ChunkServer系统结构详细设计

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **文档版本** | **修订章节** | **修订原因** | **修订日期** | **修订人** |
| **1** | 0.1 | 全文 | 新建 | 2010-8-04 | 曲山 |
| **2** | 0.2 | 2.5.6 | 修改Tablet子系统接口 | 2010-8-10 | 曲山 |
| 3 | 0.3 | 2.6.2.2 | 修改SSTableReader子系统 | 2010-8-16 | 段飞 |
| 4 | 0．4 | 2.6.2.2 | 修改SSTableReader子系统 | 2010-8-17 | 段飞 |
| 5 | 1.0 | all | 根据后续调整修改文档 | 2010-10-28 | 曲山 |

# 系统设计

## 系统综述

ChunkServer存储系统所有表的静态数据部分，CS采用SSTable的形式存储表数据，按行排序好后存储。每个数据存储单位，称为Tablet,每个Tablet大小相当，负责一个表中某个范围内的数据。每个Tablet会复制为至少3个备份存放在不同的CS上。

ChunkServer还需接收UpdateServer的补丁文件，与本地的静态数据进行合并。

## 名词解释

OB：全称OceanBase，核心系统研发部的海量存储系统，一期解决淘宝的收藏夹应用；

UPS：全称Update Server，OB系统更新子系统，通过主备保证可靠性；

CS：全称Chunk Server，OB静态表系统，存储每天合并UPS更新数据生成的静态表；

MS：全称MergeServer，OB系统中负责合并CS数据和UPS数据，形成最终结果返回给客户。

MemTable：内存表，UPS内存中通过MemTable实现每天更新数据的组织和查询。根据是否冻结又可区分为frozen Memtable和active Memtable；

**SSTable**: static sorted table, 一个表格在某个范围某段时间内的数据。目前ChunkServer上的只保存SSTable数据

**Tablet**: 一个表格的的某个range全量数据SSTable + MemTable

Commit Log：操作日志，UPS为了保证可靠性，首先将更新操作序列化到主备机的文件中，称为操作日志；

API：全称client API，客户端通过API访问OB系统，UPS也可通过API访问CS静态表数据；

UPS Master(Master)：全称Update Server Master，处理实际读写请求的Update Server；

UPS Slave(Slave)：全称Update Server Slave，用于实时备份的Update Server；

## 性能指标

假设：16核Intel(R) Xeon(R) CPU E5520 @ 2.27GHz ，24GB内存，300GSAS \* 6硬盘，3000W条记录，平均大小为300字节（假设100字节的收藏夹单条记录膨胀3倍）。

读性能：单机读取并发量 > 100/s，4个9的读取操作延时 < 59ms；

扫描性能：单行扫描并发量与读操作并发量相同，多行扫描带宽 > 30MB/s；

## 系统结构

表数据被切分成Tablet以后(大小相当)，拷贝三份并均衡分布存储在不同的CS上。

客户端所有的查询请求，都先通过RootServer找到对应的CS后，在指定的CS读取具体的数据，会找到CS对应的MergeServer ，MergeServer会挑选一台距离最近，数据最新的CS进行查询，然后根据CS返回数据的版本，从UPS上取更新的数据进行合并后返回给客户端。

# 子系统设计

## 子系统划分

ChunkServer在逻辑上划分为如下子系统：

网络处理子系统：

负责网络数据的接收和发送，数据的解包，打包。

任务队列子系统：

处理网络子系统接收到的数据包，进行任务排队和任务调度。

接口子系统：

将任务队列里面的请求包，判定请求类型，检查参数合法性，并分解为对其他子系统的功能调用。

接口子系统还包括CS对外部系统的调用接口，比如汇报tablets，心跳信息，对UpdateServer的调用。

Tablet子系统：

Tablet定位、查询对应SSTable的数据，并查询UPS中Memtable中的数据与之合并。同时根据schema中的join规则进行合并。

SSTable子系统

SSTable数据读取、SSTable文件的生成(按照SSTable的格式)以及in-memory的SSTable实现等

Cache子系统

管理系统中所有需要内存化的数据，提供淘汰机制。

每日合并子系统：

将CS上的SSTable静态数据与UPS上的动态更新数据做一个全量的合并，合并以后生成新的SSTable文件。

## 网络处理子系统

### 子系统描述

负责网络数据的接收和发送，并将接收到的数据解包反序列化为系统定义的消息，另外也负责将调用远程接口的数据包序列化为数据流并发送给远端。

网络子系统需要自己管理多个Connection，(包括一个被动Accetpor和多个数据链接)，并处理这些Connection上的事件(网络事件和超时事件).

当网络子系统从网络上接收到数据时，会将数据读入自己的缓冲区，然后调用应用提供的解包接口，将网络数据解包为消息包，交给任务队列处理。

应用处理完消息以后，将响应包发给网络子系统，网络子系统推入消息发送队列，然后调用应用的序列化接口将包序列化为字节流，拷入自己的发送缓冲区，发送给网络。

### 子系统接口

目前的网络处理子系统使用tbnet，沿用原有的处理接口。

/\*\*

\* 解包打包接口

\*

\*/

**class** ObPacketStreamer

{

**public**:

/\*

\* 读取包头，判断包长度

\* @param [in] input, socket读入数据缓冲

\* @param [in] header, 根据输入字节流解析包头信息

\* @param [out] broken, 包是否合法

\*/

virtual bool get\_packet\_info(const ObDataBuffer\* input, ObPacketHeader\* header, bool& broken);

/\*

\* 反序列化，解析一个数据包

\* @param [in] input, socket读入数据缓冲

\* @param [out] header, 上一个接口解析的包头信息

\* @return 返回解析好的包对象

\*/

virtual ObPacket\* decode(const ObDataBuffer\* input, const ObPacketHeader\* header);

/\*

\* 序列化一个数据包

\* param [in] packet 需要序列化的包对象

\* param [out] output 序列化字节流缓冲区

\*/

virtual ObPacket\* encode(const ObPacket\* packet, ObDataBuffer\* output);

};

/\*\*

\* 事件处理接口

\*

\*/

**class** ObServiceHandler

{

**public**:

virtual **int** handle(**const** ObPacket\* packet) = 0;

};

## 任务队列与工作线程池

### 子系统描述

任务队列负责接收网络子系统接收到的包，而工作线程池是在任务队列上等待的一组执行线程，每一组（个）线程等待一个任务队列，一旦任务队列中有包，则当前空闲的线程便将消息包取出，交给特定的应用逻辑去处理。

当任务队列空的时候，所有的线程在队列上等待；当任务队列满以后，线程池中所有的队列都在忙于处理任务，如果网络子系统有新的包需要处理，则任务队列会选择将该包丢弃，或

是直接返回用户繁忙。

在ChunkServer中，所有的请求包都推入到系统唯一的任务队列，有工作在上面的一组工作线程处理。

### 任务队列

#### 模块描述

任务队列是一个先进先出队列，用于网络请求的排队，所有的网络数据包，经过网络子系统的处理以后，都会转化为消息包的形式，为了避免阻塞网络子系统，提高系统响应速度和吞吐量，这些消息包并不会立即在网络子系统的接收线程中进行处理，而是先进入任务队列进行排队，等待工作在任务队列上的工作线程池进行处理。

任务队列在空的状况下，系统空闲，无任务可以处理，所有工作线程池中的线程都阻塞等待任务进入队列，当任务队列满的情况下，工作线程池中所有线程都在等待，无法处理处理此请求，此时，要么将此消息包丢弃（直接返回客户端系统繁忙），要么将队列前端的任务丢弃；

#### 对外接口

/\*\*

\* 任务队列,先进先出队列

\* 网络上收到的消息包，先放入任务队列，

\* 然后由工作在任务队列上的线程池来处理。

\*/

**class** ObTaskQueue

{

**public**:

/\*

\* 将消息压入队列

\* @param [in] packet 压入的消息包

\* @return 压入是否成功

\*/

bool push(**const** ObPacket\* packet);

/\*

\* 从队列中取出一个消息包

\* @return 消息包的指针

\*/

**const** ObPacket\* pop();

/\*

\* 判断任务队列是否为空

\*/

bool empty();

/\*

\* 判断任务队列是否满

\*/

bool full();

/\*

\* 队列当前的长度

\*/

int32\_t size() **const**;

};

### 工作线程池

#### 模块描述

CS系统有一个任务队列，并有一个工作与其上的工作线程池，所有的工作线程都在等待任务队列里面新的任务消息包，一旦有消息包进入任务队列，将有一个线程被唤醒，并从任务队列中取出一个消息包，交给应用提供的handler去处理。消息处理完成以后，该线程重新开始等待任务队列中新的任务消息包。

#### 对外接口

/\*\*

\* 线程池消息包处理接口

\*/

**class** ObThreadPoolPacketHandler

{

**public**:

virtual **int** handle(**const** ObPacket\* packet, **void**\* args) = 0;

};

**class** ObThreadPool

{

**public**:

/\*

\* 初始化线程池

\* @param thread\_count 线程个数

\* @param handler 事件处理对象

\* @param args 附带参数

\* @return

\*/

**int** set\_parameter(int32\_t thread\_count, ObThreadPoolPacketHandler\* handler, **void**\* args);

/\*

\* 将一个消息交给线程池处理，线程池会将其加入到自己维护的任务队列当中

\* @param [in] packet 需要处理的消息包

\* @return 处理是否成功

\*/

**int** push(**const** ObPacket\* packet);

/\*

\* 线程池启动

\*/

**void** start();

/\*

\* 线程池停止运行

\*/

**void** stop();

/\*

\* 等待线程池结束工作

\*/

**void** wait();

**private**:

ObTaskQueue \* task\_queue\_;

ObThreadPoolPacketHandler \* handler\_;

};

### 子系统接口

该子系统实现网络子系统的消息处理接口，把所有的消息包压入自己内部的任务队列。

**class** TaskQueueSubSystem : public ObServiceHandler

{

**public**:

virtual **int** handle(**const** ObPacket\* packet);

};

## 接口子系统

### 子系统描述

将任务队列里面的请求包，判定请求类型，检查参数合法性，并分解为对其他子系统的功能调用。

接口子系统还包括CS对外部系统的调用接口，比如汇报tablets，心跳信息，对UpdateServer的调用。

接口子系统还需要处理与RootServer的心跳与租约。

接口子系统提供一个处理消息包的handler，供任务队列的线程池调度运行。工作线程池从任务队列中拿到一个包，然后交给接口子系统的handler进行处理。

接口子系统中的服务接口分为很多类，每一类服务接口由一组线程池来处理。(比如查询服务接口处理线程池，控制命令接口处理线程池)

### 对外服务接口

#### 描述

主要处理客户端或是其他系统发送过来的查询请求，控制命令。

该模块实现一个线程池系统的handler接口，供上层线程池在处理消息包的时候处理具体的消息包，根据消息包的类别，该模块会将消息包的参数进行检查，分解，或是转化为其他子系统的调用，来实现服务功能。

#### 对外接口

这个接口实现任务队列子系统的事件处理接口。

**class** ChunkServerServiceHandler : **public** ObThreadPoolPacketHandler

{

**public**:

**int** handle(**const** ObPacket\* packet, **void**\* args);

};

对于不同的服务接口类型，有不同的处理逻辑，将在handle中判断包类型分发到其他子系统进行处理。

### 外部调用接口

#### 描述

CS与其他系统进行交互的接口，通常是以远程调用的方式进行，外部调用接口模块负责封装对其他子系统的调用，并处理返回结果，对于CS内部来说，调用[调用接口模块]所提供的服务，就与本地函数调用一样。

#### 对外接口

##### 向RootServer汇报数据

/\*

\* ChunkServer向RootServer汇报其管理的tablets;

\* @param [in] server 汇报的ChunkServer

\* @param [in] tablets 管理的所有tablets信息

\*/

ObResultCode report\_tablets(const Server & server, const int64\_t memtable\_version, const TabletReportInfoList & tablets);

/\*

\* ChunkServer向RootServer汇报Tablets结束

\* @param [in] server 汇报的ChunkServer

\* @param [in] timestamp, 汇报的tablets的schema版本信息

\*/

ObResultCode report\_tablets\_over(const Server & server, const int64\_t memtable\_version);

/\*

\* ChunkServer向RootServer注册

\* @param [in] server 汇报的ChunkServer

\* @param [in] status == 0 表述系统初次启动 status == 1系统已经启动

\*/

ObResultCode register\_self(const Server & server, int32\_t& status);

/\*

\* ChunkServer向RootServer汇报磁盘容量信息

\* @param [in] server 汇报的ChunkServer

\* @param [in] total\_capacity 总的容量

\* @param [in] total\_used 总的使用容量

\*/

ObResultCode report\_capacity\_info(const Server & server,const int64\_t total\_capacity, const int64\_t total\_used);

##### 获取当前最新的schema

/\*

\* 总是获取最新的schema

\* @param [out] schema 最新的schema信息

\*/

ObResultCode fetch\_schema(const int64\_t version, ObSchemaManager& schema);

### 心跳与租约

#### 描述

CS启动时，会向RootServer发送一个注册信息，RootServer会返回一个状态：

status == 0， 表示整个系统是初次启动，这时RootServer会等待系统中所有的CS都注册以后，才开始进行手工启动。

CS这时需要等待RS的初始启动命令。

status == 1， 表示系统已经启动过了，CS是重启（或是第1次加入系统），这时CS会加载本地已有（如果有的话）的tablets,并向RS进行汇报。

一旦注册成功，RS会向所有的CS发送心跳信息，CS在接收到RS的消息以后需要更新自己的租约，如果租约没有过期，才可以对外提供服务。

#### 对外接口

##### RootServer心跳

/\*

\* RootServer向ChunkServer发送心跳，传递租约信息

\* @param [in] lease\_duration 租约时常

\*/

ObResultCode cs\_heart\_beat(const int64\_t lease\_duration);

## Tablet子系统

### 子系统描述

Tablet定位、查询对应SSTable的数据。

Tablet子系统从接口子系统接收到派发分解过来的查询需求，首先定位到指定的Tablet，查询委托给具体的Tablet进行查询。

Tablet查询分两块，一部分是静态数据查询，也就是本地SSTable的查询，交给SSTable子系统处理。另一部分是查询UPS发送过来的sstable patch files。其中sstable patch files的数据查询的输入依赖于SSTable查询的结果，需要在SSTable查询完成以后进行。最后Tablet会将两块查询结果按照Schema进行Merge与join操作，得到最后的结果后返回给用户。

Tablet子系统中分为几个部分，TabletManager负责管理和定位所有的Tablet对象，Tablet本身负责管理自身拥有的SSTable(目前只有一个)，TabletScanner负责查询以及合并数据，返回查询结果。

### TabletManager

#### 模块描述

TabletManager用于管理CS中所有的Tablet对象，其自身维护两个hashtable，一个是当前服务的Tablets hashtable,一个是正在进行合并的Tablets hashtable；(这个hashtable只有在合并期间才有内容)

#### 对外接口

TabletManager需要提供查询接口，根据sstabe\_file\_id查找到指定的tablet对象；在每日合并子系统在进行合并操作时，需要提供扫描接口遍历所有的Tablet，而且在合并成功后需要提供接口Load指定的SSTable，生成Tablet对象。

另外，当RootServer完成切换以后，会通知CS切换到新表服务，CS会将老的Tablets全部卸载，（或是分批卸载，用老表服务客户端的缓存一段时间）；对所有的请求用新Tablets进行服务，TabletManager需要提供一个接口完成这个过程。

**class** TabletManager

{

**public**:

/\*

\* 查找一个tablet对象

\* @param [in] range 查询Tablet对应的范围信息

\* @param [in] sstable\_file\_id SSTable的文件id,

\* @param [out] tablet 找到的Tablet对象，如果没有找到，置为NULL

\* @param [type] 找到的Tablet对象的类型，是前天的老数据还是新的合并数据,这个类型决定查找UPS数据的类型

\* @return 0 找到, otherwise 没有找到，或是找到以后与要求的Range不符合

\*/

**int** get\_tablet(**const** Range& range, uint64\_t sstable\_file\_id, **const** Tablet\* &tablet) **const**;

/\*

\* 根据sstable路径,装载一个SSTable,并生成一个Tablet对象，

\* @param [in] range 查询Tablet对应的范围信息

\* @param [in] sstable\_file\_id SSTable的文件id,

\* @param [in] path sstable文件的路径

\* @param [in] type 是合并后的表还是初次初始化扫描的表

\* @param [out] tablet 生成的Tablet对象，如果没有找到，置为NULL

\* @return 0 装载成功

\*/

**int** load\_tablet(**const** Range& range, uint64\_t sstable\_file\_id, **const** ObString& path, **const** int32\_t type, **const** Tablet\* &tablet);

/\*

\* RootServer切换到新的RootTable以后，通知ChunkServer进行切换

\* 这时所有的新Tablet已经装载完毕.卸载所有的老表(或是分批)

\*/

**int** switch\_to\_new\_tablets();

};

### Tablet

#### 模块描述

概念上来讲，一个Tablet就是一个Table在某个Range之间的全部数据，而这些数据分为俩个部分，一部分是静态数据，存放在本地的SSTable，（每隔一段时间刷新一次，目前在CS中，SSTable每隔一天刷新一次，而且是刷新的同时即时合并，所以SSTable只有一个），另外一部分是内存中的动态数据，存放在远程的UPS上。

#### 对外接口

**class** Tablet

{

**public**:

SSTable\* get\_sstable() **const**;

int64\_t get\_time\_stamp() **const**;

**const** Schema\* get\_schema() **const**;

/\*

\* 创建一个Scanner对象对本Tablet进行数据查询

\* param [in] scan\_param 查询参数，包括查询的范围，条数限制，要查询的列

\* @return 返回一个Scanner,该Scanner查询并合并数据

\*/

TabletScanner\* create\_scanner(**const** ObScanParam\* scan\_param) **const**;

};

### TabletScanner

#### 模块描述

TabletScanner负责真正的查询操作，因为TabletScanner是针对Tablet的Scanner，查询出来的完整的数据。TabletScanner需要查询本地静态数据(这部分操作委托给SSTable子系统)，然后根据本地结果和schema查询UPS上的动态数据（这部分操作委托给接口子系统），然后对两个结果进行合并。

#### 对外接口

/\*\*

\* ObScanner Interface

\* ObScanner用于数据扫描操作，

\*/

**class** TabletScanner {

**public**:

/\*\*

\* 实际的查询方法，一旦do\_scan调用完毕，可以调用下面的get,next方法取数据了

\*/

int do\_scan(const ObScanParam\* param);

/\*\*

\* 重置遍历位置接口，如果需要进行二次遍历

\*/

int reset();

/\*\* move curson to next row, if no row remain, return false \*/

**bool** **next**();

/\*\* get timestamp of this scan tablet \*/

**const** int64\_t **get\_timestamp**() **const**;

/\*\* get current row \*/

**const** ObRowInfo\* **get\_row**() **const**;

/\*\* get the table name of current row \*/

**const** ObString **get\_table\_name**() **const**;

/\*\* get the table id of current row \*/

**const** uint64\_t **get\_table\_id**() **const**;

/\*\* get the column value of the specify column name \*/

**const** ObObj\* **get\_value\_by\_column\_name**(**const** ObString& column\_name) **const**;

/\*\* add a new row to this scanner \*/

**int** **add\_row**(**const** ObRowInfo& row\_info);

**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**:

int64\_t current\_index;

ObRowInfo row\_list\_[MAX\_ROW\_NUMBER\_PER\_QUERY];

ObArrayHelper<ObRowInfo> row\_info\_list\_;

};

### MemTableScanner

#### 模块描述

TabletScanner的数据来源有两部分，一部分是SSTable子系统提供的静态数据，也以Scanner的接口形式提供，另外一份是UPS上的memtable部分，需要调用【接口子系统】的外部接口来查询。

MemTableScanner只是对UPS memtable查询接口的一个数据封装，用来提供给TabletScanner关于memtable的数据源。

MemTableScanner的输入依赖于SSTable静态数据的查询结果，需要根据schema的merge和join规则查询指定的UPS内存表。

#### 对外接口

与TabletScanner的接口一致。其实现上是直接调用UPS接口取数据。

### 子系统接口

#### 数据查询接口

scanner作为一个存放结果的对象，其接口参见接口文档中的ObScanner类。

用户（比如【接口子系统】或是【每日合并子系统】）调用Tablet提供的查询接口以后，调用scanner对象的get\*,next方法使用数据（或是序列化成数据流）以后，需要destroy scanner对象。

单条查询：

ObResultCode **cs\_get**(**const** ObGetParam& get\_param, ObScanner\* &scanner, ObOperateResult& operation);

范围查询：

ObResultCode **cs\_scan**(**const** ObScanParam& scan\_param, ObScanner\* &scanner, **bool**& end\_flag);

批量查询：

ObResultCode **cs\_batch\_get**(**const** int64\_t batch\_num, **const** ObGetParam\* get\_params, ObScanner\* &scanner, ObOperateResult\* operations);

## SSTable 子系统

## 子系统描述

SSTable子系统负责SSTable的文件的生成，数据查找、读取。

SSTable静态数据查询：由上层接收到外部请求以后，找到指定的Tablet进行查询，Tablet查询分两块，一部分是静态数据查询，也就是本地SSTable的查询，交给SSTable子系统处理。另一部分交给接口子系统到UPS处理。

SSTable静态数据查询是一个分级查询过程，一级查询在SSTable中的block index中进行查询，block index是在首次查询时被装载到内存当中（保存在Cache子系统中的LRU链表里），找到指定的block以及在SSTable中的偏移，二级查询将找到的Block装载到内存中（具体内存也保存在Cache子系统中），找到具体的key-value对。查找完成以后会将value根据查询参数分解为结构化的cell信息，返回给请求端。

SSTable文件生成：系统根据指定的Schema，按SSTable的格式，生成bloom filter, block index, block internal index, trailer，写入磁盘指定的位置。

## 二级子系统划分

SSTable不会动态更新，故将SSTable子系统分为两个二级子系统：SSTableWriter和SSTableReader，SSTableWriter负责生成SSTable，后者负责从SSTable中读取、查询数据。

SSTable的数据可以经过压缩后放在磁盘上。磁盘上的位置由【每日合并子系统】中的【SSTable文件管理】模块指定。

## SSTableWriter

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。

## IndexBuilder

###### 模块描述

为便于查找，SSTable会建立一些信息来索引数据，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

\*/

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

/\*\*

\* 序列化

\*/

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

};

## Bloom Filter

###### 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。

###### 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);

};

## Compress

###### 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。

###### 模块接口

class Compress

{

/\*

\* 数据压缩接口

\* dst\_buff 存放结果数据的缓冲区

\* dst\_buff\_size 传入的结果数据缓冲区的大小

\* dst\_data\_size 压缩结果数据的大小

\* src\_buff 输出数据缓冲区

\* src\_data\_size 输入数据大小

\*/

virtual com\_errno\_t compress(char \*dst\_buff,

const uint64\_t dst\_buff\_size,

uint64\_t &dst\_data\_size,

const char \*src\_buff,

const uint64\_t src\_data\_size) = 0;

/\*

\* 数据解压缩接口

\* 参数说明同compress

\*/

virtual com\_errno\_t decompress(char \*dst\_buff,

const uint64\_t dst\_buff\_size,

uint64\_t &dst\_data\_size,

const char \*src\_buff,

const uint64\_t src\_data\_size) = 0;

/\*

\* 根据传入的名字加载动态库， 返回一个压缩方法实例

\* reload\_lib 如果动态库已经被加载过，是否重新加载

\*/

compressor\_t \*create\_compressor(const char \*compressor\_lib\_name, bool reload\_lib);

/\*

\* 销毁一个压缩方法实例

\* unload\_lib 在没有压缩方法实例引用这个动态库的情况下，是否卸载

\*/

void destroy\_compressor(compressor\_t \*compressor, bool unload\_lib);

};

##### SSTableWriter接口

**class** ObSSTableWriter

{

**public**:

/\*\*

\* 初始化

\* 根据schema新建一个tablet文件，准备写入

\* 需要生成一个新的chunkserver唯一的tablet\_id

\* 需要根据规则生成tablet文件的path

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

\* **@param** [in] path 写入磁盘的路径

\* **@param** [in] id 唯一序列号

\* @param [in] range 该SSTable负责的range.

\* **@return**

\*/

**int** initialize(**const** ObSchema& schema, const ObString& path, const uint64\_t id, const ObDataInterval& range);

/\*\*

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

\* 函数内部需要对cells对应的列做schema检查，

\* 另外判断rowkey>start\_key,判断row\_key传入的顺序

\* add\_row的同时，需要生成

\* block index

\* bloom filter

\* tablet index

\*

\* **@param** [in] rowkey 传入行的rowkey

\* **@param** [in] cells 传入的列数据

\* **@return** 0 成功 != 0 失败

\*/

**int** append\_row(**const** ObString& rowkey, **const** ObCell \* cells);

/\*\*

\* 一个tablet的数据加入完毕以后,真正写入tablet文件

\* 写入block index(这个可能在add\_row的时候一并写入)

\* 写入tablet index

\* 写入bloom filter bitmap

\* 写入tablet trailer

\* **@return**

\*/

**int** finalize();

};

## SSTableReader

SSTableReader主要负责数据读取，查询相关工作，SSTableReader可以分为以下几个子模块：

## ObSSTableReader

###### 模块描述

ObSSTableReader主要功能包括: 打开SSTable文件，读取trailer， bloomfilter等相关信息，创建decompress\_t等, ObSSTableReader只会在open时打开文件，读取相关的信息以后就会close，它不管理SSTable文件的fd， fd和SSTable其它数据的读取管理都委托给ObCaceh做。

ObSSTableReader负责具体数据的查询，具体数据的查询工作委托给ObSSTableScanner

###### 模块接口

**class** ObSSTableReader

{

**public**:

**ObSSTableReader**() {}

**virtual** **~ObSSTableReader**() {}

/\*\*

\*加载一个SSTable文件, 打开SSTbale文件读出trailer, bloom\_filter等相关信息,并创建decompress\_t

\*@param [in] fileName SSTable文件的文件名(全路径)

\*@param [in] trailer\_offset trailer在SSTable文件中的偏移

\*@return OB\_SUCCESS: 成功, OB\_ERROR: 失败

\*/

**int** **open**(**const** **char**\* fileName, **const** int64\_t trailer\_offset);

/\*\*

\*获取该sstable所属的table的id

\*/

uint64\_t **get\_table\_id**() **const**;

/\*\*

\*获取该sstable所拥有的行数

\*/

int64\_t **get\_total\_rows\_count**() **const**;

/\*\*

\*获取该sstable的schema

\*/

common::ObSchema\* **getSchema**() **const**;

/\*\*

\*获取range开始key

\*/

**const** ObString& **get\_start\_key**() **const**

/\*\*

\*获取range的结束key

\*/

**const** ObString& **get\_end\_key**() **const**;

/\*\*

\*创建该sstable所使用的压缩/解压器

\*/

decompress\_t\* **get\_decompressor**();

/\*\*

\*判断一个rowkey是否在本sstable中

\*/

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

**private**:

/\*\*

\*加载bloom filter

\*/

int32\_t **load\_bloom\_filter**();

};

## ObSSTableScanner

模块描述

ObSSTableScanner主要提供的功能包括: 单条/范围查询, 加载下一个block, 数据过滤

在查询过程中需要用到索引信息，索引信息是由ObCache管理，ObSSTableScanner用到索引时需要向ObCache申请，用完以后需要还给ObCache，申请索引信息需要用到SSTable Id 和索引在SSTable文件中的位置，以及索引信息的大小，这些信息需要由ObSSTableReader提供

ObSSTableScanner是遍历一个SSTable静态文件的类，SSTable静态文件是分成多个Block存储，所以在进行范围查询时可能会跨跃多个Block

ObSSTableScanner调用ObSSTableBlockIndex查询block相关的信息找到一个具体的block，然后向ObCache申请加载block，遍历block的任务交给ObSSTableBlock去做，ObSSTableScanner将遍历后的数据进行过滤并放入缓冲区，这一过程由do\_scan完成，调用do\_scan以后所有的数据已经存储在缓冲中

###### 模块接口

SSTableScanner是实现TabletScanner的一部分，SSTableScanner提供的数据就是TabletScanner需要的静态数据部分。

**class** ObIterator

{

**public**:

**virtual** **bool** **next\_cell**() = 0;

**virtual** **int** **get\_cell**(ObCellInfo\* cellinfo) = 0;

};

**class** ObSSTableScanner : **public** ObIterator

{

**public**:

**ObSSTableScanner**();

**virtual** **~ObSSTableScanner**();

/\*\*

\* move cursor to next cell, if no cell remain, return false

\*/

**bool** **next\_cell**();

/\*\*

\* 获取一个cell

\* @param[out] cell相关信息

\* @return OB\_SUCCESS: 成功, OB\_ERROR: 失败

\*/

**int** **get\_cell**(ObCellInfo\* cell);

/\*\*

\* 设置scan所用到的参数

\* @param [in] context scan所用到的外部参数

\* @param [in] sstable\_reader SSTableReader

\*/

**int** **set\_scan\_param**(**const** common::ObScanParam& context, **const** ObSSTableReader& sstable\_reader);

};

## ObSSTableBlockIndex

###### 模块描述

ObSSTableBlockIndex 存储了SSTable文件中每一个block在SSTable文件中的偏移位置的信息，以及每一个block最后一个key相关信息, ObSSTableBlockIndex主要完成以下几个功能:

1．根据rowkey查询出rowkey所在的block在SSTable文件中的偏移信息

2． 反序列化ObSSTableBlockIndex数据

###### 模块接口

**template**<**typename** **T**>

**class** ObSSTableBlockIndex

{

**typedef** **T**\* iterator;

**typedef** **const** **T**\* const\_iterator;

**public**:

**ObSSTableBlockIndex**();

**virtual** **~ObSSTableBlockIndex**();

/\*\*

\*在blockindex中查询block的位置信息

\*@param [in] key 要查找的key

\*@param [in] search\_mode 查询模式

\*@param [in|out] pos\_info 查询出来的结果

\*SearchMode:

\*LESSEQUAL,LESSTHAN 返回满足条件块，按照主key降序

\*EQUAL,GREATEREQUAL,GREATETHAN 返回满足条件块，按照主key升序

\*/

**int** **search**(**const** ObString& key, **const** SearchMode search\_mode, ObBlockPositionInfos& pos\_info);

NEED\_SERIALIZE\_AND\_DESERIALIZE;

**private**:

**T**\* index\_begin\_;

**T**\* index\_end\_;

**char**\* base\_;/\*\*向ObCache申请存储反序列化后的BlockIndex相关信息的内存开始指针,主要用于存储offset, key, ObString\*/

int64\_t base\_length\_; /\*\* base\_的长度\*/

common::ObArrayHelper<common::ObString> row\_end\_key\_;/\*\*rowkey数组\*/

int32\_t index\_entries\_;/\*\*有多少个offset\*/

};

## Cache与内存管理子系统

### 子系统描述

cache子系统提供的功能包括：文件系统缓存、定长对象缓存、kv缓存、定长内存池、边界对齐定长内存池、循环内存buffer。在一期的实现中，处于项目排期考虑，可以不实现kv缓存。

定长内存池：定长内存池是最简单的，顾名思义从定长内存池分配的内存都是固定长度的。定长内存池可以作为全局的“临时”的内存分配器，也可以作为定长对象工厂的内存分配器。对于全局的内存分配器，可以使用16kB作为定长内存的长度，如果分配的内存小于16k，则从内存池中分配，否则则直接调用malloc。但是对于所有的cache子系统而言，他们都不能够直接使用全局的定长内存池分配内存，因为这样会导致较大的内存浪费，因此他们必须有自己的内存池。

边界对齐定长内存池：由于dio方式读取磁盘文件，要求每次用于读取的内存的首地址必须按照一定的边界大小对其，必须提供按照边界对齐的内存池。比如，按照A字节对齐，要求分配总共为B字节的内存空间，这种情况下内存池需要向系统分配A+B-1个字节的内存空间，并返回给调用者按照A字节对齐的首地址。边界对齐内存池是特殊的定长内存池，可以通过定长内存池的方式实现。

文件系统缓存对外提供pread接口，这意味着每次读取数据都会存在一次缓冲区的拷贝。内部实现，所有的读取操作都采用dio的方式每次读取按照磁盘物理块大小对齐的固定大小的块，这些块以文件名+offset作为key存放在内存hash表中，按照lru的方式进行淘汰。由于每次读取始终会进行一次内存拷贝，那么相对于系统cache而言，我们实现的文件系统缓存唯一的优势是能够更加“准确”的控制cache对内存的使用。文件系统缓存包含了block cache。从前面的描述可以看到文件系统缓存中的内存都是从边界对齐内存池中分配的。文件系统缓存按照lru策略进行淘汰。

定长对象缓存：block index在sstable文件中的存储格式是串行化后且与主机字节序无关的无关的结果，因此要通过block index进行查找必须先执行反串行化。同时，block index是系统中访问频度最高的一块，如果每次都需要进行反串行化，性能上会存在问题。因此，block index不能够直接通过文件系统缓存的方式进行cache，这里假设block index对象的长度比较接近，选择最大长度来作为block index的固定长度不会浪费太多的内存。定长对象缓存从定长内存池中分配内存，使用定长的二进制buffer作为key。

kv缓存：kv缓存用于保存单次查询的结果，相当于行级缓存。特征是key和value的长度变化比较较大，不适合使用定长内存池来管理内存。kv缓存的内存管理不使用完全通用的变长内存管理，将多个变长的key-value对放到一个大块的定长内存中，这一大块的内存作为lru淘汰的一个item，而不是每个kv单独淘汰。因此kv缓存仍然使用定长内存池。kv缓存的key和value都是二进制缓冲区。

循环内存buffer：把一个很大的buffer看作一个循环的字节数组，每次需要内存的时候就从循环字节数组的头分配相应的内存缓冲区长度。循环内存buffer要求内存释放的顺序与分配的顺序相同。

### 子系统结构图

### 子系统对外接口

#### 定长内存池

template <int32\_t ITEM\_SIZE>

class ObFixedMemPool

{

public:

struct MPHandle;

public:

// 申请一块定长内存

// @return NullHandle 分配失败

// 非NullHandle 分配成功

MPHandle malloc();

// 释放一块定长内存

// @param [in] MPHandle 代表内存的handle

// @return 0 成功

// -1 失败

int free(const MPHandle &handle);

// 直接清空内存池

void clear();

// 获取当前已分配的内存块数量

// @return >=0 正常

// <0 异常

int64\_t get\_item\_num() const;

// 根据handle获取内存指针

// @param [in] MPHandle 代表内存的handle

// @return NULL 失败

// 非NULL 成功

static void \*get\_memory(const MPHandle &handle) const;

};

/// 全局临时内存定长内存块大小

const int32\_t TEMPORARY\_MEM\_FIXED\_SIZE=64\*1024;

/// 全局临时内存分配器

class ObMemPool

{

private:

ObMemPool();

ObMemPool & operator=();

ObMemPool(const ObMemPool &pool);

public:

/// 获取独粒

static ObMemPool \* instance();

/// 分配内存

void \*malloc(int32\_t nbyte);

/// 释放分配的内存

void free(void \*ptr);

private:

ObFixedMemPool<TEMPORARY\_MEM\_FIXED\_SIZE> mem\_pool\_;

};

/// 在对象析构的时候释放分配的内存，减少内存分配与释放不匹配的bug

class ObMemBuffer

{

public:

/// 从ObMemPool中分配nbyte的内存

ObMemBuffer(int32\_t nbyte);

/// 析构函数，释放分配的内存

~ObMemBuffer();

/// 得到分配的内存

void \*get\_buffer();

};

#### BlockCache

class ObBlockCache

{

public:

// 读取tablet文件上的数据

// @param [in] tabletID 要读取的tablet的ID

// @param [in] sstable\_fname 要读取的sstable完整的磁盘文件路径

// @param [in] count 要读取的字节数

// @param [in] offset 要读取的offset

// @param [in] timeo\_us 阻塞等待的微秒极时间

// @param [out] buffer\_handle 封装了结果内存的指针,提供get\_buffer获取指针,析构后自动通知cache释放

// @return >=0 读取的数据量

// <0 失败

int32\_t pread(uint64\_t tabletID,

const char \*sstable\_fname,

int32\_t count,

int32\_t offset,

int64\_t timeo\_us,

BufferHandle &buffer\_handle);

};

#### ObjCache：定长对象缓存

ObjCache要求key和value都是定长数据

/// KeyT and ValueT must be fixed size

template <class KeyT, class ValueT>

class ObjCache

{

/\*\*

\* @fn int ObjCache::init(int64\_t cache\_mem\_size, int64\_t max\_non\_active\_usec)

\* 初始化ObjCache

\*

\* @param cache\_mem\_size cache 使用的内存大小, <0表示不限制内存使用

\* @param max\_non\_active\_usec

\* -# >0 单个item最大允许不被访问的时间, 超过该时间item会被淘汰

\* -# =0 按照lru淘汰

\* -# <0 不被淘汰，这种情况下忽略cache\_mem\_size参数

\*/

int init(int64\_t cache\_mem\_size, int64\_t max\_non\_active\_usec);

/// 分配定长的key和value对象

int malloc(KeyT \* &key, ValueT \* &value);

/// 往cache中增加内容, key和value通过malloc成员函数获得

int put(KeyT \* &key, ValueT \* &value);

/// 通过key获取对应的value，这里返回内部指针

int get(const KeyT &key, ValueT \*&value) const;

/// revert之前通过get获取的value

int revert(const KeyT &key, ValueT \*&value) const;

/// 显示从cache中删除一个item

void remove(const KeyT &key);

};

#### kv缓存

class ObKVCache

{

/\*\*

\* @fn int KVCache::init(int64\_t cache\_mem\_size, int64\_t max\_non\_active\_usec)

\* 初始化ObjCache

\*

\* @param cache\_mem\_size cache 使用的内存大小, <0表示不限制内存使用

\* @param max\_non\_active\_usec

\* -# >0 单个item最大允许不被访问的时间, 超过该时间item会被淘汰

\* -# =0 按照lru淘汰

\* -# <0 不被淘汰，这种情况下忽略cache\_mem\_size参数

\*/

int init(int64\_t cache\_mem\_size, int64\_t max\_non\_active\_usec);

/// 往cache中增加内容

int put(uint64\_t tableID, const ObString &key, const ObString &value);

/// 通过key获取对应的value, 这里会有一次拷贝

int get(uint64\_t tabletID, const ObString &key, ObString &value) const;

/// 显示从cache中删除一个item

void remove(const ObString &key);

};

##### 循环内存buffer

class ObCircularBuffer

{

public:

/// 初始化循环buffer

int init(int64\_t nbytes);

/// 分配内存

void \*malloc(int32\_t nbyte);

/// 释放分配的内存

int free(void \*ptr);

};

### 二级子系统BlockIndexCache

#### 系统描述

BlockIndexCache中缓存的信息是BlockIndexInfo对象。因此BlockIndexCache实际上是定长对象cache。但是它需要与BlockCache、BlockIndexInfo打叫道，并且需要理解BlockIndexInfo的语义。BlockIndexCache对外仅仅暴露查询接口，内部负责文件的读取，BlockIndexInfo的反串行化，以及BlockIndexInfo的缓存管理。

#### 处理流程

在收到上层查询请求后，BlockIndexCache的处理流程如下：

BlockIndexInfo = ObjCache.get()

if 找ObjCache中找到index info

调用BlockIndexInfo的相应查询接口

return

else

调用BlockCache::pread接口

从ObjCache中分配内存，创建BlockIndexInfo对象

调用BlockIndexInfo::deserialize进行反序列化

调用BlockIndexInfo的查询接口

调用ObjCache::put接口将BlockIndexInfo存放到cache中

endif

#### 对外接口

class ObBlockIndexCache

{

public:

/\*\*

\* @fn int BlockIndexCache::lower\_bound(const uint64\_t sstabe\_file\_id, const char \*file\_name, const ObString &key, int32\_t \*offsets, int32\_t noffset)

\* 查询大于等于rowkey的第一个block在sstable文件中的偏移

\*

\* @param sstabe\_file\_id sstable file id

\* @param file\_name sstable文件名

\* @param key key

\* @param [out] offsets 存储结果的offset数组，可能会同时查询多个offset

\* @param [in/out] noffset offsets数组大小，返回表示实际得到的offset数量

\*/

int lower\_bound(const uint64\_t sstabe\_file\_id,

const char \*file\_name,

const ObString &key,

int32\_t \*offsets,

int32\_t &noffset);

int upper\_bound(const uint64\_t sstabe\_file\_id,

const char \*file\_name,

const ObString &key,

int32\_t \*offsets,

int32\_t &noffset);

/\*\*

\* @fn int BlockIndexCache::next\_offset(int32\_t cur\_offset, int32\_t \*offsets, int32\_t &noffset)

\* 根据当前的offset获得后续n个block的offset

\*

\* @param cur\_offset 当前block在sstable文件中的offset

\* @param [out] offsets offset数组

\* @param [in/out] noffset 输入表示offsets数组的大小，输出表示实际得到的offset数量

\*

\* @return int

\*/

int next\_offset(int32\_t cur\_offset, int32\_t \*offsets,int32\_t &noffset);

};

## 每日合并子系统

### 子系统描述

每日合并子系统负责合并当天的动态数据和本地所有静态数据，生成新的SSTable，并上报新的Tablet信息。

每日合并子系统需要负责管理SSTable文件在磁盘上的位置，生成新的SSTable文件时，需要为其指定存储路径，生成唯一序列号。(新生成的SSTable文件中记入新schema的timestamp,可以在重启的时候进行核对)

每日当RootServer读入新的schema以后，会通知ChunkServer进行当日数据合并，每日合并子系统接收到合并请求以后，会先将新的Schema从RootServer取回来，然后利用Tablet子系统的接口遍历所有的Tablet，查询每个Tablet的所有数据信息（包括静态数据与动态数据），然后调用SSTablet子系统的接口，按新的schema生成新的SSTable文件；每生成一个新的SSTable文件，便将其装载到新的Tablet，加入Tablet子系统中，汇报给RootServer，待所有SSTable合并完成以后，向RootServer报告汇报完成的消息。

初次启动ChunkServer时，子系统将扫描当前最新的SSTable目录，装载新的SSTable文件(这里只需要打开文件，并读入trailer，无需装载index，延迟到查询请求时候进行)，并生成新的Tablets对象，交由Tablet子系统管理，并汇报给RootServer。

### 子系统接口

**class** DayMergeScheduler

{

**public**:

/\*

\* 开始进行每日合并

\* foreach Tablet in Tablets

\* {

\* 读入tablet合并数据

\* 生成新的SSTable文件

\* }

\*/

**int** do\_merge(int timestamp, TabletManager\* manager, **const** SchemaList\* new\_schema\_list);

/\*

\* 扫描新的SSTable文件目录，装载SSTable文件

\*/

**int** scan(**int64\_t timestamp**);

};

### SSTable文件合并

#### 模块描述

文件合并作为一个单独的任务来进行，由RootServer触发，在接收到上层接口子系统发起的命令后，每日合并子系统从RootServer拉回来最新的Schema信息(timestamp)，然后调用Tablet子系统的接口遍历每一个Tablet，查询其全部Range的数据并做合并，让数据逐条写入新的SSTable文件(使用新的schema)。

因为每天合并之前都需要从RootServer拿到新的timestamp，而CS也需要保留这个timestamp查询最新的SSTable文件，判断是否加入集群提供服务。CS会将新生成的SSTable文件放入以这个timestamp为名字的目录中。

#### 模块接口

**class** TabletMergeTask

{

**public**:

**int** run(int timestamp, TabletManager\* manager, **const** SchemaList\* new\_schema\_list);

};

### SSTable文件存储管理

#### 模块描述

ChunkServer上的SSTable文件存储在多个独立的硬盘上，文件存储管理负责管理每块盘上存储的SSTable文件，使用配额；将SSTable文件均匀的分布在各个磁盘上。

存储路径描述：

文件存储管理还负责在系统初始化的时候扫描磁盘上的SSTable文件(当前最新的)，重建整个Tablet表。

#### 模块接口

**class** SSTableFileManager

{

struct DiskUsage

{

int64\_t total\_bytes;

int64\_t use\_bytes;

int64\_t free\_bytes;

int64\_t sstable\_count;

};

**public**:

/\*

\* 扫描当天的所有SSTable文件，用于系统启动时，重建Tablet hashtable.

\*/

**int** scan(int timestamp);

/\*

\* 获取某个磁盘的使用情况

\*/

**int** get\_disk\_usage(int32\_t disk\_id, DiskUsage& disk\_usage);

/\*

\* 分配一个新的sstable\_file\_id,并指定存放路径

\* SSTableFileManager会根据当前的磁盘分配情况，选取合适的路径；

\*/

**int** allocate\_new\_sstable(const ObString& src\_path, uint64\_t& sstable\_file\_id, ObString& desc\_path);

/\*

\* 写入一个新的sstable file

\*/

**int** write\_new\_sstable(int32\_t disk\_id, const uint64\_t sstable\_file\_id, const ObString& path);

};