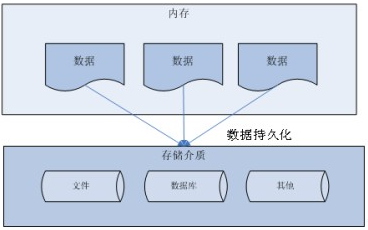
JDBC

# JDBC相关的基础概念

## 1.1JDBC的基础

**一、数据持久化**

* 持久化(persistence)：把数据保存到可掉电式存储设备中以供之后使用。大多数情况下，特别是企业级应用，数据持久化意味着将内存中的数据保存到硬盘上加以”固化”，而持久化的实现过程大多通过各种关系数据库来完成。
* 持久化的主要应用是将内存中的数据存储在关系型数据库中，当然也可以存储在磁盘文件、XML数据文件中。



**二、Java中的数据库存储技术**

* 在Java中，数据库存取技术可分为如下几类：
  + **JDBC直接访问数据库**
  + JDO技术
  + 第三方O/R工具，如Hibernate, Mybatis 等
* JDBC是java访问数据库的基石，JDO, Hibernate等只是更好的封装了JDBC。

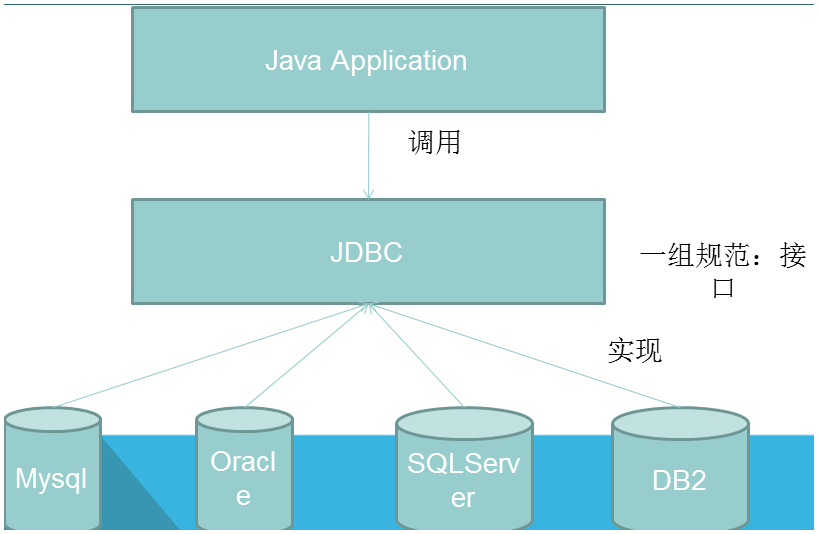
**三、JDBC介绍**

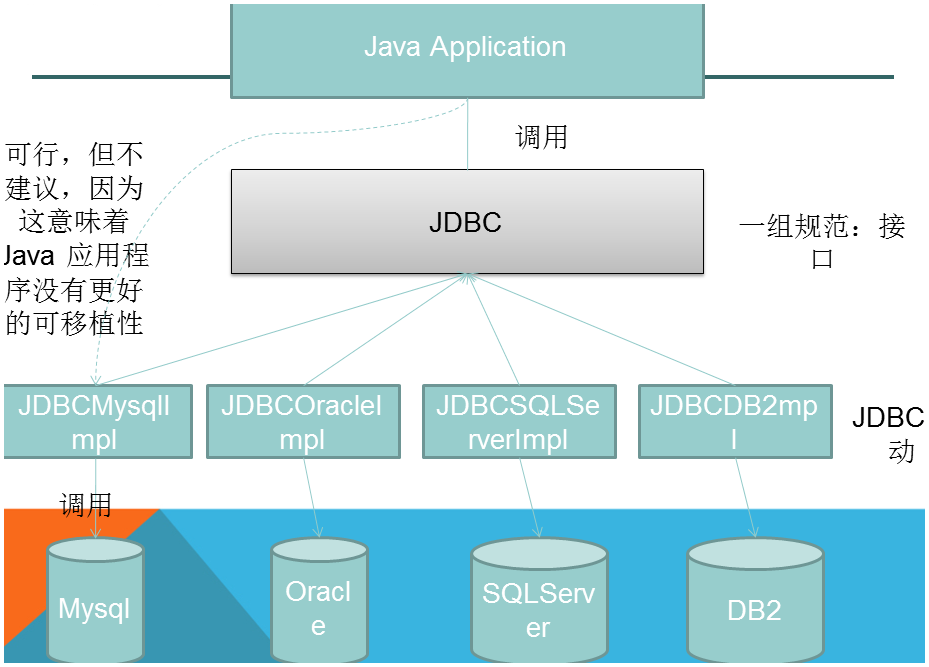
JDBC(Java Database Connectivity)是一个**独立于特定数据库管理系统**、**通用的SQL数据库存取和操作的公共接口**（一组API），定义了用来访问数据库的标准Java类库，使用这个类库可以以一种标准的方法、方便地访问数据库资源

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。

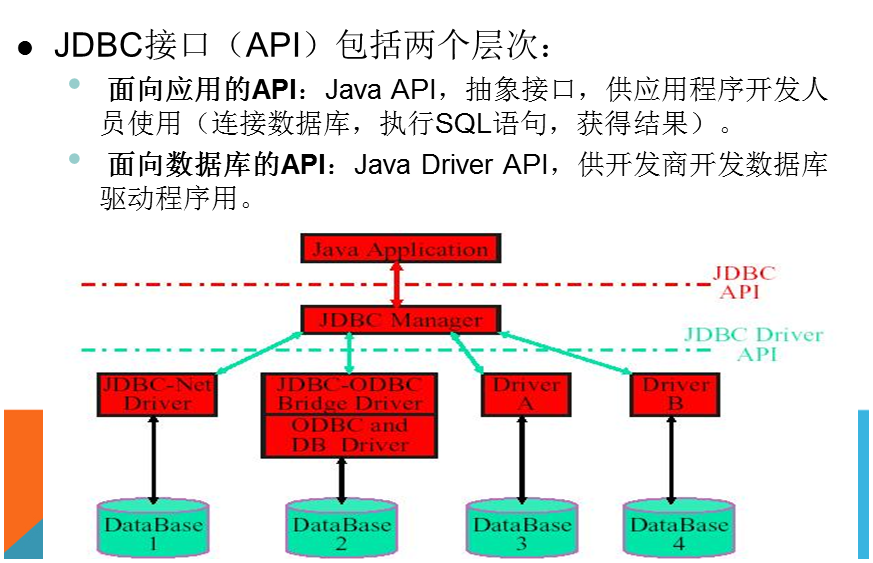
JDBC的目标是使Java程序员使用JDBC可以连接任何**提供了JDBC驱动程序**的数据库系统，这样就使得程序员无需对特定的数据库系统的特点有过多的了解，从而大大简化和加快了开发过程。

## 1.2JDBC的作用

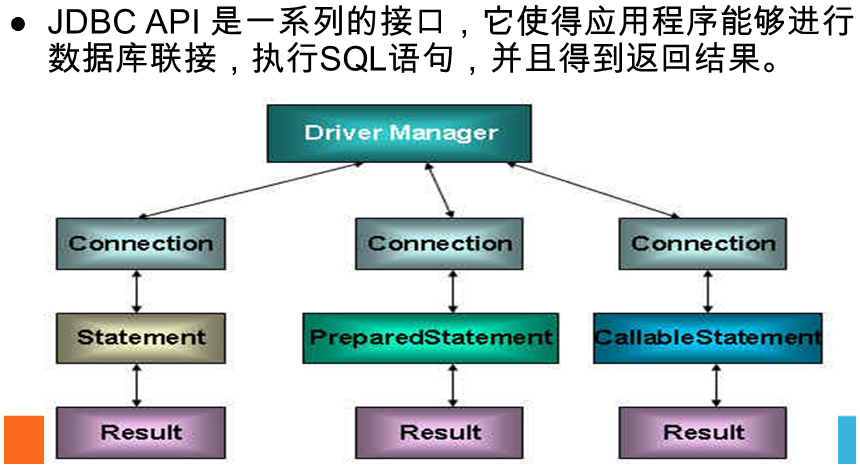




## 1.3JDBC的体系结构



## 1.4JDBC的API规范

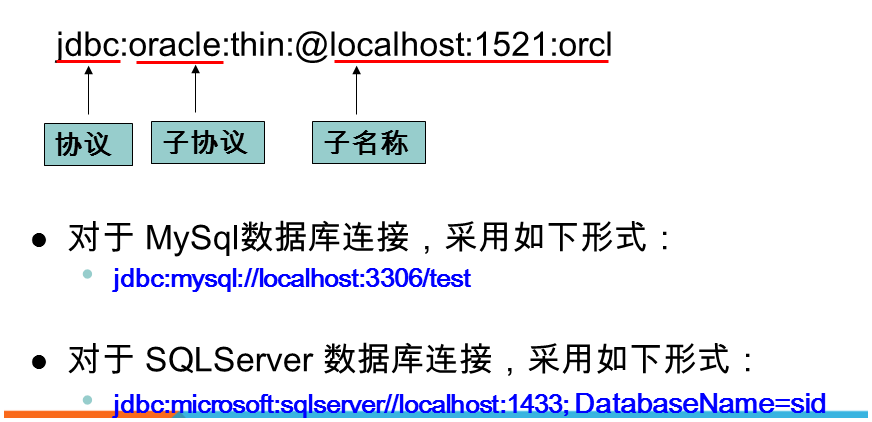


## 1.5API相关的接口

### 1.5.1Driver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

### 1.5.2数据库的JDBC URL



* 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

### 1.5.3加载和注册JDBC驱动

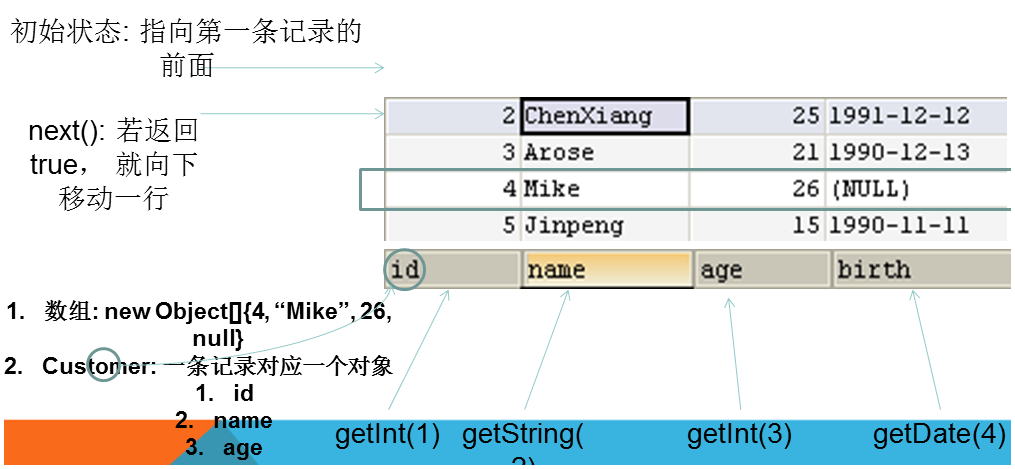
* 加载 JDBC 驱动需调用 Class 类的静态方法 forName()，向其传递要加载的 JDBC 驱动的类名
* DriverManager 类是驱动程序管理器类，负责管理驱动程序
* 通常不用显式调用 DriverManager 类的 registerDriver() 方法来注册驱动程序类的实例，因为 Driver 接口的驱动程序类都包含了静态代码块，在这个静态代码块中，会调用 DriverManager.registerDriver() 方法来注册自身的一个实例

### 1.5.4Statement的作用

* 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)

### 1.5.5ResultSet结果集

* 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()
  + …



## 1.6CRUD

**package** com.guigu.utils;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** java.util.Properties;

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

// 定义连接数据库的四个属性

// 定义用户名

// private static final String username = "guigu";

// 定义密码

// private static final String password = "guigu";

// 定义路径

// private static final String url = "jdbc:oracle:thin:@192.168.118.5:1521:ORCL";

// 定义驱动

// private static final String driver = "oracle.jdbc.driver.OracleDriver";

// 定义用户名

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

// 定义密码

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

// 定义路径

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

// 定义驱动

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

**static** {

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

//通过类加载器 把指定的文件通过流的形式加载进来

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

**try** {

properties.load(in);

} **catch** (IOException e) {

e.printStackTrace();

}

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

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

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

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

}

/\*\*

\* 获取Connection连接

\*/

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

// 建立与数据局之间的连接

Connection conn = **null**;

**try** {

// 加载驱动

Class.*forName*(***driver***);

conn = DriverManager.*getConnection*(***url***, ***username***, ***password***);

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** conn;

}

/\*\*

\* 统一关闭资源

\*

\*/

**public** **static** **void** freeAll(Connection conn,Statement st, ResultSet rs) {

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

**try** {

conn.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

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

**try** {

st.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

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

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

**Jdbc.properties文件**

jdbc.username=guigu

jdbc.password=guigu

jdbc.url=jdbc:oracle:thin:@192.168.118.5:1521:ORCL

jdbc.driver=oracle.jdbc.driver.OracleDriver

**package** com.guigu.crud;

**import** java.sql.Connection;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** com.guigu.utils.JdbcUtils;

**public** **class** CRUD {

**public** **static** **void** main(String[] args) {

// createTable();

// saveData();

// updateData();

*queryData*();

// deleteData();

}

/\*\*

\*5.删除数据

\*/

**public** **static** **void** deleteData() {

String sql = "delete from student where sid=1";

Connection conn = **null**;

// 3.构建Statment对象

Statement st = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

// 关闭打开的资源

JdbcUtils.*freeAll*(conn, st, **null**);

}

}

/\*\*

\* 4.普通的查询查询

\* 1)调用Statment对象的executeQuery(sql)方法 查询数据 可以得到结果集ResultSet

\* 2)ResultSet结果集本身就是一张数据表 ，其中有一个指针指向数据表的第一行 之前 。

\* 3)可以通过next()方法 来检测是否有下一行数据 和取出下一行数据 这里的next和Iterator中的hasNext（）和next的结合体

\* 4)通过getXxxx获取数据 getInt(1) getString(2)..

\* 5)关闭资源

\*

\*/

**public** **static** **void** queryData() {

String sql = "select \* from student";

Connection conn = **null**;

// 3.构建Statment对象

Statement st = **null**;

//定义结果集ResultSet

ResultSet rs =**null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

rs=st.executeQuery(sql);

//通过ResultSet这个结果集 取数据

**while**(rs.next()) {

// System.out.println(rs.getInt("sid"));

// System.out.println(rs.getString("sname"));

// System.out.println(rs.getInt("sage"));

//如果不写列名 必须和数据表的顺序一致

// System.out.println(rs.getInt(1));

// System.out.println(rs.getString(2));

// System.out.println(rs.getInt(3));

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

System.***out***.println(rs.getObject(2));

System.***out***.println(rs.getObject(3));

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

}

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

// 关闭打开的资源

JdbcUtils.*freeAll*(conn, st, rs);

}

}

/\*\*

\* 3.普通的更新数据

\*/

**public** **static** **void** updateData() {

String sql = "update student set sname='归谷' where sid=1";

Connection conn = **null**;

// 3.构建Statment对象

Statement st = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

// 关闭打开的资源

JdbcUtils.*freeAll*(conn, st, **null**);

}

}

/\*\*

\* 2.普通数据的保存操作

\*

\*/

**public** **static** **void** saveData() {

String sql = "insert into student values(3,'王五',35)";

Connection conn = **null**;

// 3.构建Statment对象

Statement st = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, st, **null**);

}

}

/\*\*

\* 1.创建表 通过JDBC如何向数据库中发送数据

\*

\* 1)创建Statement 可以通过Connection对象的createStatment创建， 这个类的主要作用是用于执行sql语句的对象

\* 2)通过excuteQuery完成查询 通过excuteUpdate完成更新操作

\* 3)关闭资源

\*/

**public** **static** **void** createTable() {

// 1.保证这条sql是能正确执行的

String sql = "create table student(sid number(10) primary key, sname varchar2(30) ,sage number(4))";

// 2.获取与数据库的连接

Connection conn = **null**;

// 3.构建Statment对象

Statement st = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

// 4.执行sql语句

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

// 关闭打开的资源

**try** {

st.close();

conn.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

使用绑定参数的形式完成数据的更新

**package** com.guigu.crud;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.guigu.model.Student;

**import** com.guigu.utils.JdbcUtils;

**public** **class** CRUD2 {

**public** **static** **void** main(String[] args) {

Student stu = **new** Student(7, "杨戬", 33);

// saveData(stu);

// updateData(stu);

// Student stu2 = queryData(7);

// System.out.println(stu2);

// System.out.println("---------");

// List<Student> stus = queryData();

// System.out.println(stus);

*deleteStu*(7);

}

// 绑定参数删除

**public** **static** **void** deleteStu(**int** id) {

String sql = "delete from student where sid =?";

Connection conn = **null**;

PreparedStatement ps = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

// 预编译sql

ps = conn.prepareStatement(sql);

ps.setInt(1, id);

// 这里不需要传递sql 否则报错

ps.execute();

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, **null**);

}

}

// 绑定参数查询

// 查询所有学生信息

**public** **static** List<Student> queryData() {

String sql = "select \* from student";

Connection conn = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

Student stu = **null**;

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

**try** {

conn = JdbcUtils.*getConnection*();

// 预编译sql

ps = conn.prepareStatement(sql);

// 这里不需要传递sql 否则报错

rs = ps.executeQuery();

**while** (rs.next()) {

stu = **new** Student(rs.getInt("sid"), rs.getString("sname"), rs.getInt("sage"));

stuList.add(stu);

}

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, rs);

}

**return** stuList;

}

// 根据id查询学生信息

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

String sql = "select \* from student where sid =?";

Connection conn = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

Student stu = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

// 预编译sql

ps = conn.prepareStatement(sql);

ps.setInt(1, id);

// 这里不需要传递sql 否则报错

rs = ps.executeQuery();

**while** (rs.next()) {

stu = **new** Student(rs.getInt("sid"), rs.getString("sname"), rs.getInt("sage"));

}

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, rs);

}

**return** stu;

}

// 绑定参数更新

**public** **static** **void** updateData(Student stu) {

String sql = "update student set sname=? , sage =? where sid =?";

Connection conn = **null**;

PreparedStatement ps = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

// 预编译sql

ps = conn.prepareStatement(sql);

ps.setString(1, stu.getSname());

ps.setInt(2, stu.getSage());

ps.setInt(3, stu.getSid());

// 这里不需要传递sql 否则报错

ps.execute();

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, **null**);

}

}

// 绑定参数添加

**public** **static** **void** saveData(Student stu) {

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

Connection conn = **null**;

PreparedStatement ps = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

// 预编译sql

ps = conn.prepareStatement(sql);

ps.setInt(1, stu.getSid());

ps.setString(2, stu.getSname());

ps.setInt(3, stu.getSage());

// 这里不需要传递sql 否则报错

ps.execute();

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, **null**);

}

}

}

**课堂练习作业**

# 2.JDBC相关的操作

## 2.1SQL注入的问题

* 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

**package** com.guigu.crud;

**import** java.sql.Connection;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** com.guigu.model.User;

**import** com.guigu.utils.JdbcUtils;

**public** **class** TestSqlInjection {

**public** **static** **void** main(String[] args) {

User user = **new** User();

user.setUsername("a' or password=");

user.setPassword("or '1'='1");//

**if**(*checkUser*(user)) {

System.***out***.println("用户名和名正确,登录成功....");

}**else** {

System.***out***.println("用户名和密码不正确,,登录失败....");

}

}

**public** **static** **boolean** checkUser(User user) {

String sql = "select \* from t\_user where username='" + user.getUsername() + "' and password='"+ user.getPassword() + "' ";

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

Connection conn = **null**;

Statement st = **null**;

ResultSet rs = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

rs = st.executeQuery(sql);

**if** (rs.next()) {

**return** **true**;

} **else** {

**return** **false**;

}

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, st, rs);

}

**return** **false**;

}

}

**可以使用PrepareStatment防止sql注入**

* 可以通过调用 Connection 对象的 preparedStatement() 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() 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

**package** com.guigu.crud;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** com.guigu.model.User;

**import** com.guigu.utils.JdbcUtils;

**public** **class** TestSqlInjection2 {

**public** **static** **void** main(String[] args) {

User user = **new** User();

user.setUsername(" a' or password=");

user.setPassword(" or '1'='1");//

**if**(*checkUser*(user)) {

System.***out***.println("用户名和名正确,登录成功....");

}**else** {

System.***out***.println("用户名和密码不正确,,登录失败....");

}

}

**public** **static** **boolean** checkUser(User user) {

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

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

Connection conn = **null**;

// Statement st = null;

PreparedStatement ps=**null**;

ResultSet rs = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

ps=conn.prepareStatement(sql);

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

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

rs=ps.executeQuery();

**if** (rs.next()) {

**return** **true**;

} **else** {

**return** **false**;

}

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, rs);

}

**return** **false**;

}

}

## 2.2PreparedStatement vs Statement

**面试题**

* 代码的可读性和可维护性.
* 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

## 2.3数据库事务

### 2.3.1数据库事务概念

**一、数据库事务的概念**

* 在数据库中,所谓事务是指**一组逻辑操作单元**,**使数据从一种状态变换到另一种状态**。
* 为确保数据库中数据的**一致性**,数据的操纵应当是离散的成组的逻辑单元:当它全部完成时,数据的一致性可以保持,而当这个单元中的一部分操作失败,整个事务应全部视为错误,所有从起始点以后的操作应全部回退到开始状态。
* 事务的操作:先定义开始一个事务,然后对数据作修改操作,这时如果**提交**(COMMIT),这些修改就永久地保存下来,如果**回退**(ROLLBACK),数据库管理系统将放弃所作的所有修改而回到开始事务时的状态。

二、**数据库事务拥有四大特性：**

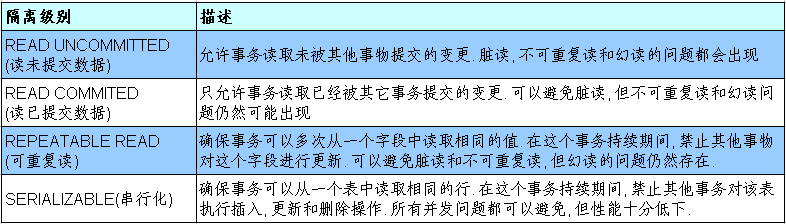
* 事务的ACID(acid)属性
  + 1. 原子性（Atomicity）  
    原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
  + 2. 一致性（Consistency）  
    事务必须使数据库从一个一致性状态变换到另外一个一致性状态。
  + 3. 隔离性（Isolation）  
    事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
  + 4. 持久性（Durability）  
    持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响

**三、由于数据库事务的隔离性引发的并发问题**

* 对于同时运行的多个事务, 当这些事务访问数据库中相同的数据时, 如果没有采取必要的隔离机制, 就会导致各种并发问题:
  + **脏读**: 对于两个事物 T1, T2, T1 读取了已经被 T2 更新但还**没有被提交**的字段. 之后, 若 T2 回滚, T1读取的内容就是临时且无效的.
  + **不可重复读**: 对于两个事物 T1, T2, T1 读取了一个字段, 然后 T2 **更新**了该字段. 之后, T1再次读取同一个字段, 值就不同了.
  + **幻读**: 对于两个事物 T1, T2, T1 从一个表中读取了一个字段, 然后 T2 在该表中**插入**了一些新的行. 之后, 如果 T1 再次读取同一个表, 就会多出几行.
* 数据库事务的隔离性: 数据库系统必须具有隔离并发运行各个事务的能力, 使它们不会相互影响, 避免各种并发问题.
* 一个事务与其他事务隔离的程度称为隔离级别. 数据库规定了多种事务隔离级别, 不同隔离级别对应不同的干扰程度, 隔离级别越高, 数据一致性就越好, 但并发性越弱

**四、为了解决数据库隔离性引发的并发问题定义了四种隔离级别**

* 数据库提供的 4 种事务隔离级别:



* Oracle 支持的 2 种事务隔离级别：**READ COMMITED**, SERIALIZABLE. Oracle 默认的事务隔离级别为: READ COMMITED
* Mysql 支持 4 中事务隔离级别. Mysql 默认的事务隔离级别为: REPEATABLE READ

**package** com.guigu.crud;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** com.guigu.utils.JdbcUtils;

**public** **class** TransactionTest {

**public** **static** **void** main(String[] args) {

*testTransfer*();

}

/\*\*

\* 完成转账小程序

\*/

**public** **static** **void** testTransfer() {

Connection conn = **null**;

Statement st = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

st = conn.createStatement();

//在事务开始时 设置事务默认行为 不自动提交

conn.setAutoCommit(**false**);

conn.setTransactionIsolation(Connection.***TRANSACTION\_SERIALIZABLE***);

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

String sql = "update t\_stu set money=money-500 where id=1";

st.execute(sql);

//.....转账需要消耗一定的时间

// System.out.println(10/0);

sql = "update t\_stu set money=money+500 where id=2";

st.execute(sql);

conn.commit();

} **catch** (Exception e) {

e.printStackTrace();

**try** {

conn.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

} **finally** {

JdbcUtils.*freeAll*(conn, st, **null**);

}

}

}

### 2.3.2批量处理JDBC语句提高处理速度

**package** com.guigu.jdbctest;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** com.guigu.utils.JdbcUtils;

**public** **class** BatchStatmentTest {

/\*\*

\* 向oracel数据库 中 插入10万条记录

\*

\*/

**public** **static** **void** main(String[] args) {

// testStatementBatch();

// testStatementBatch2();

// testPrepareStatementBatch();

*testPrepareStatementBatch2*();

}

//方式一

**public** **static** **void** testStatementBatch() {

Connection conn = **null**;

Statement st = **null**;

String sql = **null**;

**try** {

conn=JdbcUtils.*getConnection*();

st = conn.createStatement();

// 记录开始时间

**long** begin = System.*currentTimeMillis*();

**for** (**int** i = 0; i < 100000; i++) {

// insert into customer values(1,'name\_guigu\_1')

sql = "insert into customer values(" + (i + 1) + ", 'nane\_guigu\_" + i + "')";

st.execute(sql);

}

**long** end = System.*currentTimeMillis*();

System.***out***.println("总的耗时时间是：" + (end - begin)); //146028

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, st, **null**);

}

}

//方式二 使用批量操作

**public** **static** **void** testStatementBatch2() {

Connection conn = **null**;

Statement st = **null**;

String sql = **null**;

**try** {

conn=JdbcUtils.*getConnection*();

st = conn.createStatement();

// 记录开始时间

**long** begin = System.*currentTimeMillis*();

**for** (**int** i = 0; i < 100000; i++) {

// insert into customer values(1,'name\_guigu\_1')

sql = "insert into customer values(" + (i + 1) + ", 'nane\_guigu\_" + i + "')";

st.addBatch(sql);

}

st.executeBatch();

**long** end = System.*currentTimeMillis*();

System.***out***.println("总的耗时时间是：" + (end - begin)); //总的耗时时间是：125488

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, st, **null**);

}

}

//方式三 使用PrepareStatement

**public** **static** **void** testPrepareStatementBatch() {

Connection conn = **null**;

PreparedStatement ps = **null**;

String sql = **null**;

**try** {

conn=JdbcUtils.*getConnection*();

sql="insert into customer values(?,?)";

ps=conn.prepareStatement(sql);

// 记录开始时间

**long** begin = System.*currentTimeMillis*();

**for** (**int** i = 0; i < 1000000; i++) {

ps.setInt(1, i+1);

ps.setString(2, "name\_guigu\_"+i);

ps.addBatch();

}

ps.executeBatch();

**long** end = System.*currentTimeMillis*();

System.***out***.println("总的耗时时间是：" + (end - begin)); //总的耗时时间是：829 总的耗时时间是：41564

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, **null**);

}

}

//方式四 使用小算法

**public** **static** **void** testPrepareStatementBatch2() {

Connection conn = **null**;

PreparedStatement ps = **null**;

String sql = **null**;

**try** {

conn=JdbcUtils.*getConnection*();

sql="insert into customer values(?,?)";

ps=conn.prepareStatement(sql);

// 记录开始时间

**long** begin = System.*currentTimeMillis*();

**for** (**int** i = 0; i < 1000000; i++) {

ps.setInt(1, i+1);

ps.setString(2, "name\_guigu\_"+i);

//积累sql

ps.addBatch();

**if**((i+1)%500==0) {

ps.executeBatch();//批量执行固定内容的数据

ps.clearBatch(); //清空缓存中的内容

}

}

**long** end = System.*currentTimeMillis*();

System.***out***.println("总的耗时时间是：" + (end - begin)); //总的耗时时间是：1420 总的耗时时间是：3812

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, ps, **null**);

}

}

}

## 2.4DBUtils

* commons-dbutils 是 Apache 组织提供的一个开源 JDBC工具类库，它是对JDBC的简单封装，学习成本极低，并且使用dbutils能极大简化jdbc编码的工作量，同时也不会影响程序的性能。
* API介绍：
  + org.apache.commons.dbutils.QueryRunner
  + org.apache.commons.dbutils.ResultSetHandler
  + 工具类
    - org.apache.commons.dbutils.DbUtils、。
* 该类简单化了SQL查询，它与ResultSetHandler组合在一起使用可以完成大部分的数据库操作，能够大大减少编码量。
* QueryRunner类提供了两个构造方法：
  + 默认的构造方法
  + 需要一个 javax.sql.DataSource 来作参数的构造方法。

**package** com.guigu.dbutilstest;

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** java.util.List;

**import** java.util.Map;

**import** org.apache.commons.dbutils.QueryRunner;

**import** org.apache.commons.dbutils.handlers.BeanListHandler;

**import** org.apache.commons.dbutils.handlers.MapHandler;

**import** org.apache.commons.dbutils.handlers.MapListHandler;

**import** org.apache.commons.dbutils.handlers.ScalarHandler;

**import** org.junit.Test;

**import** com.guigu.model.Student;

**import** com.guigu.utils.JdbcUtils;

**public** **class** DBUtilsTest {

// 1.创建QueryRunner的实体类

QueryRunner queryRunner = **new** QueryRunner();

Connection conn = **null**;

// 查询某个字段的数据

@Test

**public** **void** testScalarHandler() {

String sql = "select sname from student where sid=?";

**try** {

conn = JdbcUtils.*getConnection*();

Object obj =queryRunner.query(conn, sql, **new** ScalarHandler(), 3);

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

} **catch** (Exception e) {

e.printStackTrace();

}**finally** {

JdbcUtils.*freeAll*(conn, **null**, **null**);

}

}

// 想把查询的结果封装为List<Map<String,Object>> mapList

@Test

**public** **void** testMapListHandler() {

String sql = "select \* from student";

List<Map<String, Object>> listMap = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

listMap = queryRunner.query(conn, sql, **new** MapListHandler());

} **catch** (Exception e) {

e.printStackTrace();

}**finally** {

JdbcUtils.*freeAll*(conn, **null**, **null**);

}

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

}

// 查询出结果类型是Map<> student1={sid=1,sname='张三',sage=33}

@Test

**public** **void** testMapHandler() {

String sql = "select \* from student where sid=?";

Map<String, Object> map = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

map = queryRunner.query(conn, sql, **new** MapHandler(), 3);

} **catch** (Exception e) {

e.printStackTrace();

}**finally** {

JdbcUtils.*freeAll*(conn, **null**, **null**);

}

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

}

// 查询List<T> 的操作

@Test

**public** **void** testBeanListHander() {

String sql = "select \* from student";

List<Student> students = **null**;

**try** {

conn = JdbcUtils.*getConnection*();

students = queryRunner.query(conn, sql, **new** BeanListHandler<>(Student.**class**));

} **catch** (Exception e) {

e.printStackTrace();

}**finally** {

JdbcUtils.*freeAll*(conn, **null**, **null**);

}

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

}

// 删除操作

@Test

**public** **void** testQueryRuunnerUpdate() {

String sql = "delete from t\_user where id in(?,?,?)";

**try** {

conn = JdbcUtils.*getConnection*();

queryRunner.update(conn, sql, 2, 3, 4);

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

JdbcUtils.*freeAll*(conn, **null**, **null**);

}

}

}

### 2.4.2设计通用的DAO

**package** com.guigu.dbutilstest;

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** java.util.List;

**public** **interface** BaseDAO<T> {

/\*\*

\* 批量处理的方法

\* **@param** connection

\* **@param** sql

\* **@param** args 填充占位符

\* **@throws** SQLException

\*/

**void** batch(Connection connection,String sql,Object[] ...args) **throws** SQLException;

/\*\*

\* 返回一个具体的值 比如 查询总的人数 查询某个人的email

\* **@param** connection

\* **@param** sql

\* **@param** args

\* **@return**

\* **@throws** SQLException

\*/

<E> E getForValue(Connection connection,String sql,Object ...args) **throws** SQLException;

/\*\*

\* 返回一个T类型的列表

\* **@param** connection

\* **@param** sql

\* **@param** args

\* **@return**

\* **@throws** SQLException

\*/

List<T> getForList(Connection connection,String sql,Object ...args) **throws** SQLException;

/\*\*

\* 返回一个T类型的对象

\* **@param** connection

\* **@param** sql

\* **@param** args

\* **@return**

\* **@throws** SQLException

\*/

T get(Connection connection,String sql,Object...args)**throws** SQLException;

/\*\*

\* INSERT UPDATE DELETE都可以使用这个方法

\* **@param** connection

\* **@param** sql

\* **@param** args

\* **@throws** SQLException

\*/

**void** update(Connection connection,String sql, Object ...args) **throws** SQLException;

}

**package** com.guigu.dbutilstest;

**import** java.lang.reflect.ParameterizedType;

**import** java.lang.reflect.Type;

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** java.util.List;

**import** org.apache.commons.dbutils.QueryRunner;

**import** org.apache.commons.dbutils.handlers.BeanHandler;

**import** org.apache.commons.dbutils.handlers.BeanListHandler;

**import** org.apache.commons.dbutils.handlers.ScalarHandler;

**public** **class** BaseDAOImpl<T> **implements** BaseDAO<T> {

**private** QueryRunner queryRunner = **null**;

// 定义类的类型

**private** Class<T> type;

**public** BaseDAOImpl() {

queryRunner = **new** QueryRunner();

//利用反射得到该类的类型

Type genType = getClass().getGenericSuperclass();

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

type = (Class) params[0];

}

@Override

**public** **void** batch(Connection connection, String sql, Object[]... args) **throws** SQLException {

queryRunner.batch(connection, sql, args);

}

@Override

**public** <E> E getForValue(Connection connection, String sql, Object... args) **throws** SQLException {

**return** (E)queryRunner.query(connection,sql, **new** ScalarHandler(),args);

}

@Override

**public** List<T> getForList(Connection connection, String sql, Object... args) **throws** SQLException {

**return** queryRunner.query(connection, sql, **new** BeanListHandler<>(type),args);

}

@Override

**public** T get(Connection connection, String sql, Object... args) **throws** SQLException {

**return** queryRunner.query(connection, sql, **new** BeanHandler<>(type),args);

}

@Override

**public** **void** update(Connection connection, String sql, Object... args) **throws** SQLException {

queryRunner.update(connection, sql, args);

}

}

## 2.5调用plsql中的存储过程和函数

*--存储过程*

create or replace procedure testAdd(

x in number,

y in number,

z in out number

)

is

begin

z:= x+y;

end;

*---函数*

create or replace function testAdd2(

x number,

y number

)

return number

is

z number;

begin

z:=x+y;

return z;

end ;

**package** com.guigu.dbutilstest;

**import** java.sql.CallableStatement;

**import** java.sql.Connection;

**import** java.sql.Types;

**import** org.junit.Test;

**import** com.guigu.utils.JdbcUtils;

**public** **class** TestPlsql {

//调用存储过程

@Test

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

Connection conn = JdbcUtils.*getConnection*();

CallableStatement cs = conn.prepareCall("{ call testAdd(?,?,?)}");

cs.setInt(1, 30);

cs.setInt(2, 20);

//注册输入参数

cs.registerOutParameter(3, Types.***INTEGER***);

cs.execute();

**int** result =cs.getInt(3);

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

}

//调用函数

@Test

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

Connection conn = JdbcUtils.*getConnection*();

CallableStatement cs = conn.prepareCall("{? = call testAdd2(?,?)}");

cs.setInt(2, 100);

cs.setInt(3, 300);

// 注册输出参数

cs.registerOutParameter(1, Types.***INTEGER***);

cs.execute();

**int** result = cs.getInt(1);

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

}

}