

---

# Table of Contents

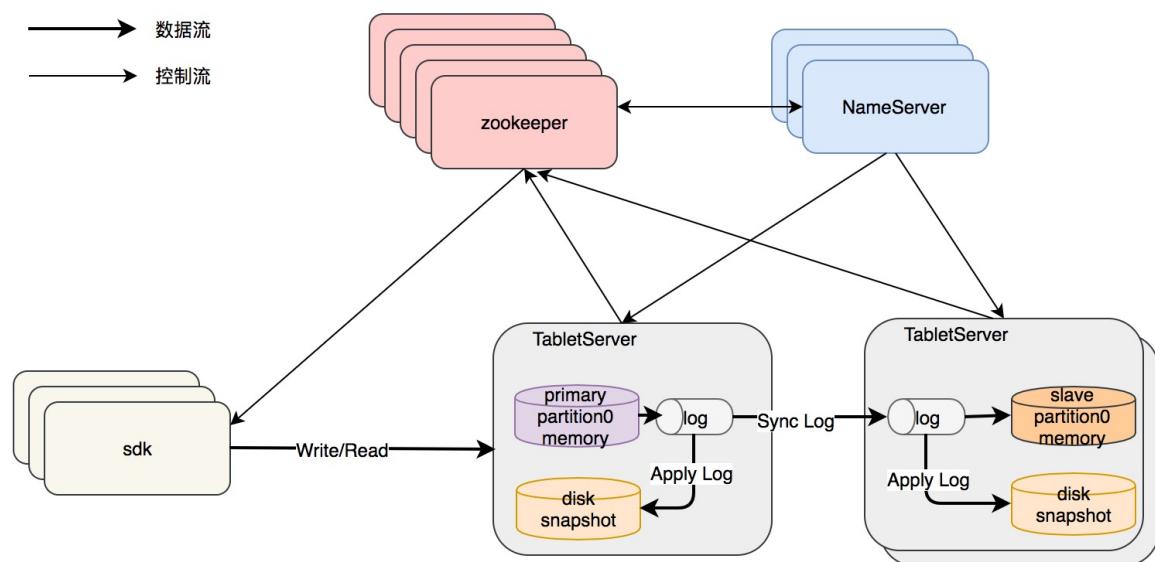
RTIDB介绍	1.1
部署安装	1.2
配置机器环境	1.2.1
一键部署RTIDB集群	1.2.2
部署zookeeper	1.2.3
部署RTIDB	1.2.4
配置文件详解	1.2.5
目录介绍	1.2.6
RTIDB使用	1.3
命令行	1.3.1
java sdk	1.3.2
返回码	1.3.3
导入工具	1.3.4
RTIDB运维	1.4
RTIDB监控	1.4.1
常见问题	1.5

# RTIDB介绍

RTIDB(Real Time Intelligence Database)是一个面向机器学习场景下的高性能内存数据库. 应用于在线实时计算和离线批量预估等多种场景. 具有如下特性:

- 高性能
- 分布式高可用
- 可扩展
- 支持kv和schema存储
- 支持多维度查询
- 支持java、c++等多种client
- 方便易用

内部架构图如下:



各模块功能:

zookeeper 服务发现和保存元数据信息

nameserver 管理tablet, 做高可用、failover等

tablet 存储数据, 主从同步数据

## 部署安装

[配置机器环境](#)

[部署zookeeper](#)

如果有现成的zookeeper或者是部署的单机版RTIDB就跳过此步

[部署RTIDB](#)

[配置文件详解](#)

[目录介绍](#)

- 资源评估
- 机器环境准备

## 资源评估

### 内存

rtidb目前只支持两中表: 按条数过期的表即latest表和按时间过期的表即absolute表

#### latest表(只保留最近1条)

##### 1 无schema

内存大小 = 数据条数 \* (主键大小 + value大小 + 188)

value大小为实际value大小

##### 2 多维度

如果不同维度的数据落在同一节点和分片下, 会共享value.

内存大小 = 维度1的主键数 \* (维度1主键的大小 + 156) + 维度2的主键数 \* (维度2主键的大小 + 156) + ... + 维度n的主键数 \* (维度n主键的大小 + 156) + n \* 数据条数 \* 56 + k \* 数据条数 \* value大小  
value大小的计算方式参考下面schema编码大小计算方式

说明:

1 主键数是指有多少个不同的主键, 如主键为卡号就指有多少个不同的卡号

2 k大于等于1小于等于n

#### absolute表

##### 1 无schema

内存大小 = 主键数 \* (主键大小 + 156) + 数据条数 \* (value大小 + 60)

value大小为实际value大小

##### 2 多维度

如果不同维度的数据落在同一节点和分片下, 会共享value

内存大小 = 维度1的主键数 \* (维度1主键的大小 + 156) + 维度2的主键数 \* (维度2主键的大小 + 156) + ... + 维度n的主键数 \* (维度n主键的大小 + 156) + n \* 数据条数 \* 60 + k \* 数据条数 \* value大小  
value大小的计算方式参考下面schema编码大小计算方式

说明:

1 主键数是指有多少个不同的主键, 如主键为卡号就指有多少个不同的卡号

2 k大于等于1小于等于n

#### schema编码大小计算方式

如果schema字段数小于128, 数据编码大小为各个字段的累加和 + 1

如果schema字段数大于等于128, 数据编码大小为各个字段的累加和 + 2

各类型字段大小如下:

- string : 2 + 字符串长度
- int16 : 4
- uint16 : 4
- int32 : 6
- int64 : 10
- uint32 : 6
- uint64 : 10
- float : 6
- double : 10
- bool : 3
- date : 10
- timestamp : 10

示例：单维度有schema的计算方式

一张交易表有5000万条数据，卡号为主键共1000万个卡号。 schema如下，其中卡号长度是10，mcc长度是5

```
column_desc {
    name : "card"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "mcc"
    type : "string"
    add_ts_idx : false
}
column_desc {
    name : "amt"
    type : "float"
    add_ts_idx : false
}
```

数据的内存占用 =  $10000000 * (10 + 156) + 50000000 * (12 + 7 + 6 + 1 + 60) = 596000000$   
0Byte

## 机器环境准备

- 关闭swap
- 关闭THP
- 时间和时区

### 关闭操作系统swap

查看当前系统swap是否关闭

\$ free	total	used	free	shared	buff/cache	available
---------	-------	------	------	--------	------------	-----------

Mem:	264011292	67445840	2230676	3269180	194334776	191204160
Swap:	0	0	0	0	0	0

如果swap一项全部为0表示已经关闭，否则运行下面命令关闭swap

```
$ swapoff -a
```

## 关闭THP(Transparent Huge Pages)

查看THP是否关闭

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled  
always [madvise] never  
$ cat /sys/kernel/mm/transparent_hugepage/defrag  
[always] madvise never
```

如果上面两个配置中"never"没有被方括号圈住就需要设置一下

```
$ echo 'never' > /sys/kernel/mm/transparent_hugepage/enabled  
$ echo 'never' > /sys/kernel/mm/transparent_hugepage/defrag
```

查看是否设置成功，如果"never"被方括号圈住表明已经设置成功，如下所示：

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled  
always madvise [never]  
$ cat /sys/kernel/mm/transparent_hugepage/defrag  
always madvise [never]
```

## 时间和时区设置

rtidb数据过期删除机制依赖于系统时钟，如果系统时钟不正确会导致过期数据没有删掉或者删掉了没有过期的数据

```
$ date  
Wed Aug 22 16:33:50 CST 2018
```

如果时区或者时间不对需要设置下

# 一键部署RTIDB集群

## 准备包

- 下载jdk包 wget <http://pkg.4paradigm.com/rtidb/dev/jdk-8u121-linux-x64.tar.gz>
- 下载zookeeper包 wget <http://pkg.4paradigm.com/rtidb/dev/zookeeper-3.4.10.tar.gz>
- 下载rtidb包 wget <http://pkg.4paradigm.com/rtidb/rtidb-cluster-1.4.1.tar.gz>
- 解压rtidb包

## 修改配置文件tools/conf.json

```
{
  "zookeeper": {
    "path": "/home/rtidb/env/test", // zk的部署路径
    "package": "./package/zookeeper-3.4.10.tar.gz", // 放zk包地址的路径
    "java_package": "./package/jdk-8u121-linux-x64.tar.gz", // java jdk路径
    "inner_port1": 2881, // zk内部通信端口
    "inner_port2": 3881, // zk选主端口
    "need_deploy": true, // 是否要部署zk, 如果已经部署有zk集群此处填false
    "address_arr": [
      {
        "address": "172.27.129.198:12201", // 配置zk的部署节点及端口
        "path": "/home/rtidb/env/zk", // 单独配置节点的zk部署路径
        "inner_port1": 5881, // 单独配置zk内部通信端口
        "inner_port2": 6881 // 单独配置zk选主端口
      },
      {
        "address": "172.27.128.33:12202" // 配置zk的部署节点及端口
      }
    ]
  },
  "zk_root_path": "/rtidb_cluster",
  "nameserver": {
    "path": "/home/rtidb/env/test/nameserver", // nameserver的部署路径
    "package": "./package/rtidb-cluster-1.4.0.tar.gz", // rtidb包路径
    "address_arr": [
      {
        "address": "172.27.129.198:6927" // nameserver的ip和port
      },
      {
        "address": "172.27.129.33:6927", // nameserver的ip和port
        "path": "/home/rtidb/env/test/ns" // nameserver的部署路径, 如果单独配了path就用此path
      }
    ]
  },
  "tablet": {
    ...
  }
}
```

```
"path":"/home/rtidb/env/test/tablet", // tablet的部署路径
"package": "./package/rtidb-cluster-1.4.0.tar.gz", // rtidb包路径
"address_arr": [
    {
        "address": "172.27.129.198:9027", // tablet的ip和port
        "path": "/home/rtidb/env/test/tb" // tablet的部署路径, 如果单独配了path
就用此path
    },
    {
        "address": "172.27.128.33:9027"
    }
]
}
```

注: 配置文件是一个json, 修改后必须还是一个合法的json

## 建立互信机制

建立互信的方法可以参考<https://blog.csdn.net/chenghuikai/article/details/52807074>

## 设定系统环境参数(关闭swap和THP)

- 切换到root账户
- 执行python tools/deploy.py tools/conf.json init\_env

注: 此处可以自行修改系统参数, 如果已经修改好就跳过此步

## 开始部署

- 切换到工作账户
- 执行python tools/deploy.py tools/conf.json

## 部署zookeeper

部署集群版rtidb需要部署zookeeper集群。如果有已经部署好的zookeeper集群可以复用现有的zookeeper集群。

部署三个节点或者三个以上节点的zookeeper集群才能保证高可用，建议在生产环境中至少部署三个节点。

- 部署单机版
- 部署集群版

### 部署单机版zk

#### 下载zookeeper安装包

```
$ wget http://pkg.4paradigm.com:81/rtidb/dev/zookeeper-3.4.10.tar.gz
$ tar -zxvf zookeeper-3.4.10.tar.gz
$ cd zookeeper-3.4.10
$ cp conf/zoo_sample.cfg conf/zoo.cfg
```

#### 修改配置文件

打开文件conf/zoo.cfg 修改dataDir和clientPort

```
dataDir=./data
clientPort=6181
```

#### 启动zookeeper

```
$ sh bin/zkServer.sh start
```

### 部署集群版zk

#### 下载zookeeper安装包

```
$ wget http://pkg.4paradigm.com:81/rtidb/dev/zookeeper-3.4.10.tar.gz
$ tar -zxvf zookeeper-3.4.10.tar.gz
$ cd zookeeper-3.4.10
$ cp conf/zoo_sample.cfg conf/zoo.cfg
```

#### 修改配置文件 conf/zoo.cfg

- 修改dataDir和clientPort。dataDir是zk保存数据的目录，clientPort是zk对外提供的端口号

- 添加server配置 格式为server.id=ip:port1:port2, 假如部署三个节点就在最后添加三行

```
dataDir=./data
clientPort=6181
server.1=172.27.128.31:2881:3881
server.2=172.27.128.32:2882:3882
server.3=172.27.128.33:2883:3883
```

**注: port1和port2不能和对应ip机器已占用端口冲突**

参考的配置文件如下

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=./data
# the port at which the clients will connect
clientPort=6181
# the maximum number of client connections.
# increase this if you need to handle more clients
maxClientCnxns=60
#
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in dataDir
#autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature
#autopurge.purgeInterval=1
server.1=172.27.128.31:2881:3881
server.2=172.27.128.32:2882:3882
server.3=172.27.128.33:2883:3883
```

## 添加myid文件

分别在各个server的数据目录下创建myid文件并设置id, 如果dataDir配置的是"dataDir=./data", 执行如下命令:

在server1机器上执行下面命令

```
$ echo 1 > ./data/myid
```

在server2机器上执行

```
$ echo 2 > ./data/myid
```

在server3机器上执行

```
$ echo 3 > ./data/myid
```

## 启动zk

在三台机器上分别执行启动命令

```
$ sh bin/zkServer.sh start
```

## 测试zk集群运行状态

用zkCli连上集群执行 "ls /" 能输出zookeeper说明集群运行正常

```
$ ./bin/zkCli.sh -server 172.27.128.31:6181,172.27.128.32:6181,172.27.128.33:6181
[zk: 172.27.128.31:6181,172.27.128.32:6181,172.27.128.33:6181(CONNECTED) 0] ls /
[zookeeper]
```

# 部署RTIDB

- 部署单机版
- 部署集群版

## 部署单机版

单机版可以只部署一个节点也可以部署两个节点的一主一从或者三个节点的一主两从模式。主从模式就是单独起两个或者三个tablet节点，通过建表和添加副本命令建立主从关系

### 1 下载RTIDB部署包. 如果已经有部署包就跳过这步

```
$ wget http://pkg.4paradigm.com:81/rtidb/rtidb-cluster-1.4.1.tar.gz
$ tar -zxvf rtidb-cluster-1.4.1.tar.gz
$ mv rtidb-cluster-1.4.1 rtidb-tablet-1.4.1
$ cd rtidb-tablet-1.4.1
```

### 2 修改配置文件

修改conf/tablet.flags中的endpoint. endpoint包括服务启动的地址和端口号

```
--endpoint=172.27.128.33:9527
```

注1: 如果此处使用的域名, 所有使用rtidb的client所在的机器都需要配上对应的host. 不然会访问不到

注2: 部署单机版时conf/tablet.flags配置中的zk\_cluster和zk\_root\_path需要注释掉

### 3 启动tablet

```
$ sh bin/start.sh start
```

## 部署集群版

RTIDB集群由nameserver和tablet两种组件组成。tablet存储表的数据, nameserver负责管理tablet以及failover等。部署集群时至少部署两个nameserver和三个tablet。

## 部署nameserver

### 1 下载RTIDB部署包

```
$ wget http://pkg.4paradigm.com:81/rtidb/rtidb-cluster-1.4.1.tar.gz
$ tar -zxvf rtidb-cluster-1.4.1.tar.gz
$ mv rtidb-cluster-1.4.1 rtidb-ns-1.4.1
```

```
$ cd rtidb-ns-1.4.1
```

## 2 修改配置文件

- 修改endpoint
- 修改zk\_cluster为已经启动的zk集群地址。ip为zk所在机器的ip, port为zk配置文件中clientPort配置的端口号。如果zk是集群模式用逗号分割, 格式为ip1:port1,ip2:port2,ip3:port3
- 如果和其他RTIDB共用zk需要修改zk\_root\_path

```
--endpoint=172.27.128.31:6527
--role=nameserver
--zk_cluster=172.27.128.33:7181,172.27.128.32:7181,172.27.128.31:7181
--zk_root_path=/rtidb_cluster
```

**注1:** endpoint不能用0.0.0.0和127.0.0.1

**注2:** 如果修改了db的路径, 同时也需要将recycle修改为db相同目录下

## 3 启动服务

```
$ sh bin/start_ns.sh start
```

**注1:** 服务启动后会在bin目录下产生ns.pid文件, 里边保存启动时的进程号。如果该文件内的pid正在运行则会启动失败

## 部署tablet

### 1 下载RTIDB部署包

```
$ wget http://pkg.4paradigm.com:81/rtidb/rtidb-cluster-1.4.1.tar.gz
$ tar -zxvf rtidb-cluster-1.4.1.tar.gz
$ mv rtidb-cluster-1.4.1 rtidb-tablet-1.4.1
$ cd rtidb-tablet-1.4.1
```

## 2 修改配置文件

- 修改endpoint
- 修改zk\_cluster为已经启动的zk集群地址
- 如果和其他RTIDB共用zk需要修改zk\_root\_path

```
--endpoint=172.27.128.33:9527
--role=tablet

# if tablet run as cluster mode zk_cluster and zk_root_path should be set
--zk_cluster=172.27.128.33:7181,172.27.128.32:7181,172.27.128.31:7181
--zk_root_path=/rtidb_cluster
```

注1: endpoint不能用0.0.0.0和127.0.0.1

注2: 如果此处使用的域名, 所有使用rtidb的client所在的机器都得配上对应的host. 不然会访问不到

注3: zk\_cluster和zk\_root\_path配置和nameserver的保持一致

### 3 启动服务

```
$ sh bin/start.sh start
```

注1: 服务启动后会在bin目录下产生tablet.pid文件, 里边保存启动时的进程号。如果该文件内的pid正在运行则会启动失败

## 部署monitor(该模块是收集监控数据的, 如qps、cpu内存磁盘等)

### 1 修改配置文件conf/monitor.conf

- 修改tablet\_endpoint为当前机器部署tablet的endpoint
- 修改tablet\_log\_dir为当前机器部署tablet的日志目录
- 如果当前机器部署了nameserver修改ns\_log\_dir为当前机器nameserver的日志目录, 如果没有部署nameserver就把这项注释掉

### 2 启动monitor

```
bash bin/start_monitor.sh start
```

- nameserver配置文件
- tablet配置文件
- monitor配置文件
- 配置优化

## nameserver配置文件

用#注释的配置一般不需要改动

- **endpoint** // 启动时指定的endpoint
- **role** // 启动时指定的角色, 如果启动的是nameserver就指定为nameserver
- **zk\_cluster** // 指定zk cluster的地址
- **zk\_root\_path** // 指定root path
- **log\_dir** // 配置日志路径
- **log\_file\_count** // 保留文件数量, 一般不需要改动
- **log\_file\_size** // 配置单个日志文件大小, 一般不需要改动
- **log\_level** // 配置日志级别, 建议在生产环境中为info级别
- **auto\_failover** // 配置是否开启auto failover, 默认是开启的. 如果设置开启就会自动做故障转移和恢复
- **thread\_pool\_size** // 配置brpc内部占用线程数
- **request\_max\_retry** // 请求失败时的最大重试次数
- **request\_timeout\_ms** // 请求的超时时间, 单位是ms
- **request\_sleep\_time** // 请求失败时的等待时间, 单位是ms
- **zk\_session\_timeout** // 设置zk session的超时时间, 单位是ms
- **zk\_keep\_alive\_check\_interval** // 检查和zk连接的时间间隔, 如果断开会自动重连. 单位是ms
- **tablet\_heartbeat\_timeout** // 配置心跳超时时间, 如果超过了配置的时间还没有连上zk就将相应节点置为下线状态
- **tablet\_offline\_check\_interval** // 如果nameserver检测到tablet下线, 以配置的时间间隔去检查是否上线直到超时
- **name\_server\_task\_pool\_size** // 配置nameserver任务线程池的大小
- **name\_server\_task\_wait\_time** // 任务执行框架中任务队列为空时的休眠时间, 单位是ms
- **get\_task\_status\_interval** // 配置获取tablet任务状态的时间间隔, 单位是ms
- **max\_op\_num** // 配置nameserver内存中保存op的最大数
- **replica\_num** // 创建表时默认分片数
- **partition\_num** // 创建表时默认副本数(包含主节点)
- **name\_server\_task\_concurrency** // 配置执行op的默认并发数
- **name\_server\_task\_max\_concurrency** // 配置最大的op并发数
- **name\_server\_op\_execute\_timeout** // 配置任务执行超时时间. 如果超过这个时间就会打印warning日志
- **get\_table\_status\_interval** // 更新表状态的时间间隔
- **check\_binlog\_sync\_progress\_delta** // 配置CheckBinlogSyncsProgress任务的offset阈值. 如果主从之间的offset之差小于该值就认为数据已追平

配置文件demo如下

```

# nameserver.conf
#endpoint不能用0.0.0.0和127.0.0.1
--endpoint=172.27.128.31:6527
--role=nameserver
--zk_cluster=172.27.128.33:7181,172.27.128.32:7181,172.27.128.31:7181
--zk_root_path=/rtidb_cluster

--log_dir=./logs
--log_file_count=24
--log_file_size=1024
--log_level=info

--auto_failover=true

#--thread_pool_size=16
#--request_max_retry=3
--request_timeout_ms=5000
#--request_sleep_time=1000

--zk_session_timeout=10000
#--zk_keep_alive_check_interval=15000
--tablet_heartbeat_timeout=300000
#--tablet_offline_check_interval=1000

#--name_server_task_pool_size=8
#--name_server_task_concurrency=2
#--name_server_task_max_concurrency=8
#--name_server_task_wait_time=1000
#--name_server_op_execute_timeout=7200000
#--get_task_status_interval=2000
#--get_table_status_interval=2000
#--check_binlog_sync_progress_delta=100000
#--max_op_num=10000

#--replica_num=3
#--partition_num=16

```

## tablet配置文件

- **endpoint** // 指定节点的ip和port, 示例172.27.128.31:9991(注:endpoint不能用0.0.0.0和127.0.0.1)
- **role** // 启动时指定的角色, 如果启动的是tablet就指定为tablet
- **zk\_cluster** // 集群模式时指定连接zookeeper集群的地址, 如果zookeeper是多节点用逗号分开, 示例 172.27.2.51:6330,172.27.2.52:6331
- **zk\_root\_path** // 指定在zk写入数据的根节点
- **thread\_pool\_size** // 配置brpc内部占用线程数
- **scan\_concurrency\_limit** // 配置scan的最大并发数
- **put\_concurrency\_limit** // 配置put的最大并发数
- **get\_concurrency\_limit** // 配置get的最大并发数

- **zk\_session\_timeout** // 设置zk session的超时时间, 单位是ms
- **zk\_keep\_alive\_check\_interval** // 检查和zk连接的时间间隔, 如果断开会自动重连. 单位是ms
- **log\_dir** // 配置日志文件目录
- **log\_file\_count** // 保留日志文件最大个数
- **log\_file\_size** // 单个日志文件的最大值, 单位是MB
- **log\_level** // 配置日志级别
- **binlog\_coffee\_time** // 没有读出最新数据的等待时间, 单位是ms
- **binlog\_match\_logoffset\_interval** // 同步从节点offset任务的时间间隔, 单位是ms
- **binlog\_notify\_on\_put** // 配置为true时, 有写操作会立即同步到从节点
- **binlog\_single\_file\_max\_size** // 配置binlog文件的大小, 单位是byte
- **binlog\_sync\_batch\_size** // 主从同步时一次同步的最大记录条数
- **binlog\_sync\_to\_disk\_interval** // 数据刷磁盘的时间间隔, 单位是ms
- **binlog\_sync\_wait\_time** // 主从同步时没有数据同步的等待时间, 单位是ms
- **binlog\_name\_length** // 配置binlog名字的长度
- **binlog\_delete\_interval** // 删除binlog任务的时间间隔, 单位是ms
- **binlog\_enable\_crc** // 配置binlog是否要开启crc校验
- **io\_pool\_size** // 执行io任务的线程池大小
- **task\_pool\_size** // tablet线程池的大小
- **db\_root\_path** // 配置binlog和snapshot的存放目录
- **recycle\_bin\_root\_path** // 配置droptable回收站的目录
- **make\_snapshot\_time** // 配置make snapshot的时间, 如配置为23表示每天23点make snapshot
- **make\_snapshot\_check\_interval** // 检查make snapshot的时间间隔, 单位是ms
- **make\_snapshot\_threshold\_offset** // 执行makesnapshot的offset阈值, 当前offset和上次makesnapshot的offset之差大于这个值才会做snapshot
- **gc\_interval** // 执行过期键删除任务的时间间隔, 单位是分钟
- **gc\_pool\_size** // 执行过期删除任务的线程池大小
- **gc\_safe\_offset** // ttl时间的偏移, 单位是分钟. 一般不用配置该项
- **send\_file\_max\_try** // 发送文件失败时的最大重试次数
- **retry\_send\_file\_wait\_time\_ms** // 发送文件失败时重试等待时间.
- **stream\_close\_wait\_time\_ms** // 发送完一个文件的等待时间
- **stream\_block\_size** // streaming一次发送数据块的最大值, 单位是byte
- **stream\_bandwidth\_limit** // streaming方式发送数据时的最大速度, 单位是byte/s
- **request\_max\_retry** // 请求失败时的最大重试次数
- **request\_timeout\_ms** // 请求失败时的最大重试次数
- **request\_sleep\_time** // 请求失败时的等待时间

配置文件demo如下

```
# tablet.conf
#endpoint不能用0.0.0.0和127.0.0.1
--endpoint=172.27.128.33:9527
--role=tablet

# if tablet run as cluster mode zk_cluster and zk_root_path should be set
--zk_cluster=172.27.128.33:7181,172.27.128.32:7181,172.27.128.31:7181
```

```

#--zk_root_path=/rtidb_cluster

# concurrency conf
# thread_pool_size建议和cpu核数一致
--thread_pool_size=24
--scan_concurrency_limit=16
--put_concurrency_limit=8
--get_concurrency_limit=16

--zk_session_timeout=10000
#--zk_keep_alive_check_interval=15000

# log conf
--log_dir=./logs
--log_file_count=24
--log_file_size=1024
--log_level=info

# binlog conf
#--binlog_coffee_time=1000
#--binlog_match_logoffset_interval=1000
--binlog_notify_on_put=true
--binlog_single_file_max_size=2048
#--binlog_sync_batch_size=32
--binlog_sync_to_disk_interval=5000
#--binlog_sync_wait_time=100
#--binlog_name_length=8
#--binlog_delete_interval=60000
#--binlog_enable_crc=false

#--io_pool_size=2
#--task_pool_size=8

--db_root_path=./db
--recycle_bin_root_path=./recycle

# snapshot conf
# 每天23点做snapshot
--make_snapshot_time=23
#--make_snapshot_check_interval=600000
#--make_snapshot_threshold_offset=100000

# garbage collection conf
# 60m
--gc_interval=60
#--gc_pool_size=2
# 1m
#--gc_safe_offset=1

# send file conf
#--send_file_max_try=3

```

```
#--stream_close_wait_time_ms=1000  
#--stream_block_size=1048576  
--stream_bandwidth_limit=10485760  
#--request_max_retry=3  
#--request_timeout_ms=5000  
#--request_sleep_time=1000  
#--retry_send_file_wait_time_ms=3000
```

## monitor配置文件

```
# port不用改  
port=8000  
# 数据采样频率, 默认为1秒  
interval=1  
# monitor log目录  
log_dir=./logs  
  
tablet_endpoint=172.27.128.37:9827  
# 要监控tablet的log目录  
tablet_log_dir=./logs  
  
# 要监控nameserver的log目录  
ns_log_dir=/home/denglong/env/rtidb_test/ns/logs
```

## 配置优化

- 针对高并发的场景, tablet将以下配置加大些
  - thread\_pool\_size
  - scan\_concurrency\_limit
  - put\_concurrency\_limit
  - get\_concurrency\_limit
- 如果部署目录空间较小, 可以将数据放到其他目录上
  - db\_root\_path
  - recycle\_bin\_root\_path

## 目录介绍

RTIDB部署目录结构如下

- bin // 脚本和二进制存放目录
  - rtidb // RTIDB二进制文件
  - mon // 守护进程文件
  - start.sh // tablet启动脚本
  - start\_ns.sh // nameserver启动脚本
- conf
  - tablet.flags // tablet配置文件
  - nameserver.flags // nameserver配置文件
- db // binlog和snapshot存放目录. nameserver没有此目录
  - tid\_pid tid是表的id, pid是分片id. 有多少个(tid, pid)对, 就有多少个对应的目录
    - table\_meta.txt // 记录表元数据文件, 创建表的一些信息可以从该文件中查到
    - binlog // 该目录下放的是binlog文件
    - snapshot // 该目录下放的是snapshot相关文件
      - MANIFEST // 该文件是对snapshot文件的一个汇总, 包含offset, snapshot文件名, 记录条数和term
      - xxx.sdb // snapshot文件
- logs // 日志目录
  - rtidb\_mon.log // 守护进程启动tablet的日志文件
  - rtidb\_ns\_mon.log // 守护进程启动nameserver的日志文件
  - nameserver.info.log // nameserver日志文件
  - tablet.info.log // tablet日志文件
- recycle // 回收站目录. drop table之后就会把db目录下对应的tid\_pid目录mv到这个目录下.

# 快速入门

## 创建表

### 1 命令行方式创建

```
$ ./bin/rtidb --zk_cluster=172.27.2.52:12200 --zk_root_path=/onebox --role=ns_client
> create test 144000 8 3 card:string:index mcc:string:index value:float
Create table ok
> showschema test
#   name      type      index
-----
0   card      string    yes
1   mcc       string    yes
2   value     float     no
> showtable test
  name  tid  pid  endpoint          role      ttl      is_alive  compress_type
  offset record_cnt  memused
-----
test  23   0   172.27.128.37:9992  leader    144000min  yes      kNoCompress
0      0      0.000
test  23   0   172.27.128.37:9993  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   0   172.27.128.37:9991  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   1   172.27.128.37:9993  leader    144000min  yes      kNoCompress
0      0      0.000
test  23   1   172.27.128.37:9991  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   1   172.27.128.37:9992  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   2   172.27.128.37:9992  leader    144000min  yes      kNoCompress
0      0      0.000
test  23   2   172.27.128.37:9993  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   2   172.27.128.37:9991  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   3   172.27.128.37:9992  leader    144000min  yes      kNoCompress
0      0      0.000
test  23   3   172.27.128.37:9993  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   3   172.27.128.37:9991  follower  144000min  yes      kNoCompress
0      0      0.000
test  23   4   172.27.128.37:9993  leader    144000min  yes      kNoCompress
0      0      0.000
```

test	23	4	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	4	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	5	172.27.128.37:9991	leader	144000min	yes	kNoCompress
0	0	0.000					
test	23	5	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	5	172.27.128.37:9993	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	6	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0.000					
test	23	6	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	6	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	7	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0.000					
test	23	7	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	7	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0.000					

## 2 用建表文件创建

准备一个建表文件table\_meta.txt

```

name : "test"
ttl: 144000
ttl_type : "kAbsoluteTime"
partition_num: 8
replica_num: 3
column_desc {
    name : "card"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "mcc"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "amt"
    type : "float"
    add_ts_idx : false
}

```

连上ns client创建表

```

$ ./bin/rtidb --zk_cluster=172.27.2.52:12200 --zk_root_path=/onebox --role=ns_client
> create table_meta.txt
Create table ok
> showschema test
#   name    type     index
-----
0   card    string   yes
1   mcc     string   yes
2   amt     float    no
> showtable test
  name    tid    pid    endpoint           role      ttl       is_alive compress_type
  offset   record_cnt   memused
-----
test  23    0    172.27.128.37:9992  leader  144000min  yes  kNoCompress
0      0      0.000
test  23    0    172.27.128.37:9993  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    0    172.27.128.37:9991  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    1    172.27.128.37:9993  leader  144000min  yes  kNoCompress
0      0      0.000
test  23    1    172.27.128.37:9991  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    1    172.27.128.37:9992  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    2    172.27.128.37:9992  leader  144000min  yes  kNoCompress
0      0      0.000
test  23    2    172.27.128.37:9993  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    2    172.27.128.37:9991  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    3    172.27.128.37:9992  leader  144000min  yes  kNoCompress
0      0      0.000
test  23    3    172.27.128.37:9993  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    3    172.27.128.37:9991  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    4    172.27.128.37:9993  leader  144000min  yes  kNoCompress
0      0      0.000
test  23    4    172.27.128.37:9991  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    4    172.27.128.37:9992  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    5    172.27.128.37:9991  leader  144000min  yes  kNoCompress
0      0      0.000
test  23    5    172.27.128.37:9992  follower 144000min  yes  kNoCompress
0      0      0.000
test  23    5    172.27.128.37:9993  follower 144000min  yes  kNoCompress

```

0	0	0.000					
test	23	6	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0.000					
test	23	6	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	6	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	7	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0.000					
test	23	7	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0.000					
test	23	7	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0.000					

## 插入和查询数据

```
$ ./bin/rtidb --zk_cluster=172.27.2.52:12200 --zk_root_path=/onebox --role=ns_client
> put test 1534992854000 card0 mcc0 1.5
Put ok
> put test 1534992888000 card0 mcc1 12.3
Put ok
> put test 1534992910000 card1 mcc1 10.7
Put ok
> scan test card0 card 1534992910000 0
# ts          card  mcc  amt
-----
1 1534992888000  card0  mcc1  12.3000002
2 1534992854000  card0  mcc0  1.5
> scan test card1 card 1534992910000 0
# ts          card  mcc  amt
-----
1 1534992910000  card1  mcc1  10.6999998
> scan test mcc1 mcc 1534992910000 0
# ts          card  mcc  amt
-----
1 1534992910000  card1  mcc1  10.6999998
2 1534992888000  card0  mcc1  12.3000002
> get test card0 card 1534992854000
# ts          card  mcc  amt
-----
1 1534992854000  card0  mcc0  1.5
> get test card0 card 0
# ts          card  mcc  amt
-----
1 1534992888000  card0  mcc1  12.3000002
```



RTIDB命令行有两种client: tablet client和ns client, 分别连到不同的server. 由启动时的--role来指定

- ns client
- tablet client

## NS Client

连接到ns client 需要指定zk\_cluster、zk\_root\_path和role。其中zk\_cluster是zk地址, zk\_root\_path是集群在zk的根路径

```
$ ./bin/rtidb --zk_cluster=172.27.2.52:12200 --zk_root_path=/onebox --role=ns_client
```

- [create](#) 创建表
- [showtable](#) 查看表信息
- [showschema](#) 查看表schema
- [drop](#) 删除表
- [put](#) 插入数据
- [scan](#) 查询一个区间的数据
- [get](#) 查询单条数据
- [delete](#) 删除pk
- [count](#) 统计pk下的记录条数
- [preview](#) 数据预览
- [showtablet](#) 查看tablet信息
- [addreplica](#) 添加副本
- [delreplica](#) 删除副本
- [makesnapshot](#) 产生snapshot
- [migrate](#) 副本迁移
- [confset](#) 修改配置
- [confget](#) 获取配置
- [offlineendpoint](#) 下线节点
- [recoverendpoint](#) 恢复节点数据
- [changeleader](#) 分片主从切换
- [recoverable](#) 恢复分片数据
- [showopstatus](#) 显示操作执行信息
- [cancelop](#) 取消操作
- [gettablepartition](#) 导出分片数据信息
- [settablepartition](#) 修改分片数据信息
- [updatetablealive](#) 修改分片alive状态
- [settll](#) 修改ttl
- [showsns](#) 显示nameserver节点和角色
- [exit](#) 退出客户端
- [quit](#) 退出客户端
- [help](#) 获取帮助信息

- `man` 获取帮组信息

## create

创建表有命令行方式和表元数据文件两种方式

### 1 命令行方式

命令格式: `create table_name ttl partition_num replica_num [column_name1:type:index column_name2:type ...]`

- `table_name` 表名
- `ttl` 设置ttl。如果absolute表, 直接设置为ttl大小, 单位是分钟, 如果设置为0表示不过期。如果创建latest表, 格式为latest:num, 其中冒号后面的数字为一个key下保留的最大记录条数
- `partition_num` 分片数目
- `replica_num` 副本数
- schema(可选) 格式为 `column_name:type[:index]`, 多列之间以空格分割。支持的type有int16, uint16, int32, uint32, int64, uint64, float, double, string。如果某一列是index, 则注明为index
  - `card:string:index` 列名为card, 类型是string, 这一列是index
  - `money:float` 列名是money, 类型是float

示例1: 创建名字为test1的kv表, ttl类型是absolute, ttl大小是144000分钟, 分片数设置为8, 三副本

```
> create test1 144000 8 3
Create table ok
```

示例2: 创建test2的kv表, ttl类型为latest, ttl大小为1保留最近一条记录, 分片数设置为1, 副本数设置为2

```
> create test2 latest:1 1 2
Create table ok
```

示例3: 创建表名为test3的有schema的表, ttl类型为absolute, ttl大小是100分钟, 分片数设置为8, 副本数设置为1, 有三列分别为card、mcc和money, card和mcc是index类型是string, money的类型为float

```
> create test3 100 8 1 card:string:index mcc:string:index money:float
```

创建表可以通过纯粹命令行方式和表元数据文件方式

### 2 指定文件方式

命令格式: `create table_meta_file_path`

- `table_meta_file_path` 指定表文件路径

表文件格式如下:

```

name : "test"          #表名 注意加双引号
ttl_type : "kAbsoluteTime" #指定ttl类型, 这个字段是可选的, 如果没有这个字段默认为absolute
。如果要创建latest表需指定为latest或者kLatestTime
ttl: 144000           #ttl大小, 如果ttl_type是absolute表示过期时间单位是分钟, 如果t
tl_type是latest表示保留的最大记录条数
partition_num: 8      #指定分片数, 这个字段是可选的, 如果没有这个字段默认分片数是8
replica_num: 3         #指定副本数, 这个字段是可选的, 如果没有这个字段默认副本数是3。副
本数表示一份数据保存多少份, 如果设置为3表示数据保存三份, 一主两从模式
compress_type: "snappy" #指定压缩类型, 这个字段是可选的, 如果没有这个字段默认不压缩数据。
取值为[nocompress, snappy]
key_entry_max_height: 8 #指定第二层skiplist的最大高度, 这个字段是可选的. absolute表默
认是4, latest表默认为1
column_desc {
    name : "card"          #column_desc指定schema信息, 如果是kv表不用设置column_desc
    type : "string"        #指定列名为card
    add_ts_idx : true       #指定这一列的类型为string。目前支持的类型有int32, uint32, int
64, uint64, float, double, string
}
column_desc {           #有多少列就设置多少个column_desc结构
    name : "mcc"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "amt"
    type : "float"
    add_ts_idx : false
}

```

设置好表文件后运行创建命令即可创建表

```
> create table_file_path.txt
```

示例1: 创建名字为test1的kv表, ttl类型是absolute, 过期时间为144000分钟, 分片数设置为16, 三副
本。保存文件名为test1.txt

```

name : "test1"
ttl: 144000

```

```
> create test1.txt
Create table ok
```

示例2: 创建名字为test2的kv表, ttl类型是latest, 保留最近一条记录, 分片数设置为8, 三副本。后面示
例省略了创建步骤, 只给出表文件

```
name : "test2"
```

```
ttl_type: "latest"
ttl: 1
partition_num: 8
```

示例3: 创建名字为test3的kv表, ttl类型是absolute, 设置数据永不过期, 数据用snappy压缩, 分片数设置为8, 两副本

```
name : "test3"
ttl: 0
partition_num: 8
replica_num: 2
compress_type: "snappy"
```

示例4: 创建名字为test4的有schema的表, ttl类型是absolute, 过期时间为120分钟, 分片数为16, 三副本, 总共三列分别为card、mcc、money, 其中card和mcc是索引维度列类型是string, money列类型是float

```
name : "test4"
ttl: 120
column_desc {
    name : "card"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "mcc"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "amt"
    type : "float"
    add_ts_idx : false
}
```

示例4: 创建名字为test5的有schema的表, 指定时间戳列为ts1(只有type为int64,uint64和timestamp的列才能指定为时间戳列)

```
name : "test5"
ttl: 120
column_desc {
    name : "card"
    type : "string"
    add_ts_idx : true
}
column_desc {
    name : "mcc"
    type : "string"
```

```

    add_ts_idx : true
}
column_desc {
  name : "amt"
  type : "float"
  add_ts_idx : false
}
column_desc {
  name : "ts1"
  type : "int64"
  is_ts_col : true
}

```

示例5: 创建名字为test6表, 指定时间戳列为ts1和ts2, 其中ts2的ttl为100. card和mcc为组合key对应的时间戳为ts1, card对应的时间戳列为ts1和ts2

```

name : "test5"
ttl: 120
column_desc {
  name : "card"
  type : "string"
  add_ts_idx : true
}
column_desc {
  name : "mcc"
  type : "string"
  add_ts_idx : true
}
column_desc {
  name : "amt"
  type : "float"
}
column_desc {
  name : "ts1"
  type : "int64"
  is_ts_col : true
}
column_desc {
  name : "ts2"
  type : "timestamp"
  is_ts_col : true
  ttl : 100
}
column_key {
  index_name : "card_mcc"
  col_name : "card"
  col_name : "mcc"
  ts_name : "ts1"
}
column_key {

```

```

    index_name : "card"
    ts_name : "ts1"
    ts_name : "ts2"
}

```

注: latest表过期时间最大为1000条, absolute表ttl最大值为30年(15768000)

## showtable

showtable可以查看所有表, 也可以指定看某一个表

命令格式: showtable [table\_name]

```

> showtable
      name      tid   pid endpoint           role      ttl       is_alive compress_typ
      e     offset record_cnt memused
      -----
      flow      24     0   172.27.128.37:9993  leader    0min      yes      kNoCompress
      0         0          0.000
      flow      24     0   172.27.128.37:9991  follower  0min      yes      kNoCompress
      0         0          0.000
      flow      24     0   172.27.128.37:9992  follower  0min      yes      kNoCompress
      0         0          0.000
      flow      24     1   172.27.128.37:9991  leader    0min      yes      kNoCompress
      0         0          0.000
      flow      24     1   172.27.128.37:9992  follower  0min      yes      kNoCompress
      0         0          0.000
      flow      24     1   172.27.128.37:9993  follower  0min      yes      kNoCompress
      0         0          0.000
      test     23     0   172.27.128.37:9992  leader   144000min  yes      kNoCompress
      0         0          0.000
      test     23     0   172.27.128.37:9993  follower 144000min  yes      kNoCompress
      0         0          0.000
      test     23     0   172.27.128.37:9991  follower 144000min  yes      kNoCompress
      0         0          0.000
      test     23     1   172.27.128.37:9993  leader   144000min  yes      kNoCompress
      0         0          0.000
      test     23     1   172.27.128.37:9991  follower 144000min  yes      kNoCompress
      0         0          0.000
      test     23     1   172.27.128.37:9992  follower 144000min  yes      kNoCompress
      0         0          0.000
      test     23     2   172.27.128.37:9992  leader   144000min  yes      kNoCompress
      0         0          0.000
      test     23     2   172.27.128.37:9993  follower 144000min  yes      kNoCompress
      0         0          0.000
      test     23     2   172.27.128.37:9991  follower 144000min  yes      kNoCompress
      0         0          0.000
      test     23     3   172.27.128.37:9992  leader   144000min  yes      kNoCompress
      0         0          0.000

```

test	23	3	172.27.128.37:9993	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	3	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	4	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	4	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	4	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	5	172.27.128.37:9991	leader	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	5	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	5	172.27.128.37:9993	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	6	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	6	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	6	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	7	172.27.128.37:9993	leader	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	7	172.27.128.37:9991	follower	144000min	yes	kNoCompress
0	0	0	0.000				
test	23	7	172.27.128.37:9992	follower	144000min	yes	kNoCompress
0	0	0	0.000				
> showtable flow							
name tid pid endpoint role ttl is_alive compress_type off							
et record_cnt memused							
-----							
flow	24	0	172.27.128.37:9993	leader	0min	yes	kNoCompress
0	0	0	0.000				0
flow	24	0	172.27.128.37:9991	follower	0min	yes	kNoCompress
0	0	0	0.000				0
flow	24	0	172.27.128.37:9992	follower	0min	yes	kNoCompress
0	0	0	0.000				0
flow	24	1	172.27.128.37:9991	leader	0min	yes	kNoCompress
0	0	0	0.000				0
flow	24	1	172.27.128.37:9992	follower	0min	yes	kNoCompress
0	0	0	0.000				0
flow	24	1	172.27.128.37:9993	follower	0min	yes	kNoCompress
0	0	0	0.000				0

## showschema

查看表的schema信息

命令格式: showschema table\_name

```
> showschema flow
table flow has not schema
> showschema test
# name type index
-----
0 card string yes
1 mcc string yes
2 amt float no
> showschema test5
# name type
-----
0 card string
1 mcc string
2 amt float
3 ts1 int64
4 ts2 timestamp

#ColumnKey
# index_name col_name ts_col ttl
-----
0 card_mcc card|mcc ts1 120min
1 card card ts1 120min
2 card card ts2 100min
```

## drop

删除表

命令格式: drop table\_name

```
> drop test
Drop table test? yes/no
yes
drop ok
```

## put

插入数据

### 1 kv表插入

命令格式: put table\_name pk ts value

- table\_name 表名
- pk 主键
- ts 时间戳, 精确到毫秒, ts不能为0

- value 主键对应的值

```
> put flow key1 1535371622000 value1
Put ok
```

## 2 schema表插入

命令格式: put table\_name ts value1 value2 value3 ...

- table\_name 表名
- ts 时间戳, 精确到毫秒, **ts不能为0**
- schema各字段对应的value, 字段先后顺序可以执行showschemata查看

```
> showschema test
# name type index
-----
0 card string yes
1 mcc string yes
2 amt float no
> put test 1535371622000 card0 mcc0 1.5
Put ok
```

## 3 有指定时间列的表插入

命令格式: put table\_name value1 value2 value3 ...

- table\_name 表名
- schema各字段对应的value, 字段先后顺序可以执行showschemata查看

```
172.27.128.37:6627> showschema test5
# name type
-----
0 card string
1 mcc string
2 amt float
3 ts1 int64
4 ts2 timestamp

#ColumnKey
# index_name col_name ts_col ttl
-----
0 card_mcc card|mcc ts1 120min
1 card card ts1 120min
2 card card ts2 100min
> put test card0 mcc0 1.5 1560859166000 1560859166001
Put ok

### scan {#scan}
```

查询一定范围内的数据

```
##### 1 查询kv表数据
```

命令格式: scan table\\_name pk start\\_time end\\_time \[limit\]

- \* table\\_name 表名
- \* pk 主键
- \* start\\_time 查询范围的起始时间
- \* end\\_time 查询范围的结束时间, 如果end\\_time设置为0查询从起始时刻的所有数据
- \* limit 限制最多返回条数, 该字段是可选的, 如果不设置就返回查询范围内所有数据

```
scan flow key1 1535372567000 153537100000
```

## Time Data

```
1 1535372567000 value3 2 1535372566000 value2 scan flow key1 1535372567000 0
```

## Time Data

```
1 1535372567000 value3 2 1535372566000 value2 3 9527 value1 scan flow key1  
1535372567000 0 2
```

## Time Data

```
1 1535372567000 value3 2 1535372566000 value2 `
```

## 2 查询schema表数据

命令格式: scan table\\_name key col\\_name start\\_time end\\_time [limit]

- table\\_name 表名
- key 主键名
- col\\_name 要查询的主键所在的列名, 只有创建表时设置为index列才能查询
- start\\_time 查询范围的起始时间
- end\\_time 查询范围的结束时间, 如果end\\_time设置为0查询从起始时刻的所有数据
- limit 限制最多返回条数, 该字段是可选的, 如果不设置就返回查询范围内所有数据

```
> scan test card0 card 1535373299010 1535373271000
# ts          card    mcc    amt
-----
1 1535373299010  card0  mcc2  5.80000019
2 1535373272000  card0  mcc1  10.3999996
> scan test card0 card 1535373299010 0
# ts          card    mcc    amt
-----
```

```

1 1535373299010 card0 mcc2 5.80000019
2 1535373272000 card0 mcc1 10.3999996
3 1535371622000 card0 mcc0 1.5
>scan test card0 card 1535373299010 0 1
# ts           card   mcc   amt
-----
1 1535373299010 card0 mcc2 5.80000019

```

## delete

删除pk

命令格式: delete table\_name key [col\_name]

- table\_name 表名
- key 主键值
- col\_name 列名. 该字段是可选的, 在多维度下如果不指定列名默认是第一个索引列

```

> delete test key1
Delete ok
> delete test1 card0 card
Delete ok

```

## count

统计一个pk下的记录条数

命令格式: count table\_name key [col\_name] [filter\_expired\_data]

- table\_name 表名
- key 主键值
- col\_name 列名. 该字段时可选的
- filter\_expired\_data 是否过滤过期数据[true/false]. 默认值为false. 如果置为true, 会遍历pk下的链表性能较差

```

> count test card1077 card
count: 1041

```

## preview

数据预览

命令格式: preview table\_name [limit]

- table\_name 表名
- limit 限制条数. 该字段是可选的, 默认为100条

```
> preview test 10
#   ts          card      mcc      amt
-----
1 1551167860977 card1077  mcc1342  9.6655599454491412
2 1551167860879 card1077  mcc72    9.4099887366740891
3 1551167860815 card1077  mcc1208  9.870822430478027
4 1551167860715 card1077  mcc277  9.2822292386566794
5 1551167860629 card1077  mcc958  9.263361336226696
6 1551167860615 card1077  mcc547  9.2201535836470647
7 1551167860254 card1077  mcc769  9.9815119658220137
8 1551167860162 card1077  mcc592  9.5524055792939109
9 1551167860069 card1077  mcc87   9.2436255334913984
10 1551167860020 card1077  mcc637  9.0073365887995038
```

## get

查询单条数据

### 1 查询kv表数据

命令格式: get table\_name key ts

- table\_name 表名
- key 主键值
- ts 查询的ts, 如果是0则返回最新的一条

```
> get flow key1 1535372567000
value :value3
> get flow key1 1535372566000
value :value2
> get flow key1 0
value :value3
```

### 2 查询schema表数据

命令格式: get table\_name key col\_name ts

- table\_name 表名
- key 要get的键值
- col\_name 要查询的key所在的列名
- ts 查询的ts, 如果是0则返回最新的一条

```
> get test card0 card 1535373272000
#   ts          card      mcc      amt
-----
1 1535373272000 card0  mcc1  10.3999996
> get test card0 card 1535371622000
#   ts          card      mcc      amt
```

```

-----  

1 1535371622000 card0 mcc0 1.5  

> get test card0 card 0  

# ts          card    mcc    amt  

-----  

1 1535373299010 card0 mcc2 5.80000019

```

## showtablet

查看tablet信息

```

> showtablet
  endpoint           state      age
-----  

172.27.128.31:8541 kTabletHealthy 3d  

172.27.128.32:8541 kTabletHealthy 9h  

172.27.128.33:8541 kTabletHealthy 3d  

172.27.128.37:8541 kTabletOffline 3d

```

## addreplica

添加副本

命令格式: addreplica table\_name pid\_group endpoint

- table\_name 表名
- pid\_group 分片id集合. 可以有以下几种情况
  - 单个分片
  - 多个分片, 分片间用逗号分割. 如1,3,5
  - 分片区间, 区间为闭区间. 如1-5表示分片1,2,3,4,5
- endpoint 要添加为副本的节点endpoint

```

> showtable test1
  name  tid  pid  endpoint           role      ttl  is_alive  compress_type  off
et  record_cnt  memused
-----  

-----  

test1 13  0    172.27.128.31:8541  leader    0min  yes    kNoCompress  0
      0    0.000  

test1 13  1    172.27.128.31:8541  leader    0min  yes    kNoCompress  0
      0    0.000  

test1 13  2    172.27.128.31:8541  leader    0min  yes    kNoCompress  0
      0    0.000  

test1 13  3    172.27.128.31:8541  leader    0min  yes    kNoCompress  0
      0    0.000  

> addreplica test1 0 172.27.128.33:8541
AddReplica ok
> showtable test1

```

```

      name  tid  pid  endpoint          role    ttl  is_alive  compress_type  off
et  record_cnt  memused
-----
test1  13   0   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
test1  13   0   172.27.128.33:8541  follower  0min  yes    kNoCompress  0
      0       0.000
test1  13   1   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
test1  13   2   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
test1  13   3   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
> addreplica test1 1,2,3 172.27.128.33:8541
AddReplica ok
> addreplica test1 1-3 172.27.128.33:8541
AddReplica ok

```

## delreplica

### 删除副本

命令格式: delreplica table\_name pid\_group endpoint

- table\_name 表名
- pid\_group 分片id集合. 可以有以下几种情况
  - 单个分片
  - 多个分片, 分片间用逗号分割. 如1,3,5
  - 分片区间, 区间为闭区间. 如1-5表示分片1,2,3,4,5
- endpoint 要删除副本的endpoint

```

> showtable test1
      name  tid  pid  endpoint          role    ttl  is_alive  compress_type  off
et  record_cnt  memused
-----
test1  13   0   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
test1  13   0   172.27.128.33:8541  follower  0min  yes    kNoCompress  0
      0       0.000
test1  13   1   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
test1  13   1   172.27.128.33:8541  follower  0min  yes    kNoCompress  0
      0       0.000
test1  13   2   172.27.128.31:8541  leader  0min  yes    kNoCompress  0
      0       0.000
test1  13   2   172.27.128.33:8541  follower  0min  yes    kNoCompress  0
      0       0.000

```

```

test1 13 3 172.27.128.31:8541 leader 0min yes kNoCompress 0
0 0.000
test1 13 3 172.27.128.33:8541 follower 0min yes kNoCompress 0
0 0.000
> delreplica test1 0 172.27.128.33:8541
DelReplica ok
> showtable test1
  name  tid  pid  endpoint          role    ttl  is_alive  compress_type  off
et  record_cnt  memused
-----
test1 13 0 172.27.128.31:8541  leader  0min  yes  kNoCompress 0
0 0.000
test1 13 1 172.27.128.31:8541  leader  0min  yes  kNoCompress 0
0 0.000
test1 13 1 172.27.128.33:8541  follower 0min  yes  kNoCompress 0
0 0.000
test1 13 2 172.27.128.31:8541  leader  0min  yes  kNoCompress 0
0 0.000
test1 13 2 172.27.128.33:8541  follower 0min  yes  kNoCompress 0
0 0.000
test1 13 3 172.27.128.31:8541  leader  0min  yes  kNoCompress 0
0 0.000
test1 13 3 172.27.128.33:8541  follower 0min  yes  kNoCompress 0
0 0.000
> delreplica test1 1,2,3 172.27.128.33:8541
DelReplica ok
> delreplica test1 1-3 172.27.128.33:8541
DelReplica ok

```

## makesnapshot

产生snapshot

命令格式: makesnapshot table\_name pid

- table\_name 表名
- pid 分片id

```

> makesnapshot flow 0
MakeSnapshot ok

```

## migrate

副本迁移

命令格式: migrate src\_endpoint table\_name pid\_group des\_endpoint

- src\_endpoint 需要迁出的节点

- `table_name` 表名
- `pid_group` 分片id集合. 可以有以下几种情况
  - 单个分片
  - 多个分片, 分片间用逗号分割. 如1,3,5
  - 分片区间, 区间为闭区间. 如1-5表示分片1,2,3,4,5
- `des_endpoint` 迁移的目的节点

```
> migrate 172.27.2.52:9991 table1 1 172.27.2.52:9992
partition migrate ok
> migrate 172.27.2.52:9991 table1 1-5 172.27.2.52:9992
partition migrate ok
> migrate 172.27.2.52:9991 table1 1,2,3 172.27.2.52:9992
partition migrate ok
```

## confget

获取配置信息, 目前只支持auto\_failover

命令格式: confget [conf\_name]

- `conf_name` 配置项名字, 可选的

```
> confget
  key          value
-----
  auto_failover    false
> confget auto_failover
  key          value
-----
  auto_failover  false
```

## confset

修改配置信息, 目前只支持auto\_failover

命令格式: confset conf\_name value

- `conf_name` 配置项名字
- `value` 配置项设置的值

```
> confset auto_failover true
set auto_failover ok
```

## offlineendpoint

下线节点。此命令是异步的返回成功后可通过showopstatus查看运行状态

命令格式: offlineendpoint endpoint [concurrency]

- endpoint是发生故障节点的endpoint。该命令会对该节点下所有分片执行如下操作:
  - 如果是主, 执行重新选主
  - 如果是从, 找到主节点然后从主节点中删除当前endpoint副本
  - 修改is\_alive状态为no
- concurrency 控制任务执行的并发数. 此配置是可选的, 默认为2(name\_server\_task\_concurrency配置可配), 最大值为name\_server\_task\_max\_concurrency配置的值

```
> offlineendpoint 172.27.128.32:8541
offline endpoint ok
>showtable
  name    tid   pid   endpoint           role     ttl      is_alive  compress_type
  offset   record_cnt   memused
-----
-----
  flow    4    0    172.27.128.32:8541  leader    0min      no       kNoCompress
  0        0          0.000
  flow    4    0    172.27.128.33:8541  follower  0min      yes      kNoCompress
  0        0          0.000
  flow    4    0    172.27.128.31:8541  follower  0min      yes      kNoCompress
  0        0          0.000
  flow    4    1    172.27.128.33:8541  leader    0min      yes      kNoCompress
  0        0          0.000
  flow    4    1    172.27.128.31:8541  follower  0min      yes      kNoCompress
  0        0          0.000
  flow    4    1    172.27.128.32:8541  follower  0min      no       kNoCompress
  0        0          0.000
```

该命令执行成功后所有分片都会有yes状态的leader

## recoverendpoint

恢复节点数据。此命令是异步的返回成功后可通过showopstatus查看运行状态

命令格式: recoverendpoint endpoint [need\_restore] [concurrency]

- endpoint是要恢复节点的endpoint
- need\_restore 表拓扑是否要恢复到最初的状态, 此配置是可选的, 默认为false. 如果设置为true, 一个分片在该节点下为leader, 执行完recoverendpoint恢复数据后依然是leader
- concurrency 控制任务执行的并发数. 此配置是可选的, 默认为2(name\_server\_task\_concurrency配置可配), 最大值为name\_server\_task\_max\_concurrency配置的值

```
> recoverendpoint 172.27.128.32:8541
recover endpoint ok
> recoverendpoint 172.27.128.32:8541 true
recover endpoint ok
> recoverendpoint 172.27.128.32:8541 true 3
```

```
recover endpoint ok
```

注:

1. 执行此命令前确保节点已经上线(**showtablet**命令查看)

## changeleader

对某个指定的分片执行主从切换。此命令是异步的返回成功后可通过**showopstatus**查看运行状态

命令格式: **changeleader** table\_name pid [candidate\_leader]

- **table\_name** 表名
- **pid** 分片id
- **candidate\_leader** 候选leader. 该参数是可选的, 如果设置成auto即使其他节点alive状态是yes也能切换

```
> changeleader flow 0
change leader ok
> changeleader flow 0 172.27.128.33:8541
change leader ok
> changeleader flow 0 auto
change leader ok
```

## recoverable

恢复某个分片数据。此命令是异步的返回成功后可通过**showopstatus**查看运行状态

命令格式: **recoverable** table\_name pid endpoint

- **table\_name** 表名
- **pid** 分片id
- **endpoint** 要恢复分片所在的节点endpoint

```
> recoverable flow 1 172.27.128.31:8541
recover table ok
```

## cancelop

取消一个正在执行或者待执行的操作. 取消后任务就状态变成kCanceled

命令格式: **cancelop** op\_id

- **op\_id** 需要取消的操作id

```
> cancelop 5
Cancel op ok!
```

## showopstatus

显示操作执行信息

命令格式: showopstatus [table\_name pid]

- table\_name 表名
- pid 分片id

```
> showopstatus
  op_id  op_type          name    pid  status   start_time      execute_time  e
  nd_time       cur_task
-----
  51      kMigrateOP        flow    0    kDone    20180824163316  12s           2
0180824163328  -
  52      kRecoverTableOP    flow    0    kDone    20180824195252  1s           2
0180824195253  -
  53      kRecoverTableOP    flow    1    kDone    20180824195252  1s           2
0180824195253  -
  54      kUpdateTableAliveOP  flow   0    kDone    20180824195838  2s           2
0180824195840  -
  55      kChangeLeaderOP    flow   0    kDone    20180824200135  0s           2
0180824200135  -
  56      kOfflineReplicaOP   flow   1    kDone    20180824200135  1s           2
0180824200136  -
  57      kReAddReplicaOP    flow   0    kDone    20180827114331  12s           2
0180827114343  -
  58      kAddReplicaOP      test1  0    kDone    20180827205907  8s           2
0180827205915  -
  59      kDelReplicaOP      test1  0    kDone    20180827210248  4s           2
0180827210252  -
> showopstatus flow
  op_id  op_type          name    pid  status   start_time      execute_time  e
  nd_time       cur_task
-----
  51      kMigrateOP        flow    0    kDone    20180824163316  12s           20
180824163328  -
  52      kRecoverTableOP    flow    0    kDone    20180824195252  1s           20
180824195253  -
  53      kRecoverTableOP    flow    1    kDone    20180824195252  1s           20
180824195253  -
  54      kUpdateTableAliveOP  flow   0    kDone    20180824195838  2s           20
180824195840  -
  55      kChangeLeaderOP    flow   0    kDone    20180824200135  0s           20
180824200135  -
  56      kOfflineReplicaOP   flow   1    kDone    20180824200135  1s           20
180824200136  -
  57      kUpdateTableAliveOP  flow   0    kDone    20180824200212  0s           20
180824200212  -
```

```
>showopstatus flow 1
  op_id  op_type          name  pid  status  start_time      execute_time en
d_time      cur_task
-----
-----  

  53    kRecoverTableOP      flow  1    kDone   20180824195252  1s        20
180824195253  -  

  56    kOfflineReplicaOP    flow  1    kDone   20180824200135  1s        20
180824200136  -
```

## gettablepartition

导出分片数据信息到当前目录，文件名为table\_name\_pid.txt

命令格式: gettablepartition table\_name pid

- table\_name 表名
- pid 分片id

```
> gettablepartition flow 0
get table partition ok
> quit
bye
$ cat flow_0.txt
pid: 0
partition_meta {
  endpoint: "172.27.128.32:8541"
  is_leader: true
  is_alive: false
}
partition_meta {
  endpoint: "172.27.128.33:8541"
  is_leader: true
  is_alive: false
}
partition_meta {
  endpoint: "172.27.128.31:8541"
  is_leader: true
  is_alive: true
}
term_offset {
  term: 15
  offset: 0
}
term_offset {
  term: 17
  offset: 1
}
term_offset {
  term: 19
```

```
    offset: 7154917
}
term_offset {
    term: 21
    offset: 7154917
}
term_offset {
    term: 23
    offset: 7154917
}
```

## settablepartition

修改分片数据信息

命令格式: settablepartition table\_name partition\_file\_path

- table\_name 表名
- partition\_file\_path 分片数据文件

用gettablepartiton导出相应文件后相应的修改

```
> settablepartition flow flow_0.txt
set table partition ok
```

注: 此命令会改编nameserver中分片的拓扑信息, 谨慎使用

## updatetablealive

修改分片alive状态

命令格式: updatetablealive table\_name pid endpoint is\_alive

- table\_name 表名
- pid 分片id, 如果要修改某个表的所有分片将pid指定为\*
- endpoint 节点endpoint
- is\_alive 节点状态, 只能填yes或者no

```
> updatetablealive test * 172.27.128.31:8541 no
update ok
> updatetablealive test 1 172.27.128.31:8542 no
update ok
```

## shows

显示nameserver节点及其角色

命令格式: shows

```
>shows
  endpoint          role
-----
  172.27.128.31:6527  leader
  172.27.128.32:6527  standby
  172.27.128.33:6527  standby
```

注: 此命令不可当做故障恢复使用. 一般用于切流量, 操作方式为将某个节点表的alive状态改为no读请求就不会落在该节点上

## setttl

修改ttl

命令格式: setttl table\_name ttl\_type ttl

- table\_name 表名
- ttl\_type 过期类型[absolute/latest]
- ttl ttl的值

```
> setttl test absolute 10000
Set ttl ok !
> setttl test_latest latest 5
Set ttl ok !
```

注1: 不能将ttl从非0值改为0, 也不能从0改为非0值

注2: 修改后在下一轮gc执行后才会生效

## exit

退出客户端

```
> exit
bye
```

## quit

退出客户端

```
> quit
bye
```

## help

获取帮助信息

命令格式: help [cmd]

- cmd 命令, 这个参数是可选的, 如果不写就列出所有命令

```
> help
create - create table
drop - drop table
put - insert data into table
scan - get records for a period of time
get - get only one record
showtable - show table info
showtablet - show tablet info
showschema - show schema info
showopstatus - show op info
cancelop - cancel the op
makesnapshot - make snapshot
addreplica - add replica to leader
delreplica - delete replica from leader
migrate - migrate partition form one endpoint to another
confset - update conf
confget - get conf
changeleader - select leader again when the endpoint of leader offline
offlineendpoint - select leader and delete replica when endpoint offline
recoverable - recover only one table partition
recoverendpoint - recover all tables in endpoint when onlinemigrate - migrate parti
tion form one endpoint to another
gettablepartition - get partition info
settablepartition - update partition info
updatetablealive - update table alive status
setttl - set table ttl
showns - show nameserver role
exit - exit client
quit - exit client
help - get cmd info
man - get cmd info
> help create
desc: create table
usage: create table_meta_file_path
usage: create table_name ttl partition_num replica_num [colum_name1:type:index colu
m_name2:type ...]
ex: create ./table_meta.txt
ex: create table1 144000 8 3
ex: create table2 latest:10 8 3
ex: create table3 latest:10 8 3 card:string:index mcc:string:index value:float
```

## man

获取帮组信息

命令格式: man [cmd]

- cmd 命令，这个参数是可选的，如果不写就列出所有命令

```
> man
create - create table
drop - drop table
put - insert data into table
scan - get records for a period of time
get - get only one record
showtable - show table info
showtablet - show tablet info
showschemainfo - show schema info
showopstatus - show op info
cancelop - cancel the op
makesnapshot - make snapshot
addreplica - add replica to leader
delreplica - delete replica from leader
migrate - migrate partition form one endpoint to another
confset - update conf
confget - get conf
changeleader - select leader again when the endpoint of leader offline
offlineendpoint - select leader and delete replica when endpoint offline
recoverable - recover only one table partition
recoverendpoint - recover all tables in endpoint when onlinemigrate - migrate partition form one endpoint to another
gettablepartition - get partition info
settablepartition - update partition info
updatetablealive - update table alive status
setttl - set table ttl
showns - show nameserver role
exit - exit client
quit - exit client
help - get cmd info
man - get cmd info
> man drop
desc: drop table
usage: drop table_name
ex: drop table1
```

## Tablet Client

连接到tablet client需要指定endpoint和role

```
$ ./rtidb --endpoint=172.27.2.52:9520 --role=client
```

- [create](#) 创建表
- [screate](#) 创建多维表
- [drop](#) 删除表

- `put` 插入数据
- `sput` 给有schema的表插入数据
- `scan` 查询一定范围内的数据
- `sscan` 查询有schema表一定范围内的数据
- `get` 查询单条数据
- `sget` 查询有schema表的单条数据
- `delete` 删除pk
- `count` 统计某个pk下的记录条数
- `preview` 预览数据
- `addreplica` 添加副本
- `delreplica` 删除副本
- `makesnapshot` 产生snapshot
- `pausesnapshot` 暂停snapshot
- `recoversnapshot` 恢复snapshot
- `sendsnapshot` 发送snapshot
- `loadtable` 创建表并加载数据
- `changerole` 改变表的leader角色
- `setexpire` 设置是否过期
- `showschema` 查看表的schema
- `gettablestatus` 获取表信息
- `getfollower` 查看从节点信息
- `setttl` 设置ttl
- `setlimit` 设置并发限制
- `exit` 退出客户端
- `quit` 退出客户端
- `help` 获取帮助信息
- `man` 获取帮组信息

## create

创建表

命名格式: `create table_name tid pid ttl segment_cnt [is_leader compress_type]`

- `table_name` 为要创建表名称
- `tid` 指定table的id
- `pid` 指定table的分片id
- `ttl` 指定过期时间,默认过期类型为absolute. absolute表ttl单位为分钟, 如果设置为0表示不过期。如果过期类型设置latest, 格式为latest:保留条数(ex: latest:10, 保留最近10条)
- `segment_cnt` 为表的segement 个数, 一般设置为8
- `is_leader` 指定是否为leader. 默认为leader
- `compress_type` 指定字段压缩类型. 取值为[nocompress, snappy], 默认不压缩

示例1 创建表名为test1的leader表, ttl类型为absolute过期时间为144000分钟。tid设置为1, pid为0

```
> create test1 1 0 144000 8
Create table ok
```

示例2 创建表名为test2的follower表， ttl类型为latest保留最近两条记录。tid设置为2， pid为0

```
> create test2 2 0 latest:2 8 false
Create table ok
```

示例3 创建表名为test3的leader表， ttl类型为absolute设置不过期， 数据用snappy压缩。tid设置为3， pid为0

```
> create test3 3 0 0 8 true snappy
Create table ok
```

**注: tid和pid唯一确定一张表， 创建时不能和已有表重复**

## screate

创建多维表

命名格式: screate table\_name tid pid ttl segment\_cnt is\_leader schema

- table\_name 为要创建表名称
- tid 指定table的id
- pid 指定table的分片id
- ttl 指定过期时间,默认过期类型为absolute. absolute表ttl单位为分钟, 如果设置为0表示不过期。如果过期类型设置latest, 格式为latest:保留条数(ex: latest:10, 保留最近10条)
- segment\_cnt 为表的segement 个数, 一般设置为8
- is\_leader 指定是否为leader. 默认为leader
- schema 格式为 colum\_name:type[:index], 多列之间以空格分割。支持的type有int32, uint32, int64, uint64, float, double, string。如果某一列是index, 则注明为index
  - card:string:index 列名为card, 类型是string, 这一列是index
  - money:float 列名是money, 类型是float

示例1 创建表名为test1的leader表, ttl类型为absolute过期时间为144000分钟。有card, mcc和money三列, 其中card和mcc为index列类型为string, money列的类型为float。tid设置为1, pid为0

```
> screate test1 1 0 144000 8 true card:string:index mcc:string:index money:float
Create table ok
```

示例2 创建表名为test2的follower表, ttl类型为latest最多保留最近两条记录。有card, mcc和money三列, 其中card是index列类型为uint64, mcc列类型为string, money列的类型为float。tid设置为2, pid为0

```
> screate test2 2 0 latest:2 8 false card:uint64:index mcc:string money:float
Create table ok
```

## drop

删除表

命令格式: drop tid pid

- tid 指定table的id
- pid 指定table的分片id

```
> drop 1 0  
Drop table ok
```

## put

插入数据

命令格式: put tid pid pk ts value

- tid 指定table的id
- pid 指定table的分片id
- pk 主键
- ts 时间戳, 精确到毫秒, ts不能为0
- value 主键对应的值

```
> put 2 0 key1 1535371622000 value1  
Put ok
```

## sput

给有schema的表插入数据

命令格式: sput tid pid ts value1 value2 value3 ...

- tid 指定table的id
- pid 指定table的分片id
- ts 时间戳, 精确到毫秒, ts不能为0
- schema 各字段对应的value, 字段先后顺序可以执行showschem查看

```
> sput 1 0 1535371622000 card0 mcc0 1.5  
Put ok
```

## scan

查询一定范围内的数据

命令格式: scan tid pid pk start\_time end\_time [limit]

- tid 指定table的id
- pid 指定table的分片id
- pk 主键
- start\_time 查询范围的起始时间
- end\_time 查询范围的结束时间 如果end\_time设置为0查询从起始时刻的所有数据
- limit 限制最多返回条数, 该字段是可选的, 如果不设置就返回查询范围内所有数据

```
> scan 2 0 test1 1535446708000 1535446686000
#      Time      Data
1    1535446708000    value3
2    1535446688000    value2
> scan 2 0 test1 1535446708000 0
#      Time      Data
1    1535446708000    value3
2    1535446688000    value2
3    1528858466000    value0
> scan 2 0 test1 1535446708000 0 1
#      Time      Data
1    1535446708000    value3
```

## sscan

查询有schema表一定范围内的数据

命令格式: sscan tid pid key col\_name start\_time end\_time [limit]

- tid 指定table的id
- pid 指定table的分片id
- key 主键
- col\_name 要查询的主键所在的列名, 只有创建表时设置为index列才能查询
- start\_time 查询范围的起始时间
- end\_time 查询范围的结束时间 如果end\_time设置为0查询从起始时刻的所有数据
- limit 限制最多返回条数, 该字段是可选的, 如果不设置就返回查询范围内所有数据

```
> sscan 1 0 card0 card 1535446708000 1535446687000
#      ts          card      mcc      money
-----
1  1535446708000  card0  mcc0  1.29999995
2  1535446688000  card0  mcc1  1.5
>sscan 1 0 card0 card 1535446708000 0
#      ts          card      mcc      money
-----
1  1535446708000  card0  mcc0  1.29999995
2  1535446688000  card0  mcc1  1.5
3  1528858466000  card0  mcc3  0.4000000006
>sscan 1 0 card0 card 1535446708000 0 1
#      ts          card      mcc      money
-----
```

```
1 1535446708000 card0 mcc0 1.29999995
```

## get

查询单条数据

命令格式: get tid pid key ts

- tid 指定table的id
- pid 指定table的分片id
- key 主键
- ts 时间戳, 如果是0则返回最新的一条

```
> get 2 0 test1 1535446688000
value :value2
> get 2 0 test1 0
value :value3
```

## sget

查询有schema表的单条数据

命令格式: sget tid pid key col\_name ts

- tid 指定table的id
- pid 指定table的分片id
- key 主键
- col\_name 要查询的key所在的列名
- ts 时间戳, 如果是0则返回最新的一条

```
> sget 1 0 card0 card 1535446688000
# ts          card    mcc    money
-----
1 1535446688000 card0  mcc1  1.5
> sget 1 0 card0 card 0
# ts          card    mcc    money
-----
1 1535446708000 card0  mcc0  1.29999995
```

## delete

删除pk

命令格式: delete tid pid key [col\_name]

- tid 指定table的id
- pid 指定table的分片id

- key 主键
- col\_name 要删除key所在的列名. 该字段是可选的

```
> delete 1 0 key1
Delete ok
```

## preview

预览数据

命令格式: preview tid pid [limit]

- tid 指定table的id
- pid 指定table的分片id
- limit 限定输出的条数. 该字段是可选的. 默认为100条

```
>preview 9 0 10
#      ts          card      mcc      amt
-----
1 1551168689441 card1077  mcc889  9.2206023558557888
2 1551168689431 card1077  mcc632  9.3409316056017353
3 1551168689285 card1077  mcc723  9.001503218597831
4 1551168689276 card1077  mcc951  9.4250239440250034
5 1551168689074 card1077  mcc101  9.6827419040776679
6 1551168688788 card1077  mcc993  9.0653753412878455
7 1551168688632 card1077  mcc659  9.8579586148719489
8 1551168688617 card1077  mcc249  9.3870601038906241
9 1551168688439 card1077  mcc448  9.1521059211468661
10 1551168688216 card1077  mcc112  9.7253728651383557
```

## count

统计某一pk下的记录条数

命令格式: count tid pid key [col\_name] [filter\_expired\_data]

- tid 指定table的id
- pid 指定table的分片id
- key 主键
- col\_name key所在的列名. 该字段是可选的
- filter\_expired\_data 是否过滤过期数据[true/false]. 默认值为false. 如果置为true, 会遍历pk下的链表  
性能较差

```
> count 9 0 card1077 card true
count: 7612
```

## addreplica

添加副本

命令格式: addreplica tid pid endpoint

- tid 指定table的id
- pid 指定table的分片id
- endpoint 要添加为副本的节点endpoint, 此endpoint上得有对应tid和pid的follower表

```
> addreplica 2 0 172.27.128.32:8541
AddReplica ok
```

## delreplica

删除副本

命令格式: delreplica tid pid endpoint

- tid 指定table的id
- pid 指定table的分片id
- endpoint 要删除副本的节点endpoint

```
> delreplica 2 0 172.27.128.32:8541
DelReplica ok
```

注: 删除副本操作并不会实际删除副本节点上的表, 只是断开了主从关系

## makesnapshot

产生snapshot, 会在db/tid\_pid/snapshot目录下产生snapshot文件

命令格式: makesnapshot tid pid

- tid 指定table的id
- pid 指定table的分片id

```
> makesnapshot 1 0
MakeSnapshot ok
```

## pausesnapshot

暂停snapshot, 执行完之后运行gettablestatus状态变为kSnapshotPaused

命令格式: pausesnapshot tid pid

- tid 指定table的id
- pid 指定table的分片id

```
> pausesnapshot 1 0
PauseSnapshot ok
> gettablestatus 1 0
  tid  pid  offset  mode          state          enable_expire  ttl          ttl_of
fset  memused  compress_type
-----
-----
  1    0     4      kTableLeader  kSnapshotPaused  true          144000min  0s
  1.313 K  kNoCompress
```

## recoversnapshot

恢复snapshot

命令格式: recoversnapshot tid pid

- tid 指定table的id
- pid 指定table的分片id

```
> recoversnapshot 1 0
RecoverSnapshot ok
```

## sendsnapshot

发送snapshot

命令格式: sensnapshot tid pid endpoint

- tid 指定table的id
- pid 指定table的分片id
- endpoint 将sendsnapshot发送到endpoint指定的节点上

```
> sendsnapshot 1 0 172.27.128.32:8541
SendSnapshot ok
```

注: 运行sendsnapshot前必须执行下pausesnapshot, 发送完再执行recoversnapshot

## loadtable

创建表并加载数据

命令格式: loadtable table\_name tid pid ttl segment\_cnt

- table\_name 表名
- tid 指定table的id
- pid 指定table的分片id
- ttl 指定ttl

- segment\_cnt 指定segment\_cnt, 一般设置为8

```
> loadtable table1 1 0 144000 8
```

注: loadtable创建的表是follower表

## changerole

改变表的leader角色

命令格式: changerole tid pid role [term]

- tid 指定table的id
- pid 指定table的分片id
- role 要修改的角色, 取值为[leader, follower]
- term 设定term, 该项是可选的, 默认为0

```
> changerole 1 0 follower
ChangeRole ok
> changerole 1 0 leader
ChangeRole ok
> changerole 1 0 leader 1002
ChangeRole ok
```

## setexpire

设置是否过期

命令格式: setexpire tid pid is\_expire

- tid 指定table的id
- pid 指定table的分片id
- is\_expire 是否过期, 取值为[true, false]

```
>setexpire 1 0 false
setexpire ok
>setexpire 1 0 true
setexpire ok
```

## showschema

查看表的schema

命令格式: showschema tid pid

- tid 指定table的id
- pid 指定table的分片id

```
> showschema 1 0
# name      type     index
-----
0 card      string   yes
1 mcc       string   yes
2 money     float    no
> showschema 2 0
No schema for table
```

## gettablestatus

获取表信息

命令格式: gettablestatus [tid pid]

- tid 指定table的id
- pid 指定table的分片id

```
> gettablestatus
tid  pid  offset  mode           state          enable_expire  ttl
    ttl_offset  memused  compress_type
-----
1    0    4        kTableLeader    kSnapshotPaused true           144000min
    0s          1.313 K  kNoCompress
2    0    4        kTableLeader    kTableNormal   false          0min
    0s          689.000  kNoCompress
> gettablestatus 2 0
tid  pid  offset  mode           state          enable_expire  ttl  ttl_offset  me
mused  compress_type
-----
2    0    4        kTableLeader    kTableNormal   false          0min  0s           68
9.000  kNoCompress
```

## getfollower

查看从节点信息

命令格式: getfollower tid pid

- tid 指定table的id
- pid 指定table的分片id

```
> getfollower 4 1
# tid  pid  leader_offset  follower          offset
-----
0  4    1    5923724        172.27.128.31:8541  5920714
```

```
1 4 1 5923724 172.27.128.32:8541 5921707
```

## setttl

命令格式: setttl tid pid ttl\_type ttl

- tid 指定table的id
- pid 指定table的分片id
- ttl\_type ttl类型
- ttl 设置ttl的值

```
> setttl 1 0 absolute 1000
Set ttl ok !
> setttl 2 0 latest 5
Set ttl ok !
```

注1: 不能将ttl从非0值改为0, 也不能从0改为非0值

注2: 修改后在下一轮gc执行后才会生效

## setlimit

命令格式: setlimit method limit

- method 方法名(区分大小写). 如果是Server表示整体最大并发数
- limit 最大并发数. 如果limit为0表示不限制

```
> setlimit Server 5
Set Limit ok
> setlimit Server 0
Set Limit ok
> setlimit Put 10
Set Limit ok
```

## exit

退出客户端

```
> exit
bye
```

## quit

退出客户端

```
> quit
```

```
bye
```

## help

获取帮助信息

命令格式: help [cmd]

```
> help
create - create table
screate - create multi dimension table
drop - drop table
put - insert data into table
sput - insert data into table of multi dimension
scan - get records for a period of time
sscan - get records for a period of time from multi dimension table
get - get only one record
sget - get only one record from multi dimension table
addreplica - add replica to leader
delreplica - delete replica from leader
makesnapshot - make snapshot
pausesnapshot - pause snapshot
recoversnapshot - recover snapshot
sendsnapshot - send snapshot to another endpoint
loadtable - create table and load data
changerole - change role
setexpire - enable or disable ttl
showschema - show schema
gettablestatus - get table status
getfollower - get follower
settll - set ttl for partition
setlimit - set tablet max concurrency limit
exit - exit client
quit - exit client
help - get cmd info
man - get cmd info
> help create
desc: create table
usage: create name tid pid ttl segment_cnt [is_leader compress_type]
ex: create table1 1 0 144000 8
ex: create table1 1 0 144000 8 true snappy
ex: create table1 1 0 144000 8 false
```

## man

获取帮组信息

命令格式: man [cmd]

```
> man
create - create table
screate - create multi dimension table
drop - drop table
put - insert data into table
sput - insert data into table of multi dimension
scan - get records for a period of time
sscan - get records for a period of time from multi dimension table
get - get only one record
sget - get only one record from multi dimension table
addreplica - add replica to leader
delreplica - delete replica from leader
makesnapshot - make snapshot
pausesnapshot - pause snapshot
recoversnapshot - recover snapshot
sendsnapshot - send snapshot to another endpoint
loadtable - create table and load data
changerole - change role
setexpire - enable or disable ttl
showschema - show schema
gettablestatus - get table status
getfollower - get follower
settll - set ttl for partition
setlimit - set tablet max concurrency limit
exit - exit client
quit - exit client
help - get cmd info
man - get cmd info
> man create
desc: create table
usage: create name tid pid ttl segment_cnt [is_leader compress_type]
ex: create table1 1 0 144000 8
ex: create table1 1 0 144000 8 true snappy
ex: create table1 1 0 144000 8 false
```

## java sdk文档

- 配置maven
- 集群模式使用说明
- 单机模式使用说明

## 配置maven pom

添加如下依赖配置, 其中version配置java sdk版本. 最新版本可以从[这里](#)获取

```
<dependency>
    <groupId>com._4paradigm</groupId>
    <artifactId>rtidb-client</artifactId>
    <version>1.4.1-RELEASE</version>
</dependency>
```

## 集群模式

```
package example;

import com._4paradigm.rtidb.client.GetFuture;
import com._4paradigm.rtidb.client.PutFuture;
import com._4paradigm.rtidb.client.ScanFuture;
import com._4paradigm.rtidb.client.KvIterator;
import com._4paradigm.rtidb.client.TableAsyncClient;
import com._4paradigm.rtidb.client.TableSyncClient;
import com._4paradigm.rtidb.client.TabletException;
import com._4paradigm.rtidb.client.ha.RTIDBClientConfig;
import com._4paradigm.rtidb.client.ha.impl.NameServerClientImpl;
import com._4paradigm.rtidb.client.ha.impl.RTIDBClusterClient;
import com._4paradigm.rtidb.client.ha.TableHandler.ReadStrategy;
import com._4paradigm.rtidb.client.impl.TableSyncClientImpl;
import com._4paradigm.rtidb.client.impl.TableAsyncClientImpl;
import com._4paradigm.rtidb.client.schema.ColumnDesc;
import com._4paradigm.rtidb.ns.NS;
import com.google.protobuf.ByteString;
import com._4paradigm.rtidb.common.Common.ColumnDesc;
import com._4paradigm.rtidb.common.Common.ColumnKey;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeoutException;

public class ClusterExample {
    private static String zkEndpoints = "172.27.128.31:8090,172.27.128.32:8090,172.
27.128.33:8090"; // 配置zk地址, 和集群启动配置中的zk_cluster保持一致
    private static String zkRootPath = "/rtidb_cluster"; // 配置集群的zk根路径, 和集
```

```

群启动配置中的zk_root_path保持一致
// 下面这几行变量定义不需要改
private static String leaderPath = zkRootPath + "/leader";
// NameServerClientImpl要么做成单例，要么用完之后就调用close，否则会导致fd泄露
private static NameServerClientImpl nsc = new NameServerClientImpl(zkEndpoints,
leaderPath);
private static RTIDBClientConfig config = new RTIDBClientConfig();
private static RTIDBClusterClient clusterClient = null;
// 发送同步请求的client
private static TableSyncClient tableSyncClient = null;
// 发送异步请求的client
private static TableAsyncClient tableAsyncClient = null;

static {
    try {
        nsc.init();
        config.setZkEndpoints(zkEndpoints);
        config.setZkRootPath(zkRootPath);
        // 设置读策略。默认是读local
        // 可以单独设置全局值和表级别的值，表级别优先于全局值
        // config.setGlobalReadStrategies(TableHandler.ReadStrategy.kReadLeader
    );
        // Map<String, ReadStrategy> strategy = new HashMap<String, ReadStrateg
y>();
        // strategy put传入的参数是表名和读策略。读策略可以设置为读leader(kReadLeader)
或者读本地(kReadLocal)。
        // 读本地的策略是客户端优先选择读取部署在当前机器的节点，如果当前机器没有部署tablet
则随机选择一个从节点读取，如果没有从节点就读主
        // strategy.put("test1", ReadStrategy.kReadLocal);
        // config.setReadStrategies(strategy);
        // 如果要过滤掉同一个pk下面相同ts的值添加如下设置
        // config.setRemoveDuplicateByTime(true);
        // 设置最大重试次数
        // config.setMaxRetryCnt(3);
        clusterClient = new RTIDBClusterClient(config);
        clusterClient.init();
        tableSyncClient = new TableSyncClientImpl(clusterClient);
        tableAsyncClient = new TableAsyncClientImpl(clusterClient);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// 创建kv表
public void createKVTable() {
    String name = "test1";
    NS.TableInfo.Builder builder = NS.TableInfo.newBuilder();
    builder = NS.TableInfo.newBuilder()
        .setName(name) // 设置表名
        //.setReplicaNum(3) // 设置副本数，此设置是可选的，默认为3
        //.setPartitionNum(8) // 设置分片数，此设置是可选的，默认为16
}

```

```

        //.setCompressType(NS.CompressType.kSnappy) // 设置数据压缩类型, 此设置
是可选的默认为不压缩
        //.setTtlType("kLatestTime") // 设置ttl类型, 此设置是可选的, 默认为"kAb
soluteTime"按时间过期
        .setTtl(144000); // 设置ttl
    NS.TableInfo table = builder.build();
    // 可以通过返回值判断是否创建成功
    boolean ok = nsc.createTable(table);
    clusterClient.refreshRouteTable();
}

// 创建有schema的表
public void createSchemaTable() {
    String name = "test2";
    NS.TableInfo.Builder builder = NS.TableInfo.newBuilder();
    builder = NS.TableInfo.newBuilder()
        .setName(name) // 设置表名
        //.setReplicaNum(3) // 设置副本数. 此设置是可选的, 默认为3
        //.setPartitionNum(8) // 设置分片数. 此设置是可选的, 默认为16
        //.setCompressType(NS.CompressType.kSnappy) // 设置数据压缩类型. 此设置
是可选的默认为不压缩
        //.setTtlType("kLatestTime") // 设置ttl类型. 此设置是可选的, 默认为"kAb
soluteTime"按时间过期
        .setTtl(144000); // 设置ttl. 如果ttl类型是kAbsoluteTime, 那么ttl
的单位是分钟.
    // 设置schema信息
    ColumnDesc col0 = ColumnDesc.newBuilder()
        .setName("card") // 设置字段名
        .setAddTsIdx(true) // 设置是否为index, 如果设置为true表示该字段为维度列,
查询的时候可以通过此列来查询, 否则设置为false
        .setType("string") // 设置字段类型, 支持的字段类型有[int32, uint32, int
64, uint64, float, double, string]
        //.setIsTsCol(true) // 设置是否为时间戳列
        .build();
    ColumnDesc col1 = ColumnDesc.newBuilder().setName("mcc").setAddTsIdx(true).
setType("string").build();
    ColumnDesc col2 = ColumnDesc.newBuilder().setName("money").setAddTsIdx(false).
 setType("float").build();
    // 将schema添加到builder中
    builder.addColumnDescV1(col0).addColumnDescV1(col1).addColumnDescV1(col2);
    NS.TableInfo table = builder.build();
    // 可以通过返回值判断是否创建成功
    boolean ok = nsc.createTable(table);
    clusterClient.refreshRouteTable();
}

// 创建带有组合key的表
public void createSchemaTable() {
    String name = "test3";
    NS.TableInfo.Builder builder = NS.TableInfo.newBuilder();
    builder = NS.TableInfo.newBuilder()

```

```

        .setName(name) // 设置表名
        //.setReplicaNum(3) // 设置副本数. 此设置是可选的, 默认为3
        //.setPartitionNum(8) // 设置分片数. 此设置是可选的, 默认为16
        //.setCompressType(NS.CompressType.kSnappy) // 设置数据压缩类型. 此设置
是可选的默认为不压缩
        //.setTtlType("kLatestTime") // 设置ttl类型. 此设置是可选的, 默认为"kAb
soluteTime"按时间过期
        .setTtl(144000); // 设置ttl. 如果ttl类型是kAbsoluteTime, 那么ttl
的单位是分钟.
        // 设置schema信息
        ColumnDesc col0 = ColumnDesc.newBuilder().setName("card").setAddTsIdx(false)
.setType("string").build();
        ColumnDesc col1 = ColumnDesc.newBuilder().setName("mcc").setAddTsIdx(false)
.setType("string").build();
        ColumnDesc col2 = ColumnDesc.newBuilder().setName("amt").setAddTsIdx(false)
.setType("double").build();
        ColumnDesc col3 = ColumnDesc.newBuilder().setName("ts").setAddTsIdx(false)
.setType("int64").setIsTsCol(true).build();
        ColumnKey colKey1 = ColumnKey.newBuilder().setIndexName("card_mcc").addColumn
ame("card").addColumnName("mcc").addTsName("ts").build();
        // 将schema添加到builder中
        builder.addColumnDescV1(col0).addColumnDescV1(col1).addColumnDescV1(col2).a
ddColumnDescV1(col3).addColumnKey(colKey1);
        NS.TableInfo table = builder.build();
        // 可以通过返回值判断是否创建成功
        boolean ok = nsc.createTable(table);
        clusterClient.refreshRouteTable();
    }

    // kv表put, scan, get
    public void syncKVTable() {
        String name = "test1";
        try {
            // 插入时指定的ts精确到毫秒
            long ts = System.currentTimeMillis();
            // 通过返回值可以判断是否插入成功
            boolean ret = tableSyncClient.put(name, "key1", ts, "value0");
            ret = tableSyncClient.put(name, "key1", ts + 1, "value1");
            ret = tableSyncClient.put(name, "key2", ts + 2, "value2");

            // scan数据, 查询范围需要传入st和et分别表示起始时间和结束时间, 其中起始时间大于结
束时间
            // 如果结束时间et设置为0, 返回起始时间之前的所有数据
            KvIterator it = tableSyncClient.scan(name, "key1", ts + 1, 0);
            while (it.valid()) {
                byte[] buffer = new byte[it.getValue().remaining()];
                it.getValue().get(buffer);
                String value = new String(buffer);
                System.out.println(value);
                it.next();
            }
        }
    }
}

```

```

// 可以通过limit限制返回条数
// 如果st和et都设置为0则返回最近N条记录
int limit = 1;
it = tableSyncClient.scan(name, "key1", ts + 1, 0, limit);
while (it.valid()) {
    byte[] buffer = new byte[it.getValue().remaining()];
    it.getValue().get(buffer);
    String value = new String(buffer);
    System.out.println(value);
    it.next();
}

// get数据，查询指定ts的值。如果ts设置为0，返回最新插入的一条数据
ByteString bs = tableSyncClient.get(name, "key1", ts);
// 如果没有查询到bs就是null
if (bs != null) {
    String value = new String(bs.toByteArray());
    System.out.println(value);
}
bs = tableSyncClient.get(name, "key1");
// 如果没有查询到bs就是null
if (bs != null) {
    String value = new String(bs.toByteArray());
    System.out.println(value);
}
} catch (TimeoutException e) {
    e.printStackTrace();
} catch (TabletException e) {
    e.printStackTrace();
}
}

// schema表put, scan, get
public void syncSchemaTable() {
    String name = "test2";
    try {
        // 插入数据。插入时指定的ts精确到毫秒
        long ts = System.currentTimeMillis();
        // 通过map传入需要插入的数据，key是字段名，value是该字段对应的值
        Map<String, Object> row = new HashMap<String, Object>();
        row.put("card", "card0");
        row.put("mcc", "mcc0");
        row.put("money", 1.3f);
        // 通过返回值可以判断是否插入成功
        boolean ret = tableSyncClient.put(name, ts, row);
        row.clear();
        row.put("card", "card0");
        row.put("mcc", "mcc1");
        row.put("money", 15.8f);
        ret = tableSyncClient.put(name, ts + 1, row);
    }
}

```

```

// 也可以通过object数组方式插入
// 数组顺序和创建表时schema顺序对应
// 通过RTIDBClusterClient可以获取表schema信息
// List<ColumnDesc> schema = clusterClient.getHandler(name).getSchema();

Object[] arr = new Object[]{"card1", "mcc1", 9.15f};
ret = tableSyncClient.put(name, ts + 2, arr);

// scan数据
// key是需要查询字段的值, idxName是需要查询的字段名
// 查询范围需要传入st和et分别表示起始时间和结束时间, 其中起始时间大于结束时间
// 如果结束时间et设置为0, 返回起始时间之前的所有数据
KvIterator it = tableSyncClient.scan(name, "card0", "card", ts + 1, 0);
while (it.valid()) {
    // 解码出来是一个object数组, 顺序和创建表时的schema对应
    // 通过RTIDBClusterClient可以获取表schema信息
    // List<ColumnDesc> schema = clusterClient.getHandler(name).getSchema();
    ma();
    Object[] result = it.getDecodedValue();
    System.out.println(result[0]);
    System.out.println(result[1]);
    System.out.println(result[2]);
    it.next();
}
// 可以通过limit限制返回条数
// 如果st和et都设置为0则返回最近N条记录
int limit = 1;
it = tableSyncClient.scan(name, "card0", "card", ts + 1, 0, limit);
while (it.valid()) {
    Object[] result = it.getDecodedValue();
    System.out.println(result[0]);
    System.out.println(result[1]);
    System.out.println(result[2]);
    it.next();
}

// get数据
// key是需要查询字段的值, idxName是需要查询的字段名
// 查询指定ts的值. 如果ts设置为0, 返回最新插入的一条数据

// 解码出来是一个object数组, 顺序和创建表时的schema对应
// 通过RTIDBClusterClient可以获取表schema信息
// List<ColumnDesc> schema = clusterClient.getHandler(name).getSchema();

Object[] result = tableSyncClient.getRow(name, "card0", "card", ts);
for (int idx = 0; idx < result.length; idx++) {
    System.out.println(result[idx]);
}
result = tableSyncClient.getRow(name, "card0", "card", 0);
for (int idx = 0; idx < result.length; idx++) {

```

```

        System.out.println(result[idx]);
    }
} catch (TimeoutException e) {
    e.printStackTrace();
} catch (TabletException e) {
    e.printStackTrace();
}

}

// kv表异步put, 异步scan, 异步get
public void AsyncKVTable() {
    String name = "test2";
    try {
        // 插入时指定的ts精确到毫秒
        long ts = System.currentTimeMillis();
        PutFuture pf1 = tableAsyncClient.put(name, "akey1", ts, "value4");
        PutFuture pf2 = tableAsyncClient.put(name, "akey1", ts + 1, "value5");
        PutFuture pf3 = tableAsyncClient.put(name, "akey2", ts + 2, "value6");
        // 调用get会阻塞到返回
        // 通过返回值可以判断是否插入成功
        boolean ret = pf1.get();
        ret = pf2.get();
        ret = pf3.get();

        // scan数据, 查询范围需要传入st和et分别表示起始时间和结束时间, 其中起始时间大于结束时间
        // 如果结束时间et设置为0, 返回起始时间之前的所有数据
        ScanFuture sf = tableAsyncClient.scan(name, "akey1", ts + 1, 0);
        KvIterator it = sf.get();
        while (it.valid()) {
            byte[] buffer = new byte[it.getValue().remaining()];
            it.getValue().get(buffer);
            String value = new String(buffer);
            System.out.println(value);
            it.next();
        }

        // 可以通过limit限制返回条数
        // 如果st和et都设置为0则返回最近N条记录
        int limit = 1;
        sf = tableAsyncClient.scan(name, "akey1", ts + 1, 0, limit);
        it = sf.get();
        while (it.valid()) {
            byte[] buffer = new byte[it.getValue().remaining()];
            it.getValue().get(buffer);
            String value = new String(buffer);
            System.out.println(value);
            it.next();
        }
    }
}

```

```

        // get数据, 查询指定ts的值. 如果ts设置为0, 返回最新插入的一条数据
        GetFuture gf = tableAsyncClient.get(name, "akey1", ts);
        ByteString bs = gf.get();
        // 如果没有查询到bs就是null
        if (bs != null) {
            String value = new String(bs.toByteArray());
            System.out.println(value);
        }
        gf = tableAsyncClient.get(name, "akey1");
        bs = gf.get();
        // 如果没有查询到bs就是null
        if (bs != null) {
            String value = new String(bs.toByteArray());
            System.out.println(value);
        }
    } catch (TabletException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// schema表异步put, 异步scan, 异步get
public void AsyncSchemaTable() {
    String name = "test2";
    try {
        // 插入数据. 插入时指定的ts精确到毫秒
        long ts = System.currentTimeMillis();
        // 通过map传入需要插入的数据, key是字段名, value是该字段对应的值
        Map<String, Object> row = new HashMap<String, Object>();
        row.put("card", "acard0");
        row.put("mcc", "amcc0");
        row.put("money", 1.3f);
        PutFuture pf1 = tableAsyncClient.put(name, ts, row);
        row.clear();
        row.put("card", "acard0");
        row.put("mcc", "amcc1");
        row.put("money", 15.8f);
        PutFuture pf2 = tableAsyncClient.put(name, ts + 1, row);

        // 也可以通过object数组方式插入
        // 数组顺序和创建表时schema顺序对应
        // 通过RTIDBClusterClient可以获取表schema信息
        // List<ColumnDesc> schema = clusterClient.getHandler(name).getSchema();

        Object[] arr = new Object[]{"acard1", "amcc1", 9.15f};
        PutFuture pf3 = tableAsyncClient.put(name, ts + 2, arr);
        // 调用get会阻塞到返回
        // 通过返回值可以判断是否插入成功
        boolean ret = pf1.get();
        ret = pf2.get();
    }
}

```

```

    ret = pf3.get();

    // scan数据
    // key是需要查询字段的值, idxName是需要查询的字段名
    // 查询范围需要传入st和et分别表示起始时间和结束时间, 其中起始时间大于结束时间
    // 如果结束时间et设置为0, 返回起始时间之前的所有数据
    ScanFuture sf = tableAsyncClient.scan(name, "acard0", "card", ts + 1, 0
);
    KvIterator it = sf.get();
    while (it.valid()) {
        // 解码出来是一个object数组, 顺序和创建表时的schema对应
        // 通过RTIDBClusterClient可以获取表schema信息
        // List<ColumnDesc> schema = clusterClient.getHandler(name).getSchema();
        Object[] result = it.getDecodedValue();
        System.out.println(result[0]);
        System.out.println(result[1]);
        System.out.println(result[2]);
        it.next();
    }
    // 可以通过limit限制返回条数
    // 如果st和et都设置为0则返回最近N条记录
    sf = tableAsyncClient.scan(name, "acard0", "card", ts + 1, 0, limit);
    it = sf.get();
    while (it.valid()) {
        Object[] result = it.getDecodedValue();
        System.out.println(result[0]);
        System.out.println(result[1]);
        System.out.println(result[2]);
        it.next();
    }

    // get数据
    // key是需要查询字段的值, idxName是需要查询的字段名
    // 查询指定ts的值. 如果ts设置为0, 返回最新插入的一条数据

    // 解码出来是一个object数组, 顺序和创建表时的schema对应
    // 通过RTIDBClusterClient可以获取表schema信息
    // List<ColumnDesc> schema = clusterClient.getHandler(name).getSchema();

    GetFuture gf = tableAsyncClient.get(name, "acard0", "card", ts);
    Object[] result = gf.getRow();
    for (int idx = 0; idx < result.length; idx++) {
        System.out.println(result[idx]);
    }
    gf = tableAsyncClient.get(name, "acard0", "card", 0);
    result = gf.getRow();
    for (int idx = 0; idx < result.length; idx++) {
        System.out.println(result[idx]);
    }
} catch (TabletException e) {
}

```

```

        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// 组合key和指定ts列的put, scan, get
try {
    Map<String, Object> data = new HashMap<String, Object>();
    data.put("card", "card0");
    data.put("mcc", "mcc0");
    data.put("amt", 1.5);
    data.put("ts", 1234L);
    tableSyncClient.put(name, data);
    data.clear();
    data.put("card", "card0");
    data.put("mcc", "mcc1");
    data.put("amt", 1.6);
    data.put("ts", 1235L);
    tableSyncClient.put(name, data);
    Map<String, Object> scan_key = new HashMap<String, Object>();
    scan_key.put("card", "card0");
    scan_key.put("mcc", "mcc0");
    KvIterator it = tableSyncClient.scan(name, scan_key, "card_mcc", 1235L,
0L, "ts", 0);
    Object[] row = it.getDecodedValue();
    scan_key.put("mcc", "mcc1");
    it = tableSyncClient.scan(name, scan_key, "card_mcc", 1235L, 0L, "ts", 0
);
    scan_key.put("mcc", "mcc2");
    it = tableSyncClient.scan(name, scan_key, "card_mcc", 1235L, 0L, "ts", 0
);
    row = tableSyncClient.getRow(name, new Object[] {"card0", "mcc0"}, "car
d_mcc", 1234, "ts", null);

    Map<String, Object> key_map = new HashMap<String, Object>();
    key_map.put("card", "card0");
    key_map.put("mcc", "mcc1");
    row = tableSyncClient.getRow(name, key_map, "card_mcc", 1235, "ts", null
);

    data.clear();
    data.put("card", "card0");
    data.put("mcc", "mcc1");
    data.put("amt", 1.6);
    data.put("ts", 1240L);
    tableSyncClient.put(name, data);

    data.clear();
    data.put("card", "card0");
}

```

```

        data.put("mcc", "mcc1");
        data.put("amt", 1.7);
        data.put("ts", 12451);
        PutFuture pf = tableAsyncClient.put(name, data);
        pf.get();

        ScanFuture sf = tableAsyncClient.scan(name, key_map, "card_mcc", 1245, 0
        , "ts", 0);
        it = sf.get();
        row = it.getDecodedValue();

        GetFuture gf = tableAsyncClient.get(name, key_map, "card_mcc", 1235, "t
        s", null);
        row = gf.getRow();

        data.clear();
        data.put("card", "card0");
        data.put("mcc", "");
        data.put("amt", 1.8);
        data.put("ts", 12501);
        tableSyncClient.put(name, data);
        row = tableSyncClient.getRow(name, new Object[] {"card0", ""}, "card_mc
        c", 0, "ts", null);
    } catch (Exception e) {
        e.printStackTrace();
    }

// 遍历全表
// 建议将removeDuplicateByTime设为true，避免同一个key下相同ts比较多时引起client死循环
// 建议将重试次数MaxRetryCnt往大设下，默认值为1.
public void Traverse() {
    String name = "test2";
    try {
        for (int i = 0; i < 1000; i++) {
            rowMap = new HashMap<String, Object>();
            rowMap.put("card", "card" + (i + 9529));
            rowMap.put("mcc", "mcc" + i);
            rowMap.put("amt", 9.2d);
            boolean ok = tableSyncClient.put(name, i + 9529, rowMap);
        }
        KvIterator it = tableSyncClient.traverse(name, "card");
        while (it.valid()) {
            // 一次迭代只能调用一次getDecodedValue
            Object[] row = it.getDecodedValue();
            System.out.println(row[0] + " " + row[1] + " " + row[2]);
            it.next();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```



## 单机模式

```
package example;

import com._4paradigm.rtidb.client.GetFuture;
import com._4paradigm.rtidb.client.PutFuture;
import com._4paradigm.rtidb.client.ScanFuture;
import com._4paradigm.rtidb.client.KvIterator;
import com._4paradigm.rtidb.client.TabletException;
import com._4paradigm.rtidb.client.ha.RTIDBClientConfig;
import com._4paradigm.rtidb.client.ha.impl.RTIDBSingleNodeClient;
import com._4paradigm.rtidb.client.impl.TableAsyncClientImpl;
import com._4paradigm.rtidb.client.impl.TableSyncClientImpl;
import com._4paradigm.rtidb.client.impl.TabletClientImpl;
import com._4paradigm.rtidb.client.schema.ColumnDesc;
import com._4paradigm.rtidb.client.schema.ColumnType;
import com._4paradigm.rtidb.tablet.Tablet;
import com.google.protobuf.ByteString;
import io.brpc.client.EndPoint;

import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SingleModeExample {
    private static EndPoint endpoint = new EndPoint("172.27.128.31:8541"); // 配置节点endpoint
    private static RTIDBClientConfig config = new RTIDBClientConfig();
    private static RTIDBSingleNodeClient snc = new RTIDBSingleNodeClient(config, endpoint);
    private static TabletClientImpl tabletClient = null;
    // 同步client
    private static TableSyncClientImpl tableClient = null;
    // 异步client
    private static TableAsyncClientImpl tableAsyncClient = null;
    static {
        try {
            snc.init();
        } catch (Exception e) {
            e.printStackTrace();
        }
        tableClient = new TableSyncClientImpl(snc);
        tableAsyncClient = new TableAsyncClientImpl(snc);
        tabletClient = new TabletClientImpl(snc);
    }
}
```

```

public void createKVTable() {
    int tid = 10;
    int pid = 0; // pid设置为0, 不要改成其他值
    // tid和pid唯一确定一张表, 创建表时不能和已有表重复
    int segCnt = 8; // 一般指定为8
    // 通过返回值可以查看是否创建成功
    // 创建absolute表, 过期时间设置为144000分钟. 如果过期时间设置为0表示不过期
    boolean ret = tabletClient.createTable("test1", tid, 0, 144000, segCnt);
    // 创建latest表, 保留最近1条记录
    // latest表不能scan
    ret = tabletClient.createTable("test2", tid + 1, pid, 1, Tablet.TTLType.kLatestTime, segCnt);
}

public void createSchemaTable() {
    int tid = 20;
    int pid = 0; // pid设置为0, 不要改成其他值
    // tid和pid唯一确定一张表, 创建表时不能和已有表重复
    int segCnt = 8; // 一般指定为8

    // 设置表schema
    List<ColumnDesc> schema = new ArrayList<ColumnDesc>();
    ColumnDesc desc1 = new ColumnDesc();
    desc1.setName("card"); // 设置字段名
    desc1.setAddTsIndex(true); // 设置是否为索引列, 如果设置为true可以通过此列来查询数据

    desc1.setType(ColumnType.kString); // 设置类型
    schema.add(desc1);
    ColumnDesc desc2 = new ColumnDesc();
    desc2.setName("mcc");
    desc2.setAddTsIndex(true);
    desc2.setType(ColumnType.kString);
    schema.add(desc2);
    ColumnDesc desc3 = new ColumnDesc();
    desc3.setName("money");
    desc3.setAddTsIndex(false);
    desc3.setType(ColumnType.kFloat);
    schema.add(desc3);
    // 通过返回值可以查看是否创建成功
    // 默认创建的是absolute表, 如果过期时间设置为0表示不过期
    boolean ret = tabletClient.createTable("test3", tid, pid, 144000, segCnt, schema);
    // 创建latest表, 保留最近10条记录
    // latest表不能scan
    ret = tabletClient.createTable("test4", tid + 1, pid, 10, Tablet.TTLType.kLatestTime, segCnt, schema);
}

// kv表put, scan, get
public void syncKVTable() {
}

```

```

int tid = 10;
int pid = 0;
long ts = System.currentTimeMillis();
try {
    // 可以通过返回值来判断是否put成功
    boolean ret = tableClient.put(tid, pid, "key1", ts, "value0");
    ret = tableClient.put(tid, pid, "key1", ts + 1, "value1");
    ret = tableClient.put(tid, pid, "key2", ts + 2, "value3");

    // scan数据
    // 需要传入st和et, 分别表示起始时间和结束时间, 其中起始时间大于结束时间. 返回起始时间和结束时间之间的数据
    // 如果结束时间et设置为0, 则返回从起始时间以来所有没有过期的数据
    KvIterator it = tableClient.scan(tid, pid, "key1", ts + 1, 0);
    while (it.valid()) {
        byte[] buffer = new byte[it.getValue().remaining()];
        it.getValue().get(buffer);
        String value = new String(buffer);
        System.out.println("scan-" + value);
        it.next();
    }

    // 可以通过limit限制返回条数
    // 如果st和et都设置为0则返回最近N条记录
    it = tableClient.scan(tid, pid, "key1", ts + 1, 0, 1);
    while (it.valid()) {
        byte[] buffer = new byte[it.getValue().remaining()];
        it.getValue().get(buffer);
        String value = new String(buffer);
        System.out.println("scan1-" + value);
        it.next();
    }

    // get数据
    ByteString buffer = tableClient.get(tid, 0, "key1", ts);
    // 如果没有查到数据返回null
    if (buffer != null) {
        System.out.println("get" + buffer.toString(Charset.forName("utf-8")));
    }
    // ts设置为0或者不设置返回最新插入的数据
    buffer = tableClient.get(tid, 0, "key1", 0);
    // buffer = tableClient.get(tid, 0, "key1");
    if (buffer != null) {
        System.out.println("get1" + buffer.toString(Charset.forName("utf-8")));
    }
} catch (TabletException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}

```

```

    }

}

// schema表put, scan, get
public void syncSchemaTable() {
    int tid = 20;
    int pid = 0;
    long ts = System.currentTimeMillis();
    try {
        // 通过map传入需要插入的数据, key是字段名, value是该字段对应的值
        Map<String, Object> row = new HashMap<String, Object>();
        row.put("card", "card0");
        row.put("mcc", "mcc0");
        row.put("money", 1.3f);
        // 可以通过返回值来判断是否put成功
        boolean ret = tableClient.put(tid, pid, ts, row);
        row.clear();
        row.put("card", "card0");
        row.put("mcc", "mcc1");
        row.put("money", 15.8f);
        ret = tableClient.put(tid, pid, ts + 1, row);
        // 也可以通过object数组方式插入
        // 数组顺序和创建表时schema顺序对应
        // 通过RTIDBSingleNodeClient可以获取表schema信息
        // List<ColumnDesc> schema = snc.getHandler(tid).getSchema();
        Object[] arr = new Object[]{"card1", "mcc1", 9.15f};
        ret = tableClient.put(tid, pid, ts + 2, arr);

        // scan数据
        // 需要传入st和et, 分别表示起始时间和结束时间, 其中起始时间大于结束时间. 返回起始时间和结束时间之间的数据
        // 如果结束时间et设置为0, 则返回从起始时间以来所有没有过期的数据
        KvIterator it = tableClient.scan(tid, pid, "card0", "card", ts + 1, 0);
        while (it.valid()) {
            // 解码出来是一个object数组, 顺序和创建表时的schema对应
            // 通过RTIDBSingleNodeClient可以获取表schema信息
            // List<ColumnDesc> schema = snc.getHandler(tid).getSchema();
            Object[] result = it.getDecodedValue();
            System.out.println("scan-" + result[0]);
            System.out.println("scan-" + result[1]);
            System.out.println("scan-" + result[2]);
            it.next();
        }

        // 可以通过limit限制返回的记录条数
        // 如果st和et都设置为0则返回最近N条记录
        it = tableClient.scan(tid, pid, "card0", "card", ts + 1, 0, 1);
        while (it.valid()) {
            Object[] result = it.getDecodedValue();
            System.out.println("scan1-" + result[0]);
            System.out.println("scan1-" + result[1]);
        }
    }
}

```

```

        System.out.println("scan1-" + result[2]);
        it.next();
    }

    // get数据
    // key是需要查询字段的值, idxName是需要查询的字段名
    // 查询指定ts的值. 如果不设置ts或者ts设置为0, 返回最新插入的一条数据

    // 解码出来是一个object数组, 顺序和创建表时的schema对应
    // 通过RTIDBSingleNodeClient可以获取表schema信息
    // List<ColumnDesc> schema = clusterClient.getHandler(tid).getSchema();
    Object[] result = tableClient.getRow(tid, pid, "card0", "card", ts);
    for (int idx = 0; idx < result.length; idx++) {
        System.out.println("get-" + result[idx]);
    }
    result = tableClient.getRow(tid, pid, "card0", "card", 0);
    // result = tableClient.getRow(tid, pid, "card0", "card");
    for (int idx = 0; idx < result.length; idx++) {
        System.out.println("get1-" + result[idx]);
    }
} catch (TabletException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
}

// kv表异步put, 异步scan, 异步get
public void aSyncKVTable() {
    int tid = 10;
    int pid = 0;
    long ts = System.currentTimeMillis();
    try {
        PutFuture pf1 = tableAsyncClient.put(tid, pid, "akey1", ts, "value0");
        PutFuture pf2 = tableAsyncClient.put(tid, pid, "akey1", ts + 1, "value1");
    };
    PutFuture pf3 = tableAsyncClient.put(tid, pid, "akey2", ts + 2, "value3");
};

    // 调用get会阻塞到返回
    // 通过返回值可以判断是否插入成功
    boolean ret = pf1.get();
    ret = pf2.get();
    ret = pf3.get();

    // scan数据, 查询范围需要传入st和et分别表示起始时间和结束时间, 其中起始时间大于结束时间
    // 如果结束时间et设置为0, 返回起始时间之前的所有数据
    ScanFuture sf = tableAsyncClient.scan(tid, pid, "akey1", ts + 1, 0);
    KvIterator it = sf.get();
    while (it.valid()) {
        byte[] buffer = new byte[6];

```

```

        it.getValue().get(buffer);
        String value = new String(buffer);
        System.out.println(value);
        it.next();
    }
    // 可以通过limit限制返回的记录条数
    // 如果st和et都设置为0则返回最近N条记录
    int limit = 1;
    sf = tableAsyncClient.scan(tid, pid, "akey1", ts + 1, 0, limit);
    it = sf.get();
    while (it.valid()) {
        byte[] buffer = new byte[6];
        it.getValue().get(buffer);
        String value = new String(buffer);
        System.out.println(value);
        it.next();
    }

    // get数据
    // 查询指定ts的值. 如果不设置ts或者设置为0, 返回最新插入的一条数据
    GetFuture gf = tableAsyncClient.get(tid, pid, "akey1", ts);
    ByteString bs = gf.get();
    // 如果没有查询到bs就是null
    if (bs != null) {
        String value = new String(bs.toByteArray());
        System.out.println(value);
    }
    gf = tableAsyncClient.get(tid, pid, "akey1");
    //如果不设置ts和设置为0是一样的
    // gf = tableAsyncClient.get(tid, pid, "akey1", 0);

    bs = gf.get();
    // 如果没有查询到bs就是null
    if (bs != null) {
        String value = new String(bs.toByteArray());
        System.out.println(value);
    }
} catch (TabletException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}

}

// schema表异步put, 异步scan, 异步get
public void aSyncSchemaTable() {
    int tid = 20;
    int pid = 0;
    long ts = System.currentTimeMillis();
    try {

```

```

    // 通过map传入需要插入的数据, key是字段名, value是该字段对应的值
    Map<String, Object> row = new HashMap<String, Object>();
    row.put("card", "acard0");
    row.put("mcc", "amcc0");
    row.put("money", 1.3f);
    PutFuture pf1 = tableAsyncClient.put(tid, pid, ts, row);
    row.clear();
    row.put("card", "acard0");
    row.put("mcc", "amcc1");
    row.put("money", 15.8f);
    PutFuture pf2 = tableAsyncClient.put(tid, pid, ts + 1, row);

    // 也可以通过object数组方式插入
    // 数组顺序和创建表时schema顺序对应
    // 通过RTIDBSingleNodeClient可以获取表schema信息
    // List<ColumnDesc> schema = snc.getHandler(tid).getSchema();
    Object[] arr = new Object[]{"acard1", "amcc1", 9.15f};
    PutFuture pf3 = tableAsyncClient.put(tid, pid, ts + 2, arr);
    // 调用get会阻塞到返回
    // 通过返回值可以判断是否插入成功
    boolean ret = pf1.get();
    ret = pf2.get();
    ret = pf3.get();

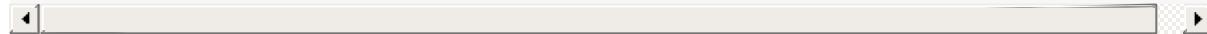
    // scan数据
    // key是需要查询字段的值, idxName是需要查询的字段名
    // 查询范围需要传入st和et分别表示起始时间和结束时间, 其中起始时间大于结束时间
    // 如果结束时间et设置为0, 返回起始时间之前的所有数据
    ScanFuture sf = tableAsyncClient.scan(tid, pid, "acard0", "card", ts + 1
    , 0);
    KvIterator it = sf.get();
    while (it.valid()) {
        // 解码出来是一个object数组, 顺序和创建表时的schema对应
        // 通过RTIDBSingleNodeClient可以获取表schema信息
        // List<ColumnDesc> schema = snc.getHandler(tid).getSchema();
        Object[] result = it.getDecodedValue();
        System.out.println(result[0]);
        System.out.println(result[1]);
        System.out.println(result[2]);
        it.next();
    }
    // 可以通过limit限制返回的记录条数
    // 如果st和et都设置为0则返回最近N条记录
    int limit = 1;
    sf = tableAsyncClient.scan(tid, pid, "acard0", "card", ts + 1, 0, limit
);
    it = sf.get();
    while (it.valid()) {
        Object[] result = it.getDecodedValue();
        System.out.println(result[0]);
        System.out.println(result[1]);
    }
}

```

```
        System.out.println(result[2]);
        it.next();
    }

    // get数据
    // key是需要查询字段的值, idxName是需要查询的字段名
    // 查询指定ts的值. 如果不设置ts或者设置为0, 返回最新插入的一条数据

    // 解码出来是一个object数组, 顺序和创建表时的schema对应
    // 通过RTIDBSingleNodeClient可以获取表schema信息
    // List<ColumnDesc> schema = snc.getHandler(tid).getSchema();
    GetFuture gf = tableAsyncClient.get(tid, pid, "acard0", "card", ts);
    Object[] result = gf.getRow();
    for (int idx = 0; idx < result.length; idx++) {
        System.out.println(result[idx]);
    }
    gf = tableAsyncClient.get(tid, pid, "acard0", "card");
    //如果不设置ts和设置为0一样的
    // gf = tableAsyncClient.get(tid, pid, "acard0", "card", 0);
    result = gf.getRow();
    for (int idx = 0; idx < result.length; idx++) {
        System.out.println(result[idx]);
    }
} catch (TabletException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
}
```



## 返回码说明

code	message	说明
0	ok	成功
100	table is not exist	表不存在
101	table already exists	表已经存在
102	table is leader	表是leader
103	table is follower	表是follower
104	table is loading	表正在loading
105	table status is not kNormal	表不是kNormal状态
106	table status is kMakingSnapshot	表状态是kMakingSnapshot
107	table status is not kSnapshotPaused	表状态不是kSnapshotPaused
108	idx name not found	找不到索引列名
109	key not found	找不到key
110	replicator is not exist	replicator不存在
111	snapshot is not exist	snapshot不存在
112	ttl type mismatch	ttl类型不匹配
114	ts must be greater than zero	插入时ts必须大于0
115	invalid dimension parameter	维度参数无效
116	put failed	put失败
117	st less than et	起始时间小于结束时间
118	reache the scan max bytes size	scan的大小超过了最大值
119	replica endpoint already exists	副本节点已存在
120	fail to add replica endpoint	添加副本失败
121	replicator role is not leader	replicator不是leader
122	fail to append entries to replicator	添加entries失败
123	file receiver init failed	file receiver初始化失败
124	cannot find receiver	找不到receiver
125	block_id mismatch	block_id 不匹配
126	receive data error	数据接收错误
127	write data failed	数据写入错误
128	snapshot is sending	正在发送snapshot
129	table meta is illegal	table meta不合法

130	table db path is not exist	表的db路径不存在
131	create table failed	表创建失败
132	ttl is greater than conf value	ttl超过了最大值
133	cannot update ttl between zero and nonzero	ttl不能在0和非0值之间转换
134	no follower	没有follower
135	invalid concurrency	无效的concurrency
136	delete failed	删除失败
137	ts name not found	找不到时间戳列名
300	nameserver is not leader	当前nameserver不是leader
301	auto_failover is enabled	auto_failover是开启的
302	endpoint is not exist	endpoint不存在
303	tablet is not healthy	tablet已经下线
304	set zk failed	写zk失败
305	create op failed	创建op失败
306	add op data failed	添加op失败
307	invalid parameter	参数错误
308	pid is not exist	分片id不存在
309	leader is alive	leader是健康的
310	no alive follower	没有alive的follower
311	partition is alive	分片是健康的
312	op status is not kDoing or kInitiated	任务status不是kDoing或者kInitiated
313	drop table error	删除表错误
314	set partition info failed	设置partition结构失败
315	convert column desc failed	转换schema失败
316	create table failed on tablet	在tablet上创建表失败
317	pid already exists	分片已经存在
318	src_endpoint is not exist or not healthy	源节点不存在或者已经下线
319	des_endpoint is not exist or not healthy	目的节点不存在或者已经下线
320	migrate failed	副本迁移失败
321	no pid has update	分片信息没更新
322	fail to update ttl from tablet	tablet上ttl更新失败



# 导入工具

- 使用说明
- 配置文件
- 性能

## 使用说明

### 功能

实现将文件中数据导入rtidb的功能，支持parquet、csv和orc三种存储格式文件的导入，可以导入单一指定文件，也可以导入指定目录下的多个文件，如把多个parquet文件放在同一目录下进行操作。

程序运行时会根据文件后缀名自动进入相应解析程序。对于parquet文件和orc文件，程序可以读取文件得到schema，继而将数据插入rtidb；对于csv文件，用户需要在csv\_schema.conf配置文件中定义schema。

### 执行步骤

- 1、在<http://pkg.4paradigm.com/rtidb/tools/dataimporter-util.jar> 下载dataimporter.jar； config.properties和csv\_schema.conf自己创建，可以复制本文档中模版进行适当修改。  
注意：配置文件和jar包需要放在同一目录下
- 2、准备本地的数据文件（xxx.parquet、xxx.orc或者xxx.csv文件）
- 3、配置config.properties，指定数据文件的绝对路径（若导入一个目录下的多个文件，配置目录的绝对路径且目录文件中只能包含parquet、csv或者orc一种格式的文件，不能包含其它类型文件）和表名；如果导入csv文件，配置csv\_schema.conf文件，自定义schema。
- 4、运行本地jar包，执行命令：java -cp jar包名 com.\_4paradigm.dataimporter.Main  
如 java -cp dataimporter.jar com.\_4paradigm.dataimporter.Main

### 支持数据类型

#### 1、parquet

支持8种基本类型：int32,int64,int96,float,double,boolean,binary,fixed\_len\_byte\_array

parquet类型	对应rtidb类型
int32	int32
int64	int64
int96	timestamp
float	float
double	double
boolean	bool
binary	string

fixed_len_byte_array	string
----------------------	--------

## 2、csv

支持9种基本类型： string,int16,int32,int64,double,float,timestamp,date,bool

csv类型和rtidb类型一一对应

## 3、orc

支持struct中包含14种基本类型：

binary,boolean,byte,date,double,float,int,long,short,string,timestamp,char,varchar,decimal

orc类型	对应rtidb类型
binary	string
boolean	bool
byte	int16
date	date
double	double
float	float
int	int32
long	int64
short	int16
string	string
timestamp	timestamp
char	string
varchar	string
decimal	double

## 配置文件

### config.properties

```
#文件或者目录的绝对路径
filePath=/Users/innerpeace/Desktop/test.orc
#表名
tableName=orcTest
#用户配置rtidb中是否已经提前创建成功表，若没有，程序会先创建表，后导入数据
tableExist=false
#指定索引列的列名，多个之间分号分割
index=mdcardno
#指定文件中时间戳列index
timeStampIndex=0
#指定parquet文件和orc文件的列下标，若没有配置，导入全部列
inputColumnIndex=0,1,2,4,5,6
```

```

#csv文件schema配置文件的相对路径, 即文件名
csv.schemaPath=csv_schema.conf

#csv文件的分隔符, 通常都为,
csv.separator=,
#csv文件的编码格式
csv.encodingFormat=UTF-8

#csv文件是否有表头
csv.hasHeader=true

#配置zk地址, 和集群启动配置中的zk_cluster保持一致
zkEndpoints=172.27.128.37:7181,172.27.128.37:7182,172.27.128.37:7183

#配置集群的zk根路径, 和集群启动配置中的zk_root_path保持一致
zkRootPath=/rtidb_cluster

#副本数
replicaNum=1

#分片数
partitionNum=1

#kAbsoluteTime或者kLatestTime
ttlType=kAbsoluteTime

# 0对应kNoCompress ; 1对应kSnappy
compressType=0

#生存时间, 设置ttl. 如果ttl类型是kAbsoluteTime, 那么ttl的单位是分钟, ttl为0时, 表示不过期;
#如果ttl类型是kLatestTime, ttl表示保留的最大记录条数
ttl=0

#线程池相关, 一般不需要重新配置
#线程池的核心线程数和最大线程数, 执行put操作的最大线程数为maximumPoolSize+1
maximumPoolSize=6
#阻塞队列大小
blockingQueueSize=10000
#成功put一定数量数据输出一条日志
log.interval=1000

```

## csv\_schema.conf

### 方式一

```

#csv schema的字段名, 字段类型和字段下标。
columnName=col_1;columnType=int32;columnIndex=0
columnName=col_2;columnType=int64;columnIndex=1
columnName=col_3;columnType=string;columnIndex=2
columnName=col_4;columnType=float;columnIndex=3
columnName=col_5;columnType=double;columnIndex=4
columnName=col_6;columnType=boolean;columnIndex=5

```

### 方式二

```

#如果配置文件中没有columnIndex这一列, 则导入文件中全部列
columnName=col_1;columnType=int32

```

```
columnName=col_2;columnType=int64
columnName=col_3;columnType=string
columnName=col_4;columnType=float
columnName=col_5;columnType=double
columnName=col_6;columnType=boolean
```

## 性能

### 测试环境

OS	<b>Linux m7-pce-dev01 3.10.0-862.el7.x86_64 #1 SMP Fri Apr 20 16:44:24 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux</b>
CPU	processor : 40    Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
MEM	503G

### 测试过程

```
数据量: 1000w
put线程数: 7
rtidb副本数: 1
rtidb分片数: 1
rtidb表类型: absolute表
```

rtidb表schema如下:

#	name	type	index
0	mdcardno	string	yes
1	CARDSTAT	string	no
2	CARDKIND	string	no
3	SYNFLAG	string	no
4	GOLDFLAG	string	no
5	OPENDATE	string	no
6	CDSQUOTA	string	no
7	CDTQUOTA	string	no
8	ACTCHANEL	string	no
9	ACTDATE	string	no
10	SALECOD	string	no

其中mdcardno长度为8, 其它字段长度均为2。

在单台机器环境下导入10,000,000数据用时3min5s, 平均qps达到50,000以上。



# RTIDB集群运维

- 宕机与恢复
- 扩容
- 升级

## 宕机与恢复

RTIDB高可用可配置为自动模式和手动模式。在自动模式下如果节点宕机和恢复时系统会自动做故障转移和数据恢复, 否则需要用手动处理

通过修改auto\_failover配置可以切换模式, 默认是开启自动模式。通过以下方式可以获取配置状态和修改配置

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> confget
  key          value
-----
  auto_failover    true
> confset auto_failover false
set auto_failover ok
>confget
  key          value
-----
  auto_failover    false
```

**auto\_failover**开启的话如果一个节点下线了, **showtable**的**is\_alive**状态就会变成no

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> showtable
  name      tid  pid endpoint           role      ttl      is_alive  compress_type
  offset    record_cnt  memused
-----
  flow      4    0   172.27.128.32:8541  leader    0min      no       kNoCompress
  0        0            0.000
  flow      4    0   172.27.128.33:8541  follower  0min      yes      kNoCompress
  0        0            0.000
  flow      4    0   172.27.128.31:8541  follower  0min      yes      kNoCompress
  0        0            0.000
  flow      4    1   172.27.128.33:8541  leader    0min      yes      kNoCompress
  0        0            0.000
  flow      4    1   172.27.128.31:8541  follower  0min      yes      kNoCompress
  0        0            0.000
  flow      4    1   172.27.128.32:8541  follower  0min      no       kNoCompress
```

0	0	0.000
---	---	-------

在手动模式下节点下线和恢复时需要手动操作。有两个命令offlineendpoint和recoverendpoint

命令格式: offlineendpoint endpoint

endpoint是发生故障节点的endpoint。该命令会对该节点下所有分片执行如下操作:

- 如果是主, 执行重新选主
- 如果是从, 找到主节点然后从主节点中删除当前endpoint副本

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> offlineendpoint 172.27.128.32:8541
offline endpoint ok
>showtable
  name      tid  pid  endpoint          role      ttl      is_alive  compress_type
  offset    record_cnt  memused
-----
  flow      4    0   172.27.128.32:8541  leader    0min      no       kNoCompress
  0        0           0.000
  flow      4    0   172.27.128.33:8541  follower  0min      yes      kNoCompress
  0        0           0.000
  flow      4    0   172.27.128.31:8541  leader    0min      yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.33:8541  leader    0min      yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.31:8541  follower  0min      yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.32:8541  follower  0min      no       kNoCompress
  0        0           0.000
```

执行完offlineendpoint后每一个分片都会分配新的leader。如果某个分片执行失败可以单独对这个分片执行changeleader, 命令格式为: changeleader table\_name pid

如果节点已经恢复, 就可以执行recoverendpoint来恢复数据

命令格式: recoverendpoint endpoint

endpoint是状态已经变为healthy节点的endpoint

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> showtablet
  endpoint          state      age
-----
  172.27.128.31:8541  kTabletHealthy  3h
  172.27.128.32:8541  kTabletHealthy  7m
  172.27.128.33:8541  kTabletHealthy  3h
```

```
> recover endpoint 172.27.128.32:8541
recover endpoint ok
> showopstatus
  op_id  op_type          name  pid  status  start_time      execute_time  end_
time      cur_task
-----
54      kUpdateTableAliveOP  flow  0    kDone   20180824195838  2s           2018
0824195840 -
55      kChangeLeaderOP    flow  0    kDone   20180824200135  0s           2018
0824200135 -
56      kOfflineReplicaOP  flow  1    kDone   20180824200135  1s           2018
0824200136 -
57      kUpdateTableAliveOP flow  0    kDone   20180824200212  0s           2018
0824200212 -
58      kRecoverTableOP    flow  0    kDone   20180824200623  1s           2018
0824200624 -
59      kRecoverTableOP    flow  1    kDone   20180824200623  1s           2018
0824200624 -
60      kReAddReplicaOP   flow  0    kDoing  20180824200624  4s           -
kLoadTable
61      kReAddReplicaOP   flow  1    kDoing  20180824200624  4s           -
kLoadTable
```

执行showopstatus查看任务运行进度，如果status是doing状态说明任务还没有运行完

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> showopstatus
  op_id  op_type          name  pid  status  start_time      execute_time  end_
time      cur_task
-----
54      kUpdateTableAliveOP  flow  0    kDone   20180824195838  2s           2018
0824195840 -
55      kChangeLeaderOP    flow  0    kDone   20180824200135  0s           2018
0824200135 -
56      kOfflineReplicaOP  flow  1    kDone   20180824200135  1s           2018
0824200136 -
57      kUpdateTableAliveOP flow  0    kDone   20180824200212  0s           2018
0824200212 -
58      kRecoverTableOP    flow  0    kDone   20180824200623  1s           2018
0824200624 -
59      kRecoverTableOP    flow  1    kDone   20180824200623  1s           2018
0824200624 -
60      kReAddReplicaOP   flow  0    kDone   20180824200624  9s           2018
0824200633 -
61      kReAddReplicaOP   flow  1    kDone   20180824200624  9s           2018
0824200633 -
> showtable
```

```
>showtable
  name    tid   pid   endpoint          role      ttl      is_alive  compress_type
  offset   record_cnt   memused
  -----
  flow     4     0   172.27.128.32:8541  follower  0min      yes      kNoCompress
  0         0           0.000
  flow     4     0   172.27.128.33:8541  follower  0min      yes      kNoCompress
  0         0           0.000
  flow     4     0   172.27.128.31:8541  leader    0min      yes      kNoCompress
  0         0           0.000
  flow     4     1   172.27.128.33:8541  leader    0min      yes      kNoCompress
  0         0           0.000
  flow     4     1   172.27.128.31:8541  follower  0min      yes      kNoCompress
  0         0           0.000
  flow     4     1   172.27.128.32:8541  follower  0min      yes      kNoCompress
  0         0           0.000
```

showtable如果都变成yes表示已经恢复成功。如果某些分片恢复失败可以单独执行recoverable, 命令格式为: recoverable table\_name pid endpoint

**注: 执行recoverendpoint前必须执行过一次offlineendpoint**

## 扩容

随着业务的发展, 当前集群的拓扑不能满足要求就需要动态的扩容。扩容就是把一部分分片从现有tablet节点迁移到新的tablet节点上从而减小内存占用

### 1 启动一个新的tablet节点

启动方式参见前面章节部署RTIDB部分

启动后查看新增节点是否加入集群。如果执行showtablet命令列出了新节点endpoint说明已经加入到集群中

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> showtablet
  endpoint          state        age
  -----
  172.27.128.31:8541  kTabletHealthy  15d
  172.27.128.32:8541  kTabletHealthy  15d
  172.27.128.33:8541  kTabletHealthy  15d
  172.27.128.37:8541  kTabletHealthy  1min
```

### 2 迁移副本

副本迁移用到的命令是migrate。命令格式: migrate src\_endpoint table\_name partition des\_endpoint, 详细用法参考命令使用部分

迁移的时候只能迁移从分片, 不能迁移主分片

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> showtable
  name      tid  pid  endpoint          role    ttl     is_alive  compress_type
  offset    record_cnt  memused
-----
  flow      4    0   172.27.128.32:8541  leader   0min    yes      kNoCompress
  0        0           0.000
  flow      4    0   172.27.128.33:8541  follower 0min    yes      kNoCompress
  0        0           0.000
  flow      4    0   172.27.128.31:8541  follower 0min    yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.33:8541  leader   0min    yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.31:8541  follower 0min    yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.32:8541  follower 0min    yes      kNoCompress
  0        0           0.000
> migrate 172.27.128.33:8541 flow 0 172.27.128.37:8541
> showopstatus flow
  op_id  op_type      name  pid  status  start_time       execute_time  end_time
  cur_task
-----
  51      kMigrateOP  flow  0    kDone   20180824163316  12s            2018082416332
8      -
> showtable
  name      tid  pid  endpoint          role    ttl     is_alive  compress_type
  offset    record_cnt  memused
-----
  flow      4    0   172.27.128.32:8541  leader   0min    yes      kNoCompress
  0        0           0.000
  flow      4    0   172.27.128.37:8541  follower 0min    yes      kNoCompress
  0        0           0.000
  flow      4    0   172.27.128.31:8541  follower 0min    yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.33:8541  leader   0min    yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.31:8541  follower 0min    yes      kNoCompress
  0        0           0.000
  flow      4    1   172.27.128.32:8541  follower 0min    yes      kNoCompress
  0        0           0.000
```

## 升级

## 1 升级nameserver

- 备份旧版本bin和conf目录
- 下载新版本bin和conf
- 对比配置文件diff并修改必要的配置，如endpoint、zk\_cluster等
- 执行sh bin/start\_ns.sh stop (如果启动时版本小于1.3.11, 就直接kill进程)
- 执行sh bin/start\_ns.sh start 启动nameserver
- 对剩余nameserver重复以上步骤

## 2 升级tablet

- 备份旧版本bin和conf目录
- 下载新版本bin和conf
- 对比配置文件diff并修改必要的配置，如endpoint、zk\_cluster等
- 执行sh bin/start.sh stop (如果启动时版本小于1.3.11, 就直接kill进程)
- 执行sh bin/start.sh start启动tablet
- 如果auto\_failover关闭时得连上ns client执行如下操作恢复数据。其中命令后面的**endpoint**为重启节点的**endpoint**
  - offline endpoint endpoint
  - recover endpoint endpoint

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
> offline endpoint 172.27.128.32:8541
offline endpoint ok
> recover endpoint 172.27.128.32:8541
recover endpoint ok
```

- 参考宕机和恢复中 showtable和showopstatus命令查看恢复进度
- 用tablet client连接到其他tablet上查看数据恢复进度
  - 执行gettablestatus查看leader的tid和pid
  - 执行getfollower查看同步进度
  - 当重启节点的**offset**和**leader**的**offset**差别小于100000说明数据已基本同步完成

```
$ ./bin/rtidb --endpoint=172.27.128.33:8541 --role=client
> gettablestatus
  tid  pid  offset    mode          state        enable_expire  ttl   ttl_offset
memused  compress_type
-----
-----  

  4    0    6454924  kTableFollower  kTableNormal  false           0min  0s
1.091 G  kNoCompress
  4    1    5976249  kTableLeader    kTableNormal  false           0min  0s
1.012 G  kNoCompress
> getfollower 4 1
#  tid  pid  leader_offset  follower          offset
-----
```

0	4	1	5923724	172.27.128.31:8541	5920714
1	4	1	5923724	172.27.128.32:8541	5921707

- 一个tablet节点升级完成后，对其他tablet重复上述步骤。(必须等到数据同步完才能升级下一个节点)

### 3 升级java client

- 更新pom文件中java client版本号
- 更新依赖包

## 进程内存磁盘监控

监控rtidb的进程、内存和磁盘使用情况

## RTIDB日志监控

文件	监控关键词	事件	重要等级
logs/rtidb_mon.log	mon : kill	tablet进程被kill	非常重 要
logs/rtidb_mon.log	mon : exit	tablet进程退出	非常重 要
logs/tablet.info.log	reconnect zk	tablet和zk断开重连	重要
logs/rtidb_ns_mon.log	mon : kill	nameserver进程被kill	重要
logs/rtidb_ns_mon.log	mon : exit	nameserver进程退出	重要
logs/nameserver.info.log	offline tablet with endpoint	节点下线	重要
logs/nameserver.info.log	Run OfflineEndpoint	执行节点下线	非常重 要
logs/nameserver.info.log	Run RecoverEndpoint	执行恢复节点	非常重 要
logs/nameserver.info.log	reconnect zk	nameserver和zk断开 重连	重要
logs/nameserver.info.log	kFailed	任务执行失败	重要
logs/nameserver.info.log	The execution time of op is too long	任务执行超时	重要

## RTIDB数据和状态监控

- 分片alive状态监控 ns client执行showtable 查看分片alive状态是否为yes
- 分片主从同步监控 ns client执行showtable 对比主从的offset之差，如果超过一定值就报警(建议值为100000)

# 常见问题

## 1 插入的数据scan不出来

- 查看put的返回值是否put成功
- start\_time和end\_time是不是写反了, start\_time必须大于end\_time
- 如果能get出来却scan不出来, 查看创建的表是不是latest表, latest表不支持scan。(showtable 命令可以查看表类型, ttl后面是min则为absolute表否则为latest表)
- 排查数据是不是过期了。创建表是absolute指定的ttl是以分钟为单位的, 插入时指定的ts精确到毫秒

## 2 连上client后输入的命令不支持或者提示参数不对

client分为tablet client和ns client, 由role指定。如果指定--role=ns\_client启动的是ns client, 如果--role=client启动的是tablet client

ns client连接的是nameserver, 集群版大部分交互命令都是基于ns client的

```
$ ./bin/rtidb --zk_cluster=172.27.2.52:12200 --zk_root_path=/onebox --role=ns_client
```

而tablet client连接的是tablet

```
$ ./rtidb --endpoint=172.27.2.52:9520 --role=client
```

## 3 集群版创建表失败

- 执行showtable看下是不是表已经存在了, 如果已经存在并且确认已有的表没用了drop后再重新创建下或者再换个表名
- 查看磁盘是不是满了
- ttl超过了最大值。默认配置中latest最大为1000条, absolute表最长为30年
- 设置的副本数大于了节点数
- 查看建表schema是否正确, 引号冒号等是否是英文的
- 到nameserver leader的机器日志路径下查看失败原因 ./logs/nameserver.info.log。连接nsclient的时候会显示ns leader

```
$ ./bin/rtidb --zk_cluster=172.27.128.31:8090,172.27.128.32:8090,172.27.128.33:8090
--zk_root_path=/rtidb_cluster --role=ns_client
Welcome to rtidb with version 1.3.10
xxxxxxxxxxxx
ns leader: 172.27.128.33:6312
```

## 4 启动失败

- 查看端口是否被占用

- 查看配置的endpoint中ip是否为本机ip
- 查看zk是否正常

## 5 drop后数据没有放到recycle目录下

检查是否修改了db的路径, 将recycle路径必须和db配到相同的目录下

## 6 节点重启后表alive的状态没有变为no

查看auto\_failover是否开启. auto\_failover关闭情况下不会做任何操作

## 7 java client遍历全表时陷入死循环

遍历全表时如果同一个pk下相同的记录条数超过一次请求的条数(默认为200条)就会导致java client陷入死循环. 可以将RemoveDuplicateByTime设置为true

```
config.setRemoveDuplicateByTime(true);
```

## 8 java client遍历全表时抛出BufferUnderflowException异常

查看迭代器在一次迭代中getDecodedValue是否调用了多次. 在一次迭代中getDecodedValue只能调用一次

## 9 删除pk后内存没有释放掉, 记录数没有变

删除pk后会将pk加入待删除链表, 默认经过两轮gc后会删除

## 10 tablet配的域名, client访问不到集群(不能建表插入查询等)

检查client所在机器是否配置了对应的host