JAVA WEB 核心编程技能掌握

1. Servlet 综合应用

要求： 编写一个 Servlet, 能够检索并处理请求中的参数, 获取初始化参数, 在四大作用域范围内存取数据, 按要求进行页面响应、请求的转发或响应重定向; 能够在 `web.xml` 中配置一个 Servlet (包括初始化参数) 及其 Mapping。

答案:

`ComprehensiveServlet.java`

```java
package com.example.servlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

public class ComprehensiveServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // --- 1. 获取初始化参数 ---
        // 从 web.xml 的 <init-param> 中获取配置的管理员邮箱
        String adminEmail = this.getServletConfig().getInitParameter("adminEmail");

        // --- 2. 获取请求参数 ---
        // 从 URL 获取参数, 例如: ?username=Alice&action=forward
        String username = request.getParameter("username");
        String action = request.getParameter("action");

        // --- 3. 在四大作用域中存取数据 ---
        // a. Request 作用域 (一次请求有效)
        request.setAttribute("requestData", "这是存储在 Request 中的数据");

        // b. Session 作用域 (一次会话有效)
        HttpSession session = request.getSession();
        session.setAttribute("sessionData", "你好, " + username + "! 这是你的会话数据。");

        // c. Application (ServletContext) 作用域 (整个应用生命周期有效)
        ServletContext application = this.getServletContext();
        application.setAttribute("applicationData", "这个数据对所有用户可见。");
        application.setAttribute("adminEmailInfo", "管理员邮箱是: " + adminEmail);

        // Page 作用域只能在 JSP 页面中使用, 此处无法演示。

```

```
        // --- 4. 页面响应、转发或重定向 ---
        response.setContentType("text/html;charset=UTF-8"); // 设置响应类型和编码

        if ("forward".equalsIgnoreCase(action)) {
            // a. 请求转发: URL 地址不变, 共享 request 数据
            System.out.println("执行请求转发到 /result.jsp");
            request.getRequestDispatcher("/result.jsp").forward(request, response);
        } else if ("redirect".equalsIgnoreCase(action)) {
            // b. 响应重定向: URL 地址改变, 不共享 request 数据
            // 通常重定向前会把需要传递的简单数据存入 session
            session.setAttribute("redirectMessage", "你被重定向了! ");
            System.out.println("执行响应重定向到 /result.jsp");
            response.sendRedirect(request.getContextPath() + "/result.jsp");
        } else {
            // c. 直接响应: 直接向浏览器输出内容
            System.out.println("执行直接响应");
            response.getWriter().println("<h1>直接响应</h1>");
            response.getWriter().println("<p>你好, " + username + "!</p>");
            response.getWriter().println("<p>你的请求动作为: " + action + "</p>");
        }
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 为了方便测试, 让 POST 请求也执行和 GET 相同的逻辑
        doGet(req, resp);
    }
}
```

`result.jsp` (用于接收转发和重定向)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>结果页面</title>
</head>
<body>
    <h1>数据显示页面</h1>

    <%-- pageContext 作用域仅在此页面有效 --%>
    <% pageContext.setAttribute("pageData", "这是 Page 作用域的数据"); %>

    <h3>四大作用域数据:</h3>
    <p><strong>Page Scope:</strong> ${pageScope.pageData}</p>
    <p><strong>Request Scope:</strong> ${requestScope.requestData} (如果是重定向, 这里会是空的)</p>
    <p><strong>Session Scope:</strong> ${sessionScope.sessionData}</p>
    <p><strong>Application Scope:</strong> ${applicationScope.applicationData}</p>

    <hr/>
```

```
    <h3>其他信息:</h3>
    <p><strong>从初始化参数获取的信息:</strong> ${applicationScope.adminEmailInfo}</p>
    <p><strong>重定向带来的消息:</strong> ${sessionScope.redirectMessage}</p>
</body>
</html>
```

`web.xml` 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!-- 1. 声明 Servlet -->
    <servlet>
        <!-- Servlet 的内部名称 -->
        <servlet-name>ComprehensiveServlet</servlet-name>
        <!-- Servlet 类的完整路径 -->
        <servlet-class>com.example.servlet.ComprehensiveServlet</servlet-class>

        <!-- 配置初始化参数 -->
        <init-param>
            <param-name>adminEmail</param-name>
            <param-value>admin@example.com</param-value>
        </init-param>
    </servlet>

    <!-- 2. 映射 Servlet 的访问 URL -->
    <servlet-mapping>
        <!-- 必须与上面声明的 servlet-name 一致 -->
        <servlet-name>ComprehensiveServlet</servlet-name>
        <!-- 客户端浏览器访问的 URL 模式 -->
        <url-pattern>/process</url-pattern>
    </servlet-mapping>
</web-app>
```

---

2. JavaBean 与 DAO 类的编写

要求： 能够按要求熟练编写封装数据的 JavaBean 类和封装数据库操作的 DAO 类。

答案：

`Product.java` (JavaBean)

```
package com.example.bean;

import java.io.Serializable;

// 1. 实现 Serializable 接口，推荐做法
```

```java
public class Product implements Serializable {

    // 2. 属性私有化
    private int id;
    private String name;
    private double price;

    // 3. 提供公共的无参构造方法
    public Product() {
    }

    // (可选) 提供带参数的构造方法
    public Product(int id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // 4. 为所有私有属性提供 public 的 getter 和 setter 方法
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}
```

`ProductDAO.java` (DAO 类)

```java
package com.example.dao;

import com.example.bean.Product;
import com.example.util.DBUtil; // 假设有一个数据库连接工具类

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```java
import java.util.ArrayList;
import java.util.List;

// DAO 类用于封装所有对 Product 表的数据库操作
public class ProductDAO {

    // 根据 ID 查询单个产品
    public Product getProductById(int id) {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        Product product = null;

        try {
            conn = DBUtil.getConnection();
            String sql = "SELECT id, name, price FROM products WHERE id = ?";
            ps = conn.prepareStatement(sql);
            ps.setInt(1, id);
            rs = ps.executeQuery();

            if (rs.next()) {
                product = new Product();
                product.setId(rs.getInt("id"));
                product.setName(rs.getString("name"));
                product.setPrice(rs.getDouble("price"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            DBUtil.close(conn, ps, rs);
        }
        return product;
    }

    // 添加新产品
    public void addProduct(Product product) {
        Connection conn = null;
        PreparedStatement ps = null;

        try {
            conn = DBUtil.getConnection();
            String sql = "INSERT INTO products (name, price) VALUES (?, ?)";
            ps = conn.prepareStatement(sql);
            ps.setString(1, product.getName());
            ps.setDouble(2, product.getPrice());
            ps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            DBUtil.close(conn, ps, null);
        }
    }

    // ... 其他方法如 updateProduct, deleteProduct, getAllProducts 等
}
```

3. JSP 指令、动作和小脚本

要求： 能够熟练编写 JSP 指令、动作和小脚本；会使用 JSP 动作调用 JavaBean 模型, 给 JavaBean 的属性赋值、输出 JavaBean 的属性值。

答案：

`bean-demo.jsp`

```jsp
<%-- 1. JSP 指令 (Directives) --%>
<%-- page 指令: 设置页面属性, 如编码、导入 Java 类等 --%>
<%@ page contentType="text/html;charset=UTF-8" language="java"
import="java.util.Date, com.example.bean.Product" %>
<%-- taglib 指令: 引入标签库 (这里以 JSTL 核心库为例) --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
    <title>JSP 综合演示</title>
</head>
<body>
    <%-- 2. JSP 动作 (Actions) --%>
    <%-- useBean: 查找或创建一个 JavaBean 实例 --%>
    <%-- id: Bean 的名字; class: Bean 的完整类名; scope: Bean 的作用域 --%>
    <jsp:useBean id="myProduct" class="com.example.bean.Product"
scope="page"/>

    <%-- setProperty: 设置 Bean 的属性 --%>
    <%-- name: 对应 useBean 的 id; property: 属性名; value: 要设置的值 --%>
    <jsp:setProperty name="myProduct" property="id" value="101"/>
    <jsp:setProperty name="myProduct" property="name" value="笔记本电脑"/>
    <jsp:setProperty name="myProduct" property="price" value="5999.99"/>

    <h2>产品信息 (通过 JSP Actions)</h2>
    <p>产品ID: <jsp:getProperty name="myProduct" property="id"/></p>
    <p>产品名称: <jsp:getProperty name="myProduct" property="name"/></p>
    <p>产品价格: <jsp:getProperty name="myProduct" property="price"/></p>

    <hr/>

    <%-- 3. 小脚本 (Scriptlets & Expressions) --%>
    <%-- 警告: 在现代 MVC 开发中应尽量避免使用小脚本, 推荐使用 EL 和 JSTL --%>
    <%
        // Scriptlet: 嵌入任意 Java 代码
        String welcomeMessage = "欢迎使用 JSP 小脚本!";
        if (myProduct.getPrice() > 5000) {
            welcomeMessage += " 这是一款高端产品!";
        }
    %>

    <h2>小脚本演示</h2>
    <p>
        <%-- Expression: 用于输出表达式的值到页面 --%>
```

```
        <%= welcomeMessage %>
    </p>
    <p>
        当前时间: <%= new Date() %>
    </p>
</body>
</html>
```

---

4. 传统 JDBC 数据库访问

要求： 会写代码使用传统方法进行数据库的连接与访问, 包括加载驱动程序、建立连接对象、创建语句对象、执行 SQL 语句, 获取执行结果, 访问结果集数据等。

答案:
(此答案与第 2 点中的 `ProductDAO.java` 示例相同，此处为强调步骤再次列出并详细注释)

`TraditionalJDBCExample.java` (在一个独立的 main 方法中演示)

```java
import com.example.bean.Product;
import java.sql.*;

public class TraditionalJDBCExample {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            // 1. 加载驱动程序
            Class.forName("com.mysql.cj.jdbc.Driver");

            // 2. 建立连接对象
            String url = "jdbc:mysql://localhost:3306/testdb?useSSL=false&serverTimezone=UTC";
            String user = "root";
            String password = "your_password"; // 替换为你的密码
            conn = DriverManager.getConnection(url, user, password);

            // 3. 创建语句对象
            stmt = conn.createStatement();

            // 4. 执行 SQL 语句
            String sql = "SELECT id, name, price FROM products WHERE id = 101";
            rs = stmt.executeQuery(sql);

            // 5. 访问结果集数据
            if (rs.next()) {
                Product product = new Product();
                product.setId(rs.getInt("id"));
                product.setName(rs.getString("name"));
                product.setPrice(rs.getDouble("price"));
                System.out.println("查询成功: " + product.getName());
            }
```

```
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        } finally {
            // 6. 关闭资源 (顺序: ResultSet -> Statement -> Connection)
            try {
                if (rs != null) rs.close();
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

---

5. 使用数据源进行数据库访问

要求： 会配置数据源, 会写代码使用数据源获取数据库的连接、创建语句对象、执行 SQL 语句, 获取执行结果, 访问结果集数据等。

答案:

a. 在 Tomcat 的 `conf/context.xml` 中配置数据源

```
<Context>
    <!-- ... 其他配置 ... -->
    <Resource name="jdbc/myAppDB"
              auth="Container"
              type="javax.sql.DataSource"
              username="root"
              password="your_password"
              driverClassName="com.mysql.cj.jdbc.Driver"
              url="jdbc:mysql://localhost:3306/testdb?
useSSL=false&serverTimezone=UTC"
              maxTotal="20"
              maxIdle="10"
              maxWaitMillis="-1"/>
</Context>
```

b. 使用数据源获取连接的工具类 `DataSourceUtil.java`

```
package com.example.util;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

public class DataSourceUtil {
    private static DataSource dataSource = null;
```

```
12
13      static {
14          try {
15              // 通过 JNDI 查找已配置的数据源
16              Context initContext = new InitialContext();
17              Context envContext = (Context)
    initContext.lookup("java:/comp/env");
18              dataSource = (DataSource) envContext.lookup("jdbc/myAppDB");
19          } catch (NamingException e) {
20              throw new RuntimeException("Cannot find DataSource!", e);
21          }
22      }
23
24      // 从连接池获取一个连接
25      public static Connection getConnection() throws SQLException {
26          return dataSource.getConnection();
27      }
28
29      // 关闭资源的方法可以复用
30  }
```

c. DAO 类改用数据源

```
1   // 在 ProductDAO 中，只需要修改获取连接的方式即可
2   public Product getProductById(int id) {
3       // ...
4       try {
5           // 原来的: conn = DBUtil.getConnection();
6           // 现在: 从数据源获取连接
7           conn = DataSourceUtil.getConnection();
8           // ... 后续代码完全一样 ...
9       } catch (SQLException e) {
10          // ...
11      } finally {
12          // ...
13      }
14      return product;
15  }
```

---

6. EL 表达式的应用

要求： 会使用 EL 访问 JSP 隐含变量、作用域变量、访问 JavaBeans 属性等。

答案：

a. `ELServlet.java` (用于准备数据)

```
// ...
// 在 Servlet 的 doGet 方法中
request.setAttribute("requestMessage", "Hello from Request");
request.getSession().setAttribute("sessionUser", new Product(1, "手机",
2999));
this.getServletContext().setAttribute("appCounter", 12345);

// 转发到 JSP 页面
request.getRequestDispatcher("/el-demo.jsp").forward(request, response);
// ...
```

b. `el-demo.jsp`

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="com.example.bean.Product" %>
<html>
<head>
    <title>EL 表达式演示</title>
</head>
<body>
    <h2>访问作用域变量</h2>
    <%-- EL 会自动从 page -> request -> session -> application 顺序查找 --%>
    <p>Request Message: ${requestMessage}</p>

    <h2>访问 JavaBean 属性</h2>
    <%-- 访问 sessionUser 对象的 name 属性 (实际调用 getName() 方法) --%>
    <p>用户名: ${sessionUser.name}</p>
    <p>价格: ${sessionUser.price}</p>

    <h2>访问 JSP 隐含变量 (隐式对象)</h2>
    <%-- pageContext 是一个常用入口,可以访问其他对象 --%>
    <p>Web 应用根路径: ${pageContext.request.contextPath}</p>
    <p>Session ID: ${pageContext.session.id}</p>

    <h2>EL 运算符</h2>
    <p>产品价格 + 100: ${sessionUser.price + 100}</p>
    <p>价格是否大于 2000: ${sessionUser.price > 2000}</p>
    <p>用户名是否为空: ${empty sessionUser.name}</p>
</body>
</html>
```

---

7. Web 事件监听器

要求: 会编写 Web 事件监听程序, 对 Web 的应用上下文事件、请求事件、会话事件进行监听; 会在 `web.xml` 中注册监听器。

答案:

`MyAppContextListener.java` (应用上下文监听器)

```java
package com.example.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class MyAppContextListener implements ServletContextListener {
    // 应用启动时调用
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("Web 应用已启动. sce.getServletContext(): " + sce.getServletContext());
    }
    // 应用关闭时调用
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("Web 应用即将关闭.");
    }
}
```

`MySessionListener.java` (会话监听器)

```java
package com.example.listener;

import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class MySessionListener implements HttpSessionListener {
    // Session 创建时调用
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("一个新的 Session 已创建, ID: " + se.getSession().getId());
    }
    // Session 销毁时调用
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("一个 Session 已销毁, ID: " + se.getSession().getId());
    }
}
```

`MyRequestListener.java` (请求监听器)

```java
package com.example.listener;

import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;

public class MyRequestListener implements ServletRequestListener {
    // 请求初始化(到达服务器)时调用
    @Override
    public void requestInitialized(ServletRequestEvent sre) {
```

```
        System.out.println("一个请求已到达, 来自 IP: " +
sre.getServletRequest().getRemoteAddr());
    }
    // 请求销毁(响应已发送)时调用
    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        System.out.println("一个请求已处理完毕.");
    }
}
```

`web.xml` 中注册所有监听器

```
<web-app ...>
    <!-- ... servlet 配置 ... -->
    <listener>
        <listener-class>com.example.listener.MyAppContextListener</listener-
class>
    </listener>
    <listener>
        <listener-class>com.example.listener.MySessionListener</listener-
class>
    </listener>
    <listener>
        <listener-class>com.example.listener.MyRequestListener</listener-
class>
    </listener>
</web-app>
```

---

8. Web 过滤器

要求: 会编写 Web 过滤器, 会对过滤器的初始化参数进行读取, 对指定请求模式进行过滤处理; 会在 `web.xml` 中配置过滤器及初始化参数等。

答案:

`PerformanceLogFilter.java`

```
package com.example.filter;

import javax.servlet.*;
import java.io.IOException;

public class PerformanceLogFilter implements Filter {

    private String filterName;

    // 1. 初始化过滤器时调用, 用于读取配置参数
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // 读取 web.xml 中配置的初始化参数
        this.filterName = filterConfig.getInitParameter("filterName");
        System.out.println("过滤器 [" + this.filterName + "] 初始化...");
    }
```

```
17
18      // 2. 每次请求匹配时调用
19      @Override
20      public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
21              throws IOException, ServletException {
22
23          long startTime = System.currentTimeMillis();
24          System.out.println("[" + this.filterName + "] 请求处理开始...");
25
26          // 将请求传递给过滤器链中的下一个组件 (下一个过滤器或目标 Servlet)
27          chain.doFilter(request, response);
28
29          long endTime = System.currentTimeMillis();
30          System.out.println("[" + this.filterName + "] 请求处理结束. 耗时: " +
(endTime - startTime) + "ms");
31      }
32
33      // 3. 过滤器销毁时调用
34      @Override
35      public void destroy() {
36          System.out.println("过滤器 [" + this.filterName + "] 已销毁.");
37      }
38  }
```

`web.xml` 中配置过滤器

```
1   <web-app ...>
2       <!-- ... 其他配置 ... -->
3
4       <!-- 1. 定义过滤器 -->
5       <filter>
6           <filter-name>PerformanceLogFilter</filter-name>
7           <filter-class>com.example.filter.PerformanceLogFilter</filter-class>
8
9           <!-- 为过滤器设置初始化参数 -->
10          <init-param>
11              <param-name>filterName</param-name>
12              <param-value>PerformanceLogger</param-value>
13          </init-param>
14      </filter>
15
16      <!-- 2. 映射过滤器 -->
17      <filter-mapping>
18          <filter-name>PerformanceLogFilter</filter-name>
19          <!--
20              指定请求模式
21              /* 表示拦截所有请求
22              /admin/* 表示拦截所有以 /admin/ 开头的请求
23              *.jsp 表示拦截所有 .jsp 请求
24          -->
25          <url-pattern>/*</url-pattern>
26      </filter-mapping>
27  </web-app>
```