# HBASE

# 第 1 章NoSQL简介



## 1.1 关系型数据库的查询瓶颈

当用户表的数据达到几千万甚至几亿级别的时候，对单条数据的检索将花费数秒甚至达到分钟级别。实际情况更复杂，查询的操作速度将会受到以下两个因素的影响：

①高并发的更新(插入、修改、删除)操作。大中型网站的并发操作一般能达到几十乃至几百并发，此时单条数据查询的延时将轻而易举地达到分钟级别。

②多表关联后的复杂查询，以及频繁的group by或者order by操作，此时，性能下降较为明显。

## 1.2 CAP定理

分摊读写压力的有效方式是将单个关系型数据库扩展为分布式数据库。但是，随之而来的问题则是很难保证原子性。没有了原子性，事务也无从谈起，关系型数据库也就没有了存在的意义。

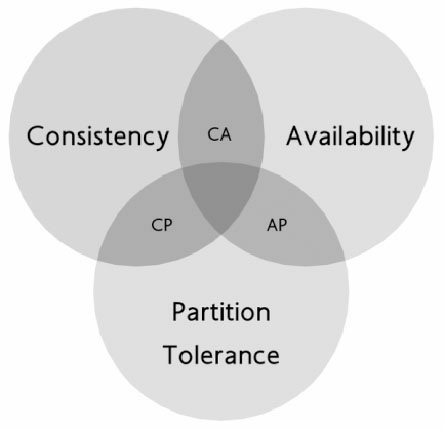
为了保证原子性，则需要增加很多额外的必要操作，此时一次写操作的性能却面临大幅下降了。

20世纪90年代初期Berkerly大学有位Eric Brewer教授提出了一个CAP理论。全称是**Consistency Availability and Partition tolerance**。

* Consistency（强一致性）：数据更新操作的一致性，所有数据变动都是同步的。
* Availability（高可用性）：良好的响应性能。
* Partition tolerance（高分区容错性）：可靠性。

Brewer教授给出的定理是：任何分布式系统只可同时满足二点，没法三者兼顾。

Brewer教授给出的忠告是：架构师不要将精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。所以专家们始终没有办法构建出一个既有完美原子性又兼具高性能的分布式数据库。



## 1.3. NoSQL

有些数据库在实现性能的同时会牺牲一部分一致性，即数据在更新时，不会立刻同步，而是经过了一段时间才达到一致性。这个特性也称之为最终一致性！例如你发了一条朋友圈，你的一部分朋友立马看到了这条信息，而另一部分朋友可能要等到1分钟之后才能刷出这条消息。虽然有延时，但是对于这样一个社交的场景，这个延时是可以容忍的。而如果使用传统关系型数据库，可能这些即时通信软件就早已崩溃！

**NoSQL**数据库最初指不使用SQL标准的数据库，现在泛指非关系型数据库。NoSQL一词最早出现于1998年，是Carlo Strozzi开发的一个轻量、开源、不提供SQL功能的数据库。

现在NoSQL被普遍理解理解为“Not Only SQL”，意为不仅仅是SQL。NoSQL和传统的关系型数据库在很多场景下是相辅相成的，谁也不能完全替代谁。

# 第 2 章 HBase简介



## 2.1 Hbase的来源

2006年Google技术人员Fay Chang发布了一篇文章Bigtable: ADistributed Storage System for Structured Data。该文章向世人介绍了一种分布式的数据库，这种数据库可以在局部几台服务器崩溃的情况下继续提供高性能的服务。

2007年Powerset公司的工作人员基于此文研发了BigTable的Java开源版本，即HBase。刚开始它只是Hadoop的一部分。

2008年HBase成为了Apache的顶级项目。HBase几乎实现了BigTable的所有特性。它被称为一个开源的非关系型分布式数据库。

2010年HBase的开发速度打破了一直以来跟Hadoop版本一致的惯例，因为HBase的版本发布速度已经超越了Hadoop。它的版本号一下从0.20.x跳跃到了0.89.x。其Logo也进行了更换！



## 2.2 HBase定义

HBase是一种分布式、可扩展、支持海量数据存储的NoSQL数据库。

Hbase面向列存储，构建于Hadoop之上，类似于Google的BigTable，提供对10亿级别表数据的快速随机实时读写！

## 2.3 Hbase的特点

### 2.3.1 HBase的特点

1）海量存储

HBase适合存储PB级别的海量数据，在PB级别的数据以及采用廉价PC存储的情况下，能在几十到百毫秒内返回数据。这与HBase的极易扩展性息息相关。正式因为HBase良好的扩展性，才为海量数据的存储提供了便利。

2）列式存储

这里的列式存储其实说的是列族存储，HBase是根据列族来存储数据的。列族下面可以有非常多的列，列族在创建表的时候就必须指定。

3）极易扩展

HBase的扩展性主要体现在两个方面，一个是基于上层处理能力（RegionServer）的扩展，一个是基于存储的扩展（HDFS）。

通过横向添加RegionSever的机器，进行水平扩展，提升HBase上层的处理能力，提升Hbsae服务更多Region的能力。

4）高并发

由于目前大部分使用HBase的架构，都是采用的廉价PC，因此单个IO的延迟其实并不小，一般在几十到上百ms之间。这里说的高并发，主要是在并发的情况下，HBase的单个IO延迟下降并不多。能获得高并发、低延迟的服务。

5）稀疏

稀疏主要是针对HBase列的灵活性，在列族中，你可以指定任意多的列，在列数据为空的情况下，是不会占用存储空间的。

### 2.3.2 Hbase的优点

①HDFS有高容错，高扩展的特点，而Hbase基于HDFS实现数据的存储，因此Hbase拥有与生俱来的超强的扩展性和吞吐量。

②HBase采用的是Key/Value的存储方式，这意味着，即便面临海量数据的增长，也几乎不会导致查询性能下降。

③HBase是一个列式数据库，相对于于传统的行式数据库而言。当你的单张表字段很多的时候，可以将相同的列(以regin为单位)存在到不同的服务实例上，分散负载压力。

### 2.3.3 Hbase的缺点

①架构设计复杂，且使用HDFS作为分布式存储，因此只是存储少量数据，它也不会很快。在大数据量时，它慢的不会很明显！

②Hbase不支持表的关联操作，因此数据分析是HBase的弱项。常见的 group by或order by只能通过编写MapReduce来实现！

③Hbase部分支持了ACID

### 2.3.4 Hbase的总结

适合场景：单表超千万，上亿，且高并发！

不适合场景：主要需求是数据分析，比如做报表。数据量规模不大，对实时性要求高！

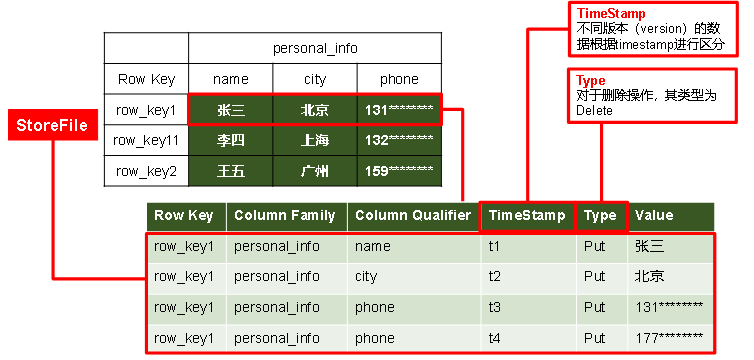
## 2.4 HBase数据模型

逻辑上，HBase的数据模型同关系型数据库很类似，数据存储在一张表中，有行有列。但从HBase的底层物理存储结构（K-V）来看，HBase更像是一个multi-dimensional map。

### 2.4.1 HBase逻辑结构



### 2.4.2 HBase物理存储结构



### 2.4.3 数据模型

**1）Name Space**

命名空间，类似于关系型数据库的database概念，每个命名空间下有多个表。HBase两个自带的命名空间，分别是hbase和default，hbase中存放的是HBase内置的表，default表是用户默认使用的命名空间。

一个表可以自由选择是否有命名空间，如果创建表的时候加上了命名空间后，这个表名字以<Namespace>:<Table>作为区分！

**2）Table**

类似于关系型数据库的表概念。不同的是，HBase定义表时只需要声明列族即可，数据属性，比如超时时间（TTL），压缩算法（COMPRESSION）等，都在列族的定义中定义，不需要声明具体的列。

这意味着，往HBase写入数据时，字段可以动态、按需指定。因此，和关系型数据库相比，HBase能够轻松应对字段变更的场景。

**3）Row**

HBase表中的每行数据都由一个RowKey和多个Column（列）组成。一个行包含了多个列，这些列通过列族来分类,行中的数据所属列族只能从该表所定义的列族中选取,不能定义这个表中不存在的列族，否则报错NoSuchColumnFamilyException。

**4) RowKey**

Rowkey由用户指定的一串不重复的字符串定义，是一行的唯一标识！数据是按照RowKey的字典顺序存储的，并且查询数据时只能根据RowKey进行检索，所以RowKey的设计十分重要。

如果使用了之前已经定义的RowKey，那么会将之前的数据更新掉！

**5）Column Family**

列族是多个列的集合。一个列族可以动态地灵活定义多个列。表的相关属性大部分都定义在列族上，同一个表里的不同列族可以有完全不同的属性配置，但是同一个列族内的所有列都会有相同的属性。

列族存在的意义是HBase会把相同列族的列尽量放在同一台机器上，所以说，如果想让某几个列被放到一起，你就给他们定义相同的列族。

官方建议一张表的列族定义的越少越好，列族太多会极大程度地降低数据库性能，且目前版本Hbase的架构，容易出BUG。

**6) Column Qualifier**

Hbase中的列是可以随意定义的，一个行中的列不限名字、不限数量，只限定列族。因此列必须依赖于列族存在！列的名称前必须带着其所属的列族！例如info：name，info：age。

因为HBase中的列全部都是灵活的，可以随便定义的，因此创建表的时候并不需要指定列！列只有在你插入第一条数据的时候才会生成。其他行有没有当前行相同的列是不确定，只有在扫描数据的时候才能得知！

**7）TimeStamp**

用于标识数据的不同版本（version）。时间戳默认由系统指定，也可以由用户显式指定。

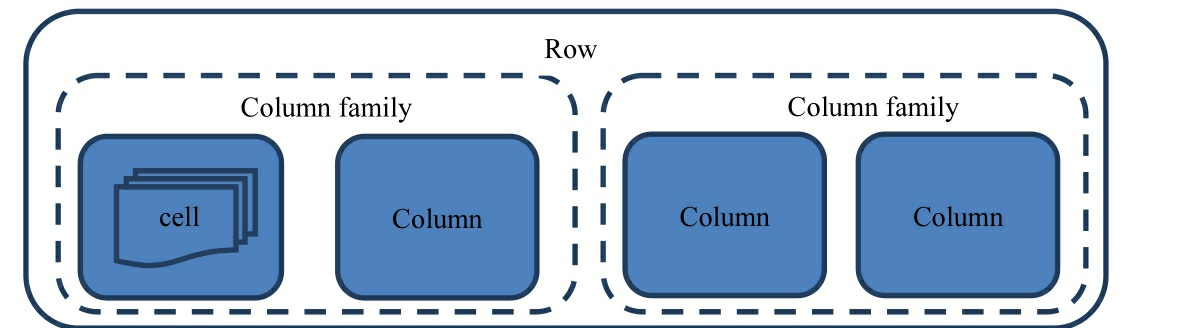
在读取单元格的数据时，版本号可以省略，如果不指定，Hbase默认会获取最后一个版本的数据返回！

**8）Cell**

一个列中可以存储多个版本的数据。而每个版本就称为一个单元格（Cell）。

Cell由{rowkey, column Family：column Qualifier, time Stamp}确定。

Cell中的数据是没有类型的，全部是字节码形式存贮。



**9）Region**

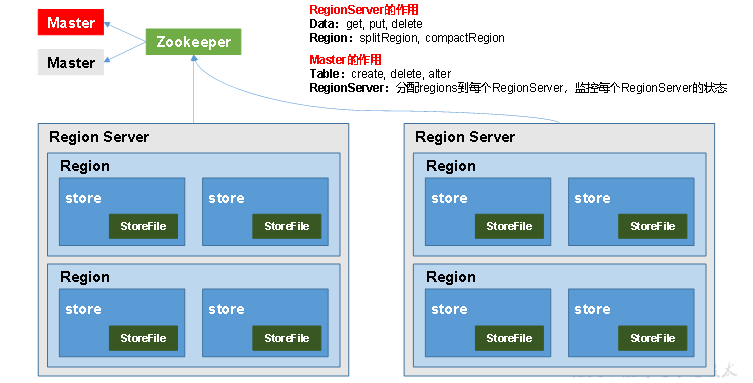
Region由一个表的若干行组成！在Region中行的排序按照行键（rowkey）字典排序。

Region不能跨RegionSever，且当数据量大的时候，HBase会拆分Region。

Region由RegionServer进程管理。HBase在进行负载均衡的时候，一个Region有可能会从当前RegionServer移动到其他RegionServer上。

Region是基于HDFS的，它的所有数据存取操作都是调用了HDFS的客户端接口来实现的。

## 2.5 HBase基本架构



**架构角色：**

**1）Region Server**

RegionServer是一个服务，负责多个Region的管理。其实现类为HRegionServer，主要作用如下:

对于数据的操作：get, put, delete；

对于Region的操作：splitRegion、compactRegion。

客户端从ZooKeeper获取RegionServer的地址，从而调用相应的服务，获取数据。

**2）Master**

Master是所有Region Server的管理者，其实现类为HMaster，主要作用如下：

对于表的操作：create, delete, alter，这些操作可能需要跨多个ReginServer，因此需要Master来进行协调！

对于RegionServer的操作：分配regions到每个RegionServer，监控每个RegionServer的状态，负载均衡和故障转移。

即使Master进程宕机，集群依然可以执行数据的读写，只是不能进行表的创建和修改等操作！当然Master也不能宕机太久，有很多必要的操作，比如创建表、修改列族配置，以及更重要的分割和合并都需要它的操作。

**3）Zookeeper**

RegionServer非常依赖ZooKeeper服务，ZooKeeper管理了HBase所有RegionServer的信息，包括具体的数据段存放在哪个RegionServer上。

客户端每次与HBase连接，其实都是先与ZooKeeper通信，查询出哪个RegionServer需要连接，然后再连接RegionServer。Zookeeper中记录了读取数据所需要的元数据表

hbase:meata,因此关闭Zookeeper后，客户端是无法实现读操作的！

HBase通过Zookeeper来做master的高可用、RegionServer的监控、元数据的入口以及集群配置的维护等工作。

**4）HDFS**

HDFS为Hbase提供最终的底层数据存储服务，同时为HBase提供高可用的支持。

# 第 3 章 Hbase安装

## 3.1 Hbase安装前

### 3.1.1 启动Zookeeper

首先保证Zookeeper集群的正常部署，并启动之：

[woaini@hadoop102 zookeeper-3.4.10]$ bin/zkServer.sh start

[woaini@hadoop103 zookeeper-3.4.10]$ bin/zkServer.sh start

[woaini@hadoop104 zookeeper-3.4.10]$ bin/zkServer.sh start

### 3.1.2 启动Hadoop

Hadoop集群的正常部署并启动：

[woaini@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh

[woaini@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh

### 3.1.3 HBase的解压

解压HBase到指定目录：

[woaini@hadoop102 software]$ tar -zxvf HBase-1.3.1-bin.tar.gz -C /opt/module

## 3.2 HBase的安装

### 3.2.1 HBase的配置

修改HBase对应的配置文件。

1）HBase-env.sh修改内容：

export JAVA\_HOME=/opt/module/jdk1.6.0\_144

export HBASE\_MANAGES\_ZK=false

2）HBase-site.xml修改内容：

<configuration>

<property>

<name>hbase.rootdir</name>

<value>hdfs://hadoop102:9000/HBase</value>

</property>

<property>

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

<value>true</value>

</property>

<!-- 0.98后的新变动，之前版本没有.port,默认端口为60000 -->

<property>

<name>hbase.master.port</name>

<value>16000</value>

</property>

<property>

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

<value>hadoop102:2181,hadoop103:2181,hadoop104:2181</value>

</property>

<property>

<name>hbase.zookeeper.property.dataDir</name>

<value>/opt/module/zookeeper-3.4.10/zkData</value>

</property>

</configuration>

3）regionservers：

|  |
| --- |
| hadoop102  hadoop103  hadoop104 |

4）软连接hadoop配置文件到HBase：

[woaini@hadoop102 module]$ ln -s /opt/module/hadoop-2.7.2/etc/hadoop/core-site.xml

/opt/module/HBase/conf/core-site.xml

[woaini@hadoop102 module]$ ln -s /opt/module/hadoop-2.7.2/etc/hadoop/hdfs-site.xml

/opt/module/HBase/conf/hdfs-site.xml

### 3.2.2 分发HBase到其他节点

[woaini@hadoop102 module]$ xsync HBase/

## 3.3 HBase服务的启动

**1．启动方式1**

[woaini@hadoop102 HBase]$ bin/HBase-daemon.sh start master

[woaini@hadoop102 HBase]$ bin/HBase-daemon.sh start regionserver

提示：如果集群之间的节点时间不同步，会导致regionserver无法启动，抛出ClockOutOfSyncException异常。

修复提示：

a、同步时间服务

请参看帮助文档：《hi大数据技术之Hadoop入门》

b、属性：hbase.master.maxclockskew设置更大的值

|  |
| --- |
| <property>  <name>hbase.master.maxclockskew</name>  <value>180000</value>  <description>Time difference of regionserver from master</description>  </property> |

**2．启动方式2**

[woaini@hadoop102 HBase]$ bin/start-HBase.sh

对应的停止服务：

[woaini@hadoop102 HBase]$ bin/stop-HBase.sh

## 3.4 查看HBase页面

启动成功后，可以通过“host:port”的方式来访问HBase管理页面，例如：[http://hadoop102:16010](http://linux01:16010)

# 第 4 章 HBase Shell操作

使用hbase shell可以进入一个shell命令行界面！

[woaini@hadoop102 HBase]$ bin/HBase shell

## 4.1 其他操作

**1.查看集群状态**

使用status可以查看集群状态，默认为summary，可以选择‘simple’和‘detailed’来查看详情。

hbase(main):011:0> status

1 active master, 0 backup masters, 3 servers, 0 dead, 0.6667 average load

**2.查看版本**

hbase(main):002:0> version

1.3.1, r930b9a55528fe45d8edce7af42fef2d35e77677a, Thu Apr 6 19:36:54 PDT 2017

**3.查看操作用户及组信息**

hbase(main):003:0> whoami

woaini (auth:SIMPLE)

groups: woaini

**4.查看表操作信息**

hbase(main):004:0> table\_help

Help for table-reference commands.

**5.查看帮助信息**

hbase(main):005:0> help

HBase Shell, version 1.3.1, r930b9a55528fe45d8edce7af42fef2d35e77677a, Thu Apr 6 19:36:54 PDT 2017

Type 'help "COMMAND"', (e.g. 'help "get"' -- the quotes are necessary) for help on a specific command.

Commands are grouped. Type 'help "COMMAND\_GROUP"', (e.g. 'help "general"') for help on a command group.

**6.查看具体命令的帮助**

hbase(main):006:0> help 'get'

Get row or cell contents; pass table name, row, and optionally

a dictionary of column(s), timestamp, timerange and versions. Examples:

hbase> get 'ns1:t1', 'r1'

注意引号是必须的！

## 4.2 表的操作

### 4.2.1. list

hbase(main):008:0> list

TABLE

0 row(s) in 0.0410 seconds

list后可以使用\*等通配符来进行表的过滤！

### 4.2.2 create

创建表时，需要指定表名和列族名，而且至少需要指定一个列族，没有列族的表是没有任何意义的。

创建表时，还可以指定表的属性，表的属性需要指定在列族上！

格式：

create '表名', { NAME => '列族名1', 属性名 => 属性值}, {NAME => '列族名2', 属性名 => 属性值}, …

如果你只需要创建列族，而不需要定义列族属性，那么可以采用以下快捷写法：

create'表名','列族名1' ,'列族名2', …

HBase(main):002:0> create 'student','info'

### 4.2.3 desc

hbase(main):003:0> describe 'person'

hbase(main):004:0> desc 'person'

### 4.2.4 disable

停用表后，可以防止在对表做一些维护时，客户端依然可以持续写入数据到表。一般在删除表前，必须停用表。

在对表中的列族进行修改时，也需要停用表。

hbase(main):005:0> disable 'person'

0 row(s) in 2.4250 seconds

disable\_all ‘正则表达式’ 可以使用正则来匹配表名。

is\_disabled 可以用来判断表是否被停用。

hbase(main):006:0> is\_disabled 'person'

true

0 row(s) in 0.0160 seconds

### 4.2.5 enable

**和停用表类似。enable ‘表名’用来启用表，is\_enabled ‘表名’用来判断一个表是否被启用。**

**enable\_all ‘正则表达式’可以通过正则来过滤表，启用复合条件的表。**

### 4.2.6. exists

hbase(main):008:0> exists 'person'

Table person does exist

0 row(s) in 0.0210 seconds

### 4.2.7. count

hbase(main):012:0> count 'person'

1 row(s) in 0.0240 seconds

=> 1

### 4.2.8. drop

**删除表前，需要先disable表，否则会报错。ERROR: Table xxx is enabled. Disable it first.**

hbase(main):011:0> drop 'person'

### 4.2.9 truncate

hbase(main):013:0> truncate 'person'

Truncating 'person' table (it may take a while):

- Disabling table...

- Truncating table...

0 row(s) in 4.0010 seconds

### 4.2.10 get\_split

hbase(main):015:0> get\_splits 'person'

Total number of splits = 1

=> []

获取表所对应的Region个数。每个表在一开始只有一个region，之后记录增多后，region会被自动拆分。

### 4.2.11 alter

alter命令可以修改表的属性，通常是修改某个列族的属性。

alter ‘表名’， ‘delete’ => ‘列族名’

hbase(main):050:0> alter 'myns:t1',{NAME => 'info',VERSIONS => '5'}

hbase> alter 'ns1:t1', 'delete' => 'f1'

## 4.3 数据操作

### 4.3.1 scan

**scan命令可以按照rowkey的字典顺序来遍历指定的表的数据。**

**scan ‘表名’：默认当前表的所有列族。**

**scan ‘表名’,{COLUMNS=> [‘列族:列名’],…} ： 遍历表的指定列**

**scan '表名', { STARTROW => '起始行键', ENDROW => '结束行键' }：指定rowkey范围。如果不指定，则会从表的开头一直显示到表的结尾。区间为左闭右开。**

**scan '表名', { LIMIT => 行数量}： 指定返回的行的数量**

**scan '表名', {VERSIONS => 版本数}：返回cell的多个版本**

**scan '表名', { TIMERANGE => [最小时间戳, 最大时间戳]}：指定时间戳范围**

**注意：此区间是一个左闭右开的区间，因此返回的结果包含最小时间戳的记录，但是不包含最大时间戳记录**

**scan '表名', { RAW => true, VERSIONS => 版本数}**

**显示原始单元格记录，在Hbase中，被删掉的记录在HBase被删除掉的记录并不会立即从磁盘上清除，而是先被打上墓碑标记，然后等待下次major compaction的时候再被删除掉。注意RAW参数必须和VERSIONS一起使用，但是不能和COLUMNS参数一起使用。**

**scan '表名', { FILTER => "过滤器"} and|or { FILTER => "过滤器"}: 使用过滤器扫描**

HBase(main):008:0> scan 'student'

HBase(main):009:0> scan 'student',{STARTROW => '1001', STOPROW => '1001'}

HBase(main):010:0> scan 'student',{STARTROW => '1001'}

### 4.3.2 put

put可以新增记录还可以为记录设置属性。

put '表名', '行键', '列名', '值'

put '表名', '行键', '列名', '值',时间戳

put '表名', '行键', '列名', '值', { '属性名' => '属性值'}

put '表名', '行键', '列名', '值',时间戳, { '属性名' =>'属性值'}

HBase(main):012:0> put 'student','1001','info:name','Nick'

HBase(main):003:0> put 'student','1001','info:sex','male'

HBase(main):004:0> put 'student','1001','info:age','18'

HBase(main):005:0> put 'student','1002','info:name','Janna'

HBase(main):006:0> put 'student','1002','info:sex','female'

HBase(main):007:0> put 'student','1002','info:age','20'

### 4.3.3 get

get支持scan所支持的大部分属性，如COLUMNS，TIMERANGE，VERSIONS，FILTER

HBase(main):014:0> get 'student','1001'

HBase(main):015:0> get 'student','1001','info:name'

### 4.3.4 delete

删除某rowkey的全部数据：

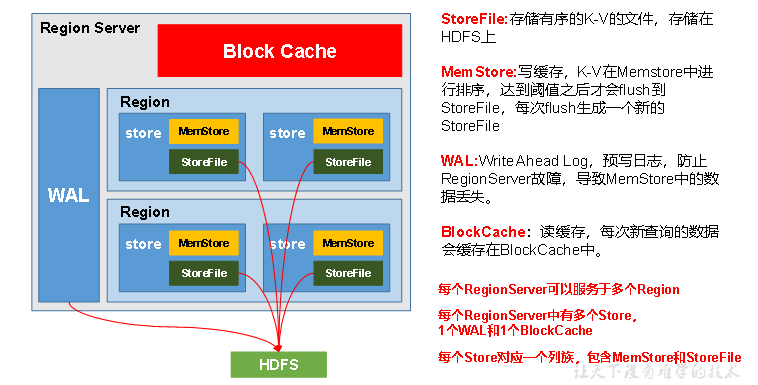
HBase(main):016:0> deleteall 'student','1001'

删除某rowkey的某一列数据：

HBase(main):017:0> delete 'student','1002','info:sex'

# 第 5 章 Hbase进阶

## 5.1 RegionServer 架构



**1）StoreFile**

保存实际数据的物理文件，StoreFile以Hfile的形式存储在HDFS上。每个Store会有一个或多个StoreFile（HFile），数据在每个StoreFile中都是有序的。

**2）MemStore**

写缓存，由于HFile中的数据要求是有序的，所以数据是先存储在MemStore中，排好序后，等到达刷写时机才会刷写到HFile，每次刷写都会形成一个新的HFile。

**3）WAL**

由于数据要经MemStore排序后才能刷写到HFile，但把数据保存在内存中会有很高的概率导致数据丢失，为了解决这个问题，数据会先写在一个叫做Write-Ahead logfile的文件中，然后再写入MemStore中。所以在系统出现故障的时候，数据可以通过这个日志文件重建。

每间隔hbase.regionserver.optionallogflushinterval(默认1s)， HBase会把操作从内存写入WAL。

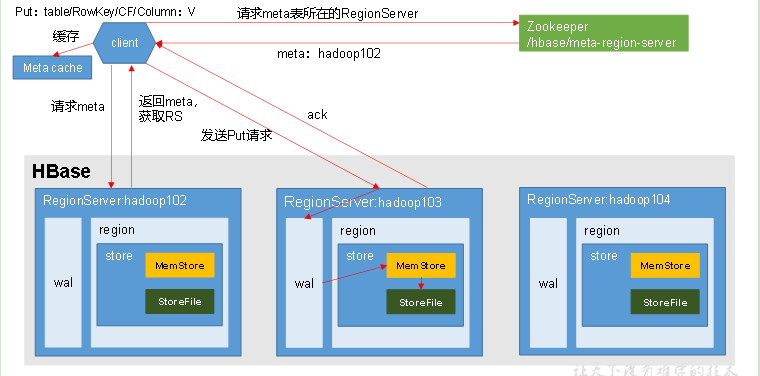
一个RegionServer上的所有Region共享一个WAL实例。

WAL的检查间隔由hbase.regionserver.logroll.period定义，默认值为1小时。检查的内容是把当前WAL中的操作跟实际持久化到HDFS上的操作比较，看哪些操作已经被持久化了，被持久化的操作就会被移动到.oldlogs文件夹内（这个文件夹也是在HDFS上的）。一个WAL实例包含有多个WAL文件。WAL文件的最大数量通过hbase.regionserver.maxlogs（默认是32）参数来定义。

**4）BlockCache**

读缓存，每次查询出的数据会缓存在BlockCache中，方便下次查询。

## 5.2 写流程



写流程：

1）Client先访问zookeeper，获取hbase:meta表位于哪个Region Server。

2）访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问。

3）与目标Region Server进行通讯；

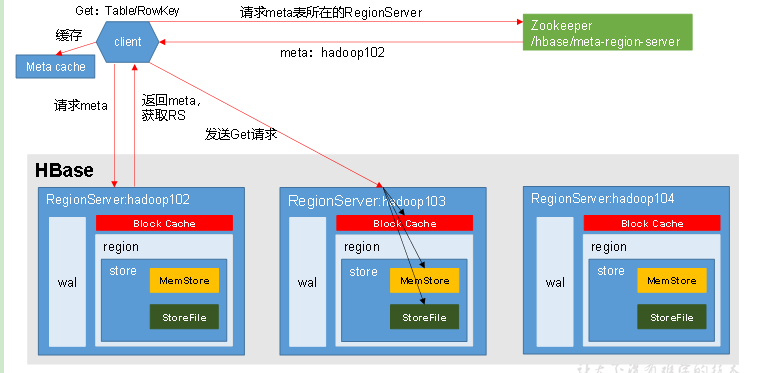
4）将数据顺序写入（追加）到WAL；

5）将数据写入对应的MemStore，数据会在MemStore进行排序；

6）向客户端发送ack；

7）等达到MemStore的刷写时机后，将数据刷写到HFile。

## 5.3 读流程



**读流程**

1）Client先访问zookeeper，获取hbase:meta表位于哪个Region Server。

2）访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问。

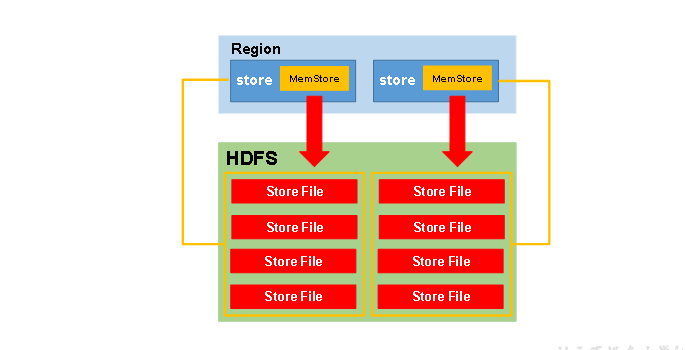
3）与目标Region Server进行通讯；

4）分别在Block Cache（读缓存），MemStore和Store File（HFile）中查询目标数据，并将查到的所有数据进行合并。此处所有数据是指同一条数据的不同版本（time stamp）或者不同的类型（Put/Delete）。

5）将查询到的数据块（Block，HFile数据存储单元，默认大小为64KB）缓存到Block Cache。

6）将合并后的最终结果返回给客户端。

## 5.4 MemStore Flush



**MemStore存在的意义是在写入HDFS前，将其中的数据整理有序。**

**MemStore刷写时机：**

1.当某个memstore的大小达到了**hbase.hregion.memstore.flush.size（默认值128M）**，其所在region的所有memstore都会刷写。

当memstore的大小达到了

**hbase.hregion.memstore.flush.size（默认值128M）**

**\*** **hbase.hregion.memstore.block.multiplier（默认值4）**

时，会阻止继续往该memstore写数据。

2.当region server中memstore的总大小达到

**java\_heapsize**

**\*hbase.regionserver.global.memstore.size（默认值0.4）**

**\*hbase.regionserver.global.memstore.size.lower.limit（默认值0.95）**，

region会按照其所有memstore的大小顺序（由大到小）依次进行刷写。直到region server中所有memstore的总大小减小到上述值以下。

当region server中memstore的总大小达到

**java\_heapsize**

**\*hbase.regionserver.global.memstore.size（默认值0.4）**

时，会阻止继续往所有的memstore写数据。

3. 到达自动刷写的时间，也会触发memstore flush。自动刷新的时间间隔由该属性进行配置**hbase.regionserver.optionalcacheflushinterval（默认1小时）**。

4.当WAL文件的数量超过**hbase.regionserver.max.logs**，region会按照时间顺序依次进行刷写，直到WAL文件数量减小到**hbase.regionserver.max.log**以下（该属性名已经废弃，现无需手动设置，最大值为32）。

## 5.5 StoreFile Compaction

由于Hbase依赖HDFS存储，HDFS只支持追加写。所以，当新增一个单元格的时候，HBase在HDFS上新增一条数据。当修改一个单元格的时候，HBase在HDFS又新增一条数据，只是版本号比之前那个大（或者自定义）。当删除一个单元格的时候，HBase还是新增一条数据！只是这条数据没有value，类型为DELETE，也称为墓碑标记（Tombstone）

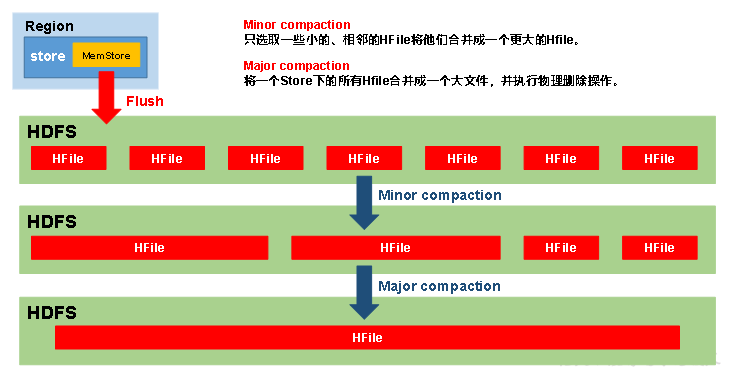
HBase每间隔一段时间都会进行一次合并（Compaction），合并的对象为HFile文件。合并分为两种

minor compaction和major compaction。

在HBase进行major compaction的时候，它会把多个HFile合并成1个HFile，在这个过程中，一旦检测到有被打上墓碑标记的记录，在合并的过程中就忽略这条记录。这样在新产生的HFile中，就没有这条记录了，自然也就相当于被真正地删除了

由于memstore每次刷写都会生成一个新的HFile，且同一个字段的不同版本（timestamp）和不同类型（Put/Delete）有可能会分布在不同的HFile中，因此查询时需要遍历所有的HFile。为了减少HFile的个数，以及清理掉过期和删除的数据，会进行StoreFile Compaction。

Compaction分为两种，分别是Minor Compaction和Major Compaction。Minor Compaction会将临近的若干个较小的HFile合并成一个较大的HFile，但**不会**清理过期和删除的数据。Major Compaction会将一个Store下的所有的HFile合并成一个大HFile，并且**会**清理掉过期和删除的数据。



## 5.6 Region Split

默认情况下，每个Table起初只有一个Region，随着数据的不断写入，Region会自动进行拆分。刚拆分时，两个子Region都位于当前的Region Server，但处于负载均衡的考虑，HMaster有可能会将某个Region转移给其他的Region Server。

Region Split时机：

1.当1个region中的某个Store下所有StoreFile的总大小超过hbase.hregion.max.filesize，该Region就会进行拆分（0.94版本之前）。

2. 0.94版本之后的切分策略

默认使用IncreasingToUpperBoundRegionSplitPolicy策略切分region, getSizeToCheck()是检查region的大小以判断是否满足切割切割条件。

|  |
| --- |
| **protected** **long** getSizeToCheck(**final** **int** tableRegionsCount) {  // safety check for 100 to avoid numerical overflow in extreme cases  **return** tableRegionsCount == 0 || tableRegionsCount > 100  ? getDesiredMaxFileSize()  : Math.*min*(getDesiredMaxFileSize(),  initialSize \* tableRegionsCount \* tableRegionsCount \* tableRegionsCount);  } |

tableRegionsCount：为当前Region Server中属于该Table的region的个数。

getDesiredMaxFileSize() 这个值是hbase.hregion.max.filesize参数值，默认为10GB。

initialSize的初始化比较复杂，由多个参数决定。

|  |
| --- |
| @Override  **protected** **void** configureForRegion(HRegion region) {  **super**.configureForRegion(region);  Configuration conf = getConf();  //默认hbase.increasing.policy.initial.size 没有在配置文件中指定  initialSize = conf.getLong("hbase.increasing.policy.initial.size", -1);  **if** (initialSize > 0) {  **return**;  }  // 获取用户表中自定义的memstoreFlushSize大小，默认也为128M  HTableDescriptor desc = region.getTableDesc();  **if** (desc != **null**) {  initialSize = 2 \* desc.getMemStoreFlushSize();  }  // 判断用户指定的memstoreFlushSize是否合法，如果不合法，则为hbase.hregion.memstore.flush.size，默认为128.  **if** (initialSize <= 0) {  initialSize = 2 \* conf.getLong(HConstants.***HREGION\_MEMSTORE\_FLUSH\_SIZE***,  HTableDescriptor.***DEFAULT\_MEMSTORE\_FLUSH\_SIZE***);  }  } |

具体的切分策略为tableRegionsCount在0和100之间，则为

initialSize（默认为2\*128） \* tableRegionsCount^3,例如：

第一次split：1^3 \* 256 = 256MB

第二次split：2^3 \* 256 = 2048MB

第三次split：3^3 \* 256 = 6912MB

第四次split：4^3 \* 256 = 16384MB > 10GB，因此取较小的值10GB

后面每次split的size都是10GB了。

tableRegionsCount超过100个，则超过10GB才会切分region。



hbase.regionserver.region.split.policy：

# 第 6 章 HBase API

## 6.1 添加依赖

新建项目后在pom.xml中添加依赖：

|  |
| --- |
| <dependency>  <groupId>org.apache.hbase</groupId>  <artifactId>hbase-server</artifactId>  <version>1.3.1</version>  </dependency>  <dependency>  <groupId>org.apache.hbase</groupId>  <artifactId>hbase-client</artifactId>  <version>1.3.1</version>  </dependency>  <dependency>  <groupId>jdk.tools</groupId>  <artifactId>jdk.tools</artifactId>  <version>1.8</version>  <scope>system</scope>  <systemPath>${JAVA\_HOME}/lib/tools.jar</systemPath>  </dependency> |

在对HBase执行增删改查时，只需要引入hbase-client模块即可，运行MR操作hbase时，需要引入hbase-server。

拷贝hdfs-site.xml文件到客户端的类路径下！

## 6.2 Hbase核心API

### 6.2.1 获取Configuration对象

Connection代表对集群的连接对象，封装了与实际服务器的低级别单独连接以及与zookeeper的连接。

Connection可以通过ConnectionFactory类实例化。

Connection的生命周期由调用者管理，使用完毕后需要执行close()以释放资源。

Connection是线程安全的，多个Table和Admin可以共用同一个Connection对象。因此一个客户端只需要实例化一个连接即可。

反之，Table和Admin不是线程安全的！因此不建议并缓存或池化这两种对象。

|  |
| --- |
| public static Configuration conf;  static{  //使用HBaseConfiguration的单例方法实例化  conf = HBaseConfiguration.create();  conf.set("HBase.zookeeper.quorum", "192.166.9.102");  conf.set("HBase.zookeeper.property.clientPort", "2181");  } |

### 6.2.2 判断表是否存在

Admin为HBase的管理类，可以通过Connection.getAdmin（）获取实例，且在使用完成后调用close（）关闭。

Admin可用于创建，删除，列出，启用和禁用以及以其他方式修改表，以及执行其他管理操作。

public static boolean isTableExist(String tableName) throws MasterNotRunningException,

ZooKeeperConnectionException, IOException{

//在HBase中管理、访问表需要先创建HBaseAdmin对象

//Connection connection = ConnectionFactory.createConnection(conf);

//HBaseAdmin admin = (HBaseAdmin) connection.getAdmin();

HBaseAdmin admin = new HBaseAdmin(conf);

return admin.tableExists(tableName);

}

|  |
| --- |
|  |

### 6.2.3 创建表

|  |
| --- |
| public static void createTable(String tableName, String... columnFamily) throws  MasterNotRunningException, ZooKeeperConnectionException, IOException{  HBaseAdmin admin = new HBaseAdmin(conf);  //判断表是否存在  if(isTableExist(tableName)){  System.out.println("表" + tableName + "已存在");  //System.exit(0);  }else{  //创建表属性对象,表名需要转字节  HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(tableName));  //创建多个列族  for(String cf : columnFamily){  descriptor.addFamily(new HColumnDescriptor(cf));  }  //根据对表的配置，创建表  admin.createTable(descriptor);  System.out.println("表" + tableName + "创建成功！");  }  } |

### 6.2.4 删除表

|  |
| --- |
| public static void dropTable(String tableName) throws MasterNotRunningException,  ZooKeeperConnectionException, IOException{  HBaseAdmin admin = new HBaseAdmin(conf);  if(isTableExist(tableName)){  admin.disableTable(tableName);  admin.deleteTable(tableName);  System.out.println("表" + tableName + "删除成功！");  }else{  System.out.println("表" + tableName + "不存在！");  }  } |

### 6.2.5 向表中插入数据

|  |
| --- |
| public static void addRowData(String tableName, String rowKey, String columnFamily, String  column, String value) throws IOException{  //创建HTable对象  HTable hTable = new HTable(conf, tableName);  //向表中插入数据  Put put = new Put(Bytes.toBytes(rowKey));  //向Put对象中组装数据  put.add(Bytes.toBytes(columnFamily), Bytes.toBytes(column), Bytes.toBytes(value));  hTable.put(put);  hTable.close();  System.out.println("插入数据成功");  } |

### 6.2.6 删除多行数据

|  |
| --- |
| public static void deleteMultiRow(String tableName, String... rows) throws IOException{  HTable hTable = new HTable(conf, tableName);  List<Delete> deleteList = new ArrayList<Delete>();  for(String row : rows){  Delete delete = new Delete(Bytes.toBytes(row));  deleteList.add(delete);  }  hTable.delete(deleteList);  hTable.close();  } |

### 6.2.7 获取所有数据

|  |
| --- |
| public static void getAllRows(String tableName) throws IOException{  HTable hTable = new HTable(conf, tableName);  //得到用于扫描region的对象  Scan scan = new Scan();  //使用HTable得到resultcanner实现类的对象  ResultScanner resultScanner = hTable.getScanner(scan);  for(Result result : resultScanner){  Cell[] cells = result.rawCells();  for(Cell cell : cells){  //得到rowkey  System.out.println("行键:" + Bytes.toString(CellUtil.cloneRow(cell)));  //得到列族  System.out.println("列族" + Bytes.toString(CellUtil.cloneFamily(cell)));  System.out.println("列:" + Bytes.toString(CellUtil.cloneQualifier(cell)));  System.out.println("值:" + Bytes.toString(CellUtil.cloneValue(cell)));  }  }  } |

### 6.2.8 获取某一行数据

|  |
| --- |
| public static void getRow(String tableName, String rowKey) throws IOException{  HTable table = new HTable(conf, tableName);  Get get = new Get(Bytes.toBytes(rowKey));  //get.setMaxVersions();显示所有版本  //get.setTimeStamp();显示指定时间戳的版本  Result result = table.get(get);  for(Cell cell : result.rawCells()){  System.out.println("行键:" + Bytes.toString(result.getRow()));  System.out.println("列族" + Bytes.toString(CellUtil.cloneFamily(cell)));  System.out.println("列:" + Bytes.toString(CellUtil.cloneQualifier(cell)));  System.out.println("值:" + Bytes.toString(CellUtil.cloneValue(cell)));  System.out.println("时间戳:" + cell.getTimestamp());  }  } |

### 6.2.9 获取某一行指定“列族:列”的数据

|  |
| --- |
| public static void getRowQualifier(String tableName, String rowKey, String family, String  qualifier) throws IOException{  HTable table = new HTable(conf, tableName);  Get get = new Get(Bytes.toBytes(rowKey));  get.addColumn(Bytes.toBytes(family), Bytes.toBytes(qualifier));  Result result = table.get(get);  for(Cell cell : result.rawCells()){  System.out.println("行键:" + Bytes.toString(result.getRow()));  System.out.println("列族" + Bytes.toString(CellUtil.cloneFamily(cell)));  System.out.println("列:" + Bytes.toString(CellUtil.cloneQualifier(cell)));  System.out.println("值:" + Bytes.toString(CellUtil.cloneValue(cell)));  }  } |

# 第 7 章 HBase-MapReduce

## 7.1 MapReduce

统计的需要：我们知道HBase的数据都是分布式存储在RegionServer上的，所以对于类似传统关系型数据库的group by操作，扫描器是无能为力的，只有当所有结果都返回到客户端的时候，才能进行统计。这样做一是慢，二是会产生很大的网络开销，所以使用MapReduce在服务器端就进行统计是比较好的方案。

性能的需要：说白了就是“快”！如果遇到较复杂的场景，在扫描器上添加多个过滤器后，扫描的性能很低；或者当数据量很大的时候扫描器也会执行得很慢，原因是扫描器和过滤器内部实现的机制很复杂，虽然使用者调用简单，但是服务器端的性能就不敢保证了

通过HBase的相关JavaAPI，我们可以实现伴随HBase操作的MapReduce过程，比如使用MapReduce将数据从本地文件系统导入到HBase的表中，比如我们从HBase中读取一些原始数据后使用MapReduce做数据分析。

## 7.2 官方HBase-MapReduce

1．查看HBase的MapReduce任务的执行

$ bin/HBase mapredcp

2．环境变量的导入

让Hadoop加载Hbase的jar包，最简单的就是把HBase的jar包复制到Hadoop的lib里面，或者把HBase的包地址写到Hadoop的环境变量里面。

（1）执行环境变量的导入（临时生效，在命令行执行下述操作）

$ export HBASE\_HOME=/opt/module/HBase-1.3.1

$ export HADOOP\_HOME=/opt/module/hadoop-2.7.2

$ export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`

（2）永久生效：在/etc/profile配置

export HBASE\_HOME=/opt/module/HBase-1.3.1

export HADOOP\_HOME=/opt/module/hadoop-2.7.2

并在hadoop-env.sh中配置：（注意：在for循环之后配）

export HADOOP\_CLASSPATH=$HADOOP\_CLASSPATH:/opt/module/HBase/lib/\*

3．运行官方的MapReduce任务

-- 案例一：统计Student表中有多少行数据

$ /opt/module/hadoop-2.7.2/bin/yarn jar lib/HBase-server-1.3.1.jar rowcounter student

-- 案例二：使用MapReduce将本地数据导入到HBase

1）在本地创建一个tsv格式的文件：fruit.tsv

|  |
| --- |
| 1001 Apple Red  1002 Pear Yellow  1003 Pineapple Yellow |

2）创建HBase表

HBase(main):001:0> create 'fruit','info'

3）在HDFS中创建input\_fruit文件夹并上传fruit.tsv文件

$ /opt/module/hadoop-2.7.2/bin/hdfs dfs -mkdir /input\_fruit/

$ /opt/module/hadoop-2.7.2/bin/hdfs dfs -put fruit.tsv /input\_fruit/

1. 执行MapReduce到HBase的fruit表中

$ /opt/module/hadoop-2.7.2/bin/yarn jar lib/HBase-server-1.3.1.jar importtsv \

-Dimporttsv.columns=HBASE\_ROW\_KEY,info:name,info:color fruit \

hdfs://hadoop102:9000/input\_fruit

1. 使用scan命令查看导入后的结果

HBase(main):001:0> scan ‘fruit’

## 7.2 自定义HBase-MapReduce1

目标：将fruit表中的一部分数据，通过MR迁入到fruit\_mr表中。

分步实现：

1．构建ReadFruitMapper类，用于读取fruit表中的数据

|  |
| --- |
| package com.woaini;  import java.io.IOException;  import org.apache.hadoop.HBase.Cell;  import org.apache.hadoop.HBase.CellUtil;  import org.apache.hadoop.HBase.client.Put;  import org.apache.hadoop.HBase.client.Result;  import org.apache.hadoop.HBase.io.ImmutableBytesWritable;  import org.apache.hadoop.HBase.mapreduce.TableMapper;  import org.apache.hadoop.HBase.util.Bytes;  public class ReadFruitMapper extends TableMapper<ImmutableBytesWritable, Put> {  @Override  protected void map(ImmutableBytesWritable key, Result value, Context context)  throws IOException, InterruptedException {  //将fruit的name和color提取出来，相当于将每一行数据读取出来放入到Put对象中。  Put put = new Put(key.get());  //遍历添加column行  for(Cell cell: value.rawCells()){  //添加/克隆列族:info  if("info".equals(Bytes.toString(CellUtil.cloneFamily(cell)))){  //添加/克隆列：name  if("name".equals(Bytes.toString(CellUtil.cloneQualifier(cell)))){  //将该列cell加入到put对象中  put.add(cell);  //添加/克隆列:color  }else if("color".equals(Bytes.toString(CellUtil.cloneQualifier(cell)))){  //向该列cell加入到put对象中  put.add(cell);  }  }  }  //将从fruit读取到的每行数据写入到context中作为map的输出  context.write(key, put);  }  } |

2． 构建WriteFruitMRReducer类，用于将读取到的fruit表中的数据写入到fruit\_mr表中

|  |
| --- |
| package com.woaini.HBase\_mr;  import java.io.IOException;  import org.apache.hadoop.HBase.client.Put;  import org.apache.hadoop.HBase.io.ImmutableBytesWritable;  import org.apache.hadoop.HBase.mapreduce.TableReducer;  import org.apache.hadoop.io.NullWritable;  public class WriteFruitMRReducer extends TableReducer<ImmutableBytesWritable, Put, NullWritable> {  @Override  protected void reduce(ImmutableBytesWritable key, Iterable<Put> values, Context context)  throws IOException, InterruptedException {  //读出来的每一行数据写入到fruit\_mr表中  for(Put put: values){  context.write(NullWritable.get(), put);  }  }  } |

3．构建Fruit2FruitMRRunner extends Configured implements Tool用于组装运行Job任务

|  |
| --- |
| //组装Job  public int run(String[] args) throws Exception {  //得到Configuration  Configuration conf = this.getConf();  //创建Job任务  Job job = Job.getInstance(conf, this.getClass().getSimpleName());  job.setJarByClass(Fruit2FruitMRRunner.class);  //配置Job  Scan scan = new Scan();  scan.setCacheBlocks(false);  scan.setCaching(500);  //设置Mapper，注意导入的是mapreduce包下的，不是mapred包下的，后者是老版本  TableMapReduceUtil.initTableMapperJob(  "fruit", //数据源的表名  scan, //scan扫描控制器  ReadFruitMapper.class,//设置Mapper类  ImmutableBytesWritable.class,//设置Mapper输出key类型  Put.class,//设置Mapper输出value值类型  job//设置给哪个JOB  );  //设置Reducer  TableMapReduceUtil.initTableReducerJob("fruit\_mr", WriteFruitMRReducer.class, job);  //设置Reduce数量，最少1个  job.setNumReduceTasks(1);  boolean isSuccess = job.waitForCompletion(true);  if(!isSuccess){  throw new IOException("Job running with error");  }  return isSuccess ? 0 : 1;  } |

4．主函数中调用运行该Job任务

|  |
| --- |
| public static void main( String[] args ) throws Exception{  Configuration conf = HBaseConfiguration.create();  int status = ToolRunner.run(conf, new Fruit2FruitMRRunner(), args);  System.exit(status);  } |

5．打包运行任务

$ /opt/module/hadoop-2.7.2/bin/yarn jar ~/softwares/jars/HBase-0.0.1-SNAPSHOT.jar

com.z.HBase.mr1.Fruit2FruitMRRunner

提示：运行任务前，如果待数据导入的表不存在，则需要提前创建。

提示：maven打包命令：-P local clean package或-P dev clean package install（将第三方jar包一同打包，需要插件：maven-shade-plugin）

## 7.3 自定义HBase-MapReduce2

目标：实现将HDFS中的数据写入到HBase表中。

分步实现：

1．构建ReadFruitFromHDFSMapper于读取HDFS中的文件数据

|  |
| --- |
| package com.woaini;  import java.io.IOException;  import org.apache.hadoop.HBase.client.Put;  import org.apache.hadoop.HBase.io.ImmutableBytesWritable;  import org.apache.hadoop.HBase.util.Bytes;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class ReadFruitFromHDFSMapper extends Mapper<LongWritable, Text, ImmutableBytesWritable, Put> {  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  //从HDFS中读取的数据  String lineValue = value.toString();  //读取出来的每行数据使用\t进行分割，存于String数组  String[] values = lineValue.split("\t");    //根据数据中值的含义取值  String rowKey = values[0];  String name = values[1];  String color = values[2];    //初始化rowKey  ImmutableBytesWritable rowKeyWritable = new ImmutableBytesWritable(Bytes.toBytes(rowKey));    //初始化put对象  Put put = new Put(Bytes.toBytes(rowKey));    //参数分别:列族、列、值  put.add(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes(name));  put.add(Bytes.toBytes("info"), Bytes.toBytes("color"), Bytes.toBytes(color));    context.write(rowKeyWritable, put);  }  } |

2．构建WriteFruitMRFromTxtReducer类

|  |
| --- |
| package com.z.HBase.mr2;  import java.io.IOException;  import org.apache.hadoop.HBase.client.Put;  import org.apache.hadoop.HBase.io.ImmutableBytesWritable;  import org.apache.hadoop.HBase.mapreduce.TableReducer;  import org.apache.hadoop.io.NullWritable;  public class WriteFruitMRFromTxtReducer extends TableReducer<ImmutableBytesWritable, Put, NullWritable> {  @Override  protected void reduce(ImmutableBytesWritable key, Iterable<Put> values, Context context) throws IOException, InterruptedException {  //读出来的每一行数据写入到fruit\_hdfs表中  for(Put put: values){  context.write(NullWritable.get(), put);  }  }  } |

3．创建Txt2FruitRunner组装Job

|  |
| --- |
| public int run(String[] args) throws Exception {  //得到Configuration  Configuration conf = this.getConf();  //创建Job任务  Job job = Job.getInstance(conf, this.getClass().getSimpleName());  job.setJarByClass(Txt2FruitRunner.class);  Path inPath = new Path("hdfs://hadoop102:9000/input\_fruit/fruit.tsv");  FileInputFormat.addInputPath(job, inPath);  //设置Mapper  job.setMapperClass(ReadFruitFromHDFSMapper.class);  job.setMapOutputKeyClass(ImmutableBytesWritable.class);  job.setMapOutputValueClass(Put.class);  //设置Reducer  TableMapReduceUtil.initTableReducerJob("fruit\_mr", WriteFruitMRFromTxtReducer.class, job);  //设置Reduce数量，最少1个  job.setNumReduceTasks(1);  boolean isSuccess = job.waitForCompletion(true);  if(!isSuccess){  throw new IOException("Job running with error");  }  return isSuccess ? 0 : 1;  } |

4．调用执行Job

|  |
| --- |
| public static void main(String[] args) throws Exception {  Configuration conf = HBaseConfiguration.create();  int status = ToolRunner.run(conf, new Txt2FruitRunner(), args);  System.exit(status);  } |

5．打包运行

$ /opt/module/hadoop-2.7.2/bin/yarn jar HBase-0.0.1-SNAPSHOT.jar com.woaini.HBase.mr2.Txt2FruitRunner

提示：运行任务前，如果待数据导入的表不存在，则需要提前创建之。

提示：maven打包命令：-P local clean package或-P dev clean package install（将第三方jar包一同打包，需要插件：maven-shade-plugin）

# 第 8 章 与Hive集成

### 8.1 HBase与Hive的对比

1．Hive

(1) 数据仓库

Hive的本质其实就相当于将HDFS中已经存储的文件在Mysql中做了一个双射关系，以方便使用HQL去管理查询。

(2) 用于数据分析、清洗

Hive适用于离线的数据分析和清洗，延迟较高。

(3) 基于HDFS、MapReduce

Hive存储的数据依旧在DataNode上，编写的HQL语句终将是转换为MapReduce代码执行。

2．HBase

(1) 数据库

是一种面向列存储的非关系型数据库。

(2) 用于存储结构化和非结构化的数据

适用于单表非关系型数据的存储，不适合做关联查询，类似JOIN等操作。

(3) 基于HDFS

数据持久化存储的体现形式是Hfile，存放于DataNode中，被ResionServer以region的形式进行管理。

(4) 延迟较低，接入在线业务使用

面对大量的企业数据，HBase可以直线单表大量数据的存储，同时提供了高效的数据访问速度。

### 8.2 HBase与Hive集成使用

尖叫提示：HBase与Hive的集成在最新的两个版本中无法兼容。所以，我们只能含着泪勇敢的重新编译：hive-HBase-handler-1.2.2.jar！！

**环境准备**

因为我们后续可能会在操作Hive的同时对HBase也会产生影响，所以Hive需要持有操作HBase的Jar，那么接下来拷贝Hive所依赖的Jar包（或者使用软连接的形式）。

|  |
| --- |
| export HBASE\_HOME=/opt/module/HBase  export HIVE\_HOME=/opt/module/hive  ln -s $HBASE\_HOME/lib/HBase-common-1.3.1.jar $HIVE\_HOME/lib/HBase-common-1.3.1.jar  ln -s $HBASE\_HOME/lib/HBase-server-1.3.1.jar $HIVE\_HOME/lib/HBase-server-1.3.1.jar  ln -s $HBASE\_HOME/lib/HBase-client-1.3.1.jar $HIVE\_HOME/lib/HBase-client-1.3.1.jar  ln -s $HBASE\_HOME/lib/HBase-protocol-1.3.1.jar $HIVE\_HOME/lib/HBase-protocol-1.3.1.jar  ln -s $HBASE\_HOME/lib/HBase-it-1.3.1.jar $HIVE\_HOME/lib/HBase-it-1.3.1.jar  ln -s $HBASE\_HOME/lib/htrace-core-3.1.0-incubating.jar $HIVE\_HOME/lib/htrace-core-3.1.0-incubating.jar  ln -s $HBASE\_HOME/lib/HBase-hadoop2-compat-1.3.1.jar $HIVE\_HOME/lib/HBase-hadoop2-compat-1.3.1.jar  ln -s $HBASE\_HOME/lib/HBase-hadoop-compat-1.3.1.jar $HIVE\_HOME/lib/HBase-hadoop-compat-1.3.1.jar |

同时在hive-site.xml中修改zookeeper的属性，如下：

|  |
| --- |
| <property>  <name>hive.zookeeper.quorum</name>  <value>hadoop102,hadoop103,hadoop104</value>  <description>The list of ZooKeeper servers to talk to. This is only needed for read/write locks.</description>  </property>  <property>  <name>hive.zookeeper.client.port</name>  <value>2181</value>  <description>The port of ZooKeeper servers to talk to. This is only needed for read/write locks.</description>  </property> |

1．案例一

**目标：**建立Hive表，关联HBase表，插入数据到Hive表的同时能够影响HBase表。

**分步实现：**

(1) 在Hive中创建表同时关联HBase

|  |
| --- |
| CREATE TABLE hive\_HBase\_emp\_table(  empno int,  ename string,  job string,  mgr int,  hiredate string,  sal double,  comm double,  deptno int)  STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")  TBLPROPERTIES ("hbase.table.name" = "hbase\_emp\_table"); |

提示：完成之后，可以分别进入Hive和HBase查看，都生成了对应的表

(2) 在Hive中创建临时中间表，用于load文件中的数据

提示：不能将数据直接load进Hive所关联HBase的那张表中

|  |
| --- |
| CREATE TABLE emp(  empno int,  ename string,  job string,  mgr int,  hiredate string,  sal double,  comm double,  deptno int)  row format delimited fields terminated by '\t'; |

(3) 向Hive中间表中load数据

|  |
| --- |
| hive> load data local inpath '/home/admin/softwares/data/emp.txt' into table emp; |

(4) 通过insert命令将中间表中的数据导入到Hive关联HBase的那张表中

|  |
| --- |
| hive> insert into table hive\_hbase\_emp\_table select \* from emp; |

(5) 查看Hive以及关联的HBase表中是否已经成功的同步插入了数据

Hive：

|  |
| --- |
| hive> select \* from hive\_HBase\_emp\_table; |

HBase：

|  |
| --- |
| HBase> scan ‘HBase\_emp\_table’ |

2．案例二

**目标：**在HBase中已经存储了某一张表HBase\_emp\_table，然后在Hive中创建一个外部表来关联HBase中的HBase\_emp\_table这张表，使之可以借助Hive来分析HBase这张表中的数据。

**注：**该案例2紧跟案例1的脚步，所以完成此案例前，请先完成案例1。

**分步实现：**

(1) 在Hive中创建外部表

|  |
| --- |
| CREATE EXTERNAL TABLE relevance\_HBase\_emp(  empno int,  ename string,  job string,  mgr int,  hiredate string,  sal double,  comm double,  deptno int)  STORED BY  'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  WITH SERDEPROPERTIES ("hbase.columns.mapping" =  ":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")  TBLPROPERTIES ("hbase.table.name" = "HBase\_emp\_table"); |

(2) 关联后就可以使用Hive函数进行一些分析操作了

|  |
| --- |
| hive (default)> select \* from relevance\_HBase\_emp; |

# 第9章 HBase优化

## 9.1 高可用

在HBase中Hmaster负责监控RegionServer的生命周期，均衡RegionServer的负载，如果Hmaster挂掉了，那么整个HBase集群将陷入不健康的状态，并且此时的工作状态并不会维持太久。所以HBase支持对Hmaster的高可用配置。

1．关闭HBase集群（如果没有开启则跳过此步）

[woaini@hadoop102 HBase]$ bin/stop-HBase.sh

2．在conf目录下创建backup-masters文件

[woaini@hadoop102 HBase]$ touch conf/backup-masters

3．在backup-masters文件中配置高可用HMaster节点

[woaini@hadoop102 HBase]$ echo hadoop103 > conf/backup-masters

4．将整个conf目录scp到其他节点

[woaini@hadoop102 HBase]$ scp -r conf/ hadoop103:/opt/module/HBase/

[woaini@hadoop102 HBase]$ scp -r conf/ hadoop104:/opt/module/HBase/

5．打开页面测试查看

[http://hadooo102:16010](http://linux01:16010)

## 9.2 预分区

每一个region维护着startRow与endRowKey，如果加入的数据符合某个region维护的rowKey范围，则该数据交给这个region维护。那么依照这个原则，我们可以将数据所要投放的分区提前大致的规划好，以提高HBase性能。

1．手动设定预分区

HBase> create 'staff1','info','partition1',SPLITS => ['1000','2000','3000','4000']

2．生成16进制序列预分区

create 'staff2','info','partition2',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

3．按照文件中设置的规则预分区

创建splits.txt文件内容如下：

|  |
| --- |
| aaaa  bbbb  cccc  dddd |

然后执行：

create 'staff3','partition3',SPLITS\_FILE => 'splits.txt'

4．使用JavaAPI创建预分区

|  |
| --- |
| //自定义算法，产生一系列Hash散列值存储在二维数组中  byte[][] splitKeys = 某个散列值函数  //创建HBaseAdmin实例  HBaseAdmin hAdmin = new HBaseAdmin(HBaseConfiguration.create());  //创建HTableDescriptor实例  HTableDescriptor tableDesc = new HTableDescriptor(tableName);  //通过HTableDescriptor实例和散列值二维数组创建带有预分区的HBase表  hAdmin.createTable(tableDesc, splitKeys); |

## 9.3 RowKey设计

一条数据的唯一标识就是rowkey，那么这条数据存储于哪个分区，取决于rowkey处于哪个一个预分区的区间内，设计rowkey的主要目的 ，就是让数据均匀的分布于所有的region中，在一定程度上防止数据倾斜。接下来我们就谈一谈rowkey常用的设计方案。

1．生成随机数、hash、散列值

|  |
| --- |
| 原本rowKey为1001的，SHA1后变成：dd01903921ea24941c26a48f2cec24e0bb0e8cc7  原本rowKey为3001的，SHA1后变成：49042c54de64a1e9bf0b33e00245660ef92dc7bd  原本rowKey为5001的，SHA1后变成：7b61dec07e02c188790670af43e717f0f46e8913  在做此操作之前，一般我们会选择从数据集中抽取样本，来决定什么样的rowKey来Hash后作为每个分区的临界值。 |

2．字符串反转

|  |
| --- |
| 20170524000001转成10000042507102  20170524000002转成20000042507102 |

这样也可以在一定程度上散列逐步put进来的数据。

3．字符串拼接

|  |
| --- |
| 20170524000001\_a12e  20170524000001\_93i7 |

## 9.4 内存优化

HBase操作过程中需要大量的内存开销，毕竟Table是可以缓存在内存中的，一般会分配整个可用内存的70%给HBase的Java堆。但是不建议分配非常大的堆内存，因为GC过程持续太久会导致RegionServer处于长期不可用状态，一般16~48G内存就可以了，如果因为框架占用内存过高导致系统内存不足，框架一样会被系统服务拖死。

## 9.5 基础优化

1．允许在HDFS的文件中追加内容

hdfs-site.xml、HBase-site.xml

|  |
| --- |
| 属性：dfs.support.append  解释：开启HDFS追加同步，可以优秀的配合HBase的数据同步和持久化。默认值为true。 |

2．优化DataNode允许的最大文件打开数

hdfs-site.xml

|  |
| --- |
| 属性：dfs.datanode.max.transfer.threads  解释：HBase一般都会同一时间操作大量的文件，根据集群的数量和规模以及数据动作，设置为4096或者更高。默认值：4096 |

3．优化延迟高的数据操作的等待时间

hdfs-site.xml

|  |
| --- |
| 属性：dfs.image.transfer.timeout  解释：如果对于某一次数据操作来讲，延迟非常高，socket需要等待更长的时间，建议把该值设置为更大的值（默认60000毫秒），以确保socket不会被timeout掉。 |

4．优化数据的写入效率

mapred-site.xml

|  |
| --- |
| 属性：  mapreduce.map.output.compress  mapreduce.map.output.compress.codec  解释：开启这两个数据可以大大提高文件的写入效率，减少写入时间。第一个属性值修改为true，第二个属性值修改为：org.apache.hadoop.io.compress.GzipCodec或者其他压缩方式。 |

5．设置RPC监听数量

HBase-site.xml

|  |
| --- |
| 属性：HBase.regionserver.handler.count  解释：默认值为30，用于指定RPC监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。 |

6．优化HStore文件大小

HBase-site.xml

|  |
| --- |
| 属性：HBase.hregion.max.filesize  解释：默认值10737418240（10GB），如果需要运行HBase的MR任务，可以减小此值，因为一个region对应一个map任务，如果单个region过大，会导致map任务执行时间过长。该值的意思就是，如果HFile的大小达到这个数值，则这个region会被切分为两个Hfile。 |

7．优化HBase客户端缓存

HBase-site.xml

|  |
| --- |
| 属性：HBase.client.write.buffer  解释：用于指定HBase客户端缓存，增大该值可以减少RPC调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少RPC次数的目的。 |

8．指定scan.next扫描HBase所获取的行数

HBase-site.xml

|  |
| --- |
| 属性：HBase.client.scanner.caching  解释：用于指定scan.next方法获取的默认行数，值越大，消耗内存越大。 |

9．flush、compact、split机制

当MemStore达到阈值，将Memstore中的数据Flush进Storefile；compact机制则是把flush出来的小文件合并成大的Storefile文件。split则是当Region达到阈值，会把过大的Region一分为二。

**涉及属性：**

即：128M就是Memstore的默认阈值

|  |
| --- |
| HBase.hregion.memstore.flush.size：134217728 |

即：这个参数的作用是当单个HRegion内所有的Memstore大小总和超过指定值时，flush该HRegion的所有memstore。RegionServer的flush是通过将请求添加一个队列，模拟生产消费模型来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发OOM。

|  |
| --- |
| HBase.regionserver.global.memstore.upperLimit：0.4  HBase.regionserver.global.memstore.lowerLimit：0.38 |

即：当MemStore使用内存总量达到HBase.regionserver.global.memstore.upperLimit指定值时，将会有多个MemStores flush到文件中，MemStore flush 顺序是按照大小降序执行的，直到刷新到MemStore使用内存略小于lowerLimit

# 第10章 扩展

## 11.1 HBase在商业项目中的能力

**每天：**

1) 消息量：发送和接收的消息数超过60亿

2) 将近1000亿条数据的读写

3) 高峰期每秒150万左右操作

4) 整体读取数据占有约55%，写入占有45%

5) 超过2PB的数据，涉及冗余共6PB数据

6) 数据每月大概增长300千兆字节。

## 11.2 布隆过滤器

在日常生活中，包括在设计计算机软件时，我们经常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是要判断它是否在已知的字典中）；在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上；在网络爬虫里，一个网址是否被访问过等等。最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表（hash table）来存储的。它的好处是快速准确，缺点是费存储空间。当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来了。比如说，一个像 Yahoo,Hotmail 和 Gmai 那样的公众电子邮件（email）提供商，总是需要过滤来自发送垃圾邮件的人（spamer）的垃圾邮件。一个办法就是记录下那些发垃圾邮件的 email 地址。由于那些发送者不停地在注册新的地址，全世界少说也有几十亿个发垃圾邮件的地址，将他们都存起来则需要大量的网络服务器。如果用哈希表，每存储一亿个 email 地址， 就需要 1.6GB 的内存（用哈希表实现的具体办法是将每一个 email 地址对应成一个八字节的信息指纹googlechinablog.com/2006/08/blog-post.html，然后将这些信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此一个 email 地址需要占用十六个字节。一亿个地址大约要 1.6GB， 即十六亿字节的内存）。因此存贮几十亿个邮件地址可能需要上百 GB 的内存。除非是超级计算机，一般服务器是无法存储的。

布隆过滤器只需要哈希表 1/8 到 1/4 的大小就能解决同样的问题。

Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

下面我们具体来看Bloom Filter是如何用位数组表示集合的。初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0，如图9-5所示。



图9-5

为了表达S={x1, x2,…,xn}这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素x，第i个哈希函数映射的位置hi(x)就会被置为1（1≤i≤k）。注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。如图9-6所示，k=3，且有两个哈希函数选中同一个位置（从左边数第五位）。

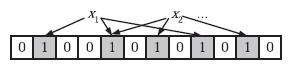


图9-6

在判断y是否属于这个集合时，我们对y应用k次哈希函数，如果所有hi(y)的位置都是1（1≤i≤k），那么我们就认为y是集合中的元素，否则就认为y不是集合中的元素。如图9-7所示y1就不是集合中的元素。y2或者属于这个集合，或者刚好是一个false positive。

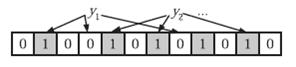


图9-7

· 为了add一个元素，用k个hash function将它hash得到bloom filter中k个bit位，将这k个bit位置1。

· 为了query一个元素，即判断它是否在集合中，用k个hash function将它hash得到k个bit位。若这k bits全为1，则此元素在集合中；若其中任一位不为1，则此元素比不在集合中（因为如果在，则在add时已经把对应的k个bits位置为1）。

· 不允许remove元素，因为那样的话会把相应的k个bits位置为0，而其中很有可能有其他元素对应的位。因此remove会引入false negative，这是绝对不被允许的。

布隆过滤器决不会漏掉任何一个在黑名单中的可疑地址。但是，它有一条不足之处，也就是它有极小的可能将一个不在黑名单中的电子邮件地址判定为在黑名单中，因为有可能某个好的邮件地址正巧对应一个八个都被设置成一的二进制位。好在这种可能性很小，我们把它称为误识概率。

布隆过滤器的好处在于快速，省空间，但是有一定的误识别率，常见的补救办法是在建立一个小的白名单，存储那些可能个别误判的邮件地址。

布隆过滤器具体算法高级内容，如错误率估计，最优哈希函数个数计算，位数组大小计算，请参见<http://blog.csdn.net/jiaomeng/article/details/1495500>。

## 11.3 HBase2.0新特性

2017年8月22日凌晨2点左右，HBase发布了2.0.0 alpha-2，相比于上一个版本，修复了500个补丁，我们来了解一下2.0版本的HBase新特性。

最新文档：

[http://HBase.apache.org/book.html#ttl](http://hbase.apache.org/book.html" \l "ttl)

官方发布主页：

[http://mail-archives.apache.org/mod\_mbox/www-announce/201706.mbox/<CADcMMgFzmX0xYYso-UAYbU7V8z-Obk1J4pxzbGkRzbP5Hps+iA@mail.gmail.com](http://mail-archives.apache.org/mod_mbox/www-announce/201708.mbox/%3cCADcMMgFzmX0xYYso-UAYbU7V8z-Obk1J4pxzbGkRzbP5Hps+iA@mail.gmail.com)

**举例：**

**1) region进行了多份冗余**

主region负责读写，从region维护在其他HregionServer中，负责读以及同步主region中的信息，如果同步不及时，是有可能出现client在从region中读到了脏数据（主region还没来得及把memstore中的变动的内容flush）。

**2) 更多变动**

<https://issues.apache.org/jira/secure/ReleaseNote.jspa?version=12340859&styleName=&projectId=12310753&Create=Create&atl_token=A5KQ-2QAV-T4JA-FDED%7Ce6f233490acdf4785b697d4b457f7adb0a72b69f%7Clout>