**UpdateServer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 | 2010-7-28 | 日照\无施\郁白\刘珂 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 系统设计

## 系统综述

UpdateServer是OceanBase的单点更新模块，为了保证可靠性，采用主备实时同步机制。淘宝的许多应用的每天修改量大约千万到亿次级别(高峰数千tps)，每次修改平均百字节，即每天大约1GB~10GB，即使这个量增加5~10倍，每天的更新可以存放到一台机器的内存中，然后选取一段空闲时间进行数据合并。因此，设计中UpdateServer单机服务OceanBase的所有更新请求，暂不支持系统中多个UpdateServer服务不同的数据范围功能。由于UpdateServer 在内存中保存一天的所有更新操作，实现时需要特别注意节约使用内存。

UpdateServer支持OB和DB两种语义，DB语义和数据库相同，OB语义将插入和更改操作模糊化：插入存在行视为更新，更改不存在行视为插入。DB语义实现时需要查询ChunkServer的静态表数据，性能差，外部应用应尽量少使用。UpdateServer采用单线程更新的实现，从而严格保证同一个客户端更新操作的顺序性。

## 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；

**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；

**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）。

写性能：OB语义且无备机，并发量 > 5000/s；OB语义且有一到两台同机房备机，并发量 > 2000/s，并发量为1000时4个9的写操作延时 < 20ms；DB语义性能暂不考虑；

读性能：读取并发量 > 50000/s，4个9的读取操作延时 < 5ms；

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

UpdateServer由多台机器做热备（一主一备或一主两备），提供对外唯一的VIP，绑定在主服务器上，主负责提供所有对外服务，并且实时将数据同步到备机，并保证数据的顺序和一致性。主服务不可用时候，HA将VIP漂移到备机，将备机提升为主。

客户端所有的更新请求，都将由UpdateServer处理，UpdateServer将所有的更新记录写入commit log并同步到备机后再写入内存中。UpdateServer Slave启动后首先和Master同步，同步到一致状态后Master通过Lease机制给Slave授权，此时Slave可以切换为Master。当Master宕机后，HA将VIP漂移到备机，备机检测到VIP和本机IP相同且持有的Lease有效则自动切换为Master对外服务。

UpdateServer通过调用client api读取Chunk Server的静态表数据。

## 网络框架

服务器端，典型的处理流程是：从套接字读取请求；解析请求（反串行化）；处理请求；发送响应。网络框架主要处理网络数据的接收，发送；套接字管理；接收与发送数据缓冲区内存管理。选取网络框架同时也选取了服务器的线程模型。

为了实现上述目标，使用单独的网络线程来管理套接字并处理所有的网络读写，socket采用非阻塞的方式进行读写，使用epoll来管理所有的套接字，称这个线程为网络线程。请求的分析与处理线程称为工作线程。网络线程与工作线程通过消息队列的方式进行通信。这样网络框架与服务器本身的处理逻辑相对独立，而消息队列作为中间件连接处理线程与网络线程，接收与发送消息的缓冲区管理由中间件负责。处理线程可能需要把不同的请求分发的不同的线程(池)进行处理，因此在网络线程与工作线程之间可能存在多个请求队列。因此网络框架包含两部分：消息队列、网络收发线程。网络框架如下图所示：



由上图可以看出，整个框架是一个生产消费者模型。

网络处理线程不断的从套接字中读取request并把request push到接收消息队列中。工作线程不断的从接收消息队列中pop request，在请求处理完成后将响应push到发送消息队列中。网络处理线程同时不断的从发送消息队列中pop response，并把response发送到对应的客户端。在发送response的时候，网络线程需要知道response应该发送给哪个套接字，为了解决这个问题为每个套接字分配一个全局唯一的64位id，请求和响应消息中都必须包含该64位id。

由于对tbnet框架不是很熟悉，又缺少相关文档，因此**设计中暂时没有考虑结合tbnet框架**，在深入了解tbnet后会考虑使用tbnet构建设计中的网络框架。

## 内存管理

对象构造函数中传入内存池，对象运行过程中需要的内存通过传入的内存池分配。UPS运行过程中有三块内存：编程框架相关（包括网络收发及任务队列等），存储结构相关（HashMap结构的Memtable）和工作线程相关。其中，Memtable的内存管理由Memtable内部单独实现，工作线程相关内存使用全局内存池。

全局内存池希望使用定长内存池，假设固定块大小为16K：

1, 如果申请的内存大小 <= 16K，申请16K大小的内存，使用完成后把内存块放回内存池缓存链表；

2, 如果申请的内存大小 > 16K，直接new出比申请的内存块更大的最接近的16K整数倍内存，使用完成后delete。

工作线程相关内存均为临时使用，且工作线程数可控，占用内存稳定且浪费不多。

网络框架部分的任务队列内存框架请参考网络框架模块的设计。

# 模块设计

## 概述

UPS按照功能将模块划分如下：

如上图，各个模块说明如下：

网络模块：处理网络收发相关，单线程收发网络包，与工作线程通过任务队列通信，工作线程从任务队列获取请求任务，执行成功后将结果回复任务放到网络模块的回复队列中。

MemTable：内存数据结构，将UPS的更新操作在内存中组织起来供读取和扫描；

LogMgr：日志模块，负责日志备份，将日志写入到备机后再写入本机；

Lease模块：管理UPS Lease相关信息，通过Lease防止多个Master同时存在的情况；

TabletMgr：子表管理，目前UPS没有根据数据范围划分子表，每个表只有一个子表，每个子表有冻结和活跃两个MemTable。所有读写相关操作流程通过TabletMgr模块操作MemTable。

请求处理流程：每一种任务由一种请求处理函数处理，请求分为两类，读/扫描请求和更新请求，其中更新请求由单线程处理，读/扫描请求由线程池的工作线程处理。更新流程需要先将操作日志通过LogMgr先后写入UPS Slave和Master的磁盘，然后通过TabletMgr模块操作Memtable。冻结/卸载内存表也属于一种更新操作，需要通过操作日志备份到UPS Slave从而保证二者的状态完全一致。

配置管理和监控：辅助模块，配置管理模块用于加载系统配置并提供统一的配置参数获取接口，监控模块定时将监控信息写入磁盘或者其它外部监控系统。

## 网络模块

### 模块描述

由前文网络框架部分的描述我们知道，网络框架部分包括两个组件：消息队列与网络线程。

#### 消息队列

消息队列作为进程间通信的中间件维护FIFO的消息队列，并负责管理存放消息的内存缓冲区。

内存管理，消息队列负责提供存放消息所需要的内存，消息队列的长度一般来说是可以预期，不会太大，因此这部分的内存使用可以进行特定的优化。

消息队列将多个固定大小的内存块组织在空闲链表和已分配链表中。每个固定大小的内存块包含一个引用计数，表示已经在该内存块上分配的小内存的数量。每一块定长的内存块作为一个小的内存池，分配算法就是简单的只分配，不释放，通过向前移动空闲指针来分配内存。每次需要从消息队列中分配内存的时候，首先查看空闲链表头的第一个内存块，如果该内存块中剩余内存足够满足需求，则直接从该内存块中分配；如果该内存块中的剩余内存不够满足需求，则将该内存块从空闲链表中移出并放入已分配链表中，然后从空闲链表的下一块空闲内存中分配需要的内存空间。每次释放内存的时候，只需要将内存所在的存储块的引用计数减1，如果发现引用计数为0，则把该内存块归还空闲内存链表。

#### 网络线程

网络线程接收请求，然后把请求push到消息队列中。在面向流的传输协议中，网络线程必须能够区分不同请求的边界。因此，使用定长头+变长请求数据的通信协议。定长的协议头定义如下：

/// @struct ObHead, tcp protocal, [ObHead][content]

**struct** ObHead

{

uint16\_t version;

/// @property command command number used to identify request type

uint16\_t command;

/// @property client client identifier

**char** client[**sizeof**(int64\_t) + **sizeof**(int32\_t)];

uint32\_t magic;

int32\_t body\_len;

/// @property logid used for request track

uint64\_t logid;

/// @property req\_id

uint64\_t req\_id;

};

其中req\_id是便于客户端使用纯异步的模型：在发出一个请求后不等待服务器端返回就直接复用当前socket发送下一次请求。logid便于追查定位问题。网络线程根据command字段将请求分发到不同请求消息队列，工作处理线程根据该字段判断请求的类型。

对于客户端的一次请求，服务器的处理流程如下：

1, 网络线程首先接收定长的协议头，根据body\_len字段从接收消息队列中分配内存以存放消息，在消息接收完以后将消息push到消息队列中；

2, 工作线程从消息队列中pop任务，并执行任务，然后将响应push到发送消息队列；

3, 网络线程从发送消息队列pop响应，并把响应发送到客户端

每个socket包含唯一的64位id标示。同时为了保证请求或者响应在消息队列中的时候，它所对应的socket不被释放，每个socket必须包含引用计数，表示当前pending在消息队列中的请求的数量，只有任务队列中不存在该socket的pending的请求的时候，socket资源才能够被释放。

从上面的描述可以看到，socket至少包含3个字段：文件描述符、64位id、引用计数。Socket的定义如下：

/// @enum socket state

**enum** ObSockState

{

*LISTEN*,

/// @property READ\_OBHEAD if socket readable, should read ObHead

*READ\_OBHEAD*,

/// @property READ\_MSG if socket readable, should read request

*READ\_MSG*,

/// @property READ\_FINISH socket read finish a single request

*READ\_FINISH*,

*TIMEOUT*,

*BROKEN*

};

/// @struct socket struct

**struct** ObSocket

{

uint64\_t sockid;

int32\_t sockfd;

uint32\_t ref\_num;

/// @property status socket status

int32\_t state;

/// @property 当前state下已经读取的字节数

int32\_t readed\_length;

/// @property 当前响应已经写入套接字多少字段

int32\_t writed\_length;

/// @property 用于读取ObHead的内存缓冲区

ObHead req\_head;

/// @property current buffer used to read request

ObSockMsg \* current\_read\_msg;

/// @property write\_buf\_list list of ObSockMsg

SLINK write\_buf\_list;

};

### 关键数据结构与算法

#### 消息队列内存管理算法

首先看一下消息队列中管理的定长内存块的定义：

/// @struct MemBlock managed by MsgQueue, used to allocate message buffer

**struct** MemBlock

{

/// @property next memblock list pointer

SLINK next;

/// @property offset next available byte

int32\_t offset;

/// @property sub\_block\_num number of sub block allocated on this block

int32\_t sub\_block\_num;

**char** buffer[0];

};

消息队列（MsgQueue）维护两个MemBlock的队列：空闲队列（free\_block\_list）、已经分配的队列（used\_block\_list）。定长内存会的大小为block\_size

分配算法：每次从空闲队列头的定长内存块中分配，如果该内存块能够满足分配需求则直接分配，并增加该内存块的引用计数。如果空闲队列头的定长内存块的剩余内存无法满足分配需求，则把该块放入已分配的定长块队列中，如果此时空闲内存块队列为空，则分配一个新的定长内存块，放到空闲队列中。然后从现在的空闲队列的头指向的定长块中分配内存。

释放算法：查找内存指针所在的定长内存会，将该内存块的引用记数减1，如果内存块的引用计数为0，则把内存会放到空闲内存块列表中。

#### 网络线程mainloop流程

网络线程mainloop流程如下：

调用epoll\_wait监听所有套接字

处理所有发送消息队列中的消息

处理所有可读套接字

处理所有可写套接字

处理所有错误套接字

处理发送消息队列中的消息

while 发送消息队列不为空

获取下一个发送消息

根据sockid定位socket

socket.ref\_num –

/// 套接字处于错误状态

if socket.state == TIMEOUT or socket.state == BROKEN

print warning log

释放消息资源

/// 套接字处于错误状态，且没有pending的任务

If socket.ref\_num == 0

释放socket资源

endif

else 将消息放入socket.write\_buf\_list中

endif

endwhile

套接字读流程

func socket\_read

if socket.state == READ\_HEAD

读取head剩余部分sizeof(ObHead) – socket.readed\_len

socket.readed\_len += 读取的字节数

if socket.readed\_len == sizeof(ObHead)

socket.state = READ\_MSG

socket.readed\_len = 0

从消息队列中分配读取消息需要的内存

socket\_read()//递归调用

endif

elseif socket.state == READ\_MSG

读取msg剩余部分 ObHead.body\_len - socket.readed\_len

socket.readed\_len += 读取的字节数

if socket.readed\_len == ObHead.body\_len

socket.state = READ\_HEAD

socket.readed\_len = 0

将读取的请求push到请求消息队列

socket\_read() //递归调用

endif

endif

处理所有可读套接字

for each readable socket

/// 监听套接字

if socket.state == LISTEN

accept

create new\_socket

new\_socket.ref = 1

new\_socket.state = READ\_HEAD

new\_socket.readed\_len = 0

将socket放入epoll监听队列中

continue

endif

socket\_read()

endfor

处理所有可写套接字

for each writable socket

while head(socket.write\_buf\_list) != NULL

send当前write buffer所有未写的数据

if current\_write\_buffer写完成:

pop\_front(socket.write\_buf\_list)

释放current\_write\_buffer到发送消息队列

endif

endwhile

endfor

处理所有错误的套接字（这里的错误套接字包含超时的套接字）：

for each error socket:

close(socket)

socket.state = BROKEN

socket.ref\_num --

for each msg in socket.write\_buf\_list:

print warning log

释放msg资源

endfor

if socket.ref\_num == 0:

释放socket资源

endif

endfor

在读写过程中如果遇到套接字错误，处理流程与上述流程相同。

### 模块对外接口

#### 消息队列对外接口

/// @struct MsgQueue管理的msg

**struct** ObMsg

{

/// @property slink used to link

SLINK slink;

/// @property buffer\_size variable buffer size

int32\_t buffer\_size;

/// @property magic magic number, used for debug, memory overwrite

uint32\_t magic;

**char** msgbuf[0];

};

/// @class ObMsgQueue message or task queue

**class** ObMsgQueue

{

**public**:

/// 分配空闲的message，用于存放准备的message内容

ObMsg \***allocate\_empty\_msg**(**const** int64\_t msg\_size);

/// 往消息队列中enqueue msg，其中msg通过allocate\_empty\_msg获取

**int** **enqueue\_msg**(ObMsg \*msg, **const** **bool** high\_priority);

**public**:

/// 从消息队列头获取下一个消息，不需要用户提供缓冲区

ObMsg\* **dequeue\_msg**(**const** int64\_t timeout\_usec);

/// free a msg dequeued by calling dequeue\_msg or msg allocated by

/// calling allocate\_empty\_msg but not enqueued by calling enqueue\_msg

**int** **free\_msg**(ObMsg \*msg);

**public**:

/// 往消息队列中enqueue msg，其中msg是用户自己管理的缓冲区

**int** **enqueue\_msg**(**const** **char** \*msg, **const** int64\_t msg\_size, **const** **bool** high\_priority);

/// 从消息队列头获取下一个消息，msg\_buffer的缓冲区由用户管理

**int** **dequeue\_msg**(**char** \*msg\_buffer, int64\_t &msg\_buf\_size, **const** int64\_t timeout\_usec);

};

#### 网络线程对外接口

/// @struct message in ObNetworkMan's message queue

**struct** ObSockMsg

{

uint64\_t sockid;

/// @property request\_head for debuging and logging

ObHead request\_head;

**char** request[0];

};

/// @class ObNetworkMan, include network thread, task queue

**class** ObNetworkMan

{

**public**:

/// @typedef dispatch request according to ObHead.command

**typedef** **int** (\*req\_dispatch\_func\_t)(**const** ObHead &head);

/// 初始化

**int** **init**(**const** int32\_t request\_queue\_num = 1, **const** req\_dispatch\_func\_t dispatch\_func = NULL);

/// 启动网络线程

**int** **run**(**const** int32\_t epoll\_timeout\_msec, **const** int32\_t sock\_read\_timeout\_msec,

**const** int32\_t sock\_write\_timeout\_msec, **const** int32\_t sock\_idl\_timeout\_msec);

/// 设置退出标志

**int** **stop**();

/// 获取消息队列中下一个请求

ObSockMsg \* **pop\_request**(**const** **int** queue\_id = 0, **const** int64\_t timeout\_usec);

/// free request got by dequeue\_request

**int** **free\_request**(ObSockMsg \*request);

/// 分配发送消息使用的缓冲

ObSockMsg \***allocate\_response\_sockmsg**(**const** int64\_t buf\_size);

/// free那些通过allocate\_response\_sockmsg但是确没有调用send\_response发送的ObSockMsg

**int** **free\_unsent\_response**(ObSockMsg \* response);

/// 将需要发送的消息放到发送消息队列中

/// @param high\_priority 为true会把响应放到发送消息队列的头

**int** **send\_response**(ObSockMsg \* response, **const** **bool** high\_priority);

};

## MemTable

### 模块描述

memtable是updateserver用来保存和查询数据修改的内存结构，每次数据修改都将在memtable中产生一条记录，在查询的时候将memtable中针对同一个rowkey所有修改记录与chunkserver中的静态数据合并之后得到最终查询结果。memtable使用单行粒度的原子操作在保证多线程并发读取和单线程更新。

### 关键数据结构和算法

**一、概述：**

考虑到实现的复杂度，初期我们将使用一个固定桶个数的hashmap来保存memtable，因此对于rowkey前缀查询的支持需要通过额外的索引来实现。并且memtable不解释数据的具体格式，数据修改操作将做为多个mutator对象存储在memtable中。

为了保证逻辑清晰，每个table在updateserver中对应两个memtable，一个活跃表一个冻结表，updateserver提供接口接受rootserver命令释放冻结表。

**二、对范围查询的支持：**

updateserver需要活跃memtable支持客户端指定rowkey前缀的查询，以及冻结memtable支持chunkserver指定全rowkey范围的查询，在以后使用B树的情况下可以自然的支持范围查询，目前使用hash的话memtable采用两个方法对这个两种查询进行支持，入下图所示：

1、hashtable记录的链式存储，对于rowkey前缀相同的记录，使用一个双向链表串联起来(下称前缀链表)，每次插入hashtable同时根据rowkey前缀查询外部索引，找到并更新这个链表。链表头尾指针分别指向第一个插入的记录和最后一个插入的记录，在按照rowkey前缀查询的时候，遍历链表即可。

2、活跃表外部索引，为了减少更新线程对读线程的影响，使用一个定长hash保存rowkey前缀到上述链表头尾指针的映射，在指定rowkey前缀查询的时候，先查询这个索引，定位到链表指针，再遍历链表获取数据即可。

3、冻结表排序，为了支持chunkserver指定前缀范围的查询，需要对冻结表按照rowkey进行排序，如果有map结构的话可以使用map，否则就使用排序+二分查找。

**三、更新事务的支持：**

单个memtable接受一个mutator中多个cellmutator的原子性提交，在涉及多行更新的情况下使用memtable粒度的读写锁保证原子性，在大部分操作只是对单行更新的情况进行优化，使用行粒度的锁保证原子性，同时不阻塞其他行的读取操作。

**四、内存管理：**

memtable内存管理，用一个简单的页链表来实现，每个页是一个大小固定的BUFFER，每次分配内存的时候，只需移动这个页的偏移就好了，如果当前页内存不能满足此次内存分配操作，就向系统申请一个新的页。为每一个memtable建立一个这样的内存分配器，当冻结表被删除时，删除对应的内存分配器。Memtable的clean操作直接将固定大小的内存块链表还回系统。

mutator数据的存储：由于ObCellMutator中带有tablename和columnname等冗余信息，在memtable中存储的mutator对象只需要保留ObCellMutator中必要的数据更新信息，因此我们在序列化mutator的时候只保存其中必要的信息，并且尽量压缩存储空间。

**五、mutator代理**

memtable逻辑上由mutator\_proxy和memtable\_base组成，

我们希望底层的memtable\_base提供相对简单的mutator存储和查询，而不去关心mutator的具体内容，但是mutator在存储到memtable之前仍需要进行一些必要的处理，这些处理与mutator结构的耦合度比较高，因此我们抽象mutator\_proxy接口对memtable进行封装，并做一些mutator的预处理和查询结果返回前的处理。

memtable\_base是一个以rowkey为key，ObTinyCellMutator链表和RowStat为value的映射表，提供insert，update，delete，get，scan功能，其中get和scan返回结果为迭代器。

mutator\_proxy对于更新操作将在memtable\_base接口之前被调用，输入是ObCellMutator输出是ObTinyCellMutator和memtable\_base操作类型；对于查询操作将在memtable\_base接口之后被调用，输入ObTinyCellMutator输出是ObCellMutator。它需要实现的功能包括：

1、ob/db语义解析，判断是否需要查询chunkserver（输入预处理）

2、区分不同语义的不同操作分别转发给memtable的接口（输入预处理）

3、测试ObCellMutator携带的逻辑是否可以在memtable中正确更新

4、delete操作转化为set null（输入预处理）

5、填充sysdate字段（输入预处理）

6、ObCellMutator与ObTinyCellMutator的互转（输入预处理/输出预处理）

**六、memtable迭代器**

所有对memtable的查询都将以迭代器的方式返回，每个迭代器指向一个mutator，在指定rowkey查询的情况下，可以迭代当前行的所有mutator；在范围查询的情况下，可以迭代查询范围内所有行的mutator，迭代的结果按行聚合。

迭代器返回的结果可以是直接指向memtable内部空间的ObTinyCellMutator的指针，也可以经过mutator\_filter处理后返回一个ObCellMutator的拷贝。

**七、更新操作流程：**

为了支持ob和db两种语义，在memtable中为每一行维护一个RowStat状态标志，包括如下4个状态：

**DB\_UNKNOW**

没有查询过chunkserver(默认状态)

**DB\_EXISTS**

记录存在(查询chunkserver结果为记录存在，或已执行过OB语义的insert)

**DB\_DELETED**

记录已删除(已经对该行执行过delete)

**DB\_NOT\_EXISTS**

记录不存在(查询chunkserver结果为记录不存在，并且没有对该行执行过insert)

RowStat状态转移如下：

**1、判断本次mutator是否需要查询chunkserver：**

对于schema中含有sequence字段，并且操作为insert的情况下，都需要查询chunkserver，这一逻辑的判断由TabletMgr实现，memtable不需要关心。

memtable对外提供**need\_query**接口，接受一个ObCellMutator为参数，根据这个mutator并查询memtable中保存的行状态来判断本次插入的mutator是否需要查询chunkserver。server之后调用memtable对外提供的**set\_row\_exist**接口，设置chunkserver的查询结果。冻结表在被释放之后，它缓存的RowStat也随之丢失，需要重新查询chunkserver。

**2、insert操作：**

ob语义：

1、获取memtable更新互斥锁

2、查询或创建rowkeyPrefixHash索引，获取前缀链表，增加引用计数

3、**在hashtable提供的原子操作中完成(读写锁保护)**：将mutator存入memtable\_base，同时尝试与已存在的mutator进行merge；更新hashvalue中的链表指针

4、更新行的RowStat为DB\_EXISTS

**注意：**如果表中含有sequence字段，则ob语义也要按照db语义执行insert，sequence逻辑的处理由上层tablemgr生成一个修改sequence的mutator实现。

Db语义：

1、获取memtable更新互斥锁

2、检查RowStat，如果为DB\_EXISTS或DB\_UNKNOW则直接返回失败

3、如果RowStat为DB\_DELETED或DB\_NOT\_EXISTS，则可以执行ob语义的insert

**3、update操作：**

ob语义：

1、获取memtable更新互斥锁

2、在hashtable提供的原子操作中完成 ：将mutator存入memtable\_base，同时尝试与已存在的mutator进行merge

db语义：

1、获取memtable更新互斥锁

2、检查RowStat，如果为DB\_DELETED，DB\_NOT\_EXISTS或DB\_UNKNOW，直接返回失败

3、如果RowStat为DB\_EXISTS则可以执行ob语义的update

4**、delete操作：**

ob语义：

1、获取memtable更新互斥锁

2、查询rowkeyPrefixHash索引，引用计数减一，当减到0的时候删除这个索引

3、**在hashtable提供的原子操作中完成(读写锁保护)** ：清空memtable\_base中rowkey对应的mutators；加入一个delete mutator；更新hashvalue中的链表指针

4、更新行的RowStat为DB\_DELETED

db语义：

1、获取memtable更新互斥锁

2、检查RowStat，如果为DB\_DELETED，DB\_NOT\_EXISTS或DB\_UNKNOW，直接返回失败

3、如果RowStat为DB\_EXISTS则可以执行ob语义的delete

**5、get和scan操作**

get操作将直接查询rowkey的hashtable，scan操作在活跃表中将先查询rowkeyPrefix的hashtable，在冻结表中将直接查询排好序的rowkey的map。

对memtable\_base中数据的迭代将利用hashtable提供的单行粒度的读写锁来保证多线程安全。

### 模块对外接口

**class** UpsMemtable

{

**public**:

// 将指定rowkey的更新操作记入memtable，要获取更新互斥锁

// 在rowkey不存在的情况下只有insert操作返回成功

// @param [in] mutator 对指定rowkey的cell的更新

// @return 0 成功

**int** **set**(**const** ObMutator &mutator);

// 根据mutator判断其更新逻辑是否可以更新到memtable

// @param [in] ObCellMutator mutator

// @return true 可以正确执行

// false 不确定或不可以正确执行

**bool** **check\_semantic**(**const** ObCellMutator &mutator) **const**;

// 根据mutator判断其更新逻辑是否需要查询chunkserver

// @param [in] ObCellMutator mutator

// @return true 需要查询chunkserver

// false 不需要查询chunkserver

**bool** **need\_query**(**const** ObCellMutator &mutator) **const**;

// 设置cs查询结果，要获取更新互斥锁

// @param [in] rowkey 行号

**int** **set\_row\_exist**(**const** ObString &rowkey, **bool** exist);

// 根据rowkey获取对这一行数据的所有mutator迭代器

// @param [in] rowkey 行号

// @param [out] ObCellMutatorInterator 迭代器

// @return 0 成功

**int** **get**(**const** ObString &rowkey, ObCellMutatorInterator &iter) **const**;

// 范围查询获取多行的mutator的迭代器

// @param [in] rowkey 行号

// @param [in] BorderFlag 范围查询参数

// @param [out] ObCellMutatorInterator 迭代器

// @return 0 成功

**int** **scan**(**const** ObString &rowkey, **const** BorderFlag &borderflag, ObCellMutatorInterator &iter) **const**;

// 建立排序索引，要获取更新互斥锁

// 不可重入，调用完成之后的修改操作都将失败

// @return 0 成功

**int** **create\_index**();

// 获取当前数据的bloomfilter

// @param [out] bloomfilter

// @return 0 成功

**int** **get\_bloomfilter**(ObBloomfilter &bloomfilter) **const**;

**private**:

// 保证更新操作互斥的锁

pthread\_mutex\_t update\_mutex\_;

};

**class** ObCellMutatorInterator

{

**public**:

// 迭代器自加

// 迭代结束后变为空

// @return 0 迭代成功

// -1迭代结束或出错

**int** **increase**();

// 获取迭代器指向的ObTinyCellMutator的指针（内部空间指针）

// 可以用于反序列化ObCellMutator

// @return 非空指针 成功

**const** ObTinyCellMutator \***operator ->** () **const**;

// 将迭代器指向的ObTinyCellMutator转化为ObCellMutator返回

// @return 0 成功

**int** **get\_cell\_mutator**(ObCellMutator &cellmutator) **const**;

};

## Commit Log

### 模块描述

为了保证可靠性，采用主备实时同步方案（一主一备或者以主两备），UPS Master将更新操作实时同步到所有的UPS Slave机器后记录本地日志并应用到MemTable中。UPS Slave启动后首先和Master同步，同步到一致状态后Master通过Lease机制给Slave授权，此时Slave可以切换为Master。

为了保证更新的时序性，UPS更新采用单线程的设计，一次性从UPS任务队列读取MB级别大小的尽可能多的更新操作批量写到Slave和本机操作日志。每个操作日志增加一个crc检验码以提高可靠性。

UPS逻辑上有一个全局的commit log文件，物理上可以按照大小对commit log进行切分。日志文件直接采用64位无符号整数命名。

新的Slave接入时，首先向Master注册，Master切换日志文件并将当前最大的日志编号文件编号回复Slave，实现时需要保证切换日志文件和获取当前最大日志编号的原子性。切换日志文件后新的修改的操作日志发往Slave，Slave接收新修改的操作日志同时还需要获取之前落后的操作日志文件。

当Slave和Master的日志完全同步后，Slave需向Master申请获取租约（租约有效期N秒），Master根据Slave是否存在于操作日志同步机器列表内确定是否授权，取得授权的Slave机器在租约有效期内可以提供读服务或切换为Master。租约有效期必须大于HA VIP漂移时间，当租约有效时间缩减至接近HA VIP漂移时间时，Slave向Master重新发起租约申请。Master往Slave同步操作日志过程中如果发生错误，将Slave从同步机器列表中移除，以后Slave的租约申请命令都将被拒绝。

Master除了将正常的写操作日志同步到Slave，还需要同步冻结Memtable和卸载Memtable指令。机器宕机时需要从日志回放点开始恢复，为了减少日志回放的数量，每次卸载Memtable后可以将日志回放点移动到上次冻结Memtable处。有一个触发器定时执行，根据日志回放点删除不再使用的操作日志文件。

### 关键数据结构和算法

#### **UPS Master**

写日志：

// 写入批量操作日志，日志类型可能为一般的修改或者freeze/drop memtable操作；

**int** **write\_log**(**const** int64\_t batch\_num, **const** ObUpsMutator\* mutators, ObCommitLogInfo\* infos);

1, 计算mutators的序列化长度，并根据计算结果申请缓冲区；

2, 通过mutators的serialize方法将mutators逐一序列化到申请的缓冲区；

3, 对于同步机器列表的每台Slave机器：

3.1, 将日志缓冲区的数据通过tcp socket发送到这台机器；

3.2, 如果发送失败，在Lease Mgr把这台机器剔除同步机器列表，拒绝这台机器后续的租约申请请求；

4, 加互斥锁锁住当前日志文件号更新；

5, 将日志缓冲区的数据写入当前日志文件(cur\_log\_file\_id)；

6, 如果当前日志文件太大，切换文件，即++cur\_log\_file\_id；

7, 解除日志文件号更新互斥锁；

切换日志文件：

// 主动切换日志文件，Slave注册到Master时，Master先主动切换日志文件并并将当前最大的日志文件号告知Slave

// @param [out] max\_log\_file\_id 切换日志文件时刻的最大的文件号

// @param [out] log\_file\_size 切换日志文件时刻最后一个文件大小，Slave通过该值判断最后一个日志文件回放是否结束

**int** **switch\_log**(int32\_t& max\_log\_file\_id, uint32\_t& log\_file\_size);

1, 加互斥锁锁住当前日志文件号更新；

2, 自增当前日志文件编号：++cur\_log\_file\_id;

3, 记录当前日志文件编号：max\_log\_file\_id = cur\_log\_file\_id;

4, 解除日志文件号更新互斥锁；

#### **UPS Slave**

UPS Slave操作日志相关有三个线程：

* 注册线程：

UPS Slave启动后需要向Master注册，注册线程执行流程如下：

1, 向Master发送注册命令，等待返回，超时则重试若干次；

2, 解析Master的回复包，取出cur\_log\_file\_id，通过wget或其它方式从Master获取落后的commit log文件；

3, 全部落后的commit log获取完成后注册线程退出；

* 日志同步线程：

Slave监听一个日志同步端口，Slave向Master注册，Master切换日志文件后将操作日志同步到Slave，日志同步线程不断接收Master的操作日志并写入Slave本地操作日志文件；

* 日志回放线程：

日志回放线程按顺序回放commit log，如果没有最新的commit log或者操作日志出现空洞（部分落后的数据没有拷贝完成），回放线程sleep一段时间（ms级别）后重试；当日志回放线程把cur\_log\_file\_id之前落后的日志文件回放完成时（通过Master发送的最后一个日志文件大小判断回放是否完成），需要生成一个内部任务，从Master申请租约；

### 模块对外接口

// Log sequence号

**struct** ObLogSequence

{

uint32\_t file\_id;

uint32\_t offset;

};

// 写每条操作日志返回的结果信息

**struct** ObCommitLogInfo

{

ObLogSequence log\_id;

**int** err;

};

// Ups内部的Mutator，可能是一般的写操作，也可能是冻结或者卸载Memtable

**class** ObUpsMutator

{

**public**:

**void** **set\_version**(**const** int32\_t version);

int32\_t **get\_version**() **const**;

**public**:

ObMutator& **get\_mutator**() **const**

{

**return** mutator\_;

}

// 设置为冻结memtable操作

**void** **set\_freeze\_memtable**();

// 设置为卸载memtable操作

**void** **set\_drop\_memtable**();

**bool** **is\_freeze\_memtable**() **const**;

**bool** **is\_drop\_memtable**() **const**;

**public**:

// 序列化/反序列化接口

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

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

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

**private**:

int32\_t version\_;

ObMutator mutator\_;

int64\_t freeze\_flag\_;

uint64\_t table\_id\_;

};

// 操作日志写入类，通过它写入操作日志

**class** ObCommitLogMgr

{

**public**:

// 写入批量操作日志，日志类型可能为一般的修改或者freeze/drop memtable操作；

// @param [in] batch\_num 操作个数

// @param [in] mutators 写操作数组

// @param [out] infos 每个写操作日志的返回信息，包括log\_id等；

**int** **write\_log**(**const** int64\_t batch\_num, **const** ObUpsMutator\* mutators, ObCommitLogInfo\* infos);

// 主动切换日志文件，Slave注册到Master时，Master先主动切换日志文件并并将当前最大的日志文件号告知Slave

// @param [out] max\_log\_file\_id 当前的最大日志文件号

// @param [out] log\_file\_size 最后一个日志文件的大小

**int** **switch\_log**(int32\_t& max\_log\_file\_id, uint32\_t& log\_file\_size);

// 设置日志回放点

// @param [in] log\_replay\_point 日志回放点

**int** **set\_log\_replay\_point**(**const** ObLogSequence& log\_replay\_point);

// 获取日志回放点

// @param [out] log\_replay\_point 日志回放点

**int** **get\_log\_replay\_point**(ObLogSequence& log\_replay\_point);

**public**:

// 获取最大的log sequence号

**void** **get\_max\_log\_sequence**(ObLogSequence& max\_log\_id);

**public**:

// 回收commit log空间

**int** **gc**(**void**);

};

// 操作日志读取类，通过它来实现日志重放

**class** ObCommitLogReader

{

**public**:

// 从操作日志中迭代读取一个更新操作

// @param [in] last\_log\_seq 当前操作的log seq

// @param [out] mutator 获取的更新操作

// @param [out] next\_log\_seq 下一个操作的log seq

**int** **get\_next\_mutator**(**const** ObLogSequence& last\_log\_seq, ObUpsMutator& mutator, ObLogSequence& next\_log\_seq);

};

## Lease模块

### 模块描述

UPS Slave是UPS Master的实时备份，当UPS Master宕机时，UPS Slave切换为Master。为了避免网络分区导致同时有两个Master的情况，引入Lease机制。UPS Slave向Master注册后同步操作日志，同步完成时需要向Master申请Lease，Lease有效期大于HA VIP漂移时间。当Lease剩余时间减少至接近HA VIP漂移时间时，UPS Slave重新请求Master更新Lease。

Master和Slave有一个单独的线程定时检测租约：

1, 对于Master：如果租约没有过期且VIP在自己身上，延长租约的时间；Slave请求更新租约时，如果Slave在同步机器列表中则延长Slave的租约，否则拒绝更新租约请求；Master往Slave同步操作日志失败将把Slave从同步机器列表中剔除；

2. 对于Slave：如果租约快要过期（Lease剩余时间减少至接近HA VIP漂移时间），请求Master更新Lease；如果租约没有过期且VIP在自己身上，切换为Master；

无论是UPS Master还是Slave，Lease过期都不能提供服务而选择优雅退出并报警。

当满足下列条件之一时，UPS Slave可以切换为Master：

* 管理员强制一台update server从不可用状态变成可用状态, 强制使这台update server成为主，管理员一定要保证VIP在当前机器自己身上, 并且Commitlog的sequences是系统中最大的(不然可能会丢数据)
* Slave发现自己的租约没有过期，并且VIP在自己身上

### 模块对外接口

// 租约管理类

**class** ObLeaseMgr

{

**public**:

// 检查Lease：

// 1, 如果租约过期，设置exit标志，程序优雅退出并报警；

// 2, 如果租约没有过期：对于Master，如果VIP在自己身上，延长租约时间；

// 3, 如果租约没有过期：对于Slave，如果租约剩余时间减小至接近HA VIP漂移时间，向Master申请更新租约；

**int** **check\_lease**();

// Master给Slave更新租约，如果Slave不在同步机器列表则拒绝租约申请

**int** **grant\_lease**(**const** ObServer& server);

// Slave向Master申请更新租约；

**int** **renew\_lease**();

**public**:

// 判断租约是否合法，读写请求前调用

**bool** **is\_lease\_valid**(**void**) **const**;

};

## TabletMgr

### 模块描述

模块主要负责管理tablet：memtable的freeze、drop；freeze表、active表的结果merge；active与freeze表的切换。该模块仅负责管理内存数据结构，不涉及与其他任意模块交互。这样的假设意味着，对于数据库语义的操作，TabletMgr认为在执行更新操作的时候，TabletMgr中已经包含了chunkserver上关于该记录是否存在的信息。

### 关键数据结构和算法

#### tablet组织方式

在TabletMgr中，需要将所有的表格组织起来，便于通过tableid定位相应的表格。这里假设表的总数量不多，并且添加删除表格的操作是及其不频繁的操作。基于上述假设，我们按照tableid有序的方式将table组织在一个数组中，数组中存放指向table的指针，数组初始大小为1024，每次以乘以2的速度进行扩张，数组的内存并不减小，因为每个表格占用8个字节的内存空间，8M大小的数组就可以管理100w个表格，完全没有内存问题。

处理逻辑不会对表格id进行去重操作，假设表格id是递增且不复用的。

通过tableid定位表格就是简单的二分查找，表格的增加和删除是先查找，接着删除表格，最后移动数组剩余的元素保证数组元素的有序性。

在当前实现中一个tablet就是一个table，定义如下：

/// @struct TableInfo

**struct** TableInfo

{

/// @property tableid of the table

uint64\_t table\_id;

/// @property freezed table and active table

MemTable \*table[MEMTABLE\_NUM];

/// @property sequence id allocator

uint64\_t sequence\_allocator;

/// @property sequence column id, <0 means table do not contain sequence column

int64\_t sequence\_column\_id;

pthread\_rwlock\_t rwlock;

};

#### 更新处理流程

通过tableid定位TableInfo

/// 处理sequence

if TableInfo.sequence\_column\_id > 0 and rowkey 不存在:

TableInfo.sequence\_allocator ++

构造一个Insert类型的ObCellMutator

调用ObMutator::add\_cell\_mutator将生成的ObCellMutator添加到ObMutator中

endif

MemTable::begin\_mutation()

for each ObCellMutator in Mutator

MemTable::set()

Endfor

MemTable::end\_mutation()

#### 查询流程

查询流程每次从GetParam中取出最多给定数量的cell，这些要求属于同一个table的同一行，这样可以减少对memtable中同一行Mutator list中的节点的遍历次数。因此，如果GetParam中的cell按照行进行聚集，对于UPS的实现来说是最高效的。

for (it = GetParam.begin(); it != GetParam.end(); )

/// 每次最多处理给定数量的cell (64)，要求这些cell属于同一个表的同一行

output\_info[64]

cur\_cell\_num = 0

cur\_tableid = it->tablid

cur\_rowkey = it->rowkey

output\_info[cur\_cell\_num].column\_id = it->coumn\_id

it ++

cur\_cell\_num ++

while it->table\_id == cur\_tableid and it->rowkey == cur\_rowkey and cur\_cell\_num < 64

output\_info[cur\_cell\_num].column\_id = it->column\_id

it++

cur\_cell\_num ++

endwhile

MemTable.get(cur\_rowkey, cellmutatoriterator)

遍历cellmutatoriterator，如果当前列在output\_info中，则把mutator追加到相应output\_info数组元素的结果中

遍历output\_info，输出最终结果

endfor

#### scan流程

通过tableid定位TableInfo

tableid = ScanParam的tableid

ObCellMutatorIterator cellmutatoriterator

MemTable::scan(rowkey, ranginfor, cellmutateriterator)

调用GetParam::get\_columns获取查询要获取的列数组column\_ids

for each cellmutator in cellmutatoriterator

if cellmutator.column\_id in columnids

将当前的cellmutator追加到结果中

endif

endfor

对于查询流程和scan流程，这里都没有描述存在freezetable的情况，因为在存在freezetable的情况下，只是需要相同的流程现在freezetable上执行一遍，然后再到activetable上执行。

#### freeze流程

for all tablet in TabletMgr

TableInfo.table[FREEZED\_TABLE] = TableInfo.table[ACTIVE\_TABLE]

TableInfo.table[ACTIVE\_TABLE] = new MemTable

endfor

这里发起异步任务执行索引生成是为了防止索引生成的操作阻塞更新。

#### drop流程

drop流程就是简单的把所有freezed的table释放掉。需要说明一点的是，在freezedtable被drop掉以后所有缓存的chunkserver上关于某个key是否存在的信息就丢失了，需要二次查询。

### 模块对外接口

/// @enum MemTableIdx identify freezed table index and active table index

**enum** MemTableIdx

{

*FREEZED\_TABLE*,

*ACTIVE\_TABLE*,

/// @property number of memtable for each real table

*MEMTABLE\_NUM*

};

/// @class UpsTalbletMgr manage all memory tables

**class** UpsTabletMgr

{

**public**:

/// 增加一个tablet

**int** **add\_tablet**(**const** uint64\_t table\_id, **const** Schema & table\_schema);

/// 删除一个tablet

**int** **delete\_tablet**(**const** uint64\_t table\_id);

/// freeze所有的tablet

**int** **freeze\_tablets**();

/// drop 所有的freezed tablets

**int** **drop\_freezed\_tablets**();

/// 更新接口

**int** **batch\_apply**(**const** int64\_t batch\_num, **const** ObMutator \* mutators,

ObOperationResult \*operations);

/// 设置key从chunkserver获取的信息

**int** **set\_exist**(**const** uint64\_t table\_id, **const** ObString & key, **bool** exist);

/// 查询接口

**int** **batch\_get**(**const** int64\_t batch\_num, **const** ObGetParam \*get\_params,

ObScanner \*scanners, ObOperationResult \*operations) **const**;

/// 遍历接口

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

/// 获取bloomfilter

**int** **get\_bloom\_filter**(**const** uint64\_t table\_id, ObBloomFilter &filter) **const**;

/// 是否需要查询chunkserver

**bool** **need\_query**(**const** uint64\_t table\_id, **const** RowKey & key) **const**;

// 设置冻结时的日志点，drop时需要将日志回放点设置为上一次冻结的日志点从而减少日志回放量

// @param [in] freeze\_point 冻结时日志回放点

**void** **set\_freeze\_point**(**const** ObLogSequence& freeze\_point);

// 获取日志回放点

// @param [out] freeze\_point 上次冻结的日志回放点

**void** **get\_freeze\_point**(ObLogSequence& freeze\_point) **const**;

};

## 请求处理流程

### 模块描述

### 关键数据结构和算法

#### 更新请求处理流程

更新请求包括：update、insert、delete、freeze、drop。处于以下两个目的：

1. 日志合并，减少写日志对性能的冲击
2. chunkserver查询合并，减少UPS和chunkserve之间网络通信的代价

我们会将多个更新请求进行合并。这里会产生3个配置项：更新合并最大允许等待时间、更新合并最大请求数据量之和、最大合并更新请求的数量。

更新流程分为以下几个阶段：

1. 更新合并

while (等待时间<配置等待时间 and 更新请求的数据量 < 配置最小更新合并数据量 \

and 已经获取的请求数量 < 配置的单次处理请求数量）

从请求消息队列中获取下一个请求（设置阻塞时间）

endwhile

/// 检验主从状态

调用LeaseManager check主从状态

if 当前updateserver状态为从 and 命令不为freeze and 命令不为drop

设置所有请求结果为失败

调用ObNetworkMan::send\_response发送请求

调用ObNetWorkMan::free\_request释放请求资源

执行结束

endif

1. 检查schema，更新exist缓存信息

定义ChunkGetParam

/// 检测schema合法性

for each request in all request but not requested

将request反序列化为Mutator

调用ObNetworkMan::free\_request释放请求资源

for each CellMutator in Mutator

调用SchemaManager检查更新schema的合法性

if checkfail

设定当前request的处理结果为fail

break

endif

调用TabletManager::need\_query

if need\_query

设置GetParam

endif

endfor

if 当前请求的处理结果为失败

调用ObNetworkMan::send\_response发送响应

endif

endfor

/// 向chunkserver发送查询请求

调用client API向chunkserver发送数据库语义更新的查询请求

for each response from chunkserver

根据结果调用TabletManager::set\_exist设置chunkserver缓存信息

endfor

1. 检查数据库语义

/// check 数据库语义

for each request in all request but not requested

for each CellMutator in Mutator

调用SchemaManager检查db语义

if checkfail

设定当前request的处理结果为fail

break

endif

endfor

if 当前请求的处理结果为失败

调用ObNetworkMan::send\_response发送响应

endif

endfor

1. 写（同步）日志

/// 进行日志同步

调用LogManager::writelog进行日志同步

1. 应用到内存

if 日志同步成功

for each not responsed request

调用TabletManager::batch\_apply 执行更新

if 更新失败

设置全局退出标志

endif

/// drop和freeze的处理见后续描述

endfor

else

设置所有请求结果为失败

end

调用ObNetWorkMan::send\_response发送响应

freeze命令和drop命令在整体上与所有的更新命令处理逻辑相同，唯一不同的地方是在日志同步成功后的处理逻辑，下面将详细描述这部分的逻辑。

#### freeze处理流程

freeze命令会导致产生两个耗时较长的动作：生成排序索引、向metaserver 得到shcema。为了防止这两个耗时长的操作阻塞更新，它们都通过异步的方式执行。在写日志成功后，freeze的处理流程如下：

调用TabletManager::set\_freeze\_point设置freeze点

发起异步任务在所有freezed tablets上建立有序的索引

发起异步任务向rootserver更新schema

对于freeze命令的响应需要等到异步的索引建立成功以后才能够返回命令发起的客户端。

#### drop处理流程

drop命令的执行流程为：

调用TableManager::get\_freeze\_point获取freeze点

调用LogManager::set\_replay\_point设置上次冻结时的日志回放点

这里没有处理删除日志的问题，日志的删除工作由LogManager定时完成

#### 查询请求处理流程

查询请求处理流程相对简单

while 全局结束标志为工作状态

调用ObNetworkMan::get\_request获取下一个请求

执行反序列化

调用ObNetworkMan::free\_request释放请求

调用TabletManager::batch\_get执行查询查询

调用ObNetworkMan::send\_response发送响应

endwhile

## 统计监控

### 模块描述

监控信息包括但不限于：

Ip地址：

绑定的VIP的记录

slave，master的自身的ip记录，物理地址记录

日志同步：

Master和slave最大日志号，slave落后的日志条数

超时重试次数

一天产生新的日志数量，所占磁盘空间大小

对于client：

读取/扫描/更新操作的个数及带宽占用

每天从client发来的请求中，数据库语义更新次数，ob语义更新次数，数据库语义删除次数统计，ob语义删除次数统计。

每天从client发来的请求中，不同错误类型的请求个数（如schema非法，超时）

对于Chunk server：

每天从Chunk server发来的请求数，应答这些请求所产生流量大小，应答这些请求的失败统计。

每天从update server向Chunk server发出的请求数，应答失败统计。

对于update server：

对内存的使用：

每个table的记录数目，该table的hash表占用的内存大小,该表的大小变动情况（如每天该表大小增长（将少）多少G，记录增加（减少）多少条），有多少线程在同时读写这个table

内存中表的大小，冻结表的大小，被成功释放的冻结表大小

查询队列任务统计，内存空间使用情况

更新队列任务统计，内存空间使用情况

一天序列化和反序列化失败统计

磁盘流量

磁盘使用率

磁盘句柄数

磁盘上的文件数

磁盘磁道，扇区数量统计，坏道统计

磁盘文件格式

网络线程，工作处理线程的数目，工作线程的内存占用情况，线程一天处理的请求数量

# 接口设计

## 写操作接口（Client）

// ObMutator由多个ObCellMutator组成一个链表，可以操作一行或多行，甚至可以跨表操作，OceanBase保证ObMutator操作的原子性。

**class** ObMutator

{

**public**:

// 加入一个对cell的修改操作

// @param [in] mutator 待加入mutator

**int** **add\_cell\_mutator**(**const** ObCellMutator& mutator);

**public**:

// 序列化/反序列化接口

**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);

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

};

// 一次Mutator的执行结果

**struct** ObOperateResult

{

int32\_t err\_step; // 执行第几个ColumnMutator的时候出错

int32\_t err\_code; // 整个row mutation的执行结果

int32\_t affect; // row mutation影响行数，数据库语义使用

ObString err\_msg;

};

// 执行一次Mutator，保证mutator操作原子性

//

// @param [in] mutator 行操作链

// @param [out] operation 返回结果

// @return 成功或者其它错误包括Lease非法、schema非法、写日志失败、网络错误

ObResultCode **ups\_apply**(**const** ObMutator& mutator, ObOperateResult& operation);

// 批量执行Mutator，保证单个Mutator原子性但不保证整体原子性

// @param [in] batch\_num 批量个数

// @param [in] mutators 批量行操作链

// @param [out] operations 每个mutator执行后的返回结果

// @return 成功或者其它错误包括Lease非法，某些mutator执行失败

ObResultCode **ups\_batch\_apply**(**const** int64\_t batch\_num, **const** ObMutator\* mutators, ObOperateResult\* operations);

## 读/扫描操作接口 (Chunk Server)

**enum** BorderFlag

{

*INCLUSIVE\_START* = 1,

*INCLUSIVE\_END* = 2,

*MIN* = 4, // 表示start\_key为最小row\_key

*MAX* = 8, // 表示end\_key为最大row\_key

};

**struct** ObDataInterval

{

OBString start\_key;

OBString end\_key;

BorderFlag border\_flag;

};

// 读取类型，合并过程中CS需要读取UPS更新数据，是否包括frozen\_mem\_table依赖CS合并是否完成

**enum** ObReadType

{

*FROZEN\_MEMTABLE* = 1, // 读取frozen\_mem\_table

*ACTIVE\_MEMTABLE* = 2, // 读取active\_mem\_table

*ALL\_MEMTABLE* = 3, // 读取所有的mem\_table

};

// 数据扫描接口，可以存放多行数据，使用方法：

// ObCellMutator\* pcell\_mutator = NULL;

// // 迭代取出所有的cell\_mutator

// while (!next())

// {

// get\_cell\_mutator(&p\_cell\_mutator);

// }

**class** ObScanner

{

**public**:

// 迭代器指针移动到下一个cell mutator

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

// 读取当前的cell mutator, 使用ObScanner内存

**virtual** **int** **get\_cell\_mutator**(ObCellMutator\*\* mutator);

// 读取当前table\_name, 使用ObScanner内存

**virtual** **int** **get\_table\_name**(ObString\*\* table\_name);

// 读取当前row key, 使用ObScanner内存

**virtual** **int** **get\_row\_key**(ObString\*\* row\_key);

// 读取当前列名，使用ObScanner内存

**virtual** **int** **get\_column\_name**(ObString\*\* column\_name);

// 读取当前列号

**virtual** **int** **get\_column\_id**(uint64\_t& column\_id);

// 读取当前列值，使用ObScanner内存

**virtual** **int** **get\_column\_value**(ObObj\*\* column\_name);

**public**:

// 序列化/反序列化接口

**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);

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

};

**class** ObGetParam

{

**public**:

// 加入待读取cell

**void** **add\_cell**(**const** ObReadType read\_type, **const** ObString& table\_name, **const** OBString& row\_key, **const** OBString\* column\_name);

// 设置待读取某列的列号

**void** **set\_column\_id**(**const** ObString& column\_name, **const** uint64\_t column\_id);

**public**:

// 序列化/反序列化接口

**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);

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

};

//

// 读取一批数据

// @param [in] get\_param get操作相关参数

// @param [out] scanner 读取结果，通过它可以迭代取出所有的操作

// @param [out] operation 读取操作结果

// @return 成功或其它错误包括Lease非法、Schema错误

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

// 批量读取接口

// @param [in] batch\_num 批量个数

// @param [in] get\_params 读取操作参数数组

// @param [out] scanner 每个读取操作的读取结果

// @param [out] operations 每个读取操作的执行结构

// @return 成功或其它错误包括Lease非法和某些读取错误

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

**class** ObScanParam

{

**public**:

// 设置扫描相关参数

// @param [in] read\_type 扫描类型

// @param [in] table\_name 表名

// @param [in] range 扫描范围

**void** **set**(**const** ObReadType& read\_type, **const** ObString& table\_name, **const** ObDataInterval& range);

// 加入需要扫描列，扫描器一次扫描指定行范围的多个列

**void** **add\_column**(**const** OBString\* column\_name);

// 设置待扫描某列的列号

**void** **set\_column\_id**(**const** ObString& column\_name, **const** uint64\_t column\_id);

// 设置扫描大小，可选，一般为1MB ~ 4MB，默认值为2M

**void** **set\_scan\_size**(**const** int64\_t scan\_size);

**public**:

// 序列化/反序列化接口

**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);

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

};

// 扫描多行数据

// @param [in] scan\_param scan操作相关参数

// @param [out] scanner 多行扫描结果，通过它可以迭代出每行的所有操作；

// @param [out] end\_flag 扫描是否结束

// @return 成功或其他错误包括Lease非法，schema非法，不支持的操作（如果扫描包括active\_memtable且开始行和结束行的 row\_key split部分不同）

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

// 获取bloom filter信息

// @param [in] table\_name 表名

// @param [out] filter 指定table冻结的修改操作的bloom\_filter数据

ObResultCode **ups\_get\_bloom\_filter**(**const** ObString& table\_name, ObBloomFilter& filter);

## 元数据&Memtable (Root Server)

// 从Root Server网络获取table schema信息

// @param [out] schema 获取的schema信息，所有table的schema信息放在一个结构内

ObResultCode **ups\_fetch\_schema**(ObTableSchema& schema);

// 冻结Memtable，每天合并前由Root Server发送指令冻结

ObResultCode **ups\_freeze\_memtable**();

// 卸载Memtable，等到所有的ChunkServer合并完成以后，Root Server发送卸载Memtable指令

ObResultCode **ups\_drop\_memtable**();