尚硅谷大数据技术之 HBase

官网：www.atguigu.com

日期	修订版本	修改章节	修改描述	作者
2017-09-11	1.0		内部稿	尽际
2017-09-30	1.1	章节结构变动	内部稿	尽际
2017-10-01	1.2	增加更多案例	内部稿	尽际
2017-10-07	1.3	修改 API 说明	内部稿	尽际
2017-10-08	1.3	修缮部分错误、bug 调改	内部稿	尽际
2017-10-10	1.4	更新 API	内部稿	尽际

ShangGuigu Technologies Co., Ltd.

尚硅谷技术有限公司

一、HBaes 介绍

1.1、HBase 的起源

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

官方网站：http://hbase.apache.org

-- 2006 年 Google 发表 BigTable 白皮书

-- 2006 年开始开发 HBase

-- 2008 年北京成功开奥运会，程序员默默地将 HBase 弄成了 Hadoop 的子项目

-- 2010 年 HBase 成为 Apache 顶级项目

-- 现在很多公司二次开发出了很多发行版本，你也开始使用了。

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、RegionServer

功能：

1) 负责存储 HBase 的实际数据

2) 处理分配给它的 Region

3) 刷新缓存到 HDFS

4) 维护 HLog

5) 执行压缩

6) 负责处理 Region 分片

组件：

1) Write-Ahead logs

HBase 的修改记录，当对 HBase 读写数据的时候，数据不是直接写进磁盘，它会在内存中保留一段时间（时间以及数据量阈值可以设定）。但把数据保存在内存中可能有更高的概率引起数据丢失，为了解决这个问题，数据会先写在一个叫做 Write-Ahead logfile 的文件中，然后再写入内存中。所以在系统出现故障的时候，数据可以通过这个日志文件重建。

2) HFile

这是在磁盘上保存原始数据的实际的物理文件，是实际的存储文件。

3) Store

HFile 存储在 Store 中，一个 Store 对应 HBase 表中的一个列族。

4) MemStore

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

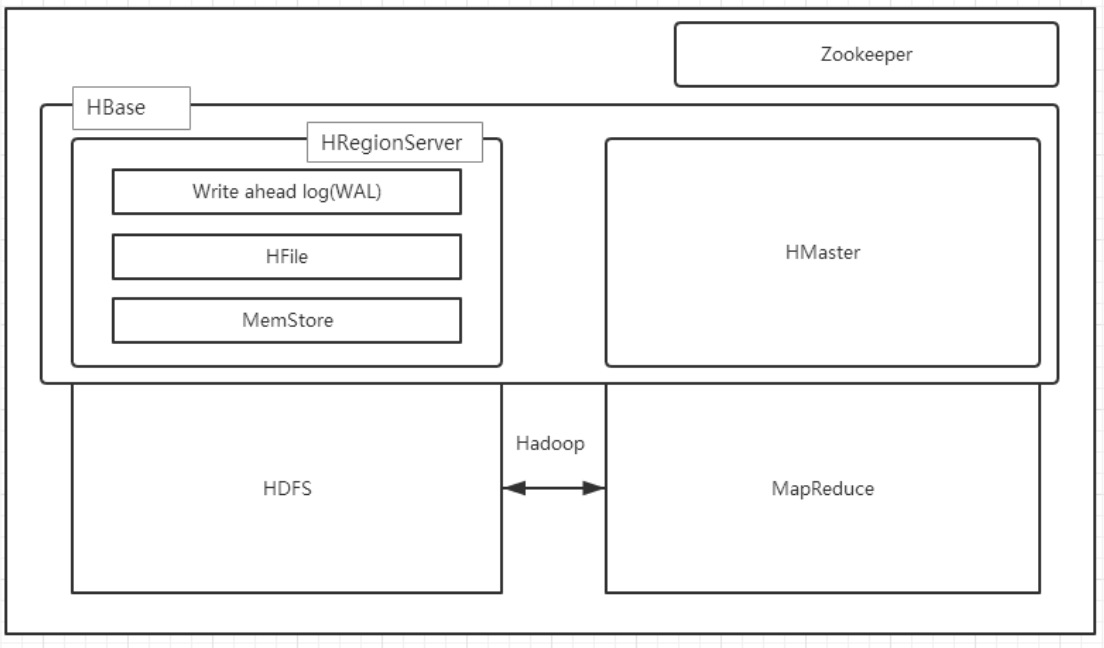
5) Region

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

1.3、HBase 的架构

HBase 一种是作为存储的分布式文件系统，另一种是作为数据处理模型的 MR 框架。因为日常开发人员比较熟练的是结构化的数据进行处理，但是在 HDFS 直接存储的文件往往不具有结构化，所以催生出了 HBase 在 HDFS 上的操作。如果需要查询数据，只需要通过键值便可以成功访问。

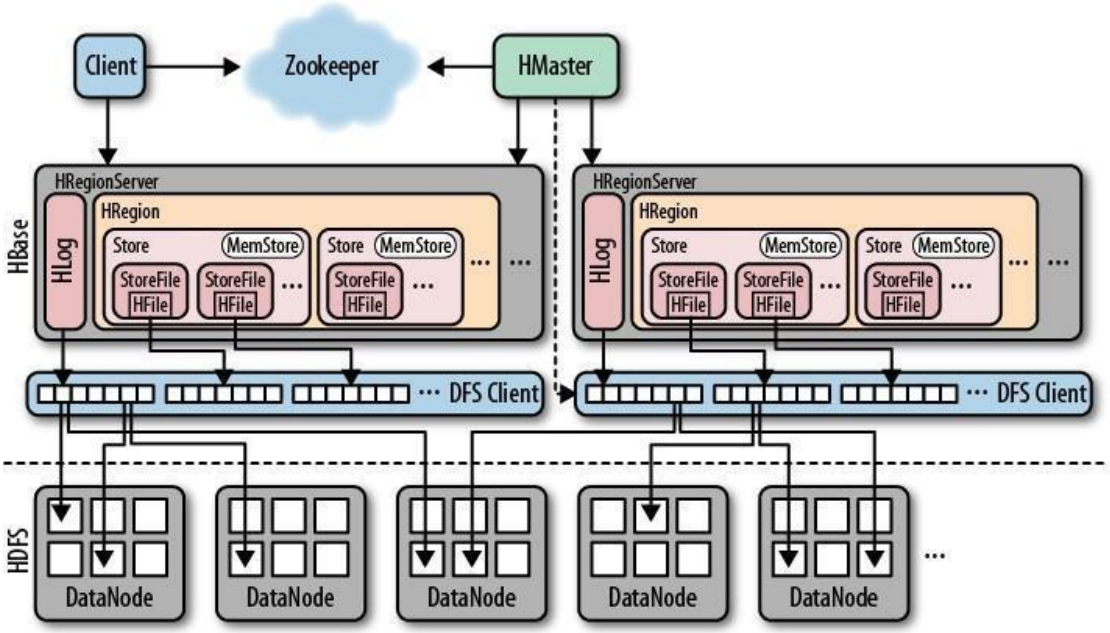
架构图如下图所示：



HBase 内置有 Zookeeper，但一般我们会有其他的 Zookeeper 集群来监管 master 和 regionserver，Zookeeper 通过选举，保证任何时候，集群中只有一个活跃的 HMaster，HMaster 与 HRegionServer 启动时会向 ZooKeeper 注册，存储所有 HRegion 的寻址入口，实时监控 HRegionserver 的上线和下线信息。并实时通知给 HMaster，存储 HBase 的 schema 和 table 元数据，默认情况下，HBase 管理 ZooKeeper 实例，Zookeeper 的引入使得 HMaster 不再是单点故障。一般情况下会启动两个 HMaster，非 Active 的 HMaster 会定期的和 Active HMaster 通信以获取其最新状态，从而保证它是实时更新的，因而如果启动了多个 HMaster 反而增加了 Active HMaster 的负担。

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：

在进行Region切分的时候，Store、HLog、Memstore、StoreFile都要进行切分



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：

```
$ ~/modules/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：

```
$ ~/modules/hadoop-2.7.2/sbin/start-dfs.sh

$ ~/modules/hadoop-2.7.2/sbin/start-yarn.sh
```

2.1.3、HBase 的解压

解压 HBase 到指定目录：

```
$ tar -zxf ~/softwares/installations/hbase-1.3.1-bin.tar.gz -C ~/modules/
```

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=/home/admin/modules/jdk1.8.0_121

export HBASE_MANAGES_ZK=false
```

hbase-site.xml 修改内容：

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://linux01:8020/hbase</value>
    </property>

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

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

    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>linux01:2181,linux02:2181,linux03:2181</value>
    </property>

    <property>
        <name>hbase.zookeeper.property.dataDir</name>
     <value>/home/admin/modules/zookeeper-3.4.5/zkData</value>
    </property>
</configuration>
```

regionservers：

```
linux01

linux02

linux03
```

2.1.5、HBase 需要依赖的 Jar 包

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 /home/admin/modules/hbase-1.3.1/lib/hadoop-*
$ rm -rf /home/admin/modules/hbase-1.3.1/lib/zookeeper-3.4.6.jar
```

2) 拷贝新 jar，涉及的 jar 有：

```
hadoop-annotations-2.7.2.jar
hadoop-auth-2.7.2.jar
hadoop-client-2.7.2.jar
hadoop-common-2.7.2.jar
hadoop-hdfs-2.7.2.jar
hadoop-mapreduce-client-app-2.7.2.jar
hadoop-mapreduce-client-common-2.7.2.jar
hadoop-mapreduce-client-core-2.7.2.jar
hadoop-mapreduce-client-hs-2.7.2.jar
hadoop-mapreduce-client-hs-plugins-2.7.2.jar
hadoop-mapreduce-client-jobclient-2.7.2.jar
hadoop-mapreduce-client-jobclient-2.7.2-tests.jar
hadoop-mapreduce-client-shuffle-2.7.2.jar
hadoop-yarn-api-2.7.2.jar
hadoop-yarn-applications-distributedshell-2.7.2.jar
hadoop-yarn-applications-unmanaged-am-launcher-2.7.2.jar
hadoop-yarn-client-2.7.2.jar
hadoop-yarn-common-2.7.2.jar
hadoop-yarn-server-applicationhistoryservice-2.7.2.jar
hadoop-yarn-server-common-2.7.2.jar
hadoop-yarn-server-nodemanager-2.7.2.jar
hadoop-yarn-server-resourcemanager-2.7.2.jar
hadoop-yarn-server-tests-2.7.2.jar
```

```
hadoop-yarn-server-web-proxy-2.7.2.jar
zookeeper-3.4.5.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 /home/admin/modules/hadoop-2.7.2/ -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

```
$ ln -s ~/modules/hadoop-2.7.2/etc/hadoop/core-site.xml
~/modules/hbase-1.3.1/conf/core-site.xml
$ ln -s ~/modules/hadoop-2.7.2/etc/hadoop/hdfs-site.xml
~/modules/hbase-1.3.1/conf/hdfs-site.xml
```

2.1.7、HBase 远程 scp 到其他集群

```
$ scp -r /home/admin/modules/hbase-1.3.1/ linux02:/home/admin/modules/
$ scp -r /home/admin/modules/hbase-1.3.1/ linux03:/home/admin/modules/
```

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。

修复提示：

a、同步时间服务

请参看帮助文档：《大数据帮助文档 1.0》

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>
```

启动方式 2：

```
$ bin/start-hbase.sh
```

对应的停止服务：

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

尖叫提示：如果使用的是 JDK8 以上版本，则应在 hbase-evn.sh 中移除“HBASE_MASTER_OPTS”和“HBASE_REGIONSERVER_OPTS”配置。

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://linux01:16010
```

2.2、简单使用

2.2.1、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
```

2.2.2、表的操作

1) 创建表

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

2) 插入数据到表

```
hbase(main) > put 'student','1001','info:name','Thomas'
hbase(main) > put 'student','1001','info:sex','male'
hbase(main) > put 'student','1001','info:age','18'
hbase(main) > put 'student','1002','info:name','Janna'
hbase(main) > put 'student','1002','info:sex','female'
hbase(main) > put 'student','1002','info:age','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'}
```

4) 查看表结构

```
hbase(main):012:0> describe ‘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'
```

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'
```

删除某 rowkey 的某一列数据：

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

8) 清空表数据 本质是先删除表，然后再创建一个空表

```
hbase(main) > truncate 'student'
```

尖叫提示：清空表的操作顺序为先 disable，然后再 truncating。

9) 删除表

首先需要先让该表为 disable 状态：

```
hbase(main) > disable '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}
```

2.3、读写流程 HBase自身框架对比，写要比读快

2.3.1、HBase 读数据流程 在整个读过程中没有用到HMaster

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

2) 接着 Client 通过刚才获取到的 HRegionServer 的 IP 来访问 Meta 表所在的

HBase版本：老版本<--0.98-->新版本

访问时为什么先访问内存？因为新的数据都在内存中

HRegionServer，从而读取到 Meta，进而获取到 Meta 表中存放的元数据。 即具体数据存放在哪个HRegionServer上

3) Client 通过元数据中存储的信息，访问对应的 HRegionServer，然后扫描所在 HRegionServer 的 Memstore 和 Storefile 来查询数据。 先内存Memstore，后缓存BlockCache，再磁盘StoreFile,访问磁盘时，会把查询到的数据先写到缓存BlockCache中，再返回给客户端。

4) 最后 HRegionServer 把查询到的数据响应给 Client。

2.3.2、HBase 写数据流程

HMaster短时间挂掉，不影响读写过程，但时间长了会有危险，会造成数据倾斜，因为Region切分与合成都是由HMaster触发请求，然后具体由HRegionServer去执行完成的

1) Client 也是先访问 zookeeper，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。

此时即可向客户端返回写入成功的信息

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。 Region切分的时候，也要对元数据表meta进行更新，更新过程由HMaster完成

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、安装 Maven 并配置环境变量

```
$ tar -zxf ~/softwares/installations/apache-maven-3.5.0-bin.tar.gz -C ~/modules/
```

在环境变量中添加：

```
MAVEN_HOME=/home/admin/modules/apache-maven-3.5.0

export PATH=$PATH:$MAVEN_HOME/bin
```

2.4.2、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>

<dependency>
        <groupId>jdk.tools</groupId>
        <artifactId>jdk.tools</artifactId>
        <version>1.6</version>
        <scope>system</scope>
        <systemPath>${JAVA_HOME}/lib/tools.jar</systemPath>
</dependency>
```

2.4.3、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", "192.168.216.20");
        conf.set("hbase.zookeeper.property.clientPort", "2181");
}
```

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 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！");
        }
}
```

5) 向表中插入数据

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

```
        hTable.put(put);
        hTable.close();
        System.out.println("插入数据成功");
}
```

6) 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)));
        }
}
```

2.5、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。

2.5.1、官方 HBase-MapReduce

1) 查看 HBase 的 MapReduce 任务的执行

```
$ bin/hbase mapredcp
```

2) 执行环境变量的导入

```
$ export HBASE_HOME=/home/admin/modules/hbase-1.3.1
$ export HADOOP_HOME=/home/admin/modules/hadoop-2.7.2
$ 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

```
$ ~/modules/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
```

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

```
$ ~/modules/hadoop-2.7.2/bin/hdfs dfs -mkdir /input_fruit/
$ ~/modules/hadoop-2.7.2/bin/hdfs dfs -put fruit.tsv /input_fruit/
```

(4) 执行 MapReduce 到 HBase 的 fruit 表中

```
$ ~/modules/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://linux01:8020/input_fruit
```

(5) 使用 scan 命令查看导入后的结果

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

2.5.2、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.z.hbase_mr;


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.z.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

```
$ ~/modules/hadoop-2.7.2/bin/yarn jar ~/softwares/jars/hbase-0.0.1-SNAPSHOT.jar
com.z.hbase.mr1.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。

尖叫提示：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）

2.5.3、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.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.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://linux01:8020/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

```
$ ~/modules/hadoop-2.7.2/bin/yarn jar ~/softwares/jars/hbase-0.0.1-SNAPSHOT.jar
com.z.hbase.mr2.Txt2FruitRunner
```

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。

尖叫提示：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）

2.6、与 Hive 的集成

2.6.1、HBase 与 Hive 的对比

1) Hive 分析

(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 存储

(1) 数据库

是一种面向列存储的非关系型数据库。

(2) 用于存储结构化和非结构话的数据

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。

2.6.2、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=/home/admin/modules/hbase-1.3.1
$ export HIVE_HOME=/home/admin/modules/apache-hive-1.2.2-bin

$ 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>linux01,linux02,linux03</value>
    <description>The list of ZooKeeper servers to talk to. This is only needed for read/write
locks.</description>
</property>
<property>
    <name>hive.zookeeper.client.port</name>
    <value>2181</value>
    <description>The port of ZooKeeper servers to talk to. This is only needed for read/write
locks.</description>
</property>
```

1) 案例一

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

分步实现：

(1) 在 Hive 中创建表同时关联 HBase

```
CREATE TABLE hive_hbase_emp_table(
empno int,
ename string,
job string,
mgr int,
hiredate string,
sal double,
comm double,
deptno int)
```

```
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

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 '/home/admin/softwares/data/emp.txt' into table emp;
```

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

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

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

Hive：

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

HBase：

```
hbase> scan ‘hbase_emp_table’
```

2) 案例二

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

注：该案例 2 紧跟案例 1 的脚步，所以完成此案例前，请先完成案例 1。

分步实现：

(1) 在 Hive 中创建外部表

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

(2) 关联后就可以使用 Hive 函数进行一些分析操作了

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

2.7、与 Sqoop 的集成

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=/home/admin/modules/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

```
$ bin/sqoop import \
--connect jdbc:mysql://linux01:3306/db_library \
--username root \
--password 123456 \
--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

解决方案：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。

2.8、常用的 Shell 操作

1) satus

例如：显示服务器状态

```
hbase> status ‘linux01’
```

2) whoami

显示 HBase 当前用户，例如：

```
hbase> whoami
```

3) list

显示当前所有的表

```
hbase> list
```

4) count

统计指定表的记录数，例如：

```
hbase> count 'hbase_book'
```

5) describe

展示表结构信息

```
hbase> describe 'hbase_book'
```

6) exist

检查表是否存在，适用于表量特别多的情况

```
hbase> exist 'hbase_book'
```

7) 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'
```

10) drop

删除一张表，记得在删除表之前必须先禁用

```
hbase> drop 'hbase_book'
```

11) delete

删除一行中一个单元格的值，例如：

```
hbase> delete 'hbase_book', 'rowKey', 'CF:C'
```

12) truncate

清空表数据，即禁用表-删除表-创建表

```
hbase> truncate 'hbase_book'
```

13) create

创建表，例如：

```
hbase> create 'table', 'cf'
```

创建多个列族：

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

2.9、数据的备份与恢复

2.9.1、备份

停止 HBase 服务后，使用 distcp 命令运行 MapReduce 任务进行备份，将数据备份到另一个地方，可以是同一个集群，也可以是专用的备份集群。

即，把数据转移到当前集群的其他目录下（也可以不在同一个集群中）：

```
$ bin/hadoop distcp \
hdfs://linux01:8020/hbase \
hdfs://linux01:8020/HbaseBackup/backup20171009
```

尖叫提示：执行该操作，一定要开启 Yarn 服务

2.9.2、恢复

非常简单，与备份方法一样，将数据整个移动回来即可。

```
$ bin/hadoop distcp \
hdfs://linux01:8020/HbaseBackup/backup20170930 \
hdfs://linux01:8020/hbase
```

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。

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：

1) 停止负载平衡器

```
hbase> balance_switch false
```

2) 在退役节点上停止 RegionServer

```
hbase> 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

6) RegionServer 的 region 服务得到重新分配

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) RegionServer 先卸载所管理的 region

```
$ bin/graceful_stop.sh <RegionServer-hostname>
```

2) 自动平衡数据

3) 和之前的 2~6 步是一样的

2.11、版本的确界

1) 版本的下界

默认的版本下界是 0，即禁用。row 版本使用的最小数目是与生存时间（TTL Time To Live）相结合的，并且我们根据实际需求可以有 0 或更多的版本，使用 0，即只有 1 个版本的值写入 cell。

2) 版本的上界

之前默认的版本上界是 3，也就是一个 row 保留 3 个副本（基于时间戳的插入）。该值不要设计的过大，一般的业务不会超过 100。如果 cell 中存储的数据版本号超过了 3 个，再次插入数据时，最新的值会将最老的值覆盖。（现版本已默认为 1）

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 linux02 > conf/backup-masters
```

4) 将整个 conf 目录 scp 到其他节点

```
$ scp -r conf/ linux02:/opt/modules/cdh/hbase-0.98.6-cdh5.3.6/
$ scp -r conf/ linux03:/opt/modules/cdh/hbase-0.98.6-cdh5.3.6/
```

5) 打开页面测试查看

0.98 版本之前：http://linux01:60010

0.98 版本之后：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，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 => '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/

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。

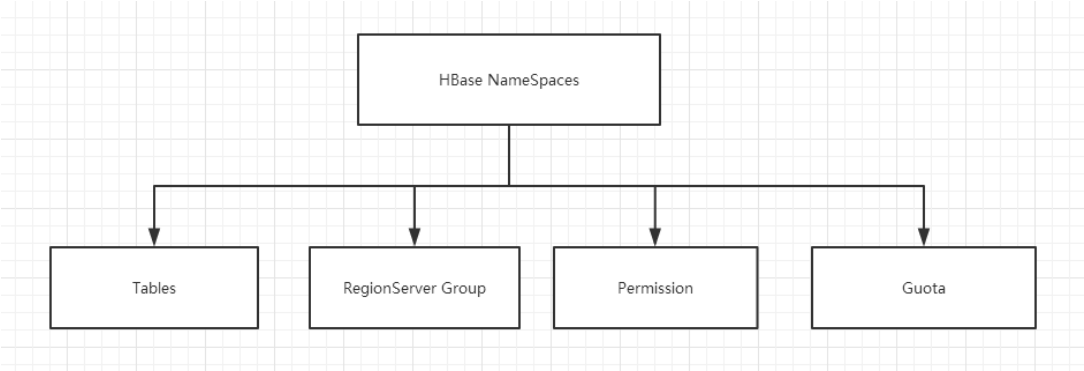
hbase.regionserver.global.memstore.upperLimit：0.4 hbase.regionserver.global.memstore.lowerLimit：0.38

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

四、HBase 项目

4.1、涉及概念梳理：命名空间

4.1.1、命名空间的结构



1) Table：表，所有的表都是命名空间的成员，即表必属于某个命名空间，如果没有指定，则在 default 默认的命名空间中。

2) RegionServer group：一个命名空间包含了默认的 RegionServer Group。

3) Permission：权限，命名空间能够让我们来定义访问控制列表 ACL（Access Control List）。例如，创建表，读取表，删除，更新等等操作。

4) Quota：限额，可以强制一个命名空间可包含的 region 的数量。（属性：hbase.quota.enabled）

4.1.2、命名空间的使用

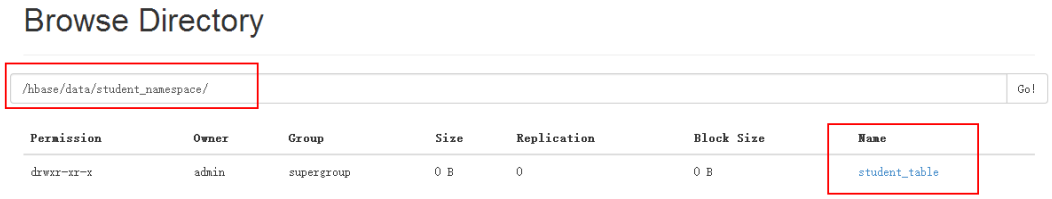
1) **创建命名空间**

```
hbase(main):002:0> create_namespace 'ns_school'
```

2) **创建表时指定命名空间**

```
hbase(main):004:0> create 'ns_school:tbl_student','info'
```

3) 观察 HDFS 中的目录结构的变化



4.2、微博系统

4.1.1、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

4.1.2、代码实现

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

1) 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("ns_weibo:content");
//用户关系表的表名
private static final byte[] TABLE_RELATION = Bytes.toBytes("ns_weibo:relation");
//微博收件箱表的表名
private static final byte[] TABLE_INBOX = Bytes.toBytes("ns_weibo:inbox");


/**
 * 初始化命名空间
 * @param args
 */
public void initNamespace(){
    HBaseAdmin admin = null;
    try {
        Connection connection = ConnectionFactory.createConnection(conf);
        admin = (HBaseAdmin) connection.getAdmin();
        //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("ns_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();
                }
            }
        }
}
```

2) 创建微博内容表

表结构：

方法名	creatTableeContent
Table Name	ns_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:ns_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;
        Connection connection = null;
        try {
                connection = ConnectionFactory.createConnection(conf);
                admin = (HBaseAdmin) connection.getAdmin();
                //创建表表述
                HTableDescriptor contentTableDescriptor = new
HTableDescriptor(TableName.valueOf(TABLE_CONTENT));
                //创建列族描述
                HColumnDescriptor infoColumnDescriptor = new
HColumnDescriptor(Bytes.toBytes("info"));
                //设置块缓存
                infoColumnDescriptor.setBlockCacheEnabled(true);
                //设置块缓存大小
                infoColumnDescriptor.setBlocksize(2097152);
                //设置压缩方式
//                      infoColumnDescriptor.setCompressionType(Algorithm.SNAPPY);
                //设置版本确界
                infoColumnDescriptor.setMaxVersions(1);
                infoColumnDescriptor.setMinVersions(1);
                contentTableDescriptor.addFamily(infoColumnDescriptor);
                admin.createTable(contentTableDescriptor);
```

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

3) 创建用户关系表

表结构：

方法名	createTableRelations
Table Name	ns_weibo:relation
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:ns_weibo:relation
 * 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 createTableRelation(){
    HBaseAdmin admin = null;
    try {
        Connection connection = ConnectionFactory.createConnection(conf);
        admin = (HBaseAdmin) connection.getAdmin();
        HTableDescriptor relationTableDescriptor = new
HTableDescriptor(TableName.valueOf(TABLE_RELATION));

        //关注的人的列族
        HColumnDescriptor attendColumnDescriptor = new
HColumnDescriptor(Bytes.toBytes("attends"));
        //设置块缓存
        attendColumnDescriptor.setBlockCacheEnabled(true);
        //设置块缓存大小
        attendColumnDescriptor.setBlocksize(2097152);
        //设置压缩方式
//            attendColumnDescriptor.setCompressionType(Algorithm.SNAPPY);
        //设置版本确界
        attendColumnDescriptor.setMaxVersions(1);
        attendColumnDescriptor.setMinVersions(1);

        //粉丝列族
        HColumnDescriptor fansColumnDescriptor = new
```

```
HColumnDescriptor(Bytes.toBytes("fans"));
            fansColumnDescriptor.setBlockCacheEnabled(true);
            fansColumnDescriptor.setBlocksize(2097152);
            fansColumnDescriptor.setMaxVersions(1);
            fansColumnDescriptor.setMinVersions(1);


            relationTableDescriptor.addFamily(attendColumnDescriptor);
            relationTableDescriptor.addFamily(fansColumnDescriptor);
            admin.createTable(relationTableDescriptor);


        } 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();
                }
            }
        }
}
```

4) 创建微博收件箱表

表结构：

方法名	createTableInbox
Table Name	ns_weibo:inbox
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: ns_weibo:inbox
 * 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 createTableInbox(){
    HBaseAdmin admin = null;
    try {
        Connection connection = ConnectionFactory.createConnection(conf);
        admin = (HBaseAdmin) connection.getAdmin();

        HTableDescriptor inboxTableDescriptor = new
HTableDescriptor(TableName.valueOf(TABLE_INBOX));
        HColumnDescriptor infoColumnDescriptor = new
HColumnDescriptor(Bytes.toBytes("info"));
```

```
            infoColumnDescriptor.setBlockCacheEnabled(true);
            infoColumnDescriptor.setBlocksize(2097152);
            infoColumnDescriptor.setMaxVersions(1000);
            infoColumnDescriptor.setMinVersions(1000);

            inboxTableDescriptor.addFamily(infoColumnDescriptor);;
            admin.createTable(inboxTableDescriptor);
        } 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();
                }
            }
        }
}
```

5) 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.z.hbase.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){
        Connection connection = null;
        try {
                connection = ConnectionFactory.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
                Table contentTable = connection.getTable(TableName.valueOf(TABLE_CONTENT));
                //组装 Rowkey
                long timestamp = System.currentTimeMillis();
                String rowKey = uid + "_" + timestamp;
                //添加微博内容
                Put put = new Put(Bytes.toBytes(rowKey));
                put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("content"), timestamp,
Bytes.toBytes(content));
                contentTable.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
```

```
            Table relationTable = connection.getTable(TableName.valueOf(TABLE_RELATION));
            //b.2、取出目标数据
            Get get = new Get(Bytes.toBytes(uid));
            get.addFamily(Bytes.toBytes("fans"));

            Result result = relationTable.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
            Table inboxTable = connection.getTable(TableName.valueOf(TABLE_INBOX));
            //每一个粉丝，都要向收件箱中添加该微博的内容，所以每一个粉丝都是一个 Put
对象
            List<Put> puts = new ArrayList<Put>();
            for(byte[] fan : fans){
                Put fansPut = new Put(fan);
                fansPut.addColumn(Bytes.toBytes("info"), Bytes.toBytes(uid), timestamp,
Bytes.toBytes(rowKey));
                puts.add(fansPut);
            }

            inboxTable.put(puts);
        } catch (IOException e) {
```

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

6) 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;
```

```
        }

        Connection connection = null;
        try {
            connection = ConnectionFactory.createConnection(conf);

            //用户关系表操作对象（连接到用户关系表）
            Table relationTable = connection.getTable(TableName.valueOf(TABLE_RELATION));
            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.addColumn(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.addColumn(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);
            relationTable.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
            Table contentTable = 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 = contentTable.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 放置于当前操作的用户的收件箱中
            if(rowkeys.size() <= 0) return;
            //得到微博收件箱表的操作对象
            Table inboxTable = connection.getTable(TableName.valueOf(TABLE_INBOX));
```

```
            //用于存放多个关注的用户的发布的多条微博 rowkey 信息
            List<Put> inboxPutList = new ArrayList<Put>();
            for(byte[] rk : rowkeys){
                Put put = new Put(Bytes.toBytes(uid));
                //uid_timestamp
                String rowKey = Bytes.toString(rk);
                //截取 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.addColumn(Bytes.toBytes("info"), Bytes.toBytes(attendUID), timestamp, rk);
                inboxPutList.add(put);
            }
            inboxTable.put(inboxPutList);

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

7) 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;

    try {
        Connection connection = ConnectionFactory.createConnection(conf);

        //a、在微博用户关系表中，删除已关注的好友
        Table relationTable = connection.getTable(TableName.valueOf(TABLE_RELATION));
        //待删除的用户关系表中的所有数据
        List<Delete> deleteList = 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.addColumn(Bytes.toBytes("attends"), Bytes.toBytes(attend));
```

```
                //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）
                Delete fansDelete = new Delete(Bytes.toBytes(attend));
                fansDelete.addColumn(Bytes.toBytes("fans"), Bytes.toBytes(uid));
                deleteList.add(fansDelete);
            }

            deleteList.add(attendDelete);
            relationTable.delete(deleteList);

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

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

8) 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){
    Connection connection = null;
    try {
        connection = ConnectionFactory.createConnection(conf);

        Table inboxTable = connection.getTable(TableName.valueOf(TABLE_INBOX));
        //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 = inboxTable.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
        Table contentTable = 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 = contentTable.get(gets);
            //将每一条微博内容都封装为消息对象
            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;
}
```

9) 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);
}
```

五、总结

不一定所有的企业都会使用 HBase，大数据的框架可以是相互配合相互依赖的，同时，根据不同的业务，部分框架之间的使用也可以是相互独立的。例如有些企业在处理整个业务时，只是用 HDFS+Spark 部分的内容。所以在学习 HBase 框架时，一定要有宏观思维，了解其框架特性，不一定非要在所有的业务中使用所有的框架，要具体情况具体分析，酌情选择。

5.1、HBase 在商业项目中的能力

每天：

1) 消息量：发送和接收的消息数超过 60 亿

2) 将近 1000 亿条数据的读写

3) 高峰期每秒 150 万左右操作

4) 整体读取数据占有约 55%，写入占有 45%

5) 超过 2PB 的数据，涉及冗余共 6PB 数据

6) 数据每月大概增长 300 千兆字节。

5.2、HBase2.0 新特性

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

最新文档：

http://hbase.apache.org/book.html#ttl

官方发布主页：

http://mail-archives.apache.org/mod_mbox/www-announce/201708.mbox/<CADcMMgFzmX0xYYso-UAYbU7V8z-Obk1J4pxzbGkRzbP5Hps+iA@mail.gmail.com

举例：

1) region 进行了多份冗余

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

2) 更多变动

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