



JSP 2.0 技术手册

作者：林上杰、林康司

出版：电子工业出版社

内容简介：

本书图文并茂，以丰富的实例为引导，全面介绍了主流的 Java Web 开发技术——JSP 2.0，重点介绍 Java 在展示层的两项重要技术：Java Servlet 与 JavaServer Pages。它们是最重要的 Java 核心技术。对这两项技术的深入了解，将有助于您未来对于 JavaServer Faces(JSF)技术以及 Java Web Services 技术的学习。

注：

本内容仅供学习研究参考，为尊重作者劳动成果及版权，请购买原版书籍。

第一章 安装执行环境

1-1 安装 J2SDK 1.4.2

第一步：执行 j2sdk-1_4_2_03-windows-i586-p.exe（见图 1-2）；

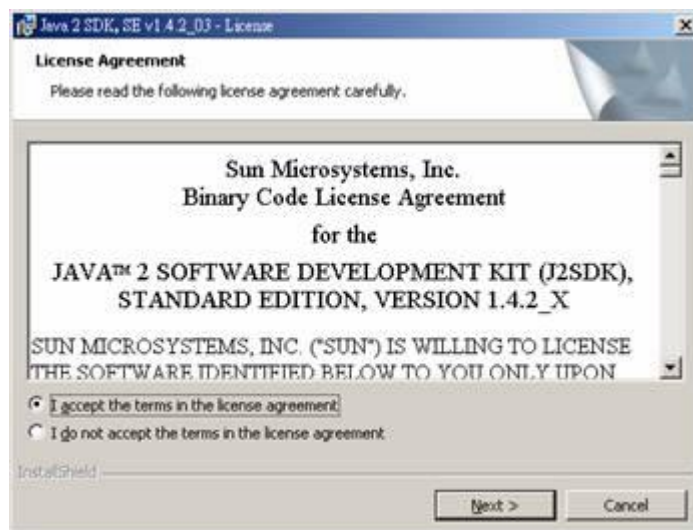


图 1-2 执行 j2sdk-1_4_2_03-windows-i586-p.exe

先按【Next】，选择【I accept the terms in the license agreement】后，再按【Next】。

第二步：选择安装路径及安装内容（见图 1-3）；

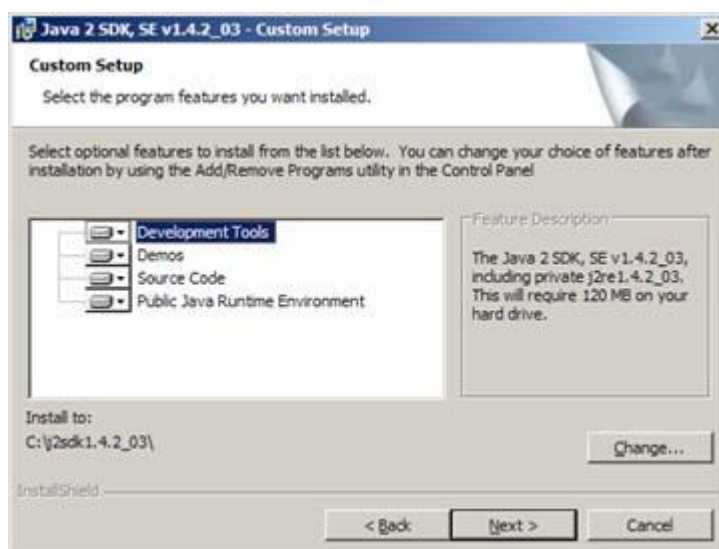


图 1-3 选择安装路径及安装内容

一般来说，我们通常都默认安装在 C:\j2sdk1.4.2_03 的目录下，假若你要安装在其他路径时，请按【Change】，改变 J2SDK 安装路径。确认无误后，再按【Next】。

随后开始安装 Java Plug-in 至浏览器上，一般都选择【Microsoft Internet Explorer】。再按下【Install】，正式开始执行安装程序，假若安装成功后，你会看到如图 1-4。



图 1-4 成功安装 J2SDK 1.4.2_03

第三步：设定 J2SDK 1.4.2_03（见图 1-5）；

从【开始】→【设置】→【控制面板】→【系统】→【高级】→【环境变量】→【系统变量】，然后到【新建】。

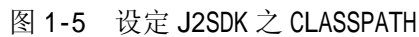
```
JAVA_HOME = C:\j2sdk1.4.2_03
```

```
PATH = %JAVA_HOME%\bin
```

```
CLASSPATH = C:\j2sdk1.4.2_03\lib\tools.jar;C:\j2sdk1.4.2_03\lib\dt.jar;
```

注意

1. CLASSPATH 的设定中，分号(;)用来分开两路径，切勿任意空格；
2. CLASSPATH 的设定中，分号的最后还有一个点“.”。



不论 Windows 2000 或 Windows XP 皆可依上述方法设定。

撰写一个 HelloWorld.java 程序，放置在 C:\HelloWorld.java 中。

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
  
        System.out.println("Hello World");  
  
    }  
  
}
```

打开命令提示符，在 C:\ 下输入 `javac HelloWorld.java`，然后再输入 `java HelloWorld`，执行 HelloWorld 程序，假若顺利成功，则会显示“Hello World”，如图 1-6 所示。



图 1-6 编译且执行 HelloWorld 程序

成功安装 J2SDK 1.4.2_03 之后，紧接下来安装 Tomcat 5.0.16。

1-2 安装 Tomcat 5.0.16

Tomcat 目前版本为 5.0.16，它是由 JavaSoft 和 Apache 开发团队共同提出合作计划(Apache Jakarta Project)下的产品。Tomcat 能支持 Servlet 2.4 和 JSP 2.0 并且是免费使用。

Tomcat 5.0.16 可以从 <http://jakarta.apache.org/tomcat/index.html> 网站自行免费下载，或者可以直接使用本书 CD 光盘中的 Tomcat 5.0.16，软件名称为：jakarta-tomcat-5.0.16.exe。

第一步：执行 jakarta-tomcat-5.0.16.exe（见图 1-7）；

先按【Next】，选择【I Agree】后，再按【Next】。

第二步：选择安装路径及安装内容（见图 1-8）；

通常我们会选择完全安装(Full)，即如图 1-8。在图 1-9【Tomcat】的选项中，主要有：Core、Service、Source Code 和 Documentation，假若选择安装 Service 时，尔后我们可以利用 Windows 的服务(控制面板 | 管理工具 | 服务)来设定重新开机启动时，Tomcat 能够自动启动。



图 1-7 执行 jakarta-tomcat-5.0.16.exe

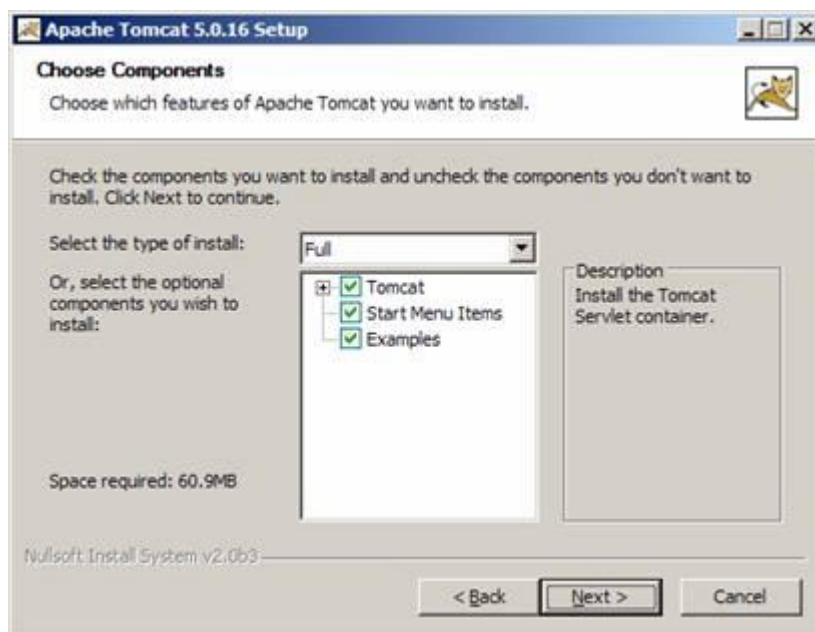


图 1-8 选择安装内容

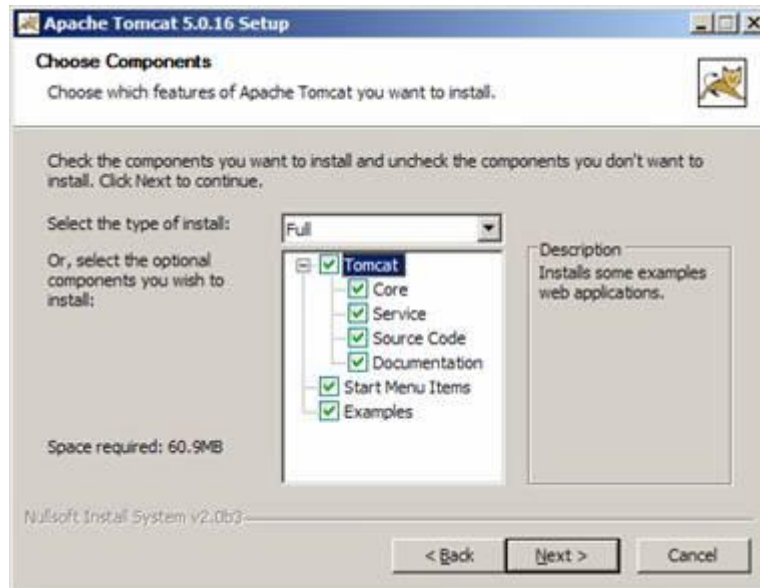


图 1-9 选择安装 Service

选择完全安装后，按【Next】。开始选择安装路径，如图 1-10 所示。

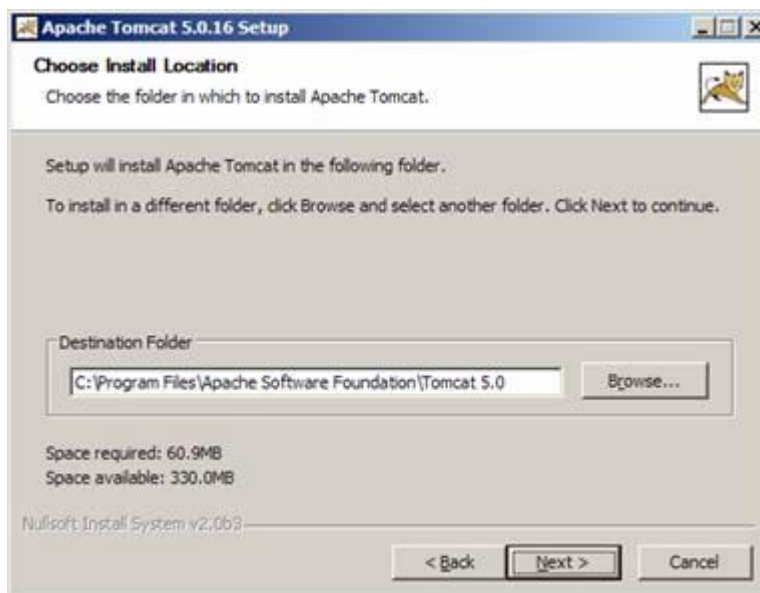


图 1-10 选择安装路径

第三步：设定 Tomcat Port 和 Administrator Login（见图 1-11）；

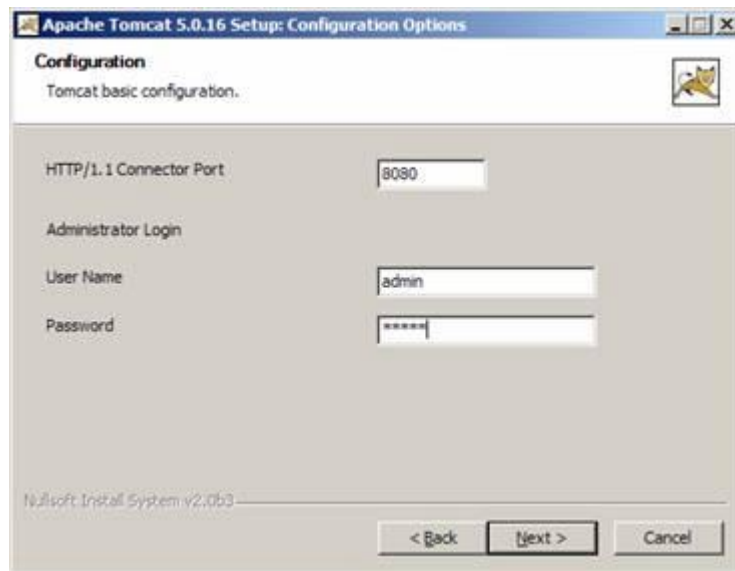


图 1-11 设定 Tomcat Port 和 Administrator Login

第四步：设定 Tomcat 使用的 JVM（见图 1-12）；

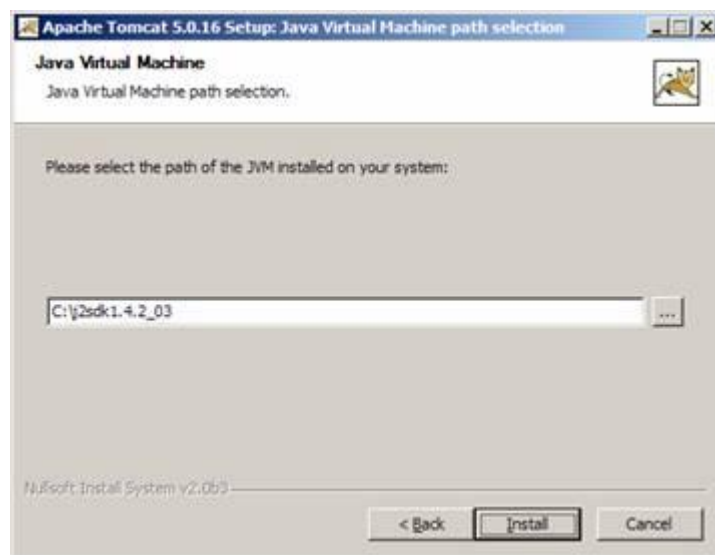


图 1-12 选择 Tomcat 使用的 JVM

确认无误后，按下【Install】，正式开始执行安装程序。安装成功后，会看到如图 1-13 的结果。

假若你勾选了【Run Apache Tomcat】，按下【Finish】之后，会直接启动 Tomcat 5.0.16，然后在你计算机的右下角，会出现绿色箭头的符号，如图 1-14。

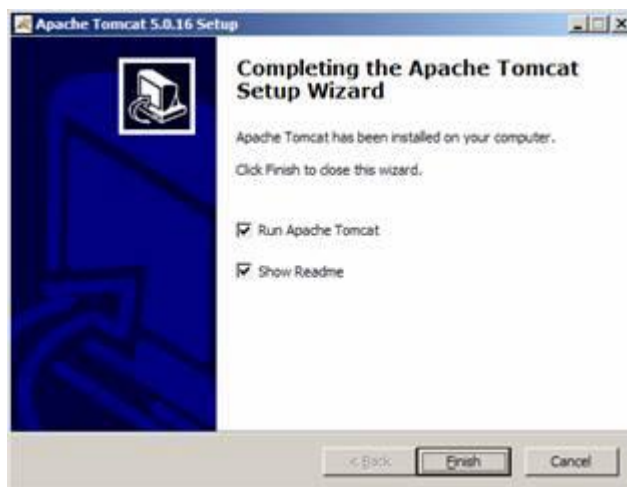


图 1-13 成功安装 Tomcat 5.0.16

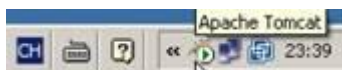


图 1-14 Tomcat 图标

第五步：测试 Tomcat。

打开浏览器，如 IE，输入 <http://localhost:8080>，假若 Tomcat 安装成功，则会看到如图 1-15 的情形。



图 1-15 连接 <http://localhost:8080/>，测试 Tomcat 5.0.16

本书“第十二章：JSP 执行环境与开发工具”，对于 Tomcat 的使用及设定有更详细的介绍。

1-3 安装 JSPBook 站台范例

读者可以在 CD 光盘中找到本书的范例，程序文件名为 JSPBook.war。

第一步：安装 JSPBook.war；

安装的方法很简单，只要将 JSPBook.war 移至 {Tomcat_Install}\webapps\ 目录下 (例如：C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook.war)，然后 JSPBook.war 会自动被 Tomcat 解压缩成 JSPBook 的目录，如图 1-16。

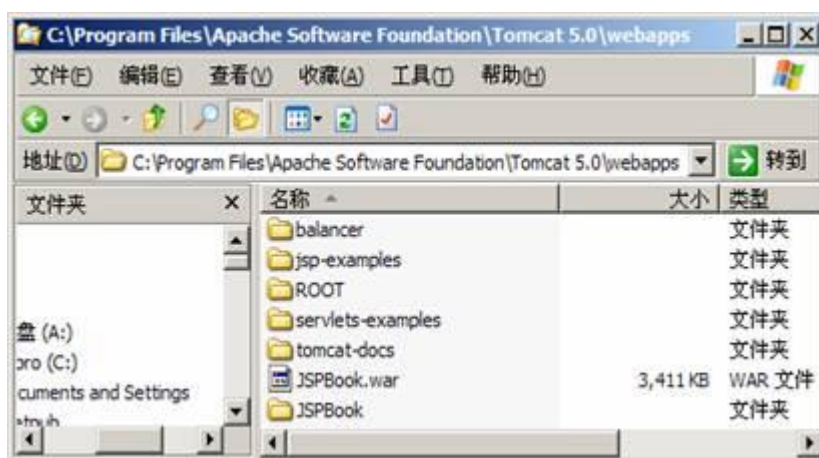


图 1-16 安装 JSPBook.war

第二步：设定 JSPBook 站台；

在 Tomcat 上建立一个 JSPBook 站台时，我们须修改 Tomcat 的 server.xml 文件，server.xml 位于 {Tomcat_Install}\conf\server.xml (例如：C:\Program Files\Apache Software Foundation\Tomcat 5.0\conf\server.xml)。

server.xml

```
.....

<!-- Tomcat Root Context -->

<!--

    <Context path="" docBase="ROOT" debug="0">

-->
```

```
<Context path="/JSPBook" docBase="JSPBook" debug="0"
    crosscontext="true" reloadable="true" >

</Context>

</Host>

</Engine>

</Service>

</Server>
```

这部分主要是设定 JSPBook 站台，其中 `path="/JSPBook"` 代表网域名称，即 `http://IP_DomaninName/JSPBook`；`docBase="JSPBook"` 代表站台的目录位置，即 `{Tomcat_Install}\webapps\JSPBook`；`debug` 则是设定 `debug level`，0 表示提供最少的信息，9 表示提供最多的信息；`reloadable` 则表示 Tomcat 在执行时，当 `class`、`web.xml` 被更新过时，都会自动重新加载，不需要重新启动 Tomcat。

注意

`<Context>...</Context>` 的位置必须在 `<Host>...</Host>` 之间，不可任意更动位置。

第三步：执行 JSPBook 站台（见图 1-17）；



图 1-17 执行 JSPBook 站台

第四步：JSPBook 站台目录结构。

JSPBook 目录下包含：

- (1) 各章节的 HTML/JSP 程序；
- (2) dist 目录：存放在 JSPBook 站台压缩后的 JSPBook.war；
- (3) build.xml：Ant 文件；
- (4) WEB-INF 目录：包含 \classes、\lib、\tags 和 \src；
- (5) src 目录：存放范例的源程序，如：JavaBean、Filter、Servlet，等等；
- (6) Images 目录：存放范例程序的图片。

图 1-18 为 JSPBook 站台目录结构。

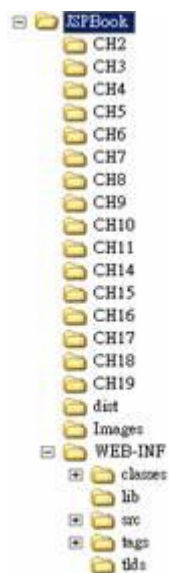


图 1-18 JSPBook

1-4 安装 Ant 1.6

修改 JSP 程序时，Tomcat 会自动将 JSP 重新转译为 Servlet，并且编译 Servlet。但是，倘若修改 Servlet、JavaBean 或 Filter 时，我们就必须自行先编译它们，然后再将它们重新部署至 WEB-INF\classes 下。

为了方便编译这些程序，笔者提供 JSPBook 站台的 build.xml 文件，因此，建议读者先安装 Ant 1.6，并且学习使用 Ant。

目前 Ant 的最新版本为 1.6，读者可以自行至 <http://ant.apache.org> 下载最新版本，如图 1-19 所示，或者直接使用本书 CD 光盘中的 Ant 1.6，软件名称为：apache-ant-1.6.0-bin.zip。

第一步：将 apache-ant-1.6.0-bin.zip 解压缩；

解压缩 apache-ant-1.6.0-bin.zip 之后，在 apache-ant-1.6.0-bin 目录下会有一目录 apache-ant-1.6.0，然后将 apache-ant-1.6.0 整个目录搬移至 C:\ 底下。

第二步：设定 Ant 1.6（见图 1-20）；

从【开始】→【设定】→【控制面板】→【系统】→【高级】→【环境变量】→【系统变量】，然后到【新建】。

```
ANT_HOME = C:\apache-ant-1.6.0
```

```
PATH = %ANT_HOME%\bin
```

第三步：测试 Ant 1.6；

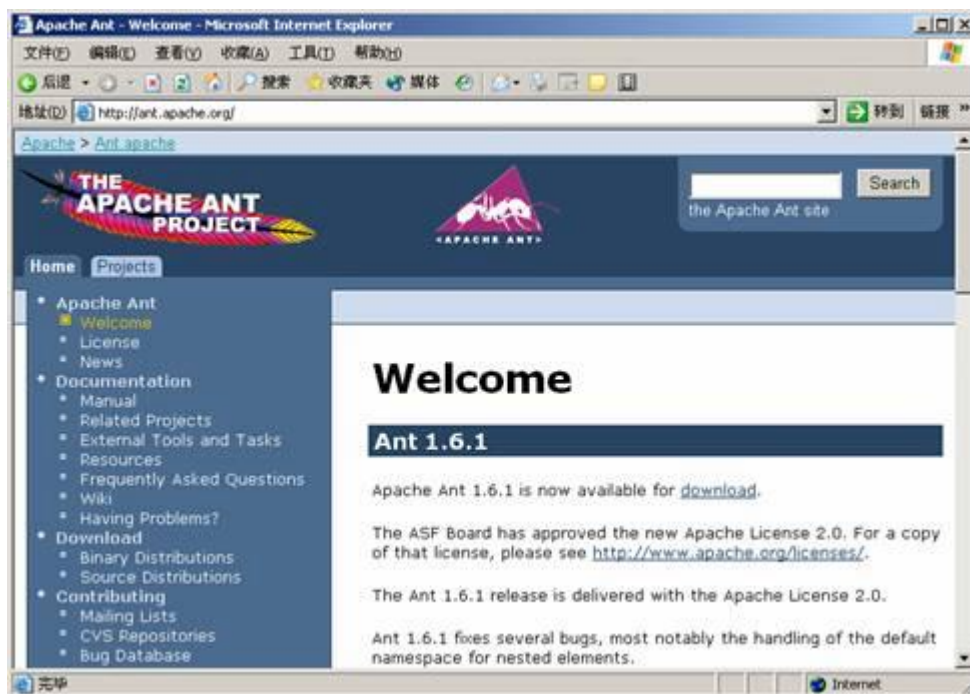


图 1-19 <http://ant.apache.org>



图 1-20 设定 Ant 1.6

打开命令提示符，输入 `ant -version`，假若执行成功，则会有如图 1-21 所示的结果。

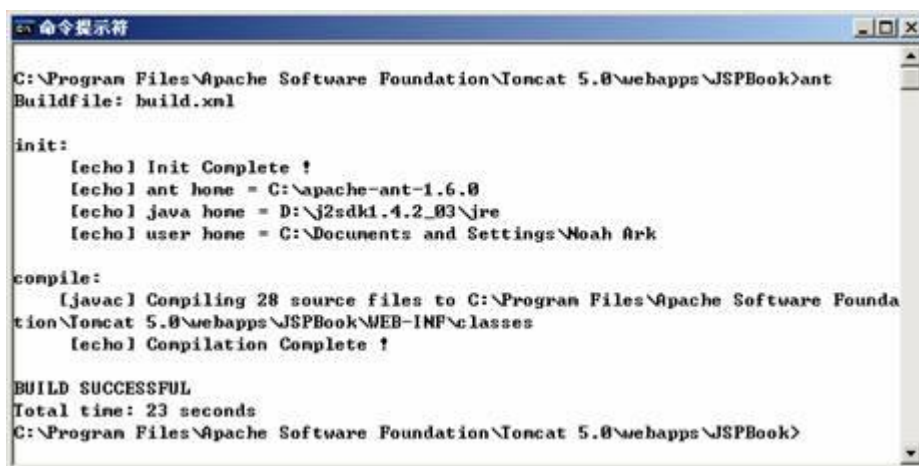


图 1-21 测试 Ant 1.6

第四步：使用 Ant 1.6 编译 JSPBook\WEB-INF\src 中的程序。

要编译修改过的 JSPBook\WEB-INF\src 中的程序时，首先打开命令提示符，移至 JSPBook 站台的所在目录，例如：C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook。

然后执行 ant，它会先自动找到 JSPBook\build.xml 文件，根据 build.xml 的设定，编译 C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook\WEB-INF\src 目录下所有的 Java 源文件，然后将产生的类文件存放至 C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook\WEB-INF\classes 目录下。图 1-22 为执行 ant 指令的结果。



```
C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook>ant
Buildfile: build.xml

init:
[echo] Init Complete !
[echo] ant home = C:\apache-ant-1.6.0
[echo] java home = D:\j2sdk1.4.2_03\jre
[echo] user home = C:\Documents and Settings\Noah Ark

compile:
[javac] Compiling 28 source files to C:\Program Files\Apache Software Founda
tion\Tomcat 5.0\webapps\JSPBook\WEB-INF\classes
[echo] Compilation Complete !

BUILD SUCCESSFUL
Total time: 23 seconds
C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook>
```

图 1-22 执行 ant 指令

第二章 Servlet 2.4 简介

2-1 Servlet 简介

自 1997 年 3 月 Sun Microsystems 公司所组成的 JavaSoft 部门将 Servlet API 定案以来，推出了 Servlet API 1.0。就当时功能来说，Servlet 所提供的功能包含了当时的 CGI (Common Gateway Interface) 与 Netscape Server API (NSAPI) 之类产品的功能。

发展至今，Servlet API 的最新版本为 2.4 版。它依旧是一个具有跨平台特性、100% Pure Java 的 Server-Side 程序，它相对于在 Client 端执行的 Applet。Servlet 不只限定于 HTTP 协议，开发人员可以利用 Servlet 自定义或延伸任何支持 Java 的 Server —— 包括 Web Server、Mail Server、Ftp Server、Application Server 或任何自定义的 Server。

Servlet 有以下优点：

● 可移植性(Portability)

Servlet 皆是利用 Java 语言来开发的，因此，延续 Java 在跨平台上的表现，不论 Server 的操作系统是 Windows、Solaris、Linux、HP-UX、FreeBSD、Compaq Tru 64、AIX 等等，都能够将我们所写好的 Servlet 程序放在这些操作系统上执行。借助 Servlet 的优势，就可以真正达到 Write Once, Serve Anywhere 的境界，这正是从事 Java 程序员最感到欣慰也是最骄傲的地方。

当程序员在开发 Applet 时，时常为了“可移植性”(portability)让程序员感到绑手绑脚的，例如：开发 Applet 时，为了配合 Client 端的平台（即浏览器版本的不同，plug-in 的 JDK 版本也不尽相同），达到满足真正“跨平台”的目的时，需要花费程序员大量时间来修改程序，为的就是能够让用户皆能够执行。但即使如此，往往也只能满足大部分用户，而其他少数用户，若要执行 Applet，仍须先安装合适的 JRE (Java Runtime Environment)。

但是 Servlet 就不同了，主要原因在于 Servlet 是在 Server 端上执行的，所以，程序员只要专心开发能在实际应用的平台环境下测试无误即可。除非你是从事做 Servlet Container 的公司，否则不须担心写出来的 Servlet 是否能在所有的 Java Server 平台上执行。

● 强大的功能

Servlet 能够完全发挥 Java API 的威力，包括网络和 URL 存取、多线程 (Multi-Thread)、影像

处理、RMI (Remote Method Invocation)、分布式服务器组件 (Enterprise Java Bean)、对象序列化 (Object Serialization) 等。若想写个网络目录查询程序，则可利用 JNDI API；想连接数据库，则可利用 JDBC，有这些强大功能的 API 做后盾，相信 Servlet 更能够发挥其优势。

● 性能

Servlet 在加载执行之后，其对象实体(instance)通常会一直停留在 Server 的内存中，若有请求(request)发生时，服务器再调用 Servlet 来服务，假若收到相同服务的请求时，Servlet 会利用不同的线程来处理，不像 CGI 程序必须产生许多进程 (process)来处理数据。在性能的表现上，大大超越以往撰写的 CGI 程序。最后补充一点，那就是 Servlet 在执行时，不是一直停留在内存中，服务器会自动将停留时间过长一直没有执行的 Servlet 从内存中移除，不过有时候也可以自行写程序来控制。至于停留时间的长短通常和选用的服务器有关。

● 安全性

Servlet 也有类型检查(Type Checking)的特性，并且利用 Java 的垃圾收集(Garbage Collection)与没有指针的设计，使得 Servlet 避免内存管理的问题。

由于在 Java 的异常处理(Exception-Handling)机制下，Servlet 能够安全地处理各种错误，不会因为发生程序上逻辑错误而导致整体服务器系统的毁灭。例如：某个 Servlet 发生除以零或其他不合法的运算时，它会抛出一个异常(Exception)让服务器处理，如：记录在记录文件中(log file)。

2-2 First Servlet Sample Code

HelloServlet.java

```
package tw.com.javaworld.CH2;

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

public class HelloServlet extends HttpServlet {

    //Initialize global variables
```

```
public void init(ServletConfig config) throws ServletException {

    super.init(config);

}

    //Process the HTTP Get request

    public void doGet(HttpServletRequest request,
                        HttpServletResponse response)

        throws ServletException, IOException {

        response.setContentType("text/html;charset=GB2312");

        PrintWriter out = response.getWriter();

        out.println("<html>");

        out.println("<head><title>CH2 - HelloServlet</title></head>");

        out.println("<body>");

        out.println(" Hello World <br>");

        out.println("大家好");

        out.println("</body>");

        out.println("</html>");

        out.close();

    }

//Get Servlet information

    public String getServletInfo() {

        return "tw.com.javaworld.CH2.HelloServlet Information";

    }

}
```

注意: `HelloServlet.java` 范例程序位于 `JSPBook\WEB-INF\src\tw\com\javaworld\CH2`, 其中 JSPBook 范例程序的安装方法, 请参见 1-3 节“安装 JSPBook 站台范例” 和 1-4 节“安装 Ant 1.6”。

一开始我们必须导入(`import`) `javax.servlet.*`、`javax.servlet.http.*`。

`javax.servlet.*` : 存放与 HTTP 协议无关的一般性 `Servlet` 类;

`javax.servlet.http.*` : 除了继承 `javax.servlet.*` 之外, 并且还增加与 HTTP 协议有关的功能。

所有 `Servlet` 都必须实现 `javax.servlet.Servlet` 接口(`Interface`), 但是通常我们都会从 `javax.servlet.GenericServlet` 或 `javax.servlet.http.HttpServlet` 择一来实现。若写的 `Servlet` 程序和 HTTP 协议无关, 那么必须继承 `GenericServlet` 类; 若有关, 就必须继承 `HttpServlet` 类。

`javax.servlet.*` 里的 `ServletRequest` 和 `ServletResponse` 接口提供存取一般的请求和响应; 而 `javax.servlet.http.*` 里的 `HttpServletRequest` 和 `HttpServletResponse` 接口, 则提供 HTTP 请求及响应的存取服务。

```
public void init(ServletConfig config) throws ServletException {  
  
    super.init(config);  
  
}
```

这个例子中, 一开始和 `Applet` 一样, 也有 `init()` 的方法。当 `Servlet` 被 `Container` 加载后, 接下来就会先执行 `init()` 的内容, 因此, 我们通常利用 `init()` 来执行一些初始化的工作。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)  
  
    throws ServletException, IOException {  
  
    response.setContentType("text/html");  
  
    PrintWriter out = response.getWriter();  
  
    out.println("<html>");  
  
    out.println("<head><title>CH2 - HelloServlet</title></head>");  
  
    out.println("<body>");  
  
    out.println("Hello World <br>");  
  
}
```

```
out.println("大家好");

out.println("</body>");

out.println("</html>");

out.close();

}
```

Servlet 可以利用 `HttpServletResponse` 类的 `setContentType()` 方法来设定内容类型，我们要显示为 HTML 网页类型，因此，内容类型设为 `"text/html"`，这是 HTML 网页的标准 MIME 类型值。之后，Servlet 用 `getWriter()` 方法取得 `PrintWriter` 类型的 `out` 对象，它与 `PrintStream` 类似，但是它能对 Java 的 Unicode 字符进行编码转换。最后，再利用 `out` 对象把 `"Hello World"` 的字符串显示在网页上。

```
public void destroy() {
    .....

    ..... Servlet 结束时，会自动调用执行的程序

    .....
}
```

若当 Container 结束 Servlet 时，会自动调用 `destroy()`，因此，我们通常利用 `destroy()` 来关闭资源或是写入文件，等等。

编译 `HelloServlet.java` 的方法：

(1) 将 `servlet-api.jar` 加入至 CLASSPATH 之中，直接使用 `javac` 来编译 `HelloServlet.java`。其中 `servlet-api.jar` 可以在 `{Tomcat_Install}\common\lib` 找到。

(2) 直接使用 Ant 方式编译 `HelloServlet.java`，请参见 1-4 节“安装 Ant 1.6”。

编译好 `HelloServlet.java` 之后，再来设定 `web.xml`，如下：

```
<servlet>

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

```
<servlet-class>tw.com.javaworld.CH2.HelloServlet</servlet-class>

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>
```

最后, HelloServlet.java 的执行结果如图 2-1 所示。



图 2-1 HelloServlet.java 的执行结果

顺利完成第一个 Servlet 程序后, 不知道读者有没有发现, 在 HelloServlet.java 主程序中, 其实大部分都是一些用来显示 HTML 的 `out.println("...")` 程序代码, 这就是 Servlet 用在开发 Web-based 系统时最麻烦的地方。假若 Servlet 要显示表格统计图时, 我想那时候程序员一定会疯掉, 因为你会发现, 其实你所有的时间都在 `out.println()`, 因此, Servlet 适合在简单的用户接口 (User Interface) 系统中。不过, 幸好有 JSP 技术来解决这项极为不方便的问题。

2-3 Servlet 的生命周期

当 Servlet 加载 Container 时, Container 可以在同一个 JVM 上执行所有 Servlet, 所以 Servlet 之间可以有效地共享数据, 但是 Servlet 本身的私有数据亦受 Java 语言机制保护。

Servlet 从产生到结束的流程如图 2-2 所示。

(1) 产生 Servlet, 加载到 Servlet Engine 中, 然后调用 `init()` 这个方法来进行初始化工作。

(2) 以多线程的方式处理来自 Client 的请求。

(3) 调用 `destroy()` 来销毁 Servlet，进行垃圾收集 (garbage collection)。

Servlet 生命周期的定义，包括如何加载、实例化、初始化、处理客户端请求以及如何被移除。这个生命周期由 `javax.servlet.Servlet` 接口的 `init()`、`service()` 和 `destroy()` 方法表达。

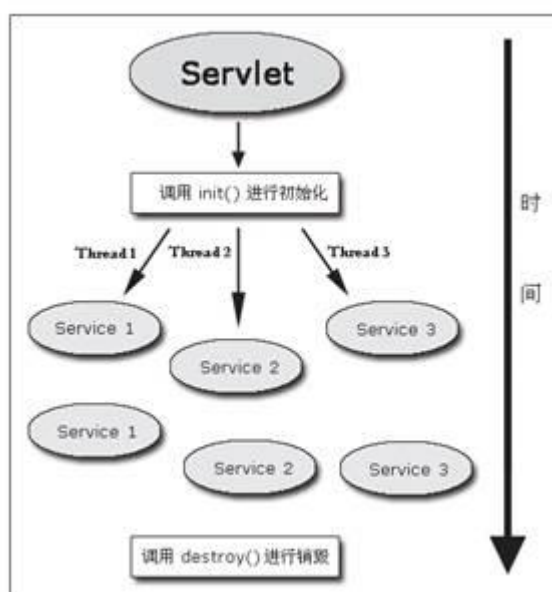


图 2-2 Servlet 从产生到结束的流程

1. 加载和实例化

当 Container 一开始启动,或是客户端发出请求服务时,Container 会负责加载和实例化一个 Servlet。

2. 初始化

Servlet 加载并实例化后,再来 Container 必须初始化 Servlet。初始化的过程主要是读取配置信息(例如 JDBC 连接)或其他须执行的任务。我们可以借助 `ServletConfig` 对象取得 Container 的配置信息,例如:

```
<servlet>

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

    <servlet-class>tw.com.javaworld.CH2.HelloServlet</servlet-class>
```

```
<init-param>

    <param-name>user</param-name>

    <param-value>browser</param-value>

</init-param>

</servlet>
```

其中 **user** 为初始化的参数名称；**browser** 为初始化的值。因此，可以在 **HelloServlet** 程序中使用 **ServletConfig** 对象的 **getInitParameter("user")** 方法来取得 **browser**。

3. 处理请求

Servlet 被初始化后，就可以开始处理请求。每一个请求由 **ServletRequest** 对象来接收请求；而 **ServletResponse** 对象来响应该请求。

4. 服务结束

当 **Container** 没有限定一个加载的 **Servlet** 能保存多长时间，因此，一个 **Servlet** 实例可能只在 **Container** 中存活几毫秒，或是其他更长的任意时间。一旦 **destroy()** 方法被调用时，**Container** 将移除该 **Servlet**，那么它必须释放所有使用中的任何资源，若 **Container** 需要再使用该 **Servlet** 时，它必须重新建立新的实例。

2-4 Servlet 范例程序

为了说明 **Servlet** 和网页是如何沟通的，笔者在此举一个 **Sayhi** 的范例程序。这个范例程序分为两部分：**Sayhi.html** 和 **Sayhi.java**。

在 **Sayhi.html** 中，用户可以填入姓名，然后按下【提交】后，将数据传到 **Sayhi.java** 做处理，而 **Sayhi.java** 负责将接收到的数据显示到网页上。

Sayhi.html

```
<html>

<head>

    <title>CH2 - Sayhi.html</title>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">

</head>

<body>

  <h2>Servlet 范例程序</h2>

  <form name="Sayhi" Method="Post" action="/JSPBook/CH2/Sayhi" >

    <p>请访问者输入姓名: <input type="text" name="Name" size="30"></p>

    <input type="submit" value="提交">

    <input type="reset" value="清除">

  </form>

</body>

</html>
```

Sayhi.html 的执行结果如图 2-3 所示。



图 2-3 Sayhi.html 的执行结果

Sayhi.java

```
package tw.com.javaworld.CH2;

import javax.servlet.*;

import javax.servlet.http.*;

import java.io.*;
```



```
public class Sayhi extends HttpServlet {

    //Initialize global variables

    public void init(ServletConfig config) throws ServletException {

        super.init(config);

    }

    //Process the HTTP Get request

    public void doPost(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

        response.setContentType("text/html;charset=GB2312");

        PrintWriter out = response.getWriter();

        request.setCharacterEncoding("GB2312");

        String Name = request.getParameter("Name");

        out.println("<html>");

        out.println("<head><title>CH2 - Sayhi</title></head>");

        out.println("<body>");

        out.println("Hi: "+Name);

        out.println("</body>");

        out.println("</html>");

        out.close();

    }

    //Get Servlet information

    public String getServletInfo() {

        return "tw.com.javaworld.CH2.Sayhi Information";

    }

}
```

```
}

public void destroy() {

}

}
```

从 Sayhi.java 的程序当中, 可以发现 Servlet 是利用 HttpServletRequest 类的 getParameter() 方法来取得由网页传来的数据。不过数据通过 HTTP 协议传输时会被转码, 因此在接收时, 必须再做转码的工作, 才能够正确地接收到数据。下面这段程序是做转码的动作:

```
request.setCharacterEncoding("GB2312");
```

编译 Sayhi.java 之后, 再来设定 web.xml:

```
<servlet>

    <servlet-name>Sayhi</servlet-name>

    <servlet-class>tw.com.javaworld.CH2.Sayhi</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>Sayhi</servlet-name>

    <url-pattern>/CH2/Sayhi</url-pattern>

</servlet-mapping>
```

执行 <http://localhost:8080/JSPBook/CH2/Sayhi>, 结果如图 2-4 所示。



图 2-4 Sayhi.html 按下【提交】后, 经过 Sayhi.java 处理后的结果

2-5 Servlet 2.4 的新功能

2003 年 11 月底, J2EE 1.4 规范正式发布, Servlet 也从原本的 2.3 版升级至 2.4 版。其中主要新增的功能有以下三点:

- (1) web.xml DTD 改用 XML Schema;
- (2) 新增 Filter 四种设定;
- (3) 新增 Request Listener、Event 和 Request Attribute Listener、Event。

2-5-1 web.xml 改用 XML Schema

Servlet 在 2.4 版之前, web.xml 都是使用 DTD(Document Type Definition)来定义 XML 文件内容结构的, 因此, Servlet 2.3 版 web.xml 一开始的声明如下:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app

    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    .....

</web-app>
```

到了 Servlet 2.4 版之后, web.xml 改为使用 XML Schema, 此时 web.xml 的声明如下:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"

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

```
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"

version="2.4">

.....

</web-app>
```

由 DTD 改为 Schema，主要加强两项功能：

- (1) 元素可不依照顺序设定；
- (2) 更强大的验证机制。

下面的范例，在 Servlet 2.3 版是不合规则的 web.xml 文件：

```
<web-app>

...

<servlet>

    <servlet-name>ServletA</servlet-name>

    <servlet-class>tw.com.javaworld.servlet.ServletA</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>ServletA</servlet-name>

    <url-pattern>/ServletA/*</url-pattern>

</servlet-mapping>

<servlet>

    <servlet-name>ServletB</servlet-name>
```

```
<servlet-class> tw.com.javaworld.servlet.ServletB</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>ServletB</servlet-name>

    <url-pattern>/ServletB /*</url-pattern>

</servlet-mapping>

...

</web-app>
```

因为<servlet-mapping>元素必须在<servlet>元素之后，因此，上述的范例要改为：

```
<web-app>

...

<servlet>

    <servlet-name>ServletA</servlet-name>

    <servlet-class>tw.com.javaworld.servlet.ServletA</servlet-class>

</servlet>

<servlet>

    <servlet-name>ServletB</servlet-name>

    <servlet-class> tw.com.javaworld.servlet.ServletB</servlet-class>

</servlet>
```

```
<servlet-mapping>

    <servlet-name>ServletA</servlet-name>

    <url-pattern>/ServletA/*</url-pattern>

</servlet-mapping>

<servlet-mapping>

    <servlet-name>ServletB</servlet-name>

    <url-pattern>/ServletB /*</url-pattern>

</servlet-mapping>

...

</web-app>
```

不过在 Servlet 2.4 版之后，原来的范例也算是一个合法的 web.xml 文件，不再须注意元素的顺序。

除此之外，Servlet 2.4 版 web.xml 的 Schema 更能提供强大的验证机制，例如：

(1) 可检查元素的值是否为合法的值。例如：<filter-mapping>的<dispatcher>元素，其值只能为 REQUEST、FORWARD、INCLUDE 和 ERROR，如下所示：

```
<filter-mapping>

    <filter-name>Hello</filter-name>

    <url-pattern>/CH11/*</url-pattern>

    <dispatcher>REQUEST</dispatcher>

    <dispatcher>FORWARD</dispatcher>

</filter-mapping>
```

若<dispatcher>元素的值不为上述四种时，此 web.xml 将会发生错误。

(2) 可检查如 Servlet、Filter 或 EJB-ref 等等元素的名称是否惟一。例如：

```
<servlet>

    <servlet-name>ServletA</servlet-name>

    <servlet-class>tw.com.javaworld.servlet.ServletA</servlet-class>

</servlet>


<servlet>

    <servlet-name>ServletA</servlet-name>

    <servlet-class>tw.com.javaworld.servlet.ServletB</servlet-class>

</servlet>
```

(3) 可检查元素值是否为合法文字字符或数字字符。例如：

```
<filter-mapping>

    <filter-name>Hello</filter-name>

    <url-pattern>/CH11/*</url-pattern>

</filter-mapping>
```

2-5-2 新增 Filter 四种设定

Servlet 2.3 版新增了 Filter 的功能，不过它只能由客户端发出请求来调用 Filter，但若使用 `RequestDispatcher.forward()` 或 `RequestDispatcher.include()` 的方法调用 Filter 时，Filter 却不会执行。因此，在 Servlet 2.4 版中，新增 Filter 的设定 `<dispatcher>` 来解决这个问题。有关 Filter 的部分在本书“第十一章：Filter 与 Listener”有更详细的介绍。

Servlet 2.4 版新增的 Filter 四种设定为：REQUEST、FORWARD、INCLUDE 和 ERROR。假若你有一个 `SimpleFilter`，它只允许由客户端发出请求或由 `RequestDispatcher.include()` 的方式来调用执行 `SimpleFilter`，此时 `SimpleFilter` 的设定如下：

```
<filter>

    <filter-name>SimpleFilter</filter-name>

    <filter-class>tw.com.javaworld.CH11.SimpleFilter</filter-class>

</filter>

<filter-mapping>

    <filter-name>SimpleFilter</filter-name>

    <url-pattern>/CH11/*</url-pattern>

    <dispatcher>REQUEST</dispatcher>

    <dispatcher>INCLUDE</dispatcher>

</filter-mapping>
```

2-5-3 新增 Request Listener、Event 和 Request Attribute Listener、Event

在 Servlet 2.3 版中，新增许多的 Listener 接口和 Event 类（见表 2-1）：

表 2-1

Listener 接口	Event 类
ServletContextListener	ServletContextEvent
ServletContextAttributeListener	ServletContextAttributeEvent
HttpSessionListener	HttpSessionEvent
HttpSessionActivationListener	
HttpSessionAttributeListener	

在 Servlet 2.4 版陆续又多新增 Request Listener、Event 和 Request Attribute Listener、Event（见表 2-2）：

表 2-2

Listener 接口	Event 类
ServletRequestListener	ServletRequestEvent
ServletRequestAttributeListener	ServletRequestAttributeEvent

这部分在“第十一章：Filter 与 Listener”中有更详细的介绍。

2-5-4 Servlet 2.4 的其他变更

Servlet 2.4 其他较显著的变更如：

(1) 取消 SingleThreadModel 接口。当 Servlet 实现 SingleThreadModel 接口时，它能确保同时间内，只能有一个 thread 执行此 Servlet。

(2) <welcome-file-list>可以为 Servlet。例如：

```
<servlet>

    <servlet-name>Index</servlet-name>

    <servlet-class>tw.com.javaworld.IndexServlet</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>Index</servlet-name>

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

</servlet-mapping>

<welcome-file-list>

    <welcome-file>Index</welcome-file>

</welcome-file-list>
```

(3) ServletRequest 接口新增一些方法，如：

```
public String getLocalName( );

public String getLocalAddr( );

public int getLocalPort( );

public int getRemotePort( );
```

第三章 JSP 2.0 简介

3-1 JavaServer Pages 技术

JavaServer Pages 技术是一个纯 Java 平台的技术,它主要用来产生动态网页内容,包括:HTML、DHTML、XHTML 和 XML。JavaServer Pages 技术能够让网页人员轻易建立起功能强大、有弹性的动态内容。

JavaServer Pages 技术有下列优点:

● Write Once, Run Anywhere 特性

作为 Java 平台的一部分,JavaServer Pages 技术拥有 Java 语言“一次编写,各处执行”的特点。随着越来越多的供货商将 JavaServer Pages 技术添加到他们的产品中,您可以针对自己公司的需求,做出审慎评估后,选择符合公司成本及规模的服务器,假若未来的需求有所变更时,更换服务器平台并不影响之前所投下的成本、人力所开发的应用程序。

● 搭配可重复使用的组件

JavaServer Pages 技术可依赖于重复使用跨平台的组件(如:JavaBean 或 Enterprise JavaBean 组件)来执行更复杂的运算、数据处理。开发人员能够共享开发完成的组件,或者能够加强这些组件的功能,让更多用户或是客户团体使用。基于善加利用组件的方法,可以加快整体开发过程,也大大降低公司的开发成本和人力。

● 采用标签化页面开发

Web 网页开发人员不一定是熟悉 Java 语言的程序员。因此,JSP 技术能够将许多功能封装起来,成为一个自定义的标签,这些功能是完全根据 XML 的标准来制订的,即 JSP 技术中的标签库(Tag Library)。因此,Web 页面开发人员可以运用自定义好的标签来达成工作需求,而无须再写复杂的 Java 语法,让 Web 页面开发人员亦能快速开发出一动态内容网页。

今后,第三方开发人员和其他人员可以为常用功能建立自己的标签库,让 Web 网页开发人员能够使用熟悉的开发工具,如同 HTML 一样的标签语法来执行特定功能的工作。本书将在“第十五章:JSP Tag Library”和“第十六章:Simple Tag 与 Tag File”中详细地为各位介绍如何制作标签。

● N-tier 企业应用架构的支持

有鉴于网际网络的发展,为因应未来服务越来越繁杂的要求,且不再受地域的限制,因此,必须放弃以往 Client-Server 的 Two-tier 架构,进而转向更具威力、弹性的分散性对象系统。由于

JavaServer Page 技术是 Java 2 Platform Enterprise Edition (J2EE) (相关信息请参阅 www.javasoft.com/products/j2ee) 集成中的一部分, 它主要是负责前端显示经过复杂运算后之结果内容, 而分散性的对象系统则是主要依赖 EJB (Enterprise JavaBean) 和 JNDI (Java Naming and Directory Interface) 构建[1]而成。

3-2 What is JSP

JSP(JavaServer Pages)是由 Sun 公司倡导、许多别的公司参与一起建立的一种新动态网页技术标准, 类似其他技术标准, 如 ASP、PHP 或是 ColdFusion, 等等。

在传统的网页 HTML 文件(*.htm, *.html)中加入 Java 程序片段(Scriptlet)和 JSP 标签, 构成了 JSP 网页(*.jsp)。Servlet/JSP Container 收到客户端发出的请求时, 首先执行其中的程序片段, 然后将执行结果以 HTML 格式响应给客户端。其中程序片段可以是: 操作数据库、重新定向网页以及发送 E-Mail 等等, 这些都是建立动态网站所需要的功能。所有程序操作都在服务器端执行, 网络上传送给客户端的仅是得到的结果, 与客户端的浏览器无关, 因此, JSP 称为 Server-Side Language。

3-3 JSP 与 Servlet 的比较

Sun 公司首先发展出 Servlet, 其功能非常强大, 且体系设计也很完善, 但是它输出 HTML 语法时, 必须使用 `out.println()` 一句一句地输出, 例如下面一段简单的程序:

```
out.println("<html>");  
  
out.println("<head><title>demo1</title></head>");  
  
out.println(" Hello World <br>");  
  
out.println("<body>");  
  
out.println("大家好");  
  
out.println("</body>");  
  
out.println("</html>");
```

由于这是一段简单的 Hello World 程序，还看不出来其复杂性，但是当整个网页内容非常复杂时，那么你的 Servlet 程序可能大部分都是用 `out.println()` 输出 HTML 的标签了！

后来 Sun 公司推出类似于 ASP 的嵌入型 Scripting Language，并且给它一个新的名称：JavaServer Pages，简称为 JSP。于是上面那段程序改为：

```
<html>

<head><title>www.javaworld.com.tw - 台湾 Java 论坛</title></head>

<body>

<%

    out.println(" Hello World <br>");

    out.println("大家好");

%>

</body>

</html>
```

这样就简化了 Web 网页程序员的负担，不用为了网页内容编排的更动，又需要由程序员来做修改。

3-4 JSP 的执行过程

在介绍 JSP 语法之前，先向读者说明一下 JSP 的执行过程（见图 3-1）。

- (1) 客户端发出 Request（请求）；
- (2) JSP Container 将 JSP 转译成 Servlet 的源代码；
- (3) 将产生的 Servlet 的源代码经过编译后，并加载到内存执行；
- (4) 把结果 Response（响应）至客户端。

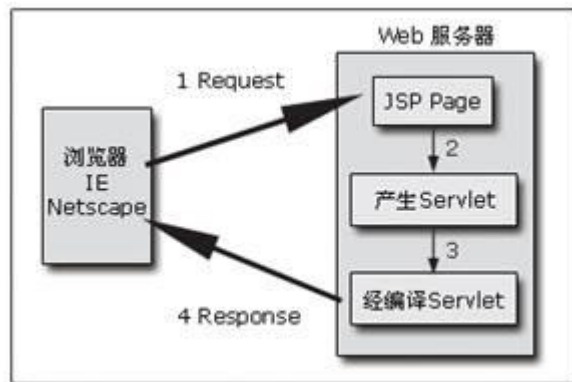


图 3-1 JSP 的执行过程

一般人都会以为 JSP 的执行性能会和 Servlet 相差很多，其实执行性能上的差别只在**第一次的执行**。因为 JSP 在执行第一次后，会被编译成 Servlet 的类文件 [\[玉玉 2\]](#)，即为 XXX.class，当再重复调用执行时，就直接执行第一次所产生的 Servlet，而不用再重新把 JSP 编译成 Servlet。因此，除了第一次的编译会花较久的时间之外，之后 JSP 和 Servlet 的执行速度就几乎相同了。

在执行 JSP 网页时，通常可分为两个时期：转译时期(Translation Time)和请求时期(Request Time)（见图 3-2）。

转译时期：JSP 网页转译成 Servlet 类。

请求时期：Servlet 类执行后，响应结果至客户端。

补充

转译期间主要做了两件事情：将 JSP 网页转译为 Servlet 源代码(.java)，此段称为转译时期(Translation time)；将 Servlet 源代码(.java)编译成 Servlet 类(.class)，此段称为编译时期(Compilation time)。

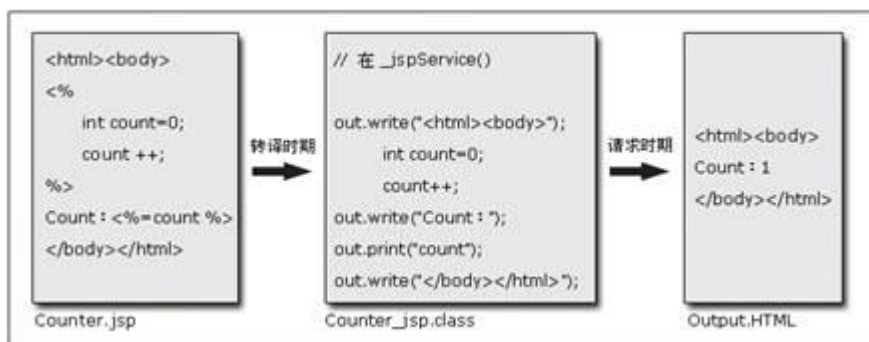


图 3-2 转译时期与请求时期程序图

当 JSP 网页在执行时, JSP Container 会做检查的工作, 若发现 JSP 网页有更新修改时, JSP Container 才会再次编译 JSP 成 Servlet; JSP 没有更新时, 就直接执行前面所产生的 Servlet。

笔者在这里以 Tomcat 为例, 看看 Tomcat 如何将 JSP 转译成 Servlet。首先笔者写一个简单的 JSP 网页 —— HelloJSP.jsp:

HelloJSP.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

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

<html>

<head>

    <title>CH3 - HelloJSP.jsp</title>

</head>

<body>

<h2>JSP 将会被转译为 Servlet</h2>

<%!

    int k = 0;

%>

<c:out value="Hi" />

<%

    String name = "browser";

    out.println("大家好 !!");

%>

<%= name %>

</body>

</html>
```

JSP2.0 技术手册

当执行 HelloJSP.jsp 时，Tomcat 会将它先转译为 Servlet。这个 Servlet 程序是放在 {Tomcat_Install}\Apache Software Foundation\Tomcat 5.0\work\Catalina\localhost\JSPBook\org\apache\jsp\CH3 目录下的 HelloJSP_jsp.java 和 HelloJSP_jsp.class。其中 HelloJSP_jsp.java 就是 HelloJSP.jsp 所转译的 Servlet 源代码，它的程序如下：

HelloJSP_jsp.java

```
package org.apache.jsp.CH3;

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

public final class HelloJSP_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    int k = 0;

    private static java.util.Vector _jspx_dependants;

    private org.apache.jasper.runtime.TagHandlerPool _jspx_tagPool_c_out_value;

    public java.util.List getDependants() {
        return _jspx_dependants;
    }

    public void _jspInit() {
        _jspx_tagPool_c_out_value =
            org.apache.jasper.runtime.TagHandlerPool.getTagHandlerPool(
                getServletConfig());
    }

    public void _jspDestroy() {
        _jspx_tagPool_c_out_value.release();
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
        response) throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
```

```
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
JspWriter _jspx_out = null;

try {
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=GB2312");
    pageContext = _jspxFactory.getPageContext(this, request, response,
        null, true, 8192, true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("\r\n");
    out.write("\r\n\r\n");
    out.write("<html>\r\n");
    out.write("<head>\r\n  ");
    out.write("<title>CH3 - HelloJSP.jsp");
    out.write("</title>\r\n");
    out.write("</head>\r\n");
    out.write("<body>\r\n\r\n");
    out.write("<h2>JSP 将会被转译为 Servlet");
    out.write("</h2>\r\n\r\n");
    out.write("\r\n");
    if (_jspx_meth_c_out_0(pageContext))
        return;
    out.write("\r\n");

    String name = "browser";

    out.println("大家好 !!");

    out.write("\r\n");
    out.print( name );
    out.write("\r\n\r\n");
    out.write("</body>\r\n");
    out.write("</html>");
} catch (Throwable t) {
```



```
    if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            out.clearBuffer();
        if (pageContext != null) pageContext.handlePageException(t);
    }
} finally {
    if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
}
}

private boolean _jspx_meth_c_out_0(PageContext pageContext)
    throws Throwable {
    JspWriter out = pageContext.getOut();
    // c:out
    org.apache.taglibs.standard.tag.rt.core.OutTag _jspx_th_c_out_0 =
        (org.apache.taglibs.standard.tag.rt.core.OutTag) _jspx_tagPool_c_out_value.
        get( org.apache.taglibs.standard.tag.rt.core.OutTag.class);
    _jspx_th_c_out_0.setPageContext(pageContext);
    _jspx_th_c_out_0.setParent(null);
    _jspx_th_c_out_0.setValue(new String("Hi"));
    int _jspx_eval_c_out_0 = _jspx_th_c_out_0.doStartTag();
    if (_jspx_th_c_out_0.doEndTag() == javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
        return true;
    _jspx_tagPool_c_out_value.reuse(_jspx_th_c_out_0);
    return false;
}
}
```

当 JSP 被转译成 Servlet 时，内容主要包含三部分：

```
public void _jspInit() {

    ... 略

}

public void _jspDestroy() {

    ... 略

}
```

```
}

public void _jspService(HttpServletRequest request, HttpServletResponse
                        response) throws java.io.IOException, ServletException {

    ... 略

}
```

`_jspInit()`：当 JSP 网页一开始执行时，最先执行此方法。因此，我们通常会把初始化的工作写在此方法中。

`_jspDestroy()`：JSP 网页最后执行的方法。

`_jspService()`：JSP 网页最主要的程序都是在此方法中。

接下来笔者将 `HelloJSP.jsp` 和 `HelloJSP_jsp.java` 做一个简单的对照：

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```
response.setContentType("text/html; charset=GB2312");
```

```
<%! int k = 0; %>
```

```
int k = 0; // 此为全局变量
```

```
<html>
```

```
<head>
```

```
<title>CH3 - HelloJSP.jsp</title>

</head>

<body>

<h2>JSP 将会被转译为 Servlet</h2>
```

```
out.write("\r\n");

out.write("\r\n\r\n");

out.write("<html>\r\n");

out.write("<head>\r\n  ");

out.write("<title>CH3 - HelloJSP.jsp");

out.write("</title>\r\n");

out.write("</head>\r\n");

out.write("<body>\r\n\r\n");

out.write("<h2>JSP 将会被转译为 Servlet");

out.write("</h2>\r\n\r\n");

out.write("\r\n");
```

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

<c:out value="Hi" />
```

```
if (_jspx_meth_c_out_0(pageContext))
```

```
    return;
... 略
private boolean _jspx_meth_c_out_0(PageContext pageContext)
    throws Throwable {
```

```
JspWriter out = pageContext.getOut();
// c:out
org.apache.taglibs.standard.tag.rt.core.OutTag _jspx_th_c_out_0 =
(org.apache.taglibs.standard.tag.rt.core.OutTag) _jspx_tagPool_c_out_value.
get(org.apache.taglibs.standard.tag.rt.core.OutTag.class);
_jjspx_th_c_out_0.setPageContext(pageContext);
_jjspx_th_c_out_0.setParent(null);
_jjspx_th_c_out_0.setValue(new String("Hi"));
int _jspx_eval_c_out_0 = _jspx_th_c_out_0.doStartTag();
if (_jspx_th_c_out_0.doEndTag() == javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
    return true;
_jjspx_tagPool_c_out_value.reuse(_jspx_th_c_out_0);
return false;
}
```

```
<%
    String name = "browser";

    out.println("大家好 !!");
%>

<%= name %>
```

```
String name = "browser";

out.println("大家好 !!");

out.write("\r\n");

out.print( name );
```

3-5 JSP 与 ASP 和 ASP+的比较

JSP 与 ASP 的比较

一般说来, Sun 公司的 JavaServer Pages (JSP) 和 Microsoft 的 Active Server Pages (ASP)

在技术方面有许多相似之处。两者都为动态网页的技术，并且双方都能够替代 CGI 技术，使网站的开发时程能够大大缩短，在性能上也有较高的表现，更重要的一点是，两者都能够为程序员提供组件设计的功能，通过组件设计，将网页中逻辑处理部分交由组件负责处理(ASP 使用 COM 组件、JSP 则有 JavaBean 组件)，而和网页上的排版、美工分离。

尽管 JavaServer Pages 技术和 Active Server Pages (ASP) 在许多方面都很相似，但仍然存在很多不同之处，其中本质上的区别在于：两者是来源于不同的技术规范组织。以下就来比较两大技术有哪些不同点，而又为各自带来哪些优势。

平台和服务器的弹性

ASP (Active Server Pages)技术主要在微软(Microsoft)公司的 Windows 平台上运行，其中包括 Windows 2000、Windows XP 和 Windows 2003，并且搭配其 WEB 服务器 IIS (Internet Information Services)。但是，在其他的平台运行时，不是性能低落，就是根本不支持，因此，当在开发网站系统时，选择 NT+IIS+ASP 的体系结构时，未来当系统无法负荷时，也只能继续选择 Windows 平台的服务器，无法改写在性能表现相当优异的 UNIX 平台上。

JSP (JavaServer Pages)技术主要运行在操作系统上的一个 Java Virtual Machine (JVM)虚拟机器上，因此，它能够跨越所有的平台，例如：NT、Windows 2000、Solaris、Linux、OS/390、AIX、HP-UX，等等，除了能在各式各样的操作系统上执行，并且能搭配现有的 WEB 服务器：Apache、IIS、Netscape Enterprise Server，等等，将静态的 HTML 网页交由执行速度较快的 Web Server 处理，而动态产生网页的部分，就交由 JSP Container 来执行。由上述可知，JSP (JavaServer Pages)技术在跨平台的表现比 ASP 来得更有弹性。

WEB 网页程序员未来在开发电子商务平台时，就不需要再考虑客户厂商的操作系统平台，可更专心于系统功能的开发。相应地，厂商在使用 JavaServer Pages 技术开发的系统平台时，不再需要担心未来在扩充软、硬件时，是否产生不兼容的问题。光这一点，就能为企业省下一大笔的费用，这是 JSP 的主要优点。

语法结构

ASP 语法结构上，是以 "<% "和"%>"作为标记符号，而 JSP 也是使用相同标记符号作为程序的区

段范围的。但不同的是，标记符号之间所使用的语言：ASP 为 JavaScript 或 VBScript；而 JSP 为 Java。Java 是有严格规划、强大且易扩充的语言，远优于 VBScript 语言。

Java 使程序员的工作在其他方面也变得一样容易、简单。例如：当 ASP 应用程序在 Windows NT 系统可能会造成系统 Crash (当机)时，由于 JSP 是在 JVM 上执行程序，且提供强大的异常事件处理机制，因此，不会因为程序撰写的疏忽，而导致服务器操作系统的损毁。

并且 Java 语言提供防止直接存取内存的功能，存取内存产生的错误，通常也正是造成服务器损毁的最主要原因之一。最后，最重要的原因，Java 语言是一个有严谨规范、有系统组织的语言，对一个专业的 Java 程序员来说，也真正达到 Learn Once, Write Anywhere(学一次，皆可开发)的境界。

开放的开发环境

自从 1995 年，Sun 公司已经开放技术与国际 Java 组织合作开发和修改 Java 技术与规范。针对 JSP 的新技术，Sun 公司授权工具供货商（如 Macromedia）、同盟公司（如 Apache、Netscape）、协力厂商及其他公司。最近，Sun 公司将最新版本的 Servlet 2.4 和 JSP 2.0 的源代码发放给 Apache，以求 JSP 与 Apache 紧密地相互发展。Apache、Sun 和许多其他的公司及个人公开成立一个咨询机构，以便任何公司和个人都能免费取得信息。（详见：<http://jakarta.apache.org>）

JSP 应用程序接口（API）毫无疑问已经取得成功，并随着 Java 组织不断扩大其应用的范围，目前全力发展 Java 技术的厂商不胜枚举，例如：最近 IBM 公司强力推广的 WebSphere 家族，正是完全支持 J2EE 标准而开发。数据库厂商 Oracle 也发展自己的 Application Server 来和自己公司本身数据库产品 Oracle 9i 做一紧密的结合。那也更不用提 Amazon 系统的供货商 BEA 公司，它的产品 WebLogic 也是完全支持 JavaServer Pages 技术和 J2EE 规范的。

相反，ASP 技术仅依靠微软本身的推动，其发展建立在独占、封闭的基础之上，并且微软本身的技术又只允许在微软相关平台的服务器上执行，因此，在标准方面显得有点力不从心。

语法的延展性

ASP 和 JSP 都使用标签与 Scripting Language 来制作动态 WEB 网页，JavaServer Pages 2.0 新规范中，能够让程序员自由扩展 JSP 标签来应用。JSP 开发者能自定义标签库 (Tag Library)，所

以网页制作者能充分利用与 XML 兼容的标签技术强大的功能，大大减低对 Java 语法的依赖，并且也可以利用 XML 强大的功能，做到数据、文件格式的标准化。相关标签库请参考“第十五章：JSP Tag Library”，其中有更加完整的说明。

执行性能表现

ASP 和 JSP 在执行性能的表现上，有一段显著的差距，JSP 除了在一开始加载的时间会比较久外，之后的表现就远远比 ASP 的表现来得好。原因在于：JSP 在一开始接受到请求时，会产生一份 Servlet 实体(instance)，它会先被暂存在内存中，我们称之为持续(Persistence)，当再有相同请求时，这实体会产生一个线程(thread)来服务它。如果过了一段时间都不再用到此实体时，Container 会自动将其释放，至于时间的长短，通常都是可以在 Container 上自行设定的。

而 ASP 在每次接收到请求时，都必须重新编译，因此，JSP 的执行比每次都要编译执行的 ASP 要快，尤其是程序中存在循环操作时，JSP 的速度要快上 1 到 2 倍。不过，ASP 在这部分的缺陷，将随 ASP+的出现有所改观，在新版的 ASP+技术中，性能表现上有很大的突破。

JSP 与 ASP+的比较

1. [\[玉玉3\]](#)面向对象性

C#为一种面向对象语言，从很多方面来看，C#将给 ASP+带来类似于 Java 的功能，并且它和 Windows 环境紧密结合，因此，具备更快的性能。笔者认为，C#是微软在市场上击败 Java 的主要工具。

2. 数据库连接

ASP 最大的优点是它使用 ADO 对象，因此，ASP Web 数据库应用开发特别简单。ASP+发展了更多的功能，因为有了 ADO+，ADO+带来了更强大更快速的功能。JSP 和 JDBC 目前在易用性和性能上和 ASP/ADO 相比已有些落后，当新版本 ASP+/ADO+出现后这样的差别会更明显，这部分希望 Sun 尽快追赶 ASP+/ADO+的组合。

3. 大型网站应用

ASP+将对大型网站有更好的支持。事实上，微软在这方面付出了巨大的努力，ASP+可以让你考虑到多服务器(multiple servers)的场合，当你需要更强大的功能时，仅仅只需要增加一台服务器。ASP+现在可以在大型项目方面与 JSP 一样具有等同的能力，而且 ASP+还有价格方面的优势，因为所有的组件将是服务器操作系统的一部分。

结论:

除了上述 ASP、ASP+和 JSP 之外, 笔者再提供一篇在网络上 ASP 和 JSP 比较的文章:

<http://www.indiawebdevelopers.com/technology/Java/jsp.asp>, 希望能带给读者更客观的评论。

除了 ASP 之外, PHP 和 ColdFusion 皆为近年来常用来开发 WEB 动态网页内容的工具, 各开发工具皆有其优、缺点。ASP 和 PHP 最大的好处就是开发中、小型网站非常快速, 市面上的书籍也较多, 学习起来能较快上手。尤其因为 PHP 的环境大都为 UNIX 的环境, 因此, 在规划、构建时, 所花需的成本为最低, 但 PHP 并未将 Presentation Layer 和 Business Layer 做一个适当的处理, 因此, 往往一个系统越来越庞大、越来越复杂时, 维护起来就会越来越吃力, 并且本身并没有一个强而有力的技术在支持它, 当开发系统要求为分布式的体系结构时, 那么 PHP 可能就英雄无用武之地了。

3-6 JSP 2.0 新功能

J2EE 1.4 正式发布之后, Servlet 和 JSP 同时也做了一些变动, Servlet 从 2.3 更新至 2.4; 而 JSP 从 1.2 更新至 2.0。两者平心而论, JSP 的变动较 Servlet 来得多, 其中 JSP 2.0 较 JSP 1.2 新增的功能如下:

- (1) Expression Language;
- (2) 新增 Simple Tag 和 Tag File;
- (3) web.xml 新增<jsp-config>元素。

3-6-1 Expression Language

JSP 2.0 之后, 正式将 EL 纳入 JSP 的标准语法。EL 主要的功用在于简化 JSP 的语法, 方便 Web 开发人员的使用。例如:

使用 JSP 传统语法:

```
<%  
  
String str_count = request.getParameter("count");  
  
int count = Integer.parseInt(str_count);
```



```
count = count + 5;

out.println("count: " + count);

%>
```

使用 EL 语法:

```
count: ${param.count + 5}
```

对于 EL 的部分, 本书的“第六章: Expression Language”有详尽的介绍。

3-6-2 新增 Simple Tag 和 Tag File

JSP 2.0 提供一些较为简单的方法, 让开发人员来撰写自定义标签。JSP 2.0 提供两种新的机制, 分别为 Simple Tag 和 Tag File。

Simple Tag Handler 和其他 Tag Handler(如: Body Tag Handler、Tag Handler 和 Iteration Tag Handler) 不同之处在于: Simple Tag Handler 并无 doStartTag() 和 doEndTag(), 它只有 doTag(), 因此, 实现标签能比以往更为方便。

Tag File 就更为简单, 你可以把它当做直接使用 JSP 的语法来制作标签。例如:

Hello.tag

```
<%
    out.println("Hello from tag file.");
%>
```

我们先制作一个名为 Hello.tag 的 Tag File, 然后将它放置在 WEB-INF/tags/ 目录下。在 JSP 网页使用 Hello.tag 的方法如下:

```
<%@ taglib prefix="myTag" tagdir="/WEB-INF/tags" %>

<myTag:Hello />
```

最后执行的结果如下：

```
Hello from tag file.
```

有关 Simple Tag Handler 和 Tag File 的部分，在“第十六章：Simple Tag 与 Tag File”有更详细的说明。

3-6-3 web.xml 新增<jsp-config>元素

<jsp-config> 元素主要用来设定 JSP 相关配置，<jsp-config> 包括 <taglib> 和 <jsp-property-group> 两个子元素。其中 <taglib> 元素在 JSP 1.2 时就已经存在；而 <jsp-property-group> 是 JSP 2.0 新增的元素。

<jsp-property-group> 元素主要有八个子元素，它们分别为：

<description>：设定的说明；

<display-name>：设定名称；

<url-pattern>：设定值所影响的范围，如：/CH2 或 /*.jsp；

<el-ignored>：若为 true，表示不支持 EL 语法；

<scripting-invalid>：若为 true，表示不支持<% scripting %>语法；

<page-encoding>：设定 JSP 网页的编码；

<include-prelude>：设置 JSP 网页的抬头，扩展名为.jspf；

<include-coda>：设置 JSP 网页的结尾，扩展名为.jspf。

图 3-3 所示的所谓 JSP 网页的抬头为网页最上方的 This banner included with <include-prelude>；结尾为网页最下方的 This banner included with <include-coda>。

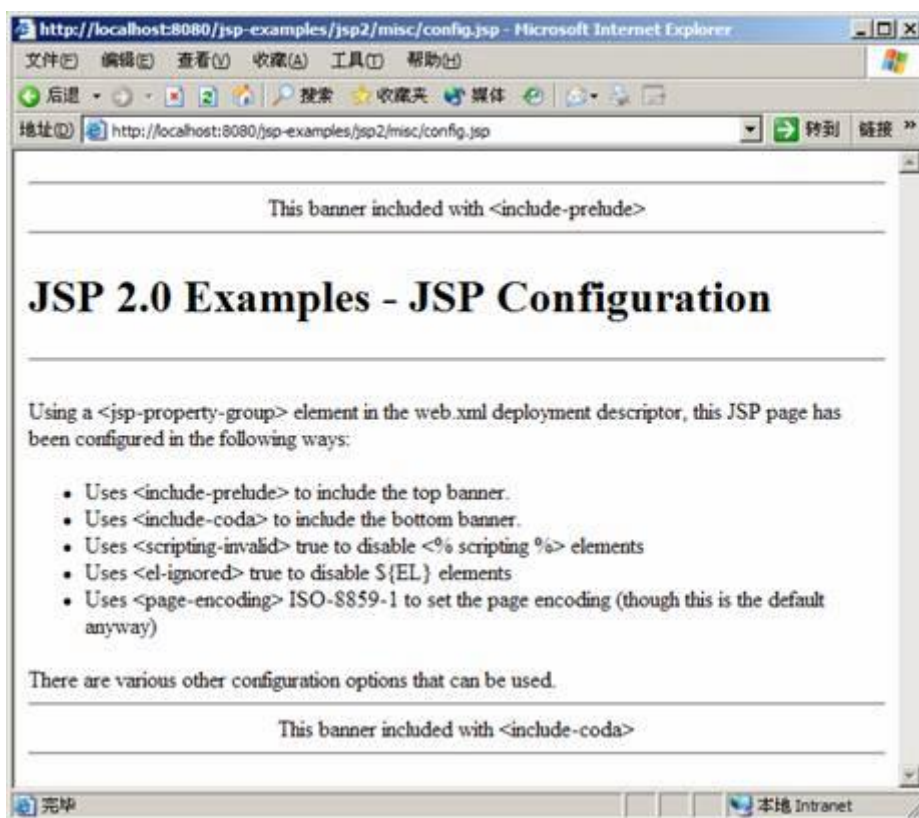


图 3-3 Tomcat 上的<include-include>范例程序

其中抬头的源程序为：

prelude.jspf

```
<hr>

<center>

This banner included with &lt;include-include>;

</center>

<hr>
```

结尾的源程序为：

coda.jspf

```
<hr>
```

```
<center>

This banner included with &lt;include-coda>;

</center>

<hr>
```

下面是一个简单的<jsp-config>元素完整配置:

```
<jsp-config>

  <taglib>

    <taglib-uri>TagLib</taglib-uri>

    <taglib-location>/WEB-INF/tlds/MyTagLib.tld</taglib-location>

  </taglib>

  <jsp-property-group>

    <description>

      Special property group for JSP Configuration JSP example.

    </description>

    <display-name>JSPConfiguration</display-name>

    <url-pattern>/jsp/* </url-pattern>

    <el-ignored>true</el-ignored>

    <page-encoding>GB2312</page-encoding>

    <scripting-invalid>true</scripting-invalid>

    <include-prelude>/include/prelude.jspf</include-prelude>

    <include-coda>/include/coda.jspf</include-coda>

  </jsp-property-group>

</jsp-config>
```

第四章 JSP 语法

4-1 Elements 和 Template Data

JSP 网页主要分为 Elements 与 Template Data 两部分。

Template Data: JSP Container 不处理的部分，例如：HTML 的内容，会直接送到 Client 端执行。

Elements: 必须经由 JSP Container 处理的部分，而大部分 Elements 都以 XML 作为语法基础，并且大小写必须要一致。

Elements 有两种表达式，第一种为起始标签(包含 Element 名称、属性)，中间为一些内容，最后为结尾标签。如下所示：

```
<mytag attr1="attribute value" ...>
    body
</mytag>
```

另一种是标签中只有 Element 的名称、属性，称为 Empty Elements。如下所示：

```
<mytag attr1="attribute value" .../>
```

Elements 有四种类型：Directive Elements、Scripting Elements、Action Elements 和 EL Elements，接下来的章节会针对前三种类型的 Elements 加以说明。至于 EL Elements 是 JSP 2.0 新增的功能，笔者将在“第六章：Expression Language”中详细介绍它。

4-2 批注 (Comments)

一般批注可分为两种：一种为在客户端显示的批注；另外一种就是客户端看不到，只给开发程序员专用的批注。

● 客户端可以看到的批注：

```
<!-- comment [ <%= expression %> ] -->
```

例如:

```
<!-- 现在时间为: <%= (new java.util.Date()).toLocaleString() %> -->
```

在客户端的 HTML 源文件中显示为:

```
<!--现在时间为: January 1, 2004 -->
```

这种批注的方式和 HTML 中很像, 它可以使用“查看源代码”来看到这些程序代码, 但是惟一有些不同的是, 你可以在批注中加上动态的表达式(如上例所示)。

● 开发程序员专用的批注:

```
<%-- comment --%>
```

或者

```
<% /** this is a comment */ %>
```

接下来看下面这个范例:

```
<%@ page language="java" %>

<html>

<head><title>A Comment Test</title></head>

<body>

<h2>A Test of Comments</h2>

<%-- 这个批注不会显示在客户端 --%>

</body>

</html>
```

从用户的浏览器中, 看到的源代码如下:

```
<html>

<head><title>A Comment Test</title></head>

<body>
```

```
<h2>A Test of Comments</h2>

</body>

</html>
```

之前加上去的批注在客户端的浏览器上看不出来，并且用此批注的方式，在 JSP 编译时会被忽略掉。这对隐藏或批注 JSP 程序是实用的方法，通常程序员也会利用它来调试(Debug)程序。

JSP Container 不会对 `<!--` 和 `-->` 之间的语句进行编译，它不会显示在客户端的浏览器上，也无法从源文件中看到。接下来介绍 Quoting 和 Escape 的规则。

4-3 Quoting 和 Escape 规则

Quoting 主要是为了避免与语法产生混淆所使用的转换功能，和 HTML 的标签语法类似，JSP 是以 `<%` 标签作为程序的起始、`%>` 标签作为程序的结束，所以当你在 JSP 程序中要加上 `<%` 或是 `%>` 这些符号时，应该这么做：

Quoting.jsp

```
<%@ page contentType="text/html; charset=GB2312 " %>

<html>

<head>

    <title>CH4 - Quoting.jsp</title>

</head>

<body>

    <h2>Quoting 范例程序</h2>
```

```
<%  
  
    out.println("JSP 以%>作为结束符号");  
  
%>  
  
</body>  
  
</html>
```

程序执行时，JSP 在执行到 %>时，JSP Container 就直接告诉你程序有错误，如图 4-1 所示：

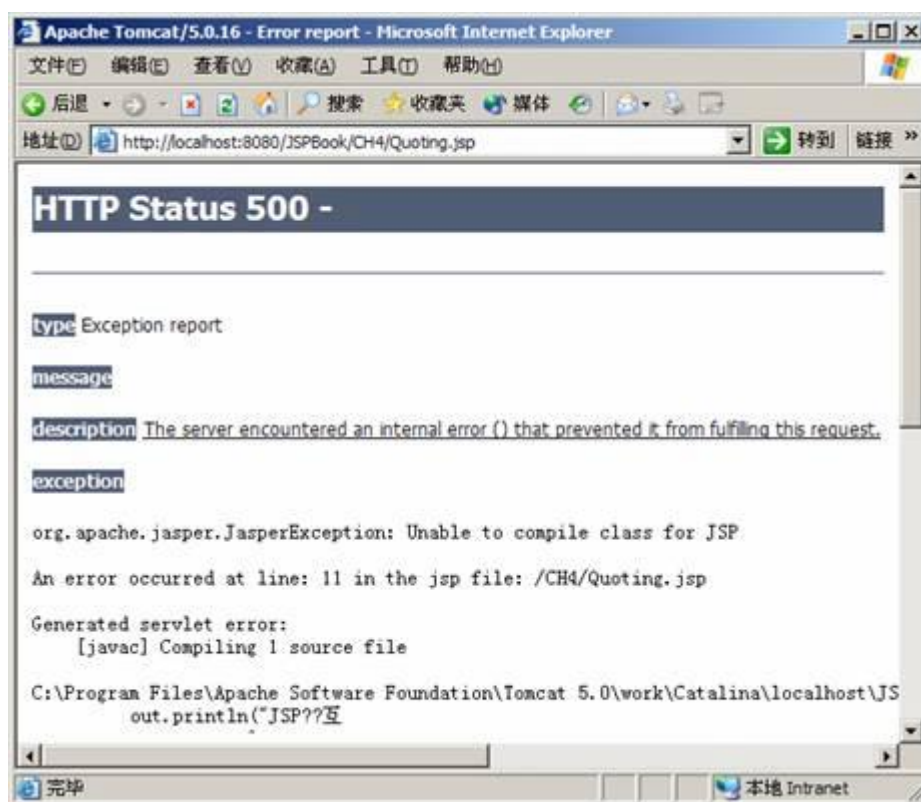


图 4-1 Quoting.jsp 的执行结果

通常为了避免产生这样的结果，因此程序中遇到显示%>时，要改写为%\>，所以上面的程序代码应改写为：

Quoting1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```



```
<html>

<head>

  <title>CH4 - Quoting1.jsp</title>

</head>

<body>

<h2>Quoting 范例程序 2</h2>

<%

    out.println("JSP 以%\>作为结束符号");

%>

</body>

</html>
```

Quoting1.jsp 的执行结果如图 4-2 所示：



图 4-2 Quoting1.jsp 的执行结果

除了 %> 要改为 %\>，当遇到<%、'、"、\ 时都要做适当修改，如下所示：

单引号 ' 改为 \'

双引号 " 改为 \"

斜线 \ 改为 \\

起始标签 <% 改为 <%

结束标签 %> 改为 %\>

最后再举个例子，如下：

Quoting2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

  <title>CH4 - Quoting2.jsp</title>

</head>

<body>

<h2>Quoting 范例程序 3</h2>

<%

    out.println("JSP 遇到 \'、\"、\\、&lt;%、%\> 时需要做适当的修改");

%>
```

```
</body>
```

```
</html>
```

执行的结果如图 4-3 所示:



图 4-3 Quoting2.jsp 的执行结果

4-4 Directives Elements(1)

指令(Directives)主要用来提供整个 JSP 网页相关的信息, 并且用来设定 JSP 网页的相关属性, 例如: 网页的编码方式、语法、信息等。

起始符号为: `<%@`

终止符号为: `%>`

内文部分就是一些指令和一连串的属性设定, 如下所示:

```
<%@ directive { attribute ="value" } * %>
```

什么叫做一连串的属性设定? 举例来说, 当我们设定两个属性时, 可以将之合二为一, 如下:

```
<%@ directive attribute1 = "value1" %>
```

```
<%@ directive attribute2 = "value2" %>
```

亦可以写成:

```
<%@ directive attribute1 = "value1" attribute2 = "value2" %>
```

在 JSP 1.2 的规范中，有三种指令：page、include 和 taglib，每一种指令都有各自的属性。接下来，我们会依序为各位读者介绍这三种指令。

JSP 2.0 新增 Tag File 的功能，Tag File 是以 .tag 作为扩展名的。在 Tag File 中，因为它并不是 JSP 网页，所以不能使用 page 指令，但是它可以使用 include 和 taglib 指令。除此之外，Tag File 还有自己本身的指令可以使用，如：tag、attribute 和 variable。

有关 Tag File 的指令，笔者在“第十六章：Simple Tag 与 Tag File”中再做详细介绍，本章先暂时不谈论它。

4-4-1 page 指令

page 指令是最复杂的 JSP 指令，它的主要功能为设定整个 JSP 网页的属性和相关功能。page 指令的基本语法如下：

```
<%@ page attribute1="value1" attribute2= "value2" attribute3=...%>
```

page 指令是以<%@ page 起始，以%>结束。像所有 JSP 标签元素一样，page 指令也支持以 XML 为基础的语法，如下所示：

```
<jsp:directive.page attribute1="value1" attribute2= "value2" />
```

page 指令有 11 个属性，如表 4-1 所示：

表 4-1

属 性	定 义
language "scriptingLanguage"	= 主要指定 JSP Container 要使用什么语言来编译 JSP 网页。JSP 2.0 规范中指出，目前只可以使用 Java 语言，不过未来不排除增加其他语言，如 C、C++、Perl 等等。默认值为 Java
extends = "className"	主要定义此 JSP 网页产生的 Servlet 是继承哪个父类
import = "importList"	主要定义此 JSP 网页可以使用哪些 Java API
session = "true false"	决定此 JSP 网页是否可以使用 session 对象。默认值为 true
buffer = "none size in kb"	决定输出流 (output stream) 是否有缓冲区。默认值为 8KB 的缓冲区
autoFlush = "true false"	决定输出流的缓冲区是否要自动清除，缓冲区满了会产生异常 (Exception)。默认值为 true
isThreadSafe = "true false"	主要是告诉 JSP Container，此 JSP 网页能处理超过一个

false"	以上的请求。默认值为 true，如果此值设为 false，SingleThreadModel 将会被使用。SingleThreadModel 在 Servlet 2.4 中已经声明不赞成使用(deprecate)
info = "text"	主要表示此 JSP 网页的相关信息
errorPage = "error_url"	表示如果发生异常错误时，网页会被重新指向那一个 URL
isErrorPage = "true false"	表示此 JSP Page 是否为处理异常错误的网页
contentType = "ctinfo"	表示 MIME 类型和 JSP 网页的编码方式
pageEncoding = "ctinfo"	表示 JSP 网页的编码方式
isELIgnored = "true false"	表示是否在此 JSP 网页中执行或忽略 EL 表达式。如果为 true 时，JSP Container 将忽略 EL 表达式；反之为 false 时，EL 表达式将会被执行

下面的范例都合乎语法规则：

```
<%@ page info = "this is a jsp page"%>

<%@ page language = "java" import = "java.net.* " %>

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

下面的范例也是 page 指令，不过并不合乎语法规则，因为 session 属性重复设定两次：

```
<%@ page language = "java" import = "java.net.* " session = "false" buffer = "16kb"
    autoFlush = "false" session = "false" %>
```

注意：只有 import 这个属性可以重复设定，其他则否。

```
<%@ page import = "java.net.* " %>

<%@ page import = "java.util.List " %>
```

另外再举个较常见的错误例子：

```
<%@ page language="java" contentType="text/html";charset = "Big5" %>
```

应该改为：

```
<%@ page language="java" contentType="text/html; charset=Big5" %>
```

通常我们都以为只要把 `charset` 设为 `Big5` 的编码方式，就能够顺利显示出所需要的中文，不过有时候在显示一些特别的中文字时，例如：碁，会变成乱码。如下范例：

Big5.jsp

```
<%@ page contentType="text/html; charset=Big5" %>

<html>

<head>

  <title>CH4 - Big5.jsp</title>

</head>

<body>

<h2>使用 Big5 編碼，無法正確顯示某些中文字</h2>

<%

    out.println("宏? 碁 q 脑公司");

%>

</body>

</html>
```

Big5.jsp 的执行结果如图 4-4 所示：



图 4-4 Big5.jsp 的执行结果

那么我们应该如何解决这个问题？很简单，只要把之前的 `charset = Big5` 改为 `charset = MS950` 的编码方式就能够解决这个问题，我们看下面这个范例：MS950.jsp。

MS950.jsp

```
<%@ page contentType="text/html; charset=MS950" %>

<html>

<head>

  <title>CH4 - MS950.jsp</title>

</head>

<body>

<h2>使用 MS950 編碼，能正確顯示"碁"</h2>

<%

    out.println("宏碁電腦公司");

%>
```

```
</body>
```

```
</html>
```

MS950.jsp 的执行结果如图 4-5:



图 4-5 MS950.jsp 的执行结果

使用最基本的 page 指令的范例程序 (二): Date.jsp

Date.jsp

```
<%@ page contentType="text/html; charset=GB2312 " %>

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

<html>

<head>

    <title>CH4 - Date.jsp</title>

</head>

<body>

<h2>使用 java.util.Date 显示目前时间</h2>
```



```
<%  
  
    Date date = new Date();  
  
    out.println("现在时间: "+date);  
  
%>  
  
</body>  
  
</html>
```

因为 `Date.jsp` 要显示出现在的时间, 所以要先导入(import) `java.util` 这个套件, 才可以使用 `Date()` 类。执行结果如图 4-6 所示:



图 4-6 `Date.jsp` 的执行结果

如果在一个 JSP 网页同时须要导入很多套件时:

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

亦可以写为:

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

直接使用逗号 “,” 分开, 然后就可以一直串接下去。

4-4-2 include 指令

`include` 指令表示：在 JSP 编译时插入一个包含文本或代码的文件，这个包含的过程是静态的，而包含的文件可以是 JSP 网页、HTML 网页、文本文件，或是一段 Java 程序。

注意

包含文件中要避免使用 `<html>`、`</html>`、`<body>`、`</body>`，因为这将会影响在原来 JSP 网页中同样的标签，这样做有时会导致错误。

`include` 指令的语法如下：

```
<%@ include file = "relativeURLspec" %>
```

`include` 指令只有一个属性，那就是 `file`，而 `relativeURLspec` 表示此 `file` 的路径。像所有 JSP 标签元素一样，`include` 指令也支持以 XML 为基础的语法，如下所示：

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

注意

`<%@ include %>` 指令是静态包含其他的文件。所谓的静态是指 `file` 不能为一变量 URL，例如：

```
<% String URL="JSP.html" ; %>
```

```
<%@ include file = "<%= URL %>" %>
```

也不可以在 `file` 所指定的文件后接任何参数，如下：

```
<%@ include file = "javaworld.jsp?name=browser" %>
```

同时，`file` 所指的路径必须是相对于此 JSP 网页的路径。

笔者写了一个 `Include.jsp` 范例程序，它 `include` 一份 HTML 的网页，网页文件名叫做 `Hello.html`，看看执行之后，会有什么结果产生。

`Include.jsp`

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```
<html>
```

```
<head>

  <title>CH4 - Include_Html.jsp</title>

</head>

<body>

<h2>include 指令</h2>

<%@ include file="Hello.html" %>

<%

    out.println("欢迎大家进入 JSP 的世界");

%>

</body>

</html>
```

Hello.html

JSP 2.0 Tech Reference

执行结果如图 4-7。

注意
Hello.html 网页内容中有中文时，Tomcat 5.0.16 执行 Include.jsp 时，无法正确显示 Hello.html 网页的中文，会产生乱码。但是笔者使用 Resin 来执行时，却可以顺利显示中文。



图 4-7 Include.jsp 的执行结果

4-4-3 taglib 指令

taglib 指令是 JSP 1.1 新增进来的功能，能够让用户自定义新的标签。这里只是先做一个简单介绍，在第十五章再为各位读者详细介绍。

taglib 指令的语法如下：

```
<%@ taglib uri = "tagLibraryURI" prefix="tagPrefix" %>
```

像所有 JSP 标签元素一样，taglib 指令也支持以 XML 为基础的语法，如下所示（见表 4-2）：

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

表 4-2

属 性	定 义
uri = "tagLibraryURI"	主要是说明 taglibrary 的存放位置
prefix="tagPrefix"	主要用来区分多个自定义标签

范例：

```
<%@ taglib uri = "/supertags/" prefix="super" %>

.....

<super:doMagic>

    .....

    .....

</super:doMagic>
```

4-5 Scripting Elements

Scripting Elements 包含三部分:

1. 声明(Declarations)
2. Scriptlets
3. 表达式 (Expressions)

示例:

```
<%! 这是声明 %>
```

```
<% 这是 Scriptlets %>
```

```
<%= 这是表达式 %>
```

4-5-1 声明 (Declarations)

在 JSP 程序中声明合法的变量和方法。声明是以<%! 为起始; %> 为结尾。

声明的语法:

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

范例:

```
<%! int i = 0; %>

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

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

<%! public String f(int i) {if ( i < 3 ) return ("i 小于 3");}; %>
```

使用<%! %>可以声明你在 JSP 程序中要用的变量和方法, 你可以一次声明多个变量和方法, 只要最后以分号 “;” 结尾就行, 当然这些声明在 Java 中要是合法的。

每一个声明仅在一个页面中有效, 如果你想每个页面都用到一些声明, 最好把它们写成一个单

独的 JSP 网页，然后用`<%@ include %>`或`<jsp:include %>`元素包含进来。

注意

使用`<%! %>`方式所声明的变量为全局变量，即表示：若同时 n 个用户在执行此 JSP 网页时，将会共享此变量。因此笔者强烈建议读者，千万别使用`<%! %>`来声明您的变量。

若要声明变量时，请直接在`<% %>`之中声明使用即可。

4-5-2 Scriptlets

Scriptlet 中可以包含有效的程序片段，只要是合乎 Java 本身的标准语法即可。通常我们主要的程序也是写在这里面，Scriptlet 是以 `<%` 为起始；`%>` 为结尾。

Scriptlet 的语法：

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

范例：

```
<%  
  
    String name = null;  
  
    if (request.getParameter("name") == null) {  
  
%>  
.....  
.....  
  
<%  
  
    }  
  
    else  
  
    {  
  
        out.println("HI ... "+name);  
  
    }  
  
%>
```

Scriptlet 能够包含多个语句，方法，变量，表达式，因此它能做以下的事：

(1) 声明将要用到的变量或方法；

(2) 显示出表达式；

(3) 使用任何隐含的对象和使用<jsp:useBean>声明过的对象，编写 JSP 语句（如果你在使用 Java 语言，这些语句必须遵从 Java Language Specification）；

(4) 当 JSP 收到客户端的请求时，Scriptlet 就会被执行，如同 Servlet 的 doGet()、doPost()，如果 Scriptlet 有显示的内容会被存在 out 对象中，然后再利用 out 对象中的 println() 方法显示出结果。

4-5-3 表达式 (Expressions)

Expressions 标签是以 <%= 为起始；%> 为结尾，其中间内容包含一段合法的表达式，例如：使用 Java 的表达式。

Expressions 的语法：

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

范例：

```
<font color="blue"><%= getName() %></font>  
<%= (new java.util.Date()).toLocaleString() %>
```

表达式在执行后会被自动转化为字符串，然后显示出来。

当你在 JSP 中使用表达式时请记住以下几点：

1. 不能使用分号 “;” 来作为表达式的结束符号，如下：

```
<%= (new java.util.Date()).toLocaleString(); %>
```

但是同样的表达式用在 Scriptlet 中就需要以分号来结尾了。

2. 这个表达式元素能够包括任何 Java 语法，有时候也能作为其他 JSP 元素的属性值。

第四章 JSP 语法

4-6 Action Elements

JSP 2.0 规范中定义一些标准 action 的类型，JSP Container 在实现时，也完全遵照这个规范而制定。Action 元素的语法以 XML 为基础，所以，在使用时大小写是有差别的，例如：
<jsp:getproperty>和<jsp:getProperty>是有所差别的，因此在撰写程序时，必须要特别注意。

目前 JSP 2.0 规范中，主要有 20 项 Action 元素：

<jsp:useBean>

<jsp:setProperty>

<jsp:getProperty>

<jsp:include>

<jsp:forward>

<jsp:param>

<jsp:plugin>

<jsp:params>

<jsp:fallback>

<jsp:root>

<jsp:declaration>

<jsp:scriptlet>

<jsp:expression>

<jsp:text>

<jsp:output>

<jsp:attribute>

<jsp:body>

`<jsp:element>`

`<jsp:invoke>`

`<jsp:doBody>`

笔者将这 20 个 `action` 元素分为五类：

第一类有 3 个 `action` 元素，它们都用来存取 `JavaBean`，因此这部分将在“第八章：JSP 与 `JavaBean`”详细地介绍。

第二类有 6 个 `action` 元素，这部分是 JSP 1.2 原有的 `action` 元素，接下来将会介绍它们。

第三类有 6 个 `action` 元素，它们主要用在 JSP Document 之中。其中 `<jsp:output>` 是 JSP 2.0 新增的元素。

第四类有 3 个 `action` 元素，它们主要用来动态产生 XML 元素标签的值，这 3 个都是在 JSP 2.0 中加入进来的元素。

第五类有 2 个 `action` 元素，它们主要用在 Tag File 中，这部分将在“第十六章：Simple Tag 与 Tag File”再来介绍。

补充

JSP Document：使用 XML 语法所写成的 JSP 网页。例如：

```
<jsp:scriptlet>
    String name="Mike";
</jsp:scriptlet>
Hi !<jsp:expression>name</jsp:expression>
```

4-6-1 `<jsp:include>`

`<jsp:include>` 元素允许你包含动态和静态文件，这两种产生的结果是不尽相同的。如果包含进来的只是静态文件，那么只是把静态文件的内容加到 JSP 网页中；如果包含进来的为动态文件，那

么这个被包含的文件也会被 JSP Container 编译执行。

一般而言,你不能直接从文件名称上来判断一个文件是动态的还是静态的,例如: `Hello.jsp` 就有可能只是单纯包含一些信息而已,而不须要执行。但是 `<jsp:include>` 能够自行判断此文件是动态的还是静态的,于是能同时处理这两种文件。

`<jsp:include>`的语法:

```
<jsp:include page="{urlSpec | <%= expression %>}" flush="true | false" />
```

或

```
<jsp:include page="{urlSpec | <%= expression %>}" flush="true | false" >
    <jsp:param name="PN" value="{PV | <%= expression %>}" /> *
</jsp:include>
```

说明:

`<jsp:include>`有两个属性: `page` 和 `flush`。

page: 可以代表一个相对路径,即你所要包含进来的文件位置或是经过表达式所运算出的相对路径。

flush: 接受的值为 `boolean`, 假若为 `true`, 缓冲区满时,将会被清空。`flush` 的默认值为 `false`。

在此需要补充一点: 在 JSP 1.2 之前, `flush` 必须设为 `true`。

你还可以用 `<jsp:param>` 传递一个或多个参数给 JSP 网页。

范例:

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

<jsp:include page="Hello.html" />

<jsp:include page="scripts/login.jsp">

    <jsp:param name="username" value="browser" />

    <jsp:param name="password" value="1234" />

</jsp:include>
```

4-6-2 <jsp:forward>

`<jsp:forward>`这个标签的定义：将客户端所发出来的请求，从一个 JSP 网页转交给另一个 JSP 网页。不过有一点要特别注意，`<jsp:forward>`标签之后的程序将不能执行。笔者用例子来说明：

```
<%
    out.println("会被执行 !!! ");
%>

<jsp:forward page="Quoting2.jsp">

<jsp:param name="username" value="Mike" />

</jsp:forward>

<%
    out.println("不会执行 !!!");
%>
```

上面这个范例在执行时，会打印出“会被执行 !!!”，不过随后马上会转入到 `SayHello.jsp` 的网页中，至于 `out.println("不会执行 !!! ")` 将不会被执行。

`<jsp:forward>`的语法：

```
<jsp:forward page="{relativeURL" | "<%= expression %>"}" />
```

或

```
<jsp:forward page="{relativeURL" | "<%= expression %>"}" >

    <jsp:param name="PN" value="{PV | <%= expression %>}" /> *

</jsp:forward>
```

说明：

如果你加上`<jsp:param>`标签，你就能够向目标文件传送参数和值，不过这些目标文件必须也是一个能够取得这些请求参数的动态文件，例如：`.cgi`、`.php`、`.asp` 等等。

`<jsp:forward>`只有一个属性 `page`。`page` 的值，可以是一个相对路径，即你所要重新导向的网页位置，亦可以是经过表达式运算出的相对路径。

范例：

```
<jsp:forward page="/SayHello.jsp" />
```

或者

```
<jsp:forward page="/SayHello.jsp">
    <jsp:param name="username" value="Mike" />
</jsp:forward>
```

4-6-3 <jsp:param>

`<jsp:param>`用来提供 `key/value` 的信息，它也可以与`<jsp:include>`、`<jsp:forward>`和`<jsp:plugin>` 一起搭配使用。

当在用`<jsp:include>`或者`<jsp:forward>`时，被包含的网页或转向后的网页会先看看 `request` 对象里除了原本的参数值之外，有没有再增加新的参数值，如果有增加新的参数值时，则新的参数值在执行时，有较高的优先权。例如：

一个 `request` 对象有一个参数 `A = foo`；另一个参数 `A = bar` 是在转向时所传递的参数，则网页中的 `request` 应该会为 `A = bar,foo`。注意：新的参数值有较高的优先权。

`<jsp:param>`的语法：

```
<jsp:param name="ParameterName" value="ParameterValue" />
```

`<jsp:param>`有两个属性：`name` 和 `value`。`name` 的值就是 `parameter` 的名称；而 `value` 的值就是 `parameter` 的值。

范例：

```
<jsp:param name="username" value="Mike" />
<jsp:param name="password" value="Mike007" />
```

4-6-4 <jsp:plugin>、<jsp:params>和<jsp:fallback>

<jsp:plugin>用于在浏览器中播放或显示一个对象(通常为 Applet 或 Bean)。

当 JSP 网页被编译后送往浏览器执行时，<jsp:plugin>将会根据浏览器的版本替换成<object>标签或者<embed>标签。一般来说，<jsp:plugin>会指定对象 Applet 或 Bean，同样也会指定类的名字和位置，另外还会指定将从哪里下载这个 Java 组件。

注意

<object>用于 HTML 4.0，<embed>用于 HTML 3.2。

<jsp:plugin>的语法：

```
<jsp:plugin type="bean | applet"

    code="objectCode"

    codebase="objectCodebase"

    [ align="alignment" ]

    [ archive="archiveList" ]

    [ height="height" ]

    [ hspace="hspace" ]

    [ jreversion="jreversion" ]

    [ name="ComponentName" ]

    [ vspace="vspace" ]

    [ width="width" ]

    [ nspluginurl="URL" ]

    [ iepluginurl="URL" ] >

[ <jsp:params>

    [ <jsp:param name="PN" value="{PV | <%= expression %>}" /> ] +

    </jsp:params> ]
```

```
[ <jsp:fallback> text message for user </jsp:fallback> ]  
  
</jsp:plugin>
```

说明:

- `type="bean | applet"`:

对将被执行的对象类型, 你必须指定是 **Bean** 还是 **Applet**, 因为这个属性没有默认值。

- `code="objectCode"`:

将被 **Java Plugin** 执行的 **Java** 类名称, 必须以 `.class` 结尾, 并且 `.class` 类文件必须存在于 `codebase` 属性所指定的目录中。

- `codebase="objectCodebase"`:

如果你没有设定将被执行的 **Java** 类的目录 (或者是路径) 的属性, 默认值为使用 `<jsp:plugin>` 的 **JSP** 网页所在目录。

- `align="alignment"` :

图形、对象、**Applet** 的位置。`align` 的值可以为:

`bottom`、`top`、`middle`、`left`、`right`

- `archive=" archiveList"`:

一些由逗号分开的路径名用于预先加载一些将要使用的类, 此做法可以提高 **Applet** 的性能。

- `name=" ComponentName"`:

表示这个 **Bean** 或 **Applet** 的名字。

- `height="height" width="width"`:

显示 **Applet** 或 **Bean** 的长、宽的值, 单位为像素 (`pixel`)。

- `hspace="hspace" vspace="vspace"`:

表示 **Applet** 或 **Bean** 显示时在屏幕左右、上下所需留下的空间, 单位为像素 (`pixel`)。

- `jreversion="jreversion"`:

表示 Applet 或 Bean 执行时所需的 Java Runtime Environment (JRE) 版本, 默认值是 1.1。

- `nspluginurl="URL"`:

表示 Netscape Navigator 用户能够使用的 JRE 的下载地址, 此值为一个标准的 URL。

- `iepluginurl="URL"`:

表示 IE 用户能够使用的 JRE 的下载地址, 此值为一个标准的 URL。

- `<jsp:params>`

[`<jsp:param name="PN" value="{PV | <%= expression %>}" />`] +

`</jsp:params>`

你可以传送参数给 Applet 或 Bean。

- `<jsp:fallback> unable to start plugin </jsp:fallback>`

一段文字用于: 当不能启动 Applet 或 Bean 时, 那么浏览器会有一段错误信息。

范例:

```
<jsp:plugin type="applet" code="Molecule.class" codebase="/html">

  <jsp:params>

    <jsp:param name="molecule" value="molecules/benzene.mol" />

  </jsp:params>

  <jsp:fallback>

    <p>Unable to start plugin</p>

  </jsp:fallback>

</jsp:plugin>
```

4-6-5 `<jsp:element>`、`<jsp:attribute>`和`<jsp:body>`

`<jsp:element>`元素用来动态定义 XML 元素标签的值。

`<jsp:element >`的语法:

```
<jsp:element name="name">

    本 体 内 容

</jsp:element>
```

或

```
<jsp:element name="name">

    <jsp:attribute>
    ...
    </jsp:attribute>

    ...

    <jsp:body>
    ...
    </jsp:body>

</jsp:element>
```

`<jsp:element>` 只有一个属性 `name`。`name` 的值就是 XML 元素标签的名称。

范例 1:

```
<jsp:element name="firstname"></jsp:element>
```

执行的结果如下:

```
<firstname></firstname>
```

范例 2:

```
<jsp:element name="firstname">

    <jsp:attribute name="name">Mike</jsp:attribute>

    <jsp:body>Hello</jsp:body>

</jsp:element>
```

执行的结果如下:


```
<firstname name="Mike">Hello</firstname>
```

4-6-6 <jsp:attribute>

<jsp:attribute>元素主要有两个用途:

- (1) 当使用在<jsp:element>之中时, 它可以定义 XML 元素的属性, 如上述的范例 2。
- (2) 它可以用来设定标准或自定义标签的属性值。如下范例 1:

<jsp:attribute>的语法:

```
<jsp:attribute name="name" trim="true | false">  
    本内容  
</jsp:attribute >
```

<jsp:attribute>有两个属性: **name** 和 **trim**。其中 **name** 的值就是标签的属性名称。**trim** 可为 **true** 或 **false**。假若为 **true** 时, <jsp:attribute>本内容的前后空白, 将被忽略; 反之, 若为 **false**, 前后空白将不被忽略。**trim** 的默认值为 **true**。

范例:

```
<jsp:useBean id="foo" class="jsp2.examples.FooBean">  
    Bean created!  Setting foo.bar...<br>  
    <jsp:setProperty name="foo" property="bar">  
        <jsp:attribute name="value">  
            Hello World  
        </jsp:attribute>  
    </jsp:setProperty>  
</jsp:useBean>  
  
<br>  
Result: <jsp:getProperty name="foo" property="bar">
```

执行的结果如下：

```
Bean created! Setting foo.bar...  
Result: Hello World
```

其实上述的范例和下面的例子一样：

```
<jsp:useBean id="foo" class="jsp2.examples.FooBean">  
    Bean created! Setting foo.bar...<br>  
    <jsp:setProperty name="foo" property="bar" value="Hello World" >  
    </jsp:setProperty>  
</jsp:useBean>  
  
<br>  
Result: <jsp:getProperty name="foo" property="bar">
```

有关<jsp:useBean>、<jsp:setProperty>和<jsp:getProperty>在“第八章：JSP 与 JavaBean”会有更详细的说明。

4-6-7 <jsp:body>

<jsp:body>用来定义 XML 元素标签的本体内容。

<jsp:body >的语法：

```
<jsp:body>  
    本体内容  
</jsp:body>
```

<jsp:body>没有任何的属性。

范例 1：

```
<jsp:element name="firstname">  
    <jsp:attribute name="name">Mike</jsp:attribute>
```

```
<jsp:body>Hello</jsp:body>

</jsp:element>
```

执行的结果如下：

```
<firstname name="Mike">Hello</firstname>
```

范例 2：

```
<jsp:element name="firstname">

    <jsp:attribute name="name">Mike</jsp:attribute>

</jsp:element>
```

执行的结果如下：

```
<firstname name="Mike" />
```

4-7 错误处理

通常 JSP 在执行时，在两个阶段下会发生错误。

JSP 网页 → Servlet 类

Servlet 类处理每一个请求时

在第一阶段时，产生的错误我们称为 Translation Time Processing Errors；在第二阶段时，产生的错误我们称为 Client Request Time Processing Errors。接下来我们将针对这两个阶段产生错误的原因和解决方法，为各位读者做详细的介绍。

4-7-1 Translation Time Processing Errors

Translation Time Processing Errors 产生的主要原因：我们在撰写 JSP 时的语法有错误，导致 JSP Container 无法将 JSP 网页编译成 Servlet 类文件(.class)，例如：500 Internal Server Error，500 是指 HTTP 的错误状态码，因此是 Server Error，如图 4-8。

通常产生这种错误时，可能是 JSP 的语法有错误，或是 JSP Container 在一开始安装、设定时，有不适当的情形发生。解决的方法就是再一次检查程序是否有写错的，不然也有可能是 JSP Container 的 bug。

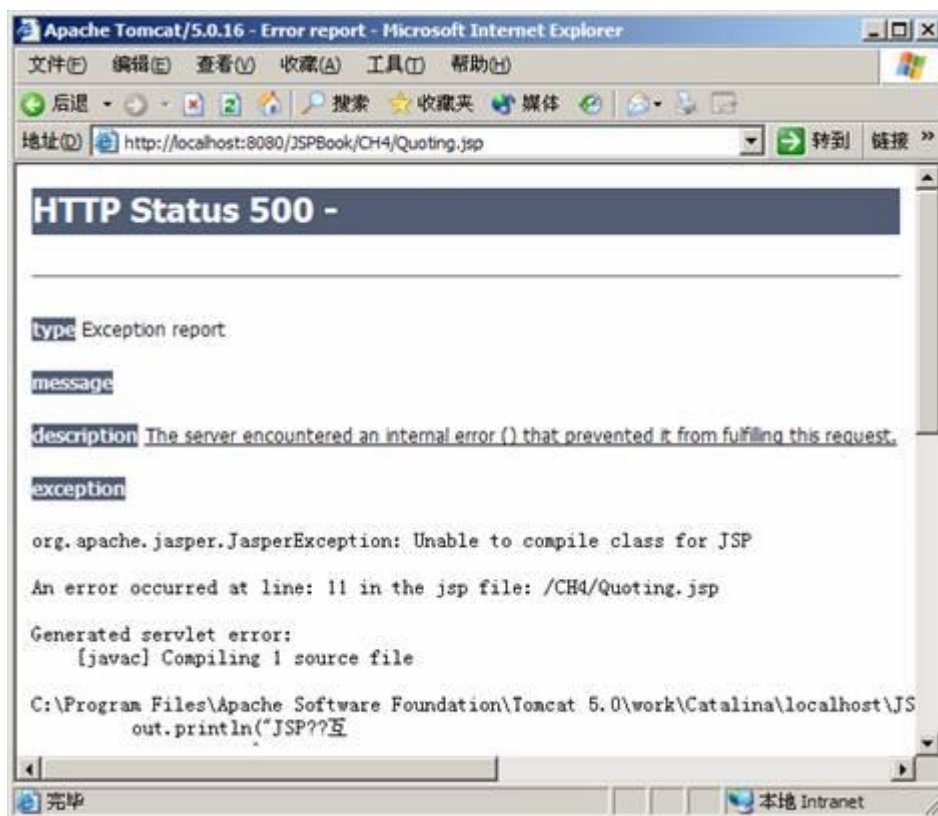


图 4-8 500 Internal Server Error

4-7-2 Client Request Time Processing Errors

Client Request Time Processing Errors 错误的发生, 往往不是语法错误, 而可能是逻辑上的错误, 简单地说, 你写一个计算除法的程序, 当用户输入的分母为零时, 程序会发生错误并抛出异常(Exception), 交由异常处理(Exception Handling)机制做适当的处理。

对于这种错误的处理, 我们通常会交给 `errorPage` 去处理。下面举个例子, 我想各位读者应该能够很容易了解。

使用 `errorPage` 的范例程序 (一): `ErrorPage.jsp`

`ErrorPage.jsp`

```
<%@ page contentType="text/html; charset=GB2312" errorPage="Error.jsp" %>

<html>
```

```
<head>

    <title>CH4 - ErrorPage.jsp</title>

</head>

<body>

<h2>errorPage 的范例程序</h2>

<%!

    private double toDouble(String value)

    {

        return(Double.valueOf(value).doubleValue());

    }

%>

<%

    double num1 = toDouble(request.getParameter("num1"));

    double num2 = toDouble(request.getParameter("num2"));

%>

您传入的两个数字为: <%= num1 %> 和 <%= num2 %><br>

两数相加为 <%= (num1+num2) %>

</body>

</html>
```

ErrorPage.jsp 程序中，我们使用 page 指令中的 errorPage 属性，告诉 JSP Container，如果在程序中有错误产生时，会自动交给 Error.jsp 处理。

```
<%@ page contentType="text/html; charset=GB2312" errorPage="Error.jsp" %>
```

我们声明一个函数叫 `toDouble`，主要把我们接收进来的字符串转成 `Double` 的类型，才能做加法运算。

```
<%!  
  
    private double toDouble(String value)  
  
    {  
  
        return(Double.valueOf(value).doubleValue());  
  
    }  
  
%>
```

接收 `num1` 和 `num2` 两个字符串，并且将它们转为 `double` 的类型，并各自命名为 `num1` 和 `num2`。

```
<%  
  
    double num1 = toDouble(request.getParameter("num1"));  
  
    double num2 = toDouble(request.getParameter("num2"));  
  
%>
```

`Error.jsp`

```
<%@ page contentType="text/html; charset=GB2312" isErrorPage="true" %>  
  
<%@ page import="java.io.PrintWriter" %>  
  
<html>  
  
<head>  
  
    <title>CH4 - Error.jsp</title>  
  
</head>  
  
<body>
```

```
<h2>errorPage 的范例程序</h2>

<p>ErrorPage.jsp 错误产生: <l><%= exception %></l></p><br>

<pre>

问题如下: <% exception.printStackTrace(new PrintWriter(out)); %>

</pre>

</body>

</html>
```

Error.jsp 主要处理 ErrorPage.jsp 所产生的错误, 所以在 ErrorPage.jsp 中 page 指令的属性 errorPage 设为 Error.jsp, 因此, 若 ErrorPage.jsp 有错误发生时, 会自动转到 Error.jsp 来处理。

Error.jsp 必须设定 page 指令的属性 isErrorPage 为 true, 因为 Error.jsp 是专门用来处理错误的网页。

```
<%@ page contentType="text/html; charset=GB2312" isErrorPage="true" %>
```

下面这几行代码用来显示出错误发生的原因:

```
<p>ErrorPage.jsp 错误产生: <l><%= exception %></l></p><br>

<PRE>

问题如下: <% exception.printStackTrace(new PrintWriter(out)); %>

</PRE>
```

由于在这个程序中并没有做一个窗体来输入两个数字, 所以必须手动在 URL 后输入 num1 和 num2 的值。如图 4-9 所示:



图 4-9 ErrorPage.jsp 的执行结果

上一个范例的参数设定为:

```
http://localhost:8080/JSPBook/CH4/ErrorPage.jsp?num1=100&num2=245
```

此时 num1 的值为 100, num2 的值为 245, 所以相加的结果就为 345。但是, 如果 num1 或 num2 的参数不是数字的话, 例如:

```
http://localhost:8080/JSPBook/CH4/ErrorPage.jsp?num1=javaWorld&num2=245
```

那么就会产生错误结果如图 4-10。

当 ErrorPage.jsp 产生错误时, 就会交由 Error.jsp 去处理, 所以您看到的结果, 其实是执行 Error.jsp 后的结果。



图 4-10 ErrorPage.jsp 执行时产生错误

5-1 属性(Attribute)与范围(Scope)

有些 JSP 程序员会将 request、session、application 和 pageContext 归为一类，原因在于：它们皆能借助 setAttribute()和 getAttribute()来设定和取得其属性(Attribute)，通过这两种方法来做到**数据分享**。

我们先来看下面这段小程序：

Page1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

  <title>CH5 - Page1.jsp</title>

</head>

<body>

<br>

<%

    application.setAttribute("Name","mike");

    application.setAttribute("Password","browser");

%>

<jsp:forward page="Page2.jsp" />

</body>

</html>
```

在这个程序中,笔者设定两个属性:Name、Password,其值为:mike、browser。然后再转交(forward)到 Page2.jsp。我只要在 Page2.jsp 当中加入 application.getAttribute(),就能取得在 Page1.jsp 设定的数据。

Page2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - Page2.jsp</title>

</head>

<body>

<%

    String Name = (String) application.getAttribute("Name");

    String Password = (String) application.getAttribute("Password");

    out.println("Name = "+Name);

    out.println("Password = "+ Password);

%>

</body>

</html>
```

Page1.jsp 的执行结果如图 5-1 所示。



图 5-1 Page1.jsp 的执行结果

从图 5-1 可以看出,此时 Name 的值就会等于 mike, Password 的值就等于 browser。看完这个小范例之后,读者有没有发现网页之间要传递数据时,除了可以使用窗体、隐藏字段来完成之外,JSP 技术还提供给开发人员一项传递数据的机制,那就是利用 `setAttribute()` 和 `getAttribute()` 方法,如同 Page1.jsp 和 Page2.jsp 的做法。不过它还是有些限制的,这就留到下一节来说明。

在上面 Page1.jsp 和 Page2.jsp 的程序当中,是将数据存入到 `application` 对象之中。除了 `application` 之外,还有 `request`、`pageContext` 和 `session`,也都可以设定和取得属性值,那它们之间有什么分别吗?

它们之间最大的差别在于**范围(Scope)**不一样,这个概念有点像 C、C++ 中的全局变量和局部变量的概念。接下来就介绍 JSP 的范围。

5-1-1 JSP Scope—Page

JSP 有四种范围,分别为 Page、Request、Session、Application。所谓的 Page,指的是单单一页 JSP Page 的范围。若要将数据存入 Page 范围时,可以用 `pageContext` 对象的 `setAttribute()` 方法;若要取得 Page 范围的数据时,可以使用 `pageContext` 对象的 `getAttribute()` 方法。我们将之前的范例做小幅度的修改,将 `application` 改为 `pageContext`。

PageScope1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

<title>CH5 - PageScope1.jsp</title>
```

```
</head>

<body>

<h2>Page 范围 - pageContext</h2>

<%

    pageContext.setAttribute("Name","mike");

    pageContext.setAttribute("Password","browser");

%>

<jsp:forward page="PageScope2.jsp" />

</body>

</html>
```

PageScope2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - PageScope2.jsp</title>

</head>

<body>

<h2>Page 范围 - pageContext</h2>

</br>

<%

    String Name = (String)pageContext.getAttribute("Name");
```

```
String Password = (String)pageContext.getAttribute("Password");

out.println("Name = "+Name);

out.println("Password = "+ Password);

%>

</body>

</html>
```

执行结果如图 5-2 所示。



图 5-2 PageScope1.jsp 的执行结果

这个范例程序和之前有点类似，只是之前的程序是 `application`，现在改为 `pageContext`，但是结果却大不相同，`PageScope2.jsp` 根本无法取得 `PageScope1.jsp` 设定的 `Name` 和 `Password` 值，因为在 `PageScope1.jsp` 当中，是把 `Name` 和 `Password` 的属性范围设为 `Page`，所以 `Name` 和 `Password` 的值只能在 `PageScope1.jsp` 当中取得。若修改 `PageScope1.jsp` 的程序，重新命名为 `PageScope3.jsp`，如下：

PageScope3.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - PageScope3.jsp</title>

</head>
```

```
<body>

<h2>Page 范围 - pageContext</h2>

</br>

<%

    pageContext.setAttribute("Name","mike");

    pageContext.setAttribute("Password","browser");


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

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


    out.println("Name = "+Name);

    out.println("Password = "+ Password);

%>

</body>

</html>
```

PageScope3.jsp 的执行结果如图 5-3 所示。



图 5-3 PageScope3.jsp 的执行结果

经过修改后的程序，Name 和 Password 的值就能顺利显示出来。这个范例主要用来说明一个概念：若数据设为 Page 范围时，数据只能在同一个 JSP 网页上取得，其他 JSP 网页却无法取得该数据。

5-1-2 JSP Scope—Request

接下来介绍第二种范围：Request。Request 的范围是指在一 JSP 网页发出请求到另一个 JSP 网页之间，随后这个属性就失效。设定 Request 的范围时可利用 request 对象中的 `setAttribute()` 和 `getAttribute()`。我们再来看下列这个范例：

RequestScope1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - RequestScope1.jsp</title>

</head>

<body>

<h2>Request 范围 - request</h2>

<%

    request.setAttribute("Name","mike");

    request.setAttribute("Password","browser");

%>

<jsp:forward page="RequestScope2.jsp"/>
```

```
</body>
```

```
</html>
```

RequestScope2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```
<html>
```

```
<head>
```

```
    <title>CH5 - RequestScope2.jsp</title>
```

```
</head>
```

```
<body>
```

```
<h2>Request 范围 - request</h2>
```

```
<%
```

```
    String Name = (String) request.getAttribute("Name");
```

```
    String Password = (String) request.getAttribute("Password");
```

```
    out.println("Name = "+Name);
```

```
    out.println("Password = "+ Password);
```

```
%>
```

```
</body>
```

```
</html>
```

RequestScope1.jsp 的执行结果如图 5-4 所示。

现在将 Name 和 Password 的属性范围设为 Request, 当 RequestScope1.jsp 转向到 RequestScope2.jsp 时, RequestScope2.jsp 也能取得 RequestScope1.jsp 设定的 Name 和 Password 值。不过其他的 JSP 网页

无法得到 Name 和 Password 值, 除非它们也和 RequestScope1.jsp 有请求的关系。



图 5-4 RequestScope1.jsp 的执行结果

除了利用转向(forward)的方法可以存取 request 对象的数据之外, 还能使用包含(include)的方法。

假若我将 RequestScope1.jsp 的

```
<jsp:forward page="RequestScope2.jsp"/>
```

改为

```
<jsp:include page="RequestScope2.jsp" flush="true"/>
```

执行 RequestScope1.jsp 时, 结果还是和图 5-4 一样。表示使用<jsp:include>标签所包含进来的网页, 同样也可以取得 Request 范围的数据。

5-1-3 JSP Scope—Session、Application

表 5-2 介绍了最后两种范围: Session、Application。

表 5-2

范 围	说 明
Session	Session 的作用范围为一段用户持续和服务端所连接的时间, 但与服务端断线后, 这个属性就无效。只要将数据存入 session 对象, 数据的范围就为 Session
Application	Application 的作用范围在服务器一开始执行服务, 到服务器关闭为止。Application 的范围最大、停留的时间也最久, 所以使用时要特别注意, 不然可能会造成服务器负载越来越重的情况。只要将数据存入 application 对象, 数据的 Scope 就为 Application

表 5-3 列出了一般储存和取得属性的方法, 以下 pageContext、request、session 和 application 皆可使用。

注意

pageContext 并无 getAttributeNames()方法。

表 5-3

方 法	说 明
void setAttribute(String name, Object value)	设定 name 属性的值为 value
Enumeration getAttributeNamesInScope(int scope)	取得所有 scope 范围的属性
Object getAttribute(String name)	取得 name 属性的值
void removeAttribute(String name)	移除 name 属性的值

当我们使用 `getAttribute(String name)` 取得 name 属性的值时，它会回传一个 `java.lang.Object`，因此，我们还必须根据 name 属性值的类型做转换类型(Casting)的工作。例如：若要取得 String 类型的 Name 属性时：

```
String Name = (String)pageContext.getAttribute("Name");
```

若是 Integer 类型的 Year 属性时：

```
Integer Year = (Integer)session.getAttribute("Year");
```

到目前已大约介绍完 JSP 中四种范围(Scope)：Page、Request、Session 和 Application。假若我的数据要设为 Page 范围时，则只需要：

```
pageContext.setAttribute("Year", new Integer(2001));
```

若要为 Request、Session 或 Application 时，就分别存入 request、session 或 application 对象之中，如下：

```
request.setAttribute("Month", new Integer(12) );  
session.setAttribute("Day", new Integer(27) );  
application.setAttribute("Times", new Integer(10));
```

接下来就正式进入本章的主题：隐含对象(Implicit Object)。

5-2 与 Servlet 有关的隐含对象

与 Servlet 有关的隐含对象有两个：page 和 config。page 对象表示 Servlet 本身；config 对

象则是存放 `Servlet` 的初始参数值。

page 对象

`page` 对象代表 JSP 本身,更准确地说,它代表 JSP 被转译后的 `Servlet`,因此,它可以调用 `Servlet` 类所定义的方法,不过实际上,`page` 对象很少在 JSP 中使用。我们来看看以下的范例程序:

PageInfo.jsp

```
<%@ page info="JSP 2.0 技术手册" contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - PageInfo.jsp</title>

</head>

<body>

<h2>page 隐含对象</h2>

Page Info = <%= ((javax.servlet.jsp.HttpJspPage)page).getServletInfo() %>

</body>

</html>
```

这个例子中,我们先设定 `page` 指令的 `info` 属性为“JSP 2.0 技术手册”,`page` 对象的类型为 `java.lang.Object`,我们调用 `javax.servlet.jsp.HttpJspPage` 中 `getServletInfo()` 的方法,将 `Info` 打印出来,执行结果如图 5-5 所示。



图 5-5 PathInfo.jsp 的执行结果

config 对象

config 对象里存放着一些 Servlet 初始的数据结构，config 对象和 page 对象一样都很少被用到。config 对象实现于 `javax.servlet.ServletConfig` 接口，它共有下列四种方法：

```
public String getInitParameter(name)

public java.util.Enumeration getInitParameterNames( )

public ServletContext getServletContext( )

public String getServletName( )
```

上述前两种方法可以让 config 对象取得 Servlet 初始参数值，如果此数值不存在，就传回 null。

例如：当我们在 web.xml 中设定如下时：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"

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

    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"

    version="2.4">

:

    <servlet>
```

```
<servlet-name>ServletConfigurator</servlet-name>

<servlet-class>

    org.logicalcobwebs.proxool.configuration.ServletConfigurator

</servlet-class>


    <init-param>

        <param-name>propertyFile</param-name>

        <param-value>

            WEB-INF/classes/Proxool.properties

        </param-value>

    </init-param>

    <load-on-startup>1</load-on-startup>

</servlet>

:

</web-app>
```

那么我们就可以直接使用 `config.getInitParameter("propertyFile")` 来取得名称为 `propertyFile`、其值为 `WEB-INF/classes/Proxool.properties` 的参数。如下范例：

```
String propertyFile = (String)config.getInitParameter("propertyFile");
```

5-3 与 Input / Output 有关的隐含对象

本节中，我们将介绍和 Input/Output 有关的隐含对象，它们包括：`out`、`request` 和 `response` 对象。`request` 对象表示客户端请求的内容；`response` 对象表示响应客户端的结果；而 `out` 对象负责把数据的结果显示到客户端的浏览器。

request 对象

`request` 对象包含所有请求的信息，如：请求的来源、标头、`cookies` 和请求相关的参数值等等。在 JSP 网页中，`request` 对象是实现 `javax.servlet.http.HttpServletRequest` 接口的，`HttpServletRequest` 接口所提供的方法，可以将它分为四大类：

- (1) 在 5-1-3 小节提到的储存和取得属性方法；
- (2) 能够取得请求参数的方法，如表 5-4：

表 5-4 取得请求参数的方法

方 法	说 明
<code>String getParameter(String name)</code>	取得 <code>name</code> 的参数值
<code>Enumeration getParameterNames()</code>	取得所有的参数名称
<code>String [] getParameterValues(String name)</code>	取得所有 <code>name</code> 的参数值
<code>Map getParameterMap()</code>	取得一个要求参数的 Map

- (3) 能够取得请求 HTTP 标头的方法，如表 5-5：

表 5-5 取得请求标头的方法

方 法	说 明
<code>String getHeader(String name)</code>	取得 <code>name</code> 的标头
<code>Enumeration getHeaderNames()</code>	取得所有的标头名称
<code>Enumeration getHeaders(String name)</code>	取得所有 <code>name</code> 的标头
<code>int getIntHeader(String name)</code>	取得整数类型 <code>name</code> 的标头
<code>long getDateHeader(String name)</code>	取得日期类型 <code>name</code> 的标头
<code>Cookie [] getCookies()</code>	取得与请求有关的 <code>cookies</code>

- (4) 其他的方法，例如：取得请求的 URL、IP 和 `session`，如表 5-6：

表 5-6 其他请求的方法

方 法	说 明
<code>String getContextPath()</code>	取得 Context 路径(即站台名称)
<code>String getMethod()</code>	取得 HTTP 的方法(GET、POST)
<code>String getProtocol()</code>	取得使用的协议 (HTTP/1.1、HTTP/1.0)
<code>String getQueryString()</code>	取得请求的参数字符串，不过，HTTP 的方法必须为 GET
<code>String getRequestedSessionId()</code>	取得用户端的 Session ID
<code>String getRequestURL()</code>	取得请求的 URL，但是不包括请求的参数字符串
<code>String getRemoteAddr()</code>	取得用户的 IP 地址
<code>String getRemoteHost()</code>	取得用户的主机名称
<code>int getRemotePort()</code>	取得用户的主机端口

String getRemoteUser()	取得用户的名称
void setCharacterEncoding(String encoding)	设定编码格式，用来解决窗体传递中文的问题

我们来看下面这个程序范例，相信对读者会更加有帮助。

Request.html

```
<html>

<head>

  <title>CH5 - Request.html</title>

<meta http-equiv="Content-Type" content="text/html; charset=GB2312">

</head>

<body>

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

    Name: <input type="text" name="Name" size="20" maxlength="20"><br>

    Number: <input type="text" name="Number" size="20" maxlength="20"><br><br>

    <input type="submit" value="传送">

  </form>

</body>

</html>
```

Request.html 的执行结果如图 5-6 所示，笔者在 Name 的字段中输入 browser；Number 字段中输入 123456789。



图 5-6 Request.html 的执行结果

Request.jsp

```
<%@ page language="java" contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - Request.jsp</title>

</head>

<body>

<h2>javax.servlet.http.HttpServletRequest 接口所提供的方法</h2>

getParameter("Name"): <%= request.getParameter("Name") %><br>
getParameter("Number"): <%= request.getParameter("Number") %><br>
getAttribute("Name"): <%= request.getAttribute("Name") %><br>
getAttribute("Number"): <%= request.getAttribute("Number") %><br><br>
getAuthType(): <%= request.getAuthType() %><br>
```



```
getProtocol( ): <%= request.getProtocol() %><br>

getMethod( ): <%= request.getMethod() %><br>

getScheme( ): <%= request.getScheme() %><br>

getContentType( ): <%= request.getContentType() %><br>

getContentLength( ): <%= request.getContentLength() %><br>

getCharacterEncoding( ): <%= request.getCharacterEncoding() %><br>

getRequestedSessionId( ): <%= request.getRequestedSessionId() %><br><br>

getContextPath( ): <%= request.getContextPath() %><br>

getServletPath( ): <%= request.getServletPath() %><br>

getPathInfo( ): <%= request.getPathInfo() %><br>

getRequestURI( ): <%= request.getRequestURI() %><br>

getQueryString( ): <%= request.getQueryString() %><br><br>

getRemoteAddr( ): <%= request.getRemoteAddr() %><br>

getRemoteHost( ): <%= request.getRemoteHost() %><br>

getRemoteUser( ): <%= request.getRemoteUser() %><br>

getRemotePort( ): <%= request.getRemotePort() %><br>

getServerName( ): <%= request.getServerName() %><br>

getServerPort( ): <%= request.getServerPort() %><br>

</body>

</html>
```

Request.jsp 的执行结果如图 5-7 所示。

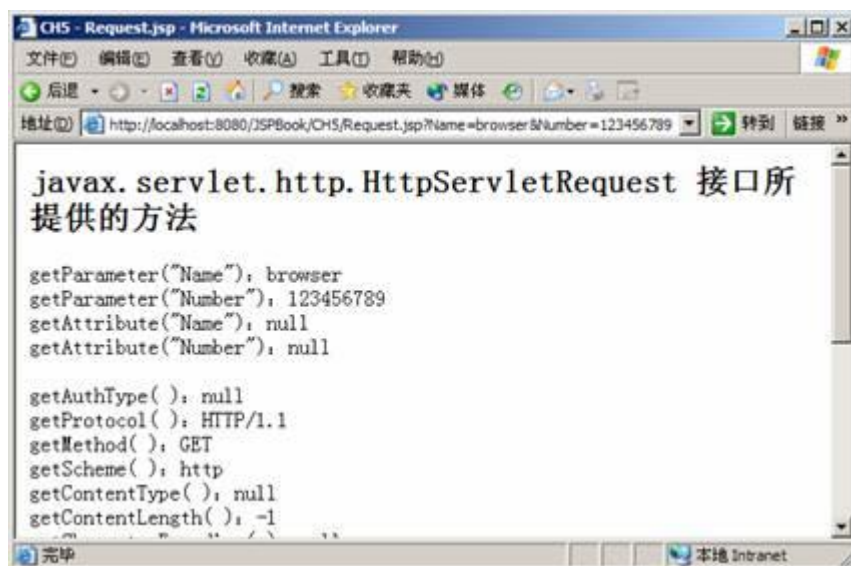


图 5-7 Request.jsp 的执行结果

在 Request.jsp 中, 使用 `request.getParameter("Name")` 和 `request.getParameter("Number")`, 能够取得 Request.html 窗体的值。除了取得请求的参数值之外, 也能取得一些相关信息, 如: 使用的协议、方法、URI 等等。

response 对象

response 对象主要将 JSP 处理数据后的结果传回到客户端。response 对象是实现 `javax.servlet.http.HttpServletResponse` 接口。表 5-7、表 5-8、表 5-9 列出了 response 对象的方法。

表 5-7 设定表头的方法

方 法	说 明
<code>void addCookie(Cookie cookie)</code>	新增 cookie
<code>void addDateHeader(String name, long date)</code>	新增 long 类型的值到 name 标头
<code>void addHeader(String name, String value)</code>	新增 String 类型的值到 name 标头
<code>void addIntHeader(String name, int value)</code>	新增 int 类型的值到 name 标头
<code>void setDateHeader(String name, long date)</code>	指定 long 类型的值到 name 标头
<code>void setHeader(String name, String value)</code>	指定 String 类型的值到 name 标头
<code>void setIntHeader(String name, int value)</code>	指定 int 类型的值到 name 标头

表 5-8 设定响应状态码的方法

方 法	说 明
<code>void sendError(int sc)</code>	传送状态码(status code)
<code>void sendError(int sc, String msg)</code>	传送状态码和错误信息
<code>void setStatus(int sc)</code>	设定状态码

表 5-9 用来 URL 重写(rewriting)的方法

方 法	说 明
<code>String encodeRedirectURL(String url)</code>	对使用 <code>sendRedirect()</code> 方法的 URL 予以编码

有时候,当我们修改程序后,产生的结果却是之前的数据,执行浏览器上的刷新,才能看到更改数据后的结果,针对这个问题,有时是因为浏览器会将之前浏览过的数据存放在浏览器的 **cache** 中,所以当我们再次执行时,浏览器会直接从 **cache** 中取出,因此,会显示之前旧的数据。笔者将写一个 `Non-cache.jsp` 程序来解决这个问题。

`Non-cache.jsp`

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

  <title>CH5 - Non-cache.jsp</title>

</head>

<body>

<h2>解决浏览器 cache 的问题 - response</h2>

<%

    if (request.getProtocol().compareTo("HTTP/1.0") == 0)

        response.setHeader("Pragma", "no-cache");

%>
```

```
else if (request.getProtocol().compareTo("HTTP/1.1") == 0)

    response.setHeader("Cache-Control", "no-cache");

response.setDateHeader("Expires", 0);

%>

</body>

</html>
```

先用 `request` 对象取得协议，如果为 HTTP/1.0，就设定标头，内容为 `setHeader ("Pragma", "no-cache")`；若为 HTTP/1.1，就设定标头为 `response.setHeader ("Cache-Control", "no-cache")`，最后再设定 `response.setDateHeader("Expires", 0)`。这样 `Non-cache.jsp` 网页在浏览过后，就不会再存放到浏览器或是 proxy 服务器的 cache 中。表 5-10 列出了 HTTP/1.1 Cache-Control 标头的设定参数：

表 5-10 HTTP/1.1 Cache-Control 标头的设定参数

参 数	说 明
public	数据内容皆被储存起来，就连有密码保护的网页也是一样，因此安全性相当低
private	数据内容只能被储存到私有的 caches，即 non-shared caches 中
no-cache	数据内容绝不被储存起来。proxy 服务器和浏览器读到此标头，就不会将数据内容存入 caches 中
no-store	数据内容除了不能存入 caches 中之外，亦不能存入暂时的磁盘中，这个标头防止敏感性的数据被复制
must-revalidate	用户在每次读取数据时，会再次和原来的服务器确定是否为最新数据，而不再通过中间的 proxy 服务器
proxy-revalidate	这个参数有点像 must-revalidate，不过中间接收的 proxy 服务器可以互相分享 caches
max-age=xxx	数据内容在经过 xxx 秒后，就会失效，这个标头就像 Expires 标头的功能一样，不过 max-age=xxx 只能服务 HTTP/1.1 的用户。假设两者并用时，max-age=xxx 有较高的优先权

有时候, 我们想要让网页自己能自动更新, 因此, 须使用到 **Refresh** 这个标头。举个例子, 我们告诉浏览器, 每隔三分钟, 就重新加载此网页:

```
response.setIntHeader("Refresh", 180)
```

如果想要过十秒后, 调用浏览器转到 <http://Server/Path> 的网页时, 可用如下代码:

```
response.setHeader("Refresh","10; URL=http://Server/Path" )
```

如果大家对 HTML 语法还熟悉, 则 HTML 语法中也有类似的功能:

```
<META HTTP-EQUIV="Refresh" CONTENT=" 10; URL=http://Server/Path" >
```

上述两种方法皆可以做到自动重新加载。

out 对象

out 对象能把结果输出到网页上。通常我们最常使用 `out.println(String name)`和 `out.print(String name)`, 它们两者最大的差别在于 `println()`在输出的数据后面会自动加上换行的符号, 例如: 你在 Dos Console 的窗口下, 发现到它输出数据后会自动换行; 反之, `print()`不会在数据后自动换行。

out 对象除了这两种方法最常使用之外, 它还有一些方法 (见表 5-11), 这些方法主要是用来控制管理输出的缓冲区(buffer)和输出流(output stream)。

表 5-11 out 对象方法

方 法	说 明
<code>void clear()</code>	清除输出缓冲区的内容
<code>void clearBuffer()</code>	清除输出缓冲区的内容
<code>void close()</code>	关闭输出流, 清除所有的内容
<code>int getBufferSize()</code>	取得目前缓冲区的大小(KB)
<code>int getRemaining()</code>	取得目前使用后还剩下的缓冲区大小(KB)
<code>boolean isAutoFlush()</code>	如果回传为 <code>true</code> , 表示如缓冲区满了, 会自动清除; 若为 <code>false</code> , 表示如果缓冲区满了, 不会自动清除, 而会产生异常处理

我们在这里举个例子, 说明如何知道目前输出缓冲区的大小。

Out.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```
<html>

<head>

  <title>CH5 - Out.jsp</title>

</head>

<body>

<h2>javax.servlet.jsp.JspWriter - out 对象</h2>

<%

    int BufferSize = out.getBufferSize();

    int Available = out.getRemaining();

    int Used = BufferSize - Available;

%>

BufferSize : <%= BufferSize %><br>

Available : <%= Available %><br>

Used : <%= Used %><br>

</body>

</html>
```

BufferSize 是一开始默认缓冲区的大小，默认值为 8KB；**Available** 则是表示经过程序的执行，目前缓冲区还剩下多少可以使用；而 **Used** 则表示我们使用了多少的缓冲区。**Out.jsp** 执行结果如图 5-8。



图 5-8 Out.jsp 的执行结果

5-4 与 Context 有关的隐含对象(1)

在本节中，我们要介绍 `session`、`application`、`pageContext` 这三个对象。`session` 对象提供一些机制，让服务器能个别辨认用户。当程序在执行时，`application` 对象能提供服务端(Server-Side)的 `Context`，说明哪些资源是可利用的，哪些信息是可获取的。`pageContext` 对象提供存取所有在此网页中可被利用的隐含对象，并且可以管理它们的属性。

session 对象

`session` 对象表示目前个别用户的会话(session)状况，用此项机制可以轻易识别每一个用户，然后针对每一个别用户的要求，给予正确的响应。例如：购物车最常使用 `session` 的概念，当用户把物品放入购物车时，他不须重复做身份确认的动作(如：Login)，就能把物品放入用户的购物车。但服务器利用 `session` 对象，就能确认用户是谁，把它的物品放在属于用户的购物车，而不会将物品放错到别人的购物车。除了购物车之外，`session` 对象也通常用来做追踪用户的功能，这在第十章有更加详细的说明。

`session` 对象实现 `javax.servlet.http.HttpSession` 接口，表 5-12 列出了一些常用的方法。

表 5-12 javax.servlet.http.HttpSession 接口所提供的方法

方 法	说 明
<code>long getCreationTime()</code>	取得 session 产生的时间,单位是毫秒,由 1970 年 1 月 1 日零时算起
<code>String getId()</code>	取得 session 的 ID

续表

方 法	说 明
<code>long getLastAccessedTime()</code>	取得用户最后通过这个 <code>session</code> 送出请求的时间，单位是毫秒，由 1970 年 1 月 1 日零时算起
<code>long getMaxInactiveInterval()</code>	取得最大 <code>session</code> 不活动的时间，若超过这时间， <code>session</code> 将会失效，时间单位为秒
<code>void invalidate()</code>	取消 <code>session</code> 对象，并将对象存放的内容完全抛弃
<code>boolean isNew()</code>	判断 <code>session</code> 是否为"新"的，所谓"新"的 <code>session</code> ，表示 <code>session</code> 已由服务器产生，但是 <code>client</code> 尚未使用
<code>void setMaxInactiveInterval(int interval)</code>	设定最大 <code>session</code> 不活动的时间，若超过这时间， <code>session</code> 将会失效，时间单位为秒

`session` 对象也可以储存或取得用户相关的数据，例如：用户的名称、用户所订购的物品、用户的权限，等等，这些要看我们的程序如何去设计。例如：我要设定某些网页必须要求用户先做登录(Login)的动作，确定是合法的用户时，才允许读取网页内容，否则把网页重新转向到登录的网页上。

Login.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - Login.jsp</title>

</head>

<body>

    <h2>javax.servlet.http.HttpSession - session 对象</h2>

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

        Login Name: <input type="text" name="Name"><br>

        Login Password: <input type="text" name="Password" ><br>
```



```
<input type="submit" value="Send"><br>

<form>

<% if (request.getParameter("Name") != null &&
request.getParameter("Password") != null) {

    String Name = request.getParameter("Name");

    String Password = request.getParameter("Password");

    if (Name.equals("mike") && Password.equals("1234")) {

        session.setAttribute("Login", "OK");

        response.sendRedirect("Member.jsp");

    }

    else {

        out.println("登录错误, 请输入正确名称");

    }

}

%>

</body>

</html>
```

在 Login.jsp 的程序中, 我要求用户分别输入名称和密码, 如果输入的名称和密码分别为 mike 和 1234 时, 就把名称为 Login、其值为 OK 的属性, 加入到 session 对象当中, 然后进入 Member.jsp 网页, 如图 5-9; 若输入错误时, 就显示出“登录错误, 请输入正确名称”。不允许登录至 Member.jsp, 如图 5-10 所示。



图 5-9 登录成功，顺利进入 Member.jsp

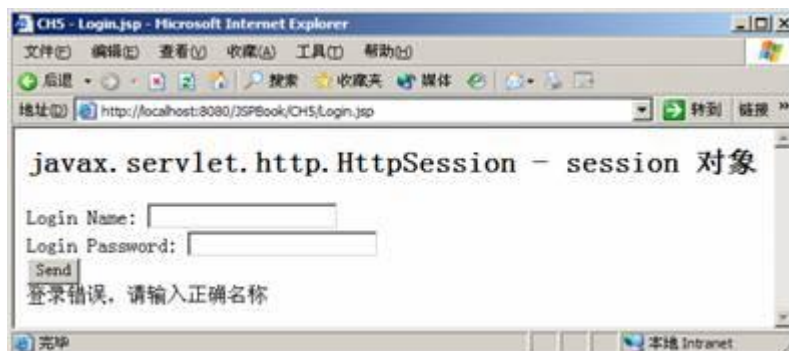


图 5-10 登录失败画面

这时大家一定会想，如果我不通过 Login.jsp 网页，直接执行 Member.jsp，那不就能够进去了。没错，因此我们还要在 Member.jsp 中加入一段程序代码，来确认用户是否有先通过 Login.jsp 的身份确认，然后再到 Member.jsp 中。Member.jsp 程序如下：

Member.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>

<head>

    <title>CH5 - Member.jsp</title>

</head>

<body>

<h2>javax.servlet.http.HttpSession - session 对象</h2>
```

```
<%  
  
    String Login = (String)session.getAttribute("Login");  
  
    if (Login != null && Login.equals("OK")) {  
  
        out.println("欢迎进入");  
  
        session.invalidate();  
  
    }  
  
    else {  
  
        out.println("请先登录，谢谢")    ;  
  
        out.println("<br>经过五秒之后，网页会自动返回 Login.jsp");  
  
  
        response.setHeader("Refresh","5;URL=Login.jsp");  
  
    }  
  
%>  
  
</body>  
  
</html>
```

在 Member.jsp 中我利用 session.getAttribute("Login"), 如果用户是通过 Login.jsp 网页进入, 并且顺利通过身份确认取得 Login=OK, 到 Member.jsp 再做确认时, 也能顺利通过; 否则, 如果直接连接到 Member.jsp 时, Login 的值会等于 NULL, 则程序经过五秒后, 重新加载 Login.jsp, 要求用户先行登录。若直接执行 Member.jsp, 而没有经过登录手续时, 就会发现如图 5-11。



图 5-11 直接执行 Member.jsp, 并未经过登录手续

最后要提醒读者一点, `session` 对象不像其他的隐含对象, 可以在任何的 JSP 网页中使用, 如果在 JSP 网页中, `page` 指令的属性 `session` 设为 `false` 时, 使用 `session` 对象就会产生编译错误 (`javax.servlet.ServletException: Compilation error occurred`), 如下所示:

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

<%

    String Login = (String)session.getAttribute("Login");

    ...

    ...

%>
```

在本书“第十章: Session Tracking”中对 `session` 有更多更详细的介绍。

application 对象

`application` 对象实现 `javax.servlet.ServletContext` 接口, 它主要功用在于取得或更改 `Servlet` 的设定。下面程序用来说明 JSP 网页被编译成 `Servlet` 时, `application` 对象是如何初始化的:

```
pageContext = JspFactory.getPageContext ( this , request , response ,
                                           "errorpage.jsp" , true , 8192 , true );

application = pageContext.getServletContext( );
```

你可以看到产生的 `Servlet` 取得了目前的 `ServletContext`，并且将它储存在 `application` 对象当中。`application` 对象拥有 `Application` 的范围，意思就是说它的生命周期是由服务器产生开始至服务器关机为止。表 5-13、表 5-14、表 5-15 列出了其相关方法：

表 5-13 `javax.servlet.ServletContext` 接口容器相关信息的方法

方 法	说 明
<code>int getMajorVersion()</code>	取得 Container 主要的 Servlet API 版本，如：2
<code>int getMinorVersion()</code>	取得 Container 次要的 Servlet API 版本，如：4
<code>String getServerInfo()</code>	取得 Container 的名称和版本

```
<%= application.getMajorVersion() %><br>
```

```
<%= application.getMinorVersion() %><br>
```

```
<%= application.getServerInfo() %><br>
```

上述的 `getMajorVersion()` 和 `getMinorVersion()` 是取得 Servlet Engine 的版本信息，假如想要取得 JSP 容器的版本信息，则可能就要使用到下面这段程序代码：

GetJspVersion.jsp

```
<%@ page import="javax.servlet.jsp.JspFactory"
contentType="text/html; charset=GB2312" %>
<html>
<head>
  <title>CH5 - GetJspVersion.jsp</title>
</head>
<body>
<h2>取得 JSP Container 版本 - JspFactory 对象</h2>
<%
    JspFactory factory = JspFactory.getDefaultFactory();
    out.println("JSP v 2.0"+
               factory.getEngineInfo().getSpecificationVersion());
%>

</body>
</html>
```

执行结果如图 5-12 所示。

表 5-14 javax.servlet.ServletContext 接口有关服务端的路径和文件的方法

方 法	说 明
String getMimeType(String file)	取得指定文件的 MIME 类型
ServletContext getContext(String uripath)	取得指定 Local URL 的 Application context
String getRealPath(String path)	取得本地端 path 的绝对路径

范例：

```
<%= application.getMimeType("MyFile") %>  
  
<%= application.getContext("/") %>  
  
<%= application.getRealPath("/") %>
```

表 5-15 javax.servlet.ServletContext 接口有关信息记录的方法

方 法	说 明
void log(String message)	将信息写入 log 文件中
void log(String message, Throwable throwable)	将 stack trace 所产生的异常信息写入 log 文件中

application 对象最常被使用在存取环境的信息,因为环境的信息通常都储存在 ServletContext 中,所以常利用 application 对象来存取 ServletContext 中的信息。



图 5-12 GetJspVersion.jsp 的执行结果

pageContext 对象

pageContext 对象能够存取其他隐含对象。当隐含对象本身也支持属性时, pageContext 对象也提供存取那些属性的方法。不过在使用下列方法时,需要指定范围的参数:

```
Object getAttribute(String name, int scope)

Enumeration getAttributeNamesInScope(int scope)

void removeAttribute(String name, int scope)

void setAttribute(String name, Object value, int scope)
```

范围参数有四个常数，分别代表四种范围：PAGE_SCOPE 代表 Page 范围，REQUEST_SCOPE 代表 Request 范围，SESSION_SCOPE 代表 Session 范围，最后 APPLICATION_SCOPE 代表 Application 范围（见表 5-16、表 5-17、表 5-18）。

表 5-16 javax.servlet.jsp.PageContext 类取得其他隐含对象的方法

方 法	说 明
Exception getException()	回传目前网页的异常，不过此网页要为 error page，例如：exception 隐含对象
JspWriter getOut()	回传目前网页的输出流，例如：out 隐含对象
Object getPage()	回传目前网页的 Servlet 实体(instance)，例如：page 隐含对象
ServletRequest getRequest()	回传目前网页的请求，例如：request 隐含对象
ServletResponse getResponse()	回传目前网页的响应，例如：response 隐含对象
ServletConfig getServletConfig()	回传目前此网页的 ServletConfig 对象，例如：config 隐含对象
ServletContext getServletContext()	回传目前此网页的执行环境(context)，例如：application 隐含对象
HttpSession getSession()	回传和目前网页有联系的会话(session)，例如：session 隐含对象

表 5-17 javax.servlet.jsp.PageContext 类所提供取得属性的方法

方 法	说 明
Object getAttribute(String name, int scope)	回传 name 属性，范围为 scope 的属性对象，回传类型为 java.lang.Object
Enumeration getAttributeNamesInScope(int scope)	回传所有属性范围为 scope 的属性名称，回传类型为 Enumeration
int getAttributesScope(String name)	回传属性名称为 name 的属性范围
void removeAttribute(String name)	移除属性名称为 name 的属性对象
void removeAttribute(String name, int scope)	移除属性名称为 name，范围为 scope 的属性对象

<code>void setAttribute(String name, Object value, int scope)</code>	指定属性对象的名称为 <code>name</code> 、值为 <code>value</code> 、范围为 <code>scope</code>
<code>Object findAttribute(String name)</code>	寻找在所有范围中属性名称为 <code>name</code> 的属性对象

表 5-18 `javax.servlet.jsp.PageContext` 类所提供范围的变量

常 数	说 明
<code>PAGE_SCOPE</code>	存入 <code>pageContext</code> 对象的属性范围
<code>REQUEST_SCOPE</code>	存入 <code>request</code> 对象的属性范围
<code>SESSION_SCOPE</code>	存入 <code>session</code> 对象的属性范围
<code>APPLICATION_SCOPE</code>	存入 <code>application</code> 对象的属性范围

接下来示范一个小程序，让读者能够更加明白。

`PageContext.jsp`

```
<%@ page import="java.util.Enumeration"
contentType="text/html; charset=GB2312" %>

<html>

<head>

  <title>CH5 - PageContext.jsp</title>

</head>

<body>

<h2>javax.servlet.jsp.PageContext - pageContext </h2>

<%

  Enumeration enum =
    pageContext.getAttributeNamesInScope(PageContext.APPLICATION_SCOPE );

  while (enum.hasMoreElements())

  {

    out.println("application attribute: "+enum.nextElement( )
+"<br>");
  }

%>
```



```
%>

</body>

</html>
```

`PageContext.jsp` 主要目的是：在这页当中，取得所有属性范围为 `Application` 的属性名称，然后再依序显示出来这些属性。

首先要记得导入 `java.util.Enumeration`。`pageContext.getAttributeNamesInScope()` 会回传所有指定范围的属性名称，因此，我们产生 `Enumeration` 对象 `enum`，利用 `enum` 来收集所有属性范围为 `Application` 的数据，然后再一一地取出打印出来。这里最重要的是让读者了解如何设定 `scope` 的参数，因此下面这行代码：

```
PageContext.APPLICATION_SCOPE
```

是最主要的。有了这个范例程序之后，读者应该能够快速学会使用 `pageContext` 对象所提供的方法。

`pageContext` 对象除了提供上述的方法之外，另外还有两种方法：`forward (String Path)`、`include (String Path)`，这两种方法的功能和之前提到的 `<jsp:forward>` 与 `<jsp:include>` 相似，因此在这也不多加讨论。

5-5 与 Error 有关的隐含对象

最后一类的隐含对象只有一个成员：`exception` 对象。当 JSP 网页有错误时会产生异常，而 `exception` 对象就来针对这个异常做处理。

`exception` 对象

`exception` 对象和 `session` 对象一样，并不是在每一个 JSP 网页中都能够使用。若要使用 `exception` 对象时，必须在 `page` 指令中设定。

```
<%@ page isErrorPage="true" %>
```

才能使用，不然在编译时会产生错误。

Exception.jsp

```
<%@ page contentType="text/html; charset=GB2312" isErrorPage="true" %>

<html>

<head>

    <title>CH5 - Exception.jsp</title>

</head>

<body>

<h2> exception 对象</h2>

Exception: <%= exception %><br>

Message: <%= exception.getMessage() %><br>

Localized Message: <%= exception.getMessage() %><br>

Stack Trace: <% exception.printStackTrace(new java.io.PrintWriter(out));
%><br>

</body>

</html>
```

一般 error page 的程序代码和 Exception.jsp 程序相似，它已经将所有该打印出来的错误信息包括进来。在这段程序代码中使用了三个方法：`getMessage()`、`getMessage()`、`printStackTrace(new java.io.PrintWriter(out))`，其中 `printStackTrace()` 的参数要为 `PrintWriter` 而不是 `JspWriter`。

第六章 Expression Language

6-1 EL 简介

6-1 EL 简介

EL 全名为 Expression Language，它原本是 JSTL 1.0 为方便存取数据所自定义的语言。当时 EL 只能在 JSTL 标签中使用，如下：

```
<c:out value="${ 3 + 7}">
```

程序执行结果为 10。但是你却不能直接在 JSP 网页中使用：

```
<p>Hi ! ${ username }</p>
```

到了 JSP 2.0 之后，EL 已经正式纳入成为标准规范之一。因此，只要是支持 Servlet 2.4 / JSP 2.0 的 Container，就都可以在 JSP 网页中直接使用 EL 了。

除了 JSP 2.0 建议使用 EL 之外，JavaServer Faces(JSR-127) 也考虑将 EL 纳入规范，由此可知，EL 如今已经是一项成熟、标准的技术。

注意

假若您所用的 Container 只支持 Servlet 2.3/JSP 1.2，如：Tomcat 4.1.29，您就不能在 JSP 网页中直接使用 EL，必须安装支持 Servlet 2.4 / JSP 2.0 的 Container。

6-2 EL 语法

EL 语法很简单，它最大的特点就是使用上很方便。接下来介绍 EL 主要的语法结构：

```
${sessionScope.user.sex}
```

所有 EL 都是以 \${ 为起始、以 } 为结尾的。上述 EL 范例的意思是：从 Session 的范围中，取得用户的性别。假若依照之前 JSP Scriptlet 的写法如下：

```
User user = (User)session.getAttribute("user");  
String sex = user.getSex( );
```

两者相比较之下，可以发现 EL 的语法比传统 JSP Scriptlet 更为方便、简洁。

6-2-1 . 与 [] 运算符

EL 提供 . 和 [] 两种运算符来存取数据。下列两者所代表的意思是一样的：

```
${sessionScope.user.sex}
```

等于

```
${sessionScope.user["sex"]}
```

. 和 [] 也可以同时混合使用，如下：

```
${sessionScope.shoppingCart[0].price}
```

回传结果为 shoppingCart 中第一项物品的价格。

不过，以下两种情况，两者会有差异：

(1) 当要存取的属性名称中包含一些特殊字符，如 . 或 - 等并非字母或数字的符号，就一定要使用 []，例如：

```
${user.My-Name }
```

上述是不正确的方式，应当改为：

```
${user["My-Name" ] }
```

(2) 我们来考虑下列情况：

```
${sessionScope.user[data]}
```

此时，data 是一个变量，假若 data 的值为"sex"时，那上述的例子等于\${sessionScope.user.sex}；假若 data 的值为"name"时，它就等于\${sessionScope.user.name}。因此，如果要动态取值时，就可以用上述的方法来做，但 . 无法做到动态取值。

接下来，我们更详细地来讨论一些情况，首先假设有一个 EL：

```
${expr-a[expr-b]}
```

(1) 当 `expr-a` 的值为 `null` 时，它会回传 `null`。

(2) 当 `expr-b` 的值为 `null` 时，它会回传 `null`。

(3) 当 `expr-a` 的值为 `Map` 类型时：

- 假若 `!value-a.containsKey(value-b)` 为真，则回传 `null`。

- 否则回传 `value-a.get(value-b)`。

(4) 当 `expr-a` 的值为 `List` 或 `array` 类型时：

- 将 `value-b` 的值强制转型为 `int`，假若不能转型为 `int` 时，会产生 `error`。

- 然后，假若 `value-a.get(value-b)` 或 `Array.get(value-a, value-b)` 产生 `ArrayIndexOutOfBoundsException` 或 `IndexOutOfBoundsException` 时，则回传 `null`。

- 假若 `value-a.get(value-b)` 或 `Array.get(value-a, value-b)` 产生其他的异常时，则会产生 `error`。

- 最后都没有任何异常产生时，回传 `value-a.get(value-b)` 或 `Array.get(value-a, value-b)`。

(5) 当 `expr-a` 的值为 `JavaBean` 对象时：

- 将 `value-b` 的值强制转型为 `String`。

- 假若 `getter` 产生异常时，则会产生 `error`。若没有异常产生时，则回传 `getter` 的结果。

6-2-2 EL 变量

EL 存取变量数据的方法很简单，例如：`${username}`。它的意思是取出某一范围中名称为 `username` 的变量。因为我们并没有指定哪一个范围的 `username`，所以它的默认值会先从 `Page` 范围找，假如找不到，再依序到 `Request`、`Session`、`Application` 范围。假如途中找到 `username`，就直接回传，不再继续找下去，但是假如全部的范围都没有找到时，就回传 `null`（见表 6-1）：

表 6-1

属性范围	在 EL 中的名称
Page	PageScope
Request	RequestScope
Session	SessionScope
Application	ApplicationScope

自动搜索顺序

我们也可以指定要取出哪一个范围的变量（见表 6-2）：

表 6-2

范 例	说 明
<code>\${pageScope.username}</code>	取出 Page 范围的 username 变量
<code>\${requestScope.username}</code>	取出 Request 范围的 username 变量
<code>\${sessionScope.username}</code>	取出 Session 范围的 username 变量
<code>\${applicationScope.username}</code>	取出 Application 范围的 username 变量

其中，`pageScope`、`requestScope`、`sessionScope` 和 `applicationScope` 都是 EL 的隐含对象，由它们的名称可以很容易猜出它们所代表的意思，例如：`${sessionScope.username}` 是取出 Session 范围的 username 变量。这种写法是不是比之前 JSP 的写法：

```
String username = (String) session.getAttribute("username");
```

容易、简洁许多。有关 EL 隐含对象在 6-3 节中有更详细的介绍。

6-2-3 自动转变类型

EL 除了提供方便存取变量的语法之外，它另外一个方便的功能就是：自动转变类型，我们来看下面这个范例：

```
${param.count + 20}
```

假若窗体传来 `count` 的值为 10 时，那么上面的结果为 30。之前没接触过 JSP 的读者可能会认为上面的例子是理所当然的，但是在 JSP 1.2 之中不能这样做，原因是从窗体所传来的值，它们的类型一律是 `String`，所以当你接收之后，必须再将它转为其他类型，如：`int`、`float` 等等，然后才能执行一些数学运算，下面是之前的做法：

```
String str_count = request.getParameter("count");
```

```
int count = Integer.parseInt(str_count);  
  
count = count + 20;
```

接下来再详细说明 EL 类型转换的规则:

(1) 将 A 转为 String 类型

- 假若 A 为 String 时: 回传 A
- 否则, 当 A 为 null 时: 回传 ""
- 否则, 当 A.toString() 产生异常时: 错误!
- 否则, 回传 A.toString()

(2) 将 A 转为 Number 类型的 N

- 假若 A 为 null 或 "" 时: 回传 0
- 假若 A 为 Character 时: 将 A 转为 new Short((short)a.charValue())
- 假若 A 为 Boolean 时: 错误!
- 假若 A 为 Number 类型和 N 一样时: 回传 A
- 假若 A 为 Number 时:
 - 假若 N 是 BigInteger 时:
 - 假若 A 为 BigDecimal 时: 回传 A.toBigInteger()
 - 否则, 回传 BigInteger.valueOf(A.longValue())
 - 假若 N 是 BigDecimal 时:
 - 假若 A 为 BigInteger 时: 回传 A.toBigDecimal()
 - 否则, 回传 BigDecimal.valueOf(A.doubleValue())
 - 假若 N 为 Byte 时: 回传 new Byte(A.byteValue())
 - 假若 N 为 Short 时: 回传 new Short(A.shortValue())
 - 假若 N 为 Integer 时: 回传 new Integer(A.intValue())

- 假若 N 为 Long 时: 回传 `new Long(A.longValue())`
- 假若 N 为 Float 时: 回传 `new Float(A.floatValue())`
- 假若 N 为 Double 时: 回传 `new Double(A.doubleValue())`
- 否则, 错误!

● 假若 A 为 String 时:

- 假若 N 是 BigDecimal 时:
 - 假若 `new BigDecimal(A)` 产生异常时: 错误!
 - 否则, 回传 `new BigDecimal(A)`
- 假若 N 是 BigInteger 时:
 - 假若 `new BigInteger(A)` 产生异常时: 错误!
 - 否则, 回传 `new BigInteger(A)`
- 假若 `N.valueOf(A)` 产生异常时: 错误!
- 否则, 回传 `N.valueOf(A)`

● 否则, 错误!

(3) 将 A 转为 Character 类型

- 假若 A 为 null 或 "" 时: 回传 `(char)0`
- 假若 A 为 Character 时: 回传 A
- 假若 A 为 Boolean 时: 错误!
- 假若 A 为 Number 时: 转换为 Short 后, 然后回传 Character
- 假若 A 为 String 时: 回传 `A.charAt(0)`
- 否则, 错误!

(4) 将 A 转为 Boolean 类型

- 假若 A 为 null 或 "" 时：回传 false
- 否则，假若 A 为 Boolean 时：回传 A
- 否则，假若 A 为 String，且 Boolean.valueOf(A) 没有产生异常时：回传 Boolean.valueOf(A)
- 否则，错误！

6-2-4 EL 保留字

EL 的保留字如表 6-3:

表 6-3

And	eq	gt	true
Or	ne	le	false
No	lt	ge	null
instanceof	empty	div	mod

所谓保留字的意思是指变量在命名时，应该避开上述的名字，以免程序编译时发生错误。

6-3 EL 隐含对象

笔者在“第五章：隐含对象（Implicit Object）”中，曾经介绍过 9 个 JSP 隐含对象，而 EL 本身也有自己的隐含对象。EL 隐含对象总共有 11 个（见表 6-4）:

表 6-4

隐含对象	类 型	说 明
PageContext	javax.servlet.ServletContext	表示此 JSP 的 PageContext
PageScope	java.util.Map	取得 Page 范围的属性名称所对应的值
RequestScope	java.util.Map	取得 Request 范围的属性名称所对应的值
sessionScope	java.util.Map	取得 Session 范围的属性名称所对应的值
applicationScope	java.util.Map	取得 Application 范围的属性名称所对应的值
param	java.util.Map	如同 ServletRequest.getParameter(String name)。回传 String 类型的值

续表

隐含对象	类 型	说 明
paramValues	java.util.Map	如同

隐含对象	类 型	说 明
		<code>ServletRequest.getParameterValues(String name)</code> 。回传 <code>String []</code> 类型的值
<code>header</code>	<code>java.util.Map</code>	如同 <code>ServletRequest.getHeader(String name)</code> 。回传 <code>String</code> 类型的值
<code>headerValues</code>	<code>java.util.Map</code>	如同 <code>ServletRequest.getHeaders(String name)</code> 。回传 <code>String []</code> 类型的值
<code>cookie</code>	<code>java.util.Map</code>	如同 <code>HttpServletRequest.getCookies()</code>
<code>initParam</code>	<code>java.util.Map</code>	如同 <code>ServletContext.getInitParameter(String name)</code> 。回传 <code>String</code> 类型的值

这 11 个隐含对象(Implicit Object)，笔者将它分成三类：

1. 与范围有关的隐含对象

`applicationScope`

`sessionScope`

`requestScope`

`pageScope`

2. 与输入有关的隐含对象

`param`

`paramValues`

3. 其他隐含对象

`cookie`

`header`

`headerValues`

`initParam`

`pageContext`

接下来笔者会依照上面的分类顺序，为读者介绍这些隐含对象。

6-3-1 属性(Attribute)与范围(Scope)

与范围有关的 EL 隐含对象包含以下四个：pageScope、requestScope、sessionScope 和 applicationScope，它们基本上就和 JSP 的 pageContext、request、session 和 application 一样，所以笔者在这里只稍略说明。不过必须注意的是，这四个隐含对象只能用来取得范围属性值，即 JSP 中的 `getAttribute(String name)`，却不能取得其他相关信息，例如：JSP 中的 request 对象除可以存取属性之外，还可以取得用户的请求参数或表头信息等等。但是在 EL 中，它就只能单纯用来取得对应范围的属性值，例如：我们要在 session 中储存一个属性，它的名称为 username，在 JSP 中使用 `session.getAttribute("username")` 来取得 username 的值，但是在 EL 中，则是使用 `${sessionScope.username}` 来取得其值的。接下来分别对这四个隐含对象做简短的说明：

● pageScope

范围和 JSP 的 Page 相同，也就是单单一页 JSP Page 的范围(Scope)。

● requestScope

范围和 JSP 的 Request 相同，requestScope 的范围是指从一个 JSP 网页请求到另一个 JSP 网页请求之间，随后此属性就会失效。

● sessionScope

范围和 JSP Scope 中的 session 相同，它的属性范围就是用户持续在服务器连接的时间。

● applicationScope

范围和 JSP Scope 中的 application 相同，它的属性范围是从服务器一开始执行服务，到服务器关闭为止。

6-3-2 与输入有关的隐含对象

与输入有关的隐含对象有两个：param 和 paramValues，它们是 EL 中比较特别的隐含对象。一般而言，我们在取得用户的请求参数时，可以利用下列方法：

```
request.getParameter(String name)
request.getParameterValues(String name)
```

在 EL 中则可以使用 param 和 paramValues 两者来取得数据。

```
${param.name}
```

```
${paramValues.name}
```

这里 `param` 的功能和 `request.getParameter(String name)` 相同，而 `paramValues` 和 `request.getParameterValues(String name)` 相同。如果用户填了一个表格，表格名称为 `username`，我们就可以使用 `${param.username}` 来取得用户填入的值。

为了让读者更加了解 `param` 和 `paramValues` 隐含对象的使用，再来看下面这个范例。此范例共有两个文件，分别为给用户输入值用的 `Param.html` 和显示出用户所传之值的 `Param.jsp`。

`Param.html`

```
<html>

<head>

  <title>CH6 - Param.html</title>

</head>

<body>

<h2>EL 隐含对象 param、paramValues</h2>

<form method = "post" action = "Param.jsp">

<p>姓名: <input type="text" name="username" size="15" /></p>

<p>密码: <input type="password" name="password" size="15" /></p>

<p>性别: <input type="radio" name="sex" value="Male" checked/> 男

        <input type="radio" name="sex" value="Female" /> 女</p>

<p>年龄:
```

```
<select name="old">

<option value="10">10 - 20</option>

<option value="20" selected>20 - 30</option>

<option value="30">30 - 40</option>

<option value="40">40 - 50</option>

</select>

</p>

<p>兴趣:

<input type="checkbox" name="habit" value="Reading"/>看书

<input type="checkbox" name="habit" value="Game"/>玩游戏

<input type="checkbox" name="habit" value="Travel"/>旅游

<input type="checkbox" name="habit" value="Music"/>听音乐

<input type="checkbox" name="habit" value="Tv"/>看电视

</p>

<p>

<input type="submit" value="传送"/>

<input type="reset" value="清除"/>

</p>

</form>

</body>

</html>
```

Param.html 的执行结果如图 6-1 所示。当我们把窗体填好后按下传送钮，它将会把信息传送到 Param.jsp 做处理。



图 6-1 Param.html 的执行结果，并填入信息

接下来，Param.jsp 接收由 Param.html 传来的信息，并且将它显示出来：

Param.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

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

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

<html>

<head>

    <title>CH6 - Param.jsp</title>

</head>

<body>

    <h2>EL 隐含对象 param、paramValues</h2>
```

```
<fmt:requestEncoding value="GB2312" />

姓名:  ${param.username}</br>

密码:  ${param.password}</br>

性别:  ${param.sex}</br>

年龄:  ${param.old}</br>

兴趣:  ${paramValues.habit[0]}

      ${paramValues.habit[1]}

</body>

</html>
```

由 Param.html 窗体传过来的值，我们必须指定编码方式，才能够确保 Param.jsp 能够顺利接收中文，传统的做法为：

```
<%

    request.setCharacterEncoding("GB2312");

%>
```

假若是使用 JSTL 写法时，必须使用 I18N 格式处理的标签库，如下：

```
<fmt:requestEncoding value="GB2312" />
```

Param.jsp 主要使用 EL 的隐含对象 param 来接收数据。但是必须注意：假若要取得多重选择的复选框的值时，必须使用 paramValues，例如：使用 paramValues 来取得“兴趣”的值，不过这里笔者最多只显示两笔“兴趣”的值：

```
${param.username}

.....

${paramValues.habit[0]}

${paramValues.habit[1]}
```

有关 JSTL 的使用，第七章有更加详细的说明。图 6-2 是 Param.jsp 的执行结果：

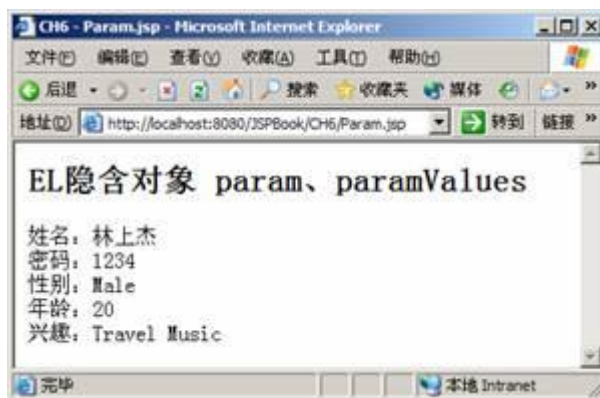


图 6-2 Param.jsp 的执行结果

6-3-3 其他隐含对象

介绍完上面六个隐含对象后，接下来将介绍最后五个隐含对象。

● cookie

所谓的 **cookie** 是一个小小的文本文件，它是以 **key**、**value** 的方式将 **Session Tracking** 的内容记录在这个文本文件内，这个文本文件通常存在于浏览器的暂存区内。**JSTL** 并没有提供设定 **cookie** 的动作，因为这个动作通常都是后端开发者必须去做的事情，而不是交给前端的开发者。假若我们在 **cookie** 中设定一个名称为 **userCountry** 的值，那么可以使用 `${cookie.userCountry}` 来取得它。

● header 和 headerValues

header 储存用户浏览器和服务端用来沟通的数据，当用户要求服务端的网页时，会送出一个记载要求信息的标头文件，例如：用户浏览器的版本、用户计算机所设定的区域等其他相关数据。假若要取得用户浏览器的版本，即 `${header["User-Agent"]}`。另外在鲜少机会下，有可能同一标头名称拥有不同的值，此时必须改为使用 **headerValues** 来取得这些值。

注意

因为 **User-Agent** 中包含 “-” 这个特殊字符，所以必须使用 “[]”，而不能写成 `$(header.User-Agent)`。

● initParam

就像其他属性一样，我们可以自行设定 **web** 站台的环境参数 (**Context**)，当我们想取得这些参数

时，可以使用 `initParam` 隐含对象去取得它，例如：当我们在 `web.xml` 中设定如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"

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

    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"

    version="2.4">

:

    <context-param>

        <param-name>userid</param-name>

        <param-value>mike</param-value>

    </context-param>

:

</web-app>
```

那么我们就可以直接使用 `${initParam.userid}` 来取得名称为 `userid`，其值为 `mike` 的参数。下面是之前的做法：

```
String userid = (String)application.getInitParameter("userid");
```

● `pageContext`

我们可以使用 `${pageContext}` 来取得其他有关用户要求或页面的详细信息。表 6-5 列出了几个比较常用的部分。

表 6-5

Expression	说 明
<code>\${pageContext.request.queryString}</code>	取得请求的参数字符串
<code>\${pageContext.request.requestURL}</code>	取得请求的 URL，但不包括请求之参数字符串
<code>\${pageContext.request.contextPath}</code>	服务的 web application 的名称

<code>\${pageContext.request.method}</code>	取得 HTTP 的方法(GET、POST)
<code>\${pageContext.request.protocol}</code>	取得使用的协议(HTTP/1.1、HTTP/1.0)
<code>\${pageContext.request.remoteUser}</code>	取得用户名称
<code>\${pageContext.request.remoteAddr }</code>	取得用户的 IP 地址
<code>\${pageContext.session.new}</code>	判断 session 是否为新的, 所谓新的 session, 表示刚由 server 产生而 client 尚未使用
<code>\${pageContext.session.id}</code>	取得 session 的 ID
<code>\${pageContext.servletContext.serverInfo}</code>	取得主机端的服务信息

我们来看下面这个范例: `pageContext.jsp`, 相信对读者来说能更加了解 `pageContext` 的用法。

`pageContext.jsp`

```
<%@ page contentType="text/html; charset=GB2312" %>

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

<html>

<head>

    <title>CH6 - pageContext.jsp</title>

</head>

<body>

<h2>EL 隐含对象 pageContext</h2>

\${pageContext.request.queryString}:${pageContext.request.queryString}</br>

\${pageContext.request.requestURL}:${pageContext.request.requestURL}</br>

\${pageContext.request.contextPath}:${pageContext.request.contextPath}</br>

\${pageContext.request.method}:${pageContext.request.method}</br>

\${pageContext.request.protocol}:${pageContext.request.protocol}</br>
```

```
\${pageContext.request.remoteUser}:${pageContext.request.remoteUser}</br>

\${pageContext.request.remoteAddr }:${pageContext.request.remoteAddr}</br>

\${pageContext.session.new}:${pageContext.session.new}</br>

\${pageContext.session.id}:${pageContext.session.id}</br>

</body>

</html>
```

pageContext.jsp 的执行结果如图 6-3，执行时必须在 pageContext.jsp 之后加上?test=1234，即 PageContext.jsp?test=1234，这样`\${pageContext.request.queryString}`才会显示 test=1234。



图 6-3 pageContext.jsp 的执行结果

注意

因为 `\${}` 在 JSP 2.0 中是特殊字符，JSP 容器会自动将它当做 EL 来执行，因此，假若要显示 `\${}` 时，必须在 `$` 前加上 `\`，如：`\${ XXXXX }`

6-4 EL 算术运算符

EL 算术运算符主要有以下五个（见表 6-6）：

表 6-6

算术运算符	说 明	范 例	结 果
+	加	<code>\\${ 17 + 5 }</code>	22
-	减	<code>\\${ 17 - 5 }</code>	12

*	乘	<code>\${ 17 * 5 }</code>	85
/ 或 div	除	<code>\${ 17 / 5 }</code> 或 <code>\${ 17 div 5 }</code>	3
% 或 mod	余数	<code>\${ 17 % 5 }</code> 或 <code>\${ 17 mod 5 }</code>	2

接下来，我们依照下列几种情况，详细说明 EL 算术运算符的规则：

(1) `A {+ , - , *}` B

- 假若 A 和 B 为 `null`：回传 (Long)0
- 假若 A 或 B 为 `BigDecimal` 时，将另一个也转为 `BigDecimal`，则：
 - 假若运算符为 `+` 时：回传 `A.add(B)`
 - 假若运算符为 `-` 时：回传 `A.subtract(B)`
 - 假若运算符为 `*` 时：回传 `A.multiply(B)`
- 假若 A 或 B 为 `Float`、`Double` 或包含 `e / E` 的字符串时：
 - 假若 A 或 B 为 `BigInteger` 时，将另一个转为 `BigDecimal`，然后依照运算符执行运算
 - 否则，将两者皆转为 `Double`，然后依照运算符执行运算
- 假若 A 或 B 为 `BigInteger` 时，将另一个也转为 `BigInteger`，则：
 - 假若运算符为 `+` 时：回传 `A.add(B)`
 - 假若运算符为 `-` 时：回传 `A.subtract(B)`
 - 假若运算符为 `*` 时：回传 `A.multiply(B)`
- 否则，将 A 和 B 皆转为 `Long`，然后依照运算符执行运算
- 假若运算结果产生异常时，则错误！

(2) `A {/ , div}` B

- 假若 A 和 B 为 `null`：回传 (Long)0
- 假若 A 或 B 为 `BigDecimal` 或 `BigInteger` 时，皆转为 `BigDecimal`，然后回传 `A.divide(B, BigDecimal.ROUND_HALF_UP)`

- 否则，将 A 和 B 皆转为 Double，然后依照运算符执行运算
- 假若运算结果产生异常时，则错误！

(3) A {% , mod} B

- 假若 A 和 B 为 null：回传 (Long)0
- 假若 A 或 B 为 BigDecimal、Float、Double 或包含 e / E 的字符串时，皆转为 Double，然后依照运算符执行运算
- 假若 A 或 B 为 BigInteger 时，将另一个转为 BigInteger，则回传 A.reminder(B)
- 否则，将 A 和 B 皆转为 Long，然后依照运算符执行运算
- 假若运算结果产生异常时，则错误！

(4) -A

- 假若 A 为 null：回传 (Long)0
- 假若 A 为 BigDecimal 或 BigInteger 时，回传 A.negate()
- 假若 A 为 String 时：
 - 假若 A 包含 e / E 时，将转为 Double，然后依照运算符执行运算
 - 否则，转为 Long，然后依照运算符执行运算
 - 假若运算结果产生异常时，则错误！
- 假若 A 为 Byte、Short、Integer、Long、Float 或 Double
 - 直接依原本类型执行运算
 - 假若运算结果产生异常时，则 错误！
- 否则，错误！

Tomcat 上的 jsp-examples 中，有一个 EL 算术运算符的范例 basic-arithmetic.jsp。它的程序很简单，所以不在这里多做说明，它的执行结果如图 6-4 所示。

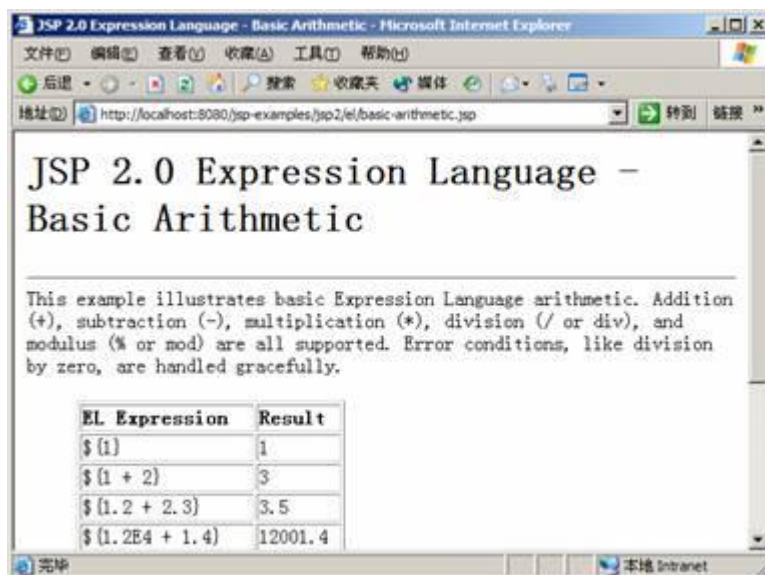


图 6-4 basic-arithmetic.jsp 的执行结果

6-5 EL 关系运算符

EL 关系运算符有以下六个运算符（见表 6-7）：

表 6-7

关系运算符	说 明	范 例	结 果
<code>=</code> 或 <code>eq</code>	等于	<code>\${ 5 == 5 }</code> 或 <code>\${ 5 eq 5 }</code>	true
<code>!=</code> 或 <code>ne</code>	不等于	<code>\${ 5 != 5 }</code> 或 <code>\${ 5 ne 5 }</code>	false
<code><</code> 或 <code>lt</code>	小于	<code>\${ 3 < 5 }</code> 或 <code>\${ 3 lt 5 }</code>	true
<code>></code> 或 <code>gt</code>	大于	<code>\${ 3 > 5 }</code> 或 <code>\${ 3 gt 5 }</code>	false
<code><=</code> 或 <code>le</code>	小于等于	<code>\${ 3 <= 5 }</code> 或 <code>\${ 3 le 5 }</code>	true
<code>>=</code> 或 <code>ge</code>	大于等于	<code>\${ 3 >= 5 }</code> 或 <code>\${ 3 ge 5 }</code>	false

注意

在使用 EL 关系运算符时，不能够写成：

```
${param.password1} == ${param.password2}
```

或者

```
${ ${param.password1} == ${ param.password2 } }
```

而应写成

```
${ param.password1 == param.password2 }
```

接下来，我们依照下列几种情况，详细说明 EL 关系运算符的规则：

(1) A {<, >, <=, >=, lt, gt, le, ge} B

- 假若 A = B，运算符为 <=, le, >=, ge 时，回传 true，否则回传 false
- 假若 A 为 null 或 B 为 null 时，回传 false
- 假若 A 或 B 为 BigDecimal 时，将另一个转为 BigDecimal，然后回传 A.compareTo(B) 的值
- 假若 A 或 B 为 Float、Double 时，皆转为 Double 类型，然后依其运算符运算
- 假若 A 或 B 为 BigInteger 时，将另一个转为 BigInteger，然后回传 A.compareTo(B) 的值
- 假若 A 或 B 为 Byte、Short、Character、Integer 或 Long 时，皆转为 Long 类型，然后依其运算符运算
- 假若 A 或 B 为 String 时，将另一个也转为 String，然后做词汇上的比较
- 假若 A 为 Comparable 时，则：
 - 假若 A.compareTo(B) 产生异常时，则错误！
- 否则，采用 A.compareTo(B) 的比较结果
- 假若 B 为 Comparable 时，则：
 - 假若 B.compareTo(A) 产生异常时，则错误！
- 否则，采用 A.compareTo(B) 的比较结果
- 否则，错误！

(2) A {=, !=, eq, ne} B

- 假若 A = B，依其运算符运算
- 假若 A 为 null 或 B 为 null 时：= / eq 则回传 false，!= / ne 则回传 true
- 假若 A 或 B 为 BigDecimal 时，将另一个转为 BigDecimal，则：
 - 假若运算符为 = / eq，则 回传 A.equals(B)
 - 假若运算符为 != / ne，则 回传 !A.equals(B)

- 假若 A 或 B 为 Float、Double 时，皆转为 Double 类型，然后依其运算符运算
- 假若 A 或 B 为 BigInteger 时，将另一个转为 BigInteger，则：
 - 假若运算符为 `=` / `eq`，则 回传 `A.equals(B)`
 - 假若运算符为 `!=` / `ne`，则 回传 `!A.equals(B)`
- 假若 A 或 B 为 Byte、Short、Character、Integer 或 Long 时，皆转为 Long 类型，然后依其运算符运算
- 假若 A 或 B 为 Boolean 时，将另一个也转为 Boolean，然后依其运算符运算
- 假若 A 或 B 为 String 时，将另一个也转为 String，然后做词汇上的比较
- 否则，假若 `A.equals(B)` 产生异常时，则 错误！
- 否则，然后依其运算符运算，回传 `A.equals(B)`

Tomcat 上的 `jsp-examples` 中，有一个 EL 关系运算符的范例 `basic-comparisons.jsp`。它的程序很简单，所以不在这里多做说明，大家直接看它的执行结果（如图 6-5 所示）：

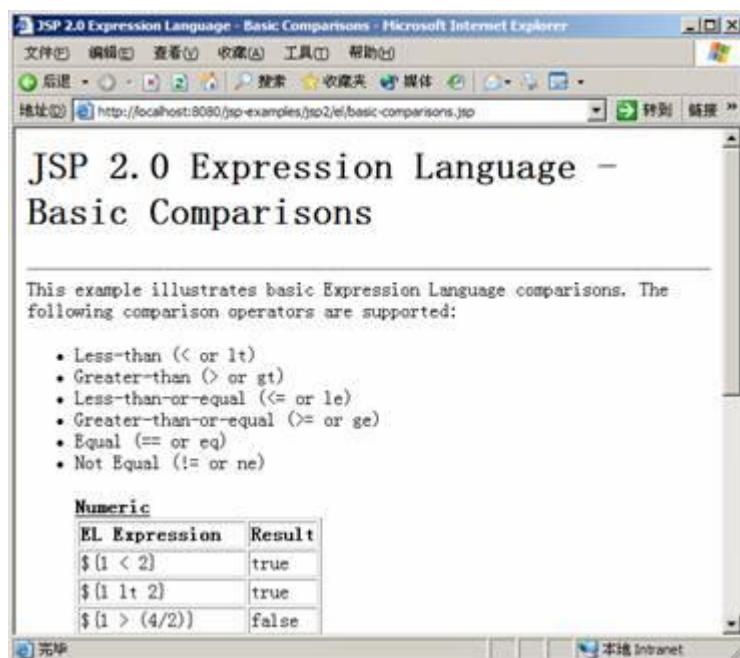


图 6-5 `basic-comparisons.jsp` 的执行结果

6-6 EL 逻辑运算符

EL 逻辑运算符只有三个（见表 6-8）：

表 6-8

逻辑运算符	说 明	范 例	结 果
&& 或 and	交集	<code>\${ A && B }</code> 或 <code>\${ A and B }</code>	true / false
或 or	并集	<code>\${ A B }</code> 或 <code>\${ A or B }</code>	true / false
! 或 not	非	<code>\${ !A }</code> 或 <code>\${ not A }</code>	true / false

下面举几个例子：

```
${ param.month == 7 and param.day == 14 }  
  
${ param.month == 7 || param.day == 14 }  
  
${ not param.choice }
```

EL 逻辑运算符的规则很简单：

(1) A {&&, and, || 或 or } B

- 将 A 和 B 转为 Boolean，然后依其运算符运算

(2) {!, not}A

- 将 A 转为 Boolean，然后依其运算符运算

6-7 EL 其他运算符

EL 除了上述三大类的运算符之外，还有下列几个重要的运算符：

(1) Empty 运算符

(2) 条件运算符

(3) () 括号运算符

6-7-1 Empty 运算符

Empty 运算符主要用来判断值是否为 null 或空的，例如：

```
${ empty param.name }
```

接下来说明 Empty 运算符的规则:

(1) {empty} A

- 假若 A 为 null 时, 回传 true
- 否则, 假若 A 为空 String 时, 回传 true
- 否则, 假若 A 为空 Array 时, 回传 true
- 否则, 假若 A 为空 Map 时, 回传 true
- 否则, 假若 A 为空 Collection 时, 回传 true
- 否则, 回传 false

6-7-2 条件运算符

所谓条件运算符如下:

```
${ A ? B : C }
```

意思是说, 当 A 为 true 时, 执行 B; 而 A 为 false 时, 则执行 C。

6-7-3 括号运算符

括号运算符主要用来改变执行优先权, 例如: `${ A * (B+C) }`

至于运算符的优先权, 如下所示(由高至低, 由左至右):

- [], .
- ()
- - (负)、not、!、empty
- *, /、div、%、mod
- +、- (减)
- <、>、<=、>=、lt、gt、le、ge
- ==、!=、eq、ne
- &&、and

- ||、or
- \${ A ? B : C }

最后笔者写一个 `ELOperator.jsp` 范例，将所有运算符实际操作一遍。

`ELOperator.jsp`

```
<%@ page contentType="text/html; charset=GB2312" %>

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

<html>

<head>

<title>CH6 - ELOperator.jsp</title>

</head>

<body>

<h2>EL 的运算符</h2>

<c:set value="mike" var="username" scope="request" />

<table border="1" width="50%" align="left">

<TR>

<TR>

<TH>运算式</TH>

<TH>结果</TH>

</TR>

<TR><TD>14 + 3</TD><TD>${14 + 3}</TD></TR>
```

```
<TR><TD>14 - 3</TD><TD>${14 - 3}</TD></TR>

<TR><TD>14 * 3</TD><TD>${14 * 3}</TD></TR>

<TR><TD>14 / 3</TD><TD>${14 / 3}</TD></TR>

<TR><TD>14 % 3</TD><TD>${14 % 3}</TD></TR>

<TR><TD>14 == 3</TD><TD>${14 == 3}</TD></TR>

<TR><TD>14 != 3</TD><TD>${14 != 3}</TD></TR>

<TR><TD>14 < 3</TD><TD>${14 < 3}</TD></TR>

<TR><TD>14 > 3</TD><TD>${14 > 3}</TD></TR>

<TR><TD>14 <= 3</TD><TD>${14 <= 3}</TD></TR>

<TR><TD>14 >= 3</TD><TD>${14 >= 3}</TD></TR>

<TR><TD>true && false</TD><TD>${true && false}</TD></TR>

<TR><TD>true || false</TD><TD>${true || false}</TD></TR>

<TR><TD>! false</TD><TD>${! false}</TD></TR>

<TR><TD>empty username</TD><TD>${empty username}</TD></TR>

<TR><TD>empty password</TD><TD>${empty password}</TD></TR>

</table>

</body>

</html>
```

EL 的数学运算符、相等运算符、关系运算符和逻辑运算符就跟其他程序语言一样，并没有特别的地方。但是它的 `empty` 运算符就比较特别，为了测试它，笔者写了这样一程序代码：

```
<c:set value="mike" var="username" scope="request" />
```

这样 Request 属性范围里就存在一个名称为 `username`、值为 `mike` 的属性。执行此程序时，读者将会发现 `${empty username}` 为 `false`；`${empty password}` 为 `true`，其代表的意义就是：它可以在四种属性范围中找到 `username` 这个属性，但是找不到 `password` 这个属性。ELOperator.jsp 的执行

结果如图 6-6:



运算式	结果
14 + 3	17
14 - 3	11
14 * 3	42
14 / 3	4.666666666666667
14 % 3	2
14 == 3	false
14 != 3	true
14 < 3	false
14 > 3	true
14 <= 3	false
14 >= 3	true
true && false	false
true false	true
! false	true
empty username	false
empty password	true

图 6-6 ELOperator.jsp 的执行结果

6-8 EL Functions

前面几节主要介绍 EL 语法的使用和规则, 本节笔者将介绍如何自定义 EL 的函数(functions)。

EL 函数的语法如下:

```
ns:function( arg1, arg2, arg3 ... argN)
```

其中 ns 为前置名称(prefix), 它必须和 taglib 指令的前置名称一样。如下范例:

```
<% @ taglib prefix="my"
    uri="http://jakarta.apache.org/tomcat/jsp2-example-taglib" %>
.....
${my:function(param.name)}
```

前置名称都为 my, 至于 function 为 EL 函数的名称, 而 arg1、arg2 等等, 都是 function 的传入值。在 Tomcat 5.0.16 中有一个简单的 EL 函数范例, 名称为 functions.jsp, 笔者接下来将依此范例来说明如何自定义 EL 函数。

6-8-1 Tomcat EL 函数范例

Tomcat 提供的 EL 函数范例中，自定义两个 EL 函数：reverse 和 countVowels，其中：

reverse 函数：将传入的字符串以反向顺序输出。

countVowels 函数：计算传入的字符串中，和 aeiouAEIOU 吻合的字符个数。

图 6-7 是 functions.jsp 程序的执行结果：

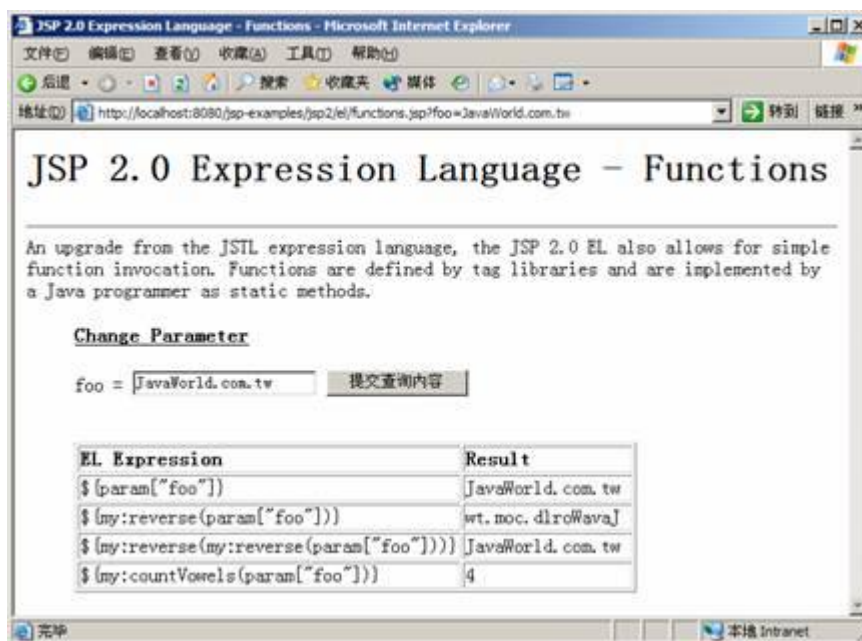


图 6-7 functions.jsp 的执行结果

输入 JavaWorld.com.tw 字符串至 reverse 函数后，回传 wt.moc.dIroWavaJ 的结果；若传入 countVowels 函数后，因为有两个 a 和 o，总共四个字符吻合，所以回传 4。

Tomcat 的 EL 函数范例，主要分为四个部分（见表 6-9）：

表 6-9

web.xml	设定 taglib 的 TLD 文件位置
functions.jsp	使用 EL 函数的范例程序
jsp2-example-taglib.tld	EL 函数、标签库的设定文件
jsp2.examples.el.Functions.java	EL 函数主要程序逻辑处理部分

这四个部分环环相扣，都互有关系，笔者依 functions.jsp 为中心，然后再慢慢说明其他部分。首先我们直接来看 functions.jsp 程序：

6-8-2 functions.jsp

functions.jsp

```
<%@ taglib prefix="my"
uri="http://jakarta.apache.org/tomcat/jsp2-example-taglib"%>

<html>

  <head>

    <title>JSP 2.0 Expression Language - Functions</title>

  </head>

  <body>

    <h1>JSP 2.0 Expression Language - Functions</h1>

    ... 略

    <blockquote>

      <u><b>Change Parameter</b></u>

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

        foo = <input type="text" name="foo" value="${param['foo']}">

          <input type="submit">

      </form>

      <br>

      <code>

        <table border="1">

          <thead>

            <td><b>EL Expression</b></td>

            <td><b>Result</b></td>
```

```
</thead>

<tr>

    <td>\${param["foo"]}</td>

    <td>\${param["foo"]}&nbsp;</td>

</tr>

<tr>

    <td>\${my:reverse(param["foo"])}</td>

    <td>\${my:reverse(param["foo"])}&nbsp;</td>

</tr>

<tr>

    <td>\${my:reverse(my:reverse(param["foo"]))}</td>

    <td>\${my:reverse(my:reverse(param["foo"]))}&nbsp;</td>

</tr>

<tr>

    <td>\${my:countVowels(param["foo"])}</td>

    <td>\${my:countVowels(param["foo"])}&nbsp;</td>

</tr>

</table>

</code>

</blockquote>

</body>

</html>
```

functions.jsp 程序中,一开始定义 taglib,它的前置名称为 my;uri 为 <http://jakarta.apache.org/>

tomcat/jsp2-example-taglib, 如下所示:

```
<%@ taglib prefix="my"
uri="http://jakarta.apache.org/tomcat/jsp2-example-taglib"%>
```

当 Container 执行这段程序时, 它会根据 uri 的值, 到 web.xml 中找相对应的 TLD (Tag Library Descriptor) 文件。至于 web.xml 如何设定两者之间的对应关系, 我们在 6-8-3 小节再说明。

functions.jsp 中包含一个窗体(form), 当用户在文本 [\[玉玉 1 \]](#) 输入框(text input)中输入字符串, 按下按钮时, 底下会显示字符串经过 EL 函数处理后的结果。functions.jsp 程序最重要的部分是调用 EL 函数:

```
${my:reverse(param["foo"])} 
```

上述的意思是接收 foo 参数, 然后传入 reverse 函数。调用 EL 函数的方式很简单, 只要前置名称: 其中 EL 函数名称是被定义在 TLD 文件中, 这会在 6-8-4 小节详细说明。至于 reverse 函数的逻辑运算, 则是被定义在 jsp2.examples.el.Functions.java 程序中, 这部分会在 6-8-5 小节中说明。

注意

TLD 文件主要为标签的设定文件, 其中包含标签的名称、参数等等。在 JSP 2.0 之后, 相关 EL 函数的设定, 也可以在 TLD 文件中定义。

6-8-3 web.xml

web.xml 是每个 web 站台最主要的设定文件, 在这个设定文件中, 可以设定许多东西, 如: Servlet、Resource、Filter 等等。不过现在关心的是如何在 web.xml 中设定 taglib 的 uri 是对应到哪个 TLD 文件。笔者从范例的 web.xml 中节录出设定的片段程序如下:

web.xml

```
<jsp-config>

  <taglib>

    <taglib-uri>

      http://jakarta.apache.org/tomcat/jsp2-example-taglib
```

```
</taglib-uri>

<taglib-location>

    /WEB-INF/jsp2/jsp2-example-taglib.tld

</taglib-location>

</taglib>

</jsp-config>
```

在 web.xml 中，<taglib>用来设定标签的 TLD 文件位置。<taglib-uri>用来指定 taglib 的 uri 位置，用户可以自行给定一个 uri，例如：

```
<taglib-uri>http://www.javaworld.com.tw/jute</taglib-uri>

<taglib-uri>tw.com.javaworld</taglib-uri>
```

<taglib-location>用来指定 TLD 文件的位置。依照范例，它是指定在 WEB-INF/jsp2/目录下的 jsp2-example-taglib.tld。

因此，笔者所节录下来的 web.xml，它所代表的意思是：taglib 的 uri 为 <http://jakarta.apache.org/tomcat/jsp2-example-taglib>，它的 TLD 文件是在 WEB-INF/jsp2/目录下的 jsp2-example-taglib.tld。

6-8-4 jsp2-example-taglib.tld

在 jsp2-example-taglib.tld 中定义许多标签，其中笔者节录一段定义 EL 函数：

jsp2-example-taglib.tld

```
<function>

    <description>Reverses the characters in the given String</description>

    <name>reverse</name>

    <function-class>jsp2.examples.el.Functions</function-class>

    <function-signature>
```

```
        java.lang.String reverse( java.lang.String )

    </function-signature>

</function>

<function>

    <description>Counts the number of vowels (a,e,i,o,u) in the given
        String</description>

    <name>countVowels</name>

    <function-class>jsp2.examples.el.Functions</function-class>

    <function-signature>

        java.lang.String numVowels( java.lang.String )

    </function-signature>

</function>
```

上述定义两个 EL 函数，用<name>来设定 EL 函数名称，它们分别为 reverse 和 countVowels；用<function-class>设定 EL 函数的 Java 类，本范例的 EL 函数都是定义在 jsp2.examples.el.Functions；最后用<function-signature>来设定 EL 函数的传入值和回传值，例如：

```
<function-signature>java.lang.String
reverse( java.lang.String )</function-signature>
```

表示 reverse 函数有一 String 类型的传入值，然后回传 String 类型的值。最后我们再来看 reverse 和 countVowels 的程序。

6-8-5 Functions.java

Functions.java 主要定义三个**公开静态**的方法，分别为：reverse、numVowels 和 caps（见表 6-10）。下面是 Functions.java 完整的程序代码：

Functions.java

```
package jsp2.examples.el;

import java.util.*;

/**
 * Defines the functions for the jsp2 example tag library.
 * <p>Each function is defined as a static method.</p>
 */
public class Functions {

    public static String reverse( String text ) {

        return new StringBuffer( text ).reverse().toString();

    }

    public static int numVowels( String text ) {

        String vowels = "aeiouAEIOU";

        int result = 0;

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

            if( vowels.indexOf( text.charAt( i ) ) != -1 ) {

                result++;

            }

        }

        return result;

    }

}
```

```
}

public static String caps( String text ) {

    return text.toUpperCase();

}

}
```

表 6-10

String reverse(String text)	将 text 字符串的顺序反向处理，然后回传反向后的字符串
int numVowels(String text)	将 text 字符串比对 aeiouAEIOU 等字符，然后回传比对中的次数
String caps(String text)	将 text 字符串都转为大写，然后回传此字符串

注意

在定义 EL 函数时，都必须为公开静态(public static)

<OVER>