### 14.3 JDBC典型用法

掌握了标准的SQL命令之后，就可以使用JDBC进行数据库应用开发了。

#### 14.3.1 JDBC常用接口和类简介

JDBC提供了独立于数据库的统一的API，用来执行SQL命令。JDBC的API由以下常用的接口和类组成

DriverManager：用于管理JDBC驱动的服务类。程序中使用该类的主要功能是获取Connection对象，该类包含如下方法：

* public static synchronized Connection getConnection(String url,String user,String pass) throws SQLException：该方法获得url对应数据库的连接。

Connection：代表数据库连接对象，每个Connection代表一个物理连接会话。要想访问数据库，必须先获得数据库连接。该接口的常用方法如下：

* Statement createStatement() throws SQLException：该方法返回一个Statement对象。
* PreparedStatement prepareStatement(String sql) throws SQLException：该方法返回预编译的Statement对象，即将SQL语句提交到数据库进行预编译。
* CallableStatement prepareCall(String sql) throws SQLException：该方法返回CallableStatement对象，该对象用于调用存储过程。

以上3个方法都返回用于执行SQL语句的Statement对象，PrepareStatement、CallableStatement是Statement的子类，只有获取了Statement之后，才可以执行SQL语句。

除此之外，Connection还有如下几个用于控制事物的方法。

* SavePoint setSavepoint()：创建一个保存点。
* SavePoint setSavePoint(String name)：以指定名字来创建一个保存点。
* void setTransactionIsolation(int level)：设置事物的隔离级别。
* void rollback()：回滚事物
* void rollback(Savepoint point)：将事物回滚到指定的保存点。
* void setAutoCommit(boolean autoCommit)：关闭自动提交，打开事物。
* void commit()：提交事物

**Statement：用于执行SQL语句的工具接口。该对象既可以用于执行DDL、DCL语句，也可以用于执行DML语句，还可以用于执行SQL查询。当执行SQL查询时，返回查询到的结果集**。它的常用方法如下：

* **ResultSet executeQuery(String sql) throws SQLException：**该方法用于执行**查询语句**，并返回查询结果对应的ResultSet对象，该方法只能用于执行查询语句。
* **int executeUpdate(String sql) throws SQLException：**该方法用于执行**DML语句**，并返回受影响的**行数**，该方法也可以用于执行DDL语句，执行DDL语句将返回0.
* **boolean execute(String sql) throws SQLException：**该方可执行任何SQL语句。如果执行后第一个结果为ResultSet对象，则返回true；如果执行后第一个结果为受影响的行数或没有任何结果则返回false。

**PreparedStatement：预编译的Statement对象。PreparedStatement是Statement的子接口，它允许数据库预编译SQL语句，这些SQL语句通常带有参数，以后每次只改变SQL命令的参数，避免数据库每次都需要编译SQL语句，因此性能更好。相对于Statement而言，使用PreparedStatement执行SQL语句时，无须再传入SQL语句，只要为预编译的SQL传入参数即可。**因此它比Statement多了如下方法：

* void setXxx(int parameterIndex,Xxx value)：该方法根据传入参数值的类型不同，需要使用不同的方法。传入的值根据索引传给SQL语句中指定位置的参数。

**ResultSet：结果集对象。该对象包含访问查询结果的方法，ResultSet可以通过列索引或列名获得列数据。它包含了如下常用方法来移动记录指针。**

* void close() throws SQLException：释放ResultSet对象。
* boolean absolute(int row)：将结果集的记录指针移动到第row行，如果row是负数，则移动到倒数第row行。如果移动后的记录指针指向一条有效记录，则该方法返回true。
* void beforeFirst()：将ResultSet的记录指针定位到首行之前，这是ResultSet结果集记录指针的初始状态——记录指针的起始位置位于第一行之前。
* boolean first()：将ResultSet的记录指针定位到首行，如果移动后的记录指针指向一条有效记录，则返回true。
* boolean previous()：将ResultSet的记录指针定位到上一行，如果移动后的记录指针指向一条有效记录，则返回true。
* boolean next()：将ResultSet的记录指针定位到下一行，如果移动后的记录指针指向一条有效记录，则返回true。
* boolean last()：将ResultSet的记录指针定位到最后一行，如果移动后的记录指针指向一条有效记录，则该方法返回true。
* void afterLast()：将ResultSet的记录指针定位到最后一行。

当把记录指针移动到指定的行后，Result可以通过getXXX(int index)或者getXXX(String columnLabel)的方法来获取当前行指定列的值。前者通过列索引来获取值，后者通过列名来获取值。在Java7.0的版本之后增加了T getObject(int index,class type)方法和T getObject(String columnLabel,class Type)的两个增加泛型的方法来获取任意类型的值。

#### 14.3.2 JDBC编程步骤

大致了解了JDBC API的相关接口和类之后，下面就可以进行JDBC编程了，JDBC编程大致按如下步骤进行。

1. 加载数据库驱动。通常我们使用Class类的forName()静态方法来加载驱动，例如如下代码：

Class.forName(driverName); //加载MySQL的JDBC驱动类

上面代码中的driverName就是数据库驱动类所对应的字符串。例如：加载MySQL的驱动类代码如下：

Class.forName(“com.mysql.jdbc.Driver”);

加载SQLServer的驱动类代码如下：

Class.forName(“com.microsoft.sqlserver.jdbc.SQLServerDriver”);

加载Oracle的驱动类代码如下：

Class.forName(“oracle.jdbc.driver.OracleDriver”);

从上面代码可以看出，加载驱动时并不是真正使用数据库的驱动类，而只是使用数据库驱动类名的字符串而已。

1. 通过DriverManager获取数据库连接。DriverManager提供了如下方法：

//获取数据库连接

DriverManager.getConnection(String url,String userName,String passwrod);

当使用DriverManager来获取数据库连接时，通常需要传入三个参数，数据库的URL，登录数据库的用户名和密码。其中用户名和密码有DBA（数据库管理人员）来分配给开发者。而且这用户具有相应的权限才可以执行相应的SQL命令：

数据库URL通常遵循如下写法：

jdbc:subprotocol:other:stuff

上面URL中的jdbc是固定的，subprotocol是指连接到特定数据库的驱动，other、stuff都不是固定的，没有比较强的规律可循，不同数据库的URL写法可能存在较大的差异。MySQL数据库的URL写法如下：

jdbc:mysql://hostname:port/databaseName

SQLServer数据库的URL写法如下：

jdbc:sqlserver://hostname:prot/databaseName

Oracle数据库的URL写法如下：

jdbc:oracle:thin@hostname:port:databaseName

举例MySQL：

public String userName = "root";

public String password = "123456";

publi String url="jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

DriverManager.getConnection(url,dataName,dataPassword);

举例SQLServer：

public String userName = "sa";

public String password = "123456";

public String url="jdbc:sqlserver://127.0.0.1:1433;DatabaseName=test ";

DriverManager.getConnection(url,dataName,dataPassword);

举例Oracle：

public String userName = "scott";

public String password = "tiger";

public String url="jdbc:oracle:thin:@127.0.0.1:1521:test ";

DriverManager.getConnection(url,dataName,dataPassword);

1. 通过Connection对象创建Statement对象。Connection创建statement对象的方法有如下3个：

* createStatement()：创建基本的Statement对象。
* prepareStatement(String sql)：根据传入的SQL语句创建预编译的Statement对象。
* prepareCall(String sql)：根据传入的SQL语句创建CallableStatement对象。

1. 使用Statement执行SQL语句。所有的statement对象都有如下3个方法来执行SQL语句：

* execute()：可以执行任何SQL语句，但是比较麻烦。
* executeUpdate()：主要执行DML和DDL语句。执行DML语句返回受SQL语句影响的行数，执行DDL语句返回0.
* executeQuery()：主要用于执行查询语句，执行后返回代表查询结果的ResultSet对象。

1. 操作结果集，如果执行的SQL语句是查询语句，则执行结果返回一个ResultSet对象，该对象里保存了SQL语句查询的结果。程序可以通过操作该ResultSet对象来取出查询结果ResultSet对象主要提供了如下两类方法。

* next()、previous()、first()、last()、beforeFirst()、afterLast()、absolute()等移动记录指针的方法。
* getXxx()方法获取记录指针指向行、特定列的值。该方法既可使用列索引作为参数，也可以使用列名作为参数。使用列索引作为参数性能更好，使用列名作为参数可读性更好。

1. 回收数据库资源，包括关闭ResultSet、Statement和Connection等资源。

如下代码简单示范JDBC编程，并通过ResultSet获取结果集的过程。

举例1：

public List<Product> selectAll() throws Exception{

//加载驱动类

Class.forName(dirver);

//数据库链接类

Connection conn = DriverManager.getConnection(url,dataName,dataPassword);

String sql = "select \* from product";

PreparedStatement ps = conn.prepareStatement(sql);

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

ResultSet rs = ps.executeQuery();

while(rs.next()){

Product pt = new Product();

pt.setP\_id(rs.getString("p\_id"));

pt.setP\_name(rs.getString("p\_name"));

pt.setP\_price(rs.getString("p\_price"));

pt.setP\_note(rs.getString("p\_note"));

list.add(pt);

}

return list;

}

### 14.4 执行SQL语句的方式

前面介绍了JDBC执行查询等示例程序，实际上JDBC不仅可以执行查询，也可以执行DDL、DML等SQL语句，从而允许通过JDBC最大限度地控制数据库。

#### 14.4.1 使用executeUpdate方法执行DDL和DML语句

Statement提供了3个方法来执行SQL语句，前面已经介绍了使用executeQuery()来执行查询语句，下面将介绍使用executeUpdate()方法来执行DDL(数据定义)和DML(数据操纵)语句。使用Statement执行DDL和DML语句的步骤与执行普通查询语句的步骤基本相似，区别在于执行DDL语句返回值为0，执行DML语句后返回值为受影响的记录条数。

下面程序示范了使用executeUpdate()方法来创建数据表。在下面的程序中我们将连接数据库的信息写入到一个mysql.ini的配置文件中，并没有直接写入到程序中，这是比较成熟的的做法。因为当需要把程序从开发环境移植到生产环境中时，只需要更改配置文件就可以了，而无需更改程序代码。

举例：通过查询获取一条记录，对某条记录进行更改，首先查询出多有的记录！

商品类：Product

package com.langsin.jdbc;

public class Product {

private String id;

private String p\_name;

private String p\_price;

private String p\_code;

private String p\_type;

private String p\_fac;

private String p\_store;

public Product(String id, String p\_name, String p\_price, String p\_code,String p\_type, String p\_fac, String p\_store) {

this.id = id;

this.p\_name = p\_name;

this.p\_price = p\_price;

this.p\_code = p\_code;

this.p\_type = p\_type;

this.p\_fac = p\_fac;

this.p\_store = p\_store;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getP\_name() {

return p\_name;

}

public void setP\_name(String p\_name) {

this.p\_name = p\_name;

}

public String getP\_price() {

return p\_price;

}

public void setP\_price(String p\_price) {

this.p\_price = p\_price;

}

public String getP\_code() {

return p\_code;

}

public void setP\_code(String p\_code) {

this.p\_code = p\_code;

}

public String getP\_type() {

return p\_type;

}

public void setP\_type(String p\_type) {

this.p\_type = p\_type;

}

public String getP\_fac() {

return p\_fac;

}

public void setP\_fac(String p\_fac) {

this.p\_fac = p\_fac;

}

public String getP\_store() {

return p\_store;

}

public void setP\_store(String p\_store) {

this.p\_store = p\_store;

}

@Override

public String toString() {

return "主键："+this.id+" 商品名称："+this.p\_name+" 商品内码："+this.p\_code+" 商品类型："+this.p\_type+" 商品品牌："+this.p\_type+" 商品数量："+this.p\_store;

}

}

操作类：Test

package com.langsin.jdbc;

import java.io.FileReader;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.util.ArrayList;

import java.util.List;

import java.util.Properties;

import java.util.Scanner;

public class Test {

private String username;

private String password;

private String url;

private String driver;

private Connection conn = null;

private PreparedStatement statement = null;

private ResultSet rs = null;

private void getInformation() throws Exception{

Properties prop = new Properties();

prop.load(new FileReader("./user.properties"));

this.username = prop.getProperty("username");

this.password = prop.getProperty("password");

this.url = prop.getProperty("url");

this.driver = prop.getProperty("driver");

}

public void start() throws Exception{

this.getInformation();

Class.forName(this.driver);

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

String sql = "select \* from product";

this.statement = this.conn.prepareStatement(sql);

this.rs = this.statement.executeQuery();

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

while(this.rs.next()){

String id = this.rs.getString("id");

String p\_name = this.rs.getString("p\_name");

String p\_price = this.rs.getString("p\_price");

String p\_code = this.rs.getString("p\_code");

String p\_type = this.rs.getString("p\_type");

String p\_fac = this.rs.getString("p\_fac");

String p\_store = this.rs.getString("p\_store");

Product prod = new Product(id, p\_name, p\_price, p\_code, p\_type, p\_fac, p\_store);

list.add(prod);

}

System.out.println(" 主键 商品名称 内码 商品类型 商品数量：");

for(int i=0;i<list.size();i++){

System.out.println(list.get(i));

}

Scanner scan = new Scanner(System.in);

System.out.println("请输入要查询的商品名称：");

String name = scan.nextLine();

System.out.println(name);

Product prod = null;

for(int i=0;i<list.size();i++){

if(list.get(i).getP\_name().contains(name)){

prod = list.get(i);

break;

}

}

if(prod==null){

System.out.println("查询无结果！");

}else{

System.out.println("请输入要更改的数量：");

String num = scan.nextLine();

prod.setP\_store(num);

}

sql = "update product set p\_store=? where id=?";

this.statement = this.conn.prepareStatement(sql);

this.statement.setInt(1, Integer.parseInt(prod.getP\_store()));

this.statement.setString(2, prod.getId());

int result = this.statement.executeUpdate();

if(result>0){

System.out.println("更新成功");

System.out.println(prod);

}else{

System.out.println("更新失败！");

}

}

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

new Test().start();

}

}

举例2：使用executeUpdate来执行DDL语句，如创建表、建立约束、索引、视图、触发器等，如下示例来创建一个视图。

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "create or replace view user\_role as "+

"select user.\*,role.role\_cname from user,role where user.role\_id=role.role\_id";

PreparedStatement statement = conn.prepareStatement(sql);

statement.executeUpdate();

conn.close();

}

}

使用executeUpdate方法执行DDL语句时，大部分情况下返回的是0。

#### 14.4.2 使用execute方法来执行SQL语句

statement的execute方法几乎可以执行任何SQL语句，但它执行SQL语句时比较麻烦，通常我们没有必要使用execute方法来执行SQL语句，而是使用executeUpdate和executeQuery这两个更为简单的方法。但是如果我们不清楚要执行SQL语句的类型，则只能使用execute方法来执行SQL语句了。

使用execute方法执行SQL语句返回的值只是boolean类型，它表明执行该SQL语句是否返回了ResultSet对象。那么如何来获取执行SQL语句后得到的ResultSet对象呢？Statement对象提供了如下两个方法来获取执行结果。

* getResultSet方法：获取Statement执行查询语句所返回的ResultSet对象。
* getUpdateCount方法：获取Statement执行DML语句所影响的记录的行数。

下面程序示范了使用Statement对象的execute方法来执行任意的SQL语句，执行SQL语句时产生不同的输出。

示例1：

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "select \* from student";

Statement statement = conn.createStatement();

boolean flag = statement.execute(sql);

if(flag){

ResultSet set = statement.getResultSet();

while(set.next()){

System.out.println("姓名："+set.getString("stud\_cname")+" 年龄："+set.getString("stud\_age"));

}

}

conn.close();

}

}

示例2：

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "insert into student values('AB001','admin','管理员','123123',22,'详细地址',1)";

Statement statement = conn.createStatement();

boolean flag = statement.execute(sql);

if(flag==false){

int count = statement.getUpdateCount();

System.out.println("执行了SQL影响了"+count+"行！");

}

conn.close();

}

}

#### 14.4.3 使用PreparedStatement执行SQL语句

如果我们需要反复执行一条结构相似的SQL语句，例如如下SQL所示：

insert into student values(null,’zhangsan’,’22岁’);

insert into student values(null,’lisi,’22岁’);

对于这两条SQL而言，它们的结构相似，只是执行插入时插入的值不同而已。在这种情况下，我们可以使用如下带占位符（？）参数的SQL语句来代替它。

insert into student values(null,?,?)

Statement对象执行SQL语句时不允许使用问号占位符参数，而且这个问号占位符参数必须获得值以后才可以执行。为了满足这种功能，JDBC提供了PreparedStatement接口，它是Statement接口的子接口，它可以预编译SQL语句，预编译后的SQL语句存储在PreparedStatement对象中，然后可以使用该对象多次高效地执行该语句。简而言之，使用PreparedStatement比使用Statement效率要高。

创建PreparedStatement对象需要调用Connection对象的preparedStatement()方法，该方法需要传入一个SQL字符串，该SQL字符串可以包含占位符参数，如下SQL所示：

String sql = "update product set p\_store=? where id=?";

PreparedStatement statement = conn.prepareStatement(sql);

PreparedStatement也提供了execute()、executeUpate()、executeQuery()三个方法来执行SQL语句，不过这三个方法不需要传入参数，因为PreparedStatement已经存储了预编译的SQL语句。

使用PreparedStatement预编译SQL语句时，该SQL语句可以带占位符参数，因此在执行SQL语句之前必须为这些参数传入参数值，PreparedStatement提供了一系列的setXXX(int index,XXX value)方法来传入参数值。

下面程序示范了使用Statement和PreparedStatement分别插入100条记录的对比。使用Statement对象需要传入100条SQL语句，而使用PreparedStatement对象只需要传入1条SQL语句，然后100次为该PreparedStatement对象传入参数即可。

package com.langsin.jdbc;

import java.io.File;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

import java.util.Scanner;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

Scanner scan = new Scanner(new File("./user.txt"));

while(scan.hasNextLine()){

String line = scan.nextLine();

String[] values = line.split(",");

String sql = "insert into student values('"+values[0]+"','"+values[1]+"','"+values[2]+"','"+values[3]+"',"+values[4]+",'详细地址',1)";

Statement statement = conn.createStatement();

boolean flag = statement.execute(sql);

if(flag==false){

int count = statement.getUpdateCount();

System.out.println("执行了SQL影响了"+count+"行！");

}

}

conn.close();

}

}

示例2：

package com.langsin.jdbc;

import java.io.File;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.Statement;

import java.util.Scanner;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

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

PreparedStatement statement = conn.prepareStatement(sql);

Scanner scan = new Scanner(new File("./user.txt"));

while(scan.hasNextLine()){

String line = scan.nextLine();

String[] values = line.split(",");

statement.setString(1, values[0]);

statement.setString(2, values[1]);

statement.setString(3, values[2]);

statement.setString(4, values[3]);

statement.setString(5, values[4]);

statement.executeUpdate();

}

conn.close();

}

}

通过多次运行上面的程序比较，会发现使用PreparedStatement对象插入100条记录所花费的时间比使用Statement插入100记录所使用的时间要少很多，这表明PreparedStatement比Statement的执行效率要高。

除此之外，使用PreparedStatement的另外一个优势就是：当SQL语句中要使用参数时，无须拼接SQL字符串，而使用Statement则需要拼接SQL字符串。如果要拼接的参数有多个时，并且涉及到SQL中的参数为字符串时，需要使用单引号括起来，非常容易出错，而且还会导致SQL注入。而使用PreparedStatement则只需要使用问号占位符来代替这些参数即可，减少编程的复杂度。

使用PreparedStatement还有另外一个好处就是防止SQL注入。

如下示例所示：使用Statement时的SQL注入

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

import java.util.Scanner;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

Scanner scan = new Scanner(System.in);

System.out.println("请输入用名：");

String name = scan.nextLine();

System.out.println("请输入密码：");

String pass = scan.nextLine();

String sql = "select \* from student where user\_name='"+name+"' and user\_pass='"+pass+"'";

Statement statement = conn.createStatement();

boolean flag = statement.execute(sql);

if(flag){

System.out.println("登录成功！");

}

conn.close();

}

}

如果在程序运行时输入：

用户名：‘ or true or ‘ 密码即使不输入也能登录成功，这是因为SQL一旦拼接后变成了如下所示的SQL“

select \* from student where user\_name=’’ or true or ‘’ and stud\_pass=’’;

看到此SQL语句大家就不难理解为何会登录成功了，这是因为true在SQL中是当成常量来使用，而or是关键字。而使用PreparedStatement就不会出现这样的情况了，因此使用PreparedStatement比使用Statement有如下三个好处：

* PreparedStatement预编译SQL，性能更好
* PreparedStatement无须拼接SQL，编程更简单
* PreparedStatement可以防止SQL注入，安全性更好。

基于以上3点，因此推荐使用PreparedStatement来执行SQL语句，而不使用Statement来执行SQL语句。

#### 14.4.4 使用CallableStatement调用存储过程

调用存储过程需要使用CallableStatement对象，可以通过Connection的prepareCall()的方法来创建一个CallableStatement对象。创建该对象时，需要传入调用该存储过程的SQL语句。调用存储过程的语句的语法格式为：{call 存储过程名(?,?,?...)}，其中问号作为存储过程参数的占位符。如下代码所示：

String sql = "{call query\_sub\_action(?,?,?)}";

CallableStatement call = conn.prepareCall(sql);

存储过程的参数既有传入参数，又有传出参数。所谓的传入参数就是Java必须为这些参数传入值，可以通过CallableStatement对象的setXXX()方法为传入参数设置值，所谓的传出参数就是Java可以通过参数获取存储过程里的值，CallableStatement需要调用call.registerOutParameter(int parameterIndex, int sqlType)方法来注册该参数。

如下代码所示：

call.registerOutParameter(3, Types.VARCHAR);

经过上面的步骤之后，就可以调用CallableStatement对象的execute()方法来执行存储过程了，执行结束后就可以通过CallableStatement对象的getXXX(int index)方法来获取指定传出参数的值。如下程序示范了如何调用存储过程：

package com.langsin.jdbc;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

**String sql = "{call query\_sub\_action(?,?,?)}";**

**CallableStatement call = conn.prepareCall(sql);**

**call.setString(1, "张三");**

**call.setString(2, "CD003");**

**call.setInt(3, 5);**

**call.execute();**

conn.close();

}

}

### 14.5 管理结果集

JDBC使用ResultSet来封装执行查询得到的查询结果，然后通过移动ResultSet的记录指针来取出结果集的内容。除此之外，JDBC还允许通过ResultSet来更新记录，并提供了ResultSetMetaData来获取ResultSet对象的相关信息。

#### 14.5.1 可滚动、可更新的结果集

ResultSet定位记录指针的方法有absolute()、previous()等方法，但前面程序自始至终都只用了next()方法来移动指针，实际上也可以使用absolute()、previous()、last()等方法来移动记录指针。可以使用absolute()、previous()、afterLast()等方法自由移动记录指针的ResultSet被称为可滚动的结果集。

以默认方式打开的ResultSet是不可更新的，如果希望创建可更新的ResultSet，则必须在创建Statement或PreparedStatement时传入额外的参数。Connection在创建Statement或PreparedStatement时还可额外传入如下两个参数。

1. **resultSetType：**控制ResultSet的类型，该参数可以取得如下三个值

* **ResultSet.TYPE\_FORWARD\_ONLY**：该常量控制记录指针只能**向前移动**。这是JDK1.4版本以前的默认值。
* **ResultSet.TYPE\_SCROLL\_INSENSITIVE**：该常量控制记录指针可以**自由移动**，但底层数据的改变不会影响ResultSet的内容。
* **ResultSet.TYPE\_SCROLL\_SENSITIVE**：该常量控制记录指针可以**自由移动**，而且底层数据的改变会影响ResultSet的内容。

注意：ResultSet.TYPE\_SCROLL\_INSENSITIVE和ResultSet.TYPE\_SCROLL\_SENSITIVE两个常量的作用需要底层数据库驱动的支持，对于某些数据库驱动来说，这两个常量没有太大的区别。

1. **resultSetConcurrency：**控制Result的并发类型，该参数可以接收如下两个值。

* **ResultSet. CONCUR\_READ\_ONLY**：该常量指示ResultSet是只读的并发模式。
* **ResultSet.CONCUR\_UPDATABLE**：该常量指示ResultSet是可更新的并发模式。

下面代码通过这两个参数创建一个PreparedStatement对象，由该对象生成的ResultSet对象就是可滚动、可更新的结果。如下所示：

statement = conn.preparedStatement(sql,Result.TYPE\_SCROLL\_INSENSITIVE,ResultSet.CONCUR\_UPDATABLE);

注意：可更新的结果集还需要满足如下两个条件。

1）、所有数据都应该来自一个表

2）、所选出的数据集必须包含主键列。

通过该PreparedStatement对象创建的ResultSet就是可滚动、可更新的，程序可调用Result的updateXXX(int columIndex,XXX value)方法来修改记录指针所指的记录、特定列的值，最后调用ResultSet的updateRow()方法来提交修改。

Java 8为ResultSet添加了updateObject(String columnLabel,Object x,SQLType targetSqlType)和updateObject(int columnIndex,Object x,SQLType targetSqlType)两个默认方法，这两个方法可以直接用Object来修改记录指针所指记录、特定列的值，其中SQLType用于指定该数据列的类型。目前最新的MySQL驱动不支持该方法。

示例：

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

public class CreateView {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "select \* from student";

**PreparedStatement ps = conn.prepareStatement(sql, ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE); //可滚动、可更新的**

**ResultSet set = ps.executeQuery();**

**set.last();**

**int count = set.getRow();**

**for(int i=count;i>0;i--){**

**set.absolute(i);**

**set.updateString("stud\_cname", "张三"+i);**

**set.updateRow();**

**}**

conn.close();

}

}

#### 14.5.2 处理Blob类型数据

Blob（Binary Long Object）是二进制长对象的意思，Blob列通常用于存储大文件，典型的Blob内容是一张图片或一个声音文件，由于他们的特殊性，必须使用特殊的方式来存储。使用Blob列可以把图片、声音等文件的二进制数据保存到数据库里，并可以从数据库里恢复指定文件。

如果需要将图片插入到数据库，显然不能直接通过普通的SQL语句来完成，因为有一个关键的问题——Blob常量无法表示。所以将Blob数据插入数据库需要使用PreparedStatement，该对象有一个方法：setBinaryStream(int parameterIndex,InputStream x)，该方法可以为指定参数传入二进制输入流，从而可以实现将Blob数据保存到数据库的功能。

当我们需要从ResultSet里取出Blob数据时，可以调用ResultSet的getBlob(int columnIndex)，该方法将返回一个Blob对象，Blob对象提供了getBinaryStream()方法来获取该Blob数据的输入流，也可以使用Blob对象提供的getByte()方法直接取出Blob对象封装的二进制数据。

如下示例所示：将一张图片存储到数据库中，然后从数据库中取得该图片。

重点代码：//将字节流，通过PreparedStatement对象的setBinaryStream方法将指定的参数为二进制流

ResultSet set = ps.executeQuery();

//绝对定位到结果集的第几行

set.absolute(1);

Blob data = set.getBlob("img");

//通过Blob对象的getBytes方法将数据转换为byte数值，然后通过ImageIcon类转换为图片对象

//getBytes的两个参数，long points，int length ，

//1、从二进制那个位置开始读取数据，2、要读取的文件的长度

ImageIcon image = new ImageIcon(data.getBytes(1, (int)data.length()));

Image img = image.getImage();

示例代码：

package com.langsin.jdbc;

import java.awt.BorderLayout;

import java.awt.Button;

import java.awt.Canvas;

import java.awt.FileDialog;

import java.awt.Frame;

import java.awt.Graphics;

import java.awt.Image;

import java.awt.Panel;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.WindowAdapter;

import java.awt.event.WindowEvent;

import java.awt.image.BufferedImage;

import java.io.File;

import java.io.FileInputStream;

import java.io.InputStream;

import java.sql.Blob;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import javax.imageio.ImageIO;

import javax.swing.ImageIcon;

public class BlobTest {

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

final Frame frame = new Frame("Blob字段示例");

frame.setBounds(100, 100, 500, 500);

Panel panel = new Panel();

Button btn1 = new Button("插入图片");

Button btn2 = new Button("读取图片");

panel.add(btn1);

panel.add(btn2);

frame.add(panel,BorderLayout.NORTH);

final BufferedImage image = new BufferedImage(500, 400, BufferedImage.TYPE\_INT\_RGB);

final Graphics grap = image.getGraphics();

//默认在画布上画一张图片

Image img = ImageIO.read(new File("./image/Jellyfish.jpg"));

grap.drawImage(img, 0, 0, 500, 400, null);

class MyCanvas extends Canvas{

@Override

public void paint(Graphics g) {

g.drawImage(image, 0, 0, null);

}

}

final MyCanvas can = new MyCanvas();

frame.add(can);

//为插入图片按钮添加事件，指定一张图片存入到数据库中

btn1.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

FileDialog dialog = new FileDialog(frame, "保存文件", FileDialog.LOAD);

dialog.setBounds(100, 100, 400, 400);

dialog.setVisible(true);

String path = dialog.getDirectory();

String name = dialog.getFile();

File file = new File(path+name);

try {

//将文件读入到IO字节流中

InputStream is = new FileInputStream(file);

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

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

PreparedStatement ps = conn.prepareStatement(sql);

//将字节流，通过PreparedStatement对象的setBinaryStream方法将指定的参数为二进制流

ps.setBinaryStream(1, is, file.length());

ps.executeUpdate();

conn.close();

} catch (Exception e1) {

e1.printStackTrace();

}

}

});

btn2.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

try {

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "select \* from image";

PreparedStatement ps = conn.prepareStatement(sql);

ResultSet set = ps.executeQuery();

//绝对定位到结果集的第几行

set.absolute(1);

Blob data = set.getBlob("img");

conn.close();

//通过Blob对象的getBytes方法将数据转换为byte数值，然后通过ImageIcon类转换为图片对象

//getBytes的两个参数，long points，int length ，1、从二进制那个位置开始读取数据，2、要读取的文件的长度

ImageIcon image = new ImageIcon(data.getBytes(1, (int)data.length()));

Image img = image.getImage();

grap.drawImage(img, 0, 0, 500, 400, null);

can.repaint();

} catch (Exception e2) {

}

}

});

frame.addWindowListener(new WindowAdapter() {

@Override

public void windowClosing(WindowEvent e) {

frame.dispose();

}

});

frame.setVisible(true);

}

}

#### 14.5.3 使用ResultSetMetaData分析结果集

当执行SQL查询后可以通过移动记录指针来遍历ResultSet的每条记录，但是程序可能不知道该ResultSet结果集中包含了哪些数据列，以及每个数据列的数据类型。那么可以通过ResultSetMetaData来获取关于ResultSet的描述信息。

MetaData是元数据的意思，即描述其他数据的数据。因此ResultSetMetaData封装了描述ResultSet对象的数据；后面还会介绍的DatabaseMetaData则封装了关于Database的相关数据。

ResultSet提供了一个getMetaData()的方法，该方法返回了该ResultSet对应的ResultMetaData对象。一旦获得了ResultSetMetaData对象，就可以通过ResultSetMetaData对象提供的大量的方法，来获取相关ResultSet的描述信息，常用的方法有如下几个：

1. int getColumnCount()：返回该ResultSet的列数量。
2. String getColumnName(int columnIndex)：返回指定索引的列名。
3. int getColumnType(int column)：返回指定索引的列类型。

示例：根据查询语句将结果集在SQL客户端展示的样例在控制台输出出来，列名是通过ResultSetMetaData对象获得。

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.ResultSetMetaData;

public class ResultSetMetaDataTest {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "select stud\_name,stud\_pass,stud\_cname,stud\_address from student";

PreparedStatement ps = conn.prepareStatement(sql, ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_READ\_ONLY);

ResultSet rs = ps.executeQuery();

//获取ResultSet的元数据对象

ResultSetMetaData rsmd = rs.getMetaData();

int columnNum = rsmd.getColumnCount();

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

System.out.print(rsmd.getColumnName(i)+" ");

}

System.out.println();

rs.last();

int rowCount = rs.getRow();

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

rs.absolute(i);

for(int j=1;j<=columnNum;j++){

System.out.print(rs.getString(j)+" ");

}

System.out.println();

}

conn.close();

}

}

注意：虽然ResultSetMetaData可以准确的分析出ResultSet中包含多少列，以及每列的列名，数据类型等，但是使用ResultSetMetaData需要一定的系统开销，因此在编程的过程中已经知道了ResultSet里包含多少列，以及每列的列名、每列数据类型等信息，就没有必要使用ResultSetMetaData来分析ResultSet对象了。

### 14.6 Java7的RowSet

RowSet接口继承了ResultSet接口，RowSet接口下包含了JdbcRowSet、CachedRowSet、FilteredRowSet、JoinRowSet和WebRowSet常用子接口。除了JdbcRowSet需要保持与数据库的连接外，其余4个子接口都是离线的RowSet，无须保持与数据库的连接。

与ResultSet相比，RowSet默认是可滚动、可更新、可序列化的结果集，而且作为JavaBean的使用，RowSet可以很方便的在网络中传输，用于同步两端的数据。对于离线的RowSet而言，程序在创建RowSet时已经把数据从底层数据库读取到了内存，因此可以充分利用计算机的内存，从而降低数据库服务器的负载，提高程序性能。

#### 14.6.1 离线RowSet

在使用ResultSet的时代，程序查询得到ResultSet之后必须立即读取或处理它对应的记录，否则一旦Connection关闭，再去通过ResultSet读取记录就会引发异常。在这种模式下JDBC编程十分繁琐——假设应用程序架构被分为两层：数据库访问层和视图显示层，当应用程序在数据库访问层查询得到ResultSet之后，对ResultSet的处理有如下两种常见方式：

* 使用迭代访问ResultSet里的记录，并将这些记录转换成JavaBean，再将多个JavaBean封装成一个List集合对象，也就是完成“ResultSet🡪Java Bean集合”的转换。转换完成后可以关闭Connection资源，然后将JavaBean集合传到视图层显示层，视图显示层可以显示查询得到的数据。
* 直接将ResultSet传到视图层——这就要求当视图层显示数据时，底层Connection一直处于打开状态，否则ResultSet无法读取记录。

第一种方式比较安全，但是编程方式比较繁琐；第二种方式则要求Connection一直处于打开状态，这不仅不安全，而且对程序性能有很大影响。

通过使用离线RowSet可以十分“优雅”的解决上面问题，离线RowSet会直接将底层数据读入到内存中，封装成RowSet对象，而且RowSet对象完成可以当成JavaBean来使用。因此不仅安全，而且编程十分简单。CachedRowSet是所有离线RowSet的父接口，因此下面以CachedRowSet为例进行介绍。

示例1：

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import javax.sql.rowset.CachedRowSet;

import javax.sql.rowset.RowSetFactory;

import javax.sql.rowset.RowSetProvider;

public class ResultSetMetaDataTest {

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

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

String sql = "select stud\_id,stud\_name,stud\_pass,stud\_cname,stud\_address from student";

PreparedStatement ps = conn.prepareStatement(sql);

ResultSet rs = ps.executeQuery();

//创建离线RowSet工厂对象，此对象通过RowSet供应商对象RowSetProvider创建

RowSetFactory factory = RowSetProvider.newFactory();

//创建缓存CachedRowset对象

CachedRowSet crs = factory.createCachedRowSet();

//将查询结果集ResultSet对象封装到RowSet对象中

crs.populate(rs);

conn.close();

//将光标移动到集合的最后一行

crs.last();

while(crs.previous()){

System.out.println(crs.getString(1)+"\t"+crs.getString(2)+"\t"+crs.getString(3));

if(crs.getString("stud\_cname").equals("张三1")){

//查到昵称为这个的更改昵称，此时无法更新的数据库中，需要重新连接提交结果集

crs.updateString("stud\_cname", "孙悟空");

crs.updateRow();

}

}

//重新获取连接

conn = DriverManager.getConnection(url, user, password);

//将连接的自动提交关闭，默认为是自动提交

conn.setAutoCommit(false);

crs.acceptChanges(conn);

conn.commit();

conn.close();

}

}

通过程序的运行，可以看到在Connection关闭的情况下，程序依然可以读取、修改RowSet里的记录。程序通过acceptChanges()方法把对离线结果集所做的修改同步到底层数据库，调用该方法时需要串入一个连接对象。

#### 14.6.2 离线RowSet的查询分页

由于CachedRowSet会将数据记录直接装载到内存中，因此如果SQL查询返回的结果集过大，CachedRowSet将会占用大量的内存，在某些极端的情况下，它甚至会直接导致内存溢出。

为了解决该问题，CachedRowSet提供了分页功能。所谓的分页功能就是CachedRowSet一次只能装载ResultSet里某几条记录，这样就可以解决CachedRowSet占用内存过大的问题。

CachedRowSet提供了如下几个方法来控制分页：

* populate(ResultSet rs,int startRow)：使用给定的ResultSet填装RowSet，从ResultSet的第startRow行开始填装。
* setPageSize(int pageSize)：设置CachedRowSet每次返回多少条记录。
* previousPage()：在底层ResultSet可用的情况下，让CachedRowSet读取上一页记录。
* nextPage()：在底层ResultSet可用的情况下，让CachedRowSet读取下一页记录。

示例：

package com.langsin.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import javax.sql.rowset.CachedRowSet;

import javax.sql.rowset.RowSetFactory;

import javax.sql.rowset.RowSetProvider;

public class CachedRowSetTest {

public CachedRowSet query(String sql,int pageSize,int page) throws Exception{

Class.forName("com.mysql.jdbc.Driver");

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

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

PreparedStatement ps = conn.prepareStatement(sql);

ResultSet set = ps.executeQuery();

RowSetFactory factory = RowSetProvider.newFactory();

CachedRowSet crs = factory.createCachedRowSet();

crs.setPageSize(pageSize);

crs.populate(set, (page-1)\*pageSize+1);

conn.close();

return crs;

}

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

String sql = "select \* from student";

CachedRowSet crs = new CachedRowSetTest().query(sql, 6, 1);

while(crs.next()){

System.out.println(crs.getString(1)+"\t"+crs.getString(2)+"\t"+crs.getString(3));

}

}

}

### 14.7 事务处理

对于任何数据库而言，事务都是非常重要的，事务是保证底层数据完整的重要手段，没有事务支持的数据库应用将是非常脆弱的。

#### 14.7.1 事务的概念和MySQL的事务支持

事务我们之前已经简单的介绍了一点，下面再来回顾一下事务的内容。事务是由一步或几步数据库操作序列组成的逻辑单元。这系列操作要么全部执行，要么全部放弃执行。程序和事务是两个不同的概念，一般一个程序中可能会包含多个事务。

当事务所包含的全部数据库操作都成功执行后，应该提交事务，使这些修改永久生效。事务的提交方式有两种：显示提交和自动提交。

* 显示提交：使用commit关键字。
* 自动提交：执行DDL或DML语句后，或者程序正常退出。

当事务所包含的任意一个数据库操作执行失败后，应该回滚事务，使该事务所做的修改全部失效。事务的回滚方式有两种：显示回滚和自动回滚。

* 显示回滚：使用rollback关键字
* 自动回滚：系统错误或者强行退出。

MySQL是默认关闭事务的，在默认情况下，用户在MySQL的控制台输入一条DML语句，这条语句所做的更改会立刻保存到数据库里。为了开启MySQL的事务支持，可以调用如下命令：

set autocommit=[0|1] 0表示关闭自动提交，表示开启事务。1表示自动提交，关闭事务

一旦在MySQL的命令行窗口输入set autocommit=0开启了事务，该命令行窗口中所执行的所有的DML语句都不会立刻生效，上一个事务结束后的第一个DML语句将开始新的一个事务，而后续的所有的SQL语句都处于这个事务中，除非显示使用commit提交事务，或者程序正常退出，或者执行DDL或DCL语句导致事务隐式的提交。当然也可以使用rollback回滚事务，使用rollback结束事务将导致本次事务中的DML语句所做的修改全部失效。

#### 14.7.2 JDBC的事务支持

JDBC连接也提供了事务支持，JDBC连接的事务支持由Connection提供，Connection默认打开自动提交，即关闭事务。在这种情况下，每条SQL语句一旦执行，便会立即提交到数据库，永久生效，无法对其进行回滚操作。

可以调用Connection的setAutoCommit(boolean auto)方法来关闭自动提交，开始事务，如下代码所示：

conn.setAutoCommit(false);

一旦事务开始后，程序可以像平常所写代码一样创建Statement对象，创建Statement对象后，可以执行任意多条DML语句。如下所示：

statement.executeUpdate(…….);

statement.executeUpdate(…….);

statement.executeUpdate(…….);

如上SQL语句虽然被执行了，但这些SQL语句所做的修改不会立刻生效，因为事务还没有结束。如果所有的SQL语句都执行成功了，程序可以调用Connection的commit()方法来提交事务，如下代码所示：

conn.commit();

如果任意一条SQL执行失败，则应该调用Connection的rollback()方法回滚事务，如下代码所示：

conn.rollback();

示例1：

package com.langsin.jdbc;

import java.io.File;

import java.io.FileNotFoundException;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.util.Scanner;

public class TransactionTest {

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

try {

Class.forName("com.mysql.jdbc.Driver");

} catch (Exception e) {

e.printStackTrace();

}

String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";

String user = "root";

String password = "123456";

Connection conn = null;

try {

conn = DriverManager.getConnection(url, user, password);

Scanner scan = new Scanner(new File("./sql.txt"));

//关闭自动提交，开启事务

conn.setAutoCommit(false);

while(scan.hasNextLine()){

String sql = scan.nextLine();

PreparedStatement statement = conn.prepareStatement(sql);

statement.executeUpdate();

}

conn.commit();

conn.close();

} catch (SQLException e) {

try {

conn.rollback();

} catch (SQLException e1) {

e1.printStackTrace();

}

}

}

}

### 14.8 使用连接池管理连接

数据库连接的建立以及关闭是极其耗费系统资源的操作，在多层结构的应用环境中，这种资源的耗费对系统性能影响尤为明显。通过前面章节中所提到的使用DriverManager获得的数据库连接，一个数据库连接对象对应一个物理数据库连接，每次操作都打开一个物理连接，使用完成后立即关闭连接。频繁地打开、关闭连接将造成系统性能的降低。

数据库连接池的解决方案是：当应用程序启动时，系统主动建立足够的数据库连接，并将这些连接组成一个连接池。每次应用程序请求数据库连接时，无须重新打开连接，而是从连接池中取出已有的连接使用，使用完成后不再关闭数据库连接，而是直接将连接归还给连接池。通过使用连接池，将大大提高程序的运行效率。

对于共享资源的情况，有一个通用的设计模式：资源池（Resource Pool），用于解决资源的频繁请求、释放所造成的性能下降。为了解决数据库连接的频繁请求、释放。JDBC2.0规范引入了数据库连接池技术，数据库连接池是Connection对象的工厂。数据库连接池的常用参数如下：

* 数据库的初始连接数
* 连接池的最大连接数
* 连接池的最小连接数
* 连接池每次增加的容量

JDBC的数据库连接池使用DataSource来表示，DataSource只是一个接口，该接口通常由商用服务器，例如WebLogic或者WebSphere等提供实现，也有一些开源组织提供实现，例如DBCP和C3P0等

#### 14.8.1 DBCP数据源

DBCP是Apache软件基金组织下的开源连接池实现，该连接池依赖该组织下的另外一个开源系统：common-pool。如果需要使用该连接池实现，则应该在系统中增加如下两个jar文件：

1. common-dbcp.jar：连接池的实现
2. common-pool.jar：连接池的依赖库

登录<http://commons.apache.org/> 站点即可下载common-dbcp.zip和common-pool.zip两个压缩文件，解压这两个文件即可得到上面所说的两个jar文件。

Tomcat的连接池正是采用该连接池实现的。数据库连接池既可以与应用服务器整合使用，也可以有应用程序独立使用。下面代码示范了如何使用DBCP来获得数据库连接的方式。

package com.langsin.jdbc;

import java.sql.Connection;

import org.apache.commons.dbcp2.BasicDataSource;

public class DbcpConnection {

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

//创建数据源对象

BasicDataSource dataSource = new BasicDataSource();

//设置连接池所需要的驱动

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

//设置连接数据库的URL

dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8");

//设置连接数据库的用户名

dataSource.setUsername("root");

//设置连接数据库的密码

dataSource.setPassword("123456");

//设置连接池的初始化连接数

dataSource.setInitialSize(5);

//设置连接池最多有多少个活动连接数

dataSource.setMaxTotal(20);

//设置连接池少有几个空闲的连接

dataSource.setMinIdle(2);

//获取连接

Connection conn = dataSource.getConnection();

System.out.println(conn);

}

}

数据源和数据库连接不同，数据源无须创建多个，它是产生数据库连接的工厂，因为整个应用只需要一个数据源即可。也就是说，对于一个应用，上面的代码只要执行一次即可。建议把上面程序中的dataSource设置static成员变量，并且在应用开始时立即初始化数据源对象，程序中所有需要获取连接数据库连接的地方直接访问该dataSource对象，并获取数据库连接即可。通过DataSource获取数据库连接的代码示例如上所示：

当程序访问结束后，程序还是想以前一样关闭数据库连接，如上所示，只不过并没有关闭数据库的物理连接，它仅仅是把数据库连接释放，归还给连接池而已，可以让其他用户继续使用该连接。

#### 14.8.2 C3P0数据源

相比之下，C3P0数据源性能更胜一筹，Hibernate就推荐使用该连接池。C3P0连接池不仅可以自动清理不再使用的Connection，还可以自动清理Statement和ResultSet。C3P0连接池需要版本为1.3以上的JRE，推荐使用1.3以上，如果需要使用C3P0连接池，则应该在系统中增加如下JAR文件：

c3p0-0.9.1.2.jar：C3P0连接池的实现包。

登录<http://mvnrepository.com/>站点，即可下载我们所需要的所有的第三方的Jar包，在查询框中输入C3P0即可，得到我们所需要的依赖，进项下载。

通过下面代码即可通过C3P0连接池获取数据库连接。如下示例所示：

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

//创建数据库连接池实例对象

ComboPooledDataSource cpds = new ComboPooledDataSource();

//设置数据库连接所需要的驱动

cpds.setDriverClass("com.mysql.jdbc.Driver");

//设置连接数据库的URL

cpds.setJdbcUrl("jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8");

//设置连接数据库的用户名

cpds.setUser("root");

//设置连接数据库的密码

cpds.setPassword("123456");

//设置连接池的最大连接数

cpds.setMaxPoolSize(200);

//设置连接池的最小连接数

cpds.setMinPoolSize(5);

//设置连接池的初始连接数

cpds.setInitialPoolSize(10);

//设置连接池的缓存Statement的最大数

cpds.setMaxStatements(180);

//获取数据库连接

Connection conn= cpds.getConnection();

//下面就是JDBC编程了

System.out.println(conn);

conn.close();

}

### 14.9 在Tomcat中配置数据库连接池

#### 14.9.1 JNDI概念

JNDI（java naming and directory interface）, )是SUN公司提供的一种标准的Java命名服务和目录服务的接口，开发者可以通过它来进行使用命名和目录服务，就像JDBC一样。JNDI包括命名服务和目录服务两部分。其中目录服务包含目录对象directory object，它包含若干属性对象，提供了对属性的很多操作。

#### 14.9.2 命名和目录服务

命名和目录服务我们一直在使用，例如操作系统的文件系统。它给我们提供对文件的操作，查询、添加、删除等；NDS服务系统将URL与IP地址绑定在一起等。命名和服务系统最重要的是将name和对象绑定在一起。在它的基础之上还提供了更多的功能，例如根据名称查找对象的lookup,search等功能，而且存储的对象是有一定层次结构的。使用这样的服务可以让我们对对象进行更加有效的管理和操作。

命名服务将一个名称映射到一个对象上。

目录服务也存放对象，但目录服务识别这些对象的属性。

J2SE为JNDI提供了5个扩展包：

1、 javax.naming：为访问命名服务提供的api

2、 javax.naming.directory：为访问目录服务提供的api

3、 javax.naming.event：支持命名和目录服务中事件通知。

4、 javax.naming.ldap：支持ldap的包

5、 javax.naming.spi：提供了不同命名和目录服务可以挂接他们的实现方法。

Context：是一套name-to-object的绑定，可以理解为层次或目录，它还可以包括下一层subContext。在使用时创建InititalContext对象。Context是对整个命名空间操作的入口。在目录服务中使用DriContext。

#### 14.9.3 DBCP数据库连接池

下面我们介绍如果使用JNDI实现数据库连接池的操作：

首先更改配置文件：

在tomcat目录下的conf目录下，打开context.xml文件，在<Context>标签中加入如下标签

<Resource name="java/webdb" 数据源名称

auth="Container"

type="javax.sql.DataSource" 连接的数据源类型

driverClassName="com.mysql.jdbc.Driver" 数据库驱动类

url="jdbc:mysql://localhost:3306/test?useUnicode=true;charactorEncoding=UTF-8;" URL

username="root" 用户名

password="123456" 密码

maxActive="200" 最大连接数

maxIdle="50" 最大空闲连接数

maxWait="3000" 在请求连接，且没有空闲连接时，最长等待时间

/>

然后在我们的web应用中引用该数据库资源，在WEB-INF目录下，打开web.xml文件，在web-app标签下加入如下标签

<resource-ref>

<description>mysql数据库连接池</description>

<res-ref-name>java/webdb</res-ref-name>

<res-type>java.sql.DataSource</res-type>

<res-auth>Container</res-auth>

<res-sharing-scope>Shareable</res-sharing-scope>

</resource-ref>

这样在我们的应用中就配置好了数据库连接池，接下来就是如何使用的问题了：

//初始化JNDI上下文对象

Context ctx = new InitialContext();

//通过lookup查找我们的数据库连接池的名称所在的目录,“java:/comp/env”为固定目录

ctx = (Context)ctx.lookup("java:/comp/env");

//找到目录后，查找我们定义的数据源对象，仍然使用lookup查找，通过名称java/webdb

DataSource source = (DataSource)ctx.lookup("java/webdb");

//然后根据数据库连接池，在里面拿到一个空闲连接

Connection conn = source.getConnection();

//剩下的就是普通的JDBC编程了

System.out.print(conn);

conn.close();

#### 14.9.4 C3P0数据库连接池

首先更改Context.xml文件，在Resource标签中，属性与DBCP有所不同，如下所示：

<Resource name="java/webdb" auth="Container"

type="com.mchange.v2.c3p0.ComboPooledDataSource" 数据库连接池的数据源类型

factory="org.apache.naming.factory.BeanFactory" 数据源连接池的javaBean工厂对象

driverClass="com.mysql.jdbc.Driver" 驱动类

jdbcUrl="jdbc:mysql://localhost:3306/test?useUnicode=true"

user="root"

password="123456"

minPoolSize="5" 最小连接数

maxPoolSize="200" 最大连接数

initialPoolSize="10" 初始化连接数

maxStatements="0" 缓存的sql执行对象的个数

/>

同样，在我们的web应用中引用该数据库资源，在WEB-INF目录下，打开web.xml文件，在web-app标签下加入如下标签

<resource-ref>

<description>mysql数据库连接池</description>

<res-ref-name>java/webdb</res-ref-name>

<res-type>java.sql.DataSource</res-type>

<res-auth>Container</res-auth>

<res-sharing-scope>Shareable</res-sharing-scope>

</resource-ref>