



# 菜鸟学SSH

---

极客学院出版

# 前言

---

SSH 为 struts+spring+hibernate的一个集成框架，是目前较流行的一种Web应用程序开源框架.本文主要针对初学者讲解了此部分内容.

## | 读者

本文适用于初学SSH人员。

## | 预备知识

学习本教程之前强烈建议你先熟悉一下，j2se、jsp。

致谢

内容撰写: <http://blog.csdn.net/column/details/myssh.html>

作者: 刘水镜

更新日期	更新内容
2015-06-11	菜鸟学 SSH

## 目录

---

前言 .....	1
第 1 章 Struts实现简单登录（附源码） .....	3
第 2 章 Struts2 国际化手动切换版 .....	7
第 3 章 Struts2 国际化自动检测浏览器语言版 .....	12
第 4 章 Struts2 拦截器 .....	15
第 5 章 Struts2 上传文件 .....	21
第 6 章 Spring 事务管理 .....	27
第 7 章 Spring jar包详解 .....	33
第 8 章 Hibernate 对象的三种状态 .....	37
第 9 章 Hibernate——Session之save()方法 .....	41
第 10 章 Hibernate 核心接口 .....	44
第 11 章 Hibernate之SchemaExport+配置文件生成表结构 .....	48
第 12 章 Hibernate与Spring 配合生成表结构 .....	53
第 13 章 Spring 容器IOC解析及简单实现 .....	58
第 14 章 Spring 容器AOP的实现原理——动态代理 .....	63
第 15 章 简单模拟Hibernate实现原理 .....	70
第 16 章 Struts2 内部是如何工作的 .....	75
第 17 章 基于注解的SSH将配置精简到极致 .....	80
第 18 章 Hibernate动态模型+JRebel实现动态创建表 .....	89
第 19 章 提高用户体验之404处理 .....	92



1



## Struts实现简单登录（附源码）



从今天开始，一起跟各位聊聊java的三大框架——SSH。先从Struts开始说起，Struts对MVC进行了很好的封装，使用Struts的目的是为了帮助我们减少在运用MVC设计模型来开发Web应用的时间。如果我们想混合使用Servlets和JSP的优点来建立可扩展的应用，struts是一个不错的选择。今天通过一个简单的例子来说说Struts。

登录页面：这里面没啥东西，主要就是将action命名成“.do”的形式，让Struts通过配置文件来执行相应操作。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=GB18030">
<title>Insert title here</title>
</head>
<body>
  <form action="login.do" method="post">
    用户: <input type="text" name="username"><br>
    密码: <input type="password" name="password"></br>
    <input type="submit" value="登录">
  </form>
</body>
</html>
```

PS：表单中的name值必须跟对应的actionform的get、set属性一致。

web.xml：它的作用就是告诉Struts它的配置文件（struts-config.xml）在哪，让Struts可以找到，还有执行的动作（.do），还有加载顺序之类的。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <welcome-file-list>
    <welcome-file>login.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
```

```

<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>

```

struts-config.xml: 从名字上就能看出来, 这是Struts的配置文件。form-bean用来接收提交的表单数据, action-mappings用来执行相应的动作。

```

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

<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
  <form-beans>
    <form-bean name="loginForm" type="com.lsj.struts.LoginActionForm"/>
  </form-beans>

  <action-mappings>
    <action path="/login"
      type="com.lsj.struts.LoginAction"
      name="loginForm"
      scope="request"
    >
      <forward name="success" path="/login_success.jsp" />
      <forward name="error" path="/login_error.jsp" />
    </action>
  </action-mappings>
</struts-config>

```

这个类就是用来处理用户登录的业务逻辑

```

package com.lsj.struts;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

```

```
public class LoginAction extends Action {

    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        LoginActionForm laf = (LoginActionForm)form;
        String username = laf.getUsername();
        String password = laf.getPassword();

        UserManager userManager = new UserManager();
        try {
            userManager.login(username, password);
            return mapping.findForward("success");
        } catch (UserNotFoundException e) {
            e.printStackTrace();
            request.setAttribute("msg", "用户不能找到, 用户名称=【 " + username + " 】");
        } catch (PasswordErrorException e) {
            e.printStackTrace();
            request.setAttribute("msg", "密码错误");
        }
        return mapping.findForward("error");
    }

}
```

通过上面这个简单的小实例，大家可以清晰的看出，两个配置文件起到了一个很好的串联作用。也正是因为有配置文件，所以才是程序变得非常的灵活。这个小例子比较简单，配置文件承担的责任不是很多，后面的东西会更多的用到配置文件，体会就会更加的深刻了。最后为大家附上源码方便一起研究讨论。



2

## Struts2 国际化手动切换版





国际化(internationalization,i18n)和本地化(localization,l10n)指让产品（出版物，软件，硬件等）能够适应非本地环境，特别是其他的语言和文化。程序在不修改内部代码的情况下，能根据不同语言及地区显示相应的界面。

## 国际化原理：

国际化资源文件：用不同国家的语言描述相同的信息，并放在各自对应的.properties属性文件中，程序根据运行时环境决定加载哪个文件。国际化主要通过以下类完成：java.util.Locale：对应一个特定的国家/区域、语言环境。java.util.ResourceBundle：用于加载一个资源包。I18nInterceptor：struts2所提供的国际化拦截器，负责处理Locale相关信息。国际化流程：程序得到当前运行环境的国家/区域、语言环境并存放于Locale，ResourceBundle根据Locale中信息自动搜索对应的国际化资源文件并加载。当某个Action被执行前，I18nInterceptor负责检测Locale相关信息来寻找对应的国际化资源

先来看一下实例的效果图：

默认中文：

英文中文

姓名:

密码:

点击英文，页面变成英文显示：

英文中文

name:

pass:

接下来看具体的实现：

LoginAction

```
package com.lsj.action;

import java.util.Locale;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport {
```

```

private static final long serialVersionUID = 1L;

private String flag;
public String getFlag() {
    return flag;
}
public void setFlag(String flag) {
    this.flag = flag;
}

public String ha() throws Exception {
    this.flag=ServletActionContext.getRequest().getParameter("flag");
    Locale l = Locale.getDefault();
    if(this.flag==null){
        l = new Locale("zh", "CN");
    }else if (this.flag.equals("2")) {
        l = new Locale("zh", "CN");
    } else if (this.flag.equals("1")) {
        l = new Locale("en", "US");
    }
    ActionContext.getContext().setLocale(l);
    return "success";
}
}

```

web.xml

```

<?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>

```

```

<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

#### struts.xml

```

<?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>

  <!-- 自动发布 -->
  <constant name="struts.devMode" value="true" />
  <!-- 过滤器 -->
  <constant name="struts.i18n.encoding" value="GBK"/>
  <!-- 配置i18n -->
  <constant name="struts.custom.i18n.resources" value="message"></constant>

  <!-- i18n控制 -->
  <package name="i18n" namespace="/" extends="struts-default">
    <action name="i18n" class="com.lsj.action.LoginAction" method="ha">
      <result name="success">
        /i18nlogin.jsp
      </result>
      <result name="input">
        /i18nlogin.jsp
      </result>
    </action>
  </package>

</struts>

```

#### login.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";
%>

```







```

<html>
<head>
  <base href="<%=basePath%>">
  <title>My JSP 'i18nlogin.jsp' starting page</title>
  <meta http-equiv="pragma" content="no-cache">
  <meta http-equiv="cache-control" content="no-cache">
  <meta http-equiv="expires" content="0">
  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
  <meta http-equiv="description" content="This is my page">
</head>

<body>
<s:i18n name="local.message">
  <s:form action="/i18n.action" method="post">
    <s:textfield name="user.userName" label="%{getText('login.name')}" />
    <s:password name="user.userPassword" label="%{getText('login.pass')}" />
    <s:submit value="%{getText('login.submit')}"></s:submit>
    <a href="i18n.action?flag=1">英文</a>
    <a href="i18n.action?flag=2">中文</a>
  </s:form></s:i18n>
</body>
</html>

```

需要引入的jar包：

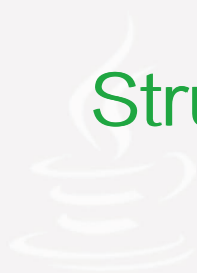
-  commons-fileupload-1.2.1.jar
-  commons-io-1.3.2.jar
-  freemarker-2.3.15.jar
-  ognl-2.7.3.jar
-  struts2-core-2.1.8.1.jar
-  xwork-core-2.1.6.jar

OK到这里就实现了一个简单的国际化的小实例，有的说弄国际化没多大必要，因为咱们做的东西几乎都只是用于国内。我想说的是：为什么就觉得我们做的东西只会给我们自己用？为什么就不会觉得随着我们不断的努力，我们做的东西会越来越好，然后我们的东西会逐渐打入国际市场，让老外用我们做的东西呢？你们有信心吗？



3

Struts2 国际化自动检测浏览器语言版



前几天发了一篇Struts国际化的博客——《菜鸟学习SSH（二）——Struts2国际化手动切换版》，有网友提了一个意见，见下图：

国际化是米国打开是英文，天朝打开是中文，正确的练习是去浏览器，修改语言设置，设置成中文打开网站显示中文，设置英文打开显示英文，java代码自动获取而不能是你这样new一个Locale，通过超链接触发。这样虽然说能实现国际化的转换，但是还不能算是国际化，顶多可以放置在顶部，让别人手动自己切换喜欢的语言。

支持(0) 反对(0)

于是就有了下面修改的版本：

web.xml

```
<?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">
  <welcome-file-list>
    <welcome-file>index</welcome-file>
  </welcome-file-list>
  <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>
```

struts.xml

```
<?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.devMode" value="true"></constant>
  <constant name="struts.custom.i18n.resources" value="bbs2009"></constant>
  <package name="/" namespace="/" extends="struts-default">

    <action name="*-*" class="com.lsj.action.{1}Action" method="{2}">
      <result>/{1}-{2}.jsp</result>
    </action>
  </package>
</struts>
```

```

</package>

</struts>

```

## 登录页

```

<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";
%>
<%@taglib uri="/struts-tags" prefix="s" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<base href="<%=basePath%>">

<title>登录</title>

<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="This is my page">

</head>

<body>
<form action="Login-login" method="post">
<s:property value="getText('login.username')"/> <input name="username" />
<s:property value="getText('login.password')"/> <input name="password" type="password" />
<input type="submit" value="<s:property value="getText('login.login')"/>" />
</form>

</body>
</html>

```

OK，这样将浏览器的语言设置成中文，那么页面就以中文显示；把浏览器语言设置成英文，页面就以英文显示。这一个版本就是上面那位网友说的那种效果。这个版本实现起来要比上一个要简单，不过之前那种手动修改也有它的作用，大家在上网的时候都见过很多网站上有手动切换显示语言的链接吧（像微软、谷歌的网站上都有的）！这是为了让那些身在国外的人能够自己切换到母语的显示页面而做的。这两种方式各有各的作用。

两种方式各司其职，具体要怎么用还得看你想要实现什么样的需求了。这是两种不同的国际化实现方法，今天把这种自动识别浏览器语言的方法也写出来供大家把玩把玩，希望各位玩的开心。



4

Struts2 拦截器



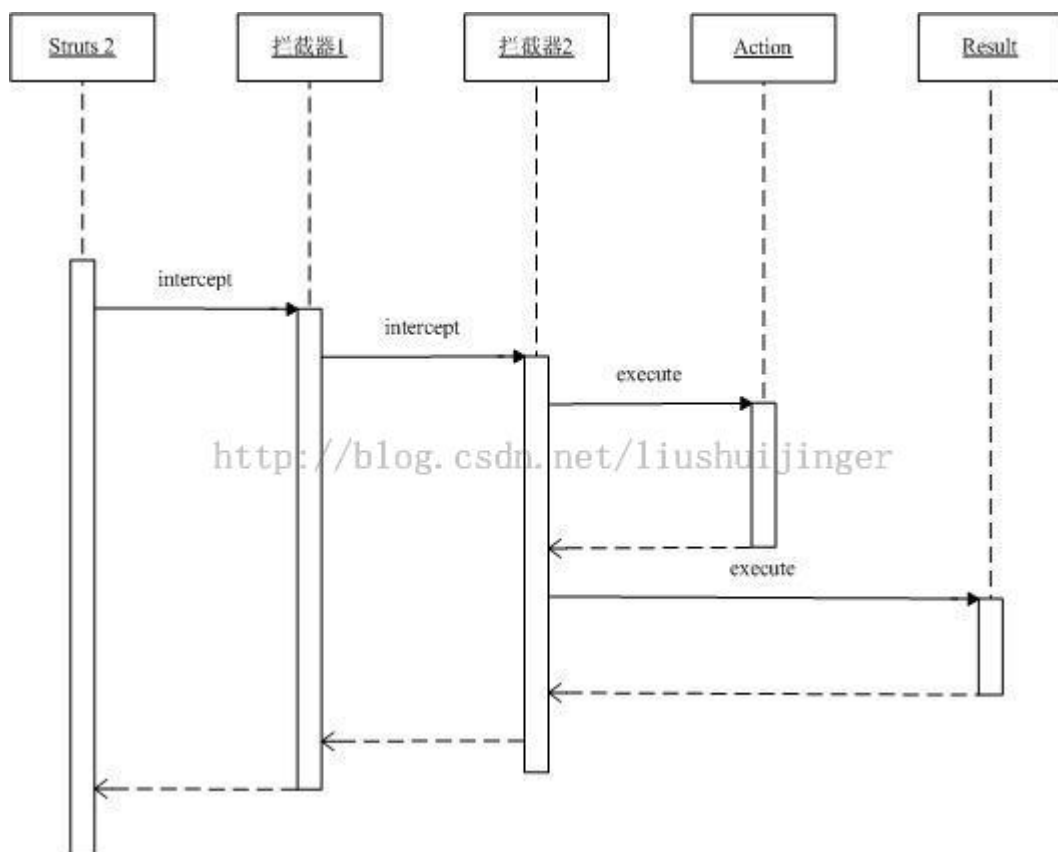


## 什么是拦截器

拦截器（Interceptor）是Struts 2的一个强有力的工具，有许多功能都是构建于它之上，如国际化（前两篇博客介绍过）、转换器，校验等。拦截器是动态拦截Action调用的对象。它提供了一种机制可以使开发者可以定义在一个action执行的前后执行的代码，也可以在一个action执行前阻止其执行。同时也是提供了一种可以提取action中可重用的部分的方式。说到拦截器有一个东西不能落下——拦截器链（Interceptor Chain，在Struts 2中称为拦截器栈Interceptor Stack）。拦截器链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

Struts 2的拦截器实现相对简单。当请求到达Struts 2的ServletDispatcher时，Struts 2会查找配置文件，并根据其配置实例化相对的拦截器对象，然后串成一个列表（list），最后一个一个地调用列表中的拦截器。



Struts 2提供了丰富多样的，功能齐全的拦截器实现。大家可以到struts2-all-2.0.1.jar或struts2-core-2.0.1.jar包的struts-default.xml查看关于默认的拦截器与拦截器链的配置。

相关说明如下：

拦截器	名字	说明
Alias Interceptor	alias	在不同请求之间将请求参数在不同名字件转换，请求内容不变
Chaining Interceptor	chain	让前一个Action的属性可以被后一个Action访问，现在和chain类型的result ( ) 结合使用。
Checkbox Interceptor	checkbox	添加了checkbox自动处理代码，将没有选中的checkbox的内容设定为false，而html默认情况下不提交没有选中的checkbox。
Cookies Interceptor	cookies	使用配置的name,value来是指cookies
Conversion Error Interceptor	conversion Error	将错误从ActionContext中添加到Action的属性字段中。
Create Session Interceptor	createSession	自动的创建HttpSession，用来为需要使用到HttpSession的拦截器服务。
Debugging Interceptor	debugging	提供不同的调试用的页面来展现内部的数据状况。
Execute and Wait Interceptor	execAndWait	在后台执行Action，同时将用户带到一个中间的等待页面。
Exception Interceptor	exception	将异常定位到一个画面
File Upload Interceptor	fileUpload	提供文件上传功能
I18n Interceptor	i18n	记录用户选择的locale
Logger Interceptor	logger	输出Action的名字
Message Store Interceptor	store	存储或者访问实现ValidationAware接口的Action类出现的消息，错误，字段错误等。
Model Driven Interceptor	model-driven	如果一个类实现了ModelDriven，将getModel得到的结果放在Value Stack中。
Scoped Model Driven	scoped-model-driven	如果一个Action实现了ScopedModelDriven，则这个拦截器会从相应的Scope中取出model调用Action的setModel方法将其放入Action内部。
Parameters Interceptor	params	将请求中的参数设置到Action中去。
Prepare Interceptor	prepare	如果Action实现了Preparable，则该拦截器调用Action类的prepare方法。
Scope Interceptor	scope	将Action状态存入session和application的简单方法。
Servlet Config Interceptor	servletConfig	提供访问HttpServletRequest和HttpServletResponse的方法，以Map的方式访问。
Static Parameters Interceptor	staticParams	从struts.xml文件中将中的中的内容设置到对应的Action中。
Roles Interceptor	roles	确定用户是否具有JAAS指定的Role，否则不予执行。

拦截器	名字	说明
Timer Interceptor	timer	输出Action执行的时间
Token Interceptor	token	通过Token来避免双击
Token Session Interceptor	tokenSession	和Token Interceptor一样，不过双击的时候把请求的数据存储在Session中
Validation Interceptor	validation	使用action-validation.xml文件中定义的内容校验提交的数据。
Workflow Interceptor	workflow	调用Action的validate方法，一旦有错误返回，重新定位到INPUT画面
Parameter Filter Interceptor	N/A	从参数列表中删除不必要的参数
Profiling Interceptor	profiling	通过参数激活profile

#### 配置拦截器

#### 定义拦截器

```
<package name="MyInterceptor" extends="struts-default" namespace="/">
  <interceptors>
    <!-- 配置拦截器 -->
    <interceptor name="拦截器名" class="拦截器实现类完整路径"/>
    <!-- 配置拦截器栈 -->
    <interceptor-stack name="拦截器栈名">
      <interceptor-ref name="拦截器一"/>
      <interceptor-ref name="拦截器二"/>
      |.....
    </interceptor-stack>
  </interceptors>
</package>
```

#### 将拦截器配置到action

```
<action name="login" class="com.action.LoginAction">
  <result name="success">/success.jsp</result>
  <result name="error">/error.jsp</result>
  <!-- 引用拦截器,一般配置在result后面 -->
  <interceptor-ref name="拦截器名或拦截器栈名"/>
  <!-- 引用Struts默认拦截器 -->
  <interceptor-ref name="defaultStack"/>
</action>
```

需要注意的是，必须要引用Struts默认的拦截器，否则会报错。

实现自定义拦截器：

通过实现接口`com.opensymphony.xwork2.interceptor.Interceptor`可自定义拦截器。该接口提供了三个方法：

- 1) `void init()`; 在该拦截器被初始化之后，在该拦截器执行拦截之前，系统回调该方法。对于每个拦截器而言，此方法只执行一次。
- 2) `void destroy()`;该方法跟`init()`方法对应。在拦截器实例被销毁之前，系统将回调该方法。
- 3) `String intercept(ActionInvocation invocation) throws Exception`; 该方法是用户需要实现的拦截动作。该方法会返回一个字符串作为逻辑视图。

除此之外，通过继承类`com.opensymphony.xwork2.interceptor.AbstractInterceptor`等方法也可自定义拦截器，这里不过多介绍。

#### 拦截器实例

自定义拦截器`MyInterceptor`，实现`Interceptor`接口

```
package com.interceptor;

import java.util.Map;

import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.Interceptor;
import com.sun.org.apache.xml.internal.security.keys.content.RetrievalMethod;

public class MyInterceptor implements Interceptor {

    @Override
    public void destroy() {
        System.out.println("-----destroy()-----");
    }

    @Override
    public void init() {
        System.out.println("-----init()-----");
    }

    @Override
    public String intercept(ActionInvocation invocation) throws Exception {
        System.out.println("-----intercept()-----");
        Map session = invocation.getInvocationContext().getSession();
    }
}
```

```

    if (session.get("username") != null) {
        return invocation.invoke();
    } else {
        return "login";
    }
}
}
}

```

### 拦截器的配置

```

<?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.action.extension" value=","/>
    <package name="testLogin" namespace="/" extends="struts-default" >
        <interceptors>

            <interceptor name="MyInterceptor" class="com.interceptor.MyInterceptor"/>

            <interceptor-stack name="defaultInterceptorStack">
                <interceptor-ref name="MyInterceptor"/>
                <interceptor-ref name="defaultStack"/>
            </interceptor-stack>
        </interceptors>

        <!--配置默认拦截器，所有该包内的action如果没有单独配置拦截器，则默认执行默认拦截器-->
        <default-interceptor-ref name="defaultInterceptorStack"/>

        <action name="login" class="com.action.LoginAction">
            <result name="success" type="redirect"/>success.jsp</result>
            <result name="error" type="redirect"/>error.jsp</result>
            <result name="login">login.jsp</result>
        </action>
    </package>
</struts>

```

进入系统的时候，拦截器检查是否登录，如未登录，转到登录页；如已登录，转到成功页。

拦截器是Struts 2比较重要的一个功能。通过正确地使用拦截器，我们可以编写可复用性很高的代码。



5

Struts2 上传文件



上传文件在一个系统当中是一个很常用的功能，也是一个比较重要的功能。今天我们就一起来学习一下Struts2如何上传文件。今天讲的上传文件的方式有三种：

- 1以字节为单位传输文件；
- 2Struts2封装的一种方式；
- 3以字符的方式传输文件。

其实这三种方式都差不多，都是将文件先从客户端一临时文件的形式，传输到服务器的临时文件夹下，然后在将该临时文件复制到我们要上传的目录。另外，有一个需要注意，就是上传过程中产生的这些临时文件，Struts2不会自动清理，所以我们需要手动清理临时文件，这一个下面的代码中有提到。

用Action来完成我们上传的核心功能：

```
package com.action;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.Map;

import org.apache.struts2.ServletActionContext;

import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;

public class UploadAction extends ActionSupport {

    private File upload;
    private String uploadFileName;
    private String uploadContentType;

    private long maximumSize;
    private String allowedTypes;

    public File getUpload() {
        return upload;
    }
}
```

```

public void setUpload(File upload) {
    this.upload = upload;
}
public String getUploadFileName() {
    return uploadFileName;
}
public void setUploadFileName(String uploadFileName) {
    this.uploadFileName = uploadFileName;
}

public String getUploadContentType() {
    return uploadContentType;
}
public void setUploadContentType(String uploadContentType) {
    this.uploadContentType = uploadContentType;
}
public long getMaximumSize() {
    return maximumSize;
}
public void setMaximumSize(long maximumSize) {
    this.maximumSize = maximumSize;
}
public String getAllowedTypes() {
    return allowedTypes;
}
public void setAllowedTypes(String allowedTypes) {
    this.allowedTypes = allowedTypes;
}
@Override
public String execute() throws Exception {

    File uploadFile = new File(ServletActionContext.getServletContext().getRealPath("upload"));
    if(!uploadFile.exists()) {
        uploadFile.mkdir();
    }

    //验证文件大小及格式
    if (maximumSize < upload.length()) {
        return "error";
    }

    boolean flag =false;
    String[] allowedTypesStr = allowedTypes.split(",");
    for (int i = 0; i < allowedTypesStr.length; i++) {
        if (uploadContentType.equals(allowedTypesStr[i])) {

```



```

        flag = true;
    }
}
if (flag == false) {
    Map request = (Map) ActionContext.getContext().get("request");
    request.put("errorMessage", "文件类型不合法! ");

    System.out.println(request.toString());
    return "error";
}

```

//第一种上传方式

```

// FileInputStream input = new FileInputStream(upload);
// FileOutputStream out = new FileOutputStream(uploadFile + "/" + uploadFileName);
// try {
//     byte[] b = new byte[1024];
//     int i = 0;
//     while ((i = input.read(b)) > 0) {
//         out.write(b, 0, i);
//     }
// } catch (Exception e) {
//     e.printStackTrace();
// } finally {
//     //关闭输入、输出流
//     input.close();
//     out.close();
//     //删除临时文件
//     upload.delete();
// }

```

//第二种上传方式

```

// FileUtils.copyFile(upload, new File(uploadFile + "/" + uploadFileName));
// //删除临时文件
// upload.delete();

```

//第三种上传方式

```

BufferedReader bReader = new BufferedReader(new InputStreamReader(new FileInputStream(upload)));
BufferedWriter bWriter = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(uploadFile + "/" + uploadFileName)));
try {
    char[] c = new char[1024];
    int i = 0;
    while ((i = bReader.read(c)) > 0) {
        bWriter.write(c, 0, i);
    }
} catch (Exception e) {

```

```

        e.printStackTrace();
    } finally {
        bReader.close();
        bWriter.close();
        //删除临时文件
        upload.delete();
    }

    return "success";
}
}

```

在Struts.xml文件中配置我们的Action：

```

<?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.action.extension" value=","/>
    <!-- 上传文件最大限制（如果为多文件上传，则为多个文件的总大小） -->
    <constant name="struts.multipart.maxSize" value="40000000"/>
    <!-- 存放上传文件的临时目录 -->
    <constant name="struts.multipart.saveDir" value="D:\\temp"/>

    <package name="upload" namespace="/file" extends="struts-default">
        <action name="upLoad" class="com.action.UploadAction">
            <result name="success">/success.jsp</result>
            <result name="error" >/error.jsp</result>
            <result name = "input" type ="redirect">/index.jsp</result>
            <param name="maximumSize ">1000000</param>
            <param name="allowedTypes">application/msword,application/vnd.openxmlformats-officedocument.wordprocessin

            <!--用struts拦截器限制上传文件大小及类型
            <interceptor-ref name="fileUpload">
                单个文件的大小
                <param name="maximumSize ">1000000</param>
                <param name="allowedTypes">application/msword,application/vnd.openxmlformats-officedocument.wordprocessin
            </interceptor-ref>
            <interceptor-ref name="defaultStack"></interceptor-ref>
            -->
        </action>
    </package>
</struts>

```

```
</package>

</struts>
```

文件上传的页面：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>上传文件</title>
</head>
<body>
    <!-- enctype="multipart/form-data"不对字符编码。在使用包含文件上传控件的表单时，必须使用该值。 -->
    <form action="file/upLoad" method="post" enctype="multipart/form-data">
        <input name="upload" type="file">
        <input name="btnUpload" type="submit" value="上传">
    </form>
</body>
</html>
```

上传文件之前，通常要判断一下文件的大小及类型，上面有两种方式来验证，一种是在Action里验证，一种是通过Struts2的拦截器验证。两者选其一，选择一个把另一个注释掉即可。

另外，还有一个需要注意：上传文件页面中form的enctype属性值，一定要设置成"multipart/form-data"，否则就会出错。

OK，今天上传文件就向大家介绍到这里，我们下篇博客再见！



Spring 事务管理



Spring是SSH中的管理员，负责管理其它框架，协调各个部分的工作。今天一起学习一下Spring的事务管理。Spring配置文件中关于事务配置总是由三个组成部分，分别是DataSource、TransactionManager和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。DataSource、TransactionManager这两部分只是会根据数据访问方式有所变化，比如使用Hibernate进行数据访问时，DataSource实际为SessionFactory，TransactionManager的实现为HibernateTransactionManager。下面一起来看看三种声明式事务的具体配置：

### 公共配置

```
<!-- 配置sessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="configLocation">
        <value>classpath:config/hibernate.cfg.xml</value>
    </property>
    <property name="packagesToScan">
        <list>
            <value>com.entity</value>
        </list>
    </property>
</bean>

<!-- 配置事务管理器（声明式的事务） -->
<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>

<!-- 配置DAO -->
<bean id="userDao" class="com.dao.UserDaoImpl">
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
```

### 第一种，使用tx标签方式

```
<!-- 第一种配置事务的方式，tx-->
<tx:advice id="txadvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED" rollback-for="Exception" />
        <tx:method name="modify*" propagation="REQUIRED" rollback-for="Exception" />
        <tx:method name="del*" propagation="REQUIRED" rollback-for="Exception"/>
        <tx:method name="*" propagation="REQUIRED" read-only="true"/>
    </tx:attributes>
</tx:advice>

<aop:config>
```

```

<aop:pointcut id="daoMethod" expression="execution(* com.dao.*(..))"/>
<aop:advisor pointcut-ref="daoMethod" advice-ref="txadvice"/>
</aop:config>

```

expression="execution(\* com.dao.\*(..))" 其中第一个代表返回值，第二代表dao下子包，第三个\*代表方法名，“（..）”代表方法参数。

## 第二种，使用代理方式

```

<!-- 第二种配置事务的方式，代理-->
<bean id="transactionProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean" abstract="true">
<property name="transactionManager" ref="transactionManager"></property>
<property name="transactionAttributes">
<props>
<prop key="add*">PROPAGATION_REQUIRED, -Exception</prop>
<prop key="modify*">PROPAGATION_REQUIRED, -Exception</prop>
<prop key="del*">PROPAGATION_REQUIRED, -Exception</prop>
<prop key="*">PROPAGATION_REQUIRED, readOnly</prop>
</props>
</property>
</bean>
<bean id="userDao" parent="transactionProxy">
<property name="target">
<!-- 用bean代替ref的方式-->
<bean class="com.dao.UserDaoImpl">
<property name="sessionFactory" ref="sessionFactory"></property>
</bean>
</property>
</bean>

```

将transactionProxy的abstract属性设置为"true"，然后将具体的Dao的parent属性设置为"transactionProxy"，可以精简代码。

## 第三种，使用拦截器

```

<!-- 第三种配置事务的方式，拦截器 (不常用)-->
<bean id="transactionInterceptor" class="org.springframework.transaction.interceptor.TransactionInterceptor">
<property name="transactionManager" ref="transactionManager"></property>
<property name="transactionAttributes">
<props>
<prop key="add*">PROPAGATION_REQUIRED, -Exception</prop>
<prop key="modify*">PROPAGATION_REQUIRED, -Exception</prop>
<prop key="del*">PROPAGATION_REQUIRED, -Exception</prop>
<prop key="*">PROPAGATION_REQUIRED, readOnly</prop>
</props>

```

```

    </property>
</bean>
<bean id="proxyFactory" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
        <list>
            <value>transactionInterceptor</value>
        </list>
    </property>
    <property name="beanNames">
        <list>
            <value>*Dao</value>
        </list>
    </property>
</bean>

```

Spring事务类型详解：

- PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
- PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。
- PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。
- PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。
- PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。
- PROPAGATION\_NESTED--如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。

采用注解的方式，需要注意的是，使用注解的方式需要在Spring的配置文件中加入一句话：< context:annotation-config />，其作用是开启注解的方式。具体配置如下：

```

<!--开启注解方式-->
<context:annotation-config />

<!-- 配置sessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="configLocation">
        <value>classpath:config/hibernate.cfg.xml</value>
    </property>
    <property name="packagesToScan">
        <list>

```

```

        <value>com.entity</value>
    </list>
</property>
</bean>

<!-- 配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>

<!-- 第四种配置事务的方式，注解 -->
<tx:annotation-driven transaction-manager="transactionManager"/>

```

注解文件：

```

package com.dao;

import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.entity.User;

@Transactional
public class UserDaoImpl_BAK extends HibernateTemplate {

    @Transactional(propagation=Propagation.REQUIRED,rollbackForClassName="Exception")
    public void addUser(User user) throws Exception {
        this.save(user);
    }

    @Transactional(propagation=Propagation.REQUIRED,rollbackForClassName="Exception")
    public void modifyUser(User user) {
        this.update(user);
    }

    @Transactional(propagation=Propagation.REQUIRED,rollbackForClassName="Exception")
    public void delUser(String username) {
        this.delete(this.load(User.class, username));
    }

    @Transactional(readOnly=true)
    public void selectUser() {

    }
}

```

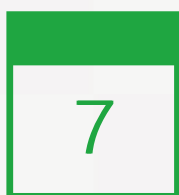


```
}
```

类头的@Transactional为默认事务配置，如方法没有自己的事务类型，则按默认事务，如有自己的配置，则按自己的配置。

以上四种配置方式最常用的还是第一、二种，第三种是比较老旧的方式，而注解的方式不太适合比较大的项目，用于简单的小项目还是很好的，其特点就是简单明了。每种方法都有每种方法的特点跟适用的环境，没有绝对的好与坏，只不过前两种在实际的工作当中用的更多一些。

好了，今天就到这里吧，看了这么长的文章，相信大家都累了，也不想再听我在这里废话了，哈哈。欢迎交流，欢迎拍砖。



## Spring jar包详解



Struts、Hibernate、Spring这类的框架给我们开发带来非常大的好处，让我们更加快速、有效的开发。所以我们在开发中通常都会用到各种框架，每个框架都有很多jar包，每个jar都有各自不同的功能。开发不同的功能用到的jar也不尽相同，所以当我们用到相关框架的时候，并不是把它所有的jar都引入系统。那么怎么确定自己将会用到框架中的哪些jar包呢？这就需要我们知道框架中每个jar都是干什么的。今天我先向大家介绍一下Spring的相关jar包。Follow me!

spring.jar中包含除了 spring-mock.jar里所包含的内容外其它所有jar包的内容（因为只有在开发环境下才会用到spring-mock.jar来进行辅助测试，正式应用系统中是用不得这些类的。），除了spring.jar文件，Spring还包括有其它13个独立的jar包，各自包含着对应的Spring组件，我们可以根据自己的需要来选择组合jar包，接下来我将为大家一一道来。

(1) spring-core.jar      这个jar文件包含Spring框架基本的核心工具类，Spring其它组件都要使用到这个包里的类，是其它组件的基本核心，当然你也可以在自己的应用系统中使用这些工具类。

(2) spring-beans.jar      这个jar文件是所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行Inversion of Control / Dependency Injection ( IoC/DI ) 操作相关的所有类。如果应用只需基本的IoC/DI支持，引入spring-core.jar及spring-beans.jar文件就可以了。

(3) spring-aop.jar      这个jar文件包含在应用中使用Spring的 AOP特性时所需的类。使用基于AOP的Spring特性，如声明型事务管理（Declarative Transaction Management），也要在应用里包含这个jar包。

(4) spring-context.jar      这个jar文件为Spring核心提供了大量扩展。可以找到使用Spring ApplicationContext特性时所需的全部类，JNDI所需的全部类，UI方面的用来与模板（Templating）引擎如 Velocity、FreeMarker、JasperReports集成的类，以及校验Validation方面的相关类。

(5) spring-dao.jar      这个jar文件包含Spring DAO、Spring Transaction进行数据访问的所有类。为了使用声明型事务支持，还需在自己的应用里包含spring-aop.jar。

(6) spring-hibernate.jar      由名字就可以知道它的用途，这个jar文件包含Spring对Hibernate 2及Hibernate 3进行封装的所有类。

(7) spring-jdbc.jar      这个jar文件包含对Spring对 JDBC数据访问进行封装的所有类。

(8) spring-orm.jar      这个jar文件包含Spring对 DAO特性集进行了扩展，使其支持 iBATIS、JDO、OJB、TopLink，因为Hibernate已经独立成包了，现在不包含在这个包里了。这个jar文件里大部分的类都要依赖 spring-dao.jar里的类，用这个包时你需要同时包含spring-dao.jar包。

(9) spring-remoting.jar      这个jar文件包含支持EJB、JMS、远程调用Remoting（RMI、Hessian、Burlap、Http Invoker、JAX-RPC）方面的类。

(10) spring-support.jar 这个jar文件包含支持缓存Cache（ehcache）、JCA、JMX、邮件服务（Java Mail、COS Mail）、任务计划Scheduling（Timer、Quartz）方面的类。

(11) spring-web.jar 这个jar文件包含Web应用开发时，用到Spring框架时所需的核心类，包括自动载入WebApplicationContext特性的类、Struts与JSF集成类、文件上传的支持类、Filter类和大量工具辅助类。

(12) spring-webmvc.jar 这个jar文件包含Spring MVC框架相关的所有类。包含国际化、标签、Theme、视图展现的FreeMarker、JasperReports、Tiles、Velocity、XSLT相关类。当然，如果你的应用使用了独立的MVC框架，则无需这个JAR文件里的任何类。

(13) spring-mock.jar 这个jar文件包含Spring一整套mock类来辅助应用的测试。Spring测试套件使用了其中大量mock类，这样测试就更加简单。模拟HttpServletRequest和HttpServletResponse类在Web应用单元测试是很方便的。

如何选择jar包，除了要了解每个jar的用途以外，还要了解jar包与jar包之间的依赖关系。有些jar包是其它jar包的基础，而有些jar则需要依赖于其它jar包才能工作。那么Spring里jar包是怎样一个依赖关系呢？听我娓娓道来。

Spring包依赖说明：

- 1 spring-core.jar依赖commons-collections.jar。
- 2 spring-beans.jar依赖spring-core.jar, cglib-nodep-2.1\_3.jar
- 3 spring-aop.jar依赖spring-core.jar, spring-beans.jar, cglib-nodep-2.1\_3.jar, aopalliance.jar
- 4 spring-context.jar依赖spring-core.jar, spring-beans.jar, spring-aop.jar, commons-collections.jar, aopalliance.jar
- 5 spring-dao.jar依赖spring-core.jar, spring-beans.jar, spring-aop.jar, spring-context.jar
- 6 spring-jdbc.jar依赖spring-core.jar, spring-beans.jar, spring-dao.jar
- 7 spring-web.jar依赖spring-core.jar, spring-beans.jar, spring-context.jar
- 8 spring-webmvc.jar依赖spring-core.jar/spring-beans.jar/spring-context.jar/spring-web.jar
- 9 spring-hibernate.jar依赖spring-core.jar, spring-beans.jar, spring-aop.jar, spring-dao.jar, spring-jdbc.jar, spring-orm.jar, spring-web.jar, spring-webmvc.jar
- 10 spring-orm.jar依赖spring-core.jar, spring-beans.jar, spring-aop.jar, spring-dao.jar, spring-jdbc.jar, spring-web.jar, spring-webmvc.jar
- 11 spring-remoting.jar依赖spring-core.jar, spring-beans.jar, spring-aop.jar, spring-dao.jar, spring-context.jar, spring-web.jar, spring-webmvc.jar

- 12 spring-support.jar依赖spring-core.jar, spring-beans.jar, spring-aop.jar, spring-dao.jar, spring-context.jar, spring-jdbc.jar
- 13 spring-mock.jar依赖spring-core.jar, spring-beans.jar, spring-dao.jar, spring-context.jar, spring-jdbc.jar

spring-core.jar是spring的核心包，其它所有jar包都依赖于它。

PS: Eclipse有个插件叫 ClassPath Checker可以帮你找找所依赖的类。在线安装地址: <http://classpathchecker.free.fr/update-site>



8

## Hibernate 对象的三种状态



前面写了几篇关于SSH的博客，但不是Struts就是Spring，Hibernate还从来没写过呢。说好是SSH的，怎么可以光写那两个，而不写Hibernate呢对吧。今天就先说说Hibernate对象的三种状态，Hibernate对象有三种状态，分别是：临时态(Transient)、持久态(Persistent)、游离态(Detached)。

**临时状态:**是指从对象通过new语句创建到被持久化之前的状态，此时对象不在Session的缓存中。

处在此状态的对象具备以下特点：

- 1不在Session缓存中，不与任何Session实例相关联。
- 2在数据库中没有与之对应的记录。

通常在下列情况下对象会进入临时状态：

- 1通过new语句创建新对象。
- 2执行delete()方法，对于游离状态的对象，delete()方法会将其与数据库中对应的记录删除；而对于持久化状态的对象，delete()方法会将其与数据库中对应的记录删除，并将其在Session缓存中删除。

例如：Object object = new Object(); 此时object为临时状态，数据库中没有对应的数据，Session缓存中也没有相关联的实例。

**持久化状态:**是指对象被持久化到Session对象被销毁之前的状态，此时对象在Session的缓存中。

处在此状态的对象具备以下特点：

- 1在Session缓存中，与Session实例相关联。
- 2在数据库中有与之对应的记录。
- 3Session在清理缓存的时候，会根据持久化对象的属性变化更新数据库。

通常在下列情况下对象会进入临时状态：

- 1执行save()或saveOrUpdate()方法，使临时对象转变为持久化对象。
- 2执行update()或saveOrUpdate()方法，使游离对象转变为持久化对象。
- 3执行load()或get()方法，返回的对象都是持久化对象。
- 4执行find()方法，返回List集合中存放的都是持久化对象。
- 5在允许级联保存的情况下，Session在清理缓存时会把与持久化对象关联的临时对象转变为持久化对象。

例如：Session session = factory.openSession(); object.setName("持久化对象"); session.save(object); 此时object对象为持久化对象，Session缓存中有相关联的实例，数据库中有相应的记录。

**游离状态：**是指从持久化对象的Session对象被销毁到该对象消失之前的状态，此时对象不在Session的缓存中。

处在此状态的对象具备以下特点：

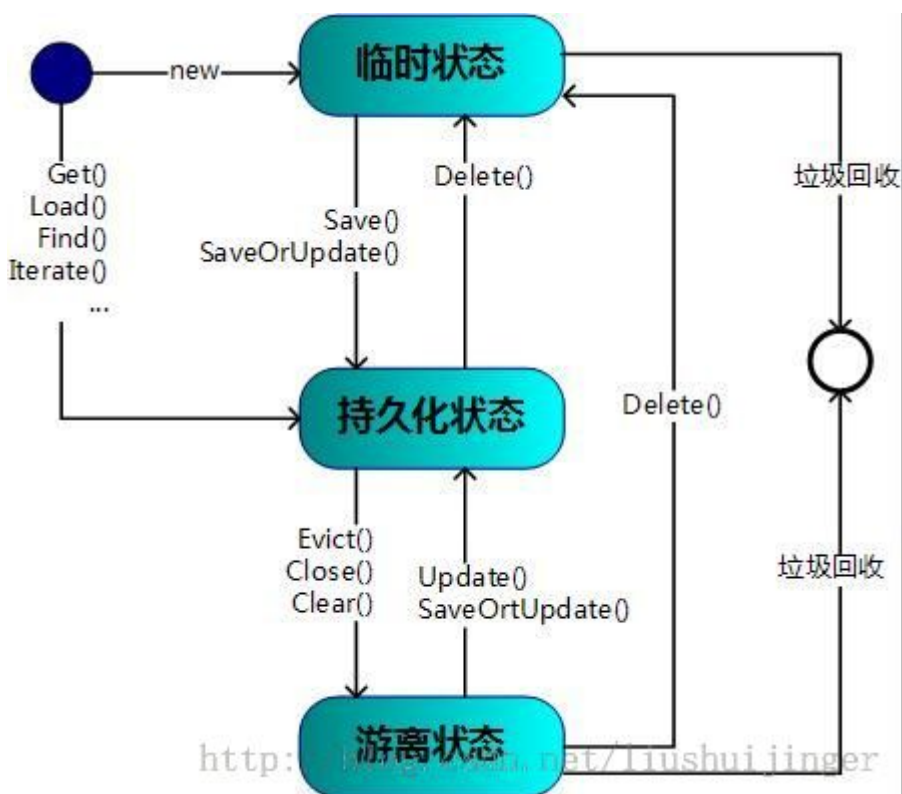
- 1不在Session缓存中，不与任何Session实例相关联。
- 2在数据库中有与之对应的记录（前提是没有其他Session实例删除该条记录）。

通常在下列情况下对象会进入临时状态：

- 1执行close()方法，将Session缓存清空，缓存中的所有持久化对象将转变成游离对象。
- 2执行evict()方法，能从缓存中删除一个持久化对象，使之转变成游离对象。

例如：session.close(); 此时上文的object对象为游离对象，Session缓存中没有有相关联的实例，数据库中有相应的记录。

三种状态之间的转换过程



对上图的解析：

- 1当一个对象被new了以后此对象处于临时态（Transient）。
- 2然后对此对象执行session的save() 或者saveOrUpdate()方法后，此对象被放入session的一级缓存进入持久态。



- 3当再对此对象执行evict()/close()/clear()的操作后此对象进入游离态（ Detached ）。
- 4游离态（ Detached ）和临时态（ Transient ）的对象由于没有被session管理会在适当的时机被java的垃圾回收站（ garbage ）回收。
- 5执行session的get()/load()/find()/iternte()等方法从数据库里查询的到的对象,处于持久态（ Persistent ）。
- 6当对数据库中的纪录进行update()/saveOrUpdate()/lock()操作后游离态的对象就过渡到持久态。
- 7处于持久态（ Persistent ）与游离态（ Detached ）的对象在数据库中都有对应的记录。
- 8临时态（ Transient ）与游离态(Detached)的对象都可以被回收可是临时态的对象在数据库中没有对应的纪录,而游离态的对象在数据库中有对用的纪录。

三种状态在程序的转换过程：

代码	对象的生命周期状态	对象的状态
Object object = new Object();	开始生命周期	开始生命周期
Session session = factory.openSession(); Transaction tx = session.beginTransaction();	在生命周期中	临时状态
session.save(object);	在生命周期中	转变为持久化状态
tx.commit();	在生命周期中	持久化状态
session.close();	在生命周期中	转变为游离状态
System.out.println(object.getName());	在生命周期中	游离状态
object = null;	结束生命周期	结束生命周期

三种状态里面，只有持久化状态在Session缓存中有相关联的实例，临时状态跟游离状态都没有。临时状态数据库里没有对应的记录，其他两种可能又记录，也可能没有记录。好了，Hibernate对象的三种状态就说到这里，接下来可能跟大家说说Hibernate的Session，期待不？



9

## Hibernate——Session之save()方法



Session的save()方法用来将一个临时对象转变为持久化对象，也就是将一个新的实体保存到数据库中。通过save()将持久化对象保存到数据库需要经过以下步骤：

- 1系统根据指定的ID生成策略，为临时对象生成一个唯一的OID；
- 2将临时对象加载到缓存中，使之变成持久化对象；
- 3提交事务时，清理缓存，利用持久化对象包含的信息生成insert语句，将持久化对象保存到数据库。

OK，下面来看一个实例：

```
//创建SessionFactory
Configuration config = new Configuration();
ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();
SessionFactory sessionFactory = config.buildSessionFactory(sr);

//打开Session
Session session = sessionFactory.openSession();

//开启事务
Transaction tx = session.beginTransaction();

//创建临时对象并赋值
User user = new User();
user.setId("1");
user.setUsername("lsj");
session.save(user);

//提交事务
tx.commit();

//关闭Session
session.close();
```

PS：

- 1从Hibernate 4之后，Configuration类中，原先常用的，获取SessionFactory的方法buildSessionFactory()被标记为过时，官方建议使用buildSessionFactory(ServiceRegistry sr)这个方法来获取SessionFactory。
- 2如果映射文件中为对象的ID指定了生成策略，那么在程序中为其ID赋值是无效的。会被系统自动生成的值覆盖，例如：

映射文件指定ID生成策略由Hibernate控制自增：

```
<id name="id" type="string">
    <generator class="increment">
</id>
```

然后程序中进行如下操作：

```
user.setId("1");
System.out.println("手动赋值为: "+user.getId());
session.save(user);
System.out.println("存入数据库的值为: "+user.getId());
```

这样，最终存入数据库的ID是Hibernate自动生成的自增ID，而不是我们手动给的值。另外，执行save()方法时并不会将对象存入数据库，在提交事务时，对象才被真正的保存到数据库中。save()方法是将对象保存到Session的缓存中，提交事务时，Hibernate会生成相应的insert语句，将对象保存到数据库。

再跟大家说一点需要注意的地方，Hibernate在提交事务的时候，会将之前对对象做的操作一并提交。就算是在save()方法之后做的操作也一样。什么意思呢？我们看下面的例子：

```
user.setUserName("old");
session.save(user);
user.setUserName("new");
tx.commit();
```

执行以上操作，最终user.getUserName();得到的值将会是“new”，而不会是“old”，因为Hibernate在提交事务的时候把对user对象的所有操作都提交了。

OK，今天就到这里，save()方法虽然看起来是一个很简单的方法，但是需要注意这些细小的地方，否则很容易出一些问题。一句话：学习框架的原理很重要，要想理解框架的原理，最直接的办法就是看源码。

每天学一点，每天进步一点。祝大家每天进步。



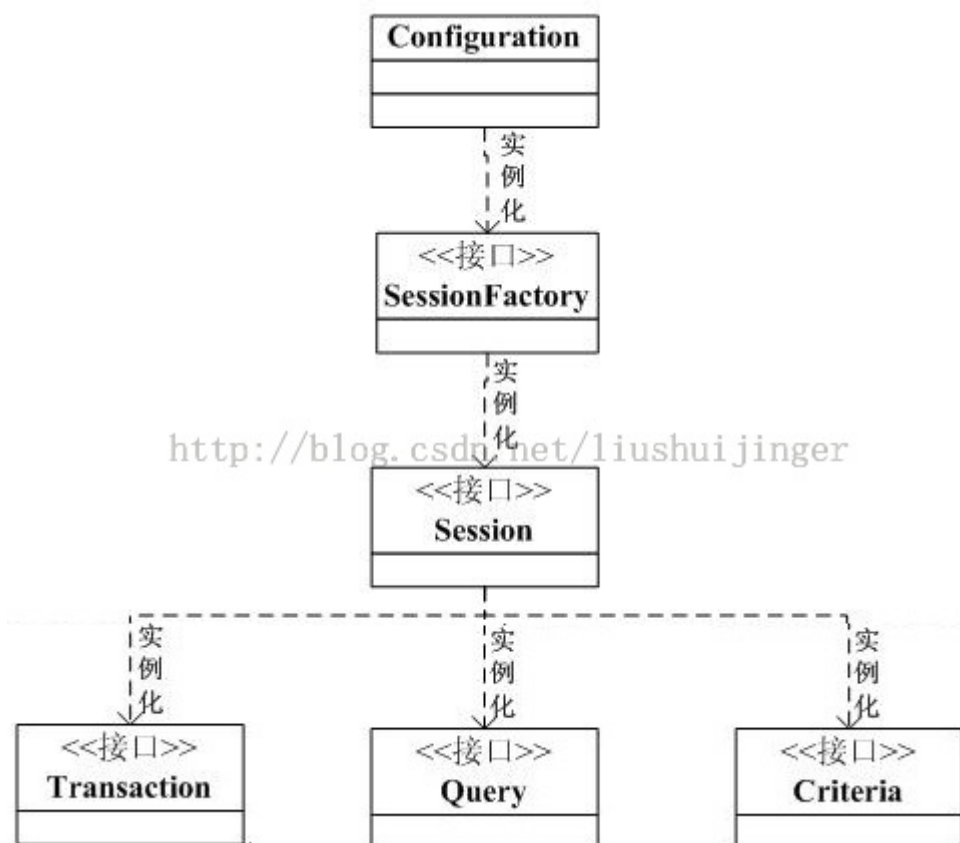
10

## Hibernate 核心接口



在使用Hibernate的时候，我们通常都会用的Configuration、SessionFactory、Session、Transaction、Query和Criteria等接口。通过这些接口可以，不仅可以存储与取出持久化对象，还可以对事务进行管理。下面对着几个接口一一介绍：

几个接口之间的层次关系如下图：



Configuration：

Configuration是Hibernate的入口，负责将配置文件信息加载到内存，并创建一个SessionFactory对象，把读入的配置信息加载到SessionFactory对象的内存里。

特点：

Configuration对象的作用是除了有读取配置文件的功能，还能创建SessionFactory对象。Configuration对象只存在于系统的初始化阶段，然后所有的持久化操作都能通过这个SessionFactory实例来进行。Configuration对象只有在Hibernate 进行初始化的时候才需要创建，当使用Configuration对象的实例创建了SessionFactory对象的实例后，其配置信息已经绑定在他返回的SessionFactory对象实例中。因此，一般情况下，得到SessionFactory对象后，Configuration对象的使命就结束了。

用法：

属性文件 (hibernate.properties) : `Configuration cfg = new Configuration();` Xml文件 (hibernate.cfg.xml) : `Configuration cfg = new Configuration().configure();`

### SessionFactory:

SessionFactory负责创建Session实例, 每个SessionFactory实例对应一个数据库。SessionFactory是重量级的, 占用缓存较大, 所以每个数据库只需创建一个SessionFactory实例, 当需要操作多个数据库时, 再为每一个数据库指定一个SessionFactory实例。

### 特点:

- 1线程安全, 同一个实例可以被应用的多个线程共享
- 2重量级, 不能随意创建和销毁他的实例, 一个数据库, 只需要创建一个SessionFactory的实例。
- 3以后对Configuration对象势力作出的修改都不会影响已经创建好的SessionFactory实例, 如果需要使用基于改动后的Configuration实例的SessionFactory, 需要从Configuration对象中重新创建新的SessionFactory实例。

### 用法:

```
Configuration config = new Configuration();
ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();
SessionFactory sessionFactory = config.buildSessionFactory(sr);
```

### Session:

Session是Hibernate持久化操作的基础, 负责管理所有与持久化有关的操作, Session与SessionFactory不同, 它是轻量级的, 也是非线程安全的。创建和销毁不会消耗太多资源, 可以为每一个请求分配一个Session。

### 特点:

- 1不是线程安全的, 应该避免多个线程共享同一个Session实例。
- 2Session实例是轻量级的。
- 3Session对象内部有一个缓存, 被称为Hibernate第一缓存, 他存放被当前工作单元中加载的对象, 每个Session实例都有自己的缓存。

### 用法:

```
Session session = sessionFactory.openSession();
常用方法: session.save(); session.update(); session.saveOrUpdate(); session.delete();
```

### Transaction:

Transaction负责Hibernate的数据库事务，其实Hibernate本身并不具备事务管理的能力，只是对底层事务接口进行了封装，这样有利于在不同环境或容器中移植，也可以直接访问底层的事务接口。

用法：

```
Transaction tx = session.beginTransaction();
```

Query和Criteria：

Query和Criteria负责Hibernate的查询操作。Query实例封装了一个HQL（Hibernate Query Language）查询语句，HQL与SQL有些类似，只是HQL是面向对象的，它操作的是持久化类的类名和属性名，而SQL操作的是表名和字段名。Criteria实例完全封装了字符串形式的查询语句，它比Query实例更加面向对象，更适合执行动态查询。

本文只是对这几个接口的一个简单介绍，它们还有很多需要我们去学习跟了解的地方，这几个接口有一个共同的目的，就是让我们用更加面向对象的方式去编程。





11

# Hibernate之SchemaExport+配置文件生成表结构



今天说点基础的东西，说说如何通过SchemaExport跟Hibernate的配置文件生成表结构。其实方法非常简单，只需要两个配置文件，两个Java类就可以完成。

首先要生成表，得先有实体类，以Person.java为例：

```
/**
 *
 * @author Administrator
 * @hibernate.class table="T_Person"
 */
public class Person {

    /**
     * @hibernate.id
     * generator-class="native"
     */
    private int id;

    /**
     * @hibernate.property
     */
    private String name;

    /**
     * @hibernate.property
     */
    private String sex;

    /**
     * @hibernate.property
     */
    private String address;

    /**
     * @hibernate.property
     */
    private String duty;

    /**
     * @hibernate.property
     */
    private String phone;

    /**
     * @hibernate.property
```

```

*/
private String description;

/**
 * @hibernate.many-to-one
 */
private Orgnization org;

public String getAddress() {
    return address;
}
public void setAddress(String address) {
    this.address = address;
}
public String getDescription() {
    return description;
}
public void setDescription(String description) {
    this.description = description;
}
public String getDuty() {
    return duty;
}
public void setDuty(String duty) {
    this.duty = duty;
}
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 String getPhone() {
    return phone;
}
public void setPhone(String phone) {
    this.phone = phone;
}
public String getSex() {

```

```

        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
    public Orgnization getOrg() {
        return org;
    }
    public void setOrg(Orgnization org) {
        this.org = org;
    }
}

```

接下来就是Person类对应的配置文件Person.hbm.xml，配置如下：

```

<hibernate-mapping>
<class table="T_Person" name="com.tgb.model.Person">
    <id name="id">
        <generator class="native"/>
    </id>
    <property name="name"/>
    <property name="sex"/>
    <property name="address"/>
    <property name="duty"/>
    <property name="phone"/>
    <property name="description"/>
    <many-to-one name="org"></many-to-one>
</class>
</hibernate-mapping>

```

还有包含Person.hbm.xml相关信息的Hibernate默认配置文件，hibernate.cfg.xml：

```

<hibernate-configuration>
<session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://127.0.0.1/test</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">123456</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <mapping resource="com/tgb/model/Person.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

万事俱备只欠东风，最后我们还需要一个根据上述内容生成数据表的小工具，即ExportDB.Java:

```
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class ExportDB {

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

        // 默认读取hibernate.cfg.xml文件
        Configuration cfg = new Configuration().configure();

        // 生成并输出sql到文件（当前目录）和数据库
        SchemaExport export = new SchemaExport(cfg);

        // 创建表结构，第一个true 表示在控制台打印sql语句，第二个true 表示导入sql语句到数据库
        export.create(true, true);
    }
}
```

完成以上步骤以后，只需要执行ExportDB类即可，当然前提是已经在mysql中创建了对应的数据库，我们这里创建了一个名为test的测试数据库。执行成功之后我们就可以看到数据库里已经有了我们的t\_person表了，如下图所示：

```
mysql> describe t_person;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(255)	YES		NULL	
sex	varchar(255)	YES		NULL	
address	varchar(255)	YES		NULL	
duty	varchar(255)	YES		NULL	
phone	varchar(255)	YES		NULL	
description	varchar(255)	YES		NULL	
org	int(11)	YES	MUL	NULL	

OK，你会了吗，就是这么简单，如果之前没弄过，就来试试吧！



12

## Hibernate与Spring 配合生成表结构



前几天向大家介绍了一种用工具类生成数据表的方法，不过之前的方法需要使用一个跟项目关系不大的工具类。不免让人觉得有些多余，所以呢，今天再向大家介绍一种方法。即Hibernate与Spring配合生成表结构。

首先，要将Spring的信息配置的web.xml，配置Spring用于初始化容器对象的监听器。

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:
  <display-name>oa_01</display-name>

  <!-- 配置Spring用于初始化容器对象的监听器 -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext*.xml</param-value>
  </context-param>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

然后，将Hibernate的信息配置到Spring的配置文件中，将Hibernate交由Spring来管理。

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-conte
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

  <!-- 自动扫描与装配bean -->
  <context:component-scan base-package="com.tgb.oa"></context:component-scan>

  <!-- 导入外部的properties文件 -->
  <context:property-placeholder location="classpath:jdbc.properties"/>

  <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- 指定hibernate配置文件的位置 -->
```

```

<property name="configLocation" value="classpath:hibernate.cfg.xml"></property>
<!-- 配置c3p0数据库连接池 -->
<property name="dataSource">
    <bean class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <!-- 数据连接信息 -->
        <property name="jdbcUrl" value="jdbc:mysql://127.0.0.1:3307/myoa"></property>
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="user" value="root"></property>
        <property name="password" value="123456"></property>

        <!-- 初始化时获取三个连接（取值应在minPoolSize与maxPoolSize之间。默认值：3） -->
        <property name="initialPoolSize" value="3"></property>
        <!-- 连接池中保留的最小连接数，默认值：3 -->
        <property name="minPoolSize" value="3"></property>
        <!-- 连接池中保留的最大连接数，默认值：15 -->
        <property name="maxPoolSize" value="5"></property>
        <!-- 当连接池中的连接数耗尽的时候，c3p0一次同时获取的连接数，默认值：3 -->
        <property name="acquireIncrement" value="3"></property>
        <!-- 控制数据源内加载的PreparedStatements数量。如果maxStatements与maxStatementsPerConnection均为0，则永不预加载。 -->
        <property name="maxStatements" value="8"></property>
        <!-- maxStatementsPerConnection定义了连接池内单个连接所拥有的最大缓存statements数。Default: 0 -->
        <property name="maxStatementsPerConnection" value="5"></property>
        <!-- 最大空闲时间,1800秒内未使用则连接被丢弃。若为0则永不丢弃。Default: 0 -->
        <property name="maxIdleTime" value="1800"></property>
    </bean>
</property>
</bean>

</beans>

```

这里我将数据库连接信息以及连接池都配置到了Spring中，当然你也可以将数据库连接信息写到Hibernate的配置文件里，Hibernate会有自己默认的连接池配置，但是它没有Spring的好用。具体写到哪看你需要吧。

接下来就是Hibernate的配置了，里面包括数据库方言、是否显示sql语句、更新方式以及实体映射文件。当然也可按上面说的将数据库连接信息写到这里面。

hibernate.cfg.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

```



```

<session-factory>

    <property name="dialect">
        org.hibernate.dialect.MySQL5InnoDBDialect
    </property>

    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>

    <mapping resource="com/tgb/oa/domain/User.hbm.xml"/>

</session-factory>
</hibernate-configuration>

```

实体映射文件，不做过多解释。

User.hbm.xml

```

<?xml version="1.0" <span style="font-family: Arial, Helvetica, sans-serif;">encoding="UTF-8"?</span>>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.tgb.oa.domain">

    <class name="User" table="T_User">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name" />
    </class>

</hibernate-mapping>

```

实体类 User.java

```

package com.tgb.oa.domain;

public class User {

    private String name;
    private Long id;

    public String getName() {
        return name;
    }
}

```

```

public void setName(String name) {
    this.name = name;
}
public Long getId() {
    return id;
}
public void setId(Long id) {
    this.id = id;
}
}

```

当tomcat启动的时候，会先找到web.xml，然后根据web.xml的配置，会找到spring中的applicationContext.xml的配置文件，在applicationContext.xml中有相应的SessionFactory的配置，里面有Hibernate的相关信息，接着就会找到Hibernate-cfg.xml，读取该文件，并找到实体映射文件User.hbm.xml，最后根据User.hbm.xml的配置映射成相应的表结构。

Tomcat启动以后，表结构也跟着生成了，生成表结构后的效果：

```

mysql> describe t_user;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id    | bigint(20)    | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255)  | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+

```

两种生成表结构的方式其实也没有哪种好，哪种不好之说。用工具类生成的方式不需要Spring的参与，但是需要一个工具类来支持；与Spring配合的方式不需要多余的东西，但是需要与Spring配合才能用。如果你只需要Hibernate那就用第一种，如果正好是配合Spring来使用那毫无疑问就用第二种。具体情况吧。



13



## Spring 容器IOC解析及简单实现



最近一段时间，“容器”两个字一直萦绕在我的耳边，甚至是吃饭、睡觉的时候都在我脑子里蹦来蹦去的。随着这些天一次次的交流、讨论，对于容器的理解也逐渐加深。理论上的东西终归要落实到实践，今天就借助Spring容器实现原理，简单说说吧。

简单的说，Spring就是通过工厂+反射将我们的bean放到它的容器中的，当我们想用某个bean的时候，只需要调用getBean("beanID")方法。

原理简单介绍：

Spring容器的原理，其实就是通过解析xml文件，或取到用户配置的bean，然后通过反射将这些bean挨个放到集合中，然后对外提供一个getBean()方法，以便我们获得这些bean。下面是一段简单的模拟代码：

```
package com.tgb.spring.factory;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;

public class ClassPathXmlApplicationContext implements BeanFactory {

    //容器的核心，用来存放注入的Bean
    private Map<String, Object> container = new HashMap<String, Object>();

    //解析xml文件，通过反射将配置的bean放到container中
    public ClassPathXmlApplicationContext(String fileName) throws Exception{
        SAXBuilder sb = new SAXBuilder();
        Document doc = sb.build(this.getClass().getClassLoader().getResourceAsStream(fileName));
        Element root = doc.getRootElement();
        List list = XPath.selectNodes(root, "/beans/bean");

        //扫描配置文件中的bean
        for (int i = 0; i < list.size(); i++) {
            Element bean = (Element) list.get(i);
            String id = bean.getAttributeValue("id");
            String clazz = bean.getAttributeValue("class");
            Object o = Class.forName(clazz).newInstance();
            container.put(id, o);
        }
    }
}
```

```

@Override
public Object getBean(String id) {
    return container.get(id);
}
}

```

首先声明一个存放bean的Map，然后通过jdom解析配置文件，循环遍历所有的节点，并通过反射将它们放到我们之前声明的Map中。然后提供一个getBean()的方法，让我们可以通过bean的Id来找到我们想要的bean。

下面是一个简单的xml配置文件：

```

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

    <bean id="E" class="com.tgb.spring.factory.England" />

    <bean id="S" class="com.tgb.spring.factory.Spain" />

    <bean id="P" class="com.tgb.spring.factory.Portugal" />

</beans>

```

客户端通过调用前面的ClassPathXmlApplicationContext，来加载上面的配置文件，然后就可以通过Id来获得我们需要的bean了：

```

package com.tgb.spring.factory;

public class Test {

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

        //加载配置文件
        BeanFactory f = new ClassPathXmlApplicationContext("applicationContext.xml");

        //英格兰
        Object oe = f.getBean("E");
        Team e = (Team)oe;
        e.say();

        //西班牙
        Object os = f.getBean("S");
        Team s = (Team)os;
        s.say();
    }
}

```

```

        //葡萄牙
        Object op = f.getBean("P");
        Team p = (Team)op;
        p.say();
    }
}

```

输出结果：

```

England：我们是欧洲的中国队，不在乎这次小组没出线...
Spain  ：我们是两届欧洲杯冠军、一届世界杯冠军！
Portugal：我们的C罗一个顶十个！

```

其他代码：

```

//工厂接口
package com.tgb.spring.factory;

public interface BeanFactory {
    Object getBean(String id);
}

//Team接口
package com.tgb.spring.factory;

public interface Team {
    void say();
}

//英格兰
package com.tgb.spring.factory;

public class England implements Team{

    public void say() {
        System.out.println("England：我们是欧洲的中国队，不在乎这次小组没出线...");
    }
}

//西班牙
package com.tgb.spring.factory;

public class Spain implements Team{

```

```
@Override
public void say() {
    System.out.println("Spain: 我们是两届欧洲杯冠军、一届世界杯冠军! ");
}

}

//葡萄牙
package com.tgb.spring.factory;

public class Portugal implements Team {

    @Override
    public void say() {
        System.out.println("Portugal: 我们的C罗一个顶十个! ");
    }

}
```

以上内容是对Spring的一个简单模拟，当然Spring远比这个要复杂的多，也强大的多，而且获取bean的方式也不止通过工厂这一种。这里只是做一个粗略的Demo说说自己对容器的简单理解，向Spring致敬。例子简陋，表达粗糙，欢迎拍砖交流。



14



## Spring 容器AOP的实现原理——动态代理





之前写了一篇关于IOC的博客——《[Spring容器IOC解析及简单实现](#)》(), 今天再来聊聊AOP。大家都知道Spring的两大特性是IOC和AOP。

IOC负责将对象动态的注入到容器,从而达到一种需要谁就注入谁,什么时候需要就什么时候注入的效果,可谓是招之则来,挥之则去。想想都觉得爽,如果现实生活中也有这本事那就爽歪歪了,至于有多爽,各位自己脑补吧;而AOP呢,它实现的就是容器的另一大好处了,就是可以让容器中的对象都享有容器中的公共服务。那么容器是怎么做到的呢?它怎么就能让在它里面的对象自动拥有它提供的公共性服务呢?答案就是我们今天要讨论的内容——动态代理。

动态代理其实并不是什么新鲜的东西,学过设计模式的人都应该知道代理模式,代理模式是一种静态代理,而动态代理就是利用反射和动态编译将代理模式变成动态的。原理跟动态注入一样,代理模式在编译的时候就已经确定代理类将要代理谁,而动态代理在运行的时候才知道自己要代理谁。

Spring的动态代理有两种:一是JDK的动态代理;另一个是cglib动态代理(通过修改字节码来实现代理)。今天咱们主要讨论JDK动态代理的方式。JDK的代理方式主要就是通过反射跟动态编译来实现的,下面咱们就通过代码来看看它具体是怎么实现的。

假设我们要对下面这个用户管理进行代理:

```
//用户管理接口
package com.tgb.proxy;

public interface UserMgr {
    void addUser();
    void delUser();
}

//用户管理的实现
package com.tgb.proxy;

public class UserMgrImpl implements UserMgr {

    @Override
    public void addUser() {
        System.out.println("添加用户.....");
    }

    @Override
    public void delUser() {
        System.out.println("删除用户.....");
    }

}
```

按照代理模式的实现方式，肯定是用一个代理类，让它也实现UserMgr接口，然后在其内部声明一个UserMgrImpl，然后分别调用addUser和delUser方法，并在调用前后加上我们需要的其他操作。但是这样很显然都是写死的，我们怎么做到动态呢？别急，接着看。我们知道，要实现代理，那么我们的代理类跟被代理类都要实现同一接口，但是动态代理的话我们根本不知道我们将要代理谁，也就不知道我们要实现哪个接口，那么要怎么办呢？我们只有知道要代理谁以后，才能给出相应的代理类，那么我们何不等知道要代理谁以后再去生成一个代理类呢？想到这里，我们好像找到了解决的办法，就是动态生成代理类！

这时候我们亲爱的反射才有了用武之地，我们可以写一个方法来接收被代理类，这样我们就可以通过反射知道它的一切信息——包括它的类型、它的方法等等（如果你不知道怎么得到，请先去看看我写的反射的博客《反射一》《反射二》）。

JDK动态代理的两个核心分别是InvocationHandler和Proxy，下面我们就用简单的代码来模拟一下它们是怎么实现的：

InvocationHandler接口：

```
package com.tgb.proxy;

import java.lang.reflect.Method;

public interface InvocationHandler {
    public void invoke(Object o, Method m);
}
```

实现动态代理的关键部分，通过Proxy动态生成我们具体的代理类：

```
package com.tgb.proxy;

import java.io.File;
import java.io.FileWriter;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.net.URL;
import java.net.URLClassLoader;
import javax.tools.JavaCompiler;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;
import javax.tools.JavaCompiler.CompilationTask;

public class Proxy {
    /**
     *
     * @param infce 被代理类的接口
     */
}
```

```

* @param h 代理类
* @return
* @throws Exception
*/
public static Object newProxyInstance(Class infce, InvocationHandler h) throws Exception {
    String methodStr = "";
    String rt = "\r\n";

    //利用反射得到infce的所有方法，并重新组装
    Method[] methods = infce.getMethods();
    for(Method m : methods) {
        methodStr += "    @Override" + rt +
            "    public " + m.getReturnType() + " " + m.getName() + "() {" + rt +
            "        try {" + rt +
            "            Method md = " + infce.getName() + ".class.getMethod(\"" + m.getName() + "\");" + rt +
            "            h.invoke(this, md);" + rt +
            "        }catch(Exception e) {e.printStackTrace();}" + rt +
            "    }" + rt ;
    }

    //生成Java源文件
    String srcCode =
        "package com.tgb.proxy;" + rt +
        "import java.lang.reflect.Method;" + rt +
        "public class $Proxy1 implements " + infce.getName() + "{" + rt +
        "    public $Proxy1(InvocationHandler h) {" + rt +
        "        this.h = h;" + rt +
        "    }" + rt +
        "    com.tgb.proxy.InvocationHandler h;" + rt +
        methodStr + rt +
        "};";
    String fileName =
        "d:/src/com/tgb/proxy/$Proxy1.java";
    File f = new File(fileName);
    FileWriter fw = new FileWriter(f);
    fw.write(srcCode);
    fw.flush();
    fw.close();

    //将Java文件编译成class文件
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    StandardJavaFileManager fileMgr = compiler.getStandardFileManager(null, null, null);
    Iterable units = fileMgr.getJavaFileObjects(fileName);
    CompilationTask t = compiler.getTask(null, fileMgr, null, null, null, units);
    t.call();
}

```

```

fileMgr.close();

//加载到内存，并实例化
URL[] urls = new URL[] {new URL("file://" + "d:/src/")};
URLClassLoader ul = new URLClassLoader(urls);
Class c = ul.loadClass("com.tgb.proxy.$Proxy1");

Constructor ctr = c.getConstructor(InvocationHandler.class);
Object m = ctr.newInstance(h);

return m;
}
}

```

这个类的主要功能就是，根据被代理对象的信息，动态组装一个代理类，生成\$Proxy1.java文件，然后将其编译成\$Proxy1.class。这样我们就可以在运行的时候，根据我们具体的被代理对象生成我们想要的代理类了。这样一来，我们就不需要提前知道我们要代理谁。也就是说，你想代理谁，想要什么样的代理，我们就给你生成一个什么样的代理类。

然后，在客户端我们就可以随意的进行代理了。

```

package com.tgb.proxy;

public class Client {
    public static void main(String[] args) throws Exception {
        UserMgr mgr = new UserMgrImpl();

        //为用户管理添加事务处理
        InvocationHandler h = new TransactionHandler(mgr);
        UserMgr u = (UserMgr)Proxy.newProxyInstance(UserMgr.class,h);

        //为用户管理添加显示方法执行时间的功能
        TimeHandler h2 = new TimeHandler(u);
        u = (UserMgr)Proxy.newProxyInstance(UserMgr.class,h2);

        u.addUser();
        System.out.println("\r\n=====华丽的分割线=====\r\n");
        u.delUser();
    }
}

```

运行结果：

```

开始时间:2014年-07月-15日 15时:48分:54秒
开启事务.....
添加用户.....
提交事务.....
结束时间:2014年-07月-15日 15时:48分:57秒
耗时: 3秒

```

=====华丽的分割线=====

```

开始时间:2014年-07月-15日 15时:48分:57秒
开启事务.....
删除用户.....
提交事务.....
结束时间:2014年-07月-15日 15时:49分:00秒
耗时: 3秒

```

这里我写了两个代理的功能，一个是事务处理，一个是显示方法执行时间的代理，当然都是非常简单的写法，只是为了说明这个原理。当然，我们可以想Spring那样将这些AOP写到配置文件，因为之前那篇已经写了怎么通过配置文件注入了，这里就不重复贴了。到这里，你可能会有一个疑问：你上面说，只要放到容器里的对象，都会有容器的公共服务，我怎么没看出来呢？好，那我们就继续看一下我们的代理功能：

#### 事务处理：

```

package com.tgb.proxy;

import java.lang.reflect.Method;

public class TransactionHandler implements InvocationHandler {

    private Object target;

    public TransactionHandler(Object target) {
        super();
        this.target = target;
    }

    @Override
    public void invoke(Object o, Method m) {
        System.out.println("开启事务.....");
        try {
            m.invoke(target);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("提交事务.....");
    }
}

```

```
}  
  
}
```

从代码中不难看出，我们代理的功能里没有涉及到任何被代理对象的具体信息，这样有什么好处呢？这样的好处就是将代理要做的事情跟被代理的对象完全分开，这样一来我们就可以在代理和被代理之间随意的进行组合了。也就是说同一个功能我们只需要一个。同样的功能只有一个，那么这个功能不就是公共的功能吗？不管容器中有多少给对象，都可以享受容器提供的服务了。这就是容器的好处。

不知道我讲的够不够清楚，欢迎大家积极交流、讨论。



15



## 简单模拟Hibernate实现原理



之前写了Spring的实现原理，今天我们接着聊聊Hibernate的实现原理，这篇文章只是简单的模拟一下Hibernate的原理，主要是模拟了一下Hibernate的Session类。好了，废话不多说，先看看我们的代码：

```
package com.tgb.hibernate;

import java.lang.reflect.Method;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;

import com.tgb.hibernate.model.User;

public class Session {

    //表名
    String tableName = "user";

    //存放数据库连接配置
    private Map<String, String> conConfig = new HashMap<String, String>();

    //存放实体属性
    private Map<String, String> columns = new HashMap<String, String>();

    //实体的get方法集合
    String methodNames[];

    public Session () {

        //初始化实体，这里就不用读取配置文件的方式了，有点麻烦。
        columns.put("id", "id");
        columns.put("name", "name");
        columns.put("password", "password");
        methodNames = new String[columns.size()];

    }
}
```



```

/**
 * 创建数据库连接
 * @return
 * @throws Exception
 */
public Connection createConnection() throws Exception {
    //解析xml文件，读取数据库连接配置
    SAXBuilder sb = new SAXBuilder();
    Document doc = sb.build(this.getClass().getClassLoader().getResourceAsStream("hibernate.cfg.xml"));
    Element root = doc.getRootElement();
    List list = XPath.selectNodes(root, "/hibernate-configuration/property");

    for (int i = 0; i < list.size(); i++) {
        Element property = (Element) list.get(i);
        String name = property.getAttributeValue("name");
        String value = property.getText();
        conConfig.put(name, value);
    }

    //根据配置文件获得数据库连接
    Class.forName(conConfig.get("driver"));
    Connection con = DriverManager.getConnection(conConfig.get("url"), conConfig.get("username"), conConfig.get("password"));

    return con;
}

/**
 * save方法，持久化对象
 * @param user
 */
public void save(User user) {

    String sql = createSql();
    System.out.println(sql);

    try {
        Connection con = createConnection();
        PreparedStatement state = (PreparedStatement) con.prepareStatement(sql);

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

            //得到每一个方法的对象
            Method method = user.getClass().getMethod(methodNames[i]);

            //得到他的返回类型

```

```

        Class cla = method.getReturnType();

        //根据返回类型来设置插入数据库中的每个属性值。
        if(cla.getName().equals("java.lang.String")) {
            String returnValue = (String)method.invoke(user);
            state.setString(i+1, returnValue);
        }
        else if(cla.getName().equals("int")) {
            Integer returnValue = (Integer) method.invoke(user);
            state.setInt(i+1, returnValue);
        }
    }

    state.executeUpdate();
    state.close();
    con.close();

} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
}

/**
 * 得到sql语句
 * @return 返回sql语句
 */
private String createSql() {

    //strColumn代表数据库中表中的属性列。并将其连接起来。
    String strColumn = "";
    int index=0;
    for(String key :columns.keySet())
    {
        strColumn +=key+",";
        String v = columns.get(key);

        //获得属性的get方法，需要将属性第一个字母大写如：getId()
        v = "get" + Character.toUpperCase(v.charAt(0)) + v.substring(1);
        methodNames[index] = v;
        index++;
    }
}

```

```

    strColumn = strColumn.substring(0, strColumn.length()-1);

    //拼接参数占位符, 即: (?, ?, ?)
    String strValue = "";
    for(int i=0;i<columns.size();i++)
        strValue += "?,";

    strValue = strValue.substring(0,strValue.length()-1);

    String sql = "insert into " + tableName + "(" + strColumn + ")" + " values (" + strValue + ")";
    return sql;
}
}

```

以上代码主要是完成了Hibernate的save()方法, 该类有一个构造方法, 一个构建sql语句的方法, 一个获得数据库连接的方法。最后通过save()方法结合前面几个方法获得结果, 将实体对象持久化到数据库。

基本原理就是: 首先, 获得数据库连接的基本信息; 然后, 获得实体的映射信息; 接着, 也是最关键的步骤, 根据前面获得的信息, 组装出各种sql语句(本例只有简单的insert), 将实体按照不同的要求查找或更新(增、删、改)到数据库。

当然Hibernate的具体实现远没有这么简单, Hibernate中大量运用了cglib的动态代理, 其中load()方法就是一个例子。大家都知道, 调用load()方法是Hibernate不会向数据库发sql语句, load()方法得到的是目标实体的一个代理类, 等到真正用到实体对象的时候才会去数据库查询。这也是Hibernate的一种懒加载的实现方式。

总结一句话, 这些框架之所以能够做到灵活, 就是因为它们都很好的利用了懒加载机制, 在运行期在确定实例化谁, 需要谁实例化谁, 什么时候需要, 什么时候实例化。这样设计出来能不灵活吗? 这些思想值得我们好好研究, 并运用到我们的设计中去。



16

## Struts2 内部是如何工作的



前面说完了Spring、Hibernate，很自然今天轮到struts了。struts的核心原理就是通过拦截器来处理客户端的请求，经过拦截器一系列的处理后，再交给Action。下面先看看struts官方的工作原理图：

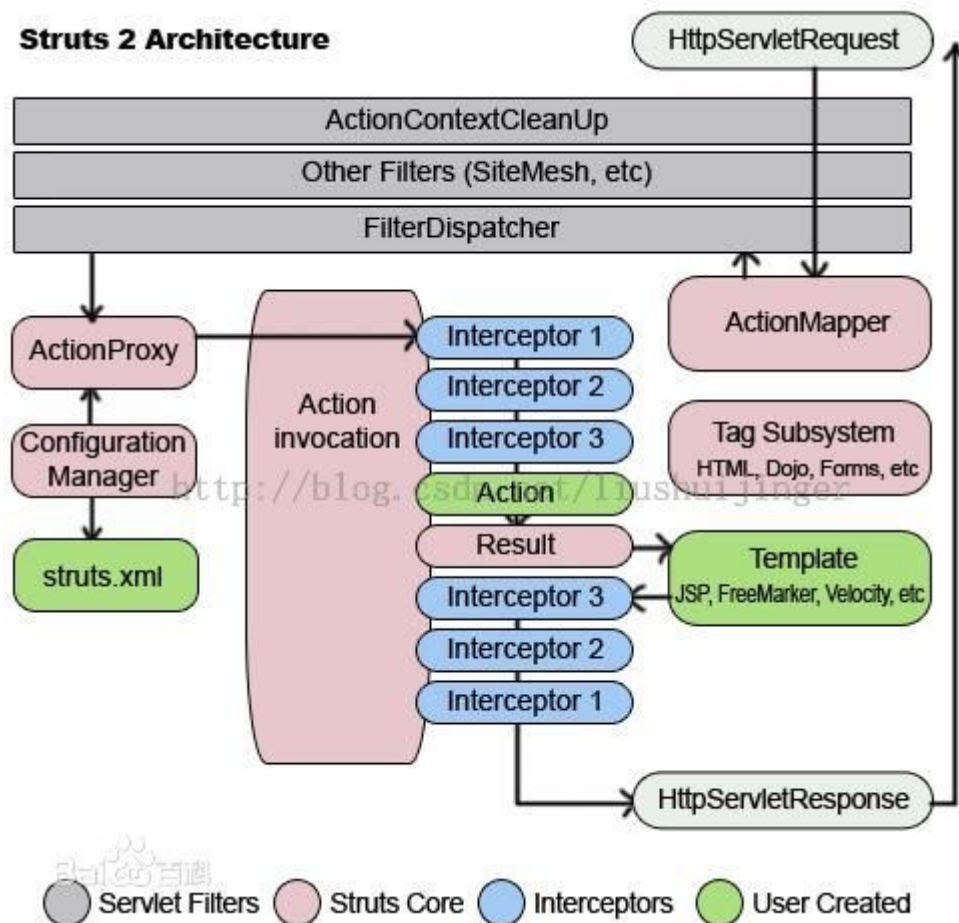


图1 struts原理图

简单分析一下：首先客户端发来HttpServletRequest请求，传递给FilterDispatcher（ActionMapper是访问静态资源（struts的jar文件等）时用的，平时很少用），然后FilterDispatcher会为我们创建一个ActionProxy，ActionProxy会通过ConfigurationManager获得struts.xml文件中的信息，ActionProxy拥有一个ActionInvocation实例，通过调用ActionInvocation的invoke()方法，来挨个处理Interceptor，最后处理Action，接着Result返回，再逆序经过Interceptor，最后得到HttpServletResponse返回给客户端。

如果不太明白呢，那就看看下面这张时序图，也许你就懂了：

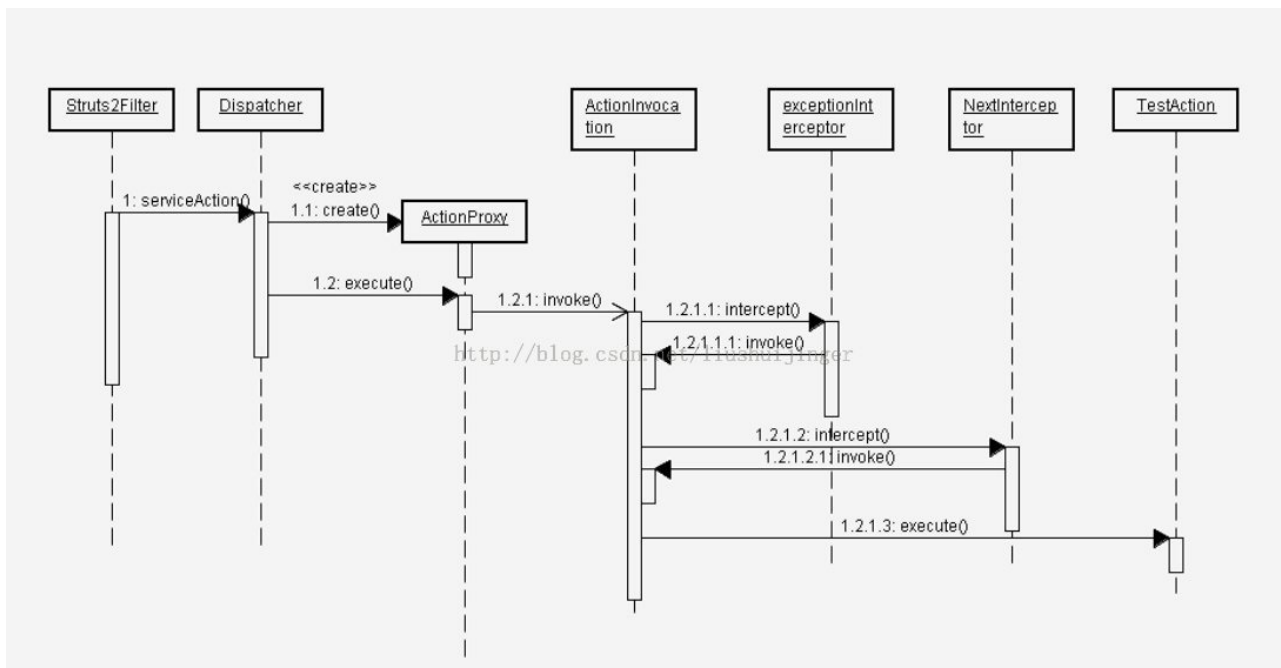


图2 struts原理时序图

上面的时序图逻辑就比较清晰了，我就不过多解释了。看完struts的原理图，我们还是需要通过代码来进一步了解它具体是怎么实现的。首先，我们需要一个ActionInvocation：

```

package com.tgb.struts;
import java.util.ArrayList;
import java.util.List;

public class ActionInvocation {
    List<Interceptor> interceptors = new ArrayList<Interceptor>();
    int index = -1;
    Action a = new Action();

    public ActionInvocation() {
        this.interceptors.add(new FirstInterceptor());
        this.interceptors.add(new SecondInterceptor());
    }

    public void invoke() {
        index ++;
        if(index >= this.interceptors.size()) {
            a.execute();
        }else {

            this.interceptors.get(index).intercept(this);
        }
    }
}
  
```

```

    }
}

```

我们实现的ActionInvocation是将Interceptor写在里面的，但实际上是通过反射加载的，原理同之前写的Spring与Hibernate的博客，相同的代码就不在这里占用篇幅了，也没啥意思。不知道怎么实现的朋友请查看前面几篇博客。

接下来是我们的Interceptor接口以及两个简单的实现：

```

package com.tgb.struts;

public interface Interceptor {
    public void intercept(ActionInvocation invocation) ;
}

package com.tgb.struts;

public class FirstInterceptor implements Interceptor {

    public void intercept(ActionInvocation invocation) {
        System.out.println("FirstInterceptor Begin...");
        invocation.invoke();
        System.out.println("FirstInterceptor End...");
    }

}

package com.tgb.struts;

public class SecondInterceptor implements Interceptor {

    public void intercept(ActionInvocation invocation) {
        System.out.println("SecondInterceptor Begin...");
        invocation.invoke();
        System.out.println("SecondInterceptor End...");
    }

}

```

然后就是我们的Action：

```

[java] view plaincopy在CODE上查看代码片派生到我的代码片
package com.tgb.struts;

public class Action {

```

```
public void execute() {
    System.out.println("Action Run...");
}
}
```

最后是我们的客户端调用：

```
package com.tgb.struts;

public class Client {
    public static void main(String[] args) {
        new ActionInvocation().invoke();
    }
}
```

差点忘了，还有我们最后的执行结果：

```
FirstInterceptor Begin...
SecondInterceptor Begin...
Action Run...
SecondInterceptor End...
FirstInterceptor End...
```

通过上面的执行结果，我们可以很清楚的看到，请求来的时候会按照顺序被所有配置的拦截器拦截一遍，然后返回的时候会按照逆序再被拦截器拦截一遍。这跟数据结构中的“栈”非常类似（FIFO-先进先出），数据结构我不太懂，也许这样比喻有些不妥。各位根据自己的认识理解吧。

最近一直在研究这三大框架，折腾半天它们都离不开集合，离不开反射。其实它们道理都是想通的，搞懂一个，其他的也就很好懂了。等着吧，早晚咱们自己写一个更好用的。





17



基于注解的SSH将配置精简到极致



很早之前就想写一篇关于SSH整合的博客了，但是一直觉得使用SSH的时候那么多的配置文件，严重破坏了我们代码整体性，比如你要看两个实体的关系还得对照`hbm.xml`文件，要厘清一个Action可能需要对照`applicationContext.xml`和`struts*.xml`文件。总之过多的配置文件破坏代码的整体性，会打乱代码的连续性，因为很多情况下你需要一边看Java代码，一边看xml的配置，采用注解就能很好的解决这个问题。

当然，即使采用注解的方式，也不能完全的丢掉配置文件，因为配置文件是程序的入口，是基础。服务器启动最先加载`web.xml`文件，读取其中的配置信息，将程序运行所需要的信息进行初始化。因为是整合SSH，所以`web.xml`文件中需要配置Spring以及Struts的信息，同时Spring跟Struts也需要进行一些基本的配置。

使用注解的方式，配置文件最少可以精简到三个，`web.xml`、`applicationContext.xml`和`struts.xml`。Hibernate可以完全交给Spring来管理，这样连`hibernate.cfg.xml`也省了。下面就一起看看这些基本的配置吧！

`web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="SSH" version="2.5">
    <display-name>ssh</display-name>
    <welcome-file-list>
        <welcome-file>addUser.jsp</welcome-file>
    </welcome-file-list>

    <!-- 配置Spring的监听器，用于初始化ApplicationContext对象 -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext*.xml</param-value>
    </context-param>

    <!-- struts2 的配置 -->
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
    <init-param>
        <param-name>filterConfig</param-name>
        <param-value>classpath:struts.xml</param-value>
    </init-param>

    <!-- 自动扫描action -->
    <init-param>
```

```

    <param-name>actionPackages</param-name>
    <param-value>com.ssh</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>struts2</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>

```

web.xml中包含了Spring和struts的基本配置，自动扫描Action的配置就是告诉tomcat，我要使用注解来配置struts。

applicationContext.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

  <!-- 自动扫描与装配bean -->
  <context:component-scan base-package="com.tgb.ssh"></context:component-scan>

  <!-- dbcp配置 -->
  <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">
    <property name="driverClassName">
      <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
      <value>jdbc:mysql://127.0.0.1:3307/ssh</value>
    </property>
    <property name="username">
      <value>root</value>
    </property>
    <property name="password">
      <value>123456</value>
    </property>
  </bean>

  <bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource">
      <ref local="dataSource" />
    </property>
  </bean>

```

```

</property>
<property name="hibernateProperties">
  <props>
    <!--配置Hibernate的方言-->
    <prop key="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </prop>
    <prop key="hibernate.hbm2ddl.auto">update</prop>

    <!--格式化输出sql语句-->
    <prop key="hibernate.show_sql">true</prop>
    <prop key="hibernate.format_sql">true</prop>
    <prop key="hibernate.use_sql_comments">false</prop>
  </props>
</property>

<!--自动扫描实体 -->
<property name="packagesToScan" value="com.tgb.ssh.model" />
</bean>

<!-- 用注解来实现事务管理 -->
<bean id="txManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
<tx:annotation-driven transaction-manager="txManager"/>

</beans>

```

applicationContext.xml里配置了数据库连接的基本信息（对hibernate的管理），还有对所有bean的自动装配管理和事务的管理。struts.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

  <!-- 开启使用开发模式，详细错误提示 -->
  <constant name="struts.devMode" value="true" />
  <!-- 将对象交给spring管理 -->
  <constant name="struts.objectFactory" value="spring" />
  <!-- 指定资源编码类型 -->
  <constant name="struts.i18n.encoding" value="UTF-8" />
  <!-- 指定每次请求到达，重新加载资源文件 -->

```

```

<constant name="struts.i18n.reload" value="false" />
<!-- 指定每次配置文件更改后，自动重新加载 -->
<constant name="struts.configuration.xml.reload" value="false" />
<!-- 默认后缀名 -->
<constant name="struts.action.extension" value="action," />

</struts>

```

struts.xml里配置了一些struts的基本参数，并告诉容器用Spring来管理自己。

到这里一个基本的SSH的配置就算完成了，配置很简单，而且每一项配置都有说明，相信理解上不会有什么问题。基础的配置就这么多，下面就是我们的注解发挥作用的时候了。

userAdd.jsp

```

<%@ page language="java" import="java.util.*" pageEncoding="utf-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>添加用户</title>
</head>

<body>
<form method="post" action="addUser">
    用户名: <input type="text" name="user.name"><br>
    密码: <input type="password" name="user.password"><br>
    <input type="submit" value="登录"/>
</form>
</body>
</html>

```

用户添加页面，将用户信息提交给用户Action。

UserAction

```

package com.tgb.ssh.action;

import javax.annotation.Resource;

import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;

import com.opensymphony.xwork2.ActionSupport;
import com.tgb.ssh.model.User;
import com.tgb.ssh.service.UserManager;

```

```

@Results( { @Result(name="success",location="/success.jsp"),
            @Result(name="failure",location="/failure.jsp") })
public class UserAction extends ActionSupport {
    @Resource
    private UserManager userManager;
    private User user;

    @Action(value="addUser")
    public String addUser() {
        try {
            userManager.addUser(user);
        } catch (Exception e) {
            e.printStackTrace();
            return "failure";
        }
        return "success";
    }

    public User getUser() {
        return user;
    }

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

```

UserAction通过注解配置Action的名字和返回的页面，通过@Resource活动Spring注入的UserManager对象，然后进行相应的操作。Action里还有@Namespace、@InterceptorRef等很多注解可以用，根据自己需要选择吧。

### UserManager

```

package com.tgb.ssh.service;

import javax.annotation.Resource;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.tgb.ssh.dao.UserDao;

```

```
import com.tgb.ssh.model.User;

@Service
@Transactional
public class UserManager {
    @Resource
    UserDao userDao;

    public void addUser(User user) {
        userDao.addUser(user);
    }
}
```

UserManager通过@Service自动装配到Spring的容器，为其他组件提供服务；通过@Transactional进行事务的管理；通过@Resource注入UserDao对象。

UserDao

```
package com.tgb.ssh.dao;

import javax.annotation.Resource;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.springframework.stereotype.Repository;

import com.tgb.ssh.model.User;

@Repository
public class UserDao {
    @Resource(name="sessionFactory")
    private SessionFactory sessionFactory;

    public void addUser(User user ) {
        Session session = sessionFactory.getCurrentSession();
        session.save(user);
    }
}
```

UserDao通过@Repository自动装配到Spring的容器，通过@Resource获得SessionFactory，将User对象持久化。

## User

```
package com.tgb.ssh.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity(name="t_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private String password;

    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 String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

User通过@Entity将实体类映射到数据库，生成t\_user表，通过@Id定义表的Id，通过@GenerateValue定义Id的生成策略。



好了，到此为止，基于注解的SSH就算是搭建完成了。基础的搭建已经使注解简洁的优势初现端倪，随着开发的进行，代码不断地增加，其简洁的风格相比传统配置文件的方式会更加明显。因为如果采用配置文件的方式，每增加一个Action都需要在struts.xml和applicationContext.xml文件增加一段代码；每多一个实体，也需要多一个\*.hbm.xml文件。配置文件泛滥是一件让人头疼的事情。

注解好处多多，而且也越来越流行，但配置文件也并不是一无是处。注解有注解的好，配置文件有配置文件的妙。还是那句话，技术没有好坏之分，只有合适不合适之别。一味的追求技术的好与坏不是明智之举，选择一个合适的才是真正的设计之道。本文主旨不在于告诉你注解比配置文件好，而是向大家介绍另一种方式，可以多一种选择，也许你会找到一种更合适的方式。



18

## Hibernate动态模型+JRebel实现动态创建表



项目用的是SSH基础框架，其中有一些信息很类似，但又不尽相同。如果每一个建一个实体的话，那样实体会太多，如果分组抽象，然后继承，又不是特别有规律。鉴于这种情况，就打算让用户自己配置要添加的字段，然后生成对应的表。

需要动态配置的部分实例：

单位	信用监管信息项	信息字段	上报方式	上报频率
市工商局	全市著名商标企业名单	企业名称、工商注册号、地址、商标名称、使用商品、负责人、联系电话、评选日期、备注	专网报送	每年1月
	中国驰名商标企业名单	企业名称、工商注册号、地址、商标名称、使用商品、负责人、联系电话、评选日期、备注	专网报送	每年1月
市国税局	A级纳税信用等级纳税人名单	纳税识别号、纳税人名称、法人代表、经营地址、登记注册类型、评定年度	专网报送	每两年即时上报
市地税局	A级纳税信用等级纳税人名单	计算机代码、税务登记证号码、纳税人名称、组织机构代码、税务机关代码、法人代表、经营地址、登记注册类型、税务机关名称、信用等级评定级别、评定年度	专网报送	每年2月
市安监局	市直安全生产先进单位名单	单位名称、工商注册号、组织机构代码、表彰年度	专网报送	每年3月
建设局	优良信息	企业名称、工商注册号、荣誉称号、决定文书号、授予日期、授予组织、有效期限	专网报送	每年1月
市国土局	优良信息	企业名称、工商注册号、荣誉称号、决定文书号、授予日期、授予组织、有效期限	专网报送	每年1月
	河北省名优产品（服务名牌）	企业名称、组织机构代码、产品名称、规格型号、授予日期、有效期	专网报送	每年1月

上图只是一小部分，一个一个组合起来大概有三百多。每一项对应一个实体，显然不好，就算是按照规律归类还是有不少，于是就想到了在运行期来确定这些东西。开始有尝试过动态编译生成实体类，后来发现在数据存取上都存在问题，因为是后来生成的，所以只能用反射来获取，这样一来无法事先确定类型，也就没法用注入的方式接收前端传过来的数据，也不能向前端提供数据了。后来决定用Hibernate的动态模型来处理这个问题，可能有的人不是很了解Hibernate的动态模型，下面我们就来介绍一下。

我们通常用实体类来跟表进行映射，当我们需要一个user表的时候，通常都需要写一个类似下面的实体类：

```
public class User {
    private Long id;
    private String name;
    private String password;

    // setter、getter...
}
```

然后用配置文件或注解来描述映射关系，如果使用动态模型的话，则不需要编写实体类，只需要写一个配置文件即可：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
```

```

<hibernate-mapping>
  <!-- 此处不需要类名，和包名 -->
  <class entity-name="User">
    <id name="id" type="java.lang.Long" column="ID">
      <generator class="native"/>
    </id>
    <property name="name" type="java.lang.String" column="name"/>
    <property name="password" type="java.lang.String" column="password"/>
  </class>
</hibernate-mapping>

```

然后通过Map进行操作：

```

session.beginTransaction();
//通过Map映射实体与数据库
Map user = new HashMap();
user.put("name", "动态模型");
user.put("password", "123456");
session.save("User", user);
session.getTransaction().commit();
session.close();

```

发出的SQL语句：insert into User (name, password) values (?, ?)

有人说动态映射存入数据很方便，但是从数据取出数据好像比较难处理，其实这个问题可以通过事先做好约定来解决。

上一篇博客介绍了JRebel，它可以让Tomcat支持热部署。JRebel+Hibernate动态模型双剑合璧，就可以实现我们动态建表的要求了。

在Spring的配置文件中加入：

```

<property name="mappingLocations">
  <list>
    <value>classpath:/com/tgb/entitycfg/*.hbm.xml</value>
  </list>
</property>

```

采用通配符来配置hbm.xml文件，就是为了兼容运行期生成的动态模型配置文件，而JRebel可以检测到配置文件的变化，从而将新增的配置加载进来，需要说明的是JRebel的动态加载属于懒加载，即在你用到修改的东西是，才会将你修改的内容重新加载进来。

我也是初次使用Hibernate动态模型，目前也算是尝试阶段吧，如果各位谁用过或者对动态模型感兴趣欢迎留言交流。



T



19

## 提高用户体验之404处理



只要做过WEB开发人对于“404”已经再熟悉不过了。当我们访问的资源不存在时，它就会跑出来跟你打招呼啦。但是默认情况下，404页面比较简陋，不是很友好。而且一般用户不知道404是个神马东东，还以为程序写的有问题呢。这样一来用户体验就打折扣了。所以通常情况下，我们都需要对这些常见的错误进行处理。

## HTTP Status 404 - /examples/servlets/

**type** Status report

**message** /examples/servlets/

**description** The requested resource (/examples/servlets/) is not available.

Apache Tomcat/5.5.23

Tomcat默认的404页

为了提升用户体验，我们需要简单的配置一下，来让程序遇到404后跳转到我们指定的页面。首先，需要在web.xml文件加入如下配置：

```
<!--404处理 -->
<error-page>
  <error-code>404</error-code>
  <location>/404.jsp</location>
</error-page>
```

以上配置帮我们处理了大部分不存在的资源访问错误，但是如果这时我们访问一个不存在的action，还是会出现Could not find action or result的错误。因为struts里并没相应的action来处理404错误，所以我们还要在struts.xml文件里加入几行配置。具体如下：

```
<!--action的404处理 -->
<package name="error" extends="struts-default">
  <default-action-ref name="notFound" />
  <action name="notFound">
    <result>/404.jsp</result>
  </action>
</package>
```

OK，这样一来无论你访问任何不存在的地址或者action，都会跳转到我们的“404.jsp”页面了。



#### 自定义的404页

对比前后两种效果，是不是觉得第二个让人觉得更舒服一些呢？而其实我们只是多做了那么一点点，事情往往就是这样，只需要在原来的基础上多做那么一点点，给人的感觉就会变得不一样。在互联网飞速发展的今天，一个企业的成功与失败可能就在转瞬之间。而那些成功的企业往往就是在某个或者某些方面比别人多做了那么一点。具体是谁我就不明说了，相信每个人心中都有自己的答案。

# 极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/ssh-noob-learning/>