# HTML U28E590D

## html的结构.

<html>

<head>

<title> </title>

</head>

<body>

</body>

</html>

## html的常用标签.

### 文本标签.

标题

h1~h6 .

<h1>标题名称</h1>

段落

escape="false"以html输出

<p>一段话</p>

长引用

<blockquote></blockquote>

短引用

<q></q>

文本方向

<bdo dir=”ltr”></bdo>

属性：ltr rtl

上下标

上标

<sup><sup>

下标

<sub><sub>

换行

<br/>

注意: 写法特殊. 开始和结束是一体的.

原样输出

<pre>文本</pre>

线条

<hr/>

列表

dl dt dd

<dl>

<dt></dt>

<dd></dd>

<dt></dt>

<dd></dd>

</dl>

有序列表

<ol>

<li>列表项</li>

<ol>

所谓的有序,就是每一个列表项开头多了编号(数字, 拉丁,英文)

无序

<ul>

<li>列表项</li>

</ul>

所谓的无序,并不是说,列表项没有顺序了,只是没有显示出编号而已,代之的是符号(圆,矩形)

### 表格

结构

<table>

<caption>

表格的标题

</caption>

<tr>

<th>表头1</th>

<th>表头2</th>

</tr>

<tr>

<td>单元格1</td>

<td>单元格2</td>

</tr>

</table>

td 就是普通的一个单元格. 其中的文本,不会居中,布局加粗.

th 通常用于表的表头, 其中的文本,居中,并加粗显示...

使用到的属性.

table的属性:

border 边框.

width 表格的宽度, 可以是像素,也可以百分比.

align 对其方式, left rigth center .一个表格默认是left .

cellspacing 单元格和单元格之间空白的距离.

cellpadding 单元格内的文本和单元格之间空白的距离.

colspan合并列

rowspan合并行

### html 表单.

标签.

form: 用于定义表单.

属性: onsubmit=”turn true” method="post" action="192.168.3.26:80" get数据量不能超过1kb

文本输入框

标签input .

属性 type="text" size=”4”

密码输入框

标签 input

属性 type="password"

单选框

标签 input

属性 type="radio" checked=”checked”

多选框

标签 input

属性 type="checkbox"

下拉列表

标签 select 和option

<select>

<option>选项一</option>

<option>选项二</option>

<option>选项三</option>

</select>

文本域

标签 textarea

height width rows cols

注意:

在form 表单中如果需要提交表单数据,或者重置表单数据. 不能使用普通的按钮. 使用还是input .

提交按钮

标签 input type="submit"

重置按钮

标签 input type="reset"

### frameset框架

<frameset row=”30%,150px,…..” col=”” noresize="noresize" >

<frame src=”….html”/>

</frameset>

用frameset就不能有body标签

### <a>标签.

1.超链接

<a href="http://www.baidu.com" >百度一下</a>

<a href="success.html" >百度一下</a> /

属性: href: 明确链接的目标

target="\_blank" 明确了在何处打开目标.

title 超链接的标题. 光标提示信息

2.锚链接

<a href="#锚点"></a>

<a name="锚点"></a>

<a href**=**"javascript:void(0);" >

### html图片

**img**

<img src="url" width=”50%” height=”50%” border=”3px” alt=”图片获取不了显示内容” align ="left" />

**map**

map客户端图像映射，指带有可点击区域的一幅图像

<img src="planets.jpg" border="0" usemap="#planetmap" alt="Planets" />

<map name="planetmap" id="planetmap">

<area shape="circle" coords="180,139,14" href ="venus.html" alt="Venus" />

<area shape="circle" coords="129,161,10" href ="mercur.html" alt="Mercury" />

<area shape="rect" coords="0,0,110,260" href ="sun.html" alt="Sun" />

</map>

# css

1.使用css .

方式一:使用style 属性

方式二:使用style 标签

<style>

css 内容...

</style>

方式三:使用<link>标签(引入外部样式)

在html的head 标签中.

<link rel="stylesheet" type="text/css" href="样式表的路径"/>

2.重要的标签div 和span

3.css选择器.

标签选择器——标签名{}

类选择器—— . 类名{}

id 选择器——#id名{}

4.选择器优先级 标签<类<id<style

5.css的属性.

1)字体属性:

color:

字体的颜色:

可以写：　颜色的名称 color:green ;

十六进制: color:#ffffff;

color: rgb(100,14,200);

font-family :

字体的名称.

例如:

font-family : '黑体';

font-style:

只有三个值. 正常(normal ),斜体(italic ),倾斜的字体(oblique).

text-decoration:

文本的修饰. 下划线,上划线,贯穿线.

font-weigh

设置字体的粗细,而不是变大.

font-size

设置字体的大小.

2)文本:

3)背景:

background-color

背景颜色

例如:

background-color :green;

background-color :#fffff;

background-color :rgb(红,绿,蓝的值);

background-image

背景图片

background-image : url ('图片的路径'|本地|网络中 )

background-image: url("comet.jpg");

4)边框

border

符合属性: 可以通过border 属性同时设置边框的粗细,样式,和颜色.

border:1px solid green;

该案例中,可以设置边框(边框的粗细,边框的样式,边框的颜色)

分开分别设置边框的属性:

如果只需要设置边框的宽度:

border-width:像素|

如果只需要设置边框的样式:

border-style:样式.

该属性可以决定边框的是实线,虚线,双线显示.

如果只需要设置变量的颜色:

border-color:颜色;

经典例子：

图片与文本框对齐问题，在文本框和图片上都添加以下的样式

style="vertical-align: middle;"

# JavaScript

1.脚本语言.

不能独立运行,依赖于宿主环境(html).

2.解释型

不存在编译的概念, 通过浏览器打开html,html 出现了js 代码. 有浏览器(内置解释器),解释执行...

3.javascript和java的不同:

1.公司不同.

sun . 网景

2.js 解释型,不需要编译.

java 需要编译为字节码. 才能运行.

3.js是弱类型 ,java是强类型(增强)

4. javascript的组成

ECMAscript基础语法

变量 数据类型 运算符 流程控制语句 函数 数组 BOM DOM

3.javascript基础语法:

1. 如何在html 中添加js 代码?

方式一:通过script标签.

<script type="text/javascript">

js代码;

</script>

方式二:通过script标签. 引入外部的.js 文件.

<script type="text/javascript" src="外部js文件的路径"></script>

注意:

通常会将 <script>标签定义在html body 的末尾...

常用的方法.

window.alert(); ->弹窗.

window.document.write(); ->将js 的内容写到html

2.变量.

3.运算符

算术 比较 赋值 逻辑 位运算.

注意:

js 中逻辑运算符:

&& || !

js 中位运算符

& | ^ ~

4.流程控制语句

5.函数

提高代码复用性. 封装独立功能程序...

通过函数名调用...

6.数组

正规表达式例子：/^\d{6,10}$/ 6到10位数字

/^[a-z\_A-Z]\w{5,15}$/ 开头必须为字母，6到16位

/^\w+@\w+(\.\w{2,3}){1,2}$/ 邮箱

/^1[34578]\d{9}$/ 手机号

判空name==""||name==undefined

传参，返回值false为不执行语句onclick="islogin(id);return false;"

# CSS经典实列

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

按钮C:\Users\ozr\feiq\RichOle\317656338.bmp

背景图片F:\Aptana Studio 3 Workspace\frameset\bbg.png

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

a**:**link **{**

color**:** #5AA0FA**;**

font-size**:** 16px**;**

text-decoration**:** none**;**

**}**

a**:**hover **{**

font-size**:** 20px**;**

color**:** #6C3454**;**

**}**

td**{**

background-image**:** url(*"bbg.png"*)**;**

width**:** 152px**;**

**}**

</style>

<a href**=**"#">添加联系人</a>

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

# JavaScript经典实列

==============================验证码==============================

--------------------------------JavaScript部分

changeCode ();

**function** changeCode **()** {

**var** datas=["A","B","C","D","E","F","G","T","Y","M","N","U","Z","L","0","1","2","3","4","5","6","7","8","9","0"];

**var** code="";

**var** index=0;

**for**(**var** i=0;i<4;i++){

**var** index=Math.floor(Math.random()\*datas.length);

code+=datas[index];

};

**var** codeSpan=window.document.getElementById("codeSpan");

codeSpan.innerHTML=code;

codeSpan.value=code;

}

-------------html部分

<input style**=**"text" name**=**"checkNum" id**=**"checkNum" size**=**"4" onclick**=**"checkBirthday();checkAddress();" onblur**=**"checkCode()"/>

<span id**=**"codeSpan" value**=**"B3Y7" style**=**"font-family**:** 黑体**;**font-weight**:** bold**;**font-style**:** italic**;**background-color**:**#BCB2B2**;**text-decoration**:**line-through**;**">B3Y7</span>

<a href**=**"#" onclick**=**"changeCode()">看不清</a>

==================================全选按钮===============================

----------------------JavaScript代码

**var** checkbox=window.document.getElementsByName("checkbox");

**function** checkboxAll**()** {

//通过name属性一次拿出所有的checkbox

**if**(checkbox[0].checked){

**for**(**var** i=1;i<checkbox.length;i++){

checkbox[i].checked = true;

}

}**else**{

**for**(**var** i=1;i<checkbox.length;i++){

checkbox[i].checked = false;

}

}

}

--------------------html代码

<input type**=**"checkbox" name**=**"checkbox" onclick**=**"checkboxAll()"/>

<input type**=**"checkbox" name**=**"checkbox" />

<input type**=**"checkbox" name**=**"checkbox" />

<input type**=**"checkbox" name**=**"checkbox" />

<input type**=**"checkbox" name**=**"checkbox" />

======================页面加载初始化表单日期=============================

-------------------------------------JavaScript部分

//js加载进来就执行getBirthday()初始化年月

getBirthday();

**function** getBirthday **()** {

getYear();

getMonth();

}

**function** getYear**()**{

**var** year=window.document.getElementById("year");

**var** minYear=1990

//获取本年

**var** maxYear=new Date().getFullYear();

**for**(**var** i=minYear;i<=maxYear;i++){

**var** option=window.document.createElement("option");

option.innerHTML=i;

option.value=i;

year.onclick=getDay;

year.appendChild(option);

}

}

**function** getMonth**()**{

**var** month=window.document.getElementById("month");

**for**(**var** i=1;i<=12;i++){

**var** option=window.document.createElement("option");

option.innerHTML=i;

option.value=i;

month.onclick=getDay;

month.appendChild(option);

}

}

**function** getDay**()**{

**var** day=window.document.getElementById("day");

**var** month=window.document.getElementById("month");

**var** year=window.document.getElementById("year");

//为了保证数据安全，防止出现4月31这种现象出现，每次点击年或者月，都把日设置回默认值，

**var** optionArr=window.document.getElementById("day").options;

optionArr[0].selected=true;

**if**(month.value=="1"||month.value=="3"||month.value=="5"||month.value=="7"||month.value=="8"||month.value=="10"||month.value=="12"){

**for**(**var** i=1;i<=31;i++){

**var** option=window.document.createElement("option");

option.innerHTML=i;

option.value=i;

day.appendChild(option);

}

}**else if**(month.value=="4"||month.value=="6"||month.value=="9"||month.value=="11"){

**for**(**var** i=1;i<=30;i++){

**var** option=window.document.createElement("option");

option.innerHTML=i;

option.value=i;

day.appendChild(option);

}

}**else if**(month.value=="2"&&(year.value%4==0&&year.value%100!=0||year.value%400==0)){

**for**(**var** i=1;i<=29;i++){

**var** option=window.document.createElement("option");

option.innerHTML=i;

option.value=i;

day.appendChild(option);

}

}**else**{

**for**(**var** i=1;i<=28;i++){

**var** option=window.document.createElement("option");

option.innerHTML=i;

option.value=i;

day.appendChild(option);

}

}

}

---------------------html部分

<select id**=**"year">

<option value**=**"">年</option>

</select>

<select id**=**"month">

<option value**=**"">月</option>

</select>

<select id**=**"day">

<option value**=**"">日</option>

</select>

-------------------日期check.js部分

**function** checkBirthday **()** {

**var** year=window.document.getElementById("year").value;

**var** month=window.document.getElementById("month").value;

**var** day=window.document.getElementById("day").value;

**var** bdSpan=window.document.getElementById("bdSpan");

**if**(year==""){

bdSpan.style.color="red";

bdSpan.innerHTML="年不能为空!!!";

**return** false;

}**else if**(month==""){

bdSpan.style.color="red";

bdSpan.innerHTML="月不能为空!!!";

**return** false;

}**else if**(day==""){

bdSpan.style.color="red";

bdSpan.innerHTML="日不能为空!!!";

**return** false;

}**else**{

bdSpan.style.color="green";

bdSpan.innerHTML="√";

**return** true;

}

}

======================页面时装============================

<script type**=**"text/javascript">

**function** \_genDate**()**{

**var** date = new Date();

**var** dateStr = date.getFullYear()+"年"+(date.getMonth()+1)+"月"+date.getDate()+"日&nbsp;"+date.getHours()+"时"+date.getMinutes()+"分"+date.getSeconds()+"秒";

document.getElementById("showTime").innerHTML = dateStr;

}

//定时器1000毫秒刷新一次

window.setInterval("\_genDate()", 1000);

</script>

页面代码：

当前时间为： <span id="showTime"></span>

==========================判断时间============================================

resTime是时间字符串yyyy-MM-dd HH:mm:ss

new Date(Date.parse(resTime.replace(/-/g, "/"))) < new Date()

# Junit单元测试工具（必须掌握）

1）概述：Junit是专业的测试工具。 Java for unit testing

2）使用juint

建立测试类，测试类中有单元测试方法

右键->建立“Junit Test Case”-> 写类名，选择需要测试的类 -> 选择需要测试的方法->finish

3)单元测试方法的编写方式

3.1 测试方法必须在方法的顶部加上@Test（注解）

3.2 方法必须是pullic，没有返回值，不能有参数，可以抛出异常

3.3 方法名称没有规定，但是建议驼峰式命名法（ test+方法名 例如： testAdd() ）4)如何查看测试结果是否成功

看junit面板

绿色：代表测试成功

红色：代表测试失败

5）如何运行测试方法

5.1 方法上运行： 双击方法-》右键run->junit test

5.2 类上运行： 双击类-》右键run->junit test

5.3 项目上运行： 双击项目 -> 右键run->junit test

6)判定结果

Assert工具类：

Assert.assertEquals(期望值，实际值)： 使用equals方法进行判断，相等时，测试成功

Assert.assertSame(期望值，实际值)： 使用==方法进行判断

Assert.assertTrue(); 结果为true时，测试成功

Assert.assertFalse(); 结果为false时，测试成功

# xml

## 二、XML概述和作用

1)XML概述

XML，eXtend Marup Language 可扩展标记语言

HTML与XML的相同点为：他们都是标记语言

HTML与 XML的不同点

作用： 负责网页的结构 \*1）描述带关系的数据结构(作为软件的配置文件)

2）作为小型“数据库”

语法 松散性 语法相对严谨

（大小写不区分） （大小写严格区分.....）

标记： 固定的标记 可以由开发者自行定义（扩展的）

2)xml作用

\*2.1 描述带关系的数据结构(作为软件的配置文件)

\*.properties 配置文件： key=value key=value key=value

xml配置文件：描述包含与被包含的关系

应用： tomcat服务器 三大框架（struts2，spring，hiberate）作为配置文件

2.2 作为小型“数据库”

结构清晰

## 三、XML语法（重点）

1）语法： <student>xxxx</student>

注意：

1）大小写严格区分

2) 开始标签和结束标签必须配对

3）标签名称不能以数字开头，中间不能有空格

4）没有定义名称空间，不能包含冒号

5）一个xml文件有且仅有一个根标签（重点）

属性

1）属性值必须使用引号包含，单引号或者双引号，不能单双混用

2）在同一个标签内，不能出现同名的属性

2）声明<?xml version="1.0" encoding="utf-8"?>

3）转义字符

< &lt; letter than

> &gt; greater than

& &amp;

CDATA块：批量转义

4）处理指令（了解）

语法： <?xml-stylesheet type="text/css" href="person.css"?>

作用： 告诉解析器当前xml文件使用什么css文件，一旦使用了样式文件，该xml文件可以展示网页的结构

## 四、XML约束之DTD约束（看懂即可）

1）搞明白两个概念

什么是良好的xml文件？ 符合xml语法的xml文件 （良好的xml不一定是有效的）

(xml语法是有w3c指定的)

什么是有效的xml文件？ 符合xml约束规则的xml文件（有效的xml文件一定是良好的）

（xml约束规则由开发者根据业务来指定的）

2）XML约束分类

DTD约束

语法简单，功能简单

struts hibernate

Schema约束

语法复杂，功能强大，为了替代DTD

spring

## DTD约束语法

3.1 如何使用dtd？

1）内部使用

dtd内容和xml内容放在一个文件中

2）外部使用

在xml文件中导入dtd文件

<!DOCTYPE note SYSTEM "note.dtd"> dtd文件放在本地的文件夹

<!DOCTYPE note PUBLIC "http://gz.itcast.cn/note.dtd"> dtd文件放在互联网上

3.2 具体语法（能够看懂内容即可，可以根据dtd约束内容来编写一个有效的xml文件）

约束标签

语法：<!ELEMENT 元素名称 类别> 或者<!ELEMENT 元素名称 (元素内容)>

类别：

EMPTY： 空标签 <to></to> <to/>

(#PCDATA)： 普通字符（不能包含子标签）

ANY ：可以是空标签，可以是普通字符串，还可以是包含子标签

(元素内容)

顺序问题：

(to,from,heading,body)：必须依次出现to,from,heading，body

数量问题：

to： 有且仅出现1个

to\*: 出现零或多个

to+： 出现1或多个

to?: 出现零或1个

约束属性

语法：<!ATTLIST 元素名称 属性名称 属性类型 默认值>

默认值： 约束属性是否应该存在

#REQUIRED 属性值是必需的

#IMPLIED 属性不是必需的

#FIXED value 属性值是固定的

属性类型：约束属性的内容

CDATA： 属性内容是普通字符串

（en1|en2.。）： 属性内容是枚举值，多选一

ID ： 属性内容必须唯一

## schema约束

1）概述

schema比dtd强大

XML Schema符合XML语法结构。

DOM、SAX等XML API很容易解析出XML Schema文档中的内容。

XML Schema对名称空间支持得非常好。

XML Schema比XML DTD支持更多的数据类型，并支持用户自定义新的数据类型。

XML Schema定义约束的能力非常强大，可以对XML实例文档作出细致的语义限制。

XML Schema不能像DTD一样定义实体，比DTD更复杂，但Xml Schema现在已是w3c

2）名称空间（重点）

名称空间是在xml文件中导入schema约束文件的关键。 （\*.xsd）

\*\*\*\*\*\*名称空间可以区分不同的schema文件\*\*\*\*\*\*\*

life:body

html:body

徐氏法则：

四步导入

第一步：书架标签受到了http:www.itcast.cn的名称空间约束

第二步: 给名称空间起一个别名：itcast

第三步: 指定名称空间所在的schema文件在哪里

第四步：定义xsi内置的名称空间（固定写法）

## 五、XML解析之JAXP解析（重点）

1）xml解析： 使用java程序读取或修改（操作）xml文件

2）两种解析xml的方案：（面试的时候经常问）

dom解析：原理一次读取xml文件。比较占用内存

sax解析：原理读取一点，解析一点。占用内存必须少

### dom解析原理

复习js的dom：

1）一句话说明js的dom：使用js代码通过dom对象操作html文件内容

2）有哪些dom对象：

\*\*document：表示一个html文件内容

\*\*element： 表示元素（标签）

\*attr：表示属性

\*text：表示文本

comment：表示注释

3）这些dom对象构建成一颗树状结构

4）js的dom方法：

查询：

getElementById("id")

getElementsByName("name")

getElementsByTagName("tagName");

添加：

document.createElement("name");

element.setAttribute("name","value")

element.getAttribute("name");

插入：

element.appendChild(element);

删除：

element.removeChild(element);

element.removeAttribute("name")

xml的dom解析：

一句话： 使用java代码通过dom对象去控制xml文件内容

4）xml的dom解析工具

\*SUN官方的jaxp

非官方的jdom:好用

\*非官方的dom4j：更好用

5）SUN官方的dom解析工具-jaxp (java xml parsing)

内置在jdk中 org.w3c.dom.\*

DOM中核心的对象：

\*\*Document：表示一个xml文件内容

\*\*Element： 表示元素（标签）

\*Attr：表示属性

\*Text：表示文本

Comment：表示注释

6）jaxp开发步骤：

6.1 读取xml文件

1) 创建DocumentBuilderFactory

DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();

2）通过DocumentBuilderFactory获取DocumentBuilder

DocumentBuilder db=factory.newDocumentBuilder();

3) 通过DocumentBuilder的parse方法读取xml文件

Document doc=db.parse("path.xml");

6.2 修改文件

1）创建TransformerFactory

TransformerFactory tFactory=TransformerFactory.newInstance();

2）通过 TransformerFactory得到Transformer

Transformer tf=tFactory.newTransformer();

附加）实现换行效果

tf.setOutputProperty(OutputKeys.INDENT,"yes");

3）通过TransformerFactory.transfer()修改文件

tf.transform(new DOMSource(doc), new StreamResult(new File("path.xml")));

### sax解析

1）概述

sax，simple api for xml parsing 因为dom解析是一次加载到内存，比较占用内存。所以出现sax弥补dom解析的效率问题。

sax解析的原理是读取一点，解析一点，占用内存比较少。

sax程序api： jdk： org.xml.sax.\*

2）sax执行原理

看

DOM与SAX的不同点（面试经常问题的）

1）原理：

DOM是一次性读入内存，然后构建成树状态结构 SAX是读取一点，解析一点

2）编码方式：

DOM是面向对象的方法 SAX是基于事件驱动的方式

3）对xml的操作方式：

DOM可以任意增删改成xml文件 SAX只能读取xml文件，不用于修改

4）读取的顺序：

DOM方式可以来回读取xml文 SAX只能从上往下依次读取，不能往回读

SAX使用

//1)创建SAXParseFactory

SAXParserFactory factory = SAXParserFactory.newInstance();

//2)得到SAXParser

SAXParser sp = factory.newSAXParser();

//3)得到XMLReader

XMLReader xr = sp.getXMLReader();

//4)选择需要使用接口的实现

xr.setContentHandler(new MyContentHandler());

//5)开始读取和解析

xr.parse("./src/exam.xml");

MyContentHandler类实现接口ContentHandler重新所有的方法，

或者MyContentHandler继承DefaultHandler适配器，重写有需要的方法就可以了

## dom4j解析（简化版）

1）概述

dom4j是简化版的dom解析工具。

2)dom4j开发步骤

导入对应的jar包

3）总结domj4的常用的方法

3.1 读取xml文件

SAXReader reader = new SAXReader();

Document document = reader.read("xml路径");

3.2 修改xml文件

XMLWriter writer = new XMLWriter(new FileOutputStream("xml路径"))；

writer.write(document);

writer.close()

3.3 常见方法

查询：

标签：

element.element("name") 查询第一个子标签

element.elements("name") 查询指定名称的所有子标签

element.elementIterator("name") 查询指定名称的所有子标签

element.elements() 查询所有子标签

element.elementIterator(); 查询所有子标签

属性：

Attribute attribute = element.attribute("name") 查询指定名称的属性对象

element.attribteValue("name") 查询指定名称的属性值

element.attributes() 查询当前标签的所有属性

文本：

element.getText() 查询当前标签的文本

element.elementText("name") 查询子标签的文本

添加：

标签：

element.addElement("name");

属性：

element.addAttribute("name","value");

文本：

element.addText("value");/ element.setText("value")

修改：

属性：

attribute.setValue("value");

element.setText("value")

删除：

标签：

element.detach();

element.remove(element);

## 六、xpath插件

1）概述

xpath的作用主要是用于快速地查询xml的内容（标签，属性，文本）

2）xpath的使用步骤

document.selectNodes("xpath表达式") 查询多个节点（标签，属性，文本）

document.selectSingleNode("xpath表达式") 查询一个节点（标签，属性，文本）

3）xpath表达式

查看文档学习

xpath表达式

/aaa 第一层目录下的aaa标签

/aaa/ccc 第一层目录下的aaa标签下的ccc 标签

/aaa/ddd/bbb 第一层目录下的aaa标签下的ddd标签下的bbb标签

//bbb 目录下所有bbb标签

//ddd/bbb 目录下所有ddd标签下的bbb标签

/aaa/ccc/ddd/\* aaa目录下的ccc目录下的ddd目录下的所有标签

/\*/\*/\*/bbb 所有第4层下的bbb

/aaa/bbb[1] aaa目录下的第一个bbb标签

/aaa/bbb[last()] aaa目录下的最后一个bbb标签

//@id 所有目录下的id属性

//bbb[@id] 所有目录下带有ID属性的bbb标签

//bbb[@id=’b1’] 所有目录下带有id属性且id值为b1的bbb标签

//bbb[normalize-space(@name)=’bbb’] 所有目录下所有目录下带有id属性且id值去掉前后空格后为b1的bbb标签

//bbb[text()=’b1’] 所有bbb目录找到文本为b1的标签

# tomcat服务器

**市场常见收费服务器**webLogic、websphere

**版本特点**

版本 支持servlet/jsp规范 默认编码

tomcat6.x servlet2.5 iso-8859-1

tomcat7.x servlet3.0 iso-8859-1

tomcat8.x servett3.0 utf-8

**端口**

8080 tomcat软件

3306 mysql

1521 oracle

80 默认端口。

**访问地址**<http://localhost:8080/工程名/hello.html>

**目录介绍：**

\*bin： 存放tomcat软件的指令 启动/关闭指令

\*conf: 存放tomcat软件的配置文件 server.xml是核心配置文件

lib: 存放支持tomcat软件运行的jar包和tomcat支持的技术jar包

logs: 日志文件

temp: 临时文件

\*webapps: 存放web应用

work： 存放jsp运行时的临时文件

**web应用的目录（文件）结构**

|- WebRoot -- web应用的根目录（自定义）

|-静态资源(html/css/js/images)

|-WEB-INF 目录，固定写法 (动态资源)

|-classes 目录，固定写法，存放class文件

\*.class

|-lib 目录，固定写法，存放jar包

\*.jar

web.xml 资源描述文件

# HTTP协议

## 概念

浏览器和服务器之间数据传输的格式规定。

## 请求

请求行：请求方式（GET/POST） 请求资源 http协议版本(1.1)

请求头：键值对 key:value

常见的请求头：

accept: 接收的数据类型 mime text/html....

\*accept-encoding: 接收的数据压缩格式 \*gzip compress

accept-lanuage: 接收的语言（优先级）

accept-charset: 接收的码表

if-modify-since: 浏览器缓存文件的最后修改时间

\*referer: 当前请求来自于哪里

\*content-type: 请求正文内容类型

application-www-form-urlencoder... key=value

multipart/form-data 文件上传时

user-agent: 浏览器类型

\*cookie: 会话

connection: 连接状态

date: 发出时间

空行

请求正文： POST方式提交的内容就是正文

## 响应

响应行： http协议版本(1.1) 状态码 状态描述

状态码：服务器处理请求的状态

\*200： 表示处理完成

302： 表示客户端需要进一步请求才能完成

304： 表示服务器资源没有修改，通知浏览器使用缓存

\*404： 表示客户端错误，资源没有找到

\*500： 表示服务器资源错误

状态描述:对状态码的补充

响应头： 键值对（暗号）

\*Location: http://www.it315.org/index.jsp 请求重定向的地址，通常和302一起使用

Server:apache tomcat 服务器名称

\*Content-Encoding: gzip （响应正文）通知浏览器使用什么压缩格式解压

Content-Length: 80 响应正文的长度（以字节为单位）

Content-Language: zh-cn 响应的支持的语言

\*Content-Type: text/html; charset=GB2312 响应正文的内容类型 mime类型（默认以text/html发送）

Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT 服务器资源的最后修改时间

\*Refresh: 1;url=http://www.it315.org 定时刷新

\*Content-Disposition: attachment; filename=aaa.zip 文件下载提示

\*Set-Cookie:SS=Q0=5Lb\_nQ; path=/search 会话

Expires: -1 建议浏览器不使用缓存

Cache-Control: no-cache 建议浏览器不使用缓存

Pragma: no-cache 建议浏览器不使用缓存

Connection: close/Keep-Alive 连接状态

Date: Tue, 11 Jul 2000 18:23:51 GMT 响应发出时间

空行

响应正文 用户在浏览器看到的内容就是响应正文

# servlet编程

## servlet概述

1）servlet就是一个接口。javax.servlet.Servlet

所有servlet必须实现javax.servlet.Servlet接口

servlet概念：servlet是一个java小程序，运行在web服务器（tomcat），通过http协议来接收来自客户端的请求以及对请求作出响应。

\*void service(ServletRequest req, ServletResponse res) ：这个方法用于接收请求和响应请求

ServletRequest req：请求对象。把请求信息都封装到这个对象中。通过这个对象可以获取请求信息。

ServletResponse res： 响应对象。响应信息都封装到这个对象中。通过这个对象修改响应信息。

## servlet开发步骤

2.1 手动开发servlet的过程

1）编写一个java类，继承javax.servlet.GenericServlet

覆盖service方法

package gz.itcast.servlet;

import javax.servlet.\*;

public class HelloServlet extends GenericServlet {

public void service(ServletRequest req, ServletResponse res)

throws ServletException, java.io.IOException {

//响应请求,向浏览器输出响应正文

res.getWriter().write("this is hello servlet!!!");

}

}

2）编译这个servlet类，得到class文件

javac -classpath C:\apache-tomcat-6.0.39\lib\servlet-api.jar -d . HelloServlet.java

3）把class文件，放到某个web应用的WEB-INF的classes目录下

4）声明servlet程序：修改web.xml文件

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

<web-app version="2.5"

xmlns="http://java.sun.com/xml/ns/javaee"

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

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

http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<!--servlet的配置 -->

<servlet>

<!-- servlet的名称 -->

<servlet-name>HelloServlet</servlet-name>

<!-- servelt的全名： 包+类-->

<servlet-class>gz.itcast.servlet.HelloServlet</servlet-class>

</servlet>

<!-- servlet的映射-->

<servlet-mapping>

<!-- 和上面的名称保持一致 -->

<servlet-name>HelloServlet</servlet-name>

<!-- servlet的访问路径 -->

<url-pattern>/hello</url-pattern>

</servlet-mapping>

</web-app>

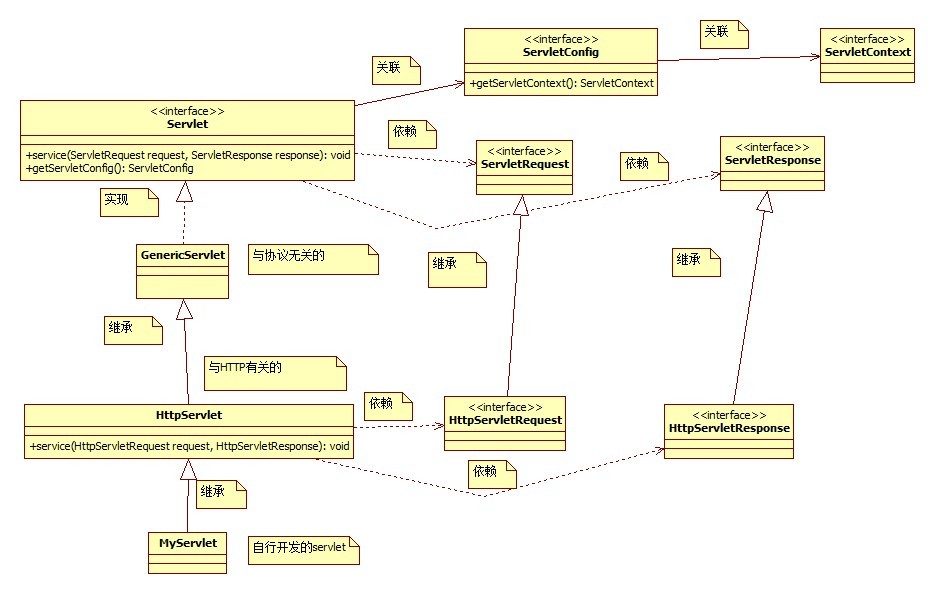
5）启动tomcat，访问

http://localhost:8080/day08\_servlet/hello

2.2 用myecplise进行开发servlet

三、servlet的执行过程

看图



四、servlet编写方式

1）两种方式

第一种：继承javax.servlet.GenericServlet 通用的和协议无关的servlet

第二种：继承javax.servlet.http.HttpServlet 基于HTTP协议的servlet，是GenericServlet的子类（以后推荐使用这种）

五、servlet生命周期\*（面试经常考）

1）研究servlet对象的 诞生，活着，死亡。

诞生：第一次访问servlet时

活着：只有servlet对象创建好了，就会常驻内存

死亡：web应用卸载或者tomcat服务器停止

2）重要的生命周期方法 （以下方法都是由web服务器调用的）

init方法：在servlet对象创建时调用。servlet默认情况下在第一次访问的时候创建对象。servlet对象在web服务器中是单例多线程的。只调用1次

\*service方法：每次请求发出时调用。调用n次

destory方法：servlet对象销毁时调用

注意：

init() 与init(ServletConfig)区别

init(ServletConfig):是servlet的生命周期方法，必定会被web服务调用。（通常不去覆盖）

init(): 是提供给开发者自定覆盖的，然后在此编写初始化代码的。（推荐覆盖这个方法作为初始化方式）

init()方法是在init(ServletConfig)方法里面调用。开发者只需要覆盖无惨的init方法即可，无需覆盖有参的init，因为GenericServelet中已经对有参的init进行了实现！

六、servlet线程安全问题（了解）

1）建议：

1.1 尽量在servlet的开发者不要使用成员变量

1.2 如果使用了成员变量，那么需要使用synchronized把使用到成员变量的代码

块锁住

七、servlet的一些细节

servlet的配置：

<servlet>

<servlet-name>HelloServlet</servlet-name>

<servlet-class>gz.itcast.servlet.HelloServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloServlet</servlet-name>

<url-pattern>/hello2</url-pattern>

</servlet-mapping>

1）<url-pattern>的写法

<url-pattern> 表示访问servlet的路径

url-pattern 浏览器的访问的URL

精确匹配 /hello http://localhost:8080/day08/hello

/itcast/hello http://localhost:8080/day08/itcast/hello

模糊匹配 /\* http://localhost:8080/day08/任意路径

\*.后缀 http://localhost:8080/day08/任意路径.后缀

（\*.do） ( http://localhost:8080/day08/任意路径.do)

（\*.action） ( http://localhost:8080/day08/任意路径.action)

注意：

1）url-pattern的开头必须是/或\*号 非法的： hello /\*.do

2) 多个url-pattern同时起作用

2.1 精确的url-pattern优先被调用!

2.2 以后缀名结尾的url-pattern优先级最低！

2)servlet的立即加载（创建对象）

默认情况下servlet是第一次访问。改变servlet创建的时机，改到服务器启动。

修改创建servlet时机：在servlet的配置里面加上<load-on-startup>

默认情况下servlet是第一次访问时，才创建了servlet对象，若在配置wab.xml里面加上<load-on-startup>，tomcat启动时就创建了servlet对象。

客户第一次时就不用创建servlet对象，访问的速度就会变快了。

如下：

<servlet>

<servlet-name>HelloServlet</servlet-name>

<servlet-class>gz.itcast.servlet.HelloServlet</servlet-class>

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

</servlet>

注意：写正整数，数字越大，优先级越低！！！

优先级是指tomcat启动时创建servlet对象的先后。

**八、ServletConfig对象**

为了不让数据写死在代码中，我就把经常被修改的数据放在配置文件中读取，所以很有必要学习怎么获取配置文件的对象，然后进行读取配置信息。

1)概念：servlet的配置对象。用来读取servlet的配置信息

2）ServletConfig对象如何得到？

web服务器创建了ServletConfig对象，通过init方法传入servlet中

最后是通过getServletConfig()方法获取对象

3）servlet配置信息（局部参数-只能在当前servlet可以用）

web.xml配置

<servlet>

<servlet-name>ServletDemo6</servlet-name>

<servlet-class>gz.itcast.servlet.ServletDemo6</servlet-class>

<init-param>

<param-name>path</param-name>

<param-value>c:/news2.txt</param-value>

</init-param>

</servlet>

servlet代码获取：

ServletConfig config = getServletConfig();

String path = config.getInitParameter("path");

**九、ServletContext对象\*（重点，难点）**

1）概念：ServletContext对象代表当前web应用（web工程），一个web应用（就是一个web工程）中只会创建一个ServletContext对象。（全局对象）一个web应用中的所有Servlet都共享一个ServletContext对象。

如果Servlet要共享一些数据，我可以用ServletContext对象去共享。

2）如何得到ServletContext对象？

通过SevletConfig对象得到：

getServletConfig().getServletContext();

或者

getServletContext() (推荐使用这种)

3）ServletContext的作用

3.1 可以作为域对象使用： 使用范围：整个web应用

Map<String,Object> map=new HashMap<String,Object> () ;

取出数据

java.lang.Object getAttribute(java.lang.String name)

保存数据

void setAttribute(java.lang.String name, java.lang.Object object)

清除数据

void removeAttribute(java.lang.String name)

3.2 配置全局参数(所有servlet都共享)

java.lang.String getInitParameter(java.lang.String name)

java.util.Enumeration getInitParameterNames()

配置全局参数

web.xm配置：

<context-param>

<param-name>encoding</param-name>

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

</context-param>

获取：

ServletContext context = getServletContext();

context.getInitParameter("encoding")

3.3 转发

getServletContext(). getRequestDispatcher(path).forward(request,response)

3.4 读取web应用的资源文件

1）在myecplise中运行java程序，当前路径指向的是当前java项目的根目录。

2）在servlet中使用当前路径，当前路径指向的是tomcat的bin目录

context.getRealPath() 返回路径的字符串

context.getResourceAsStream()返回输入流

## Response对象

1）作用：这个对象可以修改响应信息

2）常用的方法：

响应行：

http协议版本

状态码：

void setStatus(int sc) 修改状态码

描述

随着状态码的改变而改名

响应头：

void setHeader(java.lang.String name, java.lang.String value) 修改一个响应头信息

void setIntHeader(java.lang.String name, int value) 修改整数值的响应头信息

void setDateHeader(java.lang.String name, long date) 修改日期类型的响应头信息

空行：

响应正文

ServletOutputStream getOutputStream() 输出字节类型的正文（二进制（图片，文件））

java.io.PrintWriter getWriter() 输出字符类型的正文（字符串，文本）

3）案例

\*案例一：

演示输出正文作用：输出中文内容

ServletOutputStream out=response.getOutputStream();

//解决方案一：更改浏览器的码表（不建议）

//解决方案二(不建议)：向客户端输出一个<meta>标签,模拟了响应消息头：

<meta http-equiv='Content-Type' content='text/html;charset=UTF-8'>

//解决方案三（建议）:向客户端输出响应消息头：

response.setHeader("Content-Type", "text/html;charset=UTF-8");

//解决方案四（建议）：向客户端输出响应消息头，方案三的简化

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

out.write("JavaWeb很难啊".getBytes("utf-8"));

//方案五

//提醒Eclipse用utf-8编码

response.setCharacterEncoding("utf-8");

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

response.getWriter().write("你大爷啊");

\*案例二：

演示响应头： content-disposition 文件下载的提示

//下载就是一个复制的过程，把服务器里的资源复制到浏览器

File file=new File("f://bbxy.mp3");

response.setHeader("content-disposition","attachment;filename="+file.getName());

ServletOutputStream out=response.getOutputStream();

FileInputStream in=new FileInputStream(file);

byte[] buffer=new byte[1024];

int len;

while((len=in.read(buffer))!=-1){

out.write(buffer,0,len);

}

in.close();

out.close();

案例三：

生成随机验证码（了解）

案例四：定时刷新 refresh

案例五：

控制网页的缓存 expires 网页文缓存文件的过期时间（了解）

\*案例六：

请求重定向 原理： 302状态码 + location响应头

//方法一

response.setStatus(302);

response.setHeader("location", "/M9D24/baiduliangxia.html");

//方法二

response.sendRedirect("/M9D24/baiduliangxia.html");

## Request对象

1）作用：获取请求信息

2）常用的方法：

请求行：

请求方式

getMethod()

请求资源

getRequestURI() /day09\_04\_request/login.html

getRequestURL() http://localhost:8080/day09\_04\_request/login.html

http协议版本

getProtocol()

获取get方式提交的参数：

String getQueryString() get/day09\_04\_request/login.html?name=eric

请求头：

java.lang.String getHeader(java.lang.String name) 获取一个请求头

long getDateHeader(java.lang.String name) 获取日期类型的请求头

int getIntHeader(java.lang.String name) 获取整数类型的请求头

java.util.Enumeration getHeaderNames() 获取所有头的名称

java.util.Enumeration getHeaders(java.lang.String name) 获取同名多个值的请求头

空行：

请求正文：post提交的参数

ServletInputStream getInputStream()

3）案例

request.getRemoteHost()获取IP地址

**案例一：防止非法连接（防盗链）**  referer请求头

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

//防止盗链

//知道谁访问了我 referer: 当前请求来自于哪里

String referer = request.getHeader("referer");

System.out.println("referer="+referer);

//防止的情况：

//1)直接访问的， referer==null

//2)其他web应用 ！referer.constains("http://localhost:8080/day09\_04\_request”)

if(referer==null || !referer.contains("http://localhost:8080/day09\_04\_request")){

response.getWriter().write("你没有权限访问该资源.");

return;

}

response.getWriter().write("凤姐日记");

**案例二：\* 获取表单参数数据**

get提交：

getQueryString()

post提交：

getInputStream()

以上两种方法不通用

3.1 SUN公司设计通用的获取参数的方法（不管get或post都可以用）

java.lang.String getParameter(java.lang.String name) 获取一个参数 name=eric

java.lang.String[] getParameterValues(java.lang.String name) 获取一个同名多值的参数

name=eric&name=jacky 多选

java.util.Enumeration getParameterNames() 获取所有参数的名称 <String> 参数名

java.util.Map getParameterMap() 获取所有参数的 <String,String[]>

注意：使用BeanUtils拷贝表单参数到对象中

BeanUtils.populate(user, request.getParameterMap());

## 转发与重定向区别

请求重定向

response.sendRedirect("/M9D24/baiduliangxia.html");

1）发出2次请求

2）地址栏会改变

3）可以重定向到任何地址（应用内部和应用外部）

4）不能通过request域对象来共享数据

转发

getServletContext().getRequestDispatcher(path).forward(request,response)

1）发出1次请求

2）地址栏不会改变

3）只能转发到应用内部

4）可以通过request域对象来共享数据

## 验证码

// 生成一张图片

**int** width = 70, height = 25;

BufferedImage image = **new** BufferedImage(width, height,

BufferedImage.*TYPE\_INT\_RGB*);

// 拿到画笔

Graphics g = image.getGraphics();

// 设置背景颜色:1.设置画笔颜色为灰色；2.把整个图片涂满灰色；

g.setColor(Color.*gray*);

g.fillRect(0, 0, width, height);

// 字母数字集

String letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

String numbers = "0123456789";

// 创建随机数的对象

Random ran = **new** Random();

// 随机获取2个字母2个数字

**char**[] chars = **new** **char**[4];

chars[0] = letters.charAt(ran.nextInt(letters.length()));

chars[1] = letters.charAt(ran.nextInt(letters.length()));

chars[2] = numbers.charAt(ran.nextInt(numbers.length()));

chars[3] = numbers.charAt(ran.nextInt(numbers.length()));

// 将4个字符打乱顺序

String code = "";

Set<Integer> set = **new** HashSet<Integer>();

**int** size;

**while** (set.size() < 4) {

size = set.size();

**int** i = (**int**) (4 \* Math.*random*());

set.add(i);

**if** (size < set.size()) {

code += chars[i];

}

}

System.*out*.println(code);

request.getSession().setAttribute("code", code);

// 1.设置画笔的颜色；2.设置字体；3.画出来

g.setColor(Color.*black*);

g.setFont(**new** Font("宋体", Font.*ITALIC*, 20));// 宋体,斜体，大小为20号

g.drawString(code, 15, 20);

// 设置随机干扰空心圈

**for** (**int** i = 1; i <= 10; i++) {

**int** x = ran.nextInt(width);

**int** y = ran.nextInt(height);

**int** r = (**int**) (8 \* Math.*random*());

g.setColor(Color.*black*);

g.drawOval(x, y, r, r); // 实心圆g.fillOval(x, y, r, r);

}

// 设置随机干扰线

**for** (**int** i = 1; i <= 4; i++) {

**int** x1 = ran.nextInt(width);

**int** y1 = ran.nextInt(height);

**int** x2, y2;

**while** (**true**) {

x2 = ran.nextInt(40);

y2 = ran.nextInt(40);

**if** (x2 + y2 > 10) {

**break**;

}

}

g.setColor(Color.*black*);

g.drawLine(x1, y1, x1 + x2, y1 + y2);

}

// 把图片显示在网页

ServletOutputStream out = response.getOutputStream();

ImageIO.*write*(image, "png", out);

## Cookie技术

1）学习Cookie技术就是学习javax.servlet.http.Cookie对象

2) Cookie对象：

Cookie对象的概念：是一个对象，包含了一些信息（例如：商品数据），在服务器的servlet中创建，然后发送给浏览器保存。接着浏览器再发回给服务器。 （因为cookie保存到各个浏览器的缓存区中，可以标记唯一的浏览器，所以通常用于会话管理）

Cookie对象属性：

必填属性：

\*name cookie的名称

\*value cookie的值（例如：商品数据）

可选属性：

\*path cookie的路径 . 默认值：当前web应用 /day10\_01\_cookie

结论：

什么情况下能够取到浏览器发送的cookie？

答案： 只有满足 （访问cookie的路径）.startWith(保存cookie的路径)

"/day10\_01\_cookie".startWith("/day10\_01\_cookie/servlet") false 不ok

"/day10\_01\_cookie".startWith("/day10\_01\_cookie") true ok

结论：在实际开发，通常会把cookie的path设置为默认路径，那么在访问该应用下的所有资源都可以得到 这个cookie!!

domain cookie所在的主机（域名）

\*maxAge cookie的最大存活时间 。

1）（默认值）负数：如果是负数，则cookie存活在浏览器内存，如果浏览器关闭则cookie失效！

2）正数：cookie存活在浏览器的缓存文件中，单位是秒

3）0：则表示删除同名的cookie

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

comment cookie的备注

version cookie的版本

3）服务器发送给浏览器cookie数据

HttpServletResponse.addCookie(javax.servlet.http.Cookie)

响应头： set-cookie: time="2015-01-01"

注意：一个浏览器只能保存300个，一个网站只能保存20个，一个cookie最大不超过4KB

4）浏览器发送cookie给服务器

HttpServletRequest.getCookies()

请求头： cookie： time="2015-01-01"

注意：唯一的cookie： domain+path+name

## HttpSession技术

1）学习session技术就是学习一个javax.servlet.http.HttpSession对象

2）HttpSession对象也是一个域对象: 是多个请求的范围内

setAttribute("name",Object);

getAttribute("name");

removeAttribute("name")

3）创建HttpSession对象

HttpServletRequest.getSession(); 和 HttpServletRequest.getSession(true);创建和查询HttpSession对象

HttpServletRequest.getSession(false); 查询HttpSession对象

5)HttpSession的细节

5.1 默认情况下：HttpSession对象30分钟不访问，服务器自动回收

5.2 手动销毁HttpSession对象：void invalidate()

6）浏览器禁用了cookie之后，如何找到服务器HttpSession？

6.1 提示用户不要禁用cookie

6.2 使用URL重写功能

/IndexServlet;jsessionid=75964260580FE7083236EB814801E977

注意：如果没有禁用cookie，则不执行URL重写

# Jsp

1）jsp模板元素

模块就是就是jsp页面的html代码

2）jsp脚本

语法： <% %>

作用： 在jsp中嵌入java代码

原理：原封不动地翻译到java文件的\_jspService方法中

3）jsp表达式

语法：<%=变量或表达式%>

作用： 向浏览器输出变量或表达式计算的结果

原理： 翻译成out.print(name ); 输出内容

4）jsp注释

语法：<%!-- --%>

注意： jsp注释 vs html注释

html注释是会被翻译到java文件，也会被执行

jsp注释不会被翻译，不会被执行，在jsp页面中尽量使用jsp注释

5）jsp声明（面试的考）

语法： <%! %>

作用：声明jsp的变量和方法

注意：

1）声明的变量和方法作为java文件的成员变量和成员方法

2）不能声明和java文件一样的方法

3）jsp脚本中不能声明方法

四、Jsp三大指令

1）指令的作用： 在翻译jsp文件的过程中起作用，对jsp引擎起作用

2）三个指令

page： 用于定义jsp页面中的各个属性

include：用于包含其他页面

taglib：用于导入标签库文件

语法： <%@ 指令名称 属性1 属性2 %>

或者

<%@ 指令名称 属性1%>

<%@ 指令名称 属性2 %>

3)page指令： 用于定义jsp页面中的各个属性

language： 翻译时使用的语言

只能写java

import： 翻译时导入的类/包

默认导入java.lang.\*，除此之外都需要导入

快捷键： alt+/

同时导入多个包/类： import="java.util.\*,gz.itcast.Student"

contentType: 翻译时使用什么类型和码表输出正文内容

原理：翻译成response.setContentType("text/html;charset=UTF-8");

pageEncoding: 翻译时使用什么码表翻译jsp

在从jsp页面翻译到java源文件的过程中使用什么码表去翻译

注意：

1）jsp页面中有三个地方的码表问题：

a)保存jsp文件的码表：另存为

b）翻译jsp文件的码表：pageEncoding

c) 把内容输出到浏览器的码表： contentType

保存以上三个码表一致就不会出现中文乱码问题

2)实际开发中，使用myecplise工具开发

jsp文件的码表和输出给浏览器内容的码表会自动根据pageEncoding来设置 所以，只需要设置pageEncoding即可！！！

errorPage: 发生错误时指向哪个页面（在生错误的页面设置）

isErrorPage: 当前页面是否为错误页面 （在错误页面设置）

true：是错误页面，可以使用exception内置对象得到错误信息

false：不是错误页面。无法得到错误信息

全局的错误页面设置：

在web.xml文件添加配置

<error-page>

<error-code>500</error-code>

<location>/common/error.jsp</location>

</error-page>

<error-page>

<error-code>404</error-code>

<location>/common/404.html</location>

</error-page>

buffer: jsp页面的缓存区大小

默认8kb

session: 是否打开HttpSession会话功能

true: 打开session功能，提供session内置对象

false： 关闭，不提供session内置对象

isELIgnore: 是否忽略EL表达式

true： 忽略

false： 不忽略（默认值）

4）include指令：包含其他页面

file: 被包含的页面

原理： 被包含的页面内容翻译到包含页面中（先合并再翻译），只有1个源文件，叫静态包含

5）taglib指令：用于导入标签库

五、Jsp的9大内置对象（\*）

1）SUN设计Jsp的时候，为了简化在jsp页面使用一些常用的对象，在使用Jsp页面顶部为开发者定义了一些对象。这些对象就叫内置对象。

2）9个内置对象

名称 类型

request HttpServletRequest 请求对象，获取请求信息

response HttpServletResponse 响应对象，修改响应信息

session HttpSession 会话对象，会话管理

config ServletConfig servlet配置对象，读取servlet配置参数

application ServletContext servlet上下文对象，代表当前web作用。（4个作用）

out JspWriter 相对于带缓存功能的PrintWriter，用于输出内容

page Object 当前的jsp翻译的java对象

exception Throwable 异常对象，用于在jsp页面发生错误时获取异常信息

pageContext PageContext jsp的上下文对象。

3）PageContext对象

PageContext也是一个域对象。使用范围： 在当前页面范围内

作用：

3.1 通过PageContext对象可以得到其他8个内置对象。（在自定义标签的时候经常使用！！！！）

getRequest();

getResponse();

getSession();

getServletConfig()

getServletContext()

getOut()

getPage()

getException()

3.2 作为域对象共享数据（后面会用EL表达式替代）

a) 把数据保存到当前页面范围内：

setAttribute("name",Object);

getAttribute("name");

b) 把数据放到其他域范围中： request,session,application

setAttribute("name",Object,int scope);

getAttribute("name",int scope);

scope: PAGE\_SCOPE , REQUEST\_SCOPE , SESSION\_SCOPE, APPLICATION\_SCOPE

c) 自动在四个域范围内搜索数据

findAttribute("name")

从小到大依次按顺序搜索：

page\_scope - > request\_scope -> session\_scope -> application\_scope

六、Jsp的4个域对象（非常重要）

1）作用： 共享数据

2）域对象的方法

setAttribute(....)

getAtrribute(..)

removeAttribute(...)

3) 不同的与对象有不同的使用访问

PageContext -》page域： 当前jsp页面范围中有效

HttpServletRequest -> request域: 在当前的请求中有效

HttpSession - > session域： 在一次会话的多次请求中有效

ServletContext -> application域： 在当前web应用中有效

七、jsp的最佳实践

1) servlet技术 vs jsp技术

servlet技术： 擅长于写java代码（java业务逻辑）

jsp技术： 擅长于输出html代码

2）开发中设计到的逻辑

2.1 获取用户的参数数据(servlet)

2.2 执行业务逻辑，返回数据（去后台查询数据）(servlet)

2.3 显示数据效果(jsp) / 跳转到其他页面显示(servlet)

注意：在servlet中传递数据到jsp页面就必须使用域对象

七、Jsp的EL表达式（实际应用中使用这种语法代替jsp语法）

1）EL表达式作用： 代替jsp语法中的jsp表达式 EL 表达式语言

2）EL语法： ${变量或表达式}

\*\* 只能用于输出域对象中的内容 \*\*

3) EL的用途：

3.1 输出普通字符串

3.2 输出对象的属性

3.3 输出数组和List/Set集合的数据

3.4 输出Map集合的数据

4) EL运算符

算术运算： + - \* / %

比较运算：> < == != >= <=

逻辑运算： && || !

三目元素： ?:

判空： empty 既能判断null，也能判断空字符串

5）EL的内置对象

EL Jsp

${pageContext} -> <%=pageContext%>

${pageScope} -> <%=pageContext.getAttribute("xxx",PageContext.PAGE\_SCOPE)%>

${requestScope} -> <%=pageContext.getAttribute("xxx",PageContext.REQUEST\_SCOPE)%>

${sessionScope} -> <%=pageContext.getAttribute("xxx",PageContext.SESSION\_SCOPE)%>

${applicationScope} -> <%=pageContext.getAttribute("xxx",PageContext.APPLICATION\_SCOPE)%>

${param} -> <%=request.getParameter("xxx")%> 类型： Map<String,String>

${paramValues} -> <%=request.getParameterValues("xxx")%> 类型：Map<String,String[]>

${header} -> <%=request.getHeader("xxx")%> 类型：Map<String,String>

${headerValues} -> <%=request.getHeaders("xxx")%> 类型：Map<String,String[]>

${cookie} -> <%=request.getCookies()%> 类型：Map<String,Cookie>

${initParam} -><%=application.getInitParameter("xxx")%><context-param>类型： Map<String,String>

难度高的EL表达式

map根据key取值key为String类型

${GoodsMessMap['错误信息']}

嵌套

${test}为”key”

${GoodsMessMap[test]}

# 自定义标签

1）作用：自定义标签用于替代jsp中的jsp脚本 <%%>

2) 快速入门

需求： 向浏览器输出当前客户的IP地址

自定义标签实现步骤：

2.1 编写标签执行类，继承SimpleTagSupport类，覆盖doTag方法

//标签执行类， 继承SimpleTagSupport类

public class ShowIPTag extends SimpleTagSupport{

@Override

public void doTag() throws JspException, IOException {

//得到PageContext对象

PageContext pageContext = (PageContext)this.getJspContext();

//通过PageContext对象得到其他内置对象

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

JspWriter out = pageContext.getOut();

String ip = request.getRemoteHost();

out.write("客户端的IP地址为："+ip);

}

}

2.2 在WEB-INF目录下，建立一个tld文件，itcast.tld

<?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">

<!-- 标签库的版本通常为1.0 -->

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

<!-- 标签库的简单名称（前缀） -->

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

<!-- 标签库的uri名称，用于在jsp页面中导入tld文件 -->

<uri>http://gz.itcast.cn/</uri>

<!-- 一个标签的配置 -->

<tag>

<!-- 标签名称 -->

<name>showIP</name>

<!-- 标签执行类的全名 -->

<tag-class>gz.itcast.tag.ShowIPTag</tag-class>

<!-- 标签内容的输出格式 -->

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

</tag>

</taglib>

2.3 在jsp页面中使用标签（必须先导入再使用）

在jsp页面顶部使用taglib指令导入

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

在jsp页面里面使用标签：

<itcast:showIP></itcast:showIP>

3）自定义标签的执行过程

看图

标签执行类方法的执行过程：(查看SimpleTag接口)

3.1 setJspContext(JspContext pc ) 设置PageContext对象，JspContext是PageContext的父类

3.2 setParent( JspTag parent ) 设置父标签对象，如果没有父标签，则不执行次方法

3.3 setJspBody( JspFragment jspBody ) 设置当前标签的内容，如果没有标签内容，则不执行次方法

3.4 doTag() 执行标签的方法。

this.getJspContext(); // 得到jspContext（PageContext）

this.getParent();//得到父标签对象

this.getJspBody();//得到标签内容

结论：自定义标签只需要继承SimpleTagSupport类，然后覆盖doTag方法即可！！！

4) 自定义标签的作用：

4.1 控制标签内容是否显示

显示：this.getJspBody().invoke(null);

不显示： 什么不做

4.2 控制标签后面的内容是否显示

显示： 什么不做

不显示： 抛出一个SkipPageException

4.3 控制标签内容重复显示

多次调用：this.getJspBody().invoke(null);

带属性的标签：

1）标签执行类，添加一个setter方法，用于传递jsp页面的属性值

public void setNum(int num){}

2)在tld文中配置

<tag>

<attribute>

<name>

<required> 是否必填

<rtexprvalue> 是否需要支持EL表达式

<tag>

4.4 修改标签内容：

//1.创建临时容器

StringWriter sw = new StringWriter();

//2.把标签内容写出到临时容器中

JspFragment jspBody = this.getJspBody();

jspBody.invoke(sw);

//3.从临时容器中取出标签内容

String content = sw.toString();

//4.修改

content = content.toUpperCase();

//5.手动写出到浏览器

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

5)标签内容的输出格式

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

scriptless: 表示标签内容不能写java脚本（推荐）

empty: 表示必须是空标签 （注意：this.getJspBody()为null）

tagdependent: 表示标签内容可以写java脚本，但是不执行(当做普通字符串输出)。

JSP：表示标签内容可以写java脚本也可以执行。（不推荐）

# jsp标签

1)jstl标签

1.1 概述

SUN公司设计的一套通用的自定义标签库

java standard tag libaray java的标准标签库

5套标签库：

\*核心标签库 c标签 （最常用的）

国际化标签库 fmt标签

fn函数库 fn

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

xml标签库 x

sql标签库 sql

1.2 核心标签库(core或者c)

常用的标签：

<c:set></c:set> 保存数据到域对象中

<c:out value=""></c:out> 从域对象取出并显示数据

<c:if test=""></c:if> if条件判断

<c:choose></c:choose><c:when test=""></c:when><c:otherwise></c:otherwise> if..else条件判断

<c:forEach></c:forEach> for循环（数组或集合）

<c:forTokens items="" delims=""></c:forTokens> for循环（特殊字符串）

<c:redirect></c:redirect> 请求重定向

<c:url></c:url> 简化URL写法的标签

items: 表示需要遍历的数据（必须从EL取出）

begin: 开始角标

end: 结束角标

step: 每次加几 默认+1

var: 每个对象的名称

varStatus=*"status"*统计当前的循环次数，${status.count}取值

<itcast:forEach items="${list }" begin="0" end="2" step="1" var="stu">

姓名：${stu.name } &nbsp;年龄：${stu.age }<br/>

</itcast:forEach>

<c:choose>

<c:when test="${oldMess.dis==str}">

<option value=*"*${str}*"* selected=*"selected"*>${str}</option>

</c:when>

<c:otherwise>

<option value=*"*${str}*"*>${str}</option>

</c:otherwise>

</c:choose>

2)动作标签

SUN设计的标签： 特点：不同导入标签库，直接使用即可！！

# mysql基础

一、

优点:数据永远保存下来,存取相对快，查询非常方便（sql语法）

缺点:依赖数据库软件

## 二、mysql基本使用

1）登录mysql

mysql -u root -p

输入密码 回车

2）发送sql语句

三、管理数据库

1.1 查询当前所有数据库

show databases;

information\_schema | mysql的元数据

mysql | mysql软件的用户信息

performance\_schema | mysql软件运行过程中的性能数据

test | 测试数据库

1.2 创建数据库

create database day15;

create database day15 defacult character set utf8; 指定字符集

1.3 查询数据库的字符集

show create database day15;

1.4 修改数据库字符集

alter database day15 defacult character set utf8;

1.5 删除数据库

drop database day15;

四、管理表

4.1 创建表

create table student(name varchar(20),gender varchar(1),age int);

每列叫字段。字段格式： 字段名称 字段类型

字段类型：

char vs varchar

char(5): 固定长度字符串 eric 占用5个

varchar(5): 可变长度字符串 eric 占用4个

date vs datatime vs timestamp

date: 日期类型 2015-10-06

datetime： 日期时间类型 2015-10-06 10:45:30

timestamp: 时间戳，用于记录数据更新的时间（通常这个字段不会修改）

4.2 选择数据库

use day15;

4.3 查看所有表

show tables;

4.4 查看表结构

desc student

4.5 修改表

添加字段： alter table student add column remark varchar(100);

修改字段类型：alter table student modify column name varchar(50);

修改字段名称：alter table student change column name username varchar(50);

删除字段：alter table student drop column remark;

修改表名称： alter table student rename to teacher;

4.6 删除表

drop table student;

五、管理数据（CUD）

5.1 插入数据

insert into student values(值1，值2，值3...); 不指定字段

\*insert into student(username,gender) values('jacky','m'); 指定字段

注意：

1）一定要按照表的字段顺序插入

2）字段的数量一定要和值的数量保持一致

3）默认不指定字段就是插入所有字段

5.2 修改数据

update student set 字段1=值1,字段2=值2; 全表修改

\*update student set gender='m' where username='jacky'; 按条件修改一个字段

update student set gender='f',age=30 where username='jacky'; 按条件修改多个字段

5.3 删除数据

delete from student; 全表删除

delete from student where username='jacky'; 按条件删除

delete from student where username in('m','rose'); 按条件删除多个记录

truncate table student; 全表删除

注意：

delete from vs truncate table

1）delete from只能临时删除数据，如果使用事务，可以回滚；truncate table是永久删除数据，不能回滚；

2）delete from可以全表，也可以按条件删除；但是truncate table只能全表删除不能跟条件。

3）delete from不能影响表的约束;但是truncate table可以影响表的约束

(演示自增长约束)

六、查询数据（重点） select

6.1 查询所有字段

select \* from 表

6.2 查询指定字段

select 字段1,字段2 from 表

6.3 查询时使用别名

select 字段1 as 别名 from 表

6.4 查询时添加常量

select 常量 from 标签

6.5 查询时合并列

select 字段1+字段2 from 表

6.6 查询时去重 去重复

select distinct 字段1 from 表

6.7 条件查询

select \* from 表 where

1）逻辑： and or

2)比较： > >= <= = <> between and

3)判空： is null is not null ='' <>''

4)模糊： like (% \_)

6.8 分页查询

select \* from 表 limit 起始行,每次查询行数;

6.9 聚合查询

select max(字段) from 表;

select min(字段) from 表;

select avg(字段) from 表;

select count(字段) from 表;

6.10 查询后排序

select \* from 表 order by 字段

asc:升序

desc: 降序

——按照chinese升序排列

select \* from student order by chinese asc;

6.11 分组查询

select 字段 from 表 group by 字段;

——按照性别分组，并且统计数目

select gender,COUNT(\*) from staff group by gender ;

6.12 分组后筛选

select 字段 from 表 group by 字段 having 条件;

——按照性别分组，并且统计数目大于1的

select gender,COUNT(\*) from staff group by gender having COUNT(\*)>1;

## 修改mysql的用户权限

7.1 添加用户权限

GRANT SELECT ON day15.student TO 'jacky'@'%' IDENTIFIED BY '1234'

7.2 删除用户权限

REVOKE DELETE ON day15.student FROM 'jacky'@'localhost';

对sql语句进行分类：

DDL: 数据定义语句 ： create alter drop

DML: 数据操作语句 ： insert update delete truncate

DQL: 数据查询语句： select show

## 数据约束

1）数据约束概述

为了保证数据的完整性，需要对表添加约束，从而限制用户的输入，达到数据完整性的要求！！！

2）数据约束分类

默认值（default）

非空(not null)

唯一(unique)

主键(primary key)（唯一+非空）

外键(foreign key)

自增auto\_increment

作用：解决两种表的数据完整性问题

CONSTRAINT employee\_dept\_fk FOREIGN KEY(deptId) REFERENCES dept(id)

外键细节：

1）有了外键，在插入数据时，先插入主表，再插入副表

2）有了外键，在修改数据时，先修改副表，再修改主表

3) 有了外键，在删除数据时，先删除副表，再删除主表

级联操作（在外键的基础上添加级联）

作用：有了外键后，在修改或者删除数据时，应该先修改或删除副表数据，再修改或删除主表数据。但是我们希望直接修改或删除主表数据，能够影响副表数据，这时需要用到级联操作。

ON UPDATE CASCADE： 级联修改。当修改主表的时候，影响副表数据。

ON DELETE CASCADE ：级联删除。当删除主表的时候，影响副表数据。

自增长：作用：把主键的值交给了数据库维护。 从1开始，往上递增

## 三大范式

**第一范式**： 表的每个字段都必须是独立的不可分割的单元

**第二范式**：在第一范式的基础，要求表的除主键外的其他字段都和主键有依赖关系。

（一张表表达一个意思）

**第三范式**：在第二范式的基础上，要求表的除主键外的其他字段都和主键有直接决定关系

## 多表查询

**1.交叉查询**

SELECT employee.empName,dept.deptName FROM employee,dept;

**2.内连接查询**

SELECT e.empName,d.deptName

FROM employee e, dept d

WHERE e.deptId=d.id;

**或者**

SELECT e.empName,d.deptName

FROM employee e

INNER JOIN dept d

ON e.deptId=d.id;

**3.1左外部查询**

SELECT d.deptName,e.empName

FROM dept d

LEFT OUTER JOIN employee e

ON e.deptId=d.id;

**3.2右外部查询**

SELECT d.deptName,e.empName

FROM employee e

RIGHT OUTER JOIN dept d

ON e.deptId=d.id;

**4.自连接查询**

SELECT e.empName AS '员工',b.empName AS '上司'

FROM employee e

LEFT OUTER JOIN employee b

ON e.bossId=b.id;

## 存储过程

建立存储过程

DELIMITER $

CREATE PROCEDURE pro\_test1()

BEGIN

SQL语句;

END $

调用存储过程

CALL pro\_test1();

**mysql的变量分类**

全局变量：使用范围在mysql应用存活周期中，整个mysql应用是共享的

定义： set @@变量=值

查询： select @@变量

常见的全局变量：

character\_set\_client： mysql接收的数据字符集

character\_set\_results：mysql输出的数据字符集

会话变量: 使用范围在当前登录的会话中，只有在当前登录会话中有效的

定义： set @变量=值

查询： select @变量

局部变量：使用范围只在存储过程中，只有在当前存储过程中有效的

定义：declare 变量 修改： set 变量=值

查询：select 变量

练习： 编写一个存储过程

如果学生的英语平均分小于等于70分，则输出'一般'

如果学生的英语平均分大于70分，且小于等于90分，则输出‘良好’

如果学生的英语平均分大于90分，则输出‘优秀’

DELIMITER $

CREATE PROCEDURE pro\_avg(OUT str VARCHAR(20))

BEGIN

DECLARE savg FLOAT DEFAULT 0.0;

-- 统计平均分

SELECT AVG(english) INTO savg FROM student2;

IF savg<=70 THEN

SET str = '一般';

ELSEIF savg>70 AND savg<=90 THEN

SET str = '良好';

ELSE

SET str='优秀';

END IF;

SELECT savg;

END $

删除一个存储过程

DROP PROCEDURE pro\_avg;

调用存储过程

CALL pro\_avg(@str);

查看

SELECT @str;

# JDBC基础

## jdbc概述

jdbc 就是java与数据库的链接，是指使用java代码发送sql语句给数据库

jdbc其实是一套sun公司制定的java程序操作数据库的接口。

## jdbc开发步骤

//注册驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

//登录获取链接

Connection conn=DriverManager.*getConnection*("

jdbc:mysql://localhost:3306/day15","root","root");

//创建statement

Statement stmt=conn.createStatement();

//执行sql语句

ResultSet rs=stmt.executeQuery("select \* from student where id=1");

//处理结果

**if**(rs.next()){

**int** id=rs.getInt("id");

String name=rs.getString("name");

**float** chinese=rs.getFloat("chinese");

**float** english=rs.getFloat("english");

**float** math=rs.getFloat("math");

System.*out*.println(id+" "+name+" "+chinese+" "+english+" "+math);

}

//关闭

stmt.close();

conn.close();

## jdbc的api体系

核心接口：

|-Driver： 驱动程序接口。用于获取数据库连接

connect(url,properties) 连接数据库的方法

|-Connection: 代表一次数据库连接

Statement createStatement(); 创建Statement对象

PreparedStatement preprareStatement(sql) 创建PreparedStatement对象

CallableStatement prepareCall(String sql) 创建CallableStatement对象

|-Statement：发送静态sql语句

executeUpdate(sql): 发送更新类sql语句（DDL+DML）

ResultSet executeQuery(sql): 发送查询类sql语句（DQL）

|-PreparedStatement:发送预编译sql语句

executeUpdate(): 发送更新类sql语句（DDL+DML）

ResultSet executeQuery(): 发送查询类sql语句（DQL）

|-CallableStatement ：发送存储过程的sql语句（支持预编译）

executeQuery(): 执行调用存储过程的sql语句

|-ResultSet: 处理结果集。

next(): 将光标移动到下一行

getObject(...): 取出当前列的值

getInt()

getString()

**Statement接口**

用于执行静态sql语句

executeUpdate(sql)

executeQuery(sql) 返回ResultSet

**PreparedStatement接口**

Statement与 PreparedStatement区别

1）语法角度

Statement 只支持静态sql，不支持预编译sql

PreparedStatement支持预编译sql

2）执行效率

PreparedStatement会比Statement的执行效率更高

Statement不能利用数据库的sql缓存区

PreparedStatement可以利用数据库的sql缓存区

3）安全角度

Statement不能防止sql注入

PreparedStatement可以防止sql注入

**CallableStatement接口**

用于执行存储过程

1)调用带有输入参数的

// 注册驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

// 登录获取链接

Connection conn = DriverManager.*getConnection*(

"jdbc:mysql://localhost:3306/day16", "root", "root");

// 2)创建CalalbeStatment

CallableStatement stmt = conn.prepareCall("call pro\_test1(?)");

stmt.setInt(1, 1);

// 3)执行sql(调用存储过程只有executeQuery方法)

ResultSet rs = stmt.executeQuery();

// 4)处理

**while** (rs.next()) {

**int** id = rs.getInt("id");

String name = rs.getString("name");

**int** age = rs.getInt("age");

System.*out*.println(id + "\t" + name + "\t" + age);

}

// 5)关闭

stmt.close();

conn.close();

2）调用带有输出参数的

// 注册驱动程序

Class.*forName*("com.mysql.jdbc.Driver");

// 登录获取链接

Connection conn = DriverManager.*getConnection*(

"jdbc:mysql://localhost:3306/day15", "root", "root");

// 2)创建CalalbeStatment

CallableStatement stmt = conn.prepareCall("call pro\_test2(?,?)");

// 输入参数

stmt.setInt(1, 2);

// 设置输出参数

// 参数二： 注册数据库类型（注册输出参）

stmt.registerOutParameter(2, java.sql.Types.*VARCHAR*);

// 3)调用存储过程

stmt.executeQuery();

// 4)获取输出参数的值

// 注意：这里的下标和赋值的时候的下标是一样

String name = stmt.getString(2);

System.*out*.println(name);

// 5)关闭

stmt.close();

conn.close();

# JDBC加强

## jdbc批处理

addBatch(sql) 把sql语句添加到缓存区

executeBatch() 把缓存区的所有sql语句发送到数据库

clearBatch() 清空缓存区

**Statement类型例子**

Connection conn=JdbcUtil.*getConnection*();

Statement stmt=conn.createStatement();

stmt.addBatch("insert into student values(8,'哎雯',89,94,90)");

stmt.addBatch("insert into student values(9,'爱丽丝',77,85,80)");

stmt.executeBatch();

JdbcUtil.*close*(stmt, conn);

**PreparedStatement 类型**

String sql="insert into student values(?,?,?,?,?)";

Connection conn=JdbcUtil.*getConnection*();

PreparedStatement stmt=conn.prepareStatement(sql);

**for**(**int** i=1;i<=3;i++){

stmt.setInt(1, 10);

stmt.setString(2, "爱丽丝菲尔");

stmt.setFloat(3, 99);

stmt.setFloat(4, 99);

stmt.setFloat(5, 99);

stmt.addBatch();

}

stmt.executeBatch();

JdbcUtil.*close*(stmt, conn);

## jdbc获取自增长的值

String clazzSql="insert into clazz(cname) values(?)";

Connection conn=JdbcUtil.*getConnection*();

//1)指定该PreparedSatement可以返回自增长的键值

PreparedStatement stmt=conn.prepareStatement

(clazzSql,Statement.*RETURN\_GENERATED\_KEYS*);

stmt.setString(1, "math");

stmt.executeUpdate();

//2)从PreparedSatement取出自增长的键值

ResultSet rs=stmt.getGeneratedKeys();

**int** i = 0;

**if**(rs.next()){

i=rs.getInt(1);

}

String studentSql="insert into student2(name,cid) values(?,?)";

stmt=conn.prepareStatement(studentSql);

stmt.setString(1, "Yang");

stmt.setInt(2, i);

stmt.executeUpdate();

JdbcUtil.close(stmt, conn);

## jdbc处理大容量文件

用于文件存储

mysql:

字符文件： text MEDIUMTEXT longtext (txt html xml)

64kb 16M 4GB

字节文件： blob MEDIUMBLOB longblob (图片，视频，音乐....zip)

64KB 16M 4GB

oracle:

字符文件： clob longclob (txt html xml)

字节文件： blob longblob (图片，视频，音乐....zip)

mysql:int varchar

oralce:number varchar2

jdbc的操作的api：

字符文件：

写出： stmt.setClob(...)

读取： rs.getClob(..)

字节文件：

写出： stmt.setBlob(...)

读取： rs.getBlob(...)

**例子存储图片信息**

**存储：**

String sql="insert into attachments(id,name,file,addtime,author) values(?,?,?,?,?)";

Connection conn=JdbcUtil.*getConnection*();

PreparedStatement stmt=conn.prepareStatement(sql);

stmt.setString(1, UUID.*randomUUID*().toString());

stmt.setString(2, "Jack");

InputStream input=**new** FileInputStream("E:\\图片\\Hd.jpg");

stmt.setBlob(3, input);

stmt.setTimestamp(4, **new** java.sql.Timestamp(**new** Date().getTime()));

stmt.setString(5, "雨果");

stmt.executeUpdate();

JdbcUtil.*close*(stmt, conn);

**读取**

String sql="select file from attachments where id='79e8f9af-39ad-4968-80f1-7054c1ecc937'";

Connection conn=JdbcUtil.*getConnection*();

PreparedStatement stmt=conn.prepareStatement(sql);

ResultSet rs=stmt.executeQuery();

InputStream input=**null**;

**if**(rs.next()){

Blob blob=rs.getBlob(1);

input=blob.getBinaryStream();

}

OutputStream output=**new** FileOutputStream("F:/hyrz.jpg");

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

**int** len;

**while**((len=input.read(buffer))!=-1){

output.write(buffer, 0, len);

}

JdbcUtil.*close*(stmt, conn);

output.close();

input.close();

## jdbc事务

4.1 数据库事务

1)数据库事务支持

引擎：innoDB 支持事务，执行效率比MyISAM低

MyISAM 不支持事务

2）什么是事务?

事务，就是多个sql（更新类DDL+DML）语句要么一起执行成功，要么一致执行失败！

3)数据库事务操作命令

set autocommit=0; 打开事务

commit; 提交事务

rollback 回顾事务

4.2 jdbc事务

Connection.setAutoCommit(false) 打开事务

Connection.commit(); 提交事务

Connection.rollback(); 回顾事务

常用语法结构：

Connection conn = JdbcUtil.*getConnection*();

**try** {

String sql=”……”;

conn.setAutoCommit(**false**);

……

conn.commit();

} **catch** (SQLException e) {

e.printStackTrace();

//3)遇到异常，事务回滚

**try** {

conn.rollback(); // rollback;

} **catch** (SQLException e1) {

e1.printStackTrace();

}

}

4.3 事务四个特征（面试经常考）

原子性（A）：事务是不可分割的，要么一起成功，要么一起失败

一致性（C）：事务必须使数据库从一个一致性状态转到另一个一致性的状态

\*隔离性（I）：多个并发事务之间应该相互隔离

持久性（D）：事务一旦提交，应用永远生效。

4.4 隔离性

多个并发事务之间应该相互隔离。

如果并发事务没有隔离，可能会引发以下现象：

脏读：一个事务读到了另一个事务没有提交（commit）的更新数据

不可重复读：一个事务读到了另一个事务已经提交（commit）的更新数据

幻读：一个事务读到了另一个事务已经提交（commit）的插入数据

默认情况下，mysql数据库的隔离级别： repeatable-read

任何数据库都可以设置四个隔离级别：

read uncommitted 任何现象都不能防止

read committed 可以防止脏读

repeatable read 可以防止脏读和不可重复读

serizalizable 可以防止所有现象

修改数据库隔离级别：

set global transaction isolation level read committed; (全局)

查询数据库当前隔离级别：

SELECT @@global.tx\_isolation;

为了考虑数据库的安全性，才设置了隔离级别。

隔离级别越高，安全性越高！数据库并发效率越低！

# jdbc优化

## 一、连接优化

0）起因：在实际当中，多用户获取链接访问数据库，但是由于数据库创建连接时需要消耗相对大的资源，而且创建连接使用的时间又相对较长，并且极易造成数据库服务器内存溢出、拓机。

1）作用：

1.1 提高连接对象的使用率（从而提供程序的执行效率）

1.2 控制java程序和数据库的连接数，从而保护了数据库

SUN指定了连接池的接口：

javax.sql.DataSource .getConnection() 获取连接的方法

2）自定义连接池：

2.1 设计一个容器来存放若干连接对象

2.2 设计初始化连接

2.3 设计最大连接数

1.1代理模式

1）代理模式的要素:

代理接口：表示一系列功能

被代理类 ：实现代理接口

代理类：和被代理类实现同一系列的接口

2) 静态代理： 代理类是由开发者自行编写的。

代理类写法：

2.1 代理类实现和被代理类一样的接口

2.2 代理类声明一个被代理类的（代理接口）成员变量

2.3 在构造方法中传入被代理类的实例

2.4 编写代理接口中的实现方法。（改造被代理类的方法）

3）动态代理：代理类是程序生成的

使用的jdk中java.lang.reflect.Proxy类生成代理类的实例

返回代理类的实例：

static Object newProxyInstance(

ClassLoader loader, 定义类的类加载器。固定的（跟原理）

Class<?>[] interfaces, 代理类要实现的接口列表。固定写法：写被代理类实现的 接口类别

InvocationHandler h 代理之后的策略（代理策略） 用了策略设计模式

策略接口 具体使用什么策略（InvocationHandler的实现类）

Object invoke(Object proxy, 代理类实例

Method method, 代理接口的方法

Object[] args 代理方法的实际参数

)

)

1.3 使用代理模式改造Connection的close方法

问题：我们需要调用Connection的close方法时不是关闭连接，而是放回连接池

3.1 使用静态代理模式改造方案（代理类是由开发者自行编写的）

Connection conn = DriverManager.getConnection(url, user, password);

//创建代理类

Connection connProxy = new MyConnection(conn,this);

3.2 使用动态代理模式改造方法（程序生成的代理类）

//创建代理类（使用动态代理）

Connection connProxy = (Connection)Proxy.newProxyInstance(

MyDataSource.class.getClassLoader(),

new Class[]{Connection.class},

new InvocationHandler() {

//代理策略

@Override

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

throws Throwable {

//close方法需要改造

if(method.getName().equals("close")){

//把连接对象方法回连接池

pool.addLast(conn);

}else{

//其他方法不改

return method.invoke(conn, args);

}

return null;

}

});

1.4 使用市面上常用的连接池工具

必须实现javax.sql.DataSource

1) DBCP连接池

Database Connection Pool 数据库连接池

是apache组织的产品 comoms-\*

使用步骤：

1）导入jar包 （commons-dbcp.jar commons-pool.jar）

2) 创建BasicDataSource对象

getConnection方法

方式一：代码设置参数，自行创建 new BasicDataSource()

方式二： 配置文件设置参数，从BasicDataSourceFactory.createDataSource创建(推荐)

dbcp.properties

2) C3P0连接池（推荐使用）

hibernate框架推荐使用c3p0作为连接池

使用步骤：

1）导入jar包 c3p0-0.9.1.2.jar

2) 创建ComboPoolDataSource对象

getConnection方法

方式一:代码设置参数：

方式二：c3p0-config.xml配置设置参数(推荐)

## 二、代码优化

（设计一个通用的jdbc框架）

2.1 jdbc的元数据

作用：写出通用的代码

数据库元数据：DatabaseMetaData接口 专门用于获取数据库相关的信息

getDatabaseProductName() 数据库产品名称

getDatabaseMajorVersion() 数据库版本

getDatabaseMinorVersion()

驱动：

getDriverName()

getDriverMajorVersion()

getDriverMinorVersion()

\*参数元数据： ParameterMetaData接口 专门用于获取参数的信息

select \* from student where id=?

stmt = conn.prepareStatement(sql)

有几个参数：

getParameterCount()

\*结果集元数据：ResultSetMetaData接口 专门用于获取结果集的信息

select \* from student ；

ResultSet rs

有几列：

getColumnCount()

列名称：

getColumnName()

2.2 设计一个通用的jdbc框架

DBAssist类：

通用的更新方法： DDL+DML

update() 用到参数元数据

通用的查询方法： DQL

query()

# DBUtils

1）概述：DBUtils是对jdbc的简单封装 ，也是apache组织的框架

2）作用：使用java程序进行数据库操作

3）使用步骤：

3.1 导入dbutils的jar包

3.2 核心类：QueryRunner

update(...)

query(.....ResultSetHandler)

ResultSetHandler接口: 用于把结果集封装成不同的对象

DBUtils提供了默认实现

\***BeanHandler**: 把结果集的第一行记录封装一个javabean对象

\***BeanListHandler**: 把结果集的全部记录封装一个List对象（List对象存放javabean）

（约定： 表的字段名称和javabean的属性名称保持一致）

**ArrayHandler**：把结果集的第一行记录封装一个对象数组

**ArrayListHandler**：把结果集的全部记录封装一个List集合（List存放对象数组）

**ScalarHandler**：把聚合查询(count(\*) max(\*) min(\*))的结果封装成一个对象返回Long类型

**ColumnListHandler()**把查询的一列用列表封装起来

Long内有方法转换成int类型intValue()

//创建QueryRunner对象

QueryRunner qr=**new** QueryRunner(C3P0Util.*getDataSource*());

//执行sql语句

qr.update(“create table student (id int , name varchar(20))”);

Student student=(Student)qr.query(“select \* from student where id=?”,new

BeanHandler(student.class),Object[]{1});

# 分页查询

1)关键点： 设计一个分页的javabean分页对象PageBean，用于封装分页相关的数据

List<T> data 当前页数据

int firstPage 首页

int prePage; 上一页

int nextPage; 下一页

int totalPage; 总页数/尾页

int totalCount 总记录数

int curPage 当前页

int pageSize 每页显示记录数

int size 当前页数据的长度

# 条件查询

1）关键点：在于条件查询的sql语句

-- 关键：根据不同的条件，拼凑出完整的条件查询sql语句

SELECT \* FROM department WHERE 1=1

-- 如果部门名称不为空

AND deptName LIKE '%xxx%'

-- 如果负责人不为空

AND principal LIKE '%张%'

-- 如果职能不为空

AND functional LIKE '%秘书%

# jdbc多表操作

一对多

一对一

多对多

1）多表关系的表结构：

一对多： 客户 订单

表结构： 外键关联

多对多： 老师 学生

表结构： 关系表（中间表）

一对一： 公民 身份证

表结构： 1）唯一外键 2）主键关联

2）ORM概念： Object Relational Mapping 对象-关系-映射

数据库（关系） 面向对象

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

表 类

一条数据 一个对象

字段 属性

DBUtils是jdbc的简单封装，也是简单ORM框架

BeanHandler ResultSet -> JavaBean

BeanListHandler ResultSet-> List<JavaBean>

3) 关联关系：

一对多（多对一）： 客户 订单

对象关系：

客户类：Customers （一）

class Customers{

int id;

String name;

List<Orders> orders;

}

订单类: Orders（多）

class Orders{

int id;

String orderno;

Customers customer; //客户

}

关联关系是有方向的。

多对多： 老师 学生

对象关系：

老师类：Teachers

class Teachers{

int id;

String name;

List<Students> students;// 学生

}

学生类：Students

class Students{

int id;

String name;

List<Teachers> teachers;//老师

}

一对一： 公民 身份证 （很少用）

对象关系：

公民类： Persons

class Persons{

int id;

String name;

Idcards idcards; //身份证

}

身份证类： Idcards

class Idcards{

int id;

String idno;

Persons persons; //公民

}

# 使用类路径读取文件

1）使用步骤

a）得到类对象

Class clazz = JdbcUtil.class;

b)调用类对象的方法读取文件

clazz.getResourceAsStream("/");

这个斜杠指定当前项目的类路径的根目录

# 过滤器

1）什么是过滤器？

过滤器就是一个接口javax.servlet.Filter

servlet技术的三个组件:

1）javax.servlet.Servlet接口：用于处理请求和响应

2）javax.servlet.Filter接口： 过滤器 ？？？

3）javax.servlet.XXXListener接口： 监听器 ????

以上组件共同的特点：

1）开发web应用的组件

2）需要web服务器（tomcat）运行

3）需要在web应用的web.xml文件配置

<filter>

<filter-name>UserFilter</filter-name>

<filter-class>cn.itcast.login.UserFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>UserFilter</filter-name>

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

</filter-mapping>

过滤类型： 过滤器在发出什么类型的请求时会生效！

<dispatcher></dispatcher>

\*\*REQUEST：（默认）直接访问的请求（从浏览器直接发出的请求）

\*FORWARD：使用转发的方法向目标资源发出请求

\*INCLUDE： 使用包含的方法向目标资源发出请求

ERROR： 使用错误页面方法向目标资源发出请求

生命周期：与servlet相同

构造方法： 创建对象时调用（过滤器是单例的）

init： 初始化方法，创建完过滤器对象之后调用

doFilter： 每次发出请求目标资源时

destroy：销毁过滤器对象时调用

配置过滤器参数

1)在web.xml文件中

<filter>

<!-- 过滤器内部名称 -->

<filter-name>HelloFilter</filter-name>

<filter-class>gz.itcast.a\_filter.HelloFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

2）读取参数

FilterConfig.getInitParamter("encoding")

**过滤器链：**一个目标资源同时被多个过滤器所过滤，则形成一个过滤器链

过滤器链的优先级问题：是由<filter-mapping>的顺序决定的，先配置的先执行！！

实列：请求编码 类要继承Filter父类重写3个方法中的doFilter（装饰者模式:request）

权限过滤

网页数据压缩

## 请求编码

**public** **void** doFilter(ServletRequest request, ServletResponse response,

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

// POST

request.setCharacterEncoding("utf-8");

//response.setCharacterEncoding("utf-8");

//处理响应的字符编码

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

//装饰

MyHttpRequest myRequest = **new** MyHttpRequest((HttpServletRequest)request);

//放行

chain.doFilter(myRequest, response);

}

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

}

**public** **void** destroy() {

}

}

//1)编写一个装饰类，继承被装饰类

**class** MyHttpRequest **extends** HttpServletRequestWrapper{

//2)在装饰类中声明一个被装饰类的变量

**private** HttpServletRequest request;

**public** MyHttpRequest(HttpServletRequest request) {

**super**(request);

//3)在装饰类的构造方法中接收被装饰类的实例

**this**.request = request;

}

//4)增强被装饰类的方法

@Override

**public** String getParameter(String name) {

**try** {

//获取原来的数据(存在问题的数据)

String value = request.getParameter(name);

//对GET方法进行处理

**if**("GET".equals(request.getMethod())){

//手动解码

value = **new** String(value.getBytes("iso-8859-1"),"utf-8");

}

**return** value;

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

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

}

}

**public** String[] getParameterValues(String name) {

**try** {

//获取原来的数据(存在问题的数据)

String[] values = request.getParameterValues(name);

//对GET方法进行处理

**if**("GET".equals(request.getMethod())&&values!=**null**){

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

//手动解码

values[i] = **new** String(values[i].getBytes("iso-8859-1"),"utf-8");

}

}

**return** values;

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

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

}

}

**public** Map<String,String[]> getParameterMap() {

**try** {

//获取原来的数据(存在问题的数据)

Map<String,String[]> value = request.getParameterMap();

//对GET方法进行处理

**if**("GET".equals(request.getMethod())&&value!=**null**){

**for**(Entry<String,String[]> en:value.entrySet()){

**for**(**int** i=0;i<en.getValue().length;i++){

//手动解码

en.getValue()[i]= **new** String(en.getValue()[i].getBytes("iso-8859-1"),"utf-8");

}

}

}

**return** value;

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

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

}

}

## 权限过滤

//doFilter方法内代码

HttpServletRequest request=(HttpServletRequest)req;

HttpServletResponse response=(HttpServletResponse)resp;

String loginInfo=(String)request.getSession().getAttribute("loginInfo");

**if**(loginInfo!=**null**&&!loginInfo.equals("")){

chain.doFilter(request, response);

}**else**{

response.sendRedirect(request.getContextPath()+"/login.jsp");

}

# 装饰者模式

1)使用场景： 如果某个类的方法不满足我们的需要，这时可以使用装饰者模式对其进行装饰，然后增强其方法。

2）装饰者元素：

被装饰者：不满足需求的类

装饰者： 新的类

装饰方法：不满足需求方法

3）需求：

改造BufferedReader的readLine方法

4）装饰者模式步骤：

4.1 编写一个装饰类，继承被装饰类（被装饰类非final）

4.2 在装饰类中声明一个被装饰类的变量

4.3 在装饰类的构造方法中接收被装饰类的实例

4.4 增强被装饰类的方法

代理模式 vs 装饰者模式

代理模式：目的在于得到被代理类的访问权利 (可以选择修改或者不修改被代理类的内容)

装饰者模式：目的在于增强被装饰者的方法 （必须修改被装饰类的内容）

# 监听器

web事件机制

事件源： ServletContext对象，ServletRequest对象，HttpSession对象

事件： 对象创建或销毁事件， 属性的添加修改删除事件 XXXEvent

\*\*\*监听器：XXXListener

监听器分类

|  |  |  |
| --- | --- | --- |
| 事件源(对象) | 事件 | 监听器（接口） |
| ServletContext | 创建和销毁（ServletContextEvent） | ServletContextListener |
| 属性增删改（ServletContextAttributeEvent） | ServletContextAttributeListener |
| ServletRequest | 创建和销毁（ServletRequestEvent） | ServletRequestListener |
| 属性增删改（ServletRequestAttributeEvent） | ServletRequestAttributeListener |
| HttpSession | 创建和销毁（HttpSessionEvent） | HttpSessionListener |
| 属性增删改（HttpSessionBindingEvent） | HttpSessionAttributeListener |

4) ServletContextListener接口

作用：监听ServletContext对象的创建和销毁

创建：应用启动（加载）

销毁：应用卸载或服务器停止

监听器开发步骤：

1）写类，实现对应的监听器接口

//演示ServletContextListener

public class MyContextListener implements ServletContextListener{

//context对象创建时调用

public void contextInitialized(ServletContextEvent sce) {

System.out.println("context对象创建了");

}

//context对象销毁时调用

public void contextDestroyed(ServletContextEvent sce) {

System.out.println("context对象销毁了");

}

}

2）在web.xml文件注册监听器

<!-- 注册监听器 -->

<listener>

<listener-class>gz.itcast.a\_context.MyContextListener</listener-class>

</listener>

案列：统计网站的在线登录用户

# 软件的国际化

国际化： i nternetionalizatio n

简称：i18n

2.1 文字国际化

ResourceBundle类:用于文字国际化

**在jsp上实现文字国际化**

在类（src）的目录下创建一些properties配值文件

message.properties（默认）

message\_en\_GB.properties

message\_en\_US.properties

message\_zh\_CN.properties

等等

声明一个国际化标签

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

在<head>内声明basename

<fmt:setBundle basename="message"/>

<body>内使用

<fmt:message key="username"/>

2.2 日期时间国际化

格式：

中国 2015年10月16日

美国 Oct 10,2015

英国 10 Oct 2015

.....

DateFormat类：用于日期时间国际化

DataFormat format=DataFormat.getDateTimeInstance(

DataFormat.SHORT , DataFormat.SHORT , Locale.CHINA)

.MEDIUMLONG . MEDIUMLONG .UK

.FULL .FULL .FULL

日期 时间 国家代号

String date=format.format(new date())

web页面进行国际化：

文字国家化：

<fmt:setBunble/> 绑定资源包

<fmt:message key="xxxx"/> 获取资源包的值

日期时间国际化

<fmt:formatDate dateStyle="xx" timeStyle="xx" type="both"/>

数字格式化：

<fmt:formatNumber value="${num}" type="xxxx" patten="#.##"/>

type:

number: 数字格式化(默认)

percent: 按照百分比来显示数字

currency: 按照货币格式显示数字

pattern: 使用自定义模式来显示数字

# 文件上传下载

**上传**

文件上传的必要条件

1）必须有表单，表单里file组件 (<input type="file"/>)

2）表单必须是POST方式提交（不能是get提交）:

文件内容必须通过请求正文进行传递

3）表单的正文的内容类型必须为 multipart/form-data

使用apache的commons-fileupoad工具进行文件上传

**Jsp form表单**

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

选择文件<input type=*"file"* name=*"attach"*/><br/>

<input type=*"submit"* value=*"上传"*/>

</form>

**Servlet代码**

//创建DiskFileItemFactory对象

DiskFileItemFactory factory = **new** DiskFileItemFactory(1\*1024\*1024,**new** File("F:\\buffer"));

//获取缓存文件对象

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

//设置文件名的编码格式

fileUpload.setHeaderEncoding("utf-8");

//解析文件

List<FileItem> files = fileUpload.parseRequest(request);

//获取文件信息

FileItem file=files.get(0);

String name=file.getName();

**long** size=file.getSize();

String contentType=file.getContentType();

InputStream input=file.getInputStream();

//保存文件

FileUtils.*copyInputStreamToFile*(input, **new** File("F:\\文件上传"+newFileName));

//删除缓存文件

file.delete();

**下载**

//读文件

File file =**new** File("F:\\文件上传"+uri);

//获取文件字节流

FileInputStream fis=**new** FileInputStream(file);

//文件名中文编码

name = URLEncoder.*encode*(name, "utf-8");

//下载提示

response.setHeader("Content-Disposition","attachment;filename=" + name);

//创建服务器输出流，

ServletOutputStream out=response.getOutputStream();

//写出到浏览器

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

**int** len=0;

**while**((len=fis.read(buffer))!=-1){

out.write(buffer, 0, len);

}

fis.close();

out.close();

# 在线支付

1)互联网的模式

B2B: Business To Business 阿里巴巴 企业与企业

B2C: Business To Customer 京东 天猫 企业与客户

C2C: Customer To Customer 淘宝 客户与客户

O2O: Online To Offline 线上与线下

2)交易支付

在线支付

3)在线支付两种模式

①对接银行

**客户↔电子商务网站↔银行在线支付网站**

对接步骤:

1.向银行申请一个商家账号

2.分配给商家一个账号ID和密钥

3.给商家一些对接文档和API

优点：速度快

缺点：维护成本比较高，网站需要维护和不同银行的支付接口，收费高

②对接中间支付平台（支付宝，财务通，快钱，易宝支付，paypal）

**客户↔电子商务网站↔在线中间支付平台↔银行在线支付网站**

对接步骤

1.向中间支付平台申请商家账号

2.平台发送一个商家ID和密钥

3.平台发送对接文档和API工具

优点：接入成本低，收费比较低

缺点：速度相对慢

# JavaMail

1) JavaMail: 使用java程序来发送邮件

2) 了解收发邮件的原理

发邮件：通过SMTP协议来发送邮件

收邮件： 通过POP3协议来接收邮件

3）SMTP协议

Simple Mail Transfer Potocal

4) POP3协议

收邮件协议

5) JavaMail:

Sun设计的一套专门用于发送邮件的程序。

javamail属于javaee标准。

javax.mail.\* 核心类

javax.activation.\* 附件相关的类

6）开发步骤：

6.1 如果java项目，需要导入jar包 （ mail.jar activation.jar）

6.2 核心类：

Session类： 表示一次和邮件服务器的连接

MimeMessage类： 表示一封邮件

setXXX(...) 给邮件设置数据

主题 收集人 正文...

Transport类： 用于发送邮件

send(...) 发送邮件

案例：激活注册用户

# 时间

//时间格式化

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd hh:mm:ss");

String nowTime=sdf.format(new Date())); //此刻时间

String lastTime sdf.format(new Date(session.getLastAccessedTime()));//最后一次访问的时间

# Struts2

MVC模式的有点：

通过分层开发项目，便于分工合作开发！

使系统解耦，降低层与层之间的耦合（关联）

提示可维护性、可扩展性、健壮性

缺点：

对于简单的系统，开发比较麻烦

一、struts2: 是一个MVC框架，是表现层的框架 webwork+struts1

struts2:默认拦截器有：

i18n编码拦截器

参数拦截器

类型转换拦截器

文件上传拦截器

二、struts2开发步骤：

1）导入strurs2的jar包

8个jar包

2）在web.xml配置核心过滤器

<filter>

<file-name>

<file-class>org.apache.struts......StrutsPrepareAndExecuteFilter

</filter>

<filter-mapping>

<file-name>

<url-pattern>/\*

</filter-mapping>

3)在开发目录下的src目录下创建struts.xml

<package name="包名" namespace="名称空间" extends="struts-default">

<action name="访问Action名" class="Action类全名" method="Action方法">

<result name="Action的返回值">/跳转的页面</result>

</action>

</package>

4）启动tomcat，访问action

访问路径： /web应用名称/名称空间/action的名称

三、struts2执行过程

核心过滤器：StrutsPreparedAndExecuteFilter

1）初始化，执行init方法

加载xml文件

default.properties 默认参数

struts-default.xml 默认配置

struts-plugin.xml 插件配置

struts.xml \*\*\*业务配置\*\*

2）每次访问，执行doFilter方法

使用访问的路径去匹配action的名称

通配符：

<action name="\*\_\*" class="cn.itcast.{1}Action" method="{2}">

<result name="xxxx">/跳转的页面</result>

</action>

1 Struts2基本配置和功能

1.1 Struts2的Action三种编写方式

1.2 Struts2的常量配置

1.3 Struts2的全局视图配置和默认配置

1.4 Action的属性注入

2 Struts2的核心功能

2.1 sruts2的数据共享的三种方式（重点）

2.2请求参数数据的封装（重点）

2.3 自定义类型转换

2.4 文件上传和下载

## 1 struts2的Action三种使用方式

### 1.1 第一种方式，不实现Action接口

|  |
| --- |
| /\*\*  \* 第一种方式：不需要实现或继承任何接口或类  \* **@author** APPle  \*  \*/  **public** **class** UserAction2 {  **public** String login()**throws** Exception{  System.*out*.println("UserAction2.login()");  **return** "success";  }    } |

### 1.2第二种方式，实现Action接口

|  |
| --- |
| /\*\*  \* 第二种方式：实现Action接口  \* 1)定义了默认的execute方法的标准  \* 2）提供了项目中常用的视图标记  \* **@author** APPle  \*/  **public** **class** UserAction **implements** Action {  **public** String login() **throws** Exception {  System.*out*.println("执行了UserAction的login方法");  **return** *SUCCESS*;  }  **public** String execute() **throws** Exception {  **return** **null**;  }  } |

### 1.3 第三种方式， 继承ActionSupport类（推荐）

|  |
| --- |
| /\*\*  \* 第三种方式： 继承ActionSupport类(推荐使用)  \* 好处：  \* 1）提供了常用的视图标记  \* 2）提供了数据校验功能  \*/  **public** **class** UserAction3 **extends** ActionSupport{  **public** String login()**throws** Exception{  System.*out*.println("UserAction3.login()");  **return** *SUCCESS*;  }  } |

## 2 struts2的常量配置

struts2的常量就是用于在strut2的程序运行过程中使用的一些常量参数。

|  |
| --- |
| 指定默认编码集,作用于HttpServletRequest的setCharacterEncoding方法 和freemarker 、velocity的输出  <constant name="str uts.i18n.encoding" value="UTF-8"/>  自定义后缀修改常量  <constant name="struts.action.extension" value=" action,do,,"/>  设置浏览器是否缓存静态内容,默认值为true(生产环境下使用),开发阶段最好关闭  <constant name="struts.serve.static.browserCache" value="false"/>  当struts的配置文件修改后,系统是否自动重新加载该文件,默认值为false(生产环境下使用),开发阶段最好打开  <constant name="struts.configuration.xml.reload" value="true"/>  开发模式下使用,这样可以打印出更详细的错误信息  <constant name="struts.devMode" value="true" />  默认的视图主题  <constant name="struts.ui.theme" value="simple" />  与spring集成时，指定由spring负责action对象的创建  <constant name="struts.objectFactory" value="spring" />  该属性设置Struts 2是否支持动态方法调用，该属性的默认值是true。如果需要关闭动态方法调用，则可设置该属性（action名！方法名）  为 false  <constant name="struts.enable.DynamicMethodInvocation" value="false"/>  上传文件的大小限制  <constant name="struts.multipart.maxSize" value=“10701096"/>  文件上传时的临时目录  <constant name="struts.multipart.saveDir" value=“d://temp"/> |

注意：

通过struts.xml文件声明<constant name="struts.action.extension" value="action,do,,">

</constant>修改常量配置。

开发过程中通常先导入常量配置文件，在导入业务配置文件

|  |
| --- |
| <!-- 导入常量配置文件 -->  <include file="constant.xml"></include>  <!-- 导入业务配置文件 -->  <include file="gz/itcast/a\_action/struts-action.xml"></include>  <include file="gz/itcast/b\_constant/struts-constant.xml"></include> |

## 3 struts2的全局视图配置和默认配置

4.1 全局视图作用： 当该包下的所有action都使用到的一些视图就是可以放到全局视图配置中

注意：

当action中也有相同名称的视图，那么action的局部视图会覆盖全局视图。

|  |
| --- |
| <!-- 全局视图配置: 把该包下的所有action共用的视图都机集中在这里写 -->  <global-results>  <result name=*"success"*>/login.jsp</result>  </global-results> |

4.2 action的默认配置

|  |
| --- |
| <!-- 默认配置  Action：  name: 必填项  class: 可选项 。默认配置： ActionSupport类 该类继承自struts-default （<default-class-ref class="com.opensymphony.xwork2.ActionSupport" />）  method: 可选。默认配置execute  result:  name: 可选。默认配置： success  type: 可选。默认配置： dispatcher转发 redirect 重定向 chain转发到action  redirectAction重定向到action  -->  <!-- 全部使用默认配置的action的作用 ：专门用于转发到WEB-INF下的页面 -->  <action name=*"book"*>  <result>/WEB-INF/jsp/login.jsp</result>  </action> |

## 4 Action的属性注入

作用： 如果Action对象中需要把一些经常改变的参数提取到配置文件中，那么就可以使用属性注入的方法。

Action属性注入的步骤

1）在Action类中声明一个成员变量，用于接收xml配置文件传入内容

2）在Action类提供一个该变量的setter方法，该方法接收了xml配置的内容

|  |
| --- |
| //1）在action中提供一个属性  获取地址  **private** String savePath;  //2)提供属性的setter方法，用于外部的action的参数进行注入  **public** **void** setSavePath(String savePath) {  **this**.savePath = savePath;  }  传递值  映射 |

3）在对应的struts.xml文件中，找到对应的action对象的配置，然后在action中使用

<param name=""></param> 这个标签来向Action对象的属性注入内容

|  |
| --- |
| <action name=*"upload"* class=*"gz.itcast.d\_ioc.UploadAction"* method=*"upload"*>  <!-- 3)使用该配置可以往Action对象的属性注入内容(只要有setter都可以使用param进行注入)  param:  name: setter方法名。setSavePath -> savePath  -->  <param name=*"savePath"*>e:/images/</param>  <result>/login.jsp</result>  </action> |

## 5 sruts2的数据共享的三种方式

在web项目中都是使用域对象来共享数据。

struts2提供给开发者使用域对象来共享数据的方法一共有三种。

### 5.1 第一种方式

ServletActionContext类

getRequest() : 获取request对象

getRequest().getSession() :　获取session对象

getServletContext() : 获取ServletContext对象

注意：

1）该方式依赖servlet的api，耦合比较高

2）如果要通过域对象来获取域对象的相关信息必须使用该方式

### 5.2 第二种方式

ActionContext类

getContextMap() ：　获取操作request域对象数据的map集合

getSession() : 获取操作session域对象数据的map集合

getApplication() 获取操作context域对象数据的map集合

注意：

1）不依赖servlet的api，耦合性低

2）只能用在Action对象的一个方法中。不能在所有方法中都是用同一个ActionContext

### 5.3 第三种方式

使用 RequestAware , SessionAware ApplicationAware 接口

注入操作对应域对象数据的Map集合

①实现requestAware接口

提供set方法

public void setRequest(Map<String,Object> request){

this.requestMap=request;

}

②成员变量private Map<String,Object> requestMap;

③成员方法就可以使用requestMap.put(“login\_info”,”user”);

注意：

1）不依赖servlet的api

2）可以在Action对象的所有方法中共享Map集合

## 6 请求参数数据的封装

### 6.1 直接赋值给简单数据类型

|  |
| --- |
| **public** **class** UserAction **extends** ActionSupport{  //参数赋值（注入方式）  **private** String name;  **private** String password;  **private** String gender;  **private** String[] hobit;  //参数通过这个set方法注入到Action中  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** **void** setGender(String gender) {  **this**.gender = gender;  }  **public** **void** setHobit(String[] hobit) {  **this**.hobit = hobit;  } |

### 6.2 赋值给一个javabean对象

|  |
| --- |
| <form action=*"*${pageContext.request.contextPath }*/data/user\_register.action"* method=*"post"*>  用户名： <input type=*"text"* name=*"user.name"*/><br/>  密码： <input type=*"password"* name=*"user.password"*/><br/>  性别： <input type=*"radio"* name=*"user.gender"* value=*"男"*/>男  <input type=*"radio"* name=*"user.gender"* value=*"女"*/>女<br/>  爱好：  <input type=*"checkbox"* name=*"user.hobit"* value=*"篮球"*/>篮球  <input type=*"checkbox"* name=*"user.hobit"* value=*"足球"*/>足球  <input type=*"checkbox"* name=*"user.hobit"* value=*"羽毛球"*/>羽毛球<br/>  <input type=*"submit"* value=*"注册"*/>    </form> |

|  |
| --- |
| **public** **class** UserAction2 **extends** ActionSupport{  //使用一个javabean对象接收  **private** User user;  **public** User getUser() {  **return** user;  }  **public** **void** setUser(User user) {  **this**.user = user;  } |

注意：请求参数的封装通过struts2的ParametersInterceptor拦截器进行赋值.

### 6.3 使用模型驱动方式

|  |
| --- |
| **public** **class** UserAction **extends** ActionSupport **implements** ModelDriven<User>{  //注意：使用模型驱动的方式，存放数据的模型引用不能为空  **private** User user = **new** User();  **public** User getModel() {  **return** user;  }  } |

## 7 自定义类型转换

struts2框架可以在封装数据之前自动把表单的String类型数据类型转换成int、double、float、long、date

作用： 默认情况下，页面的日期类型只能接收 yyyy-MM-dd类型，如果要转换yyyy/MM/dd这种类型，则需要使用自定义类型转换器进行转换。

strut2提供了自定义类型转换器的基类： StrutsTypeConverter类

|  |
| --- |
| /\*\*  \* 自定义日期类型转换器  \* **@author** APPle  \*  \*/  **public** **class** MyDateConverter **extends** StrutsTypeConverter{    SimpleDateFormat sdf1 = **new** SimpleDateFormat("yyyy/MM/dd");    /\*\*  \* 从页面的数据到服务器的类型转换  \* 参数一： context： 值栈上下文对象  \* 参数二： values: 从页面传递过来的参数值  \* 参数三： toClass: 转换到的类型。String->java.util.Date  \*/  @Override  **public** Object convertFromString(Map context, String[] values, Class toClass) {  System.*out*.println("进入了转换器");  **try** {  //1)判断values是否为空  **if**(values==**null** || values.length==0){  **return** **null**;  }    **if**(values.length>1){  //多个值的情况  Date[] dates = **new** Date[values.length];  //2)取出数据进行转换  **for**(**int** i=0;i<dates.length;i++){  Date date = sdf1.parse(values[i]);  dates[i] = date;  }  **return** dates;  }**else**{  //一个值的情况  Date date = sdf1.parse(values[0]);  **return** date;  }  } **catch** (ParseException e) {  e.printStackTrace();  }  **return** **null**;  }  /\*\*  \* 从服务器到页面的类型转换  \*/  @Override  **public** String convertToString(Map context, Object o) {  **return** **null**;  }  } |

支持多种格式

|  |
| --- |
| //演示日期类型转换器  public class MyDateConverter extends StrutsTypeConverter {  SimpleDateFormat[] sdf = new SimpleDateFormat[] {  new SimpleDateFormat("yyyy/MM/dd"),  new SimpleDateFormat("yyyy.MM.dd"),  new SimpleDateFormat("yyyy\_MM\_dd"),  new SimpleDateFormat("yyyy-MM-dd"), };  // 前台->后台格式转换  /\*\*  \* 参数一：context： 表示ValueStack对象 参数二：values: 表单需要转换的数据 参数三：toClass:需要转换到的类型，  \* String-》java.util.Date  \*/  @Override  public Object convertFromString(Map context, String[] values, Class toClass) {  // 取出表单的数据  if (values == null) {  throw new IllegalArgumentException("参数错误");  }  if (values.length == 1) { // 1个值的情况  String birth = values[0]; // 1995/10/01  for (int i = 0; i < sdf.length; i++) {  try {  return sdf[i].parse(birth);  } catch (ParseException e) {  // e.printStackTrace();  continue;  }  }  } else {  Date[] target = new Date[values.length];  // 多个值的情况  for (int i = 0; i < values.length; i++) {  try {  target[i] = sdf[1].parse(values[i]);  } catch (ParseException e) {  //e.printStackTrace();  continue;  }  }  return target;  }  return null;  }  // 后台->前台  @Override  public String convertToString(Map context, Object o) {  return null;  }  } |

**绑定自定义转换器的方式：**

方式一： 局部绑定(只能绑定一个Action)

1）建立一个 【Action类名-conversion.properties】

2）一定和绑定的Action放在同一个目录下。

|  |
| --- |
| user.birth=gz.itcast.g\_convert.MyDateConverter |

方式二： 全局绑定（绑定整个项目多个Action）（推荐使用）

1）建立一个xwork-conversion.properties

2）该文件一定放在src目录下。

|  |
| --- |
| java.util.Date=gz.itcast.g\_convert.MyDateConverter |

## 8文件上传和下载

### 8.1 文件上传

1）三个条件：

表单有<input type=”file” name=”attach”/>

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

2)在Action中接收文件内容

File attach; (attach是file表单的name属性)

String attachContentType; 文件类型

String attachFileName; 文件名称

名字要一致

|  |
| --- |
| **public** **class** UploadAction **extends** ActionSupport {  **private** File attach; //表单的name属性  **private** String attachContentType; //文件类型  **private** String attachFileName; //文件名    **public** String saveImage(){  **try** {  //文件保存路径：服务器应用文件夹内的upload/user/  String filePath = ServletActionContext.*getServletContext*().getRealPath("upload/user/");  //文件重命名uuid+后缀  String fileName = UUID.*randomUUID*().toString() + attachFileName.substring(attachFileName.lastIndexOf("."));  //保存文件到服务器  FileUtils.*copyFile*(attach, **new** File(filePath, fileName));  **return** "success";  } **catch** (IOException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  ……get/set方法} |

细节：

修改上传大小

|  |
| --- |
| <!-- 修改默认文件上传大小 -->  <constant name=*"struts.multipart.maxSize"* value=*"100000000"*></constant>  文件上传时的临时目录  <constant name="struts.multipart.saveDir" value=“d:/temp"></constant> |

修改允许上传的文件类型和文件后缀

|  |
| --- |
| <action name=*"upload"* class=*"gz.itcast.h\_upload\_down.UploadAction"* >  <!-- 往FileUploadInterceptor拦截器的属性注入值（调用setter方法） -->    <interceptor-ref name=*"defaultStack"*>  <!-- 改变当前文件上传拦截器的允许文件类型 -->  <param name=*"fileUpload.allowedTypes"*>image/jpeg,image/jpg</param>  <!-- 允许的文件后缀 -->  <param name=*"fileUpload.allowedExtensions"*>jpg,jpeg,gif</param>  <!-- 如果以上配置都写了,那么取他们的交集 -->  </interceptor-ref>    <param name=*"savePath"*>e:/images/</param>  <result>/login.jsp</result>  <result name=*"input"*>/error.jsp</result>  </action> |

### 8.2 文件下载

视图类型一定是stream类型

|  |
| --- |
| <action name=*"down\_\*"* class=*"gz.itcast.h\_upload\_down.DownAction"* method=*"{1}"*>  <param name=*"serverPath"*>e:/images/</param>  <result name=*"list"*>/listFile.jsp</result>  <!-- 文件下载的关键: 视图类型一定是stream -->  <result name=*"down"* type=*"stream"*>  <!-- 返回给浏览器的文件类型。返回通用的二进制 -->  <param name=*"contentType"*>application/octet-stream</param>  <!-- 返回给浏览器的输入流（文件流） -->  <param name=*"inputName"*>inputStream</param>  <!--下载提示-->  <param name=*"contentDisposition"*>attachment;filename=${name}</param>  <!-- 缓存大小 -->  <param name=*"bufferSize"*>1024</param>  </result>  </action> |

在Action对象中提供一个对应的获取输入流的方法

|  |
| --- |
| //需要提供给struts写出数据的输入流  **public** InputStream getInputStream(){  **try** {  FileInputStream fis = **new** FileInputStream(**new** File(serverPath+name));  **return** fis;  } **catch** (FileNotFoundException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  **public** String getName() {  **return** name;  } |

## Ognl语法（Ognl表达式）

1）概述

OGNL是Object Graphic Navigation Language（对象图导航语言）的缩写，它是一个开源项目。 Struts2框架使用OGNL作为默认的表达式语言。

2)Ognl作用：

2.1 读取属性

2.2 赋值（操作集合）

2.3 调用方法（包括静态方法）

3）Ognl核心类

OgnlContext类：是一个Map结构

属性：

root: 根元素。 Object

values： 非根元素。 Map

常用方法：

setRoot(..): 赋值给根元素

getRoot() : 获取根元素

put("name",Object) : 赋值给非根元素

get("name"): 获取非根元素

4）Ognl表达式

不带#号：取出OgnlContext的根元素数据

带#号： 取出OgnlContext的非根元素数据

二、分析Struts2的值栈结构

1）ActionContext和OgnlContext关系

ActionContext对象封装了OgnlContext对象

new ActionContext(new OgnlContext())

三、使用struts2标签读取数据

ognl表达式：

读取根元素： 不带#号，直接读取即可

读取非根元素： 带#号，然后再读取

<s:property value="ognl表达式"/> : 使用Ognl表达式读取数据

四、Struts2常用的标签

1）JSTL vs Struts2标签

JSTL： 只支持EL表达式读取数据

Sruts2标签：支持Ognl表达式读取

2）逻辑标签

<s:set></s:set> 赋值

<s:property/> 取值

<s:if test=""></s:if> 条件判断

<s:elseif test=""></s:elseif>

<s:else></s:else>

<s:iterator></s:iterator> 循环

<s:url></s:url> 生成URL

<s:hidden name="complain.compName" value="%{#session.SYS\_USER.name}"></s:hidden>

<s:iterator value="complain.complainreplies" var="reply" status="st">

<s:property value="#st.count"/>

</s:iterator>

## 1 国际化

### 1.1 简介

文字国家化

以前： ResourceBundle类

getBundle(); 绑定指定国家的资源包。

资源包规则：

包名\_语言简写\_国家简称.properties

e.g 中国： message\_zh\_CN.properties

美国： message\_en\_US.properties

英国： message\_en\_GB.properties

jsp页面中：

<fmt:setBundle name="message"/> 绑定资源

<fmt:message key="user"/> 获取资源包的内容

### 1.2 struts2中简化国际化的使用

1）准备不同国家的资源包

中国： message\_zh\_CN.properties

user=用户名

美国： message\_en\_US.properties

英国： message\_en\_GB.properties

2）在struts.xml文件中（修改）指定资源包加载路径的常量

struts.custom.i18n.resources=message

3）在页面上使用资源包的内容

<s:text name="user"/>

## 2数据验证

struts2提供给开发者进行表单数据的后台数据验证的功能。

### 2.1 用代码方式对action的所有方法进行验证

分析：

前台： 注册表单 user\_register.action

user.username

user.password

后台：

class UserAction extends ActionSupport{

private User user;

用于存储错误信息

private Map actionErrors;

public String validate(){

表单校验的逻辑（不好，影响当前业务逻辑）

if(user.getName()==null || user.getName().eqauls("")){

出现异常

保存错误信息

actionErrors.put("name","用户名不能为空");

转发到错误显示显示jsp

}

。。。。。。。

actionErrors.put("password","密码不能为空");

}

//注册方法

public String register(){

//validate(); 这里不写这个方法调用，而把方法的调用交给拦截器去做！！！

具体的注册业务逻辑

}

}

struts2的数据验证的核心拦截器：

|  |
| --- |
| <interceptor name="validation" class="com.opensymphony.xwork2.validator.ValidationInterceptor"/> |

开发步骤：

1）Action类继承ActionSupport（为了实现Valiateable接口）

2）Action类覆盖validate方法(验证所有方法)

|  |
| --- |
| **public** **void** validate() {  //在这里写表单数据验证的逻辑  //System.out.println("调用了validate方法");  **if**(user.getName()==**null** || user.getName().equals("")){  //用户名为空  //把错误信息放入错误信息Map集合  **super**.addFieldError("user.error.requried", "用户名不能为空！");  }  **if**(user.getPassword()==**null** || user.getPassword().equals("")){  //密码为空  //把错误信息放入错误信息Map集合  **super**.addFieldError("password.error.requried", "密码不能为空！");  }    } |

3）在struts.xml文件中对应的action配置加上input视图，然后struts2就会自动把错误信息转发到input视图的页面上去

4）在input视图页面上，打印出错误信息

|  |
| --- |
| <s:fielderror></s:fielderror> |

注意：

validate()方法对当前Action下的所有方法都会有效！！！！

### 2.2 用代码方式对action的指定方法进行验证

指定方法验证：

第二步骤的验证方法名称为 validate+需要验证的方法名称

例如： validateRegister() -> 可以验证register()方法

### 2.3 用xml配置方式对action的所有方法进行验证

以上这两种方式都是使用代码在程序写死了！！！

希望更加灵活，扩展型更好！

xml配置数据验证的规则：

1）编写一个xml文件，名称： Action文件名-validation.xml

2）该xml文件必须放在Action文件的同一目录

注意： 这种配置方式会对action下的所有方法都生效！！！

### 2.4 用xml配置方式对action的指定方法进行验证

xml配置数据验证的规则：

1）编写一个xml文件，名称： Action文件名-**访问方法路径**-validation.xml

例如： UserAction的register方法： user\_register路径访问

文件名： UserAction-user\_register-validation.xml

2）该xml文件必须放在Action文件的同一目录

|  |
| --- |
| <validators>  <!-- 验证用户名 -->  <field name=*"user.name"*>  <!-- type: 代表xwork中定义的可以使用的验证类型 -->  <field-validator type=*"requiredstring"*>  <!-- 当出现错误时，提示的错误信息 -->  <message>用户名必填</message>  </field-validator>  <field-validator type=*"stringlength"*>  <param name=*"minLength"*>6</param>  <param name=*"maxLength"*>16</param>  <message>用户名必须是6-16位</message>  </field-validator>  </field>    </field>    </validators> |

|  |
| --- |
| <validators>  <validator name=*"required"* class=*"com.opensymphony.xwork2.validator.validators.RequiredFieldValidator"*/>  <validator name=*"requiredstring"* class=*"com.opensymphony.xwork2.validator.validators.RequiredStringValidator"*/>  <validator name=*"int"* class=*"com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator"*/>  <validator name=*"long"* class=*"com.opensymphony.xwork2.validator.validators.LongRangeFieldValidator"*/>  <validator name=*"short"* class=*"com.opensymphony.xwork2.validator.validators.ShortRangeFieldValidator"*/>  <validator name=*"double"* class=*"com.opensymphony.xwork2.validator.validators.DoubleRangeFieldValidator"*/>  <validator name=*"stringlength"* class=*"com.opensymphony.xwork2.validator.validators.StringLengthFieldValidator"*/>  <validator name=*"regex"* class=*"com.opensymphony.xwork2.validator.validators.RegexFieldValidator"*/>  <validator name=*"conditionalvisitor"* class=*"com.opensymphony.xwork2.validator.validators.ConditionalVisitorFieldValidator"*/>  </validators> |

## 3 拦截器

### 3.1 拦截器简介

拦截器 类似于 过滤器的功能，过滤器可以过滤项目的任何请求（servlet/jsp/html/img），但拦截器只能拦截Action资源。拦截完Action，添加逻辑代码

### 3.2 拦截器的编写规则

struts2提供Interceptor接口用于自定义拦截器

步骤：

1）编写类，实现Interceptor接口

|  |
| --- |
| **public** **class** MyInterceptor1 **implements** Interceptor{    **public** MyInterceptor1(){  System.*out*.println("1)创建了拦截器1对象");  }  **public** **void** destroy() {    }  **public** **void** init() {  System.*out*.println("2)调用了拦截器1的init方法");  }  **public** String intercept(ActionInvocation invocation) **throws** Exception {  System.*out*.println("4)拦截了Action---前面的代码");    //放行，调用下一个拦截器，如果没有下一个拦截器，那么调用目录的action  invocation.invoke();    System.*out*.println("6)拦截了Action---后面的代码");    **return** **null**;  }  } |

2）在struts.xml文件中定义和使用拦截器

|  |
| --- |
| <package name=*"inter"* extends=*"struts-default"* namespace=*"/inter"*>  <!-- 定义拦截器 -->  <interceptors>  <interceptor name=*"inter1"* class=*"gz.itcast.a\_interceptor.MyInterceptor1"*></interceptor>  </interceptors>        <action name=*"user\_\*"* class=*"gz.itcast.a\_interceptor.UserAction"* method=*"{1}"*>  <!-- 使用拦截器 -->  <interceptor-ref name=*"inter1"*></interceptor-ref>  <result>/index.jsp</result>  </action>    </package> |

注意1：拦截器的执行顺序：

启动：

1）拦截器对象创建，调用拦截器的init方法

访问

2）创建Action对象

3）执行拦截器的intercepot方法

3.1 执行拦截器前面的代码（invocation.invoke();方法之前的）

3.2 执行 invocation.invoke(); 放行执行下一个拦截器或Action的方法

3.3 执行拦截器后面的代码（invocation.invoke();方法之后的）

注意2：拦截器范围：

局部使用：action其作用

|  |
| --- |
| <action name=*"user\_\*"* class=*"gz.itcast.a\_interceptor.UserAction"* method=*"{1}"*>  <!-- 局部起作用的使用拦截器 -->  <interceptor-ref name="inter1"></interceptor-ref>  <result>/index.jsp</result>  </action> |

全局使用： 当前包其作用：

|  |
| --- |
| <package name=*"inter"* extends=*"struts-default"* namespace=*"/inter"*>  <!-- 定义拦截器 -->  <interceptors>  <interceptor name=*"inter1"* class=*"gz.itcast.a\_interceptor.MyInterceptor1"*></interceptor>  </interceptors>    <!-- 全局其作用的拦截器 -->  <default-interceptor-ref name=*"inter1"*></default-interceptor-ref> |

### 3.3 拦截器栈

概念： 当一个或多个Action同时被多个拦截器所拦截，就可以使用拦截器。

|  |
| --- |
| <interceptors>  <interceptor name=*"inter1"* class=*"gz.itcast.a\_interceptor.MyInterceptor1"*></interceptor>  <interceptor name=*"inter2"* class=*"gz.itcast.a\_interceptor.MyInterceptor2"*></interceptor>  <!-- 定义栏截器栈 -->  <interceptor-stack name=*"interStack"*>  <interceptor-ref name=*"inter1"*></interceptor-ref>  <interceptor-ref name=*"inter2"*></interceptor-ref>  </interceptor-stack>  </interceptors> |

注意1：定义拦截器栈的时候，引用拦截器的顺序决定了创建拦截器对象的顺序。（先指定就先创建）

注意2： 当前有了拦截器栈（多个拦截器）的执行顺序：

启动：

1）创建拦截器*inter1*对象，调用init方法

2）创建拦截器*inter2*对象，调用init方法

访问：

3）执行inter1的interceptor方法前面代码

4）执行inter2的interceptor方法前面代码

5）Action的业务方法

6）执行inter2的interceptor方法后面代码

7）执行inter1的interceptor方法前面代码

注意3： 当我们的包下引用了自定以拦截器，则会把默认包下的default-stack拦截器给覆盖掉！！！

这时需要这么做：

|  |
| --- |
| <interceptor-stack name=*"myStack"*>  <interceptor-ref name=*"defaultStack"*></interceptor-ref><!-- 引入了默认的18个拦截器 -->  <interceptor-ref name=*"interStack"*></interceptor-ref><!-- 引入自定义的2个拦截器 -->  </interceptor-stack> |

struts2的核心流程图（关键）：



# Hibernate

## 1. ORM规则

ORM,

Object 对象 (普通的javabean对象)

Relation 关系 (关系型数据库)

Mapping 映射 (对象 < --- -- > 关系型数据库)

思考：

1. 对象与数据库，有没有关系？

2. 能否不写sql语句，

把对象直接保存到数据库？

从数据库中取出的就是对象？

3. 如果换了数据库，能否不更改代码？(dao代码不需要改！)

面试题：

ORM 与 hibenrate关系?

ORM

是一个概念，表示“对象关系映射”

Hibernate

是基于ORM概念的实现！

## 2. hello world！

版本：hibernate3.6

步骤：

1. 建库/建表

|  |
| --- |
| -- 建库  CREATE DATABASE day31 CHARACTER SET utf8;  USE day31;  CREATE TABLE t\_user(  uid INT PRIMARY KEY AUTO\_INCREMENT,  uname VARCHAR(50),  salary DOUBLE,  birth DATE  ) |

2. 搭建hibernate开发环境

|  |
| --- |
| 1.引入hibernate 相关jar文件 |
| antlr-2.7.6.jar  c3p0-0.9.1.jar  commons-collections-3.1.jar  dom4j-1.6.1.jar  hibernate-jpa-2.0-api-1.0.0.Final.jar  hibernate3.jar  javassist-3.12.0.GA.jar  jta-1.1.jar  mysql-connector-java-5.1.7-bin.jar  slf4j-api-1.6.1.jar |

|  |
| --- |
| 2. User.java 写实体类并且写映射配置 |
| 创建一个Student的对象（代码忽略）  <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"cn.itcast.entity"*>  <class name=*"Student"* table=*"student"*>  <!-- id就表示主键映射 -->  <id name=*"id"* column=*"id"*>  <generator class=*"native"*></generator>  </id>  <!-- 非主键映射property -->  <!-- 注意：如果列名称是数据库中关键字，用反引号！ -->  <property name=*"name"* column=*"`name`"*></property>  <property name=*"chinese"* column=*"chinese"*></property>  <property name=*"math"* column=*"math"*></property>  <property name=*"english"* column=*"english"*></property>  <property name=*"gender"* column=*"gender"*></property>  <property name=*"address"* column=*"address"*></property>  </class>  </hibernate-mapping> |

|  |
| --- |
| 3. hibernate.cfg.xml 主配置文件 |
| <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <!-- 配置： 数据库连接URL、驱动、用户名、密码、方言(告诉hibernate我使用了哪种数据库！) -->  <property name=*"hibernate.connection.url"*>jdbc:mysql:///day15</property>  <property name=*"hibernate.connection.driver\_class"*>com.mysql.jdbc.Driver</property>  <property name=*"hibernate.connection.username"*>root</property>  <property name=*"hibernate.connection.password"*>root</property>  <property name=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</property>  <!-- 显示hibernate生成的sql语句 -->  <property name=*"hibernate.show\_sql"*>true</property>  <!-- 加载所有的映射文件-->  <mapping resource=*"cn/itcast/entity/Student.hbm.xml"*/>  </session-factory>  </hibernate-configuration> |

|  |
| --- |
| 4.Api |
| //创建一个对象  Student student=**new** Student();  student.setName("Yang");  student.setChinese(78);  student.setMath(88);  student.setEnglish(99);  student.setGender("female");  student.setAddress("American");  //1.创建配置管理器对象  Configuration cfg = **new** Configuration();  //2.加载主配置文件hibernate.cfg.xml  cfg.configure("hibernate.cfg.xml");  //创建SessionFactory对象  SessionFactory sf=cfg.buildSessionFactory();  //创建session对象  Session session = sf.openSession();  //开启实物  Transaction transaction = session.beginTransaction();  //保存  session.save(student);  //提交事物关闭session  transaction.commit();  session.close(); |

|  |
| --- |
| **\* 引入hibernate 相关jar文件** |
| hibernate-distribution-3.6.0.Final  hibernate3.jar + lib\required目录 + lib\jpa目录 + MySQL数据库驱动包  用户库，管理jar文件！  新建用户库方式：  ……. |
| **\* User.java 写实体类** |
| 注意：  // 实体类， 封装数据  // 也叫， POJO persistent object java object 持久化的javabean对象！  // POJO 要求： 必须有“对象唯一标识” |
| **User.hbm.xml写映射文件**  **作用：**  **1. 建立“对象”与“数据库表”的映射关系**  **2. 完全描述一张表！** |
|  |
| **hibernate.cfg.xml 主配置文件**  参考源码：hibernate-distribution-3.6.0.Final\project\etc\hibernate.properties 常用配置  ## MySQL  #hibernate.dialect org.hibernate.dialect.MySQLDialect 【方言】  #hibernate.dialect org.hibernate.dialect.MySQLInnoDBDialect  #hibernate.dialect org.hibernate.dialect.MySQLMyISAMDialect  #hibernate.connection.driver\_class com.mysql.jdbc.Driver  #hibernate.connection.url jdbc:mysql:///test  #hibernate.connection.username gavin  #hibernate.connection.password  注意：  配置时候，hibernate前缀可以写，也可以省略！ |
|  |
| 制作一个HibernateUtil |
| **public** **class** HibernateUtil {    **public** **static** Session getSession(){  //1.创建配置管理器对象  Configuration cfg = **new** Configuration();  //2.加载主配置文件hibernate.cfg.xml  cfg.configure("hibernate.cfg.xml");  //创建SessionFactory对象  SessionFactory sf=cfg.buildSessionFactory();  //创建session对象  Session session = sf.openSession();    **return** session;    }  **public** **static** Transaction getTransaction(Session session){  //开启实物  **return** session.beginTransaction();    }    **public** **static** **void** closeSession(Session session,Transaction transaction){  transaction.commit();  session.close();  }  } |

|  |
| --- |
| 在dao上制作静态代码块 |
| private static SessionFactory sf;  static {  sf = new Configuration()//  .configure()//  .addClass(User.class)//  .buildSessionFactory();  } |

## 3. Api

|-- Configuration 配置管理器对象, 主要管理配置文件

Configuration configure() 加载主配置文件

举例1: config.configure("hibernate.cfg.xml");

举例1: config.configure(); 默认加载src/hibernate.cfg.xml

Configuration buildSessionFactory(); 创建session的工厂

Configuration addClass(clazz);

|-- SessionFactory session的工厂，创建session对象！

一个应用程序、一个数据库，对应一个sessionFactory对象！单例！

Session openSession(); 每次打开一个新的session！

Session getCurrentSession(); 获取绑定到当前线程上的session！

**-session增删改**----------------------------------------------------------------------------------------------------------

save 保存 save(user)

get 获取get(User.class, 2)

update 修改的对象必须要设置主键，且主键必须要再数据库中存在!否则报错！

防止报错：// 先查询，

Object obj = session.get(User.class, 2);

// 再删除

if (obj != null&&user.getId!=null) {

session.delete(obj);

}

结果：表内数据只会更新对象中属性有值的字段，没有值的属性会把字段变为默认值

**|-- Session**

**主键查询：**

**Object session.get(clazz,id); 及时加载；**

**Id 在数据库不存在，返回null**

**Object session.load(clazz,id); 懒加载：使用数据时候才加载**

**Id 在数据库不存在，只要使用就报错！（除了用主键外！）**

**HQL查询：**

**HQL hibenate query language hibernate提供的面向对象的查询语言!**

**方式1： Query 接口 (最常用)**

//分页查询

Query q = session.createQuery("from Student");

q.setFirstResult(0);//从第0个起

q.setMaxResults(3); //每页显示3个

List<Student> list = q.list();

System.*out*.println(list);

**方式2： Criteria 接口**

//条件id=3查询

Criteria cri = session.createCriteria(Student.**class**);

cri.add(Restrictions.*eq*("id", 3));

List<Student> list = cri.list();

System.*out*.println(list);

**方式2： SQLQuery 接口**

**本地sql查询, 支持原始的sql语句！**

**不能跨数据库平台，如果改了数据库，程序需要改，因为sql语句可能变化！**

**当hql满足不了现有的复杂的需求的时候，可以试着使用这种查询！**

//查找数学成绩大于80的

String sql="select \* from student where math>80";

SQLQuery result = session.createSQLQuery(sql).addEntity(Student.**class**);

System.*out*.println(result.list());

**HQL与SQL区别？**

**SQL： 查询的一定是表、表的字段！不区分大小写！**

**HQL: 通常查询的是对象、对象的属性！ 区分大小写！**

**更新方法：**

**Void save(obj); 保存一个对象**

**Void update(obj); 更新对象：修改的对象必须要设置主键，**

**且主键必须要再数据库中存在!否则报错！**

**Void saveOrUpdate(obj) 保存或更新**

**1. 如果有设置主键，**

**主键存在， 更新**

**主键不存在，报错！**

**2. 没有设置主键**

**新增保存！**

**Void delete(obj) 删除一个对象**

|-- Transaction

|-- Query

…

## 4. hibernate crud 案例

## 5. hibernate.cfg.xml

作用：

1. 数据库连接参数配置

数据库连接字符串、驱动、连接账号、密码、方言!

2. 其他常用配置

三个。

3. 加载映射文件

1. 方言作用：

问题： hibernate如何实现跨数据库平台的？

答案： 指定“方言”

用户使用哪种数据库，需要通过方言告诉hibernate！

Hibernate会根据方言配置，生成符合当前方言对应的数据库语法的sql语句！

2. 自动建表

## auto schema export

#hibernate.hbm2ddl.auto create-drop 在创建sessionFactory的时候，建表！

在执行sessionFactory.close()方法时候删除表

#hibernate.hbm2ddl.auto create 始终创建一个新的表！

#hibernate.hbm2ddl.auto update 表不存在，就创建表； 表已经存在，则不创建！

#hibernate.hbm2ddl.auto validate 验证； 一般是在生产环境用！

启动时候，会检查映射与数据库是否一致，不一致就报错！

增多验证功能！

代码方式自动建表，

|  |
| --- |
| @Test  **public** **void** testname() **throws** Exception {  // 加载主配置文件  Configuration cfg = **new** Configuration().configure();  // 工具类自动建表  SchemaExport export = **new** SchemaExport(cfg);  // 参数1： 是否在控制台显示建表sql语句！  // 参数2： 是否执行sql  export.create(**true**, **true**);  } |

## 6. User.hbm.xml 映射配置

\* 非主键字段映射

\* 主键映射、、

\* 联合主键映射

User.hbm. xml

作用：

1. 把对象映射到数据库某个表中！

2. 完全描述一张表！

\* 联合主键映射

问题1： 什么是联合主键？

多列联合在一起作为一个主键！

问题2： 一个表是不是必须有主键？ 能否有多个主键？

不是，但大多数表都会有主键！

一个表，最多只能有一个主键！

需求：

用户名称、地址、年龄、邮编.

（能确保：用户名+ 地址唯一）

实现

\* javabean

\* 映射

\* 测试

class 一个class表示映射的一个对象！

name 指定映射的对象的类全名！

table 指定类，对应的表; 可以省略，默认与类名称一样！

property 映射一个普通字段

name 对象的属性

column 对象的属性，对应的数据库表的字段

type 指定类型

方式1: 指定为java类型，一定要是类的全名

如：java.lang.String

方式2： 指定为hibernate的类型, 注意都是小写开头

length 指定长度!

not-null 非空

unique 唯一

id 表示主键映射， hibernate操作的表，一定要有id，一定要有主键！

主键生成策略：

<generator class="native"></generator>

**class**

identity 自增长， mysql/sqlserver等支持！

sequence oracle中自增长的方式！

native 自增长！ (自动选择 identity、sequence 其中一种！) 更通用！

assigned 手动指定主键的值 (控制主键唯一！ （主键=业务编码+时间+随机数）)

uuid uuid值作为主键！

foreign 以外键生成的策略！(一对一时候讲！)

<!-- 注意：如果列名称是数据库中关键字，用反引号！ -->

<property name=*"desc"* column=*"`desc`"*></property>

## 7. 集合映射

引入：

软件生命周期 : 需求分析、数据库设计(建模)、编码、测试、实施部署、上线、验收！

* 需求：

用户购物，填写送货地址！

一个用户，对应多个送货地址！（一对多）

细化：

用户名、年龄、爱好、地址1、地址2

* 数据库设计？

Set集合映射：



|  |
| --- |
|  |
|  |

* 设计javabean

保存地址：

Set 集合

List 集合

特点：

保存的集合里面只有一个字段数据！

如果要存储更多的信息?

--🡪 1. Map 集合， 可以多存储一个字段

扩展需求：

1. 存地址，还要存储“地址简称”

2. 存储地址简称、省份、城市、邮编、地址详细

* 写映射
* 测试

## 8. 多对一与一对多映射(\*\*\*\*\*)

* 需求：

员工、性别、邮箱、薪水、

部门

* 数据库设计？

employee : id name gender email salary d\_id(fk)

department : id name

* 设计javabean

多employee :

|  |
| --- |
| **private** **int** id;  **private** String name;  **private** String gender;  **private** String email;  **private** **double** salary;  **private** Department department; |

一department :

|  |
| --- |
| **private** **int** id;  **private** String name;  **private** Set<Employee> employees=**new** HashSet<Employee>(); |

* 写映射

|  |
| --- |
| 一Department.hbm.xml |
| <hibernate-mapping package=*"cn.itcast.entity2"*>  <class name=*"Department"* table=*"department"*>  <id name=*"id"* column=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"*></property>  <set name=*"employees"* table=*"employee"*>  <key column=*"d\_id"*></key>  <one-to-many class=*"cn.itcast.entity2.Employee"*/>  </set>  </class>  </hibernate-mapping> |
| 多Employee.hbm.xml |
| <hibernate-mapping package=*"cn.itcast.entity2"*>  <class name=*"Employee"* table=*"employee"*>  <id name=*"id"* column=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* length=*"50"* ></property>  <property name=*"gender"* length=*"50"*></property>  <property name=*"email"* length=*"50"*></property>  <property name=*"salary"*></property>  <many-to-one name=*"department"* column=*"d\_id"* class=*"cn.itcast.entity2.Department"*></many-to-one>  </class>  </hibernate-mapping> |

* 测试

|  |
| --- |
| **private** **static** SessionFactory *sf*;  **static**{  *sf*=**new** Configuration()  .configure()  .addClass(Department.**class**)  .addClass(Employee.**class**)  .buildSessionFactory();  }  @Test  **public** **void** saveTest(){  Session session = *sf*.openSession();  Transaction transaction = session.beginTransaction();  //创建部门  Department dep = **new** Department();  dep.setName("宣传部");  //创建员工  Employee employee = **new** Employee();  employee.setName("Rude");  employee.setGender("female");  employee.setEmail("rude@138.com");  employee.setSalary(9999);  //关系  dep.getEmployees().add(employee);  //保存  session.save(dep);  session.save(employee);  //关闭  transaction.commit();  session.close();  } |

共性问题：

1. 学习框架，东西多，步骤多，配置多，跳来跳去………….

….. 适合自己方法： 多练习，多写，运行，错误，解决，开心！！！

2. hibernate源码

先学习用，在项目是用，会用！

再深入剖析！

## 9. inverse属性

在一对多（多对多等）维护关系关系的时候， 会用到inverse属性！

只能在一的一方使用！

Inverse 表示反转, **指控制权是否转移！**

false (默认值)不反转; 表示控制权不转移，当前方“有控制权”

true 反转; 表示控制权转移，当前方“没有控制权”

演示：

**是否设置inverse属性**，在下面操作的影响：

1) 保存数据 【Y】

// inverse=false, 一对多中，有控制权， 再保存时候才可以设置外键的值！

// inverse=true, 没有控制权，不能设置外键的值！

2) 获取数据 【N】

3) 解除关联 【Y】

只有有控制权，即inverse=false，才可以解除关系！

4) 删除数据的影响 【Y】

// inverse=false, 始终可以删除数据；如果对象有关联会解除所有对这个对象的关联，再删除自身！

// inverse=true, 如果删除的数据有被引用，删除报错，违反主外键约束错误； 否则，才可以删除！

## 10. 多对多映射

* 需求：

项目与开发人员

一个项目，对应多个开发人员！

一个开发人员，对应多个项目

数据：

项目一，

Jack，marry

项目二，

jack，luce

* 数据库设计？

|  |
| --- |
| create table project(  id int primary key auto\_increment,  name varchar(20),  time varchar(30)  )  create table devPerson(  id int primary key auto\_increment,  name varchar(20),  age int,  gender varchar(20)  )  Create table relation(  p\_id int,  d\_id int,  constraint fk1 foreign key (p\_id) references project(id),  constraint fk2 foreign key (d\_id) references devPerson(id),  primary key(p\_id,d\_id)  ) |

* 设计javabean

|  |
| --- |
| Project |
| **private** **int** id;  **private** String name;  **private** String time;  **private** Set<DevPerson> devPersons=**new** HashSet<DevPerson>(); |
| DevPerson |
| **private** **int** id;  **private** String name;  **private** **int** age;  **private** String gender;  **private** Set<Project> projects=**new** HashSet<Project>(); |

* 写映射
* 测试

1. 已经实现多对多

2. 多对多，关联关系维护（inverse属性）

思考：

一对多与多对多中，inverse属性在维护关联关系时候的区别？

Inverse属性在维护关联关系，

一对多，是否设置外键值！（有控制权，才设置值或设置NULL）

对对多，是否添加/删除关系表数据！ （有控制权，才操作！）

## 11. lazy属性

lazy属性表示懒加载

true 支持懒加载

false 关闭懒加载

extra 懒加载(在一对多、多对多，集合方使用！)

在真正使用数据的时候，才向数据库select!

如果调用set集合的size()/isEmpty()方法，只是统计！没有真实查询数据！

懒加载？

在使用数据的时候，才向数据库发送查询的sql语句！

作用：

减少数据库访问次数，提高程序执行效率！

学习lazy：

1） get/load 查询区别

get： 只要get，立刻向数据库发送查询的select

查询主键不存在，返回null

load: 懒加载,用到时候才查询数据库

查询主键不存在，使用就报错！

(在class节点可以配置lazy=false，关闭懒加载，但很少这样用！)

2） 懒加载异常

|  |
| --- |
| /\*  \* 懒加载异常  \* org.hibernate.LazyInitializationException:  \* could not initialize proxy - no Session  \* 注意：  \* session关闭后，不能使用"懒加载数据"！  \*/ |

3） 关联对象懒加载

集合属性懒加载：

## 12. hibernate中对象状态

Javabean对象，持久化的bean对象！PO对象！

* **瞬时/临时状态（Transient Objects）**

使用new操作符初始化的对象不是立刻就持久化的，他们的状态是瞬时的。

(1) 不处于Session的缓存中，也可以说，不被任何一个Session实例关联。

(2) 在数据库中没有对应的记录。

* **持久化状态（Persist Objects）**

持久实例是任何具有数据库标识的实例。它有持久化管理器Session统一管理，持久实例是在事务中进行操作的———他们的状态在事务结束时同数据库进行同步。

(1) 位于一个Session实例的缓存中，也可以说，持久化对象总是被一个Session实例关联。

(2) 持久化对象和数据库中的相关记录对应。

(3) Session在清理缓存时，会根据持久化对象的属性变化，来同步更新数据库。

持久化状态：

调用session的save()/update()/saveOrUpdate()/get()/load()等方法后，

对象就处于持久化状态！

* **离线/游离对象（Detached Objects）**

Session关闭之后，持久化对象就变为离线对象。离线表示这个对象不能再与数据库保持同步，他们不再受Hibernate管理。

(1) 不再位于Session的缓存中，也可以说，游离对象不被Session关联。

(2) 游离对象是由持久化对象转变过来的，因此在数据库中可能还存在与它对应的记录

（前提条件是没有其他程序删除了这条记录）。

## 13.一级缓存

Hibernate一级缓存，也叫做session缓存！

Session缓存：

放入缓存

只要调用session的save()/update()/saveOrUpdate()/get()/load()/list()/iterator等方法时候，操作的对象就会放入一级缓存！

(处于持久化状态的对象，都会在一级缓存里面存在！)

范围：

从创建session，到关闭session之前，一级缓存内容有效，只要关闭session，一级缓存内容就失效！每个session都维护自己独立的缓存区！

维护：

一级缓存由hibernate自身维护，用户想要用一级缓存的内容，必须用hibernate提供的方法！

作用：

什么时候缓存效果比较明显？

-🡪 时间短，范围小！

--》 短时间内，多次操作数据库的时候，session缓存的效果才可能会比较明显！

减少数据库访问次数！！ 提升效率！

相关方法：

Session.flush(); 让一级缓存内容与数据库同步

Session.clear(); 清除一级缓存中所有的对象

Sesion.evict(obj); 清除一级缓存中指定的对象

上述方法，在哪里会用到？

批量更新，1W条数据！！！

1. 每100条记录，更新

2. flush() 同步

3.clear() 清空缓存

面试题： list()与iterator查询区别？

List 一次查询所有的符合条件记录

会放入缓存，但不会从缓存获取！

Iterator： N+1次查询；

举例： 数据库有N=3条记录，通常总共查询数据库次数就是3+1=4次；

首先，先查询所有符合条件记录的主键，再根据每一个主键去查询！

查询的对象，既会放入缓存，又会从缓存中获取!

应用场景：

1. 绝大多数情况都是用list()查询，用哪些数据，一次就全部检索出来！

2. 什么情况用iterator查询？

当确保在用iterator之前，数据已经被查询出来了，此时可以选择iterator查询！ 简单来说，如果想从缓存取数据使用iterator！

## 14. 一对一映射

方法一：

* 需求

用户： 用户名、生日、爱好

身份证： 编号、地址、有效时间\_开始、、有效时间\_开始结束、地址

* 数据库

user :id username birthday hobby

card:cardId address effectiveTime invalidTime u\_id(fk、unique)

* 设计PO

|  |  |
| --- | --- |
| **private** String cardId;  **private** String address;  **private** String effectiveTime;  **private** String invalidTime;  **private** User user; | **private** **int** id;  **private** String username;  **private** String birthday;  **private** String hobby;  **private** Card card; |

* 映射

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib"*>  <class name=*"Card"* table=*"card"* >  <id name=*"cardId"*>  <generator class=*"assigned"*></generator>  </id>  <property name=*"address"* length=*"50"* ></property>  <property name=*"effectiveTime"* length=*"50"*></property>  <property name=*"invalidTime"* length=*"50"*></property>  <many-to-one name=*"user"* class=*"User"* column=*"u\_id"* unique=*"true"*>  </many-to-one>  </class>  </hibernate-mapping> |
| <hibernate-mapping package=*"cn.itcast.hib"*>  <class name=*"User"* table=*"user"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"username"* length=*"50"* ></property>  <property name=*"birthday"* length=*"50"*></property>  <property name=*"hobby"* length=*"50"*></property>  <one-to-one name=*"card"* class=*"Card"*></one-to-one>  </class>  </hibernate-mapping> |

* 测试

|  |
| --- |
| //创建user对象  User user = **new** User();  user.setUsername("Jack");  user.setBirthday("1991-2-30");  user.setHobby("run");  //创建card对象  Card card = **new** Card();  card.setCardId("44200012367i44");  card.setAddress("gz");  card.setEffectiveTime("2006-2-20");  card.setInvalidTime("2012-2-20");  //关系  card.setUser(user);  //保存  session.save(user);  session.save(card); |

方法二

方法一：

* 需求

用户： 用户名、生日、爱好

身份证： 编号、地址、有效时间\_开始、、有效时间\_开始结束、地址

* 数据库

user :id username birthday hobby

card: u\_id(fk、key)cardId address effectiveTime invalidTime

* 设计PO

|  |  |
| --- | --- |
| **private** **int** id;  **private** String username;  **private** String birthday;  **private** String hobby;  **private** Card card; | **private** **int** u\_id;  **private** String cardId;  **private** String address;  **private** String effectiveTime;  **private** String invalidTime;  **private** User user; |

* 映射

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib2"*>  <class name=*"Card"* table=*"card"* >  <id name=*"u\_id"*>  <generator class=*"foreign"*>  <param name=*"property"*>user</param>  </generator>  </id>  <property name=*"cardId"* length=*"50"* ></property>  <property name=*"address"* length=*"50"* ></property>  <property name=*"effectiveTime"* length=*"50"*></property>  <property name=*"invalidTime"* length=*"50"*></property>  <one-to-one name=*"user"* class=*"User"*></one-to-one>  </class>  </hibernate-mapping> |
| <hibernate-mapping package=*"cn.itcast.hib2"*>  <class name=*"User"* table=*"user"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"username"* length=*"50"* ></property>  <property name=*"birthday"* length=*"50"*></property>  <property name=*"hobby"* length=*"50"*></property>  <one-to-one name=*"card"* class=*"Card"*></one-to-one>  </class>  </hibernate-mapping> |

* 测试

同上

## 15. 组件映射

引入：

类与类之间的关系：

组合关系，

一个类中包含了另外一个类的引用！

举例：

汽车与车轮！

**数据库**

car：id name size color count

**设计javabean**

|  |  |
| --- | --- |
| **private** **int** id;  **private** String name;  **private** Wheel wheel; | **private** **int** count;  **private** String color;  **private** **int** size; |

映射

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib3"*>  <class name=*"Car"* table=*"car"* >  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* length=*"50"* ></property>  <!--组合映射-->  <component name=*"wheel"* class=*"Wheel"*>  <property name=*"size"*></property>  <property name=*"count"*></property>  <property name=*"color"* length=*"50"*></property>  </component>  </class>  </hibernate-mapping> |

测试

|  |  |
| --- | --- |
| //创建user对象  Wheel wheel = **new** Wheel();  wheel.setColor("黑色");  wheel.setCount(4);  wheel.setSize(30);  //创建card对象  Car car = **new** Car();  car.setName("宝马");  //关系  car.setWheel(wheel);  //保存  session.save(car); | Car car =(Car) session.get(Car.**class**, 1);  System.*out*.println(car.getId());  System.*out*.println(car.getName());  System.*out*.println(car.getWheel().getColor()); |

类组合关系的映射，也叫组件映射！

特点：

* 组件类和包含的组件类同时映射到一个表

## 16. 继承映射

继承关系，

动物：猫、狗。。。。。。。。

### 简单继承映射

实现:

1. 父类信息与子类信息写到一个映射文件中！

总结：

1. 每个子类都会有一个映射文件，文件个数太多！

2. 每个映射文件，都要重复写父类的公用的信息

### 继承映射

最简单的方法，只有一个子类继承一个父类

javabean

|  |  |  |
| --- | --- | --- |
| 动物 | 猫继承动物 | 狗继承动物 |
| **private** **int** id;  **private** String name;  **private** String sex; | private String catching; | private String play; |

映射

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib4"*>  <class name=*"Cat"* table=*"cat"* >  <!--父类信息-->  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* length=*"50"* ></property>  <property name=*"sex"* length=*"50"* ></property>  <!--子类信息-->  <property name=*"catching"*></property>  </class>  </hibernate-mapping> |
|  |

测试

|  |  |
| --- | --- |
| //创建cat对象  Cat cat=**new** Cat();  cat.setName("三脚猫");  cat.setSex("雄猫");  cat.setCatching("捉老鼠");  //保存  session.save(cat); | Cat cat =(Cat) session.get(Cat.**class**, 1);  System.*out*.println(cat);  Query q1 = session.createQuery("from Cat");  System.*out*.println(q1.list());  //查询父类1.要写全名 2.父类会查询子类的所有信息  Query q2 = session.createQuery("from cn.itcast.hib4.Animal");  System.*out*.println(q2.list()); |

一个父类，2个子类 !!!

**方式1：一张表**

整个继承结构用一张表存储！

T\_animal 仅此一张表

Id name sex catching1 play type\_(hibernate: 鉴别器字段！)

1 猫 抓.. NULL cat

2. 狗 NULL 散步 dog

hibernate:中鉴别器字段， 主要用于区分不同的子类！

①JavaBean同上

②映射Animal.hbm.xml

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib5"*>  <class name=*"Animal"* table=*"animal"* >  <!--父类信息-->  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <!-- 鉴别器 -->  <discriminator column=*"type\_"*></discriminator>  <property name=*"name"* length=*"50"* ></property>  <property name=*"sex"* length=*"50"* ></property>  <!--子类猫/鉴别器字段重命名-->  <subclass name=*"Cat"* discriminator-value=*"cat"*>  <property name=*"catching"*></property>  </subclass>  <!--子类狗-->  <subclass name=*"Dog"* discriminator-value=*"dog"*>  <property name=*"play"*></property>  </subclass>  </class>  </hibernate-mapping> |

③测试

|  |
| --- |
| //创建Cat对象  Cat cat=**new** Cat();  cat.setName("三脚猫");  cat.setSex("雄猫");  cat.setCatching("捉老鼠");  //保存  session.save(cat);  //创建Dog对象  Dog dog=**new** Dog();  dog.setName("小黑");  dog.setPlay("玩骨头");  dog.setSex("雄狗");  //保存  session.save(dog); |

总结：

1. 不管继承结构多复杂，只要一个映射文件：Animal.hbm.xml

2. 不管有多少个子类，都存储多一张表！

缺点：

表结构不符合数据库设计原则！存在大量数据冗余，不可取的！

**方式2：三张表**

|  |  |  |
| --- | --- | --- |
| animal 父表  Id name sex  100 猫 ..  200. 狗 .. | cat 猫  Animal\_id (fk) catching  100 …. | dog 狗  Animal\_id(fk) play  200 … |

①JavaBean同上

②映射Animal.hbm.xml

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib6"*>  <class name=*"Animal"* table=*"animal"* >  <!--父类信息-->  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* length=*"50"* ></property>  <property name=*"sex"* length=*"50"* ></property>  <!--子类猫-->  <joined-subclass name=*"Cat"* table=*"cat"*>  <key column=*"animal\_id"*></key><!--外键-->  <property name=*"catching"*></property>  </joined-subclass>  <!--子类狗-->  <joined-subclass name=*"Dog"* table=*"dog"*>  <key column=*"animal\_id"*></key>  <property name=*"play"*></property>  </joined-subclass>  </class>  </hibernate-mapping> |

③测试同上

总结：

父类对应表，每个子类对应一张表，且子类引用父类！

1个映射文件

缺点：

表关系太复杂，影响存储、读取效率！

数据库设计虽然符合规范，但不可取！

**方式3：两张表(推荐)**

每个子类对应一张表，父类不对应表！

①JavaBean同上

②映射Animal.hbm.xml

|  |
| --- |
| <hibernate-mapping package=*"cn.itcast.hib7"*>  <!-- 防止animal自动建表abstract="true" -->  <class name=*"Animal"* abstract=*"true"* >  <!--父类信息-->  <id name=*"id"*>  <generator class=*"assigned"*></generator>  </id>  <property name=*"name"* length=*"50"* ></property>  <property name=*"sex"* length=*"50"* ></property>  <!--子类猫-->  <union-subclass name=*"Cat"* table=*"cat"*>  <property name=*"catching"*></property>  </union-subclass>  <!--子类狗-->  <union-subclass name=*"Dog"* table=*"dog"*>  <property name=*"play"*></property>  </union-subclass>  </class>  </hibernate-mapping> |

③测试同上

|  |
| --- |
| //创建Cat对象  Cat cat=**new** Cat();  cat.setId(11);  cat.setName("三脚猫");  cat.setSex("雄猫");  cat.setCatching("捉老鼠");  //保存  session.save(cat);  //创建Dog对象  Dog dog=**new** Dog();  dog.setId(101);  dog.setName("小黑");  dog.setPlay("玩骨头");  dog.setSex("雄狗");  //保存  session.save(dog); |

总结：

1. 一个映射文件

2. 每个子类对应一张表

3. 都符合数据库设计原则；

注意：

主键不能为自增长！

## 17. cascade 级联属性

cascade 表示级联

save-update 级联保存更新

delete 级联删除

none 默认值，不级联

save-update,delete 级联保存更新、删除

all 同上

## 18. HQL 查询

Hibernate提供提供的查询方式：

概述：

1) 主键查询

get/load

|  |
| --- |
| Address address = (Address) session.get(Address.class, 7); |

2) 对象导航查询

查询对象，以及对象关联的属性！

|  |
| --- |
| Query q = session.createQuery("from Address where id=4");  // 方法1： 查询得到list集合  //System.out.println(q.list());  // 方法2：查询得到迭代器对象  //System.out.println(q.iterate());  // 方法3： 查询返回唯一结果，如果结果不唯一，报错！  //Address address = (Address) q.uniqueResult();  // 其他方法: 分页  q.setFirstResult(0);  q.setMaxResults(1);  System.out.println(q.list()); |

3) HQL 查询

|  |
| --- |
| Query q = null;  //1. 查询全部  q = session.createQuery("from Address");  q = session.createQuery("from Address a"); // 给别名  q = session.createQuery("select a from Address a"); // 查询全部  // q = session.createQuery("select \* from Address a"); // 报错  //2. 查询指定的列 (一旦查询指定的列，就不会封装为对象了！！！！)  // 返回:List<Integer>  q = session.createQuery("select a.id from Address a");  // 返回：List<Object[]>  q = session.createQuery("select a.id,a.province from Address a");  //3. 查询指定的列，封装为对象  q = session.createQuery("select new Address(a.id,a.province) from Address a");  //4. 条件查询  q = session.createQuery("from Address where id=7 or id=8");  q = session.createQuery("from Address where id in(7,8)");  q = session.createQuery("from Address where id between 7 and 8");  q = session.createQuery("from Address where address like '%天河1%'");  //5. 占位符查询  q = session.createQuery("from Address where city=? and address = ?");  // 设置占位符问号参数  q.setParameter(0, "广州市");  q.setParameter(1, "天河1区骏景花园");    // 6. 命名参数查询  String condition = "天河1";  q = session.createQuery("from Address where city=:city123 and address=:address");  q.setParameter("city123", "广州市");  q.setParameter("address", "天河1区骏景花园");  // 举例2  q = session.createQuery("from Address where address like ?");  q.setParameter(0, "%"+condition+"%");  System.out.println(q.list());  //7聚合函数统计,分组、分组筛选  /\*  \* 1. 需求2: 统计每个人的地址的数目, 显示地址人数>1  \* SELECT user\_id,COUNT(\*) FROM t\_address GROUP BY user\_id HAVING COUNT(\*) >1;  \*/  // a. 统计所有  //Query q = session.createQuery("select count(\*) from Address ");  //Long num = (Long) q.uniqueResult();  // b. 实现需求  Query q = session.createQuery("select count(\*), a.user from Address a group by a.user having count(\*) > 1");  List list = q.list();  System.out.println(list);  //8内连接  //Query q = session.createQuery("from User u inner join u.addresses");  Query q = session.createQuery("from Address a inner join a.user");  List<Object[]> list = q.list();  // 显示地址，用户名  for (Object[] objs : list){  // 获取第一个元素: 地址对象  Address add = (Address) objs[0];  // 获取第二个元素: 用户对象  User user = (User) objs[1];  // 输出:  System.out.println(add.getAddress() + "\t" + user.getName());  }  System.out.println(list);  // 内连接  Query q = session.createQuery("from Address a inner join fetch a.user");  List<Address> list = q.list();  for (Address a : list){  System.out.println(a.getAddress() + "\t" + a.getUser().getName());  }  // 查询返回地址  Query q = session.createQuery("from Address");  List<Address> list = q.list();  // 遍历  for (Address a : list){  System.out.println(a.getAddress() + "\t" + a.getUser().getName());  } |

4)Criteria 完全面向对象查询

|  |
| --- |
|  |

5) 本地sql查询！

## 19. Hibernate连接池

连接池：

提高连接的使用效率！

Hibernate对连接池的支持：

1) 自带的连接池：只维护一个连接

2) hibernate对c3p0支持

#################################

### Hibernate Connection Pool ###

#################################

hibernate.connection.pool\_size 1 默认支持1个连接

###########################

### C3P0 Connection Pool### hibernate对c3p0连接池的支持

###########################

#hibernate.c3p0.max\_size 2 最大连接数

#hibernate.c3p0.min\_size 2 最小连接数

#hibernate.c3p0.timeout 5000 超时时间 (毫秒)

#hibernate.c3p0.max\_statements 100 一次最多执行的sql语句个数

#hibernate.c3p0.idle\_test\_period 3000 空闲测试时间

#hibernate.c3p0.acquire\_increment 2 每次连接增量

#hibernate.c3p0.validate false

|  |
| --- |
| <property name="hibernate.c3p0.min\_size">3</property>  <property name="hibernate.c3p0.max\_size">6</property>  <property name="hibernate.c3p0.timeout">5000</property>  <property name="hibernate.c3p0.max\_statements">100</property>  <property name="hibernate.c3p0.idle\_test\_period">3000</property> |

## 20. 二级缓存

Hibernate一级缓存：

基于Session缓存！

Hibernate二级缓存：

Hibernate提供了二级缓存，为了解决一级缓存缓存时间短效果不明显的问题！

二级缓存数据，可以供多个session共享！ 全局的缓存！

Hibernate提供的二级缓存，以可插配的形式提供的，即是一个插件！ 用户如果想使用二级缓存，就需要在hibernate.cfg.xml 中配置即可； 不想用直接移除配置！

二级缓存插件，hibernate提供了默认支持的多种形式，hibernate也有对插件的默认实现！

二级缓存配置：

##########################

### Second-level Cache ### 关于二级缓存的配置

## disable the second-level cache

#hibernate.cache.use\_second\_level\_cache false 二级缓存默认不开启

## enable the query cache

#hibernate.cache.use\_query\_cache true 是否开启查询缓存！

## choose a cache implementation 二级缓存框架的实现

#hibernate.cache.provider\_class org.hibernate.cache.EhCacheProvider

EhCache缓存框架

#hibernate.cache.provider\_class org.hibernate.cache.EmptyCacheProvider

hibernate.cache.provider\_class org.hibernate.cache.HashtableCacheProvider 默认实现

#hibernate.cache.provider\_class org.hibernate.cache.TreeCacheProvider

#hibernate.cache.provider\_class org.hibernate.cache.OSCacheProvider OSCache缓存框架

#hibernate.cache.provider\_class org.hibernate.cache.SwarmCacheProvider

一、使用二级缓存(hibernate.cfg.xml)：

1） 开启二级缓存

2） 指定二级缓存哪种实现

3） 指定那些对象，需要放入二级缓存

|  |
| --- |
| <!-- 开启二级缓存 -->  <property name=*"hibernate.cache.use\_second\_level\_cache"*>true</property>  <!-- 实现类型 -->  <property name=*"hibernate.cache.provider\_class"*>org.hibernate.cache.HashtableCacheProvider</property>  <!-- 指定的类放入缓存 -->  <class-cache usage=*"read-only"* class=*"cn.itcast.entity.Student"*/> |

4） 测试

|  |
| --- |
| Session session1 =*sf*.openSession();  Transaction ts1 = session1.beginTransaction();    Student student1 = (Student)session1.get(Student.**class**, 8);  System.*out*.println(student1);    ts1.commit();  session1.close();  System.*out*.println("--------------------------------------------------");  Session session2 =*sf*.openSession();  Transaction ts2 = session2.beginTransaction();  //第二次查询时没有再去执行sql语句，直接去二级缓存取  Student student2 = (Student)session2.get(Student.**class**, 8);  System.*out*.println(student2);    ts2.commit();  session2.close(); |

a. 打开一个session

get

关闭

b. 打开一个session

get （如果二级缓存配置应用成功，这里没有查询数据库！）

二、集合缓存

集合缓存中，集合的元素也要加入二级缓存！

如果想把Student对象集合teachers属性也放入二级缓存，需要配置集合缓存，还要多添加一个指定类

|  |
| --- |
| <class-cache usage=*"read-only"* class=*"cn.itcast.entity.Teacher"*/>  <!-- 如果想把Student对象集合属性也放入二级缓存，需要配置集合缓存 -->  <collection-cache usage=*"read-only"* collection=*"cn.itcast.entity.Student.teachers"*/> |

三、缓存策略

usage="read-only" 只读

usage=" read-write" 读写

usage=" nonstrict-read-write" 非严格的读写

usage=" transactional" 基于事务的策略

四、查询缓存

思考：

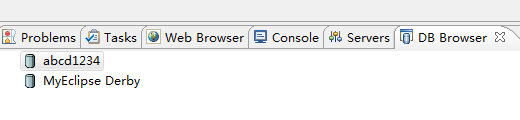
更新数据，一级缓存数据是否会更新？二级缓存数据是否会更新？

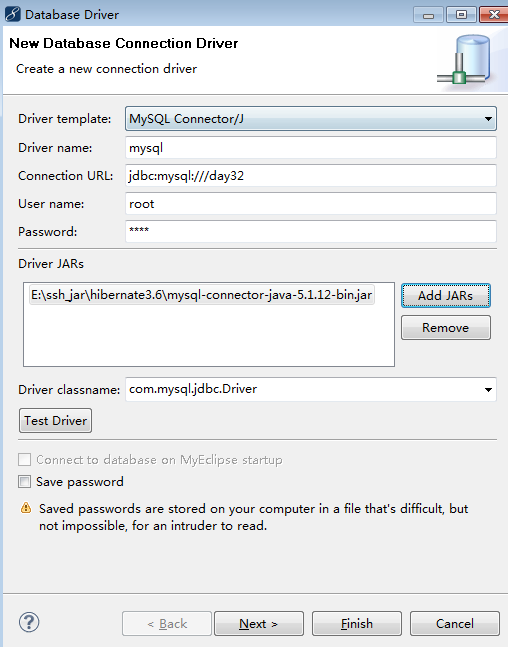
## 21. Hibernate对session创建的支持

### 创建session2种方式

|  |
| --- |
| @Test  **public** **void** session() {  /\* openSession每次都创建一个新的session  \* 注意：  \* 查询可以不开启事务！！  \*/  // Session session1 = sf.openSession();  // Session session2 = sf.openSession();  // System.out.println(session1 == session2);  // session1.close();  // session2.close();          // getCurrentSession 线程方式创建session，必须配置hibernate.current\_session\_context\_class属性为thread  /\*  \* 创建过程： 1. 第一次getCurrentSession(),  \* \* 先从当前线程上获取session  \* \* 没有获取到，就调用openSession创建session  \* \* 绑定到当前线程上  \* 注意：  \* 1. 只要这种方式创建的session，执行操作一定在事务范围！  \* 2. 可以不用关闭session，只要事务提交会自动关闭！  \*  \*/  Session session3 = *sf*.getCurrentSession();  Session session4 = *sf*.getCurrentSession();  System.*out*.println(session3 == session4);  session3.close();  //session4.close(); 关闭  } |

### 会打开MyEclipse数据库视图





### 会用MyEclipse逆向工程

注意：

需要通过MyEclipse添加对hibernate·的支持！

## 22. 案例： 项目中session管理方式：

Open Session In View 模式

1. 分析

\* 过滤器

Init()/doFilter()/destroy()

doFilter()

chain.doFilter(request,response); 放行

\* 项目中如何管理事务？

2 实现

\* 项目、实体类、映射

\* SessionFilter.java 管理session创建

\* dao

查询所有地址!

\* service

\* action

列表显示

\* WEB-INF/list.jsp

显示： 地址、用户名(懒加载)!

# Spring

## 1. 开发回顾

步骤：

1. 传统的mvc开发

2. 思考：

哪些需要优化：对象如何创建问题？

|  |
| --- |
| **public** **class** UserAction {  // 创建对象  **private** IUserService userService = **new** UserService();  **public** String execute(){  userService.save();  **return "";**  }  } |
| **public** **class** UserService **implements** IUserService {  // 创建对象 !  //private IUserDao userDao = new UserDao();  **private** IUserDao userDao = **new** UserDao\_oracle();  @Override  **public** **void** save() {  userDao.save();  }  } |

解决1： 对象创建？

1. 更换实现不改代码！

2. 某一层依赖其他层，不依赖实现，只依赖解耦，从而做到“解耦”

“开闭原则”

软件对功能的新增是开的，对功能的修改是关闭的！

解决2： 对象创建，什么时候创建？创建多少个？

Dao / service 应该是单例！ 【启动时候创建， 创建一个！】

Action 多例！ 【用到时候创建】

Spring, 可以很轻松的解决上面对象创建、创建个数、创建时间问题！

## 2. Spring 框架

### 概述

Spring 主要是解决：对象创建、对象依赖关系！提供一站式服务

**1. 六大模块**

|  |  |  |
| --- | --- | --- |
| Spring六大模块  （提供的是一站式解决方案）  Spring版本：3.2版本 | **Web service dao**  事物  整合  整合  **Struts2 Spring hibernate** | |
| ①**Spring IOC容器**  IOC inverse of control控制反转  也叫控制反转容器  对象创建交给外部容器完成 | ③**SpringAOP**  面向切面编程 | ⑤**SpringORM支持**  对hibernate支持 |
| ②**SpringWeb支持**  1. Spring与Struts整合  2. SpringMVC | ④**SpringJdbc支持**  JdbcTemplate工具栏 | ⑥**SpringJEE**  EJB组件等 |

2. 源码

3.2 版本！

Spring2.5 包含spring功能，以及spring依赖的功能！

Spring3.2 中，只有spring的功能！

-🡪 如果spring框架依赖了其他组件，必须要单独下载！

### Hello , Spring !

步骤：

1. 引入spring核心包

IOC容器支持包！

|  |
| --- |
| commons-logging-1.1.3.jar 【单例下载  或spring-framework-2.5.6\lib\jakarta-commons】  spring-beans-3.2.5.RELEASE.jar 【Spring3.2 源码】  spring-context-3.2.5.RELEASE.jar  spring-core-3.2.5.RELEASE.jar  spring-expression-3.2.5.RELEASE.jar |
|  |

2. 配置文件

applicationContext.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd"*>  </beans> |

3. Api

加载配置文件，获取对象！

|  |
| --- |
| @Test  **public** **void** testIOC() **throws** Exception {  // IOC容器 = ApplicationContext + applicationContext.xml  // 容器对象  ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");  // 从容器获取对象,1  Date date = (Date) ac.getBean("date");  // 从容器获取对象,2  User user = (User) ac.getBean("user");  } |

## 3. Spring bean对象创建细节

|  |
| --- |
| **public** **class** App {  // 容器对象 (从当前字节码所在类的路径，去找bean.xml文件)  // private ApplicationContext ac =  // new ClassPathXmlApplicationContext("bean.xml",getClass());  /\*\*  \* 对象创建细节：  \* ① 单例/多例  \* scope="singleton" 默认值； 为单例！ "dao/service"  \* 默认是在创建容器对象时候，就创建了所有的单例对象！  \* scope="prototype" 多例！  \* 只能在从容器中获取对象的时候，每次都会创建一个新的对象！ "action"  \* ② 单例对象，是否延迟初始化  \* lazy-init="default" 默认不延迟初始化，即创建容器，就创建所有单例对象！  \* lazy-init="true" 延迟初始化； 在创建容器时候不创建单例对象，再第一次使用对象时候，才创建单例的对象！ 加载xml时，如果bean内有lazy-init="true"属性，不会立即创建bean对象，等到再次使用对象是才创建。  \*  \* ③ 初始化、销毁  \* init-method="init" 初始化方法； 在对象创建之后执行  \* destroy-method="destroy" 容器销毁的时候调用每一个对象的destroy方法！  在对应的bean对象中，加入init()和destroy()方法  \* 注意： 1. 必须要手动调用容器的ac.destroy();方法  \* 2. 只有单例的对象才会执行destroy()方法！  \* 面试题： spring中bean生命周期？  \*/  @Test  **public** **void** testIOC() **throws** Exception {  // 创建容器对象 (web项目，启动就创建！！！)  ClassPathXmlApplicationContext ac =  **new** ClassPathXmlApplicationContext("cn/itcast/b\_ioc\_object\_detail/bean.xml");  System.*out*.println("----------");  // 从容器中获取（单例）对象!  User user1 = (User) ac.getBean("user");  User user2 = (User) ac.getBean("user");  System.*out*.println(user1 == user2);  // ac.destroy();  }  } |

## 4. Spring IOC

Spring IOC 控制反转容器

创建对象 ！

例如： UserService service = new UserService();

UserDao dao = new UserDao();

DI dependency injection 依赖注入!

处理对象的关系！ 给属性赋值

例如： Action依赖Service ， Service 又依赖dao!

给UserService 对象中的dao 属性赋值！

面试题：简述控制反转容器与依赖注入的关系?

### 4.1 IOC, 创建对象

1) 调用无参数构造函数

什么都不需要做，输出对象为默认值，

2）调用带参数构造函数

参数在applicationContext.xml写死

|  |
| --- |
| <!--按顺序写-->  <bean id=*"user"* class=*"cn.itcast.test1.User"*>  <constructor-arg value=*"101"*></constructor-arg>  <constructor-arg value=*"Jack"*></constructor-arg>  </bean> |
| <!--按标签读取-->  <bean id=*"user"* class=*"cn.itcast.test1.User"*>  <constructor-arg index=*"0"* value=*"101"*></constructor-arg>  <constructor-arg index=*"1"* value=*"Jack"*></constructor-arg>  </bean> |
| <!--按类型读取-->  <bean id=*"user"* class=*"cn.itcast.test1.User"*>  <constructor-arg type=*"int"* value=*"101"*></constructor-arg>  <constructor-arg type=*"java.lang.String"* value=*"Jack"*></constructor-arg>  </bean> |

3) 工厂

静态方法 非静态方法

|  |
| --- |
| <!-- 调用静态方法 -->  <bean id=*"user1"* class=*"cn.itcast.test1.UserFactory"* factory-method=*"staticMethod"*></bean>  <!-- 调用非静态方法 ，要先创建对象-->  <bean id=*"userFactory"* class=*"cn.itcast.test1.UserFactory"*></bean>  <bean id=*"user2"* factory-bean=*"userFactory"* factory-method=*"NoStaticMethod"*></bean> |

### 4.2 IOC, 依赖注入

DI， 叫依赖注入！ 就是给对象的属性赋值！

有哪些方式？

**1) 构造函数给对象属性赋值**（在4.1有例子）

**2) set 方法给对象属性赋值**

首先在封装类中添加set方法，再写配置文件

|  |
| --- |
| <bean id=*"user"* class=*"cn.itcast.test1.User"*>  <property name=*"id"* value=*"102"*></property>  <property name=*"name"* value=*"Jack"*></property>  </bean> |

最重要)Set方法给对象属性赋值

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd"*>  <!-- 创建dao 实例 -->  <bean id=*"userDao"* class=*"cn.itcast.e\_eg.UserDao"*></bean>  <!-- 创建service 实例,注入dao-->  <bean id=*"userService"* class=*"cn.itcast.e\_eg.UserService"*>  <property name=*"userDao"* ref=*"userDao"*></property>  </bean>  <!-- 创建action 实例，注入service-->  <bean id=*"userAction"* class=*"cn.itcast.e\_eg.UserAction"* scope=*"prototype"*>  <property name=*"userService"* ref=*"userService"*></property>  </bean>  </beans> |

**3) P 名称空间给属性赋值**

|  |
| --- |
| <bean id=*"user"* class=*"cn.itcast.test1.User"* p:id=*"102"* p:name=*"Jack"*></bean> |

总结：

<bean>

<property>

…….

太麻烦！ 可以使用p名称空间简化！

注意：

P 名称空间，使用前需要先引入名称空间！

|  |
| --- |
| xmlns:p="http://www.springframework.org/schema/p" |

|  |
| --- |
| <bean id=*"userDao"* class=*"cn.itcast.f\_ioc\_di\_p.UserDao"*></bean>  <bean id=*"userService"* class=*"cn.itcast.f\_ioc\_di\_p.UserService"* p:userDao-ref=*"userDao"*></bean>  <bean id=*"userAction"* class=*"cn.itcast.f\_ioc\_di\_p.UserAction"* p:userService-ref=*"userService"* scope=*"prototype"*></bean> |

**4）内部bean**（内面的bean对象的id自动调用外面bean对象property的name）

|  |
| --- |
| <bean id=*"userAction"* class=*"cn.itcast.f\_ioc\_di\_p.UserAction"* scope=*"prototype"*>  <property name=*"userService"*>  <bean class=*"cn.itcast.f\_ioc\_di\_p.UserService"*>  <property name=*"userDao"*>  <bean class=*"cn.itcast.f\_ioc\_di\_p.UserDao\_oracle"*></bean>  </property>  </bean>  </property>  </bean> |

**5）自动装配**

|  |
| --- |
| <!--  方式4： 自动装配  autowire="byName" 根据名称去IOC容器找指定的对象，再注入到set方法！  autowire="byType" 根据需要注入的类型，去容器找对象注入！  注意： 类型必须唯一！  default-autowire="byName" 定义到全局beans节点上的属性，表示整个配置文件根据名称自动装配！  总结：  好用！配置简单！属性注入交给spring自动注入（自动装配）!  但出现错误，不好排查、维护！  （自己定义xml配置时候，最好不要用！）  -->  <bean id=*"userDao"* class=*"cn.itcast.f\_ioc\_di\_p.UserDao"*></bean>  <bean id=*"userService"* class=*"cn.itcast.f\_ioc\_di\_p.UserService"*></bean>  <bean id=*"userAction"* class=*"cn.itcast.f\_ioc\_di\_p.UserAction"*></bean> |

**6）基于注解实现**

目标：简化配置！

常用注解：

@Component

把对象加入IOC容器，加入容器的对名称默认是类名，第一个字母小写

@Component("userDao")

把指定名称的对象，加入ioc容器!<bean id=userDao …. />

@Repository 同@Component ; 作为持久层的组件使用这个 (dao)

@Service 同@Component 业务逻辑层组件 (service)

@Controller 同@Component; 控制层组件 (action)

@Resource 自动从IOC容器找对象，注入到属性中!

@Value(“Jack”) 自动从IOC容器找对象，注入到属性中!用于其他赋值属性

|  |
| --- |
| 如：@Value("Jack")  **private** String name; |

查找规则：

@Resource

**private** IUserDao userDao;

1. 先根据名称找,找不到在根据类型找！

2. 如果类型不唯一，报错！

@Resource(name = "userDao") // 只根据名称找

注意：

如果根据找的时候，类型不符合，也会根据名称找！

总结：

注解与XML区别？ 哪个更常用？

注解，

更加简单！ 但维护不方便！ 如果修改对象关系，需要修改代码！

XML，

配置虽然繁琐，但维护方便！ 修改后不需要重新编译！

怎么用，

1. 注解与XML可以同时用

2. 一般情况，通用的配置，都用XML！其他可以根据需要（习惯）都可以！

## 5. Spring 与 Struts 整合

关键：

Action对象，可以选择交给spring完成！

步骤：

1) 引入jar包

Struts 核心包

SpringCore核心包

Spring-web 支持

spring-web-3.2.5.RELEASE.jar 【spring源码】

struts2-spring-plugin-2.3.4.1.jar 【struts源码】

2) 配置

Web.xml

\* struts核心过滤器

\* 创建SpringIOC容器对象

Struts.xml

Bean.xml

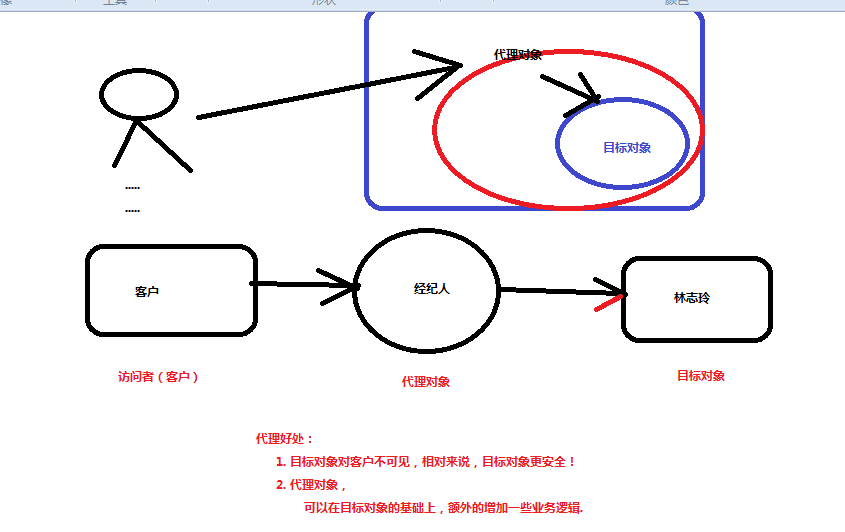
3) dao/service/action

## 6代理模式

Proxy 代理！

代理模式，

代理，这种模式，提供了对目标对象另外的访问方式，即通过代理访问目标对象！



### 静态代理

需求：

1. IUserDao.java

2. UserDao.java

…..

例子：一个企业老板要约春哥

|  |  |
| --- | --- |
| **//必须提供一个接口给目标对象和代理实现**  **interface** SingerImpl {  **void** engagement();  } | **//目标对象**  **public** **class** Singer **implements** SingerImpl {  **private** String name;  **public** Singer() {}  **public** Singer(String name) {  **this**.name = name;  }  **public** **void** engagement() {  System.*out*.println("我是"+name+"，原来是你约了我！");  }  } |
| **public** **class** SingerProxy **implements** SingerImpl {  **private** Singer singer;  **public** SingerProxy(Singer singer) {  **this**.singer = singer;  }  **public** **void** engagement() {  System.*out*.println("我是春哥的经纪人，允许你去约");  singer.engagement();  System.*out*.println("我是春哥的经纪人，本次约会结束");  }  } | **public** **static** **void** main(String[] args) {  SingerProxy sp = **new** SingerProxy(**new** Singer("春哥"));  sp.engagement();  }   |  | | --- | | 结果：  我是春哥的经纪人，允许你去约  我是春哥，原来是你约了我！  我是春哥的经纪人，本次约会结束 | |

特点：

1. 目标对象要实现接口

2. 代理对象要实现与目标对象一样的接口

3. 缺点：

\* 如果目标对象发生变化，代理也需要修改

\* 一个目标对象，要有一个代理对象，代理过多！

### JDK代理

也叫动态代理！或接口代理！

利用JdkApi，

在运行时期，在内存中动态构建代理对象！

jdk会根据目标对象实现的接口，**生成这个接口的实现类**！

要求：

目标对象一定要实现接口!

|  |  |
| --- | --- |
| static [Object](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html) | [**newProxyInstance**](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/Proxy.html#newProxyInstance(java.lang.ClassLoader, java.lang.Class[], java.lang.reflect.InvocationHandler))([ClassLoader](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/ClassLoader.html) loader, [Class](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html)<?>[] interfaces, [InvocationHandler](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/InvocationHandler.html) h)            返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。 |

参数1: 当前目标对象使用类加载器！

参数2： 指定目标对象实现的接口类型的数组！

参数3： 事件处理程序; 当执行代理对象方法时候触发（在事件处理程序中，调用目标对象，及增加额外的业务逻辑！！！！！！！）

Singer和SingerImpl其他同上

|  |
| --- |
| //代理  **public** **class** SingerProxy {  **private** Object target;  **public** SingerProxy(Object target) {  **this**.target = target;  }  //针对目标对象，生成代理对象  **public** Object getParxyInstrance(){  **return** Proxy.*newProxyInstance*(  target.getClass().getClassLoader(),//类加载器  target.getClass().getInterfaces(), //目标对象实现的接口  **new** InvocationHandler(){ //事件处理程序  @Override  **public** Object invoke(Object proxy, Method method,  Object[] args) **throws** Throwable {  //方法返回值  Object returnValue=**null**;  //获取代理方法名  String methodName=method.getName();  //代理的方法：  **if**(methodName.equals("engagement")){  System.*out*.println("我是春哥的经纪人，允许你去约");  //执行目标对象方法  returnValue=method.invoke(target, args);  System.*out*.println("我是春哥的经纪人，本次约会结束");  }**else**{  System.*out*.println("请使用正确的途径去约会");  }  **return** returnValue;  }  });  }  } |
| //测试方法  //要用接口去接收，上升转型  SingerImpl sp =(SingerImpl) **new** SingerProxy(**new** Singer("春哥")).getParxyInstrance();  sp.engagement();//执行代理的方法 |

### Cglib代理

思考：如果目标对象没有实现接口，怎么办？怎么生成代理对象？

Class UserDao { 给这个对象生成子类

Void save();

}

Class Proxy extends UserDao{

重写save方法！！！

}

这种就是子类的方式生成代理对象！即Cglib子类代理！

Spring提供了cglib子类代理！

(在spring的核心包中已经继承了这个功能！)

* **JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用CGLIB实现。**
* **CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）。**
* **CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

|  |
| --- |
| //代理  **public** **class** SingerProxy **implements** MethodInterceptor {  **private** Object target;  **public** SingerProxy(Object target) {  **this**.target = target;  }  //生成一个代理对象（子类）  **public** Object getProxyInstance(){  //创建一个字节码工具类  Enhancer en = **new** Enhancer();  //设置父类  en.setSuperclass(target.getClass());  //设置回调函数  en.setCallback(**this**);  //创建对象返回  **return** en.create();  }    //相当于事件处理程序，执行代理方法时候就会触发  @Override  **public** Object intercept(Object obj, Method method, Object[] args,  MethodProxy methodProxy) **throws** Throwable {  System.*out*.println("我是春哥的经纪人，允许你去约");  //执行目标对象  Object returnValue=method.invoke(target, args);  System.*out*.println("我是春哥的经纪人，本次约会结束");  **return** returnValue;  }  } |
| Singer s = (Singer) **new** SingerProxy(**new** Singer("春哥")).getProxyInstance();  s.engagement(); |

### 总结

在Spring的Aop编程中，

如果目标对象有实现接口，使用Jdk代理！

如果目标对象没有实现接口， 使用Cglib代理！

如果目标对象没有实现接口, 且为final，生成不了代理对象，报错！

## 7. 手动实现一个Aop编程的例子

Aop 解决什么问题？ 不用Aop行不行？

关注点代理，

很多重复执行的代码，叫关注点代码！

Aop 解决什么问题？

分离关注点代码与业务代码！

只有分离了，

关注点代码写一次，在运行时期执行业务时候，动态植入“关注点代码”!

|  |
| --- |
| public void add(User user) {  Session session = null;  Transaction trans = null;  try {  session = HibernateSessionFactoryUtils.getSession();  trans = session.beginTransaction(); // 关注点代码    **session.save(user); // 业务**    trans.commit();  } catch (Exception e) {  e.printStackTrace();  if(trans != null){  trans.rollback();  }  } finally{  HibernateSessionFactoryUtils.closeSession(session);  }  } |

# 过程化分离

# 对象化分离

# 代理模式实现: 关注点代码与业务代码, 分离！

## 8. Aop编程

### 概念、属于

Aop aspect orient programming 面向切面编程!

关注点：

重复执行的代码， 叫关注点代码！

切面：

关注点代码形成的类，叫切面类！

面向切面编程

把很多重复的功能抽取出来，形成一个切面类！面向面向编程，就是写这个切面类！

在运行时期，动态植入切面类代码！！！！

应用：

程序中的切面：事务切面、权限切面、日志切面。。。。。

切入点表达式：

通过接入点表达式规则，可以给指定的类生成代理对象！

通知:

重复的代码的方法，叫“通知”！

Aop编程：

引入Aop支持包!

spring-aop-3.2.5.RELEASE.jar 【Spring3.2 源码】

**aspectjrt.jar 【spring-framework-2.5.6\lib\aspectj】**

**aspectjweaver.jar 【spring-framework-2.5.6\lib\aspectj】**

aopalliance.jar 【spring-framework-2.5.6\lib\aopalliance】

aspectj 组件：

也可以单独下载！

### 注解方式实现

1. 引入Aop jar包

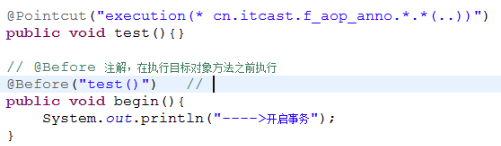
2. 引入Aop名称空间

3. 开启Aop注解（扫描aop相关的注解）

4. 使用Aop注解

@Aspect 指定一个类为切面类

@Pointcut 定义一个切入点表达式



@Before 前置通知，在执行目标对象方法之前执行，必须加上切入点表达式

@Before("execution(\* cn.itcast.aop\_xml.UserDao.save(..))")

@After 后置通知， 在目标方法之后执行 (始终执行)也必须加入切入点表达式

@After("execution(\* cn.itcast.aop\_xml.UserDao.save(..))")

@AfterReturning 返回后执行； 在执行目标方法正常结束后执行! (异常不执行)

@AfterThrowing 异常通知， 在执行目标方法出现异常的时候执行！

@Around 环绕通知

### 切入点表达式规则

|  |
| --- |
| @Aspect // 指定一个类为切面类！！  @Component  **public** **class** Aop {    /\* 1. 拦截指定的类的指定的方法  @Pointcut("execution(\* cn.itcast.f\_aop\_anno.UserDao.save(..))") \*/    /\* 2. 拦截指定的类的所有的方法  @Pointcut("execution(\* cn.itcast.f\_aop\_anno.UserDao.\*(..))") \*/    /\* 3. 拦截指定包下所有的类的所有方法  @Pointcut("execution(\* cn.itcast.f\_aop\_anno.\*.\*(..))") \*/    /\* 4. 拦截指定的包及其子包下所有类的所有方法！ 需求： cn包下所有类  @Pointcut("execution(\* cn..\*.\*(..))") \*/    /\* 5. 只拦截UserDao或OrderDao下的save方法  @Pointcut("execution(\* cn.itcast.f\_aop\_anno.UserDao.save()) || execution(\* cn.itcast.f\_aop\_anno.OrderDao.save())")  @Pointcut("execution(\* cn.itcast.f\_aop\_anno.UserDao.save()) or execution(\* cn.itcast.f\_aop\_anno.OrderDao.save())") \*/    //@Pointcut("execution(\* cn.itcast.f\_aop\_anno.UserDao.save()) && execution(\* cn.itcast.f\_aop\_anno.OrderDao.save())")  //@Pointcut("execution(\* cn.itcast.f\_aop\_anno.OrderDao.save()) and execution(\* cn.itcast.f\_aop\_anno.UserDao.save())")    /\* 6. 不拦截OrderDao.save方法\*/  // @Pointcut("!execution(\* cn.itcast.f\_aop\_anno.OrderDao.save())")    /\* 7. 拦截IOC容器的所有类的所有方法 \*/  //@Pointcut("execution(\* \*..\*.\*(..))")    // 8. 拦截容器中指定名称的bean  //@Pointcut("bean(userDao)")  // 拦截以Dao结尾的类  @Pointcut("bean(\*Dao)")  **public** **void** test(){}  // @Before 前置通知，在执行目标对象方法之前执行  @Before("test()") // 指定方法，会找方法上的@Pointcut 注解！  **public** **void** begin(){  System.*out*.println("---->开启事务");  }  // @After 后置通知， 在目标方法之后执行 (始终执行)  @After("test()")  **public** **void** commite(){  System.*out*.println("---->提交事务");  }  // @AfterReturning 返回后执行； 在执行目标方法正常结束后执行! (异常不执行)  @AfterReturning("test()")  **public** **void** afterReturning(){  System.*out*.println("---->返回后通知");  }  // @AfterThrowing 异常通知， 在执行目标方法出现异常的时候执行！  @AfterThrowing("test()")  **public** **void** afterThrowing(){  System.*out*.println("-----> 异常通知");  }  // @Around 环绕通知  @Around("test()")  **public** **void** around(ProceedingJoinPoint pjp) **throws** Throwable{  System.*out*.println("【环绕前】");  pjp.proceed(); // 执行目标方法  System.*out*.println("【环绕后】");  }  } |

### XML 配置方式实现

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>  <!-- 开启注解扫描 -->  <context:component-scan base-package=*"cn.itcast.g\_aop\_xml"*></context:component-scan>  <!-- 切面类 -->  <bean id=*"aop"* class=*"cn.itcast.g\_aop\_xml.Aop"*></bean>  <!-- aop 配置 -->  <aop:config>  <aop:pointcut expression=*"execution(\* cn.itcast.g\_aop\_xml.UserDao.\*(..))"* id=*"pt"*/>  <aop:aspect ref=*"aop"*>  <!-- 前置通知 -->  <aop:before method=*"begin"* pointcut-ref=*"pt"*/>  <!-- 后置通知 -->  <aop:after method=*"commite"* pointcut-ref=*"pt"*/>  <!-- 返回后通知-->  <aop:after-returning method=*"afterReturning"* pointcut-ref=*"pt"*/>  </aop:aspect>  </aop:config>  </beans> |

## 9. Spring 对Jdbc支持

如果项目没有用hibernate，需要写JDBC代码！

Spring提供了对JDBC模块的支持！

🡪 JdbcTemlate 工具类 （类似于QueryRunner）

引入Jar包:

spring-jdbc-3.2.5.RELEASE.jar 【对JDBC支持包】

spring-tx-3.2.5.RELEASE.jar 【事务依赖包】

步骤：

需求：保存一个用户！

1. JDBC保存

连接、执行命令的对象……

2. 优化

\* 连接对象

C3p0

\* jdbc操作操作步骤进行优化

JdbcTemplate

## 10 声明式事务管理

声明一个变量: int  *num;*

声明事务: (定义好一个事务，只要定义好，就可以直接使用！)

### 事务, 概念

* 事务是一组操作的执行单元，相对于数据库操作来讲，事务管理的是一组SQL指令，比如增加，修改，删除等，事务的一致性，要求，这个事务内的操作必须全部执行成功，如果在此过程种出现了差错，比如有一条SQL语句没有执行成功，那么这一组操作都将全部回滚
* 事务特性（ACID）
  + Atomic(原子性):要么都成功，要么都失败
  + Consistent(一致性):数据应该不被破坏
  + Isolate(隔离性):用户间操作不相混淆
  + Durable(持久性):永久保存

### 事务控制代码

JDBC:

Con.setAutoCommite(false); 开启事务; 设置事务手动提交（显示事务）

**切面类：DataSourceTransactionManager**

Hibernate:

Session.beginTransaction(); 开启事务

**切面类：HibernateTransactionManager**

### 事务控制方式

**编程式事务管理**

编写程序式的事务管理可以清楚的定义事务的边界，可以实现细粒度的事务控制，比如你可以通过程序代码来控制你的事务何时开始，何时结束等，与后面介绍的声明式事务管理相比，它可以实现细粒度的事务控制，例如jdbc,hibernate，spring中不提倡使用。

特点：

**1. 可以给任何行代码添加事务，事务控制很灵活、精细！**

**2. 每次都要写重复的事务控制代码，代码冗余、繁琐！**

**声明式事务管理**

如果你并不需要细粒度的事务控制，你可以使用声明式事务，在Spring中，你只需要在Spring配置文件中做一些配置，即可将操作纳入到事务管理中，解除了和代码的耦合， 这是对应用代码影响最小的选择，从这一点再次验证了Spring关于AOP的概念。当你不需要事务管理的时候，可以直接从Spring配置文件中移除该设置

特点：

**1 在XML中直接定义事务； 不想用直接移除**

**2. 解耦： 业务代码与事务控制代码分离！**

**开发的时候，只关注业务逻辑实现即可！重复事务代码就不用写！**

**3. Spring提供了事务切面类，用户只需要进行Aop配置!**

**4. 粗粒度的事务控制, 事务控制不够精细！**

## 11. Jdbc声明式事务管理, XML

需求：jdbc声明式事务控制！

步骤：

1. 库、表

day32/user

2. 搭建环境

\* 新建项目

\* 引入jar包

Spring-core 核心包

Spring-aop 切面编程

Spring-jdbc jdbc支持

3. 代码

IUserDao

UserDao

IuserService

UserService

App

Bean.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>  <!-- 1. C3p0连接池 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql:///day32"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  <property name=*"initialPoolSize"* value=*"3"*></property>  <property name=*"maxPoolSize"* value=*"6"*></property>  </bean>    <!-- 2. JdbcTemplate工具类 -->  <bean id=*"jdbcTempalte"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>    <!-- 3. dao 实例，注入JdbcTemplate -->  <bean id=*"userDao"* class=*"cn.itcast.a\_tx\_jdbc.UserDao"*>  <property name=*"jdbcTemplate"* ref=*"jdbcTempalte"*></property>  </bean>    <!-- 4. service 实例，注入dao -->  <bean id=*"userService"* class=*"cn.itcast.a\_tx\_jdbc.UserService"*>  <property name=*"userDao"* ref=*"userDao"*></property>  </bean>    <!--  Spring声明式事务控制  1) 配置事务管理器类（事务切面）  2) 配置事务通知 (拦截到方法后，如何控制事务：只读？读写？)  3) Aop 配置 = 切入点表单式，应用上述通知  -->  <bean id=*"txManage"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <tx:advice id=*"txAdvice"* transaction-manager=*"txManage"*>  <tx:attributes>  <tx:method name=*"get\*"* read-only=*"true"*/>  <tx:method name=*"find\*"* read-only=*"true"*/>  <tx:method name=*"\*"* read-only=*"false"*/>  </tx:attributes>  </tx:advice>  <aop:config>  <aop:pointcut expression=*"execution(\* cn.itcast.a\_tx\_jdbc.\*Service.\*(..))"* id=*"pt"*/>  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"*/>  </aop:config>    </beans> |

## 12. Jdbc声明式事务管理, 注解

步骤：

1. 配置事务管理器类

2. 开启事务注解

|  |
| --- |
| <bean id=*"txManage"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 开始事务注解 -->  <tx:annotation-driven transaction-manager=*"txManage"*/> |

3. 使用@Transactional注解

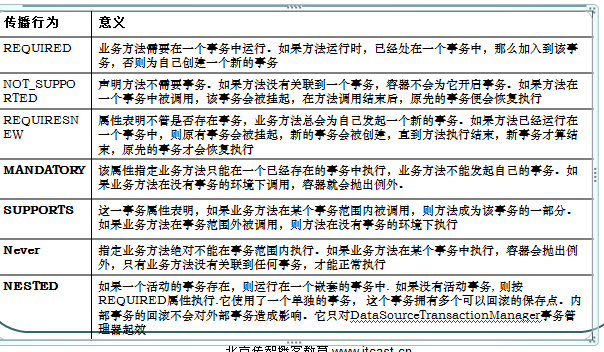
\* 定义到方法上，表示这个方法应用事务

\* 定义到类上， 表示这个类的所有方法都应用事务

\* 定义到父类上，表示执行父类方法时候应用事务

## 13. 事务属性

|  |
| --- |
| @Transactional(  readOnly=**false**, // 默认值false； 是否只读； 查询方法设置为true，其他都是false;  //rollbackFor=ArithmeticException.class // 遇到这个异常，回滚!  noRollbackFor=ArithmeticException.**class**,  timeout=-1, // 事务超时时间，设置为-1表示不设置超时时间！  isolation=Isolation.*DEFAULT*, // 隔离级别  propagation=Propagation.*REQUIRED* // 事务传播行为  )  **public** **void** save() {  userDao.save();  **int** i=1/0;  userDao.save();  } |



Propagation:

REQUIRED (默认)当前运行方法必须在事务环境下运行

如果当前执行的方法没有事务环境，则新开启一个事务！

如果当前执行的方法已经有事务环境，则会加入当前事务，没有开启新的事务！

REQUIRESNEW 当前运行方法必须在事务环境下运行

始终开启一个新的事务！

需求：

保存用户，记录日志！

Save: 【REQUIRED】

1. 先记录日志信息 (要求：始终记录！) 【REQUIRES\_NEW】

2. 再保存用户

SUPPORTS 有事务则支持，没有也可以!

## 14. Spring与Hibernate整合

关键点：

**1. SesionFactory对象创建，交给Spring**

**2. hibernate事务操作交给Spring声明式事务管理**

步骤：

1. 环境搭建

\*项目，引入jar文件

Hibernate 核心包

Spring

Core 核心相关（5个）

Orm 相关（3个）

Aop 4个

\*配置

Bean.xml

Hibernate.cfg.xml

2. 代码

entity / dao /service /app

|  |
| --- |
| <property name="hbm2ddl.auto">create</property>-->  <!-- 启动时删数据库中的表，然后创建，退出时自动删除所有表  <property name="hbm2ddl.auto">create-drop</property>-->  <!-- 自动修改，如果表结构与实体类不一致，那么就修改表使它们一致，数据会保留  <property name="hbm2ddl.auto">update</property>-->  <!-- 自动校验，如果表结构与实体类不一致，那么不做任何操作，报错  <property name="hbm2ddl.auto">validate</property>--> |

# SSH 整合案例

SSH整合：

Spring 与 Struts 整合，

可以把action交给spring创建！

Spring 与 Hibernate整合，

把hibernate的sessionFactory对象交给Spring创建！

事务，交给spring管理！

整合完整步骤：

1) 环境搭建

**①库、表**

|---day15

|---Table

|---employee

|---department

|  |  |
| --- | --- |
| employee | department |
| employeedepartmentemployee  create table employee(  id int primary key auto\_increment,  name varchar(20),  gender varchar(10),  title varchar(30),  email varchar(50),  salary double,  d\_id int,  constraint fk foreign key (d\_id) references department(id)  ) | create table department(  id int primary key auto\_increment,  name varchar(20)  ) |

**②导jar包**

|  |  |
| --- | --- |
| Struts2.3包  commons-beanutils-1.8.3.jar  commons-dbutils-1.2.jar  commons-fileupload-1.2.2.jar文件上传相关包  commons-io-2.0.1.jar 文件上传相关包  commons-lang3-3.1.jar 对lang包的扩展  commons-logging-1.1.1.jar  freemarker-2.3.19.jar 标签库模板包  javassist-3.11.0.GA.jar 字节码处理相关包  ognl-3.0.5.jar ognl表达式支持包  struts2-core-2.3.4.1.jar struts核心包  xwork-core-2.3.4.1.jar xwork核心包 | Spring3.2-aop包  aopalliance.jar  aspectjrt.jar  aspectjweaver.jar  spring-aop-3.2.5.RELEASE.jar  Spring3.2-core包  commons-logging-1.1.3.jar  spring-beans-3.2.5.RELEASE.jar  spring-context-3.2.5.RELEASE.jar  spring-core-3.2.5.RELEASE.jar  spring-expression-3.2.5.RELEASE.jar  Spring3.2-orm包  spring-jdbc-3.2.5.RELEASE.jar  spring-orm-3.2.5.RELEASE.jar  spring-tx-3.2.5.RELEASE.jar  Spring3.2-web包  spring-web-3.2.5.RELEASE.jar  struts2-spring-plugin-2.3.4.1.jar |
| Hibernate3.6包  antlr-2.7.6.jar  c3p0-0.9.1.jar  commons-collections-3.1.jar  dom4j-1.6.1.jar  hibernate3.jar  hibernate-jpa-2.0-api-1.0.0.Final.jar  javassist-3.12.0.GA.jar  jta-1.1.jar  mysql-connector-java-5.1.7-bin.jar  slf4j-api-1.6.1.jar |

**③配置web.xml**

|  |
| --- |
| <!-- 配置Struts2核心过滤器 -->  <filter>  <filter-name>struts2</filter-name>  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>struts2</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  <!-- SpringIOC容器配置 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:bean.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> |

**④用MyEclipse逆向工程,创建实体类employee.java和department.java**

1.右击工程包→MyEclipse→Add Hibernate Capabilities…

2.选择Hibernate3.3→去掉MyEclipse Libraries，勾上User Libraries→去掉框内所有选项→Next→Next→DB Driver 选择你自己定义的数据库（前提是你已经在MyEclipse数据库视图中有你自己添加的数据库）→Next→去勾→Finish

3.打开MyEclipse数据库视图→选择所有有关表→右击Hibernate Reverse Engineering

4.在Java src folder:选择你要逆向的工程项目下的src→勾上Create POJO…勾上Java DataObject→选择Create a Hibernate mapping file… 和勾上Update Hibernate configuration…→naxt

5.在Id Generator：选择nation→next→Finish

6.检查一下2个实体类的hbm.xml是否成功导入Hibernate（习惯）

**⑤编写Dao接口和Dao实现类**

|  |
| --- |
| Dao接口 |
| **public** **interface** InEmpDao {  **void** save(Employee emp);  **void** update(Employee emp);  **void** detele(Serializable id);  Employee findById(Serializable id);  List<Employee> findAll();  List<Employee> findAll(**int** index,**int** count);  Department findDepById(**int** id);  } |
| Dao实现类 |
| **public** **class** ImpEmpDao **implements** InEmpDao {  **private** SessionFactory sessionFactory;  **public** **void** setSessionFactory(SessionFactory sessionFactory) {  **this**.sessionFactory = sessionFactory;  }  **public** **void** detele(Serializable id) {  sessionFactory.getCurrentSession()  .createQuery("from Employee where id=?")  .setParameter(0, id)  .executeUpdate();  }  **public** List<Employee> findAll() {    **return** sessionFactory.getCurrentSession().createQuery("from Employee").list();  }  **public** List<Employee> findAll(**int** index, **int** count) {  Query q = sessionFactory.getCurrentSession().createQuery("from Employee");  q.setFirstResult(index);  q.setMaxResults(count);  **return** q.list();  }  **public** Employee findById(Serializable id) {    **return** (Employee)sessionFactory.getCurrentSession().get(Employee.**class**,id);  }  **public** **void** save(Employee emp) {    sessionFactory.getCurrentSession().save(emp);  }  **public** **void** update(Employee emp) {    **if**(findById(emp.getId())!=**null**){  sessionFactory.getCurrentSession().update(emp);  }  }  **public** Department findDepById(**int** id) {    **return** (Department)sessionFactory.getCurrentSession().get(Department.**class**,id);  }  } |

**⑥编写Service接口和Service实现类**

|  |
| --- |
| Service接口 |
| **public** **interface** InEmpDao {  **void** save(Employee emp);  **void** update(Employee emp);  **void** detele(Serializable id);  Employee findById(Serializable id);  List<Employee> findAll();  List<Employee> findAll(**int** index,**int** count);  Department findDepById(**int** id);  } |
| Service实现类 |
| **public** **class** ImpEmpService **implements** InEmpService {  **private** ImpEmpDao impEmpDao;    **public** **void** setImpEmpDao(ImpEmpDao impEmpDao) {  **this**.impEmpDao = impEmpDao;  }  **public** **void** detele(Serializable id) {  impEmpDao.detele(id);  }  **public** List<Employee> findAll() {    **return** impEmpDao.findAll();  }  **public** List<Employee> findAll(**int** index, **int** count) {  **return** impEmpDao.findAll(index, count);  }  **public** Employee findById(Serializable id) {    **return** impEmpDao.findById(id);  }  **public** **void** save(Employee emp) {    impEmpDao.save(emp);  }  **public** **void** update(Employee emp) {    impEmpDao.update(emp);  }  **public** Department findDepById(**int** id) {    **return** impEmpDao.findDepById(id);  }  } |

**⑦编写Action类**

|  |
| --- |
| **public** **class** EmpAction **extends** ActionSupport {  **private** InEmpService impEmpService;  **public** **void** setImpEmpService(InEmpService impEmpService) {  **this**.impEmpService = impEmpService;  }  **public** String save(){  Department department=**new** Department();  department=impEmpService.findDepById(3);  Employee employee = **new** Employee();  employee.setName("weiss");  employee.setEmail("weiss123@168com");  employee.setGender("female");  employee.setSalary(6666.0);  employee.setTitle("老师");  employee.setDepartment(department);  impEmpService.save(employee);  **return** *SUCCESS*;  }  } |

**⑧Spring配置**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>    <!-- 创建DataSource对象 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"jdbcUrl"* value=*"jdbc:mysql:///day15"*></property>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  <property name=*"initialPoolSize"* value=*"5"*></property>  <property name=*"maxPoolSize"* value=*"10"*></property>  </bean>    <!-- 创建SessionFactory对象 -->  <bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>  <!-- 注入链接池 -->  <property name=*"dataSource"* ref=*"dataSource"*></property>  <!-- 注入常用属性 -->  <property name=*"hibernateProperties"*>  <props>  <prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</prop>  <prop key=*"hibernate.show\_sql"*>true</prop>  </props>  </property>  <!-- 加载映射文件 -->  <property name=*"mappingLocations"*>  <list>  <value>classpath:cn/itcast/bean/\*.hbm.xml</value>  </list>  </property>  </bean>    <!-- 事务切面 -->  <bean id=*"txManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>  <property name=*"sessionFactory"* ref=*"sessionFactory"*></property>  </bean>  <!-- 事务通知 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"txManager"*>  <tx:attributes>  <tx:method name=*"get\*"* read-only=*"true"* propagation=*"SUPPORTS"* />  <tx:method name=*"find\*"* read-only=*"true"* propagation=*"SUPPORTS"* />  <tx:method name=*"\*"* read-only=*"false"* propagation=*"REQUIRED"*/>  </tx:attributes>  </tx:advice>    <!-- aop配置 -->  <aop:config>  <aop:pointcut expression=*"execution(\* cn.itcast.impEmpService.\*Service.\*(..))"* id=*"pt"*/>  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"*/>  </aop:config>    <!-- dao对象 -->  <bean id=*"impEmpDao"* class=*"cn.itcast.impEmpDao.ImpEmpDao"*>  <property name=*"sessionFactory"* ref=*"sessionFactory"*></property>  </bean>    <!-- service对象 -->  <bean id=*"impEmpService"* class=*"cn.itcast.impEmpService.ImpEmpService"*>  <property name=*"impEmpDao"* ref=*"impEmpDao"*></property>  </bean>    <!-- action对象 -->  <bean id=*"empAction"* class=*"cn.itcast.empAction.EmpAction"* scope=*"prototype"*>  <property name=*"impEmpService"* ref=*"impEmpService"*></property>  </bean>  </beans> |

**⑨Struts.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"  "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name=*"employee"* extends=*"struts-default"*>  <!-- action创建交给spring的ioc容器！ class:这里写短名！ -->  <action name=*"emp\_\*"* class=*"cn.itcast.empAction.EmpAction"* method=*"{1}"*>  <result>/index.jsp</result>  </action>  </package>  </struts> |

**⑩把项目加到Tomcat服务器运行**

**访问http://localhost:8080/sshProject/emp\_save.action**

# JavaScipt加强

## JavaScript对象

|  |  |
| --- | --- |
| **对象名称** | **对象说明** |
| Array | 数组 |
| Boolean | 布尔对象 |
| Date | 日期时间 |
| Math | 数学对象 |
| Number | 数值对象 |
| String | 字符串对象 |
| RegExp | 正则表达式对象 |
| Functions | 函数构造器 |
| Events | 事件 |
| 其它 | 如ActiveX、XMLHttpRequest等 |

## Browser对象

Browser Object Model，浏览器对象模型；它包括的对象：

|  |  |
| --- | --- |
| **对象名称** | **对象说明** |
| Window | Window 对象表示浏览器中打开的窗口  .onload加载完成事件  .showModalDialog(…)打开一个模式窗口，(Chrome)不能用  .open(…)打开一个窗口  .close()关闭窗口  .setInterval(…)在某个时间间隔内刷新 |
| Navigator | Navigator 对象包含有关浏览器的信息；如：navigator.appVersion 属性可返回浏览器的平台和版本信息  .appVersion 获取浏览器运行环境 |
| Screen | Screen 对象包含有关客户端显示屏幕的信息 |
| History | History 对象包含用户（在浏览器窗口中）访问过的 URL；如：history.back()返回上一个页面  .back 后退  .forward前进  .go去到指定页面 |
| Location | Location 对象包含有关当前 URL 的信息；如：location.href返回当前窗口的URL  .href |

## HTML DOM对象

Document Object Model，文档对象模型；DOM 把一个文档以树的形式表示；在DOM中，每个部分都是节点：文档本身是文档节点 所有 HTML 元素是元素节点 所有 HTML 属性是属性节点 HTML 元素内的文本是文本节点 注释是注释节点。

|  |  |
| --- | --- |
| **对象名称** | **对象说明** |
| Document | Document 对象；每个载入浏览器的 HTML 文档都会成为 Document 对象 |
| Element | Element 对象表示 HTML 元素。Element 对象可以拥有类型为元素节点、文本节点、注释节点的子节点 |
| Attribute | Attribute 对象表示 HTML 属性。HTML 属性始终属于 HTML 元素 |

## AJax方法

|  |  |
| --- | --- |
| **方法** | **描述** |
| open(method,url,async) | 规定请求的类型、URL 以及是否异步处理请求。  method：请求的类型；GET 或 POST  url：文件在服务器上的位置  async：true（异步）或 false（同步） |
| send(string) | 将请求发送到服务器。  string：仅用于 POST 请求；当为GET时可不填string对应值或者null |
| setRequestHeader(header,value) | 向请求添加 HTTP 头。  header: 规定头的名称  value: 规定头的值  **注意：**POST请求时如需提交在send方法中提交表单值；一般设置：xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded"); |
| abort() | 停止当前请求 |
| getAllResponseHeaders() | 把Http请求的所有响应首部作为键/值对返回 |
| getResponseHeader(header) | 返回指定首部属性的值 |

## AJax属性

|  |  |
| --- | --- |
| **属性** | **描述** |
| onreadystatechange | 存储函数（或函数名），每当 readyState 属性改变时，就会调用该函数。 |
| readyState | 存有 XMLHttpRequest 的状态。从 0 到 4 发生变化。  0: 请求未初始化。 还没有调用 open 方法  1: 服务器连接已建立。 open 方法已被调用，但 send 方法还没有被调用  2: 请求已接收。send 已被调用。请求已经开始  3: 请求处理中。服务器正在发送响应  4: 请求已完成，且响应已就绪 |
| responseText | 服务器的响应文本内容；如纯文本、html等 |
| responseXML | 服务器的响应内容，xml格式，xml dom对象；如可使用xmlDocument.getElementsByTagName("root") |
| status | 服务器发送的每一个响应的首部信息中的一个属性  常用状态码及其含义：  404 没找到页面(not found)  403 禁止访问(forbidden)  500 内部服务器出错(internal service error)  200 一切正常(ok)  304 没有被修改(not modified) |
| statusText | Http的状态响应文本（OK或Not Found等） |

异步刷新

|  |
| --- |
| 异步获取div时间  window.onload=**function**(){  document.getElementsByTagName("button")[0].onclick=**function**(){  //创建XMLHttpRequest对象  **var** xmlHttp = createXMLHttpRequest();  //设置回调函数  xmlHttp.onreadystatechange=**function**(){  //判断readyState和status的状态  **if**(xmlHttp.readyState==4&&xmlHttp.status==200){  document.getElementById("div1").innerHTML=xmlHttp.responseText;  }  }  //创建链接(提交表单方式，地址，是否异步)  xmlHttp.open("get","${pageContext.request.contextPath}/time",**true**);  //发送链接  xmlHttp.send(**null**);  }  }  //创建XMLHttpRequest，旧IE(6以下)创建ActiveXObject  **function** createXMLHttpRequest(){  **var** xmlHttp;  **if**(window.XMLHttpRequest){  xmlHttp=**new** XMLHttpRequest();  }**else**{  xmlHttp=**new** ActiveXObject("Microsoft.XMLHTTP");  }  **return** xmlHttp;  } |

二级联动

|  |  |
| --- | --- |
| <div id=*"div3"* style="height:*50px*;width:*400px*;border: *#000000 solid 1px*;">  <div style="padding: *15px*">  <select id=*"country"* onchange=*"selectCountry()"*>  <option>-请选择-</option>  <option value=*"cn"*>中国</option>  <option value=*"us"*>美国</option>  </select>  &nbsp&nbsp  <select id=*"city"*>  <option>-请选择-</option>  </select>  </div>  </div> |  |
| //创建XMLHttpRequest，旧IE(6以下)创建ActiveXObject  **function** createXMLHttpRequest(){  **var** xmlHttp;  **if**(window.XMLHttpRequest){  xmlHttp=**new** XMLHttpRequest();  }**else**{  xmlHttp=**new** ActiveXObject("Microsoft.XMLHTTP");  }  **return** xmlHttp;  }  **function** selectCountry(){  //获取下拉列表的值  **var** country=document.getElementById("country").value;  //创建XMLHttpRequest对象  **var** xmlHttp = createXMLHttpRequest();  //设置回调函数  xmlHttp.onreadystatechange=**function**(){  //判断readyState和status的状态  **if**(xmlHttp.readyState==4&&xmlHttp.status==200){  //获取响应文本  **var** responseText=xmlHttp.responseText;  // 转换为json格式  **var** json=eval("("+responseText+")");  **var** cityTag=document.getElementById("city");  // 先清空  cityTag.length =0;  **for**(**var** i=0 ; i<json.cities.length ; i++){  **var** optionTag=document.createElement("option");  optionTag.text=json.cities[i].city;  cityTag.appendChild(optionTag);  }  }  }  //创建链接(提交表单方式，地址，是否异步)  xmlHttp.open("post","${pageContext.request.contextPath}/time",**true**);  // 如果post提交，需要设置表单的类型  xmlHttp.setRequestHeader("content-type","application/x-www-form-urlencoded");  //发送链接  xmlHttp.send("country="+country);  } | |
| request.setCharacterEncoding("UTF-8");  response.setContentType("text/xml;charset=UTF-8");  String country=request.getParameter("country");  System.*out*.println(country);  StringBuffer sb=**new** StringBuffer();  **if**(country.equals("cn")){  sb.append("{\"cities\":[{\"city\":\"北京\"},{\"city\":\"广州\"},{\"city\":\"深圳\"},{\"city\":\"上海\"}]}");  }**else**{  sb.append("{\"cities\":[{\"city\":\"纽约\"},{\"city\":\"华盛顿\"},{\"city\":\"阿拉斯加\"},{\"city\":\"旧金山\"}]}");  }  response.getWriter().write(sb.toString()); | |

# jQuery

**jQuery是一个优秀的Javascript库**

一个兼容多浏览器

核心理念是write less,do more(写得更少,做得更多)

轻量级

出色的 DOM 操作的封装

强大的选择器

1、 jQuery对象转为Dom对象

var $div = $("#divID"); var divElement = $div[0];

或者var $div = $("#divID"); var divElement = $div.get(0);

2、 Dom对象转为jQuery对象；

var $div = $(divElement);

在使用jQuery前必须先导入核心js包jquery-x.x.js

## jQuery的使用

**基本选择**

|  |  |
| --- | --- |
| **用法** | **描述** |
| $(“#elementID”) | #表示根据id查找； |
| $(“.className”) | .表示根据class样式名称查找 |
| $(“input”) | 查找所有input标签 |

**基本取/设值**

|  |  |
| --- | --- |
| **用法** | **描述** |
| $(“#elementID”).val()  $(“#elementID”).val(“新值”) | 分别表示取值和设置值 |
| $(“#elementID”).html()  $(“#elementID”).html(“新值”) | 分别表示获取标签内部的html文本内容和设置html文本内容。类似innerHTML |
| $(“#elementID”).text()  $(“#elementID”).text(“新值”) | 分别表示获取标签内部的纯文本内容和设置纯文本内容。 |
| $(“#elementID”).attr(“name”)  $(“#elementID”).attr(“name”,”itcast”) | attr为获取对应属性的值或设置对应属性的值。 |
| $(“#elementID”).prop("checked")  $(“#elementID”).prop("checked", true); | 如果遇上具有 true 和 false 两个属性的属性，如 checked, selected 或者 disabled 则使用prop() |

**基本访问与工具**

|  |  |
| --- | --- |
| **用法** | **描述** |
| empty() | 删除元素中所有的子节点。 |
| each() | 以每一个匹配的元素作为上下文来执行一个函数；每次执行传递进来的函数时，函数中的this关键字都指向一个不同的DOM元素。  $("td").each(**function**(index){ alert(index+**this**.innerHTML); });  $("td").each(**function**(index,object){ alert(index+object.innerHTML); }); |
| jQuery.each() | 通用例遍方法，可用于例遍对象和数组  $.each($("td"),**function**(index){ alert(index+**this**.innerHTML); });  $.each($("td"),**function**(index,object){ alert(index+object.innerHTML); }); |
| jQuery.trim() | 去掉字符串起始和结尾的空格。  $.trim(" hello, how are you? ");去掉字符串起始和结尾的空格。 |
| focus() | 定焦或当元素获得焦点时，触发 focus 事件 |

**jQuery.ajax**

|  |  |
| --- | --- |
| **参数** | **描述** |
| **url** | 发送请求的地址 |
| **data** | 发送到服务器的数据。将自动转换为请求字符串格式。 |
| **type** | 请求方式 ("POST" 或 "GET")， 默认为 "GET" |
| **dataType** | 预期服务器返回的数据类型。可用值: xml,html,script,json,jsonp,text |
| **success** | 请求成功后的回调函数。参数：由服务器返回，并根据dataType参数进行处理后的数据 |
| **error** | 默认: 自动判断 (xml 或 html)) 请求失败时调用此函数。如果发生了错误，错误信息（第二个参数）除了得到null之外，还可能是"timeout", "error", "notmodified" 和 "parsererror" |
| **async** | 默认设置下，所有请求均为异步请求。如果需要发送同步请求，请将此选项设置为 false。注意，同步请求将锁住浏览器，用户其它操作必须等待请求完成才可以执行 |

列子

|  |  |
| --- | --- |
| <div id=*"div3"* style="height:*50px*;width:*400px*;border: *#000000 solid 1px*;">  <div style="padding: *15px*">  <select id=*"country"* onchange=*"selectCountry()"*>  <option>-请选择-</option>  <option value=*"cn"*>中国</option>  <option value=*"us"*>美国</option>  </select>  &nbsp&nbsp  <select id=*"city"*>  <option>-请选择-</option>  </select>  </div>  </div> |  |
| $(document).ready(**function**(){  $("#country").change(**function**(){  **var** country=$(**this**).val(); //获取国家下拉列表的值  $.ajax({  type:"post", //请求方式  url:"${pageContext.request.contextPath}/city",//访问地址  data:{"country":country}, //参数  dataType:"json", //返回数据类型  success:**function**(returnData){ //回调函数（返回数据）  **if**(returnData!=**undefined**&&returnData!=**null**&&returnData!=""){  **var** cityTag=$("#city");  cityTag.empty(); //清空城市下拉列表  //遍历返回数据  $.each(returnData.cities,**function**(index,object){  cityTag.append("<option>"+object.city+"</option>");  });  }**else**{  alert("加载失败！");  }  }  });  });  }); | |
| request.setCharacterEncoding("UTF-8");  response.setContentType("text/xml;charset=UTF-8");  String country=request.getParameter("country");  System.*out*.println(country);  StringBuffer sb=**new** StringBuffer();  **if**(country.equals("cn")){  sb.append("{\"cities\":[{\"city\":\"北京\"},{\"city\":\"广州\"},{\"city\":\"深圳\"},{\"city\":\"上海\"}]}");  }**else**{  sb.append("{\"cities\":[{\"city\":\"纽约\"},{\"city\":\"华盛顿\"},{\"city\":\"阿拉斯加\"},{\"city\":\"旧金山\"}]}");  }  response.getWriter().write(sb.toString()); | |

**jQuery.get**

$.get(url,[data],[function(returnData)],[dataType])

**jQuery.post**

$.post(url，[data],[function(returnData)],[dataType])

**jQuery.getJSON**

$.getJSON(url, [data], [function(json)])

通过远程 HTTP GET 请求载入json信息

**jQuery.getScript**

$.getScript(url, [function()])

通过远程 HTTP GET 请求载入并执行一个 JavaScript 文件。

**load**

load(url, [data], [function()])

载入远程 HTML 文件代码并插入至 DOM 中。默认使用 GET 方式 - 传递附加参数时自动转换为 POST 方式

## jQuery 选择器

console.log()方法可以在谷歌浏览器的开发者模式下的console输出

**基本选择器**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| #id | 根据给定的ID匹配一个元素 | 单个元素组成的集合 |
| element | 根据给定的元素名匹配所有元素 | 元素集合 |
| .class | 根据给定的类匹配元素 | 元素集合 |
| \* | 匹配所有元素 | 元素集合 |
| seletor1,seletor2… | 将每一个选择器匹配到的元素合并后一起返回； | 元素集合 |

**层级选择器**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| ancestor descendant | 获取后代（子孙）元素（关键是空格） | 元素集合 |
| parent > child | 获取子元素 | 元素集合 |
| prev + next | 获取下一个相邻元素（不能有相隔） | 元素集合 |
| prev ~ siblings | 获取后面的所有兄弟元素 | 元素集合 |
| Prev. siblings(…) | 获取所有兄弟元素 | 元素集合 |

**基本过滤**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| :first | 选取第一个元素$('li:first'); | 单元素 |
| :last | 选取最后一个元素$('li:last'); | 单元素 |
| :not(selector) | 去除所有与给定选择器匹配的元素。  $("input:not(:checked)")；查找所有未选中的 input 元素 | 元素集合 |
| :even | 选取索引号为偶数的元素，索引号从0开始$("tr:even") | 元素集合 |
| :odd | 选取索引号为奇数的元素，索引号从0开始$("tr:odd”) | 元素集合 |
| :eq(index) | 选取索引号为index的元素，索引号从0开始$("tr:eq(1)") | 单元素 |
| :gt(index) | 选取索引号大于index的元素，索引号从0开始$("tr:gt(1)") | 元素集合 |
| :lt(index) | 选取索引号小于index的元素，索引号从0开始$("tr:lt(1)") | 元素集合 |
| :header | 选取如 h1, h2, h3之类的标题元素$(":header").  css("background", "#EEE") | 元素集合 |
| :animated | 选取所有正在执行动画效果的元素  $("div:animated") | 元素集合 |

**内容过滤**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| :contains(text) | 选取包含给定文本的元素  $("div:contains('John')") | 元素集合 |
| :empty | 选取所有不包含子元素或者文本的空元素$("td:empty") | 元素集合 |
| :has(selector) | 选取含有选择器所匹配的元素的元素  $("div:has(p)").addClass("text");给所有包含 p 元素的 div 元素添加一个 text 样式 | 元素集合 |
| :parent | 选取父元素$("td:parent") | 元素集合 |

**属性过滤**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| [attribute] | 选取包含给定属性的元素 $("div[id]")  查找所有含有 id 属性的 div 元素 | 元素集合 |
| [attribute=value] | 选取给定的属性是某个特定值的元素 $("input[name='newsletter']") 查找所有 name 属性是 newsletter 的 input 元素 | 元素集合 |
| [attribute!=value] | 选取所有不含有指定的属性，或者属性不等于特定值的元素 $("input[name!='newsletter']") 查找所有 name 属性不是 newsletter 的 input 元素 | 元素集合 |
| [attribute^=value] | 选取给定的属性是以某些值开始的元素  $("input[name^='news']") 查找所有 name 以 'news' 开始的 input 元素 | 元素集合 |
| [attribute$=value] | 选取给定的属性是以某些值结尾的元素  $("input[name$='letter']") 查找所有 name 以 'letter' 结尾的 input 元素 | 元素集合 |
| [attribute\*=value] | 选取给定的属性是以包含某些值的元素  $("input[name\*='man']") 查找所有 name 包含 'man' 的 input 元素 | 元素集合 |
| [attrSel1][attrSel2][attrSelN] | 复合属性选择器，需要同时满足多个条件时使用  $("input[id][name$='man']") 找到所有含有 id 属性，并且它的 name 属性是以 man 结尾的 | 元素集合 |

**子元素过滤**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| :nth-child | 选取父元素下的第N个子或奇偶元素（index序号从1开始）  $("ul li:nth-child(2)") 在每个 ul 查找第 2 个li | 元素集合 |
| :first-child | 选取父元素的第一个子元素  $("ul li:first-child") 在每个 ul 中查找第一个 li | 元素集合 |
| :last-child | 选取父元素中的最后一个子元素 $("ul li:last-child") 在每个 ul 中查找最后一个 li | 元素集合 |
| :only-child | 如果某个元素是父元素中唯一的子元素，那将会被匹配；如果父元素中含有其他元素，那将不会被匹配。 $("ul li:only-child") 在 ul 中查找是唯一子元素的 li | 元素集合 |

**表单过滤**

控制select选取编译原理$("select:first").val(["编译原理"])

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| :input | 匹配所有 input, textarea, select 和 button 元素 $(":input") 查找所有的input元素。 | 元素集合 |
| :text | 匹配所有的单行文本框 $(":text") | 元素集合 |
| :password | 匹配所有密码框 $(":password") | 元素集合 |
| :radio | 单选按钮选择中国$(":radio").val(["cn"]) | 元素集合 |
| :checkbox | 匹配所有复选框 $(":checkbox") | 元素集合 |
| :submit | 匹配所有提交按钮 $(":submit") | 元素集合 |
| :image | 匹配所有图像域 $(":image") | 元素集合 |
| :reset | 匹配所有重置按钮 $(":reset") | 元素集合 |
| :button | 匹配所有按钮<button></button>，$(":button") | 元素集合 |
| :file | 匹配所有文件域 $(":file") | 元素集合 |
| :hidden | 匹配所有不可见元素（display:none），或者type为hidden的元素 $("tr:hidden").show() | 元素集合 |

**表单对象属性过滤**

|  |  |  |
| --- | --- | --- |
| **选择器** | **描述** | **返回** |
| :enabled | 选取所有可用元素$("input:enabled") | 元素集合 |
| :disabled | 选取所有不可用元素$("input:disabled") | 元素集合 |
| :checked | 选取所有被选中的元素（单、复选框）$("input:checked") | 元素集合 |
| :selected | 选取所有被选中的选项元素（下拉列表  $("select option:selected")查找所有选中的选项元素  checkbox:not(:checked)多选按钮没有被选取的元素 | 元素集合 |

**文档处理（**content文本fn jQuery表达式ele元素、标签html代码expr标签名expr jQuery表达式**）**

|  |  |
| --- | --- |
| **属性/方法** | **描述** |
| append(content|fn) | 向每个匹配的元素内部追加内容。$("p").append("<b>Hello</b>");向所有段落中追加一些HTML标记。 |
| appendTo(content) | 把所有匹配的元素追加到另一个指定的元素中。使用这个方法是颠倒了常规的$(A).appendTo(B)的操作，即不是把B追加到A中，而是把A追加到B中。$("p").appendTo("div");把所有段落追加到div元素中 |
| prepend(content|fn) | 向每个匹配的元素内部前置内容。$("p").prepend("<b>Hello</b>");向所有段落中前置一些HTML标记代码。 |
| prependTo(content) | 把所有匹配的元素前置到另一个、指定的元素元素集合中。使用这个方法是颠倒了常规的$(A).prepend(B)的操作，即不是把B前置到A中，而是把A前置到B中。$("p").prependTo("#foo"); 把所有段落追加到ID值为foo的元素中。 |
| after(content|fn) | 在每个匹配的元素之后插入内容。$("p").after("<b>Hello</b>");在所有段落之后插入一些HTML标记代码。 |
| before(content|fn) | 在每个匹配的元素之前插入内容。$("p").before("<b>Hello</b>");在所有段落之前插入一些HTML标记代码。 |
| insertAfter(content) | 把所有匹配的元素插入到另一个、指定的元素元素集合的后面。把A插入到B后面。$(A).insertAfter(B) |
| insertBefore(content) | 把所有匹配的元素插入到另一个、指定的元素元素集合的前面。把A插入到B前面。$(A).insertBefore(B) |
| wrap(html|ele|fn) | 把所有匹配的元素用其他元素的结构化标记包裹起来。  $("p").wrap("<div class='wrap'></div>"); div包裹了p标签 |
| unwrap() | 这个方法将移出元素的父元素。这能快速取消 .wrap()方法的效果。  $("p").unwrap(); 把所有段落的父元素直接移除 |
| wrapAll(html|ele) | 将所有匹配的元素用单个元素包裹起来。$("p").wrapAll("<div></div>");用一个生成的div将所有段落包裹起来 |
| wrapInner(html|ele|fn) | 将每一个匹配的元素的子内容(包括文本节点)用一个HTML结构包裹起来。$("p").wrapInner("<b></b>");把所有段落内的每个子内容加粗 |
| replaceWith(content|fn) | 将所有匹配的元素替换成指定的HTML或DOM元素。  $("p").replaceWith("<b>Paragraph. </b>");用b标签替换掉p标签 |
| replaceAll(selector) | 用匹配的元素替换掉所有 selector匹配到的元素。  $("<b>Paragraph. </b>").replaceAll("p");把b标签替换掉p标签（包括内容） |
| empty() | 删除、清空匹配的元素集合中所有的子节点。$("p").empty();把所有段落的子元素（包括文本节点）删除 |
| remove([expr]) | 从DOM中删除所有匹配的元素。$("p").remove();从DOM中把所有段落删除 |
| clone([Even[,deepEven]]) | 克隆匹配的DOM元素并且选中这些克隆的副本。（传入参数布尔值（true 或者 false）指示事件处理函数是否会被复制。）  $("b").clone()克隆所有b元素（返回这些克隆的副本）， |

**筛选**

|  |  |
| --- | --- |
| **属性/方法** | **描述** |
| first() | first() 获取第一个元素。$('li').first() |
| last() | last() 获取最后个元素。$('li').last() |
| is(expr|obj|ele|fn) | 根据选择器、DOM元素或 jQuery 对象来检测匹配元素集合，如果其中至少有一个元素符合这个给定的表达式就返回true。如果没有元素符合，或者表达式无效，都返回false。  $("input[type='checkbox']").parent().is("form");由于input元素的父元素是一个表单元素，所以返回true。 |
| has(expr|ele) | 保留包含特定后代的元素，去掉那些不含有指定后代的元素。.has()方法将会从给定的jQuery对象中重新创建一组匹配的对象。  //$("li span").addClass("cRed");  $("li").has("span").addClass("cRed");给li下的span文字设成红色 |
| find(e|o|e) | 搜索所有与指定表达式匹配的元素。  $("p").find("span");从p标签开始，进一步搜索下面的span元素。 |
| parent([expr]) | 取得一个包含着所有匹配元素的唯一父元素的元素集合。  $("p").parent(".selected");查找段落的父元素中每个类名为selected的父元素。 |
| siblings([expr]) | 取得一个包含匹配的元素集合中每一个元素的所有唯一同辈元素的元素集合。  $("div").siblings();找到每个div的所有同辈元素。 |

**样式**

|  |  |
| --- | --- |
| **属性/方法** | **描述** |
| addClass(class|fn) | 为每个匹配的元素添加指定的类名 $("p").addClass("selected"); 为匹配的元素加上 'selected' 类 |
| removeClass([class|fn]) | 从所有匹配的元素中删除全部或者指定的类 $("p").removeClass("selected"); 从匹配的元素中删除 'selected' 类 |
| toggleClass(class|fn[,sw]) | 如果存在（不存在）就删除（添加）一个类。$("p").toggleClass("selected"); 为匹配的元素切换 'selected' 类 |
| hasClass(class) | 检查元素是否含有某个特定的类，如果有，则返回true。  $("div").hasClass("protected") |
| css(name|pro|[,val|fn]) | 访问匹配元素的样式属性 $("p").css("color") 取得第一个段落的color样式属性的值。$("p").css("color","red"); 将所有段落字体设为红色$("p").css({"color":"red","font-weight":"bold"})  如果原来的样式已经存在，在使用就会覆盖该样式 |
| height/width([val|fn]) | 取得匹配元素当前计算的高/宽度值（px）  $("p").height();$("p").width(); |
| offset([coordinates]) | 获取匹配元素在当前视口的相对偏移。返回的对象包含两个整型属性：top 和 left，以像素计。此方法只对可见元素有效  var p = $("p:last");  var offset = p.offset();  p.html( "left: " + offset.left + ", top: " + offset.top ); |

**事件**

|  |  |
| --- | --- |
| **属性/方法** | **描述** |
| on(eve,[sel],[data],fn)1.7+ | 在选择元素上绑定一个或多个事件的事件处理函数  $("p").on("click", function(){  alert( $(this).text() );  }); |
| off(eve,[sel],[fn])1.7+ | 在选择元素上移除一个或多个事件的事件处理函数。off() 方法移除用.on()绑定的事件处理程序  $("p").off( "click", "\*\*" ) |
| bind(type,[data],fn) | 为每个匹配元素的特定事件绑定事件处理函数。  $(document).bind('mousemove',function(e){  $("#log").text("e.pageX: " + e.pageX + ", e.pageY: " + e.pageY);  }); |
| unbind(t,[d|f]) | bind()的反向操作，从每一个匹配的元素中删除绑定的事件。如果没有参数，则删除所有绑定的事件。  $("p").unbind( "click" ) |
| one(type,[data],fn) | 为每一个匹配元素的特定事件（像click）绑定一个一次性的事件处理函数。  $("p").one("click", function(){  alert( $(this).text() );  }); |
| hover([over,]out) | 一个模仿悬停事件（鼠标移动到一个对象上面及移出这个对象）的方法。这是一个自定义的方法，它为频繁使用的任务提供了一种“保持在其中”的状态。当鼠标移动到一个匹配的元素上面时，会触发指定的第一个函数。当鼠标移出这个元素时，会触发指定的第二个函数。  $("td").hover(  function () {  $(this).addClass("hover");  },  function () {  $(this).removeClass("hover");  }  ); |

**效果**

|  |  |
| --- | --- |
| **属性/方法** | **描述** |
| show([s,[e],[fn]]) | 显示隐藏的匹配元素。  $("p").show() 显示所有段落 |
| hide([s,[e],[fn]]) | 隐藏显示的元素。  $("p").hide() 隐藏所有段落 |
| slideDown([s],[e],[fn]) | 通过高度变化（向下增大）来动态地显示所有匹配的元素，在显示完成后可选地触发一个回调函数。  $("p").slideDown("slow"); 用600毫秒缓慢的将段落滑下 |
| slideUp([s,[e],[fn]]) | 通过高度变化（向上减小）来动态地隐藏所有匹配的元素，在隐藏完成后可选地触发一个回调函数。  $("p").slideUp("slow");用600毫秒缓慢的将段落滑上 |
| slideToggle([s],[e],[fn]) | 通过高度变化来切换所有匹配元素的可见性，并在切换完成后可选地触发一个回调函数。  $("p").slideToggle("slow");用600毫秒缓慢的将段落滑上或滑下 |
| fadeIn([s],[e],[fn]) | 通过不透明度的变化来实现所有匹配元素的淡入效果，并在动画完成后可选地触发一个回调函数。$("p").fadeIn("slow");用600毫秒缓慢的将段落淡入 |
| fadeOut([s],[e],[fn]) | 通过不透明度的变化来实现所有匹配元素的淡出效果，并在动画完成后可选地触发一个回调函数。  $("p").fadeOut("slow");用600毫秒缓慢的将段落淡出 |
| fadeTo([[s],o,[e],[fn]]) | 把所有匹配元素的不透明度以渐进方式调整到指定的不透明度，并在动画完成后可选地触发一个回调函数。  $("p").fadeTo("slow", 0.66);用600毫秒缓慢的将段落的透明度调整到0.66，大约2/3的可见度 |
| fadeToggle([s,[e],[fn]]) | 通过不透明度的变化来开关所有匹配元素的淡入和淡出效果，并在动画完成后可选地触发一个回调函数。  $("p").fadeToggle("slow");用600毫秒缓慢的将段落淡入/出 |
| animate(p,[s],[e],[fn])1.8\* | 用于创建自定义动画的函数。 |

# oracle

链接数据库sqlplus scott/tiger

## jdbc 连接

1、 在安装目录下找到oracle的驱动包；

如下路径可以找到oracle的驱动包：

c:\oracle11g\product\11.2.0\dbhome\_1\jdbc\lib 复制ojdbc6.jar到项目中进行连接测试；

2、 新建java项目测试连接；



## 显示设置

*-- 设置每行显示的最长字符数*

set linesize 120

*-- 设置一页显示的行数*

set pagesize 20

*-- 设置是否显示一页的记录数*

set feedback on/off

*-- 打开或取消oracle自带的输出方法dbms\_output，并输出内容*

set serveroutput on/off

*-- 格式化列的内容：将列名对应的列的值格式化为四位数值长度*

col 表中对应的列名 for 9999

column 表中对应的列名 format 9999

【示例】

*-- 表明将empno列名对应的列值格式为4位长度的数值型*

col empno for 9999

*-- 格式化列的内容：将列名对应的列的值格式化为10位字母长度*

col 表中对应的列名 for a10

【示例】

*-- 表明将ename列名对应的列值格式为10位长度的字符型*

col ename for a10

## 常用命令

|  |  |
| --- | --- |
| **命令** | **说明** |
| show all | 查看系统所有变量值 |
| show user | 显示当前连接用户 |
| show error | 显示错误 |
| desc 表名 | 显示表的结构；如：desc emp |
| /\* \*/  -- | 多行注释  单行注释 |
| / | 执行缓冲区中的语句 |
| ed | 打开默认编辑器，windows系统中默认是notepad.exe，把缓冲区中最后一条sql语句调入afiedt.buf文件中进行编辑（如果提示没有afiedt.buf请使用管理员身份打开slq plus）；常用于语句比较长需要修改时。 |
| spool 文件地址  spool 文件地址 append  spool off | 假脱机命令；将命令行的内容（从设置后开始的命令行内容）记录到文本。添加append的意思是在原有的文本内容上追加后续的命令行的内容；需要注意的是所有的这些内容都将在spool off之后才记录。如：  spool d:\itcast\itcast.txt  spool d:\itcast\test.sql append  spool off |
| clear screen 或者 host cls | 清屏 |
| exit | 退出sql plus |
| conn username/password | 切换用户 |

## 1.4表空间

**创建表空间**

|  |
| --- |
| 【语法】  create tablespace 表空间名  datafile '数据文件路径' size 大小  [autoextend on] [next 大小]  [maxsize 大小];  【说明】[]里面内容可选项；数据文件路径中若包含目录需要先创建  size为初始表空间大小，单位为k或者m  autoextend on 是否自动扩展  next为文件满了后扩展大小  maxsize为文件最大大小，值为数值或unlimited（表示不限大小）  【示例】  create tablespace itcast\_ts  datafile 'd:\oracle\_data\itcast01.dbf' size 10m  autoextend on; |

**查询表空间**

|  |
| --- |
| *--管理员角色查看表空间*  select file\_name,tablespace\_name,bytes,autoextensible  from dba\_data\_files  一定要大写  where tablespace\_name='itcast\_ts'; |

**修改表空间**

一定要大写

|  |
| --- |
| 【语法】  alter tablespace 表空间名  add datafile '文件路径' size 大小  [autoextend on] [next 大小]  [maxsize 大小];  【示例】  alter tablespace itcast\_ts  add datafile 'd:\oracle\_data\itcast02.dbf' size 5m  autoextend on;  一定要大写 |

**删除表空间**

|  |
| --- |
| 【语法】  drop tablespace 表空间名;  drop tablespace 表空间名 including contents and datafiles;  【说明】  第一个删除语句只删除表空间；第二个删除语句则删除表空间及数据文件  【示例】  drop tablespace itcast\_ts;  drop tablespace itcast\_ts including contents and datafiles; |

## 1.5常见创建用户命令

查询系统用户 select \* from all\_users;

锁定hr用户（不允许登录） alter user hr account islocked;

解锁hr用户 alter user hr account unlock;

创建itcast用户名，密码为itcast，使用的表空间为itcast\_ts，用户存放在temp目录下

create user itcast identified by itcast

default tablespace itcast\_ts

temporary tablespace temp;

修改用户密码 alter user itcast identified by itcast1234;

删除用户 drop user itcast cascade;

给用户授权 ，授予connect和resource

grant connect,resource to itcast;

## 1.6权限的授予与撤销

授予connect和resource两个角色给

itcast grant connect, resource to itcast;

查看resource角色下的权

select \* from dba\_sys\_privs where grantee='resource'

允许用户查看、更新 emp 表中的记录

grant select,update on scott.emp to itcast;

查看当前用户的系统权限

select \* from user\_sys\_privs;

查看当前用户的对象权限

select \* from user\_tab\_privs;

查看当前用户的所有角色

select \* from user\_role\_privs;

撤销connect和resource两个角色

revoke connect, resource from itcast;

撤销用户查看、更新 emp 表中的记录的操作

revoke select,update on scott.emp from itcast;

从现有的表创建表及复制其数据

create table emp as select empno,ename from scott.emp;

向表中添加新字段

alter table t\_student add (s\_age number(3),s\_address varchar2(20));

修改表中字段

alter table t\_student modify(s\_name varchar2(50),s\_address varchar2(100));

删除表中字段

alter table t\_student drop(s\_age,s\_address);

修改表字段名称

alter table t\_student rename column s\_id to s\_no;

## 1.7删除表

|  |
| --- |
| 【语法1】  --删除表结构及数据（删除后可在回收站查看并恢复）  drop table <table\_name>;  --删除表结构及数据（删除后不可在回收站查看并恢复）  drop table <table\_name> purge;  【示例1】  drop table t\_student; |

## 1.8回收站

|  |
| --- |
| *--查看回收站*  show recyclebin; 或 select \* from recyclebin; |

|  |
| --- |
| *--清空回收站*  purge recyclebin; |

# Web Service

## 一、Web Service介绍

### 1. Web Service的作用：

它可以解决跨平台(跨操作平台、应用平台)、跨语言的应用之间的远程调用.

Web Service也属于Java EE的规范.它是一种思想.

MVC: Struts2、SpringMVC

ORM: Hibernate4、JPA

### 2. Web Service的应用场景：

a. 手机客户端与Web应用平台进行数据交互.

b. 不同公司的业务整合:

不同公司的业务整合面临得是不同公司业务平台的整合.这个时候有可能出现不

同公司的业务平台运行环境不一样，开发语言不一样.

c. 同一个公司的业务整合：

同一个公司的业务整合，新、旧系统整合.

实现跨平台跨语言应用之间的远程调用.

### 3. Web Service与SOA(面向服务架构)

SOA：Service Oriented Architecture 面向服务架构，

一个软件由N个服务组件以完全解耦的方式组合在一起，

组件与组件之间通过ESB(企业服务总线)进行通信。

它是目前较热门的技术，由ESB与Web Service技术组成。

Enterprise Service Bus：ESB

OA: 20多个模块

20多个服务组件.

### 4. Web Service与云服务

云服务：更多是一种商业模式，对于云时代电脑而言，

所有的数据计算、存储、程序…都交给网络的“云”完成，

而“云”则由云服务商提供。

按照云计算的理念：所有用户应该按需付费(就像用水、用电一样)

。当云服务商对外提供服务给客户调用时，云服务商一定会采用一种跨平台、

跨语言的技术来提供“云服务”，此时Web Service也是一个极佳的选择。

## 二、Web Service思想的实现框架:

1. Axis ---> Axis2

2. XFire + Celtix(ESB) : CXF (SOA)

只学Web Service部分.

## 三、CXF下载与安装:

-- 它的官方网站：http://cxf.apache.org

-- apache-cxf-2.6.16.zip

-- 解压apache-cxf-2.6.16.zip

-- bin : 存放了一些命令

-- docs: api文档

-- lib: 存放了所有模块的jar与第三方框架的jar包.

-- modules： 所有模块的jar.

-- samples: 示例.

安装：把bin目录添加到环境变量.

## 四、CXF入门：

a. 开发服务端暴露Web Service服务组件.

开发步骤：

-- 拷贝jar(7个jar).

|  |
| --- |
| asm-3.3.1.jar  cxf-2.6.16.jar  neethi-3.0.3.jar  stax2-api-3.1.4.jar  woodstox-core-asl-4.4.0.jar  wsdl4j-1.6.3.jar  xmlschema-core-2.0.3.jar |

-- 开发Web Service服务组件

WS服务组件 = WS服务接口 + WS服务接口实现类.

WS服务接口 = 普通接口 + @WebService注解

WS服务接口实现类 = 普通接口实现类 + @WebService注解

@WebService(endpointInterface="接口的包名.类名", serviceName="服务名称")

-- 调用Endpoint.publish()方法来暴露WS服务组件.

访问地址：http://192.168.13.10:8080/itcast?wsdl

普通的java项目需要jetty的支持(6个jetty的jar包)

geronimo-servlet\_3.0\_spec-1.0.jar

tomcat: Web服务器, apache组织 作者是Java EE规范的制订者.

jetty: Web服务器, codehuas组织 --> eclipse组织

jetty它并不比tomcat的差，有些方面甚至比tomcat更好.

a. 开发客户端调用Web Service服务组件.

-- 前提条件：必须要有服务端提供的WebService访问地址.

http://192.168.13.10:8080/itcast?wsdl

1. 用命令生成客户端调用代码：

第一种方式：用cxf自带的wsdl2java命令.

wsdl2java http://192.168.13.10:8080/itcast?wsdl

第二种方式：用java自带的wsimport命令.

wsimport http://192.168.13.10:8080/itcast?wsdl -keep

2. 从生成的N个类中找一个Service的子类，用它来创建WebService服务工厂.

再调用该服务工厂中的getXxxxPort的方法来获取WS服务组件的代理对象.

注意：客户端可以用任何开发语言开发.

## 五、开发WebService服务组件时方法的形参类型与返回值类型.

a. CXF框架能够处理的数据类型.

1. 标量类型: 8个基本类型、包装类、String.

2. 集合: List、Set、Map(jdk1.8).

3. java Bean式复合式(dto、entity、domain、pojo).

4. 数组类型.

b. CXF框架不能够处理的数据类型.

1. Map(jdk1.7).

Map<key,value> --> List<MapItem>

key,value --> MapItem

2. File.

JAXB: Java And Xml Binding 它也是Java EE规范.

它主要是制订java语言与xml语言之间数据转换的规范.

自己写类型转换适匹器：

-- 客户端传进来的参数要转换成服务端能够处理的类型.

-- 服务端方法的返回值要转换成客户端能够处理的类型.

写自己的类型转换适匹器需要继承XmlAdapter这个类.

XmlAdapter<ValueType,BoundType>:

-- ValueType: 是JAXB它知道处理的类型.

-- BoundType:是JAXB不它知道处理的类型.

-- 不能处理的数据类型前面加：@XmlJavaTypeAdapter

## 六、Web Service三大技术基础：

-WSDL：Web Service Description Language， Web Service描述语言。

-SOAP：Simple Object Access Protocol简单对象访问协议，Web Service的通信协议。

-UDDI：Universal Description Discovery and Integration统一描述、发现、整合，服务组件注册中心(名存实亡)。

WSDL: Web Service描述语言：

WSDL文档分两部分：

-- 接口部分：

http://192.168.13.10:8080/itcast?wsdl=AdminService.wsdl

<wsdl:types.../>:

-- xs:schema : 是一份标准的schema约束文件.

-- <xs:complexType ../> : 每一个复杂类型都会生成一个Java类.

<wsdl:message .../>: N个message对应操作中的输入、输出消息.

<wsdl:portType name="AdminService"> : 端口类型(name="服务组件接口类名")

<wsdl:operation .../>: 对应接口中的方法.

-- <wsdl:input ../> : 输入消息(调用方法需要传入的参数)

-- <wsdl:output ../> : 输出消息(调用方法的返回值)

</wsdl:portType>

Login

private String arg0;

private String arg1;

LoginResponse

private String \_return;

以登录操作为例：

输入消息：

<login>

<arg0>admin</arg0>

<arg1>888888</arg1>

</login>

输出消息：

<loginResponse>

<return>admin,登录成功！</return>

</loginResponse>

-- 实现部分：http://192.168.13.10:8080/itcast?wsdl

<wsdl:binding .../> : 指定传输风格与传输协议(SOAP)

-- <soap:binding style="document"

transport="http://schemas.xmlsoap.org/soap/http"/>

<wsdl:service .../> : 生成Web Service服务工厂类.

-- <soap:address location="http://192.168.13.10:8080/itcast"/>:

指定了访问地址.

WSDL文档描述了如下三个方面：

- WHAT：描述了Web Service包含哪些可

供调用的操作。

- HOW：调用Web Service操作需要怎样的

传入、传出消息。

- WHERE：Web Service的服务地址。

一次Web Service调用的完整过程：

1.客户端将调用参数转换成XML片段。

2. 客户端将XML片段通过网络发送给服务端。

3. 服务端接收XML片段。

4. 服务端解析XML片段，提取其中数据，并转换相应的调用参数。

5. 服务端执行方法，得到返回值。

6. 服务端将返回值转换成XML片段。

7. 服务端将XML片段通过网络发送给客户端。

8. 客户端接收XML片段。

9. 客户端解析XML片段，提取其中数据，转换成调用WS的返回值。

说明：通过上面描述可以发现，一个语言要支持Web Service，只要满足如下2个要求：

- 支持XML的解析、生成。

- 支持网络通信