

## 2.6 HBase文档

### 2.6.1 Hbase来源

1. hbase是一个开源的、分布式的、多版本的、可扩展的、非关系型的数据库。
2. hbase是big table的开源的java版本，建立在hdfs基础之上，提供高可靠性、高性能的、列式存储、可伸缩、近实时读写的nosql的数据库系统
3. 数据量越来越大，传统的关系型数据库不能满足存储和查询的需求。而hive虽然能够满足存储的要求，但是hive的本质也是利用底层的mr程序，所以读写速度不快。而且hive不能满足非结构化的、半结构化的存储，hive的主要作用是做分析和统计，hive用于存储是无意义的。

### 2.6.2 Hbase的架构

#### 2.6.2.1 概念

- HBASE是一个数据库----可以提供数据的实时随机读写
- HBASE与mysql、oracle、db2、sqlserver等关系型数据库不同，它是一个NoSQL数据库（非关系型数据库）

- Hbase的表模型与关系型数据库的表模型不同：
- Hbase的表没有固定的字段定义：
- Hbase的表中每行存储的都是一些key-value对
- Hbase的表中有列簇的划分，用户可以指定将哪些kv插入哪个列族
- Hbase的表在物理存储上，是按照列簇来分割的，不同列簇的数据一定存储在不同的文件中
- Hbase的表中的每一行都固定有一个行键，而且每一行的行键在表中不能重复
- Hbase中的数据，包含行键，包含key，包含value，都是byte[]类型，hbase不负责为用户维护数据类型
- HBASE对事务的支持很差
- Hbase的表模型与关系型数据库的表模型不同：
- Hbase的表没有固定的字段定义：
- Hbase的表中每行存储的都是一些key-value对
- Hbase的表中有列簇的划分，用户可以指定将哪些kv插入哪个列族
- Hbase的表在物理存储上，是按照列簇来分割的，不同列簇的数据一定存储在不同的文件中
- Hbase的表中的每一行都固定有一个行键，而且每一行的行键在表中不能重复
- Hbase中的数据，包含行键，包含key，包含value，都是byte[]类型，hbase不负责为用户维护数据类型
- HBASE对事务的支持很差

- HBASE相比于其他nosql数据库(mongodb、redis、cassandra、hazelcast)的特点：

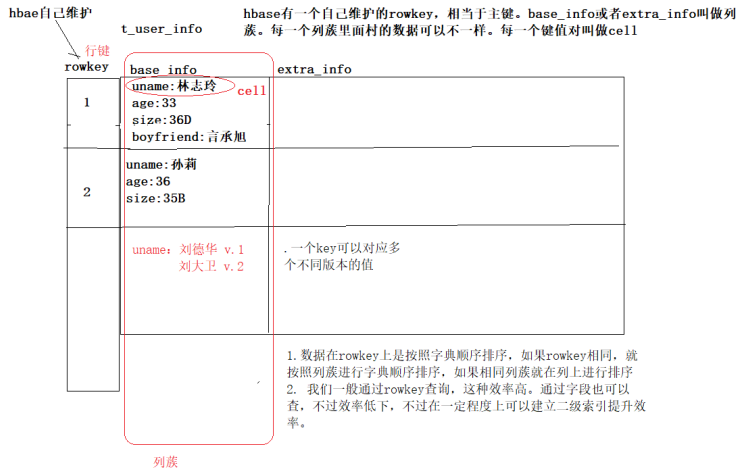
Hbase的表数据存储在HDFS文件系统中。

从而，hbase具备如下特性：存储容量可以线性扩展； 数据存储的安全性可靠性极高！

#### 2.6.2.2 HBase的表模型

传统的RDBMS按行存储数据，我们在添加字段的时候会对整个表结构产生影响，即使没有添加值，也会用null占用字段的值。不方便

t_user_intro				boyfriend
uid	uname	age	size	新增字段
1	林志玲	33	36D	言承旭
2	孙莉	36	35B	null



- hbase的表模型跟mysql之类的关系型数据库的表模型差别巨大
- hbase的表模型中有：行的概念；但没有字段的概念
- 行中存的都是key-value对，每行中的key-value对中的key可以是各种各样，每行中的key-value对的数量也可以是各种各样

## 表模型特点

- 1、一个表，有表名
- 2、一个表可以分为多个列簇（不同列簇的数据会存储在不同文件中）
- 3、表中的每一行有一个“行键rowkey”，而且行键在表中不能重复
- 4、表中的每一对kv数据称作一个cell
- 5、hbase可以对数据存储多个历史版本（历史版本数量可配置）
- 6、整张表由于数据量过大，会被横向切分成若干个region（用rowkey范围标识），不同region的数据也存储在不同文件中
- 7、hbase会对插入的数据按顺序存储：
  - 要点一：首先会按行键排序
  - 要点二：同一行里面的kv会按列簇排序，再按k排序

## HBase存储的数据类型

- hbase中只支持byte[]
- 此处的byte[] 包括了： rowkey, key, value, 列簇名, 表名

## 和Hadoop之间的关系

HBase基于hadoop：HBase的存储依赖于HDFS

## 2.6.2.3 官网介绍

### Welcome to Apache HBase™

Apache HBase™ is the Hadoop database, a distributed, scalable, big data store. Apache HBase™是Hadoop数据库、分布式、可扩展、大数据存储。

Use Apache HBase™ when you need random, realtime read/write access to your Big Data. This project's goal is the hosting of very large tables – billions of rows X millions of columns – atop clusters of commodity hardware. Apache HBase is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable: A Distributed Storage System for Structured Data by Chang et al. Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.

当你需要随机的、实时的读/写访问你的海量数据时你可以使用Apache HBase。这个项目的目标是托管非常大的表 – 位于商品硬件集群之上的十亿行 x 百万列的数据。Apache HBase是一个开源的、分布式的、版本化的、非关系数据库，它模仿了Chang等人的Bigtable：一个结构化数据的分布式存储系统。正如Bigtable利用了谷歌文件系统提供的分布式数据存储，Apache HBase在Hadoop和HDFS上提供了类似Bigtable的功能。

一条日志记录大约在0.8-1。见过的最长的是100多k

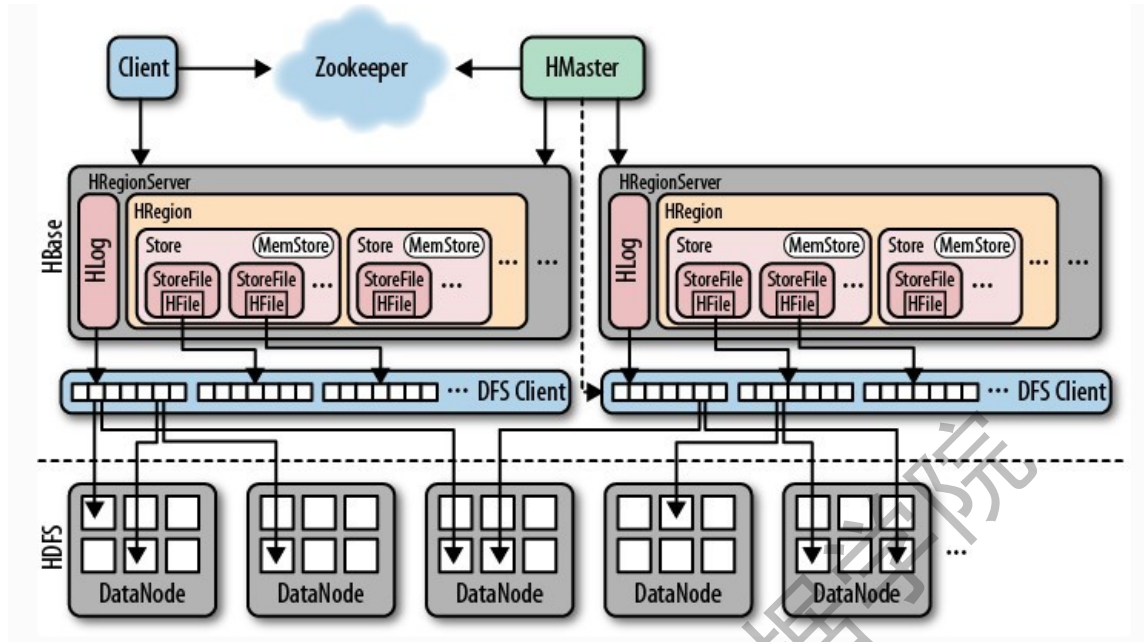
### Features

- Linear and modular scalability. 线性可伸缩的模型
- Strictly consistent reads and writes. 严格一致的读写
- Automatic and configurable sharding of tables. 自动和可配置的分片表
- Automatic failover support between RegionServers. 在RegionServers之间支持自动故障转移
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables. 使用Apache HBase表支持Hadoop MapReduce作业的方便基类
- Easy to use Java API for client access. 易于使用Java API进行客户端访问
- Block cache and Bloom Filters for real-time queries. 块缓存和Bloom过滤器用于实时查询
- Query predicate push down via server-side filters. 通过服务器端过滤器向下推送查询谓词
- Thrift gateway and a REST-ful Web service that supports XML, Protobuf, and binary data encoding options. 支持XML、Protobuf和二进制的网关和Web服务
- Extensible Juby-based (JiRB) shell. 可扩展的基于JiRB的shell
- Support for exporting metrics via the Hadoop metrics subsystem to files or Ganglia, or via JMX. 支持通过Hadoop metrics子系统或JMX将指标导出到文件或Ganglia。

## 适用场景描述

- 1 需要对海量非结构化的数据进行存储
- 2 需要随机近实时的读写管理数据

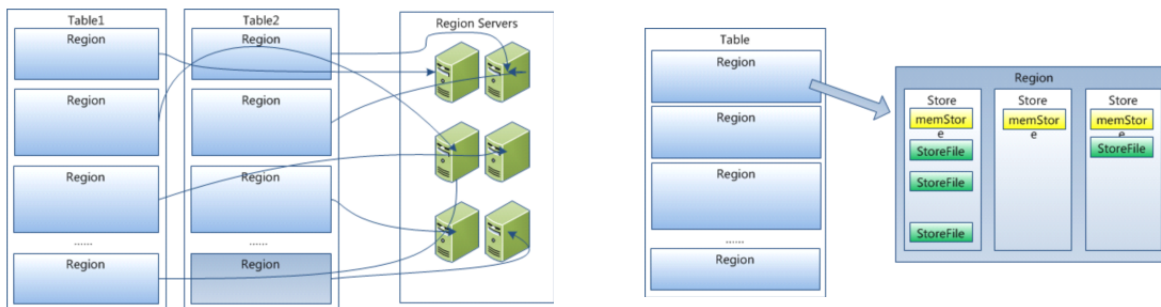
### 2.6.2.4 HBase架构



- **Client** : hbase客户端, 1.包含访问hbase的接口。比如, linux shell, java api。2.除此之外, 它会维护缓存来加速访问hbase的速度。比如region的位置信息。
- **Zookeeper** : 1.监控Hmaster的状态, 保证有且仅有一个活跃的Hmaster。达到高可用。2.它可以存储所有region的寻址入口。如: root表在哪一台服务器上。3.实时监控HregionServer的状态, 感知HRegionServer的上下线信息, 并实时通知给Hmaster。4.存储hbase的部分元数据。
- **HMaster** : 1.为HRegionServer分配Region(新建表等)。2.负责HRegionServer的负载均衡。3.负责Region的重新分配(HRegionServer宕机之后的Region分配, HRegion裂变: 当Region过大之后的拆分)。4.Hdfs上的垃圾回收。5.处理schema的更新请求
- **HRegionServer** : 1.维护HMaster分配给的Region(管理本机的Region)。2.处理client对这些region的读写请求, 并和HDFS进行交互。3.负责切分在运行过程中组件变大的Region。
- **HLog** : 1.对HBase的操作进行记录, 使用WAL写数据, 优先写入log(put操作: 先写日志再写memstore, 这样可以防止数据丢失, 即使丢失也可以回滚)。
- **HRegion** : 1.HBase中分布式存储和负载均衡的最小单元, 它是表或者表的一部分。
- **Store** : 1.相当于一个列簇
- **Memstore** : 1.内存缓冲区, 用于将数据批量刷新到hdfs中, 默认大小为128M
- **HStoreFile** : 1.和HFile概念意义, 不过是一个逻辑概念。HBase中的数据是以HFile存储在Hdfs上。

### 2.6.2.5 各个组件之间的关系

```
hmaster:hregionserver=1:*
hregionserver:hregion=1:*
hregionserver:hlog=1:1
hregion:hstore=1:*
store:memstore=1:1
store:storefile=1:*
storefile:hfile=1:1
```



## 2.6.2.6 总结

**rowkey**:行键，和mysql的主键同理，不允许重复。

**columnfamily**: 列簇，列的集合之意。

**column**:列

**timestamp**:时间戳，默认显示最新的时间戳，可用于控制k对应的多个版本值，默认查最新的数据

**version**:版本号，表示记录数据的版本

**cell**:单元格，kv就是cell

模式: 无

数据类型:只存储byte[]

多版本: 每个值都可以有多个版本

列式存储: 一个列簇存储到一个目录

稀疏存储: 如果一个kv为null，不占用存储空间

## 2.6.3 Hbase集群搭建

### 2.6.3.1 单机节点安装

- 解压

```
[root@centos1 home]# tar -zxvf hbase-1.2.1-bin.tar.gz -C /usr/local/
```

- 配置环境变量

```
export HBASE_HOME=/usr/local/hbase-1.2.1
export
PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$HBASE_HOME/bin
```

- hbase-env.sh

```
# The java implementation to use. Java 1.7+ required.
export JAVA_HOME=/usr/local/java/jdk1.8.0_45

# Tell HBase whether it should manage it's own instance of Zookeeper or not.
export HBASE_MANAGES_ZK=true
```

- hbase-site.xml

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///usr/local/hbase-1.2.1/hbasedata</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/usr/local/hbase-1.2.1/zkdata</value>
  </property>
</configuration>
```

- 启动HBase服务

```
[root@centos1 conf]# start-hbase.sh
```

```
[root@centos1 hbase-1.2.1]# jps
4593 HMaster
3272 ResourceManager
4666 Jps
2923 NameNode
3116 SecondaryNameNode
```

- hbase的客户端连接

```
[root@centos1 logs]# hbase shell
hbase(main):002:0> status
1 active master, 0 backup masters, 1 servers, 0 dead, 2.0000 average load
-----
hbase(main):003:0> version
1.2.1, r8d8a7107dc4ccbf36a92f64675dc60392f85c015, wed Mar 30 11:19:21 CDT 2016
-----
hbase(main):004:0> whoami
root (auth:SIMPLE)
groups: root
-----
hbase(main):005:0> help 'table_help'
Help for table-reference commands.
```

You can either create a table via 'create' and then manipulate the table via commands like 'put', 'get', etc.

See the standard help information for how to use each of these commands.

However, as of 0.96, you can also get a reference to a table, on which you can invoke commands.

For instance, you can get create a table and keep around a reference to it via:

```
hbase> t = create 't', 'cf'
```

or, if you have already created the table, you can get a reference to it:

```
hbase> t = get_table 't'
```

You can do things like call 'put' on the table:

```
hbase> t.put 'r', 'cf:q', 'v'
```

which puts a row 'r' with column family 'cf', qualifier 'q' and value 'v' into table t.

To read the data out, you can scan the table:

```
hbase> t.scan
```

which will read all the rows in table 't'.

Essentially, any command that takes a table name can also be done via table reference.

Other commands include things like: get, delete, deleteall, get\_all\_columns, get\_counter, count, incr. These functions, along with the standard JRuby object methods are also available via tab completion.

For more information on how to use each of these commands, you can also just type:

```
hbase> t.help 'scan'
```

which will output more information on how to use that command.

You can also do general admin actions directly on a table; things like enable, disable, flush and drop just by typing:

```
hbase> t.enable
hbase> t.flush
hbase> t.disable
hbase> t.drop
```

Note that after dropping a table, your reference to it becomes useless and further usage is undefined (and not recommended).

```
-----
hbase(main):017:0> help 'ddl'
-----
```

```
hbase(main):026:0> create 't1', {NAME => 'f1', VERSIONS => 5}
hbase(main):026:0> list
hbase(main):026:0> disable 't1'
hbase(main):026:0> drop 't1'
```

### 2.6.3.2 伪分布式安装

- 作用

在快速启动单机模式之后，可以重新配置HBase，使其在伪分布式模式下运行。伪分布式模式意味着HBase仍然完全运行在单个主机上，但是每个HBase守护进程(HMaster、HRegionServer和ZooKeeper)都作为单独的进程运行：在独立模式下，所有守护进程都运行在一个jvm进程/实例中。默认情况下，除非您配置了hbase。如quickstart中所述，您的数据仍然存储在/tmp/中。在本演练中，我们将数据存储在HDFS中，假设您有可用的HDFS。您可以跳过HDFS配置，继续在本地文件系统中存储数据。

- hbase-site.xml

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://centos1:9000/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/usr/local/hbase-1.2.1/zkdata</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
</configuration>
```

- 启动

```
[root@centos1 conf]# start-hbase.sh
[root@centos1 conf]# jps
6352 HRegionServer
6232 HMaster
3272 ResourceManager
6169 HQuorumPeer
2923 NameNode
3116 SecondaryNameNode
6445 Jps
```

- hdfs的目录

Browse Directory

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	42 B	Jan 09 22:42	1	128 MB	hbase.id
-rw-r--r--	root	supergroup	7 B	Jan 09 22:42	1	128 MB	hbase.version
drwxr-xr-x	root	supergroup	0 B	Jan 09 22:42	0	0 B	.tmp
drwxr-xr-x	root	supergroup	0 B	Jan 09 22:43	0	0 B	MasterProcWALs
drwxr-xr-x	root	supergroup	0 B	Jan 09 22:42	0	0 B	WALs
drwxr-xr-x	root	supergroup	0 B	Jan 09 22:42	0	0 B	data
drwxr-xr-x	root	supergroup	0 B	Jan 09 22:42	0	0 B	oldWALs

Showing 1 to 7 of 7 entries Previous 1 Next

2.6.3.3 全分布式安装

- 作用

HBASE是一个分布式系统

其中有一个管理角色: **HMaster**(一般2台, 一台**active**, 一台**backup**)

其他的数据节点角色: **HRegionServer**(很多台, 看数据容量)

实际上, 您需要一个完全分布式的配置来全面测试**HBase**, 并在实际场景中使用它。在分布式配置中, 集群包含多个节点, 每个节点运行一个或多个**HBase**守护进程。这些包括主实例和备份主实例、多个**Zookeeper**节点和多个**RegionServer**节点。

- 角色分配

```
centos1:namenode hmaster
centos2:datanode regionserver zookeeper backup master
centos3:datanode regionserver zookeeper
centos4:datanode regionserver zookeeper
```

- 安装zookeeper
- hbase-env.sh

```
# The java implementation to use. Java 1.7+ required.
export JAVA_HOME=/usr/local/java/jdk1.8.0_45
export HBASE_MANAGES_ZK=false
```

- hbase-site.xml

```
<configuration>
  <!-- 指定hbase在HDFS上存储的路径 -->
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://centos1:9000/hbase</value>
  </property>
  <!-- 指定hbase是分布式的 -->
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <!-- 指定zk的地址, 多个用“,”分割 -->
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>centos2:2181,centos3:2181,centos4:2181</value>
  </property>
</configuration>
```

- regionservers

```
centos2
centos3
centos4
```

- 在hbase的conf创建backup-master的文件, 并在其中添加主机名centos2

```
centos2
```

- 分发



```
scp -r hbase-1.2.1/ centos2:/usr/local/  
scp -r hbase-1.2.1/ centos3:/usr/local/  
scp -r hbase-1.2.1/ centos4:/usr/local/
```

- 启动hbase

1. 现启动hadoop和zookeeper
2. start-hbase.sh

- 解决时间差

```
yum -y install ntpdate  
ntpdate ntp1.aliyun.com
```

- 查看hdfs

## Browse Directory

/hbase

Go!

Show

25

▼

entries

Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
-rw-r--r--	root	supergroup	42 B	Oct 07 22:21	3	128 MB	hbase.id	
-rw-r--r--	root	supergroup	7 B	Oct 07 22:21	3	128 MB	hbase.version	
drwxr-xr-x	root	supergroup	0 B	Oct 07 22:21	0	0 B	.tmp	
drwxr-xr-x	root	supergroup	0 B	Oct 07 22:29	0	0 B	MasterProcWALs	
drwxr-xr-x	root	supergroup	0 B	Oct 07 22:29	0	0 B	WALs	
drwxr-xr-x	root	supergroup	0 B	Oct 07 22:29	0	0 B	corrupt	
drwxr-xr-x	root	supergroup	0 B	Oct 07 22:21	0	0 B	data	
drwxr-xr-x	root	supergroup	0 B	Oct 07 22:29	0	0 B	oldWALs	

Showing 1 to 8 of 8 entries

Previous

1

Next

Hadoop, 2017.

- 查看hbase

```
[root@centos1 apps]# jps  
1888 QuorumPeerMain  
2438 ResourceManager  
2822 HMaster  
2090 NameNode  
2284 SecondaryNameNode  
3373 Jps  
2943 HRegionServer
```

```
[root@centos1 apps]# netstat -nlt | grep 2822  
tcp        0      0 :::ffff:192.168.49.250:16000 :::*  
LISTEN     2822/java  
tcp        0      0 :::16010          :::*  
LISTEN     2822/java
```

ServerName	Start time	Version	Requests Per Second	Num. Regions
hive1,16020,1538922091216	Sun Oct 07 22:21:31 CST 2018	1.2.1	0	1
hive2,16020,1538922079751	Sun Oct 07 22:21:19 CST 2018	1.2.1	0	1
Total 2			0	2

- 在zookeeper上的记录

```
[root@hive1 zookeeper-3.4.5]# bin/zkCli.sh
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper, hbase]

[zk: localhost:2181(CONNECTED) 1] ls /hbase
[replication, meta-region-server, rs, splitWAL, backup-masters, table-lock,
flush-table-proc, region-in-transition, online-snapshot, master, running,
recovering-regions, draining, namespace, hbaseid, table]
```

## 2.6.4 Hbase的shell连接

### 2.6.4.1 普通连接

- 启动客户端

```
[root@centos1 bin]# ./hbase shell
```

- 帮助语法

```
help '命令组'
e.g.    help 'create'
```

### 2.6.4.2 测试Namespace

```
1. list_namespace: 查询所有命名空间
hbase(main):008:0> list_namespace
NAMESPACE
default
hbase

2. list_namespace_tables : 查询指定命名空间的表
hbase(main):014:0> list_namespace_tables 'hbase'
TABLE
meta
namespace

3. create_namespace : 创建指定的命名空间
hbase(main):018:0> create_namespace 'ns1'
hbase(main):019:0> list_namespace
NAMESPACE
default
```

```

hbase
ns1

4. describe_namespace : 查询指定命名空间的结构
hbase(main):021:0> describe_namespace 'ns1'
DESCRIPTION
{NAME => 'ns1'}

5. alter_namespace : 修改命名空间的结构
hbase(main):022:0> alter_namespace 'ns1', {METHOD => 'set', 'name' => 'lixixi'}

hbase(main):023:0> describe_namespace 'ns1'
DESCRIPTION
{NAME => 'ns1', name => 'lixixi'}

hbase(main):022:0> alter_namespace 'ns1', {METHOD => 'unset', NAME => 'name'}
hbase(main):023:0> describe_namespace 'ns1'

6. 删除命名空间
hbase(main):026:0> drop_namespace 'ns1'

hbase(main):027:0> list_namespace
NAMESPACE
default
hbase

7. 利用新添加的命名空间建表
hbase(main):032:0> create 'ns1:t1', 'f1', 'f2'

=> Hbase::Table - ns1:t1
hbase(main):033:0> list
TABLE
ns1:t1

=> ["ns1:t1"]

```

## 2.6.5 DDL和DML的操作

### 2.6.5.1 DDL

#### 2.6.5.1.1 建表

```

create : 建表

hbase(main):010:0> create 'user_info','base_info','extra_info'

=> Hbase::Table - user_info

hbase(main):043:0> create 'ns1:user_info', {NAME=>'base_info',
BLOOMFILTER=>'ROWCOL',VERSIONS=>'3'}

```

#### Tables

User Tables System Tables Snapshots  
1 table(s) in set [Details]

Namespace	Table Name	Online Regions	Offline Regions	Failed Regions	Split Regions	Other Regions	Description
default	<a href="#">user_info</a>	1	0	0	0	0	'user_info', (NAME => 'base_info'), (NAME => 'extra_info')

### 2.6.5.2 list : 查询所有的表

```
hbase(main):002:0> list
TABLE
ns1:t1
ns1:user_info
2 row(s) in 0.2830 seconds

=> ["ns1:t1", "ns1:user_info"]
```

### 2.6.5.3 describe : 查询表结构

```
hbase(main):003:0> describe 'ns1:user_info'
Table ns1:user_info is ENABLED
ns1:user_info
COLUMN FAMILIES DESCRIPTION
{NAME => 'base_info', BLOOMFILTER => 'ROWCOL', VERSIONS => '3', IN_MEMORY =>
'false', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL =>
'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS
=> '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
```

### 2.6.5.4 create , splits : 创建表分片

```
hbase(main):007:0> create 'ns1:t2', 'f1', SPLITS => ['10', '20', '30', '40']
```

### 2.6.5.5 修改表

- alter : 修改表, 添加修改列簇信息

```
hbase(main):009:0> alter 'ns1:t1', {NAME=>'lix_info'}

hbase(main):010:0> describe 'ns1:t1'
Table ns1:t1 is ENABLED
ns1:t1
COLUMN FAMILIES DESCRIPTION
{NAME => 'f1', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', B
LOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
{NAME => 'f2', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', B
LOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
{NAME => 'lix_info', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY =>
'false', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL =>
'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS =>
'0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
3 row(s) in 0.0250 seconds
```

- 删除列簇

```
hbase(main):014:0> alter 'ns1:t1', 'delete' => 'lixi_info'

hbase(main):015:0> describe 'ns1:t1'
Table ns1:t1 is ENABLED
ns1:t1
COLUMN FAMILIES DESCRIPTION
{NAME => 'f1', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', B
LOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
{NAME => 'f2', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', B
LOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
2 row(s) in 0.0170 seconds
```

- 删除表(先要disable表，再删除表)

```
hbase(main):016:0> disable 'ns1:t1'
0 row(s) in 2.2790 seconds

hbase(main):017:0> drop 'ns1:t1'
0 row(s) in 1.2900 seconds

hbase(main):018:0> list
TABLE
ns1:t2
ns1:user_info
2 row(s) in 0.0090 seconds

=> ["ns1:t2", "ns1:user_info"]
```

## 2.6.6.2 DML

### 2.6.2.2.1 插入数据 (put命令，不能一次性插入多条)

```
hbase(main):012:0> put 'user_info','001','base_info:username','lixi'
0 row(s) in 0.9800 seconds
```

### 2.6.2.2.2 scan扫描

```
hbase(main):024:0> scan 'user_info'
ROW                                COLUMN+CELL
001                                column=base_info:love,
timestamp=1538897913186, value=basketball
001                                column=base_info:username,
timestamp=1538897633942, value=lixi
002                                column=base_info:username,
timestamp=1538898168513, value=lishiming
2 row(s) in 0.0520 seconds
```

- 通过指定版本查询

```
hbase(main):024:0> scan 'user_info', {RAW => true, VERSIONS => 1}
ROW                                COLUMN+CELL
```

```

001
timestamp=1546922817429, value=32
001
timestamp=1546923712904, value=rock
001
timestamp=1546922881922, value=shuai
001
timestamp=1546922931075, value=111
1 row(s) in 0.0160 seconds

```

```

hbase(main):025:0> scan 'user_info', {RAW => true, VERSIONS => 2}
ROW                                COLUMN+CELL
001                                column=base_info:age,
timestamp=1546922817429, value=32
001                                column=base_info:name,
timestamp=1546923712904, value=rock
001                                column=base_info:name,
timestamp=1546922810789, value=lixu
001                                column=extra_info:feature,
timestamp=1546922881922, value=shuai
001                                column=super_info:size,
timestamp=1546922931075, value=111
1 row(s) in 0.0180 seconds

```

- 查询指定列的数据

```

hbase(main):014:0> scan 'user_info', {COLUMNS => 'base_info:name'}
ROW                                COLUMN+CELL
001                                column=base_info:name,
timestamp=1546923712904, value=rock

```

- 分页查询

```

hbase(main):021:0> scan 'user_info', {COLUMNS => ['base_info:name',
'base_info:age'], LIMIT => 10, STARTROW => '001'}
ROW                                COLUMN+CELL
001                                column=base_info:age,
timestamp=1546922817429, value=32
001                                column=base_info:name,
timestamp=1546923712904, value=rock

```

### 2.6.2.2.3 get查询

```
hbase(main):015:0> get 'user_info','001','base_info:username'
COLUMN                                CELL
  base_info:username                  timestamp=1538897633942,
value=lixi
1 row(s) in 0.2500 seconds
```

```
hbase(main):017:0> put 'user_info','001','base_info:love','basketball'
0 row(s) in 0.0140 seconds
```

```
hbase(main):018:0> get 'user_info','001'
COLUMN                                CELL
  base_info:love                      timestamp=1538897913186,
value=basketball
  base_info:username                  timestamp=1538897633942,
value=lixi
```

- 根据时间戳查询

```
hbase(main):029:0> get 'user_info', '001', {TIMERANGE => [1546922817429,
1546922931075]}
COLUMN                                CELL
  base_info:age                      timestamp=1546922817429,
value=32
  extra_info:feature                  timestamp=1546922881922,
value=shuai
```

#### 2.6.2.2.4 hbase的一个重要特性：排序特性 (rowkey)

插入到hbase中去的数据，hbase会自动排序存储：

排序规则： 首先看行键，然后看列族名，然后看列（key）名； 按字典顺序

Hbase的这个特性跟查询效率有极大的关系

比如：一张用来存储用户信息的表，有名字，户籍，年龄，职业....等信息

然后，在业务系统中经常需要：

查询某个省的所有用户

经常需要查询某个省的指定姓的所有用户

思路：如果能将相同省的用户在hbase的存储文件中连续存储，并且能将相同省中相同姓的用户连续存储，那么，上述两个查询需求的效率就会提高！！

做法：将查询条件拼到rowkey内

#### 2.6.2.2.5 更新数据

```
hbase(main):010:0> put 'user_info', '001', 'base_info:name', 'rock'
0 row(s) in 0.1420 seconds
```

#### 2.6.2.2.7 incr

```
hbase(main):053:0> incr 'user_info', '002', 'base_info:age3'
COUNTER VALUE = 1
0 row(s) in 0.0290 seconds
```

```
hbase(main):055:0> scan 'user_info'
```

ROW	COLUMN+CELL
001	column=base_info:age,
timestamp=1546922817429, value=32	
001	column=base_info:name,
timestamp=1546923712904, value=rock	
001	column=extra_info:feature,
timestamp=1546922881922, value=shuai	
001	column=super_info:size,
timestamp=1546922931075, value=111	
002	column=base_info:age2,
timestamp=1546924991038, value=11	
002	column=base_info:age3,
timestamp=1546925153059, value=\x00\x00\x00\x00\x00\x00\x00\x02	
2 row(s) in 0.1020 seconds	

#### 2.6.2.2.8 删除数据

- 删除一个kv数据

```
hbase(main):058:0> delete 'user_info', '002', 'base_info:age3'
0 row(s) in 0.0180 seconds

hbase(main):059:0> scan 'user_info'
```

ROW	COLUMN+CELL
001	column=base_info:age,
timestamp=1546922817429, value=32	
001	column=base_info:name,
timestamp=1546923712904, value=rock	
001	column=extra_info:feature,
timestamp=1546922881922, value=shuai	
001	column=super_info:size,
timestamp=1546922931075, value=111	
002	column=base_info:age2,
timestamp=1546924991038, value=11	

- 删除一行数据

```
hbase(main):028:0> deleteall 'user_info','001'
0 row(s) in 0.0100 seconds

hbase(main):029:0> scan 'user_info'
```

ROW	COLUMN+CELL
-----	-------------

- 删除指定的版本

```
hbase(main):081:0> delete 'user_info','001','extra_info:feature',
TIMESTAMP=>1546922931075
```

- 删除一个表数据

```
disable 'user_info'
drop 'user_info'
```

- 判断表是否存在



```
hbase(main):083:0> exists 'user_info'
Table user_info does exist
0 row(s) in 0.0130 seconds
```

- 表生效和失效

```
hbase(main):084:0> enable 'user_info'
0 row(s) in 0.0130 seconds
```

```
hbase(main):085:0> disable 'user_info'
0 row(s) in 2.5100 seconds
```

- 统计表行数

```
hbase(main):088:0> count 'user_info'
1 row(s) in 0.0300 seconds
```

=> 1

- 清空表数据

```
hbase(main):089:0> truncate 'user_info'
```

## 2.6.6 Hbase的Java api

### 2.6.6.1 准备工作

- 创建Maven的Java项目并配置文件
- 导入依赖

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-client</artifactId>
  <version>1.2.1</version>
</dependency>
```

- 在window中hosts (要把所有的关于master的映射也配置上, 否则无法连接)

```
192.168.49.150 hbase1
192.168.49.151 hbase2
192.168.49.152 hbase3
192.168.49.153 hbase4
```

### 2.6.6.2 HBase服务的连接

```
/**
 * 连接到HBase的服务
 */
public class Demo1_Conn {
    public static void main(String[] args) throws IOException {
        //1. 获取连接配置对象
        Configuration configuration = new Configuration();
        //2. 设置连接hbase的参数
```

```

        configuration.set("hbase.zookeeper.quorum",
            "hbase2:2181,hbase3:2181,hbase4:2181");
        //3. 获取Admin对象
        HBaseAdmin hBaseAdmin = new HBaseAdmin(configuration);
        //4. 检验指定表是否存在, 来判断是否连接到hbase
        boolean flag = hBaseAdmin.tableExists("ns1:user_info");
        //5. 打印
        System.out.println(flag);
    }
}

```

### 2.6.6.3 Namespace操作

- 设计模板类

```

/**
 * 操作namespace
 */
public class Demo2_Namespace {

    private HBaseAdmin hBaseAdmin;

    @Before
    public void before() throws IOException {
        //1. 获取连接配置对象
        Configuration configuration = new Configuration();
        //2. 设置连接hbase的参数
        configuration.set("hbase.zookeeper.quorum",
            "hbase2:2181,hbase3:2181,hbase4:2181");
        //3. 获取Admin对象
        hBaseAdmin = new HBaseAdmin(configuration);
    }

    @After
    public void close() throws IOException {
        hBaseAdmin.close();
    }
}

```

- 创建Namespace

```

/**
 * 创建namespace
 */
@Test
public void createNamespace() throws IOException {
    //1. 创建namespace对象
    NamespaceDescriptor descriptor = NamespaceDescriptor.create("lixixi").build();
    //2. 提交hbase中创建对象
    hBaseAdmin.createNamespace(descriptor);
}

```

- 解决连接对象过时间问题

```

@Before
public void before2() throws IOException {
    //1. 获取连接配置对象
    Configuration configuration = new Configuration();
    //2. 设置连接hbase的参数
    configuration.set("hbase.zookeeper.quorum",
        "hbase2:2181,hbase3:2181,hbase4:2181");
    //3. 获取Admin对象
    Connection connection = ConnectionFactory.createConnection(configuration);
    HBaseAdmin = (HBaseAdmin) connection.getAdmin();
}

```

- 提取工具类

```

/**
 * HBase Client工具类
 */
public class HBaseUtils {

    private static final Logger logger = Logger.getLogger(HBaseUtils.class);

    private final static String CONNECT_KEY = "hbase.zookeeper.quorum";
    private final static String CONNECT_VALUE =
        "hbase2:2181,hbase3:2181,hbase4:2181";

    /**
     * 获取Admin对象
     */
    public static Admin getAdmin() {
        //1. 获取连接配置对象
        Configuration configuration = new Configuration();
        //2. 设置连接hbase的参数
        configuration.set(CONNECT_KEY, CONNECT_VALUE);
        //3. 获取Admin对象
        Connection connection = null;
        Admin admin = null;
        try {
            connection = ConnectionFactory.createConnection(configuration);
            admin = connection.getAdmin();
        } catch (IOException e) {
            logger.warn("连接HBase的时候异常!", e);
        }
        return admin;
    }

    public static void close(Admin admin) {
        if(null != admin) {
            try {
                admin.close();
                admin.getConnection().close();
            } catch (IOException e) {
                logger.warn("关闭admin的时候异常!", e);
            }
        }
    }
}

```

- 使用工具类操作Namespace对象——列举Namespace

```
/**
 * 列举namespace
 */
@Test
public void listNamesapce() throws IOException {
    //1. 获取admin对象
    Admin admin = HBaseUtils.getAdmin();
    //2. 获取namespace的所有描述器
    NamespaceDescriptor[] namespaceDescriptors =
admin.listNamespaceDescriptors();
    //3. 遍历
    for (NamespaceDescriptor descriptor : namespaceDescriptors) {
        System.out.println(descriptor);
    }
    //4. 关闭
    HBaseUtils.close(admin);
}
```

- 列举Namespace对应的表名

```
@Test
public void listNamespaceTables() throws IOException {
    //1. 获取admin对象
    Admin admin = HBaseUtils.getAdmin();
    //2. 获取name所有的表名
    TableName[] tableNames = admin.listTableNamesByNamespace("ns1");
    //3. 遍历
    for(TableName tableName : tableNames) {
        System.out.println(tableName.getNameAsString());
    }
    //4. 关闭
    HBaseUtils.close(admin);
}
```

- 列举所有Namespace对应所有的表

```
/**
 * 列举所有namesapce对应表名
 */
@Test
public void listAllNamespaceTables() throws IOException {
    //1. 获取admin对象
    Admin admin = HBaseUtils.getAdmin();
    //2. 获取name所有的表名
    TableName[] tableNames = admin.listTableNames();
    //3. 遍历
    for(TableName tableName : tableNames) {
        System.out.println(tableName.getNameAsString());
    }
    //4. 关闭
    HBaseUtils.close(admin);
}
```

- 列出指定Namespace指定的表描述器

```

/**
 * 列举所有namespace对应表描述器
 */
@Test
public void listAllNamespaceTableDescriptor() throws IOException {
    //1. 获取admin对象
    Admin admin = HBaseUtils.getAdmin();
    //2. 获取name所有的表名
    HTableDescriptor[] tableDescriptors =
    admin.listTableDescriptorsByNamespace("ns1");
    //3. 遍历
    for(HTableDescriptor descriptor : tableDescriptors) {
        System.out.println(descriptor.getTable_name());
    }
    //4. 关闭
    HBaseUtils.close(admin);
}

```

- 删除Namespace

```

/**
 * 删除namespace
 * 只能删除空的namespace
 */
@Test
public void dropNamespace() throws IOException {
    //1. 获取admin对象
    Admin admin = HBaseUtils.getAdmin();
    //2. 删除
    admin.deleteNamespace("lee");
    //3. 关闭
    HBaseUtils.close(admin);
}

```

#### 2.6.6.4 Table DDL

- 模板类

```

/**
 * Table's CRUD DDL
 */
public class Demo4_Table {

    Admin admin = HBaseUtils.getAdmin();

    @After
    public void after() {
        HBaseUtils.close(admin);
    }
}

```

- 建表

```

/**
 * create 't2', {NAME => 'default', VERSIONS => 1}

```

```

*/
@Test
public void createTable() throws IOException {
    //1. 创建表描述器
    HTableDescriptor tableDescriptor = new
    HTableDescriptor(TableName.valueOf("user_info"));
    //2. 创建列簇表述qi
    HColumnDescriptor columnDescriptor = new HColumnDescriptor("base_info");
    //2.1 设置列簇版本从1到5
    columnDescriptor.setVersions(1, 5);
    columnDescriptor.setTimeToLive(24*60*60); // 秒为单位
    //      columnDescriptor.setMinVersions(1);
    //      columnDescriptor.setMaxVersions(5);
    //      columnDescriptor.setBloomFilterType(BloomType.ROW);
    //      columnDescriptor.setDFSReplication(3); // 设置HBase数据存放的副本数
    HColumnDescriptor columnDescriptor2 = new HColumnDescriptor("extra_info");
    columnDescriptor2.setVersions(1, 5);
    columnDescriptor2.setTimeToLive(24*60*60); // 秒为单位
    //2.2 将列簇添加到表中
    tableDescriptor.addFamily(columnDescriptor);
    tableDescriptor.addFamily(columnDescriptor2);
    //3. 提交
    admin.createTable(tableDescriptor);
}

```

- 修改表1：如果直接修改之前的保存数据没有了，类似于sql的修改，应该先查后改

```

/**
 * 修改表1
 *
 */
@Test
public void modifyTable() throws IOException {
    //1. 创建表描述器
    TableName tableName = TableName.valueOf("user_info");
    HTableDescriptor tableDescriptor = new HTableDescriptor(tableName);
    //2. 创建列簇表述qi
    HColumnDescriptor columnDescriptor = new HColumnDescriptor("lixix_info");
    //2.1 设置列簇版本从1到5
    columnDescriptor.setVersions(1, 5);
    columnDescriptor.setTimeToLive(24*60*60); // 秒为单位
    //2.2 将列簇添加到表中
    tableDescriptor.addFamily(columnDescriptor);
    //3. 提交
    admin.modifyTable(tableName, tableDescriptor);
}

```

- 修改表2

```

/**
 * 修改表2：在原来的基础之上进行修改
 *
 */
@Test
public void modifyTable2() throws IOException {
    //1. 创建表描述器

```

```

TableName tableName = TableName.valueOf("user_info");
HTableDescriptor tableDescriptor = admin.getTableDescriptor(tableName);
//2. 创建列簇表述qi
HColumnDescriptor columnDescriptor = new HColumnDescriptor("base_info");
//2.1 设置列簇版本从1到5
columnDescriptor.setVersions(1, 5);
columnDescriptor.setTimeToLive(24*60*60); // 秒为单位
//2.2 将列簇添加到表中
tableDescriptor.addFamily(columnDescriptor);
//3. 提交
admin.modifyTable(tableName, tableDescriptor);
}

```

- 删除列簇1

```

/**
 * 修改表，删除列簇
 */
@Test
public void deleteColumnFamily() throws IOException {
    //1. 创建表并删除列簇
    TableName tableName = TableName.valueOf("user_info");
    admin.deleteColumn(tableName, "lixix_info".getBytes());
}

```

- 删除列簇2

```

/**
 * 删除列簇2
 */
@Test
public void deleteColumnFamily2() throws IOException {
    //1. 创建表描述器
    TableName tableName = TableName.valueOf("user_info");
    HTableDescriptor tableDescriptor = admin.getTableDescriptor(tableName);
    //2. 获取要删除的列簇描述器
    HColumnDescriptor columnDescriptor =
tableDescriptor.removeFamily("base_info".getBytes());
    //3. 删除
    admin.modifyTable(tableName, tableDescriptor);
}

```

- 例举出某表的所有列簇

```

/**
 * 例举出某表的所有列簇
 */
@Test
public void listColumnFamily() throws IOException {
    //1. 获取表描述器
    HTableDescriptor tableDescriptor =
admin.getTableDescriptor(TableName.valueOf("user_info"));
    //2. 获取所有的列簇
    HColumnDescriptor[] columnFamilies = tableDescriptor.getColumnFamilies();
    //3. 遍历
    for(HColumnDescriptor columnDescriptor : columnFamilies) {

```

```

        System.out.println(columnDescriptor.getNameAsString());
        System.out.println(columnDescriptor.getBlocksize());
        System.out.println(columnDescriptor.getBloomFilterType());
    }
}

```

- 删除表

```

/**
 * 删除表
 */
@Test
public void dropTable() throws IOException {
    TableName tableName = TableName.valueOf("t1");
    if(admin.tableExists(tableName)) {
        if (!admin.isTableDisabled(tableName)) {
            admin.disableTable(tableName);
        }
        admin.deleteTable(TableName.valueOf("t1"));
    }
}

```

## 2.6.6.5 Table DML

- 获取Table对象

```

public static Table getTable() {
    return getTable("ns1:user_info");
}

public static Table getTable(String tablename) {
    Table table = null;
    if(StringUtils.isEmpty(tablename)) {
        try {
            table = connection.getTable(TableName.valueOf(tablename));
        } catch (IOException e) {
            logger.warn("获取表产生异常!", e);
        }
    }
    return table;
}

public static void close(Table table) {
    if(table != null) {
        try {
            table.close();
        } catch (IOException e) {
            logger.warn("关闭table的时候产生异常!", e);
        }
    }
}
}

```

- 修改HBaseUtils

```

/**
 * HBase Client工具类

```



```

*/
public class HBaseUtils {

    private static final Logger logger = Logger.getLogger(HBaseUtils.class);

    private final static String CONNECT_KEY = "hbase.zookeeper.quorum";
    private final static String CONNECT_VALUE =
"hbase2:2181,hbase3:2181,hbase4:2181";
    private static Connection connection;

    static {
        //1. 获取连接配置对象
        Configuration configuration = HBaseConfiguration.create();
        //2. 设置连接hbase的参数
        configuration.set(CONNECT_KEY, CONNECT_VALUE);
        //3. 获取connection对象
        try {
            connection = ConnectionFactory.createConnection(configuration);
        } catch (IOException e) {
            logger.warn("连接HBase的时候异常!", e);
        }
    }

    /**
     * 获取Admin对象
     */
    public static Admin getAdmin() {
        Admin admin = null;
        try {
            admin = connection.getAdmin();
        } catch (IOException e) {
            logger.warn("连接HBase的时候异常!", e);
        }
        return admin;
    }

    public static void close(Admin admin) {
        if(null != admin) {
            try {
                admin.close();
            } catch (IOException e) {
                logger.warn("关闭admin的时候异常!", e);
            }
        }
    }

    public static Table getTable() {
        return getTable("ns1:user_info");
    }

    public static Table getTable(String tablename) {
        Table table = null;
        if(StringUtils.isEmpty(tablename)) {
            try {
                table = connection.getTable(TableName.valueOf(tablename));
            } catch (IOException e) {
                logger.warn("获取表产生异常!", e);
            }
        }
    }
}

```

```

    }
    return table;
}

public static void close(Table table) {
    if(table != null) {
        try {
            table.close();
        } catch (IOException e) {
            logger.warn("关闭table的时候产生异常!", e);
        }
    }
}
}
}

```

- Put

```

/**
 * DML
 */
public class Demo5_Table {
    Table table = HBaseUtils.getTable();

    @After
    public void after() {
        HBaseUtils.close(table);
    }
    /**
     * 插入数据
     */
    @Test
    public void putData() throws IOException {
        //1. 获取Table对象
        Table table = HBaseUtils.getTable();
        //2. 获取Put对象,通过rowkey指定
        Put put = new Put(Bytes.toBytes("003"));
        /**
         * 3. 设置插入的数据
         * 列簇、列名、value
         */
        put.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
            Bytes.toBytes("narudo"));
        put.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("age"),
            Bytes.toBytes("15"));
        put.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("sex"),
            Bytes.toBytes("male"));
        //4. 提交
        table.put(put);
    }
}

```

- 批量插入数据

```

/**
 * 批量插入数据
 */

```

```

@Test
public void batchPutDatas() throws IOException {
    //0. 创建集合
    List<Put> list = new ArrayList<>();

    //1. 创建put对象指定行键
    Put rk004 = new Put(Bytes.toBytes("004"));
    Put rk005 = new Put(Bytes.toBytes("005"));
    Put rk006 = new Put(Bytes.toBytes("006"));

    //2. 创建列簇

    rk004.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("name"), Bytes.toBytes("gaoyuanyuan"));

    rk005.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("age"), Bytes.toBytes("18"));

    rk005.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("sex"), Bytes.toBytes("2"));

    rk006.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("name"), Bytes.toBytes("fanbinbin"));

    rk006.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("age"), Bytes.toBytes("18"));

    rk006.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("sex"), Bytes.toBytes("2"));

    //3. 添加数据
    list.add(rk004);
    list.add(rk005);
    list.add(rk006);

    table.put(list);
}

```

- Get查询：查询单个列簇的所有列

```

/**
 * get查询数据
 */
@Test
public void getData() throws IOException {
    //1. 获Get对象
    Get get = new Get(Bytes.toBytes("003"));
    //2. 通过table获取结果对象
    Result result = table.get(get);
    //3. 获取指定列簇的map集合
    NavigableMap<byte[], byte[]> base_info =
    result.getFamilyMap(Bytes.toBytes("base_info"));
    //4. 遍历map
    for(Map.Entry<byte[], byte[]> entry : base_info.entrySet()) {
        String k = new String(entry.getKey());
        String v = new String(entry.getValue());
        System.out.println(k + "=" + v);
    }
}

```

```
}  
}
```

- Get查询：获取rowkey对应的所有的列簇

```
/**  
 * get查询数据  
 */  
@Test  
public void getData2() throws IOException {  
    //1. 获Get对象  
    Get get = new Get(Bytes.toBytes("003"));  
    //2. 通过table获取结果对象  
    Result result = table.get(get);  
    //3. 获取表格扫描器  
    CellScanner cellScanner = result.cellScanner();  
    System.out.println("rowkey : " + result.getRow());  
    //4. 遍历  
    while (cellScanner.advance()) {  
        //5. 获取当前表格  
        Cell cell = cellScanner.current();  
        //5.1 获取所有的列簇  
        byte[] familyArray = cell.getFamilyArray();  
        System.out.println(new String(familyArray, cell.getFamilyOffset(),  
cell.getFamilyLength()));  
        //5.2 获取所有列  
        byte[] qualifierArray = cell.getQualifierArray();  
        System.out.println(new String(qualifierArray, cell.getQualifierOffset(),  
cell.getQualifierLength()));  
        //5.3 获取所有的值  
        byte[] valueArray = cell.getValueArray();  
        System.out.println(new String(valueArray, cell.getValueOffset(),  
cell.getValueLength()));  
    }  
}
```

- Get查询：第三种打印列的方式

```
/**  
 * get查询数据  
 */  
@Test  
public void getData2() throws IOException {  
    //1. 获Get对象  
    Get get = new Get(Bytes.toBytes("003"));  
    //2. 通过table获取结果对象  
    Result result = table.get(get);  
    //3. 获取表格扫描器  
    CellScanner cellScanner = result.cellScanner();  
    System.out.println("rowkey : " + result.getRow());  
    //4. 遍历  
    while (cellScanner.advance()) {  
        //5. 获取当前表格  
        Cell cell = cellScanner.current();  
        //5.1 获取所有的列簇  
        System.out.println(new String(CellUtil.cloneFamily(cell), "utf-8"));  
    }  
}
```

```

        System.out.println(new String(CellUtil.cloneQualifier(cell), "utf-8"));
        System.out.println(new String(CellUtil.cloneValue(cell), "utf-8"));
    }
}

```

- 批量Get查询

```

/**
 * get查询数据
 */
@Test
public void batchGetData() throws IOException {
    //1. 创建集合存储get对象
    List<Get> gets = new ArrayList<>();
    //2. 创建多个get对象
    Get get = new Get(Bytes.toBytes("001"));
    get.addColumn(Bytes.toBytes("extra_info"), Bytes.toBytes("height"));
    get.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("sex"));

    Get get1 = new Get(Bytes.toBytes("002"));
    get1.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("name"));
    get1.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("age"));

    Get get2 = new Get(Bytes.toBytes("003"));
    get2.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("sex"));
    get2.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("age"));

    Get get3 = new Get(Bytes.toBytes("004"));
    //3. 添加get对象到集合中
    gets.add(get);
    gets.add(get1);
    gets.add(get2);
    gets.add(get3);

    //4. 将集合对象添加到表中
    Result[] results = table.get(gets);

    //5. 遍历
    for(Result result : results) {
        HBaseUtils.showResult(result);
    }
}

```

- 修改HBaseUtils

```

public static void showResult(Result result){
    CellScanner cellScanner = result.cellScanner();
    System.out.print("rowKey: " + Bytes.toString(result.getRow()));
    try {
        while (cellScanner.advance()){
            Cell current = cellScanner.current();

            System.out.print("\t" + new
String(CellUtil.cloneFamily(current), "utf-8"));
            System.out.print(" : " + new
String(CellUtil.cloneQualifier(current), "utf-8"));

```

```

        System.out.print("\t" + new
String(CellUtil.cloneValue(current),"utf-8"));
    }
} catch (UnsupportedEncodingException e) {
    logger.error("判断是否有下一个单元格失败!",e);
} catch (IOException e) {
    logger.error("克隆数据失败!",e);
}
}
}

```

- Scan查询：扫描指定的表

```

/**
 * 扫描表
 */
@Test
public void scanTable() throws IOException {
    //1. 创建扫描器
    Scan scan = new Scan();
    //2. 添加扫描的行数包头不包尾
    scan.setStartRow(Bytes.toBytes("001"));
    scan.setStopRow(Bytes.toBytes("006" + "\001")); //小技巧
    //3. 添加扫描的列
    scan.addColumn(Bytes.toBytes("base_info"),Bytes.toBytes("name"));
    //4. 获取扫描器
    ResultScanner scanner = table.getScanner(scan);
    Iterator<Result> it = scanner.iterator();
    while (it.hasNext()){
        Result result = it.next();
        HBaseUtils.showResult(result);
    }
}

```

- 删除数据

```

/**
 * 删除数据
 */
@Test
public void deleteData() throws IOException {
    //1. 创建集合用于批量删除
    List<Delete> dels = new ArrayList<>();
    //2. 创建删除数据对象
    Delete del = new Delete(Bytes.toBytes("004"));
    del.addColumn(Bytes.toBytes("base_info"),Bytes.toBytes("name"));
    //3. 添加到集合
    dels.add(del);
    //4. 提交
    table.delete(dels);
}

```

## 2.6.7 Hbase的过滤器

### 2.6.7.1 SingleColumnValueFilter

```

/**
 * 高级查询
 * 过滤器链查询
 */
public class Demo6_Filter {
    /**
     * 需求:
     * select * from ns1_userinfo where age <= 18 and name = narudo
     */
    @Test
    public void listFilter() throws IOException {
        /**
         * 1. 创建过滤器链
         * and条件使用MUST_PASS_ALL作为条件
         * or条件使用MUST_PASS_ONE
         */
        FilterList filterList = new
FilterList(FilterList.Operator.MUST_PASS_ALL);
        /**
         * 2. 构造查询条件
         * 对于单列值比较器使用SingleColumnValueFilter
         * ColumnFamily, qualifier, 比较过滤器（大于、小于、等于、...），value
         */
        SingleColumnValueFilter ageFilter = new
SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("age"),
        CompareFilter.CompareOp.LESS_OR_EQUAL, Bytes.toBytes("18"));
        SingleColumnValueFilter nameFilter = new
SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
        CompareFilter.CompareOp.EQUAL, Bytes.toBytes("narudo"));
        /**
         * 3. 将条件加入到过滤器链中
         */
        filterList.addFilter(ageFilter);
        filterList.addFilter(nameFilter);

        //4. 创建扫描器进行扫描
        Scan scan = new Scan();

        //5. 将过滤条件关联到扫描器
        scan.setFilter(filterList);

        //6. 获取表对象
        Table table = HBaseUtils.getTable();

        //7. 扫描表
        ResultScanner scanner = table.getScanner(scan);

        //8. 打印数据
        Iterator<Result> iterator = scanner.iterator();
        while (iterator.hasNext()) {
            Result result = iterator.next();
            HBaseUtils.showResult(result);
        }
    }
}

```

- 说明在查询的时候

1. 这种查询如果对应的行键，如果都包含了age和name列，就会对比值，如果都不满足就将其过滤，如果满足就获取。
2. 如果包含其中也给值，name或者age，只要其中一个列满足条件就会，就会获取。
3. 如果name和age都没有，那么视为该行满足条件

- 解决方案

在某个单值过滤器总添加

```
ageFilter.setFilterIfMissing(true);
nameFilter.setFilterIfMissing(true);
```

## 2.6.7.2 比较器

- RegexStringComparator

```
/**
 *
 * 测试正则比较器
 * 需求: select * from ns1_userinfo where name like '%i%'
 */
@Test
public void regexStringComparatorTest() throws IOException {
    //1. 创建比较器，正则：以li开头的
    RegexStringComparator regexStringComparator = new
    RegexStringComparator("[a-z]i[a-z]");
    //2. 获取单列值过滤器
    SingleColumnValueFilter nameFilter = new
    SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
        CompareFilter.CompareOp.EQUAL, regexStringComparator);
    //3. 设置缺失
    nameFilter.setFilterIfMissing(true);
    //4. 创建扫描器进行扫描
    Scan scan = new Scan();
    //5. 设置过滤器
    scan.setFilter(nameFilter);
    //6. 获取表对象
    Table table = HBaseUtils.getTable();
    //7. 扫描表
    ResultScanner scanner = table.getScanner(scan);
    //8. 打印数据
    Iterator<Result> iterator = scanner.iterator();
    while (iterator.hasNext()) {
        Result result = iterator.next();
        HBaseUtils.showResult(result);
    }
}
```

- 封装HbaseUtils

```
/**
 * 显示这个过滤器扫描的对象
 */
public static void showFilterResult(Filter filter) {
    //4. 创建扫描器进行扫描
    Scan scan = new Scan();
```



```

//5. 设置过滤器
scan.setFilter(filter);
//6. 获取表对象
Table table = HBaseUtils.getTable();
//7. 扫描表
ResultScanner scanner = null;
try {
    scanner = table.getScanner(scan);
    //8. 打印数据
    Iterator<Result> iterator = scanner.iterator();
    while (iterator.hasNext()) {
        Result result = iterator.next();
        HBaseUtils.showResult(result);
    }
} catch (IOException e) {
    logger.warn("获取table的时候异常!", e);
} finally {
    try {
        table.close();
    } catch (IOException e) {
        logger.warn("关闭table的时候异常!", e);
    }
}
}

```

- 修改之前的方法

```

/**
 *
 * 测试正则比较器
 * 需求: select * from ns1_userinfo where name like '%i%'
 */
@Test
public void regexStringComparatorTest() throws IOException {
    //1. 创建比较器, 正则: 以i开头的
    RegexStringComparator regexStringComparator = new RegexStringComparator("[a-z]i[a-z]");
    //2. 获取单列值过滤器
    SingleColumnValueFilter nameFilter = new
    SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
    CompareFilter.CompareOp.EQUAL, regexStringComparator);
    //3. 设置缺失
    nameFilter.setFilterIfMissing(true);
    //4. 打印
    HBaseUtils.showFilterResult(nameFilter);
}

```

- subStringComparator

```

/**
 *
 * 测试subString比较器
 * 需求: select * from ns1_userinfo where name like '%i%'
 */
@Test
public void subStringComparatorTest() throws IOException {

```

```

//1. 创建比较器，正则：以li开头的
SubstringComparator substringComparator = new SubstringComparator("i");
//2. 获取单列值过滤器
SingleColumnValueFilter nameFilter = new
SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
CompareFilter.CompareOp.EQUAL, substringComparator);
//3. 设置缺失
nameFilter.setFilterIfMissing(true);
//4. 打印
HBaseUtils.showFilterResult(nameFilter);
}

```

- BinaryComparator

```

/**
 *
 * 测试二进制比较器
 * 需求: select * from ns1_userinfo where name = 'lixi'
 */
@Test
public void binaryComparatorTest() throws IOException {
//1. 创建比较器，正则：以li开头的
BinaryComparator binaryComparator = new
BinaryComparator(Bytes.toBytes("lixi"));
//2. 获取单列值过滤器
SingleColumnValueFilter nameFilter = new
SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
CompareFilter.CompareOp.EQUAL, binaryComparator);
//3. 设置缺失
nameFilter.setFilterIfMissing(true);
//4. 打印
HBaseUtils.showFilterResult(nameFilter);
}

```

- BinaryPrefixComparator

```

/**
 *
 * 测试二进制前缀比较器
 * 需求: select * from ns1_userinfo where name like 'li%'
 */
@Test
public void binaryPrefixComparatorTest() throws IOException {
//1. 创建比较器，正则：以li开头的
BinaryPrefixComparator binaryPrefixComparator = new
BinaryPrefixComparator(Bytes.toBytes("li"));
//2. 获取单列值过滤器
SingleColumnValueFilter nameFilter = new
SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
CompareFilter.CompareOp.EQUAL, binaryPrefixComparator);
//3. 设置缺失
nameFilter.setFilterIfMissing(true);
//4. 打印
HBaseUtils.showFilterResult(nameFilter);
}

```

### 2.6.7.3 KeyValue Metadata

- FamilyFilter1

```
/**
 * 查询以base开头的列簇
 */
@Test
public void testColumnFamily1() {
    //1. 创建正则比较器：以base开头的字符串
    RegexStringComparator regexStringComparator = new
    RegexStringComparator("^base");
    //2. 创建FamilyFilter：结果中只包含满足条件的列簇信息
    FamilyFilter familyFilter = new FamilyFilter(CompareFilter.CompareOp.EQUAL,
    regexStringComparator);
    //3. 打印
    HBaseUtils.showFilterResult(familyFilter);
}
```

- FamilyFilter2

```
/**
 * 查询包含xtr的列簇
 */
@Test
public void testColumnFamily2() {
    //1. 创建正则比较器：以base开头的字符串
    SubstringComparator substringComparator = new SubstringComparator("xtr");
    //2. 创建FamilyFilter
    FamilyFilter familyFilter = new FamilyFilter(CompareFilter.CompareOp.EQUAL,
    substringComparator);
    //3. 打印
    HBaseUtils.showFilterResult(familyFilter);
}
```

- QualifierFilter

```
/**
 * 查询包含xtr的列簇
 */
@Test
public void testQualifierFilter() {
    //1. 创建正则比较器：以base开头的字符串
    SubstringComparator substringComparator = new SubstringComparator("am");
    //2. 创建FamilyFilter
    QualifierFilter qualifierFilter = new
    QualifierFilter(CompareFilter.CompareOp.EQUAL, substringComparator);
    //3. 打印
    HBaseUtils.showFilterResult(qualifierFilter);
}
```

- ColumnPrefixFilter

```

/**
 * 查询包含xtr的列簇
 */
@Test
public void testColumnPrefixFilter() {
    //1. 创建ColumnPrefixFilter
    ColumnPrefixFilter columnPrefixFilter = new
    ColumnPrefixFilter(Bytes.toBytes("a"));
    //2. 打印
    HBaseUtils.showFilterResult(columnPrefixFilter);
}

```

- MultipleColumnPrefixFilter

```

/**
 * 查找以“a”或“n”开头的行和列族中的所有列
 */
@Test
public void testMultipleColumnPrefixFilter() {
    //1. 创建ColumnPrefixFilter
    byte[][] prefixes = new byte[][] {Bytes.toBytes("a"), Bytes.toBytes("n")};
    MultipleColumnPrefixFilter multipleColumnPrefixFilter = new
    MultipleColumnPrefixFilter(prefixes);
    //2. 打印
    HBaseUtils.showFilterResult(multipleColumnPrefixFilter);
}

```

- ColumnRangeFilter

```

/**
 * 查找以“age”到“name”的列的信息
 * minColumnInclusive:true为包含, false为不包含
 */
@Test
public void testColumnRangeFilter() {
    //1. 创建ColumnPrefixFilter
    ColumnRangeFilter columnRangeFilter = new
    ColumnRangeFilter(Bytes.toBytes("age"), false,
    Bytes.toBytes("name"), true);
    //2. 打印
    HBaseUtils.showFilterResult(columnRangeFilter);
}

```

#### 2.6.7.4 RowKey

- RowFilter

```

/**
 * 查找rowkey=002的信息
 */
@Test
public void testRowFilter() {
    //1. 创建RowFilter
    BinaryComparator binaryComparator = new
    BinaryComparator(Bytes.toBytes("002"));
    RowFilter rowFilter = new RowFilter(CompareFilter.CompareOp.EQUAL,
    binaryComparator);
    //2. 打印
    HBaseUtils.showFilterResult(rowFilter);
}

```

### 2.6.7.5 Utility

- FirstKeyOnlyFilter

```

/**
 * 查找指定表中的所有的行键的第一个列
 */
@Test
public void testFirstKeyOnlyFilter() {
    //1. 创建RowFilter
    FirstKeyOnlyFilter firstKeyOnlyFilter = new FirstKeyOnlyFilter();
    //2. 打印
    HBaseUtils.showFilterResult(firstKeyOnlyFilter);
}

```

- 最后举例

```

/**
 * 需求: select * from ns1_userinfo where age <= 18 or name = lixi
 */
@Test
public void listFilter2() {
    //1.
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ONE);
    SingleColumnValueFilter ageFilter = new
    SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("age"),
    CompareFilter.CompareOp.LESS_OR_EQUAL, Bytes.toBytes("18"));
    SingleColumnValueFilter nameFilter = new
    SingleColumnValueFilter(Bytes.toBytes("base_info"), Bytes.toBytes("name"),
    CompareFilter.CompareOp.EQUAL, Bytes.toBytes("lixi"));
    ageFilter.setFilterIfMissing(true);
    nameFilter.setFilterIfMissing(true);
    filterList.addFilter(ageFilter);
    filterList.addFilter(nameFilter);
    List<Filter> filters = filterList.getFilters();
    for(Filter filter : filters) {
        HBaseUtils.showFilterResult(filter);
    }
}

```

### 2.6.7.6 PageFilter

```

/**
 * 需求：每行显示3条记录
 * 将全部的数据分页显示出来
 *
 * 思路：
 * 1. 第一页：
 * select * from user_info where rowkey > \001 limit 3;
 * 2. 其他页
 * select * from user_info where rowkey > 第一页的maxrowkey limit 3;
 * 3. 循环什么时候结束？
 * while(true) {
 *     select * from user_info where rowkey > 第一页的maxrowkey limit 3;
 *     print 3行数据
 *     结束条件：count<3
 * }
 */
public class Demo7_PageFilter {

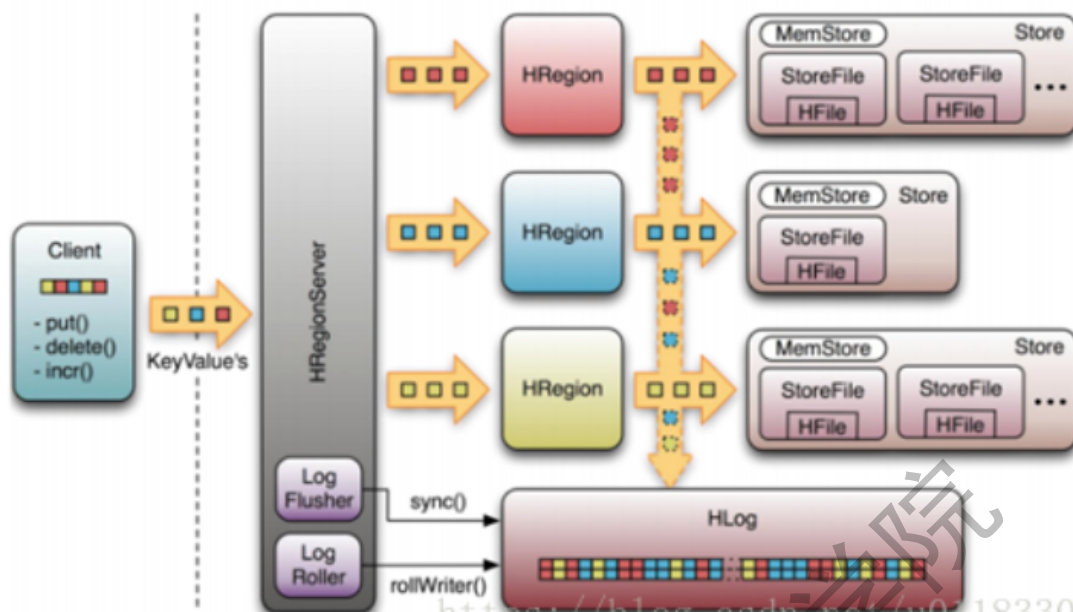
    /**
     * 测试分页显示user_info表中的所有数据，分页显示为3行记录
     */
    @Test
    public void testPageFilter() throws IOException {
        //1. 创建分页过滤器，并设置每页显示3条记录
        PageFilter pageFilter = new PageFilter(3);
        //2. 构造扫描器
        Scan scan = new Scan();
        //3. 给扫描器设置过滤器
        scan.setFilter(pageFilter);
        //4. 获取表的管理器
        Table table = HBaseUtils.getTable();
        //5. 遍历显示
        String maxKey = ""; // 最大key值记录器
        while(true) {
            int count = 0; // 计数器
            //6. 获取结构扫描器
            ResultScanner scanner = table.getScanner(scan);
            //7. 获取迭代器迭代
            Iterator<Result> iterator = scanner.iterator();
            //8. 迭代
            while (iterator.hasNext()) {
                Result result = iterator.next();
                System.out.println(new String(result.getRow()));
                count++;
                maxKey = Bytes.toString(result.getRow());
                //9. 打印
                HBaseUtils.showResult(result);
            }
            System.out.println("-----");
            //10. 判断是否可以结束
            if (count < 3) break;

            //11. 设置下一次开始查询的行键号
            scan.setStartRow(Bytes.toBytes(maxKey + "\001"));
        }
    }
}

```

## 2.6.8 Hbase的读写流程

### 2.6.8.1 写数据流程



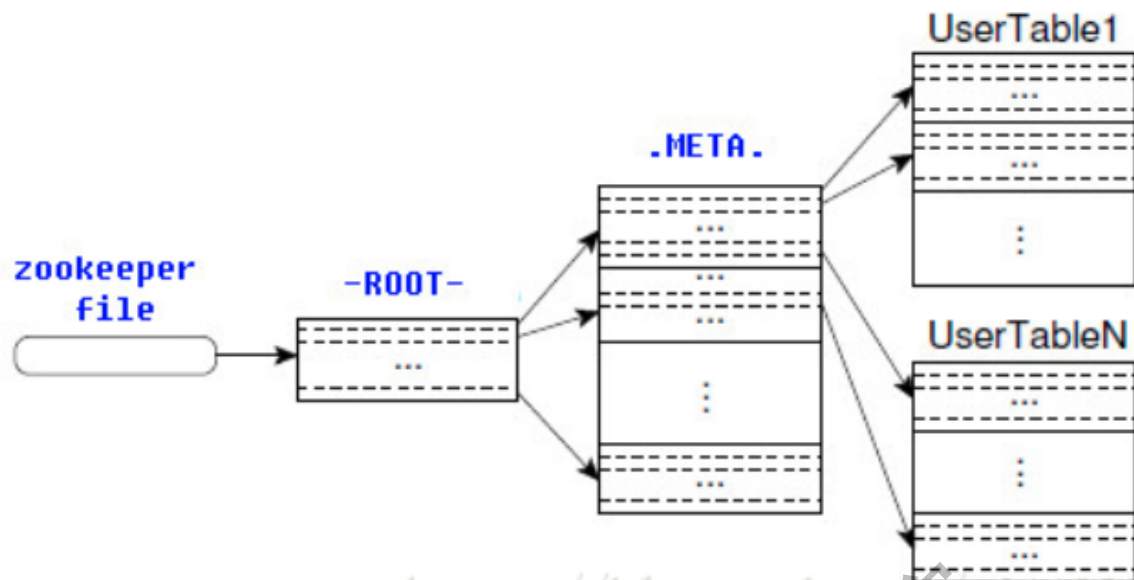
Hbase使用memstore和storefile存储对表的更新。数据在更新时首先写入hlog和memstore，memstore中的数据是排序的，当memstore累计到一定的阈值时，就会创建一个新的memstore，并将老的memstore添加到flush队列，由单独的线程flush到磁盘上，成为一个filestore。与此同时，系统会在zookeeper中记录一个checkpoint，表示这个时刻之前的数据变更已经持久化了。当系统出现意外时，可能导致memstore中的数据丢失，此时使用hlog来恢复checkpoint之后的数据。

Storefile是只读的，一旦创建之后就不可修改。因此hbase的更新就是不断追加的操作。当一个store的storefile达到一定的阈值后，就会进行一次合并操作，将对同一个key的修改合并到一起，同时进行版本合并和数据删除，形成一个大的storefile。当storefile的大小达到一定的阈值后，又会对storefile进行切分操作，等分为两个storefile。

Hbase中只有增添数据，所有的更新和删除操作都是在后续的合并中进行的，使得用户的写操作只要进入内存就可以立刻返回，实现了hbase的高速存储。

- (1) Client通过Zookeeper的调度，向RegionServer发出写数据请求，在Region中写数据。
- (2) 数据被写入Region的MemStore，直到MemStore达到预设阈值。
- (3) MemStore中的数据被Flush成一个StoreFile。
- (4) 随着StoreFile文件的不断增多，当其数量增长到一定阈值后，触发Compact合并操作，将多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除。
- (5) StoreFiles通过不断的Compact合并操作，逐步形成越来越大的StoreFile。
- (6) 单个StoreFile大小超过一定阈值后，触发Split操作，把当前Region Split成2个新的Region。父Region会下线，新Split出的2个子Region会被HMaster分配到相应的RegionServer上，使得原先1个Region的压力得以分流到2个Region上。

### 2.6.8.2 读数据流程



Hbase的所有region元数据被存储在.META表中，随着region的增多，.META表中的数据也会增大，并分裂成多个新的region。为了定位.META表中各个region的位置，把.META表中的所有region的元数据保存在-ROOT-表中，最后由zookeeper记录-ROOT-表的位置信息。所有的客户端访问数据之前，需要首先访问zookeeper获取-ROOT-表的位置，然后访问-ROOT-表获得.META表的位置，最后根据.META表中的信息确定用户数据存放的位置。

-ROOT-表永远不会被分割，它只有一个region，这样可以保证最多只需要三次跳转就可以定位任意一个region。为了加快访问速度，.META表的所有region全部保存在内存中。客户端会将查询过的位置信息缓存起来，且缓存不会主动失效。如果客户端根据缓存信息还访问不到数据，则询问相关.META表的region服务器，试图获取数据的位置，如果还是失败，则询问-ROOT-表相关的.META表在哪里。最后，如果前面的信息全部失效，则通过zookeeper重新定位region的信息。所以如果客户端上的缓存全部失效，则需要进行6次网络来定位，才能定位到正确的region。

client-->Zookeeper-->-ROOT-表-->.META.表-->RegionServer-->Region-->client

- (1) Client访问Zookeeper，查找-ROOT-表，获取.META.表信息。
- (2) 从.META.表查找，获取存放目标数据的Region信息，从而找到对应的RegionServer。
- (3) 通过RegionServer获取需要查找的数据。
- (4) Regionserver的内存分为MemStore和BlockCache两部分，MemStore主要用于写数据，

BlockCache主要用于读数据。读请求先到MemStore中查数据，查不到就到BlockCache中查，再查不到就会到StoreFile上读，并把读的结果放入BlockCache。

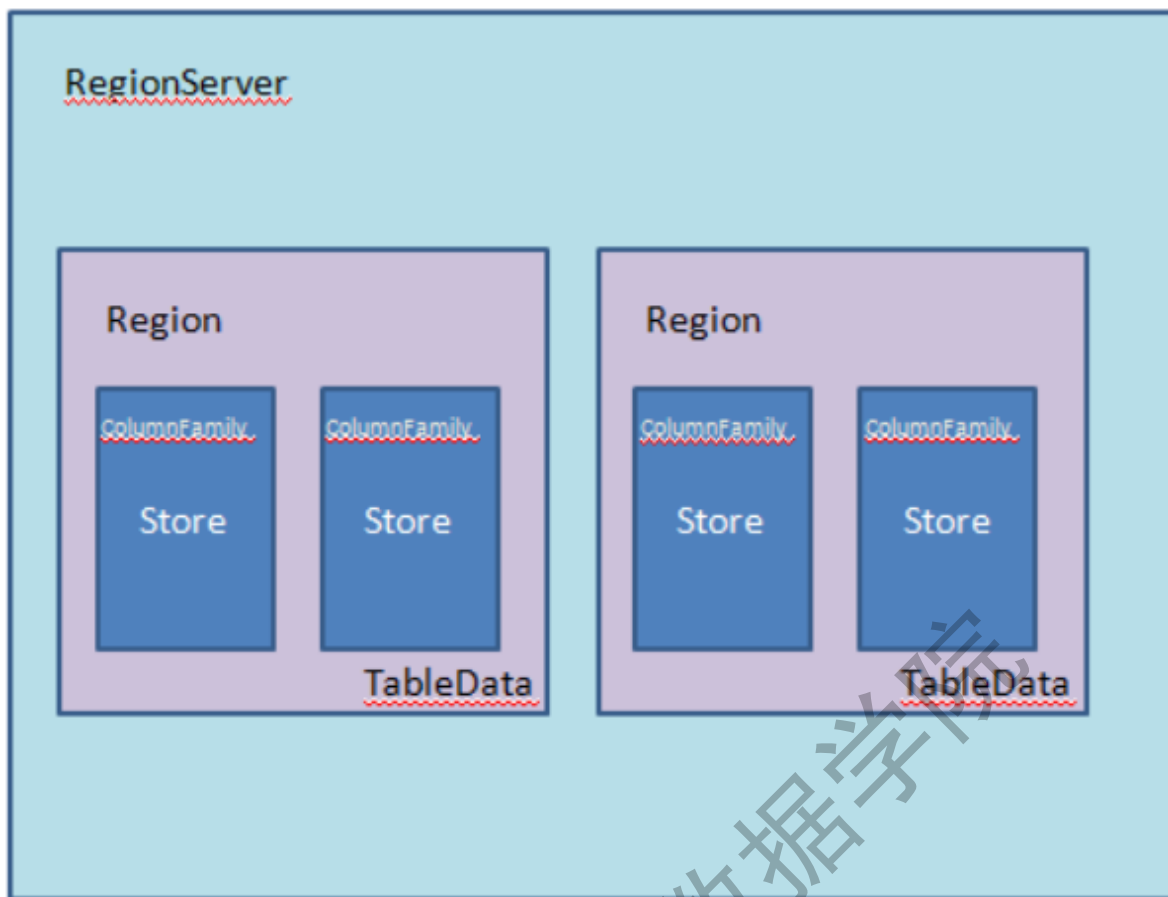
## 2.6.9 Hbase的存储机制

数据的存储是每个Region所承担的工作了

### 2.6.9.1 存储模型

- 我们知道一个Region代表的是一张 Hbase表中特定Rowkey范围内的数据，
- 而Hbase是面向列存储的数据库，所以在一个Region中，有多个文件来存储这些列。
- Hbase中数据列是由列簇来组织的，所以每一个列簇都会有对应的一个数据结构，
  - \* Hbase将列簇的存储数据结构抽象为Store，一个Store代表一个列簇。





所以在这里也可以看出为什么在我们查询的时候要尽量减少不需要的列，而经常一起查询的列要组织到一个列簇里：因为需要查询的列簇越多，意味着要扫描越多的Store文件，这就需要越多的时间

- Store中存储数据的方式:Hbase采用的是LSM树的结构，这种结构的关键是，

1. 每一次的插入操作都会先进入MemStore（内存缓冲区），
2. 当 MemStore达到上限的时候，Hbase会将内存中的数据输出为有序的StoreFile文件数据（根据Rowkey、版本、列名排序，这里已经和列簇无关了因为Store里都属于同一个列簇）。
3. 这样会在Store中形成很多个小的StoreFile，当这些小的File数量达到一个阈值的时候，Hbase会用一个线程来把这些小File合并成一个大的File。这样，Hbase就把效率低下的文件中的插入、移动操作转变成了单纯的文件输出、合并操作。

由上可知，在Hbase底层的Store数据结构中，

- 1) 每个StoreFile内的数据是有序的，
- 2) 但是StoreFile之间不一定是有序的，
- 3) Store只需要管理StoreFile的索引就可以了。

这里也可以看出为什么指定版本和Rowkey可以加强查询的效率，因为指定版本和Rowkey的查询可以利用StoreFile的索引跳过一些肯定不包含目标数据的数据。



## 2.6.9.2 布隆过滤器

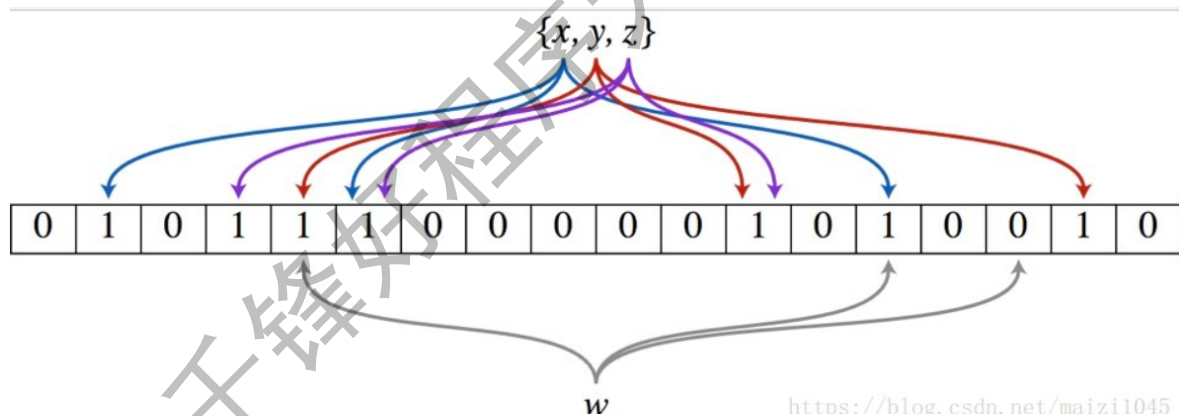
- 说明

Bloom filter 是由 Howard Bloom 在 1970 年提出的二进制向量数据结构，它具有很好的空间和时间效率，被用来检测一个元素是不是集合中的一个成员。如果检测结果为是，该元素不一定在集合中；但如果检测结果为否，该元素一定不在集合中。因此 Bloom filter 具有 100% 的召回率。这样每个检测请求返回有“在集合内（可能错误）”和“不在集合内（绝对不在集合内）”两种情况，可见 Bloom filter 是牺牲了正确率以节省空间。

- 原理

它的时间复杂度是  $O(1)$ ，但是空间占用取决于其优化的方式。它是布隆过滤器的基础。

布隆过滤器 (Bloom Filter) 的核心实现是一个超大的位数组（或者叫位向量）和几个哈希函数。假设位数组的长度为  $m$ ，哈希函数的个数为  $k$



以上图为例，具体的插入数据和校验是否存在的流程：

假设集合里面有 3 个元素  $\{x, y, z\}$ ，哈希函数的个数为 3。

**Step1:** 将位数组初始化，每位都设置为 0。

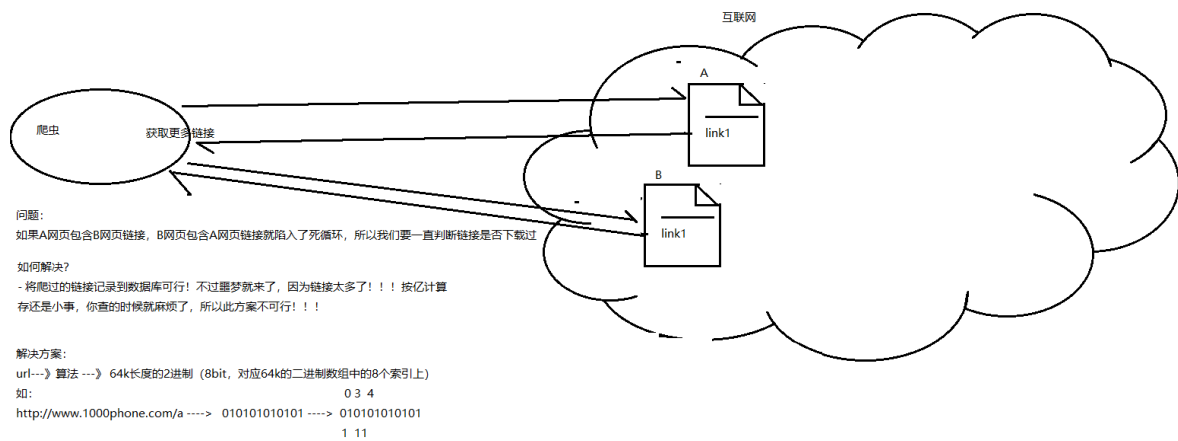
**Step2:** 对于集合里面的每一个元素，将元素依次通过 3 个哈希函数进行映射，每次映射都会产生一个哈希值，哈希值对应位数组上面的一个点，将该位置标记为 1。

**Step3:** 查询  $w$  元素是否存在集合中的时候，同样的方法将  $w$  通过哈希映射到位数组上的 3 个点。

**Step4:** 如果 3 个点的其中有一个点不为 1，则可以判断该元素一定不存在集合中。反之，如果 3 个点都为 1，则该元素可能存在集合中。注意：此处不能判断该元素是否一定存在集合中，可能存在一定的误判率。

可以从图中可以看到：假设某个元素通过映射对应下标为 4, 5, 6 这 3 个点。虽然这 3 个点都为 1，但是很明显这 3 个点是不同元素经过哈希得到的位置，因此这种情况说明元素虽然不在集合中，也可能对应的都是 1，这是误判率存在的原因。

- 布隆过滤器的应用场景1：爬虫问题



- 在hbase中布隆过滤器的原理

布隆过滤器是hbase中的高级功能, 它能够减少特定访问模式 (get/scan) 下的查询时间。不过由于这种模式增加了内存和存储的负担, 所以被默认为关闭状态。

hbase支持如下类型的布隆过滤器:

- 1、NONE 不使用布隆过滤器
- 2、ROW 行键使用布隆过滤器
- 3、ROWCOL 列键使用布隆过滤器

其中ROWCOL是粒度更细的模式。

- 原因

在介绍为什么hbase要引入布隆过滤器之前, 我们先来了解一下hbase存储文件HFile的块索引机制。我们知道hbase的实际存储结构是HFile, 它是位于hdfs系统中的, 也就是在磁盘中。而加载到内存中的数据存储在MemStore中, 当MemStore中的数据达到一定数量时, 它会将数据存入HFile中。

HFile是由一个个数据块与索引块组成, 他们通常默认为64KB。hbase是通过块索引来访问这些数据块的。而索引是由每个数据块的第一行数据的rowkey组成的。当hbase打开一个HFile时, 块索引信息会优先加载到内存当中。然后hbase会通过这些块索引来查询数据。

但是块索引是相当粗粒度的, 我们可以简单计算一下。假设一个行占100bytes的空间, 所以一个数据块64KB, 所包含的行大概有:  $(64 * 1024) / 100 = 655.53 = \sim 700$ 行。而我们只能从索引给出的一个数据块的起始行开始查询。

如果用户随机查找一个行键, 则这个行键很可能位于两个开始键 (即索引) 之间的位置。对于hbase来说, 它判断这个行键是否真实存在的唯一方法就是加载这个数据块, 并且扫描它是否包含这个键。

同时, 还存在很多情况使得这种情况更加复杂。

对于一个应用来说, 用户通常会以一定的速率进行更新数据, 这就将导致内存中的数据被刷写到磁盘中, 并且之后系统会将他们合并成更大的存储文件。在hbase的合并存储文件的时候, 它仅仅会合并最近几个存储文件, 直至合并的存储文件到达配置的最大大小。最终系统中会有很多的存储文件, 所有的存储文件都是候选文件, 其可能包含用户请求行键的单元格。如下图所示:

我们可以看到, 这些不同的文件都来自同一个列族, 所以他们的行键分布类似。所以, 虽然我们要查询更新的特定行只在某个或者某几个文件中, 但是采用块索引方式, 还是会覆盖整个行键范围。当块索引确定那些块可能含有某个行键后, regionServer需要加载每一个块来检查该块中是否真的包含该行的单元格。

- 布隆过滤器的作用

当我们随机读get数据时，如果采用hbase的块索引机制，hbase会加载很多块文件。如果采用布隆过滤器后，它能够准确判断该HFile的所有数据块中，是否含有我们查询的数据，从而大大减少不必要的块加载，从而增加hbase集群的吞吐率。这里有几点细节：

1. 布隆过滤器的存储在哪？

对于hbase而言，当我们选择采用布隆过滤器之后，HBase会在生成StoreFile（HFile）时包含一份布隆过滤器结构的数据，称其为MetaBlock；MetaBlock与DataBlock（真实的KeyValue数据）一起由LRUBlockCache维护。所以，开启bloomfilter会有一定的存储及内存cache开销。但是在大多数情况下，这些负担相对于布隆过滤器带来的好处是可以接受的。

2. 采用布隆过滤器后，hbase如何get数据？

在读取数据时，hbase会首先在布隆过滤器中查询，根据布隆过滤器的结果，再在MemStore中查询，最后再在对应的HFile中查询。

3. 采用ROW还是ROWCOL布隆过滤器？

这取决于用户的使用模式。如果用户只做行扫描，使用更加细粒度的行加列布隆过滤器不会有任何的帮助，这种场景就应该使用行级布隆过滤器。当用户不能批量更新特定的一行，并且最后的使用存储文件都含有改行的一部分时，行加列级的布隆过滤器更加有用。

例如：ROW 使用场景假设有2个Hfile文件hf1和hf2， hf1包含kv1（r1 cf:q1 v）、kv2（r2 cf:q1 v） hf2包含kv3（r3 cf:q1 v）、kv4（r4 cf:q1 v） 如果设置了CF属性中的bloomfilter（布隆过滤器）为ROW，那么get(r1)时就会过滤hf2，get(r3)就会过滤hf1。

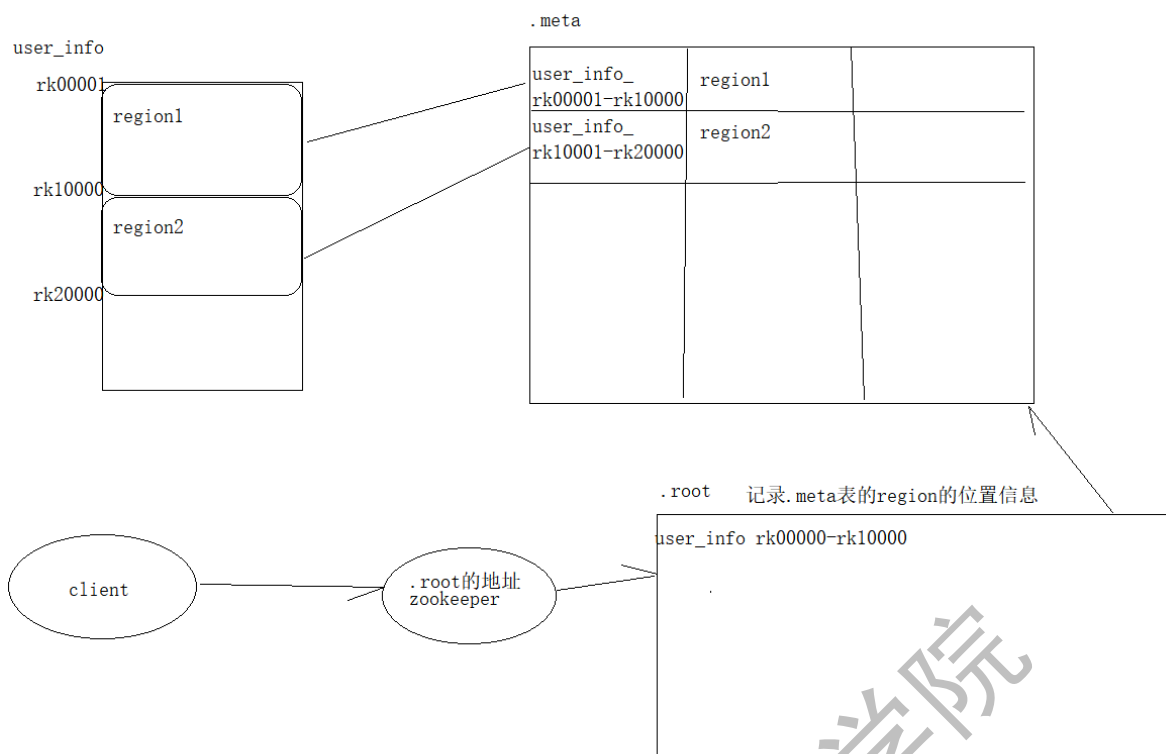
ROWCOL使用场景假设有2个Hfile文件hf1和hf2， hf1包含kv1（r1 cf:q1 v）、kv2（r2 cf:q1 v） hf2包含kv3（r1 cf:q2 v）、kv4（r2 cf:q2 v） 如果设置了CF属性中的bloomfilter为ROW，无论get(r1,q1)还是get(r1,q2)，都会读取hf1+hf2；而如果设置了CF属性中的bloomfilter为ROWCOL，那么get(r1,q1)就会过滤hf2，get(r1,q2)就会过滤hf1。

tip:

ROW和ROWCOL只是名字上有联系，但是ROWCOL并不是ROW的扩展，也不能取代ROW

## 2.6.10 Hbase的寻址机制

### 2.6.10.1 寻址示意图



## 2.6.10.2 root和meta表结构

- root

RowKey	info			history
	regioninfo	server	server startcode	
Tablename, startRowkey, Times tamp	StaryKey, EndKey, FamilyList,	address		

- meta

RowKey	info			history
	regioninfo	server	server startcode	
Table1 RK0, 10000		rsl		

### 2.6.10.3 寻址流程

现在假设我们要从Table2里面查询一条RowKey是RK10000的数据。那么我们应该遵循以下步骤：

1. 从.META.表里面查询哪个Region包含这条数据。
2. 获取管理这个Region的RegionServer地址。
3. 连接这个RegionServer，查到这条数据。

系统如何找到某个row key（或者某个 row key range）所在的region

**bigtable** 使用三层类似B+树的结构来保存region位置。

第一层：保存zookeeper里面的文件，它持有root region的位置。

第二层：root region是.META.表的第一个region其中保存了.META.表其它region的位置。通过root region，我们就可以访问.META.表的数据。

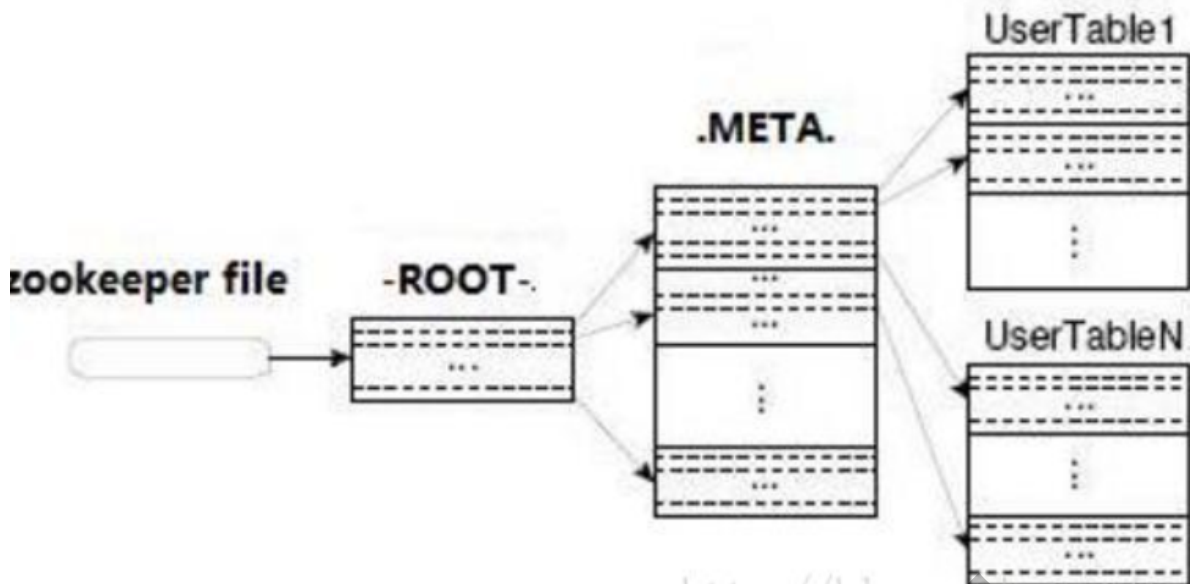
第三层：.META.表它是一个特殊的表，保存了hbase中所有数据表的region 位置信息。

说明：

- (1) root region永远不会被split，保证了最需要三次跳转，就能定位到任意region。
- (2) .META.表每行保存一个region的位置信息，row key 采用表名+表的最后一行编码而成。
- (3) 为了加快访问，.META.表的全部region都保存在内存中。
- (4) client会将查询过的位置信息缓存起来，缓存不会主动失效，因此如果client上的缓存全部失效，则需要进行最多6次网络来回，才能定位到正确的region(其中三次用来发现缓存失效，另外三次用来获取位置信息)。

### 2.6.10.4 总结

- Region定位流程



寻找RegionServer

Zookeeper-> -ROOT-(单Region)-> .META.-> 用户表

-ROOT-表

表包含.META.表所在的region列表，该表只会有一个Region；

Zookeeper中记录了-ROOT-表的location。

.META.表

表包含所有的用户空间region列表，以及RegionServer的服务器地址。

## 2.6.11 Mapreduce和Hbase的整合

### 2.6.11.1 HBase2HDFS

- 添加依赖

```

<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.8.1</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>2.8.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>2.8.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>1.2.1</version>
  </dependency>
</dependencies>
  
```

```

<!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase-client --
>

<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-common</artifactId>
    <version>1.2.1</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase-client --
>

<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>1.2.1</version>
</dependency>
</dependencies>

```

- HBase2HDFS

```

/**
 * 将HBase中的数据清洗到HDFS中
 * Mapreduce
 */
public class Demo8_HBase2HDFS implements Tool {

    //1. 创建配置对象
    private Configuration configuration;
    private final static String HBASE_CONNECT_KEY = "hbase.zookeeper.quorum";
    private final static String HBASE_CONNECT_VALUE =
        "hbase2:2181,hbase3:2181,hbase4:2181";
    private final static String HDFS_CONNECT_KEY = "fs.defaultFS";
    private final static String HDFS_CONNECT_VALUE = "hdfs://hbase1:9000";
    private final static String MAPREDUCE_CONNECT_KEY =
        "mapreduce.framework.name";
    private final static String MAPREDUCE_CONNECT_VALUE = "yarn";

    @Override
    public int run(String[] args) throws Exception {
        //1. 获取job
        Job job = Job.getInstance(configuration, "hbase2hdfs");
        //2. 设置运行jar
        job.setJarByClass(Demo8_HBase2HDFS.class);
        /*
         * 3. 设置TableMapper初始参数
         * 设置从HBase表中读取数据作为输入
         * 表名tablename, 扫描器scan,mapper类, mapper输出key的类, mapper输出的value
         类, jo
        */
        //
        job.setMapperClass(HBaseMapper.class);
        //
        job.setMapOutputKeyClass(Text.class);
        //
        job.setMapOutputValueClass(NullWritable.class);
        TableMapReduceUtil.initTableMapperJob("ns1:user_info", getScan(),
            HBaseMapper.class,
            Text.class, NullWritable.class, job);
        //4. 设置输出格式
        FileOutputFormat.setOutputPath(job, new Path(args[0]));
        /*

```



```

    * 5. 设置从HBase表中读取数据作为输入
    * 表名tablename, 扫描器scan,mapper类, mapper输出key的类, mapper输出的value
    类, job
    */
    //6. 提交
    boolean b = job.waitForCompletion(true);
    return b ? 1 : 0;
}

@Override
public void setConf(Configuration conf) {
    conf.set(HBASE_CONNECT_KEY, HBASE_CONNECT_VALUE); // 设置连接的hbase
    conf.set(HDFS_CONNECT_KEY, HDFS_CONNECT_VALUE); // 设置连接的hadoop
    conf.set(MAPREDUCE_CONNECT_KEY, MAPREDUCE_CONNECT_VALUE); // 设置使用的mr
    运行平台
    this.configuration = conf;
}

@Override
public Configuration getConf() {
    return configuration;
}

/**
 * 一、 自定义Mapper类
 * 从HBase中读取某表数据: ns1:user_info
 * 1. 读取一行
 * 003                                     column=base_info:age,
timestamp=1546957041028, value=15
 * 003                                     column=base_info:name,
timestamp=1546957041028, value=narudo
 * 003                                     column=base_info:sex,
timestamp=1546957041028, value=male
 *
 * 2. 输出
 * age:15 name:narudo sex:male*
 */
public static class HBaseMapper extends TableMapper<Text, NullWritable> {

    private Text k = new Text();

    /**
     *
     * @param key :
     * @param value : 返回根据rowkey的一行结果
     * @param context
     */
    @Override
    protected void map(ImmutableBytesWritable key, Result value, Context
    context) throws IOException, InterruptedException {
        //0. 定义字符串存放最终结果
        StringBuffer sb = new StringBuffer();
        //1. 获取扫描器进行扫描解析
        CellScanner cellScanner = value.cellScanner();
        //2. 推进
        while (cellScanner.advance()) {
            //3. 获取当前单元格

```

```

        cell cell = cellScanner.current();
        //4. 拼接字符串
        sb.append(new String(CellUtil.cloneQualifier(cell)));
        sb.append(":");
        sb.append(new String(CellUtil.cloneValue(cell)));
        sb.append("\t");
    }
    //5. 写出
    k.set(sb.toString());
    context.write(k, NullWritable.get());
}

}

public static void main(String[] args) throws Exception {
    ToolRunner.run(HBaseConfiguration.create(), new Demo8_HBase2HDFS(),
args);
}
private static Scan getScan() {
    return new Scan();
}
}
}

```

- 执行

```

[root@hbase1 home]# hadoop jar job.jar
cn.qphone.hbase.hbase2hdfs.Demo8_HBase2HDFS /hbase2hdfs/output/00

```

- 异常解决

```

Exception in thread "main" java.lang.ClassNotFoundException: cn.qphone.hbase.hbase2hdfs.Demo8_HBase2HDFS.HBaseRunner
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
    at org.apache.hadoop.util.RunJar.run(RunJar.java:227)
    at org.apache.hadoop.util.RunJar.main(RunJar.java:148)

```

1. 这里报错异常是找不到runner，但是实际原因是因为Scan这些都没有被一并打包到jar中

2. 具体可以查看打包的jar

3. 或者hadoop运行的classpath中找不到hbase中需要的jar也会产生这个错

解决方案：

1. 将jar打包的时候将相应的依赖jar包都打包进来。但是这个jar包会过大

2. export HADOOP\_CLASSPATH=\$HADOOP\_CLASSPATH:/usr/apps/hbase/hbase-1.2.1/lib/\*

如果使用第2的方式设置，在运行的地方都需要设置一次这个命令

3. 配置hadoop-env.sh

export HADOOP\_CLASSPATH=\$HADOOP\_CLASSPATH:/usr/local/hbase-1.2.1/lib/\*

同步给所有节点，最后重启hadoop集群

4. 最暴力的方式：把hbase的lib的jar包，拷贝到hadoop的classpath中

## 2.6.11.2 HDFS2HBase

- HDFS2HBase

```

/**
 * 从hdfs读取数据
 * 然后写到hbase
 * 统计每个人的年龄，并将结果存储到hbase中
 */
public class Demo9_HDFS2HBase implements Tool {

```

```

//1. 创建配置对象
private Configuration configuration;
private final static String HBASE_CONNECT_KEY = "hbase.zookeeper.quorum";
private final static String HBASE_CONNECT_VALUE =
"hbase2:2181,hbase3:2181,hbase4:2181";
private final static String HDFS_CONNECT_KEY = "fs.defaultFS";
private final static String HDFS_CONNECT_VALUE = "hdfs://hbase1:9000";
private final static String MAPREDUCE_CONNECT_KEY =
"mapreduce.framework.name";
private final static String MAPREDUCE_CONNECT_VALUE = "yarn";

@Override
public int run(String[] args) throws Exception {
    Job job = Job.getInstance(configuration);
    job.setJarByClass(Demo9_HDFS2HBase.class);
    job.setMapperClass(HBaseMapper.class);
    job.setReducerClass(HBaseReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);
    //1. 创建表
    String tablename = "user_info";
    createTable(tablename);

    //2. 设置reduce向hbase输出
    TableMapReduceUtil.initTableReducerJob(tablename, HBaseReducer.class,
job);

    //3. 设置输入路径
    FileInputFormat.setInputPaths(job, new Path(args[0]));

    return job.waitForCompletion(true) ? 1 : 0;
}

/**
 * 校验表是否存在，如果不存在就创建，存在就先删除再创建
 */
private void createTable(String tablename) {
    //1. 获取admin对象
    Admin admin = HBaseUtils.getAdmin();
    //2.
    try {
        boolean isExist = admin.tableExists(TableName.valueOf(tablename));
        if(isExist) {
            admin.disableTable(TableName.valueOf(tablename));
            admin.deleteTable(TableName.valueOf(tablename));
        }
        HTableDescriptor tableDescriptor = new
HTableDescriptor(TableName.valueOf(tablename));
        HColumnDescriptor columnDescriptor1 = new
HColumnDescriptor("value_info");
        HColumnDescriptor columnDescriptor2 = new
HColumnDescriptor("age_info");
        columnDescriptor1.setBloomFilterType(BloomType.ROW);
        columnDescriptor2.setBloomFilterType(BloomType.ROW);
        columnDescriptor1.setVersions(1, 3);
        columnDescriptor2.setVersions(1, 3);
        tableDescriptor.addFamily(columnDescriptor1);
        tableDescriptor.addFamily(columnDescriptor2);
    }
}

```

```

        admin.createTable(tableDescriptor);
    } catch (IOException e) {
        e.printStackTrace();
    }finally {
        HBaseUtils.close(admin);
    }
}

/**
 * 统计每个人的年龄
 */
public static class HBaseMapper extends Mapper<LongWritable, Text, Text,
LongWritable> {
    private Text k = new Text();
    private LongWritable v = new LongWritable(1);
    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        //1. 读取一行
        String line = value.toString();
        //2. 切fields
        String[] columns = line.split("\t");
        //3. 切kv
        for (String column : columns) {
            if(column.contains("age")) {
                String[] kv = column.split(":");
                k.set(kv[1]);
                context.write(k, v);
                break;
            }
        }
    }
}

/**
 * 统计每个人的年龄
 */
public static class HBaseReducer extends TableReducer<Text, LongWritable,
ImmutableBytesWritable> {
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values, Context
context) throws IOException, InterruptedException {
        //1. 计数器
        long count = 0l;
        //2. 迭代
        Iterator<LongWritable> iterator = values.iterator();
        while (iterator.hasNext()) {
            LongWritable n = iterator.next();
            count += n.get();
        }
        //3. 输出一定要是可以修改hbase的对象, put, delete
        Put put = new Put(Bytes.toBytes(key.toString()));
        //4. 将结果集写入put对象
        put.addColumn(Bytes.toBytes("age_info"), Bytes.toBytes("age"),
Bytes.toBytes(key.toString()));
        put.addColumn(Bytes.toBytes("value_info"),
Bytes.toBytes("ageCount"), Bytes.toBytes(count+""));
    }
}

```

```

        //5. 写
        context.write(new
ImmutableBytesWritable(Bytes.toBytes(key.toString())), put);
    }
}

@Override
public void setConf(Configuration conf) {
    conf.set(HBASE_CONNECT_KEY, HBASE_CONNECT_VALUE); // 设置连接的hbase
    conf.set(HDFS_CONNECT_KEY, HDFS_CONNECT_VALUE); // 设置连接的hadoop
    conf.set(MAPREDUCE_CONNECT_KEY, MAPREDUCE_CONNECT_VALUE); // 设置使用的mr
运行平台
    this.configuration = conf;
}

@Override
public Configuration getConf() {
    return configuration;
}

private static Scan getScan() {
    return new Scan();
}

public static void main(String[] args) throws Exception {
    ToolRunner.run(HBaseConfiguration.create(), new Demo9_HDFS2HBase(),
args);
}
}

```

- 执行

```

[root@hbase1 home]# hadoop jar day23-HBase-1.0-SNAPSHOT.jar
cn.qphone.hbase.hbase2hdfs.Demo9_HDFS2HBase /hbase2hdfs/output/02

```

## 2.6.12 Hbase与Hive的整合

### 2.6.12.1 整合目的

1. HBase的最主要的目的是做数据存储
2. Hive的最主要作用是做数据分析
3. 整合的目的是为了方便的在项目中存储+分析
4. hbase中的表数据在hive中能看到，hive中的表数据在hbase中也能看到

### 2.6.12.2 整合Hive到HBase

- 添加新的依赖，下载jar包

```

<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-hbase-handler</artifactId>
    <version>1.2.1</version>
</dependency>

```

- 启动hive出错：

启动hive出错:

Exception in thread "main" java.lang.IncompatibleClassChangeError: Found class jline.Terminal, but interface was expected

解决方案:

```
cp /usr/apps/hive/hive-1.2.1/lib/jline-2.12.jar  
$HADOOP_HOME/share/hadoop/yarn/lib/
```

- 在hive中创建表

```
create database hive2hbase;  
use hive2hbase;
```

1. 在hive中创建hbase能看到的表:

```
create table if not exists hive2hbase (  
uid int,  
uname string,  
age int,  
sex string  
)  
stored by 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  
with serdeproperties(  
"hbase.columns.mapping"=":key,base_info:name,base_info:age,base_info:sex"  
)  
tblproperties(  
"hbase.table.name"="hive2hbase1"  
);
```

- 报错

```
> |;  
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.q1.exec.DDLTask. org.apache.hadoop.hbase.HTableDescriptor.addFamily(Lorg/apache/hadoop/hbase/HColumnDescriptor;)V
```


原因很简单, hive查找hbase中的版本匹配不一致的问题

解决方案:


将源码重新打包

- 重新编译步骤

1. 进入hive源码包, 找到hbase-handler的源码。使用Eclipse创建java工程, 名称随意

 **New Java Project** — □ ×

**Create a Java Project**

Create a Java project in the workspace or in an external location. 

Project name:

☒ Use default location

Location:  Browse...

**JRE**

☒ Use an execution environment JRE:  ▼

☐ Use a project specific JRE:  ▼

☐ Use default JRE (currently 'jdk1.8.0\_25') [Configure JREs...](#)

**Project layout**

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)

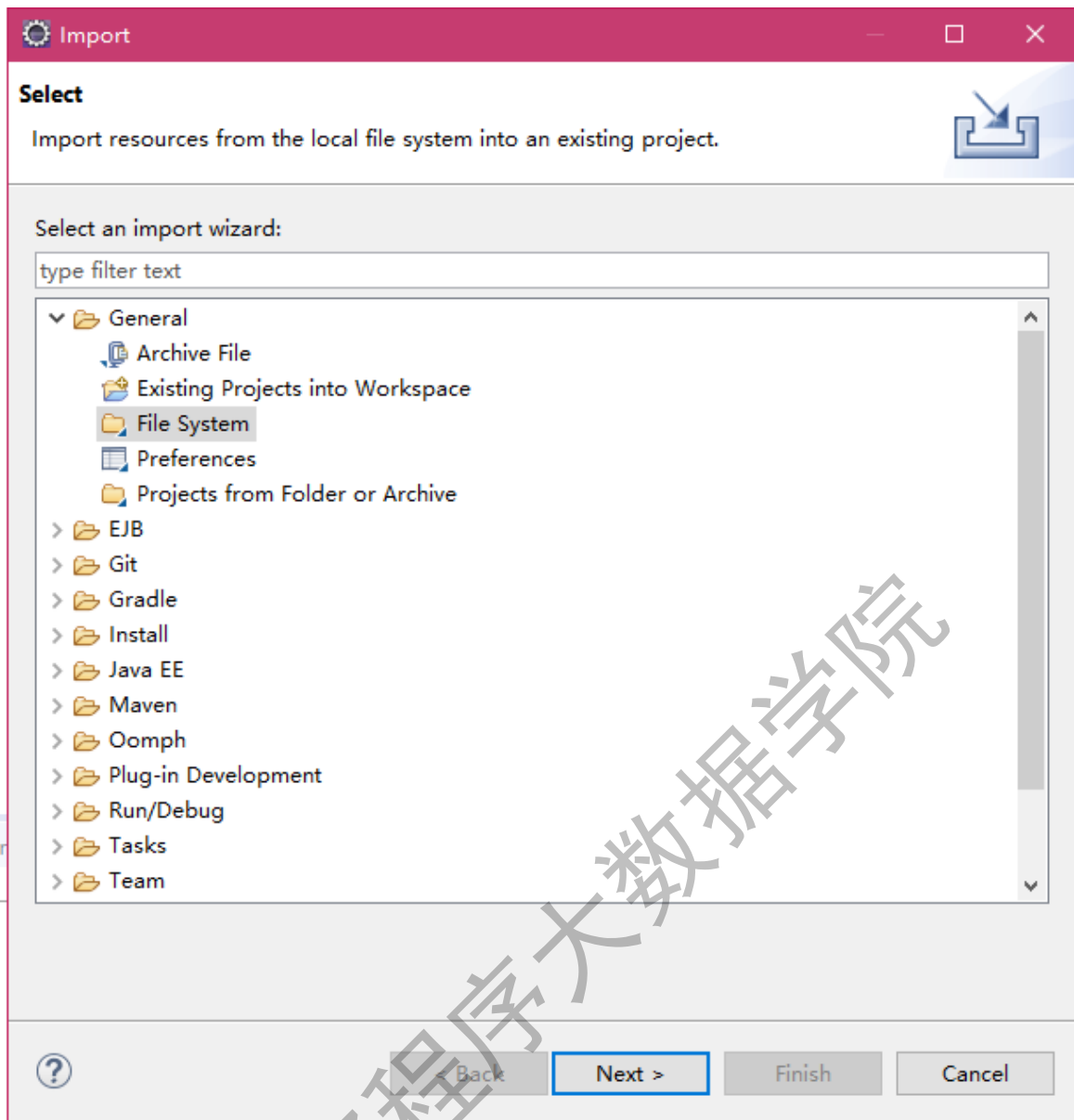
**Working sets**

☐ Add project to working sets New...

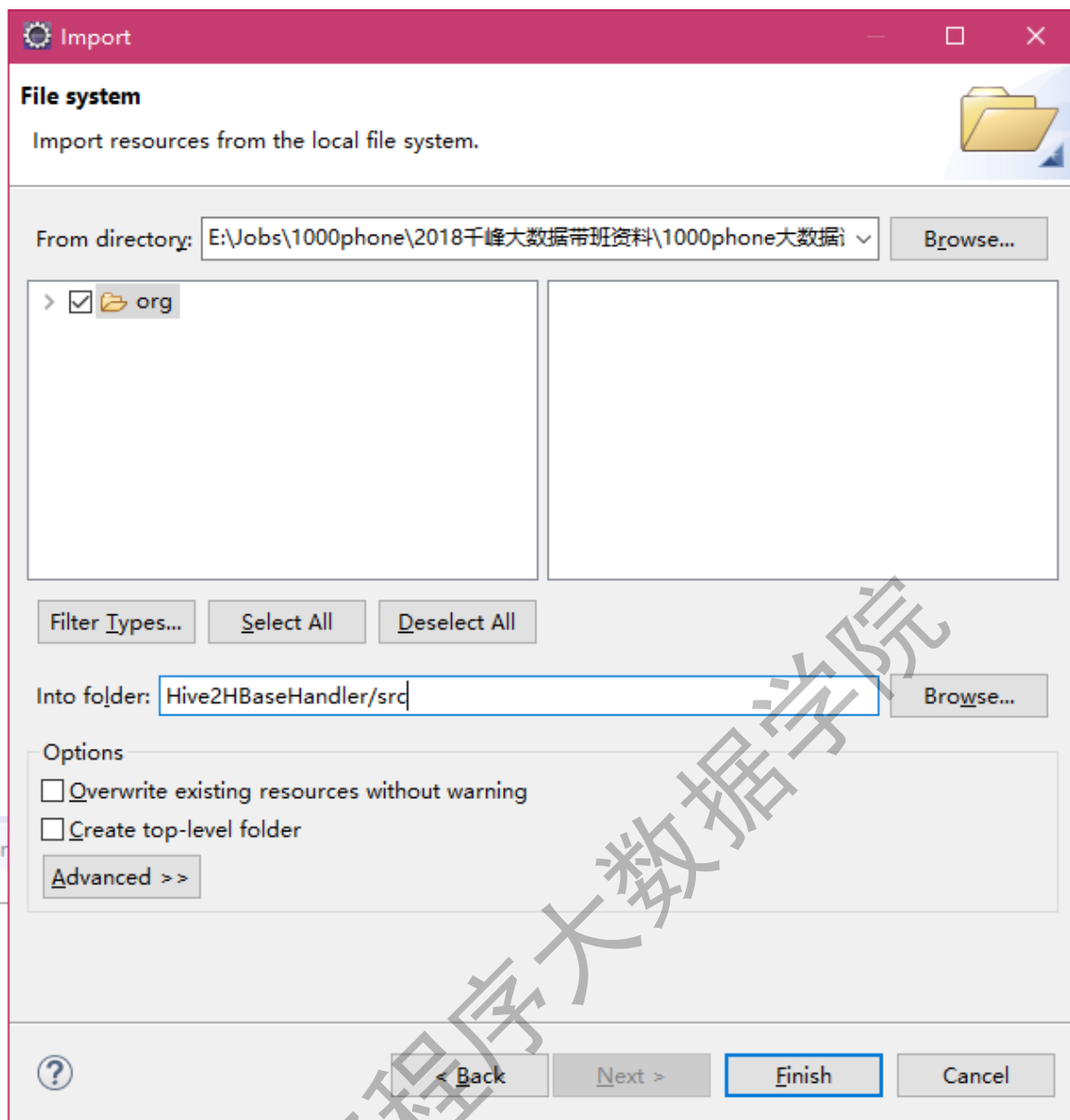
Working sets:  Select...

? < Back Next > Finish Cancel

2. 在src(请注意)下导入handler源码





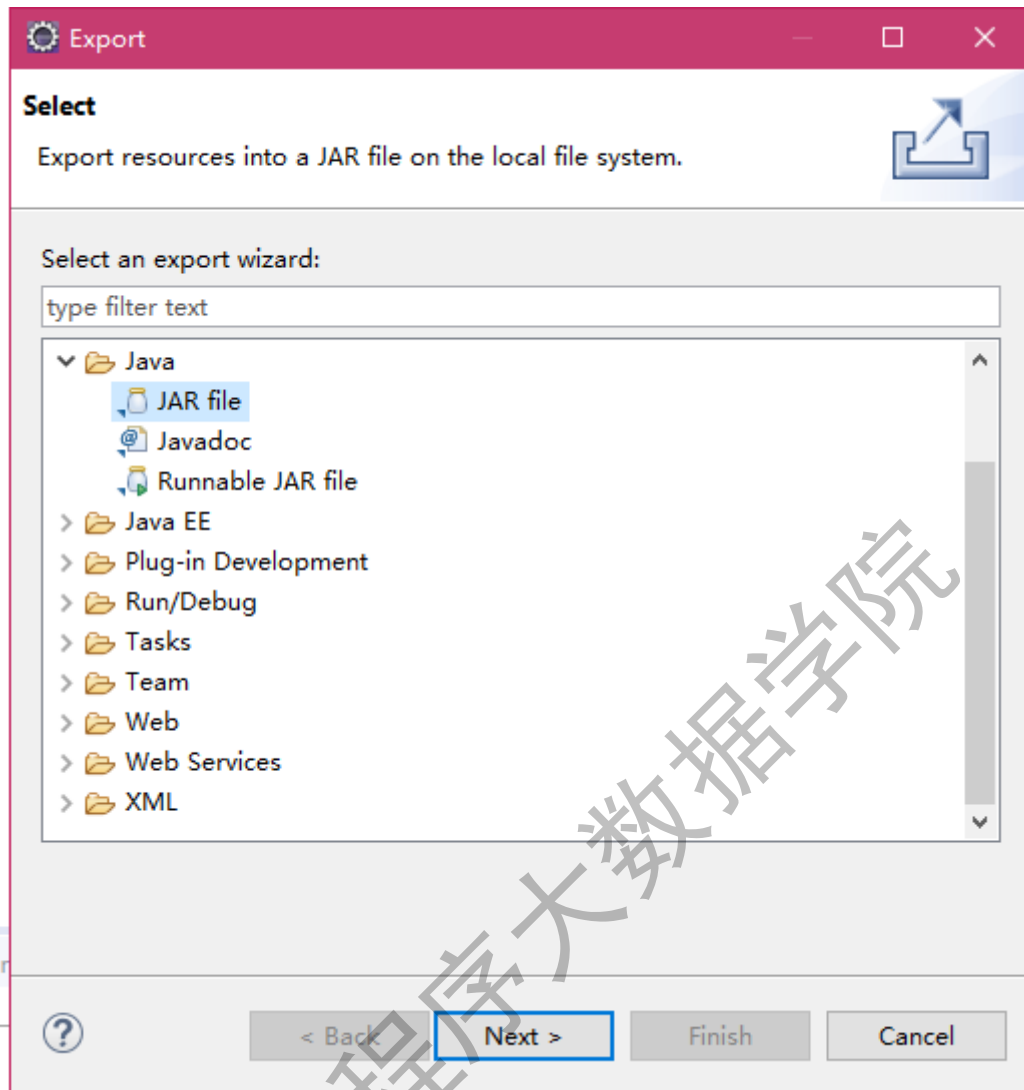


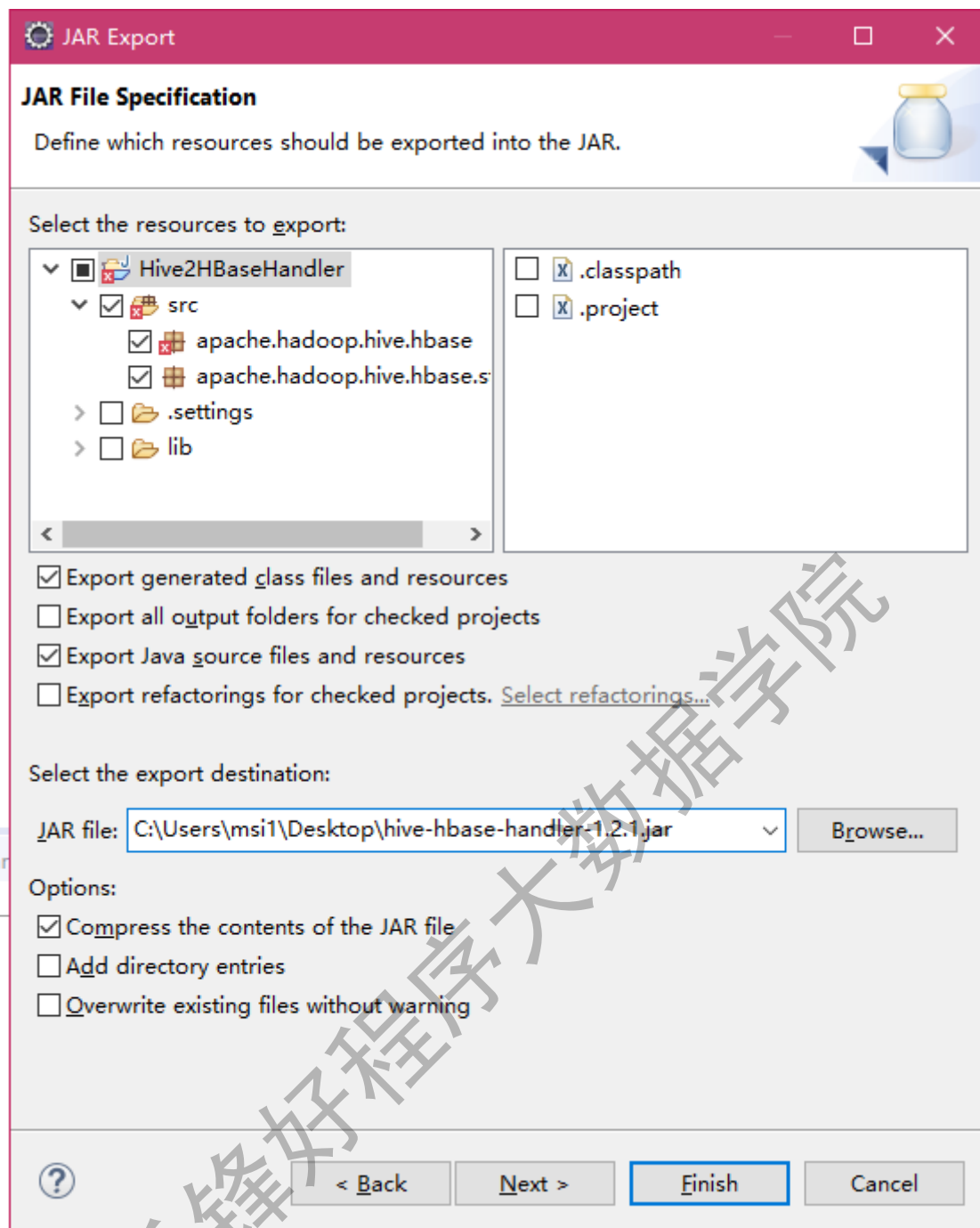
### 3. 导入关联jar包

我这里是把Hbase和Hive包下的所有jar包放入了工程下的lib文件夹中，不要将apache-curator-2.6.0.pom.xml也导入工程环境中，否则无法正确打包build-path

	commons-io-2.4.jar	2019/5/19 9:31	JAR 文件	181 KB
	commons-logging-1.2.jar	2019/5/19 9:31	JAR 文件	61 KB
	hadoop-common-2.8.1.jar	2019/5/19 9:35	JAR 文件	3,647 KB
	hadoop-mapreduce-client-core-2.8.1...	2019/5/19 9:36	JAR 文件	1,535 KB
	hbase-client-1.2.1.jar	2019/5/19 9:32	JAR 文件	1,268 KB
	hbase-common-1.2.1-tests.jar	2019/5/19 9:33	JAR 文件	223 KB
	hbase-protocol-1.2.1.jar	2019/5/19 9:33	JAR 文件	4,264 KB
	hbase-server-1.2.1.jar	2019/5/19 9:33	JAR 文件	4,043 KB
	hive-common-1.2.1.jar	2019/5/19 9:34	JAR 文件	286 KB
	hive-exec-1.2.1.jar	2019/5/19 9:34	JAR 文件	20,117 KB
	hive-metastore-1.2.1.jar	2019/5/19 9:34	JAR 文件	5,377 KB
	jsr305-3.0.0.jar	2019/5/19 9:34	JAR 文件	33 KB
	metrics-core-2.2.0.jar	2019/5/19 9:35	JAR 文件	81 KB
	zookeeper-3.4.6.jar	2019/5/19 9:34	JAR 文件	775 KB

#### 4. 导出jar包





5. 剩下的默认即可

导出jar包后，替换Hive lib 包下的相应jar包(hive-hbase-handler-1.2.1.jar)即可

- 重新建表，就可以成功了。然后在hive和hbase查看表

```
hbase(main):002:0> list
TABLE
hive2hbase1
t1
user_info
3 row(s) in 2.8950 seconds

=> ["hive2hbase1", "t1", "user_info"]

hbase(main):003:0> desc 'hive2hbase1'
Table 'hive2hbase1' is ENABLED
hive2hbase1
COLUMN FAMILIES DESCRIPTION
NAME => 'base_info', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'
1 row(s) in 4.1740 seconds
```

- 在hive中测试

1. 插入一条记录

```
insert into hive2hbase values(1, 'lixi', 32, '男');
```

```
hive (hive2hbase)> select * from hive2hbase;
OK
1      lixi      32      7
Time taken: 0.717 seconds, Fetched: 1 row(s)
```

- 在hbase中测试

1. 插入一条记录

```
put 'hive2hbase1', '2', 'base_info:age','33'
put 'hive2hbase1', '2', 'base_info:name','rock'
put 'hive2hbase1', '2', 'base_info:sex','女'
```

```
hbase(main):018:0> get 'hive2hbase1','2'
COLUMN                                CELL
base_info:age                         timestamp=1558262297104, value=33
base_info:name                        timestamp=1558262379159, value=rock
base_info:sex                         timestamp=1558262388078, value=\xE5\xA5\xB3
3 row(s) in 0.0400 seconds
hive (hive2hbase)> select * from hive2hbase where uid = '2';
OK
2      rock      33      女
```

- 在hive中加载数据(不能使用load data方式)

我们一般在hive的严格模式下，进制使用insert into values。

我们使用的是load data等命令进行导入数据

```
load data local inpath '/home/stu.txt' into table hive2hbase;
```

- insert into

```
create table if not exists t_stu (
  uid int,
  uname string,
  age int,
  sex string
)
row format delimited
fields terminated by ','
stored as textfile
;

load data local inpath '/home/stu.txt' into table t_stu;

insert into hive2hbase
select * from t_stu;
```

```
hive (hive2hbase)> select * from hive2hbase;
OK
1      lixi      32      7
2      rock      33      女
3      lee       18      man
4      curry     31      man
5      angrababy 32      woman

hbase(main):020:0> scan 'hive2hbase1'
ROW
1      COLUMN+CELL
1      column=base_info:age, timestamp=1558261982276, value=32
1      column=base_info:name, timestamp=1558261982276, value=lixi
1      column=base_info:sex, timestamp=1558261982276, value=7
2      column=base_info:age, timestamp=1558262297104, value=33
2      column=base_info:name, timestamp=1558262379159, value=rock
2      column=base_info:sex, timestamp=1558262380078, value=\xE5\xA5\xB3
3      column=base_info:age, timestamp=1558263648139, value=18
3      column=base_info:name, timestamp=1558263648139, value=lee
3      column=base_info:sex, timestamp=1558263648139, value=man
4      column=base_info:age, timestamp=1558263648139, value=31
4      column=base_info:name, timestamp=1558263648139, value=curry
4      column=base_info:sex, timestamp=1558263648139, value=man
5      column=base_info:age, timestamp=1558263648139, value=32
5      column=base_info:name, timestamp=1558263648139, value=angrababy
5      column=base_info:sex, timestamp=1558263648139, value=woman
5 row(s) in 0.6900 seconds
```

### 2.6.12.3 HBase的表映射到hive

- 查看hbase的表结构

```
hbase(main):011:0> scan 't2'
ROW
001      COLUMN+CELL
value=20      column=f1:age, timestamp=1558264169222,
001      column=f1:name, timestamp=1558264143114,
value=lixi
002      column=f1:age, timestamp=1558264164145,
value=19
002      column=f1:name, timestamp=1558264147143,
value=lixi
003      column=f1:age, timestamp=1558264156996,
value=18
003      column=f1:name, timestamp=1558264150043,
value=lixi
```

- 根据hbase的表结构建立hive表

```
create external table if not exists hbase2hive(
uid string,
age int,
name string
)
stored by 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
with serdeproperties(
"hbase.columns.mapping"=":key,f1:age,f1:age"
)
tblproperties(
"hbase.table.name"="t2"
);

tip:
:key可以省略, 省略就是以hive的第一个字段作为hbase的rowkey

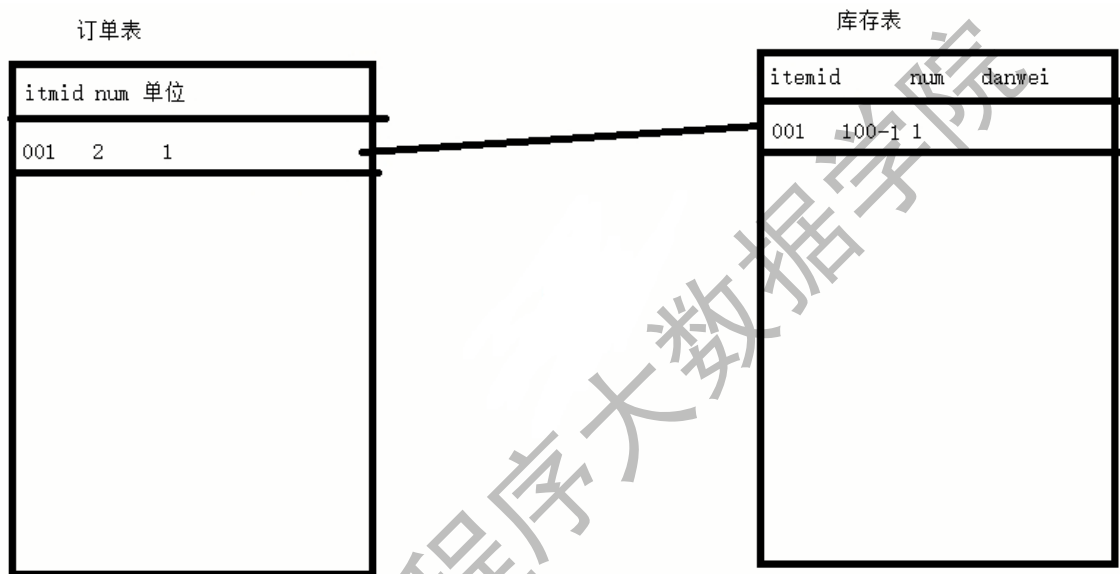
hive (hive2hbase)> select * from hbase2hive;
OK
```

001	20	20
002	19	19
003	18	18

#### • 注意事项

- 1、映射hbase的列时，要么就写:key，要么不写，否则列数不匹配，默认使用:key
- 2、hbase中表存在的时候，在hive中创建表时应该使用external关键字。
- 3、如果删除了hbase中对应的表数据，那么hive中就不能查询出来数据。
- 4、hbase中的列和hive中的列个数和数据类型应该尽量相同，hive表和hbase表的字段不是按照名字匹配，而是按照顺序来匹配的。
- 5、hive、hbase和mysql等可以使用第三方工具来进行整合。

### 2.6.13 Hbase的二级索引



当一产生订单表，库存表的数量就要减少对应的

我关注了明星

xiaochen	高圆圆 贾静雯 刘德华
高建政	高圆圆

明星被哪些粉丝关注

高圆圆	xiaochen 高建政……
贾静雯	……

## 2.6.14 Hbase协处理器案例

### 2.6.14.1 简介

协处理器允许用户在region服务器上运行自己的代码，允许用户执行region级别的操作，并且可以使用与RDBMS中触发器(trigger)类似的功能。在客户端，用户不用关心操作具体在哪里执行，HBase的分布式框架会帮助用户把这些工作变得透明。

分为：observer、endpoint

- observer

这一类协处理器与触发器(trigger)类似：回调函数（也被称作钩子函数，hook）在一些特定事件发生时被执行。这些事件包括一些用户产生的事件，也包括服务器端内部自动产生的事件。

协处理器框架提供的接口如下：

**RegionObserver**：用户可以用这种的处理器处理数据修改事件，它们与表的region联系紧密。

**MasterObserver**：可以被用作管理或DDL类型的操作，这些是集群级事件。

**WALObserver**：提供控制WAL的钩子函数

Observer提供了一些设计好的回调函数，每个操作在集群服务器端都可以被调用

- endpoint

除了事件处理之外还需要将用户自定义操作添加到服务器端。用户代码可以被部署到管理数据的服务器端，例如，做一些服务器端计算的工作。

Endpoint通过添加一下远程过程调用来动态扩展RPC协议。可以把它们理解为与RDBMS中类似的存储过程。

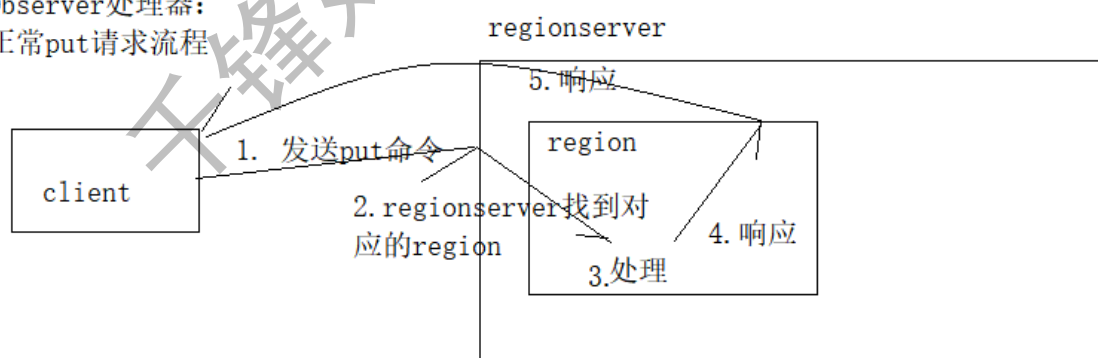
Endpoint可以与observer的实现组合起来直接作用于服务器端的状态。

### 2.6.14.2 Observer协处理器

- 普通的put请求流程

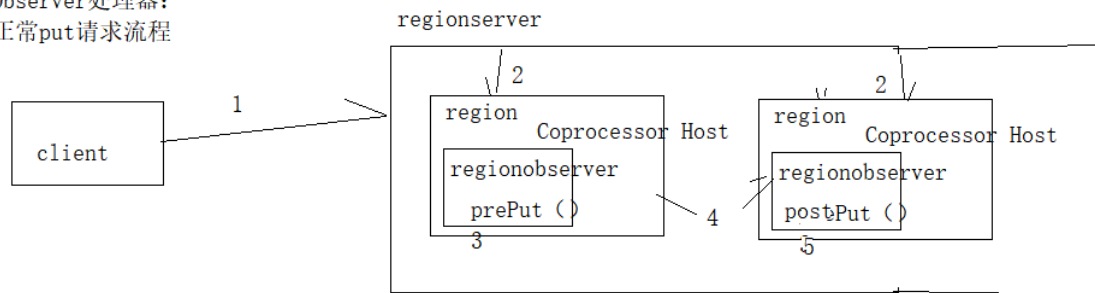
Observer处理器：

正常put请求流程



- 加入Observer协处理器的put请求流程

Observer处理器：  
正常put请求流程



- 1 客户端发出put请求
- 2 请求被分派给合适的RegionServer和Region
- 3 coprocessorHost拦截请求，然后在表上登记的每个RegionObserver上调用prePut方法
- 4 如果没有被prePut拦截，请求就继续送到region，然后进行处理
- 5 region产生了处理后的结果再次被coprocessor拦截，调用postPut方法
- 6 假设没有被postPut拦截响应，最终结果就返回给客户端

- Observer的类型

**RegionObserver** : 此组件勾在数据访问和操作阶段，所有标准的数据操作命令都可以被pre-hooks和post-hooks拦截

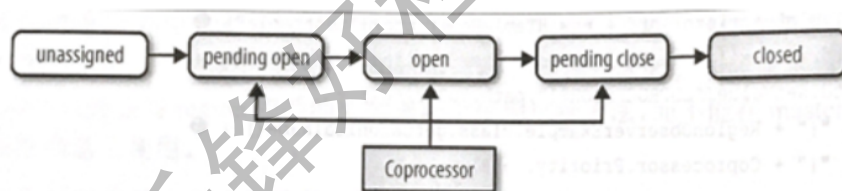
**WALObserver** : WAL所支持的observer，可用的钩子是pre-WAL和post-WAL

**MasterObserver** : 勾住DDL事件，如表创建

### 2.6.14.3 RegionObserver类

属于observer协处理器：当一个特定的region级别的操作发生时，它们的钩子函数会被触发。这些操作可以被分为两类：region生命周期变化和客户端API调用。

- 处理region生命周期事件



这些observer可以与pending open、open和pending close状态通过钩子链接。每一个钩子都被框架隐式地调用。

### 2.6.14.4 例子：处理特殊行键

```

/**
 *
 */
public class RegionObserverExample extends BaseRegionObserver{

    public static final byte[] FIXED_ROW = Bytes.toBytes("@@GETTIME@@");

    @Override
    public void preGetOp(ObserverContext<RegionCoprocessorEnvironment> e, Get
    get, List<Cell> results)

```



```

        throws IOException {
            //检查请求的行键是否匹配
            if (Bytes.equals(get.getRow(), FIXED_ROW)) {
                //创建一个特殊的keyvalue, 只包含当前的服务器时间。
                KeyValue keyValue = new KeyValue(get.getRow(), FIXED_ROW, FIXED_ROW,
                    Bytes.toBytes(System.currentTimeMillis()));
                results.add(keyValue);
            }
        }
    }
}

```

- 1.新建HBase表

```
create 'test_coprocessor', 'info'
```

- 2.写处理器打包上传到HDFS

```
/data/fz/hbase/coprocessor/RegionObserverExample.jar
```

- 3.装载协处理器

```

hbase(main):004:0> disable 'test_coprocessor'

hbase(main):006:0> alter
'test_coprocessor',METHOD=>'table_att','COPROCESSOR'=>'hdfs://cluster1/data/fz/h
base/coprocessor/RegionObserverExample.jar|com.hbase.coprocessor.RegionObserverE
xample|1001'

hbase(main):007:0> enable 'test_coprocessor'

```

- 查看挂载情况

```

hbase(main):009:0> desc 'test_coprocessor'

Table test_coprocessor is ENABLED

test_coprocessor, {TABLE_ATTRIBUTES => {METADATA => {'COPROCESSOR$1' =>
'hdfs://cluster1/data/fz/hbase/coprocessor/RegionObserverExample.jar|com.hbase.c
oprocess

or.RegionObserverExample|1001'}}}

COLUMN FAMILIES DESCRIPTION

{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE =>
'65536', REPLICATION_SCOPE => '0'}

```

- 5.协调处理器测试
- 行键@@@GETTIME@@@被observer的preGet捕获, 然后添加当前服务器时间。

```
hbase(main):054:0> get 'test_coprocessor','@@@GETTIME@@@'
```

COLUMN	CELL
@@@GETTIME@@@:@@@GETTIME@@@	timestamp=9223372036854775807, value=\x00\x00\x01e\x8E\xc6ad

1 row(s) in 0.0450 seconds

- 插入一条行键为@@@GETTIME@@@的数据

```
hbase(main):055:0> put 'test_coprocessor','@@@GETTIME@@@','info:name','nimei'
```

0 row(s) in 0.0540 seconds

- 此时再次使用get

```
hbase(main):057:0> get 'test_coprocessor','@@@GETTIME@@@'
```

COLUMN	CELL
@@@GETTIME@@@:@@@GETTIME@@@	timestamp=9223372036854775807, value=\x00\x00\x01e\x8E\xc7\xD3\x9B
info:name	timestamp=1535698764962, value=nimei

2 row(s) in 0.0130 seconds

- 如果捕获的该行键恰巧在表中同时存在，就会出现上面的情况，为了避免这种情况，可以使用 e.bypass。更新步骤如下
- 6.卸载已有协处理器

```
alter 'test_coprocessor',METHOD => 'table_att_unset',NAME =>'COPROCESSOR$1'
```

- 7.重新打包RegionObserverWithBypassExample上传HDFS

```
**
*
*/
public class RegionObserverWithBypassExample extends BaseRegionObserver{

    public static final byte[] FIXED_ROW = Bytes.toBytes("@@@GETTIME@@@");

    @Override
    public void preGetOp(ObserverContext<RegionCoprocessorEnvironment> e, Get
get, List<Cell> results)
        throws IOException {
        //检查请求的行键是否匹配
        if (Bytes.equals(get.getRow(), FIXED_ROW)) {
            //创建一个特殊的keyvalue，只包含当前的服务器时间。

```

```

        KeyValue keyValue = new KeyValue(get.getRow(), FIXED_ROW, FIXED_ROW,
Bytes.toBytes(System.currentTimeMillis()));
        results.add(keyValue);
        //一旦特殊的keyvalue被添加，之后的操作都会被跳过
        e.bypass();
    }
}
}

```

- 8.挂载新的协处理器

```
disable 'test_coprocessor'
```

```
hbase(main):034:0> alter
'test_coprocessor',METHOD=>'table_att','COPROCESSOR'=>'hdfs://cluster1/data/fz/h
base/coprocessor/RegionObserverWithBypassExample.jar|com.hbase.coprocessor.Regio
nObserverWithBypassExample|1001'
```

```
enable 'test_coprocessor'
```

- 9.查看挂载情况

```
hbase(main):036:0> desc 'test_coprocessor'
```

```
Table test_coprocessor is ENABLED
```

```
test_coprocessor, {TABLE_ATTRIBUTES => {METADATA => {'COPROCESSOR$1' =>
'hdfs://cluster1/data/fz/hbase/coprocessor/RegionObserverWithBypassExample.jar|c
om.hbase.c
```

```
oprocessor.RegionObserverWithBypassExample|1001'}}}
```

```
COLUMN FAMILIES DESCRIPTION
```

```
{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
```

```
COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE =>
'65536', REPLICATION_SCOPE => '0'}
```

```
1 row(s) in 0.0160 seconds
```

- 10.测试

```
hbase(main):037:0> get 'test_coprocessor', '@@@GETTIME@@@'
```

COLUMN

CELL

```
@@@GETTIME@@@:@@@GETTIME@@@ timestamp=9223372036854775807,  
value=\x00\x00\x01e\x8E\xDA\xDB\xC5
```

1 row(s) in 0.0210 seconds

## 2.6.15 Hbase的rowKey设计原则与案例

### 2.6.15.1 设计原则

- 长度原则

Rowkey是一个二进制码流，Rowkey的长度被很多开发者建议说设计在10~100个字节，不过建议是越短越好，不要超过16个字节。

原因如下：

(1) 数据的持久化文件HFile中是按照keyvalue存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用100\*1000万=10亿个字节，将近1G数据，这会极大影响HFile的存储效率；

(2) MemStore将缓存部分数据到内存，如果Rowkey字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好。

(3) 目前操作系统都是64位系统，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

- 散列原则

如果Rowkey是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将Rowkey的高位作为散列字段，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个Regionserver实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息将产生所有新数据都在一个 RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别RegionServer，降低查询效率。

- 唯一原则

必须在设计上保证其唯一性。

### 2.6.15.2 应用场景

- 针对事务数据Rowkey设计

事务数据是带时间属性的，建议将时间信息存入到Rowkey中，这有助于提示查询检索速度。对于事务数据建议缺省就按天为数据建表，这样设计的好处是多方面的。按天分表后，时间信息就可以去掉日期部分只保留小时分钟毫秒，这样4个字节即可搞定。加上散列字段2个字节一共6个字节即可组成唯一 Rowkey。如下图所示：

事务数据Rowkey设计						
第0字节	第1字节	第2字节	第3字节	第4字节	第5字节	...
散列字段		时间字段(毫秒)				扩展字段
0~65535(0x0000~0xFFFF)		0~86399999(0x00000000~0x05265BFF)				

这样的设计从操作系统内存管理层面无法节省开销，因为64位操作系统是必须8字节对齐。但是对于持久化存储中Rowkey部分可以节省25%的开销。也许有人要问为什么不将时间字段以主机字节序保存，这样它也可以作为散列字段了。这是因为时间范围内的数据还是尽量保证连续，相同时间范围内的数据查找的概率很大，对查询检索有好的效果，因此使用独立的散列字段效果更好，对于某些应用，我们可以考虑利用散列字段全部或者部分来存储某些数据的字段信息，只要保证相同散列值在同一时间（毫秒）唯一。

### • 针对统计数据的Rowkey设计

统计数据也是带时间属性的，统计数据最小单位只会到分钟（到秒预统计就没意义了）。同时对于统计数据我们也缺省采用按天数据分表，这样设计的好处无需多说。按天分表后，时间信息只需要保留小时分钟，那么0~1400只需占用两个字节即可保存时间信息。由于统计数据某些维度数量非常庞大，因此需要4个字节作为序列字段，因此将散列字段同时作为序列字段使用也是6个字节组成唯一Rowkey。如下图所示：

统计数据Rowkey设计						
第0字节	第1字节	第2字节	第3字节	第4字节	第5字节	...
散列字段(序列字段)				时间字段(分钟)		扩展字段
0x00000000~0xFFFFFFFF				0~1439(0x0000~0x059F)		

同样这样的设计从操作系统内存管理层面无法节省开销，因为64位操作系统是必须8字节对齐。但是对于持久化存储中Rowkey部分可以节省25%的开销。预统计数据可能涉及到多次反复的重计算要求，需确保作废的数据能有效删除，同时不能影响散列的均衡效果，因此要特殊处理。

### • 针对通用数据的Rowkey设计

通用数据采用自增序列作为唯一主键，用户可以选择按天建分表也可以选择单表模式。这种模式需要确保同时多个入库加载模块运行时散列字段（序列字段）的唯一性。可以考虑给不同的加载模块赋予唯一因子区别。设计结构如下图所示。

通用数据Rowkey设计				
第0字节	第1字节	第2字节	第3字节	...
散列字段(序列字段)				扩展字段（控制在12字节内）
0x00000000~0xFFFFFFFF				可由多个用户字段组成

### • 支持多条件查询的RowKey设计

HBase按指定的条件获取一批记录时，使用的就是scan方法。scan方法有以下特点：

- (1) scan可以通过setCaching与setBatch方法提高速度（以空间换时间）；

(2) scan可以通过setStartRow与setEndRow来限定范围。范围越小，性能越高。

通过巧妙的RowKey设计使我们批量获取记录集中的元素挨在一起（应该在同一个Region下），可以在遍历结果时获得很好的性能。

(3) scan可以通过setFilter方法添加过滤器，这也是分页、多条件查询的基础。

在满足长度、三列、唯一原则后，我们需要考虑如何通过巧妙设计RowKey以利用scan方法的范围功能，使得获取一批记录的查询速度能提高。下例就描述如何将多个列组合成一个RowKey，使用scan的range来达到较快查询速度。

例子：

我们在表中存储的是文件信息，每个文件有5个属性：文件id（long，全局唯一）、创建时间（long）、文件名（String）、分类名（String）、所有者（User）。

我们可以输入的查询条件：文件创建时间区间（比如从20120901到20120914期间创建的文件），文件名（“中国好声音”），分类（“综艺”），所有者（“浙江卫视”）。

假设当前我们一共有如下文件：

ID	CreateTime	Name	Category	UserID
1	20120902	中国好声音第1期	综艺	1
2	20120904	中国好声音第2期	综艺	1
3	20120906	中国好声音外卡赛	综艺	1
4	20120908	中国好声音第3期	综艺	1
5	20120910	中国好声音第4期	综艺	1
6	20120912	中国好声音选手采访	综艺花絮	2
7	20120914	中国好声音第5期	综艺	1
8	20120916	中国好声音录制花絮	综艺花絮	2
9	20120918	张玮独家专访	花絮	3
10	20120920	加多宝凉茶广告	综艺广告	4

这里UserID应该对应另一张User表，暂不列出。我们只需知道UserID的含义：

1代表 浙江卫视； 2代表 好声音剧组； 3代表 XX微博； 4代表赞助商。调用查询接口的时候将上述5个条件同时输入find(20120901,20121001,"中国好声音","综艺","浙江卫视")。此时我们应该得到记录应该有第1、2、3、4、5、7条。第6条由于不属于“浙江卫视”应该不被选中。我们在设计RowKey时可以这样做：采用 UserID + CreateTime + FileID组成RowKey，这样既能满足多条件查询，又能有很快的查询速度。

需要注意以下几点：

(1) 每条记录的RowKey，每个字段都需要填充到相同长度。假如预期我们最多有10万量级的用户，则userID应该统一填充至6位，如000001，000002...

(2) 结尾添加全局唯一的FileID的用意也是使每个文件对应的记录全局唯一。避免当UserID与CreateTime相同时的两个不同文件记录相互覆盖。

按照这种RowKey存储上述文件记录，在HBase表中是下面的结构：

rowKey (userID 6 + time 8 + fileID 6) name category ...

00000120120902000001

00000120120904000002

00000120120906000003

00000120120908000004

00000120120910000005

00000120120914000007

00000220120912000006

00000220120916000008

00000320120918000009

00000420120920000010

怎样用这张表？

在建立一个scan对象后，我们setStartRow(00000120120901)，setEndRow(00000120120914)。

这样，scan时只扫描userID=1的数据，且时间范围限定在这个指定的时间段内，满足了按用户以及按时间范围对结果的筛选。并且由于记录集中存储，性能很好。

然后使用

SingleColumnValueFilter (org.apache.hadoop.hbase.filter.SingleColumnValueFilter)，共4个，分别约束name的上下限，与category的上下限。满足按同时按文件名以及分类名的前缀匹配。

(注意：使用SingleColumnValueFilter会影响查询性能，在真正处理海量数据时会消耗很大的资源，且需要较长的时间)

如果需要分页还可以再加一个PageFilter限制返回记录的个数。

以上，我们完成了高性能的支持多条件查询的HBase表结构设计。

## 2.6.16 Hbase企业级调优

### 2.6.16.1服务端

1.hbase.regionserver.handler.count: rpc请求的线程数量，默认值是10，生产环境建议使用100，也不是越大越好，特别是当请求内容很大的时候，比如scan/put几M的数据，会占用过多的内存，有可能导致频繁的GC，甚至出现内存溢出。



2.hbase.master.distributed.log.splitting: 默认值为true, 建议设为false。关闭hbase的分布式日志切割, 在log需要replay时, 由master来负责重放

3.hbase.regionserver.hlog.splitlog.writer.threads: 默认值是3, 建议设为10, 日志切割所用的线程数

4.hbase.snapshot.enabled: 快照功能, 默认是false(不开启), 建议设为true, 特别是对某些关键的表, 定时用快照做备份是一个不错的选择。

5.hbase.hregion.max.filesize: 默认是10G, 如果任何一个column family里的StoreFile超过这个值, 那么这个Region会一分为二, 因为region分裂会有短暂的region下线时间(通常在5s以内), 为减少对业务端的影响, 建议手动定时分裂, 可以设置为60G。

6.hbase.hregion.majorcompaction: hbase的region主合并的间隔时间, 默认为1天, 建议设置为0, 禁止自动的major主合并, major合并会把一个store下所有的storefile重写为一个storefile文件, 在合并过程中还会把有删除标识的数据删除, 在生产集群中, 主合并能持续数小时之久, 为减少对业务的影响, 建议在业务低峰期进行手动或者通过脚本或者api定期进行major合并。

7.hbase.hregion.memstore.flush.size: 默认值128M, 单位字节, 一旦有memstore超过该值将被flush, 如果regionserver的jvm内存比较充足(16G以上), 可以调整为256M。

8.hbase.hregion.memstore.block.multiplier: 默认值2, 如果一个memstore的内存大小已经超过hbase.hregion.memstore.flush.size \* hbase.hregion.memstore.block.multiplier, 则会阻塞该memstore的写操作, 为避免阻塞, 建议设置为5, 如果太大, 则会有OOM的风险。如果在regionserver日志中出现"Blocking updates for '<threadName>' on region <regionName>: memstore size <多少M> is >= than blocking <多少M> size"的信息时, 说明这个值该调整了。

9.hbase.hstore.compaction.min: 默认值为3, 如果任何一个store里的storefile总数超过该值, 会触发默认的合并操作, 可以设置5~8, 在手动的定期major compact中进行storefile文件的合并, 减少合并的次数, 不过这会延长合并的时间, 以前的对应参数为hbase.hstore.compactionThreshold。

10.hbase.hstore.compaction.max: 默认值为10, 一次最多合并多少个storefile, 避免OOM。

11.hbase.hstore.blockingStoreFiles: 默认为7, 如果任何一个store(非.META.表里的store)的storefile的文件数大于该值, 则在flush memstore前先进行split或者compact, 同时把该region添加到flushQueue, 延时刷新, 这期间会阻塞写操作直到compact完成或者超过hbase.hstore.blockingwaitTime(默认90s)配置的时间, 可以设置为30, 避免memstore不及时flush。当regionserver运行日志中出现大量的"Region <regionName> has too many store files; delaying flush up to 90000ms"时, 说明这个值需要调整了

12.hbase.regionserver.global.memstore.upperLimit: 默认值0.4, regionserver所有memstore占用内存存在总内存中的upper比例, 当达到该值, 则会从整个regionserver中找出最需要flush的region进行flush, 直到总内存比例降到该数以下, 采用默认值即可。

13.hbase.regionserver.global.memstore.lowerLimit: 默认值0.35, 采用默认值即可。

14.hbase.regionserver.thread.compaction.small: 默认值为1, regionserver做Minor Compaction时线程池里线程数目, 可以设置为5。

15.hbase.regionserver.thread.compaction.large: 默认值为1, regionserver做Major Compaction时线程池里线程数目, 可以设置为8。

16.hbase.regionserver.lease.period: 默认值60000(60s), 客户端连接regionserver的租约超时时间, 客户端必须在这个时间内汇报, 否则则认为客户端已死掉。这个最好根据实际业务情况进行调整



17.hfile.block.cache.size: 默认值0.25, regionserver的block cache的内存大小限制, 在偏向读的业务中, 可以适当调大该值, 需要注意的是hbase.regionserver.global.memstore.upperLimit的值和hfile.block.cache.size的值之和必须小于0.8。

18.dfs.socket.timeout: 默认值60000(60s), 建议根据实际regionserver的日志监控发现了异常进行合理的设置, 比如我们设为900000, 这个参数的修改需要同时更改hdfs-site.xml

19.dfs.datanode.socket.write.timeout: 默认480000(480s), 有时regionserver做合并时, 可能会出现datanode写超时的情况, 480000 millis timeout while waiting for channel to be ready for write, 这个参数的修改需要同时更改hdfs-site.xml

## 2.6.16.2 jvm和垃圾收集参数

```
export HBASE_REGIONSERVER_OPTS="-Xms36g -Xmx36g -Xmn1g -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=15 -XX:CMSInitiatingOccupancyFraction=70 -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/data/logs/gc-$(hostname)-hbase.log"
```

由于我们服务器内存较大(96G), 我们给一部分regionserver的jvm内存开到64G, 到现在为止, 还没有发生过一次full gc, hbase在内存使用控制方面确实下了不少功夫, 比如各种blockcache的实现, 细心的同学可以看源码。

## 2.6.16.3 客户端

1.hbase.client.write.buffer: 默认为2M, 写缓存大小, 推荐设置为5M, 单位是字节, 当然越大占用的内存越多, 此外测试过设为10M下的入库性能, 反而没有5M好

2.hbase.client.pause: 默认是1000(1s), 如果你希望低延时的读或者写, 建议设为200, 这个值通常用于失败重试, region寻找等

3.hbase.client.retries.number: 默认值是10, 客户端最多重试次数, 可以设为11, 结合上面的参数, 共重试时间71s

4.hbase.ipc.client.tcpnodelay: 默认是false, 建议设为true, 关闭消息缓冲

5.hbase.client.scanner.caching: scan缓存, 默认为1, 避免占用过多的client和rs的内存, 一般1000以内合理, 如果一条数据太大, 则应该设置一个较小的值, 通常是设置业务需求的一次查询的数据条数

如果是扫描数据对下次查询没有帮助, 则可以设置scan的setCacheBlocks为false, 避免使用缓存;

6.table用完需关闭, 关闭scanner

7.限定扫描范围: 指定列簇或者指定要查询的列, 指定startRow和endRow

8.使用Filter可大量减少网络消耗

9.通过Java多线程入库和查询, 并控制超时时间。后面会共享下我的hbase单机多线程入库的代码

10.建表注意事项:

开启压缩

合理的设计rowkey

进行预分区

开启bloomfilter

## 2.6.16.14 Zookeeper调优

1. `zookeeper.session.timeout`: 默认值3分钟, 不可配置太短, 避免session超时, hbase停止服务, 线上生产环境由于配置为1分钟, 如果太长, 当regionserver挂掉, zk还得等待这个超时时间(已有patch修复), 从而导致master不能及时对region进行迁移。
2. `zookeeper`数量: 建议5个或者7个节点。给每个zookeeper 4G左右的内存, 最好有独立的磁盘。
3. `hbase.zookeeper.property.maxClientCnxns`: zk的最大连接数, 默认为300, 无需调整。
4. 设置操作系统的swappiness为0, 则在物理内存不够的情况下才会使用交换分区, 避免GC回收时会花费更多的时间, 当超过zk的session超时时间则会出现regionserver宕机的误报

## 2.6.16.15 HDFS调优

1. `dfs.name.dir`: namenode的数据存放地址, 可以配置多个, 位于不同的磁盘并配置一个nfs远程文件系统, 这样namenode的数据可以有多个备份
2. `dfs.namenode.handler.count`: namenode节点RPC的处理线程数, 默认为10, 可以设置为60
3. `dfs.datanode.handler.count`: datanode节点RPC的处理线程数, 默认为3, 可以设置为30
4. `dfs.datanode.max.xcievers`: datanode同时处理文件的上限, 默认为256, 可以设置为8192

## 2.6.16.16 其他

列族名、column名、rowkey都会存储到hfile中, 因此这几项在设计表结构时都尽量短些

regionserver的region数量不要过1000, 过多的region会导致产生很多memstore, 可能会导致内存溢出, 也会增加major compact的耗时