9

使用 JSTL 與自訂標籤

學習目標：

- 了解何謂 JSTL
- 使用 JSTL 核心標籤庫
- 了解如何使用 Tag File 自訂標籤
- 了解如何使用 Simple Tag 自訂標籤
- 了解如何使用 Tag 自訂標籤

9.1 使用 JSTL

JSP 提供了標準標籤及 EL，可用以減少 JSP 頁面上的 Scriplet 數量，將請求處理與商務邏輯封裝至 Servlet 或 JavaBean 中，網頁中僅留下與頁面相關的呈現邏輯。然而即使只留下頁面邏輯，網頁中夾雜著 Scriptlet，依舊使得網頁設計及維護困難重重。

你可以使用 **JSTL(JavaServer Pages Standard Tag Library)**來取代 JSP 頁面中用來實現頁面邏輯的 Scriptlet，讓網頁上與類似 HTML 的標籤，這會使得設計網頁簡單多了，可以隨時調整畫面而不用費心的修改 Scriplet。

9.1.1 簡介 JSTL

在 Servlet 中撰寫 HTML 進行頁面輸出當然是件麻煩的事，第 8 章學過 JSP 後，你終於可以放手在 JSP 中寫 HTML。然而，在 JSP 中寫 Scriptlet 放入 Java 程式碼也不是什麼好事，跟 Servlet 中撰寫 HTML 其實是件半斤八兩的事。

如果你使用 Model 2，你可以將請求處理放到 Servlet，將商務邏輯放到純粹 Java 物件中，在 JSP 上則使用 EL 來取得值並顯示。如果你使用 Model 1，你也可以將商務邏輯放到 JavaBean 中，在 JSP 上透過<jsp:useBean>、<jsp:setProperty>、<jsp:getProperty>等標籤以及來處理請求並取得結果進行顯示。

然而就目前你所學到的，無論如何有些東西你還是得在 Scriptlet 中撰寫 Java 程式碼，才可以讓畫面呈現你想要的結果。例如，需要依某個條件來決定是否顯示某個網頁片段，或是需要使用迴圈來顯示表格內容。然而，HTML 或 JSP 本身並沒有什麼<if>標籤，更沒什麼<for>標籤讓你達到這個目的。

所幸這些跟頁面呈現相關的邏輯判斷標籤是存在的，由 Java EE 5 平台中的 JSTL 1.2 提供，它不僅提供了條件判斷的邏輯標籤，還提供了對應 JSP 標準標籤的功能擴充標籤以及更多的功能標籤。基本上，JSTL 提供的標籤庫分作五個大類：

- 核心標籤庫
  提供條件判斷、屬性存取、URL 處理及錯誤處理等標籤。本章會針對核心標籤庫的功能與作用進行說明。
- 格式標籤庫
  提供數字、日期等的格式化功能，以及區域（Locale）、訊息、編碼處理等國際化功能的標籤。
- SQL 標籤庫
  提供基本的資料庫查詢、更新、設定資料源（DataSource）等功能之標籤。
- XML 標籤庫

提供 XML 剖析、流程控制、轉換等功能之標籤。

- 函式標籤庫

  提供常用字串處理的自訂 EL 函式標籤庫。

JSTL 本身並非在 JSP 的規範當中，所以必須另外下載 JSTL 實作：

```
http://java.sun.com/products/jsp/jstl/
```

可以透過上面這個網頁找到 JSTL 的相關下載與 API 文件說明。如果想要直接下載 JSTL，則可以在這個網址找到：

```
https://jstl.dev.java.net/
```

下載了 JSTL 實作（封裝好的 JAR 檔案）之後，必須放置到 Web 應用程式的 WEB-INF/lib 資料夾中，JSTL 1.2 實作的檔案名稱是 **jstl-impl-1.2.jar**。如果需要 API 文件說明，則可以在這個網址找到：

```
http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/
```

提示 如果你使用 Tomcat 作為 Web 容器，在 Tomcat 的範例 webapps\examples 中的 WEB-IN\lib，可以找到 JSTL 1.1，有兩個檔案 **jstl.jar** 與 **standard.jar**，同樣也是複製到 Web 應用程式的 WEB-INF/lib 之中。JSTL 1.2 與 1.1 的差別主要在於，1.2 對 JSF 的支援更完整。就標籤庫本身的功能而言，1.2 與 1.1 兩者並沒有什麼差別。JSTL 1.2 必須在支援 Servlet 2.5 的容器上才可以使用，1.1 則只要在支援 Servlet 2.4 的容器就可以使用。

JSTL 的標籤種類也蠻多的，要完整說明 JSTL 所有標籤也已超出本書範圍。本書將只說明 JSTL 核心標籤庫。其它的標籤庫則請參考 JSTL 文件說明或專書。要使用 JSTL 核心標籤庫，必須在 JSP 網頁上，使用 `taglib` 指示元素定義前置文件與 `uri` 參考。

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

前置文件設定了這個標籤庫在此 JSP 網頁中的名稱空間，以避免與其它標籤庫的標籤名稱發生衝突，慣例上使用 JSTL 核心標籤庫時，會使用 `c` 作為前置名稱。`uri` 參考則告知容器，如何參考 JSTL 標籤庫實作（如 8.3.5 節定義 TLD 時的作用，可先參考該節內容，9.2 說明自訂標籤時還會看到相關說明）。

注意 如果你必須使用 JSTL 1.0（適用於 JSP 1.2、J2EE 1.3 環境），除了要將 jstl.jar 與 standard.jar 複製至 WEB- INF/lib/，還需複製 TLD 檔案，並於 web.xml 中設定 TLD 檔案的位置。例如要使用核心標籤庫的話，需在 web.xml 中設定：

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tlds/c.tld</taglib-uri>
```

```
</taglib>
```

注意 uri 名稱與 JSTL 1.1 之後不一樣。在 JSP 網頁上，同樣也要使用 taglib 指示元素定義前置文字與 uri。

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

9.1.2 流程處理標籤

當 JSP 網頁必須根據某個條件來安排網頁輸出時，則可以使用流程標籤。例如想要依使用者輸入的名稱、密碼請求參數，來決定是否顯示某個畫面，或是想要用表格輸出十筆資料等。

首先介紹**<c:if>**標籤的使用（假設標籤前置使用"c"），這個標籤讓你根據某個運算式的結果，決定是否顯示本體內容。直接來看個範例：

JSTLDemo login.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
        <title>登入頁面</title>
    </head>
    <body>
       <c:if test="${param.name == 'caterpillar' && param.password == '123456'}">
            <h1>${param.name} 登入成功</h1>
       </c:if>
    </body>
</html>
```

<c:if>標籤的 test 屬性中可以放置 EL 運算式或 JSP 運算元素（<%= %>），如果運算式的結果是 true，則會將<c:if>本體輸出。就上面這個範例來說，如果使用者發送的請求參數中，使用者名稱與密碼正確，就會顯示使用者名稱與登入成功的訊息。

提示 為免流於語法說明的瑣碎細節，本節不會試圖說明 JSTL 每個標籤上所有屬性的作用，這些屬性基本上都不難，你可以在需要的時候，參考 JSTL 的線上文件說明。

<c:if>標籤僅在 test 的結果為 true 時顯示本體內容，但沒有相對應的<c:else>標籤。如果你想要在某條件式成立時顯示某些內容，否則就顯示另一個內容，則可以使用**<c:choose>**、**<c:when>**及**<c:otherwise>**標籤。同樣以實例來說明：

JSTLDemo login2.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<jsp:useBean id="user" class="cc.openhome.User"  />
<jsp:setProperty name="user" property="*" />
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>登入頁面</title>
    </head>
    <body>
        <c:choose>
            <c:when test="${user.valid}">
                <h1>
                    <jsp:getProperty name="user" property="name"/>登入成功
                </h1>
            </c:when>
            <c:otherwise>
                 <h1>登入失敗</h1>
            </c:otherwise>
        </c:choose>
    </body>
</html>
```

這個範例改寫自 8.2.2 節的使用者登入網頁範例，在 8.2.2 節時，為了依使用者是否發送正確名稱密碼來顯示登入成功或失敗的畫面，不可避免地使用了 Scriptlet 撰寫 Java 程式碼來作判斷。在學到<c:choose>、<c:when>及<c:otherwise>標籤之後，就可以不使用 Scriptlet 而實現這個需求。

<c:when>及<c:otherwise>必須放在<c:choose>之中。當<c:when>的 test 運算結果為 true 時，會輸出<c:when>的本體內容，而不理會<c:otherwise>的內容。<c:choose>中可以有多個<c:when>標籤，此時會從上往下進行測試，如果有個<c:when>標籤的 test 運算結果為 true 就輸出

其本體內容，而後來離開<c:choose>（就不再繼續往下測試），如果所有<c:when>測試都不成立，則會輸出<c:otherwise>的內容。

如果你打算使用迴圈來產生一連串的資料輸出。例如你有個簡單的留言版程式，使用 JavaBean 從資料庫中取得留言，留言可能有數十則，會以陣列方式傳回：

JSTLDemo MessageService.java

```
package cc.openhome;

public class MessageService {
    private Message[] fakeMessages;

    public MessageService() {
        // 放些假資料，假裝這些資料是來自資料庫
        fakeMessages = new Message[3];
        fakeMessages[0] = new Message("caterpillar", "caterpillar's message!");
        fakeMessages[1] = new Message("momor", "momor's message!");
        fakeMessages[2] = new Message("hamimi", "hamimi's message!");
    }

    public Message[] getMessages() {
        return fakeMessages;
    }
}
```

Message 物件有 name 與 text 屬性，分別表示留言者名稱與留言文字。你打算在網頁上使用表格來顯示每一則留言。若不想使用 Scriplet 撰寫 Java 程式碼的 for 迴圈，則可以使用 JSTL 的**<c:forEach>**標籤來實現這項需求。例如：

JSTLDemo message.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<jsp:useBean id="messageService"
             class="cc.openhome.MessageService"/>
<html>
    <head>
        <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8">
```

```
        <title>留言版</title>
    </head>
    <body>
        <table style="text-align: left; width: 100%;" border="1">
            <tr>
                <td>名稱</td><td>訊息</td>
            </tr>
            <c:forEach var="message" items="${messageService.messages}">
                <tr>
                    <td>${message.name}</td><td>${message.text}</td>
                </tr>
            </c:forEach>
        </table>
    </body>
</html>
```

<c:forEach>標籤的 **items** 屬性可以是陣列或 List 物件，每次會循序取出陣列或 List 物件中的一個元素，並指定給 **var** 屬性所設定的變數，之後就可以在<c:forEach>標籤本體中，使用 var 屬性所設定的變數來取得該元素。這個範例的執行畫面如下所示：

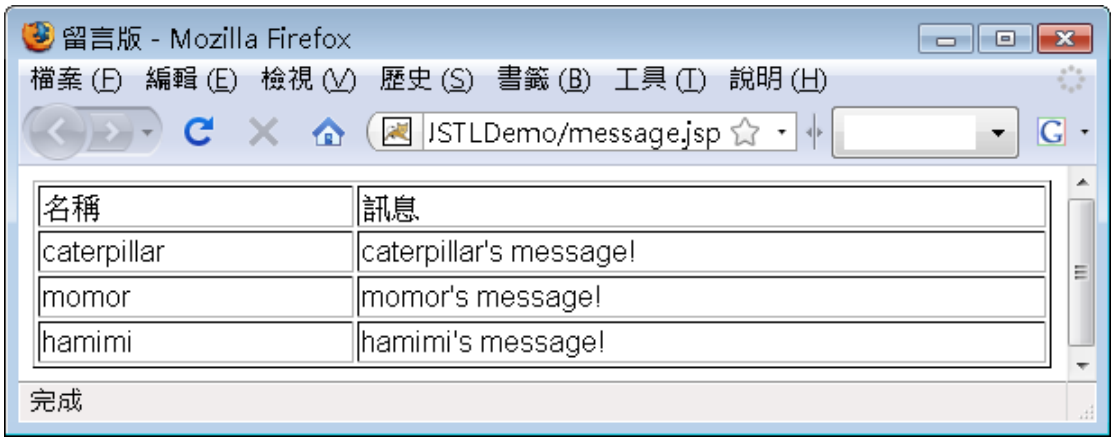


圖 9.1 <c:forEach>範例網頁執行結果

再來簡單地介紹一下**<c:forTokens>**的使用。如果你想要在 JSP 網頁上，將某個字串切割為數個字符（Token），就可以使用<c:forTokens>。例如：

```
<c:forTokens  var="token"
       delims="," items="Java,C++,C,JavaScript">
   ${token} <br>
</c:forTokens>
```

這個簡單的片段，會將"Java,C++,C,JavaScript"這個字串，依指定的 **delims** 進行切割，所以分出來的字符會分別是"Java"、"C++"、"C"與"JavaScript"四個字串。

9.1.3 錯誤處理標籤

在 8.3.1 節介紹 EL 時，曾使用一個簡單的加法網頁來示範。在那個範例中，使用了 errorPage="error.jsp"設定當錯誤發生時，轉發 error.jsp 來顯示錯誤，例如若使用者輸入的並非數字時，EL 無法進行剖析為基本型態進行加法時，就會發生錯誤，而轉發 error.jsp。

如果現在，你不想要在錯誤發生時，轉發其它網頁來顯示錯誤訊息，而打算在目前網頁捕捉例外，並顯示相關訊息，那該如何進行？

這個問題的答案似乎很簡單，撰寫 Scriptlet，在當中使用 Java 的 try-catch 語法捕捉例外。不過本書到了這一章，可以的話實在不希望再出現 Scriptlet，那該怎麼辦？

你可以使用 JSTL 的**<c:catch>**標籤。直接來看如何改寫 8.3.1 節的加法網頁，再來進行說明：

JSTLDemo add.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
        <title>加法網頁</title>
    </head>
    <body>
        <c:catch var="error">
            ${param.a} + ${param.b} = ${param.a + param.b}
        </c:catch>
        <c:if test="${error != null}">
            <br><span style="color: red;">${error.message}</span>
            <br>${error}
        </c:if>
    </body>
</html>
```

如果你要在發生例外的網頁直接捕捉例外物件，就可以使用<c:catch>將可能產生例外的網頁段落包起來。如果例外真的發生，這個例外物件會設定給 **var** 屬性所指定的名稱，這樣你才有機會使用這個例外物件。例如範例中，使用

了<c:if>標籤測試 error 是否有參考至例外物件，如果有的話，由於例外都是 Throwable 的子類別，所以擁有 getMessage() 方法，因此才能透過 ${error.message}的方式取得例外相關訊息。

注意　只有設定 isErrorPage="true"的 JSP 網頁才會有 exception 隱含物件，代表錯誤發生的來源網頁所傳進來的 Throwable 物件，所以你不可以在上面的範例中，直接使用 exception 隱含物件。

這個範例執行時如果發生例外，結果畫面如下所示：

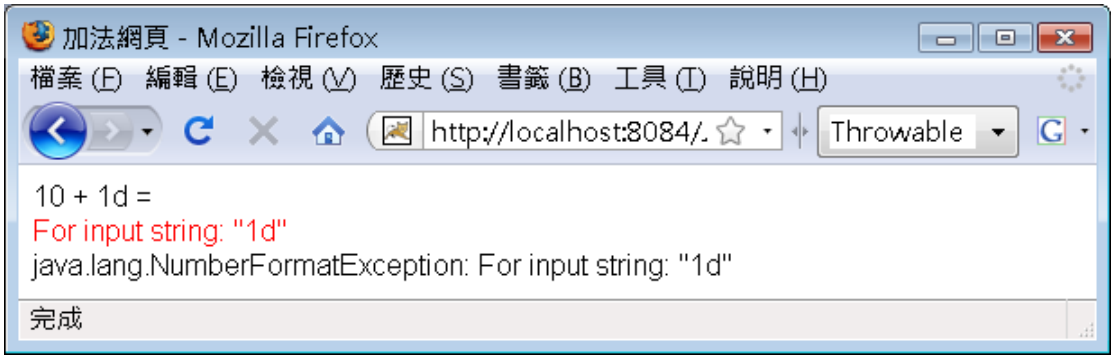


圖 9.2 <c:catch>範例網頁執行結果

9.1.4 網頁匯入、重新導向標籤

到目前為止，你學過了兩種包括其它 JSP 網頁至目前網頁的方式。一個是透過 include 指示元素，在轉譯時期直接將另一 JSP 網頁合併至目前網頁進行轉譯，如 8.1.2 節所示範過的範例：

```
<%@include file="/WEB-INF/jspf/header.jspf"%>
```

另一個方式是透過<jsp:include>標籤，可於執行時期依條件，動態決定是否包括另一個網頁，該網頁執行完畢後，再回到目前網頁，在含括另一網頁時還可以帶有參數，如 8.2.1 節所示範過的範例：

```
<jsp:include page="add.jsp">
    <jsp:param name="a" value="1" />
    <jsp:param name="b" value="2" />
</jsp:include>
```

在 JSTL 中，有個**<c:import>**標籤，可以視作是<jsp:include>的加強版，也是可以於執行時期動態匯入另一個網頁，並也可搭配**<c:param>**在匯入另一網頁時帶有參數。例如上面的<jsp:include>範例片段，也可以改寫為以下使用 JSTL 的版本：

```
<c:import url="add.jsp">
    <c:param name="a" value="1" />
    <c:param name="b" value="2" />
</c:import>
```

除了可以匯入目前 Web 應用程式中的網頁之外，<c:import>標籤還可以匯入非目前 Web 應用程式中的網頁。例如：

```
<c:import url="http://openhome" charEncoding="BIG5"/>
```

其中 **charEncoding** 屬性用來指定要匯入的網頁之編碼，如果要被匯入的網頁編碼與目前網頁編碼不同，必須使用 charEncoding 屬性加以指定，匯入的網頁才不致於產生亂碼。

再來介紹**<c:redirect>**標籤。在 Servlet／JSP 中，如果要以程式的方式進行重新導向，必須使用 HttpServletResponse 的 sendRedirect()方法，這在 3.2.2 節曾經介紹過它的用法。<c:redirect>標籤的作用，就如同 sendRedirect()方法，如此你就不用撰寫 Scriptlet 來使用 sendRedirect()方法，也可以達到重新導向的作用，如果重新導向時需要參數，也可以透過<c:param>來設定。

```
<c:redirect url="add.jsp">
    <c:param name="a" value="1"/>
    <c:param name="b" value="2"/>
</c:redirect>
```

9.1.5 屬性處理與輸出標籤

JSP 的<jsp:setProperty>功能有限，只能用來設定 JavaBean 的屬性。如果你只是要在 page、request、session、application 等範圍設定一個屬性，或者你還想要設定 Map 物件的鍵與值，則可以使用**<c:set>**標籤。

例如，若使用者登入後，你想要在 session 範圍中設定一個"login"屬性，代表使用者已經登入，則可以如下撰寫：

```
<c:set var="login" value="caterpillar" scope="session"/>
```

var 用來設定屬性名稱，而 **value** 用來設定屬性值。這段標籤設定的作用，相當於：

```
<% session.setAttribute("login", "caterpillar"); %>
```

你也可以使用 EL 來進行設定，例如：

```
<c:set var="login" value="${user}" scope="session"/>
```

如果${user}運算的結果是 User 類別的實例，則儲存的屬性就是 User 物件，也就是相當於以下這段程式碼：

```
<%
    // user 是 User 所宣告的參考名稱，參考至 User 物件
    session.setAttribute("login", user);
%>
```

<c:set>標籤也可以將 value 的設定改為本體的方式，在所要設定的屬性值冗長時，採用本體的方式會比較容易撰寫。例如：

```
<c:set var="details" scope="session">
    caterpillar,openhome.cc,caterpillar.onlyfun.net
</c:set>
```

<c:set>不設定 scope 時，則會以 page、request、session、application 的範圍尋找屬性名稱，如果在某個範圍找到屬性名稱，則在該範圍設定屬性，如果所有範圍都沒有找到屬性名稱，則會在 page 範圍中新增屬性。如果你要移除某個屬性，則可以使用**<c:remove>**標籤。例如：

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

<c:set>也可以直接用來設定 JavaBean 的屬性或是 Map 物件的鍵／值，若要設定 JavaBean 或 Map 物件，則要使用 target 屬性進行設定。例如：

```
<c:set target="${user}" property="name" value="${param.name}"/>
```

如果${user}運算出來的結果是個 JavaBean，則上例就如同呼叫 setName()並將請求參數 name 的值傳入。如果${user}運算出來的結果是個 Map，則上例就是以 **property** 屬性作為鍵，而 value 屬性作為值來呼叫 Map 物件的 put()方法。

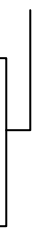
下面這個範例改寫 5.2.1 節的問卷網頁，把 Servlet 改為 JSP 實作，並且使用 JSTL 來設置屬性。

JSTLDemo question.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<c:set target="${pageContext.request}"
       property="characterEncoding" value="UTF-8"/>
<html>
   <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
      <title>Questionnaire</title>
   </head>
   <body>
      <form action="question.jsp" method="post">
```

❶設定 **request** 的字元編碼

```
            <c:choose>
                <c:when test="${param.page == 'page1'}">
                    問題一：<input type="text" name="p1q1"><br>
                    問題二：<input type="text" name="p1q2"><br>
                    <input type="submit" name="page" value="page2">
                </c:when>
                <c:when test="${param.page == 'page2'}">
                    <c:set var="p1q1"
                           value="${param.p1q1}" scope="session"/>
                    <c:set var="p1q2"
                            value="${param.p1q2}" scope="session"/>
                    問題三：<input type="text" name="p2q1"><br>
                    <input type="submit" name="page" value="finish">
                </c:when>
                <c:when test="${param.page == 'finish'}">
                    ${sessionScope.p1q1}<br>
                    ${sessionScope.p1q2}<br>
                    ${param.p2q1}<br>
                </c:when>
            </c:choose>
        </form>
    </body>
</html>
```



因為問題的答案可能是用中文填寫，為了順利取得中文，必須設定 request 的字元編碼處理方式，也就是呼叫 setCharacterEncoding()方法設定編碼。在這邊使用${pageContext.request}取得 request 物件，並透過<c:set>來進行設定❶。程式中需要判斷顯示哪些問題的部份，使用之前學習過的<c:choose>與<c:when>標籤。問卷過程中需儲存至 session 的答案，則使用<c:set>來進行設定❷。

再來介紹**<c:out>**物件，它可以讓你輸出指定的文字。例如：

```
<c:out value="${param.message}"/>
```

你也許會想！這有什麼意思？為什麼不直接寫${param.message}就好，還要加上<c:out>標籤？這不是多此一舉嗎？如果${param.message}是來自使用者於留言版所發送的訊息，而使用者故意打了 HTML 在訊息，則<c:out>會自動將角括號、單引號、雙引號等字元用替代字元取代。這個功能是由<c:out>的 **escapeXml** 屬性來控制，預設是 true，如果設定為 false，就不會作替代字元的取代。

EL 運算結果為 null 時，並不會顯示任何值，這原本是使用 EL 的好處，但

如果你希望在 EL 運算結果為 null 時，可以顯示一個預設值，就目前你所學習到的 JSTL 標籤，你可能會這麼作：

```
<c:choose>
    <c:when test="${param.a != null}">
        ${param.a}
    </c:when>
    <c:otherwise>
        0
    </c:otherwise>
</c:choose>
```

如果使用<c:out>的話，則可以更簡潔地達到這個目的：

```
<c:out value="${param.a}" defalut="0"/>
```

9.1.6 URL 處理標籤

在 5.2.3 節曾經談過使用 response 的 encodeURL() 方法來作 URL 重寫，以在使用者關閉 Cookie 功能時，仍可以繼續利用 URL 重寫來維持使用 session 進行會話管理。

如果你不想使用 Scriptlet 撰寫 response 的 encodeURL() 方法來作 URL 重寫，則可以使用 JSTL 的**<c:url>**，它會在使用者關閉 Cookie 功能時，自動用 Session ID 作 URL 重寫。例如以下的 JSP 網頁使用 JSTL 改寫 5.2.3 節的範例：

JSTLDemo count.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<c:set var="count" value="${sessionScope.count + 1}" scope="session"/>
<html>
    <head>
        <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8">
        <title>JSP Count</title>
    </head>
    <body>
        <h1>JSP Count ${sessionScope.count} </h1>
        <a href="<c:url value='count.jsp'/>">遞增</a>
    </body>
```

```
</html>
```

當你關閉瀏覽器 Cookie 功能時，這個 JSP 網頁仍可以有計數功能。一個執行的畫面如下所示：

圖 9.3 關閉 Cookie 後的<c:url>範例網頁執行結果

如果需要在 URL 上攜帶參數，則可以搭配<c:param>標籤，參數將被編碼後附加在 URL 之上。例如就以下這個片段而言，最後的 URL 將成為 some.jsp?name=Justin+Lin：

```
<c:url value="some.jsp">
    <c:param name="name" value="Justin Lin"/>
</c:url>
```

9.2 Tag File 自訂標籤

JSTL 是個標準，只要支援 Servlet／JSP 標準的 Web 容器，就可以使用 JSTL。然而也許有些需求，你無法單靠 JSTL 的標籤來完成，也許是要將既有的 HTML 元素封裝加強，或者是為了與你的應用程式更緊密地結合。例如希望有個標籤，可以直接從應用程式所自訂的物件中取出資訊，而不是透過屬性來傳遞物件或資訊。

你可以尋求自訂標籤的可能性。網路上有一些 Web 應用程式框架（Framework），為了讓使用者更簡便地取得框架的相關資訊或資源，通常會提供自己的一套自訂標籤庫。然而自訂標籤有一定的複雜度，而且自訂標籤通常會與你的應用程式產生一定程度的相依性。可以的話，先尋找現成且通用的自訂標籤實作，察看是否滿足需求會是比較好的考量。畢竟雖非標準，但如果是通用的自訂標籤庫，至少有一定數量的使用者，將來移植時可以少一些阻礙，發生問題時會較易尋找解答。

若無論如何一定要自行實作標籤庫，則本章接下來的內容，將從最簡單的 Tag File 開始介紹如何自訂標籤庫，然後進一步介紹 Simple Tag 的製作，以及最複雜的 Tag 自訂標籤。

9.2.1 簡介 Tag File

如果要自訂標籤，Tag File 是最簡單的方式，即使是不會 Java 的網頁設計人員也有能力自訂 Tag File。事實上，Tag File 本來就是為了不會 Java 的網頁設計人員而存在的。

來舉個實際的情境為例。在第 8 章綜合練習中，已經用 JSP 來實現畫面的呈現，其中新增書籤的 JSP 網頁中（add.jsp），有以下的片段：

```
<%
    List errorList = (List) request.getAttribute("errors");
    if (errorList != null) {
%>
        <ul style="color: rgb(255, 0, 0);">
<%
        Iterator<String> errors = errorList.iterator();
        while (errors.hasNext()) {
            out.println("<li>" + errors.next() + "</li>");
        }
        out.println("</ul>");
    }
%>
```

這個片段的作用，在於使用者新增書籤時沒有填寫必要欄位時而回到原網頁時，會出現相關的錯誤訊息，這些錯誤訊息是收集在一個 List 物件中，在 request 中設定 errors 屬性後傳遞過來。由於你已經學過 JSTL 了，所以可以將這個 Scriptlet 與 HTML 夾雜的片段改為：

```
<c:if test="${requestScope.errors != null}">
    <ul style="color: rgb(255, 0, 0);">
        <c:forEach var="error" items="${requestScope.errors}">
            <li>${error}</li>
        </c:forEach>
    </ul>
</c:if>
```

現在即使是網頁設計人員，也可以看懂並依需求修改這個片段了。然而，這種錯誤訊息的呈現方式，也許並不僅出現在一個網頁，其它網頁也可能需要同樣的片段。每次都得複製貼上同樣的片段也許還不成問題，但將來要修改樣式時才是一大麻煩，網頁設計人員也許會想說，這樣的片段若可以像是<html:errrors>這樣的標籤存在就好了。

如果網頁設計人員知道可以使用 Tag File，那這個需求就解決了。他們可以

撰寫一個副檔名為**.tag** 的檔案，把它們放在 **WEB-INF/tags** 底下，內容如下：

TagFileDemo Errors.tag

```
<%@tag description="顯示錯誤訊息的標籤" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<c:if test="${requestScope.errors != null}">
    <ul style="color: rgb(255, 0, 0);">
        <c:forEach var="error" items="${requestScope.errors}">
            <li>${error}</li>
        </c:forEach>
    </ul>
</c:if>
```

在這邊你看到了 **tag** 指示元素，它就像是 JSP 的 page 指示元素，用來告知容器如何轉譯這個 Tag File。description 只是一段文字描述，除了用來說明這個 Tag File 的作用之外，沒什麼其它實質用途。**pageEncoding** 屬性同樣是告知容器在轉譯時使用的編碼。Tag File 中可以使用 taglib 指示元素引用其它自訂標籤庫，所以你可以在 Tag File 中使用 JSTL。基本上，JSP 檔案中可以使用的 EL 或 Scriptlet，在 Tag File 中也可以使用。

提示 Tag File 基本上是給不會 Java 的網頁設計人員使用的，所以這邊的範例就都不會在 Tag File 中出現 Scriptlet 了。

接著只要在需要這個 Tag File 的 JSP 頁面中，使用 taglib 指示元素的 prefix 定義前置名稱，以及使用 **tagdir** 屬性定義 Tag File 的位置：

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

接著你就可以在需要呈現錯誤訊息的 JSP 頁面上，使用<html:Errors/>標籤來代替先前呈現錯誤訊息的片段。例如：

TagFileDemo add.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib  prefix="html" tagdir="/WEB-INF/tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>新增書籤</title>
        <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
    </head>
    <body>
```

定義前置與 Tag File 位置

```
        <html:Errors/>          ←—— 使用自訂的 Tag File 標籤
        <form method="post" action="add.do">
            網址 http:// <input name="url" value="${param.url}"><br>
            網頁名稱：<input name="title" value="${param.title}"><br>
            分    類：<input type="text"
                        name="category" value="${param.category}"><br>
            <input value="送出" type="submit"><br>
        </form>
    </body>
</html>
```

當然！使用這個自訂的<html:Errors/>標籤有個假設前題。錯誤訊息是收集在一個 List 物件中，在 request 中設定 errors 屬性後傳遞過來。除非是大家都公認的標準，否則自訂標籤必然與應用程式有某種程度的相依性，在自訂標籤前，於使用的方便性及相依性之間必須作出考量。

注意 注意！雖然 tagdir 可以指定 Tag File 的位置，但事實上你只能指定**/WEB-INF/tags** 的子資料夾，也就是說，以 tagdir 屬性設定的話，你的 Tag File 就只能放在/WEB-INF/tags 或子資料夾之中。

前面提過 Tag File 會被容器轉譯，實際上是轉譯為 javax.servlet.jsp.tagext.SimpleTagSupport 的子類別。以 Tomcat 為例，Errors.tag 轉譯後的類別原始碼名稱是 Errors_tag.java。在 Tag File 中可以使用 out、config、request、response、session、application、jspContext 等隱含物件，jspContext 在轉譯之後，實際上則是 javax.servlet.jsp.JspContext 物件。

提示 JspContext 是 PageContext 的父類別，JspContext 上定義的 API 不若 PageContext 使用了 Servlet API，原本的設計目的是希望 JSP 相關實現不要依賴特定技術（例如 Servlet），所以才會有 JspContext 這個父類別的存在。
當然，如果是不懂 Java 的網頁設計人員，並不用理會這些轉譯後的東西，只有那些了解如何開發 Simple Tag 的 Java 程式設計人員，才有可能對這些轉譯後的東西有興趣。

9.2.2 處理標籤屬性與本體

來考慮一個需求。網頁設計人員總必須在<header>與</header>之間加些<title>、<meta>資訊，如果今天網頁設計人員發現，Web 應用程式中的

JSP 網頁，<header>與</header>除了部份資訊不同之外（例如<title>不同），其它要加的東西都是相同的，他希望將<header>與</header>間的東西製作為 Tag File，以利之後要修改時，只需要修改 Tag File，就可以套用至全部有引用該 Tag File 的 JSP 網頁。問題在於，如何設定 Tag File 中不同的特定資訊？

答案是透過 Tag File 屬性設定。就如同 HTML 的元素都有一些屬性可以設定，你在建立 Tag File 時，也可以指定使用某些屬性，方法是透過 **attribute** 指示元素來指定，直接來看範例了解如何設定。

TagFileDemo Header.tag

```
<%@tag description="header 內容" pageEncoding="UTF-8"%>
<%@attribute name="title"%>
<head>
    <title>${title}</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
```

attribute 指示元素讓你定義使用 Tag File 時可以設定的屬性名稱，如果有多個屬性名稱，則使用多個 attribute 指示元素來設定，設定名稱之後，若有人使用 Tag File 時指定屬性值，則這個值在 Tag File 定義檔內，可以使用如範例${title}方式來取得。下面這個網頁是個使用範例。

TagFileDemo add2.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib  prefix="html" tagdir="/WEB-INF/tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <html:Header title="新增書籤"/>  ←—— 使用自訂的 Tag File 標籤
    <body>
        <html:Errors/>
        <form method="post" action="add.do">
            網址 http:// <input name="url" value="${param.url}"><br>
            網頁名稱：<input name="title" value="${param.title}"><br>
            分    類：<input type="text" name="category"
                          value="${param.category}"><br>
            <input value="送出" type="submit"><br>
        </form>
    </body>
</html>
```

目前為止，你所使用的都是沒有本體內容的 Tag File，事實上 Tag File 標籤

是可以本體內容的。舉個例子來說，如果你的 JSP 頁面中，除了<body>與</body>之間的東西是不同的之外，其它都是相同的，那麼你可以像下面的範例，撰寫一個 Tag File：

TagFileDemo Html.tag

```
<%@tag description="HTML 懶人標籤" pageEncoding="UTF-8"%>
<%@attribute name="title"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>${title}</title>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
    </head>
    <body>
        <jsp:doBody/>
    </body>
</html>
```

這個 Tag File 使用 attribute 指示元素宣告了 title 屬性，你在當中撰寫基本的 HTML 樣版，<body>與</body>出現了<jsp:doBody/>這個標籤，它可以取得自訂 Tag File 標籤使用時的本體內容。簡單地說，你可以這麼使用這個 Tag File：

TagFileDemo add3.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib  prefix="html" tagdir="/WEB-INF/tags" %>
<html:Html title="新增書籤">
    <html:Errors/>
    <form method="post" action="add.do">
        網址 http:// <input name="url" value="${param.url}"><br>
        網頁名稱：<input name="title" value="${param.title}"><br>
        分　類：<input type="text" name="category"
                       value="${param.category}"><br>
        <input value="送出" type="submit"><br>
    </form>
</html:Html>
```

你使用<html:Html>的 title 屬性設定網頁標題，而在<html:Html>與</html:Html>的本體中，可以撰寫你想要的 HTML、EL 或自訂標籤。本體的內容會在 Html.tag 的<jsp:doBody/>位置與其它內容結合在一起。

注意！前一段敘述說的，Tag File 的本體內容可以撰寫 HTML、EL 或自訂標籤，但沒有提到 Scriptlet。Tag File 的標籤在使用時若有本體，預設是不允許有 Scriptlet 的，因為定義 Tag File 時，tag 指示元素的 **body-content** 屬性預設就是 **scriptless**，也就是不可以出現<% %>、<%= %>或<%! %>元素。

```
<%@tag body-content="scriptless" pageEncoding="UTF-8"%>
```

body-content 屬性還可以設定 **empty** 或 **tagdependent**。empty 表示一定沒有本體內容，也就是只能用<html:Header/>這樣的方式來使用標籤（非 emtpty 的設定時，你可以用<html:Headers/>，或者是<html:Header>本體</html:Header>的方式）。tagdependent 表示將本體中的內容當作純文字處理，也就是如果本體中有出現 Scriptlet、EL 或自訂標籤，也只是當作純文字輸出，不會作任何的運算或轉譯。

提示 所以結論就是，Tag File 若有本體，在其中撰寫 Scriptlet 是沒有意義的，要不就不允許出現，要不就當作純文字輸出。

9.2.3 TLD 檔案

如果你將 Tag File 的*.tag 檔案放在/WEB-INF/tags 資料夾或子資料夾，並在 JSP 中使用 taglib 指示元素的 tagdir 屬性指定*.tag 的位置，你就可以使用 Tag File 了。其他人如果覺得你的 Tag File 不錯用，也只要將*.tag 複製到他們的/WEB-INF/tags 資料夾或子資料夾就可以了。

看本書的人畢竟都是 Java 程式設計人員，也許你就是偏好使用 JAR 檔把東西包一包再給別人使用，或是為了跟 Simple Tag 等自訂標籤庫一起包起來。如果你要將 Tag File 包成 JAR 檔案，那麼有幾個地方要注意一下：

- *.tag 檔案必須放在 JAR 檔的 **META-INF/tags** 資料夾或子資料夾下。
- 你要定義 **TLD（Tag Library Description）檔案**。
- TLD 檔案必須放在 JAR 檔的 **META-INF/TLDS** 資料夾下。

例如若你想將先前開發的 Errors.tag、Header.tag、Html.tag 封裝在 JAR 檔案中，則將這三個.tag 檔案放到某個資料夾的 META-INF/tags 下，不在 META-INF/TLDS 下定義 html.tld 檔案：

TagFileDemo html.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
   <tlib-version>1.0</tlib-version>
   <short-name>html</short-name>
```

```
    <uri>http://openhome.cc/html</uri>
    <tag-file>
        <name>Header</name>
        <path>/META-INF/tags/Header.tag</path>
    </tag-file>
    <tag-file>
        <name>Html</name>
        <path>/META-INF/tags/Html.tag</path>
    </tag-file>
    <tag-file>
        <name>Errors</name>
        <path>/META-INF/tags/Errors.tag</path>
    </tag-file>
</taglib>
```

其中**<uri>**設定是在 JSP 中與 taglib 指示元素的 url 屬性對應用的。每個**<tag-file>**中使用**<name>**定義了自訂標籤的名稱，使用**<path>**定義了*.tag 在 JAR 檔案中的位置。接下來你可以使用文字模式進入放置 META-INF 的資料夾中，執行以下指令產生 html.jar：

```
jar cvf ../html.jar *
```

若要這個產生的 html.jar，就將之放到 Web 應用程式的 WEB-INF/lib 資料夾中，而要使用標籤的 JSP 頁面，則可以如下撰寫：

TagFileDemo html.tld

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="html" uri="http://openhome.cc/html" %>
<html:Html title="新增書籤">
    <html:Errors/>
    <form method="post" action="add.do">
        網址 http:// <input name="url" value="${param.url}"><br>
        網頁名稱：<input name="title" value="${param.title}"><br>
        分　類：<input type="text" name="category"
                    value="${param.category}"><br>
        <input value="送出" type="submit"><br>
    </form>
</html:Html>
```

注意！這次是使用 taglib 指示元素的 **uri** 屬性，名稱對應至 TLD 檔案中的<uri>所設定的名稱。

9.3 Simple Tag 自訂標籤

有的時候，只使用 JSTL 是不夠的，有的時候你就是需要撰寫 Java 程式碼來操作某些 Java 物件，再把結果顯示在網頁上。上一節你學到了 Tag File，基本上它是設計給不會 Java 的網頁設計人員使用的，有些人會在 Tag File 中撰寫 Scriptlet 來操作 Java 物件，但並不建議這麼作，這麼作的結果只會走回 HTML 夾雜 Scriptlet 的回頭路（即使你將這個混亂約束在 Tag File 之中）。

如果你在自訂標籤時需要操作 Java 物件，可以考慮實作 Simple Tag 來自訂標籤，將 Java 程式碼撰寫在其中，絕大部份的情況下，實作 Simple Tag 可以滿足你的需求。

9.3.1 簡介 Simple Tag

相較於 Tag File 的使用，實作 Simple Tag 時有更多的東西必須了解。在這之前，先來使用 Simple Tag 模仿 JSTL 的<c:if>標籤，了解一個簡單的 Simple Tag 要如何開發，由於這是個「偽」JSTL 標籤，故且叫它作<f:if>標籤好了。

首先要撰寫標籤處理器，這是一個 Java 類別，你可以繼承 **javax.servlet.jsp.tagext.SimpleTagSupport** 來實作標籤處理器（Tag Handler），並重新定義 **doTag()** 方法來進行標籤處理。

SimpleTagDemo IfTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class IfTag extends SimpleTagSupport {
    private boolean test;

    @Override
    public void doTag() throws JspException {
        try {
            if(test) {
                getJspBody().invoke(null);
            }
        } catch (java.io.IOException ex) {
            throw new JspException("IfTag 執行錯誤", ex);
        }
```

```
    }

    public void setTest(boolean test) {
        this.test = test;
    }
}
```

在這邊的<f:if>標籤有個 test 屬性，所以你的標籤處理器必須有個接受 test 屬性的設值方法（Setter）。如果 test 屬性為 true，則呼叫 SimpleTagSupport 的 **getJspBody()**方法，這會傳回一個 **JspFragment** 物件，代表<f:if>與</f:if>間的本體內容，如果你呼叫 JspFragment 的 **invoke()**並傳入一個 null，表示執行<f:if>與</f:if>間的本體內容，如果沒有呼叫 invoke()，則<f:if>與</f:if>間的本體內容不會執行，也就不會有結果輸出至使用者的瀏覽器。

為了讓 Web 容器了解<f:if>標籤與 IfTag 標籤處理器之間的關係，你要定義一個**標籤程式庫描述檔（Tag Library Descriptor）**，也就是一個副檔名為 ***.tld** 的檔案。

SimpleTagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>f</short-name>
    <uri>http://openhome.cc/jstl/fake</uri>
    <tag>
        <name>if</name>
        <tag-class>cc.openhome.IfTag</tag-class>
        <body-content>scriptless</body-content>
        <attribute>
            <name>test</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>boolean</type>
        </attribute>
    </tag>
</taglib>
```

其中**<uri>**設定是在 JSP 中與 taglib 指示元素的 url 屬性對應用的。每

個**<tag>**標籤中使用**<name>**定義了自訂標籤的名稱，使用**<tag-class>**定義標籤處理器類別，而**<body-content>**設定為 **scriptless**，表示標籤本體中不允許使用 Scriptlet 等元素。

如果標籤上有屬性，則是使用**<attribute>**來設定，**<name>**設定屬性名稱，**<required>**表示是否一定要設定這個屬性，**<rtexprvalue>**表示屬性是否接受執行時期運算的結果（例如 EL 運算式的結果），如果設定為 `false` 或不設定`<rtexprvalue>`，表示在 JSP 上設定屬性時僅接受字串形式，**<type>**則設定屬性型態。

你可以將 TLD 檔案放在 WEB-INF 資料夾下，如此容器就會自動載入它。如果要使用這個標籤，同樣必須在 JSP 頁面上使用 `taglib` 指示元素。例如：

`SimpleTagDemo` ifTag.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="f" uri="http://openhome.cc/jstl/fake" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>自訂 if 標籤</title>
    </head>
    <body>
        <f:if test="${param.password == '123456'}">
            你的秘密資料在此！
        </f:if>
    </body>
</html>
```

在這個示範的 JSP 頁面中，使用自訂的`<f:if>`標籤，檢查 password 請求參數是否為所設定的數值，如果正確地話才會顯示`<f:if>`本體的內容。

提示　JSTL 本身並非用 Simple Tag 來實作的，在這一節中，只是用 Simple Tag 來模仿 JSTL 的功能。

9.3.2 了解架構與生命周期

看起來 Simple Tag 的開發似乎不會太難，主要就是繼承 `SimpleTagSupport` 類別、重新定義 `doTag()` 方法、定義 TLD 檔案以及使用 `taglib` 指示元素。不過實際上還有很多東西需要解釋。

SimpleTagSupport 實際上實作了 **javax.servlet.jsp.tagext.SimpleTag** 介面，而 SimpleTag 介面繼承了 **javax.servlet.jsp.tagext.JspTag** 介面。

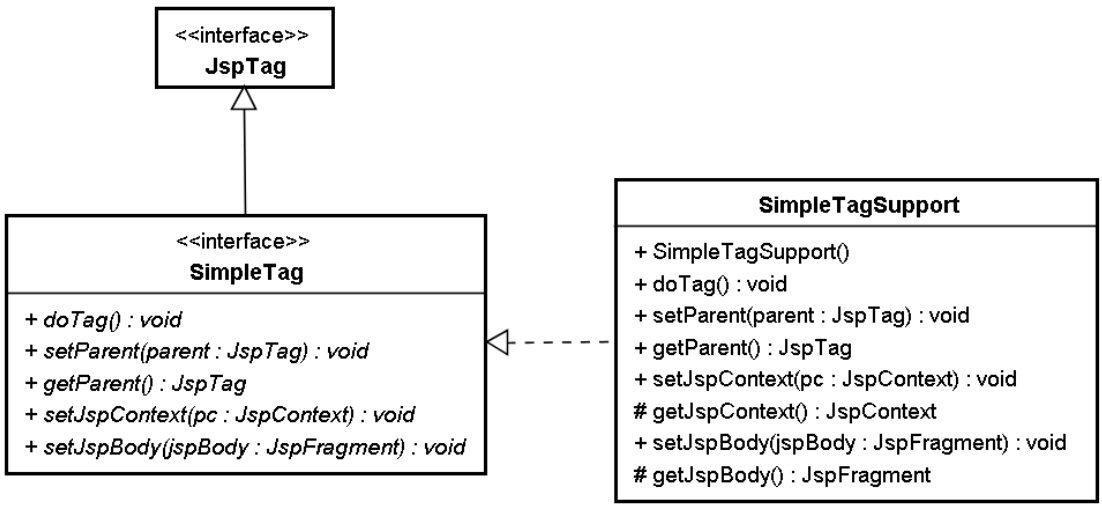


圖 9.4 Simple Tag 架構圖

所有的 JSP 自訂 Tag 都實作了 JspTag 介面， JspTag 介面只是個標示介面，本身沒有定義任何的方法。SimpleTag 介面繼承了 JspTag，定義了 Simple Tag 開發時所需的基本行為，你的 Simple Tag 標籤處理器必須實作 SimpleTag 介面，不過通常繼承 SimpleTagSupport 類別，因為該類別實作了 SimpleTag，並對所有方法作了基本實作，你只需要在繼承 SimpleTagSupport 之後，重新定義你所興趣的方法即可，通常就是重新定義 doTag() 方法。

當 JSP 網頁中包括 Simple Tag 自訂標籤，若使用者請求該網頁，在遇到自訂標籤時，會按照以下的網頁來進行處理：

1. 建立自訂標籤處理器實例。
2. 呼叫標籤處理器的 **setJspContext()** 方法設定 PageContext 實例。
3. 如果是巢狀標籤中的內層標籤，則還會呼叫標籤處理器的 **setParent()** 方法，並傳入外層標籤處理器的實例。
4. 設定標籤處理器屬性(例如這邊是呼叫 IfTag 的 setTest() 方法來設定)。
5. 呼叫標籤處理器的 **setJspBody()** 方法設定 JspFragment 實例。
6. 呼叫標籤處理器的 doTag() 方法。
7. 銷毀標籤處理器實例。

所以每一次的請求都會建立新的標籤處理器實例，而在執行 doTag() 過後就銷毀實例。由於標籤處理器中被設定了 PageContext，所以你可以用它來取得 JSP 頁面的所有物件，進行所有你在 JSP 頁面 Scriptlet 中可以執行的動作，這就是你封裝 Scriptlet 的方式。

JspFragment 就如其名稱所示，是個 JSP 頁面中的片段內容。你的自訂標籤若包括本體，將會轉譯為一個 JspFragment 實作類別，而本體內容將會在

invoke()方法進行處理。以 Tomcat 為例，<f:if>本體內容將轉譯為以下的 JspFragment 實作類別（一個內部類別）：

```
    private class Helper
        extends org.apache.jasper.runtime.JspFragmentHelper {
        // 略...
        public boolean invoke0( JspWriter out )
          throws Throwable {
          out.write("\n");
          out.write("          你的秘密資料在此！\n");
          out.write("       ");
          return false;
        }
        public void invoke( java.io.Writer writer )
            throws JspException {
            JspWriter out = null;
            if( writer != null ) {
                out = this.jspContext.pushBody(writer);
            } else {
                out = this.jspContext.getOut();
            }
            try {
                // 略...
                  invoke0( out );
                // 略...
            }
            catch( Throwable e ) {
                if (e instanceof SkipPageException)
                    throw (SkipPageException) e;
                throw new JspException( e );
            }
            finally {
                if( writer != null ) {
                    this.jspContext.popBody();
                }
            }
        }
    }
```

所以當你在 doTag()方法中使用 getJspBody()取得 JspFragment 實

例，在呼叫其 invoke()方法時傳入 null，這表示你將使用 PageContext 取得預設的 JspWriter 物件來作輸出回應（而並非不作回應）。接著進行本體內容的輸出，如果本體內容中包括 EL 或內層標籤，則會先作處理（在<body-content>設定為 scriptless 的情況下），在上面的簡單範例中，只是將<f:if>本體的 JSP 片段直接輸出（也就是 invoke1()的執行內容）。

如果呼叫 JspFragment 的 invoke()時傳入了一個 Writer 實例，則表示你要將本體內容的執行結果，以你所設定的 Writer 實例作輸出，這個稍後會再進行討論。

如果執行 doTag()的過程在某些條件下，必須中斷接下來頁面的處理或輸出，則你可以丟出 **javax.servlet.jsp.SkipPageException**，這個例外物件會在 JSP 轉譯後的_jspService()中如下處理：

```
...
try {
    // 丟出 SkipPageException 例外的地方
    // 其它 JSP 頁面片段
    // 略...
} catch (Throwable t) {
    if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            try { out.clearBuffer(); } catch (java.io.IOException e) {}
            if (_jspx_page_context != null)
                _jspx_page_context.handlePageException(t);
        }
    }
}
...
```

簡單地說，在 catch 中捕捉到例外時，若是 SkipPageException 實例，什麼事都不作！也就是你在 doTag()中若只是想中斷接下來的頁面處理，但又不想特別處理例外，則可以丟出 SkipPageException。

提示 若是丟出其它類型的例外，則在 PageContext 的 handlePageException()中會看看有無設置錯誤處理相關機制，並嘗試進行頁面轉發或包含的動作，否則的話就包裝為 ServletException 並丟給容器作預設處理，這時你就會看到 HTTP Status 500 的網頁出現了。

9.3.3 處理標籤屬性與本體

如果本體的內容需要執行多次該如何處理？例如原本 JSTL 的<c:forEach>標籤之功能，必須依所設定的陣列或 Collection 物件長度，以決定本體中的內容顯示次數。以下就來使用 Simple Tag 實作<f:forEach>標籤以模仿<c:forEach>的功能。這個<f:forEach>標籤會是這麼使用：

```
<f:forEach var="name" items="${names}">
    ${name}<br>
</f:forEach>
```

為了簡化範例，先不考慮 itesms 屬性上 EL 的運算結果是陣列的情況，而只考慮 Collection 物件。你可以設定 var 屬性來決定每次從 Collection 取得物件時，可使用哪個名稱於標籤本體中取得該物件，var 只接受字串方式來設定名稱。來看看如何實作標籤處理器。

SimpleTagDemo ForEachTag.java

```
package cc.openhome;

import java.util.Collection;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class ForEachTag extends SimpleTagSupport {
    private String var;
    private Collection items;

    @Override
    public void doTag() throws JspException {
        try {
            for(Object o : items) {
                this.getJspContext().setAttribute(var, o);
                this.getJspBody().invoke(null);
            }
        } catch (java.io.IOException ex) {
            throw new JspException("ForEachTag 執行錯誤", ex);
        }
    }

    public void setVar(String var) {
```

設定標籤本體可用的 EL 名稱

在迴圈中呼叫 invoke() 方法

```
        this.var = var;
    }

    public void setItems(Collection items) {
        this.items = items;
    }
}
```

在屬性的設定上，由於 var 屬性會是字串方式設定，所以宣告為 String 型態。items 運算的結果可接受 Collection 物件，所以型態宣告為 Collection。標籤本體可接受的 EL 名稱，事實上是取得 PageContext 後使用其 setAttribute()進行設定。<f:forEach>標籤本體內容必須執行多次，則是透過多次呼叫 invoke()來達成，簡單地說，你在 doTag()中每呼叫一次 invoke()，則會執行一次本體內容。

如果在 doTag()中透過取得的 PageContext 設定 page 範圍屬性，則 doTag()執行完畢後會自動清除屬性，你不用特地使用 removeAttribute()進行移除。所以這個範例在離開<f:forEach>標籤範圍後，就無法再透過 var 屬性所設定的名稱取得值：

```
<f:forEach var="name" items="${names}">
    ${name}<br>
</f:forEach>
${name} <!-- 運算結果是 null -->
```

接著同樣地，你要在 TLD 檔案中定義自訂標籤相關資訊：

SimpleTagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
   <tlib-version>1.0</tlib-version>
   <short-name>f</short-name>
   <uri>http://openhome.cc/jstl/fake</uri>
   // 略...
   <tag>
       <name>forEach</name>
       <tag-class>cc.openhome.ForEachTag</tag-class>
       <body-content>scriptless</body-content>
       <attribute>
           <name>var</name>
```

```
            <required>true</required>
            <type>java.lang.String</type>
        </attribute>
        <attribute>
            <name>items</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>java.util.Collection</type>
        </attribute>
    </tag>
</taglib>
```

Simple Tag 的本體內容，也就是<body-content>屬性與 Tag File 類似，除了 scriptless 之外，還可以設定 **empty** 或 **tagdependent**。empty 表示一定沒有本體內容。tagdependent 表示將本體中的內容當作純文字處理，也就是如果本體中有出現 Scriptlet、EL 或自訂標籤，也只是當作純文字輸出，不會作任何的運算或轉譯。由於 var 屬性只接受字串設定，所以不需要設定<rtexprvalue>標籤，在不設定時預設就是 false，也就是不接受執行時期的運算值作為屬性設定值。

到目前為止，你都是透過 SimpleTagSupport 的 getJspBody()取得 JspFragment，並在呼叫 invoke()時傳入 null，先前解釋過，這表示你將使用 PageContext 取得預設的 JspWriter 物件來作輸出回應，也就是預設會輸出回應至使用者的瀏覽器。

如果你在呼叫時傳入一個自訂的 Writer 物件，則標籤本體內容的處理結果，就會使用你所指定的 Writer 物件進行輸出，在你需要將處理過後的本體內容再作進一步處理時，就會採取這樣的作法。例如，你可以開發一個將本體執行結果全部轉大寫的簡單標籤：

SimpleTagDemo ToUpperCaseTag.java

```
package cc.openhome;

import java.io.StringWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class ToUpperCaseTag extends SimpleTagSupport {
    @Override
    public void doTag() throws JspException {
        try {
            StringWriter writer = new StringWriter();
```

```
            this.getJspBody().invoke(writer);
            String upper = writer.toString().toUpperCase();
            this.getJspContext().getOut().print(upper);
        } catch (java.io.IOException ex) {
            throw new JspException("ToUpperCaseTag 執行錯誤", ex);
        }
    }
}
```

本體執行結果將輸出至 StringWriter 物件

在這個標籤處理器中執行 invoke()後，標籤本體執行的結果將輸出至 StringWriter 物件，你再呼叫 StringWriter 物件的 toString()取得輸出的字串結果，並呼叫 toUpperCase()方法將結果轉為大寫。如果這個轉換大寫後的字串結果是要輸出至使用者瀏覽器的話，則再透過 PageContext 的 getOut()取得 JspWriter 物件，再呼叫 print()方法輸出結果。

記得在 TLD 檔案中加入這個自訂標籤的定義：

SimpleTagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>f</short-name>
    <uri>http://openhome.cc/jstl/fake</uri>
    // 略...
    <tag>
        <name>toUpperCase</name>
        <tag-class>cc.openhome.ToUpperCaseTag</tag-class>
        <body-content>scriptless</body-content>
    </tag>
</taglib>
```

你可以如下使用這個標籤，結果是 items 的字串都會被轉為大寫：

```
<f:toUpperCase>
    <f:forEach var="name" items="${names}">
        ${name} <br>
    </f:forEach>
</f:toUpperCase>
```

還記得 9.3.2 節那段轉譯後的內部 Helper 類別嗎？如果你呼叫 invoke()方法時設定了一個 Writer 物件，則會呼叫 pageContext 的 **pushBody()**方

法並傳入該物件，這會將 pageContext 的 getOut()方法所取得的物件設定為你的 Writer 物件，並在堆疊中加以記錄原有的 JspWriter 物件。

```
JspWriter out = null;
if( writer != null ) {
    out = this.jspContext.pushBody(writer);
} else {
    out = this.jspContext.getOut();
}
```

所以若標籤本體內容中還有內層標籤，透過 getOut()所取得的就是你所設定的 Writer 物件（除非內層標籤在呼叫 invoke()時，也設定了它自己的 Writer 物件）。pushBody()傳回的是 BodyContent 物件，為 JspWriter 的子類別，包裹了你傳入的 Writer 物件，BodyContent 顧名思義，在輸出結束後，將會包括所有標籤本體的執行結果（包括內層標籤），因為執行結果都會輸出至你所設定的 Writer 物件。

在 invoke()結束前會呼叫 pageContext 的 **popBody()**方法，從堆疊中恢復原本 getOut()所應傳回的 JspWriter 物件。

提示 這邊針對 pushBody()、popBody()方法的說明屬於比較進階的觀念，可搭配 PageContext 關於 pushBody()、popBody()的原始碼來了解。如果腦袋暫時有點打結，則只要記得結論：「如果你呼叫 invoke()時傳入了 Writer 物件，則標籤本體執行結果將輸出至你的 Writer 物件。」

9.3.4 與父標籤溝通

如果你要設計的自訂標籤是放置在某個標籤之中，而且必須與外層標籤作溝通，例如 JSTL 中的<c:when>、<c:otherwise>必須放在<c:choose>中，且<c:when>或<c:otherwise>必須得知，先前的<c:when>是否已經測試通過並執行了本體內容，如果是的話就不再執行後續的測試。

在 9.3.2 節中談過，當 JSP 中包括自訂標籤時，會建立自訂標籤處理器的實例，呼叫 setJspContext()設定 PageContext 實例，再來若是巢狀標籤中的內層標籤，則還會呼叫標籤處理器的 **setParent()**方法，並傳入外層標籤處理器的實例，這就是你與外層標籤接觸的機會。

接下來將以模仿 JSTL 的<c:choose>、<c:when>、<c:otherwise>標籤為例，製作自訂的<f:choose>、<f:when>、<f:otherwise>標籤，了解內層標籤如何與外層標籤溝通。首先來看看<f:choose>的標籤處理器如何撰寫：

SimpleTagDemo ChooseTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class ChooseTag extends SimpleTagSupport {
    private boolean matched;

    @Override
    public void doTag() throws JspException {
        try {
            this.getJspBody().invoke(null);
        } catch (java.io.IOException ex) {
            throw new JspException("ChooseTag 執行錯誤", ex);
        }
    }

    public boolean isMatched() {
        return matched;
    }

    public void setMatched(boolean matched) {
        this.matched = matched;
    }
}
```

ChooseTag 基本上沒什麼事，只是內含一個 boolean 型態的成員 matched，預設是 false。一旦內部的<f:when>有測試成功的情況，會將 matched 設定為 true。ChooseTag 的 doTag()只需要作一件事，取得 JspFragment 並呼叫 invoke(null)執行標籤本體內容。

再來看看<f:when>的標籤處理器實作：

SimpleTagDemo WhenTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.JspTag;
import javax.servlet.jsp.tagext.SimpleTagSupport;
```

```
public class WhenTag extends SimpleTagSupport {
    private boolean test;

    @Override
    public void doTag() throws JspException {
        try {
            JspTag parent = null;
            if (!((parent = getParent()) instanceof ChooseTag)) {
                throw new JspTagException("必須置於 choose 標籤中");
            }

            if(((ChooseTag) parent).isMatched()) {
               return;
            }

            if(test) {
                ((ChooseTag) parent).setMatched(true);
                this.getJspBody().invoke(null);
            }
        } catch (java.io.IOException ex) {
            throw new JspException("WhenTag 執行錯誤", ex);
        }
    }

    public void setTest(boolean test) {
        this.test = test;
    }
}
```

❶無法取得 parent 或不為 ChooseTag 類型，表示不在 choose 標籤中

❷parent 的 matched 為 true，表示先前有 when 通過測試

❸通過測試，設定 parent 的 matched 為 true

❹執行標籤本體

<f:when>可以設定 test 屬性來看看是否執行本體內容。在測試開始前，必須先嘗試取得 parent，如果無法取得（為 null 的情況），表示不在任何標籤之中，或是 parent 不為 ChooseTag 型態，表示不是置於<f:choose>之中，這是個錯誤的使用方式，所以必須丟出例外❶。

如果確實是置於<f:choose>標籤之中，接著嘗試取得 parent 的 matched 狀態，如果已經被設定為 true，表示先前有<f:when>已經通過測試並執行了其本體內容，那麼目前這個<f:when>就不用再作測試了❷。如果是置於<f:choose>之中，而且先前沒有<f:when>通過測試，接著就可以進行目前這個<f:when>的測試，如果測試成功，則設定 parent 的 matched 為 true，

並執行標本體。

接著來看<f:otherwise>的標籤處理器如何撰寫：

SimpleTagDemo OtherwiseTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.JspTag;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class OtherwiseTag extends SimpleTagSupport {
    @Override
    public void doTag() throws JspException {
        try {           JspTag parent = null;
            if (!((parent = getParent()) instanceof ChooseTag)) {
                throw new JspTagException("WHEN_OUTSIDE_CHOOSE");
            }

            if(((ChooseTag) parent).isMatched()) {
               return;
            }
                                    直接執行標籤本體內容
                                          ↓
             this.getJspBody().invoke(null);
        } catch (java.io.IOException ex) {
            throw new JspException("Error in OtherwiseTag tag", ex);
        }
    }
}
```

<f:otherwise>標籤的處理基本上與<c:when>類似，必須確認是否置於<f:choose>標籤之中；必須確認先前是否有<c:when>測試成功，如果先前沒有<c:when>測試成功的話，就直接執行標籤本體內容

提示 WhenTag 與 OtherwiseTag 的 doTag()執行流程類似，可以為它們製作一個父類別，以避免重複程式碼的問題。基本上，這也是 JSTL 的作法。

接著記得定義 TLD 檔，在當中加入自訂標籤定義：

SimpleTagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
   <tlib-version>1.0</tlib-version>
   <short-name>f</short-name>
   <uri>http://openhome.cc/jstl/fake</uri>
   // 略...
   <tag>
      <name>choose</name>
      <tag-class>cc.openhome.ChooseTag</tag-class>
      <body-content>scriptless</body-content>
   </tag>
   <tag>
      <name>when</name>
      <tag-class>cc.openhome.WhenTag</tag-class>
      <body-content>scriptless</body-content>
      <attribute>
         <name>test</name>
         <required>true</required>
         <rtexprvalue>true</rtexprvalue>
         <type>boolean</type>
      </attribute>
   </tag>
   <tag>
      <name>otherwise</name>
      <tag-class>cc.openhome.OtherwiseTag</tag-class>
      <body-content>scriptless</body-content>
   </tag>
</taglib>
```

接下來使用自訂的<f:choose>、<f:when>、<f:otherwise>標籤改寫 9.1.2 節中 login2.jsp 作為示範：

SimpleTagDemo login.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="f" uri="http://openhome.cc/jstl/fake"%>
<jsp:useBean id="user" class="cc.openhome.User"  />
```

```
<jsp:setProperty name="user" property="*" />
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>登入頁面</title>
    </head>
    <body>
        <f:choose>
            <f:when test="${user.valid}">
                <h1>${user.name}登入成功</h1>
            </f:when>
            <f:otherwise>
                 <h1>登入失敗</h1>
            </f:otherwise>
        </f:choose>
    </body>
</html>
```

執行的方式與結果與 9.1.2 節是相同的，只不過這次用的是自訂的「偽」JSTL 標籤。

你可以使用 getParent()取得 parent 標籤，也就是目前標籤的上一層標籤。如果在一個數個巢狀的標籤中，想要直接取得某個指定類型的外層標籤，則可以透過 SimpleTagSupport 的 findAncestorWithClass()靜態方法。例如：

```
SomeTag ancestor = (SomeTag) findAncestorWithClass(
                              this, SomeTag.class);
```

findAncestorWithClass()方法會在目前標籤的外層標籤中尋找，直到找到指定的類型之外層標籤物件後傳回。

9.3.5 TLD 檔案

你可以將 TLD 檔案直接置放在 Web 應用程式的 WEB-INF 資料夾或其子資料夾中，容器會在 WEB-INF 資料夾或子資料夾中找到 TLD 檔案並載入。如果你要用 JAR 檔案來封裝自訂標籤處理器與 TLD 檔案，則與 9.2.3 節說明的類似，不過這次 TLD 檔案不一定要放在 JAR 檔案的 META-INF/tlds 資料夾中，而只要是在 JAR 檔案的 META-INF 資料夾或子資料夾即可。也就是：

- JAR 檔案根目錄下放置編譯好的類別（包含對應套件的資料夾）
- JAR 檔案 META-INF 資料夾或子資料夾中放置 TLD 檔案

例如，你可以將這一節所開發的 Simple Tag 如下放置在一個 fake 資料夾中：

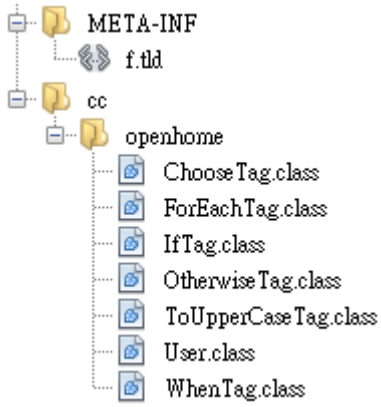


圖 9.5 準備製作 JAR 檔的資料夾

接著在文字模式中進入 fake 資料夾，執行以下的指令：

```
jar cvf ../fake.jar *
```

如此在 fake 資料夾上一層目錄中，就會產生 fake.jar 檔案，若想使用這個 fake.jar，只要將之置入 WEB-INF/lib 中，就可以開始使用你自訂的標籤庫了。

9.4 Tag 自訂標籤

使用 Simple Tag 實作自訂標籤算是簡單，所有要實作的內容都是在 `doTag()` 方法中進行。絕大多數的情況下，使用 Simple Tag 應能滿足你自訂標籤的需求。然而，Simple Tag 是從 JSP 2.0 之後才加入至標準之中，在 JSP 2.0 之前實作自訂標籤，則是透過 `Tag` 介面下相關類別的實作來完成。

你應該很少有機會使用 `Tag` 介面下的相關類別來自訂標籤（也不太可能會想這麼作），不過你可能會遇到 JSP 2.0 之前實作出來的自訂標籤（例如 JSTL），甚至必須對它進行修改，所以還是有需要了解如何使用 `Tag` 介面下的相關類別。

這一節將使用 `Tag` 介面下的相關類別，實作出 9.3 節使用 Simple Tag 所自訂的標籤，讓你了解兩者在自訂標籤上的不同實作方式。

9.4.1 簡介 Tag

9.3.1 節曾經使用 Simple Tag 開發了一個`<f:if>`自訂標籤，在這邊則改用 Tag 介面下的相關類別來實作`<f:if>`標籤。首先同樣地，你要定義標籤處理器，這可以透過繼承 **`javax.servlet.jsp.tagext.TagSupport`** 來實作。例如：

TagDemo IfTag.java

```
package cc.openhome;
```

```
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;

public class IfTag extends TagSupport {
    private boolean test;

    @Override
    public int doStartTag() throws JspException {
        if(test) {
            return EVAL_BODY_INCLUDE; ◄— 測試通過會執行標籤本體內容
        }
        return SKIP_BODY;  ◄— 執行到這邊表示測試失敗，所以忽略本體內容
    }

    public void setTest(boolean test) {
        this.test = test;
    }
}
```

當 JSP 中開始處理標籤時，會呼叫 **doStartTag()** 方法，後續是否執行本體則是用 doStartTag() 的傳回值決定。如果 doStartTag() 方法傳回 **EVAL_BODY_INCLUDE** 常數（定義在 Tag 介面），則會執行本體內容，傳回 **SKIP_BODY** 常數（定義在 Tag 介面），則不執行本體內容。

提示　看起來只是從 doTag() 實作改為 doStartTag() 嗎？因為這邊還是簡介，事實上繼承 TagSupport 類別後，針對標籤處理的不同時機，可以重新定義的方法有 doStartTag()、doAfterBody() 與 doEndTag()。在開始討論這些方法時，你的腦袋可要保持清楚。

接著定義 TLD 檔案的內容：

TagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>f</short-name>
```

```
    <uri>http://openhome.cc/jstl/fake</uri>
    <tag>
        <name>if</name>
        <tag-class>cc.openhome.IfTag</tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>test</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>boolean</type>
        </attribute>
    </tag>
</taglib>
```

基本上，在定義 TLD 檔案時與使用 Simple Tag 時是相同的，除了在 <body-content> 的設定值上，在這邊可以設定的有 empty、**JSP** 與 tagdependent（在 Simple Tag 中可以設定的是 empty、scriptless 與 tagdependent）。其中 JSP 的設定值表示本體中若包括動態內容，如 Scriptlet 等元素、EL 或自訂標籤都會被執行。

再來你可以如 9.3.1 節的範例來使用這個標籤，基於簡介時範例的完整性，再貼上來一次：

TagDemo ifTag.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="f" uri="http://openhome.cc/jstl/fake" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>自訂 if 標籤</title>
    </head>
    <body>
        <f:if test="${param.password == '123456'}">
            你的秘密資料在此！
        </f:if>
    </body>
</html>
```

同樣地，如果你的請求中包括請求參數 password 且值為 123456，則會顯

示本體內容，否則你只會看到一片空白。

9.4.2 了解架構與生命周期

在開發<f:if>中雖然省略了許多的細節，但也略為看到與 Simple Tag 開發的不同。在 Simple Tag 開發中，你只要定義 doTag()方法就好了，但在實作 Tag 介面相關類別時，依不同的時機，你要定義不同的 doXxxTag()方法，並依需求傳回不同的值。

那些不同的 doXxxTag()方法，實際上是分別定義在 **Tag** 與 **IterationTag** 介面上的方法，繼承與實作架構如下所示：

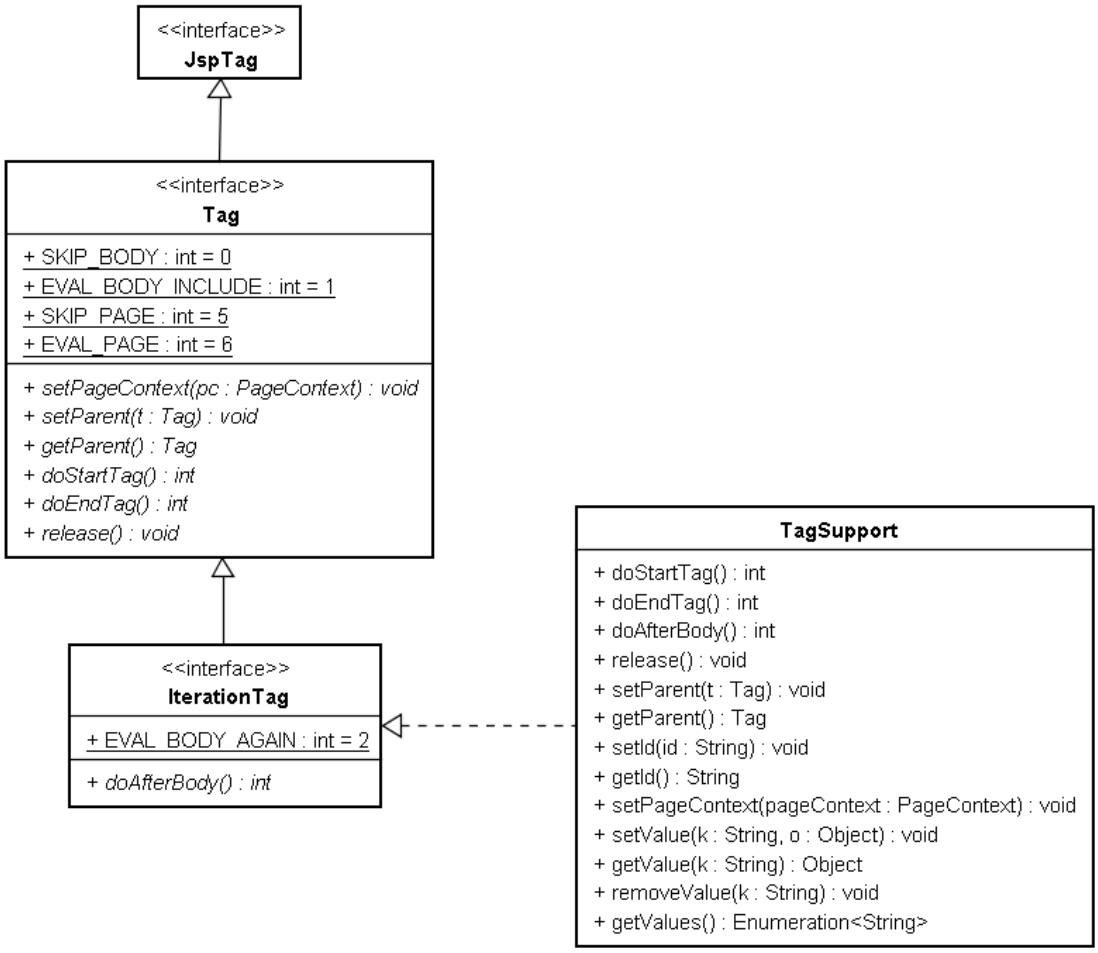


圖 9.6 Tag、IterationTag 與 TagSupport

類似 SimpleTag 介面，Tag 介面繼承自 JspTag 介面，它定義了基本的 Tag 行為，像是設定 PageContext 實例的 setPageContext()、設定外層父標籤物件的 **setParent()**方法、標籤物件銷毀前呼叫的 **release()**方法等。單是使用 Tag 介面的話，無法重複執行本體內容，這是用子介面 IterationTag 介面的 doAfterBody()所定義（稍後會看到如何重複執行本體內容）。TagSupport 類別實作了 IteratorTag 介面，對介面上所有方法作了基本實

作，你只需要在繼承 TagSupport 之後，針對必要的方法重新定義即可。

當 JSP 中遇到 TagSupport 自訂標籤時，會進行以下的動作：

1. 嘗試從標籤池（Tag Pool）找到可用的標籤物件，如果找到就直接使用，如果沒找到就建立新的標籤物件。
2. 呼叫標籤處理器的 setPageContext() 方法設定 PageContext 實例。
3. 如果是巢狀標籤中的內層標籤，則還會呼叫標籤處理器的 setParent() 方法，並傳入外層標籤處理器的實例。
4. 設定標籤處理器屬性（例如這邊是呼叫 IfTag 的 setTest() 方法來設定）
5. 呼叫標籤處理器的 doStartTag() 方法，並依不同的傳回值決定是否執行本體或呼叫 doAfterBody()、doEndTag() 方法（稍後詳述）。
6. 將標籤處理器實例置入標籤池中以便再度使用。

首先注意到第 1 點與第 6 點，沒錯！這邊的 Tag 實例是可以重複使用的（SimpleTag 實例則是每次請求都建立新物件，用完就銷毀回收），所以自訂 Tag 類別時，要注意你的物件狀態是否會保留下來，必要的時候，在 doStartTag() 方法中，可以進行狀態重置的動作。別以為可以使用 **release()** 方法來作狀態重置，因為 release() 方法只會在標籤實例真正被銷毀回收前被呼叫。

接著來詳細說明第 5 點。JSP 頁面會根據標籤處理器各方法呼叫的不同傳回值，來決定要呼叫哪一個方法或進行哪一個動作，這個直接使用流程圖來說明會比較清楚：

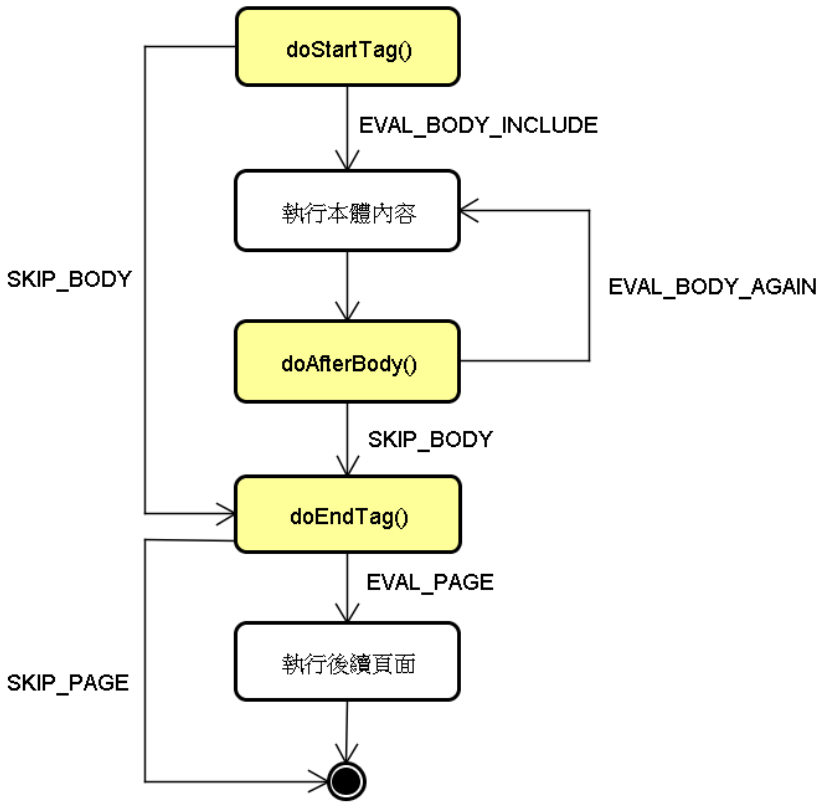


圖 9.7 標籤處理器流程圖

doStartTag()可以回傳 **EVAL_BODY_INCLUDE** 或 **SKIP_BODY**。如果傳回 EVAL_BODY_INCLUDE 則會執行本體內容，而後呼叫 doAfterBody()（就相當於 SimpleTag 的 doTag()中呼叫了 JspFragment 的 invoke()方法)。如果不想執行本體內容，則可傳回 SKIP_BODY（就相當於 SimpleTag 的 doTag()不呼叫JspFragment的invoke()方法)，此時就會呼叫doEndTag()方法。

這邊先不討論 doAfterBody()方法的傳回值，因為 doAfterBody()預設傳回值是 SKIP_BODY，如果你不重新定義 doAfterBody()方法，無論有無執行本體，流程最後都會來到 doEndTag()。在 **doEndTag()**中，你可傳回 **EVAL_PAGE** 或 **SKIP_PAGE**。如果傳回 EVAL_PAGE，則自訂標籤後續的 JSP 頁面才會繼續執行，如果傳回 SKIP_PAGE 就不會執行後續的 JSP 頁面（相當於 SimpleTag 的 doTag()中丟出 SkipPageException 的作用)。

實際上，由於 TagSupport 類別對 IterationTag 介面作了基本實作，doStartTag()、doAfterBody()與 doEndTag()都有預設的傳回值，依序分別是 SKIP_BODY、SKIP_BODY 及 EVAL_PAGE，也就是預設不處理本體，標籤結束後會執行後續的 JSP 頁面。

提示 實際上在 Tomcat 中，如果你觀看 JSP 轉譯後的 Servlet 原始碼，會發現只要 doStartTag()的傳回值不是 SKIP_BODY，就會執行本體內容並呼叫 doAfterBody()方法。doEndTag()只要傳回值不是 SKIP_PAGE，就會執行後續的 JSP 頁面。

9.4.3 重複執行標籤本體

如果你想實作 9.3.3 節的<f:forEach>標籤，可以根據所給定的 Collection 物件個數來決定重複執行標籤本體的次數。那麼你該在哪個方法中實作？doStartTag()？根據圖 9.7，doStartTag()只會執行一次！doEndTag()？這時本體內容處理已經結束了！

根據圖 9.7，在 **doAfterBody()**方法執行過後，如果你傳回 **EVAL_BODY_AGAIN**，則會再重複執行一次本體內容，而後再度呼叫 doAfterBody()方法，除非你在 doAfterBody()中傳回 SKIP_BODY 才會呼叫 doEndTag()。顯然地，doAfterBody()是你可以實作<f:forEach>標籤重複處理特性的地方。

不過這邊有點小陷阱！當 doStartTag()傳回 EVAL_BODY_INCLUDE 後，會先執行本體內容後再呼叫 doAfterBody()方法，也就是說，實際上本體已經執行過一遍了！所以正確的作法應該是，doStartTag()與 doAfterBody()都要實作，doStartTag()實作第一次的處理，doAfterBody()實作後續的重複處理。例如：

TagDemo ForEachTag.java

```
package cc.openhome;

import java.util.Collection;
import java.util.Iterator;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

public class ForEachTag extends TagSupport {
    private String var;
    private Iterator iterator;

    @Override
    public int doStartTag() throws JspException {
        if(iterator.hasNext()) {
            this.pageContext.setAttribute(var, iterator.next());
            return EVAL_BODY_INCLUDE;
        }
        return SKIP_BODY;
    }

    @Override
    public int doAfterBody() throws JspException {
        if(iterator.hasNext()) {
            this.pageContext.setAttribute(var, iterator.next());
            return EVAL_BODY_AGAIN;
        }
        return SKIP_BODY;
    }

    public void setVar(String var) {
        this.var = var;
    }

    public void setItems(Collection items) {
        this.iterator = items.iterator();
    }
}
```

❶測試並執行第一次的處理

❷進行本體執行後呼叫 doAfterBody()

❸測試並執行後續的處理

❹再執行一次本體後呼叫 doAfterBody()

在<f:forEach>的標籤處理器實作中，必須先為第一次的本體執行作屬性設定❶，如此傳回 EVAL_BODY_INCLUDE 後第一次執行本體內容時❷，才可以有 var 所設定的屬性名稱可以存取。接著呼叫 doAfterBody() 方法，其中再為第二次之後的本體處理作屬性設定❸，如果需要再執行一次本體，則傳回 **EVAL_BODY_AGAIN**❹，再度執行完本體後又會呼叫 doAfterBody() 方法，如果不想執行本體了，則傳回 **SKIP_PAGE**，則流程會來到 doEndTag() 的執行（在 SimpleTag 的 doTag() 中直接使用迴圈語法，顯然直覺多了）。

接著同樣在定義 TLD 檔案中定義標籤：

TagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>f</short-name>
    <uri>http://openhome.cc/jstl/fake</uri>
    // 略...
    <tag>
        <name>forEach</name>
        <tag-class>cc.openhome.ForEachTag</tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>var</name>
            <required>true</required>
            <type>java.lang.String</type>
        </attribute>
        <attribute>
            <name>items</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>java.util.Collection</type>
        </attribute>
    </tag>
</taglib>
```

與9.3.3節的TLD檔案中定義之差別，其實僅在<body-content>是用**JSP**而不是 scriptless。你可以使用 9.3.3 節的 JSP 檔案來測試這個<f:forEach>標籤，基於篇幅限制，這邊就不再列出。

提示　實際上在 Tomcat 中，如果你觀看 JSP 轉譯後的 Servlet 原始碼，會發現只要 doAfterBody() 的傳回值不是 EVAL_BODY_AGAIN，就會再度執行本體內容並呼叫 doAfterBody()方法。

9.4.4 處理本體執行結果

如果你想要在本體執行過後，取得執行的結果並作適當處理該如何進行？例如實作一個 9.3.3 節的<f:toUpperCase>標籤？只是繼承 TagSupport 的話沒辦法達到這個目的！你可以繼承 **javax.servlet.jsp.tagext.BodyTagSupport** 類別來實作，先來看看其類別架構：

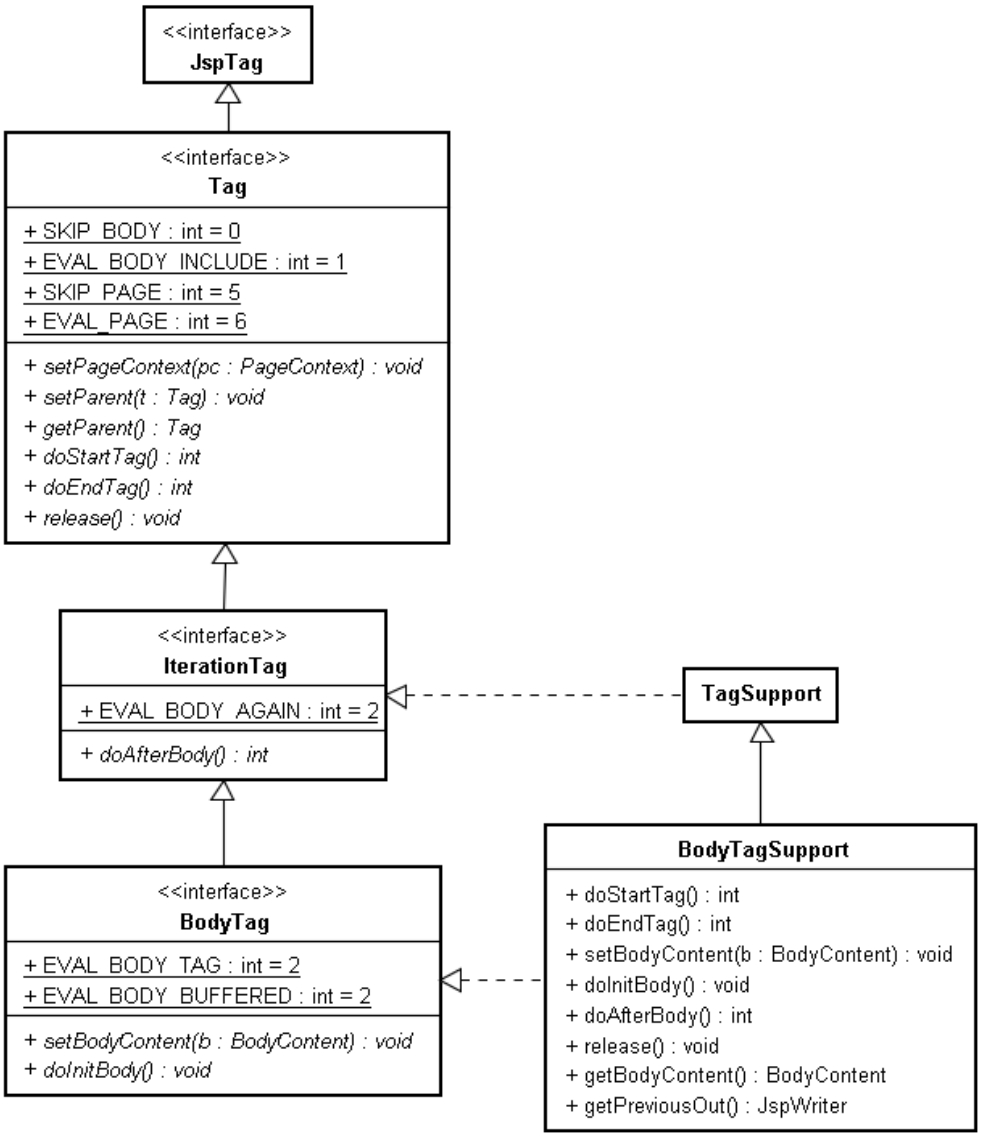


圖 9.8 加上 BodyTag 與 BodyTagSupport 後的架構圖

在上圖中，多了 BodyTag 介面，其繼承自 IterationTag 介面，新增了

setBodyContent()與**doInitBody()**兩個方法，而 BodyTagSupport 則繼承自 TagSupport 類別，將 **doStartTag()** 的預設傳回值改為 **EVAL_BODY_BUFFERED**，並針對 BodyTag 介面作了簡單的實作。

在繼承 BodyTagSupport 類別實作自訂標籤時，如果 doStartTag()傳回了 EVAL_BODY_BUFFERED，則會呼叫 setBodyContent()方法而後呼叫 doInitBody()方法，接著再執行標籤本體，也就是圖 9.7 的流程將變成以下：

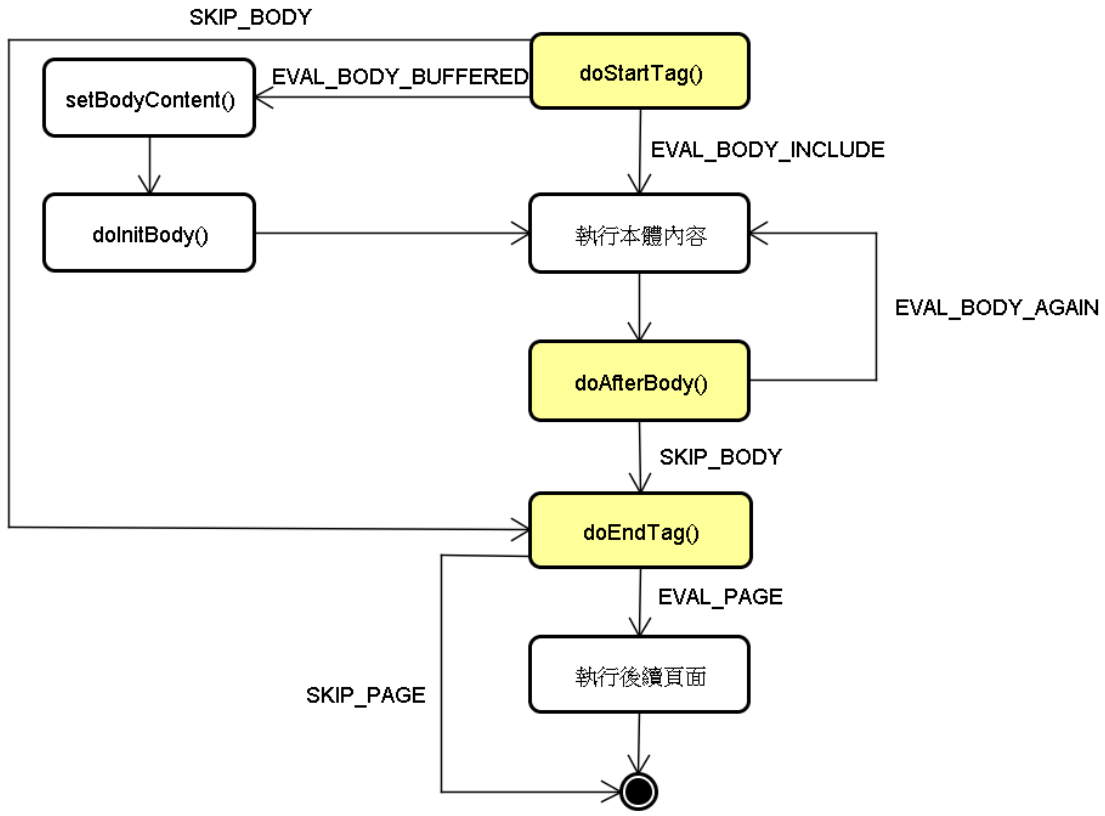


圖 9.9 加上 SKIP_BODY 後的流程圖

基本上，在使用 BodyTagSupport 實作自訂標籤時，並不需要去重新定義 setBodyContent()與 doInitBody()方法，只需要知道這兩個方法執行過後，在 doAfterBody()或 doEndTag()方法中，你就可以透過 getBodyContent()取得一個 BodyContent 物件（Writer 的子物件），這個物件中包括本體內容執行後的結果，例如透過 BodyContent 的 getString()方法，以字串的方式傳回執行後的本體內容。

如果你要將加工後的本體內容輸出使用者的瀏覽器，通常會在 doEndTag()中使用 pageContext 的 getOut()取得 JspWriter 物件，然後利用它來輸出內容至使用者的瀏覽器。如果你在 doAfterBody()中使用 pageContext 的 getOut()方法，所取得的物件與 getBodyContent()所取得的其實是相同的物件。如果你一定要在 doAfterBody()中取得 JspWriter 物件，則必須透過 BodyContent 的 **getEnclosingWriter()**方法。

提示 原因可以在 JSP 轉譯後的 Servlet 程式碼中找到。如果 doStartTag() 傳回 EVAL_BODY_BUFFERED，則會使用 PageContext 的 pushBody()將目前的 JspWriter 置入堆疊中，並傳回一個 BodyContent 物件，而後呼叫 setBodyContent()並傳入這個 BodyContent 物件，然後呼叫 doInitBody()方法，而在呼叫 doEndTag()方法前，如果先前 doStartTag() 傳回 EVAL_BODY_BUFFERED，則會呼叫 PageContext 的 popBody()，將原本的 JspWriter 從堆疊中取出。

以下使用 BodyTagSupport 類別來實作出 9.3.3 節的<f:toUpperCase>標籤處理器作為示範：

TagDemo ToUpperCaseTag.java

```
package cc.openhome;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class ToUpperCaseTag extends BodyTagSupport {
    @Override
    public int doEndTag() throws JspException {
        String upper = this.getBodyContent().getString().toUpperCase();
        try {
            pageContext.getOut().write(upper);
        } catch (IOException ex) {
            Logger.getLogger(ToUpperCaseTag.class.getName())
                    .log(Level.SEVERE, null, ex);
        }
        return EVAL_PAGE;
    }
}
```

在這邊於 doEndTag()中透過 getBodyContent()取得 BodyContent 物件，並呼叫其 getString()取得執行過後的標籤本體內容，再進行轉字母為大寫的動作。轉換後的本體內容，再透過 pageContext 的 getOut()取得 JspWriter 進行輸出。

記得在 TLD 檔案中定義標籤：

TagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>f</short-name>
    <uri>http://openhome.cc/jstl/fake</uri>
    // 略...
    <tag>
        <name>toUpperCase</name>
        <tag-class>cc.openhome.ToUpperCaseTag</tag-class>
        <body-content>JSP</body-content>
    </tag>
</taglib>
```

接著就如同 9.3.2 節的示範，你可以如下使用這個標籤：

```
<f:toUpperCase>
    <f:forEach var="name" items="${names}">
        ${name} <br>
    </f:forEach>
</f:toUpperCase>
```

9.4.5 與父標籤溝通

就如同 9.3.4 節介紹 Simple Tag 時的說明，如果有一些標籤是必須與外層標籤作溝通，則可以透過 getParent()來取得外層標籤實例，這對 Tag 介面相關類別之實作也是如此。9.4.2 節中提過，如果是巢狀標籤中的內層標籤，則還會呼叫標籤處理器的 setParent()方法，並傳入外層標籤處理器的實例。

同樣地，在這邊以開發<f:choose>、<f:when>與<f:otherwise>作為示範。首先是標籤處理器的開發：

TagDemo ChooseTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
```

```
public class ChooseTag extends TagSupport {
    private boolean matched;

    @Override
    public int doStartTag() throws JspException {
        return EVAL_BODY_INCLUDE;
    }

    public boolean isMatched() {
        return matched;
    }

    public void setMatched(boolean matched) {
        this.matched = matched;
    }
}
```

ChooseTag 基本上什麼都不作，之所以要重新定義 doStartTag()，因為 TagSupport 的 doStartTag() 方法預設傳回 SKIP_BODY，因為<f:choose>用來包括內層標籤，你不能忽略本體內容，所以必須傳回 EVAL_BODY_INCLUDE。

TagDemo WhenTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.tagext.JspTag;
import javax.servlet.jsp.tagext.Tag;

public class WhenTag extends BodyTagSupport {
    private boolean test;

    @Override
    public int doStartTag() throws JspException {
        JspTag parent = null;
        if (!((parent = getParent()) instanceof Tag)) {
            throw new JspTagException("必須置於 choose 標籤中");
        }
```

```
        if (((ChooseTag) parent).isMatched() || !test) {
            return SKIP_BODY;
        }

        ((ChooseTag) parent).setMatched(true);
        return EVAL_BODY_INCLUDE;
    }

    public void setTest(boolean test) {
        this.test = test;
    }
}
```

在這邊，doStartTag()基本上的檢查流程與 9.3.4 節類似，判斷是否包括在<f:choose>標籤中，判斷先前的<f:when>是否曾經通過測試，以決定是否要執行或忽略自己的本體內容。

TagDemo OtherwiseTag.java

```
package cc.openhome;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.JspTag;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;

public class OtherwiseTag extends TagSupport {
    @Override
    public int doStartTag() throws JspException {
        JspTag parent = null;
        if (!((parent = getParent()) instanceof Tag)) {
            throw new JspTagException("必須置於 choose 標籤中");
        }

        if (((ChooseTag) parent).isMatched()) {
            return SKIP_BODY;
        }

        ((ChooseTag) parent).setMatched(true);
```

```
        return EVAL_BODY_INCLUDE;
    }

}
```

基本上，OtherwiseTag 的 doStartTag()與 WhenTag 是類似的，只不過不用檢查 test 屬性。記得在 TLD 檔案中加入標籤定義：

TagDemo f.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" 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-jsptaglibrary_2_0.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>f</short-name>
    <uri>http://openhome.cc/jstl/fake</uri>
    // 略...
    <tag>
        <name>choose</name>
        <tag-class>cc.openhome.ChooseTag</tag-class>
        <body-content>JSP</body-content>
    </tag>
    <tag>
        <name>when</name>
        <tag-class>cc.openhome.WhenTag</tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>test</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>boolean</type>
        </attribute>
    </tag>
    <tag>
        <name>otherwise</name>
        <tag-class>cc.openhome.OtherwiseTag</tag-class>
        <body-content>JSP</body-content>
    </tag>
</taglib>
```

同樣地，你可以使用 9.3.4 節的 JSP 網頁來測試這邊自訂的<f:choose>、<f:when>與<f:otherwise>標籤。

提示　有機會的話，可以看看 JSTL 的實作原始碼，這是讓你更了解如何使用 Tag 介面下相關類別實作自訂標籤的最好方式。JSTL 的原始碼可以在這邊下載：
http://jakarta.apache.org/site/downloads/downloads_taglibs-standard.cgi

9.5 綜合練習／線上書籤

課後練習