Jsp学习总览

JSP学习笔记为本科期间学习过程中根据官网以及部分教程以及菜鸟文档进行整理，仅供本人在JSP开发过程中的学习记录以及学习过程中的问题总结（末尾小tips内）。

使用的是Tomcat10.1.28，同时也记录了Tomcat8与10的版本代码问题。

jsp就是一种动态网页编程技术，像HTML是静态页面，不能动态从数据库获取数据，但是数据不能全部在页面上写死。

JSP：

1. JSP使用
2. Scriptlet脚本小程序
3. JSP的指令标签：include静态包含和动态包含。
4. JSP的四大域对象：page、request、session、application
5. EL表达式的使用

JSTL：

1. 标签库的介绍和使用
2. 常用标签
3. if标签
4. choose、when和otherwise标签
5. formatDate标签

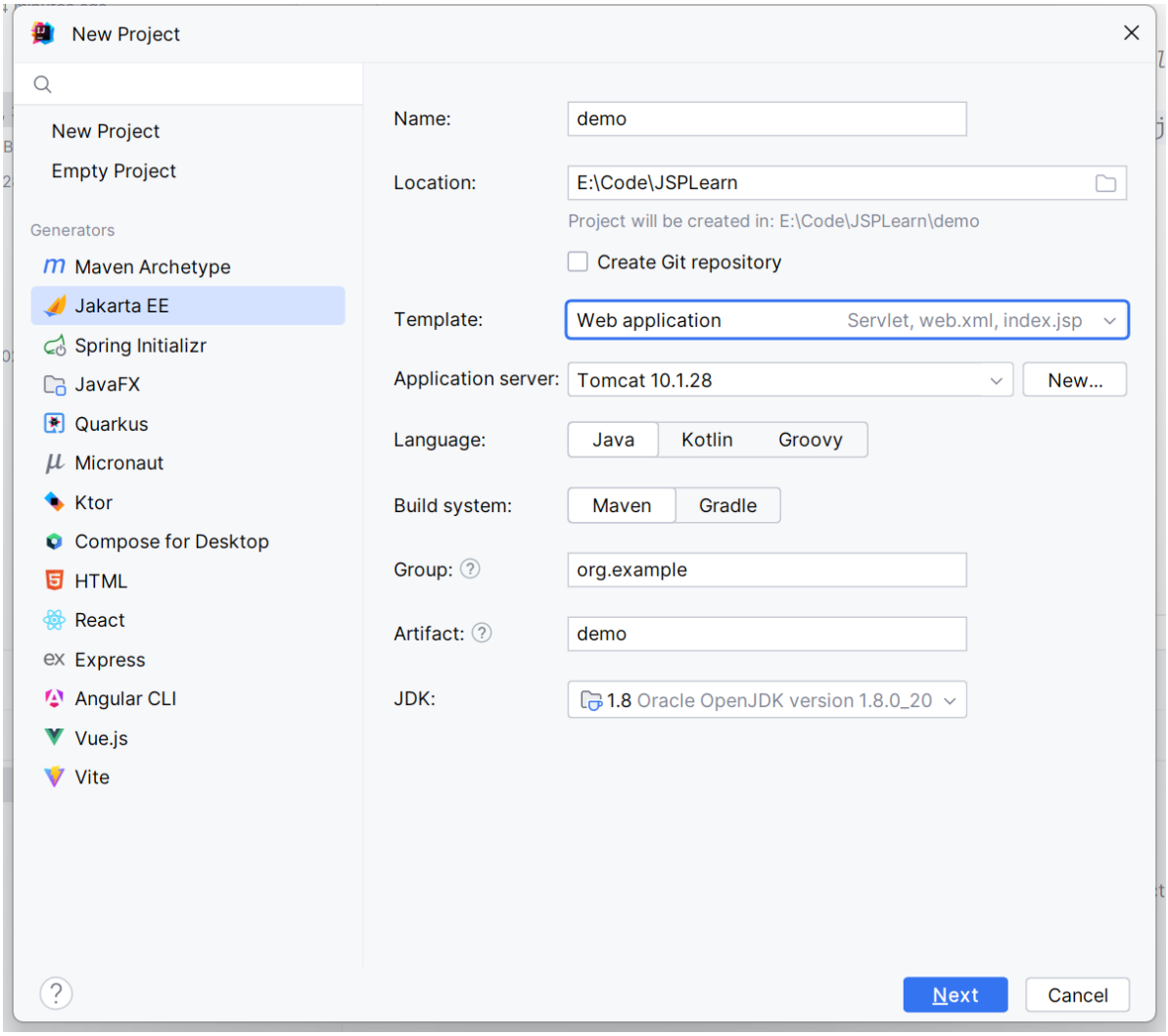
本文档参考教程/相关网站：

1. 菜鸟文档：https://www.runoob.com/jsp/jsp-tutorial.html
2. JSTL下载：https://archive.apache.org/dist/jakarta/taglibs/standard/binaries/
3. Tomcat下载：https://tomcat.apache.org/
4. 相关视频教程：https://www.bilibili.com/video/BV1W64y1C7N8?p=11&vd_source=15c40e07ee8855c2a499ddf65f379043

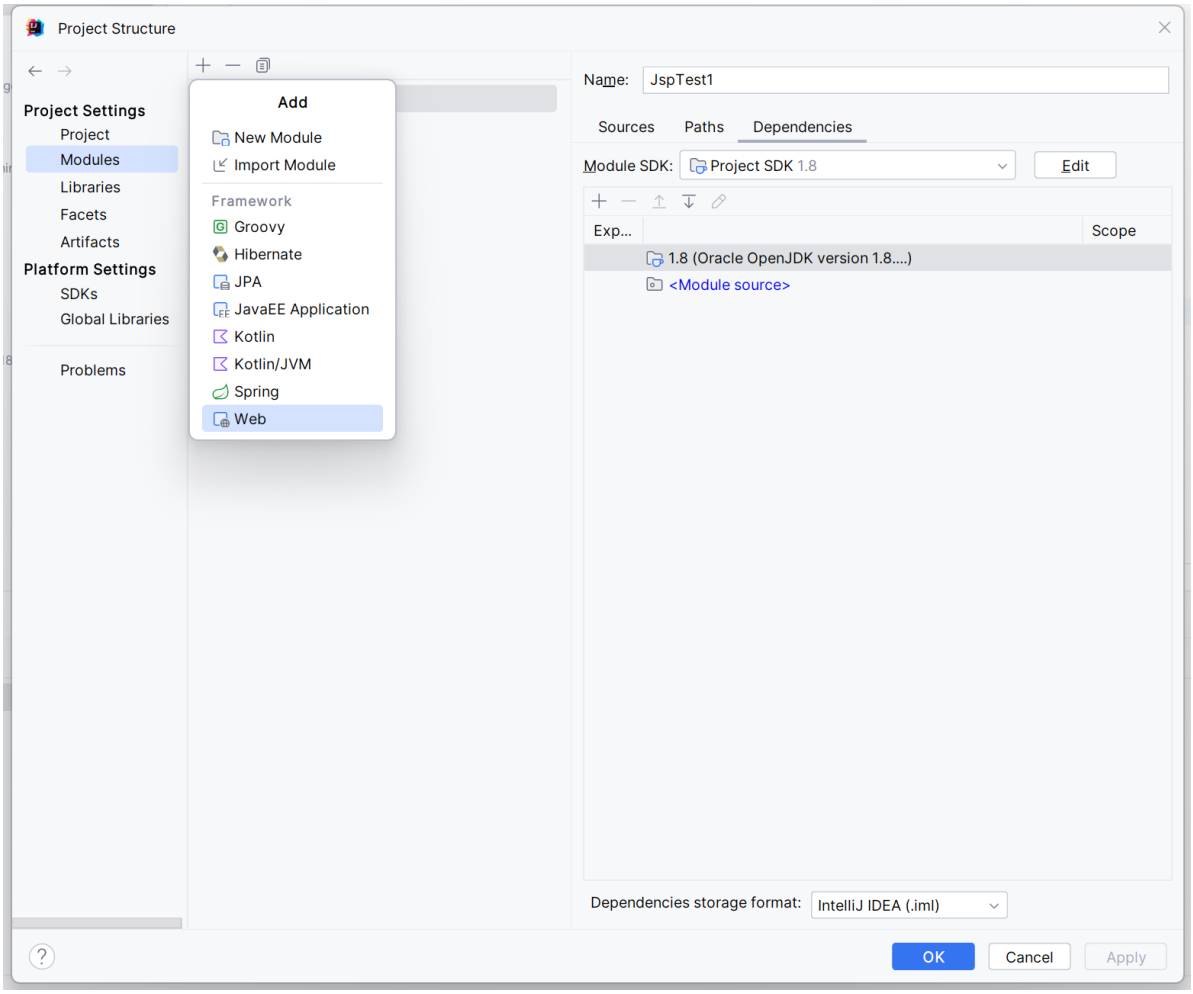
注：默认您已经下载：`Eclipse` 或 `IntelliJ IDEA`、 `JDK` 、`Tomcat`

1 Idea2022新建项目

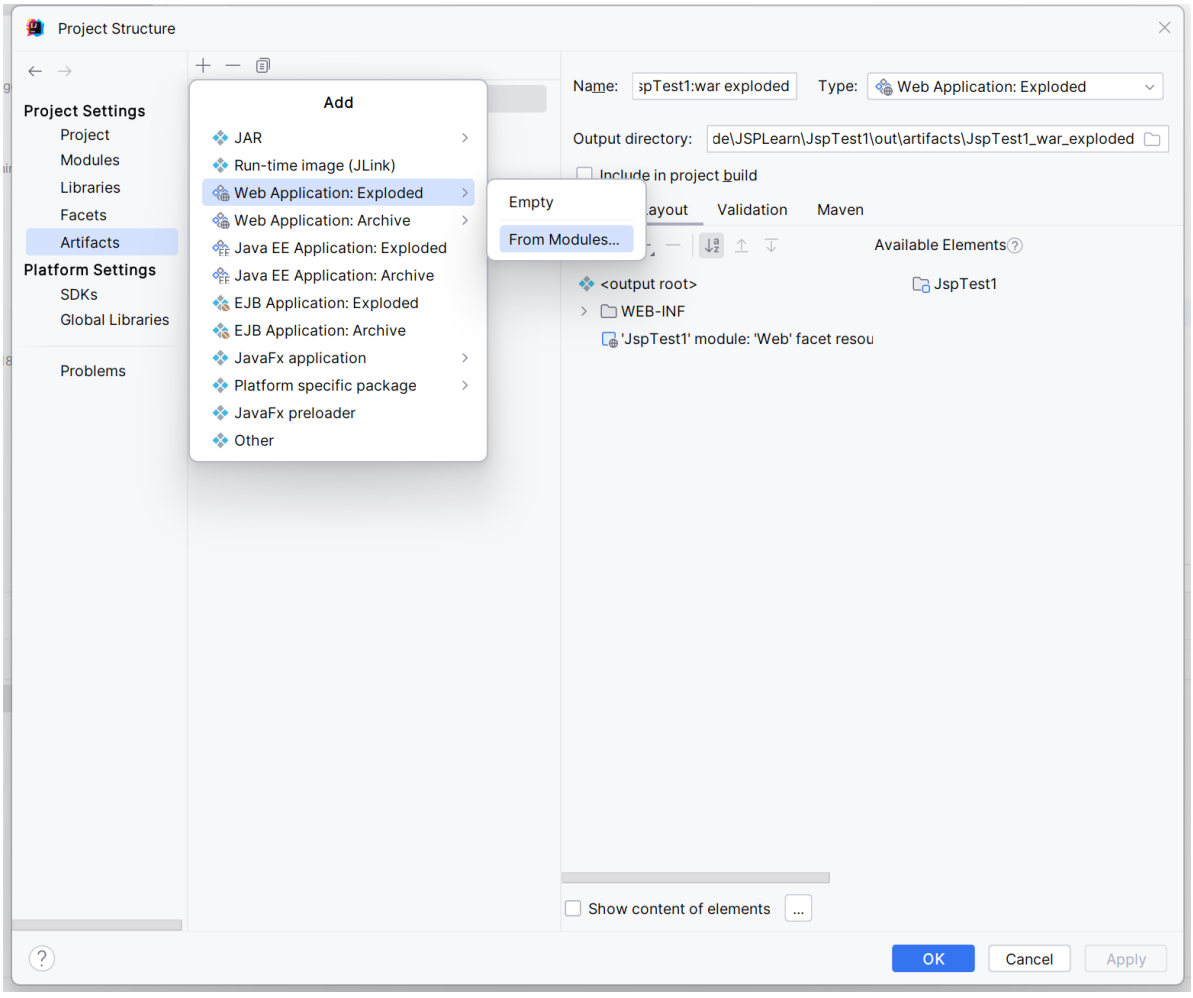
新建项目：


New Project
New Project
Empty Project
Generators
Maven Archetype
Jakarta EE
Spring Initializr
JavaFX
Quarkus
Micronaut
Ktor
Compose for Desktop
HTML
React
Express
Angular CLI
Vue.js
Vite
Name:
demo
Location:
E:\Code\JSPLearn
Project will be created in: E:\Code\JSPLearn\demo
Create Git repository
Template:
Web application
Servlet, web.xml, index.jsp
Application server:
Tomcat 10.1.28
New...
Language:
Java
Kotlin
Groovy
Build system:
Maven
Gradle
Group:
org.example
Artifact:
demo
JDK:
1.8 Oracle OpenJDK version 1.8.0_20
Next
Cancel

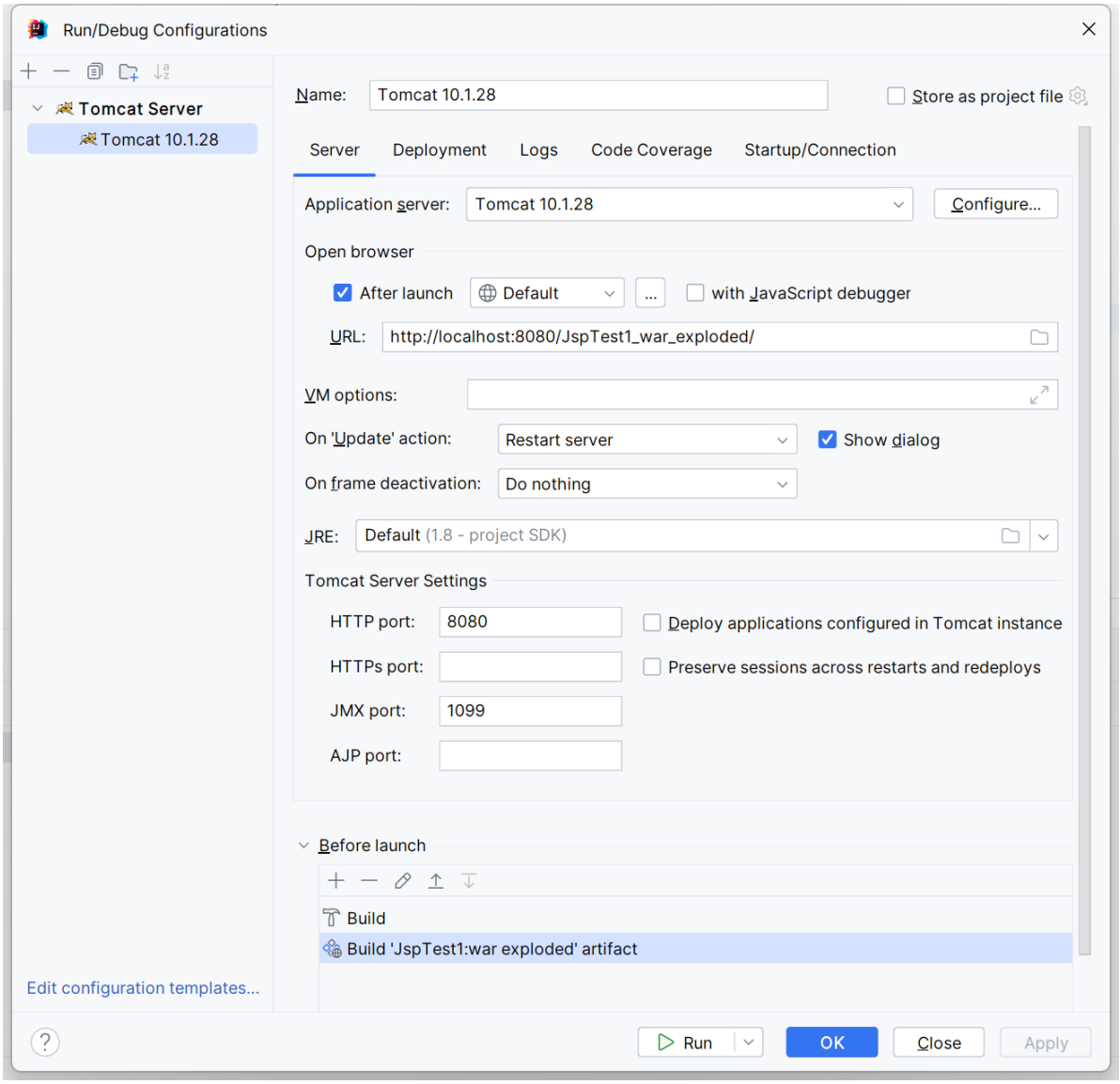
导入Web模块：


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Add
New Module
Import Module
Framework
Groovy
Hibernate
JPA
JavaEE Application
Kotlin
Kotlin/JVM
Spring
Web
Name: JspTest1
Sources
Paths
Dependencies
Module SDK: Project SDK 1.8
Edit
Exp...
Scope
1.8 (Oracle OpenJDK version 1.8....)
<Module source>
Dependencies storage format: IntelliJ IDEA (.iml)
OK
Cancel
Apply

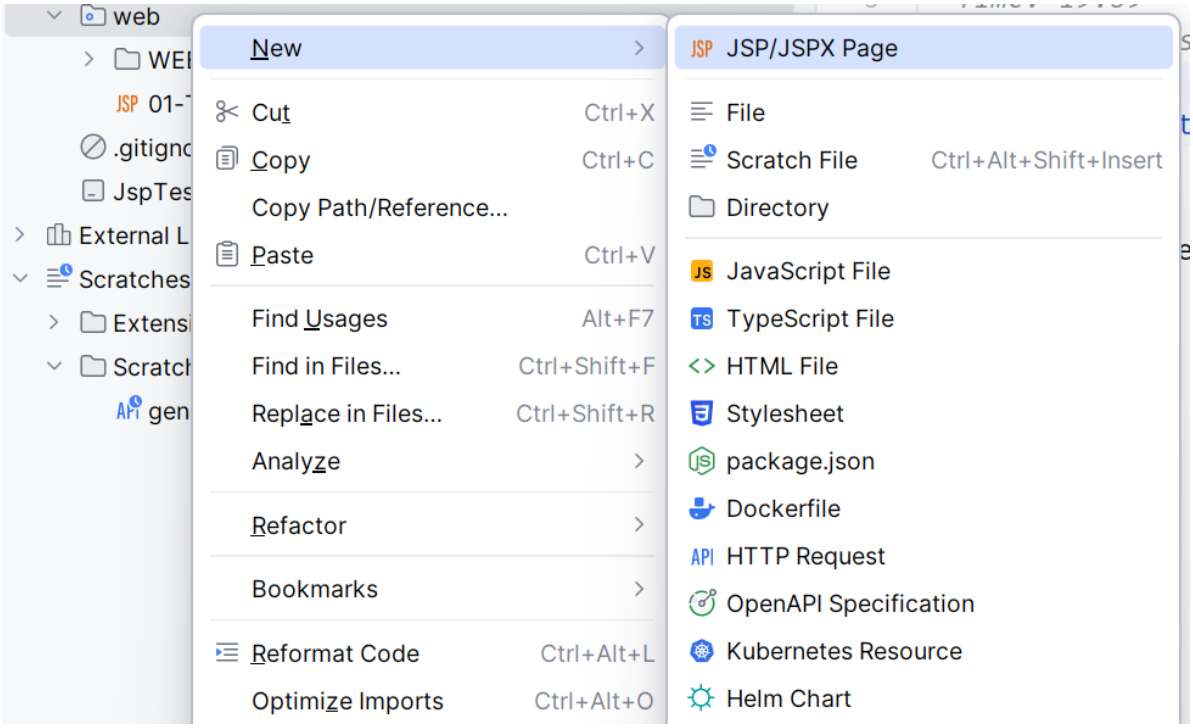
配置Artifact：


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Add
JAR
Run-time image (JLink)
Web Application: Exploded
Web Application: Archive
Java EE Application: Exploded
Java EE Application: Archive
EJB Application: Exploded
EJB Application: Archive
JavaFx application
Platform specific package
JavaFx preloader
Other
Empty
From Modules...
Name: spTest1:war exploded
Type: Web Application: Exploded
Output directory: de\JSPLearn\JspTest1\out\artifacts\JspTest1_war_exploded
Include in project build
Validation
Maven
Available Elements
<output root>
JspTest1
WEB-INF
'JspTest1' module: 'Web' facet resou
Show content of elements
OK
Cancel
Apply

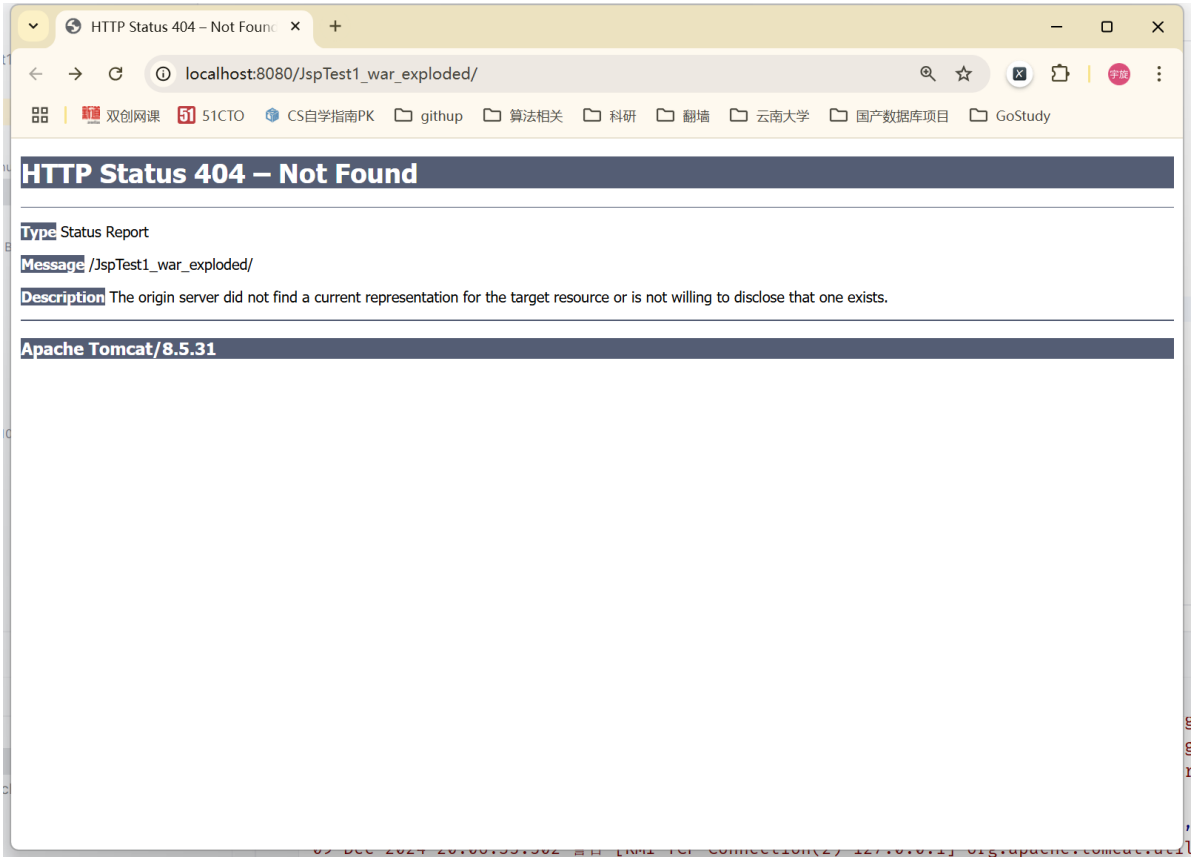
配置Tomcat：



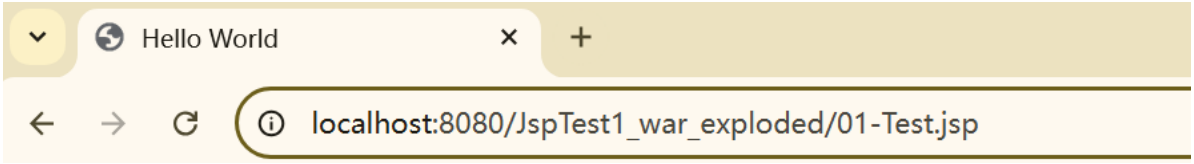
新建jsp文件：



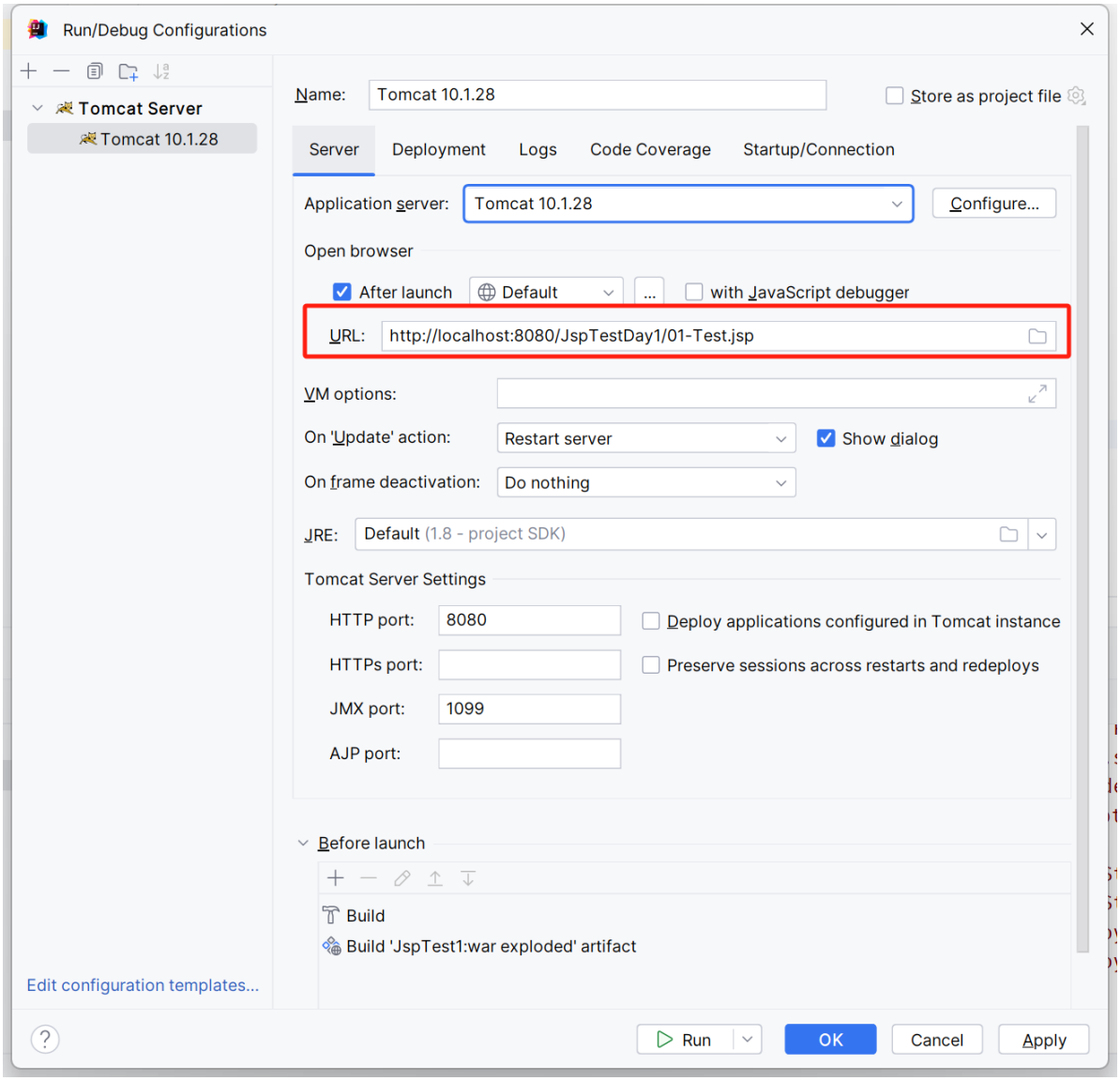
页面报错：

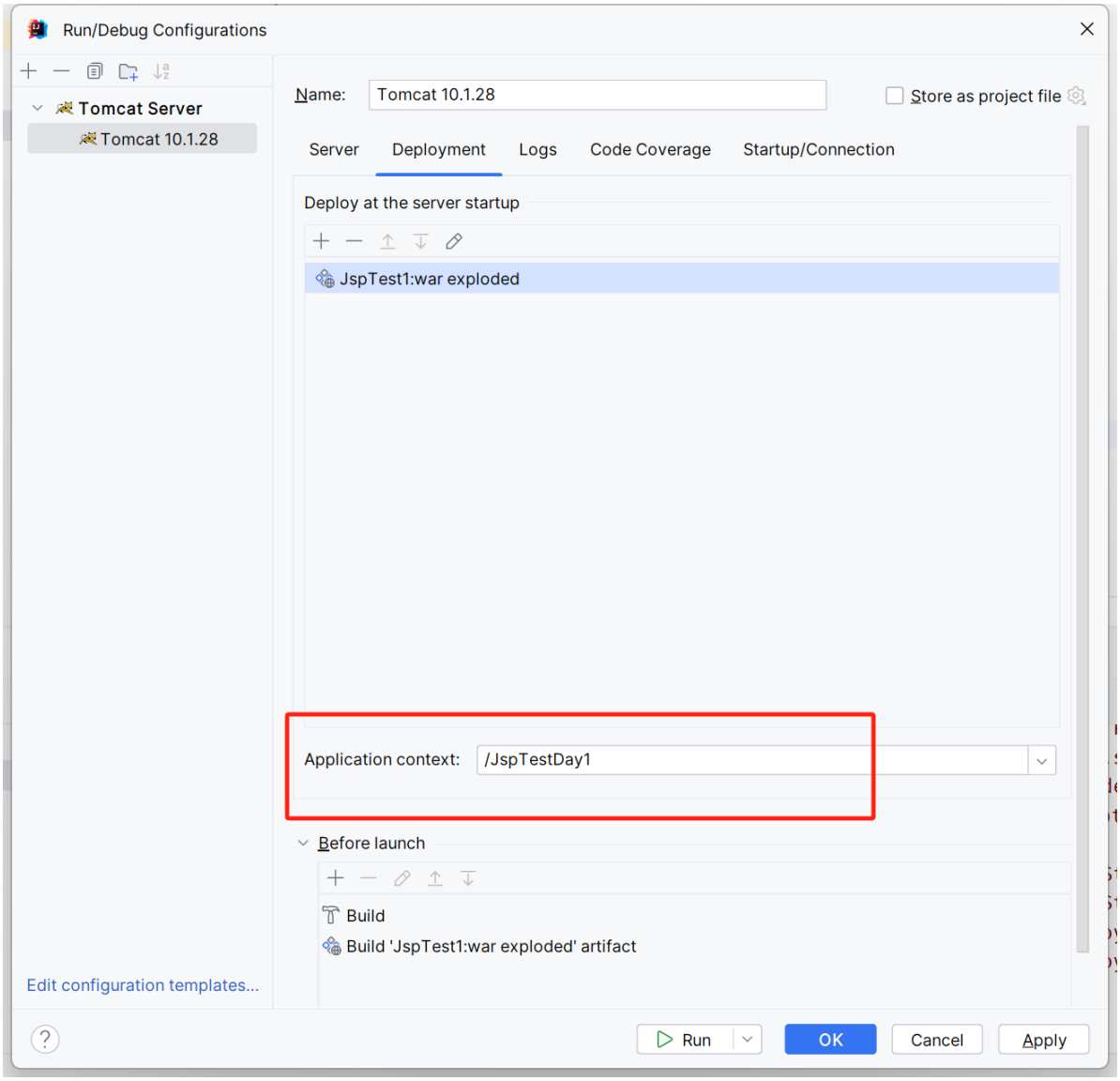


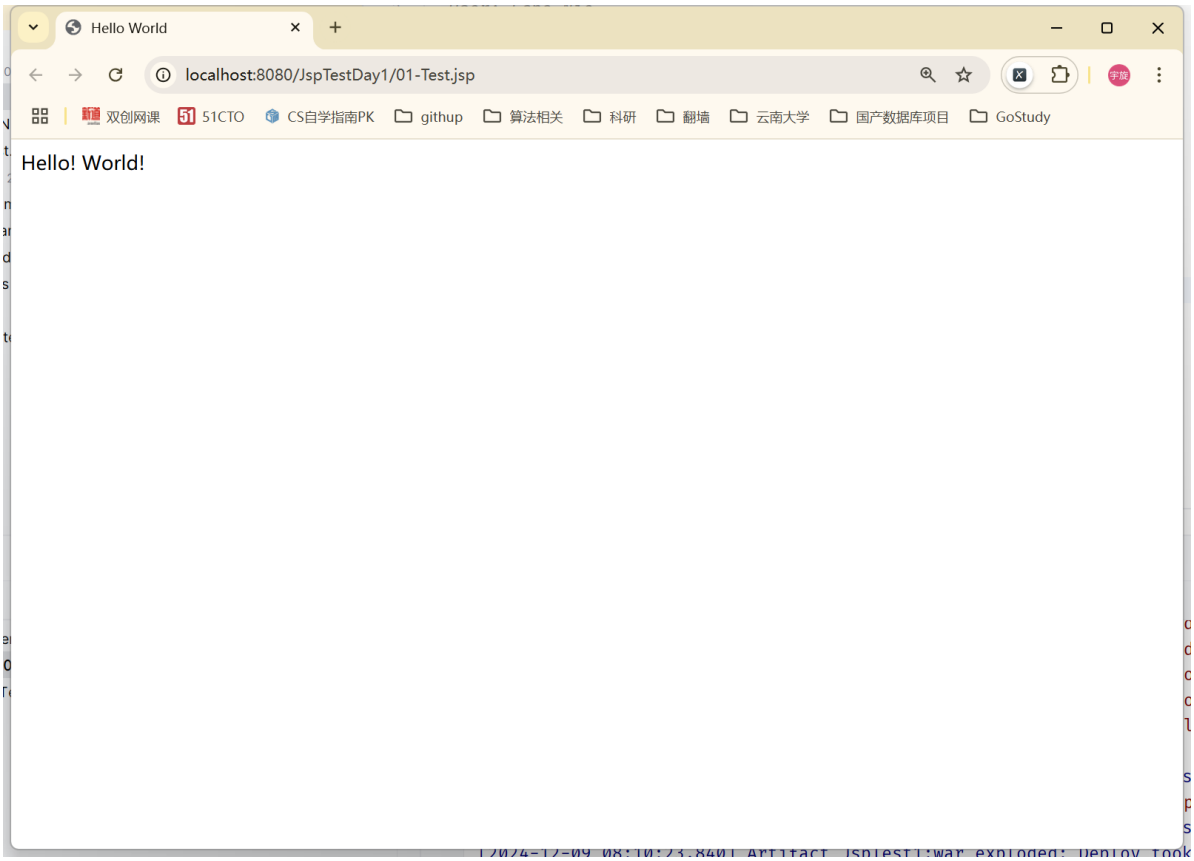
后缀加上你的项目名字：



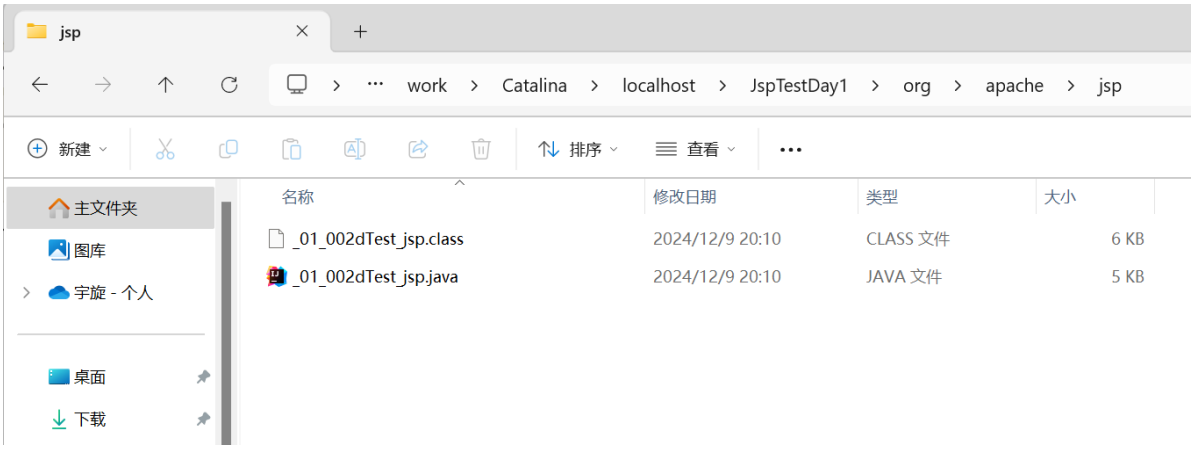
发现url很奇怪，直接自定义：


Run/Debug Configurations
Tomcat Server
Tomcat 10.1.28
Edit configuration templates...
Name: Tomcat 10.1.28
Store as project file
Server
Deployment
Logs
Code Coverage
Startup/Connection
Application server: Tomcat 10.1.28
Configure...
Open browser
After launch
Default
with JavaScript debugger
URL: http://localhost:8080/JspTestDay1/01-Test.jsp
VM options:
On 'Update' action: Restart server
Show dialog
On frame deactivation: Do nothing
JRE: Default (1.8 - project SDK)
Tomcat Server Settings
HTTP port: 8080
Deploy applications configured in Tomcat instance
HTTPs port:
Preserve sessions across restarts and redeploys
JMX port: 1099
AJP port:
Before launch
Build
Build 'JspTest1:war exploded' artifact
Run
OK
Cancel
Apply





接着会在对应目录下生成class和java文件（用Listary搜JSPTestDay1）：



2 基础语法

2.1 注释

```
//单行注释

/*多行注释*/

<!-- HTML风格注释 -->

<%--  JSP注释  --%>
```

```
<%--
  Created by IntelliJ IDEA.
  User: Echo-Nie
  Date: 2024/12/9
  Time: 20:17
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>JSP基础语法</title>
</head>
<body>
<pre>
    JSP有两种的注释：显示注释和隐式注释
        1. 显示注释：能够在客户端中查看的注释： &lt;!-- HTML 风格注释 --&gt;
        <!--HTML风格注释-->
        2. 隐式注释：不能在客户端看到的注释：
            2.1 <%--JSP自己的注释--%>&lt;%-- JSP自己的注释 --%&gt;
            2.2  JAVA注释:    //单行与多行
<%--  JAVA脚本段  --%>
<%
    //这是单行注释
    /*这是多行注释*/
%>
</pre>

</body>
</html>
```

JSP有两种的注释：显示注释和隐式注释
1. 显示注释：能够在客户端中查看的注释： <!-- HTML 风格注释 -->

2. 隐式注释：不能在客户端看到的注释：
2.1 <%-- JSP自己的注释 --%>
2.2 JAVA注释: //单行与多行

2.2 Scrptlet

在JSP中很重要，Scriplet（脚本小程序），所有嵌入在HTML代码中的Java程序。

在JSP中有三种Scriptlet带阿米，必修用Scriptlet标出来。

```
第一种：<%  %>：Java脚本段，可以定义局部变量，编写语句。
第二种：<%! %>：可以定义全局变量，方法，类。
第三种：<%= %>：表达式，数据一个变量或者具体内容。
```

```jsp
<%--
  Created by IntelliJ IDEA.
  User: Echo-Nie
  Date: 2024/12/9
  Time: 20:17
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Scriptlet</title>
</head>
<body>
<%--第一种：<%  %>：Java脚本段，可以定义局部变量，编写语句。--%>
<%--生成的代码在servlet的service方法体中--%>
<%
    String s = "hello";
//    将s输出到控制台
//    System.out.println(s);
    out.print(s);//将s输出到浏览器
    out.write("---");
    out.print("输出全局变量：num="+num);
    out.write("---");
%>

<%--第二种：<%! %>：可以定义全局变量，方法，类。--%>
<%--生成的代码在servlet的类体中--%>
<%!
    //声明全局变量
    int num = 0;
    //没有sout
%>

<%--    第三种：<%= %>：表达式，数据一个变量或者具体内容。    --%>
<%--    生成的代码在servlet的service方法体中，相当于out.print()    --%>
```

```
<%= s %>

</body>
</html>
```

ScriptletTest_jsp.java

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/8.5.31
 * Generated at: 2024-12-09 12:54:09 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;


public final class ScriptletTest_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent,
                 org.apache.jasper.runtime.JspSourceImports {


    //声明全局变量

    int num = 0;
    //没有sout


  private static final javax.servlet.jsp.JspFactory _jspxFactory =
          javax.servlet.jsp.JspFactory.getDefaultFactory();

```

查看源码发现全局变量在源码中的位置是最上面

2.3 JSP的指令标签

使用包含操作，将一些重复的代码包含进来使用，从正常的页面组成来看，有可能分为几个区域，其中有一些区域可能一直不需要改变，改变的就其中一个具体的内容区域。

方法1：每个JSP页面（HTML）都包含工具栏、头部信息、尾部信息、具体内容。

方法2：将工具栏、头部、尾部信息都分成各个独立文件，使用的时候直接导进去。

我们发现第一种方法会有代码重复，修改也不方便，所以在JSP中实现包含操作我们一般是静态包含和动态包含。静态包含使用include就行，动态包含需要使用include动作标签。

好比一个网站，有head和foot，中间是Body，将head和foot封装起来这种感觉。

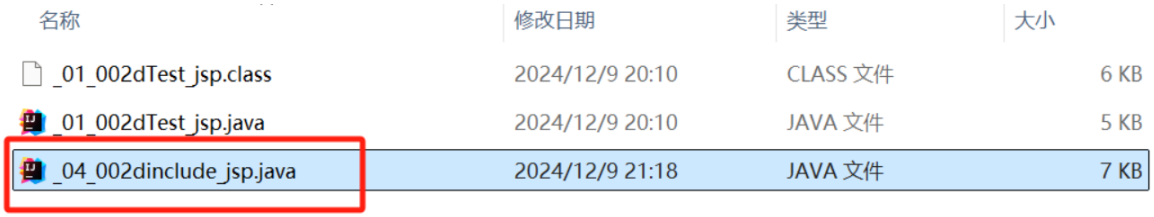
2.3.1 静态包含

```
<%@ include file ="url" %>       <!--相对路径 -->
```

静态包含就是将内容进行直接替换，好比程序中定义变量一样，在servlet引擎转译的时候，把这个文件包含进去了（将两个文件的源代码整合到一起，全部放到jspService方法中），所以只生成了一个servlet，所以两个页面不能有同名变量，耦合性高但是不够灵活。

特点：

1. 将内容进行直接替换
2. 静态包含只会生成一个源码文件，最终内容均在jspService方法中
3. 因为是一个源码文件，所以不能出现同名变量
4. 运行效率高一点，但是耦合高不灵活




```jsp
<%--
  Created by IntelliJ IDEA.
  User: Echo-Nie
  Date: 2024/12/9
  Time: 21:11
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Header头部</title>
</head>
<body>
---------<br>
这是头部<br>
---------<br>
</body>
</html>
```

```jsp
<%--
  Created by IntelliJ IDEA.
  User: Echo-Nie
  Date: 2024/12/9
  Time: 21:10
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>include</title>
    <%--
        include静态包含：<%@include file="header.jsp" %>
        特点：
        1、将内容进行直接替换
        2、静态包含只会生成一个源码文件，最终内容均在jspService方法中
        3、因为是一个源码文件，所以不能出现同名变量
        4、运行效率高一点，但是耦合高不灵活
    --%>
</head>
<body>
<%@include file="header.jsp" %>
<h2>主体部分</h2>
<%
    int num = 10;
    out.print(num);
%>
<%@include file="footer.jsp" %>
</body>
</html>
```

```jsp
<%--
  Created by IntelliJ IDEA.
  User: Echo-Nie
  Date: 2024/12/9
  Time: 21:11
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Footer底部</title>
</head>
<body>
<%
    int num = 11;
%>
-------------<br>
这是尾部<br>
-------------<br>
</body>
</html>
```

2.3.2 动态包含

动态包含在代码的编译阶段，包含和被包含是两个独立的部分，只有当运行时，才会动态包含进来，有点像java的方法调用。所以会有多个源码文件。

```jsp
<jsp:include page="include.jsp"></jsp:include>
```

PS：动态包含中间，也就是include标签之间不要加任何内容（空格也不能加），除非你确定要使用参数。如果有内容它就认为你有参数，就会去找你带参数的标签。

特点：

1. 相当于方法的调用
2. 会生成多个源码文件
3. 可以定义同名变量
4. 效率高耦合度低

2.3.3 动态包含带参数

```
<jsp:param name="str" value="string"/>
<jsp:param name="str" value="<%=str%>"/>
```

name属性不支持表达式，而value支持表达式。

```
<%--
  User: Echo-Nie
  Date: 2024/12/9
  Time: 21:27
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>include动态包含</title>
</head>
<%--
使用动态包含：
    <jsp:include page="url">
        <jsp:param name="参数名" value="参数值"/>
    </jsp:include>
    注意：name不支持表达式；value支持表达式

    获取参数：
        request.getParameter(name);通过指定参数获取变量名字
--%>
<jsp:include page="header.jsp"></jsp:include>
<h2>主体内容</h2>
<%
    int a = 10;
%>
<jsp:include page="footer.jsp"></jsp:include>
<%--第一次footer的时候，没有传参，所以取到的是null--%>
<%--动态包含传参--%>
<%
    String str = "hello";
%>
<%--第二次footer的时候传参，所以取到的是admin和hello--%>
<jsp:include page="footer.jsp">
    <jsp:param name="uname" value="admin"/>
    <jsp:param name="msg" value="<%=str%>"/>
</jsp:include>

<body>

</body>
</html>
```

```jsp
<%--page也不用写死的--%>
<%
String url = "footer.jsp";
%>
<jsp:include page="<%=url%>"></jsp:include>
```

2.4 JSP的四大域对象

2.4.1 四种属性范围

在JSP中提供了四种属性的保存范围，就是一个设置的对象，可以在多少个页面中保存并使用。

1.page范围

pageContext：只在一个页面中保存属性，跳转之后无效。

2.request范围：

request：只在一次请求中保存，服务器跳转后依然有效。

3.session范围

session：在一次会话范围中，无论何种跳转都可以使用。

4.application范围：

application：在整个服务器上保存。

method	type	description
public void setAttributes(String name,Object o)	普通	设置属性的名称及内容
public Object getAttribute(String name)	普通	根据属性名称取属性
public void removeAttribute(String name)	普通	删除指定操作

2.4.2 属性范围特点

1.page

本页面取得服务器跳转后无效

2.request

服务器跳转有效，客户端跳转无效。

如果是客户端跳转，则相当于发生了两次请求，那么第一次的请求将不存在了；如果希望不管是客户端还是服务器跳转，都能保存的话需要扩大范围。

3.session

无论客户端还是服务器都可以取得，但是如果重新开新的浏览器，则无法取得之前设置的session，因为每一个session只保存在当前浏览器中，并在相关页面取得。

如果想让属性设置一次之后，不管是否是新的浏览器都可以取得，用application

4.application

所有的application属性直接保存在服务器上，所有的用户(每一个session)都可以直接访问取得。

只要是通过application设置的属性，则所有的session都可以取得，表示公共的内容，但是如果此时服务器重启了，则无法取得了，因为关闭服务器后，所有的属性都消失了，所以需要重新设置。

使用：在合理的范围内尽可能小。

3 login简单页面

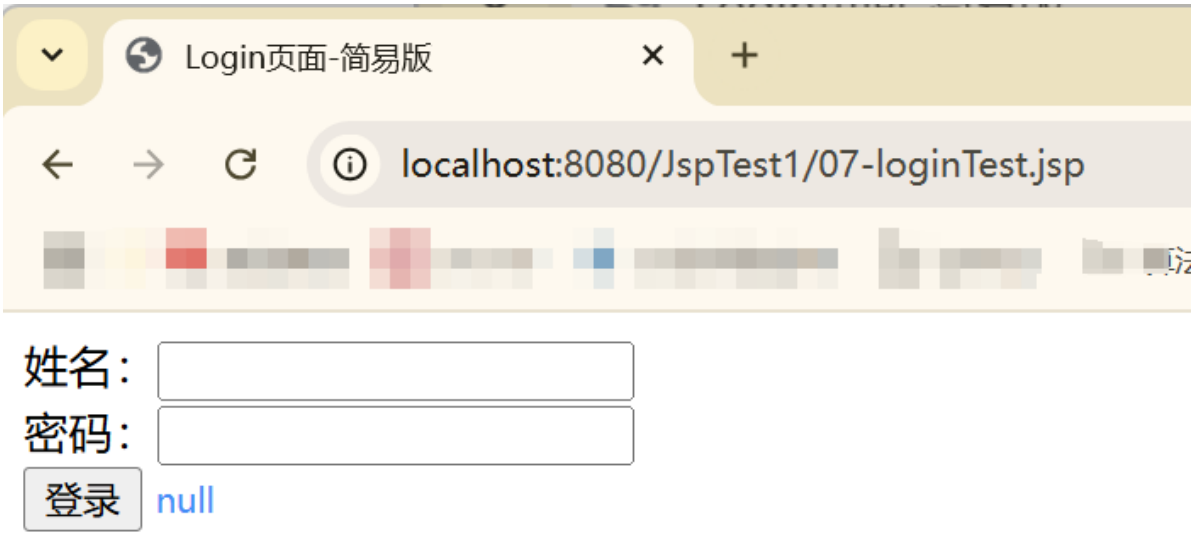
3.1 loginTest编写

```
<%--
  User: Echo-Nie
  Date: 2024/12/10
  Time: 9:39
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Login页面-简易版</title>
</head>
<body>

<form action="loginServlet" method="post">
    姓名：<input type="text" name="uname"></br>
    密码：<input type="text" name="upwd"></br>
    <button>登录</button>
    <%--获取后台设置在作用域中的数据并且显示--%>
    <span style="color: dodgerblue;font-size: 12px">
<%=request.getAttribute("msg")%></span>
</form>
</body>
</html>
```

3.2 LoginServlet编写

```
package com.ynu.controller;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * @ClassName LoginService
 * @Description
 * @Author Echo-Nie
 * @Date 2024/12/10 9:46
```

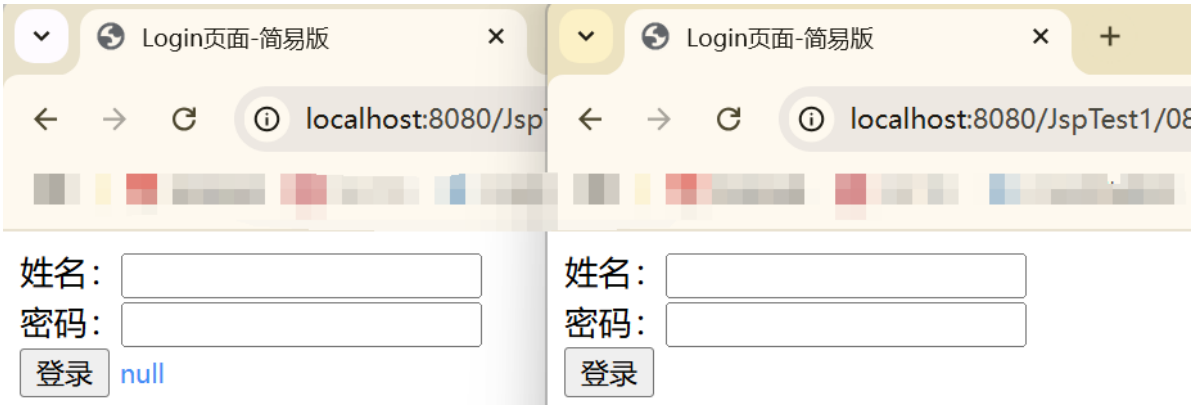
```
 * @Version V1.0
 */

@WebServlet("/loginServlet")
public class LoginServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        //设置客户端编码格式
        request.setCharacterEncoding("UTF-8");
        //接收客户端传参
        String uname = request.getParameter("uname");
        String upwd = request.getParameter("upwd");

        //判断传参为空
        if (uname == null || "".equals(uname.trim())) {
            //提示用户信息
            request.setAttribute("msg", "用户姓名不能为空");
            //请求转发跳转到loginTest.jsp
            request.getRequestDispatcher("07-loginTest.jsp").forward(request,
response);
            return;
        } else if (upwd == null || "".equals(upwd.trim())) {
            //提示用户信息
            request.setAttribute("msg", "密码不能为空");
            //请求转发跳转到login.jsp
            request.getRequestDispatcher("07-loginTest.jsp").forward(request,
response);
            return;
        }

        //判断账号密码是否正确
        if(!"nyx".equals(uname)){
            //提示用户名错误
            request.setAttribute("msg","用户名错误，登录失败");
            request.getRequestDispatcher("07-loginTest.jsp").forward(request,
response);
            return;
        }
        //判断密码
        if(!"nyx".equals(upwd)){
            request.setAttribute("msg","密码错误，登录失败");
            request.getRequestDispatcher("07-loginTest.jsp").forward(request,
response);
            return;
        }
        //登录成功
        //设置登录时的信息到session作用域
        request.getSession().setAttribute("uname",uname);
        //跳转到登录页面
        response.sendRedirect("07-index.jsp");

    }
}
```



4 EL表达式

4.1 EL基本语法

首先发现上面提示信息那一块是null，影响美观，能不能换成空串？引出EL表达式。



EL(Expression Language)是为了使JSP 写起来更加简单。表达式语言的灵感来自于ECMAScript 和 XPath 表达式语言，它提供了在JSP 中简化表达式的方法，让Jsp 的代码更加简化。

```
语法结构：${expression}
```

EL表达式操作的一般是域中的数据，操作不了局部变量。

域对象的概念在JSP 中一共有四个:**pageContext、request、session、application；**范围依次是:

本页面

一次请求

一次会话

整个应用程序。

当需要指定从某个特定的域对象中查找数据时可以使用四个域对象对应的空间对象，分是：pageScope，requestScope，sessionScope，applicationScope。

EL 默认的查找方式为从小到大查找，找到即可。当域对象全找完了还未找到则返回空字符串""。

```
<%--
  User: Echo-Nie
  Date: 2024/12/10
  Time: 16:11
  To change this template use File | Settings | File Templates.
  Description:
  EL表达式
    作用:简化JSP代码
    格式:
        ${域对象的名称}
    操作对象:
        EL表达式一般操作的是域对象，不能操作局部变量。
    操作范围:
        page范围在当前页面
        request范围在一次请求
        session范围在一次会话
        application范围在整个应用程序
     可通过pageScope、
     PS：如果EL表达式获取的域对象值为空，默认是空串；
     EL表达式默认从小到大的范围去找，找到即可，如果四个范围都没找到，显示空字符串。
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>EL表达式</title>
</head>
<body>
<%--设置数据--%>
<%
    pageContext.setAttribute("uname", "zs");
    request.setAttribute("uname", "ls");
    session.setAttribute("uname", "ww");
    application.setAttribute("uname", "zl");

//    局部变量
    String str = "hello";
%>
<%--获取数据--%>
获取局部变量：${str} </br>
获取域对象：${uname}<br>

<%--
输出如下：
获取局部变量：
获取域对象：zs
--%>

<%--获取指定域的数据--%>
<br>获取指定域的数据：<br>
  page域数据：${pageScope.uname}<br>
  request域数据：${requestScope.uname}<br>
  session域数据：${sessionScope.uname}<br>
  application域数据：${applicationScope.uname}<br>
</body>
</html>
```

4.2 EL获取数据

设置域对象中的数据

```
<%
    /*page*/
//    本页面取得服务器跳转<jsp :forward>后无效
    pageContext.setAttribute("uname","zs");

    /*request*/
//    服务器跳转有效，客户端跳转无效。
    request.setAttribute("uname","ls");

    /*session*/
//    如果重新开新的浏览器，则无法取得之前设置的session
    session.setAttribute("uname","ww");

    /*application*/
//    服务器重启了，则无法取得了
    application.setAttribute("uname","lmz");
%>
```

获取域对象的值

```
${uname}    <!-- 输出结果为:zs -->
```

获取指定域对象的值

```
<%--获取指定域的数据--%>
<br>获取指定域的数据：<br>
  page域数据：${pageScope.uname}<br>
  request域数据：${requestScope.uname}<br>
  session域数据：${sessionScope.uname}<br>
  application域数据：${applicationScope.uname}<br>
```

获取List

```
<%
    //List
    List<String> list = new ArrayList<>();
    list.add("aaa");
    list.add("bbb");
    list.add("ccc");
    request.setAttribute("list", list);
%>
<h4>获取List</h4>
获取list的size：${list.size()};<br>
获取list的指定下标值list[1]：${list[1]}<br>
```

获取Map

```
<%
    //Map
    Map map = new HashMap<>();
    map.put("aaa", "111");
    map.put("bbb", 111);
    map.put("ccc", 33);
    request.setAttribute("map", map);
%>
<h4>获取Map</h4>
获取map指定的key的value：${map.aaa} 或者 ${map["bbb"]};
```

获取JAVABean

```
<%
    //JavaBean
    User user = new User(1, "nyx", "123");
    request.setAttribute("user", user);
%>
<h4>获取JAVABean</h4>
${user}<br>
获取JAVABean中属性：uname=${user.uname}或者getUname=${user.getUname()}
<%--
输出如下：
com.ynu.po.User@62adb472
获取JAVABean中属性：uname=nyx或者getUname=nyx
--%>
```

4.3 Empty与一些运算

```
<%@ page import="java.util.List" %>
<%@ page import="java.util.ArrayList" %>
<%@ page import="java.util.Map" %>
<%@ page import="java.util.HashMap" %>
<%@ page import="com.ynu.po.User" %><%--
  User: Echo-Nie
  Date: 2024/12/11
  Time: 23:35
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>10-empty与EL运算.jsp</title>
    <%--
        empty
            判断域对象是否为空，为空返回true，否则为false
                ${empty 限域变量名}
            判断对象是否不为空
                ${!empty 限域变量名}
         对于字符串：
            不存在返回true
            有值返回false
            空串true
            null也是true
```

```
            对于List：
              null返回true
              没有长度size的也返回true

            对于Map：
              null返回true
              空Map对象返回true

            对于JAVABean：
               null返回true
               空对象返回false
    --%>

</head>
<body>
<%
    //字符串
    request.setAttribute("str1", "abc");
    request.setAttribute("str2", "");
    request.setAttribute("str3", null);
    //List
    List list1 = new ArrayList<>();
    List list2 = null;
    List list3 = new ArrayList<>();
    list3.add(1);
    request.setAttribute("list1", list1);
    request.setAttribute("list2", list2);
    request.setAttribute("list3", list3);

    //Map
    Map map = new HashMap<>();
    Map map1 = null;
    Map map2 = new HashMap<>();
    map2.put(1, 2);
    request.setAttribute("map", map);
    request.setAttribute("map1", map1);
    request.setAttribute("map2", map2);

    //JAVABean
    User user = null;
    User user1 = new User();
    User user2 = new User(1, "nyx", "123");
    request.setAttribute("user", user);
    request.setAttribute("user1", user1);
    request.setAttribute("user2", user2);
%>
<h4>判断字符串是否不存在</h4>
${empty str}返回true<br>
${empty str1}返回false<br>
${empty str2}返回true<br>
${empty str3}返回true<br>
<br>

<h4>判断List是否为空</h4>
${empty list1}返回true<br>
${empty list2}返回true<br>
${empty list3}返回false<br>
```

```
<h4>判断Map是否为空</h4>
${empty map}返回true<br>
${empty map1}返回true<br>
${empty map2}返回false<br>

<h4>判断JAVABean</h4>
${empty user}返回true<br>
${empty user1}返回false<br>
${empty user2}返回false<br>
----------------------比较两个值是否相等，==或eq----------------
<%
    request.setAttribute("a", 10);
    request.setAttribute("b", 2);
    request.setAttribute("c", "aa");
    request.setAttribute("d", "bb");
%>
${a==b}
${c==d}
${c eq d}
${a ==5}
${c =="aa"}

<br>
</body>
</html>
```

5 JSTL

5.1 标签使用

Java Server Pages Standard Tag Libray(STL):JSP 标准标签库。

是一个定制标签类库的集合，用于解决一些常见的问题，例如迭代一个映射或者集合、条件测试、XML处理，甚至数据库和访问数据库操作等。

核心标签库：包含web常见工作，如循环、表达式赋值、基本输入输出等。

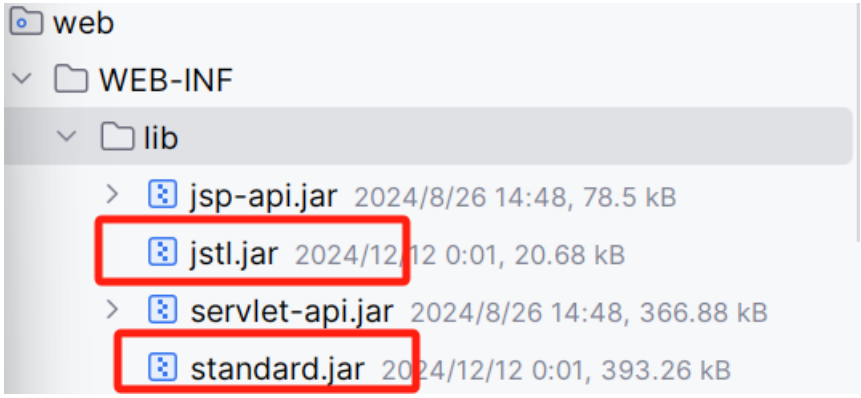
格式化标签库：用来格式化显示数据的工作，比如：对不同区域的日期格式化。

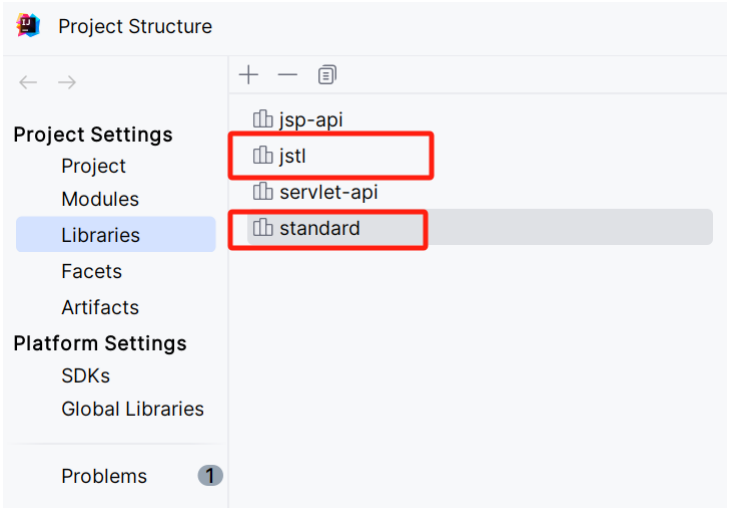
要在JSP中使用JSTL类库，必须使用下面的taglib指令：

```
<%@taglib uri="" prefix="" %>
如：
<%@taglib uri="http://java.sun.com/jsp/jstl/cire" prefix="c" %>
```

前缀可以是任意内容，遵循规范形成相同的码风会比较好，使用实现设计好的前缀即可。

在官网(https://archive.apache.org/dist/jakarta/taglibs/standard/binaries/)下载好Jakarta-taglib-standard-1.1.2.zip解压后将lib下的两个jar包拷贝到指定目录下





5.2 条件动作标签

条件动作指令用于处理页面的输出结果依赖于某些输入值的情况，在lava 中是利用 if、 if..else 和 switch 语句来进行处理的。在JSTL中也有4个标签可以执行条件式动作指令:if、choose、when和otherwise。

if标签

if它的主体内容，测试结果保存在一个Boolean 对象中，并创建一个限域变量来引用 Boolean 对象。可以利用 var 属性设置限域变量名，利用 scope 属性来指定其作用范围。

```
<c:if test="<boolean>" var="<string>" scope="<string>">

...

</c:if>
```

```
<html>
<head>
    <title>JSTL的使用</title>
    <%--
        首先拷贝两个jar包到web-INF的lib目录下；再导入jst和standard的jar包到项目中
    --%>
</head>
<body>
<c:if test="${1==1}">
    Hello JSTL
</c:if>
<%
```

```
    request.setAttribute("num", 10);
    request.setAttribute("num1", 10);
%>


<c:if test="${num>0}">
    数据大于0
</c:if>

<br>
<c:if test="${num>100}" var="flag" scope="request"></c:if>
${flag}--${pageScope.flag}${requestScope.flag}--${sessionScope.flag}

<c:if test="${num1>20}" var="flag1" scope="request"></c:if>
${flag1}--${requestScope.flag1}--${sessionScope.flag1}
<%--    false--false--  --%>
<%--
    test条件判断，操作的是域对象，接受返回结果是boolean类型的值
    var是限域对象，存放在作用域中的变量名，用来接受判断结果的值
    scope是限域变量名的范围，page-request-session-application
--%>
</body>
</html>
```

属性	描述	是否必要	默认值
test	条件	是	无
var	用于存储条件结果的变量（限域变量名）	否	无
scope	可取值：page|request|session|application	否	page

choose-when-otherwise标签

相当于就是switch-case；choose 和 when 标签的作用与Java 中的 switch 和 case 关键字相似，用于在众多选项中做出选择。也就是说:他们为相互排斥的条件式执行提供相关内容。

switch语句中有case，而choose标签中对应有when，switch语句中有default，而choose标签中有otherwise.

```
<c:choose>
    <c:when test="<boolean>">

    ...

    </c:when>
    <c:when test="<boolean>">

    ...

    </c:when>

    ...
```

```
    <c:otherwise>
        ...
    </c:otherwise>
</c:choose>
```

5.3 迭代标签

forEach 是将一个主体内容迭代多次，或者迭代一个对象集合。可以迭代的对象包括所有的java.util.Collection 和java.util.Map 接口的实现，以及对象或者基本类型的数组。

还可 以迭代java.util.Iterator 和 java.util.Enumeration,但不能在多个动作指令中使用 Iterator 或者Enumeration,因为Iterator 或者 Enumeration都不能重置(reset)。各属性含义如下:

属性	描述	是否必要	默认值
items	要被循环的数据	否	无
begin	开始的元素（0=第一个元素，1=第二个元素）	否	0
end	最后一个元素（0=第一个元素，1=第二个元素）	否	Lastelement

```
<c:forEach
    items="<object>"
    begin="<int>"
    end="<int>"
    step="<int>"
    var="<string>
    varstatus="<string>"
</c:forEach>
```

```
<!-- 遍历主体内容多次 -->

<c:forEach begin="0" end="10" var="i">
    标题 $ti<br>
</c:forEach>

<!-- 循环 -->
<%
List<String> list = new ArrayList<String>();

for (int i = 1; i <= 10; i++) {
    list.add("A:" + i);
}

pageContext.setAttribute("li", list);
%>

<!-- 循环集合 -->

<c:forEach items="${1i}" var="item">
    ${item}
</c:forEach>

<hr>

<table align="center" width="800" border="1" style="border-collapse: collapse;">
```

```
    <tr>
        <th>名称</th>
        <th>当前成员下标</th>
        <th>当前成员循环数</th>
        <th>是否第一次被循环</th>
        <th>是否最后一次被循环</th>
    </tr>
    <c:forEach items="${1i}" var="item" varStatus="itemp">
        <tr>
            <td>${item}</td>
            <td>${itemp.index}</td>
            <td>${itemp.count}</td>
            <td>${itemp.first}</td>
            <td>${itemp.last}</td>
        </tr>
    </c:forEach>
</table>

<!-- 遍历Map -->

<%
Map<String, Object> map = new HashMap<String, Object>();
map.put("map1", "aaa");
map.put("map2", "bbb");
map.put("map3", "ccc");
pageContext.setAttribute("map", map);
%>

<c:forEach items="${map}" var="mymap">
    键：${mymap.key} -- 值：${mymap.value} <br>
</c:forEach>
```

5.4 格式化动作标签（不太用）

JSTL提供了格式化和解析数字和日期的标签,我们讨论里面有:formatNumber、formatDate、parseNumber及parseDate.

5.4.1 formatNumber

用于格式化数字，百分比，货币。该标签用指定的格式或精度来格式化数字，将数值型数据转换成指定格式的字符串类型。语法模板如下：

```
<fmt :formatNumber
value="<string>"
type="<string>"
var="<string>"
scope="<string>"/>
```

属性	描述	是否必要	默认值
value	要显示的数字	**是**	无
type	number,currency, 或 percent类型	否	Number
var	存储格式化数字的变量	否	Print to page
scope	var属性的作用域	否	page

```
<fmt:formatNumber value="10" type="currency" var="num"/> ${num} <br>
<fmt:formatNumber value="10" type="percent" /><br>
<fmt:formatNumber value="10" type="currency" /><br>
<%--设置时区--%>
<fmt:setLocale value="en_US"/>
<fmt:formatNumber value="10" type="currency"/>
```

5.4.2 formatDate

标签用于使用不同的方式格式化日期。将Rate型数据转换成指定格式的字符串类型。

属性	描述	是否必要	默认值
value	要显示的日期	是	无
type	DATE, TIME, 或 BOTH	否	date
dateStyle	FULL, LONG, MEDIUM, SHORT, 或 DEFAULT	否	default
timeStyle	FULL, LONG, MEDIUM, SHORT, 或 DEFAULT	否	default
pattern	自定义格式模式	否	无
timeZone	显示日期的时区	否	默认时区
var	存储格式化日期的变量名	否	页面
scope	存储格式化日志变量的范围	否	页面

代码	描述	实例
y	年份小于10，显示不具有前导零的年份。	2002
M	月份数字。一位数的月份没有前导零。	April & 04
d	月中的某一天。一位数的日期没有前导零。	20
h	12小时制的小时。一位数的小时数没有前导零。	12
H	24小时制的小时。一位数的小时数没有前导零。	0
m	分钟。一位数的分钟数没有前导零。	45
s	秒。一位数的秒数没有前导零。	52

```
<%--格式化日期--%>
<%
    request.setAttribute("myDate",new Date());
%>
${myDate} <br>
<fmt:formatDate value="${myDate}"/> <br>
<fmt:formatDate value="${myDate}" type="date"/> <br>
<fmt:formatDate value="${myDate}" type="time"/> <br>
<fmt:formatDate value="${myDate}" type="both"/> <br>
<fmt:formatDate value="${myDate}" type="both"/> <br>
<fmt:formatDate value="${myDate}" type="both" dateStyle="FULL"/> <br>
<fmt:formatDate value="${myDate}" type="both" dateStyle="short"/> <br>
<fmt:formatDate value="${myDate}" pattern="yyyy-MM-dd"/> <br>
```

Fri Jan 17 01:14:54 CST 2025
Jan 17, 2025
Jan 17, 2025
1:14:54 AM
Jan 17, 2025, 1:14:54 AM
Jan 17, 2025, 1:14:54 AM
Friday, January 17, 2025, 1:14:54 AM
1/17/25, 1:14:54 AM
2025-01-17

5.4.3 parseNumber与Date

parseNumber标签用来解析数字，百分数，货币。将数字、货币或百分比类型的字符串转换成数值型。

```
<fmt:parseNumber
    value="<string>"
    type="<string"
    var="<string>"
    scope="<string>"/>
```

属性	描述	是否必要	默认值
value	要解析的数字	否	Body
type	NUMBER, CURRENCY, 或 PERCENT	否	number
var	存储待解析数字的变量	否	Print to page
scope	var属性的作用域	否	page

```
<%--parseNumber的使用--%>
<%--<fmt:setLocale value="zh_CN"/>--%>
<%--转换回中国时区，就可以用￥--%>
<fmt:parseNumber value="100"/><br>
<fmt:parseNumber value="100" type="number"/><br>
<fmt:parseNumber value="100%" type="percent"/><br>
<fmt:parseNumber value="$10.00" type="currency"/><br>
<%--因为前面设置了US时区，这里不能用“￥”，会报错，必须用对应时区的货币标识符才可以--%>
```

parseDate：解析日期，把指定格式的字符串转成日期。

```
<fmt:parseDate value="2024-12-11" type="date"/><br>
<fmt:parseDate value="2024/12/11" pattern="yyyy/MM/dd"/><br>
```

6 JSP实现Login

6.1 项目结构（分层架构）

为了防止出现过多bug，这里使用jdk11和tomcat8来完成。

注：这里用了手动导包（很不好的习惯，我个人主要是想熟悉一下手动导包的流程，仅此而已，实际开发请我完全按照标准开发流程使用maven进行包管理。），本项目的最终版本中，lib都是通过pom.xml进行导入的。

包结构：

```
com.ynu.edu/
├── controller/
├── entity/
├── mapper/
├── service/
├── util/
├── vo/
├── resources/
├── webapp/
|   ├── css/
|   ├── js/
|   ├── WEB-INF/
|   ├── jsp/
|   |   ├── index.jsp
|   |   └── login.jsp
|   └── Mv 编写逻辑.md
└── test/
```

细分：

```
com.ynu.edu/
├── controller/
|   ├── LogoutServlet.java
|   └── UserServlet.java
|
├── entity/
|   └── User.java
|
├── mapper/
|   └── UserMapper.java
|
├── service/
|   └── UserService.java
|
├── util/
|   ├── GetSqlSession.java
```

```
|   └── StringUtil.java
|
├── vo/
|   └── MessageModel.java
|
├── resources/
|   ├── com.ynu.edu.mapper/
|   |   ├── UserMapper.xml
|   |   └── mybatis-config.xml
|   └── mysql.properties
|
├── webapp/
|   ├── css/
|   ├── js/
|   |   └── jquery-3.4.1.js
|   ├── WEB-INF/
|   |   ├── lib/
|   |   ├── web.xml
|   |   └── index.jsp
|   ├── login.jsp
|   └── M+ 编写逻辑.md
|
└── test/
    ├── java/
    |   └── TestSession.java
    └── resources/
```

6.2 配置文件

6.2.1 UserMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ynu.edu.mapper.UserMapper">
    <select id="queryUserByName" parameterType="String"
resultType="com.ynu.edu.entity.User">
        SELECT * FROM user WHERE userName = #{userName}
    </select>
</mapper>
```

6.2.2 mybatis-config.xml

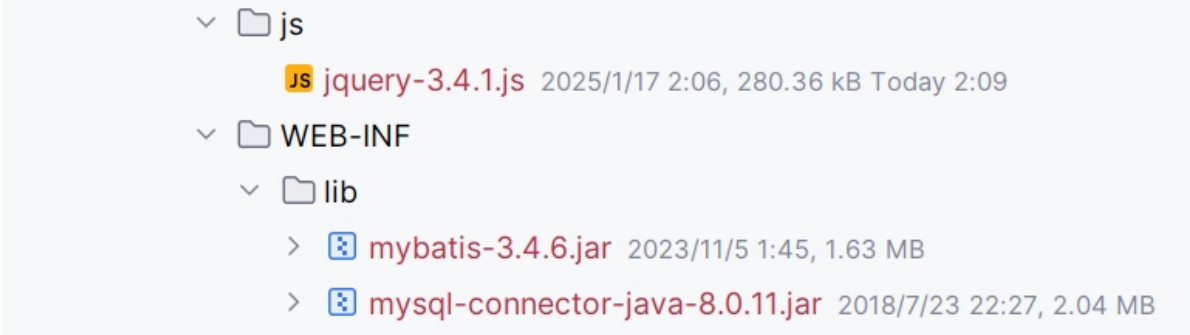
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org/DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

    <properties resource="mysql.properties"/>

    <environments default="development">
```

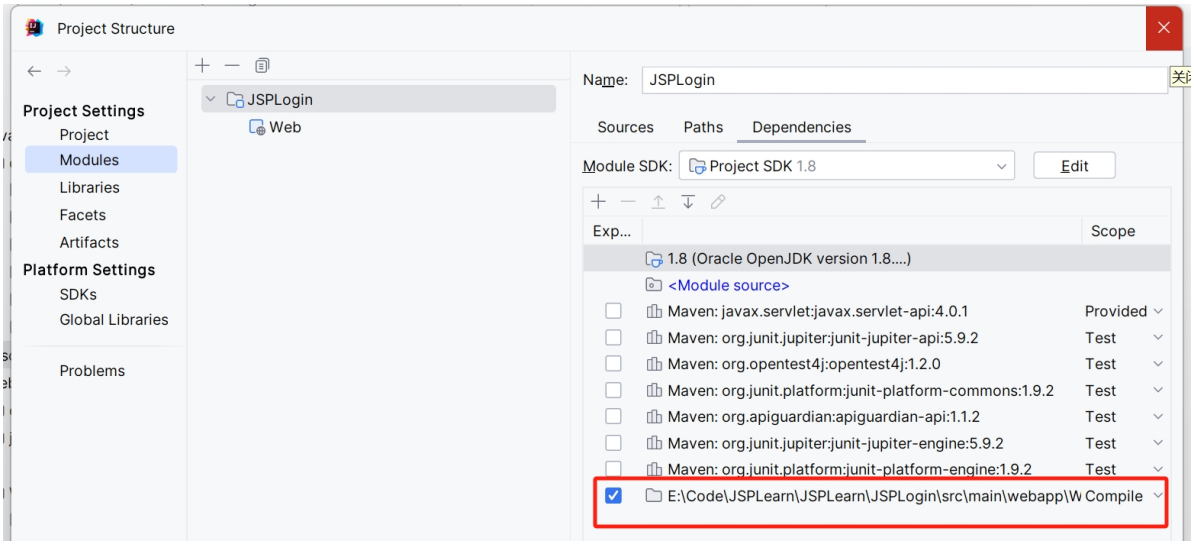
```
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <package name="com.ynu.edu.mapper"/>
    </mappers>

</configuration>
```

6.2.3 mysql.properties

```
driver=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/user?useSSL=false&serverTimezone=UTC
username=root
password=mysql123
```

6.2.4 手动导入jar包（不规范）





手动导入jar包是十分不规范的，但是遇到一些极端情况我们只能这样做。这样仅供本人自己复习用，保证知识面的完整性。

实际使用maven进行包管理，操作如下：

```
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.0</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.28</version>
        </dependency>
```

6.3 功能开发逻辑梳理

```
1. 数据库user表自行准备
2. 前台页面
    登录页面： login.jsp
        用户登录：JS校验
        登录表单验证
            1. 给登录按钮绑定点击事件
            2. 获取uname和upwd
            3. 判断是否为空（先姓名、后密码），span标签给出提示
            4. 都不为空，手动提交表单
    首页

3. 后台实现（一个优秀的后台代码是不能相信前台代码的）
    登录功能
        1. 接收客户端的请求（userName、pwd）
        2. 既然不能相信前台代码，那就要做null判断
            但是市面上我们发现有一些软件如果你有一项信息没填，会把你的所有已填的信息都清空，非常不好。
            解释：注册时候填账号密码、邮箱等，邮箱没填，把你已经填好的账号密码也清空了，是因为后台做的是直接跳转回初始页面，导致你已填好的数据丢失。
        所以这里我们要做数据回显。
            如果参数为空，通过MessageModel返回结果（设置state（success or false）、提示信息、回显数据；直接return
            将消息模型对象设置到Request作用域里面，做请求转发跳转登录页面。
        3. 登录判断习惯：单独判断userName和pwd，不要出现“用户名或密码错误”这样的错误提示，非常不友好。具体原因请自行想象

    代码编写思路：
        1.接收客户端的请求(接收参数:姓名、密码)
        2.参数的非空判断
            if NULL
                通过消息模型对象返回结果(设置状态、设置提示信息、回显数据)
                将消息模型对象设置到request作用域中请求转发
                跳转到登录页面
                return
        3.通过用户姓名查询用户对象
        4.判断用户对象是否为空
            if NULL
                通过消息模型对象返回结果(设置状态、设置提示信息、回显数据)
                将消息模型对象设置到request作用域中请求转发
                跳转到登录页面
                return
        5.将数据库中查询到的用户密码与前台传递的密码作比较
            if not equal
```

```
                通过消息模型对象返回结果(设置状态、设置提示信息、回显数据)
                将消息模型对象设置到request作用域中
                请求转发跳转到登录页面
            if equal
                登录成功
                将用户信息设置到session作用域中（因为你要知道是哪个用户登录，每个用户登录到
不同的页面）
                重定向跳转到首页
    分层结构：
        Controller：接收请求、相应结果
            1. 接受客户端请求（接收参数：name、pwd）
            2. 调用Service层的方法，返回MessageModel
            3. 判断MessageModel的状态码
                 if 失败：将消息模型对象设置到request作用域
                 if 成功：将消息模型中的用户信息设置到session中，重定向到index.jsp
            4. 请求转发跳转到登录页面
        Service：业务逻辑
            1. 参数的非空判断
                 if NULL：状态码、提示信息、回显数据设置到MessageModel中，return
            2. 调用dao层查询方法，通过uname查询用户对象
            3. 判断用户对象是否为空
               状态码、提示信息、回显数据设置到MessageModel中，return
            4. 判断数据库中查询的和前台的uname和pwd进行对比
            5. 登录成功：将success状态、提示信息、用户对象舍之道MessageModel对象中，
return
        Mapper层（DAO）：
            定义对应的接口

4. 分层思想：高内聚低耦合
        Controller层：接收请求、调用Service层、响应结果
        Service层：业务逻辑判断
        Mapper层：接口类、数据库相关操作、mapper.xml
        Entity（Po、Model）：JAVABean实体
        Util层：工具类
        Test：测试类、方法
```

6.4 测试Session

6.4.1 User类：

```java
package com.ynu.edu.entity;

/**
 * @ClassName User
 * @Description 用户类
 * @Author Echo-Nie
 * @Date 2025/1/17 2:28
 * @Version V1.0
 */
public class User {
    private Integer userId;
    private String userName;
    private String pwd;
    private int age;
    //getter and setter
}
```

6.4.2 UserMapper：

```
package com.ynu.edu.mapper;

import com.ynu.edu.entity.User;

public interface UserMapper {
    User queryUserByName(String userName);
}
```

6.4.3 GetSqlSession：

```
package com.ynu.edu.util;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import java.io.IOException;
import java.io.InputStream;

/**
 * @ClassName GetSqlSession
 * @Description 获取数据库session
 * @Author Echo-Nie
 * @Date 2025/1/17 2:44
 * @Version V1.0
 */
public class GetSqlSession {
    public static SqlSession createSqlSession(){
        SqlSessionFactory sqlSessionFactory = null;
        InputStream inpute = null;
        SqlSession session = null;
        try{
            String resource = "mybatis-config.xml";
            inpute= Resources.getResourceAsStream(resource);
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(inpute);
            session = sqlSessionFactory.openSession();
            return session;
        }catch (IOException e){
            e.printStackTrace();
            return null;
        }
    }
}
```

6.4.4 测试SqlSession

```
import com.ynu.edu.entity.User;
import com.ynu.edu.mapper.UserMapper;
import com.ynu.edu.util.GetSqlSession;
import org.apache.ibatis.session.SqlSession;
```

```java
import org.junit.jupiter.api.Test;

/**
 * @ClassName TestSession
 * @Description 测试获取user
 * @Author Echo-Nie
 * @Date 2025/1/17 13:46
 * @Version V1.0
 */
public class TestSession {
    @Test
    public void Test1(){
        SqlSession session = GetSqlSession.createSqlSession();
        UserMapper userMapper = session.getMapper(UserMapper.class);
        User user = userMapper.queryUserByName("admin");
        System.out.println(user);
    }
    //输出如下：
    //com.ynu.edu.entity.User@5c44c582
    //Process finished with exit code 0
}
```

6.5 编写项目代码

6.5.1 消息模型对象

```java
package com.ynu.edu.vo;

/**
 * @ClassName MessageModel
 * @Description 消息模型对象，做数据响应的：200表示成功，400表示失败
 *              用字符串表示
 *              回显数据：Object
 * @Author Echo-Nie
 * @Date 2025/1/17 14:11
 * @Version V1.0
 */
public class MessageModel {
    private String code = "200";//状态码，200成功，400失败
    private String msg = "成功";
    private Object object;
    //getter setter
}
```

6.5.2 编写Servlet

```java
package com.ynu.edu.controller;

import com.ynu.edu.service.UserService;
import com.ynu.edu.vo.MessageModel;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * @ClassName UserServlet
 * @Description
 * @Author Echo-Nie
 * @Date 2025/1/17 14:15
 * @Version V1.0
 */
@WebServlet("/login")
public class UserServlet extends HttpServlet {
    //实例化UserService对象
    private UserService userService = new UserService();

    /**
     * @return void
     * @Author Echo-Nie
     * @Description 用户登录：
     * 1. 接受客户端请求（接收参数：name、pwd）
     * 2. 调用Service层的方法，返回MessageModel
     * 3. 判断MessageModel的状态码
     * if 失败：将消息模型对象设置到request作用域
     * if 成功：将消息模型中的用户信息设置到session中，重定向到index.jsp
     * 4. 请求转发跳转到登录页面
     * @Date 14:35 2025/1/17
     * @Param [request, response]
     */
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
//        1. 接受客户端请求（接收参数：name、pwd）
        String uname = request.getParameter("uname");
        String upwd = request.getParameter("upwd");
//        2. 调用Service层的方法，返回MessageModel
        MessageModel messageModel = userService.userLogin(uname, upwd);
//        3. 判断MessageModel的状态码
        if (messageModel.getCode().equals("200")) {//成功
            request.getSession().setAttribute("user", messageModel.getObject());
            response.sendRedirect("index.jsp"); // 使用重定向
        } else {//失败
            request.setAttribute("messageModel", messageModel);
            request.getRequestDispatcher("login.jsp").forward(request,
response);
        }
    }
}
```

6.5.3 编写Service

```
package com.ynu.edu.service;

import com.ynu.edu.entity.User;
import com.ynu.edu.mapper.UserMapper;
import com.ynu.edu.util.GetSqlSession;
import com.ynu.edu.util.StringUtil;
import com.ynu.edu.vo.MessageModel;
```

```java
import org.apache.ibatis.session.SqlSession;

/**
 * @ClassName UserService
 * @Description 登录页面的业务逻辑层
 * @Author Echo-Nie
 * @Date 2025/1/17 14:14
 * @Version V1.0
 */
public class UserService {
    /**
    * @Author Echo-Nie
    * @Description
    1. 参数的非空判断
        if NULL：状态码、提示信息、回显数据设置到MessageModel中，return
    2. 调用dao层查询方法，通过uname查询用户对象
    3. 判断用户对象是否为空
        状态码、提示信息、回显数据设置到MessageModel中，return
    4. 判断数据库中查询的和前台的uname和pwd进行对比
    5. 登录成功：将success状态、提示信息、用户对象舍之道MessageModel对象中，return

    * @Date 14:43 2025/1/17
    * @Param [uname, upwd]
    * @return com.ynu.edu.vo.MessageModel
    **/
    public MessageModel userLogin(String uname, String upwd) {
        MessageModel messageModel = new MessageModel();
//        数据回显
        User u = new User();
        u.setUserName(uname);
        u.setPwd(upwd);
        messageModel.setObject(u);

//        1. 参数的非空判断
        if(StringUtil.isEmpty(uname)||StringUtil.isEmpty(upwd)){
//            if NULL：状态码、提示信息、回显数据设置到MessageModel中，return
            messageModel.setCode("400");
            messageModel.setMsg("用户名和密码不能为空！！");

        }
//        2. 调用dao层查询方法，通过uname查询用户对象
        SqlSession session = GetSqlSession.createSqlSession();
        UserMapper userMapper = session.getMapper(UserMapper.class);
        User user = userMapper.queryUserByName(uname);
//        3. 判断用户对象是否为空
        if(user==null){
            messageModel.setCode("400");
            messageModel.setMsg("用户不存在");
            return messageModel;
        }
//        4. 判断数据库中查询的和前台的uname和pwd进行对比
        if(!upwd.equals(user.getPwd())){
            messageModel.setCode("400");
            messageModel.setMsg("密码错误！");
            return messageModel;
        }
        messageModel.setObject(user);
```

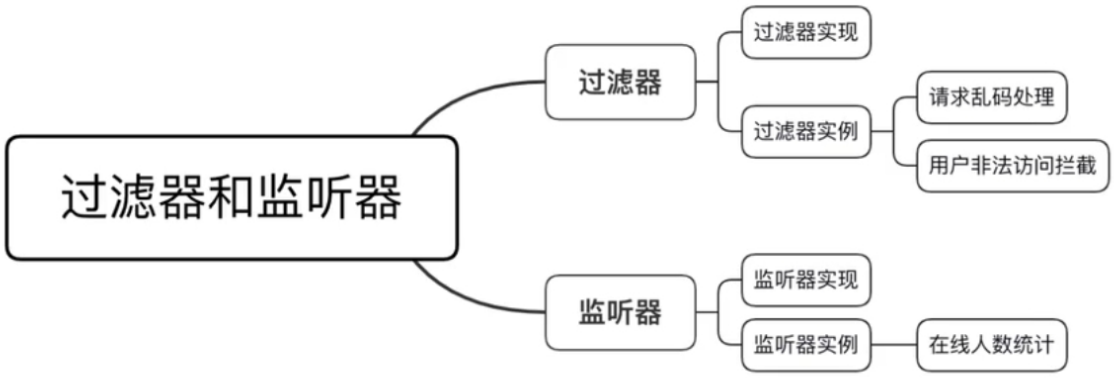
```
        return messageModel;
    }
}
```

6.5.4 login.jsp（不含css）

```
<%--
  User: Echo-Nie
  Date: 2025/1/17
  Time: 1:51
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>用户登录</title>
    <link rel="stylesheet" type="text/css" href="css/login.css">
</head>
<body>
<div id="loginForm">
    <form action="login" method="post">
        <div class="form-group">
            <label for="uname">姓名：</label>
            <input type="text" id="uname" name="uname"
value="${messageModel.object.userName}">
        </div>
        <div class="form-group">
            <label for="upwd">密码：</label>
            <input type="password" id="upwd" name="upwd"
value="${messageModel.object.pwd}">
        </div>
        <span id="msg" style="font-size: 12px;color: red">${messageModel.msg}
</span><br>
        <button type="submit" id="loginBtn">登录</button>
        <button type="button">注册</button>
    </form>
</div>

<script type="text/javascript" src="js/jquery-3.4.1.js"></script>
<script type="text/javascript">
    <%--      登录表单验证
                1. 给登录按钮绑定点击事件
                2. 获取uname和upwd
                3. 判断是否为空（先姓名、后密码），span标签给出提示
                4. 都不为空，手动提交表单
    --%>
    $("#loginBtn").click(function () {
        //获取uname和pwd
        var uname = $("#uname").val();
        var upwd = $("#upwd").val();

        if (isEmpty(uname)) {
            $("#msg").html("用户名为空！");
            return;
        }
        if (isEmpty(upwd)) {
            $("#msg").html("密码为空！");
```

```
            return;
        }
        //都不为空才能登录
        $("#loginForm").submit();
    });

    /*
    判断字符串是否为空
    * */
    function isEmpty(str) {
        return str == null || str.trim() === "";
    }
</script>
</body>
</html>
```

7 过滤器Filter



7.1 Filter机理

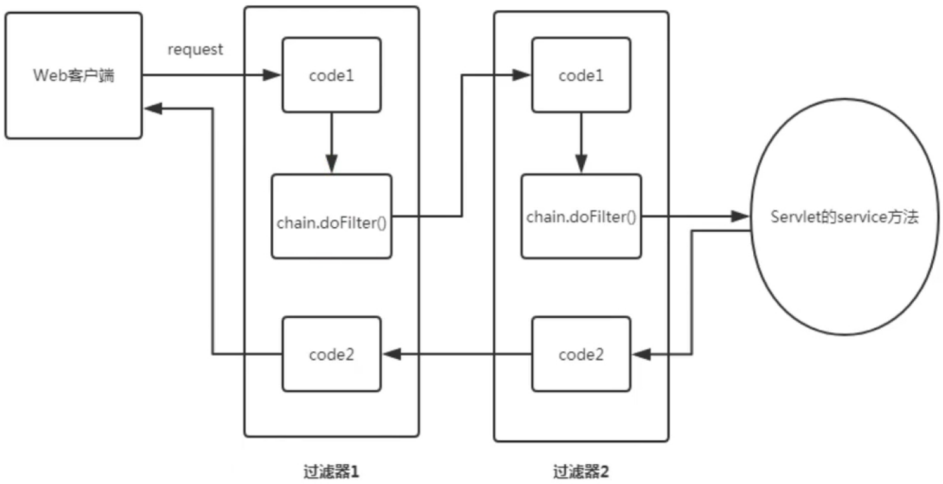
Filter，用于在 Servlet之外对 Request 或者 Response 进行修改。用于对用户请求进行预处理，也可以对 HttpServletResponse 进行后处理。

Filter完整流程: Filter 对用户请求进行预处理，接着将请求交给 Serviet进行处理并生成响应，最后 Filter 再 对服务器响应进行后处理。在一个 web 应用中，可以开发编写多个 Filter，这些 Filter 组合 起来称之为一个 Filter 链。

大白话：Filter 作为快递站点的安检环节。包裹（用户请求）到达站点后，先要经过安检（Filter 预处理），检查是否有违禁品等。安检通过后，包裹才会被送到仓库（Servlet 处理请求）。

包裹从仓库发出后，再次经过安检（Filter 后处理），这次检查包装是否完好等，确保送到客户手中的包裹是合格的。多个安检环节就构成了安检链，类似于多个 Filter 组成的 Filter 链。

对于多个过滤器：先配置的先执行（请求时的执行顺序）；响应的时候顺序相反。如下图：



在 HttpServletRequest 到达 Servlet 之前，拦截客户的 HttpServletRequest。根据需要检查 HttpServletRequest，也可以修改 HttpServletRequest 头和数据。

在HttpServletResponse 到达客户端之前，拦截 HttpServletResponse。根据需要检查 HttpServletResponse，也可以修改 HttpServletResponse头和数据。

7.2 FilterTest实现

通过实现一个叫做javax.servlet.Fileter的接口来实现一个过滤器，其中定义了 三个方法，init()、doFilter()、destroy()分别在相应的时机执行。后期观察生命周期。

Step1：编写 java 类实现 Filter接口，并实现其 doFiter 方法,
Step2：通过@WebFilter注解设置它所能拦截的资源。

```java
package com.ynu.edu.filter;

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

/**
 * @ClassName FilterTest1
 * @Description 过滤器测试
 * @Author Echo-Nie
 * @Date 2025/1/17 21:48
 * @Version V1.0
 */
@WebFilter("/ser01")
public class FilterTest1 implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("FilterTest1 init...");
    }

    /**
    * @Author Echo-Nie
    * @Description 过滤方法
    * @Date 21:51 2025/1/17
    * @Param [servletRequest, servletResponse, filterChain]
    * @return void
    **/
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        //放行资源
        filterChain.doFilter(servletRequest,servletResponse);
        //如果不放行的话这里就一直被拦截，永远不会进入Servlet01
    }

    @Override
    public void destroy() {
        System.out.println("FilterTest1 destroy...");
    }
}
```

```java
package com.ynu.edu.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * @ClassName Servlet01
 * @Description
 * @Author Echo-Nie
 * @Date 2025/1/17 21:52
 * @Version V1.0
```

```
 */
@WebServlet("/ser01")
public class Servlet01 extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("Servlet01 start...");
    }
}
```

```
package com.ynu.edu.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * @ClassName Servlet01
 * @Description
 * @Author Echo-Nie
 * @Date 2025/1/17 21:52
 * @Version V1.0
 */
@WebServlet("/ser02")
public class Servlet02 extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("Servlet02 start...");
    }
}
```

```
Connected to server
[2025-01-17 09:56:54,382] Artifact FilterTest:war exploded: A
17-Jan-2025 21:56:54.566 警告 [RMI TCP Connection(2)-127.0.0.1
FilterTest1 init...
[2025-01-17 09:56:54,761] Artifact FilterTest:war exploded: A
[2025-01-17 09:56:54,761] Artifact FilterTest:war exploded: D
Servlet01 start...
17-Jan-2025 21:57:04.107 信息 [localhost-startStop-1] org.apa
17-Jan-2025 21:57:04.210 信息 [localhost-startStop-1] org.apa
```

如上图：对于FilterTest1就会有init出现，但是FilterTest2没有，因为拦截器现在只拦截了FilterTest1；带*号就可以拦截所有，实际开发中一般都是带星号

```
package com.ynu.edu.filter;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;
```

```java
/**
 * @ClassName FilterTest1
 * @Description 过滤器测试
 * @Author Echo-Nie
 * @Date 2025/1/17 21:48
 * @Version V1.0
 */
@WebFilter("/*")
public class FilterTestAll implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("FilterTest1 init...");
    }

    /**
    * @Author Echo-Nie
    * @Description 过滤方法
    doFilter放行方法前，做请求拦截
    doFilter放行方法后，做响应拦截
    * @Date 21:51 2025/1/17
    * @Param [servletRequest, servletResponse, filterChain]
    * @return void
    **/
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        //放行资源
        filterChain.doFilter(servletRequest,servletResponse);
        //如果不放行的话这里就一直被拦截，永远不会进入Servlet01
    }

    @Override
    public void destroy() {
        System.out.println("FilterTest1 destroy...");
    }
}
```

7.3 LoginFilter

1. **预处理**：

   在用户登录请求到达 `UserServlet` 之前，检查用户是否已经登录。

   如果用户已经登录（即 `session` 中存在 `user` 对象），则直接重定向到首页 `index.jsp`，避免重复登录。

2. **后处理**：

   在响应返回给客户端之前，设置响应头，防止浏览器缓存页面内容。

```java
package com.ynu.edu.filter;

import com.ynu.edu.vo.MessageModel;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
```

```java
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

/**
 * @ClassName LoginFilter
 * @Description 登录过滤器
 * @Author Echo-Nie
 * @Date 2025/1/17 15:00
 * @Version V1.0
 */
@WebFilter("/login")
public class LoginFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // 初始化
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
            throws IOException, ServletException {
        // 将ServletRequest和ServletResponse转换为HttpServletRequest和
HttpServletResponse
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        // 1. 预处理：检查用户是否已经登录
        HttpSession session = httpRequest.getSession();
        if (session.getAttribute("user") != null) {
            // 如果用户已经登录，直接重定向到首页，避免重复登录
            httpResponse.sendRedirect("index.jsp");
            return; // 结束过滤链，不再继续执行后续的Filter和Servlet
        }

        // 2. 继续执行后续的Filter和Servlet
        chain.doFilter(request, response);

        // 3. 后处理：在响应返回给客户端之前，可以做一些额外的工作
        httpResponse.setHeader("Cache-Control", "no-cache, no-store, must-
revalidate");
        httpResponse.setHeader("Pragma", "no-cache"); // HTTP 1.0
        httpResponse.setDateHeader("Expires", 0);
    }

    @Override
    public void destroy() {
        // 销毁
    }
}
```

7.4 非法访问拦截

需要放行的资源:

1. 指定页面，放行 (无需登录即可访问的页面 例如:登录页面、注册页面等)
2. 静态资源，放行 (image、js、css文件等)
3. 指定操作，放行(无需登录即可执行的操作 例如:登录操作、注册操作)3.
4. 登录状态，放行(判断session中用户信息是否为类

```
package com.ynu.edu.filter;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;

/**
 * @ClassName AuthFilter
 * @Description 用户登录验证过滤器
 * @Author Echo-Nie
 * @Date 2025/1/18
 * @Version V1.0
 */
@WebFilter("/*") // 拦截所有请求
public class AuthFilter implements Filter {

    // 放行的指定页面
    private static final List<String> ALLOWED_PAGES = Arrays.asList(
            "/login.jsp", "/register.jsp"
    );

    // 放行的静态资源路径
    private static final List<String> ALLOWED_RESOURCES = Arrays.asList(
            "/css/", "/js/", "/images/"
    );

    // 放行的指定操作
    private static final List<String> ALLOWED_ACTIONS = Arrays.asList(
            "/login", "/register"
    );

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // 初始化方法，可以在这里做一些初始化工作
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
            throws IOException, ServletException {
        // 将 ServletRequest 和 ServletResponse 转换为 HttpServletRequest 和
HttpServletResponse
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        // 获取请求的 URI
        String requestURI = httpRequest.getRequestURI();
```

```
        String contextPath = httpRequest.getContextPath();
        String path = requestURI.substring(contextPath.length());

        // 检查是否为放行的指定页面
        if (ALLOWED_PAGES.contains(path)) {
            chain.doFilter(request, response);
            return;
        }

        // 检查是否为放行的静态资源
        if (isAllowedResource(path)) {
            chain.doFilter(request, response);
            return;
        }

        // 检查是否为放行的指定操作
        if (ALLOWED_ACTIONS.contains(path)) {
            chain.doFilter(request, response);
            return;
        }

        // 检查用户是否已登录
        HttpSession session = httpRequest.getSession(false); // 如果不存在
Session，则返回 null
        boolean isLoggedIn = (session != null && session.getAttribute("user") !=
null);

        if (isLoggedIn) {
            // 用户已登录，放行请求
            chain.doFilter(request, response);
        } else {
            // 用户未登录，重定向到登录页面
            httpResponse.sendRedirect(contextPath + "/login.jsp");
        }
    }

    @Override
    public void destroy() {
        // 销毁方法，可以在这里做一些清理工作
    }

    /**
     * 检查是否为放行的静态资源
     */
    private boolean isAllowedResource(String path) {
        for (String resource : ALLOWED_RESOURCES) {
            if (path.startsWith(resource)) {
                return true;
            }
        }
        return false;
    }
}
```

8 监听器Listener

监听器是Servlet 中一种的特殊的类，能帮助监听 web 中的特定事件，比如ServletContextHttpSession，ServletRequest 的创建和销毁;变量的创建、销毁和修改等。可以在某些动作前后增加处理，实现监控。

8.1 监听器的作用

监听器用于监听 Web 应用中的事件，例如：

- 请求的创建和销毁。
- Session 的创建和销毁。
- 应用上下文（ServletContext）的初始化和销毁。
- 属性的添加、修改和删除。

通过监听器，开发者可以在这些事件发生时执行自定义的逻辑，例如初始化资源、记录日志、统计在线用户等。

8.2 监听器的分类

监听器分为三类，共八种：

(1) 监听生命周期

这类监听器用于监听对象的创建和销毁事件。

1. `ServletRequestListener`：

   监听 HTTP 请求的创建和销毁。

   主要方法：

   `requestInitialized(ServletRequestEvent sre)`：请求创建时触发。

   `requestDestroyed(ServletRequestEvent sre)`：请求销毁时触发。

2. `HttpSessionListener`：

   监听 Session 的创建和销毁。

   主要方法：

   `sessionCreated(HttpSessionEvent se)`：Session 创建时触发。

   `sessionDestroyed(HttpSessionEvent se)`：Session 销毁时触发。

3. `ServletContextListener`：

   监听应用上下文（ServletContext）的初始化和销毁。

   主要方法：

   `contextInitialized(ServletContextEvent sce)`：应用启动时触发。

   `contextDestroyed(ServletContextEvent sce)`：应用关闭时触发。

---

(2) 监听值的变化

这类监听器用于监听属性的添加、修改和删除事件。

1. `ServletRequestAttributeListener`：

监听请求属性（`request.setAttribute()`）的变化。

主要方法：

`attributeAdded(ServletRequestAttributeEvent srae)`：属性添加时触发。

`attributeRemoved(ServletRequestAttributeEvent srae)`：属性删除时触发。

`attributeReplaced(ServletRequestAttributeEvent srae)`：属性修改时触发。

2. `HttpSessionAttributeListener`：

   监听 Session 属性（`session.setAttribute()`）的变化。

   主要方法：

   `attributeAdded(HttpSessionBindingEvent se)`：属性添加时触发。

   `attributeRemoved(HttpSessionBindingEvent se)`：属性删除时触发。

   `attributeReplaced(HttpSessionBindingEvent se)`：属性修改时触发。

3. `ServletContextAttributeListener`：

   监听应用上下文属性（`servletContext.setAttribute()`）的变化。

   主要方法：

   `attributeAdded(ServletContextAttributeEvent scae)`：属性添加时触发。

   `attributeRemoved(ServletContextAttributeEvent scae)`：属性删除时触发。

   `attributeReplaced(ServletContextAttributeEvent scae)`：属性修改时触发。

---

(3) 针对 Session 中的对象

这类监听器用于监听 Session 中 Java 对象（JavaBean）的绑定和解绑事件。

1. `HttpSessionBindingListener`：

   监听 Java 对象绑定到 Session 或从 Session 中解绑的事件。

   需要 JavaBean 直接实现该接口。

   主要方法：

   `valueBound(HttpSessionBindingEvent event)`：对象绑定到 Session 时触发。

   `valueUnbound(HttpSessionBindingEvent event)`：对象从 Session 中解绑时触发。

2. `HttpSessionActivationListener`：

   监听 Session 中 Java 对象的激活和钝化事件（在分布式环境中使用）。

   需要 JavaBean 直接实现该接口。

   主要方法：

   `sessionWillPassivate(HttpSessionEvent se)`：对象钝化时触发。

   `sessionDidActivate(HttpSessionEvent se)`：对象激活时触发。

8.3 监听器场景

1. **统计在线用户**：

使用 `HttpSessionListener` 监听 Session 的创建和销毁，统计在线用户数量。

2. **初始化全局资源**：

   使用 `ServletContextListener` 在应用启动时初始化全局资源（如数据库连接池）。

3. **记录请求日志**：

   使用 `ServletRequestListener` 记录每个请求的访问日志。

4. **监听属性变化**：

   使用 `HttpSessionAttributeListener` 监听 Session 中属性的变化，实现特定业务逻辑。

5. **对象绑定和解绑事件**：

   使用 `HttpSessionBindingListener` 监听 Java 对象在 Session 中的绑定和解绑事件。

8.4 监听器的配置

1. **注解配置**：

   在 Servlet 3.0 及以上版本中，可以使用 `@WebListener` 注解配置监听器。

```
@WebListener
public class MySessionListener implements HttpSessionListener {
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("Session created: " +
se.getSession().getId());
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("Session destroyed: " +
se.getSession().getId());
    }
}
```

2. **XML 配置**：

   在 `web.xml` 中配置监听器。

```
<listener>
    <listener-class>com.ynu.edu.listener.MySessionListener</listener-
class>
</listener>
```

8.5 测试样例

Listener类：

```
package com.ynu.edu.listener;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
```

```java
/**
 * @ClassName Listener01
 * @Description 监听器01号
 * @Author Echo-Nie
 * @Date 2025/1/17 23:04
 * @Version V1.0
 */
@WebListener
public class Listener01 implements HttpSessionListener {
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("Session Created...");
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("Session Destroy...");
    }
}
```

创建Session：

```java
package com.ynu.edu.servlet;

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

/**
 * @ClassName ServletListener01
 * @Description
 * @Author Echo-Nie
 * @Date 2025/1/17 23:06
 * @Version V1.0
 */
@WebServlet("/s01")
public class ServletListener01 extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("Servlet-Listener01 Create...");
        HttpSession httpSession = req.getSession();

    }
}
```

销毁Session：

```java
package com.ynu.edu.servlet;

import javax.servlet.ServletException;
```

```java
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

/**
 * @ClassName ServletListener01
 * @Description
 * @Author Echo-Nie
 * @Date 2025/1/17 23:06
 * @Version V1.0
 */
@WebServlet("/s02")
public class ServletListener02 extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("Servlet-Listener01 Destroy...");
        req.getSession().invalidate();

    }
}
```

9 在线人数监控

9.1 开发逻辑

在线人数统计：

当用户访问网站时，会创建一个 Session，`OnlineUserListener` 监听到 Session 创建事件，增加在线人数。

当用户退出或 Session 超时时，`OnlineUserListener` 监听到 Session 销毁事件，减少在线人数。

显示在线人数：

用户可以通过点击链接访问 `/onlineUsers`，查看当前在线人数。

退出功能：

用户点击“退出”按钮后，`LogoutServlet` 会销毁当前 Session 并减少在线人数，然后重定向到首页。

代码编写：

核心工具类：计数功能（`OnlineUserCounter`）

监听器：实时更新逻辑（`OnlineUserListener`）

Servlet：业务逻辑登录登出（`OnlineUserServlet` 和 `LogoutServlet`）

前端页面：用户交互（`index.jsp`）

测试和调试

9.2 代码层

index.jsp

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>在线人数监控</title>
</head>
<body>
    <h1>欢迎访问在线人数监控系统</h1>
    <p><a href="onlineUsers">查看当前在线人数</a></p>
    <form action="logout" method="post">
        <button type="submit">退出</button>
    </form>
</body>
</html>
```

LogoutServlet:

```
package com.ynu.edu.servlet;

import com.ynu.edu.util.OnlineUserCounter;

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

@WebServlet("/logout")
public class LogoutServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // 获取当前 Session
        HttpSession session = req.getSession(false);
        if (session != null) {
            // 销毁 Session
            session.invalidate();
            // 减少在线人数
            OnlineUserCounter.decrement();
        }

        // 重定向到首页
        resp.sendRedirect("index.jsp");
    }
}
```

OnlineUserListener:

```
package com.ynu.edu.listener;

import com.ynu.edu.util.OnlineUserCounter;

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

@WebListener
public class OnlineUserListener implements HttpSessionListener {

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        // Session 创建时，增加在线人数
        OnlineUserCounter.increment();
        System.out.println("Session 创建，当前在线人数: " +
OnlineUserCounter.getOnlineUsers());
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        // Session 销毁时，减少在线人数
        OnlineUserCounter.decrement();
        System.out.println("Session 销毁，当前在线人数: " +
OnlineUserCounter.getOnlineUsers());
    }
}
```

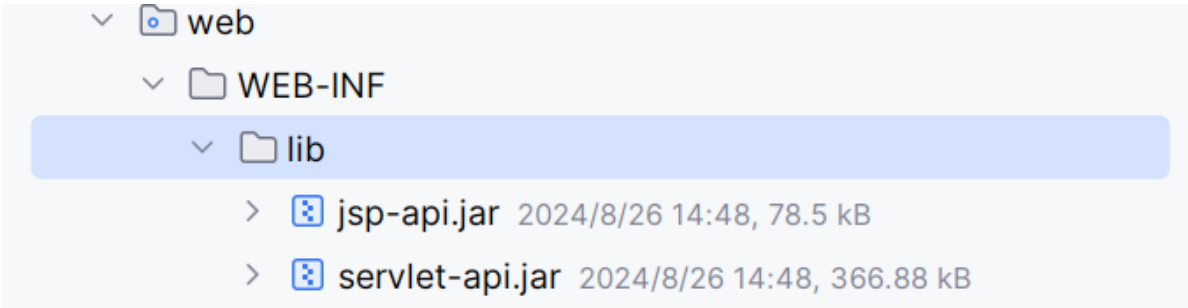
OnlineUserCounter:

```
package com.ynu.edu.util;

import java.util.concurrent.atomic.AtomicInteger;

public class OnlineUserCounter {
    private static final AtomicInteger onlineUsers = new AtomicInteger(0);

    // 增加在线人数
    public static void increment() {
        onlineUsers.incrementAndGet();
    }

    // 减少在线人数
    public static void decrement() {
        onlineUsers.decrementAndGet();
    }

    // 获取当前在线人数
    public static int getOnlineUsers() {
        return onlineUsers.get();
    }
}
```

附录：Tips

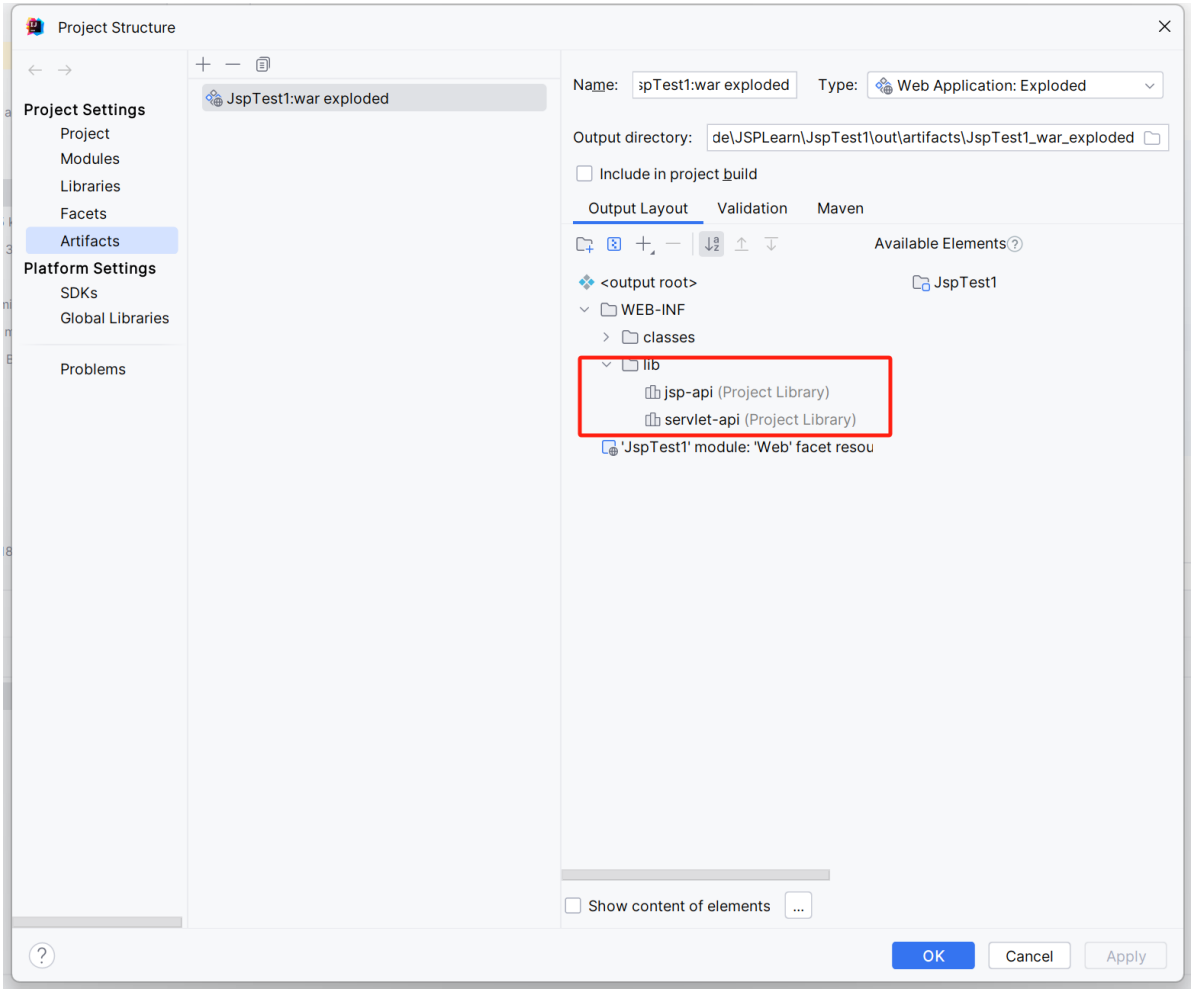
1 发现使用“out”报错

在tomcat下找到下面两个jar包，放到lib下面。



接着，右键，add as Library

最后得到下图所示即可解决问题。



2 静态include的变量重复定义

```
<%@include file="header.jsp" %>
<h2>主体部分</h2>
<%
    int num = 10;
    out.print(num);
%>
<%@include file="footer.jsp" %>
```

```
<html>
<head>
    <title>Footer底部</title>
</head>
<body>
<%
    int num = 11;
%>
-------------<br>
这是尾部<br>
-------------<br>
</body>
</html>
```

如果你在主体中有变量num，然后在foot或者head中也有同名变量，将会报如下错误。

```
An error occurred at line: [15] in the jsp file: [/footer.jsp]
Duplicate local variable num
```

```
org.apache.jasper.JasperException: Unable to compile class for JSP:

An error occurred at line: [15] in the jsp file: [/footer.jsp]
Duplicate local variable num
12: </head>
13: <body>
14: <%
15:     int num = 11;
16: %>
17: -------------<br>
18: 这是尾部<br>
```

3 一个超蠢版本问题（2024.11.09更）

在写简单的登录页面demo的过程中发生了一个404问题。

```
HTTP Status 404 – Not Found
Type Status Report

Message /JSPTest1/loginServlet

Description The origin server did not find a current representation for the
target resource or is not willing to disclose that one exists.

Apache Tomcat/8.5.31
```

这里是因为是用的8.5，JavaEE被Oracle捐献给Apache了。目前JavaEE的最高版本是 JavaEE8；Apache把JavaEE换名了，以后不叫JavaEE了，以后叫做 jakarta EE。所以JavaEE8版本升级之后叫做JakartaEE9

JavaEE8的时候对应的Servlet类名是：javax.servlet

JakartaEE9的时候对应的Servlet类名是：jakarta.servlet

在Tomcat9之前是javax，Tomcat9之后是Jakarta。如果不替换Tomcat版本将出现以下问题：

```
javax.servlet.ServletException: Class [LoginServlet] is not a Servlet

org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.jav
a:496)
    org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:81)

org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.
java:650)
    org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:342)
    org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:803)

org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)

org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.ja
va:790)

org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:14
68)

org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)

java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.j
ava:1144)

java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:642)

org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:6
1)
    java.base/java.lang.Thread.run(Thread.java:1623)
```

所以最好的办法就是下个**Tomcat9之后**的版本然后配置一下就没问题了，很蠢的问题，花了我十多分钟。。。。。记录一下自己的傻逼时刻。

补充：可以使用

```
@WebServlet("/loginServlet")
```

也可以在web.xml中配置：

```
<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/loginServlet</url-pattern>
</servlet-mapping>
```

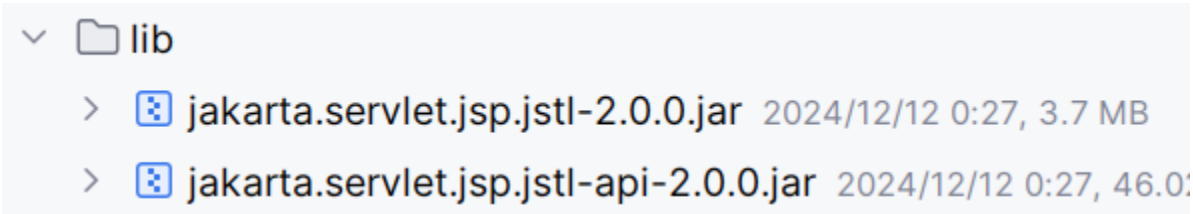
4 JSTL导包问题

如果使用Tomcat8的话，直接去官网下载前面的版本即可。但是

如果使用的是Tomcat10的话，不要下载1.1.2了，要用以下两个包（Jakarta的，官网那个是javax的。直接点击下面两个网址就能下载）：

https://repo.maven.apache.org/maven2/jakarta/servlet/jsp/jstl/jakarta.servlet.jsp.jstl-api/2.0.0/jakarta.servlet.jsp.jstl-api-2.0.0.jar

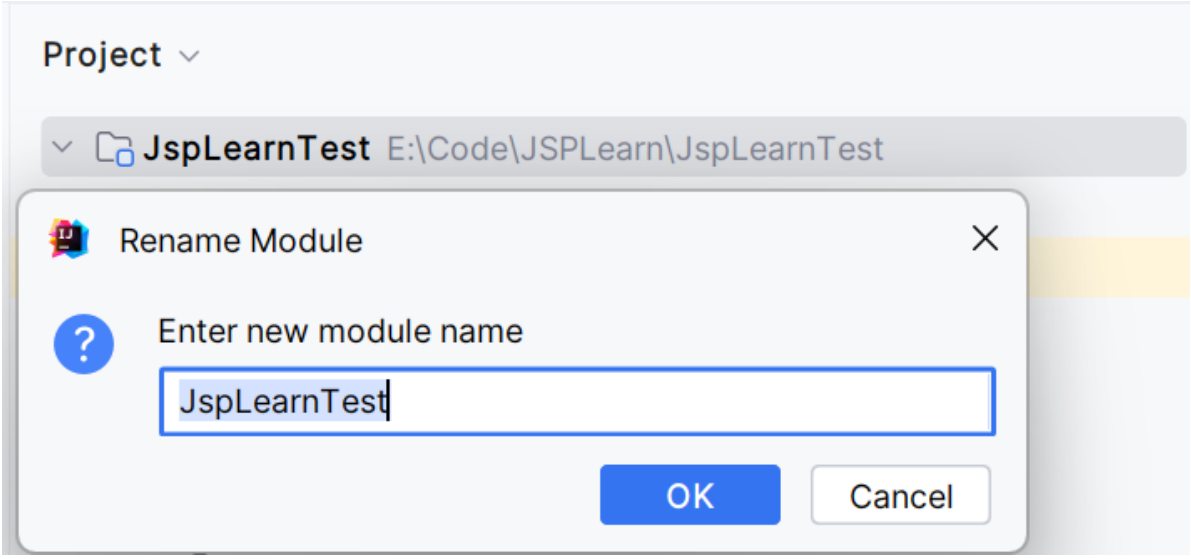
https://repo.maven.apache.org/maven2/org/glassfish/web/jakarta.servlet.jsp.jstl/2.0.0/jakarta.servlet.jsp.jstl-2.0.0.jar



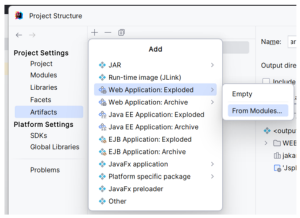
5 项目重命名问题

一个很简单的问题，大部分人也不会犯，但是我手贱试了一下，报错的时候还是有点慌的...

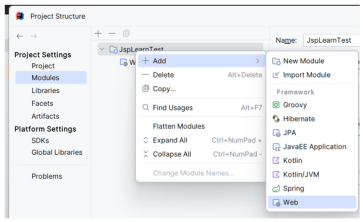
如果把项目重命名，如下：



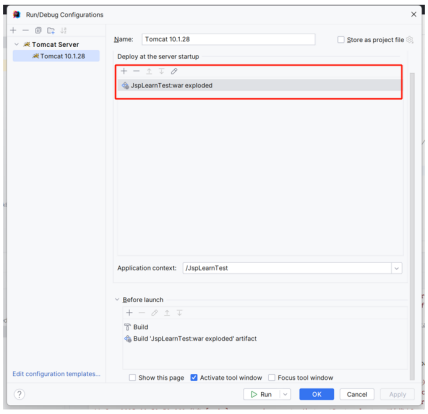
会导致404，原因很简单，假设原项目是JspLearn，被你改成了JspLearnTest，那么原本你的web项目的路径都是原来的，比如说是xxx/xxx/JspLearn路径下，但是你在xxx/xxx/JspLearnTest路径下会找不到，很合理。这个时候只需要把web重新配置即可，如下图：



1



2



3

把Artifacts重新创建，在Module里面重新添加web配置，最后把tomcat原来的Development和Server重新添加即可。

6 究极大问题mybatis

首先，网上有很多其他方法，但是我试了都不行。最后发现了一个这样的小bug。

下面是我报错的一个测试代码

```
public class Test {
    public static void main(String[] args) {
        SqlSession session = GetSqlSession.createSqlSession();
        UserMapper userMapper = session.getMapper(UserMapper.class);
        User user = userMapper.queryUserByName("admin");
        System.out.println(user);
    }
}
```

如果你发现报错：Exception in thread "main" org.apache.ibatis.binding.BindingException: Invalid bound statement (not found): com.ynu.edu.mapper.UserMapper.queryUserByName

尝试一下把创建包名改成：com/ynu/edu/mapper，而不是com.ynu.edu.mapeer

7 简单问题——请求乱码

在 Java Web 开发中，HTTP 请求的字符编码可能会导致乱码问题，尤其是在处理中文或其他非 ASCII 字符时。乱码问题的出现通常与以下因素有关：

- Tomcat 版本：Tomcat 8 及以上版本对字符编码的处理与 Tomcat 7 及以下版本不同。
- 请求类型：POST 请求和 GET 请求的字符编码处理方式不同。

POST 请求的乱码处理：

无论是 Tomcat 8 及以上版本，还是 Tomcat 7 及以下版本，POST 请求都可能出现乱码。

在 Servlet 中，通过 `request.setCharacterEncoding("UTF-8");` 设置请求的字符编码为 UTF-8。这行代码需要在获取请求参数之前调用，通常在 `doPost` 方法的最开始部分。

代码：

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8"); // 设置请求编码为 UTF-8
    String name = request.getParameter("name"); // 获取参数
    // 其他逻辑
}
```

GET 请求的乱码处理

Tomcat 8 及以上版本：GET 请求默认不会出现乱码，不需要额外处理。

Tomcat 7 及以下版本：GET 请求可能会出现乱码，需要手动处理。

对于 Tomcat 7 及以下版本，GET 请求的参数需要手动进行字符编码转换。

使用 `new String(request.getParameter("参数名").getBytes("ISO-8859-1"), "UTF-8");` 将参数从 ISO-8859-1 编码转换为 UTF-8 编码。

代码：

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String name = request.getParameter("name"); // 获取参数
    if (isTomcat7OrBelow()) { // 判断是否是 Tomcat 7 及以下版本
        name = new String(name.getBytes("ISO-8859-1"), "UTF-8"); // 转换编码
    }
    // 其他逻辑
}
```

POST 请求：

无论 Tomcat 版本如何，都需要调用 `request.setCharacterEncoding("UTF-8");` 来设置请求编码。

GET 请求：

如果是 Tomcat 8 及以上版本，不需要额外处理。

如果是 Tomcat 7 及以下版本，需要手动将参数从 ISO-8859-1 编码转换为 UTF-8 编码。

8 奇葩问题 —— Session自动创建三次

```
17-Jan-2025 23:43:53.010 信息 [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
17-Jan-2025 23:43:53.010 信息 [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet Engine: Apache Tomcat/8.5.31
17-Jan-2025 23:43:53.016 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
17-Jan-2025 23:43:53.022 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]
17-Jan-2025 23:43:53.025 信息 [main] org.apache.catalina.startup.Catalina.start Server startup in 36 ms
Connected to server
[2025-01-17 11:43:53,360] Artifact FilterTest:war exploded: Artifact is being deployed, please wait...
17-Jan-2025 23:43:53.495 警告 [RMI TCP Connection(2)-127.0.0.1] org.apache.tomcat.util.descriptor.web.WebXml.setVersion Unknown version string [4.0]. D
FilterTest1 init...
[2025-01-17 11:43:53,654] Artifact FilterTest:war exploded: Artifact is deployed successfully
[2025-01-17 11:43:53,654] Artifact FilterTest:war exploded: Deploy took 294 milliseconds
Session 创建，当前在线人数：1
Session 创建，当前在线人数：2
Session 创建，当前在线人数：3
```

原因：

当使用 IntelliJ IDEA 启动 Tomcat 时，IDEA 会自动连接到 Tomcat 并访问项目，这会创建一个 Session。

如果你在 IDEA 中配置了多个部署目标（例如多个浏览器或工具），IDEA 可能会多次访问项目，导致多个 Session 被创建。

解决方法：

取消 IDEA 的 `After launch` 选项的勾选，避免 IDEA 自动连接到项目。

操作步骤：

1. 打开 IDEA 的 `Run/Debug Configurations`。
2. 找到你的 Tomcat 配置。
3. 取消勾选 `After launch` 选项。
4. 保存并重新启动 Tomcat。

```
Connected to server
[2025-01-17 11:47:24,796] Artifact FilterTest:war exploded: Artifact is being deployed, please wait...
17-Jan-2025 23:47:24.936 警告 [RMI TCP Connection(2)-127.0.0.1] org.apache.tomcat.util.descriptor.web.WebXml.setVersi
FilterTest1 init...
[2025-01-17 11:47:25,095] Artifact FilterTest:war exploded: Artifact is deployed successfully
[2025-01-17 11:47:25,095] Artifact FilterTest:war exploded: Deploy took 299 milliseconds
Session 创建，当前在线人数：1
```

附录：源码地址

https://github.com/Echo-Nie/JSPLearn