# Struct2

## Eclipse安装

下载eclipse

<http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/helios/SR2/eclipse-jee-helios-SR2-win32.zip>

### 安装WTP

Eclipse 解压之后，需要下载 WTP(Web Tools Platform)才能进行 web 开发，因为之前下载的 Eclipse 是标准版的，不含开发 web 工程需要的插件，最简单的安装办法是在线安装 WTP 插件。

在线安装的办法是：

1. 启动 Eclipse，选择 Help->Install new software 。
2. 选择 Available Software sites，找到 <http://download.eclipse.org/webtools/updates>，如果没有则单击Add，输入以上网址。
3. 然后在地址栏里选择 <http://download.eclipse.org/webtools/updates>。
4. 勾选"Web Tools Platform (WTP) 3.1.2 "，然后进行下载安装即可。
5. 安装完毕之后会提示重启。

### 安装Mysql connector

安装MySql 成功后，为了使我们开发的web应用能够连接数据库，需要下载一个 mysql-connector。

下载完之后解压，将 mysql-connector-java-X.X.X-bin.jar拷贝到 tomcat 安装目录的 lib 文件夹下。

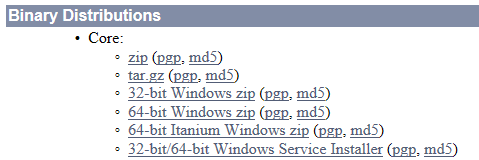
### 安装servlet-api

为了能够在工程中建立 servlet，需要将 servlet-api.jar(这个文件在 tomcat 安装目录的lib文件夹下)

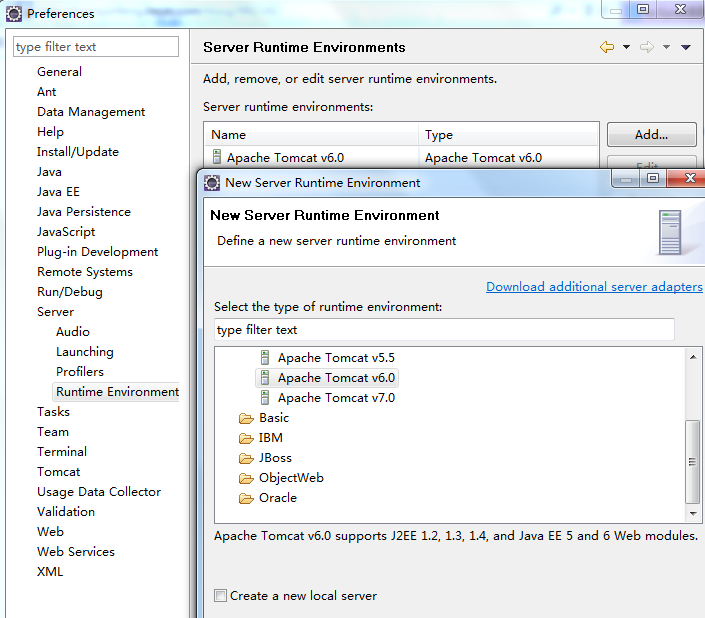
添加到 Eclipse 的引用库中，方法是 选择 Window->Preferences->Java->Installed JRE，选中 jre，进行"编辑",选择添加外部 JAR，将 servlet-api.jar 添加进去即可。

### 配置tomcat

下载zip，不建议下载windows zip，解压即可。



Window->Preferences->Server->Runtime Environments->Add->Apache->Apache Tomcat v6.0->选择tomcat的目录。



 打开“Servers”视图，通过Window->Show View->Server->选择Servers。

在“Servers”视图中，右键单击->New->Server->选择Tomcat v6.0 Server，如果在Server runtime environment中看到Apache Tomcat v6.0，则说明配置成功。

### 安装Tomcat Plugin

为了方便在 Eclipse 中启动和关闭 tomcat，可以下载一个 tomcat plugin 插件

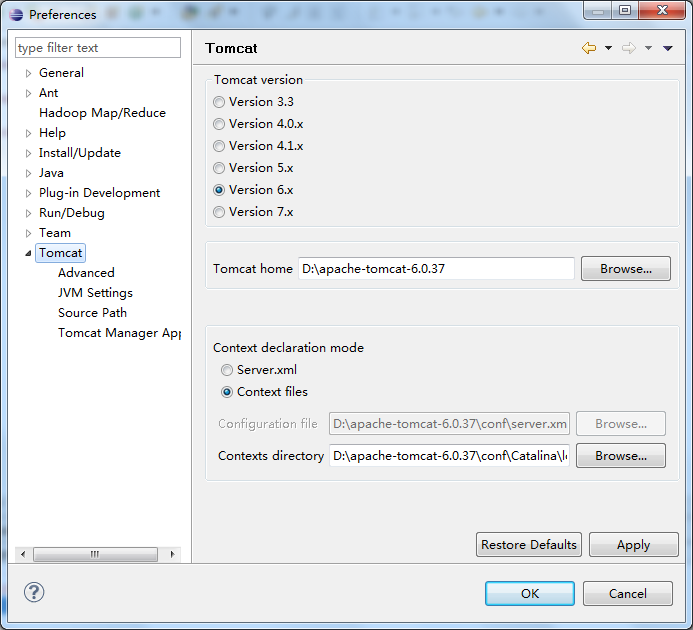
<http://www.eclipsetotale.com/tomcatPlugin.html>

注意插件与Eclipse和Tomcat版本是否对应。

解压后放到your\_Eclipse\_Home/plugins，重启后看到工具栏的三个小图标



Window—>Preferences—>Tomcat页面中，选择tomcat版本和安装目录



单击启动图标，如果出现以下信息，则配置成功。

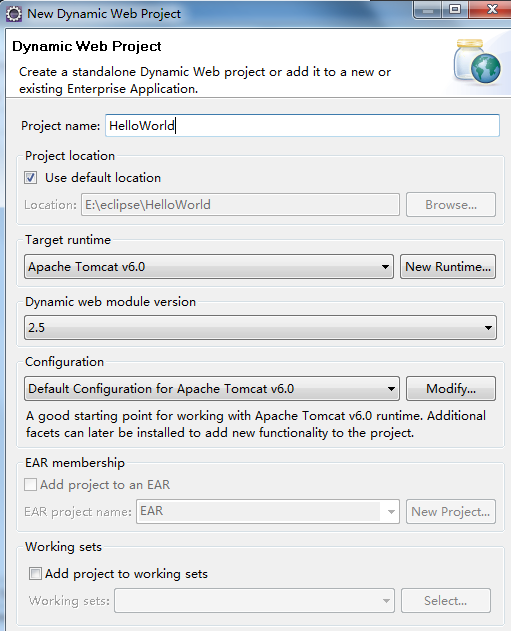
五月 15, 2013 3:51:54 下午 org.apache.catalina.startup.Catalina start

信息: Server startup in 478 ms

## 第一个Servlet项目

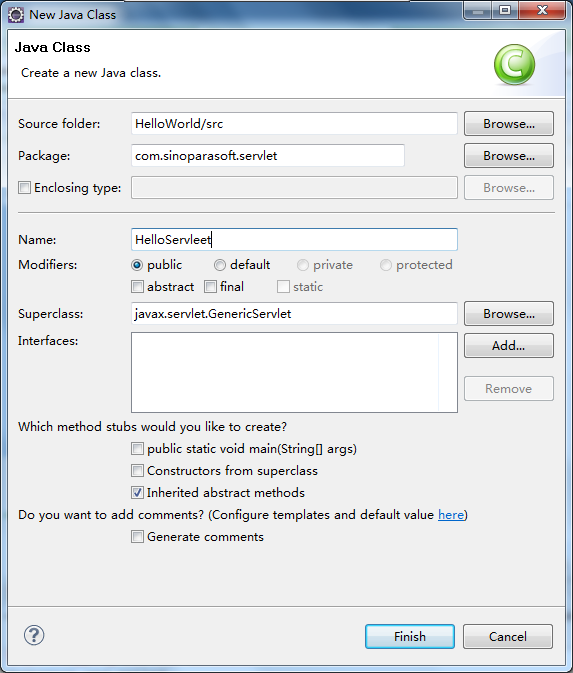
### 新建Web项目

File->New->Project->Web->Dynamic Web Project



### 新建Servlet类

Java Resources->src右键New->Class



public class HelloServlet extends GenericServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* Servlet启动时调用的函数

\*/

public void init() {

System.out.println("init() was called.");

}

@Override

public void service(ServletRequest req, ServletResponse res)

throws ServletException, IOException {

// TODO Auto-generated method stub

System.out.println("service() was called.");

// 指定输出内容以及编码方式

res.setContentType("text/html; charset=UTF-8");

// 获取输出流对象

PrintWriter out = res.getWriter();

// 发送HTML内容

out.println("<html>");

out.println("<head>");

out.println("<title>Hello, Servlet!</title>");

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

out.println("<body>");

out.println("Hello, Servlet!");

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

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

}

/\*\*

\* Servlet结束时调用的函数

\*/

public void destroy() {

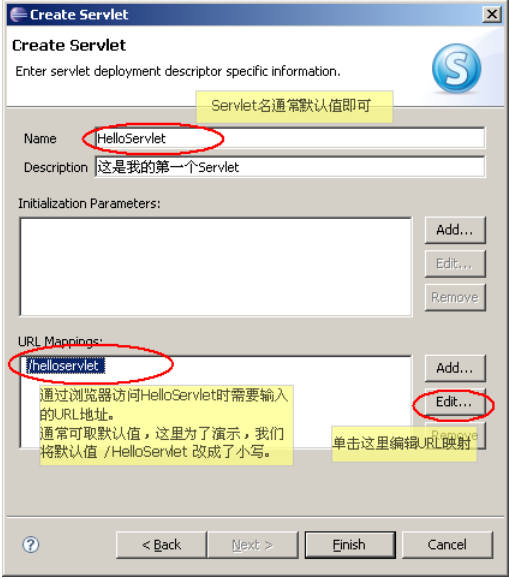
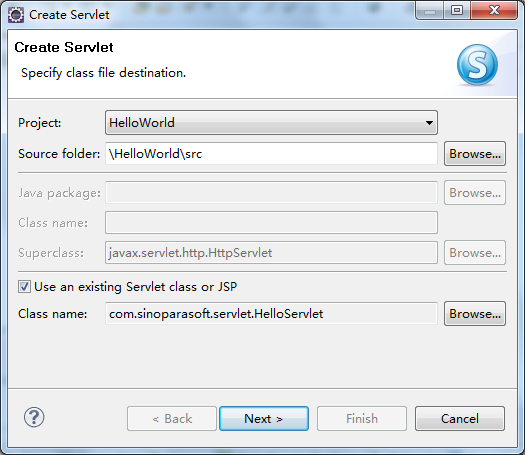
System.out.println("destroy() was called.");

}

}

### 关联URL和Servlet

Deployment Descriptor:HelloWorld右键New->Servlet，选Use an existing Servlet class or JSP，选择HelloServlet



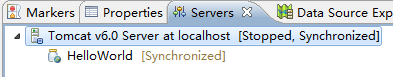
URL格式为<http://主机名/应用程序名/Servlet>的URL映射，即

<http://localhost:8080/HelloWorld/HelloServlet>

相应修改WebContent/WEB-INF/web.xml内容。

### 启动Tomcat

Servlet页面的Tomcat v6.0 Server at localhost右键Add and Remove，将HelloServlet选入右框。



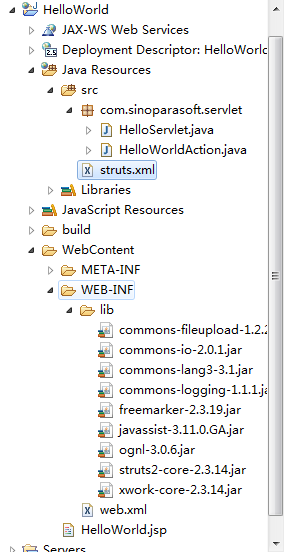
右键单击服务器，Start，在浏览器中输入<http://localhost:8080/HelloWorld/HelloServlet>

即可看到控制台Servlet输出。

## 第一个Struct2项目

### 添加Struct2类库

在struts2的lib目录中复制以下9个jar包，在项目/WebContent/WEB\_INF/lib目录上右键粘贴，包将自动加入项目路径中。



### 配置FilterDispatcher

Deployment Descriptor:HelloWorld右键New->Filter，在web.xml中定义核心Filter来拦截用户请求。

Struct2用StrutsPrepareAndExecuteFilter，不用FilterDispatcher。

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

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

</filter-mapping>

### 开发Action类

在Java Resources->src目录下新建一个Package，包名为com.sinoparasoft.struts，然后在包里新建一个class，文件名为HelloWorldAction.java

public class HelloWorldAction implements Action

{

private String message;

public String getMessage()

{

return message;

}

@Override

public String execute() throws Exception

{

message = "Hello World!";

return SUCCESS;

}

}

### 配置struts.xml

在Java Resources -->src目录下新建struts.xml，将Action与URL关联起来

package：name指明包名，extends指明要扩展的包。

action：name指明访问action的URL路径，class指明URL路径对应的Action类。

result：name指明Action的execute方法的返回值为多少时，将转到何处。

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

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"

"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

<package name="default" extends="struts-default">

<action name="HelloWorld" class="com.sinoparasoft.struts.HelloWorldAction">

<result name="success">/HelloWorld.jsp</result>

</action>

</package>

</struts>

### 编写JSP

在WebContent目录中新建HelloWorld.jsp

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<%@ taglib prefix="s" uri="/struts-tags" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Insert title here</title>

</head>

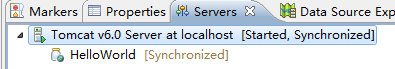
<body>

<h2><s:property value="message"/></h2>

</body>

</html>

### 启动Tomcat



右键启动，在浏览器中访问

<http://localhost:8080/HelloWorld/HelloWorld.action>

## 配置国际化

web.xml中增加以下内容：

<context-param>

<param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>

<param-value>res.message</param-value>

</context-param>

lib中增加jstl-x.x.jar

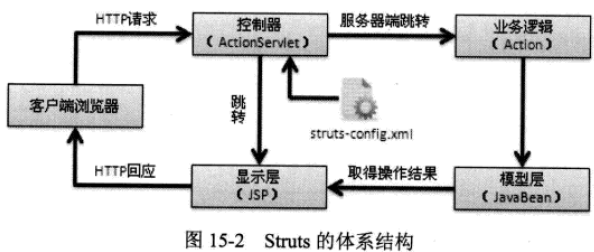
src下面增加message.properties，汉字转成Unicode码

jsp文件中

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

<fmt:message key="title.AddToFavorites"/>

## 基础知识



### 导入对象

1. **使用领域对象接受用户输入**

JSP中用(对象.属性)请求参数，如user.username

Action中包含对象和setter/getter方法，如

private User user,

public User getUser(){ return user;}

public void setUser(User user){ this.user = user;}

Struts2框架的数据绑定机制，不用创建User类的实例对象，遍历所有请求参数，自动依次实例化对象。例如Struts2尝试设置user对象的username属性时

action.getUser()返回值为null，则使用User类的默认构造方法（无参）创建User实例，并调用action.setUser(new User())，再调用action.getUser().setUsername()

1. **使用ModelDriven action**

JSP中直接使用属性，例如username

Action类实现ModelDriven<User>接口

private User user = new User();

public User getModel(){ return user;}

Struts2返回用于接收用户输入数据的模型对象。

1. **使用action的属性接受用户输入**

Action中直接设置属性

private String username;

public String getUsername(){return username;}

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

不需要JavaBean的User类。

### 访问request、session、application对象

HttpServletRequest

HttpSession

ServletContext

ActionContext.getContext()

ActionContext

context.get(“request”)

context.getSession()

context.getApplication()

RequestAware

ServletRequestAware

SessionAware

request.getSession()

ApplicationAware

ServletContextAware

ServletActionContext

getRequest()

request.getSession()

getServletContext()

ServletActionContext

耦合访问

解耦访问

JSP通过sessionScope.user.username，requestScope.greeting，application.counter引用

### 常量配置

src/struts.properties，其中Struts2的所有常量的默认值在struts2-core-2.2.1.jar的/org/apache/struts2/default.properties文件中。

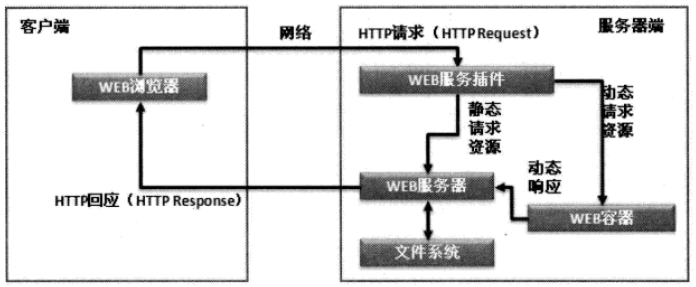
* struts.custom.i18n.resources：国际化资源文件，一般为message，即找寻src目录下的message.properties文件。
* struts.i18n.encoding：Web应用的默认编码集，一般为GBK

Struts2框架按如下顺序加载常量，后面的配置会覆盖前面的配置：

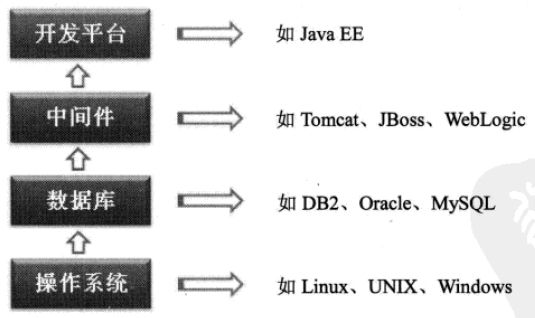
* struts-default.xml：在struts2-core-2.1.2.jar中
* struts-plugin.xml：在struts2-Xxx-2.1.2.jar等Struts2插件的JAR中
* struts.xml：src目录下Web应用默认的Struts2配置文件
* struts.properties：Struts2默认的配置文件
* web.xml：Web应用的配置文件

# Web

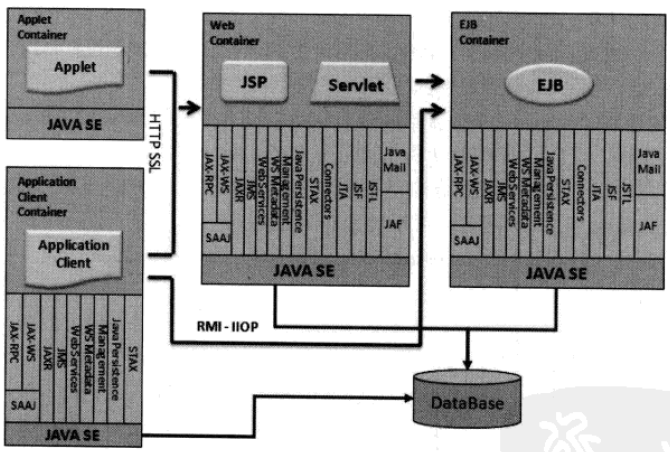
Java web技术：CGI->Servlet->JSP->J2EE



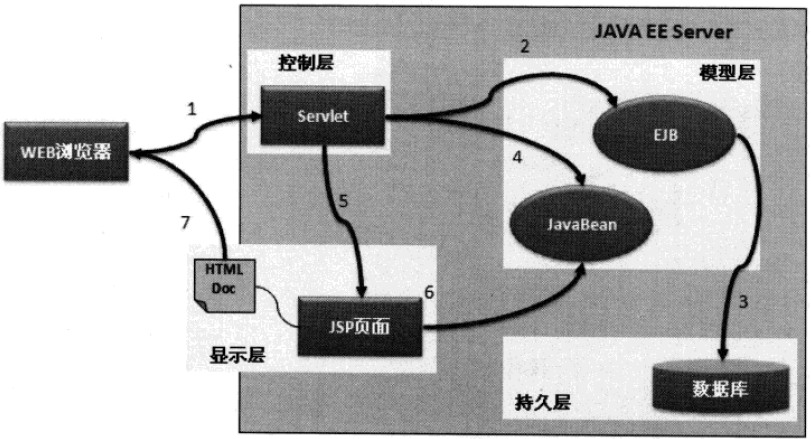
动态Web流程图，动态请求交给Web容器处理，返回拼凑代码动态生成数据，通过Web服务器返回客户端。静态Web和动态Web的区别在于是否访问数据库。动态Web实现方式有CGI、ASP、PHP、JSP\Servlet。



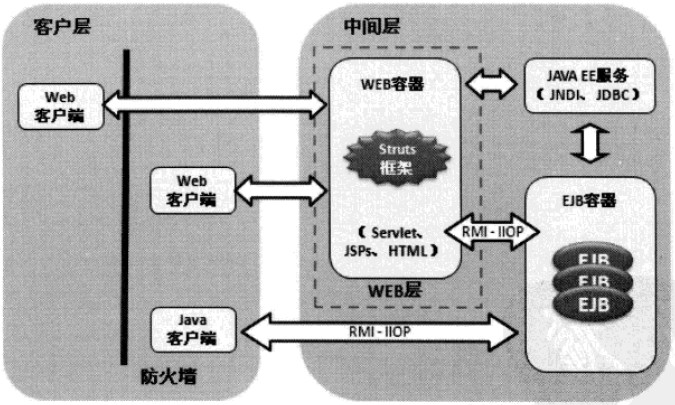
企业开发的核心架构



Java EE架构，由容器、组件、服务组成。四种容器，即Applet Container，Application Client Container，Web Container，EJB Container，每种容器包含若干种组件，如Web Container包含JSP和Servlet。



Java EE中的标准MVC设计模式，用户请求交给控制层处理，然后调用模型层中的模型组件访问持久层，结果保存在JavaBean中，最终由JSP和JavaBean一起完成显示。



Struts框架的基本组成

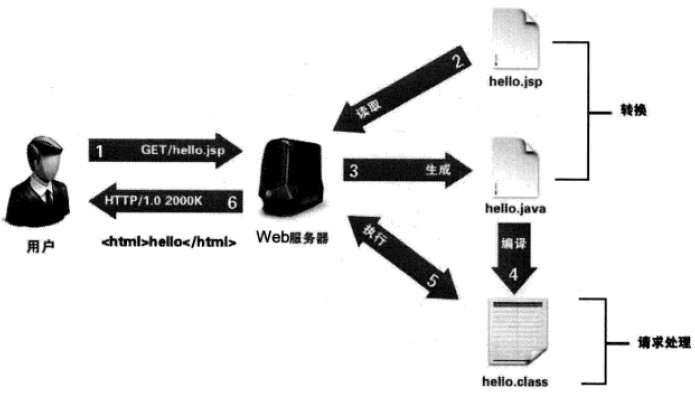
## XML

2种XML解析方式，即DOM和SAX：

* DOM解析是将所有内容读取到内存中，形成内存树，如果文件量较大则无法使用，但DOM解析可以进行文件修改。
* SAX解析是采用顺序的方式读取XML文件，不受文件大小限制，但是不允许修改。

## JSP

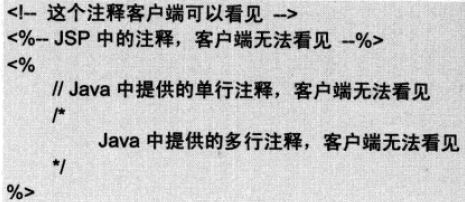
1. **原理**



JSP执行流程，将\*.jsp转换为\*\_jsp.java，最终编译为\*\_jsp.class，生成的java和class文件在TOMCAT-HOME/work/Catalina/localhost/workspace/org/apache/jsp目录中

1. **注释**

HTML注释<!-- --> JSP注释<% %>



* <%%>定义**局部**变量，编写语句，不能用private、public修饰，不能定义方法
* <%!%>定义**全局**变量、**方法**、类，对应Servlet的成员变量和方法
* <%=%>输出表达式，尽量用<%=%>代替out.println()
* <jsp:scriptlet></jsp:scriptlet>

### 3个编译指令

1. **page**

针对当前页面

<%@ page language=”java” contentType=”text/html;charset=GBK”%> //定义JSP字符的编码和页面响应的MIME类型

<%@ page language=”java” contentType=”text/html” pageEncoding=”GBK”%>//pageEncoding指定JSP文件的编码，contentType中的charset指服务器发送给客户端的内容编码

<% page import=”java.sql.\*” %>

设置错误页：

* <%@ page errorPage=”error.jsp”%> 指定错误出现时的跳转页
* <%@ page isErrorPage=”true”%> 指明此页处理错误

1. **include**

静态引入JSP页面

静态包含：先包含后处理

<%@include file=”info.htm”%>

动态包含：先处理后包含

<jsp:include page=”info.htm”/>

跳转指令：跳转后地址栏不变，属于服务器跳转，标签指令最后一定要有完结。

1. **taglib**

定义和访问自定义标签

### 7个动作指令

1. **jsp:forward**

执行页面跳转，将请求转发到下一个页面

<jsp:forward page=”forward.jsp”>

<jsp:param name=”name” value=”<%=username%>”/>

</jsp:forward>

<%=request.getParameter(“name”)%>

<jsp:forward>属于服务器跳转，执行到跳转语句时会立刻跳转，注意关闭JDBC；response.sendRedirect()属于客户端跳转，在整个页面执行完之后才执行跳转。

1. **jsp:param**

传递参数，必须与其他支持参数的标签一起使用

1. **jsp:include**

动态引入JSP页面

<jsp:include page=”scriptlet.jsp”>

<jsp:param name=”age” value=”32”>

</jsp:include>

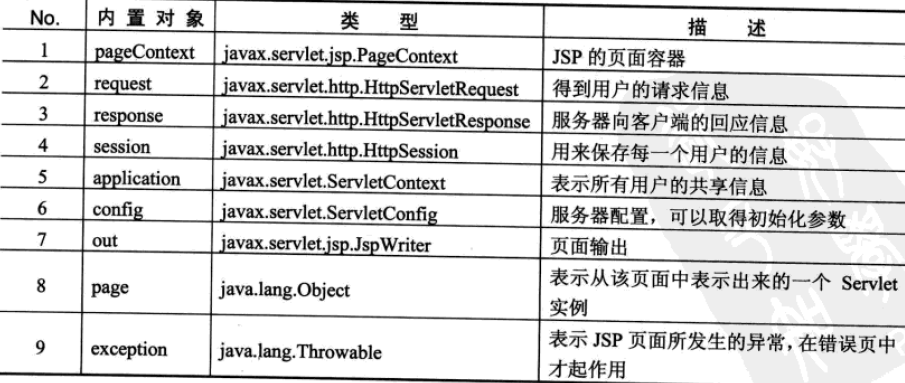
转换成Servlet为

org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response,“scriptlet.jsp”,out, false);

* 静态导入时将被导入页面的代码完全融入，两个页面融合成一个Servlet；而动态导入则在Servlet中使用include方法引入被导入页面。
* 静态导入时被导入页面的编译指令会起作用；而动态导入时被导入页面的编译指令失去作用，只插入被导入页面的body内容。
* 动态导入还可增加参数。

1. **jsp:plugin:**下载JavaBean或Applet到客户端
2. **jsp:useBean:**创建JavaBean实例
3. **jsp:setProperty:**设置JavaBean实例的属性
4. **jsp:getProperty:**输出JavaBean实例的属性

### 9个内置对象



JSP的9个内置对象，request、response是\_jspService()的形参，page、pageContext、application、config、session、out是\_jspService()的局部变量，由pageContext的方法获得

* page:只在一个页面中保存属性，跳转之后无效。
* request:只在一次请求中保存，服务器跳转后依然有效，客户端跳转后失效。
* session:在一次会话范围中，无论何种跳转都有效，但新开浏览器无法使用。
* application:在整个服务器上保存，所有用户都可以使用。JSP中直接使用application，Servlet中在service方法中调用ServletContext sc = getServletConfig().getServletContext();获得。
* config:当前JSP的配置信息，JSP中使用config，config.getInitParameter(“”)获得web.xml的配置参数
* exception:只有指定isErrorPage时，JSP才可使用exception对象。
* pageContext:可通过getAttribute(String name, int scope)访问page、request、session、application范围的变量
* request:重定向地址不变，request.getRequestDispatcher("/a.jsp”).include(request, response)和request.getRequestDispatcher("/a.jsp”).forward(request, response)
* response:重定向会失去所有请求参数，response.sendRedirect(“a.jsp”);

public void \_jspService(final javax.servlet.http.HttpServletRequest request, final javax.servlet.http.HttpServletResponse response)

throws java.io.IOException, javax.servlet.ServletException {

final javax.servlet.jsp.PageContext pageContext;

javax.servlet.http.HttpSession session = null;

final javax.servlet.ServletContext application;

final javax.servlet.ServletConfig config;

javax.servlet.jsp.JspWriter out = null;

final java.lang.Object page = this;

javax.servlet.jsp.JspWriter \_jspx\_out = null;

javax.servlet.jsp.PageContext \_jspx\_page\_context = null;

try {

response.setContentType("text/html;charset=gbk");

pageContext = \_jspxFactory.getPageContext(this, request, response,

null, true, 8192, true);

\_jspx\_page\_context = pageContext;

application = **pageContext.getServletContext**();

config = **pageContext.getServletConfig()**;

session = **pageContext.getSession()**;

out = **pageContext.getOut()**;

\_jspx\_out = out;

………

} catch (java.lang.Throwable t) {

if (!(t instanceof javax.servlet.jsp.SkipPageException)){

out = \_jspx\_out;

if (out != null && out.getBufferSize() != 0)

try { out.clearBuffer(); } catch (java.io.IOException e) {}

//如果\_jspx\_page\_context不为null，则由它处理异常，如果该页面的page中指定了errorPage属性，则将请求forward到errorPage指定的页面，否则使用系统页面输出异常。

if (\_jspx\_page\_context != null) \_jspx\_page\_context.handlePageException(t);

else throw new ServletException(t);

}

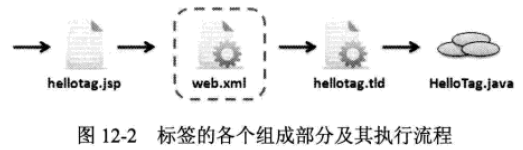
} finally {

\_jspxFactory.releasePageContext(\_jspx\_page\_context);

}

}

### 自定义标签



标准标签库：JSTL是Sun提供的一套标签库。

1. **标签处理类**

* 自定义标签继承自javax.servlet.jsp.tagext.SimpleTagSupport；
* 标签属性都有对应的getter和setter方法；
* 重写doTag()方法，生成页面内容。

public class DataTag extends TagSupport{

private String format;

public int doStartTag() throws JspException{

SimpleDateFormat df = new SimpleDateFormat(this.format);

super.pageContext.getOut().write(df.format(new Date()));

return TagSupport.SKIP\_BODY;

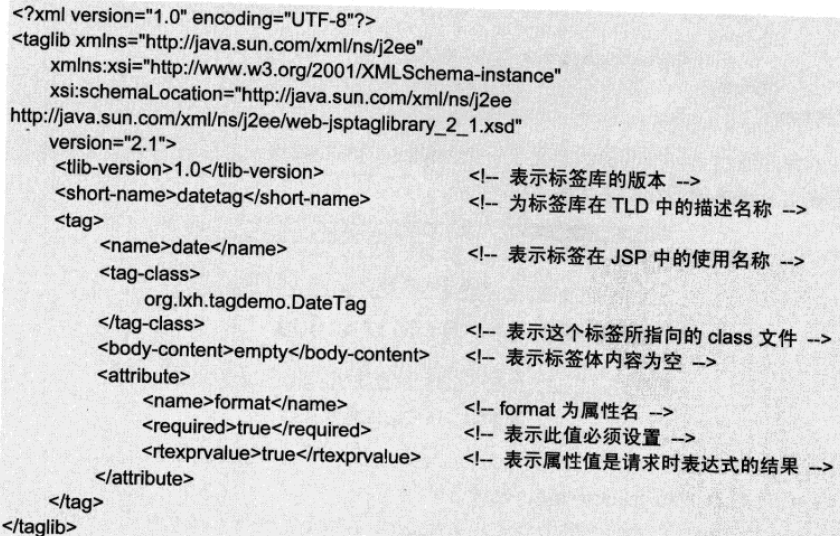
}

public String getFormat(){ return format;}

public void setFormat(String format){ this.format = format;}

}

1. **标签描述文件tld**



1. **JSP页面：通过<%@ taglib%>**

<%@ taglib prefix=”mytag” uri=”mldn\_date” %>

<mytag:date format=”yyyy-MM-dd HH:mm:ss.SSS”/>

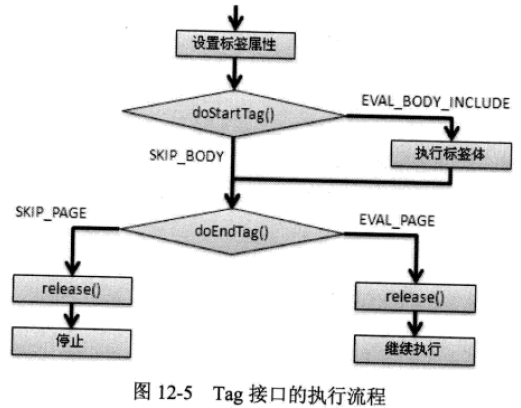
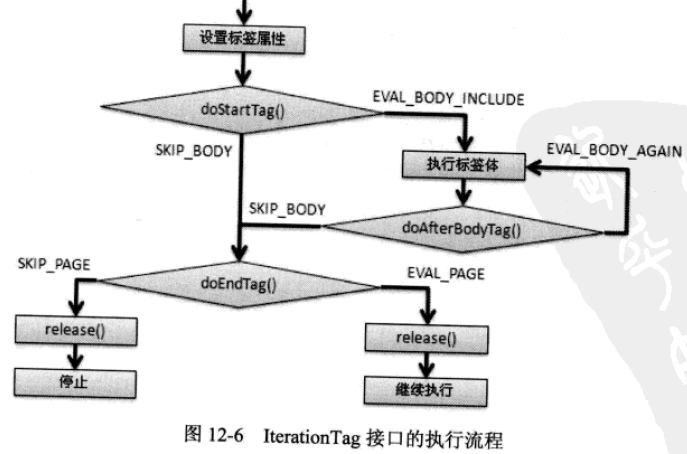
1. **在web.xml配置映射名称**

<taglib>

<taglib-uri>mldn\_date</taglib-uri>

<taglib-location>/WEB-INF/datatag.tld</taglib-location>

</taglib>

### 表单

表单提交方式分为GET和POST：

GET提交：将请求参数和值转换为字符串，附加在原URL后，因此提交内容会显示在地址栏，且传送数据量一般不超过2KB，长度受限，保密性差。

POST提交：请求参数大小不限，地址栏不会显示请求参数，可提交大文本和图片数据。

* 每个有name属性的表单才生成一个对应的请求参数。
* 如果多个表单有相同的name属性，只生成一个请求参数，该参数有多个值。
* 表单的name属性指定参数名，value指定参数值。

表达的enctype指定表单数据的编码方式，有三种值：

* application/x-www-form-urlencoded:默认编码方式，只处理表单的value值成URL编码方式。
* multipart/form-data:以二进制流方式处理表单数据，把文件内容封装到请求参数里。
* text/plain:当action属性为<mailto:URL>时，适用于直接通过表单发送邮件。

Servlet3文件上传：

<form method=”post” action=”upload” enctype=”multipart/form-data”>

<input type=”file” id=”file” name=”file”/>

<input type=”submit” value=”上传”/>

</form>

@MultipartConfig

public class UploadServlet extends HttpServlet{

public void service(HttpServletRequest request, HttpServletResponse response){

Part part = request.getPart(“file”);

part.write(getServletContext().getRealPath(“/uploadFiles”)+”/xxx.png”);

}

}

### Cookie

Cookie c = new Cookie(“name”, java.net.URLEncoder.encode(“孙悟空”,”gbk”));

c.setMaxAge(24\*3600);

response.addCookie(c);

Cookie[] cookies = request.getCookies();

for(Cookie cookie: cookies)

{

if(cookie.getName().equals(“name”))

{

out.println(java.net.URLDecoder.decode(cookie.getValue()));

}

}

## JavaBean

* JavaBean必须为public class，属性必须为private；
* JavaBean类有一个无参构造函数；
* 如果想取得或设定属性，必须使用getXXX()或setXXX()方法。

### Java Bean与表单交互

JSP中定义了<jsp:useBean>、<jsp:setProperty>、<jsp:getProperty>3个标签支持JavaBean的操作。

方法1：

<%@ page import=”com.mldn.lxh.demo.SimpleBean”%>

<%

SimpleBean simple = new SimpleBean();

simple.setName(request.getParameter(“name”));

simple.setAge(Integer.parseInt(request.getParameter(“age”)));

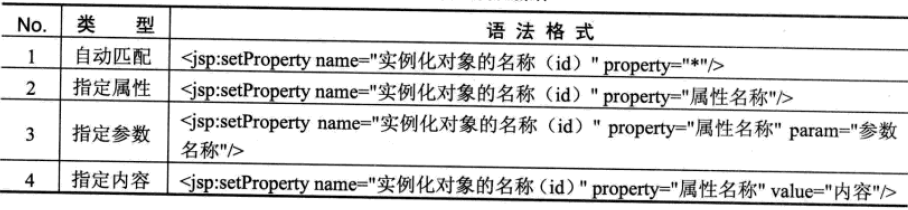
%>

方法2：

<jsp:useBean id=”simple” scope=”page” class=” com.mldn.lxh.demo.SimpleBean”/>

<jsp.setProperty name=”simple” property=”\*”/>

scope有四种属性值page|request|session|application



<jsp:getProperty name=”simple” property=”name”/>

### Dao设计模式

首先使用DAO**(Data Access Object，数据访问对象)**定义数据表的操作标准（即接口），然后通过代理类和真实类实现接口的具体操作，真实类完成具体的数据表操作，代理类负责控制数据库的打开及关闭，所有Dao接口的实例通过工厂类取得。

分为以下几个部分：

1. **DatabaseConnection**:数据库的打开与关闭

public interface DatabaseConnection{ //接口

public Connection getConnection();

public void close();

}

public class MySQLDatabaseConnection implements DatabaseConnection{

private static final String DBDRIVER = "com.mysql.jdbc.Driver";

private static final String DBURL = "jdbc:mysql://localhost:3306/mldn";

private static final String DBUSER = "root";

private static final String DBPASS = "";

private Connection conn = null;

public MySQLDatabaseConnection(){

try{

Class.forName(DBDRIVER);

conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);

}catch(Exception e){

e.printStackTrace();

}

}

public Connection getConnection{ return this.conn;}

public close(){

if(this.conn != null){

this.conn.close();

}

}

}

public class DatabaseConnectionFactory{

public static DatabaseConnection getDatabaseConnection(){

return new MySQLDatabaseConnection();

}

}

1. **VO**:JavaBean类，对应表中的一条记录
2. **DAO**:定义Impl的接口

public interface IVoDao{

//新增以doXXX命名

Public Boolean doCreate(Vo vo) throws Exception;

//查询以findXXX命名

public List<Vo> findAll(String keyword) throws Exception;

//ID查询

public Vo findById(int id) throws Exception;

}

1. **Impl**:Dao的具体实现

public class VoDaoImpl implements IVoDao{

private Connection conn = null;

public VoDaoImpl(Connection conn){

this.conn = conn;

}

…..//接口实现

}

1. **Proxy**:代理实现

public class VoDaoProxy implements IVoDao{

private DatabaseConnection dbc = null;

private IVoDao dao = null;

public VoDaoProxy()throws Exception{

this.dbc = DatabaseConnectionFactory. getDatabaseConnection();

this.dao = new VoDaoImpl(this.dbc.getConnection());

}

//接口实现

}

1. **Factory**:工厂类

public class DaoFactory{

public static IVoDao getVoDaoInstance() throws Exception{

return new VoDaoProxy();

}

}

## Servlet

Servlet继承自HttpServlet，**有2种配置方式**：

* 在Servlet上使用@WebServlet Annotaion配置

不要在web.xml的根元素<web-app>中指定metadata-complete=”true”

不要在web.xml中配置该Servlet

格式如：@WebServlet(name=”firstServlet”,

urlPatterns={”/firstServlet”},

loadOnStartup=1,

initParams={

@WebInitParam(name=”driver”, value=”com.mysql.jdbc.Driver”)

}

)

* 在web.xml中配置

<servlet>

<servlet-name></servlet-name>

<servlet-class></servlet-class>

<load-on-startup>1</load-on-startup>//启动时就载入

<init-param>

<param-name></param-name>

<param-value></param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name></servlet-name>

<url-pattern></url-pattern>

</servlet-mapping>

**web.xml配置顺序：**

1. 先配置过滤器：<filter>、<filter-mapping>
2. 再配置监听器：<listener>
3. 最后配置Servlet：<servlet>、<servlet-mapping>

**Servlet生命周期：**

1、Web容器通过web.xml加载Servlet，利用反射实例化；

2、容器调用init()初始化；

3、容器调用service()处理请求(doGet(),doPost())；

4、容器调用destroy()销毁；

5、等待垃圾回收器回收。

### 内置对象

1. **doGet，doPost直接传入HttpServletRequest，HttpServletResponse**
2. **获得ServletConfig**

配置web.xml：

<servlet>

<servlet-name></servlet-name>

<servlet-class></servlet-class>

<init-param>

<param-name></param-name>

<param-value></param-value>

</init-param>

</servlet>

覆盖HttpServlet的init方法

public void init(ServletConfig config){String tmp = config.getInitParameter(“”);}

1. **获得HttpSession**

doGet和doPost中

HttpSession ses = req.getSession();

1. **获得ServletContext**

通过GenericServlet的getServletContext()获得

ServletContext app = super.getServletContext();

### Servlet跳转

1. **客户端跳转**

resp.sendRedirect(“info.jsp”);//支持session,application传递属性

1. **服务器跳转**

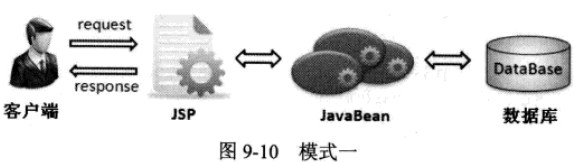


RequestDispatcher rd = req.getRequestDispatcher(“info.jsp”);

rd.forward(req,resp);

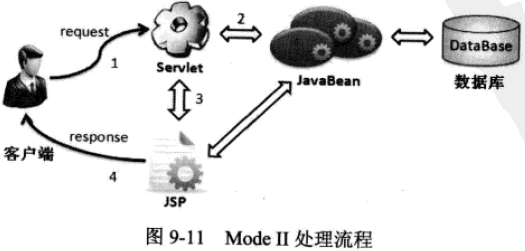
### MVC

Model I将MVC都交给JSP处理

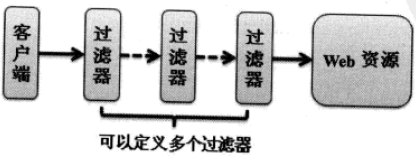


Model II：Model-View-Controller

Servlet接收请求，并调用JavaBean，将结果交给JSP显示



### 过滤器



public class EncodingFilter implements Filter{

private String charset;

public void init(FilterConfig config) throws ServletException{

this.charSet = config.getInitParameter(“charset”);

}

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

request.setCharacterEncoding(this.charSet);

}

public void destroy(){}

}

web.xml：

<filter>

<filter-name>encoding</filter-name>

<filter-class>org.lxh.filterdemo.EncodingFilter</filter-class>

<init-param>

<param-name>charset</param-name>

<param-value>GBK</param-value>

</init-param>

</filter>

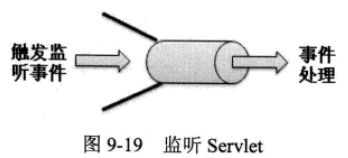
<filter-mapping>

<filter-name>encoding</filter-name>

<url-pattern>/\*</url-param>

</filter-mapping>

### 监听器



web.xml配置监听器：

<listener>

<listener-class>

org.lxh.listenerdemo.ServletContextListenerDemo

</listener-class>

</listener>

1. **对application监听**
2. **上下文状态监听: ServletContextListener**

public class ServletContextListenerDemo implements ServletContextListener{

public void contextInitialized(ServletContextEvent event){}

public void contextDestroyed(ServletContextEvent event){}

}

1. **上下文属性监听：ServletContextAttributeListener**

public class ServletContextAttributeListenerDemo implements ServletContextAttributeListener{

public void attributeAdded(ServletContextAttributeEvent event){}

public void attributeRemoved(ServletContextAttributeEvent event){}

public void attributeReplaced(ServletContextAttributeEvent event){}

}

1. **对session监听**
2. **session状态监听：HttpSessionListener**

public class HttpSessionListenerDemo implements HttpSessionListener{

public void sessionCreated(HttpSessionEvent event){}

public void sessionDestroyed(HttpSessionEvent event){}

}

1. **session属性监听：HttpSessionAttributeListener**

public class HttpSessionAttributeListenerDemo implements HttpSessionAttributeListener{

public void attributeAdded(HttpSessionBindingEvent event){}

public void attributeRemoved(HttpSessionBindingEvent event){}

public void attributeReplaced(HttpSessionBindingEvent event){}

}

1. **session属性监听：HttpSessionBindingListener(不需配置web.xml)**

public class HttpSessionBindingListenerDemo implements HttpSessionBindingListener{

public void valueBound(HttpSessionBindingEvent event){}

public void valueUnbound(HttpSessionBindingEvent event){}

}

1. **对request监听**
2. **请求状态监听：ServletRequestListener**

public class ServletRequestListenerDemo implements ServletRequestListener{

public void requestInitialized(ServletRequestEvent event){}

public void requestDestroyed(ServletRequestEvent event){}

}

1. **request属性监听：ServletRequestAttributeListener**

public class ServletRequestAttributeListenerDemo implements ServletRequestAttributeListener{

public void attributeAdded(ServletRequestAttributeEvent event){}

public void attributeRemoved(ServletRequestAttributeEvent event){}

public void attributeReplaced(ServletRequestAttributeEvent event){}

}

### 执行顺序

servlet：一种运行服务器端的java应用程序，具有独立于平台和协议的特性，并且可以动态的生成web页面，它工作在客户端请求与服务器响应的中间层。

filter：过滤器，是一个可以复用的代码片段，可以用来转换HTTP请求、响应和头信息。Filter不像Servlet，它不能产生一个请求或者响应，它只是修改对某一资源的请求，或者修改从某一的响应。

listener：监听器，从字面上可以看出listener主要用来监听。通过listener可以监听web服务器中某一个执行动作，并根据其要求作出相应的响应。通俗的语言说就是在application，session，request三个对象创建消亡或者往其中添加修改删除属性时自动执行代码的功能组件。

interceptor：是在面向切面编程的，就是在你的service或者一个方法前调用一个方法，或者在方法后调用一个方法，比如动态代理就是拦截器的简单实现，在你调用方法前打印出字符串（或者做其它业务逻辑的操作），也可以在你调用方法后打印出字符串，甚至在你抛出异常的时候做业务逻辑的操作。

servlet、filter、listener是配置到web.xml中，interceptor配置到spring.xml中。

web.xml 的加载顺序是：**context-param -> listener -> filter -> servlet** ，而同个类型之间的实际程序调用的时候的顺序是根据对应的 mapping 的顺序进行调用的。

### Servlet3新特性

1. **Annotation**

@WebServlet

@WebInitParam

@WebListener

@WebFilter

@MultipartConfig

@ServletSecurity

@HttpConstratint

@HttpMethodConstraint

1. **Web模块支持**

<webModule>.jar Web模块的JAR包

|——META-INF

|————web-fragment.xml

|——Web模块的类文件、资源文件等

web-fragment.xml用于管理WebJAR包的加载顺序：

<web-fragment ……>

<name></name>

<ordering>

<before>

<others/>

</before>

<after>

<name></name>

</after>

</ordering>

</web-fragment>

web.xml指定的加载顺序会覆盖web-fragment.xml。

<absolute-ordering>

<name></name>

<name></name>

</absolute-ordering>

1. **异步处理**

AsyncContext actx = request.startAsync();

actx.setTimeout(30\*1000);

actx.addListener(new MyAsyncListener())

actx.start(new Executor(actx));

Executor继承自Runnable

MyAsyncListener继承自AsyncListener，实现onStartAsync(AsyncEvent event)，onComplete(AsyncEvent event)，onError(AsyncEvent event)，onTimeout(AsyncEvent event)

开启Servlet异步调用：

* @WebServlet指定asyncSupported=true
* web.xml的<servlet>标签中增加<async-supported>true</async-supported>

## 表达式

${属性名称}

按page->request->session->application顺序查找属性

${pageScope.属性}、${requestScope.属性}、${sessionScope.属性}、${applicaionScope.属性}

接收请求参数：request.getParameter()=${param.属性}

一组参数${paramValues.属性}

**List操作：**

<%

List all = new ArrayList();

all.add(“li”);

all.add(“1111”);

request.setAttribute(“allinfo”,all);

%>

${allinfo[0]}

${allinfo[1]}

**Map操作：**

<%

Map map = new HashMap();

map.put(“li”,”1111”);

map.put(“se”,“2222”);

request.setAttribute(“info”, map);

%>

${info[“li”]}

${info.se}

**对象操作：**

<%

Dept dept = new Dept(); //对象属性提供getter和setter方法

dept.setDeptno(10);

dept.setDname(“xxxx”);

request.setAttribute(“deptinfo”, dept);

%>

${ deptinfo.deptno}

${ deptinfo.dname}

# Tomcat

## 配置Tomcat

1. **由TOMCAT\_HOME\webapps\manager\WEB-INF\web.xml可知**

//配置访问HTML的角色

<security-role>

<description>

The role that is required to access the HTML Manager pages

</description>

<role-name>manager-gui</role-name>

</security-role>

//能够访问路径为/html/\*的角色为manager-gui和manager

<security-constraint>

<web-resource-collection>

<web-resource-name>HTML Manager commands</web-resource-name>

<url-pattern>/html/\*</url-pattern>

</web-resource-collection>

<auth-constraint>

<!-- NOTE: 1. These roles are not present in the default users file

2. The manager role is deprecated, it will be removed in

Tomcat 7.

3. Use just the manager-gui role to take advantage of the new

CSRF protection. Assigning the manager role or manager-gui

role along with either the manager-script or manager-jmx

roles to the same user will bypass the CSRF protection. -->

<role-name>manager-gui</role-name>

<role-name>manager</role-name>

</auth-constraint>

</security-constraint>

1. **在TOMCAT\_HOME\conf\tomcat-users.xml中配置用户并分配角色**

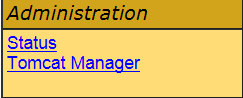
<role rolename="manager-gui"/>

<user username="manager" password="manager" roles="manager-gui"/>

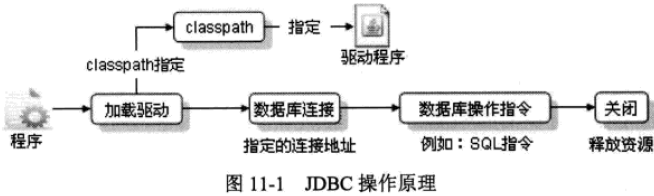
</tomcat-users>

1. **TOMCAT\_HOME\bin\startup.bat启动tomcat**

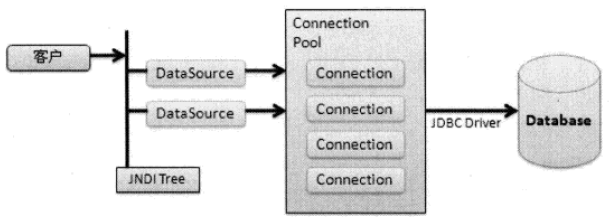
<http://localhost:8080>访问主页

点击Tomcat Manager，输入用户密码

## Tomcat数据源



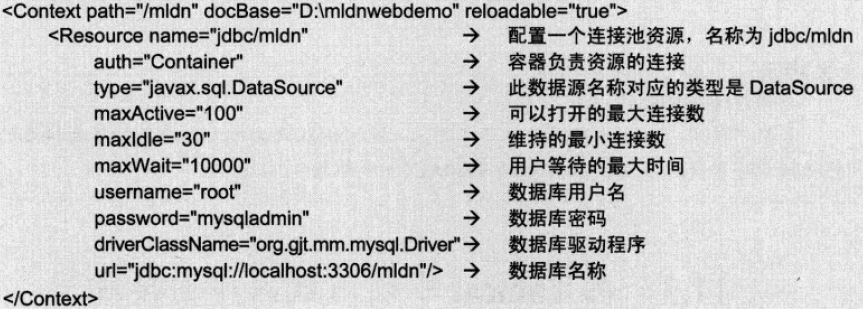
在对象池中保存多个数据库连接（Connection Pool），进行数据库操作时，直接从连接池中取出一个数据库连接，操作完成后，将连接放回数据库连接池中，等待其他用户继续使用，提升数据库操作性能，避免类加载、数据库连接、数据库关闭等重复操作。



Tomcat数据源操作，JNDI（Java Naming and Directory Interface，Java命名及目录接口，通过key查找对应的对象value）服务获得DataSource，DataSource从连接池中取得Connection，操作数据库

1. **配置数据源**

server.xml：



<Resource>配置连接池选项，name为jdbc/mldn（JNDI，Java Naming Directory Interface，Java命名和目录接口，为数据源取名为jdbc/xxx，其他程序可通过该名字访问到该数据源，由于多种资源，因此以jdbc开头，以示区分）

web.xml：



1. **操作数据源**

import javax.sql.\* //DataSource所在的包

import java.sql.\* //Connection所在的包

String DSNAME = “java:comp/env/jdbc/mldn”;

Context ctx = new InitialContext();

//java:comp/env是环境命名上下文，解决JNDI查找冲突

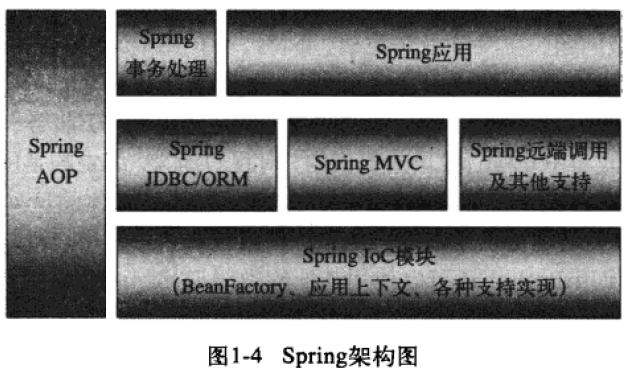
DataSource ds = (DataSource)ctx.lookup(DSNAME);

Connection conn = ds.getConnection();

# Spring

非入侵性框架，POJO开发，核心是IoC容器（控制反转/依赖注入）和AOP（面向切面/接口编程）

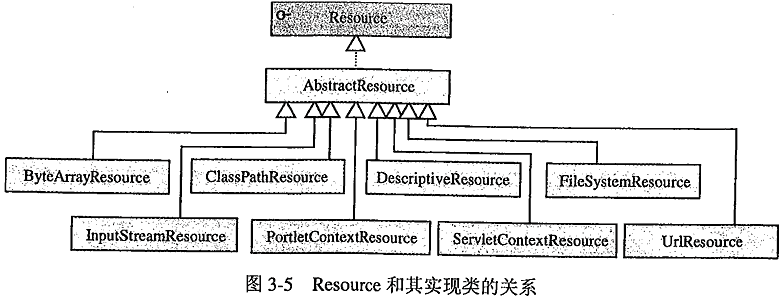
SSH：Structs2+Spring+Hibernate

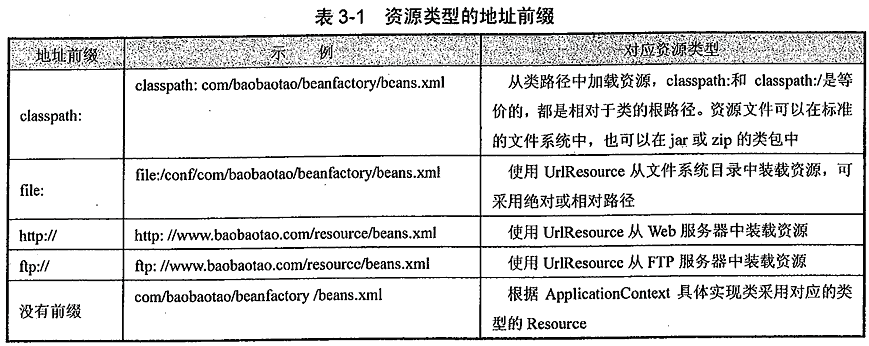


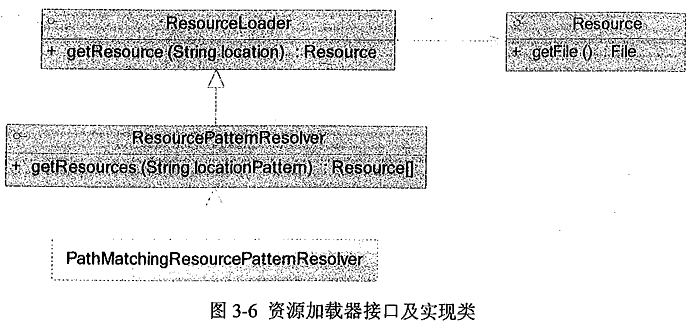
## IoC容器

IoC容器：依赖注入，控制反转，把依赖关系的管理从Java对象中解放出来，用配置的方式将对象-对象转变为对象-IoC容器-对象。

1. **Spring IoC容器的三个基本要点**
2. 面向接口编程；
3. 各组件不再由程序产生，而是由Spring容器负责产生并初始化；
4. Spring采用配置文件或注释管理Bean的实现类和依赖关系，Spring容器根据配置文件，利用反射来创建实例并为之注入依赖关系。
5. **注入方法**
6. 构造函数注入<constructor-arg ref=”” index=”0”>
7. 属性注入（推荐）<property name=”” ref=””>
8. 接口注入（不推荐）
9. **加载资源**







ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();

Resource res = resolver.getResource(“classpath:com/baobaotao/beanfactory/beans.xml”);

BeanFactory bf = new XmlBeanFactory(res);

Car car = bf.getBean(“car”, Car.class);

1. **Spring的两个核心接口**

BeanFactory（IoC容器）和ApplicationContext（应用上下文），ApplicationContext是BeanFactory的子接口。

区别是1)ApplicationContext在初始化应用时就实例化所有单实例的Bean，BeanFactory访问时才实例Bean。2)ApplicationContext会利用Java反射机制自动识别出配置文件中定义的BeanPostProessor、InstantiationAwareBeanPostProcessor和BeanFactoryPostProcessor，自动注册到应用上下文中。BeanFactory必须手工调用addBeanPostProcessor()注册。

Java EE应用采用FileSystemXmlApplicationContext，ClassPathXmlApplicationContext或AnnotationConfigApplicationContext。

Web应用采用XmlWebApplicationContext，AnnotationConfigWebApplicationContext实现类。

1. **Bean的生命周期**

通过getBean()调用某个Bean

调用InstantiantionAwareBeanPostProcessor的postProcessBeanBeforeInstantiation()方法

实例化

调用InstantiantionAwareBeanPostProcessor的postProcessBeanAfterInstantiation()方法

调用InstantiantionAwareBeanPostProcessor的postProcessPropertyValues()方法

设置属性值

调用BeanNameAware的setBeanName()方法

调用BeanFactoryAware的setBeanFactory()方法

调用BeanPostProcessor的postProcessBeforeInitialization()方法

调用InitializingBean的afterPropertiesSet()方法

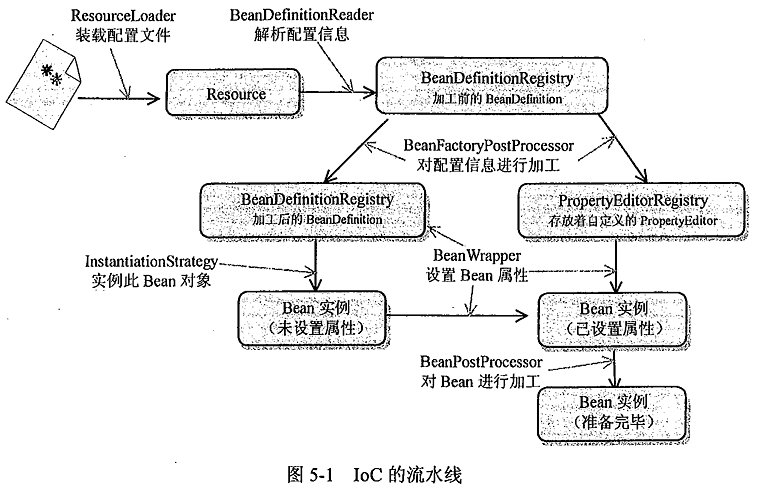
通过init-method属性配置的初始化方法

调用BeanPostProcessor的postProcessAfterInitialization()方法

如果是prototype，则将准备就绪的Bean交给调用者后就不管了

如果是singleton，则将Bean放到Spring缓存池中，容器销毁时调用DisposableBean的afterPropertiesSet()方法，再调用通过destroy-method属性配置的销毁方法

1. **内部工作机制**



## AOP

JDK动态代理：通过Proxy和InvocationHandler实现。

CGLib动态代理：通过MethodInterceptor实现。

前置增强，后置增强，环绕增强，异常抛出增强，引介增强。

# Spring MVC

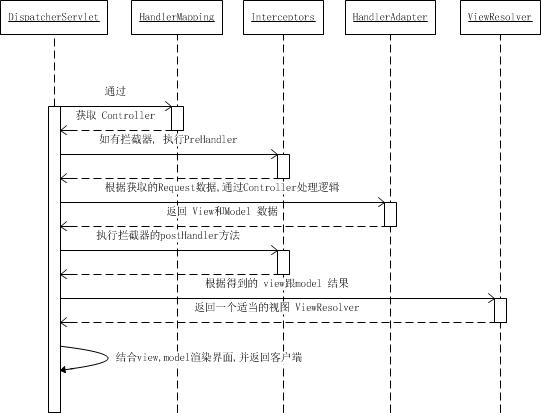
<http://my.oschina.net/lichhao/blog/99039>

* **类中所有的变量声明，几乎都以接口的形式给出，并没有绑定在具体的实现类上。**
* **使用模版方法模式，在父类中对基础行为进行定义，让子类实现模版方法扩展行为。**

Spring MVC在spring-web.jar和spring-webmvc.jar包中。

步骤为：

1. 初始化Root Web Container。
2. 配置Dispatcher Servlet
3. 实现Controller
4. 在DispatcherServlet中注册Controller
5. 实现View
6. 在DispatcherServlet中注册ViewResolver
7. 部署和测试



1. 请求首先 由 前端 DispatcherServlet 捕获；

2. DispatcherServlet 经过 HandlerMapping 获取适当的 Hanlder ，也即 Controller ，并返回给 DispatcherServlet;

3. 如果有设置 拦截器，首选执行拦截器的 preHandler 方法，并把执行结果返回 DispatcherServlet;

4. DispatcherServlet 根据捕获的请求，以及 Handler (Controller) ，获取到适当的 HandlerAdapter 处理，并把结果返回给 DispatcherServlet，结果为 (View 和 Model);

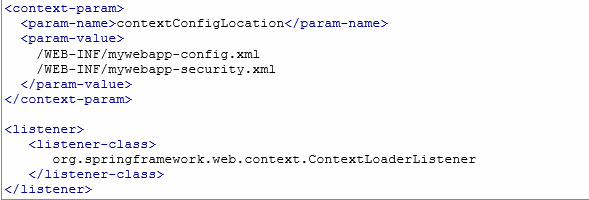
5. 如果有设置拦截器，就是执行 拦截器的 postHandler 方法，并返回 DispatcherServlet;

6. DispatcherServlet 根据获取的 view 跟 model 结合 ViewResolver，返回所指的 视图模板，并返回给 DispatcherServlet;

7. DispatcherServlet 结合 视图模型跟model ，执行 render() 渲染界面，并返回给客户端；

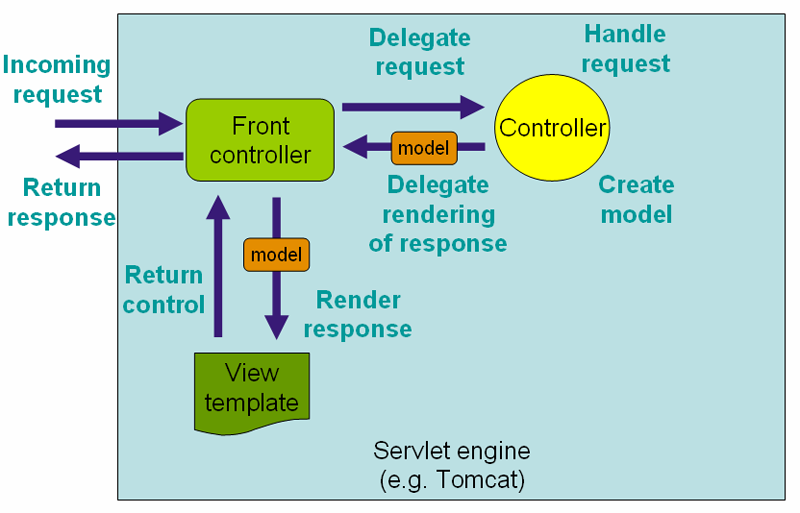
## ContextLoaderListener

**ContextLoaderListener应用于application的整个生命周期，用于在web应用中生成Bean。**



## DispatcherServlet

默认配置文件在org.springframework.web.servlet下的DsipatcherServlet.properties中。



关键对象：

private MultipartResolver **multipartResolver**; multipart格式请求处理

private LocaleResolver **localeResolver**; 本地字符集处理

private ThemeResolver **themeResolver**; 主题处理

private List<HandlerMapping> **handlerMappings**; 请求与controller映射

private List<HandlerAdapter> **handlerAdapters**;请求处理

private List<HandlerExceptionResolver> **handlerExceptionResolvers**; 出错处理

private RequestToViewNameTranslator viewNameTranslator;

private FlashMapManager flashMapManager;

private List<ViewResolver> **viewResolvers**; 视图处理

**protected** **void** doDispatch(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

HttpServletRequest processedRequest = request;

HandlerExecutionChain mappedHandler = **null**;

**boolean** multipartRequestParsed = **false**;

WebAsyncManager asyncManager = WebAsyncUtils.*getAsyncManager*(request);

**try** {

ModelAndView mv = **null**;

Exception dispatchException = **null**;

**try** {

//查看请求是否是multipart/form类型，是则转换

processedRequest = checkMultipart(request);

multipartRequestParsed = (processedRequest != request);

// 找到能够处理当前请求的HanderExecutionChain

mappedHandler = getHandler(processedRequest);

**if** (mappedHandler == **null** || mappedHandler.getHandler() == **null**) { //如果找不到就抛异常或者返回404

noHandlerFound(processedRequest, response);

**return**;

}

// 找到能够处理当前请求的HandlerAdapter

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// 对GET和HEAD请求，处理head中的Last-Modify字段

String method = request.getMethod();

**boolean** isGet = "GET".equals(method);

**if** (isGet || "HEAD".equals(method)) {

**long** lastModified = ha.getLastModified(request, mappedHandler.getHandler());

**if** (logger.isDebugEnabled()) {

logger.debug("Last-Modified value for [" + *getRequestUri*(request) + "] is: " + lastModified);

}

**if** (**new** ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {

**return**;

}

}

//执行HanderExecutionChain中包含的所有Interceptor.preHandler方法

**if** (!mappedHandler.applyPreHandle(processedRequest, response)) {

**return**;

}

**// 调用HandlerAdapter处理请求，返回ModelAndView**

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

**if** (asyncManager.isConcurrentHandlingStarted()) {

**return**;

}

//如果ModelAndView中没有ViewName，则设置默认View

applyDefaultViewName(processedRequest, mv);

//执行HanderExecutionChain中包含的所有Interceptor.postHandler方法

mappedHandler.applyPostHandle(processedRequest, response, mv);

}

**catch** (Exception ex) {

dispatchException = ex;

}

**//渲染View，处理异常**

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

}

//对所有异常，都执行HanderExecutionChain中包含的所有Interceptor.afterCompletion方法

**catch** (Exception ex) {

triggerAfterCompletion(processedRequest, response, mappedHandler, ex);

}

**catch** (Error err) {

triggerAfterCompletionWithError(processedRequest, response, mappedHandler, err);

}

**finally** {

**if** (asyncManager.isConcurrentHandlingStarted()) {

// Instead of postHandle and afterCompletion

**if** (mappedHandler != **null**) {

mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);

}

}

**else** {

// Clean up any resources used by a multipart request.

**if** (multipartRequestParsed) {

cleanupMultipart(processedRequest);

}

}

}

}

**private** **void** processDispatchResult(HttpServletRequest request, HttpServletResponse response,

HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) **throws** Exception {

**boolean** errorView = **false**;

**//如果发生异常，则处理异常**

**if** (exception != **null**) {

**if** (exception **instanceof** ModelAndViewDefiningException) {

logger.debug("ModelAndViewDefiningException encountered", exception);

mv = ((ModelAndViewDefiningException) exception).getModelAndView();

}

**else** {

Object handler = (mappedHandler != **null** ? mappedHandler.getHandler() : **null**);

//处理异常方法

mv = processHandlerException(request, response, handler, exception);

errorView = (mv != **null**);

}

}

// 如果提供mv，且不需清除，则渲染View

**if** (mv != **null** && !mv.wasCleared()) {

render(mv, request, response);

**if** (errorView) {

WebUtils.*clearErrorRequestAttributes*(request);

}

}

**else** {

**if** (logger.isDebugEnabled()) {

logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +

"': assuming HandlerAdapter completed request handling");

}

}

**if** (WebAsyncUtils.*getAsyncManager*(request).isConcurrentHandlingStarted()) {

// Concurrent handling started during a forward

**return**;

}

**if** (mappedHandler != **null**) {

mappedHandler.triggerAfterCompletion(request, response, **null**);

}

}

配置方法：



默认值：

|  |  |  |
| --- | --- | --- |
| Strategy Interface | Default Implementation | Description |
| HandlerMapping | DefaultAnnotationHandlerMapping | 根据@Controller和@RequestMapping将URL映射到Controller上 |
| HandlerAdapter | AnnotationMethodHandlerAdapter | 根据@RequestParam和@PathVariable将参数注入到handler方法中 |
| ViewResolver | InternalResourceViewResolver | 将View名称映射到路径 |
| HandlerExceptionResolver |  | 映射异常到View |

重写默认值

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="order" value="0">

</bean>

<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping">

<property name="order" value="1">

</bean>

## HandlerMapping

**HandlerMapping接口：定义了请求URL和处理器（Handler）的映射关系。**

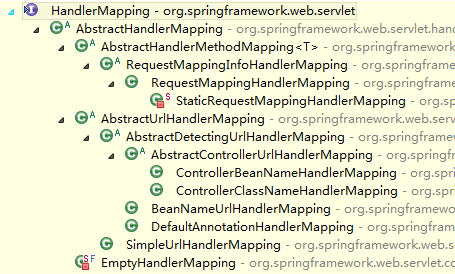
**AbstractHandlerMapping继承自HandlerMapping接口：**

1. **HandlerInterceptor列表；**
2. **defaultHandler：没有匹配的handler时默认执行的handler；**
3. **order：handler的处理优先级，org.springframework.core.Ordered；**
4. **alwaysUseFullPath：true在Servlet context的所有路径中找合适的handler，false，只在当前Servlet mapping中找。**
5. **urlDecode：true采用解密路径， false采用加密路径。**

public interface HandlerMapping {

HandlerExecutionChain getHandler(HttpServletRequest rqst) throws Exception;

}



如果上下文声明了HandlerMapping，则不提供默认值，除非显示声明<mvc:annotation-driven>

实现类：

* DefaultAnnotationHandlerMapping：默认值，提供基于注释(@RequestMapping)的映射。
* BeanNameUrlHandlerMapping：默认值，提供URL与Bean名称的映射，如URL为 “/hello”，则Spring容器中必须有一个名字为“hello”的Bean。
* SimpleUrlHandlerMapping：通过显式配置提供URL与Controller的映射。
* ControllerClassNameHandlerMapping：提供URL与类名的映射。控制器必须以Controller结尾并有@Controller注释。将控制器类名的Controller后缀删除，转换成小写后作为请求url的前缀，例如HomeController映射/home\*。

## HandlerAdapter

**Handler Mapping选择好Controller后，由Handler Adapter调用具体的handler方法，包括注入正确参数，通过Java反射调用方法，并处理返回值。**多个Handler Adapter则以DispatcherServlet最早找到的支持Handler Mapping的为准。

public interface HandlerAdapter {

//Check if controller is supported

boolean supports(Object handler);

//handle request

ModelAndView handle(HttpServletRequest rqst,

HttpServletResponse rsp,

Object handler) throws Exception;

}

* AnnotationMethodHandlerAdapter：默认值，适用于注释了@RequestMapping的方法，

|  |  |  |
| --- | --- | --- |
| Type | Description | Created By |
| Model | Map of attributes and values | Spring MVC |
| HttpServletRequest | HTTP Request | Servlet Container |
| HttpServletResponse | HTTP Response | Servlet Container |
| HttpSession | HTTP Session | Servlet Container |
| Locale | Request preferred locale | Servlet Container |
| Principal | User identify | Servlet Container |
| SessionStatus | Session status | Spring MVC |
| WebDataBinder | Data-binding and validation control | Spring MVC |
| (domain object) | subject to data-binding | Spring MVC or application |

## HttpMessageConverter

HttpMessageConverter将HTTP请求体转换为输入参数，并将输出结果转换为HTTP响应体。

默认包含以下转换类：

* ByteArrayHttpMessageConverter converts byte arrays.
* StringHttpMessageConverter converts strings.
* ResourceHttpMessageConverter converts to/from org.springframework.core.io.Resource for all media types.
* SourceHttpMessageConverter converts to/from a javax.xml.transform.Source.
* FormHttpMessageConverter converts form data to/from a MultiValueMap<String, String>.
* Jaxb2RootElementHttpMessageConverter converts Java objects to/from XML — added if JAXB2 is present and Jackson 2 XML extension is not present on the classpath.
* MappingJackson2HttpMessageConverter converts to/from JSON — added if Jackson 2 is present on the classpath.
* MappingJackson2XmlHttpMessageConverter converts to/from XML — added if [Jackson 2 XML extension](https://github.com/FasterXML/jackson-dataformat-xml) is present on the classpath.
* AtomFeedHttpMessageConverter converts Atom feeds — added if Rome is present on the classpath.
* RssChannelHttpMessageConverter converts RSS feeds — added if Rome is present on the classpath.

public interface HttpMessageConverter<T> {

// Indicate whether the given class and media type can be read by this converter.

boolean canRead(Class<?> clazz, MediaType mediaType);

// Indicate whether the given class and media type can be written by this converter.

boolean canWrite(Class<?> clazz, MediaType mediaType);

// Return the list of MediaType objects supported by this converter.

List<MediaType> getSupportedMediaTypes();

// Read an object of the given type from the given input message, and returns it.

T read(Class<T> clazz, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException;

// Write an given object to the given output message.

void write(T t, HttpOutputMessage outputMessage) throws IOException, HttpMessageNotWritableException;

}

配置方法：

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

<property name="messageConverters">

<util:list id="beanList">

<ref bean="stringHttpMessageConverter"/>

<ref bean="marshallingHttpMessageConverter"/>

</util:list>

</property

</bean>

<bean id="stringHttpMessageConverter"

class="org.springframework.http.converter.StringHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter"

class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">

<property name="marshaller" ref="castorMarshaller" />

<property name="unmarshaller" ref="castorMarshaller" />

</bean>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>

## HandlerInterceptor

在handler执行前后添加处理规则，必须继承自HandlerInterceptor

public interface HandlerInterceptor {

//invoked before controller handler invocation

boolean preHandle(HttpServletRequest rqst,

HttpServletResponse rsp,

Object handler) throws Exception;

//invoked after controller handler invocation

void postHandle(HttpServletRequest rqst,

HttpServletResponse rsp,

Object handler,

ModelAndView mv) throws Exception;

//invoked after view rendering

void afterCompletion(HttpServletRequest rqst,

HttpServletResponse rsp,

Object handler,

Exception ex) throws Exception;

}

配置方法：

<mvc:interceptors>

<bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />

<mvc:interceptor>

<mvc:mapping path="/\*\*"/>

<mvc:exclude-mapping path="/admin/\*\*"/>

<bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor" />

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path="/secure/\*"/>

<bean class="org.example.SecurityInterceptor" />

</mvc:interceptor>

</mvc:interceptors>

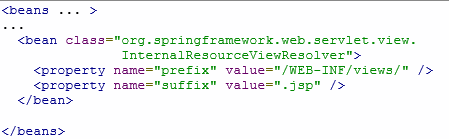
## ViewResolver

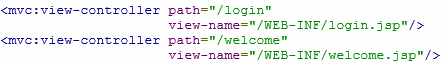
public interface ViewResolver {

View resolveViewName(String name, Locale loc);

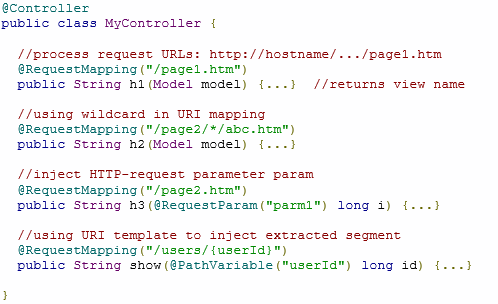
}

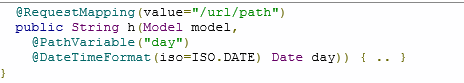
|  |  |
| --- | --- |
| 视图解析器 | 描述 |
| XmlViewResolver | 接口 ViewResolver 的实现，从 XML 配置文件中查找视图实现（默认 XML 配置文件为 /WEB-INF/views.xml）. |
| ResourceBundleViewResolver | 接口 ViewResolver 的实现，用于从 properties 文件中查找视图。. |
| UrlBasedViewResolver | 接口 ViewResolver 的实现，用于根据请求的 URL 路径返回相应的视图，该视图需为抽象类 AbstractUrlBasedView 的实现，它还有些子类，如 InternalResourceView 和 JstlView 等 . |
| InternalResourceViewResolver | UrlBasedViewResolver 的子类，通常用于查找 JSP（类 InternalResourceView）和 JSTL（类 JstlView，InternalResourceView 的子类）等视图。 |
| VelocityViewResolver /FreeMarkerViewResolver | UrlBasedViewResolver 的子类分别用于支持 Velocity（类 VelocityView）和 FreeMark 视图（类 FreeMarkerView）。 |
| ContentNegotiatingViewResolver | 接口 ViewResolver 的实现，用于根据请求文件的后缀名或请求的 header 中的 accept 字段查找视图。 |

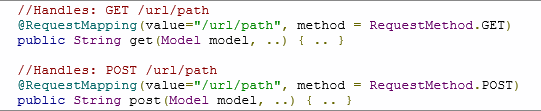


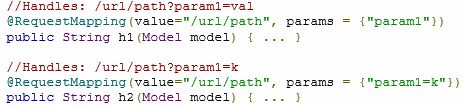


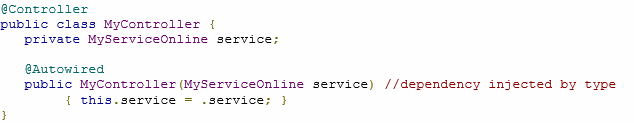
## Controller

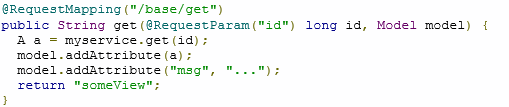










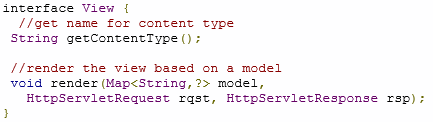


注册Controller



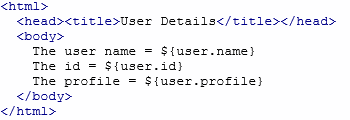
还要声明<mvc:annotation-driven/>，注解才会被支持。

## View



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| View Kind | Technology | Format | View Class | ViewResolver |
| Display | JSP | (X)HTML/XML | JstlView | InternalResourceViewResolver |
|  | Tiles | (X)HTML/XML |  | TilesViewResolver |
|  | FreeMarker | (X)HTML/XML |  | FreeMarkerViewResolver |
|  | Velocity | (X)HTML/XML |  | VelocityViewResolver |
| File-generating | POI | Excel |  |  |
|  | jExcelApi | Excel |  |  |
|  | Itext | PDF |  |  |
|  | JasperReports |  |  | JasperReportsViewResolver |
|  | XSLT | (X)HTML/XML |  | XsltViewResolver |
| Data-delivery | JSON | JSON |  |  |
|  | Object Marshalling | Java Serialization |  |  |
|  |  | XML |  | XmlViewResolver |
|  | Atom |  |  |  |
|  | RSS |  |  |  |

1. handler返回AbstractView，如JstlView，则直接使用；
2. 返回String，则DispatcherServlet根据ViewResolver解析成文件路径名；
3. 返回为null或不反悔，则根据默认规则解析，RequestToViewNameTranslator实现该规则，默认值为DefaultRequestToViewNameTranslator，删除域名、应用名、后缀名，如http://hostname/webapp/path/page.htm变为/path/page。



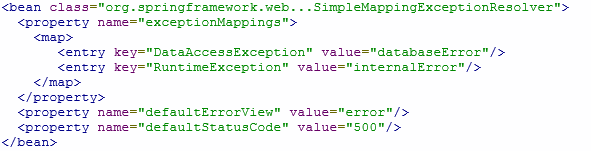
## HandlerExceptionResolver

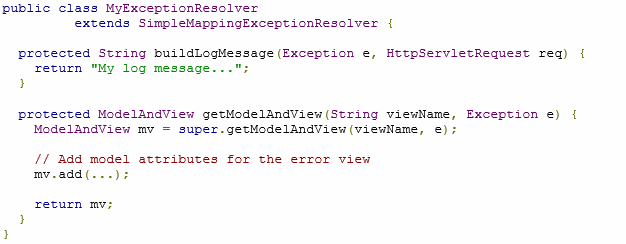
SimpleMappingExceptionResolver将异常根据名称映射到视图。

exceptionMappings属性包含一个map，用于配置异常名称到视图的映射。

defaultErrorView是没有映射关系时默认显示的视图。

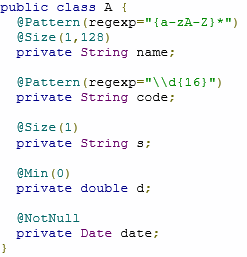
defaultStatusCode无法解析出映射关系时的返回值。

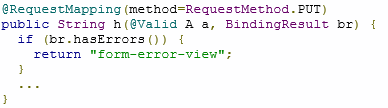




## Date Validation

|  |  |  |  |
| --- | --- | --- | --- |
| Annotation | Type(s) | Description | JSR 303 |
| @NotNull | Object | Field cannot be null | Yes |
| @Size(min, max) | String | Field must have length in range [min,max] | Yes |
| @Min | Numeric | Field minimum value | Yes |
| @Max | Numeric | Field maximum value | Yes |
| @Pattern | String | Field matches pattern | Yes |
| @NotEmpty | Object | Field length is non-zero |  |





## ContentNegotiation

内容协商，是指HTTP头中的Content-Type和Accept决定返回结果的类型，顺序如下：

1. URL后缀，如http://localhost/myapp/accounts/list.html，返回html
2. URL中包含format，如http://localhost/myapp/accounts/list?format=xls，format是默认名称
3. HTTP头中的ACCEPT属性

<bean id=*"contentNegotiationManager"* class=*"org.springframework.web.accept.ContentNegotiationManagerFactoryBean"*>

<property name=*"favorPathExtension"* value=*"true"* /> 支持后缀

<property name=*"favorParameter"* value=*"false"* /> 不支持format

<property name=*"ignoreAcceptHeader"* value=*"true"*></property> 不支持Accept

<property name=*"defaultContentType"* value=*"application/json"*></property> 默认返回类型

<property name=*"mediaTypes"*>

<value>

json=application/json

xml=application/xml

</value>

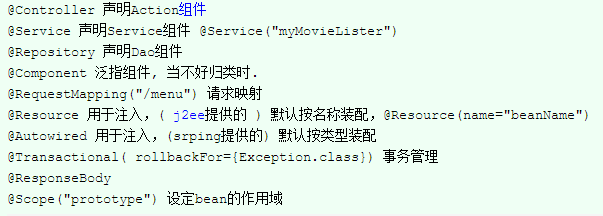
</property>

</bean>

## mvc:annotation-driven

<mvc:annotation-driven>作用：

1. 默认注册 DefaultAnnotationHandlerMapping以及AnnotationMethodHandlerAdapter两个 Bean，从而能够扫描以下注解。



1. 类型转换，格式化和有效性检查**LocalValidatorFactoryBean**
2. MessageConverter，以支持REST应用

## 问题

**问题：采用注解不成功，抛异常Fail load ApplicationContext**

spring context的版本号用4.0.0.RELEASE

**<context:annotation-config>：自动装配，但不扫描Bean**

**<context:component-scan>：自动扫描Bean和装配**

**问题：jetty访问jsp页面出现（ PWC6345: There is an error in invoking javac）**

在eclipse——>Window——>prefence——>java——>Installed JREs,选择jre点击Edit找到jdk的安装目录

**问题：事务不回滚**

数据库引擎未支持回滚（或表引擎不支持）

default-table-type=innodb

**问题：事务的readOnly属性不生效**

HibernateTransactionManager：在propagation为REQUIRED时生效，为SUPPORTS时不生效；

DataSourceTransactionManager：无论何时都不生效。

**问题：Spring MVC JSon未返回某列**

该对象的该列缺少set和get方法

DAO层用@Repository

Service层用@Service

Controller层用@Controller

@ResponseBody注解，表示该方法的返回结果直接写入HTTP response body中。

**问题：Transaction rolled back because it has been marked as rollback-only**

内层方法抛异常，中断事务，外层也设置事务

**问题：log4j:WARN No appenders could be found for logger (org.springframework.web.context.ContextLoader).**

log4j依赖于commons-logging-1.1.1.jar，log4j-1.2.14.jar两个包，可以将这两个包放到每个项目的WEB-INF/lib目录下，或者直接放到apache-tomcat-6.0.37\lib公共目录（不用每个项目都配置）。

在web.xml中配置log4j：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/appcontext.xml</param-value>

</context-param>

<context-param>

<param-name>log4jConfigLocation</param-name>

<param-value>/WEB-INF/log4j.properties</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.util.Log4jConfigListener

</listener-class>

</listener>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

将log4j.properties放到/WEB-INF/目录下

**问题：Error configuring application listener of class org.springframework.web.context.ContextLoaderListener**

原因1：检查项目是否导入web包

tomcat/webapps/项目/WEB-INF/lib/org.springframework.web-3.0.1.RELEASE-A.jar

原因2：tomcat缓存

将webapps备份成webapps\_bak，新建webapps，将webapps\_bak下的host-manager和manager目录拷贝到webapps中，重新运行工程

**问题：Exception starting filter struts2 Unable to load configuration. - [unknown location]**

不用Build Path，将struct包直接导入

**问题：Caused by: Unable to load configuration. - Class: java.net.PlainSocketImpl**

没有联网

**问题：hibernate Table doesn't exist SQLGrammarException: could not execute statement**

<!-- 根据需要自动创建数据库 -->

<property name="hbm2ddl.auto">create</property> //注释该行

**问题：Disabling contextual LOB creation as createClob() method threw error : java.lang.reflect.InvocationTargetException**

mysql-connector-java-5.1.21.jar -> mysql-connector-java-5.1.6.jar

**问题：hibernate报错No Session found for current thread**

这个没配或者配错

<aop:config>

<aop:advisor advice-ref="txAdvice" pointcut="execution(\* \*..\*Dao\*.\*(..))"/>

</aop:config>

# Spring Boot

http://qbgbook.gitbooks.io/spring-boot-reference-guide-zh/content

## 安装spring boot cli

下载压缩包，按照INSTALL.txt安装。

windows：将解压路径添加到PATH中。执行spring --version查看版本。

创建app.groovy文件。

@RestController

class ThisWillActuallyRun {

@RequestMapping("/")

String home() {

"Hello World!"

}

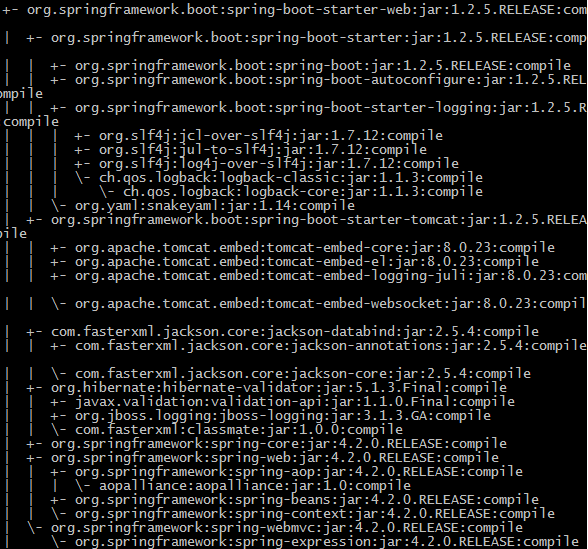
}

spring run app.groovy

访问localhost:8080查看效果。

## spring-boot-starter-web

spring-boot-starter-web，将项目设为动态网站，主要引入spring，spring mvc，jackson，logback。tomcat的包



### Start POMs

Starter是指一站式配置好spring的某个功能。

|  |  |
| --- | --- |
| 名称 | 描述 |
| spring-boot-starter | 核心Spring Boot starter，包括自动配置支持，日志和YAML |
| spring-boot-starter-actuator | 生产准备的特性，用于帮你监控和管理应用 |
| spring-boot-starter-amqp | 对"高级消息队列协议"的支持，通过spring-rabbit实现 |
| spring-boot-starter-aop | 对面向切面编程的支持，包括spring-aop和AspectJ |
| spring-boot-starter-batch | 对Spring Batch的支持，包括HSQLDB数据库 |
| spring-boot-starter-cloud-connectors | 对Spring Cloud Connectors的支持，简化在云平台下（例如，Cloud Foundry 和Heroku）服务的连接 |
| spring-boot-starter-data-elasticsearch | 对Elasticsearch搜索和分析引擎的支持，包括spring-data-elasticsearch |
| spring-boot-starter-data-gemfire | 对GemFire分布式数据存储的支持，包括spring-data-gemfire |
| spring-boot-starter-data-jpa | 对"Java持久化API"的支持，包括spring-data-jpa，spring-orm和Hibernate |
| spring-boot-starter-data-mongodb | 对MongoDB NOSQL数据库的支持，包括spring-data-mongodb |
| spring-boot-starter-data-rest | 对通过REST暴露Spring Data仓库的支持，通过spring-data-rest-webmvc实现 |
| spring-boot-starter-data-solr | 对Apache Solr搜索平台的支持，包括spring-data-solr |
| spring-boot-starter-freemarker | 对FreeMarker模板引擎的支持 |
| spring-boot-starter-groovy-templates | 对Groovy模板引擎的支持 |
| spring-boot-starter-hateoas | 对基于HATEOAS的RESTful服务的支持，通过spring-hateoas实现 |
| spring-boot-starter-hornetq | 对"Java消息服务API"的支持，通过HornetQ实现 |
| spring-boot-starter-integration | 对普通spring-integration模块的支持 |
| spring-boot-starter-jdbc | 对JDBC数据库的支持 |
| spring-boot-starter-jersey | 对Jersey RESTful Web服务框架的支持 |
| spring-boot-starter-jta-atomikos | 对JTA分布式事务的支持，通过Atomikos实现 |
| spring-boot-starter-jta-bitronix | 对JTA分布式事务的支持，通过Bitronix实现 |
| spring-boot-starter-mail | 对javax.mail的支持 |
| spring-boot-starter-mobile | 对spring-mobile的支持 |
| spring-boot-starter-mustache | 对Mustache模板引擎的支持 |
| spring-boot-starter-redis | 对REDIS键值数据存储的支持，包括spring-redis |
| spring-boot-starter-security | 对spring-security的支持 |
| spring-boot-starter-social-facebook | 对spring-social-facebook的支持 |
| spring-boot-starter-social-linkedin | 对spring-social-linkedin的支持 |
| spring-boot-starter-social-twitter | 对spring-social-twitter的支持 |
| spring-boot-starter-test | 对常用测试依赖的支持，包括JUnit, Hamcrest和Mockito，还有spring-test模块 |
| spring-boot-starter-thymeleaf | 对Thymeleaf模板引擎的支持，包括和Spring的集成 |
| spring-boot-starter-velocity | 对Velocity模板引擎的支持 |
| spring-boot-starter-web | 对全栈web开发的支持，包括Tomcat和spring-webmvc |
| spring-boot-starter-websocket | 对WebSocket开发的支持 |
| spring-boot-starter-ws | 对Spring Web服务的支持 |
| spring-boot-starter-actuator | 添加生产准备特性，比如指标和监控 |
| spring-boot-starter-remote-shell | 添加远程ssh shell支持 |
| spring-boot-starter-jetty | 导入Jetty HTTP引擎（作为Tomcat的替代） |
| spring-boot-starter-log4j | 对Log4J日志系统的支持 |
| spring-boot-starter-logging | 导入Spring Boot的默认日志系统（Logback） |
| spring-boot-starter-tomcat | 导入Spring Boot的默认HTTP引擎（Tomcat） |
| spring-boot-starter-undertow | 导入Undertow HTTP引擎（作为Tomcat的替代） |

## spring-boot-maven-plugin

spring-boot-maven-plugin，为项目生成可执行jar包

其原理是通过Manifest中增加Main-Class或<build>配置启动类，或plugin自动扫描包含public static void main(String[] args)的类，将mvn package生成的包根据<package>的内容重新打包成可执行jar或war包

如果父类设为spring-boot-starter-parent，则默认绑定到<executions>的repackage目标上。如果没有设，则需要手动绑定。



original包是由maven生成



如果没有绑定<execution>阶段，则调用mvn spring-boot:repacakge执行。



### 运行

运行包：

java -jar target/myproject-0.0.1-SNAPSHOT.jar

调试包：

java -Xdebug -Xrunjdwp:server=y,transport=dt\_socket,address=8000,suspend=n \

-jar target/myproject-0.0.1-SNAPSHOT.jar

一站式打包至运行：

**mvn spring-boot:run**

## @EnableAutoConfiguration

根据添加的jar依赖猜测如何配置Spring

## @SpringBootApplication

@SpringBootApplication 等价于同时配置了@Configuration, @EnableAutoConfiguration and @ComponentScan的默认值

包含main方法的类必须位于根目录下，才能扫描到所有的注释类

## spring-boot-dependencies

spring-boot-dependencies，不依赖于spring-boot-starter-parent，而又想利用其依赖时，配置它

<dependencyManagement>

<dependencies>

<dependency>

<!-- Import dependency management from Spring Boot -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>1.3.0.BUILD-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

## spring-boot-devtools

spring-boot-devtools，在产品发布时会自动失效，即以java -jar启动时为产品发布。

1. 自动加载新设置的默认属性，比如在application.properties文件中设置spring.thymeleaf.cache后，devtools会自动应用该设置值。
2. 自动重启，当classpath中的文件变动时自动重启项目。

spring.devtools.restart.exclude=static/\*\*,public/\*\*：这些目录发生变化不重启

spring.devtools.restart.additional-paths：除了classpath下的目录，这些目录发生变化重启

spring.devtools.restart.enabled：classpath下的文件发生变化是否重启

spring.devtools.restart.trigger-file：必须修改该文件，才能重启

1. 资源变化时刷新浏览器，Chrome，Firefox，Safari支持该扩展。

spring.devtools.livereload.enabled：资源变化时是否刷新浏览器

1. 以上参数都设置在$HOME目录下的.spring-boot-devtools.properties文件中，对所有Spring Boot应用生效。



<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

## spring-boot-starter-thymeleaf

默认html放在/resources/templates下

## 问题

**问题：NoClassDefFoundError: org/springframework/boot/logging/DeferredLog**

配置devtools出问题

# Spring Web Flow

# lombok

下载lombok的jar包，

放在eclipse安装根目录下

在eclipse.ini文件的最下方添加

-javaagent:lombok.jar

-Xbootclasspath/a:lombok.jar

重启eclipse

在pom中新增

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.16.6</version>

<scope>provided</scope>

</dependency>

新增类User，在头部引入Data注释

import lombok.Data;

@Data

public class User {

private Integer userId;

private String userName;

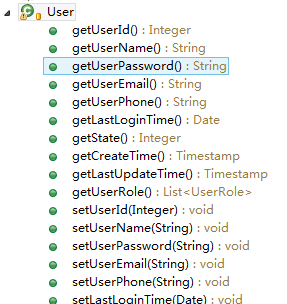
private String userPassword;

private String userEmail;

……

}

在outline页签中可看到以下内容：



@Data   ：注解在类上；提供类所有属性的 getting 和 setting 方法，此外还提供了equals、canEqual、hashCode、toString 方法  
@Setter：注解在属性上；为属性提供 setting 方法  
@Getter：注解在属性上；为属性提供 getting 方法  
@Log4j ：注解在类上；为类提供一个 属性名为log 的 log4j 日志对象  
@NoArgsConstructor：注解在类上；为类提供一个无参的构造方法  
@AllArgsConstructor：注解在类上；为类提供一个全参的构造方法

如果仍然找不到get和set方法，则项目右键->maven->update project

# slj4j & logback

## slj4j

slj4j不是一个真正的日志实现，而是一个抽象层（ [abstraction layer](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html)），它允许你在后台使用任意日志框架，如java.util.logging、log4j、logback。

maven引入

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.12</version>

</dependency>

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Wombat {

final Logger logger = LoggerFactory.getLogger(Wombat.class);

Integer t;

Integer oldT;

public void setTemperature(Integer temperature) {

oldT = t;

t = temperature;

logger.debug("Temperature set to {}. Old temperature was {}.", t, oldT);

if(temperature.intValue() > 50) {

logger.info("Temperature has risen above 50 degrees.");

}

}

}

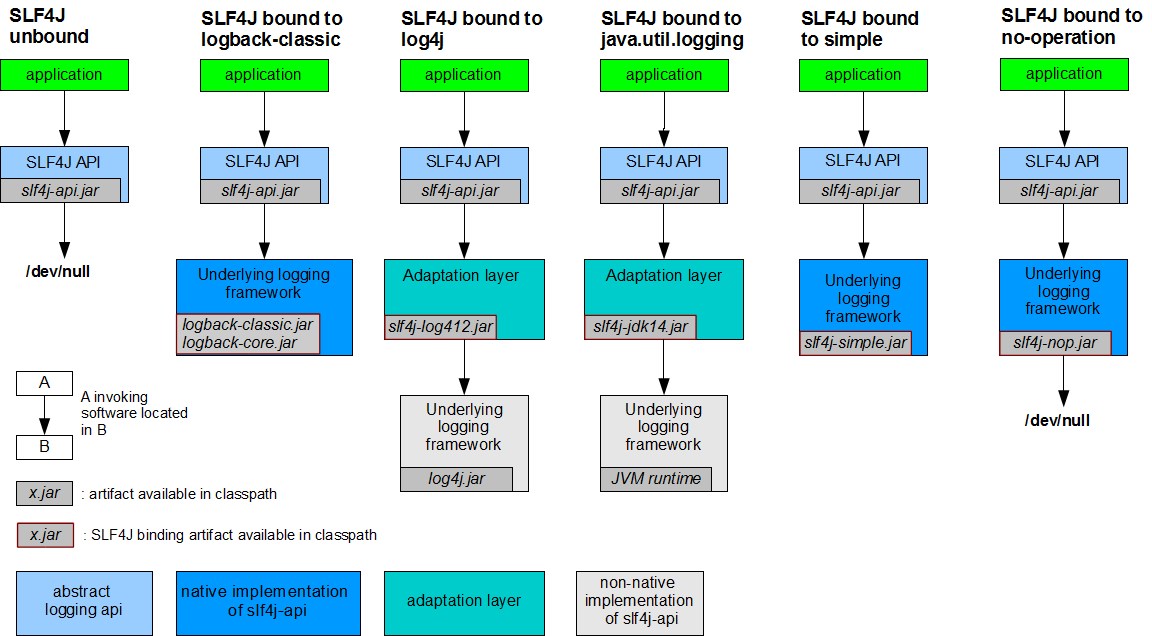
slj4j需要与其他具体日志实现类配合，如果没有，则采用no-operation实现，输出如下：



sljtj推荐日志框架如下：

* slf4j-log4j12-1.7.12.jar：与log4j配合，还需log4j-x.x.x.jar
* slf4j-jdk14-1.7.12.jar：与java.util.logging配合
* slf4j-nop-1.7.12.jar：NOP无操作
* slf4j-simple-1.7.12.jar：所有INFO以上的消息都输出到System.err中
* slf4j-jcl-1.7.12.jar：Jakarta日志
* logback-classic-1.0.13.jar：推荐框架，还需logback-core-x.x.x.jar

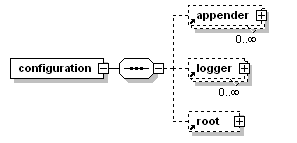
例如从java.util.logging切换到log4j，只用将slf4j-jdk14-1.7.12.jar替换为slf4j-log4j12-1.7.12.jar

[](http://www.slf4j.org/images/concrete-bindings.png)

## logback

spring采用logback输出日志

在resources目录下添加logback.xml



logback三个核心类Logger、Appender、Layout。Logger定义在logback-classic包中，Appender和Layout定义在logback-core中。

1. **logger**

通过包名创建的树结构，如”com.foo”是”com.foo.Bar”的父logger。

root logger是树的根节点。

Logger rootLogger = LoggerFactory.getLogger(org.slf4j.Logger.ROOT\_LOGGER\_NAME);

所有其他节点的logger也可通过LoggerFactory.getLogger获得。

**等级**

package org.slf4j;

public interface Logger {

  // Printing methods:

  public void trace(String message);

  public void debug(String message);

  public void info(String message);

  public void warn(String message);

  public void error(String message);

}

等级定义在ch.qos.logback.classic.Level中

节点如果没有设置等级，则从最近的父节点继承等级。至少会继承根节点的默认等级DEBUG。

|  |  |  |
| --- | --- | --- |
| Logger name | Assigned level | Effective level |
| root | DEBUG | DEBUG |
| X | none | DEBUG |
| X.Y | none | DEBUG |
| X.Y.Z | none | DEBUG |

|  |  |  |
| --- | --- | --- |
| Logger name | Assigned level | Effective level |
| root | ERROR | ERROR |
| X | INFO | INFO |
| X.Y | DEBUG | DEBUG |
| X.Y.Z | WARN | WARN |

|  |  |  |
| --- | --- | --- |
| Logger name | Assigned level | Effective level |
| root | DEBUG | DEBUG |
| X | INFO | INFO |
| X.Y | none | INFO |
| X.Y.Z | ERROR | ERROR |

|  |  |  |
| --- | --- | --- |
| Logger name | Assigned level | Effective level |
| root | DEBUG | DEBUG |
| X | INFO | INFO |
| X.Y | none | INFO |
| X.Y.Z | ERROR | ERROR |

|  |  |  |
| --- | --- | --- |
| Logger name | Assigned level | Effective level |
| root | DEBUG | DEBUG |
| X | INFO | INFO |
| X.Y | none | INFO |
| X.Y.Z | none | INFO |

**比较**

当输出级别大于等于设置级别时，输出生效

级别排序为： TRACE < DEBUG < INFO < WARN < ERROR

ch.qos.logback.classic.Logger logger =

        (ch.qos.logback.classic.Logger) LoggerFactory.getLogger("com.foo");

logger.setLevel(Level. INFO);

logger.warn("Low fuel level.");  输出

logger.debug("Starting search for nearest gas station."); 不输出

Logger barlogger = LoggerFactory.getLogger("com.foo.Bar");

barlogger.info("Located nearest gas station."); 输出

barlogger.debug("Exiting gas station search"); 不输出

1. **Appender**

用于指定日志输出的目的地，目的地可以是控制台、文件、远程套接字服务器、 MySQL、 PostreSQL、 Oracle和其他数据库等。

一个Logger可绑定多个Appender。

1. **Layout**

格式化输出成字符串，至日志中

# Spring Security

## Getting Start

samples目录下，以-jc结尾的是采用java annotation配置的项目，其他则是xml配置的项目。

### 添加依赖

<dependencies>

<!-- ... other dependency elements ... -->

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-web</artifactId>

<version>3.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-config</artifactId>

<version>3.2.5.RELEASE</version>

</dependency>

</dependencies>

spring-security-core.jar：提供核心认证、接入控制的类和接口，远程支持，基础API，必须包含。

spring-security-remoting.jar：与Spring Remoting集成，开发远程客户端时导入。

spring-security-web.jar：web-security架构，Servlet API依赖，需要web认证服务和基于URL的接入控制时导入。

spring-security-config.jar：命名控件解析代码，如果使用XML配置需导入。

spring-security-ldap.jar：LDAP认证代码。

spring-security-acl.jar：域对象ACL实现代码。

spring-security-cas.jar：CAS客户端集成，采用CAS single sign-on server时导入。

spring-security-openid.jar：OpenID web认证。

### web.xml

<filter>

<filter-name>springSecurityFilterChain</filter-name> <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

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

</filter-mapping>

### 配置参数

#### XML配置

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

<beans:beans xmlns=*"http://www.springframework.org/schema/security"*

xmlns:beans=*"http://www.springframework.org/schema/beans"*

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/security*

*http://www.springframework.org/schema/security/spring-security-3.2.xsd"*>

<http auto-config=*"true"*>

<intercept-url pattern=*"/user/\*\*"* access=*"ROLE\_USER"* />

<intercept-url pattern=*"/admin/\*\*"* access=*"ROLE\_ADMIN"* />

<intercept-url pattern=*"/monitor/\*\*"* access=*"ROLE\_DB"* />

<intercept-url pattern=*"/panorama/\*\*"* access=*"ROLE\_LEADER"* />

</http>

<authentication-manager>

<authentication-provider>

<user-service>

<user name=*"mkyong"* password=*"123456"* authorities=*"ROLE\_USER,ROLE\_ADMIN,ROLE\_DB,ROLE\_LEADER"* />

</user-service>

</authentication-provider>

</authentication-manager>

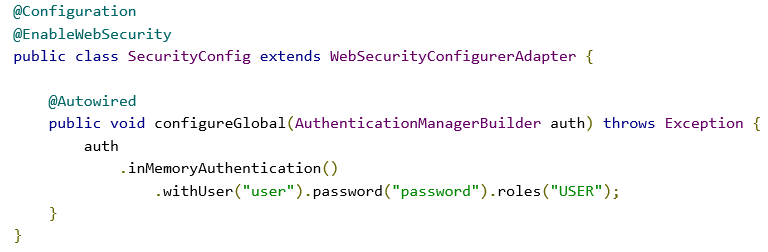
</beans:beans>

#### Java配置

##### SecurityConfig

首先提供Spring Security的Java 配置类，该类继承自WebSecurityConfigureAdapter，经扫描后生成springSecurityFilterChain，用于过滤URL，配置login form，验证用户名密码等。

1. 标注@EnableWebSecurity，@EnableWebMvcSecurity,@EnableGlobalMethodSecurity或@EnableGlobalAuthentication
2. 在方法中导入AuthenticationManagerBuilder



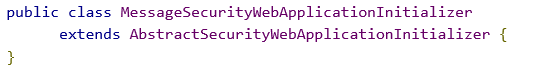
SecurityConfig：

1. 对所有URL请求设置认证；
2. 生成login form；
3. 为form基础认证设置用户名密码；
4. 允许用户logout；
5. CSRF攻击防护；
6. 与HttpServletRequest集成，可通过jsp标签输出用户名等；

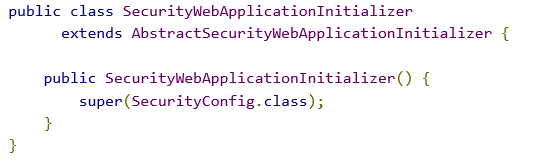
##### AbstractSecurityWebApplicationInitializer

其次让springSecurityFilterChain在war中被注册，MessageSecurityWebApplicationInitializer继承自AbstractSecurityWebApplicationInitializer，AbstractSecurityWebApplicationInitializer实现WebApplicationIntializer接口，保证在web启动时被扫描到。

是否采用该方式取决于SpringSecurity是何种方式使用，在Spring MVC中还是单独使用。



如果没有使用Spring或Spring MVC，则需要SecurityConfig传给父类。



## 问题

**问题1：permitAll不识别**

use-expressions=”true” 使配置支持SpEL

**问题2：**

<http pattern="/resources/\*\*" security="none"/>

<http auto-config=”true” use-expressions=”true”>

<intercept-url pattern=”/login” access=”permitAll”/>

<intercept-url pattern=”/static/\*\*” filters=”none”/>

</http>

access=”permitAll”是指所有请求都被允许，需要经过filter

filters=”none”是指完全不经过filter，已废弃

security=”none”，用于代替filters=”none”，完全不经过filter

新版本中支持多个http标签

<http pattern="/resources/\*\*" security="none"/>代替<intercept-url pattern="/resources/\*\*" filters="none"/>

应用场景：如果需要获取当前登陆的用户名，则必须经过filter，用permitAll，对静态资源用security=”none”

**问题3：匿名访问**

access=’IS\_AUTHENTICATED\_ANONYMOUSLY’

# Quartz

Scheduler：与scheduler交互API；由SchedulerFactory创建开始到执行shutdown()结束。可用于增加、删除Job和Trigger。从调用start()开始执行Trigger。

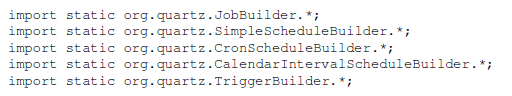
Job：scheduler执行的接口；

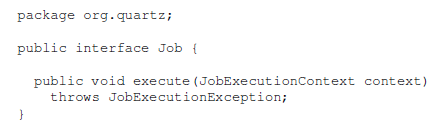
JobDetail：定义Job实例；

Trigger：定义Job如何执行的组件；

JobBuilder：生成和定义JobDetail实例；

TriggerBuilder：生成和定义Trigger实例。





# Sitemesh 3

1. 添加sitemesh 3依赖

<!-- sitemesh -->

<dependency>

<groupId>org.sitemesh</groupId>

<artifactId>sitemesh</artifactId>

<version>${sitemesh.version}</version>

</dependency>

1. web.xml中添加filter

<!-- sitemesh配置 -->

<filter>

<filter-name>sitemesh</filter-name>

<filter-class>org.sitemesh.config.ConfigurableSiteMeshFilter</filter-class>

</filter>

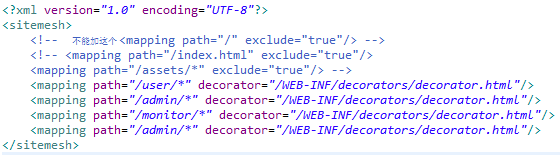
<filter-mapping>

<filter-name>sitemesh</filter-name>

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

</filter-mapping>

1. 配置/WEB-INF/sitemesh3.xml



注意 exclude的拼写 ， 不要随便用”/,/\*,\*\*,\*”通配符，优先级过高导致覆盖掉其他配置

filter拦截请求，交给Content Processer，用Decorator合并。

Decorator通过HTML定义（此处为default.html）包含<sitemesh:write>标签用于指明在何处合并内容。

1. 编写Decorator

<html>

<head>

<title>SiteMesh example: <sitemesh:write property=*'title'*/></title>

<style type=*'text/css'*>

/\* Some CSS \*/

**body** { font-family: *arial, sans-serif*; background-color: *#ffffcc*; }

**h1,** **h2,** **h3,** **h4** { text-align: *center*; background-color: *#ccffcc*;

border-top: *1px solid #66ff66*; }

*.mainBody* { padding: *10px*; border: *1px solid #555555*; }

*.disclaimer* { text-align: *center*; border-top: *1px solid #cccccc*;

margin-top: *40px*; color: *#666666*; font-size: *smaller*; }

</style>

<sitemesh:write property=*'head'*/>

</head>

<body>

<h1 class=*'title'*>SiteMesh example site: <sitemesh:write property=*'title'*/></h1>

<div class=*'mainBody'*>

<sitemesh:write property=*'body'*/>

</div>

<div class=*'disclaimer'*>Site disclaimer. This is an example.</div>

</body>

</html>

1. 编写html

<html>

<head>

<title>Hello World</title>

<meta name=*'description'* content=*'A simple page'*>

</head>

<body>

<p>Hello <strong>world</strong>!</p>

</body>

</html>

# Thymeleaf

Thymeleaf是Java Library，XML/XHTML/HTML5模版引擎，用于在template file中展示服务器返回的数据或文本。

## 基础知识

### 初始化

**public** **class** GTVGApplication {

**private** **static** TemplateEngine templateEngine;

**static** {

...

initializeTemplateEngine();

...

}

//用于配置和返回TemplateEngine

**private** **static** **void** initializeTemplateEngine() {

//Template Resolver用于配置参数

ServletContextTemplateResolver templateResolver =

**new** ServletContextTemplateResolver();

//默认值为 XHTML

templateResolver.setTemplateMode("XHTML");

// 设置视图前后缀，例如 "home" 变为"/WEB-INF/templates/home.html"

templateResolver.setPrefix("/WEB-INF/templates/");

templateResolver.setSuffix(".html");

// Template 缓存为 TTL=1h. 如果没有设置，实体通过LRU策略清除

templateResolver.setCacheTTLMs(3600000L);

//最后将配置好的Template Resolver设置到TemplateEngine中

templateEngine = **new** TemplateEngine();

templateEngine.setTemplateResolver(templateResolver);

}

...

}

**private** **boolean** process(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException {

**try** {

/\*

\* 查询controller/URL的映射关系，返回对应的controller，处理请求

\*如果没有可用的controller，则返回false，让其他filters/servlets处理请求

\*/

IGTVGController controller = GTVGApplication.resolveControllerForRequest(request);

**if** (controller == **null**) {

**return** **false**;

}

/\*

\*获得模板引擎TemplateEngine

\*/

TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();

/\*

\* 配置消息响应头

\*/

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

response.setHeader("Pragma", "no-cache");

response.setHeader("Cache-Control", "no-cache");

response.setDateHeader("Expires", 0);

/\*

\* 执行控制器，处理视图，将其写入消息响应

\*/

controller.process(

request, response, **this**.servletContext, templateEngine);

**return** **true**;

} **catch** (Exception e) {

**throw** **new** ServletException(e);

}

}

**public** **interface** IGTVGController {

**public** **void** process(

HttpServletRequest request, HttpServletResponse response,

ServletContext servletContext, TemplateEngine templateEngine);

}

**public** **class** HomeController **implements** IGTVGController {

**public** **void** process(

HttpServletRequest request, HttpServletResponse response,

ServletContext servletContext, TemplateEngine templateEngine) {

/\*

\*创建WebContent，该content继承自IContext接口，里面包含Template Engine所需的全部数据，以及本地化信息

\*/

WebContext ctx =

**new** WebContext(request, response, servletContext, request.getLocale());

//最后，通过模板名称、WebContent、response writer调用Template Engine处理请求

templateEngine.process("home", ctx, response.getWriter());

}

}

**public** **interface** IContext {

**public** VariablesMap<String,Object> getVariables();

**public** Locale getLocale();

...

}

//IWebContext继承自IContext，提供处理所需的全部参数

**public** **interface** IWebContext **extends** IContext {

**public** HttpSerlvetRequest getHttpServletRequest(); //提供HttpServletRequest

**public** HttpSession getHttpSession(); //提供Session

**public** ServletContext getServletContext(); //提供ServletContext

**public** VariablesMap<String,String[]> getRequestParameters(); //提供请求参数

**public** VariablesMap<String,Object> getRequestAttributes(); //提供请求属性

**public** VariablesMap<String,Object> getSessionAttributes(); //提供session属性

**public** VariablesMap<String,Object> getApplicationAttributes(); //提供应用属性

}

另外execInfo包含Context和WebContext，以及特殊参数，如${execInfo.templateName} ${execInfo.now}

### HTML配置

在html前加入该内容

<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:th="http://www.thymeleaf.org">

### 表达式

Simple expressions:

* Variable Expressions: ${...} 变量表达式，从当前context包含的变量中获得值显示
* Selection Variable Expressions: \*{...}，变量表达式，区别是\*针对选中的对象，而不是整个context中获得值
* Message Expressions: #{...} 消息表达式，从本地的Messages\_local.properties文件中提取相应信息显示
* Link URL Expressions: @{...}

### 基本对象

* #ctx: the context object.
* #vars: the context variables.
* #locale: the context locale.
* #httpServletRequest: (only in Web Contexts) the HttpServletRequest object.
* #httpSession: (only in Web Contexts) the HttpSession object.
* #dates: utility methods for java.util.Date objects: formatting, component extraction, etc.
* #calendars: analogous to #dates, but for java.util.Calendar objects.
* #numbers: utility methods for formatting numeric objects.
* #strings: utility methods for String objects: contains, startsWith, prepending/appending, etc.
* #objects: utility methods for objects in general.
* #bools: utility methods for boolean evaluation.
* #arrays: utility methods for arrays.
* #lists: utility methods for lists.
* #sets: utility methods for sets.
* #maps: utility methods for maps.
* #aggregates: utility methods for creating aggregates on arrays or collections.
* #messages: utility methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using #{…} syntax.
* #ids: utility methods for dealing with id attributes that might be repeated (for example, as a result of an iteration).

### 路径选择

* Absolute URLs, like http://www.thymeleaf.org
* Relative URLs, which can be:
* Page-relative, like user/login.html
* Context-relative, like /itemdetails?id=3 (context name in server will be added automatically)
* Server-relative, like ~/billing/processInvoice (allows calling URLs in another context (= application) in the same server.
* Protocol-relative URLs, like //code.jquery.com/jquery-2.0.3.min.js

### th:each

<table>

<tr>

<th>NAME</th>

<th>PRICE</th>

<th>IN STOCK</th>

</tr>

<tr th:each="prod,iterStat : ${prods}" th:class=”${iterStat.odd}?’odd’”>

<td th:text="${prod.name}">Onions</td>

<td th:text="${prod.price}">2.41</td>

<td th:text="${prod.inStock}? #{true} : #{false}">yes</td>

</tr>

</table>

iterStat必须在prod之后申明，以逗号分隔，（如果未申明，则默认为其后加Stat，即prodStat）

其属性包括index（初始为0的索引)，count（初始为1的索引），size（总数），current（当前变量），even/odd（当前是奇数/偶数），first/last（是否最后一个）

### if/else

<a href="comments.html"

th:href="@{/product/comments(prodId=${prod.id})}"

th:if="${not #lists.isEmpty(prod.comments)}">view</a>

if非空条件，boolean为true，number非0，character非’0’，String非false|no|off。

th:if=”${not }” 等价于 th:unless=”${}”

### switch

<div th:switch="${user.role}">

<p th:case="'admin'">User is an administrator</p>

<p th:case="#{roles.manager}">User is a manager</p>

<p th:case="\*">this is default</p>

</div>

### Template Layout

footer.html:

<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:th="http://www.thymeleaf.org">

<body>

<div th:fragment="copy">

&copy; 2011 The Good Thymes Virtual Grocery

</div>

</body>

</html>

th:fragment定义片段，th:include和th:replace引入该片段

<div th:include="footer :: copy"></div>

格式为templatename::domselector

::domselector和this::domselector为引入同一模板

#domselector通过ID选择，不需要fragment

<div th:include="footer :: (${user.isAdmin}? #{footer.admin} : #{footer.normaluser})"></div>

th:include取代内容，th:replace取代原有标签。

### 局部变量

在申明其的标签内可见

<div th:with="firstPer=${persons[0]},secondPer=${persons[1]}">

<p>

The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.

<span th:text="${secondPer.name}">

</p>

</div>

### 属性优先级

|  |  |  |
| --- | --- | --- |
| Order | Feature | Attributes |
| 1 | Fragment inclusion | th:include th:replace |
| 2 | Fragment iteration | th:each |
| 3 | Conditional evaluation | th:if th:unless th:switch th:case |
| 4 | Local variable definition | th:object th:with |
| 5 | General attribute modification | th:attr th:attrprepend th:attrappend |
| 6 | Specific attribute modification | th:value, th:href, th:src, etc. |
| 7 | Text (tag body modification) | th:text th:utext |
| 8 | Fragment specification | th:fragment |
| 9 | Fragment removal | th:remove |

## 与SpringMVC整合

<bean id="templateResolver"

class="org.thymeleaf.templateresolver.ServletContextTemplateResolver">

<property name="prefix" value="/WEB-INF/templates/" />

<property name="suffix" value=".html" />

<property name="templateMode" value="HTML5" />

</bean>

<bean id="templateEngine"

class="org.thymeleaf.spring4.SpringTemplateEngine">

<property name="templateResolver" ref="templateResolver" />

<property name="characterEncoding" value="UTF-8" />

</bean>

**问题1：decorate.html不被thymeleaf解析**

JSP与sitemesh搭配，Thymeleaf与thymeleaf-layout-dialect搭配

thymeleaf与sitemesh搭配，请求先经过thymeleaf解析，再经过sitemesh装饰，导致decorate.html中无法利用thymeleaf标签获得参数，从而无法显示当前返回信息。

解决方法:decorate.html采用jsp解析

# Django

## 安装

windows：

安装python：下载msi，运行，环境变量Path中添加C:\Python34\;C:\Python34\Scripts;

cmd中执行python --version

安装setuptools：下载ez\_setup.py文件，在命令行中运行C:>ez\_setup.py。

安装PIP：Python 3.4包含pip，可跳过，否则执行easy\_install pip。

安装Django：运行pip install django，运行django-admin --version验证。

查看版本

python -c "import django; print(django.get\_version())"

查看安装目录

python -c “import sys;sys.path=sys.path[1:];import django;print(django.\_\_path\_\_)”

## 使用

django-admin startproject mysite

mysite/ 项目容器

manage.py ：与Django项目交互的命令行工具

mysite/ ：项目包名，如mysite.urls

\_\_init\_\_.py ：空文件告诉Python这是Python包

settings.py：配置文件

urls.py：URL与视图（views.py）映射

models.py：数据表模型

views.py：业务逻辑

wsgi.py：WSGI web服务器的entry-point

settings.py

配置数据库：

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': 'mydatabase',

'USER': 'mydatabaseuser',

'PASSWORD': 'mypassword',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

ENGINE：可选值，'django.db.backends.sqlite3', 'django.db.backends.postgresql\_psycopg2', 'django.db.backends.mysql', or 'django.db.backends.oracle'。

NAME：数据库名称，如果用SQLite，则是文件绝对路径，其他

INSTALLED\_APPS：包含常用app

python manage.py migrate 根据INSTALLED\_APPS和DATABASES设置创建所需的数据库表

python manage.py runserver

python manage.py startapp polls：创建polls模型

python manage.py makemigrations polls：创建polls模型的数据库表

python manage.py sqlmigrate polls 0001：打印sql语句

python manage.py check：检查项目

python manage.py migrate：迁移应用模型

修改模型三个步骤：

* 修改models.py
* 执行python manage.py makemigrations
* 执行python manage.py migrate

python manage.py shell：启动shell，设置DJANGO\_SETTINGS\_MODULE环境变量

python manage.py createsuperuser：创建管理员