2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——

Java 语言编码规范(Java Code Conventions)

译者 晨光（Morning）
搜集整理 ：华竹技术实验室 http://sinoprise.com
简介 ：
本文档讲述了 Java 语言的编码规范，较之陈世忠先生《c++编码规范》的浩繁详尽，此文当属短小精悍了。而其中所列之各项条款，从编码风格，到注意事项，不单只 Java，对于其他语言，也都很有借鉴意义。因为简短，所以易记，大家不妨将此作为 handbook，常备案头，逐一对验。

目录

1 介绍(Introduction)

1.1 为什么要有编码规范(Why Have Code Conventions)

编码规范对于程序员而言尤为重要，有以下几个原因：

- 一个软件的生命周期中，80%的花费在于维护
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
- 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

为了执行规范，每个软件开发人员必须一致遵守编码规范。每个人。

1.2 版权声明(Acknowledgments)

本文档反映的是 Sun MicroSystems 公司，Java 语言规范中的编码标准部分。主要贡献者包括：Peter King，Patrick Naughton，Mike DeMoney，Jonni Kanerva，Kathy Walrath 以及 Scott Hommel。

本文档现由 Scott Hommel 维护，有关评论意见请发至 shommel@eng.sun.com

2 文件名(File Names)

这部分列出了常用的文件名及其后缀。

2.1 文件后缀(File Suffixes)

Java 程序使用下列文件后缀：

文件类别	文件后缀
Java 源文件	.java
Java 字节码文件	.class

2.2 常用文件名(Common File Names)

常用的文件名包括：

文件名	用途
GNUmakefile	makefiles 的首选文件名。我们采用 gnumake 来创建（build）软件。
README	概述特定目录下所含内容的文件的首选文件名

3 文件组织(File Organization)

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。超过 2000 行的程序难以阅读，应该尽量避免。"Java 源文件范例"提供了一个布局合理的 Java 程序范例。

3.1 Java 源文件(Java Source Files)

每个 Java 源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

Java 源文件还遵循以下规则：

- 开头注释（参见"开头注释"）
- 包和引入语句（参见"包和引入语句"）
- 类和接口声明（参见"类和接口声明"）

3.1.1 开头注释(Beginning Comments)

所有的源文件都应该在开头有一个 C 语言风格的注释，其中列出类名、版本信息、日期和版权声明：

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

3.1.2 包和引入语句(Package and Import Statements)

在多数 Java 源文件中，第一个非注释行是包语句。在它之后可以跟引入语句。例如：

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

3.1.3 类和接口声明(Class and Interface Declarations)

下表描述了类和接口声明的各个部分以及它们出现的先后次序。参见"Java源文件范例"中一个包含注释的例子。

	类/接口声明的各部分	注解
1	类/接口文档注释 (/**.....*/)	该注释中所需包含的信息，参见"文档注释"
2	类或接口的声明	
3	类/接口实现的注释	该注释应包含任何有关整个类或接口的信息，而这些信息又不适合

	(/*……*/)如果有必要的话	作为类/接口文档注释。
4	类的(静态)变量	首先是类的公共变量，随后是保护变量，再后是包一级别的变量(没有访问修饰符，access modifier)，最后是私有变量。
5	实例变量	首先是公共级别的，随后是保护级别的，再后是包一级别的(没有访问修饰符)，最后是私有级别的。
6	构造器	
7	方法	这些方法应该按功能，而非作用域或访问权限，分组。例如，一个私有的类方法可以置于两个公有的实例方法之间。其目的是为了更便于阅读和理解代码。

4 缩进排版(Indentation)

4 个空格常被作为缩进排版的一个单位。缩进的确切解释并未详细指定(空格 vs. 制表符)。一个制表符等于 8 个空格(而非 4 个)。

4.1 行长度(Line Length)

尽量避免一行的长度超过 80 个字符，因为很多终端和工具不能很好处理之。

注意：用于文档中的例子应该使用更短的行长，长度一般不超过 70 个字符。

4.2 换行(Wrapping Lines)

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开之：

- 在一个逗号后面断开
- 在一个操作符前面断开
- 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开
- 新的一行应该与上一行同一级别表达式的开头处对齐
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进 8 个空格。

以下是断开方法调用的一些例子：

```
someMethod(longExpression1, longExpression2, longExpression3,
                longExpression4, longExpression5);

var = someMethod1(longExpression1,
                          someMethod2(longExpression2,
                                      longExpression3));
```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```
  longName1 = longName2 * (longName3 + longName4 - longName5)
                    + 4 * longname6; //PREFFER

  longName1 = longName2 * (longName3 + longName4
                                        - longName5) + 4 * longname6;
//AVOID
```

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进 8 个空格

```
  //CONVENTIONAL INDENTATION
  someMethod(int anArg, Object anotherArg, String yetAnotherArg,
                    Object andStillAnother) {
    ...
  }

  //INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
  private static synchronized horkingLongMethodName(int anArg,
          Object anotherArg, String yetAnotherArg,
          Object andStillAnother) {
    ...
  }
```

if 语句的换行通常使用 8 个空格的规则，因为常规缩进(4 个空格)会使语句体看起来比较费劲。比如：

```
  //DON’T USE THIS INDENTATION
  if ((condition1 && condition2)
      || (condition3 && condition4)
      ||!(condition5 && condition6)) { //BAD WRAPS
      doSomethingAboutIt();             //MAKE THIS LINE EASY TO MISS
  }

  //USE THIS INDENTATION INSTEAD
  if ((condition1 && condition2)
          || (condition3 && condition4)
          ||!(condition5 && condition6)) {
      doSomethingAboutIt();
  }

  //OR USE THIS
  if ((condition1 && condition2) || (condition3 && condition4)
          ||!(condition5 && condition6)) {
```

```
    doSomethingAboutIt();
}
```

这里有三种可行的方法用于处理三元运算表达式：

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5 注释(Comments)

Java 程序有两类注释：实现注释(implementation comments)和文档注释(document comments)。实现注释是那些在 C++中见过的，使用/*...*/和//界定的注释。文档注释(被称为"doc comments")是 Java 独有的，并由/**...*/界定。文档注释可以通过 javadoc 工具转换成 HTML 文件。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implementation-free)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如，相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中已清晰表达出来的重复信息。多余的的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意：频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候，考虑一下重写代码使其更清晰。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

5.1 实现注释的格式(Implementation Comment Formats)

程序可以有 4 种实现注释的风格：块(block)、单行(single-line)、尾端(trailing)和行末(end-of-line)。

5.1.1 块注释(Block Comments)

块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

```
/*
 * Here is a block comment.
 */
```

块注释可以以/*-开头，这样 indent(1)就可以将之识别为一个代码块的开始，而不会重排它。

```
/*-
  * Here is a block comment with some very special
  * formatting that I want indent(1) to ignore.
  *
  *    one
  *        two
  *            three
  */
```

注意：如果你不使用 indent(1)，就不必在代码中使用/*-，或为他人可能对你的代码运行 indent(1)作让步。

参见"文档注释"

5.1.2 单行注释(Single-Line Comments)

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释(参见"块注释")。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

```
if (condition) {

  /* Handle the condition. */
  ...
}
```

5.1.3 尾端注释(Trailing Comments)

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个 Java 代码中尾端注释的例子：

```
if (a == 2) {
    return TRUE;              /* special case */
} else {
    return isPrime(a);         /* works only for odd a */
}
```

5.1.4 行末注释(End-Of-Line Comments)

注释界定符"//"，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
}

//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

5.2 文档注释(Documentation Comments)

注意：此处描述的注释格式之范例，参见"Java源文件范例"

若想了解更多，参见"How to Write Doc Comments for Javadoc"，其中包含了有关文档注释标记的信息(@return, @param, @see)：

http://java.sun.com/javadoc/writingdoccomments/index.html

若想了解更多有关文档注释和 javadoc 的详细资料，参见 javadoc 的主页：

http://java.sun.com/javadoc/index.html

文档注释描述 Java 的类、接口、构造器，方法，以及字段(field)。每个文档注释都会被置于注释定界符/**...*/之中，一个注释对应一个类、接口或成员。该注释应位于声明之前：

```
/**
  * The Example class provides ...
  */
public class Example { ...
```

注意顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(/**)不需缩进；随后的文档注释每行都缩进 1 格(使星号纵向对齐)。成员，包括构造函数在内，其文档注释的第一行缩进 4 格，随后每行都缩进 5 格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释(见 5.1.1)或紧跟在声明后面的单行注释(见 5.1.2)。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为 Java 会将位于文档注释之后的第一个声明与其相关联。

6 声明(Declarations)

6.1 每行声明变量的数量(Number Per Line)

推荐一行一个声明，因为这样以利于写注释。亦即，

```
int level;  // indentation level
int size;   // size of table
```

要优于，

```
int level, size;
```

不要将不同类型变量的声明放在同一行，例如：

```
int foo,  fooarray[];   //WRONG!
```

注意：上面的例子中，在类型和标识符之间放了一个空格，另一种被允许的替代方式是使用制表符：

```
int         level;         // indentation level
int         size;          // size of table
Object      currentEntry;  // currently selected table entry
```

6.2 初始化(Initialization)

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

6.3 布局(Placement)

只在代码块的开始处声明变量。（一个块是指任何被包含在大括号"{"和"}"中间的代码。）不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

该规则的一个例外是 for 循环的索引变量

```
for (int i = 0; i < maxLoops; i++) { ... }
```

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

6.4 类和接口的声明(Class and Interface Declarations)

当编写类和接口是，应该遵守以下格式规则：

- 在方法名与其参数列表之前的左括号"("间不要有空格
- 左大括号"{"位于声明语句同行的末尾
- 右大括号"}"另起一行，与相应的声明语句对齐，除非是一个空语句，"}"应紧跟在"{"之后

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- 方法与方法之间以空行分隔

7 语句(Statements)

7.1 简单语句(Simple Statements)

每行至多包含一条语句，例如：

```
argv++;       // Correct
argc--;       // Correct
argv++; argc--;       // AVOID!
```

7.2 复合语句(Compound Statements)

复合语句是包含在大括号中的语句序列，形如"{ 语句 }"。例如下面各段。

- 被括其中的语句应该较之复合语句缩进一个层次
- 左大括号"{"应位于复合语句起始行的行尾；右大括号"}"应另起一行并与复合语句首行对齐。
- 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如 if-else 或 for 控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入 bug。

7.3 返回语句(return Statements)

一个带返回值的 return 语句不使用小括号"()"，除非它们以某种方式使返回值更为显见。例如：

```
return;
```

```
return myDisk.size();

return (size ? size : defaultSize);
```

7.4 if，if-else，if else-if else 语句(if, if-else, if else-if else Statements)

if-else 语句应该具有如下格式：

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else{
    statements;
}
```

注意：if 语句总是用"{"和"}"括起来，避免使用如下容易引起错误的格式：

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

7.5 for 语句(for Statements)

一个 for 语句应该具有如下格式：

```
for (initialization; condition; update) {
    statements;
}
```

一个空的 for 语句(所有工作都在初始化，条件判断，更新子句中完成）应该具有如下格式：

```
for (initialization; condition; update);
```

当在 for 语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在 for 循环之前(为初始化子句)或 for 循环末尾(为更新子句)使用单独的语句。

7.6 while 语句(while Statements)

一个 while 语句应该具有如下格式

```
while (condition) {
    statements;
}
```

一个空的 while 语句应该具有如下格式：

```
while (condition);
```

7.7 do-while 语句(do-while Statements)

一个 do-while 语句应该具有如下格式：

```
do {
    statements;
} while (condition);
```

7.8 switch 语句(switch Statements)

一个 switch 语句应该具有如下格式：

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;
```

```
default:
    statements;
    break;
}
```

每当一个 case 顺着往下执行时(因为没有 break 语句)，通常应在 break 语句的位置添加注释。上面的示例代码中就包含注释/* falls through */。

7.9 try-catch 语句(try-catch Statements)

一个 try-catch 语句应该具有如下格式：

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

一个 try-catch 语句后面也可能跟着一个 finally 语句，不论 try 代码块是否顺利执行完，它都会被执行。

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

8 空白(White Space)

8.1 空行(Blank Lines)

空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应该总是使用两个空行：

- 一个源文件的两个片段(section)之间
- 类声明和接口声明之间

下列情况应该总是使用一个空行：

- 两个方法之间
- 方法内的局部变量和方法的第一条语句之间
- 块注释（参见"5.1.1"）或单行注释（参见"5.1.2"）之前
- 一个方法内的两个逻辑段之间，用以提高可读性

8.2 空格(Blank Spaces)

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开，例如：

```
while (true) {
    ...
}
```

注意：空格不应该置于方法名与其左括号之间。这将有助于区分关键字和方法调用。
- 空白应该位于参数列表中逗号的后面
- 所有的二元运算符，除了"."，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号("-")、自增("++")和自减("--")。例如：

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- for 语句中的表达式应该被空格分开，例如：

```
for (expr1; expr2; expr3)
```

- 强制转型后应该跟一个空格，例如：

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

9 命名规范(Naming Conventions)

命名规范使程序更易读，从而更易于理解。它们也可以提供一些有关标识符功能的信息，以助于理解代码，例如，不论它是一个常量，包，还是类。

标识符类型	命名规则	例子
包(Packages)	一个唯一包名的前缀总是全部小写的 ASCII 字母并且是一个顶级域名，通常是 com，edu，gov，mil，net，org，或 1981 年 ISO 3166	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	标准所指定的标识国家的英文双字符代码。包名的后续部分根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门(department)，项目(project)，机器(machine)，或注册名(login names)。	
类(Classes)	命名规则：类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被更广泛使用，像 URL，HTML)	class Raster; class ImageSprite;
接口(Interfaces)	命名规则：大小写规则与类名相似	interface RasterDelegate; interface Storing;

方法(Methods)	方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。	run();
		runFast();
		getBackground();

变量(Variables)	除了变量名外，所有实例，包括类，类常量，均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量。临时变量通常被取名为 i，j，k，m 和 n，它们一般用于整型；c，d，e，它们一般用于字符型。	char c; int i; float myWidth;
实例变量(Instance Variables)	大小写规则和变量名相似，除了前面需要一个下划线	int _employeeId; String _name; Customer _customer;

常量(Constants)	类常量和 ANSI 常量的声明，应该全部大写，单词间用下划线隔开。(尽量避免 ANSI 常量，容易引起错误)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

10 编程惯例(Programming Practices)

10.1 提供对实例以及类变量的访问控制(Providing Access to Instance and Class Variables)

若没有足够理由，不要把实例或类变量声明为公有。通常，实例变量无需显式的设置(set)和获取(gotten)，通常这作为方法调用的边缘效应 (side effect)而产生。

一个具有公有实例变量的恰当例子，是类仅作为数据结构，没有行为。亦即，若你要使用一个结构(struct)而非一个类(如果 java 支持结构的话)，那么把类的实例变量声明为公有是合适的。

10.2 引用类变量和类方法(Referring to Class Variables and Methods)

避免用一个对象访问一个类的静态变量和方法。应该用类名替代。例如：

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

10.3 常量(Constants)

位于 for 循环中作为计数器值的数字常量，除了-1,0 和 1 之外，不应被直接写入代码。

10.4 变量赋值(Variable Assignments)

避免在一个语句中给多个变量赋相同的值。它很难读懂。例如：

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如：

```
if (c++ = d++) {        // AVOID! (Java disallows)
    ...
}
```

应该写成

```
if ((c++ = d++) != 0) {
  ...
}
```

不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。例如：

```
d = (a = b + c) + r;        // AVOID!
```

应该写成

```
a = b + c;
d = a + r;
```

10.5 其它惯例(Miscellaneous Practices)

10.5.1 圆括号(Parentheses)

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。即使运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

```
if (a == b && c == d)     // AVOID!
if ((a == b) && (c == d))  // RIGHT
```

10.5.2 返回值(Returning Values)

设法让你的程序结构符合目的。例如：

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

应该代之以如下方法：

```
return booleanExpression;
```

类似地：

```
if (condition) {
    return x;
}
return y;
```

应该写做：

```
return (condition ? x : y);
```

10.5.3 条件运算符"?"前的表达式(Expressions before '?' in the Conditional Operator)

如果一个包含二元运算符的表达式出现在三元运算符" ? : "的"?"之前，那么应该给表达式添上一对圆括号。例如：

```
(x >= 0) ? x : -x;
```

10.5.4 特殊注释(Special Comments)

在注释中使用 XXX 来标识某些未实现(bogus)的但可以工作(works)的内容。用 FIXME 来标识某些假的和错误的内容。

11 代码范例(Code Examples)

11.1 Java 源文件范例(Java Source File Example)

下面的例子，展示了如何合理布局一个包含单一公共类的Java源程序。接口的布局与其相似。更多信息参见"类和接口声明"以及"文挡注释"。

```
/*
 * @(#)Blah.java         1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version     1.82 18 Mar 1999
 * @author      Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */
```

```
    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

7

第七章

JSTL 1.1

JSTL 全名为 JavaServer Pages Standard Tag Library，目前最新的版本为 1.1 版。JSTL 是由 JCP(Java Community Process)所制定的标准规范，它主要提供给 Java Web 开发人员一个标准通用的标签函数库。

Web 程序员能够利用 JSTL 和 EL 来开发 Web 程序，取代传统直接在页面上嵌入 Java 程序(Scripting)的做法，以提高程序的阅读性、维护性和方便性。

本章中，笔者将详细介绍如何使用 JSTL 中各种不同的标签，将依序介绍条件、循环、URL、U18N、XML、SQL 等标签的用法，让读者对 JSTL 有更深层的了解，并且能够学会如何使用 JSTL。本章将分 6 节来介绍：

7-1 JSTL 1.1 简介

JavaServer Pages Standard Tag Library (1.1)，它的中文名称为 JSP 标准标签函数库。JSTL 是一个标准的已制定好的标签库，可以应用于各种领域，如：基本输入输出、流程控制、循环、XML 文件剖析、数据库查询及国际化和文字格式标准化的应用等。从表 7-1 可以知道，JSTL 所提供的标签函数库主要分为五大类：

（1） 核心标签库 (Core tag library)
（2） I18N 格式标签库 (I18N-capable formatting tag library)
（3） SQL 标签库 (SQL tag library)
（4） XML 标签库 (XML tag library)
（5） 函数标签库 (Functions tag library)

表 7-1

JSTL	前置名称	URI	范 例
核心标签库	c	http://java.sun.com/jsp/jstl/core	<c:out>
I18N 格式标签库	fmt	http://java.sun.com/jsp/jstl/xml	<fmt:formatDate>
SQL 标签库	sql	http://java.sun.com/jsp/jstl/sql	<sql:query>
XML 标签库	xml	http://java.sun.com/jsp/jstl/fmt	<x:forBach>
函数标签库	fn	http://java.sun.com/jsp/jstl/functions	<fn:split>

另外，JSTL 也支持 EL(Expression Language)语法，例如：在一个标准的 JSP 页面中可能会使用到如下的写法：

```
<%= userList.getUser().getPhoneNumber() %>
```

使用 JSTL 搭配传统写法会变成这样：

```
<c_rt:out value="<%= userList.getUser( ).getPhoneNumber( ) %>" />
```

使用 JSTL 搭配 EL，则可以改写成如下的形式：

```
<c:out value="${userList.user.phoneNumber}" />
```

虽然对网页设计者来说，假如没有学过 Java Script 或者是第一次看到这种写法时，可能会搞不太懂，但是与 Java 语法相比，这应该更加容易学习。

7-1-1 安装使用 JSTL 1.1

JSTL 1.1 必须在支持 Servlet 2.4 且 JSP 2.0 以上版本的 Container 才可使用。JSTL 主要由 Apache 组织的 Jakarta Project 所实现，因此读者可以到 http://jakarta.apache.org/builds/jakarta-taglibs/releases/standard/ 下载实现好的 JSTL 1.1，或者直接使用本书光盘中 JSTL 1.1，软件名称为：*jakarta-taglibs-standard-current.zip*。

下载完后解压缩，可以发现文件夹中所包含的内容如图 7-1 所示：

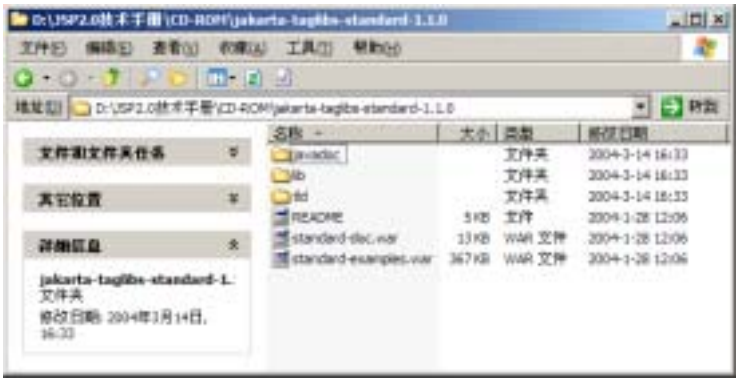

图 7-1 jakarta-taglibs-standard-1.1.0-B1 的目录结构

将 lib 中的 *jstl.jar*、*standard.jar* 复制到 Tomcat 的 *WEB-INF\lib* 中，然后就可以在 JSP 网页中使用 JSTL 了。除了复制 *.jar* 文件外，最好也把 tld 文件的目录也复制到 *WEB-INF* 中，以便日后使用。

注意
lib 目录下，除了 jstl.jar 和 standard.jar 之外，还有 old-dependencies 目录，这目录里面的东西是让之前 JSTL 1.0 的程序也能够在 JSTL 1.1 环境下使用。tld 目录下有许多 TLD 文件，其中大部分都是 JSTL 1.0 的 TLD 文件，例如：c-1_0.tld 和 c-1_0-rt.tld。

下面写一个测试用的范例程序 *HelloJSTL.jsp*，程序主要是显示浏览器的版本和欢迎的字符串。

■ *HelloJSTL.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
  <head>
  <title>测试你的第一个使用到 JSTL 的网页</title>
  </head>

  <body>
  <c:out value="欢迎测试你的第一个使用到 JSTL 的网页"/>
  </br>你使用的浏览器是：</br>
  <c:out value="${header['User-Agent']}"/>
  <c:set var="a" value="David O'Davies"  />
  <c:out value="David O'Davies" escapeXml="true"/>
  </body>
</html>
```

在 *HelloJSTL.jsp* 的范例里，笔者用到核心标签库(Core)中的标准输出功能和 EL 的 header 隐含对象。若要在 JSP 网页中使用 JSTL 时，一定要先做下面这行声明：

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

这段声明表示我将使用 JSTL 的核心标签库。一般而言，核心标签库的前置名称(prefix)都

为 c，当然你也可以自行设定。不过 uri 此时就必须为 http://java.sun.com/jsp/jstl/core。

注意
JSTL 1.0 中，核心标签库的 uri 默认为 http://java.sun.com/jstl/core，比 JSTL 1.1 少一个 jsp/ 的路径。因为 JSTL 1.1 同时支持 JSTL 1.0 和 1.1，所以假若核心标签库的 uri 为 http://java.sun.com/jstl/core，则将会使用到 JSTL 1.0 的核心标签库。

接下来使用核心标签库中的 out 标签，显示 value 的值。${header['User-Agent']}表示取得表头里的 User-Agent 的值，即有关用户浏览器的种类。

```
<c:out value="欢迎测试你的第一个使用到 JSTL 的网页" />
<c:out value="${header['User-Agent']}" />
```

HelloJSTL.jsp 的执行结果如图 7-2 所示。

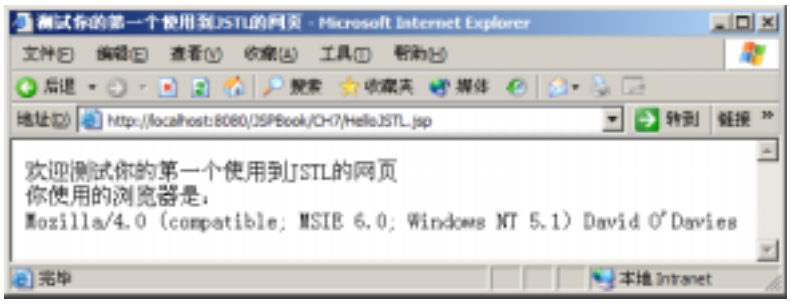

图 7-2 HelloJSTL.jsp 的执行结果

假若读者想要自定义 taglib 的 uri 时，那就必须在 *web.xml* 中加入设定值。例如：假若 uri 想要改为 http://www.javaworld.com.tw/jstl/core 时，*web.xml* 就必须加入如下设定：

```
<web-app>
:
  <jsp-config>
    <taglib>
       <taglib-uri>http://www.javaworld.com.tw/jstl/core</taglib-uri>
       <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
    </taglib>
  </jsp-config>
:
</web-app>
```

在上面的设定中，<taglib-uri>主要是设定标签库的 URI；而<taglib-location>则是用来设定标签对应的 TLD 文件。因此，使用<%@ taglib %>指令时，可以直接写成如下语句：

```
<%@ taglib prefix="c" uri="http://www.javaworld.com.tw/jsp/jstl/core" %>
```

7-1-2 JSTL 1.1 VS. JSTL 1.0

JSTL 1.0 更新至 JSTL 1.1 时，有以下几点不同：

(1) EL 原本是定义在 JSTL 1.0 的，现在 EL 已经正式纳入 JSP 2.0 标准规范中，所以在 JSTL 1.1 规范中，已经没有 EL 的部分，但是 JSTL 依旧能使用 EL。

(2) JSTL 1.0 中，又分 EL 和 RT 两种函数库，到了 JSTL 1.1 之后，已经不再分这两种了。以下说明 EL 和 RT 的差别：

EL

- 完全使用 Expression Language
- 简单
- 建议使用

RT

- 使用 Scriptlet
- Java 语法
- 供不想转换且习惯旧表示法的开发者使用

笔者在此强烈建议大家使用 EL 来做，简单又方便。

(3) JSTL 1.1 新增**函数(**functions)标签库，主要提供一些好用的字符串处理函数，例如：fn:contains、fn:containsIgnoreCase、fn:endsWith、fn:indexOf、fn:join、fn:length、fn:replace、fn:split、fn:startsWith 和 fn:substring 等等。

除了上述三项比较大的改变之外，还包括许多小改变，在此不多加说明，有兴趣的读者可以去看 JSTL 1.1 附录 B “ Changes ” 部分，那里有更详尽的说明。

7-1-3 安装 standard-examples

当解压缩 *jakarta-taglibs-standard-current.zip* 后，文件夹内(见图 7-1)有一个 *standard-examples.war* 的文件，将它移至 Tomcat 的 *webapps* 后，重新启动 Tomcat 会发现，在 *webapps* 目录下多了一个 *standard-examples* 的目录。接下来我们打开 IE，在 URL 位置上输入 http://localhost:8080/standard-examples，你将会看到图 7-3 所示的画面。

这个站台有很多 JSTL 的范例，它包括以下几部分：

- General Purpose Tags
- Conditional Tags
- Iterator Tags
- Import Tags
- I18N & Formatting Tags
- XML Tags
- SQL Tags
- Functions
- Tag Library Validators
- Miscellaneous

图 7-3 standard-examples 站台

这些范例程序几乎涵盖了所有的 JSTL 标签函数库，假若读者对哪一个标签的使用有问题，可以先来找一找这里的范例程序，应该或多或少会有所帮助。

7-2 核心标签库 (Core tag library)

首先介绍的核心标签库(Core)主要有：基本输入输出、流程控制、迭代操作和 URL 操作。详细的分类如表 7-2 所示，接下来笔者将为读者一一介绍每个标签的功能。

表 7-2

分 类	功能分类	标签名称
Core	表达式操作	out set remove catch
	流程控制	if choose when otherwise
	迭代操作	forEach forTokens

续表

分 类	功能分类	标签名称
Core	URL 操作	import param url param redirect param

在 JSP 中要使用 JSTL 中的核心标签库时，必须使用<%@ taglib %>指令，并且设定 prefix 和 uri 的值，通常设定如下：

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

上述的功用在于声明将使用 JSTL 的核心标签库。

注意
假若没有上述声明指令，将无法使用 JSTL 的核心功能，这是读者在使用 JSTL 时必须要小心的地方。

7-2-1 表达式操作

表达式操作分类中包含四个标签：<c:out>、<c:set>、<c:remove>和<c:catch>。接下来将依序介绍这四个标签的用法。

● **<c:out>**

<c:out>主要用来显示数据的内容，就像是 <%= scripting-language %> 一样，例如：

```
Hello ! <c:out value="${username}" />
```

语法

语法1：没有本体(body)内容

```
<c:out value="value" [escapeXml="{true|false}"] [default="defaultValue"] />
```

语法2：有本体内容

```
<c:out value="value" [escapeXml="{true|false}"]>
default value
</c:out>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
value	需要显示出来的值	Y	Object	是	无
default	如果 value 的值为 null，则显示 default 的值	Y	Object	否	无
escapeXml	是否转换特殊字符，如：< 转换成 <	Y	boolean	否	true

注意
表格中的 EL 字段，表示此属性的值是否可以为 EL 表达式，例如：Y 表示 attribute = "${表达式}" 为符合语法的，N 则反之。

Null 和 错误处理

· 假若 value 为 null，会显示 default 的值；假若没有设定 default 的值，则会显示一个空的字符串。

说明

一般来说，<c:out>默认会将 <、>、'、" 和 & 转换为 <、>、'、" 和 &。假若不想转换时，只需要设定<c:out>的 escapeXml 属性为 fasle 就可以了（见表 7-3）。

表 7-3

字符	Entity
<	&lt;
>	&gt;
'	&#039;
"	&#034;
&	&amp;

范例

```
<c:out value="Hello JSP 2.0 !! " />
<c:out value="${ 3 + 5 }" />
<c:out value="${ param.data }" default="No Data" />
<c:out value="<p>有特殊字符</p>" />
<c:out value="<p>有特殊字符</p>" escapeXml="false" />
```

1．在网页上显示 Hello JSP 2.0 !! ；
2．在网页上显示 8；
3．在网页上显示由窗体传送过来的 data 参数之值，假若没有 data 参数，或 data 参数的值为 null 时，则网页上会显示 No Data；
4．在网页上显示 " 有特殊字符 "；
5．在网页上显示 " 有特殊字符 "。

● **<c:set>**

<c:set>主要用来将变量储存至 JSP 范围中或是 JavaBean 的属性中。

语法

语法1：将 value的值储存至范围为scope的 varName 变量之中

```
<c:set value="value" var="varName"
[scope="{ page|request|session|application }"]/>
```

语法2：将本体内容的数据储存至范围为scope的 varName 变量之中

```
<c:set var="varName" [scope="{ page|request|session|application }"]>
… 本体内容
</c:set>
```

语法3：将 value的值储存至 target 对象的属性中

```
<c:set value="value" target="target" property="propertyName" />
```

语法4：将本体内容的数据储存至 target 对象的属性中

```
<c:set target="target" property="propertyName">
... 本体内容
</c:set>
```

属性

名 称	说 明	EL	类型	必须	默认值
value	要被储存的值	Y	Object	否	无
var	欲存入的变量名称	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	page
target	为一 JavaBean 或java.util.Map 对象	Y	Object	否	无
property	指定 target 对象的属性	Y	String	否	无

Null 和 错误处理

- 语法 3 和语法 4 会产生异常错误，有以下两种情况：
  - target 为 null
  - target 不是 java.util.Map 或 JavaBean 对象
- 假若 value 为 null 时：将由储存变量改为移除变量
  - 语法 1：由 var 和 scope 所定义的变量，将被移除
    - 若 scope 已指定时，则 PageContext.removeAttribute(varName, scope)
    - 若 scope 未指定时，则 PageContext.removeAttribute(varName)
  - 语法 3：
    - 假若 target 为 Map 时，则 Map.remove(property)
    - 假若 target 为 JavaBean 时，property 指定的属性为 null

说明

使用<c:set>时，var 主要用来存放表达式的结果；scope 则是用来设定储存的范围，例如：假若 scope="session"，则将会把数据储存在 session 中。如果<c:set>中没有指定 scope 时，则它会默认存在 Page 范围里。

注意
var 和 scope 这两个属性不能使用表达式来表示，例如：我们不能写成 scope="${ourScope}"或者是 var="${username}"。

我们考虑下列的写法：

```
<c:set var="number" scope="session" value="${1 + 1}"/>
```

把 1+1 的结果 2 储存到 number 变量中。如果<c:set>没有 value 属性，此时 value 之值在<c:set>和</c:set>之间，本体内容看下面的范例：

```
<c:set var="number" scope="session">
<c:out value="${1+1}" />
</c:set>
```

上面的 <c:out value="${1+1}" /> 部分可以改写成 2 或是 <%=1+1%> ，结果都会一样，也就是说，<c:set>是把本体(body)运算后的结果来当做 value 的值。另外，<c:set>会把 body 中最开头和结尾的空白部分去掉。如：

```
<c:set var="number" scope="session">
        1 + 1
</c:set>
```

则 number 中储存的值为 1 + 1 而不是 1 + 1。

范例

```
<c:set var="number" scope="request" value="${1 + 1}" />
<c:set var="number" scope="session" />
${3 + 5}
</c:set>
<c:set var="number" scope="request" value="${ param.number }" />
<c:set target="User" property="name" value="${ param.Username}" />
```

1．将 2 存入 Request 范围的 number 变量中；
2．将 8 存入 Session 范围的 number 变量中；
3．假若 ${param.number}为 null 时,则**移除** Request 范围的 number 变量 ;若${param.number}不为 null 时，则将 ${param.number}的值存入 Request 范围的 number 变量中；
4．假若 ${param.Username}为 null 时，则设定 User(JavaBean)的 name 属性为 null；若不为 null 时，则将 ${param.Username}的值存入 User(JavaBean)的 name 属性(setter 机制)。

注意
上述范例的 3.中，假若 ${param.number}为 null 时，则表示**移除** Request 范围的 number 变量。

- **<c:remove>**

  <c:remove>主要用来移除变量。

语法

```
<c:remove var="varName" [scope="{ page|request|session|application }"] />
```

属性

名 称	说 明	EL	类型	必须	默认值
var	欲移除的变量名称	N	String	是	无
scope	var 变量的 JSP 范围	N	String	否	page

说明

<c:remove>必须要有 var 属性，即要被移除的属性名称，scope 则可有可无，例如：

```
<c:remove var="number" scope="session" />
```

将 number 变量从 Session 范围中移除。若我们不设定 scope，则<c:remove>将会从 Page、Request、Session 及 Application 中顺序寻找是否存在名称为 number 的数据，若能找到时，则将它移除掉，反之则不会做任何的事情。

范例

笔者在这里写一个使用到<c:set>和<c:remove>的范例，能让读者可以更快地了解如何使用它们，此范例的名称为 *Core_set_remove.jsp*。

■ *Core_set_remove.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_set_remove.jsp</title>
</head>
<body>

<h2><c:out value="<c:set>和<c:remove> 的用法" /></h2>

<c:set scope="page" var="number">
<c:out value="${1+1}"/>
</c:set>

<c:set scope="request" var="number">
<%= 3 %>
</c:set>

<c:set scope="session" var="number">
4
</c:set>

初始设置
<table border="1" width="30%">
<tr>
  <th>pageScope.number</th>
  <td><c:out value="${pageScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>requestScope.number</th>
  <td><c:out value="${requestScope.number}" default="No Data" /></td>
```

```
</tr>
<tr>
  <th>sessionScope.number</th>
  <td><c:out value="${sessionScope.number}" default="No Data" /></td>
</tr>
</table></br>

<c:out value='<c:remove var="number" scope="page" />之后'/>
<c:remove var="number" scope="page" />
<table border="1" width="30%">
<tr>
  <th>pageScope.number</th>
  <td><c:out value="${pageScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>requestScope.number</th>
  <td><c:out value="${requestScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>sessionScope.number</th>
  <td><c:out value="${sessionScope.number}" default="No Data" /></td>
</tr>
</table></br>

<c:out value='<c:remove var="number" />之后'/>
<c:remove var="number" />
<table border="1" width="30%">
<tr>
  <th>pageScope.number</th>
  <td><c:out value="${pageScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>requestScope.number</th>
  <td><c:out value="${requestScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>sessionScope.number</th>
  <td><c:out value="${sessionScope.number}" default="No Data" /></td>
</tr>
</table>
</body>
</html>
```

笔者一开始各在 Page、Request 和 Session 三个属性范围中储存名称为 number 的变量。然后先使用<c:remove var="number" scope="page" />把 Page 中的 number 变量移除，最后再使用<c:remove var="number" />把所有属性范围中 number 的变量移除。*Core_set_remove.jsp* 的执行结果如图 7-4 所示：

图 7-4 Core_set_remove.jsp 的执行结果

● **<c:catch>**

<c:catch>主要用来处理产生错误的异常状况，并且将错误信息储存起来。

语法

```
<c:catch [var="varName"] >
… 欲抓取错误的部分
</c:catch>
```

属性

名 称	说 明	EL	类型	必须	默认值
var	用来储存错误信息的变量	N	String	否	无

说明

<c:catch>主要将可能发生错误的部分放在<c:catch>和</c:catch>之间。如果真的发生错误，可以将错误信息储存至 varName 变量中，例如：

```
<c:catch var="message">
:  //可能发生错误的部分
</c:catch>
```

另外，当错误发生在<c:catch>和</c:catch>之间时，则只有<c:catch>和</c:catch>之间的程序会被中止忽略，但整个网页不会被中止。

范例

笔者写一个简单的范例，文件名为 *Core_catch.jsp*，来让大家看一下<c:catch>的使用方式。

■ *Core_catch.jsp*

```
<%@ page contentType="text/html;charset=GB2312 " %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_catch.jsp</title>
</head>
<body>

<h2><c:out value="<c:catch> 的用法" /></h2>

<c:catch var="error_Message">
<%
    String eFormat = "not number";
    int i = Integer.parseInt(eFormat);
%>
</c:catch>
${error_Message}
</body>
</html>
```

笔者将一个字符串转成数字，如果字符串可以转为整数，则不会发生错误。但是这里笔者故意传入一个不能转成数字的字符串，让<c:catch>之间产生错误。当错误发生时，它会自动将错误存到 error_Message 变量之中，最后再用<c:out>把错误信息显示出来，执行结果如图 7-5 所示。

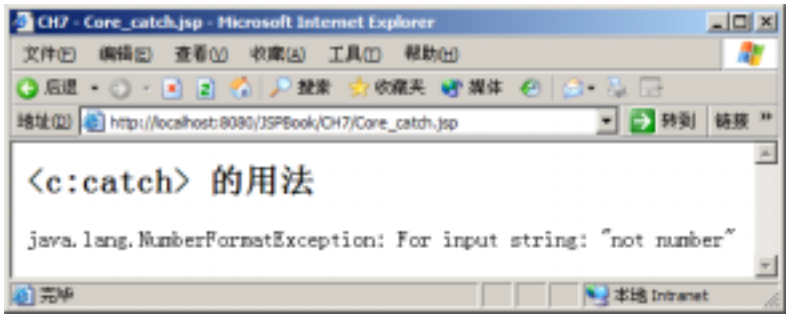

图 7-5 Core_catch.jsp 的执行结果

可以发现到网页确实显示格式错误的信息。如果我们不使用<c:catch>，而把范例中的<c:catch>和</c:catch>拿掉，结果如图 7-6 所示。

图 7-6 Core_catch.jsp 没有 < c:catch > 和 < /c:catch > 的执行结果

7-2-2 流程控制

流程控制分类中包含四个标签：<c:if>、<c:choose>、<c:when>和<c:otherwise>，笔者依此顺序依次说明这四个标签的使用。

● **<c:if>**

<c:if>的用途就和我们一般在程序中用的 if 一样。

语法

语法1：没有本体内容(body)

```
<c:if test="testCondition" var="varName"
                  [scope="{page|request|session|application}"]/>
```

语法2：有本体内容

```
<c:if test="testCondition" [var="varName"]
                  [scope="{page|request|session|application}"]>
本体内容
</c:if>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
test	如果表达式的结果为 true，则执行本体内容，false 则相反	Y	boolean	是	无
var	用来储存 test 运算后的结果，即 true 或 false	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	page

说明

<c:if> 标签必须要有 test 属性，当 test 中的表达式结果为 true 时，则会执行本体内容；如果为 false，则不会执行。例如：${param.username = = 'admin'}，如果 param.username 等于 admin 时，结果为 true；若它的内容不等于 admin 时，则为 false。

接下来看下列的范例：

```
<c:if test="${param.username = = 'admin' }">
ADMIN 您好!! //body 部分
</c:if>
```

如果名称等于 admin，则会显示"ADMIN 您好!! "的动作，如果相反，则不会执行<c:if>的 body 部分，所以不会显示"ADMIN 您好!! //body 部分"。另外<c:if>的本体内容除了能放纯文字，还可以放任何 JSP 程序代码(Scriptlet)、JSP 标签或者 HTML 码。

除了 test 属性之外，<c:if>还有另外两个属性 var 和 scope。当我们执行<c:if>的时候，可以将这次判断后的结果存放到属性 var 里；scope 则是设定 var 的属性范围。哪些情况才会用到 var 和 scope 这两个属性呢？例如：当表达式过长时，我们会希望拆开处理，或是之后还须使用此结果时，也可以用它先将结果暂时保留，以便日后使用。

范例

笔者写了一个简单的范例，名称为 *Core_if.jsp*。

■ *Core_if.jsp*

```
<%@ page contentType="text/html;charset=GB2312 " %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html>
<head>
  <title>CH7 - Core_if.jsp</title>
</head>
<body>

<h2><c:out value="<c:if> 的用法" /></h2>

<c:if test="${param.username == 'Admin'}" var="condition" scope="page">
您好 Admin 先生
</c:if></br>
```

```
执行结果为: ${condition}
</body>
</html>
```

笔者在判断用户送来的参数时，如果 username 的值等于 Admin 时，则会将 condition 设为 true 并存放于 pageScope 中，否则存放于 condition 中，最后再显示结果。因为 JSTL 会自动找寻 condition 所存在的属性范围，因此只须使用 ${condition}，而不用 ${pageScope.condition}。*Core_if.jsp* 的执行结果如图 7-7。

注意
执行本范例时，请在 Core_if.jsp 后加上?username=Admin。

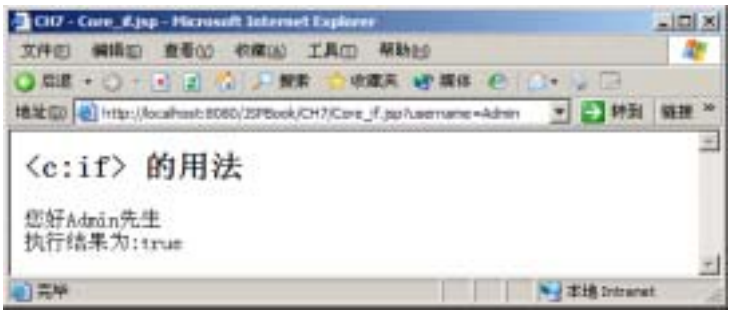

图 7-7 Core_if.jsp 的执行结果

- **<c:choose>**

  <c:choose>本身只当做 <c:when> 和 <c:otherwise> 的父标签。

语法

```
<c:choose>
   本体内容( <when> 和 <otherwise> )
</c:choose>
```

属性

无

限制

<c:choose>的本体内容只能有：

· 空白
· 1 或多个 <c:when>
· 0 或多个 <c:otherwise>

说明

若使用<c:when>和<c:otherwise>来做流程控制时，两者都必须为<c:choose>的子标签，即：

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

● **<c:when>**

<c:when> 的用途就和我们一般在程序中用的 when 一样。

语法

```
<c:when test="testCondition" >
本体内容
</c:when>
```

属性

名 称	说 明	EL	类型	必须	默认值
test	如果表达式的结果为 true，则执行本体内容，false 则相反	Y	boolean	是	无

限制

'<c:when> 必须在 <c:choose> 和 </c:choose>之间
'在同一个 <c:choose> 中时，<c:when> 必须在 <c:otherwise> 之前

说明

<c:when>必须有 test 属性，当 test 中的表达式结果为 true 时，则会执行本体内容；如果为 false 时，则不会执行。

● **<c:otherwise>**

在同一个 <c:choose> 中，当所有 <c:when> 的条件都没有成立时，则执行 <c:otherwise> 的本体内容。

语法

```
<c:otherwise>
本体内容
</c:otherwise>
```

属性

无

限制

· <c:otherwise> 必须在 <c:choose> 和 </c:choose>之间
· 在同一个 <c:choose> 中时，<c:otherwise> 必须为最后一个标签

说明

在同一个 <c:choose> 中，假若所有 <c:when> 的 test 属性都不为 true 时，则执行 <c:otherwise> 的本体内容。

范例

笔者举一个典型的 <c:choose>、<c:when>和<c:otherwise>范例：

```
<c:choose>

   <c:when test="${condition1}">
   condition1 为 true
   </c:when>

   <c:when test="${ condition2}">
   condition2 为 true
   </c:when>

   <c:otherwise>
   condition1 和 condition2 都为 false
   </c:otherwise>

</c:choose>
```

范例说明：当condition1为true时，会显示 “ condition1为true ”；当condition1为false且condition2为true时，会显示 “ condition2为true ”，如果两者都为false，则会显示 “ condition1和condition2都为false ”。

注意
假若 condition1 和 condition2 两者都为 true 时，此时只会显示"condition1 为 true"，这是因为在同一个<c:choose>下，当有好几个<c:when>都符合条件时，只能有一个<c:when>成立。

7-2-3 迭代操作

迭代(Iterate)操作主要包含两个标签：<c:forEach>和<c:forTokens>，笔者依此顺序依次说明这两个标签的使用。

● **<c:forEach>**

<c:forEach> 为循环控制，它可以将集合(Collection)中的成员循序浏览一遍。运作方式为当条件符合时，就会持续重复执行<c:forEach>的本体内容。

语法

语法1：迭代一集合对象之所有成员

```
<c:forEach [var="varName"] items="collection" [varStatus="varStatusName"]
        [begin="begin"] [end="end"] [step="step"]>
本体内容
</c:forEach>
```

语法2：迭代指定的次数

```
<c:forEach [var="varName"] [varStatus="varStatusName"] begin="begin"
        end="end" [step="step"]>
本体内容
</c:forEach>
```

属性

名 称	说 明	EL	类型	必须	默认值
var	用来存放现在指到的成员	N	String	否	无
items	被迭代的集合对象	Y	Arrays Collection Iterator Enumeration Map String	否	无
varStatus	用来存放现在指到的相关成员信息	N	String	否	无
begin	开始的位置	Y	int	否	0
end	结束的位置	Y	int	否	最后一个成员
step	每次迭代的间隔数	Y	int	否	1

限制

· 假若有 begin 属性时，begin 必须大于等于 0
· 假若有 end 属性时，必须大于 begin

· 假若有 step 属性时，step 必须大于等于 0

Null 和 错误处理

· 假若 items 为 null 时，则表示为一空的集合对象
· 假若 begin 大于或等于 items 时，则迭代不运算

说明

如果要循序浏览一个集合对象，并将它的内容显示出来，就必须有 items 属性。

范例

下面的范例 *Core_forEach.jsp* 是将数组中的成员一个个显示出来的：

■ *Core_forEach.jsp*

```
<%@ page contentType="text/html;charset=GB2312 " %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> 的用法" /></h2>

<%
    String atts[] = new String [5];
    atts[0]="hello";
    atts[1]="this";
    atts[2]="is";
    atts[3]="a";
    atts[4]="pen";
    request.setAttribute("atts", atts);
%>

<c:forEach items="${atts}" var="item" >
${item}</br>
</c:forEach>

</body>
</html>
```

在上述范例中，笔者先产生一个字符串数组，然后将此数组 atts 储存至 Request 的属性范围中，再用<c:forEach>将它循序浏览一遍。这里 items 表示被浏览的集合对象，var 用来存放指定的集合对象中成员，最后使用<c:out>将 item 的内容显示出来，执行结果如图 7-8 所示。

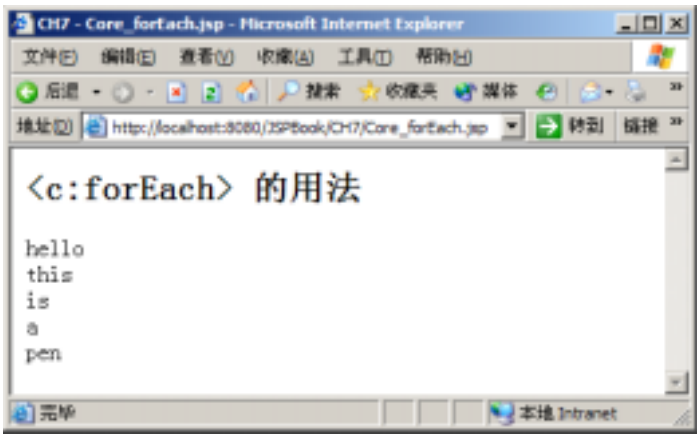

图 7-8 Core_forEach.jsp 的执行结果

注意
varName 的范围只存在<c:forEach>的本体中，如果超出了本体，则不能再取得 varName 的值。上个例子中，若${item} 是在</c:forEach>之后执行时，如： `<c:forEach items="${atts}" var="item" > </c:forEach> ${item}</br>` ${item}则不会显示 item 的内容。

<c:forEach>除了支持数组之外，还有标准 J2SE 的集合类型，例如：ArrayList、List、LinkedList、Vector、Stack 和 Set 等等；另外还包括 java.util.Map 类的对象，例如：HashMap、Hashtable、Properties、Provider 和 Attributes。

<c:forEach>还有 begin、end 和 step 这三种属性：begin 主要用来设定在集合对象中开始的位置(注意：第一个位置为 0)；end 用来设定结束的位置；而 step 则是用来设定现在指到的成员和下一个将被指到成员之间的间隔。我们将之前的范例改成如下：

■ *Core_forEach1.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach1.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> begin、end 和 step 的用法" /></h2>

<%
```

```
    String atts[] = new String [5];
    atts[0]="hello";
    atts[1]="this";
    atts[2]="is";
    atts[3]="a";
    atts[4]="pen";
    request.setAttribute("atts", atts);
%>

<c:forEach items="${atts}" var="item" begin="1" end="4" step="2" >
${item}</br>
</c:forEach>

</body>
</html>
```

<c:forEach>中指定的集合对象 atts 将会从第 2 个成员开始到第 5 个成员，并且每执行一次循环都会间隔一个成员浏览。因此结果是只显示 atts[1]和 atts[3]的内容，如图 7-9 所示。

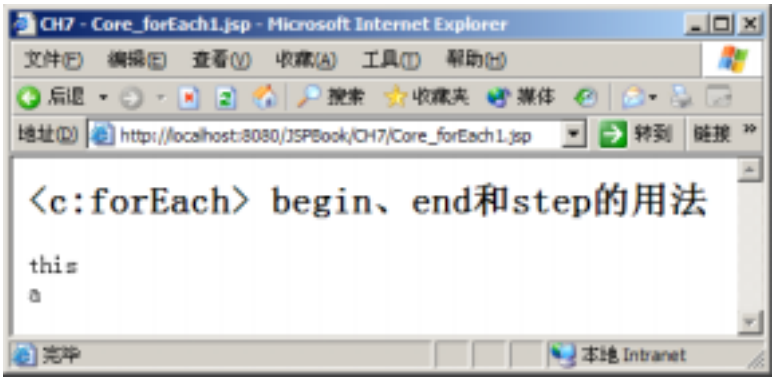

图 7-9 Core_forEach1.jsp 的执行结果

为了方便详细介绍 begin、end 和 step 的不同设定下所产生的结果，笔者将上面的范例改成如下：

```
<%
    int atts[] = {1,2,3,4,5,6,7,8,9,10};
    request.setAttribute("atts", atts);
%>
<c:forEach items="${atts}" var="item" begin="0" end="9" step="1" >
${item}</br>
</c:forEach>
```

这里笔者改变 begin、end 和 step 的值时，在网页上输出结果的变化如表 7-4。

表 7-4

begin	end	step	结 果
-	-	-	1 2 3 4 5 6 7 8 9 10
5	-	-	6 7 8 9 10
-	5	-	1 2 3 4 5 6
-	-	5	1 6
5	5	-	6
5	5	5	6
0	8	2	1 3 5 7 9
0	8	3	1 4 7
0	8	4	1 5 9
15	20	-	无
20	8	-	空白结果
0	20	-	1 2 3 4 5 6 7 8 9 10

从表 7-4 中可以发现：

(1) 当 begin 超过 end 时将会产生空的结果；
(2) 当 begin 虽然小于 end 的值，但是当两者都大过容器的大小时，将不会输出任何东西；
(3) 最后如果只有 end 的值超过集合对象的大小，则输出就和没有设定 end 的情况相同；
(4) <c:forEach>并不只是用来浏览集合对象而已，读者可以从表 7-4 中发现，items 并不是一定要有的属性，但是当没有使用 items 属性时，就一定要使用 begin 和 end 这两个属性。下面为一个简单的范例：

■ *Core_forEach2.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach2.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> 循环" /></h2>

<c:forEach begin="1" end="10" var="item" >
${item}</br>
</c:forEach>

</body>
</html>
```

上述范例中，我们并没有执行浏览集合对象，只是设定 begin 和 end 属性的值，这样它就变成一个普通的循环。此范例是将循环设定为：从 1 开始跑到 10，总共会重复循环 10 次，并

且将数字放到 item 的属性当中。*Core_forEach2.jsp* 的执行结果如图 7-10 所示。

图 7-10 Core_forEach2.jsp 的执行结果

当然它也可以搭配 step 使用，如果将 step 设定为 2，结果如图 7-11 所示。

图 7-11 当 step 设定为 2 时的结果

另外，<c:forEach>还提供 varStatus 属性，主要用来存放现在指到之成员的相关信息。例如：我们写成 varStatus="s"，那么将会把信息存放在名称为 s 的属性当中。varStatus 属性还有另外四个属性：index、count、first 和 last，它们各自代表的意义如表 7-5：

表 7-5

属　性	类　型	意　义
index	number	现在指到成员的索引
count	number	总共指到成员的总数
first	boolean	现在指到的成员是否为第一个成员
last	boolean	现在指到的成员是否为最后一个成员

我们可以使用 varStatus 属性取得循环正在浏览之成员的信息，下面为一个简单的范例：

■ *Core_forEach3.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach3.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> varStatus 的四种属性" /></h2>

<%
    String atts[] = new String [5];
    atts[0]="hello";
    atts[1]="this";
    atts[2]="is";
    atts[3]="a";
    atts[4]="pen";
    request.setAttribute("atts", atts);
%>
<c:forEach items="${atts}" var="item"varStatus="s">
<h2><c:out value="${item}"/>的四种属性：</h2>
index：${s.index}</br>
count：${s.count}</br>
first：${s.first}</br>
last：${s.last}</br>
</c:forEach>

</body>
</html>
```

执行结果如图 7-12 所示。

● **<c:forTokens>**

<c:forTokens> 用来浏览一字符串中所有的成员，其成员是由定义符号(delimiters)所分隔的。

语法

```
<c:forTokens items="stringOfTokens" delims="delimiters" [var="varName"]
   [varStatus="varStatusName"] [begin="begin"] [end="end"] [step="step"]>
本体内容
</c:forTokens>
```

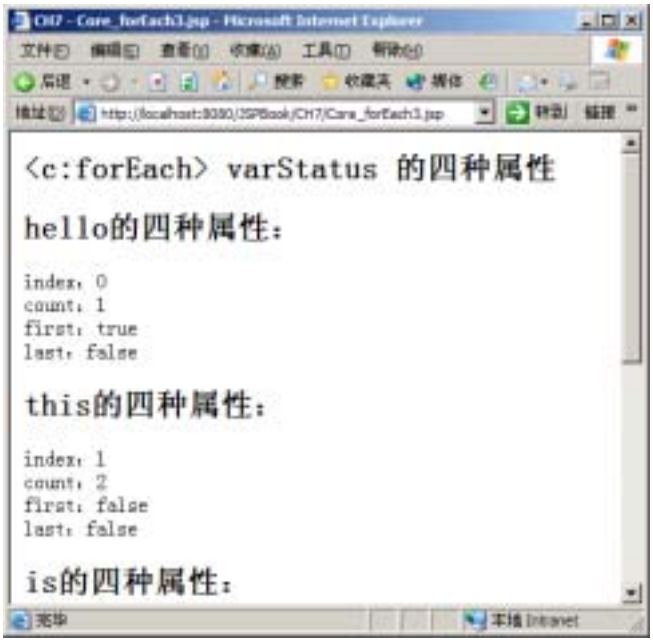

图 7-12 Core_forEach3.jsp 的执行结果

属性

名 称	说 明	EL	类 型	必 须	默认值
var	用来存放现在指到的成员	N	String	否	无
items	被迭代的字符串	Y	String	是	无
delims	定义用来分割字符串的字符	N	String	是	无
varStatus	用来存放现在指到的相关成员信息	N	String	否	无
begin	开始的位置	Y	int	否	0
end	结束的位置	Y	int	否	最后一个成员
step	每次迭代的间隔数	Y	int	否	1

限制

· 假若有 begin 属性时，begin 必须大于等于 0
· 假若有 end 属性时，必须大于 begin
· 假若有 step 属性时，step 必须大于等于 0

Null 和 错误处理

· 假若 items 为 null 时，则表示为一空的集合对象
· 假若 begin 大于或等于 items 的大小时，则迭代不运算

说明

<c:forTokens>的 begin、end、step、var 和 varStatus 用法都和<c:forEach>一样，因此，笔者在这里就只介绍 items 和 delims 两个属性：items 的内容必须为字符串；而 delims 是用来分割 items 中定义的字符串之字符。

范例

下面为一个典型的<c:forTokens>的范例：

```
<c:forTokens items="A,B,C,D,E" delims="," var="item" >
${item}
</c:forTokens>
```

上面范例执行后，将会在网页中输出 ABCDE。它会把符号“ , ”当做分割的标记，拆成 5 个部分，也就是执行循环 5 次，但是并没有将 A,B,C,D,E 中的“ , ”显示出来。items 也可以放入 EL 的表达式，如下：

```
<%
   String phoneNumber = "123-456-7899";
   request.setAttribute("userPhone", phoneNumber);
%>
<c:forTokens items="${userPhone}" delims="-" var="item" >
${item}
</c:forTokens>
```

这个范例将会在网页上打印 1234567899，也就是把 123-456-7899 以“ - ”当做分割标记，将字符串拆为 3 份，每执行一次循环就将浏览的部分放到名称为 item 的属性当中。delims 不只指定一种字符来分割字符串，它还可以一次设定多个分割字符串用的字符。如下面这个范例：

```
<c:forTokens items="A,B;C-D,E" delims=",;-" var="item" >
${item}
</c:forTokens>
```

此范例会在网页输出 ABCDE，也就是说，delims 可以一次设定所有想当做分割字符串用的字符。其实用<c:forEach>也能做到分割字符串，写法如下：

```
<c:forEach items="A,B,C,D,E" var="item" >
${item}
</c:forEach>
```

上述范例同样也会在网页输出 ABCDE。<c:forEach>并没有 delims 这个属性，因此<c:forEach>无法设定分割字符串用的字符，而<c:forEach>分割字符串用的字符只有“ , ”，这和使用<c:forTokens>，delims 属性设为“ , ”的结果相同。所以如果使用<c:forTokens>来分割字符串，功能和弹性上会比使用<c:forEach>来得较大。

7-2-4 URL 操作

JSTL 包含三个与 URL 操作有关的标签，它们分别为：<c:import>、<c:redirect>和<c:url>。它们主要的功能是：用来将其他文件的内容包含起来、网页的导向，还有 url 的产生。笔者将

依序介绍这三个标签。

● **<c:import>**

<c:import> 可以把其他静态或动态文件包含至本身 JSP 网页。它和 JSP Action 的<jsp:include>最大的差别在于：<jsp:include>只能包含和自己同一个 web application 下的文件；而<c:import>除了能包含和自己同一个 web application 的文件外，亦可以包含不同 web application 或者是其他网站的文件。

语法

语法1：

```
<c:import url="url" [context="context"] [var="varName"]
      [scope="{page|request|session|application}"]
      [charEncoding="charEncoding"]>
本体内容
</c:import>
```

语法2：

```
<c:import url="url" [context="context"] varReader="varReaderName"
      [charEncoding="charEncoding"]>
本体内容
</c:import>
```

属性

名　称	说　明	EL	类　型	必须	默认值
url	一文件被包含的地址	Y	String	是	无
context	相同 Container 下，其他 web 站台必须以“ / ”开头	Y	String	否	无
var	储存被包含的文件的内容(以 String 类型存入)	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page
charEncoding	被包含文件之内容的编码格式	Y	String	否	无
varReader	储存被包含的文件的内容(以 Reader 类型存入)	N	String	否	无

Null 和 错误处理

· 假若 url 为 null 或空时，会产生 JspException

说明

首先<c:import>中必须要有 url 属性，它是用来设定被包含网页的地址。它可以为绝对地址或是相对地址，使用绝对地址的写法如下：

```
<c:import url="http://java.sun.com" />
```

<c:import>就会把 http://java.sun.com 的内容加到网页中。

另外<c:import>也支持 FTP 协议，假设现在有一个 FTP 站台，地址为 ftp.javaworld.com.tw，它里面有一个文件 *data.txt*，那么可以写成如下方式将其内容显示出来：

```
<c:import url="ftp://ftp.cse.yzu.edu.tw/data.txt" />
```

如果是使用相对地址，假设存在一个文件名为 *Hello.jsp*，它和使用<c:import>的网页存在于同一个 *webapps* 的文件夹时，<c:import>的写法如下：

```
<c:import url="Hello.jsp" />
```

如果以“ /”开头，那么就表示跳到 web 站台的根目录下，以 Tomcat 为例，即 *webapps* 目录。假设一个文件为 *hello.txt*，存在于 *webapps/examples/images* 里，而 context 为 examples，可以写成以下方式将 *hello.txt* 文件包含进我们的 JSP 页面之中：

```
<c:import url="images/hello.txt" />
```

接下来如果要包含在同一个服务器上，但并非同一个 web 站台的文件时，就必须加上 context 属性。假设此服务器上另外还有一个 web 站台，名为 others，others 站台底下有一个文件夹为 *jsp*，且里面有 *index.html* 这个文件，那么就可以写成如下方式将此文件包含进来：

```
<c:import url="/jsp/index.html" context="/others" />
```

注意

被包含文件的 web 站台必须在 server.xml 中定义过，且<Context>的 crossContext 属性值必须为 true，这样一来，others 目录下的文件才可以被其他 web 站台调用。
server.xml 的设定范例如下：
`:`
`<Context path="/others" docBase="others" debug="0"`
`reloadable="true" crossContext="true"/>`
`:`

除此之外，<c:import>也提供 var 和 scope 属性。当 var 属性存在时，虽然同样会把其他文件的内容包含进来，但是它并不会输出至网页上，而是以 String 的类型储存至 varName 中。scope 则是设定 varName 的范围。储存之后的数据，我们在需要用时，可以将它取出来，代码如下：

```
<c:import url="/images/hello.txt" var="s" scope="session" />
```

我们可以把常重复使用的商标、欢迎语句或者是版权声明，用此方法储存起来，想输出在网页上时，再把它导入进来。假若想要改变文件内容时，可以只改变被包含的文件，不用修改其他网页。

另外，可以在<c:import>的本体内容中使用<c:param>，它的功用主要是：可以将参数传

递给被包含的文件，它有两个属性 name 和 value，如表 7-6 所示：

表 7-6

名 称	说 明	EL	类 型	必 须	默认值
name	参数名称	Y	String	是	无
value	参数的值	Y	String	否	本体内容

这两个属性都可以使用 EL，所以我们写成如下形式：

```
<c:import url="http://java.sun.com" >
<c:param name="test" value="1234" />
</c:import>
```

这样的做法等于是包含一个文件，并且所指定的网址会变成如下：

```
http://java.sun.com?test=1234
```

范例

下面为一用到<c:import>、<c:param>及属性范围的范例，*Core_import.jsp* 和 *Core_imported.jsp*：

■ *Core_import.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_import.jsp</title>
</head>
<body>

<h2><c:out value="<c:import> 的用法" /></h2>


<c:set var="input1" value="使用属性范围传到 Core_imported.jsp 中"
  scope="request"/>包含 core_imported.jsp 中<hr/>

<c:import url="Core_imported.jsp"charEncoding="GB2312" >
<c:param name="input2" value="使用<c:param>传到 Core_imported.jsp 中"/>
</c:import><hr/>

${output1}

</body>
</html>
```

程序中，笔者分别使用<c:set>和<c:param>来传递参数。

■ *Core_imported.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>CH7 - Core_imported.jsp</title>
</head>
<body>

<fmt:requestEncoding value="GB2312" />
<c:set var="output1" value="使用属性范围传到 Core_import.jsp 中"
scope="request"/>
${input1}</br>
<c:out value="${param.input2}" escapeXml="true" />

</body>
</html>
```

Core_imported.jsp 是被包含的文件，它会把从 *Core_import.jsp* 传来的参数分别输出到页面上，必须注意的是 input1 参数是使用属性范围来传递的，因此可以直接用${input1}来得到参数，而 input2 则必须使用${param.input2}来得到参数。

此外，笔者还使用<c:set>来传递值给 *Core_import.jsp*，这就是<c:param>无法做到的动作，<c:param>只能从包含端抛给被包含端，但是在属性范围中，可以让包含端也能得到被包含端传来的数据。*Core_import.jsp* 的执行结果如图 7-13 所示：

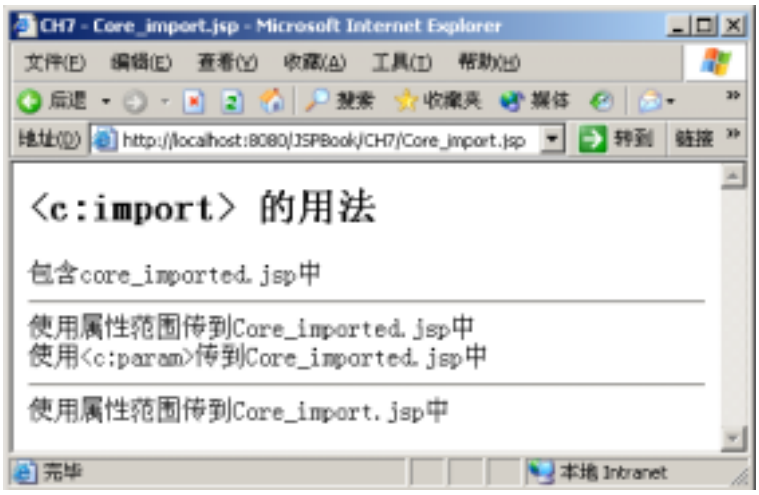

图 7-13 Core_import.jsp 的执行结果

● **<c:url>**

<c:url>主要用来产生一个 URL。

语法

语法1：没有本体内容

```
<c:url value="value" [context="context"] [var="varName"]
       [scope="{page|request|session|application}"] />
```

语法2：本体内容代表查询字符串(Query String)参数

```
<c:url value="value" [context="context"] [var="varName"]
     [scope="{page|request|session|application}"] >
  <c:param> 标签
</c:url>
```

属性

名称	说　明	EL	类型	必 须	默认值
value	执行的 URL	Y	String	是	无
context	相同 Container 下，其他 web 站台必须以 “/” 开头	Y	String	否	无
var	储存被包含文件的内容(以 String 类型存入)	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

在这里笔者直接使用例子来说明。

```
<c:url value="http://www.javaworld.com.tw " >
<c:param name="param" value="value"/>
</c:url>
```

读者可以发现<c:url>也可以搭配<c:param>使用，上面执行结果将会产生一个网址为 http://www.javaworld.com.tw?param=value，我们更可以搭配 HTML 的<a>使用，如下：

```
<a href="
   <c:url value="http://www.javaworld.com.tw " >
   <c:param name="param" value="value"/>
</c:url>">台湾 Java 技术论坛</a>
```

另外<c:url>还有三个属性，分别为 context、var 和 scope。context 属性和之前的<c:import>相同，可以用来产生一个其他 web 站台的网址。如果<c:url>有 var 属性时，则网址会被存到 varName 中，而不会直接输出网址。

哪些状况下才会去使用<c:url>？例如：当我们须动态产生网址时，有可能传递的参数不固定，或者是需要一个网址能连至同服务器的其他 web 站台之文件，而且<c:url>更可以将产生的网址储存起来重复使用。另外，在以前我们必须使用相对地址或是绝对地址去取得需要的图文件或文件，现在我们可以直接利用<c:url>从 web 站台的角度来设定需要的图文件或文件的地址，如下：

```
<img src="<c:url value="/images/code.gif" />" />
```

如此就会自动产生连到 *image* 文件夹里的 *code.gif* 的地址，不再须耗费精神计算相对地址，并且当网域名称改变时，也不用再改变绝对地址。

● **<c:redirect>**

<c:redirect>可以将客户端的请求从一个 JSP 网页导向到其他文件。

语法

语法1：没有本体内容

```
<c:redirect url="url" [context="context"] />
```

语法2：本体内容代表查询字符串(Query String)参数

```
<c:redirect url="url" [context="context"] >
  <c:param>
</c:redirect >
```

属性

名 称	说 明	EL	类 型	必须	默认值
url	导向的目标地址	Y	String	是	无
context	相同 Container 下，其他 web 站台必须以 “/” 开头	Y	String	否	无

说明

url 就是设定要被导向到的目标地址，它可以是相对或绝对地址。例如：我们写成如下：

```
<c:redirect url="http://www.javaworld.com.tw" />
```

那么网页将会自动导向到 http://www.javaworld.com.tw。另外，我们也可以加上 context 这个属性，用来导向至其他 web 站台上的文件，例如：导向到 */others* 下的 */jsp/index.html* 时，写法如下：

```
<c:redirect url="/jsp/index.html" context="/others" />
```

<c:redirect> 的功能不止可以导向网页，同样它还可以传递参数给目标文件。在这里我们同样使用<c:param>来设定参数名称和内容。

范例

■ *Core_redirect.jsp*

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
```

```
<head>
  <title>CH7 - Core_redirect.jsp</title>
</head>
<body>

<h2><c:out value="<c:redirect> 的用法" /></h2>

<c:redirect url="http://java.sun.com">
<c:param name="param" value="value"/>
</c:redirect>

<c:out value="不会执行喔!!!" />

</body>
</html>
```

执行结果如图 7-14，可以发现网址部分为 http://java.sun.com/?param=value：

图 7-14 Core_redirect.jsp 的执行结果