尚硅谷大数据技术之 HBase

(作者：尚硅谷大数据研发部)

版本：V1.3

第 1 章 HBase 简介

http://abloz.com/hbase/book.html#schema.creation

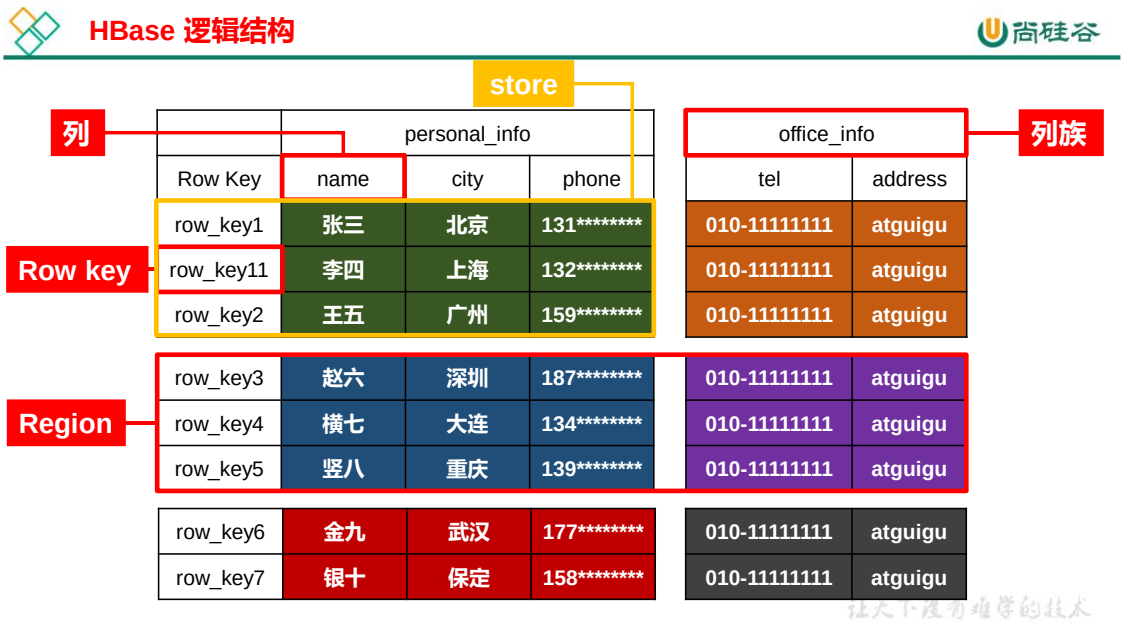
1.1 HBase 定义

HBase 是一种分布式、可扩展、支持海量数据存储的 NoSQL 数据库。

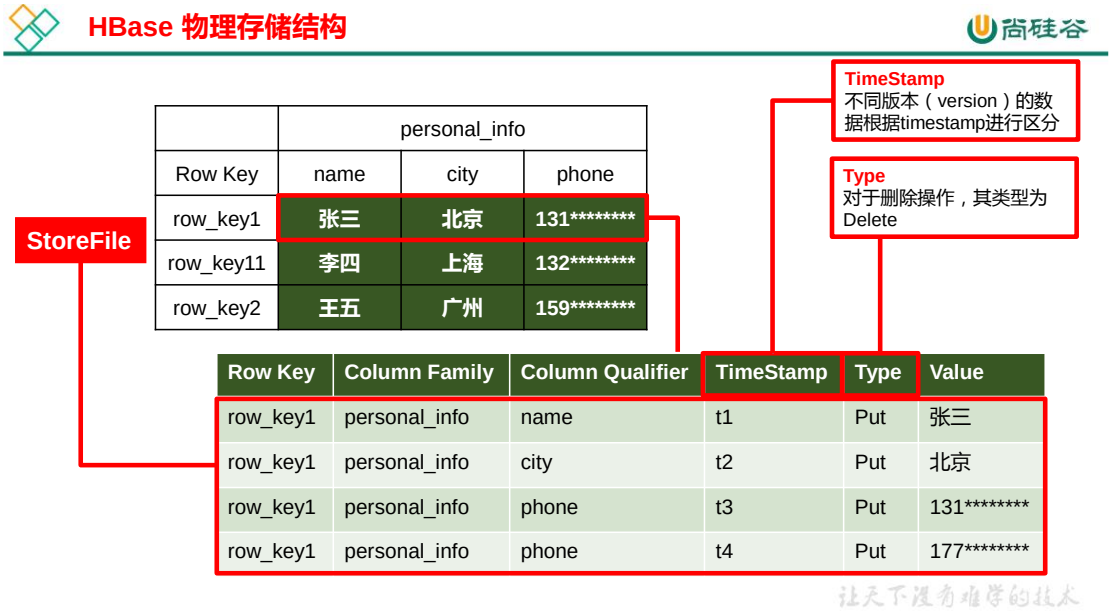
1.2 HBase 数据模型

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

1.2.1 HBase 逻辑结构



1.2.2 HBase 物理存储结构



1.2.3 数据模型

1）Name Space

命名空间，类似于关系型数据库的 DatabBase 概念，每个命名空间下有多个表。HBase 有两个自带的命名空间，分别是 hbase 和 default，hbase 中存放的是 HBase 内置的表，default 表是用户默认使用的命名空间。

2）Region

类似于关系型数据库的表概念。不同的是，HBase 定义表时只需要声明列族即可，不需要声明具体的列。这意味着，往 HBase 写入数据时，字段可以动态、按需指定。因此，和关系型数据库相比，HBase 能够轻松应对字段变更的场景。

3）Row

HBase 表中的每行数据都由一个 RowKey 和多个 Column（列）组成，数据是按照 RowKey 的字典顺序存储的，并且查询数据时只能根据 RowKey 进行检索，所以 RowKey 的设计十分重要。

4）Column

HBase 中的每个列都由 Column Family(列族)和 Column Qualifier（列限定符）进行限定，例如 info：name，info：age。建表时，只需指明列族，而列限定符无需预先定义。

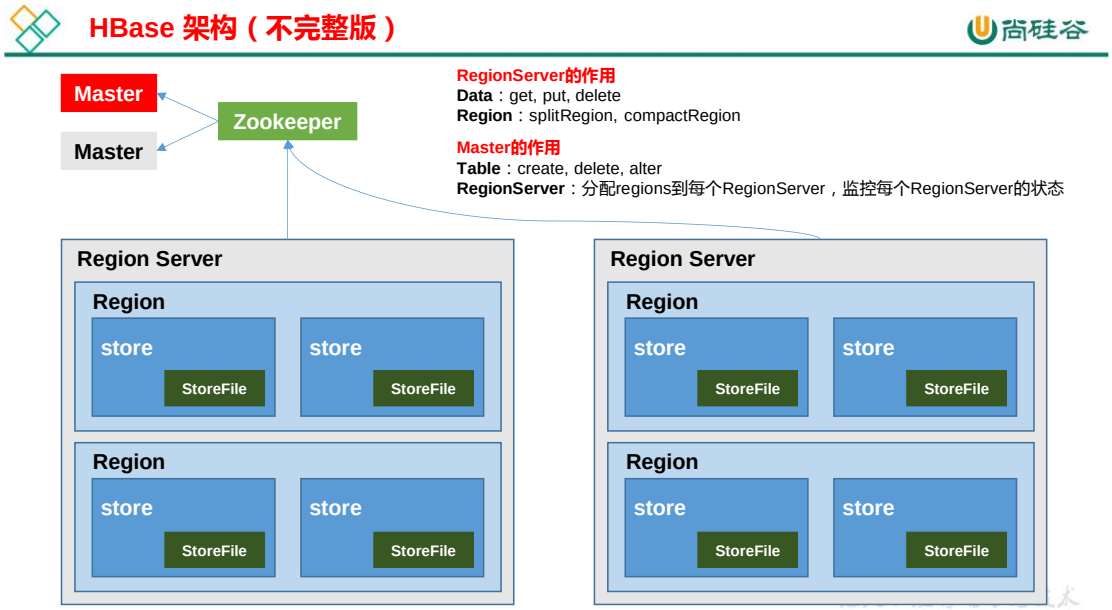
5）Time Stamp

用于标识数据的不同版本（version），每条数据写入时，如果不指定时间戳，系统会自动为其加上该字段，其值为写入 HBase 的时间。

6）Cell

由{rowkey, column Family：column Qualifier, time Stamp} 唯一确定的单元。cell 中的数据是没有类型的，全部是字节码形式存贮。

1.3 HBase 基本架构



架构角色：

1）Region Server

Region Server 为 Region 的管理者，其实现类为 HRegionServer，主要作用如下：

对于数据的操作：get, put, delete;

对于 Region 的操作：splitRegion、compactRegion。

2）Master

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

对于表的操作：create, delete, alter

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

3）Zookeeper

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

4）HDFS

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

第 2 章 HBase 快速入门

2.1 HBase 安装部署

2.1.1 Zookeeper 正常部署

首先保证 Zookeeper 集群的正常部署，并启动之：

```
[atguigu@hadoop102 zookeeper-3.4.10]$ bin/zkServer.sh start
[atguigu@hadoop103 zookeeper-3.4.10]$ bin/zkServer.sh start
[atguigu@hadoop104 zookeeper-3.4.10]$ bin/zkServer.sh start
```

2.1.2 Hadoop 正常部署

Hadoop 集群的正常部署并启动：

```
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh
[atguigu@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh
```

2.1.3 HBase 的解压

解压 Hbase 到指定目录：

```
[atguigu@hadoop102 software]$ tar -zxvf hbase-1.3.1-bin.tar.gz -C
/opt/module
```

2.1.4 HBase 的配置文件

修改 HBase 对应的配置文件。

1）hbase-env.sh 修改内容：

```
export JAVA_HOME=/opt/module/jdk1.6.0_144
export HBASE_MANAGES_ZK=false
```

2）hbase-site.xml 修改内容：

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://hadoop102:9000/HBase</value>
    </property>

    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>

   <!-- 0.98 后的新变动，之前版本没有.port,默认端口为 60000 -->
    <property>
        <name>hbase.master.port</name>
        <value>16000</value>
    </property>

    <property>
        <name>hbase.zookeeper.quorum</name>
         <value>hadoop102,hadoop103,hadoop104</value>
```

```
    </property>

    <property>
        <name>hbase.zookeeper.property.dataDir</name>
         <value>/opt/module/zookeeper-3.4.10/zkData</value>
    </property>
</configuration>
```

3）regionservers：

```
hadoop102
hadoop103
hadoop104
```

4）软连接 hadoop 配置文件到 HBase：

```
[atguigu@hadoop102 module]$ ln -s /opt/module/hadoop-2.7.2/etc/hadoop/core-site.xml /opt/module/hbase/conf/core-site.xml
[atguigu@hadoop102 module]$ ln -s /opt/module/hadoop-2.7.2/etc/hadoop/hdfs-site.xml /opt/module/hbase/conf/hdfs-site.xml
```

2.1.5 HBase 远程发送到其他集群

```
[atguigu@hadoop102 module]$ xsync hbase/
```

2.1.6 HBase 服务的启动

1．启动方式

启动方式1只能启动单节点

```
 [atguigu@hadoop102 hbase]$ bin/hbase-daemon.sh start master
[atguigu@hadoop102 hbase]$ bin/hbase-daemon.sh start regionserver
```

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

修复提示：

a、同步时间服务

请参看帮助文档：《尚硅谷大数据技术之 Hadoop 入门》

b、属性：hbase.master.maxclockskew 设置更大的值

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

2．启动方式 2

集群启动使用启动方式2

```
[atguigu@hadoop102 hbase]$ bin/start-hbase.sh
```

对应的停止服务：

```
[atguigu@hadoop102 hbase]$ bin/stop-hbase.sh
```

2.1.7 查看 HBase 页面

启动成功后，可以通过“host:port”的方式来访问 HBase 管理页面，例如：

http://hadoop102:16010

2.2 HBase Shell 操作

2.2.1 基本操作

1．进入 HBase 客户端命令行

```
[atguigu@hadoop102 hbase]$
```

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
```

2.2.2 表的操作

1．创建表

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

2．插入数据到表

```
hbase(main):003:0> put 'student','1001','info:sex','male'
hbase(main):004:0> put 'student','1001','info:age','18'
hbase(main):005:0> put 'student','1002','info:name','Janna'
hbase(main):006:0> put 'student','1002','info:sex','female'
hbase(main):007:0> put 'student','1002','info:age','20'
```

3．扫描查看表数据

```
hbase(main):008:0> scan 'student'
hbase(main):009:0> scan 'student',{STARTROW => '1001', STOPROW  =>
'1001'}
hbase(main):010:0> scan 'student',{STARTROW => '1001'}
```

4．查看表结构

```
hbase(main):011:0> describe 'student'
```

5．更新指定字段的数据

```
hbase(main):012:0> put 'student','1001','info:name','Nick'
hbase(main):013:0> put 'student','1001','info:age','100'
```

6．查看“指定行”或“指定列族:列”的数据

```
hbase(main):014:0> get 'student','1001'
hbase(main):015:0> get 'student','1001','info:name'
```

7．统计表数据行数

```
hbase(main):021:0> count 'student'
```

8．删除数据

删除某 rowkey 的全部数据：

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

删除某 rowkey 的某一列数据：

```
list_namespace：查看命名空间
list_namespace_tables 'ns1'：列出指定命名空间下的所有表

因为hbase中没有rename命令，所以更改表名比较复杂。
disable 'tableName'    # 1.停止表继续插入
snapshot 'tableName', 'tableSnapshot' # 2。制作快照
clone_snapshot 'tableSnapshot', 'newTableName' # 3.克隆快照为新的名字
delete_snapshot 'tableSnapshot' # 4.删除快照
drop 'tableName' # 5.删除原来表
```

```
create 'student',{NAME=>'info',VERSIONS=>3}
// 插入数
put 'student','1001','info:sex','male'
put 'student','1001','info:age','18'
put 'student','1002','info:name','Janna'
put 'student','1001','info:sex','female'
// 查询  插入的1001会放到1002之前，说明会插入到region的对应位置
hbase(main):051:0> scan 'student', {COLUMNS => ['info'], VERSIONS=>3}
ROW           COLUMN+CELL
 1001         column=info:age, timestamp=1646294030492, value=18
 1001         column=info:sex, timestamp=1646294031442, value=female
 1001         column=info:sex, timestamp=1646294030459, value=male
 1002         column=info:name, timestamp=1646294030522, value=Janna
2 row(s)
```

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

9．清空表数据

```
hbase(main):018:0> truncate 'student'
```

提示：清空表的操作顺序为先 disable，然后再 truncate。

10．删除表

首先需要先让该表为 disable 状态：

```
hbase(main):019:0> disable 'student'
```

然后才能 drop 这个表：

```
hbase(main):020:0> drop 'student'
```

提示：如果直接 drop 表，会报错：ERROR: Table student is enabled. Disable it first.

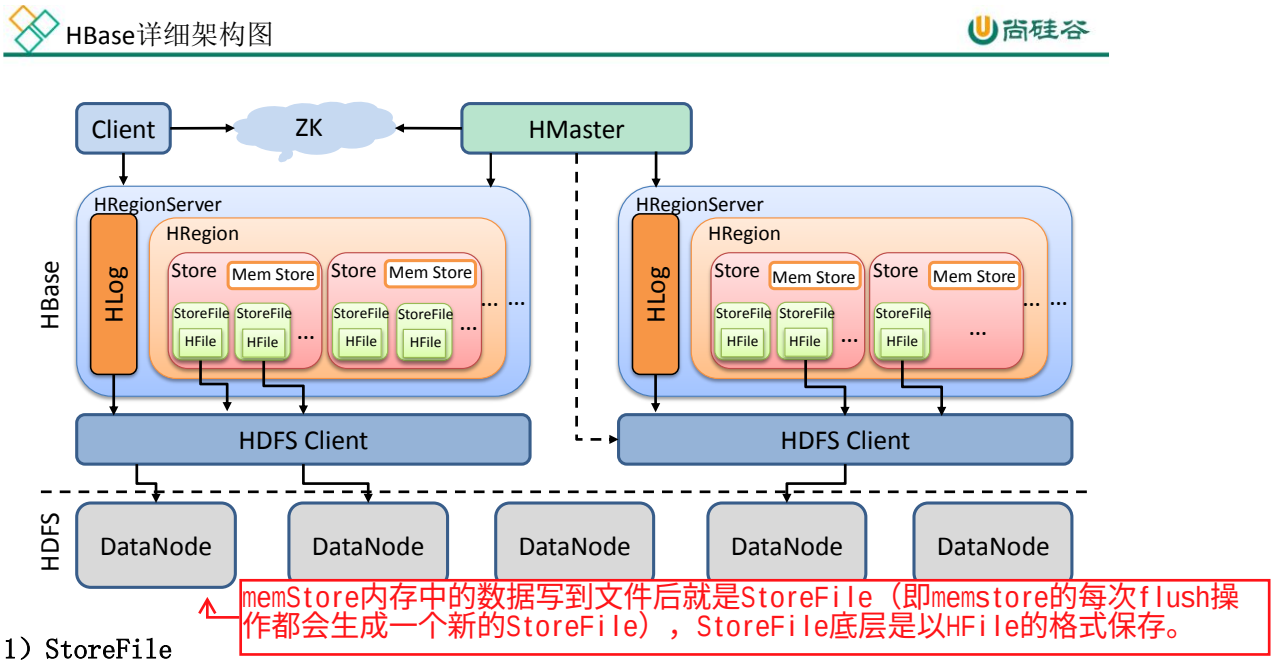
11．变更表信息

将 info 列族中的数据存放 3 个版本：

```
hbase(main):022:0> alter 'student',{NAME=>'info',VERSIONS=>3}
hbase(main):022:0>                                                  get
'student','1001',{COLUMN=>'info:name',VERSIONS=>3}
```

第 3 章 HBase 进阶

3.1 架构原理



1）StoreFile

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

2）MemStore ← HFile是HBase中KeyValue数据的存储格式，是hadoop的二进制格式文件。一个StoreFile对应着一个HFile。

写缓存，由于 HFile 中的数据要求是有序的，所以数据是先存储在 MemStore 中，排好

meta表存放的就是region的元数据信息。meta表格式如下：

```
ROW                                                  COLUMN+CELL
user,,1577522582939.b84a96d0e074272569b6fa79946e79df.    column=info:regioninfo,
timestamp=1577522584346, value={ENCODED => b84a96d0e074272569b6fa79946e79df, NAME =>
'user,,1577522582939.b84a96d0e074272569b6fa79946e79df.', STARTKEY => '', ENDKEY => ''}

user,,1577522582939.b84a96d0e074272569b6fa79946e79df.    column=info:seqnumDuringOpen,
timestamp=1583992456271, value=\x00\x00\x00\x00\x00\x00\x00[

user,,1577522582939.b84a96d0e074272569b6fa79946e79df.    column=info:server,
timestamp=1583992456271, value=hadoop01:16020

user,,1577522582939.b84a96d0e074272569b6fa79946e79df.    column=info:serverstartcode,
timestamp=1583992456271, value=1583992398626
```

ROW

hbase:meta表的一个rowkey就对应该表的一个region

hbase:meta 表的 rowkey 结构如下：

TableName,StartKey,Timestamp.EncodedName.
TableName：表名称
StartKey：表示当前 table 的 region 中存储的第一个 rowkey。如果这个地方为空的话，表明这是 table 的第一个 region。并且如果一个 region 中 StartKey 和 EndKey 都为空的话，表明这个 table 只有一个 region；比如上面的user表就只有一个region(STARTKEY => '', ENDKEY => '')。
Timestamp：region 创建的时间戳
EncodedName：TableName,StartKey,Timestamp字符串的MD5 Hex值。
COLUMN+CELL

每一行数据又分为4列，分别是info:regioninfo、info:seqnumDuringOpen、info:server、info:serverstartcode。

info:regioninfo：该列对应的 Value 主要存储4个信息，即EncodedName、RegionName、Region的StartRow、Region的EndRow。

info:seqnumDuringOpen：该列对应的 Value 主要存储Region打开时的sequenceId。

info:server：该列对应的 Value 主要存储Region落在哪个RegionServer上。

info:serverstartcode：该列对应的 Value 主要存储所在RegionServer的启动Timestamp。

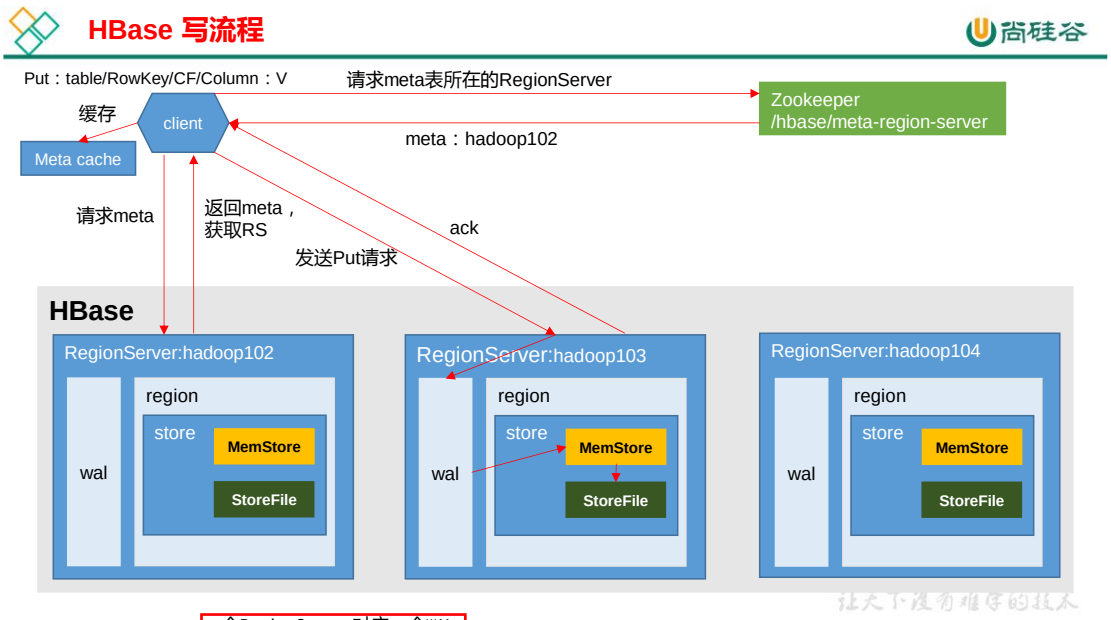
因此，通过meta表，客户端就知道所要查询的数据放在哪个RegionServer上的哪个region。本例子返回的就是RegionServer1中的region1。

序后，等到达刷写时机才会刷写到 HFile，每次刷写都会形成一个新的 HFile。

3）WAL

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

3.2 写流程



写流程：

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

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

3）与目标 Region Server 进行通讯；

4）将数据顺序写入（追加）到 WAL；

按照字典序排序：基于字母顺序排列的单词按字母顺序排列的方法

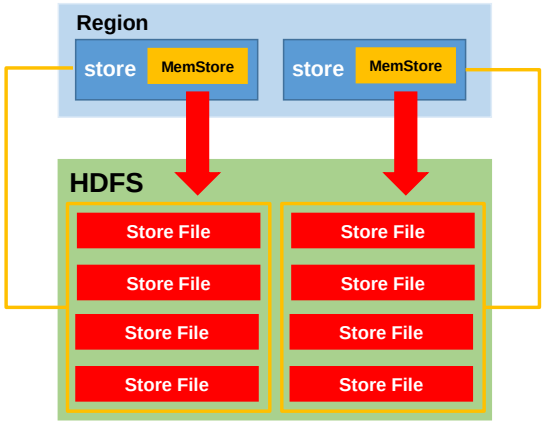
5）将数据写入对应的 MemStore，数据会在 MemStore 进行排序；

6）向客户端发送 ack；

7）等达到 MemStore 的刷写时机后，将数据刷写到 HFile。

3.3 MemStore Flush

MemStore Flush



MemStore 刷写时机：

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

当 memstore 的大小达到了

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

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

时，会阻止继续往该 memstore 写数据。

2.当 region server 中 memstore 的总大小达到

java_heapsize

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

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

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

当 region server 中 memstore 的总大小达到

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

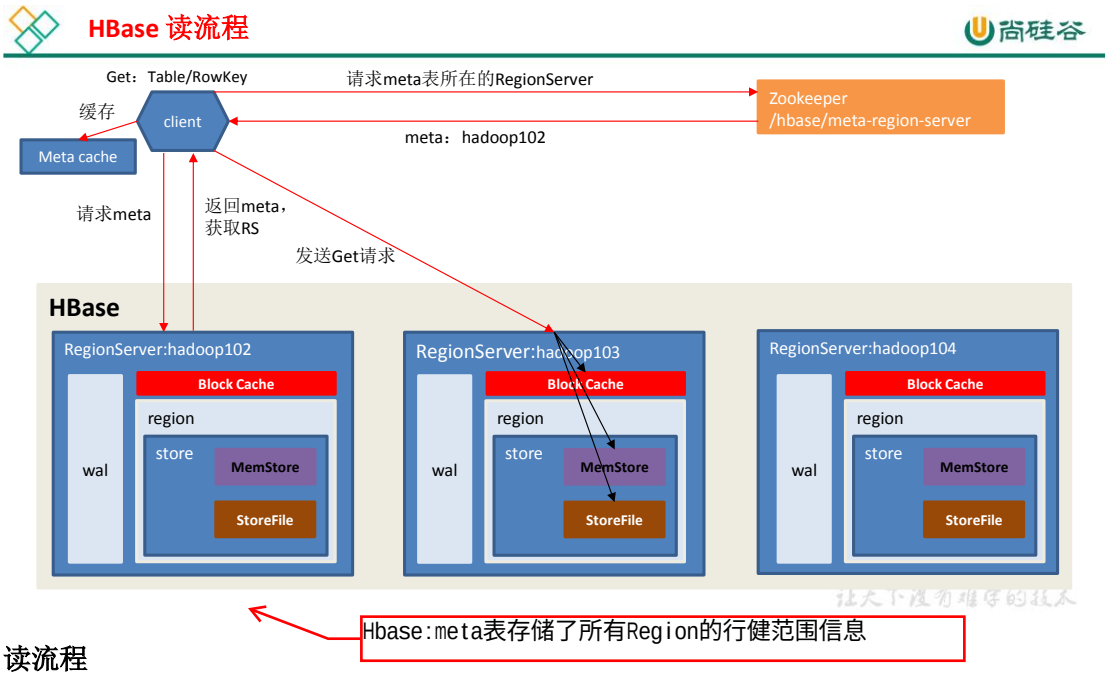
时，会阻止继续往所有的 memstore 写数据。

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

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

3.4 读流程

一个WAL实例包含有多个WAL文件。WAL文件的最大数量通过
hbase.regionserver.maxlogs（默认是32）参数来定义。
https://blog.csdn.net/x950913/article/details/107279790



读流程

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

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

3）与目标 Region Server 进行通讯；

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

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

6）将合并后的最终结果返回给客户端。

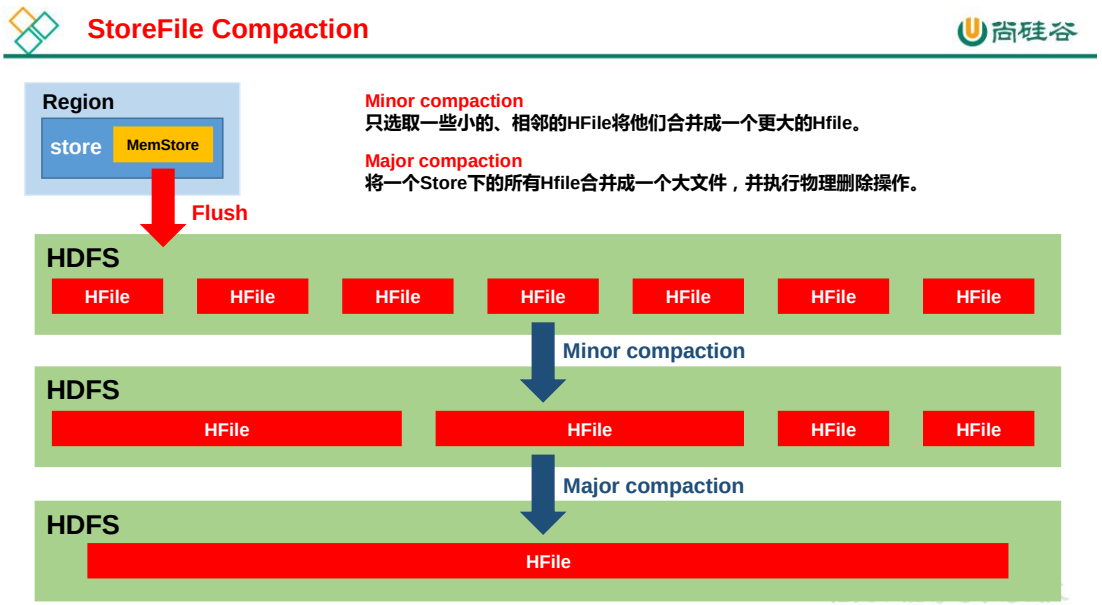
3.5 StoreFile Compaction

由于 memstore 每次刷写都会生成一个新的 HFile，且同一个字段的不同版本（timestamp）和不同类型（Put/Delete）有可能会分布在不同的 HFile 中，因此查询时需要遍历所有的 HFile。

为了减少 HFile 的个数，以及清理掉过期和删除的数据，会进行 StoreFile Compaction。

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

合并时会进行一个排序



3.6 Region Split

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

默认值是 10G，也就是当 Region 的大小达到 10G 的时候，会自动拆分成两个 Region.

Region Split 时机：

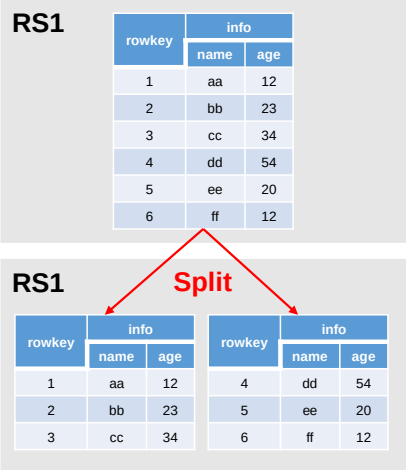
1.当 1 个 region 中的某个 Store 下所有 StoreFile 的总大小超过 hbase.hregion.max.filesize，该 Region 就会进行拆分（0.94 版本之前）。

2. 当 1 个 region 中的某个 Store 下所有 StoreFile 的总大小超过 Min(R^2 * "hbase.hregion.memstore.flush.size",hbase.hregion.max.filesize")，该 Region 就会进行拆分，其中 R 为当前 Region Server 中属于该 Table 的个数（0.94 版本之后）。

region

比如，在默认memstore flush size为128MB且默认的max store size为10G时。（R为region的个数）
第一次拆分大小为：min(10G, 1*1*128M)=128M
第二次拆分大小为：min(10G, 3*3*128M)=1152M
第三次拆分大小为：min(10G, 5*5*128M)=3200M
第四次拆分大小为：min(10G, 7*7*128M)=6272M
第五次拆分大小为：min(10G, 9*9*128M)=10G
第五次拆分大小为：min(10G, 11*11*128M)=10G

Region Split



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

2.当1个region中的某个Store下所有StoreFile的总大小超过**Min(R^2 * "hbase.hregion.memstore.flush.size", " hbase.hregion.max.filesize ")** 就会拆分，其中R为当前RegionServer中属于该table的region个数（0.94版之后）

第 4 章 HBase API

4.1 环境准备

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>
```

4.2 HBaseAPI

4.2.1 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", "192.166.9.102");
    conf.set("hbase.zookeeper.property.clientPort", "2181");
}
```

4.2.2 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);
}
```

4.2.3 创建表

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

4.2.4 删除表

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

4.2.5 向表中插入数据

```
public  static  void  addRowData(String  tableName,  String  rowKey,
String columnFamily, String
 column, String value) throws IOException{
    //创建 HTable 对象
    HTable hTable = new HTable(conf, tableName);
    //向表中插入数据
    Put put = new Put(Bytes.toBytes(rowKey));
    //向 Put 对象中组装数据
```

```
    put.add(Bytes.toBytes(columnFamily),     Bytes.toBytes(column),
Bytes.toBytes(value));
    hTable.put(put);
    hTable.close();
    System.out.println("插入数据成功");
}
```

4.2.6 删除多行数据

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

4.2.7 获取所有数据

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

4.2.8 获取某一行数据

```
public static void getRow(String tableName, String rowKey) throws
IOException{
    HTable table = new HTable(conf, tableName);
    Get get = new Get(Bytes.toBytes(rowKey));
    //get.setMaxVersions();显示所有版本
    //get.setTimeStamp();显示指定时间戳的版本
    Result result = table.get(get);
    for(Cell cell : result.rawCells()){
        System.out.println("        行        键        :"          +
```

```
Bytes.toString(result.getRow()));
        System.out.println("        列        族        "          +
Bytes.toString(CellUtil.cloneFamily(cell)));
        System.out.println("          列          :"              +
Bytes.toString(CellUtil.cloneQualifier(cell)));
        System.out.println("          值          :"              +
Bytes.toString(CellUtil.cloneValue(cell)));
        System.out.println("时间戳:" + cell.getTimestamp());
    }
}
```

4.2.9 获取某一行指定“列族:列”的数据

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

4.3 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。

4.3.1 官方 HBase-MapReduce

1. 查看 HBase 的 MapReduce 任务的执行

```
$ bin/hbase mapredcp
```

2. 环境变量的导入

（1）执行环境变量的导入（临时生效，在命令行执行下述操作）

```
$ export HBASE_HOME=/opt/module/hbase
$ export HADOOP_HOME=/opt/module/hadoop-2.7.2
$ export HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`
```

（2）永久生效：在/etc/profile 配置

```
export HBASE_HOME=/opt/module/hbase
export HADOOP_HOME=/opt/module/hadoop-2.7.2
```

并在 hadoop-env.sh 中配置：（注意：在 for 循环之后配）

export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/opt/module/hbase/lib/*

3．运行官方的 MapReduce 任务

-- 案例一：统计 Student 表中有多少行数据

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

-- 案例二：使用 MapReduce 将本地数据导入到 HBase

1）在本地创建一个 tsv 格式的文件：fruit.tsv

```
1001	Apple	Red
1002	Pear	Yellow
1003	Pineapple	Yellow
```

2）创建 Hbase 表

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

3）在 HDFS 中创建 input_fruit 文件夹并上传 fruit.tsv 文件

```
$ /opt/module/hadoop-2.7.2/bin/hdfs dfs -mkdir /input_fruit/
$    /opt/module/hadoop-2.7.2/bin/hdfs    dfs    -put    fruit.tsv
/input_fruit/
```

4）执行 MapReduce 到 HBase 的 fruit 表中

```
$ /opt/module/hadoop-2.7.2/bin/yarn jar lib/hbase-server-1.3.1.jar
importtsv \
-Dimporttsv.columns=HBASE_ROW_KEY,info:name,info:color fruit \
hdfs://hadoop102:9000/input_fruit
```

5）使用 scan 命令查看导入后的结果

```
Hbase(main):001:0> scan ‘fruit’
```

4.3.2 自定义 HBase-MapReduce1

目标：将 fruit 表中的一部分数据，通过 MR 迁入到 fruit_mr 表中。

分步实现：

1．构建 ReadFruitMapper 类，用于读取 fruit 表中的数据

```
package com.atguigu;

import java.io.IOException;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.CellUtil;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.mapreduce.TableMapper;
import org.apache.hadoop.hbase.util.Bytes;

public class ReadFruitMapper extends
TableMapper<ImmutableBytesWritable, Put> {

    @Override
```

```
    protected void map(ImmutableBytesWritable key, Result value,
Context context)
    throws IOException, InterruptedException {
    //将 fruit 的 name 和 color 提取出来，相当于将每一行数据读取出来放入到 Put
对象中。
        Put put = new Put(key.get());
        //遍历添加 column 行
        for(Cell cell: value.rawCells()){
            //添加/克隆列族:info

    if("info".equals(Bytes.toString(CellUtil.cloneFamily(cell)))){
                //添加/克隆列：name

    if("name".equals(Bytes.toString(CellUtil.cloneQualifier(cell))
)){
                    //将该列 cell 加入到 put 对象中
                    put.add(cell);
                    //添加/克隆列:color
                }else
if("color".equals(Bytes.toString(CellUtil.cloneQualifier(cell))))
{
                    //向该列 cell 加入到 put 对象中
                    put.add(cell);
                }
            }
        }
        //将从 fruit 读取到的每行数据写入到 context 中作为 map 的输出
        context.write(key, put);
    }
}
```

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

```
package com.atguigu.Hbase_mr;

import java.io.IOException;
import org.apache.hadoop.Hbase.client.Put;
import org.apache.hadoop.Hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.Hbase.mapreduce.TableReducer;
import org.apache.hadoop.io.NullWritable;

public          class          WriteFruitMRReducer          extends
TableReducer<ImmutableBytesWritable, Put, NullWritable> {
    @Override
    protected void reduce(ImmutableBytesWritable key, Iterable<Put>
values, Context context)
    throws IOException, InterruptedException {
        //读出来的每一行数据写入到 fruit_mr 表中
        for(Put put: values){
            context.write(NullWritable.get(), put);
        }
    }
}
```

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

任务

```
//组装 Job
    public int run(String[] args) throws Exception {
        //得到 Configuration
        Configuration conf = this.getConf();
        //创建 Job 任务
        Job job = Job.getInstance(conf,
this.getClass().getSimpleName());
        job.setJarByClass(Fruit2FruitMRRunner.class);

        //配置 Job
        Scan scan = new Scan();
        scan.setCacheBlocks(false);
        scan.setCaching(500);

        //设置 Mapper，注意导入的是 mapreduce 包下的，不是 mapred 包下的，后者
是老版本
        TableMapReduceUtil.initTableMapperJob(
        "fruit", //数据源的表名
        scan, //scan 扫描控制器
        ReadFruitMapper.class,//设置 Mapper 类
        ImmutableBytesWritable.class,//设置 Mapper 输出 key 类型
        Put.class,//设置 Mapper 输出 value 值类型
        job//设置给哪个 JOB
        );
        //设置 Reducer
        TableMapReduceUtil.initTableReducerJob("fruit_mr",
WriteFruitMRReducer.class, job);
        //设置 Reduce 数量，最少 1 个
        job.setNumReduceTasks(1);

        boolean isSuccess = job.waitForCompletion(true);
        if(!isSuccess){
            throw new IOException("Job running with error");
        }
        return isSuccess ? 0 : 1;
    }
```

4．主函数中调用运行该 Job 任务

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

5．打包运行任务

```
$  /opt/module/hadoop-2.7.2/bin/yarn  jar  ~/softwares/jars/Hbase-
0.0.1-SNAPSHOT.jar
 com.z.Hbase.mr1.Fruit2FruitMRRunner
```

提示：运行任务前，如果待数据导入的表不存在，则需要提前创建。

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

4.3.3 自定义 Hbase-MapReduce2

目标：实现将 HDFS 中的数据写入到 Hbase 表中。

分步实现：

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

```
package com.atguigu;

import java.io.IOException;

import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class ReadFruitFromHDFSMapper extends Mapper<LongWritable,
Text, ImmutableBytesWritable, Put> {
    @Override
    protected void map(LongWritable key, Text value, Context
context) throws IOException, InterruptedException {
        //从 HDFS 中读取的数据
        String lineValue = value.toString();
        //读取出来的每行数据使用\t 进行分割，存于 String 数组
        String[] values = lineValue.split("\t");

        //根据数据中值的含义取值
        String rowKey = values[0];
        String name = values[1];
        String color = values[2];

        //初始化 rowKey
        ImmutableBytesWritable rowKeyWritable = new
ImmutableBytesWritable(Bytes.toBytes(rowKey));

        //初始化 put 对象
        Put put = new Put(Bytes.toBytes(rowKey));

        //参数分别:列族、列、值
        put.add(Bytes.toBytes("info"), Bytes.toBytes("name"),
Bytes.toBytes(name));
        put.add(Bytes.toBytes("info"), Bytes.toBytes("color"),
Bytes.toBytes(color));

        context.write(rowKeyWritable, put);
    }
}
```

2．构建 WriteFruitMRFromTxtReducer 类

```
package com.z.Hbase.mr2;

import java.io.IOException;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.mapreduce.TableReducer;
import org.apache.hadoop.io.NullWritable;
```

```
public class WriteFruitMRFromTxtReducer extends
TableReducer<ImmutableBytesWritable, Put, NullWritable> {
    @Override
    protected void reduce(ImmutableBytesWritable key, Iterable<Put>
values, Context context) throws IOException, InterruptedException
{
        //读出来的每一行数据写入到 fruit_hdfs 表中
        for(Put put: values){
            context.write(NullWritable.get(), put);
        }
    }
}
```

3．创建 Txt2FruitRunner 组装 Job

```
public int run(String[] args) throws Exception {
//得到 Configuration
Configuration conf = this.getConf();

//创建 Job 任务
Job job = Job.getInstance(conf, this.getClass().getSimpleName());
job.setJarByClass(Txt2FruitRunner.class);
Path inPath = new
Path("hdfs://hadoop102:9000/input_fruit/fruit.tsv");
FileInputFormat.addInputPath(job, inPath);

//设置 Mapper
job.setMapperClass(ReadFruitFromHDFSMapper.class);
job.setMapOutputKeyClass(ImmutableBytesWritable.class);
job.setMapOutputValueClass(Put.class);

//设置 Reducer
TableMapReduceUtil.initTableReducerJob("fruit_mr",
WriteFruitMRFromTxtReducer.class, job);

//设置 Reduce 数量，最少 1 个
job.setNumReduceTasks(1);

boolean isSuccess = job.waitForCompletion(true);
if(!isSuccess){
throw new IOException("Job running with error");
}

return isSuccess ? 0 : 1;
}
```

4．调用执行 Job

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

5．打包运行

```
$ /opt/module/hadoop-2.7.2/bin/yarn jar hbase-0.0.1-SNAPSHOT.jar
com.atguigu.hbase.mr2.Txt2FruitRunner
```

需求：将我们hdfs上面的这个路径/hbase/input/user.txt的数据文件，转换成HFile格式，表里面去

4.3.4 通过bulkload的方式批量加载数据到HBase

1.导入过程不占用Region资源

2.能快速导入海量的数据

3.节省内存　　：　优点

第一步：定义我们的mapper类

```
public class BulkLoadMapper extends Mapper<LongWritable,Text,ImmutableBytesWritable,Put> {
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String[] split = value.toString().split("\t");
        Put put = new Put(split[0].getBytes());
        put.addColumn("f1".getBytes(),"name".getBytes(),split[1].getBytes());
        put.addColumn("f1".getBytes(),"age".getBytes(),split[2].getBytes());
        context.write(new ImmutableBytesWritable(split[0].getBytes()),put);
    }
}
```

第二步：开发我们的main程序入口类

```
public class BulkLoadMain extends Configured implements Tool {
    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = super.getConf();
        Connection connection = ConnectionFactory.createConnection(conf);
        Table table = connection.getTable(TableName.valueOf("myuser2"));
        Job job = Job.getInstance(conf, "bulkLoad");
        //读取文件，解析成key,value对
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.addInputPath(job,new Path("hdfs://node01:8020/hbase/input"));
        //定义我们的mapper类
        job.setMapperClass(BulkLoadMapper.class);
        job.setMapOutputKeyClass(ImmutableBytesWritable.class);
        job.setMapOutputValueClass(Put.class);
        //reduce过程也省掉
        /**
         * Job job, Table table, RegionLocator regionLocator
         *  使用configureIncrementalLoad来进行配置我们的HFile加载到哪一个表里面的哪一个列族里面去
         */
        HFileOutputFormat2.configureIncrementalLoad(job,table,connection.getRegionLocator
(TableName.valueOf("myuser2")));
        //设置我们的输出类型，将我们的数据输出成为HFile格式
        job.setOutputFormatClass(HFileOutputFormat2.class);
        //设置我们的输出路径
        HFileOutputFormat2.setOutputPath(job,new Path("hdfs://node01:8020/hbase/hfile_out"));
        boolean b = job.waitForCompletion(true);
        return b?0:1;
    }
    public static void main(String[] args) throws Exception {
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");
        int run = ToolRunner.run(configuration, new BulkLoadMain(), args);
        System.exit(run);
    }
}
```

第三步：将代码打成jar包然后进行运行

```
yarn jar original-hbaseStudy-1.0-SNAPSHOT.jar  HBaseLoad
```

第四步：开发代码，加载数据

将我们的输出路径下面的HFile文件，加载到我们的hbase表当中去

```
-public class PutMain {
    public static void main(String[] args) throws IOException {
        Configuration conf = HBaseConfiguration.create();
        Connection connection = ConnectionFactory.createConnection(conf);

        Admin admin = connection.getAdmin();
        TableName tableName = TableName.valueOf("myuser2");
        Table table = connection.getTable(tableName);
        LoadIncrementalHFiles load = new LoadIncrementalHFiles(conf);
        load.doBulkLoad(new Path("hdfs://node01:
8020/hbase/hfile_out"),admin,table,connection.getRegionLocator(tableName));
    }
```

或者我们也可以通过命令行来进行加载数据

先将hbase的jar包添加到hadoop的classpath路径下

```
export HBASE_HOME=/export/servers/hbase-2.0.0/
export HADOOP_HOME=/export/servers/hadoop-2.7.5/
export HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`
```

然后执行以下命令，将hbase的HFile直接导入到表myuser2当中来

```
yarn jar bulkload-1.0-SNAPSHOT.jar PutMain /hbase/hfile_out myuser2
```

提示：运行任务前，如果待数据导入的表不存在，则需要提前创建之。
提示：maven 打包命令：-P local clean package 或-P dev clean package install（将第三方 jar 包一同打包，需要插件：maven-shade-plugin）

4.4 与 Hive 的集成

4.4.1 HBase 与 Hive 的对比

1．Hive
(1) 数据仓库
Hive 的本质其实就相当于将 HDFS 中已经存储的文件在 Mysql 中做了一个双射关系，以方便使用 HQL 去管理查询。
(2) 用于数据分析、清洗
Hive 适用于离线的数据分析和清洗，延迟较高。
(3) 基于 HDFS、MapReduce
Hive 存储的数据依旧在 DataNode 上，编写的 HQL 语句终将是转换为 MapReduce 代码执行。

2．HBase
(1) 数据库
是一种面向列族存储的非关系型数据库。
(2) 用于存储结构化和非结构化的数据
适用于单表非关系型数据的存储，不适合做关联查询，类似 JOIN 等操作。
(3) 基于 HDFS
数据持久化存储的体现形式是 HFile，存放于 DataNode 中，被 ResionServer 以 region 的形式进行管理。
(4) 延迟较低，接入在线业务使用
面对大量的企业数据，HBase 可以直线单表大量数据的存储，同时提供了高效的数据访问速度。

4.4.2 HBase 与 Hive 集成使用

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

环境准备

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

```
export HBASE_HOME=/opt/module/hbase
export HIVE_HOME=/opt/module/hive

ln            -s            $HBASE_HOME/lib/hbase-common-1.3.1.jar
$HIVE_HOME/lib/hbase-common-1.3.1.jar
ln -s $HBASE_HOME/lib/hbase-server-1.3.1.jar $HIVE_HOME/lib/hbase-
server-1.3.1.jar
ln -s $HBASE_HOME/lib/hbase-client-1.3.1.jar $HIVE_HOME/lib/hbase-
```

```
client-1.3.1.jar
ln           -s           $HBASE_HOME/lib/hbase-protocol-1.3.1.jar
$HIVE_HOME/lib/hbase-protocol-1.3.1.jar
ln  -s  $HBASE_HOME/lib/hbase-it-1.3.1.jar  $HIVE_HOME/lib/hbase-it-
1.3.1.jar
ln       -s       $HBASE_HOME/lib/htrace-core-3.1.0-incubating.jar
$HIVE_HOME/lib/htrace-core-3.1.0-incubating.jar
ln        -s        $HBASE_HOME/lib/hbase-hadoop2-compat-1.3.1.jar
$HIVE_HOME/lib/hbase-hadoop2-compat-1.3.1.jar
ln         -s         $HBASE_HOME/lib/hbase-hadoop-compat-1.3.1.jar
$HIVE_HOME/lib/hbase-hadoop-compat-1.3.1.jar
```

同时在 hive-site.xml 中修改 zookeeper 的属性，如下：

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

1．案例一

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

分步实现：

(1) 在 Hive 中创建表同时关联 HBase

```
CREATE TABLE hive_hbase_emp_table(
empno int,
ename string,
job string,
mgr int,
hiredate string,
sal double,
comm double,
deptno int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH       SERDEPROPERTIES       ("hbase.columns.mapping"       =
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:co
mm,info:deptno")
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

提示：完成之后，可以分别进入 Hive 和 HBase 查看，都生成了对应的表

(2) 在 Hive 中创建临时中间表，用于 load 文件中的数据

提示：不能将数据直接 load 进 Hive 所关联 HBase 的那张表中

```
CREATE TABLE emp(
empno int,
ename string,
job string,
mgr int,
```

```
hiredate string,
sal double,
comm double,
deptno int)
row format delimited fields terminated by '\t';
```

(3) 向 Hive 中间表中 load 数据

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

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

```
hive> insert into table hive_hbase_emp_table select * from emp;
```

(5) 查看 Hive 以及关联的 HBase 表中是否已经成功的同步插入了数据

Hive：

```
hive> select * from hive_hbase_emp_table;
```

HBase：

```
Hbase> scan ‘hbase_emp_table’
```

2．案例二

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

注：该案例 2 紧跟案例 1 的脚步，所以完成此案例前，请先完成案例 1。

分步实现：

(1) 在 Hive 中创建外部表

```
CREATE EXTERNAL TABLE relevance_hbase_emp(
empno int,
ename string,
job string,
mgr int,
hiredate string,
sal double,
comm double,
deptno int)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:co
mm,info:deptno")
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

(2) 关联后就可以使用 Hive 函数进行一些分析操作了

```
hive (default)> select * from relevance_hbase_emp;
```

第 5 章 HBase 优化

5.1 高可用

在 HBase 中 HMaster 负责监控 HRegionServer 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）

```
[atguigu@hadoop102 hbase]$ bin/stop-hbase.sh
```

2．在 conf 目录下创建 backup-masters 文件

```
[atguigu@hadoop102 hbase]$ touch conf/backup-masters
```

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

```
[atguigu@hadoop102 hbase]$ echo hadoop103 > conf/backup-masters
```

4．将整个 conf 目录 scp 到其他节点

```
[atguigu@hadoop102 hbase]$ scp -r conf/
hadoop103:/opt/module/hbase/
[atguigu@hadoop102 hbase]$ scp -r conf/
hadoop104:/opt/module/hbase/
```

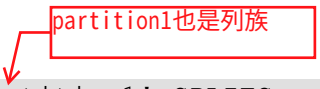
5．打开页面测试查看

http://hadooo102:16010

5.2 预分区

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

1．手动设定预分区




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

2．生成 16 进制序列预分区

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

3．按照文件中设置的规则预分区

创建 splits.txt 文件内容如下：

```
aaaa
bbbb
cccc
dddd
```

然后执行：

```
create 'staff3','partition3',SPLITS_FILE => 'splits.txt'
```

4．使用 JavaAPI 创建预分区

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

5.3 RowKey 设计

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

1．生成随机数、hash、散列值

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

5.4 内存优化

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

5.5 基础优化

1．允许在 HDFS 的文件中追加内容

hdfs-site.xml、hbase-site.xml

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

2．优化 DataNode 允许的最大文件打开数

hdfs-site.xml

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

3．优化延迟高的数据操作的等待时间

hdfs-site.xml

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

4．优化数据的写入效率

mapred-site.xml

属性：

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

5．设置 RPC 监听数量

hbase-site.xml

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

6．优化 HStore 文件大小

hbase-site.xml

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

7．优化 HBase 客户端缓存

hbase-site.xml

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

8．指定 scan.next 扫描 HBase 所获取的行数

hbase-site.xml

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

9．flush、compact、split 机制

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

涉及属性：

即：128M 就是 Memstore 的默认阈值

```
hbase.hregion.memstore.flush.size：134217728
```

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


```
hbase.regionserver.global.memstore.upperLimit：0.4
hbase.regionserver.global.memstore.lowerLimit：0.38
```

即：当 MemStore 使用内存总量达到 hbase.regionserver.global.memstore.upperLimit 指定值时，

将会有多个 MemStores flush 到文件中，MemStore flush 顺序是按照大小降序执行的，直到刷新到 MemStore 使用内存略小于 lowerLimit

第 6 章 HBase 实战之谷粒微博

6.1 需求分析

1) 微博内容的浏览，数据库表设计
2) 用户社交体现：关注用户，取关用户
3) 拉取关注的人的微博内容

6.2 代码实现

6.2.1 代码设计总览：

1) 创建命名空间以及表名的定义
2) 创建微博内容表
3) 创建用户关系表
4) 创建用户微博内容接收邮件表
5) 发布微博内容
6) 添加关注用户
7) 移除（取关）用户
8) 获取关注的人的微博内容
9) 测试

6.2.2 创建命名空间以及表名的定义

```
//获取配置 conf
private Configuration conf = HbaseConfiguration.create();

//微博内容表的表名
private      static      final      byte[]      TABLE_CONTENT      =
Bytes.toBytes("weibo:content");
//用户关系表的表名
private      static      final      byte[]      TABLE_RELATIONS      =
Bytes.toBytes("weibo:relations");
//微博收件箱表的表名
private   static   final   byte[]   TABLE_RECEIVE_CONTENT_EMAIL   =
Bytes.toBytes("weibo:receive_content_email");
public void initNamespace(){
    HbaseAdmin admin = null;
    try {
        admin = new HbaseAdmin(conf);
        //命名空间类似于关系型数据库中的 schema，可以想象成文件夹
        NamespaceDescriptor weibo = NamespaceDescriptor
                .create("weibo")
                .addConfiguration("creator", "Jinji")
                .addConfiguration("create_time",
System.currentTimeMillis() + "")
                .build();
```

```
            admin.createNamespace(weibo);
        } catch (MasterNotRunningException e) {
            e.printStackTrace();
        } catch (ZooKeeperConnectionException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            if(null != admin){
                try {
                    admin.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
}
```

6.2.3 创建微博内容表

表结构：

方法名	creatTableeContent
Table Name	weibo:content
RowKey	用户 ID_时间戳
ColumnFamily	info
ColumnLabel	标题,内容,图片
Version	1 个版本

代码：

```
/**
 * 创建微博内容表
 * Table Name:weibo:content
 * RowKey:用户 ID_时间戳
 * ColumnFamily:info
 * ColumnLabel:标题  内容       图片 URL
 * Version:1 个版本
 */
public void createTableContent(){
    HbaseAdmin admin = null;
    try {
        admin = new HbaseAdmin(conf);
        //创建表表述
        HTableDescriptor          content          =          new
HTableDescriptor(TableName.valueOf(TABLE_CONTENT));
        //创建列族描述
        HColumnDescriptor           info           =          new
HColumnDescriptor(Bytes.toBytes("info"));
        //设置块缓存
        info.setBlockCacheEnabled(true);
        //设置块缓存大小
        info.setBlocksize(2097152);
        //设置压缩方式
//        info.setCompressionType(Algorithm.SNAPPY);
        //设置版本确界
        info.setMaxVersions(1);
```

```
        info.setMinVersions(1);

        content.addFamily(info);
        admin.createTable(content);

    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

6.2.4 创建用户关系表

表结构：

方法名	createTableRelations
Table Name	weibo:relations
RowKey	用户 ID
ColumnFamily	attends、fans
ColumnLabel	关注用户 ID，粉丝用户 ID
ColumnValue	用户 ID
Version	1 个版本

代码：

```
/**
 * 用户关系表
 * Table Name:weibo:relations0
 * RowKey:用户 ID
 * ColumnFamily:attends,fans
 * ColumnLabel:关注用户 ID，粉丝用户 ID
 * ColumnValue:用户 ID
 * Version: 1 个版本
 */
public void createTableRelations(){
    HbaseAdmin admin = null;
    try {
        admin = new HbaseAdmin(conf);
        HTableDescriptor          relations          =          new
HTableDescriptor(TableName.valueOf(TABLE_RELATIONS));

        //关注的人的列族
        HColumnDescriptor           attends           =           new
HColumnDescriptor(Bytes.toBytes("attends"));
        //设置块缓存
        attends.setBlockCacheEnabled(true);
```

```
        //设置块缓存大小
        attends.setBlocksize(2097152);
        //设置压缩方式
//          info.setCompressionType(Algorithm.SNAPPY);
        //设置版本确界
        attends.setMaxVersions(1);
        attends.setMinVersions(1);

        //粉丝列族
        HColumnDescriptor            fans            =            new
HColumnDescriptor(Bytes.toBytes("fans"));
        fans.setBlockCacheEnabled(true);
        fans.setBlocksize(2097152);
        fans.setMaxVersions(1);
        fans.setMinVersions(1);


        relations.addFamily(attends);
        relations.addFamily(fans);
        admin.createTable(relations);

    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

6.2.5 创建微博收件箱表

表结构：

方法名	createTableReceiveContentEmails
Table Name	weibo:receive_content_email
RowKey	用户 ID
ColumnFamily	info
ColumnLabel	用户 ID
ColumnValue	取微博内容的 RowKey
Version	1000

代码：

```
/**
 * 创建微博收件箱表
 * Table Name: weibo:receive_content_email
 * RowKey:用户 ID
 * ColumnFamily:info
```

```
 * ColumnLabel:用户 ID-发布微博的人的用户 ID
 * ColumnValue:关注的人的微博的 RowKey
 * Version:1000
 */
public void createTableReceiveContentEmail(){
    HbaseAdmin admin = null;
    try {
        admin = new HbaseAdmin(conf);
        HTableDescriptor      receive_content_email      =      new
HTableDescriptor(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL));
        HColumnDescriptor           info           =           new
HColumnDescriptor(Bytes.toBytes("info"));

        info.setBlockCacheEnabled(true);
        info.setBlocksize(2097152);
        info.setMaxVersions(1000);
        info.setMinVersions(1000);

        receive_content_email.addFamily(info);;
        admin.createTable(receive_content_email);
    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

6.2.6 发布微博内容

a、微博内容表中添加 1 条数据

b、微博收件箱表对所有粉丝用户添加数据

代码：Message.java

```
package com.atguigu.weibo;

public class Message {
    private String uid;
    private String timestamp;
    private String content;

    public String getUid() {
        return uid;
    }
    public void setUid(String uid) {
        this.uid = uid;
    }
    public String getTimestamp() {
        return timestamp;
```

```
    }
    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
    @Override
    public String toString() {
        return "Message [uid=" + uid + ", timestamp=" + timestamp +
", content=" + content + "]";
    }
}
```

代码：public void publishContent(String uid, String content)

```
/**
 * 发布微博
 * a、微博内容表中数据+1
 * b、向微博收件箱表中加入微博的 Rowkey
 */
public void publishContent(String uid, String content){
    HConnection connection = null;
    try {
        connection = HConnectionManager.createConnection(conf);
        //a、微博内容表中添加 1 条数据，首先获取微博内容表描述
        HTableInterface                contentTBL                =
connection.getTable(TableName.valueOf(TABLE_CONTENT));
        //组装 Rowkey
        long timestamp = System.currentTimeMillis();
        String rowKey = uid + "_" + timestamp;

        Put put = new Put(Bytes.toBytes(rowKey));
        put.add(Bytes.toBytes("info"),    Bytes.toBytes("content"),
timestamp, Bytes.toBytes(content));

        contentTBL.put(put);

        //b、向微博收件箱表中加入发布的 Rowkey
        //b.1、查询用户关系表，得到当前用户有哪些粉丝
        HTableInterface                relationsTBL                =
connection.getTable(TableName.valueOf(TABLE_RELATIONS));
        //b.2、取出目标数据
        Get get = new Get(Bytes.toBytes(uid));
        get.addFamily(Bytes.toBytes("fans"));

        Result result = relationsTBL.get(get);
        List<byte[]> fans = new ArrayList<byte[]>();

        //遍历取出当前发布微博的用户的所有粉丝数据
        for(Cell cell : result.rawCells()){
            fans.add(CellUtil.cloneQualifier(cell));
        }
        //如果该用户没有粉丝，则直接 return
        if(fans.size() <= 0) return;
```

```
        //开始操作收件箱表
        HTableInterface                    recTBL                    =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));
        List<Put> puts = new ArrayList<Put>();
        for(byte[] fan : fans){
            Put fanPut = new Put(fan);
            fanPut.add(Bytes.toBytes("info"),    Bytes.toBytes(uid),
timestamp, Bytes.toBytes(rowKey));
            puts.add(fanPut);
        }
        recTBL.put(puts);
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != connection){
            try {
                connection.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

6.2.7 添加关注用户

a、在微博用户关系表中，对当前主动操作的用户添加新关注的好友

b、在微博用户关系表中，对被关注的用户添加新的粉丝

c、微博收件箱表中添加所关注的用户发布的微博

代码实现：public void addAttends(String uid, String... attends)

```
/**
 * 关注用户逻辑
 * a、在微博用户关系表中，对当前主动操作的用户添加新的关注的好友
 * b、在微博用户关系表中，对被关注的用户添加粉丝（当前操作的用户）
 * c、当前操作用户的微博收件箱添加所关注的用户发布的微博 rowkey
 */
public void addAttends(String uid, String... attends){
    //参数过滤
    if(attends == null || attends.length <= 0 || uid == null ||
uid.length() <= 0){
        return;
    }
    HConnection connection = null;
    try {
        connection = HConnectionManager.createConnection(conf);
        //用户关系表操作对象（连接到用户关系表）
        HTableInterface                relationsTBL                =
connection.getTable(TableName.valueOf(TABLE_RELATIONS));
        List<Put> puts = new ArrayList<Put>();
        //a、在微博用户关系表中，添加新关注的好友
        Put attendPut = new Put(Bytes.toBytes(uid));
        for(String attend : attends){
            //为当前用户添加关注的人
            attendPut.add(Bytes.toBytes("attends"),
```

```
Bytes.toBytes(attend), Bytes.toBytes(attend));
            //b、为被关注的人，添加粉丝
            Put fansPut = new Put(Bytes.toBytes(attend));
            fansPut.add(Bytes.toBytes("fans"),   Bytes.toBytes(uid),
Bytes.toBytes(uid));
            //将所有关注的人一个一个的添加到 puts（List）集合中
            puts.add(fansPut);
        }
        puts.add(attendPut);
        relationsTBL.put(puts);

        //c.1、微博收件箱添加关注的用户发布的微博内容（content）的 rowkey
        HTableInterface                  contentTBL                  =
connection.getTable(TableName.valueOf(TABLE_CONTENT));
        Scan scan = new Scan();
        //用于存放取出来的关注的人所发布的微博的 rowkey
        List<byte[]> rowkeys = new ArrayList<byte[]>();

        for(String attend : attends){
            //过滤扫描 rowkey，即：前置位匹配被关注的人的 uid_
            RowFilter            filter            =            new
RowFilter(CompareFilter.CompareOp.EQUAL,                        new
SubstringComparator(attend + "_"));
            //为扫描对象指定过滤规则
            scan.setFilter(filter);
            //通过扫描对象得到 scanner
            ResultScanner result = contentTBL.getScanner(scan);
            //迭代器遍历扫描出来的结果集
            Iterator<Result> iterator = result.iterator();
            while(iterator.hasNext()){
                //取出每一个符合扫描结果的那一行数据
                Result r = iterator.next();
                for(Cell cell : r.rawCells()){
                    //将得到的 rowkey 放置于集合容器中
                    rowkeys.add(CellUtil.cloneRow(cell));
                }

            }
        }
```

//c.2、将取出的微博 rowkey 放置于当前操作用户的收件箱中

```
        if(rowkeys.size() <= 0) return;
        //得到微博收件箱表的操作对象
        HTableInterface                    recTBL                    =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));
        //用于存放多个关注的用户的发布的多条微博 rowkey 信息
        List<Put> recPuts = new ArrayList<Put>();
        for(byte[] rk : rowkeys){
            Put put = new Put(Bytes.toBytes(uid));
            //uid_timestamp
            String rowKey = Bytes.toString(rk);
            //借取 uid
            String       attendUID       =       rowKey.substring(0,
rowKey.indexOf("_"));
```

```
            long                        timestamp                       =
Long.parseLong(rowKey.substring(rowKey.indexOf("_") + 1));
            //将微博 rowkey 添加到指定单元格中
            put.add(Bytes.toBytes("info"), Bytes.toBytes(attendUID),
timestamp, rk);
            recPuts.add(put);
        }

        recTBL.put(recPuts);

    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != connection){
            try {
                connection.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

6.2.8 移除（取关）用户

a、在微博用户关系表中，对当前主动操作的用户移除取关的好友(attends)

b、在微博用户关系表中，对被取关的用户移除粉丝

c、微博收件箱中删除取关的用户发布的微博

代码：public void removeAttends(String uid, String... attends)

```
/**
 * 取消关注（remove）
 * a、在微博用户关系表中，对当前主动操作的用户删除对应取关的好友
 * b、在微博用户关系表中，对被取消关注的人删除粉丝（当前操作人）
 * c、从收件箱中，删除取关的人的微博的 rowkey
 */
public void removeAttends(String uid, String... attends){
    //过滤数据
    if(uid == null || uid.length() <= 0 || attends == null ||
attends.length <= 0) return;
    HConnection connection = null;

    try {
        connection = HConnectionManager.createConnection(conf);
        //a、在微博用户关系表中，删除已关注的好友
        HTableInterface                relationsTBL                =
connection.getTable(TableName.valueOf(TABLE_RELATIONS));

        //待删除的用户关系表中的所有数据
        List<Delete> deletes = new ArrayList<Delete>();
        //当前取关操作者的 uid 对应的 Delete 对象
        Delete attendDelete = new Delete(Bytes.toBytes(uid));
        //遍历取关，同时每次取关都要将被取关的人的粉丝-1
        for(String attend : attends){
            attendDelete.deleteColumn(Bytes.toBytes("attends"),
```

```
Bytes.toBytes(attend));
            //b
            Delete fansDelete = new Delete(Bytes.toBytes(attend));
            fansDelete.deleteColumn(Bytes.toBytes("fans"),
Bytes.toBytes(uid));
            deletes.add(fansDelete);
        }

        deletes.add(attendDelete);
        relationsTBL.delete(deletes);

        //c、删除取关的人的微博 rowkey 从 收件箱表中
        HTableInterface                    recTBL                    =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));

        Delete recDelete = new Delete(Bytes.toBytes(uid));
        for(String attend : attends){
            recDelete.deleteColumn(Bytes.toBytes("info"),
Bytes.toBytes(attend));
        }
        recTBL.delete(recDelete);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

6.2.9 获取关注的人的微博内容

a、从微博收件箱中获取所关注的用户的微博 RowKey

b、根据获取的 RowKey，得到微博内容

代码实现：public List<Message> getAttendsContent(String uid)

```
/**
 * 获取微博实际内容
 * a、从微博收件箱中获取所有关注的人的发布的微博的 rowkey
 * b、根据得到的 rowkey 去微博内容表中得到数据
 * c、将得到的数据封装到 Message 对象中
 */
public List<Message> getAttendsContent(String uid){
    HConnection connection = null;
    try {
        connection = HConnectionManager.createConnection(conf);
        HTableInterface                    recTBL                    =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));
        //a、从收件箱中取得微博 rowKey
        Get get = new Get(Bytes.toBytes(uid));
        //设置最大版本号
        get.setMaxVersions(5);
        List<byte[]> rowkeys = new ArrayList<byte[]>();
        Result result = recTBL.get(get);
        for(Cell cell : result.rawCells()){
            rowkeys.add(CellUtil.cloneValue(cell));
        }
        //b、根据取出的所有 rowkey 去微博内容表中检索数据
        HTableInterface                    contentTBL                =
```

```
connection.getTable(TableName.valueOf(TABLE_CONTENT));
        List<Get> gets = new ArrayList<Get>();
        //根据 rowkey 取出对应微博的具体内容
        for(byte[] rk : rowkeys){
            Get g = new Get(rk);
            gets.add(g);
        }
        //得到所有的微博内容的 result 对象
        Result[] results = contentTBL.get(gets);

        List<Message> messages = new ArrayList<Message>();
        for(Result res : results){
            for(Cell cell : res.rawCells()){
                Message message = new Message();

                String                    rowKey                    =
Bytes.toString(CellUtil.cloneRow(cell));
                String       userid       =       rowKey.substring(0,
rowKey.indexOf("_"));
                String                   timestamp                  =
rowKey.substring(rowKey.indexOf("_") + 1);
                String                    content                   =
Bytes.toString(CellUtil.cloneValue(cell));

                message.setContent(content);
                message.setTimestamp(timestamp);
                message.setUid(userid);

                messages.add(message);
            }
        }
        return messages;
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        try {
            connection.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return null;
}
```

6.2.10 测试

-- 测试发布微博内容

public void testPublishContent(WeiBo wb)

-- 测试添加关注

public void testAddAttend(WeiBo wb)

-- 测试取消关注

public void testRemoveAttend(WeiBo wb)

-- 测试展示内容

public void testShowMessage(WeiBo wb)

代码：

```
/**
 * 发布微博内容
 * 添加关注
 * 取消关注
 * 展示内容
 */
public void testPublishContent(WeiBo wb){
    wb.publishContent("0001", "今天买了一包空气，送了点薯片，非常开心!!");
    wb.publishContent("0001", "今天天气不错。");
}

public void testAddAttend(WeiBo wb){
    wb.publishContent("0008", "准备下课！");
    wb.publishContent("0009", "准备关机！");
    wb.addAttends("0001", "0008", "0009");
}

public void testRemoveAttend(WeiBo wb){
    wb.removeAttends("0001", "0008");
}

public void testShowMessage(WeiBo wb){
    List<Message> messages = wb.getAttendsContent("0001");
    for(Message message : messages){
        System.out.println(message);
    }
}

public static void main(String[] args) {
    WeiBo weibo = new WeiBo();
    weibo.initTable();

    weibo.testPublishContent(weibo);
    weibo.testAddAttend(weibo);
    weibo.testShowMessage(weibo);
    weibo.testRemoveAttend(weibo);
    weibo.testShowMessage(weibo);
}
```