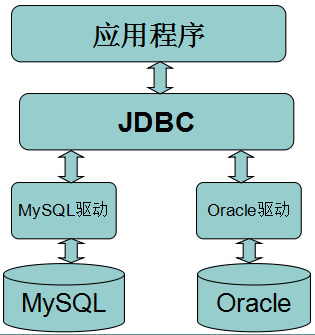
# 一、JDBC

JDBC全称为：Java Data Base Connectivity,它是由一些接口和类构成的API，可以执行SQL语句的Java API

* 市面上有非常多的数据库，本来我们是需要根据不同的数据库学习不同的API，sun公司为了简化这个操作，定义了JDBC API【接口】,数据库厂商负责实现。
* 对于我们来说，操作数据库都是在JDBC API【接口】上，使用不同的数据库，只要用数据库厂商提供的数据库驱动程序即可。这大大简化了我们的学习成本



　JDBC全称为：Java Data Base Connectivity（java数据库连接），它主要由接口组成。  
　　组成JDBC的２个包：  
　　　java.sql  
　　　javax.sql  
　　开发JDBC应用需要以上2个包的支持外，还需要导入相应JDBC的数据库实现(即数据库驱动)。

## 编写简单版JDBC程序

1、在mysql中创建一个库，并创建user表和插入表的数据。

create database day14 character set utf8 collate utf8\_general\_ci;

use day14;

create table users(

id int primary key,

name varchar(40),

password varchar(40),

email varchar(60),

birthday date

);

insert into users(id,name,password,email,birthday) values(1,'zhansan','123456','zs@sina.com','1980-12-04');

insert into users(id,name,password,email,birthday) values(2,'lisi','123456','lisi@sina.com','1981-12-04');

insert into users(id,name,password,email,birthday) values(3,'wangwu','123456','wangwu@sina.com','1979-12-04');

2、新建一个Java工程，并导入数据驱动。

建lib包，导入驱动。

public class JdbcFirstDemo {

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

//要连接的数据库URL

**String url = "jdbc:mysql://localhost:3306/day14?serverTimezone=UTC&useSSL=false";**

//连接的数据库时使用的用户名

String username = "root";

//连接的数据库时使用的密码

String password = "111111";

//1.加载驱动

//DriverManager.registerDriver(new com.mysql.jdbc.Driver());不推荐使用这种方式来加载驱动

Class.forName("com.mysql.jdbc.Driver");//推荐使用这种方式来加载驱动

**Class.forName()的作用就是把类加载到JVM中，它返回一个与带有给定字符串名的类或接口相关联的Class对象，并且JVM会加载这个类，同时JVM会执行该类的静态代码段。**

//2.获取与数据库的链接

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

**//3.获取用于向数据库发送sql语句的statement**

Statement st = conn.createStatement();

String sql = "select id,name,password,email,birthday from users";

//4.向数据库发sql,并获取代表结果集的resultset

ResultSet rs = st.executeQuery(sql);

//5.取出结果集的数据

while(rs.next()){

System.out.println("id=" + rs.getObject("id"));

System.out.println("name=" + rs.getObject("name"));

System.out.println("password=" + rs.getObject("password"));

System.out.println("email=" + rs.getObject("email"));

System.out.println("birthday=" + rs.getObject("birthday"));

}

//6.关闭链接，释放资源

rs.close();

st.close();

conn.close();

}

# 二、JDBC相关类详解

## DriverManager类讲解

Jdbc程序中的DriverManager用于加载驱动，并创建与数据库的链接，这个API的常用方法：

1. DriverManager.registerDriver(new Driver())
2. DriverManager.getConnection(url, user, password)，

注意：**在实际开发中并不推荐采用registerDriver方法注册驱动**。原因有二：  
　　　　1、查看**Driver的源代码**可以看到，如果采用此种方式，会导致驱动程序注册两次，也就是在内存中会有两个Driver对象。在类加载，有静态的代码注册过一次，再new会重复。  
　　　　2、程序依赖mysql的api，脱离mysql的jar包，程序将无法编译，将来程序切换底层数据库将会非常麻烦。

public class Driver extends NonRegisteringDriver implements java.sql.Driver {

static {

try {

java.sql.DriverManager.registerDriver(new Driver());

} catch (SQLException E) {

throw new RuntimeException("Can't register driver!");

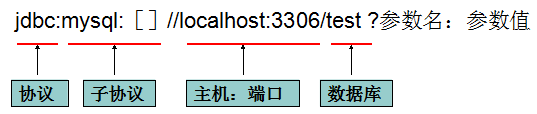
} }

根据类加载机制，当执行 Class.forName(driverClass) 获取其Class对象时，com.mysql.jdbc.Driver 就会被JVM加载，连接，并进行初始化，初始化就会执行静态代码块

推荐方式：**Class.forName("com.mysql.jdbc.Driver");**  
　　采用此种方式不会导致驱动对象在内存中重复出现，并且采用此种方式，程序仅仅只需要一个字符串，不需要依赖具体的驱动，使程序的灵活性更高。

## 数据库URL讲解

URL用于标识数据库的位置，通过URL地址告诉JDBC程序连接哪个数据库，URL的写法为：



　　常用数据库URL地址的写法：

* Oracle写法：jdbc:oracle:thin:@localhost:1521:sid
* SqlServer写法：jdbc:microsoft:sqlserver://localhost:1433; DatabaseName=sid
* MySql写法：jdbc:mysql://localhost:3306/sid

如果连接的是**本地的Mysql数据库**，并且连接使用的端口是3306，那么的url地址可以简写为： **jdbc:mysql:///数据库**

## Connection类讲解

Jdbc程序中的Connection，它用于代表数据库的链接，Collection是数据库编程中最重要的一个对象，**客户端与数据库所有交互都是通过connection对象完成的**，这个对象的常用方法：

createStatement()：创建向数据库发送sql的statement对象。

prepareStatement(sql) ：创建向数据库发送**预编译sql的PrepareSatement对象**。

prepareCall(sql)：创建执行存储过程的callableStatement对象。

setAutoCommit(boolean autoCommit)：设置事务是否自动提交。

commit() ：在链接上提交事务。

rollback() ：在此链接上回滚事务。

## Statement类讲解

Jdbc程序中的Statement对象用于向数据库发送SQL语句， Statement对象常用方法：

executeQuery(String sql) ：用于向数据发送查询语句。**executeQuery方法返回代表查询结果的ResultSet对象。**

executeUpdate(String sql)：用于向数据库发送insert、update或delete语句。**用于向数据库发送增、删、改的sql语句，executeUpdate执行完后，将会返回一个整数(即增删改语句导致了数据库几行数据发生了变化)。**

execute(String sql)：用于向数据库发送任意sql语句

addBatch(String sql) ：把多条sql语句放到一个批处理中。

executeBatch()：向数据库发送一批sql语句执行。

## ResultSet类讲解

Jdbc程序中的ResultSet用于代表Sql语句的执行结果。Resultset封装执行结果时，采用的类似于表格的方式。ResultSet 对象维护了**一个指向表格数据行的游标**，初始的时候，游标在第一行之前，调用ResultSet.next() 方法，可以使游标指向具体的数据行，进行调用方法**获取该行的数据。**

　　ResultSet既然用于封装执行结果的，所以该对象提供的都是用于获取数据的get方法：

　　获取任意类型的数据

　　　　getObject(int index)

　　　　getObject(string columnName)

　　获取指定类型的数据，例如：

**//根据获取列的数据类型，分别调用rs的相应方法映射到java对象中**

　　　　getString(int index)

　　　　getString(String columnName)

　　ResultSet还提供了对结果集进行滚动的方法：

next()：移动到下一行

Previous()：移动到前一行

absolute(int row)：移动到指定行

beforeFirst()：移动resultSet的最前面。

afterLast() ：移动到resultSet的最后面。

## 释放资源

Jdbc程序运行完后，切记要释放程序在运行过程中，创建的那些与数据库进行交互的对象，这些对象通常是**ResultSet, Statement和Connection对象**，特别是Connection对象，它是非常稀有的资源，用完后必须马上释放，如果Connection不能及时、正确的关闭，极易导致系统宕机。Connection的使用原则是尽量晚创建，尽量早的释放。

**为确保资源释放代码能运行，资源释放代码也一定要放在finally语句中**。

# 三、使用jdbc对数据库增删改查

## 在mysql中创建一个库，并创建user表和插入表的数据

create database day14 character set utf8 collate utf8\_general\_ci;

use day14;

create table users(

id int primary key,

name varchar(40),

password varchar(40),

email varchar(60),

birthday date

);

insert into users(id,name,password,email,birthday) values(1,'zhansan','123456','zs@sina.com','1980-12-04');

insert into users(id,name,password,email,birthday) values(2,'lisi','123456','lisi@sina.com','1981-12-04');

insert into users(id,name,password,email,birthday) values(3,'wangwu','123456','wangwu@sina.com','1979-12-04');

## 在src目录下创建一个db.properties文件

driver=com.mysql.jdbc.Driver

**url = jdbc:mysql://localhost:3306/day14?serverTimezone=UTC&useSSL=false**

username=root

password=111111

## 编写一个JdbcUtils工具类

**用于连接数据库，获取数据库连接和释放数据库连接**

package me.gacl.utils;

import java.io.InputStream;

import java.sql.Connection;**//不要mysql，后面不好维护**

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Properties;

public class JdbcUtils {

private static String driver = null;//直接拿

private static String url = null;

private static String username = null;

private static String password = null;

//只加载一次，静态代码块

static{

try{

//读取db.properties文件中的数据库连接信息

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

Properties prop = new Properties();

prop.load(in);

//获取数据库连接驱动

driver = prop.getProperty("driver");

//获取数据库连接URL地址

url = prop.getProperty("url");

//获取数据库连接用户名

username = prop.getProperty("username");

//获取数据库连接密码

password = prop.getProperty("password");

//加载数据库驱动

**Class.forName(driver);**

}catch (Exception e) {

**throw new ExceptionInInitializerError(e);**

}

}

public static Connection getConnection() throws SQLException{

return DriverManager.getConnection(url, username,password);

}

/\*\*

\* @Method: release

\* @Description: 释放资源，

\* 要释放的资源包括Connection数据库连接对象，负责执行SQL命令的Statement对象，存储查询结果的ResultSet对象

\*/

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

if(rs!=null){

try{

//关闭存储查询结果的ResultSet对象

rs.close();

}catch (Exception e) {

e.printStackTrace();

}

rs = null;

}

if(st!=null){

try{

//关闭负责执行SQL命令的Statement对象

st.close();

}catch (Exception e) {

e.printStackTrace();

}

}

if(conn!=null){

try{

//关闭Connection数据库连接对象

conn.close();

}catch (Exception e) {

e.printStackTrace();

}

}

}

}

### 采用单例的方式

**上面直接类名.方法调用，不需要实例。**

完成上面的JdbcUtils类：首先把上面类中的static方法都改成不是静态的，然后对外提供一个获得对象的方法：

// private static JdbcUtilsSing instance = new JdbcUtilsSing();

**private** **static** JdbcUtilsSing instance = **null**;

**private** JdbcUtilsSing() {

}

**public** **static** JdbcUtilsSing getInstance() {

    //延迟加载：等要调用我这个方法的时候才new一个对象，否则在static时候创建了一个对象，  有可能会用不到。

**if** (instance == **null**) {

        //解决多线程访问的问题，把这个同步如果加到方法上，那么我们每次的调用都会进行同步会影响性能 如果我们按照下面的方法

        //将同步缩小了范围之后只是会同步一次，下一次有对象了之后instance ！= null 同步方法就不会执行

**synchronized** (JdbcUtilsSing.**class**) {

**if** (instance == **null**) {

                instance = **new** JdbcUtilsSing();

            }

        }

    }

**return** instance;

}

## statement 进行CRUD操作

package me.gacl.demo;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import me.gacl.utils.JdbcUtils;

import org.junit.Test;

/\*\*

\* @ClassName: JdbcCRUDByStatement

\* @Description: **通过Statement对象完成对数据库的CRUD操作**

\*

\*/

public class JdbcCRUDByStatement {

@Test

public void insert(){

Connection conn = null;

Statement st = null;

ResultSet rs = null;

try{

//获取一个数据库连接

conn = JdbcUtils.getConnection();

//通过conn对象获取负责执行SQL命令的Statement对象

st = conn.createStatement();

//要执行的SQL命令

String sql = "insert into users(id,name,password,email,birthday) values(3,'白虎神皇','123','bhsh@sina.com','1980-09-09')";

//执行插入操作，executeUpdate方法返回成功的条数

int num = st.executeUpdate(sql);

if(num>0){

System.out.println("插入成功！！");

}

}catch (Exception e) {

e.printStackTrace();

}finally{

//SQL执行完成之后释放相关资源

JdbcUtils.release(conn, st, rs);

}

}

@Test

public void delete(){

Connection conn = null;

Statement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

String sql = "delete from users where id=3";

st = conn.createStatement();

int num = st.executeUpdate(sql);

if(num>0){

System.out.println("删除成功！！");

}

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

@Test

public void update(){

Connection conn = null;

Statement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

String sql = "update users set name='孤傲苍狼',email='gacl@sina.com' where id=3";

st = conn.createStatement();

int num = st.executeUpdate(sql);

if(num>0){

System.out.println("更新成功！！");

}

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

@Test

public void find(){

Connection conn = null;

Statement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

String sql = "select \* from users where id=3";

st = conn.createStatement();

rs = st.executeQuery(sql);

if(rs.next()){

System.out.println(rs.getString("name"));

}

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

}

## jdbc中的数据类型与日期问题

//birthday = new Date(rs.getDate("birthday").getTime());

//取值的时候是java.sql.Date赋值给一个父类类型java.util.Date是没有问题的

birthday = rs.getDate("birthday");  子类转化为父类

 //插入的时候传递的是java.util.Date而方法要求的是java.sql.Date,所以要进行一下转化

   ps.setDate(2, **new** java.sql.Date(birthday.getTime()));

## PreparedStatement对象

1. Statement对象编译SQL语句时，如果SQL语句有变量，就需要使用分隔符来隔开，如果变量非常多，就会使SQL变得非常复杂。**PreparedStatement可以使用占位符，简化sql的编写**
2. Statement会频繁编译SQL。**PreparedStatement可对SQL进行预编译，提高效率，预编译的SQL存储在PreparedStatement对象中**
3. **PreparedStatement防止SQL注入**。【Statement通过分隔符'++',编写永等式，可以不需要密码就进入数据库】

**注意**：rs = ps.executeQuery()如果你写成了rs = ps.executeQuery(sql)编译器也是不会报错的，因为PerparedSatement 是从Satement继承来的，所以如果你调用的是ps.executeQuery(String sql)这个带有参数的方法的时候，其实是调用的 Statement接口的方法，因为你上面sql语句是用PerparedStatement的方式进行构造的 用?代替了参数部分，**这个时候会包 错误说 ？ 是非法字符**，因为调用的是Statement的方法他不会对？进行处理直接传递给数据库执行，而数据库是不能解析这个字符的。

@Test

public void insert(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

//获取一个数据库连接

conn = JdbcUtils.getConnection();

//要执行的SQL命令，SQL中的参数使用?作为占位符

String sql = "insert into users(id,name,password,email,birthday) values(?,?,?,?,?)";

//通过conn对象获取负责执行SQL命令的prepareStatement对象

st = conn.prepareStatement(sql);

**//为SQL语句中的参数赋值，注意，索引是从1开始的**

/\*\*

\* SQL语句中各个字段的类型如下：

\* +----------+-------------+

| Field | Type |

+----------+-------------+

| id | int(11) |

| name | varchar(40) |

| password | varchar(40) |

| email | varchar(60) |

| birthday | date |

+----------+-------------+

\*/

st.setInt(1, 1);//id是int类型的

st.setString(2, "白虎神皇");//name是varchar(字符串类型)

st.setString(3, "123");//password是varchar(字符串类型)

st.setString(4, "bhsh@sina.com");//email是varchar(字符串类型)

st.setDate(5, new java.sql.Date(new Date().getTime()));//birthday是date类型.utils的Date转化成sql.date

//执行插入操作，executeUpdate方法返回成功的条数

int num = st.executeUpdate();

if(num>0){

System.out.println("插入成功！！");

}

}catch (Exception e) {

e.printStackTrace();

}finally{

//SQL执行完成之后释放相关资源

JdbcUtils.release(conn, st, rs);

}

}

# 四、[JDBC处理MySQL大数据](https://www.cnblogs.com/xdp-gacl/p/3982581.html)

## 基本概念

**大数据也称之为LOB(Large Objects)，LOB又分为：clob和blob，clob用于存储大文本，blob用于存储二进制数据**，例如图像、声音、二进制文等。

对MySQL而言只有blob，而没有clob，mysql存储大文本采用的是Text，Text和blob分别又分为：

**TINYTEXT、TEXT、MEDIUMTEXT和LONGTEXT  
TINYBLOB、BLOB、MEDIUMBLOB和LONGBLOB**

## 使用JDBC处理MySQL的大文本

对于MySQL中的Text类型，可调用如下方法设置

PreparedStatement.setCharacterStream(index, reader, length);//注意length长度须设置，并且设置为int型

对MySQL中的Text类型，可调用如下方法获取

reader = resultSet. getCharacterStream(String columnLabel);

string s = resultSet.getString(String columnLabel);

**测试范例**

**1、编写SQL测试脚本**

create database jdbcstudy;

use jdbcstudy;

create table testclob

(

id int primary key auto\_increment,

resume text

);

**2、编写测试代码如下：**

\* @Description:向数据库中插入大文本数据

public class JdbcOperaClob {

@Test

public void add(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

Reader reader = null;

try{

conn = JdbcUtils.getConnection();

String sql = "insert into testclob(resume) values(?)";

st = conn.prepareStatement(sql);

//这种方式获取的路径，其中的空格会被使用“%20”代替

String path = JdbcOperaClob.class.getClassLoader().getResource("data.txt").getPath();

//将“%20”替换回空格

path = path.replaceAll("%20", " ");

File file = new File(path);

reader = new FileReader(file);

**st.setCharacterStream(1, reader,(int) file.length());**

int num = st.executeUpdate();

if(num>0){

System.out.println("插入成功！！");

}

//关闭流

reader.close();

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

\* @Description: 读取数据库中的大文本数据

@Test

public void read(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

String sql = "select resume from testclob where id=2";

st = conn.prepareStatement(sql);

rs = st.executeQuery();

String contentStr ="";

String content = "";

if(rs.next()){

//使用resultSet.getString("字段名")获取大文本数据的内容

content = rs.getString("resume");

//使用resultSet.getCharacterStream("字段名")获取大文本数据的内容

Reader reader = rs.getCharacterStream("resume");

char buffer[] = new char[1024];

int len = 0;

FileWriter out = new FileWriter("D:\\1.txt");

while((len=reader.read(buffer))>0){

contentStr += new String(buffer);

out.write(buffer, 0, len);

}

out.close();

reader.close();

}

System.out.println(content);

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

System.out.println(contentStr);

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

}

## 使用JDBC处理MySQL的二进制数据

　　对于MySQL中的BLOB类型，可调用如下方法设置：

1 PreparedStatement. setBinaryStream(i, inputStream, length);

　　对MySQL中的BLOB类型，可调用如下方法获取：

1 InputStream in = resultSet.getBinaryStream(String columnLabel);

2 InputStream in = resultSet.getBlob(String columnLabel).getBinaryStream();

**4.1、 测试范例**

编写SQL测试脚本

create table testblob

(

id int primary key auto\_increment,

image longblob

);

编写测试代码如下：

public class JdbcOperaBlob {

/\*\*

\* @Method: add

\* @Description:向数据库中插入二进制数据

\* @Anthor:孤傲苍狼

\*

\*/

@Test

public void add(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

String sql = "insert into testblob(image) values(?)";

st = conn.prepareStatement(sql);

//这种方式获取的路径，其中的空格会被使用“%20”代替

String path = JdbcOperaBlob.class.getClassLoader().getResource("01.jpg").getPath();

//将“%20”替换会空格

path = path.replaceAll("%20", " ");

File file = new File(path);

FileInputStream fis = new FileInputStream(file);//生成的流

st.setBinaryStream(1, fis,(int) file.length());

int num = st.executeUpdate();

if(num>0){

System.out.println("插入成功！！");

}

fis.close();

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

/\*\*

\* @Method: read

\* @Description: 读取数据库中的二进制数据

\* @Anthor:孤傲苍狼

\*

\*/

@Test

public void read() {

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try {

conn = JdbcUtils.getConnection();

String sql = "select image from testblob where id=?";

st = conn.prepareStatement(sql);

st.setInt(1, 1);

rs = st.executeQuery();

if (rs.next()) {

//InputStream in = rs.getBlob("image").getBinaryStream();//这种方法也可以

InputStream in = rs.getBinaryStream("image");

int len = 0;

byte buffer[] = new byte[1024];

FileOutputStream out = new FileOutputStream("D:\\1.jpg");

while ((len = in.read(buffer)) > 0) {

out.write(buffer, 0, len);

}

in.close();

out.close();

}

} catch (Exception e) {

e.printStackTrace();

} finally {

JdbcUtils.release(conn, st, rs);

}

}

}

# 五、获取数据库的自动主键列、批处理

## 自动主键列

为什么要获取数据库的自动主键列数据?

应用场景:

有一张老师表，一张学生表。**现在来了一个新的老师，学生要跟着新老师上课。**

我首先要知道老师的id编号是多少，学生才能知道跟着哪个老师学习【**学生外键参照老师主键**】。

@Test  
public void test() {  
    Connection connection = null;  
    PreparedStatement preparedStatement = null;  
    ResultSet resultSet = null;  
    try {  
        connection = JdbcUtils.getConnection();  
        String sql = "INSERT INTO test(name) VALUES(?)";  
        preparedStatement = connection.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);  
        preparedStatement.setString(1, "ouzicheng");  
        if (preparedStatement.executeUpdate() > 0) {  
            //获取到自动主键列的值  
            resultSet = preparedStatement.getGeneratedKeys();

//返回一个ResultSet类型呢？因为他考虑到了联合主键    
            if (resultSet.next()) {  
                int id = resultSet.getInt(1);  
                System.out.println(id);  
            }  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        JdbcUtils.release(connection, preparedStatement, null);  
    }  
}

## 使用JDBC的批处理功能

在实际的项目开发中，有时候需要向数据库发送一批SQL语句执行，这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。  
　　JDBC实现批处理有两种方式：statement和preparedstatement

　 1、使用Statement对象添加要批量执行SQL语句，如下：

1 Statement.addBatch(sql1);

2 Statement.addBatch(sql2);

3 Statement.addBatch(sql3);

2、执行批处理SQL语句：Statement.executeBatch();  
　　3、清除批处理命令：Statement.clearBatch();

**采用Statement.addBatch(sql)方式实现批处理的优缺点**

采用Statement.addBatch(sql)方式实现批处理：

优点：可以向数据库发送多条不同的ＳＱＬ语句。

缺点：SQL语句没有预编译。当向数据库发送多条语句相同，但仅参数不同的SQL语句时，需重复写上很多条SQL语句。

    //批量插入数据

**static** **void** createBatch() **throws** SQLException {

        Connection conn = **null**;

        PreparedStatement ps = **null**;

        ResultSet rs = **null**;

**try** {

            conn = JdbcUtils.getConnection();

            String sql = "insert into user(name,birthday, money) values (?, ?, ?) ";

            ps = conn.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);

            //每循环一次都会形成一条新的sql语句被打包，直到循环完成然后进行批量的处理

            //那么可不可以无限量的增加呢？其实会产生内存溢出的情况，到底循环多少次进行打包才合适

            //这个值要经过测试

**for** (**int** i = 0; i < 100; i++) {

                ps.setString(1, "batch name" + i);

                ps.setDate(2, **new** Date(System.currentTimeMillis()));

                ps.setFloat(3, 100f + i);

                ps.addBatch();

            }

**int**[] is = ps.executeBatch();

        } **finally** {

            JdbcUtils.free(rs, ps, conn);

        }

    }

采用PreparedStatement.addBatch()实现批处理  
　　　　优点：发送的是预编译后的SQL语句，执行效率高。  
　　　　缺点：只能应用在SQL语句相同，但参数不同的批处理中。因此此种形式的批处理经常用于在同一个表中批量插入数据，或批量更新表的数据。

# 六、事务的概念

## 演示不同隔离级别下的并发问题

　　同时打开两个窗口模拟2个用户并发访问数据库

**1、当把事务的隔离级别设置为read uncommitted时，会引发脏读、不可重复读和虚读**

　　A窗口  
　　　　set transaction isolation level  read uncommitted;--设置A用户的数据库隔离级别为Read uncommitted(读未提交)  
　　　　start transaction;--开启事务  
　　　　select \* from account;--查询A账户中现有的钱，转到B窗口进行操作  
　　　　select \* from account--发现a多了100元，这时候A读到了B未提交的数据（脏读）

B窗口  
　　　　start transaction;--开启事务  
　　　　update account set money=money+100 where name='A';--不要提交，转到A窗口查询

**2、当把事务的隔离级别设置为read committed时，会引发不可重复读和虚读，但避免了脏读**

A窗口  
　　　　set transaction isolation level  read committed;  
　　　　start transaction;  
　　　　select \* from account;--发现a帐户是1000元，转到b窗口  
　　　　select \* from account;--发现a帐户多了100,这时候，a读到了别的事务提交的数据，两次读取a帐户读到的是不同的结果（不可重复读）  
　　B窗口  
　　　　start transaction;  
　　　　update account set money=money+100 where name='a';  
　　　　commit;--转到a窗口

**3、当把事务的隔离级别设置为repeatable read(mysql默认级别)时，会引发虚读，但避免了脏读、不可重复读**

A窗口  
　　　　set transaction isolation level repeatable read;  
　　　　start transaction;  
　　　　select \* from account;--发现表有4个记录，转到b窗口  
　　　　select \* from account;--可能发现表有5条记录，这时候发生了a读取到另外一个事务插入的数据（虚读）  
　　B窗口  
　　　　start transaction;  
　　　　insert into account(name,money) values('ggg',1000);  
　　　　commit;--转到a窗口

**4、当把事务的隔离级别设置为Serializable时，会避免所有问题**

A窗口  
　　　　set transaction isolation level Serializable;  
　　　　start transaction;  
　　　　select \* from account;--转到b窗口  
  
　　B窗口  
　　　　start transaction;  
　　　　insert into account(name,money) values('ggg',1000);--发现不能插入，只能等待a结束事务才能插入

## JDBC中使用事务

当Jdbc程序向数据库获得一个Connection对象时，默认情况下这个Connection对象会自动向数据库提交在它上面发送的SQL语句。若想关闭这种默认提交方式，让多条SQL在一个事务中执行，可使用下列的JDBC控制事务语句

* Connection.setAutoCommit(false);//开启事务(start transaction)
* Connection.rollback();//回滚事务(rollback)
* Connection.commit();//提交事务(commit)

JDBC使用事务范例

create table account(

id int primary key auto\_increment,

name varchar(40),

money float

);

insert into account(name,money) values('A',1000);

insert into account(name,money) values('B',1000);

insert into account(name,money) values('C',1000);

public class TransactionDemo1 {

@Test

public void testTransaction1(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

conn.setAutoCommit(false);//通知数据库开启事务(start transaction)

String sql1 = "update account set money=money-100 where name='A'";

st = conn.prepareStatement(sql1);

st.executeUpdate();

String sql2 = "update account set money=money+100 where name='B'";

st = conn.prepareStatement(sql2);

st.executeUpdate();

conn.commit();//上面的两条SQL执行Update语句成功之后就通知数据库提交事务(commit)

System.out.println("成功！！！"); //log4j

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

/\*\*

\* @Method: testTransaction1

\* @Description: 模拟转账过程中出现异常导致有一部分SQL执行失败后让数据库自动回滚事务

\*

\*/

@Test

public void testTransaction2(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

conn.setAutoCommit(false);//通知数据库开启事务(start transaction)

String sql1 = "update account set money=money-100 where name='A'";

st = conn.prepareStatement(sql1);

st.executeUpdate();

//用这句代码模拟执行完SQL1之后程序出现了异常而导致后面的SQL无法正常执行，事务也无法正常提交，此时数据库会自动执行回滚操作

int x = 1/0;

String sql2 = "update account set money=money+100 where name='B'";

st = conn.prepareStatement(sql2);

st.executeUpdate();

conn.commit();//上面的两条SQL执行Update语句成功之后就通知数据库提交事务(commit)

System.out.println("成功！！！");

}catch (Exception e) {

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

/\*\*

\* @Method: testTransaction1

\* @Description: 模拟转账过程中出现异常导致有一部分SQL执行失败时手动通知数据库回滚事务

\* @Anthor:孤傲苍狼

\*

\*/

@Test

public void testTransaction3(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

conn = JdbcUtils.getConnection();

conn.setAutoCommit(false);//通知数据库开启事务(start transaction)

String sql1 = "update account set money=money-100 where name='A'";

st = conn.prepareStatement(sql1);

st.executeUpdate();

//用这句代码模拟执行完SQL1之后程序出现了异常而导致后面的SQL无法正常执行，事务也无法正常提交

int x = 1/0;

String sql2 = "update account set money=money+100 where name='B'";

st = conn.prepareStatement(sql2);

st.executeUpdate();

conn.commit();//上面的两条SQL执行Update语句成功之后就通知数据库提交事务(commit)

System.out.println("成功！！！");

}catch (Exception e) {

try {

//捕获到异常之后手动通知数据库执行回滚事务的操作

conn.rollback();

} catch (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

}

## 设置事务回滚点

\* JDBC中使用事务来模似转帐

create table account(

id int primary key auto\_increment,

name varchar(40),

money float

);

insert into account(name,money) values('A',1000);

insert into account(name,money) values('B',1000);

insert into account(name,money) values('C',1000);

\*

public class TransactionDemo2 {

@Test

public void testTransaction1(){

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

Savepoint sp = null;

try{

conn = JdbcUtils.getConnection();

conn.setAutoCommit(false);//通知数据库开启事务(start transaction)

String sql1 = "update account set money=money-100 where name='A'";

st = conn.prepareStatement(sql1);

st.executeUpdate();

//设置事务回滚点

sp = conn.setSavepoint();

String sql2 = "update account set money=money+100 where name='B'";

st = conn.prepareStatement(sql2);

st.executeUpdate();

//程序执行到这里出现异常，后面的sql3语句执行将会中断

int x = 1/0;

String sql3 = "update account set money=money+100 where name='C'";

st = conn.prepareStatement(sql3);

st.executeUpdate();

conn.commit();

}catch (Exception e) {

try {

/\*\*

\* 我们在上面向数据库发送了3条update语句，

\* sql3语句由于程序出现异常导致无法正常执行，数据库事务而已无法正常提交，

\* 由于设置的事务回滚点是在sql1语句正常执行完成之后，sql2语句正常执行之前,

\* 那么通知数据库回滚事务时，不会回滚sql1执行的update操作

\* 只会回滚到sql2执行的update操作，也就是说，上面的三条update语句中，sql1这条语句的修改操作起作用了

\* sql2的修改操作由于事务回滚没有起作用，sql3由于程序异常没有机会执行

\*/

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

**conn.rollback(sp);//回滚到设置的事务回滚点**

//回滚了要记得通知数据库提交事务

conn.commit();

} catch (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

}finally{

JdbcUtils.release(conn, st, rs);

}

}

}

## JTA分布式事务的简要介绍

跨越多个数据源的事务，使用JTA容器实现事务。   
分成两阶段提交。

**Java代码**

javax.transaction.UserTransaction tx = (UserTransaction)ctx.lookup(“jndiName");

    tx.begin();

    //connection1 connection2 (可能来自不同的数据库)…

    tx.commit();//tx.rollback();

# 七、使用JDBC调用存储过程

**调用存储过程的语法：**

{call <procedure-name>[(<arg1>,<arg2>, ...)]}

**调用函数的语法：**

{?= call <procedure-name>[(<arg1>,<arg2>, ...)]}

**如果是Output类型的，那么在JDBC调用的时候是要注册的**。如下代码所示：

/\*

### Mysql query Brower 中创建的一个存储过程

模拟添加用户

DELIMITER $$

DROP PROCEDURE IF EXISTS `jdbc`.`addUser` $$

CREATE PROCEDURE `jdbc`.`addUser` (in name varchar(45), in sex varchar(10), out pid int)

BEGIN

insert into user(name, sex) values(name, sex);

select last\_insert\_id() into pid;

END $$

DELIMITER ;

//在`addUser` (in name varchar(45), in sex varchar(10), out pid int)中

//in:表示输入参数 out：表示输出参数

//last\_insert\_id()是mysql所特有的一个函数，可以查询出最后一次插入到数据库的那条数据的id

### jdbc调用存储过程

delimiter $$  
    CREATE PROCEDURE demoSp(IN inputParam VARCHAR(255), INOUT inOutParam varchar(255))  
    BEGIN  
        SELECT CONCAT('zyxw---', inputParam) into inOutParam;  
    END $$  
delimiter ;  
\*/  
//我们在JDBC调用存储过程,就像在调用方法一样  
public class Demo9 {  
    public static void main(String[] args) {  
        Connection connection = null;  
        CallableStatement callableStatement = null;  
        try {  
            connection = JdbcUtils.getConnection();

 // call :是固定写法， addUser 是我们在数据库中定义的存储过程的名字后面指定参数

         //如果没有任何参数 addUser后的括号也要写上

            callableStatement = connection.prepareCall("{call demoSp(?,?)}");  
            callableStatement.setString(1, "nihaoa");  
            //注册第2个参数,类型是VARCHAR  
            callableStatement.registerOutParameter(2, Types.VARCHAR);  
            callableStatement.execute();

            //获取传出参数[获取存储过程里的值]  
            String result = callableStatement.getString(2);  
            System.out.println(result);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }finally {  
            try {  
                connection.close();  
                callableStatement.close();  
            } catch (SQLException e) {  
                e.printStackTrace();  
            }  
        }  
  
    }  
}

### 参考资料

----------------------------------------------------------------------------------过程  
#修改mysql语句的结果符为//  
mysql > delimiter //  
  
#定义一个过程，获取users表总记录数，将10设置到变量count中  
create procedure simpleproc(out count int)  
begin  
    select count(id) into count from users;  
end  
//  
  
#修改mysql语句的结果符为;  
mysql > delimiter ;  
  
#调用过程，将结果覆给变量a，@是定义变量的符号  
call simpleproc(@a);  
  
#显示变量a的值  
select @a;  
  
//以下是Java调用Mysql的过程  
    String sql = "{call simpleproc(?)}";  
    Connection conn = JdbcUtil.getConnection();  
    CallableStatement cstmt = conn.prepareCall(sql);  
    cstmt.registerOutParameter(1,Types.INTEGER);  
    cstmt.execute();  
    Integer count = cstmt.getInt(1);  
    System.out.println("共有" + count + "人");  
  
----------------------------------------------------------------------------------函数  
  
#修改mysql语句的结果符为//  
mysql > delimiter //  
  
#定义一个函数，完成字符串拼接  
create function hello( s char(20) ) returns char(50)   
return concat('hello，',s,'!');  
//  
  
#修改mysql语句的结果符为;  
mysql > delimiter ;  
  
#调用函数  
select hello('world');  
  
//以下是Java调用Mysql的函数  
    String sql = "{? = call hello(?)}";  
    Connection conn = JdbcUtil.getConnection();  
    CallableStatement cstmt = conn.prepareCall(sql);  
    cstmt.registerOutParameter(1,Types.VARCHAR);  
    cstmt.setString(2,"zhaojun");  
    cstmt.execute();  
    String value = cstmt.getString(1);  
    System.out.println(value);  
    JdbcUtil.close(cstmt);  
    JdbcUtil.close(conn);