第七章 JDBC

课前回顾

1. 数据库设计的步骤是什么

```
收集信息
标识实体
找出实体的属性
找出实体之间的关系
ER => 数据库模型图
使用三大范式规范化数据库设计
```

2. 数据库三大范式是什么

```
第一范式：  保证数据库的每一列都具有原子性，不可再分
第二范式：  在第一范式的基础上，保证每一列都直接依赖于主键
第三范式：  在第二范式的基础上，保证每一列均不存在传递依赖
```

章节内容

- JDBC操作步骤 **重点**
- 反射 **重点** **难点**
- JDBC操作工具类 **重点** **难点**

章节目标

- 掌握JDBC操作步骤
- 掌握反射
- 掌握JDBC操作工具类的编写

第一节 JDBC

1. 概念

JDBC(Java Database Connection)是Java数据库连接技术的简称，提供连接数据库的能力。

2. JDBC API

Java 作为目前世界上最流行的高级开发语言，当然不可能考虑去实现各种数据库的连接与操作。但 Java 语言的开发者对数据库的连接与操作提供了相关的接口，供各大数据库厂商去实现。这些接口位于 java.sql 包中。

2.1 `Driver`

`java.sql.Driver`：数据库厂商提供的`JDBC`驱动包中必须包含该接口的实现，该接口中就包含连接数据库的功能。

```
//根据给定的数据库url地址连接数据库
Connection connect(String url, java.util.Properties info) throws SQLException;
```

2.2 `DriverManager`

`java.sql.DriverManager`：数据库厂商的提供的`JDBC`驱动交给`DriverManager`来管理，`DriverManager`主要负责获取数据库连接对象`Connection`

```
//通过给定的账号、密码和数据库地址获取一个连接
public static Connection getConnection(String url, String user,
                                       String password) throws SQLException
```

2.3 `Connection`

`java.sql.Connection`：连接接口，数据库厂商提供的`JDBC`驱动包中必须包含该接口的实现，该接口主要提供与数据库的交互功能

```
//创建一个SQL语句执行对象
Statement createStatement() throws SQLException;
//创建一个预处理SQL语句执行对象
PreparedStatement prepareStatement(String sql) throws SQLException;
//创建一个存储过程SQL语句执行对象
CallableStatement prepareCall(String sql) throws SQLException;
//设置该连接上的所有操作是否执行自动提交
void setAutoCommit(boolean autoCommit) throws SQLException;
//提交该连接上至上次提交以来所作出的所有更改
void commit() throws SQLException;
//回滚事务，数据库回滚到原来的状态
void rollback() throws SQLException;
//关闭连接
void close() throws SQLException;
//设置事务隔离级别
void setTransactionIsolation(int level) throws SQLException;
```

```
//不支持事务
int TRANSACTION_NONE             = 0;
//读取未提交的数据
int TRANSACTION_READ_UNCOMMITTED = 1;
//读取已提交的数据
int TRANSACTION_READ_COMMITTED   = 2;
//可重复读
int TRANSACTION_REPEATABLE_READ  = 4;
//串行化
int TRANSACTION_SERIALIZABLE     = 8;
```

2.4 `Statement`

`java.sql.Statement`：SQL语句执行接口，数据库厂商提供的`JDBC`驱动包中必须包含该接口的实现，该接口主要提供执行`SQL`语句的功能

```
//执行查询，得到一个结果集
ResultSet executeQuery(String sql) throws SQLException;
//执行更新，得到受影响的行数
int executeUpdate(String sql) throws SQLException;
//关闭SQL语句执行器
void close() throws SQLException;
//将SQL语句添加到批处理执行SQL列表中
void addBatch( String sql ) throws SQLException;
//执行批处理，返回列表中每一条SQL语句的执行结果
int[] executeBatch() throws SQLException;
```

2.5 `ResultSet`

`java.sql.ResultSet`：查询结果集接口，数据库厂商提供的`JDBC`驱动包中必须包含该接口的实现，该接口主要提供查询结果的获取功能

```
//光标从当前位置（默认位置位为0）向前移动一行，如果存在数据，则返回true，否则返回false
boolean next() throws SQLException;
//关闭结果集
void close() throws SQLException;
//获取指定列的字符串值
String getString(int columnIndex) throws SQLException;
//获取指定列的布尔值
boolean getBoolean(int columnIndex) throws SQLException;
//获取指定列的整数值
int getInt(Sting columnName) throws SQLException;
//获取指定列的对象
Object getObject(int columnIndex, Class type) throws SQLException;
//获取结果集元数据：查询结果的列名称、列数量、列别名等等
ResultSetMetaData getMetaData() throws SQLException;
//光标从当前位置（默认位置位为0）向后移动一行，如果存在数据，则返回true，否则返回false
boolean previous() throws SQLException;
```

3. `JDCB`操作步骤

3.1 引入驱动包

新建工程后，将`mysql-connector-java.jar`引入工程中

3.2 加载驱动

```
//MySQL 5.0
//Class.forName("com.mysql.jdbc.Driver");
//MySQL 8.0
Class.forName("com.mysql.cj.jdbc.Driver");
```

3.3 获取连接

```
Connection connection = DriverManager.getConnection(url, username, password);
```

3.4 创建SQL语句执行器

```
Statement statement = connection.createStatement();
```

3.5 执行SQL语句

```
//查询
ResultSet rs = statement.executeQuery(sql);
while(rs.next()){
    //获取列信息
}


//更新
int affectedRows = statement.executeUpdate();
```

3.6 释放资源

```
rs.close();
statement.close();
connection.close();
```

示例

```
package com.cyx.jdbc;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class JdbcTest {

    public static void main(String[] args) {
        //jdbc:使用jdbc连接技术
        //mysql://localhost:3306 使用的是MySQL数据库协议，访问的是本地计算机3306端口
        String url = "jdbc:mysql://localhost:3306/lesson?
serverTimezone=Asia/Shanghai";
        String username = "root";
        String password = "root";
        List<Account> accounts = new ArrayList<>();
        //MySQL8.0
        try {
            //加载驱动
            Class.forName("com.mysql.cj.jdbc.Driver");
            //获取连接
            Connection conn = DriverManager.getConnection(url, username,
password);
            //在连接上创建SQL语句执行器
            Statement s = conn.createStatement();
//            String sql = "SELECT account,balance,state FROM account";
```

```
//            //使用执行器执行查询，得到一个结果集
//            ResultSet rs = s.executeQuery(sql);
//            while (rs.next()){//光标移动
//                //通过列名称获取列的值
//                String account = rs.getString("account");
//                double balance = rs.getDouble(2);
//                int state = rs.getInt("state");
//                Account a = new Account(account, balance, state);
//                accounts.add(a);
//            }
//            rs.close();
            String updateSql = "UPDATE account SET balance = balance + 1000
WHERE account=123457";
            //执行更新时，返回的都是受影响的行数
            int affectedRows = s.executeUpdate(updateSql);
            System.out.println(affectedRows);
            s.close();
            conn.close();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        accounts.forEach(System.out::println);
    }
}
```

4. 预处理SQL

在日常开发中，我们经常会根据用户输入的信息从数据库中进行数据筛选，现有`stu`表数据如下：

id	name	sex	age
D2021001	张华	男	20
D2021002	李刚	男	20
D2021003	金凤	女	21
D2021004	龙华	男	22

现要根据用户输入的学生姓名查询学生信息。

```
Scanner sc = new Scanner(System.in);
System.out.println("请输入学生姓名：");
String name = sc.next();
String sql = "SELECT id, name, sex, age FROM stu WHERE name='" + name + "'";
```

如果此时用户输入信息为：`张华' or 1='1`，那么，上面的代码执行后`SQL`语句变为：

```
SELECT id, name, sex, age FROM stu WHERE name='张华' or 1='1'
```

明显查询的结果发生了变化，这样的情况被称作为`SQL`注入。为了防止`SQL`注入，Java 提供了`PreparedStatement`接口对`SQL`进行预处理，该接口是`Statement`接口的子接口，其常用方法如下：

```
//执行查询，得到一个结果集
ResultSet executeQuery() throws SQLException;
//执行更新，得到受影响的行数
int executeUpdate() throws SQLException;
//使用给定的整数值设置给定位置的参数
void setInt(int parameterIndex, int x) throws SQLException;
//使用给定的长整数值设置给定位置的参数
void setLong(int parameterIndex, long x) throws SQLException;
//使用给定的双精度浮点数值设置给定位置的参数
void setDouble(int parameterIndex, double x) throws SQLException;
//使用给定的字符串值设置给定位置的参数
void setString(int parameterIndex, String x) throws SQLException;
//使用给定的对象设置给定位置的参数
void setObject(int parameterIndex, Object x) throws SQLException;
//获取结果集元数据
ResultSetMetaData getMetaData() throws SQLException;
```

如何获取`PreparedStatement`接口对象呢?

```
PreparedStatement ps = connection.prepareStatement(sql);
```

`PreparedStatement`是如何进行预处理的?

使用`PreparedStatement`时，`SQL`语句中的参数一律使用`?`号来进行占位，然后通过调用setXxx()方法来对占位的`?`号进行替换。从而将参数作为一个整体进行查询。

上面的示例使用`PreparedStatement`编写`SQL`语句为：

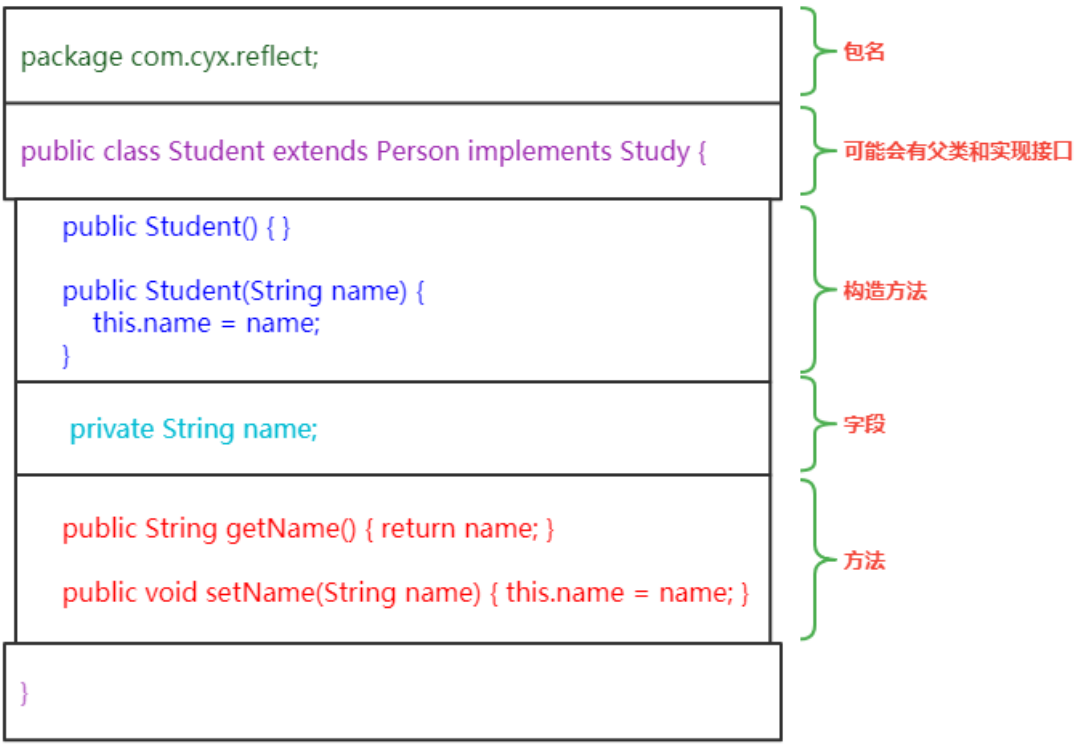
```
Scanner sc = new Scanner(System.in);
System.out.println("请输入学生姓名：");
String name = sc.next();
String sql = "SELECT id, name, sex, age FROM stu WHERE name=?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, name);
ResultSet rs = ps.executeQuery();
```

第二节 反射

1. Class类

我们编写的 Java 程序先经过编译器编译，生成class文件，而class文件的执行场所是在`JVM`中，那么`JVM`如何存储我们编写的类的信息?

首先我们回想一下，一个类有哪些组成部分?



如果要定义一个类来描述所有类的共有特征，应该如何来设计？

```
public class Class {

    private String name; //类名

    private Package pk; //包名

    private Constructor[] constructors; //构造方法，因为可能存在多个，所以使用数组

    private Field[] fields; //字段，因为可能存在多个，所以使用数组

    private Method[] methods; //方法，因为可能存在多个，所以使用数组

    private Class<?> interfaces; //实现的接口，因为可能存在多个，所以使用数组

    private Class<?> superClass; //继承的父类

    //省略getter/setter
}
```

为什么要设计这样的类？因为我们编写的程序从本质上来说也是文件，`JVM`加载类的过程相当于对文件内容进行解析，解析内容就需要找到共有特征（Class 类定义），然后再将这特征（使用 Class 对象）存储起来，在使用的时候再取出来。通过 Class 对象反向推到我们编写的类的内容，然后再进行操作，这个过程就称为反射。

在`JDK`中已经提供了这样的类：`java.lang.Class`，因此，我们不需要再来设计，只需要学习它即可。

如何获取一个类对应的 Class 对象呢？

```
Class<类名> clazz = 类名.class;
Class<类名> clazz = 对象名.getClass();
Class<类名> clazz = clazz.getSuperClass();
Class clazz = Class.forName("类的全限定名");//类的全限定名=包名 + "." + 类名
Class<类名> clazz = 包装类.TYPE;
```

Class 类常用方法

```
//获取类中使用public修饰的字段
public Field[] getFields() throws SecurityException;
//获取类中定义的所有字段
public Field[] getDeclaredFields() throws SecurityException;
//通过给定的字段名获取类中定义的字段
public Field getField(String name) throws NoSuchFieldException,
SecurityException;
//获取类中使用public修饰的方法
public Method[] getMethods() throws SecurityException;
//获取类中定义的所有方法
public Method[] getDeclaredMethods() throws SecurityException;
//通过给定的方法名和参数列表类型获取类中定义的方法
public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
        throws NoSuchMethodException, SecurityException;
//获取类中使用public修饰的构造方法
public Constructor<?>[] getConstructors() throws SecurityException;
//通过给定的参数列表类型获取类中定义的构造方法
public Constructor<T> getConstructor(Class<?>... parameterTypes)
        throws NoSuchMethodException, SecurityException;
//获取类的全限定名
public String getName();
//获取类所在的包
public Package getPackage();
//判断该类是否是基本数据类型
public native boolean isPrimitive();
//判断该类是否是接口
public native boolean isInterface();
//判断该类是否是数组
public native boolean isArray();
//通过类的无参构造创建一个实例
public T newInstance() throws InstantiationException, IllegalAccessException;
```

`java.lang.reflect.AccessibleObject`

```
//修改访问权限
public void setAccessible(boolean flag) throws SecurityException;
```

示例

```
package com.cyx.jdbc.reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;
```

```
public class ReflectionTest {

    public static void main(String[] args) {
        //构建一个学生对象，并为每个字段赋值
        Class<Student> clazz = Student.class;
        try {
            Constructor<? extends Student> c = clazz.getDeclaredConstructor();
            //Student类中的无参构造方法是私有的，因此需要先修改访问权限
            c.setAccessible(true);
            Student s = c.newInstance();
            Field nameField = clazz.getDeclaredField("name");
            nameField.setAccessible(true);
            //给指定对象中的该字段赋值
            nameField.set(s, "李四");

            Field ageField = clazz.getDeclaredField("age");
            ageField.setAccessible(true);
            ageField.set(s, 20);

            // get name => get + N + ame
            String fieldName = nameField.getName();
            String methodName = "get"+ fieldName.substring(0,1).toUpperCase() +
fieldName.substring(1);
            Method m = clazz.getDeclaredMethod(methodName);
            m.setAccessible(true);
            String name = (String) m.invoke(s);
            System.out.println(name);

            methodName = "set" + fieldName.substring(0,1).toUpperCase() +
fieldName.substring(1);
            m = clazz.getDeclaredMethod(methodName, nameField.getType());
            m.invoke(s, "李刚");
            System.out.println(s);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void getMethod(){
        Class<Student> clazz = Student.class;
        Method[] methods = clazz.getDeclaredMethods();
        for(Method method: methods){
            System.out.print(method.getModifiers() + " ");
            System.out.print(method.getName() + " (");
            Class[] types = method.getParameterTypes();
            for(Class c: types){
                System.out.print(c.getName() + ",");
            }
            System.out.println(")");
        }
        System.out.println("============================");
        try {
            Method method = clazz.getDeclaredMethod("setName", String.class);
            System.out.print(method.getModifiers() + " ");
            System.out.print(method.getName() + " (");
            Class[] types = method.getParameterTypes();
            for(Class c: types){
                System.out.print(c.getName() + ",");
```

```
            }
            System.out.println(")");
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }

    private static void getField(){
        Class<Student> clazz = Student.class;
        Field[] fields = clazz.getDeclaredFields();
        for(Field f: fields){
            System.out.print(f.getModifiers() + " ");
            System.out.print(f.getType().getName() + " ");
            System.out.println(f.getName());
        }
        System.out.println("==========================");
        try {
            Field f = clazz.getDeclaredField("name");
            System.out.print(f.getModifiers() + " ");
            System.out.print(f.getType().getName() + " ");
            System.out.println(f.getName());
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        }
    }

    private static void getConstructor(){
        Class<Student> clazz = Student.class;
        //获取在类中定义的构造方法
        Constructor[] constructors = clazz.getDeclaredConstructors();
        for(Constructor c: constructors){
            System.out.println(c.getModifiers());
            String name = c.getName(); //构造方法的名字
            Class[] types = c.getParameterTypes();
            System.out.print(name + " ");
            System.out.println(Arrays.toString(types));
        }
        System.out.println("============================");
        constructors = clazz.getConstructors();
        for(Constructor c: constructors){
            System.out.println(c.getModifiers());
            String name = c.getName(); //构造方法的名字
            Class[] types = c.getParameterTypes();
            System.out.print(name + " ");
            System.out.println(Arrays.toString(types));
        }
        System.out.println("============================");
        try {
            Constructor c = clazz.getDeclaredConstructor(String.class,
int.class);
            System.out.println(c.getModifiers());
            String name = c.getName(); //构造方法的名字
            Class[] types = c.getParameterTypes();
            System.out.print(name + " ");
            System.out.println(Arrays.toString(types));
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
```

```
    }

    private static void getClazz(){
        Class<Student> c1 = Student.class;
        System.out.println(c1.getName());
        Student stu = new Student("张三", 20);
        Class<? extends Student> c2 = stu.getClass();
        //获取父类
        Class<? super Student> c3 = c1.getSuperclass();
        System.out.println(c3.getName());
        try {
            Class c4 = Class.forName("com.cyx.jdbc.reflection.Student");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Class c5 = Integer.TYPE;
        Class c6 = int.class;
        System.out.println(c5.getName());
    }
}
```

2. 反射与数据库

数据库查询出的每一条数据基本上都会封装为一个对象，数据库中的每一个字段值都会存储在对象相应的属性中。如果查询结果的每一个字段都与对象中的属性名保持一致，那么就可以使用反射来完成万能查询。

`JdbcUtil` 构建演示

```
package com.cyx.jdbc.reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class JdbcUtil {
    private static final String url = "jdbc:mysql://localhost:3306/lesson?
serverTimezone=Asia/Shanghai";
    private static final String username = "root";
    private static final String password = "root";

    static {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.out.println("驱动程序未加载");
        }
    }
```

```
    public static void main(String[] args) {
        String sql = "SELECT id,name,number,price,agent_id agentId FROM goods
WHERE name LIKE ? AND price > ?";
        Object[] params = { "%魅%", 1000};
        List<Goods> goodsList = query(sql, Goods.class, params);
        goodsList.forEach(System.out::println);

        sql = "SELECT id,name,region_id regionId FROM agent WHERE name LIKE ?";
        params = new Object[]{"%魅%"};
        List<Agent> agents = query(sql, Agent.class, params);
        agents.forEach(System.out::println);
    }

    public static int update(String sql, Object...params){
        int result = 0;
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            conn = DriverManager.getConnection(url, username, password);
            ps = createPreparedStatement(conn, sql, params);
            result = ps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            close(ps, conn);
        }
        return result;
    }

    private static PreparedStatement createPreparedStatement(Connection conn,
String sql, Object...params) throws SQLException {
        PreparedStatement ps = conn.prepareStatement(sql);
        if(params != null && params.length > 0){
            for(int i=0; i<params.length; i++){
                ps.setObject(i+1, params[i]);
            }
        }
        return ps;
    }

    /**
     * 关闭连接、执行器、结果集
     * @param closeables
     */
    private static void close(AutoCloseable... closeables){
        if(closeables != null && closeables.length > 0){
            for(AutoCloseable ac: closeables){
                if(ac != null){
                    try {
                        ac.close();
                    } catch (Exception e) {
                    }
                }
            }
        }
    }
    /**
     * 万能查询通过反射实现，必须要保证类中定义字段名与查询结果展示的列名称保持一致
```

```
     * @param sql
     * @param clazz
     * @param params
     * @param <T>
     * @return
     */
    public static<T> List<T> query(String sql,Class<T> clazz, Object...params){
        List<T> dataList = new ArrayList<>();
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = DriverManager.getConnection(url, username, password);
            ps = createPreparedStatement(conn, sql, params);
            rs = ps.executeQuery();
            while (rs.next()){
                T t = createInstance(clazz, rs);
                dataList.add(t);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            close(rs, ps, conn);
        }
        return dataList;
    }

    private static<T> T createInstance(Class<T> clazz, ResultSet rs) throws
Exception{
        Constructor<T> c = clazz.getConstructor();//获取无参构造
        T t = c.newInstance();//创建对象
        Field[] fields = clazz.getDeclaredFields(); //获取类中定义的字段
        for(Field field: fields){
            String fieldName = field.getName();
            //setId => set id => set + I + d
            String methodName = "set" + fieldName.substring(0, 1).toUpperCase()
+ fieldName.substring(1);
            Method m = clazz.getDeclaredMethod(methodName, field.getType());
            try {
                Object value = rs.getObject(fieldName, field.getType());
                m.invoke(t, value);
            } catch (Exception e){}
        }
        return t;
    }

//    public static List<Goods> getGoods(){
//        String url = "jdbc:mysql://localhost:3306/lesson?
serverTimezone=Asia/Shanghai";
//        String username = "root";
//        String password = "root";
//        List<Goods> goodsList = new ArrayList<>();
//        try {
//            Class.forName("com.mysql.cj.jdbc.Driver");
//            Connection conn = DriverManager.getConnection(url, username,
password);
//            String sql = "SELECT id,name,number,price,agent_id FROM goods
WHERE name LIKE ? AND price > ?";
```

```
//            PreparedStatement ps = conn.prepareStatement(sql);
//            ps.setString(1, "%魅%");
//            ps.setDouble(2, 1000.00);
//            ResultSet rs = ps.executeQuery();
//            while (rs.next()){
//                Goods goods = new Goods();
//                goods.setId(rs.getLong("id"));
//                goods.setName(rs.getString("name"));
//                goods.setNumber(rs.getInt("number"));
//                goods.setPrice(rs.getDouble("price"));
//                goods.setAgentId(rs.getLong("agent_id"));
//                goodsList.add(goods);
//            }
//            rs.close();
//            ps.close();
//            conn.close();
//        } catch (Exception e) {
//            e.printStackTrace();
//        }
//        goodsList.forEach(System.out::println);
//        return goodsList;
//    }
//
//    public static List<Agent> getAgents(){
//        String url = "jdbc:mysql://localhost:3306/lesson?
serverTimezone=Asia/Shanghai";
//        String username = "root";
//        String password = "root";
//        List<Agent> agents = new ArrayList<>();
//        try {
//            Class.forName("com.mysql.cj.jdbc.Driver");
//            Connection conn = DriverManager.getConnection(url, username,
password);
//            String sql = "SELECT id,name,region_id FROM agent WHERE name LIKE
?";
//            PreparedStatement ps = conn.prepareStatement(sql);
//            ps.setString(1, "%魅%");
//            ResultSet rs = ps.executeQuery();
//            while (rs.next()){
//                Agent agent = new Agent();
//                agent.setId(rs.getLong("id"));
//                agent.setName(rs.getString("name"));
//                agent.setRegionId(rs.getInt("region_id"));
//                agents.add(agent);
//            }
//            rs.close();
//            ps.close();
//            conn.close();
//        } catch (Exception e) {
//            e.printStackTrace();
//        }
//        agents.forEach(System.out::println);
//        return agents;
//    }
}
```