1 概述

1.1 Java Servlet 及其特点

Servlet 是 Java 技术对 CGI 编程的回答。Servlet 程序在服务器端运行，动态地生成 Web 页面。与传统的 CGI 和许多其他类似 CGI 的技术相比，Java Servlet 具有更高的效率，更容易使用，功能更强大，具有更好的可移植性，更节省投资（更重要的是， Servlet 程序员收入要比 Perl 程序员高:-）：

高效

在传统的 CGI 中，每个请求都要启动一个新的进程，如果 CGI 程序本身的执行时间较短，启动进程所需要的开销很可能反而超过实际执行时间。而在 Servlet 中，每个请求由一个轻量级的 Java 线程处理（而不是重量级的操作系统进程）。

在传统 CGI 中，如果有 N 个并发的对同一 CGI 程序的请求，则该 CGI 程序的代码在内存中重复装载了 N 次；而对于 Servlet，处理请求的是 N 个线程，只需要一份 Servlet 类代码。在性能优化方面，Servlet 也比 CGI 有着更多的选择，比如缓冲以前的计算结果，保持数据库连接的活动，等等。

方便

Servlet 提供了大量的实用工具例程，例如自动地解析和解码 HTML 表单数据、读取和设置 HTTP 头、处理 Cookie、跟踪会话状态等。

功能强大

在 Servlet 中，许多使用传统 CGI 程序很难完成的任务都可以轻松地完成。例如，Servlet 能够直接和 Web 服务器交互，而普通的 CGI 程序不能。Servlet 还能够在各个程序之间共享数据，使得数据库连接池之类的功能很容易实现。

可移植性好

Servlet 用 Java 编写，Servlet API 具有完善的标准。因此，为 I-Planet Enterprise Server 写的 Servlet 无需任何实质上的改动即可移植到 Apache、Microsoft IIS 或者 WebStar?负跋　械闹髁鞣　衿鞠贾苯踊蛲ü　寮　C諷 ervlet。

节省投资

不仅有许多廉价甚至免费的 Web 服务器可供个人或小规模网站使用，而且对于现有的服务器，如果它不支持 Servlet 的话，要加上这部分功能也往往是免费的（或只需要极少的投资）。

1.2 JSP 及其特点

JavaServer Pages（JSP）是一种实现普通静态 HTML 和动态 HTML 混合编码的技术，有关 JSP 基础概念的说明请参见《JSP 技术简介 》。

许多由 CGI 程序生成的页面大部分仍旧是静态 HTML，动态内容只在页面中有限的几个部分出现。但是包括 Servlet 在内的大多数 CGI 技术及其变种，总是通过程序生成整个页面。JSP

使得我们可以分别创建这两个部分。例如，下面就是一个简单的 JSP 页面：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>欢迎访问网上商店</TITLE></HEAD>
<BODY>
<H1>欢迎</H1>
<SMALL>欢迎,
<!-- 首次访问的用户名字为"New User" -->
<% out.println(Utils.getUserNameFromCookie(request)); %>
要设置帐号信息，请点击
<A HREF="Account-Settings.html">这里</A></SMALL>
<P>
页面的其余内容...
</BODY></HTML>
```

下面是 JSP 和其他类似或相关技术的一个简单比较：

JSP 和 Active Server Pages（ASP）相比

Microsoft 的 ASP 是一种和 JSP 类似的技术。JSP 和 ASP 相比具有两方面的优点。首先，动态部分用 Java 编写，而不是 VB Script 或其他 Microsoft 语言，不仅功能更强大而且更易于使用。第二，JSP 应用可以移植到其他操作系统和非 Microsoft 的 Web 服务器上。

JSP 和纯 Servlet 相比

JSP 并没有增加任何本质上不能用 Servlet 实现的功能。但是，在 JSP 中编写静态 HTML 更加方便，不必再用 println 语句来输出每一行 HTML 代码。更重要的是，借助内容和外观的分离，页面制作中不同性质的任务可以方便地分开：比如，由页面设计专家进行 HTML 设计，同时留出供 Servlet 程序员插入动态内容的空间。

JSP 和服务器端包含（Server-Side Include，SSI）相比

SSI 是一种受到广泛支持的在静态 HTML 中引入外部代码的技术。JSP 在这方面的支持更为完善，因为它可以用 Servlet 而不是独立的程序来生成动态内容。另外，SSI 实际上只用于简单的包含，而不是面向那些能够处理表单数据、访问数据库的“真正的”程序。

JSP 和 JavaScript 相比

JavaScript 能够在客户端动态地生成 HTML。虽然 JavaScript 很有用，但它只能处理以客户端环境为基础的动态信息。除了 Cookie 之外，HTTP 状态和表单提交数据对 JavaScript 来说都是不可用的。另外，由于是在客户端运行，JavaScript 不能访问服务器端资源，比如数据库、目录信息等等。

2 安装

2.1 安装 Servlet 和 JSP 开发工具

要学习 Servlet 和 JSP 开发，首先你必须准备一个符合 Java Servlet 2.1/2.2 和 JavaServer Pages1.0/1.1 规范的开发环境。Sun 提供免费的 JavaServer Web Development Kit（JSWDK），可以从 http://java.sun.com/products/servlet/下载。

安装好 JSWDK 之后，你还要告诉 javac，在编译文件的时候到哪里去寻找 Servlet 和 JSP 类。JSWDK 安装指南对此有详细说明，但主要就是把 servlet.jar 和 jsp.jar 加入 CLASSPATH。CLASSPATH 是一个指示 Java 如何寻找类文件的环境变量，如果不设置 CLASSPATH，Java 在当前目录和标准系统库中寻找类；如果你自己设置了 CLASSPATH，不要忘记包含当前目录（即在 CLASSPATH 中包含“.”）。

另外，为了避免和其他开发者安装到同一 Web 服务器上的 Servlet 产生命名冲突，最好把自己的 Servlet 放入包里面。此时，把包层次结构中的顶级目录也加入 CLASSPATH 会带来不少方便。请参见下文具体说明。

2.2 安装支持 Servlet 的 Web 服务器

除了开发工具之外，你还要安装一个支持 Java Servlet 的 Web 服务器，或者在现有的 Web 服务器上安装 Servlet 软件包。如果你使用的是最新的 Web 服务器或应用服务器，很可能它已经有了所有必需的软件。请查看 Web 服务器的文档，或访问 http://java.sun.com/products/servlet/industry.html 查看支持 Servlet 的服务器软件清单。

虽然最终运行 Servlet 的往往是商业级的服务器，但是开始学习的时候，用一个能够在台式机上运行的免费系统进行开发和测试也足够了。下面是几种当前最受欢迎的产品。

Apache Tomcat.

Tomcat 是 Servlet 2.2 和 JSP 1.1 规范的官方参考实现。Tomcat 既可以单独作为小型 Servlet、JSP 测试服务器，也可以集成到 Apache Web 服务器。直到 2000 年早期，Tomcat 还是唯一的支持 Servlet 2.2 和 JSP 1.1 规范的服务器，但已经有许多其它服务器宣布提供这方面的支持。

Tomcat 和 Apache 一样是免费的。不过，快速、稳定的 Apache 服务器安装和配置起来有点麻烦，Tomcat 也有同样的缺点。和其他商业级 Servlet 引擎相比，配置 Tomcat 的工作量显然要多一点?咛迩氩渭鹅 ttp://jakarta.apache.org/。

JavaServer Web Development Kit (JSWDK).

JSWDK 是 Servlet 2.1 和 JSP 1.0 的官方参考实现。把 Servlet 和 JSP 应用部署到正式运行它们的服务器之前，JSWDK 可以单独作为小型的 Servlet、JSP 测试服务器。JSWDK 也是免费的，而且具有很好的稳定性，但它的安装和配置也较为复杂？咛迩氩渭鹅 ttp://java.sun.com/products/servlet/download.html。

Allaire JRun.

JRun 是一个 Servlet 和 JSP 引擎，它可以集成到 Netscape Enterprise 或 FastTrack Server、IIS、Microsoft Personal Web Server、版本较低的 Apache、O'eilly 的 WebSite 或者 StarNine Web STAR。最多支持 5 个并发连接的限制版本是免费的，商业版本中不存在这个限制，而且增加了远程管理控制台之类的功能?咛迩氩渭鹅 ttp://www.allaire.com/products/jrun/。

New Atlanta 的 ServletExec

ServletExec 是一个快速的 Servlet 和 JSP 引擎，它可以集成到大多数流行的 Web 服务器，支持平台包括 Solaris、Windows、MacOS、HP-UX 和 Linux。ServletExec 可以免费下载和使用，但许多高级功能和管理工具只有在购买了许可之后才可以使用。New Atlanta 还提供一个免费的 Servlet 调试器，该调试器可以在许多流行的 Java IDE 下工作?咛迩氩渭鹕 ttp://newatlanta.com/。

Gefion 的 LiteWebServer (LWS)

LWS 是一个支持 Servlet 2.2 和 JSP 1.1 的免费小型 Web 服务器。 Gefion 还有一个免费的 WAICoolRunner 插件，利用该插件可以为 Netscape FastTrack 和 Enterprise Server 增加 Servlet 2.2 和 JSP 1.1 支持?咛迩氩渭鹕 ttp://www.gefionsoftware.com/。

Sun 的 Java Web Server.

该服务器全部用 Java 写成，而且是首先提供 Servlet 2.1 和 JSP 1.0 规范完整支持的 Web 服务器之一。虽然 Sun 现在已转向 Netscape/I-Planet Server，不再发展 Java Web Server，但它仍旧是一个广受欢迎的 Servlet、JSP 学习平台。要得到免费试用版本，请访问 http://www.sun.com/software/jwebserver/try/。

3 Servlet

3.1 Servlet 基本结构

下面的代码显示了一个简单 Servlet 的基本结构。该 Servlet 处理的是 GET 请求，所谓的 GET 请求，如果你不熟悉 HTTP，可以把它看成是当用户在浏览器地址栏输入 URL、点击 Web 页面中的链接、提交没有指定 METHOD 的表单时浏览器所发出的请求。Servlet 也可以很方便地处理 POST 请求。POST 请求是提交那些指定了 METHOD=“POST”的表单时所发出的请求，具体请参见稍后几节的讨论。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SomeServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {

  // 使用“request”读取和请求有关的信息（比如 Cookies）
  // 和表单数据

  // 使用“response”指定 HTTP 应答状态代码和应答头
  // （比如指定内容类型，设置 Cookie）

  PrintWriter out = response.getWriter();
  // 使用 "out"把应答内容发送到浏览器
  }
}
```

如果某个类要成为 Servlet，则它应该从 HttpServlet 继承，根据数据是通过 GET 还是 POST 发送，覆盖 doGet、doPost 方法之一或全部。doGet 和 doPost 方法都有两个参数，分别为 HttpServletRequest 类型和 HttpServletResponse 类型。HttpServletRequest 提供访问有关请求的信息的方法，例如表单数据、HTTP 请求头等等。HttpServletResponse 除了提供用于指定 HTTP 应答状态（200，404 等）、应答头（Content-Type，Set-Cookie 等）的方法之外，最重要的是它提供了一个用于向客户端发送数据的 PrintWriter 。对于简单的 Servlet 来说，它的大部分工作是通过 println 语句生成向客户端发送的页面。

注意 doGet 和 doPost 抛出两个异常，因此你必须在声明中包含它们。另外，你还必须导入 java.io 包（要用到 PrintWriter 等类）、javax.servlet 包（要用到 HttpServlet 等类）以及 javax.servlet.http 包（要用到 HttpServletRequest 类和 HttpServletResponse 类）。

最后，doGet 和 doPost 这两个方法是由 service 方法调用的，有时你可能需要直接覆盖 service 方法，比如 Servlet 要处理 GET 和 POST 两种请求时。

3.2 输出纯文本的简单 Servlet

下面是一个输出纯文本的简单 Servlet。

3.2.1 HelloWorld.java

```
package hall;

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

public class HelloWorld extends HttpServlet {
      public void doGet(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, IOException {
         PrintWriter out = response.getWriter();
         out.println("Hello World");
   }
}
```

3.2.2 Servlet 的编译和安装

不同的 Web 服务器上安装 Servlet 的具体细节可能不同，请参考 Web 服务器文档了解更权威的说明?俣衣褂肭 ava Web Server（JWS）2.0，则 Servlet 应该安装到 JWS 安装目录的 servlets 子目录下。在本文中，为了避免同一服务器上不同用户的 Servlet 命名冲突，我们把所有 Servlet 都放入一个独立的包 hall 中；如果你和其他人共用一个服务器，而且该服务器没有“虚拟服务器”机制来避免这种命名冲突，那么最好也使用包。把 Servlet 放入了包 hall 之后，HelloWorld.java 实际上是放在 servlets 目录的 hall 子目录下。

大多数其他服务器的配置方法也相似，除了 JWS 之外，本文的 Servlet 和 JSP 示例已经在 BEA WebLogic 和 IBM WebSphere 3.0 下经过测试。WebSphere 具有优秀的虚拟服务器机制，因此，如果只是为了避免命名冲突的话并非一定要用包。

对于没有使用过包的初学者，下面我们介绍编译包里面的类的两种方法。

一种方法是设置 CLASSPATH，使其指向实际存放 Servlet 的目录的上一级目录（Servlet 主目录），然后在该目录中按正常的方式编译。例如，如果 Servlet 的主目录是 C:\JavaWebServer\servlets，包的名字（即主目录下的子目录名字）是 hall，在 Windows 下，编译过程如下：

```
DOS> set CLASSPATH=C:\JavaWebServer\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer\servlets\hall
DOS> javac YourServlet.java
```

第二种编译包里面的 Servlet 的方法是进入 Servlet 主目录，执行“javac directory\YourServlet.java”（Windows）或者“javac directory/YourServlet.java”（Unix）。例如，再次假定 Servlet 主目录是 C:\JavaWebServer\servlets，包的名字是 hall，在 Windows 中编译过程如下：

```
DOS> cd C:\JavaWebServer\servlets
DOS> javac hall\YourServlet.java
```

注意在 Windows 下，大多数 JDK 1.1 版本的 javac 要求目录名字后面加反斜杠(\)。JDK1.2 已经改正这个问题，然而由于许多 Web 服务器仍旧使用 JDK 1.1，因此大量的 Servlet 开发者仍旧在使用 JDK 1.1。

最后，Javac 还有一个高级选项用于支持源代码和.class 文件的分开放置，即你可以用 javac 的“-d”选项把.class 文件安装到 Web 服务器所要求的目录。

3.2.3 运行 Servlet

在 Java Web Server 下，Servlet 应该放到 JWS 安装目录的 servlets 子目录下，而调用 Servlet 的 URL 是 http://host/servlet/ServletName。注意子目录的名字是 servlets（带“s”），而 URL 使用的是“servlet”。由于 HelloWorld Servlet 放入包 hall，因此调用它的 URL 应该是 http://host/servlet/hall.HelloWorld。在其他的服务器上，安装和调用 Servlet 的方法可能略有不同。

大多数 Web 服务器还允许定义 Servlet 的别名，因此 Servlet 也可能用 http://host/any-path/any-file.html 形式的 URL 调用?咛迦绾谓　信渲猛耆　览涤诜　衿骼嘈停　氩慰挤　衿魑牡盗私庇附淞?

3.3 输出 HTML 的 Servlet

大多数 Servlet 都输出 HTML，而不象上例一样输出纯文本。要输出 HTML 还有两个额外的步骤要做：告诉浏览器接下来发送的是 HTML；修改 println 语句构造出合法的 HTML 页面。

第一步通过设置 Content-Type（内容类型）应答头完成。一般地，应答头可以通过 HttpServletResponse 的 setHeader 方法设置，但由于设置内容类型是一个很频繁的操作，因此 Servlet API 提供了一个专用的方法 setContentType。注意设置应答头应该在通过 PrintWriter 发送内容之前进行。下面是一个实例：

HelloWWW.java

```
package hall;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {
  public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">\n" +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>");
    }
  }
```

3.4 几个 HTML 工具函数

通过 println 语句输出 HTML 并不方便，根本的解决方法是使用 JavaServer Pages（JSP）。然而，对于标准的 Servlet 来说，由于 Web 页面中有两个部分（DOCTYPE 和 HEAD）一般不会改变，因此可以用工具函数来封装生成这些内容的代码。

虽然大多数主流浏览器都会忽略 DOCTYPE 行，但严格地说 HTML 规范是要求有 DOCTYPE 行的，它有助于 HTML 语法检查器根据所声明的 HTML 版本检查 HTML 文档合法性。在许多 Web 页面中，HEAD 部分只包含<TITLE>。虽然许多有经验的编写者都会在 HEAD 中包含许多 META 标记和样式声明，但这里只考虑最简单的情况。

下面的 Java 方法只接受页面标题为参数，然后输出页面的 DOCTYPE、HEAD、TITLE 部分。清单如下：

ServletUtilities.java

```
package hall;

public class ServletUtilities {
  public static final String DOCTYPE =
              "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">";

  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" + "<HTML>\n" +
              "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
  }

  // 其他工具函数的代码在本文后面介绍
}
```

HelloWWW2.java

下面是应用了 ServletUtilities 之后重写 HelloWWW 类得到的 HelloWWW2：

```
package hall;

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

public class HelloWWW2 extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println(ServletUtilities.headWithTitle("Hello WWW") +
               "<BODY>\n" +
               "<H1>Hello WWW</H1>\n" +
               "</BODY></HTML>");
   }
}
```

4 表单

4.1 表单数据概述

如果你曾经使用过 Web 搜索引擎，或者浏览过在线书店、股票价格、机票信息，或许会留意到一些古怪的 URL，比如“http://host/path?user=Marty+Hall&origin=bwi&dest=lax”。这个 URL 中位于问号后面的部分，即“user=Marty+Hall&origin=bwi&dest=lax”，就是表单数据，这是将 Web 页面数据发送给服务器程序的最常用方法。对于 GET 请求，表单数据附加到 URL 的问号后面（如上例所示）；对于 POST 请求，表单数据用一个单独的行发送给服务器。

以前，从这种形式的数据提取出所需要的表单变量是 CGI 编程中最麻烦的事情之一。首先，GET 请求和 POST 请求的数据提取方法不同：对于 GET 请求，通常要通过 QUERY_STRING 环境变量提取数据；对于 POST 请求，则一般通过标准输入提取数据。第二，程序员必须负责在“&”符号处截断变量名字-变量值对，再分离出变量名字（等号左边）和变量值（等号右边）。第三，必须对变量值进行 URL 反编码操作。因为发送数据的时候，字母和数字以原来的形式发送，但空格被转换成加号，其他字符被转换成“%XX”形式，其中 XX 是十六进制表示的字符 ASCII（或者 ISO Latin-1）编码值。例如，如果 HTML 表单中名为“users”的域值为“~hall, ~gates, and ~mcnealy”，则实际向服务器发送的数据为“users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy”。最后，即第四个导致解析表单数据非常困难的原因在于，变量值既可能被省略（如“param1=val1＆param2=＆param3=val3”），也有可能一个变量拥有一个以上的值，即同一个变量可能出现一次以上（如“param1=val1＆param2=val2＆param1=val3”）。

Java Servlet 的好处之一就在于所有上述解析操作都能够自动完成。只需要简单地调用一下 HttpServletRequest 的 getParameter 方法、在调用参数中提供表单变量的名字（大小写敏感）即可，而且 GET 请求和 POST 请求的处理方法完全相同。

getParameter 方法的返回值是一个字符串，它是参数中指定的变量名字第一次出现所对应

的值经反编码得到得字符串（可以直接使用）。如果指定的表单变量存在，但没有值，getParameter返回空字符串；如果指定的表单变量不存在，则返回 null。如果表单变量可能对应多个值，可以用 getParameterValues 来取代 getParameter。getParameterValues 能够返回一个字符串数组。

最后，虽然在实际应用中 Servlet 很可能只会用到那些已知名字的表单变量，但在调试环境中，获得完整的表单变量名字列表往往是很有用的，利用 getParamerterNames 方法可以方便地实现这一点。getParamerterNames 返回的是一个 Enumeration，其中的每一项都可以转换为调用 getParameter 的字符串。

4.2 实例：读取三个表单变量

下面是一个简单的例子，它读取三个表单变量 param1、param2 和 param3，并以 HTML 列表的形式列出它们的值。请注意，虽然在发送应答内容之前必须指定应答类型（包括内容类型、状态以及其他 HTTP 头信息），但 Servlet 对何时读取请求内容却没有什么要求。

另外，我们也可以很容易地把 Servlet 做成既能处理 GET 请求，也能够处理 POST 请求，这只需要在 doPost 方法中调用 doGet 方法，或者覆盖 service 方法（service 方法调用 doGet、doPost、doHead 等方法）。在实际编程中这是一种标准的方法，因为它只需要很少的额外工作，却能够增加客户端编码的灵活性。

如果你习惯用传统的 CGI 方法，通过标准输入读取 POST 数据，那么在 Servlet 中也有类似的方法，即在 HttpServletRequest 上调用 getReader 或者 getInputStream，但这种方法对普通的表单变量来说太麻烦。然而，如果是要上载文件，或者 POST 数据是通过专门的客户程序而不是 HTML 表单发送，那么就要用到这种方法。

注意用第二种方法读取 POST 数据时，不能再用 getParameter 来读取这些数据。

ThreeParams.java

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ThreeParams extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "读取三个请求参数";
out.println(ServletUtilities.headWithTitle(title) +
"<BODY>\n" +
"<H1 ALIGN=CENTER>" + title + "</H1>\n" +
"<UL>\n" +
" <LI>param1: "
+ request.getParameter("param1") + "\n" +
" <LI>param2: "
+ request.getParameter("param2") + "\n" +
" <LI>param3: "
```

```
+ request.getParameter("param3") + "\n" +
"</UL>\n" +
"</BODY></HTML>");
}

public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
doGet(request, response);
}
}
```

4.3 实例：输出所有的表单数据

下面这个例子寻找表单所发送的所有变量名字，并把它们放入表格中，没有值或者有多个值的变量都突出显示。

首先，程序通过 HttpServletRequest 的 getParameterNames 方法得到所有的变量名字，getParameterNames 返回的是一个 Enumeration。接下来，程序用循环遍历这个 Enumeration，通过 hasMoreElements 确定何时结束循环，利用 nextElement 得到 Enumeration 中的各个项。由于 nextElement 返回的是一个 Object，程序把它转换成字符串后再用这个字符串来调用 getParameterValues。

getParameterValues 返回一个字符串数组，如果这个数组只有一个元素且等于空字符串，说明这个表单变量没有值，Servlet 以斜体形式输出“No Value”；如果数组元素个数大于 1，说明这个表单变量有多个值，Servlet 以 HTML 列表形式输出这些值；其他情况下 Servlet 直接把变量值放入表格。

ShowParameters.java

注意，ShowParameters.java 用到了前面介绍过的 ServletUtilities.java。

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ShowParameters extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "读取所有请求参数";
out.println(ServletUtilities.headWithTitle(title) +
"<BODY BGCOLOR=\"#FDF5E6\">\n" +
"<H1 ALIGN=CENTER>" + title + "</H1>\n" +
"<TABLE BORDER=1 ALIGN=CENTER>\n" +
"<TR BGCOLOR=\"#FFAD00\">\n" +
"<TH>参数名字<TH>参数值");
```

```
Enumeration paramNames = request.getParameterNames();
while(paramNames.hasMoreElements()) {
String paramName = (String)paramNames.nextElement();
out.println("<TR><TD>" + paramName + "\n<TD>");
String[] paramValues = request.getParameterValues(paramName);
if (paramValues.length == 1) {
String paramValue = paramValues[0];
if (paramValue.length() == 0)
out.print("<I>No Value</I>");
else
out.print(paramValue);
} else {
out.println("<UL>");
for(int i=0; i<paramValues.length; i++) {
out.println("<LI>" + paramValues[i]);
}
out.println("</UL>");
}
}
out.println("</TABLE>\n</BODY></HTML>");
}

public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
doGet(request, response);
}
}
```

测试表单

下面是向上述 Servlet 发送数据的表单 PostForm.html?拖袼　邪　　劳胧淙胗虻谋淼ㄏ谎帽淼ㄐ股 OST 方法发送数据。我们可以看到，在 Servlet 中同时实现 doGet 和 doPost 这两种方法为表单制作带来了方便。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>示例表单</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">用 POST 方法发送数据的表单</H1>

<FORM ACTION="/servlet/hall.ShowParameters"
METHOD="POST">
Item Number:
<INPUT TYPE="TEXT" NAME="itemNum"><BR>
Quantity:
<INPUT TYPE="TEXT" NAME="quantity"><BR>
Price Each:
<INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
<HR>
```

```
First Name:
<INPUT TYPE="TEXT" NAME="firstName"><BR>
Last Name:
<INPUT TYPE="TEXT" NAME="lastName"><BR>
Middle Initial:
<INPUT TYPE="TEXT" NAME="initial"><BR>
Shipping Address:
<TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
Credit Card:<BR>
<INPUT TYPE="RADIO" NAME="cardType"
VALUE="Visa">Visa<BR>
<INPUT TYPE="RADIO" NAME="cardType"
VALUE="Master Card">Master Card<BR>
<INPUT TYPE="RADIO" NAME="cardType"
VALUE="Amex">American Express<BR>
<INPUT TYPE="RADIO" NAME="cardType"
VALUE="Discover">Discover<BR>
<INPUT TYPE="RADIO" NAME="cardType"
VALUE="Java SmartCard">Java SmartCard<BR>
Credit Card Number:
<INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
Repeat Credit Card Number:
<INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
<CENTER>
<INPUT TYPE="SUBMIT" VALUE="Submit Order">
</CENTER>
</FORM>

</BODY>
</HTML>
```

5 HTTP 请求头

5.1 HTTP 请求头概述

HTTP 客户程序（例如浏览器），向服务器发送请求的时候必须指明请求类型（一般是 GET 或者 POST）。如有必要，客户程序还可以选择发送其他的请求头。大多数请求头并不是必需的，但 Content-Length 除外。对于 POST 请求来说 Content-Length 必须出现。

下面是一些最常见的请求头

Accept：浏览器可接受的 MIME 类型。

Accept-Charset：浏览器可接受的字符集。

Accept-Encoding：浏览器能够进行解码的数据编码方式，比如 gzip。Servlet 能够向支持 gzip 的浏览器返回经 gzip 编码的 HTML 页面。许多情形下这可以减少 5 到 10 倍的下载时间 。

Accept-Language：浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到。

Authorization：授权信息，通常出现在对服务器发送的 WWW-Authenticate 头的应答中。

Connection：表示是否需要持久连接。如果 Servlet 看到这里的值为“Keep-Alive”，或者看到请求使用的是 HTTP 1.1（HTTP 1.1 默认进行持久连接），它就可以利用持久连接的优点，当页面包含多个元素时（例如 Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet 需要在应答中发送一个 Content-Length 头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。

Content-Length：表示请求消息正文的长度。

Cookie：这是最重要的请求头信息之一，参见后面《Cookie 处理》一章中的讨论。

From：请求发送者的 email 地址，由一些特殊的 Web 客户程序使用，浏览器不会用到它。

Host：初始 URL 中的主机和端口。

If-Modified-Since：只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回 304“Not Modified”应答。

Pragma：指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝。

Referer：包含一个 URL，用户从该 URL 代表的页面出发访问当前请求的页面。

User-Agent：浏览器类型，如果 Servlet 返回的内容与浏览器类型有关则该值非常有用。

UA-Pixels，UA-Color，UA-OS，UA-CPU：由某些版本的 IE 浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和 CPU 类型。

有关 HTTP 头完整、详细的说明，请参见 http://www.w3.org/Protocols/的 HTTP 规范。

5.2 在 Servlet 中读取请求头

在 Servlet 中读取 HTTP 头是非常方便的，只需要调用一下 HttpServletRequest 的 getHeader 方法即可。如果客户请求中提供了指定的头信息，getHeader 返回对应的字符串；否则，返回 null。部分头信息经常要用到，它们有专用的访问方法：getCookies 方法返回 Cookie 头的内容，经解析后存放在 Cookie 对象的数组中，请参见后面有关 Cookie 章节的讨论；getAuthType 和 getRemoteUser 方法分别读取 Authorization 头中的一部分内容；getDateHeader 和 getIntHeader 方法读取指定的头，然后返回日期值或整数值。

除了读取指定的头之外，利用 getHeaderNames 还可以得到请求中所有头名字的一个 Enumeration 对象。

最后，除了查看请求头信息之外，我们还可以从请求主命令行获得一些信息。getMethod 方法返回请求方法，请求方法通常是 GET 或者 POST，但也有可能是 HEAD、PUT 或者 DELETE。getRequestURI 方法返回 URI（URI 是 URL 的从主机和端口之后到表单数据之前的那一部分）。getRequestProtocol 返回请求命令的第三部分，一般是“HTTP/1.0”或者“HTTP/1.1”。

5.3 实例：输出所有的请求头

下面的 Servlet 实例把所有接收到的请求头和它的值以表格的形式输出。另外，该 Servlet 还会输出主请求命令的三个部分：请求方法，URI，协议/版本。

ShowRequestHeaders.java

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "显示所有请求头";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<B>Request Method: </B>" +
    request.getMethod() + "<BR>\n" +"<B>Request URI: </B>" +
    request.getRequestURI() + "<BR>\n" +"<B>Request Protocol: </B>" +
    request.getProtocol() + "<BR><BR>\n" +"<TABLE BORDER=1 ALIGN=CENTER>\n" +
                            "<TR BGCOLOR=\"#FFAD00\">\n" +
                            "<TH>Header Name<TH>Header Value");
    Enumeration headerNames = request.getHeaderNames();
    while(headerNames.hasMoreElements()) {
      String headerName = (String)headerNames.nextElement();
      out.println("<TR><TD>" + headerName);
      out.println(" <TD>" + request.getHeader(headerName));
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }

  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
      doGet(request, response);
    }
  }
```

关于 CLASSPATH

2002-02-17 22:17

钝刀
管理员

Online
注册日期: Dec 2001
来自: 上海
发帖数量 1093

|

Java Servlet 和 JSP 教程(6-8)
6 CGI 变量

6.1 CGI 变量概述

如果你是从传统的 CGI 编程转而学习 Java Servlet，或许已经习惯了“CGI 变量”这一概念。CGI 变量汇集了各种有关请求的信息：

部分来自 HTTP 请求命令和请求头，例如 Content-Length 头；

部分来自 Socket 本身，例如主机的名字和 IP 地址；

也有部分与服务器安装配置有关，例如 URL 到实际路径的映射。

6.2 标准 CGI 变量的 Servlet 等价表示

下表假定 request 对象是提供给 doGet 和 doPost 方法的 HttpServletRequest 类型对象。

CGI 变量 含义 从 doGet 或 doPost 访问
AUTH_TYPE 如果提供了 Authorization 头，这里指定了具体的模式（basic 或者 digest）。 request.getAuthType()
CONTENT_LENGTH 只用于 POST 请求，表示所发送数据的字节数 严格地讲，等价的表达方式应该是 String.valueOf(request.getContentLength())（返回一个字符串）。但更？ 氖怯胃equest.getContentLength()返回含义相同的整数
CONTENT_TYPE 如果指定的话，表示后面所跟数据的类型。 request.getContentType()
DOCUMENT_ROOT 与 http://host/对应的路径。 getServletContext().getRealPath("/")
注意低版本 Servlet 规范中的等价表达方式是 request.getRealPath("/")。
HTTP_XXX_YYY 访问任意 HTTP 头。 request.getHeader("Xxx-Yyy")
PATH_INFO URL 中的附加路径信息，即 URL 中 Servlet 路径之后、查询字符串之前的那部分。 request.getPathInfo()
PATH_TRANSLATED 映射到服务器实际路径之后的路径信息。 request.getPathTranslated()
QUERY_STRING 这是字符串形式的附加到 URL 后面的查询字符串，数据仍旧是 URL 编码的。在 Servlet 中很少需要用到未经解码的数据，一般使用 getParameter 访问各个参数。

request.getQueryString()
REMOTE_ADDR 发出请求的客户机的 IP 地址。 request.getRemoteAddr()
REMOTE_HOST 发出请求的客户机的完整的域名，如 java.sun.com。如果不能确定该域名，则返回 IP 地址。 request.getRemoteHost()
REMOTE_USER 如果提供了 Authorization 头，则代表其用户部分。它代表发出请求的用户的名字。 request.getRemoteUser()
REQUEST_METHOD 请求类型。通常是 GET 或者 POST。但偶尔也会出现 HEAD，PUT，DELETE，OPTIONS，或者 TRACE. request.getMethod()
SCRIPT_NAME URL 中调用 Servlet 的那一部分，不包含附加路径信息和查询字符串。request.getServletPath()
SERVER_NAME Web 服务器名字。 request.getServerName()
SERVER_PORT String.valueOf 服务器监听的端口。 严格地说，等价表达应该是返回字符串的(request.getServerPort())。 但经常使用返回整数值的 request.getServerPort()。
SERVER_PROTOCOL 请求命令中的协议名字和版本（即 HTTP/1.0 或 HTTP/1.1）request.getProtocol()
SERVER_SOFTWARE Servlet 引擎的名字和版本。 getServletContext().getServerInfo()

6.3 实例：读取 CGI 变量

下面这个 Servlet 创建一个表格，显示除了 HTTP_XXX_YYY 之外的所有 CGI 变量。HTTP_XXX_YYY 是 HTTP 请求头信息，请参见上一节介绍。

ShowCGIVariables.java

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ShowCGIVariables extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   String[][] variables =
      { { "AUTH_TYPE", request.getAuthType() },
      { "CONTENT_LENGTH", String.valueOf(request.getContentLength()) },
      { "CONTENT_TYPE", request.getContentType() },
      { "DOCUMENT_ROOT", getServletContext().getRealPath("/") },
      { "PATH_INFO", request.getPathInfo() },
      { "PATH_TRANSLATED", request.getPathTranslated() },
      { "QUERY_STRING", request.getQueryString() },
      { "REMOTE_ADDR", request.getRemoteAddr() },
      { "REMOTE_HOST", request.getRemoteHost() },
      { "REMOTE_USER", request.getRemoteUser() },
      { "REQUEST_METHOD", request.getMethod() },
      { "SCRIPT_NAME", request.getServletPath() },
      { "SERVER_NAME", request.getServerName() },
      { "SERVER_PORT", String.valueOf(request.getServerPort()) },
```

```
        { "SERVER_PROTOCOL", request.getProtocol() },
        { "SERVER_SOFTWARE", getServletContext().getServerInfo() }
};
    String title = "显示 CGI 变量";
    out.println(ServletUtilities.headWithTitle(title) +"<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                    "<TR BGCOLOR=\"#FFAD00\">\n" +
                    "<TH>CGI Variable Name<TH>Value");
    for(int i=0; i<variables.length; i++) {
          String varName = variables[i][0];
          String varValue = variables[i][1];
          if (varValue == null)
             varValue = "<I>Not specified</I>";
             out.println("<TR><TD>" + varName + "<TD>" + varValue);
    }
    out.println("</TABLE></BODY></HTML>");
}

public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
doGet(request, response);
}
}
```

7 状态代码

7.1 状态代码概述

Web 服务器响应浏览器或其他客户程序的请求时，其应答一般由以下几个部分组成：一个状态行，几个应答头，一个空行，内容文档。下面是一个最简单的应答：

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello World
```

状态行包含 HTTP 版本、状态代码、与状态代码对应的简短说明信息。在大多数情况下，除了 Content-Type 之外的所有应答头都是可选的。但 Content-Type 是必需的，它描述的是后面文档的 MIME 类型。虽然大多数应答都包含一个文档，但也有一些不包含，例如对 HEAD 请求的应答永远不会附带文档。有许多状态代码实际上用来标识一次失败的请求，这些应答也不包含文档（或只包含一个简短的错误信息说明）。

Servlet 可以利用状态代码来实现许多功能。例如，可以把用户重定向到另一个网站；可以指示出后面的文档是图片、PDF 文件或 HTML 文件；可以告诉用户必须提供密码才能访问文档；等等。这一部分我们将具体讨论各种状态代码的含义以及利用这些代码可以做些什么。

7.2 设置状态代码

如前所述，HTTP 应答状态行包含 HTTP 版本、状态代码和对应的状态信息。由于状态信息直接和状态代码相关，而 HTTP 版本又由服务器确定，因此需要 Servlet 设置的只有一个状态代码。

Servlet 设置状态代码一般使用 HttpServletResponse 的 setStatus 方法。setStatus 方法的参数是一个整数（即状态代码），不过为了使得代码具有更好的可读性，可以用 HttpServletResponse 中定义的常量来避免直接使用整数。这些常量根据 HTTP 1.1 中的标准状态信息命名，所有的名字都加上了 SC 前缀（Status Code 的缩写）并大写，同时把空格转换成了下划线。也就是说，与状态代码 404 对应的状态信息是“Not Found”，则 HttpServletResponse 中的对应常量名字为 SC_NOT_FOUND。但有两个例外：和状态代码 302 对应的常量根据 HTTP 1.0 命名，而 307 没有对应的常量。

设置状态代码并非总是意味着不要再返回文档。例如，虽然大多数服务器返回 404 应答时会输出简单的“File Not Found”信息，但 Servlet 也可以定制这个应答。不过，定制应答时应当在通过 PrintWriter 发送任何内容之前先调用 response.setStatus。

虽然设置状态代码一般使用的是 response.setStauts(int) 方法，但为了简单起见，HttpServletResponse 为两种? 那樾翁峁 ├俗业梅椒ǎ篡 endError 方法生成一个 404 应答，同时生成一个简短的 HTML 错误信息文档；sendRedirect 方法生成一个 302 应答，同时在 Location 头中指示新文档的 URL。

7.3 HTTP 1.1 状态代码及其含义

下表显示了? 脸 TTP 1.1 状态代码以及它们对应的状态信息和含义。

应当谨慎地使用那些只有 HTTP 1.1 支持的状态代码，因为许多浏览器还只能够支持 HTTP 1.0。如果你使用了 HTTP 1.1 特有的状态代码，最好能够检查一下请求的 HTTP 版本号（通过 HttpServletRequest 的 getProtocol 方法）。

状态代码 状态信息 含义
100 Continue 初始的请求已经接受，客户应当继续发送请求的其余部分。（HTTP 1.1 新）
101 Switching Protocols 服务器将遵从客户的请求转换到另外一种协议（HTTP 1.1 新）
200 OK 一切正常，对 GET 和 POST 请求的应答文档跟在后面。如果不用 setStatus 设置状态代码，Servlet 默认使用 202 状态代码。
201 Created 服务器已经创建了文档，Location 头给出了它的 URL。
202 Accepted 已经接受请求，但处理尚未完成。
203 Non-Authoritative Information 文档已经正常地返回，但一些应答头可能不正确，因为使用的是文档的拷贝（HTTP 1.1 新）。
204 No Content 没有新文档，浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而 Servlet 可以确定用户文档足够新，这个状态代码是很有用的。
205 Reset Content 没有新的内容，但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容（HTTP 1.1 新）。
206 Partial Content 客户发送了一个带有 Range 头的 GET 请求，服务器完成了它（HTTP 1.1 新）。
300 Multiple Choices 客户请求的文档可以在多个位置找到，这些位置已经在返回的文档内列出。如果服务器要提出优先选择，则应该在 Location 应答头指明。
301 Moved Permanently 客户请求的文档在其他地方，新的 URL 在 Location 头中给出，浏览器应该自动地访问新的 URL。
302 Found 类似于 301，但新的 URL 应该被视为临时性的替代，而不是永久性的。注意，在

HTTP1.0 中对应的状态信息是“Moved Temporatily”，而 HttpServletResponse 中相应的常量是 SC_MOVED_TEMPORARILY，而不是 SC_FOUND。
出现该状态代码时，浏览器能够自动访问新的 URL，因此它是一个很有用的状态代码。为此，Servlet 提供了一个专用的方法，即 sendRedirect。使用 response.sendRedirect(url) 比使用 response.setStatus(response.SC_MOVED_TEMPORARILY)和 response.setHeader("Location",url)更好。这是因为：

首先，代码更加简洁。
第二，使用 sendRedirect，Servlet 会自动构造一个包含新链接的页面（用于那些不能自动重定向的老式浏览器）。
最后，sendRedirect 能够处理相对 URL，自动把它们转换成绝对 URL。
注意这个状态代码有时候可以和 301 替换使用。例如，如果浏览器错误地请求 http://host/~user（缺少了后面的斜杠），有的服务器返回 301，有的则返回 302。

严格地说，我们只能假定只有当原来的请求是 GET 时浏览器才会自动重定向。请参见 307。
303 See Other 类似于 301/302，不同之处在于，如果原来的请求是 POST，Location 头指定的重定向目标文档应该通过 GET 提取（HTTP 1.1 新）。
304 Not Modified 客户端有缓冲的文档并发出了一个条件性的请求（一般是提供 If-Modified-Since 头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。
305 Use Proxy 客户请求的文档应该通过 Location 头所指明的代理服务器提取（HTTP 1.1 新）。
307 Temporary Redirect 和 302（Found）相同 。许多浏览器会错误地响应 302 应答进行重定向，即使原来的请求是 POST，即使它实际上只能在 POST 请求的应答是 303 时才能重定向。由于这个原因，HTTP 1.1 新增了 307，以便更加清除地区分几个状态代码：当出现 303 应答时，浏览器可以跟随重定向的 GET 和 POST 请求；如果是 307 应答，则浏览器只能跟随对 GET 请求的重定向。 注意，HttpServletResponse 中没有为该状态代码提供相应的常量。（HTTP 1.1 新）
400 Bad Request 请求出现语法错误。
401 Unauthorized 客户试图未经授权访问受密码保护的页面。应答中会包含一个 WWW-Authenticate 头，浏览器据此显示用户名字/密码对话框，然后在填写合适的 Authorization 头后再次发出请求。
403 Forbidden 资源不可用。服务器理解客户的请求，但拒绝处理它。通常由于服务器上文件或目录的权限设置导致。
404 Not Found 无法找到指定位置的资源。这也是一个常用的应答，HttpServletResponse 专门提供了相应的方法：sendError(message)。
405 Method Not Allowed 请求方法（GET、POST、HEAD、DELETE、PUT、TRACE 等）对指定的资源不适用。（HTTP 1.1 新）
406 Not Acceptable 指定的资源已经找到，但它的 MIME 类型和客户在 Accpet 头中所指定的不兼容（HTTP 1.1 新）。
407 Proxy Authentication Required 类似于 401，表示客户必须先经过代理服务器的授权。（HTTP 1.1 新）
408 Request Timeout 在服务器许可的等待时间内，客户一直没有发出任何请求。客户可以在以后重复同一请求。（HTTP 1.1 新）
409 Conflict 通常和 PUT 请求有关。由于请求和资源的当前状态相冲突，因此请求不能成功。（HTTP 1.1 新）
410 Gone 所请求的文档已经不再可用，而且服务器不知道应该重定向到哪一个地址。它和 404 的不同在于，返回 407 表示文档永久地离开了指定的位置，而 404 表示由于未知的原因文档不可用。（HTTP 1.1 新）

411 Length Required 服务器不能处理请求，除非客户发送一个 Content-Length 头。（HTTP 1.1 新）
412 Precondition Failed 请求头中指定的一些前提条件失败（HTTP 1.1 新）。
413 Request Entity Too Large 目标文档的大小超过服务器当前愿意处理的大小。如果服务器认为自己能够稍后再处理该请求，则应该提供一个 Retry-After 头（HTTP 1.1 新）。
414 Request URI Too Long URI 太长（HTTP 1.1 新）。
416 Requested Range Not Satisfiable 服务器不能满足客户在请求中指定的 Range 头。（HTTP 1.1 新）
500 Internal Server Error 服务器遇到了意料不到的情况，不能完成客户的请求。
501 Not Implemented 服务器不支持实现请求所需要的功能。例如，客户发出了一个服务器不支持的 PUT 请求。
502 Bad Gateway 服务器作为网关或者代理时，为了完成请求访问下一个服务器，但该服务器返回了非法的应答。
503 Service Unavailable 服务器由于维护或者负载过重未能应答。例如，Servlet 可能在数据库连接池已满的情况下返回 503。服务器返回 503 时可以提供一个 Retry-After 头。
504 Gateway Timeout 由作为代理或网关的服务器使用，表示不能及时地从远程服务器获得应答。（HTTP 1.1 新）
505 HTTP Version Not Supported 服务器不支持请求中所指明的 HTTP 版本。（HTTP 1.1 新）

7.4 实例：访问多个搜索引擎

下面这个例子用到了除 200 之外的另外两个? 刺 耄?02 和 404。302 通过 sendRedirect 方法设置，404 通过 sendError 方法设置。

在这个例子中，首先出现的 HTML 表单用来选择搜索引擎、搜索字符串、每页显示的搜索结果数量。表单提交后，Servlet 提取这三个变量，按照所选择的搜索引擎的要求构造出包含这些变量的 URL，然后把用户重定向到这个 URL。如果用户不能正确地选择搜索引擎，或者利用其他表单发送了一个不认识的搜索引擎名字，则返回一个提示搜索引擎找不到的 404 页面。

SearchEngines.java

注意：这个 Servlet 要用到后面给出的 SearchSpec 类，SearchSpec 的功能是构造适合不同搜索引擎的 URL。

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public class SearchEngines extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
// getParameter 自动解码 URL 编码的查询字符串。由于我们
//  要把查询字符串发送给另一个服务器，因此再次使用
// URLEncoder 进行 URL 编码
String searchString =URLEncoder.encode(request.getParameter("searchString"));
String numResults =request.getParameter("numResults");
String searchEngine =request.getParameter("searchEngine");
```

```
SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
for(int i=0; i<commonSpecs.length; i++) {
   SearchSpec searchSpec = commonSpecs[i];
   if (searchSpec.getName().equals(searchEngine)) {
      String url =response.encodeURL(searchSpec.makeURL(searchString,numResults));
      response.sendRedirect(url);
   return;
   }
   }
   response.sendError(response.SC_NOT_FOUND,"No recognized search engine specified.");
}

public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
   doGet(request, response);
}
}
```

SearchSpec.java

```
package hall;

class SearchSpec {
private String name, baseURL, numResultsSuffix;

private static SearchSpec[] commonSpecs =
      { new SearchSpec("google",
         "http://www.google.com/search?q=",,
         "&num="),
      new SearchSpec("infoseek","http://infoseek.go.com/Titles?qt=",,"&nh="),
      new SearchSpec("lycos","http://lycospro.lycos.com/cgi-bin/pursuit?query=",,
                              "&maxhits="),
      new SearchSpec("hotbot","http://www.hotbot.com/?MT=",,"&DC=")
   };

public SearchSpec(String name,String baseURL,String numResultsSuffix) {
   this.name = name;
   this.baseURL = baseURL;
   this.numResultsSuffix = numResultsSuffix;
}

public String makeURL(String searchString, String numResults) {
   return(baseURL + searchString + numResultsSuffix + numResults);
}

public String getName() {
   return(name);
}

public static SearchSpec[] getCommonSpecs() {
   return(commonSpecs);
}
```

```
}
```

SearchEngines.html

下面是调用上述 Servlet 的 HTML 表单。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>访问多个搜索引擎</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">

<FORM ACTION="/servlet/hall.SearchEngines">
<CENTER>
搜索关键字:
<INPUT TYPE="TEXT" NAME="searchString"><BR>
每页显示几个查询结果:
<INPUT TYPE="TEXT" NAME="numResults"
VALUE=10 SIZE=3><BR>
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="google">
Google |
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="infoseek">
Infoseek |
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="lycos">
Lycos |
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="hotbot">
HotBot
<BR>
<INPUT TYPE="SUBMIT" VALUE="Search">
</CENTER>
</FORM>

</BODY>
</HTML>
```

8 HTTP 应答头

8.1 HTTP 应答头概述

Web 服务器的 HTTP 应答一般由以下几项构成：一个状态行，一个或多个应答头，一个空行，内容文档。设置 HTTP 应答头往往和设置状态行中的状态代码结合起来。例如，有好几个表示“文档位置已经改变”的状态代码都伴随着一个 Location 头，而 401（Unauthorized）状态代码则必须伴随一个 WWW-Authenticate 头。

然而，即使在没有设置特殊含义的状态代码时，指定应答头也是很有用的。应答头可以用

来完成：设置 Cookie，指定修改日期，指示浏览器按照指定的间隔刷新页面，声明文档的长度以便利用持久 HTTP 连接，……等等许多其他任务。

设置应答头最常用的方法是 HttpServletResponse 的 setHeader，该方法有两个参数，分别表示应答头的名字和值。和设置状态代码相似，设置应答头应该在发送任何文档内容之前进行。

setDateHeader 方法和 setIntHeadr 方法专门用来设置包含日期和整数值的应答头，前者避免了把 Java 时间转换为 GMT 时间字符串的麻烦，后者则避免了把整数转换为字符串的麻烦。

HttpServletResponse 还提供了许多设置? Y鹜返募虮惴椒ǎ 缦滤 荆?

setContentType：设置 Content-Type 头。大多数 Servlet 都要用到这个方法。
setContentLength：设置 Content-Length 头。对于支持持久 HTTP 连接的浏览器来说，这个函数是很有用的。
addCookie：设置一个 Cookie（Servlet API 中没有 setCookie 方法，因为应答往往包含多个 Set-Cookie 头）。
另外，如上节介绍，sendRedirect 方法设置状态代码 302 时也会设置 Location 头。

有关 HTTP 头详细和完整的说明，请参见 http://www.w3.org/Protocols/规范。

应答头 说明
Allow 服务器支持哪些请求方法（如 GET、POST 等）。
Content-Encoding 文档的编码（Encode）方法。只有在解码之后才可以得到 Content-Type 头指定的内容类型。利用 gzip 压缩文档能够显著地减少 HTML 文档的下载时间。Java 的 GZIPOutputStream 可以很方便地进行 gzip 压缩，但只有 Unix 上的 Netscape 和 Windows 上的 IE 4、IE 5 才支持它。因此，Servlet 应该通过查看 Accept-Encoding 头（即 request.getHeader("Accept-Encoding")）检查浏览器是否支持 gzip，为支持 gzip 的浏览器返回经 gzip 压缩的 HTML 页面，为其他浏览器返回普通页面。
Content-Length 表示内容长度。只有当浏览器使用持久 HTTP 连接时才需要这个数据。如果你想要利用持久连接的优势，可以把输出文档写入 ByteArrayOutputStram，完成后查看其大小，然后把该值放入 Content-Length 头，最后通过 byteArrayStream.writeTo(response.getOutputStream() 发送内容。
Content-Type 表示后面的文档属于什么 MIME 类型。Servlet 默认为 text/plain，但通常需要显式地指定为 text/html。由于经常要设置 Content-Type，因此 HttpServletResponse 提供了一个专用的方法 setContentTyep。
Date 当前的 GMT 时间。你可以用 setDateHeader 来设置这个头以避免转换时间格式的麻烦。
Expires 应该在什么时候认为文档已经过期，从而不再缓存它？
Last-Modified 文档的最后改动时间。客户可以通过 If-Modified-Since 请求头提供一个日期，该请求将被视为一个条件 GET，只有改动时间迟于指定时间的文档才会返回，否则返回一个 304（Not Modified）状态。Last-Modified 也可用 setDateHeader 方法来设置。
Location 表示客户应当到哪里去提取文档。Location 通常不是直接设置的，而是通过 HttpServletResponse 的 sendRedirect 方法，该方法同时设置状态代码为 302。
Refresh 表示浏览器应该在多少时间之后刷新文档，以秒计。除了刷新当前文档之外，你还可以通过 setHeader("Refresh", "5; URL=http://host/path")让浏览器读取指定的页面。注意这种功能通常是通过设置 HTML 页面 HEAD 区的 <META HTTP-EQUIV="Refresh" CONTENT="5;URL=http://host/path">实现，这是因为，自动刷新或重定向对于那些不能使用 CGI 或 Servlet 的 HTML 编写者十分重要。但是，对于 Servlet 来说，直接设置 Refresh 头更加方便。注意 Refresh 的意义是“N 秒之后刷新本页面或访问指定页面”，而不是“每隔 N 秒刷新本页面或

访问指定页面”。因此，连续刷新要求每次都发送一个 Refresh 头，而发送 204 状态代码则可以阻止浏览器继续刷新，不管是使用 Refresh 头还是<META HTTP-EQUIV="Refresh" ...>。注意 Refresh 头不属于 HTTP 1.1 正式规范的一部分，而是一个扩展，但 Netscape 和 IE 都支持它。
Server 服务器名字。Servlet 一般不设置这个值，而是由 Web 服务器自己设置。
Set-Cookie 设置和页面关联的 Cookie。Servlet 不应使用 response.setHeader("Set-Cookie", ...)，而是应使用 HttpServletResponse 提供的专用方法 addCookie。参见下文有关 Cookie 设置的讨论。
WWW-Authenticate 客户应该在 Authorization 头中提供什么类型的授权信息？在包含 401（Unauthorized）状态行的应答中这个头是必需的。例如，response.setHeader("WWW-Authenticate", "BASIC realm=\"executives\"")。注意 Servlet 一般不进行这方面的处理，而是让 Web 服务器的专门机制来控制受密码保护页面的访问（例如.htaccess）。

8.3 实例：内容改变时自动刷新页面

下面这个 Servlet 用来计算大素数。因为计算非常大的数字（例如 500 位）可能要花不少时间，所以 Servlet 将立即返回已经找到的结果，同时在后台继续计算。后台计算使用一个优先级较低的线程以避免过多地影响 Web 服务器的性能。如果计算还没有完成，Servlet 通过发送 Refresh 头指示浏览器在几秒之后继续请求新的内容。

注意，本例除了说明 HTTP 应答头的用处之外，还显示了 Servlet 的另外两个很有价值的功能。首先，它表明 Servlet 能够处理多个并发的连接，每个都有自己的线程。Servlet 维护了一份已有素数计算请求的 Vector 表，通过查找素数个数（素数列表的长度）和数字个数（每个素数的长度）将当前请求和已有请求相匹配，把所有这些请求同步到这个列表上。第二，本例证明，在 Servlet 中维持请求之间的状态信息是非常容易的。维持状态信息在传统的 CGI 编程中是一件很麻烦的事情。由于维持了状态信息，浏览器能够在刷新页面时访问到正在进行的计算过程，同时也使得 Servlet 能够保存一个有关最近请求结果的列表，当一个新的请求指定了和最近请求相同的参数时可以立即返回结果。

PrimeNumbers.java

注意，该 Servlet 要用到前面给出的 ServletUtilities.java。另外还要用到：PrimeList.java，用于在后台线程中创建一个素数的 Vector；Primes.java，用于随机生成 BigInteger 类型的大数字，检查它们是否是素数。（此处略去 PrimeList.java 和 Primes.java 的代码。）

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class PrimeNumbers extends HttpServlet {
private static Vector primeListVector = new Vector();
private static int maxPrimeLists = 30;

public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
int numPrimes = ServletUtilities.getIntParameter(request, "numPrimes", 50);
int numDigits = ServletUtilities.getIntParameter(request, "numDigits", 120);
PrimeList primeList = findPrimeList(primeListVector, numPrimes, numDigits);
if (primeList == null) {
```

```
primeList = new PrimeList(numPrimes, numDigits, true);
synchronized(primeListVector) {
if (primeListVector.size() >= maxPrimeLists)
primeListVector.removeElementAt(0);
primeListVector.addElement(primeList);
}
}
Vector currentPrimes = primeList.getPrimes();
int numCurrentPrimes = currentPrimes.size();
int numPrimesRemaining = (numPrimes - numCurrentPrimes);
boolean isLastResult = (numPrimesRemaining == 0);
if (!isLastResult) {
response.setHeader("Refresh", "5");
}
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "Some " + numDigits + "-Digit Prime Numbers";
out.println(ServletUtilities.headWithTitle(title) +
"<BODY BGCOLOR=\"#FDF5E6\">\n" +
"<H2 ALIGN=CENTER>" + title + "</H2>\n" +
"<H3>Primes found with " + numDigits +
" or more digits: " + numCurrentPrimes + ".</H3>");
if (isLastResult)
out.println("<B>Done searching.</B>");
else
out.println("<B>Still looking for " + numPrimesRemaining +
" more<BLINK>...</BLINK></B>");
out.println("<OL>");
for(int i=0; i<numCurrentPrimes; i++) {
out.println(" <LI>" + currentPrimes.elementAt(i));
}
out.println("</OL>");
out.println("</BODY></HTML>");
}

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

// 检查是否存在同类型请求（已经完成，或者正在计算）。
// 如存在，则返回现有结果而不是启动新的后台线程。
private PrimeList findPrimeList(Vector primeListVector,
int numPrimes,
int numDigits) {
synchronized(primeListVector) {
for(int i=0; i<primeListVector.size(); i++) {
PrimeList primes = (PrimeList)primeListVector.elementAt(i);
if ((numPrimes == primes.numPrimes()) &&
(numDigits == primes.numDigits()))
return(primes);
}
```

```
return(null);
}
}
}
```

PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>大素数计算</TITLE>
</HEAD>
<CENTER>
<BODY BGCOLOR="#FDF5E6">
<FORM ACTION="/servlet/hall.PrimeNumbers">
<B>要计算几个素数:</B>
<INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
<B>每个素数的位数:</B>
<INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
<INPUT TYPE="SUBMIT" VALUE="开始计算">
</FORM>
</CENTER>
</BODY>
</HTML>
```

_____
凡事太急,缘份势必早尽

Java Servlet 和 JSP 教程
JSP 环境的搭建以及服务器端软件的安装
关于 CLASSPATH

2002-02-17 22:20

钝刀
管理员

Online
注册日期: Dec 2001
来自: 上海
发帖数量 1093

|

Java Servlet 和 JSP 教程(9-10)
9 Cookie

9.1 Cookie 概述

Cookie 是服务器发送给浏览器的体积很小的纯文本信息，用户以后访问同一个 Web 服务器时浏览器会把它们原样发送给服务器。通过让服务器读取它原先保存到客户端的信息，网站能够为浏览者提供一系列的方便，例如在线交易过程中标识用户身份、安全要求不高的场合避免用户重复输入名字和密码、门户网站的主页定制、有针对性地投放广告，等等。

Cookie 的目的就是为用户带来方便，为网站带来增值。虽然有着许多误传，事实上 Cookie 并不会造成严重的安全威胁。Cookie 永远不会以任何方式执行，因此也不会带来病毒或攻击你的系统。另外，由于浏览器一般只允许存放 300 个 Cookie，每个站点最多存放 20 个 Cookie，每个 Cookie 的大小限制为 4 KB，因此 Cookie 不会塞满你的硬盘，更不会被用作“拒绝服务”攻击手段。

9.2 Servlet 的 Cookie API

要把 Cookie 发送到客户端，Servlet 先要调用 new Cookie(name,value)用合适的名字和值创建一个或多个 Cookie（2.1 节），通过 cookie.setXXX 设置各种属性（2.2 节），通过 response.addCookie(cookie)把 cookie 加入应答头（2.3 节）。

要从客户端读入 Cookie，Servlet 应该调用 request.getCookies()，getCookies()方法返回一个 Cookie 对象的数组。在大多数情况下，你只需要用循环访问该数组的各个元素寻找指定名字的 Cookie，然后对该 Cookie 调用 getValue 方法取得与指定名字关联的值，这部分内容将在 2.4 节讨论。

9.2.1 创建 Cookie

调用 Cookie 对象的构造函数可以创建 Cookie。Cookie 对象的构造函数有两个字符串参数：Cookie 名字和 Cookie 值。名字和值都不能包含空白字符以及下列字符：
[ ] ( ) = , " / ? @ : ;

9.2.2 读取和设置 Cookie 属性

把 Cookie 加入待发送的应答头之前，你可以查看或设置 Cookie 的各种属性。下面摘要介绍这些方法：

getComment/setComment
获取/设置 Cookie 的注释。

getDomain/setDomain
获取/设置 Cookie 适用的域。一般地，Cookie 只返回给与发送它的服务器名字完全相同的服务器。使用这里的方法可以指示浏览器把 Cookie 返回给同一域内的其他服务器。注意域必须以点开始（例如.sitename.com），非国家类的域（如.com，.edu，.gov）必须包含两个点，国家类的域（如.com.cn，.edu.uk）必须包含三个点。

getMaxAge/setMaxAge
获取/设置 Cookie 过期之前的时间，以秒计。如果不设置该值，则 Cookie 只在当前会话内有效，即在用户关闭浏览器之前有效，而且这些 Cookie 不会保存到磁盘上。参见下面有关 LongLivedCookie 的说明。

getName/setName

获取/设置 Cookie 的名字。本质上，名字和值是我们始终关心的两个部分。由于 HttpServletRequest 的 getCookies 方法返回的是一个 Cookie 对象的数组，因此通常要用循环来访问这个数组查找特定名字，然后用 getValue 检查它的值。

getPath/setPath

获取/设置 Cookie 适用的路径。如果不指定路径，Cookie 将返回给当前页面所在目录及其子目录下的所有页面。这里的方法可以用来设定一些更一般的条件。例如，someCookie.setPath("/")，此时服务器上的所有页面都可以接收到该 Cookie。

getSecure/setSecure

获取/设置一个 boolean 值，该值表示是否 Cookie 只能通过加密的连接（即 SSL）发送。

getValue/setValue

获取/设置 Cookie 的值。如前所述，名字和值实际上是我们始终关心的两个方面。不过也有一些例外情况，比如把名字作为逻辑标记（也就是说，如果名字存在，则表示 true）。

getVersion/setVersion

获取/设置 Cookie 所遵从的协议版本。默认版本 0（遵从原先的 Netscape 规范）；版本 1 遵从 RFC 2109 ， 但尚未得到广泛的支持。

9.2.3 在应答头中设置 Cookie

Cookie 可以通过 HttpServletResponse 的 addCookie 方法加入到 Set-Cookie 应答头。下面是一个例子：

```
Cookie userCookie = new Cookie("user", "uid1234");
response.addCookie(userCookie);
```

9.2.4 读取保存到客户端的 Cookie

要把 Cookie 发送到客户端，先要创建 Cookie，然后用 addCookie 发送一个 Set-Cookie HTTP 应答头。这些内容已经在上面的 2.1 节介绍。从客户端读取 Cookie 时调用的是 HttpServletRequest 的 getCookies 方法。该方法返回一个与 HTTP 请求头中的内容对应的 Cookie 对象数组。得到这个数组之后，一般是用循环访问其中的各个元素，调用 getName 检查各个 Cookie 的名字，直至找到目标 Cookie。然后对这个目标 Cookie 调用 getValue，根据获得的结果进行其他处理。

上述处理过程经常会遇到，为方便计下面我们提供一个 getCookieValue 方法。只要给出 Cookie 对象数组、Cookie 名字和默认值，getCookieValue 方法就会返回匹配指定名字的 Cookie 值，如果找不到指定 Cookie，则返回默认值。

9.3 几个 Cookie 工具函数

下面是几个工具函数。这些函数虽然简单，但是，在和 Cookie 打交道的时候很有用。

9.3.1 获取指定名字的 Cookie 值

该函数是 ServletUtilities.java 的一部分。getCookieValue 通过循环依次访问 Cookie 对象数组的各个元素，寻找是否有指定名字的 Cookie，如找到，则返回该 Cookie 的值；否则，返回参数中给出的默认值。getCookieValue 能够在一定程度上简化 Cookie 值的提取。

```
public static String getCookieValue(Cookie[] cookies,
String cookieName,
String defaultValue) {
for(int i=0; i<cookies.length; i++) {
Cookie cookie = cookies[i];
if (cookieName.equals(cookie.getName()))
return(cookie.getValue());
}
return(defaultValue);
}
```

9.3.2 自动保存的 Cookie

下面是 LongLivedCookie 类的代码。如果你希望 Cookie 能够在浏览器退出的时候自动保存下来，则可以用这个 LongLivedCookie 类来取代标准的 Cookie 类。

```
package hall;

import javax.servlet.http.*;

public class LongLivedCookie extends Cookie {
public static final int SECONDS_PER_YEAR = 60*60*24*365;
public LongLivedCookie(String name, String value) {
super(name, value);
setMaxAge(SECONDS_PER_YEAR);
}
}
```

9.4.实例：定制的搜索引擎界面

下面也是一个搜索引擎界面的例子，通过修改前面 HTTP 状态代码的例子得到。在这个 Servlet 中，用户界面是动态生成而不是由静态 HTML 文件提供的。Servlet 除了负责读取表单数据并把它们发送给搜索引擎之外，还要把包含表单数据的 Cookie 发送给客户端。以后客户再次访问同一表单时，这些 Cookie 的值将用来预先填充表单，使表单自动显示最近使用过的数据。

SearchEnginesFrontEnd.java

该 Servlet 构造一个主要由表单构成的用户界面。第一次显示的时候，它和前面用静态 HTML 页面提供的界面差不多。然而，用户选择的值将被保存到 Cookie（本页面将数据发送到 CustomizedSearchEngines Servlet，由后者设置 Cookie）。用户以后再访问同一页面时，即使浏览器是退出之后再启动，表单中也会自动填好上一次搜索所填写的内容。

注意该 Servlet 用到了 ServletUtilities.java，其中 getCookieValue 前面已经介绍过，headWithTitle 用于生成 HTML 页面的一部分。另外，这里也用到了前面已经说明的 LongLiveCookie 类，我们用它来创建作废期限很长的 Cookie。

```
package hall;

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

```
import javax.servlet.http.*;
import java.net.*;

public class SearchEnginesFrontEnd extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
Cookie[] cookies = request.getCookies();
String searchString =
ServletUtilities.getCookieValue(cookies,
"searchString",
"Java Programming");
String numResults =
ServletUtilities.getCookieValue(cookies,
"numResults",
"10");
String searchEngine =
ServletUtilities.getCookieValue(cookies,
"searchEngine",
"google");
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "Searching the Web";
out.println(ServletUtilities.headWithTitle(title) +
"<BODY BGCOLOR=\"#FDF5E6\">\n" +
"<H1 ALIGN=\"CENTER\">Searching the Web</H1>\n" +
"\n" +
"<FORM ACTION=\"/servlet/hall.CustomizedSearchEngines\">\n" +
"<CENTER>\n" +
"Search String:\n" +
"<INPUT TYPE=\"TEXT\" NAME=\"searchString\"\n" +
" VALUE=\"" + searchString + "\"><BR>\n" +
"Results to Show Per Page:\n" +
"<INPUT TYPE=\"TEXT\" NAME=\"numResults\"\n" +
" VALUE=" + numResults + " SIZE=3><BR>\n" +
"<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
" VALUE=\"google\"" +
checked("google", searchEngine) + ">\n" +
"Google |\n" +
"<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
" VALUE=\"infoseek\"" +
checked("infoseek", searchEngine) + ">\n" +
"Infoseek |\n" +
"<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
" VALUE=\"lycos\"" +
checked("lycos", searchEngine) + ">\n" +
"Lycos |\n" +
"<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
" VALUE=\"hotbot\"" +
checked("hotbot", searchEngine) + ">\n" +
"HotBot\n" +
"<BR>\n" +
"<INPUT TYPE=\"SUBMIT\" VALUE=\"Search\">\n" +
```

```
"</CENTER>\n" +
"</FORM>\n" +
"\n" +
"</BODY>\n" +
"</HTML>\n");
}

private String checked(String name1, String name2) {
if (name1.equals(name2))
return(" CHECKED");
else
return("");
}
}
```

CustomizedSearchEngines.java

前面的 SearchEnginesFrontEnd Servlet 把数据发送到 CustomizedSearchEngines Servlet。本例在许多方面与前面介绍 HTTP 状态代码时的例子相似，区别在于，本例除了要构造一个针对搜索引擎的 URL 并向用户发送一个重定向应答之外，还要发送保存用户数据的 Cookies。

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public class CustomizedSearchEngines extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {

String searchString = request.getParameter("searchString");
Cookie searchStringCookie =
new LongLivedCookie("searchString", searchString);
response.addCookie(searchStringCookie);
searchString = URLEncoder.encode(searchString);
String numResults = request.getParameter("numResults");
Cookie numResultsCookie =
new LongLivedCookie("numResults", numResults);
response.addCookie(numResultsCookie);
String searchEngine = request.getParameter("searchEngine");
Cookie searchEngineCookie =
new LongLivedCookie("searchEngine", searchEngine);
response.addCookie(searchEngineCookie);
SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
for(int i=0; i<commonSpecs.length; i++) {
SearchSpec searchSpec = commonSpecs[i];
if (searchSpec.getName().equals(searchEngine)) {
String url =
searchSpec.makeURL(searchString, numResults);
response.sendRedirect(url);
return;
```

```
}
}
response.sendError(response.SC_NOT_FOUND,
"No recognized search engine specified.");
}

public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
doGet(request, response);
}
}
```

10 会话

10.1 会话状态概述

HTTP 协议的“无状态”（Stateless）特点带来了一系列的问题。特别是通过在线商店购物时，服务器不能顺利地记住以前的事务就成了严重的问题。它使得“购物篮”之类的应用很难实现：当我们把商品加入购物篮时，服务器如何才能知道篮子里原先有些什么？即使服务器保存了上下文信息，我们仍旧会在电子商务应用中遇到问题。例如，当用户从选择商品的页面（由普通的服务器提供）转到输入信用卡号和送达地址的页面（由支持 SSL 的安全服务器提供），服务器如何才能记住用户买了些什么？

这个问题一般有三种解决方法：

Cookie。利用 HTTP Cookie 来存储有关购物会话的信息，后继的各个连接可以查看当前会话，然后从服务器的某些地方提取有关该会话的完整信息。这是一种优秀的，也是应用最广泛的方法。然而，即使 Servlet 提供了一个高级的、使用方便的 Cookie 接口，仍旧有一些繁琐的细节问题需要处理：

从其他 Cookie 中分别出保存会话标识的 Cookie。

为 Cookie 设置合适的作废时间（例如，中断时间超过 24 小时的会话一般应重置）。

把会话标识和服务器端相应的信息关联起来。（实际保存的信息可能要远远超过保存到 Cookie 的信息，而且象信用卡号等敏感信息永远不应该用 Cookie 来保存。）

改写 URL。你可以把一些标识会话的数据附加到每个 URL 的后面，服务器能够把该会话标识和它所保存的会话数据关联起来。这也是一个很好的方法，而且还有当浏览器不支持 Cookie 或用户已经禁用 Cookie 的情况下也有效这一优点。然而，大部分使用 Cookie 时所面临的问题同样存在，即服务器端的程序要进行许多简单但单调冗长的处理。另外，还必须十分小心地保证每个 URL 后面都附加了必要的信息（包括非直接的，如通过 Location 给出的重定向 URL）。如果用户结束会话之后又通过书签返回，则会话信息会丢失。

隐藏表单域。HTML 表单中可以包含下面这样的输入域：<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">。这意味着，当表单被提交时，隐藏域的名字和数据也被包含到 GET 或 POST 数据里，我们可以利用这一机制来维持会话信息。然而，这种方法有一个很大的缺点，它要求所有页面都是动态生成的，因为整个问题的核心就是每个会话都要有一个唯一标

识符。

Servlet 为我们提供了一种与众不同的方案：HttpSession API。HttpSession API 是一个基于 Cookie 或者 URL 改写机制的高级会话状态跟踪接口：如果浏览器支持 Cookie，则使用 Cookie；如果浏览器不支持 Cookie 或者 Cookie 功能被关闭，则自动使用 URL 改写方法。Servlet 开发者无需关心细节问题，也无需直接处理 Cookie 或附加到 URL 后面的信息，API 自动为 Servlet 开发者提供一个可以方便地存储会话信息的地方。

10.2 会话状态跟踪 API

在 Servlet 中使用会话信息是相当简单的，主要的操作包括：查看和当前请求关联的会话对象，必要的时候创建新的会话对象，查看与某个会话相关的信息，在会话对象中保存信息，以及会话完成或中止时释放会话对象。

10.2.1 查看当前请求的会话对象

查看当前请求的会话对象通过调用 HttpServletRequest 的 getSession 方法实现。如果 getSession 方法返回 null，你可以创建一个新的会话对象。但更经常地，我们通过指定参数使得不存在现成的会话时自动创建一个会话对象，即指定 getSession 的参数为 true。因此，访问当前请求会话对象的第一个步骤通常如下所示：

```
HttpSession session = request.getSession(true);
```

10.2.2 查看和会话有关的信息

HttpSession 对象生存在服务器上，通过 Cookie 或者 URL 这类后台机制自动关联到请求的发送者。会话对象提供一个内建的数据结构，在这个结构中可以保存任意数量的键-值对。在 2.1 或者更早版本的 Servlet API 中，查看以前保存的数据使用的是 getValue("key")方法。getValue 返回 Object，因此你必须把它转换成更加具体的数据类型。如果参数中指定的键不存在，getValue 返回 null。

API 2.2 版推荐用 getAttribute 来代替 getValue，这不仅是因为 getAttribute 和 setAttribute 的名字更加匹配（和 getValue 匹配的是 putValue，而不是 setValue），同时也因为 setAttribute 允许使用一个附属的 HttpSessionBindingListener 来监视数值，而 putValue 则不能。

但是，由于目前还只有很少的商业 Servlet 引擎支持 2.2，下面的例子中我们仍旧使用 getValue。这是一个很典型的例子，假定 ShoppingCart 是一个保存已购买商品信息的类：

```
HttpSession session = request.getSession(true);
ShoppingCart previousItems =(ShoppingCart)session.getValue("previousItems");
if (previousItems != null) {
  doSomethingWith(previousItems);
} else {
  previousItems = new ShoppingCart(...);
  doSomethingElseWith(previousItems);
}
```

大多数时候我们都是根据特定的名字寻找与它关联的值，但也可以调用 getValueNames 得到所有属性的名字。getValuesNames 返回的是一个 String 数组。API 2.2 版推荐使用 getAttributeNames，这不仅是因为其名字更好，而且因为它返回的是一个 Enumeration，和其他

方法（比如 HttpServletRequest 的 getHeaders 和 getParameterNames）更加一致。

虽然开发者最为关心的往往是保存到会话对象的数据，但还有其他一些信息有时也很有用。

getID：该方法返回会话的唯一标识。有时该标识被作为键-值对中的键使用，比如会话中只保存一个值时，或保存上一次会话信息时。

isNew：如果客户（浏览器）还没有绑定到会话则返回 true，通常意味着该会话刚刚创建，而不是引用自客户端的请求。对于早就存在的会话，返回值为 false。

getCreationTime：该方法返回建立会话的以毫秒计的时间，从 1970.01.01（GMT）算起。要得到用于打印输出的时间值，可以把该值传递给 Date 构造函数，或者 GregorianCalendar 的 setTimeInMillis 方法。

getLastAccessedTime：该方法返回客户最后一次发送请求的以毫秒计的时间，从 1970.01.01（GMT）算起。

getMaxInactiveInterval：返回以秒计的最大时间间隔，如果客户请求之间的间隔不超过该值，Servlet 引擎将保持会话有效。负数表示会话永远不会超时。

10.2.3 在会话对象中保存数据

如上节所述，读取保存在会话中的信息使用的是 getValue 方法（或，对于 2.2 版的 Servlet 规范，使用 getAttribute）。保存数据使用 putValue（或 setAttribute）方法，并指定键和相应的值。注意 putValue 将替换任何已有的值。有时候这正是我们所需要的（如下例中的 referringPage），但有时我们却需要提取原来的值并扩充它（如下例 previousItems）。示例代码如下：

```
HttpSession session = request.getSession(true);
session.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart previousItems = (ShoppingCart)session.getValue("previousItems");
if (previousItems == null) {
    previousItems = new ShoppingCart(...);
}
String itemID = request.getParameter("itemID");
previousItems.addEntry(Catalog.getEntry(itemID));

session.putValue("previousItems", previousItems);
```

10.3 实例：显示会话信息

下面这个例子生成一个 Web 页面，并在该页面中显示有关当前会话的信息。

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import java.util.*;

public class ShowSession extends HttpServlet {
```

```
  public void doGet(HttpServletRequest request,HttpServletResponse response)
  throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Searching the Web";
    String heading;
    Integer accessCount = new Integer(0);;
    if (session.isNew()) {
      heading = "Welcome, Newcomer";
    } else {
      heading = "Welcome Back";
      Integer oldAccessCount =
      // 在 Servlet API 2.2 中使用 getAttribute 而不是 getValue
      (Integer)session.getValue("accessCount");
    if (oldAccessCount != null) {
      accessCount =
      new Integer(oldAccessCount.intValue() + 1);
    }
  }
  // 在 Servlet API 2.2 中使用 putAttribute
  session.putValue("accessCount", accessCount);

  out.println(ServletUtilities.headWithTitle(title) +"<BODY BGCOLOR=\"#FDF5E6\">\n" +
              "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
              "<H2>Information on Your Session:</H2>\n" +
              "<TABLE BORDER=1 ALIGN=CENTER>\n" +
              "<TR BGCOLOR=\"#FFAD00\">\n" +
              " <TH>Info Type<TH>Value\n" +
              "<TR>\n" +
              " <TD>ID\n" +
              " <TD>" + session.getId() + "\n" +
              "<TR>\n" +
              " <TD>Creation Time\n" +
              " <TD>" + new Date(session.getCreationTime()) + "\n" +
              "<TR>\n" +
              " <TD>Time of Last Access\n" +
              " <TD>" + new Date(session.getLastAccessedTime()) + "\n" +
              "<TR>\n" +
              " <TD>Number of Previous Accesses\n" +
              " <TD>" + accessCount + "\n" +
              "</TABLE>\n" +
              "</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
      doGet(request, response);
  }
}
```

凡事太急,缘份势必早尽

Java Servlet 和 JSP 教程
JSP 环境的搭建以及服务器端软件的安装
关于 CLASSPATH

2002-02-17 22:21

钝刀
管理员

Online
注册日期: Dec 2001
来自: 上海
发帖数量 1093

|

Java Servlet 和 JSP 教程(11-13)
11 JSP

11.1 JSP 概述

JavaServer Pages（JSP）使得我们能够分离页面的静态 HTML 和动态部分。HTML 可以用任何通常使用的 Web 制作工具编写，编写方式也和原来的一样；动态部分的代码放入特殊标记之内，大部分以“<%”开始，以“%>”结束。例如，下面是一个 JSP 页面的片断，如果我们用 http://host/OrderConfirmation.jsp?title=Core+Web+Programming 这个 URL 打开该页面，则结果显示“Thanks for ordering Core Web Programming”。

```
Thanks for ordering
<I><%= request.getParameter("title") %></I>
```

JSP 页面文件通常以.jsp 为扩展名，而且可以安装到任何能够存放普通 Web 页面的地方。虽然从代码编写来看，JSP 页面更象普通 Web 页面而不象 Servlet，但实际上，JSP 最终会被转换成正规的 Servlet，静态 HTML 直接输出到和 Servlet service 方法关联的输出流。

JSP 到 Servlet 的转换过程一般在出现第一次页面请求时进行。因此，如果你希望第一个用户不会由于 JSP 页面转换成 Servlet 而等待太长的时间，希望确保 Servlet 已经正确地编译并装载，你可以在安装 JSP 页面之后自己请求一下这个页面。

另外也请注意，许多 Web 服务器允许定义别名，所以一个看起来指向 HTML 文件的 URL 实际上可能指向 Servlet 或 JSP 页面。

除了普通 HTML 代码之外，嵌入 JSP 页面的其他成分主要有如下三种：脚本元素（Scripting Element），指令（Directive），动作（Action）。脚本元素用来嵌入 Java 代码，这些 Java 代码将成为转换得到的 Servlet 的一部分；JSP 指令用来从整体上控制 Servlet 的结构；动作用来引入现有的组件或者控制 JSP 引擎的行为。为了简化脚本元素，JSP 定义了一组可以直接使用的变量

（预定义变量），比如前面代码片断中的 request 就是其中一例。

注意本文以 JSP 1.0 规范为基础。和 0.92 版相比，新版本的 JSP 作了许多重大的改动。虽然这些改动只会使 JSP 变得更好，但应注意 1.0 的 JSP 页面几乎和早期的 JSP 引擎完全不兼容。

11.2 JSP 语法概要表

JSP 元素 语法 说明 备注
JSP 表达式 <%= expression %> 计算表达式并输出结果 等价的 XML 表达是：
<jsp:expression>
expression
</jsp:expression>

可以使用的预定义变量包括：request，response，out，session，application，config，pageContext。这些预定义变量也可以在 JSP Scriptlet 中使用。
JSP Scriptlet <% code %> 插入到 service 方法的代码。 等价的 XML 表达是：
<jsp:scriptlet>
code
</jsp:scriptlet>

JSP 声明 <%! code %> 代码被插入到 Servlet 类（在 service 方法之外） 等价的 XML 表达是：

<jsp:declaration>
code
</jsp:declaration>
page 指令 <%@ page att="val" %> 作用于 Servlet 引擎的全局性指令 等价的 XML 表达是
<jsp:directive.page att="val"\>

合法的属性如下表，其中粗体表示默认值：
import="package.class"
contentType="MIME-Type"
isThreadSafe="true|false"
session="true|false"
buffer="size kb|none"
autoflush="true|false"
extends="package.class"
info="message"
errorPage="url"
isErrorPage="true|false"
language="java"
include 指令 <%@ include file="url" %> 当 JSP 转换成 Servlet 时，应当包含本地系统上的指定文件 等价的 XML 表达是：
<jsp:directive.include file="url"\>

其中 URL 必须是相对 URL。

利用 jsp:include 动作可以在请求的时候（而不是 JSP 转换成 Servlet 时）引入文件。
JSP 注释 <%-- comment --%> 注释；JSP 转换成 Servlet 时被忽略 如果要把注释嵌入结果 HTML 文档，使用普通的 HTML 注释标记<-- comment -->。
jsp:include 动作 <jsp:include page="relative URL" flush="true"/> 当 Servlet 被请求时，引入指定的文件 如果你希望在页面转换的时候包含某个文件，使用 JSP include 指令

注意：在某些服务器上，被包含文件必须是 HTML 文件或 JSP 文件，具体由服务器决定（通常根据文件扩展名判断）。
jsp:useBean 动作 <jsp:useBean att=val*/> 或者
<jsp:useBean att=val*>
...
</jsp:useBean> 寻找或实例化一个 Java Bean 可能的属性包括：
id="name"
scope="page|request
|session|application"
class="package.class"
type="package.class"
beanName="package.class"
jsp:setProperty 动作 <jsp:setProperty att=val*/> 设置 Bean 的属性，既可以设置一个确定的值，也可以指定属性值来自请求参数。 合法的属性包括：
name="beanName"
property="propertyName|*"
param="parameterName"
value="val"

jsp:getProperty 动作 <jsp:getProperty
name="propertyName"
value="val"/> 提取并输出 Bean 的属性。
jsp:forward 动作 <jsp:forward
page="relative URL"/> 把请求转到另外一个页面。
jsp:plugin 动作 <jsp:plugin
attribute="value"*>
...
</jsp:plugin> 根据浏览器类型生成 OBJECT 或者 EMBED 标记，以便通过 Java Plugin 运行 Java Applet。

11.3 关于模板文本（静态 HTML）

许多时候，JSP 页面的很大一部分都由静态 HTML 构成，这些静态 HTML 也称为“模板文本”。模板文本和普通 HTML 几乎完全相同，它们都遵从相同的语法规则，而且模板文本也是被 Servlet 直接发送到客户端。此外，模板文本也可以用任何现有的页面制作工具来编写。

唯一的例外在于，如果要输出“<%”，则模板文本中应该写成“<\%”。

12

12.1 JSP 脚本元素

JSP 脚本元素用来插入 Java 代码，这些 Java 代码将出现在由当前 JSP 页面生成的 Servlet 中。脚本元素有三种格式：

表达式格式<%= expression %>：计算表达式并输出其结果。

Scriptlet 格式<% code %>：把代码插入到 Servlet 的 service 方法。

声明格式<%! code %>：把声明加入到 Servlet 类（在任何方法之外）。

下面我们详细说明它们的用法。

12.1.1 JSP 表达式

JSP 表达式用来把 Java 数据直接插入到输出。其语法如下：

<%= Java Expression %>

计算 Java 表达式得到的结果被转换成字符串，然后插入到页面?扑阁谠诵惺苯　校丁趁姹磺肭第保　　虼丝梢苑梦屎颓肭笥泄氐娜　啃畔ｉ@　纾　旅娴拇　胂允疽趁姹磺肭蟮娜掌?时间：
Current time: <%= new java.util.Date() %>

为简化这些表达式，JSP 预定义了一组可以直接使用的对象变量。后面我们将详细介绍这些隐含声明的对象，但对于 JSP 表达式来说，最重要的几个对象及其类型如下：

request：HttpServletRequest;
response：HttpServletResponse;
session：和 request 关联的 HttpSession
out：PrintWriter（带缓冲的版本，JspWriter），用来把输出发送到客户端

下面是一个例子：

Your hostname: <%= request.getRemoteHost() %>

最后，如果使用 XML 的话，JSP 表达式也可以写成下面这种形式：

<jsp:expression>
Java Expression
</jsp:expression>

请记住 XML 元素和 HTML 不一样。XML 是大小写敏感的，因此务必使用小写。有关 XML 语法的说明，请参见《XML 教程 》

12.1.2 JSP Scriptlet

如果你要完成的任务比插入简单的表达式更加复杂，可以使用 JSP Scriptlet。JSP Scriptlet 允许你把任意的 Java 代码插入 Servlet。JSP Scriptlet 语法如下：

<% Java Code %>

和 JSP 表达式一样，Scriptlet 也可以访问所有预定义的变量。例如，如果你要向结果页面输出内容，可以使用 out 变量：

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

注意 Scriptlet 中的代码将被照搬到 Servlet 内，而 Scriptlet 前面和后面的静态 HTML（模板文本）将被转换成 println 语句。这就意味着，Scriptlet 内的 Java 语句并非一定要是完整的，没有关闭的块将影响 Scriptlet 外的静态 HTML。例如，下面的 JSP 片断混合了模板文本和 Scriptlet:

```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!
<% } else { %>
Have a <B>lousy</B> day!
<% } %>
```

上述 JSP 代码将被转换成如下 Servlet 代码：

```
if (Math.random() < 0.5) {
out.println("Have a <B>nice</B> day!");
} else {
out.println("Have a <B>lousy</B> day!");
}
```

如果要在 Scriptlet 内部使用字符“%>”，必须写成“%\>”。另外，请注意<% code %>的 XML 等价表达是：

```
<jsp:scriptlet>
Code
</jsp:scriptlet>
```

12.1.3 JSP 声明

JSP 声明用来定义插入 Servlet 类的方法和成员变量，其语法如下：

```
<%! Java Code %>
```

由于声明不会有任何输出，因此它们往往和 JSP 表达式或 Scriptlet 结合在一起使用。例如，下面的 JSP 代码片断输出自从服务器启动（或 Servlet 类被改动并重新装载以来）当前页面被请求的次数：

```
<%! private int accessCount = 0; %>
```

自从服务器启动以来页面访问次数为：

```
<%= ++accessCount %>
```

和 Scriptlet 一样，如果要使用字符串“%>”，必须使用“%\>”代替。最后，<%! code %>的 XML 等价表达方式为：

```
<jsp:declaration>
Code
</jsp:declaration>
```

12.2 JSP 指令

JSP 指令影响 Servlet 类的整体结构，它的语法一般如下：

<%@ directive attribute="value" %>

另外，也可以把同一指令的多个属性结合起来，例如：

<%@ directive attribute1="value1"
attribute2="value2"
...
attributeN="valueN" %>

JSP 指令分为两种类型：第一是 page 指令，用来完成下面这类任务：导入指定的类，自定义 Servlet 的超类，等等；第二是 include 指令，用来在 JSP 文件转换成 Servlet 时引入其他文件。JSP 规范也提到了 taglib 指令，其目的是让 JSP 开发者能够自己定义标记，但 JSP 1.0 不支持该指令，有希望它将成为 JSP 1.1 的主要改进之一。

12.2.1 page 指令

page 指令的作用是定义下面一个或多个属性，这些属性大小写敏感。

import="package.class"，或者 import="package.class1,...,package.classN"：

用于指定导入哪些包，例如：<%@ page import="java.util.*" %>。import 是唯一允许出现一次以上的属性。

contentType="MIME-Type" 或 contentType="MIME-Type; charset=Character-Set"：

该属性指定输出的 MIME 类型。默认是 text/html。例如，下面这个指令：

<%@ page contentType="text/plain" %>。

和下面的 Scriptlet 效果相同：

<% response.setContentType("text/plain"); %>
isThreadSafe="true|false"

默认值 true 表明 Servlet 按照标准的方式处理，即假定开发者已经同步对实例变量的访问，由单个 Servlet 实例同时地处理多个请求。如果取值 false，表明 Servlet 应该实现 SingleThreadModel，请求或者是逐个进入，或者多个并行的请求分别由不同的 Servlet 实例处理。

session="true|false"

默认值 true 表明预定义变量 session（类型为 HttpSession）应该绑定到已有的会话，如果不存在已有的会话，则新建一个并绑定 session 变量。如果取值 false，表明不会用到会话，试图访问变量 session 将导致 JSP 转换成 Servlet 时出错。

buffer="size kb|none"

该属性指定 JspWrite out 的缓存大小。默认值和服务器有关，但至少应该是 8 KB。

autoflush="true|false"

默认值 true 表明如果缓存已满则刷新它。autoflush 很少取 false 值，false 值表示如果缓存已满则抛出异常。如果 buffer="none"，autoflush 不能取 false 值。

extends="package.class"

该属性指出将要生成的 Servlet 使用哪个超类。使用该属性应当十分小心，因为服务器可能已经在用自定义的超类。

info="message"

该属性定义一个可以通过 getServletInfo 方法提取的字符串。

errorPage="url"

该属性指定一个 JSP 页面，所有未被当前页面捕获的异? 筛靡趁娲 怼?

isErrorPage="true|false"

该属性指示当前页面是否可以作为另一 JSP 页面的错误处理页面。默认值 false。

language="java"

该属性用来指示所使用的语言。目前没有必要关注这个属性，因为默认的 Java 是当前唯一可用的语言。

定义指令的 XML 语法为：

<jsp:directive.directiveType attribute=value />

例如，下面这个指令：

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

它的 XML 等价表达是：

<jsp:directive.page import="java.util.*" />

12.2.2 include 指令

include 指令用于 JSP 页面转换成 Servlet 时引入其他文件。该指令语法如下：

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

这里所指定的 URL 是和发出引用指令的 JSP 页面相对的 URL，然而，与通常意义上的相对 URL 一样，你可以利用以“/”开始的 URL 告诉系统把 URL 视为从 Web 服务器根目录开始。包含文件的内容也是 JSP 代码，即包含文件可以包含静态 HTML、脚本元素、JSP 指令和动作。

例如，许多网站的每个页面都有一个小小的导航条。由于 HTML 框架存在不少问题，导航条往往用页面顶端或左边的一个表格制作，同一份 HTML 代码重复出现在整个网站的每个页面

上。include 指令是实现该功能的非常理想的方法。使用 include 指令，开发者不必再把导航 HTML 代码拷贝到每个文件中，从而可以更轻松地完成维护工作。

由于 include 指令是在 JSP 转换成 Servlet 的时候引入文件，因此如果导航条改变了，所有使用该导航条的 JSP 页面都必须重新转换成 Servlet。如果导航条改动不频繁，而且你希望包含操作具有尽可能好的效率，使用 include 指令是最好的选择。然而，如果导航条改动非常频繁，你可以使用 jsp:include 动作。jsp:include 动作在出现对 JSP 页面请求的时候才会引用指定的文件，请参见本文后面的具体说明。

12.3 实例：脚本元素和指令的应用

下面是一个使用 JSP 表达式、Scriptlet、声明、指令的简单例子。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JavaServer Pages</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
VLINK="#551A8B" ALINK="#FF0000">
<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
<TR><TH CLASS="TITLE">
JSP 应用实例</TABLE>
</CENTER>
<P>
下面是一些利用各种 JSP 功能生成的动态内容:
<UL>
<LI><B>表达式.</B><BR>
你的主机名: <%= request.getRemoteHost() %>.
<LI><B>JSP Scriptlet.</B><BR>
<% out.println("查询字符串: " +
request.getQueryString()); %>
<LI><B>声明（和表达式）.</B><BR>
<%! private int accessCount = 0; %>
服务器启动以来访问次数: <%= ++accessCount %>
<LI><B>指令（和表达式）.</B><BR>
<%@ page import = "java.util.*" %>
当前日期: <%= new Date() %>
</UL>

</BODY>
</HTML>
```

12.4 JSP 预定义变量

为了简化 JSP 表达式和 Scriptlet 的代码，JSP 提供了 8 个预先定义的变量（或称为隐含对象）。这些变量是 request、response、out、session、application、config、pageContext 和 page。

12.4.1 request

这是和请求关联的 HttpServletRequest，通过它可以查看请求参数（调用 getParameter），请求类型（GET，POST，HEAD，等），以及请求的 HTTP 头（Cookie，Referer，等）。严格说来，如果请求所用的是 HTTP 之外的其他协议，request 可以是 ServletRequest 的子类（而不是 HttpServletRequest），但在实践中几乎不会用到。

12.4.2 response

这是和应答关联的 HttpServletResponse。注意，由于输出流（参见下面的 out）是带缓冲的，因此，如果已经向客户端发送了输出内容，普通 Servlet 不允许再设置 HTTP 状态代码，但在 JSP 中却是合法的。

12.4.3 out

这是用来向客户端发送内容的 PrintWriter。然而，为了让 response 对象更为实用，out 是带缓存功能的 PrintWriter，即 JspWriter。JSP 允许通过 page 指令的 buffer 属性调整缓存的大小，甚至允许关闭缓存。

out 一般只在 Scriptlet 内使用，这是因为 JSP 表达式是自动发送到输出流的，很少需要显式地引用 out。

12.4.4 session

这是和请求关联的 HttpSession 对象。前面我们已经介绍过会话的自动创建，我们知道，即使不存在 session 引用，这个对象也是自动绑定的。但有一个例外，这就是如果你用 page 指令的 session 属性关闭了会话，此时对 session 变量的引用将导致 JSP 页面转换成 Servlet 时出错。

12.4.5 application

这是一个 ServletContext，也可以通过 getServletConfig().getContext()获得。

12.4.6 config

这是当前页面的 ServletConfig 对象。

12.4.7 pageContext

主要用来管理页面的属性。

12.4.8 page

它是 this 的同义词，当前用处不大。它是为了 Java 不再是唯一的 JSP 编程语言而准备的占位符。

JSP 动作利用 XML 语法格式的标记来控制 Servlet 引擎的行为。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。

JSP 动作包括：

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。

13

13.1 jsp:include 动作

该动作把指定文件插入正在生成的页面。其语法如下：

<jsp:include page="relative URL" 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。jsp:include 动作的文件引入时间决定了它的效率要稍微差一点，而且被引用文件不能包含某些 JSP 代码（例如不能设置 HTTP 头），但它的灵活性却要好得多。

例如，下面的 JSP 页面把 4 则新闻摘要插入一个“What's New ?”页面。改变新闻摘要时只需改变这四个文件，而主 JSP 页面却可以不作修改：

WhatsNew.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>What's New</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
VLINK="#551A8B" ALINK="#FF0000">

<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
<TR><TH CLASS="TITLE">
What's New at JspNews.com</TABLE>
</CENTER>
<P>
Here is a summary of our four most recent news stories:
<OL>
<LI><jsp:include page="news/Item1.html" flush="true"/>
<LI><jsp:include page="news/Item2.html" flush="true"/>
<LI><jsp:include page="news/Item3.html" flush="true"/>
<LI><jsp:include page="news/Item4.html" flush="true"/>
</OL>
```

```
</BODY>
</HTML>
```

13.2 jsp:useBean 动作

jsp:useBean 动作用来装载一个将在 JSP 页面中使用的 JavaBean。这个功能非常有用，因为它使得我们既可以发挥 Java 组件重用的优势，同时也避免了损失 JSP 区别于 Servlet 的方便性。jsp:useBean 动作最简单的语法为：

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

这行代码的含义是："创建一个由 class 属性指定的类的实例，然后把它绑定到其名字由 id 属性给出的变量上"。不过，就象我们接下来会看到的，定义一个 scope 属性可以让 Bean 关联到更多的页面。此时，jsp:useBean 动作只有在不存在同样 id 和 scope 的 Bean 时才创建新的对象实例，同时，获得现有 Bean 的引用就变得很有必要。

获得 Bean 实例之后，要修改 Bean 的属性既可以通过 jsp:setProperty 动作进行，也可以在 Scriptlet 中利用 id 属性所命名的对象变量，通过调用该对象的方法显式地修改其属性。这使我们想起，当我们说"某个 Bean 有一个类型为 X 的属性 foo"时，就意味着"这个类有一个返回值类型为 X 的 getFoo 方法，还有一个 setFoo 方法以 X 类型的值为参数"。

有关 jsp:setProperty 动作的详细情况在后面讨论。但现在必须了解的是，我们既可以通过 jsp:setProperty 动作的 value 属性直接提供一个值，也可以通过 param 属性声明 Bean 的属性值来自指定的请求参数，还可以列出 Bean 属性表明它的值应该来自请求参数中的同名变量。

在 JSP 表达式或 Scriptlet 中读取 Bean 属性通过调用相应的 getXXX 方法实现，或者更一般地，使用 jsp:getProperty 动作。

注意包含 Bean 的类文件应该放到服务器正式存放 Java 类的目录下，而不是保留给修改后能够自动装载的类的目录。例如，对于 Java Web Server 来说，Bean 和所有 Bean 用到的类都应该放入 classes 目录，或者封装进 jar 文件后放入 lib 目录，但不应该放到 servlets 下。

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。

BeanTest.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
</HEAD>

<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
Reusing JavaBeans in JSP</TABLE>
</CENTER>
<P>

<jsp:useBean id="test" class="hall.SimpleBean" />
```

```
<jsp:setProperty name="test"
property="message"
value="Hello WWW" />
<H1>Message: <I>
<jsp:getProperty name="test" property="message" />
</I></H1>

</BODY>
</HTML>
```

SimpleBean.java

BeanTest 页面用到了一个 SimpleBean。SimpleBean 的代码如下：

```
package hall;

public class SimpleBean {
private String message = "No message specified";

public String getMessage() {
return(message);
}

public void setMessage(String message) {
this.message = message;
}
}
```

13.3 关于 jsp:useBean 的进一步说明

使用 Bean 最简单的方法是先用下面的代码装载 Bean：

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

然后通过 jsp:setProperty 和 jsp:getProperty 修改和提取 Bean 的属性。不过有两点必须注意。第一，我们还可以用下面这种格式实例化 Bean：

```
<jsp:useBean ...>
Body
</jsp:useBean>
```

它的意思是，只有当第一次实例化 Bean 时才执行 Body 部分，如果是利用现有的 Bean 实例则不执行 Body 部分。正如下面将要介绍的，jsp:useBean 并非总是意味着创建一个新的 Bean 实例。

第二，除了 id 和 class 外，jsp:useBean 还有其他三个属性，即：scope，type，beanName。下表简要说明这些属性的用法。

表示该 Bean 在当前的客户请求内有效（保存在 ServletRequest 对象内）。 属性 用法
id 命名引用该 Bean 的变量。如果能够找到 id 和 scope 相同的 Bean 实例，jsp:useBean 动作将使用已有的 Bean 实例而不是创建新的实例。

class 指定 Bean 的完整包名。
scope 指定 Bean 在哪种上下文内可用，可以取下面的四个值之一：page，request，session 和 application。默认值是 page，表示该 Bean 只在当前页面内可用（保存在当前页面的 PageContext 内）。request 表示该 Bean 在当前的客户请求内有效（保存在 ServletRequest 对象内）。session 表示该 Bean 对当前 HttpSession 内的所有页面都有效。最后，如果取值 application，则表示该 Bean 对所有具有相同 ServletContext 的页面都有效。scope 之所以很重要，是因为 jsp:useBean 只有在不存在具有相同 id 和 scope 的对象时才会实例化新的对象；如果已有 id 和 scope 都相同的对象则直接使用已有的对象，此时 jsp:useBean 开始标记和结束标记之间的任何内容都将被忽略。
type 指定引用该对象的变量的类型，它必须是 Bean 类的名字、超类名字、该类所实现的接口名字之一。请记住变量的名字是由 id 属性指定的。
beanName 指定 Bean 的名字。如果提供了 type 属性和 beanName 属性，允许省略 class 属性。

13.4 jsp:setProperty 动作

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:useBean id="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:useBean id="myName" ... >
...
<jsp:setProperty name="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：

属性 说明
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。

例如，下面的代码片断表示：如果存在 numItems 请求参数的话，把 numberOfItems 属性的值设置为请求参数 numItems 的值；否则什么也不做。

```
<jsp:setProperty name="orderBean"
property="numberOfItems"
param="numItems" />
```

如果同时省略 value 和 param，其效果相当于提供一个 param 且其值等于 property 的值。进一步利用这种借助请求参数和属性名字相同进行自动赋值的思想，你还可以在 property（Bean 属性的名字）中指定“*”，然后省略 value 和 param。此时，服务器会查看所有的 Bean 属性和请求参数，如果两者名字相同则自动赋值。

下面是一个利用 JavaBean 计算素数的例子。如果请求中有一个 numDigits 参数，则该值被传递给 Bean 的 numDigits 属性；numPrimes 也类似。

JspPrimes.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>在 JSP 中使用 JavaBean</TITLE>
</HEAD>

<BODY>

<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
在 JSP 中使用 JavaBean</TABLE>
</CENTER>
<P>

<jsp:useBean id="primeTable" class="hall.NumberedPrimes" />
<jsp:setProperty name="primeTable" property="numDigits" />
<jsp:setProperty name="primeTable" property="numPrimes" />

Some <jsp:getProperty name="primeTable" property="numDigits" />
digit primes:
<jsp:getProperty name="primeTable" property="numberedList" />

</BODY>
</HTML>
```

注：NumberedPrimes 的代码略。

13.5 jsp:getProperty 动作

jsp:getProperty 动作提取指定 Bean 属性的值，转换成字符串，然后输出。jsp:getProperty 有两个必需的属性，即：name，表示 Bean 的名字；property，表示要提取哪个属性的值。下面是一个例子，更多的例子可以在前文找到。

```
<jsp:useBean id="itemBean" ... />
...
<UL>
<LI>Number of items:
<jsp:getProperty name="itemBean" property="numItems" />
<LI>Cost of each:
<jsp:getProperty name="itemBean" property="unitCost" />
</UL>
```

13.6 jsp:forward 动作

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。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，如下面的例子所示：

```
<jsp:forward page="/utils/errorReporter.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />
```

13.7 jsp:plugin 动作

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。

附录：JSP 注释和字符引用约定

下面是一些特殊的标记或字符，你可以利用它们插入注释或可能被视为具有特殊含义的字符。

语法 用途
<%-- comment --%> JSP 注释，也称为“隐藏注释”。JSP 引擎将忽略它。标记内的所有 JSP 脚本元素、指令和动作都将不起作用。
<!-- comment --> HTML 注释，也称为“输出的注释”，直接出现在结果 HTML 文档中。标记内的所有 JSP 脚本元素、指令和动作正常执行。
<\% 在模板文本（静态 HTML）中实际上希望出现“<%”的地方使用。
%\> 在脚本元素内实际上希望出现“%>”的地方使用。
\' 使用单引号的属性内的单引号。不过，你既可以使用单引号也可以使用双引号，而另外一种引号将具有普通含义。
\" 使用双引号的属性内的双引号。参见“\”的说明。