数据库大作业项目说明文档

[摘要 2](#_Toc42639423)

[运行说明 2](#_Toc42639424)

[文件管理模块 2](#_Toc42639425)

[主要功能 2](#_Toc42639426)

[索引模块 4](#_Toc42639427)

[1.概述 4](#_Toc42639428)

[2.设计过程 5](#_Toc42639429)

[3.索引模块类实现 5](#_Toc42639430)

[3.1创建B+索引树 5](#_Toc42639431)

[3.2插入数据 6](#_Toc42639432)

[3.3查询数据 11](#_Toc42639433)

[3.4删除数据 13](#_Toc42639434)

[3.5修改数据 14](#_Toc42639435)

[系统管理模块 15](#_Toc42639436)

[1.对数据库的管理 15](#_Toc42639437)

[2.对表的管理 19](#_Toc42639438)

[3.类介绍 24](#_Toc42639439)

[查询解析模块 24](#_Toc42639440)

[1.对数据表的操作 24](#_Toc42639441)

[2.类文件说明 32](#_Toc42639442)

[项目总结 32](#_Toc42639443)

## 摘要

本次实践项目，我们用java语言开发实现了一个简单的数据库管理系统，其中包括的功能模块有：数据库的文件管理、索引的管理、表的管理、系统管理、配置管理、异常管理等功能模块。由于java是基于jvm的一种开发语言，很多底层的操作对java开发人员都是透明的，所以用java实现c 语言的各种对底层的操作、系统调用等都是比较困难的。

首先说说索引，我们选择最经典也是最常用的B+树作为索引结构。其实最开始有考虑多种索引结构实现索引功能，包括哈希索引、有序数组索引等，但是查完之后对比发现，哈希索引和有序数组索引有更强的针对性，并不具备普适性，而B+树则能够普适性，这也是有余B+树本身的特性所决定的。

SqlHandler这个类是系统的最重要的类，不实现具体的逻辑，只负责任务调度，相当于系统的任务调度中心，这一思想是借鉴了经典的springMVC框架的思想，Sqlhandler的功能类似于Controller的作用。这个类作为用户层和功能实现搭起了一个桥梁，这样做的好处是相互之前隐藏细节，而且sqlhandler还能够对用户输入的sql语句做处理封装等，然后交给具体类去实现，比如说我们这里用户输入的sql语句还有面会有分号，由于分号只是在sql输入的时候做结标识的，并没有实际的业务逻辑，所以sqlhandler会去掉这个分号，所有的具体的实现类拿到的sql语句都没有分号。

异常处理，由于Sqlhandler是任务调用中心， 当然也可以实现对异常的统一处理。首先后面的具体的实现类的异常都全部抛出，sqlhandler会捕获所有异常，然后统一封装返回给用户。

由于java语言的限制和个人能力的原因，很多想法未能实现，该系统有借鉴经典的思想，但是很多细节方面还需要优化。

## 运行说明

该系统基于jdk1.8版本开发，使用了流行的maven作为统一的依赖管理。User.data 反序列化后保存的用户信息文件，需要放置在config/application.properties配置的对应的目录下MagicSql.zip 为项目源码，需要使用maven工具和jdk1.8的环境运行

## 文件管理模块

### 主要功能

新建数据库会在数据目录下创建和数据库名一样的文件夹，如果创建的数据库已存在则新建数据库失败，如果是删除数据库则同时会删除数据库目录下的所有的文件然后再删除该目录。另一方面，创建表或者对表修改后持久化操作的时候也会对当前数据库目录下的文件做修改或者删除等操作。

#### 表结构管理

我们借用java中的File类及InputStream/OutputStream/BufferInputStream等函数完成文件IO 操作，实现了表管理功能。创建表的时候会解析sql生成一个.frm为后缀的文件，里面是序列化的java数据，保存的是FrmEntity的列表，列表的大小登录属性的个数，每次对表操作的时候的会读取并解析该文件里面的内容并解析响应的sql语句，执行对应的操作。

//如果文件已存在，则替换原来的文件(目录文件不会删除)

public static void writeObject(String path,String name,Object value){

FileOutputStream fout = null;

ObjectOutputStream out = null;

File file = null;

try{

file = new File(path+File.separator+name);

if(file!=null&&file.exists()&&!file.isDirectory()){

// file.delete();

file.deleteOnExit();

}

fout = new FileOutputStream(path+File.separator+name);

out = new ObjectOutputStream(fout);

out.writeObject(value);

out.close();

fout.close();

}catch (Exception e){

throw new RuntimeException(Constant.Error.SYSTEM\_EXCEPTION);

}

}

//读取失败是否抛异常

public static Object readObject(String path,String name,boolean throwExeption){

Object obj =null;

try {

obj=readObject(path, name);

}catch (Exception e){

if(throwExeption){

throw e;

}

}

return obj;

}

//如果文件不存在，返回null

public static Object readObject(String path,String name){

FileInputStream fin = null;

ObjectInputStream in = null;

Object result = null;

File file = null;

try{

file = new File(path+File.separator+name);

if(file==null||!file.exists()||file.getTotalSpace()==0){

return null;

}

fin = new FileInputStream(path+File.separator+name);

in = new ObjectInputStream(fin);

result= in.readObject();

in.close();

fin.close();

}catch (Exception e){

throw new RuntimeException(Constant.Error.SYSTEM\_EXCEPTION);

}

return result;

}

#### 3.实现类

* 1. ApplicationConfig 系统的核心配置类，启动或者修改配置
  2. BTree 索引树，对应于数据库表级别的操作
  3. FileManage 文件管理，保存对文件的管理，持久化数据和解析数据操作
  4. FrmEntity 对应于.frm文件，和数据库表的属性一一对应
  5. Sqlhandler 系统核心类，控制中心
  6. User 对应于用户数据，用于登录操作

## 索引模块

### 1.概述

B+树是一个n叉树，每个节点通常有多个孩子，一颗B+树包含根节点、内部节点和叶子节点。B+树能够保持数据稳定且有序。基于B+树的各种优点，本数据库管理系统采用B+树作为实现索引的数据结构。

### 2.设计过程

BTree 用于数据的存储索引的实现，创建B+树索引结构额时候需要指明实体类和索引列，该B+树支持对记录的增删改查。通过对某些属性添加索引树，可以极大的提高对数据库的增删改查的效率，并且利用B+树的平衡原理和B+树非叶子节点只存储关键字，叶子节点存储关键字和数据等特性使B+树成为用于建索引的不二选择。定义该B+树有4个子节点，并且根节点大于子节点的高度，也就是树根的高度最大，其中Node为树的内部类，作为树的节点。Entity作为节点的内部数据实体类。新建索引的时候new一个空树，并将该实例序列化存入文件tableName.index中并放入缓存中，当读取的时候缓存中没有则读取文件中数据并且反序列化为BTree实体类，更新BTree时也同步对文件和缓存更新，当有新插入记录时，直接调用put方法加入索引树中。

### 3.索引模块类实现

1. BTree.java B+树索引实现类，包含Node，KeyNode，LeafNode内部类，内部类的好处是可以对外部隐藏内部的处理逻辑。
2. 其中Node为抽象类，KeyNode为非叶子节点，LeafNode为叶子节点。KeyNode和LeafNode继承了Node并实现了Node定义的增删改查的抽象方法，实现BTree的增删改查的功能。

### 3.1创建B+索引树

默认采用3阶索引树，也可自定义，maxAmount为最大的节点数，加1为了插入方便节点分裂，初始根节点为叶子节点，包含数据和关键字

//默认3阶

public BTree(){

this(3);

}

public BTree(Integer degree){

this.degree = degree;

this.maxAmount = degree+1;

this.root = new LeafNode();

this.left = null;

}

### 3.2插入数据

Node下为抽象方法，需要KeyNode和LeafNode有具体的方法实现

1)KeyNode下插入方法的实现，先找到插入的叶子节点，最后在叶子节点中插入，叶子节点插入建LeafNode中的插入方法，叶子节点插入节点如果分裂后，则继续回到非叶子节点继续分裂

@Override

Node<T, K> insert(T value, K key) {

int i = 0;

while(i < this.amount){

if(key.compareTo((K) this.keys[i]) < 0)

break;

i++;

}

if(key.compareTo((K) this.keys[this.amount - 1]) >= 0) {

i--;

}

return this.childs[i].insert(value, key);

}

叶子节点分裂后回到非叶子节点继续分裂

protected Node insertNode(Node node1, Node node2, K key){

K oldKey = null;

if(this.amount > 0)

oldKey = (K) this.keys[this.amount - 1];

if(key == null || this.amount <= 0){

this.keys[0] = node1.keys[node1.amount - 1];

this.keys[1] = node2.keys[node2.amount - 1];

this.childs[0] = node1;

this.childs[1] = node2;

this.amount += 2;

return this;

}

int i = 0;

while(key.compareTo((K)this.keys[i]) != 0){

i++;

}

//左边节点的最大值可以直接插入,右边的要挪一挪再进行插入

this.keys[i] = node1.keys[node1.amount - 1];

this.childs[i] = node1;

Object tempKeys[] = new Object[maxAmount];

Object tempChilds[] = new Node[maxAmount];

System.arraycopy(this.keys, 0, tempKeys, 0, i + 1);

System.arraycopy(this.childs, 0, tempChilds, 0, i + 1);

System.arraycopy(this.keys, i + 1, tempKeys, i + 2, this.amount - i - 1);

System.arraycopy(this.childs, i + 1, tempChilds, i + 2, this.amount - i - 1);

tempKeys[i + 1] = node2.keys[node2.amount - 1];

tempChilds[i + 1] = node2;

this.amount++;

if(this.amount <= degree){

System.arraycopy(tempKeys, 0, this.keys, 0, this.amount);

System.arraycopy(tempChilds, 0, this.childs, 0, this.amount);

return null;

}

Integer middle = this.amount / 2;

KeyNode tempNode = new KeyNode();

tempNode.amount = this.amount - middle;

tempNode.parent = this.parent;

if(this.parent == null) {

KeyNode tempKeyNode = new KeyNode();

tempNode.parent = tempKeyNode;

this.parent = tempKeyNode;

oldKey = null;

}

System.arraycopy(tempKeys, middle, tempNode.keys, 0, tempNode.amount);

System.arraycopy(tempChilds, middle, tempNode.childs, 0, tempNode.amount);

for(int j = 0; j < tempNode.amount; j++){

tempNode.childs[j].parent = tempNode;

}

this.amount = middle;

this.keys = new Object[maxAmount];

this.childs = new Node[maxAmount];

System.arraycopy(tempKeys, 0, this.keys, 0, middle);

System.arraycopy(tempChilds, 0, this.childs, 0, middle);

KeyNode parentNode = (KeyNode)this.parent;

return parentNode.insertNode(this, tempNode, oldKey);

}

}

2）LeafNode中插入方法的实现，从根节点开始遍历一直到定位到叶子节点的位置并插入，如果需要分裂则将叶节点分裂，并且递归回到非叶子节点进行分裂的判断和实现，保证满足B+树的特性。

protected Node insert(T value, K key) {

K oldKey = null;

if(this.amount > 0) {

oldKey = (K) this.keys[this.amount - 1];

}

int i = 0;

while(i < this.amount){

if(key.compareTo((K) this.keys[i]) < 0)

break;

i++;

}

Object tempKeys[] = new Object[maxAmount];

Object tempKalues[] = new Object[maxAmount];

System.arraycopy(this.keys, 0, tempKeys, 0, i);

System.arraycopy(this.values, 0, tempKalues, 0, i);

System.arraycopy(this.keys, i, tempKeys, i + 1, this.amount - i);

System.arraycopy(this.values, i, tempKalues, i + 1, this.amount - i);

tempKeys[i] = key;

tempKalues[i] = value;

this.amount++;

if(this.amount <= degree){

System.arraycopy(tempKeys, 0, this.keys, 0, this.amount);

System.arraycopy(tempKalues, 0, this.values, 0, this.amount);

Node node = this;

while (node.parent != null){

K tempkey = (K)node.keys[node.amount - 1]; if(tempkey.compareTo((K)node.parent.keys[node.parent.amount - 1]) > 0){

node.parent.keys[node.parent.amount - 1] = tempkey;

node = node.parent;

}

else {

break;

}

}

return null;

}

Integer middle = this.amount / 2;

LeafNode<T, K> tempNode = new LeafNode<T, K>();

tempNode.amount = this.amount - middle;

tempNode.parent = this.parent;

if(this.parent == null) {

KeyNode tempKeyNode = new KeyNode();

tempNode.parent = (Node)tempKeyNode;

this.parent = (Node)tempKeyNode;

oldKey = null;

}

System.arraycopy(tempKeys, middle, tempNode.keys, 0, tempNode.amount);

System.arraycopy(tempKalues, middle, tempNode.values, 0, tempNode.amount);

this.amount = middle;

this.keys = new Object[maxAmount];

this.values = new Object[maxAmount];

System.arraycopy(tempKeys, 0, this.keys, 0, middle);

System.arraycopy(tempKalues, 0, this.values, 0, middle);

this.right = tempNode;

tempNode.left = this;

KeyNode parentNode = (KeyNode)this.parent;

return parentNode.insertNode(this, tempNode, oldKey);

}

### 3.3查询数据

下面是二分查找的关键代码部分，节点内部采用二分查找，不管是增删改查都要先查找，这部分都是采用的二分查找。实现的逻辑：先在非叶子节点循环查找一直找到叶子节点，并在叶子节点获取具体的数据，数据存在则返回，不存在则返回null。

Integer left = 0;

Integer right = this.amount;

Integer middle = (left + right) / 2;

while(left < right){

K middleKey = (K) this.keys[middle];

if(key.compareTo(middleKey) == 0) {

this.values[middle]=null;

// return (T) this.values[middle];

return 1;

}else if(key.compareTo(middleKey) < 0) {

right = middle;

}else {

left = middle;

}

middle = (left + right) / 2;

}

a. 非叶子节点查找方法的实现

T find(K key) {

int i = 0;

while(i < this.amount){

if(key.compareTo((K) this.keys[i]) <= 0) {

break;

}

i++;

}

if(this.amount == i) {

return null;

}

return this.childs[i].find(key);

}

b.叶子节点查询方法的实现

@Override

protected T find(K key) {

if(this.amount <=0) {

return null;

}

Integer left = 0;

Integer right = this.amount;

Integer middle = (left + right) / 2;

while(left < right){

K middleKey = (K) this.keys[middle];

if(key.compareTo(middleKey) == 0) {

return (T) this.values[middle];

}else if(key.compareTo(middleKey) < 0) {

right = middle;

}else {

left = middle;

}

middle = (left + right) / 2;

}

return null;

}

### 3.4删除数据

二分法一次通过非叶子节点、叶子节点找到响应的数据的位置，然后删除。为了提高效率，这里采用了空间换时间的方法，删除操作只是将叶子节点的数据置空，并且保留叶子节点，这样即使有频繁的删除操作依然对性能没有影响，不用重复的删除节点和合并节点。如果删除成功会返回1，删除不存在的数据会返回0

1. KeyNode非叶子节点的删除实现

@Override

int delete(K key) {

int i = 0;

while(i < this.amount){

if(key.compareTo((K) this.keys[i]) <= 0) {

break;

}

i++;

}

if(this.amount == i) {

return 0;

}

return this.childs[i].delete(key);

}

1. LeafNode叶子节点的删除实现

@Override

protected int delete(K key) {

if(this.amount <=0) {

return 0;

}

Integer left = 0;

Integer right = this.amount;

Integer middle = (left + right) / 2;

while(left < right){

K middleKey = (K) this.keys[middle];

if(key.compareTo(middleKey) == 0) {

this.values[middle]=null;

return 1;

}else if(key.compareTo(middleKey) < 0) {

right = middle;

}else {

left = middle;

}

middle = (left + right) / 2;

}

return 0;

}

### 3.5修改数据

二分法查找，定位到叶子节点上的具体的数据，然后更新具体叶子节点的数据。

1. KeyNode 非叶子节点更新操作的实现

@Override

void update(T value, K key) {

int i = 0;

while(i < this.amount){

if(key.compareTo((K) this.keys[i]) <= 0) {

break;

}

i++;

}

if(this.amount == i){

return;

}

this.childs[i].update(value,key);

}

1. LeafNode叶子节点更新操作的实现

@Override

void update(T t, K key) {

if(this.amount <=0) {

return;

}

Integer left = 0;

Integer right = this.amount;

Integer middle = (left + right) / 2;

while(left < right){

K middleKey = (K) this.keys[middle];

if(key.compareTo(middleKey) == 0) {

this.values[middle]=t;

}else if(key.compareTo(middleKey) < 0) {

right = middle;

}else {

left = middle;

}

middle = (left + right) / 2;

}

}

## 系统管理模块

### 1.对数据库的管理

#### 启动数据库，加载配置

启动数据库的时候会加载配置文件，其中配置文件的目录和配置文件名称固定不变，配置文件位置和文件名固定不变config/application.properties,配置文件必须包含数据目录和登录用户相关信息的目录，即data.path和login.user.file，这2个属性的值，否则启动失败，启动成功在控制台会打印配置文件的数据。登录的时候只支持一次登录，用户名密码输入错误需要重启输入。

##### a.启动初始化

public static void startApp(){

try {

//加载配置

loadConfig();

//用户设置的数据目录

putCache(CACHE\_KEY\_DATAPATH, config.get(CONFIG\_DATA\_PATH));

//加载用户数据

loadLoginUser();

}catch (Exception e){

e.printStackTrace();

System.out.println("数据库启动异常");

exit(0);

}

##### b.加载配置文件实现

private static void loadConfig() throws Exception {

Properties properties = new Properties();

properties.load(new FileInputStream(new File("config/application.properties")));

Set<String> keys = properties.stringPropertyNames();

for(String key : keys){

String value = properties.getProperty(key);

config.put(key,value);

}

System.out.println(config);

if(!keys.contains(CONFIG\_DATA\_PATH)&&!keys.contains(CONFIG\_USER\_PATH)){

throw new RuntimeException(Constant.Error.SQL\_CHECK\_CONFIG);

}

}

##### c.加载用户信息

private static void loadLoginUser() throws Exception{

FileInputStream fin = new FileInputStream(config.get(Constant.ConfigKey.CONFIG\_USER\_PATH));

ObjectInputStream in = new ObjectInputStream(fin);

loginUsers.addAll((List)in.readObject());

in.close();

fin.close();

}

##### d. 测试

测试包括了异常测试和正常测试。



图1 缺少配置文件启动异常

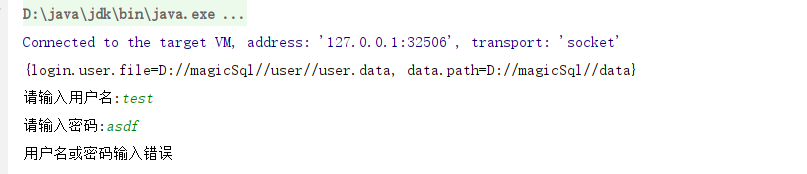


图2 用户名或密码输入错误

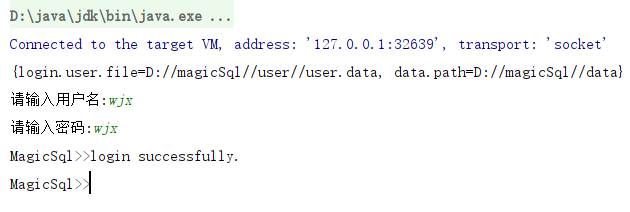


图3 正常启动并正确校验用户密码

#### 数据库的操作

可以实现对数据库的增加、删除和查询操作。实现的原理：增加的时候数据库配置的数据目录下创建一个和数据库名称相同的目录，删除数据库即删除数据目录下对应的数据库同名的目录并且会删除该目录的所有的文件(表相关的文件以及其他的文件)，查询即查询出数据库目录下所有的目录。创建数据库,会对新增的数据库校验，如果数据库已经存在，抛数据库已存在的异常，如果语法错误也会抛异常。删除数据库，如果数据库不存在，会报异常。

##### a.创建数据库

public static void createDir(String path,String dirName) {

File file = new File(path+File.separator+dirName);

if(file.exists()&&file.isDirectory()){

throw new RuntimeException(Constant.Error.SQL\_DB\_ALREADY\_EXIST);

}

file.mkdir();

}

##### b.查询数据库

private void showDatabases(String dataPath) ;

##### c.删除数据库

@Override

public String execute() {

String flag = check(sql.toLowerCase().replace("drop", "").replace(";","").trim());

String name ="0".equals(flag)?

sql.toLowerCase().replace("drop", "").replace("database", "").trim()

:sql.toLowerCase().replace("drop", "").replace("table", "").trim()+".data";

String path="0".equals(flag)?getDataPath():getDataPath()+File.separator+getCurrentDatabase();

//删除

dropFile(path,name);

System.out.println(sql+" successfully.");

return null;

}

/\*\*

\* 删除文件or文件夹

\* @param path

\* @param fileName

\*/

public static void dropFile(String path,String fileName){

File file = new File(path+File.separator+fileName);

if(file!=null&&file.exists()&&!file.isDirectory()){

file.delete();

}else if(file.isDirectory()){

File[] files = file.listFiles();

for(File f : files){

dropFile(path+File.separator+file.getName(),f.getName());

}

file.delete();

} else{

throw new RuntimeException(SQL\_COMMON\_ERROR);

}

}

##### d.测试

测试包含正确测试和异常测试

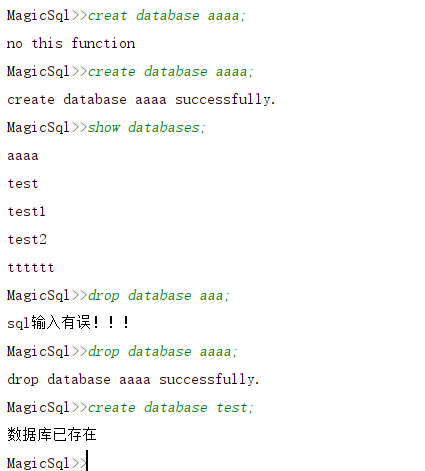
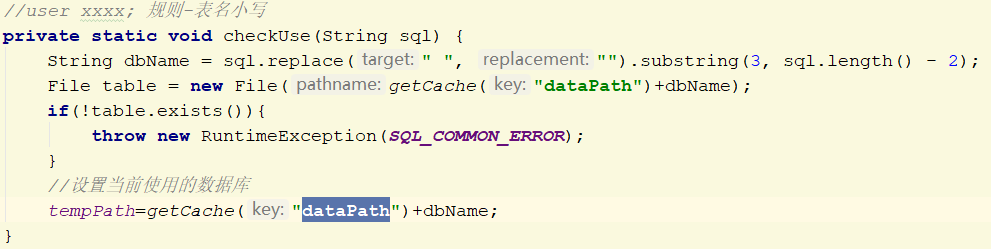
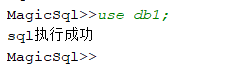


图4 对数据库的增删查测试

1. 执行use dbName;时，在检查use语法的时候修改当前内存中的目录位置





### 2.对表的管理

##### a、创建表

创建表的时候会在先检查当前是否选中了数据库，否则报错。创建完成后会生成一个表结构相关的持久化文件。FrmEntity代表一个属性，每个表对应一个属性的集合，将属性的集合保存在数据目录下的以表名命名以.frm结尾的文件下。

//创建表

private void createTable(String sql) {

String tableName = getName(sql.substring(0, sql.indexOf("(")), "create", "table");

//获取表结构数据,每个属性对应一个实体

FrmEntity[] list=getFieldEntity(sql);

System.out.println("entity is : "+ JSON.toJSONString(list));

//保存表结构

writeObject(getDataPath()+File.separator+getCurrentDatabase(),tableName+TABLE\_FRM\_SUFFIC,list);

}

private FrmEntity[] getFieldEntity(String sql) {

String[] fields = sql.substring(sql.indexOf("(")+1, sql.lastIndexOf(")")).split(",");

FrmEntity entity = null;

//主键列,可以为空

String keyName=null;

if(fields[fields.length-1].toLowerCase().contains("primary key")){

keyName = fields[fields.length-1].

substring(fields[fields.length-1].indexOf("(") + 1,

fields[fields.length-1].lastIndexOf(")"));

}

FrmEntity[] frmList = new FrmEntity[keyName==null?fields.length:fields.length-1];

//解析每一行，即每一个属性

for (int i = 0; i < fields.length; i++) {

//主键列

if(fields[i].toLowerCase().contains("primary")

&&fields[i].toLowerCase().contains("key")){

continue;

}

entity = new FrmEntity();

entity.setOrder(i);

//解析每一行

String[] values = fields[i].trim().split(" ");

for (int j = 0; j < values.length; j++) {

if(values[0].equalsIgnoreCase("PRIMARY")

&&values[1].equalsIgnoreCase("key")){

continue;

}

if(j==0){

if(values[j].equalsIgnoreCase(keyName)){

entity.setKey(true);

entity.setNull(false);

}

entity.setName(values[j]);

}else if(j==1){

String type = values[j].substring(0, values[j].indexOf("("));

String length = values[j].substring(values[j].indexOf("(")+1, values[j].lastIndexOf(")"));

entity.setType(type);

entity.setLength(length);

}else if(j==2){

if(values[j].equalsIgnoreCase("not")&&values[j+1].equalsIgnoreCase("null")){

entity.setNull(false);

}

}

}

frmList[i]=entity;

}

return frmList;

}

##### b、删除表

删除同创建操作正好相反，会将生成的文件删除

@Override

public String execute() {

String flag = check(sql.toLowerCase().replace("drop", "").replace(";","").trim());

String name ="0".equals(flag)?

sql.toLowerCase().replace("drop", "").replace("database", "").trim()

:sql.toLowerCase().replace("drop", "").replace("table", "").trim()+".data";

String path="0".equals(flag)?getDataPath():getDataPath()+File.separator+getCurrentDatabase();

//删除

dropFile(path,name);

System.out.println(sql+" successfully.");

return null;

}

调用删除文件或文件夹的公共的方法

/\*\*

\* 删除文件or文件夹

\* @param path

\* @param fileName

\*/

public static void dropFile(String path,String fileName){

File file = new File(path+File.separator+fileName);

if(file!=null&&file.exists()&&!file.isDirectory()){

file.delete();

}else if(file.isDirectory()){

File[] files = file.listFiles();

for(File f : files){

dropFile(path+File.separator+file.getName(),f.getName());

}

file.delete();

} else{

throw new RuntimeException(SQL\_COMMON\_ERROR);

}

}

##### c、查询表

show tables; 会展示当前数据库下的所有的表，实现原来是，扫描当前数据库目录下的所有的以.frm结束的数据，并读取名称去掉后缀打印出来。

private void showTables(String dataPath) {

if(isBlank(getCurrentDatabase())){

throw new RuntimeException(Constant.Error.SQL\_NO\_DB\_SELECTED);

}

File file = new File(dataPath+File.separatorChar+getCurrentDatabase());

File[] tables = file.listFiles(new FilenameFilter() {

@Override

public boolean accept(File dir, String name) {

return name.endsWith(Constant.Common.TABLE\_FILE\_SUFFIC);

}

});

for(File table : tables){

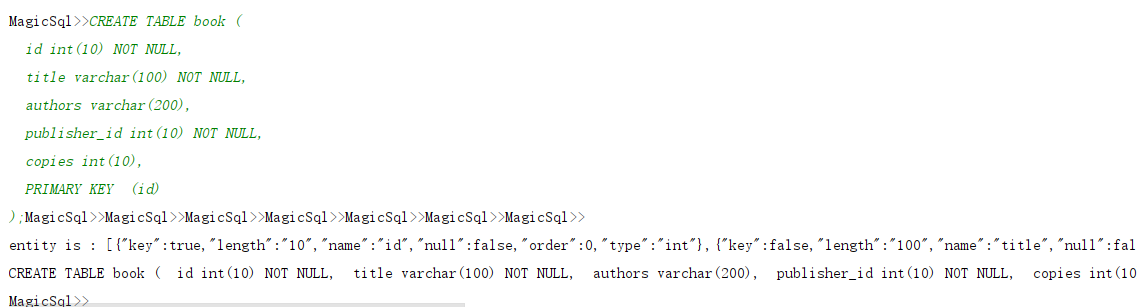
String name = table.getName();

System.out.println(name.substring(0,name.length()-5));

}

}

##### d.测试



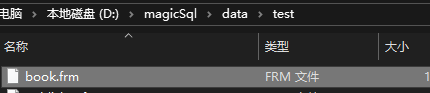


图6 创建表并在目录下生成对应的表结构文件

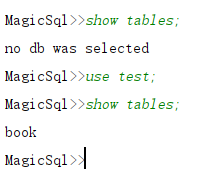
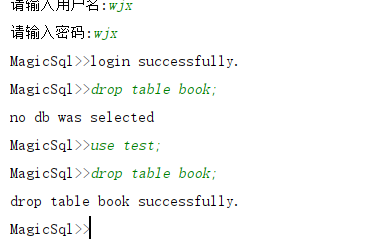


图7 查询所有表



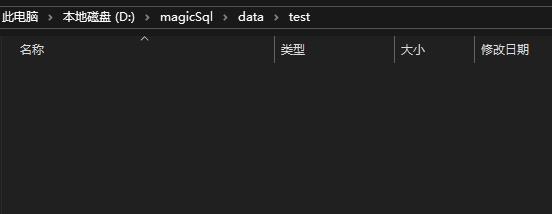


图8 删除表

### 3.类介绍

1、Check.java 用于登录校验以及对表操作的时候是否选中数据库

2、Create.java 创建表和数据库

3、Drop.java 删除表和数据库

4、FrmEntity.java 属性结构体

5、Show.java 查询表和数据库

6、Sqlhandler.java 公共类，同一个handler，所有操作的核心文件

7、Use.java 对应于使用的use命令

## 查询解析模块

该模块主要负责解析处理 insert、 delete、 update、 select 这些 SQL 命令，当输入的语句为上述语句时，对输入的字符串进行解析，对解析的结果调用记录管理模块、索引管理模块以及系统管理模块进行插入、 删除、 修改或查询，并负责将处理的结果显示到屏幕上。

### 1.对数据表的操作

该系统实现了简单的对表的增删改查，所有的对表操作的sql语句都会先选择数据库，否则抛异常无法继续操作。选择对应数据库后，由于数据库表会经常被使用，所以会把当前数据库保存在缓存中，并且在操作表表的时候会判断是否有选中数据库。每次对表的删除、修改、增加也都会持久化，由于时间关系，目前这些修改操作都是读取全部的记录，在内存中修改完成后，写回到磁盘中。

##### insert

执行顺序：判断是否符合条件->从sql提取关键信息->获取所有的记录的->读取b+树(b+树的叶子节点有所有的记录,B+树不存在则新建B+树，阶数为40)—>把所有的记录插入到索引树中—>保存索引树(会更新索引树)

@Override

public String execute() {

//判断是否选定数据库

isDataBaseUsed();

//获取表名

String trim = sql.toLowerCase().replace("insert", "").replace("into", "").trim();

String tableName = trim.substring(0, trim.toLowerCase().lastIndexOf("values")-1);

//是否存在对应的表结构文件

if(!frmMap.containsKey(tableName)){

throw new RuntimeException(SQL\_NO\_THIS\_TABLE);

}

//获取需要插入的所有记录

String[] allRecord = sql.substring(sql.indexOf("(")+1, sql.lastIndexOf(")")).replace("'","").split("\\)\\s\*,\\s\*\\(");

//插入所有的记录

insertRecode(allRecord,tableName);

return "insert into "+ tableName+" successfully.";

}

private void insertRecode(String[] allRecord,String tableName) {

//加载索引文件，是否存在索引

BTree tree=getBTree(tableName);

//获取表的结构

FrmEntity[] frmEntities = frmMap.get(tableName);

for (int i = 0; i < allRecord.length; i++) {

String record = allRecord[i].replace("'","");

//所有属性列的值

String[] allTypeValue = record.trim().substring(0, record.trim().length() - 1).split(",");

//插入的属性的格式必须和表结构一一对应

if(frmEntities.length!=allTypeValue.length){

throw new RuntimeException(SQL\_COMMON\_ERROR);

}

//主键值

Integer keyValue=null;

//主键类型数据字典，0-Long，1-Integer

int keyType=0;

for (int j = 0; j < allTypeValue.length; j++) {

if(allTypeValue[j].trim().length()>Integer.valueOf(frmEntities[j].getLength())){

throw new RuntimeException(SQL\_FIELD\_TOO\_LONG);

}

if(frmEntities[j].isKey()){

keyValue=Integer.valueOf(allTypeValue[j]);

String type = frmEntities[j].getType().trim().toLowerCase();

if(type.equals("int")||type.equalsIgnoreCase("integer")){

keyType=1;

}else if(type.equals("long")){

keyType=0;

}

}

}

//插入B+树,默认关键字是Integer类型

if(keyValue!=null) {

if(keyType==1){

tree.insert(allTypeValue, keyValue);

}else if(keyType==0){

Long value = Long.valueOf(keyValue);

tree.insert(allTypeValue, value);

}

}

}

//持久化

writeObject(getDataPath()+File.separator+getCurrentDatabase(),tableName+Constant.Common.TABLE\_INDEX\_SUFFIC,tree);

}

//读取B+索引树

private BTree getBTree(String tableName) {

BTree tree = null;

Object object= readObject(getDataPath()+ File.separator+getCurrentDatabase(),tableName+ Constant.Common.TABLE\_INDEX\_SUFFIC,false);

if(object==null){

tree=new BTree(40);

}else{

tree=(BTree)object;

}

return tree;

}

##### delete

执行顺序：判断是否符合条件->从sql提取关键信息 ->读取b+树(b+树的叶子节点有所有的记录)—>删除B+树中对应的节点—>保存索引树

@Override

public String execute() {

//判断是否选定数据库

isDataBaseUsed();

//获取tableName

String trim = sql.toLowerCase().replace("delete", "").replace("from", "").trim();

String tableName = trim.substring(0, trim.indexOf("where")).trim();

//属性

String property = trim.substring(trim.indexOf("where") + 5, trim.lastIndexOf("=")).trim();

//属性值

Integer key = Integer.valueOf(trim.substring(trim.indexOf("=")+1,trim.length()-1).replace("'","").trim());

//获取表结构

FrmEntity[] frmEntities = frmMap.get(tableName);

int keyType = 1;

for(FrmEntity entity:frmEntities){

String type = entity.getType().trim().toLowerCase();

if(entity.isKey()) {

if(!property.toLowerCase().equals(entity.getName())){

throw new RuntimeException(Constant.Error.SQL\_NO\_THIS\_CONDITION);

}

if (type.equals("int") || type.equalsIgnoreCase("integer")) {

keyType = 1;

break;

} else if (type.equals("long")) {

keyType = 0;

break;

}

}

}

//获取索引树并删除

BTree tree=(BTree) readObject(getDataPath()+ File.separator+getCurrentDatabase(),tableName+ Constant.Common.TABLE\_INDEX\_SUFFIC,false);

if(key!=null){

if(keyType==1){

tree.delete(key);

}else if(keyType==0){

tree.delete(Long.valueOf(key));

}

}

writeObject(getDataPath()+File.separator+getCurrentDatabase(),tableName+Constant.Common.TABLE\_INDEX\_SUFFIC,tree);

return "sql successfully.";

}

##### update

执行顺序：判断是否符合条件->从sql提取关键信息 ->读取b+树(b+树的叶子节点有所有的记录)—>更新B+树中对应的节点—>保存索引树

@Override

public String execute() {

//判断是否选定数据库

isDataBaseUsed();

String sqlLow = sql.toLowerCase();

//表名

String tableName = sqlLow.substring(sqlLow.indexOf("update") + 6, sqlLow.indexOf("set")).trim();

//更新的条件属性

String idName=sqlLow.substring(sqlLow.indexOf("where") + 5, sqlLow.lastIndexOf("=")).trim();

//更新条件属性的值

String idValue=sqlLow.substring(sqlLow.lastIndexOf("=")+1).replace("'","").trim();

//获取表结构

FrmEntity[] frmEntities = frmMap.get(tableName);

int keyType = 1;

for(FrmEntity entity:frmEntities){

String type = entity.getType().trim().toLowerCase();

if(entity.isKey()) {

if(!idName.toLowerCase().equals(entity.getName())){

throw new RuntimeException(Constant.Error.SQL\_NO\_THIS\_CONDITION);

}

if (type.equals("int") || type.equalsIgnoreCase("integer")) {

keyType = 1;

break;

} else if (type.equals("long")) {

keyType = 0;

break;

}

}

}

String[] resp =null;

//查询数据

BTree tree=(BTree) readObject(getDataPath()+ File.separator+getCurrentDatabase(),tableName+ Constant.Common.TABLE\_INDEX\_SUFFIC,false);

if(idValue!=null){

if(keyType==1){

resp=(String[])tree.find(Integer.valueOf(idValue));

}else if(keyType==0){

resp=(String[])tree.find(Long.valueOf(idValue));

}

}

System.out.println("resp :"+ Arrays.toString(resp));

//待更新的列

String[] split = sqlLow.substring(sqlLow.indexOf("set") + 3, sqlLow.indexOf("where") - 1).trim().split(",");

for(String s : split) {

for(FrmEntity frm : frmEntities){

//更新该列

if(frm.getName().equalsIgnoreCase(s.substring(0, s.indexOf("=")).trim())){

resp[frm.getOrder()]=s.substring(s.indexOf("=") + 1).trim().replace("'", "");

}

}

}

if(idValue!=null){

if(keyType==1){

tree.update(split,Integer.valueOf(idValue));

}else if(keyType==0){

tree.update(split,Long.valueOf(idValue));

}

}

writeObject(getDataPath()+File.separator+getCurrentDatabase(),tableName+Constant.Common.TABLE\_INDEX\_SUFFIC,tree);

return "sql successfully.";

}

##### d. select

执行顺序：判断是否符合条件->从sql提取关键信息 ->读取b+树(b+树的叶子节点有所有的记录)—>查询对应的记录，目前只支持对索引列的条件查询

@Override

public String execute() {

//判断是否选定数据库

isDataBaseUsed();

//获取tableName

String trim = sql.toLowerCase().substring(sql.toLowerCase().indexOf("from")+4);

String tableName = trim.substring(0, trim.indexOf("where")).trim();

//属性

String property = trim.substring(trim.indexOf("where") + 5, trim.lastIndexOf("=")).trim();

//属性值

Integer key = Integer.valueOf(trim.substring(trim.indexOf("=")+1,trim.length()-1).replace("'","").trim());

//获取表结构

FrmEntity[] frmEntities = frmMap.get(tableName);

int keyType = 1;

for(FrmEntity entity:frmEntities){

String type = entity.getType().trim().toLowerCase();

if(entity.isKey()) {

if(!property.toLowerCase().equals(entity.getName())){

throw new RuntimeException(Constant.Error.SQL\_NO\_THIS\_CONDITION);

}

if (type.equals("int") || type.equalsIgnoreCase("integer")) {

keyType = 1;

break;

} else if (type.equals("long")) {

keyType = 0;

break;

}

}

}

String[] resp =null;

//查询数据

BTree tree=(BTree) readObject(getDataPath()+ File.separator+getCurrentDatabase(),tableName+ Constant.Common.TABLE\_INDEX\_SUFFIC,false);

if(key!=null){

if(keyType==1){

resp=(String[])tree.find(key);

}else if(keyType==0){

resp=(String[])tree.find(Long.valueOf(key));

}

}

if(resp!=null&&resp.length>0) {

return Arrays.toString(resp);

}else{

return null;

}

}

##### e.测试

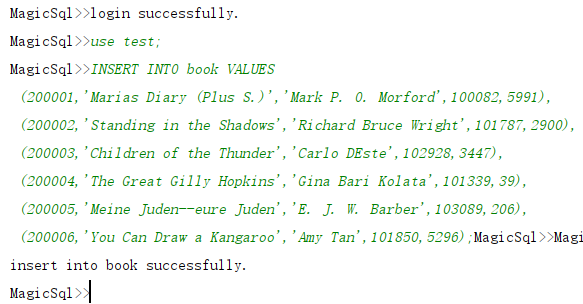


图1 插入6条测试数据

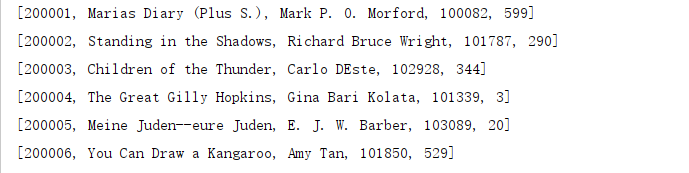


图2 查询插入的6条数据

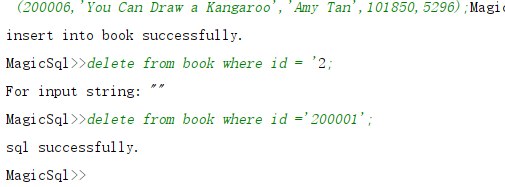


图3 删除id为200001的数据

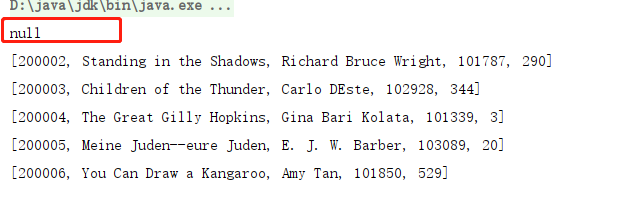


图4 删除后查询返回null

### 2.类文件说明

* 1. Select 对表的查询操作实现
  2. Delete 对表的删除操作的实现
  3. Update 对表的更新操作的实现
  4. Insert 对表的插入的实现
  5. SqlHandler 系统的公共方法，所有sql执行的控制中心