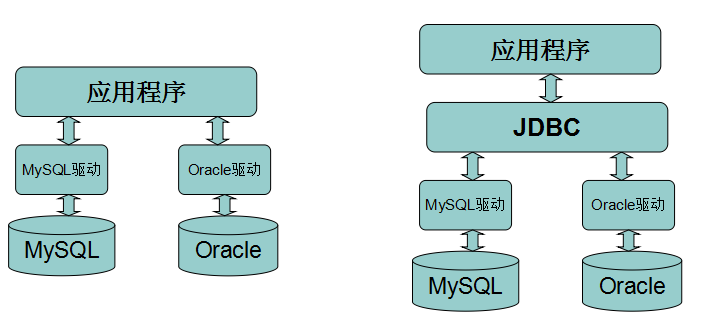
# 7.JDBC

## 7.1.JDBC概述

数据库驱动：数据库厂商提供的用来操作数据库的jar包

1.2.JDBC简介

由于各大数据库厂商提供的数据库驱动各不相同, 导致了开发人员的学习成本十分的高. SUN公司为了简化数据库的操作, 提供了一套规范, 本质上就是一大堆的接口, 要求各大数据库厂商在提供驱动时都要实现JDBC这套接口, 实现之后, 只要学会JDBC这套接口, 所有的数据库驱动就都会使用了!



JDBC由两个包组成, 分别是java.sql和javax.sql, 目前已经被集成到javase规范中!

需要注意的是: JDBC中包含的就是一段接口, 真实操作数据库的代码都在具体的数据库驱动中. 也就是说在开发数据库程序时, 除了要导入JDBC相关的包之外, 还需要导入具体的数据库驱动包.

## 7.2.实现JDBC程序

**在开发之前，首先要导入驱动jar包**

|  |
| --- |
| Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  //1.注册数据库驱动  Class.forName("com.mysql.jdbc.Driver");  //2.获取数据库连接  conn = DriverManager.getConnection("jdbc:mysql:///mydb5", "root", "root");  //3.获取传输器  stat = conn.createStatement();  //4.利用传输器，发送sql到数据库执行，返回执行结果  rs = stat.executeQuery("select \* from account");  //5.处理结果  while(rs.next()){  int id = rs.getInt(1);  String name = rs.getString("name");  double money = rs.getDouble("money");  System.out.println(id+name+money);  }  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  //6.释放资源  if(rs!=null){  try {  rs.close();  } catch (SQLException e) {  e.printStackTrace();  }finally{  rs = null;  }  }  if(stat!=null){  try {  stat.close();  } catch (SQLException e) {  e.printStackTrace();  }finally{  stat = null;  }  }  if(conn!=null){  try {  conn.close();  } catch (SQLException e) {  e.printStackTrace();  }finally{  conn = null;  }  }  } |

## 7.3.JDBC API详解

注册数据库驱动：

使用DriverManager.registerDriver(new Driver());注册数据库有两个缺点，首先，通过观察mysql的中Driver接口的实现类发现在静态代码块中注册驱动的逻辑，所以这种方式会造成驱动被注册两次。另外，这种方式导致了程序和具体的数据库驱动绑死在了一起，程序的灵活性比较低。

所以推荐使用：Class.forName(“com.mysql.jdbc.Driver”);的方式注册数据库驱动。

获取数据库连接

Connection conn = DriverManager.getConnection(url,name,psw);

数据库URL

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

jdbc:mysql://localhost:3306/test ?参数名=参数值

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:jdbc:mysql://localhost:3306/sid

Mysql的url地址的简写形式： jdbc:mysql:///sid

Connection

Jdbc程序中的Connection，它用于代表数据库的链接，Connection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PreparedSatement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

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) ：用于向数据库发送查询语句。

executeUpdate(String sql)：用于向数据库发送insert、update或delete语句

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

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：

getString(int index)

getString(String columnName)

getInt(columnIndex)

getInt(columnLabel)

getDouble(columnIndex)

getDouble(columnLabel)

...

操作游标的方法,例如:

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。

## 7.4.JDBC的增删改查

增：

|  |
| --- |
| Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  //1.注册驱动  Class.forName("com.mysql.jdbc.Driver");  //2.获取连接  conn = DriverManager.getConnection("jdbc:mysql:///mydb5","root","root");  //3.获取传输器  stat = conn.createStatement();  //4.利用传输器，发送sql，返回结果  String sql = "insert into account value(null,'张飞',888)";  int rows = stat.executeUpdate(sql);  //5.处理结果  System.out.println("影响了"+rows+"行");  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  //6.释放资源  if(rs!=null){  try {  rs.close();  } catch (Exception e) {  e.printStackTrace();  }finally{  rs = null;  }  }  if(stat!=null){  try {  stat.close();  } catch (Exception e) {  e.printStackTrace();  }finally{  stat = null;  }  }  if(conn!=null){  try {  conn.close();  } catch (Exception e) {  e.printStackTrace();  }finally{  conn = null;  }  }  } |

改

|  |
| --- |
| 1. **创建JDBCUtils类，并添加如下代码：**   public static void close(Connection conn,Statement stat,ResultSet rs){  if(rs!=null){  try {  rs.close();  } catch (Exception e) {  e.printStackTrace();  }finally{  rs = null;  }  }  if(stat!=null){  try {  stat.close();  } catch (Exception e) {  e.printStackTrace();  }finally{  stat = null;  }  }  if(conn!=null){  try {  conn.close();  } catch (Exception e) {  e.printStackTrace();  }finally{  conn = null;  }  }  }   1. **改的代码变为：**   Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  Class.forName("com.mysql.jdbc.Driver");  conn = DriverManager.getConnection("jdbc:mysql:///mydb5","root","root");  stat = conn.createStatement();  int rows = stat.executeUpdate("update account set money=666 where name='张飞'");  System.out.println("影响了"+rows+"行");  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  //释放资源  JDBCUtils.close(conn, stat, rs);  } |

删

|  |
| --- |
| 1. **在src下创建config.properties配置文件，并添加如下内容：**   driverClass=com.mysql.jdbc.Driver  jdbcUrl=jdbc:mysql:///mydb5  user=root  password=root   1. **在JDBCUtils中添加如下代码：**   private static Properties prop = new Properties();    static{  //读取配置文件  try {  //String path = "bin/config.properties";  //通过类加载器获取路径  ClassLoader loader = JDBCUtils.class.getClassLoader();  String path = loader.getResource("config.properties").getPath();  prop.load(new FileInputStream(path));  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }  }    public static Connection getConnection(){  Connection conn;  try {  String driverClass = prop.getProperty("driverClass");  String jdbcUrl = prop.getProperty("jdbcUrl");  String user = prop.getProperty("user");  String password = prop.getProperty("password");    Class.forName(driverClass);  conn = DriverManager.getConnection(jdbcUrl,user,password);  return conn;  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }  }   1. **删除代码变为：**   Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  conn = JDBCUtils.getConnection();  stat = conn.createStatement();  int rows = stat.executeUpdate("delete from account where name='张飞'");  System.out.println("影响了"+rows+"行");  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  //释放资源  JDBCUtils.close(conn, stat, rs);  } |

## 7.5.PreparedStatement

Sql注入攻击演示

|  |
| --- |
| public static void main(String[] args) {  Scanner scan = new Scanner(System.in);  //提示用户登陆  System.out.println("请登陆");  //提示输入用户名  System.out.println("请输入用户名:");  String username = scan.nextLine();  //提示输入密码  System.out.println("请输入密码:");  String password = scan.nextLine();    login(username,password);  }  private static void login(String username, String password) {  Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  conn = JDBCUtils.getConnection();  stat = conn.createStatement();  String sql = "select \* from user where username='"+username+"' and password='"+password+"'";  //当输入“张三'#”后，sql变为：  //select \* from user where username='张三'#' and password=''  //当输入“张三' or '2=2”后，sql变为：  //select \* from user where username='张三' or '2=2' and password=''  rs = stat.executeQuery(sql);  if(rs.next()){  System.out.println("恭喜，登陆成功");  }else{  System.out.println("用户名或密码错误");  }  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  JDBCUtils.close(conn, stat, rs);  }  } |

由于后台的SQL是拼接而来的, 其中的参数是用户提交的, 如果用户在提交参数时, 参杂了一些SQL关键字或者特殊符号, 就有可能会导致SQL语句语意的改变, 从而造成一些意外的操作!

PreparedStatement

PreparedStatement优点:

（1）可以防止sql注入攻击

通过PreparedStatement对象发送sql, 是先把sql语句的骨架发送给数据库编译并确定下来, 后面发送的只能是参数的值, 不能影响sql语句的骨架, 即使参数中包含sql关键字或特殊符号, 也只会当成普通的文本来处理!

（2）通过方法来设置参数, 省去了拼接SQL语句的麻烦!

（3）可以提高程序的效率:

PreparedStatement对象发送的sql语句(骨架)到数据库编译后会被数据缓存下来, 如果下次执行的sql与缓存中的相匹配, 就不再编译而是直接使用缓存中的语句, 可以减少sql语句编译的次数, 提高程序执行的效率!

Statement对象发送的sql语句到数据库之后也会编译, 但是Statement对象是先拼接好再发送sql到数据库, 如果每次参数不同, 整条sql也就不同. 所以每次都需要编译!

代码改造：

|  |
| --- |
| private static void login(String username, String password) {  Connection conn = null;  PreparedStatement ps = null;  ResultSet rs = null;  try {  conn = JDBCUtils.getConnection();  String sql = "select \* from user where username=? and password=?";  ps = conn.prepareStatement(sql);  ps.setString(1, username);  ps.setString(2, password);  rs = ps.executeQuery();    if(rs.next()){  System.out.println("恭喜，登陆成功");  }else{  System.out.println("用户名或密码错误");  }  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  JDBCUtils.close(conn, ps, rs);  }  } |

## 7.6.批处理

批处理概述

假设现有一大堆的sql要到数据库执行, 如果一条一条发送, 有多少条就 需要发送多少次, 效率低下

可以通过批处理提高发送sql语句的效率: 可以将这一大堆的sql添加到 一个批中, 一次性将批发送给数据库, 数据库收到后打开批, 依次执行其中 sql语句, 这样可以减少sql语句发送的次数, 从而提高程序执行的效率!

Statement方式来实现批处理

**优点:**

可以在一次批处理中添加结构不同的sql语句

**缺点:**

不能防止sql注入攻击

没有预编译机制, 效率略低

当发送结构相同的sql语句时, sql语句的骨架每次都需要编写。

**例如：**

|  |
| --- |
| public static void main(String[] args) {  Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  conn = JDBCUtils.getConnection();  stat = conn.createStatement();  stat.addBatch("use mydb1");  stat.addBatch("create table tb\_batch(id int primary key auto\_increment, name varchar(20))");  stat.addBatch("insert into tb\_batch values(null,'a')");  stat.addBatch("insert into tb\_batch values(null,'b')");  stat.addBatch("insert into tb\_batch values(null,'c')");  stat.addBatch("insert into tb\_batch values(null,'d')");    stat.executeBatch();  System.out.println("执行完成");  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  JDBCUtils.close(conn, stat, rs);  }  } |

PreparedStatement方式实现批处理

**优点:**

可以防止sql注入攻击

采用预编译机制, 可以提高程序执行的效率

当发送多条结构相同的sql时, sql语句的骨架可以只发送一次。

**缺点:**

不能在一次批处理中添加结构不同的sql语句

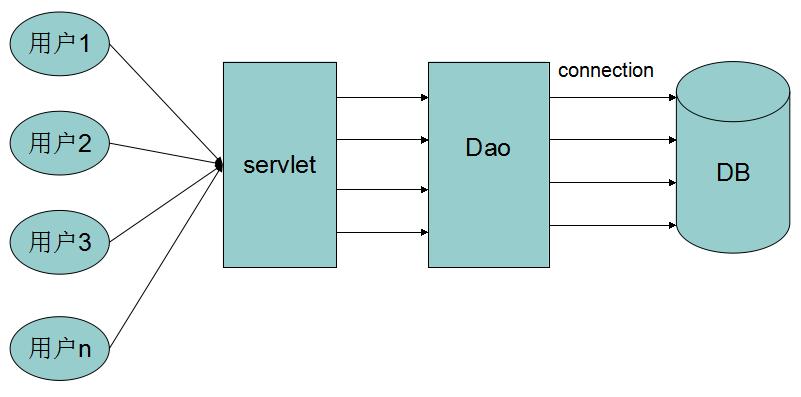
**例如：**

|  |
| --- |
| public static void main(String[] args) {  Connection conn = null;  PreparedStatement ps = null;  ResultSet rs = null;  try {  Long a = System.currentTimeMillis();  conn = JDBCUtils.getConnection();  ps = conn.prepareStatement("insert into tb\_batch values(null,?)");  for(int i=0;i<100;i++){  ps.setString(1,"b1");  ps.addBatch();  }  ps.executeBatch();  System.out.println("执行完成");  Long b = System.currentTimeMillis();  System.out.println(b-a);  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  JDBCUtils.close(conn, ps, rs);  }  } |

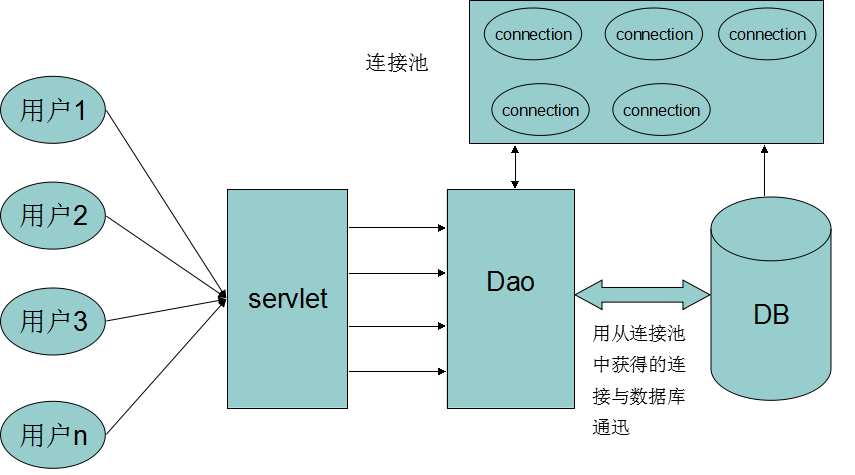
## 7.7.连接池(重要!!!)

### 7.7.1.连接池概述

用户每次请求都需要向数据库获得链接，而数据库创建连接通常需要消耗相对较大的资源，创建时间也较长。假设网站一天10万访问量，数据库服务器就需要创建10万次连接，极大的浪费数据库的资源，并且极易造成数据库服务器内存溢出、宕机。



频繁的开关连接相当的耗费资源,所以我们可以设置一个连接池,在程序启动时就初始化一批连接,在程序中共享,需要连接时从池中获取,用完再还回池中,通过池共享连接,减少开关连接的次数,提高程序的效率。



### 7.7.2.自定义连接池

|  |
| --- |
| **自定义连接池**  package cn.tedu.pool;  import java.io.PrintWriter;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.sql.SQLFeatureNotSupportedException;  import java.util.LinkedList;  import java.util.List;  import java.util.logging.Logger;  import javax.sql.DataSource;  public class MyPool implements DataSource{  private static List<Connection> list = new LinkedList<Connection>();  static{  try {  Class.forName("com.mysql.jdbc.Driver");  for (int i = 0; i < 5; i++) {  Connection conn = DriverManager.getConnection("jdbc:mysql:///mydb5","root","root");  list.add(conn);  }  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }  }    @Override  public Connection getConnection() throws SQLException {  if(list.isEmpty()){  for (int i = 0; i < 3; i++) {  Connection conn = DriverManager.getConnection("jdbc:mysql:///mydb1");  list.add(conn);  }  }  Connection conn = list.remove(0);  System.out.println("成功从数据库中获取一个连接，连接池中还剩"+list.size()+"个连接..");  return conn;  }  public void returnConn(Connection conn){  try {  if(conn != null && !conn.isClosed()){  list.add(conn);  System.out.println("成功向数据库还回一个连接，连接池中还剩"+list.size()+"个连接..");  }  } catch (SQLException e) {  e.printStackTrace();  throw new RuntimeException();  }  }        @Override  public PrintWriter getLogWriter() throws SQLException {  // TODO Auto-generated method stub  return null;  }  @Override  public void setLogWriter(PrintWriter out) throws SQLException {  // TODO Auto-generated method stub  }  @Override  public void setLoginTimeout(int seconds) throws SQLException {  // TODO Auto-generated method stub  }  @Override  public int getLoginTimeout() throws SQLException {  // TODO Auto-generated method stub  return 0;  }  @Override  public Logger getParentLogger() throws SQLFeatureNotSupportedException {  // TODO Auto-generated method stub  return null;  }  @Override  public <T> T unwrap(Class<T> iface) throws SQLException {  // TODO Auto-generated method stub  return null;  }  @Override  public boolean isWrapperFor(Class<?> iface) throws SQLException {  // TODO Auto-generated method stub  return false;  }  @Override  public Connection getConnection(String username, String password)  throws SQLException {  // TODO Auto-generated method stub  return null;  }  }  **测试自定义连接池**  package cn.tedu.pool;  import java.sql.Connection;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  public class TestMyPool {  public static void main(String[] args) {  MyPool pool = new MyPool();  Connection conn= null;  Statement stat = null;  ResultSet rs = null;  try {  conn = pool.getConnection();  stat = conn.createStatement();  rs = stat.executeQuery("select \* from account where id=1");  if(rs.next()){  System.out.println(rs.getInt("id")+":"+rs.getString("name")+":"+rs.getDouble("money"));  }  System.out.println("执行完毕");  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  if(rs!=null){  try {  rs.close();  } catch (SQLException e) {  e.printStackTrace();  }finally{  rs = null;  }  }  if(stat!=null){  try {  stat.close();  } catch (SQLException e) {  e.printStackTrace();  }finally{  stat = null;  }  }  pool.returnConn(conn);  }  }  } |

如上代码写的连接池，还需要在使用完连接后记得不能关闭连接，而是要调用returnConn方法将连接还回池中。

我们想能不能想办法改造conn的close方法，使close方法不会真的关闭连接而是将连接还回池中。

### 7.7.3.close方法的改造

继承

写一个类继承要改造的类，对于不想改造的方法不覆盖，对于想要改造的方法复写该方法，将代码改造为自己需要的逻辑代码。

这种方式只能在还没有对象的情况下使用，现在Connection对象已经存在了，再用继承复写的方式是不行的，所以我们不采用。

装饰

实现装饰设计模式:

（1）写一个装饰类, 要求装饰类和被装饰者所属的类实现同一个接口或者继承同一个父类

（2）装饰类必须提供构造方法接收被装饰者, 并将被装饰者保存在类的内部

（3）对于想要改造的方法直接进行改造, 对于不想改造的方法, 直接调用原有对象(被装设者)上的方法

**代码改造：**

|  |
| --- |
| **创建装饰者类ConnectionDecorate：只改造close方法。（不想改造的方法，右键Source->Generate Delegate Methods，选中不想改造的即可)**  public class ConnectionDecorate implements Connection{  private Connection conn = null;  private MyPool pool = null;  public ConnectionDecorate(Connection conn,MyPool pool){  this.conn = conn;  this.pool = pool;  }    @Override  public void close() throws SQLException {  pool.returnConn(conn);  }  **修改MyPool类：在获取连接方法中，return前加入如下代码：**  //先进行包装  Connection connDecorate = new ConnectionDecorate(conn,this);  return connDecorate;  **测试类最后只需关闭连接即可** |

### 7.7.4.开源数据库连接池c3p0

c3p0概述

我们手写的连接池是比较简陋的，是为了讲解连接池的原理。其实在真实开发中可以使用开源的数据库连接池。其中C3P0是比较常用的一种。

C3P0连接池应用

**(1)导入c3p0的jar包：c3p0-0.9.1.2.jar**

**(2)创建测试类TestC3p0，并添加如下代码：**

|  |
| --- |
| public static void main(String[] args) {  Connection conn = null;  Statement stat = null;  ResultSet rs = null;  try {  //创建连接池  ComboPooledDataSource cpds = new ComboPooledDataSource();  //设置连接数据库的基本信息  cpds.setDriverClass("com.mysql.jdbc.Driver");  cpds.setJdbcUrl("jdbc:mysql:///mydb5");  cpds.setUser("root");  cpds.setPassword("root");  //从连接池中获取连接  conn = cpds.getConnection();  stat = conn.createStatement();  rs = stat.executeQuery("select \* from account where id=1");  if(rs.next()){  System.out.println(rs.getInt("id")+" : "  +rs.getString("name")+" : "  +rs.getDouble("money"));  }  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException();  }finally{  JDBCUtils.close(conn, stat, rs);  }  } |

**(3)但是发现此种方式将数据写死在代码中，所以需要进行改造（推荐！！！）**

C3p0会默认读取一个配置文件，为c3p0-config.xml，（注意，名字不能改变，否则c3p0将无法读取）我们在src或者类似的源码目录下，创建一个c3p0-config.xml文件, 配置内容如下:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <c3p0-config>  <default-config>  <property name="driverClass">com.mysql.jdbc.Driver</property>  <property name="jdbcUrl">jdbc:mysql:///mydb5</property>  <property name="user">root</property>  <property name="password">root</property>  </default-config>  </c3p0-config> |

(4)另一种方式：在src或者类似的源码目录下, 创建一个c3p0.properties文件, 配置内容如下:

|  |
| --- |
| c3p0.driverClass=com.mysql.jdbc.Driver  c3p0.jdbcUrl=jdbc:mysql:///mydb5  c3p0.user=root  c3p0.password=root |

## 7.8.作业练习

1. jdbc 连接数据库过程

答：（可以写详细点）（1）注册数据库驱动

1. 获取数据库连接
2. 获取传输器
3. 利用传输器，发送sql到数据库执行，返回执行结果
4. 处理结果
5. 释放资源
6. 如何防止SQL注入攻击呢？

答：利用PreparedStatement可以防止注入攻击，通过PreparedStatement对象发送sql, 是先把sql语句的骨架发送给数据库编译并确定下来, 后面发送的只能是参数的值, 不能影响sql语句的骨架, 即使参数中包含sql关键字或特殊符号, 也只会当成普通的文本来处理!

1. 连接池的作用？

答： 建立数据库连接是相当耗时和耗费资源的，如果Web应用程序为每一个客户请求分配一个数据库连接，将导致性能的急剧下降。连接池技术预先建立多个数据库连接对象，然后将连接对象保存到连接池中，当客户请求到来时，从池中取出一个连接对象为客户服务，当请求完成后，客户程序调用close（）方法，将连接对象放回池中。这样省略了创建连接和销毁连接的过程，性能上得到了提高。

1. 说说装饰模式？

答：装饰模式指的是在不改变原类和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

实现装饰设计模式:

（1）写一个装饰类, 要求装饰类和被装饰者所属的类实现同一个接口或者继承同一个父类

（2）装饰类必须提供构造方法接收被装饰者, 并将被装饰者保存在类的内部

（3）对于想要改造的方法直接进行改造, 对于不想改造的方法, 直接调用原有对象(被装设者)上的方法