**RootServer需求分析**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **文档版本** | **修订章节** | **修订原因** | **修订日期** | **修订人** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 功能需求

## 外部功能需求

外部功能需求主要描述root server 对客户端提供的服务.

### 取得 update server 信息

返回给客户端, 系统的 update server 的vip.

### 通过 get 接口查找 tablet 信息

将 root server 所管理的tablet 信息, 以一张表的形式提供给外界查询.

### 通过 scan 接口查找tablet 信息

将 root server 所管理的tablet 信息, 以一张表的形式提供给外界查询.

## 系统内部需求

系统内部需求是 root server 对chunk server 和 update server提供的服务, 以及为了系统的正常运行, root server必须提供的服务.

### 管理schema

schema 包括下面这些部分

1 表名: 可打印的字符串

2 表类型: 这个用来表示做merge的时候, merger到chunk server的什么介质上: 1 chunk server 磁盘 2 chunk server 内存. 当这个类型为2的时候, 其实相当于该表是一张内存表(其静态部分和动态部分都在内存中)

列的属性. 列可以只存在于update server或者既存在于update server又存在于chunk server, 分别用0 和 1表示(其实就是是否持久化)

列的类型: int; float; double; varchar(x); datetime; precise;datetime; create\_time; modify\_time.

rowkey的prefix, rowkeylength

join 关系

最大列id

schema 是一个纯文本的字符流 为了方便, 尽量向配置文件的方式靠拢 采用 section 和内容这种形式

table\_id < 1000 的系统留用, 请不要使用

[app\_name]

name=collect

max\_table\_id=1002 //现在已经使用的, 当前最大的tableid

[collect\_item\_id] //table\_id

table\_id=1001

table\_type=1

column\_info=0,2,item\_name,varchar,20 //列名是item\_name 变长字符20 第一个0表示只放到update server

column\_info=0,3,new\_price,int64

rowkey\_split=0 //表示rowkey的前0个字节是拆分点 实际上这个表的rowkey 应该是 itemt\_type(char(1))+item\_id(int64)

rowkey\_max\_length=9

max\_column\_id=3 //现在已经使用的, 本表内最大的column\_id

[item\_detail] //这个地方写的是table\_name

table\_id=1002

table\_type=2 //1 放在内存中的 2 放在chunk server.

column\_info=1,2,item\_name,varchar,20 //列id是2 列名是item\_name 变长字符20 第一1表示既放到update server也放到chunk server

column\_info=1,3,item\_price,int64

rowkey\_split=8 //表示rowkey的前8个字节是拆分点 实际上这个表的rowkey 应该是 uid(int64)+itemt\_type(char)+item\_id(int64)

join=rowkey[8,16]%m\_collect\_item\_id:item\_name$item\_name,item\_price$new\_price

rowkey\_max\_length=17

max\_column\_id=3

----------------------------------------------------------

列id 总是从2开始 1 留给rowkey 0无意义

做一点join表达形式细节上的说明. %表示外键关联，:表示其后是需要join的字段，其前是关联外键的描述

在join中出现的 需要join的列, 总是$前的是本表的 $后的是另外一张表的.不同字段之间通过,分隔。

join关系的连接词中 前者不一定是rowkey(可能是rowkey的一部分, 或者是一个字段) 后者必须是一个完整的rowkey, 但是为了便于比较,rowkey 无结构, 字段是有类型的 不方便比较. 我们一期先都做成是rowkey的

如果有多种join 可以像column\_info那样 写多条

rowkey\_is\_fixed\_length 这个用来配置rowkey是否是固定长度. 0，非固定长度, 其它值, 固定长度.

我们还需要判断两个schema是否兼容, 对于兼容问题有如下约定:

哪些是不兼容的

1 表的增加 删除 都是兼容的.

2 列的改名是允许的. 列的更改类型不允许 同 table id 同 col id 必须相同type 对于varchar类型 size也要一样(此处再讨论)

3 对于table type 从 内存chunk server改变到硬盘chunk server 或者相反是允许的.

4 rowkey split 可以改, row key max length也可以改变

5 join 关系的增删是允许的

6 app\_name 不能改

注: schema 目前有如下信息: 一张表是否使用 bloom filter 一张表的 block size 一张表的压缩算法名字

能够从配置文件读取 schema 判断 schema 的合法性, 相容性.

能够让提供schema的查询服务

### 管理chunk server

维护chunk server的状态, 发放租约. 记录chunk server的处理能力. 处理能力包括chunk server可提供的硬盘空间以及iops.

检测到chunk server的加入或者退出.

### 根据chunk server的汇报信息, 建立tablet表

接受chunk server的关于其hold的tablet信息的汇报.

将信息合并入tablet 信息表.

接受 chunk server关于负载的信息的, 建立负载记录

### 控制静态动态数据的合并过程

控制update server上的数据合并到chunk server的整个过程.

### 发起强制切换的功能(这个主要用在update\_server内存可能不够的时候)

### 对于各个chunk server的汇报, 汇报结束后, root server 要能够验证是否正确.

如果发现某个表的tablet有交叉, 或者空洞, 都是不可用的数据, 要报警, 并且不能采用这种新数据.

### 支持动态的加入新的chunk server

一个chunk server加入后, root server要根据负载均衡的策略逐渐切流量过去.

对于新加入的, 暂时不提供服务的机器, 复制数据的时候可以让该机器的入口流量很大, 但是要控制迁出源的出口流量. 对于同时有大量chunk server加入的情况, 可以处理成一台一台逐步加入(负载算法会简单).

### 提供负载均衡能力, 当发现系统负载不均衡的时候, 要调整系统的负载

系统的负载分为存储负载和服务负载, root server必须同时考虑这两种因素.

另外负载均衡的时候, 要让一台chunk server上的信息尽可能均匀的分布到其它的chunk server上, 这样有机器宕机的时候恢复起来速度最快. 暂时不考虑机架信息.

调整负载的时候还要考虑到, 可以合并的tablet 尽量调整到一台chunk server上.

### 提供一主一备的热切换

### chunk server 心跳的反向

当发现某chunk server的租约快要到期的时候, 主动给chunk server发消息更新租约, 以防止chunk server数量巨大的时候, 对root server造成瞬间压力.

# 关于实现的想法

## 基本数据结构

### tablet 信息.

此结构是一个tablet的大数组, 共128k个.

存储, tablet\_info

上报的时候就是逐个添加.

### server信息

observer (ip) + port cs + port ms + last heart beat time + capacity能力

不排序, 查找用遍历的方式 最大值先设置成1000个

### root table表

root table表记录了tablet 和 server的对应关系. 同时记录了, 每个tablet 在不同server上的tablet版本号.

Root table表支持tablet的分裂, 合并, 新建, 查询.

Root table 能够检测到root 表内的数据是否完整, 可用.

## 外部接口实现

### Scan接口的实现

这个最终的对外表现形式跟chunk server一致, 内部实现上, 也可以参考chunk server, 这里不详细讨论了.

## 内部功能实现

### 汇报与切换

汇报流程是指root server要求chunk server上报自己所hold的tablet信息的过程. 该流程由root server发起, chunk server结束. 该流程发生在三种情况下

1 chunk server加入集群的过程.(可能有数据, 也可能为空)

2 系统初次建立的过程

当chunk server加入到集群的时候, 它上报自己hold的所有tablet信息以及自己的磁盘使用信息. Root server接收到这些信息后, 修改当前的root table表. 修改的原则如下: 假设chunk server上报tablet T1。 Root server会查找自己的root table 表, 找到T1所在的三个chunk server信息. 如果当前上报的chunk server是三者之一, 则更新自己的信息. 如果不是, 则看T1所在的三个chunk server是否有下线的, 或者tablet版本过旧的, 如果还没能找到, 则通过磁盘使用情况来决定, 哪个被替换掉.

一个加入到集群而发生上报的chunk server, 它所上报上来的tablet应该是root table表中已经存在了的. 所以, 对于上报的在root table表中查询不到的tablet, 可以忽略掉.

chunk server加入的流程在chunk server管理章节叙述, 合并切换流程在本节后续内容叙述.

系统初次建立的过程本来可以看成是一台一台的chunk server逐步加入的过程. 不同点在于我们认为加入机器是很少发生的过程, 所以加入机器的过程的内存使用很不经济, 这对于系统的初次建立来说不合适. 所以目前的处理, 把系统的初次建立归入到合并切换流程, 只是chunk server不用合并数据 .

切换流程是指每天例行的(或强制的)合并动态数据到cs成为静态数据的过程. 切换流程如下

1 更新本地schema:

先从本地配置文件更新schema信息.目前root表是静态的, 每天一次更新(或者手工强制更新).

2 update server的切换

root server 调用update server的 up\_freeze\_mem\_table(int64\_t& version), 通知update server开始合并过程, 其中version为传出参数, 由update server生成. Update server收到消息后, 先调用root server的 rt\_fetch\_schema(const int64\_t time\_stamp, ObSchemaManager& schema), 取得新的schema后, 冻结本地内存表, 用新的schema建立新的内存表. 然后ups 调用 root server的rt\_mem\_table\_frozen (const int64\_t version). 完成update server的准备过程.

3 发起chunk server合并和汇报过程

Root server调用所有活着的chunk server的cs\_start\_merge(const int64\_t time\_stamp, int init\_fag) 命令, 其中init\_flag表示要有合并过程还是系统的第一次汇报过程. 所有收到命令的cs节点调用rt\_get\_schema(const int64\_t time\_stamp) 从root server取得新的schema. 然后开始在本地完成合并过程(可能涉及到tablet的分裂), 通过调用root server的rt\_report\_tablets(const Server & server, const TabletList & tablets, const int64\_t version), 将合并的结果汇报给root server. 一个chunk server完成了所有的汇报后 将Chunk server生成的磁盘信息, 通过调用root server的函数 rt\_report\_capacity\_info 上报, 最后调用 root server的rt\_merge\_over(const Server & server, const int64\_t time\_stamp). 完成该chunk server的合并切换过程.

4 建立新的root table并发起切换过程

当root server发现所有的活着的chunk server都调用过rt\_schema\_changed后, root server 开始检查是否可以卸载update server的冻结表. 如果每个tablet都至少有一个chunk server合并成功并上报, 那么update server的冻结表就可以卸载. Root server会通知 chunk server 和update server 卸载老表, 从而完成合并过程. 但是如果有某tablet的三个备份都没合并成功, 则本次merge 失败, root server写警告, 不会发卸载命令.

### 新加table的建立

对于schema里新增的表, 因为没有chunk server 上有这种表的数据. Root server会挑选合适的chunk server, 去上面建立一个从 min 到 max 的tablet.

### Chunk server管理

这部分对应着需求里, 跟chunk server状态, 能力相关的所有功能, 形成的数据, 要能够为负载均衡算法提供参考依据.

建立数据结构 server均衡信息, 存储了一台server的磁盘能力, 已经使用的磁盘量, 负载能力,.

Root server 还要管理当前所有server的租约信息. 租约管理逻辑如下

建立一个结构, 存储server的最后一次续约的时间戳.

每次收到server的租约请求, 则修改时间戳并且返回租约时长..

建立扫描线程, 定时扫描所有的server, 发现某server的租约期过了一半以上, 则给该server发消息, 通知它续租.(发消息要用异步接口). 如果发现某server租约过期, 则触发删除流程(删除流程必须无阻塞, 因为处理过程是在扫描线程内完成)

对于注册进来的chunk server, 要求其上报自己的tablet, 并且将结果合并入root table表

Chunk server的删除流程, 要修改root table表, 把指向该chunk server的表项中的指针置空. 这主要是为了防止过一段时间后, 该chunk server加回来, 但是hold的tablet却变化了. 如果不置空, 这个时候可能会出问题.

### 负载均衡

一 负载均衡所用到的信息

    1 每个server的磁盘能力， 当前已经使用了的磁盘量

    server->size\_capacity/used size

     2 每个tablet的size（用作合并tablet的参考）

二 负载均衡要考虑的维度， 以及各个维度的权值

目前, 我们只考虑磁盘的容量和tablet分布的均匀程度问题.

磁盘容量和负载都设置一个高水位百分数。一个迁移触发百分数.

设置均匀程度调节值, 均匀程度差别在这个范围内的两台chunk server, 认为是一样均匀的.

三 触发负载均衡的条件

    有两种迁移过程，分别是 优化迁移 和 紧急迁移。

    简单来说， 优化迁移不会报警， 紧急迁移一定会报警

    先来看优化迁移的触发条件

    某server的磁盘使用超过迁移触发百分数， 且迁移后能使情况好转

    所谓能够使情况好转， 是指受迁移影响的servers， 迁移后， 都不会再次满足迁移条件

    紧急迁移的触发条件

    1 某tablet只剩下一份数据

    2 某tablet只剩下两份数据， 并且持续超过N时间

    3 某server的磁盘使用达到高水位

    任何紧急迁移的触发条件成立， 就必须报警

    能使情况好转是指 受迁移影响的server 迁移后不会触发紧急迁移条件。

    目前， 先将因为合并tablet而应该发起的迁移独立出来。 这种迁移比较简单， 可以在cs ups合并过程中处理。

四 迁移的策略

    首先定义一个可接受的迁移结果的条件

       所谓可接受的迁移结果， 应该满足下面的条件：

       1 任何tablet的两份数据不在同一台server上

       2 所有的server的磁盘使用都低于高水位（必须满足）

       3 所有的server的磁盘使用都低于迁移触发水位（优化迁移必须满足）

迁移过程当中要注意均衡性。

下面分两种情况具体分析:

情况1 因为某个tablet的数据拷贝不足要求的份数而引发的复制操作

复制操作,是要求增加一份数据, 首先我们确切的知道要增加哪个tablet的备份, 那么我们的任务是找到合适的chunk server增加这个备份. 假设tablet Ta 现在存在于chunk server C1和C2上, 如果我们选定增加备份的chunk server是Cx, 则迁移后, C1于Cx共有的tablet会增加1个, C2于Cx共有的tablet也会增加一个. 本着均衡的原则, 我们找于C1和C2共有的tablet最少的chunk server做为最优先的考虑对象. 因为有调节值的存在, 这一类的chunk server可能不是一个而是一批. 在这一批 chunk server中再选取磁盘使用量最低的.

情况2 因为某chunk server磁盘使用的太多了, 达到触发水位而发生的迁移操作

这种迁移操作, 是把数据从一个chunk server迁移到另一个chunk server上. 我们已经知道迁出的chunk server. 需要确定迁入的chunk server和tablet. 因为tablet的数量远大于chunk server的数量, 所以我们先确定迁入的chunk server. 确定的原则就是磁盘使用最少chunk server做为迁入. 现在假设一个tablet 原来存在于C1 C2 C3 三台chunk server上. 如果迁移后的结果是 C1 C2 Cx, 那么我们可以发现, C1于C3, C2于C3共有的tablet都会减少1个; C1于Cx, C2于Cx共有的tablet都增加一个. 这种迁移对于C3和Cx之间共有的tablet没有影响. 上面分析中, C3就是我们发现磁盘用量大要迁出的, Cx是我们找到的磁盘用量少, 要迁入的. 我们现在要挑选合适tablet. 目前采用一个简单的原则, 寻找C1, C2 于Cx共有的tablet最少的那种组合.

### 迁移和tablet的合并

主要是一些与chunk server的交互过程, 需要跟chunk server讨论确定. 逻辑上就是控制迁移过程, 把可以合并的tablet迁移到一台server上(三个备份是三台), 然后控制chunk server对其合并.

### 主备root server间必须同步的数据

#### schema的文本形式

  每次重读schema的时候 从主备份到备

#### chunk server上下线消息 merger server的注册消息.

产生点 1 注册, 2 心跳线程

#### chunk server的负载变化

1 上报负载的时候 ;2 迁移完成的时候chuk server发送的迁移完成消息

#### root table的变化

产生点 1 开始新建,过程中 chunk server的上报 2 平时chunk server的上报 3 迁移完成的消息 4 chunk server下线 5 root server在chunk server上建立新的tablet

#### 其它要同步的事件

开始汇报消息, 开始切换消息

#### 检查点

可以在每天新的root table生成之后, 将chunk server数据和root table数据dump(包括 schema的 time stamp) 产生检查点.

# Root server的接口

## 给client的接口

说明:

Root server 中存储了一张关于数据在各个chunk 上分布的信息的表, 表结构如下

rowkey

1\_port(int32) 1\_ms\_port(int32) 1\_ipv6\_1(int32) 1\_ipv6\_2(int32) 1\_ipv6\_3(int32) 1\_ipv6\_4(int32) 1\_ipv4(int32) 1\_tablet\_version(int64)

2\_port(int32) 2\_ms\_port(int32) 2\_ipv6\_1(int32) 2\_ipv6\_2(int32) 2\_ipv6\_3(int32) 2\_ipv6\_4(int32) 2\_ipv4(int32) 2\_ tablet\_version (int64)

3\_port(int32) 3\_ms\_port(int32)3\_ipv6\_1(int32) 3\_ipv6\_2(int32) 3\_ipv6\_3(int32) 3\_ipv6\_4(int32) 3\_ipv4(int32) 3\_ tablet\_version (int64)

occupy\_size(int32) record\_count(int32)

其中 rowkey 是表id加上tablet的range的结束rowkey 组成. 每行记录有三组信息, 表示数据的三个备份. 每组信息有一个port 和 一组ipv6信息(4个int32) 以及一个ipv4信息(1个int32) , occupy\_size存储了当前tablet大小, record\_count是记录的条数.

X\_port 这个端口是chunk server的端口, merge server查询的时候会关注, x\_ms\_port 是merg server的端口, 客户端关注.

对root table 的查询, 借用对chunk server的查询接口, 但是多一个参数 table\_name;

这里需要做说明的是, ObCsGetParam参数中的table\_name会被转换成id 和 rowkey 一起形成root 表里的rowkey.

这里传入的参数 table\_name 目前只能取值为 “ob\_root\_table”. 将来可能会取值为”ob\_index\_table”.

ObResultCode **rt\_get**(**const** ObCsGetParam& get\_param, const ObString& table\_name, ObScanner& scanner, ObOperateResult& operation);

ObResultCode **rt\_scan**(**const** ObCsScanParam& scan\_param, const ObString& table\_name, ObScanner& scanner, **bool**& end\_flag);

ObResultCode rt\_get\_update\_server\_info(Server& update\_server\_info);

取 update server 的地址信息.

## 注册接口

每一个chunk server或者merger server必须调用此接口向 root server 注册, 其中 status是输出值, 注册的server根据这个值决定下面的动作

当 status 为 0 表示系统正处在第一次启动中, 注册的server不需要汇报自己的tablet, 应该等待. 当 status为1 表示系统已经完成了初次启动过程, 注册的server应该汇报tablet信息

ObResultCode **rt\_register** (**const** ObServer & server, const bool is\_merge\_server, int32\_t& status);

## 汇报接口

ObResultCode up\_freeze\_mem\_table(int64\_t& version)

这个用来对update server发起merge 过程

ObResultCode cs\_start\_merge(const int64\_t version, int init\_fag);

这个是通知chunk server发起合并过程

关于init flag的含义, 如果init falg为0， 则chunk server需要跟update server做merge, 如果为1 则不需要, 一般, 系统第一次启动的时候, 传1(这个时候update server没有数据)

/\*

\* 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 rt\_report\_tablets(const ObServer& server, const ObTabletReportInfoList& tablets, uint64\_t time\_stamp);

/\*

\* update server向RootServer通知, 自己具完成了内存表相关工作, 可以进行后续合并工作了

\*

\*/

ObResultCode rt\_mem\_table\_frozen (const int64\_t version).

/\*

\* chunk server 向RootServer通知, 自己的merge过程完成了

\* @param [in] server 汇报的ChunkServer

\*/

ObResultCode rt\_rt\_merge\_over (const Observer & server, const int64\_t time\_stamp);

cs\_ unload\_old\_table (const int64\_t time\_stamp, const int64\_t old\_table\_remain)

up\_ unload\_old\_table (const int64\_t time\_stamp)

上面两个函数, 是由root server发起调用, 用于通知各个server在给定时间内卸载老于time\_stamp的表. 其中发给update\_server的命令 给定时间等于0, 这两个函数底层都用下面这个实现

/\*

\* 通知server在old\_table\_remain的时间范围内, 逐步卸载非 time\_stamp 标识的表

\* 这个是由root server主动发起的,

\* @param [in] time\_stamp 新schema的时间戳

\*/

ObResultCode unload\_old\_table (const Observer & server , uint64\_t time\_stamp, const int64\_t old\_table\_remain);

ObResultCode rt\_report\_capacity\_info(const Observer& server, const int64\_t capacity, const int64\_t used);

## 心跳接口

/\*

\* root server发心跳和租约信息给chunk server

\* @param [out] lease\_duration 租约时长

\*/

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

## 迁移接口

ObResultCode cs\_migrate(const ObRange& range, const ObServer& dest\_server, const bool keep\_src)

Root server调用cs, 表示将range从当前的chunk server迁移到 dest\_server. Keep\_src表示当前的chunk server上的备份是否保留

ObResultCode rt\_migrate\_over(const ObRange& range, const ObServer& src\_server, const ObServer& dest\_server, bool keep\_src)

Cs完成迁移后调用root server的该方法, 汇报迁移的结果.

## 生成新tablet的接口

对于新加入系统的表, root server会选择合适的chunk server在其上建立新的表.

ObResultCode rt cs\_create\_tablet(const ObRange& range range)

## Schema服务

/\*

\* 从Root Server网络获取table schema信息

\* 这个接口只有ChunkServer和UpdateServer在每天合并之前进行调用，对于RootServer

\* 来说，必须在调用之前准备好新的schema，这个时间窗口比较重要。

\* @param [in] time\_stamp schema的标识

\* @param [out] schema 获取的schema信息

\*/

ObResultCode rt\_**fetch\_schema**(const int64\_t time\_stamp, ObSchemaManager& schema);

## SchemaManager提供的接口

### Ob中Schema的约定

参见 1.2.1 系统内部需求, 管理 schema 部分

### SchemaManager的层次

SchemaManage管理一个业务下所有表的schema. 所以他是一个业务下表的schema的集合.

每个表的schema都是一个ObSchema类, 完整的描述了一个表的所有信息. 它由一个表的基本信息(比如表名) 以及多条列的信息和jonin关系信息组成.

个列的信息都是一个ObColumnSchema类, 存储着这个列的属性.

每条join关系都是一个ObJoinInfo类, 存储join关系的相关信息.

下面我们自顶向下分别介绍这些类的对外接口

### ObSchemaManager类的接口

**class** ObSchemaManager

{

**public**:

**ObSchemaManager**();

**explicit** **ObSchemaManager**(**const** int64\_t time\_stamp);

**virtual** **~ObSchemaManager**();

/\*

\* 获取该schema的时间戳

\*/

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

/\*

\* 从配置文件中读取schema信息

\* @param [in] file\_name 配置文件名

\* @param [in] 读取配置文件用的config类

\* @return true 成功 false 失败

\*/

**bool** **parse\_from\_file**(**const** **char**\* file\_name, tbsys::CConfig& config);

/\*

\* 根据表名得到表的id

\* @return 0 没有找到 其他 table\_id

\*/

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

/\*

\* 根据table\_id得到table\_name字串

\* @return NULL 没有找到

\*/

**const** **char**\* **get\_table\_name**(**const** uint64\_t table\_id) **const**;

/\*

\* 根据table\_id得到描述该table 的schema

\* @return NULL 没有找到

\*/

**const** ObSchema\* **get\_table\_schema**(**const** uint64\_t table\_id) **const**;

/\*

\* 根据table\_id和column\_name得到clumn的id

\* @return 0 没有找到

\*/

uint64\_t **get\_column\_id**(**const** uint64\_t table\_id, **const** **char**\* col\_name) **const**;

/\*

\* 查找table\_id这张表 column\_id这个列的列名

\* @return NULL 没有找到

\*/

**const** **char**\* **get\_column\_name**(**const** uint64\_t table\_id, **const** uint64\_t col\_id) **const**;

/\*

\* 取得table\_id这张表下column\_id这个列的描述信息

\* @return NULL 没有找到

\*/

**const** ObColumnSchema\* **get\_column\_schema**(**const** uint64\_t table\_id, **const** uint64\_t col\_id) **const**;

/\*

\* 取得当前应用的应用名

\*/

**const** **char**\* **get\_app\_name**() **const**;

// if this func return false; then we can not use these schemas

// to replace the old one

/\*

\* 判断新的schema是否能与老的schema相兼容

\* 兼容的标准见相关文档

\*/

**bool** **is\_compatible**(**const** ObSchemaManager& schema\_manager) **const**;

NEED\_SERIALIZE\_AND\_DESERIALIZE;

//this is for test

**void** **print\_info**() **const**;

}

### ObSchema类的接口

**class** ObSchema

{

**public**:

**enum** TableType

{

*INVALID* = 0,

*SSTABLE\_IN\_DISK*,

*SSTABLE\_IN\_RAM*,

};

**ObSchema**();

/\*

\* 通过表的一些基本信息初始化类

\* 实际上,这里用于初始化该类的信息在类完成初始化后都不可以修改

\*/

**bool** **init**(**const** uint64\_t table\_id, **const** **char**\* name,

**const** TableType table\_type, **const** int32\_t rowkey\_split, **const** int32\_t rowkye\_max\_length);

**virtual** **~ObSchema**();

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

TableType **get\_table\_type**() **const**;

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

int32\_t **get\_split\_pos**() **const**;

int32\_t **get\_rowkey\_length**() **const**;

/\*

\* 用于判断一个操作是否可以应用到当前表上, 尚未实现

\*/

**bool** **is\_available**(/\*const ObOperator& ob\_ot\*/) **const**;

/\*

\* 通过column\_id查找对应的column的信息

\* @return NULL 没有找到

\*/

**const** ObColumnSchema\* **find\_column\_info**(**const** uint64\_t column\_id) **const**;

/\*

\* 通过column\_name查找对应的column的信息

\* @return NULL 没有找到

\*/

**const** ObColumnSchema\* **find\_column\_info**(**const** **char**\* column\_name) **const**;

/\*

\* 对当前表增加一条join关系

\* 这个函数主要用于表的schema的构建过程

\*/

**void** **add\_join\_info**(**const** ObJoinInfo& join\_info);

/\*

\* 增加一条column的信息.

\* @return true 成功, false 失败, 失败是因为列超过了预定义的最大列数目

\*/

**bool** **add\_column**(**const** uint64\_t column\_id, **const** ObColumnSchema& column);

/\*

\* 查找作用于 lcolumn\_id 这个列上的join关系

\* 说明, 我们每一个列上应该最多只有一个join关系, 因为我们的join关系是用后者的值覆盖

\* 前者, 如果有多条join关系, 其实只有最后一条是有意义的.

\*/

**const** ObJoinInfo\* **find\_join\_info**(**const** uint64\_t lcolumn\_id) **const**;

/\*

\* join关系实际上存储与内部的一个数组中, 通过改接口, 可以按照数组顺序遍历join关系

\* @return NULL 以index为下标的数组中无有效的join关系

\*/

**const** ObJoinInfo\* **get\_join\_info**(**const** int64\_t index) **const**;

NEED\_SERIALIZE\_AND\_DESERIALIZE;

//this is for test

**void** **print\_info**() **const**;

}

### ObColumnSchema类的接口

**class** ObColumnSchema

{

**public**:

**ObColumnSchema**();

/\*

\* 初始化一个ColumnSchema

\* 以参数传入的属性, 在完成初始化后均不可修改

\* @param [in] maintained 表示这个类是否需要持久化到chunk server上

\*/

**bool** **init**(**const** **bool** maintained, **const** uint64\_t id, **const** **char**\* name, **const** ColumnType& type, **const** int64\_t size);

**virtual** **~ObColumnSchema**();

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

**const** **char**\* **get\_name**() **const**;

ColumnType **get\_type**() **const**;

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

**bool** **is\_maintained**() **const**;

//this is for test

**void** **print\_info**() **const**;

**static** ColumnType **convert\_str\_to\_column\_type**(**const** **char**\* str);

**bool** **operator <** (**const** ObColumnSchema& rv) **const**;

**bool** **operator <** (**const** **char**\* name) **const**;

}

### ObJoinInfo类的接口

**class** ObJoinInfo

{

**public**:

**ObJoinInfo**();

**bool** **init**(**const** uint64\_t left\_column, **const** int32\_t start\_pos, **const** int32\_t end\_pos, **const** uint64\_t table\_id\_joined);

**virtual** **~ObJoinInfo**();

/\*

\* 得到做join时, 需要使用rowkey的那一部分

\* 如果返回的start\_pos 和 end\_pos 都是0, 表示需要使用整个rowkey

\*/

//if you got start\_pos = end\_pos = 0, that means you should use the whole rowkey

**void** **get\_rowkey\_join\_range**(int32\_t& out\_start\_pos, int32\_t& out\_end\_pos) **const**;

/\*

\* 这个函数在构建join关系的时候调用

\* 我们的join关系实际上是用另外一个表的某(几个)列覆盖当前表的某(几个)列,

\* 我们称当前表被覆盖的列是左值列, 用来覆盖别人的列是右值列

\* 当前表为左表, 用于覆盖的表为右表

\*/

**void** **add\_correlated\_column**(**const** uint64\_t left\_column\_id, **const** uint64\_t right\_column\_id);

/\*

\* 得到本条join关系中的右表id

\*/

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

/\*

\* 通过右值列来找左值列

\* @return 0 没有找到

\*/

uint64\_t **find\_left\_column\_id**(**const** uint64\_t rid) **const**;

/\*

\* 通过左值列来找又值列

\* @return 0 没有找到

\*/

uint64\_t **find\_right\_column\_id**(**const** uint64\_t lid) **const**;

NEED\_SERIALIZE\_AND\_DESERIALIZE;

//this is for test

**void** **print\_info**() **const**;

}