# 过滤器入门

## 引入

### 场景1： 编写servlet，接收参数：

request.getParameter（） /getParameterValues（）

如果不做任何处理，就出现中文乱码问题。

解决编码问题： request.setCharacterEncoding("utf-8");

问题：如果在项目中的每一个servlet都加上request.setCharacterEncoding("utf-8");

显示代码重复啰嗦。能不能把这部分公共代码抽取处理，放在一个地方执行？？？？

### 场景2：

登录页面 -> 输入用户民或密码 -》 后台检查是否成功 -》 登录成功，看到用户首页

( 用户名：xxxx )

把登录数据放在session域对象中(user)

用户首页如果不登录是看不到的:

判断用户登录权限代码：

HttpSession session = request.getSession(false);

if(session==null){

跳转到登陆页面

}else{

String user = (String)session.getAttribute("user");

if(user==null){

跳转到登录页面

}

}

用户资源修改页面（需要用户登录才能访问）

HttpSession session = request.getSession(false);

if(session==null){

跳转到登陆页面

}else{

String user = (String)session.getAttribute("user");

if(user==null){

跳转到登录页面

}

}

如果用户不登录，直接访问用户主页，跳转到登录页面

在其他需要登录才能访问的页面中，同样也需要加上验证用户是否登录成功代码。

问题： 能不能把这部分公共验证用户是否登录成功代码抽取处理，在一个地方执行？？

结论： 以上两种场景出现的问题，可以使用过滤器（Filter）解决！！！！

## 过滤器简介

1. 过滤器其实就是一个Filter接口， javax.servet.Filter

2．过滤器就是一个对象，可以在请求一个资源（静态或动态资源），或响应一个资源，或请求和响应一个资源的时候，执行过滤任务！！！！

3．过滤器如何被执行？

过滤器也需要交给tomcat服务器运行！！！！

### **Servlet的三大组件：**

**Servlet接口**

**Filter接口**

**Listener接口**

**特点：**

**1．都需要交给web服务器运行**

**2．在web.xml文件中配置**

## 过滤器的生命周期

构造方法： 在web应用加载时创建过滤器对象。只执行一次。证明过滤器在web服务器中是单实例的。

init方法： 在创建完过滤器对象之后被调用。只执行一次

doFilter方法： 执行过滤任务方法。执行多次。

destroy方法： web服务器停止或者web应用重新加载，销毁过滤器对象。

## 过滤器编写步骤：

1．编写一个java类，实现Filter接口，并实现其中的所有方法

2．在web.xml文件中配置Filter

|  |
| --- |
| <!-- 过滤器配置 -->  <filter>  <!-- 内部名称 -->  <filter-name>HelloFilter</filter-name>  <!-- 类全名：包+简单类名 -->  <filter-class>gz.itcast.a\_hello.HelloFilter</filter-class>  </filter>  <!-- 过滤器映射配置 -->  <filter-mapping>  <!-- 内部名称，和上面的名称保持一致！ -->  <filter-name>HelloFilter</filter-name>  <!-- 需要拦截的路径 -->  <url-pattern>/hello</url-pattern>  </filter-mapping> |

5.3 把Filter部署到tomcat服务器运行！！！！

# 过滤器的映射路径

过滤器中的url-pattern: 表示的这个过滤器需要拦截的目标资源路径

（可以servlet路径，也可以是静态资源名称）

Servlet中的url-pattern: 表示访问这个servlet时的路径

url-pattern 浏览器访问目标资源的路径

精确过滤 /hello http://localhost:8080/day21/hello

/itcast/hello http://localhost:8080/day21/itcast/hello

模糊过滤 /\* http://localhost:8080/day21/任意路径

/itcast/\* http://localhost:8080/day21/itcast/任意路径

\*.后缀名 http://localhost:8080/day21/任意路径.后缀名

## 注意：

1．url-pattern要么以斜杠开头，要么以\*开头 例如： hello

2．不能同时使用两个模糊过滤。例如 /\*.do 是非法的

3．如果存在多个需要被过滤的资源，可以写多个url-pattern去过滤。

4．如果是动态资源servlet，可以使用servlet的访问名称，也可以使用内部名称

|  |
| --- |
| <!-- 使用servlet的内部名称 -->  <servlet-name>HelloServlet</servlet-name> |

5．过滤类型：

|  |
| --- |
| <!-- 过滤类型：声明哪种请求才可以被拦截（过滤） -->  <dispatcher>REQUEST</dispatcher><!-- 默认：来自于直接访问的请求才可以被拦截 -->  <dispatcher>FORWARD</dispatcher><!-- 来自于转发的请求才可以被拦截 -->  <dispatcher>INCLUDE</dispatcher><!-- 来自于包含的请求才可以被拦截 -->  <dispatcher>ERROR</dispatcher><!-- 来自于错误的请求才可以被拦截 --> |

# [FilterConfig](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/FilterConfig.html)对象

FilterConfig对象，过滤器配置对象，用于加载过滤器的参数配置

## 过滤器参数使用：

### 1．在web.xml文件中配置

|  |
| --- |
| <!-- 过滤器配置 -->  <filter>  <!-- 内部名称 -->  <filter-name>HelloFilter</filter-name>  <!-- 类全名：包+简单类名 -->  <filter-class>gz.itcast.a\_hello.HelloFilter</filter-class>  <init-param>  <param-name>AAA</param-name>  <param-value>AAA'value</param-value>  </init-param>  <init-param>  <param-name>BBB</param-name>  <param-value>BBB'value</param-value>  </init-param>  </filter> |

### 2．在过滤器器中使用

|  |
| --- |
| /\*\*  \* 2）init初始化方法  \*/  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  System.*out*.println("2)Filter生命周期-init方法");    /\*\*  \* 通过FilterConfig对象得到参数配置信息  \*/  //得到一个参数  System.*out*.println(filterConfig.getInitParameter("AAA"));    Enumeration<String> enums = filterConfig.getInitParameterNames();  //遍历所有参数  **while**(enums.hasMoreElements()){  String paramName = enums.nextElement();  String paramValue = filterConfig.getInitParameter(paramName);  System.*out*.println(paramName+"="+paramValue);  }    } |

# 实例

|  |
| --- |
| /\*\*  \* 第一个过滤器  \* **@author** APPle  \*/  **public** **class** FirstFilter **implements** Filter{    **public** FirstFilter(){  System.***out***.println("1)过滤器对象创建了!");  }    /\*\*  \* FilterConfig对象封装了所有当前过滤器配置的初始化参数  \*  \*/  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  System.***out***.println("2)过滤器初始化方法");    /\*\*  \* 获取初始化参数  \*/  System.***out***.println(filterConfig.getInitParameter("AAA"));  //遍历所有参数  Enumeration<String> enums = filterConfig.getInitParameterNames();  **while**(enums.hasMoreElements()){  String paramName = enums.nextElement();  String paramValue = filterConfig.getInitParameter(paramName);  System.***out***.println(paramName+"="+paramValue);  }    }  /\*\*  \* 执行过滤任务  \*/  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.***out***.println("3)过滤器正在执行过滤任务-过滤请求");    //放行  chain.doFilter(request, response);    System.***out***.println("5)过滤器正在执行过滤任务--过滤响应");  }  /\*\*  \* 过滤器对象销毁的时候才会调用  \* web项目重新部署或者tomcat服务器停止了才会销毁过滤器对象  \*/  **public** **void** destroy() {  System.***out***.println("过滤器被销毁了");  }  } |

|  |
| --- |
| /\*\*  \* 目标资源(用户向访问这个资源)  \* **@author** APPle  \*  \*/  **public** **class** TargetServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.***out***.println("4)执行了servlet");      response.getWriter().write("6)用户看到响应内容");  }  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  doGet(request, response);  }  } |

# 过滤器链

[**doFilter**](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/Filter.html#doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse, javax.servlet.FilterChain))([ServletRequest](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/ServletRequest.html) request, [ServletResponse](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/ServletResponse.html) response, [FilterChain](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/FilterChain.html) chain)

执行过滤任务

参数一： ServletRequest是HttpServletRequest的父接口。

实际上传入的是HttpServletRequest接口的实现类。

参数二： ServletResponse是HttpServletResponse的父接口。

实际上传入HttpServletResponse接口的实现类。

参数三： FilterChain 过滤器链接口

[doFilter](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/FilterChain.html#doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse))([ServletRequest](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/ServletRequest.html) request, [ServletResponse](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/ServletResponse.html) response)

执行过滤器链中的下一个过滤器，如果没有下一个过滤器则执行目标资源。

**过滤器链： 一个目标资源可以被多个过滤器过滤，那么形成一个过滤器链。**

**注意：过滤器链中的过滤器执行顺序问题：由web.xml中filter-mapping的配置决定顺序。先配置的优先被执行。**

# 装饰者模式(Decorator)

23种java设计模式。单例模式，工厂模式，适配器模式，观察者模式，代理模式。。。。。。

装饰者模式：当开发者觉得某些类的某些方法不满足需要，向增强这些类的方法。这是就可以使用装饰者模式去装饰这些类。不满足需求的这些类，叫被装饰类。开发者需要重新编写装饰类去覆盖被装饰类。

BufferedReader：被装饰类。readLine方法不满足需要。

MyBufferReader: 装饰类。

装饰步骤：

1）继承被装饰类。BufferedReader。非final

2）装饰类中声明一个成员变量（被装饰类类型）

3）在构造方法，把被装饰类的实例接收到，赋值给成员变量

4）重写被装饰的方法。

## 实例

|  |
| --- |
| /\*\*  \* 装饰者模式  \* **@author** APPle  \*  \*/  **public** **class** Demo1 {  /\*\*  \* 需求：使用BufferedReader读取一个文件内容  \* **@throws** Exception  \*/  **public** **static** **void** main(String[] args) **throws** Exception {  //oldMethod();    BufferedReader b = **new** BufferedReader(**new** FileReader("e:/news.txt"));  //对BufferedReader进行装饰  BufferedReader br = **new** MyBufferedReader(b);  String str = **null**;    **while**( (str = br.readLine())!=**null**){  System.***out***.println(str);  }  }  **private** **static** **void** oldMethod() **throws** FileNotFoundException, IOException {  BufferedReader br = **new** BufferedReader(**new** FileReader("e:/news.txt"));  String str = **null**;    **while**( (str = br.readLine())!=**null**){  System.***out***.println(str);  }  }    /\*\*  \* 改造： 在每行名字前面加上一个序号。  \* 解决方案： 希望readLine方法返回给我们的就是一个带序号的内容。对BufferedReader的readLine方法进行增强。这是就可以用到装饰者模式。  \* 装饰者模式的开发步骤：  \* 概念： 装饰者和被装饰者 BufferedReader就是被装饰者  \* 1)编写一个BufferedReader装饰者类，继承被装饰者类。(不能是final的)  \* 2）在装饰类中定义一个成员变量，用于接收被装饰者类的对象。  \* 3）在装饰者类的构造方法中传入被装饰者类，使用第二步定义的变量接收被转入的 被装饰者类。  \* 4）在装饰类类中重写被装饰者类方法，对其方法进行增强。  \*/  }  /\*\*  \* 1)编写一个BufferedReader装饰者类，继承被装饰者类。(不能是final的)  \* BufferedReader(是非final的)  \*/  **class** MyBufferedReader **extends** BufferedReader{    /\*public MyBufferedReader(){  super();  }\*/    /\*\*  \* 2）在装饰类中定义一个成员变量，用于接收被装饰者类的对象。  \*/  **private** BufferedReader br;    /\*\*  \* 3）在装饰者类的构造方法中传入被装饰者类，使用第二步定义的变量接收被转入的 被装饰者类。  \* **@param** in  \*/  **public** MyBufferedReader(Reader in) {  **super**(in);  **this**.br = (BufferedReader)in;  }  **int** count = 1;  /\*\*  \* 4）在装饰类类中重写被装饰者类方法，对其方法进行增强。  \*/  @Override  **public** String readLine() **throws** IOException {  //得到原来的真实的内容  String content = br.readLine();  //加强: 加上序号  **if**(content!=**null**){  content = count+":"+content;  count++;  }  **return** content;  }    } |

# 案例

## 案例1 - 解决中文乱码问题

使用过滤器完美解决GET和Post方式提交中文乱码问题

|  |
| --- |
| /\*\*  \* 参数中文乱码问题的过滤器  \* **@author** APPle  \*  \*/  **public** **class** EncodingFilter **implements** Filter {  **public** **void** destroy() {  }  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }    /\*\*  \* 执行过过滤任务  \*/  **public** **void** doFilter(ServletRequest req, ServletResponse resp,  FilterChain chain) **throws** IOException, ServletException {  //强制转换  HttpServletRequest request = (HttpServletRequest)req;    /\*\*  \* post提交参数  \*/  request.setCharacterEncoding("utf-8");  //创建一个HttpServletRequest实现类的装饰者类,重写getParameter方法  MyHttpRequest myRequest = **new** MyHttpRequest(request);    /\*\*  \* 放行  \* 注意：  \* 这里放行的应该是装饰后的request对象，这样在业务的servlet调用getParameter才是新的重写后的方法。  \*/  chain.doFilter(myRequest , resp );  }  }  /\*\*  \* 1）装饰者类  \* **@author** APPle  \*  \*/  **class** MyHttpRequest **extends** HttpServletRequestWrapper{  /\*\*  \* 2)声明一个被装饰者类型的成员变量  \* **@param** request  \*/  **private** HttpServletRequest request;    /\*\*  \* 3)接收被装饰者类对象  \* **@param** request  \*/  **public** MyHttpRequest(HttpServletRequest request) {  **super**(request);  **this**.request = request;  }    /\*\*  \* 重写getParameter方法  \*/  @Override  **public** String getParameter(String name) {  **try** {  /\*\*  \* 对get提交参数处理  \*/  //1)得到原来的参数  String value = request.getParameter(name); //iso-8859-1    //2）手动解码  **if**("GET".equals(request.getMethod())){  value = **new** String(value.getBytes("iso-8859-1"),"utf-8");  }  **return** value;  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    } |

## 案例2 - 解决网页内容压缩问题

为什么要进行网页内容压缩？

访问web服务器时，服务器会返回网页内容（数据）

用户访问一个页面：

100KB

100万个用户访问这个页面

1 ,000,000 \* 100Kb = 100,000,000 = 服务器消耗了100G内容 （带宽）

用户访问一个页面：

100B

100万个用户访问这个页面

1 ,000,000 \* 100B = 100,000,000 = 100M服务器消耗了100M内容 （带宽）

买服务器：

网络服务器运营商

按流量收费： 我们尽可能压缩网页内容，才输出给浏览器

**怎么对网页内容压缩？**

**使用java 的GZIPOutputStream类进行gzip压缩。**

|  |
| --- |
| /\*\*  \* 对网页内容进行压缩  \*/  //创建临时的字节数组容器  ByteArrayOutputStream byteArr = **new** ByteArrayOutputStream();  //创建GZIPOutputStream对象  GZIPOutputStream gzip = **new** GZIPOutputStream(byteArr);  //开始写出压缩内容  gzip.write(sb.toString().getBytes());  //刷新缓冲区  gzip.finish();    //从临时的字节数组容器中得到压缩后的网页内容  **byte**[] result = byteArr.toByteArray();    System.*out*.println("压缩后的数据大小："+result.length);    /\*\*  \* 注意：告诉浏览器数据压缩格式 发送响应头：content-encoding:gzip  \*/  response.setHeader("content-encoding", "gzip");    //把压缩后的内容输出到浏览器  response.getOutputStream().write(result); |

### 对网页内容进行gzip压缩

|  |
| --- |
| /\*\*  \* 集中对网页内容进行gzip压缩  \* **@author** APPle  \*  \*/  **public** **class** GZIPFilter **implements** Filter {  **public** **void** destroy() {  }  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }    **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  //1)过滤请求    //创建一个response的装饰者对象  MyHttpResponse myResponse = **new** MyHttpResponse((HttpServletResponse)response);  /\*\*  \* 放行  \*/  chain.doFilter(request, myResponse);    //3）过滤响应  //从缓存容器对象得到压缩前的内容  //注意：response对象中没有方法获取实体内容，怎么办？  **char**[] content = myResponse.getCharArray();  //gzip压缩  ByteArrayOutputStream buf = **new** ByteArrayOutputStream();  GZIPOutputStream gzip = **new** GZIPOutputStream(buf);  gzip.write(**new** String(content).getBytes());  gzip.finish();  **byte**[] result = buf.toByteArray();    //告诉浏览器发送内容的压缩格式  myResponse.setHeader("content-encoding", "gzip");    //输出  response.getOutputStream().write(result);  //myRresponse.getWriter().write(new String(result,0,result.length));  }  }  /\*\*  \* HttpServletResponse的装饰者类  \*/  **class** MyHttpResponse **extends** HttpServletResponseWrapper{  **private** HttpServletResponse response;    //定义一个缓存容器对象  **private** CharArrayWriter charArray = **new** CharArrayWriter();    /\*\*  \* 提供一个获取charArray内容的方法(包含网页内容)  \* **@param** response  \*/  **public** **char**[] getCharArray(){  **return** charArray.toCharArray();  }    **public** MyHttpResponse(HttpServletResponse response) {  **super**(response);  **this**.response = response;  }    /\*\*  \* 重写getWriter()方法，让其返回一个带缓存功能的PrintWriter  \*/  @Override  **public** PrintWriter getWriter() **throws** IOException {  /\*\*  \* 现在已经创建了一个带CharArrayWriter缓存容器的PrintWriter了，  \* 如果我们调用带缓存PrintWriter对象的write()方法，那么内容会直接写入到CharrArrayWriter缓存容器中。  \*/  **return** **new** PrintWriter(charArray);  }        } |

## 案例3 - 登录权限过滤

|  |
| --- |
| /\*\*  \* 登录权限过滤器  \* **@author** APPle  \*  \*/  **public** **class** SecurityFilter **implements** Filter {  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  // **TODO** Auto-generated method stub  }  **public** **void** doFilter(ServletRequest req, ServletResponse resp,  FilterChain chain) **throws** IOException, ServletException {  //强制转换  HttpServletRequest request = (HttpServletRequest)req;  HttpServletResponse response = (HttpServletResponse)resp;  //是否登录判断逻辑  //先判断有无session对象存在  HttpSession session = request.getSession(**false**);  **if**(session==**null**){  //没有登录  response.sendRedirect(request.getContextPath()+"/noAuth.html");  **return**;  }**else**{  String user = (String)session.getAttribute("user");  **if**(user==**null**){  //没有登录成工  response.sendRedirect(request.getContextPath()+"/noAuth.html");  **return**;  }  }    //如果已经登录成功了，则放行！  chain.doFilter(request, response);  }  **public** **void** destroy() {  // **TODO** Auto-generated method stub  }  } |

# 总结

## 过滤器：Filter

### 1．作用：

在请求资源（静态资源或动态资源），或者响应资源，或者请求和响应资源时，执行过滤任务。

### 2．Filter的生命周期：

init方法： 初始化方法。参数FilterConfig对象，用于读取Filter配置的init-param参数。

doFilter方法： 执行过滤任务。每次访问目标资源时被执行。

参数一: ServletRequest，是HttpServletRequest接口的父类，可以强制转换为HttpServletREquest

参数二： ServletResponse，是HttpServletResponse的父类，可以强制转换为HttpServletResponse

参数三： FilterChain，是过滤器链对象。当一个资源同时被多个过滤器所过滤，那么就形成一个过滤器链。调用FilterChain的doFilter(request,response)方法，执行过滤器链的下一个过滤器，如果没有过滤器则调用目标资源。

destory方法： 销毁Filter对象的时候调用。

### 3．装饰者模式：

当需要对某些类的方法进行增强，那么可以对该类进行装饰。

3.1 编写一个装饰者类，继承被装饰者类（被装饰者类是非final的）

3.2 声明一个成员变量，用于接收被装饰者类对象

3.3 在装饰者类的构造方法中，接收被装饰者类对象。

3.4 覆盖需要增强的方法。

### 4．解决参数中文乱码问题：

核心： MyHttpRequest装饰者类，对HttpServletRequest的getParameter方法进行增强。

getParameterValues("name") / getParameterMap();

### 5．网页内容压缩问题：

核心： MyHttpResponse装饰者类，对HttpServletResponse的getWriter方法进行增强。

让getWriter（）方法返回带缓存功能的PrintWriter对象，这样PrintWriter的write方法写入的内容就直接写入到PinrtWriter中的缓存对象中。然后可以通过缓存对象取出内容。

ByteArrayOutputStream buf = new ByteArrayOutputStream()

GZIPOutputStream gzip = new GZIPOutputStream(buf);

gzip.write(内容);

gzip.finish(); //内容写入到ByteArrayOutputStream中

byte[] result = buf.toByteArray(); //从ByteArrayOutputStream中得到压缩后的内容

### 6．登录权限的过滤

核心： SecurityFilter过滤器中，进行登录权限的判断，这个过滤器要过滤哪些需要权限才能 访问的资源。（注意： 不要过滤登录页面，登录的servlet）

if(session==null){

没有权限的处理

}else{

String user = (String)session.getAttribute("user");

if(user==null){

没有权限的处理

}

}

//登录成功后，应该要放行资源

chain.doFilter(request,response);

# 监听器入门

## 之前GUI事件编程

GUI编程模式：

AWT：重量级GUI。和操作系统耦合高。

SWING：轻量级GUI。和操作系统耦合低。

GUI事件编程的三要素：

事件源：JButton / Button 、 JFrame/Frame 组件

事件：KeyEvent 、 WindowEvent 、 MouseEvent

监听器：KeyListener 、WindowListener、MouseListener

## web开发事件编程

web开发事件编程三要素：

事件源： ServletContext 、ServletRequest、HttpSession

事件：

对象创建或销毁：ServletContextEvent、ServletRequestEvent、 HttpSessionEvent

操作属性： ServletContextAttributeEvent、ServletRequestAttributeEvent、 HttpSessionBindingEvent

事件监听器：

ServletContextListener 、 ServletRequestListener......

web开发监听器： 开发者在web开发的过程中，会频繁使用到三个对象。ServletContext、ServletRequest、HttpSession。在创建或销毁这些对象时，或者对这些对象进行属性操作时，触发一些事件，就可以使用web监听器去监听这些事件。

|  |  |  |
| --- | --- | --- |
| **web事件源** | **web事件对象** | **web事件监听器** |
| ServletContext对象 | ServletContextEvent  创建或销毁对象时触发 | ServletContextListener |
| ServletContext属性 | ServletContextAttributeEvent  属性操作（增加属性，修改属性，删除属性） | ServletContextAttributeListener |
| ServletRequest对象 | ServletRequestEvent  创建或销毁对象时触发 | ServletRequestListener |
| ServletRequest属性 | ServletRequestAttributeEvent  属性操作（增加属性，修改属性，删除属性） | ServletRequestAttributeListener |
| HttpSession对象 | HttpSessionEvent  创建或销毁对象时触发 | HttpSessionListener |
| HttpSession属性 | HttpSessionBindingEvent  属性操作（增加属性，修改属性，删除属性） | HttpSessionAttributeLisener |

# ServetContext对象

## 监听创建：ServletContextListener

ServletContextListener主要用于监听ServetContext对象的创建和销毁。

ServletContext对象：代表整个web应用。

创建：web应用加载

销毁：web应用重新加载或web服务停止。

步骤：

1）创建java类，实现ServletContextListener接口，实现其中的方法

2）监听器要交给tomcat服务器运行。

需要在web.xml文件中进行配置

|  |
| --- |
| <!-- 注册监听器 -->  <listener>  <!-- 监听器程序全名 -->  <listener-class>gz.itcast.a\_context.MyContextListener</listener-class>  </listener> |

## 监听属性：ServletContextAttributeListener

ServletContextAttributeListener用于监听ServletContext对象的属性操作（增加属性，修改属性，删除属性）

增加属性： setAttribute(name,Object); 第一次就是增加属性

修改属性： setAttribute(name,Object); 如果前面有增加了同名的属性，则修改。

删除属性： removeAttribute(name);

|  |
| --- |
| /\*\*  \* ServletContext的监听器  \* **@author** APPle  \*需求  \* 1）在项目启动的时候，初始化表  \* 2）在项目结束的时候，删除表  \*/  **public** **class** MyContextListener **implements** ServletContextListener,ServletContextAttributeListener{  SystemDao dao = **new** SystemDao();  /\*\*  \* 该方法用于监听ServletContext对象的创建行为  \*/  **public** **void** contextInitialized(ServletContextEvent sce) {  //System.out.println("context对象创建了");    /\*\*  \* 初始表  \*/  dao.initTable();  System.***out***.println("初始化成功");  }  /\*\*  \* 该方法用于监听ServletContext对象的销毁行为  \*/  **public** **void** contextDestroyed(ServletContextEvent sce) {  //System.out.println("context对象被销毁了");    /\*\*  \* 清除表  \*/  dao.clearTable();  System.***out***.println("清除成功");  }      /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*属性相关的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/      /\*\*  \* 属性增加  \*/  **public** **void** attributeAdded(ServletContextAttributeEvent scab) {  //System.out.println("属性增加");  //得到属性名  String name = scab.getName();  //得到属性值  Object value = scab.getValue();  System.***out***.println("属性增加： "+name+"="+value);  }  /\*\*  \* 属性修改  \* **@param** scab  \*/  **public** **void** attributeReplaced(ServletContextAttributeEvent scab) {//ServletContextAttributeEvent已经包含了事件源对象  //System.out.println("属性修改");  String name = scab.getName();  //得到修改前的属性值  //Object value = scab.getValue();    //得到修改后的属性值  //需要从ServletContext事件源对象再次获取属性，才可以得到最新的属性值。  ServletContext context = scab.getServletContext();  Object value = context.getAttribute(name);    System.***out***.println("属性修改： "+name+"="+value);  }  /\*\*  \* 属性删除  \* **@param** scab  \*/  **public** **void** attributeRemoved(ServletContextAttributeEvent scab) {  //System.out.println("属性删除");    String name = scab.getName();  //得到属性值  Object value = scab.getValue();  System.***out***.println("属性删除： "+name+"="+value);  }  } |

# ServletRequest对象

## 监听创建：ServletRequestListener

ServletRequestListener用于监听ServletRequest对象的创建和销毁。

ServletRequest对象：封装请求信息

创建：每次发出请求时

销毁：请求完毕后

## 监听属性：ServletRequestAttributeListener

ServletRequestAttributeListener用于监听ServletRequest属性操作

增加属性： setAttribute(name,Object); 第一次就是增加属性

修改属性： setAttribute(name,Object); 如果前面有增加了同名的属性，则修改。

删除属性： removeAttribute(name);

|  |
| --- |
| /\*\*  \* ServetRequest的监听器  \* **@author** APPle  \*  \*/  **public** **class** MyRequestListener **implements** ServletRequestListener,ServletRequestAttributeListener{  /\*\*  \* 用于监听request对象的创建  \*/  **public** **void** requestInitialized(ServletRequestEvent sre) {  System.***out***.println("一个请求对象被创建了");    //得到请求对象相关的信息  HttpServletRequest request = (HttpServletRequest)sre.getServletRequest();  //得到客户的ip地址  String ip = request.getRemoteHost();  //共享数据到页面  //HttpSession session = request.getSession(true);  //session.setAttribute("ip", ip);  }  /\*\*  \* 用于监听request对象的销毁  \*/  **public** **void** requestDestroyed(ServletRequestEvent sre) {  System.***out***.println("一个请求对象被销毁了");  }    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*属性相关\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public** **void** attributeAdded(ServletRequestAttributeEvent srae) {  String name = srae.getName();  Object value = srae.getValue();  System.***out***.println("属性增加："+name+"="+value);  }  **public** **void** attributeReplaced(ServletRequestAttributeEvent srae) {  String name = srae.getName();  ServletRequest request = srae.getServletRequest();  Object value = request.getAttribute(name);  System.***out***.println("属性修改："+name+"="+value);  }  **public** **void** attributeRemoved(ServletRequestAttributeEvent srae) {  String name = srae.getName();  Object value = srae.getValue();  System.***out***.println("属性删除："+name+"="+value);  }  } |

# HttpSession对象

## 监听创建：HttpSessionListener

HttpSessionListener用于监听HttpSession对象的创建和销毁

HttpSession对象:

创建：调用request.getSession()方法

销毁：

1）默认情况下，30分钟服务器自动回收

2）设置有效时长: setMaxActiveInterval(秒);

3）web.xml配置全局的过期时间

<session-config>

<session-timeout>分钟</session-timeout>

</session-config>

4）手动销毁： invalidate()方法

案例： 粗略统计在线访客人数

|  |
| --- |
| /\*\*  \* HttpSession对象的监听器  \* **@author** APPle  \*  \*/  **public** **class** MySessionListener **implements** HttpSessionListener,HttpSessionAttributeListener{  //用于存储当前网站的访客人数  **private** **int** count = 0;  /\*\*  \* 用于监听HttpSession对象的创建  \*/  /\*\*  \* 每次创建一个session对象就代表一个在线访问进入网站  \*/  **public** **void** sessionCreated(HttpSessionEvent se) {  System.***out***.println("一个session对象被创建了"+se.getSession());    ServletContext context = se.getSession().getServletContext();  /\*\*  \* 使用代码同步避免因为多个用户同时访问引发的并发问题（导致count的数量不对）  \*/  **synchronized** (MySessionListener.**class**) { // 注意：锁对应必须唯一的，类对象就是唯一的  count++;  //把count通过context域对象共享到jsp页面  //可以通过session对象获取到ServletContext对象的  context.setAttribute("onLine", count );  }  }  /\*\*  \* 用于监听HttpSession对象的销毁  \*/  /\*\*  \* 当用户对应的session对象销毁了，代表访问离线了  \*/  **public** **void** sessionDestroyed(HttpSessionEvent se) {  System.***out***.println("一个session对象被销毁了"+se.getSession());    ServletContext context = se.getSession().getServletContext();    **synchronized** (MySessionListener.**class**){  count--;  context.setAttribute("onLine", count);  }  }  **public** **void** attributeAdded(HttpSessionBindingEvent se) {  String name = se.getName();  Object value = se.getValue();  System.***out***.println("属性增加： "+name+"="+value);  }  **public** **void** attributeReplaced(HttpSessionBindingEvent se) {  String name = se.getName();  HttpSession session = se.getSession();  Object value = session.getAttribute(name);  System.***out***.println("属性修改： "+name+"="+value);  }  **public** **void** attributeRemoved(HttpSessionBindingEvent se) {  String name = se.getName();  Object value = se.getValue();  System.***out***.println("属性删除： "+name+"="+value);  }  } |

## 监听属性：HttpSessionAttributeLisener

HttpSessionAttributeLisener用于监听HttpSession的属性操作

增加属性： setAttribute(name,Object); 第一次就是增加属性

修改属性： setAttribute(name,Object); 如果前面有增加了同名的属性，则修改。

删除属性： removeAttribute(name);

案例：显示当前网站的登录用户信息

需求:

1）提供用户登录的功能，提供注销功能。

2）显示当前网站的所有登录用户信息

3）管理员可以踢除指定的登录用户。

# 软件国际化

软件国际化，合适不同的国家或地区的人使用。

软件的文字：中文 、英文 、 韩文、 日文

软件的日期：

中国： 2015年1月16日 xx时xx分xx秒

美国： Jan 16 2015

英国： 16 Jan 2015

## 文字国际化

ResourceBundle类：

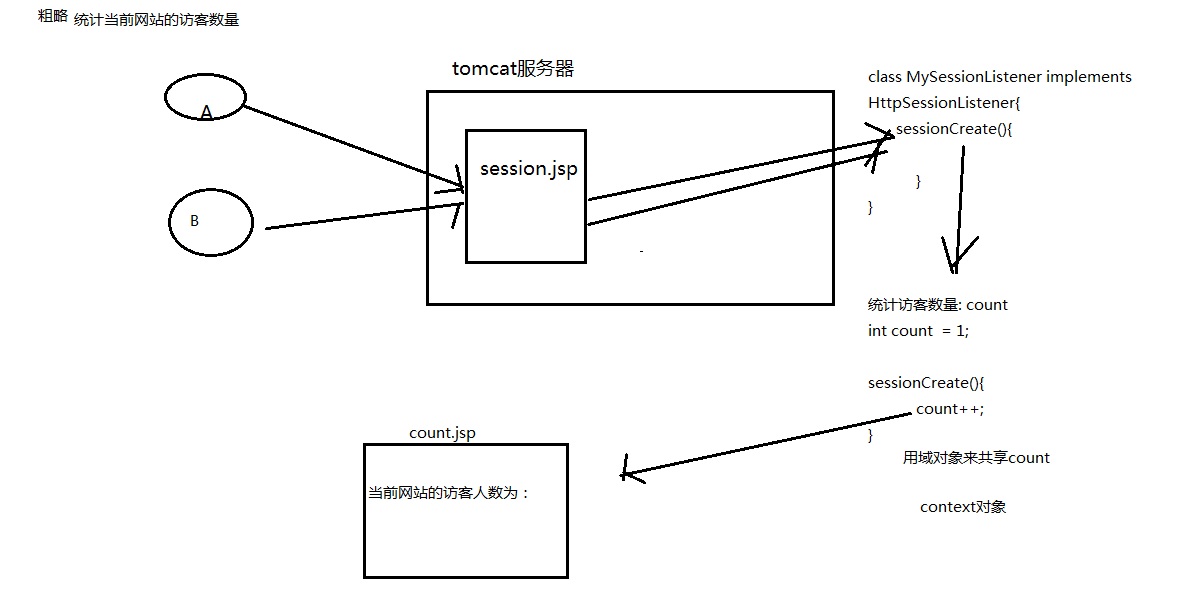
|  |
| --- |
| /\*\*  \* 文字国际化  \* **@author** APPle  \*  \*/  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  //创建ResourceBundle  /\*\*  \* 参数一： 资源包的路径: 资源包名称即可！  \* 参数二： 国家或地区的语言环境（代码）  \* 查询顺序： 指定国家 -> 系统默认国家 -> 默认文件  \*/  ResourceBundle rb = ResourceBundle.getBundle("gz/itcast/util/message", Locale.US);    //读取资源包信息  String username = rb.getString("username");  String password = rb.getString("password");  String login = rb.getString("login");  System.out.println(username+"\t"+password+"\t"+login);  }  } |

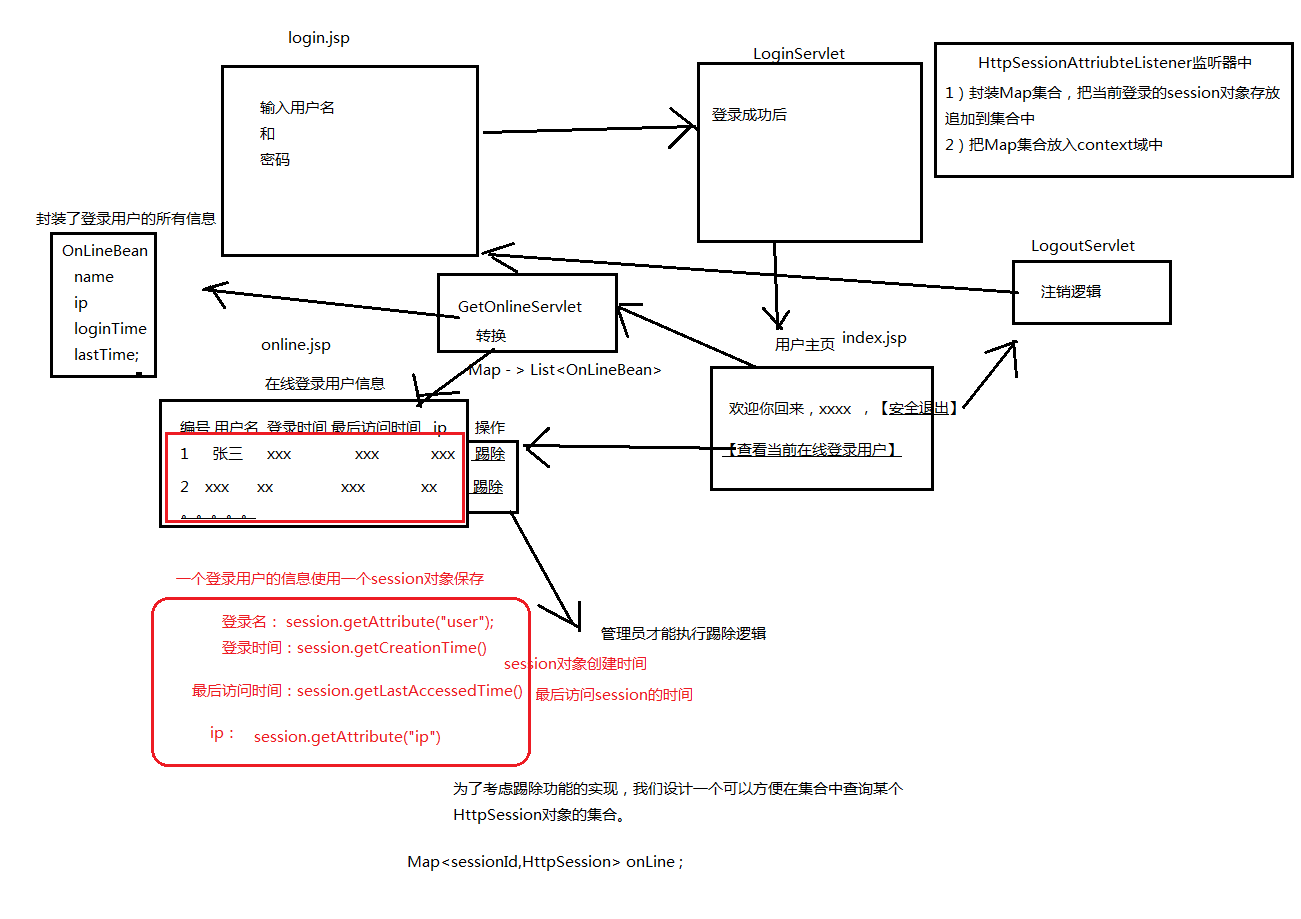
## 日期国际化

DateFormat类：

|  |
| --- |
| /\*\*  \* 日期国际化  \* **@author** APPle  \*  \*/  **public** **class** Demo2 {  /\*\*  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  /\*\*  \* 参数一： 日期样式(short->medium->long->full)  \* 参数二： 时间样式(short->medium->long->full)  \* 参数三: 国家或地区语言环境  \*/  DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL, Locale.***UK***);    String curDate = df.format(**new** Date());  System.***out***.println(curDate);  }  } |

# 案例 -- 用户信息管理





[D:\教程\Java案例\过滤器监听器 - 用户管理](file:///D:\教程\Java案例\过滤器监听器%20-%20用户管理)

# 总结

## 监听器： Listener

ServletContext对象：

ServletContextListener： 创建和销毁

ServletContextAttributeListener：属性增加，修改，删除

ServletRequest对象：

ServletRequestListener：创建和销毁

ServletRequestAttributeListener： 属性增加，修改，删除

HttpSession对象：

HttpSessionListener：创建和销毁

HttpSessionAttributeListener： 属性增加，修改，删除

### 注意：

1．实现特定的接口

2．监听必须交给tomcat服务器运行。在web.xml注册监听器

## 国际化： <fmt/> 标签

### 文字的国际化：

javase： ResourceBundle类，通过加载不同国家的资源包实现国际化

javaee: <fmt:setBundle/> <fmt:message/>

### 日期时间国际化：

javase： DateFormat类，通过设置DateForamt的日期和时间类型实现国际化

javaee: <fmt:formatDate/>