**Servlet程序开发**

Servlet(服务端小程序)是利用java语言编写的服务器端小程序。Servlet主要运行在服务器端，并由服务器调用执行，是一种按照Servlet标准开发的类。

Servlet程序是java对CGI程序的实现，但是与传统CGI程序不同的是Servlet采用了多线程处理方式，这样就使得Servlet程序的运行效率比传统程序高很多，而且Servlet还保留了可移植性的特点，使得Servlet的功能更加强大。

**Servlet处理步骤：**

1. **客户端通过HTTP提出请求。**
2. **Web服务器接收请求并将他发送到Servlet。**
3. **Servlet接收该HTTP请求并执行某种处理。**
4. **Servlet会将处理后的结果向微博服务器返回应答。**
5. **Web服务器将从Servlet接收到的应答发回给客户端。**

用户自定义的Servlet一般都不会直接继承GenericServlt类，而是根据所使用的协议继承他的相应子类HttpServlet类。

在继承HttpServlet类后，必须至少要覆写他的doGet()方法。

范例：第一个servlet程序

package org.wp.servletdemo;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet{

public void doGet(HttpServletRequest req , HttpServletResponse resp) throws ServletException,IOException{

PrintWriter out = resp.getWriter();

out.println("<html>");

out.println("<head><title>WENPAN</title></head>");

out.println("<body>");

out.println("<h1>HELLO WORLD!</h1>");

out.println("</body>");

out.println("</html>");

out.close();

}

}

在servlet中虽然可以执行输出操作，但是输出操作太过麻烦，所以一般都由jsp来输出。

Servlet中的程序映射配置：

**<servlet>**

**<servlet-name>。。。。。</servlet-name> <!--与servlet-mapping中的名称相对应-->**

**<servlet-class>包.类名称</servlet-class>**

**</servlet>**

**<servlet-mapping> <!--映射路径-->**

**<servlet-name>。。。。。</servlet-name>**

**<url-pattern>/映射路径</url-pattern> <!--页面的映射路径-->**

**</servlet-mapping>**

一个servlet也可以配置无数个访问的映射路径。值需要增加多个<servlet-mapping></servlet-mapping>元素即可。

在用户在地址栏输入一个页面的地址访问servlet时实际上就会调用doGet()方法进行处理，输入地址的方法就属于一种get请求。最常用的方法就是doGet和doPost两种方法。

**在实际的开发中经常出现找不到servlet而报404错误。这种情况下一定要耐心观察每一步的提交路径是否正确。**

**Servlet的生命周期：**

生命周期一般包括：**加载程序，初始化，服务，销毁，卸载**五个部分。

几个一般方法：

1. public void init() throws ServletException servlet初始化时调用
2. public void init(ServletConfig config) throws ServletException servlet初始化时调用,可以通过ServletConfig读取配置信息
3. Public abstract void service(...):servlet服务，一般不会直接覆写此方法，而是使用doGet,doPost方法。
4. Public void destroy() ：servlet销毁时使用。

范例：servlet生命周期

package org.wp.servletdemo;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class LifeCycleServlet extends HttpServlet{

public void init() throws ServletException{

System.out.println("\*\*1 Servlet初始化--->init()");

}

public void doGet(HttpServletRequest req,HttpServletResponse resp)throws

ServletException,IOException{

System.out.println("\*\*2 Servlet服务--->doGet(),doPost()");

}

public void doPost(HttpServletRequest req,HttpServletResponse resp) throws

ServletException,java.io.IOException{

this.doGet(req, resp);

}

public void destroy(){

System.out.println("\*\*3 Servlet销毁服务 --->destroy()");

// try {

// Thread.sleep(3000); //延时观看Servlet服务的销毁

// } catch (InterruptedException e) {

// e.printStackTrace();

// }

}

}

以上程序是在Tomcat后台打印出信息。在servlet类中只要是覆写了service方法，那么doGet和doPost方法就不再起作用，而是直接调用service处理。

初始化方法是在第一次调用的时候调用的，也可以配置启动选项，在容器启动时就自动初始化。

<servlet>

<servlet-name>life</servlet-name>

<servlet-class>包.类名称</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

**取得初始化配置信息：**

在servlet中可以通过初始化方法init(ServletConfig config)在容器加载的时候取得配置文件中的相关信息。

**范例：取得初始化配置信息**

**package** org.wp.servletdemo;

**import** javax.servlet.ServletConfig;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** Init\_paramServlet **extends** HttpServlet{

**private** String initParam = **null** ;

**public** **void** init(ServletConfig config){

**this**.initParam = config.getInitParameter("ref") ; //需要对web.xml文件进行配置

}

**public** **void** doGet(HttpServletRequest req,HttpServletResponse resp)**throws**

java.io.IOException ,ServletException{

System.***out***.println("初始化参数：" + **this**.initParam);

}

**public** **void** doPost(HttpServletRequest req,HttpServletResponse resp)**throws**

java.io.IOException ,ServletException{

**this**.doGet(req, resp) ;

}

}

配置文件的配置：

<servlet>

<servlet-name>initParam</servlet-name>

<servlet-class>

org.wp.servletdemo.Init\_paramServlet

</servlet-class>

<init-param> <!--设置初始化参数-->

<param-name>ref</param-name>

<param-value>ref属性的值： 文攀你好吗?</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>initParam</servlet-name>

<url-pattern>/InitParam</url-pattern>

</servlet-mapping>

**取得其他内置对象：**

1. 取得HttpSession实例:

方法：

public HttpSession getSession(); 返回当前的session

Public HttpSession getSession(boolean create); 返回当前session如果没有则创建一个新的session对象返回。

1. 取得ServletContext实例：

Public ServleContext() getServletContext(); 取得serveletContext对象

**Servlet跳转：**

1. **<jsp:forward>属于服务器端跳转，可将request范围属性保存到跳转页面。且执行到跳转语句后立即跳转。**
2. **Response.sendRedirect()属于客户端跳转，不可以将request属性范围保存到跳转后的页面，且要等到该页面执行完毕后再跳转。**
3. **在servlet中的服务器端跳转是使用request.getRequestDispatcher(“跳转页面”).forward(request,response);必须要实现RequestDispatcher接口。**

**范例：客户端跳转**

package org.wp.servletdemo;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class ClientRedirectDemo extends HttpServlet{ //客户端跳转

public void doGet(HttpServletRequest req,HttpServletResponse resp) throws

ServletException,java.io.IOException{

req.getSession().setAttribute("name", "文攀");

req.setAttribute("info", "文攀你好吗？");

resp.sendRedirect("get\_info.jsp"); //客户端跳转无法传递request属性范围内的值

}

public void doPost(HttpServletRequest req,HttpServletResponse resp)throws

ServletException,java.io.IOException{

this.doGet(req, resp);

}

}

**范例：服务器端跳转**

package org.wp.servletdemo;

import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class ServerRedirectDemo extends HttpServlet{ //实现服务器端的跳转

public void doGet(HttpServletRequest req,HttpServletResponse resp)throws

ServletException,java.io.IOException{

req.getSession().setAttribute("username", "文攀");

req.setAttribute("info", "文攀，你好吗？");

//实例化RequestDispatcher对象，同时指定跳转路径

**RequestDispatcher rd = req.getRequestDispatcher("get\_info.jsp");**

**rd.forward(req, resp);**  //实现服务器端的跳转

}

public void doPost(HttpServletRequest req,HttpServletResponse resp)throws

ServletException,java.io.IOException{

this.doGet(req, resp);

}

}

**WEB开发模式**

模式一：就是指在开发中将显示层，控制层，数据层的操作统一叫个jsp或者javaBean来进行处理。

模式一的两种情况：

1. 完全使用jsp开发：用户发出的请求交给jsp进行处理。
2. 使用jsp + JavaBean的模式进行开发:将显示层操作都叫个jsp处理，将数据层操作都交给Javabean处理。

Jsp + javaBean ：紧耦合，难维护，推诿，界面中的JavaScript代码较多。

Jsp + servelet:松耦合，易维护

**模型二：MVC设计模式**

**Servlet接收所有的客户端请求，然后根据请求调用相应的JavaBean，将所有的显示结果交给JSP完成。**

**DAO:只负责数据操作。**

**JSP:只负责显示。**

**Servlet:只负责接收参数，调用JavaBean，并且进行跳转功能。**

**用户的所有请求都会提交给servlet进行处理，然后由servlet调用Javabean，并将javabean的操作结果通过RequestDispatcher接口传递到JSP页面上。由于这些要显示的内容只在一次请求-回应中有效，所以所有的属性传递都将使用request属性范围进行传递。**

**范例：MVC设计模式的应用**

**package** org.wp.mvcdemo.vo;

**public** **class** User { //定义VO类

**private** String userid ; //对应userid列

**private** String name ; //对应name列

**private** String password ; //对应password列

**public** String getUserid() {

**return** userid;

}

**public** **void** setUserid(String userid) {

**this**.userid = userid;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

}

package org.wp.mvcdemo.dbc;

import java.sql.DriverManager;

import javax.print.attribute.standard.RequestingUserName;

import com.sun.corba.se.pept.transport.Connection;

public class DatabaseConnection {

private static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver" ;

private static final String DBURL = "jdbc:oracle:thin:@localhost:1521:mldn" ;

private static final String DBUSER = "scott" ;

private static final String DBPASSWORD = "tiger" ;

private java.sql.Connection conn = null ;

public DatabaseConnection()throws Exception{

try{

Class.forName(DBDRIVER) ;

this.conn = DriverManager.getConnection(DBURL,DBUSER,DBPASSWORD) ;

System.out.println("连接Oracle数据库成功！");

System.out.println("--------------------------------------------------");

}catch(Exception e){

throw e;

}

}

public java.sql.Connection getConnection(){

return this.conn ;

}

public void close()throws Exception{

if(this.conn != null){

try{

this.conn.close();

}catch(Exception e){

throw e;

}

}

}

}

**package** org.wp.mvcdemo.dao;

**public** **interface** IUserDAO {

/\*\*

\* 用户登录验证

\* **@param** user 传入VO对象

\* **@return** 验证的操作结果

\* **@throws** Exception

\* \*/

**public** **boolean** findLogin(org.wp.mvcdemo.vo.User user)**throws** Exception;

}

package org.wp.mvcdemo.dao.impl;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import org.apache.catalina.User ;

import org.wp.mvcdemo.dao.IUserDAO ;

public class UserDAOImpl implements IUserDAO{ //定义DAO具体实现类

private Connection conn = null ; //定义数据库连接对象

private PreparedStatement pstmt = null ; //定义数据库操作对象

public UserDAOImpl(Connection conn){ //设置数据库连接

this.conn = conn ;

}

@Override

public boolean findLogin(org.wp.mvcdemo.vo.User user) throws Exception {

boolean flag = false;

try{

String sql = "select name from myuser where userid = ? and password = ?" ;

this.pstmt = this.conn.prepareStatement(sql) ; //实例化操作

this.pstmt.setString(1, user.getUserid()) ; //设置ID

this.pstmt.setString(2, user.getPassword()) ; //设置password

ResultSet rs = this.pstmt.executeQuery() ; //取得查询结果

if(rs.next()){

user.setName(rs.getString(1)) ; //取得姓名

flag = true ; //登录成功

}

}catch(Exception e){

throw e ;

}finally{

if(this.pstmt != null){

try{

this.pstmt.close() ;

}catch(Exception e){

throw e ;

}

}

}

return flag ;

}

}

package org.wp.mvcdemo.dao.proxy;

import org.wp.mvcdemo.dao.IUserDAO;

import org.wp.mvcdemo.dao.impl.UserDAOImpl;

import org.wp.mvcdemo.dbc.DatabaseConnection;

import org.wp.mvcdemo.vo.User;

public class UserDAOProxy implements IUserDAO{

private DatabaseConnection dbc = null ; //定义数据库连接

private IUserDAO dao = null ; //定义DAO类接口

public UserDAOProxy(){

try{

this.dbc = new DatabaseConnection() ; //实例化数据库连接

}catch(Exception e){

e.printStackTrace() ;

}

this.dao = new UserDAOImpl(this.dbc.getConnection()) ; //实例化真实操作类

}

@Override

public boolean findLogin(User user) throws Exception {

boolean flag = false ;

try{

flag = this.dao.findLogin(user) ; //调用真实主题类

}catch(Exception e){

throw e ;

}finally{

this.dbc.close() ;

}

return flag ;

}

}

package org.wp.mvcdemo.factory;

import org.wp.mvcdemo.dao.IUserDAO;

import org.wp.mvcdemo.dao.proxy.UserDAOProxy;

public class DAOFactory { //建立工厂类取得DAO实例

public static IUserDAO getUserDAOInstance(){

return new UserDAOProxy() ; //返回代理实例

}

}

package org.wp.mvcdemo.servlet;

import java.util.ArrayList;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.apache.catalina.User;

import org.wp.mvcdemo.factory.DAOFactory;

import com.sun.xml.internal.bind.v2.schemagen.xmlschema.List;

public class LoginServlet extends HttpServlet{ //编写servlet类接收客户端发来的请求

public void doGet(HttpServletRequest req,HttpServletResponse resp) throws

ServletException,java.io.IOException{

String path = "login.jsp" ;

**String userid = req.getParameter("userid") ;**

**String userpass = req.getParameter("userpass") ;**  **// 1. 接收客户端请求**

ArrayList<String> info = new ArrayList<String>(); //保存所有的返回信息

if(userid == null || "".equals(userid)){

info.add("用户id不能为空！") ;

}

if(userpass == null || "".equals(userpass)){

info.add("用户密码不能为空！") ;

}

if(info.size() == 0){ //用户名和密码验证通过

org.wp.mvcdemo.vo.User user = new org.wp.mvcdemo.vo.User() ; //实例化VO

user.setUserid(userid);

user.setPassword(userpass);

try{

**if(DAOFactory.getUserDAOInstance().findLogin(user)){**  //验证通过 2. **调用JavaBean**

info.add("用户登录成功！欢迎：" + user.getName() + "光临！") ;

}else{

info.add("用户登录失败！错误的用户名和密码！") ;

}

}catch(Exception e){

e.printStackTrace();

}

}

**req.setAttribute("info", info) ;**  //保存错误信息 3. **将结果交给jsp进行显示**

req.getRequestDispatcher(path).forward(req, resp) ; //服务器端跳转

}

public void doPost(HttpServletRequest req,HttpServletResponse resp) throws

ServletException,java.io.IOException{

this.doGet(req, resp); //调用doGet()操作

}

}

**过滤器：**

从使用上看servlet可以分为三类：

1. **简单servlet、**
2. **过滤serve聊天**
3. **监听servlet**

过滤器是一种以组件形式绑定到web应用程序当中的，与其他web应用程序组件不同的是，过滤器采用的是“链”的方式进行处理的。

Eg: **客户端 ===>过滤器 ===> 过滤器 ===>过滤器 ===>web资源**

**在没有使用过滤器以前客户端都是直接请求web资源的，但是一旦加入了过滤器所有的请求都是先交给过滤器处理，然后再访问相应的web资源。可以达到对某些资源的访问限制。**

**如果要定义一个过滤器，则直接让一个类实现javax.servlet.Filter接口即可。**

Filter接口定义的方法：

1. public void init(FilterConfig filterConfig) throws ServletException 过滤器初始化（容器启动时初始化）时调用，可以通过FilterConfig取得初始化配置信息。
2. Public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain) throws ServletException,IOException 完成具体过滤操作，然后通过FilterChain让请求继续向下传递。
3. Public void destroy() ; 过滤器销毁时候调用

FilterChain的作用是将用户的请求继续向下传递给其他的过滤器或则Servlet。在FilterChain接口中依然定义了一个同样的doFilter()方法。这是因为在一个过滤器后面还存在一个过滤器，或则可能是请求的最终目标，这样就通过FilterChain形成了一个过滤连操作。

**范例：定义一个简单的过滤器**

package org.wp.filterdemo;

import java.io.IOException;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

public class SimpleFilter implements javax.servlet.Filter{ //实现了 javax.servlet.Filter接口就为过滤器

public void init(FilterConfig config) throws ServletException{ //初始化锅过滤器

String initParam = config.getInitParameter("ref") ; //取得初始化参数

System.out.println("\*\*过滤器初始化，初始化参数 = " + initParam); //在容器启动的时候自动加载的init方法

}

public void doFilter(ServletRequest request ,ServletResponse response,FilterChain chain) throws

IOException,ServletException{ //执行过滤

System.out.println("\*\*执行doFilter方法之前。") ;

chain.doFilter(request, response) ; //请求参数继续传递

System.out.println("\*\*执行doFilter方法之后。") ;

}

public void destroy(){

System.out.println("过滤器销毁！");

}

}

过滤器中的初始化方法是在容器启动时自动加载的，并且通过FilterConfig的getInitParameter()方法取得配置的初始化参数，只初始化一次。但是对于过滤器中的doFilter()方法会调用两次，一次是在FilterChain操作前，一次是在FilterChain方法之后。

<url-pattern>表示一个过滤器的位置，如果是/\*表示对于根目录下的一切操作都需要过滤。

**过滤器的应用：**

范例：编码过滤

package org.wp.filterdemo;

import java.io.IOException;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

public class EncodingFilter implements Filter{ //利用Servlet过滤器设置统一的字符编码

private String charSet ; //设置字符编码

public void init(FilterConfig config){

this.charSet = config.getInitParameter("charset") ; //取得初始化参数

}

public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)throws

IOException,ServletException{

request.setCharacterEncoding(this.charSet) ; //为所有页设置统一的请求编码

chain.doFilter(request, response) ;

}

public void destroy(){

}

}

范例：登录验证过滤

package org.wp.filterdemo;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpSession;

/\*\*

\* 不管是登录验证还是编码过滤都是自动完成的，init方法在容器加载时候就调用

\* \*/

public class LoginFilter implements Filter{ //登录验证过滤

public void init(FilterConfig config){ //初始化过滤器

}

public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain) throws

java.io.IOException ,ServletException {

HttpServletRequest req = (HttpServletRequest)request ; //强制性向下转型

HttpSession sess = req.getSession() ; //取得session

if(sess.getAttribute("userid") != null){ //判断是否登录

chain.doFilter(request, response) ; //传递请求

}else{

request.getRequestDispatcher("login.jsp").forward(request, response) ; //跳转到登录页面

}

}

public void destroy(){

}

}

**监听器（使用相对较少）：**

1. 对application的监听：对application的监听实际上就是对ServletContext(servlet上下文)监听，此接口定义方法如下所示：

Public void contextInitialized(ServletContextEvent sce) 容器启动时触发

Public void contextDestroyed(ServletContextEvent sce) 容器销毁时触发

在上下文监听中，一旦触发了ServletContextListener接口中定义的事件后，可以通过ServletContextEvent进行事件处理。

ServletContextEvetn类中只定义了 getServletContext()一个方法，用来取得ServletContext实例。

**范例：对servlet上下文状态监听**

package org.wp.listenerdemo;

import javax.servlet.ServletContextEvent;

import javax.servlet.ServletContextListener;

public class ServletContextListenerDemo implements ServletContextListener{

@Override

public void contextInitialized(ServletContextEvent event){ //上下文初始化时触发

System.out.println("\*\*容器初始化-->" + event.getServletContext().getContextPath());

}

@Override

public void contextDestroyed(ServletContextEvent event){

System.out.println("\*\*容器销毁-->" + event.getServletContext().getContextPath());

}

}

配置web.xml文件

<listener>

<listener-class>

包名.类名

</listener-class>

</listener>

所有的Servlet程序都必须在web.xml文件中进行配置，如果一个web.xml文件要同时配置简单servlet,过滤器，监听器的话，一般都先配置过滤器，然后配置监听器，最后配置简单servlet。

**上下文属性监听：ServletContextAttributeListener接口**

在此接口中有如下方法：

1. public void attributeAdded(ServletContextAttributeEvent event) 增加属性时触发
2. Public void attributeRemoved(ServletContextAttributeEvent event) 删除属性时触发
3. Public void attributeReplaced(ServletContextAttributeEvent event) 替换属性时触发

ServletContextAttributeEvent事件定义的方法：

1. public String getName() ; 取得设置的属性的名称
2. Public String getValue() ; 取得设置属性的值

**范例：对上下文属性的监听**

package org.wp.listenerdemo;

import javax.servlet.ServletContextAttributeEvent;

import javax.servlet.ServletContextAttributeListener;

import javax.servlet.ServletContextEvent;

public class ServletContextAttributeListenerDemo implements ServletContextAttributeListener{//上下文属性监听

@Override

public void attributeAdded(ServletContextAttributeEvent event){ //增加属性时触发

System.out.println("\*\*增加属性-->属性名称：" + event.getName() + "属性内容：" + event.getValue());

}

@Override

public void attributeRemoved(ServletContextAttributeEvent event){ //删除属性时触发

System.out.println("\*\*删除属性-->属性名称：" + event.getName() + "属性内容：" + event.getValue());

}

@Override

public void attributeReplaced(ServletContextAttributeEvent event){ //替换属性时触发

System.out.println("\*\*替换属性-->属性名称：" + event.getName() + "属性内容：" + event.getValue());

}

}

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

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>ServletContext的增加属性</title>

</head>

<body>

<%

**this**.getServletContext().setAttribute("info", "文攀你好吗？我很好") ;

%>

</body>

</html>

1. **对session的监听**

对session的监听操作主要使用 HttpSessionListener,HttpSessionAttributeListener,HttpSessionBindingListener接口。

1.session状态的监听HttpSessionListener接口：

在该接口中定义了如下方法：

1. public void sessionCreated(HttpSessionEvent event) ; session创建时调用
2. Public void sessionDestroyed(HttpSessionEvent event) ; session销毁时调用

在HttpSessionEvent接口中的方法：

Public HttpSession getSession(); 取得当前的session

范例：对session的监听

package org.wp.listenerdemo;

import javax.servlet.http.HttpSessionEvent;

import javax.servlet.http.HttpSessionListener;

public class HttpSessionListenerDemo implements HttpSessionListener{

@Override

public void sessionCreated(HttpSessionEvent event){

System.out.println("\*\*Session创建，SESSION ID = " + event.getSession().getId());

}

@Override

public void sessionDestroyed(HttpSessionEvent event){

System.out.println("\*\*Session销毁，SESSION ID = " + event.getSession().getId());

}

}

创建一个session的方法： 当新用户打开一个动态网页时，服务器自动为新用户分配session，并且触发HttpSession接口中的sessionCreated()事件，但是在用户销毁时却有两种不同的方式。

销毁一个session：

方式一：调用HttpSession接口中的invalidate()方法，让一个session失效。

方式二：超过了配置的session时间，默认为30分钟。可以更改。

Session属性监听HttpSessionAttributeListener接口：

该接口中的方法：

1. public void attributeAdded(HttpSessionBindingEvent event) ; 增加属性时触发
2. public void attributeRemoved(HttpSessionBindingEvent event) ; 删除属性时触发
3. public void attributeReplaced(HttpSessionBindingEvent event) ; 替换属性时触发

HttpSessionBindingEvent事件定义的方法：

1. public HttpSession getSession() ; 取得session
2. Public String getName() ; 取得属性的名称
3. Public Object getValue() ; 取得属性内容

**范例：对session属性的监听**

package org.wp.listenerdemo;

import java.beans.EventSetDescriptor;

import javax.servlet.http.HttpSessionAttributeListener;

import javax.servlet.http.HttpSessionBindingEvent;

public class HttpSessionAttributeListenerDemo implements HttpSessionAttributeListener {

@Override

public void attributeAdded(HttpSessionBindingEvent event){ //增加一个新的属性时调用

System.out.println( event.getSession().getId() + " \*\*增加新的session属性：属性名称-->" +

event.getName() + " 属性值-->" + event.getValue());

}

@Override

public void attributeRemoved(HttpSessionBindingEvent event){

System.out.println(event.getSession().getId() + " \*\*删除Session属性：属性名称-->" +

event.getName() + " 属性值-->" + event.getValue());

}

@Override

public void attributeReplaced(HttpSessionBindingEvent event){

System.out.println(event.getSession().getId() + " \*\*属性session替换：属性名称-->" +

event.getName() + " 属性值-->" + event.getValue());

}

}

Session属性的监听方式二（不用配置web.xml文件）：直接实现HttpSessionBindingListener接口

HttpSessionBindingListener接口中定义的方法：

1. public void valueBound(HttpSessionBindingEvent event) ; 绑定对象到session时触发
2. Public void valueUnbound(HttpSessionBindingEvent event) ; 从session中移除对象时触发

**范例：用户登录状态监听**

package org.wp.listenerdemo;

import javax.servlet.http.HttpSessionBindingEvent;

import javax.servlet.http.HttpSessionBindingListener;

public class LoginUser implements HttpSessionBindingListener { //不用配置web文件直接使用

private String name ;

public LoginUser(String name){ //构造方法 保存登录用户姓名

this.setName(name) ; //设置用户名

System.out.println("构造函数执行");

}

@Override

public void valueBound(HttpSessionBindingEvent event){ //在session中绑定

System.out.println("\*\*在session中保存LoginUser对象（name = " + this.getName() + "),session id = " +

event.getSession().getId());

System.out.println("保存的属性名称： " + event.getName() + " 保存的属性值： " + event.getValue());

}

@Override

public void valueUnbound(HttpSessionBindingEvent event){

System.out.println("\*\*从session中移除LoginUser对象（name = " + this.getName() + "),session id = " +

event.getSession().getId());

}

public void setName(String name){

this.name = name ;

}

public String getName(){

return this.name ;

}

}

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

<%@page import=*"org.wp.listenerdemo.LoginUser"*%>

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>利用HttpSessionBingdingListener接口实现不配置</title>

</head>

<body>

<%

LoginUser user = **new** LoginUser("文攀"); //实例化LoginUser对象

//out.println("这里先执行") ;

session.setAttribute("info", user) ; //在session中保存对象 触发监听事件自动调用valueBound()操作进行处理

%>

</body>

</html>

以上程序需要手工绑定，所以不需要配置。

对request的监听和对request属性的操作和对session的监听雷同。

**范例：监听器实现在线人员统计**

package org.wp.listenerdemo;

import java.awt.Event;

import java.util.Set;

import java.util.TreeSet;

import javax.servlet.ServletContext;

import javax.servlet.ServletContextEvent;

import javax.servlet.ServletContextListener;

import javax.servlet.http.HttpSessionAttributeListener;

import javax.servlet.http.HttpSessionBindingEvent;

import javax.servlet.http.HttpSessionEvent;

import javax.servlet.http.HttpSessionListener;

public class OnlineUserList implements HttpSessionListener,

HttpSessionAttributeListener,ServletContextListener {

private ServletContext app = null ;//用于application属性操作

@Override

public void contextInitialized(ServletContextEvent arg0) { //上下文初始化 容器启动时触发

this.app = arg0.getServletContext() ; //取得ServletContext实例

this.app.setAttribute("online",new TreeSet<>()); //设置空集合 在全局范围有效

}

@Override

public void attributeAdded(HttpSessionBindingEvent arg0) { //增加session属性

Set all = (Set)this.app.getAttribute("online") ; //取出已有列表

all.add(arg0.getValue()) ; //增加新用户

this.app.setAttribute("online",all) ; //重新加入集合

}

@Override

public void attributeRemoved(HttpSessionBindingEvent arg0) { //用户离开

Set all = (Set)this.app.getAttribute("online") ; //取出已有列表

all.remove(arg0.getValue()) ; //除去离开的用户

this.app.setAttribute("online", all) ; //从新加入集合

}

@Override

public void sessionDestroyed(HttpSessionEvent arg0) {

Set all = (Set)this.app.getAttribute("online") ;

all.remove(arg0.getSession().getAttribute("userid")) ; //取出设置的内容

this.app.setAttribute("online", all);

}

@Override

public void attributeReplaced(HttpSessionBindingEvent arg0) {}

@Override

public void sessionCreated(HttpSessionEvent arg0) {}

@Override

public void contextDestroyed(ServletContextEvent arg0) {}

}

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

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>登录过滤</title>

</head>

<body>

<form action = *"sucess.jsp"* method = *"post"*>

用户名：<input type = *"text"* name = *"userid"*/>

<input type = *"submit"* value = *"提交"*/>

<input type = *"reset"* value = *"重置"*/>

</form>

</body>

</html>

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

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ page import = *"java.util.\*"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>显示在线用户</title>

</head>

<body>

<h2>在线用户列表</h2>

<%

Set all = (Set)**this**.getServletContext().getAttribute("online") ;//从application中取出所有用户保存的列表

Iterator iter = all.iterator(); //实例化迭代器输出

**while**(iter.hasNext()){

%>

<h3><%=iter.next() %></h3>

<%

}

%>

</body>

</html>

**每进行登录一次就会自动触发attributeAdded事件。在此事件中在此事件中将全局的集合online取出，然后将刚才触发attributeAdded事件的登录用户名加入到集合然后将此集合重新保存到online全局集合。当一个登录用户的session失效以后就会自动触发attitudeRemoved()方法,将所有用户从集合中取出，然后将失效的用户名从集合中删除，最后将操作后的集合重新保存到全局集合online中。在list.jsp文件中进行在线用户的显示，在此文件用户集合全部取出，然后利用迭代器进行输出。**