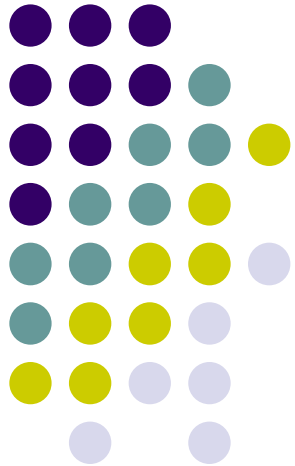
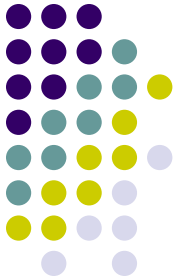
第七章 基于JSP的Web程序开发



主要内容

- JSP概述
- JSP的编程原理
- JSP所需的运行环境
- JSP简单页面示例
- JSP编程模型
- JSP标签扩展
- JSP/Servlet联合编程

JSP概述

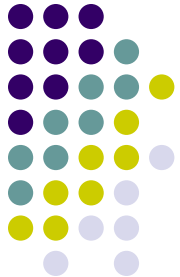


1、JSP介绍

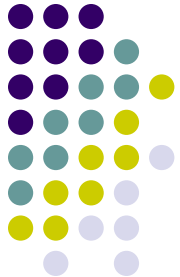
JavaServer Pages(JSP)是一种服务器端脚本技术，它可以用来生成包含动态Web内容的页面，如HTML页面，我们可以把使用JSP理解成脚本语言和Java程序代码嵌入到HTML页面中，这样就可以使用JSP的语法、Scriptlet和JavaBeans来实现各种复杂的逻辑功能。

而Servlet则不同，它使用输出流将HTML代码输出给Web服务器，然后在访问者的浏览器上显示，我们可以把Servlet看作是把HTML代嵌入Java程序代码中。

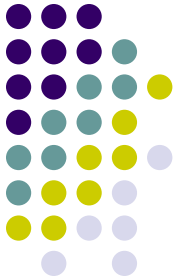
从表面上看，这两种技术并没有多大的差别，况且JSP的内部实现仍然是将JSP翻译成Servlet后运行，也就是说，在运行JSP代码的容器内部，JSP仍然是作为Servlet运行的。



但是在实际应用中，那些使用Java来编写Servlet程序的程序员往往并不是用户界面的设计者，他们并不擅长编写美观的Web页面或进行相关的美工设计，JSP技术使得Java编程工作和HTML页面设计工作分离开来，使得Web网站的开发更加有效。



如果从简单的完成动态Web页面编写这一任务来说，使用JSP也比使用Servlet要简单的多。因此JSP技术的确实Java平台技术在Web应用程序设计上的一个大发展，在Sun公司全力推广J2EE框架结构中，JSP也是作为首选技术之一，Sun公司建议开发人员尽量使用JSP技术实现表现逻辑，除非必需，尽量少用Servlet技术，同时也建议容器厂商尽量面向JSP进行优化。



2、JSP技术的主要优点：

(1) 一次编写，各处运行

JSP技术是Java平台的一部分，它继承了Java技术的平台无关性的特点。这种平台不仅反映在用它编写的动态页面上，也反映在对Web服务器和底层服务器构件的独立上，你可以在任何得平台上编写JSP页面，然后在任何得Web服务器或支持Web的应用服务器上运行，通过任何的浏览器访问，并使用在任何平台上编写的、运行在任意平台上的Java服务器构件。

(2) 丰富而且高质量的工具支持

正是由于JSP技术的平台无关性，使得越来越多的供应商支持JSP技术，这样你在使用JSP时就可以选择最适合你的、质量最好的工具，而不象ASP技术一样必须限定在微软的平台和工具上，这将使得开发效率极大的提高，并提高软件的质量。

(3) 通过构件和标签(tag)来实现重用

重用是现代软件开发的一个重要的目标，JSP技术十分强调使用Java平台提供的可重用的构件，如：JavaBean构件、EJB构件和标签库(tag library)。构件的使用极大的提高了Web应用的开发效率，相对于其他构件来说，Java平台提供的构件技术是最容易构造和维护的，这也将无形的扩展JSP的功能。

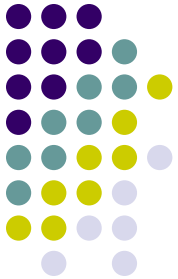
(4) 通过指令(action)和标签库增强页面开发能力

JSP技术本身已经可以很好的支持动态页面的开发，它同时还可以通过指令和标签库进一步增强页面开发能力，使得页面的开发更有效。指令和标签库可以将有用的一组功能或页面模式进行封装，这样就可以直接使用它们所创造出相应的页面部分。

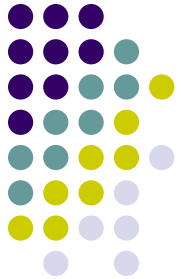
这些指令还可以被页面编辑工具处理，直接在页面编辑工具中使用，JSP标准中已经定义了一组标准的指令，用来实例化JavaBean，设置属性等。

3、JSP与其他服务器脚本语言的比较

	JSP	ASP	PHP
内部机制	Servlet	ISPAI	CGI
执行方式	编译执行(多线程)	解释执行(单线程)	解释执行(单线程)
执行开销	小	较大	较大
基本功能	由JDK支持	通过函数支持	通过函数支持
组件支持	JavaBean 、 定 制 标签、EJB	ActiveX、DLL	无
数据库支持	JDBC,支持广泛	ADO,支持广泛	内部实现，支持广泛
XML支持	支持	支持	支持
企业应用	有J2EE平台支持	由DNA框架支持	无



	JSP	ASP	PHP
传统系统集成能力	由J2EE平台支持	由DNA框架支持	无
扩展性	很好	好	不好
安全性	好	一般	一般
易用性	较好	好	好
开发工具	丰富	丰富	少
平台支持	跨平台	Windows平台	跨平台
平台移植	很好	不能移植	好



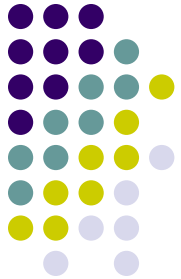
4、一个简单的JSP页面例子：

和前面一章一样，在浏览器上显示当前时间

JSP页面清单如下：

```
<HTML>
<HEAD>
<TITLE>显示当前时间-JSP</TITLE>
</HEAD>
<H1>显示当前时间-JSP</H1>
```

```
<% java.util.Date dt=new
    java.util.Date(System.currentTimeMillis());%>
  <%=dt.getHours()%>
  :<%=dt.getMinutes()%>
  :<%=dt.getSeconds()%>
</BODY>
</HTML>
```



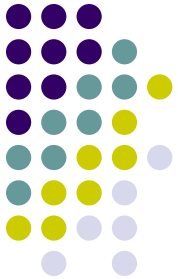
当有浏览器请求此页面时，将在浏览器上显示当前的时间。而JSP程序的实际运行过程是：

它被首先翻译成相应的Servlet，然后这个Servlet被编译成为Java字节码，系统再执行这个Java字节码程序。对于什么时候把JSP翻译成为Servlet程序，JSP标准没有硬性规定，但是一般都是用户第一次访问JSP程序时进行翻译，所以每次第一次运行时可能会慢一些，以后的访问系统直接编译好的字节码，执行速度就快了。

而且由于Servlet是常驻内存的，只要Servlet的运行实例没有被Servlet容器移走，那么直接执行内存中的程序，省去了解释执行的脚本语言(ASP、PHP)的加载脚本和解释脚本的时间。

况且一般情况下，JSP翻译成的Servlet可以按照多线程模式运行，所以可以并发地响应更多的客户访问，多个线程并发的执行的效率也要比传统的CGI方式的并发效率高。

JSP的编程原理



- JSP的基本语法和语义
- JSP的内部对象及使用
- 标准JSP指令
- JSP的生命周期

1、JSP的基本语法和语义

JSP的基本语法定义了JSP程序的基本语言规则，如注释、表达式、脚本段(scriptlet)的书写规则。JSP的基本语法还包括编译指示(Directive)，用来告诉JSP容器如何处理JSP页面，例如：设置页面属性、包含其他脚本文件等，它们是JSP的基础，也是我们最常使用到的。

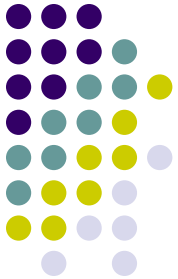
(1) 注释

JSP代码中可以包含两种注释：HTML注释和隐含注释。

HTML注释：是发送到客户端的注释，产生的效果就像是直接在HTML页面书写的注释一样，或者说这种注解的内容将显示在生成的HTML页面上。

HTML注解的语法格式为：

<!--注解 [<%= 表达式%>]-->

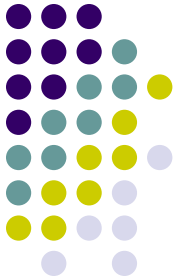


例如：<!-- 这是一段HTML注释-->

其实JSP代码中的HTML注释完全可以理解成同HTML标准定义的HTML注释语法格式基本一样，不同的是JSP代码中的HTML注释内容可以包含表达式，任何在页面中有效的表达式都可以被使用，这样HTML注解也就可以产生动态的注解内容了。

例如：

<!--这段注释在<%=(new.java.util.Darte()).toLocaleString()%>生成-->那么产生的注解:<!--这段注释在2005-3-815:49:00生成-->



隐含注释：如果吧HTML注释理解成对HTML页面的注释，那么隐含注释则是对JSP页面程序本身的注释。JSP容器将忽略这些注释信息，对它们不做任何处理，也不会发送到客户端。

隐含注释的语法格式为：

<%--注释--%>，例如<%--开始数据库操作--%>

需要注意的是注释不能嵌套，JSP页面中的scriptlet代码使用Java注释，也能产生如同JSP隐含注释一样的效果。

(2) 声明

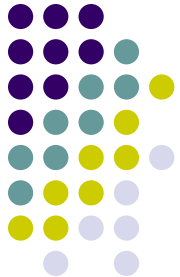
声明用于说明可以在JSP脚本段中使用的变量和方法。注意变量和方法必须在JSP程序中使用它们之前就进行说明。

声明的语法为：<%! 声明;[声明;]+...%>

例如：<%! int i=0;String a;%>

需要注意的是被声明的变量和方法的有效范围是整个页面程序。也就是声明它的页面和所有被静态包含的页面。

另外，要尽量避免使用全局变量这样的解释，因为在JSP/Servlet程序中，全局这个范围不好定义，一个Web应用程序可以包含多个JSP、Servlet程序以及许多的JavaBean和辅助支持类，JSP/Servlet程序之间也可以相互包含，用户请求可以被转发和重定向，多个用户请求可能属于同一个会话过程等，这些Web程序的复杂性使得全局这个概念不好定义。



(3) 表达式

任何编程语言的语法中都包含表达式，JSP的表达式的语法是：

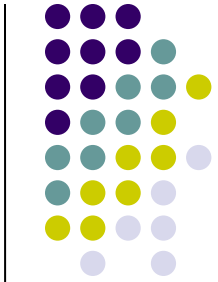
<%= 表达式 %>

JSP容器将会计算这个表达式，把结果转换成字符串，然后插入JSP文件中表达式出现的位置。因此，表达式是一种直接在JSP代码中插入动态内容，而不是用脚本段(scriptlet)的好办法。

由于表达式可以插入到JSP文件的任何地方，而且自动进行了字符串转换，所以在仅仅需要插入一小段动态内容时，表达式要比使用scriptlet灵活。

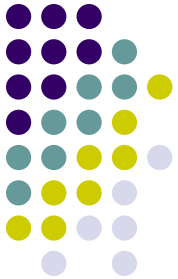
例如：

```
<% while (sqlRst.next()){
String Cname=sqlRst.getString(2);
Cname=new String(Cname.getBytes( “ISO8859_1” )
,” gb2312” );%>
    <tr>
    <td width= “100%” height= “18” >
    <a href= “list.jsp?CID=<%=sqlRst.getInt(1)%>
    &Cname=<%=Cname%>” class= “left” >
    <%=Cname%></a></td>
    </tr>
<%}%>
```



上面是一段典型的使用表达式的程序，程序中后面的三个表达式直接在生成的HTML页面的相应部分插入从数据库查询得到的数据内容。

注意因为JSP容器将表达式结果转换成字符串，因此虽然第一个表达式<%=sqlRst.getInt(1)%>计算结果是一个整数，但也不需要转换就可以直接插入到指定的位置，系统会将整数转换为字符串。另外由于JSP表达式的基础是Java语言的表达式，所以它可以任意复杂的有效Java语言表达式，但是表达式的结尾不能有分号(；)。



(4) 脚本段

脚本段(scriptlet)用来在JSP页面中包含一段Java程序。当JSP程序中需要处理比表达式更复杂的逻辑结构时，它将使用Java代码，也就是使用脚本段把所需的Java程序段包含进来。

脚本段中的程序就是Java程序，只要是有效的Java程序都可以包含在脚本段中，而其他的HTML标记、其他JSP语法单元都应该放在脚本段之外。例如前面的代码，前几行就是一个scriptlet。

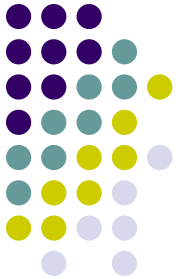
JSP程序中使用scriptlet可以充分发挥Java语言的全部功能，实现任何复杂逻辑结构，而且它同ASP、PHP等服务器端脚本语言中的脚本代码有着本质的区别。它本质上就是Java程序，将先被编译，然后再被执行，而其他的脚本代码是解释执行的。

(5) 编译指示

编译指示是JSP文件通知JSP容器关于编译信息的手段，它们自身不会产生任何输出信息。在JSP1.1标准中，编译指示都具有统一的语法格式：

<%@ directive {attr=“value”}*%>

JSP标准中定义了三种标准的编译指示：include 编译指示、page 编译指示和taglib编译指示。

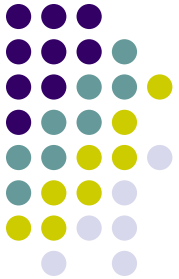


include 编译指示

include 编译指示用来在JSP程序文件中插入另一个静态文件，其语法为：<%@ include file =“文件的相对地址”%>

当编译JSP文件时，include编译指示中file属性指定的文件的内容就被插入到原JSP文件的相应位置，一起被翻译成Servlet进行编译。这种包含过程称为静态包含。JSP文件及其包含的文件被称为一个翻译单元(translate unit)，被包含的文件可以是html文件、JSP文件、文本文件或任何Java代码片断。

另外，需要注意的就是include编译指示中指定的文件URL都是相对路径。而且被包含的文件中不要出现<body></body><html></html>等会引起同包含文件的标签发生冲突的标签内容，应为文件包含后是作为一个整体被JSP容器处理，所以如果标签发生冲突，将产生意想不到的后果。



page 编译指示

page 编译指示定义作用于整个JSP文件的属性，

其语法为：

<%@ page

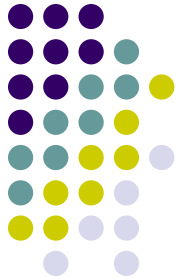
[language=“java”]

[extends=“package.class”]

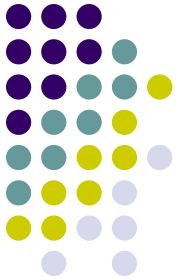
[import=“{package.class | package.*},…”]

[session=“true|false”]

[buffer=“none|8kb|sizekb”]



```
    [autoFlush= “true|false” ]
    [isThreadSafe= “true|false” ]
    [info= “text” ]
    [errorPage= “relativeURL” ]
    [contentType= “ mime Type[;charset=characterSet] ” |
     “ text/html;charset=ISO-8859-1” ]
    [isErrorPage= “true|false” ]
%>
```



例如以下合法的page编译指示：

<%@ page import=“java.util.*,java.lang.*” %>

<%@ page buffer=“5kb” autoFlush=“false” %>

<%@ page errorPage=“error.jsp” %>

page编译指示定义作用于整个JSP文件的属性信息，更确切地说，是定义作用于整个“翻译单元”的属性信息，即JSP文件及其静态包含的文件，但是作用范围不包括动态包含的文件。

在整个翻译单元内，可以多次使用page编译指示，而且不论是在那里插入<%@ page%>指示，其定义的属性都将作用于整个翻译单元。

但是，除了import属性以外，其他的属性最多只能定义一次。因为import属性类似于java语言的import语句，所以可以定义多次，包含多个Java包。虽然在JSP文件的任何地方都可以写page编译指示，但是建议还是在JSP的开头书写。

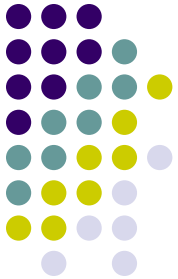
下面简单介绍page编译指示中各属性的含义：

1、language=“java”

定义JSP文件中使用的脚本语言，目前只能是Java

2、extends=“package.class”

指定JSP文件将被编译成的Java类的父类。这个属性一般不需要设定，因为JSP容器在编译JSP文件时会自动地为它设置相应的父类，不需要用户干涉。



3、import=“{package.class|package.*},…”

用来引入所需的Java包，这些包引入后就可以用于JSP程序中的脚本段、表达式和声明中了，多个包之间用逗号(,)分隔。

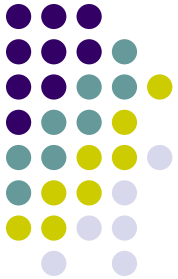
注意以下几个包是被自动引入的，自此不需要再次引入。

java.lang.*、javax.servlet.*、javax.servlet.jsp.*

和javax.servlet.http.*

4、session=“true|false”

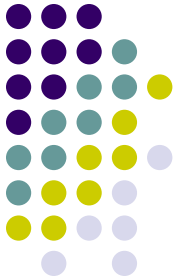
定义客户访问此JSP文件时是否将属于某个会话过程，默认值是true，如果session的属性为true，那么JSP文件就可以直接使用session对象了，否则就不能使用session对象，同时也不能使用 scope 属 性 为 session 的 <jsp:useBean> 指 令 和 作 用 域 为 session的对象



5、buffer=“none|8kb|sizekb”

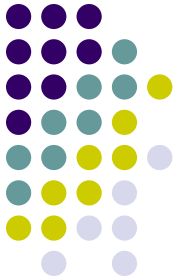
设置JSP程序向客户端浏览器输出的输出缓冲区的大小，默认值为8kb。

JSP文件最终是被编译成为一个Java Servlet类来执行的，Servlet的输出对象具有输出缓冲区，Servlet输出的信息首先被保存在缓冲区中，在输出到客户端。改变缓冲区大小将改变Servlet对输出的控制能力，例如如果把它设置为0，那么任何输出会直接发送到客户端，无法在输出到客户端之前进行内容修改，也无法重定向用户请求。



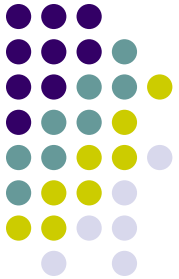
6、autoFlush=“true|false”

定义当输出缓冲区满了以后，是否自动将缓冲区中的内容发送到客户端，默认值是true，即自动发送。如果被设置为false时，缓冲区溢出时将产生一个异常，JSP程序截取到这个异常就可以在将缓冲区中的内容发送到客户端之前再次修改内容或者进行其他控制，但是如果buffer为none时，autoFlush属性设置为false。



7、isThreadSafe=“true|false”

定义JSP程序是否线程安全，默认值为true。JSP是线程安全意味着JSP程序可以处理多个客户端的并发访问，JSP容器就会把并发的客户端请求发送到这个JSP文件，即此JSP程序支持多线程并发处理。但如果为false时，这个JSP文件不支持客户端的并发访问，JSP容器也就每次发送一个客户请求给它处理，即是一种单线程模型，性能会受到影响。

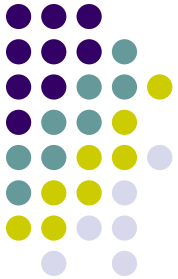


8、info=“text”

定义了一个字符串包含到被编译的页面程序中，以后如果需要就可以通过Servlet.getServletInfo()方法获得它

9、errorPage=“relativeURL”

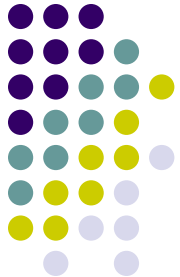
定义了一个异常处理页面，JSP程序中没有捕获的异常都将发送到这个页面。



10、isErrorPage=“true|false”

标示该JSP程序是否是另一个JSP程序的异常处理程序，默认值是false。

这个属性往往同errorPage属性成对使用，在另一个JSP程序中指定其errorPage的属性为这个JSP程序，而这个JSP程序中的isErrorPage属性设置为true，这样那个JSP文件中抛出而未被捕获的异常就会被发送到这个页面，JSP容器也会将控制流转到这个页面，而这个页面就可以自动使用Exception对象来获得这个异常，然后进行处理，这种机制往往是用来实现统一的错误和异常处理功能。



11、 contentType= “ mime Type[;charset=characterSet] ” | “ text/html;charset=ISO-8859-1”

说明JSP程序发送给客户浏览器的内容的MIME类型和字符集。MIME类型一般都是text/html，当然也可以是其他类型如image/gif，那么发送的内容就被解释成为gif图片。

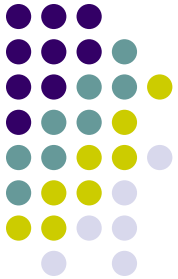
另外最常用的就是字符集了，例如要显示简体中文则需要指定字符集为gb-2312。

taglib 编译指示

taglib 编译指示声明一个标签库，并指定在JSP文件中使用的标签前缀。Taglib编译指示的语法为：

<%@ taglib
uri=“URLToTagLibrary” prefix=“tagPrefix” %>

taglib编译指示中，属性uri指明了定义标签库的统一资源标识符(URL)，属性prefix则定义了JSP程序中使用标签的标签前缀。



例如如下代码：

```
<%@ taglib uri=“struct-taglib” prefix=“struct” %>
<struct:For…>
……
</struct:For…>
```

这段代码声明了一个标签库，标签前缀是struct，标签库URI是“struct-taglib”。之后这个JSP程序中就可以使用这个标签库中定义的标签了。

2、JSP的内部对象及使用

如果你使用过其他服务器脚本语言(如ASP)编程，一定对内部对象的概念不陌生。内部对象就是一些系统预定义(声明)的对象，当编写JSP程序时，不需要声明它们就可以直接使用。

之所以系统要声明这些对象，是因为这些对象对于编程来说是十分常用和必需的，程序员往往都要去声明和使用它们。系统预先声明既方便了程序员编程，也统一了内部对象的使用和命名，对于程序的维护也更加有利。

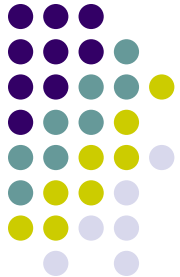
所以内部对象的使用对JSP程序设计是十分重要的，它们是JSP程序设计的重要基础。

另外时刻记住JSP是整个Java技术平台的一部分，因此每个内部对象都可以在Java类库的核心包或Servlet API包中找到它们所属的类或接口定义。对它们的使用也同其他Java对象的使用是一样的，只是每个对象实现了对动态Web程序设计有用的不同功能或接口而已。实际上这些内部对象许多都是从Servlet API实例化得到的。

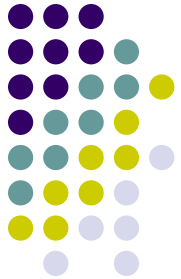
(1) request 对象

request对象封装了客户端的请求信息，例如客户访问的URL、提交的表单内容等等，它是最常用的对象之一。

request对象是从javax.servlet.ServletRequest的子类实例化得到的。因为JSP页面目前都是用于编写HTML页面，响应HTTP协议，所以一般都是从类javax.servlet.HttpServletRequest实例化得到的。



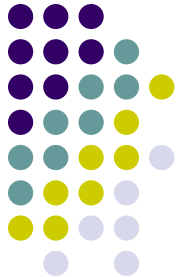
回顾Servlet程序设计，Servlet的doGet和doPost等方法的调用参数中有一个javax.servlet.HttpServletRequest类型的参数，其实request对象就相当于这个参数，在JSP翻译成Servlet中，request对象往往都是翻译成这些方法的相应参数。



request对象是最常用的对象之一，有最常用的两种使用方式，一是使用getParameter、getParameterNames和getParameterValues方法获得客户端传递的参数(通过URL或表单传递)，另一种方式是通过request对象获得关于客户端请求的各种信息数据。

(2) reponse 对象

reponse对象表示对客户端的响应。同Servlet编程一样，通过reponse对象，我们可以完全控制发送到客户端的数据，但这种控制是比较低级的控制，而JSP页面一般用来实现Web程序中的表现逻辑，所以一般不直接使用response对象控制发送的内容，最多在JSP的脚本段中使用out对象在文本级上输出内容。

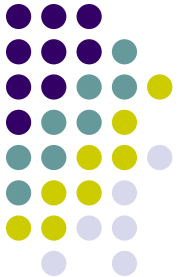


在绝大多数情况下，response对象只是用来设置响应流的一些属性，主要是响应流的头信息，例如设置响应页面的缓存属性防止被浏览器缓存；或者设置MIME属性，以输出其他类型的信息，如图片等。

(3) pageContext 对象

pageContext对象封装了关于JSP程序运行的上下文信息。实际上，它主要是用来封装JSP容器在运行JSP程序时的一些实现相关的属性，并提供相应的处理方法。

通过使用pageContext对象，JSP程序就可以使用某个JSP容器实现特有的特征，如：高性能的JspWriter，同时保证能够在所有兼容的JSP容器中运行，它属于比较深入和高级的程序设计，一般情况下是用不到pageContext对象。

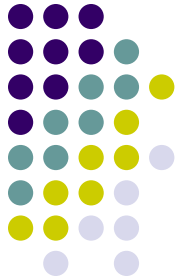


(4) session 对象

当JSP程序使用会话时，即page编译指示中session=“true”，则session对象表示了客户端和服务器之间建立的会话。

通过session对象可以在同一个会话过程的不同访问之间传递数据，而传递数据最常用的方法是使用setAttribute方法将数据写入session对象，然后在另一访问中使用getAttribute方法从session对象中读取数据，这种使用方式同Servlet是一样的。

session对象是最常用的内部对象之一，尤其是当我们需要实现象购物车等需要保持客户会话过程的程序时，session更是必须要使用的。



(5) application 对象

application对象封装了JSP程序实现类(即Servlet)的上下文，这个上下文是从Servlet配置对象中获得的。application对象封装的上下文同pageContext封装的上下文不同，application对象的上下文是Servlet容器来提供的，关于JSP的实现类的运行上下文，而pageContext主要是封装关于JSP容器的实现相关特性的信息，两者用途不同。

(6) out 对象

out对象表示一个向输出流输出文本信息的对象。它比reponse对象的输出级别高，可以直接生成用于浏览器显示的HTML文本内容。虽然out对象的输出控制级别比response高，但是它内部仍包含了一个缓冲区，因此它可以对这个缓冲区进行灵活的控制，所以一般情况下，使用out对象就足以完成各种输出要求，不必使用response。

在实际编程中，绝大多数静态的页面内容都是直接在JSP程序中书写HTML文档的方式实现。在JSP程序被翻译成Servlet类时，系统会自动把这些HTML文档转换成采用out对象输出的代码，所以对静态页面，我们没有必要采用out对象输出的方式书写。而且采用out对象输出的静态内容降低了程序的可读性，增加了编程和调试的复杂度。

好的编写Web程序的风格是，在JSP程序中尽量不是用out对象输出静态页面元素，静态页面元素直接采用HTML文档书写，动态的内容尽量采用JSP表达式的方式书写，只有以上方式无法实现的情况下，在采用out对象输出文本信息。这同时也符合JSP技术在Web编程中的地位，作为一种服务器段脚本语言，实现程序的动态表现内容和简单逻辑。

(7) config 对象

config对象封装JSP程序的实现类(Servlet)的初始化信息。它的使用同Servlet编程中的ServletConfig类的使用一样。

(8) page对象

page对象直接指向JSP程序运行的实现类，如果脚本语言使用java，page对象是即上同this是一个对象，一般不会用到，只在某些特殊场合用到。

(9) exception 对象

exception对象只能在报错页面中使用，即page编译指示的isErrorPage属性等于true的页面中使用，它捕捉源JSP页面抛出的异常。一般使用它捕捉这些异常，然后进行统一的报错和错误处理。

3、标准JSP指令

指令(action)是JSP语法的重要组成部分，指令可以影响当前的输出流，也可以创建或者修改对象。我们也可以把指令看成是一种封装形式，它把一组动作或者功能进行了封装，以脚本语言的形式进行使用。

JSP标准中定义了一些标准的指令，这些指令在所有的JSP容器中都会被实现，而用户也可以扩展JSP指令，这就是自定义标签，然后通过标签库编译指示导入JSP程序中使用。(后面将详细介绍)。

JSP标准指令可以分为两类：一类用于改变程序的控制流或输出流，如<jsp:include>和<jsp: forward>，称之为控制指令；另一类用于操作对象，如<jsp:useBean>和<jsp:plugin>，称之为指令对象。

1、控制指令

(1) <jsp:forward>指令

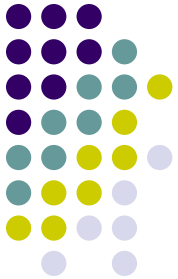
<jsp:forward>指令将客户的请求发送到(或称转移到)另一个程序(文件)处理，这个文件可以是HTML文件、JSP程序或Servlet程序。

注意，<jsp:forward>指令的执行将立即终止当前JSP程序的运行，其后面的代码也不会被执行，而且指令的执行将清除当前输出缓冲中的内容，不再发送到客户端，如果JSP程序设置了不缓冲输出，指令执行将会抛出异常。

所以，一般情况下如果使用<jsp:forward>指令，JSP程序就应该缓冲输出。指令改变控制流程，原来的输出也就没意义了。

<jsp:forward>指令的语法有不带参数和带参数两种形式，如下：第一种：

<jsp:forward page=“{relative URL |
<%=expression%>}” />



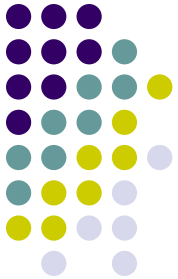
第二种：

<jsp:forward page=“{relative URL | <%=expression %>}” >

<jsp:param name=“parameterName” value=“{parameterValue|<%=expression%>” />*

</jsp:forward>

其中page=“{relativeURL|<%=expression%>}说明了目标文件的相对地址，注意目标文件的地址可以是一个表达式，这样就可以根据动态条件将程序流程转到不同的夜面处理。



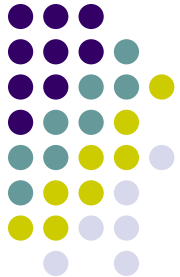
例如：<jsp:forward page=“/Servlet/login”/>

<jsp:forward page=“/Servlet/login”>

<jsp:param name=“username” value=“java”/>

</jsp:forward>

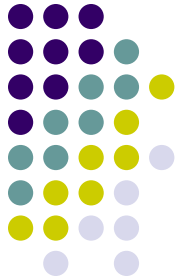
<jsp:param name=“parameterName” value=“parameterValue”|<%=expression%>”/>*可以用来向请求中增加擦参数，注意原JSP程序获得的客户请求信息将全部传递给目标文件，所以<jsp:param>子句说明的参数是添加到客户请求中的，而不是覆盖掉原有参数。



(2) <jsp:include>指令

<jsp:include>指令有两种功能，包含一个静态文件或者调用一个动态文件。它的语法形式如下：

第一种：<jsp:include page=“{relativeURL|<%=expression %>}” flush=“true”/>

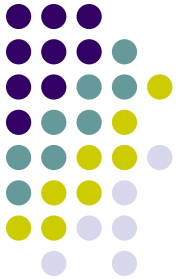


第二种：

<jsp:include page=“{relativeURL|<%=expression%>}”
flush=“true”>

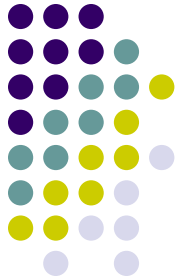
<jsp:param name=“parameterName”
value=“{parameterValue|<%=expression%>}”/>+

</jsp:include>



例如：

```
<jsp:include page= “scripts/login.jsp” />
<jsp:include page= “copyright.html” />
<jsp:include page= “index.html” />
<jsp:include page= “scripts/login.jsp” >
      <jsp:param name= “username” value= “zxg” />
</jsp:include>
```



<jsp:include>指令既可以包含一个静态文件，也可以包含一个动态文件，但是使用时注意<jsp: include>包含静态文件和包含动态文件语义是不同的。

如果包含静态文件，文件的内容是插入到JSP程序中，这同使用include编译指示包含静态文件十分相似，但如果包含动态文件，JSP容器将转到被包含的动态文件执行，执行结束后再回到原JSP文件<jsp:include>指令之后继续执行。它类似过程调用。

但是，一般情况下，我们仅仅从被包含的程序的文件名是无法判断到底是静态包含还是动态包含，因为我们可以通过Web别名机制将一个静态网页URL(如index.html)映射到一个Servlet。

所以为了编程的清晰和简单，如果是静态包含，就使用<%@ include ... %>编译指示，如果是动态包含则使用<jsp:inlcude>指令。

<jsp:include>指令的属性和<jsp:forward>指令类似，其中flush属性控制当前JSP文件的输出缓冲是否提交给用户，当前JSP标准中，flush属性的值必须为true。

2、对象指令

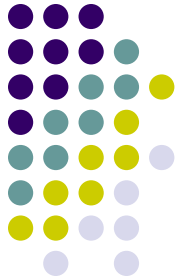
我们把<jsp:useBean>、<jsp:getProperty>、<jsp:setProperty>和<jsp:plugin>指令称为对象指令，因为它们可以在JSP程序中创建和操作JavaBean和Java applet对象。

(1) <jsp:useBean>指令

它是用来定位并实例化一个JavaBean，并给这个JavaBean赋予相应的名字，设定作用域。通过使用JavaBean，我们可以将许多数据和功能封装到一个构件对象中，而在JSP程序中使用此指令可以方便的使用这个构件。

(2) <jsp:getProperty>和<jsp:setProperty>

这两条指令分别用来获取JavaBean的属性和设置JavaBean的属性值，JavaBean有标准的模式来获取和设置其属性值，JSP标准中定义了这两条指令则使得获取和设置JavaBean的属性更加方便，如果对JavaBean的操作仅仅是设置属性值和获取属性值，则JSP程序完全不知道JavaBean的详细设计，通过脚本语言就可以完成操作，做到了程序的独立性和开发的方便性。



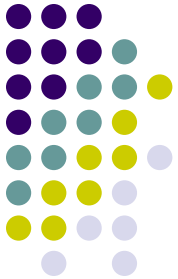
(3) <jsp:plugin>指令

它是用来执行一个Java applet或者一个JavaBean，注意这里执行的JavaBean同<jsp:useBean>指令中的JavaBean不同，这里的JavaBean是客户端的JavaBean，它是在客户端执行的，而<jsp:useBean>指令中的JavaBean是服务器端的JavaBean，它在服务器上执行，对它的所有操作在服务器上完成，而该指令则是将指定的JavaBean下载到客户端机器上，再执行它们。

4、JSP的生命周期

因为JSP最终是作为Servlet执行的，因此，毫无疑问JSP页面的生命周期明显类似于Servlet的生命周期，然而两者之间还是有差别的。

最明显的差别源于一个事实：在执行之前JSP页面必须首先翻译成Servlet代码，然后进行编译，这个翻译和编译的过程是JSP的生命周期的起点。具体如下：



(1) 翻译：

当首先建立JSP请求时，容器寻找相应的Servlet类，如果相应的类不存在，或者相应的Servlet比JSP页面要旧(表明此Servlet需要更新)，那么该JSP页面由其容器动态翻译成Servlet代码，接着编译Java代码。

(2) 编译：

在JSP页面被转换成Java代码之后，接着，这个代码被编译成Servlet类，在该代码编译之后，可以执行Servlet。

(3) 执行：

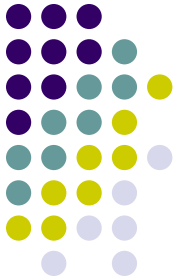
每当JSP被请求时，与JSP页面对应的Servlet由其容器调用，翻译和编译只需要发生一次，在这之后，该Servlet可以直接执行。如果JSP页面随后被修改和重新部署，翻译和编译会重复，以便确保该JSP和它对应的Servlet是同步的。

在执行的时候，JSP生命周期实际上同Servlet的生命周期是一样的。

JSP所需的运行环境

1、了解JSP规范

Sun 公司的 JSP 主页在 http://www.javasoft.com/products/jsp/index.html ，从这里还可以下载 JSP 规范，这些规范定义了供应商在创建 JSP 引擎时所必须遵从的一些规则。



2、下载所需软件：

(1)在 http://java.sun.com/jdk/处下载JDK(Java 2 SDK、Standard Edition、v 1.2.2)。

(2)在http://java.sun.com/products/jsp/处下载JSWDK(JavaServer Web Development Kit 1.0.1)。Linux用户可以在http://jakarta.apache.org/处下载Tomcat 3.0。

3、软件安装

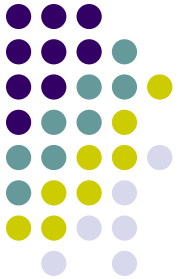
以Windows NT环境为例，JDK的安装是首先运行下载得到的jdk1_2_2-win.exe，然后修改系统环境参数，在PATH参数中加入JDK安装目录以及增加新的环境参数 CLASSPATH

JSWDK的安装仅需将jswdk1_0_1-win.zip带目录释放到硬盘根目录下（c:、d: 等），然后就可以在硬盘上找到jswdk-1.0.1目录。将来如果不想保留JSWDK，删除这个目录就可以了，没有任何系统文件和注册表遗留问题。

4、启动Web服务器

以 Windows NT 环境为例，在 jswdk-1.0.1 目录下执行 startserver.bat，就可启动JSWDK中一个支持JSP网页技术的Web服务器。为了不与现有的Web服务器（例如IIS、PWS等）冲突，JSWDK的Web服务器使用了8080端口。在浏览器的地址栏中键入http://localhost:8080或者http://127.0.0.1: 8080后，如果能看到 JSWDK 的欢迎页就说明JSP实验环境已经建成，可进入下一步实验。要关闭Web服务器则运行stopserver.bat。

JSP简单页面示例



1、生成并显示一个从 0 到 9 的字符串

```
< HTML>
   < HEAD>< TITLE>JSP 页面 < /TITLE></HEAD>
   < BODY>
   < %@ page language=“java” %>
   < %! String str=“0”; %>
   < % for (int i=1; i < 10; i++) { str = str + i; }%>
    JSP 输出之前
    <P><%=str%><P>
    JSP 输出之后
   < /BODY>
< /HTML>
```

2、会话状态维持例子

我们用三个页面模拟一个多页面的 Web 应用。

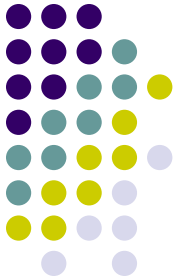
第一个页面（ q1.html ）仅包含一个要求输入用户名字的 HTML 表单，代码如下：

```
< HTML>
    < BODY>
    < FORM METHOD=POST ACTION="q2.jsp">
    请输入您的姓名：
    < INPUT TYPE=TEXT NAME="thename">
    < INPUT TYPE=SUBMIT VALUE="SUBMIT">
    < /FORM>
    < /BODY>
< /HTML>
```

第二个页面是一个 JSP 页面（ q2.jsp ），它通过 request 对象提取 q1.html 表单中的 thename 值，将它存储为 name 变量，然后将这个 name 值保存到 session 对象中。

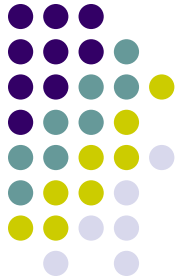
session 对象是一个名字 / 值对的集合，在这里，名字 / 值对中的名字为“ thename ”，值即为 name 变量的值。由于 session 对象在会话期间是一直有效的，因此这里保存的变量对后继的页面也有效。

q2.jsp 的另外一个任务是询问第二个问题。

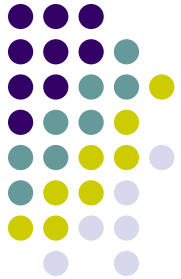


下面是它的代码：

```
< HTML>
    < BODY>
    < %@ page language="java" %>
    < %! String name=""; %>
    < % name = request.getParameter("thename");
      session.putValue("thename", name); %>
      您的姓名是： < %= name %>
    < p>
    < FORM METHOD=POST ACTION="q3.jsp">
      您喜欢吃什么？
    < INPUT TYPE=TEXT NAME="food">
    < P>
    < INPUT TYPE=SUBMIT VALUE="SUBMIT">
    < /FORM>
    < /BODY>
    < /HTML>
```



第三个页面也是一个 JSP 页面（ q3.jsp ），主要任务是显示问答结果。它从 session 对象提取 thename 的值并显示它，以此证明虽然该值在第一个页面输入，但通过 session 对象得以保留。q3.jsp 的另外一个任务是提取在第二个页面中的用户输入并显示它。



代码如下：

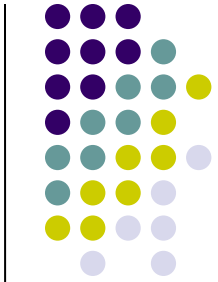
```
< HTML>
   < BODY>
   < %@ page language="java" %>
   < %! String food=""; %>
   < % food = request.getParameter("food"); String name
   = (String) session.getValue("thename"); %>
   您的姓名是： < %= name %> < P>
  您喜欢吃： < %= food %>
   < /BODY>
< /HTML>
```

3、欢迎访问者的页面

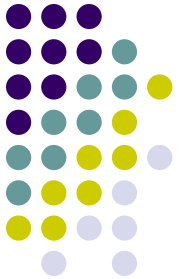
欢迎所有访问此页面的来访者，并指出它们是第几位访问者，如果是第10位访问者给出一个特殊的响应。具体JSP页面的代码：

```
<%@ page import =“hits.HitsBean”%>
<jsp:useBean id=“hits” class=“hits.HitsBean”
scope=“session”/>
<jsp:setProperty name=“hits” property=“*”/>
<!DCTYPE HTML PUBLIC “-//W3C//DTD HTML 4.0
Transitional//EN”>
```

```
<HTML>
    <HEAD>
        <TITLE>Hit Counter</TITLE>
    </HEAD>
    <BODY>
        <% if(hits.incrHits()==10){%>
        <B>Congratulations,you are the  10th person
    today!</B>
        <%}else{%>
        Welcome!You are visitor #
        <%=hits.getHits()%>
        <%}%>
```

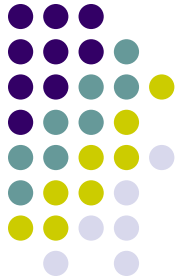


```
    <P>To revisit the page,click RELOAD below:
    <FORM method=get>
        <INPUT type=submit value= “RELOAD” >
    </FORM>
</BODY>
</HTML>
```



用于计数的HitsBean的代码如下：

```
package hits;
import java.util.*;
public class HitsBean{
    private int m_hits;
    public HitsBean(){
            resetHits();
    }
```



```
    public int incrHits(){
        return ++m_hits;
    }
    public void resetHits(){
        m_hits=0;
    }
    public int getHits(){
        return m_hits;
    }
    public void setCounter(String counter){
        m_hits=Integer.parseInt(counter);
    }
}
```

JSP编程模型

JSP的编程模型，也称应用模型，它主要涉及我们如何在Web程序中使用JSP技术，以及编写出的JSP应用程序的框架结构这类根本性问题。

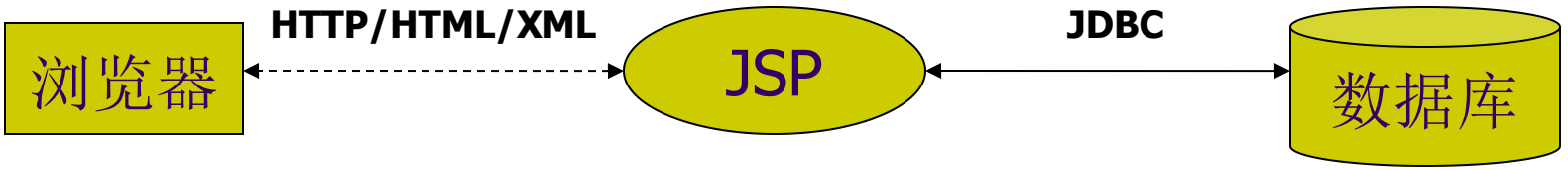
JSP的编程模型也是在不断的发展和完善中，我们主要介绍的是JSP标准中推荐的几种典型的编程模型。主要包括简单的$2^{1/2}$层模型、JSP/XML应用模型、N层模型、重定向模型、松耦合应用模型以及包含模型。

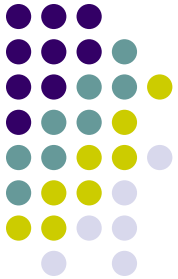
1、简单的$2^{1/2}$层模型

它是一种基本的替代传统CGI程序的应用模型，在模型中JSP程序(或Servlet程序)直接访问外部资源(如数据库)，完成对客户的请求服务。这个模型也就是Servlet的基本编程模型，只是位于中间层的不是Serlvet，而是JSP。

主要优点：简单，JSP程序员根据客户端的请求和资源的数据就可以方便的生成所需的动态内容。

主要缺点：它不能应用于需要服务大规模并发访问的应用，因为每个客户端请求都需要建立同数据库资源的连接，浪费大量的系统资源，而且此模型没有区分表现逻辑和应用逻辑，给大型项目的开发组织带来了困难，不符合现代软件工程的分工合作的思想。模型图如下：





2、N层模型

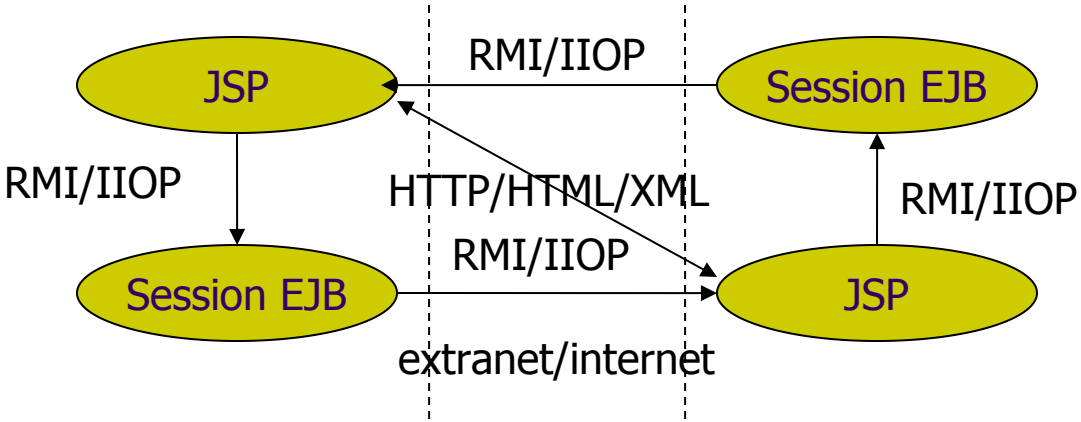
这个模型至少包含3个应用层次。JSP程序在其中作为中间层，用来实现表现逻辑，即生成对用户的显示和收集客户端请求信息。它同后台资源的交互则通过企业JavaBean(EJB)完成。

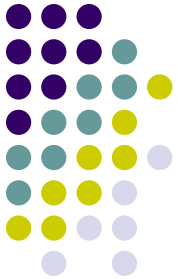
EJB服务器和EJB构件实现了对资源的统一管理，它本身可能又由多层应用结构构成，EJB层的引入克服了两层模型中的性能和任务划分问题等缺点，而且EJB服务器还可以提供事务处理、安全机制等服务，用它来开发企业级应用。

3、松耦合应用模型

松耦合模型一般应用于大型系统设计、分布式系统设计或系统集成应用之中。例如不同企业之间的供应链集成应用，这时重要的是保证系统快速灵活地集成，同时相互之间又保持充分的独立，一方的内部改动不至于导致另一方的系统修改。

在模型图示中，有两个松耦合应用。它们可以位于同一个Intranet，或位于Extranet或Internet，它们之间的关系也可以是对等的，或者“客户/服务器“关系，重要的是两个应用之间尽量通过JSP之间的HTTP协议通讯，交互HTML/XML数据，而不是通过RMI/IIOP或Java IDL这类低级接口。





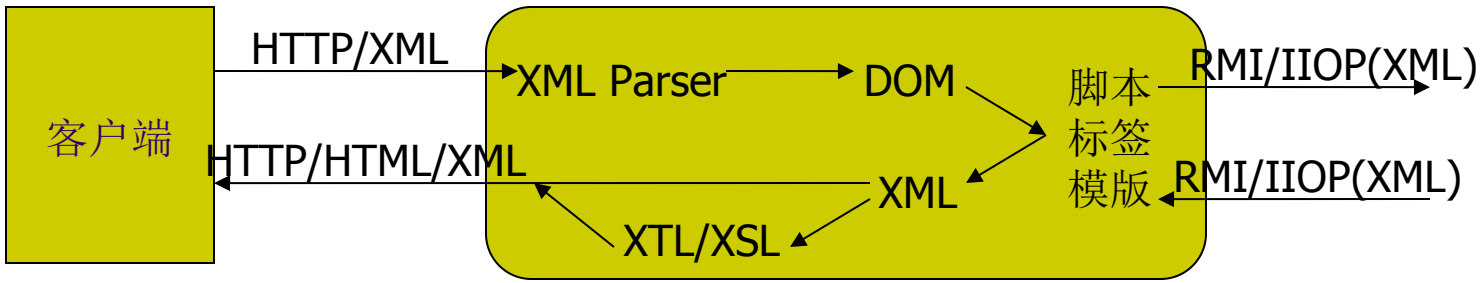
4、JSP/XML应用模型

随着XML的兴起，XML在Web程序设计中的地位变得越来越重要，现在XML已经成为Java Web应用程序设计中的一个重要部分。

简单地理解，将JSP同XML结合同传统的JSP/HTML编程没有太大的区别，但是由于XML的许多新特点和重要性，JSP中专门提出了JSP/XML模型作为一种编程模型，但这个模型仍处于发展之中。

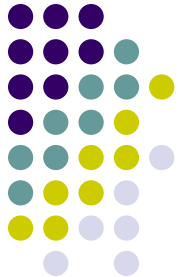
JSP/XML模型同基本JSP/HTML模型不同在于JSP程序处理的数据和产生的输出不再是HTML格式，而是XML格式。

由于XML的结构性，JSP程序直接处理XML将带来HTML所无法具有的灵活性。而且JSP技术本身也比较适合处理XML数据，例如JSP的标签扩展机制可以创建专门处理XML的指令和编译指示，使XML的处理更方便。



5、重定向模型

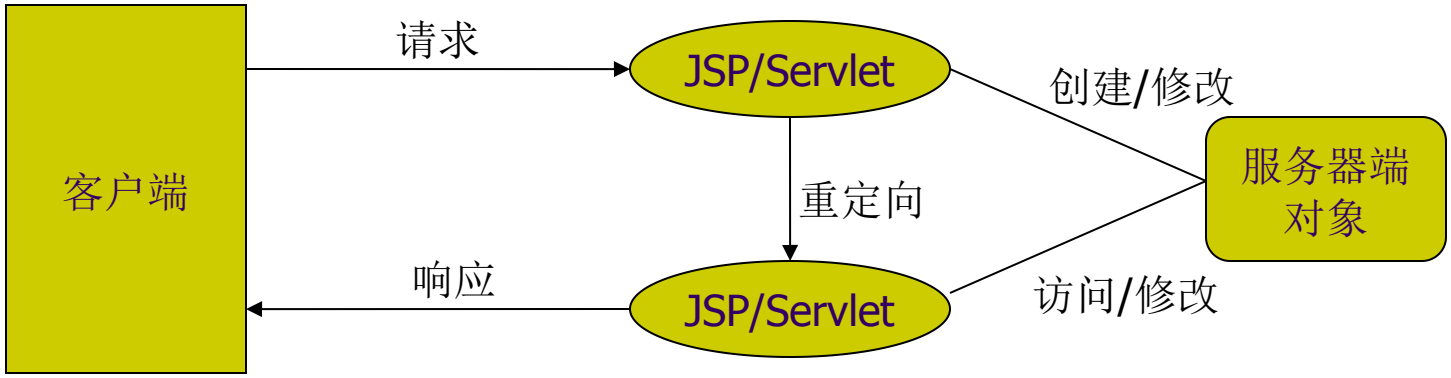
重定向模型是在简单的2层模型之上发展起来的，当Web应用中来自客户端的请求变得复杂时，专门使用一个初始JSP程序(或Servlet程序)来处理客户端的请求，然后按照需要，将客户端请求重定向到其他的JSP程序(或Servlet程序)去处理，由它们生成发送到客户端的响应。

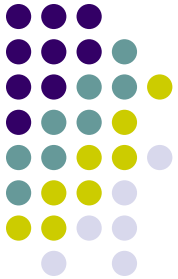


这个模型实际上在Servlet中就已经得到支持了，但是在JSP中要注意对输出缓冲的控制，因为HTTP协议的特性决定了如果输出响应流已经开始发送客户端，就不能在重定向了，所以必须对输出进行缓存控制，而在Servlet中输出都是缓存的，自然可以支持这个模型，为此JSP默认的都是带输出缓存的，这也是为了支持重定向。

将这个模型引申一下，我们可以把初始的JSP程序称作“前台JSP构件”，它不产生任何输出响应，只处理客户端请求，以及创建某些服务器对象，然后由其他的JSP程序产生输出响应，这些JSP程序称为“表现JSP构件”，它为了实现表现功能还可能去访问其他的服务器构件。这个模型基本接近了Java Web应用程序设计的一个重要模型MVC模型。

重定向模型的示意图如下：



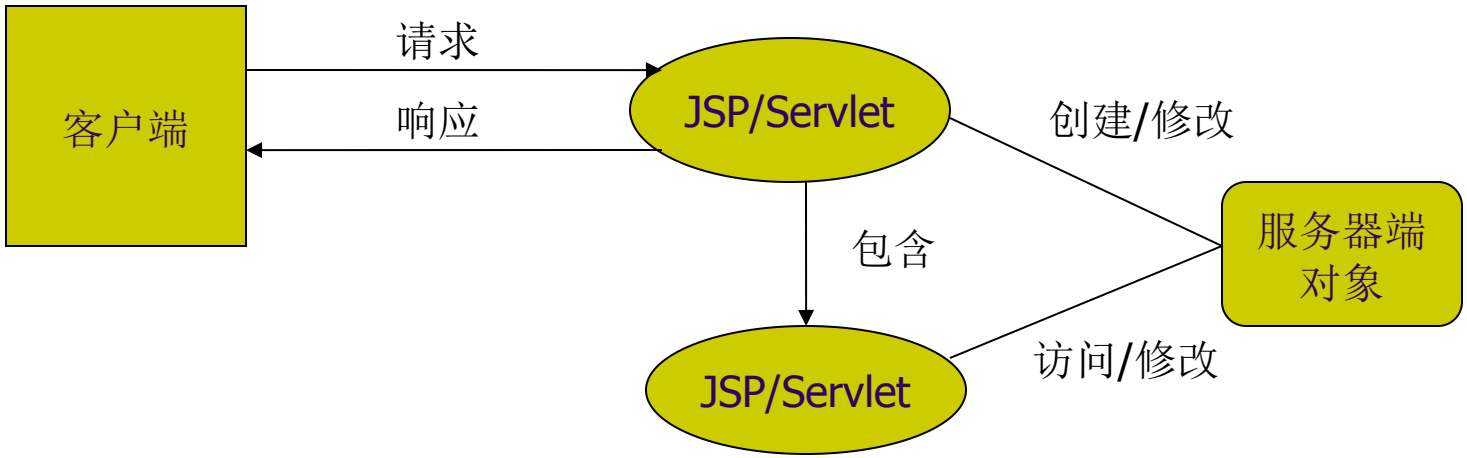


6、包含模型

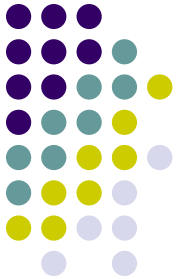
这个模型中，初始的JSP程序既处理客户端的请求，也负责生成对客户端的响应，但是在处理过程中可能会包含一些其他的JSP程序或其他文件，包含既可以是静态包含，也可以是动态包含，动态包含的效果就像是进行一次过程调用。

这个模型同“重定向模型”差别不大，也可以用在同样的场合，但是“包含模型”最常用的场合还是利用被包含的JSP程序生成同页面表现无关的内容，如XML数据，或者进行后台操作，而初始JSP程序则负责全部的表现功能。

包含模型的示意图如下：



JSP标签扩展

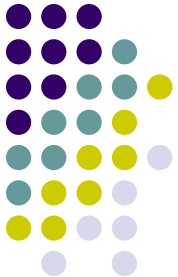


- 定制标签库(TagLib)概述
- 使用定制标签库
- 标签的分类与实现

1、定制标签库(TagLib)概述

在JSP1.1中引入了一种十分有用的新功能，那就是用户可以定义自己的JSP标签。自定义标签功能是一种十分重要的技术进步，我们都知道传统的服务器端脚本语言(如ASP、PHP等)的标签都是固定的，程序员只能使用这些固定的标签进行程序设计。

虽然固定的标签功能对于程序设计来说是足够的，但是如果能够让程序员将一些专用的复杂功能定义成自己的标签，保存在标签库中，在以后的Web程序设计中使用，将极大简化动态网页的设计，而且标签库可以被重用，开发其他程序时只要带上这个标签库就可以在程序中使用这些自己定义的标签，同样也是将极大提高程序开发的效率。



其实定制标签和使用JavaBean的目标是一致的，就是要把复杂的功能或行为封装成简单易用的形式，从而方便JSP的动态网页开发，但是标签库同JavaBean还是存在许多不同点：

(1) bean不能处理JSP程序本身的内容，而标签库可以

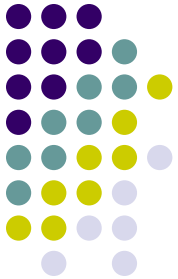
(2) 使用标签库封装服务器行为的效果往往比bean好，封装结果是同其他JSP标准预定义的标签类似的标签，使用起来比bean更方便自然

(3) 定制标签的重用比bean更方便些

(4) bean更多地用来实现应用程序的业务逻辑，或在应用程序中的不同部分之间进行消息传递，而标签库用来定义一些在一个JSP程序内使用的自包含的行为

(5) 标签库只是在JSP1.1之后才支持的，而bean则是在JSP1.0就开始支持了。

这里对两者的比较并不意味着那一个会取代另一个，两者从应用场合来讲是基本不同的，bean主要用来实现业务逻辑的封装，而定制标签则是用来实现页面表现逻辑的封装。



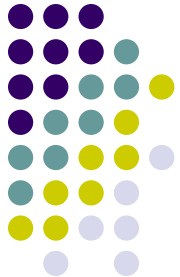
2、使用定制标签库

要使用定制标签库，JSP程序员需要设计3个部分的代码：使用Java语言编写定义标签行为的标签处理类文件、用来将XML元素映射到标签实现的标签库描述文件、以及在JSP程序中使用标签库的代码，以下就来介绍如何定义并使用自定义的标签库：

1、标签处理器

创建定制标签库的第一个工作就是定义描述系统如何处理标签的Java类，这个类就是标签处理类，也称为标签处理器。

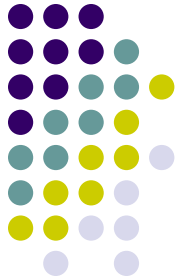
简单地说，标签处理器是一个服务器端的运行时对象，它用来在JSP页面执行时解释标签的动作，通过支持一种运行时协议来实现同JSP页面的交互。



而要从实现角度来说，它则是一个服务器端的不可见的JavaBean构件，但它实现了一些更复杂的接口，从而支持同JSP页面的通信协议，它可以通过javax.servlet.jsp.tagext.Tag接口或javax.servlet.jsp.tagext.BodyTag接口定义。

(1) Tag接口：这个接口定义了所有标签处理器都需要的基本方法，其中包括初始化标签处理器的方法以及两个重要的方法doStartTag()和doEndTag()，这个接口可以实现不需要标签内容的简单标签

(2) BodyTag接口：如果要实现更复杂的可以处理标签体内容的标签，就需要实现此接口，它扩充了两个方法：doInitBody()和doAfterBody()，用来实现对标签内容的处理。

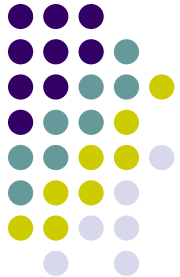


另外，在实现标签处理器时，还可以使用另外的三个支持类，分别是：

(1) TagSupport类：它可以用作实现Tag接口的标签处理器的基类。它实现了Tag接口并扩充了新的方法，以便创建更复杂的标签，它包括一个获取Tag属性的getter方法，以及一个静态方法findAncestorWithClass(Tag,class)，它可以用来实现标签之间的协同。

(2) BodyTagSupport类：它可以用作实现BodyTag接口的标签处理器的基类，它实现了BodyTag接口，并扩充了新的方法，如获取BodyTag的属性的getter方法以及在JspWriter之前取信息的方法。

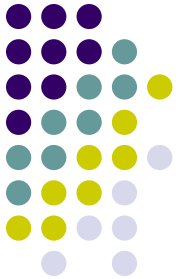
(3) BodyContent类：它是JspWriter的子类，可以用来处理页面的内容。它可以将页面信息转换成字符串，读取内容以及清除内容。



2、标签库描述文件

除了实现标签功能的标签处理器，我们还需要定义标签库描述文件(Tag Library Descriptors)，简称TLD文件。

TLD文件是一个XML文档，它包含了关于整个标签库以及库中的每一个标签的信息，JSP容器使用TLD文件来验证JSP页面中标签的使用是否正确。



下面的TLD元素用来定义一个标签库。

```
<taglib>
    <tlibversion>标签库的版本</tlibversion>
    <jspversion>JSP版本</jspversion>
    <shortname>可供JSP页面使用的短名</shortname>
    <uri>标示标签库的URI</uri>
    <info>关于标签库的描述信息</info>
    <tag>标签信息定义</tag>
</taglib>
```

对于标签库中的每个标签，TLD文件在<tag>元素和</tag>元素之间定义其内容，其中最重要的信息包括：

(1) name：标签的基本名称

(2) tagclass：指定标签处理器类

(3) info：描述标签的简短信息

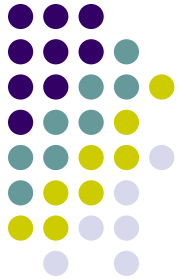
(4) bodycontent：描述标签体的类型，如果标签不含标签体，则值为EMPTY，如果标签体按照普通的JSP解释，则值为JSP，如果标签体需要被处理，则值为TAGDEPENDENT。

(5) attribute：定义标签包含的属性，注意这个元素又包含3个嵌套的元素：

name:属性名

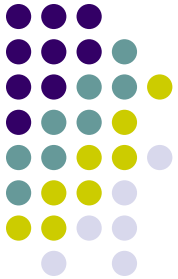
required:说明该属性是否是必须提供的(true/false)

rtexprvalue:说明赋给属性的值是否可以是一个JSP表达式(true)或者只能是固定值(false)



标签信息定义的结构：

```
<tag>
    <name>标签名称</name>
    <tagclass>标签处理器类名</tagclass>
    <info>描述信息</info>
    <bodycontent>标签体类型说明</bodycontent>
    <attribute>
        <name>属性名</name>
        <required>true/false</required>
        <rtexpvalue>true/false</rtexpvalue>
    </attribute>
</tag>
```



3、标签库在JSP页面中的使用

在实现了标签处理器类，并定义了TLD文件之后，我们就可以在JSP页面中使用定义的标签了。在使用标签之前，需要使用JSP的taglib编译指示说明标签库，其语法如下：

<%@ taglib uri=“URLToTagLibrary” prefix=“tagPrefix”%>。

其中uri是映射到标签库TLD文件的URI，它在Java Web程序的部署文件中定义。Prefix设定了在JSP页面中使用标签库的标签时加在标签前的前缀符号，形如<prefix:tag>的形式。

3、标签的分类与实现

在JSP程序设计中，最常用的标签有4种，分别是简单标签、有标签体的标签、协作标签以及创建脚本变量的标签。

1、简单标签

经常使用的简单标签是不含标签体的标签，即在起始标签和结束标签之间不含任何文本信息的标签，例如<prefix:name attribute1=“...”/>

这种标签只要使用标签处理器的doStartTag()方法就可以实现，当JSP容器处理到开始标签符号时，就会调用doStartTag()方法，程序员可以在此方法中获取标签的属性，进行各种处理然后返回常数SKIP_BODY通知系统忽略标签的起始标志和结束标志之间的任何内容。

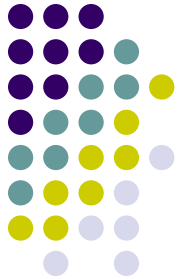
标签处理器可以直接实现Tag接口，而更一般的是继承TagSupport类。

如果标签包含属性，则在标签处理器中要为每一个属性实现一个setter方法，它一般都是将属性值存储在某个变量中，之后可以在doStartTag方法中使用这个属性。如果想标签处理器可以被别的类访问，需要再实现getter方法。

如果要向页面输出信息，doStartTag方法中首先可以通过pageContext对象的getOut方法来获得一个JspWriter对象，接着使用JspWriter对象的print或println方法进行输出，在输出时可能会抛出IOException异常，需要把print语句包含在try/catch结构中。

注意这里的pageContext对象同JSP页面中的PageContext对象是同一个对象，我们可以通过这个对象获得很多有用的对象和信息，如：getRequest、getResponse、getservletContext、getSession等。

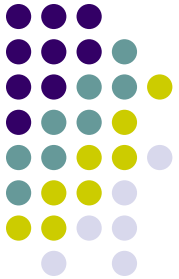
一般情况下简单标签的处理基本上在doStartTag方法中完成了，所以不需要再在doEndTag方法中实现处理代码，但是有时候我们可以通过此方法控制系统在处理完标签后时候是否还处理JSP页面的剩余部分。如果doEndTag方法返回SKIP_PAGE就不处理页面的剩余部分，如果返回EVAL_PAGE系统继续处理剩余的JSP页面。



一个简单的标签处理器，它实现标签<mytag:CurrentTime/>，该标签在页面中插入当前的日期显示。具体代码如下：

1、CurrentTimeTag.java

```
package mytag;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
```



```
public class CurrentTimeTag extends TagSupport{
    public int doStartTag(){
    try{ JspWriter out=pageContext.getOut();
                                    java.util.Date dt=new
    java.util.Date(System.currentTimeMillis());
    out.print(dt.getHours()+" :" +dt.getMinutes()+" :" +dt.get
    Seconds());}
    catch(IOException ioe)
    {System.out.println( "Error in CurrentTime Tag:" +ioe);}
    return(SKIP_BODY);
}
}
```

2、配置文件：my-taglib.tld

<?xml version=“1.0” encoding=“ISO-8859-1”?>

<!DOCTYPE taglib PUBLIC “-//Sun Microsystems,Inc.//DTD JSP Tag Library 1.1//EN”

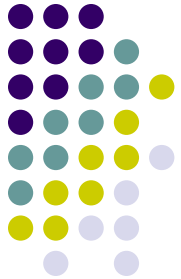
http://java.sun.com/j2ee/dtds/Web-jsptaglibrary_1_1.dtd” >

<!--a tag library descriptor-->

<taglib>

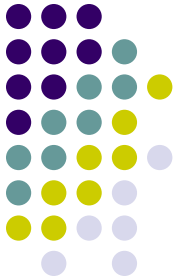
<!—after this the default space is http://java.sun.com/j2ee/dtds/jsptaglibrary_1-2.dtd-->

```
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>current time</shortname>
    <uri></uri>
    <info>A tag library for show current time</info>
    <tag>
        <name>CurrentTime</name>
        <tagclass>mytag.CurrentTimeTag</tagclass>
        <bodycontent>EMPTY</bodycontent>
    </tag>
</taglib>
```



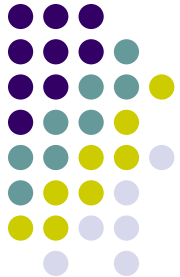
3、web.xml配置文件

```
<?xml version=“1.0” encoding=“gb2312”?>
<!DOCTYPE Web-app PUBLIC “-//Sun
    Microsystems,Inc.//DTD Web Application 2.2//EN”
     “http://java.sun.com/j2ee/dtds/Web_app_2_2.dtd” >
<Web-app>
    <taglib>
        <taglib-uri>my-taglib</taglib-uri>
        <taglib-location>/WEB-INF/jsp/my-taglib.tld</tag-
    location>
    </taglib>
</web-app>
```



4、CurrentTime.jsp

```
<%@ taglib uri=“my-taglib” prefix=“mytag” %>
<HTML>
<HEAD><TITLE>显示当前时间-JSP/TagLib</TITLE></HEAD>
<BODY>
<H1>显示当前时间-JSP/TagLib</H1>
<mytag:CurrentTime/>
</BODY>
</HTML>
```

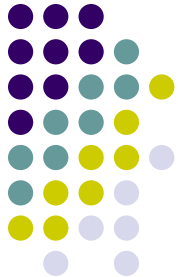


2、具有标签体的标签

通常情况下，我们使用定制标签来封装的服务器行为可能比较复杂，这时就需要使用具有标签体的标签。标签体就是标签的起始标志和结束标志的内容，它可以包括任何有效的JSP的语言单元，就像JSP页面的其他部分一样，系统会将标签体的内容翻译成servlet代码，当页面被请求时执行这些代码，比如：

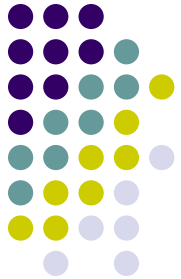
<prefix:tagname>body</prefix:tagname>

如果要解释标签体，则标签处理器的doStartTag方法需要返回EVAL_BODY_TAG，这样系统才会处理标签体。处理需要用到另外两个方法：doInitBody()和doAfterBody()，它们在BODYTag接口中定义，因此具有标签体的标签处理器应实现BODYTag接口，同样我们通过继承BodyTagSupport 类来实现标签处理器，这样要比直接实现BodyTag接口要容易一些。



当doStartTag方法执行结束(返回EVAL_BODY_TAG)，系统就创建一个BodyContent类型的嵌套流，并通过setBodyContent方法传递给BodyTag对象，然后系统调用doInitBody方法，程序员可以在这个方法中做初始化的工作。之后标签体的内容就被系统处理，保存到一个新的BodyContent对象中，最后系统再调用doAfterBody方法处理标签体，例如可以进行标签体的字符串内容的过滤、修改内容等操作。

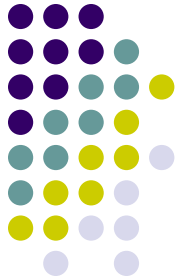
在doAfterBody方法中，我们可以使用getBodyContent方法获得BodyContent对象，而BodyContent类的方法可以用来方便的处理标签的内容，主要的方法有：getEnclosingWriter():返回doStartTag、doEndTag方法使用的JspWriter对象
getReader():返回一个Reader对象，可以用来读取标签内容
getString():将整个标签体作为一个字符串返回



带有标签体标签的示例：for循环结构标签

1、ForTag.java

```
package structtags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class ForTag extends BodyTagSupport{
  private int index;
  private int start;
  private int end;
  private int step;
  private boolean decrease;
```



```
 public void setStart(int i){
        start=i;
        index=start;
}
public void setEnd(int i){
        end=i;
}
public void setStep(int i){
        step=i;
        if(step>0)
          decrease=false;
        else
       decrease=true;
}
```

```
public int doStartTag() throws JspException{
        return EVAL_BODY_TAG;
}
public void doInitBody() throws JspException{
        pageContext.setAttribute(  “ loopIndex ”  , (new
Integer(index)));
}
public int doAfterBody(){
        BodyContent body=getBodyContent();
        try{
           JspWriter
         out=body.getEnclosingWriter();
            out.println(body.getString());
           body.clearBody();
```

```
    }
  catch(IOException ioe){
      System.out.println( “Error in ForTag:” +ioe);
    }
  index=index+step;
  if(index<end)^decrease){
      pageContext.setAttribute( “ lookIndex ” ,(new
   Integer(index)));
     return(EVAL_BODY_TAG);
   }else{ return(SKIP_BODY);}
}
}
```

for循环标签的结构如下：

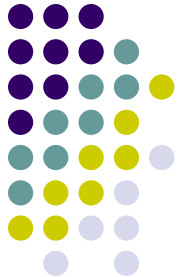
<struct:For start=“…” end=“…” step=“…” >

……

</struct:For>

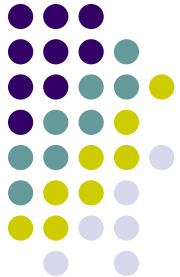
For循环标签的标签处理类ForTag通过扩展BodyTagSupport类来实现，for标签具有以下属性：start(循环的起始计数);end(循环的结束计数)step(循环计数步长，正值或负值)

ForTag类中为每一个属性实现了一个setter访问方法，它还实现了接口定义的另外3个方法：doStartTag、doInitBody和doAfterBody



doStartTag仅含一条语句，返回值EVAL_BODY_TAG，指示系统继续处理标签体内容。doInitBody设置循环指针变量loopindex，它是一个脚本变量，JSP程序中可以使用这个变量获得当前循环指针值。

对于标签体处理的代码在doAfterBody方法中定义。它取出标签体的内容，按后输出，最后根据循环条件决定是否再次重复处理标签体，如果需要再次处理返回EVAL_BODY_TAG，否则就返回SKIP_BODY。



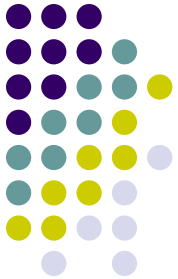
For循环标签的处理类的完整定义还需要另外的一个类ForTagExtraInfo，它是在ForTagExtraInfo.java文件中定义的，这个类也是用于定义脚本变量的，我们将在后面关于创建脚本变量的标签时再作讲解。

3、协作标签

有些服务器端的行为比较复杂，需要通过多个标签的协作才能实现，这种标签称之为协作标签，它又两种方法：隐含的协作和显示的协作

(1) 隐含协作的标签

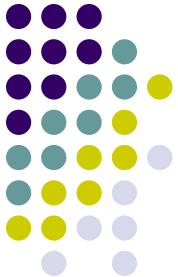
隐含协作的标签之间需要某种固定的结构，这种结构隐含了标签之间的关系，最典型的例子就是嵌套标签，即某些标签是嵌套在另外的标签中使用的。



(2) 显示协作的标签

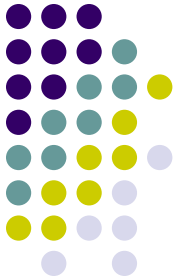
显示的标签协作就需要通过定义脚本变量来实现，其中一个标签可以定义一个或多个脚本变量，就像在JSP的<jsp:useBean>中定义脚本变量一样，由于脚本变量在整个JSP页面内都可以被访问，所以另一个标签可以访问这个变量。

通过脚本变量传递信息，从而实现了标签的协作。通过这种方式协作的标签不需要具有嵌套标签那样的固定结构，它们可以位于JSP页面的任何位置。另外定义具有显示协作关系的标签关键是实现能够创建脚本变量的标签。



4、创建脚本变量的标签

定制标签可以创建服务器端的对象，并给这个对象一个脚本变量名，使它可以被JSP页面使用。创建脚本变量的标签的标签处理器需要用javax.servlet.jsp.tagext.TagExtranInfo类，这个类中包含方法用来指定对象的脚本变量名和类型。

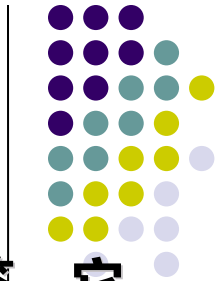


标签处理器所需要做的工作包括：

(1)在TagExtraInfo类中记录脚本变量的名称和类型；

(2) 创建所定义的对象，并把它加入到pageContext对象中。

至于脚本变量同对象之间的对应关系则是系统的JSP页面翻译器的职责了。



前面介绍的For循环标签就是一个定义脚本变量的标签，它定义了一个Integer类型的脚本变量Loopindex。

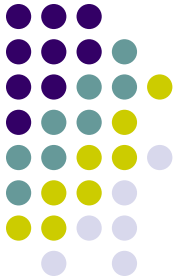
ForTagExtraInfo.java

```
package structtags;
import javax.servlet.jsp.tagext.*;
public class ForTagExtranInfo extends TagExtraInfo{
    public VariableInfo[] getVariableInfo(TagData data){
   return new VariableInfo[]
```

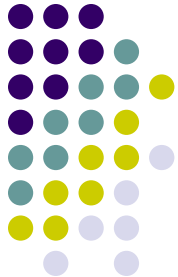
```
    {
      new
    VariableInfo(“loopIndex”,”Integer”,true,VariableInfo.
    NESTED)
    };}
}
```

这个类说明了定义的变量的信息，包括变量名称、类型以及有效范围等，注意定义的变量都是对象，例如For标签中定义的loopIndex就是一个Integer对象，而不是int类型。

JSP/Servlet联合编程

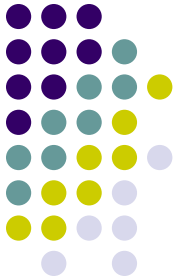


- MVC模型简介
- JSP/Servlet设计模型
- JSP/Servlet联合编程示例



1、 MVC模型简介

对任意的应用程序，都需要有对程序流程的控制，将这些控制进行抽象就可以总结出某类应用或在某种编程模式下常用的设计模型。在面向对象程序设计中，“模型-视图-控制”模型(简称MVC) 变得越来越流行，这种模型也正是Java Web服务器应用程序设计的主要设计模型。



在MVC模型中，应用程序被分解成独立的3个部分：

模型：即程序将要实现的业务逻辑

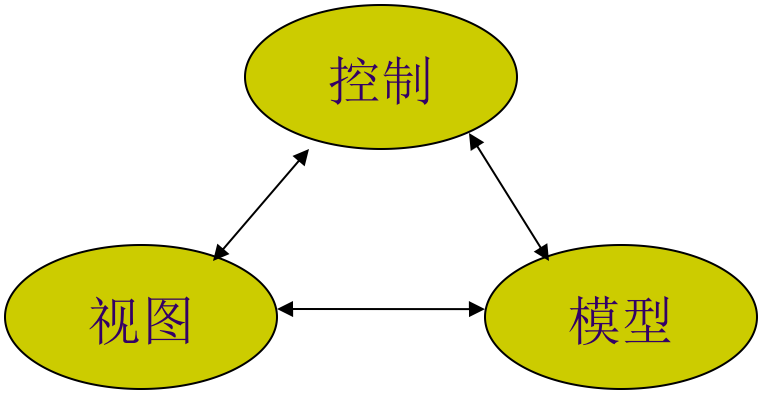
视图：又称为用户界面

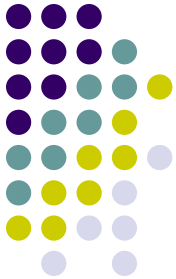
控制：用来管理模型和视图之间交互以及应用程序的流程的构件。

在早期的MVC模型中，控制主要是管理用户的输入，现在的MVC模型中控制则更多的是负责程序的流程。

使用MVC模型可以描述绝大多数的Web应用的设计，它也十分适合于进行面向对象的分析、设计和编程，因此许多采用java技术的组织都推荐其作为Web应用的主要设计模型。

下面图描述了MVC模型：



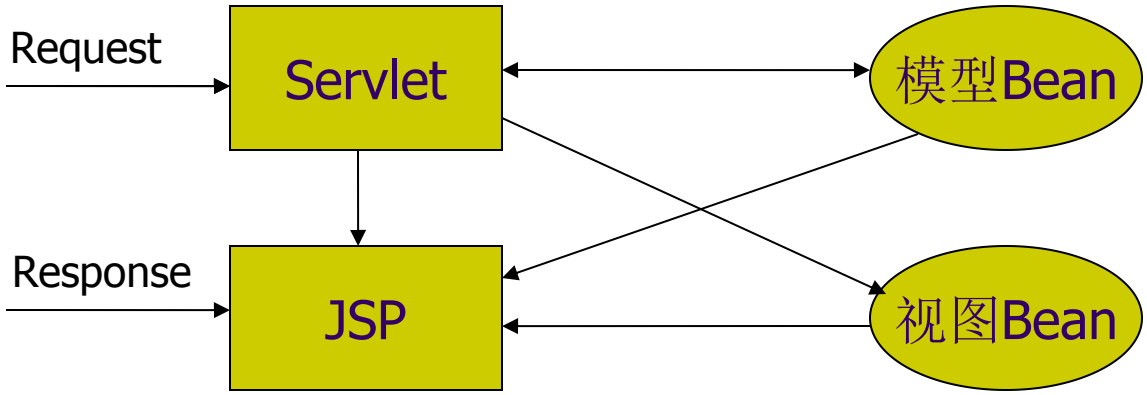


需要注意的是，MVC模型只描述了一种面向对象的应用程序设计模型，当我们按照这个模型来设计应用程序是，可以采用各种不同的实现技术，这就产生了不同的具体设计模型。具体到Java Web应用程序设计这个领域，我们可以完全采用Servlet技术或者JSP技术实现MVC模型的视图和控制，而采用JavaBean等后台技术实现其模型部分。这也是我们前面介绍的简单2层模型。

而在JSP/Servlet设计模型下，JSP主要用来实现MVC模型的视图部分，Servlet用来实现模型的控制部分，而JavaBean等后台技术实现模型部分。

2、JSP/Servlet设计模型

在JSP/Servlet设计模型中，Servlet用来实现程序的控制部分，JSP实现程序的视图部分，JavaBean 封装程序的模型部分，当然它可能会通过JDBC来访问后台的数据库系统。以下是JSP/Servlet设计模型的示图：



用户浏览器发出的访问请求都发送到Servlet统一处理，Servlet解释请求信息，进行相应处理，如果需要同实现业务逻辑的模型部分交互则同相应的模型Bean交互，这些模型Bean封装了程序的业务逻辑，将后台的模型同前台的控制与视图分离开，如果业务逻辑比较复杂，模型Bean可能还需要同其他的后台系统交互，例如数据库、EJB等，最后，Servlet调用JSP程序显示相应的页面。

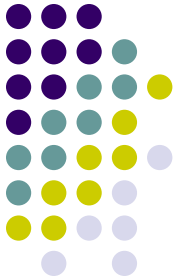
当JSP的显示需要获取模型数据时，可以有两种方法。一种方法是允许JSP程序访问模型Bean，另外一种方法是Servlet根据模型的交互，将需要显示的结果数据封装到另外的用于显示的Bean中，这些Bean称之为视图Bean，JSP不需要访问模型Bean,而只需要访问视图Bean来获取数据，和Web页面结合生成最终的动态页面显示。

JSP/Servlet设计模式增加了两个层次。一个层次就是Servlet层，Servlet和JSP进行分工，这种划分使得程序的视图部分和控制部分实现了分离，这使得两者可以独立开发，提高了开发的效率。

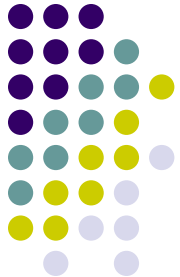
另外一层是对JavaBean进行划分，划分成封装程序的模型部分的模型Bean和用来在控制部分和视图部分传递数据的视图Bean。前者对应于模型中的不同实体，既封装了实体的数据也封装了实体的功能，后者则只封装了数据，这样使得应用的模型部分和视图部分实现了彻底的分离。

那么视图Bean在模型的控制部分(Servlet)和视图部分(JSP)之间的传递方式也有两种。

一种是将视图Bean放入session中传递，这种情况下一般都是通过forward的形式调用JSP页面，这种方式比较简单直观，但是缺点是session会占用不少的空间，这种资源消耗对于访问量不大的Web应用系统不成问题，但是如果Web应用系统的访问量很大情况下就会影响到系统的效率。



另外一种方式就是通过request对象传递，这种情况下一般采用include的方式调用JSP页面，它也会使用到session，但是session的消耗的资源不大，因为这种方式可以支持更多的访问量，但是程序的控制就要复杂一些，因为每次产生新的Request对象，而且session对象则可以在多个页面访问之间保持连续，可能在某些应用中不能满足要求，仍需要通过session传递。



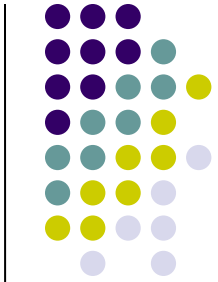
3、JSP/Servlet联合编程示例

一个简化的软件网上超市系统，那么它Web前台所需要的核心功能有：软件的分类、软件列表、软件内容介绍、会员登陆、购物推车以及一个示意性的订单提交功能。

那么在一个非常核心的部分购物推车：主要包括会员购买的软件首先放置在购物推车中，会员可以修改推车中软件的数量、删除推车中的软件商品、查阅推车中的商品以及进行各种推车管理的功能。

在一种使用JSP/JavaBean/JDBC具体的实现模式下，购物推车管理功能页面由cart.jsp和cartcontent.jsp程序实现，其中cart.jsp实现推车管理的程序控制，而cartcontent.jsp实现购物推车管理的页面显示。它们都使用购物推车bean——Cart和item实现推车的业务逻辑。

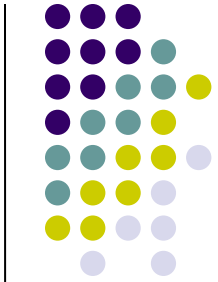
以下是使用JSP/JavaBean/JDBC实现模式的cart.jsp的代码：



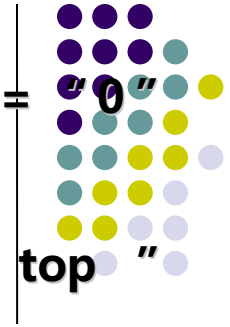
1、cart.jsp

```
<jsp:useBean id= “user” class= “User” scope= “session” />
<%if(!user.isLogin()) {%>
<jsp:forward page= “login.jsp” />
<%}
String action=request.getParameter( “action” )){
if(action!=null){
  if(action.equals(” addcart” )){
    int PID=new Integer(request.getParameter( “PID” )).intValue();
    int Qty=new Integer(request.getParameter( “Qty” )).intValue()
int Price=new Integer(request.getParameter( “Price” )).intValuye()
   String Pname=request.getParameter( “Pname” );
   user.getCart().addCartItem(PID,Qty,Price,Pname);
    }else if(action.equals( “setcart” )){
```

```
int index=new Integer(request.getParameter( “index” )).intValue();
int Qty=new Integer(request.getParameter( “Qty” )).intValue();
 user.getCart().setCartItem(index,Qty);
 }else if(action.equals( “delcart” )){
int index=new Integer(request.getParameter( “index” )).intValue();
 user.getCart().delCartItem(index);
 }
}%>
<%@page contentType= “text/html;charset=gb2312” %>
<html>
<jsp:include page= “head.jsp” flush= “true” />
<body>
<jsp:include page= “title.jsp” flush= “true” />
```

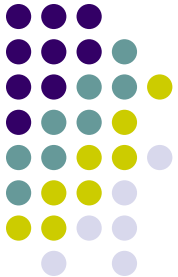


```
<div align= “center” >
<table border= “0” width= “694” height= “189” cellspacing= “0”
   cellpadding= “0” >
  <tr>
    <td width= “101” height= “189” valign= “top”
    bgcolor= “#eaebe2” >
    <jsp:include page= “left.jsp” flush= “true” />
    </td>
    <td width= “593” height= “189” valign= “top” >
        <jsp:include page= “cartcontent.jsp” flush= “true” />
    </td>
    </tr>
</table>
</div>
<jsp:include page= “foot.jsp” flush= “true” />
</body></html>
```



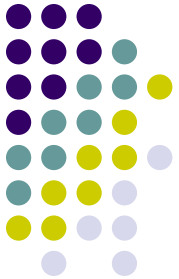
在这个例子中，虽然Cart实现了购物推车的业务逻辑，封装了推车数据，但是由于推车的管理比较复杂，包含多个功能，因此仍需要由cart.jsp程序实现推车业务逻辑操作的控制。即根据用户的不同请求，进行不同的推车管理操作。

因此这里我们根据JSP/Servlet的设计模型对原有的cart.jsp作一个改进，用Servlet类的CartServlet代替cart.jsp实现这个控制逻辑，而另外编写一个cart1.jsp程序，它只需要完成调用页面的各显示程序显示管理页面的内容就可以了。



那么在使用JSP/Servlet设计模型改进以后，我们就可以看到使用cart1.jsp来显示购物推车的管理页面，而使用CartServlet来实现根据用户请求进行的推车管理操作的控制部分，而推车本身的业务逻辑则是在购物推车实体的JavaBean中来实现的，可见模型层次非常清晰。

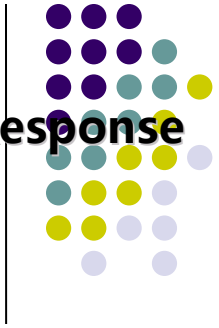
以下是CartServlet和cart1.jsp的代码：

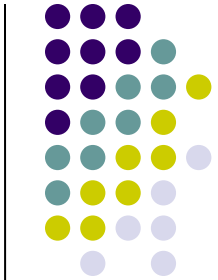


1、CartServlet.java

```
import java.util.*;
import java.io.*;
import java.sql.*;
import javax.Servlet.*;
import javax.Servlet.http.*;
import item;
import Cart;
import User;
public class CartServlet extends HttpServlet{
//Servlet的初始化
 public void init(ServletConfig conf) throws ServletException{
   super.init(conf);
 }
```

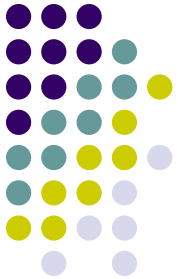
```
//处理post请求
public void doPost(HttpServletRequest request,HttpServletResponse
  response) throws ServletExcetpion,IOException {
    performTask(request,respose);
 }
//处理get请求
public void doGet(HttpServletRequest request,HttpServletResponse
  response) throws ServletExcetpion,IOException {
    performTask(request,respose);
 }
//实际处理
public void performTask(HttpServletRequest request,HttpServletResponse
  responsethrows ServletExcetpion,IOException {
```





```
HttpSession session=request.getSession(false);
User user=(User)session.getAttribute( “user” );
if((user==null)||(!user.isLogin())){
  getServletConfig().getServletContext().
getRequestDispatcher( “/login.jsp” ).forward(request,response);
  return;
 }
String action=request.getParameter( “action” );
if(action!=null){
  if(action.equals( “addcart” )){
 int PID=new Integer(request.getParameter( “PID” )).intValue();
 int Qty=new Integer(request.getParameter( “Qty” )).intValue()
int Price=new Integer(request.getParameter( “Price” )).intValuye()
    String Pname=request.getParameter( “PName” );
    user.getCart.addCartItem(PID,Qty,Price,Pname);
```

```
   }else if(action.equals( “setcart” )){
int index=new Integer(request.getParameter( “index” )).intValue();
int Qty=new Integer(request.getParameter( “Qty” )).intValue();
user.getCart().setCartItem(index,Qty);
  }else if(action.equals( “delcart” )){
int index=new Integer(request.getParameter( “index” )).intValue();
user.getCart().delCartItem(index);
}
}
Session.setAttribute( “user” ,user);
getServletConfig().getServletContext().getRequestDispatcher( “/cart1.jsp” ).forw
    ard(request,response);
}
}
```



2、cart1.jsp

```
<%@ page contentType= “text/html;charset=gb2312” %>
<html><jsp:include page= “head.jsp” flush= “true” />
<body><jsp:include page= “title.jsp” flush= “true” />
<div align= “center” >
<table border= “0” width= “694” height= “189” cellspacing= “0” cellpadding= “0” >
<tr>
<td width= “101” height= “189” valign= “top” bgcolor= “#eaebe2” >
<jsp:include page= “left.jsp” flush= “true” /></td>
<td width= “593” height= “189” valign= “top” >
<jsp:include page= “ cartcontent.jsp ” flush= “ true ” /></td>
</tr>
</table>
</div>
<jsp:include page= “foot.jsp” flush= “true” />
</body></html>
```