# HBase教案

# HBase基本介绍

**简介**

hbase是bigtable的开源java版本。是建立在hdfs之上，提供高可靠性、高性能、列存储、可伸缩、实时读写nosql的数据库系统。

它介于nosql和RDBMS之间，仅能通过主键(row key)和主键的range来检索数据，仅支持单行事务(可通过hive支持来实现多表join等复杂操作)。

主要用来存储结构化和半结构化的松散数据。

Hbase查询数据功能很简单，不支持join等复杂操作，不支持复杂的事务（行级的事务）

Hbase中支持的数据类型：byte[]

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中的表一般有这样的特点：

* 大：一个表可以有上十亿行，上百万列
* 面向列:面向列(族)的存储和权限控制，列(族)独立检索。
* 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。

HBase的发展历程

HBase的原型是Google的BigTable论文，受到了该论文思想的启发，目前作为Hadoop的子项目来开发维护，用于支持结构化的数据存储。

官方网站：<http://hbase.apache.org>

\* 2006年Google发表BigTable白皮书

\* 2006年开始开发HBase

\* 2008 HBase成为了 Hadoop的子项目

\* 2010年HBase成为Apache顶级项目

# 2、HBase与Hadoop的关系

## 1、HDFS

\* 为分布式存储提供文件系统

\* 针对存储大尺寸的文件进行优化，不需要对HDFS上的文件进行随机读写

\* 直接使用文件

\* 数据模型不灵活

\* 使用文件系统和处理框架

\* 优化一次写入，多次读取的方式

## 2、HBase

\* 提供表状的面向列的数据存储

\* 针对表状数据的随机读写进行优化

\* 使用key-value操作数据

\* 提供灵活的数据模型

\* 使用表状存储，支持MapReduce，依赖HDFS

\* 优化了多次读，以及多次写

版本对应关系参考官网说明：<http://hbase.apache.org/book.html#hadoop>

# 3、RDBMS与HBase的对比

## 1、关系型数据库

**结构：**

\* 数据库以表的形式存在

\* 支持FAT、NTFS、EXT、文件系统

\* 使用Commit log存储日志

\* 参考系统是坐标系统

\* 使用主键（PK）

\* 支持分区

\* 使用行、列、单元格

**功能：**

\* 支持向上扩展

\* 使用SQL查询

\* 面向行，即每一行都是一个连续单元

\* 数据总量依赖于服务器配置

\* 具有ACID支持

\* 适合结构化数据

\* 传统关系型数据库一般都是中心化的

\* 支持事务

\* 支持Join

## 2、HBase

**结构：**

\* 数据库以region的形式存在

\* 支持HDFS文件系统

\* 使用WAL（Write-Ahead Logs）存储日志

\* 参考系统是Zookeeper

\* 使用行键（row key）

\* 支持分片

\* 使用行、列、列族和单元格

**功能：**

\* 支持向外扩展

\* 使用API和MapReduce来访问HBase表数据

\* 面向列，即每一列都是一个连续的单元

\* 数据总量不依赖具体某台机器，而取决于机器数量

\* HBase不支持ACID（Atomicity、Consistency、Isolation、Durability）

\* 适合结构化数据和非结构化数据

\* 一般都是分布式的

\* HBase不支持事务，支持的是单行数据的事务操作

\* 不支持Join

# 4、HBase特征简要

**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。

备注：RegionServer的作用是管理region、承接业务的访问，这个后面会详细的介绍通过横向添加Datanode的机器，进行存储层扩容，提升Hbase的数据存储能力和提升后端存储的读写能力。

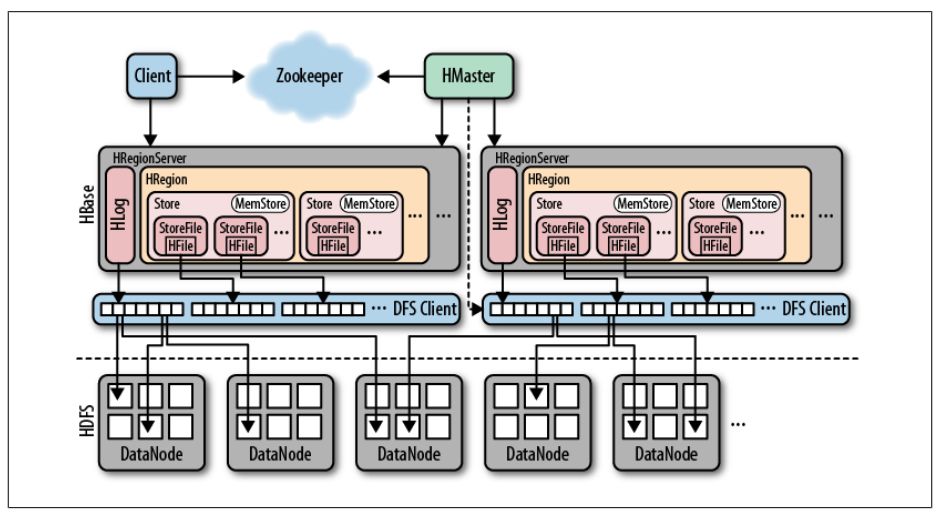
**4）高并发**

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。

# 5、HBase的基础架构



## 1、HMaster

**功能：**

1) 监控RegionServer

2) 处理RegionServer故障转移

3) 处理元数据的变更

4) 处理region的分配或移除

5) 在空闲时间进行数据的负载均衡

6) 通过Zookeeper发布自己的位置给客户端

## 2、RegionServer

**功能：**

1) 负责存储HBase的实际数据

2) 处理分配给它的Region

3) 刷新缓存到HDFS

4) 维护HLog

5) 执行压缩

6) 负责处理Region分片

**组件：**

**1) Write-Ahead logs**

HBase的修改记录，当对HBase读写数据的时候，数据不是直接写进磁盘，它会在内存中保留一段时间（时间以及数据量阈值可以设定）。但把数据保存在内存中可能有更高的概率引起数据丢失，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，然后再写入内存中。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。

**2) HFile**

这是在磁盘上保存原始数据的实际的物理文件，是实际的存储文件。

**3) Store**

HFile存储在Store中，一个Store对应HBase表中的一个列族。

**4) MemStore**

顾名思义，就是内存存储，位于内存中，用来保存当前的数据操作，所以当数据保存在WAL中之后，RegsionServer会在内存中存储键值对。

**5) Region**

Hbase表的分片，HBase表会根据RowKey值被切分成不同的region存储在RegionServer中，在一个RegionServer中可以有多个不同的region。

# 6、HBase的集群环境搭建

注意事项：Hbase强依赖于HDFS以及zookeeper，所以安装Hbase之前一定要保证Hadoop和zookeeper正常启动

## 第一步：下载对应的HBase的安装包

下载Hbase的安装包，下载地址如下：

<http://archive.apache.org/dist/hbase/2.0.0/hbase-2.0.0-bin.tar.gz>

## 第二步：压缩包上传并解压

将我们的压缩包上传到node01服务器的/export/softwares路径下并解压

cd /export/softwares/

tar -zxf hbase-2.0.0-bin.tar.gz -C /export/servers/

## 第三步：修改配置文件

node01机器进行修改配置文件

cd /export/servers/hbase-2.0.0/conf

### 修改第一个配置文件hbase-env.sh

node01机器进行修改配置文件

注释掉HBase使用内部zk

cd /export/servers/hbase-2.0.0/conf

vim hbase-env.sh

export JAVA\_HOME=/export/servers/jdk1.8.0\_141

export HBASE\_MANAGES\_ZK=false

### 修改第二个配置文件hbase-site.xml

node01机器进行修改配置文件

修改hbase-site.xml

cd /export/servers/hbase-2.0.0/conf

vim hbase-site.xml

<configuration>

<property>

<name>hbase.rootdir</name>

<value>hdfs://node01:8020/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>node01:2181,node02:2181,node03:2181</value>

</property>

<property>

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

<value>/export/servers/zookeeper-3.4.9/zkdatas</value>

</property>

</configuration>

### 修改第三个配置文件regionservers

node01机器进行修改配置文件

cd /export/servers/hbase-2.0.0/conf

vim regionservers

node01

node02

node03

### 创建back-masters配置文件，实现HMaster的高可用

node01机器进行修改配置文件

cd /export/servers/hbase-2.0.0/conf

vim backup-masters

node02

## 第四步：安装包分发到其他机器

将我们node01服务器的hbase的安装包拷贝到其他机器上面去

cd /export/servers/

scp -r hbase-2.0.0/ node02:$PWD

scp -r hbase-2.0.0/ node03:$PWD

## 第五步：三台机器创建软连接

因为hbase需要读取hadoop的core-site.xml以及hdfs-site.xml当中的配置文件信息，所以我们三台机器都要执行以下命令创建软连接

ln -s /export/servers/hadoop-2.7.5/etc/hadoop/core-site.xml /export/servers/hbase-2.0.0/conf/core-site.xml

ln -s /export/servers/hadoop-2.7.5/etc/hadoop/hdfs-site.xml /export/servers/hbase-2.0.0/conf/hdfs-site.xml

## 第六步：三台机器添加HBASE\_HOME的环境变量

三台机器执行以下命令，添加HBASE\_HOME环境变量

vim /etc/profile

export HBASE\_HOME=/export/servers/hbase-2.0.0

export PATH=:$HBASE\_HOME/bin:$PATH

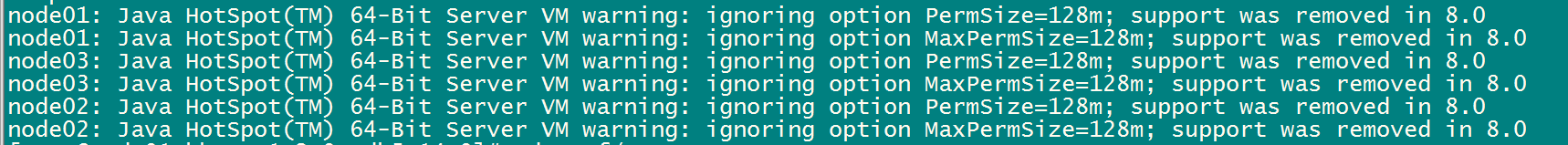
## 第七步：HBase集群启动

第一台机器执行以下命令进行启动

cd /export/servers/hbase-2.0.0

bin/start-hbase.sh

警告提示：HBase启动的时候会产生一个警告，这是因为jdk7与jdk8的问题导致的，如果linux服务器安装jdk8就会产生这样的一个警告



我们可以只是掉所有机器的hbase-env.sh当中的

“HBASE\_MASTER\_OPTS”和“HBASE\_REGIONSERVER\_OPTS”配置 来解决这个问题。不过警告不影响我们正常运行，可以不用解决

另外一种启动方式：

我们也可以执行以下命令单节点进行启动

启动HMaster命令

bin/hbase-daemon.sh start master

启动HRegionServer命令

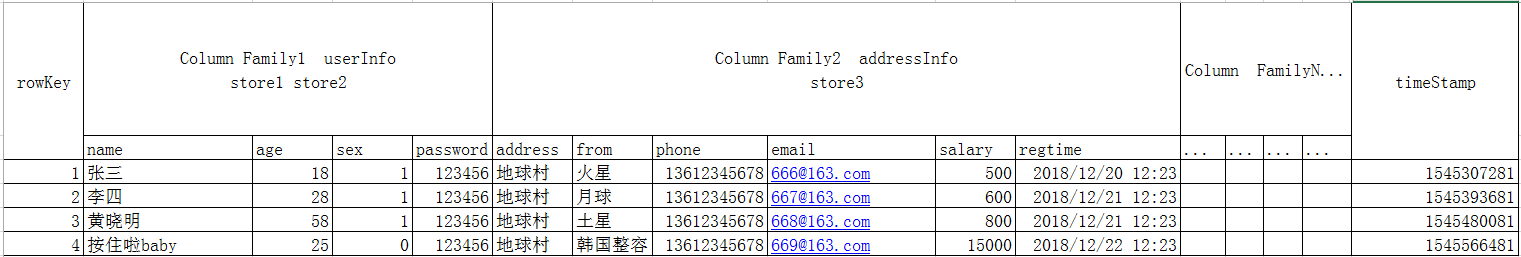
bin/hbase-daemon.sh start regionserver

## 第八步：页面访问

浏览器页面访问

<http://node01:16010/master-status>

## HBASE的表模型基本介绍



# 7、HBase常用shell操作

## 1、进入HBase客户端命令操作界面

node01服务器执行以下命令进入hbase的shell客户端

cd /export/servers/hbase-2.0.0

bin/hbase shell

## 2、查看帮助命令

hbase(main):001:0> help

## 3、查看当前数据库中有哪些表

hbase(main):002:0> list

## 4、创建一张表

创建user表，包含info、data两个列族

hbase(main):010:0> create 'user', 'info', 'data'

或者

hbase(main):010:0> create 'user', {NAME => 'info', VERSIONS => '3'}，{NAME => 'data'}

## 5、添加数据操作

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

hbase(main):011:0> put 'user', 'rk0001', 'info:name', 'zhangsan'

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

hbase(main):012:0> put 'user', 'rk0001', 'info:gender', 'female'

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

hbase(main):013:0> put 'user', 'rk0001', 'info:age', 20

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

hbase(main):014:0> put 'user', 'rk0001', 'data:pic', 'picture'

## 6、查询数据操作

### 1、通过rowkey进行查询

获取user表中row key为rk0001的所有信息

hbase(main):015:0> get 'user', 'rk0001'

### 2、查看rowkey下面的某个列族的信息

获取user表中row key为rk0001，info列族的所有信息

hbase(main):016:0> get 'user', 'rk0001', 'info'

### 3、查看rowkey指定列族指定字段的值

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

hbase(main):017:0> get 'user', 'rk0001', 'info:name', 'info:age'

### 4、查看rowkey指定多个列族的信息

获取user表中row key为rk0001，info、data列族的信息

hbase(main):018:0> get 'user', 'rk0001', 'info', 'data'

或者你也可以这样写

hbase(main):019:0> get 'user', 'rk0001', {COLUMN => ['info', 'data']}

或者你也可以这样写，也行

hbase(main):020:0> get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}

### 4、指定rowkey与列值查询

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

hbase(main):030:0> get 'user', 'rk0001', {FILTER => "ValueFilter(=, 'binary:zhangsan')"}

### 5、指定rowkey与列值模糊查询

获取user表中row key为rk0001，列标示符中含有a的信息

hbase(main):031:0> get 'user', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}

继续插入一批数据

hbase(main):032:0> put 'user', 'rk0002', 'info:name', 'fanbingbing'

hbase(main):033:0> put 'user', 'rk0002', 'info:gender', 'female'

hbase(main):034:0> put 'user', 'rk0002', 'info:nationality', '中国'

hbase(main):035:0> get 'user', 'rk0002', {FILTER => "ValueFilter(=, 'binary:中国')"}

### 6、查询所有数据

查询user表中的所有信息

scan 'user'

### 7、列族查询

查询user表中列族为info的信息

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

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

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

### 8、多列族查询

查询user表中列族为info和data的信息

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

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

### 9、指定列族与某个列名查询

查询user表中列族为info、列标示符为name的信息

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

### 10、指定列族与列名以及限定版本查询

查询user表中列族为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}

### 11、指定多个列族与按照数据值模糊查询

查询user表中列族为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'))"}

### 12、rowkey的范围值查询

查询user表中列族为info，rk范围是[rk0001, rk0003)的数据

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

### 13、指定rowkey模糊查询

查询user表中row key以rk字符开头的

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

### 14、指定数据范围值查询

查询user表中指定范围的数据

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

## 7、更新数据操作

### 1、更新数据值

更新操作同插入操作一模一样，只不过有数据就更新，没数据就添加

### 2、更新版本号

将user表的f1列族版本号改为5

hbase(main):050:0> alter 'user', NAME => 'info', VERSIONS => 5

## 8、删除数据以及删除表操作

### 1、指定rowkey以及列名进行删除

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

hbase(main):045:0> delete 'user', 'rk0001', 'info:name'

### 2、指定rowkey，列名以及字段值进行删除

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

### 3、删除一个列族

删除一个列族：

alter 'user', NAME => 'info', METHOD => 'delete'

或 alter 'user', 'delete' => 'info'

### 4、清空表数据

hbase(main):017:0> truncate 'user'

### 5、删除表

首先需要先让该表为disable状态，使用命令：

hbase(main):049:0> disable 'user'

然后才能drop这个表，使用命令：

hbase(main):050:0> drop 'user'

(注意：如果直接drop表，会报错：Drop the named table. Table must first be disabled)

## 9、统计一张表有多少行数据

hbase(main):053:0> count 'user'

# 8、HBase的高级shell管理命令

## 1、status

例如：显示服务器状态

hbase(main):058:0> status 'node01'

## 2、whoami

显示HBase当前用户，例如：

hbase> whoami

## 3、list

显示当前所有的表

## 4、count

统计指定表的记录数，例如：

hbase> count 'user'

## 5、describe

展示表结构信息

## 6、exists

检查表是否存在，适用于表量特别多的情况

## 7、is\_enabled、is\_disabled

检查表是否启用或禁用

## 8、alter

该命令可以改变表和列族的模式，例如：

**为当前表增加列族：**

hbase> alter 'user', NAME => 'CF2', VERSIONS => 2

**为当前表删除列族：**

hbase(main):002:0> alter 'user', 'delete' => 'CF2'

## 9、disable/enable

禁用一张表/启用一张表

## 10、drop

删除一张表，记得在删除表之前必须先禁用

## 11、truncate

禁用表-删除表-创建表

# 9、HBase的java代码开发

熟练掌握通过使用java代码实现HBase数据库当中的数据增删改查的操作，特别是各种查询，熟练运用

## 第一步：创建maven工程，导入jar包

<dependencies>

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

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-client</artifactId>

<version>2.0.0</version>

</dependency>

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

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-server</artifactId>

<version>2.0.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.14.3</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

<!-- <verbal>true</verbal>-->

</configuration>

</plugin>

</plugins>

</build>

## 第二步：开发javaAPI操作HBase表数据

### 1、创建表myuser，并且带有两个列族f1,f2

@Test

public void createTable() throws IOException {

//创建配置文件对象，并指定zookeeper的连接地址

Configuration configuration = HBaseConfiguration.create();

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

configuration.set("hbase.zookeeper.quorum", "node01,node02,node03");

//集群配置↓

//configuration.set("hbase.zookeeper.quorum", "101.236.39.141,101.236.46.114,101.236.46.113");

configuration.set("hbase.master", "node01:60000");

Connection connection = ConnectionFactory.createConnection(configuration);

Admin admin = connection.getAdmin();

//通过HTableDescriptor来实现我们表的参数设置，包括表名，列族等等

HTableDescriptor hTableDescriptor = new HTableDescriptor(TableName.valueOf("myuser"));

//添加列族

hTableDescriptor.addFamily(new HColumnDescriptor("f1"));

//添加列族

hTableDescriptor.addFamily(new HColumnDescriptor("f2"));

//创建表

boolean myuser = admin.tableExists(TableName.valueOf("myuser"));

if(!myuser){

admin.createTable(hTableDescriptor);

}

//关闭客户端连接

admin.close();

}

### 2、向表中添加数据

/\*\*

\* 插入数据

\*/

@Test

public void addDatas() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

//获取表

Table myuser = connection.getTable(TableName.valueOf("myuser"));

//创建put对象，并指定rowkey

Put put = new Put("0001".getBytes());

put.addColumn("f1".getBytes(),"id".getBytes(), Bytes.toBytes(1));

put.addColumn("f1".getBytes(),"name".getBytes(), Bytes.toBytes("张三"));

put.addColumn("f1".getBytes(),"age".getBytes(), Bytes.toBytes(18));

put.addColumn("f2".getBytes(),"address".getBytes(), Bytes.toBytes("地球人"));

put.addColumn("f2".getBytes(),"phone".getBytes(), Bytes.toBytes("15874102589"));

//插入数据

myuser.put(put);

//关闭表

myuser.close();

}

### 3、查询数据

#### 初始化一批数据到HBase当中用于查询

@Test

public void insertBatchData() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

//获取表

Table myuser = connection.getTable(TableName.valueOf("myuser"));

//创建put对象，并指定rowkey

Put put = new Put("0002".getBytes());

put.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(1));

put.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("曹操"));

put.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(30));

put.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("沛国谯县"));

put.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("16888888888"));

put.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("helloworld"));

Put put2 = new Put("0003".getBytes());

put2.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(2));

put2.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("刘备"));

put2.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(32));

put2.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put2.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("幽州涿郡涿县"));

put2.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("17888888888"));

put2.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("talk is cheap , show me the code"));

Put put3 = new Put("0004".getBytes());

put3.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(3));

put3.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("孙权"));

put3.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(35));

put3.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put3.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("下邳"));

put3.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("12888888888"));

put3.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("what are you 弄啥嘞！"));

Put put4 = new Put("0005".getBytes());

put4.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(4));

put4.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("诸葛亮"));

put4.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(28));

put4.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put4.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("四川隆中"));

put4.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("14888888888"));

put4.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("出师表你背了嘛"));

Put put5 = new Put("0005".getBytes());

put5.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(5));

put5.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("司马懿"));

put5.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(27));

put5.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put5.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("哪里人有待考究"));

put5.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("15888888888"));

put5.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("跟诸葛亮死掐"));

Put put6 = new Put("0006".getBytes());

put6.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(5));

put6.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("xiaobubu—吕布"));

put6.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(28));

put6.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put6.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("内蒙人"));

put6.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("15788888888"));

put6.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("貂蝉去哪了"));

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

listPut.add(put);

listPut.add(put2);

listPut.add(put3);

listPut.add(put4);

listPut.add(put5);

listPut.add(put6);

myuser.put(listPut);

myuser.close();

}

#### 按照rowkey进行查询获取所有列的所有值

查询主键rowkey为0003的人

/\*\*

\* 查询数据，按照主键id进行查询

\*/

@Test

public void searchData() throws IOException {

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Get get = new Get(Bytes.toBytes("0003"));

Result result = myuser.get(get);

Cell[] cells = result.rawCells();

//获取所有的列名称以及列的值

for (Cell cell : cells) {

//注意，如果列属性是int类型，那么这里就不会显示

System.out.println(Bytes.toString(cell.getQualifierArray(),cell.getQualifierOffset(),cell.getQualifierLength()));

System.out.println(Bytes.toString(cell.getValueArray(),cell.getValueOffset(),cell.getValueLength()));

}

myuser.close();

}

#### 按照rowkey查询指定列族下面的值，或者指定列的值

需求一：查询f1列族下面所有列的值

需求二：查询f1列族下面id列的值

//通过rowKey进行查询

Get get = new Get("0003".getBytes());

//get.addFamily("f1".getBytes());

get.addColumn("f1".getBytes(),"id".getBytes());

#### 通过startRowKey和endRowKey进行扫描

/\*\*

\* 通过startRowKey和endRowKey进行扫描查询

\*/

@Test

public void scanRowKey() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

scan.setStartRow("0004".getBytes());

scan.setStopRow("0006".getBytes());

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//遍历获取得到所有的列族以及所有的列的名称

KeyValue[] raw = result.raw();

for (KeyValue keyValue : raw) {

//获取所属列族

System.out.println(Bytes.toString(keyValue.getFamilyArray(),keyValue.getFamilyOffset(),keyValue.getFamilyLength()));

System.out.println(Bytes.toString(keyValue.getQualifierArray(),keyValue.getQualifierOffset(),keyValue.getQualifierLength()));

}

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

#### 通过scan进行全表扫描

/\*\*

\* 全表扫描

\*/

@Test

public void scanAllData() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

### 4、过滤器查询

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。 |

#### 1、比较过滤器

##### 1、rowKey过滤器RowFilter

通过RowFilter过滤比rowKey 0003小的所有值出来

/\*\*

\* hbase行键过滤器RowFilter

\*/

@Test

public void rowKeyFilter() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

RowFilter rowFilter = new RowFilter(CompareFilter.CompareOp.LESS\_OR\_EQUAL, new BinaryComparator(Bytes.toBytes("0003")));

scan.setFilter(rowFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

##### 2、列族过滤器FamilyFilter

查询比f2列族小的所有的列族内的数据

FamilyFilter familyFilter = new FamilyFilter(CompareFilter.CompareOp.LESS, new SubstringComparator("f2"));

scan.setFilter(familyFilter);

##### 3、列过滤器QualifierFilter

只查询name列的值

QualifierFilter qualifierFilter = new QualifierFilter(CompareFilter.CompareOp.EQUAL, new SubstringComparator("name"));

scan.setFilter(qualifierFilter);

##### 4、列值过滤器ValueFilter

查询所有列当中包含8的数据

ValueFilter valueFilter = new ValueFilter(CompareFilter.CompareOp.EQUAL, new SubstringComparator("8"));

scan.setFilter(valueFilter);

#### 2、专用过滤器

##### 1、单列值过滤器 SingleColumnValueFilter

SingleColumnValueFilter会返回满足条件数据的所有字段

需求：查询name值为 刘备 的数据

SingleColumnValueFilter singleColumnValueFilter = **new** SingleColumnValueFilter(**"f1"**.getBytes(), **"name"**.getBytes(), CompareOperator.***EQUAL***, **"刘备"**.getBytes());  
scan.setFilter(singleColumnValueFilter);

##### 2、列值排除过滤器SingleColumnValueExcludeFilter

与SingleColumnValueFilter相反，会排除掉指定的列，其他的列全部返回

##### 3、rowkey前缀过滤器PrefixFilter

需求：查询以00开头的所有前缀的rowkey

PrefixFilter prefixFilter = new PrefixFilter("00".getBytes());

scan.setFilter(prefixFilter);

##### 4、分页过滤器PageFilter

通过pageFilter实现分页过滤器

@Test

public void pageFilter2() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

int pageNum = 3;

int pageSize = 2;

Scan scan = new Scan();

if (pageNum == 1) {

PageFilter filter = new PageFilter(pageSize);

scan.setStartRow(Bytes.toBytes(""));

scan.setFilter(filter);

scan.setMaxResultSize(pageSize);

ResultScanner scanner = myuser.getScanner(scan);

for (Result result : scanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

// System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

//System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "phone".getBytes())));

}

}else{

String startRowKey ="";

PageFilter filter = new PageFilter((pageNum - 1) \* pageSize + 1 );

scan.setStartRow(startRowKey.getBytes());

scan.setMaxResultSize((pageNum - 1) \* pageSize + 1);

scan.setFilter(filter);

ResultScanner scanner = myuser.getScanner(scan);

for (Result result : scanner) {

byte[] row = result.getRow();

startRowKey = new String(row);

}

Scan scan2 = new Scan();

scan2.setStartRow(startRowKey.getBytes());

scan2.setMaxResultSize(Long.valueOf(pageSize));

PageFilter filter2 = new PageFilter(pageSize);

scan2.setFilter(filter2);

ResultScanner scanner1 = myuser.getScanner(scan2);

for (Result result : scanner1) {

byte[] row = result.getRow();

System.out.println(new String(row));

}

}

myuser.close();

}

#### 3、多过滤器综合查询FilterList

需求：使用SingleColumnValueFilter查询f1列族，name为刘备的数据，并且同时满足rowkey的前缀以00开头的数据（PrefixFilter）

FilterList filterList = new FilterList();

SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("f1".getBytes(), "name".getBytes(), CompareFilter.CompareOp.EQUAL, "刘备".getBytes());

PrefixFilter prefixFilter = new PrefixFilter("00".getBytes());

filterList.addFilter(singleColumnValueFilter);

filterList.addFilter(prefixFilter);

scan.setFilter(filterList);

### 5、根据rowkey删除数据

/\*\*

\* 删除数据

\*/

@Test

public void deleteByRowKey() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Delete delete = new Delete("0001".getBytes());

myuser.delete(delete);

myuser.close();

}

### 6、删除表操作

@Test

public void deleteTable() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Admin admin = connection.getAdmin();

admin.disableTable(TableName.valueOf("myuser"));

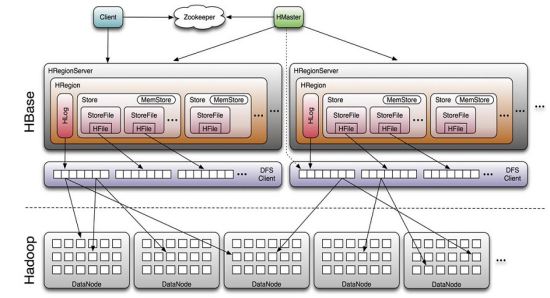
admin.deleteTable(TableName.valueOf("myuser"));

admin.close();

}

# 10、HBase底层原理

## 系统架构



Client

1 包含访问hbase的接口，client维护着一些cache来加快对hbase的访问，比如regione的位置信息。

Zookeeper

1 保证任何时候，集群中只有一个master

2 存贮所有Region的寻址入口

3 实时监控Region Server的状态，将Region server的上线和下线信息实时通知给Master

4 存储Hbase的schema,包括有哪些table，每个table有哪些column family

Master职责

1 为Region server分配region

2 负责region server的负载均衡

3 发现失效的region server并重新分配其上的region

4 HDFS上的垃圾文件回收

5 处理schema更新请求

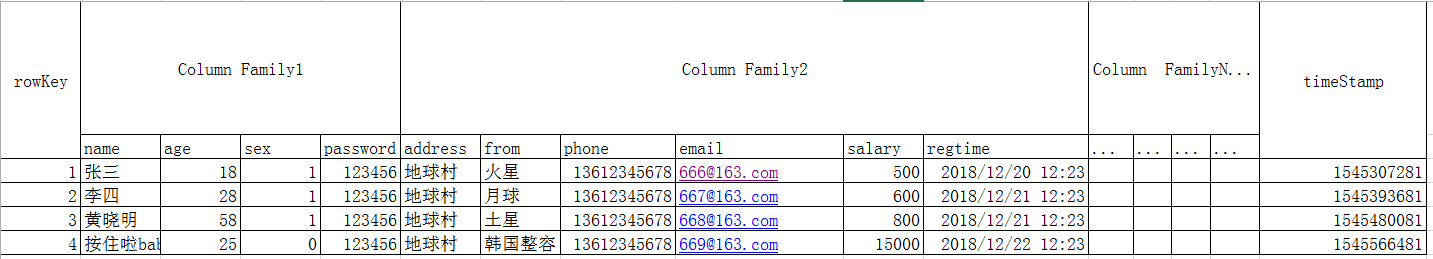
Region Server职责

1 Region server维护Master分配给它的region，处理对这些region的IO请求

2 Region server负责切分在运行过程中变得过大的region

可以看到，client访问hbase上数据的过程并不需要master参与（寻址访问zookeeper和region server，数据读写访问regione server），master仅仅维护者table和region的元数据信息，负载很低。

## HBase的表数据模型



### 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

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。

**Hbase会对表中的数据按照rowkey排序(字典顺序)**

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)

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。

### 列族Column Family

hbase表中的每个列，都归属与某个列族。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。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。

列族越多，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，不要设置太多的列族

### 列 Column

列族下面的具体列，属于某一个ColumnFamily,类似于我们mysql当中创建的具体的列

列是插入数据的时候动态指定的

### 时间戳

HBase中通过row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：

* 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族进行设置。

### Cell

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

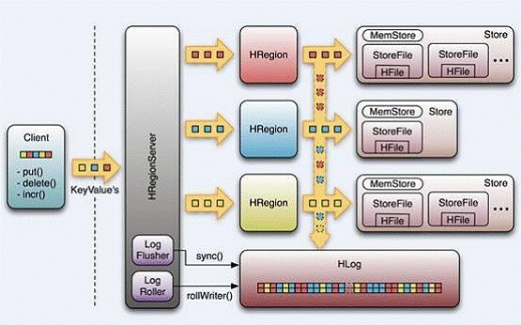
cell中的数据是没有类型的，全部是字节码形式存贮。

### VersionNum

数据的版本号，每条数据可以有多个版本号，默认值为系统时间戳，类型为Long

## 物理存储

### 1、整体结构



1 Table中的所有行都按照row key的字典序排列。

2 Table 在行的方向上分割为多个Hregion。

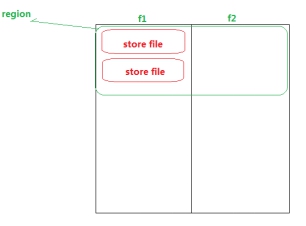
3 region按大小分割的(默认10G)，每个表一开始只有一个region，随着数据不断插入表，region不断增大，当增大到一个阀值的时候，Hregion就会等分会两个新的Hregion。当table中的行不断增多，就会有越来越多的Hregion。

4 Hregion是Hbase中分布式存储和负载均衡的最小单元。最小单元就表示不同的Hregion可以分布在不同的HRegion server上。但一个Hregion是不会拆分到多个server上的。

5 HRegion虽然是负载均衡的最小单元，但并不是物理存储的最小单元。

事实上，HRegion由一个或者多个Store组成，每个store保存一个column family。

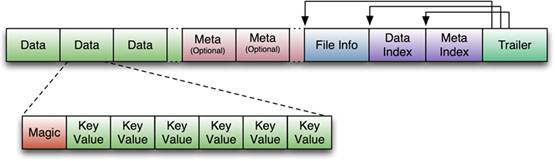
每个Strore又由一个memStore和0至多个StoreFile组成。如上图



### 2、STORE FILE & HFILE结构

StoreFile以HFile格式保存在HDFS上。

附：HFile的格式为：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0080.jpg)

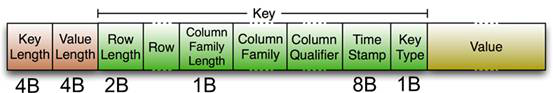
首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数 据块的起始点。

File Info中记录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等。

Data Index和Meta Index块记录了每个Data块和Meta块的起始点。

Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。 每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏。

HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0090.jpg)

开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是 RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。

HFile分为六个部分：

Data Block 段–保存表中的数据，这部分可以被压缩

Meta Block 段 (可选的)–保存用户自定义的kv对，可以被压缩。

File Info 段–Hfile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index 段–Data Block的索引。每条索引的key是被索引的block的第一条记录的key。

Meta Block Index段 (可选的)–Meta Block的索引。

Trailer–这一段是定长的。保存了每一段的偏移量，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。DataBlock Index采用LRU机制淘汰。

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。

目标Hfile的压缩支持两种方式：Gzip，Lzo。

### 3、Memstore与storefile

一个region由多个store组成，每个store包含一个列族的所有数据

Store包括位于内存的memstore和位于硬盘的storefile

写操作先写入memstore,当memstore中的数据量达到某个阈值，Hregionserver启动flashcache进程写入storefile,每次写入形成单独一个storefile

当storefile大小超过一定阈值后，会把当前的region分割成两个，并由Hmaster分配给相应的region服务器，实现负载均衡

客户端检索数据时，先在memstore找，找不到再找storefile

### 4、HLog(WAL log)

WAL 意为Write ahead log(http://en.wikipedia.org/wiki/Write-ahead\_logging)，类似mysql中的binlog,用来 做灾难恢复时用，Hlog记录数据的所有变更,一旦数据修改，就可以从log中进行恢复。

每个Region Server维护一个Hlog,而不是每个Region一个。这样不同region(来自不同table)的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一台region server下线，为了恢复其上的region，需要将region server上的log进行拆分，然后分发到其它region server上进行恢复。

HLog文件就是一个普通的Hadoop Sequence File：

* HLog Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是”写入时间”，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。
* HLog Sequece File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue，可参见上文描述。

## 读写过程

### 1、读请求过程：

HRegionServer保存着meta表以及表数据，要访问表数据，首先Client先去访问zookeeper，从zookeeper里面获取meta表所在的位置信息，即找到这个meta表在哪个HRegionServer上保存着。

接着Client通过刚才获取到的HRegionServer的IP来访问Meta表所在的HRegionServer，从而读取到Meta，进而获取到Meta表中存放的元数据。

Client通过元数据中存储的信息，访问对应的HRegionServer，然后扫描所在HRegionServer的Memstore和Storefile来查询数据。

最后HRegionServer把查询到的数据响应给Client。

查看meta表信息

hbase(main):011:0> scan 'hbase:meta'

### 2、写请求过程：

Client也是先访问zookeeper，找到Meta表，并获取Meta表元数据。

确定当前将要写入的数据所对应的HRegion和HRegionServer服务器。

Client向该HRegionServer服务器发起写入数据请求，然后HRegionServer收到请求并响应。

Client先把数据写入到HLog，以防止数据丢失。

然后将数据写入到Memstore。

如果HLog和Memstore均写入成功，则这条数据写入成功

如果Memstore达到阈值，会把Memstore中的数据flush到Storefile中。

当Storefile越来越多，会触发Compact合并操作，把过多的Storefile合并成一个大的HFile。

当HFile越来越大，Region也会越来越大，达到阈值后，会触发Split操作，将Region一分为二。

细节描述：

hbase使用MemStore和StoreFile存储对表的更新。

数据在更新时首先写入Log(WAL log)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并 且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时，系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了。

当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用Log(WAL log)来恢复checkpoint之后的数据。

StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(minor\_compact, major\_compact),将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行split，等分为两个StoreFile。

由于对表的更新是不断追加的，compact时，需要访问Store中全部的 StoreFile和MemStore，将他们按row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，合并的过程还是比较快。

## Region管理

(1) region分配

任何时刻，一个region只能分配给一个region server。master记录了当前有哪些可用的region server。以及当前哪些region分配给了哪些region server，哪些region还没有分配。当需要分配的新的region，并且有一个region server上有可用空间时，master就给这个region server发送一个装载请求，把region分配给这个region server。region server得到请求后，就开始对此region提供服务。

(2) region server上线

master使用zookeeper来跟踪region server状态。当某个region server启动时，会首先在zookeeper上的server目录下建立代表自己的znode。由于master订阅了server目录上的变更消息，当server目录下的文件出现新增或删除操作时，master可以得到来自zookeeper的实时通知。因此一旦region server上线，master能马上得到消息。

(3) region server下线

当region server下线时，它和zookeeper的会话断开，zookeeper而自动释放代表这台server的文件上的独占锁。master就可以确定：

1 region server和zookeeper之间的网络断开了。

2 region server挂了。

无论哪种情况，region server都无法继续为它的region提供服务了，此时master会删除server目录下代表这台region server的znode数据，并将这台region server的region分配给其它还活着的同志。

## Master工作机制

* master上线

master启动进行以下步骤:

1 从zookeeper上获取唯一一个代表active master的锁，用来阻止其它master成为master。

2 扫描zookeeper上的server父节点，获得当前可用的region server列表。

3 和每个region server通信，获得当前已分配的region和region server的对应关系。

4 扫描.META.region的集合，计算得到当前还未分配的region，将他们放入待分配region列表。

* master下线

由于master只维护表和region的元数据，而不参与表数据IO的过程，master下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的schema，无法进行region的负载均衡，无法处理region 上下线，无法进行region的合并，唯一例外的是region的split可以正常进行，因为只有region server参与)，表的数据读写还可以正常进行。因此master下线短时间内对整个hbase集群没有影响。

从上线过程可以看到，master保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来）

因此，一般hbase集群中总是有一个master在提供服务，还有一个以上的‘master’在等待时机抢占它的位置。

# 11、HBase三个重要机制

## 1、flush机制

1.（hbase.regionserver.global.memstore.size）默认;堆大小的40%

regionServer的全局memstore的大小，超过该大小会触发flush到磁盘的操作,默认是堆大小的40%,而且regionserver级别的flush会阻塞客户端读写

2.（hbase.hregion.memstore.flush.size）默认：128M

单个region里memstore的缓存大小，超过那么整个HRegion就会flush,

3.（hbase.regionserver.optionalcacheflushinterval）默认：1h

内存中的文件在自动刷新之前能够存活的最长时间

4.（hbase.regionserver.global.memstore.size.lower.limit）默认：堆大小 \* 0.4 \* 0.95

有时候集群的“写负载”非常高，写入量一直超过flush的量，这时，我们就希望memstore不要超过一定的安全设置。在这种情况下，写操作就要被阻塞一直到memstore恢复到一个“可管理”的大小, 这个大小就是默认值是堆大小 \* 0.4 \* 0.95，也就是当regionserver级别的flush操作发送后,会阻塞客户端写,一直阻塞到整个regionserver级别的memstore的大小为 堆大小 \* 0.4 \*0.95为止

5.（hbase.hregion.preclose.flush.size）默认为：5M

当一个 region 中的 memstore 的大小大于这个值的时候，我们又触发 了 close.会先运行“pre-flush”操作，清理这个需要关闭的memstore，然后 将这个 region 下线。当一个 region 下线了，我们无法再进行任何写操作。 如果一个 memstore 很大的时候，flush 操作会消耗很多时间。"pre-flush" 操作意味着在 region 下线之前，会先把 memstore 清空。这样在最终执行 close 操作的时候，flush 操作会很快。

6.（hbase.hstore.compactionThreshold）默认：超过3个

一个store里面允许存的hfile的个数，超过这个个数会被写到新的一个hfile里面 也即是每个region的每个列族对应的memstore在fulsh为hfile的时候，默认情况下当超过3个hfile的时候就会 对这些文件进行合并重写为一个新文件，设置个数越大可以减少触发合并的时间，但是每次合并的时间就会越长

## compact机制

把小的storeFile文件合并成大的Storefile文件。

清理过期的数据，包括删除的数据

将数据的版本号保存为3个

## 3、split机制

当Region达到阈值，会把过大的Region一分为二。

默认一个HFile达到10Gb的时候就会进行切分