JDBC基础

JDBC([**Java**](http://lib.csdn.net/base/17) Database Connectivity)是一个独立于特定[**数据库**](http://lib.csdn.net/base/14)管理系统、通用的SQL数据库存取和操作的公共接口（一组API），定义了用来访问数据库的标准Java类库，使用这个类库可以以一种标准的方法、方便地访问数据库资源，JDBC为访问不同的数据库提供了一种统一的途径，为开发者屏蔽了一些细节问题。JDBC的目标是使Java程序员使用JDBC可以连接任何提供了JDBC驱动程序的数据库系统，这样就使得程序员无需对特定的数据库系统的特点有过多的了解，从而大大简化和加快了开发过程。

Driver 接口

Java.sql.Driver 接口是所有 JDBC 驱动程序需要实现的接口。这个接口是提供给数据库厂商使用的，不同数据库厂商提供不同的实现。在程序中不需要直接去访问实现了 Driver 接口的类，而是由驱动程序管理器类(java.sql.DriverManager)去调用这些Driver实现。

加载与注册 JDBC 驱动

加载 JDBC 驱动需调用 Class 类的静态方法 forName()，向其传递要加载的 JDBC 驱动的类名。DriverManager 类是驱动程序管理器类，负责管理驱动程序，通常不用显式调用 DriverManager 类的 registerDriver() 方法来注册驱动程序类的实例，因为 Driver 接口的驱动程序类都包含了静态代码块，在这个静态代码块中，会调用 DriverManager.registerDriver() 方法来注册自身的一个实例。

建立连接

可以调用 DriverManager 类的 getConnection() 方法建立到数据库的连接。JDBC URL 用于标识一个被注册的驱动程序，驱动程序管理器通过这个 URL 选择正确的驱动程序，从而建立到数据库的连接。   
JDBC URL的标准由三部分组成，各部分间用冒号分隔。   
jdbc:<子协议>:<子名称>   
协议：JDBC URL中的协议总是jdbc   
子协议：子协议用于标识一个数据库驱动程序   
子名称：一种标识数据库的方法。子名称可以依不同的子协议而变化，用子名称的目的是为了定位数据库提供足够的信息   
   
测试代码：

/\*\*

\* Driver 是一个接口: 数据库厂商必须提供实现的接口. 能从其中获取数据库连接.

\* 可以通过 Driver 的实现类对象获取数据库连接.

\*

\* 1. 加入 mysql 驱动

\* 1). 解压 mysql-connector-java-5.1.7.zip

\* 2). 在当前项目下新建 lib 目录

\* 3). 把 mysql-connector-java-5.1.7-bin.jar 复制到 lib 目录下

\* 4). 右键 build-path , add to buildpath 加入到类路径下.s

\* @throws SQLException

\*/

@Test

public void testDriver() throws SQLException {

//1. 创建一个 Driver 实现类的对象

Driver driver = new com.mysql.jdbc.Driver();

//2. 准备连接数据库的基本信息: url, user, password

String url = "jdbc:mysql://localhost:3306/test";

Properties info = new Properties();

info.put("user", "root");

info.put("password", "1230");

//3. 调用 Driver 接口的　connect(url, info) 获取数据库连接

Connection connection = driver.connect(url, info);

System.out.println(connection);

}

/\*\*

\* 编写一个通用的方法, 在不修改源程序的情况下, 可以获取任何数据库的连接

\* 解决方案: 把数据库驱动 Driver 实现类的全类名、url、user、password 放入一个

\* 配置文件中, 通过修改配置文件的方式实现和具体的数据库解耦.

\* @throws Exception

\*/

public Connection getConnection() throws Exception{

String driverClass = null;

String jdbcUrl = null;

String user = null;

String password = null;

//读取类路径下的 jdbc.properties 文件

InputStream in =

getClass().getClassLoader().getResourceAsStream("jdbc.properties");

Properties properties = new Properties();

properties.load(in);

driverClass = properties.getProperty("driver");

jdbcUrl = properties.getProperty("jdbcUrl");

user = properties.getProperty("user");

password = properties.getProperty("password");

//通过反射常见 Driver 对象.

Driver driver =

(Driver) Class.forName(driverClass).newInstance();

Properties info = new Properties();

info.put("user", user);

info.put("password", password);

//通过 Driver 的 connect 方法获取数据库连接.

Connection connection = driver.connect(jdbcUrl, info);

return connection;

}

/\*\*

\* DriverManager 是驱动的管理类.

\* 1). 可以通过重载的 getConnection() 方法获取数据库连接. 较为方便

\* 2). 可以同时管理多个驱动程序: 若注册了多个数据库连接, 则调用 getConnection()

\* 方法时传入的参数不同, 即返回不同的数据库连接。

\* @throws Exception

\*/

@Test

public void testDriverManager() throws Exception{

//1. 准备连接数据库的 4 个字符串.

//驱动的全类名.

String driverClass = "com.mysql.jdbc.Driver";

//JDBC URL

String jdbcUrl = "jdbc:mysql:///test";

//user

String user = "root";

//password

String password = "1230";

//2. 加载数据库驱动程序(对应的 Driver 实现类中有注册驱动的静态代码块.)

Class.forName(driverClass);

//3. 通过 DriverManager 的 getConnection() 方法获取数据库连接.

Connection connection =

DriverManager.getConnection(jdbcUrl, user, password);

System.out.println(connection);

}

public Connection getConnection2() throws Exception{

//1. 准备连接数据库的 4 个字符串.

//1). 创建 Properties 对象

Properties properties = new Properties();

//2). 获取 jdbc.properties 对应的输入流

InputStream in =

this.getClass().getClassLoader().getResourceAsStream("jdbc.properties");

//3). 加载 2） 对应的输入流

properties.load(in);

//4). 具体决定 user, password 等4 个字符串.

String user = properties.getProperty("user");

String password = properties.getProperty("password");

String jdbcUrl = properties.getProperty("url");

String driver = properties.getProperty("driverClass ");

//2. 加载数据库驱动程序(对应的 Driver 实现类中有注册驱动的静态代码块.)

Class.forName(driver);

//3. 通过 DriverManager 的 getConnection() 方法获取数据库连接.

return DriverManager.getConnection(jdbcUrl, user, password);

}

jdbc.properties里面的数据

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/adb

user=root

password=123

访问数据库

数据库连接被用于向数据库服务器发送命令和 SQL 语句，在连接建立后，需要对数据库进行访问，执行 sql 语句。在 java.sql 包中有 3 个接口分别定义了对数据库的调用的不同方式：

1. Statement
2. PrepatedStatement
3. CallableStatement

Statment

通过调用 Connection 对象的 createStatement 方法创建该对象   
该对象用于执行静态的 SQL 语句，并且返回执行结果。Statement 接口中定义了下列方法用于执行 SQL 语句：   
ResultSet excuteQuery(String sql)   
int excuteUpdate(String sql)

ResultSet

通过调用 Statement 对象的 excuteQuery() 方法创建该对象。ResultSet 对象以逻辑表格的形式封装了执行数据库操作的结果集，ResultSet 接口由数据库厂商实现。ResultSet 对象维护了一个指向当前数据行的游标，初始的时候，游标在第一行之前，可以通过 ResultSet 对象的 next() 方法移动到下一行。   
ResultSet 接口的常用方法：   
boolean next()   
getString()

/\*\*

\* 通过 JDBC 向指定的数据表中插入一条记录.

\*

\* 1. Statement: 用于执行 SQL 语句的对象

\* 1). 通过 Connection 的 createStatement() 方法来获取

\* 2). 通过 executeUpdate(sql) 可以执行 SQL 语句.

\* 3). 传入的 SQL 可以是 INSRET, UPDATE 或 DELETE. 但不能是 SELECT

\*

\* 2. Connection、Statement 都是应用程序和数据库服务器的连接资源. 使用后一定要关闭.

\* 需要在 finally 中关闭 Connection 和 Statement 对象.

\*

\* 3. 关闭的顺序是: 先关闭后获取的. 即先关闭 Statement 后关闭 Connection

\*/

@Test

public void testStatement() throws Exception{

//1. 获取数据库连接

Connection conn = null;

Statement statement = null;

try {

conn = getConnection2();

//3. 准备插入的 SQL 语句

String sql = null;

// sql = "INSERT INTO customers (NAME, EMAIL, BIRTH) " +

// "VALUES('XYZ', 'xyz@atguigu.com', '1990-12-12')";

// sql = "DELETE FROM customers WHERE id = 1";

sql = "UPDATE customers SET name = 'TOM' " +

"WHERE id = 4";

System.out.println(sql);

//4. 执行插入.

//1). 获取操作 SQL 语句的 Statement 对象:

//调用 Connection 的 createStatement() 方法来获取

statement = conn.createStatement();

//2). 调用 Statement 对象的 executeUpdate(sql) 执行 SQL 语句进行插入

statement.executeUpdate(sql);

} catch (Exception e) {

e.printStackTrace();

} finally{

if (statement != null) {

try {

statement.close();

} catch (Exception e2) {

e2.printStackTrace();

}

}

if (conn != null) {

try {

conn.close();

} catch (Exception e2) {

e2.printStackTrace();

}

}

}

}

/\*\*

\* ResultSet: 结果集. 封装了使用 JDBC 进行查询的结果.

\* 1. 调用 Statement 对象的 executeQuery(sql) 可以得到结果集.

\* 2. ResultSet 返回的实际上就是一张数据表. 有一个指针指向数据表的第一行的前面.

\* 可以调用 next() 方法检测下一行是否有效. 若有效该方法返回 true, 且指针下移. 相当于

\* Iterator 对象的 hasNext() 和 next() 方法的结合体

\* 3. 当指针对位到一行时, 可以通过调用 getXxx(index) 或 getXxx(columnName)

\* 获取每一列的值. 例如: getInt(1), getString("name")

\* 4. ResultSet 当然也需要进行关闭.

\*/

@Test

public void testResultSet(){

//获取 id=4 的 customers 数据表的记录, 并打印

Connection conn = null;

Statement statement = null;

ResultSet rs = null;

try {

//1. 获取 Connection

conn = JDBCTools.getConnection();

System.out.println(conn);

//2. 获取 Statement

statement = conn.createStatement();

System.out.println(statement);

//3. 准备 SQL

String sql = "SELECT id, name, email, birth " +

"FROM customers";

//4. 执行查询, 得到 ResultSet

rs = statement.executeQuery(sql);

System.out.println(rs);

//5. 处理 ResultSet

while(rs.next()){

int id = rs.getInt(1);

String name = rs.getString("name");

String email = rs.getString(3);

Date birth = rs.getDate(4);

System.out.println(id);

System.out.println(name);

System.out.println(email);

System.out.println(birth);

}

} catch (Exception e) {

e.printStackTrace();

} finally{

//6. 关闭数据库资源.

if(rs != null){

try {

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (statement != null) {

try {

statement.close();

} catch (Exception e2) {

e2.printStackTrace();

}

}

if (conn != null) {

try {

conn.close();

} catch (Exception e2) {

e2.printStackTrace();

}

}

}

}

JDBC API 小结1

1. java.sql.DriverManager用来装载驱动程序，获取数据库连接。
2. java.sql.Connection完成对某一指定数据库的联接
3. java.sql.Statement在一个给定的连接中作为SQL执行声明的容器，他包含了两个重要的子类型。
4. Java.sql.PreparedSatement 用于执行预编译的sql声明
5. Java.sql.CallableStatement用于执行数据库中存储过程的调用
6. java.sql.ResultSet对于给定声明取得结果的途径

PrepatedStatement

可以通过调用 Connection 对象的 preparedStatement(String sql) 方法获取PreparedStatement 对象。PreparedStatement 接口是 Statement 的子接口，它表示一条预编译过的 SQL 语句。PreparedStatement 对象所代表的 SQL 语句中的参数用问号(?)来表示，调用 PreparedStatement 对象的 setXXX(index,value) 方法来设置这些参数。setXXX() 方法有两个参数，第一个参数是要设置的 SQL 语句中的参数的索引(从 1 开始)，第二个是设置的 SQL 语句中的参数的值。

public void testPreparedStatement() {

Connection connection = null;

PreparedStatement preparedStatement = null;

try {

connection = JDBCTools.getConnection();

String sql = "INSERT INTO customers (name, email, birth) "

+ "VALUES(?,?,?)";

preparedStatement = connection.prepareStatement(sql);

preparedStatement.setString(1, "ATGUIGU");

preparedStatement.setString(2, "simpleit@163.com");

preparedStatement.setDate(3,

new Date(new java.util.Date().getTime()));

preparedStatement.executeUpdate();

} catch (Exception e) {

e.printStackTrace();

} finally {

if (statement != null) {

try {

statement.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (connection != null) {

try {

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

public void addNewStudent2(Student student) {

String sql = "INSERT INTO examstudent(flowid, type, idcard, "

+ "examcard, studentname, location, grade) "

+ "VALUES(?,?,?,?,?,?,?)";

JDBCTools.update(sql, student.getFlowId(), student.getType(),

student.getIdCard(), student.getExamCard(),

student.getStudentName(), student.getLocation(),

student.getGrade());

}

/\*\*

\* 执行 SQL 语句, 使用 PreparedStatement

\* @param sql

\* @param args: 填写 SQL 占位符的可变参数

\*/

public static void update(String sql, Object ... args){

Connection connection = null;

PreparedStatement preparedStatement = null;

try {

connection = JDBCTools.getConnection();

preparedStatement = connection.prepareStatement(sql);

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

preparedStatement.setObject(i + 1, args[i]);

}

preparedStatement.executeUpdate();

} catch (Exception e) {

e.printStackTrace();

} finally{

JDBCTools.releaseDB(null, preparedStatement, connection);

}

}

**PreparedStatement vs Statement**

PreparedStatement 能最大可能提高性能：

* DBServer会对预编译语句提供性能优化。因为预编译语句有可能被重复调用，所以语句在被DBServer的编译器编译后的执行代码被缓存下来，那么下次调用时只要是相同的预编译语句就不需要编译，只要将参数直接传入编译过的语句执行代码中就会得到执行。
* 在statement语句中,即使是相同操作但因为数据内容不一样,所以整个语句本身不能匹配,没有缓存语句的意义.事实是没有数据库会对普通语句编译后的执行代码缓存.这样每执行一次都要对传入的语句编译一次.
* (语法检查，语义检查，翻译成二进制命令，缓存)

PreparedStatement 可以防止 SQL 注入

SQL注入

SQL 注入是利用某些系统没有对用户输入的数据进行充分的检查，而在用户输入数据中注入非法的 SQL 语句段或命令，从而利用系统的 SQL 引擎完成恶意行为的做法。对于 Java 而言，要防范 SQL 注入，只要用 PreparedStatement 取代 Statement 就可以了

/\*\*

\* 使用 PreparedStatement 将有效的解决 SQL 注入问题.

\*/

@Test

public void testSQLInjection2() {

String username = "a' OR PASSWORD = ";

String password = " OR '1'='1";

String sql = "SELECT \* FROM users WHERE username = ? "

+ "AND password = ?";

Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

try {

connection = JDBCTools.getConnection();

preparedStatement = connection.prepareStatement(sql);

preparedStatement.setString(1, username);//从1开始往后

preparedStatement.setString(2, password);

resultSet = preparedStatement.executeQuery();

if (resultSet.next()) {

System.out.println("登录成功!");

} else {

System.out.println("用户名和密码不匹配或用户名不存在. ");

}

} catch (Exception e) {

e.printStackTrace();

} finally {

JDBCTools.releaseDB(resultSet, preparedStatement, connection);

}

}

/\*\*

\* SQL 注入.

\*/

@Test

public void testSQLInjection() {

String username = "a' OR PASSWORD = ";

String password = " OR '1'='1";

String sql = "SELECT \* FROM users WHERE username = '" + username

+ "' AND " + "password = '" + password + "'";

System.out.println(sql);

Connection connection = null;

Statement statement = null;

ResultSet resultSet = null;

try {

connection = JDBCTools.getConnection();

statement = connection.createStatement();

resultSet = statement.executeQuery(sql);

if (resultSet.next()) {

System.out.println("登录成功!");

} else {

System.out.println("用户名和密码不匹配或用户名不存在. ");

}

} catch (Exception e) {

e.printStackTrace();

} finally {

JDBCTools.releaseDB(resultSet, statement, connection);

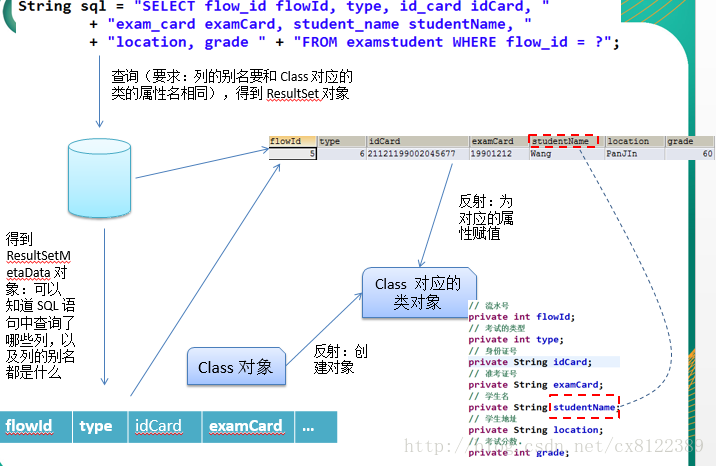
}

}

ResultSetMetaData 类

可用于获取关于 ResultSet 对象中列的类型和属性信息的对象：

* getColumnName(int column)：获取指定列的名称
* getColumnLable(int column)：获取指定列的别名（从1开始）
* getColumnCount()：返回当前 ResultSet 对象中的列数。
* getColumnTypeName(int column)：检索指定列的数据库特定的类型名称。
* getColumnDisplaySize(int column)：指示指定列的最大标准宽度，以字符为单位。
* isNullable(int column)：指示指定列中的值是否可以为 null。
* isAutoIncrement(int column)：指示是否自动为指定列进行编号，这样这些列仍然是只读的。



/\*\*

\* ResultSetMetaData: 描述结果集的元数据.

\* 可以得到结果集中的基本信息: 结果集中有哪些列, 列名, 列的别名等.

\* 结合反射可以写出通用的查询方法.

\*/

@Test

public void testResultSetMetaData(){

Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

try {

connection = JDBCTools.getConnection();

String sql = "SELECT id, name customerName, email, birth " +

"FROM customers";

preparedStatement = connection.prepareStatement(sql);

resultSet = preparedStatement.executeQuery();

//1. 得到 ResultSetMetaData 对象

ResultSetMetaData rsmd = resultSet.getMetaData();

//2. 得到列的个数

int columnCount = rsmd.getColumnCount();

System.out.println(columnCount);

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

//3. 得到列名

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

//4. 得到列的别名

String columnLabel = rsmd.getColumnLabel(i + 1);

System.out.println(columnName + ", " + columnLabel);

}

} catch (Exception e) {

e.printStackTrace();

} finally{

JDBCTools.releaseDB(resultSet, preparedStatement, connection);

}

}

/\*\*

\* 通用的查询方法：可以根据传入的 SQL、Class 对象返回 SQL 对应的记录的对象

\* @param clazz: 描述对象的类型

\* @param sql: SQL 语句。可能带占位符

\* @param args: 填充占位符的可变参数。

\* @return

\*/

public <T> T get(Class<T> clazz, String sql, Object... args) {

T entity = null;

Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

try {

//1. 得到 ResultSet 对象

connection = JDBCTools.getConnection();

preparedStatement = connection.prepareStatement(sql);

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

preparedStatement.setObject(i + 1, args[i]);

}

resultSet = preparedStatement.executeQuery();

//2. 得到 ResultSetMetaData 对象

ResultSetMetaData rsmd = resultSet.getMetaData();

//3. 创建一个 Map<String, Object> 对象, 键: SQL 查询的列的别名,

//值: 列的值

Map<String, Object> values = new HashMap<>();

//4. 处理结果集. 利用 ResultSetMetaData 填充 3 对应的 Map 对象

if(resultSet.next()){

for(int i = 0; i < rsmd.getColumnCount(); i++){

String columnLabel = rsmd.getColumnLabel(i + 1);

Object columnValue = resultSet.getObject(i + 1);

values.put(columnLabel, columnValue);

}

}

//5. 若 Map 不为空集, 利用反射创建 clazz 对应的对象

if(values.size() > 0){

entity = clazz.newInstance();

//6. 遍历 Map 对象, 利用反射为 Class 对象的对应的属性赋值.

for(Map.Entry<String, Object> entry: values.entrySet()){

String fieldName = entry.getKey();

Object value = entry.getValue();

ReflectionUtils.setFieldValue(entity, fieldName, value);

}

}

} catch (Exception e) {

e.printStackTrace();

} finally {

JDBCTools.releaseDB(resultSet, preparedStatement, connection);

}

return entity;

}

import java.lang.reflect.Field;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

import java.lang.reflect.Modifier;

import java.lang.reflect.ParameterizedType;

import java.lang.reflect.Type;

/\*\*

\* 反射的 Utils 函数集合

\* 提供访问私有变量, 获取泛型类型 Class, 提取集合中元素属性等 Utils 函数

\* @author Administrator

\*

\*/

public class ReflectionUtils {

/\*\*

\* 通过反射, 获得定义 Class 时声明的父类的泛型参数的类型

\* 如: public EmployeeDao extends BaseDao<Employee, String>

\* @param clazz

\* @param index

\* @return

\*/

@SuppressWarnings("unchecked")

public static Class getSuperClassGenricType(Class clazz, int index){

Type genType = clazz.getGenericSuperclass();

if(!(genType instanceof ParameterizedType)){

return Object.class;

}

Type [] params = ((ParameterizedType)genType).getActualTypeArguments();

if(index >= params.length || index < 0){

return Object.class;

}

if(!(params[index] instanceof Class)){

return Object.class;

}

return (Class) params[index];

}

/\*\*

\* 通过反射, 获得 Class 定义中声明的父类的泛型参数类型

\* 如: public EmployeeDao extends BaseDao<Employee, String>

\* @param <T>

\* @param clazz

\* @return

\*/

@SuppressWarnings("unchecked")

public static<T> Class<T> getSuperGenericType(Class clazz){

return getSuperClassGenricType(clazz, 0);

}

/\*\*

\* 循环向上转型, 获取对象的 DeclaredMethod

\* @param object

\* @param methodName

\* @param parameterTypes

\* @return

\*/

public static Method getDeclaredMethod(Object object, String methodName, Class<?>[] parameterTypes){

for(Class<?> superClass = object.getClass(); superClass != Object.class; superClass = superClass.getSuperclass()){

try {

//superClass.getMethod(methodName, parameterTypes);

return superClass.getDeclaredMethod(methodName, parameterTypes);

} catch (NoSuchMethodException e) {

//Method 不在当前类定义, 继续向上转型

}

//..

}

return null;

}

/\*\*

\* 使 filed 变为可访问

\* @param field

\*/

public static void makeAccessible(Field field){

if(!Modifier.isPublic(field.getModifiers())){

field.setAccessible(true);

}

}

/\*\*

\* 循环向上转型, 获取对象的 DeclaredField

\* @param object

\* @param filedName

\* @return

\*/

public static Field getDeclaredField(Object object, String filedName){

for(Class<?> superClass = object.getClass(); superClass != Object.class; superClass = superClass.getSuperclass()){

try {

return superClass.getDeclaredField(filedName);

} catch (NoSuchFieldException e) {

//Field 不在当前类定义, 继续向上转型

}

}

return null;

}

/\*\*

\* 直接调用对象方法, 而忽略修饰符(private, protected)

\* @param object

\* @param methodName

\* @param parameterTypes

\* @param parameters

\* @return

\* @throws InvocationTargetException

\* @throws IllegalArgumentException

\*/

public static Object invokeMethod(Object object, String methodName, Class<?> [] parameterTypes,

Object [] parameters) throws InvocationTargetException{

Method method = getDeclaredMethod(object, methodName, parameterTypes);

if(method == null){

throw new IllegalArgumentException("Could not find method [" + methodName + "] on target [" + object + "]");

}

method.setAccessible(true);

try {

return method.invoke(object, parameters);

} catch(IllegalAccessException e) {

System.out.println("不可能抛出的异常");

}

return null;

}

/\*\*

\* 直接设置对象属性值, 忽略 private/protected 修饰符, 也不经过 setter

\* @param object

\* @param fieldName

\* @param value

\*/

public static void setFieldValue(Object object, String fieldName, Object value){

Field field = getDeclaredField(object, fieldName);

if (field == null)

throw new IllegalArgumentException("Could not find field [" + fieldName + "] on target [" + object + "]");

makeAccessible(field);

try {

field.set(object, value);

} catch (IllegalAccessException e) {

System.out.println("不可能抛出的异常");

}

}

/\*\*

\* 直接读取对象的属性值, 忽略 private/protected 修饰符, 也不经过 getter

\* @param object

\* @param fieldName

\* @return

\*/

public static Object getFieldValue(Object object, String fieldName){

Field field = getDeclaredField(object, fieldName);

if (field == null)

throw new IllegalArgumentException("Could not find field [" + fieldName + "] on target [" + object + "]");

makeAccessible(field);

Object result = null;

try {

result = field.get(object);

} catch (IllegalAccessException e) {

System.out.println("不可能抛出的异常");

}

return result;

}

}

public class DAO {

// INSERT, UPDATE, DELETE 操作都可以包含在其中

public void update(String sql, Object... args) {

Connection connection = null;

PreparedStatement preparedStatement = null;

try {

connection = JDBCTools.getConnection();

preparedStatement = connection.prepareStatement(sql);

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

preparedStatement.setObject(i + 1, args[i]);

}

preparedStatement.executeUpdate();

} catch (Exception e) {

e.printStackTrace();

} finally {

JDBCTools.releaseDB(null, preparedStatement, connection);

}

}

// 查询一条记录, 返回对应的对象

public <T> T get(Class<T> clazz, String sql, Object... args) {

List<T> result = getForList(clazz, sql, args);

if(result.size() > 0){

return result.get(0);

}

return null;

}

/\*\*

\* 传入 SQL 语句和 Class 对象, 返回 SQL 语句查询到的记录对应的 Class 类的对象的集合

\* @param clazz: 对象的类型

\* @param sql: SQL 语句

\* @param args: 填充 SQL 语句的占位符的可变参数.

\* @return

\*/

public <T> List<T> getForList(Class<T> clazz,

String sql, Object... args) {

List<T> list = new ArrayList<>();

Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

try {

//1. 得到结果集

connection = JDBCTools.getConnection();

preparedStatement = connection.prepareStatement(sql);

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

preparedStatement.setObject(i + 1, args[i]);

}

resultSet = preparedStatement.executeQuery();

//2. 处理结果集, 得到 Map 的 List, 其中一个 Map 对象

//就是一条记录. Map 的 key 为 reusltSet 中列的别名, Map 的 value

//为列的值.

List<Map<String, Object>> values =

handleResultSetToMapList(resultSet);

//3. 把 Map 的 List 转为 clazz 对应的 List

//其中 Map 的 key 即为 clazz 对应的对象的 propertyName,

//而 Map 的 value 即为 clazz 对应的对象的 propertyValue

list = transfterMapListToBeanList(clazz, values);

} catch (Exception e) {

e.printStackTrace();

} finally {

JDBCTools.releaseDB(resultSet, preparedStatement, connection);

}

return list;

}

public <T> List<T> transfterMapListToBeanList(Class<T> clazz,

List<Map<String, Object>> values) throws InstantiationException,

IllegalAccessException, InvocationTargetException {

List<T> result = new ArrayList<>();

T bean = null;

if (values.size() > 0) {

for (Map<String, Object> m : values) {

bean = clazz.newInstance();

for (Map.Entry<String, Object> entry : m.entrySet()) {

String propertyName = entry.getKey();

Object value = entry.getValue();

BeanUtils.setProperty(bean, propertyName, value);

}

// 13. 把 Object 对象放入到 list 中.

result.add(bean);

}

}

return result;

}

/\*\*

\* 处理结果集, 得到 Map 的一个 List, 其中一个 Map 对象对应一条记录

\*

\* @param resultSet

\* @return

\* @throws SQLException

\*/

public List<Map<String, Object>> handleResultSetToMapList(

ResultSet resultSet) throws SQLException {

// 5. 准备一个 List<Map<String, Object>>:

// 键: 存放列的别名, 值: 存放列的值. 其中一个 Map 对象对应着一条记录

List<Map<String, Object>> values = new ArrayList<>();

List<String> columnLabels = getColumnLabels(resultSet);

Map<String, Object> map = null;

// 7. 处理 ResultSet, 使用 while 循环

while (resultSet.next()) {

map = new HashMap<>();

for (String columnLabel : columnLabels) {

Object value = resultSet.getObject(columnLabel);

map.put(columnLabel, value);

}

// 11. 把一条记录的一个 Map 对象放入 5 准备的 List 中

values.add(map);

}

return values;

}

/\*\*

\* 获取结果集的 ColumnLabel 对应的 List

\*

\* @param rs

\* @return

\* @throws SQLException

\*/

private List<String> getColumnLabels(ResultSet rs) throws SQLException {

List<String> labels = new ArrayList<>();

ResultSetMetaData rsmd = rs.getMetaData();

for (int i = 0; i < rsmd.getColumnCount(); i++) {

labels.add(rsmd.getColumnLabel(i + 1));

}

return labels;

}

// 返回某条记录的某一个字段的值 或 一个统计的值(一共有多少条记录等.)

public <E> E getForValue(String sql, Object... args) {

//1. 得到结果集: 该结果集应该只有一行, 且只有一列

Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

try {

//1. 得到结果集

connection = JDBCTools.getConnection();

preparedStatement = connection.prepareStatement(sql);

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

preparedStatement.setObject(i + 1, args[i]);

}

resultSet = preparedStatement.executeQuery();

if(resultSet.next()){

return (E) resultSet.getObject(1);

}

} catch(Exception ex){

ex.printStackTrace();

} finally{

JDBCTools.releaseDB(resultSet, preparedStatement, connection);

}

//2. 取得结果

return null;

}

}

使用 JDBC 驱动程序处理元数据

Java 通过JDBC获得连接以后，得到一个Connection 对象，可以从这个对象获得有关数据库管理系统的各种信息，包括数据库中的各个表，表中的各个列，数据类型，触发器，存储过程等各方面的信息。根据这些信息，JDBC可以访问一个实现事先并不了解的数据库。获取这些信息的方法都是在DatabaseMetaData类的对象上实现的，而DataBaseMetaData对象是在Connection对象上获得的。 DatabaseMetaData 类中提供了许多方法用于获得数据源的各种信息，通过这些方法可以非常详细的了解数据库的信息：

* getURL()：返回一个String类对象，代表数据库的URL。
* getUserName()：返回连接当前数据库管理系统的用户名。
* isReadOnly()：返回一个boolean值，指示数据库是否只允许读操作。
* getDatabaseProductName()：返回数据库的产品名称。
* getDatabaseProductVersion()：返回数据库的版本号。
* getDriverName()：返回驱动驱动程序的名称。
* getDriverVersion()：返回驱动程序的版本号。

/\*\*

\* DatabaseMetaData 是描述 数据库 的元数据对象.

\* 可以由 Connection 得到.

\* 了解.

\*/

@Test

public void testDatabaseMetaData(){

Connection connection = null;

ResultSet resultSet = null;

try {

connection = JDBCTools.getConnection();

DatabaseMetaData data = connection.getMetaData();

//可以得到数据库本身的一些基本信息

//1. 得到数据库的版本号

int version = data.getDatabaseMajorVersion();

System.out.println(version);

//2. 得到连接到数据库的用户名

String user = data.getUserName();

System.out.println(user);

//3. 得到 MySQL 中有哪些数据库

resultSet = data.getCatalogs();

while(resultSet.next()){

System.out.println(resultSet.getString(1));

}

//...

} catch (Exception e) {

e.printStackTrace();

} finally{

JDBCTools.releaseDB(resultSet, null, connection);

}

}

取得数据库自动生成的主键

/\*\*

\* 取得数据库自动生成的主键

\*/

@Test

public void testGetKeyValue() {

Connection connection = null;

PreparedStatement preparedStatement = null;

try {

connection = JDBCTools.getConnection();

String sql = "INSERT INTO customers(name, email, birth)" +

"VALUES(?,?,?)";

// preparedStatement = connection.prepareStatement(sql);

//使用重载的 prepareStatement(sql, flag)

//来生成 PreparedStatement 对象

preparedStatement = connection.prepareStatement(sql,

Statement.RETURN\_GENERATED\_KEYS);

preparedStatement.setString(1, "ABCDE");

preparedStatement.setString(2, "abcde@atguigu.com");

preparedStatement.setDate(3,

new Date(new java.util.Date().getTime()));

preparedStatement.executeUpdate();

//通过 getGeneratedKeys() 获取包含了新生成的主键的 ResultSet 对象

//在 ResultSet 中只有一列 GENERATED\_KEY, 用于存放新生成的主键值.

ResultSet rs = preparedStatement.getGeneratedKeys();

if(rs.next()){

System.out.println(rs.getObject(1));

}

ResultSetMetaData rsmd = rs.getMetaData();

for(int i = 0; i < rsmd.getColumnCount(); i++){

System.out.println(rsmd.getColumnName(i + 1));

}

} catch (Exception e) {

e.printStackTrace();

} finally{

JDBCTools.releaseDB(null, preparedStatement, connection);

}

}

LOB

LOB，即Large Objects（大对象），是用来存储大量的二进制和文本数据的一种数据类型（一个LOB字段可存储可多达4GB的数据）

/\*\*

\* 读取 blob 数据:

\* 1. 使用 getBlob 方法读取到 Blob 对象

\* 2. 调用 Blob 的 getBinaryStream() 方法得到输入流。再使用 IO 操作即可.

\*/

@Test

public void readBlob(){

Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

try {

connection = JDBCTools.getConnection();

String sql = "SELECT id, name customerName, email, birth, picture "

+ "FROM customers WHERE id = 13";

preparedStatement = connection.prepareStatement(sql);

resultSet = preparedStatement.executeQuery();

if(resultSet.next()){

int id = resultSet.getInt(1);

String name = resultSet.getString(2);

String email = resultSet.getString(3);

System.out.println(id + ", " + name + ", " + email);

Blob picture = resultSet.getBlob(5);

InputStream in = picture.getBinaryStream();

System.out.println(in.available());

OutputStream out = new FileOutputStream("flower.jpg");

byte [] buffer = new byte[1024];

int len = 0;

while((len = in.read(buffer)) != -1){

out.write(buffer, 0, len);

}

in.close();

out.close();

}

} catch (Exception e) {

e.printStackTrace();

} finally{

JDBCTools.releaseDB(resultSet, preparedStatement, connection);

}

}

/\*\*

\* 插入 BLOB 类型的数据必须使用 PreparedStatement：因为 BLOB 类型

\* 的数据时无法使用字符串拼写的。

\*

\* 调用 setBlob(int index, InputStream inputStream)

\*/

@Test

public void testInsertBlob(){

Connection connection = null;

PreparedStatement preparedStatement = null;

try {

connection = JDBCTools.getConnection();

String sql = "INSERT INTO customers(name, email, birth, picture)"

+ "VALUES(?,?,?,?)";

preparedStatement = connection.prepareStatement(sql);

preparedStatement.setString(1, "ABCDE");

preparedStatement.setString(2, "abcde@atguigu.com");

preparedStatement.setDate(3,

new Date(new java.util.Date().getTime()));

InputStream inputStream = new FileInputStream("Hydrangeas.jpg");

preparedStatement.setBlob(4, inputStream);

preparedStatement.executeUpdate();

} catch (Exception e) {

e.printStackTrace();

} finally{

JDBCTools.releaseDB(null, preparedStatement, connection);

}

}

最后附上链接http://blog.csdn.net/cx8122389/article/details/62422249