**JSP 教程**

JSP 与 PHP、ASP、ASP.NET 等语言类似，运行在服务端的语言。

JSP（全称Java Server Pages）是由 Sun Microsystems 公司倡导和许多公司参与共同创建的一种使软件开发者可以响应（me：动态）客户端请求，而动态生成 HTML、XML 或其他格式文档的Web网页的技术标准。

JSP 技术是以 Java 语言作为脚本语言的，JSP 网页为整个服务器端的 Java 库单元提供了一个接口来服务于HTTP的应用程序。

JSP文件后缀名为 **\*.jsp** 。

JSP开发的WEB应用可以跨平台使用，既可以运行在 Linux 上也能运行在 Windows 上。

**第一个 JSP 程序**

语言学习入门的第一个程序一般都是输出 "Hello World"，JSP输出 "Hello World" 代码如下所示：

<html>

<head>

<title>第一个 JSP 程序</title>

</head>

<body>

<%

out.println("Hello World！");

%>

</body>

</html>

**开始学习 JSP**

了解了 JSP 的基本概念后，现在让我们[开始来学习 JSP](http://www.runoob.com/jsp/jsp-intro.html)吧。

# JSP 简介

## 什么是Java Server Pages?

JSP全称Java Server Pages，是一种动态网页开发技术。它使用JSP标签在HTML网页中插入Java代码。标签通常以<%开头以%>结束。

JSP是一种Java servlet（java小服务，类似C#的ASP），主要用于实现Java web应用程序的用户界面部分。网页开发者们通过结合HTML代码、XHTML代码、XML元素以及嵌入JSP操作和命令来编写JSP。

JSP通过网页表单获取用户输入数据、访问数据库及其他数据源，然后动态地创建网页。

JSP标签有多种功能，比如访问数据库、记录用户选择信息、访问JavaBeans组件等，还可以在不同的网页中传递控制信息和共享信息。

## 为什么使用JSP？

JSP程序与CGI（me：php-cgi）程序有着相似的功能，但和CGI程序相比，JSP程序有如下优势：

* 性能更加优越，因为JSP可以直接在HTML网页中动态嵌入元素而不需要单独引用CGI文件。
* 服务器调用的是已经编译好的JSP文件，而不像CGI/Perl那样必须先载入解释器和目标脚本。
* JSP 基于Java Servlet API，因此，JSP拥有各种强大的企业级Java API，包括JDBC，JNDI，EJB，JAXP等等。
* JSP页面可以与处理业务逻辑的 Servlet 一起使用，这种模式被Java servlet 模板引擎所支持。

最后，JSP是Java EE不可或缺的一部分，是一个完整的企业级应用平台。这意味着JSP可以用最简单的方式来实现最复杂的应用。

## JSP的优势

以下列出了使用JSP带来的其他好处：

* 与ASP相比：JSP有两大优势。首先，动态部分用Java编写，而不是VB或其他MS专用语言，所以更加强大与易用。第二点就是JSP易于移植到非MS平台上。
* 与纯 Servlet 相比：JSP可以很方便的编写或者修改HTML网页而不用去面对大量的println语句。
* 与SSI相比：SSI无法使用表单数据、无法进行数据库链接。
* 与JavaScript相比：虽然JavaScript可以在客户端动态生成HTML，但是很难(ajax完全可以与服务器端交互)与服务器交互，因此不能提供复杂的服务，比如访问数据库和图像处理等等。
* 与静态HTML相比：静态HTML不包含动态信息。

## 接下来呢？

我们将会带您一步一步地来搭建JSP运行环境，这需要有一定的Java基础。

如果您还未学过Java，可以先学习我们为您提供的[Java教程](http://www.runoob.com/java/java-tutorial.html)。

**JSP 开发环境搭建**

JSP开发环境是您用来开发、测试和运行JSP程序的地方。

本节将会带您搭建JSP开发环境，具体包括以下几个步骤。

如果你使用的是 Eclipse 环境，可以直接参阅：[Eclipse JSP/Servlet 环境搭建](http://www.runoob.com/jsp/eclipse-jsp.html)。

**配置Java开发工具（JDK）**

这一步涉及Java SDK的下载和PATH环境变量的配置。

您可以从Oracle公司的Java页面中下载SDK：[Java SE Downloads](http://www.oracle.com/technetwork/java/javase/downloads/index.html)

Java SDK下载完后，请按照给定的指示来安装和配置SDK。最后，通过设置PATH和JAVA\_HOME环境变量来指明包括java和javac的文件夹路径，通常是java\_install\_dir/bin和java\_install\_dir。

假如您用的是Windows系统并且SDK的安装目录为C::\jdk1.5.0\_20，那么您就需要在 C:\autoexec.bat 文件中添加以下两行：

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

set JAVA\_HOME=C:\jdk1.5.0\_20

或者，在Windows NT/2000/XP下，您可以直接右击我的电脑图标，选择属性，然后高级，然后环境变量，接下来您就可以很方便地设置PATH变量并且确定退出就行了。

在Linux/Unix系统下，如果SDK的安装目录为/usr/local/jdk1.5.0\_20并且使用的是C shell，那么您就需要在.cshrc文件中添加以下两行：

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

setenv JAVA\_HOME /usr/local/jdk1.5.0\_20

或者，假如您正在使用类似于Borland JBuilder、Eclipse、IntelliJ IDEA和Sun ONE Studio这样的集成开发环境，可以试着编译并运行一个简单的程序来确定IDE（集成开发环境）是否已经知道 SDK的安装目录。

本步骤你也可以参考本站[Java开发环境配置](http://www.runoob.com/java/java-environment-setup.html)章节的教程。

**设置Web服务器：Tomcat**

目前，市场上有很多支持JSP和Servlets开发的Web服务器。他们中的一些可以免费下载和使用，Tomcat就是其中之一。

Apache Tomcat是一个开源软件，可作为独立的服务器来运行JSP和Servlets，也可以集成在 Apache Web Server中。以下是Tomcat的配置方法：

* 下载最新版本的Tomcat：<http://tomcat.apache.org/>。
* 下载完安装文件后，将压缩文件解压到一个方便的地方，比如Windows下的C:\apache-tomcat-5.5.29目录或者Linux/Unix下的/usr/local/apache-tomcat-5.5.29目录，然后创建CATALINA\_HOME环境变量指向这些目录。

在Windows机器下，Tomcat可以通过执行以下命令来启动：

%CATALINA\_HOME%\bin\startup.bat

或者

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

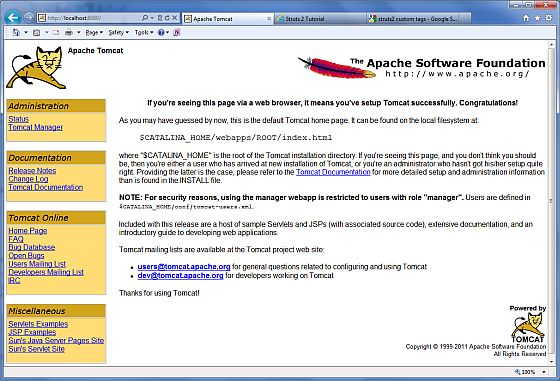
在Linux/Unix机器下，Tomcat可以通过执行以下命令来启动：

$CATALINA\_HOME/bin/startup.sh

或者

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

成功启动Tomcat后，通过访问http://localhost:8080/便可以使用Tomcat自带的一些web应用了。假如一切顺利的话，您应该能够看到以下的页面：



更多关于配置和运行Tomcat的信息可以在Tomcat提供的文档中找到，或者去Tomcat官网查阅：http://tomcat.apache.org。

在Windows机器下，Tomcat可以通过执行以下命令来停止：

%CATALINA\_HOME%\bin\shutdown

或者

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

在Linux/Unix机器下，Tomcat可以通过执行以下命令来停止：

$CATALINA\_HOME/bin/shutdown.sh

或者

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

**设置CLASSPATH环境变量**

由于servlets不是Java SE的一部分，所以您必须标示出servlet类的编译器。

假如您用的是Windows机器，您需要在C:\autoexec.bat文件中添加以下两行：

set CATALINA=C:\apache-tomcat-5.5.29

set CLASSPATH=%CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%

或者，在Windows NT/2000/XP下，您只要右击我的电脑，选择属性，然后点击高级，然后点击环境变量，接下来便可以设置CLASSPATH变量并且确定退出即可。

在Linux/Unix机器下，假如您使用的是C shell，那么您就需要在.cshrc文件中添加以下两行：

setenv CATALINA=/usr/local/apache-tomcat-5.5.29

setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH

注意：如果您的开发路径是C:\JSPDev (Windows)或者 /usr/JSPDev (Linux/Unix)，那么您就需要将这些路径添加进CLASSPATH变量中。

# Eclipse JSP/Servlet 环境搭建

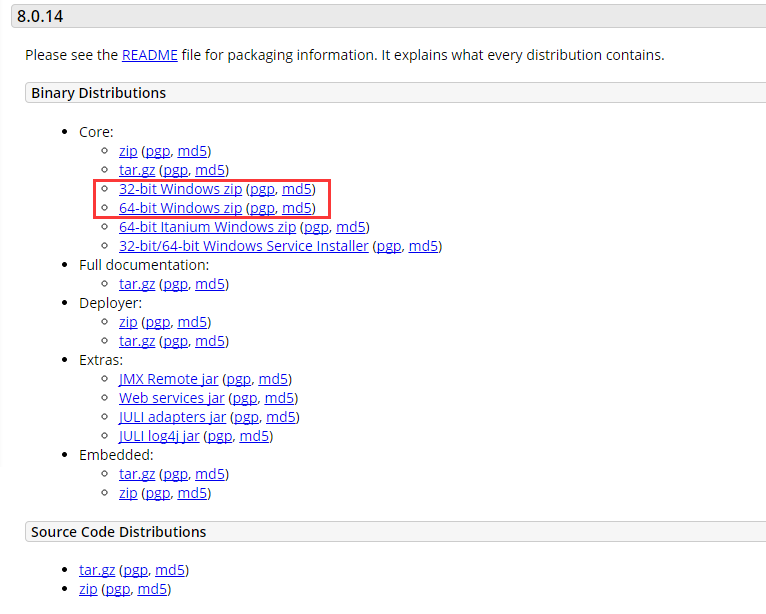
本文假定你已安装了 JDK 环境，如未安装，可参阅 [Java 开发环境配置](http://www.runoob.com/java/java-environment-setup.html)。

我们可以使用 Eclipse 来搭建 JSP 开发环境，首先我们分别下载一下软件包：

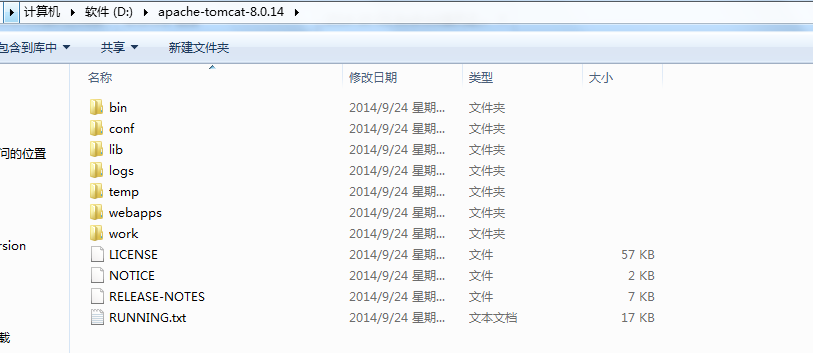
* **Eclipse J2EE：**<http://www.eclipse.org/downloads/>
* **Tomcat：**<http://tomcat.apache.org/download-70.cgi>

## Tomcat 下载安装

你可以根据你的系统下载对应的包(以下以Window系统为例)：



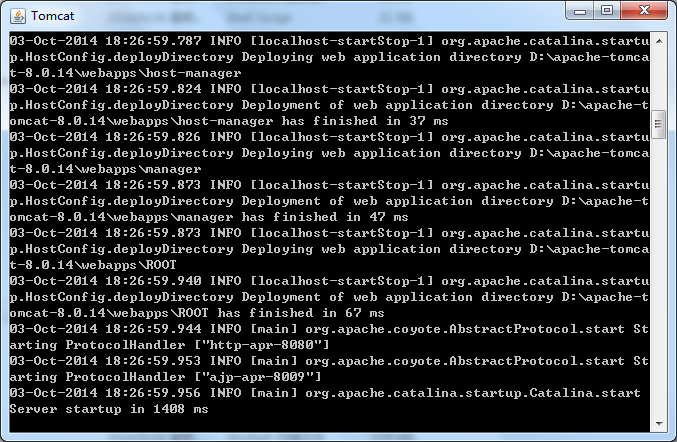
下载之后，将压缩包解压到D盘（你可以自己选择）：



注意目录名不能有中文和空格。目录介绍如下：

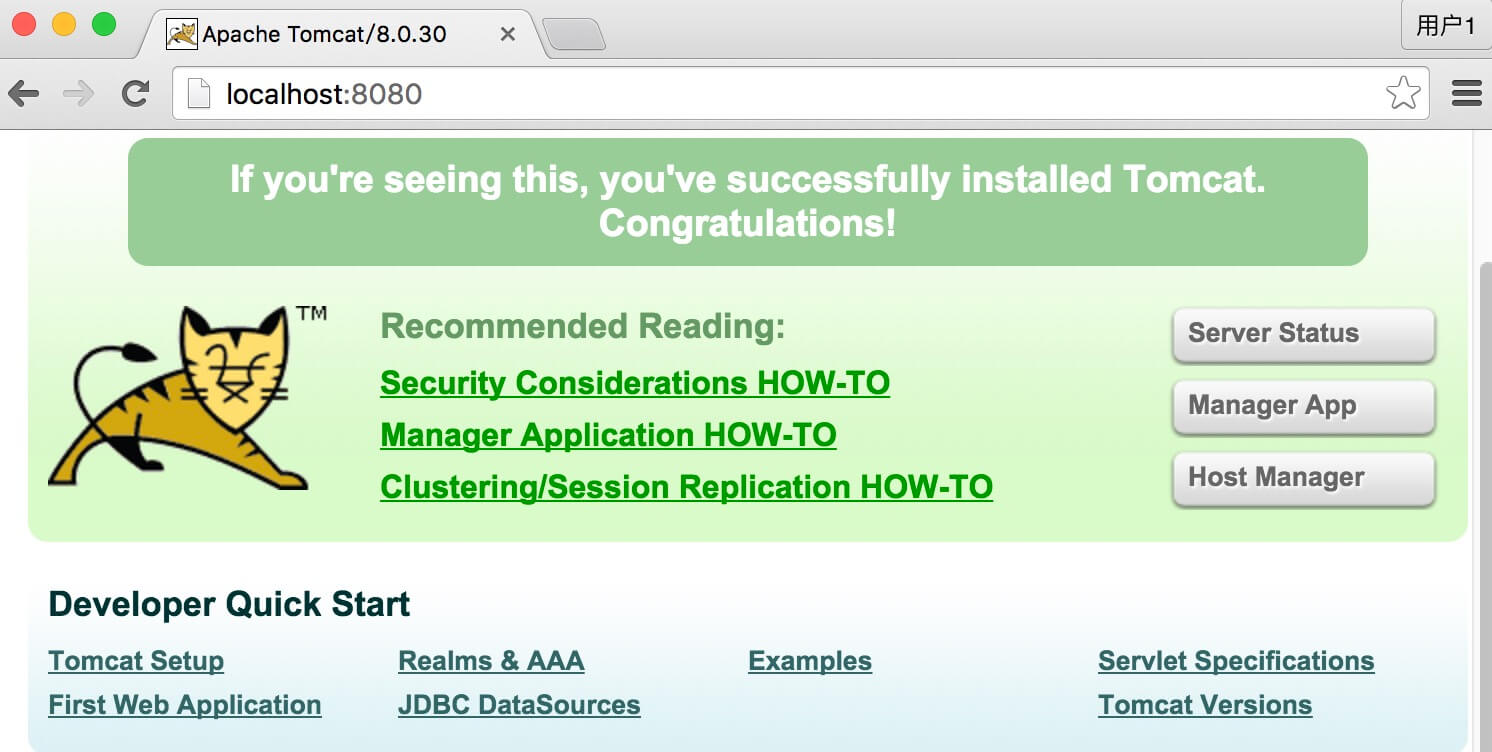
* bin：二进制执行文件。里面最常用的文件是**startup.bat**，如果是 Linux 或 Mac 系统启动文件为 **startup.sh**。
* conf:配置目录。里面最核心的文件是**server.xml**。可以在里面改端口号等。默认端口号是8080，也就是说，此端口号不能被其他应用程序占用。
* lib：库文件。tomcat运行时需要的jar包所在的目录
* logs：日志
* temp：临时产生的文件，即缓存
* webapps：web的应用程序。**web应用放置到此目录下浏览器可以直接访问**
* work：编译以后的class文件。

接着我们可以双击 startup.bat 启动 Tomcat，弹出如下界面：



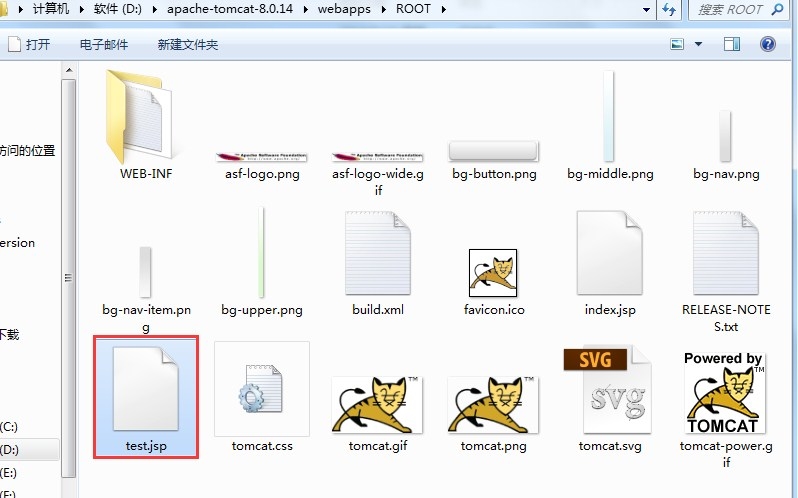
这个时候，本地的服务器就已经搭建起来了。如果想关闭服务器，可以直接关闭上面的窗口，或者在里面输入Ctrl+C禁止服务。

接着我们在浏览器中输入 **http://localhost:8080/**，如果弹出如下界面，表示tomcat安装成功并且启动起来了：



我们现在在浏览器上测试一下它吧：

首先在D:\apache-tomcat-8.0.14\webapps\ROOT目录中新建一个jsp文件：



test.jsp 文件代码如下：

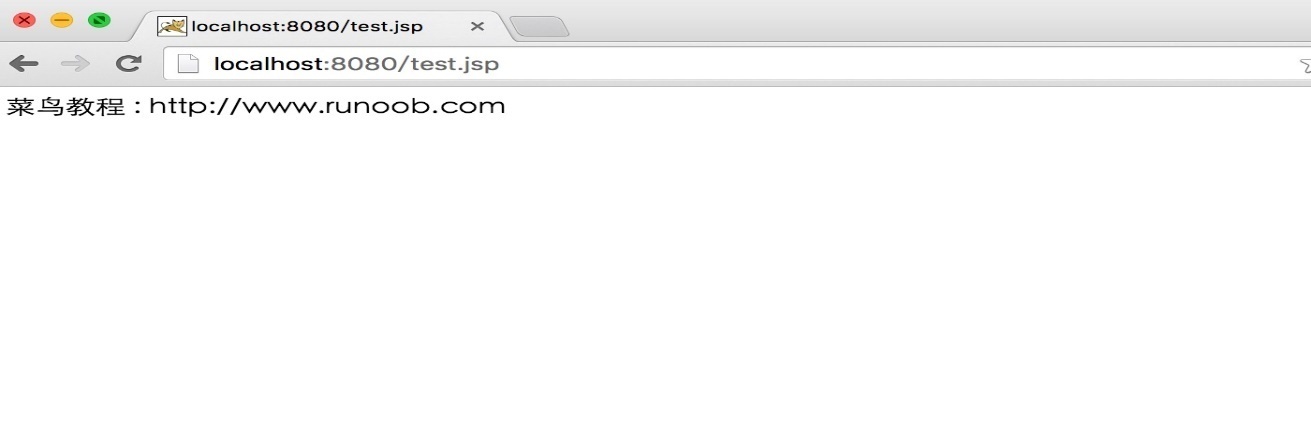
<%@ page contentType="text/html;charset=UTF-8" %>

<%

out.print("菜鸟教程 : http://www.runoob.com");

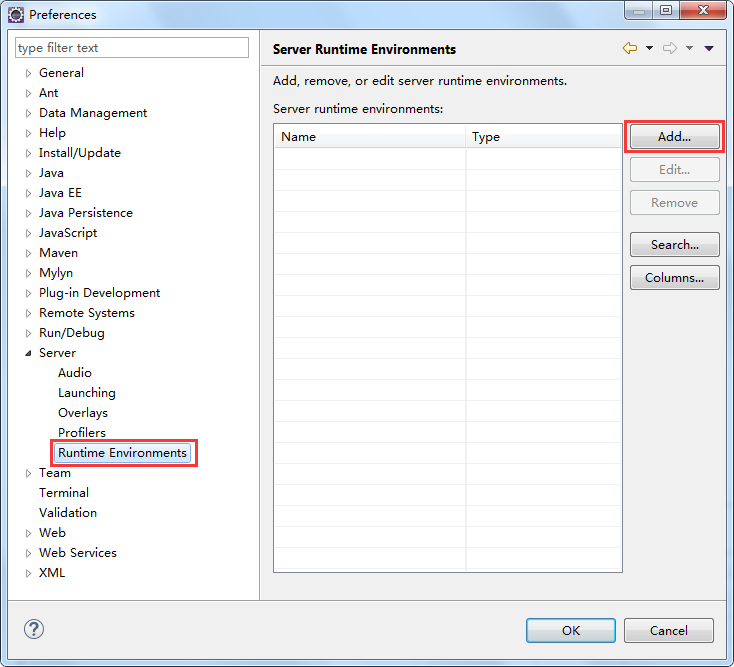
%>

接着在浏览器中访问地址 **http://localhost:8080/test.jsp**, 输出结果如下：

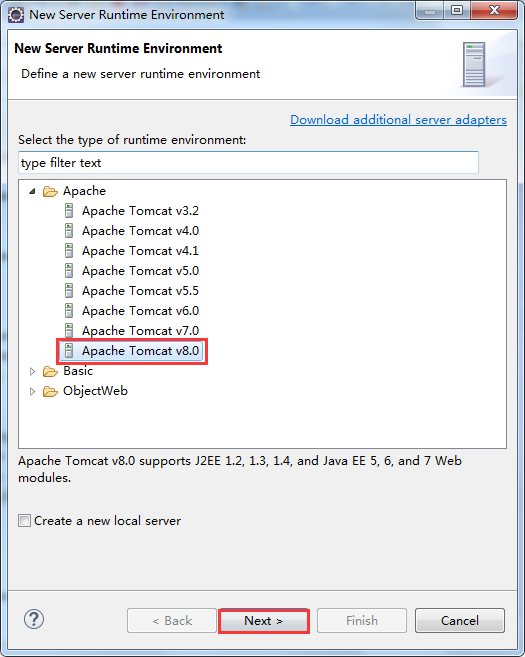


## 将 Tomcat 和 Eclipse 相关联

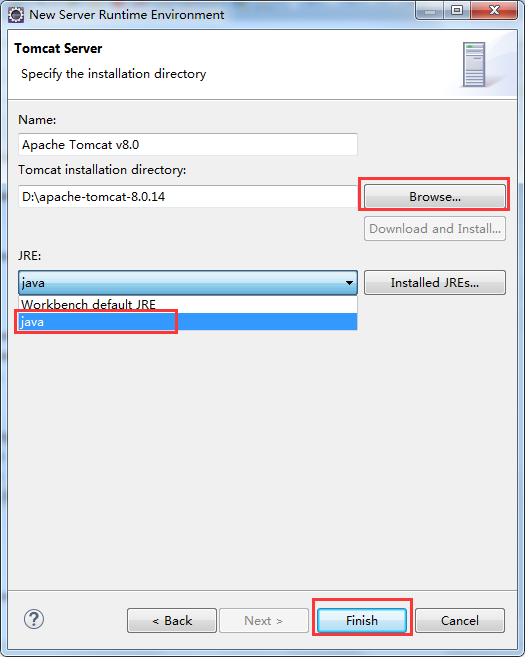
Eclipse J2EE下载后，解压即可使用，我们打开Java EE ，选择菜单栏Windows-->preferences（Mac 系统为 Eclipse-->偏好设置），弹出如下界面：



上图中，点击"add"的添加按钮，弹出如下界面：



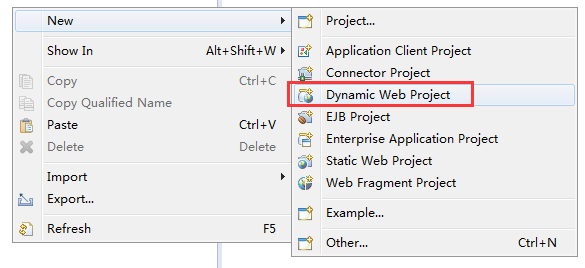
在选项中，我们选择对应的 Tomcat 版本，接着点击 "Next"，选择 Tomcat 的安装目录，并选择我们安装的 Java 环境：

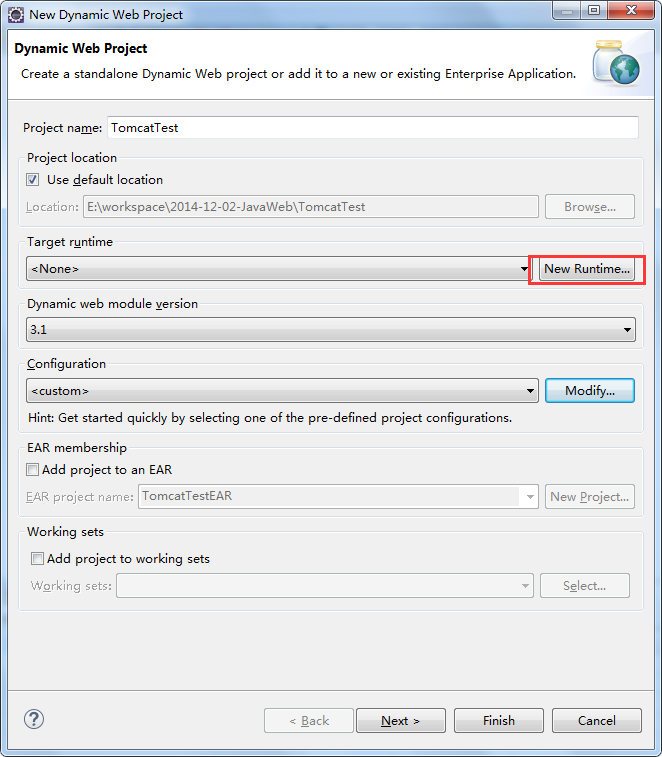


点击 "Finish"，完成配置。

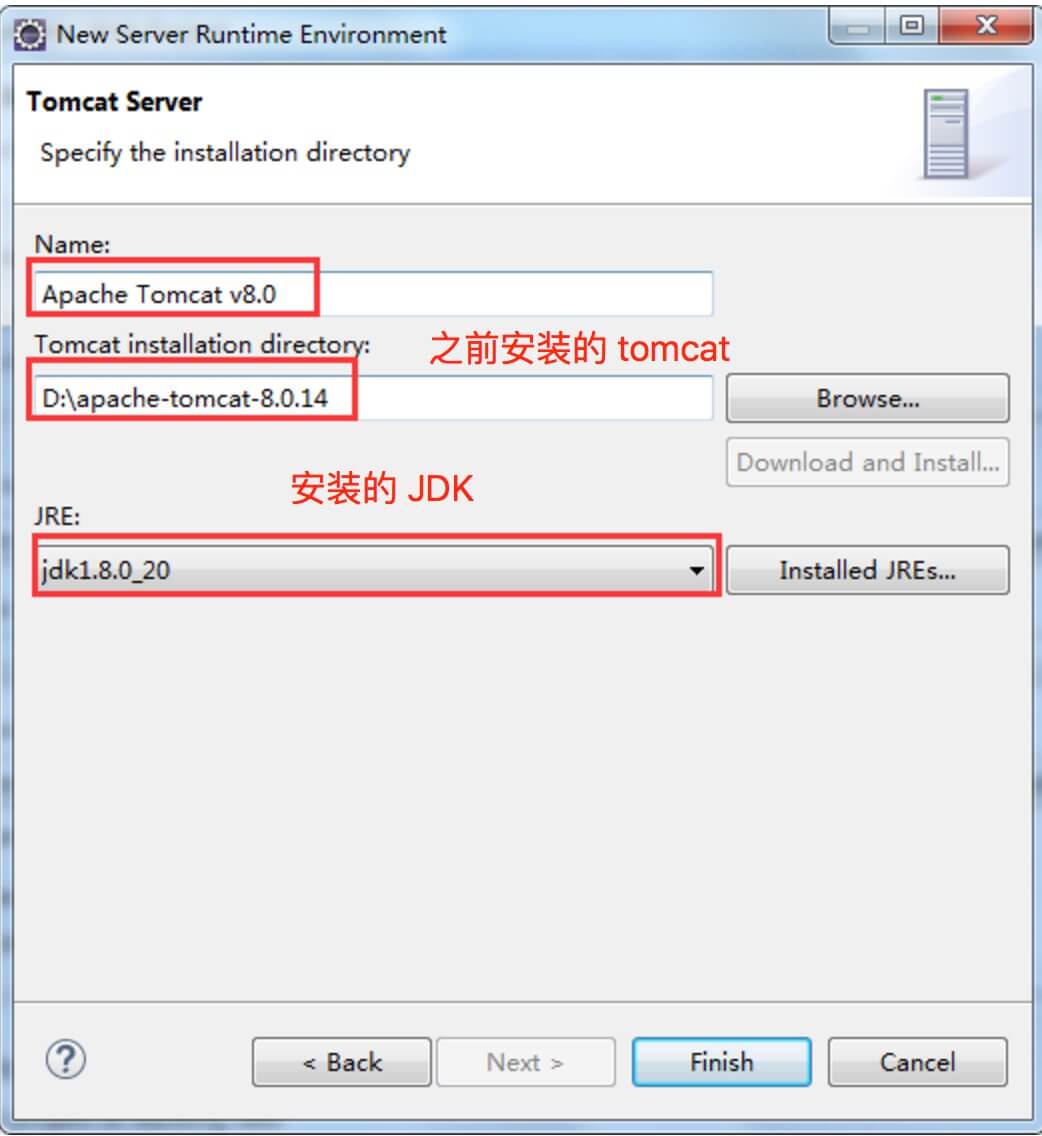
### 创建实例

选择 "File-->New-->Dynamic Web Project"，创建 TomcatTest 项目：



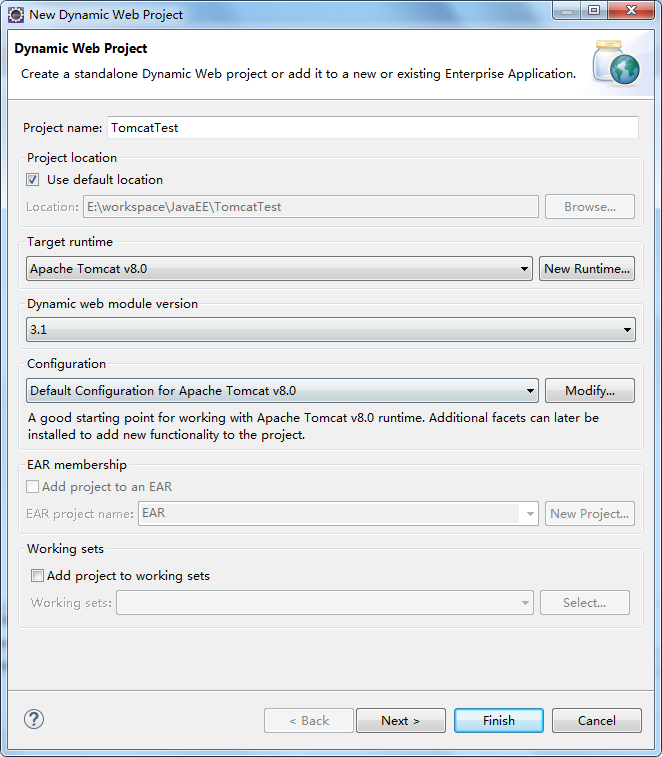


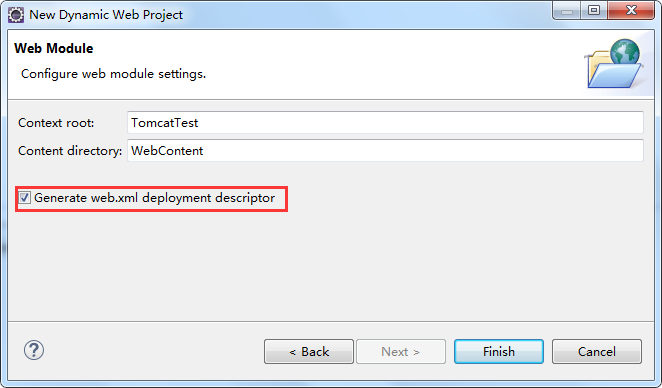
点开上图中的红框部分，弹出如下界面：



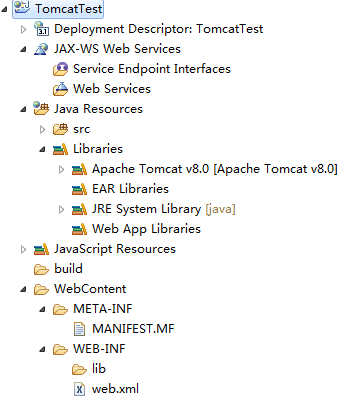
注意如果已默认选择了我们之前安装的 Tomcat 和 JDK 则可跳过此步。

然后，单击finish, 继续：





工程文件结构：



上图中各个目录解析：

* deployment descriptor：部署的描述。
* Web App Libraries：自己加的包可以放在里面。
* build：放入编译之后的文件。
* WebContent:放进写入的页面。

在WebContent文件夹下新建一个test.jsp文件。在下图中可以看到它的默认代码：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Insert title here</title>

</head>

<body>

</body>

</html>

接着我们修改下test.jsp文件代码如下所示：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>菜鸟教程</title>

</head>

<body>

<%

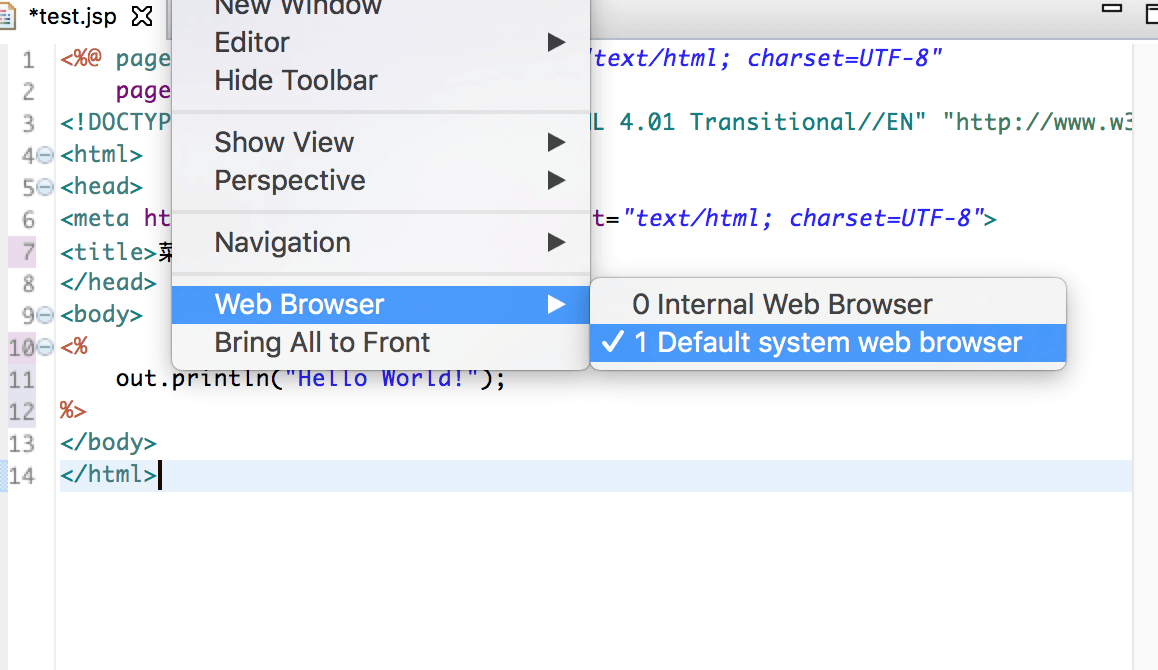
out.println("Hello World!");

%>

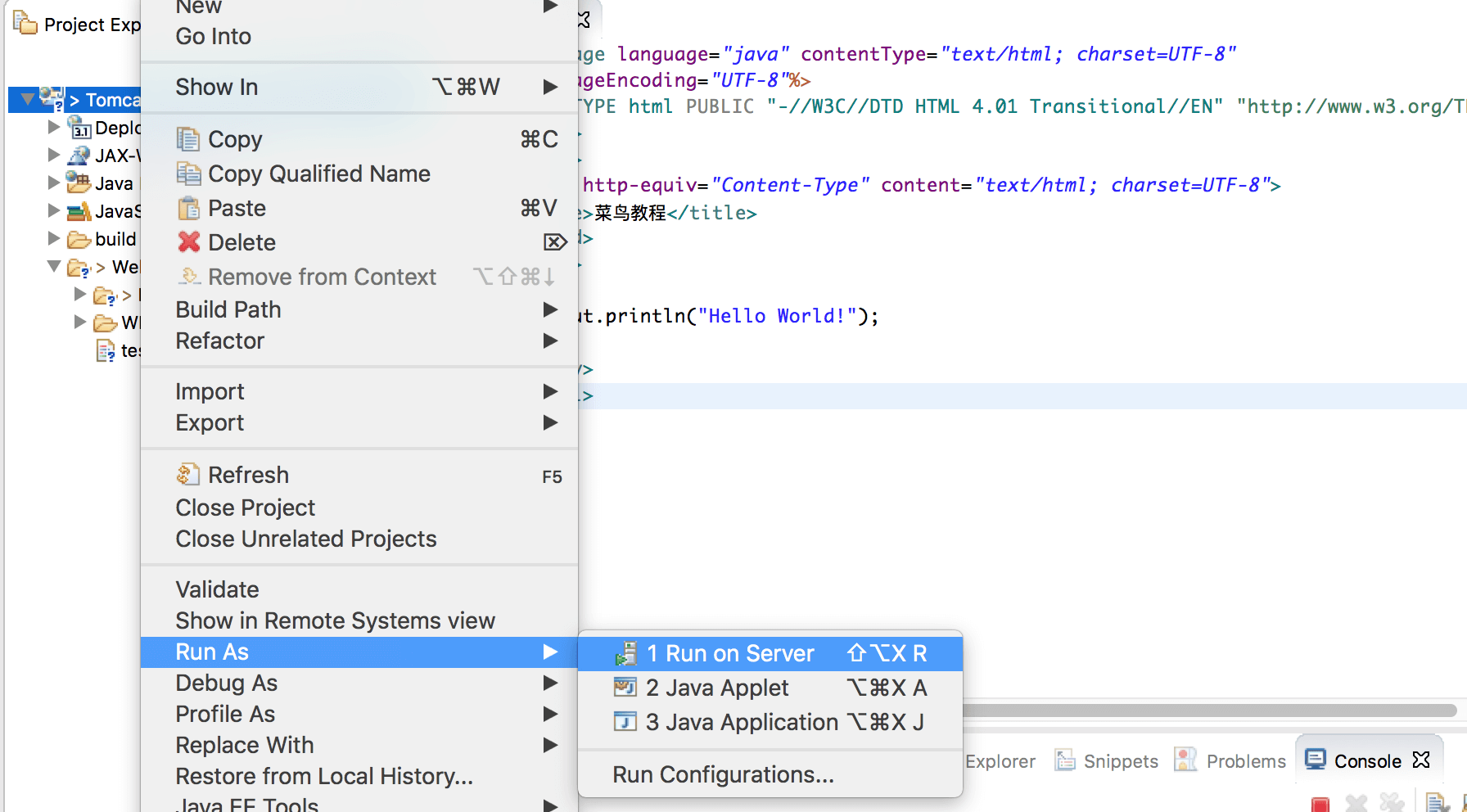
</body>

</html>

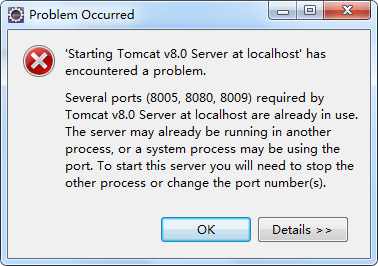
程序运行之前，我们先修改一下浏览器选项:



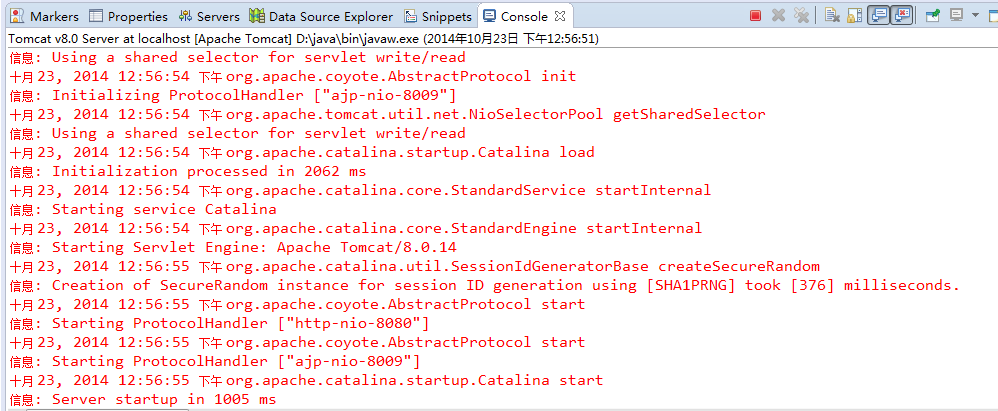
接着我们运行该项目:



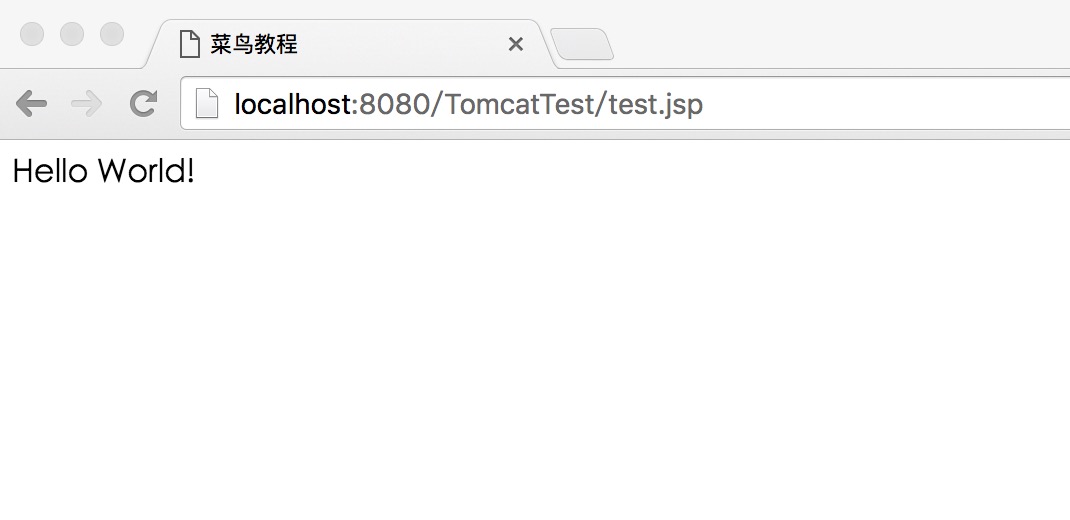
运行时，弹出如下错误：(如果没有此错误，请忽略)



原因是，我们之前点击了Tomcat安装包中的​startup.bat，这样一来就手动打开了Tomcat服务器，这明显是多余的，因为程序运行时，eclipse会自动开启Tomcat服务器。所以我们先手动关掉tomcat软件，再次运行程序，就行了。控制台信息如下：

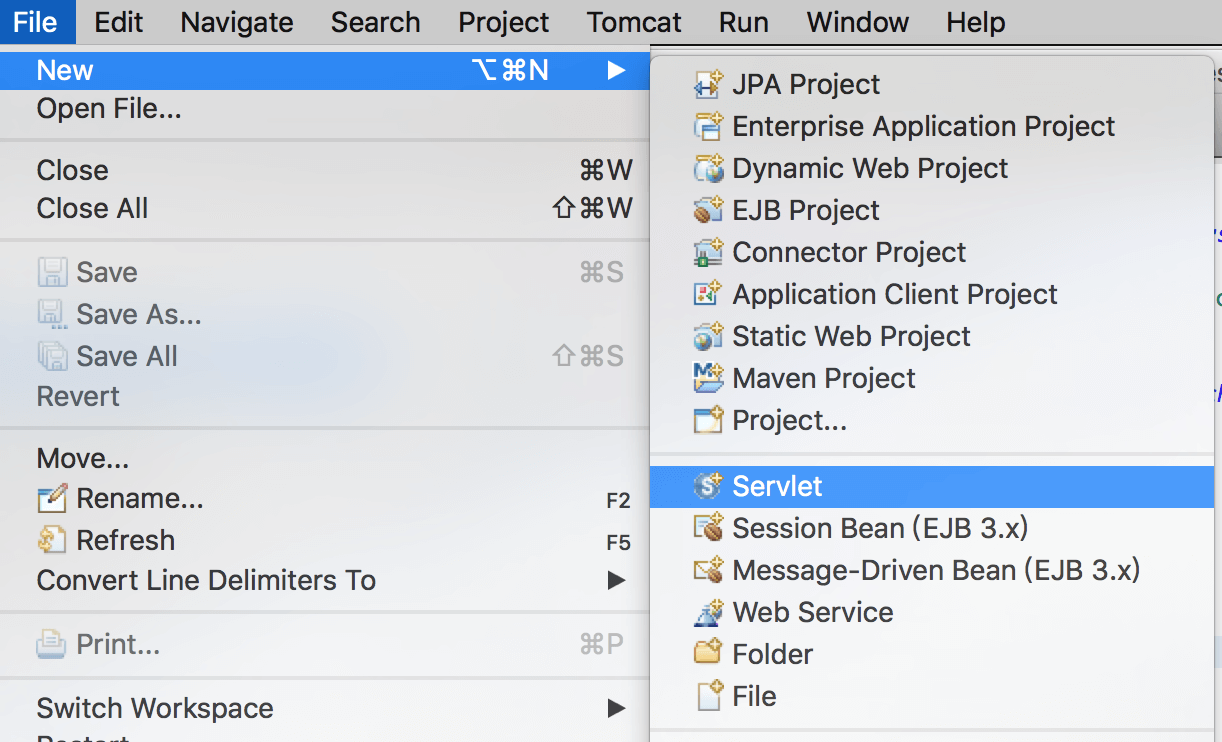


浏览器访问 **http://localhost:8080/TomcatTest/test.jsp**, 即可输出正常结果：

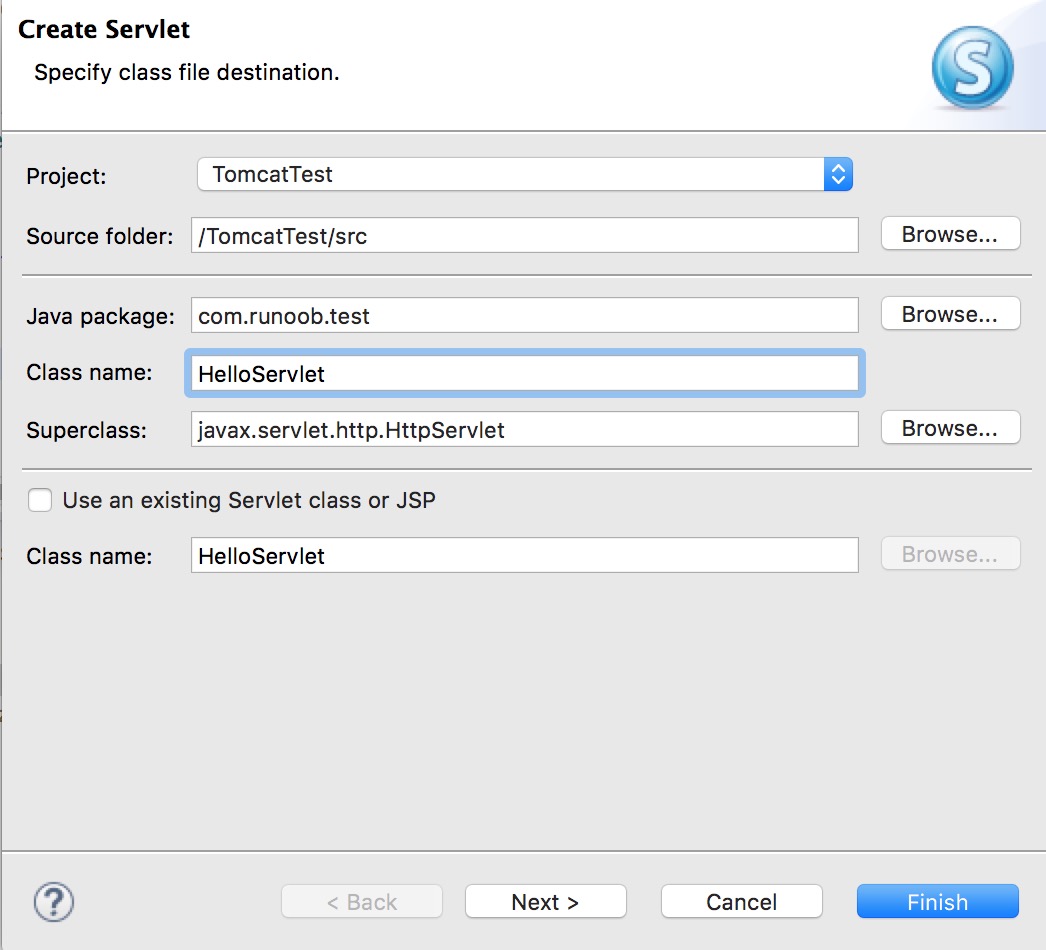


## Servlet 实例创建

我们也可以使用以上环境创建 Servlet 文件，选择 "File-->New-->Servlet":



位于 TomcatTest项目的 /TomcatTest/src 目录下创建 "HelloServlet" 类，包为 "com.runoob.test":



HelloServlet.java 代码如下所示：

package com.runoob.test;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/\*\*

\* Servlet implementation class HelloServlet

\*/

@WebServlet("/HelloServlet")

public class HelloServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#HttpServlet()

\*/

public HelloServlet() {

super();

// TODO Auto-generated constructor stub

}

/\*\*

\* @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)

\*/

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// 使用 GBK 设置中文正常显示

response.setCharacterEncoding("GBK");

response.getWriter().write("菜鸟教程：http://www.runoob.com");

}

/\*\*

\* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)

\*/

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// TODO Auto-generated method stub

doGet(request, response);

}

}

创建 /TomcatTest/WebContent/WEB-INF/web.xml 文件（如果没有），代码如下所示：

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5"

xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<servlet>

<!--类名 -->

<servlet-name>HelloServlet</servlet-name>

<!--所在的包 -->

<servlet-class>com.runoob.test.HelloServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloServlet</servlet-name>

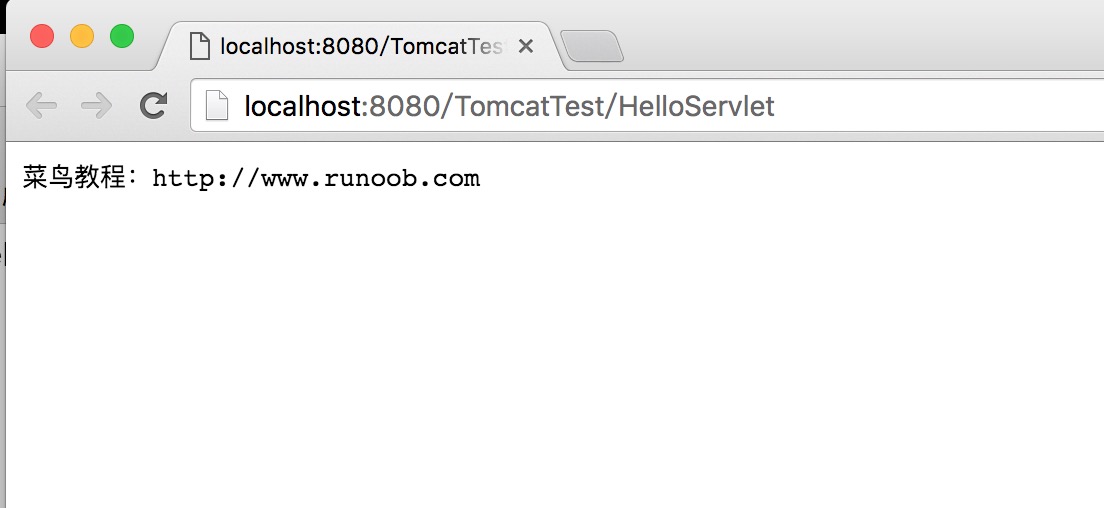
<!--访问的网址 -->

<url-pattern>/TomcatTest/HelloServlet</url-pattern>

</servlet-mapping>

</web-app>

接着重启 Tomcat，浏览器访问 **http://localhost:8080/TomcatTest/HelloServlet**：



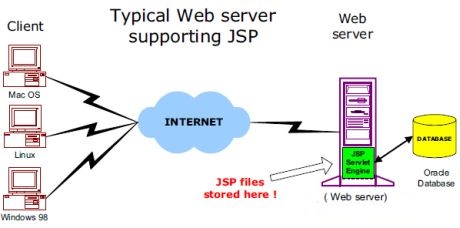
参考文章：http://www.cnblogs.com/smyhvae/p/4046862.html

# JSP 结构

网络服务器需要一个JSP引擎，也就是一个容器（me：web有web容器，spring有spring的管理容器，ejb有ejb容器等；容器就使用来封装组件的）来处理JSP页面。容器负责截获对JSP页面的请求。本教程使用内嵌JSP容器的Apache来支持JSP开发。

JSP容器与Web服务器协同合作，为JSP的正常运行提供必要的运行环境和其他服务（me：类似apache和php的组合，但解析php文件的时候会将请求转到php-cgi来处理，并将处理结果返回），并且能够正确识别专属于JSP网页的特殊元素。

下图显示了JSP容器和JSP文件在Web应用中所处的位置。

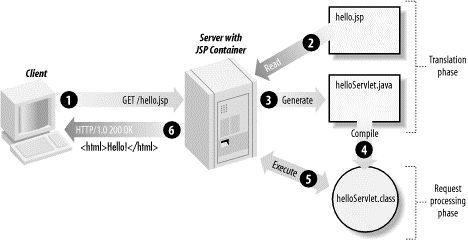


## JSP处理

以下步骤表明了Web服务器是如何使用JSP来创建网页的：

* 就像其他普通的网页一样，您的浏览器发送一个HTTP请求给服务器。
* Web服务器识别出这是一个对JSP网页的请求，并且将该请求传递给JSP引擎。通过使用URL或者.jsp文件来完成。
* JSP引擎从磁盘中载入JSP文件，然后将它们转化为servlet。这种转化只是简单地将所有模板文本改用println()语句，并且将所有的JSP元素转化成Java代码。
* JSP引擎将servlet编译成可执行类，并且将原始请求传递给servlet引擎。
* Web服务器的某组件将会调用servlet引擎，然后载入并执行servlet类。在执行过程中，servlet产生HTML格式的输出并将其内嵌于HTTP response中上交给Web服务器。
* Web服务器以静态HTML网页的形式将HTTP response返回到您的浏览器中。
* 最终，Web浏览器处理HTTP response中动态产生的HTML网页，就好像在处理静态网页一样。

以上提及到的步骤可以用下图来表示：



一般情况下，JSP引擎会检查JSP文件对应的servlet是否已经存在，并且检查JSP文件的修改日期是否早于servlet。如果JSP文件的修改日期早于对应的servlet，那么容器就可以确定JSP文件没有被修改过并且servlet有效。这使得整个流程与其他脚本语言（比如PHP）相比要高效快捷一些。

总的来说，JSP网页就是用另一种方式来编写servlet而不用成为Java编程高手。除了解释阶段外，JSP网页几乎可以被当成一个普通的servlet来对待。

[Eclipse JSP/Servlet](http://www.runoob.com/jsp/eclipse-jsp.html)

[JSP 生命周期](http://www.runoob.com/jsp/jsp-life-cycle.html)

## 笔记列表

1. 学习++

  1336705531@qq.com

Web服务器使用JSP来创建网页的步骤：

1.客户端向服务器发送一个HTTP请求。

2.服务器识别是对JSP网页（me：通过路由映射或者请求的文件路径的后缀名如.jsp）的请求后将其请求传递给Servlet引擎。

Jsp->servlet类->编译成class->执行

Jsp是先转换然后编译，而PHP是直接解释执行（其实也是先分词，然后编译）

3.Servlet引擎将磁盘中的Jsp文件转化为Servlet类，并将所有的元素转化为Java代码

4.JSP引擎将Servlet类编译成可执行的文件并将最原始的请求发送给Servlet引擎

5.服务器通过调用Servlet引擎来执行Servlet可执行文件，Servlet会产生HTML格式的输出内嵌在HTTP Response中并将其上传给服务器

6.服务器将其以HTML的格式上传到客户端的浏览器中以静态页面的形式展现给用户。

# JSP 生命周期

理解JSP底层功能的关键就是去理解它们所遵守的生命周期。

JSP生命周期就是从创建到销毁的整个过程，类似于servlet生命周期，区别在于JSP生命周期还包括将JSP文件编译成servlet。

以下是JSP生命周期中所走过的几个阶段：

* **编译阶段：**

servlet容器编译servlet源文件，生成servlet类

* **初始化阶段：**

加载与JSP对应的servlet类，创建其实例，并调用它的初始化方法

* **执行阶段：**

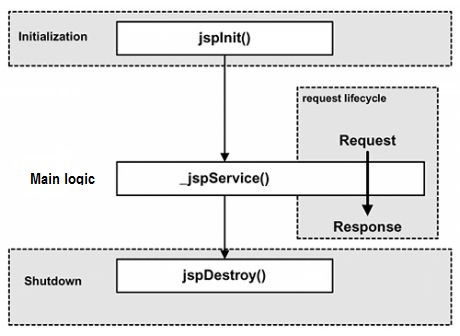
调用与JSP对应的servlet实例的服务方法

* **销毁阶段：**

调用与JSP对应的servlet实例的销毁方法，然后销毁servlet实例

编译类->加载类，构建对象->执行相应的对象方法->servlet实例销毁

很明显，JSP生命周期的四个主要阶段和servlet生命周期非常相似，下面给出图示：



## JSP编译

当浏览器请求JSP页面时，JSP引擎会首先去检查是否需要编译这个文件。如果这个文件没有被编译过，或者在上次编译后被更改过，则编译这个JSP文件。

编译的过程包括三个步骤：

* 解析JSP文件。
* 将JSP文件转为servlet。
* 编译servlet。

## JSP初始化

容器载入JSP文件后，它会在为请求提供任何服务前调用jspInit()方法。如果您需要执行自定义的JSP初始化任务，复写jspInit()方法就行了，就像下面这样：

public void jspInit(){

// 初始化代码

}

一般来讲程序只初始化一次，servlet也是如此。通常情况下您可以在jspInit()方法中初始化数据库连接、打开文件和创建查询表。

## JSP执行

这一阶段描述了JSP生命周期中一切与请求相关的交互行为，直到被销毁。

当JSP网页完成初始化后，JSP引擎将会调用\_jspService()方法。

\_jspService()方法需要一个HttpServletRequest对象和一个HttpServletResponse对象作为它的参数，就像下面这样：

void \_jspService(HttpServletRequest request,

HttpServletResponse response)

{

// 服务端处理代码

}

\_jspService()方法在每个request中被调用一次并且负责产生与之相对应的response，并且它还负责产生所有7个HTTP方法的回应，比如GET、POST、DELETE等等。

## JSP清理

JSP生命周期的销毁阶段描述了当一个JSP网页从容器中被移除时所发生的一切。

jspDestroy()方法在JSP中等价于servlet中的销毁方法。当您需要执行任何清理工作时复写jspDestroy()方法，比如释放数据库连接或者关闭文件夹等等。

jspDestroy()方法的格式如下：

public void jspDestroy()

{

// 清理代码

}

### 实例

JSP生命周期代码实例如下所示：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<html>

<head>

<title>life.jsp</title>

</head>

<body>

<%!

private int initVar=0;

private int serviceVar=0;

private int destroyVar=0;

%>

<%!

public void jspInit(){

initVar++;

System.out.println("jspInit(): JSP被初始化了"+initVar+"次");

}

public void jspDestroy(){

destroyVar++;

System.out.println("jspDestroy(): JSP被销毁了"+destroyVar+"次");

}

%>

<%

serviceVar++;

System.out.println("\_jspService(): JSP共响应了"+serviceVar+"次请求");

String content1="初始化次数 : "+initVar;

String content2="响应客户请求次数 : "+serviceVar;

String content3="销毁次数 : "+destroyVar;

%>

<h1>菜鸟教程 JSP 测试实例</h1>

<p><%=content1 %></p>

<p><%=content2 %></p>

<p><%=content3 %></p>

</body>

</html>

浏览器打开该页面，输出结果为：



# JSP 语法

本小节将会简单地介绍一下JSP开发中的基础语法。

## 脚本程序

（任意的java代码或者xml格式）

脚本程序可以包含任意量的Java语句、变量、方法或表达式，只要它们在脚本语言中是有效的。

脚本程序的语法格式：

<%代码片段 %>

或者，您也可以编写与其等价的XML语句，就像下面这样：

<jsp:scriptlet>

代码片段

</jsp:scriptlet>

任何文本、HTML标签、JSP元素必须写在脚本程序的外面。

下面给出一个示例，同时也是本教程的第一个JSP示例：

<html>

<head><title>Hello World</title></head>

<body>

Hello World!<br/>

<%

out.println("Your IP address is "+ request.getRemoteAddr());

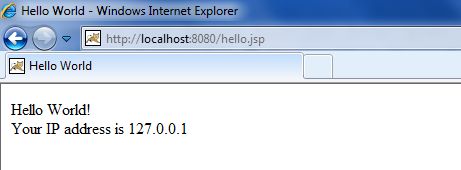
%>

</body>

</html>

**注意：**请确保Apache Tomcat已经安装在C:\apache-tomcat-7.0.2目录下并且运行环境已经正确设置。

将以上代码保存在hello.jsp中，然后将它放置在 C:\apache-tomcat-7.0.2\webapps\ROOT目录下，打开浏览器并在地址栏中输入http://localhost:8080/hello.jsp。运行后得到以下结果：



### 中文编码问题

如果我们要在页面正常显示中文，我们需要在 JSP 文件头部添加以下代码：<>

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

接下来我们将以上程序修改为：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

Hello World!<br/>

<%

out.println("你的 IP 地址 "+ request.getRemoteAddr());

%>

</body>

</html>

这样中文就可以正常显示了。

## JSP声明

一个声明语句可以声明一个或多个变量、方法，供后面的Java代码使用。在JSP文件中，您必须先声明这些变量和方法然后才能使用它们。

JSP声明的语法格式：

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

或者在普通的语法中声明也可以

<% declaration; [declaration;]%>

或者，您也可以编写与其等价的XML语句，就像下面这样：

<jsp:declaration>

代码片段

</jsp:declaration>

程序示例：

<%!int i =0; %>

<%!int a, b, c; %>

<%!Circle a =newCircle(2.0); %>

## JSP表达式

一个JSP表达式中包含的脚本语言表达式，先被转化成String，然后插入到表达式出现的地方。

由于表达式的值会被转化成String，所以您可以在一个文本行中使用表达式而不用去管它是否是HTML标签。

表达式元素中可以包含任何符合Java语言规范的表达式，但是不能使用分号来结束表达式。

JSP表达式的语法格式：

<%=表达式 %> //这样就不需输出了。表达式的结果将转换为字符串并填充到该表达式出现地地方

同样，您也可以编写与之等价的XML语句：

<jsp:expression>

表达式

</jsp:expression>

程序示例：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<p>

今天的日期是: <%=(newjava.util.Date()).toLocaleString()%>

</p>

</body>

</html>

运行后得到以下结果：

今天的日期是:2016-6-2513:40:07

## JSP注释

JSP注释主要有两个作用：为代码作注释以及将某段代码注释掉。

JSP注释的语法格式：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<%--该部分注释在网页中不会被显示--%>

<p>

今天的日期是: <%=(newjava.util.Date()).toLocaleString()%>

</p>

</body>

</html>

运行后得到以下结果：

今天的日期是:2016-6-2513:41:26

不同情况下使用注释的语法规则：

|  |  |
| --- | --- |
| **语法** | **描述** |
| <%-- 注释 --%> | JSP注释，注释内容不会被发送至浏览器甚至不会被编译 |
| <!--注释 --> | HTML注释，通过浏览器查看网页源代码时可以看见注释内容 |
| <\% | 代表静态<%常量 |
| %\> | 代表静态 %>常量 |
| \' | 在属性中使用的单引号 |
| \" | 在属性中使用的双引号 |

## JSP指令

JSP指令用来设置与整个JSP页面相关的属性。

JSP指令语法格式：

<%@ directive attribute="value" %>

这里有三种指令标签：

|  |  |
| --- | --- |
| **指令** | **描述** |
| <%@ page ... %> | 定义页面的依赖属性，比如脚本语言、error页面、缓存需求等等 |
| <%@ include ... %> | 包含其他文件 |
| <%@ taglib ... %> | 引入标签库的定义，可以是自定义标签 |

## JSP行为

JSP行为标签使用XML语法结构来控制servlet引擎。它能够动态插入一个文件，重用JavaBean组件，引导用户去另一个页面，为Java插件产生相关的HTML等等。

行为标签只有一种语法格式，它严格遵守XML标准：

<jsp:action\_nameattribute="value"/>

行为标签基本上是一些预先就定义好的函数，下表罗列出了一些可用的JSP行为标签：：

|  |  |
| --- | --- |
| **语法** | **描述** |
| jsp:include | 用于在当前页面中包含静态或动态资源 |
| jsp:useBean | 寻找和初始化一个JavaBean组件 |
| jsp:setProperty | 设置 JavaBean组件的值 |
| jsp:getProperty | 将 JavaBean组件的值插入到 output中 |
| jsp:forward | 从一个JSP文件向另一个文件传递一个包含用户请求的request对象 |
| jsp:plugin | 用于在生成的HTML页面中包含Applet和JavaBean对象 |
| jsp:element | 动态创建一个XML元素 |
| jsp:attribute | 定义动态创建的XML元素的属性 |
| jsp:body | 定义动态创建的XML元素的主体 |
| jsp:text | 用于封装模板数据 |

## JSP隐含对象

JSP支持九个自动定义的变量（me：类似PHP中的全局常量），江湖人称隐含对象。这九个隐含对象的简介见下表：

|  |  |
| --- | --- |
| **对象** | **描述** |
| request | **HttpServletRequest**类的实例 |
| response | **HttpServletResponse**类的实例 |
| out | **PrintWriter**类的实例，用于把结果输出至网页上 |
| session | **HttpSession**类的实例 |
| application | **ServletContext**类的实例，与应用上下文有关 |
| config | **ServletConfig**类的实例 |
| pageContext | **PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问 |
| page | 类似于Java类中的this关键字 |
| Exception | **Exception**类的对象，代表发生错误的JSP页面中对应的异常对象 |

## 控制流语句

JSP提供对Java语言的全面支持。您可以在JSP程序中使用Java API甚至建立Java代码块，包括判断语句和循环语句等等。

## 判断语句

类似PHP

If…else块，请看下面这个例子：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%!int day =3; %>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h3>IF...ELSE 实例</h3>

<%if(day ==1| day ==7){ %>

<p>今天是周末</p>

<%}else{ %>

<p>今天不是周末</p>

<%} %>

</body>

</html>

运行后得到以下结果：

IF...ELSE 实例

今天不是周末

现在来看看switch…case块，与if…else块有很大的不同，它使用out.println()，并且整个都装在脚本程序的标签中，就像下面这样：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%!int day =3; %>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h3>SWITCH...CASE 实例</h3>

<%

switch(day){

case0:

out.println("星期天");

break;

case1:

out.println("星期一");

break;

case2:

out.println("星期二");

break;

case3:

out.println("星期三");

break;

case4:

out.println("星期四");

break;

case5:

out.println("星期五");

break;

default:

out.println("星期六");

}

%>

</body>

</html>

浏览器访问，运行后得出以下结果：

SWITCH...CASE 实例

星期三

## 循环语句

在JSP程序中可以使用Java的三个基本循环类型：for，while，和 do…while。

让我们来看看for循环的例子，以下输出的不同字体大小的"菜鸟教程"：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%!int fontSize; %>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h3>For 循环实例</h3>

<%for( fontSize=1; fontSize <=3; fontSize++){ %>

<font color="green" size="<%= fontSize %>">

菜鸟教程

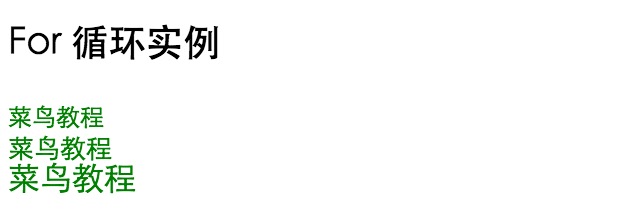
</font><br/>

<%}%>

</body>

</html>

运行后得到以下结果：



将上例改用while循环来写：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%!int fontSize; %>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h3>While 循环实例</h3>

<%while( fontSize<=3){ %>

<font color="green" size="<%= fontSize %>">

菜鸟教程

</font><br/>

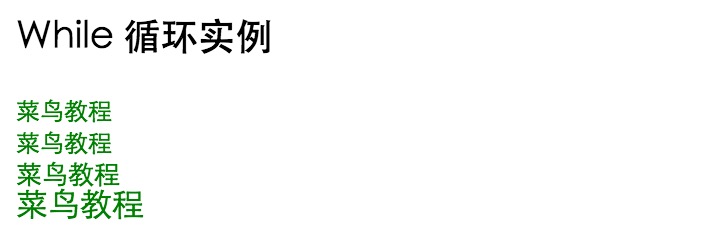
<%fontSize++;%>

<%}%>

</body>

</html>

浏览器访问，输出结果为（fontSize 初始化为0，所以多输出了一行）：



JSP运算符

JSP支持所有Java逻辑和算术运算符。

下表罗列出了JSP常见运算符，优先级从高到底：

|  |  |  |
| --- | --- | --- |
| **类别** | **操作符** | **结合性** |
| 后缀 | () [] . (点运算符) | 左到右 |
| 一元 | ++ - - ! ~ | 右到左 |
| 可乘性 | \* / % | 左到右 |
| 可加性 | + - | 左到右 |
| 移位 | >>>>><< | 左到右 |
| 关系 | >>= <<= | 左到右 |
| 相等/不等 | == != | 左到右 |
| 位与 | & | 左到右 |
| 位异或 | ^ | 左到右 |
| 位或 | | | 左到右 |
| 逻辑与 | && | 左到右 |
| 逻辑或 | || | 左到右 |
| 条件判断 | ?: | 右到左 |
| 赋值 | = += -= \*= /= %= >>= <<= &= ^= |= | 右到左 |
| 逗号 | , | 左到右 |

## JSP 字面量

JSP语言定义了以下几个字面量：

* 布尔值(boolean)：true 和 false;
* 整型(int)：与 Java 中的一样;
* 浮点型(float)：与 Java 中的一样;
* 字符串(string)：以单引号或双引号开始和结束;
* Null：null。

**JSP 指令**

JSP指令用来设置整个JSP页面相关的属性，如网页的编码方式和脚本语言。

语法格式如下：

<%@ directive attribute="value" %>

指令可以有很多个属性，它们以键值对的形式存在，并用逗号隔开。

JSP中的三种指令标签：

|  |  |
| --- | --- |
| **指令** | **描述** |
| <%@ page ... %> | 定义网页依赖属性，比如脚本语言、error页面、缓存需求等等 |
| <%@ include ... %> | 包含其他文件 |
| <%@ taglib ... %> | 引入标签库的定义 |

**Page指令**

Page指令为容器提供当前页面的使用说明。一个JSP页面可以包含多个page指令。

Page指令的语法格式：

<%@ page attribute="value" %>

等价的XML格式：

<jsp:directive.pageattribute="value"/>

**属性**

下表列出与Page指令相关的属性：

|  |  |
| --- | --- |
| **属性** | **描述** |
| buffer | 指定out对象使用缓冲区的大小 |
| autoFlush | 控制out对象的 缓存区 |
| contentType | 指定当前JSP页面的MIME类型和字符编码 |
| errorPage | 指定当JSP页面发生异常时（一般在jsp初始化后，如果发生异常就会自动跳转到这个错误页面）需要转向的错误处理页面 |
| isErrorPage | 指定当前页面是否可以作为另一个JSP页面的错误处理页面 |
| extends | 指定servlet从哪一个类继承（类似C#的inherit） |
| import | 导入要使用的Java类 |
| info | 定义JSP页面的描述信息 |
| isThreadSafe | 指定对JSP页面的访问是否为线程安全 |
| language | 定义JSP页面所用的脚本语言，默认是Java |
| session | 指定JSP页面是否使用session |
| isELIgnored | 指定是否执行EL表达式 |
| isScriptingEnabled | 确定脚本元素能否被使用 |

**Include指令**

JSP可以通过include指令来包含其他文件。被包含的文件可以是JSP文件、HTML文件或文本文件。包含的文件就好像是该JSP文件的一部分，会被同时编译执行。

Include指令的语法格式如下：

<%@ include file="文件相对 url 地址" %>

**include** 指令中的文件名实际上是一个相对的 URL 地址。

如果您没有给文件关联一个路径，JSP编译器默认在当前路径下寻找。

等价的XML语法：

<jsp:directive.includefile="文件相对 url 地址"/>

**Taglib指令**

JSP API允许用户自定义标签，一个自定义标签库就是自定义标签的集合。

Taglib指令引入一个自定义标签集合的定义，包括库路径、自定义标签。

Taglib指令的语法：

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

uri属性确定标签库的位置，prefix属性指定标签库的前缀。

等价的XML语法：

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

# JSP 动作元素

与JSP指令元素不同的是，JSP动作元素在请求处理阶段起作用。JSP动作元素是用XML语法写成的。

利用JSP动作可以动态地插入文件、重用JavaBean组件、把用户重定向到另外的页面、为Java插件生成HTML代码。

动作元素只有一种语法，它符合XML标准：

<jsp:action\_nameattribute="value"/>

动作元素基本上都是预定义的函数，JSP规范定义了一系列的标准动作，它用JSP作为前缀，可用的标准动作元素如下：

|  |  |
| --- | --- |
| **语法** | **描述** |
| jsp:include | 在页面被请求的时候引入一个文件。 |
| jsp:useBean | 寻找或者实例化一个JavaBean。 |
| jsp:setProperty | 设置JavaBean的属性。 |
| jsp:getProperty | 输出某个JavaBean的属性。 |
| jsp:forward | 把请求转到一个新的页面。 |
| jsp:plugin | 根据浏览器类型为Java插件生成OBJECT或EMBED标记。 |
| jsp:element | 定义动态XML元素 |
| jsp:attribute | 设置动态定义的XML元素属性。 |
| jsp:body | 设置动态定义的XML元素内容。 |
| jsp:text | 在JSP页面和文档中使用写入文本的模板 |

## 常见的属性

所有的动作要素都有两个属性：id属性和scope属性。

* **id属性：**

id属性是动作元素的唯一标识，可以在JSP页面中引用。动作元素创建的id值可以通过PageContext来调用。

* **scope属性：**

该属性用于识别动作元素的生命周期。 id属性和scope属性有直接关系，scope属性定义了相关联id对象的寿命。 scope属性有四个可能的值： (a) page, (b)request, (c)session, 和 (d) application。

## <jsp:include>动作元素

<jsp:include>动作元素用来包含静态和动态的文件。该动作把指定文件插入正在生成的页面。语法格式如下：

<jsp:include page="相对 URL 地址" flush="true"/>

　前面已经介绍过include指令，它是在JSP文件被转换成Servlet的时候引入文件，而这里的jsp:include动作不同，插入文件的时间是在页面被请求的时候。

以下是include动作相关的属性列表。

|  |  |
| --- | --- |
| **属性** | **描述** |
| page | 包含在页面中的相对URL地址。 |
| flush | 布尔属性，定义在包含资源前是否刷新缓存区。 |

### 实例

以下我们定义了两个文件 **date.jsp** 和 **main.jsp**，代码如下所示：

date.jsp文件代码：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<p>

今天的日期是: <%=(newjava.util.Date()).toLocaleString()%>

</p>

main.jsp文件代码：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h2>include 动作实例</h2>

<jsp:includepage="date.jsp"flush="true"/>

</body>

</html>

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

include 动作实例

今天的日期是:2016-6-2514:08:17

## <jsp:useBean>动作元素

**jsp:useBean** 动作用来加载（实例化）一个将在JSP页面中使用的JavaBean。

这个功能非常有用，因为它使得我们可以发挥 Java 组件复用的优势。

jsp:useBean动作最简单的语法为：

<jsp:useBeanid="name"class="package.class"/>

在类载入后，我们既可以通过 jsp:setProperty 和 jsp:getProperty 动作来修改和检索bean的属性。

以下是useBean动作相关的属性列表。

|  |  |
| --- | --- |
| **属性** | **描述** |
| class | 指定Bean的完整包名。 |
| type | 指定将引用该对象变量的类型。 |
| beanName | 通过 java.beans.Beans 的 instantiate() 方法指定Bean的名字。 |

在给出具体实例前，让我们先来看下 jsp:setProperty 和 jsp:getProperty 动作元素：

## <jsp:setProperty>动作元素

jsp:setProperty用来设置已经实例化的Bean对象的属性，有两种用法。首先，你可以在jsp:useBean元素的外面（后面）使用jsp:setProperty，如下所示：

<jsp:useBeanid="myName" ... />

...

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

此时，不管jsp:useBean是找到了一个现有的Bean，还是新创建了一个Bean实例，jsp:setProperty都会执行。第二种用法是把jsp:setProperty放入jsp:useBean元素的内部，如下所示：

<jsp:useBeanid="myName" ... >

...

<jsp:setPropertyname="myName"property="someProperty" .../>

</jsp:useBean>

此时，jsp:setProperty只有在新建Bean实例时才会执行，如果是使用现有实例则不执行jsp:setProperty。

jsp:setProperty动作有下面四个属性,如下表：

|  |  |
| --- | --- |
| **属性** | **描述** |
| name | name属性是必需的。它表示要设置属性的是哪个Bean。 |
| property | property属性是必需的。它表示要设置哪个属性。有一个特殊用法：如果property的值是"\*"，表示所有名字和Bean属性名字匹配的请求参数都将被传递给相应的属性set方法。 |
| value | value 属性是可选的。该属性用来指定Bean属性的值。字符串数据会在目标类中通过标准的valueOf方法自动转换成数字、boolean、Boolean、 byte、Byte、char、Character。例如，boolean和Boolean类型的属性值（比如"true"）通过Boolean.valueOf转换，int和Integer类型的属性值（比如"42"）通过Integer.valueOf转换。　　value和param不能同时使用，但可以使用其中任意一个。 |
| param | param 是可选的。它指定用哪个请求参数作为Bean属性的值。如果当前请求没有参数，则什么事情也不做，系统不会把null传递给Bean属性的set方法。因此，你可以让Bean自己提供默认属性值，只有当请求参数明确指定了新值时才修改默认属性值。 |

## <jsp:getProperty>动作元素

jsp:getProperty动作提取指定Bean属性的值，转换成字符串，然后输出。语法格式如下：

<jsp:useBeanid="myName" ... />

...

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

下表是与getProperty相关联的属性：

|  |  |
| --- | --- |
| **属性** | **描述** |
| name | 要检索的Bean属性名称。Bean必须已定义。 |
| property | 表示要提取Bean属性的值 |

### 实例

以下实例我们使用了Bean:

packagecom.runoob.main;

publicclassTestBean{

privateString message ="菜鸟教程";

public String getMessage(){

return(message);

}

Public void setMessage(String message){

this.message= message;

}

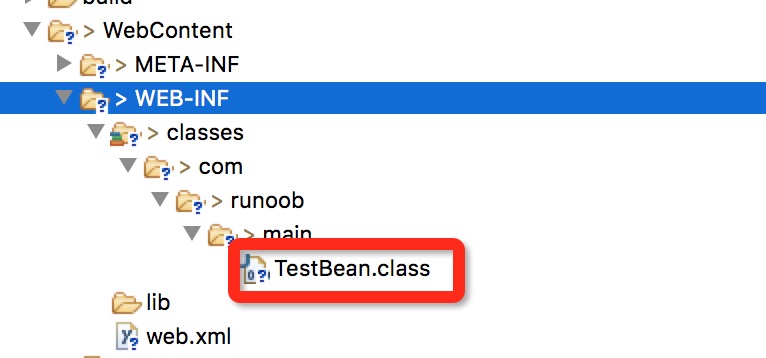
}

编译以上实例文件 TestBean.java ：

$ javac TestBean.java

编译完成后会在当前目录下生成一个 **TestBean.class** 文件，将该文件拷贝至当前 JSP 项目的 **WebContent/WEB-INF/classes/com/runoob/main** 下( com/runoob/main 包路径，没有需要手动创建)。

下面是一个 Eclipse 中目录结构图：



下面是一个很简单的例子，它的功能是装载一个Bean，然后设置/读取它的message属性。

现在让我们在main.jsp文件中调用该Bean:

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h2>Jsp 使用 JavaBean 实例</h2>

<jsp:useBean id="test" class="com.runoob.main.TestBean"/>

<jsp:setPropertyname="test"

property="message"

value="菜鸟教程..."/>

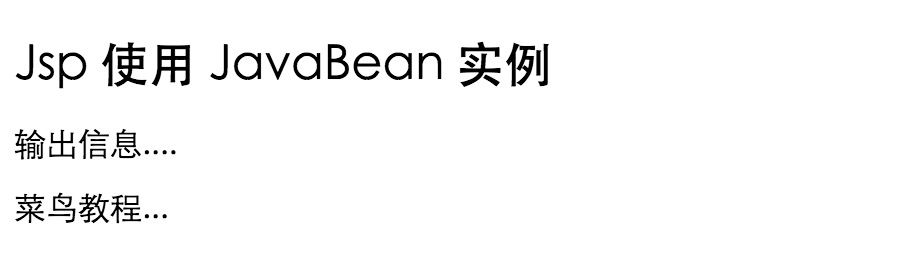
<p>输出信息....</p>

<jsp:getPropertyname="test"property="message"/>

</body>

</html>

浏览器访问，执行以上文件，输出如下所示：



## <jsp:forward>动作元素

　jsp:forward动作把请求转到另外的页面。jsp:forward标记只有一个属性page。语法格式如下所示：

<jsp:forwardpage="相对 URL 地址"/>

以下是forward相关联的属性：

|  |  |
| --- | --- |
| **属性** | **描述** |
| page | page属性包含的是一个相对URL。page的值既可以直接给出，也可以在请求的时候动态计算，可以是一个JSP页面或者一个 Java Servlet. |

### 实例

以下实例我们使用了两个文件，分别是： date.jsp 和 main.jsp。

date.jsp 文件代码如下：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<p>

今天的日期是: <%=(newjava.util.Date()).toLocaleString()%>

</p>

main.jsp文件代码：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<metacharset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h2>forward 动作实例</h2>

<jsp:forwardpage="date.jsp"/>

</body>

</html>

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

今天的日期是:2016-6-2514:37:25

## <jsp:plugin>动作元素

jsp:plugin动作用来根据浏览器的类型，插入通过Java插件运行Java Applet所必需的OBJECT或EMBED元素。

如果需要的插件不存在，它会下载插件，然后执行Java组件。 Java组件可以是一个applet或一个JavaBean。

plugin动作有多个对应HTML元素的属性用于格式化Java 组件。param元素可用于向Applet 或 Bean 传递参数。

以下是使用plugin 动作元素的典型实例:

<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>

如果你有兴趣可以尝试使用applet来测试jsp:plugin动作元素，<fallback>元素是一个新元素，在组件出现故障的错误是发送给用户错误信息。

## <jsp:element>、<jsp:attribute>、<jsp:body>动作元素

<jsp:element>、<jsp:attribute>、<jsp:body>动作元素动态定义XML元素。动态是非常重要的，这就意味着XML元素在编译时是动态生成的而非静态。

以下实例动态定义了XML元素：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<jsp:element name="xmlElement">

<jsp:attribute name="xmlElementAttr">

属性值

</jsp:attribute>

<jsp:body>

XML 元素的主体

</jsp:body>

</jsp:element>

</body>

</html>

浏览器访问以下页面，输出结果如下所示：



## <jsp:text>动作元素

<jsp:text>动作元素允许在JSP页面和文档中使用写入文本的模板，语法格式如下：

<jsp:text>模板数据</jsp:text>

以上文本模板不能包含其他元素，只能包含文本和EL表达式（注：EL表达式将在后续章节中介绍）。请注意，在XML文件中，您不能使用表达式如 ${whatever > 0}，因为>符号是非法的。你可以使用 ${whatever gt 0}表达式或者嵌入在一个CDATA部分的值。

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

如果你需要在 XHTML 中声明 DOCTYPE,必须使用到<jsp:text>动作元素，实例如下：

<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>

你可以对以上实例尝试使用<jsp:text>及不使用该动作元素执行结果的区别。

**JSP 隐式对象**

JSP隐式对象是JSP容器为每个页面提供的Java对象，开发者可以直接使用它们而不用显式声明。JSP隐式对象也被称为预定义变量。

JSP所支持的九大隐式对象：

|  |  |
| --- | --- |
| **对象** | **描述** |
| request | **HttpServletRequest**类的实例 |
| response | **HttpServletResponse**类的实例 |
| out | **JspWriter**类的实例，用于把结果输出至网页上 |
| session | **HttpSession**类的实例 |
| application | **ServletContext**类的实例，与应用上下文有关 |
| config | **ServletConfig**类的实例 |
| pageContext | **PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问 |
| page | 类似于Java类中的this关键字 |
| Exception | **Exception**类的对象，代表发生错误的JSP页面中对应的异常对象 |

**request对象**

request对象是javax.servlet.http.HttpServletRequest 类的实例。每当客户端请求一个JSP页面时，JSP引擎就会制造一个新的request对象来代表这个请求。

request对象提供了一系列方法来获取HTTP头信息，cookies，HTTP方法等等。

**response对象**

response对象是javax.servlet.http.HttpServletResponse类的实例。当服务器创建request对象时会同时创建用于响应这个客户端的response对象。

response对象也定义了处理HTTP头模块的接口。通过这个对象，开发者们可以添加新的cookies，时间戳，HTTP状态码等等。

**out对象**

out对象是 javax.servlet.jsp.JspWriter 类的实例，用来在response对象中写入内容。

最初的JspWriter类对象根据页面是否有缓存来进行不同的实例化操作。可以在page指令中使用buffered='false'属性来轻松关闭缓存。

JspWriter类包含了大部分java.io.PrintWriter类中的方法。不过，JspWriter新增了一些专为处理缓存而设计的方法。还有就是，JspWriter类会抛出IOExceptions异常，而PrintWriter不会。

下表列出了我们将会用来输出boolean，char，int，double，String，object等类型数据的重要方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| **out.print(dataType dt)** | 输出Type类型的值 |
| **out.println(dataType dt)** | 输出Type类型的值然后换行 |
| **out.flush()** | 刷新输出流 |

**session对象**

session对象是 javax.servlet.http.HttpSession 类的实例。和Java Servlets中的session对象有一样的行为。

session对象用来跟踪在各个客户端请求间的会话。

**application对象**

application对象直接包装了servlet的ServletContext类的对象，是javax.servlet.ServletContext 类的实例。

这个对象在JSP页面的整个生命周期中都代表着这个JSP页面。这个对象在JSP页面初始化时被创建，随着jspDestroy()方法的调用而被移除。

通过向application中添加属性，则所有组成您web应用的JSP文件都能访问到这些属性。

**config对象**

config对象是 javax.servlet.ServletConfig 类的实例，直接包装了servlet的ServletConfig类的对象。

这个对象允许开发者访问Servlet或者JSP引擎的初始化参数，比如文件路径等。

以下是config对象的使用方法，不是很重要，所以不常用：

config.getServletName();

它返回包含在<servlet-name>元素中的servlet名字，注意，<servlet-name>元素在 WEB-INF\web.xml 文件中定义。

**pageContext 对象**

pageContext对象是javax.servlet.jsp.PageContext 类的实例，用来代表整个JSP页面。

这个对象主要用来访问页面信息，同时过滤掉大部分实现细节。

这个对象存储了request对象和response对象的引用。application对象，config对象，session对象，out对象可以通过访问这个对象的属性来导出。

pageContext对象也包含了传给JSP页面的指令信息，包括缓存信息，ErrorPage URL,页面scope等。

PageContext类定义了一些字段，包括PAGE\_SCOPE，REQUEST\_SCOPE，SESSION\_SCOPE， APPLICATION\_SCOPE。它也提供了40余种方法，有一半继承自javax.servlet.jsp.JspContext 类。

其中一个重要的方法就是removeArribute()，它可接受一个或两个参数。比如，pageContext.removeArribute("attrName")移除四个scope中相关属性，但是下面这种方法只移除特定scope中的相关属性：

pageContext.removeAttribute("attrName", PAGE\_SCOPE);

**page 对象**

这个对象就是页面实例的引用。它可以被看做是整个JSP页面的代表。

page 对象就是this对象的同义词。

**exception 对象**

exception 对象包装了从先前页面中抛出的异常信息。它通常被用来产生对出错条件的适当响应。

**JSP 客户端请求**

当浏览器请求一个网页时，它会向网络服务器发送一系列不能被直接读取的信息，因为这些信息是作为HTTP信息头的一部分来传送的。您可以查阅HTTP协议来获得更多的信息。

下表列出了浏览器端信息头的一些重要内容，在以后的网络编程中将会经常见到这些信息：

|  |  |
| --- | --- |
| **信息** | **描述** |
| Accept | 指定浏览器或其他客户端可以处理的MIME类型。它的值通常为 **image/png** 或 **image/jpeg** |
| Accept-Charset | 指定浏览器要使用的字符集。比如 ISO-8859-1 |
| Accept-Encoding | 指定编码类型。它的值通常为 **gzip** 或**compress** |
| Accept-Language | 指定客户端首选语言，servlet会优先返回以当前语言构成的结果集，如果servlet支持这种语言的话。比如 en，en-us，ru等等 |
| Authorization | 在访问受密码保护的网页时识别不同的用户 |
| Connection | 表明客户端是否可以处理HTTP持久连接。持久连接允许客户端或浏览器在一个请求中获取多个文件。**Keep-Alive** 表示启用持久连接 |
| Content-Length | 仅适用于POST请求，表示 POST 数据的字节数 |
| Cookie | 返回先前发送给浏览器的cookies至服务器 |
| Host | 指出原始URL中的主机名和端口号 |
| If-Modified-Since | 表明只有当网页在指定的日期被修改后客户端才需要这个网页。 服务器发送304码给客户端，表示没有更新的资源 |
| If-Unmodified-Since | 与If-Modified-Since相反， 只有文档在指定日期后仍未被修改过，操作才会成功 |
| Referer | 标志着所引用页面的URL。比如，如果你在页面1，然后点了个链接至页面2，那么页面1的URL就会包含在浏览器请求页面2的信息头中 |
| User-Agent | 用来区分不同浏览器或客户端发送的请求，并对不同类型的浏览器返回不同的内容 |

**HttpServletRequest类**

request对象是javax.servlet.http.HttpServletRequest类的实例。每当客户端请求一个页面时，JSP引擎就会产生一个新的对象来代表这个请求。

request对象提供了一系列方法来获取HTTP信息头，包括表单数据，cookies，HTTP方法等等。

接下来将会介绍一些在JSP编程中常用的获取HTTP信息头的方法。详细内容请见下表：

|  |  |
| --- | --- |
| **序号** | **方法& 描述** |
| 1 | **Cookie[] getCookies()**  返回客户端所有的Cookie的数组 |
| 2 | **Enumeration getAttributeNames()**  返回request对象的所有属性名称的集合 |
| 3 | **Enumeration getHeaderNames()**  返回所有HTTP头的名称集合 |
| 4 | **Enumeration getParameterNames()**  返回请求中所有参数的集合 |
| 5 | **HttpSession getSession()**  返回request对应的session对象，如果没有，则创建一个 |
| 6 | **HttpSession getSession(boolean create)**  返回request对应的session对象，如果没有并且参数create为true，则返回一个新的session对象 |
| 7 | **Locale getLocale()**  返回当前页的Locale对象，可以在response中设置 |
| 8 | **Object getAttribute(String name)**  返回名称为name的属性值，如果不存在则返回null。 |
| 9 | **ServletInputStream getInputStream()**  返回请求的输入流 |
| 10 | **String getAuthType()**  返回认证方案的名称，用来保护servlet，比如 "BASIC" 或者 "SSL" 或 null 如果 JSP没设置保护措施 |
| 11 | **String getCharacterEncoding()**  返回request的字符编码集名称 |
| 12 | **String getContentType()**  返回request主体的MIME类型，若未知则返回null |
| 13 | **String getContextPath()**  返回request URI中指明的上下文路径 |
| 14 | **String getHeader(String name)**  返回name指定的信息头 |
| 15 | **String getMethod()**  返回此request中的HTTP方法，比如 GET,，POST，或PUT |
| 16 | **String getParameter(String name)**  返回此request中name指定的参数，若不存在则返回null |
| 17 | **String getPathInfo()**  返回任何额外的与此request URL相关的路径 |
| 18 | **String getProtocol()**  返回此request所使用的协议名和版本 |
| 19 | **String getQueryString()**  返回此 request URL包含的查询字符串 |
| 20 | **String getRemoteAddr()**  返回客户端的IP地址 |
| 21 | **String getRemoteHost()**  返回客户端的完整名称 |
| 22 | **String getRemoteUser()**  返回客户端通过登录认证的用户，若用户未认证则返回null |
| 23 | **String getRequestURI()**  返回request的URI |
| 24 | **String getRequestedSessionId()**  返回request指定的session ID |
| 25 | **String getServletPath()**  返回所请求的servlet路径 |
| 26 | **String[] getParameterValues(String name)**  返回指定名称的参数的所有值，若不存在则返回null |
| 27 | **boolean isSecure()**  返回request是否使用了加密通道，比如HTTPS |
| 28 | **int getContentLength()**  返回request主体所包含的字节数，若未知的返回-1 |
| 29 | **int getIntHeader(String name)**  返回指定名称的request信息头的值 |
| 30 | **int getServerPort()**  返回服务器端口号 |

**HTTP信息头示例**

在这个例子中，我们会使用HttpServletRequest类的getHeaderNames()方法来读取HTTP信息头。这个方法以枚举的形式返回当前HTTP请求的头信息。

获取Enumeration对象后，用标准的方式来遍历Enumeration对象，用hasMoreElements()方法来确定什么时候停止，用nextElement()方法来获得每个参数的名字。

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h2>HTTP 头部请求实例</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");

}

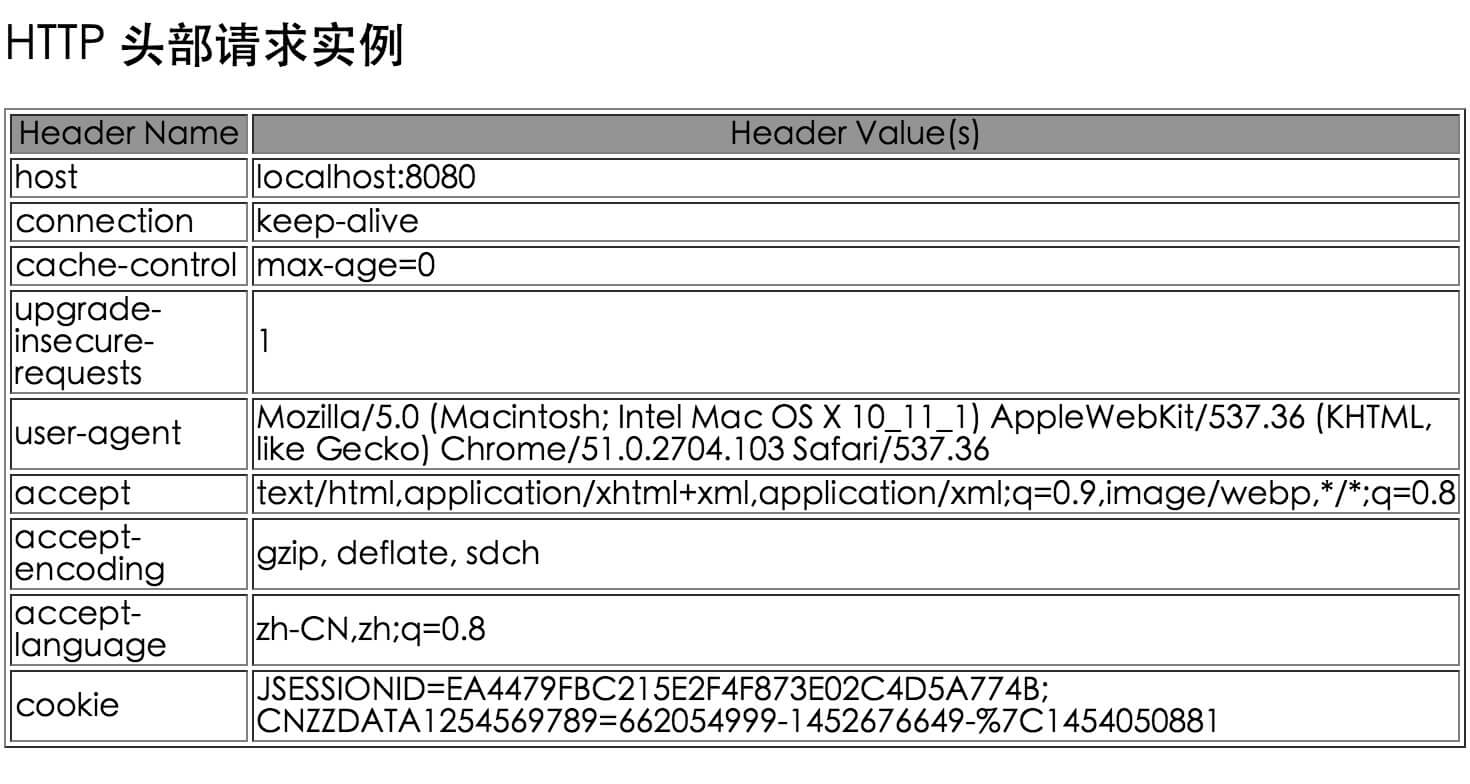
%>

</table>

</body>

</html>

访问main.jsp，将会得到以下结果：



您可以在上面代码中尝试HttpServletRequest类的其它方法。

**JSP 服务器响应**

Response响应对象主要将JSP容器处理后的结果传回到客户端。可以通过response变量设置HTTP的状态和向客户端发送数据，如Cookie、HTTP文件头信息等。

一个典型的响应看起来就像下面这样：

HTTP/1.1 200 OK

Content-Type: text/html

Header2: ...

...

HeaderN: ...

(空行)

<!doctype ...>

<html>

<head>...</head>

<body>

...

</body>

</html>

状态行包含HTTP版本信息，比如HTTP/1.1，一个状态码，比如200，还有一个非常短的信息对应着状态码，比如OK。

下表摘要出了HTTP1.1响应头中最有用的部分，在网络编程中您将会经常见到它们：

|  |  |
| --- | --- |
| **响应头** | **描述** |
| Allow | 指定服务器支持的request方法（GET，POST等等） |
| Cache-Control | 指定响应文档能够被安全缓存的情况。通常取值为 **public，private** 或**no-cache** 等等。 Public意味着文档可缓存，Private意味着文档只为单用户服务并且只能使用私有缓存。No-cache 意味着文档不被缓存。 |
| Connection | 命令浏览器是否要使用持久的HTTP连接。**close值** 命令浏览器不使用持久HTTP连接，而keep-alive 意味着使用持久化连接。 |
| Content-Disposition | 让浏览器要求用户将响应以给定的名称存储在磁盘中 |
| Content-Encoding | 指定传输时页面的编码规则 |
| Content-Language | 表述文档所使用的语言，比如en， en-us,，ru等等 |
| Content-Length | 表明响应的字节数。只有在浏览器使用持久化 (keep-alive) HTTP 连接时才有用 |
| Content-Type | 表明文档使用的MIME类型 |
| Expires | 指明啥时候过期并从缓存中移除 |
| Last-Modified | 指明文档最后修改时间。客户端可以 缓存文档并且在后续的请求中提供一个 **If-Modified-Since**请求头 |
| Location | 在300秒内，包含所有的有一个状态码的响应地址，浏览器会自动重连然后检索新文档 |
| Refresh | 指明浏览器每隔多久请求更新一次页面。 |
| Retry-After | 与503 (Service Unavailable)一起使用来告诉用户多久后请求将会得到响应 |
| Set-Cookie | 指明当前页面对应的cookie |

**HttpServletResponse类**

response 对象是 javax.servlet.http.HttpServletResponse 类的一个实例。就像服务器会创建request对象一样，它也会创建一个客户端响应。

response对象定义了处理创建HTTP信息头的接口。通过使用这个对象，开发者们可以添加新的cookie或时间戳，还有HTTP状态码等等。

下表列出了用来设置HTTP响应头的方法，这些方法由HttpServletResponse 类提供：

|  |  |
| --- | --- |
| **S.N.** | **方法 & 描述** |
| 1 | **String encodeRedirectURL(String url)**  对sendRedirect()方法使用的URL进行编码 |
| 2 | **String encodeURL(String url)**  将URL编码，回传包含Session ID的URL |
| 3 | **boolean containsHeader(String name)**  返回指定的响应头是否存在 |
| 4 | **boolean isCommitted()**  返回响应是否已经提交到客户端 |
| 5 | **void addCookie(Cookie cookie)**  添加指定的cookie至响应中 |
| 6 | **void addDateHeader(String name, long date)**  添加指定名称的响应头和日期值 |
| 7 | **void addHeader(String name, String value)**  添加指定名称的响应头和值 |
| 8 | **void addIntHeader(String name, int value)**  添加指定名称的响应头和int值 |
| 9 | **void flushBuffer()**  将任何缓存中的内容写入客户端 |
| 10 | **void reset()**  清除任何缓存中的任何数据，包括状态码和各种响应头 |
| 11 | **void resetBuffer()**  清除基本的缓存数据，不包括响应头和状态码 |
| 12 | **void sendError(int sc)**  使用指定的状态码向客户端发送一个出错响应，然后清除缓存 |
| 13 | **void sendError(int sc, String msg)**  使用指定的状态码和消息向客户端发送一个出错响应 |
| 14 | **void sendRedirect(String location)**  使用指定的URL向客户端发送一个临时的间接响应 |
| 15 | **void setBufferSize(int size)**  设置响应体的缓存区大小 |
| 16 | **void setCharacterEncoding(String charset)**  指定响应的编码集（MIME字符集），例如UTF-8 |
| 17 | **void setContentLength(int len)**  指定HTTP servlets中响应的内容的长度，此方法用来设置 HTTP Content-Length 信息头 |
| 18 | **void setContentType(String type)**  设置响应的内容的类型，如果响应还未被提交的话 |
| 19 | **void setDateHeader(String name, long date)**  使用指定名称和值设置响应头的名称和内容 |
| 20 | **void setHeader(String name, String value)**  使用指定名称和值设置响应头的名称和内容 |
| 21 | **void setIntHeader(String name, int value)**  使用指定名称和值设置响应头的名称和内容 |
| 22 | **void setLocale(Locale loc)**  设置响应的语言环境，如果响应尚未被提交的话 |
| 23 | **void setStatus(int sc)**  设置响应的状态码 |

**HTTP响应头程序示例**

接下来的例子使用setIntHeader()方法和setRefreshHeader()方法来模拟一个数字时钟：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h2>自动刷新实例</h2>

<%

// 设置每隔5秒自动刷新

response.setIntHeader("Refresh", 5);

// 获取当前时间

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("当前时间: " + CT + "\n");

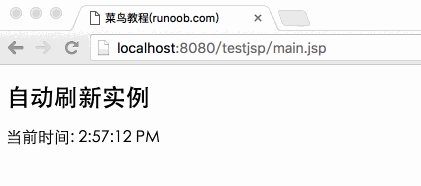
%>

</body>

</html>

将以上代码保存为main.jsp，然后通过浏览器访问它。它将会每隔5秒显示一下系统当前时间。

我们可以看下以下 Gif 演示图：



您也可以自己动手修改以上代码，试试使用其他的方法，将能得到更深的体会。

**JSP HTTP 状态码**

HTTP请求与HTTP响应的格式相近，都有着如下结构：

* 以状态行+CRLF（回车换行）开始
* 零行或多行头模块+CRLF
* 一个空行，比如CRLF
* 可选的消息体比如文件，查询数据，查询输出

举例来说，一个服务器响应头看起来就像下面这样：

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，一个状态码，和状态码相对应的短消息。

下表列出了可能会从服务器返回的HTTP状态码和与之关联的消息：

|  |  |  |
| --- | --- | --- |
| **状态码** | **消息** | **描述** |
| 100 | Continue | 只有一部分请求被服务器接收，但只要没被服务器拒绝，客户端就会延续这个请求 |
| 101 | Switching Protocols | 服务器交换机协议 |
| 200 | OK | 请求被确认 |
| 201 | Created | 请求时完整的，新的资源被创建 |
| 202 | Accepted | 请求被接受，但未处理完 |
| 203 | Non-authoritative Information |  |
| 204 | No Content |  |
| 205 | Reset Content |  |
| 206 | Partial Content |  |
| 300 | Multiple Choices | 一个超链接表，用户可以选择一个超链接并访问，最大支持5个超链接 |
| 301 | Moved Permanently | 被请求的页面已经移动到了新的URL下 |
| 302 | Found | 被请求的页面暂时性地移动到了新的URL下 |
| 303 | See Other | 被请求的页面可以在一个不同的URL下找到 |
| 304 | Not Modified |  |
| 305 | Use Proxy |  |
| 306 | *Unused* | 已经不再使用此状态码，但状态码被保留 |
| 307 | Temporary Redirect | 被请求的页面暂时性地移动到了新的URL下 |
| 400 | Bad Request | 服务器无法识别请求 |
| 401 | Unauthorized | 被请求的页面需要用户名和密码 |
| 402 | Payment Required | *目前还不能使用此状态码* |
| 403 | Forbidden | 禁止访问所请求的页面 |
| 404 | Not Found | 服务器无法找到所请求的页面 |
| 405 | Method Not Allowed | 请求中所指定的方法不被允许 |
| 406 | Not Acceptable | 服务器只能创建一个客户端无法接受的响应 |
| 407 | Proxy Authentication Required | 在请求被服务前必须认证一个代理服务器 |
| 408 | Request Timeout | 请求时间超过了服务器所能等待的时间，连接被断开 |
| 409 | Conflict | 请求有矛盾的地方 |
| 410 | Gone | 被请求的页面不再可用 |
| 411 | Length Required | "Content-Length"没有被定义，服务器拒绝接受请求 |
| 412 | Precondition Failed | 请求的前提条件被服务器评估为false |
| 413 | Request Entity Too Large | 因为请求的实体太大，服务器拒绝接受请求 |
| 414 | Request-url Too Long | 服务器拒绝接受请求，因为URL太长。多出现在把"POST"请求转换为"GET"请求时所附带的大量查询信息 |
| 415 | Unsupported Media Type | 服务器拒绝接受请求，因为媒体类型不被支持 |
| 417 | Expectation Failed |  |
| 500 | Internal Server Error | 请求不完整，服务器遇见了出乎意料的状况 |
| 501 | Not Implemented | 请求不完整，服务器不提供所需要的功能 |
| 502 | Bad Gateway | 请求不完整，服务器从上游服务器接受了一个无效的响应 |
| 503 | Service Unavailable | 请求不完整，服务器暂时重启或关闭 |
| 504 | Gateway Timeout | 网关超时 |
| 505 | HTTP Version Not Supported | 服务器不支持所指定的HTTP版本 |

**设置HTTP状态码的方法**

下表列出了HttpServletResponse 类中用来设置状态码的方法：

|  |  |
| --- | --- |
| **S.N.** | **方法 & 描述** |
| 1 | **public void setStatus ( int statusCode )**  此方法可以设置任意的状态码。如果您的响应包含一个特殊的状态码和一个文档，请确保在用PrintWriter返回任何内容前调用setStatus方法 |
| 2 | **public void sendRedirect(String url)**  此方法产生302响应，同时产生一个 *Location* 头告诉URL 一个新的文档 |
| 3 | **public void sendError(int code, String message)**  此方法将一个状态码(通常为 404)和一个短消息，自动插入HTML文档中并发回给客户端 |

**HTTP状态码程序示例**

接下来的例子将会发送407错误码给浏览器，然后浏览器将会告诉您"Need authentication!!!"。

<html>

<head>

<title>Setting HTTP Status Code</title>

</head>

<body>

<%

// 设置错误代码，并说明原因

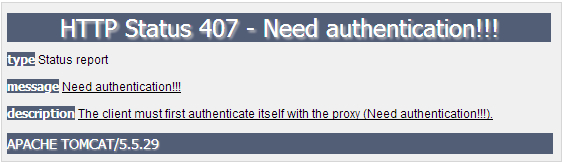
response.sendError(407, "Need authentication!!!" );

%>

</body>

</html>

访问以上JSP页面，将会得到以下结果：



您也可以试试使用其他的状态码，看会不会得到什么意想不到结果。

**JSP 表单处理**

我们在浏览网页的时候，经常需要向服务器提交信息，并让后台程序处理。浏览器中使用 GET 和 POST 方法向服务器提交数据。

**GET 方法**

GET方法将请求的编码信息添加在网址后面，网址与编码信息通过"?"号分隔。如下所示：

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

GET方法是浏览器默认传递参数的方法，一些敏感信息，如密码等建议不使用GET方法。

用get时，传输数据的大小有限制 （注意不是参数的个数有限制）。

**POST 方法**

一些敏感信息，如密码等我们可以通过POST方法传递，POST提交数据是隐式的。

POST提交数据是不可见的，GET是通过在url里面传递的（可以看一下你浏览器的地址栏）。

JSP使用getParameter()来获得传递的参数，getInputStream()方法用来处理客户端的二进制数据流的请求。

**JSP 读取表单数据**

* **getParameter():** 使用 request.getParameter() 方法来获取表单参数的值。
* **getParameterValues():** 获得如checkbox类（名字相同，但值有多个）的数据。 接收数组变量 ，如checkbox类型
* **getParameterNames():**该方法可以取得所有变量的名称，该方法返回一个Emumeration。
* **getInputStream():**调用此方法来读取来自客户端的二进制数据流。

**使用URL的 GET 方法实例**

以下是一个简单的URL,并使用GET方法来传递URL中的参数：

http://localhost:8080/testjsp/main.jsp?name=菜鸟教程&url=http://ww.runoob.com

testjsp 为项目地址。

以下是 main.jsp 文件的JSP程序用于处理客户端提交的表单数据，我们使用getParameter()方法（me：该方法可以获取get方式和post方式提交的数据）来获取提交的数据：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h1>使用 GET 方法读取数据</h1>

<ul>

<li><p><b>站点名:</b>

<%= request.getParameter("name")%>

</p></li>

<li><p><b>网址:</b>

<%= request.getParameter("url")%>

</p></li>

</ul>

</body>

</html>

接下来我们通过浏览器访问 ***http://localhost:8080/testjsp/main.jsp?name=菜鸟教程&url=http://ww.runoob.com*** 输出结果如下所示：



**使用表单的 GET 方法实例**

以下是一个简单的 HTML 表单，该表单通过GET方法将客户端数据提交 到 **main.jsp** 文件中：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<form action="main.jsp" method="GET">

站点名: <input type="text" name="name">

<br />

网址: <input type="text" name="url" />

<input type="submit" value="提交" />

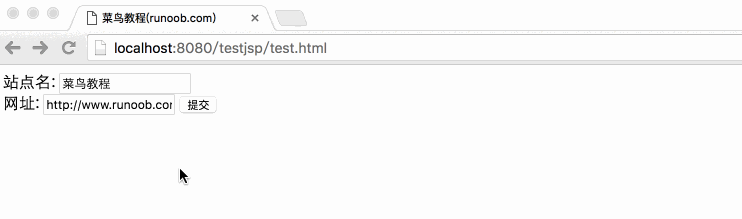
</form>

</body>

</html>

将以上HTML代码保存到test.htm文件中。 将该文件放置于当前jsp项目的 WebContent 目录下（与 main.jsp 同一个目录）。

通过访问 ***http://localhost:8080/testjsp/test.html*** 提交表单数据到 main.jsp 文件，演示 Gif 图如下所示：



在 "站点名" 与 "网址" 两个表单中填入信息，并点击 "提交" 按钮，它将输出结果。

**使用表单的 POST 方法实例**

接下来让我们使用POST方法来传递表单数据，修改main.jsp与Hello.htm文件代码，如下所示：

main.jsp文件代码：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h1>使用 POST 方法读取数据</h1>

<ul>

<li><p><b>站点名:</b>

<%

// 解决中文乱码的问题

String name = new String((request.getParameter("name")).getBytes("ISO-8859-1"),"UTF-8");

%>

<%=name%>

</p></li>

<li><p><b>网址:</b>

<%= request.getParameter("url")%>

</p></li>

</ul>

</body>

</html>

代码中我们使用 **new String((request.getParameter("name")).getBytes("ISO-8859-1"),"UTF-8")**来转换编码，防止中文乱码的发生。

以下是test.htm修改后的代码：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<form action="main.jsp" method="POST">

站点名: <input type="text" name="name">

<br />

网址: <input type="text" name="url" />

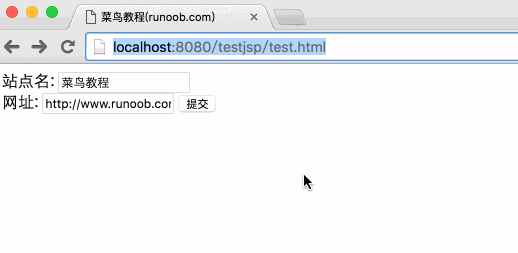
<input type="submit" value="提交" />

</form>

</body>

</html>

通过访问 ***http://localhost:8080/testjsp/test.html*** 提交表单数据到 main.jsp 文件，演示 Gif 图如下所示：



**传递 Checkbox 数据到JSP程序**

复选框 checkbox 可以传递一个甚至多个数据。

以下是一个简单的HTML代码，并将代码保存在test.htm文件中：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<form action="main.jsp" method="POST" target="\_blank">

<input type="checkbox" name="google" checked="checked" /> Google

<input type="checkbox" name="runoob" /> 菜鸟教程

<input type="checkbox" name="taobao" checked="checked" />

淘宝

<input type="submit" value="选择网站" />

</form>

</body>

</html>

以上代码在浏览器访问如下所示：

以下为main.jsp文件代码，用于处理复选框数据：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h1>从复选框中读取数据</h1>

<ul>

<li><p><b>Google 是否选中:</b>

<%= request.getParameter("google")%>

</p></li>

<li><p><b>菜鸟教程是否选中:</b>

<%= request.getParameter("runoob")%>

</p></li>

<li><p><b>淘宝是否选中:</b>

<%= request.getParameter("taobao")%>

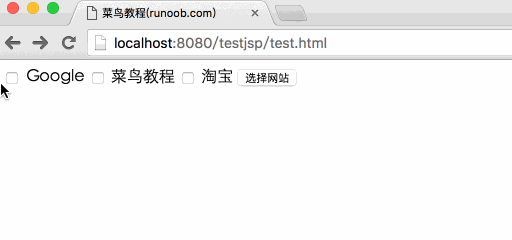
</p></li>

</ul>

</body>

</html>

通过访问 ***http://localhost:8080/testjsp/test.html*** 提交表单数据到 main.jsp 文件，演示 Gif 图如下所示：



**读取所有表单参数**

以下我们将使用 **HttpServletRequest** 的 **getParameterNames()** 来读取所有表单参数,该方法可以取得所有变量的名称，该方法返回一个枚举。

一旦我们有了一个 Enumeration（枚举），我们就可以调用 hasMoreElements() 方法来确定是否还有元素，以及使用nextElement（）方法来获得每个参数的名称。

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<h1>读取所有表单参数</h1>

<table width="100%" border="1" align="center">

<tr bgcolor="#949494">

<th>参数名</th><th>参数值</th>

</tr>

<%

Enumeration paramNames = request.getParameterNames();

while(paramNames.hasMoreElements()) {

String paramName = (String)paramNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n");

String paramValue = request.getParameter(paramName);

out.println("<td> " + paramValue + "</td></tr>\n");

}

%>

</table>

</body>

</html>

以下是test.htm文件的内容:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<form action="main.jsp" method="POST" target="\_blank">

<input type="checkbox" name="google" checked="checked" /> Google

<input type="checkbox" name="runoob" /> 菜鸟教程

<input type="checkbox" name="taobao" checked="checked" />

淘宝

<input type="submit" value="选择网站" />

</form>

</body>

</html>

现在我们通过浏览器访问 test.htm 文件提交数据，输出结果如下：

通过访问 ***http://localhost:8080/testjsp/test.html*** 提交表单数据到 main.jsp 文件，演示 Gif 图如下所示：



你可以尝试使用以上的JSP代码读取其它对象，如文本框，单选按钮或下拉框等等其他形式的数据。

# JSP 过滤器

JSP 和 Servlet 中的过滤器都是 Java 类。

过滤器可以动态地拦截请求和响应，以变换或使用包含在请求或响应中的信息。

可以将一个或多个过滤器附加到一个 Servlet 或一组 Servlet。过滤器也可以附加到 JavaServer Pages (JSP) 文件和 HTML 页面。

过滤器是可用于 Servlet 编程的 Java 类，可以实现以下目的：

* 在客户端的请求访问后端资源之前，拦截这些请求。
* 在服务器的响应发送回客户端之前，处理这些响应。

根据规范建议的各种类型的过滤器：

* 身份验证过滤器（Authentication Filters）。
* 数据压缩过滤器（Data compression Filters）。
* 加密过滤器（Encryption Filters）。
* 触发资源访问事件过滤器。
* 图像转换过滤器（Image Conversion Filters）。
* 日志记录和审核过滤器（Logging and Auditing Filters）。
* MIME-TYPE 链过滤器（MIME-TYPE Chain Filters）。
* 标记化过滤器（Tokenizing Filters）。
* XSL/T 过滤器（XSL/T Filters），转换 XML 内容。

过滤器通过 Web 部署描述符（web.xml）中的 XML 标签来声明，然后映射到您的应用程序的部署描述符中的 Servlet 名称或 URL 模式。

当 Web 容器启动 Web 应用程序时，它会为您在部署描述符中声明的每一个过滤器创建一个实例。

Filter的执行顺序与在web.xml配置文件中的配置顺序一致，一般把Filter配置在所有的Servlet之前。

## Servlet 过滤器方法

过滤器是一个实现了 javax.servlet.Filter 接口的 Java 类。javax.servlet.Filter 接口定义了三个方法：

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **public void doFilter (ServletRequest, ServletResponse, FilterChain)** 该方法完成实际的过滤操作，当客户端请求方法与过滤器设置匹配的URL时，Servlet容器将先调用过滤器的doFilter方法。FilterChain用户访问后续过滤器。 |
| 2 | **public void init(FilterConfig filterConfig)** web 应用程序启动时，web 服务器将创建Filter 的实例对象，并调用其init方法，读取web.xml配置，完成对象的初始化功能，从而为后续的用户请求作好拦截的准备工作（filter对象只会创建一次，init方法也只会执行一次）。开发人员通过init方法的参数，可获得代表当前filter配置信息的FilterConfig对象。 |
| 3 | **public void destroy()** Servlet容器在销毁过滤器实例前调用该方法，在该方法中释放Servlet过滤器占用的资源。 |

### FilterConfig 使用

Filter 的 init 方法中提供了一个 FilterConfig 对象。

如 web.xml 文件配置如下：

<filter>

<filter-name>LoginFilter</filter-name>

<filter-class>com.runoob.test.LogFilter</filter-class>

<init-param>

<param-name>Site</param-name>

<param-value>菜鸟教程</param-value>

</init-param>

</filter>

在 init 方法使用 FilterConfig 对象获取参数：

public void init(FilterConfig config) throws ServletException {

// 获取初始化参数

String site = config.getInitParameter("Site");

// 输出初始化参数

System.out.println("网站名称: " + site);

}

## JSP 过滤器实例

以下是 Servlet 过滤器的实例，将输出网站名称和地址。本实例让您对 Servlet 过滤器有基本的了解，您可以使用相同的概念编写更复杂的过滤器应用程序：

//导入必需的 java 库

import javax.servlet.\*;

import java.util.\*;

//实现 Filter 类

public class LogFilter implements Filter {

public void init(FilterConfig config) throws ServletException {

// 获取初始化参数

String site = config.getInitParameter("Site");

// 输出初始化参数

System.out.println("网站名称: " + site);

}

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws java.io.IOException, ServletException {

// 输出站点名称

System.out.println("站点网址：http://www.runoob.com");

// 把请求传回过滤链

chain.doFilter(request,response);

}

public void destroy( ){

/\* 在 Filter 实例被 Web 容器从服务移除之前调用 \*/

}

}

DisplayHeader.java 文件代码如下：

//导入必需的 java 库

import java.io.IOException;

import java.io.PrintWriter;

import java.util.Enumeration;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/DisplayHeader")

//扩展 HttpServlet 类

public class DisplayHeader extends HttpServlet {

// 处理 GET 方法请求的方法

public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException

{

// 设置响应内容类型

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

String title = "HTTP Header 请求实例 - 菜鸟教程实例";

String docType =

"<!DOCTYPE html> \n";

out.println(docType +

"<html>\n" +

"<head><meta charset=\"utf-8\"><title>" + title + "</title></head>\n"+

"<body bgcolor=\"#f0f0f0\">\n" +

"<h1 align=\"center\">" + title + "</h1>\n" +

"<table width=\"100%\" border=\"1\" align=\"center\">\n" +

"<tr bgcolor=\"#949494\">\n" +

"<th>Header 名称</th><th>Header 值</th>\n"+

"</tr>\n");

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");

}

out.println("</table>\n</body></html>");

}

// 处理 POST 方法请求的方法

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

doGet(request, response);

}

}

### Web.xml 中的 Servlet 过滤器映射（Servlet Filter Mapping）

定义过滤器，然后映射到一个 URL 或 Servlet，这与定义 Servlet，然后映射到一个 URL 模式方式大致相同。在部署描述符文件 **web.xml** 中为 filter 标签创建下面的条目：

<?xml version="1.0" encoding="UTF-8"?>

<web-app>

<filter>

<filter-name>LogFilter</filter-name>

<filter-class>com.runoob.test.LogFilter</filter-class>

<init-param>

<param-name>Site</param-name>

<param-value>菜鸟教程</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>LogFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<servlet>

<!-- 类名 -->

<servlet-name>DisplayHeader</servlet-name>

<!-- 所在的包 -->

<servlet-class>com.runoob.test.DisplayHeader</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>DisplayHeader</servlet-name>

<!-- 访问的网址 -->

<url-pattern>/TomcatTest/DisplayHeader</url-pattern>

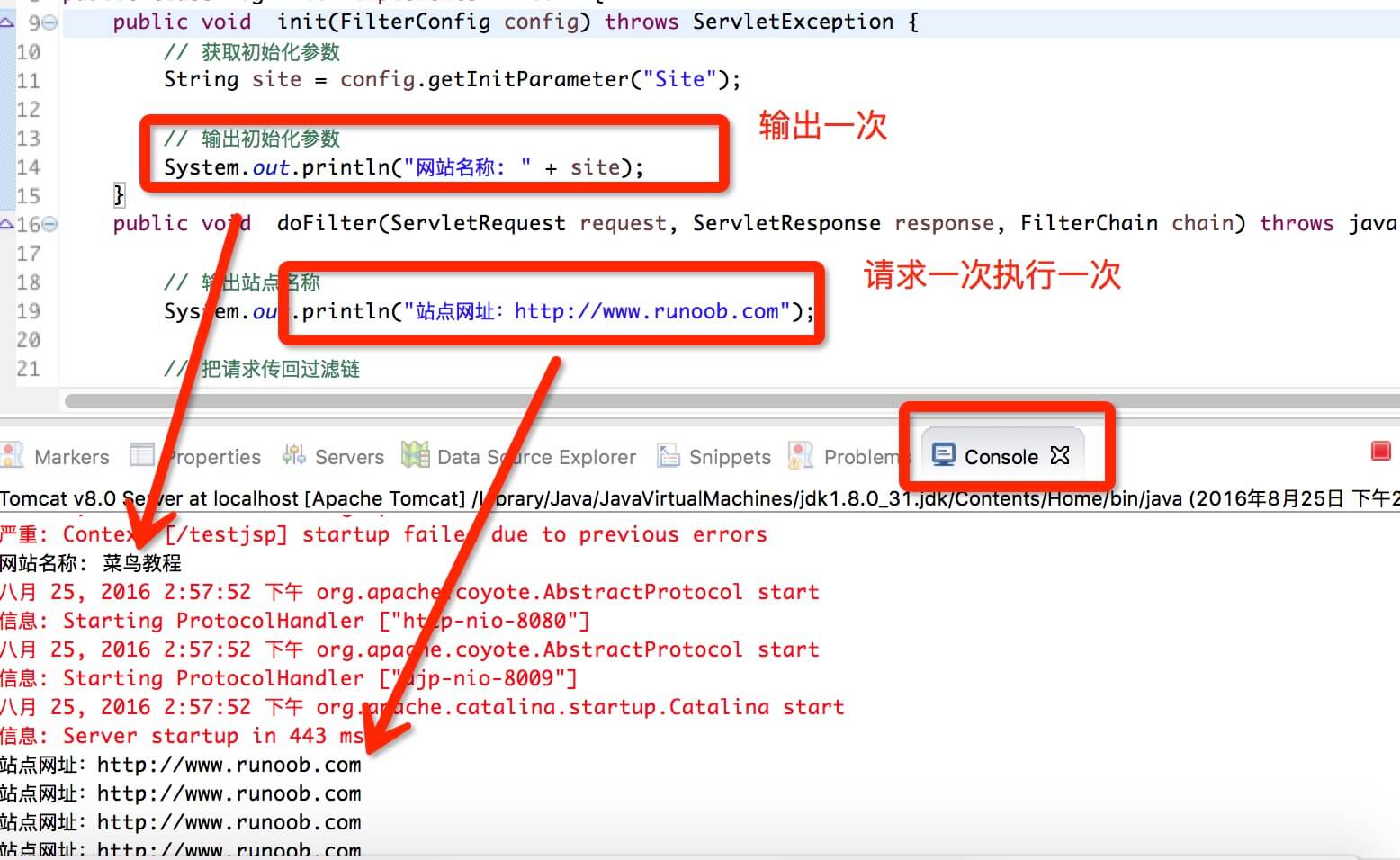
</servlet-mapping>

</web-app>

上述过滤器适用于所有的 Servlet，因为我们在配置中指定 **/\*** 。如果您只想在少数的 Servlet 上应用过滤器，您可以指定一个特定的 Servlet 路径。

现在试着以常用的方式调用任何 Servlet，您将会看到在 Web 服务器中生成的日志。您也可以使用 Log4J 记录器来把上面的日志记录到一个单独的文件中。

接下来我们访问这个实例地址 **http://localhost:8080/TomcatTest/DisplayHeader**, 然后在控制台看下输出内容，如下所示：



## 使用多个过滤器

Web 应用程序可以根据特定的目的定义若干个不同的过滤器。假设您定义了两个过滤器 *AuthenFilter* 和 *LogFilter*。您需要创建一个如下所述的不同的映射，其余的处理与上述所讲解的大致相同：

<filter>

<filter-name>LogFilter</filter-name>

<filter-class>com.runoob.test.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>com.runoob.test.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

web.xml 中的 filter-mapping 元素的顺序决定了 Web 容器应用过滤器到 Servlet 的顺序。若要反转过滤器的顺序，您只需要在 web.xml 文件中反转 filter-mapping 元素即可。

例如，上面的实例将先应用 LogFilter，然后再应用 AuthenFilter，但是下面的实例将颠倒这个顺序：

<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>

## web.xml配置各节点说明

* <filter>指定一个过滤器。
  + <filter-name>用于为过滤器指定一个名字，该元素的内容不能为空。
  + <filter-class>元素用于指定过滤器的完整的限定类名。
  + <init-param>元素用于为过滤器指定初始化参数，它的子元素<param-name>指定参数的名字，<param-value>指定参数的值。

在过滤器中，可以使用FilterConfig接口对象来访问初始化参数。

* <filter-mapping>元素用于设置一个 Filter 所负责拦截的资源。一个Filter拦截的资源可通过两种方式来指定：Servlet 名称和资源访问的请求路径
  + <filter-name>子元素用于设置filter的注册名称。该值必须是在<filter>元素中声明过的过滤器的名字
  + <url-pattern>设置 filter 所拦截的请求路径(过滤器关联的URL样式)
* <servlet-name>指定过滤器所拦截的Servlet名称。
* <dispatcher>指定过滤器所拦截的资源被 Servlet 容器调用的方式，可以是REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST。用户可以设置多个<dispatcher>子元素用来指定 Filter 对资源的多种调用方式进行拦截。
* <dispatcher>子元素可以设置的值及其意义
  + REQUEST：当用户直接访问页面时，Web容器将会调用过滤器。如果目标资源是通过RequestDispatcher的include()或forward()方法访问时，那么该过滤器就不会被调用。
  + INCLUDE：如果目标资源是通过RequestDispatcher的include()方法访问时，那么该过滤器将被调用。除此之外，该过滤器不会被调用。
  + FORWARD：如果目标资源是通过RequestDispatcher的forward()方法访问时，那么该过滤器将被调用，除此之外，该过滤器不会被调用。
  + ERROR：如果目标资源是通过声明式异常处理机制调用时，那么该过滤器将被调用。除此之外，过滤器不会被调用。