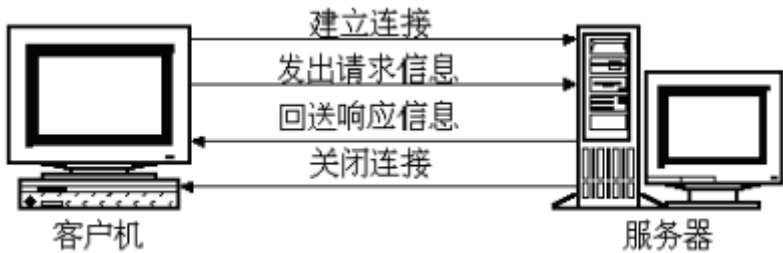


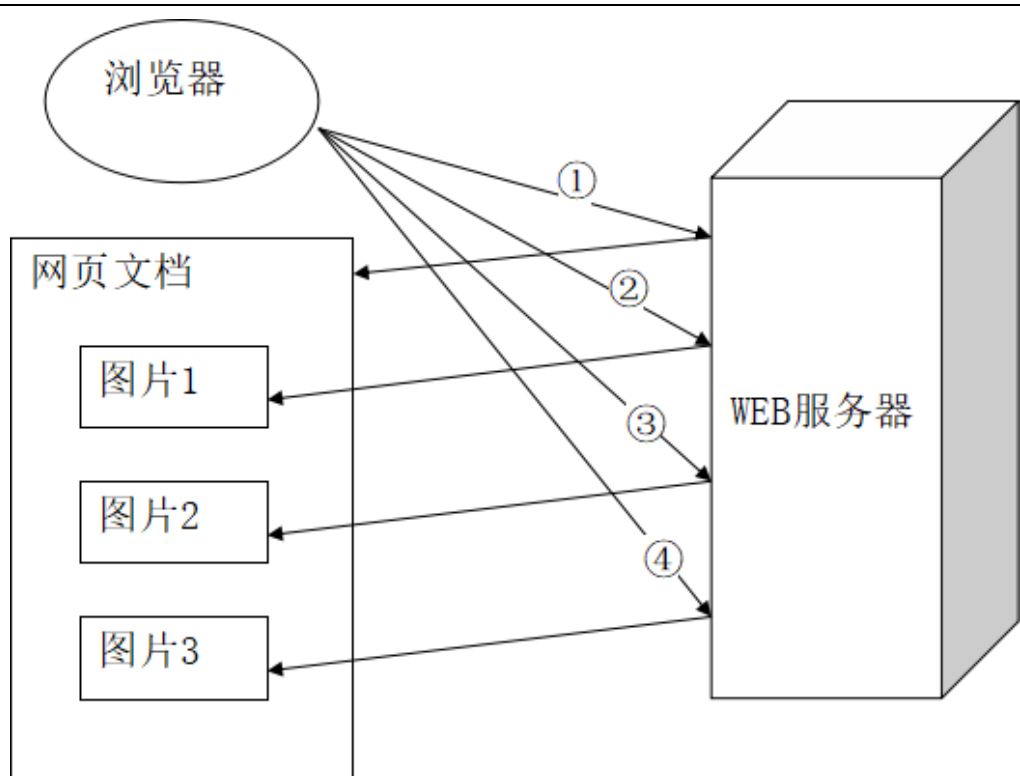
## 东软睿道教案（首页）

TTC:

课程/项目名称	Javaweb	授课教师	于洪超	总学时：60 学时
教学目的和要求	了解 MVC 思想、掌握 Servlet、 JSP 及过滤器、监听器、ajax 请求等内容。			
教学重点、难点	重点： servlet，jsp，过滤器，监听器， 难点： 过滤器，监听器			
教学资源	教材及参考资料： 教材： 内部课件			
	其他教学资源：			
教学环境				

## 东软睿道教案

章节	第 1 章 web 应用开发入门						
教学目标与要求	了解web起源，web与web应用，Web应用开发技术介绍，Web应用工作原理，URL与URI区别，HTTP协议简介，Web服务器简介，Tomcat服务器安装和配置，JavaWeb常见应用						
重点难点	重点： Tomcat服务器安装和配置 难点： Tomcat服务器安装和配置						
教学内容设计及学时分配	<p><b>【教学进程安排】</b></p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: left;">Http</td><td style="text-align: right;">2 课时</td></tr> <tr> <td style="text-align: left;">Tomcat 服务器安装和配置</td><td style="text-align: right;">4 课时</td></tr> <tr> <td style="text-align: left;">JavaWeb 应用</td><td style="text-align: right;">2 课时</td></tr> </table> <p><b>【课程主要内容】</b></p> <h3>1. HTTP 简介</h3> <p>WEB 浏览器与 WEB 服务器之间的一问一答的交互过程必须遵循一定的规则，这个规则就是 HTTP 协议。</p> <p>HTTP 是 hypertext transfer protocol（超文本传输协议）的简写，它是 TCP/IP 协议集中的一个应用层协议，用于定义 WEB 浏览器与 WEB 服务器之间交换数据的过程以及数据本身的格式。</p> <h3>2. HTTP 的会话方式</h3> <p>四个步骤：</p> <div style="text-align: center;">  <pre> sequenceDiagram     participant Client as 客户机     participant Server as 服务器     Client-&gt;&gt;Server: 建立连接     Client-&gt;&gt;Server: 发出请求信息     Server--&gt;&gt;Client: 回送响应信息     Server-&gt;&gt;Client: 关闭连接             </pre> </div> <p>浏览器与 WEB 服务器的连接过程是短暂的，每次连接只处理一个请求和响应。对每一个页面的访问，浏览器与 WEB 服务器都要建立一次单独的连接。</p> <p>浏览器到 WEB 服务器之间的所有通讯都是完全独立分开的请求和响应对。</p>	Http	2 课时	Tomcat 服务器安装和配置	4 课时	JavaWeb 应用	2 课时
Http	2 课时						
Tomcat 服务器安装和配置	4 课时						
JavaWeb 应用	2 课时						



### 3.HTTP 请求消息（了解即可）

请求消息的结构：

一个请求行、若干消息头、以及实体内容，其中的一些消息头和实体内容都是可选的，消息头和实体内容之间要用空行隔开。

#### • 举例：

```

GET /books/java.html HTTP/1.1
Accept: */*
Accept-Language: en-us
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/links.asp
User-Agent: Mozilla/4.0
Accept-Encoding: gzip, deflate

```

←请求行

←多个消息头

←一个空行

### 4.HTTP 响应消息（了解即可）

响应消息的结构：

一个状态行、若干消息头、以及实体内容，其中的一些消息头和实体内容

容都是可选的，消息头和实体内容之间要用空行隔开。

- 举例：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 13 Jul 2000 05:46:53 GMT
Content-Length: 2291
Content-Type: text/html
Cache-control: private

<HTML>
<BODY>
.....
```

← 状态行

← 多个消息头

← 一个空行

← 实体内容

这里举个小实例：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <form action="/loginServlet" method="post">
        user:<input type="text" name="user">
        password:<input type="password" name="password">
        <input type="submit" value="Submit">
    </form>

</body>
</html>
```

运行服务器，在浏览器中打开这个网页，开发者工具，提交表单，提示 404（申请资源不存在），在 chrome network 中查看请求及响应消息。

## 5.url 与 uri 区别

URI 是统一资源标识符，而 URL 是统一资源定位符。因此，笼统地说，每个 URL 都是 URI，但不一定每个 URI 都是 URL。这是因为 URI 还包括一个子类，即统一资源名称 (URN)，它命名资源但不指定如何定位资源。上面的 mailto、news 和 isbn URI 都是 URN 的示例。

## 6 应用框架介绍 b/s 与 c/s 架构

硬件环境不同:

C/S 一般建立在专用的网络上, 小范围里的网络环境, 局域网之间再通过专门服务器提供连接和数据交换服务. B/S 建立在广域网之上的, 不必是专门的网络硬件环境, 例与电话上网, 租用设备. 信息自己管理. 有比 C/S 更强的适应范围, 一般只要有操作系统和浏览器就行

对安全要求不同 :

C/S 一般面向相对固定的用户群, 对信息安全的控制能力很强. 一般高度机密的信息系统采用 C/S 结构适宜. 可以通过 B/S 发布部分可公开信息.

B/S 建立在广域网之上, 对安全的控制能力相对弱, 面向是不可知的用户群.

对程序架构不同:

C/S 程序可以更加注重流程, 可以对权限多层次校验, 对系统运行速度可以较少考虑.

B/S 对安全以及访问速度的多重的考虑, 建立在需要更加优化的基础之上. 比 C/S 有更高的要求 B/S 结构的程序架构是发展的趋势, 从 MS 的 .Net 系列的 BizTalk 2000 Exchange 2000 等, 全面支持网络的构件搭建的系统. SUN 和 IBM 推的 JavaBean 构件技术等, 使 B/S 更加成熟.

软件重用不同:

C/S 程序可以不可避免的整体性考虑, 构件的重用性不如在 B/S 要求下的构件的重用性好.

B/S 对的多重结构, 要求构件相对独立的功能. 能够相对较好的重用. 就入买来的餐桌可以再利用, 而不是做在墙上的石头桌子

系统维护不同 :

系统维护是软件生存周期中, 开销大, -----重要

C/S 程序由于整体性, 必须整体考察, 处理出现的问题以及系统升级. 升级难. 可能是再做一个全新的系统

B/S 构件组成, 方面构件个别的更换, 实现系统的无缝升级. 系统维护开销减到最小. 用户从网上自己下载安装就可以实现升级.

处理问题不同:

C/S 程序可以处理用户面固定, 并且在相同区域, 安全要求高需求, 与操作系统相关. 应该都是相同的系统

B/S 建立在广域网上, 面向不同的用户群, 分散地域, 这是 C/S 无法作到的. 与操作系统平台关系最小.

用户接口不同

C/S 多是建立的 Window 平台上, 表现方法有限, 对程序员普遍要求较高

B/S 建立在浏览器上, 有更加丰富和生动的表现方式与用户交流. 并且大部分难度减低, 减低开发成本.

## 7. JavaWeb 应用部署

Tomcat 作为 Servlet/JSP 容器(服务器)挺不错的, 开源免费, 需要知道的是 Tomcat 是一个 Web 服务器, 其符合 Servlet/JSP 规范, 但是却没有实现所有 JavaEE 规范, 所以我们还是应该规范说法, 称 Tomcat 为一个 JavaWeb 服务器, 而不是 JavaEE 服

务器

我们使用 IDE 部署 JavaWeb 项目基本上都是一键完成,这是 IDE 给我们该来的好处,但是还是有必要了解一下部署项目的细节

要将 JavaWeb 应用部署到 Tomcat, 需要显式或隐式定义一个 Tomcat 上下文。在 Tomcat 中, 每一个 Tomcat 上下文都表示一个 Web 应用程序。所以我们也分为隐式和显式两种方式进行总结

显式部署

显式部署总的来说就是以目录结构的方式部署, 不打包, 只要项目根目录 (Maven 为 webapp 普通项目为 webRoot/webContent) 有 WEB-INF (WEB-INF 下有 classes, jar 包, 配置文件), 有静态资源 (HTML, CSS, JS, 图片等) 即可。再通过一个在 Tomcat 中的配置文件来将我们的程序发布出来。

显式部署有两种方式:

在 Tomcat 的 conf/Catalina/localhost 目录下创建一个 XML 文件

在 Tomcat 的 conf/server.xml 文件中添加一个 Context 元素

方式一

如果决定给每一个上下文都创建一个 XML 文件 (第一种方式), 那么这文件名就很重要了, 因为上下文路径就是从这个文件名的衍生得到的。例如, 将一个 Demo1.xml 文件放在 conf/Catalina/localhost 目录下, 那么这个应用程序的上下文路径就是 Demo1, 访问的 URL 就为: http://localhost:8080/Demo1

这个上下文文件只有一行代码:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Context docBase="d:/Demo1" reloadable="true"></Context>
```

这里的 docBase 是必须的属性, 它定义了应用程序的位置。reloadable 属性是可选的, 如果存在并且值为 true, 那么一旦程序中 Java 类文件或者是其他资源文件有任何添加, 减少或者是更新, TOMcat 都可以检测到, 并且重新加载该应用程序

当把上下文文件添加到 Tomcat 的指定目录下, Tomcat 就会自动加载应用程序。当删除这个文件时, Tomcat 就会自动卸载应用程序

方式二

在 conf/server.xml 文件中的 Host 标签下添加一个 Context 元素

```
<Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true">
```

```
    <Context path="/Demo2" docBase="d:/Demo1" reloadable="true"></Context>
```

```
</Host>
```

以第一种方式不同之处在于此处定义上下文需要给上下文路径定义 path 属性, 这个 path 属性的值表示需要访问的项目名, 访问的 URL 就为: http://localhost:8080/Demo2 一般来说, 不建议通过 server.xml 来管理上下文, 因为修改后只有重启 Tomcat 后, 配置才能生效。不过, 如果有多个应用程序需要测试, 使用这种方式或许更为方便, 因为可以在一个文件中同时管理所有的应用程序

隐式部署

方式三

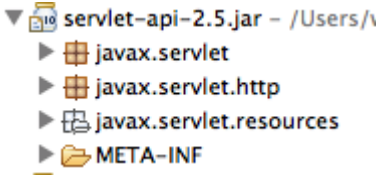
隐式部署真是太方便了, 非常建议使用这种方式部署项目。通过将一个 war 文件或者整个应用程序复制到 Tomcat 的 webapps(Tomcat 默认的部署项目位置, 可以在 server.xml 文件中修改, 但不建议修改)目录下, 启动服务器就可以了

	<p><b>**将应用部署到 Tomcat 根目录的三种方法</b></p> <p>将应用部署到 Tomcat 根目录的目的是可以通过“http://[ip]:[port]”直接访问应用，而不是使用“http://[ip]:[port]/[appName]”上下文路径进行访问。</p> <p>方法一：（最简单直接的方法）</p> <p>删除原 webapps/ROOT 目录下的所有文件，将应用下的所有文件和文件夹复制到 ROOT 文件夹下。</p> <p>方法二：</p> <p>删除原 webapps/ROOT 目录下的所有文件，修改文件“conf/server.xml”，在 Host 节点下增加如下 Context 的内容配置：</p> <pre>&lt;Host name="localhost"  appBase="webapps" unpackWARs="true" autoDeploy="true"       xmlValidation="false" xmlNamespaceAware="false"&gt;     .....     &lt;Context path="" docBase="C:/apache-tomcat-6.0.32/myapps/bc.war"&gt;&lt;/Context&gt; &lt;/Host&gt;</pre> <p>注意：</p> <ol style="list-style-type: none"> <li>1) path 的值设置为空；</li> <li>2) 应用不要放到 tomcat 的 webapps 目录下(如上述配置是放到自定义的文件夹 myapps 内的)，否则访问时路径会有问题；</li> <li>3) docBase 指定到绝对路径。</li> </ol> <p>如配置此设置后重启 tomcat，如果 docBase 指向的是 war 文件，则会自动将 war 解压到 webapps/ROOT 目录；如果 docBase 指向的是应用已解压好的目录，如 docBase="C:/apache-tomcat-6.0.32/myapps/bc"，则 tomcat 不会生成 webapps/ROOT 目录（这种情况下，这样就可以不用删除 webapps/ROOT 目录，但 webapps/ROOT 目录内的内容是无法访问的），访问时将直接使用 docBase 指定的目录。</p> <p>方法三：</p> <p>与方法二类似，但不是修改全局配置文件“conf/server.xml”，而是在“conf/Catalina/localhost”目录下增加新的文件“ROOT.xml”（注意大小写），文件内容如下：</p> <pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;Context      path=""          docBase="C:/apache-tomcat-6.0.32/myapps/bc.war" reloadable="true"&gt; &lt;/Context&gt;</pre>
考核点	
学员问题汇总	
作业	
课后总结分析	





## 东软睿道教案

章节	第 2 章 servlet 开发技术						
教学目标与要求	利用servlet实现动态页面，servlet应用请求与转发，servletconfig接口，servlet请求与响应						
重点难点	重点：servlet应用请求与转发，servlet请求与响应 难点：servlet应用请求与转发						
教学内容设计及学时分配	<p><b>【教学进程安排】</b></p> <table><tr><td>Servlet 介绍</td><td>2 课时</td></tr><tr><td>Servletconfig 与 servletcontext 接口</td><td>1 课时</td></tr><tr><td>Servlet 请求与响应</td><td>2 课时</td></tr></table> <p><b>【课程主要内容】</b></p> <h3>1Servlet 简介</h3> <p>Java Servlet 是和平台无关的服务器端组件，它运行在 Servlet 容器中。Servlet 容器负责 Servlet 和客户的通信以及调用 Servlet 的方法，Servlet 和客户的通信采用“请求/响应”的模式。</p> <h2>Servlet 体系结构</h2> <p>以 TOMCAT 为例，&lt;CATALINA_HOME&gt;/lib/servlet-api.jar 文件为 Servlet API 的类库文件。</p> <p>Servlet API 主要由两个 Java 包组成： javax.servlet 和 javax.servlet.http</p>  <ul style="list-style-type: none"><li>在 javax.servlet 包中定义了 Servlet 接口及相关的通用接口和类；</li><li>在 javax.servlet.http 包中主要定义了与 HTTP 协议相关的 HttpServletRequest 类，HttpServletRequest 接口和 HttpServletResponse 接口；</li></ul>	Servlet 介绍	2 课时	Servletconfig 与 servletcontext 接口	1 课时	Servlet 请求与响应	2 课时
Servlet 介绍	2 课时						
Servletconfig 与 servletcontext 接口	1 课时						
Servlet 请求与响应	2 课时						

## 1.1. Servlet 接口

在 Servlet 接口中定义了 5 个方法，其中 3 个方法都是由 Servlet 容器来调用的，容器会在 Servlet 的生命周期的不同阶段调用特定的方法：

▼ ⓘ Servlet

- `init(ServletConfig) : void`
- `getServletConfig() : ServletConfig`
- `service(ServletRequest, ServletResponse) : void`
- `getServletInfo() : String`
- `destroy() : void`

- `init(ServletConfig)` —— 负责初始化 Servlet 对象，容器在创建好 Servlet 对象后，就会调用该方法；
- `service(ServletRequest req, ServletResponse res)` —— 负责相应客户的请求，为客户提供相应服务。当容器接受到客户端要求访问特定 Servlet 对象的请求时，就会调用该 Servlet 对象的 `service()` 方法；
- `destroy()` —— 负责释放 Servlet 对象占用的资源。当 Servlet 对象结束声明周期时，容器会调用此方法；

## 1.2. GenericServlet 抽象类

Servlet - javax.servlet

▼ ⓘ Servlet

- ▼ ⓘ GenericServlet
- ⓘ HttpServlet
- ▼ ⓘ JspPage
- ⓘ HttpJspPage

GenericServlet 抽象类为 Servlet 接口提供了通用实现，它与任何网络应用层协议无关。

GenericServlet 除了实现了 Servlet 接口，还实现了 ServletConfig 接口和 Serializable 接口：

[java] [view plain copy](#)

```
1. public abstract class GenericServlet
2.     implements Servlet, ServletConfig, java.io.Serializable
```

GenericServlet 类实现了 Servlet 接口中的 `init(ServletConfig config)` 初始化方法。GenericServlet 类有一个 `ServletConfig` 类型的 `private` 成员变量，当 Servlet 容器调用 GenericServlet 的 `init(ServletConfig)` 方法时，该

方法使得私有变量引用由容器传入的 `ServletConfig` 对象。`GenericServlet` 类还定义了一个不带参数的 `init()` 方法，`init(ServletConfig)` 方法会调用此方法。因此在子类中重写 `init` 时，最好重写 `init()` 方法，若重写 `init(ServletConfig)` 方法，还需要先调用父类的 `init(ServletConfig)` 方法（`super.init(config)`）。

`GenericServlet` 类没有实现 `Servlet` 接口中的 `service()` 方法。`service()` 方法是 `GenericServlet` 类中唯一的抽象方法，`GenericServlet` 类的具体子类必须实现该方法。

`GenericServlet` 类实现了 `Servlet` 接口的 `destroy()` 方法，但实际什么也没做。

`GenericServlet` 类实现了 `ServletConfig` 接口的所有方法。

### 1.3. `HttpServlet` 抽象类

`HttpServlet` 类是 `GenericServlet` 类的子类。`HttpServlet` 类为 `Servlet` 接口提供了与 HTTP 协议相关的通用实现，也就是说，`HttpServlet` 对象适合运行在与客户端采用 HTTP 协议通信的 `Servlet` 容器或者 Web 容器中。

在我们自己开发的 Java Web 应用中，自定义的 `Servlet` 类一般都扩展自 `HttpServlet` 类。

`HttpServlet` 类实现了 `Servlet` 接口中的 `service(ServletRequest, ServletResponse)` 方法，而该方法实际调用的是它的重载方法 `HttpServlet.service(HttpServletRequest, HttpServletResponse)`；

在上面的重载 `service()` 方法中，首先调用 `HttpServletRequest` 类型的参数的 `getMethod()` 方法，获得客户端的请求方法，然后根据该请求方式来调用匹配的服务方法；如果为 GET 方式，则调用 `doGet()` 方法，如果为 POST 方式，则调用 `doPost()` 方法。

`HttpServlet` 类为所有的请求方式，提供了默认的实现 `doGet()`、`doPost()`、`doPut()`、`doDelete()` 方法；这些方法的默认实现都会向客户端返回一个错误。

对于 `HttpServlet` 类的具体子类，一般会针对客户端的特定请求方法，覆盖 `HttpServlet` 类中的相应的 `doXXX` 方法。如果客户端按照 GET 或 POST 方式请求访问 `HttpServlet`，并且这两种方法下，`HttpServlet` 提供相同的服务，那么可以只实现 `doGet()` 方法，并且让 `doPost()` 方法调用 `doGet()` 方法。

### 1.4. `ServletRequest` 接口

ServletRequest 表示来自客户端的请求；当 Servlet 容器接收到客户端要求访问特定 Servlet 的请求时，容器先解析客户端的原始请求数据，把它包装成一个 ServletRequest 对象。

ServletRequest 接口提供了一系列用于读取客户端的请求数据的方法，例如：

- `getContentTypeLength()` —— 返回请求正文的长度，如果请求正文的长度未知，则返回-1；
- `getContentType()` —— 获得请求正文的 MIME 类型，如果请求正文的类型为止，则返回 `null`；
- `getInputStream()` —— 返回用于读取请求正文的输入流；
- `getLocalAddr()` —— 返回服务端的 IP 地址；
- `getLocalName()` —— 返回服务端的主机名；
- `getLocalPort()` —— 返回服务端的端口号；
- `getParameters()` —— 根据给定的请求参数名，返回来自客户请求中的匹配的请求参数值；
- `getProtocol()` —— 返回客户端与服务器端通信所用的协议名称及版本号；
- `getReader()` —— 返回用于读取字符串形式的请求正文的 `BufferedReader` 对象；
- `getRemoteAddr()` —— 返回客户端的 IP 地址
- `getRemoteHost()` —— 返回客户端的主机名
- `getRemotePort()` —— 返回客户端的端口号

### 1.5.    [HttpServletRequest 接口](#)

HttpServletRequest 接口是 ServletRequest 接口的子接口。

HttpServletRequest 接口提供了用于读取 HTTP 请求中的相关信息的方法：

- `getContextPath()` —— 返回客户端请求方法的 Web 应用的 URL 入口，例如，如果客户端访问的 URL 为 `http://localhost:8080/helloapp/info`，那么该方法返回 `“/helloapp”`；
- `getCookies()` —— 返回 HTTP 请求中的所有 Cookie；
- `getHeader(String name)` —— 返回 HTTP 请求头部的特定项；
- `getHeaderName()` —— 返回一个 Enumeration 对象，它包含了 HTTP 请求头部的所有项目名；
- `getMethod()` —— 返回 HTTP 请求方式；
- `getRequestURL()` —— 返回 HTTP 请求的头部的第一行中的 URL；
- `getQueryString()` —— 返回 HTTP 请求中的查询字符串，即 URL 中的 `“？”` 后面的内容；

## 1.6. ServletResponse 接口

Servlet 通过 ServletResponse 对象来生成响应结果。

ServletResponse 接口定义了一系列与生成响应结果相关的方法，例如：

- `setCharacterEncoding()` —— 设置相应正文的字符编码。响应正文的默认字符编码为 ISO-8859-1；
- `setContentLength()` —— 设置响应正文的长度；
- `setContentType()` —— 设置响应正文的 MIME 类型；
- `getCharacterEncoding()` —— 获得响应正文的字符编码
- `getContentType()` —— 获得响应正文的 MIME 类型
- `setBufferSize()` —— 设置用于存放响应正文数据的缓冲区的大小
- `getBufferSize()` —— 获得用于存放响应正文数据的缓冲区的大小；
- `reset()` —— 清空缓冲区内的正文数据，并且清空响应状态代码及响应头
- `resetBuffer()` —— 仅仅清空缓冲区的正文数据，不清空响应状态代码及响应头；
- `flushBuffer()` —— 强制性地吧缓冲区内的响应正文数据发送到客户端；
- `isCommitted()` —— 返回一个 boolean 类型的值，如果为 true，表示缓冲区内的数据已经提交给客户，即数据已经发送到客户端；
- `getOutputStream()` —— 返回一个 ServletOutputStream 对象，**Servlet 用它来输出二进制的正文数据；**
- `getWriter()` —— 返回一个 PrintWriter 对象，**Servlet 用它来输出字符串形式的正文数据；**

**ServletResponse 中响应正文的默认 MIME 类型是 text/plain, 即纯文本类型，而 HttpServletResponse 中响应正文的默认 MIME 类型为 text/html，即 HTML 文档类型。**

为了提高输出数据的效率，ServletOutputStream 和 PrintWriter 首先把数据写到缓冲区内。当缓冲区内数据被提交给客户后，ServletResponse 的 `isComitted` 方法返回 true。在以下几种情况下，缓冲区内数据会被提交给客户，即数据被发送到客户端：

- 当缓冲区内数据已满时，ServletOutPutStream 或 PrintWriter 会自动把缓冲区内数据发送给客户端，并且清空缓冲区；
- Servlet 调用 ServletResponse 对象的 `flushBuffer` 方法；
- Servlet 调用 ServletOutputStream 或 PrintWriter 对象的 `flush` 方法或 `close` 方法；

**为了确保 ServletOutputStream 或 PrintWriter 输出的所有数据都会被提交给客户，比较安全的做法是在所有数据都输出完后，调用 ServletOutputStream 或 PrintWriter 的 `close()` 方法**（Tomcat 中，会自动关闭）。

**如果要设置响应正文的 MIME 类型和字符编码，必须先调用 ServletResponse 对象的 `setContentType()` 和 `setCharacterEncoding()` 方法，然后再调用 ServletResponse 的 `getOutputStream()` 或 `getWriter()` 方法，提交缓冲区内正文**

数据；只有满足这样的操作顺序，所做的设置才能生效。

## 1.7. HttpServletResponse 接口

HttpServletResponse 接口提供了与 HTTP 协议相关的一些方法，Servlet 可通过这些方法来设置 HTTP 响应头或向客户端写 Cookie。

- `addHeader()` —— 向 HTTP 响应头中加入一项内容
- `sendError()` —— 向客户端发送一个代表特定错误的 HTTP 响应状态码
- `setHeader()` —— 设置 HTTP 响应头中的一项内容，如果在响应头中已经存在这项内容，则原来的设置被覆盖
- `setStatus()` —— 设置 HTTP 响应的状态码
- `addCookie()` —— 向 HTTP 响应中加入一个 Cookie

在 HttpServletResponse 接口中定义了一些代表 HTTP 响应状态码的静态常量。

## 1.8. ServletConfig 接口

当 Servlet 容器初始化一个 Servlet 对象时，会为此 Servlet 对象创建一个 ServletConfig 对象。

在 Servlet 对象中包含了 Servlet 的初始化参数信息。

ServletConfig 接口中定义了以下方法：

- `getInitParameter(String name)` —— 返回匹配的初始化参数值
- `getInitParameterNames()` —— 返回一个 Enumeration 对象，里面包含了所有的初始化参数名
- `getServletContext()` —— 返回一个 ServletContext 对象
- `getServletName()` —— 返回 Servlet 的名字，即 web.xml 文件中相应<servlet>元素的<servlet-name>子元素的值；如果没有为 servlet 配置<servlet-name>子元素，则返回 Servlet 类的名字

HttpServlet 类继承了 GenericServlet 类，而 GenericServlet 类实现了 ServletConfig 接口，因此 HttpServlet 或

GenericServlet 类及子类中都可以直接调用 ServletConfig 接口中的方法。

## 1.9. ServletContext 接口

ServletContext 是 Servlet 与 Servlet 容器之间直接通信的接口。

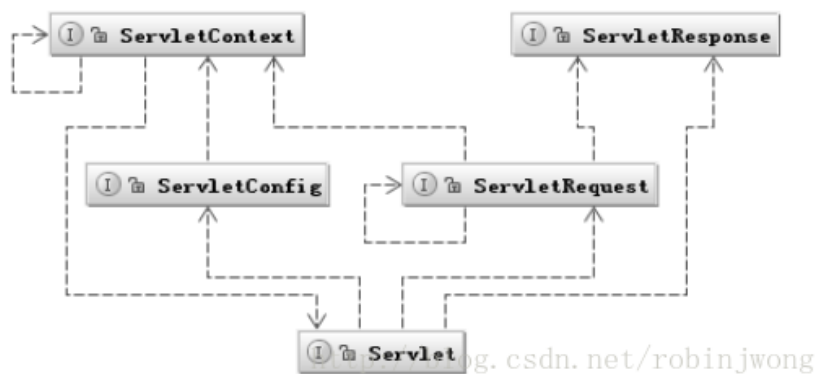
Servlet 容器在启动一个 Web 应用时，会为它创建一个 ServletContext 对象。每个 Web 应用都有唯一的 ServletContext 对象，可以把 ServletContext 对象形象地理解为 Web 应用的总管家，同一个 Web 应用中的所有 Servlet 对象都共享一个 ServletContext，Servlet 对象可以通过其访问容器中的各种资源。

ServletContext 接口提供的方法可以分为以下几种类型：

- 用于在 web 应用范围内存取共享数据的方法：
  - `setAttribute(String name, Object object)` —— 把一个 Java 对象与一个属性名绑定，并存入到 ServletContext 中；
  - `getAttribute()` —— 返回指定属性名的属性值
  - `getAttributeNames()` —— 返回一个 Enumeration 对象，包含所有存放在 ServletContext 中的属性名
  - `removeAttributes()` —— 从 ServletContext 中删除匹配的属性
- 访问当前 Web 应用的资源：
  - `getContextPath()` —— 返回当前 Web 应用的 URL 入口
  - `getInitParameter()` —— 返回 Web 应用范围内的匹配的初始化参数值。在 web.xml 中，直接在 <web-app> 根元素下定义的 <context-param> 元素表示应用范围内的初始化参数
  - `getServletContextName()` —— 返回 Web 应用的名字，即 web.xml 文件中 <display-name> 元素的值
  - `getRequestDispatcher()` —— 返回一个用于向其他 WEB 组件转发请求的 RequestDispatcher 对象
- 访问 Servlet 容器中的其他 WEB 应用：
- 访问 Servlet 容器的相关信息：
- 访问服务器端的文件系统资源：
  - `getRealPath()` —— 根据参数指定的虚拟路径，返回文件系统中的真实路径
  - `getResources()` —— 返回一个映射到参数指定的路径的 URL
  - `getResourceAsStream()` —— 返回一个用于读取参数指定的文件的输入流
  - `getMimeType()` —— 返回参数指定的文件 MIME 类型
- 输出日志：

- `log(String msg)` —— 向 Servlet 的日志文件中写日志
- `log(String message, Throwable throwable)` —— 向 Servlet 的日志文件中写入错误日志，以及异常的堆栈信息

## 2. Servlet 相关类的关系



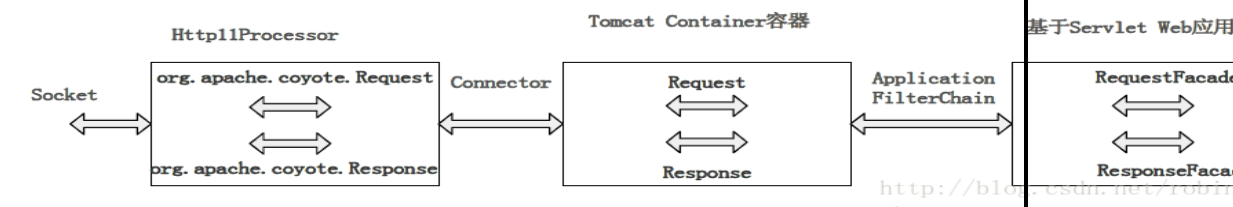
与 Servlet 主动关联的是三个类，分别是 ServletConfig，ServletRequest 和 ServletResponse。这三个类都是通过容器传递给 Servlet 的；其中，ServletConfig 是在 Servlet 初始化时传给 Servlet 的，后两个是在请求到达时调用 Servlet 传递过来的。

对于 Request 和 Response，以 TOMCAT 为例，tomcat 接到请求首先将会创建 `org.apache.coyote.Request` 和 `org.apache.coyote.Response`，这两个类是 Tomcat 内部使用的描述一次请求和相应的信息类，它们是一个轻量级的类，作用就是在服务器接收到请求后，经过简单解析将这个请求快速分配给后续线程去处理。接下来当交给一个用户线程去处理这个请求时又创建 `org.apache.catalina.connector.Request` 和 `org.apache.catalina.connector.Response` 对象。这两个对象一直贯穿整个 Servlet



et 容器直到要传给 Servlet, 传给 Servlet 的是 Request 和 Response 的 Facade 类。

Request 和 Response 的转变过程：



### 3. Servlet 如何工作

当用户从浏览器向服务器发起的一个请求通常会包含如下信息：

[java] [view plain copy](#)

1. `http://hostname:port/contextpath/servletpath`

hostname 和 port 用来与服务器建立 TCP 连接，后面的 URL 用来选择在服务器中哪个子容器服务用户的请求。

在 Tomcat7 中，这种映射工作由专门的一个类完成：`org.apache.tomcat.util.http.mapper`，这个类保存了 tomcat 的 container 容器中的所有子容器的信息。`org.apache.catalina.connector.Request` 类在进入 Container 容器之前，Mapper 将会根据这次请求的 hostname 和 contextpath 将 host 和 context 容器设置到 Request 的 mappingData 属性中，所以当 Request 进入 container 容器之前，对于它要访问哪个子容器就已经确定了。

### 4. Servlet 的实际使用

- 我们自己定义的 servlet 通常去继承 `HttpServlet` 或 `GenericServlet` 类。
- 采用 MVC 框架的实现中，其基本原理是将所有的请求都映射到一个 Servlet，然后去实现 service 方法，这

个方法也就是 MVC 框架的入口。

**Servlet** 可完成如下功能：

创建并返回基于客户请求的动态 HTML 页面。

与其它服务器资源（如数据库或基于 Java 的应用程序）进行通信。



**Servlet** 本质上就是一个运行在 **Servlet** 容器中的 Java 类，现在充当 **Servlet** 容器的就是 Tomcat

第一个 **Servlet**——HelloWorld

1. 创建一个 Java 类实现 **Servlet** 接口，并实现其中所有的方法；

```
package servlet;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
@WebServlet("/hello")
```

```
public class HelloServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
```

```
    }
```

```
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
```

```
        System.out.println("HelloServlet");
```

```
    }
```

```
}
```

```
@WebServlet("/hello")
```

表示的是为 `helloServlet` 映射一个访问路径 `/hello`, 我们在浏览器上输入 <http://localhost:8080/hello>, 代码执行程序流会进入到 `helloServlet` 的 `doGet` 方法。

2. 运行 Tomcat, 在浏览器中输入 <http://localhost:8080/hello>, 可以发现控制台输出了一些内容, 首先是构造器, 然后是 `init`, 再然后是 `service`, 然后每次刷新, 每次都只出现了 `service`, 然后关闭服务, 发现出现 `destory`。

以上是 3.0 配置, 由于课程大纲要求依旧使用 2.5 在此列出 2.5 的配置  
servlet 2.5 与 3.0 版本配置

#### web.xml v2.5

Xml 代码 ☆

```
1. <?xml version="1.0" encoding="UTF-8"?>
2.
3. <web-app xmlns="http://java.sun.com/xml/ns/javaee"
4.
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6.
7. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
8.
9. version="2.5">
10.
11. </web-app>
```

#### web.xml v3.0

Xml 代码 ☆

```
1. <?xml version="1.0" encoding="UTF-8"?>
2.
3. <web-app
4.     version="3.0"
5.     xmlns="http://java.sun.com/xml/ns/javaee"
6.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7.     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
8.
9. </web-app>
```

## 2.Servlet 的生命周期

### ①服务器加载 Servlet

### ②创建 Servlet 实例

--只有第一次请求 Servlet 时，创建 Servlet 实例，调用构造器

### ③初始化 init()

--只被调用一次，在创建好实例后立即被调用，用于初始化当前 Servlet

### ④service() 处理用户请求

--可以被多次调用，每次请求都会调用 service 方法，实际用于响应请求的，根据用户请求的类型（get 或者 post），调用 doGet 或者 doPost 方法。

### ⑤destory() 销毁

--只被调用一次，在当前 Servlet 所在的 WEB 应用被卸载前调用，用于释放当前 Servlet 所占用的资源

其中在 init 方法中需要注意

### ServletContext 对象与 ServletConfig

#### 一、ServletContext 对象详解

ServletContext 代表当前 web 应用

当服务器启动，web 应用加载完成后，立即创建一个 ServletContext 对象代表当前 web 应用。从此驻留在内存中唯一的代表当前 web 应用。直到服务器关闭或 web 应用移除出容器时，随着 web 应用的销毁而销毁。

#### 1.获取 ServletContext

```
servletConfig.getServletContext();
```

```
this.getServletContext();
```

在 web.xml 中的<web-app>根标签下定义<context-param>,可以给全局定义初始化参数，比如 encode

在 servlet 中调用时，this.getServletContext().getInitParamter("encode"), 得到 String 型的变量

#### 2.加载资源文件：

路径难题

ServletContext.getRealPath("xxxxx");//会在传入的路径前拼接上当前 web 应用的硬盘

路径。~另外在没有 ServletContext 的环境下，我们可以使用类加载器加载资源。但是要注意，类加载器加载资源时，路径要相对于平常加载类的位置进行计算。

```
ClassLoader.getResource().getPath();
```

```
ClassLoader.getResourceAsStream();
```

3.读取 web 应用的初始化:

我们可以在 web.xml 的根目录下为整个 web 应用配置一些初始化参数。

在<Context-param>标签下添加

可以通过 ServletContext 对象来读取这些整个 web 应用的初始化参数。

```
servletContext.getInitParamter();
```

```
servletContext.getInitParamterNames();
```

4.作为域对象使用:

域: 一个对象具有了一个可以被看见的范围, 那么利用这个对象身上的 Map, 可以在这个范围内共享数据, 这样的对象叫做域对象。

javaweb 中一共有四大作用域, 其中 ServletContext 就是其中最大的一个。

```
setAttribute(String name,Object value);
```

```
getAttribute(String name);
```

```
removeAttribute(String name);
```

```
getAttributeNames();
```

5.ServletContext 域的范围:

在整个 web 应用中共享数据

生命周期:

服务器启动 web 应用加载 ServletContext 创建, 直到服务器关闭或 web 应用移除出容器时随着 web 应用的销毁, ServletContext 销毁。

主要的作用:

在整个 web 应用中共享数据。paramter -- 请求参数

initparamter -- 初始化参数

attribute -- 域属性

二、ServletConfig 对象详解

在 Servlet 的配置文件中, 可以用一个或多个<init-param>标签为 servlet 配置一些初始化参数。

当 servlet 配置了初始化参数之后, web 容器在创建 servlet 实例对象时, 会自动将这些初始化参数封装到 ServletConfig 对象中, 并在调用 servlet 的 init 方法时, 将 ServletConfig 对象传递给 Servlet。

进而, 程序员通过 Servlet 对象得到当前 servlet 的初始化参数信息。获取 ServletConfig 中初始化信息步骤:

1. 创建私有变量:

```
private ServletConfig config = null;
```

2、重写 init 方法:

```
this.config = config, 从而获取 ServletConfig 对象;
```

3、获取<init-param>配置:

```
//获取初始化参数 String value1 = this.config.getInitParameter("x1");
```

```
// 获得 配置文档中 <inti-param> 标签下 name 对应的 value String value2 = this.config.getInitParameter("x2");
```

```
//获取所有初始化参数 Enumeration e = this.config.getInitParameterNames();
```

```
while(e.hasMoreElements()){ String name = (String) e.nextElement(); String value = this.config.getInitParameter(name); System.out.println(name+"="+value); }
```

4、ServletConfig 的作用：

获取字符集编码：String charset = this.config.getInitParameter("charset");

获得数据库连接信息：String url = this.config.getInitParameter("url");

String username = this.config.getInitParameter("username");

String password = this.config.getInitParameter("password");

获得配置文件：String configFile = this.config.getInitParameter("config");

### 3.load-on-startup

可以指定 Servlet 被创建的时机

在加载完 Servlet 后直接就创建了 Servlet 实例，并进行了初始化，在此期间我们并没有提出请求。

如果配置了多个 Servlet，load-on-startup 的值越小越先启动（包括 0，负数不会被启动，还是需要发出请求才创建实例并初始化）；

## 4.Servlet 映射细节

在 Servlet 映射到的 URL 中也可以使用 \* 通配符，但是只能有两种固定的格式：

一种格式是“\*.扩展名”

```
@WebServlet(value = "*.do")
public class HelloServlet extends HttpServlet
{
}
```

另一种格式是以正斜杠 (/) 开头并以“/\*”结尾。

```
@WebServlet(value = "/*")
public class HelloServlet extends HttpServlet
{
}
```

**注意：**既带 / 又带 \* 的，又带扩展名的，是不合法的（例如：/.do）。

## 5.POST 和 GET 请求

post 请求方式我们可以看到请求参数在请求体里面  
将请求方式改为 get 后再运行观察

```
<form action="/loginServlet" method="get">
    user:<input type="text" name="user">
    password:<input type="password" name="password">
    <input type="submit" value="Submit">
</form>
```

可以观察到地址栏是这样的:

<http://localhost:8080/loginServlet?user=123&password=123>

get 请求把请求参数附着在 url 后面, 中间以“?” 分割。

## 6. 使用 GET 方式传递参数

①在浏览器地址栏中输入某个 URL 地址或单击网页上的一个超链接时, 浏览器发出的 HTTP 请求消息的请求方式为 GET。

②如果网页中的<form>表单元素的 method 属性被设置为了“GET”, 浏览器提交这个 FORM 表单时生成的 HTTP 请求消息的请求方式也为 GET。

③使用 GET 请求方式给 WEB 服务器传递参数的格式:

<http://www.neusoft.net/counter.jsp?name=yzn&password=123>

④使用 GET 方式传送的数据量一般限制在 1KB 以下。

## 7. 使用 POST 方式传递参数

①POST 请求方式主要用于向 WEB 服务器端程序提交 FORM 表单中的数据。

②POST 方式将各个表单字段元素及其数据作为 HTTP 消息的实体内容发送给 WEB 服务器, 传送的数据量要比使用 GET 方式传送的数据量大得多。

3. 如果网页中的<form>表单元素的 method 属性被设置为了“POST”, 浏览器提交这个 FORM 表单时生成的 HTTP 请求消息的请求方式也为 POST。

## 8.如何在 Servlet 中获取信息

其中, 方法的参数 `HttpServletRequest request` 和 `HttpServletResponse response` 封装了请求和响应信息

## 9 如何获取请求信息

**HttpServletRequest** 常用的方法:

① **String getParameter(String name)**

--根据请求参数的名字, 返回参数值, 特别常用

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    String user = request.getParameter("user");
    String password = request.getParameter("password");
    System.out.println(user+" "+password);
}
```

② **String[] getParameterValues(String name)**

--根据请求参数的名字, 返回请求参数对应的字符串数组 (例如: 一组复选框-->名字是一样的)

### 解决请求中的中文乱码问题:

```
request.setCharacterEncoding("utf-8");
```

### 如何返回响应信息

**HttpServletResponse** 常用的方法:

① **getWriter()** 方法

--返回 **PrintWriter** 对象, 调用这个对象的 **println()** 方法可以将信息直接打印在客户的浏览器上

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {

    PrintWriter out = response.getWriter();
    out.println("hello...");

}
```

### 解决回应中的中文乱码问题

```
//1. 设置 response 的编码为 utf-8
response.setCharacterEncoding("utf-8");
```

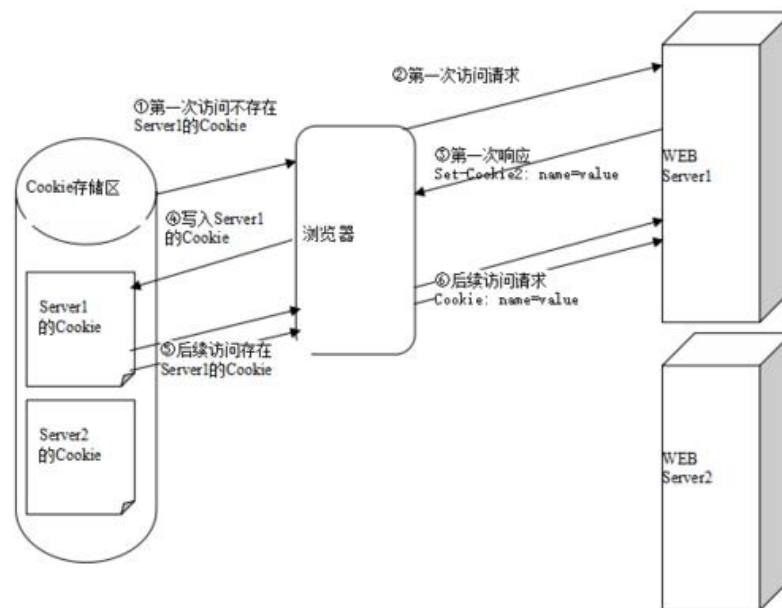


	<pre>//2. 通知浏览器，以 UTF-8 的编码打开 response.setContentType("text/html;charset=UTF-8");</pre>
考核点	
学员问题汇总	
作业	
课后 总结分析	

## 东软睿道教案

章节	第三章 Servlet 会话跟踪								
教学目标 与要求	掌握cookie与session技术，了解cookie，session的生命周期，其他会话跟踪技术								
重点 难点	重点： cookie与session的生命周期 难点： cookie与session配合使用								
教学内容设计 及学时分配	<p><b>【教学进程安排】</b></p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: left;">Cookie 技术</td><td style="text-align: right;">3 课时</td></tr> <tr> <td style="text-align: left;">Session 技术</td><td style="text-align: right;">3 课时</td></tr> <tr> <td style="text-align: left;">url 重写</td><td style="text-align: right;">1 课时</td></tr> <tr> <td style="text-align: left;">隐藏表单</td><td style="text-align: right;">1 课时</td></tr> </table> <p><b>【课程主要内容】</b></p> <p><b>1. Cookie 机制</b></p> <p>cookie 机制采用的是在客户端保持 HTTP 状态信息的方案。</p> <p>Cookie 意为“甜饼”，是由 W3C 组织提出，最早由 Netscape 社区发展的一种机制。目前 Cookie 已经成为标准，所有的主流浏览器如 IE、Netscape、Firefox、Opera 等都支持 Cookie。</p> <p>由于 HTTP 是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是 Cookie 的工作原理。</p> <p>Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器。服务器检查该 Cookie，以此来辨认用户状态。</p> <p>一个 Cookie 只能标识一种信息，它至少含有一个标识该信息的名称（NAME）和设置值（VALUE）。</p>	Cookie 技术	3 课时	Session 技术	3 课时	url 重写	1 课时	隐藏表单	1 课时
Cookie 技术	3 课时								
Session 技术	3 课时								
url 重写	1 课时								
隐藏表单	1 课时								

## 2.Cookie 的传送过程示意图



演示代码

```
@WebServlet("/cookie")
public class CookieServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        // 在 JavaWEB 规范中使用 Cookie 类代表 cookie
        // 1. 创建一个 Cookie 对象
        Cookie cookie = new Cookie("name", "neusoft");
        // 2. 调用 response 的一个方法把 Cookie 传给客户端
        response.addCookie(cookie);
    }
}
```

Servlet API 中提供了一个 `javax.servlet.http.Cookie` 类来封装 Cookie 信息，它包含有生成 Cookie 信息和提取 Cookie 信息的各个属性的方法。

Cookie 类的方法：

--构造方法: public Cookie(String name,String value)  
--getName 方法  
--setValue 与 getValue 方法  
--setMaxAge 与 getMaxAge 方法  
--setPath 与 getPath 方法

HttpServletResponse 接口中定义了一个 addCookie 方法,它用于在发送给浏览器的 HTTP 响应消息中增加一个 Set-Cookie 响应头字段。

HttpServletRequest 接口中定义了一个 getCookies 方法,它用于从 HTTP 请求消息的 Cookie 请求头字段中读取所有的 Cookie 项。

```
@WebServlet("/GetCookieServlet")
public class GetCookieServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        PrintWriter out = response.getWriter();
        // 1. 获取 Cookie (没有单独获取某一个 Cookie 的方法)
        Cookie[] cookies = request.getCookies();
        if(cookies != null && cookies.length >= 1){
            boolean bFind = false;
            for(Cookie cookie:cookies){
                if(cookie.getName().equals("name"))
                {
                    bFind = true;
                }
            }
            if(bFind)
            {
                out.print("neusoft Cookie exist");
            }
            else {
                out.print("no Cookie,create one and return");
                // 1. 创建一个 Cookie 对象
                Cookie cookie = new Cookie("name","neusoft");

                // 2. 调用 response 的一个方法 Cookie 传给客户端
                response.addCookie(cookie);
            }
        }
    }
}
```

```

    }else{
        out.print("no Cookie,create one and return");
        // 1. 创建一个 Cookie 对象
        Cookie cookie = new Cookie("name","neusoft");

        // 2. 调用 response 的一个方法 Cookie 传给客户端
        response.addCookie(cookie);
    }
}
}

```

如果创建了一个 cookie，并将他发送到浏览器，默认情况下它是一个会话级别的 cookie；存储在浏览器的内存中，用户退出浏览器之后被删除。若希望浏览器将该 cookie 存储在磁盘上，则需要使用 maxAge，并给出一个以秒为单位的时间。将最大时效设为 0 则是命令浏览器删除该 cookie。

发送 cookie 需要使用 HttpServletResponse 的 addCookie 方法，将 cookie 插入到一个 Set-Cookie HTTP 响应报头中。由于这个方法并不修改任何之前指定的 Set-Cookie 报头，而是创建新的报头，因此将这个方称为是 addCookie，而非 setCookie。

**setMaxAge(秒)设置 Cookie 的最大时效，若为 0 代表立即上除该 Cookie，若为负数表述不存储该 Cookie**

```

@WebServlet("/GetCookieServlet")
public class GetCookieServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        PrintWriter out = response.getWriter();
        // 1. 获取 Cookie（没有单独获取某一个 Cookie 的方法）
        Cookie[] cookies = request.getCookies();
        if(cookies != null && cookies.length >= 1){
            boolean bFind = false;
            for(Cookie cookie:cookies){
                if(cookie.getName().equals("name"))
                {
                    bFind = true;
                }
            }
        }
    }
}

```

```

    }
    if(bFind)
    {
        out.print("neusoft Cookie exist");
    }
    else {
        out.print("no Cookie, create one and return");
        // 1. 创建一个 Cookie 对象
        Cookie cookie = new Cookie("name", "neusoft");
        cookie.setMaxAge(200);
        // 2. 调用 response 的一个方法 Cookie 传给客户端
        response.addCookie(cookie);
    }
} else {
    out.print("no Cookie, create one and return");
    // 1. 创建一个 Cookie 对象
    Cookie cookie = new Cookie("name", "neusoft");
    cookie.setMaxAge(200);
    // 2. 调用 response 的一个方法 Cookie 传给客户端
    response.addCookie(cookie);
}
}
}

```

### 3. Cookie 的 Path 问题

```

@WebServlet("/path/PathServlet")
public class PathServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        Cookie cookie = new Cookie("name", "neusoft");
        response.addCookie(cookie);
    }
}

@WebServlet("/GetCookieServlet")
public class GetCookieServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,

```

```

HttpServletResponse response) throws ServletException,
IOException {

    }

    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        PrintWriter out = response.getWriter();
        // 1. 获取 Cookie (没有单独获取某一个 Cookie 的方法)
        Cookie[] cookies = request.getCookies();
        if(cookies != null && cookies.length >= 1){
            boolean bFind = false;
            for(Cookie cookie:cookies){
                if(cookie.getName().equals("name"))
                {
                    bFind = true;
                }
            }
            if(bFind)
            {
                out.print("neusoft Cookie exist");
            }
            else {
                out.print("no Cookie, create one and return");
            }
        }else{
            out.print("no Cookie, create one and return");
        }
    }
}

```

运行是读取不到的，路径调换一下，可以读到  
 Cookie 的作用范围：可以作用当前目录和当前目录的子目录，但不能作用与当前目录的上一级目录  
 如何解决：

```

@WebServlet("/path/PathServlet")
public class PathServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

    }
}

```

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    Cookie cookie = new Cookie("name", "neusoft");
    cookie.setPath("/");
    response.addCookie(cookie);
}
}
```

## 5.Cookie 小结

### (1) 简介

Cookie 机制采用的是在客户端保持 HTTP 状态信息的方案。

Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器。服务器检查该 Cookie，以此来辨认用户状态。

### (2) 作用

Cookie 的根本作用就是在客户端存储用户访问网站的一些信息。典型的应用有：

自动登录。

### (3) 缺陷

①Cookie 会被附加在每个 HTTP 请求中，所以无形中增加了流量。

②由于在 HTTP 请求中的 Cookie 是明文传递的，所以安全性成问题。（除非用 HTTPS）

③Cookie 的大小限制在 4KB 左右。对于复杂的存储需求来说是不够用的。

### (4) 常用方法



创建 Cookie: `Cookie cookie = new Cookie(name,value)`

向浏览器发送 Cookie: `response.addCookie(cookie)`

设置最大时效: `cookie.setMaxAge(秒)`, 当设置为 0 的时候, 使用 `response.addCookie(cookie)`, 表示删除该 cookie。

### Session 简单介绍

在 WEB 开发中, 服务器可以为每个浏览器用户创建一个会话对象(session 对象), 注意: 一个浏览器用户独占一个 session 对象(默认情况下)。因此, 在需要保存用户数据时, 服务器程序可以把用户数据写到浏览器用户独占的 session 中, 当用户使用浏览器访问该 web 应用的其它 servlet 时, 其它 servlet 可以从用户的 session 中取出该用户的数据, 为用户服务, 从而实现数据在多个页面中的共享。

### Session 和 Cookie 的主要区别

- Cookie 是把用户的数据写到用户的浏览器。
- Session 技术把用户的数据写到用户独占的 session 中, tomcat 服务器内存中。
- Session 对象由服务器创建, 开发人员可以调用 request 对象的 `getSession` 方法得到 session 对象

### session 机制演示图



服务器通过 request 对象的 `getSession` 方法创建出 session 对象后, 会把 session 的 id 号, 以 cookie 的形式回写给客户机, 这样, 只要客户机的浏览器不关, 再去访问服务器时, 都会带着 session 的 id 号去, 服务器发现客户机浏览器带 session id 过来了, 就会使用内存中与之对应的

session 为之服务。

## 6. Session cookie

session 通过 SessionID 来区分不同的客户，session 是以 cookie 为基础的，系统会创建一个名为 JSESSIONID 的输出 cookie，这称之为 session cookie，以区别 persistent cookies(也就是我们通常所说的 cookie)，session cookie 是存储于浏览器内存中的，并不是写到硬盘上的，session cookie 针对某一次会话而言，会话结束 session cookie 也就随着消失了，而 persistent cookie 只是存在于客户端硬盘上的一段文本。

关闭浏览器，只会使浏览器端内存里的 session cookie 消失，但不会使保存在服务器端的 session 对象消失，同样也不会使已经保存到硬盘上的持久化 cookie 消失。

## 7. 持久化 session cookie

```
@WebServlet(name = "TestServlet",urlPatterns = "/test")
public class TestServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        HttpSession httpSession = request.getSession();
        Cookie cookie = new
        Cookie("JSESSIONID",httpSession.getId());
        cookie.setMaxAge(300);
    }
}
```

关闭浏览器，5 分钟之内打开浏览器，session 依然有效。

### (1) HttpSession 的生命周期

问题：用户开一个浏览器访问一个网站，服务器是不是针对这次会话创建一个 session？

答：不是的。session 的创建时机是在程序中第一次去执行

request.getSession();这个代码，服务器才会为你创建 session。  
问题：关闭浏览器，会话结束，session 是不是就销毁了呢？  
答：不是的。session 是 30 分钟没人用了才会死，服务器会自动摧毁 session。

## **(2) session 何时会销毁**

A. 调用 invalidate() 方法，该方法使 HttpSession 立即失效

```
@WebServlet(name = "TestServlet",urlPatterns = "/test")
public class TestServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        HttpSession httpSession = request.getSession();
        httpSession.invalidate();
    }
}
```

B. 服务器卸载了当前 WEB 应用

C. 超出 session 过期时间

```
@WebServlet(name = "TestServlet",urlPatterns = "/test")
public class TestServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        HttpSession httpSession = request.getSession();
        httpSession.setMaxInactiveInterval(30); //单位秒
    }
}
```

web.xml 文件中也可以修改 session 的过期时间(全局的,所有的 session 的过期时间,若想单独设置还是像上面那样单独设置),单位分钟。

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

### **8.利用 session 完成用户登陆**

首先创建代表用户的 JavaBean——User.java。

```
public class User {
    private String username;
    private String password;

    public User() {
        super();
        // TODO Auto-generated constructor stub
    }

    public User(String username, String password) {
        super();
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

然后创建网站首页——index.jsp。

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
```

```

        pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    欢迎您: ${user.username }&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<a
href="/day07/login.html">登录</a>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<a
href="/day07/LogoutServlet">退出登录</a>
    <br/><br/><br/>
    <a href="/day07/SessionDemo1">购买</a><br/>
    <a href="/day07/SessionDemo2">结账</a>
</body>
</html>

```

接着创建登录页面——login.html。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
    <form action="/day07/LoginServlet" method="post">
        用户名: <input type="text" name="username"><br/>
        密码: <input type="password" name="password"><br/>
        <input type="submit" value="登录">
    </form>
</body>
</html>

```

再创建处理用户登录的 Servlet——LoginServlet。

```

public class LoginServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

```

```

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();

        String username = request.getParameter("username");
        String password = request.getParameter("password");

        List<User> list = DB.getAll();
        for(User user : list) {
            // 用户登录成功
            if(user.getUsername().equals(username) &&
user.getPassword().equals(password)) {
                request.getSession().setAttribute("user",
user); // 登录成功, 向 session 中存入一个登录标记
                response.sendRedirect("/day07/index.jsp");
                return;
            }
        }

        out.print("用户名或密码不正确!!!");
    }

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

class DB {
    public static List<User> list = new ArrayList<User>();

    static {
        list.add(new User("aaa", "123"));
        list.add(new User("bbb", "123"));
        list.add(new User("ccc", "123"));
    }

    public static List<User> getAll() {
        return list;
    }
}

```

最后创建用户注销的 Servlet——LogoutServlet。

// 完成用户注销

```
public class LogoutServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(false);
        if(session == null) {
            response.sendRedirect("/day07/index.jsp");
            return;
        }

        session.removeAttribute("user"); // 移除登录标记
        response.sendRedirect("/day07/index.jsp");

    }

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

## Session 的生命周期

1.服务器会把长时间没有活动的 Session 从服务器内存中清除，此时 Session 便失效。Tomcat 中 Session 的默认失效时间为 20 分钟。

2.调用 Session 的 invalidate 方法。

注:当禁用 cookie 时也是不能使用 session 的。

## 其他会话跟踪技术

话跟踪是一种灵活、轻便的机制，它使 Web 上的状态编程变为可能。

HTTP 是一种无状态协议，每当用户发出请求时，服务器就会做出响应，客户端与服务器之间的联系是离散的、非连续的。

	<p>当用户在同一网站的多个页面之间转换时，根本无法确定是否是同一个客户，会话跟踪技术就可以解决这个问题。</p> <p>当一个客户在多个页面间切换时，服务器会保存该用户的信息。</p> <p>有四种方法可以实现会话跟踪技术：<b>URL 重写</b>、<b>隐藏表单域</b>、<b>Cookie</b>、<b>Session</b>。</p> <p><b>a) URL 重写:</b>URL(统一资源定位符)是 Web 上特定页面的地址，URL 重写的技术就是在 URL 结尾添加一个附加数据以标识该会话,把会话 ID 通过 URL 的信息传递过去，以便在服务器端进行识别不同的用户</p> <p><b>b) 隐藏表单域:</b>将会话 ID 添加到 HTML 表元素中提交到服务器,此表元素并不在客户端显示</p> <p><b>c) Cookie:</b>Cookie 是 Web 服务器发送给客户端的一小段信息，客户端请求时可以读取该信息发送到服务器端，进而进行用户的识别。对于客户端的每次请求，服务器都会将 Cookie 发送到客户端,在客户端可以进行保存,以便下次使用。客户端可以采用两种方式保存这个 Cookie 对象，一种方式是 保存在 客户端内存中，称为临时 Cookie，浏览器关闭后 这个 Cookie 对象将消失。另外一种方式是保存在 客户机的磁盘上，称为永久 Cookie。以后客户端只要访问该网站，就会将这个 Cookie 再次发送到服务器上，前提是 这个 Cookie 在有效期内。 这样就实现了对客户的跟踪。 Cookie 是可以被禁止的。</p> <p><b>d) session:</b> 每一个用户都有一个不同的 session，各个用户之间是不能共享的，是每个用户所独享的，在 session 中可以存放信息。 在服务器端会创建一个 session 对象，产生一个 sessionId 来标识这个 session 对象,然后将这个 sessionId 放入到 Cookie 中发送到客户端，下一次访问时,sessionId 会发送到服务器，在服务器端进行识别不同的用户 Session 是依赖 Cookie 的，如果 Cookie 被禁用，那么 session 也将失效</p>
考核点	
学员问题汇总	
作业	
课后 总结分析	



## 东软睿道教案

章节	第四章 jsp						
教学目标与要求	通过本章的学习，使学生掌握jsp技术，掌握jsp内置对象。						
重点难点	重点：jsp常用标签，jsp内置对象 难点：jsp内置对象						
教学内容设计及学时分配	<p><b>【教学进程安排】</b></p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">1. JSP 基本语法</td><td style="width: 50%;">2 课时</td></tr> <tr> <td>2.jsp 内置对象</td><td>2 课时</td></tr> <tr> <td>3.jsp 与 javabean</td><td>2 课时</td></tr> </table> <p><b>【课程主要内容】</b></p> <p>JSP——Java Server Page:Java 服务端页面，在 html 页面中编写 Java 代码的页面。</p> <p>绝大多数时候，我们希望响应的不是简简单单一句话，而是一个页面，我们用 PrintWriter 对象去写一个页面也是可以的，但缺点太明显。</p> <p>所以直接返回一个页面，并且能够写 Java 代码就能大大简化我们的开发，这就是——JSP。</p> <p>当前不提倡在 jsp 页面中写 java 代码，代替使用 EL 表达式。</p> <p>JSP 是简化 Servlet 编写的一种技术，它将 Java 代码和 HTML 语句混合在同一个文件中编写，只对网页中的要动态产生的内容采用 Java 代码来编写，而对固定不变的静态内容采用普通静态 HTML 页面的方式编写。</p> <p><b><u>建立对 JSP 的直观认识</u></b></p> <p>在 JSP 页面中编写的 Java 代码需要嵌套在&lt;%和%&gt;中，嵌套在&lt;%和%&gt;之间的 Java 代码被称之为<b>脚本片段</b>（Scriptlets），没有嵌套在&lt;%和%&gt;之间的内容被称之为 JSP 的<b>模版元素</b>。</p> <p>WEB 容器（Servlet 引擎）接收到以 .jsp 为扩展名的 URL 的访问请求时，它将把该访问请求交给 JSP 引擎去处理。</p>	1. JSP 基本语法	2 课时	2.jsp 内置对象	2 课时	3.jsp 与 javabean	2 课时
1. JSP 基本语法	2 课时						
2.jsp 内置对象	2 课时						
3.jsp 与 javabean	2 课时						

每个 JSP 页面在第一次被访问时，JSP 引擎将它翻译成一个 Servlet 源程序，接着再把这个 Servlet 源程序编译成 Servlet 的 class 类文件，然后再由 WEB 容器（Servlet 引擎）像调用普通 Servlet 程序一样的方式来装载和解释执行这个由 JSP 页面翻译成的 Servlet 程序——JSP 本质上就是一个 Servlet。

idea 中 jsp 被编译出的 servlet 被放在哪里？

C:\Users\Administrator.IntelliJ IDEA 2016.1\system\tomcat\Tomcat\_8\_0\_32\_web1\work\Catalina\localhost\web\org\apache\jsp

新建一个 jsp 页面 hello.jsp

```
<%@page import="java.util.Date"%>
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <%
        Date date = new Date();
        System.out.println(date);
    %>
</body>
</html>
```

运行可以看到控制台上有时间输出。

## JSP 的运行原理

(1) WEB 容器 JSP 页面的访问请求时，它将把该访问请求交给 JSP 引擎去处理。Tomcat 中的 JSP 引擎就是一个 Servlet 程序，它负责解释和执行 JSP 页面。

(2) 每个 JSP 页面在第一次被访问时，JSP 引擎先将它翻译成一

个 Servlet 源程序，接着再把这个 Servlet 源程序编译成 Servlet 的 class 类文件，然后再由 WEB 容器像调用普通 Servlet 程序一样的方式来装载和解释执行这个由 JSP 页面翻译成的 Servlet 程序。

.一个 JSP 页面由五个元素组成：

1.普通 HTML 标记 2.JSP 标记 3.变量和方法的声明 4.Java 程序片 5.Java 表达式

#### JSP 脚本元素：

- 1、声明语句：<%! 声明语句 %>。使用声明语句的变量为全局变量，也就是说当有多个用户在执行此 JSP 页面时，将共享该变量。
- 2、JSP Scriptlets：<% Java 代码 %>。可以包含多个语句，如方法，变量，表达式等。
- 3、JSP 表达式：<%= Java 代码 %>。JSP 表达式中的代码会首先执行，然后转换成字符串并显示到网页上。
- 4、注释：HTML 注释格式，<!-- 注释内容 -->，该注释会在 jsp 源代码中显示；jsp 注释格式，<%-- 注释内容 --%>，不会在 jsp 源代码中显示；还可以使用 Java 注释，如“//”，同样不会在源代码中显示。

#### JSP 指令元素

1、page 指令：主要用来设定 JSP 页面的全局属性，语法格式：<%@ page 属性 1="属性值 1" 属性 2="属性值 2" %>。几个常用属性的作用如下：

- 1、language，用来声明所使用的脚本语言种类，一般为 java。
- 2、extends，用来指定该 JSP 页面生成的 Servlet 继承于哪个父类。一般不建议使用。
- 3、import，用来指定导入的 Java 包。默认导入的包括  
java.lang.\*;javax.servlet.\*;javax.servlet.jsp.\*;javax.servlet.http.\*;
- 4、session，指定该 JSP 页面中是否可以使用 session 对象。如果为 true，则可以使用。
- 5、buffer，指定输出流是否具有缓冲区，并设置大小。<%@page buffer="none|40KB"%>
- 6、autoFlush，指定缓冲区是否自动进行强制输出。
- 7、isThreadSafe，指定该 JSP 文件是否支持多线程使用。
- 8、contentType，指定该 JSP 文件的 MIME 格式，以及网页编码格式。
- 9、pageEncoding，指定网页的编码格式。

2、include 指令：用于在 JSP 文件中插入一个包含文件或代码的文件。只有当被包含的文件执行完成后，该 JSP 文件才会恢复执行。

语法格式为：<%@ include file="被包含文件地址" %>。

只是将页面静态的包含进来，如果被包含的文件中含有 JSP 代码，则会执行该代码，不会区分文件是动态还是静态。比如 txt 文件中含有 JSP 代码，则会执行代码，而不是将内容打出来。

3、taglib 指令：用来声明 JSP 文件使用了自定义标签，同时引用所指定的标签库并设置标签库的前缀。

语法格式：<%@ taglib prefix="tagPrefix" uri="URIToTagLibrary"%>

### JSP 动作元素

1、<jsp:include>动作元素，用来包含静态和动态的文件，若为静态，则只单纯的加到 JSP 页面中；若为动态，则会先进行处理，然后将处理的结果加到 JSP 页面中。

语法格式：<jsp:include page="包含文件 URL 地址" flush="true|false">

flush 属性用来指定缓冲区满时，是否进行清空。

2、<jsp:forward>动作元素，用来转移用户的请求，是服务器端跳转，用户的地址栏不发生变化。

语法格式：<jsp:forward page="URL 地址">

3、<jsp:param>动作元素，用来传递参数，一般与 include 和 forward 联合使用。

语法格式：<jsp:param name="" value=""/>

## Jsp 访问 javaBean 的语法

### 1.导入 javaBean 类

```
<%@ page import= "com.anllin.bean.Person" %>
```

### 2.声明 javaBean 对象

```
<jsp:useBean id= "person" class= "com.anllin.bean.Person"> </jsp:useBean>
```

其实相当于

```
<%  
  
    Person person = new Person();  
  
%>
```

Id 表示生成一个类的实例，一般 id 不能重复，用于表示不同的对象，如

果相同则表示同一个对象，这种情况下 jsp 会报错。

### 3.访问 javaBean 属性

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

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

其实相当于

```
<%  
  
    person.setName("jack");  
  
    person.getName();  
  
%>
```

### JSP 中 9 个隐式对象

隐式对象（或隐含变量）：在 JSP 当中我们没有手动声明创建，但实际存在，可以直接使用的对象。

```
final javax.servlet.jsp.PageContext pageContext;  
javax.servlet.http.HttpSession session = null;  
final javax.servlet.ServletContext application;  
final javax.servlet.ServletConfig config;  
javax.servlet.jsp.JspWriter out = null;  
final java.lang.Object page = this;  
// 还有 request 和 response 以及 exception 对象，一共 9 个
```

①request:客户端的请求信息被封装在 request 对象中,通过它才能了解用户的需求,然后做出响应

②response: 包含了响应客户请求的有关信息,但在 JSP 中使用很少

③pageContext:页面的上下文,是 PageContext 的一个对象,可以从该对象中获取到其他 8 个隐含对象,也可以获取到当前页面的其他信息

④session: 指的是客户端与服务器的一次会话,从客户端连到服务器的一个 WebApplication 开始,知道客户端与服务器断开连接为止

⑤application:代表当前 web 应用,是 ServletContext 对象,能实现用户间数据的共享,可存放全局变量,它开始于服务器的启动,直到服务器的关闭,在此期间,此对象一直存在;这样在用户的前后连接或不同用户之间的连接中,可以对此对象的同一属性进行操作;在任何地方对此对象属性的操作,都将影响到其他用户对此的访问。服务器的启动和关闭决定了 application 对象的生命周期

⑥config:当前 JSP 对应的 Servlet 的 ServletConfig 对象,可获取该 Servlet 的初始化参数(开发时基本不用),需要通过映射的地址才可以。

⑦out:JspWriter 对象,调用 out.println()可以直接把字符串打印到浏览器上

⑧page:page 对象就是指向当前 JSP 页面本身,类型为 Object,有点类似于类中的 this,几乎不使用

⑨exception:该对象是一个例外对象,只有页面是一个错误页面,即 isErrorPage 设置为 true 的时候(默认为 false)才能使用,否则无法编译。

**注意: JSP 可以放置在 WEB 应用程序中的除了 WEB-INF 及其子目录外的其他任何目录中**

### **JSP 模板元素**

JSP 页面中的静态 HTML 内容称之为 JSP 模版元素(比如 html,body 等等),在静态的 HTML 内容之中可以嵌套 JSP 的其他各种元素来产生动态内容和执行业务逻辑。

JSP 模版元素定义了网页的基本骨架,即定义了页面的结构和外观。

### JSP 表达式

JSP 表达式 (expression) 提供了将一个 java 变量或表达式的计算结果输出到客户端的简化方式, 它将要输出的变量或表达式直接封装在 `<%=` 和 `%>` 之中。

在 JSP 表达式中嵌套的变量或表达式后面不能有分号。

```
<body>
    <%
        Date date = new Date();
    %>

    <%= date %>
</body>
```

### JSP 脚本片断

像片段一样的 JSP 表达式, 嵌套在 `<%` 和 `%>` 中, 必须全部是符合 java 语法的语句。

```
<body>

    <%
        String ageStr = request.getParameter("age");
        int age = Integer.parseInt(ageStr);
        if(age > 18) {
    %>

        成年....

    <% }else{ %>

        未成年...

    <% } %>
```

```
</body>
```

```
<%--
```

```
Created by IntelliJ IDEA.
```

```
User: Administrator
```

```
Date: 2018/3/9
```

```
Time: 21:45
```

```
To change this template use File | Settings | File Templates.
```

```

--%>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<form action="index.jsp" method="post">
    <input type="text" name="age">
    <input type="submit">
</form>
</body>
</html>

```

### 页面间跳转的 2 种方式---请求转发和请求重定向（重点）

#### 一.请求转发

请求转发需要借助于一个接口，RequestDispatcher 接口，利用这个接口的 forward 方法实现请求转发。

#### 二.请求重定向

用 HttpServletResponse 的 sendRedirect 方法实现请求重定向。

请求转发

```

<body>
    <a href="forwardServlet">Forward</a>
</body>
@WebServlet("/forwardServlet")
public class ForwardServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        System.out.println("ForwardServlet's doGet");
        // 请求转发
        // 1. 调用 HttpServletRequest 的 getRequestDispatcher() 方法
        获得 RequestDispatcher 对象
        // 调用 getRequestDispatcher() 需要传入要转发的地址 (/代

```



表当前 WEB 应用的根目录)

```
RequestDispatcher requestDispatcher =  
request.getRequestDispatcher("/testServlet");  
// 2. 调用 HttpServletRequest 的 forward(request, response)  
进行请求转发
```

```
requestDispatcher.forward(request, response);  
}  
  
}  
@WebServlet("/testServlet")  
public class TestServlet extends HttpServlet {  
  
    protected void doGet(HttpServletRequest request,  
HttpServletResponse response) throws ServletException,  
IOException {  
        System.out.println("TestServlet's doGet 方法");  
    }  
  
}
```

运行 html, 点击查看控制台

```
信息: Starting ProtocolHandl  
一月 14, 2018 10:43:57 下午 org  
信息: Starting ProtocolHandl  
一月 14, 2018 10:43:57 下午 org  
信息: Server startup in 652  
ForwardServlet's doGet  
TestServlet's doGet方法
```

请求重定向

```
<body>  
    <a href="forwardServlet">Forward</a>  
    <a href="redirectServlet">Redirect</a>  
</body>  
@WebServlet("/redirectServlet")  
public class RedirectServlet extends HttpServlet {  
  
    protected void doGet(HttpServletRequest request,  
HttpServletResponse response) throws ServletException,  
IOException {
```

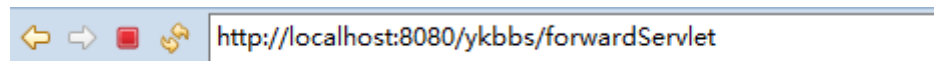
```
        System.out.println("RedirectServlet's doGet");
        response.sendRedirect("testServlet");
    }
}
```

再次运行 html, 点击 Redirect, 查看控制台

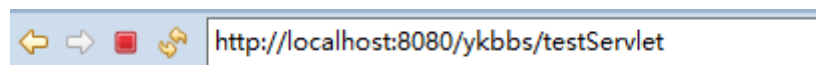
```
信息: Starting ProtocolHandle
一月 14, 2018 10:52:43 下午 org.
信息: Starting ProtocolHandle
一月 14, 2018 10:52:43 下午 org.
信息: Server startup in 729 m
RedirectServlet's doGet
TestServlet's doGet方法
```

貌似与请求转发没什么区别, 但实际区别非常之大。

观察用请求转发和请求重定向时地址栏的变化  
请求转发地址栏没变



请求重定向地址栏变化了



说明一个重要问题 (本质区别):

**请求转发只发出了一个请求**

**请求重定向发出了 2 个请求**

请求转发和请求重定向的目标可以是一个 servlet 也可以是一个 jsp  
`request.getRequestDispatcher("/XXServlet").forward(request, response)`

和

`response.sendRedirect("/XXServlet")` 只能在 `doGet` 或 `doPost` 方法中调用一次

也即, 在 `doGet` 或 `doPost` 方法中, 只能转发或重定向到一个页面, 不能

到多个页面

### 属性相关方法

①设置属性: void setAttribute(String name, Object o)

②获取指定属性: Object getAttribute(String name)

能够使用这些方法的对象有 4 个:

pageContext, request, session, application-->这 4 个对象也称为域对象

### 域对象（重点）

域:

2 个维度: 一份数据可以在多少个页面间共享, 可以在多长的时间范围内共享。

新建 ServletA 书写一下代码运行测试

```
@WebServlet("/aaa")
public class ServletA extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        javax.servlet.jsp.PageContext pageContext =
        javax.servlet.jsp.JspFactory.getDefaultFactory().getPageContext
        (this, request, response,
            null, true, 8192, true);
        pageContext.setAttribute("pageAttr", "pageValue");
        String strPageContext =
        (String)pageContext.getAttribute("pageAttr");
        System.out.println(strPageContext);

        request.setAttribute("requestAttr", "requestValue");

        HttpSession session = request.getSession();
```

```

        session.setAttribute("sessionAttr", "sessionValue");

request.getServletContext().setAttribute("applicationAttr", "app
licationValue");

request.getRequestDispatcher("/ccc").forward(request, response);

    }
}

```

ServletC

```

@WebServlet("/ccc")
public class ServletC extends HttpServlet {
    protected void doPost(HttpServletRequest request,
HttpServletRequest response) throws ServletException,
IOException {

        }

    protected void doGet(HttpServletRequest request,
HttpServletRequest response) throws ServletException,
IOException {
        System.out.println("ServletC");
        javax.servlet.jsp.PageContext pageContext =
javax.servlet.jsp.JspFactory.getDefaultFactory().getPageContext
(this, request, response,
            null, true, 8192, true);

        String strPageContext =
(String)pageContext.getAttribute("pageAttr");
        System.out.println(strPageContext);

        String strRequestContext =
(String)request.getAttribute("requestAttr");
        System.out.println(strRequestContext);

        String strSessionContext =
(String)request.getSession().getAttribute("sessionAttr");
        System.out.println(strSessionContext);
    }
}

```

```

        String strAppContext =
        (String)request.getServletContext().getAttribute("applicationAttr");

        System.out.println(strAppContext);
    }
}

ServletB

@WebServlet("/bbb")
public class ServletB extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        }

        protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
            javax.servlet.jsp.PageContext pageContext =
            javax.servlet.jsp.JspFactory.getDefaultFactory().getPageContext
            (this, request, response,
                null, true, 8192, true);
            pageContext.setAttribute("pageAttr", "pageValue");
            String strPageContext =
            (String)pageContext.getAttribute("pageAttr");
            System.out.println(strPageContext);

            request.setAttribute("requestAttr", "requestValue");

            HttpSession session = request.getSession();
            session.setAttribute("sessionAttr", "sessionValue");

            request.getServletContext().setAttribute("applicationAttr", "app
            licationValue");

            response.sendRedirect("/ccc");
        }
}

```

pageContext:属性的作用范围仅限于当前 servlet 或 JSP 页面

request:属性的作用范围仅限于同一个请求（考虑页面转发的情况）

session:属性的作用范围限于一次会话（浏览器打开直到关闭，称为一次会话，前提是在此期间会话没有失效），数据是用户独立的。（后面细讲）。

application:属性的作用范围限于当前 WEB 应用，是范围最大的属性作用范围，只要在一处设置属性，在其他各处的 JSP 或 Servlet 中都可以获取，直到服务器关闭。数据所有用户共享。

```
@WebServlet(name = "CountServlet",urlPatterns = "/count")
public class CountServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        Object obj = request.getSession().getAttribute("count");
        if(obj == null)
        {
            request.getSession().setAttribute("count",1);
        }
        else
        {
            int count = Integer.parseInt(obj.toString());
            count++;
            request.getSession().setAttribute("count",count);
        }

        response.sendRedirect("index.jsp");

    }
}

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
    <head>
        <title>$Title$</title>
```

	<pre>&lt;/head&gt; &lt;body&gt; &lt;a href="/count"&gt;加 1&lt;/a&gt;  \${count} &lt;/body&gt; &lt;/html&gt;</pre> <p>servlet 中通过 <code>getAttribute(String name)</code>，获取指定属性。而 jsp 中使用 EL 表达式。</p>
考核点	
学员问题汇总	
作业	
课后 总结分析	

东软睿道教案

章节	第五章 el&jstl
教学目标 与要求	通过本章学习，使学员掌握了解el与jsp区别，el内置对象，jstl

<p>重点 难点</p>	<p>重点：EL表达式，jstl核心标签库 难点：el内置对象</p>				
<p>教学内容设计 及学时分配</p>	<p><b>【教学进程安排】</b></p> <table border="0"> <tr> <td>1. el 表达式</td><td>4 课时</td></tr> <tr> <td>2.jstl 标签库</td><td>2 课时</td></tr> </table> <p><b>【课程主要内容】</b></p> <p><u>EL 技术：</u></p> <p><b>1.EL 表达式概述：</b></p> <p>EL (Express Lanuage) 表达式可以嵌入在 jsp 页面内部，减少 jsp 脚本的编写，EL 出现的目的是要替代 jsp 页面中脚本的编写。</p> <p><b>2.EL 从域中取出数据(EL 最重要的作用)：</b></p> <p>jsp 脚本：&lt;%=request.getAttribute(name)%&gt; EL 表达式替代上面的脚本：\${requestScope.name}</p> <p>EL 最主要的作用是获得四大域中的数据，格式\${EL 表达式} EL 获得 pageContext 域中的值：\${pageScope.key}; EL 获得 request 域中的值：\${requestScope.key}; EL 获得 session 域中的值：\${sessionScope.key}; EL 获得 application 域中的值：\${applicationScope.key};</p> <p>EL 从四个域中获得某个值\${key}; ——同样是依次从 pageContext 域, request 域, session 域, application 域中 获取属性，在某个域中获取后将不在向后寻找</p> <p><b>案例：</b></p> <ol style="list-style-type: none"> <li>1) 获得普通字符串</li> <li>2) 获得 User 对象的值</li> <li>3) 获得 List&lt;User&gt;的值</li> </ol> <pre>&lt;%     //存储字符串     request.setAttribute("company", "baidu");      //存储一个对象     User user = new User();     user.setId(1);</pre>	1. el 表达式	4 课时	2.jstl 标签库	2 课时
1. el 表达式	4 课时				
2.jstl 标签库	2 课时				



```

        user.setName("zhangsan");
        user.setPassword("123");
        session.setAttribute("user", user);

        //存储一个集合
        List<User> list = new ArrayList<User>();
        User user1 = new User();
        user1.setId(2);
        user1.setName("lisi");
        user1.setPassword("123");
        list.add(user1);
        User user2 = new User();
        user2.setId(3);
        user2.setName("wangwu");
        user2.setPassword("123");
        list.add(user2);
        application.setAttribute("list", list);
    %>

```

<!-- 使用 EL 表达式获得域中的值 -->

```

        ${requestScope.company }
        ${sessionScope.user.name }
        ${applicationScope.list[1].name}

```

<!-- 使用 el 表达式 全域查找 -->

```

        ${company }
        ${user.name }
        ${list[1].name}

```

### 3. EL 的内置对象(11 个) :

获取 JSP 中域中的数据:

```

pageScope;
requestScope;
sessionScope;
applicationScope。

```

接收参数:

```

param -----相当于
request.getParameter()
paramValues-----相当于
request.getParameterValues()

```

获取请求头信息:

```

header-----相当于

```

request.getHeader(name)  
headerValues

获取全局初始化参数:

initParam -----相当于  
this.getServletContext().getInitParameter(name)

WEB 开发中 cookie:

cookie-----相当于  
request.getCookies() ---cookie.getName() ---cookie.getValue()

WEB 开发中的 pageContext:

pageContext

`${pageContext.request.contextPath}`: 获得当前应用名称。

### JSTL 技术:

#### 1. JSTL 概述:

JSTL (JSP Standard Tag Library), JSP 标准标签库, 可以嵌入在 jsp 页面中使用标签的形式完成业务逻辑等功能。jstl 出现的目的同 el 一样也是要代替 jsp 页面中的脚本代码。JSTL 标准标准标签库有 5 个子库, 但随着发展, 目前常使用的是他的核心库:

STL 一共包含四大标签库:

- core: 核心标签库, 我们学习的重点;
- fmt: 格式化标签库
- sql: 数据库标签库
- xml: xml 标签库
- JSTL 函数标签库

JSTL 的核心标签库标签共 13 个, 使用这些标签能够完成 JSP 页面的基本功能, 减少编码工作。

从功能上可以分为 4 类: 表达式控制标签、流程控制标签、循环标签、URL 操作标签。

(1) **表达式控制标签**: out 标签、set 标签、remove 标签、catch 标签。

(2) **流程控制标签**: if 标签、choose 标签、when 标签、otherwise 标签。

(3) 循环标签: **forEach** 标签、**forEachTokens** 标签。

(4) URL 操作标签: **import** 标签、**url** 标签、**redirect** 标签、**param** 标签。

在 JSP 页面引入核心标签库的代码为: `<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>`

l18n 标签库

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

118N 格式标签库提供了 11 个标签, 这些 标签从功能上可以划分为 3 类如下:

1) 数字日期格式化。formatNumber 标签、formatData 标签、parseNumber 标签、parseDate 标签、timeZone 标签、setTimeZone 标签。

2) 读取消息资源。bundle 标签、message 标签、setBundle 标签。

3) 国际化。setlocale 标签、requestEncoding 标签。

## 2. JSTL 下载与导入:

JSTL 下载:

从 Apache 的网站下载 JSTL 的 JAR 包, 解压后, 在 lib 目录下可以看到两个 JAR 文件, 分别为 jstl.jar 和 standard.jar。

其中, jstl.jar 文件包含 JSTL 规范中定义的接口和相关类, standard.jar 文件包含用于实现 JSTL 的.class 文件以及 JSTL 中 5 个标签库描述符文件 (TLD)

将两个 jar 包导入我们工程的 lib 中;

使用 jsp 的 taglib 指令导入核心标签库;

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

## 3. JSTL 核心库的常用标签:

1) `<c:if test=" " >` 标签

其中 test 是返回 boolean 的条件

2) `<c:forEach>` 标签

使用方式有两种组合形式:

自定义函数库:

1、定义类和方法 (方法必须是 public static)

2、编写自定义 tld 文件, 并且将此文件放到 WEB-INF 或 WEB-INF 任意子目录下

3、在 jsp 中采用 taglib 指令引入自定义函数库

4、采用 前缀+冒号 (:)+函数名 调用即可

以下通过显示省份来看实现步骤:

第一步: 新建一个类如下:

UtilFunction.java

Java 代码

```
package demo;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
// 测试
```

```
// 自定义 JSTL 函数
```

```
public class UtilFunction {
```

```
    // 获取省份
```

```
    public static List getProvinces() {
```

```
        List provinces = new ArrayList();
```

```
        // 暂时添加几个测试
```

```
        provinces.add("广东省");
```

```
        provinces.add("广西省");
```

```
        provinces.add("山东省");
```

```
        provinces.add("四川省");
```

```
        provinces.add("江西省");
```

```
        return provinces;
```

```
    }
```

```
}
```

第二步：编写 tld 标签函数注册文件

myfunctions.tld

Xml 代码

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<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
```

```
http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
```

```
    version="2.0">
```

```
    <tlib-version>1.0</tlib-version>
```

```
    <short-name>my</short-name>
```

```
    <uri>http://www.changtusoft.cn/test/functions</uri> ( 其中 , 若 uri 为  
/WEB-INF/xxx.tld, 则无需再下面 tomcat 中注册)
```

```
    <!-- JSTL 自定义函数 获取省份 -->
```

```
    <function>
```

```
        <name>getProvinces</name>
```

```
        <function-class>demo.UtilFunction</function-class>
```

```
        <function-signature>java.util.List getProvinces()</function-signature>
```

</function>

</taglib>

第三步：在 web.xml 文件中注册 tld

Xml 代码

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<web-app version="2.5"
```

```
  xmlns="http://java.sun.com/xml/ns/javaee"
```

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

```
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
```

```
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

```
<welcome-file-list>
```

```
  <welcome-file>index.jsp</welcome-file>
```

```
</welcome-file-list>
```

<!-- 注册 JSTL 函数 -->

```
<jsp-config>
```

```
  <taglib>
```

```
    <taglib-uri>http://www.changtusoftware.cn/test/functions</taglib-uri>
```

```
    <taglib-location>/WEB-INF/myfunctions.tld</taglib-location>
```

```
  </taglib>
```

```
</jsp-config>
```

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

第四步：编写 jsp 进行测试

index.jsp

Java 代码

```
<% @ page language="java" import="java.util.*" pageEncoding="gbk"%>
```

```
<!-- 导入 jstl 标签库 -->
```

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

```
<!-- 导入自定义 jstl 函数 -->
```

```
<% @ taglib prefix="my" uri="http://www.changtusoftware.cn/test/functions" %>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

```
<html>
```

```
  <head>
```

```
    <title>自定义 JSTL 函数</title>
```

```
  </head>
```

```
  <body>
```

```
    省份:
```

```
    <select name="provinces">
```

```
      <option>--请选择省份--</option>
```

```
      <c:forEach items="{ my:getProvinces()}" var="p">
```

```
        <option>${p }</option>
```

```
</c:forEach>
</select>
</body>
</html>
```

部署例子到 tomcat 测试: [http://localhost:8080/jstl\\_functions/index.jsp](http://localhost:8080/jstl_functions/index.jsp)

#### 案例:

- 1) 遍历 List<String>的值
- 2) 遍历 List<User>的值
- 3) 遍历 Map<String, String>的值
- 4) 遍历 Map<String, User>的值

```
<%
    //模拟 List<String> strList
    List<String> strList = new ArrayList<String>();
    strList.add("a1");
    strList.add("b1");
    strList.add("c1");
    strList.add("d1");
    request.setAttribute("strList", strList);

    //遍历 List<User>的值
    List<User> userList = new ArrayList<User>();
    User user1 = new User();
    user1.setId(1);
    user1.setName("zhangsan");
    user1.setPassword("123");
    userList.add(user1);
    User user2 = new User();
    user2.setId(2);
    user2.setName("lisi");
    user2.setPassword("456");
    userList.add(user2);
    application.setAttribute("userList", userList);

    //遍历 Map<String, String>的值
    Map<String, String> strMap = new HashMap<String, String>();
    strMap.put("name", "lucy");
    strMap.put("age", "18");
    strMap.put("addr", "西三旗");
    strMap.put("email", "lucy@qq.com");
    session.setAttribute("strMap", strMap);
```

	<pre> //遍历 Map&lt;String, User&gt;的值 Map&lt;String, User&gt; userMap = new HashMap&lt;String, User&gt;(); userMap.put("user1", user1); userMap.put("user2", user2); request.setAttribute("userMap", userMap);  %&gt;  &lt;h1&gt;取出 strList 中的数据&lt;/h1&gt; &lt;c:forEach items="\${strList}" var="str"&gt;     \${str }&lt;br /&gt; &lt;/c:forEach&gt;  &lt;h1&gt;取出 userList 的数据&lt;/h1&gt; &lt;c:forEach items="\${userList}" var="user"&gt;     user 的 name: \${user.name }-----user 的 password:     \${user.password }&lt;br /&gt; &lt;/c:forEach&gt;  &lt;h1&gt;取出 strMap 的数据&lt;/h1&gt; &lt;c:forEach items="\${strMap}" var="entry"&gt;     \${entry.key }-----\${entry.value }&lt;br /&gt; &lt;/c:forEach&gt;  &lt;h1&gt;取出 userMap 的数据&lt;/h1&gt; &lt;c:forEach items="\${userMap}" var="entry"&gt;     \${entry.key }:      \${entry.value.name }-----\${entry.value.password }&lt;br /&gt; &lt;/c:forEach&gt; </pre>
考核点	
学员问题汇总	
作业	
课后 总结分析	

## 东软睿道教案

章节	第六章 监听器与过滤器													
教学目标与要求	通过本章节学习，使学生掌握监听器的使用，及触发监听的情框													
重点难点	重点：监听器与过滤器的使用 难点：监听器与过滤器的使用													
教学内容设计及学时分配	【教学进程安排】 监听器的生命周期与监听对象                    2 课时 过滤器的使用    2 课时													
	【课程主要内容】													
	1. 监听器 Listener													
	<p>监听器就是监听某个对象的的状态变化的组件。监听器的相关概念事件源：</p> <ul style="list-style-type: none"><li>被监听的对象（三个域对象 request，session，servletContext）</li><li>监听器：监听事件源对象， 事件源对象的状态的变化都会触发监听器 。</li><li>注册监听器：将监听器与事件源进行绑定。</li><li>响应行为：监听器监听到事件源的状态变化时，所涉及的功能代码（程序员编写代码）</li></ul> <p>按照被监听的对象划分：ServletRequest 域 ； HttpSession 域 ； ServletContext 域。按照监听的内容分：监听域对象的创建与销毁的； 监听域对象的属性变化的。</p> <table><tr><td></td><td>ServletContext域</td><td>HttpSession域</td><td>ServletRequest域</td></tr><tr><td>域对象的创建与销毁</td><td>ServletContextListener</td><td>HttpSessionListener</td><td>ServletRequestListener</td></tr><tr><td>域对象内的属性的变化</td><td>ServletContextAttributeListener</td><td>HttpSessionAttributeListener</td><td>ServletRequestAttributeListener</td></tr></table>				ServletContext域	HttpSession域	ServletRequest域	域对象的创建与销毁	ServletContextListener	HttpSessionListener	ServletRequestListener	域对象内的属性的变化	ServletContextAttributeListener	HttpSessionAttributeListener
	ServletContext域	HttpSession域	ServletRequest域											
域对象的创建与销毁	ServletContextListener	HttpSessionListener	ServletRequestListener											
域对象内的属性的变化	ServletContextAttributeListener	HttpSessionAttributeListener	ServletRequestAttributeListener											



## 三大域对象的创建与销毁的监听器

### 4.1.1. ServletContextListener

监听 ServletContext 域的创建与销毁的监听器，Servlet 域的生命周期：在服务器启动创建，服务器关闭时销毁；监听器的编写步骤：

- 编写一个监听器类去实现监听器接口
- 覆盖监听器的方法

ServletContextListener 监听器的主要作用：

初始化的工作：初始化对象；初始化数据。

例子：MyServletContextListener.java

```
@WebListener()
public class MyServletContextListener implements
ServletContextListener{

    @Override
    //监听 context 域对象的创建
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("context 创建了....");
    }

    //监听 context 域对象的销毁
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("context 销毁了....");
    }

}
```

HttpSessionListener

监听 HttpSession 域的创建与销毁的监听器。HttpSession 对象的生命周期：第一次调用 request.getSession 时创建；销毁有以下几种情况（服务器关闭、session 过期、手动销毁）

1、HttpSessionListener 的方法

```
/**
 * Created by yang on 2017/7/27.
```

```

    */
    public class listenerDemo implements HttpSessionListener {
        @Override
        public void sessionCreated(HttpSessionEvent
httpSessionEvent) {
            System.out.println("session 创建
"+httpSessionEvent.getSession().getId());
        }

        @Override
        public void sessionDestroyed(HttpSessionEvent
httpSessionEvent) {
            System.out.println("session 销毁");
        }
    }
}

```

创建 session 代码:

```

/**
 * Created by yang on 2017/7/24.
 */
public class SessionDemo extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException {
        doPost(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse resp) throws ServletException, IOException {

        request.getSession().setAttribute("code", "abc");

    }
}

```

当创建 session 时，监听器中的代码将执行。

ServletRequestListener

监听 ServletRequest 域创建与销毁的监听器。ServletRequest 的生命周期：每一次请求都会创建 request，请求结束则销毁。

1、ServletRequestListener 的方法

```

/**
 * Created by yang on 2017/7/27.
 */
public class RequestListenerDemo implements
ServletRequestListener {
    @Override
    public void requestDestroyed(ServletRequestEvent
ServletRequestEvent) {
        System.out.println("request 被销毁了");
    }

    @Override
    public void requestInitialized(ServletRequestEvent
ServletRequestEvent) {
        System.out.println("request 被创建了");
    }
}

```

只要客户端发起请求，监听器中的代码就会被执行。

监听三大域对象的属性变化的  
域对象的通用的方法

setAttribute(name, value)

触发添加属性的监听器的方法

触发修改属性的监听器的方法

removeAttribute(name)：触发删除属性的监听器的方法

ServletContextAttributeListener 监听器

```

/**
 * Created by yang on 2017/7/27.
 */
public class ServletContextAttrDemo implements
ServletContextAttributeListener {
    @Override
    public void attributeAdded(ServletContextAttributeEvent
scab) {
        //放到域中的属性
        System.out.println(scab.getName()); //放到域中的 name
        System.out.println(scab.getValue()); //放到域中的 value
    }

    @Override
    public void attributeRemoved(ServletContextAttributeEvent

```

```

scab) {
    System.out.println(scab.getName()); //删除的域中的 name
    System.out.println(scab.getValue()); //删除的域中的
value
}

@Override
public void attributeReplaced(ServletContextAttributeEvent
scab) {
    System.out.println(scab.getName()); //获得修改前的 name
    System.out.println(scab.getValue()); //获得修改前的
value
}
}

```

测试代码:

```

/**
 * Created by yang on 2017/7/27.
 */
public class ListenerTest extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequestRequest req,
HttpServletResponse resp) throws ServletException, IOException {
        ServletContext context=getServletContext();
        context.setAttribute("aaa","bbb");
        context.setAttribute("aaa","ccc");
        context.removeAttribute("aaa");
        doPost(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequestRequest req,
HttpServletResponse resp) throws ServletException, IOException {

    }
}

```

HttpSessionAttributeListener 监听器 (同上)

ServletRequestAttributeListener 监听器 (同上)

## 5. 与 session 中的绑定的对象相关的监听器（对象感知监听器）

### (1) 即将要被绑定到 session 中的对象有几种状态

绑定状态：就一个对象被放到 session 域中

解绑状态：就是这个对象从 session 域中移除了

钝化状态：是将 session 内存中的对象持久化（序列化）到磁盘

活化状态：就是将磁盘上的对象再次恢复到 session 内存中

绑定与解绑的监听器 HttpSessionBindingListener

```
public class Person implements HttpSessionBindingListener{

    private String id;
    private String name;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    //绑定的方法
    public void valueBound(HttpSessionBindingEvent event) {
        System.out.println("person 被绑定了");
    }
    @Override
    //解绑方法
    public void valueUnbound(HttpSessionBindingEvent event) {
        System.out.println("person 被解绑了");
    }
}
```

测试类：

```
public class TestPersonBindingServlet extends HttpServlet {
```

```

        protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {

            HttpSession session = request.getSession();

            //将 person 对象绑到 session 中
            Person p = new Person();
            p.setId("100");
            p.setName("zhangsanfeng");
            session.setAttribute("person", p);
            //将 person 对象从 session 中解绑
            session.removeAttribute("person");
        }

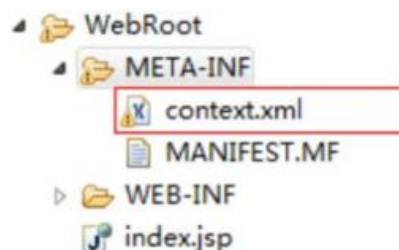
        protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
            doGet(request, response);
        }
    }

```

#### 4.1.2. 钝化与活化监听器

##### (3) 钝化与活化的监听器 HttpSessionActivationListener

可以通过配置文件 指定对象钝化时间 --- 对象多长时间不用被钝化  
在 META-INF 下创建一个 context.xml



#### 4.1.3.

##### 自定义 Customer 类

必须要实现: implements  
HttpSessionActivationListener, Serializable 这两个接口

```
package www.test.domian;
```

```
import java.io.Serializable;
```

```

import javax.servlet.http.HttpSessionActivationListener;
import javax.servlet.http.HttpSessionEvent;

public class Customer implements
HttpSessionActivationListener, Serializable{

    private String id;
    private String name;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    //钝化
    public void sessionWillPassivate(HttpSessionEvent se) {
        System.out.println("customer 被钝化了");
    }
    @Override
    //活化
    public void sessionDidActivate(HttpSessionEvent se) {
        System.out.println("customer 被活化了");
    }

}

```

配置文件 context.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <!-- maxIdleSwap:session 中的对象多长时间不使用就钝化，单位
分钟 -->
    <!-- directory:钝化后的对象的文件写到磁盘的哪个目录下 配置

```

钝化的对象文件在 work/catalina/localhost/钝化文件 -->

```
<Manager
className="org.apache.catalina.session.PersistentManager"
maxIdleSwap="1">
    <Store
className="org.apache.catalina.session.FileStore"
directory="webtest23" />
    </Manager>
</Context>
```

TestCustomerActiveServlet 测试钝化

```
package www.test.domian;
```

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```

```
public class TestCustomerActiveServlet extends HttpServlet {
```

```
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
```

```
        HttpSession session = request.getSession();
```

```
        //将 customer 放到 session 中
```

```
        Customer customer =new Customer();
```

```
        customer.setId("200");
```

```
        customer.setName("lucy");
```

```
        session.setAttribute("customer", customer);
```

```
        System.out.println("customer 被放到 session 域中了");
```

```
    }
```

```
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
```

```
        throws ServletException, IOException {
        doGet(request, response);
```

```
    }
```

```
}
```



TestCustomerActiveServlet2 测试活化

```
package www.test.domian;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class TestCustomerActiveServlet2 extends HttpServlet {

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        //从 session 域中获得 customer
        HttpSession session = request.getSession();
        Customer customer = (Customer)
session.getAttribute("customer");

        System.out.println(customer.getName());

    }
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

钝化后文件被保存的位置:

C:\Users\ttc\.IntelliJ IDEA2016.2\system\tomcat\Tomcat\_8\_0\_21\_markdownDemo\work\Catalina\localhost\ROOT\webtest23

#### **4.1.4. 1个月免登录**

index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
```

```
<title>Title</title>
</head>
<body>
欢迎您${userinfo.name}
<a href="LoginServlet.do?username=zhangsan">登录</a>
</body>
</html>
```

LoginServlet.java

```
@WebServlet(name = "LoginServlet",urlPatterns =
"/LoginServlet.do")
public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

        }

    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        String username = request.getParameter("username");
        Customer customer = new Customer();
        customer.setName(username);
        request.getSession().setAttribute("userinfo",customer);

        Cookie cookie = new
Cookie("JSESSIONID",request.getSession().getId());
        cookie.setMaxAge(60*60*24);
        response.addCookie(cookie);

    }
}
```

Customer.java

```
package www.test.domian;

import java.io.Serializable;

import javax.servlet.http.HttpSessionActivationListener;
import javax.servlet.http.HttpSessionEvent;
```

```

public class Customer implements
HttpSessionActivationListener, Serializable{

    private String id;
    private String name;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    //钝化
    public void sessionWillPassivate(HttpSessionEvent se) {
        System.out.println("customer 被钝化了");
    }
    @Override
    //活化
    public void sessionDidActivate(HttpSessionEvent se) {
        System.out.println("customer 被活化了");
    }
}

```

配置文件 context.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <!-- maxIdleSwap:session 中的对象多长时间不使用就钝化，单位
    分钟 -->
    <!-- directory:钝化后的对象的文件写到磁盘的哪个目录下 配置
    钝化的对象文件在 work/catalina/localhost/钝化文件 -->
    <Manager
    className="org.apache.catalina.session.PersistentManager"
    maxIdleSwap="1">

```

```
<Store  
className="org.apache.catalina.session.FileStore"  
directory="webtest23" />  
</Manager>  
</Context>
```

## 4.2. 文件上传

实现 web 开发中的文件上传功能，需完成如下两步操作：

1. 在 jsp 页面中添加上传输入项
2. 在 servlet 中读取上传文件的数据，并保存到服务器硬盘中

如何在 jsp 页面中添加上传输入项？

`<input type="file">` 标签用于在 jsp 页面中添加文件上传输入项，设置文件上传输入项时须注意：

必须要设置 input 输入项的 name 属性，否则浏览器将不会发送上传文件的数据。

必须把 form 的 enctype 属性值设为 multipart/form-data 。其实 form 表单在你不写 enctype 属性时，也默认为其添加了 enctype 属性值，默认值是 enctype="application/x-www-form-urlencoded" 设置该值后，浏览器在上传文件时，将把文件数据附带在 http 请求消息体中，并使用 MIME 协议对上传的文件进行描述，以方便接收方对上传数据进行解析和处理。

表单的提交方式必须是 post，因为上传文件可能较大。

get：以【明文】方式，通过 URL 提交数据，数据在 URL 中可以看到。提交数据最多不超过【2KB】。安全性较低，但效率比 post 方式高。适合提交数据量不大，且安全要求不高的数据：比如：搜索、查询等功能。

post：将用户提交的信息封装在 HTML HEADER 内，数据在 URL 中【不能看到】适合提交数据量大，安全性高的用户信息。如：注册、修改、上传等功能。

区别：

- post 隐式提交，get 显式提交。
- post 安全，get 不安全。

- get 提交数据的长度有限(255 字符之内), post 无限。

示例:

```
<form action="xx.action" method="post"
enctype="multipart/form-data">
</form>
```

#### 4.2.1. 第二步

如何在 Servlet 中读取文件上传数据, 并保存到本地硬盘中?

Request 对象提供了一个 `getInputStream` 方法, 通过这个方法可以读取到客户端提交过来的数据 (具体来说是 http 的请求体 entity)。但由于用户可能会同时上传多个文件, 在 Servlet 端编程直接读取上传数据, 并分别解析出相应的文件数据是一项非常麻烦的工作。

首先我们先写个简单的 JSP 页面, 代码如下:

```
<form action="/day20/upload" method="post" enctype="multipart/form-data">
  用户名 <input type="text" name="username"/><br/>
  上传文件 <input type="file" name="upload"/><br/>
  <input type="submit" name="submit" value="提交"/>
</form>
```

然后打开 Tomcat , 填写用户名, 选择上传一个 md 文件

在 Servlet 中的代码如下:

```
public class UploadServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");// 没法解决乱码, 因为这是针对
        URL 编码 解决乱码
        // request 提供 getInputStream 方法, 用来获得请求体信息
        InputStream in = request.getInputStream();
        int temp;
        while ((temp = in.read()) != -1) {
            System.out.write(temp);
        }
        System.out.flush();
        in.close();
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
```

```
throws ServletException, IOException {  
    doGet(request, response);  
}
```

```
}
```

点击提交按钮，在控制台可以看到输出内容：

## 过滤器

### 过滤器的生命周期

之前已经说过 Servlet 的生命周期，那么 Filter 的生命周期也就没有太大难度了！

1 **init(FilterConfig)**：在服务器启动时会创建 Filter 实例，并且每个类型的 Filter 只创建一个实例，从此不再创建！在创建完 Filter 实例后，会马上调用 init()方法完成初始化工作，这个方法只会被执行一次；

1 **doFilter(ServletRequest req,ServletResponse res,FilterChain chain)**：这个方法会在用户每次访问“目标资源(<url->pattern>index.jsp</url-pattern>)”时执行，如果需要“放行”，那么需要调用 FilterChain 的 doFilter(ServletRequest,ServletResponse)方法；

1 **destroy()**：服务器会在创建 Filter 对象之后，把 Filter 放到缓存中一直使用，通常不会销毁它。一般会在服务器关闭时销毁 Filter 对象，在销毁 Filter 对象之前，服务器会调用 Filter 对象的 destroy()方法。

### FilterConfig

你已经看到了吧，Filter 接口中的 init()方法的参数类型为 FilterConfig 类型。它的功能与 ServletConfig 相似，与 web.xml 文件中的配置信息对应。下面是 FilterConfig 的功能介绍：

1 **ServletContext getServletContext()**：获取 ServletContext 的方法；

1 **String getFilterName()**：获取 Filter 的配置名称；与<filter-name>元素对应；

1 **String getInitParameter(String name)**：获取 Filter 的初始化配置，与<init-param>元素对应；

1 **Enumeration getInitParameterNames()**：获取所有初始化参数的名称。

### FilterChain

doFilter()方法的参数中有一个类型为 FilterChain 的参数，它只有一个方法：  
doFilter(ServletRequest,ServletResponse)。

doFilter()方法的放行，让请求流访问目标资源！其实调用该方法的意思是，当前 Filter 放行了，但不代表其他过滤器也放行。一个目标资源上，可能部署了多个过滤器，所以调用 FilterChain 类的 doFilter()方法表示的是执行下一个过滤器的 doFilter()方法，或者是执行目标资源！

如果当前过滤器是最后一个过滤器，那么调用 chain.doFilter()方法表示执行目标资源，而不是最后一个过滤器，那么 chain.doFilter()表示执行下一个过滤器的 doFilter()方法。

### 多个过滤器执行顺序

一个目标资源可以指定多个过滤器，过滤器的执行顺序是在 web.xml 文件中的部署顺序：

```

<filter>
  <filter-name>myFilter1</filter-name>
  <filter-class>cn.cloud.filter.MyFilter1</filter-class>
</filter>
<filter-mapping>
  <filter-name>myFilter1</filter-name>
  <url-pattern>/index.jsp</url-pattern>
</filter-mapping>
<filter>
  <filter-name>myFilter2</filter-name>
  <filter-class>cn. cloud.filter.MyFilter2</filter-class>
</filter>
<filter-mapping>
  <filter-name>myFilter2</filter-name>
  <url-pattern>/index.jsp</url-pattern>
</filter-mapping>

```

```

public class MyFilter1 extends HttpFilter {
    public void doFilter(HttpServletRequest request, HttpServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        System.out.println("filter1 start...");
        chain.doFilter(request, response);//放行，执行 MyFilter2 的 doFilter()方法
        System.out.println("filter1 end...");
    }
}

```

```

public class MyFilter2 extends HttpFilter {
    public void doFilter(HttpServletRequest request, HttpServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        System.out.println("filter2 start...");
        chain.doFilter(request, response);//放行，执行目标资源
        System.out.println("filter2 end...");
    }
}

```

```

<body>
  This is my JSP page. <br>
  <h1>index.jsp</h1>
  <%System.out.println("index.jsp"); %>
</body>

```

当有用户访问 index.jsp 页面时，输出结果如下：

```
filter1 start...
```

```
filter2 start...
index.jsp
filter2 end...
filter1 end...
```

#### 四种拦截方式

写一个过滤器，指定过滤的资源为 `b.jsp`，然后在浏览器中直接访问 `b.jsp`，会发现过滤器执行了。但是，当在 `a.jsp` 中 `request.getRequestDispatcher("/b.jsp").forward(request,response)` 时，就不会再执行过滤器了！也就是说，默认情况下，只能直接访问目标资源才会执行过滤器，而 `forward` 执行目标资源，不会执行过滤器！

```
public class MyFilter extends HttpFilter {
    public void doFilter(HttpServletRequest request,
        HttpServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        System.out.println("myfilter...");
        chain.doFilter(request, response);
    }
}
```

```
<filter>
  <filter-name>myfilter</filter-name>
  <filter-class>cn.itcast.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/b.jsp</url-pattern>
</filter-mapping>
```

```
<body>
  <h1>b.jsp</h1>
</body>
```

```
<h1>a.jsp</h1>
<%
    request.getRequestDispatcher("/b.jsp").forward(request, response);
%>
</body>
```

`http://localhost:8080/filtertest/b.jsp`--> 直接访问 `b.jsp` 时，会执行过滤器内容；  
`http://localhost:8080/filtertest/a.jsp`--> 访问 `a.jsp`，但 `a.jsp` 会 `forward` 到 `b.jsp`，这时就不会执行过滤器！



过滤器有四种拦截方式！分别是：REQUEST、FORWARD、INCLUDE、ERROR。

1 REQUEST：直接访问目标资源时执行过滤器。包括：在地址栏中直接访问、表单提交、超链接、重定向，只要在地址栏中可以看到目标资源的路径，就是 REQUEST；

1 FORWARD：转发访问执行过滤器。包括 RequestDispatcher#forward() 方法、<jsp:forward>标签都是转发访问；

1 INCLUDE：包含访问执行过滤器。包括 RequestDispatcher#include() 方法、<jsp:include>标签都是包含访问；

1 ERROR：当目标资源在 web.xml 中配置为<error-page>中时，并且真的出现了异常，转发到目标资源时，会执行过滤器。

可以在<filter-mapping>中添加 0~n 个<dispatcher>子元素，来说明当前访问的拦截方式。

```
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/b.jsp</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

```
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/b.jsp</url-pattern>
</filter-mapping>
```

```
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/b.jsp</url-pattern>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

最为常用的就是 REQUEST 和 FORWARD 两种拦截方式，而 INCLUDE 和 ERROR 都比较少用！其中 INCLUDE 比较好理解，ERROR 方式不易理解，下面给出 ERROR 拦截方式的例子：

```
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/b.jsp</url-pattern>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
<error-page>
  <error-code>500</error-code>
  <location>/b.jsp</location>
</error-page>
```

```
<body>
<h1>a.jsp</h1>
<%
if(true)
throw new RuntimeException("嘻嘻~");
%>
</body>
```

过滤器的应用场景

过滤器的应用场景：

1 执行目标资源之前做预处理工作，例如设置编码，这种试通常都会放行，只是在目标资源执行之前做一些准备工作；

1 通过条件判断是否放行，例如校验当前用户是否已经登录，或者用户 IP 是否已经被禁用；

1 在目标资源执行后，做一些后续的特殊处理工作，例如把目标资源输出的数据进行处理；

设置目标资源

在 web.xml 文件中部署 Filter 时，可以通过“\*”来执行目标资源：

```
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

特性与 Servlet 完全相同！通过这一特性，可以在用户访问敏感资源时，执行过滤器，例如：<url-pattern>/admin/\*</url-pattern>，可以把所有管理员才能访问的资源放到 /admin 路径下，这时可以通过过滤器来校验用户身份。

还可以为<filter-mapping>指定目标资源为某个 Servlet，例如：

```
<servlet>
  <servlet-name>myservlet</servlet-name>
  <servlet-class>cn.cloud.servlet.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/abc</url-pattern>
</servlet-mapping>
<filter>
  <filter-name>myfilter</filter-name>
  <filter-class>cn.cloud.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <servlet-name>myservlet</servlet-name>
```

	<div>&lt;/filter-mapping&gt;</div> <p>当用户访问 <code>http://localhost:8080/filtertest/abc</code> 时，会执行名字为 <code>myservlet</code> 的 <code>Servlet</code>，这时会执行过滤器。</p> <p>Filter 小结</p> <p>Filter 的三个方法：</p> <ul style="list-style-type: none"><li>1 <code>void init(FilterConfig)</code>：在 Tomcat 启动时被调用；</li><li>1 <code>void destroy()</code>：在 Tomcat 关闭时被调用；</li><li>1 <code>void doFilter(ServletRequest,ServletResponse,FilterChain)</code>：每次有请求时都调用该方法；</li></ul> <p><code>FilterConfig</code> 类：与 <code>ServletConfig</code> 相似，用来获取 Filter 的初始化参数</p> <ul style="list-style-type: none"><li>1 <code>ServletContext getServletContext()</code>：获取 <code>ServletContext</code> 的方法；</li><li>1 <code>String getFilterName()</code>：获取 Filter 的配置名称；</li><li>1 <code>String getInitParameter(String name)</code>：获取 Filter 的初始化配置，与 <code>&lt;init-param&gt;</code> 元素对应；</li><li>1 <code>Enumeration getInitParameterNames()</code>：获取所有初始化参数的名称。</li></ul> <p><code>FilterChain</code> 类：</p> <ul style="list-style-type: none"><li>1 <code>void doFilter(ServletRequest,ServletResponse)</code>：放行！表示执行下一个过滤器，或者执行目标资源。可以在调用 <code>FilterChain</code> 的 <code>doFilter()</code> 方法的前后添加语句，在 <code>FilterChain</code> 的 <code>doFilter()</code> 方法之前的语句会在目标资源执行之前执行，在 <code>FilterChain</code> 的 <code>doFilter()</code> 方法之后的语句会在目标资源执行之后执行。</li></ul> <p>四各拦截方式：REQUEST、FORWARD、INCLUDE、ERROR，默认是 REQUEST 方式。</p> <ul style="list-style-type: none"><li>1 REQUEST：拦截直接请求方式；</li><li>1 FORWARD：拦截请求转发方式；</li><li>1 INCLUDE：拦截请求包含方式；</li><li>ERROR：拦截错误转发方式。</li></ul>
考核点	
学员问题汇总	
作业	
课后 总结分析	

## 东软睿道教案

章节	第七章 其它 Web 开发技术
教学目标 与要求	通过本章节学习，使学生掌握文件上传与下载，数据库连接池，分页查询
重点 难点	重点： 文件上传与下载，数据库连接池，分页查询 难点： 监听器与过滤器的使用
教学内容设计 及学时分配	<p><b>【教学进程安排】</b></p> <p>监听器的生命周期与监听对象                      2 课时</p> <p>文件上传下载    2 课时</p> <p><b>【课程主要内容】</b></p> <p style="text-align: center;"><b>4.3. 文件上传</b></p> <p>实现 web 开发中的文件上传功能，需完成如下两步操作：</p> <ol style="list-style-type: none"> <li>3. 在 jsp 页面中添加上传输入项</li> <li>4. 在 servlet 中读取上传文件的数据，并保存到服务器硬盘中</li> </ol> <p>如何在 jsp 页面中添加上传输入项？</p> <p>&lt;input type="file"&gt;标签用于在 jsp 页面中添加文件上传输入项，设置文件上传输入项时须注意：</p> <p>必须要设置 input 输入项的 name 属性，否则浏览器将不会发送上传文件的数据。</p> <p>必须把 form 的 enctype 属性值设为 multipart/form-data 。其实 form 表单在你不写 enctype 属性时，也默认为其添加了 enctype 属性</p>

值，默认值是 `enctype="application/x-www-form-urlencoded"` 设置该值后，浏览器在上传文件时，将把文件数据附带在 http 请求消息体中，并使用 MIME 协议对上传的文件进行描述，以方便接收方对上传数据进行解析和处理。

表单的提交方式必须是 post，因为上传文件可能较大。

get：以【明文】方式，通过 URL 提交数据，数据在 URL 中可以看到。提交数据最多不超过【2KB】。安全性较低，但效率比 post 方式高。适合提交数据量不大，且安全要求不高的数据：比如：搜索、查询等功能。

post：将用户提交的信息封装在 HTML HEADER 内，数据在 URL 中【不能看到】适合提交数据量大，安全性高的用户信息。如：注册、修改、上传等功能。

区别：

- post 隐式提交，get 显式提交。
- post 安全，get 不安全。
- get 提交数据的长度有限(255 字符之内)，post 无限。

示例：

```
<form action="xx.action" method="post"
enctype="multipart/form-data">
</form>
```

### 4.3.1. 第二步

如何在 Servlet 中读取文件上传数据，并保存到本地硬盘中？

Request 对象提供了一个 `getInputStream` 方法，通过这个方法可以读取到客户端提交过来的数据（具体来说是 http 的请求体 entity）。但由于用户可能会同时上传多个文件，在 Servlet 端编程直接读取上传数据，并分别解析出相应的文件数据是一项非常麻烦的工作。

首先我们先写个简单的 JSP 页面，代码如下：

```
<form action="/day20/upload" method="post" enctype="multipart/form-data">
  用户名 <input type="text" name="username"/><br/>
  上传文件 <input type="file" name="upload"/><br/>
  <input type="submit" name="submit" value="提交"/>
</form>
```

然后打开 Tomcat，填写用户名，选择上传一个 md 文件

在 Servlet 中的代码如下：

```
public class UploadServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");// 没法解决乱码，因为这是针对
        URL 编码 解决乱码
        // request 提供 getInputStream 方法，用来获得请求体信息
        InputStream in = request.getInputStream();
        int temp;
        while ((temp = in.read()) != -1) {
            System.out.write(temp);
        }
        System.out.flush();
        in.close();
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }

}
```

点击提交按钮，在控制台可以看到输出内容：

采用文件流的方式实现文件下载

采用文件流输出的方式下载

Jsp 代码

```
<% @page language="java" contentType="application/x-msdownload" pageEncoding="gb2312"%>
```

```
<%
```

```
//关于文件下载时采用文件流输出的方式处理：
```

```
//加上 response.reset()，并且所有的%>后面不要换行，包括最后一个；
```

```
response.reset();//可以加也可以不加
```

```
response.setContentType("application/x-download");
```

```
//application.getRealPath("/main/mvplayer/CapSetup.msi");获取的物理路径
```

```
String filedownload = "想办法找到要提供下载的文件物理路径+文件名";
```

```
String filedisplay = "给用户提供的下载文件名";
```

```
String filedisplay = URLEncoder.encode(filedisplay,"UTF-8");
```

```
response.addHeader("Content-Disposition","attachment;filename=" + filedisplay);
```

```
java.io.OutputStream outp = null;
```

```

java.io.FileInputStream in = null;
try {
    outp = response.getOutputStream();
    in = new FileInputStream(filenamedownload);
    byte[] b = new byte[1024];
    int i = 0;
    while((i = in.read(b)) > 0) {
        outp.write(b, 0, i);
    }
    outp.flush();
    //要加以下两句话，否则会报错
    //java.lang.IllegalStateException: getOutputStream() has already been called for //th
is response
    out.clear();
    out = pageContext.pushBody();
} catch(Exception e){
    System.out.println("Error!");
    e.printStackTrace();
}finally{
    if(in != null){
        in.close();
        in = null;
    }
    //这里不能关闭
    //if(outp != null) {
        //outp.close();
        //outp = null;
    //}
}
%>

```

## 数据库连接池技术

数据库连接池（connection pool）

数据库连接池简单介绍

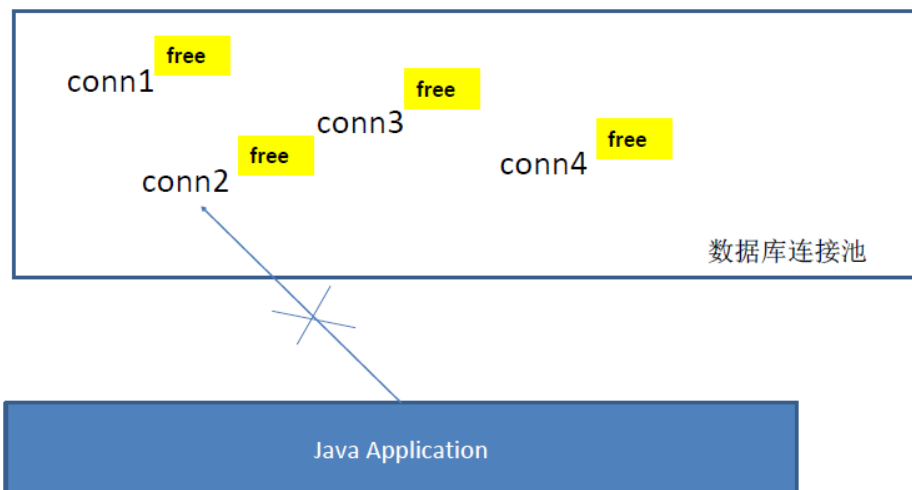
为解决传统开发中的数据库连接问题，可以采用数据库连接池技术。

数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。

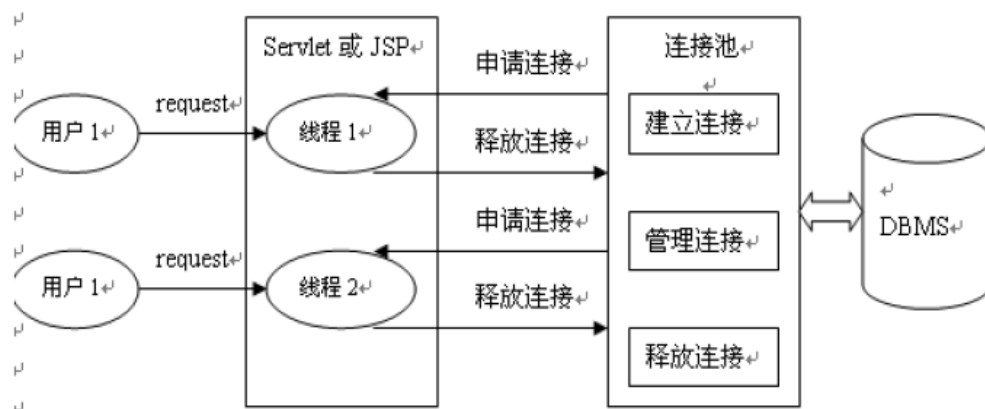
数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是重新建立一个。

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数来设定的。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大

连接数量时，这些请求将被加入到等待队列中。



数据库连接池工作原理：



数据库连接池技术的优点

资源重用：

①由于数据库连接得以重用，避免了频繁创建，释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增加了系统运行环境的平稳性。

更快的系统反应速度：

数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于连接池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而减少了系统的响应时间

新的资源分配手段：

对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接池的配置，实现某一应用最大可用数据库连接数的限制，避免某一应用独占所有的数据库资源

统一的连接管理，避免数据库连接泄露：

在较为完善的数据库连接池实现中，可根据预先的占用超时设定，强制回收被占用连接，从而避免了常规数据库连接操作中可能出现的资源泄露

三、两种开源的数据库连接池

JDBC 的数据库连接池使用 `javax.sql.DataSource` 来表示，`DataSource` 只是一个接口，该接口通常由服务器(Weblogic, WebSphere, Tomcat)提供实现，也有一些开



源组织提供实现：

①DBCP 数据库连接池

②C3P0 数据库连接池

DataSource 通常被称为数据源，它包含连接池和连接池管理两个部分，习惯上也经常把 DataSource 称为连接池

数据源和数据库连接不同，数据源无需创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。

当数据库访问结束后，程序还是像以前一样关闭数据库连接：conn.close(); 但上面的代码并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给了数据库连接池。

DBCP 数据源

DBCP 是 Apache 软件基金组织下的开源连接池实现，该连接池依赖该组织下的另一个开源系统：Common-pool.如需使用该连接池实现，应在系统中增加如下两个 jar 文件：

①Commons-dbc.jar：连接池的实现

②Commons-pool.jar：连接池实现的依赖库

Tomcat 的连接池正是采用该连接池来实现的。该数据库连接池既可以与应用服务器整合使用，也可由应用程序独立使用。

使用范例：

```
1    /**
2    * 使用 DBCP 数据库连接池
3    * 1. 加入 jar 包(2 个 jar 包). 依赖于 Commons Pool
4    * 2. 创建数据库连接池
5    * 3. 为数据源实例指定必须的属性
6    * 4. 从数据源中获取数据库连接
7    *
8    * @throws SQLException
9    */
10   @Test
11   public void testDBCP() throws SQLException {
12       final BasicDataSource dataSource = new BasicDataSource();
13
14       // 2. 为数据源实例指定必须的属性
15       dataSource.setUsername("root");
16       dataSource.setPassword("");
17       dataSource.setUrl("jdbc:mysql:///jdbc?useSSL=false");
18       dataSource.setDriverClassName("com.mysql.jdbc.Driver");
19
20       // 3. 指定数据源的一些可选的属性.
21       // 1). 指定数据库连接池中初始化连接数的个数
22       dataSource.setInitialSize(5);
23
24       // 2). 指定最大的连接数：同一时刻可以同时向数据库申请的连接数
25       dataSource.setMaxActive(5);
```

```
26
27 // 3). 指定小连接数: 在数据库连接池中保存的最少的空闲连接的数量
28 dataSource.setMinIdle(2);
29
30 // 4).等待数据库连接池分配连接的最长时间. 单位为毫秒. 超出该时间将抛出异常.
31 dataSource.setMaxWait(1000 * 5);
32
33 // 4. 从数据源中获取数据库连接
34 Connection connection = dataSource.getConnection();
35 System.out.println(connection.getClass());
36
37 connection = dataSource.getConnection();
38 System.out.println(connection.getClass());
39
40 connection = dataSource.getConnection();
41 System.out.println(connection.getClass());
42
43 connection = dataSource.getConnection();
44 System.out.println(connection.getClass());
45
46 Connection connection2 = dataSource.getConnection();
47 System.out.println(">" + connection2.getClass());
48
49 new Thread() {
50     public void run() {
51         Connection conn;
52         try {
53             conn = dataSource.getConnection();
54             System.out.println(conn.getClass());
55         } catch (SQLException e) {
56             e.printStackTrace();
57         }
58     };
59 }.start();
60
61 try {
62     Thread.sleep(3000);
63 } catch (InterruptedException e) {
64     e.printStackTrace();
65 }
66
67 connection2.close();
68 }
```

使用配置文件的方式:

```
1  /**
2      * 1. 加载 dbcp 的 properties 配置文件: 配置文件中的键需要来自
BasicDataSource 的属性.
3      * 2. 调用 BasicDataSourceFactory 的 createDataSource 方法创建
DataSource 实例
4      * 3. 从 DataSource 实例中获取数据库连接.
5  */
6  @Test
7  public void testDBCPWithDataSourceFactory() throws Exception {
8
9      Properties properties = new Properties();
10         InputStream inStream =
JDBCTest.class.getClassLoader().getResourceAsStream("dbcp.properties");
11     properties.load(inStream);
12
13     DataSource dataSource = BasicDataSourceFactory.createDataSource(properties);
14
15     System.out.println(dataSource.getConnection());
16
17     // BasicDataSource basicDataSource =
18     // (BasicDataSource) dataSource;
19     //
20     // System.out.println(basicDataSource.getMaxWait());
21 }
```

注意: 这里使用配置文件的方式应遵循 Java EE 的 setter 和 getter 规范, 不能随便命名:

dbcp.properties 配置文件 (属性的意义和第一种方式对应):

username=root

password=password

driverClassName=com.mysql.jdbc.Driver

#这里 MySQL 高版本驱动需要设置 useSSL=false 或 true, 否则控制台会有警告

url=jdbc:mysql://localhost:3306/jdbc?useUnicode=true&characterEncoding=UTF-8&useSSL=false

initialSize=10

maxActive=50

minIdle=5

maxWait=5000

C3P0 数据源

一般方式创建数据库连接池:

```
1  /**
```

```

2      * 使用 set 方法为属性赋值
3      */
4      @Test
5      public void testC3P0() throws Exception {
6          ComboPooledDataSource cpds = new ComboPooledDataSource();
7          cpds.setDriverClass("com.mysql.jdbc.Driver"); // loads the jdbc driver
8          cpds.setJdbcUrl("jdbc:mysql:///jdbc?useSSL=false");
9          cpds.setUser("root");
10         cpds.setPassword("");
11
12         System.out.println(cpds.getConnection());
13     }

```

使用 xml 配置文件：

①Java 代码：

```

1      /**
2          * 1. 创建 c3p0-config.xml 文件，参考帮助文档中 Appendix B:
Configuration Files 的内容
3          * 2. 创建 ComboPooledDataSource 实例；
4          *      DataSource dataSource = new ComboPooledDataSource("helloc3p0");
5          * 3. 从 DataSource 实例中获取数据库连接.
6          */
7          @Test
8          public void testC3poWithConfigFile() throws Exception {
9              DataSource dataSource = new ComboPooledDataSource("helloc3p0");
10
11             System.out.println(dataSource.getConnection());
12
13             ComboPooledDataSource      comboPooledDataSource      =
(ComboPooledDataSource) dataSource;
14             System.out.println(comboPooledDataSource.getMaxStatements());
15         }

```

②C3P0-config.xml 配置文件：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <c3p0-config>
3
4     <named-config name="helloc3p0">
5
6         <!-- 指定连接数据源的基本属性 -->
7         <property name="user">root</property>
8         <property name="password"></property>
9         <property name="driverClass">com.mysql.jdbc.Driver</property>
10        <property name="jdbcUrl">jdbc:mysql:///jdbc?useSSL=false</property>
11
12        <!-- 若数据库中连接数不足时，一次向数据库服务器申请多少个连接

```

```

-->
13     <property name="acquireIncrement">5</property>
14     <!-- 初始化数据库连接池时连接的数量 -->
15     <property name="initialPoolSize">5</property>
16     <!-- 数据库连接池中的最小的数据库连接数 -->
17     <property name="minPoolSize">5</property>
18     <!-- 数据库连接池中的最大的数据库连接数 -->
19     <property name="maxPoolSize">10</property>
20
21     <!-- C3P0 数据库连接池可以维护的 Statement 的个数 -->
22     <property name="maxStatements">20</property>
23     <!-- 每个连接同时可以使用的 Statement 对象的个数 -->
24     <property name="maxStatementsPerConnection">5</property>
25
26 </named-config>
27
28 </c3p0-config>

```

## 分页

### 一、 分页的 SQL 语句:

```
select * from (selectt.*,row_number() over(order by ROWNUM ASC) rn from goods t)
where rn>=? andrn<=?
```

Ø goods : 为表名, 数据从 goods 中查得

Ø 第一个? : 起始行号

Ø 第二个? : 终点行号

### 二、 分页工具类 Pagenation 的细节:

**Pagenation 类:** 类似 JavaBean 的工具类, 封装了分页后的细节,  
主要作用: 封装后台分页细节, 前台调用分页细节。

简介: 实例化时需要传入 3 个参数, 然后类内部计算得到所有分页细节

注意: 后期还需要调用 setList()方法将分页结果也存入其中。

用法: 后台用此类封装分页结果, request 给前台, 前台展示。

传入参数: 当前页数(pageNum)+每页大小(size)+记录总行数(rowCount)

### 三、 分页原理:

1. 后台 sevlet 根据参数调用分页 service()方法, 返回值一个 Pagenation 类的对象, 封装了当前要求的分页信息;
2. service()方法处理过程: 根据传入的三个参数, 初始化 Pagenation 类, 然后调用 dao 包下的分页查询类, 返回值为一个 List, 将这个 List 也存入 Pagenation 对象中, 返回返回该对象;
3. 返回的 Pagenation 对象包含了有关分页所有内容, 将此对象放在 request 中传到前台, 跳转到前台;
4. 在前台中取出 Pagenation 的相关信息, 此时的 Pagenation 对象中存放着分页信

息，用 EL 和 JSTL 显示分页信息。

#### 四、 代码实现：

##### 1. 分页工具类(核心中的核心类)：

##### 2. 其他代码：

###### (1)、前端展示代码：

```
package com.test.util;
```

```
import java.util.List;
```

```
/**
```

```
 * Pagenation 分页工具类：用于数据的分页  
 * 根据传入的 pageNum 参数来确定从哪页开始分页  
 * 根据传入的 size 参数确定每页分多少条记录  
 * 根据传入的 rowCount 参数来确定数据总条数  
 *  
 */
```

```
public class Pagenation {
```

```
/* 指定的参数 */
```

```
private int pageNum; //当前页号
```

```
private int size; //页面大小：每页显示多少条数据
```

```
/* DB 中查找的数据 */
```

```
private long rowCount; //数据总条数：共有多少条数据
```

```
private List list; //数据内容
```

```
/* 由以上属性计算得到的属性 */
```

```
private int pageCount; //页面总数
```

```
private int startRow; //当前页面开始行，第一行为第 0 行
```

```
private int first = 1; //第一页 页号
```

```
private int last; //最后一页 页号
```

```
private int prev; //前一页 页号
```

```
private int next; //后一页 页号
```

```
private int startNav; //导航栏 起始页号
```

```
private int endNav; //导航栏 末尾页号
```

```
private int navCount = 10; //导航栏长度 页号显示数量，最多显示 numCount+1 条，  
这里显示 11 页
```

```
/**
```

```
 * 构造器：初始化基本的三个参数(pageNum, size, rowCount)，其他参数由计算得到  
 */
```

```
public Pagenation(int pageNum, int size, long rowCount) {
```

```
 //初始化基本参数
```

```

this.pageNum = pageNum;
this.size = size;
this.rowCount = rowCount;

//计算得到其他参数
this.pageCount = (int)Math.ceil(this.rowCount/(double)size);
this.last = pageCount;

this.pageNum = Math.min(pageNum, pageCount); //一般情况下 pageNum 会等于传入
的 pageNum,但当传入的 pageNum 如果大于页面总数,则 pageNum 就等于最大页面
数,即最后一页的页数
this.pageNum = Math.max(1, this.pageNum); //一般情况下 pageNum 会等于传入的
pageNum,但当传入的 pageNum 如果小于 1,则 pageNum 就等于 1,即第一页的页数

this.startRow = pageNum*size - (size-1);

this.prev = (this.pageNum-1>1)?(this.pageNum-1):1; //如果<前一页>为第一页,则显
示 1,否则是{本页页数-1}
this.next = (this.pageNum+1<this.pageCount)?(this.pageNum+1):this.pageCount; //如果
<后一页>为最后一页,则返回{最后一页页数}, 否则返回{本页数+1}

//导航处理
// this.startNav = (this.pageNum-5>1)?(this.pageNum-5):1; //导航来开始按钮为{当前
页数-5}, 如果当前页数不足 5, 则导航栏开始按钮为{第 1 页}按钮
this.startNav =
(this.pageNum-(this.navCount/2)>1)?(this.pageNum-(this.navCount/2)):1; //优化后的开
始按钮, 使当前页位于导航栏中间, 开始按钮已不是{当前页数-5}, 而是{当前页数
-导航栏总长度的一半}
this.endNav = (this.startNav+navCount<this.last)?(this.startNav+navCount):this.last; //
导航栏结束按钮为{开始页数+导航栏长度},如果{开始页数+导航栏长度}超过页面
总数, 则结束按钮为{最后一页}
}

```

#### 批量删除

1.页面上怎样将数据提交到服务器端.

1.1.可以创建一个表单, 将表单数据提交就可以。

1.2.直接使用 js 操作

需要手动拼接出 url 路径

2.在服务器端怎样批量删除.

2.1.得到所有要删除的 id 值

```
request.getParameterValues("ck");
```

2.2.在 dao 中使用 QueryRunner 的 batch 方法

```
batch(sql,Object[][]);
```

注意:参数二维数据, 它代表的是每一条 sql 的参数。

	$\{1,2,3\} \rightarrow \{\{1\},\{2\},\{3\}\}$
考核点	
学员问题汇总	
作业	
课后 总结分析	

东软睿道教案

章节	第 8 章 MVC 模式
教学目标 与要求	通过本章学习，使学员掌握同步与异步区别，利用ajax引擎实现异步传输，mvc设计模式。
重点 难点	重点：MVC设计模式， 难点：mvc模式的设计



## 【教学进程安排】

Mvc 设计模式

2 课时

## 【课程主要内容】

MVC 模式:

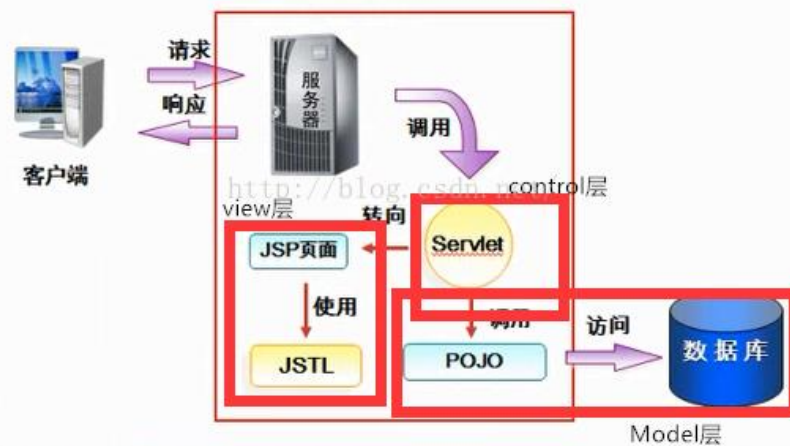
1.MVC: ---- web 开发的设计模式

M: Model---模型 javaBean: 封装数据

V: View-----视图 jsp: 单纯进行页面的显示

C: Controller----控制器 Servlet: 获取数据--对数据进行封装--传递数据-- 指派显示的 jsp 页面

### MVC请求过程



2.javaEE 的三层架构:

web 层: 与客户端交互

service 层: 复杂业务处理

dao 层: 与数据库进行交互

开发实践时 三层架构通过包结构体现

三层架构案例:

使用包结构来实现分层:

web 层:

ProductListServlet.java 代码如下:

```
package com.zl.web;  
import java.io.IOException;  
import java.sql.SQLException;  
import java.util.List;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import com.zl.domain.Product;  
import com.zl.service.ProductService;
```

教学内容设计  
及学时分配

```

public class ProductListServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        //传递请求到 service 层
        ProductService service = new ProductService();
        List<Product> productList = null;
        try {
            productList = service.findAllProduct();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        //全部的商品数据准备好了，转发给 jsp 进行展示
        request.setAttribute("productList", productList);
        request.getRequestDispatcher("/product_list.jsp").forward(request, response);

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        doGet(request, response);
    }
}

service 层:
ProductService.java 代码如下:
package com.zl.service;
import java.sql.SQLException;
import java.util.List;
import com.zl.dao.ProductDao;
import com.zl.domain.Product;

public class ProductService {

    public List<Product> findAllProduct() throws SQLException {
        //没有复杂业务
        //传递请求到 dao 层
        ProductDao dao = new ProductDao();
        List<Product> productList = dao.findAllProduct();
        return productList;
    }
}

```

dao 层:

ProductDao.java 代码如下:

```
package com.zl.dao;
import java.sql.SQLException;
import java.util.List;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanListHandler;
import com.zl.domain.Product;
import com.zl.utils.DataSourceUtils;

public class ProductDao {

    public List<Product> findAllProduct() throws SQLException {
        // 操作数据库
        QueryRunner runner = new QueryRunner(DataSourceUtils.getDataSource());
        String sql = "select * from product";
        List<Product> productList = runner.query(sql, new
        BeanListHandler<Product>(Product.class));
        return productList;
    }
}
```

product\_list.jsp 部分代码如下:

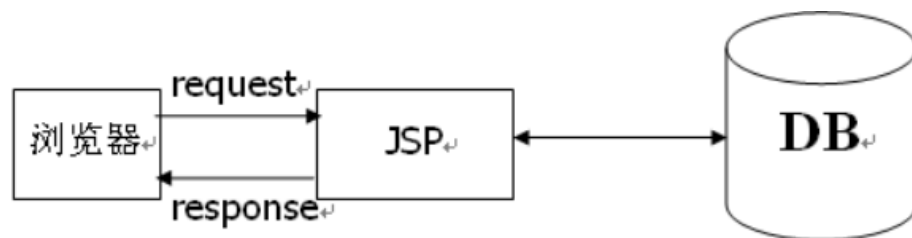
```
<c:forEach items="${productList}" var="product">
    <div class="col-md-2" style="height:250px">
        <a href="product_info.htm">
            
        </a>
        <p>
            <a href="product_info.html" style='color:
green'>${product.pname }</a>
        </p>
        <p>
            <font color="#FF0000"> 商 城 价 :
&yen;${product.shop_price }</font>
        </p>
    </div>
</c:forEach>
```

## 二、Javaweb 经典三层架构

### 1.Javaweb 经历了三个时期

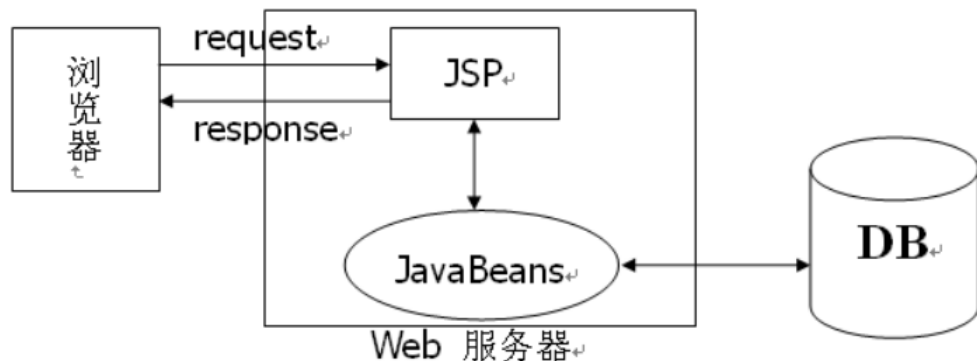
### ①JSP Model1 第一代

JSP Model1 是 JavaWeb 早期的模型，它适合小型 Web 项目，开发成本低！Model1 第一代时期，服务器端只有 JSP 页面，所有的操作都在 JSP 页面中，连访问数据库的 API 也在 JSP 页面中完成。也就是说，所有的东西都耦合在一起，对后期的维护和扩展极为不利。



### ②JSP Model1 第二代

JSP Model1 第二代有所改进，把业务逻辑的内容放到了 JavaBean 中，而 JSP 页面负责显示以及请求调度的工作。虽然第二代比第一代好了些，但还让 JSP 做了过多的工作，JSP 中把视图工作和请求调度（控制器）的工作耦合在一起了。



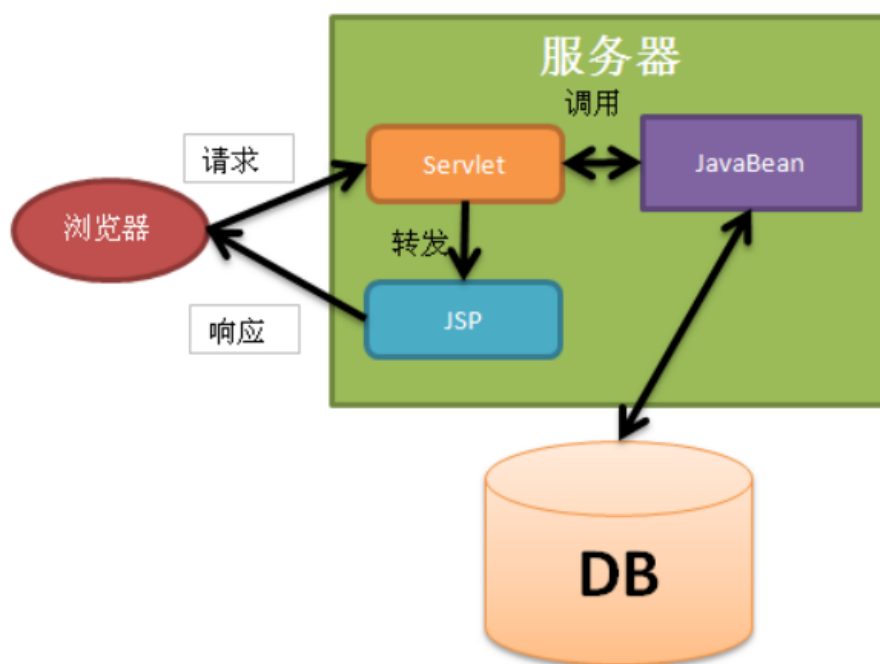
### ③JSP Model2

JSP Model2 模式已经可以清晰的看到 MVC 完整的结构了。

·JSP: 视图层，用来与用户打交道。负责接收传来的数据，以及显示数据给用户；

·Servlet: 控制层，负责找到合适的模型对象来处理业务逻辑，转发到合适的视图；

JavaBean: 模型层，完成具体的业务工作，例如：开启、转账等。



这就是 javaweb 经历的三个年代，JSP Model2 适合多人合作开发大型的 Web 项目，各司其职，互不干涉，有利于开发中的分工，有利于组件的重用。但是，Web 项目的开发难度加大，同时对开发人员的技术要求也提高了。

考核点

学员问题汇总

作业

课后  
总结分析