**三大框架技术**

**struts2**

**hibernate**

**spring**

北京尚学堂科技 内部讲义 2011年

[第一篇 STRUTS2基础 7](#_Toc288088698)

[MVC概述 7](#_Toc288088699)

[Struts2概述 8](#_Toc288088700)

[Struts1、struts2、webwork关系 8](#_Toc288088701)

[为什么要使用struts2框架 8](#_Toc288088702)

[Struts2的优势 8](#_Toc288088703)

[Struts2的下载和目录分析 10](#_Toc288088704)

[第一个struts2程序 10](#_Toc288088705)

[Struts2工作流程分析 13](#_Toc288088706)

[STRUTS2框架内部流程 13](#_Toc288088707)

[Struts2的线程安全问题 14](#_Toc288088708)

[Action详解和配置 14](#_Toc288088709)

[不继承任何类的Action 14](#_Toc288088710)

[实现Action接口 15](#_Toc288088711)

[继承ActionSupport类 16](#_Toc288088712)

[Action中自定义方法和通过URI动态执行(DMI) 17](#_Toc288088713)

[Model-Driven和Property-Driven 17](#_Toc288088714)

[配置文件详解 20](#_Toc288088715)

[指定web应用默认字符集 20](#_Toc288088716)

[开发模式的使用 21](#_Toc288088717)

[package配置 21](#_Toc288088718)

[配置命名空间(namespace) 27](#_Toc288088719)

[配置Action 27](#_Toc288088720)

[配置Result 27](#_Toc288088721)

[通配符配置Action 29](#_Toc288088722)

[配置拦截器 30](#_Toc288088723)

[多配置文件实现 30](#_Toc288088724)

[通过注解实现action的零配置 31](#_Toc288088725)

[第二篇 STRUTS2 深入 31](#_Toc288088726)

[数据类型转换器 31](#_Toc288088727)

[使用Validator配置文件校验数据 33](#_Toc288088728)

[手动数据校验 36](#_Toc288088729)

[ActionContext对象 39](#_Toc288088730)

[用ActionContext获取Web容器信息 41](#_Toc288088731)

[用ServletActionContext获得真正web容器信息 42](#_Toc288088732)

[Valuestack 42](#_Toc288088733)

[<s:debug>标签查看ActionContext 42](#_Toc288088734)

[OGNL 42](#_Toc288088735)

[OGNL获得上下文中的属性和对象导航 45](#_Toc288088736)

[OGNL中运算符 46](#_Toc288088737)

[OGNL操作集合 48](#_Toc288088738)

[OGNL其他技术 51](#_Toc288088739)

[第三篇 struts2标签 53](#_Toc288088740)

[struts2标签和EL表达式 53](#_Toc288088741)

[常用数据标签 53](#_Toc288088742)

[property 输出变量值 53](#_Toc288088743)

[set 设置变量值 53](#_Toc288088744)

[action 执行Action 54](#_Toc288088745)

[date 输出日期 55](#_Toc288088746)

[i18n、text资源国际化 56](#_Toc288088747)

[include 包含其他资源执行结果 56](#_Toc288088748)

[param 传参 57](#_Toc288088749)

[常用控制标签 57](#_Toc288088750)

[if elseif else 流程控制标签 57](#_Toc288088751)

[iterator 循环迭代标签 58](#_Toc288088752)

[常用表单UI标签 59](#_Toc288088753)

[form 表单 59](#_Toc288088754)

[textfield 文本域 61](#_Toc288088755)

[textarea 多行文本域 61](#_Toc288088756)

[file 文件域 61](#_Toc288088757)

[checkbox 62](#_Toc288088758)

[checkboxlist 复选框 62](#_Toc288088759)

[radio 单选按钮 63](#_Toc288088760)

[selet下拉框 64](#_Toc288088761)

[autocompleter自动完成框 64](#_Toc288088762)

[autocompleter动态数据加载 65](#_Toc288088763)

[datetimepicker 标签日期选择器 65](#_Toc288088764)

[doubleselect 联动下拉框 66](#_Toc288088765)

[optiontransferselect 选项移动下拉框 66](#_Toc288088766)

[optgroup 选项分组下拉框 67](#_Toc288088767)

[常用非表单UI标签 67](#_Toc288088768)

[fielderror、actionerror、actionmessage 输出错误信息 67](#_Toc288088769)

[debug 输出调试信息 67](#_Toc288088770)

[div、submit、a 67](#_Toc288088771)

[tabbedpanel tab页效果 68](#_Toc288088772)

[tree 树结构 70](#_Toc288088773)

[资源国际化 71](#_Toc288088774)

[资源国际化开发技术 71](#_Toc288088775)

[占位符的处理 72](#_Toc288088776)

[%、#在OGNL中的用法 73](#_Toc288088777)

[第四篇 struts2 拦截器 74](#_Toc288088778)

[什么是拦截器、拦截器栈? 74](#_Toc288088779)

[拦截器和过滤器的区别 74](#_Toc288088780)

[Struts2拦截器示意图 75](#_Toc288088781)

[拦截器在struts2中的应用 75](#_Toc288088782)

[默认拦截器说明 75](#_Toc288088783)

[计时拦截器 76](#_Toc288088784)

[token防止表单重复提交拦截器 77](#_Toc288088785)

[自定义拦截器 78](#_Toc288088786)

[拦截器执行顺序 79](#_Toc288088787)

[方法拦截器 79](#_Toc288088788)

[文件上传 80](#_Toc288088789)

[拦截器栈 81](#_Toc288088790)

[默认拦截器(栈) 82](#_Toc288088791)

[第一篇Hibernate基础 83](#_Toc288088792)

[持久化、持久层 83](#_Toc288088793)

[DAO模式 83](#_Toc288088794)

[ORM是什么? 83](#_Toc288088795)

[常见ORM框架比较和介绍 84](#_Toc288088796)

[JPA是什么?他和Hibernate什么关系? 85](#_Toc288088797)

[EJB3.0和JPA是javaee5.0主要的亮点！ 85](#_Toc288088798)

[Hibernate介绍 85](#_Toc288088799)

[**Relational Persistence for Java and .NET** 85](#_Toc288088800)

[Hibernate优势 86](#_Toc288088801)

[版本历史 86](#_Toc288088802)

[第一个Hibernate程序 86](#_Toc288088803)

[Hibernate程序开发流程总结 94](#_Toc288088804)

[使用hibernate的项目开发流程对比 95](#_Toc288088805)

[Hibernate架构 95](#_Toc288088806)

[Hibernate核心API 96](#_Toc288088807)

[Configuration 96](#_Toc288088808)

[SessionFactory (org.hibernate.SessionFactory) 96](#_Toc288088809)

[**SessionFactory使用要点如下：** 96](#_Toc288088810)

[Session (org.hibernate.Session) 97](#_Toc288088811)

[Session对象可以由SessionFactory对象创建。 97](#_Toc288088812)

[Transaction 97](#_Toc288088813)

[标准的Hibernate代码 97](#_Toc288088814)

[Hibernate对象的状态和生命周期 99](#_Toc288088815)

[Transient 瞬时/临时态 99](#_Toc288088816)

[Persist 持久态 99](#_Toc288088817)

[Detached托管/游离态 99](#_Toc288088818)

[通过代码测试三种状态之间的转换和区别 100](#_Toc288088819)

[Hibernate配置文件hibernate.cfg.xml详解 103](#_Toc288088820)

[Hibernate映射文件 103](#_Toc288088821)

[Hibernate基本数据类型 104](#_Toc288088822)

[Hibernate主键生成方式 104](#_Toc288088823)

[UUID经常用来生成主键值完全唯一的情况，即全局唯一标识！ 105](#_Toc288088824)

[写出自己的Hibernate 105](#_Toc288088825)

[第二篇 Hibernate高级映射 106](#_Toc288088826)

[实体映射 106](#_Toc288088827)

[基本实体映射技术 106](#_Toc288088828)

[**Element :** class 106](#_Toc288088829)

[联合主键映射 106](#_Toc288088830)

[组合关系映射 108](#_Toc288088831)

[大对象映射 110](#_Toc288088832)

[Clob:文本大对象，最长4G 110](#_Toc288088833)

[关联映射 112](#_Toc288088834)

[一对多/多对一 112](#_Toc288088835)

[双向关联和单向关联 115](#_Toc288088836)

[cascade属性 115](#_Toc288088837)

[inverse属性 117](#_Toc288088838)

[**Inverse=false，关联关系可以由双方维护！** 120](#_Toc288088839)

[一对一 120](#_Toc288088840)

[多对多 124](#_Toc288088841)

[集合映射 127](#_Toc288088842)

[**Set(一般使用Set)** 127](#_Toc288088843)

[**<key>** 127](#_Toc288088844)

[**</key>** 127](#_Toc288088845)

[**List** 127](#_Toc288088846)

[<key> 127](#_Toc288088847)

[Map 128](#_Toc288088848)

[<key column=*"deptid"></key>* 128](#_Toc288088849)

[<one-to-many class=*"Employee" />* 128](#_Toc288088850)

[继承映射 128](#_Toc288088851)

[第三篇 Annotation 注解的应用 128](#_Toc288088852)

[Hibernate Annotation介绍 128](#_Toc288088853)

[使用Annotation开发项目 129](#_Toc288088854)

[常见的注解及其典型用法 131](#_Toc288088855)

[一对一 132](#_Toc288088856)

[一对多/多对一 133](#_Toc288088857)

[多对多 135](#_Toc288088858)

[继承映射 137](#_Toc288088859)

[第四篇 Hibernate查询 138](#_Toc288088860)

[HQL查询(Hibernate Query Language) 138](#_Toc288088861)

[基本规则 138](#_Toc288088862)

[第一个HQL查询 138](#_Toc288088863)

[根据返回类型划分HQL查询 139](#_Toc288088864)

[Where子句和参数传递 141](#_Toc288088865)

[运算符和统计函数 141](#_Toc288088866)

[HQL分页查询 141](#_Toc288088867)

[跨表查询和对象导航 142](#_Toc288088868)

[Join(内连接、外连接) 142](#_Toc288088869)

[SQL原生查询(Native SQL) 143](#_Toc288088870)

[条件查询(Criteria Queries) 143](#_Toc288088871)

[第五篇 hibernate缓存机制和事务隔离机制 143](#_Toc288088872)

[一级缓存（ Session缓存） 143](#_Toc288088873)

[二级缓存(SessionFactory缓存) 144](#_Toc288088874)

[各种二级缓存插件 144](#_Toc288088875)

[查询缓存 145](#_Toc288088876)

[缓存算法问题 145](#_Toc288088877)

[1+N问题 145](#_Toc288088878)

[List和iterator区别 146](#_Toc288088879)

[缓存机制的选用 146](#_Toc288088880)

[数据批量处理 146](#_Toc288088881)

[事务隔离级别 146](#_Toc288088882)

[乐观锁和悲观锁 147](#_Toc288088883)

[附录 148](#_Toc288088884)

[Hibernate依赖库中相关jar包作用 148](#_Toc288088885)

[使用myeclipse自动开发hibernate程序(oracle 为例) 149](#_Toc288088886)

[Hibernate实践总结 155](#_Toc288088887)

[SPRING预备知识 156](#_Toc288088888)

[工厂模式 156](#_Toc288088889)

[单例模式((singleton) 157](#_Toc288088890)

[1、控制资源的使用，通过线程同步来控制资源的并发访问； 157](#_Toc288088891)

[动态代理模式 158](#_Toc288088892)

[面向接口编程 158](#_Toc288088893)

[spring 入门 159](#_Toc288088894)

[spring优势 159](#_Toc288088895)

[spring如何选用 159](#_Toc288088896)

[项目中配置spring 160](#_Toc288088897)

[和我们的第一个spring程序 160](#_Toc288088898)

[IOC&DI控制反转/依赖注入 162](#_Toc288088899)

[IOC(Inverse Of Control ) 162](#_Toc288088900)

[DI(Dependency Injection) 162](#_Toc288088901)

[IOC&DI的关系 163](#_Toc288088902)

[IOC/DI实现技术：setter注入 163](#_Toc288088903)

[IOC/DI实现技术：构造方法注入 163](#_Toc288088904)

[Annotation实现注入操作 163](#_Toc288088905)

[IOC容器管理的bean作用域 164](#_Toc288088906)

[自动扫描将bean纳入IOC容器管理(spring2.5新增) 165](#_Toc288088907)

[完成自己的IOC容器 165](#_Toc288088908)

[AOP面向切面编程 166](#_Toc288088909)

[AOP介绍 166](#_Toc288088910)

[AOP基本概念 166](#_Toc288088911)

[通知的类型 167](#_Toc288088912)

[spring中如何配置AOP 167](#_Toc288088913)

[Annotation方式配置AOP 167](#_Toc288088914)

[XML方式配置AOP 168](#_Toc288088915)

[框架整合 169](#_Toc288088916)

[整合hibernate 169](#_Toc288088917)

[事务管理 170](#_Toc288088918)

[annotation方式的事务管理 172](#_Toc288088919)

[XML配置方式管理事务(推荐) 172](#_Toc288088920)

[HibernateTemplate 173](#_Toc288088921)

[模板方法模式/回调方法/钩子函数 174](#_Toc288088922)

[HibernateDaoSupport 174](#_Toc288088923)

[OpenSessionInView管理session 175](#_Toc288088924)

[ThreadLocal模式管理session 175](#_Toc288088925)

[整合struts2 177](#_Toc288088926)

[spring抽象服务 178](#_Toc288088927)

[邮件处理 178](#_Toc288088928)

[任务调度(Quatz) 178](#_Toc288088929)

# 第一篇 STRUTS2基础

## MVC概述

MVC是三个单词的缩写,分别为： 模型(Model),[视图](http://baike.baidu.com/view/71981.htm)(View) 和控制Controller)。 MVC模式的目的就是实现Web系统的职能分工。

Model层实现系统中的业务逻辑，通常可以用JavaBean或EJB来实现。

View层用于与用户的交互，通常用JSP来实现。

Controller层是Model与View之间沟通的桥梁，它可以分派用户的请求并选择恰当的视图以用于显示，同时它也可以解释用户的输入并将它们映射为模型层可执行的操作。

## Struts2概述

Struts这个名字来源于在建筑和旧式飞机中使用的金属支架。Struts2比内部实现更加复杂，但是使用起来更加简单，功能更加强大。

Struts是apache基金会jakarta项目组的一个开源项目，采用MVC模式，能够很好的帮助我们提高开发web项目的效率。Struts主要采用了servlet和jsp技术来实现，把servlet、jsp、标签库等技术整合到整个框架中。

## Struts1、struts2、webwork关系

Struts1是第一个广泛流行的mvc框架，使用及其广泛。但是，随着技术的发展，尤其是JSF、ajax等技术的兴起，struts1有点跟不上时代的步伐，以及他自己在设计上的一些硬伤，阻碍了他的发展。

同时，大量新的mvc框架渐渐大踏步发展，尤其是webwork。Webwork是opensymphony组织开发的。Webwork实现了更加优美的设计，更加强大而易用的功能。

后来，struts和webwork两大社区决定合并两个项目，完成struts2.事实上，struts2是以webwork为核心开发的，更加类似于webwork框架，跟struts1相差甚远。

## 为什么要使用struts2框架

框架可以大大提高我们的开发效率。并且框架是一种主动是的设计，我们使用框架必须遵守框架制定好的开发流程。

Rickard Oberg(webwork和jboss的创造者)曾经说过:”框架的强大之处不是源自他能让你做什么，而是他不能让你做什么”。也就是说，使用框架进行开发，我们可以使用标准的流程工作，避免了开发的混乱，避免杂乱的jsp造成的混乱。

Struts2无疑是当今最新的mvc框架，发展前景无可限量。事实上，早已经在众多项目中开始使用。

## Struts2的优势

* *Build!*
  + **Easy startup** - Jumpstart new projects with our bootstrap tutorial and template application or Maven archetype.
  + **Improved Design** - Code clean against HTTP-independant framework interfaces.
  + **Enhanced Tags** - Code less with stylesheet-driven form tags that provide their own markup.
  + **Stateful Checkboxes** - Avoid special handling with smart checkboxes that know when they are toggled.
  + **Flexible Cancel Buttons** - Go directly to a different action on cancel.
  + **First-class AJAX support** - Add interactivity and flexibility with AJAX tags that look and feel just like standard Struts tags.
  + **Easy Spring integration** - Inject dependencies into Actions using Spring without glue code or red tape. (Plexus support also available.)
  + **Enhanced Results** - Do more with speciality results for JasperReports, JFreeChart, Action chaining, and file downloading.
  + **POJO forms** - No more ActionForms! Use any JavaBean to capture form input or put properties directly on an Action class. Use both binary and String properties!
  + **POJO Actions** - Use any class as an Action class -- even the interface is optional!
* *Deploy!*
  + **Easy plugins** - Add framework extensions by dropping in a JAR. No manual configuration required! Bundled plugins add support for JavaServer Faces, JasperReports, JFreeChart, Tiles, and more ...
  + **Integrated profiling** - Peek inside Struts2 to find where the cycles are going!
  + **Precise Error Reporting** - Flip directly to the location and line of an error.
* *Maintain!*
  + **Easy-to-test Actions** - Test Struts2 Actions directly, without resorting to mock HTTP objects.
  + **Intelligent Defaults** - Skip obvious and redundant settings. Most framework configuration elements have a default value that we can set and forget. Say it once!
  + **Easy-to-customize controller** - Customize the request handling per action, if desired. Struts2 only does what you want it to do!
  + **Integrating Debugging** - Research problem reports with built-in debugging tools.
  + **Easy-to-tweak tags** - Customize tag markup by editing a FreeMarker template. No need to grok the taglib API! *JSP, FreeMarker, and Velocity tags are fully supported.*

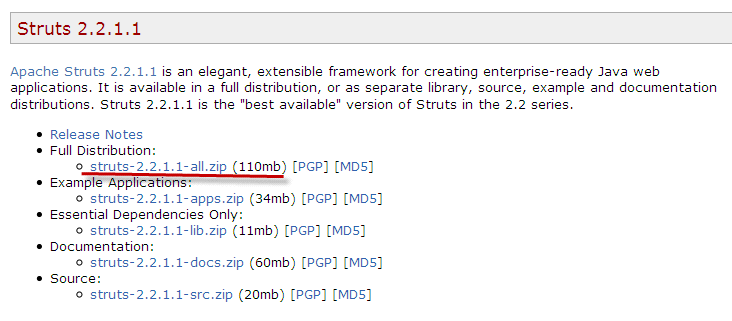
## Struts2的下载和目录分析

1. 下载struts2

官网地址：<http://struts.apache.org/>

最新版本是struts2.2，我们下载使用struts2.2. 下载地址如下：

[http://struts.apache.org/download.cgi](http://struts.apache.org/download.cgi#struts2014)



1. 分析struts2目录及使用自带的示例

src: 源代码

docs: api文档

lib: 依赖库

apps: 官方例子(尤其注意：strut2-blank包)

## 第一个struts2程序

1. 建立web项目

2. 解压struts2/apps/struts-blank.war包。

3. 引入strut2相关jar包

将struts-blank项目下的依赖库coypy到自己项目中

4. 配置web.xml，增加struts2提供的过滤器(参考struts-blank项目)：

|  |
| --- |
| <?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"*>  <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>    </web-app> |

5. 建立包：com.bjsxt.struts.test，并增加普通java类，代码如下：

|  |
| --- |
| **package** com.bjsxt.struts.test;  **public** **class** FirstAction {    **private** String msg;    **public** String execute() **throws** Exception{  System.*out*.println("FirstAction.test1()");  setMsg("为了让生活美好！");  **return** "success";  }  **public** String getMsg() {  **return** msg;  }  **public** **void** setMsg(String msg) {  **this**.msg = msg;  }  } |

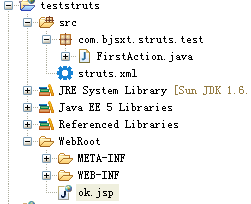
6. 在src下增加struts.xml，并增加FirstAction类的配置内容：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  "http://struts.apache.org/dtds/struts-2.0.dtd">  <struts>  <package name=*"default"* namespace=*"/"* extends=*"struts-default"*>  <action name=*"first"* class=*"com.bjsxt.struts.test.FirstAction"*>  <result name=*"success"*>ok.jsp</result>  </action>  </package>  <!-- Add packages here -->  </struts> |

7. 增加ok.jsp页面，用来显示FirstAction中的属性msg：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <body>  <s:property value=*"msg"* />  </body>  </html> |

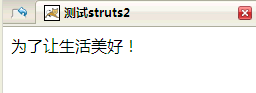
最终web项目文件结构如下：



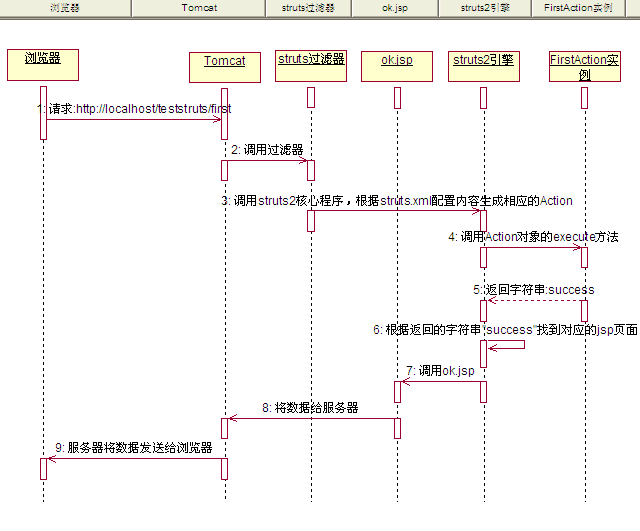
8. 项目发布到tomcat下。

9. 测试。在地址栏输入：<http://localhost/teststruts/first>。控制台打印：

FirstAction.test1() ，页面显示：



## Struts2工作流程分析



## STRUTS2框架内部流程

1. 客户端发送请求的tomcat服务器。服务器接受，将HttpServletRequest传进来。

2. 请求经过一系列过滤器(如：ActionContextCleanUp、SimeMesh等)

3. FilterDispatcher被调用。FilterDispatcher调用ActionMapper来决定这个请求是否要调用某个Action

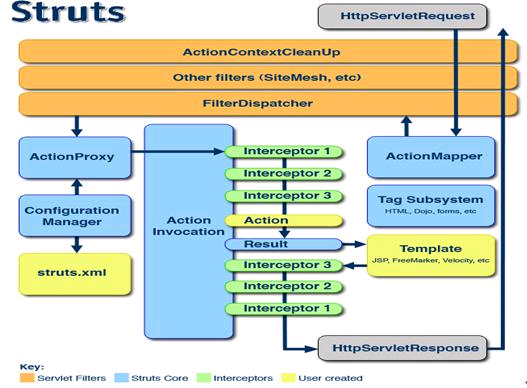
4. ActionMapper决定调用某个ActionFilterDispatcher把请求交给ActionProxy

5. ActionProxy通过Configuration Manager查看struts.xml，从而找到相应的Action类

6. ActionProxy创建一个ActionInvocation对象

7. ActionInvocation对象回调Action的execute方法

8. Action执行完毕后，ActionInvocation根据返回的字符串，找到对应的result。然后将Result内容通过HttpServletResponse返回给服务器。



## Struts2的线程安全问题

Struts1中所有的Action都只有一个实例，被反复的调用。Struts2中的Action处理一次请求会生成一个实例，这些实例彼此完全独立，使用完毕后就立马销毁。

struts1中的Action线程不安全，一般将不能有可写的属性(即将Action设计成无状态类)。

Struts2中的Action线程安全，可以任意设置属性！

## Action详解和配置

Struts1中的action需要实现特定的接口。Struts2中的Action相当的灵活，既可以实现接口、也可以不实现而只是个普通java类。就像我们前面写的helloworld程序一样。

### 不继承任何类的Action

这种方式的好处是，我们写的Action类完全不和struts2框架发生耦合，代码不依赖struts2的类库。当然，弊端也比较明显不能使用struts2中的某些功能。代码如下：

|  |
| --- |
| **package** com.bjsxt.struts.test;  **public** **class** FirstAction {    **private** String msg;    **public** String execute() **throws** Exception{  System.*out*.println("FirstAction.test1()");  setMsg("为了让生活美好！");  **return** "success";  }  **public** String getMsg() {  **return** msg;  }  **public** **void** setMsg(String msg) {  **this**.msg = msg;  }  } |

### 实现Action接口

Struts2的Action接口中只定义了execute方法和几个预定义的常量。代码如下：

|  |
| --- |
| package com.opensymphony.xwork2;  public interface Action {  public static final String SUCCESS = "success";  public static final String NONE = "none";  public static final String ERROR = "error";  public static final String INPUT = "input";  public static final String LOGIN = "login";  public String execute() throws Exception;  } |

开发中，我们尽量使用Action接口中预定义的常量，实现规范编程。如下为我们改版的Action类代码：

|  |
| --- |
| **package** com.bjsxt.struts.test;  import com.opensymphony.xwork2.Action;  **public** **class** FirstAction **implements Action** {    **private** String msg;    **public** String execute() **throws** Exception{  System.*out*.println("FirstAction.test1()");  setMsg("为了让生活美好！");  **return** ***SUCCESS***; //直接使用接口中的常量  }  **public** String getMsg() {  **return** msg;  }  **public** **void** setMsg(String msg) {  **this**.msg = msg;  }  } |

### 继承ActionSupport类

ActionSupport类实现了Action接口，我们自定义的Action类一般都采用继承ActionSupport类的方式。使用ActionSupport的好处是我们可以直接使用这个类中已经定义好的方法。

代码如下：

|  |
| --- |
| **package** com.bjsxt.struts.test;  **import** com.opensymphony.xwork2.ActionSupport;  **public** **class** FirstAction extends ActionSupport {    **private** String msg;    **public** String execute() **throws** Exception{  System.*out*.println("FirstAction.test1()");  setMsg("为了让生活美好！");  **return** *SUCCESS*;  }  **public** String getMsg() {  **return** msg;  }  **public** **void** setMsg(String msg) {  **this**.msg = msg;  }  } |

### Action中自定义方法和通过URI动态执行(DMI)

Action中可以定义多个任意名称的方法，不是只有execute方法。我们为Action增加test2方法：

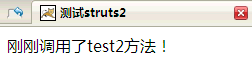
|  |
| --- |
| **public** String test2() **throws** Exception {  System.*out*.println("FirstAction.test2()");  setMsg("刚刚调用了test2方法！");  **return** *SUCCESS*;  } |

通过DMI(Dynamic Method Invoke)动态方法调用，我们可以通过URL即可调用相应的方法，相当方便。格式为：actionname!methodname

地址栏输入：<http://localhost/teststruts/first!test2>

控制台输出：FirstAction.test2()

页面显示：



### Model-Driven和Property-Driven

Struts1中我们可以将提交表单的参数使用FormBean进行封装传递给我们的Action。在struts2中我们可以使用Model-Driven和Property-Driven进行表单数据的封装，便于我们在Action中直接使用。

#### Property-Driven使用示例：

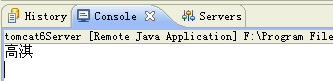
|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <body>  <form action=*ceshi/User*>  用户名：<input type=*text* name=*uname* /><br/>  密码：<input type=*password* name=*pwd* /><br/>  <input type=*submit* value=*登陆*/>  </form>  </body>  </html> |
| package com.bjsxt.struts.test;  import javax.servlet.http.HttpServletRequest;  import org.apache.struts2.ServletActionContext;  import com.opensymphony.xwork2.ActionContext;  import com.opensymphony.xwork2.ActionSupport;  import com.opensymphony.xwork2.ModelDriven;  public class UserAction extends ActionSupport{    private String uname;  private String pwd;    public String execute() throws Exception {  System.out.println(getUname());  return this.SUCCESS;  }  public String getUname() {  return uname;  }  public void setUname(String uname) {  this.uname = uname;  }  public String getPwd() {  return pwd;  }  public void setPwd(String pwd) {  this.pwd = pwd;  }  } |
| **package** com.bjsxt.struts.test;  **public** **class** User {  **private** **int** id;  **private** String uname;  **private** String pwd;  //get和set方法省略！  } |

属性驱动时，只要保持表单域名字和Action属性名一致，即可自动进行值得填充。

#### Model-Driven使用示例：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <body>  <form action=*ceshi/User*>  用户名：<input type=*text* name=*uname* /><br/>  密码：<input type=*password* name=*pwd* /><br/>  <input type=*submit* value=*登陆*/>  </form>  </body>  </html> |
| package com.bjsxt.struts.test;  import javax.servlet.http.HttpServletRequest;  import org.apache.struts2.ServletActionContext;  import com.opensymphony.xwork2.ActionContext;  import com.opensymphony.xwork2.ActionSupport;  import com.opensymphony.xwork2.ModelDriven;  public class UserAction extends ActionSupport implements ModelDriven<User> {    private User user = new User();    public String execute() throws Exception {  System.out.println(user.getUname());  return this.SUCCESS;  }  @Override  public User getModel() {  return user;  }  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  } |
| **package** com.bjsxt.struts.test;  **public** **class** User {  **private** **int** id;  **private** String uname;  **private** String pwd;  //限于篇幅，get和set方法省略！  } |

测试：<http://localhost/teststruts/login.jsp> 提交表单，发现后台正确获取输入的数据：



**属性驱动和模型驱动的选取：**

* **属性驱动方便简洁。表单域较少时可用。**
* **表单域较多且定义了相应的pojo，可以使用模型驱动。**

## 配置文件详解

### 指定web应用默认字符集

<constant name=*"struts.i18n.encoding"* value=*"gbk"* />

### 开发模式的使用

开发阶段，我们一般将指定开发模式，这样我们修改action、修改xml文件后不需要再重启服务器，直接就可以看到结果。 但是，项目上线后，需要关闭开发模式。

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

### package配置

此处package(包)不是指java中的包。而是struts配置文件中的<package>元素。

package就是把action、result、type、interceptor等这些元素打包到一个逻辑单元中。从而便于统一管理。package可以被其他包继承，可以拥有自己独立的部分，包中内容可以被子包重写。

|  |
| --- |
| <package name=*"default"* namespace=*"/"* **extends=*"struts-default"***>  <action name=*"first"* class=*"com.bjsxt.struts.test.FirstAction"*>  <result name=*"success"*>aa.txt</result>  </action>  </package> |

我们定义的包，继承了struts-default包。这个包位于：struts-default.xml中。他的内容如下：

|  |
| --- |
| <package name="struts-default" abstract="true">  <result-types>  <result-type name="chain" class="com.opensymphony.xwork2.ActionChainResult"/>  <result-type name="dispatcher" class="org.apache.struts2.dispatcher.ServletDispatcherResult" default="true"/>  <result-type name="freemarker" class="org.apache.struts2.views.freemarker.FreemarkerResult"/>  <result-type name="httpheader" class="org.apache.struts2.dispatcher.HttpHeaderResult"/>  <result-type name="redirect" class="org.apache.struts2.dispatcher.ServletRedirectResult"/>  <result-type name="redirectAction" class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>  <result-type name="stream" class="org.apache.struts2.dispatcher.StreamResult"/>  <result-type name="velocity" class="org.apache.struts2.dispatcher.VelocityResult"/>  <result-type name="xslt" class="org.apache.struts2.views.xslt.XSLTResult"/>  <result-type name="plainText" class="org.apache.struts2.dispatcher.PlainTextResult" />  </result-types>  <interceptors>  <interceptor name="alias" class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>  <interceptor name="autowiring" class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringInterceptor"/>  <interceptor name="chain" class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>  <interceptor name="conversionError" class="org.apache.struts2.interceptor.StrutsConversionErrorInterceptor"/>  <interceptor name="cookie" class="org.apache.struts2.interceptor.CookieInterceptor"/>  <interceptor name="clearSession" class="org.apache.struts2.interceptor.ClearSessionInterceptor" />  <interceptor name="createSession" class="org.apache.struts2.interceptor.CreateSessionInterceptor" />  <interceptor name="debugging" class="org.apache.struts2.interceptor.debugging.DebuggingInterceptor" />  <interceptor name="externalRef" class="com.opensymphony.xwork2.interceptor.ExternalReferencesInterceptor"/>  <interceptor name="execAndWait" class="org.apache.struts2.interceptor.ExecuteAndWaitInterceptor"/>  <interceptor name="exception" class="com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor"/>  <interceptor name="fileUpload" class="org.apache.struts2.interceptor.FileUploadInterceptor"/>  <interceptor name="i18n" class="com.opensymphony.xwork2.interceptor.I18nInterceptor"/>  <interceptor name="logger" class="com.opensymphony.xwork2.interceptor.LoggingInterceptor"/>  <interceptor name="modelDriven" class="com.opensymphony.xwork2.interceptor.ModelDrivenInterceptor"/>  <interceptor name="scopedModelDriven" class="com.opensymphony.xwork2.interceptor.ScopedModelDrivenInterceptor"/>  <interceptor name="params" class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"/>  <interceptor name="actionMappingParams" class="org.apache.struts2.interceptor.ActionMappingParametersInteceptor"/>  <interceptor name="prepare" class="com.opensymphony.xwork2.interceptor.PrepareInterceptor"/>  <interceptor name="staticParams" class="com.opensymphony.xwork2.interceptor.StaticParametersInterceptor"/>  <interceptor name="scope" class="org.apache.struts2.interceptor.ScopeInterceptor"/>  <interceptor name="servletConfig" class="org.apache.struts2.interceptor.ServletConfigInterceptor"/>  <interceptor name="sessionAutowiring" class="org.apache.struts2.spring.interceptor.SessionContextAutowiringInterceptor"/>  <interceptor name="timer" class="com.opensymphony.xwork2.interceptor.TimerInterceptor"/>  <interceptor name="token" class="org.apache.struts2.interceptor.TokenInterceptor"/>  <interceptor name="tokenSession" class="org.apache.struts2.interceptor.TokenSessionStoreInterceptor"/>  <interceptor name="validation" class="org.apache.struts2.interceptor.validation.AnnotationValidationInterceptor"/>  <interceptor name="workflow" class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>  <interceptor name="store" class="org.apache.struts2.interceptor.MessageStoreInterceptor" />  <interceptor name="checkbox" class="org.apache.struts2.interceptor.CheckboxInterceptor" />  <interceptor name="profiling" class="org.apache.struts2.interceptor.ProfilingActivationInterceptor" />  <interceptor name="roles" class="org.apache.struts2.interceptor.RolesInterceptor" />  <interceptor name="jsonValidation" class="org.apache.struts2.interceptor.validation.JSONValidationInterceptor" />  <interceptor name="annotationWorkflow" class="com.opensymphony.xwork2.interceptor.annotations.AnnotationWorkflowInterceptor" />  <interceptor name="multiselect" class="org.apache.struts2.interceptor.MultiselectInterceptor" />  <!-- Basic stack -->  <interceptor-stack name="basicStack">  <interceptor-ref name="exception"/>  <interceptor-ref name="servletConfig"/>  <interceptor-ref name="prepare"/>  <interceptor-ref name="checkbox"/>  <interceptor-ref name="multiselect"/>  <interceptor-ref name="actionMappingParams"/>  <interceptor-ref name="params">  <param name="excludeParams">dojo\..\*,^struts\..\*</param>  </interceptor-ref>  <interceptor-ref name="conversionError"/>  </interceptor-stack>  <!-- Sample validation and workflow stack -->  <interceptor-stack name="validationWorkflowStack">  <interceptor-ref name="basicStack"/>  <interceptor-ref name="validation"/>  <interceptor-ref name="workflow"/>  </interceptor-stack>  <!-- Sample JSON validation stack -->  <interceptor-stack name="jsonValidationWorkflowStack">  <interceptor-ref name="basicStack"/>  <interceptor-ref name="validation">  <param name="excludeMethods">input,back,cancel</param>  </interceptor-ref>  <interceptor-ref name="jsonValidation"/>  <interceptor-ref name="workflow"/>  </interceptor-stack>  <!-- Sample file upload stack -->  <interceptor-stack name="fileUploadStack">  <interceptor-ref name="fileUpload"/>  <interceptor-ref name="basicStack"/>  </interceptor-stack>  <!-- Sample model-driven stack -->  <interceptor-stack name="modelDrivenStack">  <interceptor-ref name="modelDriven"/>  <interceptor-ref name="basicStack"/>  </interceptor-stack>  <!-- Sample action chaining stack -->  <interceptor-stack name="chainStack">  <interceptor-ref name="chain"/>  <interceptor-ref name="basicStack"/>  </interceptor-stack>  <!-- Sample i18n stack -->  <interceptor-stack name="i18nStack">  <interceptor-ref name="i18n"/>  <interceptor-ref name="basicStack"/>  </interceptor-stack>  <!-- An example of the paramsPrepareParams trick. This stack  is exactly the same as the defaultStack, except that it  includes one extra interceptor before the prepare interceptor:  the params interceptor.  This is useful for when you wish to apply parameters directly  to an object that you wish to load externally (such as a DAO  or database or service layer), but can't load that object  until at least the ID parameter has been loaded. By loading  the parameters twice, you can retrieve the object in the  prepare() method, allowing the second params interceptor to  apply the values on the object. -->  <interceptor-stack name="paramsPrepareParamsStack">  <interceptor-ref name="exception"/>  <interceptor-ref name="alias"/>  <interceptor-ref name="i18n"/>  <interceptor-ref name="checkbox"/>  <interceptor-ref name="multiselect"/>  <interceptor-ref name="params">  <param name="excludeParams">dojo\..\*,^struts\..\*</param>  </interceptor-ref>  <interceptor-ref name="servletConfig"/>  <interceptor-ref name="prepare"/>  <interceptor-ref name="chain"/>  <interceptor-ref name="modelDriven"/>  <interceptor-ref name="fileUpload"/>  <interceptor-ref name="staticParams"/>  <interceptor-ref name="actionMappingParams"/>  <interceptor-ref name="params">  <param name="excludeParams">dojo\..\*,^struts\..\*</param>  </interceptor-ref>  <interceptor-ref name="conversionError"/>  <interceptor-ref name="validation">  <param name="excludeMethods">input,back,cancel,browse</param>  </interceptor-ref>  <interceptor-ref name="workflow">  <param name="excludeMethods">input,back,cancel,browse</param>  </interceptor-ref>  </interceptor-stack>  <!-- A complete stack with all the common interceptors in place.  Generally, this stack should be the one you use, though it  may do more than you need. Also, the ordering can be  switched around (ex: if you wish to have your servlet-related  objects applied before prepare() is called, you'd need to move  servletConfig interceptor up.  This stack also excludes from the normal validation and workflow  the method names input, back, and cancel. These typically are  associated with requests that should not be validated.  -->  <interceptor-stack name="defaultStack">  <interceptor-ref name="exception"/>  <interceptor-ref name="alias"/>  <interceptor-ref name="servletConfig"/>  <interceptor-ref name="i18n"/>  <interceptor-ref name="prepare"/>  <interceptor-ref name="chain"/>  <interceptor-ref name="debugging"/>  <interceptor-ref name="scopedModelDriven"/>  <interceptor-ref name="modelDriven"/>  <interceptor-ref name="fileUpload"/>  <interceptor-ref name="checkbox"/>  <interceptor-ref name="multiselect"/>  <interceptor-ref name="staticParams"/>  <interceptor-ref name="actionMappingParams"/>  <interceptor-ref name="params">  <param name="excludeParams">dojo\..\*,^struts\..\*</param>  </interceptor-ref>  <interceptor-ref name="conversionError"/>  <interceptor-ref name="validation">  <param name="excludeMethods">input,back,cancel,browse</param>  </interceptor-ref>  <interceptor-ref name="workflow">  <param name="excludeMethods">input,back,cancel,browse</param>  </interceptor-ref>  </interceptor-stack>  <!-- The completeStack is here for backwards compatibility for  applications that still refer to the defaultStack by the  old name -->  <interceptor-stack name="completeStack">  <interceptor-ref name="defaultStack"/>  </interceptor-stack>  <!-- Sample execute and wait stack.  Note: execAndWait should always be the \*last\* interceptor. -->  <interceptor-stack name="executeAndWaitStack">  <interceptor-ref name="execAndWait">  <param name="excludeMethods">input,back,cancel</param>  </interceptor-ref>  <interceptor-ref name="defaultStack"/>  <interceptor-ref name="execAndWait">  <param name="excludeMethods">input,back,cancel</param>  </interceptor-ref>  </interceptor-stack>  </interceptors>  <default-interceptor-ref name="defaultStack"/>  <default-class-ref class="com.opensymphony.xwork2.ActionSupport" />  </package> |

**显然，struts-default包中定义的内容，由于我们继承了他，所以可以直接使用。我们自定义的包一般都需要继承struts-default包或者他的子包。**

### 配置命名空间(namespace)

我们可以通过不同的命名空间划分不同的功能模块，这样，即使Action的名字相同只要位于不同的命名空间和不同的保重中也可以正常执行。有利于多人协作开发时保持相互之间的独立性，从而提高开发效率。

|  |
| --- |
| <package name=*"test"* namespace=*"/ceshi"* extends=*"struts-default"*>  <action name=*"first"* class=*"com.bjsxt.struts.test.FirstAction"*>  <result name=*"success"*>/ok.jsp</result>  </action>  </package> |

我们在地址栏输入：<http://localhost/teststruts/ceshi/first!test2> 即可正常执行！

### 配置Action

Action的配置信息是将：url、Action类、展示页面。三者联系起来。

url跟action的name属性对应。

class属性指定了Action类

<result>元素将Action中返回的字符串(通过result的name属性匹配)跟展示页面(通过path或者result元素的文本值指定)映射起来。

|  |
| --- |
| <action name=*"first"* class=*"com.bjsxt.struts.test.FirstAction"*>  <result name=*"success"*>/ok.jsp</result>  <!—可以有多个result元素 -->  </action> |

### 配置Result

<result>元素告诉了我们调用完Action后下一步做什么。<result>一般作为<action>的子元素，或者我们可以通过<global-result>定义全局result。

**<result>有三个基本内容：**

1. name (跟action返回的字符串匹配，从而根据不同的字符串匹配不同的result)

name如果不指定值，默认是”success”。

Action接口中定义了多个常量，方便我们统一指定name。

|  |
| --- |
| /\*\*  \* The action execution was successful. Show result  \* view to the end user.  \*/  **public** **static** **final** String *SUCCESS* = "success";  /\*\*  \* The action execution was successful but do not  \* show a view. This is useful for actions that are  \* handling the view in another fashion like redirect.  \*/  **public** **static** **final** String *NONE* = "none";  /\*\*  \* The action execution was a failure.  \* Show an error view, possibly asking the  \* user to retry entering data.  \*/  **public** **static** **final** String *ERROR* = "error";  /\*\*  \* The action execution require more input  \* in order to succeed.  \* This result is typically used if a form  \* handling action has been executed so as  \* to provide defaults for a form. The  \* form associated with the handler should be  \* shown to the end user.  \* <p/>  \* This result is also used if the given input  \* params are invalid, meaning the user  \* should try providing input again.  \*/  **public** **static** **final** String *INPUT* = "input";  /\*\*  \* The action could not execute, since the  \* user most was not logged in. The login view  \* should be shown.  \*/  **public** **static** **final** String *LOGIN* = "login"; |

2. type (默认是dispatcher，类似于Servlet中的forward请求转发)

|  |  |
| --- | --- |
| type值 | 使用说明 |
| chain | 用来处理Action链，将一个action的执行与另外一个配置好的action串连起来。用第一个action的getter方法和第二个action的setter方法来完成action之间属性的复制。 |
| dispatcher | 用来转向JSP页面，这是默认的结果类型，如果在action配置中没有配置其他的结果类型，它就会被使用 |
| freemaker | 处理FreeMarker模板 |
| httpheader | 控制特殊HTTP行为的结果类型 |
| redirect | 重定向到一个URL |
| redirectAction | 重定向到一个Action |
| stream | 向浏览器发送InputSream对象，通常用来处理文件下载，还可用于返回AJAX数据 |
| velocity | 处理Velocity模板 |
| xslt | 处理XML/XLST模板 |
| plainText | 显示原始文件内容，例如文件源代码 |

3.path 对应跟result对应的处理程序(一般为展示页面或者某个action)

path可以通过<result>的文本值直接指定！

**全局result的配置实例：**

|  |
| --- |
| <package name=*"test"* namespace=*"/ceshi"* extends=*"struts-default"*>  <global-results>  <result name=*"success"*>/ok.jsp</result>  <result name=*"ok"* type=*"plainText"*>/aa.txt</result>  </global-results>  <action name=*"first"* class=*"com.bjsxt.struts.test.FirstAction"*>  </action>  </package> |

### 通配符配置Action

当配置文件中action较多时，并且配置都很类似。那么我们可以使用通配符来做合理的配置。在路径中使用多个\*代替变化的部分，配置文件中使用{N}来访问url中\*指定部分的值。

注：{N}, N为从1-9之间的值

将FirstAction对应的配置修改如下：

|  |
| --- |
| <action name=*"\*\_\*"* class=*"com.bjsxt.struts.test.{1}Action"* method=*"{2}"* >  <result name=*"success"*>/{1}\_{2}\_ok.jsp</result>  </action> |

将ok.jsp的名称改成：First\_test2\_ok.jsp。

地址栏输入：<http://localhost/teststruts/ceshi/First_test2> (注意First中F是大写)

执行后，发现正常跳转到了success指定页面

### 配置拦截器

详情请见拦截器一章！

### 多配置文件实现

项目经常需要多人协作开发，但是如果大家都是用同一个struts.xml文件，会互相影响不利于开发的正常开展。这样，我们可以通过<include>元素指定多个配置文件。

可以在src下面建立多个struts配置文件。然后再struts.xml中分别引入：

**struts.xml内容如下：**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  "http://struts.apache.org/dtds/struts-2.0.dtd">  <struts>  <constant name=*"struts.i18n.encoding"* value=*"gbk"* />  <constant name=*"struts.devMode"* value=*"true"* />    <include file=*"struts\_1.xml"*></include>  <include file=*"struts\_2.xml"*></include>  </struts> |

struts\_1.xml中的内容如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  "http://struts.apache.org/dtds/struts-2.0.dtd">  <struts>  <package name=*"test"* namespace=*"/ceshi"* extends=*"struts-default"*>  <action name=*"\*\_\*"* class=*"com.bjsxt.struts.test.{1}Action"* method=*"{2}"* >  <result name=*"success"*>/{1}\_{2}\_ok.jsp</result>  </action>  </package>  </struts> |

执行：<http://localhost/teststruts/ceshi/First_test2> 发现结果一切正常！

## 通过注解实现action的零配置

我们可以通过Annotation注解替代了传统xml文件配置的限制。我们可以通过struts2提供的注解：@Namespace,@Result,@Results等完全代替xml文件的配置。

本节内容，目前工作中用的并不多，一般写在配置文件中。

本节内容，作为自学内容！大家网上查资料，写出文档,写出测试类.大家来讲这个内容！

# 第二篇 STRUTS2 深入

## 数据类型转换器

Jsp提交的参数全部是string类型，struts2通过拦截器可以将String类型的参数自动转化成常用的数据类型，比如：数字、时间等，然后通过get和set方法设置到对应的action中的属性。

但是，有时候struts2默认的类型转换器不能符合我们的要求。这时候，需要我们进行自定义数据类型转换器。

下面是我们自定义数据类型转换器的步骤：

1. login.jsp页面内容：

|  |
| --- |
| <form action="User.action">  用户名：<input type=text name=user.uname /> <br/>  生日：<input type=text name=user.birthday /> <br/>  年龄：<input type=text name=user.age /> <br/>  <input type=submit value=登陆 />  </form> |

1. 对应的bean：

|  |
| --- |
| package com.gao.vo;  import java.util.Date;  public class User {  private String uname;  private Date birthday;  private int age;    //省略对应的get和set方法  public User(String uname, Date birthday, int age) {  super();  this.uname = uname;  this.birthday = birthday;  this.age = age;  }  public User() {  }  } |

1. UserConvertor自定义的类型转换器

|  |
| --- |
| package com.gao.convertor;  import java.text.ParseException;  import java.text.SimpleDateFormat;  import java.util.Map;  import org.apache.struts2.util.StrutsTypeConverter;  import com.gao.vo.User;  public class UserConvertor extends StrutsTypeConverter {  @Override  public Object convertFromString(Map context, String[] values /\*使用字符串数组代表表单域的值，使用字符串数组是为了考虑最通用的情况！ \*/ , Class toClass) {  System.out.println("开始转化！");  User user = new User();  user.setUname(values[0]);  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  try {  user.setBirthday(sdf.parse(values[1]));  } catch (ParseException e) {  e.printStackTrace();  }  user.setAge(new Integer(values[2]));    return user;  }  @Override  public String convertToString(Map arg0, Object arg1) {    return null;  }    } |

1. 在src下新建xwork-conversion.properties，配置全局转换器，内容如下：

com.gao.vo.User=com.gao.convertor.UserConvertor

1. Action中代码示例：

|  |
| --- |
| package com.gao.action;  import com.gao.vo.User;  import com.opensymphony.xwork2.ActionSupport;  //使用ActionSupport可以大大简化开发  public class UserAction extends ActionSupport {  private User user;    public String execute() throws Exception{  System.out.println(user.getUname());  System.out.println(user.getBirthday());    return SUCCESS;  }  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }      } |

配置了类型转换器后，他会自动产生作用，不需要我们调用。

类型转换器一般像上面一样配置成全局转换器，如果想只对某个action起作用，可以在struts.xml中<action>元素中配置他的属性:converter即可。

## 使用Validator配置文件校验数据

我们在提交表单数据时，通常都需要增加数据的校验处理。数据校验分为：客户端校验、服务器端校验。Struts2给我们提供了丰富的支持！

Struts2提供了很多数据校验器，常用的有：

|  |  |
| --- | --- |
| 校验器 | 用法 |
| required | 验证被校验表单域是否为null |
| requiredstring | 验证被校验String是否为null或空字符串，可选参数trim，可在对字符串执行trim操作 |
| stringlength | 检验字符串的长度范围，可选参数trim、mixLength、maxLength |
| int/double | 检查输入是否为数字，并且可判断数值范围。可选参数：min、max等 |
| date | 检查date的范围，可选参数：min、max |
| email | 检查输入的是否符合email格式 |
| url | 检查输入的是否符合url格式 |
| regex | 检查是否能匹配到正则表达式。基本可以符合我们全部的验证要求！！ |

使用struts2的数据校验需要按照如下步骤编程：

1. testFormLabel.jsp页面中表单代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <s:form action=*"test"* name=*"regFrm"* validate=*"true"* >  <s:textfield label=*"用户名"* name=*"uname"*></s:textfield>  <s:password label=*"密码"* name=*"pwd"* ></s:password>  <s:textfield label=*"邮箱"* name=*"email"*></s:textfield>  <s:submit value=*"注册"* ></s:submit>  </s:form>  </body>  </html> |

2. Action类需要实现接口：Validateable。通常继承ActionSupport即可，该类实现了Validateable接口。

|  |
| --- |
| **package** com.bjsxt.struts.test;  **import** com.opensymphony.xwork2.ActionSupport;  **public** **class** TestValidateAction **extends** ActionSupport/\*需要继承从而支持校验\*/{    **private** String uname;  **private** String pwd;  **private** String email;    **public** String execute() **throws** Exception {  //增加测试代码    **return** **this**.*SUCCESS*;  }  **//省略get和set方法！**  } |

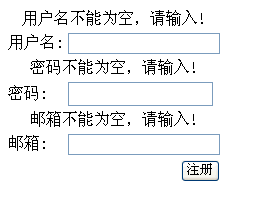
1. 在Action类所在的包中增加校验文件：Action类名[-action别名]-validation.xml。内容如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE validators PUBLIC  "-//OpenSymphony Group//XWork Validator 1.0//EN"  "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">  <validators>  <field name=*"uname"*>  <field-validator type=*"requiredstring"*>  <param name=*"trim"*>true</param>  <message>用户名不不能为空，请输入!</message>  </field-validator>  <field-validator type=*"stringlength"*>  <param name=*"trim"*>true</param>  <param name=*"minLength"*>5</param>  <param name=*"maxLength"*>10</param>  <message>用户名长度在5-10之间!</message>  </field-validator>  </field>  <field name=*"pwd"*>  <field-validator type=*"requiredstring"*>  <param name=*"trim"*>true</param>  <message>密码不不能为空，请输入!</message>  </field-validator>  <field-validator type=*"stringlength"*>  <param name=*"trim"*>true</param>  <param name=*"minLength"*>5</param>  <param name=*"maxLength"*>10</param>  <message>密码长度在5-10之间!</message>  </field-validator>  </field>  <field name=*"email"*>  <field-validator type=*"requiredstring"*>  <param name=*"trim"*>true</param>  <message>邮箱不不能为空，请输入!</message>  </field-validator>  <field-validator type=*"email"*>  <param name=*"trim"*>true</param>  <message>邮箱格式不合法！</message>  </field-validator>  </field>    </validators> |

1. struts.xml中action的相关配置中，增加input页面：

|  |
| --- |
| <action name=*"testValidate"* class=*"com.bjsxt.struts.test.TestValidateAction"*>  <result name=*"success"*>/ok.jsp</result>  **<result name=*"input"*>/testFormLabel.jsp</result>**  </action> |

1. 一般需要增加客户端验证，<s:form>中增加属性validate=true即可。
2. 测试，访问testFormLabel.jsp页面。



## 手动数据校验

我们可以手动进行数据校验，这种方式更加灵活。通过封装后，代码也比较简单。

增加reg2.jsp:

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <body>  <s:form action=*"testValidate2!register"* name=*"regFrm"* method=*"get"* >  <s:textfield label=*"用户名"* name=*"uname"*></s:textfield>  <s:password label=*"密码"* name=*"pwd"* ></s:password>  <s:textfield label=*"邮箱"* name=*"email"*></s:textfield>  <s:submit value=*"注册"* ></s:submit>  </s:form>  </body>  </html> |

TestValidate2Action的代码如下：

|  |
| --- |
| **package** com.bjsxt.test;  **import** java.util.regex.Pattern;  **import** com.opensymphony.xwork2.ActionSupport;  **public** **class** TestValidate2Action **extends** ActionSupport/\*需要继承从而支持校验\*/{    **private** String uname;  **private** String pwd;  **private** String email;    public void validateRegister(){  System.*out*.println("TestValidate2Action.validateRegister()");  if(uname==null||"".equals(uname.trim())){  this.addFieldError("uname", "用户名不能为空！");  }else if(!Pattern.*matches*(".{5,10}", uname.trim())){  this.addFieldError("uname", "用户名长度应该在5-10之间！");  }else if(!Pattern.*matches*("\\w{5,10}", uname.trim())){  this.addFieldError("uname", "用户名应该是字母、数字、下划线！");  }  }    public String register() throws Exception {  System.*out*.println("TestValidate2Action.register()");  return *SUCCESS*;  }      **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }  **public** String getPwd() {  **return** pwd;  }  **public** **void** setPwd(String pwd) {  **this**.pwd = pwd;  }  **public** String getEmail() {  **return** email;  }  **public** **void** setEmail(String email) {  **this**.email = email;  }  } |

Struts.xml中的配置如下：

|  |
| --- |
| <action name=*"testValidate2"* class=*"com.bjsxt.test.TestValidate2Action"*>  <result name=*"success"*>/ok.jsp</result>  <result name=*"input"*>/reg2.jsp</result>  </action> |

自学内容：

* + 1. 校验器短路
    2. ajax验证(完成类似用户名唯一的验证)
    3. 改变struts2错误消息样式(修改模板文件)

**个人认为：**我们自己封装的校验器已经可以满足使用要求！而且足够方便和灵活！只是你需要对regular expretion有足够的了解才行！

## ActionContext对象

Struts1的Action必须依赖于web容器，他的extecute方法会自动获得HttpServletRequest、HttpServletResponse对象，从而可以跟web容器进行交互。

Struts2的Action不用依赖于web容器，本身只是一个普通的java类而已。但是，在web开发中我们往往需要获得request、session、application等对象。这时候，可以通过ActionContext来处理。

ActionContext正如其名，是Action执行的上下文。他内部有个map属性，它存放了Action执行时需要用到的对象。

在每次执行Action之前都会创建新的ActionContext对象，所以Actioncontext是线程安全的。新new的ActionContext是保存在一个Threadlocal变量中，即采用**Threadlocal模式**。Threadlocal变量为每个线程提供独立的变量值的副本，使每个线程都可以独立的使用自己的副本，而不会和其他线程发生冲突。

通过ActionContext获取的session、request、application并不是真正的HttpServletRequest、HttpServletResponse、ServletContext对象，而是将这三个对象里面的值重新包装成了map对象。这样的封装，我们及获取了我们需要的值，同时避免了跟Web容器直接打交道，实现了完全的解耦。

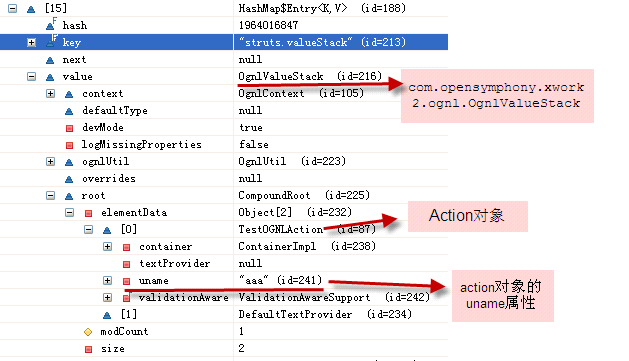
测试代码：

|  |
| --- |
| **public** **class** TestActionContextAction **extends** ActionSupport{  **private** String uname;  **public** String execute() **throws** Exception {  ActionContext ac = ActionContext.*getContext*();  System.*out*.println(ac); //在此处定义断点  **return** **this**.*SUCCESS*;  }  //限于篇幅，get和set方法省略！  } |

我们定义断点，debug进去，跟踪ac对象的值。发现他有个table属性，该属性内部包含一个map属性，该map中又有多个map属性，他们分别是：

request、session、application、action、attr、parameters等。

同时，我们跟踪request进去，发现属性attribute又是一个table，再进去发现一个名字叫做”struts.valueStack”属性。内容如下：



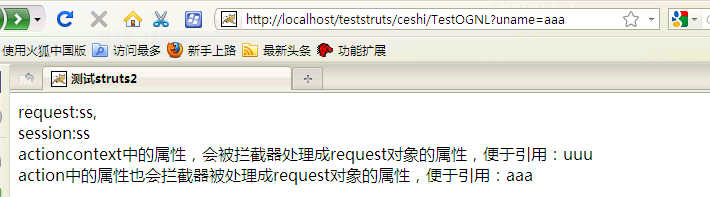
OgnlValueStack可以简单看做List，里面还放了Action对象的引用，通过它可以得到该Action对象的引用。

下图说明了几个对象的关系：

虽然我们已经从debug中得到上图的关系。为了验证我们的想法，我们写个代码进行测试：

|  |
| --- |
| **private** String uname;    **public** String execute() **throws** Exception {  ActionContext ac = ActionContext.*getContext*();  ac.put("uu", "uuu");  ac.getSession().put("s", "ss"); //直接使用：${sessionScope.s}即可获取  ((Map)ac.get("request")).put("s", "request作用域中的s参数");  System.*out*.println(ac);  **return** **this**.*SUCCESS*;  } |
| request:${requestScope.s} <br/>  session:${sessionScope.s} <br/>  actioncontext中的属性，会被拦截器处理成request对象的属性，便于引用：${requestScope.uu}<br/>  action中的属性也会拦截器被处理成request对象的属性，便于引用：${requestScope.uname }<br/> |

结果如下：



**总结：**

1. ActionContext、Action本身和HttpServletRequest对象没有关系。但是为了能够使用EL表达式、JSTL直接操作他们的属性。会有**一个拦截器**将ActionContext、Action中的属性通过类似request.setAttribute()方法置入request中(webwork2.1之前的做法)。这样，我们也可以通过：${requestScope.uname}即可访问到ActionContext和Action中的属性。

注：struts2后，使用装饰器模式来实现上述功能。

1. Action的实例，总是放到value stack中。因为Action放在stack中，而stack是root(根对象)，所以对Action中的属性的访问就可以省略#标记。

## 用ActionContext获取Web容器信息

上面我们已经测试过，通过ActionContext可以获得session、request、application，但他们并不是真正的HttpServletRequest、HttpServletResponse、ServletContext对象，而是将这三个对象里面的值重新包装成了map对象。 Struts框架通过他们来和真正的web容器对象交互。

获得session：ac.getSession().put("s", "ss");

获得request：Map m = ac.get("request");

获得application： ac.getApplication();

注意事项：

getSession()等这类代码不要在Action的构造函数里或直接给属性赋值，因为此时ActionContext里的一些值也许没有设置,这时通过ActionContext取得的值也许是null

## 用ServletActionContext获得真正web容器信息

有时，我们需要真正的HttpServletRequest、HttpServletResponse、ServletContext对象，怎么办? 我们可以通过ServletActionContext类来得到相关对象，代码如下：

HttpServletRequest req = ServletActionContext.*getRequest*();

ServletActionContext.*getRequest*().getSession();

ServletActionContext.*getServletContext*();

## Valuestack

1. OGNL查询值时是在根对象(struts2中的根对象就是ValueStack)查询。

2. 由于Action对象是放在valueStack中的，因此Action中所有的属性可以通过OGNL直接读取到。

3. 除了ValueStack中的值，其他ActionContext中的内容需要增加#来标示。加#，告诉ognl不要在根对象中寻找，而是在其他上下文对象中寻找相关值。

5. attr对象，依次从：PageContext、request、session、application作用域中寻找属性

## <s:debug>标签查看ActionContext

我们可以使用<s:debug/>查看context对象中的值。

## OGNL

OGNL全称是Object-Graph Navigation Language(对象图形导航语言)，相对于EL语言，除了保持EL语言的优点外，他的其他优点如下：

* 能够访问对象的普通方法
* 能够访问类的静态属性和静态方法
* 强大的操作集合类对象的能力
* 支持赋值操作和表达式串联
* 访问OGNL上下文和ActionContext

为了测试OGNL，我们写一个基本的TestOGNLAction，以及一个显示结果的页面ok.jsp。同时为了配合操作，我们写两个javabean：User和Address，他们的内容和配置如下：

|  |
| --- |
| **package** com.bjsxt.struts.test;  **public** **class** Address {  **private** String country;  **private** String city;  **private** String street;  //限于篇幅，get和set方法省略！  **public** Address(String country, String city, String street) {  **super**();  **this**.country = country;  **this**.city = city;  **this**.street = street;  }    **public** Address() {  }  } |
| **package** com.bjsxt.struts.test;  **public** **class** User {  **private** **int** id;  **private** String uname;  **private** String pwd;  **private** Address addr;    **public** User(**int** id, String uname, String pwd) {  **super**();  **this**.id = id;  **this**.uname = uname;  **this**.pwd = pwd;  }  **public** User(String uname, String pwd) {  **super**();  **this**.uname = uname;  **this**.pwd = pwd;  }  **public** User() {  }  //限于篇幅，get和set方法省略！  } |
| **package** com.bjsxt.struts.test;  **import** com.opensymphony.xwork2.ActionSupport;  **public** **class** TestOGNLAction **extends** ActionSupport{    **private** String uname;  **private** User user;    **public** String execute() **throws** Exception {  //在此处增加测试代码  **return** **this**.*SUCCESS*;  }  **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }  **public** User getUser() {  **return** user;  }  **public** **void** setUser(User user) {  **this**.user = user;  }  } |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  "http://struts.apache.org/dtds/struts-2.0.dtd">  <struts>  <constant name=*"struts.i18n.encoding"* value=*"gbk"* />  <constant name=*"struts.devMode"* value=*"true"* />  <package name=*"test"* namespace=*"/"* extends=*"struts-default"*>  <action name=*"testOgnl"* class=*"com.bjsxt.struts.test.TestOGNLAction"*>  <result name=*"success"*>/ok.jsp</result>  </action>  </package>  </struts> |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试OGNL</title>  </head>  <body>    </body>  </html> |

### OGNL获得上下文中的属性和对象导航

* 操作OgnlValueStack中属性

|  |
| --- |
| **private** String uname;  **private** User user;    **public** String execute() **throws** Exception {  //增加测试代码  uname="老高";  user = **new** User("root","123456");  user.setAddr(**new** Address("中国","北京","海淀区"));  **return** **this**.*SUCCESS*;  } |
| <body>  <p><s:property value=*"uname"* /></p>  <p><s:property value=*"user.addr.country"* /></p>  <s:debug></s:debug> <!-- 我们可以通过debug标签查看ActionContext中的情况 -->  </body> |
| 结果如下： |

* 操作AcitionContext其他域中的属性

|  |
| --- |
| **public** String execute() **throws** Exception {  //增加测试代码  User u = **new** User("root","123456");  u.setAddr(**new** Address("中国","北京","海淀区"));    ActionContext context = ActionContext.*getContext*();  context.put("a", "ActionContext中的值");  ((Map)context.get("request")).put("b","request中的值");  ((Map)context.get("session")).put("c","session中的值");  ((Map)context.get("application")).put("d","session中的值");    **return** **this**.*SUCCESS*;  } |
| <body>  <%  request.setAttribute("r","rrr");  pageContext.setAttribute("p","ppp");  %>    <p><s:property value=*"#a"* /></p>  <p><s:property value=*"#request.b"* /></p>  <p><s:property value=*"#parameters.param1[0]"* /></p>  <p><s:property value=*"#session.c"* /></p>  <p><s:property value=*"#application.d"* /></p>  <p><s:property value=*"#attr.p"* /></p>  <s:debug></s:debug> <!-- 我们可以通过debug标签查看ActionContext中的情况 -->  </body> |
| 结果如下： |

### OGNL中运算符

* .和[]操作符

|  |
| --- |
| <body>  <%  request.setAttribute("r","rrr");  %>    <p><s:property value=*"#request.r"* /></p>  <p><s:property value=*'#request["r"]'* /></p>  <p><s:property value=*'#request[#parameters.param2[0]]'* /></p>  <s:debug></s:debug> <!-- 我们可以通过debug标签查看ActionContext中的情况 -->  </body> |
| 测试： |

通过上面测试，我们总结：

* + 1. 点运算符操作方便，较常用。
    2. []运算符操作不方便，但是功能强大，可以传入变量值，动态取值！

* 普通运算符

OGNL支持所有的JAVA运算符。

|  |
| --- |
| <body>  <%  request.setAttribute("r","rrr");  %>  <p><s:property value=*"3\*3"* /></p>  <p><s:property value=*"3==3"* /></p>  <p><s:property value=*"#request.r==null"* /></p>  <p><s:property value=*"3+3"* /></p>  <p><s:property value=*"3+'3'"* />--${3+'3'}</p> <!-- 字符串连接符,这点不同于EL表达式-->  <p><s:property value=*"3>2&&1>2"* /></p>  <p><s:property value=*"3>2?'aa':'bb'"* /></p>  <hr/>  <s:debug></s:debug> <!-- 我们可以通过debug标签查看ActionContext中的情况 -->  </body> |
| 结果： |

* 赋值操作

|  |
| --- |
| <body>  <%  request.setAttribute("r","rrr");  request.setAttribute("r2","r2r2");  %>  <p><s:property value=*"#request.r='ppp'"* /></p>  <p><s:property value=*"#request.r=#request.r2"* /></p>  <hr/>  <s:debug></s:debug> </body> |
| 结果： |

### OGNL操作集合

* List/ Set

List的语法：{e1,e2,e3}。

|  |
| --- |
| <body>  <s:set var=*"list"* value=*"{'a','b','c'}"* scope=*"request"*></s:set>  <p><s:property value=*"#request.list.get(2)"* /></p>  </body> |
| 结果： |

* Map

|  |
| --- |
| <body>  <%  request.setAttribute("a","123456");  %>  <s:set var=*"m"* value="#{'name':'老高','pwd':#request.a}" scope=*"request"*></s:set>  <p><s:property value=*"#request.m.name"* /></p>  <p><s:property value=*"#request.m.pwd"* /></p>  </body> |
| 结果： |

* 集合中查找元素：in和not in操作符

|  |
| --- |
| <body>  <s:set var=*"s"* value=*"{'a','b','c'}"* scope=*"request"*></s:set>  <s:set var=*"m"* value="#{'name':'老高','pwd':'123456'}" scope=*"request"*></s:set>  <p><s:property value=*"'a' in {'a','b','c'}"* /></p>  <p><s:property value=*"'a' not in #request.s"* /></p>  <p><s:property value=*"'name' in #request.m.keys"* /></p>  <p><s:property value=*"'老高' in #request.m.values"* /></p>  </body> |
|  |

* 过滤/投影

我们把过滤集合中的元素产生一个子集合，叫做投影

?---符合选择逻辑的所有元素

^---符合选择逻辑的第一个元素

$---符合选择逻辑的最后一个元素

#this---表示集合是的元素

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*,com.bjsxt.struts.test.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试OGNL</title>  </head>  <body>  <%  User u1 = **new** User("张三","123456");  User u3 = **new** User("王五","123456");  User u5 = **new** User("高淇","654321");  User u6 = **new** User("马士兵","654321");  List list = **new** ArrayList();  list.add(u1);  list.add(u3);  list.add(u5);  list.add(u6);  request.setAttribute("s",list);  %>  <s:set var=*"s1"* value=*"#request.s.{?#this.pwd=='654321'}"* scope=*"request"* ></s:set>  <s:iterator value=*"#request.s1"* var=*"temp"* >  <p><s:property value=*"#temp.uname"* /></p>  </s:iterator>  </body>  </html> |
| 执行结果： |

### OGNL其他技术

* 调用对象普通方法

直接调用即可：<s:property value=*"#request.t5.test1()"* />

* 调用类的静态方法和静态属性

@class@method(args)　　　　　//调用静态方法

@class@field　　　　　　　　 //调用静态字段

注意：

* + 其中class必须给出完整的类名(包括包名),如果省略class，那么默认使用的类是java.util.Math，如：

　　@@min(5,3) @@max(5,3)　　@@PI

* + 需要修改struts.xml配置，增加：

|  |
| --- |
| <constant name=*"struts.ognl.allowStaticMethodAccess"* value=*"true"* /> |

增加测试类Test5.java：

|  |
| --- |
| **package** com.bjsxt.struts.test;  **public** **class** Test5 {  **public** **static** **int** *age*=15;  **public** **void** test1(){  System.*out*.println("调用对象普通方法！");  }  **public** String test2(){  System.*out*.println("调用对象普通方法！并有返回值");  **return** "ttt222";  }  **public** **static** String test3(){  System.*out*.println("调用类的静态方法！");  **return** "ccc";  }  } |

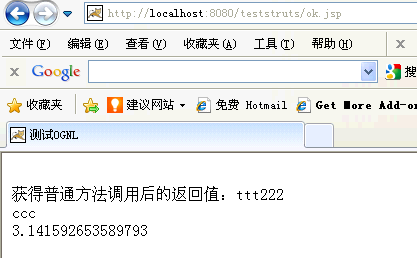
在struts.xml中增加配置：

|  |
| --- |
| <constant name=*"struts.ognl.allowStaticMethodAccess"* value=*"true"* /> |

ok.jsp中代码：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*,com.bjsxt.struts.test.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试OGNL</title>  </head>  <body>  <%  Test5 t = **new** Test5();  request.setAttribute("t5",t);  %>  <s:property value=*"#request.t5.test1()"* /> <br/>  <s:set var=*"s1"* value=*"#request.t5.test2()"* scope=*"request"* ></s:set>  获得普通方法调用后的返回值：<s:property value=*"#request.s1"* /> <br/>  <s:set var=*"s2"* value=*"@com.bjsxt.struts.test.Test5@test3()"* scope=*"request"* ></s:set>  <s:property value=*"#request.s2"* /><br/>  <s:property value=*"@@PI"* /><br/>    </body>  </html> |

执行结果：



# 第三篇 struts2标签

## struts2标签和EL表达式

**struts2标签库**开始 可以支持EL表达式。但是从2.0.11开始，不再支持EL表达式。Struts推荐更为强大的OGNL语言。

## 常用数据标签

### property 输出变量值

property标签用于输出变量的值，输出OGNL表达式的结果。前面我们已经调用了多次!下面是示例代码：

|  |
| --- |
| <s:property value=*"3\*3"* />  <s:property value=*"#request.a"* /> |

### set 设置变量值

通过set标签，我们可以给指定的变量设值，或进行集合的初始化。

|  |
| --- |
| <s:set var=*"s"* value=*"{00,11,22,33,44,55,66,77,88,99}"* ></s:set>  <s:set var=*"s2"* value=*"'aaa'"* scope=*"request"*></s:set> |

### action 执行Action

action标签用于访问某个action，并将结果包含进来或者保存到指定变量中。这里有这样几个参数需要我们理解：

name 将要执行的action在配置文件中的名字(不需要加.action后缀)

namesapce 将要执行的action所在的命名空间(可选，但是强烈建议加上！)

executeResult 是否将action执行的结果页面包含在本页面中

**ignoreContextParam:  可选，指定该页面的请求参数是否需要传入action(如果传参，置为false)**

**var 将该action对象以这里指定的名字放入StackContext中，便于后续引用**

**测试代码如下：**

|  |
| --- |
| **package** com.bjsxt.struts.test;  **import** com.opensymphony.xwork2.ActionSupport;  **public** **class** TestOGNLAction **extends** ActionSupport{    **private** String uname;    **public** String execute() **throws** Exception {  //增加测试代码  System.*out*.println("TestOGNLAction.execute()");  **return** **this**.*SUCCESS*;  }  **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }  } |
| <action name=*"testOgnl"* class=*"com.bjsxt.struts.test.TestOGNLAction"*>  <result name=*"success"*>/ok.jsp</result>  </action> |
| Ok.jsp的内容如下：  <%@ page language=*"java"* import=*"java.util.\*,com.bjsxt.struts.test.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试OGNL</title>  </head>  <body>  <h2>(((<s:property value=*"uname"* />)))</h2>  </body>  </html> |
| Test.jsp内容如下：  <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <body>  **<s:action var=*"a"* name=*"testOgnl"* namespace=*"/"* executeResult=*"true"* ignoreContextParams=*"false"* >**  **<s:param name=*"uname"*>老高</s:param>**  **</s:action>**  <s:property value=*"#a.uname"* />  --<s:property value=*"uname"* />  <s:debug></s:debug>  </body>  </html> |

### date 输出日期

date标签用于格式化输出时间。类似于我们学的DateFormat类的作用。

测试代码：

|  |
| --- |
| <%  request.setAttribute("d",**new** Date(23432432L));  %>  <s:date name=*"#request.d"* format=*"yyyy-MM-dd"* nice=*"true"* /> <!-- 增加nice，format属性不起作用 -->  <hr/>  <s:date name=*"#request.d"* format=*"yyyy-MM-dd"* var=*"c"*/> <!-- 使用var，时间不会打印出，而是保存在c变量中！ -->  <hr/>  <s:property value=*"#c"* /> |

输出结果：



### i18n、text资源国际化

见资源国际化章节！

### include 包含其他资源执行结果

作用类似于<jsp:include>用于包含其他资源的执行结果。

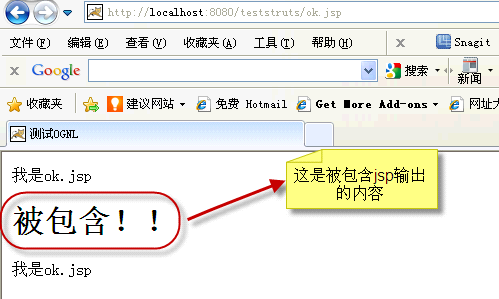
新建1.jsp，代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <h1>被包含！！</h1> |

Ok.jsp,代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*,com.bjsxt.struts.test.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试OGNL</title>  </head>  <body>  <a>我是ok.jsp</a>  <s:include value=*"1.jsp"*></s:include>  <a>我是ok.jsp</a>  </body>  </html> |

结果显示为：



### param 传参

用于给其他标签传参。比如给include标签传递参数。

## 常用控制标签

### if elseif else 流程控制标签

if标签可以独立使用。Elseif和else标签需要跟if标签搭配才能使用。说白了就是替代我们的if、elseif、else语句的功能。在jsp中使用更加方便，下面是使用例子：

|  |
| --- |
| <%  request.setAttribute("random",**new** Random().nextInt(100));  %>  <s:if test=*"#request.random>=80"*>  <b>运气不错！</b>  </s:if>  <s:elseif test=*"#request.random>=60"*>  <b>运气一般！</b>  </s:elseif>  <s:else>  <b>运气不怎么样！</b>  </s:else> |

### iterator 循环迭代标签

iterator主要用来对集合(list/set/map/数组)进行迭代。

常用的属性有：

1. value 将要迭代的集合，使用OGNL表示

2. var 指向集合中正被迭代的对象

3. begin 如果指定，迭代器从begin指定的索引开始

4. end 如果指定，迭代器到end指定的索引结束（包含end索引指向的对象）

5. step 如果指定，迭代器每次递增的增量。如果不指定，默认是1

6. status 用来表示迭代器状态的变量。他有如下的方法：

int getCount() 返回迭代次数

int getIndex() 返回当前迭代的索引值

boolean isEven() 当前迭代的索引值是否偶数

boolean isFirst() 当前迭代的索引值是否第一个元素

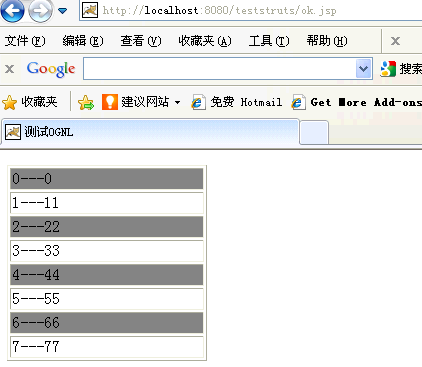
boolean isLast() 当前迭代的索引值是否最后一个元素

boolean isOdd() 当前迭代的索引值是否奇数

下面是测试代码：

|  |
| --- |
| <body>  <s:set var=*"s"* value=*"{00,11,22,33,44,55,66,77,88,99}"* ></s:set>  <table width=*"200px"* border=*"1"*>  <s:iterator value=*"#request.s"* var=*"temp"* begin=*"0"* end=*"7"* step=*"1"* status=*"idx"*>  <s:if test=*"#idx.odd"*>  <tr bgcolor=*"gray"*><td><s:property value=*"#idx.index"*/>---<s:property value=*"#temp"* /></td></tr>  </s:if>  <s:else>  <tr><td><s:property value=*"#idx.index"*/>---<s:property value=*"#temp"* /></td></tr>  </s:else>  </s:iterator>  </table>  </body> |

执行结果：



## 常用表单UI标签

**学习UI标签，我们要结合生成的HTML源代码，即可很深刻的理解标签的作用和含义。**

### form 表单

<s:form>标签用于生成HTML中的<form>，以及相关的布局代码(这里是用table标签来做布局)。

代码示例和生成的HTML代码：

|  |
| --- |
| <s:form action=*"test"* name=*"regFrm"*>  <s:textfield label=*"用户名"* name=*"uname"*></s:textfield>  <s:password label=*"密码"* name=*"pwd"* ></s:password>  <s:textfield label=*"邮箱"* name=*"email"*></s:textfield>  <s:submit value=*"注册"* ></s:submit>  </s:form> |
| <form id=*"test"* name=*"regFrm"* action=*"test"* method=*"post"*>  <table class=*"wwFormTable"*>  <tr>  <td class=*"tdLabel"*>  <label for=*"test\_uname"* class=*"label"*>  用户名:  </label>  </td>  <td>  <input type=*"text"* name=*"uname"* value=*""* id=*"test\_uname"* />  </td>  </tr>  <tr>  <td class=*"tdLabel"*>  <label for=*"test\_pwd"* class=*"label"*>  密码:  </label>  </td>  <td>  <input type=*"password"* name=*"pwd"* id=*"test\_pwd"* />  </td>  </tr>  <tr>  <td class=*"tdLabel"*>  <label for=*"test\_email"* class=*"label"*>  邮箱:  </label>  </td>  <td>  <input type=*"text"* name=*"email"* value=*""* id=*"test\_email"* />  </td>  </tr>  <tr>  <td colspan=*"2"*>  <div align=*"right"*>  <input type=*"submit"* id=*"test\_0"* value=*"&#27880;&#20876;"* />  </div>  </td>  </tr>  </table>  </form> |

根据生成的HTML源代码，我们可以深刻的理解<s:form>标记的含义。同时，我们通过javascript操作时也可以更加深刻，控制力更强。

### textfield 文本域

对应HTML中的<input type=text />，即普通文本域。

|  |
| --- |
| <s:textfield label=*"用户名"* name=*"uname"*></s:textfield> |
| <tr>  <td class=*"tdLabel"*>  <label for=*"test\_uname"* class=*"label"*>  用户名:  </label>  </td>  <td>  <input type=*"text"* name=*"uname"* value=*""* id=*"test\_uname"* />  </td>  </tr> |

### textarea 多行文本域

对应html标签中多行文本域<textarea>。

|  |
| --- |
| <s:textarea label=*"自我介绍"* name=*"selfIntro"* rows=*"5"* cols=*"50"* value=*"请输入自我说明"*></s:textarea> |
| <tr>  <td class=*"tdLabel"*>  <label for=*"test\_selfIntro"* class=*"label"*>  自我介绍:  </label>  </td>  <td>  <textarea name=*"selfIntro"* cols=*"50"* rows=*"5"* id=*"test\_selfIntro"*>&#35831;&#36755;&#20837;&#33258;&#25105;&#35828;&#26126;</textarea>  </td>  </tr> |

### file 文件域

对应文件域。示例代码如下：

|  |
| --- |
| <s:file label=*"上传头像"* name=*"logo"*></s:file> |
| <tr>  <td class=*"tdLabel"*><label for=*"test\_logo"* class=*"label"*>上传头像:</label></td>  <td><input type=*"file"* name=*"logo"* value=*""* id=*"test\_logo"*/></td>  </tr> |

### checkbox

checkbox对应HTML中的checkbox标签。示例如下：

|  |
| --- |
| <s:checkbox value=*"true"* name=*"favorite"* label=*"已阅读协议"* fieldvalue=*"1"*></s:checkbox> |
| <tr>  <td valign=*"top"* align=*"right"*>  </td>  <td valign=*"top"* align=*"left"*>  <input type=*"checkbox"* name=*"favorite"* value=*"true"*  **checked=*"checked"*** id=*"test\_favorite"* />  <input type=*"hidden"* id=*"\_\_checkbox\_test\_favorite"*  name=*"\_\_checkbox\_favorite"* value=*"true"* />  <label for=*"test\_favorite"* class=*"checkboxLabel"*>  已阅读协议  </label>  </td>  </tr> |

### checkboxlist 复选框

由于复选框往往是多个一组使用，Checkboxlist帮助我们跟集合结合使用，很容易写出多个复选框。示例如下：

|  |
| --- |
| <s:set var=*"a"* value=*"{'学java','用java','教java','发呆'}"* scope=*"request"*></s:set>  <s:checkboxlist list=*"#request.a"* name=*"favorite"* label=*"喜欢做的事"* ></s:checkboxlist> |
| <tr>  <td class=*"tdLabel"*>  <label for=*"test\_favorite"* class=*"label"*>  喜欢做的事:  </label>  </td>  <td>  <input type=*"checkbox"* name=*"favorite"* value=*"学java"* id=*"favorite-1"* />  <label for=*"favorite-1"* class=*"checkboxLabel"*>  学java  </label>  <input type=*"checkbox"* name=*"favorite"* value=*"用java"* id=*"favorite-2"* />  <label for=*"favorite-2"* class=*"checkboxLabel"*>  用java  </label>  <input type=*"checkbox"* name=*"favorite"* value=*"教java"* id=*"favorite-3"* />  <label for=*"favorite-3"* class=*"checkboxLabel"*>  教java  </label>  <input type=*"checkbox"* name=*"favorite"* value=*"发呆"* id=*"favorite-4"* />  <label for=*"favorite-4"* class=*"checkboxLabel"*>  发呆  </label>  <input type=*"hidden"* id=*"\_\_multiselect\_test\_favorite"*  name=*"\_\_multiselect\_favorite"* value=*""* />  </td>  </tr> |

另一种常见用法：

|  |
| --- |
| <s:checkboxlist list="#{'1':'学java','2':'用java','3':'教java','4':'发呆'}" value=*"{'1','2'}"* name=*"favorite"* label=*"喜欢做的事"* ></s:checkboxlist> |
| 页面效果： |

### radio 单选按钮

对应HTML中的单选按钮radio。示例代码如下：

|  |
| --- |
| <s:radio list="#{'1':'男','0':'女'}" label=*"性别"* name=*"gender"* value=*"1"* ></s:radio> |
| <tr>  <td class=*"tdLabel"*>  <label for=*"test\_gender"* class=*"label"*>  性别:  </label>  </td>  <td>  <input type=*"radio"* name=*"gender"* id=*"test\_gender1"*  checked=*"checked"* value=*"1"* />  <label for=*"test\_gender1"*>  男  </label>  <input type=*"radio"* name=*"gender"* id=*"test\_gender0"* value=*"0"* />  <label for=*"test\_gender0"*>  女  </label>  </td>  </tr> |

### selet下拉框

生成HTML中的<select>标签。示例代码如下：

|  |
| --- |
| <s:select label=*"所在地"* list=*"{'北京','上海','深圳'}"*></s:select> |
| <tr>  <td class="tdLabel"><label for="test\_" class="label">所在地:</label></td>  <td>  <select name="" id="test\_">  <option value="北京">北京</option>  <option value="上海">上海</option>  <option value="深圳">深圳</option>  </select>  </td>  </tr> |
| 另一种写法：  <s:select label=*"所在地"* list="#{'1':'北京','2':'上海','3':'深圳'}"></s:select> |
| 生成的源代码： |

### autocompleter自动完成框

**使用autocompeter之前，一定要在页面中增加(代码中阴影显示的行)：**

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <%@ taglib prefix=*"sx"* uri=*"/struts-dojo-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <sx:head /> |

项目中引入：struts2-dojo-plugin-2.2.1.1.jar。 struts2中将所有ajax相关的内容都放入了这个包中。

Autocompleter简单示例：

|  |
| --- |
| <sx:autocompleter label=*"专业"* name=*"major"* list=*"{'计算机应用','计算机工程','计算机数学','英语专业','英语外贸'}"* /> |
| 执行效果，在框中输入文字： |

### autocompleter动态数据加载

实际中，使用autocomleter时，数据量通常较大，不可能一下子进行加载，需要我们使用ajax按需加载，类似于Google Suggest的效果。我们使用动态数据来完成更复杂的autocomplete的使用：

data.jsp功能是生成数据，代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  ['计算机软件','计算机应用','计算机理论','人工智能','英语口语','英语外贸','日语外贸','中文'] |

test.jsp代码如下：

|  |
| --- |
| <sx:autocompleter label=*"专业"* name=*"major"* href=*"data.jsp"*  loadOnTextChange=*"true"* loadMinimumCount=*"0"* autoComplete=*"false"*  showDownArrow=*"true"* /> |

效果：



### datetimepicker 标签日期选择器

典型使用代码：

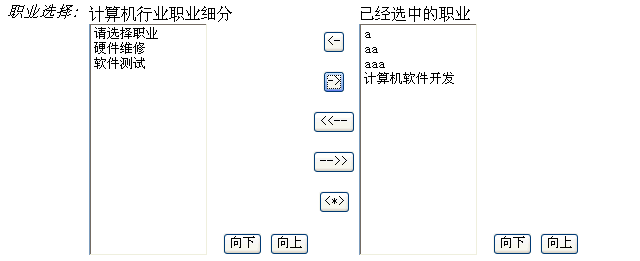
|  |
| --- |
| <sx:datetimepicker name=*"birthday"* displayFormat=*"yyyy-MM-dd"* label=*"输入出生日期"*></sx:datetimepicker> |

效果：

### doubleselect 联动下拉框

不太灵活，一般建议手动完成！ 作为扩展自学内容！

### optiontransferselect 选项移动下拉框



实现上面效果，代码如下：

|  |
| --- |
| <s:optiontransferselect label=*"职业选择"* name=*"alljob"* list=*"{'计算机软件开发','硬件维修','软件测试'}"* doubleList=*"{'a','aa','aaa'}"* doubleName=*"myjob"*  leftTitle=*"计算机行业职业细分"* rightTitle=*"已经选中的职业"*  leftDownLabel=*"向下"* leftUpLabel=*"向上"*  rightUpLabel=*"向上"* rightDownLabel=*"向下"*  headerKey=*""* headerValue=*"请选择职业"*  ></s:optiontransferselect> |

### optgroup 选项分组下拉框

作为扩展自学内容！写出文档！ 其实有html标签可以写：

|  |
| --- |
| <select>  <optgroup label=*"Swedish Cars"*>  <option value =*"volvo"*>Volvo</option>  <option value =*"saab"*>Saab</option>  </optgroup>    <optgroup label=*"German Cars"*>  <option value =*"mercedes"*>Mercedes</option>  <option value =*"audi"*>Audi</option>  </optgroup>  </select> |

效果如下：



### <sx:textarea /> 文本域

代码如下：

<sx:textarea label=*"简介"* name=*"aa"* value=*"ccc"* ></sx:textarea>

效果：



## 常用非表单UI标签

### fielderror、actionerror、actionmessage 输出错误信息

跟数据校验结合紧密。见数据校验相关内容！

### debug 输出调试信息

用于显示ValueStack、ActionContext中的信息。通常用于调试。典型代码：

<:s:debug />

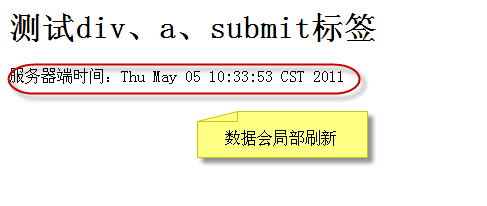
### div、submit、a

该标签用来实现局部刷新，即使用ajax方式更新div数据。

代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <%@ taglib prefix=*"sx"* uri=*"/struts-dojo-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>测试struts2</title>  </head>  <sx:head />  <body>  <h1>测试div、a、submit标签</h1>  <sx:div href=*"data.jsp"* updateFreq=*"3000"* ></sx:div>  </body>  </html> |

效果：



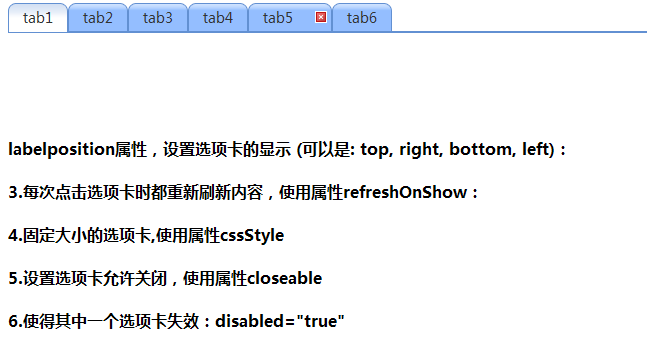
|  |
| --- |
| <h1>aaaaaaaa</h1>  <%--<sx:div href="reg.jsp" updateFreq="5000"></sx:div>每隔5秒请求一次 --%>  <%--<sx:div href="reg.jsp" delay="5000"></sx:div> 5秒后请求一次 --%>  <h1>bbbbbbbbb</h1>  <sx:a href=*"reg.jsp"* targets=*"aaa"* loadingText=*"数据正在加载..."*>aa</sx:a>  <sx:submit href=*"reg.jsp"* targets=*"aaa"* loadingText=*"数据正在加载..."*>aa</sx:submit>  <div id=*"aaa"*></div> |

### tabbedpanel tab页效果

代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"*%>  <%@ taglib prefix=*"sx"* uri=*"/struts-dojo-tags"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>My JSP 'Testtabbedpanel.jsp' starting page</title>  </head>  <sx:head />  <body>  <sx:tabbedpanel id=*"tab1"* labelposition=*"top"* >  <sx:div label=*"tab1"* >  </sx:div>  <sx:div label=*"tab2"* href=*"index.jsp"*>  </sx:div>  <sx:div label=*"tab3"* href=*"index.jsp"* refreshOnShow=*"true"*>  </sx:div>  <sx:div label=*"tab4"* href=*"index.jsp"*  cssStyle=*"width: 300px; height: 500px;"* >  </sx:div>  <sx:div label=*"tab5"* href=*"index.jsp"* closable=*"true"* preload=*"fasle"*>  </sx:div>  <sx:div label=*"tab6"* href=*"index.jsp"* disabled=*"true"*>  </sx:div>  </sx:tabbedpanel>  <br /><br /><br /><br /><br />  <b> labelposition属性，设置选项卡的显示 (可以是: top, right, bottom, left)：</b><br />  <br />  <b> 3.每次点击选项卡时都重新刷新内容，使用属性refreshOnShow：</b><br />  <br />  <b> 4.固定大小的选项卡,使用属性cssStyle</b><br />  <br />  <b> 5.设置选项卡允许关闭，使用属性closeable</b><br />  <br />  <b> 6.使得其中一个选项卡失效：disabled="true"</b><br />  <br />  </body>  </html> |

效果如下：

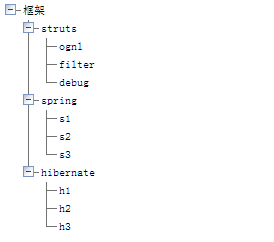


### tree 树结构

代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"utf-8"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <%@ taglib prefix=*"sx"* uri=*"/struts-dojo-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>tree</title>  </head>  <sx:head/>  <body>  <sx:tree id=*"struts"* label=*"框架"* showRootGrid=*"true"* showGrid=*"true"* toggleDuration=*"1000"*>  <sx:treenode label=*"struts"* id =*"1"*>  <sx:treenode label=*"ognl"* id=*"11"*/>  <sx:treenode label=*"filter"* id=*"12"* />  <sx:treenode label=*"debug"* id=*"13"* />  </sx:treenode>  <sx:treenode label=*"spring"* id =*"2"* >  <sx:treenode label=*"s1"* id=*"11"* />  <sx:treenode label=*"s2"* id=*"12"* />  <sx:treenode label=*"s3"* id=*"13"* />  </sx:treenode>  <sx:treenode label=*"hibernate"* id =*"3"* >  <sx:treenode label=*"h1"* id=*"11"* />  <sx:treenode label=*"h2"* id=*"12"* />  <sx:treenode label=*"h3"* id=*"13"* />  </sx:treenode>  </sx:tree>  </body>  </html> |

效果如下：

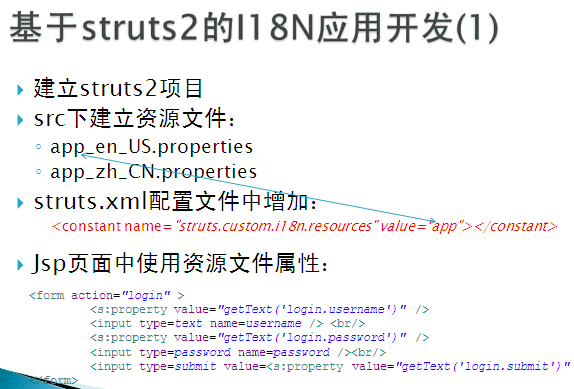


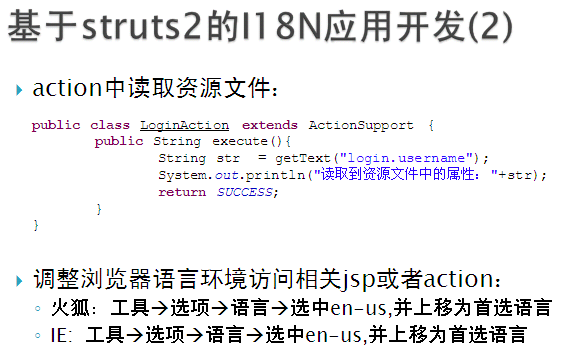
## 资源国际化

### 资源国际化开发技术

* 人们常把I18N作为“国际化”的简称，其来源是英文单词 internationalization的首末字符i和n。18为中间的字符数。
* 一个Web应用程序在运行时能够根据客户端请求所来自的国家和语言显示不同的用户界面。这样，当需要在应用程序中添加对一种新的语言的支持时，无需修改应用程序的代码。
* **注意：建立资源文件时，国家名称必须要大写！(例如：US,CN等)**







### 占位符的处理

**资源文件中的占位符：**

welcome.msg=welcome,**{0}**,**{1}**

Action中填充占位符示例：

String str = getText("welcome.msg",**new** String[]{"aaa","ccc"});

JSP中填充占位符示例：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"*%>  <s:text name=*"welcome.msg"*>  <s:param value=*"%{'aaa'}"* ></s:param>  <s:param value=*"%{'ccc'}"* ></s:param>  </s:text> |

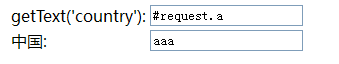
## %、#在OGNL中的用法

#: 从actioncontext中取值,前面加上#。相当于：ActionContext.getContext()。

%: %{#username}百分号的作用就是将大括号中的值当成ognl表达式。测试代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"*%>  <%  request.setAttribute("a","aaa");  %>  <s:form>  <s:textfield value=*"#request.a"* label=*"getText('country')"*></s:textfield>  <s:textfield value=*"%{#request.a}"* label=*"%{getText('country')}"* ></s:textfield>  </s:form> |

执行上面代码，显示效果如下：



# 第四篇 struts2 拦截器

## 什么是拦截器、拦截器栈?

Interceptor拦截器类似于我们前面学过的过滤器,是可以在action执行前后执行的代码。是我们做web开发时经常用的技术。比如：权限控制、日志等。我们也可以将多个Interceptor连在一起组成Interceptor栈。

Struts2拦截器，每个拦截器类只有一个对象实例，即采用单例模式，所有引用这个拦截器的Action都共享这一拦截器类的实例，因此，在拦截器中如果使用类变量，要注意同步问题。

* 拦截器是在访问某个方法，字段之前或之后实施拦截。
* 拦截器是AOP的一种实现
* 拦截器栈（Interceptor Stack）
  + 拦截器栈就是将拦截器按一定的顺序联结成一条链。
  + 在访问被拦截的方法或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。
* 实现原理
  + Struts2拦截器的实现原理相对简单，当请求struts2的action时，Struts 2会查找配置文件，并根据其配置实例化相对应拦截器对象，然后串成一个列表，最后一个一个地调用列表中的拦截器。

## 拦截器和过滤器的区别

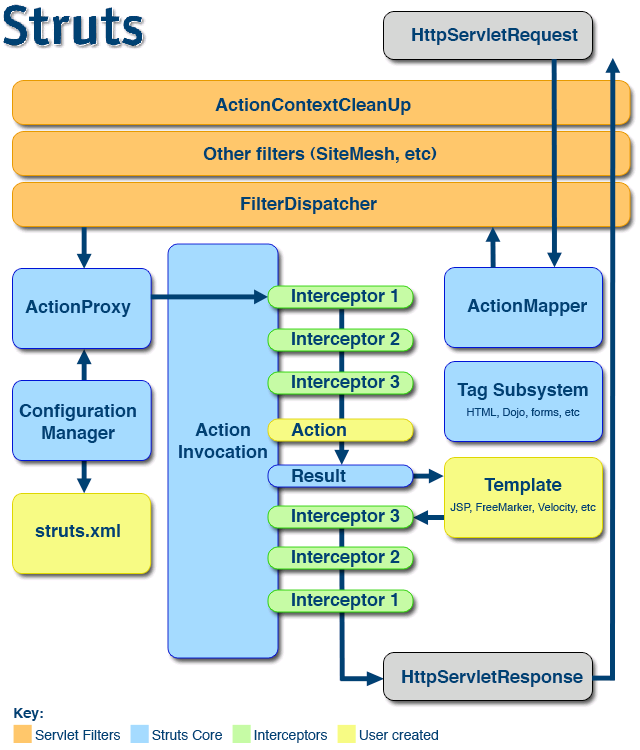
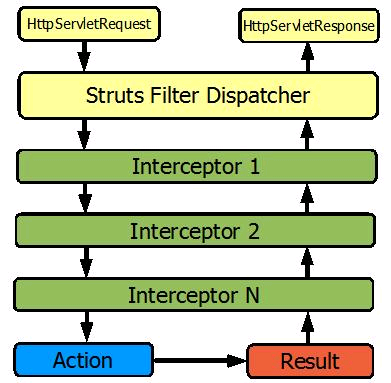
1. 拦截器和过滤器的概念非常类似

2. 过滤器是web中的概念，可以过滤一切对象(包括action、servlet、jsp、html等等)。

3. 而拦截器是struts2中的概念，只能拦截action(无法拦截对jsp的请求)。

4. 过滤器内部采用函数回调来实现。拦截器采用动态代理来实现！

## Struts2拦截器示意图

## 拦截器在struts2中的应用

* 对于Struts2框架而言，正是大量的内置拦截器完成了大部分操作。比如：
  + 像params拦截器将http请求中参数解析出来赋值给Action中对应的属性。
  + Servlet-config拦截器负责把请求中HttpServletRequest实例和HttpServletResponse实例传递给Action
  + …
* struts-default.xml中有一个默认的引用，在默认情况下（也就是<action>中未引用拦截器时）会自动引用一些拦截器。

## 默认拦截器说明

* alias：对于HTTP请求包含的参数设置别名。
* autowiring：将某些JavaBean实例自动绑定到其他Bean对应的属性中。有点类似Spring的自动绑定，在Spring部分会详细说明。
* chain：在Web项目开发中，以前使用Struts开发时候经常碰到两个Action互相传递参数或属性的情况。该拦截器就是让前一Action的参数可以在现有Action中使用。
* conversionError：从ActionContext中将转化类型时候发生的错误添加到Action的值域错误中，在校验时候经常被使用到来显示类型转化错误的信息。
* cookie：从Struts2.0.7版本开始，可以把cookie注入Action中可设置的名字或值中。
* createSession：自动创建一个HTTP的Session，尤其是对需要HTTP的Session的拦截器特别有用。比如下面介绍的TokenInterceptor。
* debugging：用来对在视图间传递的数据进行调试。
* execAndWait：不显式执行Action，在视图上显示给用户的是一个正在等待的页面，但是Action其实是在“背后”正在执行着。该拦截器尤其在对进度条进行开发的时候特别有用。
* exception：将异常和Action返回的result相映射。
* fileUpload：支持文件上传功能的拦截器。
* i18n：支持国际化的拦截器。
* logger：拥有日志功能的拦截器。
* modelDriven：Action执行该拦截器时候，可以将getModel方法得到的result值放入值栈中。
* scopedModelDriven：执行该拦截器时候，它可以从一个scope范围检索和存储model值，通过调用setModel方法去设置model值。
* params：将HTTP请求中包含的参数值设置到Action中。
* prepare：假如Action继承了Preparable接口，则会调用prepare方法。
* staticParams：对于在struts.xml文件中Action中设置的参数设置到对应的Action中。
* scope：在session或者application范围中设置Action的状态。
* servletConfig：该拦截器提供访问包含HttpServletResquest和HttpServletResponse对象的Map的方法。
* timer：输出Action的执行时间。
* token：避免重复提交的校验拦截器。
* tokenSession：和token拦截器类似，但它还能存储提交的数据到session里。
* validation：运行在action-validation.xml(校验章节介绍)文件中定义的校验规则。
* workflow：在Action中调用validate校验方法。如果Action有错误则返回到input视图。
* store：执行校验功能时候，该拦截器提供存储和检索Action的所有错误和正确信息的功能。
* checkbox：视图中如果有checkbox存在的情况，该拦截器自动将unchecked的checkbox当作一个参数（通常值为“false”）记录下来。这样可以用一个隐藏的表单值来记录所有未提交的checkbox，而且缺省unchecked的checkbox值是布尔类型的，如果视图中checkbox的值设置的不是布尔类型，它就会被覆盖成布尔类型的值。
* profiling：通过参数来激活或不激活分析检测功能，前提是Web项目是在开发模式下。（涉及到调试和性能检验时使用）
* roles：进行权限配置的拦截器，如果登录用户拥有相应权限才去执行某一特定Action。

## 计时拦截器

timer拦截器能够统计每个action的执行时间。原理很简单：就是在执行action前纪录一个时间，执行action后纪录一个时间。两个时间纪录相减即可获得action的执行时间。

timer拦截器使用方式非常简单：在action的配置中增加timer拦截器配置即可：

|  |
| --- |
| <package name=*"test"* namespace=*"/"* extends=*"struts-default"*>  <action name=*"testValidate"* class=*"com.bjsxt.struts.test.TestValidateAction"*>  <interceptor-ref name=*"timer"*></interceptor-ref>  <result name=*"success"*>/ok.jsp</result>  <result name=*"input"*>/testFormLabel.jsp</result>  </action>  </package> |

这样我们访问相应的action，控制台就会打印：

|  |
| --- |
| 2011-5-5 14:03:42 com.opensymphony.xwork2.util.logging.jdk.JdkLogger info  信息: Executed action [//testValidate!execute] took 78 ms. |

注：使用默认拦截器，我们的<package>元素必须要有：extends=*"struts-default"*

## token防止表单重复提交拦截器

表单重复提交，是我们在写系统时一定要避免的！尤其是在购物、银行等系统中。通过令牌机制防止表单的重复提交(我们在中级项目中已经做过！)。

在struts2中，我们可以通过token拦截器轻松实现防止表单重复提交功能！

1. 在相应的action配置中增加：

|  |
| --- |
| <package name=*"test"* namespace=*"/"* extends=*"struts-default"*>  <action name=*"testValidate"* class=*"com.bjsxt.struts.test.TestValidateAction"*>  **<interceptor-ref name=*"token"*></interceptor-ref>**  <result name=*"success"*>/ok.jsp</result>  <result name=*"input"*>/testFormLabel.jsp</result>  **<result name=*"invalid.token"*>/tokenInvalid.jsp</result>**  </action>  </package> |

2. 增加tokenInvalid.jsp文件,代码如下：

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"gbk"*%>  <h1>表单不能重复提交！！</h1> |

3. 在所提交的表单上增加:<s:token></s:token>标记。

|  |
| --- |
| <s:form action=*"testValidate"* >  <s:textfield name=*"uname"* ></s:textfield>  <s:token></s:token>  <s:submit></s:submit>  </s:form> |

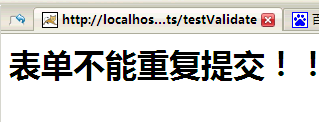
测试：



提交表单到相应的action，点击提交按钮后，顺利提交。地址栏变为：

<http://localhost:8080/teststruts/testValidate>

直接点击刷新按钮，刷新地址栏后，跳转到tokenInvalid.jsp：



## 自定义拦截器

* 直接或间接实现接口com.opensymphony.xwork2.interceptor.Interceptor
* 或者继承类com.opensymphony.xwork2.interceptor.AbstractInterceptor
* 通过<interceptor>元素来定义拦截器
* 通过<interceptor-ref>元素来使用拦截器
* 重写**public String intercept(ActionInvocation ai) throws Exception**
* 注意：
  + 如果为Action指定了一个拦截器，则系统默认的拦截器栈将会失去作用。为了继续使用默认拦截器，所以上面配置文件中手动引入了默认拦截器
* 取得Action对象，从而取得对Action的完全控制

|  |
| --- |
| **TestAction testaction = (TestAction) ai.getAction();**  **System.out.println("action中的属性id："+testaction.getId());** |

* 继续执行action，执行后对响应的处理

|  |
| --- |
| **ai.invoke();**  **System.out.println("执行完action后，action中的属性id："+testaction.getId());** |

* 取得ActionContext对象(存放Action执行时需要用到的对象)，将Servlet相关数据转换为与ServletAPI无关的Map对象（即ActionContext）

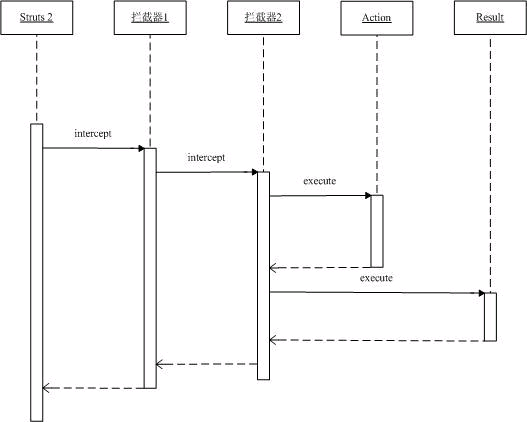
|  |
| --- |
| **ActionContext ac = ai.getInvocationContext();**  **System.out.println(ac.getSession().get("a")); //得到HttpSession的Map对象**  **Map requestmap = (Map) ac.get("request"); //得到HttpServletRequest的Map对象**  **System.out.println("-----"+requestmap.get("id"));**  **Map applicationMap=ac.getApplication(); //得到ServletContext的Map对象** |

* 获得servlet中相关对象

|  |
| --- |
| **HttpServletRequest req = ServletActionContext.*getRequest();***  **ServletActionContext.*getResponse();***  **ServletActionContext.*getServletContext();***  **ServletActionContext.*getRequest().getSession();*** |

## 拦截器执行顺序

* 根据拦截器执行情况：
  + 拦截方法之前执行：配置在前面的拦截器，先执行。
  + 拦截方法之前执行：配置在前面的拦截器，后执行。



## 方法拦截器

* **Struts2提供MethodFilterInterceptor类，该类是AbstractInerceptor的子类**，可以实现对Action方法的拦截.
* 重写：doIntercept方法
* MethodFilterInterceptor中参数配置：
  + **如果一个方法同时在excludeMethods和includeMethods中出现,则会被拦截**
  + **如果不配置这两个参数，所在action所有方法都会拦截**

|  |
| --- |
| **<interceptor name="testmethod" class="com.cssxt.interceptor.TestMethodInterceptor">**  **<param name="includeMethods">add,add2</param>**  **<param name="excludeMethods">add3,add4</param>**  **</interceptor>** |

## 文件上传

struts2为我们提供了简便的上传支持，内部仍然使用了apache commons fileupload组件。我们只需要按照如下步骤操作，即可轻松完成文件上传操作：

1. action代码如下：

|  |
| --- |
| package com.bjsxt.gao;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import com.opensymphony.xwork2.ActionSupport;  public class Upload2Action extends ActionSupport {  private String title;    private File upload; //封装上传文件域  private String uploadContentType; //封装上传文件类型  private String uploadFileName; //封装上传文件名    private String savePath;    //get和set方法，省略！  public String execute() throws Exception{  System.out.println(uploadFileName);  System.out.println(uploadContentType);  FileOutputStream fos = new FileOutputStream(getSavePath()+getUploadFileName());  FileInputStream fis = new FileInputStream(getUpload());  byte[] buf = new byte[1024];  int len = 0;  while((len = fis.read(buf))>0){  fos.write(buf, 0, len);  }    fos.close();  fis.close();    return SUCCESS;    }  } |

2. 配置action：

|  |
| --- |
| <action name="Upload" class="com.bjsxt.gao.Upload2Action">  <param name="savePath">d:/temp/</param> <!-- 可以通过这种方式在配置文件中设置action属性的值 -->  <result name="input">/index.jsp</result>  <result name="error">/error.jsp</result>  <result name="success">/ok.jsp</result>  </action> |

3. jsp页面如下：

|  |
| --- |
| <form action="Upload.action" enctype="multipart/form-data" method=post >  文件标题:<input type=text name=title /> <br/>  选择文件：<input type=file name=upload /><br/>  <input type=submit value=上传 />  </form> |

通过上面的步骤。我们可以发现struts2中的上传处理。所上传的文件很直观的封装到对应的File对象中。

**自学：**

1. 如何上传多个文件！(提示：File数组即可实现)

2. 增加文件类型、上传文件最大大小的限制验证！

3. struts2上传进度条的实现

## 拦截器栈

定义拦截器栈：

<interceptor-stack name=*"stack1"*>

<interceptor-ref name=*"timer"*></interceptor-ref>

<interceptor-ref name=*"my2"*></interceptor-ref>

</interceptor-stack>

直接在action中引用即可：

<action name=*"test3"* class=*"com.bjsxt.test.Test3Action"*>

<interceptor-ref name=*"stack1"*></interceptor-ref>

<result name=*"success"*>/ok.jsp</result>

</action>

## 默认拦截器(栈)

* + 1. 默认拦截器只针对<package>中没有手动配置拦截器的action有效！
    2. 包中定义的默认拦截器生效，则struts-default包中定义的默认拦截器失效。

# 第一篇Hibernate基础

## 持久化、持久层

持久化，persistence，就是把数据保存到可掉电式存储设备中供之后使用。大多数情况下特别是企业级应用，数据持久化往往也就意味着将内存中的数据保存到磁盘上加以固化，而持久化的实现过程则大多通过各种关系数据库来完成。

对于应用系统而言，数据持久功能大多是必不可少的组成部分。

所谓“持久层”，也就是在系统逻辑层面上，专著于实现数据持久化的一个相对独立的领域(Domain)。

持久层是负责向（或者从）一个或者多个数据存储器中存储（或者获取）数据的一组类和组件。这个层必须包括一个业务领域实体的模型（即使只是一个元数据模型）。

## DAO模式

DAO是Data Access Object数据访问接口，数据访问：顾名思义就是与数据库打交道。夹在业务逻辑与数据库资源中间。

为了建立一个健壮的J2EE应用，应该将所有对数据源的访问操作抽象封装在一个公共API中。用程序设计的语言来说，就是建立一个接口，接口中定义了此[应用程序](http://baike.baidu.com/view/330120.htm)中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口在逻辑上对应这个特定的数据存储。

　 J2EE开发人员使用数据访问对象(DAO)设计模式把底层的数据访问逻辑和高层的商务逻辑分开.实现DAO模式能够更加专注于编写数据访问代码.

事实上，我们在前面web项目中已经大量使用了DAO模式

## ORM是什么?

对象-关系映射（Object/Relation Mapping，简称ORM），是随着面向对象的[软件开发方法](http://baike.baidu.com/view/1471861.htm)发 展而产生的。面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系数据库是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数 据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法 直接表达多对多关联和继承关系。因此，对象-关系映射(ORM)系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

面向对象是从软件工程基本原则(如耦合、聚合、封装)的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象,对象关系映射技术应运而生。

让我们从O/R开始。字母O起源于"对象"(Object),而R则来自于"关系"(Relational)。几乎所有的程序里面，都存在对象和关系数据 库。在业务逻辑层和用户界面层中，我们是面向对象的。当对象信息发生变化的时候，我们需要把对象的信息保存在关系数据库中。

总结：

* + 将关系数据库中的数据转化成对象，这样，开发人员就可以以一种完全面向对象的方式来实现对数据库的操作。
  + 我们以前讲JAVA对象的数据存入数据库，需要通过JDBC进行繁琐的转换，反之亦然。ORM框架，比如Hibernate就是将我们这部分工作进行了封装，简化了我们的操作。

## 常见ORM框架比较和介绍

|  |  |  |
| --- | --- | --- |
|  | 优势 | 劣势 |
| Entity EJB | EJB作为一种重量级、高花费的ORM技术，具有不可比拟的优势。就其架构设计来讲，依然是非常优秀，而且值得借鉴的。最新的EJB3.1也采用了敌侵入式设计，增加了Annotation支持，具有极大吸引力。 | 必须运行在EJB容器内，而且学习曲线陡峭，开发周期、成本相对较高，因而限制EJB的广泛使用。 |
| Hibernate | 目前最流行的开源ORM框架。Hibernate灵巧的设计、优秀的性能，还有丰富的文档的都是其风靡全球的重要因素。  a.Hibernate使用 Java 反射机制 而不是字节码增强程序来实现透明性。 b.Hibernate的性能非常好，因为它是个轻量级框架。 映射的灵活性很出色。 c.它支持各种关系数据库，从一对一到多对多的各种复杂关系。 | 它限制您所使用的对象模型。(例如，一个持久性类不能映射到多个表) |
| IBATIS | 特别是对一些数据库访问特别灵活的地方，IBATIS更加灵活，它允许开发人员直接编写SQL语句。 | 不支持纯粹的面向对象操作 |
| TopLink | TOPLink为在关系数据库表中存储 Java 对象和企业 Java 组件 (EJB) 提供高度灵活和高效的机制。TopLink 为开发人员提供极佳的性能和选择，可以与任何数据库、任何应用服务器、任何开发工具集和过程以及任何 J2EE 体系结构协同工作。 | 作为商业产品使用，需要收取费用 |
| OJB | 1.灵活性  2.可缩放性  3.强大的功能性 | Hibernate光芒太盛，OJB开发文档不是很多，影响了OJB的流行 |

## JPA是什么?他和Hibernate什么关系?

JPA全称Java Persistence API. JPA通过JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体[对象持久化](http://baike.baidu.com/view/402359.htm)到数据库中。

Sun引入新的JPA ORM规范出于两个原因：其一，简化现有Java EE和Java SE应用的[对象持久化](http://baike.baidu.com/view/402359.htm)的开发工作；其二，Sun希望整合对ORM技术，实现天下归一。

JPA的总体思想和现有Hibernate、TopLink，JDO等ORM框架大体一致。

JPA是SUN官方提出的持久化的规范。Hibernate、Spring、OpenJPA、Toplink等是JPA的实现。

Hibernate是JPA的具体实现。但是Hibernate出现的时间早于JPA。JPA是SUN在持久化框架发展起来后提出的规范。Hibernate 从3.2开始，就开始兼容JPA。Hibernate3.2获得了Sun TCK的JPA(Java Persistence API) 兼容认证。

EJB3.0和JPA是javaee5.0主要的亮点！

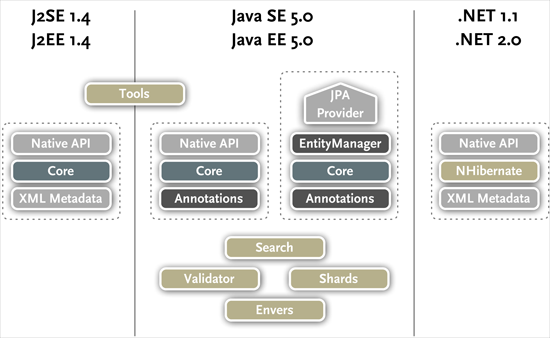
## Hibernate介绍

Hibernate是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲的使用对象编程思维来 操纵数据库。 Hibernate可以应用在任何使用JDBC的场合，既可以在Java的客户端程序实用，也可以在Servlet/JSP的Web应用中使用，最具革命 意义的是，Hibernate可以在应用EJB的J2EE架构中取代CMP，完成数据持久化的重任。

官方网站：<http://www.hibernate.org/>

**Relational Persistence for Java and .NET**

[Historically](http://www.hibernate.org/hibernate/about/history.html), Hibernate facilitated the storage and retrieval of Java domain objects via [Object/Relational Mapping](http://www.hibernate.org/hibernate/about/orm.html).  Today, Hibernate is a collection of related projects enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping.



## Hibernate优势

* 简化开发，提高开发效率。

尤其是持久层关系映射复杂情况下，hibernate开发效率相当高。

* 更加面向对象设计
* 更好的性能

如果JDBC代码写的完美，优化做好，那么JDBC效率是最高的。但是，实际开发中非常不现实，对程序员要求太高。

一般情况下，hibernate内部进行了JDBC的优化处理，以及增加缓存机制，大大提高了运行效率。

* 更好的移植性

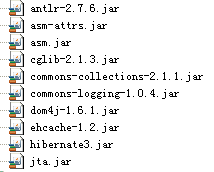
Hibernate已经为我们屏蔽了数据库之间的差异。

## 版本历史

* + Hibernate3是一个跨越。Hibernate3以前的版本，用的人很少。 也是一个非常成熟的框架。最新的是3.6,我们学的是3.3。Hibernate4.0 alpha版也已发布。

## 第一个Hibernate程序

1. 官网下载相关内容：
   * [www.hibernate.org](http://www.hibernate.org/)
2. 建立Java Project
   * 增加hibernate开发需要的jar包：
     + hibernate.jar和lib下如下所示的jar包(或者将lib下所有jar包导入)



* + - 再增加mysql的驱动包或者oracle的驱动包

注：可参考：附录🡪Hibernate依赖库中相关jar包作用

1. 增加配置文件
   * 打开etc目录，将hibernate.cfg.xml拷贝到项目src下
   * 修改配置文件hibernate.cfg.xml内容，结合etc/hibernate.properties（文件中搜索”mysql”），完成后配置内容如下：

|  |
| --- |
| <!DOCTYPE hibernate-configuration PUBLIC常常  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name=*"show\_sql"*>true</property>  <property name=*"hibernate.dialect"*>  org.hibernate.dialect.MySQLDialect  </property>  <property name=*"connection.driver\_class"*>  com.mysql.jdbc.Driver  </property>  <property name=*"hibernate.connection.url"*>  jdbc:mysql://localhost:3306/testhib  </property>  <property name=*"connection.username"*>  root  </property>  <property name=*"connection.password"*>  123456  </property>  </session-factory>  </hibernate-configuration> |

注：

1. 我们需要新建数据库testhib
2. <session-factory name=*"foo"*> 把name属性去掉，不然会报Could not bind factory to JNDI javax.naming.NoInitialContextException. 多了name属性hibernate会尝试将sessionfactory注册到jndi上，因此报错。
3. 新建pojo类(Plain Old Java Objects 简单的java对象，实际上就是我们讲的普通javabean对象)：User

|  |
| --- |
| **package** com.bjsxt.hib.bean;  **import** java.util.Date;  **public** **class** User {  **private** **int** id;  **private** String name;  **private** Date birthday;    **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Date getBirthday() {  **return** birthday;  }  **public** **void** setBirthday(Date birthday) {  **this**.birthday = birthday;  }  **public** User(**int** id, String name, Date birthday) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.birthday = birthday;  }  **public** User(String name, Date birthday) {  **super**();  **this**.name = name;  **this**.birthday = birthday;  }    **public** User() {  }  } |

1. 增加映射文件User.hbm.xml（写法可以参考：eg/User.hbm.xml）

映射文件hbm.xml说明了pojo类和表的对应关系，以及pojo类中属性和表中字段的对应关系。

注：本映射文件增加到跟pojo同一个包中

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.bjsxt.hib.bean"*>  <class name=*"User"* table=*"\_user"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*/>  </id>  <property name=*"name"* type=*"java.lang.String"*  not-null=*"true"*  length=*"15"*  column=*"\_name"*/>  <property name=*"birthday"* type=*"java.util.Date"* column=*"birthday"*/>  </class>  </hibernate-mapping> |

1. 在hibernate.cfg.xml中增加User.hbm.xml文件的配置，让hibernate知道本映射关系。在<session-factory>元素下增加：

|  |
| --- |
| <mapping resource=*"com/bjsxt/hib/bean/User.hbm.xml"*/> |

1. 修改hibernate.cfg.xml文件，在<session-factory>下增加hbm2ddl.auto的配置：(数据定义语言 建表 建约束等)

|  |
| --- |
| <property name=*"hibernate.hbm2ddl.auto"*>update</property> |

* create-drop: 运行时，先创建，运行完，在删除。
* create:每次运行前都会删除已有的。在创建。 **测试时，可以使用create.**
* update:映射文件和表。不会重新创建表及不会重新执行ddl语句，只会更新表中的记录。
* validate:看映射文件和表是不是对应，如果不对应，他也不会更新，会报错。经常用它，保险一些。

1. 增加Test.java测试类：

|  |
| --- |
| **package** com.bjsxt.hib.test;  **import** java.util.Date;  **import** org.hibernate.Session;  **import** org.hibernate.SessionFactory;  **import** org.hibernate.cfg.Configuration;  **import** com.bjsxt.hib.bean.User;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Configuration conf = **new** Configuration().configure();  SessionFactory factory = conf.buildSessionFactory();  Session session = factory.openSession();  User user = **new** User("高淇",**new** Date());  session.save(user);  session.close();  }  } |

1. 运行Test.java，结果如下：

|  |
| --- |
| 2011-3-1 0:50:19 org.hibernate.cfg.Environment <clinit>  信息: Hibernate 3.2.0  2011-3-1 0:50:19 org.hibernate.cfg.Environment <clinit>  信息: hibernate.properties not found  2011-3-1 0:50:19 org.hibernate.cfg.Environment buildBytecodeProvider  信息: Bytecode provider name : cglib  2011-3-1 0:50:19 org.hibernate.cfg.Environment <clinit>  信息: using JDK 1.4 java.sql.Timestamp handling  2011-3-1 0:50:19 org.hibernate.cfg.Configuration configure  信息: configuring from resource: /hibernate.cfg.xml  2011-3-1 0:50:19 org.hibernate.cfg.Configuration getConfigurationInputStream  信息: Configuration resource: /hibernate.cfg.xml  2011-3-1 0:50:19 org.hibernate.cfg.Configuration addResource  信息: Reading mappings from resource: com/bjsxt/hib/bean/User.hbm.xml  2011-3-1 0:50:19 org.hibernate.cfg.HbmBinder bindRootPersistentClassCommonValues  信息: Mapping class: com.bjsxt.hib.bean.User -> \_user  2011-3-1 0:50:19 org.hibernate.cfg.Configuration doConfigure  信息: Configured SessionFactory: null  2011-3-1 0:50:19 org.hibernate.connection.DriverManagerConnectionProvider configure  信息: Using Hibernate built-in connection pool (not for production use!)  2011-3-1 0:50:19 org.hibernate.connection.DriverManagerConnectionProvider configure  信息: Hibernate connection pool size: 20  2011-3-1 0:50:19 org.hibernate.connection.DriverManagerConnectionProvider configure  信息: autocommit mode: false  2011-3-1 0:50:19 org.hibernate.connection.DriverManagerConnectionProvider configure  信息: using driver: com.mysql.jdbc.Driver at URL: jdbc:mysql://localhost:3306/testhib  2011-3-1 0:50:19 org.hibernate.connection.DriverManagerConnectionProvider configure  信息: connection properties: {user=root, password=\*\*\*\*}  2011-3-1 0:50:19 org.hibernate.cfg.SettingsFactory buildSettings  信息: RDBMS: MySQL, version: 5.1.29-rc-community  2011-3-1 0:50:19 org.hibernate.cfg.SettingsFactory buildSettings  信息: JDBC driver: MySQL-AB JDBC Driver, version: mysql-connector-java-3.1.10 ( $Date: 2005/05/19 15:52:23 $, $Revision: 1.1.2.2 $ )  2011-3-1 0:50:20 org.hibernate.dialect.Dialect <init>  信息: Using dialect: org.hibernate.dialect.MySQLDialect  2011-3-1 0:50:20 org.hibernate.transaction.TransactionFactoryFactory buildTransactionFactory  信息: Using default transaction strategy (direct JDBC transactions)  2011-3-1 0:50:20 org.hibernate.transaction.TransactionManagerLookupFactory getTransactionManagerLookup  信息: No TransactionManagerLookup configured (in JTA environment, use of read-write or transactional second-level cache is not recommended)  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Automatic flush during beforeCompletion(): disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Automatic session close at end of transaction: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: JDBC batch size: 15  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: JDBC batch updates for versioned data: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Scrollable result sets: enabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: JDBC3 getGeneratedKeys(): enabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Connection release mode: auto  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Maximum outer join fetch depth: 2  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Default batch fetch size: 1  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Generate SQL with comments: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Order SQL updates by primary key: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory createQueryTranslatorFactory  信息: Query translator: org.hibernate.hql.ast.ASTQueryTranslatorFactory  2011-3-1 0:50:20 org.hibernate.hql.ast.ASTQueryTranslatorFactory <init>  信息: Using ASTQueryTranslatorFactory  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Query language substitutions: {}  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: JPA-QL strict compliance: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Second-level cache: enabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Query cache: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory createCacheProvider  信息: Cache provider: org.hibernate.cache.NoCacheProvider  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Optimize cache for minimal puts: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Structured second-level cache entries: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Echoing all SQL to stdout  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Statistics: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Deleted entity synthetic identifier rollback: disabled  2011-3-1 0:50:20 org.hibernate.cfg.SettingsFactory buildSettings  信息: Default entity-mode: pojo  2011-3-1 0:50:20 org.hibernate.impl.SessionFactoryImpl <init>  信息: building session factory  2011-3-1 0:50:20 org.hibernate.impl.SessionFactoryObjectFactory addInstance  信息: Not binding factory to JNDI, no JNDI name configured  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.SchemaUpdate execute  信息: Running hbm2ddl schema update  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.SchemaUpdate execute  信息: fetching database metadata  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.SchemaUpdate execute  信息: updating schema  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.TableMetadata <init>  信息: table found: testhib.\_user  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.TableMetadata <init>  信息: columns: [id, birthday, \_name]  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.TableMetadata <init>  信息: foreign keys: []  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.TableMetadata <init>  信息: indexes: [primary]  2011-3-1 0:50:20 org.hibernate.tool.hbm2ddl.SchemaUpdate execute  信息: schema update complete  Hibernate: insert into \_user (\_name, birthday) values (?, ?) |

查看mysql数据库，我们发现表已经创建成功，但是数据未插入表中。



1. 我们发现执行时，hibernate打印了过多的信息。为了解决这个问题，我们将Log4j配置的项目中，然后调整打印级别。步骤如下：
2. 将log4j.jar拷贝到项目类库中
3. 将etc/log4j.properties文件copy到项目的src目录下，并调整打印级，内容如下：

|  |
| --- |
| ### direct log messages to stdout ###  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.Target=System.out  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c**{1}**:%L - %m%n  log4j.rootLogger=warn, stdout  log4j.logger.org.hibernate=warn  log4j.logger.org.hibernate.type=debug  log4j.logger.org.hibernate.tool.hbm2ddl=debug |

1. 上一次执行，我们发现表创建成功但是数据记录并没有插入表中。jdbc是自动提交，autocommit。hibernate缺省是false. 因此，我们必须很明确的开启事务才行。我们将Test.java文件内容修改如下：

|  |
| --- |
| package com.bjsxt.hib.test;  import java.util.Date;  import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.Transaction;  import org.hibernate.cfg.Configuration;  import com.bjsxt.hib.bean.User;  public class Test {  public static void main(String[] args) {  Configuration conf = new Configuration().configure();  SessionFactory factory = conf.buildSessionFactory();  Session session = factory.openSession();  **Transaction transaction = session.beginTransaction();**    User user = new User("高淇",new Date());  session.save(user);    transaction.commit();//保存之后，关闭session之前，显示提交事务.(必须调用，不然不会提交。即使session关闭，也不回自动提交)  session.close();  }  } |

控制台打印：

|  |
| --- |
| 01:02:10,421 INFO SchemaUpdate:115 - Running hbm2ddl schema update  01:02:10,421 INFO SchemaUpdate:126 - fetching database metadata  01:02:10,421 INFO SchemaUpdate:138 - updating schema  01:02:10,437 INFO TableMetadata:40 - table found: testhib.\_user  01:02:10,437 INFO TableMetadata:41 - columns: [id, birthday, \_name]  01:02:10,437 INFO TableMetadata:43 - foreign keys: []  01:02:10,437 INFO TableMetadata:44 - indexes: [primary]  01:02:10,437 INFO SchemaUpdate:160 - schema update complete  Hibernate: insert into \_user (\_name, birthday) values (?, ?)  01:02:10,515 DEBUG StringType:80 - binding '高淇' to parameter: 1  01:02:10,515 DEBUG TimestampType:80 - binding '2011-03-01 01:02:10' to parameter: 2 |

打开数据库，我们发现刚刚的pojo数据已经插入数据库中：



## Hibernate程序开发流程总结

1. 增加相关jar包
2. Src下增加hibernate.cfg.xml
3. 增加pojo和对应的映射文件，并将映射文件告诉hibernate.cfg.xml
4. 增加log4j配置(可选，有了log4j，我们可以自由调整调试信息打印级别)
5. 开始开发程序，基本代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {    Configuration conf = **new** Configuration().configure();  SessionFactory factory = conf.buildSessionFactory();  Session session = factory.openSession();  Transaction transaction = session.beginTransaction();    User user = **new** User("高淇",**new** Date());  session.save(user);    transaction.commit();//保存之后，关闭session之前，显示提交事务.(必须调用，不然不会提交。即使session关闭，也不回自动提交)  session.close();  } |

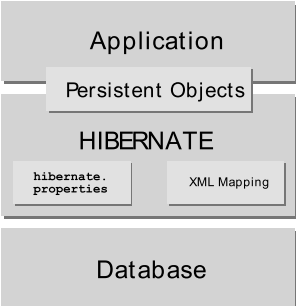
## 使用hibernate的项目开发流程对比

* **从Domain Object**🡪**mapping**🡪**db.（官方推荐）**
* 从数据库开始，用工具生成mapping和Domain Object**（实际使用较多）**
* 由映射文件开始。(理论上可行，实际没人用！)

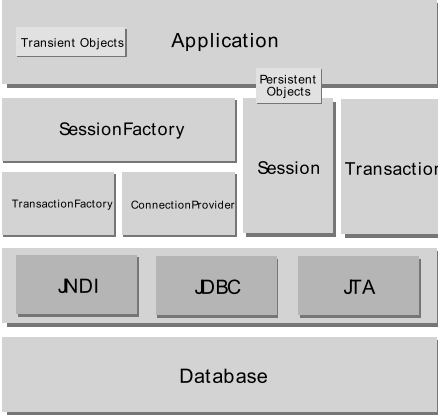
注：等我们将hibernate基本内容学完后，会告诉大家如何完成上面三种流程的开发！

## Hibernate架构

下图提供了hibernate体系的高层视图：



Hibernate全面解决方案：



## Hibernate核心API

### Configuration

负责管理数据库的配置信息。数据库的配置信息包含了Hibernate连接数据库的一些基本信息(**hibernate.cfg.xml**)，Configuration对象可以通过加载配置文件获取并管理这些信息.

Configuration config = new Configuration().configure();

或者：

File file = new File(“C:\\myhibernate.xml”);

Configuration config = new Configuration().configure(file);

### SessionFactory (org.hibernate.SessionFactory)

针对单个数据库映射关系经过编译后的内存镜像，是线程安全的（不可变）。 作为 Session

的工厂和 ConnectionProvider 的客户。SessionFactory 可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

SessionFactory使用要点如下：

1. 负责创建Session对象，可以通过Configuration对象创建SessionFactory对象
2. SessionFactory 对象中保存了当前的数据库配置信息和所有映射关系以及预定义的SQL语句。
3. SessionFactory还负责维护Hibernate的二级缓存。
4. SessionFactory对象的创建会有较大的开销，而且SessionFactory对象采取了线程安全的设计方式，因此在实际中SessionFactory对象可以尽量的共享，在大多数情况下，一个应用中针对一个数据库可以共享一个SessionFactory实例

SessionFactory创建代码如下：

Configuration config = new Configuration().configure();

SessionFactory factory = config.buildSessionFactory();

### Session (org.hibernate.Session)

表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短。其隐藏了

JDBC 连接，也是 Transaction 的工厂。它会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。

Session定义了添加、更新、删除和查询等操作，是持久化操作的基础。Session的设计是非线程安全的，因此，一个Session对象只可以由一个线程使用。

Session对象可以由SessionFactory对象创建。

Configuration config = new Configuration().configure();

SessionFactory factory = config.buildSessionFactory();

Session session = factory.openSession();

### Transaction

将应用代码从底层的事务实现中抽象出来——这可能是一个JDBC事务，一个JTA用户事务或者甚至是一个公共对象请求代理结构（CORBA）——允许应用通过一组一致的API控制事务边界。

使用Hibernate进行操作时（增、删、改）必须显示的调用Transaction（默认：autoCommit=false）

Transaction tx = session.beginTransaction();

## 标准的Hibernate代码

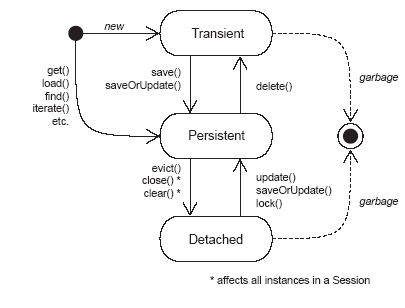
* 封装HibUtil类
  + SessionFactory只能有一个**（单例模式或者变为static属性）**，大家共享使用。

|  |
| --- |
| **public** **class** HibUtil {  **private** **static** SessionFactory *factory*;    **private** HibUtil(){  }  **static** {  Configuration configuration = **new** Configuration();  // configuration.configure("hibernate22.cfg.xml");  configuration.configure();  *factory* = configuration.buildSessionFactory();  }  **public** **static** SessionFactory getFactory() {  **return** *factory*;  }  **public** **static** Session getSession(){  **return** *factory*.openSession();  }  } |

Test2.java中示例代码如下：

|  |
| --- |
| **package** com.bjsxt.hib.test;  **import** java.util.Date;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** com.bjsxt.hib.bean.User;  **public** **class** Test2 {  **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();  session.save(**new** User("高淇",**new** Date()));  tx.commit();  }/\* catch (Exception e) { //省掉catch块和不省的写法效果是一样的。  //没有catch他也会向外抛出。数据库没有收到正常提交也会自动回滚。  if(tx!=null){  tx.rollback();  }  throw e; //一定要把这个异常报告到上一层。  }\*/**finally**{  **if**(session!=**null**){  session.close();  }  }  }  } |

## Hibernate对象的状态和生命周期



* 注意：
  + find、iterate方法在hibernate3后已经取消。
  + 使用debug详细观察程序运行，加深理解。

### Transient 瞬时/临时态

* 使用new 操作符初始化的对象不是立刻就持久的。
* 跟session没有任何关系
* 跟数据库没有任何关系，数据库中没有对应记录存在

### Persist 持久态

* 和session对象相关。以map形式存入到session中
* 在数据库中有对应的记录

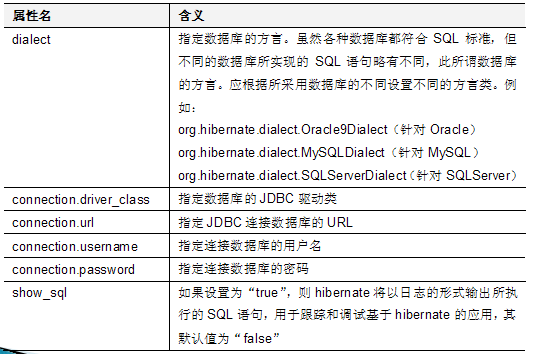
### Detached托管/游离态

* 和session对象无关
* 在数据库中有对应的记录

### 通过代码测试三种状态之间的转换和区别

|  |
| --- |
| public class Test3 {    /\*\*  \* 测试从瞬时对象到持久对象  \*/  public static void testSave(){  Session s = null;  Transaction ts = null;  try {  s = HibUtil.getSession();  ts= s.beginTransaction();    User u = new User("sss",new Date());  s.save(u);  // s.saveOrUpdate(u);  u.setName("s"); //处于持久化状态的对象，当发生值得改变时，hibernate能检测到，立刻修改数据库的数据。如果在save之后，会执行两条sql语句，效率较低。建议对持久对象值进行修改在save之前最好！  u.setName("ooo");  ts.commit();  } finally{  if(s!=null){  s.close();  }  }  }    /\*\*  \* 测试从持久状态到瞬时状态  \*/  public static void testDelete(){  Session s = null;  Transaction ts = null;  try {  s = HibUtil.getSession();  ts= s.beginTransaction();    User u = new User("sss",new Date());  User u2 = new User("ttt",new Date());  s.save(u);  s.delete(u);  u.setName("s");// u已经变成瞬时状态，他的状态更新不会同步到数据库中。  s.save(u2);  ts.commit();  } finally{  if(s!=null){  s.close();  }  }  }  /\*\*  \* 测试从持久状态到游离状态的互相转化  \*/  public static void testPersistent2Deteched(){  Session s = null;  Transaction ts = null;  try {  s = HibUtil.getSession();  ts= s.beginTransaction();    User u = new User("sss",new Date());  User u2 = new User("ttt",new Date());  s.save(u); //debug一下！  s.save(u2);    s.evict(u); //从session 中清除对象u.执行delete方法变成瞬时状态时，会执行delete语句将数据库中对应的记录删除；但此时，数据库仍然存在相应的记录；  u.setName("s2");  u.setName("s3");  u.setName("s4");  // s.save(u); //插入一条新的记录。  s.update(u); //修改数据库中对应的记录  s.saveOrUpdate(u);    // s.clear(); //将session中所有对象清除      ts.commit();  } finally{  if(s!=null){  s.close();  }  }  }  /\*\*  \* 直接得到持久化对象,测试get和load方法！  \*/  public static void testGetLoad(){  Session s = null;  Transaction ts = null;  try {  s = HibUtil.getSession();  ts= s.beginTransaction();    //若对应id=1的记录不存在，返回null;  //get方法先在session缓存(一级缓存)中查找，若无，再在sessionfacotory(二级缓存)中查找，若无，再去数据库中查找，还无，返回null.  // User u = (User) s.get(User.class, 1);    //若对应id=1的记录不存在，则抛出异常。  //load一般用于我们可以保证这个记录一定存在的情况。该方法有懒加载！  User u = (User) s.load(User.class, 1);  System.out.println(u.getName());  u.setName("hhh");    ts.commit();  } finally{  if(s!=null){  s.close();  }  }  }      public static void main(String[] args) {  // testSave();  // testDelete();  // testPersistent2Deteched();    testSave();  testGetLoad();  }  } |

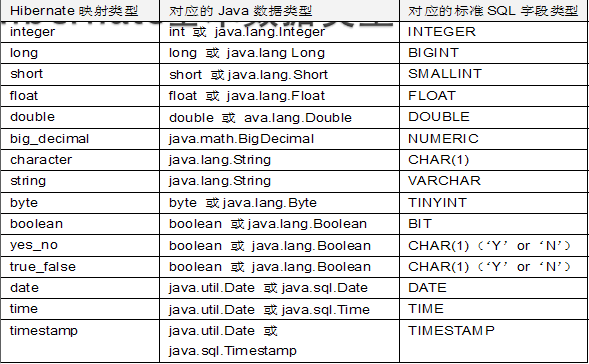
## Hibernate配置文件hibernate.cfg.xml详解



## Hibernate映射文件

|  |
| --- |
| <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name="Customer" table="CUSTOMER" schema="SCOTT">  <id name="id" type="java.lang.Long">  <column name="ID" precision="6" scale="0" />  <generator class="sequence" >  <param name="sequence">cust\_seq</param>  </generator>  </id>  <property name="name" type="string">  <column name="NAME" length="20" not-null="true" />  </property>  <property name="birthday" type="java.util.Date">  <column name="BIRTHDAY" length="7" not-null="true" />  </property>  </class>  </hibernate-mapping> |

## Hibernate基本数据类型



## Hibernate主键生成方式

* **native(很常用)**
  + 对于 oracle 采用 Sequence 方式，对于MySQL 和 SQL Server 采用identity（自增主键生成机制），native就是将主键的生成工作交由数据库完成，会根据底层数据库的能力，从identity、sequence、hilo中选择一个，灵活性更强 hibernate不管
* **assigned(很常用)**
  + 在插入数据的时候主键由用户自己添加，hibernate不管
* sequence
  + Oracle很常用。
* identity
  + 使用SQL Server 和 MySQL 的自增字段，这个方法不能放到 Oracle 中，Oracle 不支持自增字段，要设定sequence（MySQL 和 SQL Server 中很常用）
* hilo
* **uuid(Universally Unique Identifier)**

hibernate2中uuid分为:uuid.string和uuid.hex。hibernate3以后不再支持uuid.string，只有uuid.hex。

uuid使用一个128-bit的UUID算法生成字符串类型的标识符，UUID被编码成一个32位16进制数字的字符串。

UUID包含：IP地址、JVM启动时间、系统时间（精确到1/4秒）和一个计数器值（JVM中唯一）hibernate会算出一个128位的唯一值插入。

UUID 的目的，是让[分布式系统](http://baike.baidu.com/view/991489.htm)中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识资讯的指定。如此一来，每个人都可以建立不与其它人冲突的 UUID。在这样的情况下，就不需考虑数据库建立时的名称重复问题。

**UUID经常用来生成主键值完全唯一的情况，即全局唯一标识！**

将User的id生成方式修改成uuid。需要修改两个地方，pojo类和映射文件：

|  |
| --- |
| **public** **class** User {  **private** String id;  **private** String name;  **private** Date birthday;  //getter、settter方法限于篇幅，不列出  } |
| 修改User.hbm.xml中id的生成方式：  <id name=*"id"*>  <generator class=*"uuid"*/>  </id> |

执行Test2.java，然后观察数据库的变化：



**注：**如果hbm2ddl.auto=update，那么由于不会重新执行DDL语句，表结构仍然是上次的结构，从而不能正常插入数据。 请将数据库中的表删除，重新生成即可。或者修改hbm2ddl.auto=create.

## 写出自己的Hibernate

通过反射机制和JDBC技术，以及结合我们现在讲到的持久化的概念，写一个简易的Hibernate。

增加：Configuration、SessionFactory、Session等。

# 第二篇 Hibernate高级映射

## 实体映射

### 基本实体映射技术

实体映射就是指pojo类和数据库表之间的映射定义,hibernate中映射关系显然是通过hbm.xml映射文件实现的(当然，可以使用注解来做)。 映射关系包含三重含义：

1. 类和表的映射

通过<class>元素来实现映射关系定义。

Element : class

default: unqualified class name Root entity mapping. Poorly named as entities do not have to be represented by classes at all. Mapped entities may be represented via different methodologies (POJO, Map, Dom4j).

示例：

|  |
| --- |
| <hibernate-mapping  package=*"com.bjsxt.hib.bean"*>  <class name=*"User"* table=*"\_user"* lazy=*"true"*>  … |

通过这些属性说明了实体类com.bjsxt.hib.bean.User和表\_user进行对应。

1. 主键映射

|  |
| --- |
| <id name=*"id"* column=*"id"* type=*"java.lang.String"* >  <generator class=*"uuid"*/>  </id> |

说明了将实体类中id属性映射到表中字段为id，生成策略为uuid

1. 普通字段映射

|  |
| --- |
| <property name=*"name"* type=*"java.lang.String"*  not-null=*"true"*  length=*"15"*  column=*"\_name"*/> |

### 联合主键映射

我们前面一般采用一个字段来作为主键。实际开发中，可能会使用多个字段作为联合主键。那么，这种情况下hibernate应用如何处理?

下面的DDL语句定义了一张采用联合主键的表：

|  |
| --- |
| CREATE TABLE students (  firstname varchar2(16) NOT NULL,  lastname varchar2(16) NOT NULL,  major varchar2(100),  PRIMARY KEY (firstname,lastname)  ); |

联合主键我们可以通过将联合主键相关字段定义到一个pojo中。上面的DDL对应如下pojo：

|  |
| --- |
| **public** **class** StudentId **implements** java.io.Serializable { //composite-id class must implement Serializable  **private** String firstname;  **private** String lastname;  //省略get和set方法，限于篇幅  } |
| **public** **class** Student {  **private** StudentId id;  **private** String major;  //省略get和set方法，限于篇幅  } |

注：本例子只用来说明语法定义。逻辑上并不通！

映射文件Student.hbm.xml，内容如下：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.bjsxt.hib.bean"*>  <class name=*"Student"* table=*"\_student"* lazy=*"true"*>  <composite-id name=*"id"* class=*"StudentId"* >  <key-property name=*"firstname"* type=*"string"* column=*"firstname"*></key-property>  <key-property name=*"lastname"*></key-property>  </composite-id>  <property name=*"major"*></property>  </class>  </hibernate-mapping> |

将student.hbm.xml增加到hibernate.cfg.xml中。

|  |
| --- |
| <mapping resource=*"com/bjsxt/hib/bean/Student.hbm.xml"*/> |

测试代码为：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();  Student s = **new** Student();  StudentId sid = **new** StudentId();  sid.setFirstname("士兵");  sid.setLastname("马");  s.setId(sid);  session.save(s);  tx.commit();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行测试代码后，查看数据库，发现新增表\_student，对应的ddl语句如下：

|  |
| --- |
| -- Table "\_student" DDL  CREATE TABLE `\_student` (  `firstname` varchar(255) NOT NULL,  `lastname` varchar(255) NOT NULL,  `major` varchar(255) default NULL,  PRIMARY KEY (`firstname`,`lastname`)  ) ENGINE=InnoDB DEFAULT CHARSET=gbk; |

同时，数据库中已经存在数据：



### 组合关系映射

组合关系是我们在面向对象设计中经常涉及的内容。那么，在hibernate中如何将组合关系映射到数据中呢?

我们定义两个类，构成组合关系：

|  |
| --- |
| **public** **class** Address {  **private** String city;  **private** String street;  **private** String zipcode;  //限于篇幅，省略get和set方法！  } |
| **package** com.bjsxt.hib.bean;  **public** **class** Person {  **private** **int** id;  **private** String name;  **private** Address address;  //限于篇幅，省略get和set方法！  } |

增加Person.hbm.xml配置文件，并将该文件增加到hibernate.cfg.xml文件中。

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.bjsxt.hib.bean"*>  <class name=*"Person"* table=*"\_person"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*/>  </id>  <property name=*"name"* />  <component name=*"address"* class=*"Address"* >  <property name=*"city"*></property>  <property name=*"street"*></property>  <property name=*"zipcode"*></property>  </component>  </class>  </hibernate-mapping> |

测试代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Address a = **new** Address();  a.setCity("北京");  a.setStreet("长安街");    Person p = **new** Person();  p.setName("高淇");  p.setAddress(a);    session.save(p);  tx.commit();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行上面的代码，数据库中生成一张表\_person，对应ddl语句如下：

|  |
| --- |
| CREATE TABLE `\_person` (  `id` int(11) NOT NULL auto\_increment,  `name` varchar(255) default NULL,  `city` varchar(255) default NULL,  `street` varchar(255) default NULL,  `zipcode` varchar(255) default NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=gbk; |

已经将对象相关信息插入表中：



注：组合关系会生成大量的冗余数据。因此，实际开发中用的不多！

### 大对象映射

Clob:文本大对象，最长4G

Blob:二进制数据大对象，最长4G

我们pojo中如果使用了大对象blob和clob。那么，我们可以按照如下操作进行：

定义一个User2类，该类的属性中使用了blob和clob：

|  |
| --- |
| **package** com.bjsxt.hib.bean;  **import** java.sql.Blob;  **import** java.sql.Clob;  **public** **class** User2 {  **private** **int** id;  **private** String name;  **private** String password;  **private** Blob image;  **private** Clob resume;  //省略了get和set方法  } |

User2.hbm.xml内容如下：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.bjsxt.hib.bean"*>  <class name=*"User2"* table=*"\_user2"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*/>  </id>  <property name=*"name"* />  <property name=*"password"* />  <property name=*"image"* type=*"java.sql.Blob"* />  <property name=*"resume"* type=*"java.sql.Clob"* />  </class>  </hibernate-mapping> |

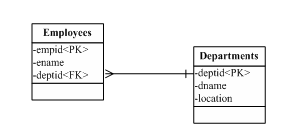
测试代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    InputStream is = **new** FileInputStream(**new** File("d:/1.png"));  Blob b = Hibernate.*createBlob*(is);  Clob c = Hibernate.*createClob*("我是中国人！");  User2 u2 = **new** User2();  u2.setName("老高");  u2.setImage(b);  u2.setResume(c);    session.save(u2);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

## 关联映射

### 一对多/多对一

“一对多/多对一”是数据库表关系中最常见的一种关系。两张表通过外键进行关联，实现表达“一对多/多对一”关系。外键通常位于”多方”表中。如一个部门有很多员工，一个学校有很多学生等。如下图中两张表表示了雇员跟部门的关系。



对应的pojo类：

|  |
| --- |
| **package** com.bjsxt.hib.bean;  **import** java.util.HashSet;  **import** java.util.Set;  **public** **class** Department {  **private** **int** deptid;  **private** String dname;  **private** String location;  **private** Set<Employee> employees = **new** HashSet<Employee>();  //对应的get和set方法限于篇幅，省略！  } |
| **package** com.bjsxt.hib.bean;  **public** **class** Employee {  **private** **int** empid;  **private** String ename;  **private** Department department;  //对应的get和set方法限于篇幅，省略！  } |

定义两个hbm文件Employee.hbm.xml和Department.hbm.xml，表示上面两个pojo类和表的映射关系：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.bjsxt.hib.bean"*>  <class name=*"Employee"* table=*"\_employee"* lazy=*"true"*>  <id name=*"empid"*>  <generator class=*"native"*></generator>  </id>  <property name=*"ename"* type=*"java.lang.String"* column=*"ename"* ></property>  <many-to-one name=*"department"* class=*"Department"* cascade=*"save-update"*>  <column name=*"deptid"*></column> <!-- 外键 -->  </many-to-one>  </class>  </hibernate-mapping> |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Department"* table=*"\_department"* lazy=*"true"*>  <id name=*"deptid"*>  <generator class=*"native"*></generator>  </id>  <property name=*"dname"* type=*"java.lang.String"* column=*"dname"*></property>  <property name=*"location"* type=*"java.lang.String"* column=*"location"*></property>  <set name=*"employees"* cascade=*"all"* inverse=*"true"*>  <key>  <column name=*"deptid"*></column>  </key>  <one-to-many class=*"Employee"* />  </set>  </class>  </hibernate-mapping> |

测试代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Department d = **new** Department();  d.setDname("市场部");  d.setLocation("北京市长安街");    Employee e1 = **new** Employee();  e1.setEname("张三");  Employee e2 = **new** Employee();  e2.setEname("李四");    e1.setDepartment(d);  e2.setDepartment(d);    session.save(e1);  session.save(e2);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

数据库中出现表和相关数据内容：





### 双向关联和单向关联

单向关联：只能通过A查询B，或者B查询A。下面pojo之间的关系即为单向关联：

|  |
| --- |
| **public** **class** Department {  **private** **int** deptid;  **private** String dname;  **private** String location;  //省略get和set方法  } |
| **public** **class** Employee {  **private** **int** empid;  **private** String ename;  **private** Department department;  //省略get和set方法  } |

我们可以通过Employee的department属性查询到department数据。反之，因为Department中没有Employee定义，则不能查询到Employee数据。

双向关联：既可以通过A查询B，也可以通过B查询A。(**一般，我们使用双向关联！**)

|  |
| --- |
| **public** **class** Department {  **private** **int** deptid;  **private** String dname;  **private** String location;  **private** Set<Employee> employees = **new** HashSet<Employee>();  } |
| **public** **class** Employee {  **private** **int** empid;  **private** String ename;  **private** Department department;  } |

**实际开发中，我们一般使用双向关联！**

### cascade属性

* 级联操作。操作一个对象时将该对象相关属性对象也进行同样操作。
  + all：进行任何操作都级联。
  + save-update：保存和更新操作时
  + delete：删除操作时级联
  + all-delete-orphan：当被关联对象失去宿主时，将其级联删除。
  + none(默认) ：不级联

测试save-update属性：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Department d = **new** Department();  d.setDname("市场部");  d.setLocation("北京市长安街");    Employee e1 = **new** Employee();  e1.setEname("张三");  Employee e2 = **new** Employee();  e2.setEname("李四");    e1.setDepartment(d);  e2.setDepartment(d);    session.save(e1);  session.save(e2);    tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

控制台打印语句：

|  |
| --- |
| **Hibernate: insert into \_department (dname, location) values (?, ?)**  09:19:41,140 DEBUG StringType:80 - binding '市场部' to parameter: 1  09:19:41,140 DEBUG StringType:80 - binding '北京市长安街' to parameter: 2  **Hibernate: insert into \_employee (ename, deptid) values (?, ?)**  09:19:41,156 DEBUG StringType:80 - binding '张三' to parameter: 1  09:19:41,156 DEBUG IntegerType:80 - binding '1' to parameter: 2  **Hibernate: insert into \_employee (ename, deptid) values (?, ?)**  09:19:41,156 DEBUG StringType:80 - binding '李四' to parameter: 1  09:19:41,156 DEBUG IntegerType:80 - binding '1' to parameter: 2 |

我们可以看到虽然我们保存的是e1和e2，但是d对象也被存储到数据库中。

测试all属性，将cascade属性值置为all。代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Department d2 = (Department) session.load(Department.**class**, 1);  session.delete(d2);    tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

控制台打印执行的sql语句：

|  |
| --- |
| **Hibernate: update \_employee set deptid=null where deptid=?**  09:31:44,750 DEBUG IntegerType:80 - binding '1' to parameter: 1  **Hibernate: delete from \_employee where empid=?**  09:31:44,750 DEBUG IntegerType:80 - binding '2' to parameter: 1  Hibernate: delete from \_employee where empid=?  09:31:44,750 DEBUG IntegerType:80 - binding '1' to parameter: 1  Hibernate: delete from \_department where deptid=?  09:31:44,750 DEBUG IntegerType:80 - binding '1' to parameter: 1 |

表employee中外键值为deptid=1的记录也被删除！

### inverse属性

* 由哪一方维护外键的值。默认为false, 双方都可以维护关联关系； 为true, 则表示由“多方”维护；
* 建议：
  + 为true,由多方维护提高效率。调用必须调用“多方”(**一般由多方维护**)。
  + 为false，写程序方便，运行效率低下。

在一对多关系中，一方来维护关系，就会发生在插入或者删除一方时，去更新多方中所有关联数据。 而如果是多方维护关系，则直接插入或者删除对象即可。

当inverse=false时，我们做第一个测试：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Department d = **new** Department();  d.setDname("市场部");  d.setLocation("北京市长安街");    Employee e1 = **new** Employee();  e1.setEname("张三");  Employee e2 = **new** Employee();  e2.setEname("李四");    e1.setDepartment(d);  e2.setDepartment(d);  session.save(e1);  session.save(e2);    tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

上面这个测试，保存的是e1和e2，我们发现执行了如下sql语句：

|  |
| --- |
| Hibernate: insert into \_department (dname, location) values (?, ?)  09:40:24,421 DEBUG StringType:80 - binding '市场部' to parameter: 1  09:40:24,437 DEBUG StringType:80 - binding '北京市长安街' to parameter: 2  Hibernate: insert into \_employee (ename, deptid) values (?, ?)  09:40:24,437 DEBUG StringType:80 - binding '张三' to parameter: 1  09:40:24,437 DEBUG IntegerType:80 - binding '1' to parameter: 2  Hibernate: insert into \_employee (ename, deptid) values (?, ?)  09:40:24,453 DEBUG StringType:80 - binding '李四' to parameter: 1  09:40:24,453 DEBUG IntegerType:80 - binding '1' to parameter: 2 |

显然，d也被存入数据库中。并且\_employee表中两个新纪录的外键值全部关联到了d中的新纪录。

上面这个实验，说明双方的关联关系是可以由多方来维护和操作。

当inverse=false时，我们做第二个测试：

|  |
| --- |
| 在Department类中增加方法：  **public** **void** addEmployee(Employee e){  **this**.employees.add(e);  } |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Department d = **new** Department();  d.setDname("市场部");  d.setLocation("北京市长安街");    Employee e1 = **new** Employee();  e1.setEname("张三");  Employee e2 = **new** Employee();  e2.setEname("李四");    d.addEmployee(e1);  d.addEmployee(e2);  session.save(d);    tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行上面测试代码，控制台打印：

|  |
| --- |
| Hibernate: insert into \_department (dname, location) values (?, ?)  09:46:23,625 DEBUG StringType:80 - binding '市场部' to parameter: 1  09:46:23,625 DEBUG StringType:80 - binding '北京市长安街' to parameter: 2  Hibernate: insert into \_employee (ename, deptid) values (?, ?)  09:46:23,640 DEBUG StringType:80 - binding '张三' to parameter: 1  09:46:23,640 DEBUG IntegerType:73 - binding null to parameter: 2  Hibernate: insert into \_employee (ename, deptid) values (?, ?)  09:46:23,640 DEBUG StringType:80 - binding '李四' to parameter: 1  09:46:23,640 DEBUG IntegerType:73 - binding null to parameter: 2  Hibernate: update \_employee set deptid=? where empid=?  09:46:23,671 DEBUG IntegerType:80 - binding '1' to parameter: 1  09:46:23,671 DEBUG IntegerType:80 - binding '1' to parameter: 2  Hibernate: update \_employee set deptid=? where empid=?  09:46:23,671 DEBUG IntegerType:80 - binding '1' to parameter: 1  09:46:23,671 DEBUG IntegerType:80 - binding '2' to parameter: 2 |

我们保存了d对象，但是跟他关联的e1和e2都被保存，同时他们之间的关系通过deptid也被保存到数据库中(两个update语句)。

通过这个实验，说明关联关系可以由一方维护。

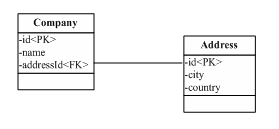
综上，我们可以总结如下：

**Inverse=false，关联关系可以由双方维护！**

**但是我们一般将关联关系交给”多方”来维护。上面两次实验，第一次交给了多方维护，执行了3条SQL语句。第二交给了一方维护，执行了3条insert语句和2条update语句(假如增加200个雇员那么就会有200个update语句)。如果我们交给一方来维护，显然会有多余的SQL语句执行，从而降低了效率。因此，我们一般建议关联关系由多方维护！那么这时候，我们可以强制通过inverse=true，指定由多方维护！**

### 一对一

* 一对一关联指两个表之间的记录是一一对应的关系。分为两种：外键关联和主键关联。
* 比如一家公司(Company)和它所在的地址(Address)。在业务逻辑中要求一家公司只有唯一的地址，一个地址也只有一家公司。 下图表现为外键关联关系。



对应的pojo定义如下：

|  |
| --- |
| **package** com.bjsxt.hib.bean;  **public** **class** Company {  **private** **int** id;  **private** String name;  **private** Address2 address2;  //省略get和set方法  } |
| **package** com.bjsxt.hib.bean;  **public** **class** Address2{  **private** **int** id;  **private** String city;  **private** String country;  **private** Company company;  //省略get和set方法  } |

一对一关系我们可以采用为一个外键关联或主键关联两种方式！

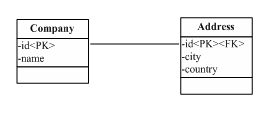
#### 唯一外键关联

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Address2"* table=*"\_address2"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"city"*/>  <property name=*"country"*/>  <one-to-one name=*"company"* class=*"Company"* property-ref=*"address2"*  cascade=*"all"*></one-to-one>  </class>  </hibernate-mapping> |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Company"* table=*"\_company"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"*/>  <many-to-one name=*"address2"* class=*"Address2"* cascade=*"all"*  unique=*"true"*>  <column name=*"addressId"* />  </many-to-one>  </class>  </hibernate-mapping> |

**注：实际开发中，一对一关系我们一般采用唯一外键关联！**

#### 主键关联

* 一对一的另一种解决方式就是主键关联，在这种关联关系中，要求两个对象的主键必须保持一致，通过两个表的主键建立关联关系，无须外键参与。



|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Address2"* table=*"\_address2"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"city"*/>  <property name=*"country"*/>  <one-to-one name=*"company"* class=*"Company"* cascade=*"all"* ></one-to-one>  </class>  </hibernate-mapping> |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Company"* table=*"\_company"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"*/>  <one-to-one name=*"address2"* class=*"Address2"* cascade=*"all"* ></one-to-one>  </class>  </hibernate-mapping> |

测试一对一关系代码(映射文件配置采用唯一外键关联或者主键关联)：

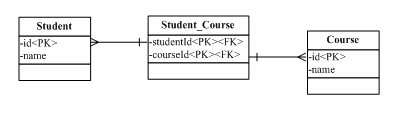
|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Company c = **new** Company();  c.setName("北京尚学堂");  Address2 a = **new** Address2();  a.setCity("北京");    c.setAddress2(a);    session.save(c);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行上面代码，控制台打印如下SQL语句：

|  |
| --- |
| Hibernate: insert into \_address2 (city, country) values (?, ?)  10:19:55,500 DEBUG StringType:80 - binding '北京' to parameter: 1  10:19:55,515 DEBUG StringType:73 - binding null to parameter: 2  Hibernate: insert into \_company (name, addressId) values (?, ?)  10:19:55,515 DEBUG StringType:80 - binding '北京尚学堂' to parameter: 1  10:19:55,515 DEBUG IntegerType:80 - binding '1' to parameter: 2 |

### 多对多

* 多对多的实体关系模型也是很常见的，比如学生和课程的关系。一个学生可以选修多门课程，一个课程可以被多名学生选修。在关系型数据库中对于多对多关联关系的处理一般采用中间表的形式，将多对多的关系转化成两个一对多的关系。



下面两个pojo代表学生和课程：

|  |
| --- |
| **package** com.bjsxt.hib.bean;  **import** java.util.HashSet;  **import** java.util.Set;  **public** **class** Student2 {  **private** **int** id;  **private** String name;  **private** Set courses = **new** HashSet();  **public** **void** addCourse(Course c) {  **this**.courses.add(c);  }    //省略get和set方法！  } |
| **public** **class** Course {  **private** Long id;  **private** String name;  **private** Set students = **new** HashSet();  **public** **void** addStudent(Student2 s) {  **this**.students.add(s);  }  //省略了get和set方法  } |

Course.hbm.xml内容如下：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Course"* table=*"\_course"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* />  <set name=*"students"* table=*"stu\_course"* cascade=*"save-update"*>  <key column=*"courseId"* />  <many-to-many class=*"Student2"* column=*"stuId"* />  </set>  </class>  </hibernate-mapping> |

Student2.hbm.xml内容如下：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package=*"com.bjsxt.hib.bean"*>  <class name=*"Student2"* table=*"\_stu"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* />  <set name=*"courses"* table=*"stu\_course"* cascade=*"save-update"*>  <key column=*"stuId"* />  <many-to-many class=*"Course"* column=*"courseId"* />  </set>  </class>  </hibernate-mapping> |

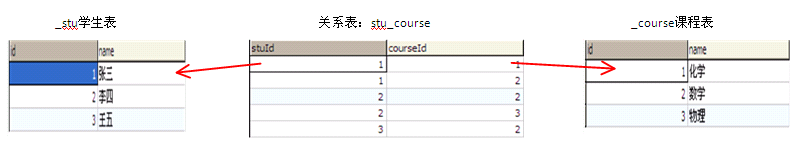
测试代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Course c = **new** Course();  c.setName("物理");  Course c2 = **new** Course();  c2.setName("化学");  Course c3 = **new** Course();  c3.setName("数学");    Student2 s1 = **new** Student2();  s1.setName("张三");  Student2 s2 = **new** Student2();  s2.setName("李四");  Student2 s3 = **new** Student2();  s3.setName("王五");    s1.addCourse(c3);  s1.addCourse(c2);    s2.addCourse(c);  s2.addCourse(c3);    s3.addCourse(c3);    session.save(s1);  session.save(s2);  session.save(s3);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行代码后，控制台打印如下SQL语句：

|  |
| --- |
| Hibernate: insert into \_stu (name) values (?)  10:50:41,828 DEBUG StringType:80 - binding '张三' to parameter: 1  Hibernate: insert into \_course (name) values (?)  10:50:41,843 DEBUG StringType:80 - binding '化学' to parameter: 1  Hibernate: insert into \_course (name) values (?)  10:50:41,843 DEBUG StringType:80 - binding '数学' to parameter: 1  Hibernate: insert into \_stu (name) values (?)  10:50:41,843 DEBUG StringType:80 - binding '李四' to parameter: 1  Hibernate: insert into \_course (name) values (?)  10:50:41,843 DEBUG StringType:80 - binding '物理' to parameter: 1  Hibernate: insert into \_stu (name) values (?)  10:50:41,843 DEBUG StringType:80 - binding '王五' to parameter: 1  Hibernate: insert into stu\_course (stuId, courseId) values (?, ?)  10:50:41,875 DEBUG IntegerType:80 - binding '1' to parameter: 1  10:50:41,875 DEBUG LongType:80 - binding '1' to parameter: 2  Hibernate: insert into stu\_course (stuId, courseId) values (?, ?)  10:50:41,875 DEBUG IntegerType:80 - binding '1' to parameter: 1  10:50:41,875 DEBUG LongType:80 - binding '2' to parameter: 2  Hibernate: insert into stu\_course (stuId, courseId) values (?, ?)  10:50:41,875 DEBUG IntegerType:80 - binding '2' to parameter: 1  10:50:41,875 DEBUG LongType:80 - binding '3' to parameter: 2  Hibernate: insert into stu\_course (stuId, courseId) values (?, ?)  10:50:41,875 DEBUG IntegerType:80 - binding '2' to parameter: 1  10:50:41,875 DEBUG LongType:80 - binding '2' to parameter: 2  Hibernate: insert into stu\_course (stuId, courseId) values (?, ?)  10:50:41,875 DEBUG IntegerType:80 - binding '3' to parameter: 1  10:50:41,875 DEBUG LongType:80 - binding '2' to parameter: 2 |

数据库中的数据如下所示：



## 集合映射

**Set(一般使用Set)**

**<set name=*"employees" cascade="all" inverse="false" >***

<key>

<column name=*"deptid"></column>*

</key>

**<one-to-many class=*"Employee" />***

**</set>**

**List**

<list name=*"emps" cascade="all" inverse="false" >*

<key>

<column name=*"deptid"></column>*

</key>

<list-index column=*"shunxu" ></list-index>*

<one-to-many class=*"Employee" />*

</list>

Map

<map name="empsMap" cascade="all" >

<key column="deptid"></key>

<map-key type="string" column="ename"></map-key>

<one-to-many class="Employee" />

</map>

# 继承映射

* 单表继承(整个类继承结构一个表)
  + 优点：只有一张表，操作方便，效率高。
  + 缺点：冗余数据多，增加子类要修改表结构，数据多时表非常大。
* joined-subclass（父类映射成表，一个类一张表，互相关联，不独立）
  + 优点：冗余字段少，表结构合理(关系模型设计上优良)
  + 缺点：多态查询慢(一般不要用，最好直接指定类型。比如查询man，则会把man的子类表也查一遍)、查询需要外连接
* union-subclass （父类不映射成表，每个子类一张表，互不关联，独立）
  + 优点：表都独立利于移植，查询不用外连接速度快
  + 缺点：数据库包含重复字段过多，包含了父类中的字段
  + 注意：父类和子类id不能重复，用自动递增不行，可用increment/hilo/uuid等。
* 如何选用?（一般使用joined-subclass）
  + 如果子类属性不多，总数据量不大，选用单表继承
  + **如果子类属性较多，可用joined-subclass.**
  + union-subclass用的很少

我们定义三个javabean，他们存在继承关系。代码如下：

|  |
| --- |
| **package** com.sxt.pojo;  **public** **class** Person {  **private** **int** id;  **private** String name;  //限于篇幅，省略get和set方法！  } |
| **package** com.sxt.pojo;  **public** **class** Student **extends** Person {  **private** String school;  **private** **int** studentNum; //学号  //限于篇幅，省略get和set方法！  }  **package** com.sxt.pojo;  **public** **class** Teacher **extends** Person {  **private** String salary;  **private** String teacherNum; //教师编号  //限于篇幅，省略get和set方法！  } |
|  |

## 单表继承(整个类继承结构一个表)

* 优点：只有一张表，操作方便，效率高。
* 缺点：冗余数据多，增加子类要修改表结构，数据多时表非常大。

开发步骤如下：

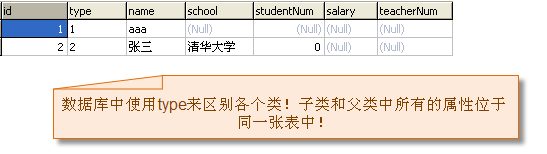
映射文件内容如下：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.sxt.pojo"*>  <class name=*"Person"* table=*"t\_person"* lazy=*"true"* **discriminator-value=*"1"*** >  <id name=*"id"*>  <generator class=*"native"*/>  </id>  **<discriminator column=*"type"*></discriminator>**  <property name=*"name"*/>  <subclass name=*"Student"* **discriminator-value=*"2"***>  <property name=*"school"*/>  <property name=*"studentNum"*/>  </subclass>  <subclass name=*"Teacher"* **discriminator-value=*"3"***>  <property name=*"salary"*/>  <property name=*"teacherNum"*/>  </subclass>  </class>  </hibernate-mapping> |

测试代码如下：

|  |
| --- |
| package com.sxt.test;  import org.hibernate.Session;  import org.hibernate.Transaction;  import com.sxt.pojo.Person;  import com.sxt.pojo.Student;  public class Test1 {  public static void savePerson(Person p){  Session session = null;  Transaction tx = null;  try {  session = HibUtil.getSession();  tx = session.beginTransaction();  session.save(p);  tx.commit();  } catch (Exception e) {  e.printStackTrace();  }finally{  if(session!=null){  session.close();  }  }  }    public static void main(String[] args) {  Person p = new Person();  p.setName("aaa");  savePerson(p);  Student stu = new Student();  stu.setName("张三");  stu.setSchool("清华大学");  savePerson(stu);  }  } |

我们查看数据库中的数据内容，结果如下：



## joined-subclass

父类映射成表，每个类一张表。每张表仅包含当前类的属性，不包含父类属性。各表之间通过关联关系连接，互相不独立。这种方式最符合关系模型，设计较优良，冗余数据较少。

* 优点：冗余字段少，表结构合理(关系模型设计上优良)
* 缺点：多态查询慢(一般不要用，最好直接指定类型。比如查询man，则会把man的子类表也查一遍)、查询需要外连接

开发步骤：

1. 映射文件内容如下：

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.sxt.pojo"*>  <class name=*"Person"* table=*"t\_person"* lazy=*"true"*>  <id name=*"id"*>  <generator class=*"native"*/>  </id>  <property name=*"name"*/>    **<joined-subclass name=*"Student"* table=*"t\_student"*>**  **<key column=*"personid"* foreign-key=*"id"*></key>**  **<property name=*"school"*/>**  **<property name=*"studentNum"*/>**  **</joined-subclass>**  **<joined-subclass name=*"Teacher"*>**  **<key column=*"personid"* foreign-key=*"id"*></key>**  **<property name=*"salary"*/>**  **<property name=*"teacherNum"*/>**  **</joined-subclass>**  </class>  </hibernate-mapping> |

1. 测试代码如下：

|  |
| --- |
| package com.sxt.test;  import org.hibernate.Session;  import org.hibernate.Transaction;  import com.sxt.pojo.Person;  import com.sxt.pojo.Student;  import com.sxt.pojo.Teacher;  public class Test1 {  public static void savePerson(Person p){  Session session = null;  Transaction tx = null;  try {  session = HibUtil.getSession();  tx = session.beginTransaction();  session.save(p);  tx.commit();  } catch (Exception e) {  e.printStackTrace();  }finally{  if(session!=null){  session.close();  }  }  }    public static void main(String[] args) {  Teacher t = new Teacher();  t.setName("高淇");  t.setSalary(3000+"");  Student stu = new Student();  stu.setName("张三");  stu.setSchool("清华大学");  savePerson(stu);  savePerson(t);  }  } |

1. 结果为：



t\_person表的内容如下：



t\_student表的内容如下：



teacher表的内容如下：



## union-subclass

* 优点：表都独立利于移植，查询不用外连接速度快
* 缺点：数据库包含重复字段过多，包含了父类中的字段

注意：父类和子类id不能重复，用native不行，可用increment/hilo/uuid等。

开发步骤：

1. 父类需要定义成抽象类！

|  |
| --- |
| **package** com.sxt.pojo;  **public *abstract*** **class** Person {  **private** **int** id;  **private** String name;  //限于篇幅，省略get和set方法！  } |

1. 映射文件名称仍然使用父类！
2. 子类信息定义在映射文件中，使用<union-subclass>

|  |
| --- |
| <?xml version=*"1.0"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping  package=*"com.sxt.pojo"*>  <class name=*"Person"* table=*"t\_person"* lazy=*"true"* **abstract=*"true"*** >  <id name=*"id"*>  <generator class=*"****increment****"*/>  </id>  <property name=*"name"*/>    **<union-subclass name=*"Student"* table=*"t\_student"*>**  **<property name=*"school"*/>**  **<property name=*"studentNum"*/>**  **</union-subclass>**  **<union-subclass name=*"Teacher"*>**  **<property name=*"salary"*/>**  **<property name=*"teacherNum"*/>**  **</union-subclass>**  </class>  </hibernate-mapping> |

测试代码如下：

|  |
| --- |
| package com.sxt.test;  import org.hibernate.Session;  import org.hibernate.Transaction;  import com.sxt.pojo.Person;  import com.sxt.pojo.Student;  import com.sxt.pojo.Teacher;  public class Test1 {  public static void savePerson(Person p){  Session session = null;  Transaction tx = null;  try {  session = HibUtil.getSession();  tx = session.beginTransaction();  session.save(p);  tx.commit();  } catch (Exception e) {  e.printStackTrace();  }finally{  if(session!=null){  session.close();  }  }  }    public static void main(String[] args) {  Teacher t = new Teacher();  t.setName("高淇");    Student stu = new Student();  stu.setName("张三");  stu.setSchool("清华大学");  savePerson(stu);  savePerson(t);  }  } |

结果如下：



student表中，记录已经插入：



teacher表中，记录如下：



我们发现一个问题，teacher表中的记录的id并不是1，而是2. 这里就是说明，两个子类表中的id的值都不能重复！

**作业：**

1. **查资料，使用注解完成继承映射！！**

# 第三篇 Annotation 注解的应用

## Hibernate Annotation介绍

Hibernate3.2之前，依靠外部的XML文件来配置持久化对象，说明类和表之间、对象和记录之间、属性和字段之间的关系。不管是手动写映射文件还是自动生成，都会产生大量的XML映射文件。这为我们的开发带来了不便。

从hibernate3.2开始，出现了Hibernate Annotation库，它建立在java5的注解之上。由于hibernate3.2之后已经完全兼容JPA规范，及支持最新的EJB3持久化规范。

注解的使用，使我们不再向以前一样频繁的写XML配置文件，一定程度上提高了开发效率。

我们可以使用EJB3的持久化规范中的注解方式，实现我们前面学过的关于映射关系的所有配置。

注：ejb3注解的API定义在：javax.persistence.\*包中。

## 使用Annotation开发项目

下面我们新建一个项目，在这个项目中我们使用Annotation，完成第一篇和第二篇中所有的hibernate操作，大家会发现通过注解大大简化了我们开发流程，使我们从繁琐的XML配置中解放出来。

1. 新建一个java项目。并将hibernate需要的相关jar包导入。
2. 使用annotation需要额外增加3个jar包：
   * hibernate-annotations.jar
   * ejb3-persistence.jar
   * hibernate-commons-annotations.jar
3. 新建一个pojo并增加注解：

|  |
| --- |
| **package** com.bjsxt.hib.bean;  **import** java.util.Date;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  @Entity  **public** **class** User {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;  **private** String name;  **private** Date birthday;  //get和set方法省略  } |

1. 在hibernate.cfg.xml中增加：

|  |
| --- |
| <mapping class=*"com.bjsxt.hib.bean.User"* /> |

1. HibUtil代码如下：

|  |
| --- |
| package com.bjsxt.hib.test;  import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.cfg.AnnotationConfiguration;  import org.hibernate.cfg.Configuration;  public class HibUtil {  private static SessionFactory factory;    private HibUtil(){  }  static {  **Configuration configuration = new** **AnnotationConfiguration();**  configuration.configure();  factory = configuration.buildSessionFactory();  }  public static SessionFactory getFactory() {  return factory;  }  public static Session getSession(){  return factory.openSession();  }  } |

1. Test2.java代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    User u = **new** User();  u.setName("高淇");  session.save(u);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行上面的测试代码，控制台打印：

|  |
| --- |
| Hibernate: insert into User (name, birthday) values (?, ?)  13:37:53,296 DEBUG StringType:80 - binding '高淇' to parameter: 1  13:37:53,296 DEBUG TimestampType:73 - binding null to parameter: 2 |

## 常见的注解及其典型用法

|  |  |  |
| --- | --- | --- |
| **注解名称** | **作用** | **典型代码/其他** |
| @Entity | 将一个类声明为一个实体bean(即一个持久化POJO类) | @Entity  **public** **class** User {  //…类体省略  } |
| @Table | 注解声明了该实体bean映射指定的表（table）,目录（catalog）和schema的名字 | @Entity  @Table(name="\_user")  **public** **class** User {  //…省略类体  } |
| @Id | 注解声明了该实体bean的标识属性（对应表中的主键） | @Entity  **public** **class** User {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;  **private** String name;  **private** Date birthday;  //省略了get和set方法  } |
| @GeneratedValue | 声明了主键的生成策略。该注解有如下属性 |
| *@Column* | 声明了属性到列的映射 | @Column(name="\_uname")  **private** String name; |
| @Transient | 声明的属性不会跟数据库关联 | @Transient  **private** Date birthday;  则数据库中不会有birthday列 |
| @Formula | 用一个查询语句动态的生成一个类的属性. 表示这个属性是一个虚拟的列，表中并没有这个列。需要通过查询语句计算出来。 | @Formula("(select count(\*) from \_user u where u.id>id)")  **private** **int** countUser;  要点：   1. Sql语句必须位于括号中 2. 这里最好写完全的sql语句。表名最好使用别名 3. 如果要引用当前对象的属性值，可以直接使用属性，如：   u.id>id  第**二个id就是对象**的值！ |

## 一对一

两个pojo类：

|  |
| --- |
| @Entity  **public** **class** Company {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;  **private** String name;  @OneToOne(cascade=CascadeType.*ALL*)  **private** Address2 address2;  //省略了get和set方法  } |
| @Entity  **public** **class** Address2{  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;  **private** String city;  **private** String country;    @OneToOne(cascade=CascadeType.*ALL*,mappedBy="address2")  **private** Company company;  //省略了get和set方法  } |

注：一般都要用mappedBy属性，由某一端来维护关联关系。若不加的话，两张表都会出现关联关系维护字段，如下图所示：

Address2表：



Company表：



测试代码如下：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Address2 a = **new** Address2();  a.setCity("北京");    Company c = **new** Company();  c.setName("尚学堂");    c.setAddress2(a);    session.save(c);  session.save(a);    tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |
| 控制台打印：  Hibernate: insert into Address2 (city, country, company\_id) values (?, ?, ?)  14:50:10,109 DEBUG StringType:80 - binding '北京' to parameter: 1  14:50:10,109 DEBUG StringType:73 - binding null to parameter: 2  14:50:10,109 DEBUG IntegerType:73 - binding null to parameter: 3  Hibernate: insert into Company (name, address2\_id) values (?, ?)  14:50:10,125 DEBUG StringType:80 - binding '尚学堂' to parameter: 1  14:50:10,125 DEBUG IntegerType:80 - binding '1' to parameter: 2 |

## 一对多/多对一

两个pojo代码如下：

|  |
| --- |
| @Entity  **public** **class** Employee {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** empid;  **private** String ename;  @ManyToOne  **private** Department department;  //get和set方法省略  } |
| @Entity  **public** **class** Department {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** deptid;  **private** String dname;  **private** String location;  @OneToMany(mappedBy="department")  **private** Set<Employee> employees = **new** HashSet<Employee>();  //get和set方法省略  } |

测试代码：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Department d = **new** Department();  d.setDname("市场部");  d.setLocation("北京市长安街");    Employee e1 = **new** Employee();  e1.setEname("张三");  Employee e2 = **new** Employee();  e2.setEname("李四");    e1.setDepartment(d);  e2.setDepartment(d);    session.save(e1);  session.save(e2);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行结果：

|  |
| --- |
| Hibernate: insert into Department (dname, location) values (?, ?)  15:13:05,515 DEBUG StringType:80 - binding '市场部' to parameter: 1  15:13:05,531 DEBUG StringType:80 - binding '北京市长安街' to parameter: 2  Hibernate: insert into Employee (ename, department\_deptid) values (?, ?)  15:13:05,531 DEBUG StringType:80 - binding '张三' to parameter: 1  15:13:05,531 DEBUG IntegerType:80 - binding '1' to parameter: 2  Hibernate: insert into Employee (ename, department\_deptid) values (?, ?)  15:13:05,531 DEBUG StringType:80 - binding '李四' to parameter: 1  15:13:05,531 DEBUG IntegerType:80 - binding '1' to parameter: 2 |

## 多对多

两个pojo类：

|  |
| --- |
| @Entity  **public** **class** Student2 {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **int** id;  **private** String name;  @ManyToMany(cascade=CascadeType.*ALL*)  **private** Set<Course> courses = **new** HashSet<Course>();    //省略getter和setter方法  } |
| @Entity  **public** **class** Course {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** Long id;  **private** String name;  @ManyToMany(mappedBy="courses")  **private** Set<Student2> students = **new** HashSet<Student2>();  //省略getter和setter方法  } |

测试代码：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  tx = session.beginTransaction();    Course c = **new** Course();  c.setName("物理");  Course c2 = **new** Course();  c2.setName("化学");  Course c3 = **new** Course();  c3.setName("数学");    Student2 s1 = **new** Student2();  s1.setName("张三");  Student2 s2 = **new** Student2();  s2.setName("李四");  Student2 s3 = **new** Student2();  s3.setName("王五");    s1.addCourse(c3);  s1.addCourse(c2);    s2.addCourse(c);  s2.addCourse(c3);    s3.addCourse(c3);    session.save(s1);  session.save(s2);  session.save(s3);  tx.commit();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  } |

执行代码后，发现生成三张表：

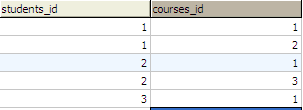
Course表：



Student2表：



中间关系表student2\_course：



## 继承映射

课下学习！

# 第四篇 Hibernate查询

为了测试HQL查询，同时也为了测试Hibernate处理oracle数据库的能力，我们连接oracle 10g进行查询。大家可以参考附录：使用myeclipse自动开发hibernate程序(oracle 为例)。将oracle 10g 中已有的几张表用来测试我们的HQL学习。

## HQL查询(Hibernate Query Language)

Hibernate 配备了一种非常强大的查询语言，这种语言看上去很像 SQL。但是不要被语法结构上的相似所迷惑，HQL 是非常有意识的被设计为完全面向对象的查询，它可以理解如继承、多态和关联之类的概念。

### 基本规则

1. HQL语法类似于SQL，也是一种select from结构的语句。但是他后面跟的不是表名和字段名，而是类名和属性名。
2. HQL基本查询语法跟SQL很类似
3. HQL大小写不敏感。但是，设计java类名、包名、属性名时大小写敏感。
4. 包名的使用情况。比如：如果注册的实体类Emp只有一个类，那么查询时可以不加包名，hibernate会自动检索到Emp类。但是如果注册多个实体类，名字都叫Emp。此时就要增加包名来区别多个实体类。

### 第一个HQL查询

|  |
| --- |
| **package** com.bjsxt.hib.test.hql;  **import** java.util.List;  **import** org.hibernate.Query;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** com.bjsxt.hib.bean.Emp;  **public** **class** Test2 {  **public** **static** **void** main(String[] args) {  Session session = **null**;  Transaction tx = **null**;  **try** {  session = HibUtil.*getSession*();  **String hql = "from Emp";**  **Query q = session.createQuery(hql);**    **List list = q.list();**  **for(int i=0;i<list.size();i++){**  **Emp c = (Emp) list.get(i);**  **System.*out*.println(c.getEname());**  **}**    } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **if**(session!=**null**){  session.close();  }  }  }  } |

### 根据返回类型划分HQL查询

#### 单个对象

|  |
| --- |
| String hql = "select count(\*) from Emp";  Query q = session.createQuery(hql);  Number n = (Number) q.uniqueResult();  //count(\*)可能为Short、Integer、Long等各种类型，但是这些类型都是Number的子类。  System.*out*.println(n.intValue()); |

#### List集合

参考第一个HQL查询中的内容！

#### Object[]数组

有时候，我们不需要讲实体类的所有属性查找出来，只需要查找部分属性。这时，可以将返回的内容放入Object[]中，而不是实体对象中。

|  |
| --- |
| String hql = "select e.ename,e.dept.deptno from Emp e ";  Query q = session.createQuery(hql);    List<Object[]> list = q.list();  **for**(**int** i=0;i<list.size();i++){  Object[] os = list.get(i);  System.*out*.println("雇员名字："+os[0]+"-雇员部门编号："+os[1]);  } |
|  |

#### Map集合

也可以将查询出的记录放入map中，map中将包含列名和值。

|  |
| --- |
| String hql = "select new map(e.ename as ename,e.dept.deptno as deptno) from Emp e "; //一般建议起别名处理查出的字段  Query q = session.createQuery(hql);  List<Map> list = q.list();  **for**(**int** i=0;i<list.size();i++){  Map map = list.get(i);  System.*out*.println("雇员名字："+map.get("ename")+"-雇员部门编号："+map.get("deptno"));  } |

#### 实体对象

对于只查询部分属性的情况，Object数组、Map都很方便。实际上，我们也可以通过构造方法，将查出的数据直接封装到实体对象中。

|  |
| --- |
| 新增构造方法：  **public** Emp(Short empno, String ename) {  **super**();  **this**.empno = empno;  **this**.ename = ename;  } |
| String hql = "select new Emp(e.empno,e.ename) from Emp e ";  Query q = session.createQuery(hql);    List<Emp> list = q.list();  **for**(**int** i=0;i<list.size();i++){  Emp e = list.get(i);  System.*out*.println("雇员编号："+e.getEmpno()+"-雇员名字"+e.getEname());  } |

### Where子句和参数传递

|  |
| --- |
| // String hql = "from Emp where ename=?";  String hql = "from Emp where ename=:ename"; //使用参数名称动态绑定！(推荐使用！)  Query q = session.createQuery(hql);  // q.setString(0, "SMITH"); //参数索引从0开始计数，而不像jdbc一样从1开始。  q.setString("ename", "SMITH");  List list = q.list();  **for**(**int** i=0;i<list.size();i++){  Emp c = (Emp) list.get(i);  System.*out*.println(c.getEname());  } |

### 运算符和统计函数

我们在SQL中学习的运算符和统计函数在HQL中**仍然适用**！

### HQL分页查询

分页显示是项目中必不可少的功能，不同的数据库有不同的分页方式，hibernate替我们屏蔽了数据库中之间的差异。我们通过如下简单的代码即可实现分页功能(如果分页的原理和做法忘记了，可以参考之前讲授的项目内容)。

|  |
| --- |
| String hql = "from Emp";  Query q = session.createQuery(hql);  **q.setFirstResult(0); //从第几条开始取数据**  **q.setMaxResults(10); //设置每页最多显示记录的个数**    List list = q.list();  **for**(**int** i=0;i<list.size();i++){  Emp c = (Emp) list.get(i);  System.*out*.println(c.getEname());  } |

### 跨表查询和对象导航

SQL中复杂的表连接查询，跨表操作。在HQL中进行了相当的简化，我们只需要简单的使用属性即可，类似于我们前面学过的EL表达式。这样，我们可以用简单的代码写出比较复杂的查询。

|  |
| --- |
| String hql = "from Emp e where e.dept.deptno=? ";  Query q = session.createQuery(hql);  q.setInteger(0, 10); |
| 上面的HQL语句生成如下的SQL语句：  Hibernate: select emp0\_.EMPNO as EMPNO0\_, emp0\_.COMM as COMM0\_, emp0\_.DEPTNO as DEPTNO0\_, emp0\_.ENAME as ENAME0\_, emp0\_.HIREDATE as HIREDATE0\_, emp0\_.JOB as JOB0\_, emp0\_.MGR as MGR0\_, emp0\_.SAL as SAL0\_ from SCOTT.EMP emp0\_ where emp0\_.DEPTNO=?  23:24:38,812 DEBUG IntegerType:133 - binding '10' to parameter: 1 |

### Join(内连接、外连接)

SQL中我们有内连接、右外连接、左外连接、全外连接，在HQL中我们也有这些概念。不过，有如下几点不一致：

1. 如果两个实体类之间没有任何关系，那么不能使用join
2. 由于只有两个实体类之间有关联关系才能使用join，因此不需要像SQL那样通过on指明连接条件。

代码示例如下：

|  |
| --- |
| String hql = "select e.ename,d.dname from Emp e left join e.dept d ";  Query q = session.createQuery(hql); |

## SQL原生查询(Native SQL)

有时候HQL可能不能满足我们的要求。我们需要使用原始的SQL来完成我们的功能。我们可以通过如下方式，在hibernate中使用SQL查询：

|  |
| --- |
| String sql = "select ename,sal from emp where empno=:id";  SQLQuery q = session.createSQLQuery(sql);  q.setInteger("id", 7369);  List list = q.list(); //返回的结果为List<Object[]>  **for**(**int** i=0;i<list.size();i++){  Object[] c = (Object[]) list.get(i);  System.*out*.println(c[0]+"-"+c[1]);  } |
| String sql = "select **\*** from emp where empno=:id";  SQLQuery q = session.createSQLQuery(sql);  q.setInteger("id", 7369);  **q.addEntity(Emp.class);**  List<Emp> list = q.list();  **for**(**int** i=0;i<list.size();i++){  Emp c = list.get(i);  System.*out*.println(c.getEname()+"-"+c.getSal());  } |

## 条件查询(Criteria Queries)

查看hibernate开发文档中的内容，自学并整理成word文档！

# 第五篇 hibernate缓存机制和事务隔离机制

## 一级缓存（ Session缓存）

* *一级缓存*的管理
  + 应用程序调用Session的save()、update()、saveOrUpdate()、get()或load()，以及调用查询接口的 list()、iterate() 时，如果在Session缓存中还不存在相应的对象，Hibernate就会把该对象加入到第一级缓存中。
  + 可以通过close/clear/evict清空缓存
* 作用

因为Session的生命期往往很短，存在于Session内部的第一级最快缓存的生命期当然也很短，所以第一级缓存的命中率是很低的。其对系统性能的改善也是很有限的。Session内部缓存的主要作用是保持Session内部数据状态同步。

## 二级缓存(SessionFactory缓存)

* 开启：
  + <property name=*"hibernate.cache.use\_second\_level\_cache">true</property>*
  + <property name=*"hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider</property>*
* 如何使用：
  + 类定义前面：@cache，指该类的对象都会放入二级缓存。  
    @Cache(usage=CacheConcurrencyStrategy.*READ\_WRITE) //放入二级缓存中也可以被修改。一般用它。*
* *什么内容时候放入二级缓存：* 
  + *经常被访问、改动不频繁、数量有限。*
* get/load会使用二级缓存。
* iterate也会使用二级缓存。
* list默认会往二级缓存中存放数据，即通过list查出的结果会放入二级缓存。但是list本身查询时不会使用二级缓存。

## 各种二级缓存插件



## 查询缓存

* **查询缓存只对query.list()起作用**
* **查询缓存依赖于二级缓存，因此一定要打开二级缓存。**
* **查询缓存实现机制：以查询语句为key，查到的对象的id为value**
* **查询缓存的配置和使用：** 
  + 开启二级缓存
  + <property name="hibernate.cache.use\_query\_cache">true</property> //默认是fasle
  + 在程序中必须手动启用查询缓存，如：query.setCacheable(true);

## 缓存算法问题

* 缓存满了后，将内存中哪个对象清掉。
  + LRU
    - Least Recently Used 最近最少被使用的。每个缓存对象都记录一个最后使用时间。
  + LFU
    - Least Frequently Used 最近使用频率最少。
  + FIFO
    - First In First Out

## 1+N问题

* 在一对多/多对一中，经常出现1+N问题。
  + 在1方，查找得到了n个对象， 那么又需要将n个对象关联的集合取出，于是本来的一条sql查询变成了n+1条。
* 解决方案：
  + **懒加载(延时加载、延迟加载)**
    - @OneToMany(mappedBy=“banji”,fetch=FetchType.*LAZY) (默认即为此)*
  + *二级缓存* 
    - 在对象更新，删除，添加相对于查询要少得多时， 二级缓存的应用将不怕n+1问题，因为即使第一次查询很慢，之后直接缓存命中也是很快的，刚好又利用了n+1。

## List和iterator区别

* List仅仅会填充二级缓存，却不能利用二级缓存。
* iterator可以读二级缓存，对于一条查询语句，它会先从数据库中找出所有符合条件的记录的ID，再通过ID去缓存找，对于缓存中没有的记录，再构造语句从数据库中查出。在缓存中没有命中的话，效率较低。
* 最好的办法就是：
  + 在应用启动时和数据被修改时使用list。平时则使用iterator。（只针对修改不频繁的数据！）

## 缓存机制的选用

* 一般开始开发并不使用缓存机制。
* 根据需求如果不能满足性能要求，才增加缓存。
  + 二级缓存：缓存数据内容变化频率不高的内容。
  + 查询缓存
* 很多系统经常在应用层增加缓存：
  + **OSCACHE**在J2EE中的应用

## 数据批量处理

* 建议：
  + 大批量数据的处理不要使用hibernate,优先考虑JDBC的批量处理。（一般使用JDBC）
  + 如果对性能要求极高，可以考虑**PL/SQL**

## 事务隔离级别

* **事务基本概念**
* ACID即是atomicity（原子性），consistency（一致性），isolation（隔离性）和durability（执久性）的首字母的缩写
  + 原子性表示一个事务内的所有操作是一个整体，要 么全部成功，要么全失败；
  + 一致性表示一个事务内有一个操作失败时，所有的更改过的数据都必须回滚到修改前的状态；
  + 隔离性：事务查看数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看中间状态的数据。
  + 持久性事务完成之后，它对于系统的影响是永久性的。
* 事务隔离级别从低到高：
  + **读取未提交（Read Uncommitted)**
  + **读取已提交(Read Committed)**
  + 可重复读（Repeatable Read)
  + 序列化（serializable)
* **读取未提交（Read Uncommitted)**
  + **这是最低的事务隔离级别，读事务不会阻塞读事务和写事务，写事务也不会阻塞读事务，但是会阻塞写事务。**
  + **写事务不阻塞读事务，可以读取未提交的数据，容易造成脏读**
  + **脏读现象：**
  + **脏读解决方案：** 
    - 如果在第一个事务提交前，任何其他事务不可读取其修改过的值，则可  以避免该问题。
* **读取已提交(Read Committed)**
  + **写事务就会阻塞读事务和写事务**，**但是读事务不会阻塞读事务和写事务。读事务不阻塞写事务，但是有可能造成不可重复（在同一个事务中，再次读取数据时【就是你的select操作】，所读取的数据，和第1次读取的数据，不一样了。查询的结果将是不确定的）。**
  + 不可重复读解决方案：
    - 锁住已经查询出来的记录！不让其他事物进行写操作
* 可重复读（Repeatable Read)
  + **读事务会阻塞写事务**，**但是读事务不会阻塞读事务**，**写事务会阻塞写事务和读事务。**
  + 读事务不阻塞读事务(针对的是记录而不是表)，可能会造成幻读问题
  + 幻读解决方案：
    - 解决办法是锁表，不让产生幻读的记录插入或删除。
    - 不过，一般不要考虑幻读问题。
* 序列化（serializable)
  + 此种隔离级别是最严格的隔离级别，**如果设置成这个级别，那么就不会出现以上所有的问题（脏读，不可重复读，幻影读）**。
  + 性能极低，一般不用！
* 我们一般采用读取已提交或者更低的事务隔离级别，配合各种并发访问控制策略来达到并发事务控制的目的。
* 如何使用：

<!-- 制定事务隔离级别 ：1,2,4,8。二进制中：0001, 0010,0100,1000。这样直接采用位运算即可。权限控制中经常采用二进制位运算-->

<property name=*"hibernate.connection.isolation">2</property>*

## 乐观锁和悲观锁

* 乐观锁**Optimistic Locking**
  + **顾名思义就是保持一种乐观的态度，我们认为系统中的事务并发更新不会很频繁，即使冲突了也没事，大不了重新再来一次。**
  + **它的基本思想就是每次提交一个事务更新时，我们想看看要修改的东西从上次读取以后有没有被其它事务修改过，如果修改过，那么更新就会失败。**
  + **常用实现方法：** 
    - **在我们的实体中增加一个版本控制字段，每次事务更新后就将版本(Version)字段：版本字段的值加1.**
      * **在实体类中增加@Version, private int version；getset 即可。**
* 悲观锁**Pessimistic Locking**
  + 基本思想就是每次一个事务读取某一条记录后，就会把这条记录锁住，这样其它的事务要想更新，必须等以前的事务提交或者回滚解除锁。
  + 悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）

**乐观锁和悲观锁的比较：**

* 乐观锁：
  + 优势：并发性好，性能较高。
  + 缺点：用户体验不好，录入了半天，提交时被告知已经修改！
* 悲观锁：
  + 优势：会锁住记录，一个用户修改完成前，其他用户不能操作该记录。
  + 缺点：并发性不好，性能不高。
* 对于悲观锁是针对并发的可能性比较大，而一般在我们的应用中*用乐观锁*足以。

**乐观锁和悲观锁测试代码：**

|  |
| --- |
| package test.cache;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Version;  @Entity  public class Account {  @Id  @GeneratedValue(strategy=GenerationType.AUTO)  private int id;    private String aname;  private int money;    @Version  private int version;    public int getVersion() {  return version;  }  public void setVersion(int version) {  this.version = version;  }    public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getAname() {  return aname;  }  public void setAname(String aname) {  this.aname = aname;  }  public int getMoney() {  return money;  }  public void setMoney(int money) {  this.money = money;  }  public Account(String aname, int money) {  super();  this.aname = aname;  this.money = money;  }    public Account() {  }    } |
| package test.cache;  import org.hibernate.LockMode;  import org.hibernate.Session;  import org.hibernate.Transaction;  import test2.HibUtil;  /\*\*  \* 测试事务隔离级别和乐观锁悲观锁  \* @author gaoqi  \*  \*/  public class Test7 {    static void saveAccount(){  Session s = HibUtil.getSession();  Transaction tx = s.beginTransaction();  Account a1= new Account("a",100);  Account a2= new Account("b",200);  s.save(a1);  s.save(a2);  tx.commit();  s.close();  }      /\*  \* 测试悲观锁  \*/  static void testPessimisticLock(){  Session s = HibUtil.getSession();    Transaction tx = s.beginTransaction();  Account a = (Account) s.get(Account.class, 1,LockMode.UPGRADE); //悲观锁。  //乐观锁，类中增加一个Version即可。hibernate自动管理！  Session s2 = HibUtil.getSession();  Transaction tx2 = s2.beginTransaction();  Account a2 = (Account) s2.get(Account.class, 1);  a2.setAname("abcde"); //切换属性，会造成死锁！  s2.saveOrUpdate(a2);  tx2.commit();  s2.close();    tx.commit();  s.close();  }    /\*  \* 测试乐观锁  \*/  static void testOptimisticLock(){  Session s = HibUtil.getSession();    Transaction tx = s.beginTransaction();  Account a = (Account) s.load(Account.class, 1); //乐观锁，类中增加一个Version即可。hibernate自动管理！    a.setAname(a.getAname()+"a");    Session s2 = HibUtil.getSession();  Transaction tx2 = s2.beginTransaction();  Account a2 = (Account) s2.get(Account.class, 1);  a2.setAname(a2.getAname()+"b");  s2.saveOrUpdate(a2);  tx2.commit();  s2.close();    tx.commit();  s.close();    }      public static void main(String[] args) {  saveAccount();    // testPessimisticLock();    testOptimisticLock();    }  } |

# 附录

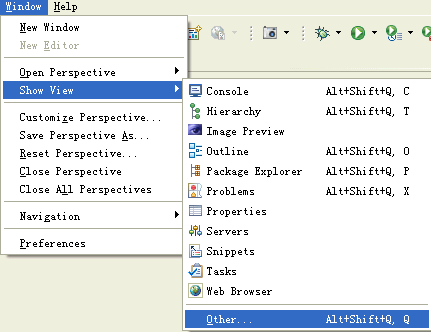
## Hibernate依赖库中相关jar包作用

|  |  |  |
| --- | --- | --- |
| 包 | 作用 | 说明 |
| jta.jar | 标准的JTA API | **必要** |
| commons-logging.jar | 日志功能 | 必要 |
| commons-collections.jar | 集合类 | 必要 |
| antlr.jar | ANother Tool for Language Recognition(antlr) | 必要 |
| dom4j.jar | XML配置和映射解释器 | 必要 |
| Hibernate3.jar | 核心库 | 必要 |
| asm.jar | ASM字节码库 | 如果使用“cglib”则必要 |
| asm-attrs.jar | ASM字节码库 | 如果使用“cglib”则必要 |
| ehcache.jar | EHCache缓存 | 如果没有其它的缓存，则它是必要的 |
| cglib.jar | CGLIB 字节码解释器 | 如果使用“cglib”则必要 |
| 以下包可选 | | |
| versioncheck.jar | 版本检查 |  |
| swarmcache.jar |  |  |
| jboss-cache.jar | TreeCache |  |
| jgroups.jar |  |  |
| xml-apis.jar | JAXP API |  |
| c3p0-0.9.1.jar | C3PO JDBC链接池 |  |
| connector.jar | JCA API |  |
| jboss-system.jar |  | 使用TreeCache时必要 |
| jacc-1\_0-fr.jar | JACC 库 |  |
| checkstyle-all.jar |  |  |
| junit.jar |  |  |
| javassist.jar | Javassist 字节码解释器 |  |
| ant-lanuncher.jar |  |  |
| jaas.jar | JAAS API |  |
| jdbc2\_0-stdext.jar | JDBC扩展API |  |
| ant-antlr.jar | Ant antlr支持 |  |
| jboss-jmx.jar |  |  |
| cleanimports.jar | cleanimports |  |
| xerces.jar | SAX parser | JDK版本低于1.4时必要 |
| jaxen-1.1-beta-7.jar | Jaxen | 如果想提高启动性能则去使用 |
| ant-junit.jar | Ant junit support |  |
| ant-swing.jar | ant swing support |  |
| ant.jar |  |  |
| proxool.jar | Proxool JDBC连接池 |  |
| concurrent.jar |  | 使用TreeCache需要 |
| syndiag2.jar |  |  |

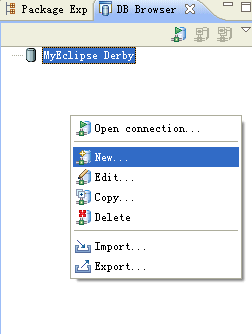
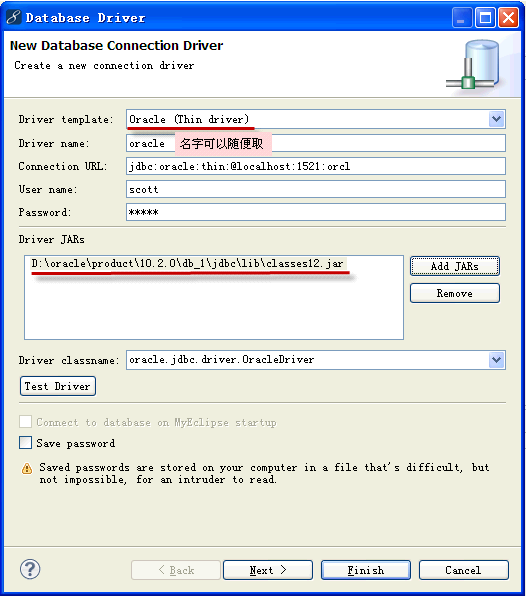
## 使用myeclipse自动开发hibernate程序(oracle 为例)

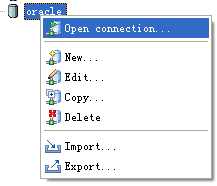
**由于已经有了表，我们可以使用myeclipse的自动生成pojo。操作如下：**

1. 打开db视图，建立连接

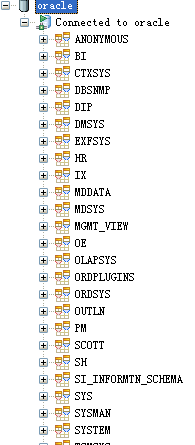




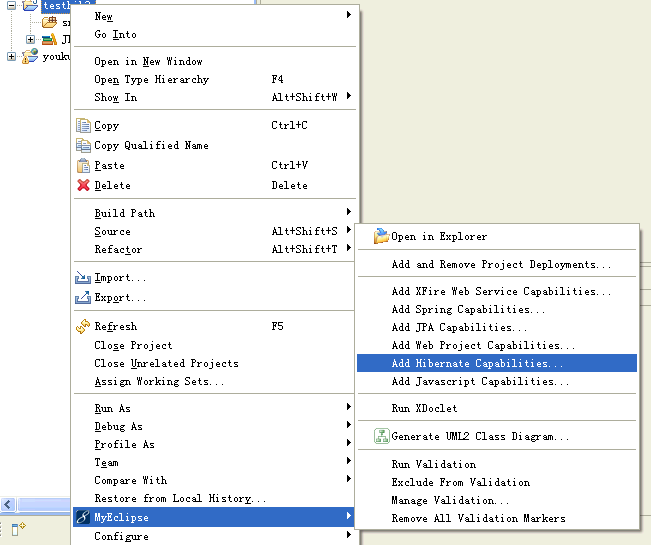
 

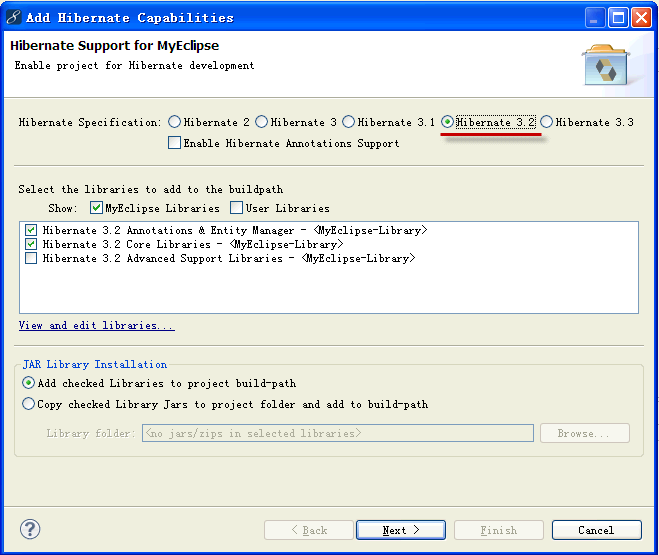


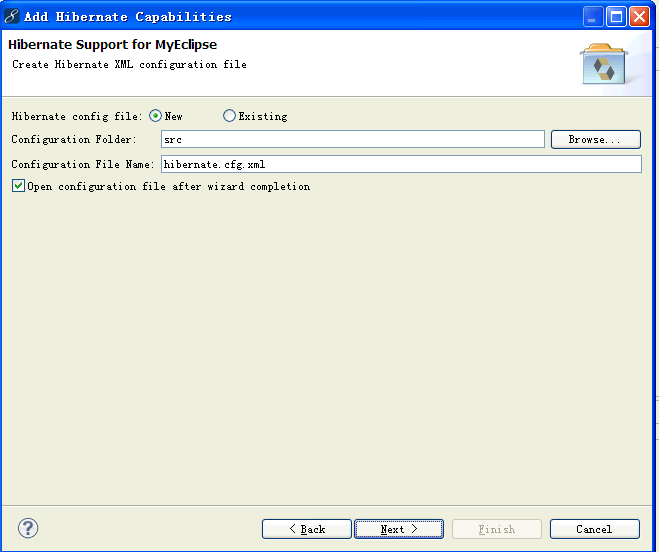
输入密码tiger，即可打开connection：

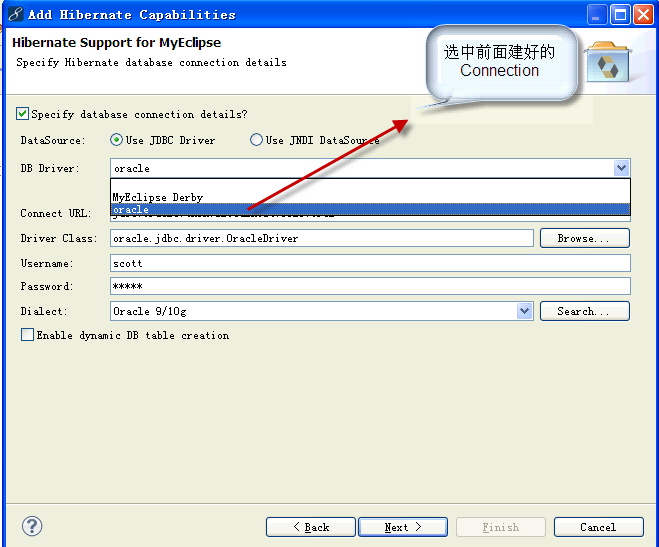


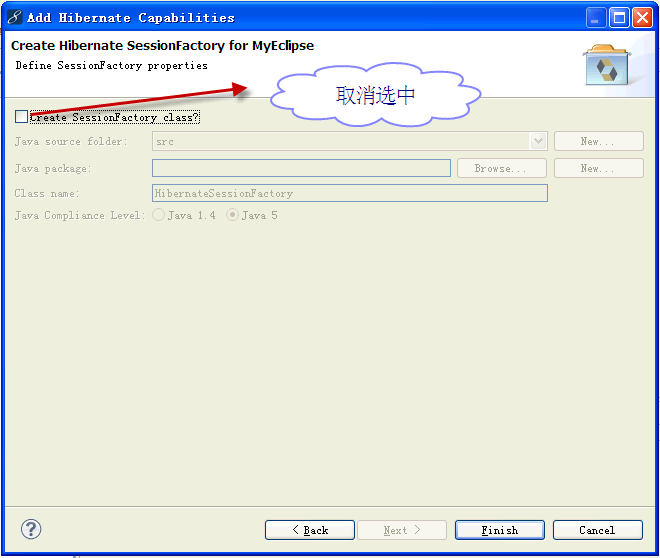
1. 建立新的java项目
2. 增加hibernate支持，右键单击项目🡪myeclipse🡪add hibernate capacities



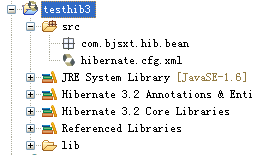




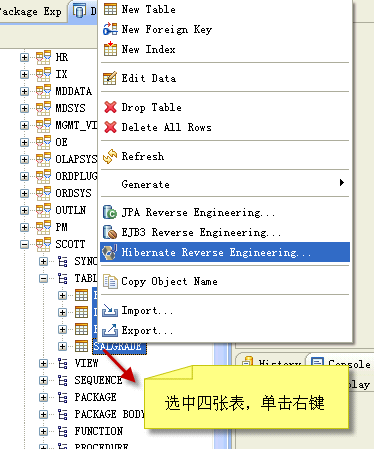
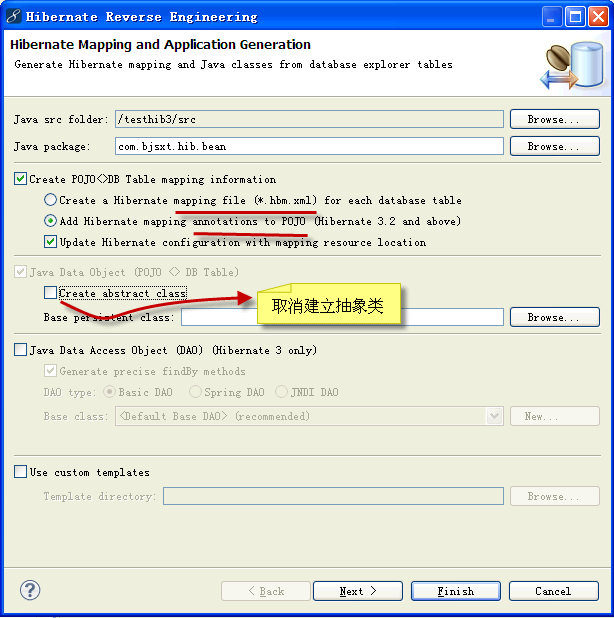


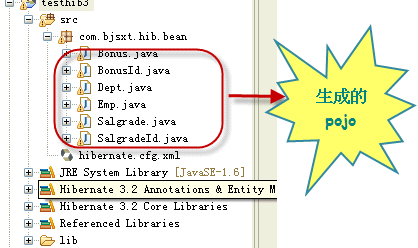


1. 完成上面三步，项目建立成功。可以在项目中建立包：com.bjsxt.hib.bean



1. 打开db browser，打开建好的connection：oracle，开打scott：



## Hibernate实践总结

* 数据量巨大，性能要求高，hibernate由于在ORM映射中对系统资源消耗也比较高，所以不适合。
* Hibernate适合：逻辑复杂，数据量不大。
* sessionFactory的创建非常消耗资源，整个应用一般只要一个。
* 将所有的集合属性配置设置为懒加载。 Hibernate2.x默认是false, hibernate3.x默认是true
* 在定义关联关系时，集合首选Set，如果集合中的实体存在重复，则选择List，数组性能最差。
* 在一对多的双向关联中，一般将集合的inverse设置为true，让集合的对方维护关联关系。
* HQL子句本身大小写无关，但是其中出现的类名和属性名必须注意大小写区分。
* 对大数据量查询时，慎用list() 返回查询结果
* 在性能瓶颈的地方使用JDBC。
* 使用双向关联。在大型应用中，几乎所有的关联必须在查询中可以双向导航。

# SPRING预备知识

## 工厂模式

工厂模式是最常用的设计模式之一。 工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A a=new A() 工厂模式也是用来创建实例对象的，所以以后new时就要多个心眼，是否可以考虑使用工厂模式，虽然这样做，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。

下面是关于简单工厂模式的示例代码：

|  |
| --- |
| /\*\*  \* 测试简单工厂模式  \* **@author** gaoqi  \*  \*/  **public** **class** TestSimpleFactoryPattern {  **public** **static** **void** main(String[] args) {  CarFactory factory = **new** CarFactory();  Car c = factory.createCar("audi");  c.run();  }  }  **class** CarFactory {  **public** Car createCar(String type){  **if**("audi".equalsIgnoreCase(type)){  **return** **new** Audi();  }**else** **if**("auto".equalsIgnoreCase(type)){  **return** **new** Auto();  }**else**{  **return** **null**;  }  }  }  **interface** Car {  **public** **void** run();  }  **class** Audi **implements** Car {  @Override  **public** **void** run() {  System.*out*.println("奥迪车跑的快一点！");  }  }  **class** Auto **implements** Car {  @Override  **public** **void** run() {  System.*out*.println("奥拓车跑的慢一点！");  }  } |

## 单例模式((singleton)

单例模式的意思就是只有一个实例。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。我们前面学习的很多类都是单例的。比如最典型的就是Servlet类！Servlet类被设计成单例，被所有线程共享！

Java Singleton模式为我们提供了这样实现的可能。使用Singleton的好处还在于可以节省内存，因为它限制了实例的个数，有利于Java垃圾回收（garbage collection）。

 单例模式也是一种比较常见的设计模式，它到底能带给我们什么好处呢？其实无非是三个方面的作用：

1、控制资源的使用，通过线程同步来控制资源的并发访问；

     2、控制实例产生的数量，达到节约资源的目的。

     3、作为通信媒介使用，也就是数据共享，它可以在不建立直接关联的条件下，让多个不相关的两个线程或者进程之间实现通信。

缓存、日志、创建比较耗时的往往可以设计成单例

|  |
| --- |
| /\*\*  \* 测试单例模式  \*  \* @author gaoqi  \*  \*/  public class TestSingleton {  public static void main(String[] args) {  ClassicSingleton s1 = ClassicSingleton.getInstance();  ClassicSingleton s2 = ClassicSingleton.getInstance();  System.out.println(s1==s2);  }  }  class ClassicSingleton {  private static ClassicSingleton instance = null;  private ClassicSingleton() { //构造器私有，外部不能调用  }  public static synchronized ClassicSingleton getInstance() {  if (instance == null) {  instance = new ClassicSingleton();  }  return instance;  }  } |

## 动态代理模式

讲解数据库连接池时，已经为大家做了讲解！可以参考相关资料！

* 动态代理模式是AOP的内部机制和核心！
* 其实所有代理都是一样，通过代理对象去调用目标对象

## 面向接口编程

在一个面向对象的系统中，系统的各种功能是由许许多多的不同对象协作完成的。在这种情况下，各个对象内部是如何实现自己的对系统设计人员来讲就不那么重要了；而各个对象之间的协作关系则成为系统设计的关键。小到不同类之间的通信，大到各模块之间的交互，在系统设计之初都是要着重考虑的，这也是系统设计的主要工作内容。面向接口编程就是指按照这种思想来编程。

接口从更深层次的理解，应是定义（规范，约束）与实现（名实分离的原则）的分离。

我们在一般实现一个系统的时候，通常是将定义与实现合为一体，不加分离的，我认为最为理解的系统设计规范应是所有的定义与实现分离，尽管这可能对系统中的某些情况有点繁烦。

接口的本身反映了系统设计人员对系统的抽象理解。

**面向接口实现了面向规范的编程，更加抽象，更加稳定！事实上，我们前面写的项目都是使用了面向接口编程！**

# spring 入门

[http://imgsrc.baidu.com/baike/abpic/item/027a45b5f11c1fd537d3ca36.jpg](http://baike.baidu.com/image/027a45b5f11c1fd537d3ca36)Rod Johnson在2002年编著的《Expert one to one J2EE design and development》一 书中，对Java EE正统框架臃肿、低效、脱离现实的种种现状提出了质疑，并积极寻求探索革新之道。以此书为指导思想，他编写了interface21框架，这是一个力图冲破Java EE传统开发的困境，从实际需求出发，着眼于轻便、灵巧，易于开发、测试和部署的轻量级开发框架。Spring框架即以interface21框架为基础，经过重新设计，并不断丰富其内涵，于2004年3月24日，发布了1.0正式版。同年他又推出了一部堪称经典的力作《Expert one-to-one J2EE Development without EJB》，该书在Java世界掀起了轩然大波，不断改变着Java开发者程序设计和开发的思考方式。在该书中，作者根据自己多年丰富的实践经验，对[EJB](http://baike.baidu.com/view/3542.htm)的各种笨重臃肿的结构进行了逐一的分析和否定，并分别以简洁实用的方式替换之。至此一战功成，Rod Johnson成为一个改变Java世界的大师级人物。

传统J2EE应用的开发效率低，应用服务器厂商对各种技术的支持并没有真正统一，导致J2EE的应用没有真正实现Write Once及Run Anywhere的承诺。Spring作为开源的中间件，独立于各种应用服务器，甚至无须应用服务器的支持，也能提供应用服务器的功能，如声明式事务等。

Spring致力于J2EE应用的各层的解决方案，而不是仅仅专注于某一层的方案。可以说Spring是企业应用开发的“一站式”选择，并贯穿表现层、业务层及持久层。然而，Spring并不想取代那些已有的框架，而是与它们无缝地整合。

## spring优势

* 依赖注入/控制反转(DI/IOC). 说白了就是根据配置文件动态组装复杂对象，实现了松耦合
* AOP（面向切面编程）
* 提供简易的服务抽象
* 集成管理各种框架

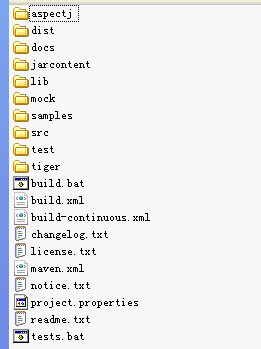
## spring如何选用

* 版本
  + 2.0(较广)
  + 2.5(较广)
  + 3.0(最新)
* 什么项目适合?
  + 小项目亦可
  + 中项目最适合
  + 大型分布式项目(EJB)

## 项目中配置spring

项目中配置spring按照如下步骤实现即可：

1. 去官方网站下载：<http://www.springframework.org/>
2. 下载完成后，解压：

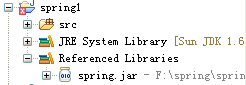


1. 打开dist目录，将spring.jar添加到项目依赖库中。

## 我们的第一个spring程序

我们开始使用spring，完成我们的第一个最简单的helloworld。步骤如下：

1. 建立java项目spring1(注：普通java项目即可)
2. 将dist目录下的spring.jar包添加到项目类依赖库中



1. 建立一个UserDao类：

|  |
| --- |
| **package** com.sxt.test;  **public** **class** UserDao {  **public** **void** add(String uname,String pwd){  System.*out*.println("增加一个用户!");  }  } |

1. 增加配置文件beans.xml，内容如下：

|  |
| --- |
| <?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"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"*>  <bean id=*"userDao"* class=*"com.sxt.test.UserDao"*></bean>  </beans> |

通过上面的配置文件，spring框架知道了UserDao类的存在！可以通过反射机制自动将UserDao类的对象new出！ 所以注意托管给spring的类必须符合基本的javabean规范：

1. 如果有属性，则必须有相应的get/set方法。
2. 必须要无参的构造器
3. 建立Test.java类

|  |
| --- |
| **package** com.sxt.test;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args){  // UserDao userDao = new UserDao();  // userDao.add("a", "123");    ApplicationContext context = **new** ClassPathXmlApplicationContext(**new** String[]{"beans.xml"});  UserDao userdao = (UserDao) context.getBean("userDao");  userdao.add("d", "222");  }  } |

上面的代码中，我们可以使用context.getBean("userDao")代替了new UserDao(这样的代码，也就是spring内部有个工厂类，替我们完成了new对象的操作！而这个工厂类是通过读取beans.xml文件知道了字符串”userDao”和com.sxt.test.UserDao类之间的关系！

1. 直接运行Test.java类即可。

**注意：还需要增加commons-logging.jar包！**

# IOC&DI控制反转/依赖注入

IOC和DI其实是同一个技术，同一个东西！为什么会有两个不同的名字，这是因为同一个东西从不同角度出发的不同的理解。

## IOC(Inverse Of Control )

控制反转是一种将组件依赖关系的创建和管理置于程序外部的技术。控制反转IoC意味着将你设计好的类交给系统去控制，而不是在你的类内部控制。将对象的创建和获取提取到外部。由外部容器提供需要的组件。理解这个关键在于如下两点：

1. 由容器控制程序之间的关系，而不是由代码直接控制
2. 由于控制权由代码转向了容器，所以称为反转

我们以下面的代码为例：

* UserServiceImpl中的：
  + **private UserDao userDao = new UserDaoImpl();**
  + **也就是说UserServiceImpl类中有哪个具体类属性由他自己决定。**
* 控制反转后：
  + UserServiceImpl中的：
    - **private UserDao userDao;**
    - **//没有指定具体是哪个类，只制定了接口，指定具体类的权利交给了spring框架：**

|  |
| --- |
| <bean id=*"userDao"* class=*"com.sxt.test.UserDaoImpl"*></bean>  <bean id=*"userService" class="com.sxt.test.UserService">*  <property name=*"userDao" ref="userDao"></property>*  </bean> |

## DI(Dependency Injection)

* Dependency Injection 依赖注入。如何理解DI呢？我们看下面的例子：
* UserServiceImpl中的：
  + **private UserDao userDao;**

//没有指定具体是哪个类，只制定了接口。具体是哪个类有spring框架来管理，来实例化，来赋值。好像这个属性的内容是spring框架从外部“注射进来”似地。

依赖注入理解的角度是从微观操作去做解释和理解。依赖注入所注入的对象通常是个可变因素 .

将变化的因素由代码转化到配置文件中。从而实现更加灵活的编程，实现最佳松耦合、实现组件的可插拔.

## IOC&DI的关系

控制反转和依赖注入其实是一回事。控制反转是宏观上看控制权的转换，依赖注入是微观上看控制权是如何转换的，只是谈的角度不同。

## IOC/DI实现技术之一：setter注入(设值注入)

* Setter方法注入(用的最多最广泛，一般用此方式)
  + 属性必须要有对应的set方法。
  + 配置文件内容同上一个spring程序

## IOC/DI实现技术之二：构造方法注入

* 构造方法注入
  + 所有依赖关系均在构造函数中设定！了解即可。

## IOC/DI实现技术之三：接口注入

这种方式要求我们的类必须实现容器给定的接口，然后容器会利用这个接口给我们的类注射它所依赖的类。这种方式，侵入性很强，目前不推荐使用！我们学习servlet时，HttpServletRequest 和 HttpServletResponse 实例由 Servlet Container注入。就是这种接口注入的方式！接口注入模式因为历史较为悠久，在很多容器中都已经得到应用。

要求：知道即可。

## Annotation实现注入操作

我们也可以使用Annotation来实现注入操作，提高我们写代码的灵活性和效率。spring中要使用annotation，需要在配置文件中增加：

|  |
| --- |
| **<beans xmlns="http://www.springframework.org/schema/beans"**  **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**  **xmlns:context="http://www.springframework.org/schema/context"**  **xsi:schemaLocation="http://www.springframework.org/schema/beans**  **http://www.springframework.org/schema/beans/spring-beans-2.5.xsd**  **http://www.springframework.org/schema/context**  **http://www.springframework.org/schema/context/spring-context-2.5.xsd">**    **<context:component-scan base-package=*"com.bjsxt"*/>**  **</beans>** |

我们只需要在相关类的属性前或者set方法前增加：

* @Resource （推荐）
  + 默认按名称装配
* @AutoWired（不推荐）
  + 默认按类型装配。根据属性类型，在容器中跟该类型匹配的bean。
* 示例：

@Resource(name="userDao2") //不指定名字，默认就是属性名。一般不需单独指定

userSeviceImpl中代码如下：

|  |
| --- |
| public class UserServiceImpl implements UserService {    @Resource  private UserDao userDao; // = new UserDaoImpl();    @Override  public void add(User user) {  userDao.add(user);  }  public UserDao getUserDao() {  return userDao;  }    public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }  } |

## IOC容器管理的bean作用域

IOC容器中的bean，我们可以使用单例模式、也可以使用工厂模式。甚至可以将bean放入request、session等作用域中。那么，如何处理bean?我们可以通过<bean>的scope属性设置，默认为singleton。

|  |  |
| --- | --- |
| **作用域** | **说明** |
| singleton（默认） | 一个bean类型对应一个对象 |
| prototype | 一个bean类型对应多个对象。每次从容器中取到的对象都是新对象 |
| request | 在基于web的spring applicationcontext才有效。 |
| session |
| global session |

## 自动扫描将bean纳入IOC容器管理(spring2.5新增)

spring  2.5新增自动扫描bean的功能，大大简化了配置文件中的配置内容。将程序员从繁琐的配置中解放出来！xml文件需要增加声明：

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-2.5.xsd">*    **<context:component-scan base-package=*"com.sxt"/>***  **<!-- 有了本配置，<context:annotation-config />可以不要。 -->**  **<!-- 会扫描该包下的子类，以及该包子包下的子类 -->**  </beans> |

配置<bean>的根本目的，就是让spring容器知道这个bean是要加载的。我们也可以通过注解来实现让spring知道哪些bean需要加载。

* + @Service：标记业务层bean
  + @Controller：标记控制层，比如：struts中的Action
  + @Repository：标记数据访问组件，即DAO类
  + @Component：当组件不好归类时使用该标记

为我们相关类增加注解后，spring会自动扫描com.sxt包及该包下所有子类！这样就将我们从繁琐的配置文件配置中解放出来！ 让我们使用spring更加简单！

## 完成自己的IOC容器

本作业是为了让我们加深对IOC的理解。两人一组完成，一人负责XML配置实现，一人负责annotation配置实现。并增加bean的作用域管理：prototype、singleton！

# AOP面向切面编程

## AOP介绍

* Aspect Oriented Programing 面向切面编程
  + 如果说面向对象编程是关注将需求功能划分为不同的并且相对独立，封装良好的类，并让它们有着属于自己的行为，依靠继承和多态等来定义彼此的关系的话；那么 面向方面编程则是希望能够将通用需求功能从不相关的类当中分离出来，能够使得很多类共享一个行为，一旦发生变化，不必修改很多类，而只需要修改这个行为即 可。
* AOP主要应用于日志记录，性能统计，安全控制,事务处理等方面。

前面我们讲的struts2中的拦截器，就是AOP的一种实现！

## AOP基本概念

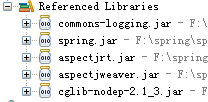
* 连接点（Joinpoint）
  + 在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。 在Spring AOP中，一个连接点 总是 代表一个方法的执行。
* 切入点（Pointcut）
  + 匹配连接点（Joinpoint）的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。
* 通知（Advice）
  + 在切面的某个特定的连接点（Joinpoint）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。
* 切面（Aspect）
  + 一个关注点的模块化，这个关注点可能会横切多个对象。包含了：连接点、切入点、通知。可以通过@Aspect定义为一个类
* 目标对象（Target Object）
  + 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做被通知（advised） 对象。
* 织入（Weaving）
  + 把切面（aspect）连接到其它的应用程序类型或者对象上，并创建一个被通知（advised）的对象。

## 通知的类型

* 前置通知（Before advice）
  + 在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
* 返回后通知（After returning advice）
  + 在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
* 抛出异常后通知（After throwing advice）
  + 在方法抛出异常退出时执行的通知。
* 后通知（After （finally）advice）
  + 当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
* 环绕通知（Around Advice）
  + 包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

## spring中如何配置AOP

增加jar包，列表如下：



在spring中使用AOP需要在配置文件中增加如下内容：

|  |
| --- |
| <?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:aop=*"http://www.springframework.org/schema/aop"***  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  ***http://www.springframework.org/schema/aop***  ***http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">***  **<aop:aspectj-autoproxy /> <!– 支持AOP的注解方式-->**  </beans> |

### Annotation方式配置AOP

* 切面类声明
  + @Aspect
  + @Component //这个必须，告诉本类交给spring初始化
* 切入点声明
* 切入点语法示例和细节：
  + \*代替任意字符。 ..表示任意个参数。 ！表示非
* 各种通知类型的定义

|  |
| --- |
| package com.sxt.interceptor;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.After;  import org.aspectj.lang.annotation.AfterReturning;  import org.aspectj.lang.annotation.AfterThrowing;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.aspectj.lang.annotation.Pointcut;  import org.springframework.stereotype.Component;  import com.sxt.bean.User;  /\*\*  \* 日志拦截器  \*/  @Aspect  @Component  public class TestInterceptor {  @Pointcut("execution(public \* com.sxt.test.\*.test\*(..))") //声明一个名字为log的切入点  private void log(){  }    @Before("log()"+"&&args(user)") //使用pointcut-->log .前置通知.处理传入的参数跟下面的形参保持一致即可。  private void before(User user){  System.out.println("方法执行前："+user.getUname());  user.setUname("lisi");  System.out.println("方法执行前！");  }    @After("log()") //最终后置通知。不管有无异常，最终都会执行！  private void after(){  System.out.println("方法执行后！");  }  @AfterReturning("log()") //方法正常结束，后置通知  private void afterReturning(){  System.out.println("方法执行后！after returning!");  }  @AfterThrowing("log()") //抛出异常会执行，异常通知  private void afterThrowing(){  System.out.println("方法执行后！after throwing!");  }      @Around("log()") //环绕通知在一个方法执行之前和之后执行  private Object around(ProceedingJoinPoint pjp){  System.out.println("环绕通知,方法执行前！");  Object obj = null;  try {  obj = pjp.proceed();  } catch (Throwable e) {  e.printStackTrace();  }  System.out.println("环绕通知,方法执行后！");  return obj;  }  } |
| package com.sxt.test;  import org.springframework.stereotype.Component;  import com.sxt.bean.User;  @Component("userDao")  public class UserDao {    public void test1(User user){  System.out.println(user.getUname());  System.out.println("UserDao.test1()");  }  public void test2(){  System.out.println("UserDao.test2()");  }    public void aaa(){  System.out.println("UserDao.aaa()");  }  } |
| **package** com.sxt.bean;  **public** **class** User {  **private** **int** id;  **private** String uname;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }      } |
| **package com.sxt.test;**  **import org.springframework.context.ApplicationContext;**  **import org.springframework.context.support.ClassPathXmlApplicationContext;**  **import com.sxt.bean.User;**  **public class Test {**  **public static void main(String[] args) {**  **ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{"beans.xml"});**  **UserDao us = (UserDao) context.getBean("userDao");**  **User u = new User();**  **u.setUname("zhangsan");**  **us.test1(u);**  **us.test2();**  **us.aaa();**  **}**  **}** |

### XML方式配置AOP

配置文件中配置：

|  |
| --- |
| <?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:aop=*"http://www.springframework.org/schema/aop"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-2.5.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-2.5.xsd*  *"*>  <bean id=*"testInterceptor"* class=*"com.sxt.interceptor.TestInterceptor"*></bean>  <bean id=*"userDao"* class=*"com.sxt.test.UserDao"*></bean>  <aop:config>  <aop:pointcut expression=*"execution(public !java.lang.String com.sxt.test.\*.test\*(com.sxt.bean.User,..))"* id=*"log"*/>  <aop:aspect id=*"logAspect"* ref=*"testInterceptor"*>  <aop:before method=*"before"* arg-names=*"n"* pointcut=*"execution(public !java.lang.String com.sxt.test.\*.test\*(com.sxt.bean.User,..)) and args(n)"* />  <aop:after method=*"after"* pointcut-ref=*"log"* />  </aop:aspect>  </aop:config>      </beans> |

# 框架整合

## 整合hibernate

* 增加相关jar包
  + 见项目：spring-hiberante-3所引入的包。



* + 注意：**宁少勿多！**如果提示ClassNotFoundException可以将相关jar包再加入。
* 配置文件内容：

|  |
| --- |
| <?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:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-2.5.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-2.5.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-2.5.xsd  ">  <bean id="dataSource"  class="org.apache.commons.dbcp.BasicDataSource">  <property name="driverClassName"  value="com.mysql.jdbc.Driver">  </property>  <property name="url" value="jdbc:mysql://localhost:3306/testhib"></property>  <property name="username" value="root"></property>  <property name="password" value="123456"></property>  </bean>  <bean id="sessionFactory"  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">  <property name="dataSource">  <ref bean="dataSource" />  </property>  <property name="hibernateProperties">  <props>  <!-- key的名字前面都要加hibernate. -->  <prop key="hibernate.dialect">  org.hibernate.dialect.MySQLDialect  </prop>  <prop key="hibernate.show\_sql">true</prop>  <prop key="hibernate.hbm2ddl.auto">create</prop>  <!--  -->  </props>  </property>  <property name="packagesToScan">  <value>com.cssxt.po</value>  </property>  </bean>  </beans> |

* packageToScan的使用

|  |
| --- |
| <property name=*"packagesToScan"*>  <value>com.sx\*.po</value>  </property> |

### 事务管理

* 两种方式：
  + Annotation
    - 加入annotation.xsd
    - 加入txManager bean
    - 加入<tx:annotaion driven>
    - 在需要事务的方法上加上：@Transactional
    - 需要注意的是：使用sessionFactory.getCurrentSession而不是sessionFactory.openSession
  + xml(推荐使用！一次性设置多个方法)
* 事务在DAO层还是service层?
  + 推荐在service层使用！
* 事务传播属性
  + REQUIRED(默认为它，一般用它！)
    - 声明方法需要一个事务运行。如果方法运行时，已经处在事务中，那么加入该事务，否则创一个新的事务
  + NOT\_ SUPPORTED
    - 声明方法不需要事务。如果方法没有在事务中，则不会创建新事务；如果方法已经在一个事务中，则该事务挂起，直到该方法执行完毕后再恢复。
  + REQUIRESNEW
    - 声明方法不管在不在一个事务中，总会创建新事务。
  + MANDATORY
    - 声明方法只能在一个已经存在的事务中。如果不在，容器抛出异常
  + SUPPORTS
    - 如果声明方法在事务中执行，则成为该事务的一部分；在事务外调用，则跟事务无关
  + NEVER
    - 声明方法绝对不会在事务中执行。否则，抛出异常
  + NESTED
    - 声明方法如果在一个活动事务中，则在事务内部运行一个嵌套事务；否则，新建一个事务
* 注意事项：
  + 不管什么异常，如果我们catch了，spring就会放弃管理

我们需要在spring配置文件中增加：

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  **xmlns:tx=*"http://www.springframework.org/schema/tx"***  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  ***http://www.springframework.org/schema/tx***  ***http://www.springframework.org/schema/tx/spring-tx-2.5.xsd***  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-2.5.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-2.5.xsd*  *">* |

## annotation方式的事务管理

如果只用annotation：

|  |
| --- |
| <bean id=*"txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager" >*  <property name=*"sessionFactory" ref="sessionFactory"></property>*  </bean>  <tx:annotation-driven transaction-manager=*"txManager" />* |

我们的类中使用示例：

|  |
| --- |
| **@Transactional(propagation=Propagation.*REQUIRED)*** *//默认就是REQUIRED，可不加。*  **public void add(User user) {**  System.*out.println("UserDaoImpl.add()");*  Session session = sessionFactory.getCurrentSession(); //.openSession();  session.save(user);  //session.close(); //session关闭问题。托管给spring控制，不需要手工关闭！  } |

## XML配置方式管理事务(推荐)

配置代码如下：

|  |
| --- |
| <bean id=*"txManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"* >  <property name=*"sessionFactory"* ref=*"sessionFactory"*></property>  </bean>  <tx:annotation-driven transaction-manager=*"txManager"* />  <aop:config>  <aop:pointcut expression=*"execution(public \* com.sxt.service.\*.\*(..))"* id=*"bussinessService"*/>  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"bussinessService"* />  </aop:config>  <tx:advice id=*"txAdvice"* transaction-manager=*"txManager"* >  <tx:attributes>  <tx:method name=*"get\*"* read-only=*"true"* propagation=*"NOT\_SUPPORTED"* />  <!-- get开头的方法不需要在事务中运行 。  有些情况是没有必要使用事务的，比如获取数据。开启事务本身对性能是有一定的影响的-->  <tx:method name=*"\*"*/> <!-- 其他方法在实务中运行 -->  </tx:attributes>  </tx:advice> |

### HibernateTemplate

HibernateTemplate类可让我们将Hibernate的使用模板化，使我们对hibernate的调用更加简单！使用他，我们只需要在配置文件中增加：

|  |
| --- |
| <bean id=*"hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate" >*  <property name=*"sessionFactory" ref="sessionFactory"></property>*  </bean> |

Dao类中增加代码如下：

|  |
| --- |
| @Resource  **private HibernateTemplate hibernateTemplate;**  **Get、set方法限于篇幅原因，省略** |

使用示例：

|  |
| --- |
| **public void add(User user) {**  hibernateTemplate.save(user);  } |

HibernateTemplate中常见的方法有：

|  |
| --- |
| * + void delete(Object entity)：删除指定持久化实例   + deleteAll(Collection entities)：删除集合内全部持久化类实例   + find(String queryString)：根据HQL查询字符串来返回实例集合   + get(Class entityClass, Serializable id)：根据主键加载特定持久化类的实例   + save(Object entity)：保存新的实例   + saveOrUpdate(Object entity)：根据实例状态，选择保存或者更新   + update(Object entity)：更新实例的状态，要求entity是持久状态   + **setMaxResults(int maxResults)：设置分页的大小(无setFirstResult方法)** |

我们在项目中使用spring框架技术，就是希望将一些繁琐的、重复性的东西交给spring打理，我们只关注于业务逻辑本身。Spring管理hibernate后，打开session、关闭session、打开事务、提交事务等这些重复操作统统交给spring处理。这时候我们就不能自己手工得到session对象，类似于：openSession()这样处理。Spring将session相关的处理完全交给了HibernateTemplate类。我们通过这使用HibernateTemplate可以完全将session等托管给spring，而不需要我们处理其他事情。同时，HibernateTemplate还提供了很多有用的工具方法。但是，HibernateTemplate有个问题就是我们不同通过它直接得到session对象，如果我们必须要获取session对象才能正常完成代码，这时候就要用到HibernateTemplate提供的HibernateCallback技术。  
 HibernateTemplate提供HibernateCallback，就是为了满足这种使用了HibernateTemplate的情况下，仍然需要直接访问Session的需求而来的。它提供了在HibernateTemplate里面直接访问Session的能力，程序员不必人为的管理session对象, 这个就是使用HibernateCallback的原因。

HibernateTemplate的典型用法：

1. 需要直接获得session对象的处理方式(比如：分页处理)

|  |
| --- |
| **public void add(final User user) {**  //hibernateTemplate.save(user);  hibernateTemplate.execute(**new HibernateCallback() {**  @Override  **public Object doInHibernate(Session session) throws HibernateException,**  SQLException {  session.save(user);  **return null;**  }  });  } |

1. 不需要直接获得session对象的情况

|  |
| --- |
| **public void getAll(){**  hibernateTemplate.find("from User");  List list = hibernateTemplate.find("from User where uname=? and id=?", **new Object[]{"ddd3",1});**  } |

### 模板方法模式/回调方法/钩子函数

Java中的回调方法(模板方法模式)：

|  |
| --- |
| **public** **void** execute(MyHibernateCallback callback){  Session s = **null**;  Transaction tx = **null**;  **try** {  s = sessionFactory.openSession();  tx = s.beginTransaction();  callback.doInHibernate(s);  tx.commit();  }**finally**{  **if**(s!=**null**){  s.close();  钩子  }  }  } |

|  |
| --- |
| template.execute(**new** MyHibernateCallback() {  @Override  **public** Object doInHibernate(Session session) {  session.save(user);  **return** **null**;  }  }); |

### HibernateDaoSupport

封装了HibernateTemplate！常见用法如下：

|  |
| --- |
| **public** **class** UserDaoImpl3 **extends** HibernateDaoSupport {  **public** **void** add(User u) {  **this**.getHibernateTemplate().save(u);  }  } |
| **public** **class** UserDaoImpl3 **extends** HibernateDaoSupport {  **public** **void** add(User u) {  Session s = **this**.getSession();  s.save(u);  releaseSession(s); //手动释放session资源  }  } |

### OpenSessionInView管理session

OpenSessionInViewFilter是Spring提供的一个针对Hibernate的一个支持类，其主要意思是在发起一个页面请求时打开 Hibernate的Session，一直保持这个Session，直到这个请求结束，具体是通过一个Filter来实现的。

由于Hibernate引入了Lazy Load特性，使得脱离Hibernate的Session周期的对象如果再想通过getter方法取到其关联对象的值，Hibernate会抛出一个 LazyLoad的Exception。所以为了解决这个问题，Spring引入了这个Filter，使得Hibernate的Session的生命周期变长。

OpenSessionInViewFilter：org.springframework.orm.hibernate3.support.OpenSessionInViewFilter]是 Spring提供的一个针对Hibernate的一个支持类，其主要意思是在发起一个页面请求时打开Hibernate的Session，一直保持这个 Session，直到这个请求结束，具体是通过一个Filter来实现的。

　　由于Hibernate引入了Lazy Load特性，使得脱离Hibernate的Session周期的对象如果再想通过getter方法取到其关联对象的值，Hibernate会抛出一个 LazyLoad的Exception。所以为了解决这个问题，Spring引入了这个Filter，使得Hibernate的Session的生命周期 变长。

有两种方式可以配置实现OpenSessionInView，分别是 OpenSessionInViewInterceptor和OpenSessionInViewFilter，功能完全相同，只不过一个在 web.xml配置，另一个在application.xml配置而已。

我们可以在web.xml中配置opensessioninview，代码如下：

|  |
| --- |
| <!-- 配置Spring自动管理Session. **要配置到struts过滤器之前！-**->  <filter>  <filter-name>hibernateSessionFilter</filter-name>  <filter-class>  org.springframework.orm.hibernate3.support.OpenSessionInViewFilter  </filter-class>  </filter>  <filter-mapping>  <filter-name>hibernateSessionFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

### ThreadLocal模式管理session

我们知道Session是由SessionFactory负责创建的，而SessionFactory的实现是线程安全的，多个并发的线程可以同时访问一 个SessionFactory并从中获取Session实例，那么Session是否是线程安全的呢？很遗憾，答案是否定的。

早在Java1.2推出之时，Java平台中就引入了一个新的支持：java.lang.ThreadLocal，给我们在编写多线程程序时提供了 一种新的选择。ThreadLocal是什么呢？其实ThreadLocal并非是一个线程的本地实现版本，它并不是一个Thread，而是thread local variable(线程局部变量)。也许把它命名为ThreadLocalVar更加合适。线程局部变量(ThreadLocal)其实的功用非常简单， 就是为每一个使用某变量的线程都提供一个该变量值的副本，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像 每一个线程都完全拥有一个该变量。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单，在ThreadLocal类中有一个Map，用于存储每一个线程的变量的副本。

比如下面的示例实现(为了简单，没有考虑集合的泛型)：

|  |
| --- |
| public  class  ThreadLocal  { 　   private  Map  values  =  Collections.synchronizedMap(new  HashMap()); 　   public  Object  get()  { 　　 Thread  currentThread  =  Thread.currentThread();   　   Object  result  =  values.get(currentThread);   　　   if(result  ==  null&&!values.containsKey(currentThread))  { 　　　    result  =  initialValue(); 　　　    values.put(currentThread,  result);   　　       } 　　       return  result;   　       } 　   public  void  set(Object  newValue)  { 　　   values.put(Thread.currentThread(),  newValue); 　      }      public  Object  initialValue()  { 　　   return  null;   　    }    } |

那麽具体如何利用ThreadLocal来管理Session呢？Hibernate官方文档手册的示例之中，提供了一个通过ThreadLocal维护Session的好榜样：

|  |
| --- |
| public  class  HibernateUtil  {   public  static  final  SessionFactory  sessionFactory;     static  {       try  {         sessionFactory  =  new  Configuration().configure().buildSessionFactory();         } catch (Throwable  ex) {           throw  new  ExceptionInInitializerError(ex);            }          }    public static final ThreadLocal<Session>session=new ThreadLocal<Session>();    public  static  Session  currentSession()  throws  HibernateException  {        Session  s  =  session.get();        if(s  ==  null)  {          s  =  sessionFactory.openSession();          session.set(s);           }          return  s;          }    public  static  void  closeSession()  throws  HibernateException  {          Session  s  =  session.get();         if(s  !=  null)  {            s.close();           }           session.set(null);         }      } |

只要借助上面的工具类获取Session实例，我们就可以实现线程范围内的Session共享，从而避免了线程中频繁的创建和销毁Session实例。当 然，不要忘记在用完后关闭Session。写到这里，想再多说一些，也许大多数时候我们的DAO并不会涉及到多线程的情形，比如我们不会将DAO的代码写 在Servlet之中，那样不是良好的设计，我自己通常会在service层的代码里访问DAO的方法。但是我还是建议采用以上的工具类来管理 Session，毕竟我们不能仅仅考虑今天为自己做什么，还应该考虑明天为自己做什么！

# 整合struts2

整合struts2的过程相对比较简单，步骤如下：

* 导入相关jar包：
  + 增加struts2/lib的commons-logging-1.0.4.jar、freemarker-2.3.13.jar、ognl-   2.6.11.jar、struts2-core-2.1.6.jar、xwork-2.1.2.jar、commons-fileupload- 1.2.1.jar，commons-io1.3.2.jar, **struts2-spring-plugin-2.1.8.1.jar**这几个包引入到工程里。

在struts.xml中增加：

|  |
| --- |
| <?xml version=*"1.0" encoding="UTF-8" ?>*  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">  <struts>  **<constant name=*"struts.objectFactory" value="spring" />***  <package name=*"example" namespace="/" extends="struts-default">*  <!--  <action name="user" class="com.cssxt.action.UserAction">  <result name="success">index.jsp</result>  </action>  -->  <!--  <action name="\*\_\*" class="com.gao.bbs.action.{1}Action" method="{2}">  <result name="success">{1}\_{2}\_ok.jsp</result>  <result name="error">{1}\_{2}\_error.jsp</result>  <result name="input">{1}\_{2}\_input.jsp</result>  </action>  -->  </package>  <!--  <include file="admin.xml"></include>  <include file="front.xml"></include>  -->  </struts> |

在web.xml中配置struts2跟以前一致：

|  |
| --- |
| <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> |

# SSH框架整合总结

1. 整合hibernate
   1. 增加相关jar包
   2. 增加spring配置文件：applicationContext.xml，文件内容如下：
      1. 配置组件扫描。让spring管理basePackage和他的子包下所有的类。
      2. 配置dataSource数据源
      3. 配置sessionFactory
      4. 配置事务管理(事务增加在service层！)
      5. 配置HibernateTemplate
2. 整合struts
   1. 导入jar包
   2. 配置web.xml
      1. 配置<context-param>指定spring配置文件的路径和名字！
      2. 配置opensessionInview过滤器
      3. 配置struts过滤器
      4. 增加spring上下文加载的监听器
   3. 增加struts.xml：

增加<constant name=*"struts.objectFactory"* value=*"spring"* />

# spring抽象服务

### 邮件处理

自学！并写出完整技术文档！

面试题：

1. 如何发送邮件？
   1. 使用JavaMail即可。

### 任务调度(Quatz)

自学！并写出完整技术文档！

面试题：

1. 我想定时处理某一个任务，怎么办?
   1. 通过Quatz框架。
   2. 通过Timer、TimerTask类来实现。