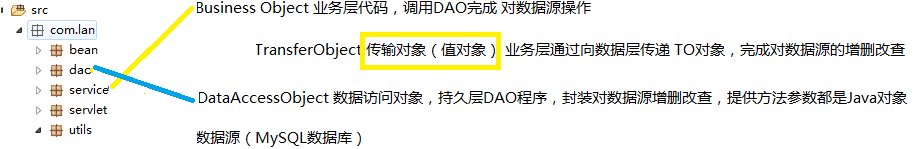
# DAO模式

## 概述

DAO模式（Data Access Object 数据访问对象）：在持久层通过DAO将数据源操作完全封装起来，业务层通过操作Java对象，完成对数据源操作

业务层无需知道数据源底层实现 ，通过java对象操作数据源



DAO模式实际上是两个模式的组合，即Data Accessor模式和Active Domain Object模式，前者实现了数据访问和业务逻辑的分离，后者实现了业务数据的对象化封装。

DAO模式通过对业务层提供数据抽象层接口，实现了以下目标：

- 数据存储逻辑的分离（业务层不会出现SQL语句和JDBC代码）

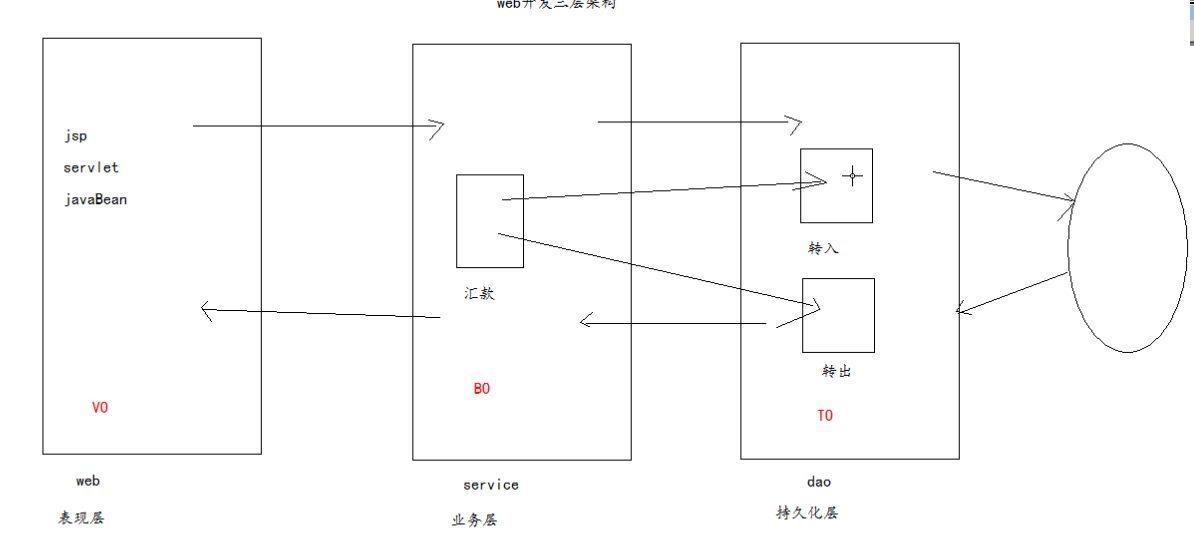
- 数据访问底层实现的分离（底层可以是数据库、文件系统、云存储服务等）

- 资源管理和调度的分离（通过连接池、缓存提升性能而不影响上层系统）

- 数据抽象（操作对象而不是数据字段使得业务逻辑更加清晰）

因此，DAO = Data + Accessor + (Domain) Object

## 图解



# PreparedStatement接口

## SQL注入

由于没有对用户输入进行充分检查，而SQL又是拼接而成，在用户输入参数时，在参数中添加一些SQL 关键字，达到改变SQL运行结果的目的，也可以完成恶意攻击。

String sql = select \* from user where username ='' and password ='' ;

例如：

一、输入 username: 老李' or '1'='1 password 随意

select \* from user where username ='老李' or '1'='1' and password ='';

and 优先级 执行 高于 or

SQL注入原理是什么?

1.在输入时连接一个永远为真的一个值

2.使用mysql 中的 – *注释*

为什么PreparedStatement 可以防止SQL注入 ？

因为它对sql语句进行预编译。

## 解决SQL注入--预编译

使用PreparedStatement 取代 Statement

PreparedStatement 解决SQL注入原理，运行在SQL中参数以?占位符的方式表示

select \* from user where username = ? and password = ? ;

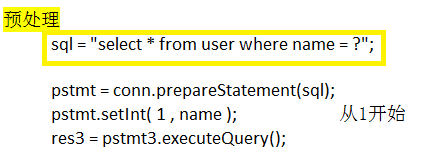
原理：

将带有?的SQL 发送给数据库完成编译 （不能执行的SQL 带有?的SQL 进行编译 叫做），在SQL编译后发现缺少两个参数PreparedStatement 可以将? 代替参数 发送给数据库服务器，因为SQL已经编译过，参数中特殊字符不会当做特殊字符编译，无法达到SQL注入的目的

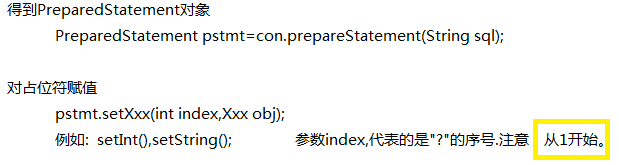
问题：

## PreparedStatement接口

### 定义sql语句



### 创建命令对象



### 命令对象执行sql语句



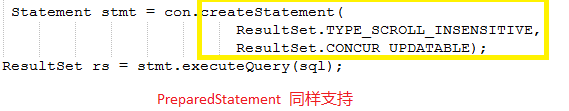
### 批处理操作



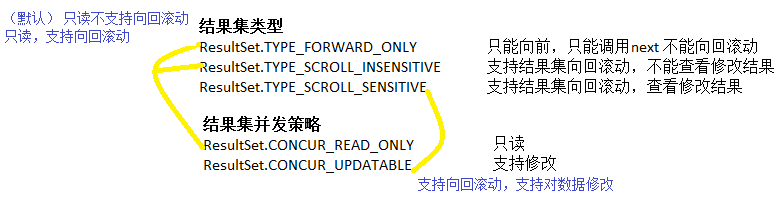
# ResultSet 高级应用 ---- 滚动结果集

## 初始化

设置结果集类型，并发策略



## 三种组合



# JdbcUtils抽取方式1

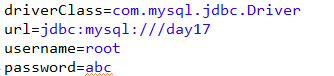
## jdbc.properties

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql:///day17

username=root

password=abc



## JdbcUtils

**public** **class** JdbcUtils {

**private** **static** **final** String *DRIVERCLASS*;

**private** **static** **final** String *URL*;

**private** **static** **final** String *USERNAME*;

**private** **static** **final** String *PASSWORD*;

**static** {

*DRIVERCLASS* = ResourceBundle.*getBundle*("jdbc").getString("driverClass");

*URL* = ResourceBundle.*getBundle*("jdbc").getString("url");

*USERNAME* = ResourceBundle.*getBundle*("jdbc").getString("username");

*PASSWORD* = ResourceBundle.*getBundle*("jdbc").getString("password");

}

**static** {

**try** {

// 将加载驱动操作，放置在静态代码块中.这样就保证了只加载一次.

Class.*forName*(*DRIVERCLASS*);

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

}

}

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

// 2.获取连接

Connection con = DriverManager.*getConnection*(*URL*, *USERNAME*, *PASSWORD*);

**return** con;

}

//关闭操作

**public** **static** **void** closeConnection(Connection con) **throws** SQLException{

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

con.close();

}

}

**public** **static** **void** closeStatement(Statement st) **throws** SQLException{

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

st.close();

}

}

**public** **static** **void** closeResultSet(ResultSet rs) **throws** SQLException{

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

rs.close();

}

}

}

## dao

**public** **class** UserDaoImpl **implements** UserDao{

// 查找用户---使用Statement完成登录操作，存在风险(sql注入)

**public** User \_findUser(User user) **throws** SQLException {

// 1.sql语句

String sql = "select \* from user where username='" + user.getUsername()

+ "' and password='" + user.getPassword() + "'";

// 2.执行sql

Connection con = **null**;

Statement st = **null**;

ResultSet rs = **null**;

**try** {

con = JdbcUtils.*getConnection*();

st = con.createStatement();

rs = st.executeQuery(sql);

**if** (rs.next()) { // 如果可以next,代表查找到了这个用户的信息，就将结果集中的信息封装到User对象中.

User u = **new** User();

u.setId(rs.getInt("id"));

u.setUsername(rs.getString("username"));

u.setPassword(rs.getString("password"));

u.setEmail(rs.getString("email"));

**return** u;

}

} **finally** {

**try** {

JdbcUtils.*closeResultSet*(rs);

JdbcUtils.*closeStatement*(st);

JdbcUtils.*closeConnection*(con);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**return** **null**;

}

// 使用PreparedStatement来完成操作，它可以解决sql注入.

**public** User findUser(User user) **throws** SQLException {

// 1.sql语句

String sql = "select \* from user where username=? and password=?";

// 2.执行sql

Connection con = **null**;

PreparedStatement pst = **null**;

ResultSet rs = **null**;

**try** {

con = JdbcUtils.*getConnection*();

pst = con.prepareStatement(sql);

pst.setString(1, user.getUsername());

pst.setString(2, user.getPassword());

rs = pst.executeQuery();// 无参数

**if** (rs.next()) { // 如果可以next,代表查找到了这个用户的信息，就将结果集中的信息封装到User对象中.

User u = **new** User();

u.setId(rs.getInt("id"));

u.setUsername(rs.getString("username"));

u.setPassword(rs.getString("password"));

u.setEmail(rs.getString("email"));

**return** u;

}

}**finally** {

**try** {

JdbcUtils.*closeResultSet*(rs);

JdbcUtils.*closeStatement*(pst);

JdbcUtils.*closeConnection*(con);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**return** **null**;

}

}

# JdbcUtils抽取方式2

## jdbc.properties

jdbc.driver = com.mysql.jdbc.Driver

jdbc.url = jdbc:mysql://127.0.0.1:3306/user?useUnicode=true&characterEncoding=UTF-8

jdbc.user = root

jdbc.password = root

## JdbcUtil

**public** **class** JdbcUtil {

**private** **static** String *driver*;

**private** **static** String *url*;

**private** **static** String *user*;

**private** **static** String *password*;

**static** {

/\*\*

\* 输入流

\*/

InputStream in = JdbcUtil.**class**.getClassLoader().getResourceAsStream("jdbc.properties");

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

**try** {

//将输入流加载到配置文件的

properties.load(in);

} **catch** (IOException e) {

System.***out***.println("jdbc.properties加载失败");

e.printStackTrace();

}

//读取配置文件参数

*driver* = properties.getProperty("jdbc.driver");

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

*user* = properties.getProperty("jdbc.user");

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

}

**static** {

**try** {

//注册驱动

Class.*forName*(*driver*);

} **catch** (ClassNotFoundException e) {

System.***out***.println("注册驱动失败");

e.printStackTrace();

}

}

/\*\*

\* 创建连接

\* **@return**

\*/

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

**try** {

**return** DriverManager.*getConnection*(*url*, *user*, *password*);

} **catch** (SQLException e) {

System.***out***.println("创建数据库连接失败");

e.printStackTrace();

**return** **null**;

}

}

**public** **static** **void** release(Connection conn,Statement stmt) {

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

**try** {

stmt.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

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

**try** {

conn.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**public** **static** **void** release(Connection conn,PreparedStatement pstmt) {

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

**try** {

pstmt.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

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

**try** {

conn.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**public** **static** **void** release(Connection conn,Statement stmt , ResultSet res) {

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

**try** {

res.close();

} **catch** (SQLException e) {

System.***out***.println("关闭ResultSet失败");

e.printStackTrace();

}

}

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

**try** {

stmt.close();

} **catch** (SQLException e) {

System.***out***.println("关闭Statement失败");

e.printStackTrace();

}

}

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

**try** {

conn.close();

} **catch** (SQLException e) {

System.***out***.println("关闭Connection失败");

e.printStackTrace();

}

}

}

**public** **static** **void** release(Connection conn,PreparedStatement pstmt , ResultSet res) {

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

**try** {

res.close();

} **catch** (SQLException e) {

System.***out***.println("关闭ResultSet失败");

e.printStackTrace();

}

}

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

**try** {

pstmt.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

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

**try** {

conn.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

## Dao

**public** ArrayList<Car> seleteAll() {

Connection conn = JdbcUtil.*getConnection*();

String sql = "select \* from car";

PreparedStatement pstmt;

ResultSet res;

ArrayList<Car> list = **null**;

**try** {

pstmt = conn.prepareStatement(sql);

res = pstmt.executeQuery();

list = **new** ArrayList<>();

**while** (res.next()) {

Car car = **new** Car();

car.setId(res.getInt(1));

car.setName(res.getString(2));

car.setType(res.getString(3));

car.setPrice(res.getDouble(4));

list.add(car);

}

JdbcUtil.*release*(conn, pstmt, res);

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** list;

}

# JDBC进行批处理

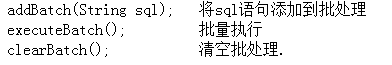
## 概念

业务场景：当需要向数据库发送一批SQL语句执行时，应避免向数据库一条条的发送执行，而应采用JDBC的批处理机制，以提升执行效率。

一次执行多条sql语句

## 第一种方式—Statement

### 方法



### 示例

public class StatementBatchTest {

public static void main(String[] args) throws SQLException {

// 定义sql语句

String sql1 = "create table person(id int,name varchar(20))";

String sql2 = "insert into person values(1,'tom')";

String sql3 = "insert into person values(2,'fox')";

String sql4 = "insert into person values(3,'tony')";

String sql5 = "update person set name='张三' where id=1";

String sql6 = "delete from person where id=3";

Connection con = JdbcUtils.getConnection();

// 得到一个Statement对象

Statement st = con.createStatement();

// 使用批处理执行sql

st.addBatch(sql1);

st.addBatch(sql2);

st.addBatch(sql3);

st.addBatch(sql4);

st.addBatch(sql5);

st.addBatch(sql6);

// 执行批处理

st.executeBatch();

st.close();

con.close();

}

}

## 第二种方式—PreparedStatement

### 方法



### 示例

public class PreparedStatementBatchTest {

public static void main(String[] args) throws SQLException {

// 向person表中插入1000条数据

String sql = "insert into person values(?,?)";

Connection con = JdbcUtils.getConnection();

PreparedStatement pst = con.prepareStatement(sql);

// 批处理

long l=System.currentTimeMillis();

for (int i = 1; i <= 100000; i++) {

pst.setInt(1, i);

pst.setString(2, "name" + i);

pst.addBatch();

if(i%1000==0){

pst.executeBatch();

pst.clearBatch(); //清空缓存。

}

}

//执行批处理

pst.executeBatch();

pst.close();

con.close();

System.out.println(System.currentTimeMillis()-l);

}

}

## 运用场景

