#### 第八章

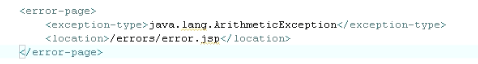
##### jsp（java server page）

jsp注释：<%-- --%>,这个注释的代码不回出现在被编译后的servlet中。

html注释：<!-- -->:，这个注释会出现在servlet中。

###### page标签：

使用JSP的errorpage=“/\*\*\*.html”(在单独的jsp页面中定义错误返回页)，要数据量大于缓存，默认jsp页面有8k的缓存，因此写测试的时候可能不回出现结果，只对单个页面有效。可用在web.xml文件中配置，针对全工程：





jsp文件中的配置优先级更高

2.page标签属性 isErrorPage = true/false 默认false，在你的错误页面中标识为true，服务器将把异常对象传递过来

3.服务器传递页面9大对象：



还有一个就是2中的exception。

（Application、config、exception、out、page、pageContext、request、response、session）

###### include标签：

静态包含，运行效率更高，更常用（编译时包含，只翻译成一个jsp）。

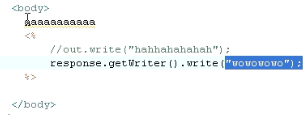
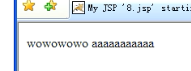
可以在jsp中假如<% request.getRequestDispatcher(“/相对服务器的路径”).include(request,response); %>（运行时包含，会翻译成多个jsp）

###### 建议使用out输出

out输出时也是交给response再一起输出。，因此在

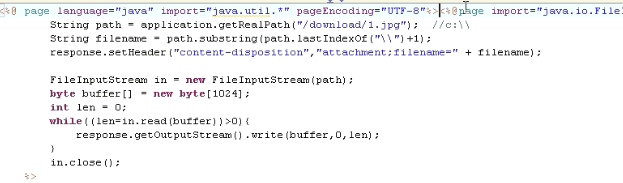
会先输出wowo再输出hah，因为缓冲。

如下输出？：

输出为：jsp都是转为servlet输出，都是使用out输出。

###### out输出的字符

使用jsp做下载：（设置response头：content-disposition：attachment；filename=‘下载文件名’）



不能出现html语句，否则会报错：使用是同时字符流、字节流输出。

###### pageContext，pageContext封装了8大对象。

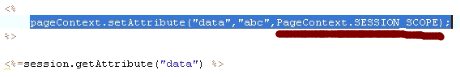
1. 有自己的域对象，但是只在当前页面有效，称为page域；

request在同一个请求中有效，请求范围内共享，请求完就没用了；

session在同一个回话中有效，莫个人共享；

application在整个网站能取出，所有人可以共享

1. pageContext往某个域里面存，sun提供统一的web操作域的接口：

框架里面用的特别多，但是一般不会像上面这样取，如下取值用的较多：



1. 还有一个findAttrbute()方法，依次查找：

这四域，这个是el的取数据原理，如：${data}就是调用上面这个方法。

1. 还有如下功能：（会在浏览器的地址栏重写url）



上面两句代码执行的是一个效果。下面使用了pageContext简化开发。还有include()方法等。

###### 总结：jsp九大内置对象。

Application、session、page、config、pageContext、request、response、out、exception

###### jsp常用内置内置标签

1. <jsp:include>

可以接收表达式：

但是是动态包含。

include标签和指令的区别，前者动态，后者静态，静态更快。

如：

1. <jsp:forward>

如：

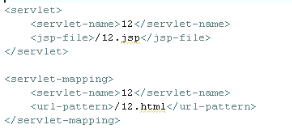
1. <jsp:param>

用于forward的时候可以带数据过去。如：



取的话：request.getParameter(“username”);

###### 10.jsp映射



在web.xml文件中有如上配置，那么访问12.html的时候就访问12.jsp了，这样 就可以任意地址配置对对应的jsp了，同时你也能直接访问12.jsp页面。

#### 第九章

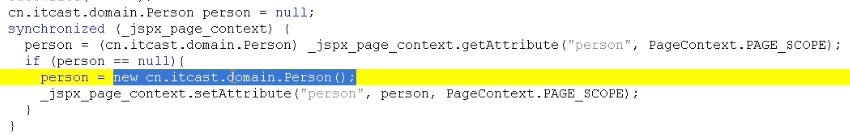
##### jsp中使用javabean

###### <jsp:userBean>



scop:javabean存放的域。

工作原理（查看编译后的servlet）：

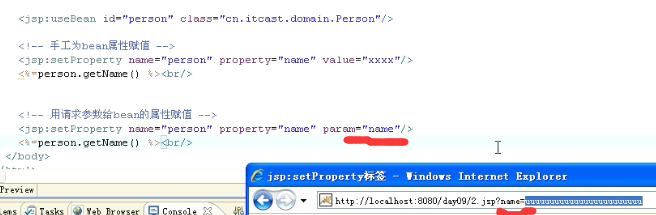


注意：



以上这个代码只会在第一次回话时运行，后面就不回运行。因为是session域。存到session域中如果有这个bean就会直接获取，没有才会new一个javabean 。

###### <jsp:setProperty>



param可以将8种基本类型进行转换，赋给javabean。

如果日期转，则需要写表达式（假如brithday是date类型，下面的url日期访问时，把value替换城param=“birthday”，会报错）：



还有写法：



但是要注意属性名字要一样。

3.<jsp:getProperty>



输出：

上面null是因为javabean中的birthday属性没有值，所以会输出null。在实际开发过程中，这个是不允许的，所以还需要判断。也基于这一点，所以在开发中使用EL表达式，EL表达式如果碰到是null，则输出的是空字符串。

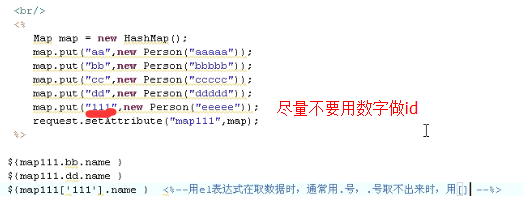
1. jsp+javaBean以及javaBean(M)+jsp(V)+servlet(C)（mvc）设计模式

##### EL表达式：

###### 获取简单对象



###### 获取集合



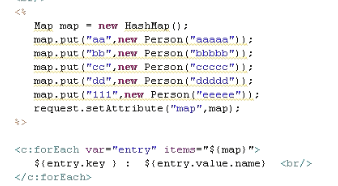
###### 获取路径 $(pageContext.request.contextPath)



注意这里获取web路径，不要在el表达式前面加“/”，因为默认有。

##### JSTL（以下代码实例的prefix=“c”）

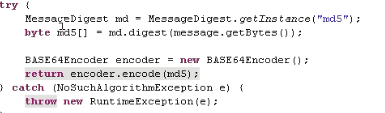
1. 使用jstl需要引入2个jar包：jstl.jar以及standerd.jar(使用el表达式也要引入standerd.jar)；在jsp页面中需要使用<%@taglib url=”” prefix=”” %>元素导入标签库。
2. 在jstl.jar中的META-INF下有c.tld文件，该文件中有对应taglib中的url地址。
3. 取出map集合中的数据，默认是调用map.entrySet()方法：



1. if标签，判断用户是否登录可以使用如下：

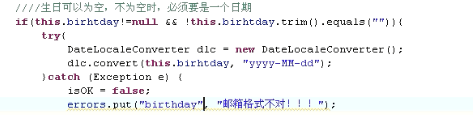


##### md5转换：



md5[]数组不能直接使用new String()的构造方法，因为它是二进制流，会乱码，要先编码，或者用十六进制编码。（md5编码是不可逆过程。base64编码为了能让编码后md5码显示成正常的可是别string）

##### 日期的判断使用beanutils.jar



##### 泛型方法



这里使用了BeanUtils.setProperty()方法，也可以使用反射

##### 图片认证码

<td>

<input class="userinput" type="text" name="client\_checkcode">

<img src="${pageContext.request.contextPath }/servlet/ImageServlet" onclick="this.src=this.src + '?' + new Date().getTime()"

height="23px" width="120px" alt="看不清" style="cursor: hand;">

</td>

上面代码可以达到点击图片验证码时刷新图片效果，但是多次点击会出现异常，因为每次this.scr的路径都是上次提交的路径加上日期，多次点击会出现长度过长，超出1k大小，就不能提交，因此这里需要每次截取一定长度。（如果onclick="this.src=this.src”则不会出现刷新，因为提交给servlet的路径都是相同，servlet会直接拿上次产生的）

也可以使用ajax打到这种局部刷新效果。

#### 第十章（购物案例，未看）

#### 第十一章

##### 自定义标签JSPTag

以下是jsp1.0自定义标签（传统标签）

###### jsp中规范是禁止出现java代码

实现步骤：

第一步，实现Tag接口，还有继承TagSupport对象，标签处理器类；

public class ViewIPTag **extends** TagSupport {

@Override

public int doStartTag() throws JspException {

//服务器会去调用serPageContex方法获得jsp的9大对象

HttpServletRequest request = (HttpServletRequest) this.pageContext.getRequest();

JspWriter out = this.pageContext.getOut();

String ip = request.getRemoteAddr();

try {

out.print(ip);

} catch (IOException e) {

throw new RuntimeException(e);

}

return super.doStartTag();

}

}

第二步，tld文件申明标签（一般放在web-inf下）；标签文件如下（可以在tomcat中的example列子中web-inf目录下找到tld文件）：

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

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation=[http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary\_2\_0.xsd](http://java.sun.com/xml/ns/j2ee%20http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd) version="2.0">

<description>A tag library exercising SimpleTag handlers.</description>

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

<short-name>ljh</short-name>

<uri>http://www.ljh.cn</uri> <!-- 这里的url地址，是你在jsp页面中需要引入的使用taglib -->

<tag>

<name>viewIP</name>

<tag-class>cn.ljh.web.tag.ViewIPTag</tag-class>

<body-content>empty</body-content><!-- empth表示是可以这样使用：<ljh/> -->

</tag>

<tag>

<name>demo1</name>

<tag-class>cn.ljh.web.tag.TagDemo1</tag-class>

<body-content>JSP</body-content><!—标签体是jsp（JSP要大写）共四种EMPTY JSP scriptless tagdepentend -- >

</tag>

</taglib>

第三步：在jsp页面中导入和使用自定义标签

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

<%@taglib uri="http://www.ljh.cn" prefix="test" %>

<!--也可以使用uri=”WEB-INF/ljh.tld” 引入标签说明文件路径 -->

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

<html>

<head>

<title>My JSP '1.jsp' starting page</title>

</head>

<body>

您的ip是：<test:viewIP/>

</body>

</html>

###### 控制jsp页面某一部分内容是否执行

public class TagDemo1 extends TagSupport {

@Override

public int doStartTag() throws JspException {

return Tag.SKIP\_BODY;

}

}

<tag>

<name>demo1</name>

<tag-class>cn.itcast.web.tag.TagDemo1</tag-class>

<body-content>JSP</body-content>

</tag>

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"http://www.itcast.cn"* prefix=*"itcast"* %>

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

<html>

<head>

<title>使用标签控制页面内容(标签体)是否输出</title>

</head>

<body>

<itcast:demo1>

aaaaa

</itcast:demo1>

</body>

</html>

###### 控制整个jsp页面是否执行

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"http://www.itcast.cn"* prefix=*"itcast"* %>

<itcast:demo2/>

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

<html>

<head>

<title>用标签控制整个jsp是否输出</title>

</head>

<body>

This is my JSP page. <br>

sdfa

</body>

</html>

/////----------------------------------------

**public** **class** TagDemo2 **extends** TagSupport {

@Override

**public** **int** doEndTag() **throws** JspException {

**return** Tag.*EVAL\_PAGE*; //继续执行结束标签之后的page

}

}

/////--------------------------------

<tag>

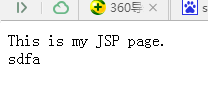
<name>demo2</name>

<tag-class>cn.itcast.web.tag.TagDemo2</tag-class>

<body-content>empty</body-content>

</tag>

结果：



案例二：

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"http://www.itcast.cn"* prefix=*"itcast"* %>

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

<html>

<head>

<title>用标签控制整个jsp是否输出</title>

</head>

<body>

<itcast:demo2>

This is my JSP page. <br>

</itcast:demo2>

sdfa

</body>

</html>/////----------------------------------------

**public** **class** TagDemo2 **extends** TagSupport {

@Override

**public** **int** doEndTag() **throws** JspException {

**return** Tag.*EVAL\_PAGE*; //继续执行结束标签之后的page

}

}

/////--------------------------------

<tag>

<name>demo2</name>

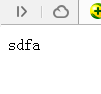
<tag-class>cn.itcast.web.tag.TagDemo2</tag-class>

<!-- <body-content>empty</body-content> -->

<body-content>JSP</body-content>

</tag>

结果：



###### 控制jsp页面内容重复执行

需要实现IteratorTag类的doAfterBody();这个方法是在标签体执行完，结束标签执行前执行。但是TagSupport也实现了IteratorTag类。

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"http://www.itcast.cn"* prefix=*"itcast"* %>

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

<html>

<head>

<title>用标签控制标签体重复执行</title>

</head>

<body>

<itcast:demo3>

This is my JSP page. <br>

</itcast:demo3>

</body>

</html>

//////---------------------------------------------

//控制标签体执行5次

**public** **class** TagDemo3 **extends** TagSupport {

**int** x = 5;

@Override

**public** **int** doStartTag() **throws** JspException {

// **TODO** Auto-generated method stub

**return** Tag.*EVAL\_BODY\_INCLUDE*;

}

@Override

**public** **int** doAfterBody() **throws** JspException {

x--;

**if**(x>0){

**return** IterationTag.*EVAL\_BODY\_AGAIN*;

}**else**{

**return** IterationTag.*SKIP\_BODY*;

}

}

}

----------------------

<tag>

<name>demo3</name>

<tag-class>cn.itcast.web.tag.TagDemo3</tag-class>

<body-content>JSP</body-content>

</tag>

###### 修改jsp页面内容输出

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"http://www.itcast.cn"* prefix=*"itcast"* %>

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

<html>

<head>

<title>用标签修改jsp页面内容</title>

</head>

<body>

<itcast:demo4>

aaaaaa

</itcast:demo4>

</body>

</html>//////---------------------------------------------\

//修改标签体(把标签体改为大写)

**public** **class** TagDemo4 **extends** BodyTagSupport {

@Override

**public** **int** doStartTag() **throws** JspException {

// **TODO** Auto-generated method stub

**return** BodyTag.*EVAL\_BODY\_BUFFERED*;

}

@Override

**public** **int** doEndTag() **throws** JspException {

BodyContent bc = **this**.getBodyContent(); //得到标签体

String content = bc.getString();

content = content.toUpperCase();

**try** {

**this**.pageContext.getOut().write(content);

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

**return** Tag.*EVAL\_PAGE*;

}}

----------------------

<tag>

<name>demo4</name>

<tag-class>cn.itcast.web.tag.TagDemo4</tag-class>

<body-content>JSP</body-content>

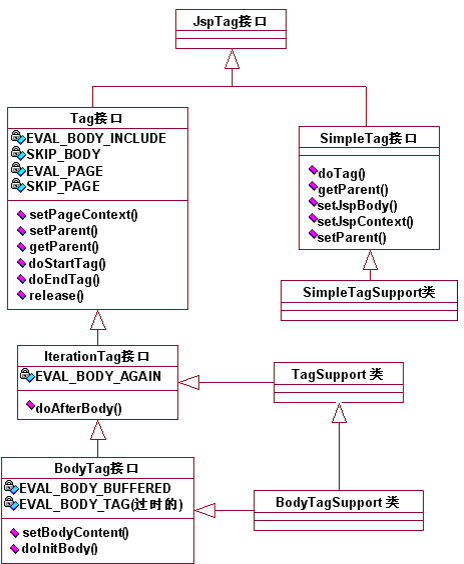
</tag>

###### tld文件中的四种标签体类型

* + EMPTY JSP scriptless tagdepentend（基本上不用，标签内容只是给标签用，不做JSP输出，比如定义一个标签里面存放sql语句）

###### ------------以上是jsp2.0以前技术(左边)，

JSP2.0以后使用SimpleTag接口（右边→是jsp2.0）,术语：实现左边的标签叫“传统标签”，实现右边叫“简单标签”



##### 简单标签SimpleTag

接口SimpleTag里面没有开始结束标签之分，服务器加载时会调用setJspContext()方法。只要实现SimpleTagSupport，再重写doTag()方法；

简单标签的类型scriptless（无脚本）

###### JspFragment

JspFragment. invoke(this.getJspContext.getOut());或者JspFragment. invoke(null)，null默认就是out对象

###### 修改标签体案例：

//修改标签体

**public** **class** SimpleTagDemo3 **extends** SimpleTagSupport {

@Override

**public** **void** doTag() **throws** JspException, IOException {

JspFragment jf = **this**.getJspBody();

StringWriter sw = **new** StringWriter(); //相当于一个缓冲String

jf.invoke(sw); //如果invoke的参数out不为空的话，那么可以使用这个out参数得到tag的body的内容。然后使用getJspContext().getOut()将上面通过out得到的内容处理完成之后，输出到页面中。

String content = sw.toString();

content = content.toUpperCase();

**this**.getJspContext().getOut().write(content);

}

}

////-------------------

<tag>

<name>demo3</name>

<tag-class>cn.itcast.web.simpletag.SimpleTagDemo3</tag-class>

<body-content>scriptless</body-content>

</tag>

//////---------------------------

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"/simpletag"* prefix=*"sitcast"* %>

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

<html>

<head>

<title>修改标签体</title>

</head>

<body>

<sitcast:demo3>

aaaaaaa

</sitcast:demo3>

</body>

</html>

###### 带属性标签

**public** **class** SimpleTagDemo5 **extends** SimpleTagSupport {

**private** **int** count; **private** Date date;

**public** **void** setCount(**int** count) {

**this**.count = count;

}

**public** **void** setDate(Date date) {

**this**.date = date;

}

**public** **void** doTag() **throws** JspException, IOException {

JspFragment jf = **this**.getJspBody();

**this**.getJspContext().getOut().write(date.~~toLocaleString~~() + "<br/>");

**for**(**int** i=0;i<count;i++){

jf.invoke(**null**);

}

}}

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

<taglib xmlns=*"http://java.sun.com/xml/ns/j2ee"*

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

xsi:schemaLocation=*"http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary\_2\_0.xsd"*

version=*"2.0"*>

<description>A tag library exercising SimpleTag handlers.</description>

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

<short-name>itcast</short-name>

<uri>/simpletag</uri>

<tag>

<name>demo5</name>

<tag-class>cn.itcast.web.simpletag.SimpleTagDemo5</tag-class>

<body-content>scriptless</body-content>

<attribute>

<name>count</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

<attribute>

<name>date</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

</taglib>

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>

<%@taglib uri=*"/simpletag"* prefix=*"sitcast"* %>

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

<html>

<head>

<title>开发带属性的标签</title>

</head>

<body>

<sitcast:demo5 count=*"3"* date=*"*<%=new Date() %>*"*>

aaaaaa

</sitcast:demo5>

</body>

</html>

###### 总结（注意）

**简单标签**执行流程：

1.jsp引擎首先创建类对象

2.调用setPageContext方法，把页面对象传进去

3.调用setParent()方法；

4.调用setJSPBody()方法；

5.调用doTag()方法；

6.传统标签执行完成不会直接调用release()而是驻留在内存为后续调用使用，简单标签执行完会摧毁。

###### 开发带属性的标签

1.在标签处理器中编写每个属性对应的setter方法

2.在TLD文件中描术标签的属性

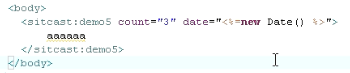
<tag>

<name>demo</name>

<tag-class>cn.ljh.web.simpletag.SimpleTagTest</tag-class>

<body-content>scriptless</body-content> <!—表示这个标签内不能出现脚本-->

<attribute> <!—服务器可以自动8中基本数据类型转换，如下传入的属性值要转为date对象，则会报错，需要注册转换器 如下使用el表达式转换传入-->

<!----------------- ------------->

<name>count</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue><!—runtime expression 运行时表达式true表示可以是运行时表达式，也就是说这个count属性的值可以使用如el表达式等-->

</attribute>

<attribute>

<name>date</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

<type>java.lang.Date</type> <!—页面中只能是Date对象，否则报错 -->

</attribute>

</tag>

###### 余下jsp页面不执行throw new SkipPageException()

###### 自定义开发if else标签

将if else标签写在同一个父标签内

###### 自定义开发迭代标签

JSP代码：

<body>

<%

List list = new ArrayList();

list.add("aaa");

list.add("bbb");

request.setAttribute("list",list);

%>

<c:foreach var="str" items="${list }">

${str }

</c:foreach>

</body>

public class ForeachTag extends SimpleTagSupport {（只能迭代list集合）

private Object items; //这里是Object类型

private String var; //这里注意：类型是String而不是list存放对象类型

public void setItems(Object items) {

this.items = items;

}

public void setVar(String var) {

this.var = var;

}

@Override

public void doTag() throws JspException, IOException {

List list = (List) items;

Iterator it = list.iterator();

while(it.hasNext()){

Object value = it.next();

//var类型String的原因，存放在PageContex对象中，Java快速for循环是集合存的对象类型，这里不同，需要注意

this.getJspContext().setAttribute(var, value); this.getJspBody().invoke(null);

}

}

}

public class ForeachTag2 extends SimpleTagSupport {（能迭代任何类型）

private String var; private Object items; private Collection collection;

public void setItems(Object items) {

this.items = items;

if(items instanceof Collection){ //list set

collection = (Collection) items;

}

if(items instanceof Map){

Map map = (Map) items;

collection = map.entrySet(); //set

}

/////8中基本数据类型的迭代，因为int[]不属于Object[]

if(items.getClass().isArray()){

this.collection = new ArrayList();

int length = Array.getLength(items);

for(int i=0;i<length;i++){

Object value = Array.get(items, i);

this.collection.add(value);

}

}

/\*if(items instanceof Object[]){ //int[]时不行，8中基本数据类型都不行

Object obj[] = (Object[]) items;

collection = Arrays.asList(obj);

}

//这里判断是否是int类型，则需要对不同数组都判断，比如是String、Enumerration等，JSP源码就是这样判断的，以上items.getClass().isArray()能优化

if(items instanceof int[]){

int arr[] = (int[]) items;

this.collection = new ArrayList();

for(int i : arr){

this.collection.add(i);

}

}\*/

}

public void setVar(String var) {

this.var = var;

}

public void doTag() throws JspException, IOException {

Iterator it = this.collection.iterator();

while(it.hasNext()){

Object value = it.next();

this.getJspContext().setAttribute(var, value);

this.getJspBody().invoke(null);

}}}

###### 开发html转义标签

论坛里面常用，直接输出html代码

###### 打jar包

jar的配置文件全部房子啊META-INF文件夹下

使用工具exe4J工具打包jar包

##### JSTL

在jstl.jar包中的META-INF目录下c.tld是核心标签，fmt是国际化标签，fn是el函数，sql数据库标签，x是xml标签，c-1.0类似的是1.0规范中的标签

核心标签：

<body>

<br/>-------------------c:out---------------------------<br/>

<%

request.setAttribute("data","bbbb");

%>

<c:out value="${data}" default="aaaaa" escapeXml="true"></c:out>

<br/>-------------------c:set可以操给各个域 javabean Map集合---------------------------<br/>

<c:set var="data" value="xxx" scope="page"/>

${data }

<%

Map map = new HashMap();

request.setAttribute("map",map);

%> //map集合中存入dd和xxx

<c:set property="dd" value="xxx" target="${map}"/>

${map.dd }

<%

Person p = new Person();

request.setAttribute("p",p);

%> //Persion中的name属性中存uuu

<c:set property="name" value="uuuu" target="${p}"/>

${p.name }

<br/>-------------------c:catch---------------------------<br/>

<c:catch var="myex"> //异常对戏那个为myex

<%

int x = 1/0;

%>

</c:catch>

${myex.message}

<br/>-------------------c:if---------------------------<br/>

<c:if var="aaa" test="${user==null}" scope="page">//将if结果存在page域

aaaa

</c:if>

${aaa }

<br/>-------------------c:foreach---------------------------<br/>

<%

List list = new ArrayList();

list.add("aaa");

list.add("bbb");

list.add("ccc");

list.add("dddd");

request.setAttribute("list",list);

%>

<c:forEach var="str" items="${list}"> //str对象是String类型，注意!

${str }

</c:forEach>

<br/>

<c:forEach var="num" begin="1" end="9" step="1"> //jsp分页

${num }

</c:forEach>

<br/>-------------------c:foreach实现表格间色显示---------------------------<br/>

<%

list = new ArrayList();

list.add("aaa");

list.add("bbb");

list.add("ccc");

list.add("dddd");

list.add("eeee");

list.add("ffff");

list.add("gggg");

list.add("hhhh");

list.add("kkkkk");

request.setAttribute("list",list);

%>

<%--${status }获取到了一个对象，这个对象记住了当前是第几次迭代 --%>

<style>

.odd{background-color: #FF99FF;}

.even{background-color: #FF6633;}

tr:hover{background-color: #0000FF;}

</style>

<!-- 使用td：hover需要修改文本的申明为XHTML

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

-->

<table border="1" width="20%">

<c:forEach var="str" items="${list}" varStatus="status">

<tr class="${status.count%2==0?'even':'odd' }">

<td>${str }</td>

</tr>

</c:forEach>

</table>

<br/>-------------------c:url标签---------------------------<br/>

//重建url地址，自己写需要在地址的后面加上sessionid（防止用户禁用了cookie），使用这个标签会自动加上。如下截图是使用后在html页面的源码：



注意：同时没有var值，将会直接输出，而不是重新。

value的值会加上工程路径

<c:url var="index" value="/index.jsp">

<c:param name="name" value="中国"></c:param> //url中文会自动编码

</c:url>

<a href='${index }'>点点</a>

<br/>-------------------c:url标签---------------------------<br/>

<%

pageContext.setAttribute("data","aa,bb,cc,dd");

%>

<c:forTokens var="ss" items="${data}" delims=",">

${ss }

</c:forTokens>

</body>

#### 第十二章

##### EL表达式（Expression Language）

###### 获取数据

$（data） ：pageContext.findAttribute(”data”):从四个域（page，request，session，application）去找，没找到返回空字符串

###### 表达式

$(empty(list)): 判断list是否是null

$(user!=null?user.name: “”):判断用户是否登录。

###### $()获取11个el内置对象

和获取数据比较：EL表达式会先判断是否是内置对象

|  |  |
| --- | --- |
| **隐含对象名称** | **描 述** |
| pageContext | 对应于JSP页面中的pageContext对象（注意：取的是pageContext对象。） |
| pageScope | 代表page域中用于保存属性的Map对象 |
| requestScope | 代表request域中用于保存属性的Map对象 |
| sessionScope | 代表session域中用于保存属性的Map对象（在这里判断用户是否登录，更严谨） |
| applicationScope | 代表application域中用于保存属性的Map对象 |
| param | 表示一个保存了所有请求参数的Map对象(数据回显用的特别多，只能一个值，下面对象多个值) 很重要的功能 |
| paramValues | 表示一个保存了所有请求参数的Map对象，它对于某个请求参数，返回的是一个string[] |
| header | 表示一个保存了所有http请求头字段的Map对象 |
| headerValues | 同上，返回string[]数组。注意：如果头里面有“-” ，例Accept-Encoding，则要headerValues[“Accept-Encoding”] |
| cookie | 表示一个保存了所有cookie的Map对象 |
| initParam | 表示一个保存了所有web应用初始化参数的map对象 |

###### EL调用java方法

1. 方法必须是static的
2. 还需要在tld文件中描述

tld文件在tomcat/example例子中的jsp2文件夹下面

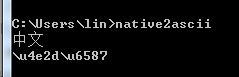
是否可以移出代码，替换自定义标签：不能替代，因为只能移出和web开发无关的代码，比如方法中request，el是无法移出的，因为没有web开发的对象

##### SUN公司：EL函数库

导入jstl.jar包

##### java-web国际化

###### java中提供了一个工具“native2ascii”：



###### *创建资源包和资源文件*

一个资源包中的每个资源文件都必须拥有共同的基名. 例如：一个资源包的基名是“myproperties”，则与中文、英文环境相对应的资源文件名则为:

“myproperites\_zh.properties” “myproperites\_en.properties”

###### ResourceBundle类提供了一个静态方法getBundle，该方法用于装载资源文件，并创建ResourceBundle实例：

Locale currentLocale = Locale.getDefault();

ResourceBundle myResources =

ResourceBundle.getBundle(basename, currentLocale);

basename为资源包基名(且必须为完整路径)。

如果与该locale对象匹配的资源包子类找不到。一般情况下，则选用默认资源文件予以显示。

public static void main(String[] args) {

ResourceBundle bundler =

ResourceBundle.getBundle("cn.itcast.resource.MessageResource",Locale.CANADA);

String username = bundler.getString("username");

String password = bundler.getString("password");

System.out.println(username);

System.out.println(password);

}

###### *DateFormat类(国际化日期)*

###### NumberFormat类

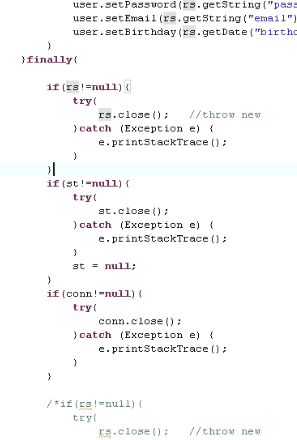
###### MessageFormat(动态文本)

##### 在线支付（未看）

#### 第十三章（mysql 未看）

#### 第十四章

##### 关闭数据库代码：



##### 每一层设计自己的异常，方便定位

#### 第十五章

##### mysql分页

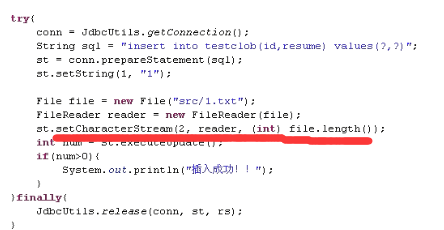
mysql:select \* from test limit 0,5;

oracle:使用rownum

##### 存储大文本（这里是字符流）

一般不在数据库中存储大数据，除非重要资料，如淘宝认证扫描证件等。

大文本需要使用流去存储（和数据库关联，然后按照给定的长度一点点读取）

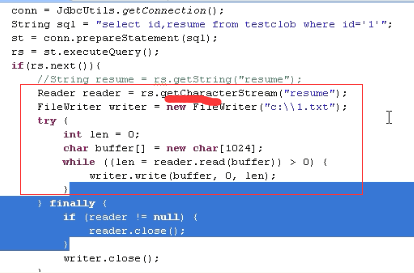


File file = new File(“src/1.txt”);

FileReader reader = new FileReader(file);

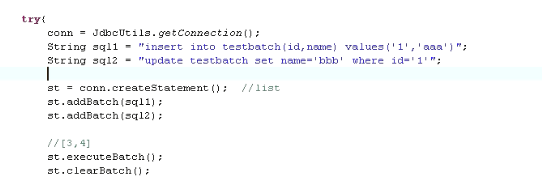
st.setCharacterStream(2,reader,file.length());

##### 读取大数据（这里是字符流）

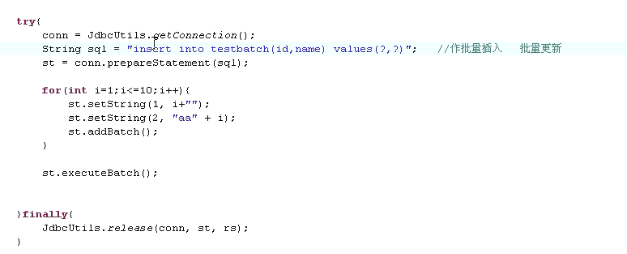


##### 批量处理sql数据两种方式：

第一种：



第二种：



##### JDBC获取主键

PreparedStatement prepareStatement(String sql,

int autoGeneratedKeys)

throws SQLException

**参数：**

sql - 可能包含一个或多个 '?' IN 参数占位符的 SQL 语句

autoGeneratedKeys - 指示是否应该返回自动生成的键的标志，它是 Statement.RETURN\_GENERATED\_KEYS 或 Statement.NO\_GENERATED\_KEYS 之一

**返回：**

一个包含预编译 SQL 语句的新 PreparedStatement 对象，该对象能够返回自动生成的键

默认返回的就是带参数的，建议带上后面的参数，方便通用。

##### JDBC调用存储过程

CallableStatement cStmt = conn.prepareCall(“call demoSp(?,?)”); //调用demoSp这个存储过程，该存储过程第二个值是返回值

cStmt.setString(1,”a”);

cStmt.registerOutParameter(2,Types.VARCHAR);//注册返回值类型

cStmt.execute();

System.out.println(cStmt.getString(2));

#### 第十六章

##### 事务

start transaction

…

commit

rollback

##### jdbc控制事务

connection.setAutoCommit(false);//默认是开启的，这句话相当于start transaction

只要抛出异常都不回成功

connection.rollback();

connection.commit();

Savepoint sp = connection.setSavepoint()//设置回滚点，返回一个回滚点

connection.rollback(sp); //回滚到sp这个回滚点，手工回滚之后要commit

connection.commit();//手工回滚之后要commit

##### 事务ACID

原子性atomicity、一致性consistency、隔离性isolation、持久性durability

##### 设置数据库隔离级别

问题：脏读、不可重复读、幻读

* 数据库共定义了四种隔离级别：
  + Serializable：可避免脏读、不可重复读、虚读情况的发生。（串行化）
  + Repeatable read：可避免脏读、不可重复读情况的发生。（可重复读）
  + Read committed：可避免脏读情况发生（读已提交）。
  + Read uncommitted：最低级别，以上情况均无法保证。(读未提交)
* set transaction isolation level 设置事务隔离级别
* select @@tx\_isolation 查询当前事务隔离级别

如set transaction isolation level read committd；//设置避免脏读取

设置了serializable其他就会等他上一个执行完

要使用cmd演示重复读、幻读，需要开始事务start transaction

mysql支持四种隔离级别，默认是repeatable read

oracle只支持两种，serializable和read committed

connection 中：void **setTransactionIsolation**(int level)

throws [SQLException](mk:@MSITStore:E:\API帮助文档\API帮助文档\JDK_API_1_6_zh_CN.CHM::/java/sql/SQLException.html)

一般常用的是原子性，隔离性比较少，一致性和持久性是数据库自己实现。

##### 数据库连接池

需要实现DataSoure接口

##### 包装设计模式

在服务器开发中，传递过来的对象含有自己的信息，使用子类去扩展功能一般还要把对象中附带的其他对象也得跟着写。使用包装设计模式就不需要了。

但是最好的方法还是使用动态代理

public class JdbcPool implements DataSource {

private static LinkedList<Connection> list = new LinkedList<Connection>();

//使用LinkedList来保存连接，方便修改（插入修改快，查询慢）

private static Properties config = new Properties();

static{

try {

config.load(JdbcUtils\_DBCP.class.getClassLoader().getResourceAsStream("db.properties"));

Class.forName(config.getProperty("driver"));

for(int i=0;i<10;i++){

Connection conn = DriverManager.getConnection(config.getProperty("url"), config.getProperty("username"), config.getProperty("password"));

list.add(conn);

}

} catch (Exception e) {

throw new ExceptionInInitializerError(e);

}

}

//conn.close()

/\* 在实际开发，发现对象的方法满足不了开发需求时，有三种方式对其进行增强

\* 1.写一个connecton子类，覆盖close方法，增强close方法

\* 2.用包装设计模式 3.用动态代理 aop 面向切面编程

\*/

public Connection getConnection() throws SQLException {

if(list.size()<=0){

throw new RuntimeException("数据库忙，请稍会再来！！");

}

Connection conn = list.removeFirst(); //mysqlconnection C

MyConnection my = new MyConnection(conn);

return my; //my-------preparedStatment commit createStatement close

}

//1.定义一个类，实现与被增强相同的接口

//2.在类中定义一个变量，记住被增强对象

//3.定义一个构造函数，接收被增强对象

//4.覆盖想增强的方法

//5.对于不想增强的方法，直接调用目标对象（被增强对象）的方法

class MyConnection implements Connection{

private Connection conn;

public MyConnection(Connection conn){

this.conn = conn;

}

public void close(){

list.add(this.conn);

}

public void clearWarnings() throws SQLException {

this.conn.clearWarnings();

}

public void commit() throws SQLException {

this.conn.commit();

}

public Statement createStatement() throws SQLException {

return this.conn.createStatement();

}

public Statement createStatement(int resultSetType,

int resultSetConcurrency, int resultSetHoldability)

throws SQLException {

return this.conn.createStatement(resultSetType, resultSetConcurrency, resultSetHoldability);

}

public Statement createStatement(int resultSetType,

int resultSetConcurrency) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public boolean getAutoCommit() throws SQLException {

// TODO Auto-generated method stub

return false;

}

public String getCatalog() throws SQLException {

// TODO Auto-generated method stub

return null;

}

public int getHoldability() throws SQLException {

// TODO Auto-generated method stub

return 0;

}

public DatabaseMetaData getMetaData() throws SQLException {

// TODO Auto-generated method stub

return null;

}

public int getTransactionIsolation() throws SQLException {

// TODO Auto-generated method stub

return 0;

}

public Map<String, Class<?>> getTypeMap() throws SQLException {

// TODO Auto-generated method stub

return null;

}

public SQLWarning getWarnings() throws SQLException {

// TODO Auto-generated method stub

return null;

}

public boolean isClosed() throws SQLException {

// TODO Auto-generated method stub

return false;

}

public boolean isReadOnly() throws SQLException {

// TODO Auto-generated method stub

return false;

}

public String nativeSQL(String sql) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public CallableStatement prepareCall(String sql, int resultSetType,

int resultSetConcurrency, int resultSetHoldability)

throws SQLException {

// TODO Auto-generated method stub

return null;

}

public CallableStatement prepareCall(String sql, int resultSetType,

int resultSetConcurrency) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public CallableStatement prepareCall(String sql) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PreparedStatement prepareStatement(String sql,

int resultSetType, int resultSetConcurrency,

int resultSetHoldability) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PreparedStatement prepareStatement(String sql,

int resultSetType, int resultSetConcurrency)

throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PreparedStatement prepareStatement(String sql,

int autoGeneratedKeys) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PreparedStatement prepareStatement(String sql,

int[] columnIndexes) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PreparedStatement prepareStatement(String sql,

String[] columnNames) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PreparedStatement prepareStatement(String sql)

throws SQLException {

// TODO Auto-generated method stub

return null;

}

public void releaseSavepoint(Savepoint savepoint) throws SQLException {

// TODO Auto-generated method stub

}

public void rollback() throws SQLException {

// TODO Auto-generated method stub

}

public void rollback(Savepoint savepoint) throws SQLException {

// TODO Auto-generated method stub

}

public void setAutoCommit(boolean autoCommit) throws SQLException {

// TODO Auto-generated method stub

}

public void setCatalog(String catalog) throws SQLException {

// TODO Auto-generated method stub

}

public void setHoldability(int holdability) throws SQLException {

// TODO Auto-generated method stub

}

public void setReadOnly(boolean readOnly) throws SQLException {

// TODO Auto-generated method stub

}

public Savepoint setSavepoint() throws SQLException {

// TODO Auto-generated method stub

return null;

}

public Savepoint setSavepoint(String name) throws SQLException {

// TODO Auto-generated method stub

return null;

}

public void setTransactionIsolation(int level) throws SQLException {

// TODO Auto-generated method stub

}

public void setTypeMap(Map<String, Class<?>> map) throws SQLException {

// TODO Auto-generated method stub

}

}

public Connection getConnection(String username, String password)

throws SQLException {

// TODO Auto-generated method stub

return null;

}

public PrintWriter getLogWriter() throws SQLException {

// TODO Auto-generated method stub

return null;

}

public int getLoginTimeout() throws SQLException {

// TODO Auto-generated method stub

return 0;

}

public void setLogWriter(PrintWriter arg0) throws SQLException {

// TODO Auto-generated method stub

}

public void setLoginTimeout(int arg0) throws SQLException {

// TODO Auto-generated method stub

}

}

##### 开源数据库连接池

###### DBCP ----apache

C3P0----spring内置

Tomcat内置连接池（DBCP）

都实现了DataSource接口

DBCP需要使用:common-dbcp-\*\*.jar以及common-poo-\*\*l.jar

C3P0需要：c3p0-\*\*.jar以及mchange-common-\*\*.jar

###### DBCP:

static{

InputStream in = JdbcUtil.**class**.getClassLoader().

getResourceAsStream("dbcpconfig.properties");

Properties prop = **new** Properties();

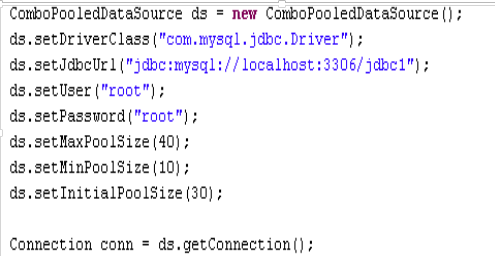
prop.load(in);

BasicDataSourceFactory factory = **new** BasicDataSourceFactory();

*dataSource* = factory.*createDataSource*(prop);

}

###### c3p0:



配置tomcat中的连接池：context可以在server.xml或者catelina文件夹下配置，还可以在META-INF中配置。

|  |  |  |
| --- | --- | --- |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | <Context ...>  ...  <Resource name="jdbc/EmployeeDB"  auth="Container"  type="javax.sql.DataSource"  username="dbusername"  password="dbpassword"  driverClassName="org.hsql.jdbcDriver"  url="jdbc:HypersonicSQL:database"  maxActive="8"  maxIdle="4"/>  ...  </Context> |  |

在tomcat 帮助文档中的JNDI Resources中查看。同时使用tomcat创建连接池，需要将对应的数据库驱动放到tomcat的lib目录下

在dao层获取数据库连接池：

|  |  |  |
| --- | --- | --- |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | Context initCtx = **new** InitialContext();  Context envCtx = (Context) initCtx.lookup("java:comp/env");  *dataSource* = (DataSource)envCtx.lookup("jdbc/datasource"); |  |

##### JNID技术

* JNDI(Java Naming and Directory Interface)，Java命名和目录接口，它对应于J2SE中的javax.naming包，
* 这套API的主要作用在于：它可以把Java对象放在一个容器中（JNDI容器），并为容器中的java对象取一个名称，以后程序想获得Java对象，只需通过名称检索即可。
* 其核心API为Context，它代表JNDI容器，其lookup方法为检索容器中对应名称的对象。

##### JDBC框架（以下内容在做自己JBDC框架需要）

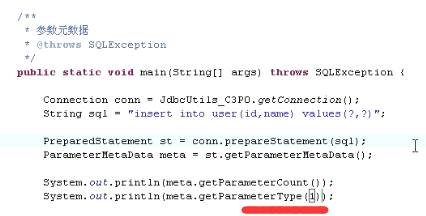
###### 数据库元数据

connection.getMetaData() //获取数据库元数据

* 元数据：数据库、表、列的定义信息。
* Connection.getDatabaseMetaData()
* DataBaseMetaData对象
  + getURL()：返回一个String类对象，代表数据库的URL。
  + getUserName()：返回连接当前数据库管理系统的用户名。
  + getDatabaseProductName()：返回数据库的产品名称。
  + getDatabaseProductVersion()：返回数据库的版本号。
  + getDriverName()：返回驱动驱动程序的名称。
  + getDriverVersion()：返回驱动程序的版本号。
  + isReadOnly()：返回一个boolean值，指示数据库是否只允许读操作。

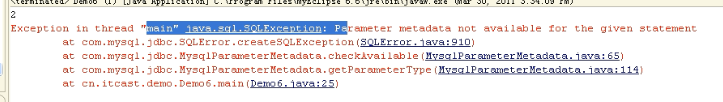
###### 参数元数据

* PreparedStatement . getParameterMetaData()
  + 获得代表PreparedStatement元数据的ParameterMetaData对象。
  + Select \* from user where name=? And password=?
* ParameterMetaData对象
  + getParameterCount()
    - 获得指定参数的个数
  + getParameterType(int param)
    - 获得指定参数的sql类型



第一个打印获取预编译需要参数个数，

第二个打印出第一个参数的数据类型（mysql不支持，会支持异常）



###### 结果元数据

* ResultSet. getMetaData()
  + 获得代表ResultSet对象元数据的ResultSetMetaData对象。
* ResultSetMetaData对象
  + getColumnCount**()**
    - 返回resultset对象的列数
  + getColumnName(int column)
    - 获得指定列的名称
  + getColumnTypeName(int column)
    - 获得指定列的类型

###### 编写框架思想：

当你不知道用户怎么处理一个对象的时候，你可以提供一个接口给用户，让用户去实现接口相应的方法。(策略模式)

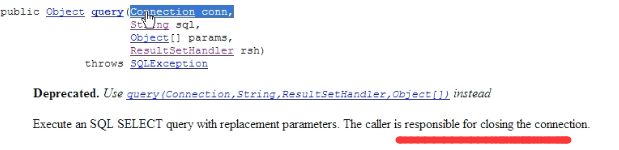
如：TreeSet中存入一个比较器去实现add数据的排序.

#### 第十七章

##### Apache-Dbutils

* commons-dbutils 是 Apache 组织提供的一个开源 JDBC工具类库，它是对JDBC的简单封装，学习成本极低，并且使用dbutils能极大简化jdbc编码的工作量，同时也不会影响程序的性能。因此dbutils成为很多不喜欢hibernate的公司的首选。
* API介绍：
  + org.apache.commons.dbutils.QueryRunner （核心）

使用这个类的无参数构造new QueryRunner（）对象在查询时需要使用：



1.这个方法中传入一个连接，同时注意需要用户自己手动去关闭连接。这种方法可以将多个执行**放入一个事务中执行，**然后关闭连接

2.结果处理器（让用户去实现对查询结果的处理）

3.这个类中还有几个protect方法，protect这个方法可以给子类或者同一个包使用

* + org.apache.commons.dbutils.ResultSetHandler
  + 工具类
    - org.apache.commons.dbutils.DbUtils、。

##### 使用Dbutils框架中new QueryRunner（Connection）就不需要自己写release方法去释放了，框架会自动释放，

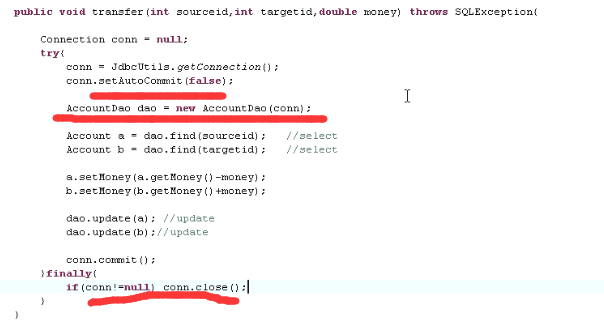
如果使用new QueryRunner（）做事务则要自己关闭连接。

##### 处理器ResultSetHandler 接口的实现类

* ArrayHandler：把结果集中的第一行数据转成对象数组。
* ArrayListHandler：把结果集中的每一行数据都转成一个数组，再存放到List中。
* BeanHandler：将结果集中的**第一行**数据封装到一个对应的JavaBean实例中。
* BeanListHandler：将结果集中的每一行数据都封装到一个对应的JavaBean实例中，存放到List里。
* ColumnListHandler：将结果集中某一列的数据存放到List中。
* KeyedHandler(name)：将结果集中的每一行数据都封装到一个Map里，再把这些map再存到一个map里，其key为指定的key。
* MapHandler：将结果集中的第一行数据封装到一个Map里，key是列名，value就是对应的值。
* MapListHandler：将结果集中的每一行数据都封装到一个Map里，然后再存放到List

还有一个ScalarHandeler处理器，将一行数据存入到一个对象中。

##### DBUTILS事务管理



但是以上不优雅。

使用ThreadLocal，new ThreadLoal（Connection）用完之后要移出，因为ThreadLocal本质是相当于一个map容易，key是当前线程，值是Connection，如果不移出，跑的时间久了，内存会崩。使用ThreadLocal写的事务可以在多线程里面使用。

##### ThreadLocal可以在多线程中实现数据共享（17.2事务再听）

**ThreadLocal可以在一个线程中使用，设置为静态，只要是一个线程中跑的对象都能获得。但是在使用后一定要remove（），否则会引起内存溢出。**（重要）

传递数据还可以在JUID容器中保存然后需要再从容器中获取。

##### 在实际开发中尽量不使用“1：多”，而使用“多：1”

比如：部门和员工。在员工在有部门对象，在部门就有员工集合，就是双向关系。

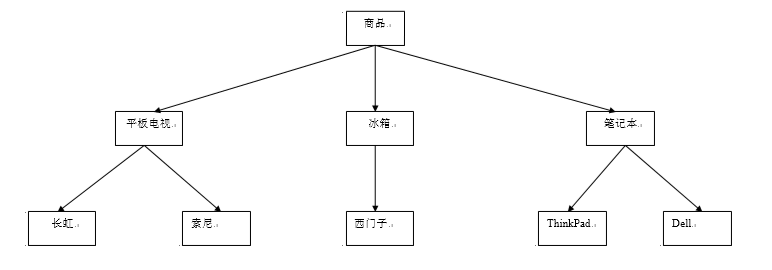
尽量不要在部门中有员工集合。因为当员工有很多的时候容易**内存溢出**。因此遵守在员工中有部门，但是部门尽量别有员工集合。

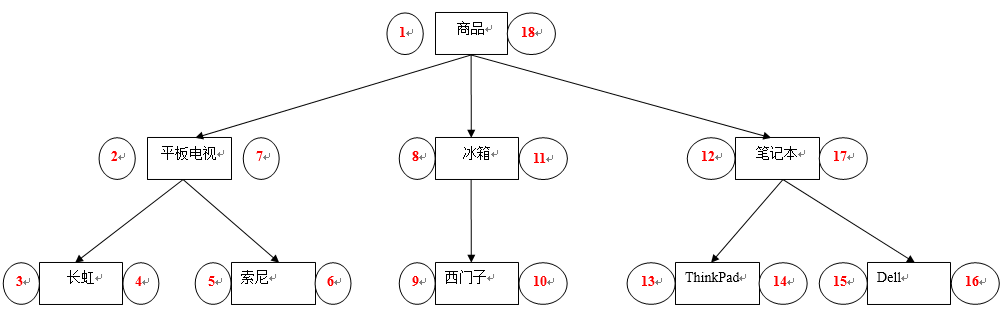
比如：订单就要双向，因为单独一个订单的基本信息没什么意义，所用要在订单中有订单项集合。但是订单项过多时，要分页。具体要不要还要看需求！在1方的显示的时候是否需要现实多方，如果要，就要1：多。否则尽量别1：多

##### 多对多使用中间表

##### Tree表（二叉树表）

需要做出如下表结构：





树状节点的特点：

1. 每一个节点都有一个左右值。
2. 如果右值-左值=1，则代表当前节点为叶子节点。
3. 如果右值-左值>1，则代表当前节点有孩子节点，值在左右值之间的所有节点，即为当前结点的所有孩子节点。

数据库表设计：

create table category

(

id varchar(40) primary key,

name varchar(100),

lft int,

rgt int

);

insert into category values('1','商品',1,18);

insert into category values('2','平板电视',2,7);

insert into category values('3','冰箱',8,11);

insert into category values('4','笔记本',12,17);

insert into category values('5','长虹',3,4);

insert into category values('6','索尼',5,6);

insert into category values('7','西门子',9,10);

insert into category values('8','thinkpad',13,14);

insert into category values('9','dell',15,16);

问题：为了在页面中显示树状结构，需要得到所有结点，以及每个结点在树中的层次：

解决思路：

1. 要得到结点的层次，就是看节点有几个父亲，例如长虹有2个父亲，则它所在层次就为2。
2. 如何知道每一个节点有几个父亲呢？这个表有个特点，父亲和孩子都在同一个表中，为得到父亲所有的孩子，可以把这张表想像成两张表，一张用于保存父亲，一张表保存孩子，如下所示：

select \* from category parent,category child;

1. 父亲下面的孩子有个特点，它的左值>父亲的左值，并且<父亲的右值，如下所示

select \* from category parent,category child where child.lft>=parent.lft and child.rgt<=parent.rgt;

以上语句会得到父亲下面所有的孩子。

1. 对父亲所有孩子的姓名进行归组，然后使用count统计函数，这时就会知道合并了几个孩子，合并了几个孩子姓名，这个孩子就有几个父亲，从而知道它所在的层次

select child.name,count(child.name) depth from category parent,category child where child.lft>=parent.lft and child.rgt<=parent.rgt group by child.name;

1. 最后根据左值排序即可

select child.name,count(child.name) depth from category parent,category child where child.lft>=parent.lft and child.rgt<=parent.rgt group by child.name order by child.lft;

#### 第十八章

##### 文件上传下载

* 实现web开发中的文件上传功能，需完成如下二步操作：
  + 在web页面中添加上传输入项
  + 在servlet中读取上传文件的数据，并保存到本地硬盘中。
* 如何在web页面中添加上传输入项?
  + <input type=“file”>标签用于在web页面中添加文件上传输入项，设置文件上传输入项时须注意：
  + 1、必须要设置input输入项的name属性，否则浏览器将不会发送上传文件的数据。
  + ２、必须把form的enctype属值设为**multipart/form-data.**设置该值后，浏览器在上传文件时，将把文件数据附带在http请求消息体中，并使用ＭＩＭＥ协议对上传的文件进行描述，以方便接收方对上传数据进行解析和处理。

##### 上传文件读取

上传文件ecctype属性设置为multipart/form-data后，则通过reequest.getParamter（”XXX”）是无法读取到的，因为会以MIME协议上传到服务器。在servlet中注意就不能采用传统方式获取数据

/\*String username = request.getParameter("username");

System.out.println(username);\*/

* 如何在Servlet中读取文件上传数据，并保存到本地硬盘中?
  + Request对象提供了一个getInputStream方法，通过这个方法可以读取到客户端提交过来的数据。但由于用户可能会同时上传多个文件，在servlet端编程直接读取上传数据，并分别解析出相应的文件数据是一项非常麻烦的工作，示例。
  + 为方便用户处理文件上传数据，Apache 开源组织提供了一个用来处理表单文件上传的一个开源组件（ Commons-fileupload ），该组件性能优异，并且其API使用极其简单，可以让开发人员轻松实现web文件上传功能，因此在web开发中实现文件上传功能，通常使用Commons-fileupload组件实现。
* 使用Commons-fileupload组件实现文件上传，需要导入该组件相应的支撑jar包：Commons-fileupload和commons-io。commons-io 不属于文件上传组件的开发jar文件，但Commons-fileupload 组件从1.1 版本开始，它工作时需要commons-io包的支持。

##### 使用fileupload解析数据

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

**try**{

//创建解析工厂

//创建解析器

//调用解析器解析request，得到保存了所有上传数据的list

//迭代list集合，拿到封装了每个输入项数据的fileItem

//判断item的类型，如果是普通字段，则直接获取数据，如果为上传文件，则调用流获取数据写到本地硬盘

DiskFileItemFactory factory = **new** DiskFileItemFactory();

ServletFileUpload upload = **new** ServletFileUpload(factory);

//upload.setHeaderEncoding(“UTF-8”);解决中文乱码问题，还需要下面代码配合

List<FileItem> list = upload.parseRequest(request);

**for**(FileItem item : list){

**if**(item.isFormField()){

//为普通输入项

String inputName = item.getFieldName();

String inputValue = item.getString();///解决中文乱码还需要设置这个方法item.getString(“UTF-8”);原理就是inputValue = new String（inputvalue.getBytes(“iso8859-1”,”UTF-8”);

System.*out*.println(inputName + "=" + inputValue);

}**else**{

//代表当前处理的item里面封装的是上传文件

//C:\Documents and Settings\ThinkPad\桌面\a.txt a.txt

String filename = item.getName().substring(item.getName().lastIndexOf("\\")+1);

InputStream in = item.getInputStream();

**int** len = 0;

**byte** buffer[] = **new** **byte**[1024];

FileOutputStream out = **new** FileOutputStream("c:\\" + filename);

**while**((len=in.read(buffer))>0){

out.write(buffer, 0, len);

}

in.close();

out.close();

}

}

}**catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

<body>

<form action=*"*${pageContext.request.contextPath }*/servlet/UploadServlet3"* enctype=*"multipart/form-data"* method=*"post"*>

上传用户：<input type=*"text"* name=*"username"*><br/>

上传文件1：<input type=*"file"* name=*"file1"*><br/>

上传文件2：<input type=*"file"* name=*"file2"*><br/>

<input type=*"submit"* value=*"上传"*>

</form>

</body>

##### 上传问题

1.上传文件的中文乱码

1.1 解决文件的乱码

ServletFileUpload.setHeaderEncoding("UTF-8")

1.2 解决普通输入项的乱码（注意，表单类型为multipart/form-data的时候，设置request的编码是无效的）

FileItem.setString("UTF-8"); //解决乱码 如上输入框！

2.在处理表单之前，要记得调用：

ServletFileUpload.isMultipartContent方法判断提交表单的类型，如果该方法返回true，则按上传方式处理，否则按照传统方式处理表单即可。

3.设置解析器缓冲区的大小，以及临时文件的删除

设置解析器缓冲区的大小 DiskFileItemFactory.setSizeThreshold(1024\*1024);

临时文件的删除：在程序中处理完上传文件后，一定要记得调用item.delete()方法，以删除临时文件

4.在做上传系统时，千万要注意上传文件的保存目录，这个上传文件的保存目录绝对不能让外界直接访问到。（比如我上传一个java脚本的.jsp文件，然后在访问，则能执行jsp中的脚本）

可以把临时文件放在WEB-INF文件下面

5.限制上传文件的类型

在处理上传文件时，判断上传文件的后缀名是不是允许的

6.限制上传文件的大小

调用解析器的ServletFileUpload.setFileSizeMax(1024\*1024\*5);就可以限制上传文件的大小，如果上传文件超出限制，则解析器会抛FileUploadBase.FileSizeLimitExceededException（这是一个内部类）异常，程序员通过是否抓到这个异常，进而就可以给用户友好提示。

7.如何判断空的上传输入项

String filename = item.getName().substring(item.getName().lastIndexOf("\\")+1); //如果没有上传输入项，改方法返回空字符串，而不是null

if(filename==null || filename.trim().equals("")){

continue;

}

要加上面这个代码，否则会报“拒绝访问”的错误，因为服务器读取到的是用户想在WEB-INF目录下直接写文件，这个是不允许的。

8、为避免上传文件的覆盖，程序在保存上传文件时，要为每一个文件生成一个唯一的文件名

public String generateFileName(String filename){

//83434-83u483-934934

return UUID.randomUUID().toString() + "\_" + filename;

}

9、为避免在一个文件夹下面保存超过1000个文件，影响文件访问性能，程序应该把上传文件打散后存储。

public String generateSavePath(String path,String filename){

int hashcode = filename.hashCode(); //121221

int dir1 = hashcode&15;

int dir2 = (hashcode>>4)&0xf; （>>移动四位，与上15，二级目录类似还可以三级目录，。。。。。总共就8层）

String savepath = path + File.separator + dir1 + File.separator + dir2;

File file = new File(savepath);

if(!file.exists()){

file.mkdirs();

}

return savepath;

}

10、监听上传进度

ServletFileUpload upload = new ServletFileUpload(factory);

upload.setProgressListener(new ProgressListener(){

public void update(long pBytesRead, long pContentLength, int pItems) {

System.out.println("当前已解析：" + pBytesRead);

}

});

11、在web页面中添加动态上传输入项

##### 上传综合案例（pass）

#### 第十九章

##### 过滤器

filter在开发中的常见应用：

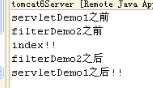
1.filter可以目标资源执行之前，进行权限检查，检查用户有无权限，如有权限则放行，如没有，则拒绝访问

2.filter可以放行之前，对request和response进行预处理，从而实现一些全局性的设置。

3.filter在放行之后，可以捕获到目标资源的输出，从而对输出作出类似于压缩这样的设置

##### filter执行顺序

根据filter-mapping顺序：



##### filter生命周期

web服务启动，运行init();

dofilter(),每次调用的时候执行

web服务停止，运行destory();

##### 统一全站字符编码的过滤器（只对post有效）

**public** **void** doFilter(ServletRequest req, ServletResponse resp,

FilterChain chain) **throws** IOException, ServletException {

//获取要设置的字符集

String charset = **this**.config.getInitParameter("charset");

**if**(charset==**null**){

charset = defaultCharset;

}

HttpServletRequest request = (HttpServletRequest) req;//先强转

HttpServletResponse response = (HttpServletResponse) resp;

request.setCharacterEncoding(charset); //post get

response.setCharacterEncoding(charset);

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

chain.doFilter(request, response);

}

##### 禁止浏览器缓存所有动态页面的过滤器

* + 有 3 个 HTTP 响应头字段都可以禁止浏览器缓存当前页面，它们在 Servlet 中的示例代码如下：
    - **response.setDateHeader("Expires",-1);**
    - **response.setHeader("Cache-Control","no-cache");**
    - **response.setHeader("Pragma","no-cache");**
  + 并不是所有的浏览器都能完全支持上面的三个响应头，因此最好是同时使用上面的三个响应头。
  + Expires数据头：值为GMT时间值，为-1指浏览器不要缓存页面
  + Cache-Control响应头有两个常用值：
    - no-cache指浏览器不要缓存当前页面。
    - max-age:xxx指浏览器缓存页面xxx秒。

##### 控制浏览器缓存页面中的静态资源的过滤器：

* + 场景：有些动态页面中引用了一些图片或css文件以修饰页面效果，这些图片和css文件经常是不变化的，所以为减轻服务器的压力，可以使用filter控制浏览器缓存这些文件，以提升服务器的性能。

**public** **void** doFilter(ServletRequest req, ServletResponse resp,

FilterChain chain) **throws** IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) req;

HttpServletResponse response = (HttpServletResponse) resp;

//1.获取到用户想访问的资源

String uri = request.getRequestURI();

//2.获取该资源的缓存时间

**int** expires = 0;

**if**(uri.endsWith(".jpg")){

expires = Integer.*parseInt*(**this**.config.getInitParameter("jpg"));

}**else** **if**(uri.endsWith(".css")){

expires = Integer.*parseInt*(**this**.config.getInitParameter("css"));

}**else**{

expires = Integer.*parseInt*(**this**.config.getInitParameter("js"));

}

response.setDateHeader("expires", System.*currentTimeMillis*()+expires\*60\*1000);

chain.doFilter(request, response);

}

##### 使用Filter实现URL级别的权限认证

* + 情景：在实际开发中我们经常把一些执行敏感操作的servlet映射到一些特殊目录中，并用filter把这些特殊目录保护起来，限制只能拥有相应访问权限的用户才能访问这些目录下的资源。从而在我们系统中实现一种URL级别的权限功能。
  + 要求：为使Filter具有通用性，Filter保护的资源和相应的访问权限通过filter参数的形式予以配置。

##### 实现用户自动登陆的过滤器

* + 在用户登陆成功后，发送一个名称为user的cookie给客户端，cookie的值为用户名和md5加密后的密码。
  + 编写一个AutoLoginFilter，这个filter检查用户是否带有名称为user的cookie来，如果有，则调用dao查询cookie的用户名和密码是否和数据库匹配，匹配则向session中存入user对象（即用户登陆标记），以实现程序完成自动登陆。

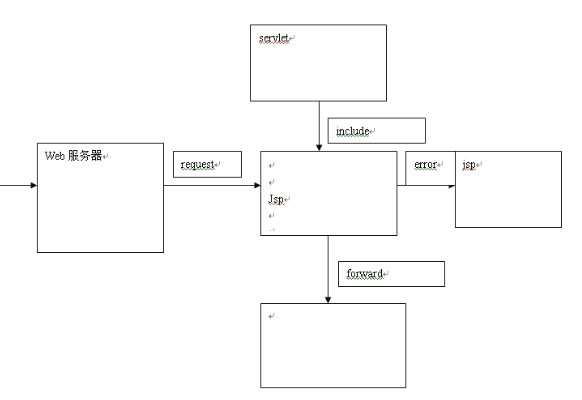
##### 拦截servlet、资源

1、<filter-name>子元素用于设置filter的注册名称。该值必须是在<filter>元素中声明过的过滤器的名字

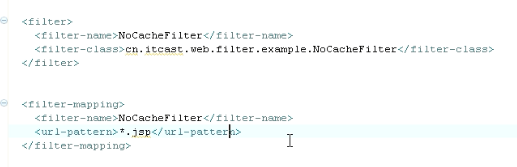
2、<url-pattern>设置 filter 所拦截的请求路径(过滤器关联的URL样式)

3、<servlet-name>指定过滤器所拦截的Servlet名称。

4、<dispatcher>指定过滤器所拦截的资源被 Servlet 容器调用的方式，可以是REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST。用户可以设置多个<dispatcher> 子元素用来指定 Filter 对资源的多种调用方式进行拦截。



第四点当使用以下配置时，直接访问jsp拦截器有效，但是当从一个jsp，forward访问时不会生效，必须配置dispatcher-FORWARD才会生效



##### dispatcher（四种方式必须大写）

* <dispatcher> 子元素可以设置的值及其意义：
  + REQUEST：当用户直接访问页面时，Web容器将会调用过滤器。如果目标资源是通过RequestDispatcher的include()或forward()方法访问时，那么该过滤器就不会被调用。
  + INCLUDE：如果目标资源是通过RequestDispatcher的include()方法访问时，那么该过滤器将被调用。除此之外，该过滤器不会被调用。
  + FORWARD：如果目标资源是通过RequestDispatcher的forward()方法访问时，那么该过滤器将被调用，除此之外，该过滤器不会被调用。
  + ERROR：如果目标资源是通过**声明式异常处理机制**调用时，那么该过滤器将被调用。除此之外，过滤器不会被调用。

##### filter高级开发

###### 增强的request对象（装饰模式）

1.写一个类，实现与被增强对象相同的接口

2.定义一个变量，记住被增强对象

3.定义一个构造方法，接收被增强对象

4.覆盖想增强的方法

5.对于不想增强的方法，直接调用被增强对象（目标对象）的方法

Servlet API 中提供了一个request对象的Decorator设计模式的默认实现类HttpServletRequestWrapper（用户去extends这个对象）

1. 设置全局的编码，当get提交时的字符
2. 字符转义

（转义代码在tomcat\apache-tomcat-6.0.41\webapps\examples\WEB-INF\classes\util，可以copy）

1. 敏感词汇过滤

###### 增强的response对象

Servlet API 中提供了response对象的Decorator设计模式的默认实现类HttpServletResponseWrapper

1. 压缩响应（较难，只看getOutputStream未看getWrite，有时间研究）

* 应用HttpServletResponseWrapper对象，压缩响应正文内容。思路:
  + 通过filter向目标页面传递一个自定义的response对象。
    - 在自定义的response对象中，重写getOutputStream方法和getWriter方法，使目标资源调用此方法输出页面内容时，获得的是我们自定义的ServletOutputStream对象。
    - 在我们自定义的ServletOuputStream对象中，重写write方法，使写出的数据写出到一个buffer中。
  + 当页面完成输出后，在filter中就可得到页面写出的数据，从而我们可以调用GzipOuputStream对数据进行压缩后再写出给浏览器，以此完成响应正文件压缩功能。

1. 缓存数据到内存

过滤器所拦截特定的servlet，过程：

//1.得到用户想访问的资源（uri）

String uri = request.getRequestURI();

//2.看map集合中是否保存了该资源的数据

byte b[] = map.get(uri);

//3.如果保存了，则直接取数据打给浏览器

if(b!=null){

response.getOutputStream().write(b);

return;

}

//4.如果没有保存数据，则放行让目标资源执行，这时还需写一个response的包装类，捕获目标资源的输出

MyResponse my = new MyResponse(response);

chain.doFilter(request, my);

byte data[] = my.getBuffer();

//5.以资源uri为关键字，打资源的数据保存map集合中，以备于下次访问

map.put(uri, data);

//6.输出数据给浏览器

response.getOutputStream().write(data);

#### 第二十章

##### 权限拦截

使用filter拦截 组粒度拦截

动态代理+注解 细粒度拦截

#### 第二十一章

##### ServletContextlistener

web应用启动和销毁时执行

##### HttpSessionListener

session创建，销毁（如果把浏览器的cookie禁止，每次访问JSP都会创建一个新的session，则每次访问都会运行sessionCreated()方法,因为禁止cookie后，每次访问服务器，都不回带sessionId，因此会每次都运行。如果用户访问的是servlet，那么sessionCreated()是不会运行的，因为只有JSP运行的时候被翻译城servlet，才会创建8大对象）

在web.xml文件中可以配置session销毁时间，默认是30分钟（以分钟为单位）



##### ServletRequestListener

request创建和销毁的时候运行

##### 集合帮助类Collections

synchronizedList(List<T> list)返回一个线程安全的线程。

##### 自定义session扫描器

线程安全问题：1。集合并发操作使用Collections

2。在管理session的时候在LinkedList中添加session以及定时器中Timer中定时remove过期的session会存在一个并发问题（在添加的同时有移出，使用syn共享一个object对象即可）

##### ServletContextAttributeListener

ServletContext的属性值变化时运行

类似的还有HttpSessionAttributeListener 、ServletRequestAttributeListener

##### 感知 Session 绑定的事件监听器

* 保存在 Session 域中的对象可以有多种状态：绑定到 Session 中(活化)；从 Session 域中解除绑定；随 Session 对象持久化到一个存储设备中（钝化）；随 Session 对象从一个存储设备中恢复
* Servlet 规范中定义了两个特殊的监听器接口来**帮助 JavaBean 对象了解自己在 Session 域中的这些状态**：HttpSessionBindingListener接口和HttpSessionActivationListener接口 ，实现这两个接口的类**不需要 web.xml 文件中进行注册**

###### HttpSessionBindingListener该监听器自己是事件源也是监听器，不需要在web.xml文件中注册

* 实现了HttpSessionBindingListener接口的 JavaBean 对象可以感知自己被绑定到 Session 中和从 Session 中删除的事件
* 当对象被绑定到 HttpSession 对象中时，web 服务器调用该对象的 **void valueBound(HttpSessionBindingEvent event)** 方法
* 当对象从 HttpSession 对象中解除绑定时，web 服务器调用该对象的 **void valueUnbound(HttpSessionBindingEvent event)**方法

分别对应上面两个方法运行

###### HttpSessionActivationListener

* 实现了HttpSessionActivationListener接口的 JavaBean 对象可以感知自己被活化和钝化的事件
* 当绑定到 HttpSession 对象中的对象将要随 HttpSession 对象被钝化之前，web 服务器调用如下方法**sessionWillPassivate(HttpSessionBindingEvent event)** 方法
* 当绑定到 HttpSession 对象中的对象将要随 HttpSession 对象被活化之后，web 服务器调用该对象的 **void sessionDidActive(HttpSessionBindingEvent event)**方法

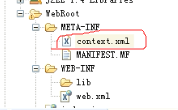
<Context>

<Manager className="org.apache.catalina.session.PersistentManager" maxIdleSwap="1">

<Store className="org.apache.catalina.session.FileStore" directory="it315"/>

</Manager>

</Context>

这个配置文件直接卸载META-INF里面(在tomcat文档有说明)

这个工程发布，服务器将把这个文件拷贝到localhost对应的工程目录下面。

###### 注意：以上活化、钝化是对象到文件夹，因此需要继承serializable

##### 回顾前21。

#### 第二十二、第二十三章

案例（未看）

#### 第二十四章

##### 泛型

* 泛形的基本术语，以ArrayList<E>为例：<>念着typeof
  + ArrayList<E>中的E称为类型参数变量
  + ArrayList<Integer>中的Integer称为实际类型参数
  + 整个称为ArrayList<E>泛型类型
  + 整个ArrayList<Integer>称为参数化的类型ParameterizedType

##### 声明泛型

public static <T> void doxx(T t); 声明一个泛型

public static <K,V> V getValue(K key) { return map.get(key);}//声明2个泛型

如果一个类多处都要用到同一个泛型，这时可以把泛形定义在类上（即类级别的泛型），语法格式如下：

public class GenericDao<T> {

private T field1;

public void save(T obj){}

public T getId(int id){}

}

注意，静态方法不能使用类定义的泛形，而应单独定义泛形。

##### 泛型的高级应用——？通配符

当接收的类型不确定时，可以使用?通配符，?可以指向任何类型，如下所示：

void print (Collection<?> c) {//Collection<?>(发音为:"collection of unknown")

for (Object e : c) {

System.*out*.println(e);

}

}

注：只要使用到了？通配符，就不能调用和泛型有关的方法，如集合的add方法（api的这个方法参数是泛型add(? e)）

此种形式下需要注意的是：由于?指向的是一种不确定的类型，因此不能调用与类型相关的方法，例如add()方法。

总结：使用?通配符主要用于引用对象，使用了?通配符

##### 有限制的通配符

限定通配符的上边界

正确：Vector<? extends Number> x = new Vector<Integer>();

错误：Vector<? extends Number> x = new Vector<String>();

限定通配符的下边界：

正确：Vector<? super Integer> x = new Vector<Number>();

错误：Vector<? super Integer> x = new Vector<Byte>();



T必须是Serializable和Cloneable的子类，T实现这2个对象

public void add(List<? extends Number> list){

list.add(1000);//只要有?就不能调用与类型相关的方法

}这个代码会报错，因为你1000有可能是存入到byte中，超出byte大小，因此报错

##### 注解annotation

* 三个基本的 Annotation:
  + @Override: 限定重写父类方法, 该注解只能用于方法
  + @Deprecated: 用于表示某个程序元素(类, 方法等)已过时
  + @SuppressWarnings: 抑制编译器警告.

##### 注解使用@interface声明

public @interface People{

String name() default ljh; //可以使用默认值

int age();

Gender gender() default Gender.male; //枚举性别

Class class() default String.class;

MyAnnotation2 my2() default @MyAnnotation2(name=”111”);//类型也是注解，并且这个注解中有个变量name

String[] ss() default {“11”,”22”};//一维数组

}

//支持8种基本类型，枚举以及其他注解，每个变量后面（）；

还能接收Class（Class对象！）

以及以上对象的一维数组

（Invalid type List<String> for the annotation attribute People.s; only primitive type, String, Class, annotation, enumeration are permitted or 1-dimensional arrays thereof）

注解还有一个简便方式：String value();使用这个属性在使用注解的时候不需要指定

@People(“ljh”)

public void test(){

}

//假如people中有String value(),如果还有其他属性String name()就不能直接这样使用了，必须：

@People(value=”ljh”,name=””)

##### JDK 的元 Annotation

* 元 Annotation指修饰Annotation的Annotation。JDK中定义了如下元Annotation：
* @Retention: 只能用于修饰一个 Annotation 定义, 用于指定该 Annotation 可以保留的域, @Rentention 包含一个 RetentionPolicy 类型的成员变量, 通过这个变量指定域。
  + RetentionPolicy.CLASS: 编译器将把注解记录在 class 文件中. 当运行 Java 程序时, JVM 不会保留注解. 这是默认值
  + RetentionPolicy.RUNTIME:编译器将把注释记录在 class 文件中. 当运行 Java 程序时, JVM 会保留注解. 程序可以通过反射获取该注释
  + RetentionPolicy.SOURCE: 编译器直接丢弃这种策略的注释

一个java类具有如下三种状态：源代码----》被编译后的class---》被加载到内存执行（运行时），以上Retention就相当于指定注解失效，比如RetentionPolicy.SOURCE:只在源代码级别上有，被编译后就没了

默认是Class级别！！！注意自己写的的时候不指定Runtime，那么将获取不到！

* **@Target**：指定注解用于修饰类的哪个成员. @Target 包含了一个名为 value，类型为ElementType的成员变量。（默认是所有地方都能用）
* @Documented: 用于指定被该元 Annotation 修饰的 Annotation 类将被 javadoc 工具提取成文档.
* @Inherited: 被它修饰的 Annotation 将具有继承性.如果某个类使用了被 @Inherited 修饰的 Annotation, 则其子类将自动具有该注解

Target（ElementType.TYPE） ElementType源码：

public enum ElementType {

/\*\* Class, interface (including annotation type), or enum declaration \*/

TYPE,

/\*\* Field declaration (includes enum constants) \*/

FIELD,

/\*\* Method declaration \*/

METHOD,

/\*\* Parameter declaration \*/

PARAMETER,

/\*\* Constructor declaration \*/

CONSTRUCTOR,

/\*\* Local variable declaration \*/

LOCAL\_VARIABLE,

/\*\* Annotation type declaration \*/

ANNOTATION\_TYPE,

/\*\* Package declaration \*/

PACKAGE

}

##### 自定义解析注解对象注入(方法及属性的注解案例)

@Retention(RetentionPolicy.*RUNTIME*)

**public** **@interface** DbInfo {

String driver();

String url();

String username();

String password();

}

///////---------------------------

**public** **class** JdbcUtils {

**private** **static** String *driver*;

**private** **static** String *url*;

**private** **static** String *username*;

**private** **static** String *password*;

**static**{

**try**{

//解析注解，获取注解配置的信息

Method method = JdbcUtils.**class**.getMethod("getConnection", **null**);

DbInfo info = method.getAnnotation(DbInfo.**class**);

*driver* = info.driver();

*url* = info.url();

*username* = info.username();

*password* = info.password();

}**catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

@DbInfo(driver="com.mysql.jdbc.Driver",url="jdbc:mysql://localhost:3306/bookstore",username="root",password="root")

**public** **static** Connection getConnection(){

System.*out*.println(*driver*);

System.*out*.println(*url*);

**return** **null**;

}

**public** **static** **void** main(String[] args) {

JdbcUtils.*getConnection*();

}

}

##### 动态代理（拦截）

被代理的对象需要实现一个接口

Proxy.*newProxyInstance*(LiyuchunProxy.**class**.getClassLoader(),chunchun.getClass().getInterfaces(),**new** InvocationHandler(){ //固定写法，传入一个代理对象的类装载其，被代理对象的接口，以及需要执行的方法

/\*

\* proxy:把代理对象自身传递进来

\* method:代表当前调用的方法

\* args:调用方法的参数

\*

\*/

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

String methodname = method.getName();

**if**(methodname.equals("sing")){

System.*out*.println("拿1万刀来！！");

**return** method.invoke(chunchun, args); //这个returne返回值是被代理对象返回的

}**else** **if**(methodname.equals("dance")){

System.*out*.println("拿2万刀来！！");

**return** method.invoke(chunchun, args);

}**else**{

System.*out*.println("春哥不支持这个功能");

}

**return** **null**;

}

案例1.:使用代理技术实现web工程乱码

2.使用代理技术实现web全站压缩

3.注解+动态代理实现权限控制

说明：动态代理技术是基于接口的代理，如果一个被代理类没有接口。那么就要使用另外一套api----cglib。spring中aop就是先判断是否有接口，有，使用动态代理，没有就基于cglib构建子类去动态代理。（这套api本质是通过构建被代理对象的子类去实现代理。）

如果一个类是final，同时又没有接口，那么没法使用代理

#### 第二十五章（javamail j2ee13种技术之一）

##### 发送邮件

smtp 常用端口25（邮件发送协议）

Pop3 常用端口110（邮件接收协议）

##### SMTP协议

* + - ehlo 主机名
    - auth login //接下来输入的经过base64编码后的用户名和密码
    - mail from:[itcast1111@sohu.com](mailto:itcast1111@sohu.com) //指定发送地点
    - rcpt to:[itcast1111@sina.com](mailto:itcast1111@sina.com)//指定接收地址
    - Data .号代表邮件内容的结束
    - quit

telnet localhost 25(以下不能有任何输错！！！否则从新开始！使用易邮邮件服务器)

220 ESMTP 易邮邮件服务器 5.2.2004.02.18 SMTP Service Ready

ehlo ljh

250-AUTH=LOGIN

250 AUTH LOGIN

auth login

334 VXNlcm5hbWU6

YWFh //base64后的编码，对应账号密码是：aaa 123

334 UGFzc3dvcmQ6

MTIz

235 OK

mail from: [aaa@ljh.com](mailto:aaa@ljh.com) //发送地址

250 OK

rcpt to: [bbb@ljh.com](mailto:bbb@ljh.com) //接收地址

250 OK

data

354 send the mail data, end with .

from: [aaa@ljh.com](mailto:aaa@ljh.com) //发送地址

to: [bbb@ljh.com](mailto:bbb@ljh.com) //接收地址

subject: test //主题

//一定要有空行

aaaaaaaaaaaaaaaaaaaa

. //.表示结束

250

quit //结束邮件发送

221 SMTP SERVICE CLOSED //表示发送成功

##### POP3协议

* + - user<SP>username<CRLF>
    - pass<SP>password<CRLF>
    - stat<CRLF> 返回邮箱的统计信息
    - list<SP>[msg#]<CRLF>返回某一封邮件的统计信息
    - retr<SP>msg#<CRLF> 最重要的一个命令 //retr 2 现实第二封邮件
    - quit<CRLF>
    - 使用POP3协议手工接收邮件

telnet localhost 110 （注意这里端口是110）

+OK 易邮邮件服务器 5.2 POP3 Service Ready

user bbb //登录用户名

+OK welcome here

pass 123 //登录密码

+OK

stat //返回邮箱统计信息

+OK 2 641

list //列出邮件

+OK 2 641

1 420 //服务器返回（这里总共有2封邮件，第一封大小420字节

2 221 //，第二封221字节）

. //这里有个点

retr 2 ///打开第二封邮件

+OK 221 octets

Return-Path: <aaa@ljh.com>

Received: from ljh (unknown [127.0.0.1])

by ljh.com with CMailServer 5.2 SMTP; Wed, 22 Jul 2015 21:13:14 +0800

from: <aaa@ljh.com>

to: <bbb@ljh.com>

subject: test

aaaaaaaaaaaaaaaaaaaa

.

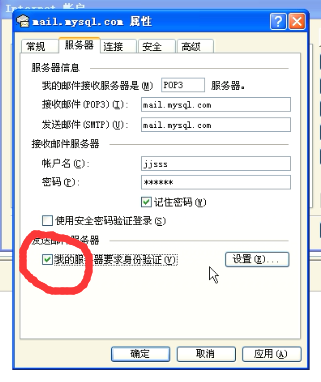
##### 创建邮件

* + RFC822文档规定了如何编写一封**简单**邮件
    - 邮件头和邮件体，两者使用**空行**分隔
    - 邮件头
      * from字段
      * to字段
      * subject字段
      * cc、bcc字段 //cc抄送 bcc密送
    - 邮件体
      * 邮件内容
  + 利用RFC822文档漏洞发送假冒邮件

##### 配置邮件服务器：

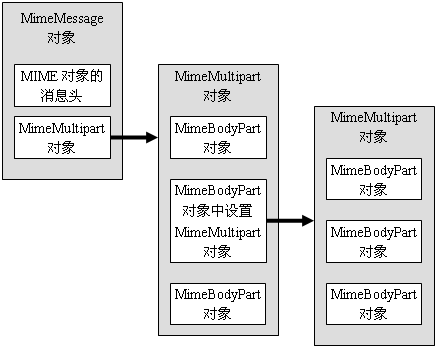
需要知道公司分配给你的邮件账号密码，以及邮件服务器地址，可以使用nslookup查找（nslookup –qy=mx ljh.com） qt : querytype mx :mail exchanger

配置完成之后需要在账号中设置（否则可能无法发送邮件）



##### javamail

邮件组成形式：



* MimeMessage类表示整封邮件。
* MimeBodyPart类表示邮件的一个MIME消息。
* MimeMultipart类表示一个由多个MIME消息组合成的组合MIME消息。

导入activation.jar 以及mail.jar

**public** **static** **void** main(String[] args) **throws** Exception {

//创建邮件

Session session = Session.*getDefaultInstance*(**new** Properties());

MimeMessage message = **new** MimeMessage(session);

message.setFrom(**new** InternetAddress("aaa@flx.com"));

message.setRecipient(Message.RecipientType.*TO*, **new** InternetAddress("bbb@flx.com"));

message.setSubject("测试");

//创建bodypart封装正文

MimeBodyPart text = **new** MimeBodyPart();

text.setContent("这是中文邮件a<img src='cid:1.jpg'>", "text/html;charset=UTF-8");

//创建bodypart封装图片

MimeBodyPart image = **new** MimeBodyPart();

image.setDataHandler(**new** DataHandler(**new** FileDataSource("src/1.jpg")));

image.setContentID("1.jpg");

//创建bodypart封装附件

MimeBodyPart attach = **new** MimeBodyPart();

DataHandler dh = **new** DataHandler(**new** FileDataSource("src/光辉岁月.mp3"));//这个使用的是j2ee中的jaf技术需要导入activation.jar

attach.setDataHandler(dh);

attach.setFileName(MimeUtility.*encodeText*(dh.getName())); //只要调用了setFileName就默认设置了content-disposition字段，成为了附件

//描述数据关系

MimeMultipart content = **new** MimeMultipart();

content.addBodyPart(text);

content.addBodyPart(image);

content.setSubType("related"); //关联关系，在文本中嵌入图片（这种方式图片存入邮件中）

MimeBodyPart mbp = **new** MimeBodyPart(); //因为MimeMultipart容易不能放MimeMultipart所以需要封装城MimeBodyPart

mbp.setContent(content);

MimeMultipart mm = **new** MimeMultipart();

mm.addBodyPart(mbp); //邮件由MimeMultipart组成，MimeMultipart容易中再放 MimeBodyPart

mm.addBodyPart(attach);

mm.setSubType("mixed"); //附件使用mixed

message.setContent(mm);

message.saveChanges();

message.writeTo(**new** FileOutputStream("c:\\1.eml"));

}

**可以获取用户信息的代码（比如是否是活跃用户验证等。）**

**public** **static** **void** main(String[] args) **throws** Exception {

//创建邮件

Session session = Session.*getDefaultInstance*(**new** Properties());

MimeMessage message = **new** MimeMessage(session);

message.setFrom(**new** InternetAddress("aaa@flx.com"));

message.setRecipient(Message.RecipientType.*TO*, **new** InternetAddress("bbb@flx.com"));

message.setSubject("test");

//servlet 1.jpg 这种方式是不安全的，可以获取用户信息，图片不是嵌入邮件中的

message.setContent("aaaaa<br/><img src='http://localhost:8080/mail/1.jpg'>", "text/html");

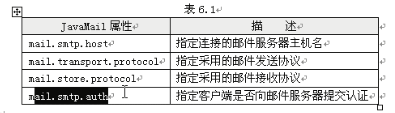
message.saveChanges();

message.writeTo(**new** FileOutputStream("c:\\1.eml"));

}

**完整的发送邮件代码**

**public** **static** **void** main(String[] args) **throws** MessagingException, UnsupportedEncodingException {



Properties prop = **new** Properties();

prop.setProperty("mail.smtp.host", "localhost");

prop.setProperty("mail.transport.protocol", "smtp");

prop.setProperty("mail.smtp.auth", "true");

javax.mail.Session session = javax.mail.Session.*getInstance*(prop);

session.setDebug(**true**); //打印与邮件服务器交互的过程（不是必须的，尽量不要使用太长了）

Message message = *createMessage*(session);

Transport ts = session.getTransport();

ts.connect("aaa", "123"); //连接服务器的用户名密码 这个方法在这里会报错！！因为javamail已经加入到j2ee技术中，你创建的web工程已经有了javamail的包，但是版本过低，因此你需要在prefacerence中输入libremove默认的javamail和jaf的包

ts.sendMessage(message,message.getAllRecipients());

ts.close();

}

**private** **static** Message createMessage(Session session) **throws** AddressException, MessagingException, UnsupportedEncodingException {

//创建邮件

MimeMessage message = **new** MimeMessage(session);

message.setFrom(**new** InternetAddress("aaa@flx.com"));

message.setRecipient(Message.RecipientType.*TO*, **new** InternetAddress("bbb@flx.com"));

message.setSubject("test");

message.setContent("aaaaaaaaaaaa", "text/html");

message.saveChanges();

**return** message;

}

在邮件发送过程中需要连接邮件服务器，这个动作是很慢的，因此需要单独写发送邮件的线程。

##### Mx和A记录

A记录：包括邮件服务器主机名 和 ip地址（A记录全称Address记录，又称IP指向）

（邮件服务在相互发送的时候会查询dns服务器这个A记录，如果发送放是A记录对应的主机名则不需要用户名密码登录这个接收服务器。 因此ehlo在这个时候后面跟的必须是自己的服务器命）在自己搭建邮件服务器的时候需要购买域名，然后服务商会提供一个web界面，你需要在这个web界面中添加一个A记录

还要注册一个MX记录（邮件服务器 和 ip），让对方找到你的邮件服务器。

邮件发送过多，需要进入邮件白名单，否则各大邮件服务会屏蔽你的地址。

##### 类加载器



BootStrp这个加载器是java引导加载器c语言写的，打印不出来。

可以使用syso(Object.class.getClassLoader())打印查看类装载器

父类委托机制：加载一个类的时候优先从Bootstrap开始，如果没有在从ExtClassLoader加载，没有再从AppClassLoader

* classloader 加载类用的是**父类委托机制**。
* **父类委托机制**：先让parent（父）类加载器 寻找，**只有在parent找不到的时候才从自己的类路径中去寻找**。
* **类加载还采用了cache机制：**如果 cache中保存了这个Class就直接返回它，如果没有才从文件中读取和转换成Class，并存入cache，这就是为什么修改了Class但是必须重新启动JVM才能生效,并且类只加载一次的原因。

注：tomcat没有采用父类委托机制。在web工程中可能出现类的定义和类的新建用的是不同的.class文件

<http://blog.csdn.net/dc_726/article/details/11873343>

#### 第二十六章

##### struts1入门程序

###### .写jsp

<form action="${pageContext.request.contextPath }/register.do" method="post">

用户名：<input type="text" name="username"><br/>

密码：<input type="password" name="password"><br/>

电子邮箱：<input type="text" name="email"><br/>

<input type="submit" value="注册">

</form>

###### .请求为了交给struts处理，这时需要导入struts开发包

###### 在web.xml文件中配置struts的actionServlet处理所有.do的请求

<servlet>

<servlet-name>ActionServlet</servlet-name>

<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

<init-param>

<param-name>config</param-name>

<param-value>/WEB-INF/struts-config.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>ActionServlet</servlet-name>

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

</servlet-mapping>

###### .在WEB-INF目录中加入struts的工作配置文件：struts-config.xml

<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"

"http://struts.apache.org/dtds/struts-config\_1\_3.dtd">

<struts-config>

</struts-config>

###### 在struts的工作配置文件：struts-config.xml，配置actionServlet收到register请求后，调用ReigsterAction处理，并把请求中的数据封装到registerForm中。

<struts-config>

<form-beans>

<form-bean name="registerForm" type="cn.itcast.web.formbean.RegisterFormBean"></form-bean>

</form-beans> //提交到action的数据将自动封装到这个bean中，bean在下面的name属性对应

<action-mappings> //这里提交的path后面不要跟.do默认就是这个后缀

<action path="/register" name="registerForm" type="cn.itcast.web.action.RegisterAction">

<forward name="message" path="/message.jsp"/> //配置执行完后这个action跳转界面

</action>

</action-mappings>

<struts-config>

###### 编写封装数据的RegisterFormBean和处理请求的RegisterAction

RegisterFormBean registerForm = (RegisterFormBean) form;

String username = registerForm.getUsername();

String password = registerForm.getPassword();

String email = registerForm.getEmail();

try{

System.out.println("向数据库中注册" + username + "用户");

request.setAttribute("message", "注册成功！！");

}catch (Exception e) {

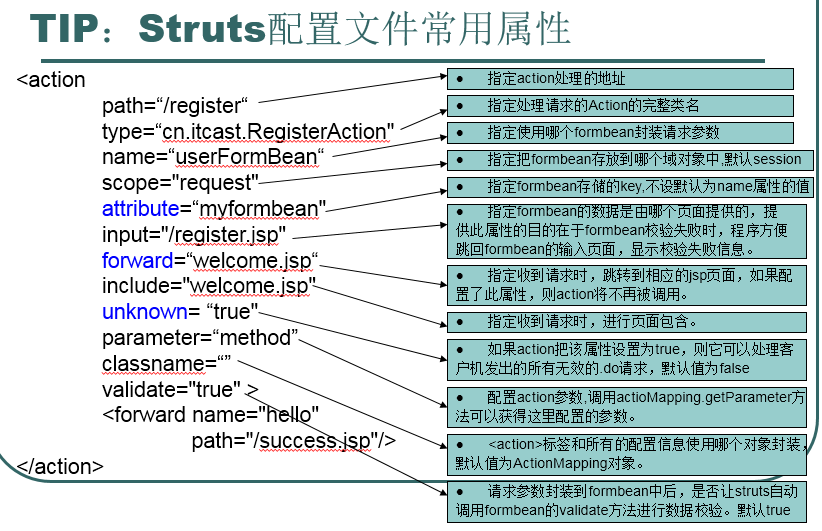
request.setAttribute("message", "注册失败！！");

}

return mapping.findForward("message");

7.编写显示全局消息的message.jsp页面

##### struts常用配置：



##### DispatchAction处理一类请求

使用这个类不需要重写execute()方法，因为默认已经实现了。

前台代码如：

struts文件配置：parmeter需要找的类的方法的前缀，如上面使用method指定。



和下面比： 这个配置文件简单，但是不能对每个连接封装单独的formbean

##### MappingDispatchAction

前台代码如：



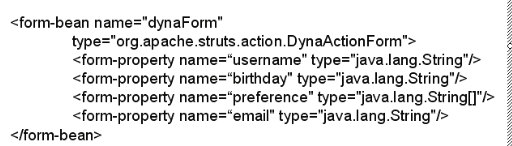
struts文件配置：



和上面比：直观，但是配置文件复杂，能单独封装formbean

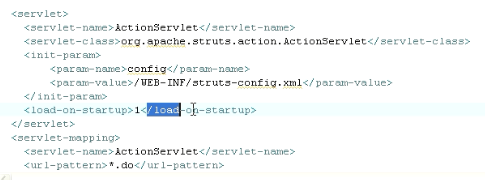
##### 使用DynaActionForm配置formbean

声明一个dynaForm的bean



##### Validator插件

##### Struts使用需要在web.xml文件中配置：





#### 第二十七章

##### struts1和2区别

struts1是基于servlet

struts2是基于filter

###### Struts 1

struts1中 formbean有验证方法： ActionErrors validate（），如果出错则会通过struts的配置文件中action-mapping中配置的input/\*\*\*.jsp页面中使用

<html:errors>配置文件来读取validate中返回的ActionErrors对象。这个对象是封装在request中的。(在提交formbean到ActionServlet的时候，会自动将错误验证结果存储到request中)

Struts中核心：ActionServlet工作：

1，接收请求，寻找formbean，如果formbean没有实例化，先实例化formbean。2，组装formbean，formbean是否需要校验，需要则执行ActionErrors validate()方法，将ActionErrors返回值给ActionServlet。

3，ActionServlet判断ActionErrors有效性，将ActionErrors封装到request域中，跳转到input指定的jsp

4，到Action，负责Action的实例化。调用execute（ActionMapping mp,ActionForm a,request,response）方法，返回ActionForm对象到ActionServlet对象中负责解析。解析完成后又天转到execute中return new ActionForward(“/xxx.jsp”)；

Struts1中ActionServlet任务过重。不易于维护。

注：Struts1中action和spring的bean单例是不同的。struts1的action是单例的是设计模式单例，spring单例是通过map集合来达到单例效果。Struts1中所有相同的处理请求都是同一个单例的action处理。

###### Struts2

Struts2 分离关注思想（来自webflow，类似spring aop）

##### ddt文件添加到eclipse中方法。

#### 第四十二章

##### Spring入门

spring包-denpendence表示含所有依赖包

需要引入spring.jar以及日志common-logging包

struts2:web

hibernate:持久化

spring:业务层.管理bean的,容器.List Map Set.

##### 体验spring:

1.创建java项目.

2.引入spring的类库.

${spring解压目录}/dist/spring.jar

${spring解压目录}/lib/jakarta-commons/commons-logging.jar

2'.创建业务类.

/\*\*

\* GreetingService

\*/

public class GreetingService {

private String greeting ;//get/set

private String greeting2 ;//get/set

/\* byeService \*/

private ByeService bs ;//get/set

public void sayGreeting(){

bs.sayBye();

}

}

/\*\*

\* ByeService

\*/

public class ByeService {

private String bye ;

public String getBye() {

return bye;

}

public void setBye(String bye) {

this.bye = bye;

}

public void sayBye(){

System.out.println(bye);

}

}

3.配置spring的配置文件.src/applicationContext.xml

schema:${spring解压目录}\docs\reference\html\_single\index.html\3.2.1.1（在帮助文档的这个章节的可以找到配置文件的头）

<?xml version="1.0"?>

<beans xmlns="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-2.5.xsd">

<!-- 欢迎服务 -->

<bean id="greetingService" class="cn.itcast.spring.service.GreetingService">

<!—id配置是唯一的，一般以类名头字母小写 其实bean后面还有一个默认的scope=singleton属性，表bean是单例，spring中ApplicationContext实例的时候会创建单例bean，因为sping容易其实是map<key,value> key其实就是配置bean的id -，使用scope = singleton不管你调用多少次都是使用同一个对象，因为该单例是饿汉式，饿汉式不能算是单例，但是实现了单例的效果（ps：真正的单例是懒汉式），scope=“prototype”表示原型bean，原型bean就是每次调用ac.getBean(“XXXX”)都是new出来的新实例->

<property name="greeting">

<value>hello world</value> //name、value两种配置方式都可以

</property>

<property name="greeting2" value="tom" />

<property name="bs" ref="byeService" /> //ref查找其他bean

</bean>

<!-- 欢送服务 -->

<bean id="byeService" class="cn.itcast.spring.service.ByeService">

<property name="bye">

<value>later</value>

</property>

</bean>

</beans>

4.创建app.

ApplicationContext ac = new ClassPathXmlApplicationContext(

"applicationContext.xml");

GreetingService gs = (GreetingService) ac.getBean("greetingService");

gs.sayGreeting();

gs.sayGreeting2();

ByeService bs = (ByeService) ac.getBean("byeService");

bs.sayBye();

spring:

ioc:inverse of control,反转控制.获得依赖对象的方式被反转了.

1.new.(spring负责对象实例化)

2.组装对象的出发点是反的.

DI:dependency injection,依赖注入.

aop:aspect oriented program,面向方面编程.

oop:面向对象编程.

soa: 面向服务的体系结构

GreetingService gs = new ...();

gs.setByeService(bs);

ByeService bs = new ...();

gs.sayGreeting();

##### 三种经常用到的实现：

###### 1.ClassPathXmlApplicationContext:从类路径中加载。

###### 2.FileSystemXmlApplicationContext:从文件系统加载。

###### 3.XmlWebApplicationContext:从web系统中加载。

##### BeanFactory以及spring单例

BeanFactory:实例化bf时,不会实例化任何bean,只有getBean的时候,才实例化相关的bean.节省资源.（一般在移动设备上就需要使用）

ApplicationContext:实例化ac时,实例化单例bean.，默认没有配置scope=protoytpe的都是单例bean

懒汉式:

public class A{

private A(){}

public static A instance = null ;

public static A getInstance(){

if(instance == null){

instance = new A();

}

return instance ;

}

}

这个单例，每次使用的是不同的对象

饿汉式:

public static A instance = new A();

每次都获得同一个对象

spring单利bean,（由sping来维护，因为就一个实例）见上spring配置文件

每次调用都new出来的，对应的bean的叫做prototype(原型bean).（原型bean每次都是新的对象，由用户来维护，所以一般尽量不用，因为用户经常忘记如何为何，所有spring一般不用原型bean，但是比如Struts2中的action就必须要使用原型bean）

ac.getBean("bs"); == new ByeService();

##### 生命周期:

1.new

2.set DI 设置依赖注入

3.BeanNameAware:bean名关注 获得你配置的id

4.BeanFactoryAware:bean工厂关注 获得工厂自身（获得bean容器可以实现这个接口）

5.BeanPostProcessor.beforeInitialization(); bean处理器

5'.调用InitializingBean.afterPropertySet()方法.(让bean实现初始化方法有两种方式，一种是配置init-method,一种是实现这个接口提供的这个方法。5’和6是等效的，5‘会先执行。但是推荐6)

6.init-method(配置的初始化方法)

7.BeanPostProcessor.afterInitialization();

经过以上bean就可以使用了

在容器关闭后

8.调用DisposableBean的destory()方法

9，调用定制的销毁方法。



##### BeanPostProcessor处理器，bean后处理器（对每个bean都处理，类似filter）

该处理器不会对自身处理。

会对每个bean进行处理。类似过滤器，会对每个bean处理

spring在ApplicationContext的bean中可以配置init-method，这个初始化方法在BeanPostProcessor.beforeInitialization();之后BeanPostProcessor.afterInitialization();

之后被调用（上面生命周期）

配置destory-method，销毁时执行，但是这个销毁不是类的销毁时执行，而是ApplicationContext关闭时执行，Ac本身没有destory()或者close方法，但是他的子类有如ClassPathApplicationContext.destory();(该类提供的destory()方法其实就是close()方法,等效)

spring()框架. 代码的入侵.（耦合spring是非侵入的）

框架一般有：

1.高侵入性

2.低侵入(迷惑人的，没有这个)

3.非侵入行

jee:jsp servlet jpa jta ejb(enterprise javabean),jndi

pojo:plain old java object.

##### spring初始化bean源码分析

有DefaultListableFactory(AbstractAutowireCapableBeanFactory).doCreateBean(String ,RootBeanDefinition,object[])中第一个参数是beanname第二个是final RootBeanDefinition mdb其实就是一个bean的上下文配置文件中配置信息封装类具体如下：

BeanDefinition:bean的定义,spring对bean的信息的封装.（关键类）

Root bean: class [cn.itcast.spring.service.GreetingService]; scope=singleton; abstract=false; lazyInit=false;

autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null;

factoryMethodName=null; initMethodName=ini; destroyMethodName=release; defined in class path resource

[applicationContext.xml]

通过如下代码实例配置的对象：

Class clazz = Class.forName("cn.itcast.spring.service.GreetingService");

clazz.newInstance();

底层就是反射

##### Scope配置：

scope拥有的值:prototype singleton request session global\_session

###### 基本装配-scope

prototype、singleton、request session、global-session

spring中的bean缺省情况下是单例模式。始终返回一个实

例。若想返回不同的实例的话需要定义成原型模式。

bean的singleton属性告诉上下文该bean是否为单例的。缺

省为true。若为false的话，为原型bean。

<bean id="foo" class="...Foo" singleton="false"/>

在spring2.5以后使用scope来指定域，并且不推荐singletone=true|false配置。

<!– spring2.5 -->

<bean scope="prototype|single|request|session|global-session">

如果配置scope是reques或者session，还需要其他配置才生效。

servlet2.4以前：

<web-app>

<listener>

<listener-class>org.springframework.web.context.request.RequestContexListener</>

</listener>

</web-app>

如果是servlet2.4以前规范的web容易，不支持Listener规范，那么只能使用Filter配置：

<filter>

<filter-name>requestContextFilter</filter-name>

<filter-class> org.springframework.web.context.request.RequestContexListener </filter-class>

</filter>

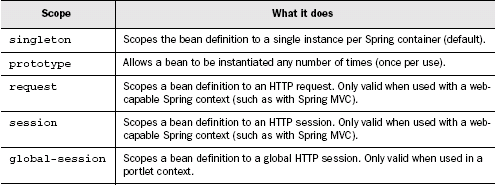
<filter-mapping>

<filter-name><requestContextFilter</filter-name>

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

</filter-mapping>

spring3.0以后还支持自定义作用域、当使用spring MVC不需要手动去配置scope，SpringDispatcherServlet或DispatcherPortlet来拦截用户请求，则不需要这些额外配置。



###### 添加一个bean(续)

实例化与销毁

spring实例化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法。

<bean class="Foo" init-method destory-method>

<!-- spring2.5-->

@PostConstruct public void ini(){…}

@PreDestroy public void destroy(){…}

使用了以上2个注解来设置实例化和销毁，需要在spring配置文件中配置注解后处理器：

<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor">

以注解方法实现声明周期

###### 添加一个bean(续)

spring也提供了两个接口来实现相同的功能：

InitializingBean和DisposableBean.

InitializingBean接口提供了一个afterPropertiesSet()方法。

DisposableBean接口提供了destroy().

不推荐使用该接口，它将你的bean和springAPI邦定在一起。

###### 基本装配

1. 基本装配

配置bean的简单属性，基本数据类型和string。

<bean id="foo" class="...Foo">

<property name="name">

<value>tom</value>

</property>

</bean>

1. 引用其它bean

<bean id="foo" class="...Foo">

<property name="name">

<ref bean="bar">

</property>

</bean>

<bean id="bar" class="...Bar">

</bean>

1. 内部bean

<bean id="foo" class="...Foo">

<property name="bar">

<bean class="...Bar">

</property>

</bean>

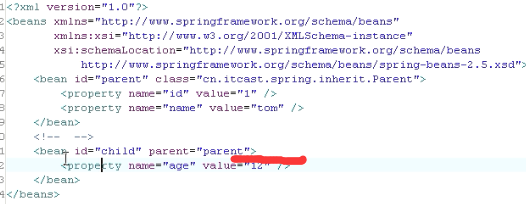
这种方式的缺点是你无法在其它地方重用这个bar实

例，原因是它是专门为foo而用。所有Bar中也没必要写id，有点类似内部类

1. 继承配置

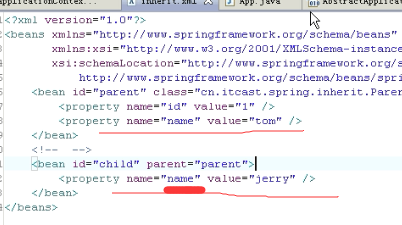
* 继承
* 继承配置
* 覆盖父 Bean配置
* 可以设置 <bean> 的abstract 属性为 true, Spring 不会实例化该Bean
* 有些属性不会被继承. 比如: autowire, abstract 等.
* 子Bean 指定自己的class. 但此时 abstract 必须设为 true

如：有Parent类有id、name属性，有Child类有id、name属性以及age属性，Parent和Child没有extends关系！如下配置：



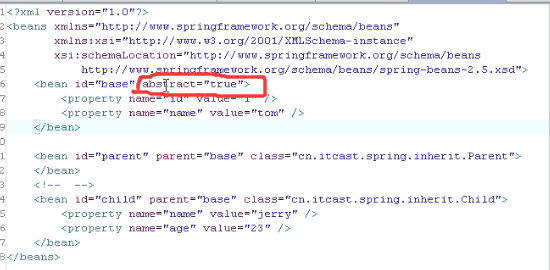
如果没有指定class，那么child会自动把parent的class属性继承过来，但是parent没有age属性，则运行会报错。

重载配置（重载name属性）：



如果设置Child的class，则正确。这样很多bean共有的属性可以简化配置

如下：



如果不增加abstract=true会报错：没有指定的类。

1. list装配集合以及util:list

若bean的属性是集合类型，按如下处理：

装配List和数组：

<property name="barlist">

<list>

<value>bar1</value>

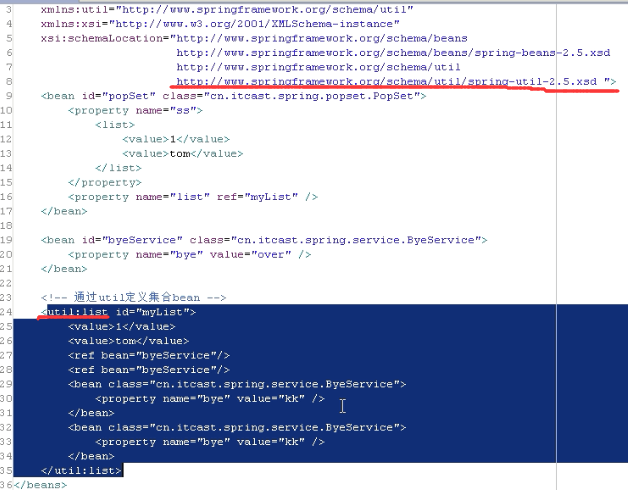
<ref bean="bar2"/>

</list> //list表示数组和集合List

</property>

<!– spring2.5 -->

<util:list>以下假如一个相同的list被多次引用可以使用util：list配置：



1. set装配以及util：set

装配set：

<property name="barlist">

<set>

<value>bar1</value>

<ref bean="bar2"/>

</set>

</property>

set使用方法和list一样，不同的是对象被装配到set

中，而list是装配到List或数组中装配

<!– spring2.5 -->

<util:set>



1. map装配以及util：map

装配map：

<property name="barlist">

<map>

<entry key="key1" value="bar1" />

<entry key="key2 value-ref="xxx" />

</map>

</property>

key值必须是string的,key-ref可以是其他bean

<!– spring2.5 -->

<util:map>



<struts>

<package namepace extends="">

1. Properties装备：

和map类似，不同的是Properties的key和value都是String

装配Properties：

<property name="barlist">

<props>

<prop key="key1">bar1</prop>

<prop key="key2">bar2</prop>

</props>

</property>

<!-- spring2.5 -->

<util:properties>

1. 设置null：

<property name="barlist">

<null/>

</property>

##### bean的构造函数：

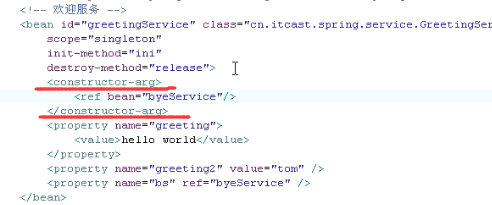
1. Set注入的替代：

set注入是一种直接方式，缺点是它假设了所

有的可变属性都可以通过set方法访问到。例如有些属性在创建时设置一次，以后不再改变。

替代方式是通过构造函数设置一些属性值。

以上也就是说明，当你的bean是通过set方法注入(一般写代码都是通过set方法注入)的时候，如果你没在xml文件中配置这个属性的实例对象，用户没有调用到这个bean的这个属性，程序不回报错。但是如果用户调用到了，就会报错，因此要出现一个set属性替代方法，（也就是增强属性和类的引用关系，必须配置属性，那么就使用类的构造函数）



通过constructor-arg去指定类GreetingService构造函数的参数值。（如果bean类没有空的构造函数，就必须使用这个属性去指定类构造函数的参数）

1. 通过构造函数注入依赖

<bean id="foo" class="...Foo">

<constructor-arg>

<value>42</value>

</constructor>

</bean>

<bean id="foo" class="...Foo">

<constructor-arg>

<ref bean="bar">

</constructor>

</bean>

set注入的缺点是无法清晰表达哪些属性是必须的，哪些是可选

的，构造注入的优势是通过构造强制依赖关系，不可能实例化不

完全的或无法使用的bean。（可以使用这个来检查你的bean属性在xml配置文件中有没有被配置了）

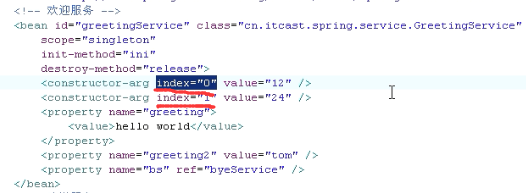
1. 解决构造函数参数不确定性

构造函数多或参数类型大都相同该如何处理？

spring并不是按照参数的顺序来配置参数的。

有两种方法来解决构造参数的不确定性：序号和类型。

<constructor-arg>有一个可选的index属性，可用来指定参数的顺序。



还有通过类型指定：



还有结合使用的：指定顺序和类型



##### Bean自动装配

<bean id="foo" class="...Foo" autowire="autowire type">

1. 有四种自动装配类型：

1.byName:寻找和属性名相同的bean,若找不到，则装不上。

2.byType:寻找和属性类型相同的bean,找不到,装不上,找到多个抛异常。

3.constructor:查找和bean的构造参数一致的一个或

多个bean，若找不到或找到多个，抛异常。按照参数的类型装配

4.autodetect: (3)和(2)之间选一个方式。不确定

性的处理与(3)和(2)一致。

<bean id="bar" class="Bar" autowire="byName"/>

<!-- spring2.5 -->

@Autowired 要使用这个自动装配注解还需要配置下面的处理器

<bean class="... AutowiredAnnotationBeanPostProcessor">

1. 处理自动装配的不确定性

使用byType和constructor自动装配时，若找到多个符合条件的bean，会报异常，因此最好的方式是不用自动组装。

spring通过构造函数参数注入依赖强化依赖关系.

自动装配:

1.byName:按照名称自动装配,寻找和bean的属性名相一致的bean的id.(bean必须有空的构造,通过set方法注入)

2.byType:按照属性的类型来自动装配,如果找到多个,抛异常.(bean必须有空的构造,通过set方法注入)

3.constructor:按照构造函数参数的类型自动装配,如果找不到或者找多个,都抛异常.

4.autodetact:自动检测,在(2)和(3)之间选择一个.

5.no.

6.default,跟<beans default-autowire属性>保持一致.在beans的标签上定义只是在bean的属性autowire前面加上default

1. 处理自动装配的不确定性

使用byType和constructor自动装配时，若找到多个符合条件的bean，会报异常，因此最好的方式是不用自动组装。

1. 混合使用手动和自动组装

<bean id="bar" class="...Bar"

autowire="byName">

<property name="cousedao">

<ref bean="somebean" />

</property>

</bean>

1. <context:annotation-config />

spring2.5提供了<context:annotation-config />配置.（使用了这个配置就不需要在使用注解的时候去配置相应的处理器了，因为默认会启动注解对应的处理器）

该配置可激活在类中探测到的各种注解,@Required @Autowire @PostConstrct @PreDestroy @Resource @EJB @PersistenceContext @WebServiceRef等等,（这些配置需要引入context这个xsd文件）

也可以选择为这些注解激活单独的后处理器.

例如:AutowiredAnnotationBeanPostProcessor(处理自动装配)

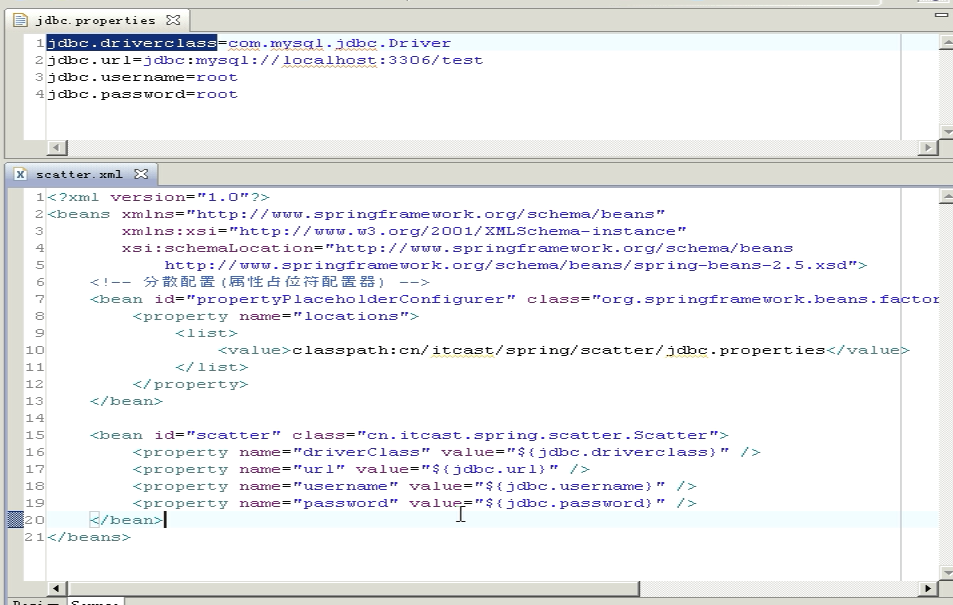
CommonAnnotationBeanPostProcessor(生命周期等)

注意:该标记不能激活事务处理.需要使用tx:annotation-driven

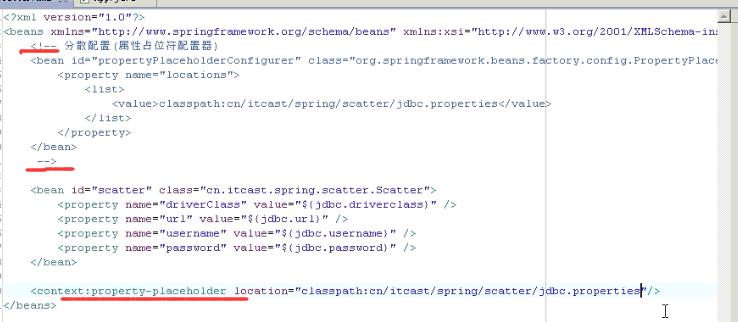
##### 分散配置:

把需要在上下文硬编码的属性拿到外部的属性文件中定义,让上下文从外部文件提取值.

当配置文件需要频繁修改，可以将这部分内容写在properties里面，如下，同时使用分配配置占位符配置器配置



还可以是用<context:property-placehoder location>简化配置。



* + 分散配置

将配置文件分成几个分散的配置文件。如数据源。

<bean class="...PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>jdbc.properties</value>

<value>securtiy.properties</value>

</list>

</property>

</bean>

<!– 名字空间配置(2.5) -->出现了简化的配置。

<context:property-placehoder location>

使用占位符变量代替bean装配文件中的硬编码配置。占位符采

${variable}形式。

<bean id="datasource"

　　　 class="...DriverManagerDataSource">

<property name="url">

<value>${database.url}</value>

</property>

......

</bean>

##### 后处理器

1. 使用spring的特殊bean

让spring特殊对待这些bean。使它们可以：

1.通过配置后加工bean，涉及到Bean和Bean工厂生命

周期。

2.改变依赖注入，将字符串转换成其它类型。

3.从属性文本装载信息，包括信息国际化。

4.监听并处理其它bean及spring发布的系统消息。

5.知道自己在spring中的唯一表识。

1. 对bean进行后处理

BeanPostProcessor接口提供机会来修改bean。

public interface BeanPostProcessor{

//Bean初始化(调用afterPropertiesSet()以及Bean的指

//定initmethod方法)之前被调用。

Object postProcessorBeforeInitialation(…);

//在初始化之后马上调用

Object postProcessorAfterInitialization(…);

}

1. 注册后处理Bean

如果使用的是Bean工厂,需要调用工厂的addBeanPostProcessor()来注册.

factory.addBeanPostProcessor(…);

如果使用的是上下文同其他Bean一样:

<bean id="" class="……" />

1. 对bean工厂进行后处理

　BeanPostProcessor在bean载入后，对bean进行一些后处理工作。而BeanFactoryPostProcessor在bean工厂载入所有bean的定义后，实例化bean之前，对Bean工厂做一些后处理工作。

　 public interface BeanFactoryPostProcessor{

public void postProcessorBeanFactory(…);

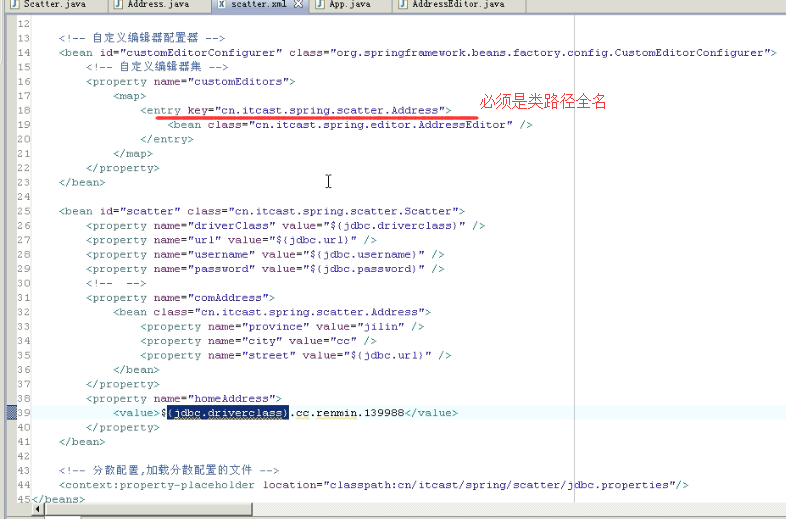
}

如果使用的是应用上下文不需要将他注册为Bean工厂后处理器。上下文会自动注册它。按照正常的Bean声明即可。

##### 自定义编辑器: (都是为了简化配置文件)

将字符串转换成相应的对象,。

假如有comAddress和homeAddress类中都有省份、城市、街道，单独配置就要每个都配置过去，如下配置文件的comAddress配置。可以使用自定义编辑器，来简化配置。如下homeAddress的配置，把多个属性配置在一个value里面，然后需要重新自定义编辑配置器，来通过·来分割，同时自定义配置还支持占位符。





##### 在eclipse中注册xsd文件：

在ide下注册spring中的schema文件.

${spring解压目录}\dist\resources\\*.xsd.

#### 第四十三章

##### aop:面向方面编程.不改变源代码,还为类增加新的功能.(代理)

切面:实现的交叉功能.

通知:切面的实际实现.

连接点:应用程序执行过程期间,可以插入切面的地点.

切入点:真正的将通知应用到目标程序中的地点,一定是连接点.切入点是连接点的子集.

引入:为类增加新的属性和方法.

目标对象:被通知的对象.

代理:把通知应用到目标对象以后,产生新的对象,该对象就称为代理对象.

织入:创建代理对象过程.

编译期织入:.java --> .class,需要特殊的编译器.

类装载期织入:将java字节码载入到jvm时,将通知织入.需要特殊的classloader.

运行期(runtime):

cglib:

aop alliance:aop联盟.定义aop的接口和规范

##### spring代理方案:

1.接口代理:jdk动态代理,创建更加松耦合的系统.

2.对类代理:cglib代理,final方法无法被代理.

##### spring aop编程:

1.aop alliance.jar(已经集成在spring.jar中) + cglib.

${spring解压目录}/lib/cglib/\*.jar

2.加入aspectj类库

${{spring解压目录}/lib/aspectj/\*.jar(aspectjrt.jar + aspectjweaver.jar)

3.创建接口和实现类

public interface WelcomeService {

public void sayName();

}

/\*\*

\* 目标类

\*/

public class WelcomeServiceImpl implements WelcomeService {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void sayName() {

System.out.println(name);

}

}

4.创建前置通知.

/\*\*

\* 前置通知(方法前通知)

\*/

public class MyMethodBeforeAdvice implements MethodBeforeAdvice {

public void before(Method method, Object[] args, Object target)

throws Throwable {

System.out.println("hello world");

}

}

5.配置文件.

<?xml version="1.0"?>

<beans xmlns="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-2.5.xsd"><!-- 前置通知(方法前通知) -->

<bean id="myMethodBeforeAdvice" class="cn.itcast.spring.aop.advice.MyMethodBeforeAdvice" />

<!-- 目标对象 -->

<bean id="welcomeServiceTarget" class="cn.itcast.spring.aop.service.WelcomeServiceImpl">

<property name="name" value="tom" />

</bean>

<!-- 代理对象 -->

<bean id="welcomeService" class="org.springframework.aop.framework.ProxyFactoryBean">

<!-- 代理接口集 -->

<property name="proxyInterfaces">

<list>

<value>cn.itcast.spring.aop.service.WelcomeService</value>

</list>

</property>

<!-- 拦截器名集 -->

<property name="interceptorNames">

<list>

<value>myMethodBeforeAdvice</value>

</list>

</property>

<!-- 指定目标对象 -->

<property name="target" ref="welcomeServiceTarget" />

</bean>

</beans>

6.App

ApplicationContext ac = new ClassPathXmlApplicationContext(

"cn/itcast/spring/aop/aop.xml");

WelcomeService ws = (WelcomeService) ac.getBean("welcomeServiceTarget");

ws.sayName();

public interface Pointcut{

ClassFilter getClassFilter();

MethodMatcher getMethodMatcher();

}

Pointcut:切入点

Advice:通知

Advisor:切入点通知,组合体,既包含通知又包含切入点.对原来的通知的包装,增加定义切入点功能

PointcutAdvisor{

Pointcut getPointcut();

Advice getAdvice();

}

引入通知:

1.定义引入通知.

Dao:data access object.数据(数据库的表数据)访问对象.

dto:data transfer object,数据传输对象. struts1(actionform jsp --> action)

集成dao.

1.引入数据源类库

${spring解压目录}/lib/c3p0/\*.jar

c3p0-0.9.1.2.jar

2.配置spring配置文件,链接数据源

insert:

conn = ds.getConn

conn.setAutocommit(false);

String sql = "insert into customers(name,age) values(?,?)" ;

ppst = conn.preparedStatement(sql)

ppst.setString(1,"tom");

//...

ppst.executeUpdate();

conn.commit();

ppst.close();

conn.close();

update:

conn = ds.getConn

conn.setAutocommit(false);

String sql = "update customers set name = ?,age=? where id= ?" ;

ppst = conn.preparedStatement(sql)

ppst.setString(1,"tom");

//...

ppst.executeUpdate();

conn.commit();

ppst.close();

conn.close();

//select

conn = ds.getConn

String sql = "select \* from customers" ;

ppst = conn.preparedStatement(sql)

ppst.setString(1,"tom");

//...

ppst.executeUpdate();

conn.commit();

ppst.close();

conn.close();

#### 第四十四章

##### spring和hibernate整合

1.引入hibernate类库.

antlr-2.7.6.jar

commons-collections-3.1.jar

dom4j-1.6.1.jar

hibernate3.jar

javassist-3.9.0.GA.jar

jta-1.1.jar

log4j-1.2.15.jar

slf4j-api-1.5.8.jar

slf4j-log4j12-1.5.0.jar

2.创建实体类的映射文件.

Customer.hbm.xml

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="cn.itcast.spring.domain.Customer" table="customers" lazy="false">

<id name="id" column="id" type="integer">

<generator class="identity" />

</id>

<property name="name" column="name" type="string" />

<property name="age" column="age" type="integer" />

</class>

</hibernate-mapping>

spring整合Hibernate配置，相当于Hibernate.cfg.xml:



3.

spring:

connection.commit()|rollback

transaction.commit()|rollback.

事务:

a:atomic,原子性.

c:isolation,

i

d

事务管理:

1.编程式,硬编码方式.

2.声明式,

事务属性:

1.传播行为:事务的传递

2.隔离级别:控制并发程度的.

脏读:读未提交.

不可重复读:读不回去.

幻读:读多了.

ansi sql:

1:读未提交.

2:读已提交.

4:可以重复读.

8:串行化.

3.只读:优化.

4.超时:释放资源.

5.回滚规则:

高内聚,低耦合.

TransactionInterceptor{

invoke(MethodInvocation mi){

tm.getTransaction();

mi.proceed();

}

}

xxxHolder:SessionHolder{

}

SessionHolder{

Session session;

Transaction tx ;

}

SessionHolder绑定到当前线程.

spring和struts整合:

1.创建web程序

2.引入struts2类库.

3.创建HelloWorldAction

package cn.itcast.struts2.action;

import com.opensymphony.xwork2.ActionSupport;

/\*\*

\* HelloWorldAction

\*/

public class HelloWorldAction extends ActionSupport {

private static final long serialVersionUID = 6480501738385774728L;

public String reg() {

System.out.println("hello world");

return SUCCESS;

}

}

3.配置web.xml

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

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<filter>

<filter-name>action</filter-name>

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

</filter>

<filter-mapping>

<filter-name>action</filter-name>

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

</filter-mapping>

<welcome-file-list>

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

</welcome-file-list>

</web-app>

4.配置struts.xml

<?xml version="1.0"?>

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"

"http://struts.apache.org/dtds/struts-2.1.7.dtd">

<struts>

<constant name="struts.devMode" value="true" />

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

<action name="HelloWorldAction\_\*" class="cn.itcast.struts2.action.HelloWorldAction" method="{1}">

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

</action>

</package>

</struts>

5.引入spring类库 + struts2-spring-plugin-2.1.8.1.jar

6.配置web.xml

<!-- 通过上下文参数指定spring配置文件的位置 -->

<context-param>

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

<param-value>classpath:beans.xml</param-value>

</context-param>

<!-- 上下文载入器监听器,确保web服务器启动时,直接完成spring容器的初始化,将ac放到application范围中 -->

<listener>

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

</listener>

7.修改struts.xml配置文件中的action

把action的class改成bean的id.

<!-- class:指定action在spring容器的id -->

<action name="HelloWorldAction\_\*" class="helloWorldAction" method="{1}">

struts2加载顺寻:

1.struts-core.jar/struts.xml

2.xxx-plugin.xml / struts-plugin.xml

3.project/struts.xml

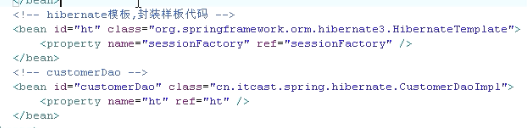
ac = new Classpathxmlac(new String[]{a.xml,b.xml});

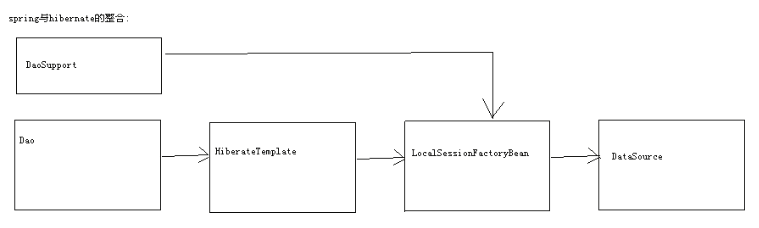
##### LocalSessionFactoryBean核心

继承了FactoryBean里面有getObject方法。

LocalSessionFactoryBean使用getObject()返回一个第三方的对象，比如使用hibernate就返回SessionFactory接口的实现类。(工厂模式)

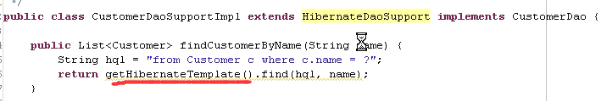
##### 使用HibernateTemplate模版类。

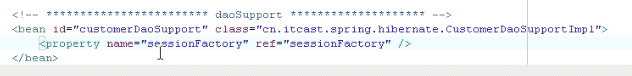




使用这个类作为CustomerDao中属性。需要以上配置。

可以使用Class CustomerDaoImpl extends HibernateDaoSuppoer implements CustomerDao｛｝来定义CustomerDao简化配置文件。





原理如下：HibernateDaosupport内部已经封装了一个HibernateTemplate对象。

public abstract class HibernateDaoSupport extends DaoSupport {

private HibernateTemplate hibernateTemplate;

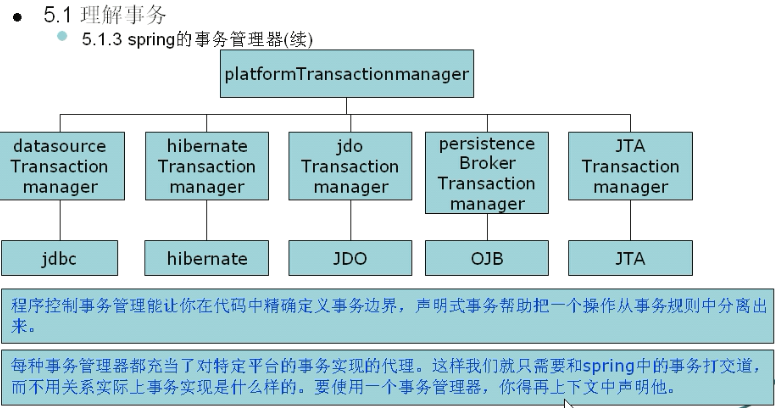
}

##### 批量保存

批量保存放在service层，因为是个长连接事务。dao层提供原子操作

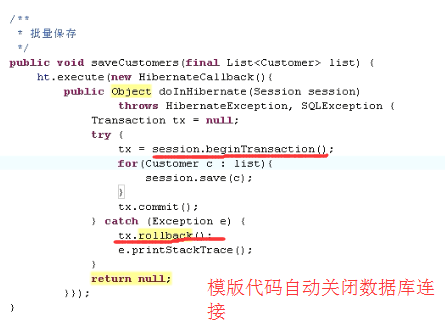
##### spring事务管理

spring不提供直接管理事务api，事务api还是由持久称提供。spring提供了平台事务管理器PlatformTransactionManager接口

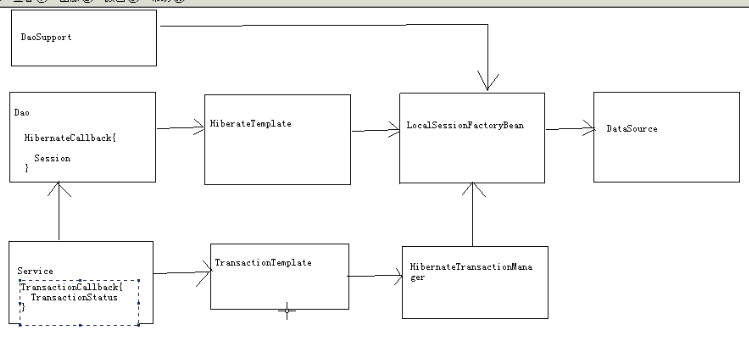


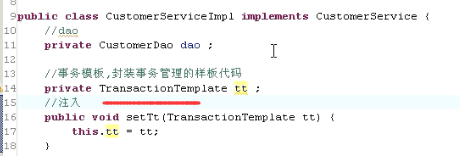
##### 使用TransactionCallback在service中实现事务管理：

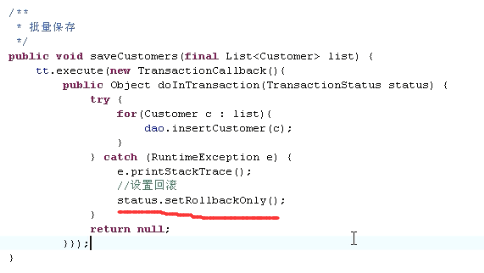
在CustomerDaoImpl中实现批量保存：在dao层实现中需要手动开启事务。（这个是最底层的实现。）



在CustomerServiceImpl中批量保存对象。







同时在spring文件中配置（配置关系就是根据上面的截图接头方向指定依赖关系）：



##### 声明式事务

以上的事务都是编程式事务（能精确定义事务边界）。

spring还提供了基于AOP的声明式事务。



最后配置一个\*是为了匹配所有的，因为有些会漏配。事务隔离级别优先从上到下匹配。

使用abstract=“true”来继承，简化配置：



##### 高内聚低耦合

在service层有插入单个的方法，同时需要一个批量插入。

那么在批量插入的方法中，是使用dao层的insert（）单个插入方法，还是service本身的insert方法呢？

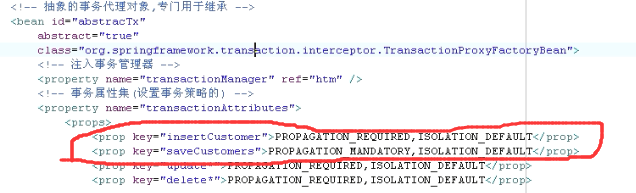
根据高内聚低耦合原则：使用this.insert方法好！（在对象的内部高读聚集，但是在对象像之间尽量减少耦合）

##### 传播属性

在service层有如下的插入insertCustomer，以及批量插入saveCustomer方法：



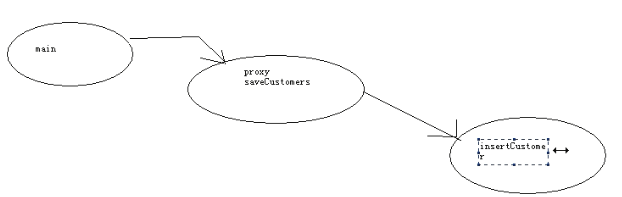
###### 定义批量保存是必须是在事务里面：



则报错：



解释：



相当于main方法中调用批量保存方法，但是main方法本身没有事务。所有报错。

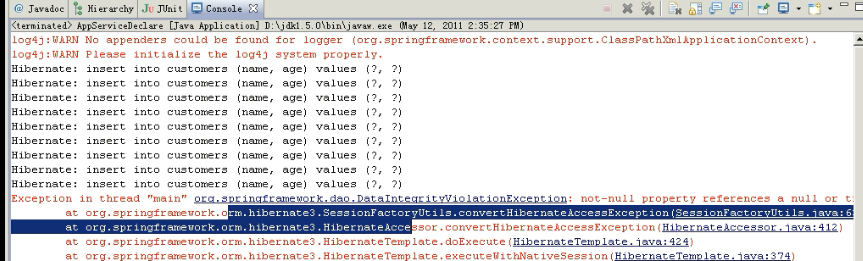
###### 定义在单个插入中用REQUEST\_NEW



含义：当前对象批量插入没有事务，则开启事务。

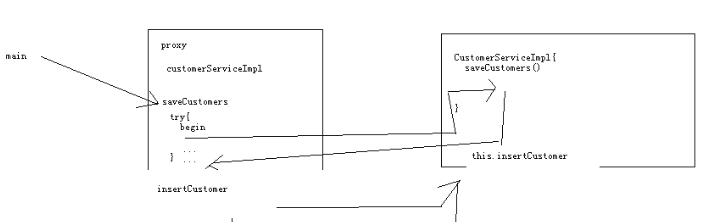
当前批量插入中有事务则暂挂，insertCustomer事务开启当内层的事务执行完，再执行外层。相当于先执行insertCustomer，执行完，再执行saveCustomer的事务。

结果：并且数据库中也没有任何记录被插入



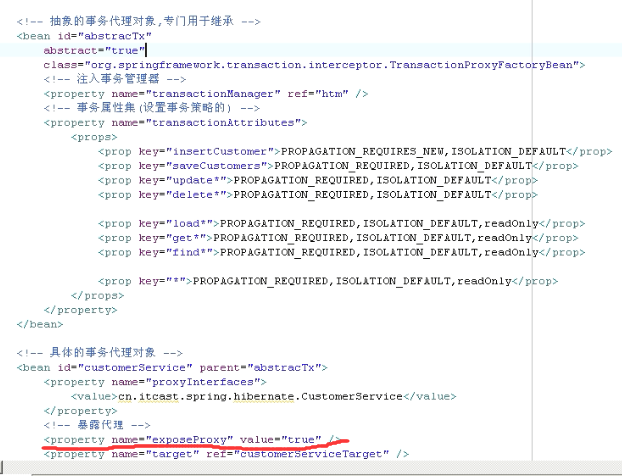
解释：因为上面的saveCustomer方法中调用的是this.insertCustomer

而在aop中CustomerService被代理，CustomerServiceProxy代理对象使用了CustomerService本身去insertCustomer，而CustomerService会在saveCustomer开始的时候就开启一个事务。因此后面出错，前面都不会存入。



那么如果才能产生插入前面9条记录，让最后一条错误的记录单独没被插入：

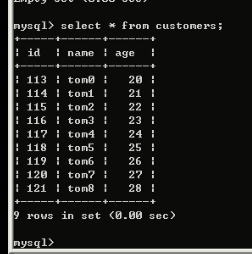
那么就需要在CustomerService中不要使用this对象，而是使用代理对象。首先修改spring配置文件：增加暴露代理对象：



同时修改批量插入方法：



执行：插入了9条，最后一条失败



##### 事务原理

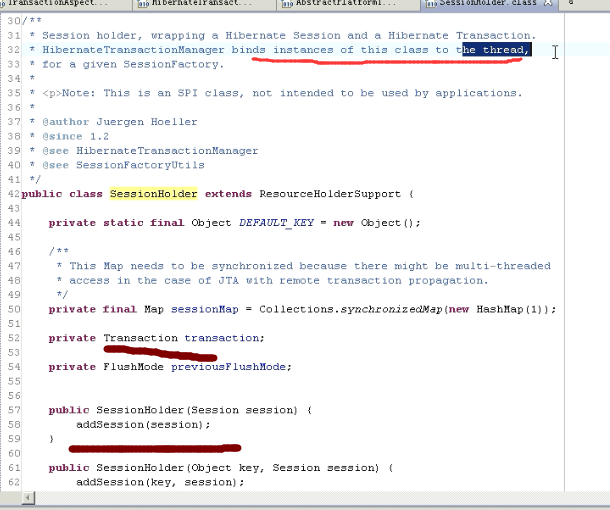
有SessionHolder（）｛ //session包容器 这个里面包装了一个Session和Transactation

Session s；

Transactation tx；

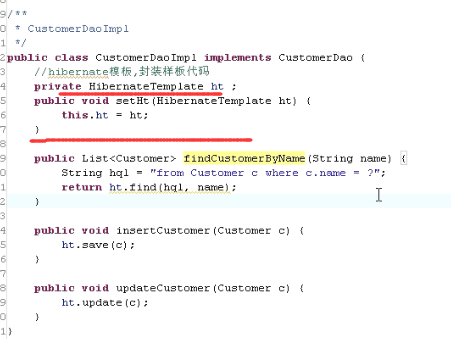
｝

事务管理器HibernateTransactionManager会绑定这个实例到当前线程上。



##### spring2.5之后引入简单的aop配置事务

在之前的Dao层实现中使用了HibernateTemplate模版，那么在dao中实现了事务。那么通过这个简单配置之后，事务放在了service层，又配置了事务的传播属性，因此有人提出在dao层中不再使用HibernateTemplate属性。



修改后如下：



同时配置文件中去除框选部分：同时修改customer 的

<property name = “sf” ref=”sessionFactory” />



整个修改后的配置文件如下：

<?xml version=*"1.0"*?>

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

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

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*

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

*http://www.springframework.org/schema/context/spring-context-2.5.xsd*

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

*http://www.springframework.org/schema/tx/spring-tx-2.5.xsd*

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

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd "*>

<!-- 指定分散配置的文件的位置 -->

<context:property-placeholder location=*"classpath:cn/itcast/spring/hibernate/tx25/jdbc.properties"* />

<!-- 配置c3p0数据源 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"${jdbc.driverclass}"* />

<property name=*"jdbcUrl"* value=*"${jdbc.url}"* />

<property name=*"user"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

<property name=*"maxPoolSize"* value=*"${c3p0.pool.size.max}"* />

<property name=*"minPoolSize"* value=*"${c3p0.pool.size.min}"* />

<property name=*"initialPoolSize"* value=*"${c3p0.pool.size.ini}"* />

<property name=*"acquireIncrement"* value=*"${c3p0.pool.size.increment}"* />

</bean>

<!-- 本地回话工厂bean,spring整合hibernate的核心入口 -->

<bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<!-- 指定hibernate自身的属性 -->

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.dialect"*>${hibernate.dialect}</prop>

<prop key=*"hibernate.show\_sql"*>${hibernate.show\_sql}</prop>

<prop key=*"hibernate.hbm2ddl.auto"*>${hibernate.hbm2ddl.auto}</prop>

</props>

</property>

<property name=*"mappingDirectoryLocations"*>

<list>

<value>classpath:cn/itcast/spring/domain</value>

</list>

</property>

</bean>

<!-- customerDao -->

<bean id=*"customerDao"* class=*"cn.itcast.spring.hibernate.tx25.CustomerDaoImpl"*>

<property name=*"sf"* ref=*"sessionFactory"* />

</bean>

<!-- hibernate事务管理器,在service层面上实现事务管理,而且达到平台无关性 -->

<bean id=*"htm"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"* />

</bean>

<!-- customerService -->

<bean id=*"customerService"* class=*"cn.itcast.spring.hibernate.tx25.CustomerServiceImpl"*>

<property name=*"dao"* ref=*"customerDao"* />

</bean>

<!-- 事务通知 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"htm"*>

<!-- 事务属性 -->

<tx:attributes>

<tx:method name=*"insert\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"*/>

<tx:method name=*"save\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"*/>

<tx:method name=*"update\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"*/>

<tx:method name=*"delete\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"*/>

<tx:method name=*"batch\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"*/>

<tx:method name=*"load\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"* read-only=*"true"*/>

<tx:method name=*"get\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"* read-only=*"true"*/>

<tx:method name=*"find\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"* read-only=*"true"*/>

<tx:method name=*"\*"* propagation=*"SUPPORTS"* isolation=*"DEFAULT"* read-only=*"false"*/>

</tx:attributes>

</tx:advice>

<!-- aop配置(配置切入点) -->

<aop:config>

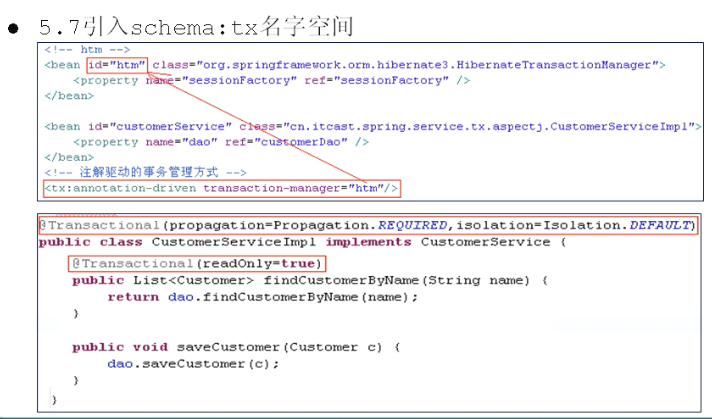
<aop:pointcut id=*"txPointcut"* expression=*"execution(\* \*..\*Service.\*(..))"*/>

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"txPointcut"* />

</aop:config>

</beans>

##### 使用注解驱动事务



##### spring和struts整合

最后2个文件。

#### 补充知识

##### 代理

参考[动态 代理（拦截）](#_动态_代理（拦截）)。

按照代理的创建时期，代理类可以分为两种。

1，静态代理：由程序员创建或特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。 (使用设计模式。)

2，动态代理：在程序运行时，运用反射机制动态创建而成。

###### 先看静态代理：

**package** com.ljh.dao;

**public** **interface** BookFacade {

**public** String addBook(String s);

}

**package** com.ljh.dao.impl;

**import** com.ljh.dao.BookFacade;

/\*\*

\* 装饰模式， 装饰对象实现被装饰对象相同的接口

\* **@author** lin

\*/

**public** **class** StaticBookFacadeProxy **implements** BookFacade {

**private** BookFacade bf;// 在装饰对象中引用被装饰对象。

**public** StaticBookFacadeProxy() {

}

**public** StaticBookFacadeProxy(BookFacade bf) {

**this**.bf = bf;

}

/\*

\* 继承了统一接口，因此和被装饰对象有相同的方法

\*/

@Override

**public** String addBook(String s) {

// **TODO** Auto-generated method stub

System.*out*.println("开始执行！");

bf.addBook(s);

System.*out*.println("结束执行！");

**return** **null**;

}

}

**public** **class** TestProxy {

@Test

**public** **void** test(){

StaticBookFacadeProxy sbfp = **new** StaticBookFacadeProxy(**new** BookFacadeImpl());

sbfp.addBook("我是装饰对象");

}

以上采用设计模式实现的代理。

缺点：装饰对象需要实现被装饰对象接口的所有方法。

###### JDK动态代理

中包含一个类和一个接口：   
InvocationHandler接口：   
public interface InvocationHandler {   
public Object invoke(Object proxy,Method method,Object[] args) throws Throwable;   
}   
参数说明：   
Object proxy：指被代理的对象。   
Method method：要调用的方法   
Object[] args：方法调用时所需要的参数

Proxy类：   
Proxy类是专门完成代理的操作类，可以通过此类为一个或多个接口动态地生成实现类，此类提供了如下的操作方法：   
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,   
InvocationHandler h)   
                               throws IllegalArgumentException   
参数说明：   
ClassLoader loader：类加载器   
Class<?>[] interfaces：得到全部的接口   
InvocationHandler h：得到InvocationHandler接口的子类实例

Ps:类加载器   
在Proxy类中的newProxyInstance（）方法中需要一个ClassLoader类的实例，ClassLoader实际上对应的是类加载器，在Java中主要有一下三种类加载器;   
Booststrap ClassLoader：此加载器采用C++编写，一般开发中是看不到的；   
Extendsion ClassLoader：用来进行扩展类的加载，一般对应的是jre\lib\ext目录中的类;   
AppClassLoader：(默认)加载classpath指定的类，是最常使用的是一种加载器。

动态代理类的字节码在程序运行时由Java反射机制动态生成

代理对象接口

**package** com.ljh.dao;

**public** **interface** BookFacade {

**public** String addBook(String s);

}

代理对象

**package** com.ljh.dao.impl;

**public** **class** BookFacadeImpl **implements** com.ljh.dao.BookFacade {

@Override

**public** String addBook(String s) {

System.*out*.println("增加图书方法");

**return** s;

}

}

动态代理的两种实现，本质都一样。

第一种：使用InvocationHandler匿名内部类

**public** **class** BookProxy {

**private** BookFacade bf;

**public** BookProxy(BookFacade bf){

**this**.bf = bf;

}

**public** BookFacade createProxy(){

**return** (BookFacade) Proxy.newProxyInstance(**this**.getClass().getClassLoader(),BookFacadeImpl.**class**.getInterfaces(),**new** InvocationHandler() {

//匿名内部类

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

**if**(method.getName().equals("addBook")){

System.*out*.println("开始增加图书事务！");

}

String result = (String) method.invoke(bf, args);

System.*out*.println("事务结束!");

**return** result;

}

});

}

}

第二种，将代理类实现InvocationHandler

**public** **class** BookFacadeProxy **implements** InvocationHandler {

**private** Object target; //////帮顶被代理对象

**public** Object bind(Object target){

**this**.target = target;

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(), **this**);

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

Object result = **null**;

System.*out*.println("事务开始");

result = method.invoke(target, args);

System.*out*.println("事务结束");

**return** **null**;

}

}

**public** **class** TestProxy {

**public** **void** test1() {

BookFacadeProxy proxy = **new** BookFacadeProxy();

BookFacade bookProxy = (BookFacade) proxy.bind(**new** BookFacadeImpl());

bookProxy.addBook("db");

}

@org.junit.Test

**public** **void** test2() {

BookProxy bookProxy = **new** BookProxy(**new** BookFacadeImpl());

BookFacade bf = bookProxy.createProxy();

bf.addBook("sb");

}

以上动态代理的实现都是需要依赖被代理对象的接口。

Proxy.newProxyInstance(**this**.getClass().getClassLoader(),BookFacadeImpl.**class**.getInterfaces()，InvocationHandler对象)

如果被代理接口没有实现接口，那么无法被代理

###### Cglib动态代理

JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理，cglib是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。

**public** **class** BookOperator { /////未实现接口类的代理。

**public** **void** add(){

System.*out*.println("普通的增加图书方法");

}

}

**package** com.ljh.proxy;

**import** java.lang.reflect.Method;

**import** net.sf.cglib.proxy.Enhancer;

**import** net.sf.cglib.proxy.MethodInterceptor;

**import** net.sf.cglib.proxy.MethodProxy;

/\*\*

\* 使用cglib代理 这里需要引入cglib的jar包。

我这里使用的是：cglib-nodep-2.1\_3.jar

\* **@author** lin

\*/

**public** **class** BookCgProxy **implements** MethodInterceptor {

**private** Object target;

**public** Object getInsatnce(Object target) {

**this**.target = target;

Enhancer enhancer = **new** Enhancer();

enhancer.setSuperclass(**this**.target.getClass());

// 回调方法

enhancer.setCallback(**this**);

// 创建代理对象

**return** enhancer.create();

}

@Override

**public** Object intercept(Object obj, Method method, Object[] args,

MethodProxy proxy) **throws** Throwable {

// **TODO** Auto-generated method stub

System.*out*.println("事物开始");

proxy.invokeSuper(obj, args);

System.*out*.println("事物结束");

**return** **null**;

}

}

**public** **void** test2() {

BookCgProxy cglib=**new** BookCgProxy();

BookOperator bookCglib=(BookOperator)cglib.getInsatnce(**new** BookOperator());

bookCglib.add();

}

##### 装饰设计模式和代理设计模式

装饰设计模式和代理设计模式非常相似，以至于自己很容易搞混，现在总结下：

总结：代理模式中，代理类对被代理的对象有控制权，决定其执行或者不执行。而装饰模式中，装饰类对代理对象没有控制权，只能为其增加一层装饰，以加强被装饰对象的功能。

一个在于“增加”职责，另一个在于“控制”访问。这是它们最本质的区别。

由此可以看到：学习设计模式重点在于“语义”上把握，而不是追求它的“形式。

如上面的静态代理的实现，根据上面这个原则，称为是代理模式，（也可以说是装饰模式，因为2个实现的类图都一样的。）

装饰模式侧重对对象增加方法，比如上面的StaticBookFacadeProxy可以改成抽象的，那么就能为再为BookFacade增加方法，然后后面的对象再去继承StaticBookFacadeProxy。

##### Tomcat

2015-08-20 15:53:34 基于：apache-tomcat-7.0.63

###### tomcat在windows配置

CATALINA\_HOME D:\apache-tomcat-6.0.41

再paht中添加% CATALINA\_HOME %\bin;

###### tomcat Manager内置角色：

manager-gui

允许访问html接口(即URL路径为/manager/html/\*)

manager-script

允许访问纯文本接口(即URL路径为/manager/text/\*)

manager-jmx

允许访问JMX代理接口(即URL路径为/manager/jmxproxy/\*)

manager-status

允许访问Tomcat只读状态页面(即URL路径为/manager/status/\*)

###### web应用部署方式

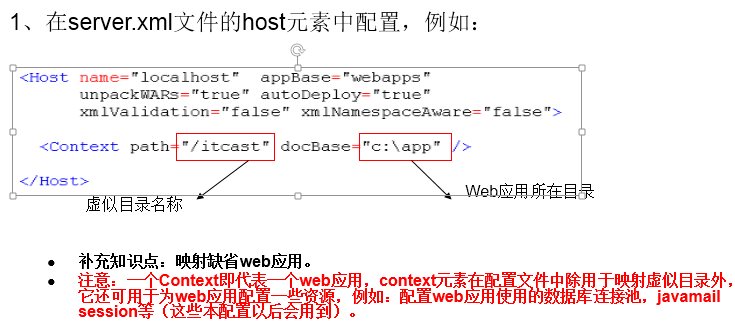
1. tomcat自动部署：直接扔到webapp
2. 利用控制台部署：通过html界面
3. 增加自定义web部署文件：在conf增加Catalina文件夹，在Catalina下新建localhost目录。在localhost目录增加ljh.xml文件

文件内容：

<context docBase=”目录” debug=“0” privileged=“true”></context>

xml文件名为访问地址。

1. 修改servlet.xml文件部署web应用：修改server.xml文件，但是这种方式必须重启服务器，不推荐，而且有可能破坏tomcat服务。



###### 配置tomcat数据源

tomcat内置了dbcp数据源实现。同时提供2中配置数据源方式：一种全局数据源（所有 web应用都能访问），一种局部数据源（单个web访问）。（配置数据源需要早tomcat/lib下放相应的数据库驱动）

如：在上面的ljh.xml文件中添加

<context docBase=”目录” debug=“0” privileged=“true”>

<resource name=”jdbc/dstest” auth=”Container”

type=“javax.sql.DataSource”

dirverClassName=”com.mysql.jdbc.Driver”

url=”jdbc:mysql//localhost:3306/javaee”

username=”root” password=”root”

maxActive=”5” maxIdle=”2” maxWait=”1000” />

</context>

使用JNDI（java naming directory interface）java命名和目录接口技术（为某个java对象取个名字）参考[c3p0:](#_c3p0:) 以及[JNID技术](#_JNID技术)

**import** javax.naming.Context;

**import** javax.naming.InitialContext;

**import** javax.sql.DataSource;

**public** **class** Test {

**public** **void** test() **throws** Exception{

//初始化

Context ctx = **new** InitialContext();

//通过jndi查找 数据源，java:comp/env是Tomcat提供的固定前缀

DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/dstest");

}

}

配置tomcat中的连接池：context可以在server.xml或者catelina文件夹下配置，还可以在META-INF中配置。

|  |  |  |
| --- | --- | --- |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | <Context ...>  ...  <Resource name="jdbc/EmployeeDB"  auth="Container"  type="javax.sql.DataSource"  username="dbusername"  password="dbpassword"  driverClassName="org.hsql.jdbcDriver"  url="jdbc:HypersonicSQL:database"  maxActive="8"  maxIdle="4"/>  ...  </Context> |  |

在tomcat 帮助文档中的JNDI Resources中查看。同时使用tomcat创建连接池，需要将对应的数据库驱动放到tomcat的lib目录下

在dao层获取数据库连接池：

|  |  |  |
| --- | --- | --- |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif |
| http://tomcat.apache.org/tomcat-6.0-doc/images/void.gif | Context initCtx = **new** InitialContext();  Context envCtx = (Context) initCtx.lookup("java:comp/env");  *dataSource* = (DataSource)envCtx.lookup("jdbc/datasource"); |  |