大数据技术之HBase

# **一、HBaes介绍**

## **1.1、HBase简介**

HBase是一个分布式的、**面向列**的开源数据库，它是一个适合于**非结构化数据存储**的数据库。另一个不同的是HBase基于列的而不是基于行的模式。

大：上亿行、百万列

面向列：面向列（簇）的存储和权限控制，列（簇）独立检索

稀疏：对于为空(null)的列，并不占用存储空间，因此，表的设计的非常的稀疏

## **1.2、HBase的角色**

### **1.2.1、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

### **1.2.2、HRegionServer**

**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表中的一个列簇。

**4) MemStore**

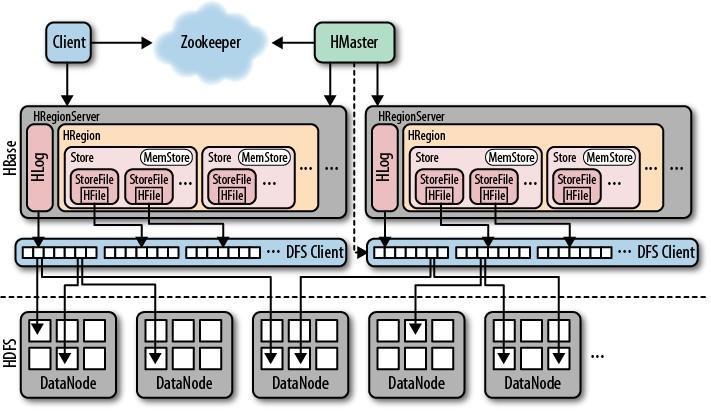
顾名思义，就是内存存储，位于内存中，用来保存当前的数据操作，所以当数据保存在WAL中之后，RegsionServer会在内存中存储键值对。

**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。

## **1.3、HBase的架构**

一个RegionServer可以包含多个HRegion，每个RegionServer维护一个HLog，和多个HFiles以及其对应的MemStore。RegionServer运行于DataNode上，数量可以与DatNode数量一致，请参考如下架构图：



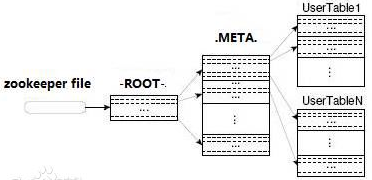
## 1.4 HBase数据模型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Rowkey | timestamp | 列簇（colume family） Store | | |
| 列1(colume) | 列2(colume) | 列3(colume) |
| region | Rowkey1 |  |  |  |  |
| Rowkey2 |  |  |  |  |
| Rowkey3 |  |  |  |  |

确定一个单元格的位置（cell），需要如下四个

rowkey + Colume Family + Colume + timestamp(版本version)，数据有版本的概念，即一个单元格可能有多个值，但是只有最新得一个对外显示。

* HBase中有两张特殊的Table，-ROOT-和.META.
* .META.：记录了用户表的Region信息，**.META.可以有多个region**
* -ROOT-：记录了.META.表的Region信息，**-ROOT-只有一个region**
* Zookeeper中记录了-ROOT-表的location
* Client访问用户数据之前需要首先访问zookeeper，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过client端会做cache缓存，注意：在0.96版本后，Hbase移除了-ROOT-表



**Row Key:** 行键，Table的主键，Table中的记录默认按照Row Key升序排序

**Timestamp:**[时间戳](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E6%88%B3)，每次数据操作对应的时间戳，可以看作是数据的**version** number

**Column Family：**

列簇，Table在水平方向有一个或者多个Column Family组成，一个Column Family中可以由**任意多个Column**组成，即Column Family支持动态扩展，无需预先定义Column的数量以及类型，所有Column均以二进制格式存储，用户需要自行进行类型转换。

**Table & Region:**

当Table随着记录数不断增加而变大后，会逐渐分裂成多份splits，成为regions，一个**region由[startkey,endkey)**表示，不同的region会被Master分配给相应的RegionServer进行管理：.

**HMaster**

HMaster没有单点问题，HBase中可以启动多个HMaster，通过Zookeeper的Master Election机制保证总有一个Master运行，HMaster在功能上主要负责Table和Region的管理工作：

1. 管理用户对Table的增、删、改、查操作

2. 管理HRegionServer的负载均衡，调整Region分布

3. 在Region Split后，负责新Region的分配

4. 在HRegionServer停机后，负责失效HRegionServer 上的Regions迁移

**HRegionServer**

HRegionServer内部管理了一系列HRegion对象，每个HRegion对应了Table中的一个Region，HRegion中由多个HStore组成。**每个HStore对应了Table中的一个Column Family**的存储，可以看出每个Column Family其实就是一个集中的[存储单元](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%8D%95%E5%85%83)，因此最好将具备共同IO特性的column放在一个Column Family中，这样最高效。

**MemStore & StoreFiles**

HStore存储是HBase存储的核心了，其中由两部分组成，**一部分是MemStore**，一部分是**StoreFiles**。MemStore是Sorted Memory Buffer，用户写入的数据**首先**会放入MemStore，当MemStore满了以后会Flush成一个StoreFile（底层实现是HFile），当StoreFile文件数量增长到一定阈值，会触发Compact合并操作，将多个StoreFiles合并成一个StoreFile，合并过程中会进行版本合并和数据删除，因此可以看出HBase其实只有增加数据，所有的更新和删除操作都是在后续的compact过程中进行的，这使得用户的写操作只要进入内存中就可以立即返回，保证了HBase I/O的高性能。当StoreFiles Compact后，会逐步形成越来越大的StoreFile，当单个StoreFile大小超过一定阈值后，会触发Split操作，同时把当前Region Split成2个Region，父Region会下线，新Split出的2个孩子Region会被HMaster分配到相应的HRegionServer上，使得原先1个Region的压力得以分流到2个Region上。

**HLog**

每个HRegionServer中都有一个HLog对象，HLog是一个实现Write Ahead Log的类，在每次用户操作写入MemStore的同时，也会写一份数据到HLog文件中，HLog文件定期会滚动出新的，并删除旧的文件（已持久化到StoreFile中的数据）。当HRegionServer意外终止后，HMaster会通过Zookeeper感知到，HMaster首先会处理遗留的 HLog文件，将其中不同Region的Log数据进行拆分，分别放到相应region的目录下，然后再将失效的region重新分配，领取 到这些region的HRegionServer在Load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成[数据恢复](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E6%81%A2%E5%A4%8D)。

**文件类型**

HBase中的所有数据文件都存储在Hadoop HDFS文件系统上，主要包括上述提出的两种文件类型：

1. HFile， HBase中KeyValue数据的存储格式，HFile是Hadoop的二进制格式文件，实际上StoreFile就是对HFile做了轻量级包装，即StoreFile底层就是HFile
2. HLog File，HBase中WAL（Write Ahead Log） 的存储格式，物理上是Hadoop的Sequence File

Zookeeper中hbase的节点的存储信息：

rs：regionserver节点信息

table-lock：hbase的除meta以外的所有表

Table：hbase的所有的表

# **二、HBase部署与使用**

## **2.1、部署**

### **2.1.1、Zookeeper正常部署**

首先保证Zookeeper集群的正常部署，并启动之：

/opt/module/zookeeper-3.4.5/bin/zkServer.sh start

### **2.1.2、Hadoop正常部署**

Hadoop集群的正常部署并启动：

/opt/module/hadoop-2.8.4/sbin/start-dfs.sh

/opt/module/hadoop-2.8.4/sbin/start-yarn.sh

### **2.1.3、HBase的解压**

解压HBase到指定目录：

tar -zxf /opt/software/hbase-1.3.1-bin.tar.gz -C /opt/module/

**2.1.4、HBase的配置文件**

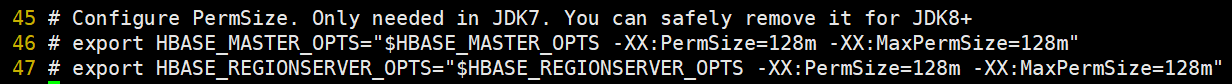
需要修改HBase对应的配置文件。

**hbase-env.sh修改内容：**

export JAVA\_HOME=/opt/module/jdk1.8.0\_121

export HBASE\_MANAGES\_ZK=false

尖叫提示：如果使用的是JDK8以上版本，注释掉hbase-env.sh的45-47行，不然会报警告



**hbase-site.xml修改内容：**

<property>

<name>hbase.rootdir</name>

<value>hdfs://bigdata111:9000/hbase</value>

</property>

<property>

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

<value>true</value>

</property>

<property>

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

<value>16000</value>

</property>

<property>

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

<value>bigdata111:2181,bigdata112:2181,bigdata113:2181</value>

</property>

<property>

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

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

</property>

<property>

<name>hbase.master.maxclockskew</name>

<value>180000</value>

</property>

**regionservers：**

bigdata111

bigdata112

bigdata113

### **2.1.5、HBase需要依赖的Jar包(额外，不用配置)**

由于HBase需要依赖Hadoop，所以替换HBase的lib目录下的jar包，以解决兼容问题：

1. 删除原有的jar：

rm -rf /opt/module/hbase-1.3.1/lib/hadoop-\*

rm -rf /opt/module/hbase-1.3.1/lib/zookeeper-3.4.10.jar

1. 拷贝新jar，涉及的jar有：

hadoop-annotations-2.8.4.jar

hadoop-auth-2.8.4.jar

hadoop-client-2.8.4.jar

hadoop-common-2.8.4.jar

hadoop-hdfs-2.8.4.jar

hadoop-mapreduce-client-app-2.8.4.jar

hadoop-mapreduce-client-common-2.8.4.jar

hadoop-mapreduce-client-core-2.8.4.jar

hadoop-mapreduce-client-hs-2.8.4.jar

hadoop-mapreduce-client-hs-plugins-2.8.4.jar

hadoop-mapreduce-client-jobclient-2.8.4.jar

hadoop-mapreduce-client-jobclient-2.8.4-tests.jar

hadoop-mapreduce-client-shuffle-2.8.4.jar

hadoop-yarn-api-2.8.4.jar

hadoop-yarn-applications-distributedshell-2.8.4.jar

hadoop-yarn-applications-unmanaged-am-launcher-2.8.4.jar

hadoop-yarn-client-2.8.4.jar

hadoop-yarn-common-2.8.4.jar

hadoop-yarn-server-applicationhistoryservice-2.8.4.jar

hadoop-yarn-server-common-2.8.4.jar

hadoop-yarn-server-nodemanager-2.8.4.jar

hadoop-yarn-server-resourcemanager-2.8.4.jar

hadoop-yarn-server-tests-2.8.4.jar

hadoop-yarn-server-web-proxy-2.8.4.jar

zookeeper-3.4.10.jar

尖叫提示：这些jar包的对应版本应替换成你目前使用的hadoop版本，具体情况具体分析。

查找jar包举例：

find /opt/module/hadoop-2.8.4/ -name hadoop-annotations\*

然后将找到的jar包复制到HBase的lib目录下即可。

### **2.1.6、HBase软连接Hadoop配置(额外，不用配置)**

ln -s /opt/module/hadoop-2.8.4/etc/hadoop/core-site.xml /opt/module/hbase-1.3.1/conf/core-site.xml

ln -s /opt/module/hadoop-2.8.4/etc/hadoop/hdfs-site.xml /opt/module/hbase-1.3.1/conf/hdfs-site.xml

### 2.1.7.0 配置环境变量

vi /etc/profile

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

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

source /etc/profile

### 2.1.7、HBase远程scp到其他集群

scp -r /opt/module/hbase-1.3.1/ bigdata112:/opt/module/

scp -r /opt/module/hbase-1.3.1/ bigdata113:/opt/module/

### 2.1.8、HBase服务的启动

**启动方式1：**

bin/hbase-daemon.sh start master

bin/hbase-daemon.sh start regionserver

尖叫提示：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。

**启动方式2：**

bin/start-hbase.sh

对应的停止服务：

bin/stop-hbase.sh

### 2.1.9、查看Hbse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://bigdata111:16010

## 2.2 基本操作

1. **进入HBase客户端命令行**

bin/hbase shell

**2) 查看帮助命令**

hbase(main)> help

**3) 查看当前数据库中有哪些表**

hbase(main)> list

**4) 查看当前数据库中有哪些命名空间**

hbase(main)> list\_namespace

### 2.2.1 表的操作

1. **创建表**

hbase(main)> create 'student','info'

hbase(main)> create 'iparkmerchant\_order','smzf'

hbase(main)> create 'staff','info'

**2) 插入数据到表**

hbase(main) > put 'student2','1001',cf1:name','Thomas'

hbase(main) > put 'student2','1001','cf1:sex','male'

hbase(main) > put 'student2','1001','cf1:age','18'

hbase(main) > put 'student2','1002','cf1:name','Janna'

hbase(main) > put 'student2','1002','cf1:sex','female'

hbase(main) > put 'student2','1002','cf1:age','20'

数据插入后的数据模型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rowkey | timestamp | cf1 | | | |
| name | sex | age |  |
| 1001 |  | Thomas | male | 18 |  |
| 1002 |  | Janna | female | 20 |  |

**3) 扫描查看表数据**

hbase(main) > scan 'student'

hbase(main) > scan 'student',{STARTROW => '1001', STOPROW => '1001'}

hbase(main) > scan 'student',{STARTROW => '1001'}

注：这个是从哪一个rowkey开始扫描

**4) 查看表结构**

hbase(main):012:0> desc 'student'

**5) 更新指定字段的数据**

hbase(main) > put 'student','1001','info:name','Nick'

hbase(main) > put 'student','1001','info:age','100'

hbase(main) > put 'student','1001','info:isNull',''（仅测试空值问题）

**6) 查看“指定行”或“指定列族:列”的数据**

hbase(main) > get 'student','1001'

hbase(main) > get 'student','1001','info:name'

**7) 删除数据**

**删除某rowkey的全部数据：**

hbase(main) > deleteall 'student','1001'

**8) 清空表数据**

hbase(main) > truncate 'student'

尖叫提示：清空表的操作顺序为先disable，然后再truncate。

**9) 删除表**

**首先需要先让该表为disable状态：**

hbase(main) > disable 'student'

**检查这个表是否被禁用**

hbase(main) > is\_enabled 'hbase\_book'

hbase(main) > is\_disabled 'hbase\_book'

**恢复被禁用得表**

enable 'student'

**然后才能drop这个表：**

hbase(main) > drop 'student'

尖叫提示：如果直接drop表，会报错：Drop the named table. Table must first be disabled

ERROR: Table student is enabled. Disable it first.

**10) 统计表数据行数**

hbase(main) > count 'student'

**11) 变更表信息**

将info列族中的数据存放3个版本：

hbase(main) > alter 'student',{NAME=>'info',VERSIONS=>3}

查看student的最新的版本的数据

hbase(main) > get 'student','1001'

查看HBase中的多版本

hbase(main) > get 'student','1001',{COLUMN=>'info:name',VERSIONS=>10}

### 2.2.2 常用Shell操作

**1) satus** 例如：显示服务器状态

hbase> status 'bigdata111'

**2) exists** 检查表是否存在，适用于表量特别多的情况

hbase> exists 'hbase\_book'

**3) is\_enabled/is\_disabled** 检查表是否启用或禁用

hbase> is\_enabled 'hbase\_book'

hbase> is\_disabled 'hbase\_book'

**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 'hbase\_book', NAME => 'CF2', VERSIONS => 2

**为当前表删除列族：**

hbase> alter 'hbase\_book', 'delete' => 'CF2'

**9) disable**禁用一张表

hbase> disable 'hbase\_book'

hbase> drop 'hbase\_book'

**10) delete**

删除一行中一个单元格的值，例如：

hbase> delete 'hbase\_book', 'rowKey', 'CF:C'

**11) truncate**清空表数据，即禁用表-删除表-创建表

hbase> truncate 'hbase\_book'

**12) create**

创建多个列族：

hbase> create 't1', {NAME => 'f1'}, {NAME => 'f2'}, {NAME => 'f3'}

## 2.3、读写流程

#### 2.3.1、HBase读数据流程

Hbase的regionserver的内存Memstore，block cache。Memstore作业主要是写，另一部分主要是读的。block cache用的是LRU，如果block cache达到上限，会启动淘汰机制。

1. HRegionServer保存着.META.的这样一张表以及表数据，要访问表数据，首先Client先去访问zookeeper，从zookeeper里面找到.META.表所在的位置信息，即找到这个.META.表在哪个HRegionServer上保存着。
2. 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。
3. Client通过元数据中存储的信息，访问对应的HRegionServer，然后扫描(scan)所在

HRegionServer的Memstore如果没有，扫描block cache（读数据的缓存），如果还没有去Storefile来查询数据，查到数据之后将数据读到block cache。

4) 最后HRegionServer把查询到的数据响应给Client。

#### 2.3.2、HBase写数据流程

1. Client也是先访问zookeeper，进而找到.META.表，并获取.META.表信息。
2. 确定当前将要写入的数据所对应的RegionServer服务器和Region。
3. Client向该RegionServer服务器发起写入数据请求，然后RegionServer收到请求并响应。
4. Client先把数据写入到HLog，以防止数据丢失。
5. 然后将数据写入到Memstore。
6. 如果Hlog和Memstore均写入成功，则这条数据写入成功。在此过程中，如果Memstore达到阈值，会把Memstore中的数据flush到StoreFile中。
7. 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Storefile。当Storefile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。

尖叫提示：因为内存空间是有限的，所以说溢写过程必定伴随着大量的小文件产生。

## **2.4、JavaAPI**

### **2.4.1 新建Maven Project**

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>

### 2.4.2 编写HBaseAPI

注意，这部分的学习内容，我们先学习使用老版本的API，接着再写出新版本的API调用方式。因为在企业中，有些时候我们需要一些过时的API来提供更好的兼容性。

**1) 首先需要获取Configuration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", "bigdata111");

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

conf.set("zookeeper.znode.parent", "/hbase");

}

**2) 判断表是否存在：**

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);

}

**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！");

}

}

**4) 删除表**

public static void dropTable(String tableName) throws Exception{

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！");

}

}

**5) 向表中插入数据**

public static void addRowData(String tableName, String rowKey, String columnFamily, String column, String value) throws Exception{

//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) 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();

}

**7) 得到所有数据**

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)));

}

}

}

**8) 得到某一行所有数据**

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());

}

}

**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)));

}

}

### 2.4.3 HBaseUtil

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HColumnDescriptor;

import org.apache.hadoop.hbase.HTableDescriptor;

import org.apache.hadoop.hbase.NamespaceDescriptor;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.Admin;

import org.apache.hadoop.hbase.client.Connection;

import org.apache.hadoop.hbase.client.ConnectionFactory;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

import java.text.DecimalFormat;

import java.util.Iterator;

import java.util.TreeSet;

/\*\*

\* @author Andy

\* 1、NameSpace ====> 命名空间

\* 2、createTable ===> 表

\* 3、isTable ====> 判断表是否存在

\* 4、Region、RowKey、分区键

\*/

public class HBaseUtil {

/\*\*

\* 初始化命名空间

\*

\* @param conf 配置对象

\* @param namespace 命名空间的名字

\* @throws Exception

\*/

public static void initNameSpace(Configuration conf, String namespace) throws Exception {

Connection connection = ConnectionFactory.createConnection(conf);

Admin admin = connection.getAdmin();

//命名空间描述器

NamespaceDescriptor nd = NamespaceDescriptor

.create(namespace)

.addConfiguration("AUTHOR", "Andy")

.build();

//通过admin对象来创建命名空间

admin.createNamespace(nd);

System.out.println("已初始化命名空间");

//关闭两个对象

close(admin, connection);

}

/\*\*

\* 关闭admin对象和connection对象

\*

\* @param admin 关闭admin对象

\* @param connection 关闭connection对象

\* @throws IOException IO异常

\*/

private static void close(Admin admin, Connection connection) throws IOException {

if (admin != null) {

admin.close();

}

if (connection != null) {

connection.close();

}

}

/\*\*

\* 创建HBase的表

\* @param conf

\* @param tableName

\* @param regions

\* @param columnFamily

\*/

public static void createTable(Configuration conf, String tableName, int regions, String... columnFamily) throws IOException {

Connection connection = ConnectionFactory.createConnection(conf);

Admin admin = connection.getAdmin();

//判断表

if (isExistTable(conf, tableName)) {

return;

}

//表描述器 HTableDescriptor

HTableDescriptor htd = new HTableDescriptor(TableName.valueOf(tableName));

for (String cf : columnFamily) {

//列描述器 ：HColumnDescriptor

htd.addFamily(new HColumnDescriptor(cf));

}

//htd.addCoprocessor("hbase.CalleeWriteObserver");

//创建表

admin.createTable(htd,genSplitKeys(regions));

System.out.println("已建表");

//关闭对象

close(admin,connection);

}

/\*\*

\* 分区键

\* @param regions region个数

\* @return splitKeys

\*/

private static byte[][] genSplitKeys(int regions) {

//存放分区键的数组

String[] keys = new String[regions];

//格式化分区键的形式 00 01 02

DecimalFormat df = new DecimalFormat("00");

for (int i = 0; i < regions; i++) {

keys[i] = df.format(i) + "";

}

byte[][] splitKeys = new byte[regions][];

//排序 保证你这个分区键是有序得

TreeSet<byte[]> treeSet = new TreeSet<>(Bytes.BYTES\_COMPARATOR);

for (int i = 0; i < regions; i++) {

treeSet.add(Bytes.toBytes(keys[i]));

}

//输出

Iterator<byte[]> iterator = treeSet.iterator();

int index = 0;

while (iterator.hasNext()) {

byte[] next = iterator.next();

splitKeys[index++]= next;

}

return splitKeys;

}

/\*\*

\* 判断表是否存在

\* @param conf 配置 conf

\* @param tableName 表名

\*/

public static boolean isExistTable(Configuration conf, String tableName) throws IOException {

Connection connection = ConnectionFactory.createConnection(conf);

Admin admin = connection.getAdmin();

boolean result = admin.tableExists(TableName.valueOf(tableName));

close(admin, connection);

return result;

}

}

### 2.4.4 PropertiesUtil

import java.io.IOException;

import java.io.InputStream;

import java.util.Properties;

public class PropertiesUtil {

public static Properties properties = null;

static {

//获取配置文件、方便维护

InputStream is = ClassLoader.getSystemResourceAsStream("hbase\_consumer.properties");

properties = new Properties();

try {

properties.load(is);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 获取参数值

\* @param key 名字

\* @return 参数值

\*/

public static String getProperty(String key){

return properties.getProperty(key);

}

}

### 2.4.5 HBaseDAO

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

public class HBaseDAO {

private static String namespace = PropertiesUtil.getProperty("hbase.calllog.namespace");

private static String tableName = PropertiesUtil.getProperty("hbase.calllog.tablename");

private static Integer regions = Integer.valueOf(PropertiesUtil.getProperty("hbase.calllog.regions"));

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

Configuration conf = HBaseConfiguration.create();

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

conf.set("hbase.zookeeper.quorum", "bigdata111");

conf.set("zookeeper.znode.parent", "/hbase");

if (!HBaseUtil.isExistTable(conf, tableName)) {

HBaseUtil.initNameSpace(conf, namespace);

HBaseUtil.createTable(conf, tableName, regions, "f1", "f2");

}

}

**2.5、MapReduce**

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。

## 2.5、官方HBase-MapReduce

**1) 查看HBase的MapReduce任务的所需的依赖**

$ bin/hbase mapredcp

**2) 执行环境变量的导入**

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

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

**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-2.8.4/bin/yarn jar lib/hbase-server-1.3.1.jar rowcounter ns\_ct:calllog



**-- 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，自己建表用\t分割数据**

1001 Apple Red

1002 Pear Yellow

1003 Pineapple Yellow

尖叫提示：上面的这个数据不要从word中直接复制，有格式错误

**(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-2.8.4/bin/hdfs dfs -mkdir /input\_fruit/

$ /opt/module/hadoop-2.8.4/bin/hdfs dfs -put fruit.tsv /input\_fruit/

**(4) 执行MapReduce到HBase的fruit表中**

$ /opt/module/hadoop-2.8.4/bin/yarn jar lib/hbase-server-1.3.1.jar importtsv \

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

hdfs://bigdata11:9000/input\_fruit

**(5) 使用scan命令查看导入后的结果**

hbase(main):001:0> scan 'fruit'

### 2.5.1、HBase2HBase

**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**

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**

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**

package MRToHBase;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Put;

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.mapreduce.Job;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

import java.io.IOException;

public class Fruit2FruitMRRunner extends Configured implements Tool {

@Override

public int run(String[] strings) 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;

}

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

Configuration conf = HBaseConfiguration.create();

int status = ToolRunner.run(conf, new Fruit2FruitMRRunner(), args);

System.exit(status);

}

}

**4) 打包运行任务**

$ /opt/module/hadoop-2.8.4/bin/yarn jar /opt/module/hbase-1.3.1/HBase-1.0-SNAPSHOT.jar MRToHBase.Fruit2FruitMRRunner

尖叫提示：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。

### 2.5.2、HDFS2HBase

**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**

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**

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**

package HDFSToHBase;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.io.ImmutableBytesWritable;

import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

import java.io.IOException;

public class Txt2FruitRunner extends Configured implements Tool {

@Override

public int run(String[] strings) 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://bigdata11:9000/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;

}

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

Configuration conf = HBaseConfiguration.create();

int status = ToolRunner.run(conf, new Txt2FruitRunner(), args);

System.exit(status);

}

}

**4) 打包运行**

/opt/module/hadoop-2.8.4/bin/yarn jar HDFSToHBase.jar HDFSToHBase.ReadFruitFromHDFSMapper

尖叫提示：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。

## 2.6、与Hive的集成

### **2.6.1、HBase与Hive的对比**

|  |  |  |
| --- | --- | --- |
|  | Hive | HBase |
| 特点 | 类SQL 数据仓库 | NoSQL （Key-value） |
| 适用场景 | 离线数据分析和清洗 | 适合在线业务 |
| 延迟 | 延迟高 | 延迟低 |
| 存储位置 | 存储在HDFS | 存储在HDFS |

### **2.6.2、HBase与Hive集成使用**

**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）。记得还有把zookeeper的jar包考入到hive的lib目录下。

#环境变量/etc/profile

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

$ export HIVE\_HOME=/opt/module/apache-hive-1.2.2-bin

#Shell执行

$ 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>bigdata11,bigdata12,bigdata13</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\_table1(

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\_table1");

尖叫提示：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**

尖叫提示：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 '/opt/module/datas/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\_table1 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\_table1"); |

**(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; |

## **2.7、Sqoop集成：MySQL TO HBase**

Sqoop supports additional import targets beyond HDFS and Hive. Sqoop can also import records into a table in HBase.

之前我们已经学习过如何使用Sqoop在Hadoop集群和关系型数据库中进行数据的导入导出工作，接下来我们学习一下利用Sqoop在HBase和RDBMS中进行数据的转储。

**相关参数：**

|  |  |
| --- | --- |
| 参数 | 描述 |
| --column-family <family> | Sets the target column family for the import  设置导入的目标列族。 |
| --hbase-create-table | If specified, create missing HBase tables  是否自动创建不存在的HBase表（这就意味着，不需要手动提前在HBase中先建立表） |
| --hbase-row-key <col> | Specifies which input column to use as the row key.In case, if input table contains composite  key, then <col> must be in the form of a  comma-separated list of composite key  attributes.  mysql中哪一列的值作为HBase的rowkey，如果rowkey是个组合键，则以逗号分隔。（注：避免rowkey的重复） |
| --hbase-table <table-name> | Specifies an HBase table to use as the target instead of HDFS.  指定数据将要导入到HBase中的哪张表中。 |
| --hbase-bulkload | Enables bulk loading.  是否允许bulk形式的导入。 |

**1) 案例**

**目标：**将RDBMS中的数据抽取到HBase中

**分步实现：**

**(1) 配置sqoop-env.sh，添加如下内容：**

|  |
| --- |
| export HBASE\_HOME=/opt/module/hbase-1.3.1 |

**(2) 在Mysql中新建一个数据库db\_library，一张表book**

|  |
| --- |
| CREATE DATABASE db\_library;  CREATE TABLE db\_library.book(  id int(4) PRIMARY KEY NOT NULL AUTO\_INCREMENT,  name VARCHAR(255) NOT NULL,  price VARCHAR(255) NOT NULL); |

**(3) 向表中插入一些数据**

|  |
| --- |
| INSERT INTO db\_library.book (name, price) VALUES('Lie Sporting', '30');  INSERT INTO db\_library.book (name, price) VALUES('Pride & Prejudice', '70');  INSERT INTO db\_library.book (name, price) VALUES('Fall of Giants', '50'); |

**(4) 执行Sqoop导入数据的操作**

手动创建HBase表

|  |
| --- |
| hbase> create 'hbase\_book','info' |

**(5) 在HBase中scan这张表得到如下内容**

|  |
| --- |
| hbase> scan 'hbase\_book' |

**思考：**尝试使用复合键作为导入数据时的rowkey。

|  |
| --- |
| $ bin/sqoop import \  --connect jdbc:mysql://bigdata11:3306/db\_library \  --username root \  --password 000000 \  --table book \  --columns "id,name,price" \  --column-family "info" \  --hbase-create-table \  --hbase-row-key "id" \  --hbase-table "hbase\_book" \  --num-mappers 1 \  --split-by id |

尖叫提示：sqoop1.4.6只支持HBase1.0.1之前的版本的自动创建HBase表的功能

## 2.9 Phoenix集成

### 1. Phoenix介绍

可以把Phoenix理解为Hbase的查询引擎，phoenix，由saleforce.com开源的一个项目，后又捐给了Apache。它相当于一个Java中间件，帮助开发者，像使用jdbc访问关系型数据库一些，访问NoSql数据库HBase。

phoenix，操作的表及数据，存储在hbase上。phoenix只是需要和Hbase进行表关联起来。然后再用工具进行一些读或写操作。

其实，可以把Phoenix只看成一种代替HBase的语法的一个工具。虽然可以用java可以用jdbc来连接phoenix，然后操作HBase，但是在生产环境中，不可以用在OLTP中。在线事务处理的环境中，需要低延迟，而Phoenix在查询HBase时，虽然做了一些优化，但**延迟还是不小**。所以依然是用在OLAT中，再将结果返回存储下来。

### 2.phoenix安装包解压缩更换目录

tar -zxvf apache-phoenix-4.14.1-HBase-1.2-bin.tar.gz -C /opt/module

mv apache-phoenix-4.14.1-HBase-1.2-bin phoenix-4.14.1

环境变量vi /etc/profile

#在最后两行加上如下phoenix配置

export PHOENIX\_HOME=/opt/module/phoenix-4.14.1

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

#使环境变量配置生效

source /etc/profile

将主节点的phoenix包传到从节点

$ scp -r phoenix-4.14.1 root@bigdata13:/opt/module

$ scp -r phoenix-4.14.1 root@bigdata12:/opt/module

拷贝hbase-site.xml（注）三台都要

cp hbase-site.xml /opt/module/phoenix-4.14.1/bin/

将如下两个jar包，目录在/opt/module/phoenix-4.14.1下，拷贝到hbase的lib目录，目录在/opt/module/hbase-1.3.1/lib/

（注）三台都要

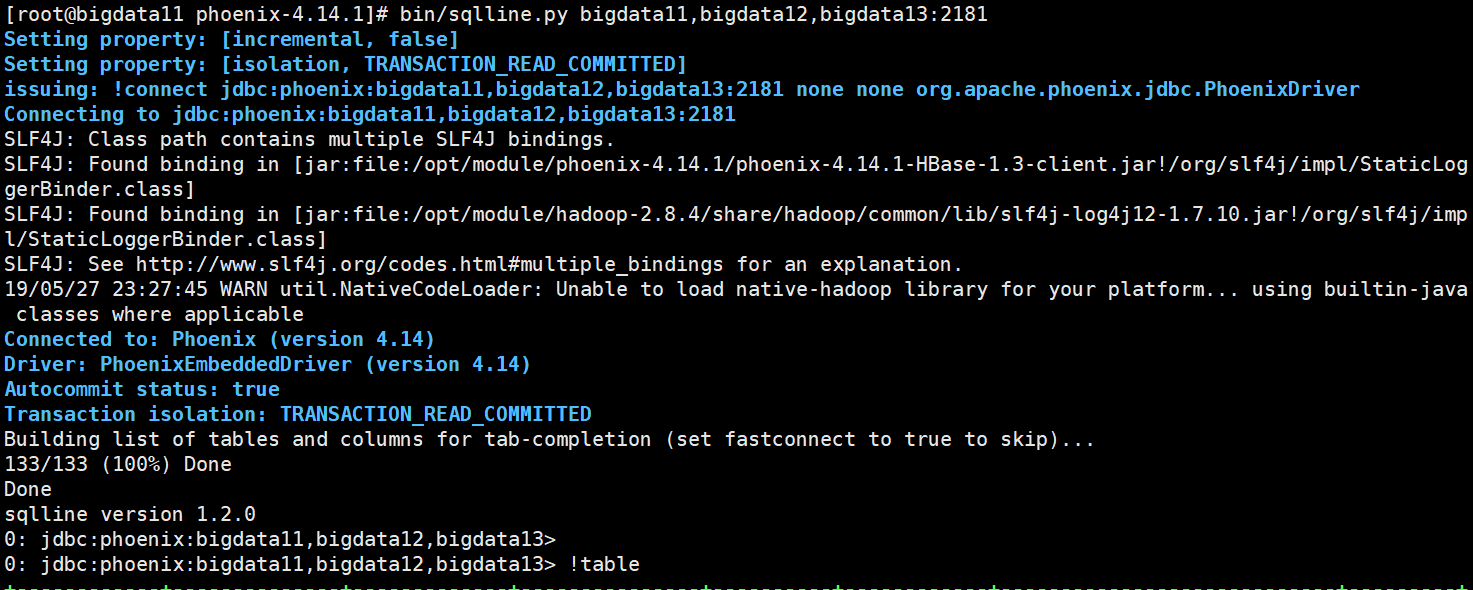
phoenix-4.10.0-HBase-1.2-server.jar

phoenix-core-4.10.0-HBase-1.2.jar

启动Phoenix

配置好之后重启下hbase。

sqlline.py bigdata11:2181



#### 基本命令

#展示表

> !table

#创建表

> create table test(id integer not null primary key,name varchar);

> create table "Andy"(

id integer not null primary key,

name varchar);

#删除表

drop table test;

#插入数据

> upsert into test values(1,'Andy');

> upsert into users(name) values('toms');

#查询数据

phoenix > select \* from test;

hbase > scan 'test'

#退出phoenix

> !q

#删除数据

delete from "Andy" where id=4;

#sum函数的使用

select sum(id) from "Andy";

#增加一列

alter table "Andy" add address varchar;

#删除一列

alter table "Andy" drop column address;

**其他语法详见：**<http://phoenix.apache.org/language/index.html>

#### 表映射

#hbase中创建表

create 'teacher','info','contact'

#插入数据

put 'teacher','1001','info:name','Jack'

put 'teacher','1001','info:age','28'

put 'teacher','1001','info:gender','male'

put 'teacher','1001','contact:address','shanghai'

put 'teacher','1001','contact:phone','13458646987'

put 'teacher','1002','info:name','Jim'

put 'teacher','1002','info:age','30'

put 'teacher','1002','info:gender','male'

put 'teacher','1002','contact:address','tianjian'

put 'teacher','1002','contact:phone','13512436987'

#在Phoenix创建映射表

create view "teacher"(

"ROW" varchar primary key,

"contact"."address" varchar,

"contact"."phone" varchar,

"info"."age" varchar,

"info"."gender" varchar,

"info"."name" varchar

);

#在Phoenix查找数据

select \* from "teacher";

## 2.10、节点的管理

### 2.10.1、服役（commissioning）

当启动regionserver时，regionserver会向HMaster注册并开始接收本地数据，开始的时候，新加入的节点不会有任何数据，平衡器开启的情况下，将会有新的region移动到开启的RegionServer上。如果启动和停止进程是使用ssh和HBase脚本，那么会将新添加的节点的主机名加入到conf/regionservers文件中。

1）$ ./bin/hbase-daemon.sh stop regionserver

2）hbase(main):001:0>balance\_switch true

### 2.10.2、退役（decommissioning）

顾名思义，就是从当前HBase集群中删除某个RegionServer，这个过程分为如下几个过程：

在0.90.2之前，我们只能通过在要卸载的节点上执行

**1) 停止负载平衡器**

|  |
| --- |
| hbase> balance\_switch false |

**2) 在退役节点上停止RegionServer**

|  |
| --- |
| [root@bigdata11 hbase-1.3.1] hbase-daemon.sh stop regionserver |

**3) RegionServer一旦停止，会关闭维护的所有region**

**4) Zookeeper上的该RegionServer节点消失**

**5) Master节点检测到该RegionServer下线，开启平衡器**

|  |
| --- |
| hbase> balance\_switch true |

**6) 下线的RegionServer的region服务得到重新分配**

这种方法很大的一个缺点是该节点上的Region会离线很长时间。因为假如该RegionServer上有大量Region的话，因为Region的关闭是顺序执行的，第一个关闭的Region得等到和最后一个Region关闭并Assigned后一起上线。这是一个相当漫长的时间。每个Region Assigned需要4s，也就是说光Assigned就至少需要2个小时。该关闭方法比较传统，需要花费一定的时间，而且会造成部分region短暂的不可用。

**另一种方案：**

1. **新方法**

自0.90.2之后，HBase添加了一个新的方法，即“graceful\_stop”,只需要在HBase Master节点执行

|  |
| --- |
| $ bin/graceful\_stop.sh <RegionServer-hostname> |

该命令会自动关闭Load Balancer，然后Assigned Region，之后会将该节点关闭。除此之外，你还可以查看remove的过程，已经assigned了多少个Region，还剩多少个Region，每个Region 的Assigned耗时

1. **开启负载平衡器**

|  |
| --- |
| hbase> balance\_switch false |

# **三、HBase的优化**

## **3.1、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）**

|  |
| --- |
| $ bin/stop-hbase.sh |

**2) 在conf目录下创建backup-masters文件**

|  |
| --- |
| $ touch conf/backup-masters |

**3) 在backup-masters文件中配置高可用HMaster节点**

|  |
| --- |
| $ echo bigdata112 > conf/backup-masters |

**4) 将整个conf目录scp到其他节点**

|  |
| --- |
| $ scp -r conf/ bigdata112:/opt/module/hbase-1.3.1  $ scp -r conf/ bigdata113:/opt/module/hbase-1.3.1 |

**5) 重新启动HBase后打开页面测试查看**

|  |
| --- |
| 0.98版本之后：[http://bigdata111:16010](http://linux01:16010) |

## **3.2、Hadoop的通用性优化**

**1) NameNode元数据备份使用SSD**

**2) 定时备份NameNode上的元数据**

每小时或者每天备份，如果数据极其重要，可以5~10分钟备份一次。备份可以通过定时任务复制元数据目录即可。

**3) 为NameNode指定多个元数据目录**

使用dfs.name.dir或者dfs.namenode.name.dir指定。这样可以提供元数据的冗余和健壮性，以免发生故障。

**4) NameNode的dir自恢复**

设置**dfs.namenode.name.dir.restore为true**，允许尝试恢复之前失败的dfs.namenode.name.dir目录，在创建checkpoint时做此尝试，如果设置了多个磁盘，建议允许。

**5) HDFS保证RPC调用会有较多的线程数**

**hdfs-site.xml**

|  |
| --- |
| 属性：dfs.namenode.handler.count  解释：该属性是NameNode服务默认线程数，的默认值是10，根据机器的可用内存可以调整为50~100  属性：dfs.datanode.handler.count  解释：该属性默认值为10，是DataNode的处理线程数，如果HDFS客户端程序读写请求比较多，可以调高到15~20，设置的值越大，内存消耗越多，不要调整的过高，一般业务中，5~10即可。 |

**6) HDFS副本数的调整**

**hdfs-site.xml**

|  |
| --- |
| 属性：dfs.replication  解释：如果数据量巨大，且不是非常之重要，可以调整为2~3，如果数据非常之重要，可以调整为3~5。 |

**7) HDFS文件块大小的调整**

**hdfs-site.xml**

|  |
| --- |
| 属性：dfs.blocksize  解释：块大小定义，该属性应该根据存储的大量的单个文件大小来设置，如果大量的单个文件都小于100M，建议设置成64M块大小，对于大于100M或者达到GB的这种情况，建议设置成256M，一般设置范围波动在64M~256M之间。 |

**8) MapReduce Job任务服务线程数调整**

**mapred-site.xml**

|  |
| --- |
| 属性：mapreduce.jobtracker.handler.count  解释：该属性是Job任务线程数，默认值是10，根据机器的可用内存可以调整为50~100 |

**9) Http服务器工作线程数**

**mapred-site.xml**

|  |
| --- |
| 属性：mapreduce.tasktracker.http.threads  解释：定义HTTP服务器工作线程数，默认值为40，对于大集群可以调整到80~100 |

**10) 文件排序合并优化**

**mapred-site.xml**

|  |
| --- |
| 属性：mapreduce.task.io.sort.factor  解释：文件排序时同时合并的数据流的数量，这也定义了同时打开文件的个数，默认值为10，如果调高该参数，可以明显减少磁盘IO，即减少文件读取的次数。 |

**11) 设置任务并发**

**mapred-site.xml**

|  |
| --- |
| 属性：mapreduce.map.speculative  解释：该属性可以设置任务是否可以并发执行，如果任务多而小，该属性设置为true可以明显加快任务执行效率，但是对于延迟非常高的任务，建议改为false，这就类似于迅雷下载。 |

**12) MR输出数据的压缩**

**mapred-site.xml**

|  |
| --- |
| 属性：mapreduce.map.output.compress、mapreduce.output.fileoutputformat.compress  解释：对于大集群而言，建议设置Map-Reduce的输出为压缩的数据，而对于小集群，则不需要。 |

**13) 优化Mapper和Reducer的个数**

**mapred-site.xml**

|  |
| --- |
| 属性：  mapreduce.tasktracker.map.tasks.maximum  mapreduce.tasktracker.reduce.tasks.maximum  解释：以上两个属性分别为一个单独的Job任务可以同时运行的Map和Reduce的数量。  设置上面两个参数时，需要考虑CPU核数、磁盘和内存容量。假设一个8核的CPU，业务内容非常消耗CPU，那么可以设置map数量为4，如果该业务不是特别消耗CPU类型的，那么可以设置map数量为40，reduce数量为20。这些参数的值修改完成之后，一定要观察是否有较长等待的任务，如果有的话，可以减少数量以加快任务执行，如果设置一个很大的值，会引起大量的上下文切换，以及内存与磁盘之间的数据交换，这里没有标准的配置数值，需要根据业务和硬件配置以及经验来做出选择。  在同一时刻，不要同时运行太多的MapReduce，这样会消耗过多的内存，任务会执行的非常缓慢，我们需要根据CPU核数，内存容量设置一个MR任务并发的最大值，使固定数据量的任务完全加载到内存中，避免频繁的内存和磁盘数据交换，从而降低磁盘IO，提高性能。 |

**大概估算公式：**

map = 2 + ⅔cpu\_core

reduce = 2 + ⅓cpu\_core

## **3.3、Linux优化**

**1) 开启文件系统的预读缓存可以提高读取速度**

|  |
| --- |
| $ sudo blockdev --setra 32768 /dev/sda |

尖叫提示：ra是readahead的缩写

**2) 关闭进程睡眠池**

即不允许后台进程进入睡眠状态，如果进程空闲，则直接kill掉释放资源

|  |
| --- |
| $ sudo sysctl -w vm.swappiness=0 |

**3) 调整ulimit上限，默认值为比较小的数字**

|  |
| --- |
| $ ulimit -n 查看允许最大进程数  $ ulimit -u 查看允许打开最大文件数 |

**优化修改：**

|  |
| --- |
| $ sudo vi /etc/security/limits.conf 修改打开文件数限制  末尾添加：  \* soft nofile 1024000  \* hard nofile 1024000  Hive - nofile 1024000  hive - nproc 1024000  $ sudo vi /etc/security/limits.d/20-nproc.conf 修改用户打开进程数限制  修改为：  #\* soft nproc 4096  #root soft nproc unlimited  \* soft nproc 40960  root soft nproc unlimited |

**4) 开启集群的时间同步NTP**

集群中某台机器同步网络时间服务器的时间，集群中其他机器则同步这台机器的时间。

**5) 更新系统补丁**

更新补丁前，请先测试新版本补丁对集群节点的兼容性。

## **3.4、Zookeeper优化**

**1) 优化Zookeeper会话超时时间**

**hbase-site.xml**

|  |
| --- |
| 参数：zookeeper.session.timeout  解释：In hbase-site.xml, set zookeeper.session.timeout to 30 seconds or less to bound failure detection (20-30 seconds is a good start).该值会直接关系到master发现服务器宕机的最大周期，默认值为30秒（不同的HBase版本，该默认值不一样），如果该值过小，会在HBase在写入大量数据发生而GC时，导致RegionServer短暂的不可用，从而没有向ZK发送心跳包，最终导致认为从节点shutdown。一般20台左右的集群需要配置5台zookeeper。 |

## **3.5、HBase优化**

### **3.5.1、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，我们可以将数据索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 'staff','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 => '/opt/module/hbase-1.3.1/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); |

### **3.5.2、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**

|  |
| --- |
| 比如：  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 |

### **3.5.3、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。

### **3.5.4、基础优化**

**1) 允许在HDFS的文件中追加内容**

不是不允许追加内容么？没错，请看背景故事：

[http://blog.cloudera.com/blog/2009/07/file-appends-in-hdfs/](http://blog.cloudera.com)

**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) 优化DataNode存储**

|  |
| --- |
| 属性：dfs.datanode.failed.volumes.tolerated  解释： 默认为0，意思是当DataNode中有一个磁盘出现故障，则会认为该DataNode shutdown了。如果修改为1，则一个磁盘出现故障时，数据会被复制到其他正常的DataNode上，当前的DataNode继续工作。 |

**6) 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。 |

**7) 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。 |

**8) 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。 |

**9) 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。 |

**10) 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