ColumnListHandler typora-copy-images-to: img

day20-连接池和DBUtils

今日内容

- 连接池
  - 自定义连接池------>难点,不需要掌握---->目的在于**理解数据库连接池的原理以及装饰者设计模式的使用**
  - **使用第三方连接池**----->重点掌握
    - C3P0
    - DRUID
- **DBUtils**----->重点掌握
- 元数据

复习

- 使用PreparedStatement进行CRUD操作步骤:
  - 1.导入MySQL的驱动包
  - 2.拷贝db.properties配置文件到src目录下
  - 3.拷贝JDBCUtils工具类到对应的包中
  - 4.开始进行CRUD操作:
    - 1.注册驱动,获得连接
    - 2.预编译sql语句,得到预编译对象
    - 3.设置参数
    - 4.执行sql语句,处理结果
    - 5.释放资源

```
public class Test1_CRUD {

    @Test
    public void insert() throws Exception{
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();

        // 2.预编译sql语句,得到预编译对象
        String sql = "insert into user values(null,?,?,?)";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setString(1,"ww");
        ps.setString(2,"123456");
        ps.setString(3,"老王");

        // 4.执行sql语句,处理结果
```

```
        int rows = ps.executeUpdate();
        System.out.println("rows:"+rows);

        // 5.释放资源
        JDBCUtils.release(null,ps,connection);

    }

    @Test
    public void update() throws Exception{
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();

        // 2.预编译sql语句,得到预编译对象
        String sql = "update user set password = ? where username = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setString(1,"abcdef");
        ps.setString(2,"ww");


        // 4.执行sql语句,处理结果
        int rows = ps.executeUpdate();
        System.out.println("rows:"+rows);

        // 5.释放资源
        JDBCUtils.release(null,ps,connection);

    }

    @Test
    public void delete() throws Exception{
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();

        // 2.预编译sql语句,得到预编译对象
        String sql = "delete from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setInt(1,4);


        // 4.执行sql语句,处理结果
        int rows = ps.executeUpdate();
        System.out.println("rows:"+rows);

        // 5.释放资源
        JDBCUtils.release(null,ps,connection);

    }

    @Test
    public void selectById() throws Exception{
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();
```

```
        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setInt(1,7);

        // 4.执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()){
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));

        }

        // 5.释放资源
        JDBCUtils.release(resultSet,ps,connection);
        System.out.println(user);

    }

    @Test
    public void selectAll() throws Exception{
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();

        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数

        // 4.执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();

        // 创建List集合
        ArrayList<User> list = new ArrayList<>();

        // 定义User变量
        User user = null;
        while (resultSet.next()){
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
            // 添加到集合中
            list.add(user);
        }
```

```
        // 5.释放资源
        JDBCUtils.release(resultSet,ps,connection);

        for (User user1 : list) {
            System.out.println(user1);
        }

    }

}
```

第一章-自定义连接池

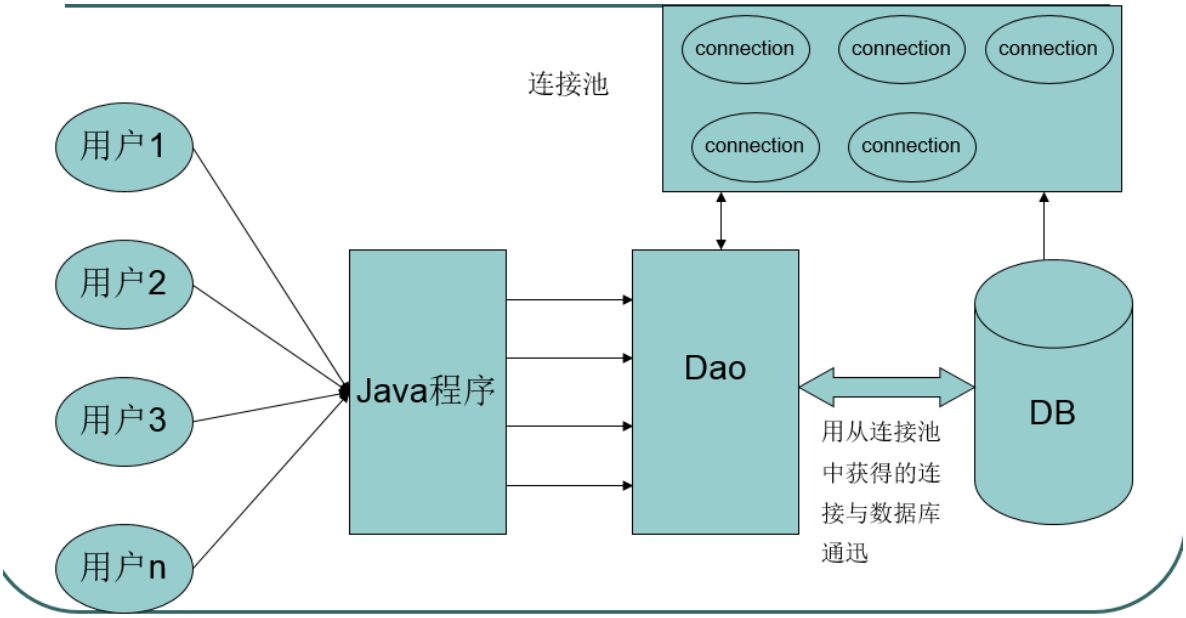
1.1 连接池概念

为什么要使用连接池

Connection对象在JDBC使用的时候就会去创建一个对象,使用结束以后就会将这个对象给销毁了(close).每次创建和销毁对象都是耗时操作.需要使用连接池对其进行优化.

程序初始化的时候，**初始化多个连接,将多个连接放入到池(集合)中**.每次获取的时候,都可以直接**从连接池中进行获取.使用结束以后,将连接归还到池中.**

连接池原理【重点】



1. 程序一开始就创建一定数量的连接，放在一个容器(集合)中，这个容器称为连接池。
2. 使用的时候直接从连接池中取一个已经创建好的连接对象, 使用完成之后 归还到池子
3. 如果池子里面的连接使用完了, 还有程序需要使用连接, 先等待一段时间(eg: 3s), 如果在这段时间之内有连接归还, 就拿去使用; 如果还没有连接归还, 新创建一个, 但是新创建的这一个不会归还了(销毁)
4. 集合选择LinkedList

- 增删比较快
- LinkedList里面的removeFirst()和addLast()方法和连接池的原理吻合

1.2 自定义连接池-初级版本

分析

- 创建连接池类
- 在连接池类中,定义一个LinkedList集合(表示连接池)
- 在连接池类的静态代码块中,创建固定数量的连接,并存储到LinkedList集合中
- 提供一个公共的非静态方法来获取连接对象(getAbc)
- 提供一个公共的非静态方法来归还连接对象(addBack)
- 提供一个公共的静态方法来获取连接池中连接的数量

实现

- 连接池:

```
package com.itheima.demo2_初级版连接池;

import com.itheima.utils.JDBCUtils;

import java.sql.Connection;
import java.util.LinkedList;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 9:37
 */
public class MyDataSource01 {
    //- 在连接池类中,定义一个LinkedList集合(表示连接池)
    private static LinkedList<Connection> pools = new LinkedList<>();

    //- 在连接池类的静态代码块中,创建固定数量的连接,并存储到LinkedList集合中
    static {
        try {
            for (int i = 0; i < 5; i++) {
                // 得到连接对象
                Connection connection = JDBCUtils.getConnection();
                // 添加到连接池中
                pools.add(connection);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //- 提供一个公共的非静态方法来获取连接对象
    public Connection getAbc(){
        Connection connection = pools.removeFirst();
        return connection;
    }

    //- 提供一个公共的非静态方法来归还连接对象
    public void addBack(Connection connection){
```

```
            pools.addLast(connection);
        }

        //- 提供一个公共的静态方法来获取连接池中连接的数量
        public static int size(){
            return pools.size();
        }
    }
```

- 测试:

```
package com.itheima.demo2_初级版连接池;

import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;
import org.junit.Test;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 9:41
 */
public class Test1 {
    public static void main(String[] args) throws Exception {
        // 1.创建连接池对象,获得连接
        MyDataSource01 dataSource = new MyDataSource01();
        System.out.println("获得连接之前,池子中的连接数
量:"+MyDataSource01.size());// 5

        Connection connection = dataSource.getAbc();


        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setInt(1, 7);

        // 4.执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
```

```
        }
        System.out.println("归还连接之前,池子中的连接数
量:"+MyDataSource01.size());// 4

        // 归还连接
        dataSource.addBack(connection);

        // 5.释放资源
        JDBCUtils.release(resultSet, ps, null);
        System.out.println(user);

        System.out.println("归还连接之后,池子中的连接数
量:"+MyDataSource01.size());// 5


    }
}
```

1.3 自定义连接池-进阶版本

分析

在初级版本版本中, 我们定义的方法是**getAbc()**. 因为是自定义的.如果改用李四的自定义的连接池,李四定义的方法是getCon(), 那么我们的源码就需要修改, 这样不方便维护. 所以sun公司定义了一个**接口DataSource**,让自定义连接池有了规范

实现

- 概述: javax.sql.DataSource是Java为数据库连接池提供的公共接口,各个厂商(用户)需要让自己的连接池实现这个接口。这样应用程序可以方便的切换不同厂商的连接池!
- 分析:
  - 创建连接池类, **实现DataSource接口,重写方法**
  - 在连接池类中,定义一个LinkedList集合(表示连接池)
  - 在连接池类的静态代码块中,创建固定数量的连接,并存储到LinkedList集合中
  - **使用重写的方法getConnection,来获取连接对象**
  - 提供一个公共的非静态方法来归还连接对象(addBack)
  - 提供一个公共的静态方法来获取连接池中连接的数量
- 实现:
  - 连接池:

```
package com.itheima.demo3_进阶版连接池;

import com.itheima.utils.JDBCUtils;

import javax.sql.DataSource;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.SQLFeatureNotSupportedException;
import java.util.LinkedList;
```

```java
import java.util.logging.Logger;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 9:37
 */
public class MyDataSource02 implements DataSource {

    //- 在连接池类中,定义一个LinkedList集合(表示连接池)
    private static LinkedList<Connection> pools = new LinkedList<>();

    //- 在连接池类的静态代码块中,创建固定数量的连接,并存储到LinkedList集合中
    static {
        try {
            for (int i = 0; i < 5; i++) {
                // 得到连接对象
                Connection connection = JDBCUtils.getConnection();
                // 添加到连接池中
                pools.add(connection);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //- 提供一个公共的非静态方法来获取连接对象
    /*public Connection getAbc(){
        Connection connection = pools.removeFirst();
        return connection;
    }*/
    @Override
    public Connection getConnection() throws SQLException {
        Connection connection = pools.removeFirst();
        return connection;
    }


    //- 提供一个公共的非静态方法来归还连接对象
    public void addBack(Connection connection){
        pools.addLast(connection);
    }

    //- 提供一个公共的静态方法来获取连接池中连接的数量
    public static int size(){
        return pools.size();
    }




    @Override
    public Connection getConnection(String username, String password)
throws SQLException {
        return null;
    }

    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
```

```
        return null;
    }

    @Override
    public boolean isWrapperFor(Class<?> iface) throws SQLException {
        return false;
    }

    @Override
    public PrintWriter getLogWriter() throws SQLException {
        return null;
    }

    @Override
    public void setLogWriter(PrintWriter out) throws SQLException {

    }

    @Override
    public void setLoginTimeout(int seconds) throws SQLException {

    }

    @Override
    public int getLoginTimeout() throws SQLException {
        return 0;
    }

    @Override
    public Logger getParentLogger() throws
SQLFeatureNotSupportedException {
        return null;
    }
}
```

- 测试:

```
package com.itheima.demo3_进阶版连接池;

import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 9:41
 */
public class Test2 {
    public static void main(String[] args) throws Exception {
        // 1.创建连接池对象,获得连接
        MyDataSource02 dataSource = new MyDataSource02();
        System.out.println("获得连接之前,池子中的连接数量:"+
MyDataSource02.size());// 5
```

```
        Connection connection = dataSource.getConnection();


        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setInt(1, 7);

        // 4.执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));

        }
        System.out.println("归还连接之前,池子中的连接数量:"+
MyDataSource02.size());// 4

        // 归还连接
        dataSource.addBack(connection);

        // 5.释放资源
        JDBCUtils.release(resultSet, ps, null);
        System.out.println(user);

        System.out.println("归还连接之后,池子中的连接数量:"+
MyDataSource02.size());// 5


    }
}
```

1.4 进阶版后存在的问题分析

编写连接池遇到的问题

- 实现DataSource接口后,addBack()又有问题了
    - 在进阶版本中, 我们定义的归还连接的方法是**addBack()**. 因为是自定义的连接池.如果改用李四的自定义的连接池,李四定义的归还连接的方法是back(), 那么我们的源码就需要修改, 这样不方便维护.

    - DataSource接口中也没有定义归还连接的方法,所以只要自定义的连接池,归还连接的方法就可以随便定义,不方便维护.
- 解决办法: 能不能不引入新的api,**直接调用之前的connection.close(),但是这个close不是关闭,而是归还**
    - Connection原有的close方法是关闭连接(销毁连接)
    - **增强close方法,把原有关闭连接的功能变成归还连接的功能,这样连接池中就不需要定义归还连接的方法了**

解决办法

- 继承
    - 条件:可以控制父类, 最起码知道父类的名字
    - 返回的连接对象所属类的类名无法得知,只知道该类是实现了Connection接口
- 装饰者模式
    - 作用：改写已存在的类的某个方法或某些方法
    - 条件:
        - 装饰类和被装饰类要实现同一个接口
        - 装饰类里面要拿到被装饰类的引用
        - 对需要增强的方法进行增强
        - 对不需要增强的方法就调用被装饰类中原有的方法
    - 案例:

```
public interface Star {
    void sing();
    void dance();
}


// 被装饰类
public class LiuDeHua implements Star {
    @Override
    public void sing() {
        System.out.println("刘德华在唱忘情水...");
    }

    @Override
    public void dance() {
        System.out.println("刘德华在跳街舞...");
    }
}

// 装饰类
public class LiuDeHuaWrapper implements Star{

    Star star;

    public LiuDeHuaWrapper(Star star) {
        this.star = star;
    }

    @Override
    public void sing() {
        // 增强
```

```
        System.out.println("刘德华在唱忘情水...");
        System.out.println("刘德华在唱冰雨...");
        System.out.println("刘德华在唱笨小孩...");
    }

    @Override
    public void dance() {
        star.dance();
    }
}


public class Test {
    public static void main(String[] args) {
        LiuDeHua ldh = new LiuDeHua();
        LiuDeHuaWrapper ldhw = new LiuDeHuaWrapper(ldh);
        ldhw.sing();
        ldhw.dance();
    }
}
```

- 动态代理

1.5 自定义连接池-终极版本

分析

- **创建增强的连接类,对close方法进行增强,其余方法依然调用原有的连接对象的方法**
- 创建连接池类, 实现DataSource接口,重写方法
- 在连接池类中,定义一个LinkedList集合(表示连接池)
- 在连接池类的静态代码块中,创建固定数量的连接,并存储到LinkedList集合中
- **使用重写的方法getConnection,来获取连接对象----->增强的连接对象**
- 提供一个公共的静态方法来获取连接池中连接的数量

- **与进阶版的区别:**
  - 连接池类中不需要提供归还连接的方法
  - getConnection获得连接的方法不再返回被增强的连接对象,而是返回增强的连接对象

实现

- 增强的连接类

```
package com.itheima.demo4_终极版连接池;

import java.sql.*;
import java.util.LinkedList;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.Executor;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 10:10
 */
```

```java
// 装饰类
public class ConnectionWrapper implements Connection {// 装饰类和被装饰类需要实现同一个接口
    // 装饰类中需要获取被装饰类的引用
    Connection connection;

    LinkedList<Connection> pools;

    public ConnectionWrapper(Connection connection, LinkedList<Connection> pools) {
        this.connection = connection;
        this.pools = pools;
    }

    // 在装饰类中对需要增强的方法进行增强----close
    @Override
    public void close() throws SQLException {
        // 归还--->连接池,连接对象
        pools.addLast(connection);
    }


    // 在装饰类中对不需要增强的方法就调用被装饰类中同名的方法
    @Override
    public Statement createStatement() throws SQLException {
        return connection.createStatement();
    }

    @Override
    public PreparedStatement prepareStatement(String sql) throws SQLException {
        return connection.prepareStatement(sql);
    }

    @Override
    public CallableStatement prepareCall(String sql) throws SQLException {
        return null;
    }

    @Override
    public String nativeSQL(String sql) throws SQLException {
        return null;
    }

    @Override
    public void setAutoCommit(boolean autoCommit) throws SQLException {

    }

    @Override
    public boolean getAutoCommit() throws SQLException {
        return false;
    }

    @Override
    public void commit() throws SQLException {

    }
```

```
    @Override
    public void rollback() throws SQLException {

    }



    @Override
    public boolean isClosed() throws SQLException {
        return false;
    }

    @Override
    public DatabaseMetaData getMetaData() throws SQLException {
        return null;
    }

    @Override
    public void setReadOnly(boolean readOnly) throws SQLException {

    }

    @Override
    public boolean isReadOnly() throws SQLException {
        return false;
    }

    @Override
    public void setCatalog(String catalog) throws SQLException {

    }

    @Override
    public String getCatalog() throws SQLException {
        return null;
    }

    @Override
    public void setTransactionIsolation(int level) throws SQLException {

    }

    @Override
    public int getTransactionIsolation() throws SQLException {
        return 0;
    }

    @Override
    public SQLWarning getWarnings() throws SQLException {
        return null;
    }

    @Override
    public void clearWarnings() throws SQLException {

    }
```

```
    @Override
    public Statement createStatement(int resultSetType, int
resultSetConcurrency) throws SQLException {
        return null;
    }

    @Override
    public PreparedStatement prepareStatement(String sql, int resultSetType,
int resultSetConcurrency) throws SQLException {
        return null;
    }

    @Override
    public CallableStatement prepareCall(String sql, int resultSetType, int
resultSetConcurrency) throws SQLException {
        return null;
    }

    @Override
    public Map<String, Class<?>> getTypeMap() throws SQLException {
        return null;
    }

    @Override
    public void setTypeMap(Map<String, Class<?>> map) throws SQLException {

    }

    @Override
    public void setHoldability(int holdability) throws SQLException {

    }

    @Override
    public int getHoldability() throws SQLException {
        return 0;
    }

    @Override
    public Savepoint setSavepoint() throws SQLException {
        return null;
    }

    @Override
    public Savepoint setSavepoint(String name) throws SQLException {
        return null;
    }

    @Override
    public void rollback(Savepoint savepoint) throws SQLException {

    }

    @Override
    public void releaseSavepoint(Savepoint savepoint) throws SQLException {

    }
```

```
    @Override
    public Statement createStatement(int resultSetType, int
resultSetConcurrency, int resultSetHoldability) throws SQLException {
        return null;
    }

    @Override
    public PreparedStatement prepareStatement(String sql, int resultSetType,
int resultSetConcurrency, int resultSetHoldability) throws SQLException {
        return null;
    }

    @Override
    public CallableStatement prepareCall(String sql, int resultSetType, int
resultSetConcurrency, int resultSetHoldability) throws SQLException {
        return null;
    }

    @Override
    public PreparedStatement prepareStatement(String sql, int
autoGeneratedKeys) throws SQLException {
        return null;
    }

    @Override
    public PreparedStatement prepareStatement(String sql, int[]
columnIndexes) throws SQLException {
        return null;
    }

    @Override
    public PreparedStatement prepareStatement(String sql, String[]
columnNames) throws SQLException {
        return null;
    }

    @Override
    public Clob createClob() throws SQLException {
        return null;
    }

    @Override
    public Blob createBlob() throws SQLException {
        return null;
    }

    @Override
    public NClob createNClob() throws SQLException {
        return null;
    }

    @Override
    public SQLXML createSQLXML() throws SQLException {
        return null;
    }

    @Override
    public boolean isValid(int timeout) throws SQLException {
```

```
        return false;
    }

    @Override
    public void setClientInfo(String name, String value) throws
SQLClientInfoException {

    }

    @Override
    public void setClientInfo(Properties properties) throws
SQLClientInfoException {

    }

    @Override
    public String getClientInfo(String name) throws SQLException {
        return null;
    }

    @Override
    public Properties getClientInfo() throws SQLException {
        return null;
    }

    @Override
    public Array createArrayOf(String typeName, Object[] elements) throws
SQLException {
        return null;
    }

    @Override
    public Struct createStruct(String typeName, Object[] attributes) throws
SQLException {
        return null;
    }

    @Override
    public void setSchema(String schema) throws SQLException {

    }

    @Override
    public String getSchema() throws SQLException {
        return null;
    }

    @Override
    public void abort(Executor executor) throws SQLException {

    }

    @Override
    public void setNetworkTimeout(Executor executor, int milliseconds)
throws SQLException {

    }
```

```
    @Override
    public int getNetworkTimeout() throws SQLException {
        return 0;
    }

    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
        return null;
    }

    @Override
    public boolean isWrapperFor(Class<?> iface) throws SQLException {
        return false;
    }
}
```

- 连接池

```
package com.itheima.demo4_终极版连接池;

import com.itheima.utils.JDBCUtils;

import javax.sql.DataSource;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.SQLFeatureNotSupportedException;
import java.util.LinkedList;
import java.util.logging.Logger;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 9:37
 */
public class MyDataSource03 implements DataSource {

    //- 在连接池类中,定义一个LinkedList集合(表示连接池)
    private static LinkedList<Connection> pools = new LinkedList<>();

    //- 在连接池类的静态代码块中,创建固定数量的连接,并存储到LinkedList集合中
    static {
        try {
            for (int i = 0; i < 5; i++) {
                // 得到连接对象
                Connection connection = JDBCUtils.getConnection();
                // 添加到连接池中
                pools.add(connection);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //- 提供一个公共的非静态方法来获取连接对象
    /*public Connection getAbc(){
        Connection connection = pools.removeFirst();
```

```
        return connection;
    }*/
    @Override
    public Connection getConnection() throws SQLException {
        // 返回被增强的连接对象
        // Connection connection = pools.removeFirst();
        // return connection;

        // 改为:返回增强的连接对象
        Connection connection = pools.removeFirst();
        // 创建增强的连接对象,传入被增强的连接对象
        ConnectionWrapper connectionWrapper = new
ConnectionWrapper(connection,pools);
        return connectionWrapper;
    }


    //- 提供一个公共的非静态方法来归还连接对象
    /*public void addBack(Connection connection){
        pools.addLast(connection);
    }*/

    //- 提供一个公共的静态方法来获取连接池中连接的数量
    public static int size(){
        return pools.size();
    }




    @Override
    public Connection getConnection(String username, String password) throws
SQLException {
        return null;
    }

    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
        return null;
    }

    @Override
    public boolean isWrapperFor(Class<?> iface) throws SQLException {
        return false;
    }

    @Override
    public PrintWriter getLogWriter() throws SQLException {
        return null;
    }

    @Override
    public void setLogWriter(PrintWriter out) throws SQLException {

    }

    @Override
    public void setLoginTimeout(int seconds) throws SQLException {
```

```
    }

    @Override
    public int getLoginTimeout() throws SQLException {
        return 0;
    }

    @Override
    public Logger getParentLogger() throws SQLFeatureNotSupportedException {
        return null;
    }
}
```

- 测试类

```
package com.itheima.demo4_终极版连接池;

import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 10:10
 */
public class Test3 {
    public static void main(String[] args) throws Exception {
        // 1.创建连接池对象,获得连接
        MyDataSource03 dataSource = new MyDataSource03();
        System.out.println("获得连接之前,池子中的连接数量:"+
MyDataSource03.size());// 5

        // 返回的是增强的连接对象
        Connection connection = dataSource.getConnection();


        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        ps.setInt(1, 7);

        // 4.执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
```

```
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));

        }
        System.out.println("归还连接之前,池子中的连接数量:"+
MyDataSource03.size());// 4

        // 归还连接
        //connection.close();// 增强连接对象的close方法--->功能:把被增强的连接对象
归还到连接池中

        // 5.释放资源
        JDBCUtils.release(resultSet, ps, connection);// connection.close()
        System.out.println(user);

        System.out.println("归还连接之后,池子中的连接数量:"+
MyDataSource03.size());// 5


    }
}
```

第二章-第三方连接池

2.1 C3P0连接池

c3p0介绍

- C3P0**开源免费**的连接池！目前使用它的开源项目有：Spring、Hibernate等。使用第三方工具需要导入jar包，c3p0使用时还需要添加配置文件c3p0-config.xml.
- 使用C3P0需要添加c3p0-0.9.1.2.jar

c3p0的使用

通过硬编码来编写【了解】

- 思路:
    - 创建C3P0连接池对象
    - 设置连接池参数
    - 获得连接
    - 预编译sql语句,得到预编译对象
    - 设置sql语句参数

    - 执行sql语句,处理结果
    - 释放资源
- 实现:

```
package com.itheima.demo5_C3P0的使用;

import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;
import com.mchange.v2.c3p0.ComboPooledDataSource;
import com.mchange.v2.c3p0.jboss.C3P0PooledDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author：pengzhilin
 * @Date: 2021/4/28 10:51
 */
public class Test1_硬编码 {
    public static void main(String[] args) throws Exception {
        //- 创建C3P0连接池对象
        ComboPooledDataSource dataSource = new ComboPooledDataSource();

        //- 设置连接池参数
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/day19_1");
        dataSource.setUser("root");
        dataSource.setPassword("root");
        dataSource.setInitialPoolSize(5);

        //- 获得连接
        Connection connection = dataSource.getConnection();

        // 预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 设置参数
        ps.setInt(1, 7);

        // 执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
        }

        // 释放资源
        JDBCUtils.release(resultSet,ps,connection);
```

```
        System.out.println(user);
    }
}
```

- C3P0连接池的配置参数----- 参考c3p0的官方文档,或者网络上直接搜索

通过配置文件来编写【重点】

- 思路:
  - 拷贝c3p0-config.xml配置文件到src路径下,然后修改配置文件中的参数值
    - **配置文件名一定不能修改**
    - **配置文件一定要放在src路径下**
    - **配置文件中标签的name属性值要与setXXX方法的方法名对应(set方法去掉set,然后首字母变小写)**
  - 创建C3P0连接池对象------>**自动读取src路径下的c3p0-config.xml配置文件**
  - 通过连接池对象获得连接对象
  - 预编译sql语句,得到预编译对象
  - 设置参数
  - 执行sql语句,处理结果
  - 释放资源
- 实现:

```
package com.itheima.demo5_C3P0的使用;

import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;
import com.mchange.v2.c3p0.ComboPooledDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 11:09
 */
public class Test1_配置文件 {
    public static void main(String[] args) throws Exception{
        // 创建C3P0连接池对象
        ComboPooledDataSource dataSource = new ComboPooledDataSource();

        // 获取连接
        Connection connection = dataSource.getConnection();

        // 预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 设置参数
        ps.setInt(1, 7);

        // 执行sql语句,处理结果
```

```
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
        }

        System.out.println("正在使用
的:"+dataSource.getNumBusyConnections());// 正在使用连接数
        System.out.println("正在空闲
的:"+dataSource.getNumIdleConnections());// 空闲连接数
        System.out.println("总的连接数:"+dataSource.getNumConnections());// 总
连接数

        // 释放资源
        JDBCUtils.release(resultSet,ps,connection);
        System.out.println(user);


        Thread.sleep(5000);

        System.out.println("正在使用
的:"+dataSource.getNumBusyConnections());// 正在使用连接数
        System.out.println("正在空闲
的:"+dataSource.getNumIdleConnections());// 空闲连接数
        System.out.println("总的连接数:"+dataSource.getNumConnections());// 总
连接数
    }
}
```

使用c3p0改写工具类【重点】

- 问题: 每次需要连接的时候,都需要创建连接池对象,用完了就销毁,所以就会不断的创建连接池,销毁连接池
- 解决: 整个程序只需要创建一个连接池对象,其余地方直接使用这个唯一的连接池对象获得连接即可
- 工具类:
  - 思路:
    - 创建唯一的连接池对象---->private static final修饰
    - 提供一个获取连接池对象的静态方法
    - 提供一个获取连接的静态方法
    - 提供一个释放资源的静态方法
  - 实现:

```
package com.itheima.utils;

import com.mchange.v2.c3p0.ComboPooledDataSource;
```

```java
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 11:18
 */
public class C3POUtils {
    // 定义为一个私有静态的连接池常量
    private static final ComboPooledDataSource DATA_SOURCE = new
ComboPooledDataSource();

    // 提供一个公共的静态方法获得连接池
    public static DataSource getDataSource(){
        return DATA_SOURCE;
    }

    // 提供一个公共的静态方法获得连接
    public static Connection getConnection() throws SQLException {
        return DATA_SOURCE.getConnection();
    }

    // 提供一个公共的静态方法是否资源
    /**
     * 释放资源
     *
     * @param resultSet
     * @param statement
     * @param connection
     */
    public static void release(ResultSet resultSet, Statement
statement, Connection connection) {
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
```

```
        }
    }

}
```

```
package com.itheima.demo5_C3P0的使用;

import com.itheima.bean.User;
import com.itheima.utils.C3P0Utils;
import com.itheima.utils.JDBCUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 11:21
 */
public class Test3_测试C3P0工具类 {
    public static void main(String[] args) throws Exception {
        // 获得连接
        Connection connection = C3P0Utils.getConnection();

        // 预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 设置参数
        ps.setInt(1, 7);

        // 执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
        }

        // 释放资源
        JDBCUtils.release(resultSet, ps, connection);
        System.out.println(user);
    }
}
```

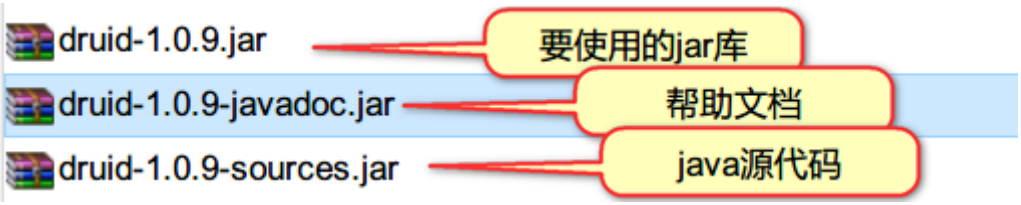
2.2 DRUID连接池

DRUID介绍

Druid是阿里巴巴开发的号称为监控而生的数据库连接池，Druid是国内目前最好的数据库连接池。在功能、性能、扩展性方面，都超过其他数据库连接池。Druid已经在阿里巴巴部署了超过600个应用，经过一年多生产环境大规模部署的严苛考验。如：一年一度的双十一活动，每年春运的抢火车票。

Druid的下载地址：https://github.com/alibaba/druid 或者 maven仓库

DRUID连接池使用的jar包：druid-1.0.9.jar



DRUID的使用

通过硬编码方式【了解】

- 思路:
  - 创建Druid连接池对象
  - 设置连接池的配置参数
  - 通过连接池获得连接
  - 预编译sql语句,得到预编译对象
  - 设置sql语句参数
  - 执行sql语句,处理结果
  - 释放资源
- 实现:

```
package com.itheima.demo6_Druid的使用;

import com.alibaba.druid.pool.DruidDataSource;
import com.alibaba.druid.pool.DruidPooledConnection;
import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 11:37
 */
public class Test1_硬编码 {
    public static void main(String[] args) throws Exception{
        //- 创建Druid连接池对象
        DruidDataSource dataSource = new DruidDataSource();

        //- 设置连接池的配置参数
```

```
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/day19_1");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        dataSource.setInitialSize(5);

        //- 通过连接池获得连接
        DruidPooledConnection connection = dataSource.getConnection();

        // 预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 设置参数
        ps.setInt(1, 7);

        // 执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
        }

        // 释放资源
        JDBCUtils.release(resultSet,ps,connection);
        System.out.println(user);
    }
}
```

通过配置文件方式【重点】

- 思路:
  - 导入Druid的jar包
  - 拷贝druid.properties配置文件到src路径下
    - **配置文件名可以修改**
    - **配置文件建议放在src路径下**
    - **配置文件中键名与setXXX方法的方法名对应(set方法去掉set,然后首字母变小写)**
  - 使用:
    - 创建Properties对象,加载配置文件中的数据
    - 创建Druid连接池对象,传入Properties对象--->不会自动读src路径下的配置文件
    - 通过连接池获得连接
    - 预编译sql语句,得到预编译对象
    - 设置sql语句参数
    - 执行slq语句,处理结果
    - 释放资源

- 实现:

```
package com.itheima.demo6_Druid的使用;

import com.alibaba.druid.pool.DruidDataSource;
import com.alibaba.druid.pool.DruidDataSourceFactory;
import com.itheima.bean.User;
import com.itheima.utils.JDBCUtils;

import javax.sql.DataSource;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Properties;

/**
 * @Author：pengzhilin
 * @Date: 2021/4/28 11:50
 */
public class Test2_配置文件 {
    public static void main(String[] args) throws Exception{
        //- 创建Properties对象,加载配置文件中的数据
        Properties pro = new Properties();
        InputStream is = Test2_配置文
件.class.getClassLoader().getResourceAsStream("druid.properties");
        pro.load(is);

        //- 创建Druid连接池对象,传入Properties对象--->不会自动读src路径下的配置文件
        DataSource dataSource =
DruidDataSourceFactory.createDataSource(pro);

        //- 通过连接池获得连接
        Connection connection = dataSource.getConnection();

        // 预编译sql语句,得到预编译对象
        String sql = "select * from user where id = ?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 设置参数
        ps.setInt(1, 7);

        // 执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();
        // 定义User变量
        User user = null;
        while (resultSet.next()) {
            // 创建User对象
            user = new User();
            // 取值,赋值
            user.setId(resultSet.getInt("id"));
            user.setUsername(resultSet.getString("username"));
            user.setPassword(resultSet.getString("password"));
            user.setNickname(resultSet.getString("nickname"));
        }

        // 释放资源
```

```
            JDBCUtils.release(resultSet,ps,connection);
            System.out.println(user);


        }
    }
```

Druid工具类的制作

- 步骤:
    - 0.定义一个DataSource成员变量
    - 1.在静态代码块中,加载配置文件,创建Druid连接池对象
    - 2.提供一个公共的静态方法获得连接池
    - 3.提供一个公共的静态方法获得连接
    - 4.提供一个公共的静态方法是否资源

```
package com.itheima.utils;

import com.alibaba.druid.pool.DruidDataSourceFactory;
import com.itheima.demo6_Druid的使用.Test2_配置文件;

import javax.sql.ConnectionEvent;
import javax.sql.DataSource;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 11:57
 */
public class DruidUtils {
    // - 0.定义一个DataSource成员变量
    private static DataSource dataSource;

    // - 1.在静态代码块中,加载配置文件,创建Druid连接池对象
    static {
        try {
            //- 创建Properties对象,加载配置文件中的数据
            Properties pro = new Properties();
            InputStream is =
DruidUtils.class.getClassLoader().getResourceAsStream("druid.properties");
            pro.load(is);

            //- 创建Druid连接池对象,传入Properties对象--->不会自动读src路径下的配置文件
            dataSource = DruidDataSourceFactory.createDataSource(pro);

        } catch (Exception e) {

        }
    }
```

```
    // - 2.提供一个公共的静态方法获得连接池
    public static DataSource getDataSource() {
        return dataSource;
    }

    // - 3.提供一个公共的静态方法获得连接
    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }

    // - 4.提供一个公共的静态方法是否资源
    // 提供一个公共的静态方法是否资源

    /**
     * 释放资源
     *
     * @param resultSet
     * @param statement
     * @param connection
     */
    public static void release(ResultSet resultSet, Statement statement,
Connection connection) {
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

第三章-DBUtils

3.1 DBUtils的介绍

DBUtils的概述

DbUtils是Apache组织提供的一个对JDBC进行简单封装的开源工具类库，使用它能够简化JDBC应用程序的开发，同时也不会影响程序的性能

DBUtils的常用API介绍

1. 创建QueryRunner对象的API

   `public QueryRunner(DataSource ds)`,提供数据源（连接池），DBUtils底层自动维护连接connection

2. QueryRunner执行增删改的SQL语句的API

   `int update(String sql, Object... params)`, params参数就是可变参数,参数个数取决于语句中问号的个数

   - eg参数1: `update user set password = ? where username = ?`
   - eg参数2: "123456","zs"

3. 执行查询的SQL语句的API

   `query(String sql, ResultSetHandler<T> rsh, Object... params)`,其中ResultSetHandler是一个接口,表示结果集处理者

   - eg参数1: `select * from user where username = ? and password = ?`
   - eg参数2: 指定查询结果封装的类型
   - eg参数3: zs , 123456

3.2 使用DBUtils完成增删改

- 实现步骤:
  - 导入DButils的jar包---->mysql驱动包,第三方数据库连接池的jar包,配置文件,工具类
  - 创建QueryRunner对象,传入连接池对象
  - 调用update方法执行sql语句
- 增

```
@Test
public void insert() throws Exception{
    // 1.创建QueryRunner对象,传入连接池
    QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

    // 2.调用update方法执行sql语句
    int rows = qr.update("insert into user values(null,?,?,?)", "zl",
"123456", "老赵");
    System.out.println("rows:" + rows);
}
```

- 改

```
@Test
public void update() throws Exception{
    // 1.创建QueryRunner对象,传入连接池
    QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

    // 2.调用update方法执行sql语句
    int rows = qr.update("update user set password = ? where id = ?",
"abcdef",8);
    System.out.println("rows:" + rows);
}
```

- 删

```
@Test
public void delete() throws Exception{
    // 1.创建QueryRunner对象,传入连接池
    QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

    // 2.调用update方法执行sql语句
    int rows = qr.update("delete from user where id = ?", 8);
    System.out.println("rows:" + rows);
}
```

3.3 JavaBean

1. JavaBean说白了就是一个类, 用来封装数据用的
2. **JavaBean要求**
    - 私有字\成员变量
    - 提供公共的get/set方法
    - 无参构造
    - 建议满参构造
    - 实现Serializable
3. 字段(成员变量)和属性
    - **字段: 成员变量** eg: `private String username;`
    - **属性:** set\get方法去掉get或者set首字母变小写 eg: `setUsername()方法-去掉set->Username-->首字母变小写->username`

    > 一般情况下,我们通过IDEA直接生成的set/get 习惯把字段和属性搞成一样而言

4. 注意:
    - **使用DBUtils查询得到的结果可以自动封装成员一个对象,但有一个前提条件就是该对象所属的类的属性名必须和表中的字段名一致,否则封装失败**
    - 也就是说: **类的set\get方法去掉set\get,然后首字母变小写之后得到的名称,必须和表中的列名一致**
    - 学习完DBUtils完成查询操作后,进行测试

```
public class User implements Serializable {
```

```java
    private Integer id;
    private String username;
    private String password;
    private String nickname;

    public User(Integer id, String username, String password, String nickname) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.nickname = nickname;
    }

    public User() {
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", nickname='" + nickname + '\'' +
                '}';
    }
}
```

3.4 使用DBUtils完成查询

ResultSetHandler结果集处理接口的实现类介绍

Handler类型	说明
ArrayHandler	将结果集中的第一条记录封装到一个Object[]数组中，数组中的每一个元素就是这条记录中的每一个字段的值
ArrayListHandler	将结果集中的每一条记录都封装到一个Object[]数组中，将这些数组在封装到List集合中。
BeanHandler	将结果集中第一条记录封装到一个指定的javaBean中。
BeanListHandler	将结果集中每一条记录封装到指定的javaBean中，将这些javaBean在封装到List集合中
ColumnListHandler	将结果集中指定的列的字段值，封装到一个List集合中
KeyedHandler	将结果集中每一条记录封装到Map<String,Object>,在将这个map集合做为另一个Map的value,另一个Map集合的key是指定的字段的值。
MapHandler	将结果集中第一条记录封装到了Map<String,Object>集合中，key就是字段名称，value就是字段值
MapListHandler	将结果集中每一条记录封装到了Map<String,Object>集合中，key就是字段名称，value就是字段值，在将这些Map封装到List集合中。
ScalarHandler	它是用于单个数据。例如select count(*) from 表。

`ArrayHandler`: 适合封装查询结果为一条记录,会把这条记录中每个字段的值封装到`Object[]`数组中
`ArrayListHandler`:适合封装查询结果为多条记录,会把每条记录中的值封装到一个数组中,再把这些数组存储到`List`集合
`BeanHandler`:适合封装查询结果为一条记录的,把这条记录的数据封装到一个指定的对象中
`BeanListHandler`:适合封装查询结果为多条记录,会把每条记录中的值封装到一个对象中,再把这些对象存储到`List`集合
`ColunmListHandler`:适合封装查询结果为单列多行的,会把当前列中所有的值存储到`List`集合
`KeyedHandler`:适合封装查询结果为多条记录的,会把每一条记录封装成一个`Map`集合,再把`Map`集合存储到另一个`Map`集合中
`MapHandler`: 适合封装查询结果为一条记录的,会把这条记录中每个字段的值封装到`Map`集合中
`MapListHandler`:适合封装查询结果为多条记录,会把每条记录中的值封装到一个`Map`集合中,再把这些`Map`集合存储到`List`集合
`ScalarHandler`:适合查询结果为单个值的

代码实现

查询一条记录(使用ArrayHandler)

```
    @Test // 查询id为1的记录,封装到数组中
    public void selectById1() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select * from user where id = ?";
        Object[] arr = qr.query(sql, new ArrayHandler(), 1);
        System.out.println(Arrays.toString(arr));
    }
```

查询一条数据封装到JavaBean对象中(使用BeanHandler)

```
    @Test // 查询id为1的记录,封装到User对象中
    public void selectById2() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select * from user where id = ?";
        User user = qr.query(sql, new BeanHandler<User>(User.class), 1);
        System.out.println(user);
    }
```

查询一条数据,封装到Map对象中(使用MapHandler)

```
@Test // 查询id为1的记录,封装到Map集合
    public void selectById3() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select * from user where id = ?";
        Map<String, Object> map = qr.query(sql, new MapHandler(), 1);
        System.out.println(map);
    }
```

查询多条数据封装到List<Object[]>中(使用ArrayListHandler)

```
    @Test // 查询所有记录,封装到List集合,每条记录封装到数组中
    public void selectAll1() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select * from user";
        List<Object[]> list = qr.query(sql, new ArrayListHandler());
        for (Object[] arr : list) {
            System.out.println(Arrays.toString(arr));
        }
    }
```

查询多条数据封装到List中(使用BeanListHandler)

```
@Test // 查询所有记录,封装到List集合,每条记录封装到javaBean对象中
    public void selectAll2() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select * from user";
        List<User> list = qr.query(sql, new BeanListHandler<User>(User.class));
        for (User user : list) {
            System.out.println(user);
        }
    }
```

查询多条数据,封装到`List<Map>`对象中(使用MapListHandler)

```
@Test // 查询所有记录,封装到List集合,每条记录封装到Map中
    public void selectAll3() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select * from user";
        List<Map<String, Object>> list = qr.query(sql, new MapListHandler());
        for (Map<String, Object> map : list) {
            System.out.println(map);
        }
    }
```

查询单个数据(使用ScalarHandler())

```
    @Test // 查询记录的总条数,封装到一个对象中
    public void selectByColumn() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select count(*) from user";
        Long count = (Long)qr.query(sql, new ScalarHandler());
        System.out.println(count);

    }
```

查询单列多个值(使用ColumnListHandler)

```
@Test // 查询某列的所有值,封装到List集合
    public void selectByColumn() throws Exception{
        // 1.创建QueryRunner对象,传入连接池对象
        QueryRunner qr = new QueryRunner(C3P0Utils.getDataSource());

        // 2.调用query方法
        String sql = "select username from user";
        List<Object> list = qr.query(sql, new ColumnListHandler());
        System.out.println(list);
    }
```

第四章-自定义DBUtils

4.1 元数据

- 概述:元数据(MetaData)，即定义数据的数据。打个比方，就好像我们要想搜索一首歌(歌本身是数据)，而我们可以通过歌名，作者，专辑等信息来搜索，那么这些歌名，作者，专辑等等就是这首歌的元数据。因此数据库的元数据就是一些注明数据库信息的数据。

  简单来说: 数据库的元数据就是 数据库、表、列的定义信息。 分类: 参数元数据,结果集元数据
- 参数:---->数据
  - 参数元数据: 参数的个数,参数的类型
- 结果集--->数据
  - 结果集元数据:
    - 结果集列的个数
    - 结果集列的名字
    - 结果集列的类型
    - 结果集列对应的java类型
    - ....

4.2 参数元数据

- 概述: 参数元素数据就是使用ParameterMetaData类来表示
- 如何获取参数元数据对象:
  - 使用PreparedStatement预编译对象来获取参数的元数据对象
    - `public ParameterMetaData getParameterMetaData()`
- ParameterMetaData相关的API
  - int getParameterCount(); 获得参数个数
  - int getParameterType(int param) 获取指定参数的SQL类型。 (注:MySQL不支持获取参数类型)
- 获取参数元素数据:
  - 步骤:
    - 获取参数的元数据对象
    - 根据参数的元数据对象获取参数的元数据

```
public class Test {
    public static void main(String[] args) throws Exception{
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();

        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user where username = ? and password =
?";
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.获取参数元数据对象--->使用预编译对象获取
        ParameterMetaData pmd = ps.getParameterMetaData();
```

```
        // 3.获取参数的个数-->使用参数元数据对象
        int count = pmd.getParameterCount();
        System.out.println("参数的个数:"+count);// 参数的个数:2

        // 3.获取参数的类型--->mysql不支持获取参数的类型
        //System.out.println(pmd.getParameterTypeName(1));
        //System.out.println(pmd.getParameterClassName(1));

    }
}
```

4.3 结果集元素数据

- 1.概述: 使用ResultSetMetaDat类来表示结果集元数据
  - 获取方式: ResultSetMetaData是由**ResultSet对象通过getMetaData方法获取而来**
  - 作用:ResultSetMetaData可用于获取有关ResultSet对象中列的类型和属性的信息。
- 2.ResultSetMetaData相关的API
  - getColumnCount(); 获取结果集中列项目的个数
  - getColumnName(int column); 获得数据指定列的列名
  - getColumnTypeName();获取指定列的SQL类型
  - getColumnClassName();获取指定列SQL类型对应于Java的类型
- 3.使用步骤:
  - 获得结果集的元数据对象
  - 根据结果集的元数据对象获取结果集中的元素数据(结果集中列的个数,列的名称,列的类型...)

```
public class Test {
    public static void main(String[] args) throws Exception {
        // 1.注册驱动,获得连接
        Connection connection = JDBCUtils.getConnection();

        // 2.预编译sql语句,得到预编译对象
        String sql = "select * from user";
        PreparedStatement ps = connection.prepareStatement(sql);
        // 3.设置参数
        // 4.执行sql语句,处理结果
        ResultSet resultSet = ps.executeQuery();

        // 获得结果集对应的结果集元数据对象
        ResultSetMetaData metaData = resultSet.getMetaData();

        // 使用结果集元数据对象获得结果的元数据(列的个数,类的名称,列的sql类型,列的java
类型....)
        int columnCount = metaData.getColumnCount();
        System.out.println("列的个数:"+columnCount);
        for (int i = 1; i <= columnCount; i++) {
            System.out.println("列的名称:"+metaData.getColumnName(i));
            System.out.println("列的sql类型:"+metaData.getColumnTypeName(i));
            System.out.println("列的java类
型:"+metaData.getColumnClassName(i));
        }
```

```
    }
}
```

4.4 自定义DBUtils增删改

1.需求

- [ ] 模仿DBUtils, 完成增删改的功能

2.分析

- 创建MyQueryRunner类
- 定义一个DataSource成员变量
- 定义一个有参构造方法,空参构造方法
- 定义一个update(String sql,Object... args)完成增删改操作

3.实现

```
/**
 * @Author: pengzhilin
 * @Date: 2021/4/28 16:18
 */
public class MyQueryRunner {
    // 定义一个DataSource连接池成员变量
    DataSource dataSource;

    // 定义一个空参和满参构造方法
    public MyQueryRunner() {
    }

    public MyQueryRunner(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    // 定义一个update方法完成增删改操作
    public int update(String sql, Object... params) throws SQLException {
        // 1.通过连接池获得连接
        Connection connection = dataSource.getConnection();

        // 2.预编译sql语句,得到预编译对象
        PreparedStatement ps = connection.prepareStatement(sql);

        // 3.设置参数
        // 3.1 获取参数的元数据对象
        ParameterMetaData pmd = ps.getParameterMetaData();

        // 3.2 通过参数的元数据对象获取参数的个数
        int count = pmd.getParameterCount();
```

```
        // 3.3 循环遍历,给参数赋值
        for (int i = 0; i < count; i++) {
            ps.setObject(i+1,params[i]);
        }

        // 4.执行sql语句
        int rows = ps.executeUpdate();

        // 5.返回结果
        return rows;
    }
}
```

总结

必须练习:

1.通过配置文件使用C3P0连接池---必须掌握
2.通过配置文件使用DRUID连接池---必须掌握
3.编写C3P0工具类
4.编写DRUID工具类
5.DBUtils的增删查改----必须\重点掌握--开发常用

- 能够理解连接池解决现状问题的原理
  解决的问题:连接可以得到重复利用
  原理:1. 程序一开始就创建一定数量的连接，放在一个容器(集合)中，这个容器称为连接池。
  2. 使用的时候直接从连接池中取一个已经创建好的连接对象， 使用完成之后 归还到池子
  3. 如果池子里面的连接使用完了， 还有程序需要使用连接， 先等待一段时间(eg: 3s)， 如果在这段时间之内有连接归还， 就拿去使用； 如果还没有连接归还， 新创建一个， 但是新创建的这一个不会归还了(销毁)

- 能够使用C3P0连接池
  1.导入jar包---c3p0\驱动包
  2.创建连接池对象 ---->配置文件(配置文件一定要放在src目录下,并且名字必须为c3p0-config.xml)
  3.根据连接池获得连接
  4.创建预编译sql语句对象
  5.设置参数
  6.执行sql语句,处理结果
  7.释放资源

- 能够使用DRUID连接池
  0.导入jar包---druid\驱动包
  1.创建Properties对象,加载配置文件中的数据---->配置文件(配置文件一定要放在src目录下,名字无所谓)
  2.创建连接池对象
  3.根据连接池获得连接
  4.创建预编译sql语句对象
  5.设置参数
  6.执行sql语句,处理结果
  7.释放资源
- 能够编写C3P0连接池工具类
  1.创建静态的连接池常量
  2.提供一个用来获取连接池的静态方法
  3.提供一个用来获取连接的静态方法
  4.提供一个用来获取释放资源的静态方法

- 能够使用DBUtils完成CRUD
  创建QueryRunner对象:public QueryRunner(DataSource datasource);
  增删改: int update(String sql,Object... args);
  查询: 返回值 query(String sql,ResultSetHandler<T> rsh,Object... args)
  ResultSetHandler接口的实现类:
    BeanHandler:适合查询结果是一条记录的,会把这条记录的数据封装到一个javaBean对象中
    BeanListHandler:适合查询结果是多条记录的,会把每条记录的数据封装到一个javaBean对象中,然后把这些javaBean对象添加到List集合中
    ColumnListHandler:适合查询结果是单列多行的,会把该列的所有数据存储到List集合中
    ScalarHandler:适合查询结果是单个值的,会把这个值封装成一个对象

- 能够理解元数据
  定义数据的数据
  ParameterMetaData类:
    概述:表示参数元数据对象,可以用来获取sql语句参数的元数据
    获取参数元数据对象: 使用预编译sql语句对象调用方法获得
      public ParameterMetaData getParameterMetaData ();

    根据参数元数据对象获取参数的元数据:使用ParameterMetaData类的方法
      - int getParameterCount(); 获得参数个数
  ResultSetMetaData类:
    概述:表示结果集的元数据对象,用来获取结果集的元数据
    使用:
      获取结果集元数据对象: 使用ResultSet结果集的方法
        public ResultSetMetaData getMetaData();

      根据结果集元数据对象获取结果集的元数据:使用ResultSetMetaData的方法
        - getColumnCount(); 获取结果集中列项目的个数
        - getColumnName(int column); 获得数据指定列的列名
        - getColumnTypeName();获取指定列的SQL类型
        - getColumnClassName();获取指定列SQL类型对应于Java的类型
- 能够自定义DBUtils
  封装jdbc操作--见案例