



# 第 12 章 JDBC 详解

控制台，图形界面客户端与数据库交互的途径是执行 SQL 语句，Java 程序也不例外。Java 中访问数据库主要使用 JDBC。JDBC 是 Java 规定的访问数据库的标准 API，目前主流的数据库都支持 JDBC。

使用 JDBC 访问 MySQL 需要 MySQL 的驱动。MySQL 的 JDBC 驱动可以从官方网站下载。本书随书光盘中附带 MySQL 5 的 JDBC 驱动 mysql-connector-java-5.0.5-bin.jar。

## 12.1 JDBC 简介

JDBC（Java 数据基础连接，Java Database Connectivity）是标准的 Java 访问数据库的 API。JDBC 定义了数据库的连接，SQL 语句的执行以及查询结果集的遍历等。JDBC 把这些操作定义为接口，位于包 `java.sql` 下面。如 `java.sql.Connection`、`java.sql.Statement`、`java.sql.ResultSet` 等。各个数据库提供商在自己的 JDBC 驱动中实现了这些接口。

### 12.1.1 查询实例：列出人员信息

下面在实例中演示 JDBC 工作的过程。先创建一个数据库、一个表并插入几条数据。在 MySQL 控制台中批量执行.sql 文件 `init.sql`。`init.sql` 的内容如下：

代码 12.1 `init.sql`

```
DROP DATABASE IF EXISTS databaseWeb;      -- 如果存在，则删除数据库 databaseWeb
CREATE DATABASE databaseWeb CHARACTER SET utf8; -- 创建数据库，使用 utf8 编码

USE databaseWeb;      -- 切换到数据库 databaseWeb，以下操作均在 databaseWeb 下

set NAMES 'gbk';          -- 控制台使用 gbk 编码

DROP TABLE IF EXISTS tb_person;           -- 如果存在，删除表 tb_person
CREATE TABLE tb_person (
    id INTEGER AUTO_INCREMENT COMMENT 'id',
    name VARCHAR(45) COMMENT '姓名',
    english_name VARCHAR(45) COMMENT '英文名',
    age INTEGER UNSIGNED COMMENT '年龄',
    sex VARCHAR(45) COMMENT '性别',
    birthday DATE COMMENT '出生日期',
    description TEXT COMMENT '备注',
    create_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP() COMMENT '创建时间',
```

```
PRIMARY KEY (id)
);

INSERT INTO tb_person          -- 插入一条数据
( name, english_name, age, sex, birthday, description ) values
( '刘京华', 'Helloweenvsfei', '25', '男', '1982-08-09', '无备注' );

INSERT INTO tb_person          -- 插入一条数据
( name, english_name, age, sex, birthday, description ) values
( '科特柯本', 'Kurt Cobain', '27', '男', '1967-02-20', 'Nirvana' );

INSERT INTO tb_person          -- 插入一条数据
( name, english_name, age, sex, birthday, description ) values
( '李四', 'Faye', '31', '女', '1969-08-08', '狮子座' );

INSERT INTO tb_person          -- 插入一条数据
( name, english_name, age, sex, birthday, description ) values
( '张三', 'Foo Bar', '18', '女', '2008-08-08', '' );
```

然后新建 Web 项目 databaseWeb，并把 MySQL 的 JDBC 驱动添加到项目的CLASSPATH 中。新建 JSP 文件 listPerson.jsp，源代码如下：

代码 12.2 listPerson.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ page import="java.sql.DriverManager"%>
<%@ page import="java.sql.Connection"%>
<%@ page import="java.sql.Statement"%>
<%@ page import="java.sql.ResultSet"%>
<jsp:directive.page import="java.sql.Date" />
<jsp:directive.page import="java.sql.Timestamp" />
<jsp:directive.page import="java.sql.SQLException"/>

<a href="addPerson.jsp">新建人员资料</a>
<%
    Connection conn = null;           // 数据库连接
    Statement stmt = null;             // Statement
    ResultSet rs = null;               // 结果集

    try{
        DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        // 注册驱动

        // 获取数据库连接。三个参数分别为连接 URL、用户名、密码
        conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/databaseWeb", "root",
            "admin");

        // 获取 Statement。Statement 对象用于执行 SQL，相当于控制台
        stmt = conn.createStatement();

        // 使用 Statement 执行 SELECT 语句，返回结果集
        rs = stmt.executeQuery("select * from tb_person");
    }
    <form action="operatePerson.jsp" method="get">
        <table bgcolor="#CCCCCC" cellspacing=1 cellpadding=5 width=
        100%>
            <tr bgcolor="#DDDDDD">
```



```
<th></th>
<th>ID</th>
<th>姓名</th>
<th>英文名</th>
<th>性别</th>
<th>年龄</th>
<th>生日</th>
<th>备注</th>
<th>记录创建时间</th>
<th>操作</th>
</tr>
<%
    // 遍历结果集。rs.next() 返回结果集中是否还有下一条记录。如果有，自动滚动到下一条记录并返回 true
    while (rs.next()) {
        int id = rs.getInt("id");           // 取 id 列，整型类型
        int age = rs.getInt("age");         // 取 age 列

        String name = rs.getString("name");
        // 取 name 列，字符串类型
        String englishName = rs.getString("english_name");
        String sex = rs.getString("sex");
        String description = rs.getString("description");

        Date birthday = rs.getDate("birthday"); // 日期类型
        Timestamp createTime = rs.getTimestamp("create_time"); // 时间戳

        out.println("<tr bgcolor=#FFFFFF>");
        out.println("  <td><input type=checkbox name=id value=" + id + "></td>");
        out.println("  <td>" + id + "</td>");
        out.println("  <td>" + name + "</td>");
        out.println("  <td>" + englishName + "</td>");
        out.println("  <td>" + sex + "</td>");
        out.println("  <td>" + age + "</td>");
        out.println("  <td>" + birthday + "</td>");
        out.println("  <td>" + description + "</td>");
        out.println("  <td>" + createTime + "</td>");
        out.println("  <td>");
        out.println("    <a href='operatePerson.jsp?action=del&id=" + id + "' onclick='return confirm(\"确定删除该记录?\")'>删除</a>");
        out.println("    <a href='operatePerson.jsp?action=edit&id=" + id + "'>修改</a>");
        out.println("  </td>");
        out.println("  </tr>");
    }
%>
</table>
<table align=left>
<tr>
    <td>
        <input type='hidden' value='del' name='action'> <a href='#' onclick="var array=document.getElementsByName('id');for(var i=0; i<array.length; i++){array[i].checked=true;}"> 全选 </a> <a href='#'>
    </td>

```

```
onclick="var array=document.getElementsByName('id');for(var i=0; i<array.length; i++){array[i].checked=false;}">取消全选 </a> <input type='submit' onclick="return confirm('即将删除所选择的记录。是否删除？'); " value='删除'>
        </td>
    </tr>
</table>
</form>
<%
}catch(SQLException e){
    out.println("发生了异常: " + e.getMessage());
    e.printStackTrace();
}finally{
    if(rs != null)                                // 关闭 rs
        rs.close();
    if(stmt != null)                               // 关闭 stmt
        stmt.close();
    if(conn != null)                               // 关闭 conn
        conn.close();
}
%>
```

listPerson.jsp 演示了 JDBC 访问数据库的一般步骤：注册驱动，获取连接，获取 Statement，执行 SQL 并返回结果集，遍历结果集显示数据，释放连接。首先注册了 MySQL 的数据库驱动类，并从 DriverManager 中获取一个 Connection。然后使用 Connection 创建一个 Statement，Statement 相当于 MySQL 的控制台，能够执行 SQL 语句，并返回相应地结果。executeQuery()方法返回的是 ResultSet 对象，遍历该结果集便可得到查询的数据。

listPerson.jsp 效果如图 12.1。



图 12.1 listPerson.jsp 运行效果

**注意：**ResultSet、Statement、Connection 对象使用完毕后均需要手工关闭。如果不关闭，会占用数据库连接，可能导致后面的程序连不上数据库。跟 java.io.\*里的对象一样，最好将关闭代码写到 try...finally 代码块里。

### 12.1.2 各种数据库的连接

由于使用了 JDBC，各种数据库获取连接的方式是相同的，只是不同的数据库的连接

URL 不同。常见数据库的连接 URL 见表 12.1。

表 12.1 常见数据库的连接URL

数 据 库	连 接 URL	说 明
MySQL	jdbc:mysql://localhost:3306/ <b>db</b>	默认端口为 3306。粗体为连接时使用的数据库模式。下同
Oracle	jdbc:oracle:thin:@localhost:1521: <b>db</b>	默认端口为 1521
DB2	jdbc:db2://localhost:6789/ <b>db</b>	默认端口为 6789
PostgreSQL	jdbc:postgresql://localhost:5432/ <b>db</b>	默认端口为 5432
Sysbase	jdbc:jtds:sybase://localhost:2638/ <b>db</b>	默认端口为 2638
SQLServer	jdbc:microsoft:sqlserver://localhost:1433;databaseName= <b>db</b>	默认端口为 1433
SQLServer 2005	jdbc:sqlserver://localhost:1433;databaseName= <b>db</b>	默认端口为 1433

## 12.2 MySQL 的乱码解决

如果没有出现乱码问题，那么恭喜读者。严格执行上面的 init.sql 的话，是不会遇到乱码问题的。如果出现了乱码问题也不要急，乱码问题是处理中文字符时常遇到的问题，常常使初学者感到束手无策。

### 12.2.1 MySQL 的乱码解决

MySQL 支持几十种编码方式，并且默认的编码 latin1（一种西方字符编码方式）对中文支持不太好，因此需要设置 MySQL 的编码方式。如果没有设置或者设置不对很容易出现中文乱码。

常用的中文编码方式有 GB2312、GBK、GB18030、UTF-8 等。GB2312 是针对中文的编码方式，每个汉字均占两个字节，解析比较简单，但是仅能编码中文字符。中文网站往往采用 GB2312 编码，例如 BAIDU。GBK 类似于 GB2312，但是比 GB2312 编码更多的中文以及中文字符。GB18030 又比 GBK 广泛，不仅可以编码中文，还可以编码少数民族的语言。

UTF-8 能够编码目前为止的任意语言，国际化的网站往往采用 UTF-8 编码，例如 Google。但是 UTF-8 编码比较浪费空间，有的汉字编码后能占到三个字节，解析也比较复杂。本书中一般采用 UTF-8 编码。设置 MySQL 编码方式有下面 3 种途径。

### 12.2.2 从控制台修改编码

从控制台输入命令。注意 MySQL 中 UTF-8 写做 utf8，例如：

```
mysql> ALTER DATABASE databaseWeb CHARACTER SET utf8;  
Query OK, 1 row affected (0.00 sec)
```

查看当前数据库编码方式端的命令为：

```
mysql> show variables like 'character_set_database';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| character_set_database | utf8  |
+-----+-----+
1 row in set (0.10 sec)
```

注意修改数据库编码不会影响已经存在的表的编码方式。如果表的编码方式原来为 latin1，还需要单独修改表的编码方式。修改某个数据表的编码方式的 SQL 为：

```
ALTER TABLE table_name CHARACTER SET utf8;
```

也可以在创建数据库的时候指定编码方式，例如：

```
CREATE DATABASE some_database CHARACTER SET utf8;
```

### 12.2.3 从配置文件修改编码

修改配置文件。用记事本打开 MySQL 目录下的 my.ini 文件（笔者机器上文件路径为 C:\Program Files\MySQL\MySQL Server 5.0\my.ini），找到下面一句话：

```
default-character-set=latin1
```

应该有两行这样的代码。将编码方式 latin1 都修改为 utf8 即可。注意修改该参数只会影响以后创建的数据库、表，但不会影响已经存在的数据库、表。

### 12.2.4 利用图形界面工具修改

利用 MySQL 自带的图形界面工具（MySQL GUI Tools）修改。利用图形界面可以很方便地修改各种参数。图形界面工具需单独从 MySQL 网站下载。在图形界面中双击表名，在弹出的对话框中选择 Table Option 属性页，如图 12.2 所示。

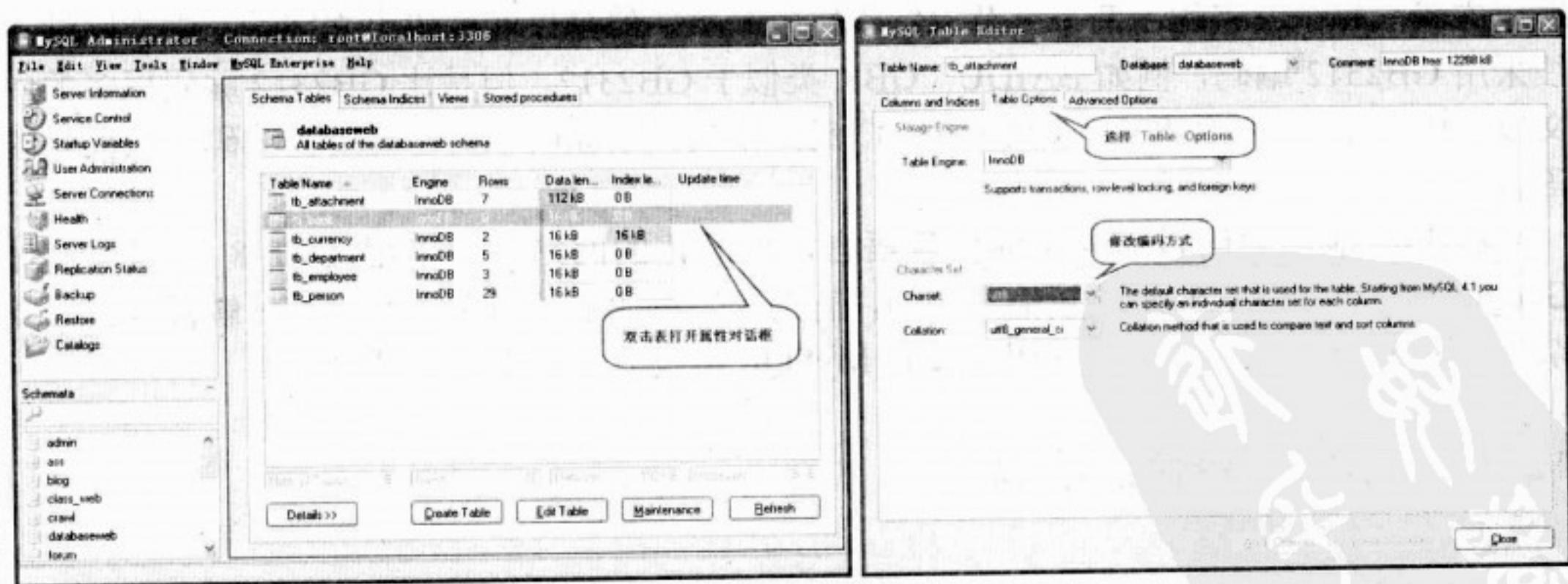


图 12.2 修改编码方式

**注意：**由于修改数据库编码不会影响到已经存在的表的编码，因此还要修改表的编码。

### 12.2.5 URL 中指定编码方式

另外，还需要指定 JDBC 连接的编码方式，方法是修改连接 URL，例如：

```
Connection conn = DriverManager.getConnection(  
    "jdbc:mysql://localhost:3306/databaseWeb?unicode=true&characterEncoding=UTF-8",  
    "root", "admin");
```

做了上述设定后，一般就没有乱码了。如果还有乱码，请检查是否对 request 进行了编码。如果没有，从 request 取到的字符串也会乱码。注意 JSP 页面编码、request 编码、response 编码、数据库编码必须一致。

Tomcat 的 server.xml 中也要修改，为 GET 方式获取数据添加 URIEncoding 参数，指定为 UTF-8 编码（默认为 ISO-8859-1 编码），代码为：

```
<Connector port="8080" protocol="HTTP/1.1"  
    connectionTimeout="20000"  
    redirectPort="8443" URIEncoding="UTF-8"/>
```

 提示：MySQL 数据库编码要做到数据库用 UTF-8 编码；Filter 中要对 request、response 进行 UTF-8 编码；JDBC 配置中 URL 要指定 UTF-8 编码，这样就不会出现乱码了。

## 12.3 JDBC 基本操作：CRUD

数据库程序常被称为 CRUD 程序，因为它包括数据的创建 Create、读取查询 Read、更新 Update、删除 Delete 等操作，取首字母缩写便是 CRUD。CRUD 概括了数据库的程序结构，程序无论大小，归根结底都是这 4 种操作。本节将介绍利用 JDBC 实现 CRUD 的全部操作。

### 12.3.1 查询数据库

前面介绍了一个查询数据库的例子。查询数据库遵循固定的流程：注册 MySQL 驱动、获取 Connection、创建 Statement、查询数据库返回 ResultSet 对象、遍历 ResultSet 输出数据、关闭 ResultSet、关闭 Statement、关闭 Connection。示例代码在前面已经讲解了，这里不再介绍。

### 12.3.2 插入人员信息

Java 程序也可以执行 INSERT 语句往数据库插入数据，方法仍然是使用 Statement 对象，不过执行 INSERT 语句时要使用 executeUpdate(String sql) 方法而不是 executeQuery(String sql) 方法。executeUpdate() 方法用于执行 INSERT、UPDATE、DELETE 等，返回数据库中

影响的行数，返回 int 类型。而 executeQuery()方法用于执行 SELECT，返回查询到的结果集，返回 ResultSet 类型。

listPerson.jsp 中也预留了“新建人员资料”的链接，下面来补齐这个空缺。新建人员资料的页面 addPerson.jsp 是个 FORM 表单，将客户填写的数据提交给 operatePerson.jsp 处理。添加数据与修改数据使用的是同一个表单，见后面 12.3.5 节的修改数据。数据处理页面源代码如下：

代码 12.3 operatePerson.jsp

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.sql.DriverManager" %>
<%@ page import="java.sql.Connection" %>
<%@ page import="java.sql.Statement" %>
<%
    public String forSQL(String sql){ // MySQL 的 SQL 中的单引号(')需要转化为 \\
        return sql.replace("'", "\\'");
    }
%>
<%
    request.setCharacterEncoding("UTF-8"); // 设置 request 编码
    String name = request.getParameter("name"); // 获取 name 参数
    String englishName = request.getParameter("englishName");
    // 获取 englishName 参数
    String age = request.getParameter("age"); // 获取 age 参数
    String birthday = request.getParameter("birthday");
    // 获取 birthday 参数
    String sex = request.getParameter("sex"); // 获取 sex 参数
    String description = request.getParameter("description");
    // 获取 description 参数
    String action = request.getParameter("action");// 获取 action 参数
    if("add".equals(action)){
        String sql = "INSERT INTO tb_person " + // INSERT SQL 语句
            "( name, english_name, age, sex, birthday, description )"
            " values " +
            " ( '" + forSQL(name) + "', '" + forSQL(englishName)
            + "', '" +
            " '" + age + "', '" + sex + "', '" + birthday +
            "', '" + forSQL(description) + "' ) ";
        Connection conn = null; // Connection 对象
        Statement stmt = null; // Statement 对象
        int result = 0; // 执行结果（影响了几条数据）
        try{
            DriverManager.registerDriver(new com.mysql.jdbc.Driver());
            // 注册 MySQL 驱动
            conn = DriverManager.getConnection( // 创建 Connection
                "jdbc:mysql://localhost:3306/databaseWeb?characterEncoding=UTF-8",
                "root", "admin");
            stmt = conn.createStatement(); // 创建 Statement
            result = stmt.executeUpdate(sql);
            // 执行 Insert SQL 语句，返回影响行数
        }
    }
%>
```

## 第 12 章 JDBC 详解

```
    }catch(SQLException e){
        out.println("执行 SQL\""+sql+"\"时发生异常：" + e.getMessage());
        return;
    }finally{
        if(stmt != null)    stmt.close();           // 关闭 stmt
        if(conn != null)   conn.close();           // 关闭 conn
    }

    out.println("<html><style>body{font-size:12px; line-height:25px; }</style><body>");
    out.println(result + " 条记录被添加到数据库中。");
    out.println("<a href='listPerson.jsp'>返回人员列表</a>");
    out.println("<br/><br/>执行的 SQL 语句为: <br/>" + sql); // 输出 SQL
    return;
}
```

代码 result=stmt.executeUpdate(sql);是关键的程序代码。表单页面 addPerson.jsp 中使用了一个 Javascript 日历控件。读者可以在随书光盘源代码中找到该文件。提交人员信息前后效果如图 12.3 所示。添加之后 listPerson.jsp 中就列出新添加的人员信息了。

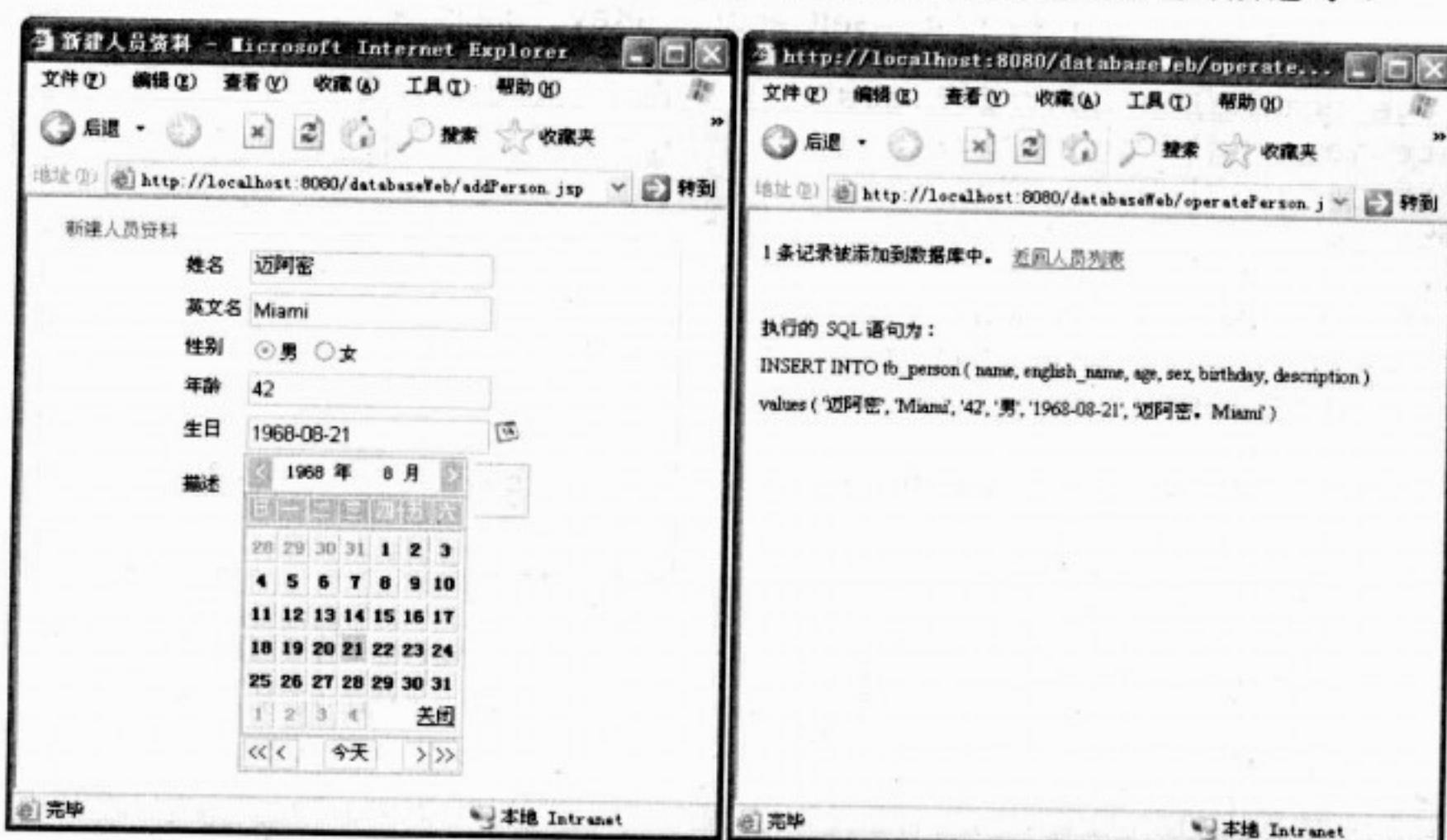


图 12.3 “新建人员信息”效果

打印出的 SQL 语句可能为：

```
INSERT INTO tb_person ( name, english_name, age, sex, birthday, description )
values ( '李四', 'Li Si', '30', '男', '2008-02-13', '籍贯：北京' )
```

**提示：**由于表 tb\_person 的 id 列是自增长类型的，因此 INSERT 语句中没有显式地插入 id 值。MySQL 数据库会自动为该列插入一个值。

### 12.3.3 注册数据库驱动

代码中注册驱动使用的是显示的注册，直接调用 DriverManager 的方法注册：

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

对于 MySQL 来说，还可以使用下面的任一种方式：

```
new com.mysql.jdbc.Driver();
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

MySQL 驱动的构造函数中会调用 DriverManager 的注册方法。

#### 12.3.4 获取自动插入的 ID

在数据库编程中，自增长类型使用起来非常方便。开发者不需要编程维护该列的值，例如查找当前值、计算下一个值等，而由数据库自己负责维护。开发者不需要关注具体细节。但有时候却需要知道数据库自动插入了什么值。例如，插入了一条人员信息后，需要再插入一条该人员的其他信息，这时就要求知道该人员信息的 id。

Statement 提供 getGeneratedKeys()方法以 ResultSet 方式返回所有自动生成的键值，遍历该 ResultSet 对象便可得到插入行的 id。示例代码如下：

代码 12.4 GetGeneratedKeysTest.java

```
package com.helloweenvsfei.test;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import com.mysql.jdbc.Driver;
public class GetGeneratedKeysTest{

    public static void main(String[] args) throws SQLException {
        new Driver();                                // 注册驱动
        Connection conn = null;                      // Connection 对象
        Statement stmt = null;                        // Statement 对象
        ResultSet rs = null;                          // ResultSet 对象
        try {
            conn = DriverManager.getConnection(      // 创建 Connection
                "jdbc:mysql://localhost:3306/databaseWeb?characterEncoding=UTF-8",
                "root", "admin");
            stmt = conn.createStatement();           // 获取 Statement
            stmt.executeUpdate("insert into tb_person " // 执行 INSERT 语句
                + " (name, english_name, age, "
                + " sex, birthday, description) "
                + " values ('Name', 'English Name', "
                + " '17', '男', current_date(), '')");
            rs = stmt.getGeneratedKeys();           // 获取自动生成的键值
            rs.next();                            // 滚动到下一条
            System.out.println("id: " + rs.getInt(1)); // 输出第一列
        } finally {
            if (rs != null)    rs.close();          // 关闭 rs
            if (stmt != null)   stmt.close();         // 关闭 stmt
            if (conn != null)  conn.close();          // 关闭 conn
        }
    }
}
```

```
}
```

这是个使用了 JDBC 的 Java Application 程序，与 JSP 中使用 JDBC 是一样的。运行结果会输出新增加的行的自增长主键值。

```
id: 26
```

### 12.3.5 删除人员信息

删除数据使用 Statement 的 executeUpdate(String sql)方法执行 DELETE 语句。与 INSERT 不同的是，DELETE 必须使用 WHERE 条件指定删除哪一行数据，否则将删除所有数据。对于有主键的表来说，可以使用主键来标识哪一行数据，因为主键值是唯一的，不可重复的。例如 tb\_person 表中的 id 列。

前面 listPerson.jsp 中已经预留了删除数据的接口，并且支持多行删除。多行删除的原理是将这些行的 ID 一块提交给程序。运行效果如图 12.4 所示。

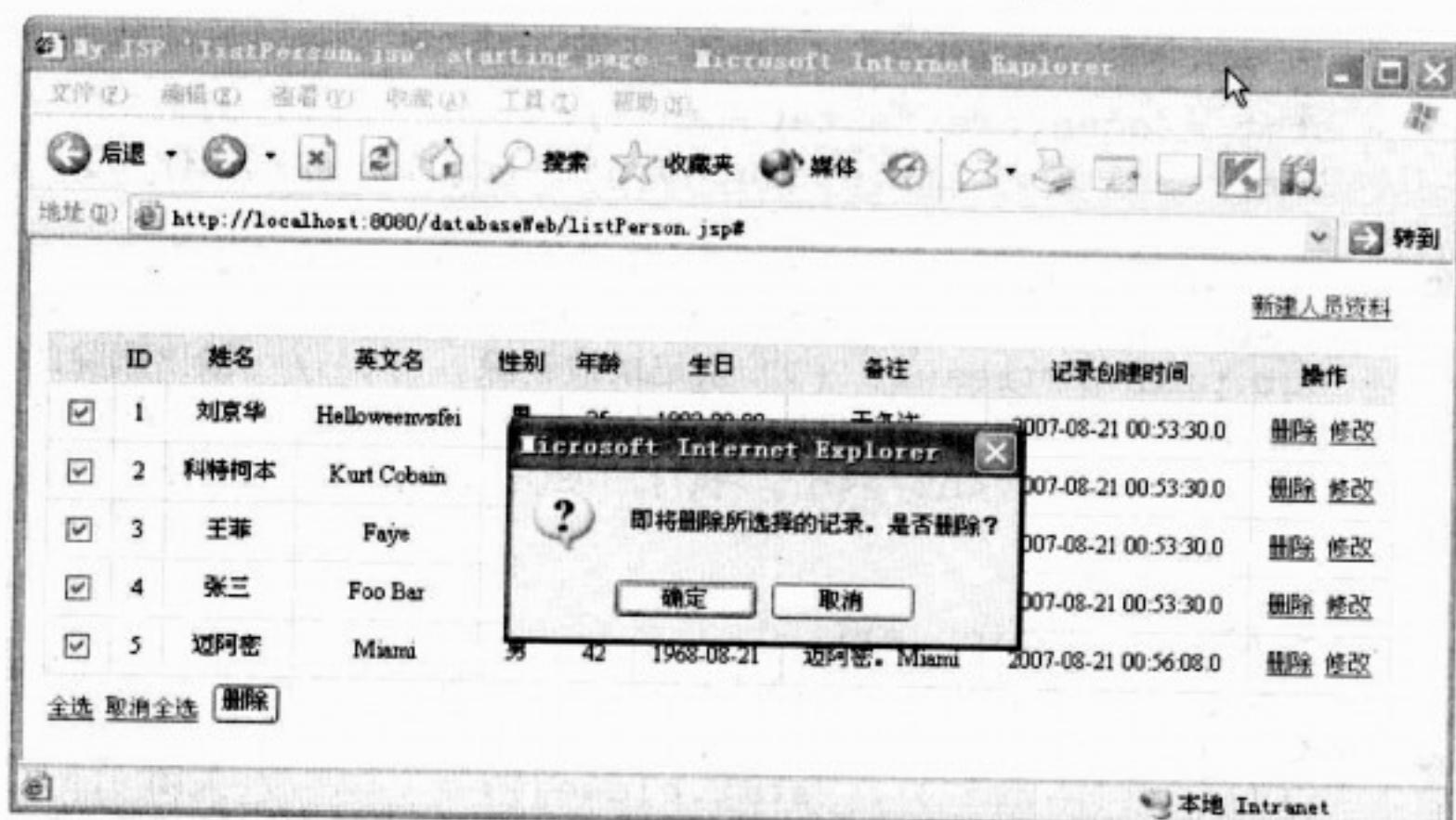


图 12.4 删除数据效果

客户单击删除后将提交删除请求。删除处理程序解析请求，判断是删除的哪一行或者哪些行数据，然后执行 DELETE 语句。operatePerson.jsp 中处理删除请求的代码如下：

代码 12.5 operatePerson.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.sql.DriverManager" %>
<%@ page import="java.sql.Connection" %>
<%@ page import="java.sql.Statement" %>
<%
    request.setCharacterEncoding("UTF-8"); // 设置 request 编码
    String action = request.getParameter("action"); // 获取 action 参数
    if("add".equals(action)){
        ...
    }
    else if("del".equals(action)){ // 删除操作
        String[] id = request.getParameterValues("id");
    }
}
```

```
// 取一个或者多个 ID 值
if(id == null || id.length == 0){  out.println("没有选中任何行");
return; }

String condition = ""; // 组织 SQL 条件
for(int i=0; i<id.length; i++){ // 将 id 连成形如 1,2,3,4 的格式
    if(i == 0) condition = "" + id[i];
    else        condition += ", " + id[i];
}
String sql = "DELETE FROM tb_person WHERE id IN ("+condition+") ";
// SQL 语句

Connection conn = null; // Connection 对象
Statement stmt = null; // Statement 对象
try{
    DriverManager.registerDriver(new com.mysql.jdbc.Driver()); // 注册驱动
    conn = DriverManager.getConnection( // 获取连接
        "jdbc:mysql://localhost:3306/databaseWeb?
        characterEncoding=UTF-8",
        "root", "admin");

    stmt = conn.createStatement(); // 创建 Statement
    int result = stmt.executeUpdate(sql); // 执行 SQL 语句

    out.println("<html><style>body{font-size:12px; line-
height:25px; }</style><body>");
    out.println(result + " 条记录被删除。"); // 输出删除的行数
    out.println("<a href='listPerson.jsp'>返回人员列表</a>");
    out.println("<br/><br/>执行的 SQL 语句为: <br/>" + sql);
    // 将 SQL 输出

} catch(SQLException e){
    out.println("执行 SQL\""+sql+"\"时发生异常: " + e.getMessage());
} finally{
    if(stmt != null)     stmt.close(); // 关闭 stmt
    if(conn != null)    conn.close(); // 关闭 conn
}
}
```

上述代码即可以删除单行数据，也可以删除多行数据。提交请求后选中的记录将被从数据库删除。打印出来的 SQL 语句可能为：

```
DELETE FROM tb_person WHERE id IN (1, 2, 3, 4, 5)
```

### 12.3.6 修改人员数据

相对于插入删除，修改要麻烦得多。修改数据需要先把原来的数据呈现出来，客户做出修改后再保存数据。也就是说，一个完整的修改操作需要多个子过程。本例中修改流程如图 12.5 所示。整个流程使用 action 参数来区分不同的操作。

人员列表 listPerson.jsp 中已经预留了修改的链接，单击“修改”链接就可以提交修改这一行的请求。operatePerson.jsp 处理修改请求的时候需要从数据库读取数据，使用 executeQuery() 方法执行 SELECT 语句，将数据显示到修改页面上。保存数据时使用

executeUpdate()方法执行 UPDATE 语句。其中处理修改请求的代码为：

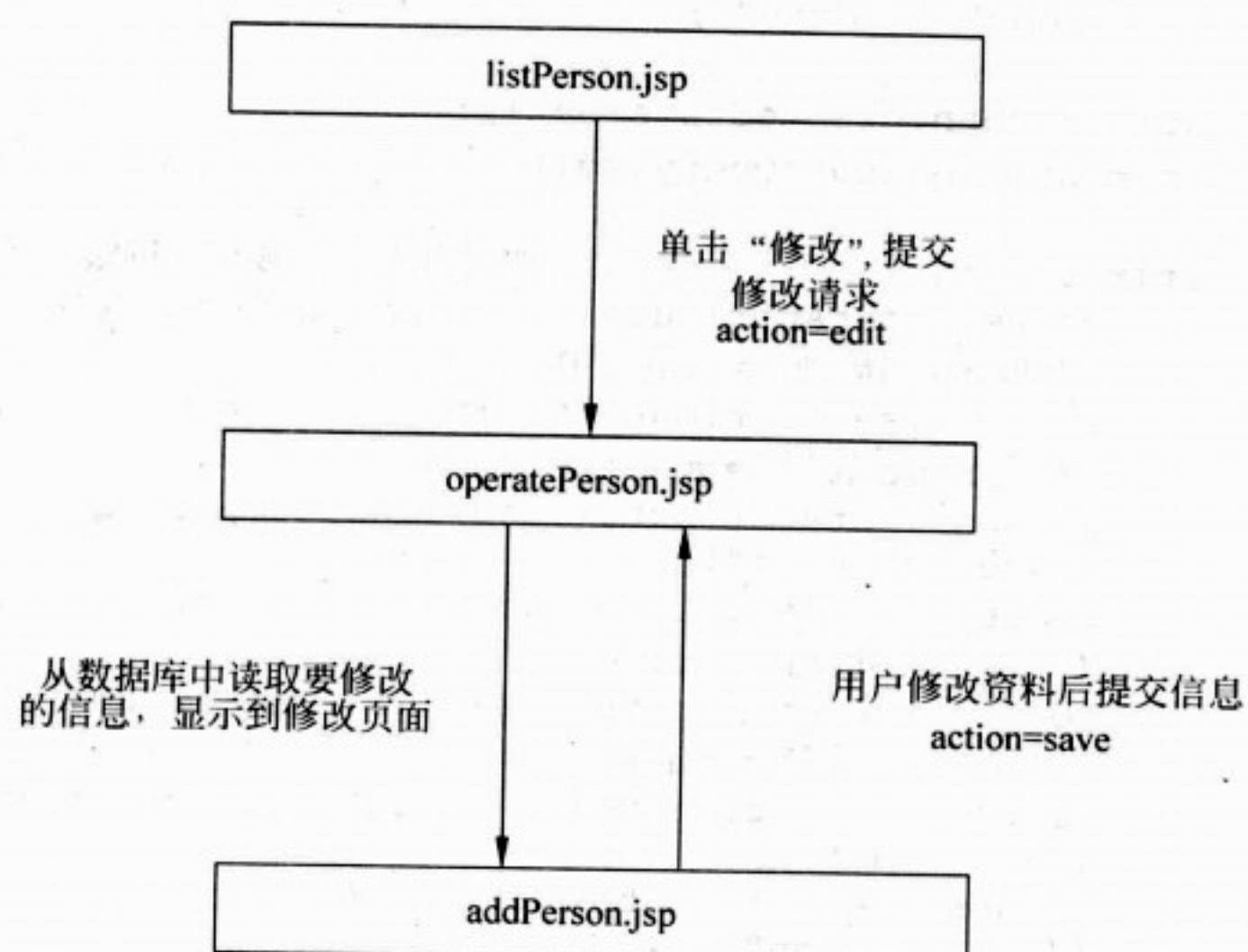


图 12.5 修改流程

#### 代码 12.6 operatePerson.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.sql.DriverManager" %>
<%@ page import="java.sql.Connection" %>
<%@ page import="java.sql.Statement" %>
<jsp:directive.page import="java.sql.ResultSet"/>
<%
    request.setCharacterEncoding("UTF-8"); // 对 request 编码
    String action = request.getParameter("action"); // action 参数

    if("add".equals(action)){
        ...
    }
    else if("del".equals(action)){
        ...
    }
    else if("edit".equals(action)){ // 显示修改页面

        String id = request.getParameter("id"); // 要修改的数据的 id
        String sql = "SELECT * FROM tb_person WHERE id = " + id;
        // 查询 SQL 语句

        Connection conn = null; // Connection 对象
        Statement stmt = null; // Statement 对象
        ResultSet rs = null; // ResultSet 对象

        try{
            DriverManager.registerDriver(new com.mysql.jdbc.Driver());
            // 注册驱动
            conn = DriverManager.getConnection( // 获取连接

```



```
"jdbc:mysql://localhost:3306/databaseWeb?characterEncoding=UTF-8",
    "root", "admin");

stmt = conn.createStatement();                      // 创建 Statement
rs = stmt.executeQuery(sql);                      // 查询结果

if(rs.next()){          // 如果有下一行数据，则滚动到下一行
    request.setAttribute("id", rs.getString("id"));
    // 取 id 列放到 request 中
    request.setAttribute("name", rs.getString("name"));
    // 取 name 放到 request
    request.setAttribute("englishName", rs.getString("english_name"));
    request.setAttribute("age", rs.getString("age"));
    // 取 age 列放到 request
    request.setAttribute("sex", rs.getString("sex"));
    // 取 sex 放到 request
    request.setAttribute("birthday", rs.getString("birthday"));
    request.setAttribute("description", rs.getString("description"));

    request.setAttribute("action", action);

    // 转到修改页面
    request.getRequestDispatcher("/addPerson.jsp").
        forward(request, response);
}
else{                                // 没有数据
    out.println("没有找到 id 为 " + id + " 的记录。");
}
}catch(SQLException e){
    out.println("执行 SQL\"" + sql + "\"时发生异常：" + e.getMessage());
    e.printStackTrace();
}finally{
    if(rs != null) rs.close();           // 关闭 rs
    if(stmt != null) stmt.close();       // 关闭 stmt
    if(conn != null) conn.close();       // 关闭 conn
}
}

%>
```

修改请求处理完毕后，将数据库中读取的字段放到 request 中，然后 forward 到修改页面。修改页面负责显示数据库中的信息。由于本例中修改页面跟添加页面是一个程序，因此需要处理好，当添加新数据时要提交添加请求（即提交的 action 参数为 add），修改数据时要显示原信息并提交修改请求（这里使用 action 参数为 save 来提交保存请求）。修改页面代码为：

代码 12.7 addPerson.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.
w3.org/TR/html4/loose.dtd">
<%
String action = (String)request.getAttribute("action");
```



```
// 取 action, 注意是getAttribute
String id = (String)request.getAttribute("id");      // 从 request 中取 id
String name = (String)request.getAttribute("name");
// 从 request 中取 name
String englishName = (String)request.getAttribute("englishName");
// 从 request 中取值
String age = (String)request.getAttribute("age"); // 从 request 中取 age
String sex = (String)request.getAttribute("sex"); // 从 request 中取 sex
String birthday = (String)request.getAttribute("birthday");
// 从 request 中取 birthday
String description = (String)request.getAttribute("description");
// 从 request 中取 description

// 判断是添加页面还是修改页面, 下文中根据此变量做相应的处理
boolean isEdit = "edit".equals(action);
%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title><%= isEdit ? "修改人员资料" : "新建人员资料" %></title>
<style type="text/css">body, td{font-size:12px; }</style>
</head>
<body>
<script type="text/javascript" src="js/calendar.js"></script>
<!-- 日期日历控件 --&gt;
&lt;form action="operatePerson.jsp" method="post"&gt;
&lt;input type="hidden" name="action" value="<%= isEdit ? "save" : "add" %&gt;"&gt;
&lt;input type="hidden" name="id" value="<%= isEdit ? id : "" %&gt;"&gt;
&lt;fieldset&gt;
&lt;legend&gt;&lt;%= isEdit ? "修改人员资料" : "新建人员资料" %&gt;&lt;/legend&gt;
&lt;table align=center&gt;
&lt;tr&gt;
&lt;td&gt;姓名&lt;/td&gt;
&lt;td&gt;&lt;input type="text" name="name" value="<%= isEdit ? name : "" %&gt;" /&gt;&lt;/td&gt;
&lt;/tr&gt;
&lt;tr&gt;
&lt;td&gt;英文名&lt;/td&gt;
&lt;td&gt;&lt;input type="text" name="englishName" value="<%= isEdit ? englishName : "" %&gt;" /&gt;&lt;/td&gt;
&lt;/tr&gt;
&lt;tr&gt;
&lt;td&gt;性别&lt;/td&gt;
&lt;td&gt;
&lt;input type="radio" name="sex" value="男" id="sex_male" &lt;%
isEdit&amp;&amp;"男".equals(sex) ? "checked" : "" %&gt; /&gt;&lt;label
for="sex_male"&gt;男&lt;/label&gt;
&lt;input type="radio" name="sex" value="女" id="sex_female"
&lt;%
isEdit&amp;&amp;"女".equals(sex) ? "checked" : "" %&gt; /&gt;&lt;label
for="sex_female"&gt;女&lt;/label&gt;
&lt;/td&gt;
&lt;/tr&gt;
&lt;tr&gt;
&lt;td&gt;年龄&lt;/td&gt;
&lt;td&gt;&lt;input type="text" name="age" value="<%= isEdit ? age : "" %&gt;" /&gt;&lt;/td&gt;
&lt;/tr&gt;
&lt;tr&gt;</pre>
```

```
<td>生日</td>
<td>
    <input type="text" name="birthday" onfocus="setday
(birthday)" value="<%=_isEdit? birthday : "" %>"/>
    
</td>
</tr>
<tr>
    <td>描述</td>
    <td><textarea name="description" ><%=_isEdit ? description :
"" %></textarea></td>
</tr>
<tr>
    <td></td>
    <td><input type="submit" value="<%=_isEdit ? "保存" : "添加人员
信息" %>"/></td>
</tr>
</table>
</fieldset>
</form>
</body>
</html>
```

需要注意的是修改信息的使用一定要保存 id 信息与 action 信息，否则保存信息时不知道是修改的哪条数据，是保存还是添加。显示原有信息时比较特别的是 checkbox 域与 textarea 域，与普通的 text 域不太一样，注意观察代码。

保存信息仍然写在 operatePerson.jsp 中，使用 action 区分。当 action 为 save 时执行保存操作，代码如下：

代码 12.8 operatePerson.jsp

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8" %>
<%@ page import="java.sql.DriverManager" %>
<%@ page import="java.sql.Connection" %>
<%@ page import="java.sql.Statement" %>
<jsp:directive.page import="java.sql.ResultSet"/>
<%
    request.setCharacterEncoding("UTF-8");                                // 设置 request 编码

    String name = request.getParameter("name");                            // 取 name 参数
    String englishName = request.getParameter("englishName");
    // 取 englishName 参数
    String age = request.getParameter("age");                                // 取 age 参数
    String birthday = request.getParameter("birthday"); // 取 birthday 参数
    String sex = request.getParameter("sex");                                // 取 sex 参数
    String description = request.getParameter("description");
    // 取 description 参数

    String action = request.getParameter("action");      // 取 action 参数

    if("add".equals(action)){                           // 添加代码略
        ...
    }
    else if("del".equals(action)){                     // 删除代码略
        ...
    }
}
```



```
}

else if("edit".equals(action)){
...
}

else if("save".equals(action)){ // 保存数据
    String id = request.getParameter("id"); // 要保存的数据 ID
    String sql = "UPDATE tb_person SET " + // 组织成 Update SQL 语句
        " name = '" + forSQL(name) + "', " +
        " english_name = '" + forSQL(englishName) + "', " +
        " sex = '" + sex + "', " +
        " age = '" + age + "', " +
        " birthday = '" + birthday + "', " +
        " description = '" + forSQL(description) + "' " +
        " WHERE id = " + id;

    Connection conn = null; // Connection 对象
    Statement stmt = null; // Statement 对象
    try{
        DriverManager.registerDriver(new com.mysql.jdbc.Driver()); // 注册驱动
        conn = DriverManager.getConnection( // 获取数据库连接
            "jdbc:mysql://localhost:3306/databaseWeb?
            characterEncoding=UTF-8",
            "root", "admin");

        stmt = conn.createStatement(); // 创建 Statement
        int result = stmt.executeUpdate(sql); // 执行 Update SQL 语句

        if(result == 0) // 判断修改的行数。正常情况应该是 1
            out.println("影响数目为 0, 修改失败。"); // 没有数据被修改
        else
            out.println(result + " 条记录被修改。"); // 修改行数

        out.println("<a href='listPerson.jsp'>返回人员列表</a>");
        out.println("<br/><br/>执行的 SQL 语句为: <br/>" + sql);
        // 输出 SQL
    }catch(SQLException e){
        out.println("执行 SQL\""+sql+"\"时发生异常: " + e.getMessage());
        e.printStackTrace();
    }finally{
        if(stmt != null) stmt.close(); // 关闭 Statement
        if(conn != null) conn.close(); // 关闭 Connection
    }
}
%>
```

由于只修改一条数据，因此返回的影响行数应该为 1。可以通过该返回值判断是否修改成功。修改页面与添加页面使用同一个程序，界面上差不多，这里就不贴出效果图了。

保存时打印出的 SQL 语句可能为：

```
UPDATE tb_person SET name = '张三', english_name='Zhang san', sex ='男',
age = '23', birthday = '2008-02-23', description = '无描述' WHERE id = 14
```

提示：修改是 Crud 操作中最复杂的，因为需要先在界面上恢复旧的数据，然后保存新的数据，还要保存 ID 等数据身份。

### 12.3.7 使用 PreparedStatement

上面的例子中查询、插入、删除、修改都是通过 Statement 对象实现的。使用 Statement 是最简单的方式，只需要组织出正确的 SQL 语句（SQL 错误会抛出 SQLException），然后执行 executeQuery 或者 executeUpdate 就可以了。

除了使用 Statement，还可以使用 PreparedStatement。PreparedStatement 接口继承自 Statement 接口，是 Statement 的子类，因此拥有 Statement 接口的所有方法，例如 executeQuery()、executeUpdate()、getGeneratedKeys() 等。

PreparedStatement 与 Statement 的最大区别是 PreparedStatement 可以使用参数，也就是（？）号。PreparedStatement 允许使用不完整的 SQL 语句，空缺的部分使用问号（？）代替，直到执行前的时候设置进去。下面的代码演示了如何使用 PreparedStatement 来保存数据。

```
if("save".equals(action)){                                // 保存数据
    String id = request.getParameter("id");                // 要保存的数据 ID

    String sql = "UPDATE tb_person SET name = ?, english_name = ?,
    sex = ?, age = ?, birthday = ?, description = ? WHERE id = ? ";
    // 带参数的 SQL 语句

    Connection conn = null;                                // Connection 对象
    PreparedStatement preStmt = null;                      // PreparedStatement 对象

    try{
        DriverManager.registerDriver(new com.mysql.jdbc.Driver()); // 注册驱动
        conn = DriverManager.getConnection(                  // 获取连接
            "jdbc:mysql://localhost:3306/databaseWeb?
            characterEncoding=UTF-8",
            "root", "admin");

        SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd");
        //日期格式化器
        Date d = new Date(f.parse(birthday).getTime()); // 格式化后的日期

        preStmt = conn.prepareStatement(sql); // 预编译带参数 SQL 语句
        preStmt.setString(1, name);          // 设置第 1 个参数
        preStmt.setString(2, englishName);   // 设置第 2 个参数
        preStmt.setString(3, sex);           // 设置第 3 个参数
        preStmt.setInt(4, Integer.parseInt(age)); // 设置第 4 个参数，int 类型
        preStmt.setDate(5, d);              // 设置第 5 个参数，日期类型
        preStmt.setString(6, description);   // 设置第 6 个参数
        preStmt.setInt(7, Integer.parseInt(id)); // 设置第 7 个参数

        int result = preStmt.executeUpdate(sql); // 执行 SQL 语句

        out.println("<html><style>body{font-size:12px; line-height:
        25px; }</style><body>");

        if(result == 0)      out.println("影响数目为 0, 修改失败. ");
        // 修改失败
    }
}
```

```
else    out.println(result + " 条记录被修改。"); // 修改成功

out.println("<a href='listPerson.jsp'>返回人员列表</a>");
out.println("<br/><br/>执行的 SQL 语句为: <br/>" + sql);
// 输出 SQL

}catch(SQLException e){
    out.println("执行 SQL\""+sql+"\"时发生异常: " + e.getMessage());
    e.printStackTrace();
}finally{
    if(preStmt != null) preStmt.close(); // 关闭 preStmt
    if(conn != null) conn.close(); // 关闭 conn
}
}
```

该段代码与前面使用 Statement 插入人员信息的效果是一样的。由于 PreparedStatement 使用问号代替了 SQL 中所有可变的部分，剩下了不变的部分，因此 JDBC 将 PreparedStatement 的 SQL 中不变的部分进行预编译，下次执行时直接高效率执行。当程序中有大量重复性的 SQL 时，更倾向于使用 PreparedStatement。流行的框架如 Hibernate、EJB3 都是倾向于使用 PreparedStatement 的。Statement 与 PreparedStatement 的区别如表 12.2 所示。

表 12.2 Statement 与 PreparedStatement 的区别

	Statement	PreparedStatement
预编译	不预编译，执行效率相对较低	不便宜，执行效率相对较高
复杂数据类型	不支持。需要调用数据库函数转化。 例如输入日期需要使用 to_date('2008-08-08','yyyy-MM-dd') (Oracle 数据库。MySQL 是可以直接 输入'2008-08-08'的)	支持。通过响应的 set 方法可以设 置。例如 setDate, setBinaryStream
大文本 (Clob 类型字段)	不支持。有些数据库中 SQL 语句有 长度限制。例如 Oracle 中最长是 4000 字节，超出了长度会报错	支持。大文本字段要通过 setClob 方 法设置
二进制数据 (Blob 类型字 段)	不支持	支持。二进制数据要通过 setBlob 方 法设置
批处理	支持。不可以设置参数	支持。可以设置参数

### 12.3.8 Statement 批处理 SQL

Statement 与 PreparedStatement 都能够批处理 SQL 语句，也就是同时执行多条 SQL 语句。这些 SQL 语句必须是 INSERT、UPDATE、DELETE 等 SQL 语句，它们执行后都返回一个 int 类型数，表示影响的行数。Statement 与 PreparedStatement 通过 addBatch()方法添加一条 SQL 语句，通过 executeBatch()方法批量执行 SQL 语句，返回一个 int 数组，代表各句 SQL 的返回值。先看 Statement 执行批处理语句的例子：

代码 12.9 BatchTest.java

```
package com.helloweenvsfei.test;
```

```
public class BatchTest {  
    public static void main(String[] args) throws SQLException {  
        new Driver(); // 注册驱动  
        Connection conn = null; // Connection 对象  
        Statement stmt = null; // Statement 对象  
        try {  
            conn = DriverManager.getConnection("// 获取连接  
                "jdbc:mysql://localhost:3306/databaseWeb?  
                characterEncoding=UTF-8",  
                "root", "admin");  
  
            stmt = conn.createStatement(); // 创建 Statement 对象  
  
            for (int i = 0; i < 5; i++) {  
                String sql = "insert into tb_person " // 完整的 SQL 语句  
                    + " ( name, english_name, age, "  
                    + " sex, birthday, description) " + " values ('Name  
                        " + i + "', 'English Name " + i + "'", "  
                        + " '17', '男', current_date(), '') ";  
  
                stmt.addBatch(sql); // 批量添加  
            }  
            int[] result = stmt.executeBatch(); // 批量执行将每句 SQL 执行结果组成 int[] 数组  
  
            System.out.print("影响的行数分别为: ");  
            for (int i = 0; i < result.length; i++) {  
                System.out.print(result[i] + ", "); // 输出每行 SQL 的结果  
            }  
        } finally {  
            if (stmt != null) stmt.close(); // 关闭 Statement  
            if (conn != null) conn.close(); // 关闭 Connection  
        }  
    }  
}
```

运行结果可能为：

影响的行数分别为：1, 1, 1, 1, 1

表示这 5 句 SQL 语句均影响了 1 条记录。

 提示：批量处理 SQL 将多条语句提交给数据库一块执行，效率更高一些。但是如果数据比较多，比如 4 万条 SQL，就需要分批次执行，例如 100 条执行一次。

### 12.3.9 PreparedStatement 批处理 SQL

Statement 的批处理 SQL 必须为完整的 SQL 语句。而 PreparedStatement 就灵活得多，既可以用完整的 SQL（因为它继承自 Statement），又可以用带参数的不完整的 SQL，例如：

代码 12.10 PreparedBatchTest.java

```
package com.halloweenvsfei.test;
```

```
public class PreparedBatchTest {
    public static void main(String[] args) throws SQLException {
        new Driver();                                // 注册驱动
        Connection conn = null;                      // Connection 对象
        PreparedStatement preStmt = null;             // Statement 对象
        try {
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/databaseWeb?characterEncoding=UTF-8", "root", "admin");

            preStmt = conn.prepareStatement("insert into tb_person "
                + " (name, english_name, age, sex, birthday, description)"
                + " values (?, ?, ?, ?, ?, ?)");
            for (int i = 0; i < 5; i++) {
                int index = 1;
                preStmt.setString(index++, "Name " + i); // 设置第 1 个参数
                preStmt.setString(index++, "English Name" + i); // 设置第 2 个参数
                preStmt.setInt(index++, 25); // 设置第 3 个参数
                preStmt.setString(index++, "男"); // 设置第 4 个参数
                preStmt.setDate(index++, new java.sql.Date(System.currentTimeMillis()));
                preStmt.setString(index++, ""); // 设置第 6 个参数
                preStmt.addBatch(); // 添加同一条带参数的 SQL 语句
            }
            int[] result = preStmt.executeBatch();
            // 批量执行返回 int[] 数组
            System.out.print("影响的行数分别为: ");
            for (int i = 0; i < result.length; i++) {
                System.out.print(result[i] + ", ");
            }
        } finally {
            if (preStmt != null) preStmt.close();
            // 关闭 PreparedStatement
            if (conn != null) conn.close(); // 关闭 Connection
        }
    }
}
```

运行结果可能为：

影响的行数分别为： 1, 1, 1, 1, 1

**注意：**能够批量执行的 SQL 必须是 INSERT、UPDATE、DELETE 等返回 int 类型的 SQL，因为 Statement 或者 PreparedStatement 的 executeBatch()方法都返回 int[] 数据。如果批量执行 SELECT 语句，会报错。另外批量执行 SQL 需要数据库的支持，某些数据库可能不支持。

## 12.4 处理结果集

查询结果都保持在 `ResultSet` 结果集中。遍历结果集便可取到数据。

### 12.4.1 查询多个结果集

实际应用中，一般会查询多个表格。查询多个表格可以使用同一个 `Statement` 或者 `PreparedStatement` 实现，返回同一个 `ResultSet` 对象，例如：

```
rs = stmt.executeQuery(" select * from 表格一 ");
while (rs.next()) {
    ... // 遍历 ResultSet
}
// 中间不需要执行 rs.close(), JDBC 默认自动关闭前一次查询的 ResultSet
rs = stmt.executeQuery(" select * from 表格二 ");
while (rs.next()) {
    ... // 遍历 ResultSet
}
```

进行第一次查询时，`stmt` 会返回一个 `ResultSet` 对象。进行第二次查询时，`stmt` 会返回一个全新的对象。中间不需要执行 `rs.close()` 关闭第一个对象。这是因为 `Statement` 与 `PreparedStatement` 有一个特性，在返回 `ResultSet` 结果集时，会自动关闭上一次查询的结果集。

### 12.4.2 可以滚动的结果集

除了常用的 `next()` 方法，`ResultSet` 接口还有其他的方法，比如 `previous()`、`first()`、`last` 等。如果后面还有记录，`next()` 会返回 `true`，同时自动向下滚动一条记录，否则返回 `false`。`previous()` 方法刚好相反，如果前面还有记录，`previous()` 会返回 `true`，同时自动向上滚动一条记录，否则返回 `false`。`first()` 是滚动到第一条记录，如果第一条记录存在的话。`last()` 是滚动到最后一条记录。

为了效率，`Statement` 默认返回的 `ResultSet` 是只可往后滚动的，因此只有 `next()`、`last()` 方法是可用的。要想使用 `previous()`、`first()` 方法，可以这样创建 `Statement`：

```
stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery(" ... ");
```

第一个参数指定 `Statement` 创建的 `ResultSet` 可以自由滚动，第二个参数指定 `Statement` 创建的 `ResultSet` 可以直接修改。

### 12.4.3 Pagination 分页显示

数据量大时需要分页显示。分页显示时只从数据库取出本页要显示的记录，而不必把



所有的记录都取出来。MySQL 中实现分页是利用 LIMIT 实现的。例如“SELECT \* FROM tb\_person LIMIT 21, 10”只取出从第 21 行开始的 10 行记录(而不是从第 21 行到第 10 行)。

分页显示时需要先查询记录总数，执行形如 SELECT count(\*) FROM tb\_person 的 SQL 语句，将返回包含总记录数的只有一行一列的 ResultSet。然后计算总页数，本页第一条记录的在数据库中的行数等。当前的页数将作为地址栏参数传递给服务器。

分页显示还要生成导航的信息，比如第一页，上一页，下一页，最后一页等。看一个实际的例子：

代码 12.11 listPagedPerson.jsp

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.sql.Connection" %>
<%@ page import="java.sql.ResultSet" %>
<jsp:directive.page import="java.sql.Date"/>
<jsp:directive.page import="java.sql.Timestamp"/>
<jsp:directive.page import="java.sql.SQLException"/>
<jsp:directive.page import="com.helloweenvsfei.util.DbManager"/>
<jsp:directive.page import="java.sql.PreparedStatement"/>
<jsp:directive.page import="com.helloweenvsfei.util.Pagination"/>
<%
    final int pageSize = 10;                                // 一页显示 10 条记录
    int pageNum = 1;                                         // 当前页数，默认为 1
    int pageCount = 1;                                       // 总页数
    int recordCount = 0;                                     // 总记录数

    try{
        pageNum = Integer.parseInt(request.getParameter("pageNum"));
        // 取当前页数
    }catch(Exception e){}

    String sql = null;                                       // 查询 SQL 语句

    Connection conn = null;                                    // Connection 对象
    PreparedStatement preStmt = null;                         // PreparedStatement 对象
    ResultSet rs = null;                                     // ResultSet 对象

    try{
        sql = "SELECT count(*) FROM tb_person "; // 查询记录总数的 SQL 语句

        recordCount = DbManager.getCount(sql);      // 封装的方法，返回记录总数
        pageCount = ( recordCount + pageSize - 1 ) / pageSize;
        // 计算总页数
        int startRecord = ( pageNum - 1 ) * pageSize;
        // 本页从 startRecord 行开始

        sql = "SELECT * FROM tb_person LIMIT ?, ? ";
        // MySQL 使用 limit 实现分页

        conn = DbManager.getConnection();           // 第二次查询
        preStmt = conn.prepareStatement(sql);       // 查询结果集
        DbManager.setParams(preStmt, startRecord, pageSize);
        // 封装的方法设置起始页参数
        rs = preStmt.executeQuery();               // 执行查询
    }
```



```
%>
<form action="operatePerson.jsp" method="get">
<table bgcolor="#CCCCCC" cellspacing="1 cellpadding="5" width="100%">
<tr bgcolor="#DDDDDD">
<th>ID</th>
<th>姓名</th>
<th>英文名</th>
<th>性别</th>
<th>年龄</th>
<th>生日</th>
<th>备注</th>
<th>记录创建时间</th>
<th>操作</th>
</tr>
<%
    while(rs.next()) {                                // 遍历结果集
        int id = rs.getInt("id");                      // 获取 id 列, int 类型
        int age = rs.getInt("age");                     // 获取 age, int 类型
        String name = rs.getString("name");           // 获取 name 列
        String englishName = rs.getString("english_name");
        // 获取 english_name
        String sex = rs.getString("sex");              // 获取 sex 列
        String description = rs.getString("description");
        // 获取 description 列
        Date birthday = rs.getDate("birthday");
        // 获取 date 类型, 只有日期没有时间
        Timestamp ts = rs.getTimestamp("create_time");
        // 时间戳类型, 既有日期又有时间

        out.println("      <tr bgcolor="#FFFFFF">");          // 打印表头
        out.println("          <td>" + id + "</td>");          // 打印 id
        out.println("          <td>" + name + "</td>");          // 打印 name
        out.println("          <td>" + englishName + "</td>");    // 打印 english_name
        out.println("          <td>" + sex + "</td>");          // 打印 sex
        out.println("          <td>" + age + "</td>");          // 打印 age
        out.println("          <td>" + birthday + "</td>");        // 打印 birthday
        out.println("          <td>" + description + "</td>");    // 打印 description
        out.println("          <td>" + ts + "</td>");          // 打印 timestamp
        out.println("          <td>");                          // 打印操作列
        out.println("              <a href='operatePerson.jsp?");   // 打印删除链接
        out.println("                  action=del&id=" + id + "'"); // 打印删除参数
        out.println("                  onclick='return confirm(\"确定删除记"); // 打印确认框
        out.println("                  录? \")'); ' >删除</a>");       // 打印删除按钮
        out.println("              <a href='operatePerson.jsp?");   // 打印修改链接
        out.println("                  action=edit&id=" + id + "'>修改</a>"); // 打印修改参数
        out.println("          </td>");                         // 打印操作列
        out.println("      </tr>");                           // 打印表尾
    }
%>
</table>
<table align="right"><tr><td> <%= Pagination.getPagination(pageNum, pageCount, recordCount, request.getRequestURI()) %> </td></tr></table>
    <!-- 输出上一页、下一页等 -->
<br/><br/><br/>
<table align="left"><tr><td>SQL: <%= sql %> </td></tr></table>
</form>
<%
    }catch(SQLException e){
        out.println("执行 SQL: " + sql + "时发生异常: " + e.getMessage());
    }
}
```



```
    e.printStackTrace();
}finally{
    if(rs != null) rs.close(); // 关闭 ResultSet
    if(preStmt != null) preStmt.close(); // 关闭 PreparedStatement
    if(conn != null) conn.close(); // 关闭 Connection
}
%>
</body>
</html>
```

代码中使用了 DbManager 与 Pagination。DbManager 封装了一些常用的 JDBC 代码，代码如下：

代码 12.12 DbManager.java

```
package com.helloweenvsfei.util;
public class DbManager {
    public static Connection getConnection() throws SQLException {
        // 获取默认数据库连接
        return getConnection("databaseWeb", "root", "admin");
    }
    public static Connection getConnection(String dbName, String userName,
        // 获取数据库连接
        String password) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/" + dbName
        // 连接字符串
        + "?useUnicode=true&characterEncoding=utf-8";
        DriverManager.registerDriver(new Driver()); // 注册驱动
        return DriverManager.getConnection(url, userName, password);
        // 获取连接
    }
    public static void setParams(PreparedStatement preStmt, Object...
        params) // 设置参数
        throws SQLException {
        if (params == null || params.length == 0) // 如果没有参数，直接返回
            return;
        for (int i = 1; i <= params.length; i++) { // 依次设置参数
            Object param = params[i - 1]; // 第 i-1 个参数，从 0 起
            if (param == null) {
                preStmt.setNull(i, Types.NULL); // 设置 NULL 参数
            } else if (param instanceof Integer) {
                preStmt.setInt(i, (Integer) param); // 设置 Integer 参数
            } else if (param instanceof String) {
                preStmt.setString(i, (String) param); // 设置 String 参数
            } else if (param instanceof Double) {
                preStmt.setDouble(i, (Double) param); // 设置 Double 参数
            } else if (param instanceof Long) {
                preStmt.setDouble(i, (Long) param); // 设置 Long 参数
            } else if (param instanceof Timestamp) {
                preStmt.setTimestamp(i, (Timestamp) param);
                // 设置 Timestamp 参数
            } else if (param instanceof Boolean) {
                preStmt.setBoolean(i, (Boolean) param); // 设置 Boolean 参数
            } else if (param instanceof Date) {
                preStmt.setDate(i, (Date) param); // 设置 Date 参数
            }
        }
    }
}
```



```
        }
    }

    public static int executeUpdate(String sql) throws SQLException {
        // 执行 SQL, 返回影响行数
        return executeUpdate(sql, new Object[] {});
    }

    public static int executeUpdate(String sql, Object... params)
        // 执行 SQL, 返回影响的行数
        throws SQLException {
        Connection conn = null; // Connection 对象
        PreparedStatement preStmt = null; // PreparedStatement 参数
        try {
            conn = getConnection(); // 获取 Connection
            preStmt = conn.prepareStatement(sql); // 预编译带参数的 SQL 语句
            setParams(preStmt, params); // 设置参数
            return preStmt.executeUpdate(); // 返回执行结果
        } finally {
            if (preStmt != null) preStmt.close(); // 关闭 preStmt
            if (conn != null) conn.close(); // 关闭 conn
        }
    }

    /**
     * 获得总数。
     * @param sql 格式必须为 SELECT count(*) FROM ...
     * @return
     * @throws SQLException
     */
    public static int getCount(String sql) throws SQLException {
        Connection conn = null; // 设置 Connection 对象为空
        Statement stmt = null; // 设置 Statement 对象为空
        ResultSet rs = null; // 设置 ResultSet 对象为空
        try {
            conn = getConnection(); // 获取连接
            stmt = conn.createStatement(); // 创建 Statement
            rs = stmt.executeQuery(sql); // 执行 SQL 语句
            return rs.getInt(1); // 返回第一列数据（查询到的记录总数）
        } finally {
            if (rs != null) rs.close(); // 关闭 rs
            if (stmt != null) stmt.close(); // 关闭 stmt
            if (conn != null) conn.close(); // 关闭 conn
        }
    }
}
```

Pagination 用于输出第一页、上一页、下一页、最后一页等信息，代码如下：

代码 12.13 Pagination.java

```
package com.helloweenvsfei.util;
public class Pagination {
    /**
     * @param pageNum 当前页数
     * @param pageCount 总页数
```

## 第 12 章 JDBC 详解

```
* @param recordCount 总记录数
* @param pageUrl 页面 URL
* @return 上一页、下一页等分页字符串
*/
public static String getPagination(int pageNum, int pageCount, int
recordCount, String pageUrl){
    String url = pageUrl.contains("?)") ? pageUrl : pageUrl + "?";
    // 如果没有"?"则添加
    if(!url.endsWith("?") && !url.endsWith("&")){// 添加"&"
        url += "&";
    }
    StringBuffer buffer = new StringBuffer(); // StringBuffer 对象
    buffer.append("第 " + pageNum + "/" + pageCount + " 页 共" +
    recordCount + " 记录 ");

    buffer.append(pageNum == 1 ? "第一页" : "<a href='"+url+"pageNum=1'>
    第一页</a> ");
    buffer.append(pageNum == 1 ? "上一页" : "<a href='"+url+"pageNum=
    "+(pageNum-1) + "'>上一页</a> ");
    buffer.append(pageNum == pageCount ? "下一页" : "<a href='"+url+
    "pageNum=" + (pageNum + 1) + "'>下一页</a> ");
    buffer.append(pageNum == pageCount ? "最后一页" : "<a href='"+
    url+"pageNum=" + pageCount + "'>最后一页</a> ");
    return buffer.toString();
}
}
```

程序执行效果如图 12.6 所示。



图 12.6 分页显示效果

一般使用整除操作计算总页数。例如总记录数为 12，每页记录数为 10，则总页数应该为  $12/10+1=2$ 。但如果总记录数为 10，每页记录数为 10，总页数应该为 1。不同的地方在于余数不一样。因此计算总页数代码应该分别对待：

```
pageCount = recordCount / pageSize;
if(recordCount % pageSize != 0)    pageCount += 1;
```

需要两行代码。listPagedPerson.jsp 中使用了另一种更简单的方式：

```
pageCount = ( recordCount + pageSize - 1 ) / pageSize.
```

读者可以自行验证一下，二者是等效的。

**注意：**SQL 并没有对分页显示制定标准，因而不同的数据库采用不同的方式实现分页。例如 MySQL 使用 LIMIT 而 Oracle 中使用 ROWNUM，语法也不一样。JDBC 编程中实现分页时要根据数据库选择不同的 SQL 语句。如果数据库不支持分页，需要使用 ResultSet.next()滚动到当前位置。

#### 12.4.4 带条件的查询

实用的查询页面能够设置复杂的查询条件。其基本原理是把查询条件转化为对应的 WHERE 子句，然后使用包含该 WHERE 子句的 SQL 查询数据库。下面的例子对 tb\_person 表进行复杂的查询：

代码 12.14 searchPerson.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%
    public String forSQL(String sql){                                // 替换 SQL 中的符号'为\
        return sql.replace("'", "\\'");
    }
%>
<%
    request.setCharacterEncoding("UTF-8");                         // 设置 request 编码
    final int pageSize = 5;                                         // 每页 5 条记录
    int pageNum = 1;                                                 // 当前页数
    try{
        pageNum = new Integer(request.getParameter("pageNum"));   // 获取当前页数
    }catch(Exception e){}
    String nameSearch = request.getParameter("name");           // 获取 name 参数
    String sexSearch = request.getParameter("sex");             // 获取 sex 参数
    String englishNameSearch = request.getParameter("englishName");
    // englishName
    String descriptionSearch = request.getParameter("description");
    // 获取 description
    String birthdayStart = request.getParameter("birthdayStart");
    // 获取生日起始日
    String birthdayEnd = request.getParameter("birthdayEnd");
    // 获取生日结束日
    String whereClause = "";                                     // where 子句
    if(nameSearch!=null && nameSearch.trim().length()!=0){
        // name 模糊匹配
        if(whereClause.length() == 0)
            whereClause += " name LIKE '%" + forSQL(nameSearch) + "%'";
        else
            whereClause += " AND name LIKE '%" + forSQL(nameSearch) + "%'";
    }
%>
```



```
}

if(sexSearch!=null && sexSearch.trim().length()!=0){ // sex 精确匹配
    if(whereClause.length() == 0)
        whereClause += " sex = '" + forSQL(sexSearch) + "' ";
    else
        whereClause += " AND sex = '" + forSQL(sexSearch) + "' ";
}
if(englishNameSearch!=null && englishNameSearch.trim().length()!=0){
    // 英文名模糊匹配
    if(whereClause.length() == 0)
        whereClause += " english_name LIKE '%" + forSQL
        (englishNameSearch) + "%' ";
    else
        whereClause += " AND english_name LIKE '%" + forSQL
        (englishNameSearch) + "%' ";
}
if(descriptionSearch!=null && descriptionSearch.trim().length()!=0){
    // 描述模糊匹配
    if(whereClause.length() == 0)
        whereClause += " description LIKE '%" + forSQL(descriptionSearch)
        + "%' ";
    else
        whereClause += " AND description LIKE '%" + forSQL
        (descriptionSearch) + "%' ";
}
if(birthdayStart!=null && birthdayStart.trim().length()!=0){
    // 生日起始日
    if(whereClause.length() == 0)
        whereClause += " birthday >= '" + birthdayStart + "' ";
    else
        whereClause += " AND birthday >= '" + birthdayStart + "' ";
}
if(birthdayEnd!=null && birthdayEnd.trim().length()!=0){
    // 生日结束日
    if(whereClause.length() == 0)
        whereClause += " birthday <= '" + birthdayEnd + "' ";
    else
        whereClause += " AND birthday <= '" + birthdayEnd + "' ";
}
if(whereClause.length() != 0){ // 添加 where 关键字
    whereClause = " WHERE " + whereClause;
}
String countSQL = " SELECT count(*) FROM tb_person " + whereClause;
// 记录数 SQL

int recordCount = DbManager.getCount(countSQL); // 获取记录总数
int pageCount = (recordCount + pageSize) / pageSize;
// 计算分页数据

String querySQL = " SELECT * FROM tb_person " + whereClause + " LIMIT "
" + (pageNum-1)*pageSize + ", " + pageSize; // 查询 SQL 语句

Connection conn = null; // Connection 对象
Statement stmt = null; // Statement 对象
ResultSet rs = null; // ResultSet 对象

try{
    conn = DbManager.getConnection(); // 获取连接
    stmt = conn.createStatement(); // 创建 Statement 对象
```



```
        rs = stmt.executeQuery(querySQL);           // 执行查询
%>
<form action="searchPerson.jsp" method="get">
<fieldset style='width:80%'>
    <legend>查询条件</legend>
    <table >
        <tr>
            <td style="text-align:right; ">姓名</td>
            <td style="text-align:left; ">
                <input type='text' name='name' value="${ param.name }"/>
            </td>
            <td style="text-align:right; ">性别</td>
            <td style="text-align:left; ">
                <select name='sex' />
                    <option value="">无限制</option>
                    <option value="男" ${ '男'==param.sex ? 'selected' : '' }>男</option>
                    <option value="女" ${ '女'==param.sex ? 'selected' : '' }>女</option>
                </select>
            </td>
        </tr>
        <tr>
            <td style="text-align:right; ">英文名</td>
            <td style="text-align:left; ">
                <input type='text' name='englishName' value="${ param.englishName }"/>
            </td>
            <td style="text-align:right; ">备注</td>
            <td style="text-align:left; ">
                <input type='text' name='description' value="${ param.description }"/>
            </td>
        </tr>
        <tr>
            <td colspan=4>
                出生日期
                从 <input type='text' name='birthdayStart' onfocus="setday(birthdayStart); " value="${ param.birthdayStart }"/>
                &ampnbsp&ampnbsp
                到 <input type='text' name='birthdayEnd' onfocus="setday(birthdayEnd); " value="${ param.birthdayEnd }"/>
                
            </td>
        </tr>
        <tr>
            <td colspan=4>
                <input type="submit" value="提交查询">
                <input type="reset" value="复位">
            </td>
        </tr>
    </table>
</fieldset>
<br/>
<table bgcolor="#CCCCCC" cellspacing=1 cellpadding=5 width=100%>
    <tr bgcolor="#DDDDDD">
```

```
<td>ID</td>
<td>姓名</td>
<td>英文名</td>
<td>性别</td>
<td>年龄</td>
<td>生日</td>
<td>备注</td>
<td>记录创建时间</td>
</tr>
<%
    while(rs.next()){
        ... // 遍历结果集。代码略
    }
%>
</form>
<%
    }catch(SQLException e){
        out.println("执行 SQL: " + querySQL + "时出错: " + e.getMessage());
    }finally{
        if(rs != null) rs.close();                                // 关闭 rs
        if(stmt != null) stmt.close();                            // 关闭 stmt
        if(conn != null) conn.close();                           // 关闭 conn
    }
%>
```

注意当客户没有填写姓名时 WHERE 条件里不能出现姓名列。程序中对 name 参数做了判断，如果 name 为空则不查询 name。另外还要注意把用户填写的信息在查询结果页面上显示。运行效果如图 12.7 所示。

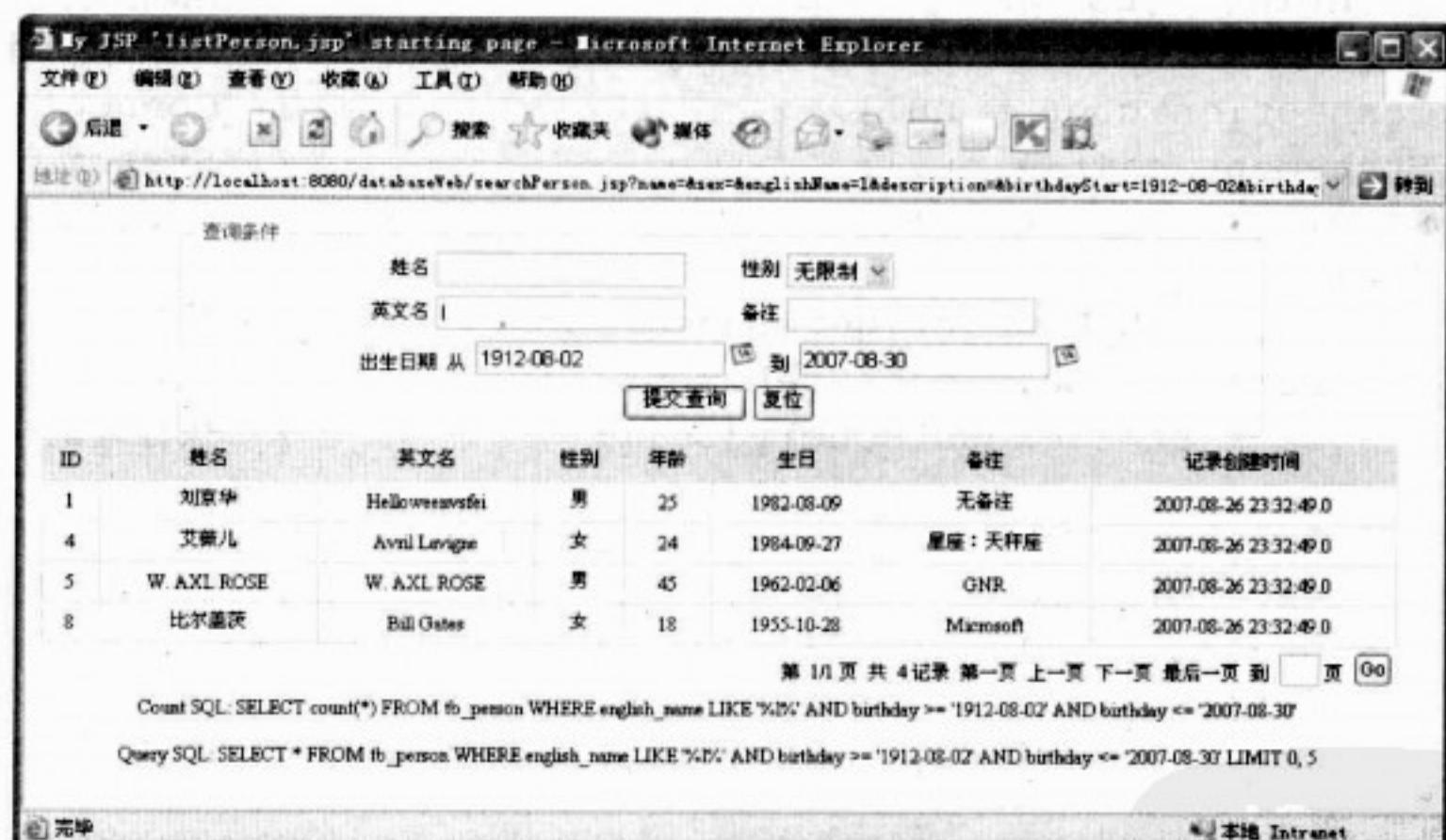


图 12.7 复杂查询效果图

程序输出的 SQL 语句可能为：

```
SELECT count(*) FROM tb_person WHERE name LIKE '%张%' AND sex = '男' AND
birthday >= '2008-01-28'

SELECT * FROM tb_person WHERE name LIKE '%张%' AND sex = '男' AND birthday
>= '2008-01-28' LIMIT 0, 5
```

### 12.4.5 ResultSetMetaData 元数据

前面的查询都是预先知道列的名字，例如 name 列，然后通过 rs.getString("name") 来获取当前行中 name 列的内容。实际上，JDBC 可以直接获取 ResultSet 对象的列名。根据获取到的列名，可以动态的显示查询的各列内容。

ResultSet 对象的列名可由 ResultSetMetaData 元数据获得。ResultSet.getMetaData 可以返回元数据，遍历元数据即可获知 ResultSet 中有哪些列，每一列是什么类型，例如：

代码 12.15 searchConsole.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<jsp:directive.page import="java.sql.Connection"/>
<jsp:directive.page import="java.sql.Statement"/>
<jsp:directive.page import="java.sql.ResultSet"/>
<jsp:directive.page import="java.sql.SQLException"/>
<jsp:directive.page import="com.helloweenvsfei.util.DbManager"/>
<%
    request.setCharacterEncoding("UTF-8"); // 对 request 编码
%>
<form action="${ pageContext.request.requestURL }" method="get">
    <textarea name="sql">${ param.sql }</textarea> <input type="submit">
</form>
<%
    String sql = request.getParameter("sql"); // 获取提交的 SQL
    out.println("SQL语句: " + sql); // 输出 SQL
    if(sql != null && sql.trim().length() > 0){ // 如果 SQL 不为空
        Connection conn = null; // Connection 对象
        Statement stmt = null; // Statement 对象
        ResultSet rs = null; // ResultSet 对象
        try{
            conn = DbManager.getConnection(); // 获取连接
            stmt = conn.createStatement(); // 创建 Statement 对象
            rs = stmt.executeQuery(sql); // 执行查询

            ResultSetMetaData meta = rs.getMetaData(); // 获取元数据
            int columnCount = meta.getColumnCount(); // 获取列数
            String[] columns = new String[columnCount]; // 列名数组
            for(int i=1; i<=columnCount; i++){
                columns[i-1] = meta.getColumnName(i); // 获取每一列的列名
            }
            StringBuffer buffer = new StringBuffer(); // 输出结果集
            buffer.append("<table>");
            buffer.append(" <tr>");
            for(String column : columns){
                buffer.append(" <th>" + column + "</th>"); // 输出表头, 列名
            }
            buffer.append(" </tr>"); // 遍历各行数据
            while(rs.next()){
                buffer.append(" <tr>");
                for(String column : columns){ // 输出各行各列数据
                    buffer.append(" <td>" + rs.getString(column) +
                    "</td>");
```

```
        }
        buffer.append("</tr>");
    }
    buffer.append("</table>");  
    out.println(buffer.toString()); // 输出到客户端
} catch (SQLException e) {
    out.println("<div class=error>执行 SQL: "+sql+"时出错: "+e.  
getMessage()+"</div>");  
    e.printStackTrace();
} finally{
    if(rs != null) rs.close(); // 关闭 rs
    if(stmt != null) stmt.close(); // 关闭 stmt
    if(conn != null) conn.close(); // 关闭 conn
}
%>
</body>
</html>
```

该程序不仅能够执行诸如“SELECT \* FROM tb\_person”的常规 SQL，显示任意表格的数据（前提是具有读取权限），还能执行如“SHOW DATABASES”、“SHOW VARIABLES”等非标准的 SQL。执行“SHOW VARIABLES”将显示 MySQL 数据库的所有变量。运行效果如图 12.8 所示。

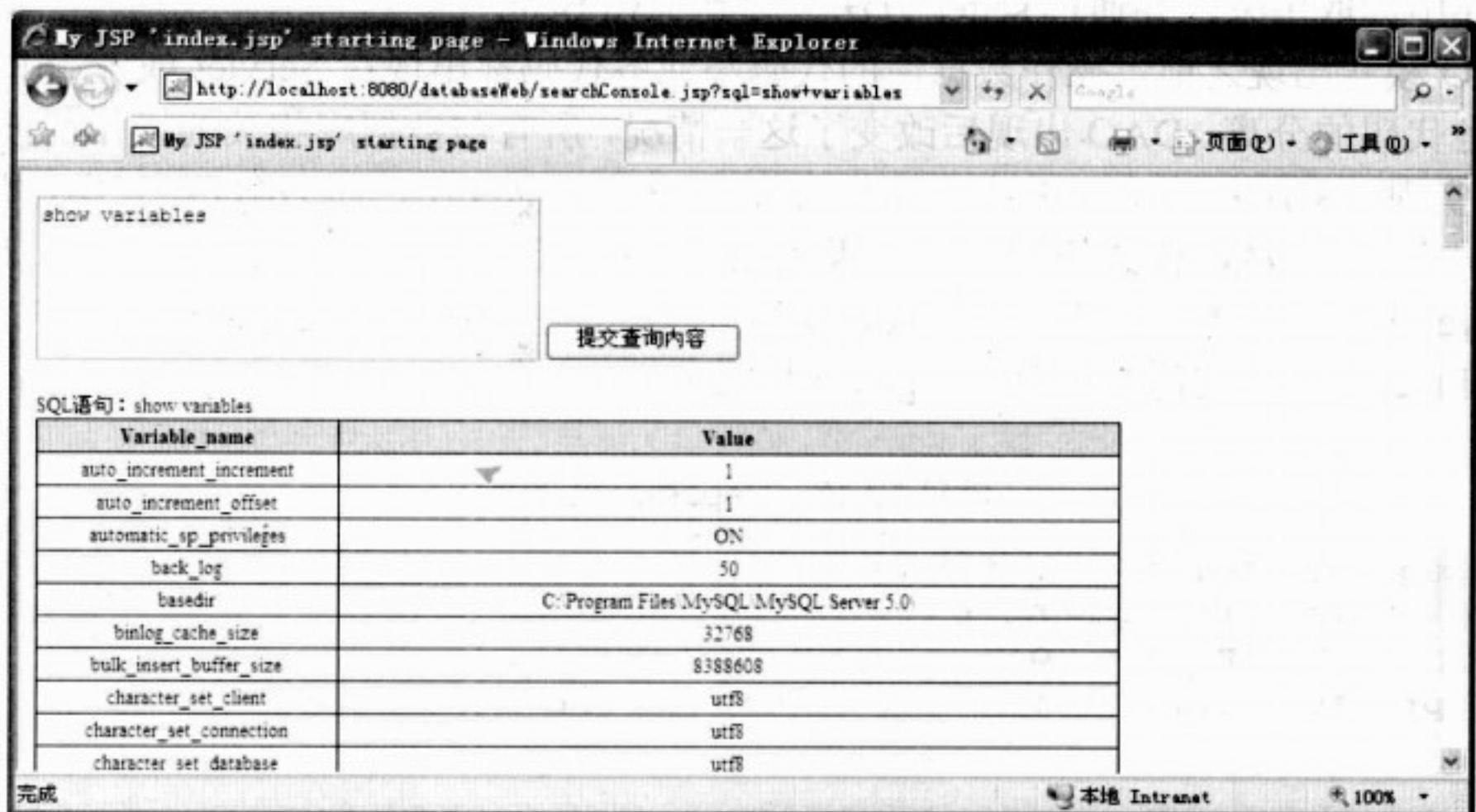


图 12.8 运行效果

**提示：**ResultSetMetaData 元数据中还包括其他信息，例如列名称、列类型、列长度等，可根据需要获取相应的信息。

#### 12.4.6 直接显示中文列名

查询的时候 SQL 可以声明别名。例如：

```
"SELECT id 编号, name 姓名, english_name 英文名, sex 性别 from tb_person"
```

使用元数据，理论上应该能显示 id、name、english\_name、sex 列的数据，并且列名分别显示为“编号”、“姓名”、“英文名”、“性别”。不过令人遗憾的是，MySQL 的驱动在处理中文列名时会有乱码。这时候可能需要对列名进行编码转化，例如：

```
String columnName = rs.getMetaData().getColumnName(1); // 获取列名
columnName = new String(columnName.getBytes("latin1"), "utf-8");
// 将列名进行编码转换
```

## 12.5 JDBC 高级应用

进行了大量的 JDBC 编程之后，开发者们渐渐地积累了一些经验，并对 JDBC 进行分层、模块化。在这些经验中，DAO 与 Java Bean 是最有用的两个。

### 12.5.1 DAO 模式与 Java Bean

DAO（数据库操作对象，Database Access Object）是 JDBC 下常用的模式，保存数据时它将 Java Bean 的属性拆分成正确的 SQL 语句，并保存到数据库中；读取数据时将数据从数据库中读取出来，并通过 setter 方法设置到 Java Bean 中。

DAO 模式出现之前，操作数据库的代码与业务代码都出现在 Servlet 或者 JSP 中，不利于业务代码的分离。DAO 出现后改变了这一情况，所有与数据库相关的操作都被拿到了 DAO 层实现，Servlet 或者 JSP 中只操作 Java Bean 与 DAO 层，而 DAO 层只操作数据库。

看一个使用 DAO 模式的例子。有两个 Java Bean 类：部门类（Department）与员工类（Employee），二者是一对多的关系，分别存储在部门表（tb\_department）与员工表（tb\_employee）中。部门类代码为（getter 与 setter 方法略）：

代码 12.16 Department.java

```
package com.halloweenvsfei.bean;
public class Department {
    private Integer id; // ID
    private String name; // 部门名称
}
```

员工类代码为（getter 与 setter 方法略）：

代码 12.17 Employee.java

```
package com.halloweenvsfei.bean;
import java.sql.Date;
public class Employee {
    private Department department; // 所在部门
    private Integer id; // ID
    private String name; // 姓名
    private String sex; // 性别
    private Date employedDate; // 创建日期
}
```



两张表的建表 SQL 为：

代码 12.18 init\_department.sql

```
create table tb_department ( id int auto_increment, name varchar(255), primary key (id));  
create table tb_employee ( id int auto_increment, department_id int, name varchar(255), sex varchar(255), employed_date date, primary key (id));
```

部门类的全部数据库操作均位于 DepartmentDAO 中，包括插入、修改、删除、列出所有数据等。外部可以使用 DepartmentDAO 直接操作 Department 对象。DepartmentDAO 的代码如下：

代码 12.19 DepartmentDAO.java

```
package com.halloweenvsfei.dao;  
import com.halloweenvsfei.bean.Department;  
import com.halloweenvsfei.util.DbManager;  
public class DepartmentDAO {  
    /**  
     * 插入一条 Department 记录  
     */  
    public static int insert(Department department) throws Exception {  
        String sql = "INSERT INTO tb_department (name) VALUES ( ? ) ";  
        return DbManager.executeUpdate(sql, department.getName());  
    }  
  
    /**  
     * 保存 Department  
     */  
    public static int save(Department department) throws Exception {  
        String sql = "UPDATE tb_department SET name = ? WHERE id = ? ";  
        return DbManager.executeUpdate(sql, department.getName(),  
            department.getId());  
    }  
  
    /**  
     * 删除 Department  
     */  
    public static int delete(Integer id) throws Exception {  
        String sql = "DELETE FROM tb_department WHERE id = ? ";  
        return DbManager.executeUpdate(sql, id);  
    }  
  
    /**  
     * 查找一条 Department 记录  
     */  
    public static Department find(Integer id) throws Exception {  
        String sql = "SELECT * FROM tb_department WHERE id = ? ";  
        // SQL 语句  
        Connection conn = null;                                // Connection 对象  
        PreparedStatement preStmt = null;                      // PreparedStatement  
        ResultSet rs = null;                                  // ResultSet 对象  
        try {
```



```
conn = DbManager.getConnection();           // 获取连接
preStmt = conn.prepareStatement(sql);
// 创建 PreparedStatement
preStmt.setInt(1, id);                   // 设置参数
rs = preStmt.executeQuery();             // 执行查询
if (rs.next()) {                         // 遍历数据
    Department department = new Department();
    // 封装为 Department 对象
    department.setId(id);                // 设置 id 属性
    department.setName(rs.getString("name"));
    // 设置 name 属性
    return department;                  // 返回 Department 对象
} else
    return null;                         // 如果没有数据, 返回 null
} finally {
    if (rs != null) rs.close();          // 关闭 rs
    if (preStmt != null) preStmt.close(); // 关闭 preStmt
    if (conn != null) conn.close();      // 关闭 conn
}
}

/**
 * 列出所有的 Department
 */
public static List<Department> listDepartments() throws Exception {

List<Department> list = new ArrayList<Department>(); // List 对象
String sql = "SELECT * FROM tb_department ORDER BY id DESC ";
// SQL 语句
Connection conn = null;                      // Connection 对象
PreparedStatement preStmt = null;              // PreparedStatement 对象
ResultSet rs = null;                          // ResultSet 对象
try {
    conn = DbManager.getConnection(); // 获取数据库连接
    preStmt = conn.prepareStatement(sql); // 创建 PreparedStatement
    rs = preStmt.executeQuery();       // 执行查询
    while (rs.next()) {               // 遍历 ResultSet
        Department department = new Department();
        // 封装为 Department 对象
        department.setId(rs.getInt("id")); // 设置 id 属性
        department.setName(rs.getString("name")); // 设置 name 属性
        list.add(department); // 将 Department 添加到 list 中
    }
} finally {
    if (rs != null) rs.close();          // 关闭 rs
    if (preStmt != null) preStmt.close(); // 关闭 preStmt
    if (conn != null) conn.close();      // 关闭 conn
}
return list;                                // 返回 List<Department> 对象
}
}
```

对员工里的数据库操作全部封装在 EmployeeDAO 中, 包括添加、删除、修改、列出所有员工等, 代码如下:



## 代码 12.20 EmployeeDAO.java

```
package com.halloweenvsfei.dao;
import com.halloweenvsfei.bean.Employee;
import com.halloweenvsfei.util.DbManager;
public class EmployeeDAO {
    /**
     * 插入一条记录
     */
    public static int insert(Employee employee) throws Exception {
        String sql = " INSERT INTO tb_employee " // SQL语句
            + " (department_id, name, sex, employed_date) VALUES "
            + "(?, ?, ?, ?) ";
        return DbManager.executeUpdate(sql, employee.getDepartment().
            getId(),
            employee.getName(), employee.getSex(), employee.
            getEmployedDate());
    }

    /**
     * 保存一条记录
     */
    public static int save(Employee employee) throws Exception {
        String sql = " UPDATE tb_employee " // SQL语句
            + " set department_id = ?, name = ?, sex = ?, employed_date "
            + " = ? "
            + " where id = ? ";
        return DbManager.executeUpdate(sql, employee.
            getDepartment().getId(),
            employee.getName(), employee.getSex(), employee
            .getEmployedDate(), employee.getId());
    }

    /**
     * 查找一条记录
     */
    public static Employee find(Integer id) throws Exception {
        String sql = "SELECT * FROM tb_employee WHERE id = ? ";
        // SQL语句
        Connection conn = null; // Connection 对象
        PreparedStatement preStmt = null; // PreparedStatement
        ResultSet rs = null; // ResultSet 对象
        try {
            conn = DbManager.getConnection(); // 获取 Connection
            preStmt = conn.prepareStatement(sql); // 创建 Statement
            preStmt.setInt(1, id); // 设置参数

            rs = preStmt.executeQuery(); // 开始查询
            if (rs.next()) { // 遍历 ResultSet
                Employee employee = new Employee(); // 封装成 Employee 对象
                employee.setId(id); // 设置 ID
                employee.setName(rs.getString("name")); // 设置 name
                employee.setEmployedDate(rs.getDate("employed_date"));
                // 设置时间
                employee.setSex(rs.getString("sex")); // 设置 sex
                Department d = DepartmentDAO.find(rs.getInt("department_id"));
                employee.setDepartment(d);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



```
( "department_id" );
employee.setDepartment(d); // 设置 Department 属性

        return employee; // 返回 Employee 对象
    } else {
        return null; // 如果没有, 返回 null
    }
} finally {
    if (rs != null) rs.close(); // 关闭 rs
    if (preStmt != null) preStmt.close(); // 关闭 preStmt
    if (conn != null) conn.close(); // 关闭 conn
}
}

/**
 * 列出所有的员工
 */
public static List<Employee> listAllEmployees() throws Exception {

    List<Employee> list = new ArrayList<Employee>();
    // List 对象
    String sql = "SELECT * FROM tb_employee ORDER BY id DESC ";
    // SQL 语句

    Connection conn = null; // Connection 对象
    PreparedStatement preStmt = null; // Statement 对象
    ResultSet rs = null; // ResultSet 对象

    try {
        conn = DbManager.getConnection(); // 获取连接
        preStmt = conn.prepareStatement(sql); // 创建 Statement
        rs = preStmt.executeQuery(); // 查询

        while (rs.next()) { // 遍历 rs
            Employee employee = new Employee(); // 封装为 Employee
            employee.setId(rs.getInt("id")); // 设置 ID
            employee.setName(rs.getString("name")); // 设置 name
            employee.setEmployedDate(rs.getDate("employed_date"));
            // 设置时间
            employee.setSex(rs.getString("sex")); // 设置 sex

            Department d = DepartmentDAO.find(rs.getInt
                ("department_id"));
            employee.setDepartment(d); // 设置部门属性
            list.add(employee); // 添加到 List 中
        }
    } finally {
        if (rs != null) rs.close(); // 关闭 rs
        if (preStmt != null) preStmt.close(); // 关闭 preStmt
        if (conn != null) conn.close(); // 关闭 conn
    }
    return list; // 返回 List<Employee>
}
}
```

篇幅原因, DAO 部分代码略。Java Bean 与 DAO 都建立起来后, Servlet 与 JSP 的编

程就简单多了。Servlet、JSP 层只与 Java Bean、DAO 层打交道，而不会有 JDBC 层的 Connection、Statement、ResultSet、SQL 语句等。例如添加一个 Department：

代码 12.21 addDepartment.jsp

```
<%
    request.setCharacterEncoding("UTF-8"); // 设置 request 编码
    String action = request.getParameter("action"); // 获取 action 参数

    if("add".equals(action)){ // 添加部门
        Department department = new Department(); // 实例化一个 Department
        department.setName(request.getParameter("name"));
        // 设置 name 属性

        DepartmentDAO.insert(department); // 调用 dao 保存进数据库
    }
%>
```

JSP 中只有寥寥几句代码。再如列出所有的 Employee，这是一个关系到 2 张数据表的查询，比较复杂，使用 DAO 层后代码可简化为：

代码 12.22 listEmployee.jsp

```
<%<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<jsp:directive.page import="com.helloweenvsfei.dao.EmployeeDAO"/>
<jsp:directive.page import="java.util.List"/>
<%
    List employeeList = EmployeeDAO.listAllEmployees();
    // 使用 DAO 列出所有的员工
    request.setAttribute("employeeList", employeeList); // 放到 request 中
%>
<form action="operatePerson.jsp" method="get">
    <table border="1" style="width: 100%; border-collapse: collapse;">
        <tr style="background-color: #CCCCCC; height: 30px; border-bottom: 1px solid black;">
            <th style="width: 10%;">ID姓名部门性别入职日期操作${employee.name}${employee.department.name}${employee.sex}${employee.employedDate}
```

```
        onclick="return confirm('确定删除?')>删除</a>
    </td>
</tr>
</c:forEach>
</table>
</form>
```

可以结合 JSTL 标签输出。运行效果如图 12.9 所示。

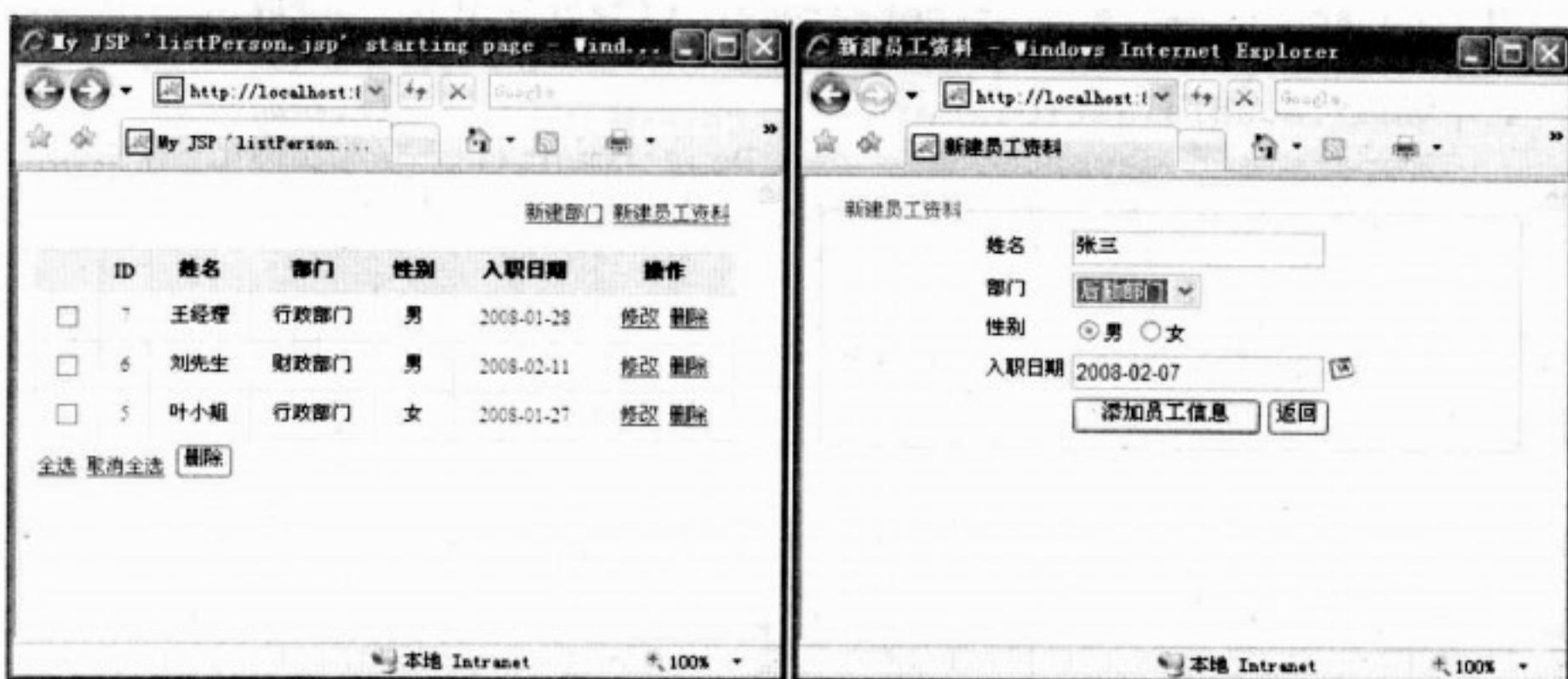


图 12.9 DAO 模式下的部门员工管理

**提示：**实践证明，DAO 是个很有用的设计模式，虽然实现一套 DAO 层代码量很大，但是可重用性、可组件化性能都比较好。甚至直到现在，在 Hibernate、EJB3 等数据库 ORM 框架出现后，DAO 模式仍然被广泛地使用。

### 12.5.2 事务实例：转账

数据库是具有事务性的，这是数据库不同于其他存储方式的区别之一。一个事务内可以执行多个操作，这些操作要么全部执行成功，要么全部执行失败。举个例子，用户 A 要向用户 B 账号内转账 200 元钱，该转账业务要分两步：第一步，A 账号内减 200 元；第二步，B 账号内加 200 元。这两步操作必须都执行成功，或者都没有执行成功，否则情理上就讲不通了。这就是事务性。

事务有两个结果，提交（Commit）与回滚（Rollback）。如果两步转账都没有错误，则可以提交，转账结果保存进数据库。如果其中任何一步出错了，则必须回滚，数据库不会有任何的改动。本例中的账号表为 tb\_currency，初始化 SQL 为：

代码 12.23 init\_currency.sql

```
create table tb_currency
( id int auto_increment,
  account varchar(255) unique,
  currency double,
  last_modified timestamp,
  primary key (id)
);
----- 创建表 TB_CURRENCY
```

```
insert into tb_currency ( account, currency, last_modified ) values ( 'A',  
1000, current_timestamp );  
insert into tb_currency ( account, currency, last_modified ) values ( 'B',  
0, current_timestamp );
```

A 账号中有 1000 元钱，B 账号中为 0 元钱。下面使用程序模拟两个账号之间的转账。如果账号余额小于转账金额，则转账失败。示例程序如下：

代码 12.24 listCurrency.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>  
  
<div style="padding: 2px; text-align: right; ">  
    <a href="#">只显示余额> |  
    <a href="#">A 向 B 转账 200 元> |  
    <a href="#">B 向 A 转账 200 元>  
</div>  
  
<%  
    String action = request.getParameter("action"); // 获取 action 参数  
    if("a2b".equals(action)){ // 如果是从 A 到 B 转账  
        out.println("业务: A 向 B 转账 200 元。<br/>");  
        Connection conn = null; // Connection 对象  
        Statement stmt = null; // Statement 对象  
        try{  
            conn = DbManager.getConnection(); // 获取 Connection  
            conn.setAutoCommit(false); // 将自动提交设为 false  
            stmt = conn.createStatement(); // 创建 Statement 对象  
  
            int result1 = stmt.executeUpdate( // A 账号扣款 200 元  
                " UPDATE tb_currency SET currency = currency - 200,  
                last_modified = current_timestamp WHERE account =  
                'A' and currency >= 200 ");  
  
            out.println("A 账号扣款 200 元, 结果: " + (result1==1 ?  
                "成功" : "失败") + "<br/>"); // 输出扣款是否成功  
  
            int result2 = stmt.executeUpdate( // B 账号充值 200 元  
                " UPDATE tb_currency SET currency = currency + 200,  
                last_modified = current_timestamp WHERE account =  
                'B' ");  
  
            out.println("B 账号存款 200 元, 结果: " + (result2==1 ?  
                "成功" : "失败") + "<br/>"); // 输出充值是否成功  
  
            if(result1 == 1 && result2 == 1){ // 扣款、充值均成功  
                conn.commit(); // 提交事务  
                out.println("转账成功, 事务提交。<br/>");  
            }  
            else{ // 扣款或者充值失败  
                conn.rollback(); // 事务回滚, 修改无效  
                out.println("转账失败, 事务回滚。<br/>");  
            }  
        }finally{
```

```
        if(stmt != null)    stmt.close(); // 关闭 stmt
        if(conn != null)   conn.close(); // 关闭 conn
    }
}
else if("b2a".equals(action)){
    // 类同于A向B转账，略
}
%>
账号余额: <%          // 显示表 tb_currency 中的内容，代码略 %>
```

注意粗体代码。先使用 `conn.setAutoCommit(false)` 将自动提交设置为 `false`（默认为 `true`，即执行完一条 SQL 后便自动提交），再执行多条 SQL 语句。当多条 SQL 语句执行完毕，确认没有错误发生时，再执行 `conn.commit()` 提交。如果有异常抛出或者返回值不对，执行 `conn.rollback()` 回滚。

由于 A 账号与 B 账号总共只有 1000 元钱，连续转账就会因为余额不足导致转账失败，这时事务就会回滚。运行效果如图 12.10 所示。

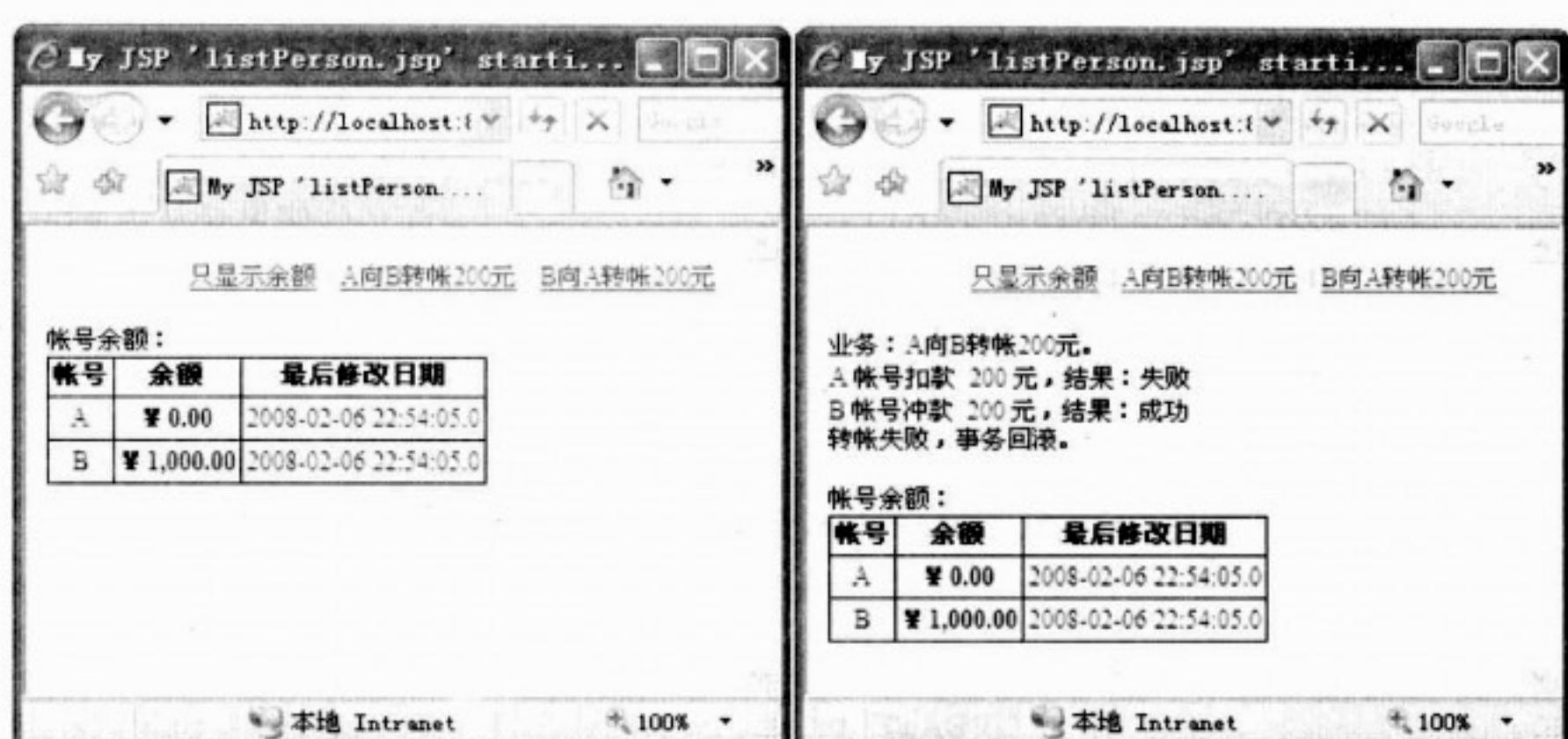


图 12.10 转账事务提交与回滚

**提示：**事务开始前要使用 `conn.setAutoCommit(false)` 将自动提交置为 `false`，事务结束后要使用 `conn.commit()` 显示的提交数据。MySQL 控制台也可以使用 `set autoComit=false` 开启事务，结束后执行 `commit` 提交或者 `rollback` 回滚。

### 12.5.3 抛出异常自动回滚

设置自动提交为 `false` 后，事务的提交必须执行 `conn.commit()`，而事务的回滚不一定要显式地执行 `conn.rollback()`。如果程序最后没有执行 `conn.commit()`，事务也会回滚。一般是直接抛出异常，终止本段程序的正常运行，例如：

```
try{
    conn = DbManager.getConnection();
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    int i = stmt.executeUpdate(" ... ");
    int j = stmt.executeUpdate(" ... ");
```

```
if(i != 1 || j != 1)
    throw new Exception("数据更新失败。");
conn.commit();
}finally{
    stmt.close();
    conn.close();
}
```

如果抛出了异常，该事务就会回滚。

**注意：**事务性是数据库区分与其他存储技术的重要特征之一。普通文件也能存储数据，但这种存储技术下，同一个数据可能被多个线程同时修改，因此有潜在的数据不一致性。基本上所有的数据库编程都会涉及事务编程。

#### 12.5.4 存储二进制数据

除了能存储文本、数字、日期等数据，数据库还可以存储二进制数据。MySQL 提供了 BLOB 类型字段用于存储二进制数据。用户上传文件后，可以将文件保存到硬盘上，只把文件名、路径等保存进数据库，也可以把文件内容直接存储到数据库中。

下面的例子将上传的附件保存在数据库中。在加载 listBlob.jsp 时会执行 SQL 语句创建附件表 tb\_attachment，包括附件名称、附件类型、附件长度、内容、上传者 IP 地址、上传时间等字段。其中附件内容保存到 blob 类型字段中，通过 InputStream 写入数据库，代码如下：

代码 12.25 listBlob.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<jsp:directive.page import="com.helloweenvsfei.util.DbManager"/>
<jsp:directive.page
import="org.apache.commons.fileupload.DiskFileUpload"/>
<jsp:directive.page import="org.apache.commons.fileupload.FileItem"/>
<%
public void init() {                                // 加载 JSP 时执行 SQL 语句创建表
    try{
        DbManager.executeUpdate("CREATE TABLE IF NOT EXISTS
            tb_attachment ( id int auto_increment, filename varchar(255),
            filetype varchar(255), size int, content LONGBLOB, ip
            varchar(255), date timestamp, primary key (id) );");
    }catch(Exception e){
        e.printStackTrace();
    }
}
%>
<%
if("POST".equalsIgnoreCase(request.getMethod())){ // 如果是 POST 表单
    DiskFileUpload diskFileUpload = new DiskFileUpload();
    // 使用 DiskFileUpload
    diskFileUpload.setHeaderEncoding("UTF-8");      // 设置编码
    List<FileItem> list = diskFileUpload.parseRequest(request);
    // 解析上传的数据
    for(FileItem fileItem : list){                  // 遍历所有上传的域
        if(!fileItem.isFormField()){                // 如果是文件域
            ...
        }
    }
}
%>
```



### 第3篇 高级篇

```
String filename = fileItem.getName().replace("\\\", "/");
// 文件路径
filename = filename.substring(filename.lastIndexOf("/") +
1); // 获取文件名

Connection conn = null; // Connection 对象
PreparedStatement preStmt = null; // PreparedStatement 对象
try{
    conn = DbManager.getConnection(); // 获取 Connection
    preStmt = conn.prepareStatement( // 预编译 SQL 语句
        "INSERT INTO tb_attachment "
        + " (filename,filetype,size,content,ip,date)
        values(?,?,?,?,?,?) ");

    preStmt.setString(1, filename); // 设置附件名称
    preStmt.setString(2, fileItem.getContentType()); // 设置附件类型
    preStmt.setInt(3, (int)fileItem.getSize()); // 附件长度
    preStmt.setString(5, request.getRemoteAddr()); // 上传者 IP 地址

    // 设置上传时间
    preStmt.setTimestamp(6, new Timestamp(System.
        currentTimeMillis()));

    preStmt.setBinaryStream(4, // 设置附件内容
        fileItem.getInputStream(), // 附件输入流
        (int)fileItem.getSize()); // 附件长度

    preStmt.executeUpdate(); // 保存进数据库
}finally{
    if(preStmt != null) preStmt.close(); // 关闭 preStmt
    if(conn != null) conn.close(); // 关闭 conn
}
}

}

}

}

%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <body>
        <a href="javascript:location=location;">刷新附件列表</a>
        <%
            Connection conn = null; // Connection 对象
            Statement stmt = null; // Statement 对象
            ResultSet rs = null; // ResultSet 对象
            try{
                conn = DbManager.getConnection(); // 获取 Connection
                stmt = conn.createStatement(); // 创建 Statement
                rs = stmt.executeQuery("SELECT * FROM tb_attachment ORDER BY id
DESC");

                out.println("<table>");
                out.println("  <tr>");
```

## 第 12 章 JDBC 详解

```
out.println("      <th>ID</th>");  
out.println("      <th>File Name</th>");  
out.println("      <th>File Type</th>");  
out.println("      <th>Size</th>");  
out.println("      <th>IP</th>");  
out.println("      <th>Date</th>");  
out.println("      <th>Operation</th>");  
out.println("      <tr>");  
  
while(rs.next()) { // 遍历输出附件信息  
    out.println("      <tr>");  
    out.println("          <td>" + rs.getInt("id") + "</td>");  
    out.println("          <td>" + rs.getString("filename") +  
    "</td>");  
    out.println("          <td>" + rs.getString("filetype") +  
    "</td>");  
    out.println("          <td>" + rs.getInt("size") + "</td>");  
    out.println("          <td>" + rs.getString("ip") + "</td>");  
    out.println("          <td>" + rs.getTimestamp("date") +  
    "</td>");  
    out.println("          <td><a href='download.jsp?id=" +  
    rs.getInt("id") + "'>下载</a></td>");  
    // 输出附件的连接。附件内容将通过 download.jsp 从数据库读出来  
    out.println("      </tr>");  
}  
out.println("</table>");  
}finally{  
    if(rs != null) rs.close(); // 关闭 rs  
    if(stmt != null) stmt.close(); // 关闭 stmt  
    if(conn != null) conn.close(); // 关闭 conn  
}  
%>  
  
<form action="${pageContext.request.requestURI}" enctype="multipart/  
form-data" method="post">  
    <input name="file" type="file" /><input type="submit" value="  
    开始上传 ">  
</form>  
</body>  
</html>
```

运行效果如图 12.11 所示。



图 12.11 存储二进制数据

提示: BLOB 用于存储二进制数据, Text 用户存储大文本, 它们的长度都不能大于 65536 个字节。如果数据过长, 可以使用 LONGBLOB、LONGTEXT。

### 12.5.5 读取二进制数据

文件写入数据库中, 还需要专门的程序将数据读出来。单击附件列表的“下载”链接会请求“download.jsp?id=1”, 该程序会将 id 为 1 的附件内容读取出来, 发送到客户端浏览器。读取文件内容同样要使用 Stream, 代码如下:

代码 12.26 download.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<
    out.clear();                                // 清空一切输出
    int id = Integer.parseInt(request.getParameter("id")); // 获取附件的 ID

    Connection conn = null;                      // Connection 对象
    PreparedStatement preStmt = null;             // PreparedStatement 对象
    ResultSet rs = null;                         // ResultSet 对象

    try{
        conn = DbManager.getConnection();          // 获取连接
        preStmt = conn.prepareStatement("select * from tb_attachment where
                                         id = ? ");
        preStmt.setInt(1, id);                    // 设置参数
        rs = preStmt.executeQuery();              // 执行查询

        if(rs.next()){                          // 如果有数据
            response.reset();                  // 重置 response
            response.setContentType(rs.getString("filetype"));
            // 设置输出的文件类型
            response.setContentLength(rs.getInt("size"));
            // 设置输出长度

            InputStream ins = null;           // 输入流, 用于读取数据库
            OutputStream ous = null;         // 输出流, 用户输出到客户端

            try{
                ins = rs.getBinaryStream("content");
                // 获取 ResultSet 的二进制流
                ous = response.getOutputStream();
                // 获取 response 的输出流
                byte[] b = new byte[1024];      // 缓存数组
                int len = 0;                   // 实际缓存长度
                while((len = ins.read(b)) != -1){ // 循环读入到缓存数组中
                    ous.write(b, 0, len);       // 循环输出到客户端
                }
            }finally{
                if(ous != null) ous.close();    // 关闭输出流
                if(ins != null) ins.close();   // 关闭输入流
            }
        }
    }
}
```

```
        else{
            out.println("没有找到附件: " + id);           // 如果没有找到附件记录
        }
    }finally{
    if(rs != null) rs.close();                      // 关闭 rs
    if(preStmt != null) preStmt.close();            // 关闭 preStmt
    if(conn != null) conn.close();                  // 关闭 conn
}
%>
```

## 12.5.6 数据源（连接池）

先前的 JDBC 编程中，每操作一次数据库，都会经过下面的过程：创建 Connection、创建 Statement 对象、获取 ResultSet、销毁 ResultSet、销毁 Statement、断开 Connection。即每操作一次数据库，都会创建连接、断开连接。

在实际的使用中，创建与断开 Connection 都会消耗一定的时间、IO 资源，在大量的并发访问时尤其明显。为了避免频繁地创建、断开数据库连接，工程师们提出了数据源技术（Data Source），也称为连接池（DBCP，Database Connection Pool）。

要操作数据库时，程序并不是直接创建 Connection，而是向连接池“申请”一个 Connection。如果连接池中有空闲的 Connection，则返回该 Connection，否则创建新的 Connection。使用完毕，Servlet 会“释放”该 Connection。连接池会将该 Connection 回收，并交付其他的线程使用，达到减少创建、断开连接次数的目的。

数据源一般实现自 javax.sql.DataSource 接口。Spring、Struts、Hibernate 等框架都有自己的数据源实现，Tomcat 中也内置了数据源支持。Tomcat 使用 Jakarta-Commons Database Connection Pool 作为数据源实现，使用时只需按照 Tomcat 文档配置即可，例如：

代码 12.27 context.xml

```
<Context cookies="true">
    <Resource name="jdbc/databaseWeb"
        auth="Container"
        type="javax.sql.DataSource"
        maxActive="100" maxIdle="30" maxWait="10000"
        username="root" password="admin"
        driverClassName="com.mysql.jdbc.Driver"

        url="jdbc:mysql://localhost:3306/databaseWeb?characterEncoding=utf
        -8"/>
</Context>
```

 注意：数据源可以配置在 tomcat/conf/server.xml 中，也可以配置在 tomcat/conf/context.xml 中。这里将数据源配置在 tomcat/conf/context.xml 中。注意此时要把 MySQL 驱动放到 Tomcat 全局 lib 里（例如 tomcat-6.0.13/lib）下面，而不能放在本 web 应用下面。

然后要在 Web 程序的 web.xml（不是 tomcat/conf/web.xml）中配置数据源引用，这样才能在 Web 程序中使用该数据源。配置代码如下：

## 代码 12.28 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/databaseWeb</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

数据源的名称为“jdbc/databaseWeb”。这是一个 JNDI 资源，Java 程序中可以通过查找该 JNDI 资源获取该数据源。例如 JSTL 代码：

## 代码 12.29 test.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%-- 使用 JNDI 数据源 查询 SQL 语句 --%>
<sql:query var="rs" dataSource="jdbc/databaseWeb">
  select id, name, sex from tb_employee
</sql:query>

<%-- 遍历 ResultSet 显示数据 --%>
<c:forEach var="row" items="${rs.rows}">
  ${row['id']}, ${row['name']}, ${row['sex']}<br />
</c:forEach>
```

或者使用 Java 代码查找 JNDI：

## 代码 12.30 test.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<jsp:directive.page import="javax.sql.DataSource" />
<jsp:directive.page import="javax.naming.Context"/>
<jsp:directive.page import="javax.naming.InitialContext"/>
<jsp:directive.page import="java.sql.Connection"/>
<jsp:directive.page import="java.sql.Statement"/>
<jsp:directive.page import="java.sql.ResultSet"/>
<%
  Context initContext = new InitialContext(); // 实例化一个 InitialContext
  Context envContext = (Context) initContext.lookup("java:/comp/env");
  // 获取所有的资源

  DataSource ds = (DataSource) envContext.lookup("jdbc/databaseWeb");
  // 获取 JNDI 数据源
  Connection conn = ds.getConnection();           // 申请一个 Connection
```

```
Statement stmt = conn.createStatement(); // 创建 Statement
ResultSet rs = stmt.executeQuery("select id, name, sex from
tb_employee"); // 查询数据库

rs.close(); // 关闭 rs
stmt.close(); // 关闭 stmt
conn.close();
// 关闭 Connection。实际上是释放 Connection 到数据源中，而不是关闭
%>
```

提示：也可以直接注射到 Servlet 中。方法见 Servlet 相关章节。

## 12.6 本 章 小 结

Java 程序通过 JDBC 执行 SQL 语句操作数据库。每个数据库都需要特定的驱动。操作数据库都遵循下面步骤：创建数据库连接、创建 Statement 或者 PreparedStatement 对象、查询返回 ResultSet 对象或者执行 SQL 语句更新数据、销毁 ResultSet、销毁 Statement 或者 PreparedStatement 对象、断开数据库连接。

执行 SQL 语句时，可以批处理同时执行多条 SQL 语句。往数据库写数据时，一般需要开启事务。在事务提交之前，如果某条 SQL 语句执行失败导致事务回滚，则所有的 SQL 语句都会回滚。

数据源（连接池）会维护打开的 Connection，利用一定的算法进行优化，而不会频繁地创建、关闭 Connection。数据源一般配置在容器中，一般的容器例如 Tomcat 都支持配置数据源。配置数据源需要将驱动复制到 Tomcat 的 lib 中，而不是 Web 程序的 lib 中。