JSP教程

极客学院出版

前言

JavaServer Pages(JSP) 是一种服务器端编程技术，能够为构建基于 Web 的应用程序创建动态的独立于平台的方法。JSP 可以访问 Java API 的整个家族，包括访问企业级数据库的 JDBC API。

适用人群

本教程帮助初学者们 Web 了解 JavaServer Pages(JSP) 的基本功能，从而开发自己的 Web 应用程序。学完本教程后你会发现自己处于一个使用 JSP 专业知识的中等水平，之后你可以达到更高的水平。

学习前提

我们假设你不知道 Web 应用程序如何在 HTTP 上工作，不知道什么是 Web 服务器，也不知道什么是 Web 浏览器。如果你稍微懂一点使用任何一种编程语言进行 Web 应用程序开发的知识，那就非常棒了。

更新日期	更新内容
2015-05-22	第一版发布

目录

1

JSP 基础教程

JSP – 概述

什么是 JavaServer Pages?

JavaServer Pages(JSP) 是一种技术，能够开发支持动态内容的网页，可以帮助开发人员在 HTML 页面中利用特殊的 JSP 标签插入 Java 代码，其中大部分标签是以 `<%` 开始，以 `%>` 结束的。

JavaServer Pages 组件是 Java Servlet 的一种，旨在为 Java web 应用程序实现一个用户界面。Web 开发人员编写 JSP 作为文本文件，结合 HTML 或 XHTML 代码，XML 元素，并嵌入 JSP 操作和命令。

使用 JSP，你可以通过 web 页面的形式收集来自用户的输入，来自数据库或其他资源的当前记录并动态地创建 web 页面。

JSP 标签可用于各种用途，如从数据库检索信息或注册用户首选项，访问 javabean 组件，在页面之间传递控制，在请求、页面之间共享信息等。

为什么使用 JSP?

JavaServer Pages 的服务通常与通用网关接口(CGI)实现程序一样。但 JSP 与 CGI 相比，有几个优势。

- 性能更好，因为 JSP 允许在 HTML 页面本身嵌入动态元素而不需要创建一个单独的 CGI 文件。
- JSP 总是在服务器处理之前进行编译，不像 CGI/Perl，每次请求页面时都需要服务器加载一个解释器和目标脚本。
- JavaServer Pages 是建立在 Java servlet API 之上的，就像 servlet，JSP 也可以访问所有强大的 Enterprise Java API，包括 JDBC，JNDI，EJB，JAXP 等等。
- JSP 页面可以与 servlet 结合使用，来处理业务逻辑，该模型是由 Java servlet 模板引擎支持的。

最后，JSP 还是 Java EE 不可分割的一部分，是 enterprise 级别应用程序的一个完整平台。这意味着 JSP 可以用于从最简单的应用程序到最复杂的应用程序中，并实现要求。

JSP 的优点

下列是 JSP 优于其他技术的另外的优点：

- **与 Active Server Pages(ASP) 相比：** JSP 的优点是双重的。首先，动态的部分是用 Java 编写的，而不是用 Visual Basic 或其他特定的语言编写，所以它使用起来更强大并且更容易。第二，它可以移植到其他操作系统和非 microsoft 的 Web 服务器中。
- **与 Pure Servlets 相比：** 与用大量的 println 语句生成 HTML 相比，JSP 能够更方便的写(和修改!)常规的 HTML。
- **与Server-Side Includes(SSI)相比：** SSI 只是用于简单的包含物，而不是用于使用表单数据、创建数据库链接等的“真正的”程序。
- **与JavaScript相比：** JavaScript 可以在客户端动态地生成 HTML，但很难与 web 服务器交互来执行复杂的任务，如数据库访问、图像处理等。
- **与Static HTML相比：** 当然，常规的 HTML 不能包含动态的信息。

后续内容

我将带你一步一步配置环境来开始 JSP 的学习。我假设你有良好的 Java 编程基础来进行 JSP 的学习。

如果你不知道 Java 编程语言，我建议你通过Java 教程 (http://wiki.jikexueyuan.com/project/java/) 来了解 Java 编程。

JSP – 环境配置

开发环境是你将开发 JSP 程序、测试程序并最终运行程序的地方。

本教程将指导你设置 JSP 开发环境，包括以下步骤：

设置 Java 开发工具包

这一步涉及到下载 Java 软件开发工具包(SDK)的实现并正确的设置 PATH 环境变量。

你可以从 Oracle 的 Java 站点下载 SDK：Java SE 下载 (http://www.oracle.com/technetwork/java/javase/downloads/index.html) 。

当你下载 Java 实现之后，按照给定的指示来安装和配置设置。最后设置路径和 JAVA_HOME 环境变量来引用目录，其中包含 java 和 javac，通常分别是 java_install_dir/bin 和 java_install_dir。

如果你运行的是 Windows，且 SDK 安装在 C:\ jdk1.5.0_20 中，将以下行添加到你的 C:\ autoexec.bat 文件中。

```
set PATH=C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME=C:\jdk1.5.0_20
```

另外，在 Windows NT / 2000 / XP 系统中，你也可以右键单击我的电脑，选择属性，然后单击高级系统设置，选择环境变量。之后，你可以更新路径值，然后按下 OK 按钮。

在 Unix(Solaris ，Linux 等等)系统中，如果 SDK 安装在 /usr/local/jdk1.5.0_20 中，你使用 C shell，将以下行添加到你 .cshrc 文件中。

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

另外，如果你使用一个集成开发环境(IDE)，如 Borland JBuilder、Eclipse、IntelliJ IDEA 或 Sun ONE Studio，就编译并运行一个简单的程序来确认 IDE 知道你在哪安装了 Java。

设置 Web 服务器：Tomcat

许多支持 JavaServer Pages 和 Servlets 开发的 Web 服务器都能在市场中找到。一些 web 服务器是免费下载的，Tomcat 就是其中之一。

Apache Tomcat 是 JSP 和 Servlet 技术的一个开源软件实现，可以作为一个独立的服务器测试 JSP 和 Servlet，并且可以与 Apache Web 服务器集成。下面是在计算机上安装 Tomcat 的步骤：

- 从 http://tomcat.apache.org/ 下载最新版本的 Tomcat。
- 下载安装后，解压二进制发行版到一个方便的位置。例如 windows 系统中的 C:\ apache-tomcat-5.5.29 位置，或在 Linux / Unix 系统中的 /usr/local/apache-tomcat-5.5.29 位置，然后创建 CATALINA_HOME 环境变量来指向这些位置。

在 Windows 系统的计算机中，可以通过执行以下命令来启动 Tomcat：

```
%CATALINA_HOME%\bin\startup.bat
```

或

```
C:\apache-tomcat-5.5.29\bin\startup.bat
```

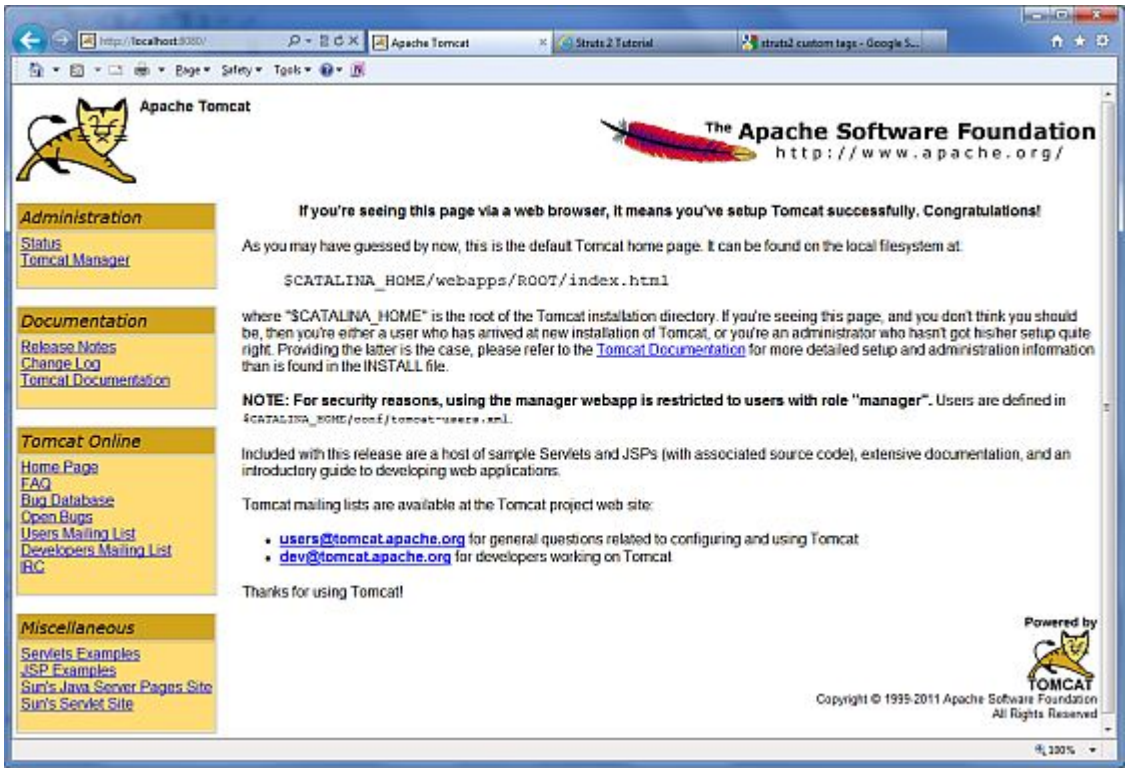
在 Unix(Solaris，Linux 等)系统的计算机中，可以通过以下命令来启动 Tomcat：

```
$CATALINA_HOME/bin/startup.sh
```

或

```
/usr/local/apache-tomcat-5.5.29/bin/startup.sh
```

启动成功后，通过访问 **http://localhost:8080 /**，包含 Tomcat 的默认的 web 应用程序将是可用的。如果一切进展顺利，那么应该显示如下结果：



图片 1.1 environment1

关于配置和运行 Tomcat 的更多信息可以在这里的文档中找到，也可以在 Tomcat 网站中找到：http://tomcat.apache.org

在 Windows 系统的计算机中，可以通过执行以下命令来结束 Tomcat：

```
%CATALINA_HOME%\bin\shutdown
```

或

```
C:\apache-tomcat-5.5.29\bin\shutdown
```

在 Unix(Solaris，Linux 等)系统的计算机中，可以通过以下命令来结束 Tomcat：

```
$CATALINA_HOME/bin/shutdown.sh
```

或

```
/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

设置类路径

由于 servlet 不是标准版 Java 平台的一部分，你必须为编译器识别 servlet 类。

如果你运行的是 Windows 系统，你需要把以下行添加到 C:\ autoexec.bat 文件中。

```
set CATALINA=C:\apache-tomcat-5.5.29
set CLASSPATH=%CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%
```

另外，在 Windows NT / 2000 / XP 系统中，你也可以右键单击我的电脑，选择属性，然后单击高级系统设置，选择环境变量。之后，你可以更新路径值，然后按下 OK 按钮。

在 Unix(Solaris ，Linux 等等)系统中，如果你使用的是 C shell，可以将以下行添加到你 .cshrc 文件中。

```
setenv CATALINA=/usr/local/apache-tomcat-5.5.29
setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH
```

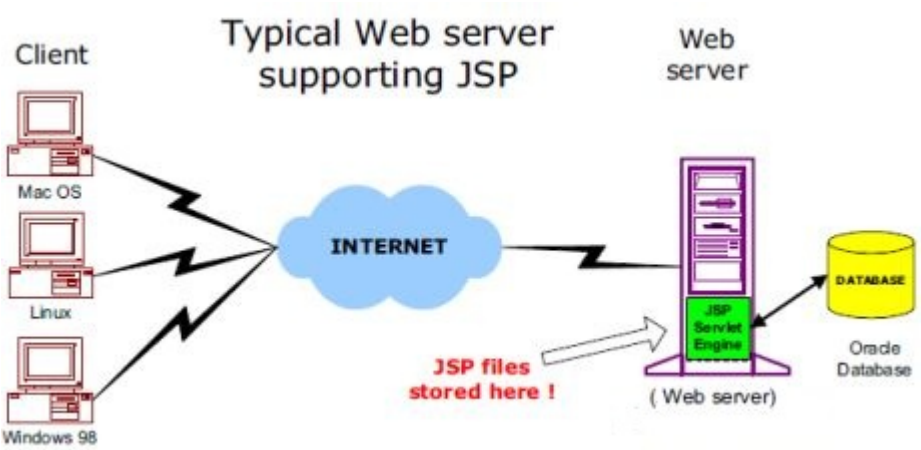
注意： 假设你的开发目录是 C:\ JSPDev(Windows 系统)或 /usr/JSPDev(Unix 系统)，那么你需要以上述类似的方式在类路径中添加这些目录。

JSP – 体系结构

Web 服务器需要一个 JSP 引擎，即处理 JSP 页面的容器。JSP 容器负责为 JSP 页面拦截请求。本教程使用了 Apache，Apache 已经内置了 JSP 容器来支持 JSP 页面开发。

JSP 容器适用于 Web 服务器，来提供 JSP 运行时环境和其他服务的需求。它知道如何理解 JSP 的部分特殊元素。

以下图表显示了 JSP 容器的位置以及在一个 Web 应用程序中的 JSP 文件。



图片 1.2 arch1

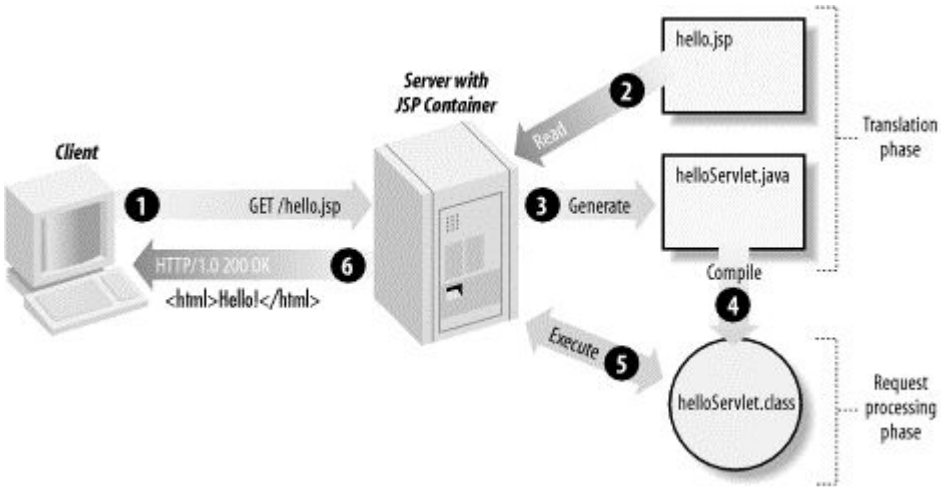
JSP 处理

下面的步骤解释了 web 服务器如何使用 JSP 创建 web 页面：

- 作为正常的页面，你的浏览器发送一个 HTTP 请求到 web 服务器。
- web 服务器承认一个 JSP 页面的 HTTP 请求，并将其转发给一个 JSP 引擎。这是通过使用 URL 或 JSP 页面实现的，该 JSP 页面是以.jsp结尾而不是以 .Html 结尾的。
- JSP 引擎从磁盘加载 JSP 页面并将其转换为一个 servlet 的内容。这种转换是非常简单的，所有模板文本转换为 println()语句，所有 JSP 元素转换为 Java 代码实现页面的相应的动态行为。
- JSP 引擎编译 servlet 到一个可执行的类中，并将原始请求转发给一个 servlet 引擎。
- 调用 servlet 引擎的 web 服务器的一部分加载 Servlet 类并执行它。执行期间，Servlet 产生一个 HTML 格式的输出，servlet 引擎将该输出传递到 HTTP 响应内的 web 服务器中。

- web 服务器将 HTTP 响应以静态 HTML 内容的形式转发到你的浏览器中。
- 最后 web 浏览器处理 HTTP 响应中的动态生成的 HTML 页面，就好像它是一个静态页面。

上述所有步骤如下图所示：



图片 1.3 architecture2

通常，JSP 引擎检查 JSP 文件的 servlet 是否已存在，JSP 的修改日期是否比 servlet 的过时。如果 JSP 的修改日期比其生成的 servlet 的修改日期过时，那么 JSP 容器假设 JSP 修改日期没有改变，且生成的 servlet 的修改日期仍然与 JSP 的内容相匹配。与其他脚本语言(比如 PHP)相比，这个过程更有效率，因此更加快速。

所以在某种程度上来说，一个 JSP 页面只是用另一种方式来写 servlet，而不需要成为一个 Java 编程奇才。除了翻译阶段，处理 JSP 页面完全就像处理一个普通的 servlet 一样。

JSP – 生命周期

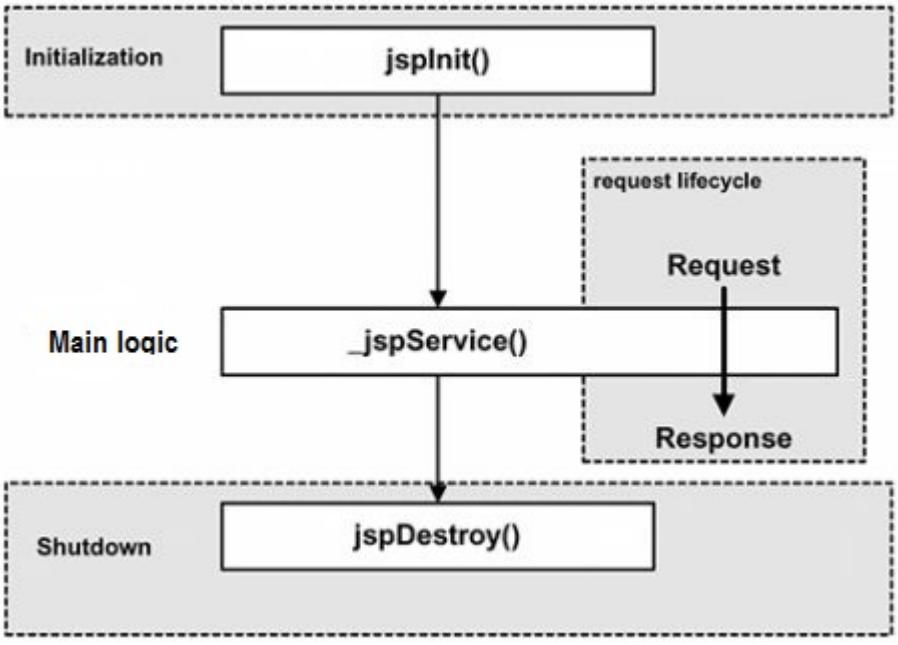
理解 JSP 的低级功能的关键是理解它们遵循的简单的生命周期。

JSP 生命周期可以被定义为从创建到销毁的整个过程，这类似于一个 servlet 的生命周期与一个额外的步骤，该步骤将一个 JSP 编译成 servlet。

以下是 JSP 带着的步骤

- 编译
- 初始化
- 执行
- 清理

JSP 生命周期的四个主要阶段非常类似于 Servlet 生命周期，它们如下所示：



图片 1.4 lifecyle1

JSP 编译

当浏览器请求一个 JSP，JSP 引擎首先检查是否需要编译页面。如果页面从未被编译，或者 JSP 自上次编译后被修改了，那么 JSP 引擎就会编译页面。

编译过程包括三个步骤：

- 解析 JSP。
- 将 JSP 转换为 servlet。
- 编译 servlet。

JSP 初始化

当容器加载 JSP时，在响应任何请求之前它会调用 jspInit()方法。如果你需要执行 JSP-specific 初始化，那么就覆盖 jspInit()方法：

```
public void jspInit(){
  // Initialization code...
}
```

通常初始化只执行一次，servlet init 方法也是只执行一次。一般初始化数据库连接，打开文件，并在 jspInit 方法中创建查找表。

JSP 执行

JSP 生命周期的这个阶段代表所有的交互请求，直到 JSP 被摧毁。

当浏览器请求一个 JSP 页面时并且该页面被加载并初始化，JSP 引擎就会在 JSP 中调用_jspService() 方法。

_jspService() 方法接受一个 HttpServletRequest 和一个 HttpServletResponse 作为其参数，如下所示：

```
void _jspService(HttpServletRequest request,
                 HttpServletResponse response)
{
  // Service handling code...
}
```

每次请求时 JSP 的 _jspService()方法都会被调用，且该方法负责生成请求的响应，并且该方法还负责生成所有七个 HTTP 方法的反应，即 GET、POST、DELETE 等。

JSP 清理

JSP 生命周期的破坏阶段代表 JSP 从容器中删除。

jspDestroy()方法是 JSP 的相当于 servlet 的销毁方法。当你需要执行任何清理时，覆盖 jspDestroy，比如释放数据库链接或关闭打开的文件。

jspDestroy()方法具有以下形式：

```
public void jspDestroy()
{
  // Your cleanup code goes here.
}
```

JSP – 语法

本教程将给出涉及 JS P开发的语法的基本思想(即元素)。

Scriptlet

scriptlet 可以包含任意数量的 JAVA 语言语句，变量或方法声明，或者在页面的脚本语言中有效的表达式。

下面是 scriptlet 的语法：

```
<% code fragment %>
```

你可以编写相当于上述语法的 XML，如下所示：

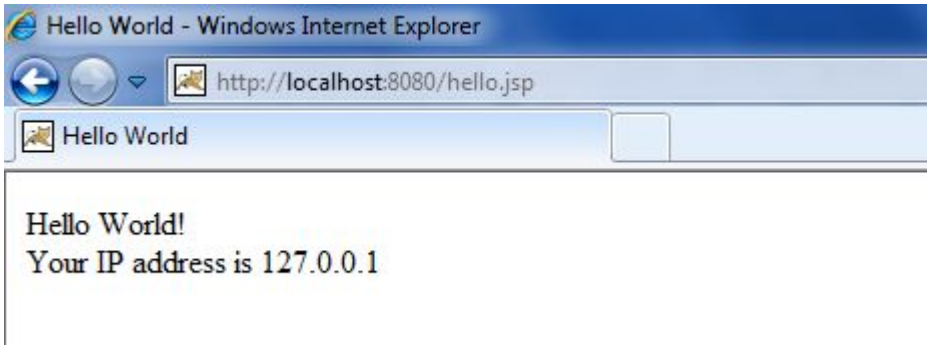
```
<jsp:scriptlet>
  code fragment
</jsp:scriptlet>
```

任何文本、HTML 标签或你编写的 JSP 元素必须在 scriptlet 之外。下面是 JSP 的第一个简单的例子：

```
<html>
<head><title>Hello World</title></head>
<body>
Hello World!<br/>
<%
out.println("Your IP address is " + request.getRemoteAddr());
%>
</body>
</html>
```

注： 假设 Apache Tomca t安装在 C:\ apache-tomcat-7.0.2 中，并且你的环境是按照教程中的环境设置配置的。

让我们把上面的代码写入 JSP 文件中的 hello.jsp 中，并把这个文件放在 C:\ apache-tomcat-7.0.2 \ webapps \ROOT 中，并试图通过 http://localhost:8080 / hello.jsp 浏览它。这将产生以下结果：



图片 1.5 syntax1

JSP 声明

JSP 声明声明了一个或多个变量或方法，你可以在 JSP 文件中的 Java 代码中使用。当你在 JSP 文件中使用变量或方法之前，你必须声明。

下面是 JSP 声明的语法：

```
<%! declaration; [ declaration; ]+ ... %>
```

你可以编写相当于上述语法的 XML，如下所示：

```
<jsp:declaration>
  code fragment
</jsp:declaration>
```

以下是 JSP 声明的一个简单的例子：

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

JSP 表达式

JSP 表达式元素包含一个脚本语言表达式，该表达式被赋值，转换成一个字符串，并插入到表达式出现在 JSP 文件中的位置。

因为表达式的值转换为一个字符串，你可以在 JSP 文件中的文本的一行使用一个表达式，无论该文本是否是 HTML 标签标记。

根据 Java 语言规范，表达式元素可以包含任何有效的表达式，但你不能使用分号来结束一个表达式。

下面是 JSP 表达式的语法：

```
<%= expression %>
```

你可以编写相当于上述语法的 XML，如下所示：

```
<jsp:expression>
  expression
</jsp:expression>
```

以下是 JSP 表达式的简单的例子：

```
<html>
<head><title>A Comment Test</title></head>
<body>
<p>
  Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
</body>
</html>
```

这会生成如下所示结果：

```
Today's date: 11-Sep-2010 21:24:25
```

JSP 注释

JSP 注释标志着 JSP 容器应该忽略的文本或语句。当你想要隐藏或“注释掉”JSP 页面的一部分时，JSP 注释非常有用。

下面是 JSP 注释的语法：

```
<%-- This is JSP comment --%>
```

以下是 JSP 注释的简单的例子：

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- This comment will not be visible in the page source --%>
</body>
</html>
```

这将生成如下所示的结果：

A Test of Comments

在一些情况下，有少量的特殊结构可以用来插入注释或字符，否则会被特殊对待。这里有一个总结：

<

table class="table table-bordered"> 语法目的 <%-- comment --%> JSP 注释。JSP 引擎会忽略。 <!-- comment --> HTML 注释。浏览器会忽略。 <\%代表静态 <% 文字。 %\>代表静态 <% 文字。 \'单引号属性，使用单引号。 \"双引号属性，使用双引号。

JSP 指令

JSP 指令影响 servlet 类的总体结构。它通常具有以下形式：

```
<%@ directive attribute="value" %>
```

有三种指令标签：

指令	描述
<%@ page ... %>	定义 page-dependent 属性，比如脚本语言，错误页面和缓冲要求。
<%@ include ... %>	包括转换阶段的一个文件。
<%@ taglib ... %>	声明一个在页面中使用的标签库，包含自定义操作。

我们会在独立的章节 JSP - Directives (页 0) 中解释 JSP 指令。

JSP 操作

JSP 操作使用 XML 语法结构来控制 servlet 引擎的行为。你可以动态地插入一个文件，重用 javabean 组件，将用户转到另一个页面，或为 Java 插件生成 HTML。

操作元素只有一个语法，因为它符合 XML 标准：

```
<jsp:action_name attribute="value" />
```

操作元素基本上是预定义的函数，有以下 JSP 操作：

语法	目的
jsp:include	当请求页面时，包含一个文件

jsp:useBean	发现或实例化一个 JavaBean
jsp:setProperty	JavaBean 的属性集
jsp:getProperty	将 JavaBean 的属性嵌入到输出中
jsp:forward	将请求转发给一个新页面
jsp:plugin	生成浏览器-特定代码，为 Java 插件创建 OBJECT 或 EMBED 标签
jsp:element	动态的定义 XML 元素
jsp:attribute	定义了动态定义的 XML 元素的属性
jsp:body	定义了动态定义 XM L元素的 body
jsp:text	用于在 JSP 页面和文档中编写模板

我们将在单独的章节 JSP - Actions (页 0) 中解释 JSP 操作。

JSP 隐式对象

JSP 支持 9 个自动定义的变量，这也称为隐式对象。这些变量是：

对象	描述
request	这是与请求关联的 HttpServletRequest 对象。
response	这是与客户端响应关联的 HttpServletResponse 对象。
out	这是用于向客户端发送输出的 PrintWriter 对象。
session	这是与请求关联的 HttpSession 对象。
application	这是与应用程序上下文关联的 ServletContext 对象。
config	这是与页面关联的 ServletConfig 对象。
pageContext	这个封装特使用特定服务器的特性，如更高的性能 jspwriter。
page	这是 this 的一个简单的同义词，是用来调用由转换的 servlet 类定义的方法。
Exception	Exception对象允许指定的 JSP 访问异常数据。

我们将在独立的章节 JSP - Implicit Objects (页 0) 中解释 JSP 隐式对象。

控制流语句

JSP 提供了强有力的 Java 工具来嵌入到你的 web 应用程序中。你可以在 JSP 编程中使用所有的 API 和 Jave 构建块，包括决策语句、循环等。

决策的语句

if…else 块像普通的 Scriptlet 一样开始，但 Scriptlet 结束于包含在 Scriptlet 标签间的 HTML 文本每一行。

```
<%! int day = 3; %>
<html>
<head><title>IF...ELSE Example</title></head>
<body>
<% if (day == 1 | day == 7) { %>
    <p> Today is weekend</p>
<% } else { %>
    <p> Today is not weekend</p>
<% } %>
</body>
</html>
```

这将产生如下所示的结果：

```
Today is not weekend
```

现在看看下面的 switch…case 块，使用 out.println()语句和内部 Scriptletas 编写，与上述例子有一点区别：

```
<%! int day = 3; %>
<html>
<head><title>SWITCH...CASE Example</title></head>
<body>
<%
switch(day) {
case 0:
  out.println("It\'s Sunday.");
  break;
case 1:
  out.println("It\'s Monday.");
  break;
case 2:
  out.println("It\'s Tuesday.");
  break;
case 3:
  out.println("It\'s Wednesday.");
  break;
case 4:
  out.println("It\'s Thursday.");
  break;
case 5:
  out.println("It\'s Friday.");
  break;
default:
  out.println("It's Saturday.");
}
%>
</body>
</html>
```

这将产生如下所示的结果：

```
It's Wednesday.
```

循环语句

你还可以在 Java 中使用循环块的三种基本类型来实现 JSP 编程：for，while，和 do…while

让我们看看下面的 for 循环的例子：

```
<%! int fontSize; %>
<html>
<head><title>FOR LOOP Example</title></head>
<body>
<%for ( fontSize = 1; fontSize <= 3; fontSize++){ %>
   <font color="green" size="<%= fontSize %>">
    JSP Tutorial
   </font><br />
<%}%>
</body>
</html>
```

这将产生如下所示的结果：

JSP Tutorial

JSP Tutorial

JSP Tutorial

上述例子也可以用 while 循环来编写：

```
<%! int fontSize; %>
<html>
<head><title>WHILE LOOP Example</title></head>
<body>
<%while ( fontSize <= 3){ %>
   <font color="green" size="<%= fontSize %>">
    JSP Tutorial
   </font><br />
<%fontSize++;%>
<%}%>
</body>
</html>
```

这也会产生如下所示的结果：

JSP Tutorial

JSP Tutorial

JSP Tutorial

JSP 运算符

JSP 支持所有 Java 支持的逻辑和算术运算符。下表列出了所有运算符的优先级，从上到下，优先级依次降低。

在一个表达式，优先级越高的运算符会越先计算。

类型	运算符	结合性
后缀运算符	() [] . (dot operator)	从左到右
一目运算符	++ − − ! ~	从右到左
倍数运算符	* / %	从左到右
加法运算符	+ −	从左到右
位移运算符	>> >>> <<	从左到右
代数运算符	> >= < <=	从左到右
赋值运算符	== !=	从左到右
按位与运算符	&	从左到右
按位异或运算符	^	从左到右
按位或运算符	|	从左到右
逻辑与运算符	&&	从左到右
逻辑或运算符	||	从左到右
条件运算符	?:	从右到左
赋值运算符	= += −= *= /= %= >>= <<= &= ^= |=	从右到左
逗号运算符	,	从左到右

JSP 文字

JSP 表达式语言定义了以下文字：

- Boolean： true and false
- Integer： 与 Java 相同
- Floating point： 与 Java 相同
- String： 带有单引号和双引号；’ ’ 转义为\’ ’，’ 转义为\’，\转义为\。
- Null： null

JSP – 指令

JSP 指令为容器提供方向和指导，告诉它如何处理 JSP 过程的某些方面。

JSP 指令影响 servlet 类的总体结构。它通常具有以下形式：

```
<%@ directive attribute="value" %>
```

指令有若干个属性，你可以以键–值对的形式列出并由逗号分隔。

@ 符号和指令名称之间的空格，以及最后一个属性和结束标志 %> 之间的空格，是可选的。

指令标签有三种类型：

指令	描述
<%@ page ... %>	定义 page–dependent 属性，比如脚本语言，错误页面和缓冲要求。
<%@ include ... %>	包含在转换阶段的文件。
<%@ taglib ... %>	声明了一个用于页面中的标签库，包括自定义操作。

页面指令

页面指令用于为属于当前 JSP 页面的容器提供指示。你可以在 JSP 页面的任何地方编写页面指令代码。按照惯例，通常在 JSP 页面的顶部编写页面指令代码。

下面是页面指令的基本语法：

```
<%@ page attribute="value" %>
```

你可以编写等同于上述语法的 XML，如下所示：

```
<jsp:directive.page attribute="value" />
```

属性

以下是页面指令相关的属性列表：

属性	目的
buffer	指定一个输出流的缓冲模型。
autoFlush	控制 servlet 输出缓冲区的行为。
contentType	定义了字符编码方案。

errorPage	定义了 Java 未检查运行时异常报告的另一个 JSP 的 URL。
isErrorPage	表明这个 JSP 页面是否是由另一个 JSP 页面的 errorPage 属性指定的 URL。
extends	指定一个超类，生成的 servlet 必须扩展
import	指定在 JSP 中使用的包或类的列表，正如 Java 导入声明为 Java 类所做的相同。
info	定义一个字符串，可以访问 servlet 的 getServletInfo()方法。
isThreadSafe	为生成的 servlet 定义线程模型。
language	定义了在 JSP 页面中使用的编程语言。
session	指定 JSP 页面是否参与 HTTP 会话
isELIgnored	指定 JSP 页面中的 EL 表达式中是否将被忽略。
isScriptingEnabled	决定是否允许使用脚本元素。

关于上述属性更详细的描述，请看 Page Directive (页 0)

包含指令

包含指令用于在转换阶段包含一个文件。这个指令告诉容器在转换阶段将其他外部文件的内容与当前 JSP 合并。你可以在你的 JSP 页面中的任何位置编写 *include* 指令。

一般使用这个指令的形式如下：

```
<%@ include file="relative url" >
```

包含指令中的文件名实际上是一个相对 URL。如果你只指定一个文件名而没有相关路径，JSP 编译器就会假定文件与你的 JSP 在同一个目录下。

你可以编写等同于上述语法的 XML，如下所示：

```
<jsp:directive.include file="relative url" />
```

关于包含指令的更详细的描述，请看 Include Directive (页 0)

taglib 指令

JSP API 允许用户定义自定义的 JSP 标签，看起来像 HTML 或 XML 标签，且标签库是一组用户定义的标签，能够实现自定义的行为。

taglib 指令声明了 JSP 页面使用一组自定义标签，识别库的位置，并提供方法来确定 JSP 页面中的自定义标签。

taglib 指令遵循以下语法：

```
<%@ taglib uri="uri" prefix="prefixOfTag" >
```

其中，uri 属性值解析为容器理解的一个位置，prefix 属性通知容器什么标记是自定义操作。

你可以编写相当于上述的语法的 XML，如下所示：

```
<jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
```

关于 taglib 指令的更详细的描述，请看 Taglib Directive (页 0)

JSP – 操作

JSP 操作使用 XML 语法结构来控制 servlet 引擎的行为。你可以动态地插入一个文件，重组 JavaBean 组件，将用户转换到另一个页面，或为 Java 插件生成 HTML。

操作元素只有一个语法，因为它符合 XML 标准：

```
<jsp:action_name attribute="value" />
```

动作元素基本上是预定义的函数，有以下 JSP 操作：

语法	目的
jsp:include	当请求页面时，包含一个文件
jsp:useBean	发现或实例化一个 JavaBean
jsp:setProperty	JavaBean 的属性集
jsp:getProperty	将 JavaBean 的属性嵌入到输出中
jsp:forward	将请求转发给一个新页面
jsp:plugin	生成浏览器-特定代码，为 Java 插件创建 OBJECT 或 EMBED 标签
jsp:element	动态的定义 XML 元素
jsp:attribute	定义了动态定义的 XML 元素的属性
jsp:body	定义了动态定义 XML 元素的 body
jsp:text	用于在 JSP 页面和文档中编写模板

共同的属性

对于所有的操作元素来说，有两个属性是共同的：id 属性和 scope 属性。

- **Id 属性：** Id 属性惟一地标识操作元素，并允许在 JSP 页面内引用操作。如果操作创建了一个对象的一个实例，那么 id 值可以通过隐式对象 PageContext 来引用该操作
- **Scope属性：** 这个属性标识了操作元素的生命周期。id 属性和 scope 属性是直接相关的，因为 scope 属性决定了 id 属性相关的对象的生命周期。scope 属性有四个可能值：(a)页面，(b)请求，(c)会话和(d)应用程序。

<jsp:include> 操作

此操作允许你将文件插入到将要生成的页面中。语法如下：

```
<jsp:include page="relative URL" flush="true" />
```

该操作与包含指令不同，包含指令在 JSP 页面转换成 servlet 时插入文件，该操作在请求页面时插入文件。

下面是与包含操作相关的属性列表：

属性	描述
page	要被包含的页面的相对 URL。
flush	布尔属性决定了包含的资源在被包含之前其缓冲区是否刷新。

例子

让我们定义以下两个文件(a)date.jsp 和(b)main.jsp，如下：

以下是 date.jsp 文件的内容：

```
<p>
  Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

以下是 main.jsp 文件的内容：

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

现在让我们把所有这些文件保存在在根目录中，并试图访问 main.jsp。这将显示如下所示结果：

The include action Example

Today's date: 12-Sep-2010 14:54:22

<jsp:useBean> 操作

useBean 操作具有多种用途。它首先利用 id 和 scope 变量搜索现有对象。如果没有找到一个对象，那么它会试图创建指定的对象。

加载 bean 的最简单的方式如下：

```
<jsp:useBean id="name" class="package.class" />
```

加载 bean 类完成后，你可以使用 jsp:setProperty 和 jsp:getProperty 操作来修改和检索 bean 属性。

下面是与 useBean 操作关联的属性列表：

属性	描述
class	指定 bean 的全部包名。
type	指定将在对象中引用的变量的类型。
beanName	通过 java.beans.Beans 类中的 instantiate()方法给定 bean 名称。

在给出与这些操作有关的有效的例子之前，让我们先讨论一下 jsp:setProperty 和 jsp:getProperty 操作。

<jsp:setProperty> 操作

setProperty 操作设置了 Bean 的属性。在定义该操作之前， Bean 一定已经预定义了。有两种使用 setProperty 操作的基本的方式：

你可以使用 jsp:setProperty 之后，但是在该操作外面，使用一个 jsp:useBean 元素，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" .../>
```

在这种情况下，无论新的 bean 是否实例化或现有的 bean 是否被发现，jsp:setProperty 都会被执行。

jsp:setProperty 可以出现的第二个背景是在 jsp:useBean 元素内部，如下所示：

```
<jsp:useBean id="myName" ... >
...
  <jsp:setProperty name="myName" property="someProperty" .../>
</jsp:useBean>
```

这里，当且仅当实例化一个新对象时，jsp:setProperty 才会被执行，如果一个现有的对象被发现时，它不会被执行。

下面是与 setProperty 操作相关的属性列表：

属性	描述
name	指定了将被设置属性的 bean。该 Bean 一定是之前定义的。
property	表明了你想设置的属性。值为“*”意味着所有请求的名字与 bean 属性名字匹配的参数将被传递给适当的 setter 方法。
value	值分配给给定属性的值。参数的值为 null 或参数不存在时，setProperty 操作将被忽略。
param	param 属性是请求参数的名称，该属性会接收该请求参数的值。你不能同时使用值和参数，但是使用其中的一个是允许的。

<jsp:getProperty> 操作

getProperty 操作用于检索给定属性的值并将它转换成一个字符串，并最终将它插入到输出中。

getProperty 操作只有两个属性，两者都是必需的，其简单的语法如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name="myName" property="someProperty" .../>
```

以下是与 setProperty 操作相关的属性属性列表：

属性	描述
name	有检索属性的 Bean 的名称。Bean 一定是之前定义的。
property	property 属性是要被检索的 Bean 属性的名称。

例子

让我们定义一个测试的 bean，在例子中使用如下所示：

```
/* File: TestBean.java */
package action;
public class TestBean {
```

```
   private String message = "No message specified";
   public String getMessage() {
      return(message);
   }
   public void setMessage(String message) {
      this.message = message;
   }
}
```

编译以上代码生成 TestBean.class 文件并确保将 TestBean.class 复制在 C:\ apache-tomcat-7.0.2 \ web apps \ web - inf \ classes \action 文件夹中，且类路径变量也应设置为该文件夹：

现在在 main.jsp 文件中使用以下代码，该文件加载 bean 并 set/get 了一个简单的字符串参数：

```
<html>
<head>
<title>Using JavaBeans in JSP</title>
</head>
<body>
<center>
<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"
                 property="message"
                 value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>
</body>
</html>
```

现在试着访问 main.jsp，将会出现如下所示的结果：

Using JavaBeans in JSP

Got message....

Hello JSP...

<jsp:forward> 操作

forward 操作终止当前页面的操作并将请求转发给另一个资源，如一个静态页面，另一个 JSP 页面，或 Java Servlet。

该操作的简单的语法如下所示：

```
<jsp:forward page="Relative URL" />
```

以下是与 forward 操作相关的属性列表：

属性	描述
page	应该包括另一个资源的相对 URL，比如静态页面，另一个 JSP 页面，或 Java Servlet。

例子

让我们重用以下两个文件(a)date.jsp (b)main.jsp，如下所示：

以下是 date.jsp 文件的内容：

```
<p>
  Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

以下是 main.jsp 文件的内容：

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
</html>
```

现在让我们把所有这些文件保存在在根目录中并试图访问 main.jsp。这将显示类似如下所示的结果。这里丢弃主页的内容，只显示来自转发页面的内容。

```
Today's date: 12-Sep-2010 14:54:22
```

<jsp:plugin> 操作

插件用于将 Java 组件插入到 JSP 页面中。它决定了浏览器的类型以及需要插入的 < object > 或 < embed > 标签的类型。

如果所需的插件不存在，它将下载插件，然后执行 Java 组件。Java 组件可以是一个 Applet 或一个 JavaBean。

插件操作有几个属性，对应常用的生成 Java 组件的 HTML 标签。< param >元素也可以用来给 Applet 或 Bean 发送参数。

下面是使用插件操作的典型的语法：

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                    width="60" height="80">
  <jsp:param name="fontcolor" value="red" />
  <jsp:param name="background" value="black" />
  <jsp:fallback>
    Unable to initialize Java Plugin
  </jsp:fallback>
</jsp:plugin>
```

如果你感兴趣的话，你可以使用applet尝试该操作。一个新元素，< fallback >元素，当组件失败时，可以用来指定一个错误字符串发送给用户。

<jsp:element> 操作

<jsp:attribute> 操作

<jsp:body> 操作

< jsp:element >，< jsp:attribute >和< jsp:body >操作是用于动态的定义 XML 元素的。动态这个词是很重要的，因为这意味着 XML 元素可以在请求时生成，而不是在编译时静态生成。

下面是一个简单的例子来动态定义 XML 元素：

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
     xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<jsp:element name="xmlElement">
<jsp:attribute name="xmlElementAttr">
  Value for the attribute
</jsp:attribute>
<jsp:body>
  Body for XML element
</jsp:body>
</jsp:element>
</body>
</html>
```

在运行时会产生如下所示的 HTML 代码：

```
<html xmlns="http://www.w3c.org/1999/xhtml"
     xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<xmlElement xmlElementAttr="Value for the attribute">
  Body for XML element
</xmlElement>
</body>
</html>
```

<jsp:text> 操作

操作可以用于在 jsp 页面和文档中编写模板文本。以下是该操作的简单的语法：

```
<jsp:text>Template data</jsp:text>
```

该模板内部不能包含其他元素；它只能包含文本和 EL 表达式(注：EL 表达式将在后续章节中解释)。注意，在 XML 文件中，你不能使用如 ${whatever > 0} 的表达式，因为大于号是非法的。相反，使用 gt 形式，如 ${whatever gt 0} 或其他方法在一个 CDATA 区域中嵌入值。

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

如果你需要包含 DOCTYPE 声明，例如对 XHTML 来说，你还必须使用 < jsp:text > 元素，如下所示：

```
<jsp:text><![CDATA[<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">]]>
</jsp:text>
<head><title>jsp:text action</title></head>
<body>
```

```
<books><book><jsp:text>
   Welcome to JSP Programming
</jsp:text></book></books>

</body>
</html>
```

在使用< jsp:text >操作和不使用该操作的两种情况下尝试上述例子。

JSP – 隐式对象

JSP 隐式对象是 Java 对象，JSP 容器使隐式对象在每一个页面中对开发人员是可用的，开发人员可以直接调用它们而不用显式声明。JSP 隐式对象也称为预定义的变量。

JSP 支持九个隐式对象，如下所示：

对象	描述
request	这是与请求关联的 HttpServletRequest 对象。
response	这是与客户端响应关联的 HttpServletResponse 对象。
out	这是用于向客户端发送输出的 PrintWriter 对象。
session	这是与请求关联的 HttpSession 对象。
application	这是与应用程序上下文关联的 ServletContext 对象。
config	这是与页面关联的 servletConfig 对象。
pageContext	这个封装特使用特定服务器的特性，如更高的性能 jspwriter。
page	这是 this 的一个简单的同义词，是用来调用由转换的 servlet 类定义的方法。
Exception	Exception 对象允许指定的 JSP 访问异常数据。

request 对象

request 对象是 javax.servlet.http.HttpServletRequest 对象的一个实例。每次客户端请求一个页面时，JSP 引擎都会创建一个新的对象来表示那个请求。

Request 对象提供方法来获取 HTTP 头信息，包括表单数据，cookies，HTTP 方法等。

我们将会在接下来的章节中看到与 request 对象相关的完整的方法集合：JSP – Client Request (页 0)

response 对象

Response 对象是 javax.servlet.http.HttpServletResponse 对象的一个实例。当服务器创建 request 对象时，它也创建了代表客户端响应的对象。

Response 对象还定义了接口，可以处理创建的新的 HTTP 头。通过这个对象 JSP 程序员可以添加新的 cookies 或日期 stamps，HTTP 状态码等。

我们将会在接下来的章节中看到与 response 对象相关的完整的方法集合：JSP – Server Response (页 0)

out 对象

out 隐式对象是 javax.servlet.jsp. JspWriter 对象的一个实例，用于在响应中发送内容。

最初的 JspWriter 对象被实例化不同程度地取决于页面是否缓冲。通过使用页面指令的 buffered='false' 属性，缓冲可以很容易地关掉。

JspWriter 对象包含大部分与 java.io.PrintWriter 类相同的方法。然而，JspWriter 对象有一些额外的方法用来处理缓冲。与 PrintWriter 对象不同，JspWriter 抛出 IOException。

以下是我们用来写布尔型 char，int，double，object，String 等等的重要方法。

方法	描述
out.print(dataType dt)	输出一个数据类型的值
out.println(dataType dt)	输出数据类型值然后用新行字符终止该行。
out.flush()	刷新数据流。

session 对象

Session 对象是 javax.servlet.http.HttpSession 的一个实例，且行为与 Java servlet 中的 session 对象完全相同。

Session 对象是用来跟踪客户端请求之间的客户端会话。我们将在后续的章节中看到完整的使用 session 对象的方法：JSP – Session Tracking (页 0)

application 对象

Application 对象是用于生成的 Servlet 的 ServletContext 对象的直接包装器，且实际上是 javax.servlet.ServletContext 对象的一个实例。

这个对象是 JSP 页面整个生命周期的一个代表。当初始化 JSP 页面时，这个对象被创建，当 JSP 页面由 jspDestroy()方法删除时，该对象也会被删除。

通过为 application 添加属性，你可以确保生成 web 应用程序的所有 JSP 文件可以访问它。

你可以在 JSP – Hits Counter (页 0) 章节中查看简单的使用 Application 对象的例子。

Config 对象

Config 对象是 javax.servlet.ServletConfig 的一个实例，且是用于生成的 servlet 的 ServletConfig 对象的直接包装器。

该对象允许 JSP 程序员访问 Servlet 或 JSP 引擎初始化参数，如路径或文件位置等。

下面的 config 方法是唯一一个你可能曾经使用的方法，且它的使用很简单：

```
config.getServletName();
```

这返回 servlet 的名称，该名称是包含在定义在 web - inf \ web.xml 文件中< servlet-name >元素中的字符串。

pageContext 对象

PageContext 对象是 javax.servlet.jsp.PageContext 对象的一个实例。pageContext 对象用于表示整个 JSP 页面。

这个对象是作为一种手段来访问页面信息的,同时避免了大部分的实现细节。

这个对象为每个请求存储了请求引用和响应对象。Application，config，session，out 对象是通过访问该对象的属性派生出来的。

pageContext 对象还包含发布到 JSP 页面的指令信息，包括缓冲信息，errorPageURL，页面范围。

PageContext 类定义了几个领域，包括 PAGE_SCOPE ，REQUEST_SCOPE，SESSION_SCOPE，和 APPLICATION_SCOPE，它确定了这四个范围。它还支持 40 多个方法，大约一半的方法是继承了 javax.servlet.jsp.JspContext 类的。

重要方法之一是 removeAttribute，该方法接受一个或两个参数。例如，pageContext.removeAttribute(“attrName”) 从全部范围中删除属性，而下面的代码仅从页面范围中删除它：

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

你可以在后续的章节中查看使用 pageContext 的例子：JSP - File Uploading (页 0)。

Page 对象

这个对象是一个页面实例的真实引用。它可以被认为是一个对象，代表了整个 JSP 页面。

Page 对象实际上是 this 对象的一个直接的同义词。

exception 对象

Exception 对象是一个包装器，包含来自先前页面的异常抛出。它通常用于为错误条件生成一个适当的响应。

我们将在后续章节中看到完整的使用该对象的例子：JSP – Exception Handling (页 0)。

JSP – 客户端请求

当浏览器请求一个网页时，它向 web 服务器发送大量的信息，信息不能直接阅读，因为这些信息作为 HTTP 请求标题的一部分行进。关于这点你可以查看 HTTP Protocol (http://wiki.jikexueyuan.com.com/project/http/) 来了解更多的信息。

以下是来自浏览器端的重要的标题，在网络编程中你将会频繁的使用：

标题	描述
Accept	该标题指定了浏览器或其他客户可以处理的 MIME 类型。image/png 或 image/jpeg 的值是两种最常见的可能性。
Accept-Charset	该标题指定了浏览器可以用来显示信息的字符集。例如 iso – 8859 – 1。
Accept-Encoding	这个标题指定了浏览器知道如何处理的编码类型。gzip 或 compressare 的值是两种最常见的可能性。
Accept-Language	这个标题指定客户的首选语言，以防 servlet 可以产生多个语言的结果。例如英语，美语，俄语等。
Authorization	这个标题是客户访问密码保护的 Web 页面时用来识别他们自己的。
Connection	这个标题表明客户端是否能处理持续的 HTTP 连接。持续连接允许客户端或其他浏览器用单个请求检索多个文件。Keep-Alive 的值意味着应该使用持续连接
Content-Length	该标题只适用于 POST 请求和以字节形式给出 POST 数据的大小。
Cookie	这个标题为之前发送它们到浏览器的服务器返回 cookies。
Host	这个标题指定主机和端口正如原始URL给出的一样。
If-Modified-Since	这个标题表明，客户只想得到在指定日期后更改的页面。服务器发送一个代码，304 意味着 没有 修改标题如果没有更新的结果是可用的。
If-Unmodified-Since	这个标题的作用与 if – modified – since 是相反的；它指定当且仅当文档比指定的日期要早时，操作应该成功。
Referer	这个标题表示了引用的 Web 页面的 URL。例如，如果你在 Web 页面 1，点击一个链接到 Web 页面 2，当浏览器请求 Web 页面 2 时，web 页面 1 的 URL 是包含在引用标题中的。
User-Agent	这个标题标识浏览器或其他做出请求的客户，对应不同类型的浏览器可以返回不同的内容。

HttpServletRequest 对象

该请求对象是 javax.servlet.http. HttpServletRequest 对象的一个实例。每次客户端请求一个页面时，JSP 引擎就会创建一个新的对象来表示这个请求。

请求对象提供方法来获取 HTTP 标题信息，包括表单数据，cookies，HTTP 方法等。

有以下重要的方法可用于读取 JSP 程序中的 HTTP 标题。有了 HttpServletRequest 对象，这些方法都可用的，该对象代表客户端对网络服务器的请求。

S.N.	方法 & 描述
1	Cookie[] getCookies() 返回一个数组，其中包含客户端用这个请求发送的所有 Cookie 对象。
2	Enumeration getAttributeNames() 返回一个枚举包含此请求可用的属性的名称。
3	Enumeration getHeaderNames() 返回一个这个请求包含的所有标题名称的枚举。
4	Enumeration getParameterNames() 返回一个字符串对象的枚举，该字符串对象包括包含在此请求中的参数的名称。
5	HttpSession getSession() 返回与此请求相关的当前会话，或者如果该请求没有会话，那么就创建一个。
6	HttpSession getSession(boolean create) 返回与这个请求相关的当前的HttpSession或，如果没有当前会话并且 create 为真，那么返回一个新的会话。
7	Locale getLocale() 返回客户会接受内容的首选区域设置，基于所包含的 accept - language 标题
8	Object getAttribute(String name) 作为一个对象返回指定属性的值，如果指定的名称没有属性，返回 null。

9	ServletInputStream getInputStream() 使用 ServletInputStream 将请求的主体作为二进制数据检索。
10	String getAuthType() 返回用于保护 servlet 的验证方案的名称，例如，“BASIC”或“SSL”，如果 JSP 没有被保护，那么返回 null
11	String getCharacterEncoding() 返回在该请求内部使用的字符编码的名称。
12	String getContentType() 返回该请求主体的 MIME 类型，如果不知道类型，返回 null。
13	String getContextPath() 返回表示请求上下文的请求 URI 的一部分。
14	String getHeader(String name) 将指定的请求标题的值作为一个字符串返回。
15	String getMethod() 返回生成该请求的 HTTP 方法的名称，比如 GET，POST，或 PUT。
16	String getParameter(String name) 将一个请求参数的值作为字符串返回，如果参数不存在，返回 null。
17	String getPathInfo() 返回与客户端生成请求时发送的 URL 相关联的任何额外的路径信息。

18	String getProtocol() 返回请求协议的名称和版本。
19	String getQueryString() 返回在路径后包含在请求 URL 的查询字符串。
20	String getRemoteAddr() 返回发送请求的客户端的互联网协议(IP)地址。
21	String getRemoteHost() 返回发送请求的客户机的全称。
22	String getRemoteUser() 如果用户已经通过身份验证，就返回发出这一请求的登录用户，如果用户没有被验证，那么返回 null。
23	String getRequestURI() 从取决于 HTTP 请求首行的查询字符串的协议名称中返回请求 URL 的一部分。
24	String getRequestedSessionId() 返回客户端指定的会话 ID。
25	String getServletPath() 返回调用 JSP 的请求 URL 的部分。
26	String[] getParameterValues(String name)

	返回一个字符串对象数组，其中包含所有的给定的请求参数的值，如果参数不存在，那么返回 null。
27	boolean isSecure() 返回一个布尔值表示是否使用一个安全通道发出了这个请求，比如 HTTPS。
28	int getContentLength() 以字节为单位，返回请求的主体长度并通过输入流使其可用，如果长度是未知的，那么返回 −1。
29	int getIntHeader(String name) 作为 int 返回指定请求标题的值。
30	int getServerPort() 返回收到这个请求的端口号。

HTTP 标题请求实例

下面是使用 HttpServletRequest 的 **getHeaderNames()** 方法读取 HTTP 标题信息的实例。该方法返回一个枚举，包含与当前 HTTP 请求相关的标题信息。

一旦得到一个枚举，我们可以以标准的方式循环枚举，使用 *hasMoreElements()* 方法来确定何时停止，使用 *nextElement()* 方法得到每个参数的名字。

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>HTTP Header Request Example</title>
</head>
<body>
<center>
<h2>HTTP Header Request Example</h2>
<table width="100%" border="1" align="center">
<tr bgcolor="#949494">
<th>Header Name</th><th>Header Value(s)</th>
</tr>
<%
  Enumeration headerNames = request.getHeaderNames();
```

```
  while(headerNames.hasMoreElements()) {
    String paramName = (String)headerNames.nextElement();
    out.print("<tr><td>" + paramName + "</td>\n");
    String paramValue = request.getHeader(paramName);
    out.println("<td> " + paramValue + "</td></tr>\n");
  }
%>
</table>
</center>
</body>
</html>
```

现在把上述代码添加到 main.jsp 中并试图访问它。这将产生的如下所示的结果：

HTTP 标题请求实例

标题名称	标题值
accept	*/*
accept-language	en-us
user-agent	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; MS-RTC LM 8)
accept-encoding	gzip, deflate
host	localhost:8080
connection	Keep-Alive
cache-control	no-cache

想要用其他方法变得更加舒服，你可以用相同的方法试试上面列出的几个方法。

JSP – 服务器响应

当一个 Web 服务器响应浏览器的 HTTP 请求时，响应通常包括一个状态行，一些响应标题，一个空行和文档。一个典型的响应如下所示：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包含 HTTP 版本(例子中的 HTTP / 1.1)，状态码(例子中的 200)和对应状态代码的短消息(例子中的 OK)。

下面是最有用的 HTTP 1.1 响应标题的一个总结，它从 web 服务器端回到浏览器，并且在 web 编程时，你会频繁使用它们：

标题	描述
Allow	这个标题指定了服务器支持的请求方法(GET、POST 等等)。
Cache-Control	这个标题指定了响应文档可以安全地被缓存的情况。它可以有public，private 或 non-chche 的值。Public 意味着文件是缓存的，private 意味着文档用于单个用户，且只能存储在私有(非共享)缓存中，non-chche 意味着永远不会被缓存。
Connection	该标题表明浏览器是否使用持久的HTTP连接。值为 close 表明浏览器不使用持续的 HTTP 连接，keep-alive 表明使用持久连接。
Content-Disposition	该标题让你请求浏览器要求用户将响应保存到给定名称的磁盘文件中。
Content-Encoding	这个标题指定了在传输过程中页面被编码的方式。
Content-Language	这个标题表明了编写文档的语言。例如，英语，美语，俄语等。
Content-Length	这个标题表明了响应中的字节数。这些信息只有在浏览器使用持久(keep-alive)的HTTP连接时才需要。
Content-Type	这个标题给出响应文档的 MIME(多用途 Internet 邮件扩展)类型。
Expires	这个标题指定了内容应该被认为是过时的时间，因此不再被缓存。
Last-Modified	这个标题表示最后一次修改文档的时间。客户端可以缓存文件并由后面的请求的 if – modified – since 请求标题提供一个日期。
Location	这个标题应该包含在所有带有 300 s 状态码的响应中。该标题通知浏览器文档的地址。浏览器自动重新连接到这个位置并且检索新文档。

Refresh	这个标题指定浏览器应该多久访问更新页面。你可以在页面刷新后，指定时间为几秒。
Retry-After	这个标题可以与 503(服务不可用)响应结合使用，告诉客户端多久以后它可以重复请求。
Set-Cookie	这个标题制定了与页面相关联的一个 cookie。

HttpServletResponse 对象

该响应对象是 javax.servlet.http.HttpServletResponse 的一个实例。正如服务器创建请求对象，它也创建了一个对象来表示客户端的响应。

响应对象还定义了接口，处理创建新的 HTTP 标题。通过这个对象 JSP 程序员可以添加新的 cookies 或日期 stamps，HTTP 状态码等。

以下方法可以用来在你的 servlet 程序中设置 HTTP 响应标题。有了代表服务器响应的 HttpServletResponse 对象，这些方法都是可用的。

S.N.	方法 & 描述
1	String encodeRedirectURL(String url) 将指定的 URL 编码用于 sendRedirect 方法，如果不需要编码，则返回的 URL 不变。
2	String encodeURL(String url) 编码由包括会话 ID 指定的 URL,或者,如果不需要编码,返回的 URL 不变。
3	boolean containsHeader(String name) 返回一个布尔值表明指定的响应标题是否已经设置。
4	boolean isCommitted() 返回一个布尔值表明响应是否已经提交。
5	void addCookie(Cookie cookie)

	将指定的 cookie 添加到响应中。
6	void addDateHeader(String name, long date) 添加一个带有给定名称和日期值的响应标题。
7	void addHeader(String name, String value) 添加一个带有给定名称和值的响应标题。
8	void addIntHeader(String name, int value) 添加一个带有给定名称和整数值的响应标题。
9	void flushBuffer() 将缓冲区的内容强行写入到客户端。
10	void reset() 清除缓冲区中的全部数据，以及状态码和标题。
11	void resetBuffer() 清除响应中没有清除头或状态码的潜在的缓冲区的内容。
12	void sendError(int sc) 使用指定的状态代码给客户端发送一个错误响应，并清除缓冲区。
13	void sendError(int sc, String msg) 使用指定的状态给客户端发送一个错误响应。
14	void sendRedirect(String location)

	使用指定的重定向位置 URL 给客户端发送一个临时的重定向响应。
15	void setBufferSize(int size) 为响应主体设置首选缓冲区大小。
16	void setCharacterEncoding(String charset) 设置将被发送到客户端的响应的字符编码(MIME字符集)例如 UTF-8。
17	void setContentLength(int len) 设置 HTTP servlet 中的响应的主体内容的长度，这种方法设置了 HTTP 内容-长度标题。
18	void setContentType(String type) 如果响应尚未提交，设置要被发送到客户端的响应的内容类型。
19	void setDateHeader(String name, long date) 用给定的名称和日期值设置一个响应标题。
20	void setHeader(String name, String value) 用给定的名称和值设置一个响应标题。
21	void setIntHeader(String name, int value) 用给定的名称和整数值设置一个响应标题。
22	void setLocale(Locale loc) 如果反应尚未提交，设置响应的语言环境。

23	void setStatus(int sc) 为响应设置状态码。

HTTP 标题响应实例

接下来的例子中将使用 setIntHeader()方法设置 Refresh 标题来模拟数字时钟：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Auto Refresh Header Example</title>
</head>
<body>
<center>
<h2>Auto Refresh Header Example</h2>
<%
  // Set refresh, autoload time as 5 seconds
  response.setIntHeader("Refresh", 5);
  // Get current time
  Calendar calendar = new GregorianCalendar();
  String am_pm;
  int hour = calendar.get(Calendar.HOUR);
  int minute = calendar.get(Calendar.MINUTE);
  int second = calendar.get(Calendar.SECOND);
  if(calendar.get(Calendar.AM_PM) == 0)
    am_pm = "AM";
  else
    am_pm = "PM";
  String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
  out.println("Current Time is: " + CT + "\n");
%>
</center>
</body>
</html>
```

现在把上面的代码添加到 main.jsp 并试图访问它。这将在每5秒后显示当前系统时间如下所示。运行 JSP，等着看结果：

Auto Refresh Header Example

Current Time is: 9:44:50 PM

想要变得更加舒适你可以以相同的方式试试上面几个方法。

JSP – HTTP 状态码

HTTP 请求格式和 HTTP 响应消息的格式一样，都有以下结构：

- 一个初始状态行+ CRLF(回车+换行，即新行)
- 零个或多个标题行+ CRLF
- 一个空行，即一个 CRLF
- 一个可选的消息体，像文件，查询数据或查询输出。

例如,一个服务器响应标题看起来如下所示：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包含 HTTP 版本(例子中的 HTTP / 1.1)，状态码(例子中的 200)和对应状态代码的短消息(例子中的 OK)。

下面是 HTTP 状态代码和相关可能从Web服务器返回的消息的一个列表：

编码：	消息：	描述：
100	Continue	只有一部分的服务器请求已经收到，但只要没有被拒绝，客户端应该继续请求
101	Switching Protocols	服务器交换了协议。
200	OK	请求是 OK
201	Created	请求已经完成，新的资源被创建
202	Accepted	请求被接受处理，但是处理还没有完成。
203	Non-authoritative Information	
204	No Content	
205	Reset Content	
206	Partial Content	

300	Multiple Choices	一个链接列表。用户可以选择一个链接然后跳转到那个位置。最多可选择 5 个地址
301	Moved Permanently	请求页面已经被移到新的 url 中
302	Found	请求页面暂时被移到新的 url 中
303	See Other	请求页面可在不同的 url 中找到
304	Not Modified	
305	Use Proxy	
306	*Unused*	该代码是在前一版本使用的。它已不再使用，但该代码保留下来了。
307	Temporary Redirect	请求页面被暂时移到新的url中。
400	Bad Request	服务器没有理解请求。
401	Unauthorized	请求页面需要用户名和密码
402	Payment Required	*你还不能使用这个代码*
403	Forbidden	不允许访问请求页面
404	Not Found	服务器找不到请求页面。
405	Method Not Allowed	在请求中指定的方法不允许使用。
406	Not Acceptable	服务器只能生成一个不被客户端接收的响应。
407	Proxy Authentication Required	在这个请求得到服务之前，你必须验证一个代理服务器。
408	Request Timeout	请求花费的时间比服务器准备等待的时间长。
409	Conflict	由于冲突请求不能实现。
410	Gone	请求页面不再可用。
411	Length Required	"内容-长度" 没有被定义。没有它服务器不会接受请求。
412	Precondition Failed	服务器给出给定的请求评估的前提为假。
413	Request Entity Too Large	服务器不会接受请求，因为请求实体太大。
414	Request-url Too Long	服务器不会接受请求，因为 url 太长。当你把“post”请求转换为带有很长的查询信息的“get”请求时，这种情况就会发生
415	Unsupported Media Type	服务器不会接受请求因为媒体类型不支持。
417	Expectation Failed	
500	Internal Server Error	请求未完成。服务器遇到了意外情况。
501	Not Implemented	请求未能完成。服务器不支持所需的功能。
502	Bad Gateway	请求未能完成。服务器从上游服务器收到无效响应
503	Service Unavailable	请求未能完成。服务器暂时过载或瘫痪。
504	Gateway Timeout	网关超时。
505	HTTP Version Not Supported	服务器不支持“http 协议”版本。

设置 HTTP 状态码的方法：

有以下方法可以用来设置 servlet 程序的 HTTP 状态码。有了 HttpServletResponse 对象，这些方法都是可用的。

S.N.	方法 &描述
1	public void setStatus (int statusCode) 此方法设置一个任意的状态码。setStatus 方法接受一个 int(状态码)作为参数。如果你的响应包含一个特殊的状态码和文档，在实际中用 PrintWriter 返回任何内容之前一定要调用 setStatus。
2	public void sendRedirect(String url) 该方法生成一个 302 响应以及一个位置标题给出新文档的 URL。
3	public void sendError(int code, String message) 这种方法发送一个状态码(通常是 404)以及一个在 HTML 文档内自动格式化的短消息并发送到客户端。

HTTP 状态码实例：

下面的例子将发送 407 错误代码到客户端浏览器，浏览器将显示“需要验证!!!”的消息。

```
<html>
<head>
<title>Setting HTTP Status Code</title>
</head>
<body>
<%
  // Set error code and reason.
  response.sendError(407, "Need authentication!!!" );
%>
</body>
</html>
```

现在调用上述代码，JSP 将显示如下所示的结果：

HTTP Status 407 - Need authentication!!!

type Status report

message Need authentication!!!

description The client must first authenticate itself with the proxy (Need authentication!!!).

Apache Tomcat/5.5.29

想要使用 HTTP 状态代码变得更加舒适，尝试设置不同的状态代码和描述。

JSP – 表单处理

当你需要从你的浏览器向 web 服务器传递一些信息并最终将信息返回到你的后端程序时，你一定遇到了很多情况。浏览器使用两种方法将此信息传递给 web 服务器。这些方法是 GET 方法和 POST 方法。

GET 方法

GET 方法发送添加到页面请求的编码用户信息。页面和编码信息是被符号？ 分开的，如下所示：

```
http://www.test.com/hello?key1=value1&key2=value2
```

GET 方法是从浏览器向 web 服务器传递信息的默认的方法，它产生一个长字符串出现在浏览器的位置框中。如果你要把密码或其他敏感信息传递到服务器，那么不要使用 GET 方法。

GET 方法有大小限制：在请求字符串中只可以有 1024 个字符。

这些信息是通过使用 QUERY_STRING 标题传递的，并将通过 QUERY_STRING 环境变量被接收，该环境变量可以使用请求对象的 getQueryString() 方法和 getParameter()方法处理。

POST 方法

通常情况下，将信息传递给后端程序的更可靠的方法是 POST 方法。

该方法打包信息的方法与 GET 方法是完全一样的，但是它不是将信息作为一个文本字符串放在 URL 中的？ 符号之后来发送信息，它是把信息作为一个单独的消息来发送该消息。这个消息是以标准输入的形式发送到后端程序的，在你的处理过程中你可以解析并使用这个消息。

JSP 处理这种类型的请求时，使用 getParameter() 方法读取简单参数，使用 getInputStream()方法读取来自客户端的二进制数据流。

使用 JSP 读取表单数据

JSP 以自动解析的方式处理表单数据，根据情况不同使用以下不同的方法：

- getParameter()： 调用 request.getParameter() 方法得到一种形式参数的值。
- getParameterValues()：如果参数出现不止一次，那么就调用这个方法并返回多个值，例如复选框。

- getParameterNames()：如果你想要在当前请求下得到一个所有参数的完整的列表，那么调用这个方法。
- getInputStream()： 调用这个方法读取来自客户端二进制数据流。

使用 URL 的 GET 方法示例

这是一个简单的 URL 示例，使用 GET 方法将两个值传递给 HelloForm 程序。

`http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI`

下面是 main.jsp JSP程序来处理由 web 浏览器给定的输入。我们将使用 getParameter() 方法使访问传递信息变得容易：

```
<html>
<head>
<title>Using GET Method to Read Form Data</title>
</head>
<body>
<center>
<h1>Using GET Method to Read Form Data</h1>
<ul>
<li><p><b>First Name:</b>
  <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
  <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

现在在你的浏览位置框中键入 `http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI` 。这将产生如下所示的结果：

Using GET Method to Read Form Data

- First Name: ZARA
- Last Name: ALI

使用表单的 GET 方法的示例

这是一个简单的例子，使用HTML表单和提交按钮传递两个值。我们将使用相同的 JSP main.jsp 来处理这个输入。

```
<html>
<body>
<form action="main.jsp" method="GET">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

将这个 HTML 保存在 hello.htm 文件中，并把它放在 < Tomcat-installation-directory > / webapps /ROOT 目录下。当你访问 http:/ / localhost:8080 /hello.htm 时，这就是在表单上面的实际输出。

First Name:

Last Name: Submit

尝试输入姓和名，然后单击提交按钮来查看 tomcat 运行的本地机器上的结果。基于提供的输入，它会产生与上面提到的例子中相似的结果。

使用表单的 POST 方法的示例

让我们在上面的 JSP 中做一些小修改来处理 GET 和 POST 方法。下面是 main.jsp JSP 程序使用 GET 或 POST 方法来处理 web 浏览器给定的输入。

事实上上面的 JSP 没有改变，因为只有传递参数的方法改变了，并且没有二进制数据被传递到 JSP 程序。文件处理相关的概念将在单独的一章中解释，在这里我们需要读取二进制数据流。

```
<html>
<head>
<title>Using GET and POST Method to Read Form Data</title>
</head>
<body>
<center>
<h1>Using GET Method to Read Form Data</h1>
```

```
<ul>
<li><p><b>First Name:</b>
  <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
  <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

以下是 hello.htm 文件中的内容：

```
<html>
<body>
<form action="main.jsp" method="POST">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

现在让我们把 main.jsp 和 hello.htm 保存到 < Tomcat-installation-directory > / webapps /ROOT 目录下。当你访问 http:/ / localhost:8080 /hello.htm 时，下面是表单的实际输出。

First Name: [ ]

Last Name: [ ] Submit

尝试输入姓名，然后单击提交按钮来查看 tomcat 运行的本地机器上的结果。

基于提供的输入，它会产生与上面提到的例子中类似的结果。

将复选框数据传递给 JSP 程序

当需要选择多个选项时，就使用复选框。

这是HTML 代码的示例，CheckBox.htm，一个表单中有两个复选框

```
<html>
<body>
<form action="main.jsp" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" />
                                  Chemistry
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

上述代码的结果如下所示：

? Maths ☐ Physics ? Chemistry Select Subject

以下是 main.jsp JSP 程序为复选框处理由浏览器给定的输入。

```
<html>
<head>
<title>Reading Checkbox Data</title>
</head>
<body>
<center>
<h1>Reading Checkbox Data</h1>
<ul>
<li><p><b>Maths Flag:</b>
  <%= request.getParameter("maths")%>
</p></li>
<li><p><b>Physics Flag:</b>
  <%= request.getParameter("physics")%>
</p></li>
<li><p><b>Chemistry Flag:</b>
  <%= request.getParameter("chemistry")%>
</p></li>
</ul>
</body>
</html>
```

以上的例子将会产生以下的结果：

Reading Checkbox Data

- Maths Flag : : on
- Physics Flag: : null
- Chemistry Flag: : on

读取全部表单参数

以下是使用 HttpServletRequest 的 getParameterNames() 方法读取所有可用的表单参数的通用的例子。该方法返回一个枚举，包含一个未指定序列的参数名称。

一旦我们得到一个枚举，我们可以以标准的方式循环这个枚举，使用 *hasMoreElements()*方法来确定何时停止循环，使用 *nextElement() 方法得到每个参数的名字。

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>HTTP Header Request Example</title>
</head>
<body>
<center>
<h2>HTTP Header Request Example</h2>
<table width="100%" border="1" align="center">
<tr bgcolor="#949494">
<th>Param Name</th><th>Param Value(s)</th>
</tr>
<%
  Enumeration paramNames = request.getParameterNames();

  while(paramNames.hasMoreElements()) {
    String paramName = (String)paramNames.nextElement();
    out.print("<tr><td>" + paramName + "</td>\n");
    String paramValue = request.getHeader(paramName);
    out.println("<td> " + paramValue + "</td></tr>\n");
  }
%>
</table>
</center>
</body>
</html>
```

以下是hello.htm 中的内容：

```
<html>
<body>
<form action="main.jsp" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" /> Chem
<input type="submit" value="Select Subject" />
```

```
</form>
</body>
</html>
```

现在试着使用上述 hello.htm 调用JSP，这将生成一个基于给定的输入的结果，类似情况如 下所示：

读取全部表单参数

参数名称	参数值
maths	on
chemistry	on

你可以试着用上述 JSP 读取其他任何有其他对象的表单数据，如文本框，单选按钮或下拉框等。

JSP – 过滤器

Servlet 和 JSP 过滤器都是 Java 类，可以在 Servlet 和 JSP 编程中用于以下目的：

- 在请求访问后端资源之前从客户端拦截请求。
- 在响应发送回客户端之前从服务器操作响应。

有各种符合规格的过滤器：

- 身份验证过滤器。
- 数据压缩过滤器
- 加密过滤器。
- 触发资源访问事件的过滤器。
- 图像转换过滤器。
- 日志记录和审计过滤器。
- MIME 类型链过滤器。
- Tokenizing 过滤器。
- 转换 XML 内容的 XSL/T 过滤器。

过滤器是部署在部署描述符文件 **web.xml** 中的，然后映射到应用程序的部署描述符中 的 servlet 或 JSP 名称或 URL 模式。部署描述符文件 web.xml 可以在 *< Tomcat-installation-directory > \ conf* 目录下找到。

当 JSP 容器启动 web 应用程序时，它会为每个在部署描述符中声明的过滤器创建一个实例。过滤器按照它们在部署描述符中声明的顺序执行。

Servlet 过滤器方法

一个过滤器是一个简单的 Java 类，实现了 javax.servlet.Filter 接口。javax.servlet.Filter接口定义了三个方法：

S.N.	方法 & 描述

1	public void doFilter (ServletRequest, ServletResponse, FilterChain) 由于客户端在链尾请求响应，每次请求/响应对通过链时，容器会调用此方法。
2	public void init(FilterConfig filterConfig) 由 web 容器调用此方法，向过滤器表明它将被放置在服务中。
3	public void destroy() 由 web 容器调用此方法，向过滤器表明它将从服务中被去掉。

JSP 过滤器示例

下面是 JSP 过滤器的示例，每次访问任何 JSP 文件时都会输出客户端 IP 地址和当前日期时间。这个例子会使你基本了解 JSP 过滤器，但是你可以使用相同的概念编写更复杂的过滤器应用程序：

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// Implements Filter class
public class LogFilter implements Filter  {
  public void  init(FilterConfig config)
                  throws ServletException{
    // Get init parameter
    String testParam = config.getInitParameter("test-param");

    //Print the init parameter
    System.out.println("Test Param: " + testParam);
  }
  public void  doFilter(ServletRequest request,
            ServletResponse response,
            FilterChain chain)
            throws java.io.IOException, ServletException {

    // Get the IP address of client machine.
    String ipAddress = request.getRemoteAddr();

    // Log the IP address and current timestamp.
    System.out.println("IP "+ ipAddress + ", Time "
                          + new Date().toString());

    // Pass request back down the filter chain
    chain.doFilter(request,response);
  }
  public void destroy( ){
```

```
    /* Called before the Filter instance is removed
    from service by the web container*/
  }
}
```

以通常的方式编译 LogFilter.java，把 LogFilter.class 类文件放在 < Tomcat-installation-directory > / web apps /ROOT/ web - inf / classes 中。

在 Web.xml 中的 JSP 过滤器映射

首先定义过滤器，然后将过滤器映射到 URL 或 JSP 文件的名字中，这几乎与定义 Servlet 然后映射到 web. Xml 文件的 URL 模式的方式相同。为在部署描述符文件 web.xml 的过滤器标签创建以下条目

```
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>LogFilter</filter-class>
  <init-param>
    <param-name>test-param</param-name>
    <param-value>Initialization Paramter</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

上面的过滤器将适用于所有 servlet 和 JSP，因为我们在配置中指定了**/ ***。如果你想把 过滤器应用到部分 servlet 或 JSP 中，你可以指定一个特定的 servlet 或 JSP 路径。

现在尝试以常用的方式调用任何 servlet 或 JSP，你会在 web 服务器日志中看到生成的登 录。你可以使用 Log 4J 日志在一个单独的文件中记录上面的登录。

使用多个过滤器

你的 web 应用程序可能定义了带有特定目的的不同的过滤器。考虑这个情况，你定义两个过滤器 *AuthenFilter* 和 *LogFilter*。剩下的过程仍将像前面解释的那样，除了你需要创建一个不同的映射，如下所示：

```
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>LogFilter</filter-class>
  <init-param>
    <param-name>test-param</param-name>
    <param-value>Initialization Paramter</param-value>
```

```
  </init-param>
</filter>
<filter>
  <filter-name>AuthenFilter</filter-name>
  <filter-class>AuthenFilter</filter-class>
  <init-param>
    <param-name>test-param</param-name>
    <param-value>Initialization Paramter</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>AuthenFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

过滤器的应用顺序

在 web.xml 中的 filter-mapping 元素的顺序决定了 web 容器把过滤器应用到 servlet 或JSP 的顺序。想要逆转滤波器的顺序，你只需要逆转 web.xml 文件中的 filter-mapping 元素。

例如，上面的例子将首先应用 LogFilter ，然后应用 AuthenFilter 到任意 servlet 或 JSP 中，但是下面的例子将逆转这个顺序：

```
<filter-mapping>
  <filter-name>AuthenFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

JSP – Cookies 处理

Cookies 是存储在客户端计算机的文本文件，保存各种跟踪目标的信息。JSP 使用底层 servlet 技术透明地支持 HTTP cookies。

确定返回用户有三个步骤：

- 服务器脚本向浏览器发送的一系列cookies。例如姓名、年龄、身份证号码等。
- 浏览器将这个信息存储在本地机器上，以供将来使用。
- 下次当浏览器向web服务器发送任何请求时，将这些 cookies 信息发送给服务器，服务器使用这些信息来识别用户或可能用于其他目的。

本章将教你如何设置或重置 cookies，如何访问它们，以及如何使用 JSP 程序删除它们。

Cookie 的剖析

Cookie 通常设置在一个 HTTP 标题中(尽管 JavaScript 也可以在浏览器中直接设置cookie)。设置 cookie 的 JSP可能发送如下所示的标题信息：

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
            path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```

正如你所看见的，Set-Cookie 标题包含一个名称值对，GMT 时间，路径和一个域。名称和 值将被 URL 编码。结束字段是在给定的时间和日期之后，向浏览器发出指令来“忘记”cookie。

如果配置浏览器来存储 cookie，然后它会保存这个信息直到截止日期。如果用户在任何与cookie 的路径和域相匹配的页面点击浏览器，它将把 cookie 重新发送到服务器。浏览器的标题看起来如下所示：

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
```

```
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

Servlet Cookies 方法

下面是与 Cookie 对象关联的有用的方法列表，你可以在 JSP 中操作 cookies 时使用：

S.N.	方法 & 描述
1	public void setDomain(String pattern) 此方法设置了 cookie 适用的领域，例如 tutorialspoint.com。
2	public String getDomain() 此方法得到了 cookie 适用的领域，例如 tutorialspoint.com。
3	public void setMaxAge(int expiry) 该方法设置了在 cookie 到期之前需要多少时间(以秒为单位)。如果你不设置这个，cookie 只持续到当前会话。
4	public int getMaxAge() 该方法返回了最大持续时间的 cookie，以秒为单位指定，默认情况下，-1表示 cookie 会持续到浏览器关闭。
5	public String getName() 该方法将返回 cookie 的名称。这个名字创建后不能更改。
6	public void setValue(String newValue) 这个方法设置了与 cookie 相关的值。
7	public String getValue()

	这个方法得到了与 cookie相关的值。
8	public void setPath(String uri) 该方法设置了 cookie 应用的路径。如果你不指定路径，那么与当前页面相同的目录以及子目录中的所有 URL都会返回 cookie。
9	public String getPath() 该方法获取 cookie 应用的路径。
10	public void setSecure(boolean flag) 此方法设置布尔值，该值表明 cookie 是否只能通过加密连接发送(例如 SSL)。
11	public void setComment(String purpose) 这种方法指定了描述 cookie 目的的评论。如果浏览器向用户展示了这个 cookie，那么评论是有用的。
12	public String getComment() 该方法返回描述 cookie 目的的评论，如果 cookie 没有评论，那么返回 null。

用 JSP 设置 Cookies

用 JSP 设置 Cookies 包括三个步骤：

(1) 创建一个 Cookie 对象： 用 cookie 的名称和值调用 cookie 构造函数，名称和值是字符串。

```
Cookie cookie = new Cookie("key","value");
```

记住，这个名字和值都不应该包含空格或任何以下字符：

```
[]()=,"/?@:;
```

(2) 设置最大持续时间： 使用 setMaxAge 指定 cookie 的有效期是多长时间(以秒为单位)。以下是建立了一个持续 24 小时的 cookie。

```
cookie.setMaxAge(60*60*24);
```

(3) 将 cookie 发送到 HTTP 响应标题中： 使用 response.addCookie 在 HTTP 响应标题中添加 cookies，如下所示：

```
response.addCookie(cookie);
```

示例：

让我们修改 表单示例 (页 0) 为名称设置 cookies。

```
<%
  // Create cookies for first and last names.
  Cookie firstName = new Cookie("first_name",
            request.getParameter("first_name"));
  Cookie lastName = new Cookie("last_name",
           request.getParameter("last_name"));

  // Set expiry date after 24 Hrs for both the cookies.
  firstName.setMaxAge(60*60*24);
  lastName.setMaxAge(60*60*24);

  // Add both the cookies in the response header.
  response.addCookie( firstName );
  response.addCookie( lastName );
%>
<html>
<head>
<title>Setting Cookies</title>
</head>
<body>
<center>
<h1>Setting Cookies</h1>
</center>
<ul>
<li><p><b>First Name:</b>
  <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
  <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

将上述代码添加到 main.jsp 文件中，并在下述的 HTML 页面中使用：

```
<html>
<body>
<form action="main.jsp" method="GET">
```

```
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

将上述 HTML 内容保存在hello.jsp 文件中，并将hello.jsp 和 main.jsp 添加到 < Tomcat-installation-directory > / webapps /ROOT 目录中。当你访问 http:// localhost:8080/hello.jsp时，这是表单上的实际输出。

First Name:

Last Name: Submit

尝试输入姓名，然后单击 submit 按钮。这将在你的屏幕上显示姓名，同时会设置 firstName 和 lastName 这两个 cookie，当你下次点击 submit 按钮时，将传回服务器。

下一节将解释如何在你的web应用程序中访问这些 cookies。

用 JSP 读取 Cookies

想要读取 cookie，你需要通过调用 HttpServletRequest 的 getCookies() 方法创建一个 *javax.servlet.http* 数组。然后通过数组循环，使用 getName() 和 getValue() 方法来访问每个 cookie 和相关的值。

示例

让我们读取之前例子中设置的 cookies：

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
   Cookie cookie = null;
   Cookie[] cookies = null;
   // Get an array of Cookies associated with this domain
   cookies = request.getCookies();
   if( cookies != null ){
      out.println("<h2> Found Cookies Name and Value</h2>");
      for (int i = 0; i < cookies.length; i++){
         cookie = cookies[i];
         out.print("Name : " + cookie.getName( ) + ",  ");
         out.print("Value: " + cookie.getValue( )+" <br/>");
      }
  }else{
```

```
    out.println("<h2>No cookies founds</h2>");
  }
%>
</body>
</html>
```

现在让我们将上面的代码放在 main.jsp 文件中并试图访问它。如果你要设置 first_name cookie为“John”，last_name cookie 为“Player”，然后运行 `http:// localhost:8080/main.jsp`，将显示下面的结果：

Found Cookies Name and Value

Name : first_name, Value: John

Name : last_name, Value: Player

用 JSP 删除 Cookies

删除 cookies 非常简单。如果你想删除一个 cookie，那么你只需要按照以下三步来处理：

- 读取一个已经存在的 cookie 并把它保存在 Cookie 对象中。
- 使用 setMaxAge() 方法将 cookie 的持续时间设置为 0 来删除一个已经存在的 cookie。
- 将这个 cookie 添加到响应标题中。

示例

以下的例子中删除了现存的命名为“first_name”的 cookie，当你下次运行 main.jsp JSP时，first_name 会返回空值。

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
  Cookie cookie = null;
  Cookie[] cookies = null;
  // Get an array of Cookies associated with this domain
  cookies = request.getCookies();
  if( cookies != null ){
    out.println("<h2> Found Cookies Name and Value</h2>");
    for (int i = 0; i < cookies.length; i++){
      cookie = cookies[i];
      if((cookie.getName( )).compareTo("first_name") == 0 ){
        cookie.setMaxAge(0);
        response.addCookie(cookie);
        out.print("Deleted cookie: " +
        cookie.getName( ) + "<br/>");
      }
      out.print("Name : " + cookie.getName( ) + ",  ");
      out.print("Value: " + cookie.getValue( )+" <br/>");
    }
 }else{
```

```
    out.println(
    "<h2>No cookies founds</h2>");
  }
%>
</body>
</html>
```

现在将上述代码添加到 main.jsp 中并尝试访问它。它会出现如下所示的结果：

Cookies Name and Value

Deleted cookie : first_name

Name : first_name, Value: John

Name : last_name, Value: Player

现在尝试再次运行 `http://localhost:8080/main.jsp`，只会出现如下所示的一个 cookie：

Found Cookies Name and Value

Name : last_name, Value: Player

你可以在 IE 浏览器中自动的删除你的 cookies。单击工具菜单并选择网络选项。要删除所有 cookie，按下 Delete Cookies。

JSP – 会话跟踪

HTTP 是一个“无状态”协议，这意味着每一次客户端检索 Web 页面时，客户端打开一个单独的 Web 服务器且服务器不会自动连接任何以前的客户端请求的记录。

Web 客户端和 web 服务器之间的会话有以下三种方式：

Cookies

网络服务器可以为每个 web 客户端和使用已接收的 cookie 可识别的来自客户端的后续请求分配一个唯一的会话 ID 作为 cookie。

这可能不是一个有效的方法，因为许多时间浏览器不支持 cookie，所以我不建议使用这个过程来维护会话。

隐藏的表单字段

一个 web 服务器可以发送一个隐藏的 HTML 表单字段以及一个独特的会话 ID，如下所示：

```
<input type="hidden" name="sessionid" value="12345">
```

这个条目意味着，提交表单时，指定的名称和值会自动包含在 GET 或 POST 数据中。每次当 web 浏览器发送回请求时，session_id 的值可以用来跟踪不同的 web 浏览器。

这可能是跟踪会话的一个有效的方式，但点击一个常规的(< a HREF…>)超文本链接不会引起表单提交，所以隐藏表单字段也不能支持通用会话跟踪。

URL重写

你可以在每个识别会话的 URL 结尾添加一些额外的数据，且服务器可以用它存储的关于会话的数据与会话标识符关联起来。

例如，`http://tutorialspoint.com/file.htm;sessionid = 12345`，会话标识符作为 `sessionid = 12345` 附加，也可以在web服务器访问来识别客户端。

当它们不支持 cookie 时，URL 重写是维持会话和适用于浏览器的一个更好的方法，但缺点是尽管页面是简单的静态 HTML 页面，但需要动态的生成每个URL来分配一个会话 ID。

会话对象

除了上面提到的三种方法，JSP 使用 servlet 提供的 HttpSession 接口，该接口提供了一种方法来识别网站中跨多个页面请求或访问的用户，并存储用户信息。

默认情况下，JSP 为每个新的客户端自动的启用会话跟踪和一个新的被实例化的 HttpSession 对象。禁用会话跟踪需要通过设置页面指令会话属性为 false 显式地关闭它，如下所示：

```
<%@ page session="false" %>
```

JSP 引擎通过隐式的 session 对象暴露了 HttpSession 对象给 JSP 开发人员。由于**会话**对象已经提供给 JSP 程序员，程序员可以立即从对象中开始存储和检索数据，不需要任何初始化或 getSession()。

这是会话对象的可用的重要方法的一个总结：

S.N.	方法 & 描述
1	public Object getAttribute(String name) 该方法返回会话中与指定的名称绑定的对象，如果没有绑定对象名称，返回 null。
2	public Enumeration getAttributeNames() 该方法返回一个字符串对象的枚举，其中包含绑定到这个会话的所有对象的名字。
3	public long getCreationTime() 该方法返回创建会话的时间，自GMT时间1970年1月1日凌晨以来，以毫秒为单位。
4	public String getId() 该方法返回一个字符串，其中包含分配给这个会话的唯一标识符。
5	public long getLastAccessedTime() 该方法返回上次客户端发送与这个会话相关的请求的时间，自GMT时间1970年1月1日凌晨以来，毫秒的数量。
6	public int getMaxInactiveInterval() 该方法返回最大时间间隔，以秒为单位，servlet 容器在客户端访问中将打开这个会话。
7	public void invalidate() 这个方法使会话无效并解除全部绑定到该会话的对象。
8	public boolean isNew(如果客户端还不知道会话或如果客户选择不加入会话，这个方法返回 true。
9	public void removeAttribute(String name) 该方法从会话中删除制定名称的绑定对象。
10	public void setAttribute(String name, Object value) 该方法使用指定的名称解除会话的一个对象。
11	public void setMaxInactiveInterval(int interval) 在 servlet 容器使这个会话无效之前，这种方法在客户端请求之间指定了时间，以秒为单位，。

会话跟踪示例

这个例子描述了如何使用 HttpSession 对象发现一个会话的创建时间和上次访问时间。如果会话不存在，我们将会使请求与一个新的会话相关联。

```
<%@ page import="java.io.*,java.util.*" %>
<%
  // Get session creation time.
  Date createTime = new Date(session.getCreationTime());
  // Get last access time of this web page.
  Date lastAccessTime = new Date(session.getLastAccessedTime());

  String title = "Welcome Back to my website";
  Integer visitCount = new Integer(0);
  String visitCountKey = new String("visitCount");
  String userIDKey = new String("userID");
  String userID = new String("ABCD");

  // Check if this is new comer on your web page.
  if (session.isNew()){
    title = "Welcome to my website";
    session.setAttribute(userIDKey, userID);
    session.setAttribute(visitCountKey,  visitCount);
  }
  visitCount = (Integer)session.getAttribute(visitCountKey);
  visitCount = visitCount + 1;
  userID = (String)session.getAttribute(userIDKey);
  session.setAttribute(visitCountKey,  visitCount);
%>
<html>
<head>
<title>Session Tracking</title>
</head>
<body>
<center>
<h1>Session Tracking</h1>
</center>
<table border="1" align="center">
<tr bgcolor="#949494">
  <th>Session info</th>
  <th>Value</th>
</tr>
<tr>
  <td>id</td>
  <td><% out.print( session.getId()); %></td>
</tr>
<tr>
  <td>Creation Time</td>
  <td><% out.print(createTime); %></td>
</tr>
<tr>
  <td>Time of Last Access</td>
  <td><% out.print(lastAccessTime); %></td>
</tr>
<tr>
  <td>User ID</td>
  <td><% out.print(userID); %></td>
</tr>
```

```
<tr>
  <td>Number of visits</td>
  <td><% out.print(visitCount); %></td>
</tr>
</table>
</body>
</html>
```

现在将上述代码添加到 main.jsp 中并尝试访问 `http://localhost:8080/main.jsp` 。当你第一次运行时将会出现如下所示的结果：

Welcome to my website

Session Infomation

Session info	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

现在尝试第二次运行相同的 JSP，将会出现如下所示的结果。

Welcome Back to my website

Session Infomation

info type	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

删除会话数据

当你完成了用户会话数据，你有以下几种选择：

- **删除一个特定的属性：** 你可以调用 * public void removeAttribute(String name) * 方法来删除与特定的键相关的值。
- **删除整个会话：** 你可以调用 * public void invalidate() * 方法来删除整个会话。
- **设置会话超时：** 你可以调用 * public void setMaxInactiveInterval(int interval) * 方法分别为每个会话设置超时。
- **注销用户：** 服务器可以支持 servlet 2.4，你可以调用 logout 来注销 Web 服务器的客户端，并使所有属于该的用户的会话无效。
- **web.xml 配置：** 如果你使用的是 Tomcat，除了上述方法外，你还可以在 web.xml文件中设置会话超时，如下所示。

```
<session-config>
  <session-timeout>15</session-timeout>
 </session-config>
```

超时以分钟为单位，在 Tomcat 中，默认的超时为 30 分钟。

在 servlet 中的 getMaxInactiveInterval() 方法为会话一个以秒为单位的超时时间。所以如果你的会话在 web.xml 中设置为 15 分钟，那么 getMaxInactiveInterval() 返回 900。

JSP – 文件上传

一个 JSP 可以用一个 HTML 表单标签，它允许用户上传文件到服务器。上传的文件是一个文本文件或二进制文件，图像文件或任何文件。

创建文件上传形式

用下面的 HTM 代码创建一个上传表单。以下的要点应该记下来：

- 表单 method 的属性应该设置为 POST 方法，不能使用 GET 方法。
- 表单 enctype 的属性应该设置为 multipart/formdata。
- 表单 action 的属性应该设置为一个把文件上传到后台服务器的 JSP 文件。下面的例子是使用程序 uploadFile.jsp 来上传文件。
- 为了上传一个单一的文件，你应该使用一个带有属性type="file"的< input .../ >标签。为了允许多个文件上传，包含多个输入标签，它们带有不同的name属性的值。浏览器将浏览按钮与每个输入标签进行关联。

```
<html>
<head>
<title>File Uploading Form</title>
</head>
<body>
<h3>File Upload:</h3>
Select a file to upload: <br />
<form action="UploadServlet" method="post"
                    enctype="multipart/form-data">
<input type="file" name="file" size="50" />
<br />
<input type="submit" value="Upload File" />
</form>
</body>
</html>
```

它将显示如下结果，它允许从本地电脑中选择一个文件，当用户点击“Upload File”， 表单将连同选中的文件提交：

File Upload:

Select a file to upload:

Browse...

```
Upload File
```

注意：以上的表单仅仅是虚拟的表单，不能运行，你应该尝试将上述文件放在你的机器中运行。

书写后台 JSP Script

首先，让我们定义一个上传文件要存储的位置。你可以在你的程序中进行硬编码或该目录名称也可以使用外部配置来添加，如在 web.xml 的 context-param 元素中，如下所示：

```
<web-app>
....
<context-param>
   <description>Location to store uploaded file</description>
   <param-name>file-upload</param-name>
   <param-value>
       c:\apache-tomcat-5.5.29\webapps\data\
    </param-value>
</context-param>
....
</web-app>
```

下面是 UploadFile.jsp 的源代码，它可以一次上传多个文件。处理开始之前，你应该确保下列事项：

- 下面的例子取决于 FileUpload，所以确保你在 classpath 中有最新的版本 commons-fileupload.x.x.jar 文件。你可以从 http://commons.apache.org/fileupload/ 中下载它。
- FileUpload 取决于 Commons IO，所以确保你在 classpath 中有最新的版本 commons-io-x.x.jar 文件。你可以从 http://commons.apache.org/io/ 中下载它。
- 当测试下面的例子时，你应该上传小于 maxFileSize 的文件，否则文件不能上传。
- 确保你提前创建好目录 c:\temp 和 c:\apache-tomcat-5.5.29\webapps\data。

```
<%@ page import="java.io.*,java.util.*, javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>
<%@ page import="org.apache.commons.fileupload.*" %>
<%@ page import="org.apache.commons.fileupload.disk.*" %>
<%@ page import="org.apache.commons.fileupload.servlet.*" %>
<%@ page import="org.apache.commons.io.output.*" %>
<%
  File file ;
  int maxFileSize = 5000 * 1024;
```

```
int maxMemSize = 5000 * 1024;
ServletContext context = pageContext.getServletContext();
String filePath = context.getInitParameter("file-upload");
// Verify the content type
String contentType = request.getContentType();
if ((contentType.indexOf("multipart/form-data") >= 0)) {
  DiskFileItemFactory factory = new DiskFileItemFactory();
  // maximum size that will be stored in memory
  factory.setSizeThreshold(maxMemSize);
  // Location to save data that is larger than maxMemSize.
  factory.setRepository(new File("c:\\temp"));
  // Create a new file upload handler
  ServletFileUpload upload = new ServletFileUpload(factory);
  // maximum file size to be uploaded.
  upload.setSizeMax( maxFileSize );
  try{
    // Parse the request to get file items.
    List fileItems = upload.parseRequest(request);
    // Process the uploaded file items
    Iterator i = fileItems.iterator();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>JSP File upload</title>");
    out.println("</head>");
    out.println("<body>");
    while ( i.hasNext () )
    {
      FileItem fi = (FileItem)i.next();
      if ( !fi.isFormField () )
      {
      // Get the uploaded file parameters
      String fieldName = fi.getFieldName();
      String fileName = fi.getName();
      boolean isInMemory = fi.isInMemory();
      long sizeInBytes = fi.getSize();
      // Write the file
      if( fileName.lastIndexOf("\\") >= 0 ){
      file = new File( filePath +
      fileName.substring( fileName.lastIndexOf("\\"))) ;
      }else{
      file = new File( filePath +
      fileName.substring(fileName.lastIndexOf("\\")+1)) ;
      }
      fi.write( file ) ;
      out.println("Uploaded Filename: " + filePath +
```

```
          fileName + "<br>");
          }
        }
        out.println("</body>");
        out.println("</html>");
      }catch(Exception ex) {
        System.out.println(ex);
      }
    }else{
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet upload</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<p>No file uploaded</p>");
      out.println("</body>");
      out.println("</html>");
    }
%>
```

现在尝试用你在上面创建的 HTML 表单来上传文件。当你运行 `http://localhost:8080/UploadFile.htm`，它将显示如下结果，它将帮助你从本地电脑上上传任意文件。

File Upload:

Select a file to upload:

Browse...

Upload File

如果你的 JSP script 运行良好，你的文件应该上传到 c:\apache-tomcat-5.5.29\webapps\data\ directory 中。

JSP – 处理日期

使用 JSP 的一个最重要的优点是，你可以使用核心 Java 中所有有效的方法。本教程将教你使用 Java 提供的 Date 类，它在 java.util 包是有效的，这个类封装了当前的日期和时间。

这个 Date 类支持两种构造函数。第一种构造函数是初始化当前日期和时间的对象。

```
Date( )
```

下面的构造函数是设置一个参数，该参数是从 1970 年 1 月 1 日凌晨 0 点开始至今的毫秒数。

```
Date(long millisec)
```

一旦你有一个有效的 Date 对象，你可以调用以下任何支持的方法实现日期：

SN	方法描述
1	boolean after(Date date) 如果调用的 Date 对象包含的日期晚于指定的日期，则返回 true，否则返回 false。
2	boolean before(Date date) 如果调用的 Date 对象包含的日期早于指定的日期，则返回 true，否则返回 false。
3	Object clone( ) 重复调用的 Date 对象。
4	int compareTo(Date date) 比较调用的 Date 对象与 date 的值。如果值是相等的，则返回 0。如果调用的 Date 对象比 date 更早，则返回一个负数。如果调用 Date 对象是晚于 date 的，则返回一个正数。
5	int compareTo(Object obj)

	如果 obj 是 Date 类，则操作与 compareTo(Date) 是同一个，否则抛出 ClassCastException 异常。
6	boolean equals(Object date) 如果调用的 Date 对象与指定的日期有相同的时间和日期，则返回 true，否则返回 false。
7	long getTime( ) 返回从 1970 年 1 月 1 日凌晨 0 点开始至今的毫秒数。
8	int hashCode( ) 返回调用对象的哈希编码
9	void setTime(long time) 由指定的时间设置时间和日期，它表示从 1970 年 1 月 1 日凌晨 0 点开始到指定时间的毫秒数。
10	String toString( ) 转换调用的 Date 对象到 string 类型，并且返回该结果。

得到当前日期 & 时间

在 JSP 程序中，很容易得到当前日期和时间。你可以使用一个简单的 Date 对象调用 toString() 方法来输出当前的日期和时间，如下所示：

```
<%@ page import="java.io.*,java.util.*, javax.servlet.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<center>
<h1>Display Current Date & Time</h1>
</center>
<%
```

```
   Date date = new Date();
   out.print( "<h2 align=\"center\">" +date.toString()+"</h2>");
%>
</body>
</html>
```

现在我们将保持 CurrentDate.jsp 中代码，然后使用 URL `http://localhost:8080/CurrentDate.jsp` 来调用此 JSP。将产生如下结果：

Display Current Date & Time

Mon Jun 21 21:46:49 GMT+04:00 2010

尝试刷新 URL `http://localhost:8080/CurrentDate.jsp` ，你将会发现每一次刷新都会有几秒钟的区别。

日期比较

正如上面提到的，你可以使用所有有效的 Java 方法在你的 JavaScript 中。如果你需要比较两个日期，下面是方法：

- 你可以用 getTime() 方法分别获得这两个对象从 1970 年 1 月 1 日凌晨 0 点开始至今的毫秒数，然后比较这两个值。
- 你可以使用方法 before()，after() 和 equals()。因为每月的 12 日在 18 日之前，例如，new Date(99,2,12).(new Date(99,2,18))，返回 true。
- 你可以使用 compareTo() 方法，它由 Comparable 接口定义并且有 Date 实现。

用 SimpleDateFormat 实现日期格式化

SimpleDateFormat 是用对语言环境敏感的方式来格式化和解析日期的具体类。SimpleDateFormat 允许你对日期时间格式来选择任何用户定义的模式开始。

让我们修改上面的例子，如下所示：

```
<%@ page import="java.io.*,java.util.*" %>
<%@ page import="javax.servlet.*,java.text.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<center>
<h1>Display Current Date & Time</h1>
</center>
<%
  Date dNow = new Date( );
  SimpleDateFormat ft =
  new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
  out.print( "<h2 align=\"center\">" + ft.format(dNow) + "</h2>");
%>
</body>
</html>
```

再次编译上述 Servlet，然后使用 URL `http://localhost:8080/CurrentDate` 来调用此 Servlet。将产生如下结果：

Display Current Date & Time

Mon 2010.06.21 at 10:06:44 PM GMT+04:00

Simple DateFormat 格式化代码

指定时间格式使用一个时间模式字符串。在这个模式中,所有 ASCII 字母被保留为模式字母，它们被定义为如下:

字符	描述	例子
G	时代指示符	AD
y	四位数的某年	2001
M	一年中的某月	July or 07
d	一月中的某日	10
h	A.M./P.M. (1~12)的某小时	12
H	一天 (0~23)中的某小时	22
m	一小时中的某分钟	30
s	一分钟中的某秒	55
S	毫秒	234
E	一周中的某天	Tuesday
D	一年中的某天	360
F	一月中的一周的某天	2 (second Wed. in July)
w	一年中的某周	40
W	一月中的某周	1
a	A.M./P.M.标记	PM
k	一天(1~24)中的某小时	24
K	A.M./P.M. (0~11)的某小时	10
z	时区	Eastern Standard Time
'	消逝的文本	Delimiter
"	单引号	`

对于一个用不变且有效的方法来操作日期的完整清单，你可以参考标准的 Java 文件。

JSP – 页面重定向

页面重定向通常用于当一个文件移动到一个新的位置，我们需要向客户端发送到这个新 的位置，或可能是因为负载平衡，或简单随机化。

请求重定向到另一个页面的最简单的方法是使用 response 对象的 sendRedirect() 方法。以下是该方法的符号描述：

```
public void response.sendRedirect(String location)
throws IOException
```

此方法返回带有状态编码和新的页面位置的响应给浏览器。你也可以一起使用 setstatus() 和 setheader() 方法实现相同的重定向。

```
....
String site = "http://www.newpage.com" ;
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
....
```

例子

这个例子显示了如何将一个 JSP 页面重定向到另一个位置：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Page Redirection</title>
</head>
<body>
<center>
<h1>Page Redirection</h1>
</center>
<%
  // New location to be redirected
  String site = new String("http://www.photofuntoos.com");
  response.setStatus(response.SC_MOVED_TEMPORARILY);
  response.setHeader("Location", site);
%>
</body>
</html>
```

现在将上面的代码放在 PageRedirect.jsp 中，并且使用 URL `http://localhost:8080/PageRedirect.jsp` 来调用此 JSP。它将跳转到页面 http://www.photofuntoos.com。

JSP – 点击计数器

一个点击计数器告诉你关于网站某个特定页面的访问量。假设人们第一次登陆你的主页，通常你在 index.jsp 页面上设置一个点击计数器。

你可以使用 Application 隐式对象和相关方法 getAttribute() 和 setAttribute() 实现一个点击计数器。

这个对象通过其整个生命周期来表示此 JSP 页面。初始化这个对象时创建 JSP 页面，当此 JSP 页面被 jspDestroy() 方法删除时该对象也被删除。

以下是在应用层设置变量的语法：

```
application.setAttribute(String Key, Object Value);
```

你可以使用上述的方法设置点击计数器的变量或者重置相同的变量。接下来描述的是一个方法，该方法是读取先前方法设置的变量。

```
application.getAttribute(String Key);
```

每次用户访问网页，你可以读取点击计数器的当前值，增加 1 并且再次设置点击计数器作为以后使用。

例子

这个例子展示了如何使用 JSP 来统计一个特定的页面的点击量。如果你想计算你的网站点击量，那么你将不得不在所有 JSP 页面包含相同的代码。

```
<%@ page import="java.io.*,java.util.*" %>

<html>
<head>
<title>Applcation object in JSP</title>
</head>
<body>
<%
   Integer hitsCount =
     (Integer)application.getAttribute("hitCounter");
   if( hitsCount ==null || hitsCount == 0 ){
      /* First visit */
      out.println("Welcome to my website!");
      hitsCount = 1;
   }else{
      /* return visit */
      out.println("Welcome back to my website!");
      hitsCount += 1;
   }
   application.setAttribute("hitCounter", hitsCount);
%>
```

```
<center>
<p>Total number of visits: <%= hitsCount%></p>
</center>
</body>
</html>
```

现在将上面的代码放在 main.jsp 中，并且使用 URL `http://localhost:8080/main.jsp` 来调用此 JSP。每当你刷新该页面时，这将显示的点击计数器值会增加。你可以尝试使用不同的浏览器访问该网页，你会发现每次点击计数器将增加，显示的结果如下：

Welcome back to my website!

Total number of visits: 12

计数器重置

如果你重新启动你的应用程序如 Web 服务器，这将重置你的应用程序变量，点击计数器将重置为零。为了避免这种损失，你可以用下面专业的方法实现点击计数器：

- 定义一个带有单一计数的数据库表，我们叫做点击量。设置它的值为 0。
- 每次点击，读取该表得到点击量的值。
- 点击量加 1，更新该表中的值。
- 显示点击计数器的新值作为总页面的点击量。
- 如果你想计算所有页面的点击量，对所有的页面实现上面的逻辑。

JSP – 自动刷新

细想一个显示在线比赛分数、股市状态或当前交易额的网页。对于所有这种类型的网页，你需要通过浏览器中的更新或者重新载入按钮定期的刷新网页。

通过提供一个在给定的间距后自动刷新网页的机制，可以使 JSP 更加容易运行。

刷新网页最简单的方法就是使用 request 对象的 setIntHeader() 方法。下面是这个方法的符号描述：

```
public void setIntHeader(String header, int headerValue)
```

此方法将标题“刷新”以及以秒为单位的时间间隔的整数值返回给浏览器。

自动页面刷新例子

下面的例子使用 setIntHeader() 方法设置 Refresh 标头来模拟数字计时器：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Auto Refresh Header Example</title>
</head>
<body>
<center>
<h2>Auto Refresh Header Example</h2>
<%
  // Set refresh, autoload time as 5 seconds
  response.setIntHeader("Refresh", 5);
  // Get current time
  Calendar calendar = new GregorianCalendar();
  String am_pm;
  int hour = calendar.get(Calendar.HOUR);
  int minute = calendar.get(Calendar.MINUTE);
  int second = calendar.get(Calendar.SECOND);
  if(calendar.get(Calendar.AM_PM) == 0)
    am_pm = "AM";
  else
    am_pm = "PM";
  String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
  out.println("Crrent Time: " + CT + "\n");
%>
</center>
</body>
</html>
```

现在将上面的代码放在 main.jsp 中，并且访问它。它将在之后每隔 5 s 显示当前系统时间。只运行此 JSP 页面，等待一会可以看到如下结果：

Auto Refresh Header Example

Current Time is: 9:44:50 PM

为了变得与其他方法更加合适，你可以用同样的形式尝试更多上面列出的方法。

JSP – 发送电子邮件

用一个 JSP 页面发送邮件虽然足够简单，但是开始你应该有 JavaMail API，而且在你的电脑上安装 Java Activation Framework (JAF)。

- 你可以从Java官方网站下载 JavaMail 最新的版本 JavaMail (Version 1.2) (http://java.sun.com/products/javamail/) 。
- 你可以从 Java 官方网站下载 JavaBeans Activation Framework 最新的版本 JAF (Version 1.0.2) (http://www.oracle.com/technetwork/java/javase/jaf-136260.html) 。

下载和解压这些文件，对于所有的应用程序，在新建的顶层目录中你将可以发现许多 jar 文件。你需要在 CLASSPATH 中添加 mail.jar 和 activation.jar 文件。

发送一个简单的电子邮件

给出一个简单的例子，从你的机器上发送一个简单的邮件。假设你的**本地主机**连接到互联网，能够足以发送一封电子邮件。同时确保所有 jar 文件从 Java Email API 包到 JAF 包在 CLASSPATH 都是可用的。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
  String result;
  // Recipient's email ID needs to be mentioned.
  String to = "abcd@gmail.com";

  // Sender's email ID needs to be mentioned
  String from = "mcmohd@gmail.com";

  // Assuming you are sending email from localhost
  String host = "localhost";

  // Get system properties object
  Properties properties = System.getProperties();

  // Setup mail server
  properties.setProperty("mail.smtp.host", host);

  // Get the default Session object.
  Session mailSession = Session.getDefaultInstance(properties);

  try{
    // Create a default MimeMessage object.
    MimeMessage message = new MimeMessage(mailSession);
    // Set From: header field of the header.
    message.setFrom(new InternetAddress(from));
    // Set To: header field of the header.
    message.addRecipient(Message.RecipientType.TO,
```

```
                        new InternetAddress(to));
      // Set Subject: header field
      message.setSubject("This is the Subject Line!");
      // Now set the actual message
      message.setText("This is actual message");
      // Send message
      Transport.send(message);
      result = "Sent message successfully....";
    }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
    }
%>
<html>
<head>
<title>Send Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

现在将上面的代码放在 SendEmail.jsp 文件，并且使用 URL `http://localhost:8080/SendEmail.jsp` 来调用此 JSP，将电子邮件发送到给定的邮箱 ID 为 abcd@gmail.com 中，显示的响应结果如下所示。

Send Email using JSP

```
Result: Sent message successfully....
```

如果你想给多个收件人发送邮件，下面的方法可用于发送邮件给指定的多个邮箱 IDs：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

下面是对参数的描述：

- **type**： 可设置为 TO，CC 或者 BCC。其中，CC 表示 Carbon Copy，BCC 表示 Black Carbon Copy。例如，*Message.RecipientType.TO*。
- **addresses**: 它是邮箱 ID 的数组。你能使用 InternetAddress() 方法，同时指定邮箱 IDs。

发送 HTML 邮件

给出一个简单的例子，从你的机器上发送一个网页邮件。假设你的**本地主机**连接到互联网，能够足以发送一封电子邮件。同时确保所有 jar 文件从 Java Email API 包到 JAF 包在 CLASSPATH 都是可用的。

这个例子与之前给出的例子非常相似，除了在这个例子中我们使用 setContent() 方法设置第二个参数为“text/html”，用来指定网页内容包含在这个信息中。

使用这个例子，你可以发送与你喜欢的网页内容一样大的邮件。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
  String result;
  // Recipient's email ID needs to be mentioned.
  String to = "abcd@gmail.com";

  // Sender's email ID needs to be mentioned
  String from = "mcmohd@gmail.com";

  // Assuming you are sending email from localhost
  String host = "localhost";

  // Get system properties object
  Properties properties = System.getProperties();

  // Setup mail server
  properties.setProperty("mail.smtp.host", host);

  // Get the default Session object.
  Session mailSession = Session.getDefaultInstance(properties);

  try{
    // Create a default MimeMessage object.
    MimeMessage message = new MimeMessage(mailSession);
```

```
      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));
      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO,
                           new InternetAddress(to));
      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Send the actual HTML message, as big as you like
      message.setContent("<h1>This is actual message</h1>",
                         "text/html" );
      // Send message
      Transport.send(message);
      result = "Sent message successfully....";
    }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
    }
%>
<html>
<head>
<title>Send HTML Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

现在可以使用上述 JSP 发送网页信息给指定的邮箱 ID。

在电子邮件中发送附件：

给定一个例子，从你的机器上发送一个带附件的邮箱：

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // Recipient's email ID needs to be mentioned.
   String to = "abcd@gmail.com";

   // Sender's email ID needs to be mentioned
   String from = "mcmohd@gmail.com";

   // Assuming you are sending email from localhost
   String host = "localhost";

   // Get system properties object
   Properties properties = System.getProperties();

   // Setup mail server
```

```
   properties.setProperty("mail.smtp.host", host);

   // Get the default Session object.
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // Create a default MimeMessage object.
      MimeMessage message = new MimeMessage(mailSession);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO,
                           new InternetAddress(to));

      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Create the message part
      BodyPart messageBodyPart = new MimeBodyPart();

      // Fill the message
      messageBodyPart.setText("This is message body");

      // Create a multipar message
      Multipart multipart = new MimeMultipart();

      // Set text message part
      multipart.addBodyPart(messageBodyPart);

      // Part two is attachment
      messageBodyPart = new MimeBodyPart();
      String filename = "file.txt";
      DataSource source = new FileDataSource(filename);
      messageBodyPart.setDataHandler(new DataHandler(source));
      messageBodyPart.setFileName(filename);
      multipart.addBodyPart(messageBodyPart);

      // Send the complete message parts
      message.setContent(multipart );

      // Send message
      Transport.send(message);
      String title = "Send Email";
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>
<html>
<head>
<title>Send Attachement Email using JSP</title>
</head>
<body>
<center>
<h1>Send Attachement Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
```

```
</p>
</body>
</html>
```

现在可以运行上述 JSP 发送一个文件作为信息的附件给指定邮箱 ID。

用户身份验证部分

为了身份验证的目的，如果需要提供用户 ID 和密码给邮箱服务器，你可以设置这些属性如下：

```
props.setProperty("mail.user", "myuser");
 props.setProperty("mail.password", "mypwd");
```

邮件发送机制的设置和上述的解释一致。

用表单发送邮件

你可以使用网页表单接受邮件参数，然后你可以使用 request 对象获得如下的所有信息：

```
String to = request.getParameter("to");
String from = request.getParameter("from");
String subject = request.getParameter("subject");
String messageText = request.getParameter("body");
```

一旦你有所有信息，你可以使用以上提到的代码发送邮件。

2

JSP 高级教程

JSP – 标准标签库

JSP 标准标签库（JSTL）是一组有效的 JSP 标签，它封装了很多 JSP 应用程序的核心功能。

JSTL 支持普通、结构任务，如迭代和条件，操作 XML 文件标签，国际化标签，和 SQL 标签。它也提供了一个整合现有 JSTL 标签和自定义标签的框架。

JSTL 标签根据它们的功能进行分类，划分到 JSTL 标签库中，当创建一个 JSP 页面时，可以使用这些标签：

- 核心标签库
- 格式标签库
- SQL 标签库
- XML 标签库
- JSTL 函数标签库

安装 JSTL 标签库

如果你使用的是 Apache Tomcat 容器，然后遵循以下两个简单的步骤：

- 从 Apache Standard Taglib，下载二进制分布，打开压缩文件。
- 为了从 Jakarta Taglibs 分布中使用标准标签库，简单的复制分布库目录中的 JAR 文件到你的应用程序的 webapps\ROOT\WEB-INF\lib 目录中。

为了使用所有的库，你必须在要使用该库的每个 JSP 页面的头部中引入指令。

核心标签库

核心标签库是被广泛使用的 JSTL 标签库。下面是在 JSP 页面中引入核心标签库时需要使用的指令：

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

接下来的是核心标签库的标签：

<

table class="table table-bordered"> 标签描述 <c:out > (jsp-tag/c-out.md) 类似于 java 表达式<%= ... >，但是表达式。 <c:set > (jsp-tag/c-set.md) 在某个范围内设置表达式的值。 <c:remove > (jsp-tag/c-remove.md) 删除一个域变量（从一个特殊的被指定的范围）。 <c:catch> (jsp-tag/c-catch.md) 抛出任何发生在它的主体中的异常，并且有选择的公开它。 <c:if> (jsp-tag/c-if.md) 简单的条件标签，如果提供的条件是 true，则执行标签体的内容。 <c:choose> (jsp-tag/c-choose.md) 简单的条件标签，用标签< when>和< otherwise>建立一个互斥条件操作的上下文。 <c:when> (jsp-tag/c-choose.md) < choose>的子标签，如果它的条件为“true”，则运行标签体的内容。 <c:otherwise > (jsp-tag/c-choose.md) 的子标签，它出现在< when>标签之后，只有当先前的条件结果为“false”运行它。 <c:import> (jsp-tag/c-import.md) 检索绝对或相对的 URL 并且显示它的内容到其他的页面，在“var”中的一个 String 类型，或者在“varReader”中的一个 Reader 类型。 <c:forEach > (jsp-tag/c-forEach.md) 基本的迭代标签，接受多种不同的集合类型，支持子集和其他的功能。 <c:forTokens> (jsp-tag/c-forEach.md) 迭代使用分隔符，分隔提供的定界符。 <c:param> (jsp-tag/c-param.md) 添加一个参数到包含“import”标签的 URL。 <c:redirect > (jsp-tag/c-redirect.md) 重新定向到的一个新的 URL。 <c:url> (jsp-tag/c-url.md) 创建一个带有选项查询参数的 URL。

格式标签库

JSTL 的格式标签库是用于格式化和显示国际化网址的文本、日期、时间和数字。下面是在 JSP 页面中引入格式标签库时需要使用的指令：

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

接下来的是格式标签库的标签：

标签	描述
<fmt:formatNumber> (jsp-tag/fmt-formatNumber.md)	用特定的精度或格式呈现数值。
<fmt:parseNumber> (jsp-tag/fmt-parseNumber.md)	将一个表示数值、货币或百分比格式化字符串解析成数值对象。
<fmt:formatDate> (jsp-tag/fmt-formatDate.md)	将日期和/或时间对象格式化成指定的风格和模式
<fmt:parseDate> (jsp-tag/fmt-parseDate.md)	将用字符串表示的日期和/或时间解析成日期对象。
<fmt:bundle> (jsp-tag/fmt-bundle.md)	使用它的标签体来加载一个资源包。
<fmt:setLocale> (jsp-tag/fmt-setLocale.md)	在区域配置变量中存储指定的区域。
<fmt:setBundle> (jsp-tag/fmt-setBundle.md)	加载一个资源包并且将其存储在指定作用域或者资源包配置变量中。

<fmt:timeZone> (jsp-tag/fmt-timeZone.md)	在标签体的内容中，对指定的时区进行时间格式化和解析操作。
<fmt:setTimeZone> (jsp-tag/fmt-setTimeZone.md)	将指定的时区存储在时区配置变量中。
<fmt:message> (jsp-tag/fmt-message.md)	显示国际化的信息。
<fmt:requestEncoding> (jsp-tag/fmt-requestEncoding.md)	设置请求中字符的编码格式。

SQL 标签库

JSTL 的 SQL 标签库提供了标签给交互的关系数据库(RDBMSs)如 Oracle、mySQL 或 Microsoft SQL Server。

下面是在 JSP 页面中引入格式标签库时需要使用的指令：

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

接下来的是格式标签库的标签：

标签	描述
<sql:setDataSource> (jsp-tag/sql-setDataSource.md)	创建一个简单的只适合原型的数据源。
<sql:query> (jsp-tag/sql-query.md)	在它的标签体中或通过 SQL 属性，执行定义的 SQL 查询操作。
<sql:update> (jsp-tag/sql-update.md)	在它的标签体中或通过 SQL 属性，执行定义的 SQL 更新操作。
<sql:param> (jsp-tag/sql-param.md)	将 SQL 语句中的一个参数设置为指定的值。
<sql:dateParam> (jsp-tag/sql-dateParam.md)	将 SQL 语句中的一个参数设置为指定的 java.util.Date 值。
<sql:transaction > (jsp-tag/sql-transaction.md)	为嵌套数据库操作元素提供了一个共享连接，建立执行所有语句组成一个事务。

XML 标签库

JSTL 的 XML 标签库提供了创建和操纵 XML 文档的 JSP-centric 方式。下面是在 JSP 页面中引入 XML 标签库时需要使用的指令：

对于交互的 XML 数据，JSTL 的 XML 标签库有自定义标签。它包括解析 XML、转换XML 数据和基于 XPath 语言表达式的流程控制。

```
<%@ taglib prefix="x"
        uri="http://java.sun.com/jsp/jstl/xml" %>
```

在你处理这个例子之前，你应该需要复制以下两个 XML 和 XPath 相关的库到你的 \lib 中：

- XercesImpl.jar: 从 http://www.apache.org/dist/xerces/j/ 下载
- xalan.jar: 从 http://xml.apache.org/xalan-j/index.html 下载

接下来的是格式标签库的标签：

Following is the list of XML JSTL Tags:

标签	描述
<x:out> (jsp-tag/x-out.md)	类似于 java 表达式 <%= ... >，但是 XPath 表达式。
<x:parse> (jsp-tag/x-parse.md)	要么通过属性要么在标签体中，用于解析特定的XML数据。
<x:set > (jsp-tag/x-set.md)	设置 XPath 表达式的变量值。
<x:if > (jsp-tag/x-if.md)	计算测试的XPath表达式，如果它是 true，就执行它的标签体中的内容。如果测试条件是 false，忽略这个标签体中内容。
<x:forEach> (jsp-tag/x-forEach.md)	在 XML 文件中循环顶点。
<x:choose> (jsp-tag/x-choose.md)	简单的条件标签，用标签< when>和< otherwise>建立一个互斥条件操作的上下文。
<x:when > (jsp-tag/x-choose.md)	< choose>的子标签，如果它的条件为“true”，则运行标签体的内容。
<x:otherwise > (jsp-tag/x-choose.md)	< choose>的子标签，它出现在< when>标签之后，只有当先前的条件结果为“false”运行它。
<x:transform > (jsp-tag/x-transform.md)	应用一个 XSL 转换到一个 XML 文件中。
<x:param > (jsp-tag/x-param.md)	在 XSLT 样式表中使用转换标签设置参数。

JSTL 函数标签库

JSTL 函数标签库包含了大量的标准函数，其中大多数都是御用字符串处理的函数。下面是在 JSP 页面中引入 JSTL 函数标签库时需要使用的指令：

```
<%@ taglib prefix="fn"
        uri="http://java.sun.com/jsp/jstl/functions" %>
```

接下来的是 JSTL 函数标签库的标签：

函数	描述
fn:contains() (jsp-tag/fn-contains.md)	判断输入的字符串是否包含指定的子字符串。
fn:containsIgnoreCase() (jsp-tag/fn-containsIgnoreCase.md)	判断输入的字符串是否包含指定的子字符串，判断时忽略大小写。
fn:endsWith() (jsp-tag/fn-endsWith.md)	判断输入的字符串是否以指定的字符串作为后缀。
fn:escapeXml() (jsp-tag/fn-escapeXml.md)	忽略字符串中的 XML 标签。
fn:indexOf() (jsp-tag/fn-indexOf.md)	返回字符串在指定的字符串第一次出现的位置。
fn:join() (jsp-tag/fn-join.md)	连接数组中所有的元素到一个字符串中。
fn:length() (jsp-tag/fn-length.md)	返回集合中的项数或者字符串的字符数。
fn:replace() (jsp-tag/fn-replace.md)	返回用输入字符串替换到给定的字符串所有出现的位置中所得到的新字符串。
fn:split() (jsp-tag/fn-split.md)	将一个字符串分割成子字符串数组。
fn:startsWith() (jsp-tag/fn-startsWith.md)	判断输入的字符串是否以指定的字符串作为开头。
fn:substring() (jsp-tag/fn-substring.md)	返回字符串的子串
fn:substringAfter() (jsp-tag/fn-substringAfter.md)	返回在指定子字符串之前的字符串的子串
fn:substringBefore() (jsp-tag/fn-substringBefore.md)	返回在指定子字符串之后的字符串的子串
fn:toLowerCase() (jsp-tag/fn-toLowerCase.md)	转换字符串中的所有字符为小写字符
fn:toUpperCase() (jsp-tag/fn-toUpperCase.md)	转换字符串中的所有字符为大写字符
fn:trim() (jsp-tag/fn-trim.md)	去除字符串两端的空格

JSP – 访问数据库

该教材假定你已经完全理解了 JDBC 应用程序的工作原理。在通过 JSP 访问数据库之前，确保你的数据库有适合的 JDBC 环境设置。

对于如何用 JDBC 和环境设置访问数据库的更多细节，你可以阅读我们的 JDBC 教程 (http://wiki.jikexueyuan.com/project/jdbc/) 。

为了开始基本的概念，我们先创建简单的表和新的记录在该表中，如下所示：

创建表

为了在 EMP 数据库中创建 Employees 表，用下面的步骤：

步骤 1

打开一个**命令提示符**并且改变安装路径，如下所示：

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

步骤 2

登陆数据库，如下所示：

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

步骤 3

在 TEST 数据库中创建 Employee 表，如下所示：

```
mysql> use TEST;
mysql> create table Employees
   (
    id int not null,
```

```
    age int not null,
    first varchar (255),
    last varchar (255)
   );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

创建数据记录

最后你在 Employee 表中创建一些记录，如下所示：

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)
mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)
mysql>
```

查询操作

下面的例子显示了我们如何用 JSTL 在 JSP 程序中执行 SQL 查询语句：

```
<%@ page import="java.io.,java.util.,java.sql."%>
<%@ page import="javax.servlet.http.,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>


<html>
<head>
<title>SELECT Operation</title>
</head>
<body>


<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/TEST"
    user="root"  password="pass123"/>


<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>
```

```
<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>



</body>
</html>
```

现在尝试访问上述的 JSP 页面，显示的结果如下所示：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28

插入操作

下面的例子显示了我们如何用 JSTL 在 JSP 程序中执行 SQL 插入语句：

```
<%@ page import="java.io.,java.util.,java.sql.*"%>
<%@ page import="javax.servlet.http.,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>



<html>
<head>
<title>JINSERT Operation</title>
</head>
<body>



<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/TEST"
    user="root"  password="pass123"/>
```

```
<sql:update dataSource="${snapshot}" var="result">
INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');
</sql:update>


<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>


<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>


</body>
</html>
```

现在尝试访问上述的 JSP 页面，显示的结果如下所示：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28
104	Nuha	Ali	2

删除操作

下面的例子显示了我们如何用 JSTL 在 JSP 程序中执行 SQL 删除语句：

```
<%@ page import="java.io.,java.util.,java.sql."%>
<%@ page import="javax.servlet.http.,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
```

```
<html>
<head>
<title>DELETE Operation</title>
</head>
<body>


<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/TEST"
    user="root"  password="pass123"/>


<c:set var="empId" value="103"/>


<sql:update dataSource="${snapshot}" var="count">
 DELETE FROM Employees WHERE Id = ?
 <sql:param value="${empId}" />
</sql:update>


<sql:query dataSource="${snapshot}" var="result">
  SELECT * from Employees;
</sql:query>


<table border="1" width="100%">
<tr>
  <th>Emp ID</th>
  <th>First Name</th>
  <th>Last Name</th>
  <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
  <td><c:out value="${row.id}"/></td>
  <td><c:out value="${row.first}"/></td>
  <td><c:out value="${row.last}"/></td>
  <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>


</body>
</html>
```

现在尝试访问上述的 JSP 页面，显示的结果如下所示：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30

更新操作

下面的例子显示了我们如何用 JSTL 在 JSP 程序中执行 SQL 更新语句：

```
<%@ page import="java.io.,java.util.,java.sql."%>
<%@ page import="javax.servlet.http.,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>


<html>
<head>
<title>DELETE Operation</title>
</head>
<body>


<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/TEST"
    user="root"  password="pass123"/>


<c:set var="empId" value="102"/>


<sql:update dataSource="${snapshot}" var="count">
  UPDATE Employees SET last = 'Ali'
  <sql:param value="${empId}" />
</sql:update>


<sql:query dataSource="${snapshot}" var="result">
   SELECT * from Employees;
</sql:query>


<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>
```

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

现在尝试访问上述的 JSP 页面，显示的结果如下所示：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Ali	30

JSP – XML 数据

当你通过 HTTP 发送 XML 数据时，使用 JSP 处理传入和传出的XML文件是有意义的，例如 RSS 文档。作为 XML 文档仅仅是一堆文字，通过 JSP 创建一个 XML 文档并不比创建一个 HTML 文档困难。

从 JSP 发送 XML

你可以用 JSPs 发送 HTML 的同样的方式来发送 XML 内容。唯一的区别在于，你必须设置页面的内容类型为 text/xml。使用 < % @page % > 标签来设置内容类型，如下所示：

```
<%@ page contentType="text/xml" %>
```

下面是一个简单的例子将 XML 内容发送到浏览器：

```
<%@ page contentType="text/xml" %>
<books>
  <book>
    <name>Padam History</name>
    <author>ZARA</author>
    <price>100</price>
  </book>
</books>
```

尝试使用不同的浏览器访问上面的 XML，用来显示上面的 XML 的文档树。

在 JSP 上处理 XML

在使用 JSP 处理 XML 之前，你需要复制以下两个 XML 和 XPath 相关的库到你的 \lib 中：

- XercesImpl.jar: 从 http://www.apache.org/dist/xerces/j/ 下载
- xalan.jar: 从 http://xml.apache.org/xalan-j/index.html 下载

让我们把以下内容放入 books.xml 文件：

```
<books>
<book>
  <name>Padam History</name>
  <author>ZARA</author>
  <price>100</price>
</book>
```

```
<book>
  <name>Great Mistry</name>
  <author>NUHA</author>
  <price>2000</price>
</book>
</books>
```

现在尝试写以下 main.jsp，将其放在同一个目录下：

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



<html>
<head>
  <title>JSTL x:parse Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:import var="bookInfo" url="http://localhost:8080/books.xml"/>



<x:parse xml="${bookInfo}" var="output"/>
<b>The title of the first book is</b>:
<x:out select="$output/books/book[1]/name" />
<br>
<b>The price of the second book</b>:
<x:out select="$output/books/book[2]/price" />



</body>
</html>
```

现在尝试使用 `http://localhost:8080/main.jsp` 访问上述 JSP，产生的结果如下所示：

```
Books Info:

The title of the first book is:Padam History

The price of the second book: 2000
```

格式化 XML 和 JSP

考虑下面的 XSLT 样式表 style.xsl：

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform" version="1.0">



<xsl:output method="html" indent="yes"/>



<xsl:template match="/">
  <html>
  <body>
   <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>



<xsl:template match="books">
  <table border="1" width="100%">
    <xsl:for-each select="book">
      <tr>
        <td>
          <i><xsl:value-of select="name"/></i>
        </td>
        <td>
          <xsl:value-of select="author"/>
        </td>
        <td>
          <xsl:value-of select="price"/>
        </td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

现在考虑下面的 JSP 文件：

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



<html>
<head>
  <title>JSTL x:transform Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:set var="xmltext">
  <books>
    <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
    </book>
    <book>
      <name>Great Mistry</name>
      <author>NUHA</author>
```

```
    <price>2000</price>
   </book>
  </books>
</c:set>


<c:import url="http://localhost:8080/style.xsl" var="xslt"/>
<x:transform xml="${xmltext}" xslt="${xslt}"/>


</body>
</html>
```

产生的结果如下所示：

Books Info:

Padam History	ZARA	100
Great Mistry	NUHA	2000

关于用 JSTL 处理 XML 更多的细节，你可以查询 JSP Standard Tag Library (页 0)。

JSP - JavaBeans

JavaBean 是在编写 Java 时专门创建的 Java 类，根据 JavaBean API 规范进行编码 。 以下是区分 JavaBean 和其他 Java 类的特有的特征：

- 它提供了一个默认的无参数构造函数。
- 它应该是可序列化的，实现 serializable 接口。
- 它可能有大量可以读或写的属性。
- 它可能有大量“getter”和“setter”方法的属性。

JavaBean 属性

JavaBean 属性是一个命名属性，这个属性是用户可以访问的对象。属性可以是任何 Java 数据类型，包括自定义的类。

JavaBean 属性可以读、写、只读或只写。JavaBean 属性是在 JavaBean 实现类中通过两种方法访问的：

方法	描述
get**PropertyName**()	例如，如果属性名称是 firstName，你的方法名称应该是 getFirstName()，它可以读该属性。该方法叫做访问器。
set**PropertyName**()	例如，如果属性名称是 firstName，你的方法名称应该是 setFirstName()，它可以写该属性。该方法叫做赋值方法。

一个只读属性只会有一个 get**PropertyName**() 方法，而一个只写属性将只有一个 set**PropertyName**() 方法。

JavaBeans 例子

考虑一个带有一些属性的 student 类：

```
package com.tutorialspoint;
public class StudentsBean implements java.io.Serializable
{
  private String firstName = null;
  private String lastName = null;
  private int age = 0;
  public StudentsBean() {
  }
```

```
  public String getFirstName(){
    return firstName;
  }
  public String getLastName(){
    return lastName;
  }
  public int getAge(){
    return age;
  }
  public void setFirstName(String firstName){
    this.firstName = firstName;
  }
  public void setLastName(String lastName){
    this.lastName = lastName;
  }
  public void setAge(Integer age){
    this.age = age;
  }
}
```

访问 JavaBeans

在一个 JSP 页面使用时，useBean 操作声明一个对象。一旦声明，bean 成为脚本变量，在使用它的 JSP 页面中，它可以通过脚本元素和其他自定义标签访问。useBean 标签的完整的语法如下：

```
<jsp:useBean id="bean's name" scope="bean's scope" typeSpec/>
```

根据你的需求，这里 scope 属性值可能是页面、请求、会话或应用程序。id 属性的值可以是任何值，在同一个 JSP 页面中，只要它是一个唯一的名字与其他 useBean 声明。

下面的例子显示了其简单的用法：

```
<html>
<head>
<title>useBean Example</title>
</head>
<body>


<jsp:useBean id="date" class="java.util.Date" />
<p>The date/time is <%= date %>


</body>
</html>
```

产生的结果如下所示：

```
The date/time is Thu Sep 30 11:18:11 GST 2010
```

访问 javabean 属性

随着 < jsp:useBean…>，你可以使用 < jsp:getProperty / > 操作来访问方法和 < jsp:setProperty / > 操作来访问设置方法。这是完整的语法：

```
<jsp:useBean id="id" class="bean's class" scope="bean's scope">
   <jsp:setProperty name="bean's id" property="property name"

                  value="value"/>
   <jsp:getProperty name="bean's id" property="property name"/>
   .....
</jsp:useBean>
```

该名称属性通过 useBean 操作把先前介绍 JavaBean 的 id 引用到 JSP 页面中。该属性的属性是应该被调用 get 或 set 方法的名称。

下面是一个使用上面的语法访问数据的简单的例子：

```
<html>
<head>
<title>get and set properties Example</title>
</head>
<body>



<jsp:useBean id="students"
                  class="com.tutorialspoint.StudentsBean">
   <jsp:setProperty name="students" property="firstName"
                  value="Zara"/>
   <jsp:setProperty name="students" property="lastName"
                  value="Ali"/>
   <jsp:setProperty name="students" property="age"
                  value="10"/>
</jsp:useBean>



<p>Student First Name:
   <jsp:getProperty name="students" property="firstName"/>
</p>
<p>Student Last Name:
   <jsp:getProperty name="students" property="lastName"/>
</p>
<p>Student Age:
   <jsp:getProperty name="students" property="age"/>
</p>
```

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

让我们在 CLASSPATH 中使 StudentsBean.class 有效，并且尝试访问上面的 JSP。这将产生以下结果：

```
Student First Name: Zara


Student Last Name: Ali


Student Age: 10
```

JSP – 自定义标签

自定义标签是一种用户定义的 JSP 语言元素。当一个包含自定义标签的 JSP 页面翻译成一个 servlet 时，该标签转换为在一个对象上的操作称为标签处理程序。然后当执行 JSP 页面的 servlet 时，Web 容器调用这些操作。

扩展的 JSP 标签允许你创建新标签，你可以直接插入到一个 JavaServer Page（JSP）中，就像在先前章节中学习的内置标签。为了编写这些自定义标签，JSP 2.0 规范引入了简单的标签处理程序。

要编写一个自定义标签，你可以简单地扩展 SimpleTagSupport 类，并且重写 doTag( ) 方法，其中你可以添加你的代码来生成标签的内容。

创建 “Hello” 标签

假如你想定义一个自定义标签名为 ，你想要在以下没有标签体的方式中使用，如下所示：

```
<ex:Hello />
```

要创建一个自定义 JSP 标签，你必须首先创建一个 Java 类作为标记处理程序。因此，创建 HelloTag 类如下：

```
package com.tutorialspoint;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
public class HelloTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.println("Hello Custom Tag!");
  }
}
```

上面的代码是简单的编码，其中 doTag() 方法使用 getJspContext() 产生的当前 JspWriter 对象，并使用它发送“Hello Custom Tag!”给当前的 JspWriter 对象。

编译上面的类，并将其复制到 CLASSPATH 环境变量的有效目录中。最后创建以下标签库文件：webapps\ROOT\WEB-INF\custom.tld。

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD</short-name>
```

```
 <tag>
  <name>Hello</name>
  <tag-class>com.tutorialspoint.HelloTag</tag-class>
  <body-content>empty</body-content>
 </tag>
</taglib>
```

在我们的 JSP 程序中，使用上面定义的自定义标签 Hello，如下所示：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
 <head>
  <title>A sample custom tag</title>
 </head>
 <body>
  <ex:Hello/>
 </body>
</html>
```

尝试调用上述JSP，显示结果如下所示：

```
Hello Custom Tag!
```

访问标签体

你可以标签体中引用在标准标签体中你所见的消息。假如你想定义一个自定义标签名为 ，你想要在以下含有标签体的方式中使用，如下所示：

```
<ex:Hello>
  This is message body
</ex:Hello>
```

在上面标签的代码中进行如下修改来处理标签的主体：

```
package com.tutorialspoint;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
public class HelloTag extends SimpleTagSupport {
  StringWriter sw = new StringWriter();
  public void doTag()
    throws JspException, IOException
  {
    getJspBody().invoke(sw);
    getJspContext().getOut().println(sw.toString());
  }
}
```

在这种情况下，在调用中产生的输出是在写入与标签相关的 JspWriter 对象之前第一个捕获到的 StringWriter。因此我们需要改变 TLD 文件，如下所示：

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

现在让我们调用上述带有标签体的标签，如下所示：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello>
       This is message body
    </ex:Hello>
  </body>
</html>
```

显示结果如下所示：

```
This is message body
```

自定义标签属性

你可以使用自定义标签的各种属性。要得到一个属性值，一个自定义标签类需要实现 setter 方法，和 JavaBean 相同的 setter 方法如下所示：

```
package com.tutorialspoint;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
public class HelloTag extends SimpleTagSupport {
   private String message;
   public void setMessage(String msg) {
      this.message = msg;
   }
   StringWriter sw = new StringWriter();
   public void doTag()
```

```
    throws JspException, IOException
  {
    if (message != null) {
      /* Use message from attribute /
      JspWriter out = getJspContext().getOut();
      out.println( message );
    }
    else {
      /use message from the body */
      getJspBody().invoke(sw);
      getJspContext().getOut().println(sw.toString());
    }
  }
}
```

属性的名称是“message”，因此 setter 方法是 setMessage( )。现在让我们用 元素添加该属性到 TLD 文件中，如下所示：

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>message</name>
    </attribute>
  </tag>
</taglib>
```

现在尝试下面的带有消息属性的 JSP 页面，如下所示：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello message="This is custom tag" />
  </body>
</html>
```

显示结果如下所示：

```
This is custom tag
```

希望上面的例子对你有帮助。值得注意的是，一个属性可以包含下列属性：

属性	目的
name	名称元素定义了一个属性的名称。对于一个特定的标签，每个属性名称必须是唯一。
required	这指定如果这个属性是必需的或可选的，则指定它。对于可选的，默认为 false。
rtexprvalue	如果对于一个标签属性，运行时间表达式的值是有效的，则声明它。
type	定义了该属性的Java类类型。默认情况下它是假设为**字符串**
description	可以提供信息的描述。
fragment	如果这个属性值应该被视做一个 JspFragment，则声明它。

下面是指定一个属性的相关属性的例子：

```
.....
   <attribute>
    <name>attribute_name</name>
    <required>false</required>
    <type>java.util.Date</type>
    <fragment>false</fragment>
   </attribute>
.....
```

如果你使用的是两个属性，那么你可以修改你的 TLD 如下所示：

```
.....
   <attribute>
    <name>attribute_name1</name>
    <required>false</required>
    <type>java.util.Boolean</type>
    <fragment>false</fragment>
   </attribute>
   <attribute>
    <name>attribute_name2</name>
    <required>true</required>
    <type>java.util.Date</type>
   </attribute>
.....
```

JSP – 表达式语言

JSP 表达式语言(EL)可以方便地访问存储在 javabean 组件中的应用程序的数据。JSP EL 允许创建表达式(a)算术和(b)逻辑。在一个 JSP EL 表达式中，你可以使用整数、浮点型数字、字符串、内置的布尔常量值为 true 和 false 和 null。

简单的语法

通常，当你给 JSP 标签指定一个属性值时，你只需使用一个字符串。例如：

```
<jsp:setProperty name="box" property="perimeter" value="100"/>
```

JSP EL 允许你给表达式指定这些属性值。一个简单的 JSP EL 语法如下：

```
${expr}
```

这里的 expr 是指定表达式本身。在 JSP EL 中最常见的操作符是 . 和 []。这两个操作符允许访问 Java beans 和内置 JSP 对象的各种属性。

例如可以用一个表达式编写上面的语法 < jsp:setProperty >标签，如下：

```
<jsp:setProperty name="box" property="perimeter"
                value="${2box.width+2box.height}"/>
```

当 JSP 编译器在一个属性中看到了 $ { } 形式，它可以生成计算表达式的代码，并且替换表达式的值。

你还可以在模板文本中对一个标签使用 JSP EL 表达式。例如，把 < jsp:text > 标签简单的插入到一个 JSP 的主体内容中。在下面的 < jsp:文本> 声明中，插入 < h1>Hello JSP!< /h1> 到 JSP 输出：

```
<jsp:text>
<h1>Hello JSP!</h1>
</jsp:text>
```

你可以在< jsp:text> 标签(或任何其他标签)的标签体内引入一个 JSP EL 表达式，对属性使用 $ { } 语法。例如：

```
<jsp:text>
Box Perimeter is: ${2box.width + 2box.height}
</jsp:text>
```

EL 表达式可以使用括号组成子表达式。例如，$ {(1 + 2)* 3 } = 9，但 $ { 1 +(2 * 3)} = 7。

禁止 EL 表达式的计算，我们指定页面指令的 isELIgnored 属性如下：

```
<%@ page isELIgnored ="true|false" %>
```

这个属性的有效值是 true 和 false。如果它是 true，当它们出现在静态文本或标签属性时，EL 表达式被忽略。如果它是 false，EL 表达式都由容器进行计算。

EL 的基本操作

JSP 表达式语言(EL)支持大多数 Java 支持的算术和逻辑运算符。下面是最常用的操作符的列表：

操作	描述
.	访问bean属性或映射项
[]	访问数组或链表元素
( )	组成子表达式来修改计算顺序
+	加
–	减或负数
*	乘
/ or div	除
% or mod	模计算/余数
== or eq	等于
!= or ne	不等于
< or lt	小于
> or gt	大于
<= or le	小于等于
>= or gt	大于等于
&& or and	逻辑与
|| or or	逻辑或
! or not	一元布尔补集
empty	空的变量值

JSP EL 中的函数

JSP EL 同样允许使用函数表达式。这些功能必须定义在自定义标签库中。一个函数利用下面的语法使用：

```
${ns:func(param1, param2, ...)}
```

其中，ns 是函数的命名空间，func 是函数名，param1 是第一个参数值。例如，函数 fn:length，它是 JSTL 库的一部分，可以使用它得到一个字符串的长度，如下所示。

```
${fn:length("Get my length")}
```

要从(标准或自定义)标签库中使用一个函数，你必须在你的服务器上安装库，而且必须在 JSP 中使用 < taglib > 指令来引用该库，这个在 JSTL 章有解释。

JSP EL 隐式对象

JSP 表达式语言支持以下隐式对象：

隐式对象	描述
pageScope	变量范围是页面范围
requestScope	变量范围是请求范围
sessionScope	变量范围是会话范围
applicationScope	变量范围是应用范围
param	请求参数作为字符串
paramValues	请求参数作为字符串集合
header	HTTP请求标头作为字符串
headerValues	HTTP请求标头作为字符串集合
initParam	上下文初始化参数
cookie	Cookie 值
pageContext	当前页面的 JSP PageContext 对象

你可以在表达式中使用这些对象，把它们看做是变量。这里有一些明确概念的例子：

pageContext 对象

pageContext 对象允许你访问 pageContext JSP 对象。通过 pageContext 对象，你可以访问请求对象。例如，要访问请求传入的查询字符串，你可以使用下面的表达式：

```
${pageContext.request.queryString}
```

Scope 对象

sessionScope、pageScope、requestScop、applicationScope 变量在每个级别范围提供变量存储。

例如，如果你需要显式地在应用范围内访问 box 变量，你可以通过 applicationScope 变量作为 applicationScope.box 来访问。

param 和 paramValues 对象

param 和 paramValues 对象通常通过 request.getParameter 和 request.getParameterValues 方法来有效的访问参数值。

例如，使用表达式 ${param.order} 或 ${param["order"]} 来访问命名为 order 的参数。

下面是访问命名为 username 的请求参数的例子：

```
<%@ page import="java.io.,java.util." %>
<%
   String title = "Accessing Request Param";
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>${param["username"]}</p>
</div>
</body>
</html>
```

param 对象返回单个字符串值，而 paramValues 对象返回字符串数组。

header 和 headerValues 对象

header 和 headerValue 对象通常通过 request.getHeader 和 request.getHeaders 方法来有效的访问标头值。

例如，使用表达式 ${header.user-agent} 或 ${header["user-agent"]} 来访问命名得 user-agent 的标头。

下面是访问命名为 user-agent 的标头参数的例子：

```
<%@ page import="java.io.,java.util." %>
<%
   String title = "User Agent Example";
%>
```

```
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>${header["user-agent"]}</p>
</div>
</body>
</html>
```

显示的结果如下：

User Agent Example

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30
```

header 对象返回单个字符串值，而 headerValues 对象返回字符串数组。

JSP – 异常处理

当你写 JSP 代码的时候，程序员有可能会留下一个编码错误，并且它会出现在代码的任何一个部分。在你的 JSP 代码中你会有以下类型的错误：

- **检测异常：** 一个检测异常通常是一个用户错误或者是一个有程序员无法预见的错误引起的异常。例如，如果要打开一个文件，但是无法找到该文件，这时就会出现异常。这些异常在编译时不能简单的忽略掉。
- **运行异常：** 一个运行异常可能是程序员本来可以避免的一个异常。和检测异常相反，运行异常在编译时可以被忽略。
- **错误：** 这原本不是异常，是超出用户或者程序员的控制而引起的问题。错误通常在你的代码中会被忽略，因为你对一个错误能够做的很少。例如，如果一个堆栈发生溢出，那么就会出现一个错误。在编译时他们也会被忽略。

本教程将会对你的 JSP 代码，给你一些简单而又优雅的处理运行异常和错误的方法。

使用异常对象

异常对象是 Throwable 子类的一个实例（例如，java.lang.NullPointerException），它只能在错误页面是可用的。下面是 Throwable 类中可用的重要方法的列表。

序号	方法及描述
1	public String getMessage() 返回发生异常的详细信息。这个消息是在 Throwable 构造函数里初始化的。
2	public Throwable getCause() 返回发生异常的原因，用一个 Throwable 对象表示。
3	public String toString() 返回与 getMessage() 相连接的类名。
4	public void printStackTrace() 输出 toString() 和堆栈跟踪的 System.err 的结果，错误输出流。

5	public StackTraceElement [] getStackTrace() 返回一个数组，其中包含堆栈跟踪的每一个元素。索引值为0的元素表示调用堆栈的顶部， 数组中最后一个元素在调用堆栈的底部代表方法。
6	public Throwable fillInStackTrace() 用当前的堆栈跟踪填满 Throwable 对象的堆栈跟踪，添加任何先前的堆栈跟踪信息。

JSP 会给你一个选项来指定每一个 JSP 的错误页面。不管何时页面抛出一个异常，JSP 容器都会自动的调用错误页面。

下面是 main.jsp 中一个特定错误页面的例子。为了创建一个错误页面，使用 <%@ page errorPage=”xxx”%> 指令。

```
<%@ page errorPage="ShowError.jsp" %>

<html>
<head>
  <title>Error Handling Example</title>
</head>
<body>
<%
  // Throw an exception to invoke the error page
  int x = 1;
  if (x == 1)
  {
    throw new RuntimeException("Error condition!!!");
  }
%>
</body>
</html>
```

现在你需要写一个错误处理的 JSP ShowError.jsp，下面给出了代码。注意，错误处理页面包括 <%@ page isErrorPage=”true”%> 指令。这个指令使 JSP 编译器生成异常实例变量。

```
<%@ page isErrorPage="true" %>
<html>
<head>
<title>Show Error Page</title>
</head>
<body>
<h1>Opps...</h1>
```

```
<p>Sorry, an error occurred.</p>
<p>Here is the exception stack trace: </p>
<pre>
<% exception.printStackTrace(response.getWriter()); %>
</pre>
</body>
</html>
```

现在试图访问 main.jsp，它将会生成如下结果：

```
java.lang.RuntimeException: Error condition!!!
......

Opps...
Sorry, an error occurred.

Here is the exception stack trace:
```

在错误页面使用 JSTL 标签

你可以使用 JSTL 标签来编写一个错误页面 ShowError.jsp。这个页面和上面的例子中几乎使用的是相同的逻辑，但是它有更好的结构，并且他提供了更多的信息：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page isErrorPage="true" %>
<html>
<head>
<title>Show Error Page</title>
</head>
<body>
<h1>Opps...</h1>
<table width="100%" border="1">
<tr valign="top">
<td width="40%"><b>Error:</b></td>
<td>${pageContext.exception}</td>
</tr>
<tr valign="top">
<td><b>URI:</b></td>
<td>${pageContext.errorData.requestURI}</td>
</tr>
<tr valign="top">
<td><b>Status code:</b></td>
<td>${pageContext.errorData.statusCode}</td>
</tr>
```

```
<tr valign="top">
<td><b>Stack trace:</b></td>
<td>
<c:forEach var="trace"
       items="${pageContext.exception.stackTrace}">
<p>${trace}</p>
</c:forEach>
</td>
</tr>
</table>
</body>
</html>
```

现在试图访问 main.jsp，它将会生成如下结果：

Opps...

Error:	java.lang.RuntimeException: Error condition!!!
URI:	/main.jsp
Status code:	500
Stack trace:	org.apache.jsp.main_jsp._jspService(main_jsp.java:65) org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:68) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) .....

使用 Try...Catch 块

如果你想要在同一个页面中处理错误，使用一些动作而不是释放一个错误页面，那么你可以利用 Try...catch 块。

下面显示的是如何使用 try...catch 块的一个简单的例子。让我们将下面发的代码放到 main.jsp 中：

```
<html>
<head>
  <title>Try...Catch Example</title>
</head>
<body>
<%
  try{
    int i = 1;
    i = i / 0;
    out.println("The answer is " + i);
  }
  catch (Exception e){
    out.println("An exception occurred: " + e.getMessage());
  }
%>
</body>
</html>
```

现在试图访问 main.jsp，它将会生成如下结果：

```
An exception occurred: / by zero
```

JSP – 调试

测试或者调试一个 JSP 和 servlets 总是很困难的。JSP 和 servlets 往往涉及到大量的客户端/服务器之间的交互，可能会出现错误并且很难再生成。

这里有一些提示和建议，可能会在你的调试中帮助你。

使用 System.out.println()

System.out.println() 在测试中作为一个标记很容易使用，不管某段代码是否被执行。我们也可以输出变量值。另外：

- 由于系统对象是 Java 对象核心的一部分，它可以在任何地方被使用而不需要安装额外的类。这包括 Servlets， JSP， RMI， EJB's， ordinary Beans 和 classes， 和独立的应用程序。
- 与停在断点相比较，写到 System.out 中并没有对应用程序正常的执行流产生过多的干扰，当时间至关重要时，这使得它非常有价值。

下面是使用 System.out.println() 的语法：

```
System.out.println("Debugging message");
```

下面是使用 System.out.println() 的一个简单的例子：

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>System.out.println</title></head>
<body>
<c:forEach var="counter" begin="1" end="10" step="1" >
  <c:out value="${counter-5}"/></br>
  <% System.out.println( "counter= " +
                pageContext.findAttribute("counter") ); %>
</c:forEach>
</body>
</html>
```

现在如果你要试图访问上面的 JSP，它将会在浏览器上产生以下的结果：

```
-4
-3
-2
```

```
-1
0
1
2
3
4
5
```

如果你使用的是 Tomcat，你还将发现这些行会被附加到日志目录里 stdout.log 文件的末尾。

```
counter=1
counter=2
counter=3
counter=4
counter=5
counter=6
counter=7
counter=8
counter=9
counter=10
```

这样你可以把变量和其他信息打印到系统日志中，可以分析找到问题的根本原因或者其他各种原因。

使用 JDB 记录器

J2SE 日志框架旨在为 JVM 中运行的任何类提供日志服务。所以我们可以利用这个框架来记录任何信息。

让我们使用 JDK 记录器 API 重写上面的示例：

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page import="java.util.logging.Logger" %>

<html>
<head><title>Logger.info</title></head>
<body>
<% Logger logger=Logger.getLogger(this.getClass().getName());%>

<c:forEach var="counter" begin="1" end="10" step="1" >
  <c:set var="myCount" value="${counter-5}" />
  <c:out value="${myCount}"/></br>
  <% String message = "counter="
            + pageContext.findAttribute("counter")
            + " myCount="
            + pageContext.findAttribute("myCount");
```

```
            logger.info( message );
  %>
</c:forEach>
</body>
</html>
```

这将在浏览器和 stdout.log 中生成相似的结果，但是会在 stdout.log 文件中有附加信息。在这里，我们使用记录器的 info 方法，因为我们只是为了信息的目的在记录消息。这是 stdout.log 文件的一个快照：

```
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=1 myCount=-4
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=2 myCount=-3
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=3 myCount=-2
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=4 myCount=-1
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=5 myCount=0
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=6 myCount=1
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=7 myCount=2
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=8 myCount=3
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=9 myCount=4
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=10 myCount=5
```

可以通过使用方便的函数发送各种级别的消息，如 severe()， warning()， info()， config()， fine()， finer()， 和 finest()。这里，finest() 方法可以用于记录最好的信息，severe() 方法可以用于记录严峻的消息。

你可以使用 Log4J Framework (http://wiki.jikexueyuan.com/project/log4j/) 在不同的文件中根据消息的严重水平和重要性来记录他们。

调试工具

NetBeans 是一个免费和开源的 Java 集成开发环境，支持独立的 Java 应用程序和 web 应用程序的开发，支持 JSP 和 servlet 规范，也包括一个 JSP 调试器。

NetBeans 支持以下基本的调试功能：

- 断点
- 单步调试
- 监视点

你可以参考 NetBeans 文档来了解上面的调试功能。

使用 JDB 调试器

你可以使用与你用来调试小程序和应用程序相同的 jdb 命令来调试 JSP 和 servlets 。

为了调试 JSP 和 servlets，你可以调试 sun.servlet.http.HttpServer，然后在来自浏览器的 HTTP 请求的响应里查看 HttpServer 正在执行的 JSP/servlets。这和如何调试小程序非常相似。不同不处是，在小程序里，真正的程序是在 sun.applet.AppletViewer 里调试的。

大多数调试器通过自动得知如何调试小程序来隐藏这些细节。直到它们对 JSP 做着同样的操作，你必须帮助你的调试器执行以下操作:

- 设置你的调试器的类路径，以便于找到 sun.servlet.Http-Server 和与其相关的类。
- 设置你的调试器的类路径，以便于找到你的 JSP 和支持的类，典型的是 ROOT\WEB-INF\classes。

一旦你已经正确的设置了类路径，开始调试 sun.servlet.http.HttpServer。对于一个给定的 JSP，你可以在你感兴趣的任何地方设置断点，然后通过一个 web 浏览器来发送一个请求到 HttpServer (`http://localhost:8080/JSPToDebug`)。你会看到执行会在你设置的断点处停止。

使用注释

你代码中的注释可以用不同的方法帮助你的调试过程。注释可以用于调试过程的很多其他方面。

JSP 使用 Java 命令和单线 (//...) 和多线 (/.../) 命令，可以暂时删除你 Java 代码的一部分。如果错误消失，仔细看看你的代码注释并找出问题所在。

客户端和服务器端头文件

有时，当一个 JSP 表现的不像预期的那样，查看一下原始的 HTTP 请求和响应是非常有用的。如果你熟悉 HTTP 的结构，你可以读取请求和响应，看看那些头文件中到底是社么信息。

重要的调试技巧

这里是一些关于 JSP 调试的调试技巧：

- 向浏览器查看它显示的页面的原始内容。这可以帮助识别格式问题。它通常是视图菜单下的一个选项。
- 确保浏览器不会通过强迫一个完全加载的页面来缓存先前的请求的输出。在 Netscape Navigator，使用 Shift-Reload；在 Internet Explorer ，使用 Shift-Refresh。

JSP – 安全性

JavaServer Pages 和 servlets 有几种可用的机制可以使 web 开发人员用来保护应用程序。资源可以通过在应用程序部署描述中对它们进行识别并且为它们分配一个角色来声明式地保护它们。

有几种级别的身份验证是可用的，从使用基本标示符的基本验证到复杂的使用证书的密码验证。

基本角色的验证

Servlet 规范中的认证机制使用的是一项被称为基于角色的安全技术。该想法是你通过角色来创建角色和限制资源，而不是限制用户级别的资源。

你可以定义在文件 tomcat-users.xml 中定义不同的角色，该文件位于 Tomcat 的主页目录中的 conf. 中。此文件的一个示例如下所示：

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
<role rolename="tomcat"/>
<role rolename="role1"/>
<role rolename="manager"/>
<role rolename="admin"/>
<user username="tomcat" password="tomcat" roles="tomcat"/>
<user username="role1" password="tomcat" roles="role1"/>
<user username="both" password="tomcat" roles="tomcat,role1"/>
<user username="admin" password="secret" roles="admin,manager"/>
</tomcat-users>
```

这个文件在用户名称、密码和角色之间定义了一个简单的映射。请注意，一个给定的用户可能有多个角色，例如，在“tomcat”角色中的用户名=“both”，角色是“role1”。

一旦你识别和定义了不同的角色，一个基于角色的安全限制可以通过使用 元素被放置在不同的 web 应用程序中，该元素在 WEB-INF 目录中的 web.xml 文件中是可用的。

下面是 web.xml 中一个简单的示例：

```
<web-app>
...
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>
```

```
            SecuredBookSite
          </web-resource-name>
          <url-pattern>/secured/*</url-pattern>
          <http-method>GET</http-method>
          <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
          <description>
          Let only managers use this app
          </description>
          <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>
    <security-role>
        <role-name>manager</role-name>
    </security-role>
    <login-config>
     <auth-method>BASIC</auth-method>
    </login-config>
...
</web-app>
```

以上条目将意味着：

- 任何 通过 /secured/* 对一个 URL匹配 的HTTP GET 或者 POST 请求都将被安全限制所接受。
- 一个有着管理员角色的人是可以访问被保护的资源的。
- 最后，login-config 元素是用来描述身份验证的 BASIC 形式。

现在，如果你试图浏览任何包含 /security 目录的 URL，它将显示一个询问用户名和密码的对话框。如果你提供一个用户“admin”和密码“secrer”，那么只有你可以通过 /secured/* 访问上面的 URL，因为我们已经定义了用户为管理员角色，而该角色是有权访问该资源的。

基于表单的身份验证

当时使用表单身份验证方法时，你必须提供一个登录表单来提示用户输入用户名和密码。下面是一个简单登录页面 login.jsp 的代码，用来创建一个相同目的的表单：

```
<html>
<body bgcolor="#ffffff">
  <form method="POST" action="j_security_check">
    <table border="0">
    <tr>
    <td>Login</td>
    <td><input type="text" name="j_username"></td>
```

```
    </tr>
    <tr>
    <td>Password</td>
    <td><input type="password" name="j_password"></td>
    </tr>
    </table>
    <input type="submit" value="Login!">
    </center>
  </form>
</body>
</html>
```

在这里，你必须确保登录表单中必须包含以 j_username 和 j_password 命名的表单元素。在

<

form> 标签中的动作必须是 j_security_check。POST必须以表单的方法来使用。同时你必须修改 标签来指定 auth-method 作为表单：

```
<web-app>
...
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>
           SecuredBookSite
         </web-resource-name>
         <url-pattern>/secured/*</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <description>
         Let only managers use this app
         </description>
         <role-name>manager</role-name>
      </auth-constraint>
   </security-constraint>
   <security-role>
       <role-name>manager</role-name>
   </security-role>
   <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
   </login-config>
```

```
...
</web-app>
```

现在，当你试图用 URL /secured/* 访问任何资源时，它将显示以上的表单，要求用户 id 和密码。当容器看到“j_security_check”动作时，它会使用一些内部机制来对调用方进行身份验证。

如果登录成功，调用者会被授权访问安全资源，那么从那时起容器会用一个 session-id 来识别调用者的登录会话。容器会用一个包含 session-id 的 cookie 来保持这个登录会话。服务器将 cookie 发送回客户端，并且只要调用者使用后续的请求显示这个 cookie 时，那么容器就会知道这个调用者是谁。

如果登录失败，那么服务器将发回由 form-error-page 设置识别的页面。

这里，j_security_check 是登录表单中使用基于表单登录的应用程序必须指定的一个动作。在相同的表单中，你该应该有一个名为 j_username 的文本输入控件和一个名为 j_password 的密码输入控件。当你看到这，这意味着表单中包含的信息将会被提交到服务器，服务器将会检查用户名和密码。这一步如何实现是由服务器指定的。

检查 Standard Realm Implementations (http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html) 来来了解 j_security_check 是如何为 Tomcat 容器工作的。

Servlet/JSP 中的程序性安全

HttpServletRequest 对象提供了以下方法，它可以在运行时用于挖掘安全信息：

序号	方法及描述
1	String getAuthType() getAuthType() 方法返回一个字符串对象，代表用于保护 Servlet 的身份验证方案的名称。
2	boolean isUserInRole(java.lang.String role) isUserInRole() 返回一个布尔型的值: 如果用户在给定的角色中值为真，否则值为假。
3	String getProtocol() getProtocol() 方法返回一个字符串对象，代表用户发送请求的协议。它的值可以用来检查确定一个保护协议是否被使用。

4	boolean isSecure() isSecure() 方法返回一个布尔型的值，代表请求是否是使用的 HTTPS协议。真值代表是并且连接是安全的。假值代表不是。/p>
5	Principle getUserPrinciple() getUserPrinciple() 方法返回一个 java.security.Principle 对象，该对象包含当前已验证用户的用户名。

例如，一个 JavaServer Page 连接的是管理员页面，你可能会使用以下代码：

```
<% if (request.isUserInRole("manager")) { %><a href="managers/mgrreport.jsp">Manager Report</a><a href="manager
```

在一个 JSP 或者 servlet 里，通过检查用户的角色，你可以自定义 web 网页来显示只有用户自己可以访问的条目。如果你需要用户名作为身份验证表单的输入，那么你可以在请求对象中调用 getRemoteUser 方法。

JSP – 国际化

在我们继续讨论之前，让我来解释以下三个重要的条目：

- **国际化（i18n）：**这意味着可以使网站根据访问者的语言或者国籍翻译成不同版本的内容。
- **本地化（l10n）：** 这意味着将资源添加到一个网站来适应特定的地理或文化，例如将印度语添加到网站。
- **局部区域：** 这是一个特定的文化或地理区域。它通常被称为一个语言符号，该语言符号紧随其后的是一个国家符号，它们之间用下划线分隔。例如，“en_US”代表美国的英语语言环境。

当创建一个全球性的网站时，有很多条目都需要考虑到。本教程在这方面不会提供给你完整的细节，但是在你如何很据互联网社区所在的位置，也就是局部环境，来向它们提供不同语言的网页方面，它会给你一个很好的例子。

一个 JSP 可以根据请求者的区域位置来收集适当的网站版本，并且根据当地语言、文化和需求来提供适当的网站版本。下面是请求对象返回局部区域对象的方法。

```
java.util.Locale request.getLocale()
```

检测局部区域

以下是重要的局部区域的方法，你可以用来检测请求者的位置、语言和语言环境。以下所有方法显示设置在请求者的浏览器上的国家名称和语言名称。

S.N.	方法及描述
1	String getCountry() 该方法返回国家/地区的代码，用大写的 ISO 3166 2- 字符格式表示语言环境。
2	String getDisplayCountry() 该方法返回一个局部区域境的国家名称，该名称适合显示给用户。
3	String getLanguage() 该方法为局部区域用 ISO 639 格式返回小写的语言代码。

4	String getDisplayLanguage() 该方法返回一个局部区域的语言名称，该名称可以适合显示给用户。
5	String getISO3Country() 该方法返回一个局部区域国家的三个字母的缩写。
6	String getISO3Language() 该方法返回一个局部区域语言的三个字母的缩写。

例子

这个例子展示了你如何为 JSP 中的一个请求显示一个语言和与其相关的国家：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%
  //Get the client's Locale
  Locale locale = request.getLocale();
  String language = locale.getLanguage();
  String country = locale.getCountry();
%>
<html>
<head>
<title>Detecting Locale</title>
</head>
<body>
<center>
<h1>Detecting Locale</h1>
</center>
<p align="center">
<%
  out.println("Language : " + language  + "<br />");
  out.println("Country  : " + country   + "<br />");
%>
</p>
</body>
</html>
```

语言设置

一个 JSP 可以输出一个用西方欧洲语言编写的页面，比如英语、西班牙语、德语、法语、意大利语、荷兰语等等。在这里，适当的设置内容语言标题来显示所有的字符是很重要的。

第二点是使用 HTML 实体显示所有的特殊字符，例如，"ñ" 代表"ñ"，"¡"代表 “¡”，如下：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%
   // Set response content type
   response.setContentType("text/html");
   // Set spanish language code.
   response.setHeader("Content-Language", "es");
   String title = "En Español";

%>
<html>
<head>
<title><%  out.print(title); %></title>
</head>
<body>
<center>
<h1><%  out.print(title); %></h1>
</center>
<div align="center">
<p>En Español</p>
<p>¡Hola Mundo!</p>
</div>
</body>
</html>
```

局部区域的特定的日期

你可以使用 java.text.DateFormat 类和它的静态 getDateTimeinstance() 方法来为局部区域格式化日期和时间特性。下面的例子显示了对于一个给定的局部区域，如何格式化日期特性：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.DateFormat,java.util.Date" %>
```

```
<%
   String title = "Locale Specific Dates";
   //Get the client's Locale
   Locale locale = request.getLocale( );
   String date = DateFormat.getDateTimeInstance(
                           DateFormat.FULL,
                           DateFormat.SHORT,
                           locale).format(new Date( ));
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Local Date: <%  out.print(date); %></p>
</div>
</body>
</html>
```

局部区域特定的货币

在一个局部区域特定的货币，你可以使用 java.txt.NumberFormat 类和它的静态方法 getCurrencyinstance() 来格式化一个数量值，例如一个长型或双精度类型。下面的例子显示了对于一个给定的局部区域，如何格式化货币特性：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.NumberFormat,java.util.Date" %>

<%
   String title = "Locale Specific Currency";
   //Get the client's Locale
   Locale locale = request.getLocale( );
   NumberFormat nft = NumberFormat.getCurrencyInstance(locale);
   String formattedCurr = nft.format(1000000);
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
```

```
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Formatted Currency: <%  out.print(formattedCurr); %></p>
</div>
</body>
</html>
```

局部区域特定的百分比

你可以使用 java.txt.NumberFormat 类和它的静态方法 getPercentInstance() 来获取局部区域特定的百分比。下面的例子显示了对于一个给定的局部区域，如何格式化百分比特性：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.NumberFormat,java.util.Date" %>

<%
   String title = "Locale Specific Percentage";
   //Get the client's Locale
   Locale locale = request.getLocale( );
   NumberFormat nft = NumberFormat.getPercentInstance(locale);
   String formattedPerc = nft.format(0.51);
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Formatted Percentage: <%  out.print(formattedPerc); %></p>
</div>
</body>
</html>
```

3

JSP 面试问题

JSP – 面试问题

亲爱的读者，这些 JSP **面试问题**是专门设计来让你了解问题的本质的，而这些问题都是在你面试时对 JSP 主题可能遇到的。根据我的经验，在你面试的过程中，好的面试官并不打算问你任何特殊的问题，通常的问题是以一些基本的概念为开始的，而后他们会继续在之前的基础上进行进一步的讨论以及你的回答：

问：什么是 JSP?

答：JavaServer Pages(JSP) 是一项支持动态内容的 Web 网页的开发技术，它可以帮助开发人员通过利用特殊的 JSP 标签在 HTML 页面中插入Java代码，这些标签大部分都是以 <% 和 %> 为开始。

问：使用 JSP 的优点是什么?

答：JSP 提供了如下所列的几个优势：

- 因为 JSP 允许在 HTML页面本身中嵌入动态元素而不是使用一个单独的 CGI 文件，所以性能明显更好一点。
- JSP 在被服务器处理之前总会被编译一次，它不像 CGI/Perl 那样，每一次页面被请求时，服务器都需要加载一个解释器和目标脚本。
- JavaServer Pages 是在 Java Servlets API 之上创建的，所以 JSP 可以像 Servlets那样也可以访问所有强大的企业级的 Java APIs，包括 JDBC，JNDI，EJB，JAXP 等等。
- JSP 页面可以结合 servlets 进行使用，处理业务逻辑，该模型是通过 Java servlet 模板引擎支持的。

问：和 ASP 相比，JSP 的优势是什么?

答：JSP 的优势是双重的。

首先，动态的部分是用 Java 中编写的，而不是 Visual Basic 或者其他 MS 特定的语言，所以它用来更强大也更容易。

其次，它可以很方便的移植到其他操作系统，并且是非微软 web 服务器。

问：和单纯的 Servlets 相比，JSP 的优势是什么?

答：与通过使用大量的 println 语句生成的 HTML 相比，JSP 在编写（和修改）常规的 HTML上更方便。其他的优点是：

- 在 HTML 页面中嵌入 Java 代码。
- 跨平台。
- 创建数据库驱动的 web 应用程序。
- 服务器端的编程功能。

问：和服务器端包含（SSI）相比，JSP 的优势是什么?

答：SSI 只是用于简单的包裹体，而不是用于“真正的”使用表格数据的应用程序，建立数据库连接，等等。

问：和 JavaScript 相比，JSP 的优势是什么?

答：JavaScript 可以在客户端生成动态的 HTML，却不能和 web 服务器进行交互来完成复杂的任务，例如数据库的访问和图像处理等等。

问：和静态 HTML 相比，JSP 的优势是什么?

答：当然，常规的 HTML 不能包含动态信息。

问：解释一个 JSP 的生命周期。

答：一个 JSP 的生命周期包含以下步骤：

- **编译：**当浏览器请求一个 JSP 时，JSP 引擎首先检查是否需要编译这个页面。如果这个页面从来没有被编译过，或者 JSP 在最后一次被编译后已经被修改了，那么 JSP 引擎会编译这个页面。

  编译过程包括三个步骤：

  - 解析 JSP。
  - 将 JSP 转换成一个 servlet。

    - 编译这个 servlet。

- **初始化**：当一个容器加载一个 JSP 时，在修改任何请求之前会调用 jsInit() 方法。
- **执行**：无论何时一个浏览器请求一个 JSP 时，这个页面都会被加载并且初始化，JSP 引擎在 JSP 中调用 _jspService() 方法。一个 JSP 的 _jspService() 方法在每一次请求时都会被调用，并且为该请求负责生成响应，该方法还负责生成所有的7个 HTTP 方法的响应，也就是 GET， POST，DELETE 等等。
- **清除**：当一个 JSP 被一个容器从应用中移除时是 JSP 生命周期的破坏阶段。jspDestroy() 方法相当于 servlets 中的销毁方法。

问：在 JSP 中一个 sciptlet 是什么？它的语法是什么？

答：一个 sciptlet 可以包含任意数量的 JAVA 语言的语句、变量或者方法声明，或者在页面脚本语言中有效地表达式。

Scriptlet 的语法如下：

```
<% code fragment %>
```

问：什么是 JSP声明？

答：一个声明可以声明一个或者多个变量或者方法，这些变量和方法你可以在后面的 JSP 文件中的 Java 代码里使用。你必须在你使用之前在 JSP 文件中定义这些 变量或方法。

```
<%! declaration; [ declaration; ]+ ... %>
```

问：什么是 JSP 表达式？

答：一个 JSP 表达式元素包含一个被求值的脚本语言表达式，转换成一个字符串，并插入到表达式在 JSP 文件中出现的位置。

根据 Java 语言规范，表达式元素可以包含任何有效地表达式，但是你不能使用一个分号来结束一个表达式。

它的语法是：

```
<%= expression %>
```

问：什么是 JSP 注释?

答：JSP 注释标记的是 JSP容器忽略的文本或者语句。一个 JSP 注释在你想要隐藏或者“注释掉”你 JSP 页面的一部分时是很有用的。

下面的 JSP 注释的语法：

```
<%-- This is JSP comment --%>
```

问：什么是 JSP 指令?

答：一个 JSP 指令影响着 servlet 类的总体结构。它通常具有以下形式：

```
<%@ directive attribute="value" %>
```

问：指令标签的类型是什么?

答：指令标签的类型如下：

- <%@ page ... %> : 定义 page-dependent 属性，例如脚本语言、错误页面、和缓冲需求。
- <%@ include ... %> : 在翻译阶段包含一个文件。
- <%@ taglib ... %> : 声明一个标签库，包含在页面中使用的自定义动作。

问：什么是 JSP 操作?

答：JSP 操作使用 XML 语法结构来控制 servlet 引擎的行为。你可以动态插入一个文件、重用 JavaBeans 组件、转发用户到另一个网页，或者为 Java 插件生成 HTML。

它的语法如下：

```
<jsp:action_name attribute="value" />
```

问：一些 JSP 操作的名称?

答：jsp:include， jsp:useBean， jsp:setProperty， jsp:getProperty， jsp:forward， jsp:plugin，jsp:element， jsp:attribute， jsp:body， jsp:text

问：什么是 JSP 字面值?

答：字面值是一些值，例如一个数字或者一个文本字符串，编写程序代码的一部分。JSP 表达式语言定义以下字面值：

- Boolean：真和假
- Integer：同 Java
- Floating point：同 Java
- String：使用单引号和双引号；" is escaped as \", ' is escaped as \', and \ is escaped as \
- Null:空

问：什么是页面指令?

答：****页面指令用于为属于当前 JSP 页面的容器提供指示。你可能会在你的 JSP 页面的任何位置编写页面指令。

问：页面指令的各种属性是什么?

答：页面指令包含以下13个属性。

1. language
2. extends
3. import
4. session
5. isThreadSafe
6. info
7. errorPage
8. isErrorpage
9. contentType
10. isELIgnored
11. buffer

12. autoFlush
13. isScriptingEnabled

问：一个缓冲的属性是什么?

答：缓冲属性为服务器输出响应对象指定缓冲特性。

问：当缓冲区设置为“空”时，发生了什么?

答：当缓冲区设置为“空”时，servlet 输出会直接定向到响应输出对象。

问：什么是 autoFlush 属性?

答：当一个缓冲区已满时，autoFlush 属性会指定缓冲输出是否会被自动溢出，还是应该抛出一个异常来指示缓冲区已满。

真（默认）值说明自动缓冲溢出，假值将抛出一个异常。

问：什么是 contentType 属性?

答：contentType 属性为 JSP 页面和生成的响应页面设置字符编码。默认的内容类型是 text/html，这是 HTML 页面里标准的内容类型。

问：什么是 errorPage 属性?

答：如果当前运行的页面中有一个错误的话，errorPage 属性通知 JSP 引擎显示的是哪一个页面。errorPage 属性的值是相对的 URL。

问：什么是 isErrorPage 属性?

答：isErrorPage 属性表明当前的 JSP 可以被另一个页面用于错误页面。 isErrorPage 的值不是真就是假。isErrorPage 属性的默认值是假。

问：什么是 extends 属性?

答：extends 属性指定一个超级类，该类中生成的 servlet 必须被扩展。

问：什么是 import 属性?

答：import 属性和 Java 导入语句有着相同的作用和表现。导入选项的值是你想要导入的包的名称。

什么是 info 属性?

答：info 属性让你提供对 JSP 的描述。

什么是 isThreadSafe 属性?

答：isThreadSafe 选项标志着一个页面是线程安全的。默认情况下，所有的 JSPS 都被认为是线程安全的。如果你设置 isThreadSafe 选项的值为假，JSP引擎确保一次只有一个线程在执行你的 JSP。

什么是 language 属性?

答：language 属性表明在 JSP 脚本语言中使用的编程语言。

什么是 session 属性?

答：session 属性表明 JSP 页面使用的是否是 HTTP 会话。真值意味着 JSP 页面有权访问一个内置命令的会话对象，假值则意味着 JSP 页面无权访问内置命令的会话对象。

什么是 isElIgnored 属性?

答：isElIgnored 选项使你可以禁用表达式语言（EL）表达式的值。该属性的默认值为真，意味着表达式 ${...} 的值是由 JSP 规范决定的。如果属性的值设置为假，那么表达式不会被赋值而是被认为是静态文本。

什么是 isScriptingEnabled 属性?

答：isScriptingEnabled 属性决定是否允许使用脚本元素。

默认值（真）可以使用脚本片段、表达式，和声明。如果这个属性值被设置为假，如果 JSP 使用任何脚本片段、表达式（non-EL），或者声明，那么将会出现一个 translation-time 错误。

什么是一个 include 命令?

答：包含命令被用于在翻译阶段包含一个文件。这个指令告诉容器在翻译阶段将当前的 JSP和其他外部文件的内容进行合并。你可以在你的 JSP页面的任何位置编写包含指令。

一般使用该指令是形式如下：

```
<%@ include file="relative url" >
```

什么是一个 taglib 指令?

答：taglib 指令遵循以下语法：

```
<%@ taglib uri="uri" prefix="prefixOfTag">
```

Uri 属性值解析一个容器理解的位置。

Prefix 属性通知一个容器标记位是一个自定义动作。

Taglib 指令遵循以下语法：

```
<%@ taglib uri="uri" prefix="prefixOfTag">
```

在动作元素中 id 和 scope 属性的意义是什么?

答：

- **ID 属性：**id 属性唯一的标识动作元素，允许在 JSP 页面中引用动作。如果动作创建一个对象的实例，id 值可以通过隐式对象 PageContext 对它进行引用。

- Scope 属性：这个属性标识动作元素的生命周期。Id 属性和 scope 属性是直接相关的，scope 属性决定了与 id 属性相关的对象的寿命。scope 属性有四个可能的值：（a）页面，（b）请求，（c）会话，和（d）应用程序。

<jsp:include> 动作的功能是什么?

答：该操作允许你将文件插入到生成的页面中。语法如下：

```
<jsp:include page="relative URL" flush="true" />
```

其中，page 是页面中包含的相对应的 URL。

Flush 是一个布尔型的属性，决定被包含的资源在被包含之前它的缓冲区是否被刷新。

include 操作和 include 指令之间的区别是什么?

答：include 指令是在 JSP 页面被翻译成一个 servlet 时插入一个文件，而include 操作是在页面被请求使插入文件。

什么是 <jsp:useBean> 操作?

答：useBean 操作是非常通用的。它首先利用 id 和 scope 变量搜索一个现有的对象。如果没有找到一个对象，那么它试图创建一个指定的对象。

加载一个 bean 最简单的方法如下：

```
<jsp:useBean id="name" class="package.class" />
```

什么是 <jsp:setProperty> 操作?

答：setProperty 操作设置一个 Bean 的属性。Bean 必须在该操作之前被定义。

什么是 ` 操作?

答：getProperty 操作被用于检索一个给定的属性值，并且将其转换为一个字符串，最后将它插入到输出。

什么是 <jsp:forward> 操作?

答：forward 操作终止当前的页面，并将请求转发给另一个资源，例如一个静态页面、另一个 JSP 页面，或者一个 Java Servlet。

该操作的一个简单的语法如下：

```
<jsp:forward page="Relative URL" />
```

什么是 <jsp:plugin> 操作?

答：plugin 操作用于将 Java 组件插入到一个 JSP 页面。它决定了浏览器的类型和需要插入的

或者 标签。

如果所需要的插件不存在，下载插件，然后执行 Java 组件。Java 组件可以是一个 Applet 或者是一个 JavaBean。

JSP 操作不同的 scope 值是什么?

答：scope 属性标识了一个动作元素的生命周期。它有四个可能的值：（a）页面，（b）请求，（c）会话，和（d）应用程序

什么是 JSP 隐式对象?

答：JSP 隐式对象是在每一个页面中 JSP 容器为开发人员提供的可用的 Java 对象，开发人员可以直接调用它们，而不需要显的声明。JSP 隐式对象还被称为预定义的变量。

JSP 支持的隐式对象是什么?

答：request， response， out,session， application， config， pageContext， page， Exception。

什么是 request 对象?

答：request 对象是 javax.servlet.http.HttpServletRequest 对象的一个实例。每次客户端请求一个页面时，JSP 引擎都会创一个新的对象来表示这个请求。

request 对象为获取 HTTP 头信息提供方法，这些头信息包括表单数据、 cookie、 HTTP 方法等等。

你如何读取一个请求的头信息?

答：使用HttpServletRequest 的 getHeaderNames() 方法来读取 HTTP 头信息。该方法返回一个枚举，包含与当前请求相关头信息。

什么是一个 response 对象?

答：response 对象是 javax.servlet.http.HttpServletRequest 对象的一个实例。正如服务器创建请求对象，它还创建了一个象来表示对客户端的响应。

response 对象还定义了处理创建新的 HTTP 头的接口。通过这个对象，JSP 程序员可以添加新的 cookies 或者日期戳， HT状态码等等。

什么是外部隐式对象?