**Sstable的存储结构与SstableWriter接口**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **版本** | **修订章节** | **修订原因** | **修订日期** | **修订人** |
| **1** | 1.0 | 全文 | 新建 | 2010-9-30 | 华庭 & 正祥 |
| **2** | 2.0 |  | 增加对column group的支持，多表时的block，schema的扁平化，bloomfilter增加对列的记录 | 2011-3-16 | 华庭 & 正祥 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

## Sstable的存储结构

简单易于扩展的Sstable存储结构便于管理，不容易出错，Oceanbase系统中尽量简化Sstable的存储结构，并充分考虑到Sstable的只读特性和磁盘顺序读写相对性能比较高，SstableWriter通过顺序写来创建Sstable文件，Sstable中的数据以block为存储单位，便于压缩，校验和索引。

### 1.1 总体存储结构

|  |  |
| --- | --- |
| 0 | Block-0 |
|  | Block-1 |
|  | Block-2 |
|  | Block-3 |
|  | … |
|  | Block index |
|  | Bloom filter |
|  | ObTableSchema |
|  | ObSstableTrailer(非压缩) |
|  | ObSSTableTrailerOffset |

所有block从SSTable文件头开始顺序存放。除了最后一行ObSSTableTrailerOffset，上图中每一行都是sstable文件的一个记录(record)，存储时会对每个record计算checksum，并将该checksum存在record的头部中。ObSSTableTrailerOffset为8个字节，存储于Sstable文件的最后(网络字节序)，指示了ObSstableTrailer记录在文件中的相对于文件头的偏移值。

### 1.2 记录结构：

|  |  |
| --- | --- |
| record头 | 实际数据(PayLoad) |

每个record包括两部分，记录头和实际的数据(PayLoad)。记录头包含了record的长度和校验信息等，之后紧跟着记录的实际数据(PayLoad)。各种PayLoad有自身的格式，也可能有自身的头信息等，例如每个SSTable Block就是一个记录，其PayLoad中有SSTable头、行数据、行索引等信息。

#### 1.2.1 记录头格式：

struct ObRecordHeader

{

int16\_t magic\_; *// magic number*

int16\_t header\_length\_; *// header长度*

int16\_t version\_; *// 版本号*

int16\_t header\_checksum\_;*// header checksum*

int64\_t reserved\_; *// 保留，必须置0*

int32\_t data\_length\_; *// 压缩前的数据长度*

int32\_t data\_zlength\_; *// 磁盘上的数据长度,未压缩时等于压缩前的数据长度*

*// 压缩时为压缩后数据长度，data\_zlength <= data\_length*

*// 如果二者相等，表明数据没有压缩*

int64\_t data\_checksum\_; *// 写入磁盘中数据的checksum，不包含记录头*

};

header\_checksum采用简单的奇偶校验，data\_checksum采用CRC64算法。

**该部分数据不经过序列化存储，每个项以网络序保存。**

#### 1.2.2 SSTable Block存储结构

每个SSTable Block是一个记录，其PayLoad包含了SSTableBlock头信息、排序行数据以及行索引信息等，如下图所示：

|  |
| --- |
| SSTableBlockHeader |
| 行数据[0] |
| 行数据[1] |
| …… |
| 行数据[row\_count\_-1] |
| 行索引数组[row\_count\_+1] |

**SSTableBlockHeader**定义：

struct ObSSTableBlockheader

{

int32\_t row\_index\_array\_offset\_; *// 行索引数组相对PayLoad起始点的偏移值*

int32\_t row\_count\_; *// 该block中行数*

int16\_t reserved16\_; *//保留字段，必须为0，v2.0修改*

uint16\_t column\_group\_id\_; *// column\_group\_id，v2.0修改*

uint32\_t table\_id\_; *// table id*

};

其中column\_group\_id默认值为0。

**该部分数据不经过序列化存储，每个项以网络序保存。**

**行数据**格式：

SSTableBlock的行数据有三种格式：**稠密格式(dense)**、**稀疏格式(sparse)**和**混合格式(mixed)**，此处只定义了前两种，即稠密格式和稀疏格式，混合格式留待日后定义。

**稠密格式(dense)**的定义：

rowkey + ObObjs[ ]

其中rowkey由rowkey\_length(2字节，存储时为网络字节序) + rowkey字符串组成，ObObjs[]为该行全部列的值(value)，其数量、顺序和类型由表的schema定义，包括空值对象，但不包含列名(column name)或列名ID。ObObjs[]序列化后存储。

**稀疏格式(sparse)**的定义：

rowkey + (column\_id + ObObj)[ ]

其中rowkey由rowkey\_length(2字节，存储时为网络字节序) + rowkey字符串组成，(column\_id + ObObj)[ ]数组是该行有值或有修改的列，其类型需要与表的schema一致。column\_id和ObObj都序列化后存储。

**行索引数据(row\_index\_array\_)**格式：

row\_index\_array\_[row\_count\_+1]是row\_count\_个元素的数组，每个元素是一个int32\_t，记录了每行行首相对于该SSTableBlock的PayLoad起始点的相对偏移值。最后一个元素指向最后一行的尾部。

**该部分数据不经过序列化存储，每个项以网络序保存。**

**注释：**

1. **多数时候(例如ChunkServer上)一个sstable只包含一个table，但用于存储UpdateServer的冻结表时，可能包含多个表；**
2. **每个table可以有若干个column group，每个column group包含若干列；**
3. **每个column group内数据按行存储，column group之间的存储相互独立；**
4. **为0的column\_group\_id是特殊的column group，用于代表用户定义的column group之外的所有列的column group，为1的column\_group\_id保留，>=2的column\_group\_id是用户可用的id；**
5. **每个block不能包含多个table或者一个table的多个column group的数据，一个column group的一行的数据须在一个block内，不能跨越两个或多个block；**
6. **一个table的一个column group的数据按row key排序存储在一串连续的block中；**
7. **一个sstable内的所有block按(table\_id，column\_group\_id，block\_end\_key)排序并连续存储；**

#### 1.2.4 Sstable block索引存储结构

通常情况下Sstable block索引是一个record，其PayLoad由头部信息、索引数组、end key char stream组成：

|  |
| --- |
| SSTableBlockIndexHeader |
| BlockIndexArray[sstable\_block\_count\_] |
| end key char stream |

**SSTableBlockIndexHeader**：

|  |
| --- |
| int64\_t sstable\_block\_count\_ |
| int32\_t end\_key\_char\_stream\_offset\_; |
| int32\_t reserved32\_; |
| int64\_t reserved64\_[2]; |

其中sstable\_block\_count\_是该record中block索引的数量，它通常与该SSTable中全部block的数量相同，end\_key\_char\_stream\_offset\_是该record中全体block的end key的char stream相对于payload头的偏移值。**全部block索引按(table\_id，column\_group\_id，block\_end\_key)排序**。

如果全部block索引接近或超过2GB，不能放到一个record中，可以拆分成几个相邻存放的record，每个record中包含一定数量的block索引和对应的end\_key\_char\_stream，且保持全部block索引的顺序。

**该部分数据不经过序列化存储，每个项以网络序保存。**

**BlockIndexArray[sstable\_block\_count\_]**的每个元素定义为：

struct

{

int16\_t reserved16\_; *//保留字段，必须为0，v2.0修改*

uint16\_t column\_group\_id\_; *//column\_group\_id，v2.0修改*

uint32\_t table\_id\_; *//table id*

int32\_t block\_record\_size\_; *//该block记录的尺寸(包含记录头)*

int16\_t block\_end\_key\_size\_; *//对应end key的尺寸*

int16\_t reserved\_; *//保留字段，必须置0*

};

其中block\_record\_size\_是对应block记录尺寸，所有Block在SSTable文件中按**(table\_id，column\_group\_id，block\_end\_key)**顺序存放，block\_end\_key\_size\_是对应block的end\_key尺寸。

应用层需要按照（(table\_id, column\_group\_id, row\_key)的顺序逐行写入sstable文件。

**该部分数据不经过序列化存储，每个项以网络序保存。**

**end key char stream**：

存放了每个块的end key字符串的流，所有的end key按顺序存放，中间没有分隔符等。

#### 1.2.5 Bloom filter存储结构

在这里只存储了布隆过滤器的bitmap数据，布隆过滤器的元数据信息记录在trailer中，比如hash\_count, filter\_items\_actual。如果sstable中包含多个table，则所有table共享一个bloom filter：

|  |  |
| --- | --- |
| 记录头 | bitmap |

其中bitmap只是binary数据，不需要调用序列化方法的。

**注意：由于一个sstable中可能包含多个table，每个table有多个column group，因此bloom filter的查询关键字为(16-bit-reserved，16-bit-column\_group\_id， 32-bit-table\_id，row key)以及 (16-bit-reserved，16-bit-column\_group\_id， 32-bit-table\_id，row key，32-bit-column\_id)，这个与v 1.0不同。16-bit-reserved为保留项，须置0。**

#### 1.2.6 Schema存储结构

Schema的序列化也是仅仅调用ObSSTableSchema的序列化接口，其存储结构如下图所示：

|  |  |
| --- | --- |
| 记录头 | ObSSTableSchema |

ObTableSchema头部ObSchemaHeader和ObSchemaColumnDef[]组成：

ObSchemaHeader：

struct

{

int16\_t column\_count\_; *//为了兼容v1.0，今后将废弃*

int16\_t reserved16\_; *//保留字段，必须置0*

int32\_t total\_column\_count\_; *//该sstable中所有表的column的总数，v2.0修改*

};

**该部分数据不经过序列化存储，每个项以网络序保存。**

ObSchemaColumnDef[column\_count\_]每项定义：

struct {

int16\_t reserved16\_; *//保留字段，必须置0，v2.0修改*

uint16\_t column\_group\_id\_; *//column group的ID，v2.0修改*

uint32\_t column\_name\_id\_; *//column name的ID*

int32\_t column\_value\_type\_; *//column value的ObObj的类型*

uint32\_t table\_id\_； *//table id，v2.0修改*

};

**所有table的schema存储在一起，整个数组按(table\_id，column\_group\_id，column\_name\_id)顺序存储。**

**该部分数据不经过序列化存储，每个项以网络序保存。**

### 1.3 trailer存储结构

Trailer存储结构包括两部分：

1. ObSstableTrailer，实际的trailer。

2. ObTrailerOffset，标示了trailer的在Sstable文件中的偏移。

这两个类都提供了相应的序列化反序列化方法，调用这些方法可以实现trailer的序列化和反序列化。

#### 1.3.1 ObSstableTrailer存储结构

ObSstableTrailer也是一个记录：

|  |  |
| --- | --- |
| 记录头 | ObSstableTrailer |

//此结构在V2.0中不再使用

**struct** ObTableTrailerInfo {

uint64\_t table\_id; //table id

int16\_t column\_count\_; //schema中列数量

int16\_t start\_row\_key\_length\_; //start key长度

int16\_t end\_row\_key\_length\_; //end key长度

int16\_t reversed\_; //保留字段，0

};

ObSstableTrailer定义为：

**struct** ObSstableTrailer

{

int32\_t size\_; // ObSstableTrailer的尺寸

int32\_t trailer\_version\_; //该结构的版本号，对v2.0为0x200

int64\_t table\_version\_; //table的版本号

int64\_t first\_sstable\_block\_data\_offset\_; //SStable第一个block在文件总的偏移地址，

//当一个sstale占用一个物理文件，其值为0，当多

//个sstable共用一个物理文件时，其值不一定为0

int64\_t block\_count\_; //v2.0修改，SSTable中block索引数组大小

int64\_t block\_index\_record\_offset\_; //block索引记录的偏移值

int64\_t block\_index\_record\_size\_; //block索引记录的尺寸(包含记录头)

int64\_t bloom\_filter\_hash\_count\_;

int64\_t bloom\_filter\_record\_offset\_; //bloom filter记录的偏移值

int64\_t bloom\_filter\_record\_size\_; //bloom filter记录的尺寸(包含记录头)

int64\_t schema\_record\_offset\_; //schema记录的偏移值

int64\_t schema\_record\_size\_; //schema记录的尺寸(包含记录头)

int64\_t key\_stream\_record\_offset\_; //key stream记录(table的start/end key)的偏移值

int64\_t key\_stream\_record\_size\_; //key stream记录的尺寸(包含记录头)

int64\_t blocksize\_;//块尺寸

int64\_t row\_count\_;//基于table的总行数，只要某行在某个column group中有内容就算一行

uint64\_t sstable\_checkesum\_; //sstable文件的校验和

uint64\_t first\_table\_id\_; //sstable文件中第一个table的id，用于schema cache key

int64\_t frozen\_time\_; //ups冻结大版本的时间

int64\_t reserved64\_[reserved64\_num\_]; //必须置0

**char** compressor\_name\_[ MAX\_COMPRESSION\_NAME\_SIZE];

int16\_t row\_value\_store\_style\_; //稠密(dense)，稀疏(sparse)或者混合(mixed)格式

int16\_t reserved\_; //保留项，必须置0

int32\_t table\_count\_; //table数量

ObTableTrailerInfo table\_info\_[1]； //v2.0不再使用，仅为了v1.0兼容

//以下变量用于管理内存中的key buffer，不会序列化和反序列化

char\* key\_buf\_; //key buffer 存放table的 start key 和 end key

int64\_t key\_buf\_size\_; //key buffer 大小

int64\_t key\_data\_size\_; //实际 key data 长度

bool own\_key\_buf\_; //key buffer的内存是否是trailer分配的

}

**该部分数据不经过序列化存储，每个项以网络序保存。**

#### 1.3.2 ObTrailerOffset存储结构

为了使得ObSstableTrailer能够被访问，SSTable文件的最后8字节存放了ObSstableTrailer记录在SSTable文件中的偏移值：

int64\_t trailer\_record\_offset\_; *//ObSstableTrailer记录在sstble文件中的偏移*

trailer\_record\_offset\_按网络字节序存储。

## 2．SstableWriter接口

SSTableWriter负责生成数据，上层应用将按rowkey排好序的行append到SSTableWriter中，SSTableWriter将数据按约定好的格式生成索引、bloom filter以及SSTable的元信息并写入磁盘。

SSTableWriter 会在两种情况下被调用，一是每日合并子系统会利用SSTableWriter生成新的合并后的SSTable file,另一个是初次导入数据时，外部工具提供SSTableWriter提供的接口生成初始SSTable file文件。

SstableWriter

BlockBuilder

CheckSum

Schema

Compress

BloomFilter

IndexBuilder

Trailer

TrialerInfo

BlockBuilder负责在内存中创建block数据并序列化数据到磁盘。

Checksum负责对压缩前的数据计算checksum。

Schema负责序列化Schema到磁盘。

TrailerInfo负责将TrailerInfo序列化到磁盘。

Trailer负责将Trailer序列化到磁盘。

IndexBuilder负责将block的index序列化到磁盘。

Bloomfilter负责布隆过滤器的序列化。

Compress 负责数据的压缩。

### 2.1 BlockBuilder

#### 2.1.1 模块描述

BlockBuilder用于在内存中创建block数据，上层应用将按rowkey排好序的行append到BlockBuilder中，每向BlockBuilder添加一行数据，就会被序列化好的BlockBuilder中的buffer中，当block大小大于blocksize时，立即将BlockBuilder中的数据存储到磁盘，并清空BlockBuilder中的数据。

BlockBuilder中有一个静态buffer，用于存储序列化后的key/value对，BlockBuilder还有一个用于存储index的静态数组，如果buffer写满或是存储index的数组溢出，BlockBuilder会立即把数据写入磁盘。

为了便于精确查找block中的某个key，BlockBuilder还负责存储key在block中index。

#### 2.1.2 模块接口

class ObBlockBuilder

{

public:

*/\*\**

*\* 增加一行并序列化到buffer\_中*

*\*/*

int **add\_row**(ObRow &row);

private:

StaticBuffer key\_offset\_;

StaticBuffer buffer\_;

int32\_t key\_index\_offset\_;

};

### 2.2 IndexBuilder

#### 2.2.1模块描述

为便于查找，SSTable会建立一些信息来索引block数据，比如相关key在SSTable中的偏移，IndexBuilder即用来建立这些信息。

Index 的生成和使用属于两个比较独立的模块，为了简单，将其分为两个部分。IndexBuilder专门用来生成index。

###### 模块接口

class ObIndexBuilder

{

public:

*/\*\**

*\* 增加一项index，将key拷贝到variable\_中，将offset序列化到fixed\_中*

*\*/*

int **add\_entry**(ObString &key, int32\_t offset);

};

### 2.3 CheckSum

#### 2.3.1 模块描述

Checksum模块 用于计算block的checksum，我们希望Sstable中存储的数据能够自校验，保证数据的正确性，当checksum不一致时，意味着程序编码有问题或是磁盘出问题了。根据不同的情况采取不同的措施。在OceanBase系统中，暂时采用CRC64算法计算checksum。

#### 2.3.2 模块接口

class ObCheckSum

{

public:

*/\*\* Compute crc64 checksum*

*\**

*\** **@param** *data - input data*

*\** **@param** *len - input data length in bytes*

*\*/*

int64\_t **crc64**(const char \*data, int64\_t len);

};

### 2.4 Schema

#### 2.4.1 模块描述

由于每个Sstable需要拥有自己的Schema，在存储row时没有存储column name或是column id，所以需要根据Sstable的Schema来解析row。在SstableWriter中，Schema类主要负责自己的序列化，提供自己的序列化方法。

#### 2.4.2 模块接口

class ObTableSchema

{

public:

*/\*\**

*\* 序列化*

*\*/*

int **serialize**(char\* buf, const int64\_t buf\_len, const int64\_t& pos) const;

};

### 2.5 TrailerInfo

#### 2.5.1 模块描述

为了便于trailer扩展，特将TrailerInfo存储在Sstable文件的最末端来标示trailer的偏移地址。TrailerInfo自带序列化方法，采用非压缩存储。

#### 2.5.2 模块接口

class ObTrailerInfo

{

public:

*/\*\**

*\* 序列化*

*\*/*

int **serialize**(char\* buf, const int64\_t buf\_len, const int64\_t& pos) const;

private:

int64\_t trailer\_offset\_;

}

### 2.6 Trailer

#### 2.6.1 模块描述

Trailer主要用于存储Sstable的元数据，存储在Sstable文件的尾部，TrailerInfo之前，采用非压缩存储。

#### 2.6.2 模块接口

**struct** ObTableTrailerInfo {

uint64\_t table\_id; //table id

int16\_t column\_count\_; //schema中列数量

int16\_t start\_row\_key\_length\_; //start key长度

int16\_t end\_row\_key\_length\_; //end key长度

int16\_t reversed\_; //保留字段，0

//以下变量用于管理内存中的key在key buffer中的偏移地址，不会序列化和反序列化

int32\_t start\_row\_key\_offset\_; //start key在key buffer中的偏移地址

int32\_t end\_row\_key\_offset\_; //end key在key buffer中的偏移地址

};

class ObSstableTrailer

{

public:

int **serialize**(char\* buf, const int64\_t buf\_len, int64\_t& pos) const ;

int **deserialize**(const char\* buf, const int64\_t data\_len, int64\_t& pos) ;

int64\_t **get\_serialize\_size**(void) const;

private:

int32\_t size\_; // ObSstableTrailer的尺寸

int32\_t trailer\_version\_; //该结构的版本号

int64\_t table\_version\_; //table的版本号

int64\_t first\_sstable\_block\_data \_offset\_;//SStable第一个block在文件总的偏移地址，

//当一个sstale占用一个物理文件，其值为0，当多

//个sstable共用一个物理文件时，其值不一定为0

int64\_t block\_count\_; //该SSTable中block的个数

int64\_t block\_index\_record\_offset\_; //block索引记录的偏移值

int64\_t block\_index\_record\_size\_; //block索引记录的尺寸(包含记录头)

int64\_t bloom\_filter\_hash\_count\_;

int64\_t bloom\_filter\_record\_offset\_; //bloom filter记录的偏移值

int64\_t bloom\_filter\_record\_size\_; //bloom filter记录的尺寸(包含记录头)

int64\_t schema\_record\_offset\_; //schema记录的偏移值

int64\_t schema\_record\_size\_; //schema记录的尺寸(包含记录头)

int64\_t key\_stream\_record\_offset\_; //key stream记录的偏移值

int64\_t key\_stream\_record\_size\_; //key stream记录的尺寸(包含记录头)

int64\_t blocksize\_;

int64\_t row\_count\_;

uint64\_t sstable\_chcekesum\_; //sstable文件的校验和

uint64\_t first\_table\_id\_; //sstable文件中第一个table的id，用于schema cache key

int64\_t frozen\_time\_; //ups冻结大版本的时间

int64\_t reserved64\_[reserved64\_num\_]; //必须置0

**char** compressor\_name\_[ MAX\_COMPRESSION\_NAME\_SIZE];

int16\_t row\_value\_store\_style\_; //稠密(dense)，稀疏(sparse)或者混合(mixed)格式

int16\_t reserved\_; //必须置0

int32\_t table\_count\_; //table数量

ObTableTrailerInfo table\_info\_[1]； //v2.0不再使用，仅为了v1.0兼容

//以下变量用于管理内存中的key buffer，不会序列化和反序列化

char\* key\_buf\_; //key buffer 存放table的 start key 和 end key

int64\_t key\_buf\_size\_; //key buffer 大小

int64\_t key\_data\_size\_; //实际 key data 长度

bool own\_key\_buf\_; //key buffer的内存是否是trailer分配的

};

### 2.7 Bloom Filter

#### 2.7.1模块描述

Bloom Filter是一种数据结构(参见wiki)，用来判定某一个key是否属于某个集合，（有一定误判概率，如果判定在集合内，不一定在；但是如果判定不在集合内，那么一定不在，可以挡住很多not in请求）；

Bloom Filter应用在SSTableWriter中，是用来判定一个rowkey是否存在于这个SSTable file当中；当生成SSTable file的时候，需要根据插入的rowkey同时生成Bloom Filter结构。每插入一条数据，都需要对bloom filter进行更新。

Bloom Filter比较简单，长度也固定，所以将bloom filter的生成和判断都放在一起。

#### 2.7.2 模块接口

**class** ObBloomFilter

{

**public**:

/\*\*

\* 将key放入bloom filter

\*/

**int** insert(**const** ObString& key);

/\*\*

\* 判断该key是否可能存在于本SSTable中

\*/

bool may\_contain(**const** ObString& key);

/\*\*

\* 序列化&反序列化

\*/

**int** serialize(**char**\* buf, **const** int64\_t buf\_len, **const** int64\_t& pos) **const**;

**int** deserialize(**const** **char**\* buf, **const** int64\_t data\_len, **const** int64\_t& pos);

};

### 2.8 Compress

#### 2.8.1模块描述

为了节省存储空间，SSTable file中的每一个block都可能被压缩以后存储在磁盘上，从磁盘上读出blocks的时候，需要做解压。

数据压缩模块，提供一个通用的数据压缩和解压接口，用户可以指定压缩和解压的算法。

#### 2.8.2 模块接口

**class** ObCompressor

{

**public**:

**enum**

{

COM\_E\_NOERROR = 0, // 正常

COM\_E\_INVALID\_PARAM = 1, // 参数错

COM\_E\_NOIMPL = 2, // 没有实现

COM\_E\_OVERFLOW = 3, // 缓冲区溢出

COM\_E\_DATAERROR = 4, // 数据格式错误

COM\_E\_INTERNALERROR = 5, // 内部错误

// ...

};

**private**:

**const** **static** uint64\_t INTERFACE\_VERSION = ((uint64\_t)1)<<48;

**public**:

ObCompressor() : sohandle\_(NULL)

{

**return**;

};

**virtual** ~ObCompressor()

{

**return**;

};

**public**:

/\*

\* 数据压缩接口

\* @param [in] src\_buff 输出数据缓冲区

\* @param [in] src\_data\_size 输入数据大小 \*

\* @param [in] dst\_buff 存放结果数据的缓冲区

\* @param [in] dst\_buff\_size 传入的结果数据缓冲区的大小

\* @param [in] dst\_data\_size 压缩结果数据的大小

\*/

**virtual** **int** compress(**const** **char** \*src\_buffer,

**const** int64\_t src\_data\_size,

**char** \*dst\_buffer,

**const** int64\_t dst\_buffer\_size,

int64\_t &dst\_data\_size) = 0;

/\*

\* 数据压缩接口

\* @param [in] src\_buff 输出数据缓冲区

\* @param [in] src\_data\_size 输入数据大小

\* @param [in] dst\_buff 存放结果数据的缓冲区

\* @param [in] dst\_buff\_size 传入的结果数据缓冲区的大小

\* @param [in] dst\_data\_size 压缩结果数据的大小

\*/

**virtual** **int** decompress(**const** **char** \*src\_buffer,

**const** int64\_t src\_data\_size,

**char** \*dst\_buffer,

**const** int64\_t dst\_buffer\_size,

int64\_t &dst\_data\_size) = 0;

/\*

\* 设置每个压缩块或滑动窗口大小

\* 不是所有算法都必须提供

\*

\* @param [in] compress\_block\_size 要设定的压缩块或滑动窗口大小

\*/

**virtual** **int** set\_compress\_block\_size(**const** int64\_t compress\_block\_size)

{

(**void**)(compress\_block\_size);

**return** COM\_E\_NOIMPL;

};

/\*

\* 设置数据压缩级别

\* 不是所有算法都必须提供

\*

\* @param [in] compress\_level 要设定的压缩等级

\*/

**virtual** **int** set\_compress\_level(**const** int64\_t compress\_level)

{

(**void**)(compress\_level);

**return** COM\_E\_NOIMPL;

};

/\*

\* 根据传入的大小计算压缩后最大的可能的溢出大小

\* 不是所有算法都必须提供

\*/

**virtual** int64\_t get\_max\_overflow\_size(**const** int64\_t src\_data\_size) **const**

{

(**void**)(src\_data\_size);

**return** (int64\_t)-1;

};

/\*

\* 获取当前压缩算法名

\*/

**virtual** **const** **char** \*get\_compressor\_name() **const** = 0;

/\*

\* 获取接口版本号

\* 子类不需要实现

\*/

int64\_t get\_interface\_ver() **const**

{

**return** INTERFACE\_VERSION;

};

**private**:

**friend** ObCompressor \*create\_compressor(**const** **char** \*compressor\_lib\_name);

**friend** **void** destroy\_compressor(ObCompressor \*compressor);

**void** set\_sohandle(**void** \*sohandle)

{

sohandle\_ = sohandle;

};

**void** \*get\_sohandle()

{

**return** sohandle\_;

};

**private**:

**void** \*sohandle\_;

};

/\*

\* 指向动态库要实现的函数的指针声明

\*/

**typedef** ObCompressor\* (\*compressor\_constructor\_t)();

**typedef** **void** (\*compressor\_deconstructor\_t)(ObCompressor \*compressor);

#### 2.8.3对外接口

/\*

\* 根据传入的名字加载动态库， 返回一个压缩方法实例

\*/

**extern** compressor\_t \***create\_compressor**(**const** **char** \*compressor\_lib\_name);

/\*

\* 销毁一个压缩方法实例

\*/

**extern** **void** **destroy\_compressor**(compressor\_t \*compressor);

/\*

\* 初始化压缩框架

\*/

**extern int init\_compress**();

### 2.9 SSTableWriter接口

**class** ObSSTableWriter

{

**public**:

/\*\*

\* 根据schema新建一个sstablet文件，准备写入，Sstable文件名用

\* sstable\_id,sstable文件名由上层应用提供

\* **@param** [in] schema 需要生成tablet文件的schema的表

\* **@param** [in] path 写入磁盘的Sstbale文件的全路径，包括文件名

\* **@param** [in] compressor\_name 压缩器名称

\* **@param** [in] table\_id 全局唯一table id，向rootserver汇报需要table\_id

\* **@param** [in] table\_version table版本号

\* **@param** [in] store\_style sstable存储格式，稠密，稀疏，混合

\* **@param** [in] block\_size，block大小，默认64K

\* **@param** [in] element\_count 存入sstable中的行数估计值，用于bloom fileter

\* **@return** [out] 成功返回OB\_SUCCESS，失败返回OB\_ERROR

\*/

**int** create\_sstable(**const** ObSStableSchema& schema, const ObString& path, const ObString& compressor\_name, const uint64\_t table\_id, const int64\_t table\_version, const int store\_style = 1, const int64\_t block\_size = 64 \* 1024，const int64\_t element\_count = 0);

/\*\*

\* 加入一行数据,调用append\_row的顺序和row\_key本身的排序一致

\* 函数内部需要对row做schema检查，

\* 第一个加入的rowkey作为sstable的start\_key

\* 判断rowkey > prev\_key,判断row\_key传入的顺序

\* add\_row的同时，需要更新block index和bloom filter

\* 当加入的row占用的空间大于blocksize时，将该block压缩并写入磁盘，

\* 并清空SstableWriter中的block buffer，这里可以优化，多积累几个

\* block再一起写入磁盘或是在内存中完全组装好Sstable后再写入磁盘。

\*

\* **@param** [in] row 一行数据，包含了rowkey和value

\* **@param** [out] approx\_space\_usage 当前sstable文件大概占用的磁盘

\* 空间（bytes），当上层应用发现sstable文件的磁盘空间大于

\* 某个阈值时，调用close\_file方法结束当前Sstable的写操作。

\*

\* **@return** 成功返回OB\_SUCCESS，失败返回OB\_ERROR

\*/

**int** append\_row(**const** ObSSTableRow& row，int64\_t& approx\_space\_usage);

/\*\*

\* 一个sstable的数据加入完毕以后,将SstableWriter buffer中未写入磁盘

\* 的数据写入磁盘，并更新sstable的end\_key

\* 写入block index

\* 写入bloom filter bitmap

\* 写入schema

\* 写入trialer

\* 写入trailerOffset

\* 重新初始化ObSstableWriter对象便于重用对象

\* **@param** [out] trailer\_offset trailer在文件中的偏移地址

\* **@param** [out] sstable\_size sstable在文件大小

\* **@return** 成功返回OB\_SUCCESS，失败返回OB\_ERROR

\*/

**int** close\_sstable(int64\_t& trailer\_offset);

**int** close\_sstable(int64\_t& trailer\_offset, int64\_t& sstable\_size);

};