尚硅谷大数据技术之 HBase

(作者：尚硅谷大数据研发部)

版本：V2.0

第 1 章 HBase 简介

1.1 什么是 HBase

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。

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

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

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

1.2 HBase 特点

1）海量存储

Hbase 适合存储 PB 级别的海量数据，在 PB 级别的数据以及采用廉价 PC 存储的情况下，能在几十到百毫秒内返回数据。这与 Hbase 的极易扩展性息息相关。正式因为 Hbase 良好的扩展性，才为海量数据的存储提供了便利。

2）列式存储

这里的列式存储其实说的是列族存储，Hbase 是根据列族来存储数据的。列族下面可以有非常多的列，列族在创建表的时候就必须指定。

3）极易扩展

Hbase 的扩展性主要体现在两个方面，一个是基于上层处理能力（RegionServer）的扩展，一个是基于存储的扩展（HDFS）。

通过横向添加 RegionSever 的机器，进行水平扩展，提升 Hbase 上层的处理能力，提升 Hbsae 服务更多 Region 的能力。

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

4）高并发

由于目前大部分使用 Hbase 的架构，都是采用的廉价 PC，因此单个 IO 的延迟其实并不小，一般在几十到上百 ms 之间。这里说的高并发，主要是在并发的情况下，Hbase 的单个 IO 延迟下降并不多。能获得高并发、低延迟的服务。

5）稀疏

稀疏主要是针对 Hbase 列的灵活性，在列族中，你可以指定任意多的列，在列数据为空的情况下，是不会占用存储空间的。

1.3 HBase 架构

Hbase 架构如图 1 所示：

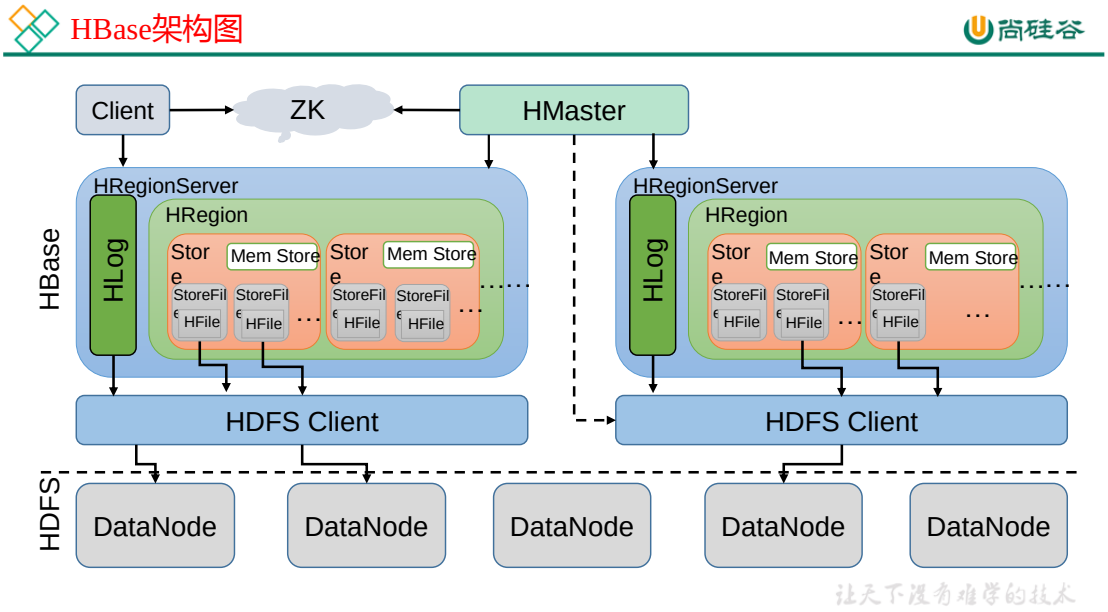


图 1 HBase 架构图

从图中可以看出 Hbase 是由 Client、Zookeeper、Master、HRegionServer、HDFS 等几个组件组成，下面来介绍一下几个组件的相关功能：

1）Client

Client 包含了访问 Hbase 的接口，另外 Client 还维护了对应的 cache 来加速 Hbase 的访问，比如 cache 的.META.元数据的信息。

2）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。具体工作如下：

通过 Zoopkeeper 来保证集群中只有 1 个 master 在运行，如果 master 异常，会通过竞争机制产生新的 master 提供服务

通过 Zoopkeeper 来监控 RegionServer 的状态，当 RegionSevrer 有异常的时候，通过回调的形式通知 Master RegionServer 上下线的信息

通过 Zoopkeeper 存储元数据的统一入口地址

3）Hmaster

master 节点的主要职责如下：

为 RegionServer 分配 Region

维护整个集群的负载均衡

维护集群的元数据信息

发现失效的 Region，并将失效的 Region 分配到正常的 RegionServer 上

当 RegionSever 失效的时候，协调对应 Hlog 的拆分

4）HregionServer

HregionServer 直接对接用户的读写请求，是真正的“干活”的节点。它的功能概括如下：

管理 master 为其分配的 Region

处理来自客户端的读写请求

负责和底层 HDFS 的交互，存储数据到 HDFS

负责 Region 变大以后的拆分

负责 Storefile 的合并工作

5）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（Hlog 存储在 HDFS）的支持，具体功能概括如下：

提供元数据和表数据的底层分布式存储服务

数据多副本，保证的高可靠和高可用性

1.3 HBase 中的角色

1.3.1 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 的分配或转移
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

1.3.2 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

1.2.3 其他组件

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．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。

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．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。StoreFile 是以 Hfile 的形式存储在 HDFS 的。

第 2 章 HBase 安装

2.1 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
```

2.2 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
```

2.3 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
```

2.4 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/jdk1.8.0_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>hadoop102:2181,hadoop103:2181,hadoop104:2181</
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
```

2.5 HBase 远程发送到其他集群

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

2.6 HBase 服务的启动

1．启动方式 1

```
[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

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

对应的停止服务：

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

2.7 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

第 3 章 HBase Shell 操作

3.1 基本操作

1．进入 HBase 客户端命令行

```
[atguigu@hadoop102 hbase]$ bin/hbase shell
```

2．查看帮助命令

```
hbase(main):001:0> help
```

3．查看当前数据库中有哪些表

```
hbase(main):002:0> list
```

3.2 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：

```
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 4 章 HBase 数据结构

4.1 RowKey

与 nosql 数据库们一样,RowKey 是用来检索记录的主键。访问 HBASE table 中的行，只有三种方式：

1.通过单个 RowKey 访问

2.通过 RowKey 的 range（正则）

3.全表扫描

RowKey 行键 (RowKey)可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在 HBASE 内部，RowKey 保存为字节数组。存储时，数据按照 RowKey 的字典序(byte order)排序存储。设计 RowKey 时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

4.2 Column Family

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

4.3 Cell

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

关键字：无类型、字节码

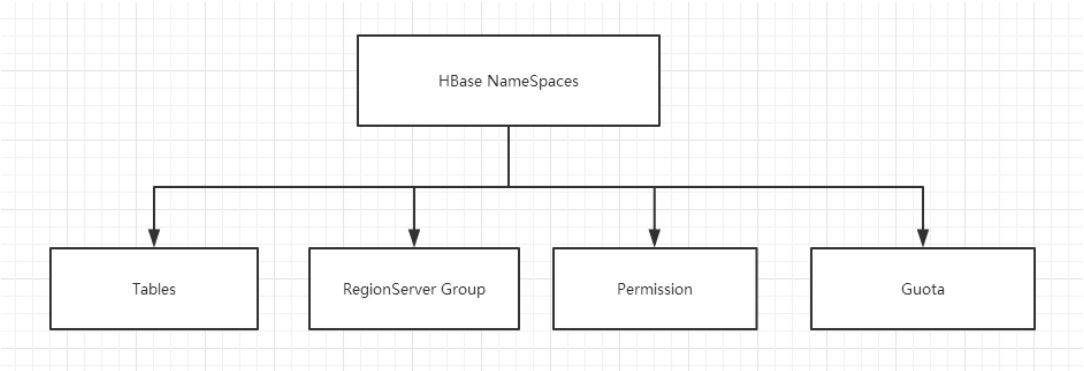
4.4 Time Stamp

HBASE 中通过 rowkey 和 columns 确定的为一个存贮单元称为 cell。每个 cell 都保存 着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 HBASE(在数据写入时自动)赋值，此时时间戳是精确到毫秒 的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版 本冲突，就必须自己生成具有唯一性的时间戳。每个 cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的的管理 (包括存贮和索引)负担，HBASE 提供 了两种数据版本回收方式。一是保存数据的最后 n 个版本，二是保存最近一段 时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

4.5 命名空间

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。

第 5 章 HBase 原理

5.1 读流程

HBase 读数据流程如图 3 所示

HBase读数据流程

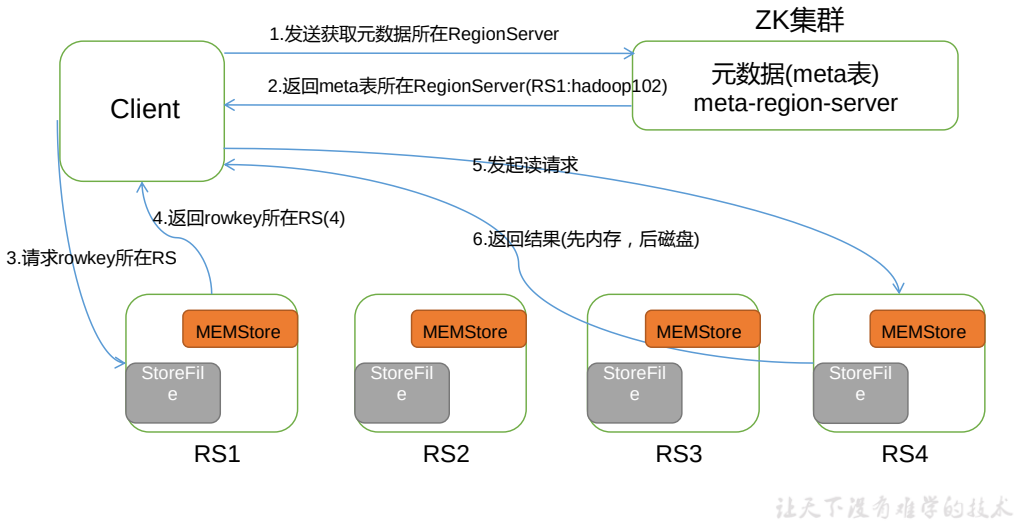


图 3 所示 HBase 读数据流程

1）Client 先访问 zookeeper，从 meta 表读取 region 的位置，然后读取 meta 表中的数据。meta 中又存储了用户表的 region 信息；

2）根据 namespace、表名和 rowkey 在 meta 表中找到对应的 region 信息；

3）找到这个 region 对应的 regionserver；

4）查找对应的 region；

5）先从 MemStore 找数据，如果没有，再到 BlockCache 里面读；

6）BlockCache 还没有，再到 StoreFile 上读(为了读取的效率)；

7）如果是从 StoreFile 里面读取的数据，不是直接返回给客户端，而是先写入 BlockCache，再返回给客户端。

5.2 写流程

Hbase 写流程如图 2 所示

HBase写数据流程

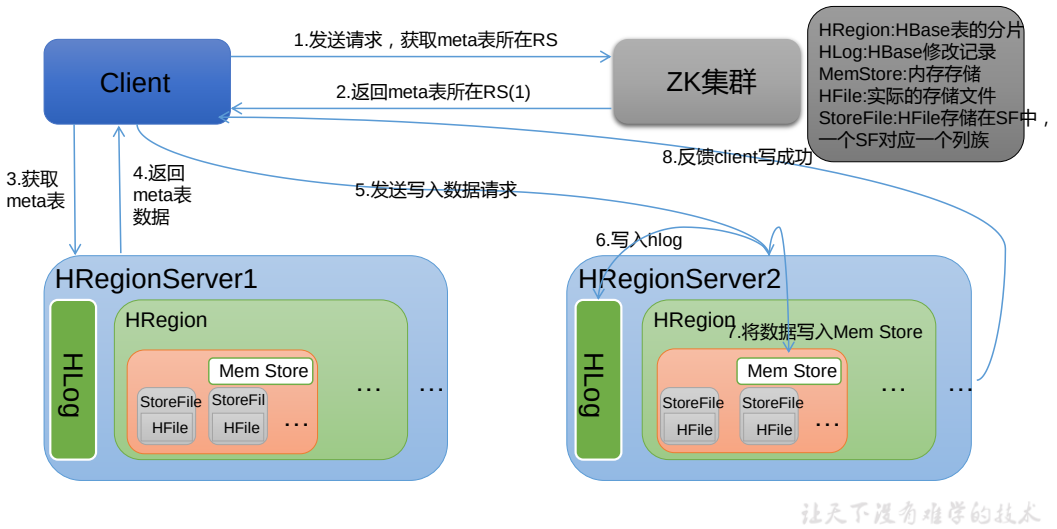


图 2　HBase 写数据流程

1）Client 向 HregionServer 发送写请求；

2）HregionServer 将数据写到 HLog（write ahead log）。为了数据的持久化和恢复；

3）HregionServer 将数据写到内存（MemStore）；

4）反馈 Client 写成功。

5.3 数据 Flush 过程

1）当 MemStore 数据达到阈值（默认是 128M，老版本是 64M），将数据刷到硬盘，将内存中的数据删除，同时删除 HLog 中的历史数据；

2）并将数据存储到 HDFS 中；

3）在 HLog 中做标记点。

5.4 数据合并过程

1）当数据块达到 4 块，Hmaster 触发合并操作，Region 将数据块加载到本地，进行合并；

2）当合并的数据超过 256M，进行拆分，将拆分后的 Region 分配给不同的 HregionServer 管理；

3）当 HregionServer 宕机后，将 HregionServer 上的 hlog 拆分，然后分配给不同的 HregionServer 加载，修改.META.；

4）注意：HLog 会同步到 HDFS。

第 6 章 HBase API 操作

6.1 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>

<dependency>
    <groupId>jdk.tools</groupId>
    <artifactId>jdk.tools</artifactId>
    <version>1.8</version>
    <scope>system</scope>
    <systemPath>${JAVA_HOME}/lib/tools.jar</systemPath>
</dependency>
```

6.2 HBaseAPI

6.2.1 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", "192.168.9.102");
conf.set("hbase.zookeeper.property.clientPort", "2181");
}
```

6.2.2 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);
}
```

6.2.3 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！");
    }
}
```

6.2.4 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！");
    }
}
```

6.2.5 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");
}
```

6.2.6 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();
}
```

6.2.7 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)));
        }
    }
}
```

6.2.8 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());
    }
}
```

6.2.9 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)));
    }
}
```

6.3 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。

6.3.1 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-1.3.1
$ 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-1.3.1
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’
```

6.3.2 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）

6.3.3 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
```

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）

6.4 与 Hive 的集成

6.4.1 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。

6.4.2 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>
```