# 编程实战（上）

## 通讯录程序

需求：

添加联系人

修改联系人

删除联系人

查询所有联系人

## 职业发展：

技术线：初级开发攻城狮 - > 高级开发攻城狮 -> 核心开发人员(系统分析师助理) -> 系统分析师

非技术线：初级开发攻城狮 -> 高级开发攻城狮 -> 项目组长（项目经理助理）-> 项目经理

### 1.1 需求分析（需求分析师）

a．功能需求：

需求：

添加联系人

修改联系人

删除联系人

查询所有联系人

b．功能流程

抽取业务模型： 图书模型： 名称，页码，出版社....

功能流程图

### 1.2 需求设计（系统分析师）

a．美工设计（设计静态原型）

b．系统设计

1．设计实体

class Contact{

String id;

String name;

String gender;

String phone;

String email;

String address;

}

2．设计“数据库”

xml文件代替“数据库”

设计contact.xml

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <contact-list>  <contact id="001">  <name>张三</name>  <gender>男</gender>  <phone>13411112222</phone>  <email>zhangsan@qq.com</email>  <address>广州天河</address>  </contact>  </contact-list> |

3．设计接口

面向接口编程：

联系人DAO类：集中编写联系人实体对象的CURD方法。（数据访问对象）

规则： 一个实体对象对应一个dao接口和类

interface ContactDao{

public void addContact(Contact contact);

public void updateContact(Contact contact); 包含修改的id

public void deleteContact(String id);

public List<Contact> findAll();

public Contact findById(String id); 根据id查询对应的联系人

}

4．设计系统目录（包）规范

|-contactSys\_web 根目录

|-src

|-gz.itast.contactSys\_web.entity 存放实体对象类

|- gz.itast.contactSys\_web.dao 存放dao接口和类

|- gz.itast.contactSys\_web.web 存放servlet类

|- gz.itast.contactSys\_web.util 存放工具类

|- gz.itast.contactSys\_web.exception 自定义异常类

|-images 存放图片

|-css 存放css文件

|-js 存放js文件

### 1.3 编程实现（开发攻城狮）

1．搭建项目环境

a．项目建立

b．建立目录结构

c．导入第三方的包

2．开发步骤：从底层往上层开发

a．“数据库" (设计xml)

b．实体对象（建立实体对象）

c． DAO接口和实现类

d．页面+servlet（跟用户直接相关的）

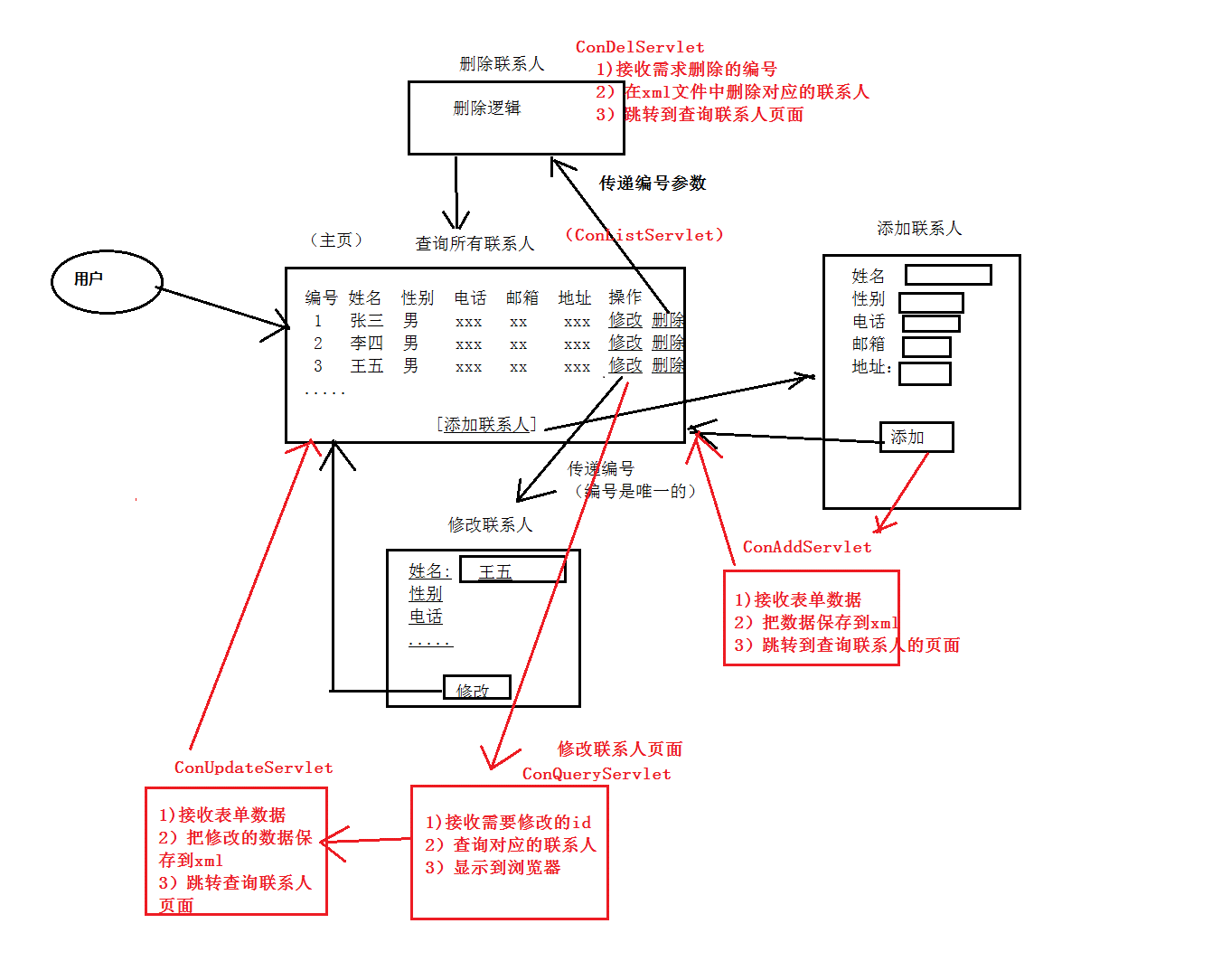
### 1.4 功能测试（测试攻城狮）

### 1.5 性能测试（测试攻城狮）

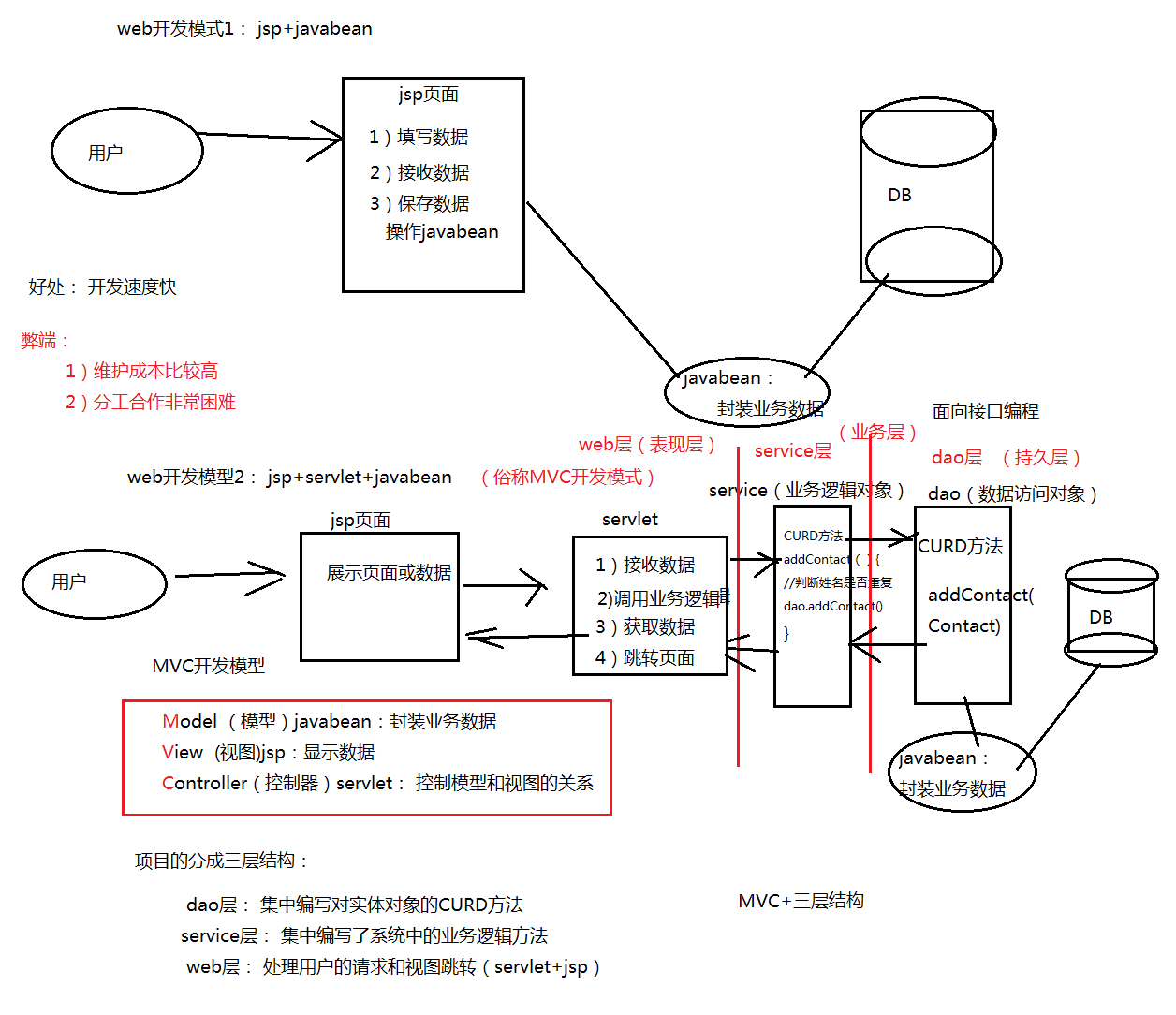
### 1.6 部署上线（实施攻城狮）

### 1.7 系统维护（实施攻城狮）

## 工程的功能流程图



## Web开发模型



### MVC开发模式：

M： javabean： 封装业务数据

V： jsp： 显示数据（jstl+el表达式）

C： servlet： 接收参数数据，调用业务逻辑，跳转视图

### 三层开发结构

dao层： 负责数据访问，实体对象的增删改成

service层： 负责处理系统的业务逻辑

web层： servlet+jsp完成用户交互

# Jsp基础

## 简介

Jsp,全称 Java Server Page java服务页面，能提供java服务的页面

jsp vs html

html: 由html标签组成的，输出静态内容。

jsp: 可以写html标签，还可以java代码。

## 特点

1．既可以写html代码，也可以写java代码

2．jsp页面需要交给tomcat服务器运行！！！

## Jsp运行的目录

tomcat的work目录，这个目录下存放jsp页面运行过程中产生的临时文件。

## 体验jsp页面如何编写和运行

向浏览器输出当前的系统时间

如何访问jsp页面？

直接通过浏览器访问即可！！（把当前html页面访问）

## Jsp页面如何被tomcat服务器运行

**第一次访问jsp：**

1．tomcat服务器把jsp页面**翻译**成java源文件。（01.hello.jsp -》 \_01\_hello\_jsp.java）

2．tomcat服务器把java源文件**编译**成class文件。\_01\_hello\_jsp.java -> \_01\_hello\_jsp.class

3．读取class文件，创建类对象。\_01\_hello\_jsp

4．执行类中的方法。\_01\_hello\_jsp类的方法

**第n次访问jsp：**

4．执行类中的方法。\_01\_hello\_jsp类的方法

直接走第4个步骤

**注意**

修改了jsp页面时，重新走翻译和编译的过程。

问题：运行的class类是什么类？

public final class \_01\_hello\_jsp extends org.apache.jasper.runtime.HttpJspBase

public abstract class org.apache.jasper.runtime.**HttpJspBase** extends javax.servlet.http.**HttpServlet** implements javax.servlet.jsp.HttpJspPage {

我们的jsp的源文件是一个servlet！

结论： jsp就是一个servlet！！！！

servlet的技术可以用在jsp上。

servlet的生命周期：

构造方法：

init方法：

service方法

destory方法

jsp的生命周期

翻译（java文件）

编译（class文件）

构造方法

\_jspInit方法

\_jspService方法

\_jspDestroy方法

# Jsp语法

## 2.1 Jsp模板

Jsp页面的html代码就是模块

## 2.2 Jsp表达式

语法： <%= 变量或表达式 %>

作用： 向浏览器输出变量的值或表达式技术的结构

注意：

1．其原理是使用out.print(name );方法向浏览器写出内容

2．结尾不能使用分号

## 2.3 Jsp脚本

语法： <% java语句 %>

作用： 执行java语句

原理： 脚本就是原封不动地翻译到java文件的\_jspServcice方法中执行。

注意：

1．其原理是直接把java脚本代码翻译到java文件的\_jspService方法中执行。

2．jsp脚本可以穿插html代码混合使用

3．jsp脚本不能声明方法

## 2.4 Jsp声明

语法：<%! 变量或方法 %>

作用：用于声明变量或方法

原理： 声明的变量是成员变量，声明的方法是成员方法

注意：

1．声明的是成员变量和成员方法。

2．方法不能在脚本中定义，而只能在声明中定义。

3．不能声明和翻译成的java文件中原有方法重复！！！

<%!

//声明方法

public void sayHello(){

System.out.print("xxxxx");

}

%>

## 2.5 Jsp注释

语法： <%-- jsp注释 --%>

作用： 注释jsp页面内容

注意：

1．html注释是会被翻译到java源文件中，也会被执行

2．jsp注释是不会翻译到java源文件中，不会被执行。

# Jsp的三大指令

**语法： <%@ 指令 %>**

**一般写在jsp的文件头**

## 3.1 taglib指令

1．导入jstl的支持jar包，在java5.0之后jstl已经是属于5.0的支持包里面。

2．使用jsp页面，需要在jsp的顶部使用taglib指令导入需要的标签库

<%@taglib uri="tld文件的路径" prefix="标签库缩写"%>

|  |
| --- |
| <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %> |

3．使用标签：

|  |
| --- |
| <c:set></c:set> |

## 3.2 include指令

作用：用于包含其他页面（可将静态页面包含进jspy页面作为固定的导航头）

|  |
| --- |
| <%--包含头部页面 --%>  <%@ include file=*"/common/header.jsp"*%> |

原理：

直接把包含与被包含页面的内容先合并在一起，然后翻译成一个java源文件，最后编译执行。（先合并再翻译）

（源码包含，也叫做静态包含）

## 3.3 page指令

**作用：告诉浏览器如何翻译jsp文件**

|  |  |
| --- | --- |
| <%@ page  **必须有的：**  language=*"java"* -- 使用什么语言来翻译jsp文件  import=*"java.util.\*"* -- 导包。导入多个使用逗号  **jsp文件的编码相关的：**  pageEncoding=*"utf-8"* *-- 使用什么编码翻译jsp文件*  contentType=*"text/html; charset=utf-8"*  *--服务器发送给浏览器的数据类型和内容编码格式*  **异常页面相关的：**  errorPage=*"/common/error.jsp"* *-- 指定jsp错误处理页面*  isErrorPage=*"true" --指定当前页面为错误处理页面，如果是错误处理页面，就可以使用exception内置对象，使用exception对象可以查询错误信息。*  **配置全局的错误处理页面，**在web.xml文件中配置：   |  | | --- | | <!-- 配置全局的错误处理页面 -->  <error-page>  <error-code>404</error-code>  <location>/common/404.html</location>  </error-page>  <error-page>  <error-code>500</error-code>  <location>/common/500.jsp</location>  </error-page> |   **功能设计相关的：**  session=*"true"* *-- 是否打开session功能，true：打开，false:关闭*  buffer=*"8kb"* *-- jsp页面的缓存区大小*  isELIgnored=*"false"* *-- 是否忽略EL表达式。true：忽略。false：不忽略。*  %> |

# Jsp的最佳实践

servlet技术： 使用java语言开发动态网页。功能：写java代码，输出html代码

jsp技术： 使用java语言开发动态网页。功能：写java代码，输出html代码

**分工问题：**

servlet技术： 写java代码（写普通java类）

jsp技术： 输出html代码（类似于html页面）

**项目中涉及的功能：**

1）接收参数信息 （servlet做）

2）执行业务逻辑（CURD） （servlet做）

3）返回结果，输出结果 （jsp做）

4）跳转页面 （servlet做）

**案例：**

显示用户信息

1．查询用户数据（servlet）

List<User> list

把用户数据保存到域对象，发送到jsp页面

2．显示用户数据（jsp）

# Jsp的内置对象

## 简介

在jsp开发有些对象使用频率比较高，（例如：requesst,respone,session，servletcontext…）如果每次要使用这些对象都自己去获取对象才能使用，会显示太麻烦了。jsp把这些对象都创建或获取好了，我们开发者之间使用这些对象即可！！！这些对象就叫做内置对象。

## 9大内置对象

jsp对象名 类型

request HttpServletRequest 请求对象

response HttpServletResponse 响应对象

config ServletConfig 配置对象

application ServletContext servlet上下文对象

在jsp中application可以写全局域，只要Tomcat不关闭就可用。

session HttpSession 会话对象

exception Throwable 异常信息对象

page Object 代表jsp文件翻译后的java类（等同this）

out JspWriter 代表输出对象，response.getWriter() PrintWriter

相当于带缓存功能的PrintWriter

pageContext PageContext 代表jsp的上下文对象

## out对象

out对象的类型JspWriter类，相当于带缓存功能的PrintWriter类

可以在page指令中设置其缓存大小。

如：<%@ page buffer=*"8kb"* %>

out对象，类型JspWriter

out.wirter("内容")

1．PrintWrite类： **直接往浏览器写出内容**

out.write（"内容"）;

2．JspWriter类： **相当于带缓存的PrintWriter，在jsp运行接收后写出。**

out.write("内容")： 把内容写入缓存区

**所以在同一个jsp中就算先out在用PrintWrite。只有out缓存没满，先出现的还是PrintWrite的内容。**

如：out.write("123");

response.getWriter().write("abc");

最后出现的是abc 123

### 注意事项：

当JspWriter缓存区满足以下条件，缓存区内容会写入到PrintWriter中。

1．缓冲区满了。

2．刷新缓冲区（JspWriet.flush()）

3．关闭缓冲区（<%@page buffer="0kb"%> buffer属性用于设置JspWriter缓存区大小，默认8KB）

4．执行完Jsp页面

**得到当前缓冲区大小： out.getBufferSize()**

**得到当前缓冲区剩余大小： out.getRemaining()**

## pageContext对象

### 作用：

### 1．通过pageContext对象获取其他8个内置对象

场景：使用自定义标签的时候

Jsp文件翻译的java类

public class \_02\_out\_jsp{

public void \_jspService(HttpServletRequest rquest, HttpServetResponse response){

HttpSession session;

ServletConfig config;

......

//Sun设计了一个PageContext对象，用于存放其他8个内置对象

PageContext pageContext;

//调用method1

method1(pageContext)

}

public void method1(PageContext pageContext){

//要使用8个内置对象

pageContext.getSession();

pageContext.getRequest();

pageContext.getResponse();

......

}

}

以上情况，在自定义标签中频繁使用到！！！

### 2．作为域对象使用

#### 2.1 保存到page域

**pageContet.setAttribute("name",Object);**

#### 2.2 保存到其他域

**pageContext.setAttribute("name,Object,PageContext.PAGE\_SCOPE")**

PAGE\_SCOPE //page域

REQUEST\_SCOPE //request域

SESSION\_SCOPE //session域

APPLICATION\_SCOPE //applivation域

#### 2.3 获取page域

**pageContext.getAttribute("name")**

#### 2.4 从其他域中获取

**pageContext.getAttribute("name,PageContext.PAGE\_SCOPE")**

PAGE\_SCOPE //page域

REQUEST\_SCOPE //request域

SESSION\_SCOPE //session域

APPLICATION\_SCOPE //applivation域

#### 2.5 自动搜索四个域数据

**pageContext.findAttribute("name")**

# jsp的四个域对象

page域： PageContext

request域： HttpServletREquest

session域： HttpSession

application域： ServletContext

## 域对象的作用

共享数据，保存数据，获取数据

## 域对象的方法

setAttribute(name,object) 保存数据方法

getAttribute(name) 获取数据

romveAttribute(name) 清除数据

## 域的作用范围

从小到大：

page域： 处于同一个jsp页面中数据共享是有效的！！

request域： 处于同一个请求中数据共享是有效的！！（使用转发）

session域： 处于同一个会话中数据共享是有效的！！（同一个session对象）

application域： 处于同一个web应用中数据共享是有效的！！

**注意：**

1．在哪个域中保存，就必须从哪个域中获取

## 实例：

<%

//session.setAttribute("name","eric");

//作用一： 可以获取其他8个内置对象

out.write("相等吗？"+(pageContext.getRequest()==request)+"<br/>");

out.write("相等吗?"+(pageContext.getResponse()==response));

out.write("<br/>");

//作用二: 作为域对象

//1．可以保存到page域中

//保存数据(默认情况下，保存到page域)

pageContext.setAttribute("message","page's message");

//2．保存到其他三个域中(request,session,application)

pageContext.setAttribute("message","request's message",PageContext.REQUEST\_SCOPE);

//等价于： request.setAttribute("message","request's message");

pageContext.setAttribute("message","session's message",PageContext.SESSION\_SCOPE);

pageContext.setAttribute("message","application's message",PageContext.APPLICATION\_SCOPE);

/\*\*

注意：

1．在哪个域中保存，就必须从哪个域中获取

\*/

//获取数据

//1．从page域中取出

/\*\*

String message = (String)pageContext.getAttribute("message");

out.write("pages' message="+message+"<br/>");

//2．从其他域中取出

message = (String)pageContext.getAttribute("message",PageContext.REQUEST\_SCOPE);

out.write("request's message="+message+"<br/>");

message = (String)pageContext.getAttribute("message",PageContext.SESSION\_SCOPE);

out.write("session's message="+message+"<br/>");

message = (String)pageContext.getAttribute("message",PageContext.APPLICATION\_SCOPE);

out.write("application's message="+message+"<br/>");

\*/

//3．自动从四个域中搜索数据

//搜索顺序： 从小到大： page->request->session->application

out.write("message="+pageContext.findAttribute("message"));

%>

# 总结：

jsp基础

1．jsp执行原理（翻译+编译+servlet）

2．jsp语法（jsp表达式+jsp脚本+jsp声明）

3．jsp的三大指令（include指令+page指令）

4．jsp的9个内置对象（out对象+pageContext对象）

5．jsp的最佳实践（servlet+jsp）

回顾重点内容：

Jsp基础

1）Jsp执行原理：

a) 翻译：tomcat服务器会把jsp文件翻译成java源文件

b）编译：tomcat服务器编译这个java文件，变为class文件

c）构造方法：tomcat服务器构造类对象

d）调用方法： tomcat服务器调用\_jspService()方法

2）Jsp的9大内置对象

request HttpServletRequest

response HttpServletResponse

config ServletConfig

application ServletContext

session HttpSession

exception Throwable

page Object

out JspWriter

pageContext PageContext

pageContext对象：jsp的上下文对象

作用：

a）可以获取其他8个内置对象

b）作为域对象使用

保存到page域： pageContext.setAttribute("name",Object);

保存到其他域中： pageContext.setAttribute("name“，Object, PageContext.REQUEST\_SCOPE)

从page域中获取： pageContext.getAttribute("name");

从其他域中获取： pageContext.getAttribute("name",PageContext.REQUEST\_SCOPE);

自动从四个域中搜索： pageContext.findAttribute("name")

顺序： page域 -》 request域 -> session域 -》 application域 -》 null

域对象的作用范围：

PageContext：page域： 在当前jsp页面中数据有效的！

HttpServletRequest：request域： 当前同一个请求中数据有效的！

HttpSession： session域： 当前同一个session（会话）中数据有效的！

ServletContext：application域： 当前同一个网站中数据有效的！

# EL表达式

## 简介

EL表达式替代jsp表达式。因为开发jsp页面的时候遵守原则：在jsp页面中尽量少些甚至不写java代码。

EL表达式作用：向浏览器输出域对象中的变量或表达式计算的结果

基本语法： **${变量或表达式} 代替<%=变量或表达式%>**

### 例：

1．EL从四个域中自动搜素

**${name} 等价于： <%=pageContext.findAttribute("name")%>**

2．EL从指定域中获取

**指定域获取的EL： ${pageScope.name }**

pageScope: page域

requestScope: request域

sessionScope: session域

applicationScope: application域

## EL获取不同类型的数据

普通字符串

普通的对象

数组或List集合

Map集合

### 例：

1．普通的字符串

<%

String email = "zhangsan@qq.com";

//一定要把数据放入域中

pageContext.setAttribute("email",email);

%>

普通字符串： ${email }

2．普通的对象

EL表达式的属性表示调用对象的getXX方法。

例如：student.name 调用了Stduent对象的getName()方法

<%

Student student = new Student("eric","123456");

pageContext.setAttribute("student",student);

%>

普通的对象： **${student}** <br/>

对象的属性： **${student.name }** - **${student.password }**

3．数组或List集合

<%

//数组

Student[] stus = new Student[3];

stus[0] = new Student("jacky","123456");

stus[1] = new Student("rose","123456");

stus[2] = new Student("lucy","123456");

pageContext.setAttribute("stus",stus);

//List

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

list.add(new Student("eric","123456"));

list.add(new Student("lily","123456"));

list.add(new Student("maxwell","123456"));

pageContext.setAttribute("list",list);

%>

**数组: <br/>**

**${stus[0].name1 } - ${stus[0].password }**<br/>

${stus[1].name1 } - ${stus[1].password }<br/>

${stus[2].name1 } - ${stus[2].password }<br/>

**List集合<br/>**

**${list[0].name1 } - ${list[0].password }**<br/>

${list[1].name1 } - ${list[1].password }<br/>

${list[2].name1 } - ${list[2].password }<br/>

4．Map集合

<%

Map<String,Student> map = new HashMap<String,Student>();

map.put("001",new Student("eric","123456"));

map.put("002",new Student("jacky","123456"));

map.put("003",new Student("rose","123456"));

pageContext.setAttribute("map",map);

%>

**Map集合： <br/>**

**${map['001'].name1 } - ${map['001'].password }**<br/>

${map['002'].name1 } - ${map['002'].password }<br/>

${map['003'].name1 } - ${map['003'].password }<br/>

## EL执行表达式

算术表达式

比较表达式

逻辑表达式

判空表达式 **empty**

### 例：

<%

int a = 10;

int b = 5;

pageContext.setAttribute("a",a);

pageContext.setAttribute("b",b);

%>

1．算术表达式： + - \* /

${a+b }

${a-b }

${a\*b }

${a/b }

2．比较表达式: > < >= <= ==

${a>b }

${a==b }

3．逻辑表达式： && （与） || （或） ！（非）

${true&&true }

${true||false }

${!true }

${!(a>b) }

4．判空表达式：**empty**

null: ${name1==null }<br/>

空字符串： ${name1=="" }<br/>

**null或空字符串： ${name1==null || name1=="" }**

<br/>

**null或空字符串：${empty name1}**

## EL的11个内置对象

pageContext jsp的pageContext内置对象

pageScope 从指定的域中获取数据

requestScope 从指定的域中获取数据

sessionScope 从指定的域中获取数据

applicatinoScope 从指定的域中获取数据

param 获取参数

paramValues 获取同名的多个参数

header 获取请求头

headerValues 获取同名的多个请求头

cookie 获取cookie

initParam 获取全局参数

### 例：

#### 1．pageContext：获取上下文路径

**${pageContext.request.contextPath }**

等价于 jsp的pageContext内置对象

#### 2．从指定域获取：

**${pageScope.name }**

pageScope: page域

requestScope: request域

sessionScope: session域

applicationScope: application域

#### 3．param： 获取参数

注意： **paramValues**返回所有参数Map集合

**${param['name']}**

等价于：<%=request.getParameter("name") %>

**${paramValues['name'][0] }**-**${paramValues['name'][1] }**

等价于：<%=request.getParameterValues("name")[1] %>

#### 4．header :获取请求头

注意： **headerValues**返回所有参数Map集合

**${header['host'] }**

等价于：<%=request.getHeader("host") %>

**${headerValues['host'][0] }**

等价于：<%=request.getHeaders("host").nextElement() %>

#### 5．cookie: 获取cookie

**${cookie['JSESSIONID'].name } - ${cookie['JSESSIONID'].value }**

等价于：<%=request.getCookies()[0].getValue() %>

#### 6．initParam: 获取全局参数

**${initParam['AAA'] }**

等价于：<%=application.getInitParameter("AAA") %><br/>

# Jsp标签

## 简介

EL表达式可以替换jsp表达式，但是EL表达式局限： 不能条件判断，不能赋值，不能迭代。jsp标签替代jsp脚本，完成条件判断，赋值，迭代等等功能。

## Jsp标签分类

1．jsp内置标签（动作标签）： 使用的时候不需要导入标签库

2．jstl标签（标准标签）： 使用的时候都需要导入标签库

3．自定义标签：使用的时候都需要导入标签库

## 动作标签

<jsp:forward/> 转发标签

<jsp:param/> 参数标签

<jsp:include/> 包含标签

注意：

1．其原理是包含与被包含的页面先各自翻译成独立的java源文件，然后再运行时合并再一起。（先翻译再合并），这叫动态包含

**==== 静态包含 vs 动态包含====**

1．原理不一样

a．静态包含（先合并再翻译）

b．动态包含（先翻译再合并）

2．语法不一样

a．静态包含（ includ指令： <%@include%> ）

b．动态包含 （include标签： <jsp:include />）

3．参数传递不同

a．静态包含不能传递参数

b．动态包含可以向被包含页面传递参数

### 例：

#### 1．<jsp：foward/> 转发

<jsp:forward page="/05.action2.jsp">

</jsp:forward>

等价于： <% request.getRequestDispatcher("/05.action2.jsp ").forward(request,response); %>

#### 2．<jsp:param/> 参数

<jsp:forward page="/05.action2.jsp">

<jsp:param value="eric" name="name"/>

</jsp:forward>

等价于：<% request.getRequestDispatcher("/05.action2.jsp?name=eric").forward(request,response); %>

#### 3． <jsp:include/> 动态包含。

**可以携带参数给被包含的页面，但是静态包含不能携带参数给被包含的页面**

<jsp:include page="/common/header.jsp">

<jsp:param value="eric" name="name"/>

</jsp:include>

**动态包含**

## JSTL标签

jstl标签，java standard tag libarary java标准标签库。

特点： 在使用jstl标签时需要导入标签库

jstl标签库分类：

核心标签库（重要）（c）

国际化标签库(fmt)

EL函数库(fn)

SQL标签库(sql)

XML标签库(x)

## 使用JSTL标签

1．导入jstl的支持jar包，在java5.0之后jstl已经是属于5.0的支持包里面。

2．使用jsp页面，需要在jsp的顶部使用taglib指令导入需要的标签库

<%@taglib uri="tld文件的路径" prefix="标签库缩写"%>

|  |
| --- |
| <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %> |

3．使用标签：

|  |
| --- |
| <c:set></c:set> |

## 核心标签库的标签

保存数据：

<c:set></c:set>

获取数据

<c:out value=*""*></c:out>

单条件判断：

<c:if test=*""*></c:if>

多条件判断

<c:choose></c:choose>

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

<c:otherwise></c:otherwise>

数据遍历

<c:forEach></c:forEach>

<c:forTokens items=*""* delims=*""*></c:forTokens>

重定向

<c:redirect></c:redirect>

### 实例：

#### 1．<c:set/> 给域对象赋值

<c:set var="name" value="jacky" scope="request"/>

var: 数据的名称

value： 保存的数据值

scope: 保存到哪个域。

page -> page域

request -> request域

session -> session域

application -> application域

#### 2．<c:out/> 获取域对象的数据

<c:out value="${name}" default="<h3>标题3</h3>" escapeXml="false"></c:out>

value: 代表获取域中的某个名称内容。如果数据在域中，必须使用EL语法去获取

default: 默认值。当前需要获取的内容为null，那么使用默认值代替

escapeXml: 默认情况下为true，out标签会把输入的内容进行转义。

如果不需要转义，设为false既可！

#### 3．<c:if/> 单条件判断

<c:if test="${10>8}">

<input type="submit" value="提交"/>

</c:if>

test: 条件表达式。返回true的时候，就会执行if标签体内容，否则，不执行。

通常用于判断是否符合要求点下一步。

#### 4．<c:choose/>+<c:when/>+<c:otherwise/> 多条件判断

<%--模拟登录 --%>

<c:set var="user" value="eric" scope="session"></c:set>

登录之后，一定会把数据存放到session域，user名称

<c:choose>

<c:when test="${!empty sessionScope.user}">

欢迎回来，你的用户名是 ：eric，<a href="">【退出登录】</a>

</c:when>

<c:otherwise>

请先<a href="">注册</a>或<a href="">登录</a>

</c:otherwise>

</c:choose>

#### 5．<c:forEach/> 用于迭代或循环

List集合：

<c:forEach items="${list}" var="student" varStatus="varSta">

序号：${varSta.count } 姓名：${student.name1 } - 密码： ${student.password }<br/>

</c:forEach>

begin: 从哪个元素开始遍历，默认从0开始

end: 到哪个元素位置，默认遍历到结束

step: 增加步长。默认step为 1

items: 需要遍历的数据。（数组|List集合|Map集合） 如果是获取域数据，使用EL表达式获取

var: 每个元素名称

varStatus: 当前状态对象。该对象封装当前元素状态信息。（用于表示遍历次数）

例如：count属性: 表示当前遍历的是哪个元素，从1开始

Map集合：

<c:forEach items="${map}" var="entry">

编号： ${entry.key } - 姓名:${entry.value.name1 } - 密码： ${entry.value.password }<br/>

</c:forEach>

注意：

forEach标签遍历Map集合时，把每个Map的对象使用Entry封装，

Entry封装键对象和值对象，通过getKey()获取键对象，通过getValue()获取值对象

在JSTL标签中使用时为.key和.value

#### 6．<c:forToekens/> 遍历特殊字符串（可以自动切割字符串）

<c:forTokens items="${str}" delims="-" var="s">

${s },

</c:forTokens>

items: 需要遍历的字符串

delims: 指定分割符号

var: 每个内容的名称

等价于：

<%

String str= (String)pageContext.getAttribute("str");

String[] strs = str.split("-");

for(int i=0;i<strs.length;i++){

out.write(strs[i]+",");

}

%>

#### 7．<c:rediect/> 重定向标签

<c:redirect url="/03.el3.jsp"></c:redirect>

等价于：

<%

response.sendRedirect(request.getContextPath()+"/03.el3.jsp");

%>

# 自定义标签

当现有的标签无法满足需求的时候，就需要开发者自行开发标签。

## 自定义标签开发步骤

### 1．编写一个标签处理器程序，继承SimpleTagSupport类

|  |
| --- |
| /\*\*  \* 标签处理器类  \* **@author** APPle  \*  \*/  **public** **class** ShowIpTag **extends** SimpleTagSupport{    /\*\*  \* 执行标签  \*/  @Override  **public** **void** doTag() **throws** JspException, IOException {  //得到request对象  PageContext pageContext = (PageContext)**this**.getJspContext();  HttpServletRequest request = (HttpServletRequest)pageContext.getRequest();    //得到客户ip  String ip = request.getRemoteHost();    //写出浏览器  pageContext.getOut().write("自定义标签实现获取客户端IP地址："+ip);  }  } |

### 2．在WEB-INT目录下建立tld文件，例如itcast.tld,内容如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <taglib 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-jsptaglibrary\_2\_1.xsd"*  version=*"2.1"*>    <!-- 标签库版本 -->  <tlib-version>1.1</tlib-version>  <!-- 标签库的缩写：简单名称 -->  <short-name>itcast</short-name>  <!-- tld文件的唯一标记 -->  <uri>http://gz.itcast.cn</uri>  <!--标签定义 -->  <tag>  <!-- 标签名称 -->  <name>showIp</name>  <!-- 标签对应的处理器程序 -->  <tag-class>gz.itcast.c\_tag.ShowIpTag</tag-class>  <!-- 标签体输出格式 -->  <body-content>scriptless</body-content>  </tag>  </taglib> |

### 3．在jsp页面中导入自定义标签库，再使用标签库中的标签

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"utf-8"*%>  <%@taglib uri=*"http://gz.itcast.cn"* prefix=*"itcast"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>第一个自定义标签</title>  </head>    <body>  </body>  </html> |

### 4．使用标签库中的标签

|  |
| --- |
| <%  //得到客户IP地址  //String ip = request.getRemoteHost();  //out.write(&quot;客户的IP地址为： &quot;+ip);  %>  <itcast:showIp></itcast:showIp> |

## 自定义标签的执行过程

前提： tomcat服务器在加载day12web应用时，读取web.xml和tld文件。

浏览器访问：http://localhost:8080/day12/07.tag.jsp

1．翻译成java文件 -> 编译成class文件 -> 构造方法 -> 调用\_jspService()方法

2．在内存tld文件中，查询是否存在一个uri名称为http://gz.itcast.cn的tld文件

3．得到对应的tld文件内容。

4．读到<itcast:showIp>内容时，截取showIp名称（就是标签）,到tld文件中查询是否存在name为showIp的tag标签。

5．取出<tag>标签的内容<tag-class>字符串： gz.itcast.a\_tag.ShowIpTag

## 自定义标签的生命周期

由tocmat服务器调用的

1．构造方法： 构造标签器对象

2．void setJspContext(JspContext pc)

设置pageContext上下文件对象。传入当前jsp页面的PageContext对象。

在标签处理类中通过getJspContext（）方法得到PageContex对象

3．void setParent(JspTag parent)

设置父标签对象。传入父标签对象；如果没有父标签则不调用此方法。

通过getParent（）方法得到父标签对象。

4．void setJspBody(JspFragment jspBody)

设置标签体内容。把标签体内容封装到JspFragment对象中然后通过此方法传入。

通过getJspBody()方法得到标签体内容。如果没有标签体则不调用次方法。

5．void doTag()

调用标签时执行方法。我们的业务逻辑就写在这个方法中。

可以调用getJspContext()、 getParent()、getJspBody();

## 查询源代码了解标签的执行过程

|  |
| --- |
| private boolean \_jspx\_meth\_itcast\_005fdemo1\_005f0(javax.servlet.jsp.tagext.JspTag \_jspx\_parent, PageContext \_jspx\_page\_context)  throws Throwable {  PageContext pageContext = \_jspx\_page\_context;  JspWriter out = \_jspx\_page\_context.getOut();  // itcast:demo1  1．实例化DemoTag1对象  gz.itcast.a\_tag.DemoTag1 \_jspx\_th\_itcast\_005fdemo1\_005f0 = new gz.itcast.a\_tag.DemoTag1();  org.apache.jasper.runtime.AnnotationHelper.postConstruct(\_jsp\_annotationprocessor, \_jspx\_th\_itcast\_005fdemo1\_005f0);  2．调用setJspContext方法，传入pageContext  \_jspx\_th\_itcast\_005fdemo1\_005f0**.setJspContext**(\_jspx\_page\_context);  3．调用setParent方法，传入父标签对象（DemoTag2）  \_jspx\_th\_itcast\_005fdemo1\_005f0**.setParent**(\_jspx\_parent);  4．调用setJspBody，传入标签体内容（JspFragment）  \_jspx\_th\_itcast\_005fdemo1\_005f0**.setJspBody**(new Helper( 1, \_jspx\_page\_context, \_jspx\_th\_itcast\_005fdemo1\_005f0, null));  5．调用doTag，执行标签  \_jspx\_th\_itcast\_005fdemo1\_005f0**.doTag**();  org.apache.jasper.runtime.AnnotationHelper.preDestroy(\_jsp\_annotationprocessor, \_jspx\_th\_itcast\_005fdemo1\_005f0);  return false;  } |

## 生命周期流程图



## 自定义标签作用

**1．控制标签体内容是否输出**

输出：调用jspBody.invokie()方法

不输出： 什么都不做！

**2．控制标签后面的内容是否输出**

执行：

什么都不做！！！

不执行：

抛出SkipPageException异常即可！

**3．控制重复出现执行标签体内容**

|  |
| --- |
| **for**(**int** i=1;i<=5;i++){  jspBody.invoke(**null**);//默认null，也是输出到浏览器  } |

**4．改变标签体内容**

|  |
| --- |
| //4.1 得到标签体内容  //4.1.1 把标签体内容拷贝到Writer流中  StringWriter sw = **new** StringWriter();  jspBody.invoke(sw);  //4.1.2 从Writr流中得到标签体内容  String content = sw.toString();  //4.2 修改标签体内容  content = content.toLowerCase();  //4.3 输出到浏览器  //jspBody.invoke(null);注意：不能用再用此方法，因为输出还是原来的内容  **this**.getJspContext().getOut().write(content); |

## 带属性的自定义标签

**1．在标签处理程序中，添加一个setXXX方法，用于给属性赋值。**

|  |
| --- |
| **public** **class** DemoTag1 **extends** SimpleTagSupport{  //1）声明一个成员变量用于接收标签属性值  **private** Integer num;  //2）用于把外面标签的属性值传入进来  **public** **void** setNum(Integer num) {  **this**.num = num;  } |

**2．在tld文件的所在标签中进行配置**

|  |
| --- |
| <tag>  <name>hello</name>  <tag-class>gz.itcast.a\_tag.HelloTag</tag-class>  <body-content>scriptless</body-content>  <!-- 配置属性 -->  <attribute>  <!-- 属性名称 -->  <name>num</name>  <!-- 属性是否必须的: true:一定要 false:可选的 -->  <required>true</required>  <!-- 该属性是否可以写EL表达式的内容  true: 可以写EL  false: 不能写EL  -->  <rtexprvalue>false</rtexprvalue>  </attribute>  </tag> |

## 标签体内容的输出格式

|  |
| --- |
| <body-content>scriptless</body-content> |

JSP： 表示标签体内容可以支持和输出jsp表达式内容。（注意：这个配置只能在传统标签中使用）

scriptless: 不能写jsp脚本或表达式，且不能执行！（可以是EL）。

empty: 标签体内容是空的。该标签是空标签

tagdependent: 标签体内容支持jsp表达式，按照脚本的原样输出（EL也是原样输出）

## 案例 - 登录页面的标签

### 1．标签处理程序

/\*\*

\* 登录页面的标签

\* @author APPle

\*/

public class LoginTag extends SimpleTagSupport{

private String name;//用户名表单的name属性

private String password;//密码输入框的name属性

public void setName(String name) {

this.name = name;

}

public void setPassword(String password) {

this.password = password;

}

@Override

public void doTag() throws JspException, IOException {

String html = "";

html += "<table align='center' border='1' width='300px'>";

html += "<caption><h3>用户登录页面</h3></caption>";

html += "<tr><th>用户名</th><td><input type='text' name='"+name+"'/></td></tr>";

html += "<tr><th>密码</th><td><input type='password' name='"+password+"'/></td></tr>";

html += "<tr><td colspan='2' align='center'><input type='submit' value='登录'/><input type='reset'/></td></tr>";

html += "</table>";

//把html代码输入到浏览器

this.getJspContext().getOut().write(html);

}

}

### 2．在标签库中设置login标签

<!-- login标签 -->

<tag>

<name>login</name>

<tag-class>gz.itcast.a\_tag.LoginTag</tag-class>

<body-content>scriptless</body-content>

<attribute>

<name>name</name>

<required>true</required>

<rtexprvalue>false</rtexprvalue>

</attribute>

<attribute>

<name>password</name>

<required>true</required>

<rtexprvalue>false</rtexprvalue>

</attribute>

</tag>

### 3．使用login标签

<body>

<itcast:login password="password" name="name"></itcast:login>

</body>

# JavaBean规范

## 简介

JavaBean就是一个普通的java类，必须遵守以下规则的java类才能称为javabean。

1．必须有一个无参的构造方法

2．把类的属性私有化。（不能直接访问属性） private

3．必须提供公开的getter和setter方法 。 （通过getter和setter操作属性）

应用场景：

1．项目的实体对象符合javabean的规范

2．EL表达式，访问对象的属性。${student.name} 原理：调用 getName();

3．自定义标签处理，给标签属性赋值。 setNum(int num)

4．JSP页面中可以使用javabean

JavaBean作用：

封装对象（数据）

设置数据: setXXX方法

获取数据： getXXX方法 / isXXX方法（boolean）

# 总结：

jsp加强：

1．EL表达式：(重点)

${变量或表达式}

2．Jsp标签

2.1 Jsp动作：

<jsp:forward/>

<jsp:param/>

<jsp:include/>

2.2 jstl的核心标签库（重点）

<c:set/>

<c:out/>

<c:if/>

<c:choose/>+<c:when/>+<c:otherwise/>

<c:forEach/>

<c:forTokens/>

<c:redirect/>

2.3 自定义标签

开发步骤： （使用）

标签处理类+tld文件+使用（taglib）

执行原理：(理解)

setJspContext()

setParent()

setJspBody()

doTag()

1．自定义标签：（应用）

自定义标签作用：

控制标签体内容是否输出

控制标签后面的内容是否输出

重复输出标签体内容

改变标签体内容

带属性的标签

2．编程实战：

MVC开发模式：

M： javabean： 封装业务数据

V： jsp： 显示数据（jstl+el表达式）

C： servlet： 接收参数数据，调用业务逻辑，跳转视图

三层开发结构

dao层： 负责数据访问，实体对象的增删改成

service层： 负责处理系统的业务逻辑

web层： servlet+jsp完成用户交互