       list_for_each(pos, &config->node){
          trigger_configuration_t* tmp = list_entry(pos,struct
trigger_configuration_t,node);
          printf("triggerName: %s, isDefault: %s\n", tmp->triggerName.data,
tmp->isDefault.data);
       }
    }
```

4.2.5. 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。

代码片：

```
    void put_bucket_lifecycle_test(request_options_t *options, oos_status_t
*status){
        lifecycle_configuration_t* config;
        config = oos_pcalloc(options->pool, sizeof(*config));
        oos_str_set(&config->prefix,"test/");
        oos_str_set(&config->status,"Disabled");
        config->days=100;
        lifecycle_configuration_t configs[1];
        memcpy((char *)&configs[0],(char *)config,sizeof(*config));
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);

        status = oos_put_bucket_lifecycle(
             options,BUCKET_NAME,configs,1,&resp_headers);
        printf("service code %d\n", status->code);
    }

    void get_bucket_lifecycle_test(request_options_t *options,
             oos_status_t *status){
        lifecycle_configuration_t* config=NULL;
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       status = oos_get_bucket_lifecycle(options, BUCKET_NAME,
             &config,&resp_headers);
        printf("service code %d\n", status->code);

        struct list_head *pos;
        list_for_each(pos, &config->node){
           lifecycle_configuration_t* tmp = list_entry(pos,
               struct lifecycle_configuration_t,node);
           printf("ruleid: %s, prefix: %s\n", tmp->ruleid.data,
tmp->prefix.data);
        }
    }
```

4.2.6. 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。

代码片：

```
    void put_bucket_logging_test(request_options_t *options,oos_status_t
*status){
       logging_configuration_t* configlog;
       configlog = oos_pcalloc(options->pool,sizeof(*configlog));
       oos_str_set(&configlog->targetBucket,"test");
       oos_str_set(&configlog->targetPrefix,"test/");
       oos_str_set(&configlog->triggerTargetBucket,"triggerTargetBucket");
       oos_str_set(&configlog->triggerTargetPrefix,"triggerTargetPrefix/");
       oos_str_set(&configlog->triggerSourceBucket,"triggerSourceBucket");
       oos_str_set(&configlog->triggerSourcePrefix,"triggerSourcePrefix/");
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       status =
oos_put_bucket_logging(options,BUCKET_NAME,configlog,&resp_headers);
       printf("service code %d\n", status->code);
    }

    void get_bucket_logging_test(request_options_t *options,oos_status_t
*status){
       logging_configuration_t* config;
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       status =
oos_get_bucket_logging(options,BUCKET_NAME,&config,&resp_headers);
       printf("service code %d\n", status->code);
       printf("config targetPrefix  %s\n", config->targetBucket.data);
    }
```

4.3. Object 操作

4.3.1. Put/Get 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。

添加内存对象代码片：

```
    void put_buf_object_test(request_options_t *options,oos_status_t
*status){
       // status = put_buf_object(options,BUCKET_NAME,"test4.xml","hello
world!",sizeof("hello world!"));
       oos_table_t* medadata = oos_table_make(options->pool, 1);
       oos_table_set(medadata, "x-amz-meta-media_key1", "value1");
       oos_table_set(medadata, "x-amz-meta-media_key2", "value2");
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       status =
oos_put_object_from_buffer(options,BUCKET_NAME,"test6.xml","hello
world!",sizeof("hello world!"),medadata,&resp_headers);
       printf("service code %d\n", status->code);
       printf("service msg %s\n", status->error_msg);
       int pos = 0;
       const oos_array_header_t *tarr;
       const oos_table_entry_t *telts;
       if (resp_headers) {
          tarr = oos_table_elts(resp_headers);
          telts = (oos_table_entry_t*)tarr->elts;
          for (pos = 0; pos < tarr->nelts; ++pos) {
                printf("--!- %s:%s\n",telts[pos].key, telts[pos].val);
          }
       }
    }
```

获取内存对象代码片：

```
    void get_buf_object_test(request_options_t *options,oos_status_t
*status){
       oos_string_t* content = (oos_string_t*)oos_pcalloc(options->pool,
sizeof(*content));;
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       int pos = 0;
       const oos_array_header_t *tarr;
       const oos_table_entry_t *telts;
       oos_table_t* medadata = NULL;
       status =
oos_get_object_to_buffer(options,BUCKET_NAME,"test6.xml",content,
              &medadata,&resp_headers);
       if (medadata) {
           tarr = oos_table_elts(medadata);
           telts = (oos_table_entry_t*)tarr->elts;
           for (pos = 0; pos < tarr->nelts; ++pos) {
                  printf("--%s: %s\n",telts[pos].key, telts[pos].val);
           }
       }

       printf("service code %d\n", status->code);
       printf("service msg service %s\n", status->error_msg);
       printf("content %s\n", content->data);
    }
```

添加文件对象代码片：

```
    void put_file_object_test(request_options_t *options,oos_status_t
*status){
       oos_table_t* medadata = oos_table_make(options->pool, 1);
       oos_table_t* resp_headers = oos_table_make(options->pool, 1);
       oos_table_set(medadata, "x-amz-meta-media_key1", "value1");
       oos_table_set(medadata, "x-amz-meta-media_key2", "value2");
       status = oos_put_object_from_file(options,BUCKET_NAME,"你好.xml",
FILE_PATH,medadata,&resp_headers);
       printf("service code %d\n", status->code);
    }
```

获取文件对象代码片：

```
    void get_file_object_test(request_options_t *options,oos_status_t
*status){
        char* file_path ="/Users/test/temp/abc3.txt";
        int pos = 0;
        const oos_array_header_t *tarr;
        const oos_table_entry_t *telts;
        oos_table_t* medadata = NULL;
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status =
oos_get_object_to_file(options,BUCKET_NAME,"test5.xml",file_path,
&medadata,&resp_headers);
        if (medadata) {
            tarr = oos_table_elts(medadata);
            telts = (oos_table_entry_t*)tarr->elts;
            for (pos = 0; pos < tarr->nelts; ++pos) {
                   printf("%s:%s\n",telts[pos].key, telts[pos].val);
            }
        }
        printf("service code %d\n", status->code);
    }
```

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。

代码片：

```
   void init_multipart_upload_test(request_options_t *options,oos_status_t
*status){
      initiate_multipart_upload_result_t* result;
      oos_table_t* resp_headers = oos_table_make(options->pool, 1);
      oos_table_t* medadata = oos_table_make(options->pool, 1);
      oos_table_set(medadata, "x-amz-meta-media_key1", "value1");
      oos_table_set(medadata, "x-amz-meta-media_key2", "value2");
      status = oos_init_multipart_upload(options,BUCKET_NAME,"1132.txt",
          &result, medadata,&resp_headers);
      printf("service code %d\n", status->code);
      printf("upload id %s\n", result->uploadId.data);
   }
```

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：

```
    void upload_part_test(request_options_t *options, oos_status_t *status){
        FILE* file = NULL;
        size_t file_size;
        size_t part_size = 5*1024*1024;
        oos_string_t* etag = (oos_string_t*)oos_pcalloc(options->pool,
sizeof(*etag));
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);

        file = fopen("/Users/test/temp/123.txt", "rb");
        if (file == NULL) {
            fprintf(stderr, "open file %s failed : %m\n", "sdsfdsdfsd");
            return ;
        }

        fseek(file, 0, SEEK_END);
        file_size = ftell(file);
        fseek(file, 0, SEEK_SET);
        char* buff;
        buff = (char*)oos_pcalloc(options->pool, file_size);
        int a = fread(buff, 1,part_size, file);
        status =
oos_upload_part_from_buffer(options,BUCKET_NAME,"test/1128.txt","1543389114
666279991",1,buff,a,etag,&resp_headers);
        printf("====== fist part complete : etag - %s\n", etag->data);

        //第二片
        oos_str_set(&options->endpoint, HOST_NAME); //?
        resp_headers = oos_table_make(options->pool, 1);
        fseek(file, part_size, SEEK_SET);
        buff = (char*)oos_pcalloc(options->pool, file_size);
        int n = fread(buff, 1, (file_size-part_size), file);
        oos_str_set(&options->endpoint, HOST_NAME);
        status =
oos_upload_part_from_buffer(options,BUCKET_NAME,"test/1128.txt","1543389114
666279991",2,buff,n,etag,&resp_headers);
        printf("====== etag%s\n", etag->data);
    }
```

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：

```
void complete_multipart_upload_test(request_options_t *options,oos_status_t
*status){
    part_etags* tag1;
    part_etags tags[2];
    tag1 = oos_pcalloc(options->pool,sizeof(*tag1));
    oos_table_t* resp_headers = oos_table_make(options->pool, 1);
    tag1->partnumber =1;
    oos_str_set(&tag1->etag,"34de6d91703dc1a95015ccd19b9443f8");
    memcpy((char *)&tags[0],(char *)tag1,sizeof(*tag1));

    part_etags* tag2;
    tag2 = oos_pcalloc(options->pool,sizeof(*tag2));
    tag2->partnumber =2;
    oos_str_set(&tag2->etag,"975e38a32ccd25501bac3742c6c57c72");
    memcpy((char *)&tags[1],(char *)tag2,sizeof(*tag2));

    status =
oos_complete_multipart_upload(options,BUCKET_NAME,"test/1128.txt","15433891
14666279991",tags,2,&resp_headers);
    printf("service code %d\n", status->code);

}
```

4.3.5. 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：

```
void abort_multipart_upload_test(request_options_t *options,oos_status_t
*status){
    oos_table_t* resp_headers = oos_table_make(options->pool, 1);
    status =
oos_abort_multipart_upload(options,BUCKET_NAME,"test/multipartfile123.txt",
"1543308368586624129",&resp_headers);
    printf("service code %d\n", status->code);
}
```

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。

代码片：

```
    void create_access_key_test(request_options_t *options,oos_status_t
*status){
        oos_list_access_key_member_t* key;
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status = oos_ctyun_create_access_key(options,&key,&resp_headers);
        printf("service code %d\n", status->code);
    }

    void delete_ctyun_access_key_test(request_options_t
*options,oos_status_t *status){
        oos_table_t* resp_headers = oos_table_make(options->pool, 1);
        status =
oos_ctyun_delete_access_key(options,"c2fa33fbc44a92d1457c",&resp_headers);
        printf("service code %d\n", status->code);
    }
```