1. **前言**

**一 web服务体系**

Server

Http请求

Client

容器

http响应

（1）http请求流关注：

http方法（get/post/...）

请求的资源

请求传递的参数

（2）http响应流关注：

响应码

响应的内容类型

内容

1. **Web体系结构**

**一 容器**

1，servlet没有main方法，它受控于另一Java应用，即为容器（常用为Tomcat）。

当客户请求发送到web服务器时，服务器会将请求交给容器，容器再根据请求找到相应的servlet，并调用servlet的service方法处理请求。

2，容器主要提供的功能有：

1. 通信支持：实现web服务器与servlet交流
2. Servlet生命周期管理
3. 多线程管理：容器会为每一个请求创建一个新的线程，用于调用service方法处理请求。
4. 声明方式实现安全
5. Jsp支持

采用容器管理和运行servlet，最主要的 目的就是让servlet只关心处理用户请求，完成业务逻辑。

**二 URL与servlet的映射**

1，容器使用部署描述文件（DD）根据URL请求找到对应的servlet，如Tomcat中即为web.xml

2，一个servlet有3个名称：完全限定名（servlet的路径名+类名，servlet的实际地址）、部署名（DD内部用于描述）、url请求名（用户通过请求名请求资源）

在DD中需要完成内部、外部的映射关系。

|  |
| --- |
| <!-- 将servlet类的完全限定名(在服务器中的地址，包括包名和类名)映射中内部的部署名 -->  <servlet>  <servlet-name>beer</servlet-name> <!-- 部署名-->  <servlet-class>com.web.BeerSelect</servlet-class> <!-- 完全限定名-->  </servlet>    <!-- 将外部的URL映射为内部的部署名，用户可直接输入URL找到对应的servlet类 -->  <servlet-mapping>  <servlet-name>beer</servlet-name>  <url-pattern>/SelectBeer.do</url-pattern> <!-- 外部url-->  </servlet-mapping> |

容器首先根据请求的外部url，在外部映射<servlet-mappin>中找到内部部署名，再在内部映射<servlet>根据部署名找到servlet类的具体地址。

三 MVC模式

1，最开始情况时，容器将用户请求交给servlet，由servlet独自完成处理用户请求、将显示页面添加到响应等，servlet完全包揽了所有工作，当想要修改时显得十分复杂，因此产生了MVC模式：

设置一个普通的Java类来处理请求，即为模型（Model）

使用jsp显示页面并返回给用户，即为视图（View）

Servlet用于从请求中获取请求参数将其传递给Model处理，获取处理结果后传递给视图，总起控制作用，即为控制器（Controller）

Servlet

Controller

普通Java类

modle

View

Jsp 数据库

DB

1. **MVC模式**

**一 构建web应用（小）**

1，构建一个web应用，流程如下：

1. 分析用户视图：设计前端
2. 构建应用高层体系结构
3. 创建开发与部署环境
4. 各组件的迭代开发与测试

注意在迭代开发中要一步步完善体系结构。

**第四章 Servlet**

1，servlet响应过程：

1. 用户点击一个链接，指向一个servlet
2. 容器“看见”链接指向servlet，所以容器创建两个对象：

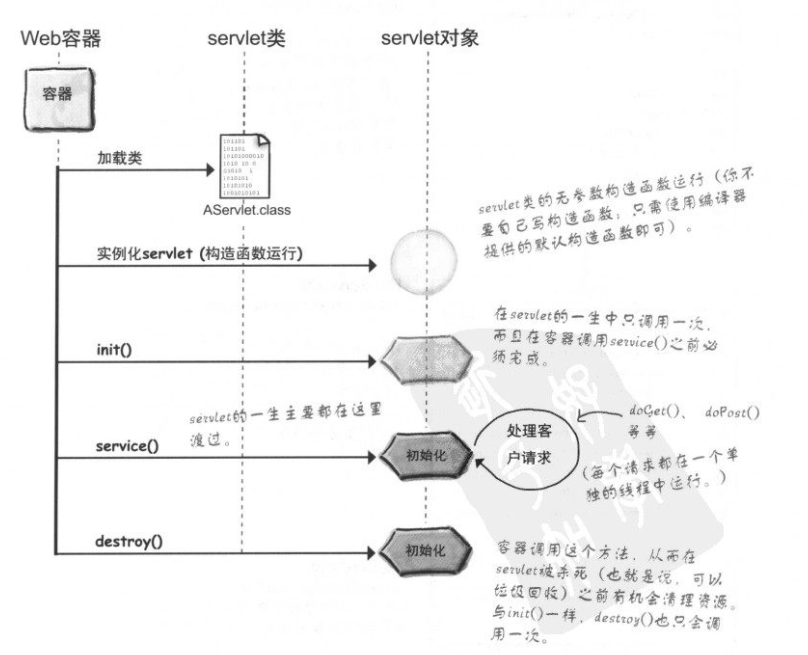
HttpServletRequest和HttpServletResponse

1. 容器根据部署描述文件（DD）映射找到对应的servlet，为这个请求创建或分配一个线程，并调用这个servlet的service()方法，将创建的请求和响应对象作为参数传递给它。
2. Service()方法根据http方法确定调用doGet或doPost处理客户请求。
3. Servlet将响应对象通过响应写回给客户。
4. Service()结束，线程撤销或返回到线程池，请求和响应对象垃圾回收。

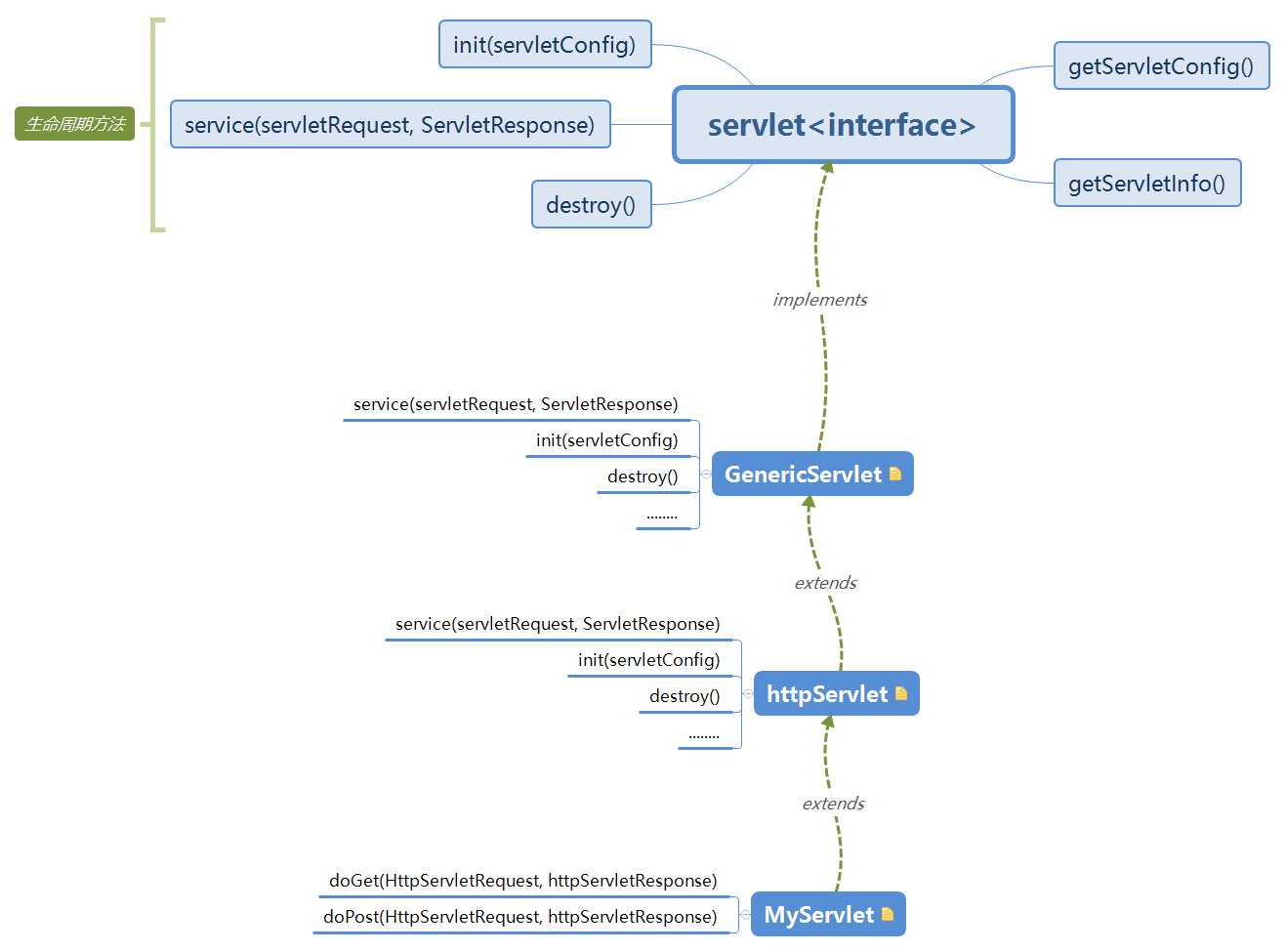
2，servlet生命周期

1. 容器启动时，会搜索servlet类并将之加载至内存。
2. Servlet类通过构造函数实例化生成servlet对象（此时还不是servlet）
3. 在调用service()服务之前，Init()初始化生成servlet
4. 容器调用servlet的service()方法为客户服务
5. 全部服务结束，容器调用destroy()方法清理servlet

Tips：Init和destroy方法在servlet一生只调用一次。



3，servlet体系



4，servlet调用：

1. 在servlet实例创建之后，为用户提供服务之前，容器调用Init()方法初始化servlet
2. 用户到来之后，容器会为每一个请求创建一个线程，并创建新的请求和响应对象，并调用service()
3. Service()方法根据http方法确定调用doGet或doPost

也就是说，容器通过运行多个线程负责处理对一个servlet的多次请求服务，每一个线程只是负责运行service()方法（doGet或doPost方法）。

对一个servlet的多次请求，并不是创建多个servlet实例，而是创建多个线程。

5，在调用Init之前，servlet实例只是一个普通的对象；调用之后，servlet才会得到所有特权，如从ServletConfig何ServletContext得到容器信息。

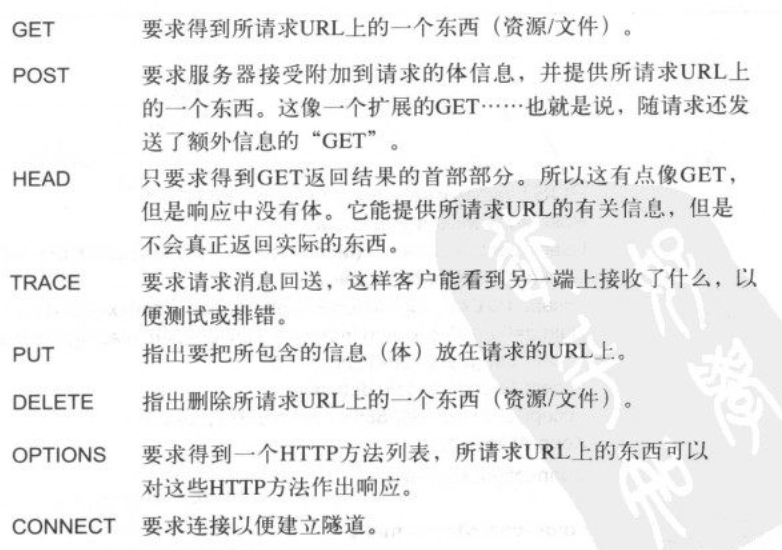
ServletConfig：每一个Servlet都有各自的ServletConfig，用于向Servlet传递部署时信息，也可以用于获取ServletContext，其参数在DD描述

ServletContext：每一个Web应用都有自己的ServletContext，用于存储web应用参数信息，也在DD部署，可获取服务器信息等

6，Servlet的真正任务是处理请求。

**二，请求和响应**

1，http方法：



2，Get和Post的区别

Get和Post都能发送参数；（默认选择get方法）

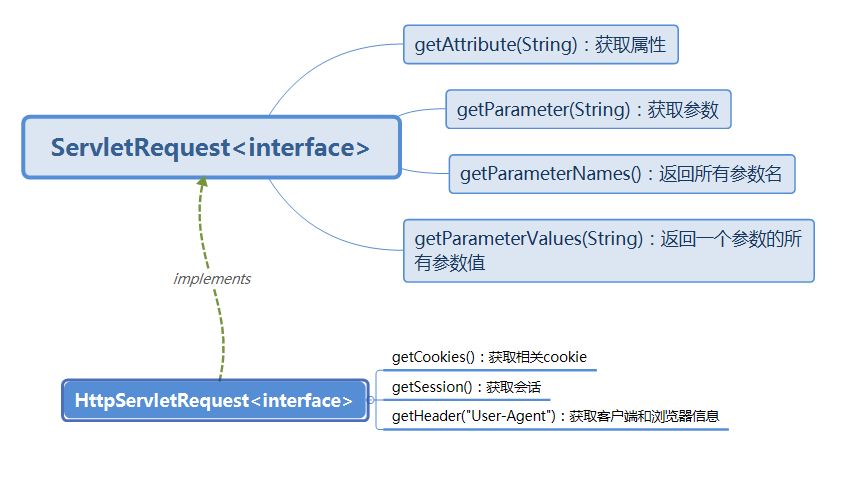
（1）get发送的参数数据有限制，且只能加在url后面（url?...&）,可直接在URL看见

（2）Post发送的参数无限制，且放在发送的消息体中

3，幂等请求、非幂等

1. post提交的数据可能用于不可逆转的事务，若客户端重复某种请求，可能产生不好的影响，因此是非幂等的。
2. Get是幂等的，重复某件事情不会产生预想不到的副作用。

4，请求对象Request



5，响应对象Response

1. 响应是为了向客户发送数据，主要用setContentType(String)设置响应内容类型和getWriter()获取输出流。
2. 文件类型



1. Servlet采用响应重定向让用户重新定位URL：

用法：response.sendRedirect(String);

String两种方式：

1，绝对URL地址

2，相对URL地址：前面加”/”，则此时重定向的URL在当前地址下重新定位

1. 请求分派：request.getRequestDispatcher(“url”).forward(req,resp); 将当前请求重新分配到另一个Servlet或jsp，直接在服务器中完成。

重定向：response.sendRedirect(String); 将重定向的地址返回给用户，让用户重新请求。

1. **web应用整体**

**一，servlet初始化参数**

1，当想要配置servlet参数信息时，而不是硬部署到servlet代码中去，可以在DD（web.xml）部署servlet初始化参数。

|  |
| --- |
| <servlet>  .....  <init-param>  <param-name>mainEmail</param-name>  <param-value>zzxy@126.com</param-value>  </init-param>  </servlet> |

然后，在servlet中通过servletConfig获取初始化参数获取。

|  |
| --- |
| getServletConfig().getInitParameter(“mainEmail”); |

2，servlet初始化之前（init）不能获取初始化参数。

容器只在初始化servlet之前读一次初始化参数，读完后就不能改变了；改变参数只能重新部署容器。

3，过程

1. 容器在初始化之前读取DD，并为这个servlet创建一个ServletConfig实例。
2. 容器为每一个初始化参数提供String键值对，并向ServletConfig提供指向键值对参数的引用。
3. 容器在init()初始化Servlet时，传入servletConfig的引用（参照前面init方法声明）

4，servlet初始化参数只有该servlet才能调用，想要其他jsp等调用，可以将参数传入request属性，再通过获取属性获取。

|  |
| --- |
| req.setAttribute(n, v);  String s = req.getAttribute(n); |

**二，ServletContext初始化参数**

1，想要整个web应用都能获取某项初始化参数，可以部署ServletContext初始化参数，再通过ServletContext对象获取参数。

设置参数：

|  |
| --- |
| <web-app>  ....  <context-param>  <param-name>mainEmail</param-name>  <param-value>xzssd@zju.com</param-value>  </context-param>  </web-app> |

获取：

|  |
| --- |
| getServletContext().getInitParameter(“mainEmail”); |

2，Servlet初始化参数：只有部署了该<init-param>的Servlet才能获取

上下文初始化参数：整个web应用的Servlet和jsp都能应用

3，整个web应用只有一个ServletContext，应用下的各个Servlet都有各自的ServletConfig

4，web应用初始化：

1. 容器读取DD，获取<context-param>中的参数并为其创建string键值对。
2. 容器创建ServletContext对象，并为其提供到各键值对的引用。
3. Web 应用上的Servlet和jsp即可通过ServletContext访问初始化参数。

5，初始化参数是部署时参数，直接在DD中部署值，可在运行中获取，而不能设置。

6，获取ServletContext对象：

1. [this.]getServletContext();
2. getServletConfig().getServletContext(); 通过ServletConfig获取（当Servlet未继承HttpServlet 或GenericServlet，由于getServletContext()是从GenericServlet继承的）

7，当想要保存的是一个对象而不是string类时，可以用ServletContext属性进行保存获取

setAttribute(string,object)：设置属性

object getAttribute(string)：获取属性

removeAttribute(string)：删除属性

Tips：

1. 当重复设置同一个属性时，后设置的属性值将替换前面的属性值，并返回前面的属性值。
2. 删除属性时，返回被删除的属性值。

**三，监听器**

1，当想要初始化之前执行某些操作，由于Servlet没有main方法，可以采用监听器监听某些动作，当监听动作发生时，会执行相应的触发动作。如上下文监听器ServletContextListener接口。

2，ServletContextListener有两个触发动作

|  |  |
| --- | --- |
| **contextDestroyed**([ServletContextEvent](file:///C:/Users/%E8%83%A1%E4%BA%AE/Desktop/servlet%20api/javax.servlet.ServletContextListener.htm" \l "top#top) sce) | 上下文销毁时触发 |
| **contextInitialized**([ServletContextEvent](file:///C:/Users/%E8%83%A1%E4%BA%AE/Desktop/servlet%20api/javax.servlet.ServletContextListener.htm" \l "top#top) sce) | 上下文(web)初始化时触发 |

3，监听流程：

1. 首先在DD注册监听器，格式如下：

|  |
| --- |
| <listener>  <listener-class>com.web.MyServletContextListener</listener-class>  <listener> |

1. 容器部署时会读取DD文件，根据描述找到实现了ServletContextListener接口的com.web.MyServletContextListener类，并为其实例化。
2. 当上下文初始化时，调用监听者MyServletContextListener的contextInitialized方法，其中ServletContextEvent有一个ServletContext的引用，可以通过getServletContext()获取上下文对象获取参数进行操作。

4，HttpSessionBindingListener：实现了该监听器接口的类，当该监听器被添加到会话属性中去（setAttribute）或者从会话中删除时，会触发相应的动作。

|  |  |
| --- | --- |
| ValueBound(HttpSessionBindingEvent e) | 处于会话中触发 |
| ValueUnbound(HttpSessionBindingEvent e) | 从会话中移除触发 |

5，每实现一个监听器类，就需要在DD进行部署，注意会话绑定(HttpSessionBindingListener)不需要在DD配置。

**四，属性**

1. 属性是绑定在3个API对象ServletContext、HttpSession、HttpServletRequest中的键值对（名：String，值：Object对象）。

（2）属性与参数

|  |  |  |
| --- | --- | --- |
|  | 属性 | 参数 |
| 类型 | 上下文、会话、请求对象 | 上下文初始化参数  Servlet初始化参数  请求参数 |
| 设置方法 | setAttribute(String,Object) | 部署时参数，都在DD中部署，不能设置 |
| 获取方法 | Object getAttribute(String) | getInitParameter(String) |
| 返回类型 | Object | String |

注意：

1，请求参数是用户向服务端发送请求时传递的参数，可以调整查询串，用getParameter(S)获取。

2，获取属性后注意要进行类型转换

1. 属性作用域

|  |  |  |  |
| --- | --- | --- | --- |
|  | 可访问性 | 生命周期 | 适用于 |
| 上下文 | 整个web应用的所有对象 | ServletContext的生命周期，即整个web应用的生命周期 | 整个应用共享的数据，如数据库连接 |
| 会话 | 能访问该特定会话的所有Servlet或jsp | 会话的生命周期 | 与客户会话有关的数据与资源，如购物车 |
| 请求对象 | 能访问该请求对象的所有部分，一般即为接收转发请求的所有Servlet和jsp，加上监听器 | 请求的生命周期，即service()方法结束。 | 将model状态从controller转移到view中或传递客户请求的数据。 |

1. 操作方法

3个属性作用域操作属性的方法是一致的。

|  |
| --- |
| Object getAttribute(String)：获取属性对象值，注意转换类型 |
| void setAttribute(String, Object)：设置属性 |
| viod removeAttribute(String)：删除属性 |
| Enumeration getAttributeNames()：获取属性名称的Enumeration |

**五，线程安全**

（1）ServletContext上下文属性

1，上下文属性是非线程安全的，由于整个web应用的Servlet都能方法上下文属性，可能产生多个线程访问该属性

2，同步方法

1. 直接同步服务方法doGet或doPost，无效，同步服务方法会导致一个Servlet只能创建一个线程，且也无法阻止其他Servlet的线程来访问上下文属性。
2. 对上下文对象加锁，对所有访问上下文属性的语句加上下文对象的锁，使之每一个线程同步访问上下文属性，格式：

|  |
| --- |
| **synchronized**(getServletContext()){  //对上下文对象加锁，封装所有操作上下文属性的方法  ServletContext sc = getServletContext();    sc.setAttribute("ball", "22");  String mon = (String)sc.getAttribute("ball");  out.println("money="+mon);  } |

1. 会话属性

1，HttpSession同样不是线程安全的，一个会话中的多个Servlet也可以访问会话属性

2，同步方法：对会话对象加锁，封装所有操作会话属性的语句

|  |
| --- |
| HttpSession session = request.getSession();  **synchronized**(session){  //对会话对象加锁  session.setAttribute("bot", "442");  out.println("money="+(String)session.getAttribute("bot"));  } |

1. 只有请求属性和局部变量（servlet）是线程安全的。

1，由于每一线程负责运行客户的每一个请求，因此请求属性、局部变量都对应一个线程，因此是安全的。

2，Servlet中实例变量不是线程安全的，若存在一个Servlet的多个线程，则实例变量可被每个线程访问。因此，若想要线程安全的状态，Servlet中一般不设置实例变量，若要设置，则应声明为final属性。

**六，请求分派与请求属性**

1. 请求分派

1，从request对象中获取：

|  |
| --- |
| RequestDispatcher view = req.getRequestDispatcher("url"); |

其中，若url不加”/”，则表示为相对路径，相对于当前路径下

若加“/”,表示从web应用的根目录下开始

2，从ServletContext获取：

|  |
| --- |
| RequestDispatcher view=getServletContext().getRequestDispatcher("/result.jsp"); |

其中只能加”/”，从根目录下开始。

1. 转发属性

|  |
| --- |
| view.forward(request, response); |

1. 若Servlet已经response提交响应(out.println())，再请求转发或重定向会发生illegalStateException。
2. **会话管理**

1，对应同一个客户，HttpSession对象可以跨多个请求(Servlet线程)保持会话模式。

即，与一个用户的会话期间，HttpSession对象可以持久化存储，对于会话期间，客户所做的所有请求信息都可以在Session中保存获取。

2，对于容器来说，每一个请求都默认为来自一个新的客户，创建一个新的会话。

3，想要容器区分每次请求是属于哪个客户的会话，因此需要为每一个用户设置一个唯一的会话ID，流程：

1. 针对客户的第一次请求，容器会为客户生成一个唯一的会话ID，并通过响应返回给客户。
2. 客户在以后的每一次请求中都发送自己的会话ID,容器会通过该ID找到客户对应的会话。

4，客户和容器是通过cookie来交换会话ID的 ，且容器会默认地完成所有cookie的工作。

|  |
| --- |
| 客户第一次请求，容器创建会话返回会话ID：  HttpSession session = requset.getSession();  //只需要向请求对象申请一个会话，其他的创建会话ID、创建cookie、将会话ID放在cookie中、将cookie作为响应的一部分返回给客户等都由容器自动完成。 |

|  |
| --- |
| 从客户请求根据返回的会话ID找到用户对应的会话:  HttpSession session = requset.getSession();  //与创建会话ID方法一样，会话id不可见，容器自动完成。 |

综合：

|  |
| --- |
| HttpSession session = request.getSession();  **if** (session.isNew()){  //会话为新建的会话，即直接返回给用户    }**else**{  //为用户返回的会话ID，找到与该ID匹配的会话  } |

5，若想要已经存在的客户会话，

|  |
| --- |
| //返回一个已有会话，若无与用户匹配的会话，返回null  HttpSession session = request.getSession(**false**);  **if**(session == **null**){  //创建新会话  session = request.getSession();  }**else**{  //操作  } |

|  |
| --- |
| getSession(Boolean);  参数为true时，当请求没有匹配的会话，则会创建一个新会话，无参数同理。  若为false，当没匹配会话时，返回null |

6，当客户不支持cookie时，可以使用URL重写在客户和容器之间传递会话ID

1. 在写至响应的HTML中，URL重写会把会话ID添加到其中所有URL的后面，格式：

URL+;jsessionid=XXX

1. URL重写中，会话id会作为请求URL的最后“额外”信息返回给容器。
2. 若客户不接受cookie，容器会自动地完成URL重写，但必须显示地对所有重写的URL进行编码，其他工作（添加会话id到URL）容器会自动地完成。
3. 对URL进行编码，需要调用response.encodeURL(String);

|  |
| --- |
| out.println("<a href=\""+response.encodeURL("/BeerSelect.do")+"\">Click here</a>"); |

1. 若想在重定向中使用会话，可以用response.encodeRedirectURL("/BeerSelect.do");编码重定位URL。
2. 容器在使用会话时，首先会尝试使用cookie进行会话ID交换，若不支持cookie，再回采用URL重写，此时需要编码所有重写的URL。
3. 不能对静态页面完成URL重写，使用会话必须使用动态页面。

7，删除会话

1. 会话超时
2. 在DD中配置会话超时时间

|  |
| --- |
| <!--设置会话超时--> <session-config>  <session-timeout>15</session-timeout> <!--分钟为单位--> </session-config> |

1. 会话对象设置时间

|  |
| --- |
| session.setMaxInactiveInterval(15\*60); //秒数为单位 |
|  |

会话时间设为-1后，会话永远不会到期

1. 会话对象调用invalidate()
2. 应用结束（崩溃或取消部署）

8，Cookie

1. Cookie不仅可用于在客户端和服务器间传递会话ID,还可用于在客户和服务器中传递信息，每一个Cookie都是一个String键值对对象，Cookie对象声明：new Cookie(String, String).
2. 当客户想要长时间保持信息时（如用户名），客户在第一次请求时，Servlet从请求参数中获取信息，创建Cookie并发送给用户，用户保存在客户端，当用户再次发送请求时，用户会在每一个请求中将Cookie发回给服务端（自动），服务器从Cookie得到信息分清客户。
3. 当客户会话消失时，会话Cookie就会消失，但可以设置Cookie时间使之在客户端持久存储。

QuickExample1：服务端设置Cookie

|  |
| --- |
| String name = request.getParameter("name");  //创建Cookie  Cookie cookie = **new** Cookie("username", name);  //设置Cookie在客户端保存时间  cookie.setMaxAge(30\*60);  //将Cookie发给客户  response.addCookie(cookie); |

QuickExample2：服务端获取Cookie

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

由于Cookie没有getValue(String)的方法，只能首先获取Cookie数组，再在数组中寻找。

9，会话生命周期事件

|  |  |  |
| --- | --- | --- |
| 事件 | | 监听器与监听事件 |
| 生命周期 | 创建会话：容器第一次创建会话，即此时用户未使用该会话访问服务器。  删除会话：容器置会话无效。 | HttpSessionListener  HttpSessionEvent |
| 属性 | 增加属性：setAttribute  删除属性：removeAttribute  替换属性：重新调用setAttribute | HttpSessionAttributeListener  HttpSessionBindingEvent |
| 迁移 | 会话钝化：容器在将会话迁移到另一个VM前  会话激活：容器已经将会话迁移到另一个VM | HttpSessionActivationListener  HttpSessionEvent |

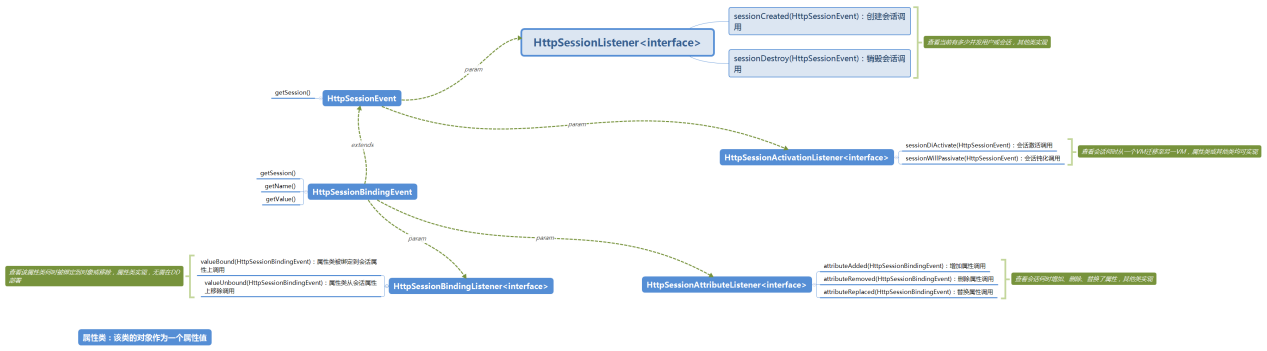
10，会话迁移

1. 分布式web应用：应用的各部分复制到网络的多个结点。一个用户的请求可能发送到多个JVM环境中，则web应用中ServletConfig、HttpSession、ServletContext其分布如下：
2. 只有HttpSession会话对象会从一个VM 迁移到另一个VM，其它均在每一个VM中进行复制
3. 每一个VM中都有一个web应用的ServletContext，每个VM中的每一个Servlet都有自己的ServletConfig；但对于web应用一个特定的会话ID，整个分布式系统上只有一个HttpSession对象。
4. 迁移过程：

1）当客户的第一次请求经过负载平衡服务器发送到VM1后，第二个请求发送到VM2，VM2根据会话ID发送当前会话对象在VM1，则将会话对象从VM1迁移到VM2（VM1会话钝化，VM2会话激活）

2）会话从一个VM迁移到另一个VM时，实现了java.io.Serializable的属性值（串行化）会转移到新的JVM中

11，会话监听器



1. **Jsp**

1，服务器端的jsp页面最终会转换、编译成Servlet为用户提供服务。

2，语法

1. jsp脚本：<% Java语句 %>
2. Jsp指令：在页面转换时向容器发出特殊的提示。<%@ page|include|taglib ... %>

其中，<%@ page import=”java.util.\*, java.io.\*, .... ”>可导入包

1. 表达式： <%= 表达式 %>; 表达式成为out.println()中的参数。
2. Jsp声明：<%! Java语句 %>

3, jsp与Servlet映射：

|  |  |
| --- | --- |
| XXX.jsp | XXX\_jsp.java |
| jsp脚本<%%> | \_jspService()语句 |
| Jsp指令 | 指令生成 |
| 表达式 | \_jspService()中out.println() |
| jsp声明<%! %> | servlet中实例成员 |

4,jsp隐式对象

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

1. 只有指定的“错误页面”才能用exception对象
2. pageContext封装了其他隐式对象，可以根据pageContext获取其他隐式对象的引用。

5，jsp注释

1. html注释：<!--- html -->，容器直接传送会客户，浏览器将其注释
2. jsp注释： <%-- jsp -->，容器将其隐藏起来，不返回给客户

6，容器根据jsp生成一个类，该类实现了java.servlet.jsp.HttpJspPage接口，主要有3个方法组成。

|  |  |
| --- | --- |
| jspInit() | Init()方法调用，可覆盖（jsp声明中覆盖） |
| jspDestroy() | Destroy()方法调用，可覆盖 |
| \_jspService() | Service()方法调用，对于每一个请求由一个单独的线程负责运行，不可覆盖 |

7，jsp生命周期

1. XXX.jsp文件部署为web应用的一部分，应用启动，容器读取DD，在请求该jsp页面之前并不做任何处理。
2. 用户请求XXX.jsp页面，容器先尝试将XXX.jsp页面转换为XXX\_jsp.java，再将该源文件编译为.class类文件。
3. 容器加载类文件，并对其调用init()初始化，init()调用jspInit()方法生成一个正式的Servlet。
4. 针对用户请求，容器创建一个线程负责调用Servlet的\_jspService()方法处理请求。
5. 后面即为Servlet处理请求的部分。

8，jsp整个生命周期内，只转换和编译一次，编译生成.class类文件后，一旦经过加载和初始化后，每次请求只会发生一件事，即创建一个新线程负责运行\_jspService()服务方法。

9，初始化jsp

1. 由于jsp最终转换为Servlet，想要jsp获取Servlet初始化参数：
2. 在DD中配置Servlet初始化参数：在<servlet>标记中将<servlet-class>换为<jsp-file>:

|  |
| --- |
| <servlet>  <servlet-name>JspTestInit</servlet-name>  <jsp-file>/test.jsp</jsp-file>    <init-param>  <param-name>email</param-name>  <param-value>zzhh@zju.com</param-value>  </init-param>  </servlet>    <servlet-mapping>  <servlet-name>JspTestInit</servlet-name>  <url-pattern>/jsp.do</url-pattern>  </servlet-mapping> |

<jsp-file>实际上是将对Servlet的映射指向了jsp页面，并将初始化参数一并传过去。

1. 覆盖jspInit()

容器在将jsp编译的类文件初始化为Servlet时，会通过init()调用jspInit()方法，因此在运行jspInit()时已有ServletConfig和ServletContext可调用，可在jsp声明中覆盖jspInit()方法。

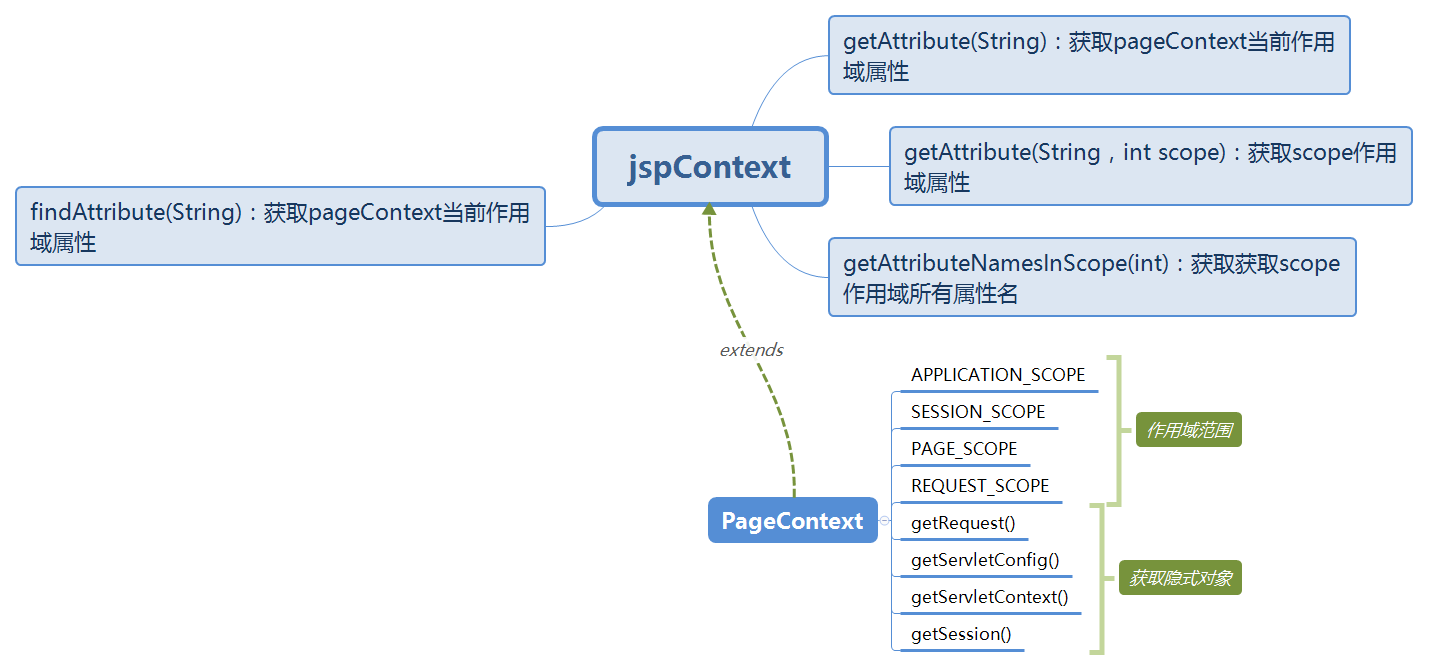
|  |
| --- |
| <%!  **public** **void** jspInit(){    ServletConfig jsc = getServletConfig();  //获取ServletConfig参数  String email = jsc.getInitParameter("email");  //添加到ServletContext属性  getServletContext().setAttribute("email", email);  }  %> |

注意：jsp隐式对象的作用范围仅限于\_jspService()服务方法，因此在jsp声明中不能使用隐式对象。

10，jsp属性：jsp中可通过隐式对象直接作用4个作用域的属性。

|  |  |  |
| --- | --- | --- |
| 作用域 | Servlet | Jsp(使用隐式对象) |
| 应用 | getServletContext().setAttribute() | application.setAttribute() |
| 会话 | getSession().setAttribute() | session.setAttribute() |
| 请求对象 | request.setAttribute() | request.setAttribute() |
| 页面 | 不适用 | pageContext.setAttribute() |

11，可以使用pageContext获取任意作用域的属性



如：

|  |
| --- |
| <!-- 设置属性，未设置为页面作用域 -->  <% pageContext.setAttribute("n", "tt"); %>  <!-- 设置会话作用域属性 -->  <!--相当于session.setAttribute(“n”, “tt”);-->  <% pageContext.setAttribute("n", "tt", pageContext.SESSION\_SCOPE); %>  <!-- 获取会话属性 -->  <!--相当于session.getAttribute(“n”);-->  <%String s = (String)pageContext.setAttribute("n", pageContext.SESSION\_SCOPE) %> |

1. pageContext.findAttribute(“n”);findAttribut查找属性首先在页面作用域查找属性，若未找到，再在其他作用域查找，从最严格的作用域开始，逐步到不那么严格的作用域。(请求-会话-上下文作用域)

12，jsp指令

1. page指令：<%@ page ... %>设置页面特有的属性，如字符编码、内容类型、包的导入等。
2. taglib指令：<%@ taglib tagdir=””... %>：定义jsp可用的标记库
3. include指令：<%@ include file=”” ... % >：定义在转换时添加到当前页面的文本和代码。

13，jsp插入Java比较麻烦，可以使用el(expression language)表达式替代Java语言。

（1）禁用Java：<script-invalid>true</script-invalid> 默认可使用

1. 禁用el：默认启动，page指令优先级高于DD设置

1）DD中放置<el-ignored>true</el-ignored>

1. page指令设置：<%@page isELIgnored=”true”>

|  |
| --- |
| <jsp-config>  <jsp-property-gruop>  <url-pattern>\*.jsp</url-pattern>  <script-invalid>true</script-invalid>  <el-ignored>true</el-ignored>  </jsp-property-gruop>  </jsp-config> |

14，jsp动作：

1. 标准动作：

<jsp:include page=”\*.jsp”>

1. 非标准动作：

<c:set var=”r” value=”32”>

1. **无脚本jsp**

**一，jsp动作**

1，当获取属性中的对象值时，Java脚本比较复杂，可以使用jsp动作获取。

如request设置了Person的JavaBean对象，在jsp页面获取Person对象的Name值：

|  |
| --- |
| <jsp:useBean id=*"person"* class=*"com.web.Person"* scope=*"request"*/>  Name: <jsp:getProperty name=*"person"* property=*"name"* /> |

2，动作

（1）<jsp:useBean>声明和初始化一个Bean属性；其中

id : Bean对象在属性中的标识符（名称），即serAttribute中所用的名称。

class : Bean对象类的类型，完全限定名

scope : Bean对象的作用域，即属性的作用域，默认为page

注意：当useBean在属性作用域中找不到名为”person”的属性对象时，会自动创建一个类型为class的对象。

1. <jsp:getProperty>得到Bean对象的成员变量值，其中

name : 标识具体的Bean对象，与useBean中的id一致

Property: 想要获取的Bean对象中的成员变量名。

1. <jsp:setProperty>:设置Bean对象的成员值，主要用于当useBean发现对象不存在时，创建对象并设置对象的成员值。存放于useBean的体中，若有对象，则不设置。

|  |
| --- |
| <jsp:useBean id=*"person"* class=*"com.web.Person"* scope=*"request"*>  <jsp:setProperty name=*"person"* property=*"name"* value=*"defalut"* />  </jsp:useBean> |

3，Bean法则：当一个对象要作为Bean时，一般要遵守

1. 必须有一个无参数的构造函数
2. 必须按状态名约定获取和设置状态的方法。
3. 结合jsp使用时，状态类型必须是String。

4，useBean多态

1. useBean动作可以添加一个type属性用于表示引用类型，如Person是一个抽象类，Employee继承了Person类。

|  |
| --- |
| <jsp:useBean id=*"person"* type=*"com.web.Person"* class=*"com.web.Employee"* scope=*"request"* /> |

生成的Servlet表示为：Person person = new Employee();

1. type一般是class类及其父类，class不能是抽象类，且存储无参构造
2. 若只存在type，不存在class，则作用域中必须存在名为person的属性键值对，否则会抛出异常。
3. Type-引用类型，class-对象类型，格式：type t = new class();

5,请求直接发送到jsp，可以直接在jsp页面先用useBean标记创建一个属性类，并setProperty设置类中状态，再利用getProperty标记获取状态。

|  |
| --- |
| <jsp:useBean id=*"person"* class=*"com.web.Person"* scope=*"request"*>  <%-- <jsp:setProperty name="person" property="name" value='<%=request.getParameter("name") %>' /> --%>  <%-- <jsp:setProperty name="person" property="name" param="name" /> --%>  <jsp:setProperty name=*"person"* property=*"\*"* />  </jsp:useBean> |

获取请求参数：

1. Java脚本：request.getParameter
2. setProperty的param属性：param属性值设为请求参数键值对的名称，可直接获取参数值；且当属性类的状态名称和请求参数的名称一致时，即Person类的”name”和request的getParameter(“name”)同名时，可省略param，容器为其完成。
3. setProperty会自动地完成基本类型性质转换

6，当属性对象不止基本类型、String时，即对象中还存在对象时，此时getProperty只能打印属性对象的性质，不能打印属性对象的性质的性质。

Tips：jsp标记注意要加结束标记

**二，EL表达式**

1，el表达式格式：${expression}；一般expression为first.sec,第一个变量即可以是一个隐式对象，也可以为属性名。相当于jsp表达式<%= ... %>

（1）el隐式对象：

|  |  |
| --- | --- |
| PageScope | 作用域属性的map |
| requestScope |
| SessionScope |
| applicationScope |
| param | 请求参数的map |
| paramValues |
| header | 请求首部的map |
| HeaderValues |
| cookie | Cookie的map |
| initParam | 上下文初始化参数 |
| pageContext | PageContext隐式对象，可获取其他jsp隐式对象 |

El隐式对象中只有pageContext不是map，其他对象都是映射。

2，“.”操作符

（1）采用“.”操作符左边要么是一个映射(Map),要么是一个Bean，右边必须符合Java命名规则（由a-zA-Z0-9,\_,$符合组成，但首字母不能是数字）。如：${person.name}

3,“[]”操作符

1. 采用“[]”左边可以是Map、Bean、List或是数组。
2. 当左边是一个List或数组时，如果索引是一个String直接量，则该索引会被强制转换为int型
3. Servlet配置属性：

|  |
| --- |
| String[] favoriteMusic = {"Maroon 5", "BT", "JB"};  request.setAttribute("musicList", favoriteMusic); |

1. Jsp用el:

|  |
| --- |
| Music 0 : ${musicList[0]}  Music 1 : ${musicList["1"]} <!-- “1”会被转为1 --> |

1. 当左边是Map或Bean时，“.”和“[]”都可以使用

1）Servlet配置属性：

|  |
| --- |
| Map<String, String> musicMap = **new** HashMap<String, String>();  musicMap.put("Maroon 5", "Sugar");  musicMap.put("BT", "DJ");  musicMap.put("JB", "sin");    request.setAttribute("musicMap", musicMap);  request.setAttribute("ecc", "Maroon 5"); |

2）js获取

|  |
| --- |
| MusicMap : ${musicMap.JB }<br/>  MusicMap : ${musicMap["Maroon 5"] }<br/>  MusicMap : ${musicMap[ecc] }<br/> |

注意：

当[]中不是String直接量时，容器会根据中括号中的值在属性中查找对应的值，再替换进去，否则返回null。 （ecc在属性中映射为”Maroon 5”）

1. []里面可以进行嵌套，但不能叠加，如music[][]

|  |
| --- |
| MusicMap : ${musicMap[musicList[1]] }<br/>  <!--相当于 ${musicMap[“BT”]}--> |

4，el隐式对象

1. param：请求参数只有一个值，用param

paramValues：请求参数有多个值

|  |
| --- |
| name : ${param.name }<br/>  id : ${param.id }<br/>  food : ${param.food }<br/>  paramValues food : ${paramValues.food[0] }<br/>  paramValues food : ${paramValues.food[1] }<br/> |

1. header获取请求头部

|  |
| --- |
| host: ${header.host }<br/> <!--返回主机名--> |

1. PageContext可以获取jsp隐式对象

|  |
| --- |
| Method: ${pageContext.request.method }<br/> |

通过pageContext获取request隐式对象，再获取http方法(GET)

1. 作用域隐式对象：将不符合Java命名规则的属性名(左边)转换为符合规则

|  |
| --- |
| person : ${requestScope["web.foo"].name }<br/> |

1. cookie隐式对象：cookie直接获取所选择的cookie的值

1）设置cookie

|  |
| --- |
| Cookie cookie = **new** Cookie("username", "tan");  cookie.setMaxAge(30\*60);  response.addCookie(cookie); |

2）直接获取username的值，注意中间与cookie名一致

|  |
| --- |
| cookie : ${cookie.username.value }<br/> |

1. 初始化参数initParam：获取上下文参数值

1）DD配置上下文初始化参数

|  |
| --- |
| <context-param>  <param-name>email</param-name>  <param-value>zzzh@zju.edu.com</param-value>  </context-param> |

1. 获取initParam右边接参数名称

|  |
| --- |
| email: ${initParam.email } |

5，el能很好地处理null，当它找不到指定的属性/性质/键时，就在页面上什么都不显示。

而在算术表达式中，把未知值当做0，逻辑表达式中，把未知值当做false。

6，el表达式存储变量；${var}，如${username}会依次从page、request、Session、application作用域中寻找名称为username的属性值。

7，taglib

1. el表达式可以利用taglib使用普通Java类的公共静态方法：
   1. 创建普通Java类

|  |
| --- |
| **package** com.web;  **public** **class** DiceRoller {  **public** **static** **int** rollDice(){  **return** (**int**) ((Math.*random*()\*6) + 1);  }  } |

* 1. 创建tld文件，将函数名映射到Java类的静态方法

|  |
| --- |
| <uri>DiceFunc</uri>  <function>  <name>rollIt</name>  <function-class>com.web.DiceRoller</function-class>  <function-signature>  int rollDice()  </function-signature>  </function> |

<name>标志函数名，<function-class>标志Java类的完全限定名，

<function-signature>标志静态方法，<uri>告诉容器tld的名字。

* 1. 在jsp文件调用tld文件的函数名来使用静态文件

|  |
| --- |
| <%@ taglib prefix=*"mine"* uri=*"DiceFunc"* %>  Num:${mine:rollIt()}<br/> |

**三，可重用模板**

1，include指令和jsp动作

1. 指令：<%@ include file="header.jsp" %>

include指令在被插入的jsp页面转换时，插入jsp页面的源代码，一同编译为Servlet。---转换时插入

1. Jsp动作：<jsp:include page=*"header.jsp"* />

<jsp:include>动作是在被插入的jsp页面编译成Servlet后，当调用Servlet时，该Servlet调用由header.jsp产生的Servlet（header.jsp在某一时间经过转换编译初始化生成Servlet） ---运行时调用

1. 指令对位置敏感，所包含内容会插入到指令所在位置，且其属性是file。

而动作属性是page

1. 可以使用<jsp:param>定制插入的内容
   * 1. Jsp页面设置参数

|  |
| --- |
| <jsp:include page=*"header.jsp"* >  <jsp:param value=*"rollIt"* name=*"subTitle"*/>  </jsp:include> |

J. Header中获取

|  |
| --- |
| <em><strong> ${param.subTitle }</strong></em> |

4，<jsp:forward page=/>将当前jsp转发到其他的jsp或Servlet，并在跳转之前清空当前缓冲区（即清除当前页面，转到响应页面）

5，include指令原用于静态页面的布局模板，采用include指令，所包含的页面源代码将成为含有include指令的“外围”页面的一部分。

1. **JSTL(jsp standard tag lib)jsp标准标记库**

**一，jstl标记**

1，使用jstl标记前，需要声明标记来源，如使用c前缀标记，必须声明：

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

2，<c:forEach>标记实现循环，处理数组和集合。

|  |
| --- |
| String[] movieList1 = {"matrix", "ET", "residents"};  request.setAttribute("movieList", movieList);  <!-- jsp页面 -->  <c:forEach var=*"movie"* items=*"*${movieList }*"*>  <td>${movie}</td>&nbsp;&nbsp;&nbsp;  </c:forEach> |

1. 标记中var设置一个参数名，items选择要处理的数组或集合名称，利用var获取实际值。
2. 还可设置一个varStatus=*"movieLoopCount"属性，用*${movieLoopCount.count}获取循环次数。

3，<c:if test=””>完成条件查询，test是判断条件，格式：

|  |
| --- |
| <c:if test=”条件”>  判断为真执行  </c:if> |

4,<c:choose>完成选择，格式：

|  |
| --- |
| <c:choose>  <c:when test=”条件”>  ...  </c:when>  <c:otherwise>  ...  </c:otherwise>  </c:choose> |

5, <c:set>

1. <c:set>类似<jsp:setProperty>设置属性值，但<c:set>不仅能设置Bean中性质，还能设置属性值、Map值等。
2. <c:set>有两种属性设置：var和target，var用于设置属性变量，target用于设置Bean性质或Map值，且两种都存在有体和没体两种形式。体用于放置值。
3. <c:set var ...>设置属性变量，value不一定为String，可以为对象，注意作用域
   1. 没体

|  |
| --- |
| <c:set var=*"userStatus"* scope=*"request"* value=*"Brilliant"*/>  userStatus: ${userStatus}<br/> |

* 1. 有体

|  |
| --- |
| <c:set var=*"us"* scope=*"request"*>  Brilliant,Good,Bad  </c:set>  us: ${us}<br/> |

有体直接计算输入值；若值计算为null，则在该作用域内删除该属性变量

1. <c:set target= ...>设置Bean性质或Map值
   1. 没体

|  |
| --- |
| <c:set target=*"*${Person}*"* property=*"dname"* value=*"Dan"*/> |

* 1. 有体

|  |
| --- |
| <c:set target=*"*${Person}*"* property=*"dname"*>  ${foo.name}  </c:set> |

注意：

1，Target是目标（Bean或者Map）,不能为null;其值需要解析为一个真正的对象，因此一般是el表达式或jsp表达式。

2，若目标是Bean，则设置的是Bean中的性质值，target.property

若目标是Map，则设置的是Map中的键值对，target[property]

1. <c:remove>删除属性

|  |
| --- |
| <c:remove var=*"userStatus"* scope=*"request"*/> |

1. <c:import>将资源添加到jsp中

|  |
| --- |
| <c:import url=*"http://java.sun.com/"*>  <c:param name=*"first"* value=*"hu"* />  </c:import>    Firsname : ${first} |

1，include指令、jsp中include动作都能将资源添加进jsp，但<c:import>能导入外部web的资源，include只能导入其所在web的资源。

2，<c:import>也能类似include动作定制导入的内容，参照上面

1. <c:url>可以对地址进行重写和编码
   1. 重写，容器在cookie禁用时完成传递会话ID

|  |
| --- |
| <c:url value=*"/index.jsp" /*> |

* 1. 编码：将URL中不安全/保留的字符替换为其他字符

|  |
| --- |
| <c:url value=*"/index.jsp"* var=*"inputUrl"*>  <c:param name=*"first"* value=*"*${first }*"* />  <c:param name=*"last"* value=*"*${sec }*"* />  </c:url>  The Url: ${inputUrl }  输出：  The Url: /09JstlTest/index.jsp?first=hiden+dsf&last=hidensec++dfs |

编码查询参数，如将空格换成+

**二，错误页面**

1，page指令中设置错误页面，错误页面中标明

|  |
| --- |
| <%@ page .....errorPage=*"errorpage.jsp"*%> |
| <!-- errorpage.jsp -->  <%@ page .....isErrorPage=*"true"*%> |

2，DD中配置错误页面

1. 声明一般错误页面

|  |
| --- |
| <error-page>  <!--异常可以声明为更加明显的异常 -->  <exception-type>java.lang.Throwable</exception-type>  <location>/errorpage.jsp</location>  </error-page> |

1. 根据http状态码声明错误页面

|  |
| --- |
| <error-page>  <error-code>404</error-code>  <location>/NotFound.jsp</location>  </error-page> |

3，在错误页面可以获取异常对象 ：${paggeContext.exception}

4，<c:catch>标记完成一种try/catch，将有风险的地方包括起来，当捕获到异常时，直接跳至标记的后面。还可以设置一个var属性，用于在标记结束后访问异常。

|  |
| --- |
| <c:catch var=*"myExp"*>  <% **int** s = 10 / 0; %>  </c:catch>  You will see ${myExp.message } |

**三 定制标记库**

1，自己定制标记库（非jstl），需要使用tld描述定制标记，（tld还可用于建立el函数<function>：参考上章）。

2，定义tld库：

|  |
| --- |
| <short-name>RandomTags</short-name>  <uri>randomThings</uri> <!-- 唯一标记tag -->  <tag>  <description>random advice</description>  <name>advice</name> <!-- 标记名称 ,如<my:advice> -->  <tag-class>com.tag.TagHandler</tag-class> <!-- 标记引用的类 -->  <body-content>empty</body-content> <!-- 设置标记不能有体 -->  <!-- 若所引用的标记类有属性，则每一个属性都必须设置一个attribute -->  <attribute>  <name>username</name>  <required>true</required> <!--表示标记库中必须有username属性 -->  <rtexprvalue>true</rtexprvalue> <!--表明属性是一个运行时表达式值，不一定非是String直接量 -->  </attribute>  </tag> |

3，使用advice标记

|  |
| --- |
| <%@taglib prefix=*"mine"* uri=*"randomThings"* %>  Advice: <br>  <!-- username可以为运行时表达式，el表达式即可 -->  <mine:advice username=*"*${username}*"*/> |

4，设置标记引用的Java方法，标记类必须扩展SimpleTagSupport方法

|  |
| --- |
| //username属性  **private** String username;  //jsp利用tld中声明名称调用标记时，会调用doTag方法  **public** **void** doTag() **throws** IOException {  getJspContext().getOut().write("Hello "+username+"<br>");  getJspContext().getOut().write("Your advice is "+getAdvice());  }  **public** **void** setUsername(String name){  **this**.username = name;  }  String getAdvice(){  String[] adOptions = {"blue", "yellow", "green"};  **int** random = (**int**)(Math.*random*() \* adOptions.length);  **return** adOptions[random];  } |

5，当tld中<rtexprvalue>未设置或设为false时，再设置标记属性时，只能设置String直接量，如<mine:advice username=”hu”/>，而不能使用表达式。

1. 当允许为运行时表达式时，可以设置
   1. El表达式：<mine:advice username=”${username}”/>
   2. 脚本表达式：<mine:advice username=”<%= request.getAttribute()%>”>
   3. Jsp:attribute标准动作：

<mine:advice>

<jsp:attribute name=”username”>${username}</jsp:attribute>

</mine:advice>

6，<body-content>标记体中可放置：

empty：标记不能有体

Sciptless: 标记不能有脚本，可以为el、html模板等

Tagdependent：标记体看成纯文本，所以不会计算el

Jsp:能放jsp中的都能放进去

1. 对于没有体的标记，有3中设置方式：
2. 空标记：<mine:advice username=”${username}”/>
3. 标记间无内容： <mine:advice username=”${username}”></mine:advice>
4. 开始结束标记之间只有<jsp:attribute>，参照5.(2).c)

<jsp:attribute>并不算体内容

7，uri唯一标记tag，容器当在jsp中读到URI时，会自动查找WEB-INF下面的TLD，并根据taglib中的uri匹配tld中的uri(uri并不一定指具体标记的位置)

8，当使用多个标记库时，每个tld都要有一个单独的tld，且注意：

1. 每一个taglib uri都是唯一的，即多个指令不能有相同的uri
2. 不能使用保留前缀(prefix)：jsp、jspx、Java、javax、Servlet、sun、sunw

Tips:

1，jsp中有4大作用域：request、session、servletContext、pageContext，每个作用域都可以设置和获取属性（setAttribute、getAttribute），（servlet只有3个，没有pageContext），设置属性值方法有：

1. 利用jsp脚本设置：request可替换为session、application、pageContext

|  |
| --- |
| <%  request.setAttribute("user", p1);  Person p = (Person)request.getAttribute("user");  %> |

（2）<c:set>设置属性值

|  |
| --- |
| <c:set var=*"user"* scope=*"request"* value=*"Brilliant"*/>  userStatus: ${userStatus}<br/> |

var设置属性名，scope设置属性作用域范围，value设置属性值。

1. **定制标记**

**一 概念**

1，记住标记是为了方便web开发人员不在jsp中使用Java语言，而利用标记这一标签替代Java语言脚本等，用标记表示脚本。通常有两种形式，一是标记文件，二是标记处理类。

**二 标记文件**

1，标记文件可看成一种“轻型标记处理器”，直接利用标记文件开发定制标记，用标记代表标记文件，通常标记文件时include指令或动作插入的内容，而不再编写复杂的Java标记处理器类。

2，流程：

1. 取一个被包含文件(如header.jsp)，将其重命名为后缀为.tag的文件(header.tag)。
2. 将tag文件放在WEB-INF目录下的tags子目录(WEB-INF/tags)
3. 在被插入的jsp文件中，放置taglib指令，并调用该标记。

|  |
| --- |
| <%@ taglib prefix=*"myTags"* tagdir=*"/WEB-INF/tags"* %>  <myTags:header/> |

prefix声明前缀，tagdir标明标记文件位置，标记名称：前缀:文件名(前)

3，当想要向标记文件中发送信息(定制文件)：

1. <jsp:include>动作中使用<jsp:param>设置请求参数，在标记文件中用${param.name}获取。
2. 使用定制标记文件时，可在标记中设置属性，再在标记文件中获取。

|  |
| --- |
| //标记中设置属性  <myTags:header subTitle="roll It"/>  //标记文件(header.tag)中获取  <%@attribute name="subTitle" required="true" rtexprvalue="true" %>  <img src=*"head\_small.jpg"* /> <br>  <em><strong> ${subTitle }</strong></em><br> |

且为了表示标记文件需要一个属性，在标记文件中使用<%@attribute name=”属性名” ...>来提醒使用标记时需要设置属性。

1. <jsp:param>设置的是请求参数，可在参与该请求的所有组件中获取；

<%@ attribute%>则设置的是属性，只能在该标记文件中使用。

4，当标记包含的属性值很大时，可以将属性值放置在标记体中；

然后在标记文件中用<jsp:doBody/>获取标记体。且需要在标记文件中声明标记体类型。

<%@ tag body-contentt=”tagdependent/scriptless/empty” %>

1. 标记中设置标记体

|  |
| --- |
| <myTags:header fontColor=*"#660099"*>  we take Take note the latest version can be found in the libusb-win32 Sourceforge........  </myTags:header> |

1. 标记文件中声明与使用

|  |
| --- |
| <%@ attribute name=*"fontColor"* required=*"true"* %>  <%@ tag body-content=*"tagdependent"* %>  <img src=*"head\_small.jpg"* /> <br>  <em><strong><font color=*"*${fontColor }*"*>  <jsp:doBody />  </font></strong></em><br> |

5，标记文件一般放置在WEB-INF中，但若标记文件放置在jar的子目录下，则必须为该标记文件设置一个tld，表示所在位置

**二 标记处理类**

1，标记处理类可以完成复杂灵活的操作，用标记表示标记处理类，不仅在页面上显示简单，而且便于其他人员操作。

2，当读到jsp语言某个标记时，会自动地调用该标记所对应的标记处理类的doTag方法；

而容器是如何找到该标记所对应的标记处理类？则需要tld文件，容器会根据tld文件找到所读标记的标记处理类。具体设置与流程参照第九章-三-定制标记库。

3，标记处理类主要有两种类型：简单(jsp2.0以后)与传统

4，简单标记处理类要实现SimpleSupportTag类，并覆盖其doTag方法，当jsp要使用该标记时，会自动调用该标记处理类的doTag方法。其生命周期：

1. web容器加载标记处理类进内存
2. 当某个jsp使用该标记时，容器首先会实例化标记处理类生成对象。
3. 容器调用标记处理类的setJspContext方法，这为处理类提供了一个pageContext对象的引用
4. 当标记是嵌套的，即该标记是被其他的标记调用的，则调用setParent()方法与外层的标记建立联系
5. 当标记处理类存在属性时（调用标记使用了属性），则调用标记每个属性的设置方法（标记的属性设置相当于Bean中属性设置）
6. 当标记处理类的tld中<body-content>未声明为empty，且标记有体时，则需要调用setJspBody(JspFragment)方法
7. 最后调用doTag()进行处理。
8. doTag方法结束，就回撤销处理类实例，并不会重用处理类实例。

5，当使用标记替代标记处理类时，想要标记处理类显示实时信息，则需要向标记处理类提供信息，一般有2种方法：

1. 调用标记时使用属性，利用属性向标记标记传递信息：

当使用属性向标记处理类传递信息时，要求

1. 标记处理类必须设置相应属性，并实现Bean的设置属性方法用于接收属性。
2. 处理类对应的tld必须利用<attribute>声明属性相关信息：name、required、rtexprvalue.
3. jsp利用属性传递信息时必须遵守tld设置要求。
4. 在调用标记时将信息放置标记体中，处理类在doTag中利用getJspBody()获取并处理体信息。

（3）具体实现：

1. jsp中设置属性、标记体

|  |
| --- |
| <mine:advice username=*"Jone"* >  I love this choice!  </mine:advice> |

1. 标记处理类中处理

|  |
| --- |
| **public** **void** doTag() **throws** IOException, JspException {  getJspContext().getOut().write("Hello "+ username +"<br>");  getJspContext().getOut().write("Your advice is "+getAdvice());  getJspBody().invoke(**null**);  } |

1. 解析：

a)当读取到标记含有属性和体时，容器在找到标记处理类执行doTag()方法之前，会先调用处理类为属性设置的属性设置方法（setXXX()）根据标记中的值设置属性值，然后再调用setJspBody(JspFragment)方法，将与包含有体中内容的jspfragment联系起来。

后面在doTag方法即可获取属性值，并利用getJspBody()获取；想要获取体内容，可利用getJspBody()获取jspfragment对象，并进行下一步操作。

b)jspFragment对象中封装了标记体的内容，其主要有2个方法：getJspContext()、invoke(java.io.write)，invoke方法是将体中的内容封装到某个输出流，当writer设为null时，即把体发送到响应，在响应页面中输出。

6，（1）当标记体中设置的是使用属性的el表达式，可以在标记处理类doTag方法中使用getJspContext()获取pageContext对象的引用，再在页面作用域中设置属性（setAttribute），使得el表达式能获取所取得属性值。注意要在将体中内容输出之前(invoke)，在jspContext中设置属性值，否则输出后未设置属性值，输出为空。

1. 若标记属性取值用的是使用属性的el表达式（${attribute}），则在doTag设置属性值不能传递到标记属性，是由于在调用doTag之前先调用标记属性的set方法，若在此前未设置值，则无法用set设置，也就无法传递到标记的属性中，进而再传入doTag中。需要在调用set前已设好el属性的值。（体是由jspfragment封装，并未直接设置）。
2. 如果标记体中el表达式是一个值，但我们的目标是向为集合中的每一个元素生成一行，我们可以在doTag中为标记设置循环，并每次循环都调用体（invoke）。
3. 标记中设置el表达式

|  |
| --- |
| <mine:advice>  ${username} I love this choice!<br>  </mine:advice> |

1. doTag循环体

|  |
| --- |
| String[] users = {"Mike", "Dan", "Jonh"};  **for**(**int** i = 0; i < users.length; i++){  String names = users[i];  getJspContext().setAttribute("username", names); //设置属性  getJspBody().invoke(**null**); //循环将体输出到响应  } |

始终记住，调用标记就是调用标记处理类的doTag方法。

7，标记体的内容或doTag中的内容都不是直接输出到响应，而是

体中内容，getJspBody().invoke(null); //输出到响应

DoTag内容，getJspContext().getOut.write() //输出到响应

8，jsp调用了标记，若想要设置标记出现错误时的处理（类似异常处理），可以在doTag中设置抛出SkipPageException异常（throw new SkipPageException()）,则在jsp调用标签之后的页面都不会被显示，但throw new SkipPageException()之前的所有内容都能显示，包括doTag输出到响应的内容。

1. 如果一个页面包含另一个页面，而被包含页面调用标记，且标记抛出异常，但SkipPageExceptionzhi只停止显示直接调用标记的页面，外层页面照常显示。

9，传统---

1. **部署web应用**

1，Tomcat中将整个web应用进行打包压缩，给定一个.war（web archive)，即为war文件，当容器要访问该web应用时，会解开war文件，并创建web应用上下文目录。

在一个war文件中，可以在META-INF/MANIFEST.MF文件中声明库依赖性，使得在部署时使得容器找到应用依赖的包和类。

2，web应用部署环境:

1. 静态html和jsp可以放在web应用根目录下或网页子目录下，但不能放在web-inf或meta-inf目录下，会影响访问。
2. DD必须命名为web.xml，且必须放在web-inf下（不能子目录）
3. 标记文件(.tag)必须放在web-inf/tags目录及其子目录下。
4. 所有类文件（servlet、监听类、辅助类、bean、标记处理类等）的包结构都必须放在web-inf/class下。
5. Jar类的包结构必须放在jar中，且jar目录必须放在WEB-INF/lib目录下。
6. WEB-INF目录下的meta-inf必须放在放在jar目录下，tld文件必须放在meta-inf目录下。
7. 设置库依赖目录及文件：META-INF/MANIFEST.MF

3，servlet映射

1. servlet映射在外部采用的url模式不一定是真实地址名，只是逻辑地址名。设置url模式时，有三种设置方式：
   1. 完全匹配

|  |
| --- |
| <url-pattern>/Beer/Select.do</url-pattern> |

* 1. 目录匹配

|  |
| --- |
| <url-pattern>/Beer/\*</url-pattern> |

* 1. 扩展名匹配

|  |
| --- |
| <url-pattern> \*.do</url-pattern> |

a,b必须以/开头，c不能以/开头，需用\*开头。

1. 匹配原则：
   1. 容器先查找完全匹配，若未找到，再查找目录匹配，最后再查找扩展名匹配。
   2. 若一个请求和多个url模式匹配，容器会选择最长匹配。如请求/foo/bar/myStuff.do，它能匹配/foo/bar/\*、/foo/\*，但其选择/foo/bar/\*。

4，当客户请求是一个目录，而未指向任何servlet或jsp，则需要在DD中配置欢迎界面。

|  |
| --- |
| <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>defalut.jsp</welcome-file>  </welcome-file-list> |

当客户请求的是目录时，容器会从<welcome-file-list>第一个欢迎文件开始，在所请求目录下寻找欢迎文件。

5，servlet初始化：

1. 容器会在第一个请求到来时初始化servlet类，这说明第一个客户将承受类加载、实例化、初始化等一系列开销，才能正常工作。我们可以在DD中为servlet设置<load-on-startup>，只要其值不为0，则容器会在部署时提前加载并初始化该servlet。

|  |
| --- |
| <servlet>  <servlet-name>beer</servlet-name>  <servlet-class>com.web.BeerSelect</servlet-class> <load-on-startup>1</load-on-startup>  </servlet> |

1. 当想预加载多个servlet时，容器会根据<load-on-startup>的值绝对加载的顺序，如果值相同，容器会根据其子集被初始化的顺序来加载。
2. **web应用安全**

1，web网站需要预防：假冒者（Impersonator非法登录）、非法升级者（Upgrader，非法权限升级）、窃听者（EavesDropper，非法获取信息），应做到：

1. 认证：需用提供用户名、密码进行用户认证
2. 授权：对于认证的用户，授予不同的权限
3. 机密性和数据完整性：对数据进行加密处理

2，当用户请求某个资源时，容器查找资源，并确认是否为有安全限制的资源，若是，容器必须对用户进行认证，确认用户身份（一般为提供用户名和密码）；当用户完成认证后，容器还必须查看该用户是否有权限访问该资源。

**一，授权**

1，Tomcat/conf存在一个名为tomcat-users.xml的文件，上面标注了安全权限的角色与用户。

|  |
| --- |
| <tomcat-users ...>  <role rolename="Admin"/>  <role rolename="Member"/>  <role rolename="Guest"/>  <user username="Alime" password="root" roles="Admin,Member,Guest"/>  <user username="John" password="coder" roles="Member,Guest"/>  <user username="Dan" password="newby" roles="Guest"/>  </tomcat-users> |

2，将用户文件的角色(role)映射到部署描述文件DD的建立的角色。

|  |
| --- |
| <security-role>  <role-name>Admin</role-name>  </security-role>  <security-role>  <role-name>Member</role-name>  </security-role>  <security-role>  <role-name>Guest</role-name>  </security-role>  <!-- 基本认证 -->  <login-config>  <auth-method>BASIC</auth-method>  </login-config> |

3，定义资源/方法安全约束，在<security-constraint>中申明约束资源、方法以及权限角色。

|  |
| --- |
| <security-constraint>  <web-resource-collection>  <web-resource-name>check</web-resource-name>  <description>url</description>    <url-pattern>/CheckName</url-pattern>  <url-pattern>/el</url-pattern>    <http-method>GET</http-method>  <http-method>POST</http-method>  </web-resource-collection>    <auth-constraint>  <role-name>Admin</role-name>  <role-name>Member</role-name>  </auth-constraint>    </security-constraint> |

<web-resource-collection>用于申明带有安全限制的资源以及http方法，<auth-constraint>申明可以访问受限资源的用户。

4，<web-resource-collection>中有两个主要子类：<url-pattern>和<http-method>，url至少一个，http可以为空。

1. <web-resource-name></web-resource-name>用于描述资源名，可以任意指定。
2. <description>可选用于描述。
3. <http-method>方法包括：get、post、put、trace、delete、head、options
4. 如果没有指定任何http方法，即没有<http-method>，则所有http方法都是受限的；若指定了http方法，则只有指定的方法是受限的，其他方法不受约束。
5. <auth-constraint>作用于<web-resource-collection>受限的资源和http方法。

5，角色申明

1. <auth-constraint>中<role-name>可选，选入<role-name>的角色能访问受限资源。
2. 当有<auth-constraint>但没有<role-name>，则所有用户都会被拒绝。
3. <role-name>\*</role-name>所有用户都得到许可。
4. 用户名区分大小写。
5. 在<security-constraint>中<auth-constraint>是可选的，若没有<auth-constraint>，则所有用户都会得到许可。

6，当有多个<security-constraint>元素交互时，即<security-constraint>存储相同的受限资源和相同的角色，但授予了不同的权限，如一个允许guest访问，一个不允许guest访问。其合并规则为：

1. 如果两个非空的<auth-constraint>应用于同一受限资源，则两个<auth-constraint>所有角色的并集都允许访问。
2. 如果一个<auth-constraint>的角色设为“\*”,则所有角色都允许访问。
3. 如果一个<auth-constraint>的角色为空时，则所有角色都不允许访问。
4. 如果一个没有<auth-constraint>，则所有角色都允许访问。

7，isUserInRole()可以不在http方法层次（get、post等），而直接在服务方法（doGet/doPost）的某些部分完成授权。使用流程：

1. 调用isUserInRole()前对用户进行认证，若认证失败直接返回FALSE。
2. 认证成功后，容器调用request.isUserInRole(string)得到参数，与用户的角色进行比较。
3. 若用户被映射到该角色，即返回TRUE，即可进行授权操作。

|  |
| --- |
| If(request.isUserInRole(“manager”){  ...  } |

8，当将安全角色名硬编码到servlet中时，程序员设置的安全角色名可能并不是一个真正的角色，或者多家部门进行共同开发，使用的安全角色名不一致，可以在DD中将servlet中硬编码的安全角色名映射到正式的<security-role>中去。

|  |
| --- |
| <servlet>  <servlet-name>...</servlet-name>  <servlet-class>硬编码所在的servlet</servlet-class>  <security-role-ref>  <role-name>manager</role-name>  <role-link>Admin</role-link>  </security-role-ref>  </servlet> |

通过<security-role-ref>将manager映射到正式的角色Admin

**二，认证**

1，对于j2ee服务器，认证可以总结为：询问一个用户名和口令，再验证是否匹配。

容器可以提供4中类型的认证，其区别就是所传输的用户名和口令有多安全。

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

2，4中认证类型：

1. （基本）BASIC：以一种编码的形式（未加密）传递用户名和口令，由于其编码机制Base64广为所知，其安全性并不高。
2. （摘要）DIGEST：以一种更安全的形式传输用户信息，但未广泛使用，并不要求j2ee容器支持。
3. （客户整数）CLIENT-CERT：使用公共密钥证书（Public Key Certificate，PKC）传递信息，十分安全，缺点是需要客户必须要有一个证书才能认证，用于B2B应用。
4. （表单）FORM：允许根据合法的HTML建立自己的登录表单，传输最不安全，用户名和口令都在请求中发送，并未加密。

3，区别：

1. 前3种都使用浏览器标准弹出窗口进行登录，第4种则在HTML中表单登录。
2. 表单认证的DD设置时，需要设置登录表单的HTML网页及错误网页。

|  |
| --- |
| <login-config>  <auth-method>FORM</auth-method>    <form-login-config>  <form-login-page>/index.html</form-login-page>  <form-error-page>/loginError.html</form-error-page>  </form-login-config>  </login-config> |

4，form认证：

1. DD中设置 <login-config>
2. 创建一个HTML登录表单，且html登录表单中form的action必须是j\_security\_check，用户名必须设为j\_username，口令必须设置为j\_password。

|  |
| --- |
| <form action=*"CheckName"* method=*"j\_security\_check"*>  input your name:<input type=*"text"* name=*"j\_username"* />  input your password: <input type=*"password"* name=*"j\_password"*/>  <input type=*"submit"* value=*"Submit"*>  </form> |

1. 创建一个HTML错误表单

**三，机密性和数据完整性**

1，当想要使用form定制登录表单时，又想保护数据的机密性，可以使用基于ssl的https维持一个有保护的传输层链接，使得传输时数据保密。方式：

|  |
| --- |
| <security-constraint>  <web-resource-collection>  <web-resource-name>check</web-resource-name>  <url-pattern>/CheckName</url-pattern>  <http-method>POST</http-method>  </web-resource-collection>    <auth-constraint>  <role-name>Admin</role-name>  </auth-constraint>    <user-data-constraint>  <transport-guarantee>CONFIDENTIAL</transport-guarantee>  </user-data-constraint>  </security-constraint> |

3个子元素可以连起来读：只有角色为Admin的用户才能对CheckName做Post请求，且保证传输是安全的。

2，传输保护有3种形式：

1. NONE:默认值，没有数据保护
2. INTEGRAL:数据在传输过程中不可更改
3. CONFIDENT:数据在传输过程中不可见

2,3都能提供保护数据的完整性和机密性，一般建议使用confident。

3，用户保护请求数据过程：  
（1）用户使用常规的http请求请求某一资源，容器找到DD中发现该资源需要传输安全保证，而用户所用的是不安全的传输请求。

1. 容器便向浏览器发送301响应，即告诉客户需要使用一个安全传输完成请求的重定向。
2. 客户通过一个安全的请求连接请求资源（https）,容器找到资源，发现为受限资源，需要进行认证，于是向浏览器发送一个401。
3. 浏览器收到401重新发送一个安全请求，并在请求首部提供了用户数据。容器收到用户数据后对用户进行认证。

也就是说，使用安全传输的请求一共要进行3次请求，而不安全的只要2次，少了第二次请求（4）。

1. **过滤器与包装器**

**一，过滤器**

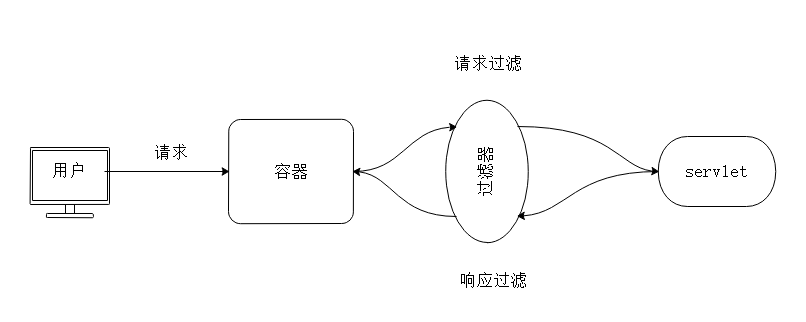
1，过滤器是Java组件，用于在请求发送到servlet之前，过滤器可以拦截和获取请求，另外，在servlet工作结束后，将响应返回到用户（容器）之前，可以用过滤器处理响应。

1.1 请求过滤器：

1. 完成安全检查
2. 重新格式化请求首部或体
3. 建立请求审计或日记

1.2 响应过滤器：

1. 压缩响应流
2. 追加或修改响应流
3. 创建一个完全不一样的响应



2，容器根据DD中的声明来确定何时调用响应，在DD中要明确对于哪个资源哪个url调用哪些过滤器。

3，过滤器filter类似于servlet：

1. 过滤器有属于自己的api，普通的Java一旦实现了filter接口就成了过滤器，可以访问servletcontext，还可以与其他过滤器链接。
2. 容器管理过滤器的生命周期，过滤器也有自己的生命周期：init、destroy、doFilter。
3. 过滤器设置需要在DD声明。

4，过滤器的生命周期：

每个过滤器必须实现Filter接口的三个方法：init、destroy、doFilter

1. init():当容器决定实例化一个过滤器类时，就必须在调用过滤器前在init()中完成所有初始化工作。
2. doFilter():容器对当前请求调用过滤器时，就会doFilter()方法，有3个参数：

ServletRequest、ServletResponse、FilterChain，过滤器的功能在doFilter中实现。

1. destroy(): 容器删除过滤器实例时，调用destroy()完成清理。

5，过滤器调用：

5.1过滤器相当于一个个模块，在FilterChain中进行组织确定调用过滤器顺序；

在doFilter方法中，调用FilterChain.doFilter()确定调用当前过滤器后面的过滤器的doFilter()方法，若为最后一个过滤器，则调用Servlet的service()方法。

5.2，可以将过滤器调用看成在栈的压入弹出，如在请求过滤时，先调用过滤器1，2，再调用Servlet的service()方法；响应时，将Servlet弹出，再调用过滤器2,1的响应过滤。

6，DD中配置过滤器：

1. 声明过滤器 <filter>声明，其中<init-param>可省
2. 将过滤器映射到想要过滤的web资源：映射有两种方式

2.1，过滤器映射到URL模式

2.2，过滤器映射到Servlet名称

1. 组织映射，创建过滤器调用序列

|  |
| --- |
| <filter>  <filter-name>BeerRequest</filter-name>  <filter-class></filter-class>    <init-param>  <param-name>LoginFileName</param-name>  <param-value>UserLogin.txt</param-value>  </init-param>  </filter>  <!--映射到URL模式-->  <filter-mapping>  <filter-name>BeerRequest</filter-name>  <url-pattern>\*.do</url-pattern>  </filter-mapping>  <!--映射到Servlet名-->  <filter-mapping>  <filter-name>BeerRequest</filter-name>  <servlet-name>test</servlet-name>  </filter-mapping> |

6.2（1）<filter>中必须有<filter-name><filter-class>

1. <filter-mapping>必须有<filter-name>，用于指向过滤器元素，

<url-pattern><servlet-name>必须有一个。

6.3，容器确定过滤器调用顺序：当一个web资源有多个过滤器映射时，

1. 匹配URL模式的过滤器放在FilterChain的前面(先被调用)，在FilterChain中的顺序和它们在DD中声明的顺序一致。
2. 匹配Servlet名称的过滤器放在FilterChain后面，位置也参考声明的顺序。

7，过滤请求转发器：过滤器不仅可过滤来自用户的请求，还可以过滤来自请求转发或错误处理等的请求，可以在DD中为请求转发的web资源申请一个过滤器映射。

|  |
| --- |
| <filter-mapping>  <filter-name>BeerRequest</filter-name>  <url-pattern>\*.do</url-pattern>  <dispatcher>REQUEST</dispatcher>  <dispatcher>INCLUDE</dispatcher>  <dispatcher>FORWARD</dispatcher>  <dispatcher>ERROR</dispatcher>  </filter-mapping> |

1. REQUEST：默认，对客户请求启动过滤器
2. INCLUDE: 对include()调用的请求启动过滤器
3. FORWARD: 对forward()转发的请求启动过滤器
4. ERROR: 对错误处理器调用的资源启动过滤器

8，对一个响应启动过滤器：过滤器首先完成请求过滤，然后调用FilterChain.doFilter移交请求，当Servlet完成请求后，控制返回到原来过滤器的doFilter方法，再完成响应过滤。（类似栈）

|  |
| --- |
| **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  //请求过滤放在前面  ....  chain.doFilter(request, response);  //响应过滤放在后面  ....  } |

但注意若未做任何处理，当Servlet完成请求后，响应会直接返回到用户而不会经过过滤器。由于Servlet的响应是通过HttpServletResponse响应对象获取响应输出流的（response.getWriter()）,而响应又是通过chain.doFilter(request, response)传递过去的，因此我们可以设置一个定制响应对象传到Servlet，Servlet在完成请求后通过定制响应可将响应传到过滤器。定制响应对象可以通过包装器来实现。

### 二，包装器

1，包装器为所包装的东西实现了所需的所有方法，并将所有调用委托给底层的请求或响应对象。（装饰模式）

2，sun为请求或响应对象实现了4个特定的包装对象，我们可以扩展该包装类实现自己定制的对象。

1. ServletRequestWrapper （2）HttpServletRequestWrapper
2. ServletResponseWrapper （4）HttpServletResponseWrapper

3，响应压缩过滤器设计

1. 使用响应包装器包装响应对象，并定制压缩输出流：继承响应包装对象并扩展。

|  |
| --- |
| **public** **class** CompressionResponseWrapper **extends** HttpServletResponseWrapper{    **private** GZIPServletOutputStream servletGzipOS = **null**; //响应的压缩输出流  **private** PrintWriter pw = **null**;    //构建包装器对象，包装器对象要包装响应对象  **public** CompressionResponseWrapper(HttpServletResponse response) {  **super**(response);  }    **public** GZIPOutputStream getGZIPOutputStream() {  **return** **this**.servletGzipOS.internalGzipOS;  }    **public** GZIPServletOutputStream getOutputSreamm() **throws** IOException  {  //定制压缩流  ...  }  **public** PrintWriter getWriter() **throws** IOException{  ...  }  } |

1. 在过滤器的doFilter方法设置请求和响应过滤

|  |
| --- |
| **public** **class** MyCompressionFilter **implements** Filter {      **public** **void** init(FilterConfig fConfig) **throws** ServletException {  //初始化，主要是保存配置对象，还可保存对上下文的引用，以完成日记记录  **this**.cfg = fConfig;  ctx = cfg.getServletContext();  ctx.log(cfg.getFilterName()+" initialized.");  }    **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  //请求过滤放在前面    HttpServletRequest req = (HttpServletRequest)request;  HttpServletResponse resp = (HttpServletResponse)response;    String valid\_encoding = req.getHeader("Accept-Encoding");    **if**(valid\_encoding.indexOf("gzip") > -1){  //客户接受gzip压缩，压缩包装器包装响应对象  CompressionResponseWrapper crw = **new** CompressionResponseWrapper(resp);  crw.setHeader("Content-Encoding", "gzip");  //链到下一个组件，将定制响应对象添加到doFilter  chain.doFilter(request, crw);  //响应过滤放在后面,获取压缩响应流  GZIPOutputStream gzos = crw.getGZIPOutputStream();  gzos.flush();    ctx.log(cfg.getFilterName()+" finished the request.");  }**else**{  ctx.log(cfg.getFilterName()+" no encoding.");  }  }    **public** **void** destroy() {  // **TODO** Auto-generated method stub  ctx = **null**;  cfg = **null**;  }  } |

1. **企业设计模式**

**一，模式**

1，大型应用中常见结构体系是按照功能层次配置硬件，如web应用中jsp，servlet等放置在“表示层”，业务逻辑放置在“业务层”。

2，软件设计原则：

1. 遵循接口编写代码
2. 关注分离点，增加内聚
3. 隐藏复杂性
4. 松耦合：两个类相互通信时，彼此之间的了解越少，相互的耦合就越松；如类A想调用类B的方法，常见做法是在两者之间增加一个接口，类B实现接口，类A通过此接口使用类B。
5. 远程代理
6. 增强声明式控制：以声明的方式控制应用

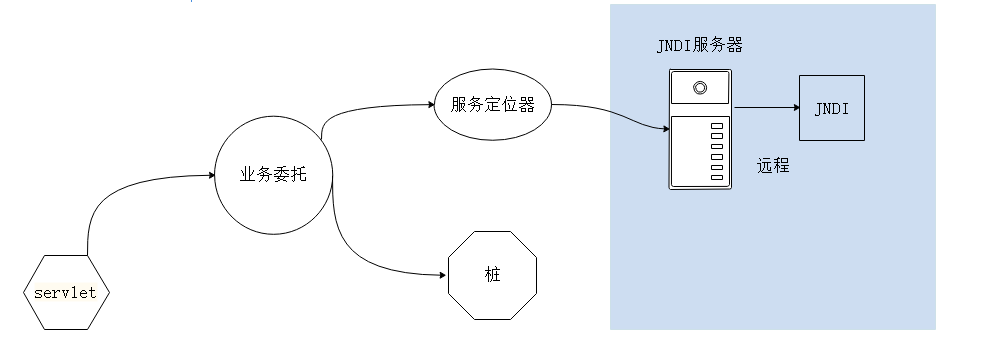
3，当一个web应用的组件分布在不同的服务器上（JVM）时，组件之间需要远程调用，这就需要JNDI或RMI；

1. JNDI（Java Naming and Directory interface,Java命名和目录接口）：当一些对象需要被其他网络上的组件调用时，可以向JNDI注册这些对象，其他组件可以使用JNDI查找获取对象。
2. RMI（Remote Method Invocation,远程方法调用）：当远程客户需要使用某个对象时，通过RMI创建一个代理，并把这个对象注册到注册表上。远程客户可以在注册表上查找，获取该远程代理的副本（桩），客户就会在该远程代理上调用，假装它是服务器，远程代理处理所有的通信细节。（服务端也有个代理，称为“骨架”）

4，模型简化：

4.1，web应用中，如一servlet请求远程代理，由于在servlet直接进行查找JNDI使得功能混杂，可以创建一个中间对象，用于获取远程代理，中间对象查找JNDI并获取桩并完成业务，servlet只需调用中间对象即可，这就是业务委托。

4.2，针对每个业务委托都需要查找JNDI，存在重复代码，可设置一个服务定位器，用于完成JNDI查询的所有逻辑，业务委托只需处理业务方法



5，当jsp获取远程模型的信息时，需要通过业务委托得到一个“顾客桩”，jsp每次调用一个方法时，“顾客桩”都会使用一个远程调用，将产生通信消耗。可以使用传输对象进行大量数据远程传递。传输对象是一个可串行化的Java对象，其中包含了大量的实例对象，可通过业务委托一次获取大量数据存入传输对象中，jsp从传输对象中得到数据。（缺点是数据更新不及时）。

6，MVC模式改进——Struts1.1

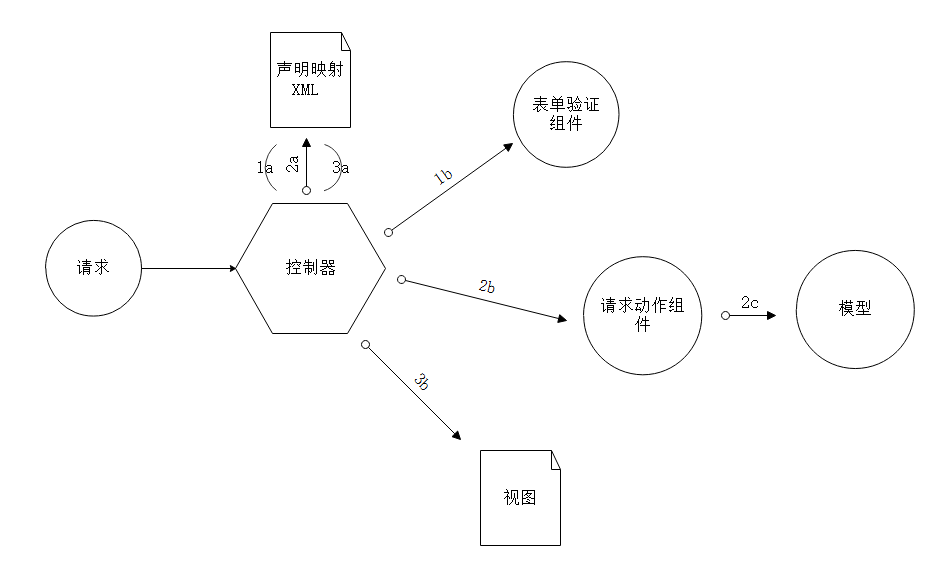
6.1 改进MVC模式；

|  |  |
| --- | --- |
| 控制器3大任务 | 改进 |
| 获取和处理请求参数 | 交给一个单独的表单验证组件，由它获取请求参数、完成转换、进行验证、处理验证错误，并创建一个对象保存参数值。 |
| 调用模型 | 不把模型硬编码到控制器中，以声明的方式调用模型组件 |
| 分派到视图 | 以声明的方式分派到视图 |

6.2，新的控制器伪代码

|  |
| --- |
| **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  // **TODO** Auto-generated method stub  //以声明方式调用验证组件，验证错误也由其处理    //以声明方式调用请求处理组件来调用模型组件    //以声明方式分派组图  } |

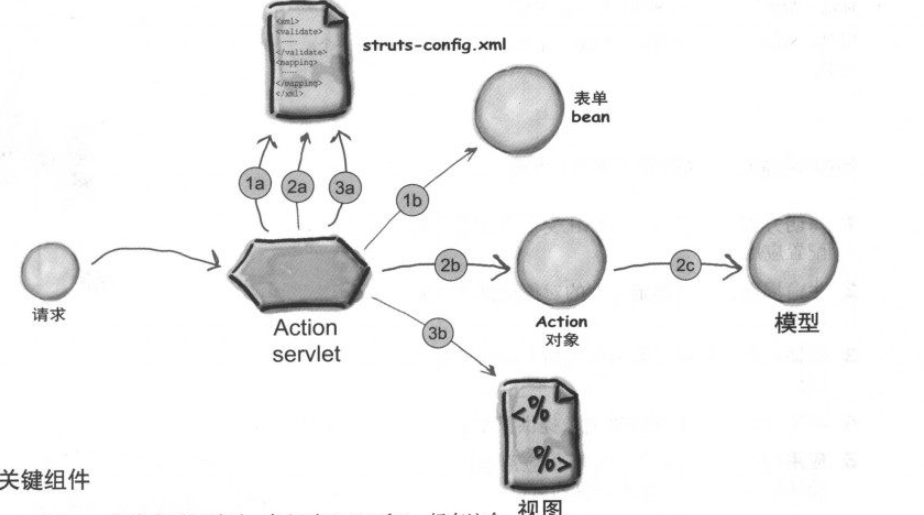
6.3，体系结构



6.4，过程：

1. 接收到请求后，控制器在声明XML文件中找到对应的表单验证组件，控制器调用该组件，发送请求，若验证组件发现错误，就告诉控制器调用哪个视图。
2. 通过声明XML找到请求动作组件，由它调用模型。
3. 通过声明XML文件找到并调用视图。

6.5，该体系框架即为Struts框架，其对应名称如下：



1. Action Servlet：每一个应用只有一个Action Servlet，由Struts提供。
2. Bean表单：对于应用要处理的各个html表单，都要为其设置一个表单Bean。都是JavaBean，一旦Action Servlet调用了Bean表单的设置方法（传入表单参数），它就会调用bean的validate()方法。
3. Action对象：action映射到用例中的一个活动，有一个类似回调的方法execute()，可以获取验证过的参数，并调用模型组件，可认为是一个“轻型的Servlet”。
4. Struts-config.xml:Struts特殊的部署文件，用于描述Action Servlet到各组件的声明映射，主要有：请求URL到Action、Action到表单bean以及Action到视图。

6.6，Struts与Servlet容器相似的地方：

1. 声明方式：以XML声明方式配置应用
2. 生命周期：
3. 回调：完成生命周期方法的自动回调
4. API
5. 应用控制

**第十五章 Servlet高级**

**一，Servlet上传与下载文件**

1.1 jsp中上传标签是input, type为file，注意：

上传文件时form表单method一定是post，其编码类型是multipart/form-data.

|  |
| --- |
| <form action=*"Upload"* method=*"post"* enctype=*"multipart/form-data"* >  上传用户：<input type=*"text"* name=*"username"*/><br>  file1: <input type=*"file"* name=*"file1"*/><br><br>  file2: <input type=*"file"* name=*"file2"*/><br><br>  description：<input type=*"text"* name=*"desc"*><br>  <input type=*"submit"* value=*"提交"*>  </form> |

1.2，为简化对multipart/form-data类型数据的处理，Apache提供了两个jar文件实现文件上传：commons-fileupload-1.3.1.jar，commons-io-2.5.jar；fileupload负责文件上传，io负责输入输出（fileupload基于io）。

1.3，针对类型为multipart/form-data的http请求，fileupload将请求正文（post）包含的复合表单的每一个input部分都看成一个FileItem对象，FileItem对象有两个类型：（1）formField类型：普通的表单域类型，文本框（text）、提交按钮等都是。

1. 非formField类型：上传文件类型，它包含了上传文件的数据。

1.4，FileItemFactory是创建FileItem的工厂，DiskFileItem和DiskFileItemFactory分别实现了FileItem接口和FileItemFactory接口，DiskFileItem对象能将文件写到本地磁盘。

1.5，处理上传文件流程：

1. 创建DiskFileItemFactory工厂，并进行相应设置
2. 利用工厂创建DiskFileItem的子类ServletFileUpload，利用其parseRequest(request)方法解析请求正文的信息，获得List。
3. 遍历List对象，对于上传文件保存到本地硬盘。

|  |
| --- |
| **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {    String tempPath1 = request.getServletContext().getInitParameter("tempPath");  String stagePath1 = request.getServletContext().getInitParameter("filePath");  //获取绝对路径  tempPath = **this**.getServletContext().getRealPath(tempPath1);  stagePath = **this**.getServletContext().getRealPath(stagePath1);    System.***out***.println(tempPath);  System.***out***.println(stagePath);    File repository = **new** File(tempPath);  **if**(!repository.exists()){  //不存在创建临时目录  repository.mkdir();  }  String message = **null**;    **try**{  //创建一个基于硬盘的FileItem工厂  DiskFileItemFactory factory = **new** DiskFileItemFactory();  //设置向硬盘写入数据所有缓冲区的大小,当上传文件超过缓冲区大小时，生成一个临时文件存放在临时目录  factory.setSizeThreshold(4\*1024);  //设置临时目录  factory.setRepository(repository);  //创建一个文件上传处理器  ServletFileUpload upload = **new** ServletFileUpload(factory);  //设置单个文件上传的最大大小  upload.setFileSizeMax(1\*1024\*1024);  //设置允许所有上传文件的最大量  upload.setSizeMax(4\*1024\*1024);    //解决中文乱码  upload.setHeaderEncoding("utf-8");    //监听文件的上传进度  upload.setProgressListener(**new** ProgressListener() {    **public** **void** update(**long** pBytesRead, **long** pContentLength, **int** arg2) {  System.***out***.println("文件大小：" + pContentLength + " , 当前已上传： "+ pBytesRead);  }  });    //解析请求参数，返回一组FileItem的List集合  List<FileItem> items = upload.parseRequest(request);  //迭代  Iterator<FileItem> it = items.iterator();  **while**(it.hasNext()){  //获取fileitem，分析是为普通的表单域，还是上传文件  FileItem fileitem = it.next();    **if**(fileitem.isFormField()){  //普通表单域  // String name = fileitem.getString("username");  String name = fileitem.getFieldName();  //解决中文乱码  String desc = fileitem.getString("utf-8");  System.***out***.println("name = "+name+", desc="+desc);    }**else**{  //上传文件  String name = fileitem.getName(); //获得上传文件名(包括路径)    **int** index = name.lastIndexOf("\\");  String filename = name.substring(index+1, name.length()); //获取文件名  String fileExtName = filename.substring(filename.lastIndexOf(".")+1); //获取文件的扩展名  System.***out***.println("上传文件扩展名："+ fileExtName);  **long** size = fileitem.getSize();    //创建保存地址  String saveName = makeName(filename);  System.***out***.println(saveName + " is saved-----");  File path = **new** File(stagePath);  **if**(!path.exists()){  path.mkdir();  }    File uploadedFile = **new** File(stagePath + "\\" + saveName);  System.***out***.println(uploadedFile.getName());    //创建文件输出流  FileOutputStream output = **new** FileOutputStream(uploadedFile);    System.***out***.println(name + " is saved");    //获取item的输入流  InputStream in = fileitem.getInputStream();    //创建缓冲区  **byte**[] buf = **new** **byte**[1024];  **int** len = 0;  **while**((len = in.read(buf)) != -1){  output.write(buf);  }    //关闭流  output.close();  in.close();    //打印文件信息  System.***out***.println(name + " is saved");  System.***out***.println("The size of " + filename + " is "+ size);  message = "上传成功";  }  }    }**catch**(Exception e){  message = "上传失败";  **throw** **new** ServletException();  }    request.setAttribute("msg", message);  request.getRequestDispatcher("result.jsp").forward(request, response);  }    **public** String makeName(String name){  //为防止同一目录下文件覆盖现象，为上传文件设置一个唯一的文件名  **return** UUID.*randomUUID*().toString() + "\_" +name;  }    //为防止一目录下出现太多文件，根据文件名将文件用hash算法打散  **public** String makePath(String name, String path){  **int** hashcode = name.hashCode();  //打散目录  **int** dir1 = hashcode & 0xf; //0-15位  **int** dir2 = (hashcode & 0xf0) >> 4; //0-15    //构造目录  String dir = path + "\\" + dir1 + "\\" + dir2;    File file = **new** File(dir);  //如果目录不存在创建  **if**(!file.exists()){  file.mkdir();  }  System.***out***.println(dir);  **return** dir;  } |

**二，JavaMail**