HBase

第一章 简介

Hadoop的局限

HBase 是一个构建在 Hadoop 文件系统之上的面向列的数据库管理系统。

要想明白为什么产生 HBase，就需要先了解一下 Hadoop 存在的限制？Hadoop 可以通过 HDFS 来存储结构化、半结构甚至非结构化的数据，它是传统数据库的补充，是海量数据存储的最佳方法，它针对大文件的存储，批量访问和流式访问都做了优化，同时也通过多副本解决了容灾问题。

但是 Hadoop 的缺陷在于它只能执行批处理，并且只能以顺序方式访问数据，这意味着即使是最简单的工作，也必须搜索整个数据集，无法实现对数据的随机访问。实现数据的随机访问是传统的关系型数据库所擅长的，但它们却不能用于海量数据的存储。在这种情况下，必须有一种新的方案来解决海量数据存储和随机访问的问题，HBase 就是其中之一 (HBase，Cassandra，couchDB，Dynamo 和 MongoDB 都能存储海量数据并支持随机访问)。

> 注：数据结构分类：
>
> - 结构化数据：即以关系型数据库表形式管理的数据；
> - 半结构化数据：非关系模型的，有基本固定结构模式的数据，例如日志文件、XML 文档、JSON 文档、Email 等；
> - 非结构化数据：没有固定模式的数据，如 WORD、PDF、PPT、EXL，各种格式的图片、视频等。

HBase简介

HBase 是一个构建在 Hadoop 文件系统之上的面向列的数据库管理系统。

HBase 是一种类似于 `Google’s Big Table` 的数据模型，它是 Hadoop 生态系统的一部分，它将数据存储在 HDFS 上，客户端可以通过 HBase 实现对 HDFS 上数据的随机访问。它具有以下特性：

- 不支持复杂的事务，只支持行级事务，即单行数据的读写都是原子性的；
- 由于是采用 HDFS 作为底层存储，所以和 HDFS 一样，支持结构化、半结构化和非结构化的存储；
- 支持通过增加机器进行横向扩展；
- 支持数据分片；
- 支持 RegionServers 之间的自动故障转移；
- 易于使用的 Java 客户端 API；
- 支持 BlockCache 和布隆过滤器；
- 过滤器支持谓词下推。

HBase Table

HBase 是一个面向 `列` 的数据库管理系统，这里更为确切的而说，HBase 是一个面向 `列族` 的数据库管理系统。表 schema 仅定义列族，表具有多个列族，每个列族可以包含任意数量的列，列由多个单元格（cell ）组成，单元格可以存储多个版本的数据，多个版本数据以时间戳进行区分。

下图为 HBase 中一张表的：

- RowKey 为行的唯一标识，所有行按照 RowKey 的字典序进行排序；
- 该表具有两个列族，分别是 personal 和 office;
- 其中列族 personal 拥有 name、city、 phone 三个列，列族 office 拥有 tel、addres 两个列。

Hbase 的表具有以下特点：

- 容量大：一个表可以有数十亿行，上百万列；
- 面向列：数据是按照列存储，每一列都单独存放，数据即索引，在查询时可以只访问指定列的数据，有效地降低了系统的 I/O 负担；
- 稀疏性：空 (null) 列并不占用存储空间，表可以设计的非常稀疏 ；
- 数据多版本：每个单元中的数据可以有多个版本，按照时间戳排序，新的数据在最上面；
- 存储类型：所有数据的底层存储格式都是字节数组 (byte[])。

Phoenix

`Phoenix` 是 HBase 的开源 SQL 中间层，它允许你使用标准 JDBC 的方式来操作 HBase 上的数据。在 `Phoenix` 之前，如果你要访问 HBase，只能调用它的 Java API，但相比于使用一行 SQL 就能实现数据查询，HBase 的 API 还是过于复杂。 `Phoenix` 的理念是 `we put sql SQL back in NOSQL` ，即你可以使用标准的 SQL 就能完成对 HBase 上数据的操作。同时这也意味着你可以通过集成 `Spring Data JPA` 或 `Mybatis` 等常用的持久层框架来操作 HBase。

其次 `Phoenix` 的性能表现也非常优异， `Phoenix` 查询引擎会将 SQL 查询转换为一个或多个 HBase Scan，通过并行执行来生成标准的 JDBC 结果集。它通过直接使用 HBase API 以及协处理器和自定义过滤器，可以为小型数据查询提供毫秒级的性能，为千万行数据的查询提供秒级的性能。同时 Phoenix 还拥有二级索引等 HBase 不具备的特性，因为以上的优点，所以 `Phoenix` 成为了 HBase 最优秀的 SQL 中间层。

第二章 基本概念

Row Key

`Row Key` 是用来检索记录的主键。想要访问 HBase Table 中的数据，只有以下三种方式：

- 通过指定的 `Row Key` 进行访问；
- 通过 Row Key 的 range 进行访问，即访问指定范围内的行；
- 进行全表扫描。

`Row Key` 可以是任意字符串，存储时数据按照 `Row Key` 的字典序进行排序。这里需要注意以下两点：

- 因为字典序对 Int 排序的结果是 1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,…,9,91,92,93,94,95,96,97,98,99。如果你使用整型的字符串作为行键，那么为了保持整型的自然序，行键必须用 0 作左填充。

- 行的一次读写操作时原子性的 (不论一次读写多少列)。

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，所以列族需要在创建表时进行定义。列族的所有列都以列族名作为前缀，例如 `courses:history` ， `courses:math` 都属于 `courses` 这个列族。

Column Qualifier (列限定符)

列限定符，你可以理解为是具体的列名，例如 `courses:history` ， `courses:math` 都属于 `courses` 这个列族，它们的列限定符分别是 `history` 和 `math` 。需要注意的是列限定符不是表 Schema 的一部分，你可以在插入数据的过程中动态创建列。

Column(列)

HBase 中的列由列族和列限定符组成，它们由 `:` (冒号) 进行分隔，即一个完整的列名应该表述为 `列族名 : 列限定符` 。

Cell

`Cell` 是行，列族和列限定符的组合，并包含值和时间戳。你可以等价理解为关系型数据库中由指定行和指定列确定的一个单元格，但不同的是 HBase 中的一个单元格是由多个版本的数据组成的，每个版本的数据用时间戳进行区分。

Timestamp(时间戳)

HBase 中通过 `row key` 和 `column` 确定的为一个存储单元称为 `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，也可以由客户显式指定。每个 `Cell` 中，不同版本的数据按照时间戳倒序排列，即最新的数据排在最前面。

第三章 常用Shell命令

基本命令

打开 Hbase Shell：

```
1    # hbase shell
```

获取帮助

```
# 获取帮助
help
# 获取命令的详细信息
help 'status'
```

查看服务器状态

```
status
```

查看版本信息

```
version
```

关于表的操作

查看所有表

```
list
```

创建表

命令格式： create '表名称', '列族名称 1','列族名称 2','列名称 N'

```
# 创建一张名为Student的表,包含基本信息（baseInfo）、学校信息（schoolInfo）两个列族
create 'Student','baseInfo','schoolInfo'
```

查看表的基本信息

命令格式：desc '表名'

```
describe 'Student'
```

表的启用/禁用

enable 和 disable 可以启用/禁用这个表,is_enabled 和 is_disabled 来检查表是否被禁用

```
# 禁用表
disable 'Student'
# 检查表是否被禁用
is_disabled 'Student'
# 启用表
enable 'Student'
# 检查表是否被启用
is_enabled 'Student'
```

检查表是否存在

```
exists 'Student'
```

删除表

```
# 删除表前需要先禁用表
disable 'Student'
# 删除表
drop 'Student'
```

增删改

添加列族

命令格式： alter '表名', '列族名'

```
alter 'Student', 'teacherInfo'
```

删除列族

命令格式：alter '表名', {NAME => '列族名', METHOD => 'delete'}

```
alter 'Student', {NAME => 'teacherInfo', METHOD => 'delete'}
```

更改列族存储版本的限制

默认情况下，列族只存储一个版本的数据，如果需要存储多个版本的数据，则需要修改列族的属性。修改后可通过 `desc` 命令查看。

```
alter 'Student',{NAME=>'baseInfo',VERSIONS=>3}
```

插入数据

命令格式：put '表名', '行键','列族:列','值'

注意：如果新增数据的行键值、列族名、列名与原有数据完全相同，则相当于更新操作

```
put 'Student', 'rowkey1','baseInfo:name','tom'
put 'Student', 'rowkey1','baseInfo:birthday','1990-01-09'
put 'Student', 'rowkey1','baseInfo:age','29'
put 'Student', 'rowkey1','schoolInfo:name','Havard'
put 'Student', 'rowkey1','schoolInfo:localtion','Boston'

put 'Student', 'rowkey2','baseInfo:name','jack'
put 'Student', 'rowkey2','baseInfo:birthday','1998-08-22'
put 'Student', 'rowkey2','baseInfo:age','21'
put 'Student', 'rowkey2','schoolInfo:name','yale'
put 'Student', 'rowkey2','schoolInfo:localtion','New Haven'

put 'Student', 'rowkey3','baseInfo:name','maike'
put 'Student', 'rowkey3','baseInfo:birthday','1995-01-22'
put 'Student', 'rowkey3','baseInfo:age','24'
put 'Student', 'rowkey3','schoolInfo:name','yale'
put 'Student', 'rowkey3','schoolInfo:localtion','New Haven'

put 'Student', 'wrowkey4','baseInfo:name','maike-jack'
```

获取指定行、指定行中的列族、列的信息

```
# 获取指定行中所有列的数据信息
get 'Student','rowkey3'
# 获取指定行中指定列族下所有列的数据信息
get 'Student','rowkey3','baseInfo'
# 获取指定行中指定列的数据信息
get 'Student','rowkey3','baseInfo:name'
```

删除指定行、指定行中的列

```
# 删除指定行
deleteall 'Student','rowkey3'
# 删除指定行中指定列的数据
delete 'Student','rowkey3','baseInfo:name'
```

查询

hbase 中访问数据有两种基本的方式：

- 按指定 rowkey 获取数据：get 方法；
- 按指定条件获取数据：scan 方法。

`scan` 可以设置 begin 和 end 参数来访问一个范围内所有的数据。get 本质上就是 begin 和 end 相等的一种特殊的 scan。

Get查询

```
# 获取指定行中所有列的数据信息
get 'Student','rowkey3'
# 获取指定行中指定列族下所有列的数据信息
get 'Student','rowkey3','baseInfo'
# 获取指定行中指定列的数据信息
get 'Student','rowkey3','baseInfo:name'
```

查询整表数据

```
scan 'Student'
```

查询指定列簇的数据

```
scan 'Student', {COLUMN=>'baseInfo'}
```

条件查询

```
# 查询指定列的数据
scan 'Student', {COLUMNS=> 'baseInfo:birthday'}
```

除了列 `（COLUMNS）` 修饰词外，HBase 还支持 `Limit` （限制查询结果行数），`STARTROW` （`ROWKEY` 起始行，会先根据这个 `key` 定位到 `region`，再向后扫描）、`STOPROW` (结束行)、`TIMERANGE` （限定时间戳范围）、`VERSIONS` （版本数）、和 `FILTER` （按条件过滤行）等。

如下代表从 `rowkey2` 这个 `rowkey` 开始，查找下两个行的最新 3 个版本的 name 列的数据：

```
scan 'Student', {COLUMNS=> 'baseInfo:name',STARTROW => 'rowkey2',STOPROW => 'wrowkey4',LIMIT=>2,
VERSIONS=>3}
```

条件过滤

Filter 可以设定一系列条件来进行过滤。如我们要查询值等于 24 的所有数据：

```
scan 'Student', FILTER=>"ValueFilter(=,'binary:24')"
```

值包含 yale 的所有数据：

```
scan 'Student', FILTER=>"ValueFilter(=,'substring:yale')"
```

列名中的前缀为 birth 的：

```
scan 'Student', FILTER=>"ColumnPrefixFilter('birth')"
```

FILTER 中支持多个过滤条件通过括号、AND 和 OR 进行组合：

```
# 列名中的前缀为birth且列值中包含1998的数据
scan 'Student', FILTER=>"ColumnPrefixFilter('birth') AND ValueFilter
ValueFilter(=,'substring:1998')"
```

`PrefixFilter` 用于对 Rowkey 的前缀进行判断：

```
scan 'Student', FILTER=>"PrefixFilter('wr')"
```

第四章 HBase的JavaAPI

1 单例连接

```
import org.apache.flink.table.filesystem.FileSystemOutputFormat;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.AsyncConnection;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.CompletableFuture;

/**
 * Project:  BigDataCode
 * Create date:  2023/6/20
 * Created by fujiahao
 */

/**
 * 单线程使用连接
 */

public class HBaseConnection {

    // 声明静态属性
    public static Connection connection = null;
```

```
    static {
        // 创建连接
        // 默认使用同步连接
        try {
            // 使用读取本地文件的形式添加参数 hbase-site.xml
            connection = ConnectionFactory.createConnection();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void closeConnection() throws IOException {
        // 判断连接是否为空值
        if (connection != null){
            connection.close();
        }
    }

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

        /*// 创建连接配置对象
        Configuration conf = new Configuration();
        conf.set("hbase.zookeeper.quorum", "master,slave1,slave2");

        // 创建连接
        // 默认使用同步连接
        Connection connection = ConnectionFactory.createConnection(conf);

        // 可以使用异步连接
        // 不推荐
        // CompletableFuture<AsyncConnection> asyncConnection =
ConnectionFactory.createAsyncConnection(conf);

        // 使用连接
        System.out.println(connection);

        // 关闭连接
        connection.close();*/

        // 直接使用创建好的链接
        // 不要在main线程里面单独创建
        System.out.println(HBaseConnection.connection);

        // 在main线程的最后记得关闭连接
        HBaseConnection.closeConnection();
    }
}

```

2 插入数据

```
/**
     * 插入数据
     * @param namespace 命名空间
     * @param tableName 表名
     * @param rowKey 行号
     * @param columnFamily 列族
```

```
         * @param columnName 列明
         * @param value 值
         */
        public static void putCell(String namespace, String tableName, String rowKey, String columnFamily, String columnName, String value) throws IOException {
                // 1.获取table
                Table table = connection.getTable(TableName.valueOf(namespace, tableName));

                // 2.创建put对象
                Put put = new Put(Bytes.toBytes(rowKey));

                // 3.添加列
                put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(columnName), Bytes.toBytes(value));
                try {
                        table.put(put);
                } catch (IOException e) {
                        e.printStackTrace();
                }

                // 关闭table
                table.close();
        }
```

3 创建表格

```
/**
         * 创建表格
         * @param namespace 命名空间名称
         * @param tableName 表格名称
         * @param columnFamilies 列族名称(可以有多个)
         */
        public static void createTable(String namespace, String tableName, String... columnFamilies) throws IOException {
                // 判断是否有至少一个列族
                if (columnFamilies.length == 0){
                        System.out.println("创建表格至少有一个列族");
                        return;
                }

                // 判断表格是否存在
                if (isTableExists(namespace, tableName)){
                        System.out.println("表格已经存在");
                        return;
                }

                // 1.获取admin
                Admin admin = connection.getAdmin();

                // 2.创建表格
                // 2.1 创建表格描述 => 建造者模式
                TableDescriptorBuilder builder = TableDescriptorBuilder.newBuilder(TableName.valueOf(namespace, tableName));

                // 2.2 添加参数
                for (String columnFamily : columnFamilies) {

```

```
                // 2.3 创建列族描述的建造者
                ColumnFamilyDescriptorBuilder columnFamilyDescriptorBuilder =
ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes(columnFamily));

                // 2.4 对应当前的列族添加参数
                // 添加版本参数
                columnFamilyDescriptorBuilder.setMaxVersions(5);

                // 2.5 创建添加完参数的列族描述
                builder.setColumnFamily(columnFamilyDescriptorBuilder.build());
            }

            // 2.3 创建对应的表格描述
            try {
                admin.createTable(builder.build());
            } catch (IOException e) {
                e.printStackTrace();
            }

            // 3.关闭admin
            admin.close();
        }
```

4 创建命名空间

```
/**
     * 创建命名空间
     * @param namespace 命名空间名称
     */
    public static void createNamespace(String namespace) throws IOException {
        // 1.获取admin
        // 此处异常先不抛出，最后统一处理
        // admin连接：轻量级，不是线程安全的，不推荐池化或者缓存该连接
        Admin admin = connection.getAdmin();

        // 2.调用方法创建命名空间
        // 代码相对shell更底层，所以shell能够实现的功能，代码一定能实现
        // 所以需要填写完整的命名空间描述

        // 2.1 创建命名空间描述的建造者 => 设计师
        NamespaceDescriptor.Builder builder = NamespaceDescriptor.create(namespace);

        // 2.2 给命名空间添加需求
        builder.addConfiguration("user", "atguigu");

        // 2.3 使用builder构造出对应的添加完参数的对象
        // 创建命名空间出现的问题，都属于本方法自身问题
        try {
            admin.createNamespace(builder.build());
        } catch (IOException e) {
            System.out.println("命名空间已经存在");
            e.printStackTrace();
        }

        // 3.关闭admin
        admin.close();

```

```
    }
```

5 带过滤的扫描

```
/**
     * 带过滤的扫描
     * @param namespace 命名空间
     * @param tableName 表格名
     * @param startRow 开始
     * @param stopRow 结束
     * @param columnFamily 列族
     * @param columnName 列明
     * @param value 值
     * @throws IOException
     */
    public static void filterScan(String namespace, String tableName, String startRow,
                                  String stopRow, String columnFamily, String columnName, String value) throws IOException {
        // 1.获取table
        Table table = connection.getTable(TableName.valueOf(namespace, tableName));

        // 2.创建scan对象
        Scan scan = new Scan();
        // 如果直接调用，就扫描整张表
        // 添加参数来控制扫描数据
        scan.withStartRow(Bytes.toBytes(startRow));
        scan.withStopRow(Bytes.toBytes(stopRow));

        // 可以添加多个过滤
        FilterList filterList = new FilterList();

        // 创建过滤器
        // (1) 结果只保留当前列的数据
        ColumnValueFilter columnValueFilter = new ColumnValueFilter(
                // 列族名称
                Bytes.toBytes(columnFamily),
                // 列名
                Bytes.toBytes(columnName),
                // 比较关系
                CompareOperator.EQUAL,
                // 值
                Bytes.toBytes(value)
        );

        // (2) 结果保留整行数据
        // 结果会同时保留没有当前列的数据
        // 比如："bigdata:student"中"rowKey"为"1001"的"info:age"值是18,但是"info:name"为空值
        // 也会保留该行数据
        SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter(
                // 列族名称
                Bytes.toBytes(columnFamily),
                // 列名
                Bytes.toBytes(columnName),
                // 比较关系
                CompareOperator.EQUAL,
                // 值
                Bytes.toBytes(value)
```

```
            );

            // 本身可以添加多个过滤器
            filterList.addFilter(singleColumnValueFilter);

            // 添加过滤
            scan.setFilter(filterList);

            try {
                  // 读取多行数据
                  ResultScanner scanner = null;
                  scanner = table.getScanner(scan);
                  // result来记录一行数据
                  // ResultScanner来记录多行数据
                  for (Result result : scanner) {
                        Cell[] cells = result.rawCells();
                        for (Cell cell : cells) {
                              System.out.print(new String(CellUtil.cloneRow(cell)) + "-"
                                          + new String(CellUtil.cloneFamily(cell)) + "-"
                                          + new String(CellUtil.cloneQualifier(cell)) + "-"
                                          + new String(CellUtil.cloneValue(cell)) + "\t");
                        }
                        System.out.println();
                  }
            } catch (IOException e) {
                  e.printStackTrace();
            }

            table.close();
      }
```

6 读取数据

```
/**
       * 读取数据
       * @param namespace 命名空间
       * @param tableName 表名
       * @param rowKey 行号
       * @param columnFamily 列族
       * @param columnName 列明
       */
      public static void getCells(String namespace, String tableName, String rowKey, String columnFamily, String columnName) throws IOException {
            // 1.获取table对象
            Table table = connection.getTable(TableName.valueOf(namespace, tableName));

            // 2.创建get对象
            Get get = new Get(Bytes.toBytes(rowKey));

            // 如果直接调用get方法读取数据，此时读一整行数据
            // 如果想要读取某一列数据，需要添加对应的参数
            get.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(columnName));
            // 设置读取数据的版本
            get.readAllVersions();

            // 读取数据，得到result对象
            try {
```

```
                Result result = null;
                result = table.get(get);
                // 处理数据
                Cell[] cells = result.rawCells();
                for (Cell cell : cells) {
                    // cell存储数据比较底层
                    // 会乱码
                    String value = new String(CellUtil.cloneValue(cell));
                    System.out.println(value);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

            table.close();
    }
```

7 判断表格是否存在

```
/**
     * 判断表格是否存在
     * @param namespace 命名空间名称
     * @param tableName 表格名称
     * @return true表示存在
     */
    public static boolean isTableExists(String namespace, String tableName) throws IOException {
        // 1.获取admin
        Admin admin = connection.getAdmin();

        // 2.判断表格是否存在
        boolean b = false;
        try {
            b = admin.tableExists(TableName.valueOf(namespace, tableName));
        } catch (IOException e) {
            e.printStackTrace();
        }

        // 3.关闭连接
        admin.close();

        // 4.返回结果
        return b;
    }
```

8 扫描数据

```
/**
     * 扫描数据
     * @param namespace 命名空间
     * @param tableName 表名
     * @param startRow 开始行(包含)
     * @param stopRow 结束行(不包含)
     */
    public static void scanRows(String namespace, String tableName, String startRow, String stopRow) throws IOException {
        // 1.获取table
```

```
            Table table = connection.getTable(TableName.valueOf(namespace, tableName));

            // 2.创建scan对象
            Scan scan = new Scan();
            // 如果直接调用，就扫描整张表
            // 添加参数来控制扫描数据
            scan.withStartRow(Bytes.toBytes(startRow));
            scan.withStopRow(Bytes.toBytes(stopRow));

            try {
                  // 读取多行数据
                  ResultScanner scanner = null;
                  scanner = table.getScanner(scan);
                  // result来记录一行数据
                  // ResultScanner来记录多行数据
                  for (Result result : scanner) {
                        Cell[] cells = result.rawCells();
                        for (Cell cell : cells) {
                              System.out.print(new String(CellUtil.cloneRow(cell)) + "-"
                                          + new String(CellUtil.cloneFamily(cell)) + "-"
                                          + new String(CellUtil.cloneQualifier(cell)) + "-"
                                          + new String(CellUtil.cloneValue(cell)) + "\t");
                        }
                        System.out.println();
                  }
            } catch (IOException e) {
                  e.printStackTrace();
            }

            table.close();
      }
```

9 删除表格

```
/**
       * 删除表格
       * @param namespace 命名空间
       * @param tableName 表名
       * @return 是否删除
       */
      public static boolean deleteTable(String namespace, String tableName) throws IOException {
            // 判断表格是否存在
            if (isTableExists(namespace, tableName)){
                  System.out.println("表格不存在，无法删除");
                  return false;
            }

            // 1.获取admin
            Admin admin = connection.getAdmin();

            // 2.删除表格
            try {
                  // HBase删除表格之前一定要先标记表格为不可用
                  admin.disableTable(TableName.valueOf(namespace, tableName));
                  admin.deleteTable(TableName.valueOf(namespace, tableName));
            } catch (IOException e) {
                  e.printStackTrace();
```

```
            }

            // 3.关闭连接
            admin.close();

            return true;
      }
```

10 删除一行中的一列数据

```
/**
       * 删除一行中的一列数据
       * @param namespace 命名空间
       * @param tableName 表名
       * @param rowKey 行号
       * @param columnFamily 列族
       * @param columnName 列名
       */
      public static void deleteColumn(String namespace, String tableName, String rowKey, String columnFamily, String columnName) throws IOException {
            Table table = connection.getTable(TableName.valueOf(namespace, tableName));

            Delete delete = new Delete(Bytes.toBytes(rowKey));
            // 删除一个版本
            delete.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(columnName));
            // 删除所有版本
            delete.addColumns(Bytes.toBytes(columnFamily), Bytes.toBytes(columnName));

            try {
                  table.delete(delete);
            } catch (IOException e) {
                  e.printStackTrace();
            }

            table.close();
      }
```

11 修改表格

```
/**
       * 修改表格
       * @param namespace 命名空间
       * @param tableName 表名
       * @param columnFamily 列族
       * @param version 版本
       */
      public static void modifyTable(String namespace, String tableName, String columnFamily, int version) throws IOException {

            // 判断表格是否存在
            if (!isTableExists(namespace, tableName)){
                  System.out.println("表格不存在");
                  return;
            }

            // 1.获取admin
```

```
17              Admin admin = connection.getAdmin();
18
19              try {
20                    // 2.修改表
21                    // 2.0 获取之前的表格描述!!!!!!!!!(这个很重要!!!!!!!)
22                    TableDescriptor descriptor = admin.getDescriptor(TableName.valueOf(namespace,
    tableName));
23                    // 2.1 创建一个表格描述
24                    // 如果直接填写tableName，相当于创建了一个新的表格描述建造者，没有之前的信息。
25                    // 如果想要修改之前的信息，必须调用方法填写一个旧的表格描述
26                    TableDescriptorBuilder tableDescriptorBuilder =
    TableDescriptorBuilder.newBuilder(descriptor);
27
28                    // 2.2 对应建造者进行表格数据的修改
29                    // 需要填写旧的列族描述
30                    ColumnFamilyDescriptor columnFamily1 =
    descriptor.getColumnFamily(Bytes.toBytes(columnFamily));
31                    ColumnFamilyDescriptorBuilder columnFamilyDescriptorBuilder =
    ColumnFamilyDescriptorBuilder.newBuilder(columnFamily1);
32                    // 修改对应版本
33                    columnFamilyDescriptorBuilder.setMaxVersions(version);
34                    // 此处修改的时候，如果填写新创建的，那么别的参数会初始化
35
      tableDescriptorBuilder.modifyColumnFamily(columnFamilyDescriptorBuilder.build());
36
37                    admin.modifyTable(tableDescriptorBuilder.build());
38              } catch (IOException e) {
39                    e.printStackTrace();
40              }
41
42               // 关闭admin
43               admin.close();
44        }
```

第五章 Phoenix

简介

`Phoenix` 是 HBase 的开源 SQL 中间层，它允许你使用标准 JDBC 的方式来操作 HBase 上的数据。在 `Phoenix` 之前，如果你要访问 HBase，只能调用它的 Java API，但相比于使用一行 SQL 就能实现数据查询，HBase 的 API 还是过于复杂。`Phoenix` 的理念是 `we put sql SQL back in NOSQL`，即你可以使用标准的 SQL 就能完成对 HBase 上数据的操作。同时这也意味着你可以通过集成 `Spring Data JPA` 或 `Mybatis` 等常用的持久层框架来操作 HBase。

其次 `Phoenix` 的性能表现也非常优异，`Phoenix` 查询引擎会将 SQL 查询转换为一个或多个 HBase Scan，通过并行执行来生成标准的 JDBC 结果集。它通过直接使用 HBase API 以及协处理器和自定义过滤器，可以为小型数据查询提供毫秒级的性能，为千万行数据的查询提供秒级的性能。同时 Phoenix 还拥有二级索引等 HBase 不具备的特性，因为以上的优点，所以 `Phoenix` 成为了 HBase 最优秀的 SQL 中间层。

Phoenix 简单使用

创建表

```
CREATE TABLE IF NOT EXISTS us_population (
        state CHAR(2) NOT NULL,
        city VARCHAR NOT NULL,
        population BIGINT
         CONSTRAINT my_pk PRIMARY KEY (state, city));
```

新建的表会按照特定的规则转换为 HBase 上的表，关于表的信息，可以通过 Hbase Web UI 进行查看

插入数据

Phoenix 中插入数据采用的是 `UPSERT` 而不是 `INSERT`,因为 Phoenix 并没有更新操作，插入相同主键的数据就视为更新，所以 `UPSERT` 就相当于 `UPDATE` + `INSERT`

```
UPSERT INTO us_population VALUES('NY','New York',8143197);
UPSERT INTO us_population VALUES('CA','Los Angeles',3844829);
UPSERT INTO us_population VALUES('IL','Chicago',2842518);
UPSERT INTO us_population VALUES('TX','Houston',2016582);
UPSERT INTO us_population VALUES('PA','Philadelphia',1463281);
UPSERT INTO us_population VALUES('AZ','Phoenix',1461575);
UPSERT INTO us_population VALUES('TX','San Antonio',1256509);
UPSERT INTO us_population VALUES('CA','San Diego',1255540);
UPSERT INTO us_population VALUES('TX','Dallas',1213825);
UPSERT INTO us_population VALUES('CA','San Jose',912332);
```

改数据

```
-- 插入主键相同的数据就视为更新
UPSERT INTO us_population VALUES('NY','New York',999999);
```

删除数据

```
DELETE FROM us_population WHERE city='Dallas';
```

查询数据

```
SELECT state as "州",count(city) as "市",sum(population) as "热度"
FROM us_population
GROUP BY state
ORDER BY sum(population) DESC;
```

退出命令

```
!quit
```

扩展

从上面的操作中可以看出，Phoenix 支持大多数标准的 SQL 语法。关于 Phoenix 支持的语法、数据类型、函数、序列等详细信息，因为涉及内容很多，可以参考其官方文档，官方文档上有详细的说明：

- **语法 (Grammar)**： https://phoenix.apache.org/language/index.html
- **函数 (Functions)**： http://phoenix.apache.org/language/functions.html
- **数据类型 (Datatypes)**： http://phoenix.apache.org/language/datatypes.html
- **序列 (Sequences)**：http://phoenix.apache.org/sequences.html

- **联结查询 (Joins)**： http://phoenix.apache.org/joins.html