**EFS-SDK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 | 2013-12-01 | **1.0** |
| 2 | 新增bucket内容 | **2014-01-13** | **1.1** |
| 3 | 根据检视意见和新错误码修改 | 2014-01-17 | 1.2 |
| 4 | 接口返回值修改 | 2014-02-17 | 1.3 |
| 5 | 根据二期新接口修改 | 2014-05-19 | 2.0 |
| 6 | 新增流程图和C接口说明 | 2014-05-27 | 2.1 |
| 7 | 根据小文件特性修改接口 | 2014-06-21 | 3.0 |
| 8 | 修改SDK错误码 | 2014-08-16 | 3.1 |
| 9 | 根据最新版本修改用户手册 | 2015-08-20 | 4.0 |
| 10 | 用户手册板式更新 | 2016-08-06 | 5.0 |
| 11 | 云存储6.0版本新增接口更新 | 2016-09-28 | 6.0 |
| 12 | setOption接口增加读缓存模式设置 | 2016-11-14 | 6.1 |
| 13 | 用户手册勘误 | 2017-03-18 | 6.2 |
| 14 | 增加云存储7.0大文件Append功能 | 2017-05-23 | 7.0 |
| 15 | 用户手册勘误，增加FAQ章节 | 2018-05-03 | 7.1 |
| 16 | 用户手册全新改版 | 2018-06-05 | 8.0 |
| 17 | Java允许用户指定加载动态库路径 | 2019-01-28 | 8.1 |
| 18 | 新增历史版本BUG章节 | 2019-02-23 | 8.2 |

# 概述

本文档是大华云存储（EFileSystem，简称EFS）的开发指南，描述了大华云存储EFS-SDK的使用。EFS-SDK是EFS的一个网络客户端库，其提供接口实现文件创建、读写、删除、获取系统信息等功能。EFS-SDK接口包括C、C++、Java语言版本，简化用户的编程。

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。

# 使用说明

## 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**
* **bucket名前后不能有空格**
* 必须使用UTF-8编码
* 不区分大小写

保证bucket名称全局唯一的实例：

用户名+时间+随机数

## File

File是EFS存储数据的基本单元，不同于一般文件系统，不同File之间不存在任何目录关系。但用户在使用上可以通过File的文件名使之存在一定关系，通过EFS-SDK提供的接口实现一般文件系统中的目录操作。

File必须属于某一个Bucket，因此File的创建、读写、删除必须指定全路径“<bucketname>/<filename>”。File分为BigFile和SmallFile，BigFile支持的最大大小为2TB，SmallFile支持的最大大小为100MB。

* <bucketname>/<filename>必须是唯一的，否则创建会失败

File的命名规范如下：

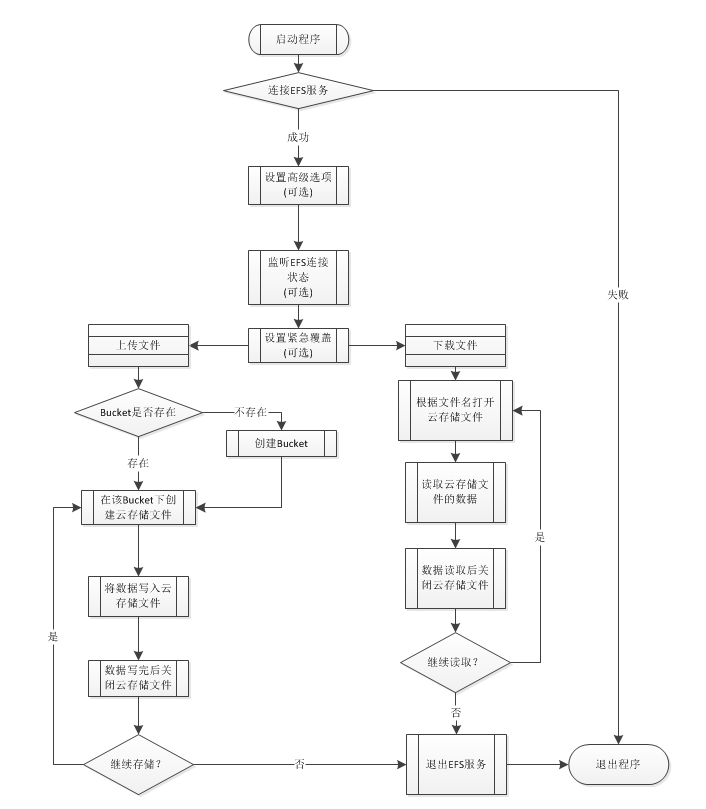
* <bucketname>/<filename>长度必须小于等于255字节，其中bucketname的命名规范见上文
* filename不能包含\ \* ? “ < > | ‘ :这9个字符
* **filename前后不能有空格**
* 必须使用UTF-8编码
* 不区分大小写，推荐以全小写方式命名

## 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。

# 快速入门

## 基本流程



# 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\_EFSSDK\_Eng\_[Platform]\_[bits]bit\_PS\_[Version].tar.gz

实例如下：

* Linux
  + 32位：DH\_EFSSDK\_Eng\_32bit\_PS\_V1.070.0001.5.R.180429.tar.gz
  + 64位：DH\_EFSSDK\_Eng\_64bit\_PS\_V1.070.0001.5.R.180429.tar.gz
* Windows
  + 32位：DH\_EFSSDK\_Eng\_Windows\_32bit\_PS\_V1.070.0001.5.R.180429.tar.gz
  + 64位：DH\_EFSSDK\_Eng\_Windows\_64bit\_PS\_V1.070.0001.5.R.180429.tar.gz

上述实例中的[Version]的070，代表为7.0版本，可以此类推。

安装包内包含库以及头文件，头文件在sdk\_include.tar中。

### 编译

需要包含的头文件：IntTypes.h、Defs.h、EFileSystem.h、Bucket.h、File.h

指定链接的库：

* Linux平台
  + 32位：libEFSClient.so，librt.so，libdl.so
  + 64位：libEFSClient64.so，librt.so，libdl.so
* windows平台
  + 32位：EFSClient.lib（EFSClient.dll的导入库文件）
  + 64位：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，不能包含中文路径。

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
    - libEFSClient.so
    - libEFSClientCore.so (仅在 5.0版本之后需要)
  + 64位
    - libEFSClient64.so
    - libEFSClientCore64.so (仅在 5.0版本之后需要)
* Windows
  + 32位
    - EFSHelper.dll
    - EFSClient.dll
    - EFSClientCore.dll (仅在 5.0版本之后需要)
  + 64位
    - EFSHelper64.dll
    - EFSClient64.dll
    - EFSClientCore64.dll (仅在 5.0版本之后需要)

### 升级

如果使用的是5.0版本之后的SDK和EFS，且服务端启动了SDK动态升级服务，那么客户端无需手动更新SDK库，当服务端升级后，SDK会从服务端下载最新的SDK库。否则需要获取新版本的安装包，取出其中的库直接替换即可。

SDK动态升级服务名为SDKLibUpgradeServer，可在EFS的元数据服务器上通过ps –ef|grep SDKLibUpgradeServer命令查看服务是否启动。

如果是从4.0+升级到5.0之后的版本时，需要特别注意。以Linux32位为例，除了更新libEFSClient.so，还需要安装libEFSClientCore.so，否则可能因为SDK动态升级服务未启动，不能正常下载使用libEFSClientCore.so，导致SDK初始化失败。

Linux下需要删除/tmp/sdk目录下的文件，Windows下需要删除程序所在目录的upgrade目录

### 兼容性

目前无接口兼容性问题。

4.0+SDK与7.0版本云存储配套存在问题，需使用更新版本的SDK（6.0版本之后含6.0）

6.0版本之后（含6.0）的SDK与4.0+版本之后的云存储均可配套使用。

## 使用介绍

### 初始化

以下接口均不允许多线程调用，且在进程生命周期内，一般只需调用一次。以下接口原型省略了命名空间Dahua::EFS。

#### 连接EFS服务

* **接口原型**

bool CEFileSystem::initialize( const Config& cfg );

* **功能描述**

初始化CEFileSystem对象，登陆到EFS服务端。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| cfg | [IN] 登陆配置信息  typedef struct Config  {  const char\* address; //EFS服务器IP地址，必传参数  uint32\_t port; //EFS服务端口，必传参数  const char\* ilename; //云存储用户名  const char\* password; //云存储用户密码  const char\* defaultRegion; //<默认region，可不填写  char reserved[116]; //保留字段  }Config; |

* **接口返回值**
  + true：初始化成功。
  + false：初始化失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| Dahua::EFS::Config cfg;  cfg.address = “192.168.1.1”; // EFS服务地址  cfg.port = 38100; // EFS服务端口，固定不变  cfg.userName = “root”;// 存储用户名  cfg.password = “123456”; // 存储用户密码  Dahua::EFS::CEFileSystem\* efs = new Dahua::EFS::CEFileSystem;  // 初始化  if( efs->initialize( cfg ) ){  // 使用efs进行操作，比如创建bucket和文件  // 关闭后不得再使用efs  efs->close();  }else{  std::cout << “initialize failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  delete efs;  efs = NULL; |

* **提示**
  + 您的工程中可以有多个CEFileSystem实例，也可以只有一个CEFileSystem实例。如无特殊需求，推荐只创建一个EFileSystem实例。
  + 因全局变量初始化顺序问题，所以不得以全局变量的方式实例化CEFileSystem对象，可以使用全局指针变量。
  + 如发生网络异常，内部会自动重连。如设置了监听，那么将会回调通知使用者。

#### 高级选项配置

* **接口定义**

bool CEFileSystem:: setOption(EFSOption key, const void\* value, uint32\_t len);

* **功能描述**

设置系统高级配置信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| key | [IN ]系统配置选项。  enum EFSOption  {  efsConcurrent, //并发数,取值范围[1, 20]  efsBufferSize, //文件缓存大小  efsTimeOut, //消息超时时间  efsLogOutput, //日志文件输出路径  efsLogLevel, //日志级别  efsRwMode, //读写模式(0异步1同步)  efsSmallFileBufferSize, //小文件缓存大小（7.2版本后弃用）  efsBufferMode ///<读缓存模式（7.2版本后弃用）  }; |
| value | [IN ]配置项的值。 |
| len | [IN]配置项value的大小。单位B。 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| Dahua::EFS::CEFileSystem\* efs = new Dahua::EFS::CEFileSystem;  // 设置日志目录(目录必须存在)。推荐设置，可以有效定位问题。（可实现为程序配置项）  const char\* log\_path = “./log”;  if( efs->setOption( Dahua::EFS::efsLogOutput, log\_path, strlen( log\_path ) ) ){  std::cout << “set log path” << std::endl;  }else{  std::cout << “set log path failed” << std::endl;  }  // 设置日志级别，对接过程中推荐设置为DEBUGF。（可实现为程序配置项）  // 可选值由Defs.h中的日志级别的枚举确定  int32\_t log\_level = Dahua::EFS::DEBUGF;  if( efs->setOption( Dahua::EFS::efsLogLevel, &log\_level, sizeof( log\_level ) ) ){  std::cout << “set log level” << std::endl;  }else{  std::cout << “set log level failed” << std::endl;  } |

* **提示**
  + 推荐设置日志目录和级别，方便定位问题。日志文件占用最大空间为30\*50MB，会自行覆盖。对接测试时建议设置为DEBUG，发布时可配置为INFO。因有改动需求，建议实现为配置项。
  + efsConcurrent默认为8，为测试时最佳性能值，一般无需修改
  + efsBufferSize默认8个条带的大小，大文件的默认缓存是8\*32K\*(N+M)，小文件的默认条带是8\*4K\*(N+M)
  + 其他配置一般无需设置，可维持默认。
  + Windows下设置日志目录时请使用Linux格式，即选择以”/”作为目录分割符，比如d:/a/b/c

#### 监听EFS连接状态

* **接口定义**

bool CEFileSystem::setListener(IEFSListener\* listener);

* **功能描述**

监听EFS连接状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| listener | [IN ] 文件系统监听者，为NULL时表示删除监听。  IEFSListener内部有一个虚函数  virtual void onEFSStateChanged(CEFileSystem\* efs, EFSState state)其中，efs为需要设置的文件系统实例，state表示需要监控的文件系统状态。  enum EFSState {  efsStateInit, //初始状态  efsStateNormal, //efs客户端正常  efsStateException //efs客户端异常，会进行自动恢复  efsStateError ///<efs客户端错误，可能是用户不存在或者密码错误，不可自动恢复，需要用户介入.  }; |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| // 继承自Dahua::EFS::IEFSListener，实现EFS状态处理  class EFSListener : public Dahua::EFS::IEFSListener  {  public:  virtual void onEFSStateChanged( Dahua::EFS::CEFileSystem\* efs, Dahua::EFS::EFSState state )  {  // 可根据EFS状态进行处理，  // 常规用法是：当EFS连接状态异常时，切换为其他存储模式，保证数据安全  std::cout << “EFS state is “ << state << std::endl;  return;  }  };  // 监听对象  EFSListener\* listener = new EFSListener;  // 监听文件状态  if( efs->setListener( listener ) ){  std::cout << “set efs listener” << std::endl;  }else{  // 一般不会失败  }  // 如果与EFS连接状态出现变化，会调用EFSListener::onEFSStateChanged()  // 设置为NULL，不再监听EFS状态  if( efs->setListener( NULL ) ){  std::cout << “set efs listener to NULL” << std::endl;  }else{  // 一般不会失败  }  // 销毁监听对象  delete listener;  listener = NULL; |

* **提示**
  + 7.2版本前，多次调用该接口，则只有最后一次生效。
  + 7.2版本后，允许在一个CEFileSystem对象上设置多个监听，但是一旦设为NULL则清除所有监听。
  + 一般场景下，无需使用该功能。
  + 由于是异步回调，请注意锁的使用，避免加锁调用CEFileSystem类的接口，避免死锁

#### 设置用户空间紧急覆盖策略

* **接口定义**

bool CEFileSystem::setRecyclePolicy( bool enableRecycle , RecyclePolicy policy);

* **功能描述**

设置用户空间回收策略。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| enableRecycle | [IN ] 是否使能用户空间回收 |
| policy | [IN ] 回收策略  enum RecyclePolicy{  reduceByTime = 0, ///<等时间缩减  reduceByRatio = 1, ///<等比例缩减  }; |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| bool enable\_recycle = true; // 使能紧急覆盖  Dahua::EFS::RecyclePolicy policy = Dahua::EFS::reduceByTime; // 等时间删除  // 设置紧急覆盖后，只对该用户下的设置了生命周期的Bucket起作用  bool ret = efs->setRecyclePolicy( enable\_recycle, policy );  if( ret ){  std::cout << “set recycle policy” << std::endl;  }else{  std::cout << “set recycle policy failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 6.0版本后支持该功能。
  + 紧急覆盖仅对使能生命周期的Bucket有效。默认情况下，用户的使用容量达到95%时，开始删除文件，释放空间。
  + 等时间缩减指该用户下所有Bucket删除<Bucket生命周期+固定时间>时间点前创建的文件
  + 等比例缩减指该用户下所有Bucket删除<Bucket生命周期\*固定比例>时间点前创建的文件

#### 获取用户空间紧急覆盖策略

* **接口定义**

bool CEFileSystem::getRecyclePolicy( bool& enableRecycle, RecyclePolicy& policy);

* **功能描述**

获取用户空间回收策略配置。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| enableRecycle | [OUT ] 是否使能用户空间回收 |
| policy | [OUT ] 回收策略  enum RecyclePolicy{  reduceByTime = 0, ///<等时间缩减  reduceByRatio = 1, ///<等比例缩减  }; |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| bool enable\_recycle; // 使能紧急覆盖  Dahua::EFS::RecyclePolicy policy; // 等时间删除  // 获取紧急覆盖策略  ret = efs->getRecyclePolicy( enable\_recycle, policy );  if( ret ){  std::cout << “get recycle policy” << std::endl;  }else{  std::cout << “get recycle policy failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 6.0版本后支持该功能。

#### 退出EFS服务

* **接口定义**

bool CEFileSystem::close();

* **功能描述**

关闭文件系统。

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

|  |
| --- |
| Dahua::EFS::CEFileSystem\* efs = new Dahua::EFS::CEFileSystem;  if( efs->initialize( cfg ) ){  // 初始化成功后，当不再使用时，必须关闭  (void)efs->close();  delete efs;  efs = NULL;  }else{  std::cout << “initialize failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 调用CEFileSystem::close()前，请务必保证通过该对象创建或打开的文件都已经关闭。
  + 一般无需判断CEFileSystem::close()的返回值。
  + CEFileSystem::close()仅在确定后续不使用CEFileSystem实例时调用，比如进程退出前。
  + 一旦调用CEFileSystem::close()后，该CEFileSystem实例不可再使用。

### 管理Bucket

以下接口允许多线程调用，接口原型省略了命名空间Dahua::EFS。

#### 创建Bucket

* **接口定义**

**CBucket CEFileSystem::createBucket( const char\* name );**

* **功能描述**

**创建Bucket**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + **CBucket**的对象实例。必须使用**CBucket**:: isValid()来检测其有效性。若**CBucket**:: isValid()返回false，则表示创建失败，可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->createBucket( bucket\_name );  // 必须检查返回的CBucket对象是否有效，无效说明创建失败了  if( bucket.isValid() ){  std::cout << “create bucket “ << bucket\_name << std::endl;  // 操作CBucket对象  // ...  }else if( -80005 == Dahua::EFS::getLastError() ){  // 错误码是-80005（Bucket已经存在），则需要通过获取Bucket接口获得CBucket对象  }else{  std::cout << “create bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 创建的Bucket的访问权限继承自存储用户的访问权限，可通过接口重新设置
  + Bucket名字必须全局唯一，您需要保证Bucket名不会与其他用户冲突。
  + 当创建失败且错误码是-80005（Bucket已存在）时，可直接获取Bucket。
  + 因为单个Bucket下的文件过多时，会影响读取性能，所以推荐每天创建不同的Bucket进行存储。
  + 创建bucket的数量不要超过运维上面创建用户时指定的bucket数量，不然会报-60010
  + Bucket命名规范可以参考：[bucket注意事项](#bucket注意事项)

#### 删除Bucket

* **接口定义**

**bool CEFileSystem::****removeBucket( const char\* name );**

* **功能描述**

**删除指定名称的bucket。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **name** | **[IN] 需要删除的bucket名称** |

* **接口返回值**
  + true：删除成功。
  + false：删除失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket1”; // bucket名必须符合命名规范  if( efs->removeBucket( bucket\_name ) ){  std::cout << “remove bucket “ << bucket\_name << std::endl;  }else{  std::cout << “remove bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 删除Bucket前，必须先删除bucket下的所有文件，不然会返回失败，错误码为-80006。

#### 获取Bucket

* **接口定义**

**CBucket CEFileSystem::getBucket( const char\* name );**

* **功能描述**

**根据bucket名称获取bucket对象。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + **CBucket**的对象实例。必须使用**CBucket**:: isValid()来检测其有效性，若**CBucket**:: isValid()返回false，则表示获取失败，可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucket\_name );  if( bucket.isValid() ){  std::cout << “get bucket “ << bucket\_name << std::endl;  // 操作CBucket对象  // ...  }else if( -80004 == Dahua::EFS::getLastError() ){  // 错误码是-80004（Bucketn不存在），则需要通过创建Bucket接口获得CBucket对象  }else{  std::cout << “get bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 当调用该接口失败且错误码是-80004（Bucket不存在）时，需要先创建Bucket

#### 列出Bucket

* **接口定义**

**bool CEFileSystem::listBucket(CToken& token, uint32\_t maxNumber, CNameList& list);**

* **功能描述**

列出Bucket。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **Token** | **[IN] 记录查询Bucket的分页信息** |
| **maxNumber** | **[IN] 最多可能列出的Bucket个数，建议取值小于1024。** |
| **list** | **[OUT] 存放列出的bucket名** |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| Dahua::EFS::CToken token;  do{  // 接口内部会修改token的值，保证依次列出bucket  // 如果在执行的过程中有增删bucket的操作，那么此处列出的Bucket可能不准确。  Dahua::EFS::CNameList namelist; // 每次list前必须重新构造，否则可能包含重复的信息  if( efs->listBucket( token, 1024, namelist ) ){  // 输出bucket名字  std::cout << “list bucket, num:” << namelist.size() << std::endl;  for( int32\_t I = 0; I < namelist.size(); ++I ){  std::cout << namelist.getName( I ) << std::endl;  }    if( namelist.size() < 1024 ){  // bucket已经全部获取到  break;  }  }else{  // 获取失败，不再获取  std::cout << “list bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  break;  }  }while( 1 ); |

* **提示**
  + **在用户list过程中，如存在删除bucket的操作，list可能存在遗漏的情况。**

#### 设置Bucket访问权限

* **接口定义**

**bool CBucket::setPrivilege(AccessPrivilege privilege);**

* **功能描述**

设置bucket权限。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **Privilege** | **[IN] 需要设置的bucket权限。**  enum AccessPrivilege {  privilegePrivate= 0, //私有，其他用户不可见  privilegeRead = 1, //只读，其他用户无创建、删除、写入权限  privilegeWrite = 2 //读写，其他用户有创建、删除、写入权限  }; |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_pri”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->createBucket( bucketName );  // 检查bucket是否有效  if( bucket.isValid() ){  // 设置bucket的访问权限  if( bucket.setPrivilege( Dahua::EFS::privilegePrivate ) ){  std::cout << “set bucket privilege: Private” << std::endl;  }else{  std::cout << “set bucket privilege failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “create bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + bucket默认访问权限为存储用户的访问权限
  + 无特殊需求的情况下无需重新设置Bucket的访问权限

#### 查询Bucket访问权限

* **接口定义**

**bool CBucket::getPrivilege(AccessPrivilege& privilege);**

* **功能描述**

**获取bucket权限。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **Privilege** | **[OUT] 获取到的bucket权限。**  enum AccessPrivilege {  privilegePrivate= 0, //私有，其他用户不可见  privilegeRead = 1, //只读，其他用户无创建、删除、写入权限  privilegeWrite = 2 //读写，其他用户有创建、删除、写入权限  }; |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_pri”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->createBucket( bucketName );  // 检查bucket是否有效  if( bucket.isValid() ){  // 获取bucket的访问权限  Dahua::EFS::AccessPrivilege pri;  if( bucket.getPrivilege( pri ) ){  std::cout << “The privilege of “ << bucketName << “ is “ << pri << std::endl;  }else{  std::cout << “get privilege failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “create bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

#### 设置Bucket配额容量

* **接口定义**

**bool CBucket::****setCapacity( uint64\_t** **megaBytes );**

* **功能描述**

**设置bucket最大容量。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **megaBytes** | **[IN] 需要设置的bucket容量大小，单位MB。-1代表不限制配额** |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_cap”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->createBucket( bucketName );  // 检查bucket是否有效  if( bucket.isValid() ){  // 设置bucket配额容量为1024MB  if( bucket.setCapacity ( 1024 ) ){  std::cout << “set bucket capacity” << std::endl;  }else{  std::cout << “set bucket capacity failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “create bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + Bucket默认配额容量是存储用户的剩余容量，同一用户下的Bucket抢占式使用存储用户的剩余容量。
  + 无特殊需求时无需设置Bucket的配额容量
  + 一旦设置Bucket配额容量，这部分容量将会从用户剩余容量中扣除，被该Bucket独占式使用。
  + 设置的容量不得大于用户的剩余容量。

#### 查询Bucket配额容量

* **接口定义**

**bool CBucket::getCapacity(uint64\_t& megaBytes, uint64\_t& usedMegaBytes);**

* **功能描述**

获取bucket最大容量和已经使用的容量。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **megaBytes** | **[OUT] 获取到的bucket最大容量，单位MB。** |
| **usedMegaBytes** | **[OUT] 获取到的bucket已经使用的容量，单位MB。** |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_cap”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucketName );  // 检查bucket是否有效  if( bucket.isValid() ){  uint64\_t totalCap = 0;  uint64\_t usedCap = 0;  if( bucket.getCapacity (totalCap, usedCap) ){  std::cout << “total is “ << totalCap << “MB, used “ << usedCap << “MB” << std::endl;  }else{  std::cout << “get bucket capacity failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “get bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 未设置Bucket配额容量时调用该接口，则只有Bucket的已使用容量有效。

#### 设置Bucket生命周期

* **接口定义**

bool CBucket::setLifeCycle( int64\_t delay,

bool enableAction,

bool removeWhenBucketEmpty,

RecycleAction act=doDelete,

const char\* prefix=NULL );

* **功能描述**

设置bucket生命周期。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| delay | [IN ] bucket下文件的过期时间，单位为秒，最小值为3600秒 |
| enableAction | [IN ] 是否使能参数act所指定的操作 |
| removeWhenBucketEmpty | [IN ] 当bucket下无文件时，是否删除 |
| act | [IN ] bucket过期时执行的操作，暂仅支持删除操作  enum RecycleAction{  doDelete = 0 ///<删除  }; |
| prefix | [IN ] 保留字段 |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_lifecycle”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->createBucket( bucketName );  // 检查bucket是否有效  if( bucket.isValid() ){  // 设置bucket生命周期，bucket下的文件从创建30天后删除，且当Bucket下无文件时，删除Bucket  int64\_t delay = 30 \* 24 \* 3600;  bool enableAction = true;  bool removeWhenBucketEmpty = true;  Dahua::EFS::RecycleAction act = Dahua::EFS::doDelete;  const char\* prefix = NULL; // 目前未使用必须为NULL  if( bucket.setLifeCycle( delay, enable\_action, remove\_when\_bucket\_empty, act, prefix ) ){  std::cout << “set bucket lifecycle” << std::endl;  }else{  std::cout << “set bucket lifecycle failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  // 取消Bucket生命周期  enable\_action = false; // prefix必须为NULL，其他字段维持原来的值即可  if( bucket.setLifeCycle( delay, enable\_action, remove\_when\_bucket\_empty, act, prefix ) ){  std::cout << “cancle bucket lifecycle” << std::endl;  }else{  std::cout << “cancle bucket lifecycle failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “create bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 建议生命周期的时间不少于1天，生命周期过短可能导致文件正在写入时就被删除，这会导致正在写入的数据丢失。
  + 如果参数enableAction设置为false，不会执行删除文件的操作，生命周期不会起作用，并且设置的delay也不会生效。

#### 查询Bucket生命周期

* **接口定义**

bool CBucket::getLifeCycle( int64\_t& delay,

bool& enableAction,

bool& removeWhenBucketEmpty,

RecycleAction& act=doDelete,

const char\* prefix=NULL );

* **功能描述**

获取bucket生命周期配置。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| delay | [OUT ] bucket下文件的过期时间，单位为秒 |
| enableAction | [OUT ] 是否使能参数act所指定的操作 |
| removeWhenBucketEmpty | [OUT ] 当bucket为空时，是否删除 |
| act | [OUT ] bucket过期时执行的操作 |
| prefix | [IN ] 保留字段 |

* **接口返回值**

（1） true：设置成功。

（2） false：设置失败。可以调用getLastError()获取错误码。

* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_lifecycle”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucketName );  // 检查bucket是否有效  if( bucket.isValid() ){  // 查询bucket生命周期配置  int64\_t delay;  bool enableAction;  bool removeWhenBucketEmpty;  Dahua::EFS::RecycleAction act;  const char\* prefix = NULL; // 目前未使用必须设为NULL  if( bucket.getLifeCycle( delay, enable\_action, remove\_when\_bucket\_empty, act, prefix ) ){  std::cout << “get bucket lifecycle” << std::endl;  }else{  std::cout << “get bucket lifecycle failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “create bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

#### 获取Bucket下的文件个数

* **接口定义**

bool CBucket:: getFileNum( uint64\_t& bigFileNum, uint64\_t& smallFileNum )

* **功能描述**

获取bucket下的文件个数。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| bigFileNum | [OUT ] 大文件个数 |
| smallFileNum | [OUT ] 小文件个数 |

* **接口返回值**

（1） true：获取成功。

（2） false：获取失败。可以调用getLastError()获取错误码。

* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucket\_name );  // 检查bucket是否有效  if( bucket.isValid() ){  uint64\_t big\_file\_num = 0;  uint64\_t small\_file\_num = 0;  if( bucket.getFileNum( big\_file\_num, small\_file\_num ) ){  std::cout << “big file num:”<< big\_file\_num << “ small file num:” << small\_file\_num << std::endl;  }else{  std::cout << “get bucket file num failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “get bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 该接口在V1.072.0000003.7.R版本后提供
  + 因EFS延时删除文件的机制，该接口获取的文件数是有延迟的，即文件删除后，通过该接口获取的文件数不能及时反映。请不要在类似“当文件数达到一定值时，开始删除文件“这样的逻辑中使用该接口。
  + 该接口获取的小文件个数并不准确，实际返回的是Container的个数，请不要依赖该数值做逻辑处理。每个Container可能包含多个小文件

### 管理File

对同一个文件对象不支持多线程并发调用其接口。接口原型省略了命名空间Dahua::EFS。

#### 创建文件

* **接口定义**

**CFile CEFileSystem::create( const char\* ilename,**

uint16\_t n = 4,

uint16\_t m = 1,

**bool bigFile = true );**

* **功能描述**

**通过用户传入文件名来创建指定文件。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **fileName** | [IN] 文件名，必须是bucketname/filename的形式，必须符合命名规范 |
| n | [IN] 冗余规则N+M中的N，默认是4。 |
| m | [IN] 冗余规则N+M中的M，默认是1。 |
| bigFile | [IN] 是否大文件，true表示创建大文件，false表示创建小文件，默认值true。（建议：100M以下的建议创建小文件，100M以上的建议创建大文件） |

* **接口返回值**
  + 不管是创建成功还是失败, 接口都会返回一个CFile的对象。必须使用CFile:: isValid()检测文件是否有效，若返回CFile:: isValid()返回false，则可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| // 创建文件时，必须保证Bucket已存在  const char\* bucket\_name = “bucket1”;  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 以4+1模式创建大文件  Dahua::EFS::CFile file = efs->create( filename.c\_str(), 4, 1, true );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( file.isValid() ){  // 文件创建成功，该文件只能用于写  // 结束写后，必须关闭文件  file.close();  }else{  // 如果最后错误码是-80003，则说明文件已经存在  std::cout << “create file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + Bucket必须是已经存在的
  + 根据系统规模及需求选择合适的N+M，保证性能以及空间利用率。建议将N+M实现为可配置项。
  + 通过该接口获取到的CFile对象只可用于写入
  + 文件名不区分大小写
  + 小文件写入的数据不得超过100MB
  + 大文件写入的数据不得超过2TB
  + 文件命名规范：[File注意事项](#File注意事项)
* **创建模拟目录**

EFS中不存在目录概念，但可通过该接口创建一个模拟目录，本质上是创建了一个size为0的文件，该文件也可进行读写，但不推荐对该文件进行读写。创建模拟目录时，只需要filename以’/’结尾即可。

|  |
| --- |
| // 创建目录时，必须保证Bucket已存在  const char\* bucket\_name = “bucket1”;  // 创建时指定的目录名，必须是包含有bucket名字，注意要有斜杠’/’分割，且最后一个字符是’/’  std::string filename = std::string( bucket\_name ) + “/” + “helloworld/”;  // 以任意N+M模式创建，  Dahua::EFS::CFile file = efs->create( filename.c\_str(), 1, 1, true );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( file.isValid() ){  // 不要对该CFile对象进行操作，直接关闭即可  file.close();  }else{  std::cout << “create directory failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

#### 打开文件

* **接口定义**

CFile CEFileSystem::open( const char\* ilename, FileMode mode )

* **功能描述**

打开指定名称的文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN] 文件名，文件名组成为bucketName+’/’+fileName |
| mode | [IN] 打开模式。  可选值：  Dahua::EFS:: fileModeRead 读打开  Dahua::EFS::fileModeAppend 追加写打开（**暂时不支持**） |

* **接口返回值**
  + 不管是创建成功还是失败, 接口都会返回一个CFile的对象。必须使用CFile:: isValid()检测文件是否有效，若返回CFile:: isValid()返回false，则可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 对文件进行读取操作    // 关闭文件  (void)file.close();  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 该接口操作的文件必须是已经存在的
  + 该接口获取到的CFile对象只可用读取或追加写入，这取决于第二个参数指定的模式

#### 上传文件

* **接口定义**

int CFile::write( const char\* buf, uint32\_t len );

* **功能描述**

**写入指定长度的数据。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buf | [IN] 需要写入文件的数据，空间由调用者申请和释放。buf不能为NULL |
| len | [IN] 需要写入的数据字节数，len必须大于0 |

* **接口返回值**
  + > 0：成功写入的字节数
  + 0：暂时不可写，请在休眠一段时间后，再次调用本接口。
  + -1：写入出错。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| // 创建文件时，必须保证Bucket已存在  const char\* bucket\_name = “bucket1”; // bucket名必须符合命名规范  // 创建时指定的文件名，必须是包含有bucket名字  std::string filename = bucket\_name + “/” + “helloworld”;  // 以4+1模式创建大文件  Dahua::EFS::CFile file = efs->create( filename.c\_str(), 4, 1, true );  // 或者可通过CFile::open接口以追加写模式打开文件（文件必须已经存在），进行追加写入  // Dahua::EFS::CFile file = efs->open( filename.c\_str(),Dahua::EFS::fileModeAppend );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( file.isValid() ){  // 文件创建成功，该文件只能用于写  uint32\_t want\_write\_len = 300 << 20; // 期望写入300M  uint32\_t write\_once\_len = 32768\*4\*5; // 推荐每次写入的长度是32K\*N的倍数，此处N是4，倍数设置为5  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 = file.write( buf + write\_once\_offset, write\_once\_len – write\_once\_offset );  if( 0 < ret ){  write\_once\_offset += ret;  write\_len += 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 << “write failed, error:” << Dahua::EFS::getLastError() << std::endl;  break;  }else{  // 目前不存在其他返回值  assert( 0 );  }  }  // 结束写后，必须关闭文件  file.close();  delete[] buf;  buf = NULL;  std::cout << “write over, write length:” << write\_len << std::endl;  }else{  std::cout << “create file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 该接口为同步非阻塞接口，并非一定写入指定的len长度的数据，实际写入长度以返回值为准。
  + 大小文件只在创建时有区别，写入流程是一样的
  + 调用接口写入的数据会先保存在本地内存中，达到一个条带大小后，才会真正写入云存储。异常情况下可能丢失本地内存中的数据。
  + 调用接口时，每次写入的长度推荐为32K\*N的倍数
  + 可通过CFile::open()接口以追加写模式打开文件（文件必须已经存在），然后上传数据。使用追加写前，必须确认所使用的版本是否支持该功能（7.0版本之后支持）。
  + 请不要对一个文件多线程调用该接口，除非您能保证数据写入的顺序。

#### 下载文件

* **接口定义**

int CFile::read( char\* buf, uint32\_t len );

* **功能描述**

**从文件中读指定长度的数据。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buf | [IN] 从文件中读出数据放入buffer，buf的空间由外部申请和释放。 |
| len | [IN] 需要读到的字节数。 |

* **接口返回值**
  + > 0：实际读取的字节数。
  + 0：暂时不可读，建议用户在休眠一段时间后，重新调用该接口读取。
  + -1：读取错误。可以调用getLastError()获取错误码。
  + -2：读到文件末尾。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 文件有效，开始读取  uint32\_t want\_read\_len = 10 << 20; // 期望读取10M  uint32\_t read\_once\_len = 32768\*4\*5; // // 推荐每次读取的长度是32K\*N的倍数，此处N是4，倍数设置为5  char\* buf = new char[read\_once\_len]; // 读取数据的缓存  uint32\_t read\_len = 0;  // 读取指定长度  while( read\_len < want\_read\_len ){  int ret = file.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;  }  }  // 打开文件成功后，关闭文件  file.close();  delete[] buf;  buf = NULL;  std::cout << “read over, read length:” << read\_len << std::endl;  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 该接口为同步非阻塞接口，并非一定读取指定的长度，实际读取长度以返回值为准。
  + 如果文件是边写边读，因写入机制，可能使得最后写入不满一个条带的数据可能无法立即读取，需要关闭文件后才能读取到
  + 如果文件是边写边读，立即写入立即读取有可能会读取失败，需要确保写入一个条带的数据后（推荐至少写入1M数据）再开始读取数据
  + 可结合CFile::seek()接口实现随机读取
  + 请不要多线程调用该接口，除非您能保证读取数据的顺序。

#### 关闭文件

* **接口定义**

bool CFile::close();

* **功能描述**

**关闭文件。**

* **接口返回值**
  + true：关闭成功。
  + false：关闭过程中存在错误
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 文件创建或打开成功后，必须关闭文件，一般无需关心关闭的结果  (void)file.close();  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 一旦调用该接口关闭文件，不得调用该文件对象的任意接口

#### 删除文件

* **接口定义**

**bool CEFileSystem::remove( const char\* ilename )；**

* **功能描述**

**删除指定名称的文件。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **fileName** | **[IN] 需要删除的文件名。** |

* **接口返回值**
  + true：删除成功。
  + false：删除失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;    if( efs->remove( filename.c\_str() ) ){  std::cout << “remove file” << std::endl;  }else{  std::cout << “remove file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 删除文件前必须确保文件已经关闭

#### 锁定文件

* **接口定义**

bool CEFileSystem::lockFile( const char\* ilename);

* **功能描述**

锁定指定文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN ] 锁定的文件名 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 锁定文件后，该文件无法被删除，但可以被重命名  if( efs->lockFile( filename.c\_str() ) ){  std::cout << “lock file” << std::endl;  }else{  std::cout << “unlock file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 锁定的文件不可被删除，但可以被重命名
  + 该功能在6.0版本之后才提供

#### 解锁文件

* **接口定义**

bool CEFileSystem::unlockFile( const char\* ilename);

* **功能描述**

解锁指定文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN ] 待解锁的文件名 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 解锁文件后，文件可被删除  if( efs->unlockFile( filename.c\_str() ) ){  std::cout << “unlock file” << std::endl;  }else{  std::cout << “unlock file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 解锁后的文件可被删除
  + 该功能在6.0版本之后才提供

#### 获取文件锁定状态

* **接口定义**

int32\_t CEFileSystem::getFileLockStat(const char\* ilename);

* **功能描述**

获取文件锁定状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN ] 文件名 |

* **接口返回值**
  + 1：文件已锁定。
  + 0：文件未锁定。
  + -1：获取状态失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 获取文件锁定状态  int32\_t state = efs->getFileLockStat( filename.c\_str() );  if( 1 == state ){  // 文件已锁定  std::cout << “file locked” << std::endl;  }else if( 0 == state ){  // 文件未锁定  std::cout << “file not lock” << std::endl;  }else if( -1 == state ){  // 获取失败  std::cout << “get lock state failed” << std::endl;  }else{  // 无其他错误码  assert( 0 );  } |

* **提示**
  + 该功能在6.0版本之后才提供

#### 重命名文件

* **接口定义**

bool CEFileSystem::rename(const char\* oldFileName, const char\* newFileName);

* **功能描述**

**重命名文件名。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| oldFileName | **[IN] 旧文件名字，**该文件名的组成是bucket名+’/’+真实文件名 |
| newFileName | **[IN] 新文件名字，**该文件名的组成是bucket名+’/’+真实文件名 |

* **接口返回值**

1. true：重命名成功。
2. false：重名失败，可以调用getLastError()获取错误码。

* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string old\_filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 新文件名的Bucket必须与老文件名的Bucket一致  std::string new\_filename = std::string( bucket\_name ) + “/” + “byeworld”;    if( efs->rename( old\_filename.c\_str(), new\_filename.c\_str() ) ){  std::cout << “rename file” << std::endl;  }else{  std::cout << “rename file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 新文件名中的bucket必须与老文件名中的bucket一致，即不允许跨bucket重命名
  + 重命名时必须保证文件已经关闭

#### 监听文件状态

* **接口定义**

bool CFile::setListener(IFileListener\* listener);

* **功能描述**

监听文件状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| listener | [IN ] 文件监听者，为NULL时表示删除监听。  IFileListener内部有一个虚函数  virtual void onFileStateChanged(CFile\* file, FileState state)，其中file为文件对象，state表示该文件对象的状态。  enum FileState {  fileStateInit, //初始状态  fileStateNormal, //文件正常  fileStateWarning, //文件有风险  fileStateException //文件异常  }; |

* **接口返回值**
  + true：设置成功
  + false：设置失败
* **示例**

|  |
| --- |
| // 继承自Dahua::EFS::IFileListener，实现文件状态处理  class FileListener : public Dahua::EFS::IFileListener  {  public:  virtual void onFileStateChanged( Dahua::EFS::CFile\* file, Dahua::EFS::FileState state )  {  // 可根据文件状态进行处理，常规只处理Dahua::EFS::fileStateWarning状态  std::cout << “file state is “ << state << std::endl;  if( Dahua::EFS::fileStateWarning == state ){  // 该状态说明N+M写入时，小于等于M个Block写入异常，  // 为保证数据冗余度，可选择设置关闭文件的标识，  // 写入线程检查该标识，关闭文件并重新创建文件写入  // 无特殊需求则不必使用该功能  }  return;  }  };  // 创建文件时，必须保证Bucket已存在  const char\* bucket\_name = “bucket1”;  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 以4+1模式创建大文件  Dahua::EFS::CFile file = efs->create( filename.c\_str(), 4, 1, true );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( file.isValid() ){  // 监听对象  FileListener\* listener = new FileListener;  // 监听文件状态  if( file.setListener( listener ) ){  std::cout << “set file listener to NULL” << std::endl;  }else{  // 一般不会失败  }  // 对文件进行操作,如果在此过程中文件状态出现变化，会调用FileListener::onFileStateChanged()  // 不再监听文件状态  if( file.setListener( NULL ) ){  std::cout << “set file listener to NULL” << std::endl;  }else{  // 一般不会失败  }  //销毁监听对象  delete listener;  listener = NULL;  file.close();  }else{  std::cout << “create file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 不能直接在onFileStateChanged()中关闭文件，否则可能导致异常
  + 不论读写都可监听文件状态变化
  + 7.2版本之前对一个File重复设置监听时，内部使用最后一次设置的监听。7.2版本开始支持设置多个监听对象，但设置为NULL时，会取消全部监听。
  + 除非确实需要对Dahua::EFS::fileStateWarning状态进行处理，否则一般情况下，只需根据CFile对象的接口返回值进行业务处理。
  + 由于是异步回调，请注意锁的使用，避免死锁

#### 获取文件属性

* **接口定义**

**bool CFile::stat( FileStat& stat );**

* **功能描述**

**获取指定文件的文件属性。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **stat** | **[OUT] 文件属性**  **typedef struct FileStat**  **{**  **uint8\_t mode; ///<读写模式，值为Dahua::EFS::FileMode**  **uint8\_t isAppend; ///<1-正在写入，0-未在写入**  **uint8\_t dataNum; ///<数据分片数，即N+M中N的值**  **uint8\_t parityNum; ///<冗余分片数，即N+M中M的值**  **uint32\_t cTime; ///<创建时间**  **uint32\_t mTime; ///<最后修改时间**  **uint64\_t totalSize; ///<文件大小**  **char reserved2[108];**  **}FileStat;** |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 获取文件状态  Dahua::EFS::FileStat s;  if( file.stat( s ) ){  std::cout << “get file stat:” << std::endl;  std::cout << “mode:” << s.mode << std::endl;  std::cout << “writing ? “ << ( 0 == s.isAppend ? “false” : “true” ) << std::endl;  std::cout << “N+M is “ << s.dataNum << “+” << s.parityNum << std::endl;  std::cout << “ctime: “ << s.cTime << std::endl;  std::cout << “mtime: “ << s.mTime << std::endl;  std::cout << “size: “ << s.totalSize << std::endl;  }else{  std::cout << “get file stat failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

#### 设置文件选项

* **接口定义**

bool CFile::setOption(EFSOption key, const void\* value, uint32\_t len);

* **功能描述**

**设置CFile高级配置信息。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| key | [IN ]系统配置选项。  enum EFSOption  {  efsRwMode //读写模式(0异步1同步，默认为0)  }; |
| value | [IN ]对应配置项的值。 |
| len | [IN]对应配置项value缓冲区的长度。单位byte。 |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 设置为同步阻塞式读取（默认为非阻塞读取）  int option = 1;  if( file.setOption( Dahua::EFS::efsRwMode, &option, sizeof( option ) ) ){  std::cout << “set file option” << std::endl;  }else{  std::cout << “set file option failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* 提示
  + 一般情况下无需设置该属性

#### 获取文件选项

* **接口定义**

bool CFile::getOption(EFSOption key, void\* value, uint32\_t len);

* **功能描述**

获取CFile高级配置信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| key | [IN ]系统配置选项。  enum EFSOption  {  efsRwMode //读写模式(0异步1同步)  }; |
| value | [OUT ]获取到对应配置项的值。 |
| len | [IN]获取对应配置项value缓冲区的长度。 |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  option = 0;  if( file.getOption( Dahua::EFS::efsRwMode, &option, sizeof( option ) ) ){  // 获取到的选项值必然为之前设置的值  std::cout << “get file option” << std::endl;  }else{  std::cout << “get file option failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

#### 修改文件偏移

* **接口定义**

**bool CFile::seek( int64\_t offset, int whence );**

* **功能描述**

**根据指定基准位置定位到指定偏移。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **offset** | **[IN] 相对于whence位置后在文件中的偏移** |
| **whence** | **[IN] 偏移的基准位置**  **enum FileLocation**  **{**  **efsBegin, //文件头位置**  **efsCurrent, //当前文件位置**  **efsEnd //文件尾位置**  **};** |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 修改偏移  if( file.seek( 100, Dahua::EFS::efsBegin ) ){  std::cout << “seek to 100B” << std::endl;  }else{  std::cout << “file seek failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  file.close();  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 结合CFile::read()可实现随机读取
  + 创建写**和追加写模式的File不支持调用该接口，即不支持随机写入**

#### 获取文件偏移

* **接口定义**

int64\_t CFile::tell();

* **功能描述**

**获取文件当前位置。**

* **接口返回值**
  + >= 0：当前读到的位置相对于文件头的偏移。
  + < 0：获取失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket1”;  // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  std::string filename = std::string( bucket\_name ) + “/” + “helloworld”;  // 读打开文件  Dahua::EFS::CFile file = efs->open( filename.c\_str(), Dahua::EFS::fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( file.isValid() ){  // 获取文件当前偏移  int64\_t offset = file.tell();  if( 0 <= offset ){  std::cout << “current offset is “ << offset << std::endl;  }else{  std::cout << “get file current offset failed, error:” << Dahua::EFS::getLastError() << std::endl;  }  file.close();  }else{  std::cout << “open file failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

#### 分页查询Bucket下的文件名

* **接口定义**

**bool CBucket::****list( const char\*** **begin,**

**const char\* end,**

**uint32\_t maxNumber,**

**CNameList& list );**

* **功能描述**

列出bucket内文件名落在(begin,end)范围的文件名列表。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **begin** | [IN] 起始文件名。无需包含Bucket名。如果begin为NULL，则从Bucket下具备最小值的文件开始查询，且包含该最小值的文件。 |
| **end** | [IN] 结束文件名。无需包含Bucket名。如果end为NULL，则查询到该Bucket下具有最大值的文件。 |
| **maxNumber** | [IN] 查询的最大文件个数。取值范围[1, 2048] |
| **list** | [OUT] 存放获取到的文件名。 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucket\_name );  if( bucket.isValid() ){  std::cout << “get bucket “ << bucket\_name << std::endl;  // 假设该Bucket下存在文件 a,b,c,d  std::string begin\_name = “b”;  std::string end\_name = “d”;  uint32\_t max\_num = 1024;  while( 1 ){  Dahua::EFS::CNameList namelist;  if( bucket.list( begin\_name.empty() ? NULL : begin\_name.c\_str(), end\_name.empty() ? NULL : end\_name.c\_str(), max\_num, namelist ) ){  // 这里将列出文件c，d，查询到的文件名不包含Bucket名  for( int I = 0; I < namelist.size(); ++I ){  std::cout << namelist.getName( I ) << std::endl;  }  if( max\_num > namelist.size() ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else{  // 可能还有文件，修改begin\_name继续查询  begin\_name = namelist.getName( namelist.size() – 1 );  }  }else{  std::cout << “list file failed, error:” << Dahua::EFS::getLastError() << std::endl;  break;  }  }  }else{  std::cout << “get bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果。
  + 虽然示例中begin和end不会为空，但可参考示例中的实现，如果begin和end为空时，需要传递NULL

#### 分页查询Bucket下的文件信息

* **接口定义**

bool CBucket::list( const char\* begin,

const char\* end,

uint32\_t maxNumber,

CFileList& list );

* **功能描述**

列出bucket内文件名落在(begin,end)范围的文件信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **begin** | [IN] 起始文件名。无需包含Bucket名。如果begin为NULL，则从Bucket下具备最小值的文件开始查询，且包含该最小值的文件。 |
| **end** | [IN] 结束文件名。无需包含Bucket名。如果end为NULL，则查询到该Bucket下具有最大值的文件。 |
| **maxNumber** | [IN] 查询的最大文件个数。取值范围[1, 2048] |
| **list** | [OUT] 存放获取到的文件信息。 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用getLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucket\_name );  if( bucket.isValid() ){  std::cout << “get bucket “ << bucket\_name << std::endl;  // 假设该Bucket下存在文件 a,b,c,d  std::string begin\_name = “b”;  std::string end\_name = “d”;  uint32\_t max\_num = 1024;  while( 1 ){  Dahua::EFS::CFileList filelist;  if( bucket.list( begin\_name.empty() ? NULL : begin\_name.c\_str(), end\_name.empty() ? NULL : end\_name.c\_str(), max\_num, filelist ) ){  // 这里将列出文件c，d，查询到的文件名不包含Bucket名  for( int I = 0; I < filelist.size(); ++I ){  std::cout << “filename: “ << filelist.getName( I ) << std::endl;  std::cout << “size: “ << filelist.getFileSize( I ) << std::endl;  std::cout << “create time: “ << filelist.getCTime( I ) << std::endl;  std::cout << “modify time: “ << filelist.getMTime( I ) << std::endl;  std::cout << std::endl;  }  if( max\_num > filelist.size() ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else{  // 可能还有文件，修改begin\_name继续查询  begin\_name = filelist.getName( filelist.size() – 1 );  }  }else{  std::cout << “list file failed, error:” << Dahua::EFS::getLastError() << std::endl;  break;  }  }  }else{  std::cout << “get bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果。
  + 虽然示例中begin和end不会为空，但可参考示例中的实现，如果begin和end为空时，需要传递NULL

#### 分页查询指定目录下的文件信息

* **接口定义**

bool CBucket::list( const char\* dir,

const char\* delimiter,

const char\* begin,

uint32\_t maxNumber,

CFileList& list);

* **功能描述**

查询bucket内dir目录下的文件信息

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **dir** | [IN] 需要查找的文件夹路径，最后字符必须为’/’。无需包含Bucket名 |
| **delimiter** | [IN] 文件夹路径分隔符,目前仅支持’/’ |
| **begin** | [IN] 起始文件名。无需包含目录名。如果为NULL，则查询dir下所有文件信息 |
| **maxNumber** | [IN] 查找的最大文件数量。取值范围[1, 2048] |
| **list** | [OUT] 存放获取到的文件信息 |

* **接口返回值**
  + true：查询成功。
  + false：查询失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Dahua::EFS::CBucket bucket = efs->getBucket( bucket\_name );  if( bucket.isValid() ){  std::cout << “get bucket “ << bucket\_name << std::endl;  /\*\*  \* 假设该Bucket下存在文件，a/, a/b, a/c/，a/c/d, a/e。可以认为其存在如下目录树  \* a/  \* |--  \* |--b  \* |--c/  \* | |--d  \* |--e  \*/  std::string dir = “a/”; // 必须以’/’结尾  std::string begin\_name = “b”;  uint32\_t max\_num = 1024;  while( 1 ){  // 列出a/目录下的文件，最后一个字符必须是’/’  // 分隔符只支持’/’  // begin是”b”，因此查询的结果是c/,e  Dahua::EFS::CFileList filelist;  if( bucket.list( dir.c\_str(), “/”, begin\_name.empty() ? NULL : begin\_name.c\_str(), max\_num, filelist ) ){  // 这里将列出文件c/,e。查询到的文件名不包含目录名  // 不会递归查询子目录  for( int I = 0; I < filelist.size(); ++I ){  std::cout << “filename: “ << filelist.getName( I ) << std::endl;  std::cout << “size: “ << filelist.getFileSize( I ) << std::endl;  std::cout << “create time: “ << filelist.getCTime( I ) << std::endl;  std::cout << “modify time: “ << filelist.getMTime( I ) << std::endl;  std::cout << std::endl;  }  if( max\_num > filelist.size() ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else{  // 可能还有文件，修改begin\_name继续查询  begin\_name = filelist.getName( filelist.size() – 1 );  }  }else{  std::cout << “lsit file failed, error:” << Dahua::EFS::getLastError() << std::endl;  break;  }  }  }else{  std::cout << “get bucket failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果。
  + 虽然示例中begin不会为空，但可参考示例中的实现，如果begin为空时，需要传递NULL

### 获取系统信息

#### 获取系统资源信息

* **接口定义**

bool CEFileSystem::getSystemInfo( SystemInfo& info );

* **功能描述**

获取当前EFileSystem的信息，包括集群节点数，客户端运行个数等。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| Info | [OUT ]系统信息。  typedef struct SystemInfo  {  uint32\_t totalNodes; ///< 系统中的节点总数  uint32\_t totalClient;///< 系统中正在运行的总客户端数  uint32\_t curWriteFiles;///< 系统中正在写的文件数  uint64\_t totalFiles; ///< 用户的文件总数  uint64\_t totalSpace;///< 用户的总存储空间，单位MB  uint64\_t spaceLeft; ///< 用户真正可用的剩余存储空间，单位MB  char uuid[32]; ///< 系统uuid,32个字符  uint64\_t spaceQuotaLeft; ///< 用户剩余可分配空间，单位MB  uint64\_t spaceUsed; ///<用户已使用容量, 单位MB  char reserved[44]; ///<保留  }SystemInfo; |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| Dahua::EFS::SystemInfo info;  bool ret = efs->getSystemInfo( info );  if( ret ){  // 以下为所有用户的统计信息  std::cout << “system uuid: “ << info.uuid << std::endl; // 系统唯一标识  std::cout << “total nodes: “ << info.totalNodes << std::endl; // 系统中的总节点数  std::cout << “total clients: “ << info.totalClient << std::endl; // 系统当前正运行的客户端  std::cout << “current writing files: “ << info.curWriteFiles << std::endl; // 系统当前正在写的文件总数  // 以下为当前用户的信息，请注意在系统实际容量不足时，spaceLeft+spaceUsed不一定等于totalSpace  std::cout << “total files: “ << info.totalFiles << std::endl; // 当前用户的文件总数  std::cout << “total space: “ << info.totalSpace << std::endl; // 当前用户的总空间  std::cout << “space left: “ << info.spaceLeft << std::endl; // 用户真正可用的剩余空间，可能是系统实际剩余空间  std::cout << “space quota left: “ << info.spaceQuotaLeft << std::endl; // 剩余的可分配空间，即可以给Bucket的设定的最大配额  std::cout << “used space: “ << info.spaceUsed << std::endl; // 用户已经使用了的空间  }else{  std::cout << “get system info failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 其中totalFiles并非大文件个数和小文件个数的总和，而是大文件个数和container个数的总和，而一个container可能包含多个小文件。请不要在类似“当文件数达到一定值时，开始删除文件“这样的逻辑中依赖该数值。

#### 获取系统支持的N+M

* **接口定义**

bool CEFileSystem::getRedundanceCaps(CRedundanceCaps& caps);

* **功能描述**

获取系统支持的N+M

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| caps | [OUT ]冗余信息。  其中CRedundanceCaps有三个方法:  int CRedundanceCaps:: size() //总共支持的组合类型  uint16\_t getDataNum(int index) //第index个组合中的N  uint16\_t getParityNum(int index) //第index个组合中的M |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| Dahua::EFS::CRedundanceCaps caps;  bool ret = efs->getRedundanceCaps( caps );  if( ret ){  std::cout << “support:” << std::endl;  for( int32\_t I = 0; I < caps.size(); ++I ){  std::cout << caps.getDataNum( I ) << “+” << caps.getParityNum( I ) << std::endl;  }  }else{  std::cout << “get n+m failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 可根据该接口获取N+M模式创建文件。

## 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。

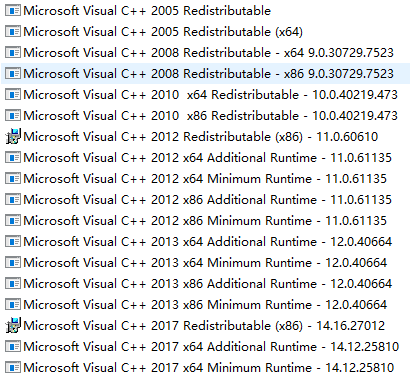
提供的Demo是Release 32位的，请参照该Demo，自行搭建64位工程。

## 常见问题

### Windows平台SDK初始化失败

SDK日志中会打印EFSClientCore.dll加载失败，errno是193（不是有效的win32程序）或者126（找不到指定的模块）

参考安装章节，确保所有库是放在库搜索目录下的。如果确定库已经正确放置，那么可能是缺少vc++的运行库。可搜索visual c++ AIO Installer安装所有版本的Visual C++运行库，包括



# 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\_EFSSDK\_Eng\_[Platform]\_[bits]bit\_PS\_[Version].tar.gz

实例如下：

* Linux
  + 32位：DH\_EFSSDK\_Eng\_32bit\_PS\_V1.070.0001.5.R.180429.tar.gz
  + 64位：DH\_EFSSDK\_Eng\_64bit\_PS\_V1.070.0001.5.R.180429.tar.gz
* Windows
  + 32位：DH\_EFSSDK\_Eng\_Windows\_32bit\_PS\_V1.070.0001.5.R.180429.tar.gz
  + 64位：DH\_EFSSDK\_Eng\_Windows\_64bit\_PS\_V1.070.0001.5.R.180429.tar.gz

上述实例中的[Version]的070，代表为7.0版本，可以此类推。

### 编译

1. 从安装包中获得EFSClient.jar和jna-4.1.0.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:EFSClient.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

### 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，不能包含中文路径。

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
  + ldconfig配置文件中指定的目录
* windows平台
  + 应用程序所在目录
  + 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
    - libEFSAdapter.so
    - libEFSClient.so
    - libEFSClientCore.so（5.0版本之后才有）
  + 64位
    - libEFSClient64.so
    - libEFSClient64.so
    - libEFSClientCore64.so（5.0版本之后才有）
* Windows
  + 32位
    - EFSAdapter.dll
    - EFSClient.dll
    - EFSClientCore.dll（5.0版本之后才有）
    - EFSHelper.dll（7.2版本后，必须安装在C:/Windows/System32目录下）
  + 64位
    - EFSAdapter64.dll
    - EFSClient64.dll
    - EFSClientCore64.dll（5.0版本之后才有）
    - EFSHelper64.dll（7.2版本后，必须安装在C:/Windows/System32目录下）

### 升级

取最新的安装包，替换对应的SDK库和jar包即可。如果是从4.0+升级到5.0之后的版本，需要特别注意，不要遗漏了EFSClientCore库。Linux下需要删除/tmp/sdk目录下的文件，Windows下需要删除程序所在目录的upgrade目录

### 兼容性

目前无接口兼容性问题。

4.0+SDK与7.0版本云存储配套存在问题，需使用更新版本的SDK（6.0版本之后含6.0）

6.0版本之后的SDK与4.0+版本之后的云存储均可配套使用。

## 使用介绍

### 初始化

以下接口均不允许多线程调用，且在进程生命周期内，一般只需调用一次。

#### 设置加载动态库路径

* **接口原型**

void Utils.setEFSLibPath(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EFS SDK库路径为D盘  Utils.setEFSLibPath(“d:\\”);  System.out.println( “EFS lib path is “+ Utils.getEFSLibPath() ); |

* **提示**
  + 使用该接口需要import com.dahuatech.efs.utils;
  + 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

#### 连接EFS服务

* **接口原型**

ilenam EFileSystem.init(EFSConfig cfg);

* **功能描述**

初始化EFileSystem对象，登陆到EFS服务端。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| cfg | [IN] 登陆配置信息，需要设置的字段如下  public String address; // EFS服务地址  public int port; // EFS服务端口  public String ilename; // 存储用户名  public String password; // 存储用户密码 |

* **接口返回值**
  + true：初始化成功。
  + false：初始化失败，错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| // 创建efs对象  EFileSystem efs = new EFileSystem();  EFSConfig.ByReference cfg = new EFSConfig.ByReference();  cfg.address =”10.35.47.246”; // EFS服务地址  cfg.port = 38100; // EFS服务端口，固定不变  cfg.userName = “root”; // 存储用户名  cfg.password = “123456”; // 存储用户密码  // 初始化efs，重复初始化会抛出RuntimeException( “init already!” )  if( efs.init( cfg ) ){  System.out.println( “efs initialized” );  // 初始化成功后，必须关闭，且在关闭前，必须保证所有打开的文件已经关闭  efs.close();  }else{  System.out.println( “initialize failed, error: “ + EFileSystem.getLastError() );  } |

* **提示**
  + 您的工程中可以有多个EFileSystem实例，也可以只有一个EFileSystem实例。如无特殊需求，推荐只创建一个EFileSystem实例。
  + 如发生网络异常，内部会自动重连。如设置了监听，那么将会回调通知使用者。

#### 高级选项配置

* **接口定义**

ilenam EFileSystem.setOptions(EFSOption option, Object obj);

* **功能描述**

设置系统高级配置信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| key | [IN ]系统配置选项。  public enum EFSOption  {  efsConcurrent, ///<并发数  efsBufferSize, ///<缓存大小，单位字节  efsTimeOut, ///<消息超时时间，单位毫秒  efsLogOutput, ///<日志文件输出路径  efsLogLevel, ///<日志级别  efsRwMode, ///<读写模式(0异步1同步)  efsSmallFileBufferSize, ///<小文件缓存大小(7.2版本后弃用)  efsBufferMode ///<缓存模式(1单缓存2双缓存)(7.2版本后弃用)  } |
| value | [IN ]配置项的值。 |
| len | [IN]配置项value的大小。单位B。 |

* **接口返回值**
  + true：成功。
  + false：失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| // 设置日志目录(目录必须存在)。推荐设置，可以有效定位问题。（可实现为程序配置项）  if( efs.setOptions( EFSOption.efsLogOutput, “./log/” ) ){  System.out.println( “set log path” );  }else{  System.out.println( “set log path failed, error: “ + EFileSystem.getLastError() );  }  // 设置日志级别，对接过程中推荐设置为DEBUGF,稳定后可设置为INFOF。（可实现为程序配置项）  if( efs.setOptions( EFSOption.efsLogLevel, EFSLogLevel.DEBUGF ) ){  System.out.println( “set log level” );  }else{  System.out.println( “set log level failed, error: “ + EFileSystem.getLastError() );  }  // 其他高级选项类似，一般无需设置 |

* **提示**
  + 此接口只能在init成功后调用，否则会直接返回false。
  + 推荐设置日志目录和级别，方便定位问题。日志文件占用最大空间为30\*50MB，会自行覆盖。对接测试时建议设置为DEBUG，发布时可配置为INFO。因有改动需求，建议实现为配置项。
  + 日志目录建议设为绝对路径，方便查找，且调用setOptions()时，日志目录参数不能为空。
  + efsConcurrent默认为8，为测试时最佳性能值，一般无需修改
  + efsBufferSize默认8个条带的大小，大文件的默认缓存是8\*32K\*(N+M)，小文件的默认条带是8\*4K\*(N+M)
  + 其他配置一般无需设置，可维持默认。
  + Windows下设置日志目录时请使用Linux格式，即选择以”/”作为目录分割符，比如d:/a/b/c

#### 监听EFS连接状态

* **接口定义**

ilenam EFileSystem.setListener(IEFSListener listener);

* **功能描述**

监听EFS连接状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| listener | [IN ] 文件系统监听者，为NULL时表示删除监听。可监听到的状态如下  public enum EFSState  {  efsStateInit, ///初始状态  efsStateNormal, ///<efs客户端正常  efsStateException, ///<efs客户端异常，会进行自动恢复  efsStateError ///<efs客户端异常，可能是用户不存在或者密码错误，不可自动恢复，需要用户介入.  } |

* **接口返回值**
  + true：成功。
  + false：失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| // EFS状态处理类  public static class EFSListener implements EFileSystem.IEFSListener  {  @Override  public void onStateChanged(EFileSystem efs, EFSState state )  {  // 可根据EFS状态进行处理，常规用法是：当EFS连接状态异常时，切换为其他存储模式，保证数据安全  System.out.println( “EFS state is “ + state );  }  }    EFSListener listener = new EFSListener();  // 监听EFS连接状态  if( efs.setListener( listener ) ){  System.out.println( “set efs listener” );  }else{  // 一般不会失败  System.out.println( “set efs listener failed, error: “ + EFileSystem.getLastError() );  }  // 在此过程中，如果EFS连接状态发生变化，则会调用onStateChanged()  // 不再监听EFS连接状态，NOTE:这会取消掉该efs对象上所有的listener  if( efs.setListener( null ) ){  System.out.println( “set efs listener to null” );  }else{  // 一般不会失败  System.out.println( “set efs listener failed, error: “ + EFileSystem.getLastError() );  } |

* **提示**
  + 7.2版本后，允许在一个EFileSystem对象上设置多个监听，但是一旦设为NULL将清除所有监听。
  + 一般场景下，无需使用该功能。
  + 由于是异步回调，请注意锁的使用，避免加锁调用EFileSystem类的接口，避免死锁

#### 设置用户空间紧急覆盖策略

* **接口定义**

ilenam EFileSystem .setRecyclePolicy( ilenam enableRecycle ,

EFSRecyclePolicy policy );

* **功能描述**

设置用户空间回收策略。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| enableRecycle | [IN ] 是否使能用户空间回收 |
| policy | [IN ] 回收策略  public enum EFSRecyclePolicy{  reduceByTime, ///<等时间缩减  reduceByRatio ///<等比例缩减  } |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| ilenam enable\_recycle = true; // 使能紧急覆盖  EFSRecyclePolicy policy = EFSRecyclePolicy.reduceByTime; // 等时间删除  // 设置紧急覆盖后，只对该用户下的设置了生命周期的Bucket起作用  ilenam ret = efs.setRecyclePolicy( enable\_recycle, policy );  if( ret ){  System.out.println( “set recycle policy” );  }else{  System.out.println( “set recycle policy failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 6.0版本后支持该功能。
  + 紧急覆盖仅对使能生命周期的Bucket有效。默认情况下，用户的使用容量达到95%时，开始删除文件，释放空间。
  + 等时间缩减指该用户下所有Bucket删除<Bucket生命周期+固定时间>时间前创建的文件
  + 等比例缩减指该用户下所有Bucket删除<Bucket生命周期\*固定比例>时间前创建的文件

#### 获取用户空间紧急覆盖策略

* **接口定义**

ilenam EFileSystem.getRecyclePolicy( bool& enableRecycle, RecyclePolicy& policy);

* **功能描述**

获取用户空间回收策略配置。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| enableRecycle | [OUT ] 是否使能用户空间回收 |
| policy | [OUT ] 回收策略  public enum EFSRecyclePolicy{  reduceByTime, ///<等时间缩减  reduceByRatio ///<等比例缩减  } |

* **接口返回值**
  + true：获取成功。
  + false：获取失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| // 获取紧急覆盖策略  IntByReference get\_enable = new IntByReference();  IntByReference get\_policy = new IntByReference();  ret = efs.getRecyclePolicy( get\_enable, get\_policy );  if( ret ){  System.out.println( “enable: “ + get\_enable.getValue() + “, policy: “ + get\_policy.getValue() );  }else{  System.out.println( “get recycle policy failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 6.0版本后支持该功能。

#### 退出EFS服务

* **接口定义**

ilenam EFileSystem.close();

* **功能描述**

关闭文件系统。

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

|  |
| --- |
| if( efs.init( cfg ) ){  System.out.println( “efs initialized” );  // 初始化成功后，必须关闭，且在关闭前，必须保证所有打开的文件已经关闭  // 一般无需关注其结果  efs.close();  }else{  System.out.println( “initialize failed, error: “ + EFileSystem.getLastError() );  } |

* **提示**
  + 调用EFileSystem.close()前，请务必保证通过该对象创建或打开的文件已经关闭。
  + 一般无需判断EFileSystem.close()的返回值。
  + EFileSystem.close()仅在确定不再使用CEFileSystem实例时调用，比如进程退出前。
  + 一旦调用EFileSystem.close()后，该EFileSystem实例不可再使用

### 管理Bucket

以下接口允许多线程调用

#### 创建Bucket

* **接口定义**

**Bucket EFileSystem.createBucket(String name);**

* **功能描述**

**创建Bucket**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + **非null：成功**
  + **null：失败。**可以调用EFileSystem.getLastError()获取错误码。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Bucket bucket = efs.createBucket( bucket\_name );  // bucket为null时无效  if( null != bucket ){  System.out.println( “create bucket “ + bucket\_name );  // 操作Bucket对象  // ...    // 创建或获取成功后必须关闭  bucket.close();  }else if( -80005 == EFileSystem.getLastError() ){  // 错误码是-80005（Bucket已经存在），则可以通过获取Bucket接口获得Bucket对象  }else{  System.out.println( “create bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 创建的Bucket的访问权限继承自存储用户的访问权限，可通过接口重新设置
  + Bucket名字必须全局唯一，您需要保证Bucket名不会与其他用户冲突。
  + 当创建失败且错误码是-80005（Bucket已存在）时，可直接获取Bucket。
  + 因为单个Bucket下的文件过多时，会影响读取性能，所以推荐每天创建不同的Bucket进行存储。
  + Bucket命名规范可以参考：[bucket注意事项](#bucket注意事项)

#### 删除Bucket

* **接口定义**

**ilenam EFileSystem.removeBucket(String name );**

* **功能描述**

**删除指定名称的bucket。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + true：删除成功。
  + false：删除失败，错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  if( efs.removeBucket( bucket\_name ) ){  System.out.println( “remove bucket “ + bucket\_name );  }else{  System.out.println( “remove bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 删除Bucket前，必须先删除bucket下的所有文件，不然会返回失败，错误码为-80006。

#### 获取Bucket

* **接口定义**

**Bucket EFileSystem.getBucket( String name );**

* **功能描述**

**根据bucket名称获取bucket对象。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + **非null：成功**
  + **null：失败。**可以调用EFileSystem.getLastError()获取错误码。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket1”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  if( null != bucket ){  System.out.println( “get bucket “ + bucket\_name );  // 操作Bucket对象  // ...    // 创建或获取成功后必须关闭  bucket.close();  }else if( -80004 == EFileSystem.getLastError() ){  // 错误码是-80004（Bucketn不存在），则需要通过创建Bucket接口获得Bucket对象  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 当调用该接口失败且错误码是-80004（Bucket不存在）时，需要先创建Bucket

#### 关闭Bucket

* **接口定义**

**ilenam Bucket.close();**

* **功能描述**

关闭Bucket。

* **接口参数**

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

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

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Bucket bucket = efs.createBucket( bucket\_name );  // bucket为null时无效  if( null != bucket ){  System.out.println( “create bucket “ + bucket\_name );  // 操作Bucket对象  // ...    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “create bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + **创建或获取Bucket句柄成功后，必须关闭，一般无需关系关闭的结果**

#### 列出Bucket

* **接口定义**

**List<String> EFileSystem.getAllBucket();**

* **功能描述**

列出所有Bucket。

* **接口参数**

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

* **接口返回值**
  + **包含获取到的Bucket名**
* **示例**

|  |
| --- |
| List<String> buckets = efs.getAllBucket();  System.out.println( “bucket num: “ + buckets.size() );  for( String b : buckets ) {  System.out.println( b );  } |

* **提示**
  + **在用户list过程中，如存在删除bucket的操作，list出的Bucket可能存在遗漏。**

#### 设置Bucket访问权限

* **接口定义**

**ilenam Bucket.** **setBucketPrivilege(EFSAccessPrivilege privilege)**

* **功能描述**

设置bucket权限。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **privilege** | **[IN] 设置的bucket权限。**  public enum EFSAccessPrivilege  {  privilegePrivate,///<私有，其他用户不可见  privilegeRead,///<只读，其他用户无创建、删除、写入权限  privilegeWrite///<读写，其他用户有创建、删除、写入权限  } |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_pri”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  // 设置bucket的访问权限为私有，即其他用户不可访问  if( bucket.setBucketPrivilege( EFSAccessPrivilege.privilegePrivate ) ){  System.out.println( “set bucket privilege: Private” );  }else{  System.out.println( “set bucket privilege failed, error:” + EFileSystem.getLastError() );  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + bucket默认访问权限为存储用户的访问权限
  + 无特殊需求的情况下无需重新设置Bucket的访问权限

#### 查询Bucket访问权限

* **接口定义**

**EFSAccessPrivilege Bucket.getBucketPrivilege();**

* **功能描述**

**获取bucket权限。**

* **接口参数**

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

* **接口返回值**
  + 非null：bucket的权限
  + null：失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_pri”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  // 获取bucket的访问权限  EFSAccessPrivilege pri = bucket.getBucketPrivilege();  if( null != pri ){  System.out.println( “The privilege of “ + bucket\_name + “ is “ + pri );  }else{  System.out.println( “get privilege failed, error:” + EFileSystem.getLastError() );  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

#### 设置Bucket配额容量

* **接口定义**

**ilenam Bucket.setCapacity(long megaBytes);**

* **功能描述**

**设置bucket最大容量。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **megaBytes** | **[IN] 需要设置的bucket容量大小，单位MB。-1代表不限制配额** |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_cap”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  // 设置bucket配额容量为1024MB  if( bucket.setCapacity ( 1024 ) ){  System.out.println( “set bucket capacity” );  }else{  System.out.println( “set bucket capacity failed, error:” + EFileSystem.getLastError() );  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + Bucket默认配额容量是存储用户的剩余容量，同一用户下的Bucket抢占式使用存储用户的剩余容量。
  + 无特殊需求时无需设置Bucket的配额容量
  + 一旦设置Bucket配额容量，这部分容量将会从用户剩余容量中扣除，被该Bucket独占式使用。
  + 设置的容量不得大于用户的剩余容量。

#### 查询Bucket配额容量

* **接口定义**

**long Bucket.getCapacity();**

* **功能描述**

获取bucket的配额容量。

* **接口参数**

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

* **接口返回值**
  + >0：Bucket的配额容量
  + -1：获取失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_cap”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  // 获取bucket的配额容量  long totalCap = bucket.getCapacity();  if( -1 != totalCap ){  System.out.println( “total is “ + totalCap + “MB” );  }else{  System.out.println( “get bucket capacity failed, error:” + EFileSystem.getLastError() );  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 未设置Bucket配额容量时调用该接口，则返回值也为-1

#### 设置Bucket生命周期

* **接口定义**

ilenam Bucket.setLifeCycle( long delay,

ilenam enableAction,

ilenam removeWhenBucketEmpty,

EFSRecycleAction act,

String prefix )

* **功能描述**

设置bucket生命周期。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| delay | [IN ] bucket下文件的过期时间，单位为秒，最小值为3600秒 |
| enableAction | [IN ] 是否使能参数act所指定的操作 |
| removeWhenBucketEmpty | [IN ] 当bucket下无文件时，是否删除 |
| act | [IN ] bucket过期时执行的操作，暂仅支持删除操作  public enum EFSRecycleAction{  doDelete ///<删除  } |
| prefix | [IN ] 保留字段 |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_lifecycle”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  // 设置bucket生命周期，bucket下的文件从创建30天后删除，且当Bucket下无文件时，删除Bucket  long delay = 30 \* 24 \* 3600;  ilenam enable\_action = true;  ilenam remove\_when\_bucket\_empty = true;  EFSRecycleAction act = EFSRecycleAction.doDelete;  String prefix = null; // 目前未使用必须为空  if( bucket.setLifeCycle( delay, enable\_action, remove\_when\_bucket\_empty, act, prefix ) ){  System.out.println( “set bucket lifecycle” );  }else{  System.out.println( “set bucket lifecycle failed, error:” + EFileSystem.getLastError() );  }  // 取消Bucket生命周期  enable\_action = false; // prefix必须为NULL，其他字段维持原来的值即可  if( bucket.setLifeCycle( delay, enable\_action, remove\_when\_bucket\_empty, act, prefix ) ){  System.out.println( “cancle bucket lifecycle” );  }else{  System.out.println( “cancle bucket lifecycle failed, error:” + EFileSystem.getLastError() );  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 建议生命周期的时间不少于1天，生命周期过短可能导致文件正在写入时就被删除，这会导致数据写入丢失。
  + 如果参数enableAction设置为false，不会执行删除文件的操作，生命周期不会起作用，并且设置的delay也不会生效。

#### 查询Bucket生命周期

* **接口定义**

ilenam Bucket.getLifeCycle( LongByReference delay,

IntByReference enableAction,

IntByReference removeWhenBucketEmpty,

IntByReference act,

String prefix );

* **功能描述**

获取bucket生命周期配置。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| delay | [OUT ] bucket下文件的过期时间，单位为秒 |
| enableAction | [OUT ] 是否使能参数act所指定的操作 |
| removeWhenBucketEmpty | [OUT ] 当bucket为空时，是否删除 |
| act | [OUT ] bucket过期时执行的操作 |
| prefix | [IN] 保留字段 |

* **接口返回值**

（1） true：获取成功。

（2） false：获取失败。错误码可以调用EFileSystem.getLastError()获取。

* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_lifecycle”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  // 获取bucket生命周期配置  LongByReference get\_delay = new LongByReference();  IntByReference get\_enable\_action = new IntByReference();  IntByReference get\_remove\_when\_bucket\_empty = new IntByReference();  IntByReference get\_act = new IntByReference();  String prefix = null; // 目前未使用必须为空  if( bucket.getLifeCycle( get\_delay, get\_enable\_action, get\_remove\_when\_bucket\_empty, get\_act, prefix ) ){  System.out.println( “get bucket lifecycle” );  }else{  System.out.println( “get bucket lifecycle failed, error:” + EFileSystem.getLastError() );  }  // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

#### 获取Bucket下的文件个数

* **接口定义**

ilenam Bucket.getFileNum ( LongByReference bigFileNum,

LongByReference smallFileNum )

* **功能描述**

获取bucket下的文件个数。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| bigFileNum | [OUT ] 大文件个数 |
| smallFileNum | [OUT ] 小文件个数 |

* **接口返回值**

（1） true：获取成功。

（2） false：获取失败。可以调用EFileSystem.getLastError()获取错误码。

* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  // 检查bucket是否有效  if( null != bucket ){  LongByReference big\_file\_num = new LongByReference();  LongByReference small\_file\_num = new LongByReference();  if( bucket.getFileNum( big\_file\_num, small\_file\_num ) ){  System.out.println( “big file num:” + big\_file\_num.getValue()+ “, small file num:” + small\_file\_num.getValue() );  }else{  System.out.println( “get bucket file num failed, error:” + EFileSystem.getLastError() );  }  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 该接口在V1.072.0000003.7.R版本后提供
  + 因EFS延时删除文件的机制，该接口获取的文件数是有延迟的，即文件删除后，通过该接口获取的文件数不能及时反映。请不要在类似“当文件数达到一定值时，开始删除文件“这样的逻辑中使用该接口。
  + 该接口获取的小文件个数并不准确，实际返回的是Container的个数，请不要依赖该数值做逻辑处理。每个Container可能包含多个小文件。

### 管理File

对同一个文件对象不支持多线程并发调用其接口。

#### 创建文件

* **接口定义**

**File EFileSystem.createFile( String ilename, int n, int m, ilenam bigFile );**

* **功能描述**

**通过用户传入文件名来创建指定文件。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **fileName** | [IN] 文件名，必须是bucketname/filename的形式，必须符合命名规范 |
| n | [IN] 冗余规则N+M中的N，默认是4。 |
| m | [IN] 冗余规则N+M中的M，默认是1。 |
| bigFile | [IN] 是否大文件，true表示创建大文件，false表示创建小文件，默认值true。（建议：100M以下的建议创建小文件，100M以上的建议创建大文件） |

* **接口返回值**
  + 非null：成功。
  + null：失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| // 创建文件时，必须保证Bucket已存在  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 以4+1模式创建大文件，必须是系统支持的N+M模式，N+M模式最好实现为可配置项。可根据系统规模确定N+M  File file = efs.createFile( filename, 4, 1, true );  // 返回值为null说明创建失败了  if( null != file ){  // 文件创建成功，该文件只能用于写  // 结束写后，必须关闭文件  file.close();  System.out.println( “write over, write length:” + write\_len );  }else{  // 如果最后错误码是-80003，则说明文件已经存在  System.out.println( “create file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + Bucket必须是已经存在的
  + 根据系统规模及需求选择合适的N+M，保证性能以及空间利用率。建议将N+M实现为可配置项。
  + 通过该接口获取到的CFile对象只可用于写入
  + 文件名不区分大小写
  + 小文件写入的数据不得超过100MB
  + 大文件写入的数据不得超过2TB
  + 文件命名规范：[File注意事项](#File注意事项)
* **创建模拟目录**

EFS中不存在目录概念，但可通过该接口创建一个模拟目录，本质上是创建了一个size为0的文件，该文件也可进行读写，但不推荐对该文件进行读写。创建模拟目录时，只需要filename以’/’结尾即可。

|  |
| --- |
| // 创建时指定的目录名，必须是包含有bucket名字，注意要有斜杠’/’分割，且最后一个字符是’/’  String filename = “bucket1/helloworld/”;  // 以1+1模式创建大文件  File file = efs.createFile( filename, 1, 1, true );  // 返回值为null说明创建失败了  if( null != file ){  // 目录创建成功后，立即关闭，不推荐写数据  file.close();  }else{  System.out.println( “create dir failed, error:” + EFileSystem.getLastError() );  } |

#### 打开文件

* **接口定义**

File EFileSystem.openFile( String ilename, EFSFileMode mode );

* **功能描述**

打开指定名称的文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN] 文件名，文件名组成为bucketName+’/’+fileName |
| mode | EFSFileMode.fileModeRead, ///<读  EFSFileMode .fileModeAppend ///<大文件追加写(**暂时不支持**) |

* **接口返回值**
  + 非null：成功。
  + null：失败。错误码可以调用EFileSystem.getLastError()获取。
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  // null时说明打开失败  if( null != file ){  // 文件有效，开始读取  // 打开文件成功后，关闭文件  file.close();  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 该接口操作的文件必须是已经存在的
  + 该接口获取到的File对象只可用读取或追加写入，取决于第二个参数指定的模式

#### 上传文件

* **接口定义**

int File.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 |
| buf | [IN] 需要写入文件的数据，空间由调用者申请和释放。buf不能为NULL |
| offset | [IN] buf的偏移，写入从buf+offset开始的len大小的数据 |
| len | [IN] 需要写入的数据字节数，len必须大于0 |

* **接口返回值**
  + > 0：成功写入的字节数
  + 0：暂时不可写，请在休眠一段时间后，再次调用本接口。
  + -1：写入出错。可以调用EFileSystem.getLastError()获取错误码。
* **示例**

|  |
| --- |
| // 创建文件时，必须保证Bucket已存在  // 创建文件时，必须保证Bucket已存在  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 以4+1模式创建大文件，必须是系统支持的N+M模式，N+M模式最好实现为可配置项。可根据系统规模确定N+M  File file = efs.createFile( filename, 4, 1, true );  // 或者可通过CFile::open接口以追加写模式打开文件（文件必须已经存在），进行追加写入  // File file = efs.open( filename, Dahua::EFS::fileModeAppend );  // 返回值为null说明创建失败了  if( null != file ){  // 文件创建成功，该文件只能用于写  int want\_write\_len = 300 << 20; // 期望写入300M  int write\_once\_len = 32768\*4\*5; // 推荐每次写入的长度是32K\*N的倍数，此处N是4，倍数设置为5  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，保证第一次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;  // 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 = file.write( buf, write\_once\_offset, write\_once\_len – write\_once\_offset );  if( 0 < ret ){  write\_once\_offset += ret;  write\_len += 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( “write failed, error:” + EFileSystem.getLastError() );  break;  }  }  // 结束写后，必须关闭文件  file.close();  System.out.println( “write over, write length:” + write\_len );  }else{  System.out.println( “create file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 该接口为同步非阻塞接口，并非一定写入指定的len长度的数据，实际写入长度以返回值为准。
  + 大小文件只在创建时有区别，写入流程是一样的
  + 调用接口写入的数据会先保存在本地内存中，达到一个条带大小后，才会真正写入云存储。异常情况下可能丢失本地内存中的数据。
  + 可通过File.open()接口以追加写模式打开文件（文件必须已经存在），然后上传数据。使用追加写前，必须确认所使用的版本是否支持该功能（7.0版本之后支持）。

#### 下载文件

* **接口定义**

int File.read(byte[] buffer, int len)

* **功能描述**

**从文件中读指定长度的数据。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| buf | [IN] 从文件中读出数据放入buffer，buf的空间由外部申请和释放。 |
| len | [IN] 需要读到的字节数。 |

* **接口返回值**
  + > 0：实际读取的字节数。
  + 0：暂时不可读，建议用户在休眠一段时间后，重新调用该接口读取。
  + -1：读取错误。可以调用EFileSystem.getLastError()获取错误码。
  + -2：读到文件末尾。
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  // null时说明打开失败  if( null != file ){  // 文件有效，开始读取  int want\_read\_len = 10 << 20; // 期望读取10M  int read\_once\_len = 32768\*4\*5; // // 推荐每次读取的长度是32K\*N的倍数，此处N是4，倍数设置为5  byte[] buf = new byte[read\_once\_len]; // 读取的缓存  int read\_len = 0;  // 当读取到指定长度或文件尾或错误时退出  while( read\_len < want\_read\_len ){  int ret = file.read( buf, read\_once\_len );  if( 0 < ret ){  read\_len += ret;  int 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  }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;  }  }  // 打开文件成功后，关闭文件  file.close();  System.out.println( “read over, read length:” + read\_len );  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 该接口为同步非阻塞接口，并非一定读取指定的长度，实际读取长度以返回值为准。
  + 如果文件是边写边读，因写入机制，可能使得最后写入不满一个条带的数据可能无法立即读取，需要关闭文件后才能读取到
  + 如果文件是边写边读，立即写入立即读取有可能会读取失败，需要确保写入一个条带的数据后（推荐至少写入1M数据）再开始读取数据
  + 可结合File.seek()接口实现随机读取

#### 关闭文件

* **接口定义**

ilenam File.close();

* **功能描述**

**关闭文件。**

* **接口返回值**
  + true：关闭成功。
  + false：关闭过程中存在错误
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  // null时说明打开失败  if( null != file ){  // 文件创建或打开成功后，必须关闭文件，一般无需关心关闭的结果  file.close();  System.out.println( “read over, read length:” + read\_len );  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 一旦调用该接口关闭文件，不得调用该文件对象的任何接口

#### 删除文件

* **接口定义**

**ilenam EFileSystem.removeFile( String ilename );**

* **功能描述**

**删除指定名称的文件。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **fileName** | **[IN] 需要删除的文件名。** |

* **接口返回值**
  + true：删除成功。
  + false：删除失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  if( efs.removeFile( filename ) ){  System.out.println( “remove file” );  }else{  System.out.println( “remove file failed, error:” + EFileSystem.getLastError() );  } |

* + 删除文件前必须确保文件已经关闭

#### 锁定文件

* **接口定义**

ilenam EFileSystem.lockFile(String ilename);

* **功能描述**

锁定指定文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN ] 锁定的文件名 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 锁定文件后，该文件无法被删除，但可以被重命名  if( efs.lockFile( filename ) ){  System.out.println( “lock file” );  }else{  System.out.println( “unlock file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 锁定的文件不可被删除，但可以被重命名
  + 该功能在6.0版本之后才提供

#### 解锁文件

* **接口定义**

ilenam EFileSystem.unlockFile(String ilename);

* **功能描述**

解锁指定文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN ] 待解锁的文件名 |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 解锁文件后，文件可被删除  if( efs.unlockFile( filename ) ){  System.out.println( “unlock file” );  }else{  System.out.println( “unlock file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 解锁后的文件可被删除
  + 该功能在6.0版本之后才提供

#### 获取文件锁定状态

* **接口定义**

int EFileSystem.getFileLockStat(String ilename);

* **功能描述**

获取文件锁定状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| fileName | [IN ] 文件名 |

* **接口返回值**
  + 1：文件已锁定。
  + 0：文件未锁定。
  + -1：获取状态失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 获取文件锁定状态  int state = efs.getFileLockStat( filename );  if( 1 == state ){  // 文件已锁定  System.out.println( “file locked” );  }else if( 0 == state ){  // 文件未锁定  System.out.println( “file not lock” );  }else if( -1 == state ){  // 获取失败  System.out.println( “get lock state failed” + EFileSystem.getLastError() );  }else{  // 无其他错误码  throw new RuntimeException( “no other result” );  } |

* **提示**
  + 该功能在6.0版本之后才提供

#### 重命名文件

* **接口定义**

ilenam EFileSystem.renameFile( String oldFileName, String newFileName );

* **功能描述**

**重命名文件名。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| oldFileName | **[IN] 旧文件名字，**文件名的组成是bucket名+’/’+真实文件名 |
| newFileName | **[IN] 新文件名字，**文件名的组成是bucket名+’/’+真实文件名 |

* **接口返回值**

1. true：重命名成功。
2. false：重名失败。可以调用EFileSystem.getLastError()获取错误码。

* **示例**

|  |
| --- |
| // 指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String old\_filename = “bucket1/helloworld”;  // 新文件名的Bucket必须与老文件名的Bucket一致  String new\_filename = “bucket1/byeworld”;  if( efs.renameFile( old\_filename, new\_filename ) ){  System.out.println( “rename file” );  }else{  System.out.println( “rename file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 新文件名中的bucket必须与老文件名中的bucket一致，即不允许跨bucket重命名
  + 重命名时必须保证文件已经关闭

#### 监听文件状态

* **接口定义**

ilenam File.setListener(IFileListener listener);

* **功能描述**

监听文件状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| listener | [IN ] 文件监听者，为null时表示删除监听。 |

* **接口返回值**
  + true：设置成功
  + false：设置失败
* **示例**

|  |
| --- |
| // 继承自Dahua::EFS::IFileListener，实现文件状态处理  public static class FileListener implements IFileListener  {  @Override  public void onStateChanged(File file, EFSFileState state)  {  // 可根据文件状态进行处理，常规只处理Dahua::EFS::fileStateWarning状态  System.out.println( “file state is “ + state );  if( EFSFileState.fileStateWarning == state ){  // 该状态说明N+M写入时，小于等于M个Block写入异常，  // 为保证数据冗余度，可选择设置关闭文件的标识，  // 写线程检查该标识，关闭文件并重新创建文件写入  // 无特殊需求不建议使用该功能  }    return;  }  }  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 以4+1模式创建大文件  File file = efs.createFile( filename, 4, 1, true );  if( null != file ){  // 监听对象  FileListener listener = new FileListener();  // 监听文件状态  if( file.setListener( listener ) ){  System.out.println( “set file listener” );  }else{  // 一般不会失败  }    // 对文件进行操作,如果在此过程中文件状态出现变化，会调用FileListener::onFileStateChanged()    // 不再监听文件状态  if( file.setListener( null ) ){  System.out.println( “set file listener to null” );  }else{  // 一般不会失败  }    file.close();  }else{  System.out.println( “create file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 不能直接在onFileStateChanged()中关闭文件，否则可能导致异常
  + 不论读写都可监听文件状态变化
  + 对一个File对象重复设置监听对象时，内部使用最后一次设置的监听对象。
  + 除非确实需要对EFSFileState.fileStateWarning状态进行处理，否则一般情况下，只需根据File对象的接口返回值进行业务处理。
  + 由于是异步回调，请注意锁的使用，避免死锁

#### 获取文件属性

* **接口定义**

**EFSFileStat File.getFileStat();**

* **功能描述**

**获取指定文件的文件属性。**

* **接口返回值**
  + 非null：文件属性。
  + null：获取失败。可以调用EFileSystem.getLastError()获取错误码。
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  if( null != file ){  // 获取文件状态  EFSFileStat s = file.getFileStat();  if( null != s ){  System.out.println( “get file stat” );  System.out.println( “mode:” + s.mode );  System.out.println( “writing ? “ + ( 0 == s.isAppend ? “false” : “true” ) );  System.out.println( “N+M is “ + s.dataNum + “+” + s.parityNum );  System.out.println( “ctime: “ + s.cTime );  System.out.println( “mtime: “ + s.mTime );  System.out.println( “size: “ + s.totalSize );  }else{  System.out.println( “get file stat failed, error:” + EFileSystem.getLastError() );  }    file.close();  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

#### 设置文件选项

* **接口定义**

ilenam File.setOption(EFSOption option, Object obj);

* **功能描述**

**设置CFile高级配置信息。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| option | [IN ]系统配置选项。支持如下选项  EFSOption.efsRwMode //读写模式(0异步1同步，默认为0) |
| obj | [IN ]对应配置项的值。 |

* **接口返回值**
  + true：设置成功。
  + false：设置失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( null != file ){  // 设置为同步阻塞式读取（默认为非阻塞读取）  int option = 1;  if( file.setOption( EFSOption.efsRwMode, option ) ){  System.out.println( “set file option” );  }else{  System.out.println( “set file option failed, error:” + EFileSystem.getLastError() );  }  file.close();  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

* 提示
  + 一般情况下无需设置该属性

#### 获取文件选项

* **接口定义**

Object File.getOption(EFSOption option);

* **功能描述**

获取File的高级配置信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| option | [IN ]系统配置选项。支持如下选项  EFSOption.efsRwMode //读写模式(0异步1同步，默认为0) |

* **接口返回值**
  + 非null：对应的option的值
  + null：获取失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( null != file ){  Object get\_option = file.getOption( EFSOption.efsRwMode );  if( null != get\_option ){  int num = (Integer) get\_option;  System.out.println( “get file option:” + num );  }else{  System.out.println( “get file option failed, error:” + EFileSystem.getLastError() );  }  file.close();  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

#### 修改文件偏移

* **接口定义**

**ilenam File.seek(long offset , EFSFileLocation whence);**

* **功能描述**

**根据指定基准位置定位到指定偏移。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **offset** | **[IN] 相对于whence位置后在文件中的偏移** |
| **whence** | **[IN] 偏移的基准位置**  **public enum EFSFileLocation**  **{**  **efsBegin, ///<开始位置**  **efsCurrent, ///<当前位置**  **efsEnd ///<文件尾**  **}** |

* **接口返回值**
  + true：成功。
  + false：失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  if( null != file ){  // 修改偏移  if( file.seek( 100, EFSFileLocation.efsBegin ) ){  System.out.println( “seek to 100B” );  }else{  System.out.println( “file seek failed, error:” + EFileSystem.getLastError() );  }  file.close();  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 结合File.read()可实现随机读取
  + **EFileSystem.create()获取到的File不支持调用该接口，即不支持随机写入**

#### 获取文件偏移

* **接口定义**

long File.tell();

* **功能描述**

**获取文件当前位置。**

* **接口返回值**
  + >= 0：当前读到的位置相对于文件头的偏移。
  + < 0：获取失败。可以调用EFileSystem.getLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  String filename = “bucket1/helloworld”;  // 读打开文件  File file = efs.openFile( filename, EFSFileMode.fileModeRead );  if( null != file ){  // 获取文件当前偏移  long offset = file.tell();  if( 0 <= offset ){  System.out.println( “current offset is “ + offset );  }else{  System.out.println( “get file current offset failed, error:” + EFileSystem.getLastError() );  }  file.close();  }else{  System.out.println( “open file failed, error:” + EFileSystem.getLastError() );  } |

#### 分页查询Bucket下的文件名

* **接口定义**

**List<String> Bucket.listBucketFile(String begin, String end, int maxNumber);**

* **功能描述**

列出bucket内文件名落在(begin,end)范围的文件名列表。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **begin** | [IN] 起始文件名。无需包含Bucket名。如果begin为NULL，则从Bucket下具备最小值的文件开始查询，且包含该最小值的文件。 |
| **end** | [IN] 结束文件名。无需包含Bucket名。如果end为NULL，则查询到该Bucket下具有最大值的文件。 |
| **maxNumber** | [IN] 查询的最大文件个数。取值范围[1, 2048] |

* **接口返回值**
  + Bucket下的文件名列表
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  if( null != bucket ){  System.out.println( “get bucket “ + bucket\_name );  // 假设该Bucket下存在文件 a,b,c,d  String begin\_name = “b”;  String end\_name = “e”;  int max\_num = 1024;  while( true ) {  List<String> namelist = bucket.listBucketFile( begin\_name, end\_name, max\_num );  if( !namelist.isEmpty() ){  // 这里将列出文件c，d，查询到的文件名不包含Bucket名  for( String name : namelist ){  System.out.println( name );  }  if( max\_num > namelist.size() ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else {  // 可能还有文件，修改begin\_name继续查询  begin\_name = namelist.get( namelist.size() – 1 );  }  }else{  System.out.println( “list file failed or bucket is empty” );  break;  }  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果，可能导致死循环。
  + 如果需要获取该Bucket下的所有文件名，可以调用Bucket. getAllBucketFile ()。但请注意，如果Bucket下文件很多，Bucket. getAllBucketFile ()接口的耗时和占用的内存大小可能不符合预期。

#### 分页查询Bucket下的文件信息

* **接口定义**

List<EFSFileInfo> Bucket.listBucketFileInfo(String begin, String end, int maxNumber);

* **功能描述**

列出bucket内文件名落在(begin,end)范围的文件信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **begin** | [IN] 起始文件名。无需包含Bucket名。如果begin为NULL，则从Bucket下具备最小值的文件开始查询，且包含该最小值的文件。 |
| **end** | [IN] 结束文件名。无需包含Bucket名。如果end为NULL，则查询到该Bucket下具有最大值的文件。 |
| **maxNumber** | [IN] 查询的最大文件个数。取值范围[1, 2048] |

* **接口返回值**
  + Bucket下的文件信息列表
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  if( null != bucket ){  System.out.println( “get bucket “ + bucket\_name );  // 假设该Bucket下存在文件 a,b,c,d  String begin\_name = “b”;  String end\_name = “d”;  int max\_num = 1024;  while( true ) {  List<EFSFileInfo> filelist = bucket.listBucketFileInfo( begin\_name, end\_name, max\_num );  if( !filelist.isEmpty() ){  // 这里将列出文件c，d，查询到的文件名不包含Bucket名  for( EFSFileInfo file : filelist ){  System.out.println( “filename: “ + file.getName() );  System.out.println( “size: “ + file.size );  System.out.println( “create time: “ + file.cTime );  System.out.println( “modify time: “ + file.mTime );  }  if( max\_num > filelist.size() ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else {  // 可能还有文件，修改begin\_name继续查询  begin\_name = filelist.get( filelist.size() – 1 ).getName();  }  }else{  System.out.println( “list file failed or bucket is empty” );  break;  }  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果，可能导致死循环。

#### 分页查询指定目录下的文件信息

* **接口定义**

List<EFSFileInfo> Bucket.listBucketFileInfoByPrefix( String dir,

String delimiter,

String begin,

int maxNumber);

* **功能描述**

查询bucket内dir目录下的文件信息

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **dir** | [IN] 需要查找的文件夹路径，最后字符必须为’/’。无需包含Bucket名 |
| **delimiter** | [IN] 文件夹路径分隔符,目前仅支持’/’ |
| **begin** | [IN] 起始文件名。无需包含目录名。如果为NULL，则查询dir下所有文件信息 |
| **maxNumber** | [IN] 查找的最大文件数量。取值范围[1, 2048] |

* **接口返回值**
  + 目录下的文件信息列表。
* **示例**

|  |
| --- |
| String bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  Bucket bucket = efs.getBucket( bucket\_name );  if( null != bucket ){  System.out.println( “get bucket “ + bucket\_name );  /\*\*  \* 假设该Bucket下存在文件，a/, a/b, a/c/，a/c/d, a/e。可以认为其存在如下目录树  \* a/  \* |--  \* |--b  \* |--c/  \* | |--d  \* |--e  \*/  String dir = “a/”;  String begin\_name = “b”;  int max\_num = 1024;  while( true ){  // 列出a/目录下的文件，最后一个字符必须是’/’  // 分隔符只支持’/’  // begin是”b”，因此查询的结果是c/,e  List<EFSFileInfo> filelist = bucket.listBucketFileInfoByPrefix( dir, “/”, begin\_name, max\_num );  if( !filelist.isEmpty() ){  // 这里将列出文件c/,e。查询到的文件名不包含目录名  // 不会递归查询子目录  for( EFSFileInfo file : filelist ){  System.out.println( “filename: “ + file.getName() );  System.out.println( “size: “ + file.size );  System.out.println( “create time: “ + file.cTime );  System.out.println( “modify time: “ + file.mTime );  }  if( max\_num > filelist.size() ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else {  // 可能还有文件，修改begin\_name继续查询  begin\_name = filelist.get( filelist.size() – 1 ).getName();  }  }else{  System.out.println( “list file failed or bucket is empty” );  break;  }  }    // 创建或获取成功后必须关闭  bucket.close();  }else{  System.out.println( “get bucket failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果。
  + 此处使用”目录”进行描述，但实际是获取指定前缀的文件名，且只匹配到指定分隔符为止

### 获取系统信息

#### 获取系统资源信息

* **接口定义**

EFSSystemInfo EFileSystem.getSystemInfo();

* **功能描述**

获取当前EFileSystem的信息，包括集群节点数，客户端运行个数等。

* **接口返回值**
  + 非null：系统资源信息
  + null：获取失败。可以调用EFileSystem.getLastError()获取错误码。
* **示例**

|  |
| --- |
| EFSSystemInfo info = efs.getSystemInfo();  if( null != info ){  // 以下为所有用户的统计信息  System.out.println( “system uuid: “ + info.uuid ); // 系统唯一标识  System.out.println( “total nodes: “ + info.totalNodes ); // 系统中的总节点数  System.out.println( “total clients: “ + info.totalClient ); // 系统当前正运行的客户端  System.out.println( “current writing files: “ + info.curWriteFiles ); // 系统当前正在写的文件总数  // 以下为当前用户的信息，请注意在系统实际容量不足时，spaceLeft+spaceUsed不一定等于totalSpace  System.out.println( “total files: “ + info.totalFiles ); // 当前用户的文件总数  System.out.println( “total space: “ + info.totalSpace ); // 当前用户的总空间  System.out.println( “space left: “ + info.spaceLeft ); // 用户真正可用的剩余空间，可能是系统实际剩余空间  System.out.println( “space quota left: “ + info.spaceQuotaLeft ); // 剩余的可分配空间，即可以给Bucket的设定的最大配额  System.out.println( “used space: “ + info.spaceUsed ); // 用户已经使用了的空间  }else{  System.out.println( “get system info failed, error:” + EFileSystem.getLastError() );  } |

* **提示**
  + 其中totalFiles并非大文件个数和小文件个数的总和，而是大文件个数和container个数的总和，而一个container可能包含多个小文件。请不要在类似“当文件数达到一定值时，开始删除文件“这样的逻辑中依赖该数值。

#### 获取系统支持的N+M

* **接口定义**

List<EFSRedundanceCap> EFileSystem.getRedundanceCaps();

* **功能描述**

获取系统支持的N+M

* **接口返回值**
  + 非null：系统支持的N+M列表
  + null：获取失败。可以调用EFileSystem.getLastError()获取错误码。
* **示例**

|  |
| --- |
| List<EFSRedundanceCap> caps = efs.getRedundanceCaps();  if( !caps.isEmpty() ){  System.out.println( “support:” );  for( EFSRedundanceCap cap : caps ){  System.out.println( cap.dataNum + “+” + cap.parityNum );  }  }else{  System.out.println( “not support any N+M or get n+m failed” );  } |

## 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

### 程序启动时出现” java.lang.UnsatisfiedLinkError”

错误信息类似下面，不同平台下库名会有所不同

|  |
| --- |
| Exception in thread “main” java.lang.UnsatisfiedLinkError: libEFSClientCore64.so: cannot open shared object file: No such file or directory  at com.sun.jna.Native.open(Native Method)  at com.sun.jna.Native.open(Native.java:1759)  at com.sun.jna.NativeLibrary.loadLibrary(NativeLibrary.java:260)  at com.sun.jna.NativeLibrary.getInstance(NativeLibrary.java:398)  at com.sun.jna.Library$Handler.<init>(Library.java:147)  at com.sun.jna.Native.loadLibrary(Native.java:412)  at com.sun.jna.Native.loadLibrary(Native.java:391)  at com.dahuatech.efs.lib.EFSAdapter.<clinit>(EFSAdapter.java:28)  at com.dahuatech.efs.EFileSystem.init(EFileSystem.java:60)  at example.Example.main(Example.java:46) |

出现该问题是因为动态库并没有安装到动态库加载目录或缺少动态库，请阅读《[安装](#_安装)》章节，确定所有的动态库均已正确安装。

### Allocation size must be greater than zero

如果异常的堆栈如下，则是因为通过setOptions()设置日志目录时，指定的日志目录为空。

|  |
| --- |
| java.lang.IllegalArgumentException: Allocation size must be greater than zero  at com.sun.jna.Memory.<init>(Memory.java:87) ~[jna-4.1.0.jar:4.1.0 (b4)]  at com.dahuatech.efs.EFileSystem.setOptions(EFileSystem.java:233) ~[efsclient-1.0.jar:?] |

### Windows平台SDK初始化失败

当使用7.2版本后的SDK时可能出现该问题，SDK日志中会打印EFSClientCore.dll加载失败。安装该问题的原因可能是因为EFSHelper.dll所在目录不是系统搜索库的目录，即dll没有放在安装章节指定的目录下，也可能是因为未安装vc++运行库，可搜索visual c++ AIO Installer安装所有版本的Visual C++运行库。

# 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\_EFSSDK\_Eng\_[Platform]\_[bits]bit\_PS\_[Version].tar.gz

实例如下：

* Linux
  + 32位：DH\_EFSSDK\_Eng\_32bit\_PS\_V1.070.0001.5.R.180429.tar.gz
  + 64位：DH\_EFSSDK\_Eng\_64bit\_PS\_V1.070.0001.5.R.180429.tar.gz
* Windows
  + 32位：DH\_EFSSDK\_Eng\_Windows\_32bit\_PS\_V1.070.0001.5.R.180429.tar.gz
  + 64位：DH\_EFSSDK\_Eng\_Windows\_64bit\_PS\_V1.070.0001.5.R.180429.tar.gz

上述实例中的[Version]的070，代表为7.0版本，可以此类推。

安装包内包含库以及头文件，头文件在sdk\_include.tar中。

### 编译

需要包含的头文件：IntTypes.h、EFSAdapter.h

指定链接的库：

* Linux平台
  + 32位：libEFSAdapter.so
  + 64位：libEFSAdapter64.so
* windows平台
  + 32位：EFSAdapter.lib（EFSAdapter.dll的导入库文件）
  + 64位：EFSAdapter64.lib（EFSAdapter 64.dll的导入库文件）

具体编译过程可参考[Demo](#_Demo_1)

编译时

### 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，不能包含中文路径。

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
  + ldconfig配置文件中指定的目录
  + 在编译时通过`-Wl,-rpath`指定的目录，（由于编译和安装的目录结构可能不一样，不推荐该用法）
* windows平台
  + 应用程序所在目录
  + 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
    - libEFSAdapter.so
    - libEFSClient.so
    - libEFSClientCore.so（5.0版本之后才有）
  + 64位
    - libEFSAdapter64.so
    - libEFSClient64.so
    - libEFSClientCore64.so（5.0版本之后才有）
* Windows
  + 32位
    - EFSAdapter.dll
    - EFSClient.dll
    - EFSClientCore.dll（5.0版本之后才有）
    - EFSHelper.dll
  + 64位
    - EFSAdapter64.dll
    - EFSClient64.dll
    - EFSClientCore64.dll（5.0版本之后才有）
    - EFSHelper64.dll

### 升级

取最新的安装包，替换对应的SDK库即可。如果是从4.0+升级到5.0之后的版本，需要特别注意，不要遗漏了EFSClientCore库。Linux下需要删除/tmp/sdk目录下的文件，Windows下需要删除程序所在目录的upgrade目录

### 兼容性

目前无接口兼容性问题。

4.0+SDK与7.0版本云存储配套存在问题，需使用更新版本的SDK（6.0版本之后含6.0）

6.0版本之后的SDK与4.0+版本之后的云存储均可配套使用。

## 使用介绍

### 初始化

以下接口均不允许多线程调用，且在进程生命周期内，一般只需调用一次。

#### 连接EFS服务

* **接口原型**

EFSHandle efsCreate( EFSConfig\* cfg );

* **功能描述**

初始化并登陆到EFS服务端。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| cfg | [IN] 登陆配置信息  typedef struct \_\_EFSConfig  {  const char\* address; ///<EFS服务IP地址  uint32\_t port; ///<EFS服务端口  const char\* ilename; ///<登录名  const char\* password; ///<登录密码  char reserved[120]; ///<保留  }EFSConfig; |

* **接口返回值**
  + 不管是创建成功还是失败，该接口均返回一个EFS操作句柄，需要使用isEFSHandleValid()检测其是否有效。
* **示例**

|  |
| --- |
| EFSConfig cfg;  cfg.address = “10.35.68.136”; // EFS服务地址  cfg.port = 38100; // EFS服务端口，固定不变  cfg.userName = “root”;// 存储用户名  cfg.password = “12345678”; // 存储用户密码  // 连接EFS服务，获得对应的操作句柄  EFSHandle efs\_handle = efsCreate( &cfg );  // 必须检查返回的EFSHandle是否有效  if( EFS\_TRUE == isEFSHandleValid( efs\_handle ) ){  // 使用efs\_handle进行操作，比如创建bucket和文件  // 关闭后不得再使用efs\_handle，必须保证在此之前所有该efs\_handle创建打开的文件都已经关闭  efsClose( efs\_handle );  }else{  printf( “\ninitialize failed, error: %d\n”, efsGetLastError() );  } |

* **提示**
  + 您的工程中可以有多个EFSHandle，也可以只有一个EFSHandle。如无特殊需求，推荐只创建一个EFSHandle。
  + 如发生网络异常，内部会自动重连。如设置了监听，那么将会回调通知使用者。

#### 高级选项配置

* **接口定义**

EFS\_BOOL efsSetOption( EFSHandle handle,

EFSOption key,

const void\* value,

uint32\_t len );

* **功能描述**

设置系统高级配置信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] 需要配置的EFS操作句柄 |
| key | [IN] 配置项，可选值如下  typedef enum \_\_EFSOption  {  efsConcurrent, ///<并发数,取值范围[1, 20]  efsBufferSize, ///<缓存大小，以字节为单位  efsTimeOut, ///<消息超时时间，以毫秒为单位  efsLogOutput, ///<日志文件输出路径  efsLogLevel, ///<日志级别  efsRwMode, ///<读写模式(0异步1同步)  efsSmallFileBufferSize, ///<小文件缓存大小（7.2版本后弃用）  efsBufferMode ///<读缓存模式（7.2版本后弃用）  }EFSOption; |
| value | [IN ]配置项的值。 |
| len | [IN]配置项value的大小。单位B。 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。错误码可以调用efsGetLastError()获取
* **示例**

|  |
| --- |
| // NOTE: efs\_handle必须先创建好  // 设置日志目录(目录必须存在)。推荐设置，可以有效定位问题。（可实现为程序配置项）  const char\* log\_path = “/dev/shm”;  if( EFS\_TRUE == efsSetOption( efs\_handle, efsLogOutput, log\_path, strlen( log\_path ) ) ){  printf( “set log path\n” );  }else{  printf( “set log path failed, error: %d\n”, efsGetLastError() );  }  // 设置日志级别，对接过程中推荐设置为6即DEBUG。（可实现为程序配置项）  // 可选值由EFSAdapter.h中的日志级别的枚举确定，如果枚举值冲突或者可能被宏覆盖，请直接使用数值  int32\_t log\_level = DEBUGF;  if( EFS\_TRUE == efsSetOption( efs\_handle, efsLogLevel, &log\_level, sizeof( log\_level ) ) ){  printf( “set log level\n” );  }else{  printf( “set log path failed, error: %d\n”, efsGetLastError() );  } |

* **提示**
  + 必须在有效的EFSHandle上执行这些选项配置操作。
  + 推荐设置日志目录和级别，方便定位问题。日志文件占用最大空间为30\*50MB，会自行覆盖。对接测试时建议设置为DEBUG，发布时可配置为INFO。因有改动需求，建议实现为配置项。
  + efsConcurrent默认为8，为测试时最佳性能值，一般无需修改
  + efsBufferSize默认8个条带的大小，大文件的默认缓存是8\*32K\*(N+M)，小文件的默认条带是8\*4K\*(N+M)
  + 其他配置一般无需设置，可维持默认。
  + Windows下设置日志目录时请使用Linux格式，即选择以”/”作为目录分割符，比如d:/a/b/c

#### 监听EFS连接状态

* **接口定义**

EFS\_BOOL efsSetStateCallBack( EFSHandle handle, efsStateCallBack cb );

* **功能描述**

监听EFS连接状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] 需要监听的EFSHandle |
| cb | [IN] 状态变化时使用的回调函数，回调时可能的状态如下  typedef enum \_\_EFSState {  efsStateInit, ///初始状态  efsStateNormal, ///<efs客户端正常  efsStateException, ///<efs客户端异常，会进行自动恢复  efsStateError ///<efs客户端错误，可能是用户不存在或者密码错误，不可自动恢复，需要用户介入  }EFSState; |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| // EFS连接状态变化处理函数  void onEFSStateChanged( EFSHandle handle, EFSState state )  {  // 可根据EFS状态进行处理，常规用法是：当EFS连接状态异常时，切换为其他存储模式，保证数据安全  printf( “EFS state is %d\n”, state );  return;  }  // 监听EFS连接状态  if( EFS\_TRUE == efsSetStateCallBack( handle, onEFSStateChanged ) ){  printf( “set efs listener\n” );  }else{  // 一般不会失败  }  // 设置为NULL，不再监听EFS连接状态  if( EFS\_TRUE == efsSetStateCallBack( handle, NULL ) ){  printf( “set efs listener\n” );  }else{  // 一般不会失败  } |

* **提示**
  + 7.2版本后，允许在一个EFSHandle上设置多个监听，但是一旦设为NULL将清除所有监听。
  + 一般场景下，无需使用该功能。
  + 由于是异步回调，请注意锁的使用，避免加锁调用SDK的接口，避免死锁

#### 设置用户空间紧急覆盖策略

* **接口定义**

EFS\_BOOL efsSetRecyclePolicy( EFSHandle handle,

EFS\_BOOL enableRecycle ,

RecyclePolicy policy);

* **功能描述**

设置用户空间回收策略。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS句柄 |
| enableRecycle | [IN] 是否使能用户空间回收 |
| policy | [IN] 回收策略  typedef enum \_\_RecyclePolicy{  reduceByTime = 0, ///<等时间缩减  reduceByRatio = 1, ///<等比例缩减  }RecyclePolicy; |

* **接口返回值**
  + EFS\_TRUE：设置成功。
  + EFS\_FALSE：设置失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| EFS\_BOOL enable\_recycle = EFS\_TRUE; // 使能紧急覆盖  RecyclePolicy policy = reduceByTime; // 等时间删除  // 设置紧急覆盖后，只对该用户下的设置了生命周期的Bucket起作用  EFS\_BOOL ret = efsSetRecyclePolicy( efs, enable\_recycle, policy );  if( EFS\_TRUE == ret ){  printf( “set recycle policy\n” );  }else{  printf( “set recycle policy failed, error: %d\n”, Dahua::EFS::getLastError() );  } |

* **提示**
  + 6.0版本后支持该功能。
  + 紧急覆盖仅对使能生命周期的Bucket有效。默认情况下，用户的使用容量达到95%时，开始删除文件，释放空间。
  + 等时间缩减指该用户下所有Bucket删除<Bucket生命周期+固定时间>时间前创建的文件
  + 等比例缩减指该用户下所有Bucket删除<Bucket生命周期\*固定比例>时间前创建的文件

#### 获取用户空间紧急覆盖策略

* **接口定义**

EFS\_BOOL efsGetRecyclePolicy( EFSHandle handle,

EFS\_BOOL\* enableRecycle ,

RecyclePolicy\* policy );

* **功能描述**

获取用户空间回收策略配置。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS句柄 |
| enableRecycle | [OUT ] 是否使能用户空间回收 |
| policy | [OUT ] 回收策略  enum RecyclePolicy{  reduceByTime = 0, ///<等时间缩减  reduceByRatio = 1, ///<等比例缩减  }; |

* **接口返回值**
  + EFS\_TRUE：设置成功。
  + EFS\_FALSE：设置失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| EFS\_BOOL enable\_recycle;  RecyclePolicy policy;  // 获取紧急覆盖策略  ret = efsGetRecyclePolicy( efs, &enable\_recycle, &policy );  if( EFS\_TRUE == ret ){  printf( “get recycle policy” );  }else{  printf( “get recycle policy failed, error: %d\n”, efsGetLastError() );  } |

* **提示**
  + 6.0版本后支持该功能。

#### 退出EFS服务

* **接口定义**

EFS\_BOOL efsClose( EFSHandle handle );

* **功能描述**

断开与EFS服务的连接。

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

|  |
| --- |
| EFSHandle efs = efsCreate( &cfg );  if( EFS\_TRUE == isEFSHandleValid( efs ) ){  // 初始化成功后，当不再使用时，必须关闭  (void)efsClose( efs );  }else{  std::cout << “initialize failed, error:” << Dahua::EFS::getLastError() << std::endl;  } |

* **提示**
  + 调用efsClose()前，请务必保证通过该EFSHandle创建或打开的文件已经关闭。
  + 一般无需判断efsClose()的返回值。
  + efsClose()仅在确定不再使用EFSHandle时调用，比如进程退出前。
  + 一旦调用efsClose()后，该EFSHandle不可再使用

### 管理Bucket

对于一个Bucket，以下接口均不允许多线程调用，但多个Bucket之间互不影响。

#### 创建Bucket

* **接口定义**

**EFSBucketHandle efsCreateBucket( EFSHandle handle, const char\* name );**

* **功能描述**

**创建Bucket**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] EFS操作句柄** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + 不管是创建成功还是失败，该接口返回的是一个**Bucket的操作句柄**，必须使用**isEFSBucketHandleValid()**检测其是否有效，若返回**isEFSBucketHandleValid()**返回EFS\_FALSE，则可以调用efsGetLastError()获取错误码。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsCreateBucket( efs, bucket\_name );  // 必须检查返回的CBucket对象是否有效，无效说明创建失败了  EFS\_BOOL ret = isEFSBucketHandleValid( bucket );  if( EFS\_TRUE == ret ){  printf( “create bucket %s”, bucket\_name );  // 操作CBucket对象  // ...  // 创建或获取Bucket成功后，必须关闭  efsCloseBucket( bucket );  }else if( -80005 == efsGetLastError() ){  // 错误码是-80005（Bucket已经存在），则需要通过获取Bucket接口获得CBucket对象  }else{  printf( “create bucket failed, error:%d”, efsGetLastError() );  } |

* **提示**
  + 创建的Bucket的访问权限继承自存储用户的访问权限，可通过接口重新设置
  + Bucket名字必须全局唯一，您需要保证Bucket名不会与其他用户冲突。
  + 当创建失败且错误码是-80005（Bucket已存在）时，可直接获取Bucket。
  + 因为单个Bucket下的文件过多时，会影响读取性能，所以推荐每天创建不同的Bucket进行存储。
  + Bucket命名规范可以参考：[bucket注意事项](#bucket注意事项)

#### 删除Bucket

* **接口定义**

**EFS\_BOOL efsRemoveBucket( EFSHandle handle, const char\* name );**

* **功能描述**

**删除指定名称的bucket。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
|  |  |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + true：删除成功。
  + false：删除失败，错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket1”; // bucket名必须符合命名规范  // 删除Bucket  if( EFS\_TRUE == efsRemoveBucket( efs, bucket\_name ) ){  printf( “remove bucket %s”, bucket\_name );  }else{  printf( “remove bucket failed, error:%d”, efsGetLastError() );  } |

* **提示**
  + 删除Bucket前，必须先删除bucket下的所有文件，不然会返回失败，错误码为-80006。

#### 获取Bucket

* **接口定义**

**EFSBucketHandle efsGetBucket( EFSHandle handle, const char\* name );**

* **功能描述**

**根据bucket名称获取bucket对象。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] EFS的操作句柄** |
| **name** | **[IN] bucket名称，必须符合命名规范** |

* **接口返回值**
  + 不管是创建成功还是失败，该接口返回的是一个**Bucket的操作句柄**，必须使用**isEFSBucketHandleValid()**检测其是否有效。
* **示例**

|  |
| --- |
| // 确定Bucket已存在，直接获取  const char\* bucket\_name = “bucket1”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  printf( “get bucket %s\n”, bucket\_name );  // 操作CBucket对象  // ...  // 创建或获取Bucket成功后，必须关闭  efsCloseBucket( bucket );  }else if( -80004 == efsGetLastError() ){  // 错误码是-80004（Bucketn不存在），则需要通过创建Bucket接口获得CBucket对象  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 当调用该接口失败且错误码是-80004（Bucket不存在）时，需要先创建Bucket

#### 关闭Bucket

* **接口定义**

**EFS\_BOOL efsCloseBucket( EFSBucketHandle handle );**

* **功能描述**

关闭Bucket。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] Bucket的操作句柄** |

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

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsCreateBucket( efs, bucket\_name );  // 必须检查返回的CBucket对象是否有效，无效说明创建失败了  EFS\_BOOL ret = isEFSBucketHandleValid( bucket );  if( EFS\_TRUE == ret ){  printf( “create bucket %s”, bucket\_name );  // 操作CBucket对象  // ...  // 创建或获取Bucket成功后，必须关闭，一般无需关心结果  (void)efsCloseBucket( bucket );  }else{  printf( “create bucket failed, error:%d”, efsGetLastError() );  } |

* **提示**
  + **创建或获取Bucket句柄成功后，必须关闭，一般无需关系关闭的结果**

#### 列出Bucket

* **接口定义**

**int32\_t efsListBucket( EFSHandle handle,**

**EFSTokenHandle token,**

**uint32\_t maxNumber,**

**char buffer[][MAX\_BUCKET\_NAME\_LEN] );**

* **功能描述**

列出Bucket。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] EFS的操作句柄** |
| **token** | **[IN] 记录查询Bucket的分页信息** |
| **maxNumber** | **[IN] 最多可能列出的Bucket个数，建议取值小于1024。** |
| **buffer** | **[OUT] 存放列出的bucket名, MAX\_BUCKET\_NAME\_LEN为64，因为Bucket名不得超过64** |

* **接口返回值**
  + >=0：获取成功，实际获取到的bucket个数，如果小于maxNumber说明当前EFS中的Bucket已经全部获取结束。
  + -1：获取失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| typedef char (\*BucketList)[MAX\_BUCKET\_NAME\_LEN];  EFSTokenHandle token = efsGetToken(); // 请注意 ，后续必须释放token，否则存在内存泄露  uint32\_t max\_num = 1024;  BucketList buffer = (BucketList)malloc( max\_num \* MAX\_BUCKET\_NAME\_LEN );  assert( NULL != buffer );  do{  int32\_t size = efsListBucket( efs, token, max\_num, buffer );  if( 0 <= size ){  // 输出bucket名字  printf( “list bucket, num:%d\n”, size );  for( int32\_t I = 0; I < size; ++I ){  printf( buffer[i] );  }    if( size < 1024 ){  // bucket已经全部获取到  break;  }  }else{  // 获取失败，不再获取  printf( “list bucket failed, error:%d\n”, efsGetLastError() );  break;  }  }while( 1 );  // 请注意 ，必须释放token，否则存在内存泄露  efsReleaseToken( token );  free( buffer );  buffer = NULL; |

* **提示**
  + **在用户list过程中，如存在删除bucket的操作，list可能存在遗漏的情况。**

#### 设置Bucket访问权限

* **接口定义**

**EFS\_BOOL efsSetBucketPrivilege( EFSBucketHandle handle, AccessPrivilege privilege );**

* **功能描述**

设置bucket权限。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] Bucket的操作句柄** |
| **privilege** | **[IN] 需要设置的bucket权限。**  typedef enum \_\_AccessPrivilege {  privilegePrivate = 0,///<私有，其他用户不可见  privilegeRead = 1,///<只读，其他用户无创建、删除、写入权限  privilegeWrite = 2///<读写，其他用户有创建、删除、写入权限  }AccessPrivilege; |

* **接口返回值**
  + EFS\_TRUE：设置成功。
  + EFS\_FALSE：设置失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_pri”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  // 设置bucket的访问权限  if( EFS\_TRUE == efsSetBucketPrivilege( bucket, privilegePrivate ) ){  printf( “set bucket privilege: Private” );  }else{  printf( “set bucket privilege failed, error:%d”, efsGetLastError() );  }  }else{  printf( “get bucket failed, error:%d”, efsGetLastError() );  } |

* **提示**
  + bucket默认访问权限为存储用户的访问权限
  + 无特殊需求的情况下无需重新设置Bucket的访问权限

#### 查询Bucket访问权限

* **接口定义**

**EFS\_BOOL efsGetBucketPrivilege( EFSBucketHandle handle, AccessPrivilege\* privilege );**

* **功能描述**

**获取bucket权限。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] Bucket的操作句柄** |
| **privilege** | **[IN] 获取到的bucket权限。**  typedef enum \_\_AccessPrivilege {  privilegePrivate = 0,///<私有，其他用户不可见  privilegeRead = 1,///<只读，其他用户无创建、删除、写入权限  privilegeWrite = 2///<读写，其他用户有创建、删除、写入权限  }AccessPrivilege; |

* **接口返回值**
  + EFS\_TRUE：设置成功。
  + EFS\_FALSE：设置失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_pri”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  // 获取bucket的访问权限  AccessPrivilege pri;  if( EFS\_TRUE == efsGetBucketPrivilege( bucket, &pri ) ){  printf( “The privilege of %s is %d”, bucket\_name, pri );  }else{  printf( “get privilege failed, error:%d”, efsGetLastError() );  }  }else{  printf( “get bucket failed, error:%d”, efsGetLastError() );  } |

#### 设置Bucket配额容量

* **接口定义**

**EFS\_BOOL efsSetBucketCapacity( EFSBucketHandle handle, uint64\_t megaBytes );**

* **功能描述**

**设置bucket最大容量。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] Bucket的操作句柄** |
| **megaBytes** | **[IN] 需要设置的bucket容量大小，单位MB。-1代表不限制配额** |

* **接口返回值**
  + EFS\_TRUE：设置成功。
  + EFS\_FALSE：设置失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_cap”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  // 设置bucket配额容量为1024MB  if( EFS\_TRUE == efsSetBucketCapacity( bucket, 1024 ) ){  printf( “set bucket capacity\n” );  }else{  printf( “set bucket capacity failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + Bucket默认配额容量是存储用户的剩余容量，同一用户下的Bucket抢占式使用存储用户的剩余容量。
  + 无特殊需求时无需设置Bucket的配额容量
  + 一旦设置Bucket配额容量，这部分容量将会从用户剩余容量中扣除，被该Bucket独占式使用。
  + 设置的容量不得大于用户的剩余容量。

#### 查询Bucket配额容量

* **接口定义**

**EFS\_BOOL efsGetBucketCapacity( EFSBucketHandle handle,**

**uint64\_t\* megaBytes,**

**uint64\_t\* usedMegaBytes );**

* **功能描述**

获取bucket最大容量和已经使用的容量。

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] bucket的操作句柄** |
| **megaBytes** | **[OUT] 获取到的bucket最大容量，单位MB。** |
| **usedMegaBytes** | **[OUT] 获取到的bucket已经使用的容量，单位MB。** |

* **接口返回值**
  + EFS\_TRUE：查询成功。
  + EFS\_FALSE：查询失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_cap”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  // 获取bucket的配额容量以及使用容量  uint64\_t totalCap = 0;  uint64\_t usedCap = 0;  if( EFS\_TRUE == efsGetBucketCapacity( bucket, &totalCap, &usedCap ) ){  printf( “total is %”PRIu64”MB, used %”PRIu64”MB\n”, totalCap, usedCap );  }else{  printf( “get bucket capacity failed, error:%d\n”, efsGetLastError() );  }  }else{  p |

* **提示**
  + 未设置Bucket配额容量时调用该接口，则只有Bucket的已使用容量有效。

#### 设置Bucket生命周期

* **接口定义**

EFS\_BOOL efsSetBucketLifeCycle( EFSBucketHandle handle,

int64\_t delay,

EFS\_BOOL enableAction,

EFS\_BOOL removeWhenBucketEmpty,

RecycleAction act,

const char\* prefix);

* **功能描述**

设置bucket生命周期。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] bucket的操作句柄 |
| delay | [IN ] bucket下文件的过期时间，单位为秒，最小值为3600秒 |
| enableAction | [IN ] 是否使能参数act所指定的操作 |
| removeWhenBucketEmpty | [IN ] 当bucket下无文件时，是否删除 |
| act | [IN ] bucket过期时执行的操作，暂仅支持删除操作  enum RecycleAction{  doDelete = 0 ///<删除  }; |
| prefix | [IN ] 保留字段 |

* **接口返回值**
  + EFS\_TRUE：设置成功。
  + EFS\_FALSE：设置失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_lifecycle”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  // 检查bucket是否有效  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  // 设置bucket生命周期，bucket下的文件从创建30天后删除，且当Bucket下无文件时，删除Bucket  int64\_t delay = 30 \* 24 \* 3600;  EFS\_BOOL enableAction = EFS\_TRUE;  EFS\_BOOL removeWhenBucketEmpty = EFS\_TRUE;  RecycleAction act = doDelete;  const char\* prefix = NULL; // 目前未使用必须为NULL  if( EFS\_TRUE == efsSetBucketLifeCycle( bucket, delay, enableAction, removeWhenBucketEmpty, act, prefix ) ){  printf( “set bucket lifecycle\n” );  }else{  printf( “set bucket lifecycle failed, error:%d\n”, efsGetLastError() );  }    // 取消Bucket生命周期  enableAction = EFS\_FALSE; // prefix必须为NULL，其他字段维持原来的值即可  if( EFS\_TRUE == efsSetBucketLifeCycle( bucket, delay, enableAction, removeWhenBucketEmpty, act, prefix ) ){  printf( “cancle bucket lifecycle\n” );  }else{  printf( “cancle bucket lifecycle failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 建议生命周期的时间不少于1天，生命周期过短可能导致文件正在写入时就被删除，这会导致数据写入丢失。
  + 如果参数enableAction设置为false，不会执行删除文件的操作，生命周期不会起作用，并且设置的delay也不会生效。

#### 查询Bucket生命周期

* **接口定义**

EFS\_BOOL efsGetBucketLifeCycle( EFSBucketHandle handle,

int64\_t\* delay,

EFS\_BOOL\* enableAction,

EFS\_BOOL\* removeWhenBucketEmpty,

RecycleAction\* act,

const char\* prefix );

* **功能描述**

获取bucket生命周期配置。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] bucket的操作句柄 |
| delay | [OUT ] bucket下文件的过期时间，单位为秒 |
| enableAction | [OUT ] 是否使能参数act所指定的操作 |
| removeWhenBucketEmpty | [OUT ] 当bucket为空时，是否删除 |
| act | [OUT ] bucket过期时执行的操作 |
| prefix | [OUT ] 保留字段 |

* **接口返回值**
  + EFS\_TRUE：查询成功。
  + EFS\_FALSE：查询失败。错误码可以调用efsGetLastError()获取。
* **示例**

|  |
| --- |
| const char\* bucketName = “bucket\_lifecycle”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  // 检查bucket是否有效  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  int64\_t delay;  EFS\_BOOL enableAction;  EFS\_BOOL removeWhenBucketEmpty;  RecycleAction act;  const char\* prefix = NULL; // 目前未使用必须为NULL  // 获取bucket生命周期配置  if( EFS\_TRUE == efsGetBucketLifeCycle( bucket, &delay, &enableAction, &removeWhenBucketEmpty, &act, prefix ) ){  printf( “get bucket lifecycle\n” );  }else{  printf( “get bucket lifecycle failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

#### 获取Bucket下的文件个数

* **接口定义**

EFS\_BOOL efsGetBucketFileNum ( EFSBucketHandle handle,

uint64\_t\* bigFileNum,

uint64\_t\* smallFileNum )

* **功能描述**

获取bucket下的文件个数。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] Bucket的操作句柄 |
| bigFileNum | [OUT ] 大文件个数 |
| smallFileNum | [OUT ] 小文件个数 |

* **接口返回值**

（1） EFS\_TRUE：获取成功。

（2） EFS\_FALSE：获取失败。可以调用efsGetLastError()获取错误码。

* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  // 检查bucket是否有效  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  uint64\_t big\_file\_num = 0;  uint64\_t small\_file\_num = 0;  if( efsGetBucketFileNum( bucket, &big\_file\_num, &small\_file\_num ) ){  printf( “big file num:%”PRIu64”, small file num:%”PRIu64”\n”, big\_file\_num, small\_file\_num );  }else{  printf( “get bucket file num failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 该接口在V1.072.0000003.7.R版本后提供
  + 因EFS延时删除文件的机制，该接口获取的文件数是有延迟的，即文件删除后，通过该接口获取的文件数不能及时反映。请不要在类似“当文件数达到一定值时，开始删除文件“这样的逻辑中使用该接口。
  + 该接口获取的小文件个数并不准确，实际返回的是Container的个数，请不要依赖该数值做逻辑处理。每个Container可能包含多个小文件。

### 管理File

对于一个File，以下接口均不允许多线程调用，但不同File之间互不影响。

#### 创建文件

* **接口定义**

EFSFileHandle efsCreateFile( EFSHandle handle,

const char\* ilename,

uint16\_t n,

uint16\_t m,

EFS\_BOOL bigFile );

* **功能描述**

**通过用户传入文件名来创建指定文件。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS的操作句柄 |
| **fileName** | [IN] 文件名，必须是bucketname/filename的形式，必须符合命名规范 |
| n | [IN] 冗余规则N+M中的N。 |
| m | [IN] 冗余规则N+M中的M。 |
| bigFile | [IN] 是否大文件，EFS\_TRUE表示创建大文件，EFS\_FALSE表示创建小文件。（建议：100M以下的建议创建小文件，100M以上的建议创建大文件） |

* **接口返回值**
  + 不管是创建成功还是失败, 接口都会返回一个EFSFileHandle。必须使用isEFSFileHandleValid ()检测文件是否有效，若返回isEFSFileHandleValid ()返回EFS\_FALSE，则可以调用efsGetLastError()获取错误码。
* **示例**

|  |
| --- |
| // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 以4+1模式创建大文件  EFSFileHandle file = efsCreateFile( efs, filename, 4, 1, EFS\_TRUE );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 文件创建成功，该文件只能用于写  // 结束写后，必须关闭文件  efsCloseFile( file );  }else{  // 如果最后错误码是-80003，则说明文件已经存在  printf( “create file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + Bucket必须是已经存在的
  + 根据系统规模及需求选择合适的N+M，保证性能以及空间利用率。建议将N+M实现为可配置项。
  + 通过该接口获取到的CFile对象只可用于写入
  + 文件名不区分大小写
  + 小文件写入的数据不得超过100MB
  + 大文件写入的数据不得超过2TB
  + 文件命名规范：[File注意事项](#File注意事项)
* **创建模拟目录**

EFS中不存在目录概念，但可通过该接口创建一个模拟目录，本质上是创建了一个size为0的文件，该文件也可进行读写，但不推荐对该文件进行读写。创建模拟目录时，只需要filename以’/’结尾即可。

|  |
| --- |
| // 创建时指定的目录名，必须是包含有bucket名字，注意要有斜杠’/’分割，且最后一个字符是’/’  const char\* dirname = “ bucket1/helloworld/”;  // 以1+1模式创建  EFSFileHandle dir = efsCreateFile( efs, dirname, 1, 1, EFS\_TRUE );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 目录创建成功后，立即关闭，不推荐写数据  efsCloseFile( dir );  }else{  printf( “create directory failed, error:%d\n”, efsGetLastError() );  } |

#### 打开文件

* **接口定义**

EFSFileHandle efsOpenFile( EFSHandle handle,

const char\* ilename,

EFSFileMode mode );

* **功能描述**

打开指定名称的文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS的操作句柄 |
| fileName | [IN] 文件名，文件名组成为bucketName+’/’+fileName |
| mode | [IN] 打开模式。  可选值：  fileModeRead 读打开  fileModeAppend 追加写打开（**暂时不支持**） |

* **接口返回值**

不管是创建成功还是失败, 接口都会返回一个EFSFileHandle。必须使用isEFSFileHandleValid ()检测文件是否有效，若返回isEFSFileHandleValid ()返回EFS\_FALSE，则可以调用efsGetLastError()获取错误码。

* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 文件有效，开始读取    // 结束操作后，必须关闭文件  efsCloseFile( file );  }else{  printf( “open file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 该接口操作的文件必须是已经存在的
  + 该接口获取到的File只可用读取或追加写入，这取决于第二个参数指定的模式

#### 上传文件

* **接口定义**

int32\_t efsWrite( EFSFileHandle handle, const char\* buf, uint32\_t len );

* **功能描述**

**写入指定长度的数据。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS的操作句柄 |
| buf | [IN] 需要写入文件的数据，空间由调用者申请和释放。buf不能为NULL |
| len | [IN] 需要写入的数据字节数，len必须大于0 |

* **接口返回值**
  + > 0：成功写入的字节数
  + 0：暂时不可写，请在休眠一段时间后，再次调用本接口。
  + -1：写入出错。可以调用getLastError()获取错误码。
* **示例**

|  |
| --- |
| // 创建文件时，必须保证Bucket已存在  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 以4+1模式创建大文件  EFSFileHandle file = efsCreateFile( efs, filename, 4, 1, EFS\_TRUE );  // 或者可通过efsOpenFile接口以追加写模式打开文件（文件必须已经存在），进行追加写入  // EFSFileHandle file = efsOpenFile( efs, filename, fileModeAppend );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 文件有效，开始写入  uint32\_t want\_write\_len = 300 << 20; // 期望写入300M  uint32\_t write\_once\_len = 32768\*4\*5; // 推荐每次写入的长度是32K\*N的倍数，此处N是4，倍数设置为5  char\* buf = (char\*)malloc( write\_once\_len ); // 待写入数据的缓存  assert( NULL != buf );  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 = efsWrite(file, buf + write\_once\_offset, write\_once\_len – write\_once\_offset );  if( 0 < ret ){  write\_once\_offset += ret;  write\_len += 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( “write failed, error:%d\n”, efsGetLastError() );  break;  }else{  // 目前不存在其他返回值  assert( 0 );  }  }  // 创建文件成功后，必须关闭文件  efsCloseFile( file );  free( buf );  buf = NULL;  printf( “write over, write length:%u\n”, write\_len );  }else{  printf( “create file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 该接口为同步非阻塞接口，并非一定写入指定的len长度的数据，实际写入长度以返回值为准。
  + 大小文件只在创建时有区别，写入流程是一样的
  + 调用接口写入的数据会先保存在本地内存中，达到一个条带大小后，才会真正写入云存储。异常情况下可能丢失本地内存中的数据。
  + 调用接口时，每次写入的长度推荐为32K\*N的倍数
  + 可通过efsOpenFile()接口以追加写模式打开文件（文件必须已经存在），然后上传数据。使用追加写前，必须确认所使用的版本是否支持该功能（7.0版本之后支持）。
  + 请不要多线程调用该接口，除非您能确定写入数据的正确性。

#### 下载文件

* **接口定义**

int32\_t efsRead( EFSFileHandle handle, char\* buf, uint32\_t len );

* **功能描述**

**从文件中读指定长度的数据。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] File的操作句柄 |
| buf | [IN] 从文件中读出数据放入buffer，buf的空间由外部申请和释放。 |
| len | [IN] 需要读到的字节数。 |

* **接口返回值**
  + > 0：实际读取的字节数。
  + 0：暂时不可读，建议用户在休眠一段时间后，重新调用该接口读取。
  + -1：读取错误。可以调用efsGetLastError()获取错误码。
  + -2：读到文件末尾。
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 文件有效，开始读取  uint32\_t want\_read\_len = 10 << 20; // 期望读取10M  uint32\_t read\_once\_len = 32768\*4\*5; // // 推荐每次读取的长度是32K\*N的倍数，此处N是4，倍数设置为5  char\* buf = (char\*)malloc( read\_once\_len ); // 读取数据的缓存  assert( NULL != buf );  uint32\_t read\_len = 0; // 实际读取的长度  // 读取指定长度  while( read\_len < want\_read\_len ){  int ret = efsRead( file, 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, error:%d\n”, efsGetLastError() );  break;  }else if( -2 == ret ){  // 读取到文件尾  printf( “read end\n” );  break;  }  }  // 结束操作后，必须关闭文件  efsCloseFile( file );  free( buf );  buf = NULL;  printf( “read over, read length:%u\n”, read\_len );  }else{  printf( “open file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 该接口为同步非阻塞接口，并非一定读取指定的长度，实际读取长度以返回值为准。
  + 如果文件是边写边读，因写入机制，可能使得最后写入不满一个条带的数据可能无法立即读取，需要关闭文件后才能读取到
  + 如果文件是边写边读，立即写入立即读取有可能会读取失败，需要确保写入一个条带的数据后（推荐至少写入1M数据）再开始读取数据
  + 调用接口时，每次读取的长度可以设置为处理该数据的最佳长度。比如从EFS读取数据后，会写入本地文件，因为写入本地文件时每次写32K最佳，所以从EFS读取时每次读取32K
  + 可结合efsSeek()接口实现随机读取
  + 请不要多线程调用该接口，除非您能确定这样的方式读取数据时，数据的正确性。

#### 关闭文件

* **接口定义**

EFS\_BOOL efsCloseFile( EFSFileHandle handle );

* **功能描述**

**关闭文件。**

* **接口返回值**
  + EFS\_TRUE：关闭成功。
  + EFS\_FALSE：关闭过程中存在错误
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的CFile对象是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){    // 文件创建或打开成功后，必须关闭文件，一般无需关心关闭的结果  (void)efsCloseFile( file );  }else{  printf( “open file failed, error:%d\n”, efsGetLastError() );  } |

* 一旦调用该接口关闭文件，不得调用该文件的任意接口

#### 删除文件

* **接口定义**

**EFS\_BOOL efsRemoveFile( EFSHandle handle, const char\* ilename )；**

* **功能描述**

**删除指定名称的文件。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] EFS的操作句柄** |
| **fileName** | **[IN] 需要删除的文件名。** |

* **接口返回值**
  + EFS\_TRUE：删除成功。
  + EFS\_FALSE：删除失败。可以调用efsGetLastError ()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  if( EFS\_TRUE == efsRemoveFile( efs, filename ) ){  printf( “remove file\n” );  }else{  printf( “remove file failed, error:%d\n”, efsGetLastError() );  } |

* + 删除文件前必须确保文件已经关闭

#### 锁定文件

* **接口定义**

EFS\_BOOL efsLockFile(EFSHandle handle, const char\* ilename);

* **功能描述**

锁定指定文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | **[IN] EFS的操作句柄** |
| fileName | [IN ] 锁定的文件名 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。可以调用efsGetLastError ()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 锁定文件后，该文件无法被删除，但可以被重命名  if( efsLockFile( efs, filename ) ){  printf( “lock file\n” );  }else{  printf( “unlock file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 锁定的文件不可被删除，但可以被重命名
  + 该功能在6.0版本之后才提供

#### 解锁文件

* **接口定义**

EFS\_BOOL efsUnlockFile(EFSHandle handle, const char\* ilename);

* **功能描述**

解锁指定文件。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | **[IN] EFS的操作句柄** |
| fileName | [IN ] 待解锁的文件名 |

* **接口返回值**
  + EFS\_TRUE：成功。
  + EFS\_FALSE：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 解锁文件后，文件可被删除  if( efsUnlockFile( efs, filename ) ){  printf( “unlock file\n” );  }else{  printf( “unlock file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 解锁后的文件可被删除
  + 该功能在6.0版本之后才提供

#### 获取文件锁定状态

* **接口定义**

int32\_t efsGetFileLockStat(EFSHandle handle, const char\* ilename);

* **功能描述**

获取文件锁定状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | **[IN] EFS的操作句柄** |
| fileName | [IN ] 文件名 |

* **接口返回值**
  + 1：文件已锁定。
  + 0：文件未锁定。
  + -1：获取状态失败。可以调用efsGetLastError ()获取错误码
* **示例**

|  |
| --- |
| // 指定的文件名，必须包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 获取文件锁定状态  int32\_t state = efsGetFileLockStat( efs, filename );  if( 1 == state ){  // 文件已锁定  printf( “file locked\n” );  }else if( 0 == state ){  // 文件未锁定  printf( “file not lock\n” );  }else if( -1 == state ){  // 获取失败  printf( “get lock state failed, error:%d\n”, efsGetLastError() );  }else{  // 无其他错误码  assert( 0 );  } |

* **提示**
  + 该功能在6.0版本之后才提供

#### 重命名文件

* **接口定义**

EFS\_BOOL efsRenameFile( EFSHandle handle,

const char\* oldFileName,

const char\* newFileName );

* **功能描述**

**重命名文件名。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] EFS的操作句柄** |
| oldFileName | **[IN] 旧文件名字，**该文件名的组成是bucket名+’/’+真实文件名 |
| newFileName | **[IN] 新文件名字，**该文件名的组成是bucket名+’/’+真实文件名 |

* **接口返回值**

1. true：重命名成功。
2. false：重名失败，可以调用getLastError()获取错误码。

* **示例**

|  |
| --- |
| // 指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* old\_filename = “bucket1/helloworld”;  // 新文件名的Bucket必须与老文件名的Bucket一致  const char\* new\_filename = “bucket1/byeworld”;  if( efsRenameFile( efs, old\_filename, new\_filename ) ){  printf( “rename file\n” );  }else{  printf( “rename file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 新文件名中的bucket必须与老文件名中的bucket一致，即不允许跨bucket重命名
  + 重命名时必须保证文件已经关闭

#### 监听文件状态

* **接口定义**

EFS\_BOOL efsSetFileStateCallBack(EFSFileHandle handle, fileStateCallBack cb);

* **功能描述**

监听文件状态。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| listener | [IN ] 文件监听者，为NULL时表示删除监听。  IFileListener内部有一个虚函数  virtual void onFileStateChanged(CFile\* file, FileState state)，其中file为文件对象，state表示该文件对象的状态。  enum FileState {  fileStateInit, //初始状态  fileStateNormal, //文件正常  fileStateWarning, //文件有风险  fileStateException //文件异常  }; |

* **接口返回值**
  + EFS\_TRUE：设置成功
  + EFS\_FALSE：设置失败
* **示例**

|  |
| --- |
| // 状态处理函数  void onFileStateChanged( EFSFileHandle file, EFSFileState state )  {  // 可根据文件状态进行处理，常规只处理Dahua::EFS::fileStateWarning状态  printf( “file state is “ << state );  if( Dahua::EFS::fileStateWarning == state ){  // 该状态说明N+M写入时，小于等于M个Block写入异常，  // 为保证数据冗余度，可设置关闭文件的标识，  // 写入线程检查该标识，关闭文件并重新创建文件写入  // 无特殊需求不建议使用该功能  }  return;  }  // 创建时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 以4+1模式创建大文件  EFSFileHandle file = efsCreateFile( efs, filename, 4, 1, EFS\_TRUE );  // 必须检查返回的CFile对象是否有效，无效说明创建失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 监听文件状态  if( EFS\_TRUE == efsSetFileStateCallBack( file, onFileStateChanged ) ){  printf( “set file listener\n” );  }else{  // 一般不会失败  }    // 对文件进行操作,如果在此过程中文件状态出现变化，会调用onFileStateChanged()    // 不再监听文件状态  if( EFS\_TRUE == efsSetFileStateCallBack( file, NULL ) ){  printf( “set file listener to NULL\n” );  }else{  // 一般不会失败  }    efsCloseFile( file );  }else{  printf( “create file failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 不能直接在onFileStateChanged()中关闭文件，否则可能导致异常
  + 不论读写都可监听文件状态变化
  + 7.2版本之前对一个File重复设置监听时，内部使用最后一次设置的监听。7.2版本开始支持设置多个监听对象，但设置为NULL时，会取消全部监听。
  + 除非确实需要对fileStateWarning状态进行处理，否则一般情况下，只需根据CFile对象的接口返回值进行业务处理。
  + 由于是异步回调，请注意锁的使用，避免死锁

#### 获取文件属性

* **接口定义**

**EFS\_BOOL efsGetFileStat( EFSFileHandle handle, EFSFileStat\* stat );**

* **功能描述**

**获取指定文件的文件属性。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] File的操作句柄** |
| **stat** | **[OUT] 文件属性**  **typedef struct \_\_EFSFileStat**  **{**  **uint8\_t mode; ///<读写模式**  **uint8\_t isAppend; ///<是否正在被写入 1是 0否**  **uint8\_t dataNum; ///<数据块个数N**  **uint8\_t parityNum; ///<冗余块个数M**  **uint32\_t cTime; ///<创建时间**  **uint32\_t mTime; ///<最后修改时间**  **uint64\_t totalSize; ///<文件大小，以字节为单位**  **char reserved2[108]; ///<保留**  **}EFSFileStat;** |

* **接口返回值**
  + EFS\_TRUE：成功
  + EFS\_FALSE：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的file是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 获取文件状态  EFSFileStat s;  if( EFS\_TRUE == efsGetFileStat( file, &s ) ){  printf( “get file stat:\n” );  printf( “mode:%d\n”, s.mode );  printf( “writing ? %s”, 0 == s.isAppend ? “false” : “true” );  printf( “N+M is %u+%u\n”, s.dataNum, s.parityNum );  printf( “ctime: %u\n”, s.cTime );  printf( “mtime: %u\n”, s.mTime );  printf( “size: %”PRIu64”\n”, s.totalSize );  }else{  printf( “get file stat failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “open file failed, error:%d\n”, efsGetLastError() );  } |

#### 设置文件选项

* **接口定义**

EFS\_BOOL efsSetFileOption(EFSFileHandle handle,

EFSOption key,

const void\* value,

uint32\_t len);

* **功能描述**

**设置CFile高级配置信息。**

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **handle** | **[IN] File的操作句柄** |
| key | [IN ]系统配置选项。  enum EFSOption  {  efsRwMode //读写模式(0异步1同步，默认为0)  }; |
| value | [IN ]对应配置项的值。 |
| len | [IN]对应配置项value缓冲区的长度。单位byte。 |

* **接口返回值**
  + EFS\_TRUE：成功
  + EFS\_FALSE：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的file是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 设置为同步阻塞式读取（默认为非阻塞读取）  int option = 1;  if( EFS\_TRUE == efsSetFileOption( file, efsRwMode, &option, sizeof( option ) ) ){  printf( “set file option\n” );  }else{  printf( “set file option failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “open file failed, error:%d\n”, efsGetLastError() );  } |

* 提示
  + 一般情况下无需设置该属性

#### 获取文件选项

* **接口定义**

EFS\_BOOL efsGetFileOption(EFSFileHandle handle,

EFSOption key,

void\* value,

uint32\_t len);

* **功能描述**

获取CFile高级配置信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **handle** | **[IN] File的操作句柄** |
| key | [IN ]系统配置选项。  enum EFSOption  {  efsRwMode //读写模式(0异步1同步)  }; |
| value | [OUT ]获取到对应配置项的值。 |
| len | [IN]获取对应配置项value缓冲区的长度。 |

* **接口返回值**
  + EFS\_TRUE：成功
  + EFS\_FALSE：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的file是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 设置为同步阻塞式读取（默认为非阻塞读取）  int option;  if( EFS\_TRUE == efsGetFileOption( file, efsRwMode, &option, sizeof( option ) ) ){  printf( “get file option\n” );  }else{  printf( “get file option failed, error:%d\n”, efsGetLastError() );  }  }else{  printf( “open file failed, error:%d\n”, efsGetLastError() );  } |

#### 修改文件偏移

* **接口定义**

**EFS\_BOOL efsSeek( EFSFileHandle handle, int64\_t offset, EFSFileLocation whence );**

* **功能描述**

**根据指定基准位置定位到指定偏移。**

* **接口参数**

|  |  |
| --- | --- |
| **参数名** | **含义** |
| **handle** | **[IN] File的操作句柄** |
| **offset** | **[IN] 相对于whence位置后在文件中的偏移** |
| **whence** | **[IN] 偏移的基准位置**  **typedef enum \_\_EFSFileLocation**  **{**  **efsBegin, ///<开始位置**  **efsCurrent, ///<当前位置**  **efsEnd ///<文件尾**  **}EFSFileLocation;** |

* **接口返回值**
  + EFS\_TRUE：成功
  + EFS\_FALSE：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的file是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 修改偏移  if( EFS\_TRUE == efsSeek( file, 100, efsBegin ) ){  printf( “seek to 100B\n” );  }else{  printf( “file seek failed, error:%d\n”, efsGetLastError() );  }  efsCloseFile( file );  }else{  printf( “open file failed, error:%d”, efsGetLastError() );  } |

* **提示**
  + 结合efsRead()可实现随机读取
  + 创建写**和追加写模式的File不支持调用该接口，即不支持随机写入**

#### 获取文件偏移

* **接口定义**

int64\_t efsTell( EFSFileHandle handle );

* **功能描述**

**获取文件当前位置。**

* **接口返回值**
  + >= 0：当前读到的位置相对于文件头的偏移。
  + < 0：获取失败。可以调用efsGetLastError ()获取错误码
* **示例**

|  |
| --- |
| // 打开时指定的文件名，必须是包含有bucket名字，注意要有斜杠’/’分割  const char\* filename = “bucket1/helloworld”;  // 读打开文件  EFSFileHandle file = efsOpenFile( efs, filename, fileModeRead );  // 必须检查返回的file是否有效，无效说明打开失败了  if( EFS\_TRUE == isEFSFileHandleValid( file ) ){  // 获取文件当前偏移  int64\_t offset = efsTell( file );  if( 0 <= offset ){  printf( “current offset is %”PRIi64”.\n”, offset );  }else{  printf( “get file current offset failed, error:%d\n”, efsGetLastError() );  }  efsCloseFile( file );  }else{  printf( “open file failed, error:%d”, efsGetLastError() );  } |

#### 分页查询Bucket下的文件名

* **接口定义**

**int32\_t efslistBucketFile( EFSBucketHandle handle,**

**const char\* begin,**

**const char\* end,**

**uint32\_t maxNumber,**

**char buffer[][MAX\_FILE\_NAME\_LEN] );**

* **功能描述**

列出bucket内文件名落在(begin,end)范围的文件名列表。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **handle** | [IN] Bucket的操作句柄 |
| **begin** | [IN] 起始文件名。无需包含Bucket名。如果begin为NULL，则从Bucket下具备最小值的文件开始查询，且包含该最小值的文件。 |
| **end** | [IN] 结束文件名。无需包含Bucket名。如果end为NULL，则查询到该Bucket下具有最大值的文件。 |
| **maxNumber** | [IN] 查询的最大文件个数。取值范围[1, 2048] |
| **buffer** | [OUT] 存放获取到的文件名。 |

* **接口返回值**
  + >=0：获取到文件信息个数。
  + <0：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| typedef char (\*NameList)[MAX\_FILE\_NAME\_LEN];  const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  printf( “get bucket %s\n”, bucket\_name );  // 假设该Bucket下存在文件 a,b,c,d  NameList namelist = (NameList)malloc( 1024 \* MAX\_FILE\_NAME\_LEN );  char begin\_name[MAX\_FILE\_NAME\_LEN] = “b”;  char end\_name[MAX\_FILE\_NAME\_LEN] = “d”;  uint32\_t max\_num = 1024;  while( 1 ){  int32\_t size = efslistBucketFile( bucket, begin\_name, end\_name, max\_num, namelist );  if( 0 <= size ){  // 这里将列出文件c，d，查询到的文件名不包含Bucket名  for( int I = 0; I < size; ++I ){  printf( “%s\n”, namelist[i] );  }  if( max\_num > size ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else{  // 可能还有文件，修改begin\_name继续查询  assert( MAX\_FILE\_NAME\_LEN > strlen( namelist[size-1] ) );  strcpy( begin\_name, namelist[size-1] );  }  }else{  printf( “lsit file failed, error:%d”, efsGetLastError() );  break;  }  }  free( namelist );  namelist = NULL;  }else{  printf( “get bucket failed, error:%d”, efsGetLastError() );  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果，可能导致死循环。

#### 分页查询Bucket下的文件信息

* **接口定义**

int32\_t efsListBucketFileInfo( EFSBucketHandle handle,

const char\* begin,

const char\* end,

uint32\_t maxNumber,

EFSFileInfo\* pinfo );

* **功能描述**

列出bucket内文件名落在(begin,end)范围的文件信息。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| **handle** | [IN] Bucket的操作句柄 |
| **begin** | [IN] 起始文件名。无需包含Bucket名。如果begin为NULL，则从Bucket下具备最小值的文件开始查询，且包含该最小值的文件。 |
| **end** | [IN] 结束文件名。无需包含Bucket名。如果end为NULL，则查询到该Bucket下具有最大值的文件。 |
| **maxNumber** | [IN] 查询的最大文件个数。取值范围[1, 2048] |
| **pinfo** | [OUT] 存放获取到的文件信息。 |

* **接口返回值**
  + >=0：获取到文件信息个数。
  + <0：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  printf( “get bucket %s\n”, bucket\_name );  // 假设该Bucket下存在文件 a,b,c,d  EFSFileInfo\* filelist = (EFSFileInfo\*)malloc( 1024 \* sizeof(EFSFileInfo) );  assert( NULL != filelist );  char begin\_name[MAX\_FILE\_NAME\_LEN] = “b”;  char end\_name[MAX\_FILE\_NAME\_LEN] = “d”;  uint32\_t max\_num = 1024;  while( 1 ){  int32\_t size = efsListBucketFileInfo( bucket, begin\_name, end\_name, max\_num, filelist );  if( 0 <= size ){  // 这里将列出文件c，d，查询到的文件名不包含Bucket名  for( int I = 0; I < size; ++I ){  printf( “filename: %s\n”, filelist[i].name );  printf( “size: %”PRIu64”\n”, filelist[i].size );  printf( “create time: %u\n”, filelist[i].cTime );  printf( “modify time: %u\n”, filelist[i].mTime );  }  if( max\_num > size ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else{  // 可能还有文件，修改begin\_name继续查询  assert( MAX\_FILE\_NAME\_LEN > strlen( filelist[size-1].name ) );  strcpy( begin\_name, filelist[size-1].name );  }  }else{  printf( “list file failed, error:%d\n”, efsGetLastError() );  break;  }  }  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果，可能导致死循环。

#### 分页查询指定目录下的文件信息

* **接口定义**

int32\_t efsListBucketFileInfoByPrefix( EFSBucketHandle handle,

const char\* dir,

const char\* delimiter,

const char\* begin,

uint32\_t maxNumber,

EFSFileInfo\* pinfo );

* **功能描述**

查询bucket内dir目录下的文件信息

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] Bucket的操作句柄 |
| **dir** | [IN] 需要查找的文件夹路径，最后字符必须为’/’。无需包含Bucket名 |
| **delimiter** | [IN] 文件夹路径分隔符,目前仅支持’/’ |
| **begin** | [IN] 起始文件名。无需包含目录名。如果为NULL，则查询dir下所有文件信息 |
| **maxNumber** | [IN] 查找的最大文件数量。取值范围[1, 2048] |
| **pinfo** | [OUT] 存放获取到的文件信息 |

* **接口返回值**
  + >=0：获取到文件信息个数。
  + <0：失败。可以调用efsGetLastError()获取错误码
* **示例**

|  |
| --- |
| const char\* bucket\_name = “bucket\_test”; // bucket名必须符合命名规范  EFSBucketHandle bucket = efsGetBucket( efs, bucket\_name );  if( EFS\_TRUE == isEFSBucketHandleValid( bucket ) ){  printf( “get bucket %s”, bucket\_name );  /\*\*  \* 假设该Bucket下存在文件，a/, a/b, a/c/，a/c/d, a/e。可以认为其存在如下目录树  \* a/  \* |--  \* |--b  \* |--c/  \* | |--d  \* |--e  \*/  EFSFileInfo\* filelist = (EFSFileInfo\*)malloc( 1024 \* sizeof(EFSFileInfo) );  assert( NULL != filelist );  char dir[MAX\_FILE\_NAME\_LEN] = “a/”; // 必须以’/’结尾  char begin\_name[MAX\_FILE\_NAME\_LEN] = “b”;  uint32\_t max\_num = 1024;  while( 1 ){  // 列出a/目录下的文件，最后一个字符必须是’/’  // 分隔符只支持’/’  // begin是”b”，因此查询的结果中不会包含”a/b”  int32\_t size = efsListBucketFileInfoByPrefix( bucket, dir, “/”, begin\_name, max\_num, filelist );  if( 0 <= size ){  // 这里将列出文件c/,e。查询到的文件名不包含目录名  // 不会递归查询子目录  for( int I = 0; I < size; ++I ){  printf( “filename: %s\n”, filelist[i].name );  printf( “size: %”PRIu64”\n”, filelist[i].size );  printf( “create time: %u\n”, filelist[i].cTime );  printf( “modify time: %u\n”, filelist[i].mTime );  }  if( max\_num > size ){  // 实际查询到的文件数小于指定的查询的个数，说明已经查询结束  break;  }else{  // 可能还有文件，修改begin\_name继续查询  assert( MAX\_FILE\_NAME\_LEN > strlen( filelist[size-1].name ) );  strcpy( begin\_name, filelist[size-1].name );  }  }else{  printf( “lsit file failed, error:%d\n”, efsGetLastError() );  break;  }  }  }else{  printf( “get bucket failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 循环查找时，必须修改begin参数，否则总是返回第一次查询的结果，可能导致死循环。

### 获取系统信息

#### 获取系统资源信息

* **接口定义**

EFS\_BOOL efsGetSystemInfo( EFSHandle handle, EFSSystemInfo\* info );

* **功能描述**

获取当前EFileSystem的信息，包括集群节点数，客户端运行个数等。

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS的操作句柄 |
| Info | [OUT ]系统信息。  typedef struct \_\_EFSSystemInfo  {  uint32\_t totalNodes; ///<集群中的节点总数  uint32\_t totalClient; ///<当前运行的客户端总数  uint32\_t curWriteFiles; ///<当前正在写的文件数  uint64\_t totalFiles; ///<用户的文件总数  uint64\_t totalSpace; ///<用户的总存储空间，单位MB  uint64\_t spaceLeft; ///<用户的剩余存储空间，单位MB  char uuid[32]; ///<系统uuid,32个字符  uint64\_t spaceQuotaLeft; ///<用户剩余空间，单位MB  uint64\_t spaceUsed; ///<用户已使用容量, 单位MB  char reserved[44]; ///<保留  }EFSSystemInfo; |

* **接口返回值**
  + EFS\_TRUE：获取成功。
  + EFS\_FALSE：获取失败。可以调用efsGetLastError()获取错误码。
* **示例**

|  |
| --- |
| EFSSystemInfo info;  if( EFS\_TRUE == efsGetSystemInfo( efs, &info ) ){  // 以下为所有用户的统计信息  printf( “system uuid: %s\n”, info.uuid ); // 系统唯一标识  printf( “total nodes: %u\n”, info.totalNodes ); // 系统中的总节点数  printf( “total clients: %u\n”, info.totalClient ); // 系统当前正运行的客户端  printf( “current writing files: %u\n”, info.curWriteFiles ); // 系统当前正在写的文件总数  // 以下为当前用户的信息，请注意在系统实际容量不足时，spaceLeft+spaceUsed不一定等于totalSpace  printf( “total files: %”PRIu64”\n”, info.totalFiles ); // 当前用户的文件总数  printf( “total space: %”PRIu64”\n”, info.totalSpace ); // 当前用户的总空间  printf( “space left: %”PRIu64”\n”, info.spaceLeft ); // 用户真正可用的剩余空间，可能是系统实际剩余空间  printf( “space quota left: %”PRIu64”\n”, info.spaceQuotaLeft ); // 剩余的可分配空间，即可以给Bucket的设定的最大配额  printf( “used space: %”PRIu64”\n”,info.spaceUsed ); // 用户已经使用了的空间  }else{  printf( “get system info failed, error:%d\n”, efsGetLastError() );  } |

* **提示**
  + 其中totalFiles并非大文件个数和小文件个数的总和，而是大文件个数和container个数的总和，而一个container可能包含多个小文件。请不要在类似“当文件数达到一定值时，开始删除文件“这样的逻辑中依赖该数值。

#### 获取系统支持的N+M

* **接口定义**

int32\_t efsGetRedundanceCaps( EFSHandle handle ,

EFSRedundanceCap\* caps,

uint32\_t size );

* **功能描述**

获取系统支持的N+M

* **接口参数**

|  |  |
| --- | --- |
| 参数名 | 含义 |
| handle | [IN] EFS的操作句柄 |
| caps | [OUT ] 保存N+M信息的空间 |
| size | [IN] caps的大小 |

* **接口返回值**
  + >=0：获取到的N+M的个数。
  + -1：失败。可以调用getLastError()获取错误码。
  + -2：caps的大小太小，不足以保存所有的N+M信息
* **示例**

|  |
| --- |
| // 分配内存保存N+M信息  int cap\_num = 10;  do{  EFSRedundanceCap\* caps = (EFSRedundanceCap\*)malloc( cap\_num \* sizeof( EFSRedundanceCap ) );  assert( NULL != caps );  int32\_t num = efsGetRedundanceCaps( efs, caps, cap\_num );  if( 0 <= num ){  printf( “support:\n” );  for( int32\_t I = 0; I < num; ++I ){  printf( “%u+%u\n”, caps[i].dataNum, caps[i].parityNum );  }  break;  }else if( -1 == num ){  printf( “get n+m failed, error:%d\n”, efsGetLastError() );  break;  }else if( -2 == num ){  printf( “caps is too small\n” );  // 空间不够大，倍增后继续获取  cap\_num \*= 2;  free( caps );  caps = NULL;  }  }while( 1 ); |

* **提示**
  + 可根据该接口获取N+M模式创建文件。

## 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，lib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EFSAdapter.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。

# 最佳实践

待补充

# 常见问题

## SDK问题定位流程

1. 查看SDK日志，日志路径默认在程序执行路径的log目录下，日志名称是SDK\_log\_current.log
2. 查看日志里面的错误码，然后再根据[错误码对照表](#错误码对照表)查看错误码的含义，根据错误码的含义就能够定位简单的SDK问题

## 初始化EFS失败

初始化EFS失败，一般是因为

* 服务IP不对
* 用户名密码不对，错误码为-60000或-60001，

请仔细检查初始化时的配置信息是否正确。如果正确可先登陆运维界面查看服务是否正常。如果服务正常，请使用ping检查与服务IP的网络是否正常。

## 创建文件失败，错误码为-40003

-40003代表系统不支持创建文件时指定的N+M。在未开启N+M:0时，必须保证系统中在线且能提供服务的存储节点个数大于等于N+M。

## 读写失败，错误码为-60012

-60012表示用户设置的bucket容量大于用户剩余的容量，需要到云存储运维上面增加用户的容量

## 写文件失败，错误码为-70007

这种情况一般出现在开启了N+M:0的环境中，因为系统中存储节点的磁盘个数小于N+M，所以写入失败。

## 写入了数据，但无法读取到这部分数据

由于SDK内部需要计算EC，只有当写入的数据达到一个条带大小时，才会真正写入存储（大文件的条带大小为32K\*N；小文件的条带大小为4K\*N，其中N为N+M的N）。对于大文件来说，一旦数据写入存储，就可以立即读取，但是对于小文件来说只有文件关闭后，才能读取。

根据以往项目经验，客户在写视频索引文件时，该实现机制带来的影响比较大。可以以下两种方式来规避：

* 以大文件模式创建视频文件，并将视频索引数据追加在视频帧数据尾部。播放时从尾部开始读取将以魔术字匹配到索引区域，可根据该索引区域记录的前驱索引偏移，找到其前驱索引区域，以此可找到所有索引。推荐该模式，可以减少文件数且不会增加EFS系统的压力
* 以小文件模式创建视频索引文件A，每次写入索引数据后，立即关闭文件A。当需要再次更新索引时，将文件A中的数据读取到内存，在内存中拼接新索引数据，然后将这些数据写入文件B中，最后将A删除，并将B重名为A。

## 写文件时发现性能很差，带宽上不去

当前性能测试结果显示SDK本身是不存在瓶颈的，完全可以跑满带宽。不过需要注意的是因为SDK以N+M写时，会有M份校验数据，所以带宽打满之后，对于业务层来说，其写入速度=当前带宽 \* N/N+M，以3+1为例，当实际带宽为400MB/s时，业务层的写入速度应在300MB/s左右。

如果带宽无法跑满，请使用iperf等工具检测一下客户机到云存储所有节点（特别是存储节点）的实际带宽。

PS: 测试客户机为4个千兆网卡以xor模式bind，8核CPU

## 删除了文件，但是获取系统资源信息时容量没有变化

当调用SDK删除文件后，EFS服务端实际会延时批量删除文件，以避免大批量删除文件时影响业务IO，因此系统资源信息中的容量并不会在删除文件成功后立即变化。请注意不要依赖该接口获取的容量信息来做循环覆盖功能，这可能导致多删除文件。请及时升级到新版本，使用紧急覆盖和Bucket生命周期等功能。

## 创建了一个文件，只写了几个字节，但是获取系统资源信息时发现容量少了好几百MB

只创建文件不会占用系统容量，但是如果开始写数据，那么系统默认会预分配一个Object的空间，一个Object的大小为(N+M)\*128MB，以4+1算，就是640MB。当文件关闭后，会重新计算该文件的实际占用空间。

# 历史版本BUG

## 查看SDK版本号的方法：

nm -a libEFSClientCore64.so | grep EFS\_SDK

查看结果类似：EFS\_SDK\_1\_72\_2\_123456，其中72表示7.2版本的，123456表示svn号

## V1.072.0000004.4.R

|  |  |  |
| --- | --- | --- |
| **问题描述** | **问题现象** | **修复版本** |
| 网络框架线程阻塞 | 读写异常，使用SDK的进程出现大量CLOSE\_WAIT的socket。 | V1.072.0000004.5.R |

## V1.060.0000002.4.R

|  |  |  |
| --- | --- | --- |
| **问题描述** | **问题现象** | **修复版本** |
| 网络框架线程池销毁耗时长 | 进程启动初始化SDK时耗时超过5秒，已初始化SDK的进程退出时耗时超过5秒。  对于需要间隔重启的服务来说，会有比较严重的影响，比如apache的http服务。 | V1.072.0000004.4.R |
|  |  |  |

# 错误码对照表

