# JDBC简介和用JDBC连接数据库

大家好，我是一个爱举铁的程序员Shr。

本篇文章介绍JDBC，还有用JDBC连接数据库添加数据，查询数据。

# 什么是JDBC

JDBC的全称是Java DataBase Connectivity，Java数据库连接，它一套用来连接数据库的规范。我个人的理解JDBC是Java程序和数据库之间的桥梁。

# 为什么要使用JDBC

它可以和数据库建立连接，发送操作数据库的SQL语句，接收SQL语句的执行结果。

# JDBC中的重要元素

## 3.1 驱动包

项目中导入驱动包才能使用JDBC，连接不同的数据库需要对应的Jar包。

SQL Server导入sqljdbc.jar

MySQL导入mysql-connector-java-5.0.4-bin.jar（5.0.4是版本号）

Oracle导入ojdbc6.jar

## 3.2 连接地址

连接数据库系统的URL是不一样的。

SQL Server ：jdbc:sqlserver://localhost:1433;DatabaseName=数据库名

MySQL ：jdbc:mysql://127.0.0.1:3306/数据库名

Oracle ：jdbc:oracle:thin:@localhost:1521:orcl（orcl是Oracle数据库实例）

## 3.3 用户名和密码

登录数据库系统的用户名和密码

## 3.4 DriverManager类

管理数据库中所有驱动程序。

## 3.5 Connection接口

与数据库的连接，通过DriverManager类获取。

Connection connection = riverManager.getConnection();

## 3.6 Statement接口

在建立连接的基础上向数据库发送SQL语句，用于执行静态的SQL语句。通过连接对象获取。

Statement statement = connection .createStatement();

## 3.7 PreparedStatement接口

该接口继承Statement接口，用于执行动态的SQL语句。通过连接对象获取。

PreparedStatement prepareStatement = connection .prepareStatement(sql);

## 3.8 ResultSet接口

类似一张数据表，用来暂时存放数据库查询操作所获得的结果集。

ResultSet resultSet = statement.executeQuery(sql);

# JDBC操作数据

## 4.1 创建数据库，新建表

创建数据库dao\_20180603，新建表jdbc01\_student。

CREATE TABLE `jdbc01\_student` (

`stu\_id` int(11) NOT NULL,

`stu\_name` varchar(20) default NULL,

`stu\_age` int(11) default NULL,

`stu\_gender` int(11) default NULL,

PRIMARY KEY (`stu\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 4.2 连接数据库

**public** **class** DBUtil {

/\* 数据库驱动，其实就是对应的jar中的驱动类的全路径

MySQL: com.mysql.jdbc.Driver

SQL Server: com.microsoft.sqlserver.jdbc.SQLServerDriver

Oracle 11g: oracle.jdbc.driver.OracleDriver

\*/

**private** **static** **final** String ***DRIVE\_STRING*** = "com.mysql.jdbc.Driver";

/\* 数据库连接URL，数据库名为dao\_20180603

MySQL: jdbc:mysql://localhost:3306/dao\_20180603

SQL Server: jdbc:sqlserver://localhost:1433;databaseName=dao\_20180603

Oracle 11g: jdbc:oracle:thin:@localhost:1521:orcl

\*/

**private** **static** **final** String ***URL\_STRING*** = "jdbc:mysql://localhost:3306/dao\_20180603?characterEncoding=utf8";

// 数据库用户名

**private** **static** **final** String ***USER\_STRING*** = "root";

// 数据库密码

**private** **static** **final** String ***PASSWORD\_STRING*** = "shrmus";

// 数据库连接对象

**public** **static** Connection *connection*;

// SQL执行对象

**public** **static** Statement *statement*;

// 结果集对象

**public** **static** ResultSet *resultSet*;

// 加载驱动

**static** {

**try** {

Class.*forName*(***DRIVE\_STRING***);

} **catch** (ClassNotFoundException e) {

System.***err***.println("加载数据库驱动失败！");

}

}

/\*\*

\* 获取数据库连接对象

\* **@return**

\*/

**public** **static** Connection getConnection() {

**try** {

*connection* = DriverManager.*getConnection*(***URL\_STRING***, ***USER\_STRING***, ***PASSWORD\_STRING***);

} **catch** (SQLException e) {

System.***err***.println("获取数据库连接对象失败！");

}

**return** *connection*;

}

/\*\*

\* 关闭数据库连接对象

\* **@param** connection

\*/

**public** **static** **void** close(Connection connection) {

**if**(**null** != connection) {

**try** {

connection.close();

} **catch** (SQLException e) {

System.***err***.println("关闭数据库连接对象失败！");

}

}

}

/\*\*

\* 关闭SQL执行对象

\* **@param** statement

\*/

**public** **static** **void** close(Statement statement) {

**if**(**null** != statement) {

**try** {

statement.close();

} **catch** (SQLException e) {

System.***err***.println("关闭SQL执行对象失败！");

}

}

}

/\*\*

\* 关闭结果集对象

\* **@param** resultSet

\*/

**public** **static** **void** close(ResultSet resultSet) {

**if**(**null** != resultSet) {

**try** {

resultSet.close();

} **catch** (SQLException e) {

System.***err***.println("关闭结果集对象失败！");

}

}

}

/\*\*

\* 关闭数据库连接对象，SQL执行对象，结果集对象

\* **@param** resultSet

\* **@param** statement

\* **@param** connection

\*/

**public** **static** **void** close(ResultSet resultSet, Statement statement, Connection connection){

**if** (resultSet != **null**) {

**try** {

resultSet.close();

} **catch** (SQLException e) {

System.***err***.println("关闭结果集对象失败！");

}

}

**if** (statement != **null**) {

**try** {

statement.close();

} **catch** (SQLException e) {

System.***err***.println("关闭SQL执行对象失败！");

}

}

**if** (connection != **null**) {

**try** {

connection.close();

} **catch** (SQLException e) {

System.***err***.println("关闭数据库连接对象失败！");

}

}

}

}

## 4.3 添加一条记录到数据库

### 4.3.1 新建一个实体类

**public** **class** Student {

**private** **int** id;

**private** String name;

**private** **int** age;

**private** **int** gender;

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + ", age=" + age + ", gender=" + gender + "]";

}

**public** Student() {

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **int** getGender() {

**return** gender;

}

**public** **void** setGender(**int** gender) {

**this**.gender = gender;

}

}

### 4.3.2 新建一个Dao类

**public** **class** StudentDao {

/\*\*

\* 添加一条记录

\* **@param** student

\* **@return**

\*/

**public** **int** addStudent(Student student) {

// 受影响的行数

**int** acceptRow = 0;

// 获取数据库连接对象

Connection connection = DBUtil.*getConnection*();

**try** {

// 获取SQL执行对象

Statement statement = connection.createStatement();

// sql语句

String sql = "insert into jdbc01\_student(stu\_id,stu\_name,stu\_age,stu\_gender) "

+ "values("+student.getId()+",'"+student.getName()+"',"+student.getAge()+","+student.getGender()+")";

System.***out***.println(sql);

acceptRow = statement.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** acceptRow;

}

}

### 4.3.3 新建一个Service类

**public** **class** StudentService {

/\*\*

\* 添加一个学生的信息

\* **@param** student

\* **@return**

\*/

**public** **int** addStudent(Student student) {

StudentDao studentDao = **new** StudentDao();

**int** accpetRow = studentDao.addStudent(student);

**return** accpetRow;

}

}

### 4.3.4 新建一个测试类

**public** **class** StudentTest {

@Test

**public** **void** addStudent() **throws** Exception{

StudentService studentService = **new** StudentService();

Student student = **new** Student();

student.setId(1);

student.setName("张三");

student.setAge(20);

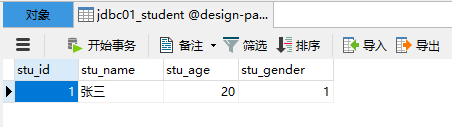
student.setGender(1);

studentService.addStudent(student);

}

}

执行测试类中的addStudent()方法就往数据库中添加了一条记录。

数据库中能查找到这条记录，那怎么在程序中查找到这条记录呢。接下来要讲的就是查找数据库中的记录。

## 4.4 查询数据库中的记录

### 4.4.1 在Dao类中添加查询方法

/\*\*

\* 查找所有学生

\* **@return**

\*/

**public** List<Student> getStudentList(){

List<Student> studentList = **new** ArrayList<>();

// 获取数据库连接对象

Connection connection = DBUtil.*getConnection*();

**try** {

// 获取SQL执行对象

Statement statement = connection.createStatement();

// sql语句

String sql = "select stu\_id,stu\_name,stu\_age,stu\_gender from jdbc01\_student";

// 获取结果集

ResultSet resultSet = statement.executeQuery(sql);

**while** (resultSet.next()) {

Student student = **new** Student();

// 根据列名获取值

String idString = resultSet.getString("stu\_id");

String nameString = resultSet.getString("stu\_name");

String ageString = resultSet.getString("stu\_age");

String genderString = resultSet.getString("stu\_gender");

**int** id = Integer.*parseInt*(idString);

**int** age = Integer.*parseInt*(ageString);

**int** gender = Integer.*parseInt*(genderString);

// 封装学生的信息

student.setId(id);

student.setName(nameString);

student.setAge(age);

student.setGender(gender);

// 添加到集合中

studentList.add(student);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** studentList;

}

### 4.4.2 在Service类中添加查询方法

/\*\*

\* 查找所有学生

\* **@return**

\*/

**public** List<Student> getStudentList(){

StudentDao studentDao = **new** StudentDao();

List<Student> studentList = studentDao.getStudentList();

**return** studentList;

}

### 4.4.3 在测试类中添加查询方法

@Test

**public** **void** getStudentList() **throws** Exception{

StudentService studentService = **new** StudentService();

List<Student> studentList = studentService.getStudentList();

**for**(Student student : studentList) {

System.***out***.println(student);

}

}

执行测试类中的查询方法就可以看到控制台打印的信息了。

Student [id=1, name=张三, age=20, gender=1]

修改和删除的代码我就不写了，读者可以自己根据查询和添加自己练习。

# JDBC元数据简介

大家好，我是一个爱举铁的程序员Shr。

本篇文章将会介绍JDBC中的元数据。

# 一、什么是元数据？

描述数据库或其组成部分的数据成为元数据（区别于那些存在数据库中的实际数据）[1]。

# 二、元数据的分类

通过JDBC可以获得三种元数据，数据库元数据，SQL语句参数元数据，结果集元数据。

# 三、举个栗子

## 3.1 新建表

在数据库dao\_20180603中新建表emp。

CREATE TABLE `emp` (

`id` int(11) NOT NULL,

`name` varchar(255) default NULL,

`address` varchar(255) default NULL,

`hireDate` datetime default NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 3.2 新建类

**public** **class** Emp {

**private** **int** id;

**private** String name;

**private** String address;

**private** Date hireDate;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** Date getHireDate() {

**return** hireDate;

}

**public** **void** setHireDate(Date hireDate) {

**this**.hireDate = hireDate;

}

}

## 3.3 测试

### 3.3.1 获取数据库元数据 JDBCUtil类是JDBC简介那篇文章中写过的DBUtil，只是换了类名。

**public** **class** MetaDataTest {

/\*\*

\* 获取数据库元数据

\*/

@Test

**public** **void** getDatabaseMetaData() **throws** Exception{

Connection connection = JDBCUtil.*getConnection*();

DatabaseMetaData databaseMetaData = connection.getMetaData();

// 获取数据库名称

String databaseProductName = databaseMetaData.getDatabaseProductName();

System.***out***.println(databaseProductName);

// 获取驱动版本

String driverName = databaseMetaData.getDriverVersion();

System.***out***.println(driverName);

// 获取数据库连接URL

String url = databaseMetaData.getURL();

System.***out***.println(url);

// 获取用户名

String userName = databaseMetaData.getUserName();

System.***out***.println(userName);

JDBCUtil.*close*(**null**, **null**, connection);

}

}

运行结果：

MySQL

mysql-connector-java-5.1.44 ( Revision: b3cda4f864902ffdde495b9df93937c3e20009be )

jdbc:mysql://localhost:3306/dao\_20180603?characterEncoding=utf8

root@

### 3.3.2 获取SQL执行对象元数据

/\*\*

\* 获取SQL执行对象元数据

\*/

@Test

**public** **void** getParameterMetaData() **throws** Exception {

Connection connection = JDBCUtil.*getConnection*();

String sql = "insert into emp(id,name,address,hiredate) values(?,?,?,?)";

PreparedStatement prepareStatement = connection.prepareStatement(sql);

ParameterMetaData parameterMetaData = prepareStatement.getParameterMetaData();

// 取到参数个数

**int** parameterCount = parameterMetaData.getParameterCount();

System.***out***.println(parameterCount);

JDBCUtil.*close*(**null**, prepareStatement, connection);

}

运行结果：

4

### 3.3.3 获取结果集元数据

/\*\*

\* 获取结果集元数据

\*/

@Test

**public** **void** getResultSetMetaData() **throws** Exception{

// 获取数据库连接对象

Connection connection = JDBCUtil.*getConnection*();

// sql语句

String sql = "select id,name,address,hiredate from emp";

// 获取SQL执行对象

PreparedStatement prepareStatement = connection.prepareStatement(sql);

// 获取结果集

ResultSet resultSet = prepareStatement.executeQuery();

// 获取结果集元数据

ResultSetMetaData resultSetMetaData = resultSet.getMetaData();

// 获取查询出来的总列数

**int** columnCount = resultSetMetaData.getColumnCount();

System.***out***.println("columnCount : " + columnCount);

// 获取指定列的类型常量，在java.sql.Types类中可以查找到

**int** columnType = resultSetMetaData.getColumnType(2);

System.***out***.println("columnType : " + columnType);

// 获取指定列的类型字符串，如INT,VARCHAR

String columnTypeName = resultSetMetaData.getColumnTypeName(2);

System.***out***.println("columnTypeName : " + columnTypeName);

// 获取指定列的列名，如hiredate

String columnName = resultSetMetaData.getColumnName(4);

System.***out***.println("columnName : " + columnName);

JDBCUtil.*close*(resultSet, prepareStatement, connection);

}

运行结果：

columnCount : 4

columnType : 12

columnTypeName : VARCHAR

columnName : hireDate

# 参考文献

[1] (美)霍斯特曼(Horstmann),科内尔(Cornell).Java核心技术卷2:高级特性(原书第8版)[M].机械工业出版社,2008

# 反射简介和反射测试

大家好，我是一个爱举铁的程序员Shr。

本篇文章介绍反射。

# 一、什么是反射？

能够分析类能力的程序成为反射[1]。

# 二、反射可以用来做什么？

在运行中分析类的能力。

在运行中查看对象，例如，编写一个toString方法供所有类使用。

实现通用的数组操作代码。

利用Method对象，这个对象很像C++中的函数指针。

# 三、反射中的重要元素

## 3.1 Class类

Class类能够保存每个类的信息。

Object类中的getClass()方法可以返回一个Class类的实例。

newInstance() 方法可以创建Class对象所表示的类的一个新实例。

## 3.2 Field类

Field类保存类中的字段的信息。

clazz.getFields()方法可以获取类中所有可访问的公共字段。

## 3.3 Constructor类

Constructor类保存类的构造方法的信息。

clazz.getDeclaredConstructor(Class<?>... parameterTypes)可以获取类的所有带参数或不带参数的构造方法。

## 3.4 Method类

Method类保存类的方法的信息。

clazz.getMethod(String name, Class<?>... parameterTypes)方法可以获得类的所有带参数或者不带参数的公共方法。

method.invoke(Object obj, Object... args)用来执行方法。第一个参数是底层调用该方法的对象，第二个可变参数就是方法的形参。

## 3.5 Modifier类

Modifier类提供了static方法和常量，对类和成员访问修饰符进行解码。

## 3.6 Type接口

Type接口保存的是类型信息，包括原始类型、参数化类型、数组类型、类型变量和基本类型。

# 四、反射实战操作

## 4.1 反射实例化新对象

### 4.1.1 新建一个类，只写字段

**public** **class** Student {

**private** **int** id;

**private** String name;

**protected** **int** age;

**public** **int** gender;

}

### 4.1.2 写一个测试类

**public** **class** StudentTest {

@Test

**public** **void** newInstance() **throws** Exception{

// 原对象

Student student = **new** Student();

System.***out***.println(student);

// 原对象的Class对象

Class<? **extends** Student> clazz = student.getClass();

// 实例化一个新对象

Student newInstance = clazz.newInstance();

System.***out***.println(newInstance);

// 获取类的全路径

String name = clazz.getName();

System.***out***.println(name);

}

}

运行结果：

com.shrmus.reflex.Student@69d0a921

com.shrmus.reflex.Student@446cdf90

com.shrmus.reflex.Student

可以看到第一个对象的地址和第二个对象的地址不一样，所以证实了反射实例化的是一个新对象。

## 4.2 反射获取字段

### 4.2.1 测试获取字段

/\*\*

\* 反射获取字段

\*/

@Test

**public** **void** getDeclaredFields() **throws** Exception {

Class clazz = Student.**class**;

// 根据字段名获取字段的信息

Field field1 = clazz.getDeclaredField("name");

System.***out***.println(field1.toGenericString());

// 获取类中的所有字段

Field[] fields = clazz.getDeclaredFields();

**for**(Field field : fields) {

System.***out***.println(Modifier.*toString*(field.getModifiers()) + " " + field.getGenericType().getTypeName() + " " + clazz.getName() + "." + field.getName());

}

}

运行结果：

private java.lang.String com.shrmus.reflex.Student.name

private int com.shrmus.reflex.Student.id

private java.lang.String com.shrmus.reflex.Student.name

protected int com.shrmus.reflex.Student.age

public int com.shrmus.reflex.Student.gender

## 4.3 反射获取构造方法

### 4.3.1 添加构造方法

**public** Student() {

}

**public** Student(**int** id) {

**this**.id = id;

}

**public** Student(String name) {

**this**.name = name;

}

**private** Student(**int** id,String name) {

**this**.id = id;

**this**.name = name;

}

**protected** Student(**int** id,String name,**int** age) {

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

**public** Student(**int** id,String name,**int** age,**int** gender) {

**this**.id = id;

**this**.name = name;

**this**.age = age;

**this**.gender = gender;

}

### 4.3.2 测试获取构造方法

/\*\*

\* 反射获取构造方法

\*/

@Test

**public** **void** getDeclaredConstructors() **throws** Exception {

Class clazz = Student.**class**;

// 根据构造方法的参数个数获取构造方法

Constructor constructor1 = clazz.getDeclaredConstructor(Integer.***TYPE***,String.**class**,Integer.***TYPE***);

System.***out***.println(constructor1.toGenericString());

// 获取所有构造方法

Constructor<?>[] constructors = clazz.getDeclaredConstructors();

**for**(Constructor constructor : constructors) {

System.***out***.print(Modifier.*toString*(constructor.getModifiers()) + " " + constructor.getName());

// 获取参数类型

Type[] genericParameterTypes = constructor.getGenericParameterTypes();

System.***out***.print("(");

**if**(genericParameterTypes.length > 0) {

**int** length = genericParameterTypes.length;

**for**(**int** i = 0; i < length - 1; i++) {

String typeName = genericParameterTypes[i].getTypeName();

System.***out***.print(typeName + ", ");

}

System.***out***.print(genericParameterTypes[length - 1].getTypeName());

}

System.***out***.print(")");

System.***out***.println();

}

}

运行结果：

protected com.shrmus.reflex.Student(int,java.lang.String,int)

public com.shrmus.reflex.Student(int, java.lang.String, int, int)

protected com.shrmus.reflex.Student(int, java.lang.String, int)

private com.shrmus.reflex.Student(int, java.lang.String)

public com.shrmus.reflex.Student()

public com.shrmus.reflex.Student(int)

public com.shrmus.reflex.Student(java.lang.String)

## 4.4 反射获取成员方法

### 4.4.1 添加成员方法

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **int** getGender() {

**return** gender;

}

**public** **void** setGender(**int** gender) {

**this**.gender = gender;

}

**private** **void** test01() **throws** RuntimeException{

}

**protected** **void** test02() **throws** SQLException{

}

**public** **static** String test03() {

**return** "test03()";

}

**public** **static** **final** Student test04(String string1,Student student) {

System.***out***.println(string1 + " " + student.getName());

**return** student;

}

### 4.4.2 测试获取成员方法

/\*\*

\* 反射获取类的所有方法

\*/

@Test

**public** **void** getDeclaredMethods() **throws** Exception {

Class<? **extends** Student> clazz = Student.**class**;

// 根据方法名获取方法对象

Method declaredMethod = clazz.getDeclaredMethod("test04",String.**class**,Student.**class**);

Student student = clazz.newInstance();

// 执行方法

String string = "haha";

student.setName("张三");

Object invoke = declaredMethod.invoke(student,string,student);

System.***out***.println(invoke);

// 获取所有方法，不包括构造方法

Method[] declaredMethods = clazz.getDeclaredMethods();

**for**(Method method : declaredMethods) {

System.***out***.print(Modifier.*toString*(method.getModifiers()) + " " + method.getGenericReturnType().getTypeName() + " " + clazz.getName() + "." + method.getName());

// 获取参数类型

Type[] genericParameterTypes = method.getGenericParameterTypes();

System.***out***.print("(");

**if**(genericParameterTypes.length > 0) {

**int** length = genericParameterTypes.length;

**for**(**int** i = 0; i < length - 1; i++) {

String typeName = genericParameterTypes[i].getTypeName();

System.***out***.print(typeName + ", ");

}

System.***out***.print(genericParameterTypes[length - 1].getTypeName());

}

System.***out***.print(")");

System.***out***.println();

}

}

运行结果：

haha 张三

com.shrmus.reflex.Student@446cdf90

public java.lang.String com.shrmus.reflex.Student.getName()

public int com.shrmus.reflex.Student.getId()

public void com.shrmus.reflex.Student.setName(java.lang.String)

public static final com.shrmus.reflex.Student com.shrmus.reflex.Student.test04(java.lang.String, com.shrmus.reflex.Student)

private void com.shrmus.reflex.Student.test01()

public static java.lang.String com.shrmus.reflex.Student.test03()

public int com.shrmus.reflex.Student.getAge()

public int com.shrmus.reflex.Student.getGender()

protected void com.shrmus.reflex.Student.test02()

public void com.shrmus.reflex.Student.setGender(int)

public void com.shrmus.reflex.Student.setId(int)

public void com.shrmus.reflex.Student.setAge(int)

# 参考文献

[1] (美)霍斯特曼(Horstmann),科内尔(Cornell).Java核心技术卷1:基础知识(原书第9版)[M].机械工业出版社,2013

# JDBC+反射教你写自己的Dao层框架

大家好，我是一个爱举铁的程序员Shr。

本篇文章将用到前几篇文章介绍过的知识自定义数据访问层框架，建议看这篇文章之前先去了解JDBC元数据和反射。

如果是初学者，觉得JDBC封装数据太麻烦，一个类十多个字段，重复的代码导致浪费了大量时间，那待会我开车的时候你可要抓紧了。

如果你用Hibernate，Mybatis用了两三年还只是停留在使用的情况，看源码太费劲，看一会就想睡觉，本篇文章将带你走进框架底层，探索精彩的世界。

本篇文章较长，耐心看完~~~

源码地址：https://github.com/ShrMus/Dao/tree/master/dao\_20180603/src/main/java/com/shrmus/jdbc02

# 一、新建数据库表

新建名为dao\_20180603的数据库，再新建emp表

CREATE TABLE `emp` (

`id` int(11) NOT NULL,

`name` varchar(255) default NULL,

`address` varchar(255) default NULL,

`hireDate` datetime default NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

# 二、新建类

**public** **class** Emp {

**private** **int** id;

**private** String name;

**private** String address;

**private** Date hireDate;

**public** Emp() {

}

**public** Emp(**int** id, String name, String address, Date hireDate){

**this**.id = id;

**this**.name = name;

**this**.address = address;

**this**.hireDate = hireDate;

}

@Override

**public** String toString() {

**return** "Emp [id=" + id + ", name=" + name + ", address=" + address + ", hireDate=" + hireDate + "]";

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** Date getHireDate() {

**return** hireDate;

}

**public** **void** setHireDate(Date hireDate) {

**this**.hireDate = hireDate;

}

}

# 三、新建数据源类

**public** **class** MyDataSource {

// 数据库驱动

**private** **final** String driverClassName;

// 数据库连接URL

**private** **final** String url;

// 数据库用户名

**private** **final** String username;

// 数据库密码

**private** **final** String password;

/\*\*

\* 构造方法注入属性值

\*/

**public** MyDataSource(){

Properties properties = **new** Properties();

InputStream inputStream;

**try** {

String path = **this**.getClass().getResource("").getPath();

inputStream = **new** FileInputStream(path + "/jdbc.properties");

properties.load(inputStream);

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

**this**.driverClassName = properties.getProperty("jdbc.driverClassName");

**this**.url = properties.getProperty("jdbc.url");

**this**.username = properties.getProperty("jdbc.username");

**this**.password = properties.getProperty("jdbc.password");

}

**public** MyDataSource(String driverClassName,String url,String username,String password){

**this**.driverClassName = driverClassName;

**this**.url = url;

**this**.username = username;

**this**.password = password;

}

**public** String getDriverClassName() {

**return** driverClassName;

}

**public** String getUrl() {

**return** url;

}

**public** String getUsername() {

**return** username;

}

**public** String getPassword() {

**return** password;

}

}

# 四、新建数据库连接属性文件

新建文件jdbc.properties，文件和数据源类在同一个目录，放在文件内容如下：

jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/dao\_20180603?characterEncoding=utf8

jdbc.username=root

jdbc.password=shrmus

# 五、新建数据库连接类

**public** **final** **class** MyConnection{

**private** Connection connection = **null**;

**public** MyConnection() {

}

**public** MyConnection(MyDataSource myDataSource) {

**try** {

Class.*forName*(myDataSource.getDriverClassName());

connection = DriverManager.*getConnection*(myDataSource.getUrl(), myDataSource.getUsername(), myDataSource.getPassword());

} **catch** (ClassNotFoundException e) {

System.***err***.println("构造MyConnection实例失败！");

} **catch** (SQLException e) {

System.***err***.println("获取MyConnection失败！");

}

}

**public** Connection getConnection() {

**return** connection;

}

/\*\*

\* 关闭数据库连接对象

\* **@param** connection

\*/

**public** **static** **void** close(Connection connection) {

**if**(**null** != connection) {

**try** {

connection.close();

} **catch** (SQLException e) {

System.***err***.println("关闭数据库连接对象失败！");

}

}

}

/\*\*

\* 关闭SQL执行对象

\* **@param** statement

\*/

**public** **static** **void** close(Statement statement) {

**if**(**null** != statement) {

**try** {

statement.close();

} **catch** (SQLException e) {

System.***err***.println("关闭SQL执行对象失败！");

}

}

}

/\*\*

\* 关闭结果集对象

\* **@param** resultSet

\*/

**public** **static** **void** close(ResultSet resultSet) {

**if**(**null** != resultSet) {

**try** {

resultSet.close();

} **catch** (SQLException e) {

System.***err***.println("关闭结果集对象失败！");

}

}

}

/\*\*

\* 关闭数据库连接对象，SQL执行对象，结果集对象

\* **@param** resultSet

\* **@param** statement

\* **@param** connection

\*/

**public** **static** **void** close(ResultSet resultSet, Statement statement, Connection connection){

**if** (resultSet != **null**) {

**try** {

resultSet.close();

} **catch** (SQLException e) {

System.***err***.println("关闭结果集对象失败！");

}

}

**if** (statement != **null**) {

**try** {

statement.close();

} **catch** (SQLException e) {

System.***err***.println("关闭SQL执行对象失败！");

}

}

**if** (connection != **null**) {

**try** {

connection.close();

} **catch** (SQLException e) {

System.***err***.println("关闭数据库连接对象失败！");

}

}

}

}

# 六、新建查询类

这个类用来对数据进行增删改查。

**public** **class** MyQuery {

// 数据库连接对象

**public** Connection connection;

**public** MyQuery() {

MyDataSource myDataSource = **new** MyDataSource();

MyConnection myConnection = **new** MyConnection(myDataSource);

**this**.connection = myConnection.getConnection();

}

**public** MyQuery(MyDataSource myDataSource) {

MyConnection myConnection = **new** MyConnection(myDataSource);

**this**.connection = myConnection.getConnection();

}

}

# 七、测试

看到这里，你已经发现和原先的JDBCUtil完全不一样了，为什么要这么做呢，为了做到功能单一。

后面关键的地方来了。

## 7.1 测试添加员工

### 7.1.1 在查询类中添加代码

/\*\*

\* 插入一条记录

\* **@param** object

\*/

**public** **void** insert(Object object) {

PreparedStatement prepareStatement = **null**;

ResultSet resultSet = **null**;

Class<? **extends** Object> clazz = object.getClass();

// 获取简单类名，数据库表名和类名一致

String simpleName = clazz.getSimpleName();

// 获取字段

Field[] declaredFields = clazz.getDeclaredFields();

String sql = "insert into " + simpleName;

String fieldString = "(";

String valueString = "values(";

// 获取字段的个数

**int** length = declaredFields.length;

**try** {

**for**(**int** i = 0; i < length - 1; i++) {

// 私有字段设置允许访问

declaredFields[i].setAccessible(**true**);

// 获取字段值

Object fieldValue = declaredFields[i].get(object);

String typeName = declaredFields[i].getGenericType().getTypeName();

**if**(typeName.toLowerCase().contains("date")) {

// 如果是日期类型

Date date = (Date) fieldValue;

DateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String format = dateFormat.format(date);

fieldValue = format;

}

// 拼接字段名

fieldString += declaredFields[i].getName() + ",";

// 拼接字段值

valueString += "'" + fieldValue + "',";

}

declaredFields[length - 1].setAccessible(**true**);

// 获取字段值

Object fieldValue = declaredFields[length - 1].get(object);

// 获取字段类型名称

String typeName = declaredFields[length - 1].getGenericType().getTypeName();

**if**(typeName.toLowerCase().contains("date")) {

// 如果是日期类型

Date date = (Date) fieldValue;

DateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String format = dateFormat.format(date);

fieldValue = format;

}

// 拼接字段名

fieldString += declaredFields[length - 1].getName() + ") ";

// 拼接字段值

valueString += "'" + fieldValue + "')";

// 拼接SQL语句

sql = sql + fieldString + valueString;

System.***out***.println("SQL = " + sql);

// 设置事务手动提交

**this**.connection.setAutoCommit(**false**);

prepareStatement = **this**.connection.prepareStatement(sql);

prepareStatement.executeUpdate();

// 提交事务

**this**.connection.commit();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

} **catch** (SQLException e) {

**try** {

// 回滚事务

**this**.connection.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **finally** {

// 关闭连接

MyConnection.*close*(resultSet, prepareStatement, connection);

}

}

### 7.1.2 编写Dao类

**public** **class** EmpDao {

/\*\*

\* 添加员工的信息

\*/

**public** **void** insert(Emp emp) {

MyQuery myQuery = **new** MyQuery();

myQuery.insert(emp);

}

}

### 7.1.3 编写测试类

**public** **class** EmpTest {

/\*\*

\* 测试添加

\*/

@Test

**public** **void** inser() **throws** Exception{

Emp emp = **new** Emp(1,"张三","长沙",**new** Date());

EmpDao empDao = **new** EmpDao();

empDao.insert(emp);

}

}

运行结果：

SQL = insert into Emp(id,name,address,hireDate) values('1','张三','长沙','2018-06-03 23:49:12')

再看数据库：

添加了一条数据。查询类中的inser()方法根据传过来的参数，获取数据库表的主键，再获取字段和字段值，方法内部拼接SQL语句。

## 7.2 测试查询所有员工

### 7.2.1 在查询类中添加代码

/\*\*

\* 查询所有信息

\* **@param** t

\* **@return**

\*/

@SuppressWarnings({ "unchecked", "resource" })

**public** <T> List<T> select(Class<T> t){

PreparedStatement prepareStatement = **null**;

ResultSet resultSet = **null**;

List<T> objectList = **new** ArrayList<>();

// 获取模板T的实例

T newInstance2 = **null**;

**try** {

newInstance2 = t.newInstance();

} **catch** (InstantiationException | IllegalAccessException e1) {

e1.printStackTrace();

}

T newInstance;

// 获取实例的Class类对象

Class<? **extends** Object> clazz = newInstance2.getClass();

// 获取简单类名，数据库表名和类名一致

String simpleName = clazz.getSimpleName();

**try** {

// 获取数据库元数据

DatabaseMetaData databaseMetaData = connection.getMetaData();

// 获取表列

resultSet = databaseMetaData.getColumns(**null**, **null**, simpleName, **null**);

List<String> columnNameList = **new** ArrayList<>();

**while**(resultSet.next()) {

// 获取列名

String columnName = resultSet.getString("COLUMN\_NAME");

columnNameList.add(columnName);

}

// 定义SQL语句

String sql = "select ";

**for**(String columnName : columnNameList) {

sql += columnName + ",";

}

// 删除最后一个逗号

**int** lastIndexOf = sql.lastIndexOf(",");

sql = sql.substring(0, lastIndexOf);

sql += " from " + simpleName;

System.***out***.println("SQL = " + sql);

prepareStatement = **this**.connection.prepareStatement(sql);

// 执行查询

resultSet = prepareStatement.executeQuery();

**while**(resultSet.next()) {

// 创建一个新的实例

newInstance = (T) clazz.newInstance();

// 获取结果集元数据

ResultSetMetaData resultSetMetaData = resultSet.getMetaData();

// 获取查询出来的总列数

**int** columnCount = resultSetMetaData.getColumnCount();

**for**(**int** i = 0; i < columnCount; i++) {

// 获取列名

String columnName = resultSetMetaData.getColumnName(i + 1);

// 获取列值

Object fieldValue = resultSet.getObject(columnName);

// 根据列名获取字段

Field declaredField = clazz.getDeclaredField(columnName);

// 私有字段设置允许访问

declaredField.setAccessible(**true**);

// 调用方法给字段赋新的值

declaredField.set(newInstance, fieldValue);

}

objectList.add(newInstance);

}

} **catch** (SQLException e) {

e.printStackTrace();

} **catch** (NoSuchFieldException e) {

e.printStackTrace();

} **catch** (SecurityException e) {

e.printStackTrace();

} **catch** (InstantiationException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

} **finally** {

// 关闭连接

MyConnection.*close*(resultSet, prepareStatement, connection);

}

**return** objectList;

}

### 7.2.2 编写Dao类

/\*\*

\* 查询所有员工

\*/

**public** List<Emp> getEmpList() {

MyQuery myQuery = **new** MyQuery();

List<Emp> select = myQuery.select(Emp.**class**);

**return** select;

}

### 7.2.3 编写测试类

/\*\*

\* 测试查询所有员工

\*/

@Test

**public** **void** getEmpList() **throws** Exception {

EmpDao empDao = **new** EmpDao();

List<Emp> empList = empDao.getEmpList();

**for**(Emp emp : empList) {

System.***out***.println(emp);

}

}

在运行这个测试方法之前我又添加了一条数据。

运行结果：

SQL = select id,name,address,hireDate from Emp

Emp [id=1, name=张三, address=长沙, hireDate=2018-06-03 23:49:12.0]

Emp [id=2, name=李四, address=株洲, hireDate=2018-06-04 00:00:18.0]

## 7.3 测试根据主键查询员工

### 7.3.1 在查询类添加代码

/\*\*

\* 根据主键查询

\* **@param** t

\* **@return**

\*/

@SuppressWarnings({ "unchecked", "resource" })

**public** <T> T selectByPrimaryKey(T t){

PreparedStatement prepareStatement = **null**;

ResultSet resultSet = **null**;

Class<? **extends** Object> clazz = t.getClass();

// 获取简单类名，数据库表名和类名一致

String simpleName = clazz.getSimpleName();

T newInstance = **null**;

**try** {

// 创建一个新的实例

newInstance = (T) clazz.newInstance();

// 获取数据库元数据

DatabaseMetaData databaseMetaData = connection.getMetaData();

// 获取给定表主键列的描述

resultSet = databaseMetaData.getPrimaryKeys(**null**, **null**, simpleName);

String primaryKey = **null**;

**if**(resultSet.next()) {

// 获取主键列的名称

primaryKey = resultSet.getString("COLUMN\_NAME");

}

// 获取表列

resultSet = databaseMetaData.getColumns(**null**, **null**, simpleName, **null**);

List<String> columnNameList = **new** ArrayList<>();

**while**(resultSet.next()) {

// 获取列名

String columnName = resultSet.getString("COLUMN\_NAME");

columnNameList.add(columnName);

}

**if**(**null** != primaryKey) {

// 获取主键在实体类中的同名字段

Field declaredField = clazz.getDeclaredField(primaryKey);

// 设置私有字段允许访问

declaredField.setAccessible(**true**);

// 获取字段的值

Object primaryKeyValue = declaredField.get(t);

// 获取所有字段

Field[] declaredFields = clazz.getDeclaredFields();

// 定义SQL语句

String sql = "select ";

**for**(String columnName : columnNameList) {

sql += columnName + ",";

}

// 删除最后一个逗号

**int** lastIndexOf = sql.lastIndexOf(",");

sql = sql.substring(0, lastIndexOf);

sql += " from " + simpleName + " where " + primaryKey + "='" + primaryKeyValue + "'";

System.***out***.println("SQL = " + sql);

prepareStatement = connection.prepareStatement(sql);

resultSet = prepareStatement.executeQuery();

**if**(resultSet.next()) {

// 获取结果集元数据

ResultSetMetaData resultSetMetaData = resultSet.getMetaData();

// 获取查找出来的总列数

**int** columnCount = resultSetMetaData.getColumnCount();

**for** (**int** i = 0; i < columnCount; i++) {

// 获取列名

String columnName = resultSetMetaData.getColumnName(i + 1);

// 获取列值

Object columnValue = resultSet.getObject(columnName);

**for**(Field field : declaredFields) {

field.setAccessible(**true**);

// 属性名和字段名一样

**if**(field.getName().equals(columnName)){

// 设置属性值

field.set(newInstance, columnValue);

}

}

}

}

} **else** {

System.***err***.println("主键为空！");

}

} **catch** (SQLException e) {

e.printStackTrace();

} **catch** (NoSuchFieldException e) {

e.printStackTrace();

} **catch** (SecurityException e) {

e.printStackTrace();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

} **catch** (InstantiationException e) {

e.printStackTrace();

} **finally** {

MyConnection.*close*(resultSet, prepareStatement, connection);

}

**return** newInstance;

}

### 7.3.2 编写Dao类

/\*\*

\* 根据主键查询员工的信息

\*/

**public** Emp selectByPrimaryKey(Emp emp) {

MyQuery myQuery = **new** MyQuery();

emp = myQuery.selectByPrimaryKey(emp);

**return** emp;

}

### 7.3.3 编写测试类

/\*\*

\* 测试根据主键查询

\*/

@Test

**public** **void** selectByPrimaryKey() **throws** Exception {

EmpDao empDao = **new** EmpDao();

Emp emp = **new** Emp();

emp.setId(1);

emp = empDao.selectByPrimaryKey(emp);

System.***out***.println(emp);

}

运行结果：

SQL = select id,name,address,hireDate from Emp where id='1'

Emp [id=1, name=张三, address=长沙, hireDate=2018-06-03 23:49:12.0]

## 7.4 测试修改员工的信息

### 7.4.1 编写查询类

/\*\*

\* 根据主键修改记录

\* **@param** object

\*/

**public** **void** updateByPrimaryKey(Object object) {

PreparedStatement prepareStatement = **null**;

ResultSet resultSet = **null**;

Class<? **extends** Object> clazz = object.getClass();

// 获取简单类名，数据库表名和类名一致

String simpleName = clazz.getSimpleName();

**try** {

// 获取数据库元数据

DatabaseMetaData databaseMetaData = connection.getMetaData();

// 获取给定表主键列的描述

resultSet = databaseMetaData.getPrimaryKeys(**null**, **null**, simpleName);

String primaryKey = **null**;

**if**(resultSet.next()) {

// 获取主键列的名称

primaryKey = resultSet.getString("COLUMN\_NAME");

}

**if**(**null** != primaryKey) {

// 获取主键在实体类中的同名字段

Field declaredField = clazz.getDeclaredField(primaryKey);

// 设置私有字段允许访问

declaredField.setAccessible(**true**);

// 获取字段的值

Object primaryKeyValue = declaredField.get(object);

// 获取字段

Field[] declaredFields = clazz.getDeclaredFields();

// 定义SQL语句

String sql = "update " + simpleName + " set ";

**for**(Field field : declaredFields) {

String fieldName = field.getName();

// 这个字段不是主键

**if**(!fieldName.equals(primaryKey)) {

// 设置允许访问

field.setAccessible(**true**);

sql += fieldName + "='";

// 获取属性值

Object fieldValue = field.get(object);

// 获取字段类型名称

String typeName = field.getGenericType().getTypeName();

// 不是基本数据类型

**if**(typeName.toLowerCase().contains("date")) {

// 如果是日期类型

Date date = (Date) fieldValue;

DateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String format = dateFormat.format(date);

sql += format + "', ";

}**else** {

sql += fieldValue + "', ";

}

}

}

// 删除最后一个逗号

**int** lastIndexOf = sql.lastIndexOf(",");

sql = sql.substring(0, lastIndexOf);

sql += " where " + primaryKey + "='" + primaryKeyValue + "' ";

System.***out***.println("SQL = " + sql);

// 开启事务

connection.setAutoCommit(**false**);

prepareStatement = connection.prepareStatement(sql);

prepareStatement.executeUpdate();

// 提交事务

connection.commit();

} **else** {

System.***err***.println("主键为空！");

}

} **catch** (SQLException e) {

**try** {

// 事务回滚

connection.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **catch** (NoSuchFieldException e) {

e.printStackTrace();

} **catch** (SecurityException e) {

e.printStackTrace();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

} **finally** {

MyConnection.*close*(resultSet, prepareStatement, connection);

}

}

### 7.4.2 编写Dao类

/\*\*

\* 根据主键修改员工的信息

\*/

**public** **void** updateByPrimaryKey(Emp emp) {

MyQuery myQuery = **new** MyQuery();

myQuery.updateByPrimaryKey(emp);

}

### 7.4.3 编写测试类

/\*\*

\* 测试修改

\*/

@Test

**public** **void** updateByPrimaryKey() **throws** Exception {

Emp emp = **new** Emp(1,"王五","广州",**new** Date());

EmpDao empDao = **new** EmpDao();

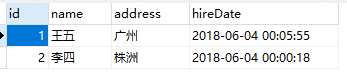
empDao.updateByPrimaryKey(emp);

}

运行结果：

SQL = update Emp set name='王五', address='广州', hireDate='2018-06-04 00:05:55' where id='1'

看看数据库：

修改成功了。激动人心吧。以前怎么没发现JDBC元数据这么好使呢。

## 7.5 测试删除员工

### 7.5.1 编写查询类

/\*\*

\* 根据主键删除记录

\* **@param** object

\*/

**public** **void** deleteByPrimaryKey(Object object) {

PreparedStatement prepareStatement = **null**;

ResultSet resultSet = **null**;

Class<? **extends** Object> clazz = object.getClass();

// 获取简单类名，数据库表名和类名一致

String simpleName = clazz.getSimpleName();

**try** {

// 获取数据库元数据

DatabaseMetaData databaseMetaData = connection.getMetaData();

// 获取给定表主键列的描述

resultSet = databaseMetaData.getPrimaryKeys(**null**, **null**, simpleName);

String primaryKey = **null**;

**if**(resultSet.next()) {

// 获取主键列的名称

primaryKey = resultSet.getString("COLUMN\_NAME");

}

**if**(**null** != primaryKey) {

// 获取主键在实体类中的同名字段

Field declaredField = clazz.getDeclaredField(primaryKey);

// 设置私有字段允许访问

declaredField.setAccessible(**true**);

// 获取字段的值

Object primaryKeyValue = declaredField.get(object);

// 定义SQL语句

String sql = "delete from " + simpleName + " where " + primaryKey + "='" + primaryKeyValue + "'";

System.***out***.println("SQL = " + sql);

// 开启事务

connection.setAutoCommit(**false**);

prepareStatement = connection.prepareStatement(sql);

prepareStatement.executeUpdate();

connection.commit();

} **else** {

System.***err***.println("主键为空！");

}

} **catch** (SQLException e) {

**try** {

// 事务回滚

connection.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **catch** (NoSuchFieldException e) {

e.printStackTrace();

} **catch** (SecurityException e) {

e.printStackTrace();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

} **finally** {

MyConnection.*close*(resultSet, prepareStatement, connection);

}

}

### 7.5.2 编写Dao类

/\*\*

\* 根据主键删除员工的信息

\*/

**public** **void** deleteByPrimaryKey(Emp emp) {

MyQuery myQuery = **new** MyQuery();

myQuery.deleteByPrimaryKey(emp);

}

### 7.5.3 编写测试类

/\*\*

\* 测试删除

\*/

@Test

**public** **void** deleteByPrimaryKey() **throws** Exception {

Emp emp = **new** Emp(1,"李四","广州",**new** Date());

EmpDao empDao = **new** EmpDao();

empDao.deleteByPrimaryKey(emp);

}

运行结果：

SQL = delete from Emp where id='1'

再看看数据库：

因为是根据主键删除的，所以测试类的员工的名字是李四就忽略了。

# 总结

如果你仔细看了查询类的代码或者你自己有练习过，你就会发现那些框架是怎样的思路了。XML，注解，映射就都能拨云见日了。以前在Dao类作死地封装数据，现在就轻松多了。

注意事项：

我写的都是基本的增删改查，没有涉及到多表查询的。

目前只支持MySQL

只支持表名和类名一样

只支持表中字段和类的属性名一样

不支持联合主键

未提供连接池

关于JDBC元数据和反射的知识点我就不多说了，可能看不懂的也看不到底下了，如果有疑问可以给我留言。