

HBase 数据库

目录

1、HBase 数据库介绍	1
1.1、产生背景.....	1
1.2、简介.....	2
1.3、表结构逻辑视图.....	3
1.3.1、行键（RowKey）	3
1.3.2、列簇（Column Family）	4
1.3.3、时间戳（TimeStamp）	4
1.3.4、单元格（Cell）	4
1.4、HBase 应用场景	5
2、hbase 集群结构.....	5
3、HBase 和 Hive 的比较	6
3.1、相同点.....	6
3.2、不同点.....	6
4、hbase 集群搭建.....	6
4.1、安装步骤.....	6
5、hbase 命令行演示.....	9
6、Hbase Java API 代码开发.....	15
6.1、基本增删改查实现.....	19
6.2、过滤器查询.....	27

1、HBase 数据库介绍

1.1、产生背景

自 1970 年以来，关系数据库用于数据存储和维护有关问题的解决方案。大数据的出现后，好多公司实现处理大数据并从中受益，并开始选择像 Hadoop 的解决方案。Hadoop 使用分布式文件系统，用于存储大数据，并使用 MapReduce 来处理。Hadoop 擅长于存储各种格式的庞大的数据，任意的格式甚至非结构化的处理。

Hadoop 的限制

Hadoop 只能执行批量处理，并且只以顺序方式访问数据。这意味着必须搜索整个数据集，即使是最简单的搜索工作。

当处理结果在另一个庞大的数据集，也是按顺序处理一个巨大的数据集。在这一点上，一个新的解决方案，需要访问数据中的任何点（随机访问）单元。

Hadoop 随机存取数据库

应用程序，如 HBase，Cassandra，CouchDB，Dynamo 和 MongoDB 都是一些存储大量数据和以随机方式访问数据的数据库。

总结：

- (1) 海量数据量存储成为瓶颈，单台机器无法**负载大量数据**
- (2) 单台机器 IO 读写请求成为海量数据存储时候**高并发大规模请求**的瓶颈
- (3) 随着数据规模越来越大，大量业务场景开始考虑数据存储**横向水平扩展**，使得存储服务可以增加/删除，而目前的关系型数据库更专注于一台机器

1.2、简介

官网：<http://hbase.apache.org/>

- 1、Apache HBase™ is the Hadoop database, a distributed, scalable, big data store.
- 2、Use Apache HBase™ when you need random, realtime read/write access to your Big Data.
- 3、This project's goal is the hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware.
- 4、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.

HBase 是 BigTable 的开源（源码使用 Java 编写）版本。是 Apache Hadoop 的数据库，是建立在 HDFS 之上，被设计用来提供高可靠性、高性能、列存储、可伸缩、多版本的 NoSQL 的分布式数据存储系统，实现对大型数据的实时、随机的读写访问。

HBase 依赖于 HDFS 做底层的数据存储，BigTable 依赖 Google GFS 做数据存储

HBase 依赖于 MapReduce 做数据计算，BigTable 依赖 Google MapReduce 做数据计算

HBase 依赖于 ZooKeeper 做服务协调，BigTable 依赖 Google Chubby 做服务协调

NoSQL = NO SQL

NoSQL = Not Only SQL：会有一些把 NoSQL 数据的原生查询语句封装成 SQL

比如 HBase 就有 Phoenix 工具

// 关系型数据库 和 非关系型数据库的典型代表：

NoSQL: hbase, redis, mongodb

RDBMS: mysql,oracle,sql server,db2

以下五点是 HBase 这个 NoSQL 数据库的要点：

- ① 它介于 NoSQL 和 RDBMS 之间，**仅能通过主键(rowkey)和主键的 range 来检索数据**
- ② HBase 查询数据功能很简单，**不支持 join 等复杂操作**
- ③ 不支持复杂的事务，**只支持行级事务**(可通过 hive 支持来实现多表 join 等复杂操作)。

Stay hungry Stay foolish -- <http://blog.csdn.net/zhongqi2513>

④ HBase 中支持的数据类型: **byte[]** (底层所有数据的存储都是字节数组)

⑤ 主要用来存储结构化和半结构化的松散数据。

结构化: 数据结构字段含义确定, 清晰, 典型的如数据库中的表结构

半结构化: 具有一定结构, 但语义不够确定, 典型的如 HTML 网页, 有些字段是确定的(title), 有些不确定(table)

非结构化: 杂乱无章的数据, 很难按照一个概念去进行抽取, 无规律性

与 Hadoop 一样, HBase 目标主要依靠横向扩展, 通过不断增加廉价的商用服务器, 来增加计算和存储能力。

HBase 中的表特点

- 1、**大:** 一个表可以有上十亿行, 上百万列
- 2、**面向列:** 面向列(族)的存储和权限控制, 列(簇)独立检索。
- 3、**稀疏:** 对于为空(null)的列, 并不占用存储空间, 因此, 表可以设计的非常稀疏。
- 4、**无模式:** 每行都有一个可排序的主键和任意多的列, 列可以根据需要动态的增加, 同一张表中不同的行可以有截然不同的列

1.3、表结构逻辑视图

HBase 以表的形式存储数据。表由行和列组成。列划分为若干个列簇 (Column Family)

HBase表结构:	rowkey	列簇: baseinfo	列簇: courseinfo	查数据流程:
rowkey: 字典排序	rk001	name:huangbo	math: 95	表
列簇: 包含一组列, 列在插入数据时指定, 列簇在建表时指定	rk002	name: xuzheng age: 35	english: 100	↓ rowkey
列: 一个列簇中会有多个列, 并且可以不同	rk003	name: wangbaoqiang age: 18 telephone:13122334455 TS1 telephone:13388779966 TS2 telephone:14485749632 TS3	algorithm: 74	↓ 列簇
时间戳: 每个列的值可以存多个版本的值, 版本号就是时间戳, 按照时间戳由近到远排序				↓ 列
				↓ 时间戳

1.3.1、行键 (RowKey)

与 NoSQL 数据库们一样, rowkey 是用来检索记录的主键。访问 HBase Table 中的行, 只有三种方式:

- 1、通过单个 row key 访问
- 2、通过 row key 的 range
- 3、全表扫描

rowkey 行键可以是任意字符串(最大长度是 **64KB**, 实际应用中长度一般为 10-100bytes), 最

好是 16。在 HBase 内部，rowkey 保存为字节数组。**HBase 会对表中的数据按照 rowkey 排序(字典顺序)**

存储时，数据按照 rowkey 的字典序(byte order)排序存储。设计 key 时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

注意：

字典序对 int 排序的结果是

1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,...,9,91,92,93,94,95,96,97,98,99。

要保持整形的自然序，行键必须用 0 作左填充。

行的一次读写是原子操作(不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

1.3.2、列簇（Column Family）

HBase 表中的每个列，都归属与某个列簇。列簇是表的 Schema 的一部分(而列不是)，必须在使用表之前定义好，而且定义好了之后就不能更改。

列名都以列簇作为前缀。例如 courses:history, courses:math 都属于 courses 这个列簇。访问控制、磁盘和内存的使用统计等都是在列簇层面进行的。

列簇越多，在取一行数据时所要参与 IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列簇（最好就一个列簇）

1.3.3、时间戳（TimeStamp）

HBase 中通过 rowkey 和 columns 确定的为一个存储单元称为 cell。每个 cell 都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 hbase(在数据写入时自动)赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的的管理(包括存贮和索引)负担，hbase 提供了两种数据版本回收方式：

保存数据的最后 n 个版本

保存最近一段时间内的版本（设置数据的生命周期 TTL）。

用户可以针对每个列簇进行设置。

1.3.4、单元格（Cell）

由{rowkey, column(=<family> + <column>), version} 唯一确定的单元。

Cell 中的数据是没有类型的，全部是字节码形式存贮。

1.4、HBase 应用场景

1、半结构化或非结构化数据

对于数据结构字段不够确定或杂乱无章很难按一个概念去进行抽取的数据适合用 HBase。而且 HBase 是面向列的，HBase 支持动态增加字段

2、记录非常稀疏

RDBMS 的行有多少列是固定的，为 null 的列浪费了存储空间。而 HBase 为 null 的 Column 是会被存储的，这样既节省了空间又提高了读性能。

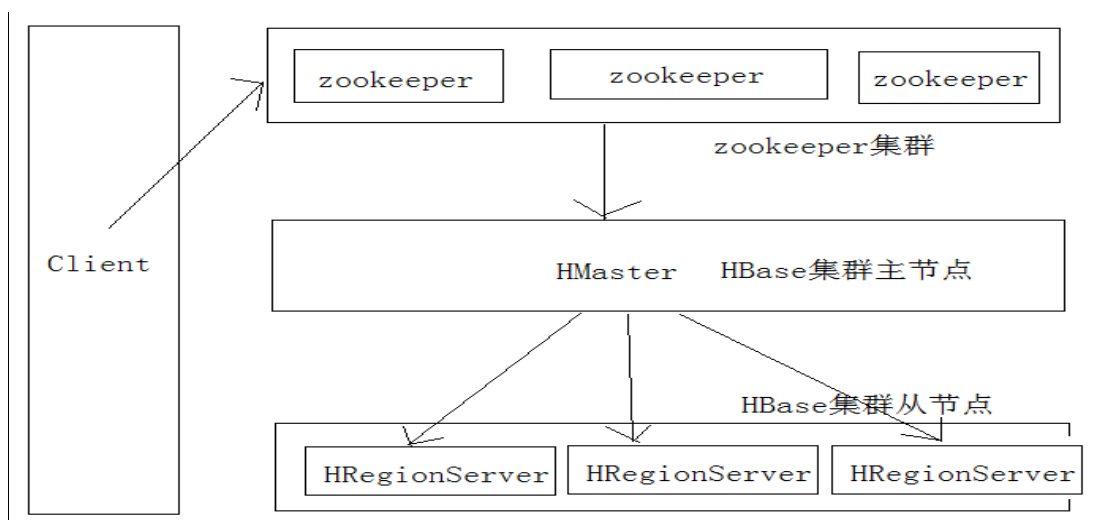
3、多版本数据

对于需要存储变动历史记录的数据，使用 HBase 就再合适不过了。HBase 根据 Row key 和 Column key 定位到的 Value 可以有任意数量的版本值。

4、超大数据量

当数据量越来越大，RDBMS 数据库撑不住了，就出现了读写分离策略，通过一个 Master 专门负责写操作，多个 Slave 负责读操作，服务器成本倍增。随着压力增加，Master 撑不住了，这时就要分库了，把关联不大的数据分开部署，一些 join 查询不能用了，需要借助中间层。随着数据量的进一步增加，一个表的记录越来越大，查询就变得很慢，于是又得搞分表，比如按 ID 取模分成多个表以减少单个表的记录数。经历过这些事的人都知道过程是多么的折腾。采用 HBase 就简单了，只需要加机器即可，HBase 会自动水平切分扩展，跟 Hadoop 的无缝集成保障了其数据可靠性（HDFS）和海量数据分析的高性能（MapReduce）。

2、hbase 集群结构



region: 是 hbase 中对表进行切割的单元，由 regionserver 负责管理

hamster: hbase 的主节点，负责整个集群的状态感知，负载分配、负责用户表的元数据(schema)

管理（可以配置多个用来实现 HA），hmaster 负载压力相对于 hdfs 的 namenode 会小很多

regionserver: hbase 中真正负责管理 region 的服务器，也就是负责为客户端进行表数据读写的服务器每一台 regionserver 会管理很多的 region，同一个 regionserver 上面管理的所有的 region 不属于同一张表

zookeeper: 整个 hbase 中的主从节点协调，主节点之间的选举，集群节点之间的上下线感知……都是通过 zookeeper 来实现

HDFS: 用来存储 hbase 的系统文件，或者表的 region

3、HBase 和 Hive 的比较

3.1、相同点

1、HBase 和 Hive 都是架构在 Hadoop 之上，用 HDFS 做底层的数据存储，用 MapReduce 做数据计算

3.2、不同点

- 1、Hive 是建立在 Hadoop 之上为了降低 MapReduce 编程复杂度的 ETL 工具。
HBase 是为了弥补 Hadoop 对实时操作的缺陷
- 2、Hive 表是纯逻辑表，因为 Hive 的本身并不能做数据存储和计算，而是完全依赖 Hadoop
HBase 是物理表，提供了一张超大的内存 Hash 表来存储索引，方便查询
- 3、Hive 是数据仓库工具，需要全表扫描，就用 Hive，因为 Hive 是文件存储
HBase 是数据库，需要索引访问，则用 HBase，因为 HBase 是面向列的 NoSQL 数据库
- 4、Hive 表中存入数据（文件）时不做校验，属于读模式存储系统
HBase 表插入数据时，会和 RDBMS 一样做 Schema 校验，所以属于写模式存储系统
- 5、Hive 不支持单行记录操作，数据处理依靠 MapReduce，操作延时高
HBase 支持单行记录的 CRUD，并且是实时处理，效率比 Hive 高得多

4、hbase 集群搭建

4.1、安装步骤

- 1、安装 zookeeper 集群，此处略
- 2、找到官网下载 hbase 安装包 hbase-1.2.6-bin.tar.gz，这里给大家提供一个下载地址：
<http://mirrors.hust.edu.cn/apache/hbase/>

对应版本的官方文档: <http://hbase.apache.org/1.2/book.html>

3、上传安装包到服务器，并解压到对应的安装目录

```
[hadoop@hadoop02 apps]# tar -zxvf hbase-1.2.6-bin.tar.gz -C /home/hadoop/apps/
```

4、修改配置文件

1、[hadoop@hadoop02 conf]# vi hbase-env.sh

修改两个地方:

export JAVA_HOME=/usr/local/java/jdk1.8.0_73, 表示修改为自己的 jdk 目录

export HBASE_MANAGES_ZK=false, 表示不引用 hbase 自带的 zookeeper, 用我们自己安装的

保存退出

2、vi hbase-site.xml

增加以下配置:

```
<configuration>
```

```
  <property>
```

```
    <!-- 指定 hbase 在 HDFS 上存储的路径 -->
```

```
    <name>hbase.rootdir</name>
```

```
    <value>hdfs://myha01/hbase</value>
```

```
  </property>
```

```
  <property>
```

```
    <!-- 指定 hbase 是分布式的 -->
```

```
    <name>hbase.cluster.distributed</name>
```

```
    <value>true</value>
```

```
  </property>
```

```
  <property>
```

```
    <!-- 指定 zk 的地址，多个用 “,” 分割 -->
```

```
    <name>hbase.zookeeper.quorum</name>
```

```
    <value>hadoop03:2181,hadoop04:2181,hadoop05:2181</value>
```

```
  </property>
```

```
</configuration>
```

保存退出

3、修改 regionservers

vi regionservers

```
hadoop02
```

```
hadoop03
```

```
hadoop04
```

```
hadoop05
```

4、修改 backup-masters (自行创建), 指定备用的主节点

该文件是不存在的, 先自行创建: vi backup-masters

```
hadoop05
```


5、最重要一步，要把 **hadoop** 的 **hdfs-site.xml** 和 **core-site.xml** 放到 **hbase-1.2.6/conf** 下

```
cp ~/apps/hadoop-2.7.5/etc/hadoop/core-site.xml ~/apps/hbase-1.2.6/conf/
cp ~/apps/hadoop-2.7.5/etc/hadoop/hdfs-site.xml ~/apps/hbase-1.2.6/conf/
```

5、分发安装到各节点

```
scp -r hbase-1.2.6 root@hadoop03:/home/hadoop/apps/
scp -r hbase-1.2.6 root@hadoop04:/home/hadoop/apps/
scp -r hbase-1.2.6 root@hadoop05:/home/hadoop/apps/
```

6、别忘了同步时间!!!!!!!

HBase 集群对于时间的同步要求的比 HDFS 严格，所以，集群启动之前千万记住要进行时间同步，要求相差不要超过 30s

7、配置环境变量

vi ~/.bashrc

添加两行:

export HBASE_HOME=/home/hadoop/apps/hbase-1.2.6

export PATH=\$PATH:\$HBASE_HOME/bin

保存退出!!! 别忘了执行 source ~/.bashrc，使配置生效

8、启动（顺序别搞错了）

1、先启动 zookeeper 集群

zkServer.sh start

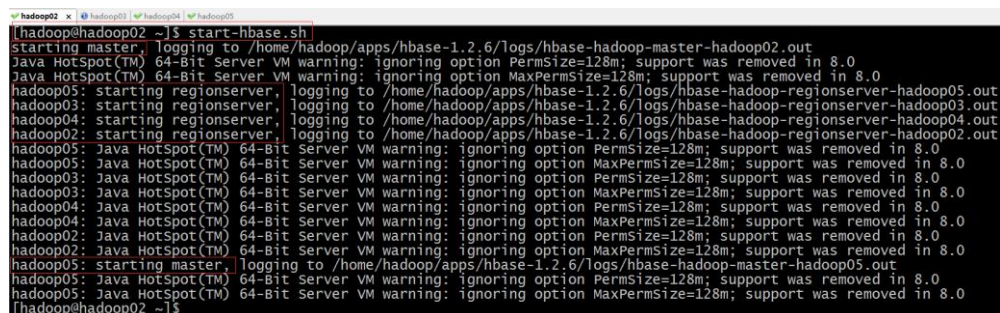
2、启动 hdfs 集群

start-dfs.sh

3、启动 hbase

保证 ZooKeeper 集群和 HDFS 集群启动正常的情况下启动 HBase 集群

启动命令: start-hbase.sh



```
[hadoop@hadoop02 ~]$ start-hbase.sh
starting master, logging to /home/hadoop/apps/hbase-1.2.6/logs/hbase-hadoop-master-hadoop02.out
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
hadoop05: starting regionserver, logging to /home/hadoop/apps/hbase-1.2.6/logs/hbase-hadoop-regionserver-hadoop05.out
hadoop03: starting regionserver, logging to /home/hadoop/apps/hbase-1.2.6/logs/hbase-hadoop-regionserver-hadoop03.out
hadoop04: starting regionserver, logging to /home/hadoop/apps/hbase-1.2.6/logs/hbase-hadoop-regionserver-hadoop04.out
hadoop02: starting regionserver, logging to /home/hadoop/apps/hbase-1.2.6/logs/hbase-hadoop-regionserver-hadoop02.out
hadoop05: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
hadoop03: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
hadoop04: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
hadoop03: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
hadoop04: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
hadoop02: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
hadoop05: starting master, logging to /home/hadoop/apps/hbase-1.2.6/logs/hbase-hadoop-master-hadoop05.out
hadoop05: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
hadoop05: Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
[hadoop@hadoop02 ~]$
```

观看启动日志可以看到:

- 1、首先在命令执行节点启动 master
- 2、然后分别在 hadoop02,hadoop03,hadoop04,hadoop05 启动 regionserver
- 3、然后在 backup-masters 文件中配置的备节点上再启动一个 master 主进程

9、查看启动是否正常，是否成功

1、检查各进程是否启动正常

主节点和备用节点都启动 hmaster 进程

各从节点都启动 hregionserver 进程

```

hadoop02 x hadoop03 x hadoop04 x hadoop05
[hadoop02@hadoop02 ~]$ jps
17441 Jps
8497 QuorumPeerMain
16946 HMaster
17079 HRegionServer
11399 DataNode
9415 JobHistoryServer
11496 JournalNode
13081 NodeManager
11628 DFSZKFailoverController
11324 NameNode
[hadoop02@hadoop02 ~]$

hadoop02 x hadoop03 x hadoop04 x hadoop05
[hadoop03@hadoop03 ~]$ jps
16752 HRegionServer
17014 Jps
8969 DataNode
9066 JournalNode
12060 NodeManager
5261 QuorumPeerMain
8894 NameNode
9199 DFSZKFailoverController
[hadoop03@hadoop03 ~]$

hadoop02 x hadoop03 x hadoop04 x hadoop05
[hadoop04@hadoop04 ~]$ jps
13365 HRegionServer
8869 DataNode
8966 JournalNode
10042 NodeManager
13579 Jps
5660 QuorumPeerMain
9932 ResourceManager
[hadoop04@hadoop04 ~]$

hadoop02 x hadoop03 x hadoop04 x hadoop05
[hadoop05@hadoop05 ~]$ jps
10000 NodeManager
10145 ResourceManager
8535 DataNode
13067 HMaster
6795 QuorumPeerMain
12959 HRegionServer
13343 Jps
[hadoop05@hadoop05 ~]$

```

按照对应的配置信息各个节点应该要启动的进程如上图所示

2、通过访问浏览器页面，格式为“主节点：16010”

<http://hadoop02:16010/>

10、如果有节点相应的进程没有启动，那么可以手动启动

hbase-daemon.sh start master

hbase-daemon.sh start regionserver

5、hbase 命令行演示

概述：大数据生态里的各种软件，基本都会给出 shell 命令行操作。所以在拿到新上手的软件的时候先找到怎么进入命令行，然后相应要想到 help 命令，查看命令帮助。别着急一股脑儿扎进去敲各种命令，这就是思路，思路很重要。

下面按照我的思路来进行练习：

1、先进入 hbase shell 命令行

在你安装的随意台服务器节点上，执行命令：hbase shell，会进入到你的 hbase shell 客户端

```
[root@hadoop01 ~]# hbase shell
```

```

hadoop02 x | hadoop03 | hadoop04 | hadoop05
[hadoop@hadoop02 ~]$ hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/hadoop/apps/hbase-1.2.6/lib/slf4j-log4j12-1.7.
SLF4J: Found binding in [jar:file:/home/hadoop/apps/hadoop-2.7.5/share/hadoop/common/
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.6, rUnknown, Mon May 29 02:25:32 CDT 2017

hbase(main):001:0>

```

2、进入之后先别着急，先看一下提示。其实是不是有一句很重要的话：

HBase Shell; enter 'help<RETURN>' for list of supported commands.

Type "exit<RETURN>" to leave the HBase Shell

意在告诉怎么获得帮助，怎么退出客户端

help 获取帮助

help 获取所有命令提示

help "dml" 获取一组命令的提示

help "put" 获取一个单独命令的提示帮助

exit 退出 hbase shell 客户端

3、下面真正进入命令的操作演示

显示 hbase 中的表

list

第一个表名，多个列簇

```
create 'user_info',{NAME=>'base_info',VERSIONS=>3},{NAME=>'extra_info',VERSIONS=>1 }
```

```
put 'user_info', 'user0000', 'base_info:name', 'luoyufeng'
```

```
put 'user_info', 'user0000', 'base_info:age', '18'
```

```
put 'user_info', 'user0000', 'base_info:gender', 'female'
```

```
put 'user_info', 'user0000', 'extra_info:size', '34'
```

```
get 'user_info', 'user0000'
```

```
create 'user_info',{NAME=>'base_info'},{NAME=>'extra_info'}
```

```
put 'user_info', 'user0001', 'base_info:name', 'zhangsan1'
```

```
put 'user_info', 'zhangsan_20150701_0001', 'base_info:name', 'zhangsan1'
```

```
put 'user_info', 'zhangsan_20150701_0002', 'base_info:name', 'zhangsan2'
```

```
put 'user_info', 'zhangsan_20150701_0003', 'base_info:name', 'zhangsan3'
```

```
put 'user_info', 'zhangsan_20150701_0004', 'base_info:name', 'zhangsan4'
```

```
put 'user_info', 'zhangsan_20150701_0005', 'base_info:name', 'zhangsan5'
```

```
put 'user_info', 'zhangsan_20150701_0006', 'base_info:name', 'zhangsan6'
```

```
put 'user_info', 'zhangsan_20150701_0007', 'base_info:name', 'zhangsan7'
```

```
put 'user_info', 'zhangsan_20150701_0008', 'base_info:name', 'zhangsan8'
```

```
put 'user_info', 'zhangsan_20150701_0001', 'base_info:age', '21'  
put 'user_info', 'zhangsan_20150701_0002', 'base_info:age', '22'  
put 'user_info', 'zhangsan_20150701_0003', 'base_info:age', '23'  
put 'user_info', 'zhangsan_20150701_0004', 'base_info:age', '24'  
put 'user_info', 'zhangsan_20150701_0005', 'base_info:age', '25'  
put 'user_info', 'zhangsan_20150701_0006', 'base_info:age', '26'  
put 'user_info', 'zhangsan_20150701_0007', 'base_info:age', '27'  
put 'user_info', 'zhangsan_20150701_0008', 'base_info:age', '28'
```

```
put 'user_info', 'zhangsan_20150701_0001', 'extra_info:Hobbies', 'music'  
put 'user_info', 'zhangsan_20150701_0002', 'extra_info:Hobbies', 'sport'  
put 'user_info', 'zhangsan_20150701_0003', 'extra_info:Hobbies', 'music'  
put 'user_info', 'zhangsan_20150701_0004', 'extra_info:Hobbies', 'sport'  
put 'user_info', 'zhangsan_20150701_0005', 'extra_info:Hobbies', 'music'  
put 'user_info', 'zhangsan_20150701_0006', 'extra_info:Hobbies', 'sport'  
put 'user_info', 'zhangsan_20150701_0007', 'extra_info:Hobbies', 'music'
```

```
put 'user_info', 'baiyc_20150716_0001', 'base_info:name', 'baiyc1'  
put 'user_info', 'baiyc_20150716_0002', 'base_info:name', 'baiyc2'  
put 'user_info', 'baiyc_20150716_0003', 'base_info:name', 'baiyc3'  
put 'user_info', 'baiyc_20150716_0004', 'base_info:name', 'baiyc4'  
put 'user_info', 'baiyc_20150716_0005', 'base_info:name', 'baiyc5'  
put 'user_info', 'baiyc_20150716_0006', 'base_info:name', 'baiyc6'  
put 'user_info', 'baiyc_20150716_0007', 'base_info:name', 'baiyc7'  
put 'user_info', 'baiyc_20150716_0008', 'base_info:name', 'baiyc8'
```

```
put 'user_info', 'baiyc_20150716_0001', 'base_info:age', '21'  
put 'user_info', 'baiyc_20150716_0002', 'base_info:age', '22'  
put 'user_info', 'baiyc_20150716_0003', 'base_info:age', '23'  
put 'user_info', 'baiyc_20150716_0004', 'base_info:age', '24'  
put 'user_info', 'baiyc_20150716_0005', 'base_info:age', '25'  
put 'user_info', 'baiyc_20150716_0006', 'base_info:age', '26'  
put 'user_info', 'baiyc_20150716_0007', 'base_info:age', '27'  
put 'user_info', 'baiyc_20150716_0008', 'base_info:age', '28'
```

```
put 'user_info', 'baiyc_20150716_0001', 'extra_info:Hobbies', 'music'  
put 'user_info', 'baiyc_20150716_0002', 'extra_info:Hobbies', 'sport'  
put 'user_info', 'baiyc_20150716_0003', 'extra_info:Hobbies', 'music'  
put 'user_info', 'baiyc_20150716_0004', 'extra_info:Hobbies', 'sport'  
put 'user_info', 'baiyc_20150716_0005', 'extra_info:Hobbies', 'music'  
put 'user_info', 'baiyc_20150716_0006', 'extra_info:Hobbies', 'sport'  
put 'user_info', 'baiyc_20150716_0007', 'extra_info:Hobbies', 'music'  
put 'user_info', 'baiyc_20150716_0008', 'extra_info:Hobbies', 'sport'
```

```
put 'user_info', 'rk0001', 'base_info:name', 'luoyufeng'
put 'user_info', 'rk0001', 'base_info:name', 'zhangsang'
get 'user_info', 'rk0001'

scan 'user_info', {COLUMNS=> 'base_info'}
scan 'user_info', {COLUMNS=> 'extra_info'}
```

创建 user 表，包含 info、data 两个列簇

```
create 'user', 'info1', 'data1'
create 'user', {NAME => 'info', VERSIONS => '3'}
```

向 user 表中插入信息，row key 为 rk0001，列簇 info 中添加 name 列标示符，值为 zhangsan

```
put 'user', 'rk0001', 'info:name', 'zhangsan'
```

向 user 表中插入信息，row key 为 rk0001，列簇 info 中添加 gender 列标示符，值为 female

```
put 'user', 'rk0001', 'info:gender', 'female'
```

向 user 表中插入信息，row key 为 rk0001，列簇 info 中添加 age 列标示符，值为 20

```
put 'user', 'rk0001', 'info:age', 20
```

向 user 表中插入信息，row key 为 rk0001，列簇 data 中添加 pic 列标示符，值为 picture

```
put 'user', 'rk0001', 'data:pic', 'picture'
```

获取 user 表中 row key 为 rk0001 的所有信息

```
get 'user', 'rk0001'
```

获取 user 表中 row key 为 rk0001, info 列簇的所有信息

```
get 'user', 'rk0001', 'info'
```

获取 user 表中 row key 为 rk0001, info 列簇的 name、age 列标示符的信息

```
get 'user', 'rk0001', 'info:name', 'info:age'
```

获取 user 表中 row key 为 rk0001, info、data 列簇的信息

```
get 'user', 'rk0001', 'info', 'data'
```

```
get 'user', 'rk0001', {COLUMN => ['info', 'data']}
```

```
get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}
```

获取 user 表中 row key 为 rk0001，列簇为 info，版本号最新 5 个的信息

```
get 'user', 'rk0001', {COLUMN => 'info', VERSIONS => 2}
```

```
get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5}
```

```
get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5, TIMERANGE =>
[1392368783980, 1392380169184]}
```

获取 user 表中 row key 为 rk0001，cell 的值为 zhangsan 的信息

```
get 'people', 'rk0001', {FILTER => "ValueFilter(=, 'binary:图片')"}

```

获取 user 表中 row key 为 rk0001，列标示符中含有 a 的信息

```
get 'people', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}
```

```
put 'user', 'rk0002', 'info:name', 'fanbingbing'
```

```
put 'user', 'rk0002', 'info:gender', 'female'
```

```
put 'user', 'rk0002', 'info:nationality', '中国'
```

```
get 'user', 'rk0002', {FILTER => "ValueFilter(=, 'binary:中国')"} 
```

查询 user 表中的所有信息

```
scan 'user'
```

查询 user 表中列簇为 info 的信息

```
scan 'user', {COLUMNS => 'info'}
```

```
scan 'user', {COLUMNS => 'info', RAW => true, VERSIONS => 5}
```

```
scan 'persion', {COLUMNS => 'info', RAW => true, VERSIONS => 3}
```

Scan 时可以设置是否开启 Raw 模式，开启 Raw 模式会返回包括已添加删除标记但是未实际删除的数据。

查询 user 表中列簇为 info 和 data 的信息

```
scan 'user', {COLUMNS => ['info', 'data']}
```

```
scan 'user', {COLUMNS => ['info:name', 'data:pic']}
```

查询 user 表中列簇为 info、列标示符为 name 的信息

```
scan 'user', {COLUMNS => 'info:name'}
```

查询 user 表中列簇为 info、列标示符为 name 的信息,并且版本最新的 5 个

```
scan 'user', {COLUMNS => 'info:name', VERSIONS => 5}
```

查询 user 表中列簇为 info 和 data 且列标示符中含有 a 字符的信息

```
scan 'user', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring:a'))"}
```

查询 user 表中列簇为 info，rk 范围是[rk0001, rk0003)的数据

```
scan 'people', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}
```

查询 user 表中 row key 以 rk 字符开头的

```
scan 'user',{FILTER=>"PrefixFilter('rk')"} 
```

查询 user 表中指定范围的数据

```
scan 'user', {TIMERANGE => [1392368783980, 1392380169184]}
```

删除数据

删除 user 表 row key 为 rk0001，列标示符为 info:name 的数据

```
delete 'people', 'rk0001', 'info:name'
```

删除 user 表 row key 为 rk0001，列标示符为 info:name，timestamp 为 1392383705316 的数据

```
delete 'user', 'rk0001', 'info:name', 1392383705316
```

清空 user 表中的数据

```
truncate 'people'
```

修改表结构

首先停用 user 表（新版本不用）

```
disable 'user'
```

添加两个列簇 f1 和 f2

```
alter 'people', NAME => 'f1'
```

```
alter 'user', NAME => 'f2'
```

启用表

```
enable 'user'
```

###disable 'user'(新版本不用)

删除一个列簇：

```
alter 'user', NAME => 'f1', METHOD => 'delete' 或 alter 'user', 'delete' => 'f1'
```

添加列簇 f1 同时删除列簇 f2

```
alter 'user', {NAME => 'f1'}, {NAME => 'f2', METHOD => 'delete'}
```

将 user 表的 f1 列簇版本号改为 5

```
alter 'people', NAME => 'info', VERSIONS => 5
```

启用表

```
enable 'user'
```

删除记录

```
delete 'person', 'rk0001', 'info:name'
```

删除字段

```
delete 'person', 'rk0001', 'info:name'
```

删除表

```
disable 'user'
```

```
drop 'user'
```

过滤器

```
get 'person', 'rk0001', {FILTER => "ValueFilter(=, 'binary:中国')"} 
```

```

get 'person', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}
scan 'person', {COLUMNS => 'info:name'}
scan 'person', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring:a'))"}
scan 'person', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}

scan 'person', {COLUMNS => 'info', STARTROW => '20140201', ENDROW => '20140301'}
scan 'person', {COLUMNS => 'info:name', TIMERANGE => [1395978233636,
1395987769587]}

alter 'person', NAME => 'ffff'
alter 'person', NAME => 'info', VERSIONS => 10

get 'user', 'rk0002', {COLUMN => ['info:name', 'data:pic']}

```

6、Hbase Java API 代码开发

几个主要 Hbase API 类和数据模型之间的对应关系：

java 类	HBase 数据模型
HBaseAdmin	数据库 (DataBase)
HBaseConfiguration	
HTable	表 (Table)
HTableDescriptor	列簇 (Column Family)
HColumnDescriptor	
Put	列修饰符 (Column Qualifier)
Get	
Delete	
Result	
Scan	
ResultScanner	

1、HBaseAdmin

关系：org.apache.hadoop.hbase.client.HBaseAdmin

作用：提供了一个接口来管理 HBase 数据库的表信息。它提供的方法包括：创建表，删除表，列出表项，使表有效或无效，以及添加或删除表列簇成员等。

返回值	函数	描述
void	addColumn(String tableName, HColumnDescriptor column)	向一个已经存在的表添加列

	checkHBaseAvailable(HBaseConfiguration conf)	静态函数，查看 HBase 是否处于运行状态
	createTable(HTableDescriptor desc)	创建一个表，同步操作
	deleteTable(byte[] tableName)	删除一个已经存在的表
	enableTable(byte[] tableName)	使表处于有效状态
	disableTable(byte[] tableName)	使表处于无效状态
HTableDescriptor[]	listTables()	列出所有用户控件表项
void	modifyTable(byte[] tableName, HTableDescriptor htd)	修改表的模式，是异步的操作，可能需要花费一定的时间
boolean	tableExists(String tableName)	检查表是否存在

用法示例：

```
HBaseAdmin admin = new HBaseAdmin(config);
admin.disableTable("tablename")
```

2、HBaseConfiguration

关系：org.apache.hadoop.hbase.HBaseConfiguration

作用：对 HBase 进行配置

返回值	函数	描述
void	addResource(Path file)	通过给定的路径所指的文件来添加资源
void	clear()	清空所有已设置的属性
string	get(String name)	获取属性名对应的值
String	getBoolean(String name, boolean defaultValue)	获取为 boolean 类型的属性值，如果其属性值类型部位 boolean,则返回默认属性值
void	set(String name, String value)	通过属性名来设置值
void	setBoolean(String name, boolean value)	设置 boolean 类型的属性值

用法示例：

```
HBaseConfiguration hconfig = new HBaseConfiguration();
hconfig.set("hbase.zookeeper.property.clientPort","2181");
```

该方法设置了"hbase.zookeeper.property.clientPort"的端口号为 2181。一般情况下，HBaseConfiguration 会使用构造函数进行初始化，然后在使用其他方法。

3、HTableDescriptor

关系：org.apache.hadoop.hbase.HTableDescriptor

作用：包含了表的名字及其对应表的列簇

返回值	函数	描述
void	addFamily(HColumnDescriptor)	添加一个列簇

HColumnDescriptor	removeFamily(byte[] column)	移除一个列簇
byte[]	getName()	获取表的名字
byte[]	getValue(byte[] key)	获取属性的值
void	setValue(String key, String value)	设置属性的值

用法示例：

```
HTableDescriptor htd = new HTableDescriptor(table);
htd.addFamily(new HColumnDescriptor("family"));
```

在上述例子中，通过一个 HColumnDescriptor 实例，为 HTableDescriptor 添加了一个列簇：family

4、HColumnDescriptor

关系：org.apache.hadoop.hbase.HColumnDescriptor

作用：维护着关于列簇的信息，例如版本号，压缩设置等。它通常在创建表或者为表添加列簇的时候使用。列簇被创建后不能直接修改，只能通过删除然后重新创建的方式。列簇被删除的时候，列簇里面的数据也会同时被删除。

返回值	函数	描述
byte[]	getName()	获取列簇的名字
byte[]	getValue(byte[] key)	获取对应的属性的值
void	setValue(String key, String value)	设置对应属性的值

用法示例：

```
HTableDescriptor htd = new HTableDescriptor(tablename);
HColumnDescriptor col = new HColumnDescriptor("content:");
htd.addFamily(col);
```

此例添加了一个 content 的列簇

5、HTable

关系：org.apache.hadoop.hbase.client.HTable

作用：可以用来和 HBase 表直接通信。此方法对于更新操作来说是非线程安全的。

返回值	函数	描述
void	checkAndPut(byte[] row, byte[] family, byte[] qualifier, byte[] value, Put put)	自动的检查 row/family/qualifier 是否与给定的值匹配
void	close()	释放所有的资源或挂起内部缓冲区中的更新
Boolean	exists(Get get)	检查 Get 实例所指定的值是否存在于 HTable 的列中
Result	get(Get get)	获取指定行的某些单元格所对

		应的值
byte[][]	getEndKeys()	获取当前一打开的表每个区域的结束键值
ResultScanner	getScanner(byte[] family)	获取当前给定列簇的 scanner 实例
HTableDescriptor	getTableDescriptor()	获取当前表的 HTableDescriptor 实例
byte[]	getTableName()	获取表名
static boolean	isTableEnabled(HBaseConfiguration conf, String tableName)	检查表是否有效
void	put(Put put)	向表中添加值

用法示例:

```
HTable table = new HTable(conf, Bytes.toBytes(tablename));
ResultScanner scanner = table.getScanner(family);
```

6、Put

关系: org.apache.hadoop.hbase.client.Put

作用: 用来对单个行执行添加操作

返回值	函数	描述
Put	add(byte[] family, byte[] qualifier, byte[] value)	将指定的列和对应的值添加到 Put 实例中
Put	add(byte[] family, byte[] qualifier, long ts, byte[] value)	将指定的列和对应的值及时间戳添加到 Put 实例中
byte[]	getRow()	获取 Put 实例的行
RowLock	getRowLock()	获取 Put 实例的行锁
long	getTimeStamp()	获取 Put 实例的时间戳
boolean	isEmpty()	检查 familyMap 是否为空
Put	setTimeStamp(long timeStamp)	设置 Put 实例的时间戳

用法示例:

```
HTable table = new HTable(conf, Bytes.toBytes(tablename));
Put p = new Put(brow); //为指定行创建一个 Put 操作
p.add(family, qualifier, value);
table.put(p);
```

7、Get

关系: org.apache.hadoop.hbase.client.Get

作用: 用来获取单个行的相关信息

返回值	函数	描述
Get	addColumn(byte[] family, byte[] qualifier)	获取指定列簇和列修饰符对应的列
Get	addFamily(byte[] family)	通过指定的列簇获取其对应列的所有列
Get	setTimeRange(long minStamp, long maxStamp)	获取指定取件的列的版本号
Get	setFilter(Filter filter)	当执行 Get 操作时设置服务器端的过滤器

用法示例:

```
HTable table = new HTable(conf, Bytes.toBytes(tablename));
Get g = new Get(Bytes.toBytes(row));
```

8、Result

关系: org.apache.hadoop.hbase.client.Result

作用: 存储 Get 或者 Scan 操作后获取表的单行值。使用此类提供的方法可以直接获取值或者各种 Map 结构 (key-value 对)

返回值	函数	描述
boolean	containsColumn(byte[] family, byte[] qualifier)	检查指定的列是否存在
NavigableMap<byte[], byte[]>	getFamilyMap(byte[] family)	获取对应列簇所包含的修饰符与值的键值对
byte[]	getValue(byte[] family, byte[] qualifier)	获取对应列的最新值

9、Scan

10、ResultScanner

11、Delete

6.1、基本增删改查实现

```
package com.ghgj.hbase;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Delete;
```

```
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;

public class HbaseDemoTest {
    // 声明静态配置
    static Configuration conf = null;

    private static final String ZK_CONNECT_STR =
"hadoop01:2181,hadoop02:2181,hadoop03:2181,hadoop04:2181,hadoop05:2181";

    static {
        conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", ZK_CONNECT_STR);
    }

    /**
     * 创建表
     * @tableName 表名
     * @family 列簇列表
     */
    public static void creatTable(String tableName, String[] family) throws Exception {
        HBaseAdmin admin = new HBaseAdmin(conf);
        HTableDescriptor desc = new HTableDescriptor(tableName);
        for (int i = 0; i < family.length; i++) {
            desc.addFamily(new HColumnDescriptor(family[i]));
        }
        if (admin.tableExists(tableName)) {
            System.out.println("table Exists!");
            System.exit(0);
        } else {
            admin.createTable(desc);
            System.out.println("create table Success!");
        }
    }

    /**
     * 为表添加数据（适合知道有多少列簇的固定表）
     * @rowKey rowKey
     */
}
```

```
* @tableName 表名
* @column1 第一个列簇列表
* @value1 第一个列的值的列表
* @column2 第二个列簇列表
* @value2 第二个列的值的列表
*/
public static void addData(String rowKey, String tableName,
    String[] column1, String[] value1, String[] column2, String[] value2)
    throws IOException {
    // 设置 rowkey
    Put put = new Put(Bytes.toBytes(rowKey));

    // HTable 负责跟记录相关的操作如增删改查等//
    HTable table = new HTable(conf, Bytes.toBytes(tableName));

    // 获取所有的列簇
    HColumnDescriptor[] columnFamilies = table.getTableDescriptor().getColumnFamilies();

    for (int i = 0; i < columnFamilies.length; i++) {
        // 获取列簇名
        String familyName = columnFamilies[i].getNameAsString();
        // article 列簇 put 数据
        if (familyName.equals("article")) {
            for (int j = 0; j < column1.length; j++) {
                put.add(Bytes.toBytes(familyName),
                    Bytes.toBytes(column1[j]), Bytes.toBytes(value1[j]));
            }
        }
        // author 列簇 put 数据
        if (familyName.equals("author")) {
            for (int j = 0; j < column2.length; j++) {
                put.add(Bytes.toBytes(familyName),
                    Bytes.toBytes(column2[j]), Bytes.toBytes(value2[j]));
            }
        }
    }
    table.put(put);
    System.out.println("add data Success!");
}

/*
* 根据 rowkey 查询
* @rowKey rowKey
* @tableName 表名
*/
```

```
*/
public static Result getResult(String tableName, String rowKey) throws IOException {
    Get get = new Get(Bytes.toBytes(rowKey));
    HTable table = new HTable(conf, Bytes.toBytes(tableName)); // 获取表
    Result result = table.get(get);
    for (KeyValue kv : result.list()) {
        System.out.println("family:" + Bytes.toString(kv.getFamily()));
        System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));
        System.out.println("value:" + Bytes.toString(kv.getValue()));
        System.out.println("Timestamp:" + kv.getTimestamp());
        System.out.println("-----");
    }
    return result;
}

/*
 * 遍历查询 hbase 表
 * @tableName 表名
 */
public static void getResultScann(String tableName) throws IOException {
    Scan scan = new Scan();
    ResultScanner rs = null;
    HTable table = new HTable(conf, Bytes.toBytes(tableName));
    try {
        rs = table.getScanner(scan);
        for (Result r : rs) {
            for (KeyValue kv : r.list()) {
                System.out.println("row:" + Bytes.toString(kv.getRow()));
                System.out.println("family:" + Bytes.toString(kv.getFamily()));
                System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));
                System.out.println("value:" + Bytes.toString(kv.getValue()));
                System.out.println("timestamp:" + kv.getTimestamp());
                System.out.println("-----");
            }
        }
    } finally {
        rs.close();
    }
}

/*
 * 遍历查询 hbase 表
 * @tableName 表名
 * 切记：包括下界，不包括上界
 */
```



```
*/
public static void getResultScann(String tableName, String start_rowkey,String stop_rowkey)
throws IOException {
    Scan scan = new Scan();
    scan.setStartRow(Bytes.toBytes(start_rowkey));
    scan.setStopRow(Bytes.toBytes(stop_rowkey));
    ResultScanner rs = null;
    HTable table = new HTable(conf, Bytes.toBytes(tableName));
    try {
        rs = table.getScanner(scan);
        for (Result r : rs) {
            for (KeyValue kv : r.list()) {
                System.out.println("row:" + Bytes.toString(kv.getRow()));
                System.out.println("family:" + Bytes.toString(kv.getFamily()));
                System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));
                System.out.println("value:" + Bytes.toString(kv.getValue()));
                System.out.println("timestamp:" + kv.getTimestamp());
                System.out.println("-----");
            }
        }
    } finally {
        rs.close();
    }
}

/*
 * 查询表中的某一列
 * @tableName 表名
 * @rowKey rowKey
 */
public static void getResultByColumn(String tableName, String rowKey,String familyName,
String columnName) throws IOException {
    HTable table = new HTable(conf, Bytes.toBytes(tableName));
    Get get = new Get(Bytes.toBytes(rowKey));

    // 获取指定列簇和列修饰符对应的列
    get.addColumn(Bytes.toBytes(familyName), Bytes.toBytes(columnName));
    Result result = table.get(get);
    for (KeyValue kv : result.list()) {
        System.out.println("family:" + Bytes.toString(kv.getFamily()));
        System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));
        System.out.println("value:" + Bytes.toString(kv.getValue()));
        System.out.println("Timestamp:" + kv.getTimestamp());
        System.out.println("-----");
    }
}
```

```
    }  
}  
  
/*  
 * 更新表中的某一列  
 * @tableName 表名  
 * @rowKey rowKey  
 * @familyName 列簇名  
 * @columnName 列名  
 * @value 更新后的值  
 */  
public static void updateTable(String tableName, String rowKey, String familyName,  
    String columnName, String value) throws IOException {  
    HTable table = new HTable(conf, Bytes.toBytes(tableName));  
    Put put = new Put(Bytes.toBytes(rowKey));  
    put.add(Bytes.toBytes(familyName), Bytes.toBytes(columnName), Bytes.toBytes(value));  
    table.put(put);  
    System.out.println("update table Success!");  
}  
  
/*  
 * 查询某列数据的多个版本  
 * @tableName 表名  
 * @rowKey rowKey  
 * @familyName 列簇名  
 * @columnName 列名  
 */  
public static void getResultByVersion(String tableName, String rowKey, String familyName,  
String columnName) throws IOException {  
    HTable table = new HTable(conf, Bytes.toBytes(tableName));  
    Get get = new Get(Bytes.toBytes(rowKey));  
    get.addColumn(Bytes.toBytes(familyName), Bytes.toBytes(columnName));  
    get.setMaxVersions(5);  
    Result result = table.get(get);  
    for (KeyValue kv : result.list()) {  
        System.out.println("family:" + Bytes.toString(kv.getFamily()));  
        System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));  
        System.out.println("value:" + Bytes.toString(kv.getValue()));  
        System.out.println("Timestamp:" + kv.getTimestamp());  
        System.out.println("-----");  
    }  
}  
  
/*
```

```
* 删除指定的列
* @tableName 表名
* @rowKey rowKey
* @familyName 列簇名
* @columnName 列名
*/
public static void deleteColumn(String tableName, String rowKey,String falilyName,
    String columnName) throws IOException {
    HTable table = new HTable(conf, Bytes.toBytes(tableName));
    Delete deleteColumn = new Delete(Bytes.toBytes(rowKey));
    deleteColumn.deleteColumns(Bytes.toBytes(falilyName), Bytes.toBytes(columnName));
    table.delete(deleteColumn);
    System.out.println(falilyName + ":" + columnName + "is deleted!");
}

/*
* 删除指定的列
* @tableName 表名
* @rowKey rowKey
*/
public static void deleteAllColumn(String tableName, String rowKey) throws IOException {
    HTable table = new HTable(conf, Bytes.toBytes(tableName));
    Delete deleteAll = new Delete(Bytes.toBytes(rowKey));
    table.delete(deleteAll);
    System.out.println("all columns are deleted!");
}

/*
* 删除表
* @tableName 表名
*/
public static void deleteTable(String tableName) throws IOException {
    HBaseAdmin admin = new HBaseAdmin(conf);
    admin.disableTable(tableName);
    admin.deleteTable(tableName);
    System.out.println(tableName + "is deleted!");
}

public static void main(String[] args) throws Exception {
    // 创建表
    /*String tableName = "blog";
    String[] family = { "article", "author" };
    creatTable(tableName, family);*/
```

```
// 为表添加数据
/*String[] column1 = { "title", "content", "tag" };
String[] value1 = {
    "Head First HBase",
    "HBase is the Hadoop database",
    "Hadoop,HBase,NoSQL" };
String[] column2 = { "name", "nickname" };
String[] value2 = { "nicholas", "lee" };
addData("rowkey1", "blog", column1, value1, column2, value2);
addData("rowkey2", "blog", column1, value1, column2, value2);
addData("rowkey3", "blog", column1, value1, column2, value2);*/

// 遍历查询， 根据 row key 范围遍历查询
// getResultScann("blog", "rowkey4", "rowkey5");

// 查询
// getResult("blog", "rowkey1");

// 查询某一列的值
// getResultByColumn("blog", "rowkey1", "author", "name");

// 更新列
// updateTable("blog", "rowkey1", "author", "name", "bin");

// 查询某一列的值
// getResultByColumn("blog", "rowkey1", "author", "name");

// 查询某列的多版本
// getResultByVersion("blog", "rowkey1", "author", "name");

// 删除一列
// deleteColumn("blog", "rowkey1", "author", "nickname");

// 删除所有列
// deleteAllColumn("blog", "rowkey1");

// 删除表
deleteTable("blog");
}
}
```

6.2、过滤器查询

引言：过滤器的类型很多，但是可以分为两大类——**比较过滤器**，**专用过滤器**

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

hbase 过滤器的比较运算符：

```
LESS <
LESS_OR_EQUAL <=
EQUAL =
NOT_EQUAL <>
GREATER_OR_EQUAL >=
GREATER >
NO_OP 排除所有
```

HBase 过滤器的比较器（指定比较机制）：

```
BinaryComparator 按字节索引顺序比较指定字节数组，采用 Bytes.compareTo(byte[])
BinaryPrefixComparator 跟前面相同，只是比较左端的数据是否相同
NullComparator 判断给定的是否为空
BitComparator 按位比较
RegexStringComparator 提供一个正则的比较器，仅支持 EQUAL 和非 EQUAL
SubstringComparator 判断提供的子串是否出现在 value 中。
```

Hbase 过滤器分类

1、比较过滤器

行键过滤器 RowFilter

```
Filter filter1 = new RowFilter(CompareOp.LESS_OR_EQUAL, new
BinaryComparator(Bytes.toBytes("user0000")));
scan.setFilter(filter1);
```

列簇过滤器 FamilyFilter

```
Filter filter1 = new FamilyFilter(CompareOp.LESS, new
BinaryComparator(Bytes.toBytes("base_info")));
scan.setFilter(filter1);
```

列过滤器 QualifierFilter

```
Filter filter = new QualifierFilter(CompareOp.LESS_OR_EQUAL, new
BinaryComparator(Bytes.toBytes("name")));
scan.setFilter(filter1);
```

值过滤器 ValueFilter

```
Filter filter = new ValueFilter(CompareOp.EQUAL, new SubstringComparator("zhangsan"));
scan.setFilter(filter1);
```

时间戳过滤器 TimestampsFilter

```
List<Long> tss = new ArrayList<Long>();  
tss.add(1495398833002l);  
Filter filter1 = new TimestampsFilter(tss);  
scan.setFilter(filter1);
```

2、专用过滤器

单列值过滤器 SingleColumnValueFilter ----会返回满足条件的整行

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(  
    Bytes.toBytes("colfam1"),  
    Bytes.toBytes("col-5"),  
    CompareFilter.CompareOp.NOT_EQUAL,  
    new SubstringComparator("val-5"));  
filter.setFilterIfMissing(true); //如果不设置为 true，则那些不包含指定 column 的行也会  
返回  
scan.setFilter(filter1);
```

单列值排除器 SingleColumnValueExcludeFilter -----返回排除了该列的结果 与上面的结果相反

前缀过滤器 PrefixFilter-----针对行键

```
Filter filter = new PrefixFilter(Bytes.toBytes("row1"));  
scan.setFilter(filter1);
```

列前缀过滤器 ColumnPrefixFilter

```
Filter filter = new ColumnPrefixFilter(Bytes.toBytes("qual2"));  
scan.setFilter(filter1);
```

分页过滤器 PageFilter

```
package com.ghgj.hbase;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.hbase.client.HTable;  
import org.apache.hadoop.hbase.client.Result;  
import org.apache.hadoop.hbase.client.ResultScanner;  
import org.apache.hadoop.hbase.client.Scan;  
import org.apache.hadoop.hbase.filter.BinaryComparator;  
import org.apache.hadoop.hbase.filter.BinaryPrefixComparator;  
import org.apache.hadoop.hbase.filter.ByteArrayComparable;  
import org.apache.hadoop.hbase.filter.ColumnPrefixFilter;  
import org.apache.hadoop.hbase.filter.CompareFilter.CompareOp;
```

```
import org.apache.hadoop.hbase.filter.FamilyFilter;
import org.apache.hadoop.hbase.filter.Filter;
import org.apache.hadoop.hbase.filter.MultipleColumnPrefixFilter;
import org.apache.hadoop.hbase.filter.PageFilter;
import org.apache.hadoop.hbase.filter.PrefixFilter;
import org.apache.hadoop.hbase.filter.QualifierFilter;
import org.apache.hadoop.hbase.filter.RegexStringComparator;
import org.apache.hadoop.hbase.filter.RowFilter;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.filter.SubstringComparator;
import org.apache.hadoop.hbase.util.Bytes;
import org.junit.Test;

public class HbasePageDemo {

    // 声明静态配置
    static Configuration conf = null;

    private static final String ZK_CONNECT_STR =
"hadooop01:2181,hadooop02:2181,hadooop03:2181,hadooop04:2181,hadooop05:2181";

    static {
        conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", ZK_CONNECT_STR);
    }

    public static void main(String[] args) throws Exception {

        String tableName = "testfilter";
        String cfName = "f1";
        final byte[] POSTFIX = new byte[] { 0x00 };
        HTable table = new HTable(conf, tableName);
        Filter filter = new PageFilter(3);
        byte[] lastRow = null;
        int totalRows = 0;
        while (true) {
            Scan scan = new Scan();
            scan.setFilter(filter);
            if(lastRow != null){
                //注意这里添加了 POSTFIX 操作，用来重置扫描边界
                byte[] startRow = Bytes.add(lastRow,POSTFIX);
                scan.setStartRow(startRow);
            }
            ResultScanner scanner = table.getScanner(scan);
```



```
        int localRows = 0;
        Result result;
        while((result = scanner.next()) != null){
            System.out.println(localRows++ + ":" + result);
            totalRows ++;
            lastRow = result.getRow();
        }
        scanner.close();
        if(localRows == 0) break;
    }
    System.out.println("total rows:" + totalRows);
}

/**
 * 多种过滤条件的使用方法
 * @throws Exception
 */
@Test
public void testScan() throws Exception{
    HTable table = new HTable(conf, "person".getBytes());
    Scan scan = new Scan(Bytes.toBytes("person_zhang_000001"),
Bytes.toBytes("person_zhang_000002"));

    //前缀过滤器----针对行键
    Filter filter = new PrefixFilter(Bytes.toBytes("person"));

    //行过滤器 ---针对行键
    ByteArrayComparable rowComparator = new
BinaryComparator(Bytes.toBytes("person_zhang_000001"));
    RowFilter rf = new RowFilter(CompareOp.LESS_OR_EQUAL, rowComparator);

    rf = new RowFilter(CompareOp.EQUAL, new
SubstringComparator("_2016-12-31_"));

    //单值过滤器 1 完整匹配字节数组
    new SingleColumnValueFilter("base_info".getBytes(), "name".getBytes(),
CompareOp.EQUAL, "zhangsan".getBytes());

    //单值过滤器 2 匹配正则表达式
    ByteArrayComparable comparator = new RegexStringComparator("zhang.");
    new SingleColumnValueFilter("info".getBytes(), "NAME".getBytes(),
CompareOp.EQUAL, comparator);

    //单值过滤器 3 匹配是否包含子串,大小写不敏感
```

```
comparator = new SubstringComparator("wu");
new SingleColumnValueFilter("info".getBytes(), "NAME".getBytes(),
CompareOp.EQUAL, comparator);

//键值对元数据过滤-----family 过滤----字节数组完整匹配
FamilyFilter ff = new FamilyFilter(CompareOp.EQUAL ,
new BinaryComparator(Bytes.toBytes("base_info")) //表中不存在
在 inf 列簇，过滤结果为空
);

//键值对元数据过滤-----family 过滤----字节数组前缀匹配
ff = new FamilyFilter(
CompareOp.EQUAL ,
new BinaryPrefixComparator(Bytes.toBytes("inf")) //表中存在以
inf 打头的列簇 info，过滤结果为该列簇所有行
);

//键值对元数据过滤-----qualifier 过滤----字节数组完整匹配
filter = new QualifierFilter(
CompareOp.EQUAL ,
new BinaryComparator(Bytes.toBytes("na")) //表中不存在 na
列，过滤结果为空
);
filter = new QualifierFilter(
CompareOp.EQUAL ,
new BinaryPrefixComparator(Bytes.toBytes("na")) //表中存在以
na 打头的列 name，过滤结果为所有行的该列数据
);

//基于列名(即 Qualifier)前缀过滤数据的 ColumnPrefixFilter
filter = new ColumnPrefixFilter("na".getBytes());

//基于列名(即 Qualifier)多个前缀过滤数据的 MultipleColumnPrefixFilter
byte[][] prefixes = new byte[][] {Bytes.toBytes("na"), Bytes.toBytes("me")};
filter = new MultipleColumnPrefixFilter(prefixes);

//为查询设置过滤条件
scan.setFilter(filter);

scan.addFamily(Bytes.toBytes("base_info"));
//一行
//
Result result = table.get(get);
//多行的数据
ResultScanner scanner = table.getScanner(scan);
```

```
for(Result r : scanner){
    /**
    for(KeyValue kv : r.list()){
        String family = new String(kv.getFamily());
        System.out.println(family);
        String qualifier = new String(kv.getQualifier());
        System.out.println(qualifier);
        System.out.println(new String(kv.getValue()));
    }
    */
    //直接从 result 中取到某个特定的 value
    byte[] value = r.getValue(Bytes.toBytes("base_info"),
Bytes.toBytes("name"));
    System.out.println(new String(value));
}
table.close();
}
```