1.产生的背景

- 从1970年开始，大多数的公司数据存储和维护使用的是关系型数据库
- 大数据技术出现后，很多拥有海量数据的公司开始选择像Hadoop的方式来存储海量数据
- Hadoop使用分布式文件系统HDFS来存储海量数据，并使用MapReduce来处理。Hadoop擅长于存储各种格式的庞大的数据，任意的格式甚至非结构化的处理
- 但是Hadoop存在局限性
  - hdfs不支持随机读写，仅支持顺序读写，hdfs适合批量数据操作，实际需要有随机读写的工具
- HBase采用java于开发，基于HDFS，是一个支持高效随机读写的NoSQL数据库

2.HBase的基本介绍

- NoSQL是一个通用术语，泛指一个数据库并不是使用SQL作为主要语言的非关系型数据库
- HBase是BIgTable的开源java版本，是建立在HDFS之上，提供高可靠性、高性能、列存储、实时读写NoSQL的数据库系统
- HBase仅能通过主键（row key）和主键的range来检索数据，本质是一个存储容器。仅支持单行事务，不支持多行事务，不支持SQL
  - 1.主键查询rowkey
  - 2.主键范围查询rowkey
  - 3.全表查询
- 主要用来存储结构化和半结构化的松散数据
- HBase查询数据功能很简单，不支持join等复杂操作，不支持复杂的事务（行级的事务），从技术上来说，HBase更像是一个【数据存储】而不是【数据库】，因为HBase缺少RDBMAS中的许多特性，例如带类型的列、二级索引以及高级查询语言等
- HBase中支持的数据类型：byte[] --所有数据都是字节
- 与Hadoop一样，HBase目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加存储和处理能力，例如，把集群中从10个节点扩展到20个节点，存储能力和处理能力都会加倍
- HBase中的表特点
  - 大：一个表可以有上十亿行，上百万列
  - 面向列：面向列（族）的存储和权限控制，列（族）独立检索
  - 稀疏：对于空（null ）的列，并不占用存储空间，因此，表可以设计的非常稀疏

3.HBase的特点

- 强一致性读/写：HBase 不是“最终一致的”数据存储，它非常适合于诸如高速计数器聚合等任务
- 自动分块：HBase表通过Region分布在集群上，随着数据的增长，区域被自动拆分成和重新分布
- 自动RegionServer故障转移
- Hadoop/HDFS集成：HBase支持HDFS开箱即用作为其分布式文件系统

- MapReduce：HBase通过MapReduce支持大规模并行处理，将HBase用作源和接收器
- Java Client API：HBase支持易于使用的Java API进行编程访问
- Thrift/REST API
-
- 块缓存换布隆过滤器：HBase支持块Cache和Bloom过滤器进行大容量查询优化
- 运行管理：HBase为业务洞察和JMX度量提供内置网页

4.HBase与RDBMS、HDFS、Hive的区别

	HBase	RDBMS
结构	1. 以表形式存在 2. 支持HDFS文件系统 3. 使用行键（row key） 4. 原生支持分布式存储、计算引擎 5. 使用行、列、列族和单元格	1. 数据库以表的形式存在 2. 支持FAT、NTFS、EXT、文件系统 3. 使用主键（PK） 4. 通过外部中间件可以支持分库分表，但底层还是单机引擎 5. 使用行、列、单元格
功能	1. 支持向外扩展 2. 使用API和MapReduce、Spark、Flink来访问HBase表数据 3. 面向列簇，即每一个列簇都是一个连续的单元 4. 数据总量不依赖具体某台机器，而取决于机器数量 HBase不支持ACID（Atomicity、Consistency、Isolation、Durability） 5. 适合结构化数据和非结构化数据 6. 一般都是分布式的 7. HBase不支持事务，支持的是单行数据的事务操作 不支持Join	1. 支持向上扩展（买更好的服务器） 2. 使用SQL查询 3. 面向行，即每一行都是一个连续单元 4. 数据总量依赖于服务器配置 5. 具有ACID支持 6. 适合结构化数据 7. 传统关系型数据库一般都是中心化的 8. 支持事务 9. 支持Join

- hbase：以表的形式存储数据，不支持sql，不支持复杂事务，仅支持单行事务，数据分布式存储，结构化数据和半结构化数据，不支持join操作
- rdbms：以表的形式存储数据，支持sql，支持复杂事务，中心化存储，支持join操作

HBase	HDFS
1. HBase构建在HDFS之上，并为大型表提供快速记录查找(和更新) 2. HBase内部将大量数据放在HDFS中名为「StoreFiles」的索引中，以便进行高速查找 3. Hbase比较适合做快速查询等需求，而不适合做大规模的OLAP应用	1. HDFS是一个非常适合存储大型文件的分布式文件系统 2. HDFS它不是一个通用的文件系统，也无法在文件中快速查询某个数据

- Hbase 依赖于hdfs，数据存在hdfs上，支持高效的随机读写
- hdfs 分布式文件系统，适合批量存储，不支持随机读写

HBase	Hive
1. NoSQL数据库 2. 是一种面向列存储的非关系型数据库。 3. 用于存储结构化和非结构化的数据 4. 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。 5. 基于HDFS 6. 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 6. 延迟较低，接入在线业务使用 8. 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	1. 数据仓库工具 2. 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 3. 用于数据分析、清洗 4. Hive适用于离线的数据分析和清洗，延迟较高 5. 基于HDFS、MapReduce 6. 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
总结: hbase与hive Hive和Hbase是两种基于Hadoop的不同技术 Hive是一种类SQL的引擎，并且运行MapReduce任务 Hbase是一种在Hadoop之上的NoSQL 的Key/value数据库 这两种工具是可以同时使用的。就像用Google来搜索，用FaceBook进行社交一样，Hive可以用来进行统计查询，HBase可以用来进行实时查询，数据也可以从Hive写到HBase，或者从HBase写回Hive	

- hbase：依赖于hdfs，数据存在hdfs上，nosql存储容器，延迟较低，可以接入在线业务
- hive：依赖于hdfs，mapreduce任务，数仓分析工具，延迟较高，一般用作离线业务分析中

5.HBase的表模型

- 在HBase中，数据存储在具有行和列的表中。这是看起来关系数据库（RDBMS）一样，但将HBASE表看成是多

个维度的map结构更容易理解

rowKey	Column Family1 userInfo store1 store2				Column Family2 addressInfo store3						Column FamilyN...				timeStamp	versionNum 版本号
	name	age	sex	password	address	from	phone	email	salary	regtime	...	...	...	...		
1	zhangsan	18	1	123456	地球村	火星	13612345678	666@163.com	500	2018/12/20 12:23					1545307281	1
2	李四	28	1	123456	地球村	月球	13612345678	667@163.com	600	2018/12/21 12:23					1545393681	2
3	黄晓明	58	1	123456	地球村	土星	13612345678	668@163.com	800	2018/12/21 12:23					1545480081	3
4	按住啦baby	25	0	123456	地球村	韩国整容	13612345678	669@163.com	15000	2018/12/22 12:23					1545566481	1

- 表（Table）：HBase中数据都是以表形式来组织的，HBase中的表由多个行组成
- 行键（row key）：类似于数据库中的主键，只能通过rowkey进行查找，rowkey不能重复，rowkey是以字典序进行排序
  - HBase中的行有一个rowkey（行键）和一个或者多个列组成，列的值与rowkey、列相关联
  - 行在存储是按行键的字典序排序
  - 行键的设计非常重要，尽量让相关的行存储在一起
- 列（Column）：HBase中的列有列族（column family）+ 列限定符（列名）（column Qualifier）组成
  - 表示如下：列族名：列限定符 例如：C1:USER_ID C1:SEX
- 列族（Column Family）：
  - 出于性能原因，列族将一组列及其值组织在一起
  - 每个列族都有一个存储属性：例如 是否应该换成在内存中，数据如何被压缩等
  - 表中的每一行都有相同的列族，但在列族中不存储任何内容
  - 所有的列族的数据全部都存储在一块（文件系统HDFS）
  - HBase官方建议所有的列族保持一样的列，并且将同一类的列放在一个列族中
- 列限定符（Column Qualifier）：一个列限定符必然属于某个列族，创建表时可以先不指定限定符，添加数据时再动态指定
  - 列族中包含一个个的列限定符，这样可以为存储的数据提供索引
  - 列族在创建表的时候是固定的，但列限定符是不做限制的
  - 不同的列可能会存在不同的列标识符
- 单元格（Cell）：单元格是行、列族和列限定符的组合，包含一个值和一个时间戳，数据以二进制存储
  - rowkey+（列族+列限定符）列名，每个cell都有时间戳和版本号
- 版本号（verson num）：每条数据逗号有版本号的概念
  - 每条数据都可以有多个版本号，默认值为系统时间戳，类型为Long
- 时间戳（timeStamp）：每个数据都会有时间戳的概念
  - 在向HBase插入更新数据时，HBase默认会将当前操作的时间记录下来，当然也可以人为指定时间
  - 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

6.HBase命令

- **进入HBase客户端**

```
hbase shell
```

- **查看帮助命令**

```
help
```

- **查看当前数据库中有哪些表**

```
list
```

- **创建一张表**

```
创建user表，包含info、data两个列族
create "表名", "列族名1"[, "列族名2", ...] # 方括号表示内容时可选的，单引号双引号都可以
create 'user', 'info', 'data'
或者，NAME是列族名，VERSIONS是版本号
 create 'user', {NAME => 'info', VERSIONS => '3'},{NAME => 'data'}
```

- **添加数据操作**

```
格式
put "表名", "rowkey", "列族:列限定符", "值"

put "day01", "rk001", "f1:name", "张三"
rowkey是自己设定的
列族是创建表时必须要指定一个
列限定符时插入数据时指定

day01
                f1
        name      age birthday
rk001   zhangsan  18 2000-3-7
rk002   lisi      20 2002-3-7

向user表中插入信息，row key为rk0001，列族info中添加name列标示符，值为zhangsan
格式：put 表名 ，rowkey ，列族：列名，  值
hbase(main):011:0> put 'user', 'rk0001', 'info:name', 'zhangsan'


向user表中插入信息，row key为rk0001，列族info中添加gender列标示符，值为female
格式：put 表名，  rowkey，  列族：列名 ，值
```

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

向user表中插入信息，row key为rk0001，列族info中添加age列标示符，值为20
put 表名， rowkey， 列族：列名， 值
hbase(main):013:0> put 'user', 'rk0001', 'info:age', 20

向user表中插入信息，row key为rk0001，列族data中添加pic列标示符，值为picture
put 表名， rowkey， 列族：列名， 值
hbase(main):014:0> put 'user', 'rk0001', 'data:pic', 'picture'
```

- **查询数据操作**

```
格式
get "表名", "rowkey"[, "列族1", "列族2"] [, "列族:列限定符"] [, "列族1", "列族2:列限定符"]

1.1通过rowkey进行查询
获取user表中row key为rk0001的所有信息
格式 ： get 表名,rowkey
hbase(main):015:0> get 'user', 'rk0001'

1.2、查看rowkey下面的某个列族的信息
获取user表中row key为rk0001，info列族的所有信息
格式：get 表名,rowkey,列族
hbase(main):016:0> get 'user', 'rk0001', 'info'

1.3、查看rowkey指定列族指定字段的值
获取user表中row key为rk0001，info列族的name、age列标示符的信息
格式：get 表名,rowkey,列族：列名,列族：列名
hbase(main):017:0> get 'user', 'rk0001', 'info:name', 'info:age’

1.4、查看rowkey指定多个列族的信息
获取user表中row key为rk0001，info、data列族的信息
格式：get 表名,rowkey,列族1，列族2...
hbase(main):018:0> get 'user', 'rk0001', 'info', 'data'
或者你也可以这样写
```

```
hbase(main):019:0> get 'user', 'rk0001', {COLUMN => ['info', 'data']}
或者你也可以这样写，也行
hbase(main):020:0> get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic’]}

1.5、指定rowkey与列值查询
获取user表中row key为rk0001，cell的值为zhangsan的信息
格式：get 表名,rowkey,{FILTER=>"VlaueFilter(=,'binary:值')"
hbase(main):030:0> get 'user', 'rk0001', {FILTER => "ValueFilter(=, 'binary:zhangsan')"}
```

- **scan命令查询**

```
格式
1 scan 进行全表扫描
scan "表名" # 全表扫描，会把整张表的数据都拿到客户端，一般不会直接进行scan操作

scan "表名" [, {COLUMNS=>["列族1","列族2"...], VERSIONS=>N, LIMIT=>N,
FORMATTER=>'toString'}]

LIMIT=>N 表示显示前N条数据
VERSIONS=>N 表示显示n个版本
FORMATTER=>'toString' 表示显示字符串，如果是中文 就显示汉字


1.6、查询所有数据 : 查询user表中的所有信息
scan 'user'
//显示中文
scan 'user' , {FORMATTER => 'toString’}
//显示前3个，并显示中文
scan 'user' , {LIMIT => 3,FORMATTER => 'toString’}

1.7、列族查询:  查询user表中列族为info的信息
scan 'user', {COLUMNS => 'info'}

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

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

1.8、多列族查询: 查询user表中列族为info和data的信息
scan 'user', {COLUMNS => ['info', 'data']}
scan 'user', {COLUMNS => ['info:name', 'data:pic']}
```

```
1.9、指定列族与某个列名查询
查询user表中列族为info、列标示符为name的信息
scan 'user', {COLUMNS => 'info:name’}

1.10、指定列族与列名以及限定版本查询
查询user表中列族为info、列标示符为name的信息,并且版本最新的5个
scan 'user', {COLUMNS => 'info:name', VERSIONS => 5}

1.12、指定多个列族与按照数据值模糊查询
查询user表中列族为info和data且列标示符中含有a字符的信息
scan 'user', {COLUMNS => ['info', 'data'], FILTER => "
(QualifierFilter(=,'substring:a'))"}

进行范围查询
格式
scan "表名", {STARTROW=>'rowkey1', ENDROW=>'rowkey2'}
# 范围是左闭右开，包含左边，不包含右边
1.13、rowkey的范围值查询
查询user表中列族为info，rk范围是[rk0001, rk0003)的数据
scan 'user', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}

1.14、指定rowkey模糊查询
查询user表中row key以rk字符开头的
scan 'user',{FILTER=>"PrefixFilter('rk')"}

1.15、指定数据范围值查询
查询user表中指定范围的数据
scan 'user', {TIMERANGE => [1392368783980, 1392380169184]}
```

- **过滤器的查询地址**
  - http://hbase.apache.org/2.2/devapidocs/index.html
- **更新数据操作**

```
1、更新数据值
更新操作同插入操作一模一样，只不过有数据就更新，没数据就添加
2、更新版本号
将user表的f1列族版本号改为5
hbase(main):050:0> alter 'user', NAME => 'info', VERSIONS => 5
```

- **删除操作**

```
格式
delete "表名", "rowkey", "列族:列限定符"
delete 删除的是指定列的最新值，剩下的是倒数第二新的值

格式
deleteall "表名", "rowkey", "列族:列限定符"
deleteall把整个列删除

deleteall "day01", "rk001", "f1:name"

deleteall "表名", "rowkey"
把指定的rowkey这一行删掉

1、指定rowkey以及列名进行删除
删除user表row key为rk0001，列标示符为info:name的数据
hbase(main):045:0> delete 'user', 'rk0001', 'info:name'
2、指定rowkey，删除一整行数据
hbase(main):045:0>deleteall 'user', 'rk0001’
注意:
1. deleteall 是在 hbase 2.0版本后出现的，在2.0版本之前，只需要使用delete这个命令即可完成所有的删除数据工作，
2. delete删除数据时候，只会删除最新版本的数据，而deleteall 直接将对应数据的所有的历史版本全部删除
3、删除一个列族:
alter 'user', NAME => 'info', METHOD => 'delete’  或 alter 'user', 'delete' => 'info'

格式
truncate "表名"
4、清空表数据
hbase(main):017:0> truncate 'user'

1 先把表禁掉
```

```
disable "表名"
2 删除表
drop "表名"

5、删除表
首先需要先让该表为disable状态，使用命令： hbase(main):049:0> disable 'user'
然后才能drop这个表，使用命令：hbase(main):050:0> drop 'user'
(注意：如果直接drop表，会报错：Drop the named table. Table must first be disabled)
```

- **统计一张表有多少行数据**

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

- **高级命令**
- **过滤器查询操作**

```
格式

scan "表名", {FILTER=>"过滤器名称(比较器运算符, 比较器表达式)"}


常见的过滤器
1 rowkey过滤器
        RowFilter: 实现行键的比较和过滤
        PrefixFilter: rowkey前缀过滤器

2 列族过滤器
        FamilyFilter: 过滤列族

3 列名过滤器
        QualifierFilter: 只显示对应的列名

4 列值过滤器
        valueFilter: 找到符合条件的键值对
        SingleColumnValueFilter 指定的列族中进行比较 把满足条件的留下
        SingleColumnExcludeFilter 把不满足条件的留下
5 其他过滤器
        PageFilter: 分页查询


```

```
比较运算符
> < >= <= != =


比较器
BinaryComparator 匹配完整字节
BinaryPrefixComparator 匹配字节前缀
NullComparator 匹配空值
SubstringComparator 模糊匹配

比较器表达式
binary:值
'binary:zhangsan'

binaryprefix:值
'binaryprefix:zhang'
null

substring:值
'substring:abc'

```

- **其他高级**

```
status  显示服务器状态  “例如：hbase(main):058:0> status 'node01'
whoami :  显示HBase当前用户，例如：  hbase> whoami
list : 显示当前所有的表
count: 统计指定表的记录数，例如：  hbase> count 'user'
describe : 展示表结构信息
exists: 检查表是否存在，适用于表量特别多的情况
is_enabled、is_disabled: 检查表是否启用或禁用
alter : 该命令可以改变表和列族的模式，
例如：  为当前表增加列族：  hbase> alter 'user', NAME => 'CF2', VERSIONS => 2
为当前表删除列族：  hbase(main):002:0> alter 'user', 'delete' => 'CF2'
disable/enable : 禁用一张表/启用一张表
drop : 删除一张表，记得在删除表之前必须先禁用
truncate : 禁用表-删除表-创建表
```

7.HBase的javaAPI操作

准备工作

- 创建maven项目
  - 1 构建父工程，pydata28sz
  - 2 删除src目录
  - 3 创建子工程 day01_hbase
- 导入相关依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>pydata28sz</artifactId>
        <groupId>cn.itcast</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>day01_hbase</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <repositories>
        <repository>
            <id>aliyun</id>
            <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
            <releases><enabled>true</enabled></releases>
            <snapshots><enabled>false</enabled></snapshots>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
```

```
            <version>2.1.0</version>
        </dependency>
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>2.6</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>6.14.3</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <target>1.8</target>
                    <source>1.8</source>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

- **创建表**

```
步骤
1 建立链接，链接到hbase
```

```
2 从链接对象中获取管理对象，
        Admin对象，对表的管理
        Table 对表的数据进行管理
3 执行相关操作：创建表 增加数据 删除
4 处理结果-- 有可能不需要的
5 释放资源
```

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.testng.annotations.Test;

import java.io.IOException;

public class TableAdminTest {
    @Test
    // 创建表
    public void test01() throws IOException {
        //1 建立链接，链接到hbase
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", "node1.itcast.cn,node2.itcast.cn,node3.itcast.cn");

        Connection hbConn = ConnectionFactory.createConnection(conf);
        //2 从链接对象中获取管理对象，
        //Admin对象，对表的管理
        //Table 对表的数据进行管理
        Admin admin = hbConn.getAdmin();

        //3 执行相关操作：创建表 增加数据 删除
        // 命令行创建表
        // create "表名", "列族名"
        boolean flag = admin.tableExists(TableName.valueOf("WATER_BILL"));
        if (!flag) {
            // 如果代码执行到这里，表示表不存在
            // 3.1 创建列族相关的构造器，设置列族名
            ColumnFamilyDescriptorBuilder columnFamilyDescriptorBuilder = ColumnFamilyDescriptorBuilder.newBuilder("C1".getBytes());
```

```
            ColumnFamilyDescriptor columnFamilyDescriptor = columnFamilyDescriptorBuilder.build();
            // 3.2 创建表相关的构造器，设置表名
            TableDescriptorBuilder tableDescriptorBuilder = TableDescriptorBuilder.newBuilder(TableName.valueOf("WATER_BILL"));
            // 3.3 别嫘祖构造器和表构造器关联，确定列族和表名
            tableDescriptorBuilder.setColumnFamily(columnFamilyDescriptor);
            TableDescriptor desc = tableDescriptorBuilder.build();

            // 3.4 创建表
            admin.createTable(desc);
        }
        //4 处理结果-- 有可能不需要的
        //5 释放资源 下面没有顺序要求
        hbConn.close();
        admin.close();
    }
}

```

- **添加数据**
- 1 创建Hbase链接对象
- 2 获取table对象，对表进行管理
- 3 执行操作：添加数据
- 4 处理结果
- 5 释放资源

```
    /**
     * 1 创建Hbase链接对象
     *
     * 2 获取table对象，对表进行管理
     *
     * 3 执行操作：添加数据
     *
     * 4 处理结果
     *
     * 5 释放资源
     */
    // 插入数据
    @Test
```

```
    public void test02() throws IOException {
        //  1 创建Hbase链接对象
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum",
"node1.itcast.cn,node2.itcast.cn,node3.itcast.cn");

        Connection hbConn = ConnectionFactory.createConnection(conf);

        // 2 获取table对象，对表进行管理
        Table table = hbConn.getTable(TableName.valueOf("WATER_BILL"));

        // 3 执行操作：添加数据
        // 命令行 put "表名", "rowkey", "列族:列限定符", "值"
        Put put = new Put(Bytes.toBytes("4944191"));
        put.addColumn("C1".getBytes(), "NAME".getBytes(), "登卫红".getBytes());
        put.addColumn("C1".getBytes(), "ADDR".getBytes(), "贵州省铜仁市德江县7单元267
室".getBytes());
        put.addColumn("C1".getBytes(), "SEX".getBytes(), "男".getBytes());
        put.addColumn("C1".getBytes(), "PAY_TIME".getBytes(), "2020-05-10".getBytes());

        // 执行put操作
        table.put(put);

        // 4. 处理结果
        // 5. 释放资源
        table.close();
        hbConn.close();
    }

```

- **抽取公共方法**

```
    private String tableName = "WATER_BILL";
    private Connection hbConn ;
    private Table table;
    private Admin admin;


    @Before
    public void  before() throws IOException {
        //  1 创建Hbase链接对象
```

```
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum",
"node1.itcast.cn,node2.itcast.cn,node3.itcast.cn");

        hbConn = ConnectionFactory.createConnection(conf);

        // 2 获取admin table对象，对表进行管理
        table = hbConn.getTable(TableName.valueOf(tableName));
        admin = hbConn.getAdmin();
    }

    @Test
    public void test03() {
        //
    }

    @After
    public void after() throws IOException {
        // 释放资源
        admin.close();
        table.close();
        hbConn.close();
    }
```

- **查询某一条数据**

```
@Test
public void test03() throws IOException {
    //查询数据
    //3.执行操作
    //get "表名","rowkey","列族：列限定符"
    Get get = new Get(Bytes.toBytes("4944191"));
    //一行数据
    Result result = table.get(get);
    //4.处理结果,把result单元格转成一个列表
    List<Cell> listCells = result.listCells();
    //遍历list,cellUtil工具类
    for (Cell cell : listCells) {
        //4.1获取rowkey
        byte[] bytes = CellUtil.cloneRow(cell);
        //转成一个String
```

```
        String rowkey = Bytes.toString(bytes);
        //4.2获取列族
        byte[] bytes1 = CellUtil.cloneFamily(cell);
        String s = Bytes.toString(bytes1);
        //4.3列限定符
        byte[] bytes2 = CellUtil.cloneQualifier(cell);
        String s1 = Bytes.toString(bytes2);
        //4.4获取值
        byte[] bytes3 = CellUtil.cloneValue(cell);
        String s2 = Bytes.toString(bytes3);
        System.out.println("rowkey:" + rowkey + " ,列族:  " + s + " ,列限定符:  " + s1 + " ,值: " + s2);
    }
}
```

- **删除数据**

```
    @Test
    public void test04() throws IOException {
        //删除数据
        // delete "表名","rowkey","列族: 列限定符"
        Delete delete = new Delete(Bytes.toBytes("4944191"));
        //删除列限定符
//        delete.addColumn("C1".getBytes(), "NAME".getBytes());
        delete.addFamily("C1".getBytes());
        table.delete(delete);
    }
```

- **删除表**

```
@Test
public void test05() throws IOException {
    //删除表
    //判断表是否存在
    boolean b = admin.tableExists(TableName.valueOf(tableName));
    if (b) {
        //判断表是否禁用掉
        boolean tableEnabled = admin.isTableEnabled(TableName.valueOf(tableName));
        if (tableEnabled) {
            //如果没有禁用则先禁用
            admin.disableTable(TableName.valueOf(tableName));
```

```
        }
        //删除表
        admin.deleteTable(TableName.valueOf(tableName));
    }
}
```

- **导入数据的操作**
- 1.把数据从本机上传到虚拟机

```
scp part-m-00000_10w root@192.168.88.161:/export/data
```

- 2.从虚拟机上传到hdfs

```
hdfs dfs -mkdir -p /water_bill/input
hdfs dfs -put /export/data/part-m-00000_10w /water_bill/input
```

- 通过hbase相关命令把数据导入到hbase中
  - 创建一张表

```
hbase(main):009:0> create "WATER_BILL", "C1"
```

- 执行导入命令，在终端中执行命令，需要启动yarn

```
# hbase org.apache.hadoop.hbase.mapreduce.Import 表名 HDFS数据文件路径
# 确保yarn是启动的
hbase org.apache.hadoop.hbase.mapreduce.Import WATER_BILL /water_bill/input
```

- 验证数据导入成功

```
list
验证表是否存在在

hbase shell中执行
count "WATER_BILL"
得到表所有行数
```

- 写代码进行查询

```
ADDRESS LATEST_DATE NAME NUM_CURRENT NUM_PREVIOUS NUM_USAGE PAY_DATE RECORD_DATE SEX
TOTAL_MONEY
rowkey

select name, num_usage from water_bill where record_date between "2020-06-01" and "2020-
06-30"
```

基于scan的扫描查询

需求：查询2020年6月所有用户用水量

实现步骤

```
1.通过构造scan对象进行全表扫描
2.构造过滤器
3.通过scan对象进行查询
4.把查询结果显示出来
    前面通过get对象获取的数据是一行数据
    现在通过scan对象获取的是多行数据
1.值过滤器，针对某个特定列（RECORD_DATE）
    SingcolumnValueFilter
2.    >= 2020-06-01 < 2020-07-01
3.BinaryComparator
```

```
@Test
public void test06() throws IOException {
    /**
     * 1.通过构造scan对象进行全表扫描
     * 2.构造过滤器
     * 3.通过scan对象进行查询
     * 4.把查询结果显示出来
     */
    //1.scan对象
    Scan scan = new Scan();
    //2.设置过滤器
    SingleColumnValueFilter startFilter = new SingleColumnValueFilter(
            "C1".getBytes(), "RECORD_DATE".getBytes(), CompareOperator.GREATER_OR_EQUAL, new BinaryComparator("2020-06-01".getBytes())
    );
    SingleColumnValueFilter endFilter = new SingleColumnValueFilter(
            "C1".getBytes(), "RECORD_DATE".getBytes(), CompareOperator.LESS, new BinaryComparator("2020-07-01".getBytes())
    );
    //构造FilterList
    FilterList filterList = new FilterList();
    filterList.addFilter(startFilter);
    filterList.addFilter(endFilter);
```

```
    scan.setFilter(filterList);
    //3.进行扫描
    ResultScanner scanner = table.getScanner(scan);
    //4.两层for循环
    for (Result result : scanner) {
        List<Cell> cells = result.listCells();
        String rowkey = null;
        String name = null;
        String record_date = null;
        Double num_usage = null;
        for (Cell cell : cells) {
            String qualifier = Bytes.toString(CellUtil.cloneQualifier(cell));
            if ("NAME".equals(qualifier)) {
                rowkey = Bytes.toString(CellUtil.cloneRow(cell));
                name = Bytes.toString(CellUtil.cloneValue(cell));

            }
            if ("RECORD_DATE".equals(qualifier)) {
                record_date = Bytes.toString(CellUtil.cloneValue(cell));
            }

            if ("NUM_USAGE".equals(qualifier)) {
                num_usage = Bytes.toDouble(CellUtil.cloneValue(cell));
            }
        }
        System.out.println("id:" + rowkey + " ,用户名: " + name + " ,日期: " + record_date + " ,用水量: " + num_usage);
    }

}
```

8.HBase的高可用

- hbase高可用就是让hbase主节点有多台，一旦其中主节点挂了，其他备份主节点可以顶上来
- 1.配置

```
cd /export/server/hbase-2.1.0/conf
vim backup-masters
添加以下内容
node2
```

```
node3
```

- 2.把配置文件分发

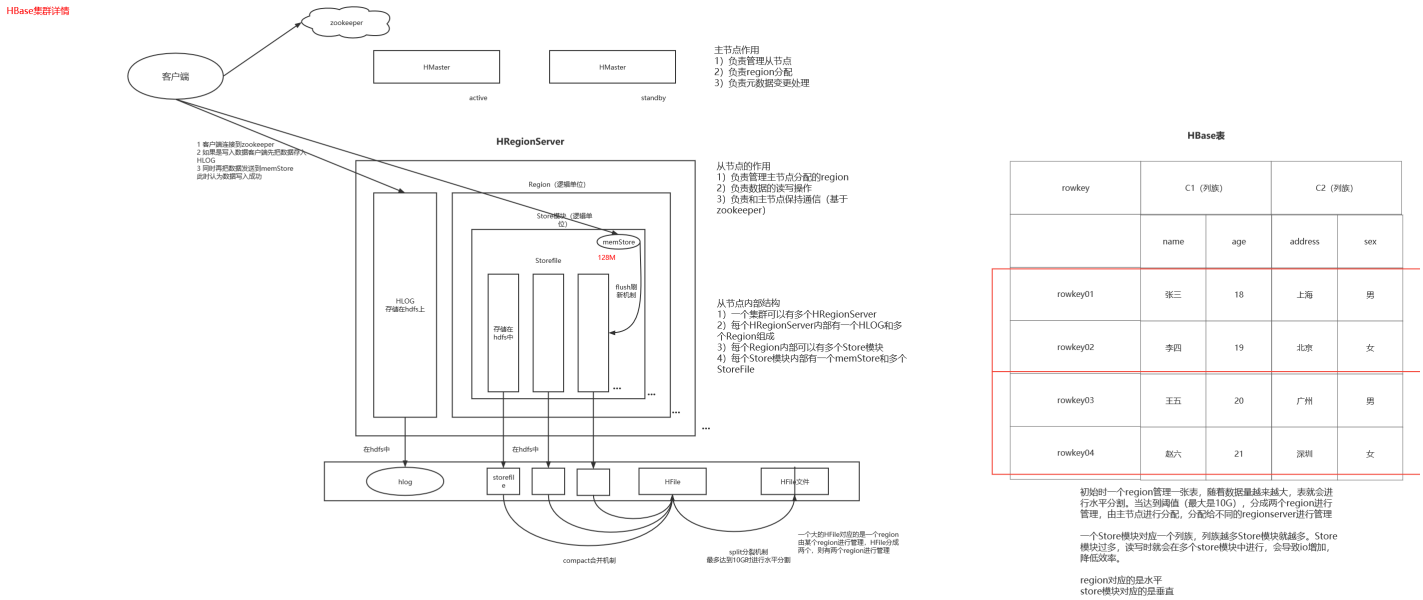
```
scp backup-masters node2:$PWD
scp backup-masters node3:$PWD
```

- 重启集群

```
stop-hbase.sh 停止集群时出问题，kill命令把进程杀掉
start-hbase.sh
```

9.HBase的集群架构

- HMaster主节点作用
  - 1.负责管理从节点
  - 2.负责region分配
  - 3.负责元数据变更处理
- HRegionServer从节点的作用
  - 1.负责管理主节点分配的region
  - 2.负责数据的读写操作
  - 3.负责和主节点保持通信（基于zookeeper）
- 从节点内部结构
  - 1.一个集群可以有多个HRegionServer
  - 2.每个HRegionServer内部有一个HLOG和多个Region组成
  - 3.每个Region内部可以有多个Store模块
  - 4.每个Store模块内部有一个MEMStore和多个StoreFile
  -



10.HBase的原理

- ## 读取数据的流程

```
hbase中有一张  hbase:meta 表，存的是元数据
客户端向集群发去读取数据的请求：  scan “表名”
1.首先会连接zookeep er，从zookeeper中获取hbase：meta表的位置信息，接着客户端读取这个
元数据表，就可以获取集群中所有表的信息，包括每张表有多少个region，每个region有哪些节点进行管理

2.连接hbase：meta所在的regionserver，获取meta表，从而获取到希望读取的表的信息

3.并发的连接响应的regionserver，从各个regionserver中读取表的数据。如果scan扫描
，把所有的数据拉取到客户端，如果是get，则从regionserver中找到具体的rowkey

从
memStore读取数据
blockcache（hdfs）经常被访问，会缓存起来
storeFile
HFile
```

- ## 数据的写入流程

```
客户端发送写入数据的请求put "day02" , "rk001" ,  "C1:name" , "zs"
1.首先连接到zookeeper，获取hbase:meta表在哪个regionserver
2.连接对应的regionserver，获取meta信息，根据day02表的，
    STARTKEY ENDKEY可以确定要写入的数据在哪个region，就知道需要写入到哪个regionserver
```

```
3.连接到对象的regionserver，开始进行写入数据操作
4.把数据写入到HLOG
5.同时把数据写入到memStore，如果有多个列族，则写入多个memStore
6.HLOG和MEMStore都写完，客户端认为写入成功

以上都是客户端流程

服务端HRegionServer

7.随着数据越来越多，memStore达到阈值（128M/1小时），触发flush刷新机制，
将memStore内容写入到HDFS中（旧版），形成一个storeFile文件

8.当flush不断进行，HDFS中storeFile（小HFile文件）越来越多，当storeFile文件
达到三个及以上，会触发compact合并压缩机制，就会把多个小的HFile文件合并成大的HFile文件

9.随着HFile越来越大，达到一定阈值时就会水平分割，分成两个新的HFile，对应着两个region，
每个region由不同的HRegionServer进行管理（这个分配过程是由HMaster完成）region在分裂过程中不接受读写请求
```

11.HBase相关的工作机制

flush刷新机制（溢写合并机制）

```
flush刷新机制的目的：把memStore的数据最终写入到hdfs，形成storefile文件
文件阈值：大小阈值128M，时间阈值1小时，满足其中一个就会触发flush刷新机制
流程
HBase2.0之后flush流程
1.关闭当前的memStore空间，接着开启一个信息的memStore空间，接收正常写入
2.把关系的memStore放入到一个只读内存队列中，会让这个队列尽可能晚的刷新到磁盘，保证读取效率
3.随着队列中的memStore越来越多，当队列达到阈值时，触发flush操作，对队列中的数据进行处理，最后写入到磁盘。形成一个storefile文件。2.0之后才支持该操作
```

- 内存合并操作，在2.0之后支持内存合并操作，默认是不开启，如果要开启需要手动设置
  - 1.全局设置

```
修改hbase-site.xml文件
hbase.hregion.compacting.memstore.type
none|basic|eager|adaptive
```

- 合并方案
  - none：不合并
  - basic基础型：合并，不关心是否有重复过期的数据，直接合并，效率高

- eager饥渴型：合并时判断是否有重复过期的数据，清理掉重复过期数据
- adaptive：根据重复过期数据比例进行判断，超过阈值选择eager型否则选择basic

- 2.创建表时指定

```
create "test_memory_compaction", {NAME=>"C1", IN_MEMORY_COMPACTION=>"BASIC"}


hbase(main):004:0> create "test_memory_compaction", {NAME=>"C1",
IN_MEMORY_COMPACTION=>"BASIC"}
Created table test_memory_compaction
Took 1.6646 seconds
=> Hbase::Table - test_memory_compaction
hbase(main):005:0>
```

storeFile的合并机制

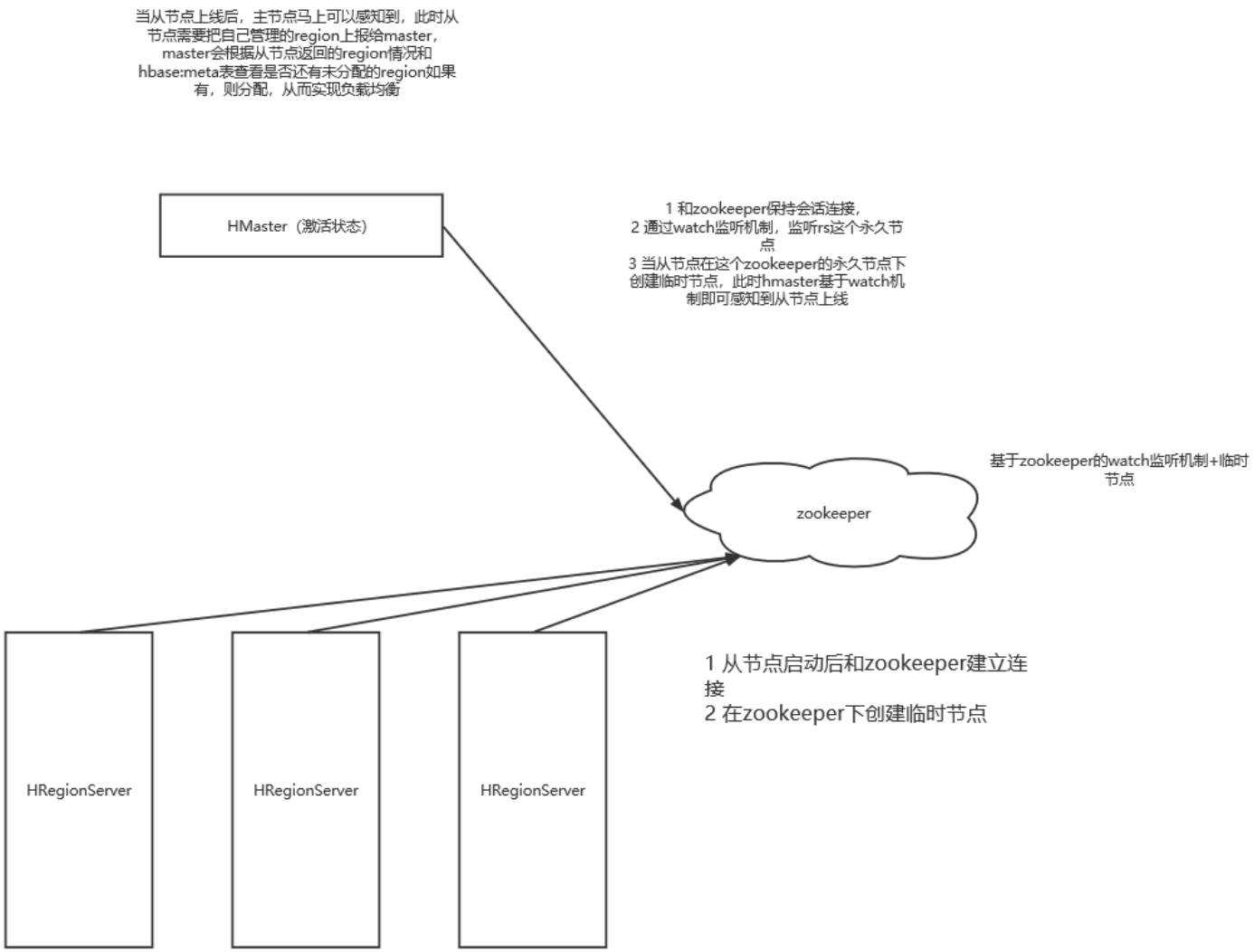
```
compact合并压缩机制
目的：把多个小的storeFile合并成大的hfile
阈值：storeFile达到3个及以上


合并过程中可以进行数据操作
minor
目的：把多个小的storeFile合并成一个较大的storeFile
流程：storeFile达到阈值，首先会合并小的，形成一个较大的storeFile
minor阶段不会对数据进行任何的删除，去重操作，仅仅进行基本的排序，合并工作，整体效率高

major
目的：是把上一步较大的storeFile合并成最终的hfile
默认情况：达到7天，或者手动执行，触发major操作，会把较大的hfile合并成最终的大Hfile
合并过程中重复，过期的数据，都会处理掉，时间长，对HBase产生影响，关闭掉定时触发机制，改成手动执行
```

split机制（region分裂）

```
分裂阈值计算公式：min（R^2 * memstoresize,hregion max filesize）
R表示的一个表内的region数据
memstoresize=128M
hregion max filesize 10G
```

```
5
6  处理值R=1
7  Min（1^2 *128M,10G)
8  Min(128M,10G)=128(这种机制是否合理)
9
10 R=2
11 Min（2^2*128,10G)
12
13 R=9
14 Min(9^2*128,10G)=10G
15 region/HFile按照10G进行分裂
```

regionServer的上线/下线流程（从节点）

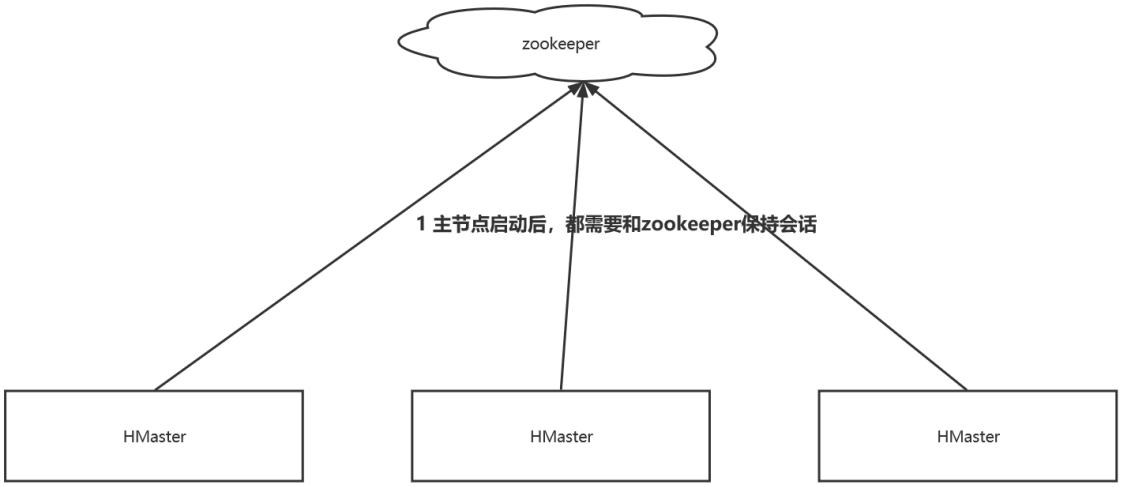


只要从节点下线，则在zookeeper中建立的临时节点就消失
主节点一直在监听rs永久节点，就会发现从节点下线

当从节点下线后，对应的从节点管理的这些region处于无人管理的状态，是未分配状态，此时主节点就需要把这些region重新分配给其他从节点。

如果刚下线的从节点，又上线，此时主节点就会从其他节点中分配一些region给它进行管理，但是不一定是刚从这里分走的

HMaster上线/下线（主节点）



2 各个主节点HMaster都向zookeeper /hbase 创建master的临时节点，谁先创建成功，谁就是激活状态的master

3 后来节点只能到backup-masters下创建临时节点，这些节点就处于备份状态
4 备份状态节点会监听激活状态的节点，一旦发现master删除，此时备份状态主节点创建master，谁先创建成功，谁是激活状态

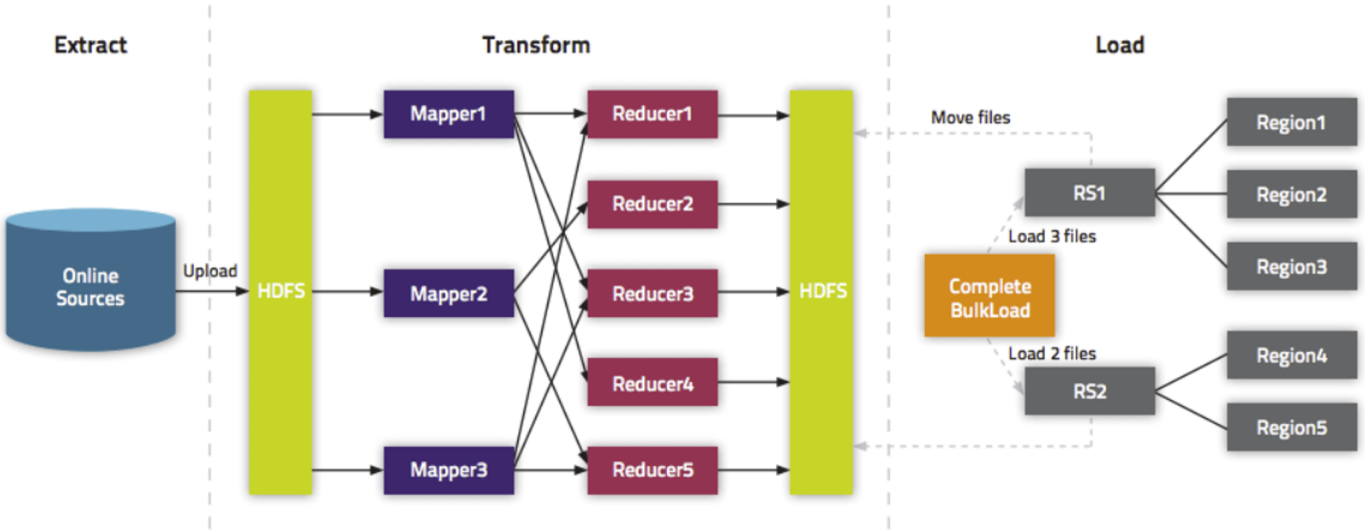
master短暂下线对集群影响，因为hbase读写操作不经过hmaster，大多数请求都是读写操作，master下线只会影响元数据的修改，创建表，修改表，删除表

12.Bulk Load 批量加载

- 假如有一批数据，需要写入到hbase中，数据量大，如果采用java api，流程：写入到HLOG memStore再从memStore到storeFile，storeFile合并到HFile，HFile有分裂。该过程占用大量资源，如果有别的请求，由于写入操作太多，其他请求就无法执行，
- 解决

```
1.把这一批数据直接转成HFile
2.把HFile导入到hbase，让hbase直接加载
跳过流程：写入HLOG memStore 再从memStore到storeFile，storeFile合并到HFile，HFile又分裂
```

- 解决的应用场景：需要一次性写入大量数据



1.1需求说明

- 目前有一份csv文件，里面有大量的转账数据，需要把数据存储到hbase中，原始数据太大，使用bulkload方式进行转换

id	ID
code	流水单号
rec_account	收款账户
rec_bank_name	收款银行
rec_name	收款人姓名
pay_account	付款账户
pay_name	付款人姓名
pay_comments	转账附言
pay_channel	转账渠道
pay_way	转账方式
status	转账状态
timestamp	转账时间
money	转账金额

1.2准备工作

- 需要在hbase中创建一张表

```
create "TRANSFER_RECORD", "C1"
```

- 把数据传到hdfs中，需要启动（hdfs yarn zookeeper hbase）

```
[root@node1 data]# hdfs dfs -mkdir -p /hbase/bulkload/input
[root@node1 data]# hdfs dfs -put bank_record.csv /hbase/bulkload/input/
```

1.3创建maven工程

- 导入依赖

```
<properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <repositories>
        <repository>
            <id>aliyun</id>
            <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
            <releases><enabled>true</enabled></releases>
            <snapshots><enabled>false</enabled></snapshots>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
            <version>2.1.0</version>
        </dependency>
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>2.6</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>6.14.3</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hbase</groupId>
```

```
            <artifactId>hbase-mapreduce</artifactId>
            <version>2.1.0</version>
        </dependency>

        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-jobclient</artifactId>
            <version>2.7.5</version>
        </dependency>

        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.4</version>
        </dependency>

        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactId>
            <version>2.7.4</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-auth</artifactId>
            <version>2.7.4</version>
        </dependency>

        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.4</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
```

```
                <configuration>
                    <target>1.8</target>
                    <source>1.8</source>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

1.4csv文件转成HFile文件

- **编写mapper代码**

```
package sz.base.habse;

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

import java.io.IOException;

public class mapper extends Mapper<LongWritable, Text, ImmutableBytesWritable, Put> {
    //ImmutableBytesWritable（接口）是一个数据类型一般作为RowKey的类型
    private ImmutableBytesWritable k2 = new ImmutableBytesWritable();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        //1.获取一行数据
        String line = value.toString();
        //2.对数据进行分割,进行判断如果不为空和null则为true
        if (line != null && !"".equals(line)) {
            //对每行数据进行切割，切割符号为：  ,
            String[] fields = line.split(",");
            //获取第0个元素，用作rowkey
            byte[] rowkey = fields[0].getBytes();
            //转换为ImmutableBytesWritable类型
            k2.set(rowkey);
            //3.构造put对象
```

```
            //rowkey是一个行健
            Put v2 = new Put(rowkey);
            //添加数据
            v2.addColumn("C1".getBytes(), "code".getBytes(), fields[1].getBytes());
            v2.addColumn("C1".getBytes(), "rec_account".getBytes(), fields[2].getBytes());
            v2.addColumn("C1".getBytes(), "rec_bank_name".getBytes(), fields[3].getBytes());
            v2.addColumn("C1".getBytes(), "rec_name".getBytes(), fields[4].getBytes());
            v2.addColumn("C1".getBytes(), "pay_account".getBytes(), fields[5].getBytes());
            v2.addColumn("C1".getBytes(), "pay_name".getBytes(), fields[6].getBytes());
            v2.addColumn("C1".getBytes(), "pay_comments".getBytes(), fields[7].getBytes());
            v2.addColumn("C1".getBytes(), "pay_channel".getBytes(), fields[8].getBytes());
            v2.addColumn("C1".getBytes(), "pay_way".getBytes(), fields[9].getBytes());
            v2.addColumn("C1".getBytes(), "status".getBytes(), fields[10].getBytes());
            v2.addColumn("C1".getBytes(), "timestamp".getBytes(), fields[11].getBytes());
            v2.addColumn("C1".getBytes(), "money".getBytes(), fields[12].getBytes());
            //4.数据写出去
            context.write(k2, v2);
        }
    }
}

```

- **编写driver代码**

```
package sz.base.java;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
```

```java
import org.apache.hadoop.hbase.mapreduce.HFileOutputFormat2;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.zookeeper.ZooKeeper;

public class BulkLoadDriver extends Configured implements Tool {
    @Override
    public int run(String[] strings) throws Exception {
        //1.创建job对象
        Job job = Job.getInstance(super.getConf(), "BulkLoadDriver");
        //2.设置yarn参数
        job.setJarByClass(BulkLoadDriver.class);
        //3.设置八步
        //3.1设置输入类，输入路径
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.addInputPath(job, new
Path("hdfs://node1:8020/hbase/bulkload/input/"));
        //3.2设置mapper，输出k2，v2
        job.setMapperClass(BulkLoadMapper.class);
        job.setMapOutputKeyClass(ImmutableBytesWritable.class);
        job.setMapOutputValueClass(Put.class);
        //3.3设置shuffle分区 排序 规约 分组
        //3.7设置reduce reduce输出 k3 , v3
        job.setNumReduceTasks(0);
        job.setOutputKeyClass(ImmutableBytesWritable.class);
        job.setOutputValueClass(Put.class);
        //3.8设置输出类 输出路径
        job.setOutputFormatClass(HFileOutputFormat2.class);
        //4.获取hbase连接对象
        Connection hbconn = ConnectionFactory.createConnection(super.getConf());
        System.out.println(hbconn);
        Table table = hbconn.getTable(TableName.valueOf("TRANSFER_RECORD"));
        //设置输出路径
        HFileOutputFormat2.configureIncrementalLoad(job, table,
hbconn.getRegionLocator(TableName.valueOf("TRANSFER_RECORD")));
        HFileOutputFormat2.setOutputPath(job, new
Path("hdfs://node1:8020/hbase/bulkload/output"));
        //提交任务
        boolean b = job.waitForCompletion(true);
```

```
51
52          return b ? 0 : 1;
53      }
54
55      public static void main(String[] args) throws Exception {
56          System.setProperty("HADOOP_USER_NAME", "root");
57          Configuration conf = HBaseConfiguration.create();
58          conf.set("hbase.zookeeper.quorum", "node1");
59          int run = ToolRunner.run(conf, new BulkLoadDriver(), args);
60          System.out.println(run);
61      }
62
63  }
64
```

- **可能出现的问题**

```
1  Mkdirs failed to create xxx/hbase-staging
```

- 在main函数中添加

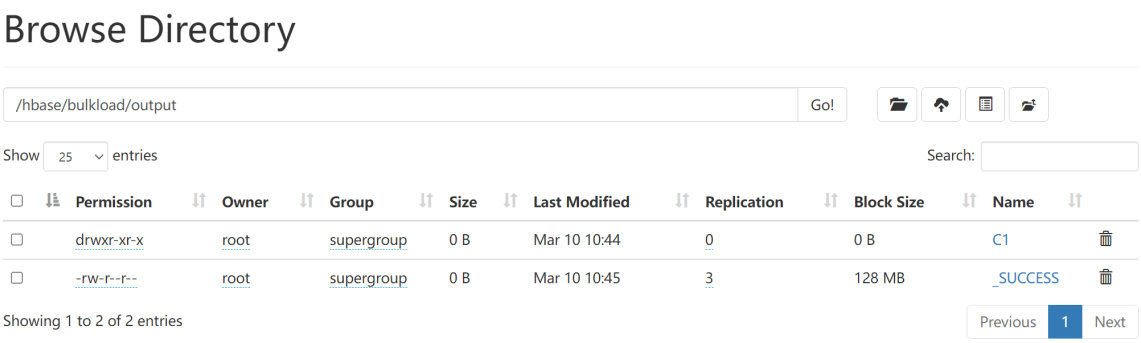
```
1  System.setProperty("HADOOP_USER_NAME", "root");
2  conf.set("fs.defaultFS", "hdfs://node1:8020");
```

- 修改配置文件
- /export/server/hadoop/etc/hadoop/core-site.xml

```
1  <property>
2      <name>fs.defaultFS</name>
3      <value>hdfs://node1.itcast.cn:8020</value>
4  </property>
```

1.5验证代码执行结果

```
1  [root@node1 hadoop]# hdfs dfs -ls /hbase/bulkload/output
2  Found 2 items
3  drwxr-xr-x   - root supergroup          0 2022-03-10 10:44 /hbase/bulkload/output/C1
4  -rw-r--r--   3 root supergroup          0 2022-03-10 10:45
   /hbase/bulkload/output/_SUCCESS
5
6  可以看到_SUCCESS表示执行成功
```



1.6把HFile格式文件加载到hbase中

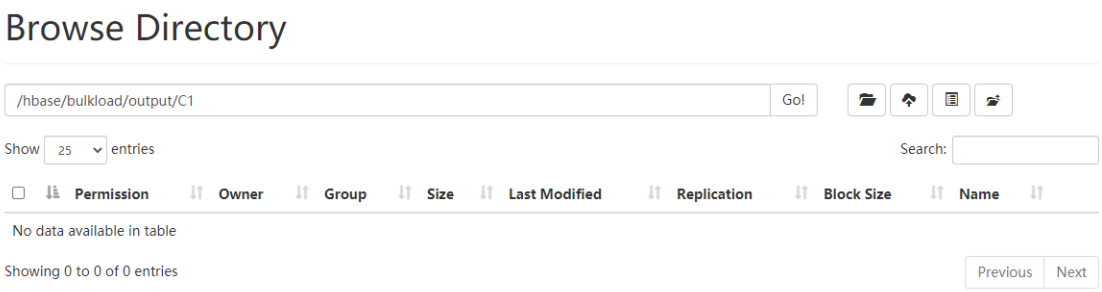
- 命令是在终端中运行，liunx中

```
格式 ： hbase org.apache.hadoop.hbase.tool.LoadIncrementalHFiles 代码产生的数据路径 Hbase表名
hbase org.apache.hadoop.hbase.tool.LoadIncrementalHFiles hdfs://node1:8020/hbase/bulkload/output/ TRANSFER_RECORD
```

- 验证数据

```
在hbase shell中统计表的行数
count "TRANSFER_RECORD"
和原始的csv文件数量一致
```

- 执行完导入命令后，通过Java代码生成的数据就被删除了。数据已经导入到hbase表



13.HBase和Hive的集成操作

1.hbase hive对比

- hive是一个数仓工具，基于hadoop数据存在datanade上，执行翻译把sql翻译成mr，主要离线分析中使用，延迟较高
- hbase数据存储容器，NoSQL数据库，基于hdfs，数据以HFile格式存在，不支持sql，延迟较低，有很高的随机读写能力
- hive hbase都是基于hadoop的不同的工具，可以同时使用，可以集成在一起使用
- 把hive hbase集成在一起，通过hive进行离线分析，通过hbase执行高效读写操作

2.hive和hbase集成操作

- 拷贝hive和hbase进行通信的包到hbase的lib目录

```
[root@node1 ~]# cd /export/server/hive/lib/
[root@node1 lib]# ll hive-hbase-handler-3.1.2.jar
-rw-r--r-- 1 root root 118060 Aug 23  2019 hive-hbase-handler-3.1.2.jar
[root@node1 lib]# cp hive-hbase-handler-3.1.2.jar /export/server/hbase-2.1.0/lib/
```

- 修改hive的配置文件

```
[root@node1 conf]# cd /export/server/hive/conf
[root@node1 conf]# vim hive-site.xml

添加以下内容

<property>
    <name>hive.zookeeper.quorum</name>
    <value>node1,node2,node3</value>
    </property>
    <property>
    <name>hbase.zookeeper.quorum</name>
    <value>node1,node2,node3</value>
    </property>
    <property>
    <name>hive.server2.enable.doAs</name>
    <value>false</value>
</property>

```

```
[root@node1 conf]# cd /export/server/hive/conf
[root@node1 conf]# vim hive-env.sh

添加以下内容

export HADOOP_HOME=/export/server/hadoop-3.3.0
export HIVE_CONF_DIR=/export/server/hive-3.1.2/conf
export HBASE_HOME=/export/server/hbase-2.1.0

```

3.分发jar包

```
[root@node1 conf]# cd /export/server/hbase-2.1.0/lib/
```

```
[root@node1 lib]# scp hive-hbase-handler-3.1.2.jar node2:$PWD
hive-hbase-handler-3.1.2.jar
      100%  115KB  37.1MB/s   00:00
[root@node1 lib]# scp hive-hbase-handler-3.1.2.jar node3:$PWD
hive-hbase-handler-3.1.2.jar
      100%  115KB  20.8MB/s   00:00
```

4.启动集群

```
1 启动hadoop集群
start-all.sh
或者
start-dfs.sh
start-yarn.sh

2 启动zookeeper，确保三台节点都启动
三台节点都要执行
zkServer.sh start

3 启动hbase集群，在node1上启动
start-hbase.sh

4 启动hive，需要把hive添加到环境变量中，修改/etc/profile文件，source /etc/profile
nohup hive --service metastore 2>&1 &
nohup hive --service hiveserver2 2>&1 &
```

- ## 5.创建hbase表（测试）

```
hbase(main):017:0> create "hbase_hive_score", "cf"
Created table hbase_hive_score
Took 0.7717 seconds

=> Hbase::Table - hbase_hive_score
hbase(main):018:0> put "hbase_hive_score", "1", "cf:name", "zhangsan"
Took 0.1243 seconds

hbase(main):019:0> put "hbase_hive_score", "1", "cf:score", "95"
Took 0.0068 seconds

hbase(main):020:0> put "hbase_hive_score", "2", "cf:name", "lisi"
```

```
Took 0.0070 seconds

hbase(main):021:0> put "hbase_hive_score", "2", "cf:score", "96"
Took 0.0053 seconds

hbase(main):022:0> put "hbase_hive_score", "3", "cf:name", "wangwu"
Took 0.0064 seconds

hbase(main):023:0> put "hbase_hive_score", "3", "cf:score", "97"
Took 0.0069 seconds

hbase(main):024:0>
```

6.创建hive表

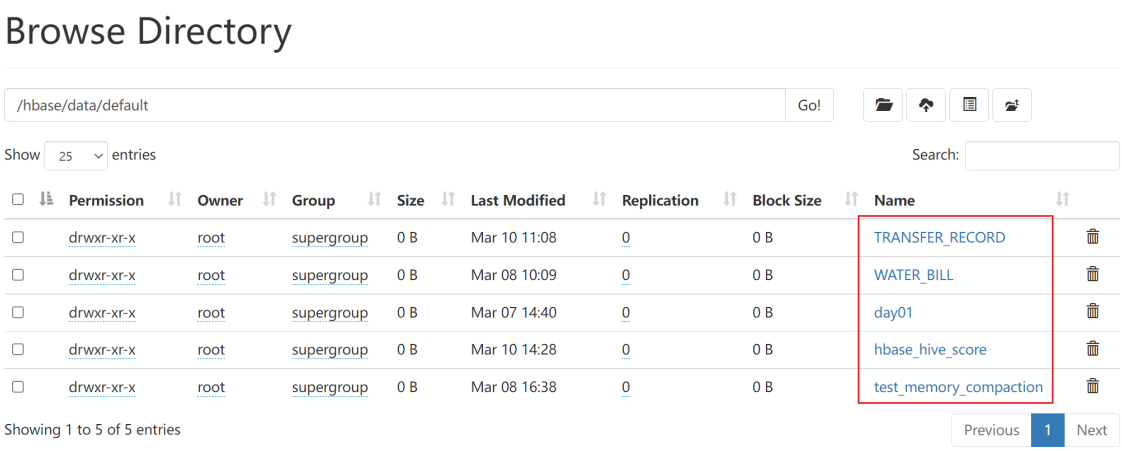
```
创建hive关联到hbase，这个hive表是一个外部表，

格式

create external table 数据库名.表名 (
字段1 类型,
字段2 类型,
字段3 类型,
....
) stored by "org.apache.hadoop.hive.hbase.HBaseStorageHandler" WITH SERDEPROPERTIES
("hbase.columns.mapping"=":key,列族1:列名1,列族2:列名2,...")
TBLPROPERTIES("hbase.table.name"="hbase表名");


例子

create external table hbase_hive_score (
id int,
name string,
score int
) stored by "org.apache.hadoop.hive.hbase.HBaseStorageHandler" WITH SERDEPROPERTIES
("hbase.columns.mapping"=":key,cf:name,cf:score")
TBLPROPERTIES("hbase.table.name"="hbase_hive_score");


查询
select * from hbase_hive_score;
```

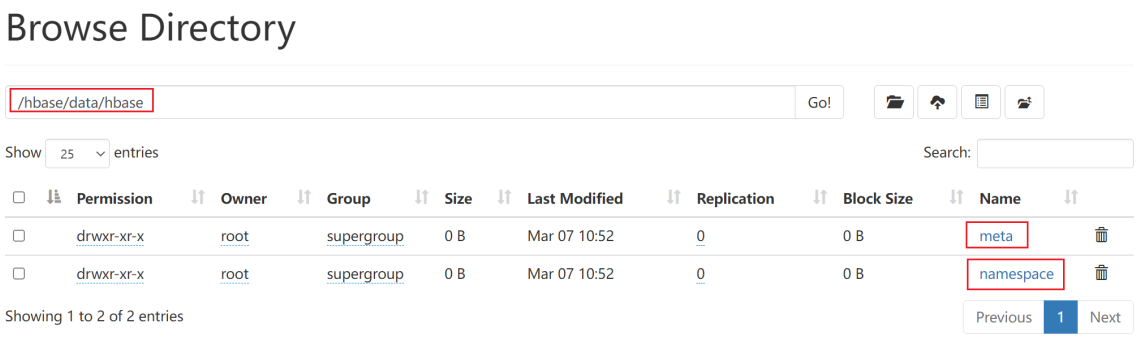
	id	name	score
1	1	zhangsan	97
2	2	lisi	90
3	3	wangwu	98

14.HBase表结构设计

1.hbase名称空间/命名空间

- 问题：MySQL hive中为什么创建数据库
  - 便于管理和维护
  - 业务划分更加明确
  - 进行权限管理
- 在hbase中，名称空间/命名空间也有同样作用
- hbase中namespace作用就是类似于MySQL hive 中数据库的作用，只不过在hbase中叫做namespace
- hbase有两个默认的名称空间 defalut和hbase





- default名称空间：默认名称空间，创建表时如果没有指定名称空间，则都是在default下创建
  - hive有一个default数据库
- hbase名称空间：专门防止hbase内部管理表的，一般不会操作，有hbase自己维护
  - meta表，就是hbase元数据表
  - 极端情况，比如系统出问题，，某些表无法删除，可以操作hbase：meta表

2.名称空间命令

```
1 创建名称空间
create_namespace "名称空间"
create_namespace "day03_space"

2 查看名称空间
list_namespace

3 查看名称空间信息
describe_namespace "名称空间"
describe_namespace "day03_space"

4 在某个名称空间中创建表
create "名称空间:表名", "列族"...
创建表时指定名称空间，后续所有操作都需要带名称空间，否则hbase会到default中找这个表

5 删除名称空间
drop_namespace "名称空间"
必须确保这个名称空间没有表
```

- ## 3.hbase表列族设计

- 原则：建表时，如果能用一个列族解决的，坚决只用一个，能少则少
- 原因：创建表时列族过多，region就会有多个store模块，每个store模块又由memStore和storeFile构成，导致读写数据时需要跨多个内存和文件，增大io操作，效率就降低，列族过多会导致region刷新触发compact压缩
- 什么场景使用多个列族
  - 假设有一张表，字段比较多，有些字段常用，可把性质相同的列放在一个列族中。如果把所有的数据都放在一个列族，在执行扫描操作时会把所有数据都取出来，降低效率
  - 假设一张表，字段比较多，对接同的业务，不同业务创建不同列族，各个业务之间读取给自的数据，互不影响
  - 2-5列族

4.hbase压缩方案

压缩算法	压缩后占比	压缩	解压缩
GZIP	13.4%	21 MB/s	118 MB/s
LZO	20.5%	135 MB/s	410 MB/s
Zippy/Snappy	22.2%	172 MB/s	409 MB/s

- 生产中如何选择

```
1 要压缩，如果数据频繁读写，可以选择snappy算法
2 要压缩，如果数据需要大量写入，但是读取很少时，采用gzip压缩算法
```

- 默认情况，创建表时不压缩

```
hbase(main):040:0> describe "WATER_BILL"
Table WATER_BILL is ENABLED
WATER_BILL
COLUMN FAMILIES DESCRIPTION
{NAME => 'C1', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE',
 CACHE_DATA_ON_WRITE => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLICATION_SCOPE => '0',
BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON
_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}
1 row(s)
Took 0.0722 seconds
hbase(main):041:0>
```

- 指定压缩方案

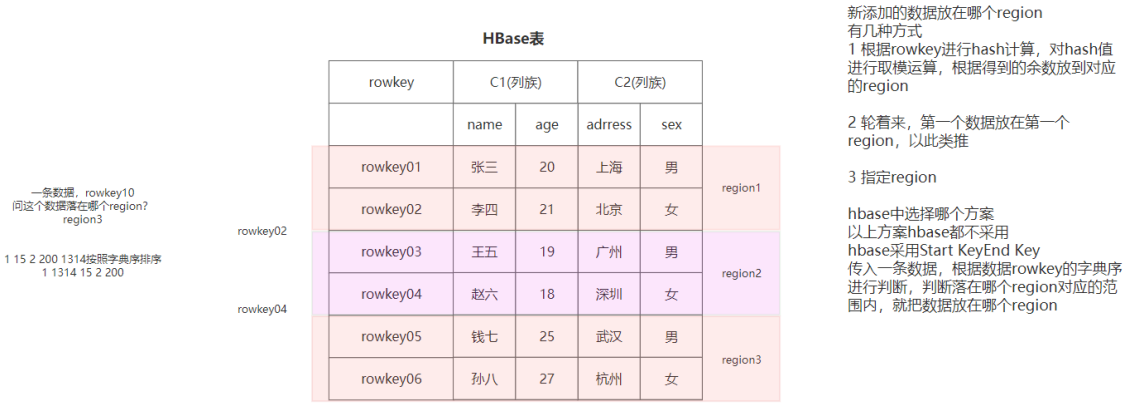
```
格式
创建表时
create "表名", {NAME=>"列族", COMPRESSION=>"GZ|SNAPPY|LZO"}

create "day03", {NAME=>"C1", COMPRESSION=>"GZ"} # 压缩方案是针对列族

对于已经存在的表，需要修改压缩方案，使用alter命令
alter "表名", {NAME=>"列族", COMPRESSION=>"GZ|SNAPPY|LZO"}

如果使用snappy压缩，需要重新编译hbase，让hbase支持snappy压缩
使用lzo压缩，需要把lzo的jar放在指定环境中

```

default	TRANSFER_RECORD	1	0	0	0	0	'TRANSFER_RECORD', {NAME => 'C1'}
default	WATER_BILL	1	0	0	0	0	'WATER_BILL', {NAME => 'C1'}
default	day01	1	0	0	0	0	'day01', {NAME => 'f1'}, {NAME => 'f2'}
default	day03	1	0	0	0	0	'day03', {NAME => 'C1', COMPRESSION => 'GZ'}
default	hbase_hive_score	1	0	0	0	0	'hbase_hive_score', {NAME => 'cf'}
default	test_memory_compaction	1	0	0	0	0	'test_memory_compaction', {NAME => 'C1', METADATA => {'IN_MEMORY_COMPACTION' => 'BASIC'}}

5.hbase表的预分区

- 默认情况，hbase一个表只有一个region，而一个region只能由一个regionserver进行管理
- 如果一开始来了大量读写操作，这个regionserver可能会宕机，因为这个regionserver管理所有的读写情况
- 如果解决

```
如果创建表时，一次性构建多个region，多个region、均匀分布在regionserver上，此时来了大量的读写请求，这些请求就可以分发达到不同regionserver，就解决了这个问题
```



- 如何实现预分区
  - **手动设定分区边界**

```
格式
create "表名", "列族", SPLITS=>[分区边界值]

create "test3", "C1", SPLITS=>['10', '20', '30', '40']
这里是四个边界值，得到五个分区
左闭右开['10', '20') '10'rowkey落在这个分区，'20'这个rowkey落在下一个分区

'200' rowkey落在哪个分区？落在第2个分区
```

Table Regions

Sort As RegionName Ascending ShowDetailName&Start/End Key Reorder

Name(5)	Region Server	ReadRequests (0)	WriteRequests (0)	StorefileSize (0 B)	Num.Storefiles (0)	MemSize (0 B)	Locality	Start Key	End Key
test3,,1646901103780.afa2495fec31fd5fdefe71cca2e63e76.	node3:16030	0	0	0 B	0	0 B	0.0		10
test3,10,1646901103780.c926ae12092b33e274b9ac982659e820.	node2:16030	0	0	0 B	0	0 B	0.0	10	20
test3,20,1646901103780.5c9a864b9cc5b2c7efaff1ebab4472a8.	node3:16030	0	0	0 B	0	0 B	0.0	20	30
test3,30,1646901103780.675797d56d07dca9cf4727605f435a32.	node2:16030	0	0	0 B	0	0 B	0.0	30	40
test3,40,1646901103780.88c1d12287d2d24aea9bf6bec1a13d6f.	node2:16030	0	0	0 B	0	0 B	0.0	40	

- **hash分区方案**

```
格式
create "表名", "列族", {NUMREGIONS=>N, SPLITALGO=>"HexStringSplit"}
通过HexStringSplit算法确定分区边界

create "test04", "C1",  {NUMREGIONS=>16, SPLITALGO=>"HexStringSplit"}
```

Table Regions

Sort As RegionName Ascending ShowDetailName&Start/End Key Reorder

Name(16)	Region Server	ReadRequests (0)	WriteRequests (0)	StorefileSize (0 B)	Num.Storefiles (0)	MemSize (0 B)	Locality	Start Key	End Key
test04,,1646901575103.0eb469aef8a4393d984fbfd60719b26b.	node2:16030	0	0	0 B	0	0 B	0.0		10000000
test04,10000000,1646901575103.ba8683913a5328d228eab690deb75218.	node2:16030	0	0	0 B	0	0 B	0.0	10000000	20000000
test04,20000000,1646901575103.491d7cdbd165676742bde321ff204d38.	node3:16030	0	0	0 B	0	0 B	0.0	20000000	30000000
test04,30000000,1646901575103.7b3c941770ba2f5b1e7af01db47cfe08.	node2:16030	0	0	0 B	0	0 B	0.0	30000000	40000000
test04,40000000,1646901575103.fc66890d5977864fe9ea3891a84ba8ae.	node3:16030	0	0	0 B	0	0 B	0.0	40000000	50000000
test04,50000000,1646901575103.3b314e131ebf39ed3318aaf49496eae2.	node2:16030	0	0	0 B	0	0 B	0.0	50000000	60000000
test04,60000000,1646901575103.e1eb3403d0ecf64869aee7eef29b9c74.	node3:16030	0	0	0 B	0	0 B	0.0	60000000	70000000
test04,70000000,1646901575103.c2bc94851367b0d4a64b3057f13c996c.	node3:16030	0	0	0 B	0	0 B	0.0	70000000	80000000
test04,80000000,1646901575103.5f83ef97c6802b7abb87b48ed33ac2c5.	node2:16030	0	0	0 B	0	0 B	0.0	80000000	90000000
test04,90000000,1646901575103.1dd765d253fa7884231c24694b026108.	node2:16030	0	0	0 B	0	0 B	0.0	90000000	a0000000
test04,a0000000,1646901575103.f2d3f1eee8a40432581bcd8ad5a37900.	node2:16030	0	0	0 B	0	0 B	0.0	a0000000	b0000000
test04,b0000000,1646901575103.9149ecc82908e171d12a89313ef565f0.	node3:16030	0	0	0 B	0	0 B	0.0	b0000000	c0000000
test04,c0000000,1646901575103.662e10bab4c02f6e2db6d1c940f21bba.	node3:16030	0	0	0 B	0	0 B	0.0	c0000000	d0000000
test04,d0000000,1646901575103.cc3ee34671f4b8a399eefc6f7c9ec6f4.	node3:16030	0	0	0 B	0	0 B	0.0	d0000000	e0000000
test04,e0000000,1646901575103.65af9ec73b32346da91c426f3f650862.	node2:16030	0	0	0 B	0	0 B	0.0	e0000000	f0000000

6.hbase中rowkey设计原则

- 问题：前面的预分区方案，有没有问题？

```
前面进行预分区目的？期望保证负载均衡，但是完全依靠预分区可能没办法解决负载不均衡问题。如果rowkey没有设计好，会出现所有的数据落在一个region中
```

- 解决方案：在预分区的基础上，设计好rowkey保证数据均匀分布在不同region上，如果rowkey设计不好，会导致数据不均衡
- 设置rowkey原则

```
1 避免使用递增的行键/时序数据
0 1 2 3 4 ...
~ 10000
10000 ~ 20000

时间戳，最前面那一位需要很久才会变化，导致数据集中在前面的分区中
2 避免使用rowkey和列名过长
因为列名和rowkey和数据一起存在hfile中，如果太长，相应数据就存不了太多，导致提前flush操作

rowkey 支持64k byte
一般建议100个字节，大部分在10-20个字节

3 使用long型比string更节省空间
100

"100" 占 3字节
100 占 一个字节= 8 bit 无符号 0-255 ，1个字节就可以存
```

```
4 保证rowkey唯一性
如果不唯一，也不会报错，只会把原来数据覆盖掉
```

- 如果避免出现热点数据
  - 热点是指大量的客户端直接访问集群的一个或者几个节点（读写），大量访问量可能会使得某个服务器节点超负荷，导致整个regionserver性能下降，其他的region也会受影响
  - 反转操作
    - 反转策略可以使rowkey随机分布，但是牺牲了rowkey的有序性
    - 比如手机号码，12345678，翻转：87654321
    - 缺点：利于get操作，但不利于scan操作，因为数据在原rowkey上的自然顺序已经被打乱
  - 加盐策略
    - 在rowkey钱加上随机字符串或者随机数
    - 随机数能保障数据在所有region件的负载均衡
    - 如果采用上面两种策略，对于相关性很强的数据，就会被打散到不同的region
      - 比如：2022031001-2022031999 2022032001-2022032999
  - 哈希策略
    - 内容相同的部分进行hash运算，得到的结果一致的，再把这个hash和其他部分进行拼接
    - 这样可以得到分布均匀并且相关性强的数据分在一起

15.hbase版本确界

- 版本确界，在hbase中历史数据的版本，定义数据的历史版本保留多少个
- 下界：MIN_VERSIONS，hbase中单元格(cell)至少要保留几个版本，即时数据已经过期，默认值0（禁用），和TTL相结合，即只有一个值写入cell
- 上界：VERSIONS，hbase中单元格最多保留几个版本，如果版本比设置的VERSIONS大，最早进入系统的数据就会被覆盖，默认值是1（基于时间戳）

1.TTL

- Time to live ，数据存活时间，hbase中可以针对数据设置过期时间，时间过期后，hbase会自动清晰数据
- 代码实现

```java
package sz.base.java.hbase.version_ttl;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.CellUtil;
```

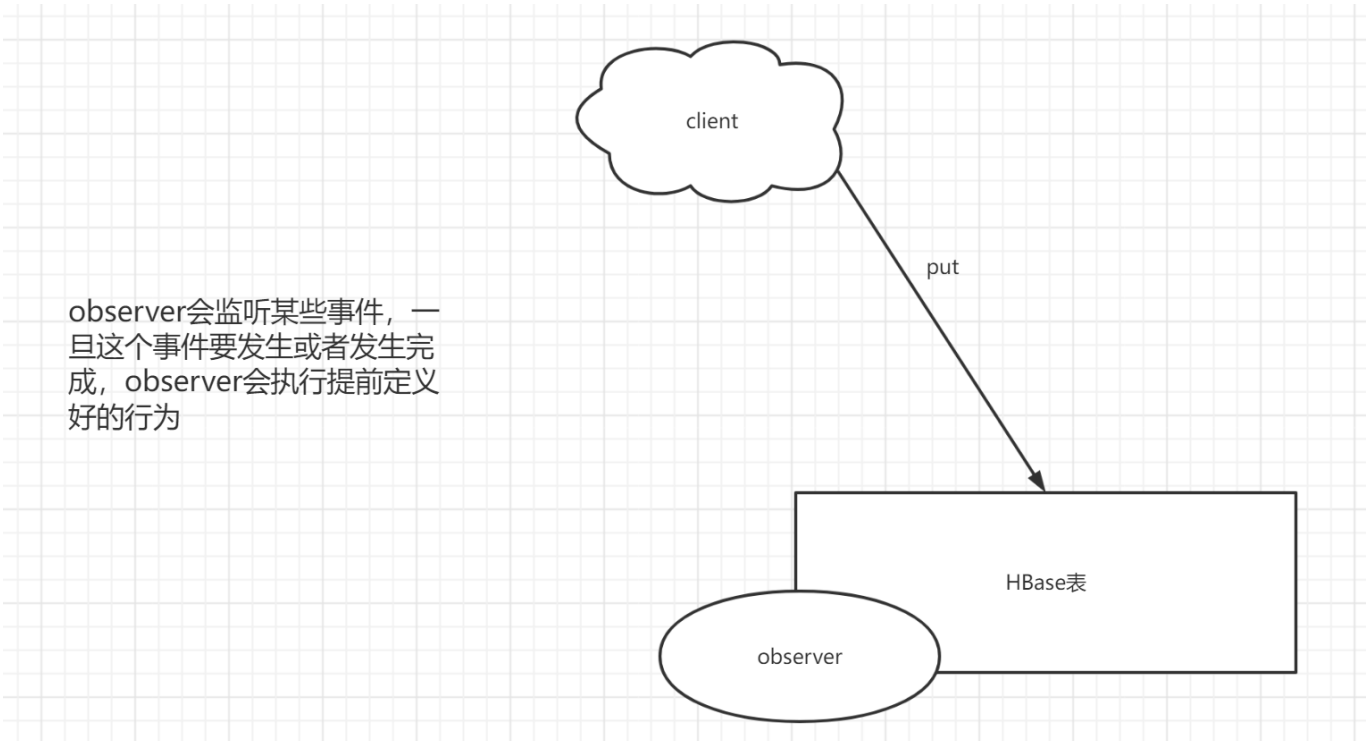
```
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBaseVersionTTLTest {
    //1.创建一个hbase表，设置上界和下界 ttl
    //2.添加数据，进行修改操作，让数据达到设置的上界
    //3.查询数据，观察现象
    public static void main(String[] args) throws IOException {
        //1.获取hbase链接对象
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", "node1");
        Connection hbcon = ConnectionFactory.createConnection(conf);
        Admin admin = hbcon.getAdmin();
        boolean flag = admin.tableExists(TableName.valueOf("day03"));
        if (!flag) {
            //如果执行代码到这里，表示不存在 把c1转换成byte
            //3.1创建列族相关的构造器，设置列族名
            ColumnFamilyDescriptorBuilder columnFamilyDescriptorBuilder = ColumnFamilyDescriptorBuilder.newBuilder("C1".getBytes());
            columnFamilyDescriptorBuilder.setMinVersions(3);//设置下界
            columnFamilyDescriptorBuilder.setMaxVersions(5);//设置上界
            //下界30秒，过期数据会被清理，下界设置了3个，则会保留最新3条数据
            columnFamilyDescriptorBuilder.setTimeToLive(30);//TTl 30秒
            ColumnFamilyDescriptor build1 = columnFamilyDescriptorBuilder.build();
            //3.2创建表相关的构造器，设置表名和列族名
            TableDescriptorBuilder mater_bill = TableDescriptorBuilder.newBuilder(TableName.valueOf("day03"));
            //设置列族名
            mater_bill.setColumnFamily(build1);
            //3.3创建构造器和表构造器关联，确定列族和表名
            TableDescriptor build = mater_bill.build();
            //3.4创建表
            admin.createTable(build);
        }
        Table table = hbcon.getTable(TableName.valueOf("day03"));
        //添加数据,上界为5添加6条
```

```
44 //        for (int i = 0; i <= 6; i++) {
45 //            Put put = new Put("rk001".getBytes());
46 //            String name = "zhangsan" + i;
47 //            System.out.println(name);
48 //            put.addColumn("C1".getBytes(), "NAME".getBytes(), name.getBytes());
49 //            table.put(put);
50 //        }
51         //查询 数据
52         Get get = new Get("rk001".getBytes());
53         get.readAllVersions();//执行这个函数才能把所有符合规则的版本都获取到
54         Result result = table.get(get);
55 
56         //打印结果
57         Cell[] cells = result.rawCells();
58         for (Cell cell : cells) {
59             System.out.println(Bytes.toString(CellUtil.cloneValue(cell)));
60         }
61     }
62 }
```

16.HBase的协处理器

1.协处理器的基本介绍

- observer协处理器



- endpoint

```
协处理器是可以看做定义了一个方法，把这个方法放置再regionserver上运行，各个regionserver运行后把结果返回给客户端
聚合操作 max sum count
```

2.如何设置协处理器

- ## 1.静态的全局设置
  - 在hbase-site.xml文件中进行设置，全局有效，设置后对每个hbase表都会有协处理器

```
<property>
    <name>hbase.coprocessor.user.region.classes</name>
    <value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
</property>
```

- ## 2.动态设置，只针对某个表有效
  - 禁用表
  - 添加协处理器

```
hbase> alter 'mytable', METHOD => 'table_att','coprocessor' =>
'|org.apache.Hadoop.hbase.coprocessor.AggregateImplementation||'
```

  - 启用表
- ## 3.如何卸载协处理器
  - 禁用表
  - 删除协处理器

```
alter 'test' , METHOD => 'table_att_unset',NAME=> 'coprocessor$1'
```

  - 启用表
- # 17.HBase调优