目录

[1. hbase简介 1](#_Toc421124195)

[1.1. 什么是hbase 1](#_Toc421124196)

[1.2. 与传统数据库的对比 1](#_Toc421124197)

[1.3. hbase集群中的角色 1](#_Toc421124198)

[2. habse安装 2](#_Toc421124199)

[2.1. hbase安装 2](#_Toc421124200)

[2.1.1. 上传 2](#_Toc421124201)

[2.1.2. 解压 2](#_Toc421124202)

[2.1.3. 重命名 2](#_Toc421124203)

[2.1.4. 修改环境变量(每台机器都要执行) 2](#_Toc421124204)

[2.1.5. 修改配置文件 2](#_Toc421124205)

[2.1.6. 分发到其他节点 2](#_Toc421124206)

[2.1.7. 启动 3](#_Toc421124207)

[2.1.8. 监控 3](#_Toc421124208)

[3. hbase数据模型 3](#_Toc421124209)

[3.1. hbase数据模型 3](#_Toc421124210)

[3.1.1. Row Key 3](#_Toc421124211)

[3.1.2. Columns Family 4](#_Toc421124212)

[3.1.3. Cell 4](#_Toc421124213)

[3.1.4. Time Stamp 4](#_Toc421124214)

[4. hbase命令 4](#_Toc421124215)

[4.1. 命令的进退 4](#_Toc421124216)

[4.2. 命令 5](#_Toc421124217)

[5. hbase依赖zookeeper 6](#_Toc421124218)

[6. hbase开发 6](#_Toc421124219)

[6.1. 配置 6](#_Toc421124220)

[6.2. 表管理类 7](#_Toc421124221)

[6.3. 表描述类 7](#_Toc421124222)

[6.4. 列族的描述类 7](#_Toc421124223)

[6.5. 创建表的操作 7](#_Toc421124224)

[6.6. 删除表 8](#_Toc421124225)

[6.7. 创建一个表的类 8](#_Toc421124226)

[6.8. 单条插入数据 8](#_Toc421124227)

[6.9. 批量插入 9](#_Toc421124228)

[6.10. 删除数据 9](#_Toc421124229)

[6.11. 单条查询 9](#_Toc421124230)

[6.12. 批量查询 10](#_Toc421124231)

[6.13. hbase过滤器 10](#_Toc421124232)

[6.13.1. FilterList 10](#_Toc421124233)

[6.13.2. 过滤器的种类 11](#_Toc421124234)

[6.13.3. 列植过滤器—SingleColumnValueFilter 11](#_Toc421124235)

[6.13.4. 列名前缀过滤器—ColumnPrefixFilter 12](#_Toc421124236)

[6.13.5. 多个列值前缀过滤器—MultipleColumnPrefixFilter 12](#_Toc421124237)

[6.13.6. rowKey过滤器—RowFilter 12](#_Toc421124238)

[7. hbase原理 12](#_Toc421124239)

[7.1. 体系图 12](#_Toc421124240)

[7.1.1. 写流程 13](#_Toc421124241)

[7.1.2. 数据flush过程 13](#_Toc421124242)

[7.1.3. hbase的读流程 13](#_Toc421124243)

[7.1.4. hmaster的职责 13](#_Toc421124244)

[7.1.5. hregionserver的职责 13](#_Toc421124245)

[7.1.6. client职责 14](#_Toc421124246)

[8. 总结 14](#_Toc421124247)

hbase

# hbase简介

## 什么是hbase

HBASE是一个高可靠性、高性能、**面向列**、可伸缩的**分布式存储系统，（写快读慢）利用HBASE**技术可在廉价PC Server上**搭建起大规模结构化存储集群**。

HBASE的目标是存储并处理大型的数据，更具体来说是仅需使用普通的硬件配置，就能够处理由成千上万的行和列所组成的大型数据。

HBASE是Google Bigtable的开源实现，但是也有很多不同之处。比如：Google Bigtable利用GFS作为其文件存储系统**，HBASE利用Hadoop HDFS作为其文件存储系统；**Google运行MAPREDUCE来处理Bigtable中的海量数据，**HBASE同样利用Hadoop MapReduce来处理HBASE中的海量数据；**Google Bigtable利用Chubby作为协同服务，**HBASE利用Zookeeper作为对应。**

## 与传统数据库的对比

1、传统数据库遇到的问题：

1）数据量很大的时候无法存储

2）没有很好的备份机制

3）数据达到一定数量开始缓慢，很大的话基本无法支撑

 2、HBASE优势：

1）线性扩展，随着数据量增多可以通过**节点扩展**进行支撑

2）数据存储在hdfs上，备份机制健全

3）通过zookeeper协调查找数据，访问速度块。

## hbase集群中的角色

1、一个或者多个主节点，Hmaster

2、多个从节点，HregionServer

# habse安装

## hbase安装

### 上传

用工具上传

### 解压

su – hadoop

tar -zxvf hbase-0.94.6.tar.gz

### 重命名

mv hbase-0.94.6 hbase

### 修改环境变量(每台机器都要执行)

su – root

vi /etc/profile

添加内容：

|  |
| --- |
| export HBASE\_HOME=/home/hadoop/hbase  export PATH=$PATH:$HBASE\_HOME/bin |

source /etc/proflie

su - hadoop

### 修改配置文件

上传配置文件

### 分发到其他节点

scp -r /home/hadoop/hbase hadoop@slave1:/home/hadoop/

scp -r /home/hadoop/hbase hadoop@slave2:/home/hadoop/

scp -r /home/hadoop/hbase hadoop@slave3:/home/hadoop/

### 启动

注意：启动hbase之前，必须保证hadoop集群和zookeeper集群是可用的。

start-hbase.sh

### 监控

1. 进入命令行

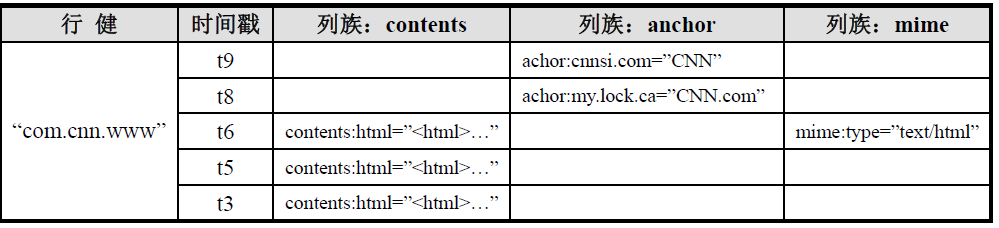
hbase shell

1. 页面监控

<http://master:60010/>

# hbase数据模型

## hbase数据模型



### Row Key

与nosql数据库们一样,row key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（正则）

3.全表扫描

Row key行键 (Row key)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)，**在HBASE内部，row key保存为字节数组。存储时，数据按照Row key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)

### Columns Family

列簇 ：HBASE表中的每个列，都归属于某个列族。列族是表的schema的一部 分(而列不是)，必须在使用表之前定义。**列名都以列族作为前缀**。例如 courses:history，courses:math都属于courses 这个列族。

### Cell

由{row key, columnFamily, 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。**

关键字：**无类型、字节码**

### Time Stamp

HBASE 中通过rowkey和columns确定的为一个存贮单元称为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。一是保存数据的最后n个版本，二是保存最近一段 时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

# hbase命令

## 命令的进退

1、hbase提供了一个shell的终端给用户交互

#$HBASE\_HOME/bin/hbase shell

2、如果退出执行quit命令

#$HBASE\_HOME/bin/hbase shell

……

>quit

## 命令

|  |  |
| --- | --- |
| **名称** | **命令表达式** |
| **创建表** | **create '表名', '列族名1','列族名2','列族名N'** |
| **查看所有表** | **list** |
| **描述表** | **describe ‘表名’** |
| 判断表存在 | exists '表名' |
| 判断是否禁用启用表 | is\_enabled '表名'  is\_disabled ‘表名’ |
| **添加记录** | **put ‘表名’, ‘rowKey’, ‘列族 : 列‘ , '值'** |
| **查看记录rowkey下的所有数据** | **get '表名' , 'rowKey'** |
| **查看表中的记录总数** | **count '表名'** |
| **获取某个列族** | get '表名','rowkey','列族' |
| **获取某个列族的某个列** | get '表名','rowkey','列族：列’ |
| **删除记录** | **delete ‘表名’ ,‘行名’ , ‘列族：列'** |
| **删除整行** | **deleteall '表名','rowkey'** |
| **删除一张表** | **先要屏蔽该表，才能对该表进行删除**  **第一步 disable ‘表名’ ，第二步 drop '表名'** |
| **清空表** | **truncate '表名'** |
| **查看所有记录** | **scan "表名"** |
| **查看某个表某个列中所有数据** | **scan "表名" , {COLUMNS=>'列族名:列名'}** |
| **更新记录** | **就是重写一遍，进行覆盖，hbase没有修改，都是追加** |

# hbase依赖zookeeper

1. 保存Hmaster的地址和backup-master地址

hmaster：

1. 管理HregionServer
2. 做增删改查表的节点
3. 管理HregionServer中的表分配
4. 保存表-ROOT-的地址

hbase默认的根表，检索表。

1. HRegionServer列表

表的增删改查数据。

和hdfs交互，存取数据。

# hbase开发

## 配置

HBaseConfiguration

包：org.apache.hadoop.hbase.HBaseConfiguration

作用：通过此类可以对HBase进行配置

用法实例：

Configuration config = HBaseConfiguration.create();

说明： **HBaseConfiguration.create() 默认会从classpath 中查找 hbase-site.xml 中的配置信息，初始化 Configuration。**

使用方法:

static Configuration config = null;

static {

config = HBaseConfiguration.create();

config.set("hbase.zookeeper.quorum", "slave1,slave2,slave3");

config.set("hbase.zookeeper.property.clientPort", "2181");

}

## 表管理类

HBaseAdmin

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

作用：提供接口关系HBase 数据库中的表信息

用法：

HBaseAdmin admin = new HBaseAdmin(config);

## 表描述类

HTableDescriptor

包：org.apache.hadoop.hbase.HTableDescriptor

作用：HTableDescriptor 类包含了表的名字以及表的列族信息

表的schema（设计）

用法：

HTableDescriptor htd =new HTableDescriptor(tablename);

htd.addFamily(new HColumnDescriptor(“myFamily”));

## 列族的描述类

HColumnDescriptor

包：org.apache.hadoop.hbase.HColumnDescriptor

作用：HColumnDescriptor 维护列族的信息

用法：

htd.addFamily(new HColumnDescriptor(“myFamily”));

## 创建表的操作

CreateTable（一般我们用shell创建表）

static Configuration config = null;

static {

config = HBaseConfiguration.create();

config.set("hbase.zookeeper.quorum", "slave1,slave2,slave3");

config.set("hbase.zookeeper.property.clientPort", "2181");

}

HBaseAdmin admin = new HBaseAdmin(config);

HTableDescriptor desc = new HTableDescriptor(tableName);

HColumnDescriptor family1 = new HColumnDescriptor(“f1”);

HColumnDescriptor family2 = new HColumnDescriptor(“f2”);

desc.addFamily(family1);

desc.addFamily(family2);

**admin.createTable(desc);**

## 删除表

HBaseAdmin admin = new HBaseAdmin(config);

admin.disableTable(tableName);

admin.deleteTable(tableName);

## 创建一个表的类

HTable

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

作用：HTable 和 HBase 的表通信

用法：

// 普通获取表

HTable table = new HTable(config,Bytes.toBytes(tablename);

// 通过连接池获取表

Connection connection = ConnectionFactory.createConnection(config);

HTableInterface table = connection.getTable(TableName.valueOf("user"));

## 单条插入数据

Put

包：org.apache.hadoop.hbase.client.Put

作用：插入数据

用法：

Put put = new Put(row);

p.add(family,qualifier,value);

说明：向表 tablename 添加 “family,qualifier,value”指定的值。

示例代码：

Connection connection = ConnectionFactory.createConnection(config);

HTableInterface table = connection.getTable(TableName.valueOf("user"));

Put put = new Put(Bytes.toBytes(rowKey));

put.add(Bytes.toBytes(family), Bytes.toBytes(qualifier),Bytes.toBytes(value));

table.put(put);

## 批量插入

批量插入

List<Put> list = new ArrayList<Put>();

Put put = new Put(Bytes.toBytes(rowKey));//获取put，用于插入

put.add(Bytes.toBytes(family), Bytes.toBytes(qualifier),Bytes.toBytes(value));//封装信息

list.add(put);

table.put(list);//添加记录

## 删除数据

Delete

包：org.apache.hadoop.hbase.client.Delete

作用：删除给定rowkey的数据

用法：

Delete del= new Delete(Bytes.toBytes(rowKey));

table.delete(del);

代码实例

Connection connection = ConnectionFactory.createConnection(config);

HTableInterface table = connection.getTable(TableName.valueOf("user"));

Delete del= new Delete(Bytes.toBytes(rowKey));

table.delete(del);

## 单条查询

Get

包：org.apache.hadoop.hbase.client.Get

作用：获取单个行的数据

用法：

HTable table = new HTable(config,Bytes.toBytes(tablename));

Get get = new Get(Bytes.toBytes(row));

Result result = table.get(get);

说明：获取 tablename 表中 row 行的对应数据

代码示例：

Connection connection = ConnectionFactory.createConnection(config);

HTableInterface table = connection.getTable(TableName.valueOf("user"));

Get get = new Get(rowKey.getBytes());

Result row = table.get(get);

for (KeyValue kv : row.raw()) {

System.out.print(new String(kv.getRow()) + " ");

System.out.print(new String(kv.getFamily()) + ":");

System.out.print(new String(kv.getQualifier()) + " = ");

System.out.print(new String(kv.getValue()));

System.out.print(" timestamp = " + kv.getTimestamp() + "\n");

}

## 批量查询

ResultScanner

包：org.apache.hadoop.hbase.client.ResultScanner

作用：获取值的接口

用法：

ResultScanner scanner = table.getScanner(scan);

For(Result rowResult : scanner){

 Bytes[] str = rowResult.getValue(family,column);

}

说明：循环获取行中列值。

代码示例：

Connection connection = ConnectionFactory.createConnection(config);

HTableInterface table = connection.getTable(TableName.valueOf("user"));

Scan scan = new Scan();

scan.setStartRow("a1".getBytes());

scan.setStopRow("a20".getBytes());

ResultScanner scanner = table.getScanner(scan);

for (Result row : scanner) {

System.out.println("\nRowkey: " + new String(row.getRow()));

for (KeyValue kv : row.raw()) {

System.out.print(new String(kv.getRow()) + " ");

System.out.print(new String(kv.getFamily()) + ":");

System.out.print(new String(kv.getQualifier()) + " = ");

System.out.print(new String(kv.getValue()));

System.out.print(" timestamp = " + kv.getTimestamp() + "\n");

}

}

## hbase过滤器

### FilterList

FilterList 代表一个过滤器列表，可以添加多个过滤器进行查询，多个过滤器之间的关系有：

与关系（符合所有）：FilterList.Operator.MUST\_PASS\_ALL

或关系（符合任一）：FilterList.Operator.MUST\_PASS\_ONE

使用方法：

FilterList filterList = new FilterList(FilterList.Operator.MUST\_PASS\_ONE);

Scan s1 = new Scan();

 filterList.addFilter(new SingleColumnValueFilter(Bytes.toBytes(“f1”),  Bytes.toBytes(“c1”),  CompareOp.EQUAL,Bytes.toBytes(“v1”) )  );

filterList.addFilter(new SingleColumnValueFilter(Bytes.toBytes(“f1”),  Bytes.toBytes(“c2”),  CompareOp.EQUAL,Bytes.toBytes(“v2”) )  );

 // 添加下面这一行后，则只返回指定的cell，同一行中的其他cell不返回

s1.addColumn(Bytes.toBytes(“f1”), Bytes.toBytes(“c1”));

 s1.setFilter(filterList);  //设置filter

 ResultScanner ResultScannerFilterList = table.getScanner(s1);  //返回结果列表

### 过滤器的种类

过滤器的种类：

列植过滤器—SingleColumnValueFilter

过滤列植的相等、不等、范围等

列名前缀过滤器—ColumnPrefixFilter

过滤指定前缀的列名

多个列名前缀过滤器—MultipleColumnPrefixFilter

过滤多个指定前缀的列名

rowKey过滤器—RowFilter

通过正则，过滤rowKey值。

### 列植过滤器—SingleColumnValueFilter

SingleColumnValueFilter 列值判断

相等 (CompareOp.EQUAL ),

不等(CompareOp.NOT\_EQUAL),

范围 (e.g., CompareOp.GREATER)…………

下面示例检查列值和字符串'values' 相等...

SingleColumnValueFilter f = new SingleColumnValueFilter(

Bytes.toBytes("cFamily") Bytes.toBytes("column"), CompareFilter.CompareOp.EQUAL,

Bytes.toBytes("values"));

s1.setFilter(f);

注意：如果过滤器过滤的列在数据表中有的行中不存在，那么这个过滤器对此行无法过滤。

### 列名前缀过滤器—ColumnPrefixFilter

过滤器—ColumnPrefixFilter

ColumnPrefixFilter 用于指定列名前缀值相等

ColumnPrefixFilter f = new ColumnPrefixFilter(Bytes.toBytes("values"));

s1.setFilter(f);

### 多个列值前缀过滤器—MultipleColumnPrefixFilter

MultipleColumnPrefixFilter 和 ColumnPrefixFilter 行为差不多，但可以指定多个前缀

byte[][] prefixes = new byte[][] {Bytes.toBytes("value1"),Bytes.toBytes("value2")};

Filter f = new MultipleColumnPrefixFilter(prefixes);

s1.setFilter(f);

### rowKey过滤器—RowFilter

RowFilter 是rowkey过滤器

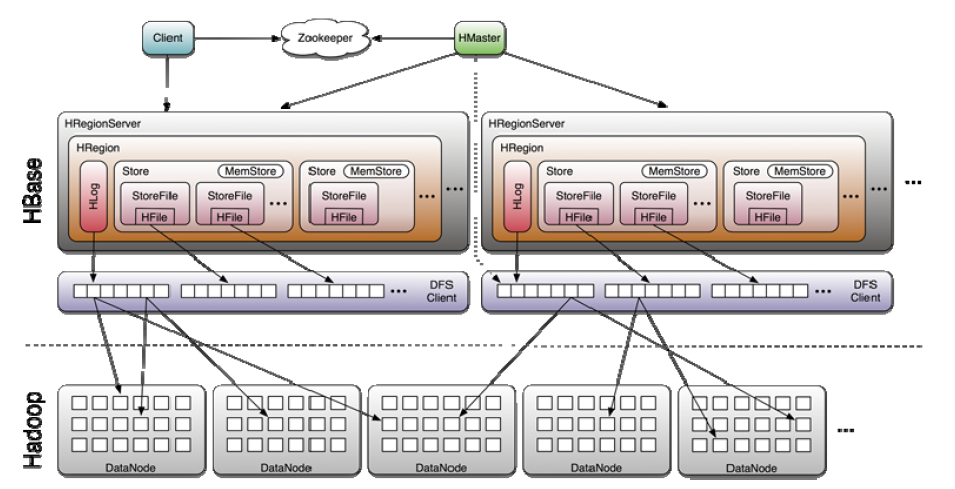
通常根据rowkey来指定范围时，使用scan扫描器的StartRow和StopRow方法比较好。

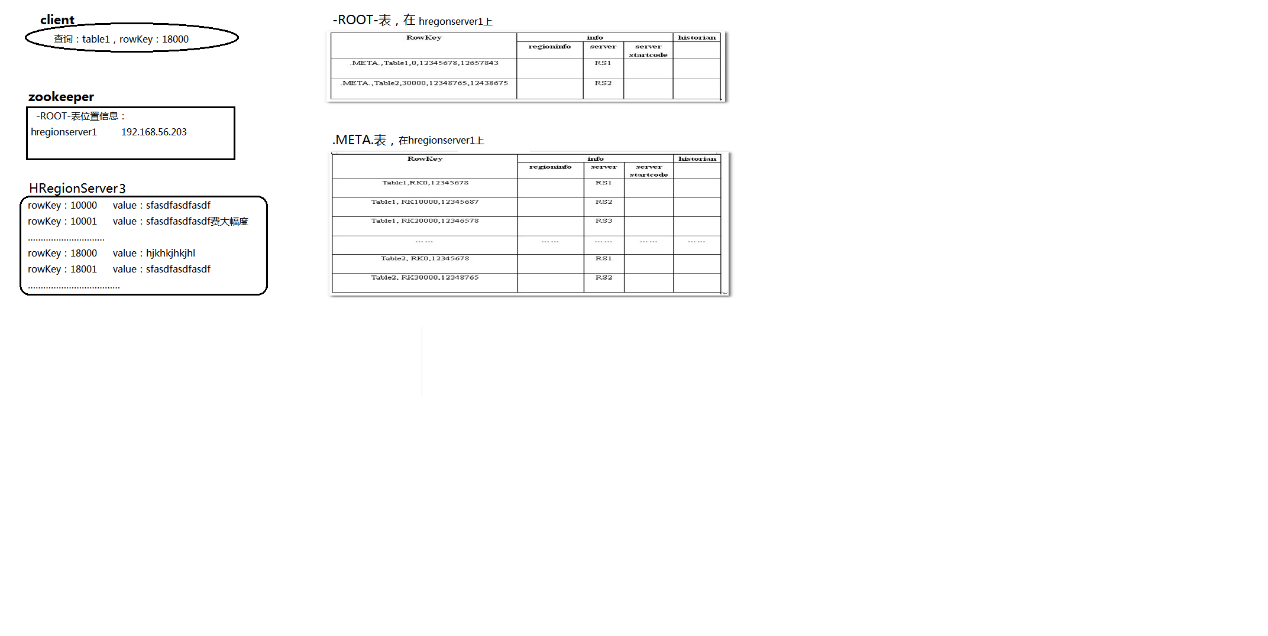
Filter f = new RowFilter(CompareFilter.CompareOp.EQUAL, new RegexStringComparator("^1234")); //匹配以1234开头的rowkey

s1.setFilter(f);

# hbase原理

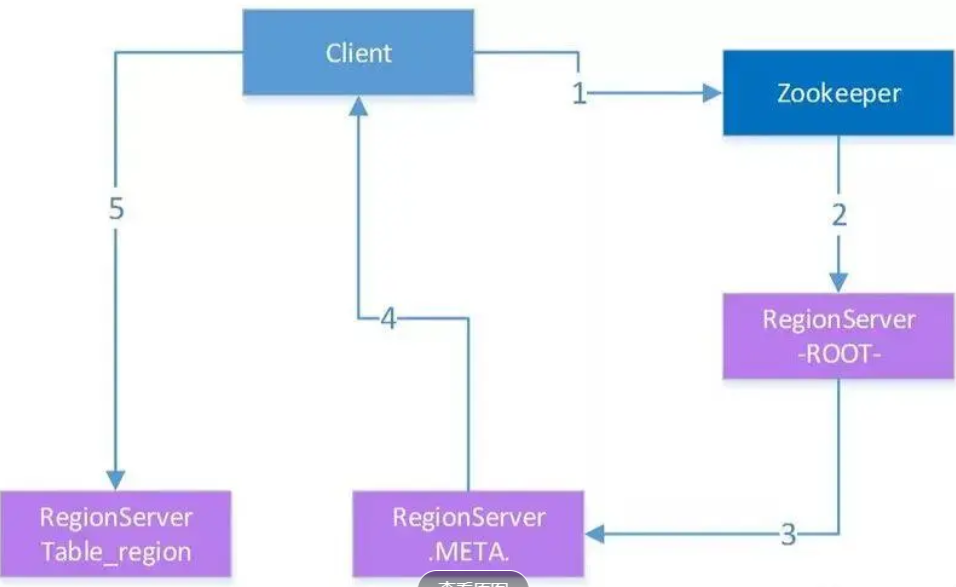
## 体系图





### RegionServer定位

访问HBase通过HBase客户端(或API)进行，整个HBase提供给外部的地址，其实是ZK的入口，前面也介绍了，ZK中有保存 -ROOT-所在的RS地址，从-ROOT-表可以获取.META.表信息，根据.META.表可以获取region在RS上的分布，整个region寻 址过程大致如下：

1.   
     
   首先，Client通过访问ZK来请求目标数据的地址。
2. ZK中保存了-ROOT-表的地址，所以ZK通过访问-ROOT-表来请求数据地址。
3. 同样，-ROOT-表中保存的是.META.的信息，通过访问.META.表来获取具体的RS。
4. .META.表查询到具体RS信息后返回具体RS地址给Client。
5. Client端获取到目标地址后，然后直接向该地址发送数据请求。

这里有几个问题：

* 既然ZK中能保存-ROOT-信息，那么为什么不把.META.信息直接保存在ZK中，而需要通过-ROOT-表来定位？
* Client查找到目标地址后，下一次请求还需要走ZK –> -ROOT- –>.META.这个流程么？

先来回答第一个问题：为什么不直接把.META.表信息直接保存到ZK中？主要是为了保存的数据量考虑，**ZK中不宜保存大量数据，**而**.META.表 主要是保存Region和RS的映射信息**，region的数量没有具体约束，只要在内存允许的范围内，region数量可以有很多，如果保存在ZK 中，ZK的压力会很大。所以，通过一个-ROOT-表来转存到RS中是一个比较理想的方案，相比直接保存在ZK中，也就多了一层-ROOT-表的查询，对 性能来说影响不大。

第二个问题：每次访问都需要走ZK –> -ROOT- —> .META.的流程么？当然不需要，Client端有缓存，第一次查询到相应region所在RS后，这个信息将被缓存到Client端，以后每次访问都 直接从缓存中获取RS地址即可。当然这里有个意外：访问的region若果在RS上发生了改变，比如被balancer迁移到其他RS上了，这个时候，通 过缓存的地址访问会出现异常，在出现异常的情况下，Client需要重新走一遍上面的流程来获取新的RS地址。总体来说，region的变动只会在极少数 情况下发生，一般变动不会很大，所以在整个集群访问过程中，影响可以忽略。

上述过程其实是一个三层索引结构，从ZK获取-ROOT-信息，再从-ROOT-获取.META.表信息，最后从.META.表中查到RS地址后缓存。

HBase通过ZK –> -ROOT- –>.META.的访问获取RS地址后，直接向该RS上进行数据写入操作，Client通过三层索引获得RS的地址后，即可向指定RS的对应region进行数据写入，HBase的数据写入采用WAL(write ahead log)的形式，先写log，后写数据。HBase是一个append类型的数据库，没有关系型数据库那么复杂的操作，所以记录HLog的操作都是简单的 put操作(delete/update操作都被转化为put进行).

**每个RS上的region都共享一个HLog，**所有对于该RS上的 region数据写入都被记录到该HLog中。HLog的主要作用就是在RS出现意外崩溃的时候，可以尽量多的恢复数据，这里说是尽量多，因为在一般情况 下，客户端为了提高性能，会把HLog的auto flush关掉，这样HLog日志的落盘全靠操作系统保证，如果出现意外崩溃，短时间内没有被fsync的日志会被丢失。

HLog的大量写入会造成HLog占用存储空间会越来越大，HBase通过HLog过期的方式进行HLog的清理，**每个RS内部都有一个HLog监控线程在运行，其周期可以通过hbase.master.cleaner.interval进行配置。**

**HLog在数据从memstore flush到底层存储上后，说明该段HLog已经不再被需要，**就会被移动到.oldlogs这个目录下，HLog监控线程监控该目录下的HLog，当该文 件夹下的HLog达到hbase.master.logcleaner.ttl设置的过期条件后，监控线程立即删除过期的HLog。  
**memstore是region内部缓存**，其大小通过HBase参数hbase.hregion.memstore.flush.size进行配 置。RS在写完HLog以后，数据写入的下一个目标就是region的memstore，memstore在HBase内部通过LSM-tree结构组 织，所以能够合并大量对于相同rowkey上的更新操作。

正是由于memstore的存在，HBase的数据写入都是异步的，而且性能非常不错，写入到memstore后，该次写入请求就可以被返 回，HBase即认为该次数据写入成功。这里有一点需要说明，**写入到memstore中的数据都是预先按照rowkey的值进行排序的，**这样有利于后续数 据查找。

memstore的数据刷盘，对region的直接影响就是：**在数据刷盘开始到结束这段时间内，该region上的访问都是被拒绝的**，这里主要是因 为在数据刷盘结束时，RS会对改region做一个snapshot，同时HLog做一个checkpoint操作，通知ZK哪些HLog可以被移 到.oldlogs下。从前面图上也可以看到，**在memstore写盘开始，相应region会被加上UpdateLock锁，写盘结束后该锁被释放。**

memstore在触发刷盘操作后会被写入底层存储，**每次memstore的刷盘就会相应生成一个存储文件HFile，storeFile即HFile在HBase层的轻量级分装。**数据量的持续写入，造成memstore的频繁flush，每次flush都会产生一个HFile，这样底层存储设备上的HFile文件数量将会越 来越多。不管是HDFS还是Linux下常用的文件系统如Ext4、XFS等，对小而多的文件上的管理都没有大文件来的有效，比如小文件打开需要消耗更多 的文件句柄；在大量小文件中进行指定rowkey数据的查询性能没有在少量大文件中查询来的快等等。

Hbase中的每张表都通过键按照一定的范围被分割成多个子表（HRegion），默认一个**HRegion超过256M就要被分割成两个**，这个过程由HRegionServer管理,**而HRegion的分配由HMaster管理**。Client访问Hbase上的数据并不需要HMaster参与，寻址访问ZooKeeper和HRegionServer，数据读写访问HRegionServer，**HMaster仅仅维护Table和Region的元数据信息，**Table的元数据信息保存在ZooKeeper上，负载很低。**HRegionServer存取一个子表时，会创建一个HRegion对象，然后对表的每个列簇创建一个Store对象，每个Store都会有一个MemStore和0或多个StoreFile与之对应，每个StoreFile都会对应一个HFile，HFile就是实际的存储文件**。因此，一个HRegion**有多少列簇就有多少个Store**。一个HRegionServer会有多个HRegion和**一个HLog**。

Table在行的方向上分割为多个HRegion，HRegion是Hbase中分布式存储和负载均衡的最小单元，即不同的HRegion可以分别在不同的HRegionServer上，但**同一个HRegion是不会拆分到多个HRegionServer上的**。HRegion按大小分割，每个表一般只有一个HRegion，随着数据不断插入表，HRegion不断增大，**当HRegion的某个列簇达到一个阀值（默认256M）时就会分成两个新的HRegion。**

每一个HRegion由一个或多个Store组成，至少是一个Store，**Hbase会把一起访问的数据放在一个Store里面，即为每个ColumnFamily建一个Store，**如果有几个ColumnFamily，也就有几个Store。一个Store由一个MemStore和0或者多个StoreFile组成。 **Hbase以Store的大小来判断是否需要切分HRegion。**

**MemStore 是放在内存里的，保存修改的数据即keyValues。**当MemStore的大小达到一个阀值时，MemStore会被Flush到文件，即生成一个快照。目前**Hbase会有一个线程来负责MemStore的Flush操作。**

首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。 Hbase中KeyValue数据的存储格式，Trailer中有指针指向其他数据块的起始点，FileInfo记录了文件的一些meta信息。Data Block是Hbase IO的基本单元，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，**当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。**

在Hbase中写入数据时，会将数据写入内存同时写wal日志,为防止日志丢失，**日志是写在hdfs上的**。

<https://blog.csdn.net/qq_18298439/article/details/88592861>

通过zookeeper定位写到哪一台HRegionServer.来了数据先写到hlog中，然后再写到内存中memstore.

### 写流程

1. client向hregionserver发送写请求。
2. hregionserver将数据写到hlog（write ahead log）。为了数据的持久化和恢复。
3. hregionserver将数据写到内存（memstore）
4. 反馈client写成功。

### 数据flush过程

1. 当memstore数据达到阈值（默认是64M），将数据刷到硬盘，将内存中的数据删除，同时删除Hlog中的历史数据。
2. 并将数据存储到hdfs中。
3. 在hlog中做标记点。

### 数据合并过程

1. **当数据块达到4块，hmaster将数据块加载到本地，进行合并**
2. 当合并的数据超过256M，进行拆分，将拆分后的region分配给不同的hregionserver管理
3. 当hregionser宕机后，将hregionserver上的hlog拆分，然后分配给不同的hregionserver加载，修改.META.
4. 注意：hlog会同步到hdfs

### hbase的读流程

1. 通过zookeeper和-ROOT- .META.表定位hregionserver。
2. 数据从内存和硬盘合并后返回给client
3. 数据块会缓存

### hmaster的职责

1、管理用户对Table的增、删、改、查操作；

2、记录region在哪台Hregion server上

3、在Region Split后，负责新Region的分配；

4、新机器加入时，管理HRegion Server的负载均衡，调整Region分布

5、在HRegion Server宕机后，负责失效HRegion Server 上的Regions迁移。

### hregionserver的职责

HRegion Server主要负责响应用户I/O请求，向HDFS文件系统中读写数据，是HBASE中最核心的模块。

HRegion Server管理了很多table的分区，也就是region。

### client职责

Client

HBASE Client使用HBASE的RPC机制与HMaster和RegionServer进行通信

管理类操作：Client与HMaster进行RPC；

数据读写类操作：Client与HRegionServer进行RPC。

# MapReduce操作Hbase

## 实现方法

Hbase对MapReduce提供支持，它实现了TableMapper类和TableReducer类，我们只需要继承这两个类即可。

1、写个mapper继承TableMapper<Text, IntWritable>

参数：Text：mapper的输出key类型； IntWritable：mapper的输出value类型。

其中的map方法如下：

map(ImmutableBytesWritable key, Result value,Context context)

参数：key：rowKey；value： Result ，一行数据； context上下文

2、写个reduce继承TableReducer<Text, IntWritable, ImmutableBytesWritable>

参数：Text:reducer的输入key； IntWritable：reduce的输入value；

ImmutableBytesWritable：reduce输出到hbase中的rowKey类型。

其中的reduce方法如下：

reduce(Text key, Iterable<IntWritable> values,Context context)

参数： key：reduce的输入key；values：reduce的输入value；

## 准备表

1、建立数据来源表‘word’，包含一个列族‘content’

向表中添加数据，在列族中放入列‘info’，并将短文数据放入该列中，如此插入多行，行键为不同的数据即可

2、建立输出表‘stat’，包含一个列族‘content’

3、通过Mr操作Hbase的‘word’表，对‘content：info’中的短文做词频统计，并将统计结果写入‘stat’表的‘content：info中’，行键为单词

## 实现

|  |
| --- |
| package com.itcast.hbase;  import java.io.IOException;  import java.util.ArrayList;  import java.util.List;  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.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.Scan;  import org.apache.hadoop.hbase.io.ImmutableBytesWritable;  import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;  import org.apache.hadoop.hbase.mapreduce.TableMapper;  import org.apache.hadoop.hbase.mapreduce.TableReducer;  import org.apache.hadoop.hbase.util.Bytes;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  /\*\*  \* mapreduce操作hbase  \* @author wilson  \*  \*/  public class HBaseMr {  /\*\*  \* 创建hbase配置  \*/  static Configuration config = null;  static {  config = HBaseConfiguration.create();  config.set("hbase.zookeeper.quorum", "slave1,slave2,slave3");  config.set("hbase.zookeeper.property.clientPort", "2181");  }  /\*\*  \* 表信息  \*/  public static final String tableName = "word";//表名1  public static final String colf = "content";//列族  public static final String col = "info";//列  public static final String tableName2 = "stat";//表名2  /\*\*  \* 初始化表结构，及其数据  \*/  public static void initTB() {  HTable table=null;  HBaseAdmin admin=null;  try {  admin = new HBaseAdmin(config);//创建表管理  /\*删除表\*/  if (admin.tableExists(tableName)||admin.tableExists(tableName2)) {  System.out.println("table is already exists!");  admin.disableTable(tableName);  admin.deleteTable(tableName);  admin.disableTable(tableName2);  admin.deleteTable(tableName2);  }  /\*创建表\*/  HTableDescriptor desc = new HTableDescriptor(tableName);  HColumnDescriptor family = new HColumnDescriptor(colf);  desc.addFamily(family);  admin.createTable(desc);  HTableDescriptor desc2 = new HTableDescriptor(tableName2);  HColumnDescriptor family2 = new HColumnDescriptor(colf);  desc2.addFamily(family2);  admin.createTable(desc2);  /\*插入数据\*/  table = new HTable(config,tableName);  table.setAutoFlush(false);  table.setWriteBufferSize(5);  List<Put> lp = new ArrayList<Put>();  Put p1 = new Put(Bytes.toBytes("1"));  p1.add(colf.getBytes(), col.getBytes(), ("The Apache Hadoop software library is a framework").getBytes());  lp.add(p1);  Put p2 = new Put(Bytes.toBytes("2"));p2.add(colf.getBytes(),col.getBytes(),("The common utilities that support the other Hadoop modules").getBytes());  lp.add(p2);  Put p3 = new Put(Bytes.toBytes("3"));  p3.add(colf.getBytes(), col.getBytes(),("Hadoop by reading the documentation").getBytes());  lp.add(p3);  Put p4 = new Put(Bytes.toBytes("4"));  p4.add(colf.getBytes(), col.getBytes(),("Hadoop from the release page").getBytes());  lp.add(p4);  Put p5 = new Put(Bytes.toBytes("5"));  p5.add(colf.getBytes(), col.getBytes(),("Hadoop on the mailing list").getBytes());  lp.add(p5);  table.put(lp);  table.flushCommits();  lp.clear();  } catch (Exception e) {  e.printStackTrace();  } finally {  try {  if(table!=null){  table.close();  }  } catch (IOException e) {  e.printStackTrace();  }  }  }  /\*\*  \* MyMapper 继承 TableMapper  \* TableMapper<Text,IntWritable>  \* Text:输出的key类型，  \* IntWritable：输出的value类型  \*/  public static class MyMapper extends TableMapper<Text, IntWritable> {  private static IntWritable one = new IntWritable(1);  private static Text word = new Text();  @Override  //输入的类型为：key：rowKey； value：一行数据的结果集Result  protected void map(ImmutableBytesWritable key, Result value,  Context context) throws IOException, InterruptedException {  //获取一行数据中的colf：col  String words = Bytes.toString(value.getValue(Bytes.toBytes(colf), Bytes.toBytes(col)));// 表里面只有一个列族，所以我就直接获取每一行的值  //按空格分割  String itr[] = words.toString().split(" ");  //循环输出word和1  for (int i = 0; i < itr.length; i++) {  word.set(itr[i]);  context.write(word, one);  }  }  }  /\*\*  \* MyReducer 继承 TableReducer  \* TableReducer<Text,IntWritable>  \* Text:输入的key类型，  \* IntWritable：输入的value类型，  \* ImmutableBytesWritable：输出类型，表示rowkey的类型  \*/  public static class MyReducer extends  TableReducer<Text, IntWritable, ImmutableBytesWritable> {  @Override  protected void reduce(Text key, Iterable<IntWritable> values,  Context context) throws IOException, InterruptedException {  //对mapper的数据求和  int sum = 0;  for (IntWritable val : values) {//叠加  sum += val.get();  }  // 创建put，设置rowkey为单词  Put put = new Put(Bytes.toBytes(key.toString()));  // 封装数据  put.add(Bytes.toBytes(colf), Bytes.toBytes(col),Bytes.toBytes(String.valueOf(sum)));  //写到hbase,需要指定rowkey、put  context.write(new ImmutableBytesWritable(Bytes.toBytes(key.toString())),put);  }  }    public static void main(String[] args) throws IOException,  ClassNotFoundException, InterruptedException {  config.set("df.default.name", "hdfs://master:9000/");//设置hdfs的默认路径  config.set("hadoop.job.ugi", "hadoop,hadoop");//用户名，组  config.set("mapred.job.tracker", "master:9001");//设置jobtracker在哪  //初始化表  initTB();//初始化表  //创建job  Job job = new Job(config, "HBaseMr");//job  job.setJarByClass(HBaseMr.class);//主类  //创建scan  Scan scan = new Scan();  //可以指定查询某一列  scan.addColumn(Bytes.toBytes(colf), Bytes.toBytes(col));  //创建查询hbase的mapper，设置表名、scan、mapper类、mapper的输出key、mapper的输出value  TableMapReduceUtil.initTableMapperJob(tableName, scan, MyMapper.class,Text.class, IntWritable.class, job);  //创建写入hbase的reducer，指定表名、reducer类、job  TableMapReduceUtil.initTableReducerJob(tableName2, MyReducer.class, job);  System.exit(job.waitForCompletion(true) ? 0 : 1);  }  } |

**3.0后新的API:**

***Admin admin = config.getAdmin()***

***TableDescriptorBuilder test\_teacher\_info = TableDescriptorBuilder.newBuilder(TableName.valueOf("Test\_teacher\_info"));***

***ColumnFamilyDescriptor of = ColumnFamilyDescriptorBuilder.of("base\_info");  
  
test\_teacher\_info.setColumnFamily(of);  
//创建列族 2  
ColumnFamilyDescriptor of1 = ColumnFamilyDescriptorBuilder.of("emp\_info");  
test\_teacher\_info.setColumnFamily(of1);  
//构建  
TableDescriptor build = test\_teacher\_info.build();***

***admin.createTable(build );***

conn = getConnection();

Table table = conn.getTable(TableName.valueOf(tableName));

Put put1 = new Put(Bytes.toBytes(rowKey));

*// 获取表中的所有列族 client version 2.0.1*

ColumnFamilyDescriptor[] columnFamilyDescriptors = table.getDescriptor().getColumnFamilies();

Put1.addColumn(Bytes.toBytes(familyName),

*// 列*

Bytes.toBytes(columns[i]),

*// 列的值*

Bytes.toBytes(values[i]));

table.put(put1);

**如何启动hmaster/regionserver:**

在任意的安装了hbase的机器上启动hmaster

命令：

hbase-deamon.sh start master

hbase-daemon.sh start regionserver

HBase 性能优化



# 修改Linux最大文件数

Linux系统最大可打开文件数一般默认的参数值是1024，如果你不进行修改并发量上来的时候会出现“Too Many Open Files”的错误，导致整个HBase不可运行

查看： ulimit -a 结果：open files (-n) 1024

临时修改： ulimit -n 4096

持久修改：

vi /etc/security/limits.conf在文件最后加上：

\* soft nofile 65535

\* hard nofile 65535

\* soft nproc 65535

\* hard nproc 65535

# 修改 JVM 配置

修改hbase-env.sh文件中的配置参数  
HBASE\_HEAPSIZE 4000 #HBase使用的 JVM 堆的大小  
HBASE\_OPTS "‐server ‐XX:+UseConcMarkSweepGC"JVM #GC 选项

参数解释：

**-client，-server**

这两个参数用于设置虚拟机使用何种运行模式，client模式启动比较快，但运行时性能和内存管理效率不如server模式，通常用于客户端应用程序。相反，server模式启动比client慢，但可获得更高的运行性能。

‐XX:+UseConcMarkSweepGC：设置为并发收集

# 修改HBase配置：hbase-site.xml

## zookeeper.session.timeout

默认值：3分钟（180000ms）,可以改成1分钟  
说明：RegionServer与Zookeeper间的连接超时时间。当超时时间到后，ReigonServer会被Zookeeper从RS集群清单中移除，HMaster收到移除通知后，会对这台server负责的regions重新balance，让其他存活的RegionServer接管.

调优：  
这个timeout决定了RegionServer是否能够及时的failover。设置成1分钟或更低，可以减少因等待超时而被延长的failover时间。  
不过需要注意的是，对于一些Online应用，RegionServer从宕机到恢复时间本身就很短的（网络闪断，crash等故障，运维可快速介入），如果调低timeout时间，反而会得不偿失。因为当ReigonServer被正式从RS集群中移除时，HMaster就开始做balance了（让其他RS根据故障机器记录的WAL日志进行恢复）。当故障的RS在人工介入恢复后，这个balance动作是毫无意义的，反而会使负载不均匀，给RS带来更多负担。特别是那些固定分配regions的场景。

## hbase.regionserver.handler.count

默认值：10  
说明：RegionServer的请求处理IO线程数。  
调优：  
这个参数的调优与内存息息相关。  
较少的IO线程，适用于处理单次请求内存消耗较高的Big PUT场景（大容量单次PUT或设置了较大cache的scan，均属于Big PUT）或ReigonServer的内存比较紧张的场景。  
较多的IO线程，适用于单次请求内存消耗低，TPS（吞吐量）要求非常高的场景。

## hbase.hregion.max.filesize

默认值：256M  
说明：在当前ReigonServer上单个Reigon的最大存储空间，单个Region超过该值时，这个Region会被自动split成更小的region。  
调优：  
小region对split和compaction友好，因为拆分region或compact小region里的storefile速度很快，内存占用低。缺点是split和compaction会很频繁。  
特别是数量较多的小region不停地split, compaction，会导致集群响应时间波动很大，region数量太多不仅给管理上带来麻烦，甚至会引发一些Hbase的bug。  
一般512以下的都算小region。  
大region，则不会经常split和compaction，因为做一次compact和split会产生较长时间的停顿，对应用的读写性能冲击非常大。

## hfile.block.cache.size

默认值：0.2  
说明：storefile的读缓存占用内存的大小百分比，0.2表示20%。该值直接影响数据读的性能。  
调优：当然是越大越好，如果写比读少很多，开到0.4-0.5也没问题。如果读写较均衡，0.3左右。如果写比读多，果断默认吧。  
HBase上Regionserver的内存分为两个部分，一部分作为Memstore，主要用来写；另外一部分作为BlockCache，主要用于读。  
写请求会先写入Memstore，Regionserver会给每个region提供一个Memstore，当Memstore满64MB以后，会启动 flush刷新到磁盘。  
读请求先到Memstore中查数据，查不到就到BlockCache中查，再查不到就会到磁盘上读，并把读的结果放入BlockCache。由于BlockCache采用的是LRU策略（Least Recently Used 近期最少使用算法），因此BlockCache达到上限(heapsize \* hfile.block.cache.size \* 0.85)后，会启动淘汰机制，淘汰掉最老的一批数据。  
一个Regionserver上有一个BlockCache和N个Memstore，它们的大小之和不能大于等于内存 \* 0.8，否则HBase不能启动。默认BlockCache为0.2，而Memstore为0.4。对于注重读响应时间的系统，可以将 BlockCache设大些，比如设置BlockCache=0.4，Memstore=0.39，以加大缓存的命中率。

## hbase.hregion.memstore.block.multiplier

默认值：2  
说明：当一个region里的memstore占用内存大小超过hbase.hregion.memstore.flush.size两倍的大小时，block该region的所有请求，进行flush，释放内存。  
虽然我们设置了region所占用的memstores总内存大小，比如64M，但想象一下，在最后63.9M的时候，我Put了一个200M的数据，此时memstore的大小会瞬间暴涨到超过预期的hbase.hregion.memstore.flush.size的几倍。这个参数的作用是当memstore的大小增至超过hbase.hregion.memstore.flush.size 2倍时，block所有请求，遏制风险进一步扩大。  
调优： 这个参数的默认值还是比较靠谱的。如果你预估你的正常应用场景（不包括异常）不会出现突发写或写的量可控，那么保持默认值即可。