**小文件打包用户使用手册**

|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **修订内容描述** | **修订日期** | **修订后版本号** |
| 1 | 初稿 | 2016-07-25 | **1.0** |
| 2. | a.设置缓存文件个数等接口废弃  b.新增CArchiveManager::init接口  c.修改c++demo代码 | **2017-05-05** | **1.1** |
| 3 | 新增C/C++版本的接口支持一次性读写文件功能 | **2017-07-28** | **1.2** |
| 4 | 新增章节说明archive.conf | **2017-09-20** | **1.3** |
| 5. | 全新改版 | **2018-07-24** | **2.0** |
| 6. | 新增特别提醒 | **2018-08-10** | **2.1** |
| 7. | 增加C++的V2版本接口说明 | **2019-09-17** | **2.2** |

# 概述

本文档是小文件打包（ArchiveTool）的开发指南。ArchiveTool在EFS-SDK的基础上，实现小文件创建、读写等功能。ArchiveTool接口包括C、C++、Java语言版本，简化用户的编程。因为ArchiveTool是依赖于EFS-SDK的，所以使用ArchiveTool时必须同时使用EFS-SDK。EFS-SDK的使用方式可阅读《EFS-SDK\_UserDocs.docx》。

本文档涵盖C、C++、Java使用说明，篇幅较长，建议打开导航栏，选择具体语言版本后开始阅读。导航栏打开方法：点击”视图”，勾选”导航窗格”。

其余公共章节也请详细阅读，特别是常见问题。

# 概念介绍

## Bucket

Bucket是一个存储空间，想在EFS上存储数据时，必须先创建Bucket，也就是所有文件都必须隶属于某一个Bucket。您可以设置或修改Bucket的属性，包括访问权限、配额容量、生命周期等。

* Bucket内的文件是扁平化的，不存在目录概念
* 每个用户可以拥有多个Bucket
* Bucket的名字在EFS内必须是唯一的，即使不同用户也不能存在同名Bucket
* 单个Bucket中存储的文件没有限制，但最好不在一个Bucket下存储太多文件(10万+)

Bucket的命名规范如下：

* 长度必须在1-63字节之间
* 不能包含**\ \* ? " < > | / ' :这10个字符**
* 必须使用UTF-8编码
* 不区分大小写

## N+M

EFS使用Erase Code（纠删码）保护数据，所以在创建文件时必须指定N+M，其中N即数据块，M为根据EC计算所得的校验块，当小于等于M个数据块损坏时，不影响数据读取。小文件打包仅支持3+1和7+2两种模式。

## 小文件打包

小文件打包指将大小小于100MB的文件，通过追加方式写入一个大文件中，写入后返回一个文件名给用户，用户可根据该文件名调用小文件打包的读取接口进行读取。

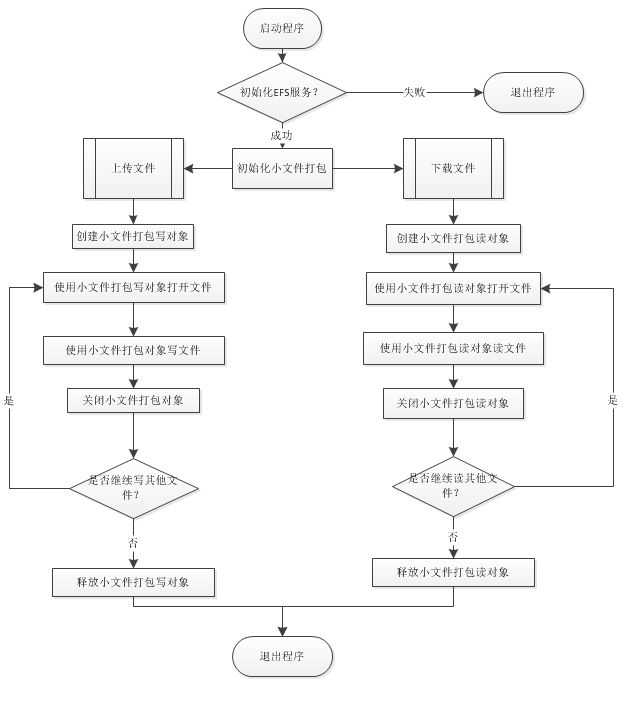
# 特别提醒

Java SDK和C SDK中的demo是基于7.2版本的EFS-SDK和Archive-SDK编写的，在其中同时使用了EFS-SDK和Archive-SDK。但是对于7.2之前的版本，EFS-SDK和Archive-SDK共同使用可能会存在[问题](#_与C版本的EFS_SDK一起使用时发生崩溃或莫名其妙的问题)（涉及Java和C版本）。

# 快速入门

因为ArchiveTool是依赖于EFS-SDK，所以部分流程需要使用EFS-SDK来完成。

## 基本流程



# C++ SDK

## 前言

### 接口版本说明

目前小文件打包提供的C++接口有两个版本，V1版本使用头文件为ArchiveFile.h的接口，V2版本为使用ArchiveFileV2.h头文件的接口，其中V2接口在V1.080.0000003.0.R之后的版本才有。

注意：

1. 使用时不要同时使用两个版本的接口。
2. 在需要windows下运行的程序是必须使用V2版本接口。
3. 一般情况建议使用V1版本的接口。

### 获取

联系项目对接人员获取对应版本的安装包，版本名规则如下：

DH\_EFS\_ARCHIVETOOL\_[Platform]\_[bits]\_PS\_[Version].tar.gz

实例如下：

* Linux
  + 32位：DH\_EFS\_ARCHIVETOOL\_32bit\_PS\_V1.072.0000004.0.R.180723.tar.gz
  + 64位：DH\_EFS\_ARCHIVETOOL\_64bit\_PS\_V1.072.0000004.0.R.180723.tar.gz
* Windows
  + 32位：

DH\_EFS\_ARCHIVETOOL\_Windows\_32bit\_PS\_V1.072.0000004.0.R.180723.tar.gz

* + 64位：

DH\_EFS\_ARCHIVETOOL\_Windows\_64bit\_PS\_V1.072.0000004.0.R.180723.tar.gz

上述实例中的[Version]的072，代表为7.2版本，可以此类推。

安装包内包含库以及头文件，头文件在archivetool\_include.tar中的ArchiveTool目录。

### 编译

需要包含的头文件：ArchiveFile.h或ArchiveFileV2.h

指定链接的库：

* Linux平台
  + 32位
    - libEFSArchiveFile.so
    - libEFSClient.so
  + 64位：
    - libEFSArchiveFile64.so
    - libEFSClient64.so
* windows平台
  + 32位
    - EFSArchive.lib（EFSArchive.dll的导入库文件）
    - EFSClient.lib（EFSClient.dll的导入库文件）
  + 64位
    - EFSArchive64.lib（EFSArchive64.dll的导入库文件）
    - EFSClient64.lib（EFSClient64.dll的导入库文件）

具体编译过程可参考[Demo](#_Demo)

### 安装

动态库必须安装在动态库加载目录中，否则可能在运行时出现无法找到库的错误。

动态库加载目录如下：

* Linux平台
  + 系统默认目录，比如32位的/lib, /usr/lib；64位的/lib64,/usr/lib64等目录
  + LD\_LIBRARY\_PATH环境变量指定的目录（推荐）
  + ldconfig配置文件中指定的目录
  + 在编译时通过`-Wl,-rpath`指定的目录
* windows平台
  + 应用程序所在目录（推荐）
  + 启动程序的目录
  + Windows SYSTEM目录
  + Windows目录
  + PATH环境变量指定的路径

必须安装的动态库：

* Linux
  + 32位
    - libEFSArchiveFile.so
    - libEFSClient.so（取自EFS-SDK包）
    - libEFSClientCore.so（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
  + 64位
    - libEFSArchiveFile64.so
    - libEFSClient64.so（取自EFS-SDK包）
    - libEFSClientCore64.so（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
* Windows
  + 32位
    - EFSArchive.dll
    - EFSClient.dll（取自EFS-SDK包）
    - EFSClientCore.dll（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
    - EFSHelper.dll（取自EFS-SDK包）
  + 64位
    - EFSArchive64.dll
    - EFSClient64.dll（取自EFS-SDK包）
    - EFSClientCore64.dll（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
    - EFSHelper64.dll（取自EFS-SDK包）

### 升级

获取新版本的安装包，取出其中的库直接替换即可。

因为ArchiveTool可使用任意版本的SDK，所以可以在不升级EFS SDK和云存储的情况，使用最新版本的ArchiveTool。

EFS SDK的升级请阅读《EFS-SDK\_UserDocs.docx》。

### 兼容性

1. 目前对外接口不存在兼容问题。
2. 内部读写功能均向前兼容。
3. V1.072.0000004.5.R版本的小文件打包必须配套V1.072.0000004.5.R之后的SDK（新增flush功能，保证写入图片实时可读）

## V1版接口使用介绍

以下介绍时，不涉及EFS SDK。请参考Demo，了解结合EFS SDK的使用。

### 全局管理

CArchiveManager为一个单例，用于创建和销毁小文件打包对象。初始化和销毁一般在进程启动和退出时进行。

#### 初始化

* **接口原型**

bool CArchiveManager::init( std::string cfgPath );

* **功能描述**

根据指定的配置文件初始化小文件打包功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| cfgPath | [IN] 配置文件路径 |

* **接口返回值**
  + true：初始化成功。
  + false：初始化失败。
* **示例**

|  |
| --- |
| // 指定配置文件路径进行初始化, 6.0版本后（含6.0）才存在该接口  if( Dahua::EFS::CArchiveManager::instance().init( "./archive.conf" ) ){  std::cout << "ArchiveManager inited" << std::endl;    // 销毁小文件打包管理对象，在此之前必须释放所有CArchiveFileWriter对象  Dahua::EFS::CArchiveManager::instance().release();  }else{  std::cout << "ArchiveManager init failed" << std::endl;  } |

* **提示**
  + 6.0版本之后（含6.0）才有该接口。为兼容以前的版本，如果外部没有主动调用该接口，则会在CArchiveManager::createFileWriter()中，以”./archive.conf”作为入参调用该接口，此时要求archive.conf必须在启动程序的目录下。
  + 如果不存在archive.conf，则会使用默认配置信息。需要注意6.0版本的默认配置信息在部署在非Datanode设备上时会存在问题。7.0版本后已经修复。
  + 可通过archive.conf指定是否使能本地存储，日志路径和级别，详细请查看[配置文件说明](#_配置文件说明)章节。

#### 释放

* **接口原型**

void CArchiveManager:: release();

* **功能描述**

释放小文件打包使用的所有资源。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**

无返回值

* **示例**

|  |
| --- |
| // 指定配置文件路径进行初始化, 6.0版本后（含6.0）才存在该接口  if( Dahua::EFS::CArchiveManager::instance().init( "./archive.conf" ) ){  std::cout << "ArchiveManager inited" << std::endl;    // 销毁小文件打包管理对象，在此之前必须释放所有CArchiveFileWriter对象  Dahua::EFS::CArchiveManager::instance().release();  }else{  std::cout << "ArchiveManager init failed" << std::endl;  } |

* **提示**
  + 构造CArchiveManager后，析构前必须调用该接口释放资源。
  + 因为小文件打包会使用EFS SDK创建文件，所以该接口必须在CEFileSystem::close()前调用，避免CEFileSystem::close()断言失败导致的崩溃。

### 上传文件

上传文件需要使用CArchiveFileWriter对象，该对象只能通过CArchiveManager对象来创建。对于一个CArchiveFileWriter对象不可多线程同时使用。

#### 创建

* **接口定义**

CArchiveFileWriter\* CArchiveManager::createFileWriter(

CEFileSystem\* ptr,

const std::string& regionname = "");

* **功能描述**

创建小文件打包对象。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| ptr | [IN ] 已经初始化好的CEFileSystem的指针。 |
| regionname | [IN ] 小文件域。 |

* **接口返回值**
  + 非NULL：成功。
  + NULL：失败。
* **示例**

|  |
| --- |
| // efs必须是已经初始化好的CEFileSystem对象  Dahua::EFS::CArchiveFileWriter\* writer = NULL;  writer = Dahua::EFS::CArchiveManager::instance().createFileWriter( efs );  if( NULL != wirter ){  std::cout << "create file writer\n";  }else{  std::cout << "create file writer failed\n";  } |

* **提示**
  + 传入的CEFileSystem的指针必须指向已经初始化好的对象。CEFileSystem的初始化可参考《EFS-SDK\_UserDocs.docx》
  + 不建议使用regionname。如果设置了，那么在返回的小文件名中会携带该regioname。
  + 必须检查返回值，避免返回NULL时，继续使用导致的崩溃
  + 一旦返回非NULL，但不使用时，必须调用CArchiveManager:: releaseFileWriter()进行释放。
  + 尽可能重复使用一个CArchiveFileWriter对象，避免反复初始化和销毁CArchiveFileWriter对象。7.0版本前，如果频繁初始化和销毁可能导致性能急剧下降。即使存在写入失败，CArchiveFileWriter仍可用于下一次写入

#### 初始化

* **接口定义**

bool CArchiveFileWriter::init( const uint16\_t n,

const uint16\_t m,

const std::string& bucketname);

* **功能描述**

初始化小文件打包对象，指定N+M以及文件所属的Bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| n | [IN ] N+M的N |
| m | [IN ] N+M的M |
| bucketname | [IN ] 已经存在的Bucket名，用于存储打包后的文件 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| Dahua::EFS::CArchiveFileWriter\* writer = NULL;  writer = Dahua::EFS::CArchiveManager::instance().createFileWriter( efs );  if( NULL != writer ){  // 指定N+M以及bucket，bucket必须是已经存在的。  if( writer->init( 3, 1, bucket\_name ) ){  // 使用该writer进行写入  }else{  // 初始化失败，不可使用该writer  }  } |

* **提示**
  + 一旦在该接口中指定N+M后，该CArchiveFileWriter的N+M不可再改变，当前只支持3+1和7+2模式，其余模式会返回失败
  + 在该接口中指定的bucket，可以通过CArchiveFileWriter::getBucket() 获取其名字
  + bucket可通过CArchiveFileWriter::setBucket()重新设置，仍需保证设置的bucket是已经存在的。不推荐对一个CArchiveFileWriter反复设置不同的bucket。

#### 打开文件

* **接口定义**

bool CArchiveFileWriter::open(const std::string& suffix = "");

* **功能描述**

打开文件，suffix指定该文件的后缀。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| suffix | [IN ] 指定返回的文件名的后缀 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| if( writer->open( "jpg") ){  // 打开成功，开始写入  } |

* **提示**
  + 该接口指定的参数主要用于拼接close()时返回的文件名，以便其他程序能正确处理文件，比如以jpg作为后缀，可以使得其他程序能以jpg格式解析文件。

#### 写文件

* **接口定义**

int CArchiveFileWriter::write(const char\* buffer, const uint32\_t length);

* **功能描述**

写入指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buffer | [IN ] 数据缓存地址，不能为NULL |
| length | [IN ] 待写入的数据长度，不能为0 |

* **接口返回值**
  + >0：实际写入的长度
  + <0：失败。需关闭，不可再写。
* **示例**

|  |
| --- |
| uint32\_t want\_write\_len = 1 << 20; // 文件大小为1M  uint32\_t write\_once\_len = 12288; // 推荐每次写入N\*4k倍数的数据，此处N为3  char\* buf = new char[write\_once\_len]; // 待写入数据的缓存  uint32\_t write\_once\_offset = write\_once\_len; // 初始设为write\_once\_len，保证进入读取原始数据的分支  uint32\_t write\_len = 0; // 已写入的总长度  while( write\_len < want\_write\_len ){  if( write\_once\_offset == write\_once\_len ){  write\_once\_offset = 0;  uint32\_t left\_write\_len = want\_write\_len - write\_len;  // 每次写的长度不能超过剩余需要写的长度  write\_once\_len = write\_once\_len > left\_write\_len ? left\_write\_len : write\_once\_len;  // 读取原始数据，此处以memset模拟  memset( buf, 0, write\_once\_len );  }  int ret = writer->write( buf + write\_once\_offset, write\_once\_len - write\_once\_offset );  if( 0 > ret ){  std::cout << "write failed\n";  break;  }    write\_once\_offset += ret;  write\_len += ret;  } |

* **提示**
  + 7.2版本之前，该接口为异步写入，缓存未满的情况下，总是返回成功，但其实际写入结果未知。
  + 全部写入的数据长度不能超过100MB，超过时会返回失败。

#### 关闭文件

* **接口定义**

bool CArchiveFileWriter::close(std::string& filename);

* **功能描述**

关闭文件，并返回文件名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [OUT ] 返回的文件名。 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string filename;  if( writer->close( filename ) ){  // 获取到文件名，可根据该文件名进行读取  } |

* **提示**
  + close()并不保证文件已经写入云存储，因此获取到文件名后，可能暂时无法读取，且可能因为异步写入失败导致永远无法读取。7.2版本之后有效解决该问题。
  + 一旦open()成功，则必须调用close()，否则会导致下一次open()失败
  + 如果write()失败，仍需调用close()，此时close()返回的文件名只能读取到已经写入的数据。

#### 释放

* **接口定义**

**bool** CArchiveManager::**releaseFileWriter(CArchiveFileWriter \*ptr);**

* **功能描述**

释放CArchiveFileWriter占用的资源

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| ptr | [IN ] 小文件打包对象的指针 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs必须是已经初始化好的CEFileSystem对象  Dahua::EFS::CArchiveFileWriter\* writer = NULL;  writer = Dahua::EFS::CArchiveManager::instance().createFileWriter( efs );  if( NULL != wirter ){  // 创建成功，一旦使用结束后，需要释放  Dahua::EFS::CArchiveManager::instance().releaseFileWriter( writer );  } |

* **提示**
  + 一旦CArchiveManager::createFileWriter()返回非NULL，则必须调用该接口，否则会导致内存泄露
  + 该接口必须在CEFileSystem::close()前调用。

#### 设置打包使用的Bucket

* **接口定义**

**bool** CArchiveFileWriter::**setBucket(const std::string& bucketname);**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **bucketname** | [IN ] bucket名。必须是已经存在的bucket |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string bucket\_name = “20180724”;  if( writer->setBucket( bucket\_name ) ){  // 设置成功后，打包后的文件将会存储在新bucket中  }else{  // 设置失败，则打包后的文件仍存储在原先的bucket中  } |

* **提示**
  + 不推荐对一个CArchiveFileWriter对象反复设置不同的Bucket。在7.2版本前，这会影响打包效果；在7.2版本后，没有影响。

#### 获取打包使用的Bucket

* **接口定义**

**bool** CArchiveFileWriter:: **getBucket( std::string& bucketname );**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **bucketname** | [OUT ] 获取的bucket名。 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string bucket\_name;  if( writer->getBucket( bucket\_name ) ){    } |

#### 获取域名

* **接口定义**

bool CArchiveFileWriter::getRegionName( std::string& regionname )**;**

* **功能描述**

获取创建小文件打包对象时指定的域名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| regionname | [OUT ] 获取的域名。 |

* **接口返回值**
  + true：成功。
  + false：失败。

#### 设置最大能缓存的文件个数（7.0后已废弃）

* **接口定义**

**bool** CArchiveFileWriter:: setMaxCachedFilesNumber( const uint32\_t number );

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 获取最大能缓存的文件个数（7.0后已废弃）

* **接口定义**

uint32\_t CArchiveFileWriter::getMaxCachedFilesNumber ();

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 获取当前缓存的文件个数（7.0后已废弃）

* **接口定义**

uint32CArchiveFileWriter:: getCurrentCachedFilesNumber ();

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 一次性写入

* **接口定义**

**bool** CArchiveFileWriter:: **writeOnce( const std::string& suffix,**

**const char\* buffer,**

**const uint32\_t length,**

**std::string& filename );**

* **功能描述**

合并了open()、write()、close()功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **suffix** | [IN ] 文件名后缀。 |
| **buffer** | [IN ] 数据缓存地址 |
| **length** | [IN ] 待写入数据的长度 |
| **filename** | [OUT] 写入成功后返回的文件名 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string suffix = "jpg";  uint32\_t length = 12288;  char\* buf = new char[length];  std::string filename;  if( writer->writeOnce( suffix, buf, length, filename ) ){  // 写入成功，可通过返回的filename读取文件  } |

* **提示**
  + 7.0版本后支持该接口

### 下载文件

下载文件需要使用CArchiveFileReader，该对象可直接构造。构造CArchiveFileReader时，必须指定一个已经初始化好的CEFileSystem对象指针。一个CArchiveFileReader对象可以读取任意次数的文件，每次读取文件时无需重新构造。

使用同一个CArchiveFileReader对象读取相同前缀的文件，可获取最佳性能。写入时返回的文件名格式为bucket/archivefile-xxxx-xx-xx-xxxxxx-xxxxxxxxxxxxxxxx:offset/size.suffix。标红字段相同就是前缀相同，意味着小文件打包在同一个大文件中。

#### 打开文件

* **接口定义**

bool CArchiveFileReader::open( const std::string& filename, uint64\_t& length );

* **功能描述**

打开指定的文件，并获取文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [IN ] 指定的文件名 |
| length | [OUT ] 文件长度 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs是一个指向已经初始化好的CEFileSystem对象的指针  Dahua::EFS::CArchiveFileReader reader( efs );  if( reader->open( filename, file\_length ) ){  // 打开成功，可以读取文件  } |

#### 读文件

* **接口定义**

int CArchiveFileReader:: read( char\* buffer, const uint32\_t length );

* **功能描述**

读取指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buffer | [IN ] 保存读取数据的缓存 |
| length | [IN ] 待读取的长度 |

* **接口返回值**
  + >0：实际读取的成都。
  + =0：暂时不可读取
  + -1：失败。
  + -2：读取到文件尾
* **示例**

|  |
| --- |
| uint32\_t want\_read\_len = file\_length;  uint32\_t read\_once\_len = 12288; // 推荐每次读取的长度是4K\*N的倍数，此处N是3  char\* buf = new char[read\_once\_len]; // 读取数据的缓存  uint32\_t read\_len = 0;  while( read\_len < want\_read\_len ){  int ret = reader->read( buf, read\_once\_len );  if( 0 < ret ){  read\_len += ret;  uint32\_t left\_read\_len = want\_read\_len - read\_len;  // 每次读取的长度不能超过剩余需要读取的长度  read\_once\_len = read\_once\_len > left\_read\_len ? left\_read\_len : read\_once\_len;  // 对读取到的数据进行操作，比如写入本地文件  // 读取多少就写入多少  // write( fd, buf, ret );  }else if( 0 == ret ){  // 休眠1ms，必须有休眠，否则可能导致CPU占用上升  SLEEP\_MS( 1 );  }else if( -1 == ret ){  std::cout << "read failed, error:" << Dahua::EFS::getLastError() << std::endl;  break;  }else if( -2 == ret ){  // 读取到文件尾  std::cout << "read end" << std::endl;  break;  }  } |

* **提示**
  + 如果必须保证文件数据是完整的，比如图片这类文件，那么请判断read\_len是否等于want\_read\_len，如果相等则认为读取成功，否则认为读取失败。

#### 关闭文件

* **接口定义**

bool CArchiveFileReader::close();;

* **功能描述**

关闭

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs是一个指向已经初始化好的CEFileSystem对象的指针  Dahua::EFS::CArchiveFileReader reader( efs );  if( reader->open( filename, file\_length ) ){  // 打开成功，读取结束后，必须关闭  reader->close();  } |

* **提示**
  + 一旦open()成功，那么不论是否读取失败，都必须调用close()接口，否则会导致下一次open()失败。

#### 释放

* **接口定义**

bool CArchiveFileReader::release();

* **功能描述**

释放对象本身的资源

* **接口参数**

**无入参**

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs是一个指向已经初始化好的CEFileSystem对象的指针  Dahua::EFS::CArchiveFileReader reader( efs );  // 一般无需关心返回值  (void)reader->release(); |

* **提示**
  + 该接口仅在确定不再使用该CArchiveFIleReader对象后调用，尽可能重复使用同一个CArchiveFileReader，避免频繁release()。
  + 该接口必须在CEFileSystem::close()前调用

#### 解析文件名

* **接口定义**

**bool getInfos4ArchName( const std::string& archname,**

**std::string& regionname,**

**std::string& filename,**

**uint64\_t& size );**

* **功能描述**

解析CArchiveFileWriter::close()返回的文件名，从中解析出域名，不含域名的文件名以及文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| archname | [IN ] 指定的文件名 |
| regionname | [OUT ] 域名 |
| filename | [OUT ] 不含域名的文件名 |
| size | [OUT ] 文件大小 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string region\_name;  std::string filename\_witout\_region;  uint64\_t file\_size = 0;  if( !Dahua::EFS::getInfos4ArchName( filename, region\_name, filename\_witout\_region, file\_size ) ){  std::cout << "parse filename failed" << std::endl;  return false;  } |

* **提示**
  + 如果archname本身就不含域名，则filename就是archname

#### 一次性读取

* **接口定义**

bool CArchiveFileReader:: readOnce( const std::string& filename,

char\* buffer,

const uint32\_t length,

int32\_t timeout );

* **功能描述**

合并了open()、read()、close()功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [IN ] 指定的文件名 |
| buffer | [IN ] 数据缓存 |
| length | [IN ] 待读取的长度。 |
| timeout | [IN ] 超时时间，单位毫秒。-1表示永不超时 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string region\_name;  std::string filename\_witout\_region;  uint64\_t file\_size = 0;  if( !Dahua::EFS::getInfos4ArchName( filename, region\_name, filename\_witout\_region, file\_size ) ){  std::cout << "parse filename failed" << std::endl;  return false;  }  char\* buf = new char[file\_size];  if( !reader->readOnce( filename, buf, file\_size, -1 ) ){  std::cout << "readOnce failed" << std::endl;  return false;  }  return true; |

* **提示**
  + 7.0版本后支持该接口
  + 需要借助getInfos4ArchName()获取文件长度，然后分配足够的缓存空间，才能调用readOnce()接口。

## V2版接口使用介绍

以下介绍时，不涉及EFS SDK。请参考Demo，了解结合EFS SDK的使用。

V2版本与V1版本的主要区别在于所有入参和出参均使用C类型数据，如std::string会用const char\*替换等，且删除了V1接口中已经废弃了的部分接口和参数。

V2接口在V1.080.0000003.0.R之后的版本才有。

注意：

1. 使用时不要同时使用两个版本的接口。
2. 在需要windows下运行的程序是必须使用V2版本接口。
3. 一般情况建议使用V1版本的接口。

### 全局管理

CArchiveManagerV2为一个单例，用于创建和销毁小文件打包对象。初始化和销毁一般在进程启动和退出时进行。

#### 初始化

* **接口原型**

bool CArchiveManagerV2::init( const char\* cfgPath );

* **功能描述**

根据指定的配置文件初始化小文件打包功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| cfgPath | [IN] 配置文件路径 |

* **接口返回值**
  + true：初始化成功。
  + false：初始化失败。
* **示例**

|  |
| --- |
| // 指定配置文件路径进行初始化  if( Dahua::EFS::CArchiveManagerV2::instance().init( "./archive.conf" ) ){  std::cout << "ArchiveManagerV2 inited" << std::endl;    // 销毁小文件打包管理对象，在此之前必须释放所有CArchiveFileWriterV2对象  Dahua::EFS::CArchiveManagerV2::instance().release();  }else{  std::cout << "ArchiveManagerV2 init failed" << std::endl;  } |

* **提示**
  + 如果外部没有主动调用该接口，则会在CArchiveManagerV2::createFileWriter()中，以”./archive.conf”作为入参调用该接口，此时要求archive.conf必须在启动程序的目录下。
  + 如果不存在archive.conf，则会使用默认配置信息。需要注意6.0版本的默认配置信息在部署在非Datanode设备上时会存在问题。7.0版本后已经修复。
  + 可通过archive.conf指定是否使能本地存储，日志路径和级别，详细请查看[配置文件说明](#_配置文件说明)章节。

#### 释放

* **接口原型**

void CArchiveManagerV2::release();

* **功能描述**

释放小文件打包使用的所有资源。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**

无返回值

* **示例**

|  |
| --- |
| // 指定配置文件路径进行初始化  if( Dahua::EFS::CArchiveManagerV2::instance().init( "./archive.conf" ) ){  std::cout << "ArchiveManagerV2 inited" << std::endl;    // 销毁小文件打包管理对象，在此之前必须释放所有CArchiveFileWriterV2对象  Dahua::EFS::CArchiveManagerV2::instance().release();  }else{  std::cout << "ArchiveManagerV2 init failed" << std::endl;  } |

* **提示**
  + 构造CArchiveManagerV2后，析构前必须调用该接口释放资源。
  + 因为小文件打包会使用EFS SDK创建文件，所以该接口必须在CEFileSystem::close()前调用，避免CEFileSystem::close()断言失败导致的崩溃。

### 上传文件

上传文件需要使用CArchiveFileWriterV2对象，该对象只能通过CArchiveManagerV2对象来创建。对于一个CArchiveFileWriterV2对象不可多线程同时使用。

#### 创建

* **接口定义**

CArchiveFileWriterV2\* CArchiveManagerV2::createFileWriter( CEFileSystem\* ptr );

* **功能描述**

创建小文件打包对象。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| ptr | [IN ] 已经初始化好的CEFileSystem的指针。 |

* **接口返回值**
  + 非NULL：成功。
  + NULL：失败。。
* **示例**

|  |
| --- |
| // efs必须是已经初始化好的CEFileSystem对象  Dahua::EFS::CArchiveFileWriterV2\* writer = NULL;  writer = Dahua::EFS::CArchiveManagerV2::instance().createFileWriter( efs );  if( NULL != wirter ){  std::cout << "create file writer\n";  }else{  std::cout << "create file writer failed\n";  } |

* **提示**
  + 传入的CEFileSystem的指针必须指向已经初始化好的对象。CEFileSystem的初始化可参考《EFS-SDK\_UserDocs.docx》。
  + 必须检查返回值，避免返回NULL时，继续使用导致的崩溃
  + 一旦返回非NULL但不使用时，必须调用CArchiveManagerV2:: releaseFileWriter()进行释放。
  + 尽可能重复使用一个CArchiveFileWriterV2对象，避免反复初始化和销毁CArchiveFileWriterV2对象。即使存在写入失败，CArchiveFileWriterV2仍可用于下一次写入。

#### 初始化

* **接口定义**

bool CArchiveFileWriterV2::init( const uint16\_t n,

const uint16\_t m,

const char\* bucketname);

* **功能描述**

初始化小文件打包对象，指定N+M以及文件所属的Bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| n | [IN ] N+M的N |
| m | [IN ] N+M的M |
| bucketname | [IN ] 已经存在的Bucket名，用于存储打包后的文件 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| Dahua::EFS::CArchiveFileWriterV2\* writer = NULL;  writer = Dahua::EFS::CArchiveManagerV2::instance().createFileWriter( efs );  if( NULL != writer ){  // 指定N+M以及bucket，bucket必须是已经存在的。  if( writer->init( 3, 1, bucket\_name .c\_str()) ){  // 使用该writer进行写入  }else{  // 初始化失败，不可使用该writer  }  } |

* **提示**
  + 一旦在该接口中指定N+M后，该CArchiveFileWriter的N+M不可再改变，当前只支持3+1和7+2模式，其余模式会返回失败
  + 在该接口中指定的bucket，可以通过CArchiveFileWriterV2::getBucket() 获取其名字
  + bucket可通过CArchiveFileWriterV2::setBucket()重新设置，仍需保证设置的bucket是已经存在的。不推荐对一个CArchiveFileWriterV2反复设置不同的bucket。

#### 打开文件

* **接口定义**

bool CArchiveFileWriterV2::open(const char\* suffix = NULL);

* **功能描述**

打开文件，suffix指定该文件的后缀。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| suffix | [IN ] 指定返回的文件名的后缀 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| if( writer->open( "jpg") ){  // 打开成功，开始写入  } |

* **提示**
  + 该接口指定的参数主要用于拼接close()时返回的文件名，以便其他程序能正确处理文件，比如以jpg作为后缀，可以使得其他程序能以jpg格式解析文件。

#### 写文件

* **接口定义**

int CArchiveFileWriterV2::write(const char\* buffer, const uint32\_t length);

* **功能描述**

写入指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buffer | [IN ] 数据缓存地址，不能为NULL |
| length | [IN ] 待写入的数据长度，不能为0 |

* **接口返回值**
  + >0：实际写入的长度
  + <0：失败。需关闭，不可再写。
* **示例**

|  |
| --- |
| uint32\_t want\_write\_len = 1 << 20; // 文件大小为1M  uint32\_t write\_once\_len = 12288; // 推荐每次写入N\*4k倍数的数据，此处N为3  char\* buf = new char[write\_once\_len]; // 待写入数据的缓存  uint32\_t write\_once\_offset = write\_once\_len; // 初始设为write\_once\_len，保证进入读取原始数据的分支  uint32\_t write\_len = 0; // 已写入的总长度  while( write\_len < want\_write\_len ){  if( write\_once\_offset == write\_once\_len ){  write\_once\_offset = 0;  uint32\_t left\_write\_len = want\_write\_len - write\_len;  // 每次写的长度不能超过剩余需要写的长度  write\_once\_len = write\_once\_len > left\_write\_len ? left\_write\_len : write\_once\_len;  // 读取原始数据，此处以memset模拟  memset( buf, 0, write\_once\_len );  }  int ret = writer->write( buf + write\_once\_offset, write\_once\_len - write\_once\_offset );  if( 0 > ret ){  std::cout << "write failed\n";  break;  }    write\_once\_offset += ret;  write\_len += ret;  } |

* **提示**
  + 全部写入的数据长度不能超过100MB，超过时会返回失败。

#### 关闭文件

* **接口定义**

bool CArchiveFileWriterV2::close(char\* filename, uint32\_t len);

* **功能描述**

关闭文件，并返回文件名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [OUT] 返回的文件名。 |
| len | [IN] 传入filename的预分配空间长度 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| const uint32\_t file\_name\_max\_len = 1024;  char filename[file\_name\_max\_len] = { 0 };  if( writer->close( filename, file\_name\_max\_len ) ){  // 获取到文件名，可根据该文件名进行读取  }  else {  // 返回失败  } |

* **提示**
  + 一旦open()成功，则必须调用close()，否则会导致下一次open()失败
  + 如果write()失败，仍需调用close()，此时close()返回的文件名只能读取到已经写入的数据。
  + 如果file\_name\_max\_len长度不足，同样会返回失败，此时可以通过查看日志发现错误原因，建议大于等于1024。

#### 释放

* **接口定义**

**bool** CArchiveManagerV2::**releaseFileWriter(CArchiveFileWriterV2 \*ptr);**

* **功能描述**

释放CArchiveFileWriterV2占用的资源

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| ptr | [IN ] 小文件打包对象的指针 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs必须是已经初始化好的CEFileSystem对象  Dahua::EFS::CArchiveFileWriterV2\* writer = NULL;  writer = Dahua::EFS::CArchiveManagerV2::instance().createFileWriter( efs );  if( NULL != wirter ){  // 创建成功，一旦使用结束后，需要释放  Dahua::EFS::CArchiveManagerV2::instance().releaseFileWriter( writer );  } |

* **提示**
  + 一旦CArchiveManagerV2::createFileWriter()返回非NULL，则必须调用该接口，否则会导致内存泄露
  + 该接口必须在CEFileSystem::close()前调用。

#### 设置打包使用的Bucket

* **接口定义**

**bool** CArchiveFileWriterV2::**setBucket(const char\* bucketname);**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **bucketname** | [IN ] bucket名。必须是已经存在的bucket |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string bucket\_name = “20180724”;  if( writer->setBucket( bucket\_name.c\_str() ) ){  // 设置成功后，打包后的文件将会存储在新bucket中  }else{  // 设置失败，则打包后的文件仍存储在原先的bucket中  } |

* **提示**
  + 不推荐对一个CArchiveFileWriterV2对象反复设置不同的Bucket。

#### 获取打包使用的Bucket

* **接口定义**

**bool** CArchiveFileWriterV2:: **getBucket( char\* bucketname, uint32\_t len );**

* **功能描述**

获取打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **bucketname** | [OUT ] 获取的bucket名。 |
| **len** | [IN] 传入bucketname的预分配空间长度 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| const uint32\_t bucket\_name\_max\_len = 128;  char bucket\_name[bucket\_name\_max\_len] = { 0 };  if( writer->getBucket( bucket\_name, bucket\_name\_max\_len ) ){    } |

* **提示**
  + 如果bucket\_name\_max\_len长度不足，同样会返回失败，此时可以通过查看日志发现错误原因，建议大于等于64。

#### 一次性写入

* **接口定义**

**bool** CArchiveFileWriterV2::**writeOnce( const char\* suffix,**

**const char\* buffer,**

**const uint32\_t length,**

**char\* filename,**

**uint32\_t len );**

* **功能描述**

合并了open()、write()、close()功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **suffix** | [IN] 文件名后缀。 |
| **buffer** | [IN] 数据缓存地址 |
| **length** | [IN] 待写入数据的长度 |
| **filename** | [OUT] 写入成功后返回的文件名 |
| **len** | [IN] 传入filename的预分配空间长度 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| std::string suffix = "jpg";  uint32\_t length = 12288;  char\* buf = new char[length];  const uint32\_t file\_name\_max\_len = 1024;  char filename[file\_name\_max\_len] = { 0 };  if( writer->writeOnce( suffix.c\_str(), buf, length, filename, len ) ){  // 写入成功，可通过返回的filename读取文件  } |

* **提示**
  + 如果file\_name\_max\_len长度不足，同样会返回失败，此时可以通过查看日志发现错误原因，建议大于等于1024。

### 下载文件

下载文件需要使用CArchiveFileReaderV2，该对象可直接构造。构造CArchiveFileReaderV2时，必须指定一个已经初始化好的CEFileSystem对象指针。一个CArchiveFileReaderV2对象可以读取任意次数的文件，每次读取文件时无需重新构造。

使用同一个CArchiveFileReaderV2对象读取相同前缀的文件，可获取最佳性能。写入时返回的文件名格式为bucket/archivefile-xxxx-xx-xx-xxxxxx-xxxxxxxxxxxxxxxx:offset/size.suffix。标红字段相同就是前缀相同，意味着小文件打包在同一个大文件中。

#### 打开文件

* **接口定义**

bool CArchiveFileReaderV2::open( const char\* filename, uint64\_t& length );

* **功能描述**

打开指定的文件，并获取文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [IN] 指定的文件名 |
| length | [OUT] 文件长度 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs是一个指向已经初始化好的CEFileSystem对象的指针  Dahua::EFS::CArchiveFileReaderV2 reader( efs );  if( reader->open( filename, file\_length ) ){  // 打开成功，可以读取文件  } |

#### 读文件

* **接口定义**

int CArchiveFileReaderV2::read( char\* buffer, const uint32\_t length );

* **功能描述**

读取指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buffer | [IN] 保存读取数据的缓存 |
| length | [IN] 待读取的长度 |

* **接口返回值**
  + >0：实际读取的成都。
  + =0：暂时不可读取
  + -1：失败。
  + -2：读取到文件尾
* **示例**

|  |
| --- |
| uint32\_t want\_read\_len = file\_length;  uint32\_t read\_once\_len = 12288; // 推荐每次读取的长度是4K\*N的倍数，此处N是3  char\* buf = new char[read\_once\_len]; // 读取数据的缓存  uint32\_t read\_len = 0;  while( read\_len < want\_read\_len ){  int ret = reader->read( buf, read\_once\_len );  if( 0 < ret ){  read\_len += ret;  uint32\_t left\_read\_len = want\_read\_len - read\_len;  // 每次读取的长度不能超过剩余需要读取的长度  read\_once\_len = read\_once\_len > left\_read\_len ? left\_read\_len : read\_once\_len;  // 对读取到的数据进行操作，比如写入本地文件  // 读取多少就写入多少  // write( fd, buf, ret );  }else if( 0 == ret ){  // 休眠1ms，必须有休眠，否则可能导致CPU占用上升  SLEEP\_MS( 1 );  }else if( -1 == ret ){  std::cout << "read failed, error:" << Dahua::EFS::getLastError() << std::endl;  break;  }else if( -2 == ret ){  // 读取到文件尾  std::cout << "read end" << std::endl;  break;  }  } |

* **提示**
  + 如果必须保证文件数据是完整的，比如图片这类文件，那么请判断read\_len是否等于want\_read\_len，如果相等则认为读取成功，否则认为读取失败。

#### 关闭文件

* **接口定义**

bool CArchiveFileReaderV2::close();

* **功能描述**

关闭

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs是一个指向已经初始化好的CEFileSystem对象的指针  Dahua::EFS::CArchiveFileReaderV2 reader( efs );  if( reader->open( filename, file\_length ) ){  // 打开成功，读取结束后，必须关闭  reader->close();  } |

* **提示**
  + 一旦open()成功，那么不论是否读取失败，都必须调用close()接口，否则会导致下一次open()失败。

#### 释放

* **接口定义**

bool CArchiveFileReaderV2::release();

* **功能描述**

释放对象本身的资源

* **接口参数**

**无入参**

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // efs是一个指向已经初始化好的CEFileSystem对象的指针  Dahua::EFS::CArchiveFileReaderV2 reader( efs );  // 一般无需关心返回值  (void)reader->release(); |

* **提示**
  + 该接口仅在确定不再使用该CArchiveFIleReaderV2对象后调用，尽可能重复使用同一个CArchiveFileReaderV2，避免频繁release()。
  + 该接口必须在CEFileSystem::close()前调用

#### 解析文件名

* **接口定义**

**bool getInfos4ArchNameV2( const char\* archname,**

**char\* filename,**

**uint32\_t len,**

**uint64\_t& size );**

* **功能描述**

解析CArchiveFileWriterV2::close()返回的文件名，从中解析出域名，不含域名的文件名以及文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| archname | [IN] 指定的文件名 |
| filename | [OUT ] 文件名 |
| len | [IN] 传入文件名预留空间大小 |
| size | [OUT ] 文件大小 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| const uint32\_t file\_name\_max\_len = 1024;  char ret\_file\_name[file\_name\_max\_len] = { 0 };  uint64\_t file\_size = 0;  if( !Dahua::EFS::getInfos4ArchName( filename, ret\_file\_name, file\_name\_max\_len, file\_size ) ){  std::cout << "parse filename failed" << std::endl;  return false;  } |

#### 一次性读取

* **接口定义**

bool CArchiveFileReaderV2::readOnce( const char\* filename,

char\* buffer,

const uint32\_t length,

int32\_t timeout );

* **功能描述**

合并了open()、read()、close()功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [IN ] 指定的文件名 |
| buffer | [IN ] 数据缓存 |
| length | [IN ] 待读取的长度。 |
| timeout | [IN ] 超时时间，单位毫秒。-1表示永不超时 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| const uint32\_t file\_name\_max\_len = 1024;  char ret\_file\_name[file\_name\_max\_len] = { 0 };  uint64\_t file\_size = 0;  if( !Dahua::EFS::getInfos4ArchName( filename, ret\_file\_name, file\_name\_max\_len, file\_size ) ){  std::cout << "parse filename failed" << std::endl;  return false;  }  char\* buf = new char[file\_size];  if( !reader->readOnce( filename, buf, file\_size, -1 ) ){  std::cout << "readOnce failed" << std::endl;  return false;  }  return true; |

* **提示**
  + 需要借助getInfos4ArchNameV2()获取文件长度，然后分配足够的缓存空间，才能调用readOnce()接口。

## Demo

demo中的example.cpp里包含V1接口所有功能的实现，exampleV2.cpp里包含V2接口所有功能的实现，可选择接口版本阅读，如实现存在错误，请联系我们。为了减小文档大小，Demo中的库全部truncate为0字节，请在编译前手动替换成同名的库，并修改对应文件中的EFS服务器地址、用户名和密码。

### Linux



解压后，选择想要编译的源文件，执行make x86编译32位程序，make x86\_64编译64位程序，执行make test\_x86或make test\_x86\_64演示简单的上传和下载功能。

### Windows



示例包含vs2005的工程，其中include目录为头文件目录，libs目录为链接库目录，生成的程序demo-c++位于bin目录，bin目录下已放置了EFSArchive.dll、EFSClient.dll、EFSClientCore.dll和EFSHelper.dll，可直接在该目录运行demo-c++，演示简单的上传下载功能。

提供的Demo是Release 32位的，请参照该Demo，自行搭建64位工程。

# Java SDK

## 前言

### 获取

联系项目对接人员获取对应版本的安装包，版本名规则如下：

DH\_EFS\_ARCHIVETOOL\_[Platform]\_[bits]\_PS\_[Version].tar.gz

实例如下：

* Linux
  + 32位：DH\_EFS\_ARCHIVETOOL\_32bit\_PS\_V1.072.0000004.0.R.180723.tar.gz
  + 64位：DH\_EFS\_ARCHIVETOOL\_64bit\_PS\_V1.072.0000004.0.R.180723.tar.gz
* Windows
  + 32位：

DH\_EFS\_ARCHIVETOOL\_Windows\_32bit\_PS\_V1.072.0000004.0.R.180723.tar.gz

* + 64位：

DH\_EFS\_ARCHIVETOOL\_Windows\_64bit\_PS\_V1.072.0000004.0.R.180723.tar.gz

上述实例中的[Version]的072，代表为7.2版本，可以此类推。

### 编译

1. 从安装包中获得EFSArchiveToolClient.jar和jna-4.1.0.jar，从EFS SDK安装包中获取EFSClient.jar
2. 在Eclipse中选择您的工程，右键选择Properties->Java Build Path->Add JARS，选中第1步中的jar包

具体工程可参考[Demo](#_Demo_2)

PS: EFSArchiveToolClient.jar还依赖C/C++动态库，请参考安装章节将其放在正确的位置。jna-4.1.0.jar与EFS SDK中的jna-4.1.0.jar是一样的，只需包含一个即可。

### 安装

动态库必须安装在动态库加载目录中，否则可能在运行时出现无法找到库的错误。

动态库加载目录如下：

* Linux平台
  + 系统默认目录，比如32位的/lib, /usr/lib；64位的/lib64,/usr/lib64等目录
  + LD\_LIBRARY\_PATH环境变量指定的目录（推荐）
  + ldconfig配置文件中指定的目录
  + 在编译时通过`-Wl,-rpath`指定的目录
* windows平台
  + 应用程序所在目录（推荐）
  + 启动程序的目录
  + Windows SYSTEM目录
  + Windows目录
  + PATH环境变量指定的路径

必须安装的动态库：

* Linux
  + 32位
    - libEFSArchiveFileTool.so
    - libEFSAdapter.so（取自EFS-SDK包）
    - libEFSClient.so（取自EFS-SDK包）
    - libEFSClientCore.so（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
  + 64位
    - libEFSArchiveFileTool64.so
    - libEFSAdapter64.so（取自EFS-SDK包）
    - libEFSClient64.so（取自EFS-SDK包）
    - libEFSClientCore64.so（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
* Windows
  + 32位
    - EFSArchiveTool.dll
    - EFSAdapter.dll（取自EFS-SDK包）
    - EFSClient.dll（取自EFS-SDK包）
    - EFSClientCore.dll（使用5.0版本后的EFS SDK才有）
    - EFSHelper.dll（取自EFS-SDK包）
  + 64位
    - EFSArchiveTool64.dll
    - EFSAdapter64.dll（取自EFS-SDK包）
    - EFSClient64.dll（取自EFS-SDK包）
    - EFSClientCore64.dll（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
    - EFSHelper64.dll（取自EFS-SDK包）

因为java运行时会根据平台不同默认搜索linux-x86，linux-x86-64，win32-x86，win32-x86-64这几个目录，所以也可以将动态库放在这几个目录下。

### 升级

获取新版本的安装包，取出其中的动态库直接替换即可。

因为ArchiveTool可使用任意版本的SDK，所以可以在不升级EFS SDK和云存储的情况，使用最新版本的ArchiveTool。

EFS SDK的升级请阅读《EFS-SDK\_UserDocs.docx》。

### 兼容性

目前对外接口不存在兼容问题。

内部读写功能均向前兼容。

## 使用介绍

以下介绍时，不涉及EFS SDK。请参考Demo，了解结合EFS SDK的使用。

请注意：在小文件打包中重复实现了EFS SDK的部分功能，如创建bucket等，请忽略这些接口，以下文档没有介绍的接口请选择使用EFS SDK中对应的接口。小文件打包功能本身仅需使用以下介绍的接口（如果头文件中的接口下面没有介绍的，可以直接忽略，属于废弃接口）。

### 全局管理

#### 设置加载动态库路径

* **接口原型**

void Utils. setArchiveLibPath(String s);

* **功能描述**

指定加载动态库路径。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| s | [IN] 动态库放置路径 |

* **示例**

|  |
| --- |
| // 设置SDK动态库加载路径为D盘  com.dahuatech.efs.utils.Utils.setEFSLibPath("d:\\");  // 设置小文件打包的动态库路径为D盘  com.dahuatech.archive.utils.Utils.setArchiveLibPath("d:\\"); |

* **提示**
  + 使用该接口需要需要使用先设置SDK 的动态库加载路径;
  + 动态库加载路径的设置需要在初始化EFS之前执行
  + 不设置动态库加载路径时，默认的动态库加载路径是“安装”章节提到的那几个路径
  + 云存储8.0开始支持该接口

#### 初始化

* **接口原型**

boolean ArchiveTool.init( EFSConfig cfg );

* **功能描述**

初始化小文件打包功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| cfg | [IN] 服务配置信息 |

* **接口返回值**
  + true：初始化成功。
  + false：初始化失败。
* **示例**

|  |
| --- |
| ArchiveTool archive\_tool = new ArchiveTool();  EFSConfig.ByReference cfg = new EFSConfig.ByReference();  cfg.address = "192.168.1.1"; // 云存储服务IP  cfg.port = 38100;  cfg.userName = "root"; // 存储用户名  cfg.password = "123456"; // 存储用户密码  if( !archive\_tool.init( cfg ) ) {  return null;  }  return archive\_tool; |

* **提示**
  + 全局只需一个ArchiveTool对象，初始化一次即可

#### 释放

* **接口原型**

boolean ArchiveTool.close();

* **功能描述**

释放小文件打包使用的所有资源。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + 无需关注返回值
* **示例**

|  |
| --- |
| ArchiveTool archive\_tool = new ArchiveTool();  EFSConfig.ByReference cfg = new EFSConfig.ByReference();  cfg.address = "192.168.1.1";  cfg.port = 38100;  cfg.userName = "root";  cfg.password = "123456";  if( archive\_tool.init( cfg ) ) {  // 使用结束后，必须调用close()否则会有资源泄露  archive\_tool.close();  } |

* **提示**
  + 一旦ArchiveTool.init()成功，则必须调用ArchiveTool.close()，否则会有资源泄露的风险

### 上传文件

上传文件需要使用ArchiveWriter对象，该对象只能通过ArchiveTool对象来创建。对于一个ArchiveTool对象不可多线程同时使用。

#### 创建

* **接口定义**

ArchiveWriter ArchiveTool. createArchiveWriter();

* **功能描述**

创建小文件打包对象。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + 非null：成功。
  + null：失败。
* **示例**

|  |
| --- |
| ArchiveWriter writer = null;  writer = archiveTool.createArchiveWriter();  if( null == writer ){  System.out.println( "create archive writer failed" );  break;  } |

* **提示**
  + 一旦返回非null，当不使用时，必须调用ArchiveWrite.release()释放资源，避免资源泄露
  + 请尽可能重复使用一个ArchiveWriter对象，避免反复初始化和销毁CArchiveFileWriter对象。7.0版本前，如果频繁初始化和销毁可能导致性能急剧下降。即使存在写入失败，ArchiveWriter仍可用于下一次写入

#### 初始化

* **接口定义**

boolean ArchiveWriter.init(byte n, byte m, String bucketname);

* **功能描述**

初始化小文件打包对象，指定N+M以及文件所属的Bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| n | [IN ] N+M的N |
| m | [IN ] N+M的M |
| bucketname | [IN ] 已经存在的Bucket名，用于存储打包后的文件 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // 初始化，指定N+M和存储的bucket  if( !writer.init( (byte)3, (byte)1, bucket\_name ) ){  System.out.println( "archive writer init failed" );  break;  } |

* **提示**
  + 一旦在该接口中指定N+M后，该ArchiveWriter的N+M不可再改变，当前只支持3+1和7+2模式，其余模式会返回失败
  + 在该接口中指定的bucket，可以通过ArchiveWriter.getBucket() 获取其名字
  + bucket可通过ArchiveWriter.setBucket()重新设置，仍需保证设置的bucket是已经存在的。不推荐对一个CArchiveFileWriter反复设置不同的bucket。

#### 打开文件

* **接口定义**

boolean ArchiveWriter.open(String suffix);

* **功能描述**

打开文件，suffix指定该文件的后缀。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| suffix | [IN ] 指定返回的文件名的后缀 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // 返回的文件名后缀为jpg  if( !writer.open( "jpg" ) ){  System.out.println( "open failed" );  return null;  } |

* **提示**
  + 该接口指定的参数主要用于拼接close()时返回的文件名，以便其他程序能正确处理文件，比如以jpg作为后缀，可以使得其他程序能以jpg格式解析文件。

#### 写文件

* **接口定义**

**int ArchiveWriter.write(byte[] buffer, int offset, int len);**

* **功能描述**

写入指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buffer | [IN ] 数据缓存地址，不能为NULL |
| offset | [IN ] 写入buffer的offset位置的数据 |
| length | [IN ] 待写入的数据长度，不能为0 |

* **接口返回值**
  + >0：实际写入的长度
  + <0：失败。需关闭，不可再写。
* **示例**

|  |
| --- |
| int want\_write\_len = 1 << 20; // 文件大小为1M  int write\_once\_len = 12288; // 推荐每次写入N\*4k倍数的数据，此处N为3  byte[] buf = new byte[write\_once\_len]; // 待写入数据的缓存  int write\_once\_offset = write\_once\_len; // 初始设为write\_once\_len，保证进入读取原始数据的分支  int write\_len = 0; // 已写入的总长度  while( write\_len < want\_write\_len ){  if( write\_once\_offset == write\_once\_len ){  write\_once\_offset = 0;  int left\_write\_len = want\_write\_len - write\_len;  // 每次写的长度不能超过剩余需要写的长度  write\_once\_len = write\_once\_len > left\_write\_len ? left\_write\_len : write\_once\_len;  // 读取原始数据，此处以假数据模拟  for(int i = 0; i < write\_once\_len; i++) {  buf[i] = (byte)i;  }  }  int ret = writer.write( buf, write\_once\_offset, write\_once\_len - write\_once\_offset );  if( 0 > ret ){  System.out.println( "write failed" );  break;  }    write\_once\_offset += ret;  write\_len += ret;  } |

* **提示**
  + 7.2版本之前，该接口为异步写入，缓存未满的情况下，总是返回成功，但其实际写入结果未知。
  + 全部写入的数据长度不能超过100MB，超过时会返回失败。

#### 关闭文件

* **接口定义**

String ArchiveWriter.close();

* **功能描述**

关闭文件，并返回文件名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + 非null：成功，返回文件名
  + null：失败。
* **示例**

|  |
| --- |
| String filename = writer.close(); |

* **提示**
  + close()并不保证文件已经写入云存储，因此获取到文件名后，可能暂时无法读取，且可能因为异步写入失败导致永远无法读取。7.2版本之后有效解决该问题。
  + 一旦open()成功，则必须调用close()，否则会导致下一次open()失败
  + 如果write()失败，仍需调用close()，此时close()返回的文件名只能读取到已经写入的数据。

#### 释放

* **接口定义**

**boolean ArchiveWriter.release();**

* **功能描述**

释放ArchiveWriter对象占用的资源。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + 无需关注返回值
* **示例**

|  |
| --- |
| // 当ArchiveWriter对象不再使用时，必须调用ArchiveWriter.release(),否则会存储在内存泄露  writer.release(); |

* **提示**
  + 一旦ArchiveTool. createArchiveWriter()返回非null，则必须调用该接口，否则会导致内存泄露

#### 设置打包使用的Bucket

* **接口定义**

**boolean** ArchiveWriter**.setBucket( String bucketname );**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **bucketname** | [IN ] bucket名。必须是已经存在的bucket |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| String bucket\_name = “20180724”;  if( writer.setBucket( bucket\_name ) ){  // 设置成功后，打包后的文件将会存储在新bucket中  }else{  // 设置失败，则打包后的文件仍存储在原先的bucket中  } |

* **提示**
  + 不推荐对一个小文件打包写对象反复设置不同的Bucket。在7.2版本前，这会影响打包效果；在7.2版本后，没有影响。

#### 获取打包使用的Bucket

* **接口定义**

**String** ArchiveWriter**.getBucket();**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + ArchiveWriter使用的bucket名
* **示例**

|  |
| --- |
| String bucket\_name = writer.getBucket(); |

#### 获取域名

* **接口定义**

**String** ArchiveWriter**.getRegionName();**

* **功能描述**

获取创建小文件打包对象时指定的域名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + 返回域名。当前不建议使用域名

#### 设置最大能缓存的文件个数（7.0后已废弃）

* **接口定义**

**boolean** ArchiveWriter**.setMaxCache( int number )**;

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 获取最大能缓存的文件个数（7.0后已废弃）

* **接口定义**

int ArchiveWriter.getMaxCache();

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 获取当前缓存的文件个数（7.0后已废弃）

* **接口定义**

int ArchiveWriter.getCurrentCache();

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

### 下载文件

使用同一个ArchiveReader对象读取相同前缀的文件，可获取最佳性能。写入时返回的文件名格式为bucket/archivefile-xxxx-xx-xx-xxxxxx-xxxxxxxxxxxxxxxx:offset/size.suffix。标红字段相同就是前缀相同，意味着小文件打包在同一个大文件中。

#### 创建

* **接口定义**

ArchiveReader ArchiveTool.createArchiveReader();

* **功能描述**

创建小文件打包读对象。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**
  + 非null：成功。
  + null：失败。
* **示例**

|  |
| --- |
| ArchiveReader reader = archiveTool.createArchiveReader();  if( null == reader ){  System.out.println( "create ArchiveReader failed" );  return;  } |

* **提示**
  + ArchiveReader一旦创建后可以进行任意次数的打开、读取、关闭操作，无需读取一个文件就创建一次ArchiveReader。
  + 请尽可能复用一个ArchiveReader，避免反复创建和释放。

#### 打开文件

* **接口定义**

long ArchiveReader.open(String filename);

* **功能描述**

打开指定的文件，并获取文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| filename | [IN ] 指定的文件名 |

* **接口返回值**
  + >0：文件长度
  + 0：失败。
* **示例**

|  |
| --- |
| long file\_length = reader.open( filename );  if( 0 == file\_length ){  System.out.println( "open failed" );  return false;  } |

* **提示**
  + 一旦ArchiveReader.open()成功，则最后必须调用ArchiveReader.close()

#### 读文件

* **接口定义**

int ArchiveReader.read(byte[] buffer, int len);

* **功能描述**

读取指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buffer | [IN ] 保存读取数据的缓存 |
| length | [IN ] 待读取的长度 |

* **接口返回值**
  + >0：实际读取的成都。
  + =0：暂时不可读取
  + -1：失败。
  + -2：读取到文件尾
* **示例**

|  |
| --- |
| int want\_read\_len = (int)file\_length; // 小文件打包写入的文件不会超过int最大值  int read\_once\_len = 12288; // 推荐每次读取的长度是4K\*N的倍数，此处N是3  byte[] buf = new byte[read\_once\_len]; // 读取数据的缓存  int read\_len = 0;  while( read\_len < want\_read\_len ){  int ret = reader.read( buf, read\_once\_len );  if( 0 < ret ){  read\_len += ret;  int left\_read\_len = (int)(want\_read\_len - read\_len);  // 每次读取的长度不能超过剩余需要读取的长度  read\_once\_len = read\_once\_len > left\_read\_len ? left\_read\_len : read\_once\_len;  // 对读取到的数据进行操作，比如写入本地文件  // 读取多少就写入多少  // write( fd, buf, ret );  }else if( 0 == ret ){  // 休眠1ms，必须有休眠，否则可能导致CPU占用上升  try{  Thread.sleep(1);  } catch (InterruptedException e) {  e.printStackTrace();  };  }else if( -1 == ret ){  System.out.println( "read failed, error:" + EFileSystem.getLastError() );  break;  }else if( -2 == ret ){  // 读取到文件尾  System.out.println( "read end" );  break;  }  } |

* **提示**
  + 如果必须保证文件数据是完整的，比如图片这类文件，那么请判断read\_len是否等于want\_read\_len，如果相等则认为读取成功，否则认为读取失败。

#### 关闭文件

* **接口定义**

boolean ArchiveReader.close();

* **功能描述**

关闭文件

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| long file\_length = reader.open( filename );  if( 0 < file\_length ){  // 读取结束后，必须关闭  reader.close();  } |

* **提示**
  + 一旦open()成功，那么不论是否读取失败，都必须调用close()接口，否则会导致下一次open()失败。

#### 释放

* **接口定义**

boolean ArchiveReader.release();

* **功能描述**

释放ArchiveReader占用的资源

* **接口参数**

**无入参**

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // 一般无需关心返回值  reader.release(); |

* **提示**
  + 该接口仅在确定不再使用该ArchiveReader对象后调用，尽可能重复使用同一个ArchiveReader，避免频繁release()。

#### 解析文件名

* **接口定义**

**ArchiveInfo ArchiveTool.getArchiveInfos( String archname );**

* **功能描述**

解析ArchiveWriter.close()返回的文件名，从中解析出域名，不含域名的文件名以及文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| archname | [IN ] 指定的文件名 |

* **接口返回值**
  + 非null：解析后的信息
  + null：失败。
* **示例**

|  |
| --- |
| // archive\_tool是ArchiveTool对象  ArchiveInfo archiveInfo = archive\_tool.getArchiveInfos(archname);  System.out.println("region : " + archiveInfo.getRegionName());  System.out.println("filename : " + archiveInfo.getFilename());  System.out.println("file length : " + archiveInfo.getLength()); |

* **提示**
  + 如果archname本身就不含域名，则filename就是archname

## Demo



demo中的example.java里包含所有功能的实现，请详细阅读，如实现存在错误，请联系我们。为了减小文档大小，Demo中的库全部truncate为0字节，请在编译前手动替换成同名的库，并修改example.java文件中的EFS服务器地址、用户名和密码。

可以使用Eclipse直接打开工程。

## 常见问题

### Exception in thread "main" java.lang.UnsatisfiedLinkError

执行java程序时出现该异常，均是因为没有参考[安装](#_安装)章节将动态库放在可被加载的目录。

### 与Java版本的EFS SDK一起使用时发生崩溃

如果程序中仅使用Archive-SDK或者仅使用EFS-SDK时不存在问题，但是共同使用时，出现崩溃或者莫名其妙的问题，可尝试将将动态库升级到7.2之后的版本（含7.2）。如果问题不存在了，那么可能是该[问题](#_与C版本的EFS_SDK一起使用时发生崩溃或莫名其妙的问题)。

# C SDK

## 前言

### 获取

联系项目对接人员获取对应版本的安装包，版本名规则如下：

DH\_EFS\_ARCHIVETOOL\_[Platform]\_[bits]\_PS\_[Version].tar.gz

实例如下：

* Linux
  + 32位：DH\_EFS\_ARCHIVETOOL\_32bit\_PS\_V1.072.0000004.0.R.180723.tar.gz
  + 64位：DH\_EFS\_ARCHIVETOOL\_64bit\_PS\_V1.072.0000004.0.R.180723.tar.gz
* Windows
  + 32位：

DH\_EFS\_ARCHIVETOOL\_Windows\_32bit\_PS\_V1.072.0000004.0.R.180723.tar.gz

* + 64位：

DH\_EFS\_ARCHIVETOOL\_Windows\_64bit\_PS\_V1.072.0000004.0.R.180723.tar.gz

上述实例中的[Version]的072，代表为7.2版本，可以此类推。

安装包内包含库以及头文件，头文件在archivetool\_include.tar中的ArchiveTool.c目录。

### 编译

需要包含的头文件：ArchiveAdapter.h

指定链接的库：

* Linux平台
  + 32位
    - libEFSArchiveTool.so
    - libEFS EFSAdapter.so
  + 64位：
    - libEFSArchiveTool64.so
    - libEFSAdapter64.so
* windows平台
  + 32位
    - EFSArchiveTool.lib（EFSArchiveTool.dll的导入库文件）
    - EFSAdapter.lib（EFSAdapter.dll的导入库文件）
  + 64位
    - EFSArchiveTool64.lib（EFSArchiveTool64.dll的导入库文件）
    - EFSAdapter64.lib（EFSAdapter64.dll的导入库文件）

具体编译过程可参考[Demo](#_Demo_1)

### 安装

动态库必须安装在动态库加载目录中，否则可能在运行时出现无法找到库的错误。

动态库加载目录如下：

* Linux平台
  + 系统默认目录，比如32位的/lib, /usr/lib；64位的/lib64,/usr/lib64等目录
  + LD\_LIBRARY\_PATH环境变量指定的目录（推荐）
  + ldconfig配置文件中指定的目录
  + 在编译时通过`-Wl,-rpath`指定的目录
* windows平台
  + 应用程序所在目录（推荐）
  + 启动程序的目录
  + Windows SYSTEM目录
  + Windows目录
  + PATH环境变量指定的路径

必须安装的动态库：

* Linux
  + 32位
    - libEFSArchiveTool.so
    - libEFSAdapter.so（取自EFS-SDK包）
    - libEFSClient.so （取自EFS-SDK包）
    - libEFSClientCore.so（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
  + 64位
    - libEFSArchiveTool64.so
    - libEFSAdapter64.so（取自EFS-SDK包）
    - libEFSClient64.so （取自EFS-SDK包）
    - libEFSClientCore64.so（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
* Windows
  + 32位
    - EFSArchiveTool.dll
    - EFSAdapter.dll（取自EFS-SDK包）
    - EFSClient.dll（取自EFS-SDK包）
    - EFSClientCore.dll（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
    - EFSHelper.dll（取自EFS-SDK包）
  + 64位
    - EFSArchiveTool64.dll
    - EFSAdapter64.dll（取自EFS-SDK包）
    - EFSClient.dll（取自EFS-SDK包）
    - EFSClientCore.dll（取自EFS-SDK包，5.0版本后的EFS-SDK才有）
    - EFSHelper64.dll（取自EFS-SDK包）

### 升级

获取新版本的安装包，取出其中的库直接替换即可。

因为ArchiveTool可使用任意版本的SDK，所以可以在不升级EFS SDK和云存储的情况，使用最新版本的ArchiveTool。

EFS SDK的升级请阅读《EFS-SDK\_UserDocs.docx》。

### 兼容性

目前对外接口不存在兼容问题。

内部读写功能均向前兼容。

## 使用介绍

以下介绍时，不涉及EFS SDK。请参考Demo，了解结合EFS SDK的使用。

请注意：在小文件打包中重复实现了EFS SDK的部分功能，如创建bucket、设置日志路径和设置日志级别等，请忽略这些接口，以下文档没有介绍的接口请选择使用EFS SDK中对应的接口。小文件打包功能本身仅需使用以下介绍的接口（如果头文件中的接口下面没有介绍的，可以直接忽略，属于废弃接口）。

### 全局管理

#### 初始化

* + - 1. 无需调用接口进行初始化（推荐）。
      2. 某些较老的版本会使用efsCreate接口进行初始化，7.2以后的版本不推荐使用，老版本的会做兼容。

#### 释放

* **接口原型**

void archiveManagerRelease();

* **功能描述**

释放小文件打包使用的所有资源。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
|  |  |

* **接口返回值**

无返回值

* **示例**

|  |
| --- |
|  |

* **提示**
  + 一旦使用过小文件打包的任意功能，则必须调用该函数销毁后台资源。

### 上传文件

上传文件需要使用ArchiveWriterHandle。一个ArchiveWriterHandle不可多线程同时使用。

#### 创建

* **接口定义**

ArchiveWriterHandle createArchiveWriter( EFSHandle handle,

const char \*regionname );

* **功能描述**

创建小文件打包对象。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 必须是有效的EFSHandle |
| regionname | [IN ] 域名。不建议使用 |

* **接口返回值**

**需要使用isArchiveWriterHandleValid()检查返回的句柄是否有效。**

* **示例**

|  |
| --- |
| writer = createArchiveWriter( efs, NULL );  if( EFS\_TRUE != isArchiveWriterHandleValid( writer ) ){  printf( "create writer failed.\n" );  break;  } |

* **提示**
  + 传入的EFSHandle必须是有效的。EFSHandle的获取可参考《EFS-SDK\_UserDocs.docx》
  + 不建议使用regionname。如果设置了，那么在返回的小文件名中会携带该regioname。
  + 一旦返回的ArchiveWriterHandle有效，那么当不使用时，必须调用archiveReleaseWriter()进行释放，否则会造成资源泄露。
  + 尽可能重复使用一个ArchiveWriterHandle，避免反复创建和销毁ArchiveWriterHandle。7.0版本前，如果频繁初始化和销毁可能导致性能急剧下降。即使存在写入失败，ArchiveWriterHandle仍可用于下一次写入

#### 初始化

* **接口定义**

EFS\_BOOL archiveWriterInit( ArchiveWriterHandle handle,

const uint16\_t n,

const uint16\_t m,

const char \*bucketname);

* **功能描述**

初始化小文件打包写句柄，指定N+M以及文件所属的Bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 有效的写句柄 |
| n | [IN ] N+M的N |
| m | [IN ] N+M的M |
| bucketname | [IN ] 已经存在的Bucket名，用于存储打包后的文件 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| if( EFS\_TRUE != archiveWriterInit( writer, 3, 1, bucket\_name ) ){  printf( "init writer failed.\n" );  break;  } |

* **提示**
  + 一旦在该接口中指定N+M后，该ArchiveWriterHandle的N+M不可再改变，当前只支持3+1和7+2模式，其余模式会返回失败
  + 在该接口中指定的bucket，可以通过archiveGetBucket() 获取其名字
  + bucket可通过archiveSetBucket()重新设置，仍需保证设置的bucket是已经存在的。不推荐对一个ArchiveWriterHandle反复设置不同的bucket。

#### 打开文件

* **接口定义**

EFS\_BOOL archiveWriterOpen( ArchiveWriterHandle handle,

const char \*suffix );

* **功能描述**

打开文件，suffix指定该文件的后缀。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 有效的写句柄 |
| suffix | [IN ] 指定返回的文件名的后缀，可以为NULL |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| if( EFS\_TRUE != archiveWriterOpen( writer, "jpg" ) ){  printf( "open failed\n" );  return -1;  } |

* **提示**
  + 该接口指定的参数主要用于拼接archiveWriterClose()时返回的文件名，以便其他程序能正确处理文件，比如以jpg作为后缀，可以使得其他程序能以jpg格式解析文件。

#### 写文件

* **接口定义**

int32\_t archiveWrite( ArchiveWriterHandle handle,

const char\* buffer,

const uint32\_t length );

* **功能描述**

写入指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 写句柄 |
| buffer | [IN ] 数据缓存地址，不能为NULL |
| length | [IN ] 待写入的数据长度，不能为0 |

* **接口返回值**
  + >0：实际写入的长度
  + <0：失败。需关闭，不可再写。
* **示例**

|  |
| --- |
| uint32\_t want\_write\_len = 1 << 20; // 文件大小为1M  uint32\_t write\_once\_len = 12288; // 推荐每次写入N\*4k倍数的数据，此处N为3  char\* buf = (char\*)malloc( write\_once\_len ); // 待写入数据的缓存  uint32\_t write\_once\_offset = write\_once\_len; // 初始设为write\_once\_len，保证进入读取原始数据的分支  uint32\_t write\_len = 0; // 已写入的总长度  while( write\_len < want\_write\_len ){  if( write\_once\_offset == write\_once\_len ){  write\_once\_offset = 0;  uint32\_t left\_write\_len = want\_write\_len - write\_len;  // 每次写的长度不能超过剩余需要写的长度  write\_once\_len = write\_once\_len > left\_write\_len ? left\_write\_len : write\_once\_len;  // 读取原始数据，此处以memset模拟  memset( buf, 0, write\_once\_len );  }  int ret = archiveWrite( writer, buf + write\_once\_offset, write\_once\_len - write\_once\_offset );  if( 0 > ret ){  printf( "write failed\n" );  break;  }    write\_once\_offset += ret;  write\_len += ret;  } |

* **提示**
  + 7.2版本之前，该接口为异步写入，缓存未满的情况下，总是返回成功，但其实际写入结果未知。
  + 全部写入的数据长度不能超过100MB，超过时会返回失败。

#### 关闭文件

* **接口定义**

EFS\_BOOL archiveWriterClose( ArchiveWriterHandle handle, char\* filename );

* **功能描述**

关闭文件，并返回文件名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 写句柄 |
| filename | [OUT ] 返回的文件名。 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| // 打开文件成功后，必须关闭文件  if( EFS\_TRUE != archiveWriterClose( writer, filename ) ){  return -1;  } |

* **提示**
  + archiveWriterClose()并不保证文件已经写入云存储，因此获取到文件名后，可能暂时无法读取，且可能因为异步写入失败导致永远无法读取。7.2版本之后有效解决该问题。
  + 一旦archiveWriterOpen()成功，则必须调用archiveWriterClose()，否则会导致下一次对该句柄的archiveWriterOpen()失败
  + 如果archiveWrite()失败，仍需调用archiveWriterClose()，此时archiveWriterClose()返回的文件名只能读取到已经写入的数据。

#### 释放

* **接口定义**

**EFS\_BOOL archiveReleaseWriter(ArchiveWriterHandle handle);**

* **功能描述**

释放写句柄占用的资源

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 写句柄 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| archiveReleaseWriter( writer ); |

* **提示**
  + 一旦createArchiveWriter ()返回的句柄是有效的，则必须调用该接口，否则会导致内存泄露
  + 该接口必须在efsClose()前调用。

#### 设置打包使用的Bucket

* **接口定义**

**EFS\_BOOL archiveSetBucket( ArchiveWriterHandle handle,**

**const char\* bucketname );**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handler | [IN ] 写句柄 |
| **bucketname** | [IN ] bucket名。必须是已经存在的bucket |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| if( EFS\_TRUE == archiveSetBucket( writer, “new\_bucket” ) ){  // 设置成功后，打包后的文件将会存储在新bucket中  }else{  // 设置失败，则打包后的文件仍存储在原先的bucket中  } |

* **提示**
  + 不推荐对一个小文件打包写句柄反复设置不同的Bucket。在7.2版本前，这会影响打包效果；在7.2版本后，没有影响。

#### 获取打包使用的Bucket

* **接口定义**

**EFS\_BOOL archiveGetBucket( ArchiveWriterHandle handle, char\* bucketname );**

* **功能描述**

设置打包后的文件存储的bucket。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 写句柄 |
| **bucketname** | [OUT ] 获取的bucket名。 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| char bucket[64]; // bucket名不会超过64字节  if( EFS\_TRUE == archiveGetBucket( writer, bucket) ){    } |

#### 获取域名

* **接口定义**

EFS\_BOOL archiveGetRegionName( ArchiveWriterHandle handle, char \*regionname )**;**

* **功能描述**

获取创建小文件打包对象时指定的域名。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 写句柄 |
| regionname | [OUT ] 获取的域名。 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。

#### 设置最大能缓存的文件个数（7.0后已废弃）

* **接口定义**

**EFS\_BOOL archiveSetMaxCache( ArchiveWriterHandle handle,**

**const uint32\_t number )**;

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 获取最大能缓存的文件个数（7.0后已废弃）

* **接口定义**

uint32\_t archiveGetMaxCache(ArchiveWriterHandle handle);

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 获取当前缓存的文件个数（7.0后已废弃）

* **接口定义**

uint32\_t archiveGetCurrentCache(ArchiveWriterHandle handle);

* **提示**
  + 已经调用该接口的程序，无需修改，调用会返回成功，但实际不生效。

#### 一次性写入

* **接口定义**

**EFS\_BOOL archiveWriteOnce(ArchiveWriterHandle handle,**

**const char\* suffix,**

**const char\* buffer,**

**const uint32\_t length,**

**char\* filename );**

* **功能描述**

合并了archiveWriterOpen()、archiveWrite()、archiveWriterClose()功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 写句柄 |
| **suffix** | [IN ] 文件名后缀。 |
| **buffer** | [IN ] 数据缓存地址 |
| **length** | [IN ] 待写入数据的长度 |
| **filename** | [OUT] 写入成功后返回的文件名 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| const char\* suffix = "jpg";  uint32\_t length = 12288;  char\* buf = (char\*)malloc(length);  if( EFS\_TRUE != archiveWriteOnce( writer, suffix, buf, length, filename ) ){  printf( "wirteOnce failed\n" );  } |

* **提示**
  + 7.0版本后支持该接口

### 下载文件

下载文件需要使用ArchiveReaderHandle，

使用同一个ArchiveReaderHandle读取相同前缀的文件，可获取最佳性能。写入时返回的文件名格式为bucket/archivefile-xxxx-xx-xx-xxxxxx-xxxxxxxxxxxxxxxx:offset/size.suffix。标红的字段相同就是前缀相同，意味着小文件打包在同一个大文件中。

#### 创建

* **接口定义**

ArchiveReaderHandle createArchiveReader( EFSHandle handle );

* **功能描述**

创建小文件打包读句柄。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 已经初始化好的EFSHandle |

* **接口返回值**

**必须使用isArchiveReaderHandleValid()判断返回的句柄是否有效，只有有效才能继续使用。**

* **示例**

|  |
| --- |
| ArchiveReaderHandle reader = createArchiveReader( efs );  if( EFS\_TRUE != isArchiveReaderHandleValid( reader ) ){  printf( "create reader failed\n" );  return;  } |

* **提示**
  + 创建的ArchiveReaderHandle与一般句柄不同，一旦创建后可以进行任意次数的打开、读取、关闭操作，无需读取一个文件就创建一次ArchiveReaderHandle。
  + 请尽可能复用一个ArchiveReader，避免反复创建和释放。

#### 打开文件

* **接口定义**

EFS\_BOOL archiveReaderOpen( ArchiveReaderHandle handle,

const char\* filename,

uint64\_t\* length );

* **功能描述**

打开指定的文件，并获取文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 读句柄 |
| filename | [IN ] 指定的文件名 |
| length | [OUT ] 文件长度 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| uint64\_t file\_length = 0;  if( EFS\_TRUE != archiveReaderOpen( reader, filename, &file\_length ) ){  printf( "open %s failed\n", filename);  return -1;  } |

#### 读文件

* **接口定义**

int32\_t archiveRead( ArchiveReaderHandle handle,

char\* buffer,

const uint32\_t length );

* **功能描述**

读取指定长度的数据。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 读句柄 |
| buffer | [IN ] 保存读取数据的缓存 |
| length | [IN ] 待读取的长度 |

* **接口返回值**
  + >0：实际读取的成都。
  + =0：暂时不可读取
  + -1：失败。
  + -2：读取到文件尾
* **示例**

|  |
| --- |
| uint32\_t want\_read\_len = file\_length;  uint32\_t read\_once\_len = 12288; // 推荐每次读取的长度是4K\*N的倍数，此处N是3  char\* buf = new char[read\_once\_len]; // 读取数据的缓存  uint32\_t read\_len = 0;  while( read\_len < want\_read\_len ){  int ret = archiveRead( reader, buf, read\_once\_len );  if( 0 < ret ){  read\_len += ret;  uint32\_t left\_read\_len = want\_read\_len - read\_len;  // 每次读取的长度不能超过剩余需要读取的长度  read\_once\_len = read\_once\_len > left\_read\_len ? left\_read\_len : read\_once\_len;  // 对读取到的数据进行操作，比如写入本地文件  // 读取多少就写入多少  // write( fd, buf, ret );  }else if( 0 == ret ){  // 休眠1ms，必须有休眠，否则可能导致CPU占用上升  SLEEP\_MS( 1 );  }else if( -1 == ret ){  printf("read failed\n" );  break;  }else if( -2 == ret ){  // 读取到文件尾  printf( "read end\n" );  break;  }  } |

* **提示**
  + 如果必须保证文件数据是完整的，比如图片这类文件，那么请判断read\_len是否等于want\_read\_len，如果相等则认为读取成功，否则认为读取失败。

#### 关闭文件

* **接口定义**

EFS\_BOOL archiveReaderClose(ArchiveReaderHandle handle);

* **功能描述**

关闭

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 读句柄 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| // 打开文件成功后，必须关闭文件.一般无需关心结果  archiveReaderClose( reader ); |

* **提示**
  + 一旦archiveReaderOpen()成功，那么不论是否读取失败，都必须调用该接口，否则会导致下一次archiveReaderOpen()失败。

#### 释放

* **接口定义**

EFS\_BOOL archiveReleaseReader(ArchiveReaderHandle handle);

* **功能描述**

释放句柄占用的资源

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 读句柄 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| // 使用完毕必须释放  archiveReleaseReader( reader ); |

* **提示**
  + 该接口仅在确定不再使用该ArchiveReaderHandle后调用，尽可能重复使用同一个ArchiveReaderHandle，避免频繁释放。
  + 该接口必须在efsClose()前调用

#### 解析文件名

* **接口定义**

**EFS\_BOOL archiveGetInfos( const char\* archname,**

**char\* regionname,**

**char \*filename,**

**uint64\_t& size );**

* **功能描述**

解析archiveReaderClose ()返回的文件名，从中解析出域名，不含域名的文件名以及文件长度。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| archname | [IN ] 指定的文件名 |
| regionname | [OUT ] 域名 |
| filename | [OUT ] 不含域名的文件名 |
| size | [OUT ] 文件大小 |

* **接口返回值**
  + true：成功。
  + false：失败。
* **示例**

|  |
| --- |
| char region\_name[256] = {0};  char filename\_witout\_region[256] = {0};  uint64\_t file\_size = 0;  // NOTE: 该接口实现时，最后一个参数是引用，如果是纯C，则无法使用  if( EFS\_TRUE != archiveGetInfos( filename, region\_name, filename\_witout\_region, file\_size ) ){  printf( "parse filename failed\n" );  return -1;  } |

* **提示**
  + 如果archname本身就不含域名，则filename就是archname
  + 该接口使用了C++的引用特性，纯C不支持。

#### 一次性读取

* **接口定义**

EFS\_BOOL archiveReadOnce(ArchiveReaderHandle handle,

const char\* filename,

char\* buffer,

const uint32\_t length,

int32\_t timeout);

* **功能描述**

合并了archiveReaderOpen()、archiveRead()、archiveReaderClose()功能。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN ] 读句柄 |
| filename | [IN ] 指定的文件名 |
| buffer | [IN ] 数据缓存 |
| length | [IN ] 待读取的长度。 |
| timeout | [IN ] 超时时间，单位毫秒。-1表示永不超时 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。
* **示例**

|  |
| --- |
| char\* buf = (char\*)malloc( file\_size );  if( EFS\_TRUE != archiveReadOnce( reader, filename, buf, file\_size, -1 ) ){  printf( "readOnce failed\n" );  } |

* **提示**
  + 7.0版本后支持该接口
  + 需要借助**archiveGetInfos** ()获取文件长度，然后分配足够的缓存空间，才能调用readOnce()接口。因为**archiveGetInfos()中使用了引用，所以纯C下该接口无法调用。不建议自行解析文件名获取长度，避免命名规则变化导致的不兼容问题。**

## Demo

demo中的example.cpp里包含所有功能的实现，请详细阅读，如实现存在错误，请联系我们。为了减小文档大小，Demo中的库全部truncate为0字节，请在编译前手动替换成同名的库，并修改example.cpp文件中的EFS服务器地址、用户名和密码。

### Linux



解压后，执行make x86编译32位程序，make x86\_64编译64位程序，执行make test\_x86或make test\_x86\_64演示简单的上传和下载功能。

### Windows



示例包含vs2005的工程，其中include目录为头文件目录，libs目录为链接库目录，生成的程序demo-c位于bin目录，bin目录下已放置了EFSArchiveTool.dll、EFSAdapter.dll、EFSClient.dll、EFSClientCore.dll和EFSHelper.dll，可直接在该目录运行demo-c，演示简单的上传下载功能。

提供的Demo是Release 32位的，请参照该Demo，自行搭建64位工程。

## 常见问题

### 类型重复定义和函数定义冲突导致的编译失败

C语言版本的小文件打包中包含了部分C语言版本EFS-SDK的类型和函数，这会导致类型重复定义或者函数定义冲突使得编译失败。

解决办法：请手动注释掉ArchiveAdapter.h中以下类型和函数

* struct \_\_EFSConfig
* struct EFSHandle
* int32\_t efsCreateBucket(EFSHandle, const char\*)

或者将ArchiveAdapter.h更新到7.2版本

### 与C版本的EFS SDK一起使用时发生崩溃

如果程序中仅使用Archive-SDK或者仅使用EFS-SDK时不存在问题，但是共同使用时，出现崩溃或者莫名其妙的问题，那么可尝试将动态库和头文件都升级到7.2之后的版本（含7.2），然后重新编译程序。如果问题消失，那大概率就是7.2版本前问题，不过请先排除其他代码问题。下面是问题原因，可作为问题定位的参考。

自5.0开始，EFS-SDK新增了动态升级功能，以Linux32位为例，EFS-SDK提供的动态库变为libEFSAdapter.so、libEFSClient.so、libEFSClientCore.so（原来只有libEFSAdapter.so、libEFSClient.so）

5.0至7.2版本之间，这三个库的链接关系为

* libEFSAdapter.so -> libEFSClientCore.so
* libEFSClient.so 中动态加载libEFSClientCore.so

Archive-SDK的libEFSArchiveTool.so一直链接的是libEFSClient.so。

当两者共同使用时，进程即加载libEFSClient.so又加载了libEFSClientCore.so。

因为最初Archive-SDK C接口设计时，包含了部分小文件打包依赖的EFS-SDK的功能，其接口名与EFS-SDK的接口是一样的，但是内部结构体不同，再因为libEFSClient.so和libEFSClientCore.so的对外接口是一样的，但实现不同，所以可能出现接口调用时执行的代码是非预期的，也就是可能出现使用Archive-SDK创建的EFSHandle被EFS-SDK使用了，通过libEFSClientCore.so的接口创建的资源，却在libEFSClient.so中尝试使用。

# 配置文件介绍

6.0版本后，小文件打包新增了一个配置文件archive.conf，格式为JSON。该配置文件主要用于设置本地存储、日志路径等。当不指定配置文件路径时，内部默认该配置文件位于“./archive.conf”，也就是启动程序的目录（非程序所在目录）。特别需要注意的是Java和C接口中没有指定配置文件路径的接口，也就是说只能将配置文件放在启动程序的目录下才能生效。

如果不存在配置文件，内部会使用默认配置。这在6.0版本中可能会存在问题。因为在6.0版本中，默认配置使能了本地存储，即enable\_local\_storage字段值为1。如果程序部署在没有Datanode服务的设备上，那么本地存储将会初始化失败，这会导致小文件打包功能异常。7.0之后的版本，默认不使能本地存储，因此无该问题。

如果使用该配置文件，则一般只需修改以下字段

* enable\_local\_storage：0-不使能本地存储，1-使能本地存储
* deploy\_mode：仅在enable\_local\_storage为1时有效，0-自动获取本地存储路径，1-手动填写本地存储路径。只有当程序部署在有Datanode服务的设备上时，才能设置为0。
* local\_storage\_path：仅在enable\_local\_storage为1且deploy\_mode为1时有效，用于指定本地存储路径
* efs\_log\_level：日志级别
* efs\_log\_path：日志路径
* set\_log\_cache：日志缓存。1-日志不实时刷新到磁盘，0-日志实时刷新到磁盘

# 本地存储

6.0版本之后，小文件打包新增了本地存储功能。本地存储的作用是当云存储发生异常时，数据会被存储到本地磁盘中，当云存储恢复正常时，这些数据会被恢复到云存储中。当数据处于本地时，无法访问这些数据，只有等到恢复后，才能访问。当前的云存储异常仅指元数据服务的异常，比如客户端与元数据服务存在网络异常，元数据服务本身异常等。

# 最佳实践

### 写入

1. 创建一定数量的常驻线程用于写入，线程数可通过配置项进行调整。
2. 每个线程创建一个小文件打包写对象，其生命期与线程的生命期相同。
3. 写线程根据当前日期创建Bucket，每一天创建一个新的Bucket。不同线程之间的Bucket日期相同但后缀不同。
4. 若小文件打包写对象未初始化，则以初始化接口设置Bucket，否则以设置Bucket接口更新小文件打包写对象使用的Bucket。
5. 写线程循环获取新文件，并使用小文件打包对象将文件写入云存储，保存关闭时返回的文件名。

### 读取

对于随机读取不同天不同时间段的文件时，目前暂无特别推荐的处理流程。

对于批处理，比如要读取某个Bucket下大量图片进行图片分析，则建议先行将待查询的图片名进行排序，保证连续读取的小文件的前缀尽可能相同，即尽可能在一个大文件，然后使用同一个小文件打包读对象读取这些前缀相同的小文件，这样可有效减少资源开销并提升性能。

# 常见问题

## 小文件打包写入的文件是否支持删除？

不支持单独删除小文件打包写入的文件，但是可结合生命周期实现周期性回收空间的功能。