

深入浅出 Servlet & JSP

1. Servlet 受控于容器: 通信支持, 生命周期管理, 多线程支持, 声明方式实现安全, JSP
 - a) 流程: 用户点击指向 servlet 的链接 ---- 容器创建 `HttpServletRequest` 和 `HttpServletResponse` 两个对象----容器创建线程, 把对象和请求传给 java 程序-----容器调用 `service()` 生成响应对象-----容器把相应用对象转化成 http 响应-----结束

```
public class Ch2Servlet extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws IOException {  
  
    PrintWriter out = response.getWriter(); ←  
    java.util.Date today = new java.util.Date();  
    out.println("<html> " +  
               "<body>" +  
               "<h1 style='text-align:center>" +  
               "HF's Chapter2 Servlet</h1>" +  
               "<br>" + today +  
               "</body>" +  
               "</html>");  
}  
}
```

- b) }
除 `printwriter` 以外还有别的
c) 一个 servlet 有三个名字

- i. 路径名
- ii. 部署名
- iii. URL 名 (在 HTML 中)

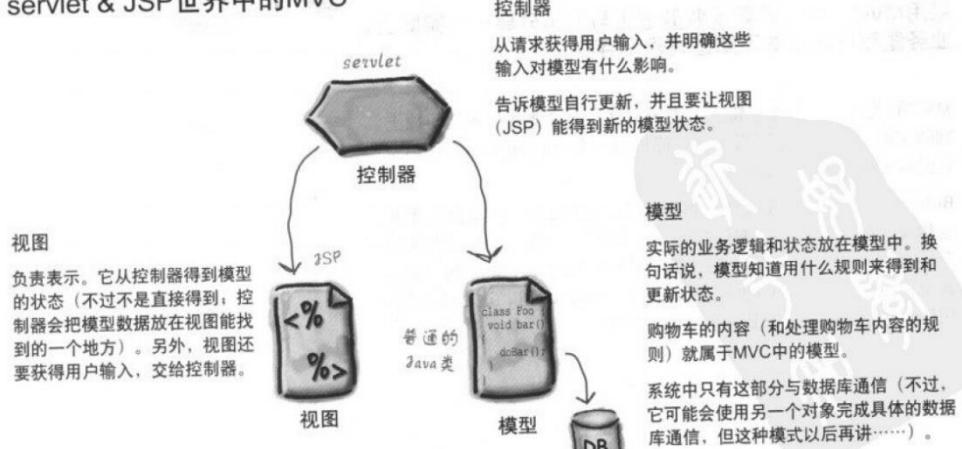
建立映射, 有利于 servlet 的灵活性和安全性

- d) Servlet 部署到 web 时, 会创建 XML 文档-> DD 部署描述文件: 提供一种声明定制 webapp, 不用修改源代码。DD 描述安全角色, 错误页面, 标记库等
<servlet> 映射内部名到类名。<web-app>前面不能有任何标签
<servlet-mapping> 映射内部名到 URL

```
<servlet> ←  
  <servlet-name>Internal name 1</servlet-name>  
  <servlet-class>foo.Servlet1</servlet-class>  
</servlet> ←  
  
<servlet>  
  <servlet-name>Internal name 2</servlet-name>  
  <servlet-class>foo.Servlet2</servlet-class>  
</servlet>  
.....  
  
<servlet-mapping> ←  
  <servlet-name>Internal name 1</servlet-name>  
  <url-pattern>/Public1</url-pattern>  
</servlet-mapping> ← 这是客户看到 (和使用
```

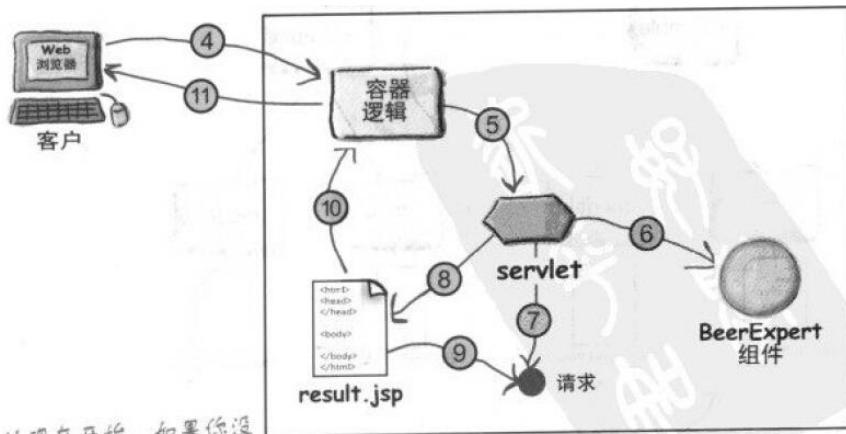
- e) Servlet 查询数据库，把输出任务委托给 JSP-> MVC
f) 业务逻辑应该单独放在一个类里，而不是放在 servlet 里

Servlet & JSP 世界中的 MVC

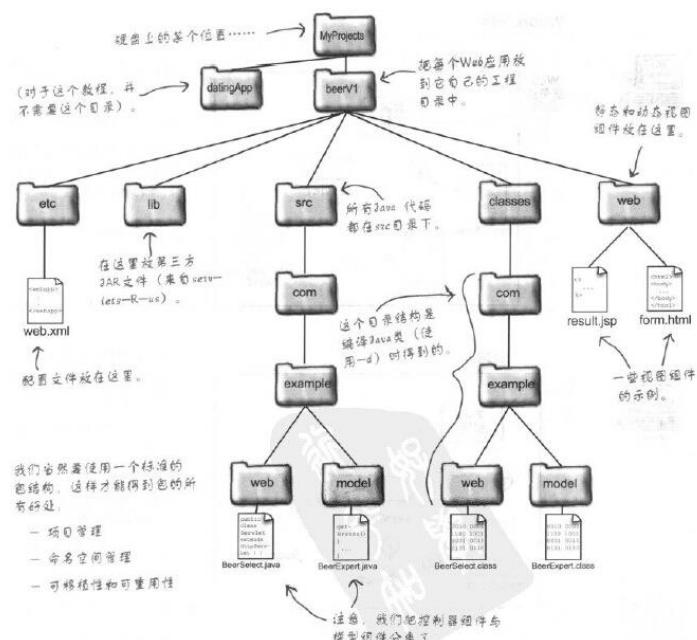


- g) J2EE: web 容器和 EJB 容器 (业务逻辑部分)

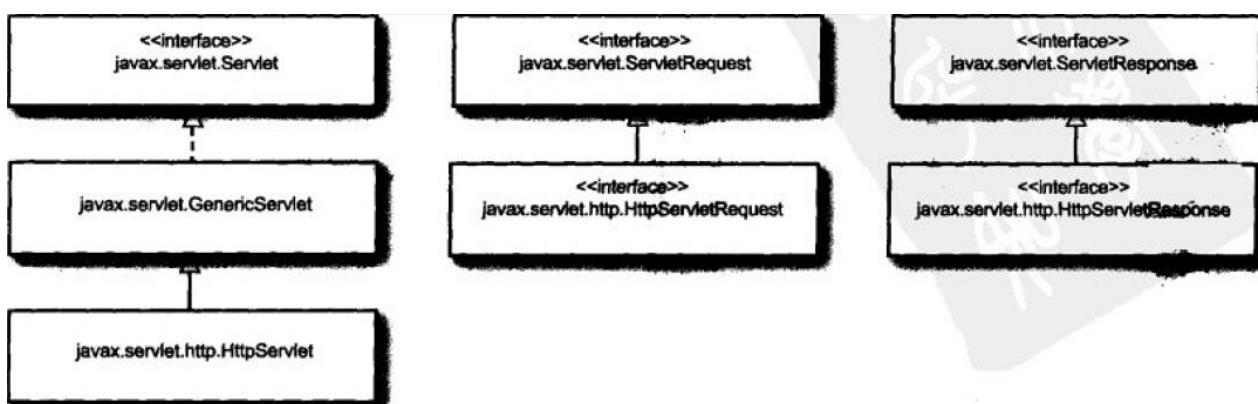
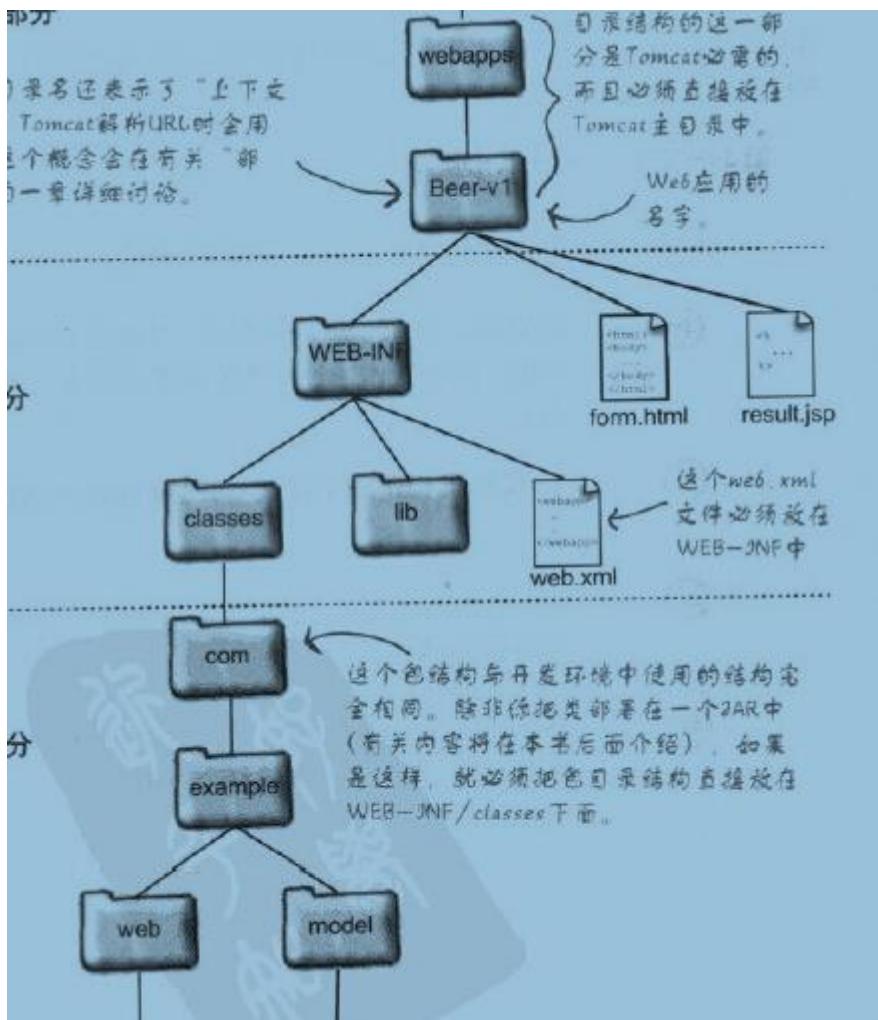
2. 第一个 Web 应用



a) 开发环境



b) 部署环境



c) Model 类

```

package com.example.model;
import java.util.*;

public class BeerExpert {
  
```

d) 调用 model

```
BeerExpert be = new BeerExpert();
List result = be.getBrands(c);
Iterator it = result.iterator();
while(it.hasNext()) {
    out.print("<br>try: " + it.next())
}
```

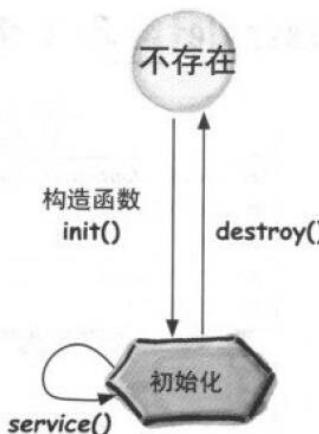
e) Controller 转发请求去 jsp

```
request.setAttribute("styles", result); ← 找

RequestDispatcher view =
    request.getRequestDispatcher("result.jsp"),
view.forward(request, response);

<%@ page import="java.util.*" %> ← 它要做什
<html>
<body>
<h1 align="center">Beer Recommendations JSP</h1> ←
<p>
<%
List styles = (List)request.getAttribute("styles");
Iterator it = styles.iterator();
while(it.hasNext()) {
    out.print("<br>try: " + it.next());
}
%>           标记里有一些标准
                Java代码（这称为scriptlet代码）。
</body>
</html>
```

3. Servlet 的生命周期: web 容器加载类---构造函数----init() |||-----service()---destroy()



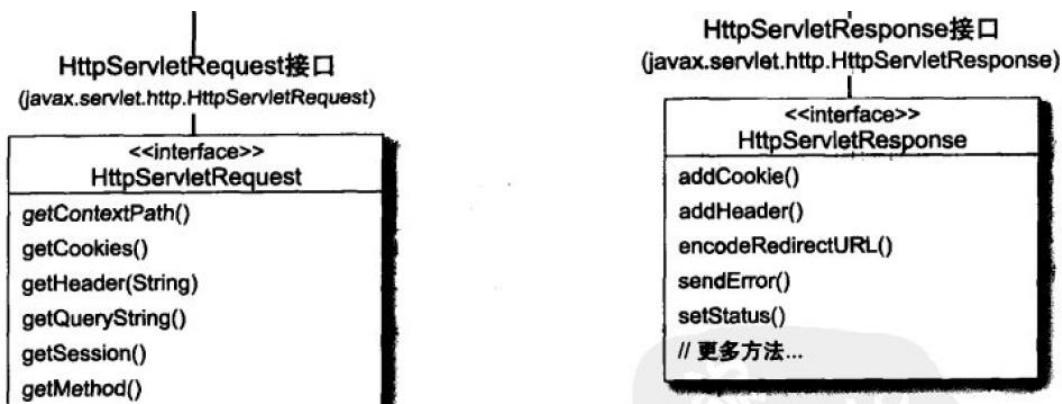
	何时调用	作用	是否覆盖?
1 init()	当servlet实例被创建之后，并在servlet能为客户请求提供服务之前，容器要对servlet调用init()。	使你在servlet处理客户请求之前有机会对其进行初始化。	可能。 如果有初始化代码（如得到一个数据库连接，或向其他对象注册），就要覆盖servlet类中的init()方法。
2 service()	当第一个客户请求到来时，容器会开始一个新线程，或者从线程池分配一个线程，并调用servlet的service()方法。	这个方法会查看请求，确定HTTP方法（GET、POST等），并在servlet上调用对应的方法，如doGet()、doPost()等。	不，不太可能。 不应该覆盖service()方法。你的任务是覆盖doGet()和/or doPost()方法，而由HTTPServlet中的service()实现来考虑该调用哪一个方法（doGet()、doPost()等）。
3 doGet() 和/or doPost()	service()方法根据请求的HTTP方法（GET、POST等）来调用doGet()或doPost()。 (这里只列出了doGet()和doPost()，因为你可能只会用到这两个方法。)	要在这里开始写你的代码！你的Web应用想要做什么，就要由这个方法负责。 当然，也可以在其他对象上调用其他方法，不过都要从这里开始。	至少要覆盖其中之一（doGet()或doPost()）。 不管覆盖哪一个，都会告诉容器你支持什么。例如，如果没有覆盖doPost()，就是在告诉容器这个servlet不支持HTTP POST请求。

- a) Servlet 对每个请求都会新开一个线程
- b) ServletConfig 对象和 ServletContext 对象

① ServletConfig 对象

- 每个servlet有一个ServletConfig对象。
- 用于向servlet传递部署时信息（例如，数据库或企业bean的查找名），而你不想把这个信息硬编码到servlet中（servlet初始化参数）。
- 用于访问ServletContext。
- 参数在部署描述文件中配置。

- c) 相关接口：HttpServletRequest 等



- d) 如果不加 method=POST, 会自动发送 get 请求
- e) 在 servlet 里获取 parameter

```
String colorParam = request.getParameter("color");
// 更多代码.....
```

(在这个例子中，String colorParam 的值为 "dark"。)

↑
这与表单中的参数名匹配。

```
String one = request.getParameterValues("sizes")[0];
String [] sizes = request.getParameterValues("sizes");
```

<<interface>>
ServletRequest
getAttribute(String)
getContentLength()
getInputStream()
getLocalPort()
getRemotePort()
getServerPort()
getParameter(String)
getParameterValues(String)
getParameterNames()
// 更多方法.....

<<interface>>
HttpServletRequest
getContextPath()
getCookies()
getHeader(String)
getIntHeader(String)
getMethod()
getQueryString()
getSession()
// 更多方法.....

F) getHeader 和 getIntHeader 的区别

问： getHeader()和getIntHeader()有什么区别？在我看来，首部都是String吧！！既然getIntHeader()方法取一个String表示首部的名，那这个方法名里的int是什么意思？

答： 首部有一个名（如“User-Agent”或“Host”），还有一个值[如“Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624 Netscape/7.1”或“www.wickedlysmart.com”]。从首部得来的值总是String，但是对于有些首部，这个String表示的是一个数。“Content-Length”首部就会返回消息体的字节数。例如，“Max-Forwards”HTTP首部会返回一个整数，指示请求可以经过的最大路由跳数（如果你想跟踪一个请求，看看它是不是陷入一个循环，就可以使用这个首部）。

可以使用getHeader()来得到“Max-Forwards”首部的值：

```
String forwards = request.getHeader("Max-Forwards");
int forwardsNum = Integer.parseInt(forwards);
```

这样是可以的。但是如果你已经知道首部的值表示一个整数，就可以使用getIntHeader()作为便利方法，这样就不用再多做一步把String解析为一个int，而是可以直接得到int：

```
int forwardsNum = request.getIntHeader("Max-Forwards");
```

f) 响应端的函数大多数情况下只发回数据，用 setContentType 即可

<<interface>> HttpServletResponse

addCookie()
addHeader()
encodeURL()
sendError()
setStatus()
sendRedirect()
// 更多方法……

但这并不是说绝对不能从servlet处理输出流。

为什么呢？

- 1) 你的提供商可能不支持JSP。还有很多比较老的服务器和容器，它们只支持servlet而不支持JSP，所以只能使用这种方法。
- 2) 出于其他原因可能没有办法使用JSP，比如你有一个很愚蠢的上司，不让你用JSP，原因只是1998年他姐夫告诉他JSP很不好。
- 3) 是谁说响应只能返回HTML？你还可以向客户返回不是HTML的其他东西。为此可能输出流更适合。

g) 用servlet返回文件

```
response.setContentType("application/jar");  
  
ServletContext ctx = getServletContext();  
InputStream is = ctx.getResourceAsStream("/bookCode.jar");  
  
int read = 0;  
byte[] bytes = new byte[1024];  
  
OutputStream os = response.getOutputStream();  
while ((read = is.read(bytes)) != -1) {  
    os.write(bytes, 0, read);  
}  
os.flush();  
os.close();
```

我们希望浏览器看出来这是一个JAR，而不是HTML，所以把内容类型设置为“application/jar”。

这仅表示“为名为bookCode.jar的资源给我一个输入流”。

这是关键部分，不过这只是普通的I/O而已！没有什么特别的，只是读取JAR字节，然后写至我们从响应对象得到的输出流。

servlet发回JAR呢？让Web服务器把它作为一个资源返回不就行了吗？换句话说，为什么不让用户点击一个指向JAR的链接，而要让链接指向一个servlet呢？难道不能把服务器配置为直接返回JAR而不要经过servlet吗？

答：不错，这是一个很好的问题。你确实可以这样配置Web服务器，让用户点击一个HTML链接，指向服务器上的某个资源，比如说JAR文件（就像其他静态资源一样，如JPEG和文本文件），服务器只是把它放在响应中返回。

不过……我们认为，在发回输出流之前你可能还想在servlet里做点别的事情。例如，你

println()写至PrintWriter
write()写至ServletOutputStream

一定要记住获得流或书写器的方法名，都是把返回类型的第一个词去掉：

ServletOutputStream
response.getOutputStream()
PrintWriter
response.getWriter()

h) 两种返回字节或者字符的方式（PrintWriter是装饰者模式的应用）

这只是一个普通的java.io，不过 `ServletResponse` 接口只提供了两个可以选择的流：`ServletOutputStream` 用于输出字节，`PrintWriter` 用于输出字符数据。

- i) `setHeader()` 增加一个新对，或者替换现有值；`addHeader()` 增加一个新对，或者给一个现有的首部增加一个值
- j) 重定向：必须在相应之前做
(绝对 URL) `sendRedirect()` 方法只接受 string，不接受 URL

```
response.sendRedirect("http://www.oreilly.com");
```

相对 URL：

假设客户原来键入的是：

`http://www.wickedlysmart.com/myApp/cool/bar.do`

请求到达名为“bar.do”的servlet时，这个servlet会基于一个相对URL来调用`sendRedirect()`，这个相对URL没有用斜线开头：

```
sendRedirect("foo/stuff.html");
```

容器会相对于原先的请求URL建立完整的URL（需要把它放在HTTP响应的“Location”首部中）：

`http://www.wickedlysmart.com/myApp/cool/foo/stuff.html`

- k) 请求分派：服务器端。重定向：客户端

2

可以用哪些HTTP方法向客户显示正在接收请求的是哪一个服务器？（选出所有正确的答案。）

- A. GET
- B. PUT
- C. TRACE - 这个方法通常用于调试除错，而不用于生产。
- D. RETURN
- E. OPTIONS

- l) HTTP 方法一览

GET	要求得到所请求URL上的一个东西（资源/文件）。
POST	要求服务器接受附加到请求的体信息，并提供所请求URL上的一个东西。这像一个扩展的GET……也就是说，随请求还发送了额外信息的“GET”。
HEAD	只要求得到GET返回结果的首部部分。所以这有点像GET，但是响应中没有体。它能提供所请求URL的有关信息，但是不会真正返回实际的东西。
TRACE	要求请求消息回送，这样客户能看到另一端上接收了什么，以便测试或排错。
PUT	指出要把所包含的信息（体）放在请求的URL上。
DELETE	指出删除所请求URL上的一个东西（资源/文件）。
OPTIONS	要求得到一个HTTP方法列表，所请求URL上的东西可以对这些HTTP方法作出响应。
CONNECT	要求连接以便建立隧道。

第五章 作为 WEB 应用

第六章 会话

1. 在 DD 中声明变量：这样的变量在 servlet 初始化以后才能调用，容器初始化 servlet 变量的时候会建立 servletconfig。只能 get 不能 set

```

<init-param>
    <param-name>adminEmail</param-name>
    <param-value>likewecare@wickedlysmart.com</param-value>
</init-param>

</servlet>

```

在servlet代码中：

```
out.println(getServletConfig().getInitParameter("adminEmail"));
```

2. 热部署：不需要关闭服务
3. 更加全局化的初始变量 Context：所有 servlet 公用（不在<servlet>标签里）

在DD文件 (web.xml) 中：

```
<servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>TestInitParams</servlet-class>
</servlet>

<context-param>
    <param-name>adminEmail</param-name>
    <param-value>clientheadererror@wickedlysmart.com</param-value>
</context-param>
```

重要提示!! <context-param> 是针对整个应用的，所以并不嵌套在某个<servlet>元素中!! 把<context-param> 放在<web-app>里，但是要放在所有<servlet>声明之外。

将<servlet>元素去掉。

指定一个 param-name 和一个 param-value，这与 servlet 初始化参数类似，不过这一次是在<context-param>元素中而非<init-param>元素中指定。

在 servlet 代码中：

```
out.println(getServletContext().getInitParameter("adminEmail"))
```

在<web-app>元素中，但是不在特定的<servlet>元素内

```
<web-app ...>
    <context-param>
        <param-name>foo</param-name>
        <param-value>bar</param-value>
    </context-param>

    <!-- other stuff including
         servlet declarations -->
</web-app>
```

注意，在有关上下文初始化参数的 DD 中没有任何地方提到“init”，而 servlet 初始化参数不同。

在每个特定 servlet 的<servlet>元素中

```
<servlet>
    <servlet-name>
        BeerParamTests
    </servlet-name>
    <servlet-class>
        TestInitParams
    </servlet-class>
    <init-param>
        <param-name>foo</param-name>
        <param-value>bar</param-value>
    </init-param>

    <!-- other stuff -->
</servlet>
```

4. ServletContext 的相关接口

可以用两种方式得到 ServletContext……

servlet 的 ServletConfig 对象拥有该 servlet 的 ServletContext 的一个引用。所以如果考试时看到这样的 servlet 代码，请不要奇怪：

```
getServletConfig().getServletContext().getInitParameter()
```

这样做不仅是合法的，而且与下面的代码是等价的：

```
this.getServletContext().getInitParameter()
```

但是，在一个非 servlet 的类中（例如，一个辅助/工具类），该怎么做呢？有人可能为这个类传递了一个 ServletConfig，类代码必须使用 getServletContext() 来得到 ServletContext 对象的一个引用。

5. 监听上下文的操作，在 servlet 运行之前运行一段代码：ServletContextListener

<i><<interface>></i>
<i>ServletContext</i>
<i>getInitParameter(String)</i>
<i>getInitParameterNames()</i>
<i>getAttribute(String)</i>
<i>getAttributeNames()</i>
<i>setAttribute(String, Object)</i>
<i>removeAttribute(String)</i>
<i>getMajorVersion()</i>
<i>getServerInfo()</i>
<i>getRealPath(String)</i>
<i>getResourceAsStream(String)</i>
<i>getRequestDispatcher(String)</i>
<i>log(String)</i>
// 更多方法

- 上下文初始化时得到通知（应用得到部署）。
 - 从ServletContext得到上下文初始化参数。
 - 使用初始化参数查找名建立一个数据库连接。
 - 把数据库连接存储为一个属性，使得Web应用的各个部分都能访问。

- 上下文撤销时得到通知（应用取消部署或结束）。
 - 关闭数据库连接。

```
import javax.servlet.*;    javax.servlet包中。
public class MyServletContextListener implements ServletContextListener {
    public void contextInitialized(ServletContextEvent event) {
        //初始化数据库连接的代码
        //and store it as a context attribute
    }
    public void contextDestroyed(ServletContextEvent event) {
        //关闭数据库连接的代码
    }
}
```

这是要得到的
会提供一个Servlet

6. 监听者的类放在 WEB-INF 的 classes 下（model 类也会放在这里），并且在 web.xml 里头加入<listener>标签(放在<web-app>标签下)

```
<listener>
    <listener-class>
        com.example.MyServletContextListener
    </listener-class>
</listener>
```

7. 获得初始化参数，创建一个对象并储存在上下文里注意事项

1. 执行 ServletContextListener 接口，重写 contextInitialized 方法
2. 调用 event.getServletContext 方法来得到上下文
3. 使用 getInitParameter 的方法得到部署变量
4. 使用 setAttribute 方法把变量保存到 Context
5. 被存储的对象最好是可序列化的
6. 取回对象时需要强制类型转换

```

public class MyServletContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = event.getServletContext(); ← 由事件得到ServletContext。
        String dogBreed = sc.getInitParameter("breed"); ← 使用上下文得到初始化参数。
        Dog d = new Dog(dogBreed); ← 建立一个新Dog。
        sc.setAttribute("dog", d); ← 使用上下文来设置属性(名/对象对)。
    } ← 这个属性就是Dog。现在应用的其他部分就能得到属性(Dog)的值了……
}

```

Dog dog = (Dog) getServletContext().getAttribute("dog");

注意：request 对象同样有 setAttribute 方法

- ③ 把3个已编译的类文件放在Tomcat的Web应用目录结构中

ListenerTester.class
MyServletCont

```

listenerTest/WEB-INF/classes/com/example/Dog.class
listenerTest/WEB-INF/classes/com/example/ListenerTester.class
listenerTest/WEB-INF/classes/com/example/MyServletContextListener.class

```

- ④ 把web.xml部署描述文件放在此Web应用的WEB-INF目录中

listenerTest/WEB-INF/web.xml

- ⑤ 关闭Tomcat后再重启，从而部署应用

8. 可以监听很多其他事件

场景	监听者接口	事件类型
你想知道一个Web应用上下文中是否增加、删除或替换了一个属性。	javax.servlet.ServletContextAttributeListener attributeAdded attributeRemoved attributeReplaced	ServletContextAttributeEvent
你想知道有多少个并发用户。也就是说，你想跟踪活动的会话（下一章再详细介绍会话）。	javax.servlet.http.HttpSessionListener sessionCreated sessionDestroyed	HttpSessionEvent
每次请求到来时你都想知道，以便建立日志记录。	javax.servlet.ServletRequestListener requestInitialized requestDestroyed	ServletRequestEvent
你有一个属性类，而且希望此类对象绑定的会话迁移到另一个JVM时得到通知。	javax.servlet.http.HttpSessionActivationListener sessionDidActivate sessionWillPassivate	HttpSessionEvent 不是 "HttpSessionActivationEvent"

你想知道什么时候增加、删除或替换一个请求属性。	javax.servlet.ServletRequestAttributeListener attributeAdded attributeRemoved attributeReplaced	ServletRequestAttributeEvent
你有一个属性类（这个类表示的对象将被放在一个属性中），而且你希望这个类型的对象绑定到一个会话或从会话删除时得到通知。	javax.servlet.http.HttpSessionBindingListener valueBound valueUnbound	HttpSessionBindingEvent
你想知道什么时候增加、删除或替换一个会话属性。	javax.servlet.http.HttpSessionAttributeListener attributeAdded attributeRemoved attributeReplaced	HttpSessionBindingEvent <small>注意，这里的命名不一致！对应 HttpSessionAttributeListener 的事件可能和你预想的不一样（你可能以为是 HttpSessionAttributeEvent）。</small>
你想知道是否创建或撤销了一个上下文。	javax.servlet.ServletContextListener contextInitialized contextDestroyed	ServletContextEvent

8. HttpSessionAttributeListener 和 HttpSessionBindingListener 的区别：后者是针对对象的，对象想知道自己何时被绑定在一个 session 上（无需在 DD 里注册）

普通的 HttpSessionAttributeListener 类只是想知道会话中何时增加、删除或替换了某种类型的属性。但是 HttpSessionBindingListener 不同，有了 HttpSessionBindingListener，属性本身才能知道它何时增加到一个会话中，或者何时从会话删除。

```
public class Dog implements HttpSessionBindingListener {
    private String breed;
    public Dog(String breed) {
        this.breed=breed;
    }
    public String getBreed() {
        return breed;
    }
    public void valueBound(HttpSessionBindingEvent event) {
        // 如果我知道我在一个会话中，就运行这些代码
    }
    public void valueUnbound(HttpSessionBindingEvent event) {
        // 如果我知道我不再在一个会话中，就运行这些代码
    }
}
```

这一次 Dog 属性也是一个监听者……它会监听 Dog 本身何时增加到会话或从会话删除……而会自动进行。

这里 bound (绑定) 的含义是“”

9. 什么是属性

属性就是一个对象，设置（也称为绑定）到另外3个 servlet API 对象中——ServletContext、 HttpServletRequest（或 ServletRequest）或者 HttpSession。可以把它简单地认为是一个映射实例对象中的名/值对（名是一个 String，值是

10. 属性和参数（部署在 DD 里的）的区别

属性		参数
类型	应用/上下文 请求 会话	应用/上下文初始化参数 请求参数 servlet初始化参数 没有会话参数一说!
设置方法	setAttribute(String name, Object value)	不能设置应用和servlet初始化参数, 它们都在DD中设置, 还记得吧? (对于请求参数, 可以调整查询串, 但这是另一回事。)
返回类型	Object	String ← 这是一个突出的区别!
获取方法	getAttribute(String name) 不要忘了必须强制转换, 因为返回类型是Object。	getInitParameter(String name)

11. 三个作用域的区别

	可访问性 (谁能看到)	作用域 (能存活多久)	适用于……
Context (上下文) (不是线程安全的!)	Web应用的所有部分, 包括servlet、JSP、Servlet-ContextListener、ServletContextAttributeListener。	ServletContext的生命期, 这意味着所部署应用的生命期。如果服务器或应用关闭, 上下文则撤销(其属性也相应撤销)。	你希望整个应用共享的资源, 包括数据库连接、JNDI查找名、email地址等。
HttpSession (会话) (不是线程安全的!)	访问这个特定会话的所有servlet或JSP。要记住, 会话从一个客户请求扩展到可能跨同一个客户的多个请求, 这些请求可能到达多个servlet。	会话的生命期。会话可以通过编程撤销, 也可能只是因为超时而撤销(有关细节将在关于“会话管理”的一章介绍)。	与客户会话有关的资源和数据, 而不只是与一个请求相关的资源。它要与客户完成一个持续的会话。购物车就是一个典型的例子。
Request (请求) (线程安全)	应用中能直接访问请求对象的所有部分。基本说来, 这意味着接收所转发请求的JSP和servlet(使用RequestDispatcher), 另外还有与请求相关的监听者。	请求的生命期, 这说明会持续到servlet的service()方法结束。换句话说, 就是线程(栈)处理这个请求的整个生命期。	将模型信息从控制器传递到视图……或者传递特定于客户请求的任何数据。

12. 属性 API

3个属性作用域（上下文、请求和会话）
分别由ServletContext、ServletRequest和
HttpSession接口处理。每个接口中的属性
API方法完全相同。

Object getAttribute(String name)
setAttribute(String name, Object value)
removeAttribute(String name)
Enumeration getAttributeNames()

13. 注意：上下文作用域不是线程安全的

不能把doGet()设计成同步方法：（编程）不能进行并发，也不能阻止其他Servlet
访问

不能把访问和设置上下文设计成同步方法：大量额外开销

解决方案：访问上下文时加锁（注意括号里的参数）

```
synchronized(getServletContext()) {           (而不)
    getServletContext().setAttribute("foo", "22");
    getServletContext().setAttribute("bar", "42");

    out.println(getServletContext().getAttribute("foo"));
    out.println(getServletContext().getAttribute("bar"));

}
```

14. 会话和实例变量也不是线程安全的（客户可以用一个浏览器的多个窗口同时发出请 求）

```
synchronized(session) {           这一次
    session.setAttribute("foo", "22");
    session.setAttribute("bar", "42");

    out.println(session.getAttribute("foo"));
    out.println(session.getAttribute("bar"));

}
```

15. 请求分配：requestDispatcher

RequestDispatcher 深密

RequestDispatcher只有两个方法：forward()和include()。这两个方法都取请求和响应对象为参数（接收转发请求的组件需要这些对象来完成任务）。在这两个方法中，forward()是目前最常用的。一般不太可能从控制器servlet调用include方法；不过，在后台，JSP可能在<jsp:include>标准动作（第8章将介绍）中调用include方法。可以采用两种办法来得到RequestDispatcher：从请求得到，或者从上下文得到。不论怎么得到，都必须告诉它要把请求转发给哪个Web组件，也就是要告知接管请求的servlet或JSP。

```
<<interface>>
RequestDispatcher
forward(ServletRequest, ServletResponse)
include(ServletRequest, ServletResponse)

javax.servlet.RequestDispatcher
```

从ServletRequest得到RequestDispatcher

```
RequestDispatcher view = request.getRequestDispatcher("result.jsp");
```

在RequestDispatcher上调用forward()

```
view.forward(request, response);
```

Include()把响应派发给别人然后再返回给调用者

注意：调用 flush()或者 write()后不能在转发请求：转发必须在响应之前

1. 如何保存用户状态

1. JAVABean 有状态会话
2. 数据库
3. HttpSession

2. 使用 HttpSession: 创建和获得一个新会话

在响应中发送一个会话cookie:

```
HttpSession session = request.getSession()  
HttpSession session = request.getSession(); ↵  
  
if (session.isNew()) { ↵  
    ↪ 如果客户还没有用这个会话ID做过响应，isNew()就  
    ↪ 返回true。
```

16. 如何得到一个旧会话: getSession(false)判断是否是旧会话

问： 通过调用request.getSession()可以得到一个会话，但这是得到会话的唯一方法吗？能否从ServletContext得到一个会话？

答： 你能从请求对象得到会话，这是因为……想看……会话是由请求标识的。在请求上调用getSession()时，就是在说：“我想为这个客户要一个会话……这个会话可以与客户发出的会话ID匹配，也可以是一个新的会话。但是不论怎样，会话对应的客户要与这个请求关联。”

但是，确实还有另外一种办法来得到会话……从会话事件对象也能得到会话。要记住，监听者类不是servlet或JSP，它只是一个想知道发生了某些事件的类。例如，监听者可

能是一个属性，想知道自己（属性对象）是否增加到一个会话，或者是否从一个会话删除。

与会话相关的监听者接口定义了事件处理方法，这些事件处理方法取一个类型为HttpSessionEvent的参数（或其子类HttpSessionBindingEvent）。HttpSessionEvent就有一个getSession()方法！

所以，如果实现了与会话相关的4种监听者接口之一（本章稍后将介绍），就可以通过事件处理回调方法来访问会话。例如，以下代码取自一个实现了 HttpSessionListener 接口的类：

```
public void sessionCreated(HttpSessionEvent event){  
    HttpSession session = event.getSession();  
    // 事件处理代码  
}
```

17. URL 重写: 该函数会自动调用 encodeURL 重写。当容器看到 getSession 时会使用双保险：如果调用了 encodeURL 一定会把 jsessionid 附在后面（或者使用重定向）

```
out.println("<a href=\"" + response.encodeURL("/BeerTest.do") + "\">click me</a>");
```

```
response.encodeRedirectURL("/BeerTest.do")
```

你不会直接用到“jsessionid”。如果看到一个“jsessionid”请求参数，说明肯定有错误。绝对不应该看到下面这样的代码：

```
String sessionId = request.getParameter("jsessionid");
```

而且，在请求或响应中，不应该看到定制的“jsessionid”首部：

POST /select/selectBeerTaste.do HTTP/1.1

User-Agent: Mozilla/5.0

JSESSIONID: 0AAB6C8DE415 ← 不能这样做！这样会被认为是一个首部！

实际上，惟一可以放“jsessionid”的地方就是在cookie首部中：

POST /select/selectBeerTaste.do HTTP/1.1

User-Agent: Mozilla/5.0

Cookie: JSESSIONID=0AAB6C8DE415

这是对的，但是不要自己这样做。

或者作为“额外信息”追加到URL的最后：

URL重写的结果（同样不要自己这样做）。

POST /select/selectBeerTaste.do;jsessionid=0AAB6C8DE415

18. HttpSession 接口：会话也可以使用 setAttribute 等一套算法

```
<<interface>>
javax.servlet.http.HttpSession
Object getAttribute(String)
long getCreationTime()
String getId()
long getLastAccessedTime()
int getMaxInactiveInterval()
ServletContext getServletContext()
void invalidate()
boolean isNew()
void removeAttribute(String)
void setAttribute(String, Object)
void setMaxInactiveInterval(int)
// 更多方法
```

不要忘了，encodeURL()方法是HttpServletResponse对象上调用的方法！不能在请求上调用这个方法，也不能在上下文上调用此方法。要提醒自己，URL编码只与响应有关。

问： URL重写是不是以开发商特定的方式处理？

答： 对，URL重写确实以开发商特定的方式来处理。Tomcat使用分号“；”将额外的信息追加到URL。其他的开发商可能使用逗号或其他分隔符。另外，Tomcat在重写的URL中增加了“jsessionid=”，但其他开发商可能只是追加会话ID本身。关键是，不论容器使用什么分隔符，请求到来时容器都能认出。所以，当容器看到它使用的分隔符时（换句话说，就是它在URL重写时增加的分隔符），就知道在此分隔符后面的所有内容都是容器放在这里的“额外信息”。也就是说，容器知道如何识别和解析它（容器）追加到URL的额外内容。

	它做什么	你用它做什么
getCreationTime()	返回第一次创建会话的时间。	得出这个会话有多“老”。你可能想把某些会话的寿命限制为一个固定的时间。例如，你可能会说“一旦登录，就必须在10分钟之内完成这个表单……”
getLastAccessedTime()	返回容器最后一次得到有此会话ID的请求的时间（毫秒数）。	得出客户最后一次访问这个会话是什么时候。可以用这个方法来确定客户是否已经离开很长时间，这样就可以向客户发出一封email，询问他们是否还回来。或者可以调用invalidate()结束会话。
setMaxInactiveInterval()	对于此会话，指定客户请求的最大间隔时间（秒数）。	如果过去了指定的时间，而客户未对此会话做任何请求，就会导致会话被撤销。可以用这个方法减少服务器中无用会话的个数。
getMaxInactiveInterval()	对应此会话，返回客户请求的最大间隔时间（秒数）。	得出这个会话可以保持多长时间不活动，而且仍“活着”。可以使用这个方法来判断一个不活动的客户在会话撤销之前还有多长的“寿命”。
invalidate()	结束会话。当前存储在这个会话中的所有会话属性也会解除绑定（更多内容见本章后面的介绍）。	如果客户已经不活动，或者你知道会话已经结束（例如，客户完成了购物结账，或已经注销），可以用这个方法删除会话。会话实例本身可能由容器回收，但是这一点我们并不关心。置无效（Invalidate）意味着会话ID不再存在，而且属性会从会话对象删除。

19. 结束会话的方法：超时、invalidate()、0 活跃时间（-1 活跃时间为永不过期）和应用结束（DD 中只能用分钟，程序只能按秒）

在DD中配置会话超时

在DD中配置会话超时与在所创建的每个会话上调用setMaxInactiveInterval()有同样的效果。

```
<web-app ...>
  <servlet>
    ...
  </servlet>
  <session-config>
    <session-timeout>15</session-timeout>
  </session-config>
</web-app>
```

“15”是指15分钟。这是说，如果客户15分钟没有对这个会话做任何请求，就杀死它。

```
session.setAttribute("foo", "42");
session.setMaxInactiveInterval(0);
```

20. 设置一个 cookie 的方法（从 request 里拿到 cookie，response 里头加 cookie）
Cookie 的方法有 getName() 和 getValue()

创建一个新Cookie

```
Cookie cookie = new Cookie("username", name);
```

设置cookie在客户端上存活多久

```
cookie.setMaxAge(30*60);
```

setMaxAge以秒为
端上存活30*60秒
置为-1，那么浏
如果在“SESSION”
你知道会返回什

把cookie发送到客户

```
response.addCookie(cookie);
```

从客户请求得到cookie (或多个cookie)

```
Cookie[] cookies = request.getCookies();
for (int i = 0; i < cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getName().equals("username")) {
        String userName = cookie.getValue();
        out.println("Hello " + userName);
        break;
    }
}
```

21. Cookie 和首部的添加方式不一样

向响应增加一个首部时，会把名和值String作为参数
传入：

```
response.addHeader("foo", "bar");
```

但是向响应增加一个Cookie时，要传递一个Cookie对
象。需要在Cookie构造函数中设置Cookie名和值。

```
Cookie cookie = new Cookie("name", name);
response.addCookie(cookie);
```

还要记住，对于首部既有setHeader()方法，又有ad-
dHeader()方法（如果有这个首部，addHeader会
向这个现有的首部增加一个值，而setHeader会替换
现有的值）。但是不存在setCookie()方法。只有一个
addCookie()方法！

22. 会话的钝化和再激活：跨 servlet 会话

迁移

会话准备钝化

容器打算把会话迁移（移动）到另一个VM中。要在会话移动之前
调用，这样就能让属性有机会做好迁移的准备。

HttpSessionEvent



HttpSessionActivationListener

会话已经激活

容器已经把会话迁移（移动）到另一个VM中。要在应用的其他
部分对会话调用getAttribute()之前调用，这样刚移动的属性就有
机会做好访问的准备。

23. HttpSessionListener, HttpSessionActivationListener 和 HttpSessionAttributeListener 都 必须在 DD 里注册，但是 HttpSessionBindingListener 和不需要在 DD 里注册

24. 会话迁移：容器需要迁移可序列化属性，但是不保证使用序列化方式

解决办法：1. 把所有属性类都设置成可序列化的

2. 让属性类实现 HttpSessionActivationListener 接口，通过两个函数迁移

27. HttpSessionListener 实例（使用静态变量）

```
package com.example;
import javax.servlet.http.*;

public class BeerSessionCounter implements HttpSessionListener {
    static private int activeSessions;
    public static int getActiveSessions() { ←
        return activeSessions;
    }

    public void sessionCreated(HttpSessionEvent event) { ←
        activeSessions++;
    }

    public void sessionDestroyed(HttpSessionEvent event) { ←
        activeSessions--;
    }
}

<web-app ...>
...
<listener>
    <listener-class>
        com.example.BeerSessionCounter
    </listener-class>
</listener>
</web-app>
```

这个类会像所有其他在
WEB-INF/classes 中。
辅助类都能访问这个；

（以上方法可能在多 jvm 环境里无法实现）

25. HttpSessionAttributeListener 实例

```
public class BeerAttributeListener implements HttpSessionAttributeListener {

    public void attributeAdded(HttpSessionBindingEvent event) {
        String name = event.getName(); } 利用 HttpSessionBindingEvent，  
Object value = event.getValue(); } 可以得到触发此事件的属性的名  
和值。
        System.out.println("Attribute added: " + name + ":" + value);
    }

    public void attributeRemoved(HttpSessionBindingEvent event) {
        String name = event.getName();
        Object value = event.getValue();
        System.out.println("Attribute removed: " + name + ":" + value);
    }

    public void attributeReplaced(HttpSessionBindingEvent event) {
```

利用 HttpSessionBindingEvent，
可以得到触发此事件的属性的名
和值。

```

<web-app ...>
...
<listener>
    <listener-class>
        com.example.BeerAttributeListener
    </listener-class>
</listener>
</web-app>

```

DD 注册

26. 调用 System.out 会 print 到那里去?

答：会输出到容器选择的任何发送目标（可能让你配置，也可能不让你配置）。换句话说，如果与特定的开发商相关，发送目标通常是一个日志文件。Tomcat 就会把输出放在 tomcat/logs/catalina.log 中。你要阅读你的服务器文档来了解容器如何处理标准输出。

27. 其他两个 Listener 的实例

```

public class Dog implements HttpSessionBindingListener,
    HttpSessionActivationListener, Serializable {
    private String breed;
    // 假设这里还有更多实例变量，包括
    // 非Serializable的实例变量

    // 假设这里是构造函数以及其他获取方法和设置方法

    public void valueBound(HttpSessionBindingEvent event) {
        // 我知道我在一个会话中时要运行的代码
    }

    public void valueUnbound(HttpSessionBindingEvent event) {
        // 我知道已经不在一个会话中时要运行的代码
    }

    public void sessionWillPassivate(HttpSessionEvent event) {
        // 这些代码将非Serializable字段置为某种状态，
        // 以便顺利地迁移到一个新VM
    }
}

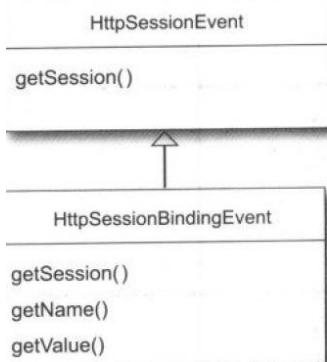
```

28. 会话监听者整理

场景	监听者接口/方法	事件类型	通常由谁实现
你想知道有多少个并发用户。也就是说，你想跟踪活动的会话。	HttpSessionListener (javax.servlet.http) sessionCreated sessionDestroyed	HttpSessionEvent	<input checked="" type="checkbox"/> 属性类 <input checked="" type="checkbox"/> 其他类
你想知道会话何时从一个 VM 移到另一个 VM。	HttpSessionActivationListener (javax.servlet.http) sessionDidActivate sessionWillPassivate	HttpSessionEvent 注意：没有指定的 HttpSessionActivationEvent。	<input checked="" type="checkbox"/> 属性类 <input checked="" type="checkbox"/> 其他类

有一个属性类（这个类的对象要用作为属性值），而且你希望此类对象绑定到会话或从会话删除时得到通知。	HttpSessionBindingListener (javax.servlet.http) valueBound valueUnbound	HttpSessionBindingEvent	 属性类 <input type="checkbox"/> 其他类
你想知道会话中什么时候增加、删除或替换会话属性。	HttpSessionAttributeListener (javax.servlet.http) attributeAdded attributeRemoved attributeReplaced	HttpSessionBindingEvent <i>注意：没有稳定的 HttpSessionAttributeEvent。</i>	 属性类 <input type="checkbox"/> 其他类

29. 两种事件的接口



第七章 JSP

1. JSP 到底是什么

最后，JSP会变成一个servlet

你的JSP最终还是会变成一个完整的servlet在Web应用中运行。它与其他的servlet很相似，只不过这个servlet类会由容器为你写好。

容器拿到你在JSP中写的代码，把这些代码转换为一个servlet类源(.java)文件，然后再把这个源文件编译为Java servlet类。在此之后，这就是名符其实的servlet了，而且这个servlet会与你自己编写和编译的servlet代码一样地运行。换句话说，容器会加载servlet类，对其实例化并初始化，为每个请求建立一个单独的线程，并调用servlet的service()方法。

2. 使用 page 指令导入包

```
<%@ page import="foo.* , java.util.*" %>
    ↑
    使用逗号来分隔多个包。整个包列表要用引
    号引起来！
```

注意到打印计数器的Java代码与page指令有什么差别吗？

Java代码放在有百分号的尖括号中间：`<% 和 %>`。而指令会为元素开始记号再增加一个字符`@!`

如果看到以`<%@`开始的JSP代码，应该知道这是一个指令（本书后面还会更详细地介绍page指令）。

3. 打印表达式结果不需要 `println`, 表达式会被自动转化，所以不需要加分号

表达式代码

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is now:
<%= Counter.getCount() %>
</body>
</html>
```

表达式更短一些，我们不需要显式地进行打印……

表达式会成为 `out.print()` 的参数

不要在表达式的最后加上分号!

```
<%= ((Math.random() + 5)*2); %>
```

不对！这里不能有分号。

注意到scriptlet代码的标记与表达式的标记有什么不同吗？scriptlet代码介于带百分号的尖括号中间：`<%`和`%>`。表达式会为元素的开始记号再增加一个字符——一个等号(`=`)。

到目前为止，我们已经看到了3种不同类型的JSP元素：

scriptlet: `<% %>`

指令: `<%@ %>`

表达式: `<%= %>`

问： 在表达式中，如果方法没有返回任何东西，会发生什么情况？

答： 会得到一个错误!! 不能，绝对不能把一个返回类型为 `void` 的方法用作表达式。容器很聪明，它知道，如果方法的返回类型为 `void`，就打印不出任何东西！

4. 变量域

```

<html>
<body>
<% int count=0; %>
The page count is now:
<%= ++count %>
</body>      让 count 增加1。
</html>

```

不会正常工作

5. Jsp 的实例变量声明

```

<%! int count=0; %>

```

↑ ↑
百分号 (%) 后面要 这不是表达式，这
放一个感叹号 (!)。 里需要有分号！

JSP 声明用于声明所生成 servlet 类的成员。这说明可以声明变量和方法！换句话说，`<%!` 和 `%>` 标记之间的所有内容都会增加到类中，而且置于服务方法之外。这意味着你可以声明静态变量和方法，还可以声明实例变量和方法。

6. JSP 的方法声明

方法声明

这个JSP:

会变成这个servlet:

```

public class basicCounter_jsp extends SomeSpecial HttpServlet {
    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response)
        throws IOException, ServletException {
        // ...
    }

    <%!
        int doubleCount() {
            count = count*2;
            return count;
        }
    %>
}

    int count=1;
    public int doubleCount() {
        count = count*2;
        return count;
    }
}

```

↑ ↑
这个方法与你在JSP中键入
的方法完全相同。
由于是Java，所以超前引用没有问题（先在方
法中使用变量，然后才声明这个变量）。

7. 容器如何处理 JSP

- ▶ 查看指令，得到转换时可能需要的信息。

- ▶ 创建一个HttpServlet子类。

对于Tomcat 5, 所生成的servlet会扩展：

org.apache.jasper.runtime.HttpJspBase

- ▶ 如果一个page指令有import属性，它会在类文件的最上面（package语句下面）写import语句。

对于Tomcat 5, package语句（你不用操心）如下：

package org.apache.jsp;

- ▶ 如果有声明，容器将这些声明写到类文件中，通常放在类声明下面，并在服务方法前面。Tomcat 5声明了自己的一个静态变量和一个实例方法。

- ▶ 建立服务方法。服务方法的具体方法名是 _jspService()。所生成的servlet会覆盖servlet超类的service()方法，_jspService()就由覆盖的service()方法调用，要接收HttpServletRequest和HttpServletResponse参数。在建立这个方法时，容器会声明并初始化所有隐式对象（在下一页你会看到更多隐式对象）。
- ▶ 将普通的HTML（称为模板文本）、scriptlet和表达式放到服务方法中，完成格式化，并写至PrintWriter响应输出。

8. JSP 隐式变量

API	隐式对象
JspWriter	out
HttpServletRequest	request
HttpServletResponse	response
HttpSession	session
ServletContext	application
ServletConfig	config
JspException	exception
PageContext	pageContext
Object	page

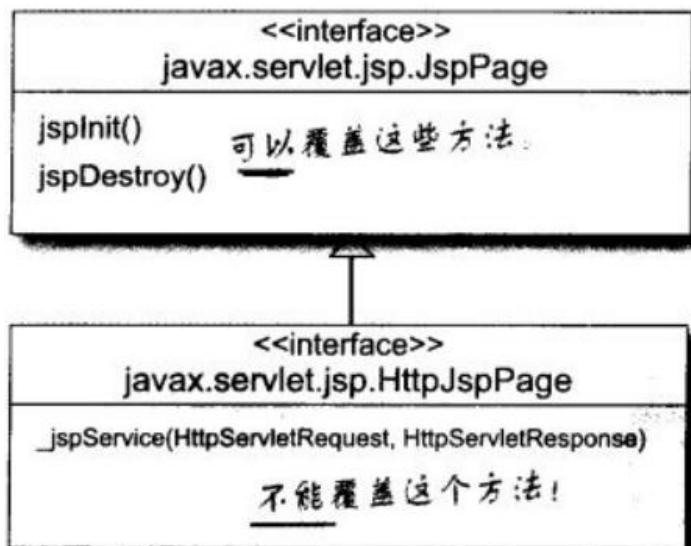
9. JSP 注释不会传回给客户

► <!-- HTML 注释 -->

容器把它直接传递给客户，在客户端，浏览器会把它解释为注释。

► <%-- JSP 注释 --%>

这些注释是为页面开发人员提供的，就像是Java源文件中的Java注释一样，转换页面时会把这些注释去掉。如果你键入了一个JSP，并想放些注释来说明你要做什么，就要像Java源文件中加注释一样地使用JSP注释。如果你希望注释作为HTML响应的一部分交给客户（不过浏览器会把它们隐藏起来，不让客户看到），就要使用HTML注释。



10. JSP 关键方法

11. JSP 的转换和编译只会发生在第一次使用

口： 尽管只是第一个客户需要等待，但大多数容器开发商确实提供了一种办法，可以让整个转换/编译工作提前进行，这样甚至第一个请求也能像所有其他servlet请求一样很快得到处理。

但是要当心，这取决于具体的开发商，而且不能保证。JSP规范(JSP 1.1.4.2)中确实提到了一种推荐的JSP预编译协议。向JSP作出请求时可以追加一个查询串“?jsp_precompile”，容器可能（如果支持）会立即完成转换/编译，而不是等到第一个请求真正到来时才进行转换和编译。

12. JSP 的初始化（使用 DD）

```
<web-app ...>
...
<servlet>
    <servlet-name>MyTestInit</servlet-name>
    <jsp-file>/TestInit.jsp</jsp-file> ←
    <init-param>
        <param-name>email</param-name>
        <param-value>wecare@wickedlysmart.com</param-value>
    </init-param>
</servlet>
```

只有这一行与常规的servlet不同。它实际上在说：“把这个<servlet>标记中的所有配置应用到由此JSP页面生成的servlet……”

13. JSP 的初始化（使用 jsplInit）：不能访问 response 和 request 对象，只能访问 servletConfig 和 servletContext 两个对象

```

<%! ← 使用一个声明来覆盖
    jspInit()方法。
public void jspInit() {
    ServletConfig sConfig = getServletConfig();
    String emailAddr = sConfig.getInitParameter("email"); ← 因为这是在一个servlet中，所以可以
                                                                调用继承的getServletConfig()方法！
    ServletContext ctx = getServletContext(); ← 这与在正常servlet中的做
    ctx.setAttribute("mail", emailAddr);      ← 得到ServletContext的一个引用，并设置一个应用作用域属性。
}
%>

```

14. 关键对象差别：对于 JSP 和 Servlet (exception 也是合法的 jsp 隐式变量)

	servlet中	JSP中 (使用隐式对象)
应用	<code>getServletContext().setAttribute("foo" , barObj);</code>	<code>application.setAttribute("foo" , barObj);</code>
请求	<code>request.setAttribute("foo" , barObj);</code>	<code>request.setAttribute("foo" , barObj);</code>
会话	<code>request.getSession().setAttribute("foo" , barObj);</code>	<code>session.setAttribute("foo" , barObj);</code>
页面	不适用！	<code>pageContext.setAttribute("foo" , barObj);</code>

哪个JSP表达式标记会打印名为“javax.sql.DataSource”的上下文参数? (35)

- A. `<%= application.getAttribute("javax.sql.DataSource") %>`
- B. `<%= application.getInitParameter("javax.sql.DataSource") %>`
- C. `<%= request.getParameter("javax.sql.DataSource") %>`
- D. `<%= contextParam.get("javax.sql.DataSource") %>`

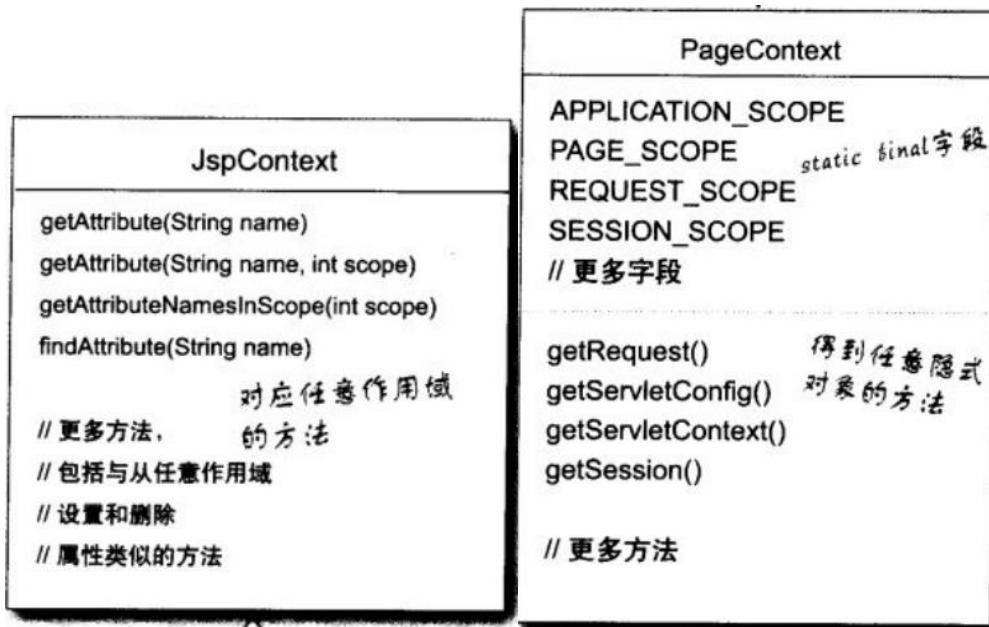
看到“上下文”时，我们要注意：属性时，servlet和JSP的命名存在着不一致，在servlet中是：

`getServletContext().getAttribute("foo")`

而在JSP中则是：

`application.getAttribute("foo")`

15. PageContext



获得一个页面作用域属性

```
<%= pageContext.getAttribute("foo") %>
```

使用pageContext 设置一个会话作用域属性

```
<% Float two = new Float(22.4); %>
<% pageContext.setAttribute("foo", two, PageContext.SESSION_SCOPE); %>
```

使用pageContext 获得一个会话作用域属性

```
<%= pageContext.getAttribute("foo", PageContext.SESSION_SCOPE) %>
(这等价于: <%= session.getAttribute("foo") %> )
```

使用pageContext 获得一个应用作用域属性

```
Email is:
<%= pageContext.getAttribute("mail", PageContext.APPLICATION_SCOPE) %>
```

参数版本就像所有其他相应方法一样，用于得到 pageContext 对象的属性。数版本可以得到4个作用域一个作用域的属性。

findAttribute()方法在哪里查找属性？它首先在页面上下文中查找，所以如果页面上下文作用域有一个“foo”属性，那么在PageContext上调用findAttribute(String name)就和在PageContext上调用getAttribute(String name)的作用相同。但是如果页面作用域没有这样一个“foo”属性，这个方法就会开始在其他作用域查找，先从最严格的作用域查起，逐步转向不那么严格的作用域；换句话说，先在请求作用域查找，再查找会话作用域，最后查找应用作用域。最先找到的那个就算“赢”，如果在一个作用域中找到指定名字的属性，就不会再在其他作用域中查找。

16. 三个常用的指令：page, taglib, include 指令

① page指令

```
<%@ page import="foo.*" session="false" %>
```

定义页面特定的属性，如字符编码、页面响应的内容类型，以及这个页面是否要有隐式的会话对象。`page`指令可以使用至多13个不同的属性（如`import`属性），不过考试中只会考其中的4个属性。

② taglib指令

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

定义JSP可以使用的标记库。我们还没有讲到如何使用定制标记和标准动作，所以现在谈这个指令不太合适。先了解一下就行了……后面很快就会用两章专门介绍标记库。

③ include指令

```
<%@ include file="wickedHeader.html" %>
```

定义在转换时增加到当前页面的文本和代码。这样你就能建立可重用的块（如标准页面标题或导航栏），这些可重用的块能增加到各个页面上，而不必在每个JSP中把所有这些代码再重复一遍。

17. 指令字典

考试时可能会考

<code>import</code>	定义Java <code>import</code> 语句，所定义的 <code>import</code> 语句会增加到生成的 <code>servlet</code> 类中。生成的 <code>servlet</code> 类中会自动（默认地）加上以下 <code>import</code> 语句： <code>java.lang</code> （当然了）、 <code>javax.servlet</code> 、 <code>javax.servlet.http</code> 和 <code>javax.servlet.jsp</code> 。
<code>isThreadSafe</code>	定义生成的 <code>servlet</code> 是否需要实现 <code>SingleThreadModel</code> ，不过你知道，这是一个很糟糕的想法。默认值是……“ <code>true</code> ”，这表示“我的应用是线程安全的，所以我不需要实现 <code>SingleThreadModel</code> ，因为我知道实现 <code>SingleThreadModel</code> 从本质上讲是很不好的”。只有需要把这个属性值设置为 <code>false</code> 时才有必要指定这个属性，这说明你希望生成的 <code>servlet</code> 使用 <code>SingleThreadModel</code> ，不过绝对不要这么做。
<code>contentType</code>	定义JSP响应的MIME类型（和可选的字符编码）。你应该知道默认值是什么。
<code>isELIgnored</code>	定义转换这个页面时是否忽略EL表达式。我们还没有谈到EL；这是下一章的内容。就目前而言，只要知道可以在页面中选择忽略EL语法就行了，告诉容器忽略EL语法有两种办法，其中一种办法就是设置这个属性。
<code>isErrorPage</code>	定义当前页面是否是另一个JSP的错误页面。默认值是“ <code>false</code> ”，但是如果这个属性值为 <code>true</code> ，页面就能访问隐式的 <code>exception</code> 对象（这是对令人讨厌的 <code>Throwable</code> 的一个引用）。如果这个属性值为 <code>false</code> ，这个JSP就不能使用隐式的 <code>exception</code> 对象。
<code>errorPage</code>	定义一个资源的URL，如果有未捕获到的 <code>Throwable</code> ，就会发送到这个资源。如果这里指定了一个JSP，该JSP的 <code>page</code> 指令中就会有 <code>isErrorPage=“true”</code> 属性。

language	定义scriptlet、表达式和声明中使用的脚本语言。现在，可取值只有一个，就是“java”，但是还是留有这个属性是为了未雨绸缪，就像那些规范开发者一样，考虑到了将来可能会使用其他的语言。
extends	JSP会变成一个servlet类，这个属性则定义了此类会以哪个类作为超类。一般不会使用这个属性，除非你真的很清楚自己在做什么——它会覆盖容器提供的类层次体系。
session	定义页面是否有一个隐式的session对象。默认值为“true”。
buffer	定义隐式out对象(JspWriter的引用)如何处理缓存。
autoFlush	定义缓存的输出是否自动刷新输出。默认值是“true”。
info	定义放到转换后页面中的串，这样就能使用所生成servlet继承的getServletInfo()方法来得到这个信息。
pageEncoding	定义JSP的字符编码。默认为“ISO-8859-1”(除非contentType属性已经定义了一个字符编码，或者页面使用XML文档语法)。

18. EL 表达式

EL表达式的形式是：

`${something}`

换句话说：表达式总是括在大括号中，而且前面有一个美元符(\$)作前缀

19. 在 DD 里禁用 JSP 脚本

以下EL表达式：

`Please contact: ${applicationScope.mail}`

下面这个Java表达式是等价的：

`Please contact: <%= application.getAttribute("mail")%>`

```
<web-app ...>
  ...
  <jsp-config>
    <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <scripting-invalid>
        true
      </scripting-invalid>
    </jsp-property-group>
  </jsp-config>
  ...
</web-app>
```

20. 关于开启 EL

在DD中放置<el-ignored>

```
<web-app ...>
...
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>
      true
    </el-ignored>
  </jsp-property-group>
</jsp-config>
...
</web-app>
```

如果DD中的<el-ignored>设置与isELIgnored page指令属性有冲突，总是听指令的！所以可以在DD中指定默认行为，而对于特定页面使用page指令来覆盖。

使用isELIgnored page指令属性

```
<%@ page isELIgnored="true" %>
page指令属性以“is”开头，但
DD标记不是！
```

DD标记是<el-ignored>，有人可能认为相应的page指令属性可能是……果有人想当然地得到这个结论就错了。不要被<is-el-igno...

当心命名上的不一致！

DD配置 <el-ignored>	page指令 isELIgnored	计算	忽略
未指定	未指定	✓	
false	未指定	✓	
true	未指定		✓
false	false	✓	
false	true		✓
true	false	✓	

21. JSP 动作

标准动作：

```
<jsp:include page="wickedFooter.js
```

其他动作：

```
<c:set var="rate" value="32" />
```

不过这有些误导，因为有些动作并不认为是标准动作，但它们仍是标准库的一部分。换句话说，后面你会了解到，有些非标准（这指的是定制）动作是……标准的（位于标准库中），但仍不认为这些动作是“标准动作”。对，可以这么说，它们是标准化的非标准定制动作。你搞清楚了吗？

第八章 无脚本的 JSP

1. JSP 标准动作

如果用脚本

```
<%= ((foo.Person) request.getAttribute("person")).getName() %>
```

使用标准动作（没有脚本）

这里没“只有两个标准动作标记”

```
<html><body>  
<jsp:useBean id="person" class="foo.Person" scope="request" />  
Person created by servlet: <jsp:getProperty name="person" property="name" />  
</body></html>
```

2. JSP 标准动作解析：如果没有变量，useBean 会创建一个 bean；bean 没有带参数的构造函数，一定要用完全限定名

用 `<jsp:useBean>` 声明和初始化一个 bean 属性

```
<jsp:useBean id="person" class="foo.Person" scope="request" />
```

标识标准动作
声明 bean 对象的标识符。这对应于以下 servlet 代码中所用的名：
request.setAttribute("person", p);
声明 bean 对象的类类型 (当然是完全限定名)。
标识这个 bean 对象的作用域。

会变成 `_jspService()` 方法中的以下代码

```
foo.Person person = null; // 根据 id 的值声明一个变量，从而允许 JSP 中的其他部分（包括其他 bean 标记）引用这个变量。  
synchronized (request) { // 尝试在指定作用域（标记中定义的作用域）得到属性，并把它赋给 id 变量。  
    person = (foo.Person) _jspx_page_context.getAttribute("person", PageContext.REQUEST_SCOPE);  
    if (person == null) { // 不过，如果这个作用域中没有这样一个属性……  
        person = new foo.Person(); // 建立一个，并把它赋给 id 变量。  
        _jspx_page_context.setAttribute("person", person, PageContext.REQUEST_SCOPE);  
    } // 最后，把这个新对象设置为所定义作用域中的一个属性。  
}
```

用 `<jsp:getProperty>` 得到 bean 属性的性质值

```
<jsp:getProperty name="person" property="name" />
```

标识标准动作

标识具体的 bean 对象。
这与 `<jsp:useBean>` 标记的 "id" 值匹配。

标识属性中的标识符（换句话说，与 bean 类中获取方法和设置方法对应的性质）。

```
<jsp:useBean id="person" class="foo.Person" scope="request" />
<jsp:setProperty name="person" property="name" value="Fred" />
```

3. 条件设置：只有在找不到这个 bean 的时候才会运行体中的标签。

```
<jsp:useBean id="person" class="foo.Person" scope="page" > 这是标记的体。
```

```
    <jsp:setProperty name="person" property="name" value="Fred" />
```

```
</jsp:useBean >
```

↑ 最后结束这个标记。开始和结束
标记之间的所有内容都是体。

← <jsp:useBean>体中的代码会有条件地运行。
只有找不到bean而且创建了一个新bean时才会
运行。

4. 创建子类：使用 type。找不到对象一定报错。Class 一定是 type 的子类或者一样

增加了type属性的新JSP

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee" scope="page" />
```

如果使用了type，但没有class，bean必须已经存在。

如果使用了class（有或没有type），class不能是抽象类，而且必须有一个无参数的公共构造函数。

5. JavaBean 法则：一定要有一个无参构造器

1) 必须有一个无参数的公共构造函数。

2) 必须按命名约定来命名公共的获取方法和设置方法，首先是“get”（或者如果是一个布尔性质，获取方法的前缀则是“is”）和“set”，后面是同一个词（getFoo()、setFoo()）。要得到性质名，先去掉“get”和“set”，并把余下部分的第一个字母变成小写。

3) 设置方法的参数类型和获取方法的返回类型必须一样。这定义了性质的类型。

```
int getFoo() void setFoo(int foo)
```

4) 性质名和类型是由获取方法和设置方法推导得出，而不是得自于类中的一个成员。例如，如果你有一个私有的 int foo 变量，这并不意味着存在这样一个性质。变量可以取你喜欢的任何名字。“foo”这个性质名是从存取方法得来的。换句话说，你有一个性质只是因为你有一对获取方法和设置方法。具体如何实现由你决定。

5) 结合JSP使用时，性质类型必须是String，或者是一个基本类型。如果不是这样，尽管也许是一个合法的bean，可是如此一来，你可能还得使用脚本，而不能完全摆脱脚本只使用标准动作。

一定要记住：

`type == 引用类型`

`class == 对象类型`

或者换种说法：

`type` 是你声明的类型（可以是抽象类）

`class` 是你要实例化的类（必须是具体类）

`type x = new class()`

6. scope 属性默认为 page

7. 表单直接投到 jsp (用 param 取代 value, 或者可以不用写, 如果指定了 property, 但是没有指定 value 或者 param, 就意味着告诉 servlet 从请求参数中取值)

利用 param 属性, 可以把bean的性质值设置为一个请求参数的值。只需指定请求参数!

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
    <jsp:setProperty name="person" property="name" param="userName" />
</jsp:useBean>
```

如果请求参数名与bean性质名匹配，就不需要在`<jsp:setProperty>`标记中为该性质指定值

还可以更好：迭代处理所有请求参数，酷！

```
<form action="TestBean.jsp">
    name: <input type="text" name="name">
    ID#: <input type="text" name="empID">
    <input type="submit">
</form>
</body></html>
```

```
foo.Emp
int getEmpID()
void setEmpID()
```

可以这样做：

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
    <jsp:setProperty name="person" property="*" />
</jsp:useBean>
```

这多酷呀！！

8. bean 标记会自动转换基本类型（使用脚本时不会自动转换）

如果你以前已经熟悉 JavaBean，就不会感到奇怪。Java-Bean 的性质可以是任何东西，但如果是 String 或基本类型，就能自动完成类型强制转换。

这样很好，你不用自己来完成解析和转换。

如果使用<jsp:setProperty>标准动作标记，其property设置为通配符；或者只有性质名而没有value或param属性（这说明，该性质名与请求参数名匹配）；或者使用一个param属性来指示请求参数，要把这个请求参数值赋给bean的性质；再或者键入一个直接量值，在这些情况下，就会自动从String转换为int。这些例子都会自动地完成转换：

```
<jsp:setProperty name="person" property="*" />
<jsp:setProperty name="person" property="empID" />
<jsp:setProperty name="person" property="empID" value="343" />
<jsp:setProperty name="person" property="empID" param="empID" />
```

但是……如果你使用脚本，就不会自动转换：

```
<jsp:setProperty name="person" property="empID" value="<%= request.getParameter("empID") %>" />
```

9. EL 表达式：打印嵌套性质

10. 隐性对象性质（用.只能访问遵循 java 命名规范的性质，用【】更加灵活）

第一个变量可以是一个隐式对象，也可以是一个属性，点号右边可以是一个映射键（如果第一个变量是映射），也可以是一个bean性质（如果第一个变量是JavaBean属性）。



```
<html><body>
```

EL表达式总是放在大括号里，而且前面有一个美元符前缀

```
Dog's name is: ${person.dog.name}
```

```
$ {person.name}
```

```
</body></html>
```

就这么简单！我们甚至没有声明
person是什么意思……它自己已经
知道。

表达式中第一个命名变量可
以是一个隐式对象，也
可以是一个属性

以下代码：

```
 ${person["name"]}
```

简单的点号操作符之所以能工作，是因为
person是一个bean，而且name是person的
一个性质。

与下面的代码等价：

```
 ${person.name}
```

但是如果person是一个数组呢？

或者如果person是一个List呢？

再或者，name不遵循正常的Java命名规则
怎么办？

记住，EL有一些隐式对象。但是这与JSP隐式对象不同（只有一个例外，即pageContext）。下面简单地列出了这些EL隐式对象，后面几页还会更详细地讨论。你会注意到，除了一个隐式对象外（又是pageContext），其他隐式对象都是简单的Map（名/值对）。

如果中括号里是一个String直接量（即用引号引起的串），这可以是一个Map键，或是一个bean性质，还可以是List或数组中的索引。

`${musicList["something"]}`

11. 中括号的索引会被强制转化成整数

如果不能用作为Java代码中的变量名，就不要把它放在点号后面

`request.setAttribute("Genre", "Ambient");`

在JSP中这是可以的：

`Music is ${musicMap[Genre]}` ————— 计算为 ————— `Music is ${musicMap["Ambient"]}`

由于有一个名为“Genre”的请求属性，它的值为“Ambient”，而且“Ambient”是musicMap的一个键。

在JSP中，下面这样是不行的（给定以上servlet代码）：

`Music is ${musicMap["Genre"]}` ————— 不变 ————— `Music is ${musicMap["Genre"]}`

因为musicMap中没有名为“Genre”的键。由于加了引号，容器不会进行计算，而只认为这是一个直接量键名。

↑
这是合法的EL表达式，但是它所做的与我们预想的不一样。

11.5. 在中括号中可以嵌套表达式

JSP中这样是可以的：

`Music is ${musicMap[MusicType[0]]}`

12. 调用表单：只有一个值就是 param，否则就应该是 paramValues

Request param name is: \${param.name}

Request param empID is: \${param.empID}

Request param food is: \${param.food}
 ←

First food request param: \${paramValues.food[0]}

Second food request param: \${paramValues.food[1]}

13. 调用 header 的 host

但是利用EL，就只能用header隐式对象：

```
Host is: ${header["host"]}
```

```
Host is: ${header.host}
```

14. EL 没有 request 隐式对象，可以使用 pageContext。EL 没有 request 对象

隐式的requestScope只是请求作用域属性的一个Map，而不是request对象本身！你想要的（HTTP方法）是request对象的一个性质，而不是请求作用域上的一个属性。换句话说，你希望在request对象上调用一个获取方法来得到你想要的东西（如果把request对象看作是一个bean）。

如果这样想就很简单了，例如，applicationScope是ServletContext的一个引用，因为应用作用域属性就绑定到这个对象。但是，就像requestScope和request对象一样，应用作用域属性的Map只是属性的一个映射（Map），仅此而已。不能把它当成一个servlet上下文，所以不要指望能从applicationScope隐式对象得到ServletContext的性质！

使用pageContext来得到其他的一切……

```
Method is: ${pageContext.request.method}
```

15. 对于类似“foo.person”的变量，，使用下面的方法 access

或者，如果你担心可能存在命名冲突，可以明确地指出想要哪个作用域里的person：

```
${requestScope.person.name}
```

太完美了！使用request-Scope对象，我们就有办法把属性名放在引号里了。
→ \${requestScope["foo.person"].name}

16. 获得 cookie 的值

但是使用EL的话，可以用cookie隐式对象：

```
${cookie.userName.value}
```

容易多了。只需提供名字，就会从cookie 名/值对的Map返回适当的值。

17. 打印上下文初始参数（不是 servlet 参数）

```
<context-param>
  <param-name>mainEmail</param-name>
  <param-value>likewecare@wickedlysmart.com</param-value>
</context-param>
```

不同。

使用EL的话，就容易多了：

```
email is: ${initParam.mainEmail}
```

EL initParam 并不对应使用 <init-param> 配置的参数!

这里有一点很容易混淆：servlet初始化参数是用<init-param>配置的，而上下文参数使用<context-param>配置；但是EL隐式对象“init-Param”对应的却是上下文参数！要是制订规范

18. 编写一个掷骰子的 jsp 程序

① 编写有一个公共静态方法的Java类。

这只是一个普通的Java类。这个方法必须是公共的，而且是一个静态方法。它可以有参数，返回类型一般都不是void（但并没有严格要求）。毕竟，这个方法存在的意义就是从JSP调用并得到一些东西，可以把返回的结果用作为表达式的一部分，或者是把结果打印出来。

把这个类文件放在/WEB-INF/classes目录结构中（对应适当的包目录结构，这与其他类是一样的）。

② 编写一个标记库描述文件（Tag Library Descriptor, TLD）。

对于一个EL函数，TLD提供了定义函数的Java类与调用函数的JSP之间的一个映射。这样一来，函数名和具体的方法名可以是不同的。例如，你使用的类可以有一个着实很傻气的方法名，但你想为使用EL的页面设计人员提供一个更明显更直观的名字。没问题，TLD会这样说：“这是一个Java类，这是函数的方法签名（包括返回类型），这是我们在EL表达式里用的名字。”换句话说，EL中使用的名字不必与具体的方法名相同。这个映射就在TLD里定义。

把TLD文件放在/WEB-INF目录中，指定扩展名为.tld（TLD还可以放在其他位置，后面有两章将讨论这个问题）。

③ 在JSP中放一个taglib指令。

taglib指令告诉容器：“我想使用这个TLD，另外，在JSP中使用这个TLD中的一个函数时，我想用这个名字作为前缀……”换句话说，这样就能定义命名空间。你可以使用多个TLD中的函数，就算是函数同名也不碍事。taglib指令就好像给你的函数提供了一个完全限定名。调用函数时要同时指定函数名和TLD前缀。前缀可以是你喜欢的任何名字。

④ 使用EL调用函数。

这一部分很容易。只需要按\${prefix:name()}的形式从表达式调用函数就行了。

19. Tld 文件（tld 和 jsp 的 uri 必须一致）

```

<tlib-version>1.2</tlib-version>
<uri>DiceFunctions</uri>
<function>
    <name>rollIt</name>
    <function-class>foo.DiceRoller</function-class>
    <function-signature>
        int rollDice()
    </function-signature>
</function>

```

The JSP

```

<%@ taglib prefix="mine" uri="DiceFunctions"%>
<html><body>
    ${mine:rollIt()}
</body></html>

```

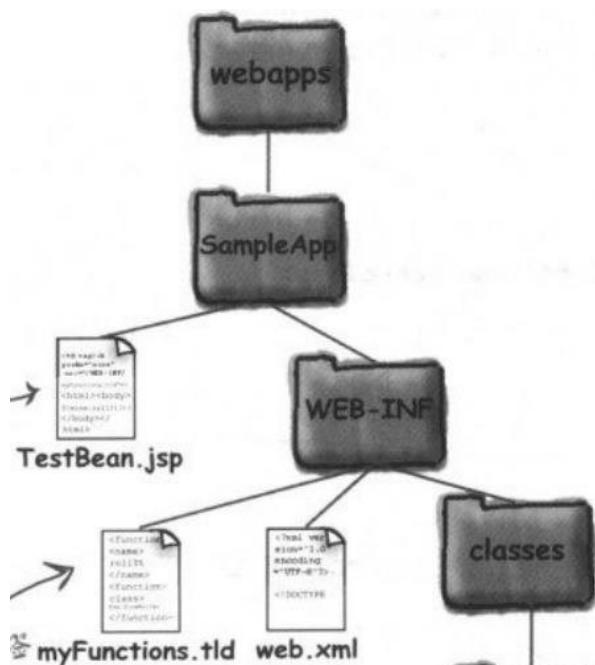
函数名 rollIt() 来自 TLD 中的 <name>, 而不是来自具体的 Java 类。

20. 结构层次

包含函数（公共的静态方法）的类应当像 servlet、bean 和监听者类一样，必须对 Web 应用可用。这说明，要放在 WEB-INF/classes 中的某个位置……

TLD 文件可以放在 WEB-INF 下的某个位置上，并确保 JSP 中 taglib 指令包含的 uri 属性与 TLD 中的 <uri> 元素匹配。

22. EL 表达式的逻辑：一个不完整的页面好于一个错误页面（只在除零的时候抛出异常）



true	这是一个布尔直接量
false	这是另外一个布尔直接量
null	它就表示…… null
instanceof	(这是为将来预留的保留字)
empty	这个操作符可以查看是否为 null 或空， 例如，当 A 为 null 或空时，\${empty A} 就返回 true (本章稍后会介绍一个具 体的示例)。

22.5 EL 函数传参的方法：一定要使用完全限定名

D: 当然可以有。只要记住一点，在 TLD 中要为每个参数指定完全限定类名（除非这是一个基本类型）。例如，如果函数取一个 Map 参数，则应当是：

```
<function-signature>
    int rollDice(java.util.Map)
</function-signature>
```

调用时则要：

```
${mine:rollDice(aMapAttribute)}
```

EL能很好地处理null。它能处理unknown或null值，即使找不到表达式中指定名的属性/性质/键，也会显示页面。

在算术表达式中，EL把null值看作是“0”。

在逻辑表达式中，EL把null值看作是“false”。

```
<%@ include file="Header.jsp"%>
<jsp:include page="Header.jsp" />
```

23. 可重用的组件：include 指令和 include 标准动作（某些容器，例如 tomcat 等重新包含）

```
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response,
    "Header.jsp", out, false);
```

include指令在转换时发生

<jsp:include>标准动作在运行时发生

如果使用include指令，这与你打开JSP页面，并粘贴上“Header.jsp”的内容没有两样。换句话说，这样就好像你把页眉文件的代码重复放在其他JSP中一样。只不过，容器会在转换时为你做这个工作，所以你不必亲自到处复制代码。你的任务只是编写页面，其中包括一个include指令，其他的任务则由容器负责，它要在转换之前把页眉代码复制到每个JSP中，并编译为生成的servlet。

<jsp:include>则完全不同。它不是从“Header.jsp”复制源代码，include标准动作会在运行时插入“Header.jsp”的响应。<jsp:include>的关键是，容器要根据页面（page）属性创建一个RequestDispatcher，并应用 include()方法。所分派/包含的JSP针对同样的请求和响应对象在同一个线程中执行。

Web应用中的一个JSP (“Contact.jsp”)

```
<html><body>
<jsp:include page="Header.jsp" />
<br>
<em>We can help.</em> <br><br>
Contact us at: ${initParam.mainEmail}
</body></html>

<html><body>
<%@ include file="Header.jsp"%>
<br>
<em>We can help.</em> <br><br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

24. include 指令是位置敏感的
25. 被包含的页面无法修改请求。不要把开始和结束的 html 标签放进要 include 的 jsp 文件
26. 给页眉传参数（只对 jsp:include 标准动作有意义）

```

<jsp:include page="Header.jspf" >
    <jsp:param name="subTitle" value="We take the sting out of SOAP." />
</jsp:include> <jsp:include> 这里 +

 <br>
<em><strong>${param.subTitle}</strong></em> <br>

```

27. 转发 jsp：以 MVC 模型思考，不应该让 jsp 转化： jsp:forward

```

<% if (request.getParameter("userName") == null) { %>
    <jsp:forward page="HandleIt.jsp" />
<% } %>

```

↗
如果参数为 null，则把请求转发到 page 属性指定的页面（就像使用 RequestDispatcher一样）。

28. Jsp:forward 缓冲区会在转发之前清空，之前所有内容都会消失，一定要先转发再刷新
29. 不用脚本的判断

这会替代 scriptlet

```

{ <c:if test="${empty param.userName}" >
    <jsp:forward page="HandleIt.jsp" />
</c:if>

```

30. 不能使用 jsp 标准动作把二进制文件 include 进来
31. 这个语句的语句体永远不会执行，因为只给了 type，一定是这个 person 变量已经被 set 进了 scope，这样就不会执行语句体；但是如果没有这个 person，也不会执行这个语句体，而是直接抛异常

答：假设你包含了一个 JSP，其中有一个 EL 表达式调用了 rollIt 函数，这个函数会生成一个随机数。按逻辑，若使用 include 指令，EL 表达式只会简单地复制到包含 JSP。所以每次访问页面时，EL 表达式都会运行，并生成一个新的随机数。一定要牢牢记住：采用 include 指令，被包含页面的源代码将成为有 include 指令的“外围”页面的一部分。

JSP 中只有这个指令的位置会带来区别。例如，如果使用一个 page 指令，可以把它放在页面中的任何位置，不过，按惯例大多数人都会把 page 指令放在最前面。

但 include 指令不同，它要告诉容器到底把被包含文件的源代码插到哪里！例如，如果要包含一个页眉和一个页脚，可能如下所示：

```

<html><body>
<%@include file="Header.html"%> <br>
<em>We can help.</em> <br><br>
Contact us at: ${initParam.mainEmail} <br>
<%@ include file="Footer.html"%>
</body></html>

```

```

<jsp:useBean id="person" type="foo.Employee" >
→ <jsp:setProperty name="person" property="name" value="Fred" />
</jsp:useBean >

```

这个请求会失败！因为 person 虽然是 employee 类，实际上却是 person。

注意不设置 scope 的时候，默认是 page。但是 person 的作用域是 request，所以无论如何都会失败

```

foo.Person p = new foo.Employee();
p.setName("Evan");
request.setAttribute("person", p);

```

32. 获得 param 的参数

给定一个HTML表单，其中使用了复选框，以便用户为一个名为**hobbies**的参数选择多个值。

以下哪个表达式能计算得到**hobbies**参数的第一个值？（选出所有正确的答案。）

A. \${param.hobbies}

33. 注意减号的不合法使用

```

<a href='mailto:${initParam.master-email}'>
    email me</a>

```

\${header.User-Agent} 等价于 **\${header["User-Agent"]}**

34. 这个会打印 param.lastname

```

${paramValues.lastname[0]}

```

第九章 jstl 定制标签

1. foreach 动作

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<strong> Movie list:</strong>
<br><br>

<table>
    <c:forEach var="movie" items="${movieList}" >
        <tr>
            <td>${movie}</td>
        </tr>
    </c:forEach>
</table>

```

循环处理整个数组（“movieList”属性），并分别在一个新行中打印各个元素（这个表中每行只有一列）。

2. 可选的循环计数器和步长

用可选的varStatus属性得到一个循环计数器

```
<table>
    <c:forEach var="movie" items="${movieList}" varStatus="movieLoopCount" >
        <tr>
            <td>Count: ${movieLoopCount.count}</td>
        </tr>
        <tr>
            <td>${movie} <br><br></td>
        </tr>
    </c:forEach>
```

`varStatus` 建立一个新变量，其中
保存 `javax.servlet.jsp.jstl.core.LoopTagStatus` 的一个实例。

很有用，`LoopTagStatus` 类
有一个 `count` 属性，可以
提供迭代计数器的当
前值。（这与 `for` 循环中
的 “`i`” 很类似。）

Count: 1	Amelie
Count: 2	Return of the King
Count: 3	

3. 嵌套 foreach，只有 foreach 可以被用来迭代一个对象集合

```
<c:forEach var="listElement" items="${movies}" >
    <c:forEach var="movie" items="${listElement}" >
        <tr>
            <td>${movie}</td>
        </tr>
    </c:forEach>
</c:forEach>
```

4. 变步长的 forEach

你知道的不只这些吧……能不能告诉我，
有没有办法改变迭代步长？例如，在实际的 Java `for` 循
环中，我不一定非得写 `i++`，而是可以用 `i+=3`，这样
就能每隔 3 个元素处理一次，而不是每个元素都要处
理……

答：没问题。`<c:forEach>` 标记有可选的 `begin` 和
`end` 属性，因为你可能想对集合的一个子集进行迭代
处理，另外如果你想跳过某些元素，还提供了可选的
`step` 属性。

4. var 变量只在某个范围内起作用

5. if 分支

```
<c:if test="${userType eq 'member'}" >
    <jsp:include page="inputComments.jsp"/>
</c:if>
```

6. 大型分支判断 which 和 choose（与 switch 不同：不会执行多个分支）

```

<c:choose>
    <c:when test="${userPref == 'performance'}">
        Now you can stop even if you <em>do</em> drive insanely fast.
    </c:when>

    <c:when test="${userPref == 'safety'}">
        Our brakes will never lock up, no matter how bad a driver you are.
    </c:when>

    <c:when test="${userPref == 'maintenance'}">
        Lost your tech job? No problem--you won't have to service these brakes
        for at least three years.
    </c:when>

    <c:otherwise>
        Our brakes are the best. ←
    </c:otherwise>
</c:choose>

```

一个分支语句，而~只能运行其中的一个体。)

如果这些< c:when>测试都不为true，< c:otherwise>就会作为默认选择运行。

7. c:set 标签。Jsp:setProperty 只能完成一项任务：设置 bean 的性质，但是 c:set 标签可以做更多

① 没有体

```
<c:set var="userLevel" scope="session" value="Cowboy" />
```

scope 是可选的，var 是必要的。必须指定一个值，不过既可以放入一个 value 属性，也可以把值放在标记体中（见以下的第2种形式）。

↑
value 不必为String.....

```
<c:set var="Fido" value="${person.dog}" />
```

如果 \${person.dog} 算为一个 Dog 对象，那么 "Fido" 的类型是 Dog。

② 有体

```
<c:set var="userLevel" scope="session">
    Sheriff, Bartender, Cowgirl
</c:set>
```

计算体，并用作为变量的值。

记住，标记有体时，这里没有斜线。

如果 value 的值是 null，这个属性会被删除，无论这个属性是否存在

8. 用<c:set>设置 target

这一类<c:set>（包括它的两个变种：有体和没有体）只能用来设置两种东西：bean性质和Map值。仅此而已。不能用它向列表或数组中增加元素。用起来很简单，只需提供对象（bean或Map）、性质/键名以及值。

● 没有体

```
<c:set target="${PetMap}" property="dogName" value="Clover" />
```

如果目标是一个bean，就设置性质“dogName”的值。
如果目标不能为null！
如果目标是一个Map，则设置“dogName”键的相应值。

● 有体

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

不要把属性的“id”名
放在这里！
体可以是一个
String或表达式
没有斜线……考试时
一定要当心这一点。

9. target 不是 id，必须在这里放置一个对象，而不是对象名字

"target" 必须计算为一个对象!不能键入bean或Map属性的String "id" 名!

这是一个重要的技巧。在<c:set>标记中，标记中的“target”属性看上去有些像<jsp:useBean>中的“id”。尽管另一个版本的<c:set>中“var”属性取一个String直接量（表示作用域属性的名）。但是……“target”可不是这样！“target”属性中不能键入表示属性名（属性以这个名字绑定到页面等作用域）。不是这样的，“target”属性需要的值要能解析为真正的对象。这说明，它需要一个EL表达式或一个脚本表达式(<%= %>)，或者是我们还没见过的<jsp:attribute>。

10. 其他注意的地方

- ▶ <c:set>中不能同时有“var”和“target”属性
- ▶ “scope”是可选的，如果没有使用这个属性，则默认为页面作用域
- ▶ 如果“value”为null，“var”指定的属性将被删除！
- ▶ 如果“var”指定的属性不存在，则会创建一个属性，但仅当“value”不为null时才会创建新属性
- ▶ 如果“target”表达式为null，容器会抛出一个异常

如果使用了“var”版本的<c:set>，但没有指定作用域，而且容器在4个作用域中都找不到有这个名字的属性，容器就会在页面作用域建立一个新的属性。

11. 使用<c:remove>删除一个属性

*var属性必须是一个
String直接量！不
能是表达式！！*

```
userStatus: ${userStatus} <br>
<c:remove var="userStatus" scope="request" />
```

12. 使用<c:import>标签动态：可以 include 进外部网址（前两种机制不行）7

```
<c:import url="http://www.wickedlysmart.com/skyler/horse.html" />
```

动态：在请求时将URL属性值指定的内容增加到当前页面。它与<jsp:include>非常相似，但是更强大，也更灵活。

不要把<c:import>（一种包含机制）与page指令的“import”属性（将一个Java import语句放在所生成的servlet中）相混淆。

13. 使用定制标签的 param

```
<c:import url="Header.jsp">    标记有一个体……
<c:param name="subTitle" value="We take the sting out of SOAP." />
</c:import>
```

14. JSP 的特殊会话跟踪<c:url>标签（使用 param 完成编码）

```
<a href="Click here</a>
  这会在“value”相对URL的最后增加
  isessionid (如果禁用了cookie)。
<c:url value="/inputComments.jsp" var="inputURL" >
  <c:param name="firstName" value="${first}" />
  <c:param name="lastName" value="${last}" />
</c:url>
```

15. 抛出异常

指定的错误页面(“errorPage.jsp”)

```
<%@ page isErrorPage="true" %>  
<html><body>  
<strong>Bummer.</strong>  
  
</body></html>
```

向容器做出确认：“不错，这是一个正式指定的错误页面。”

抛出异常的“坏”页面(“badPage.jsp”)

```
<%@ page errorPage="errorPage.jsp" %>  
<html><body>  
About to be bad.....  
<% int x = 10/0; %>  
</body></html>
```

告诉容器：“如果出问题，就把请求转发到errorPage.jsp。”

16. 在 DD 中配置更普遍的错误页面（会被 errorPage 选项覆盖）不能在一个 error-page 标签下同时使用 code 和 type;

- 普遍的声明，可以声明任何 Throwable 的异常，必须使用完全限定名

```
<error-page>  
  <exception-type>java.lang.Throwable</exception-type>  
  <location>/errorPage.jsp</location>  
</error-page>
```

- 针对异常类型

```
<error-page>  
  <exception-type>java.lang.ArithmaticException</exception-type>  
  <location>/arithmeticError.jsp</location>  
</error-page>
```

- 针对 HTTP 错误代码

```
<error-page>  
  <error-code>404</error-code>  
  <location>/notFoundError.jsp</location>  
</error-page>
```

说明 `<location>` 必须相对于 web-app 根/上下文。这说明它必须以一个斜线开头（不论错误页面根据 `<error-code>` 声明还是根据 `<exception-type>` 来设置，都是如此）。

- 自己申明的错误码

答： 好的，下面就是一个示例：

```
response.sendError(HttpServletRequest.SC_FORBIDDEN);
```

它与下面这行代码是一样的：

```
response.sendError(403);
```

如果查找HttpServletRequest接口，你会发现为常用的HTTP错误/状态码定义了许多常量。要记住，在考试中，不需要记住这些状态码！你要知道你能生成错误码，而且生成错误码的方法是response.sendError()，另外从DD中定义的错误页面来看，或者从JSP中的任何其他错误处理来看，HTTP错误由容器生成还是由程序员生成并没有区别，这就足够了。403就是403，不论它是谁发出的错误。哦，对了，还有一个重载的两参数的sendError()版本，它要取一个int和一个String消息作为参数。

17. 获得异常信息

```
You caused a ${pageContext.exception} on the server.<br>
```

18. 使用<c:catch>捕获异常

```
About to do a risky thing: <br>
<c:catch var="myException">
    Inside the catch.....
    <% int x = 10/0; %>
</c:catch>
<c:if test="${myException != null}">
    There was an exception: ${myException.message} <br>
</c:if>
```

这个创建一个新的页面作用域属性，名为“myException”，并把异常对象赋给这个变量。

19. catch 抛出异常后直接结束代码块

20. 理解 TLD

↑标记：advice。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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"
    version="2.0">
    <tlib-version>0.9</tlib-version>          必要（这是说标记是必要的，而不是非得取这个值）。
    <short-name>RandomTags</short-name>        必要：主要由工具使用.....。
    <function>
        <name>rollIt</name>
        <function-class>foo.DiceRoller</function-class>
        <function-signature>int rollDice()</function-signature> 上一章曾用到的EL函数。
    </function>
```

这是JSP 2.0使用的XML模式的版本。不用完全记住.....把它复制到你的<taglib>元素中就行了。

```

<uri>randomThings</uri> ← taglib标签中使用的惟一名!
<tag>
    <description>random advice</description> ← 可选，但是有这个标记确实很有好处……
    <name>advice</name> ← 必要！标记中就是要用这个名字（例如：<my:advice>）。
    <tag-class>foo.AdvisorTagHandler</tag-class> ← 必要！这样容器才知道有人在
        JSP中使用这个标记时要调用什么。
    <body-content>empty</body-content> ← 必要！这说明标记的体中不能有任何内容。
    <attribute> ← 如果你的标记有属性，那么每个标记属性都
        需要有一个<attribute>元素。
        <name>user</name> ← 这说明，标记中必须放一
        <required>true</required> 个“user”属性。
        <rtpvalue>true</rtpvalue> ← 这说明“user”属性可以是一个运行
            时表达式值（也就是说，不必非得是
            String直接量）。
    </attribute>
</tag>
</taglib>

```

对应advice的TLD元素

```

<taglib...>
...
<uri>randomThings</uri>
<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>user</name>
        <required>true</required>
        <rtpvalue>true</rtpvalue>
    </attribute>
</tag>
</taglib...>

```

这正是你在前一页上看到的标记，不过这里没有加注解。

使用标记的JSP

```

<html><body>
<%@ taglib prefix="mine" uri="randomThings"%>
Advisor Page<br>
<mine:advice user="${userName}" />

```

uri与TLD中的<uri>元素匹配。

在这里使用EL是可以的，因为TLD中user属性的<rtpvalue>设置为“true”（假设“userName”属性已经存在）。

1. 如果没有 rtpvalue/值为 false，则必须使用字符串直接量
2. Tag 类。使用定制标记必须使用方法名作为 doTag

```

public class AdvisorTagHandler extends SimpleTagSupport {
    private String user;
    JSP使用TLD中声明的名字调用标记时，容器会
    调用doTag()。
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().write("Hello " + user + "<br>");
        getJspContext().getOut().write("Your advice is: " + getAdvice());
    }
    public void setUser(String user) {
        this.user=user;
    }
    容器调用这个方法来将值设置为标记属性的值。
    它使用JavaBean性质命名约定得出应该向setUser-
    er()方法发送一个"user"属性。
}

```

3. 注意 rtxprvalue

如果你看到：

```

<attribute>
    <name>rate</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
</attribute>

```

或：

```

<attribute>
    <name>rate</name>
    <required>true</required>
    <rtexprvalue>          这里没有<rtexprvalue>,
    </attribute>           则默认值为false。

```

就应该知道，这样是不行的！

```

<html><body>
    <%@ taglib prefix="my" uri="myTags"%>
    <my:handleIt rate="${currentRate}" />  不行！不能是表达式……
</body></html>          必须是一个String直接量。

```

当允许使用表达式的时候，有三种写法

EL 表达式：

```
<mine:advice user="${userName}" />
```

脚本表达式

```
<mine:advice user='<%= request.getAttribute("username")%>' />
```

必须是一个表达式，而不只是一个
scriptlet。所以这里要有“三”号，而且
最后没有分号。

<jsp:attribute>标准动作

```
<mine:advice>
    <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```

即使标明为 empty 也可以在里头加入 jsp 标准动作

4. 标记体里能放什么：scriptless 里面不能放

```
<body-content>empty</body-content>          这个标记不能有体。  
<body-content>scriptless</body-content>      这个标记不能有脚本元素 (scriptlet, 脚本表达式和声明), 但是可以是模板文本和EL, 还可以是定制和标准动作。
```

```
<body-content>tagdependent</body-content>    标记体要看作是纯文本, 所以不会计算EL, 也不会触发标记/动作。
```

```
<body-content>JSP</body-content>      能放在JSP中的东西都能放在这个标记体中。
```

5. 对于没有体的标记, 有 3 种调用方法: 空标记/开始结束没有内容或者只有 jsp:attribute 的标记
6. Taglib<taglib-uri>只是一个名, 重要的是, TLD 中的<taglib-uri>必须与 taglib 指令中的 uri 相同。如果 TLD 中没有指定<uri>, 容器会尝试将 taglib 指令中的 uri 属性作为具体的 TLD 路径。这是非常糟糕的办法, 不要这么做。
7. 容器建立映射: 自动建立, 在所有可能放 TLD 的地方都搜索一遍, 在标签库里找到一个映射

① 直接在WEB-INF目录中查找

► 确保taglib uri名是惟一的。

② 直接在WEB-INF的一个子目录中查找

► 不要使用保留前缀。

保留前缀包括:

③ 在WEB-INF/lib下的JAR文件中的
META-INF目录中查找

jsp:

jspx:

java:

jaxax:

servlet:

sun:

sunw:

④ 在WEB-INF/lib下的JAR文件中的
META-INF目录的子目录中查找

第十章 定制标记开发

1. 使用标记文件。使用属性来发送参数, 标记属性只在标签内有效

从JSP调用标记

以前的做法 (使用<jsp:param>设置一个请求参数)

```
<jsp:include page="Header.jsp">  
  <jsp:param name="subTitle" value="We take the sting out of SOAP." />  
</jsp:include>
```

现在的做法 (使用有属性的标记)

```
<myTags:Header subTitle="We take the String out of SOAP" />
```

在标记文件中使用属性

以前的做法 (使用请求参数值)

```
<em><strong>$param.subTitle</strong></em>
```

现在的做法 (使用标记文件属性)

```
<em><strong>$subTitle</strong></em> <br>
```

这些代码放在具体的标记文件中 (也就是被包含的文件)。

- ① 取一个被包含文件（如“Header.jsp”），把它重命名为带有一个.tag扩展名。

```
 <br>
```

↑ 这就是整个文件……记住，我们去掉了开始和结束的<html>及<body>标记，以免这些标记在最后的JSP中重复出现。

- ② 把标记文件(Header.tag)放在WEB-INF目录下一个名为tags的目录中。

- ③ 在JSP中放一个taglib指令（有一个tagdir属性），并调用这个标记。

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

↑ taglib指令中使用“tagdir”属性，而不是指定标记库TLD的“uri”。

2. 在标记文件.tag里面使用 attribute 指令。不加会报错

```
<%@ attribute name="subTitle" required="true" rtexprvalue="true" %>  
  
 <br>  
<em><strong>${subTitle}</strong></em> <br>  
  
<myTags:Header subTitle="We take the String out of SOAP" />
```

3. 很长的属性名。在标记文件里，要使用 tag 指令来声明标记体（默认值是 scriptless）

```
<%@ tag body-content="tagdependent" %>  
↑ 这说明body-content会处理为纯文本，也就是说EL、标记和脚本都不会计算。另外两个可取值是“empty”或“scriptless”（默认值）。  
  
 <br>  
<em><strong><font color="#${fontColor}"><jsp:doBody/></font></strong></em> <br>
```

在调用这个标签的文件里

```
<myTags:Header>  
We take the sting out of SOAP. OK, so it's not Jini,<br>  
but we'll help you get through it with the least<br>  
frustration and hair loss.  
</myTags:Header>
```

4. 容器会去哪里寻找 tag 文件

.tag 文件也是 jsp 文件，因此可以使用 request/response 隐式对象，也可以使用 EL 表达式
标记文件本身可以用脚本，调用标记文件的标签可能不能用

- ① 直接在WEB-INF/tags目录中查找
- ② 在WEB-INF/tags的子目录中查找
- ③ 在WEB-INF/lib下的JAR文件的 META-INF/tags目录中查找
- ④ 在WEB-INF/lib下的JAR文件的 META-INF/tags的子目录中查找
- ⑤ 如果标记文件部署在一个JAR中，这个标记文件必须有一个TLD。

5. 如果想在一个TLD文件中声明一个标签文件，应该这么做

标记文件的相应TLD项只描述具体标记文件的位置。

标记文件的TLD如下所示：

```
<taglib .....>
  <tlib-version>1.0</tlib-version>
  <uri>myTagLibrary</uri>
  <tag-file>
    <name>Header</name>
    <path>/META-INF/tags/Header.tag</path>
  </tag-file>
</taglib>
```

注意，声明一个<tag-file>与声明一个具体的 <tag>是完全不同的。

6. 当标记文件还是不够的时候的时候，需要标签处理类。标签处理类更加灵活，可以访问标记属性、标记体，甚至是请求参数

7. 简单标记处理器

- ① 编写一个扩展SimpleTagSupport的类

```
package foo;
import javax.servlet.jsp.tagext.SimpleTagSupport;
// 需要更多import

public class SimpleTagTest1 extends SimpleTagSupport {
    // 这里放标记处理器代码
}
```

- ② 实现doTag()方法

```
public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("This is the lamest use of a custom tag");
}
```

 doTag()方法声明了一个IOException，所以不必把print包装在一个try/catch中。

在标记文件标记的体中不能使用脚本代码！

标记文件的body-content默认认为“scriptless”，所以除非你希望是另外两个值：“empty”（标记体里什么也没有）或“tagdependent”（把标记体看作是纯文本），否则不需要声明body-content。

为标记创建一个TLD

```
<taglib .....>
<tlib-version>1.2</tlib-version>
<uri>simpleTags</uri>
<tag>
    <description>worst use of a custom tag</description>
    <name>simple1</name>
    <tag-class>foo.SimpleTagTest1</tag-class>
    <body-content>empty</body-content>
</tag>
</taglib>
```

部署标记处理器和TLD

把TLD放在WEB-INF中，并把标记处理器放在WEB-INF/classes下，这里当然要遵循包目录结构。换句话说，标记处理器类要与所有其他Web应用Java类放在同一个位置上。

编写一个使用标记的JSP

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
<myTags:simple1/>
</body></html>
```

8. 使用体的标签

在 jsp 文件里

```
<myTags:simple2>
    This is the body ← 这一次，调用有体的
</myTags:simple2>                                标记……
```

在标签处理类里

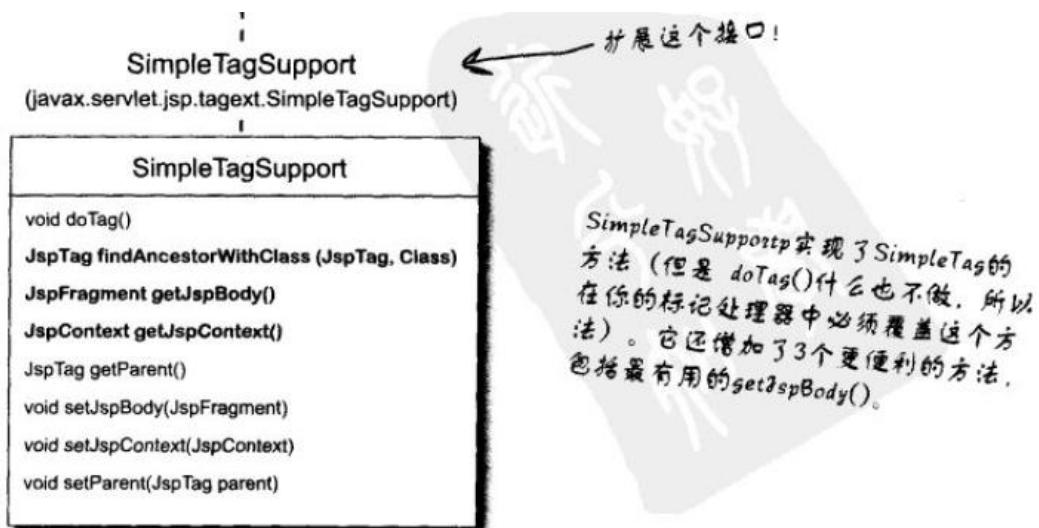
```
public void doTag() throws JspException, IOException {
    getJspBody().invoke(null); ← 这是说“处理标记的体，并把它打印到响应”。
}
```

null参数是指输出到响应，而不是输出到作为参数传入的某个书写器 (writer)。

在 tld 文件里

```
<description>marginally better use of a custom tag
<name>simple2</name>
<tag-class>foo.SimpleTagTest2</tag-class>
<body-content>scriptless</body-content>
```

9. 类层次



10. 简单标签处理器的生命周期

Web 容器加载类——实例化类(无参构造函数)——调用 `setJspContext` 方法(对 `pageContext` 的引用)——若标记是嵌套的，则调用 `setParent(JspTag)` 方法——设置属性和体(体通过 `JspFragment` 传进来，没有体的空标记不会调用这个函数)——调用 `doTag` 方法——销毁变量

11. 使用了表达式的标记的体：标记要依赖标记处理类来设置属性



12. 迭代标记体：只需要在一个循环中完成工作

JSP标记调用

```
<table>
<myTags:simple4> ${movie}</td></tr>
</myTags:simple4>
</table>
```

调用标记时 movie 属性还不存在。标记处理器会设置这个属性，并重複调用体。

标记处理器 doTag() 方法

```
String[] movies = {"Monsoon Wedding", "Saved!", "Fahrenheit 9/11"};
public void doTag() throws JspException, IOException {
    for(int i = 0; i < movies.length; i++) {
        getJspContext().setAttribute("movie", movies[i]);
        getJspBody().invoke(null);
    }
}
```

将属性值设置为数组中的下一个元素。

再次调用体。

13. 有属性的简单标记：如果需要属性，就应该在 TLD 里声明

```
<tag>
<description>takes an attribute and iterates over body</description>
<name>simple5</name>
<tag-class>foo.SimpleTagTest5</tag-class>
<body-content> scriptless </body-content>
<attribute>
    <name>movieList</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
</tag>
```

在 TLD 中使用常规的 <tag> <attribute> 声明，就像其他定制标记一样（但标记文件是例外）。

你现在的位置 ▶

在 jsp 标签调用类

```
<myTags:simple5 movieList="${movieCollection}">
<tr>
    <td>${movie.name}</td>
    <td>${movie.genre}</td>
</tr>
</myTags:simple5>
```

在标记处理器里：给属性一个 bean 式的设置方法，同时声明一个私有变量

```
private List movieList; ← 声明一个变量
public void setMovieList(List movieList) { ← 为属性编写一个 bean 式的设置方法。方法名必须与 TLD 中的属性名匹配（去掉“set”前缀，而且要把第一个字母改为小写）。
    this.movieList=movieList;
}
public void doTag() throws JspException, IOException {
    Iterator i = movieList.iterator();
    while(i.hasNext()) {
        Movie movie = (Movie) i.next();
        getJspContext().setAttribute("movie", movie);
        getJspBody().invoke(null);
    }
}
```

14. **jspFragment**: 封装 body 并发送给标签处理器。 **jspFragment** 里在得到处理的时候不能包含任何脚本（如果是 include 指令就可以）。我们无法直接获得体，只能把体输出到某处

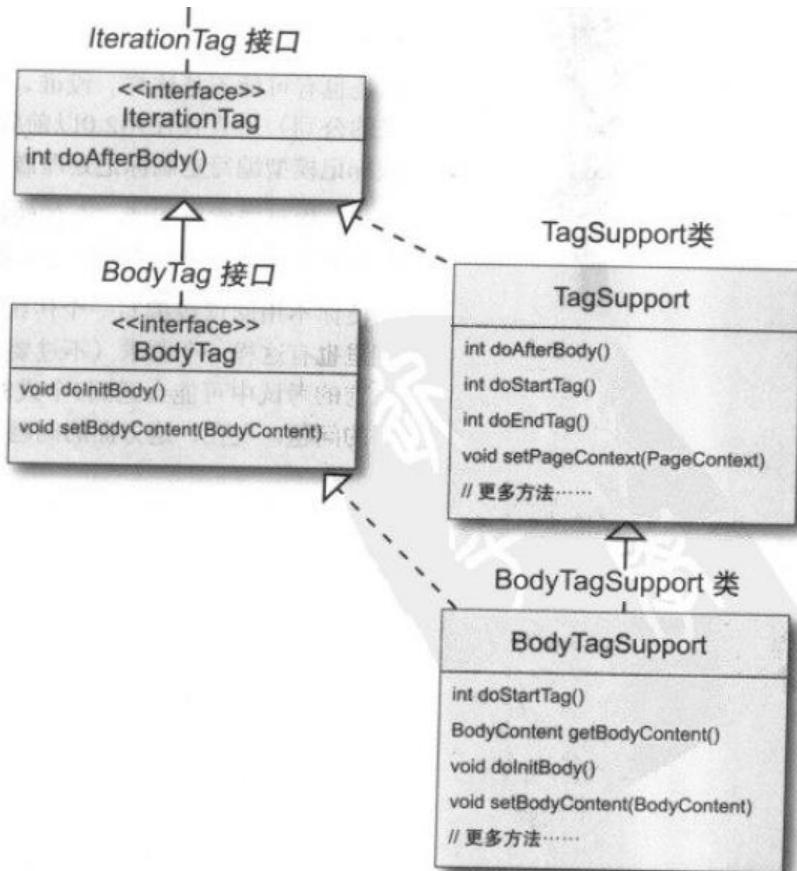
15. **skipPageException**: 保留之前的响应，跳过本页其余的所有响应，不影响调用页的响应

```

public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("Message from within doTag().<br>");
    getJspContext().getOut().print("About to throw a SkipPageException");
    if (thingsDontWork) {
        throw new SkipPageException(); ← 在这里，我们决定停止标记的余下部分和页
    }                                面的余下部分。响应中只会出现页面和标记
}                                     的前一部分（抛出异常之前的部分）

```

16. 传统标签处理器，5个类和3个API



17. 一个非常小的传统标签处理器类

jsp 标签、TLD 和原来是一样的

标签处理类

```

package foo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class Classic1 extends TagSupport { ← 通过扩展 TagSupport，我们实现了 Tag 和 Itera-
    public int doStartTag() throws JspException { ← tionTag。在此只覆盖了一个方法：doStart-
        JspWriter out = pageContext.getOut(); ← Tag()。
        try { ← 这个方法声明了 JspException，而不是
            out.println("classic tag output");
            catch(IOException ex) { ← IOException! (SimpleTag doTag() 声明
                throw new JspException("IOException- " + ex.toString()); ← 的是 IOException。)
            }
        }
        return SKIP_BODY; ← 传统标记从 TagSupport 继承了一个 pageContext 成员
    } ← 变量（而 SimpleTag 要使用 getJspContext() 方法）。
} ← 这里必须使用一个 try/catch，←
      因为我们不能声明 IOException。

```

必须返回一个 int 告诉容器接下来做什么。这一点后面还会详细介绍

18. 有两个方法的传统标记处理器：SKIP_PAGE 表示不计算体，直接跳到 doEndTag 方法；
EVAL_PAGE 表示计算其余的页面

```
public int doStartTag() throws JspException {
    out = pageContext.getOut();
    try {
        out.println("in doStartTag()");
    } catch(IOException ex) {
        throw new JspException("IOException- " + ex.toString());
    }
    return SKIP_BODY; ← 这是说：“如果有体，不要计算体。”
}
```

直接调用doEndTag()方法。

```
public int doEndTag() throws JspException {
    try {
        out.println("in doEndTag()");
    } catch(IOException ex) {
        throw new JspException("IOException- " + ex.toString());
    }
    return EVAL_PAGE; ← 这是说：“计算余下的页面。”（不同于
    SKIP_PAGE, SKIP_PAGE就相当于从Simple-
    Tag处理器抛出一个SkipPageException）
```

19. 有体的时候：简单 VS 传统

简单标记处理器

```
// package和import语句
public class SimpleTagTest extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().print("Before body.");
        getJspBody().invoke(null); ← 这会导致体计算（执行）。
        getJspContext().getOut().print("After body.");
    }
}
```

传统标记处理器

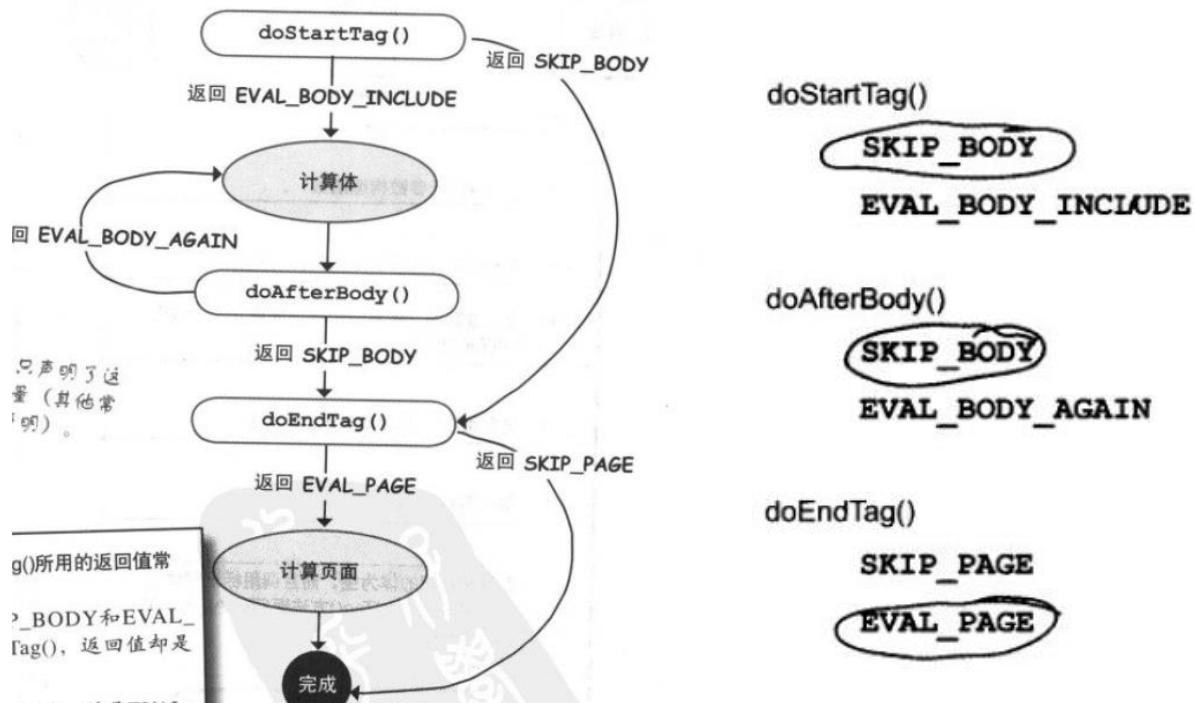
```
// package和import语句
public class ClassicTest extends TagSupport {
    JspWriter out;

    public int doStartTag() throws JspException {
        out = pageContext.getOut();
        try {
            out.println("Before body.");
        } catch(IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return EVAL_BODY_INCLUDE; ← 在传统标记处理器中这才导致计算体！
    }

    public int doEndTag() throws JspException {
        try {
            out.println("After body.");
        } catch(IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return EVAL_PAGE;
    }
}
```

20. 传统标记处理器的生命周期:

21. Web 容器加载类——实例化类（无参构造函数）——调用 setJspContext 方法（对 pageContext 的引用）——若标记是嵌套的，则调用 setParent(JspTag)方法——设置属性——调用 doStartTag 方法——如果没有声明标记的体为空，而又确实有标记，而且 doStartTag 返回 EVAL_BODY_INCLUDE，调用 doAfterBody 方法——调用 doEndTag 方法



Java 类 (doStartTag 和 doEndTag 只会调用一次), doAfterBody 只会在计算体之后调用, 所以要在 start 的时候首先过一遍; 传统标记处理器类【会被重用】, 所以应该在 doStartTag 里重新设置; 如果一个标签设置了 empty 的 body, 那么 doStartTag 必须返回 SKIP_BODY

```
public class MyIteratorTag extends TagSupport {
    String[] movies= new String[] {"Spiderman", "Saved!", "Amelie"};
    int movieCounter;

    public int doStartTag() throws JspException {
        movieCounter=0;

        return EVAL_BODY_INCLUDE;
    }
    public int doAfterBody() throws JspException {

        if (movieCounter < movies.length) {
            pageContext.setAttribute("movie", movies[movieCounter]);
            movieCounter++;
            return EVAL_BODY_AGAIN;
        } else {
            return SKIP_BODY;
        }
    }
    public int doEndTag() throws JspException {
        return EVAL_PAGE;
    }
}
```

```

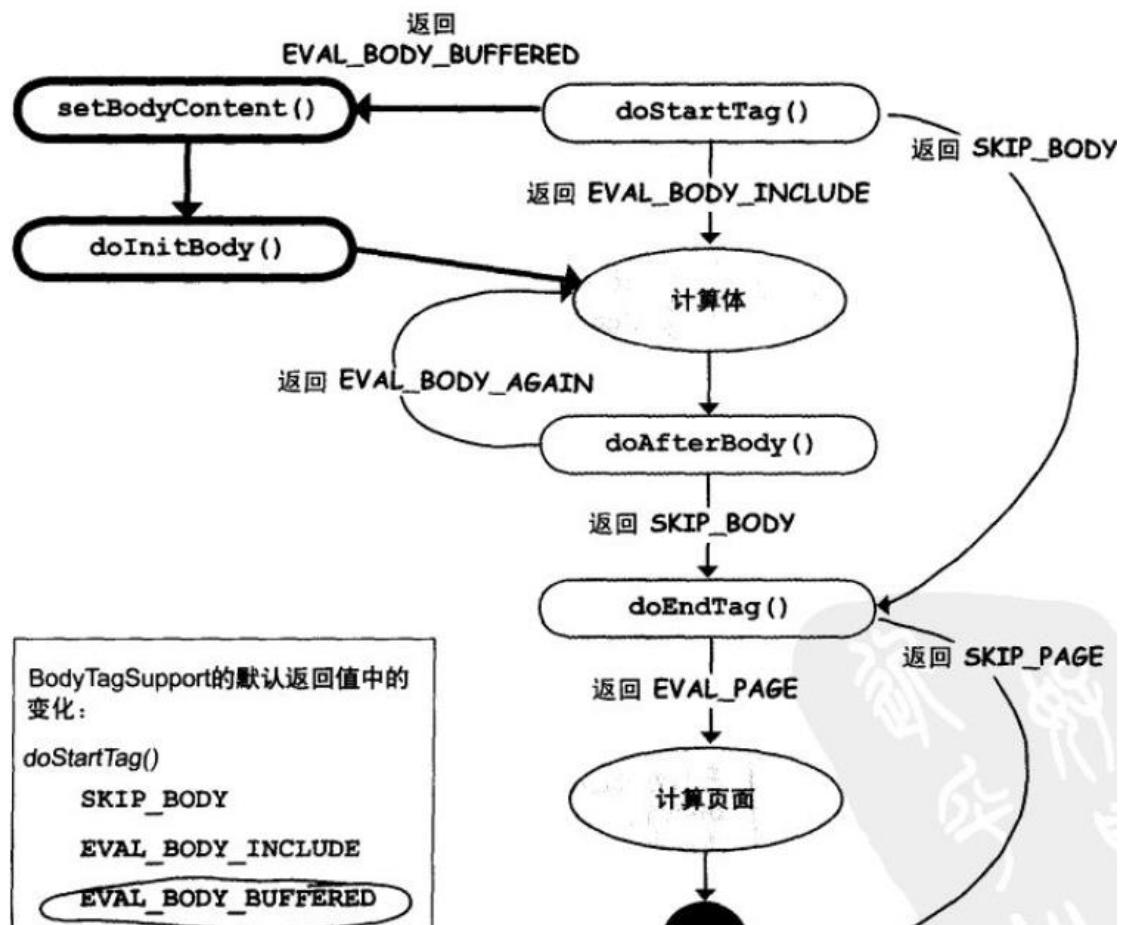
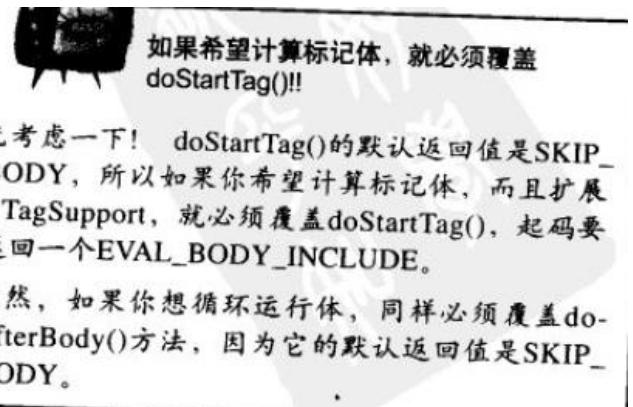
int movieCounter;

public int doStartTag() throws JspException {
    movieCounter=0;
    pageContext.setAttribute("movie", movies[movieCounter]);
    movieCounter++;
    return EVAL_BODY_INCLUDE;
}

```

必须增加这两行才能成正确的响应。

22. 如果想直接得到体：拓展 BodyTagSupport



23. 嵌套标签的协作

获得父标签引用

在传统标记处理器中得到父标记

```
public int doStartTag() throws JspException {  
    OuterTag parent = (OuterTag) getParent();  
    // 用它做些处理  
    return EVAL_BODY_INCLUDE;  
}  
不忘记类型强制  
转换!
```

在简单标记处理器中得到父标记

```
public void doTag() throws JspException, IOException {  
    OuterTag parent = (OuterTag) getParent();  
    getJspContext().getOut().print("Parent attribute is: " + parent.getName());  
}  
在父标记处理器中  
public class MyClassicParent extends TagSupport {  
    一旦有了父标记，可以像其他  
    Java对象一样对其调用方法。  
    所以可以得到父标记的属性！
```

简单标签可以访问简单/传统父标签，传统标签只能访问传统父标签

获得子标签信息：需要子标签设置属性

在子标签

子标记MenuItem：

```
public class MenuItem extends TagSupport {  
    private String itemValue;  
  
    public void setItemValue(String value) {  
        itemValue=value;  
    }  
  
    public int doStartTag() throws JspException {  
        return EVAL_BODY_INCLUDE;  
    }  
  
    public int doEndTag() throws JspException {  
        Menu parent = (Menu) getParent();  
        parent.addMenuItem(itemValue);  
        return EVAL_PAGE;  
    }  
}
```

在父标签：使用 EVAL_BODY_INCLUDE 让子标签被处理

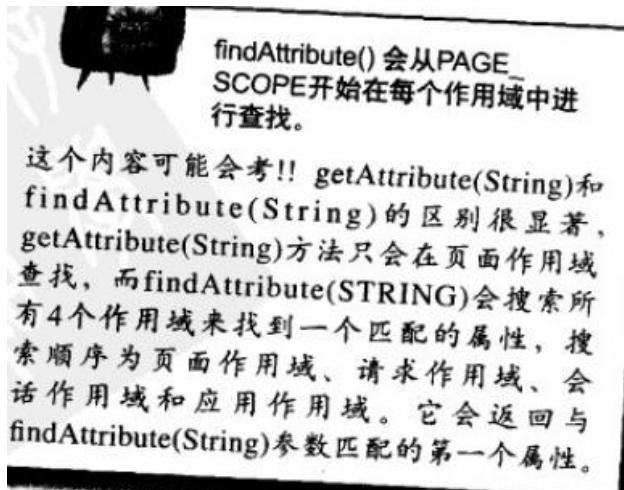
```
public void addMenuItem(String item) {  
    items.add(item);  
}  
  
public int doStartTag() throws JspException {  
    items = new ArrayList();  
    return EVAL_BODY_INCLUDE;  
}  
子标记的属性在 doStartTag() 和 doEndTag() 之间通用。  
不要忘记在 doStartTag() 中重置 ArrayList。  
因为容器可能会重用标记处理器。  
如果没有返回 EVAL_BODY_INCLUDE，  
就不会处理子标记。
```

24. 得到任意祖先：静态方法

使用findAncestorWithClass()得到任意祖先

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

25. 访问隐式对象：使用 pageContext



26. 选择题答案

A. release方法在doTag方法之后调用

-A不对，因为简单标记没有release方法。

```
<my:simpleTag>
<%@ include file="/WEB-INF/web/common/headerMenu.html" %>
</my:simpleTag>
<my:simpleTag>
```

-C是对的，因为include指令要在simpleTag的体转换为JspFragment之前得到处理；不过，所包含的内容也不能有脚本（所以这个例子包含一个HTML片段）。

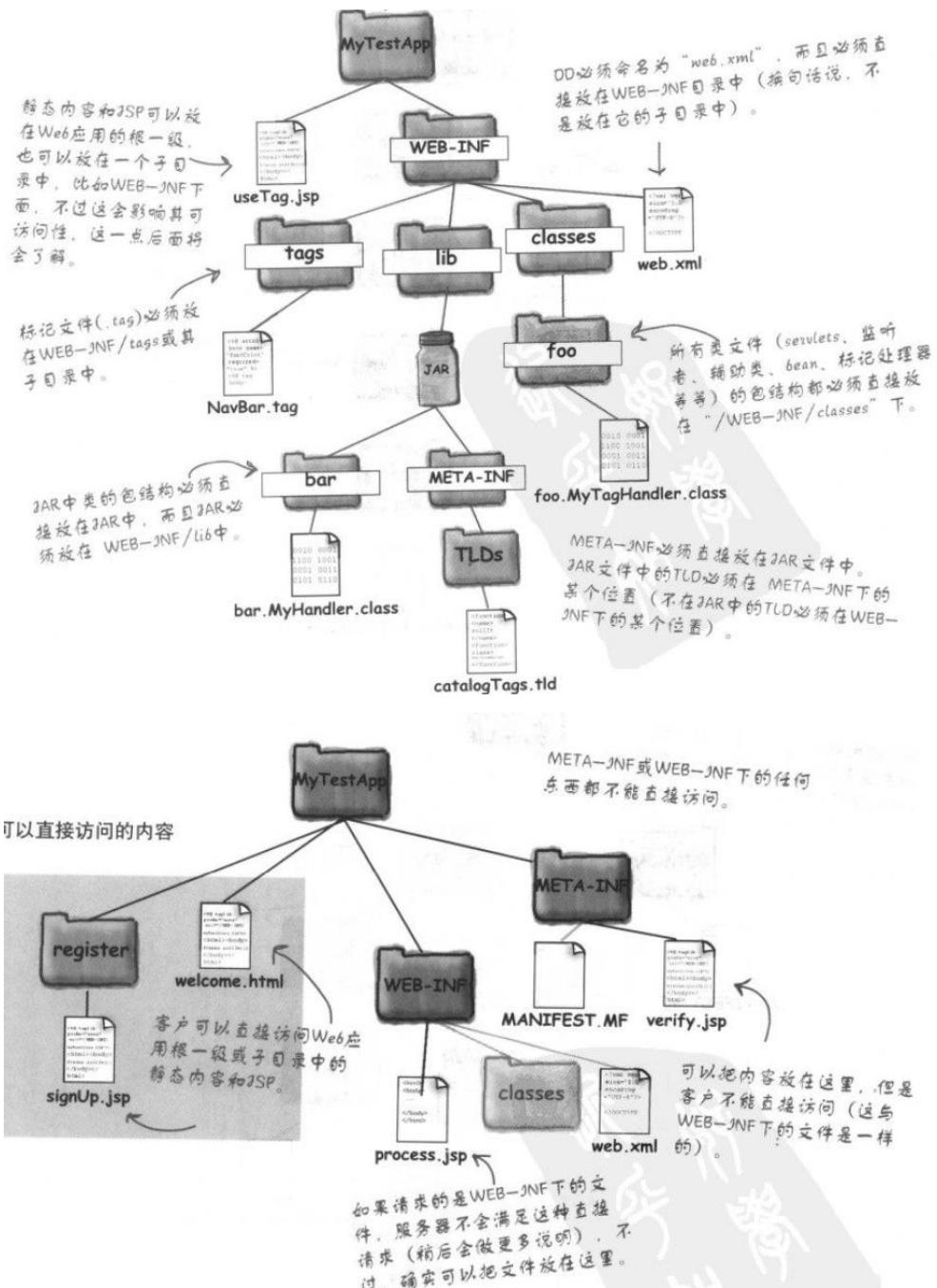
第十一章 部署 Web 应用

1. 给目录命名

要成功地部署一个Web应用，必须遵循以下目录结构。WEB-INF一定要直接放在应用上下文之下（本例中即“MyTestApp”）。classes目录必须直接放在WEB-INF目录中。classes目录中必须是该类的包结构。lib目录要直接置于WEB-INF目录下，JAR文件必须放在lib中。META-INF目录必须是JAR中的顶级目录，JAR中的TLD文件要放在META-INF目录下的某个位置（可以在任何子目录中，目录名不必是TLDs）。不在JAR中的TLD必须放在WEB-INF下的某个位置。标记文件（扩展名为.tag或.tagx的文件）必须放在WEB-INF/tags下的某个地方（除非部署在一个JAR中，如果是这样，这些TLD必须放在META-INF/tags下的某个位置）。

2. 一个正常的结构（如果使用 war 包部署，最后的目录里会多一个 META-INF 文件夹，里面的 MANIFEST.MF 指定了库依赖）。所有在 web-inf 或者 meta-inf 下的文件都不能被客户

直接访问（但是可以内部访问）



3. 如果需要访问文本文件，应该这么做

P: 这就不同了。如果你的Web应用代码需要直接访问一个资源（文本文件、JPEG等），而这个资源在一个JAR中，就要使用类加载器（classloader）的get-Resource()或getResourceAsStream()方法，这只是普通的J2SE机制，并非servlet所特有。

你可能已经知道这两个方法（getResource()和getResourceAsStream()），因为ServletContext API中也有这两个方法。区别在于，ServletContext中的方法只用于Web应用中未部署在JAR文件中的资源（考试要求你知道可以使用标准J2SE机制从JAR文件得到资源，但是无需了解任何细节）。

4. Servlet 映射

```
<servlet>
    <servlet-name>Beer</servlet-name>
    <servlet-class>com.example.BeerSelect</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Beer</servlet-name>
    <url-pattern>/Beer/SelectBeer.do</url-pattern>
</servlet-mapping>
```

这个名字主要用于DD的其他部分。
这不是客户会知道的名字。

如果有这样的请求到来，容器会在<servlet>元素中找到匹配的<servlet-name>，得出哪个类负责处理这个请求。

5. 三种 URL 映射。URL 使用最长匹配模式

三种<url-pattern>元素

① 完全匹配

```
<url-pattern>/Beer/SelectBeer.do</url-pattern>
```

↑
必须以一个斜线（/）开头 ↗
可以有扩展名，但不要带点。

② 目录匹配

```
<url-pattern>/Beer/*</url-pattern>
```

↑
必须以一个斜线开头（/）。 ↗
总是以一个斜线加星号（/*）结束。
可以是一个虚拟目录或实际目录。

③ 扩展名匹配

```
<url-pattern>*.do</url-pattern>
```

↑
必须以一个星号（*）开头（不能以句点开头）。 ↗
星号的后面必须有一个点加扩展名（.do, .as）。

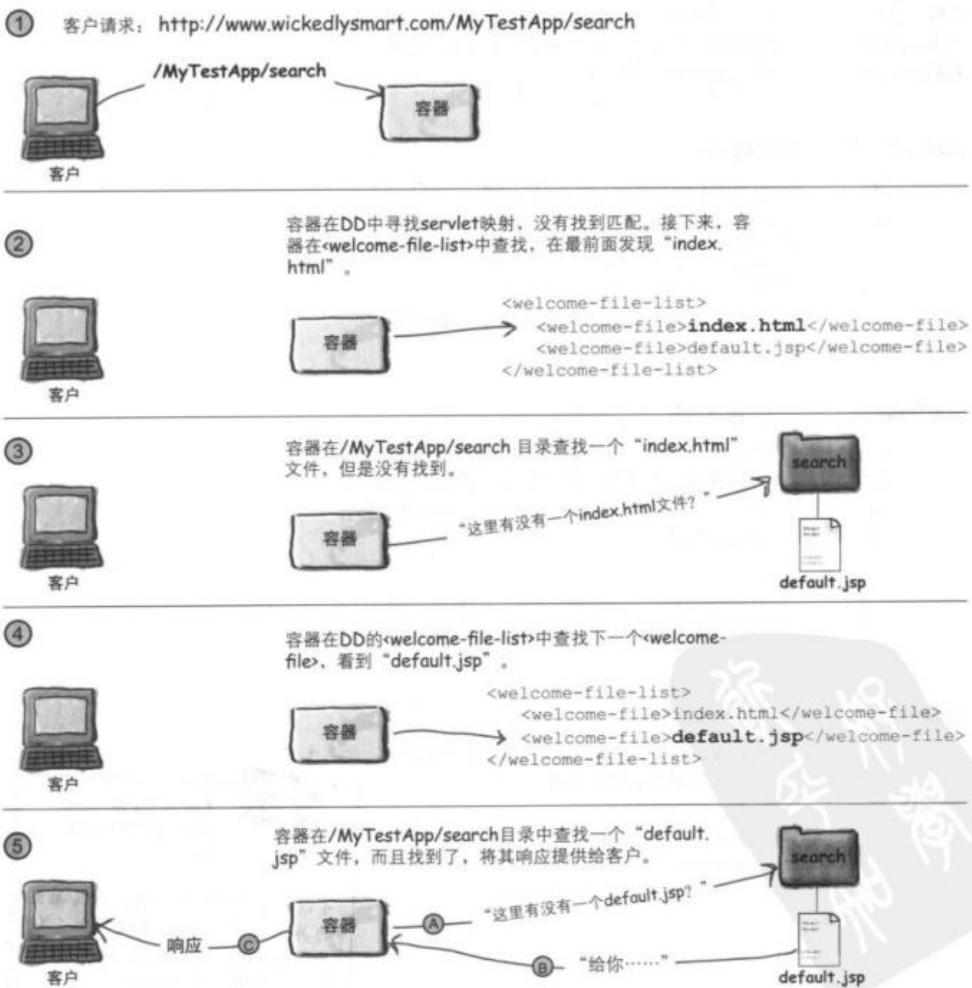
6. 配置 dd 里的欢迎文件

DD中：

```
<web-app .....>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

不能以斜线开头或
结束！

欢迎文件查找的详细步骤



7. 在 dd 中配置 servlet 初始化顺序，数字小的会被先初始化，必须大于零

DD中

```
<servlet>
  <servlet-name>KathyOne</servlet-name>
  <servlet-class>foo.DeployTestOne</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

只要是大于0，就表示“在部署时或服务器启动时初始化这个servlet，而不是等到第一个请求到来才初始化。”。

8. 与 EJB 相关的 DD 引用：什么鬼

本地bean的引用

```
<ejb-local-ref>
  <ejb-ref-name>ejb/Customer</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>com.wickedlysmart.CustomerHome</local-home>
  <local>com.wickedlysmart.Customer</local>
</ejb-local-ref>
```

代码中要用的
JNDI查找名。

这些必须是bean对外接口
的完全限制名。

远程bean的引用

```
<ejb-ref>
  <ejb-ref-name>ejb/LocalCustomer</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.wickedlysmart.CustomerHome</home>
  <remote>com.wickedlysmart.Customer</remote>
</ejb-ref>
```

(这些标记的可选子元素包括<description>和<ejb-link>, 但考试对这个内容
不做要求。)

9. DD 里 JNDI 的标记

声明应用的JNDI环境项

```
<env-entry>
  <env-entry-name>rates/discountRate</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>10</env-entry-value>
</env-entry>
```

代码中将使用的查找名。

这可以是任何类型，只要这种
类型取一个String作为构造函
数参数就行（或者，如果类型
是java.lang.Character，构造函
数则取单个的Character参数）。

这将全作为一个String传入（或者，如
果<env-entry-type>是java.lang.Char
acter，则作为单个的Character传入）。

入式的

10. DD 里的 MIME 标记：拓展名与 MIME 类型的映射，不要加拓展名

```
<mime-mapping>
  <extension>mpg</extension>
  <mime-type>video/mpeg</mime-type>
</mime-mapping>
```

不包括点“.”！

资源类型	部署位置
部署描述文件 (web.xml)	直接放在WEB-INF中（这个目录直接放在Web应用的根目录中）。
标记文件 (.tag或.tagx)	如果未部署在JAR中，标记文件必须放在WEB-INF/tags中，或WEB-INF/tags的一个子目录中。如果部署在一个JAR文件中，标记文件则必须放在META-INF/tags或META-INF/tags的一个子目录中。注意：如果标记文件部署在JAR中，那么在JAR中还必须有一个相应的TLD。
HTML和JSP (可以直接访问的 HTML和JSP)	客户可访问的HTML和JSP可以放在Web的根目录下，或者它的任何子目录下，但是不能放在WEB-INF（包括其子目录）中。如果在WAR文件中，这些页面不能放在META-INF（包括其子目录）中。
HTML和JSP (你想隐藏的HTML和 JSP，不允许直接访问)	客户不能直接访问WEB-INF（以及WAR文件中的META-INF）下的页面。
TLD (.tld)	如果不在JAR中，TLD文件必须放在WEB-INF下，或者放在WEB-INF的一个子目录下。如果部署在一个JAR中，TLD文件必须放在META-INF下，或者META-INF的一个子目录下。
Servlet类	Servlet类必须放在与包结构匹配的一个目录结构里，置于WEB-INF/classes下的一个目录中（例如，类com.example.Ring要放在WEB-INF/classes/com/example中），或者放在WEB-INF/lib下一个JAR文件里的适当包结构中。
标记处理器类	实际上，Web应用所用的所有类（类路径上类库的一部分除外）都必须像Servlet类一样遵循同样的规则，要放在WEB-INF/classes下，而且要有与包结构匹配的目录结构（或者放在WEB-INF/lib下一个JAR文件里的适当包结构中）。
JAR文件	JAR文件必须放在WEB-INF/lib目录中。

第十二章 要保密，要安全

1. Tomcat 授权：在 tomcat 的 conf/ 文件夹下

tomcat-users.xml文件

```
<tomcat-users>
  <role rolename="Guest"/>
  <role rolename="Member"/>
  <user username="Bill" password="coder" roles="Member, Guest" />
  .....
</tomcat-users>
```

对认证的控制就放在这样一种数据结构中。在Tomcat中，可以使用一个名为“tomcat-users.xml”的XML文件，其中包括一些用户名-口令-角色设置，容器在认证时就会使用这些设置。

你的应用服务器的具体做法可能和这里不一样……不过总会以某种方式把用户映射到口令和角色。

记住！这不是DD的一部分：这特定于具体的开发商。

2. 启动认证

4个<login-config>示例：

```
<web-app...>
...
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

```
</web-app>
```

—或—

```
<web-app...>
...
```

```
<login-config>
    <auth-method>DIGEST</auth-method>
</login-config>
```

```
</web-app>
```

—或—

```
<web-app...>
...
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

```
</web-app>
```

—或—

```
<web-app...>
...
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/loginPage.html</form-login-page>
        <form-error-page>/loginError.html</form-error-page>
    </form-login-config>
</login-config>
</web-app>
```

3. 在 servlet 里的 DD 配置

servlet规范：

web.xml中的DD <security-role>元素

```
<security-role>
    <role-name>Admin</role-name>
</security-role>
<role-name>Member</role-name>
<security-role>
    <role-name>Guest</role-name>
</security-role>

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

DD中的<security-constraint>元素：

```
<web-app...>
...
<security-constraint>
    <web-resource-collection>
        <web-resource-name>UpdateRecipes</web-resource-name>
        <url-pattern>/Beer/AddRecipe/*</url-pattern> ← <url-pattern>元素定义了受约束的资源。
        <url-pattern>/Beer/ReviewRecipe/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method> ← <http-method>元素描述了对于URL模式指定的资源哪些HTTP方法是受约束的（受限的）。
    </web-resource-collection>

    <auth-constraint>
        <role-name>Admin</role-name>
        <role-name>Member</role-name>
    </auth-constraint>
</security-constraint>
</web-app>
```

4. 关于<web-resource-collection>

—<web-resource-collection>的要点—

- ▶ <web-resource-collection>元素有两个主要的子元素：<url-pattern>（一个或多个）和<http-method>（可选，0个或多个）。
- ▶ URL模式和HTTP方法一同定义受限资源请求。
- ▶ <web-resource-name>元素是必要的，就算你自己可能不会用它（可以认为它要由IDE使用，或留待将来使用）。
- ▶ <description>元素是可选的。
- ▶ <url-pattern>元素使用servlet标准命名和映射规则（有关URL模式的详细内容可以返回去看关于“部署”那一章）。
- ▶ 必须至少指定一个<url-pattern>，不过也可以有多个<url-pattern>。

- ▶ <http-method>元素的合法方法包括：GET、POST、PUT、TRACE、DELETE、HEAD和OPTIONS。
- ▶ 如果没有指定任何HTTP方法，那么所有方法都是受约束的！！
- ▶ 如果确实指定了<http-method>，则只有所指定的方法是受约束的。换言之，一旦指定了一个<http-method>，就会自动使未指定的HTTP方法不受约束。
- ▶ 一个<security-constraint>中可以有多个<web-resource-collection>元素。
- ▶ <auth-constraint>元素应用于<security-constraint>中的所有<web-resource-collection>元素。

5. 如果制定<http-method>元素，未指定的所有 HTTP 都是不受约束的。如果不指定任何<http-method>元素，则所有的 HTTP 都是不受约束的。当然必须是 doGet 和 doPost 等 doxxx 方法被覆盖之后才有效

6. <auth-constraint>标签

<role-name>规则

- ▶ 在<auth-constraint>元素中，<role-name>元素是可选的。
- ▶ 如果存在<role-name>元素，它们会告诉容器哪些角色得到许可。
- ▶ 如果存在一个<auth-constraint>元素，但是没有任何<role-name>元素，那么所有用户都遭拒绝。
- ▶ 如果有<role-name>*</role-name>，那么所有用户都是允许的。
- ▶ 角色名区分大小写。

<auth-constraint>规则

- ▶ 在<security-constraint>元素中，<auth-constraint>元素是可选的。
- ▶ 如果存在一个<auth-constraint>，容器必须对相关的URL进行认证。
- ▶ 如果不存在<auth-constraint>，容器允许不经认证就能访问这些URL。
- ▶ 为提高可读性，可以在<auth-constraint>中增加一个<description>。



没有<auth-constraint>与空<auth-constraint/>的作用刚好相反！

要记住：如果没有说哪些角色受约束，那么任何角色都不受约束。但是，一旦放入一个<auth-constraint>，那么只有明确指定的角色允许访问（除非<role-name>使用了通配符“*”）。如果希望任何角色都不能访问，就必须放入<auth-constraint/>，但是它必须为一个也没有！”

这个表的规则解释：

- 1 合并单个的角色名时，所列的所有角色名都允许访问。
- 2 角色名“*”与其他设置合并时，所有人都允许访问。
- 3 空的<auth-constraint>标记与其他设置合并时，所有人都不允许访问！换句话说，空的<auth-constraint>就是最后“宣判”！
- 4 如果某个<security-constraint>元素没有<auth-constraint>元素，它与其他设置合并时，所有人都允许访问。

7. 程序式安全与声明式安全的故事

```
if( request.isUserInRole("Manager") ) {  
    // 处理UpdateRecipe页面  
    ....  
}  
else {  
    // 处理ViewRecipe页面  
    ....  
}
```

↑ 是谁提出“Manager”是一个角色名？如果写这个servlet的人不了解你公司的角色安排呢？

8. 基于表单的验证

① DD中……

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/loginPage.html</form-login-page>
    <form-error-page>/loginError.html</form-error-page>
  </form-login-config>
</login-config>
```

② loginPage.html中……

```
Please login daddy-o
<form method="POST" action="j_security_check">
  <input type="text" name="j_username"> ← 容器要求HTTP请求把用户名存储在j_username中
  <input type="password" name="j_password"> ← 容器要求HTTP请求把口令存储在j_password中
  <input type="submit" value="Enter">
</form>
```

③ loginError.html中……

```
<html><body>
  Sorry dude, wrong password
</body></html>
```

9. HTTPS 传输

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

```
<security-constraint>
  <!--app>          <transport-guarantee>的合法值
  在一起来
```

你可能不会指定NONE，因为如果你不打算保护数据，就没有必要使用<user-data-constraint>！

10. 选择题

哪些类型的认证需要一种特定的HTML action? (选出所有正确的答案。)

- A. HTTP 基本认证
- B. 基于表单的认证
- C. HTTP摘要认证
- D. HTTPS客户认证

-B是对的，要进行基于表单的认证，登录表单的action必须是j_security_check。

第十三章 过滤器的威力

1. 过滤器的基本特性

- a) 容器知道过滤器自己的 API
- b) 过滤器有自己的生命周期: init() destroy() 和 doFilter()
- c) 都要在 DD 中声明

2. 过滤器的 Java 类

真正的工作在doFilter()中完成

每次容器认为应该对当前请求应用过滤器时，就会调用doFilter()方法。doFilter()方法有3个参数：

- ▶ 一个ServletRequest
(而不是HttpServletRequest)!
- ▶ 一个ServletResponse
(而不是HttpServletResponse)!
- ▶ 一个FilterChain

过滤器的功能要在doFilter()方法中实现。如果过滤器要把用户名记录到一个文件中，就要在doFilter()中完成。你想压缩响应输出吗？也要在doFilter()中实现。

```
public class BeerRequestFilter implements Filter {  
  
    private FilterConfig fc; // 必须实现init(), 通常只需在其中  
    // 保存配置(config)对象。  
  
    public void init(FilterConfig config) throws ServletException {  
        this.fc = config;  
    }  
  
    public void doFilter(ServletRequest req,  
                        ServletResponse resp, // doFilter()中才做具体的工作……注意，  
                        FilterChain chain) // 这个方法并不取HTTP请求和响应对象  
    throws ServletException, IOException { // 而只是要常规的ServletRequest  
                                         // 和ServletResponse对象。  
  
        HttpServletRequest httpReq = (HttpServletRequest) req; // 但是我们能确定请求和  
        String name = httpReq.getRemoteUser(); // 响应对象肯定可以强  
        if (name != null) { // 制转换为相应的HTTP子  
            fc.getServletContext().log("User" + name + "is updating"); // 类型。  
        }  
  
        chain.doFilter(req, resp); // 这是接下来要调用的过滤器或servlet，  
    } // 后面几页还会更多地介绍这个内容。  
}
```

3. 调用入栈



得到请求时，容器会调用Filter3的doFilter()方法，它会在此运行，直到遇到其中的chain.doFilter()调用。

容器把Filter7的doFilter()方法压入栈顶，并执行该方法，直到遇到其中的chain.doFilter()调用。

容器把servletA的service()方法压入栈顶，执行这个方法，直到结束，然后从栈中弹出。

容器把控制交还给Filter7，执行它的doFilter()方法，直到结束，然后从栈中弹出。然后容器完成响应。

4. 部署 DD

— 二八律 — 二八律 — 二八律 — 二八律 — 二八律 —

声明过滤器

```
<filter>
  <filter-name>BeerRequest</filter-name>
  <filter-class>com.example.web.BeerRequestFilter
    </filter-class>
  <init-param>
    <param-name>LogFileName</param-name>
    <param-value>UserLog.txt</param-value>
  </init-param>
</filter>
```

声明对应URL模式的过滤器映射

```
<filter-mapping>
  <filter-name>BeerRequest</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
```

声明对应servlet名的过滤器映射

```
<filter-mapping>
  <filter-name>BeerRequest</filter-name>
  <servlet-name>AdviceServlet</servlet-name>
</filter-mapping>
```

为通过请求分派请求的Web资源声明一个过滤器映射

```
<filter-mapping>
  <filter-name>MonitorFilter</filter-name>
  <url-pattern>*.do</url-pattern>
  <dispatcher>REQUEST</dispatcher>
```

-和/或-

```
<dispatcher>INCLUDE</dispatcher>
```

-和/或-

```
<dispatcher>FORWARD</dispatcher>
```

-和/或-

```
<dispatcher>ERROR</dispatcher>
```

```
</filter-mapping>
```

<filter>的规则

- ▶ 必须有<filter-name>。
- ▶ 必须有<filter-class>。
- ▶ <init-param>是可选的，可以有多个<init-param>。

<filter-mapping>的规则

- ▶ 必须有<filter-name>，用于链接到适当的<filter>元素。
- ▶ <url-pattern>或<servlet-name>元素二者中必须有一个。
- ▶ <url-pattern>元素定义了哪些Web应用资源要使用这个过滤器。
- ▶ <servlet-name>元素定义了哪个Web应用资源（准确地说，是哪个servlet）要使用这个过滤器。

声明规则

- ▶ 必须要有<filter-name>。
- ▶ 必须要有<url-pattern>或<servlet-name>元素其中之一。
- ▶ 可以有0~4个<dispatcher>元素。
- ▶ REQUEST值表示对客户请求启用过滤器。如果没有指定<dispatcher>元素，则默认为REQUEST。
- ▶ INCLUDE值表示对由一个include()调用分派来的请求启用过滤器。
- ▶ FORWARD值表示对由一个forward()调用分派来的请求启用过滤器。
- ▶ ERROR值表示对错误处理器调用的资源启用过滤器。

5. 实现装饰着模式：response 的包装类

servlet API中的包装器类功能极其强大，它们为你要包装的东西实现了所需的所有方法，并将所有调用委托给底层的请求或响应对象。你要做的只是扩展某个包装器，如果要在哪些方法中做些特殊的定制工作，只覆盖这些方法就行了。

当然，你在J2SE API中已经见过支持类，比如GUI的监听者适配器类。JSP API中也有，如定制标记支持类。尽管这些支持类及请求和响应包装器都是便利类，但包装器稍有不同，因为它们包装了所实现类型的对象。换句话说，它们不只是提供了一个接口实现，还确实包含有相同接口类型的对象的一个引用，可以把方法调用委托给这个对象【顺便说一句，这与J2SE中的“基本类型包装器”类（如Integer、Boolean、Double等等）没有任何关系】。

```
public void doFilter( request, response, chain) {  
  
    CompressionResponseWrapper wrappedResp  
    = new CompressionResponseWrapper(response); ←  
  
    chain.doFilter(request, wrappedResp);  
    // 这里完成压缩逻辑  
}
```

关于过滤器，哪些说法是正确的？（选出所有正确的答案。）

(Servlet v2.4 51页)

- A. 过滤器可以用于创建请求或响应包装器
-B 不对，因为完全说反了。
- B. 包装器可以用于创建请求或响应过滤器
- C. 与servlet不同，所有过滤器初始化代码都应当放在构造函数中，因为没有**init()**方法
-C 不对，因为确实有一个用于过滤器初始化的**init()**方法。
- D. 过滤器支持一种初始化机制，它包含一个**init()**方法，使用过滤器处理请求之前肯定会调用这个方法
- E. 过滤器的**doFilter()**方法必须按顺序在输入的**FilterChain**对象上调用**doFilter()**，以确保所有过滤器都有机会执行
-E 不对，因为如果一个过滤器想阻塞进一步的请求处理，不一定要调用**doFilter()**。
- F. 在输入的**FilterChain**上调用**doFilter()**时，过滤器的**doFilter()**方法必须为之传入**ServletRequest**和**ServletResponse**对象，而且必须是原来传入过滤器**doFilter()**方法本身的**ServletRequest**和**ServletResponse**对象
-F 不对，因为过滤器可能“包装”请求或响应对象，再传递包装后的对象。
- G. 过滤器的**doFilter()**可以阻塞进一步的请求处理