# 环境搭建

1、创建一个web project

注意一点：add maven support

2、在pom.xml文件中，添加一个坐标，发布到tomcat下，看结构是否一致

3、把整个工程的编码和jsp的编码改成utf-8

4、建立maven约束的内容

**src/main/java**

cn.itcast.shoa.domain

存放hibernate的实体类和映射文件

cn.itcast.shoa.dao

存放的是数据库操作时候的dao的接口

cn.itcast.shoa.dao.impl

存放的是数据库操作时候的dao的实现类

cn.itcast.shoa.dao.base

存放的是数据操作的基本的dao接口

cn.itcast.shoa.dao.base.impl

存放的是数据操作的基本的dao的实现类

cn.itcast.shoa.service

存放的是业务逻辑接口

cn.itcast.shoa.service.impl

存放的是业务逻辑实现类

cn.itcast.shoa.struts2.action

存放的是struts2中的action类

cn.itcast.shoa.util

存放的是在做项目的时候的工具类

**src/main/resources**

struts2

存放struts2的配置文件

spring

spring的配置文件

hibernate

hibernate.cfg.xml

**src/test/java**

**src/test/resources**

target

pom.xml

5、编辑pom.xml文件，导入整个工程的jar包

6、在cn.itcast.shoa.domain中写持久化类和映射文件

Person.java

Person.hbm.xml

7、写hibernate的配置文件，把Person.hbm.xm映射文件引入进去

8、写spring的配置文件，引入sessionFactory

8.1 在spring的包中建立这么一个结构的配置文件

applicationContext.xml

<import resource="applicationContext-db.xml"/>

applicationContext-\*.xml

8.2 在applicationContext-db.xml文件中引入sessionFactory

有两种引入sessionFactory的方式

9、对sessionFactory进行测试

10、在spring的配置文件中，写spring的声明式事务管理

依据的原理是aop:使各个通知和目标方法实现松耦合

11、在dao层和service层建立接口和类

PersonDao

PersonDaoImpl

PersonService

PersonServiceImpl

12、把dao层和service层的类放入到spring的配置文件中

13、测试personService,看产生的对象是否是代理对象

14、写action

15、把action放入到spring配置文件中

16、写action的配置文件

17、写web.xml文件

面向接口编程： 体现了Spring强大的IOC 和 DI

/\*

\* 面向借口编程，不管datasource用什么方式传入的，最后用DataSource接口接收

任何一个datasource数据源都必须实现DataSource接口

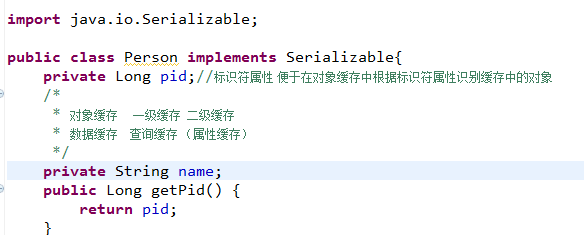
\*/

DataSource datasource=(DataSource)*context*.getBean("dataSource");

datasource.getConnection();

# 小知识点

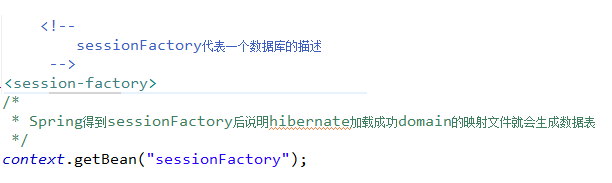
## 标识符的作用



## sessionFactory







## ssh结合后的web.xml文件的执行流程

|  |
| --- |
| <!--  **服务器启动的时候**  1、启动spring容器  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/applicationContext.xml</param-value>  </context-param>  说明：  dao层和service层所有的类将实例化  同时如果bean使用了AOP 此时也会生成代理对象  2、struts2内部  加载default.properties  struts-default.xml  struts-plugin.xml  struts.xml  -->    <!-- 使用ContextLoaderListener初始化Spring容器 -->  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <context-param>  <!-- 该参数是描述Spring配置文件的位置 -->  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/applicationContext.xml</param-value>  </context-param>  <!--  1、该配置可以不写，默认加载WEB-INF/applicationContext.xml  2、该配置可以这么写WEB-INF/\*Context.xml,WEB-INF/application\*.xml这么写不利于测试  3、就是上面的写法，推荐  -->  <!--  **当请求的时候**  1、执行OpenSessionInViewFilter中的除了finally以外的部分  1、从spring容器中把sessionFactory提取出来  2、由sessionFactory获取session  2、执行struts2核心过滤器中除了finally以外的部分  1、先执行拦截器(同一个拦截器的前半段)  2、执行action  3、执行结果集  4、再次执行拦截器(同一个拦截器的后半段)  3、执行struts2核心过滤器中的finally部分  清空actionContext  4、执行OpenSessionInViewFilter中的finally部分  关闭session  说明：  有了OpenSessionInViewFilter的时候，提前开启session 延迟关闭session，从而解决了懒加载的问题  延长了session的生命周期意味着session一级缓存中的对象数据会过长时间停留在内存中  -->    <filter>  <filter-name>OpenSessionInViewFilter</filter-name>  <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>OpenSessionInViewFilter</filter-name>  <url-pattern>\*.action</url-pattern>  </filter-mapping>  <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> |

## spring事务的结构

spring的事务

1、spring容器为事务处理提供了一个平台

1、spring容器为事务提供了一个总的接口

public interface PlatformTransactionManager {

//得到一个事务

TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

//事务的提交

void commit(TransactionStatus status) throws TransactionException;

//事务的回滚

void rollback(TransactionStatus status) throws TransactionException;

}

2、事务的定义接口

public interface TransactionDefinition {

//事务的传播属性 解决事务的嵌套问题

int PROPAGATION\_REQUIRED = 0;//一般用这个

int PROPAGATION\_SUPPORTS = 1;

int PROPAGATION\_MANDATORY = 2;

int PROPAGATION\_REQUIRES\_NEW = 3;

int PROPAGATION\_NOT\_SUPPORTED = 4;

int PROPAGATION\_NEVER = 5;

int PROPAGATION\_NESTED = 6;

//事务的隔离机制（解决多线程并发的线程安全问题造成脏读幻读重复读的问题）

int ISOLATION\_DEFAULT = -1;//一般用这个

int ISOLATION\_READ\_UNCOMMITTED = Connection.TRANSACTION\_READ\_UNCOMMITTED;

int ISOLATION\_READ\_COMMITTED = Connection.TRANSACTION\_READ\_COMMITTED;

int ISOLATION\_REPEATABLE\_READ = Connection.TRANSACTION\_REPEATABLE\_READ;

int ISOLATION\_SERIALIZABLE = Connection.TRANSACTION\_SERIALIZABLE;

//解决事务的是否只读

boolean isReadOnly();

}

3、spring事务提供了一个抽象类

public abstract class AbstractPlatformTransactionManager implements PlatformTransactionManager{

public final void commit(TransactionStatus status) throws TransactionException {

对该方法进行了实现

}

public final void rollback(TransactionStatus status) throws TransactionException {

对该方法进行了实现

}

public final TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {

Object transaction = doGetTransaction();

}

protected abstract Object doGetTransaction() throws TransactionException;

}

4、具体的实现类

1、如果是jdbc编程

public class DataSourceTransactionManager extends AbstractPlatformTransactionManager{

doGetTransaction(){

}

}

2、如果是hibernate编程

public class HibernateTransactionManager extends AbstractPlatformTransactionManager{

doGetTransaction(){

}

}

3、如果是jdo操作数据库

public class JdoTransactionManager extends AbstractPlatformTransactionManager{

doGetTransaction(){

}

}

问题1

为什么会有接口、抽象类、实现类这样的组织结构

接口是用来定义标准的：

得到事务

事务的回滚

事务的提交

抽象类：

1、实现共同的方法

2、在不同的实现方法的时候提供抽象方法

3、具体到事务

得到事务是抽象方法

getTransaction

事务的提交 在该抽象类中得到了实现

commit

事务的回滚 在该抽象类中得到了实现

rollback

说明：只要得到事务以后，事务的提交和回滚都是一样的操作

但是得到事务的方法是不一样的

问题2

spring容器为什么会提供事务接口呢？

实现完全的面向接口编程

public class PersonServiceImpl{

p rivate PlatformTransactionManager platformTransactionManager;

public void setPlatformTransactionManager(PlatformTransactionManager platformTransactionManager){

this.platformTransactionManager = platformTransactionManager;

}

public void aa(){

可以利用PlatformTransactionManager干活了

}

}

# struts2

是一个mvc框架

是一个aop容器

struts2的插件机制

### 1、在web.xml文件中

|  |
| --- |
| <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> |

在该过滤器中：

public class **StrutsPrepareAndExecuteFilter** implements StrutsStatics, Filter {

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

//1、加载了org/apache/struts/**default.properties**文件

//2、private static final String DEFAULT\_CONFIGURATION\_PATHS = "struts-default.xml,struts-plugin.xml,struts.xml";

说明：

1、加载了三种struts2的配置文件

struts-default.xml

struts-plugin.xml

struts.xml

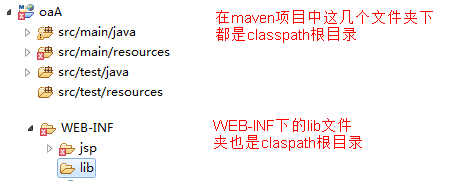
2、配置文件的特点：

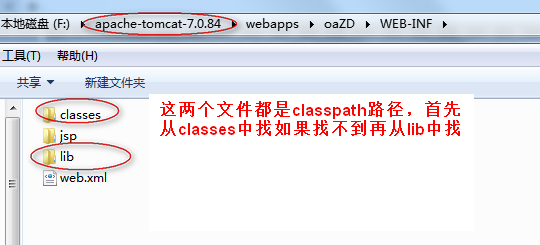
struts-default.xml文件是系统默认的配置文件

struts-plugin.xml是插件机制的配置文件

struts.xml是开发中用到的配置文件

这三种配置文件都是classpath的根目录存放





这三种配置文件的dtd约束都一样，所以在这三种配置文件中可以出现相同的元素

如果出现相同的元素，后者覆盖前者

struts-plugin.xml文件可以有很多个，而且通常情况下是在lib下的jar包中出现

}

}

### 2、对象工厂ObjectFactory （产生action对象，result对象，interceptor对象）

//该方法是struts2创建action的最低层的方法

public Object **buildAction**(String actionName, String namespace, ActionConfig config, Map<String, Object> extraContext) throws Exception {

return buildBean(config.getClassName(), extraContext);

}

//该方式是创建结果集的最低层的方法

public Result **buildResult**(ResultConfig resultConfig, Map<String, Object> extraContext) throws Exception {

}

//该方式是创建拦截器的最低层的方法

public Interceptor **buildInterceptor**(InterceptorConfig interceptorConfig, Map<String, String> interceptorRefParams) throws ConfigurationException

{

}

### 3、静态注入

1、在struts2的配置文件中有这么一些元素

bean

可以配置多个bean

<bean type="com.opensymphony.xwork2.util.ValueStackFactory" name="struts" class="com.opensymphony.xwork2.ognl.OgnlValueStackFactory" />

说明：

1、值栈的产生工厂是OgnlValueStackFactory

2、该类在tomcat容器启动的时候就被加载了，所以只加载一次

3、通过该类就把“值栈的产生工厂”赋值到struts2容器中了，这个过程称为注入

4、所以这种行为称为"静态注入" 在java中只加载一次成为“静态”

package

声明一个action

声明一个拦截器

声明一个结果集

### 4、改变action的产生方式

1、在default.properties文件中有一个常量配置项：

**struts.objectFactory该配置项指明了对象工厂**

2、写一个类继承ObjectFactory,重写buildAction方法

public class CreateActionBean extends ObjectFactory{

@Override

public Object buildAction(String actionName, String namespace,

ActionConfig config, Map<String, Object> extraContext)

throws Exception {

// TODO Auto-generated method stub

System.out.println("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");

return null;

}

}

3、在struts2的配置文件中利用bean元素把CreateActionBean放入到struts2容器中

<bean type="com.opensymphony.xwork2.ObjectFactory" name="aaa" class="cn.itcast.shoa.struts.action.CreateActionBean" />

4、**在配置文件中引入常量struts.objectFactory**

<constant name="struts.objectFactory" value="aaa"></constant>

### 5、分析struts2的插件机制

1、提供了一个jar包

struts2-spring-plugin-2.3.1.2.jar

2、在jar包的根目录下存放了一个文件

<bean type="com.opensymphony.xwork2.ObjectFactory" name="spring" class="org.apache.struts2.spring.StrutsSpringObjectFactory" />

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

重新定义了对象工厂

3、如果一个框架想要和struts2进行整合，提供一个struts-plugin.xml文件，

再提供了一个jar包，该配置文件在jar包的根目录下，然后把该jar包放入到

lib下就可以了

### 6、值栈

将数据放到值栈中 然后利用ognl表达式取数据

1、概念

struts2在后台存放数据的内存结构

2、值栈的生命周期

每一次请求

3、值栈的内存结构：

对象栈中存放:

action

map栈对象栈中存放:

request

session

application

4、ognl表达式对于对象栈的数据直接根据key值提取value值就可以了

对于map栈的数据，加#号访问

1. 对象栈的操作：

ActionContext ctx = ActionContext.getContext(); //1获取ActionContext对象

ValueStack vs =ctx.getValueStack();//2、获取ValueStack对象

ActionContext.getContext().getValueStack() 获取ValueStack对象

把一个对象放入到对象栈：

//放入到栈顶

ActionContext.getContext().getValueStack().push

ActionCOntext.getContext().getValueStack().getRoot().add(0,obj)

//放入到栈底

ActionCOntext.getContext().getValueStack().getRoot().add

//把一个栈顶的元素弹出

ActionContext.getContext().getValueStack().pop();

//得到栈顶的元素

ActionContext.getContext().getValueStack().peek();

如果把一个对象放入到对象栈中，那么可以通过ognl表达式访问其属性来访问其值

6、案例：

ActionContext.getContext().put("departmentList", departmentList);

把departmentList放入到**map栈**中，并且取名为"departmentList"

//value属性指的是要迭代的集合在哪

**iterator 该标签有一个规则：当前正在迭代的元素在栈顶也就是当前的department元素在栈顶所以后面的 name description 在取值的时候不加#号**

**如果s:iterator中的value属性不写，默认迭代栈顶的元素**

<s:iterator value="#departmentList">

<tr class="TableDetail1 template">

<td><s:property value="name"/></td>

<td><s:property value="description"/></td>

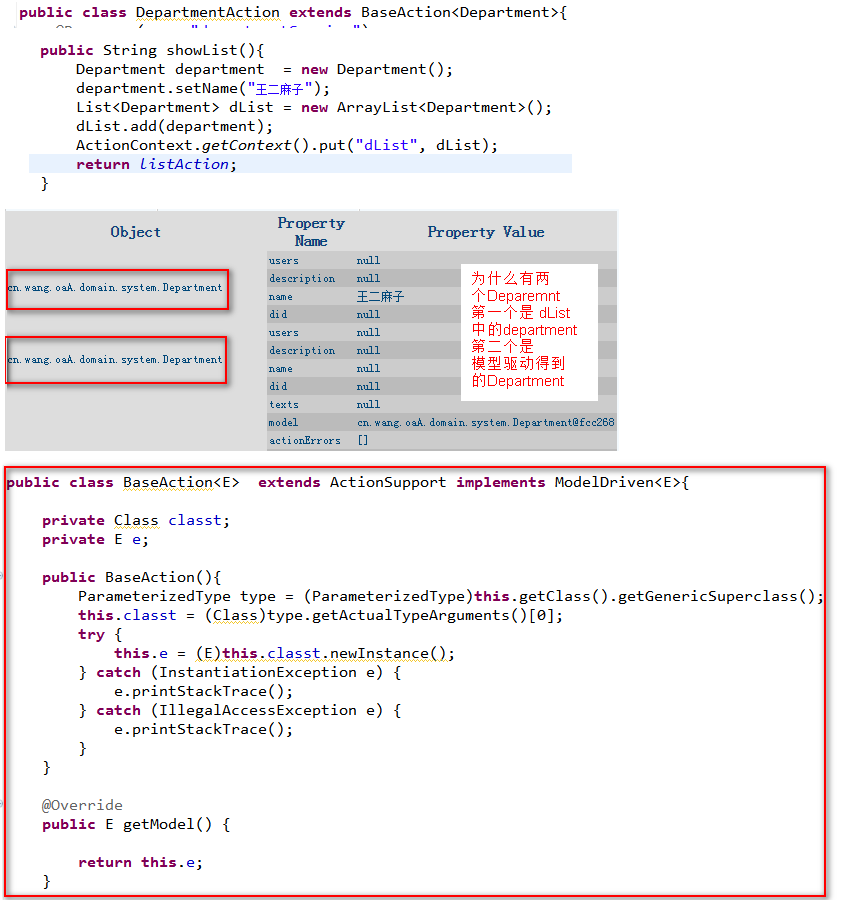
<td><a onClick="return window.confirm('这将删除所有的下级部门，您确定要删除吗？')" href="#">删除</a>

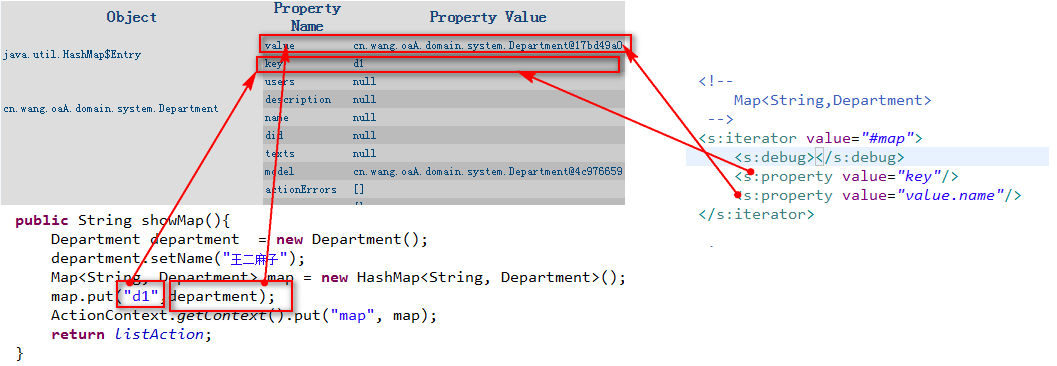
<a href="saveUI.html">修改</a>

</td>

</tr>

</s:iterator>





### 7、在struts2中获取页面上表单的数据有两种方案：

1、利用属性驱动获取表单中的数据

2、利用模型驱动获取表单中的数据

原理：

1、struts2内部有两个拦截器

在没有任何配置拦截器的情况下，struts2会执行默认的拦截器栈defaultStack

defaultStack中：

<interceptor name="params" class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"/>

2、在action中

Department department = this.departmentService.getDepartmentById(this.getModel().getDid()); 1

ActionContext.getContext().getValueStack().push(department); 2

ActionContext.getContext().getValueStack().setParameter("name", "bbbb"); 3

在第二行代码中，把department放入到对象栈的栈顶，所以department中的属性直接在对象栈中可以访问

利用第三行代码访问了对象栈中的name属性，并且把name属性的值改变成"bbbb"

3、属性驱动的原理 ParameterInterceptor

1、当前访问的action在对象栈中，所以action 中的属性就为对象栈中的key值

2、页面表单中的name属性的值和action中属性的名称保持一致

在ParameterInterceptor中

//把页面上表单中的元素的key,value封装成一个map

Map<String, Object> parameters = retrieveParameters(ac);

//从Map<String, Object>中把key和value值提取出来

String name = entry.getKey();

Object value = entry.getValue();

//给对象栈中的key值赋值相应的value的值

newStack.setParameter(name, value);

4、模型驱动的原理

ParameterInterceptor ModelDriverInterceptor

Object action = invocation.getAction();

if (action instanceof ModelDriven) {

ModelDriven modelDriven = (ModelDriven) action;

ValueStack stack = invocation.getStack();

Object model = modelDriven.getModel();

if (model != null) {

stack.push(model);

}

if (refreshModelBeforeResult) {

invocation.addPreResultListener(new RefreshModelBeforeResult(modelDriven, model));

}

}

作用就是把模型驱动对象放入到栈顶

### struts2全局异常处理

**1、步骤**

1、建立一个异常处理类

public class MyException extends ActionSupport{

private Exception exception;

public Exception getException() {

return exception;

}

public void setException(Exception exception) {

this.exception = exception;

}

public String execute(){

ActionContext.getContext().getValueStack().push(exception.getMessage());

return "error";

}

}

2、在struts2的配置文件中进行配置

<package name="struts-global" namespace="/" extends="struts-default">

<global-results>

<result name="errHandler" type="chain">

<param name="actionName">errorProcessor</param>

</result>

</global-results>

<global-exception-mappings>

<exception-mapping exception="java.lang.Exception"result="errHandler" />

</global-exception-mappings>

<action name="errorProcessor" class="cn.itcast.shoa.exception.MyException">

<result name="error">WEB-INF/jsp/error.jsp</result>

</action>

</package>

3、如果role模块用错误处理

<package name="role" namespace="/" extends="struts-global">

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

<result name="listAction">WEB-INF/jsp/system/role/list.jsp</result>

<result name="addUI">WEB-INF/jsp/system/role/add.jsp</result>

<result name="updateUI">WEB-INF/jsp/system/role/update.jsp</result>

<result name="action2action" type="redirectAction">roleAction\_showAllRole.action</result>

</action>

</package>

从上述的代码中可以看出：extends="struts-global",所以role模块就拥有了struts-global包(package)里面

的所有的功能

1. **原理：**

**谁调用了异常所在的方法，异常就抛给谁**

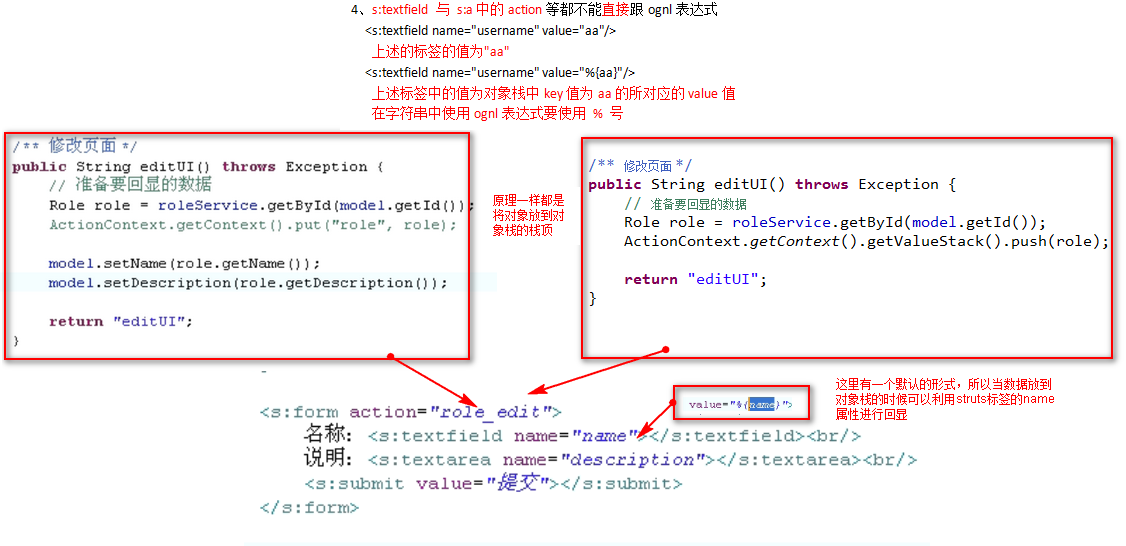
**Service中抛出异常 action是调用者，所以异常给了action ，当action中出现异常时就会调用默认的拦截器**



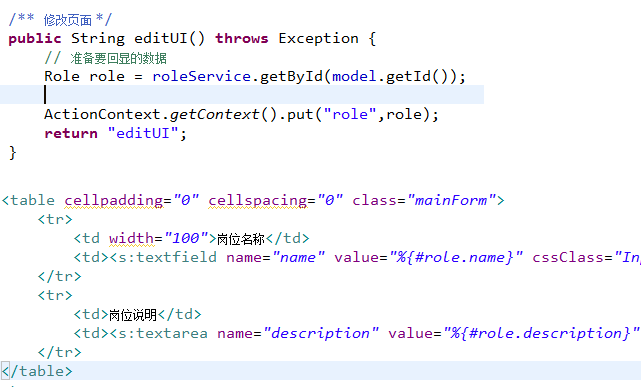
### struts2数据回显

把一个对象放入到对象栈或者map栈中都可以进行回显：

1.放在对象栈中的属性可以利用struts2标签中的name属性进行回显



2. 放在map栈中的属性可以利用struts2标签中的value属性进行回显



# Shs结合以后的执行流程

## 1、服务器tomcat启动的时候

做了两件事

Spring容器中所有的默认配置的bean全部实例化

Struts2的所有配置文件被加载

1、启动spring容器

<listener>

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

</listener>

<context-param>

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

<param-value>classpath:spring/applicationContext.xml</param-value>

</context-param>

说明：

dao层和service层所有的类将实例化包括sessionFactory也已经实例化

2、struts2内部

加载default.properties

struts-default.xml

struts-plugin.xml

struts.xml

## 2、当请求的时候（责任链模式）

1、执行OpenSessionInViewFilter中的除了finally以外的部分

1、从spring容器中把sessionFactory提取出来

2、由sessionFactory获取session

2、执行struts2核心过滤器中除了finally以外的部分

1、先执行拦截器

2、执行action

3、执行结果集

4、再次执行拦截器

3、执行struts2核心过滤器中的finally部分

清空actionContext

4、执行OpenSessionInViewFilter中的finally部分

关闭session

说明：

有了OpenSessionInViewFilter的时候，提前开启session

延迟关闭session，从而解决了懒加载的问题

懒加载的问题：

单表

根据一的一方查询多的一方（是懒加载） 只有在遍历班级中的学生的时候，才会加载学生 所以要想在页面中ognl表达式 classes.students要想获得数据session就不能关闭

根据多的一方查询一的一方（是懒加载）：比如根据学生得到班级，得到的班级是代理对象，只有在得到班级的属性的时候才会加载班级的数据

所以要想在页面中ognl表达式 Student.classes.name要想获得数据session就不能关闭

延长了session的生命周期意味着session一级缓存中的对象数据会过长时间停留在内存中

### 扩展：struts2中的拦截器也是拦截器模式

# 重构步骤

### 0、泛型

### 重构DAO

baseDao

\* Type 和 Object都太大了

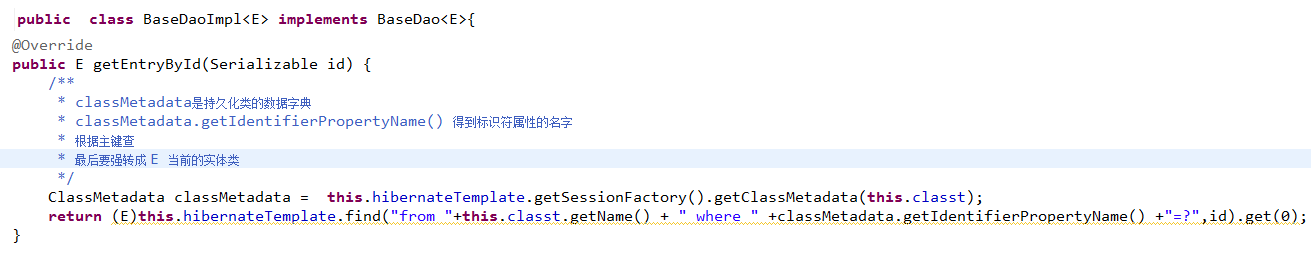
\* Serializable该类型可以接受所有的基本类型和String类型

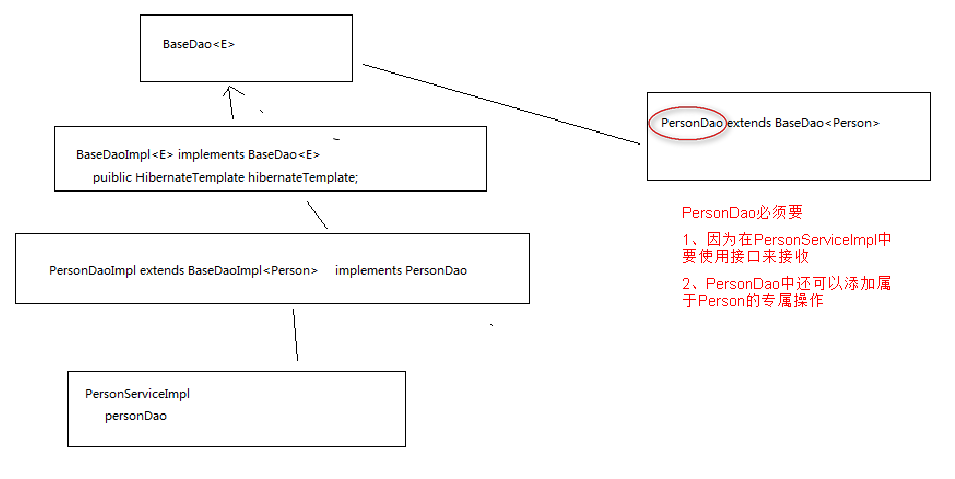
\* 所有的基本类型的包装类全部实现了Serializable接口所以可以使用Serializable来接收

\* 它可以接受所有的持久化类的主键

\* **@param** id

baseDaoImpl（不能放到spring容器进行实例化，因为他有泛型，如果放到spring容器中就没有泛型了）





### 2、扫描注解（*personDao personService personAction*）

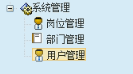
### 3、事务注解

### 4、重构action

baseAction

# 项目的模块

## 系统管理





步骤

### 1、需要分析

### 2、数据库的表的设计

### 3、写持久化类和映射文件

cn/itcast/sh05oa/domain/system

User.java

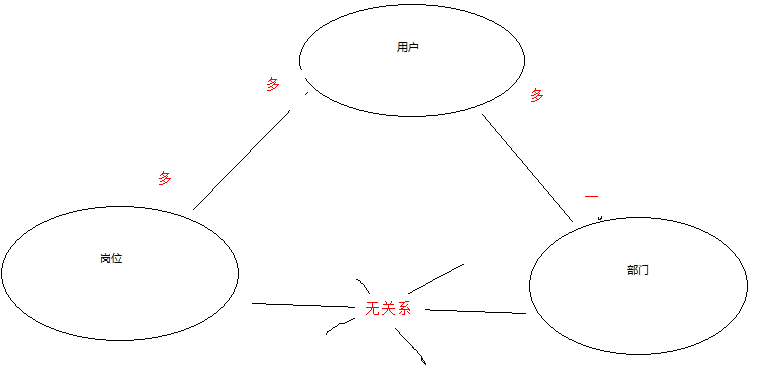
User.hbm.xml

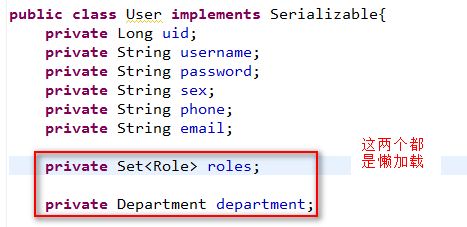
Department.java

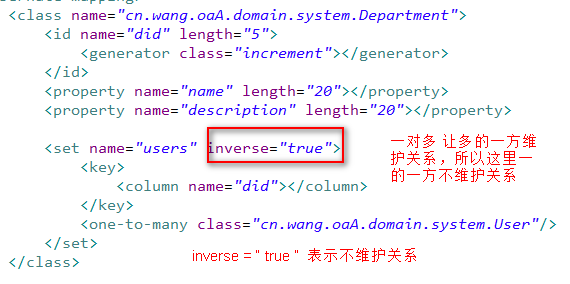
Department.hbm.xml

Role.java

Role.hbm.xml







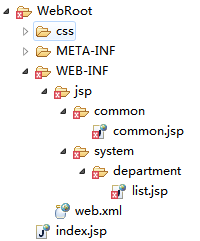
### 4、部门模块

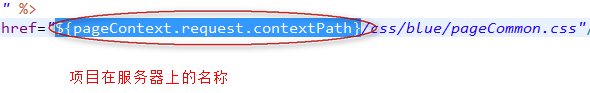
1、写dao和service层的接口和类

2、写action类

3、写struts2的配置文件

4、写jsp页面,所有的jsp页面全部在web-inf下建立（这样不能直接访问必须通过跳转或者转发，如果在 WebRoot下面则可以直接访问这样不安全）





### 列表

在jsp/system/department下创建一个jsp:list.jsp

1、引入公共的jsp文件

<%@ include file="/WEB-INF/jsp/common/common.jsp"%>

2、把样式进行替换

### 添加

1、在添加页面中add.jsp：

在form表单中，必须用struts2的标签（防止乱码问题，在javaweb中我们写过滤器处理乱码，但是struts2标签本身就可以处理乱码）

<s:form action="list.html">

<s:textfield name="name" cssClass="InputStyle"></s:textfield>

</s:form>

2、在后台action代码中：

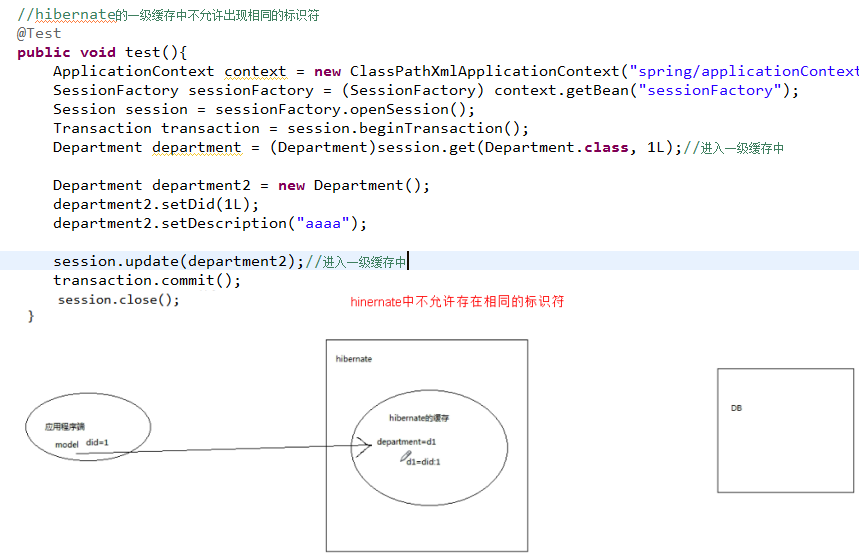
**模型驱动的作用：**

1、获取表单中的数据

2、把数据放在模型驱动中用于回显

一般情况下不要把模型驱动传递到dao层，不要让模型驱动和dao交互





<result name=*"action2action"* type=*"redirectAction"*>departmentAction\_showAllDepartment.action</result>

注意上面的这种struts配置文件的写法

3、在列表页面中：

<s:a action="departmentAction\_delete.action?did="></s:a>

说明：

1、struts2的标签只能跟ognl表达式

2、html标签只能跟el表达式

3、在struts2的标签中：

1、s:property中的value属性

2、s:iterator中的value属性

3、s:select中的value属性

以上这些属性都直接能跟ognl表达式寻找相应的数据

4、s:textfield 与 s:a中的action等都不能直接跟ognl表达式

<s:textfield name="username" value="aa"/>

上述的标签的值为"aa"

<s:textfield name="username" value="%{aa}"/>

上述标签中的值为对象栈中key值为aa的所对应的value值

在字符串中使用ognl表达式要使用 % 号

4、在部门模块中，部门和用户是有关联的

如果在用户表中有一个外键值正好是要删除的部门的主键值，这个时候，删除部门就会报错：主外键的约束

回顾：hibernate在数据库的外键方面的处理：

**1、外键的维护**

一对多

对象与集合的关系

Set集合中 inverse="true/false"

多对多

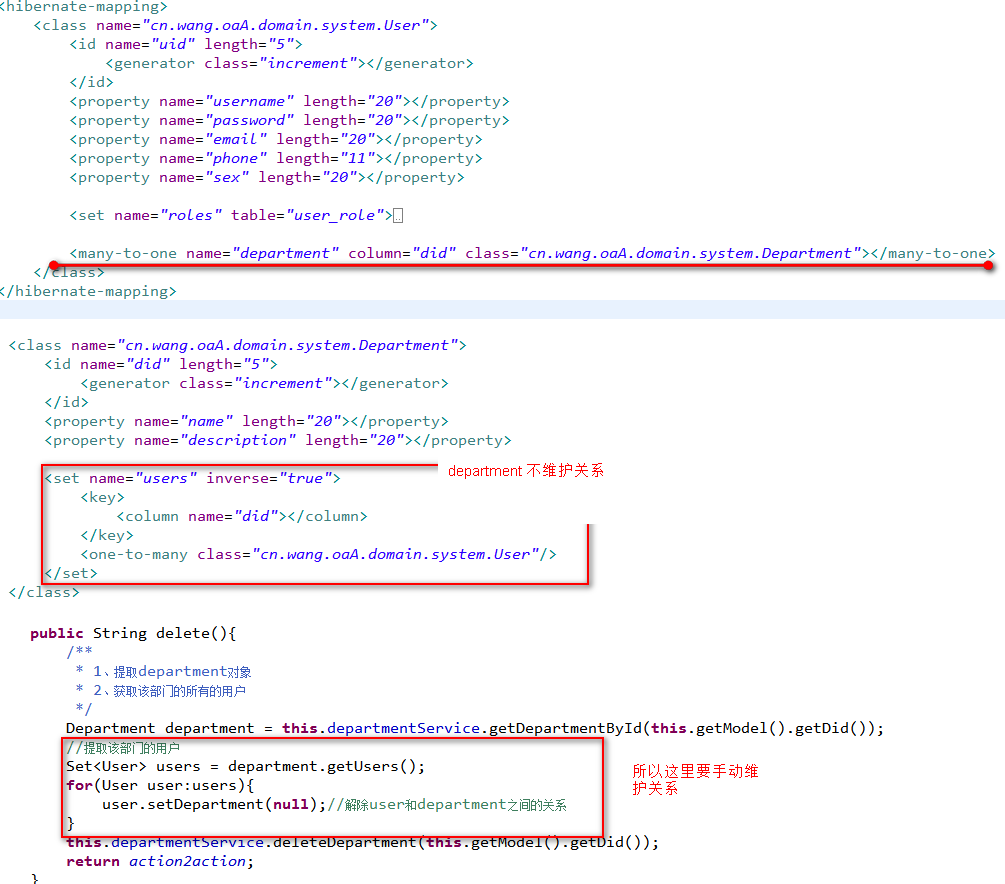
对象与集合的关系

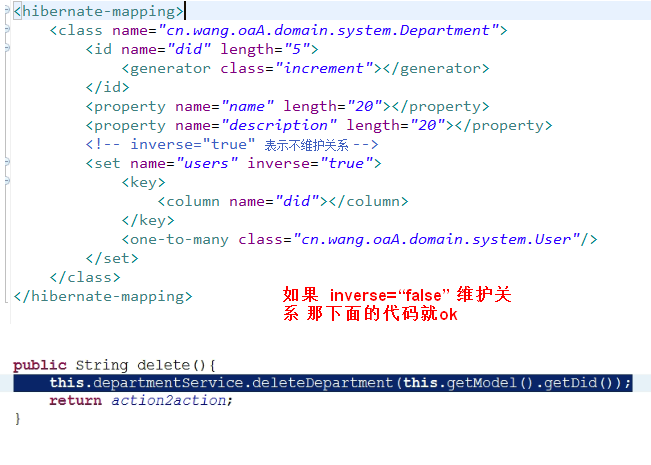
Set集合中 inverse="true/false"

多对一

对象与对象的关系

many-to-one中没有inverse属性





### 删除

删除部门所涉及的细节：

|  |
| --- |
| Collection<Department> departmentList = this.departmentService.getAllDepartment(); //1  Department department = this.departmentService.getDepartmentById(this.getModel().getDid()); //2  Set<User> users = department.getUsers(); //提取该部门的用户 //3  for(User user:users){  user.setDepartment(null);//解除user和department之间的关系 //4  }  this.departmentService.deleteDepartment(this.getModel().getDid()); //5  Hibernate: //1  select  department0\_.did as did1\_,  department0\_.name as name1\_,  department0\_.description as descript3\_1\_  from  Department department0\_  Hibernate: //2  select  department0\_.did as did1\_,  department0\_.name as name1\_,  department0\_.description as descript3\_1\_  from  Department department0\_  where  department0\_.did=?  Hibernate: //3  select  users0\_.did as did1\_1\_,  users0\_.uid as uid1\_,  users0\_.uid as uid4\_0\_,  users0\_.username as username4\_0\_,  users0\_.password as password4\_0\_,  users0\_.email as email4\_0\_,  users0\_.phone as phone4\_0\_,  users0\_.sex as sex4\_0\_,  users0\_.did as did4\_0\_  from  User users0\_  where  users0\_.did=?  Hibernate: //5  select  department0\_.did as did1\_,  department0\_.name as name1\_,  department0\_.description as descript3\_1\_  from  Department department0\_  where  department0\_.did=?  Hibernate: // 4 这里更新了user，为什么要更新user呢？  在删除department的时候会检查department中的所有东西，包括的department的关联对象user，发现对user进行了置空操作，所以就会发出更新user的操作  update  User  set  username=?,  password=?,  email=?,  phone=?,  sex=?,  did=?  where  uid=?  Hibernate: //5  delete  from  Department  where  did=?  Hibernate: //1 删除之后action跳转到action，来到列表页面，又进行了查询  select  department0\_.did as did1\_,  department0\_.name as name1\_,  department0\_.description as descript3\_1\_  from  Department department0\_  session.flush: **session flush在commit之前默认都会执行**  1、检查hibernate一级缓存中所有的持久化状态的对象（一级缓存的意义在于存储持久化对象，是一个缓存区，然后配合session flush的时候检查一级缓存区中的所有持久化对象 ）  2、把对象和快照进行对比，如果没有发生变化，则什么都不做  如果发生变化，则发出update语句  3、可以发生save的操作  4、检查关联对象  看是否维护关系  看是否有级联操作  当执行session.flush方法的时候，hibernate内部会检查一级缓存区中所有的持久化对象  会判断该对象和数据库有没有对应的数据(根据标示符id)，如果没有  则发出insert语句，如果有，则再让该对象和副本进行对比，如果和副本一样，则  什么都不做，如果不一样，则发出update语句  session.flush的时候，只不过是向数据库发送了sql语句，但是值在数据库中并不  存在。因为事务此时还没有提交。  一级缓存：  Person p = new Person();  p.setName("aa");  Person p1 = new Person();  p1.setName("bbbb");  session.save(p1);  当事务提交的时候，p1在session的一级缓存中，p不在一级缓存中  所以当事务提交的时候，只会保存p1,和p没有关系  关于session的产生方式：  1、sessionFactory.openSession  只要调用该api就会创建一个新的session  2、sessionFactory.getCurrentSession  第一次当执行getCurrentSession的时候，会判断当前线程中是否有session,  如果有，则从当前线程中把session提取出来，如果没有  则调用sessionFactory.openSession方法创建一个session,把创建后的session放入到  threadlocal中  这样可以保证在一个线程中始终只用一个session  3、spring容器启动事务用的session从threadlocal中，程序员进行crud操作的session也从threadlocal中获取  所以程序员操作的session和spring容器操作的session是同一个session  **Hibernate 操作数据需要session Spring容器开启事务需要session 这两个session必须是同一个session**  如何保证同一个 ？ 和当前线程绑定在一起。**Hibernate 操作数据 Spring容器开启事务 他俩肯定不在同一个类中，但是肯定在同一个线程中**  **关于hibernate的思考**  **1、是一个面向对象的数据库的操作框架，其低层是利用jdbc完成的**  **2、如果把一个面向对象的操作变成sql语句**  **3、任何一个ormapping框架都应该有缓存**  **缓存有私有数据和共有数据**  **该框架是怎么样保证数据的私有性**  **该框架是怎么样把数据进行公开化的**  **怎么样保证缓存和数据库的同步**  **4、怎么样对事务进行处理**  **5、怎么样去提高数据库的访问效率** |

### 更新

页面中要有一个隐藏域

<s:hidden name=*"did"*></s:hidden>

数据回显

把一个对象放入到对象栈或者map栈中都可以进行回显：

放在对象栈中的属性可以利用struts2标签中的name属性进行回显

放在map栈中的属性可以利用struts2标签中的value属性进行回显

### 注意：增删改 都需要开启事务

@Transactional(readOnly=false)

## **5、岗位模块**

异常处理 详见 **struts2的全局异常处理**

## **6、用户模块**

#### 用户的查询

在页面上显示的数据来自于后台的三张表



user

department

role

后台的hql:

"from User"

该hql语句只是把user提取出来，而user中的部门信息和岗位信息并没有提取出来

在前台：

显示部门名称:

如果在web.xml文件配置了OpenSessionInView模式

<s:iterator value="#userList">

<s:property value="department.name"/>

</s:iterator>

如果没有配置OpenSesionInView模式：

在后台，写hql语句的时候把所有的数据全部提取出来

"from User u left outer join fetch u.department d left outer join fetch u.roles r"

这样的hql语句会把user,department,role中的所有的字段全部提取出来

**UserTest.java**   

**Vo :view object**

UserView:

username

deparmentname

roles

**Bo :business object**

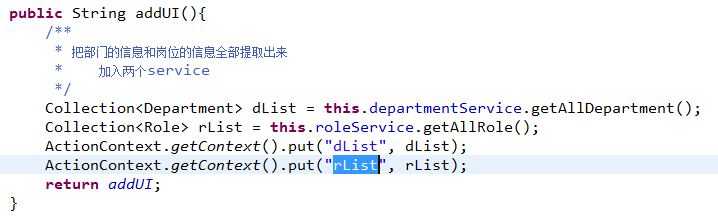
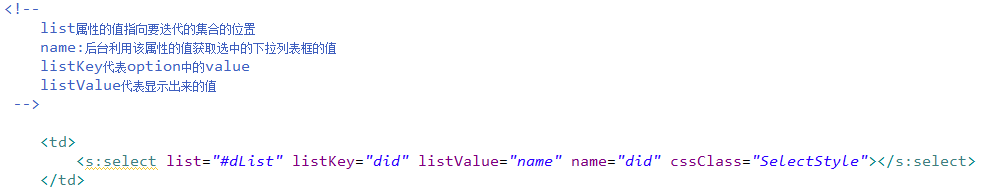
**Pojo :entity object**

如果采用vo和pojo会造成从vo到pojo的转化

**一般采用的技术方式是**

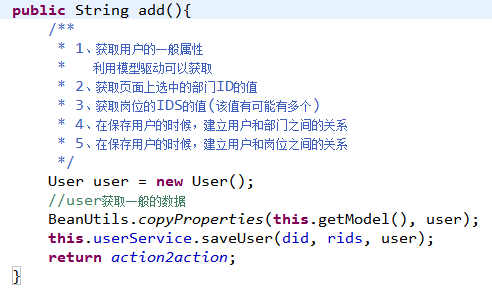
**在web.xml文件中配置OpenSessionInView模式，这样在jsp页面中session也不会关闭**

#### 用户的增加

#### 增加动作

需要同时利用模型驱动和属性驱动



在baseDao中增加一个方法 利用 Serializable[] ids 获取数据

Hibernate元数据的操作



执行的顺序：

**构造器--》hhibernateTemplate注入--》@PostConstruct**



#### 修改显示



Update.Jsp中需要一个 <s:hidden name=*"uid"*></s:hidden>

#### 修改动作

#### 删除动作

因为 user 和department user和 roles 都维护关系，所以直接调用baseDaoImpl的删除直接删除就可以

### jquery删除确认插件



### 8、创建树的步骤

1、在当前页面中导入两个js文件和一个css文件

<link rel="stylesheet" href="zTreeStyle/zTreeStyle.css" type="text/css">

<script type="text/javascript" src="jquery-1.4.2.js"></script>

<script type="text/javascript" src="jquery-ztree-2.5.js"></script>

2、页面上必须有树的容器

<ul id="tree" class="tree" style="width:230px; overflow:auto;"></ul>

说明：class为tree就是树的容器

3、调用zTree方法创建树

3.1

var setting = {

isSimpleData: true, //是否是简单数据格式

简单数据格式

var json = {

a:5,

b:function(){ }

};

复杂数据格式：

var a = {

b:{}

};

treeNodeKey: "id",//设置节点唯一标示名称

treeNodeParentKey: "pId",//设置节点的父节点名称

showLine: true,//节点与节点之间的连线是否显示

root:{

isRoot:true,

nodes:[]

}

};

3.2 设置树上的每一个节点

zNodes =[

{id:'1',pid:'0',name:'上海5期java就业班',icon:'user.gif',isParent:true},//是否为父节点

{id:'2',pid:'1',name:'头号带头大哥：老毕'},

{id:'3',pid:'1',name:'班上所有的同学'},

{id:'4',pid:'3',name:'头号同学：班长'},

{id:'5',pid:'4',name:'班秘：林志玲'}

];

3.3

调用zTree方法创建树

zTree = $("#tree").zTree(setting, zNodes);

### 9、Struts2结果集 与 ajax整合原理

1、在页面上

$.post请求一个url

$.post("menuitemAction\_showAllMenuitem.action",null,function(data){});

2、在MenuitemAction.java中

public class MenuitemAction implements BaseAction<Menuitem>{

private MenuitemService menuitemService;

//get和set方法

public String showAllMenuitem(){

Collection<Menuitem> menuitemList = this.menuitemSerice.getAllMenuitem();

ActionContext.getContext().getValueStack().push(menuitemList);//放到栈顶

return "listAction"

}

}

3、在struts2的配置文件中进行配置

<package name="menuitem" namespace="/" extends="struts-default">

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

<result></result>//这里面是不能写东西的

</action>

</package>

**利用struts2-json-plugin.jar包**

### 10、struts2自定义结果集

**1、有一个关于结果集的总的接口**

public interface Result extends Serializable {

public void execute(ActionInvocation invocation) throws Exception;

}

**2、抽象类**

StrutsResultSupport implements Result{

public void execute(ActionInvocation invocation) throws Exception {

lastFinalLocation = conditionalParse(location, invocation);**//获取result标签中间的内容**

doExecute(lastFinalLocation, invocation);

}

protected abstract void doExecute(String finalLocation, ActionInvocation invocation) throws Exception;

}

**3、具体的结果集类**

ServletDispatcherResult extends StrutsResultSupport{

doExecute(String finalLocation, ActionInvocation invocation) throws Exception{

dispatcher.forward(request, response);

}

}

ServletRedirectResult extends StrutsResultSupport{

doExecute(String finalLocation, ActionInvocation invocation) throws Exception{

sendRedirect(response, finalLocation);

}

}

4、转发和重定向都需要reault标签中的内容，如果是ajax请求name是不需要获取result标签中的内容的



5、可以自定义一个结果集JSONResult

1、 该类可以继承抽象类（目的是获取result标签中的内容），也可以直接实现接口（ajax不需要result标签中的内容）

是继承抽象类还是实现接口关键看该类是否要获取result标签中的内容

2、在struts2的配置文件中

<result-types>

<result-type name="json" class="org.apache.struts2.json.JSONResult"/>

</result-types>

上面的配置相当于声明了一个结果集类型

只要在result标签中的type属性中写"json"就可以使用该结果集了

### 11、把一个树从数据库中展现到页面中 tree.html

步骤

1、建立持久化和映射文件 MenuitemTest.java 初始化数据



2、在dao层和service层建立相应的类和接口

3、建立action类

@Controller("menuitemAction")

@Scope("prototype")

public class MenuitemAction extends BaseAction<Menuitem>{

@Resource(name="menuitemService")

private MenuitemService menuitemService;

public String showAllMenuitem(){

Collection<Menuitem> menuitems = this.menuitemService.getAllMenuitem();

**ActionContext.getContext().getValueStack().push(menuitems);**

return SUCCESS;

}

}

4、配置文件

<struts>

<package name="menuitem" namespace="/" extends="json-default">

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

<result type="json"></result>

</action>

</package>

</struts>

5、创建一个tree.html

1、引入一个css文件三个js文件

<link rel="stylesheet" href="zTreeStyle/zTreeStyle.css" type="text/css">

<script type="text/javascript" src="js/jquery-1.4.2.js"></script>

<script type="text/javascript" src="js/jquery-ztree-2.5.js"></script>

2、必须具有树的容器

<div>

<ul id="tree" class="tree" style="width:230px; overflow:auto;"></ul>

</div>

3、在js中

var tree = {

setting:{

isSimpleData: true,

treeNodeKey: "mid",

treeNodeParentKey: "pid",

showLine: true,

root:{

isRoot:true,

nodes:[]

}

},

loadTree:function(){

$.post("menuitemAction\_showAllMenuitem.action",null,function(data){

//该data数据是从对象栈的栈顶提取过来的

$("#tree").zTree(tree.setting,data);

});

}

};

$().ready(function(){

tree.loadTree();

});

### Menuitem相关的文件只是演示树型菜单与具体的项目并没有关系

### 8-----11只是演示树型菜单与具体的项目并没有关系

## 权限管理

1、通用的解释：

用户、角色、权限

2、权限

1、所有的请求资源url的控制 <http://localhost:8080/shoa/departmentAction_saveDepartment.action> **是否有该url的权限**

2、请求DeartmentAction中的showAllDepartment方法 **是否有该方法的权限**

3、页面中的内容

1、菜单 （不同的用户登录系统以后应该显示不同的菜单）

2、功能 （添加 删除 修改 查询）

4、数据

3、角色（岗位）

角色是权限的一个集合

角色在一般情况下都是公司中的一个部门中的一个职称：例如：咨询员、咨询主管

权限是角色下面的一个子元素

一般的公司的分法：

1、公司

2、部门 角色

3、员工

比较大的公司的分法：

1、集团

组织(分公司)

部门

员工

4、用户、角色、权限的关系

用户对角色 多对多 一个用户包含很多角色，一个角色属于很多用户

角色对权限 多对多 一个角色包含很多权限，一个权限属于很多角色

5、建立持久化类和映射文件

User.java

User.hbm.xml

Role.java

角色也要用tree来表示，所以要有一个pid属性表示父id

Role.hbm.xml

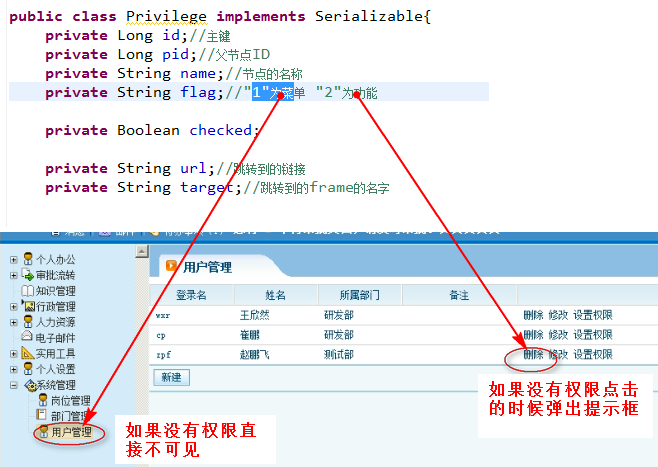
Privilege.java

这个实体类其实包含两部分权限 菜单项 和功能项

Privilege.hbm.xml

1. 根据持久化类和映射文件生成数据库

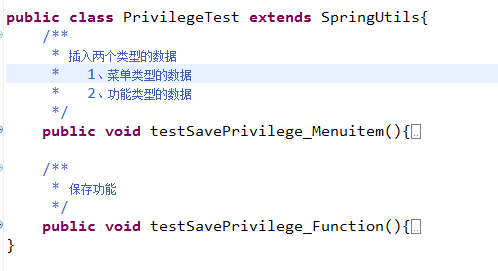
对菜单和功能的限制是不一样的



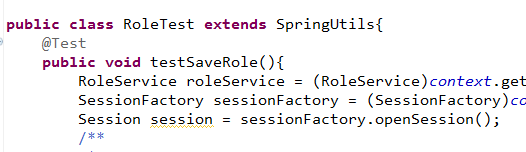


7、通过junit的方式给privilege表中添加数据

在PrivilegeTest类中准备权限数据和功能数据



1. 准备角色的数据



1. 用户设置角色 jsp/system/user/list.jsp *user\_role.js struts2-role.xml RoleAction.java*

|  |
| --- |
| 1、目标：就是为了建立用户和角色之间的关系  2、功能点：  1、把隐藏的div显示出来  2、动态的显示用户的名称  3、全选复选框按钮的功能  4、把一个用户能够访问到的角色进行回显  5、建立用户和角色之间的关系  3、js的架构  抽象：  1、在js控制页面的过程中，数据的传递是一个方面的内容 data  2、在js控制页面的过程中，涉及到了事件的触发 init  3、在点击触发事件的时候，执行所有的页面的控制工作 option  var role = {  //专门用来存放数据的  data:{ },  init:{  initData:function(){ }, //初始化数据  initEvent:function(){} //初始化事件  },  //页面的控制操作  option:{  divOpt:{ }, //涉及到div的操作  userOpt:{ }, //涉及到的用户的操作  roleTree:{} //涉及到角色树的操作  **public** **class** RoleAction  **public** String showRoleTree(){}  }  };  4、点击设置角色的功能：  1、显示div  role  option  //涉及到div的操作  divOpt: {  showDiv:function(){  /\*\*  \* 把所有的隐藏的div显示出来  \*/  $("div:hidden").show();  }  }  2、给user中的username和uid赋值  5、说明：  应该注意的内容：  1、当点击设置角色，角色树没有加载出来的时候，设置全选复选框为不可用的状态  2、当点击设置角色，角色树没有加载出来的时候，在角色树的位置应该显示loading.gif,等待图片这样交互效果比较好  以上两点都是初始化的状态  3、当角色树加载出来以后，全选复选框  全选复选框的全选与全不选的功能是由zTree 的ajax执行过后会返回一个对象zTreePlugin来实现的  role.option.roleTree.zTreePlugin = $("#roleTree").zTree(role.option.roleTree.setting,data);  在zTree 的ajax执行过后会返回一个对象zTreePlugin  对象zTreePlugin 有一个方法是checkAllNodes()      A、设置为可用的状态  B、根据角色树上的所有的角色选复选框是否被全部选中，来决定全选复选框的状态：选中还是未选中  **var** uncheckedNodes = role.option.roleTree.zTreePlugin.getCheckedNodes(**false**);得到树中没有被选中的节点    4、当角色树加载出来以后，隐藏loading,显示角色树  5、当点击角色树上的某一个复选框的时候，判断  角色树上的复选框是否被全部选中从而改变全选复选框的状态值 ztree自己的事件  setting: {  isSimpleData: **true**,  treeNodeKey: "rid",  treeNodeParentKey: "pid",  showLine: **true**,  root: {  isRoot: **true**,  nodes: []  },  /\*\* 显示树上的复选框 \*/  checkable:**true**,  /\*\* 给角色树添加事件 change事件\*/  callback:{  change:**function**(){  /\*\* 调用setAllChecked方法设置全选复选框的状态 \*/  role.option.roleTree.setAllChecked();  }  }  },   1. 保存动作： 既可以写在UserAction中也可以写在RoleAction中   这里写在RoleAction中 roleAction\_saveRole.action  在 RoleAction中添加  **private** Long uid;  **private** String checkedStr;  并添加set get方法  /\*\*  \* 保存动作  \* 建立用户和角色之间的关系  \*/  **public** String saveRole(){  User user = **this**.userService.getUserById(**this**.uid);  Set<Role> roles = **this**.roleService.getRolesByIds(**this**.checkedStr);  **if**(roles.isEmpty()){  user.setRoles(**null**);  }**else**  {  user.setRoles(roles);  }  **this**.userService.updateUser(user);  **return** *SUCCESS*;  }  7、数据回显    **先把所有的角色查找出来，然后把自己有的角色查找出来，在所有角色中将自己有的角色的checked属性变成true**  给Role.java添加checked属性并提供set get 方法 |

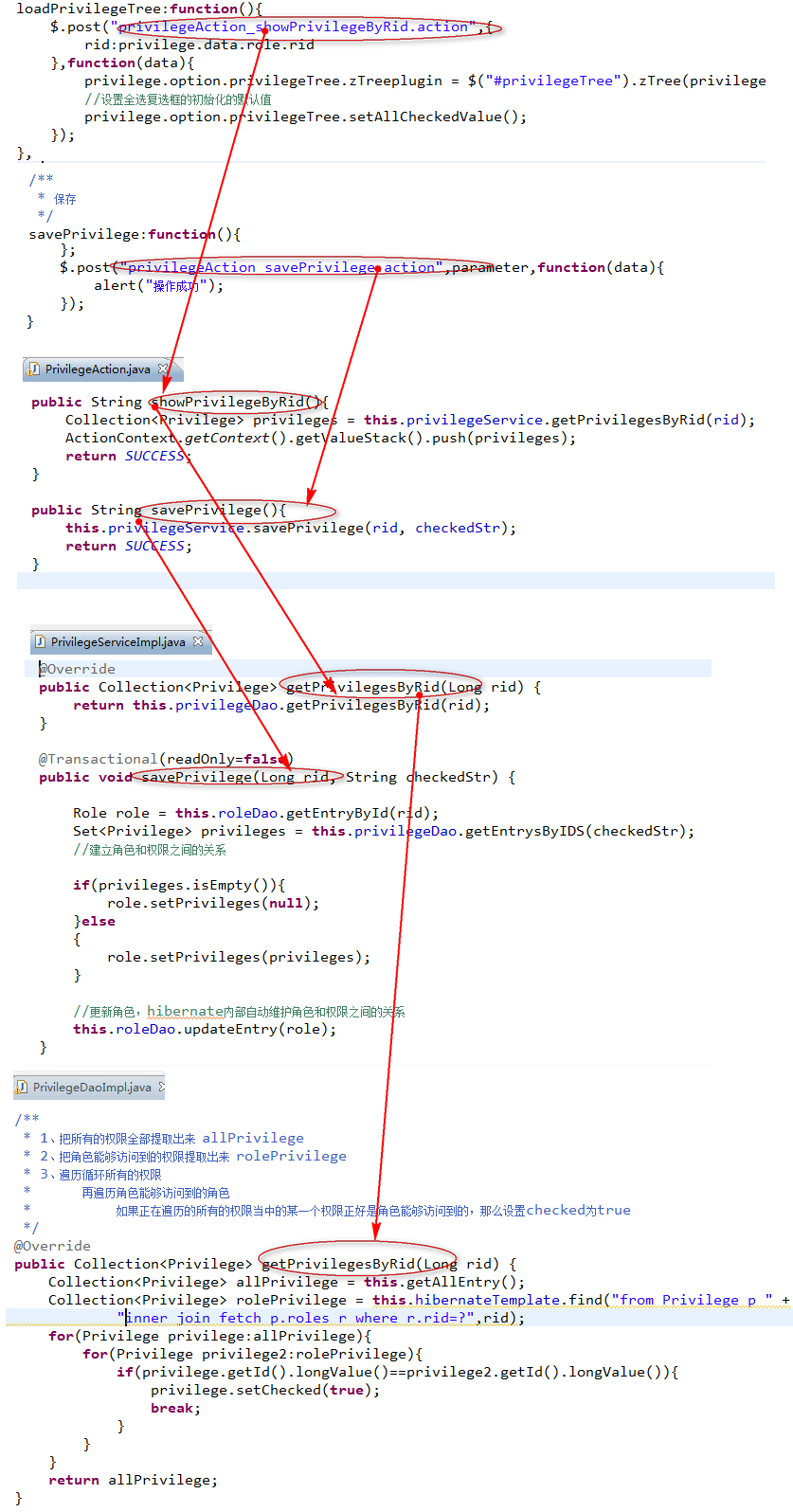
1. 角色设置权限

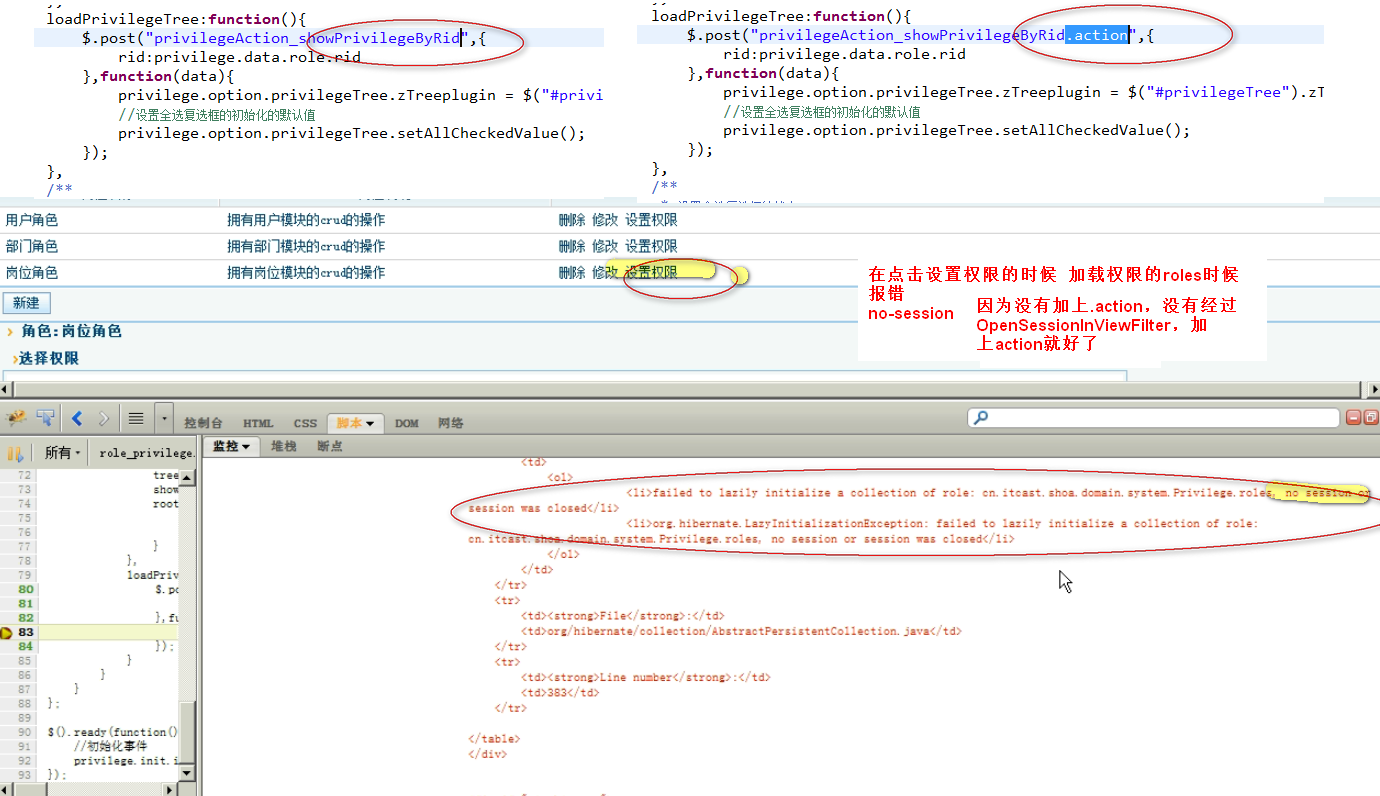
Role\_privilege.js

PrivilegeDaoImpl

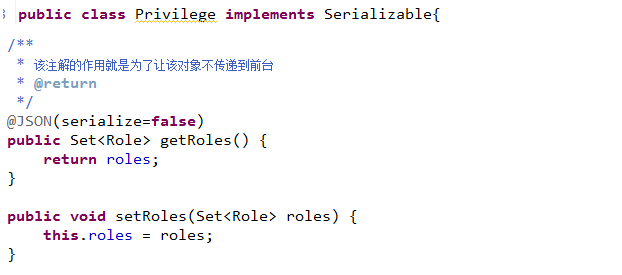
privilegeServiceImpl

PrivilegeAction





**如何让它不加载 roles？**



10、组件树开发 left.jsp使用一个通用的tree



|  |
| --- |
| 1、做一个通用的树    利用jQuery的插件机制做一个树出来  (function($){  $.fn.TreePanel = {  defaults: {  url:'',  data:null,  id:'',  setting: {  isSimpleData: true,  treeNodeKey: "id",  treeNodeParentKey: "pid",  showLine: true,  root: {  isRoot: true,  nodes: []  }  }  },  /\*\*  \* config = {  \* url  \* data  \* setting:{  \* treeNodeKey: "rid"  \* }  \* }  \* @param {Object} config  \*/  createTree: function(config){  var con = {};  /\*\*  \* 用户传s递过来的配置覆盖掉默认的配置  \* true为深度覆盖  \*/  con = $.extend(true,$.fn.TreePanel.defaults,config);  $.post(config.url, config.data, function(data){  $("#" + config.id).zTree(con.setting, data);  })  }  };  })($);  2、声明命名空间  (function($){  /\*\*  \* @param {Object} namespace  \* "cn.itcast.oa.TreePanel"  \*/  $.nameSpace = function(namespace){  /\*\*  \* ss[]={"cn","itcast","oa","TreePanel"}  \*/  var ss = namespace.split(".");  var tempNs = [];  /\*\*  \* 当第一遍循环  \* window.cn  \* 当第二遍循环  \* window.cn.itcast  \* ......  \*/  for (var i = 0; i < ss.length; i++) {  tempNs.push(ss[i]);  var n = tempNs.join(".");  if (typeof window[n] != "object") {  eval("window." + n + "={}");  }  }  }  })($);  $.nameSpace("cn.itcast.sh05.oa.PrivilegeTree");  说明在window的基础上加了window.cn.itcast.sh05.oa.PrivilegeTree  3、$.extend(cn.itcast.sh05.oa.PrivilegeTree,$.fn.TreePanel);  把$.fn.TreePanel的内容赋值给cn.itcast.sh05.oa.PrivilegeTree  4、PrivilegeTree调用createTree方法创建树 |

11、权限的控制

粗粒度的权限控制

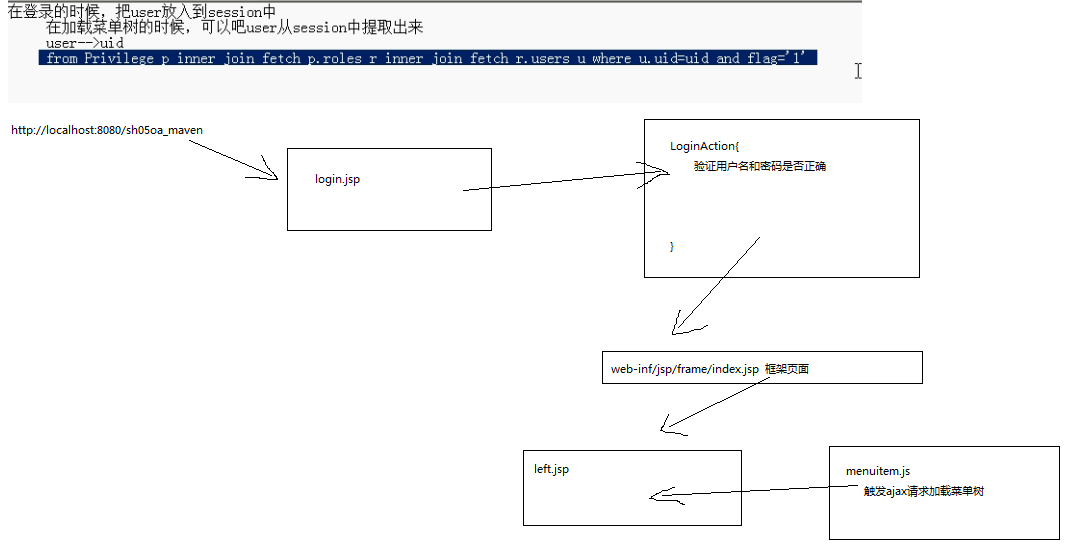
没有经过登录页面是没有办法访问主界面的

细粒度的权限控制

1、不同的用户登录系统以后，看到的左侧的菜单是不一样的

2、当点击用户管理的时候，功能是用户查询，检查当前用户是否有用户查询的权限 如果有，则查询，如果没有，则显示权限不足

12、 1、不同的用户登录系统以后，看到的左侧的菜单是不一样的



1. Login.jsp登录页面
2. LoginDaoImpl BaseDaoImpl 注意参数的类型是**final**

|  |
| --- |
| /\*\*  \* final String... objects 可变参数  \*/  @Override  **public** E getEntryByCondition(final String hql,final String... objects) {  **return** **this**.hibernateTemplate.execute(  **new** HibernateCallback<E>() {  @Override  **public** E doInHibernate(Session session) **throws** HibernateException,  SQLException {  Query query = session.createQuery(hql);  **int** index = 0;  **for**(String s:objects){  query.setParameter(index, s);  index++;  }  **return** (E)query.uniqueResult();  }  }  );  } |
|  |

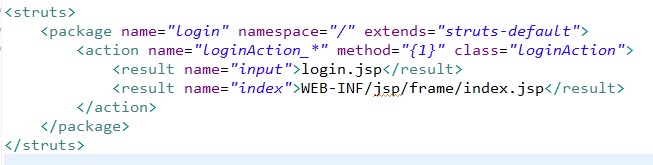
1. LoginServiceImpl

|  |
| --- |
| @Override  **public** User login(String username, String password) {    **return** **this**.loginDao.getEntryByCondition("from User where username=? and password=?", username,password);  } |

1. LoginAction

|  |
| --- |
| **public** String login(){  User user = **this**.loginService.login(**this**.getModel().getUsername(), **this**.getModel().getPassword());  **if**(user==**null**){//用户名或者密码不正确  **this**.addActionMessage("用户名或者密码错误");//这个方法就是将错误信息放到集合中然后放到当前的对象栈中  //用户名或者密码错误是个表单级别的错误不需要定位到具体的文本框  **return** "input";  }**else**{//用户名和密码正确  OAUtils.*putUserToSession*(user);//把 user放入到session中  Collection<Privilege> privileges = **this**.privilegeService.getFunctionsByUid(user.getUid());  //该用户能够访问到的功能权限放入到了session中  OAUtils.*putFunctionsToSession*(privileges);  **return** "index";  }  } |

1. Struts2-login.xml





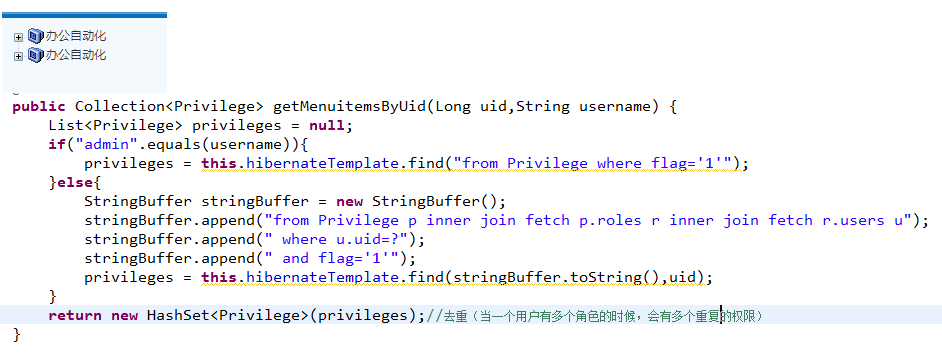
登录成功后来到/frame/index.jsp页面这是一个框架页面他需要再次加载其他的页面，所以需要一个专门用于转发的Action

FowardAction

Left.jsp页面发送加载角色树的ajax

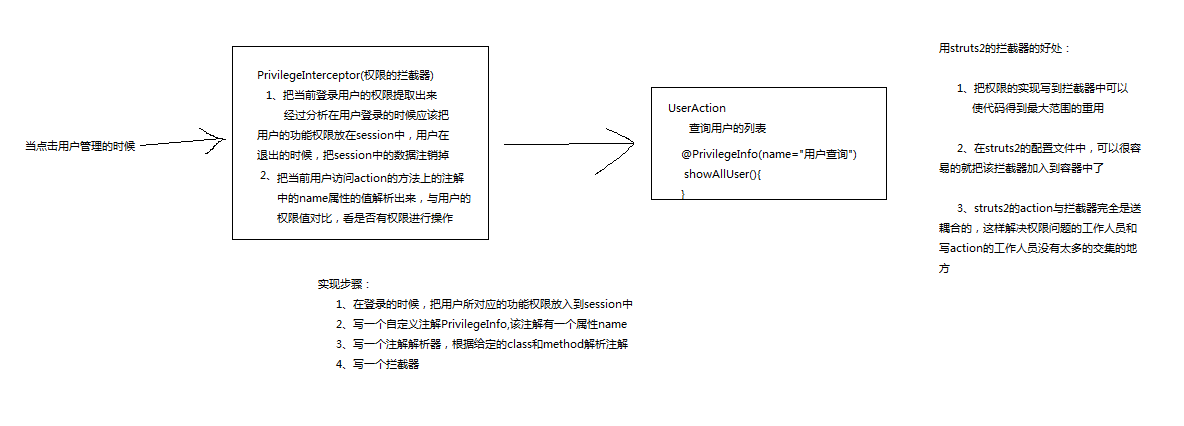
权限分为两部分 权限菜单和权限功能 现在加载的都是权限菜单

任何项目都要有一个超级管理员 拥有所有的权限



12、 2、当点击用户管理的时候，功能是用户查询，检查当前用户是否有用户查询的权限 如果有，则查询，如果没有，则显示权限不足**（注解的形式）**

|  |
| --- |
| 注解：  一、概念  用来标注一些内容的  二、注意事项  1、注解不能单独存在  2、注解的生命周期有三个  source 这个注解只能在java文件中起作用  class 这个注解能在source,class文件中起作用  runtime 该注解能在source,class,runtime时都起作用  3、注解能够标注在什么地方  public enum ElementType {  TYPE,  FIELD,  METHOD,  PARAMETER,  CONSTRUCTOR,  LOCAL\_VARIABLE,  ANNOTATION\_TYPE,  PACKAGE  }  4、每一个注解都有注解解析器  该注解要干嘛，在注解解析器中实现  三、自定义注解 |



我们不能将菜单权限放到session中（session不能乱用会导致内存泄漏）而且只是在登录成功后加载一次所以没有必要使用session，

但是可以将功能菜单放到session中。因为用户每次点击超链接都要把该用户的功能权限提取出来做对比判断是否有当前方法的权限，这叫一次会话

步骤：

1. 在用户登录的时候得到用户的功能权限放到session中 **LoginAction.java**
2. 自定义一个功能权限的注解和注解解析器
3. 案例 userAction中的 showAllUser() 方法
4. <package name=*"user"* namespace=*"/"* extends=*"privilegeOA"*><!-- struts-default -->
5. 通过注解解析器获得该方法上的 @PrivilegeInfo(name="用户查询") name属性值 “用户查询”
6. 自定义拦截器 **PrivilegeInterceptor.java (用户的每一次点击超链接都会经过该拦截器)**

\* 1、获取session中的用户能够访问的权限

\* 2、获取当前访问action中的方法的权限

\* 3、检查该用户是否具有action中的方法的访问权限 前提条件：

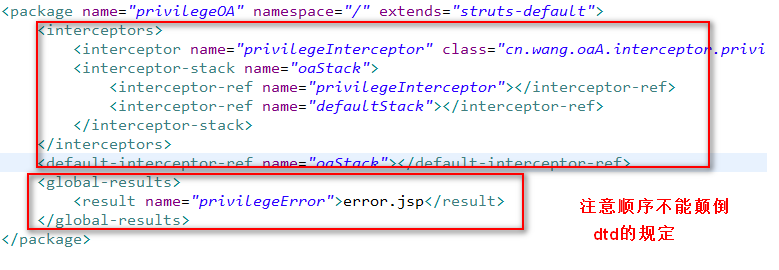
\* 如果用户名称为"admin",则放行

\* 如果有 则访问action

\* 如果没有则跳转到错误处理页面

1. 注册拦截器 在struts.xml文件中<package name=*"privilegeOA"* namespace=*"/"* extends=*"struts-default"*>





12、 2、当点击用户管理的时候，功能是用户查询，检查当前用户是否有用户查询的权限 如果有，则查询，如果没有，则显示权限不足**（XML的形式）**



## jbpm

**JBPM关联源码后api方法的参数会显示参数的类型和参数的名称**

1、概念

1、在一个超级复杂的业务环境中，利用流程控制框架可以对流程进行特别精准的控制

2、在流程控制中，会产生很多数据，这些数据将是领导决策的依据

2、执行工作流的步骤：

1、制定工作流的各个环节

2、确定工作流生效

3、启动流程实例

4、员工请假

5、领导审批

3、工作流中有四个概念

bpm: business process managerment 商业流程管理

jbpm:jboss business process managerment jboss公司实现商业流程管理

pdl:process definition language 流程定义语言

jpdl:jboss process definition language jboss公司流程定义语言

4、工作流的历史

jbpm3--->jbpm4--->jbpm5 (jboss公司)

activiti5 (jbpm原来团队的内容)

5、 jbpm4的数据库的操作部分是hibernate做的。如果数据库的数据特别多的情况下不适合用jbpm4

activiti5 的数据库的操作部分hibernate 和mybatis都可以做

**myeclipse安装jpbm插件：**

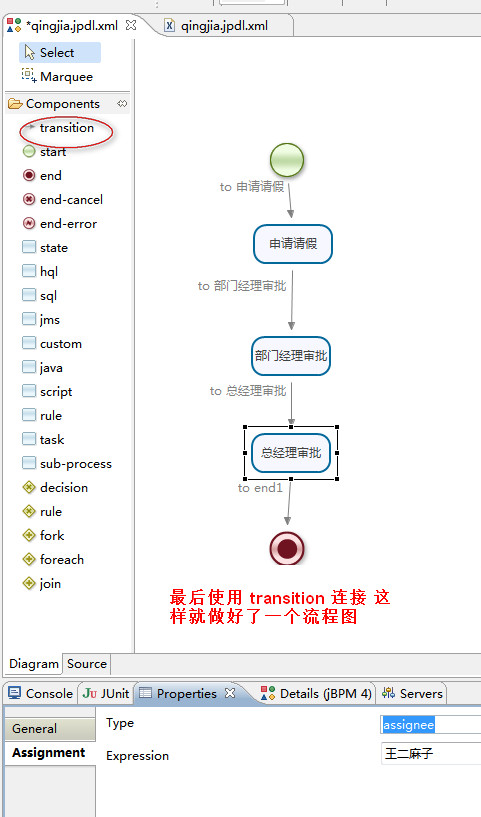
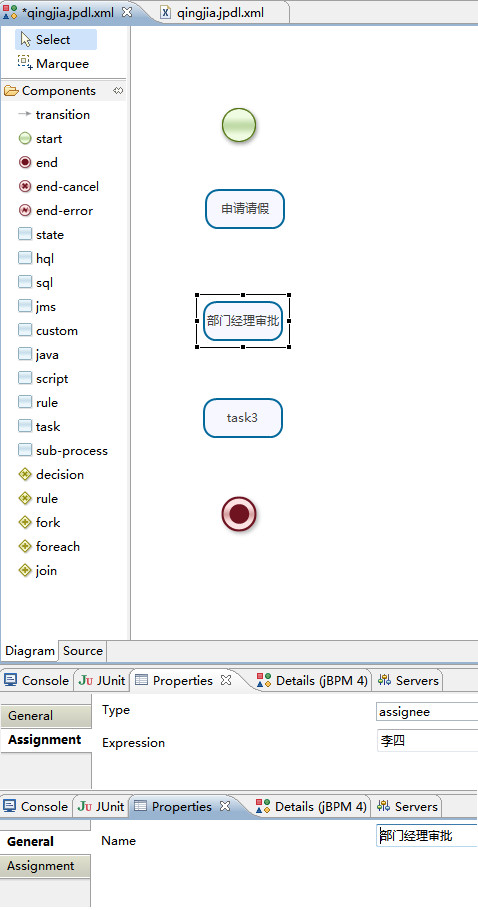
<https://jingyan.baidu.com/article/60ccbceb597c0564cab197f8.html>

**工作流的具体步骤：**

1、在eclipse中安装jbpm的插件

2、制定流程图

制定的请假流程图



说明： 申请请假是张三的任务

部门经理审批是李四的任务

总经理审批是王二麻子的任务

3、发布到jbpm的引擎中:部署的过程

4、启动流程实例

5、张三完成请假申请(自动离开开始节点，此时李四和王二麻子都看不到任务)

6、李四完成审批任务

7、王二麻子完成审批任务

注意：

1、当在"部门经理审批"的节点的时候，以总经理身份登入系统以后是看不到该任务的

2、在工作流的过程中会产生大量的数据

**环境搭建：**

1、在src

jbpm.cfg.xml文件

<jbpm-configuration>

<import resource="jbpm.default.cfg.xml" />

<import resource="jbpm.businesscalendar.cfg.xml" />

1、在jbpm内部是通过控制数据库的表来控制流程的

2、在jbpm内部是通过hibernate操作数据库的

3、默认情况下，事务是由hibernate处理的

<import resource="jbpm.tx.hibernate.cfg.xml" />

<import resource="jbpm.jpdl.cfg.xml" />

<import resource="jbpm.bpmn.cfg.xml" />

<import resource="jbpm.identity.cfg.xml" />

<!-- Job executor is excluded for running the example test cases. -->

<!-- To enable timers and messages in production use, this should be included. -->

<!--

<import resource="jbpm.jobexecutor.cfg.xml" />

-->

</jbpm-configuration>

2、在src

jbpm.hibernate.cfg.xml文件

<hibernate-configuration>

<session-factory>

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

<property name="hibernate.connection.driver\_class">org.hsqldb.jdbcDriver</property>

<property name="hibernate.connection.url">jdbc:hsqldb:mem:.</property>

<property name="hibernate.connection.username">sa</property>

<property name="hibernate.connection.password"></property>

<property name="hibernate.hbm2ddl.auto">create-drop</property>

<property name="hibernate.format\_sql">true</property>

//所有的映射文件在jbpm.jar包的根目录下能够找到

<mapping resource="jbpm.repository.hbm.xml" />

<mapping resource="jbpm.execution.hbm.xml" />

<mapping resource="jbpm.history.hbm.xml" />

<mapping resource="jbpm.task.hbm.xml" />

<mapping resource="jbpm.identity.hbm.xml" />

</session-factory>

</hibernate-configuration>

将hibernate数据库的连接换成自己的 并且使用mysql数据库

3、在srclogging.properties



3、在项目中创建lib文件夹，放入相应的jar包

a、 jbpm.jar

b、 H:\java\视频\额外视频\赵栋--OA(有代码)--A\day09\jbpm-4.4\jbpm-4.4\lib 文件夹下的所有的\*.jar包

4、创建数据库并且生成表

@Test

public void testCreateTable(){

Configuration configuration = new Configuration();

configuration.configure("jbpm.hibernate.cfg.xml");

configuration.buildSessionFactory();

}

**流程定义**

1、部署流程定义的代码

@Test

public void testDeploy(){

/\*\*获得jbpm的流程引擎 \*/

ProcessEngine processEngine = Configuration.getProcessEngine();

processEngine.getRepositoryService()

.createDeployment()

.addResourceFromClasspath("qingjia.jpdl.xml")

.addResourceFromClasspath("qingjia.png")

.deploy();

}

2、部署流程定义的时候涉及到的数据库的表

**JBPM4\_DEPLOYMENT**

1、含义

部署表，是用来描述一次部署的，把流程定义的一个新的版本进行发布

2、字段

DBID\_ : 是**主键** 也是**部署ID（**deploymentID**）**



**JBPM4\_DEPLOYPROP**

1、含义

部署属性表，是用来描述一次部署里的各种属性的

2、说明

1、部署一次生成4行记录

2、生成的4行记录 **KEY\_** 的具体解释

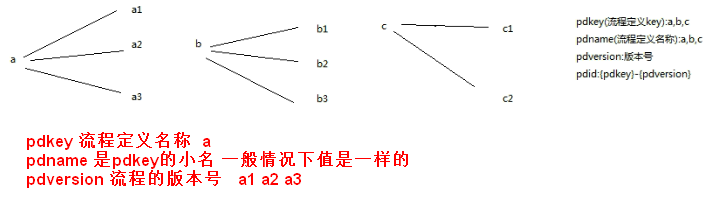
langid 流程定义语言的版本 jpdl-4.4 如果不换框架，该值保持不变

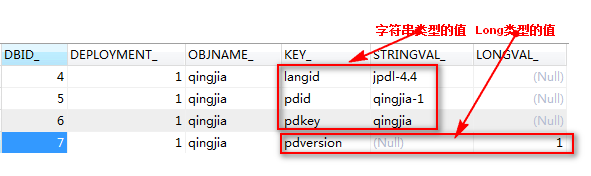
pdid: {pdkey}-{pdversion} 可以确定唯一的

pdkey 流程定义名称

pdversion 版本号 如果流程定义名称不变，每部署一次，版本号自动加1 如果流程定义名称改变，版本号从1开始计算

**理解 pdid 和pdkey pdversion之间的关系：**





3、deploymentID（**JBPM4\_DEPLOYMENT**的DBID\_）**和pdid是一一对应的关系**

**JBPM4\_LOB**

1、含义

部署的时候，xml文件和png图片全部存放在该表中

2、字段

DBID\_：主键

DEPLOYMENT\_:**部署ID（**deploymentID**）**

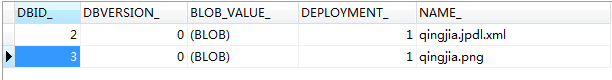
说明：

1、因为只要流程图进行修改，必须发布，在jbpm4\_deployment中必须增加一行数据

所以deploymentID和唯一的流程图是一一对应的

2、如果想查询流程图，必须找到**deloymentId**

name\_:文件的路径



3、流程定义部署的方式

1、Classpath

2、inputstream

3、zipinputstream

4、查看流程定义图片

/\*\*

\* 根据deploymentId查看流程定义图片

\*/

@Test

public void testShowImage() throws Exception{

ProcessEngine processEngine = Configuration.getProcessEngine();

InputStream inputStream = processEngine.getRepositoryService()

.getResourceAsStream("1","qingjia.png");

OutputStream outputStream = new FileOutputStream("d:/processimg.png");

for(int b=-1;(b=inputStream.read())!=-1;){

outputStream.write(b);

}

inputStream.close();

outputStream.close();

}

说明：给定一个deploymentId和一个资源的名称，得到一个InputStream,输出到磁盘上的某一个位置

5、查询部署和流程定义

a、查询流程部署 .createDeploymentQuery()

查询所有的流程部署

根据deploymentID查询流程部署（一个）

b、查询流程定义 .createProcessDefinitionQuery()

查询所有的流程定义

根据deploymentId查询流程定义

根据pdid查询流程定义

根据pdkey查询流程定义

6、流程定义的删除 流程部署的删除

只删除流程部署

void deleteDeployment(String deploymentId);

所有的流程部署的内容全部删除，包括流程实例等

void deleteDeploymentCascade(String deploymentId);

//根据pdkey删除所有的该pdkey下的流程定义

/\*\*

\* 给定一个pdkey,删除整个流程定义

\* 1、根据pdkey查询出所有的流程定义

\* 2、遍历给pdkey下的所有的流程定义，从每一个流程定义中把deploymentID提取出来

\* 3、根据deploymentID删除流程定义

\*/

@Test

**public** **void** testDeletePD(){

ProcessEngine processEngine = Configuration.*getProcessEngine*();

/\*\*

\* 1、根据pdkey获取所有的流程定义

\*/

List<ProcessDefinition> pdList = processEngine.getRepositoryService()

.createProcessDefinitionQuery()

.processDefinitionKey("qingjia")

.list();

/\*\*

\* 2、遍历给pdkey下的所有的流程定义，从每一个流程定义中把deploymentID提取出来

\* 3、根据deploymentID删除流程定义

\*/

**for**(ProcessDefinition pd:pdList){

processEngine.getRepositoryService()

.deleteDeploymentCascade(pd.getDeploymentId());

}

}

**流程实例**

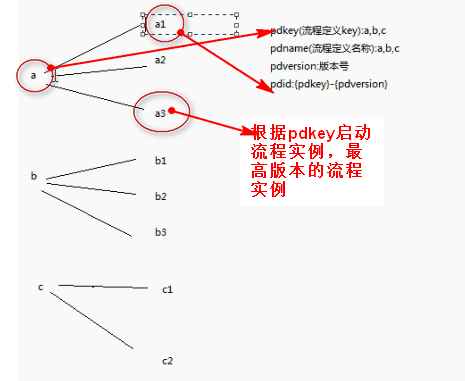
1、概念

在流程图部署完以后，在发起请假申请之前，必须启动流程实例

2、启动流程实例

1、根据pdid启动流程实例

2、根据pdkey启动流程实例，启动的是最高版本的



3、启动流程实例的时候涉及到的表

**JBPM4\_EXECUTION**

1、定义： 该表中存储了当前正在执行的流程实例（每个人请一次假就要新建一个流程实例）

2、字段

DBID\_：主键

**ID\_:流程实例ID（piid）**

activityname\_: 该流程实例正在活动的节点名称(在jBPM中有很多类型的节点，task任务节点只是其中的一种)

procdefid\_: **流程定义ID（pdid）**

**JBPM4\_TASK**

1、定义：该表中存储了当前正在执行的任务

2、字段

DBID\_: 代表主键 和 任务ID（taskid）

NAME\_:任务节点的名称

ASSIGNEE\_: 任务的执行人

EXECUTION\_ID\_:**流程实例ID（piid）**

**JBPM4\_HIST\_PROCINST**

1、定义：该表中存储了历史流程实例，当前正在执行的流程实例在历史信息里也有值，但是结束时间为null,状态(state)为active

**JBPM4\_HIST\_TASK**

1、定义：该表中存储了历史任务,当前正在执行的任务也在该表中，但是结束时间为null,状态(state)也为null

**JBPM4\_HIST\_ACTINST**

1、定义 ：该表中存储了历史节点(任务节点只是节点中的最常用的一种)

4、任务的查询

1、查询正在执行的任务

1、查询当前正在执行的所有的任务

2、根据当前的登录人查询该登录人正在执行的任务(重要)

3、根据流程实例ID查询当前正在执行的任务(重要)

4、根据pdid查询当前正在执行的任务

2、查询已经完成的任务

3、关系

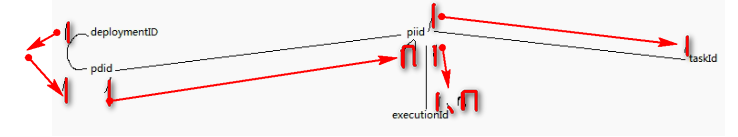
一个流程定义对应多个流程实例

一个流程实例对应一个正在执行的任务id

一份流程定义对应多个正在执行的任务id

在并发情况下一个 piid 对应多个 executionId

在没有并发情况下一个 piid 对应1个 executionId



5、完成任务

processEngine.getTaskService().completeTask("70002");

只要有一个taskID就可以完成任务

/\*\*

\* 完成任务

\* 当一个任务完成以后，该任务在jbpm4\_task表中会删除，在相应的历史表中的state值为"completed"

\* **当完成了最后一个任务以后，流程实例结束了，正在执行的流程实例在jbpm4\_execution表中删除了在jbpm4\_hist\_proceinst表中的state的值为"ended"**

\*/

**流程变量**

1、概念

在执行流程实例的过程中会产生一些数据，这些数据会以流程变量的形式保存在流程引擎中

比如说 请假时间 请假原因 等

2、流程变量的生命周期

从流程实例开始到流程实例结束

3、流程变量必须和流程实例绑定在一起

4、什么时候可以把流程变量放入到流程实例中

1、在启动流程实例的时候

@Test

public void testStartPi(){

Map<String, String> variables = new HashMap<String, String>();

variables.put("启动流程实例", "在启动流程实例的时候设置流程变量");

ProcessEngine processEngine = Configuration.getProcessEngine();

processEngine.getExecutionService()

.startProcessInstanceById("qingjia-1", variables);

}

该流程变量被保存在 **jbpm4\_variable**表中

该表中有一个字段:**execution\_**:指向了当前的流程变量所在的流程实例

2、在某一个任务的时候可以设置流程变量

3、只要在流程实例没有结束的任何时候，都可以设置流程变量

4、如果获取流程变量

1、流程实例ID

2、key值

5、jbpm4\_variable表中流程变量的数据会随着流程实例的消失而被删除掉

6、我们需要将一些流程变量保存下来不让它随着流程实例的消失而被删除掉 **Jbpm4\_hist\_var表中**

|  |
| --- |
| /\*\*  \* 一个流程变量是可以通过该方式保存起来的存到了Jbpm4\_hist\_var表中  \*/  @Test  **public** **void** testSaveVariables(){  ProcessEngine processEngine = Configuration.*getProcessEngine*();  processEngine.getExecutionService()  .createVariable("qingjia.60001", "aaa", "aaa", **true**);  } |

**给任务的执行人开始**

**任务的执行人的说法**

1、任务的执行人可以固定在流程图中

<task name="申请请假" g="227,97,92,52" assignee="张三">

<transition name="to 部门经理审批" to="部门经理审批" g="-115,6"/>

</task>

"申请请假"的任务是由张三完成的，这个时候，该任务的执行人是固化在流程图中的

2、在进入该任务节点之前给任务的执行人赋值

<task name="申请请假" g="221,146,92,52" assignee="#{applicator}">

<transition name="to 部门经理审批" to="部门经理审批" g="-121,-8"/>

</task>

在进入该节点之前，必须通过流程变量给applicator赋值



3、在进入该任务节点以后，才要给该任务节点赋值执行人

实现方案有两种：

1、可以通过程序的方式来实现

processEngine.getTaskService()

.assignTask("230002", "王二麻子");

2、在配置文件中

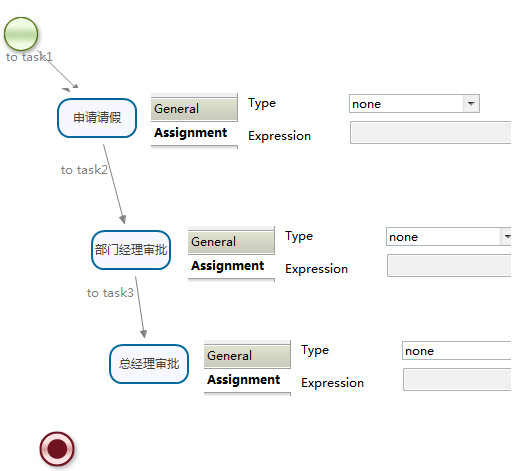
<task name="申请请假" g="233,161,92,52">

<assignment-handler class="eTask.three.MyAssignmentHandler"></assignment-handler>

<transition name="to task2" to="经理审批" g="-52,-22"/>

</task>

当整个流程进入task节点以后，会立刻执行MyAssignmentHandler中的assign方法



**组任务**

1、概念

当进入到一个任务节点的时候，很多人都有执行该任务的权限，但是只要有一个人执行完毕以后该任务就结束了

2、实例

1、可以在配置文件中

<task name="task1" g="237,188,92,52" candidate-users="工程师1,工程师2,工程师3">

<transition name="to end1" to="end1" g="-47,-17"/>

</task>

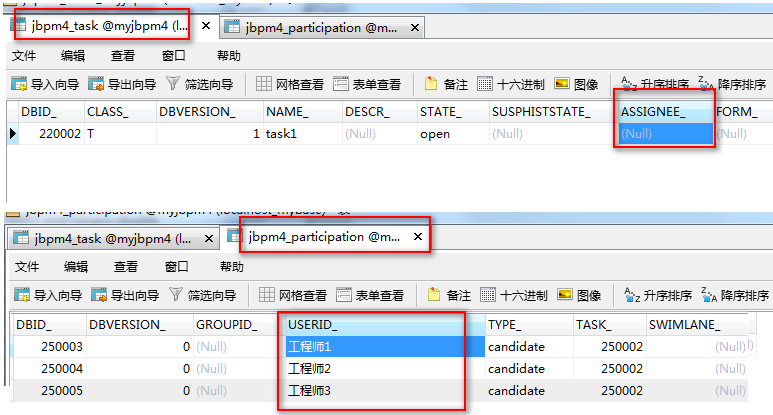
这样task就是组任务

可以对组任务进行一些操作：根据候选人查询组任务，根据组任务查询候选人接受任务。。。

缺点：任务的候选人是固定的



启动流程实例后



2、可以通过代码的形式动态的添加任务的候选人

/\*

\* 动态的添加任务的候选人有两种解决方案：

1、在service层给候选人赋值（也就是在TaskTest.java中）可以通过代码的形式动态的添加任务的候选人

processEngine.getTaskService().addTaskParticipatingUser(taskId, userId, participationType)

2、可以在配置文件中

<process name="grouptask" xmlns="http://jbpm.org/4.4/jpdl">

<start g="133,52,48,48" name="start1">

<transition g="-52,-22" name="to task1" to="task1"/>

</start>

<end g="126,307,48,48" name="end1"/>

<task candidate-users="工程师1,工程师2,工程师3" g="119,153,92,52" name="task1">

<assignment-handler class="fGroupTask.two.GroupAssignmentHandler"></assignment-handler>

<transition g="-50,-22" name="to end1" to="end1"/>

</task>

</process>

在GroupAssignmentHandler类中，可以操作数据库把候选人从数据库中提取出来

缺点：GroupAssignmentHandler类既不能和spring容器关联也不能和servlet容器关联可以利用hibernate或者jdbc技术来完成

\*/

**在工作流中的角色、人的概念**

有组的概念

processEngine.getIdentityService().createGroup("咨询部");

组里有相应的人

processEngine.getIdentityService().createUser("张三","张三", "张三");

建立组和人的关联

processEngine.getIdentityService().createMembership("张三", "咨询部");

上面的操作执行了三张表

jbpm4\_id\_group

jbpm4\_id\_user

jbpm4\_id\_menbership

在api中并没有做到group和user的关联 所以如果要用组的话得需要自己写sql语句去将三张表关联查询

**泳道**

1、在配置文件中

<swimlane name="a" assignee="aaa"></swimlane>

声明了一个泳道

2、在任务节点中

<task name="task1" g="196,150,92,52" swimlane="a">

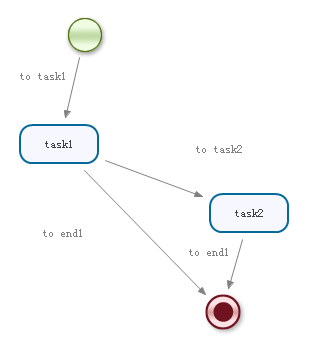
<transition name="to task2" to="task2" g="-95,6"/>

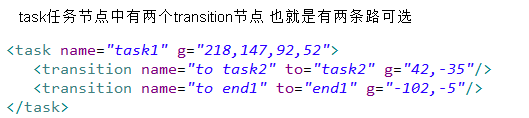
</task>

**给任务的执行人结束**

**节点：**

1、Transition





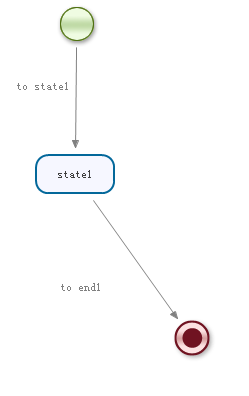
1. State

state （不是一个任务）

不需要执行任务人的参与

使用条件：

不需要人的审批 但还必须有该步骤

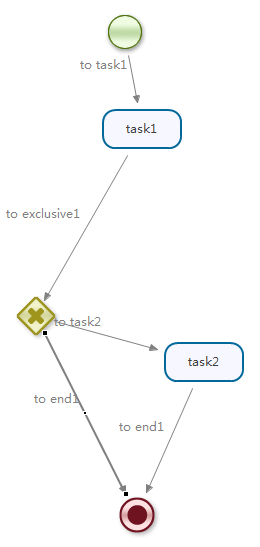


3、decisson

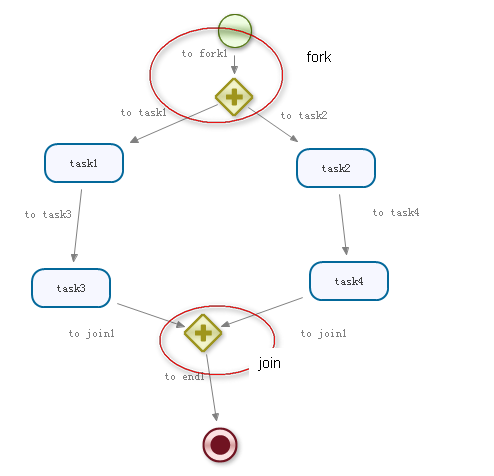
decision 判断节点

这里主要是做判断操作

判断走哪条线



4、Forkjoin



**事件：**

Event

**spring与jbpm的整合原理**

让spring容器产生processEngine

步骤：

1、编辑pom.xml文件

<!--

关于jbpm的jar包

-->

<dependency>

<groupId>org.jbpm.jbpm4</groupId>

<artifactId>jbpm-api</artifactId>

<version>4.4</version>

</dependency>

<dependency>

<groupId>org.jbpm.jbpm4</groupId>

<artifactId>jbpm-log</artifactId>

<version>4.4</version>

</dependency>

<dependency>

<groupId>org.jbpm.jbpm4</groupId>

<artifactId>jbpm-jpdl</artifactId>

<version>4.4</version>

</dependency>

<dependency>

<groupId>org.jbpm.jbpm4</groupId>

<artifactId>jbpm-pvm</artifactId>

<version>4.4</version>

</dependency>

<dependency>

<groupId>org.jbpm.jbpm4</groupId>

<artifactId>jbpm-bpmn</artifactId>

<version>4.4</version>

</dependency>

//jbpm私服的路径

<!--关于jbpm的repository-->

<repositories>

<repository>

<id>appfuse-snapshots</id>

<url>http://oss.sonatype.org/content/repositories/appfuse-snapshots</url>

<releases>

<enabled>false</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>Jboss-JBPM-Repositories</id>

<name>JbossJBPM</name>

<url>http://repository.jboss.org/nexus/content/groups/public</url>

</repository>

</repositories>

<!--关于jbpm的repository-->

2、在src/main/resources中建立一个package为jbpm

在jbpm下jbpm.cfg.xml文件

把原来的<import resource="jbpm.tx.hibernate.cfg.xml" />替换成

<import resource="jbpm.tx.spring.cfg.xml" />

3、把jbpm的相关的映射文件加入到hibernate.cfg.xml文件中

<mapping resource="jbpm.repository.hbm.xml" />

<mapping resource="jbpm.execution.hbm.xml" />

<mapping resource="jbpm.history.hbm.xml" />

<mapping resource="jbpm.task.hbm.xml" />

<mapping resource="jbpm.identity.hbm.xml" />

4、在spring的配置文件中引入processEngine

<bean id="springHelper" class="org.jbpm.pvm.internal.processengine.SpringHelper">

<property name="jbpmCfg" value="jbpm/jbpm.cfg.xml"></property>

</bean>

<bean id="processEngine" factory-bean="springHelper" factory-method="createProcessEngine" />

1. 测试processEngine:启动tomcat服务器spring容器加载processEngine 生成实例对象

5、

如果没有hibernate.cfg.xml文件 也就是说 使用Spring引入sessionFactory的第二种方式 来引入sessionFactory

如何引入jbpm的映射文件？

在applicationContext-db.xml中

<!--

<property name="mappingJarLocations">

<value>WEB-INF/lib/jbpm-4.4.jar</value>

</property>

-->

## 4、分析黑马报名系统

在该项目上线的时候就可以把整个流程图部署到jbpm引擎中了

1、学生在前台

注册

经历整个流程的所有的申请

具体的分析：

1、在提交某一个具体的课程的时候，启动流程实例

2、可以提交自荐信

2、咨询员后台

对各个环节学生提交的信息进行审批，并且打分



技术分析：

1、不管是学生还是咨询员登录系统以后，都能查询到当前正在执行的任务

根据执行的任务-->executionId--->Process Instance--->piid-->和相关的数据表就能关联在一起了

2、jbpm表做流程控制的作用，而系统的表是用来存储数据的

**他们之间经过piid进行关联**

3、对各个审批节点的候选人做了如下的处理：

<task g="258,153,92,52" name="自荐信审批">

<assignment-handler class="com.itheimaoa.domain.process.MyAssignmentHandler"/>

<transition g="2,-28" name="to 提交入学考试" to="提交入学考试"/>

<transition g="336,98:-83,-17" name="to 提交自荐信" to="提交自荐信"/>

</task>

**每一个审批节点的候选人都是由MyAssignmentHandler来赋值的（也可以使用泳道解决）**

@Override

public void assign(Assignable assignable, OpenExecution execution) throws Exception {

Connection conn=JDBCUtils.getConnection();

String jpql = " select eid from employee";

Statement st = conn.createStatement();

ResultSet rs = st.executeQuery(jpql);

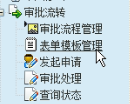
while(rs.next()){

assignable.addCandidateUser(String.valueOf(rs.getInt("eid")));

}

}

## 5、审批流转

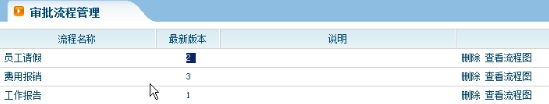


**需求分析**

1、审批流程管理: 对流程定义进行管理

crud的操作

显示流程定义列表的时候，显示的是最新版本的流程定义



2、表单模板管理

就像请假条，请假条是有模板的（时间 原因.....）每一个填写完毕的请假条就是一个form表单





3、发起申请



4、审批处理

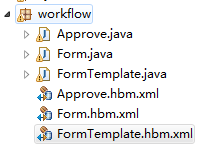


5、状态查询

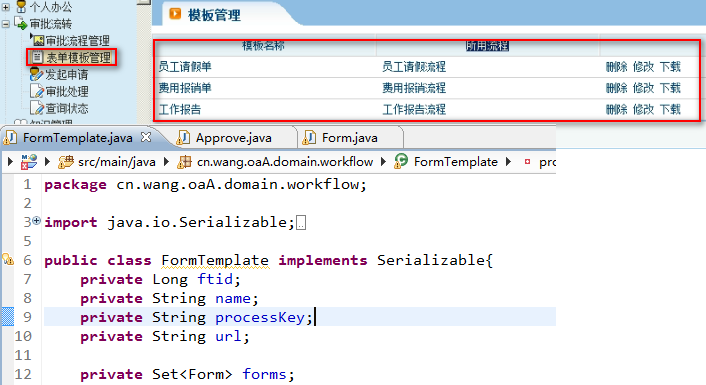


**步骤**

**1.建表**



**FormTemplate表单模板表**





**Form具体的表单表（一个模板对多个表单）**

**这个表中有一个piid字段将form表单和jbpm联系起来**



**Approve 针对每张form表单的审批流转记录（一个表单对多个审批）**



**在hibernate配置文件中加入这三张表的映射**

1. **审批流程管理**

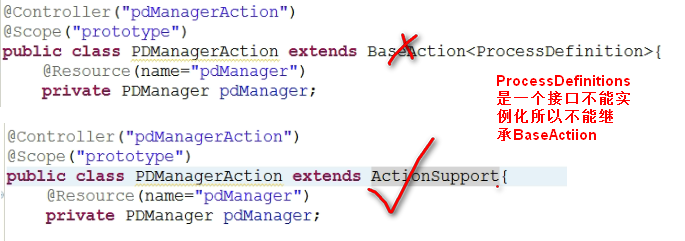
他不需要dao层直接在service层创建接口PDManager.java

A：查看所有的流程定义中的最高版本

在 PDManagerImpl.java中必须引入ProcessEngine



PDManagerAction

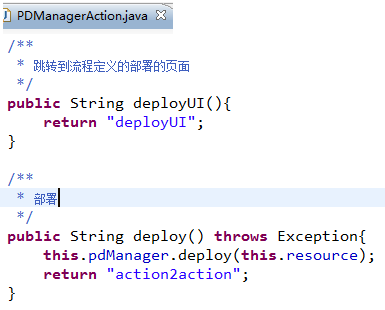


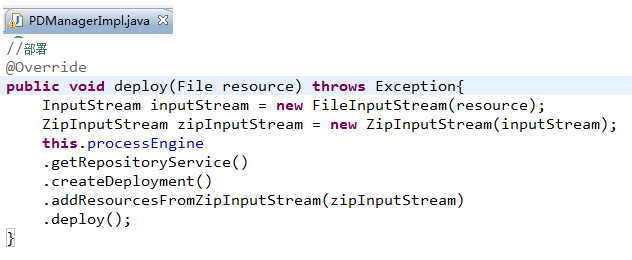
struts-workflow.xml

workflow/pdmanager/list.jsp

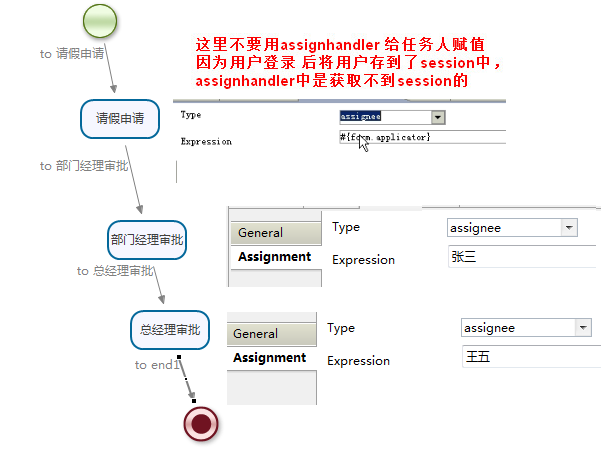
B：部署流程定义







创建请假jbmp



C：删除流程定义

D: 查看流程图

1、在service层

根据deploymentId和图片的路径获取到一个输入流

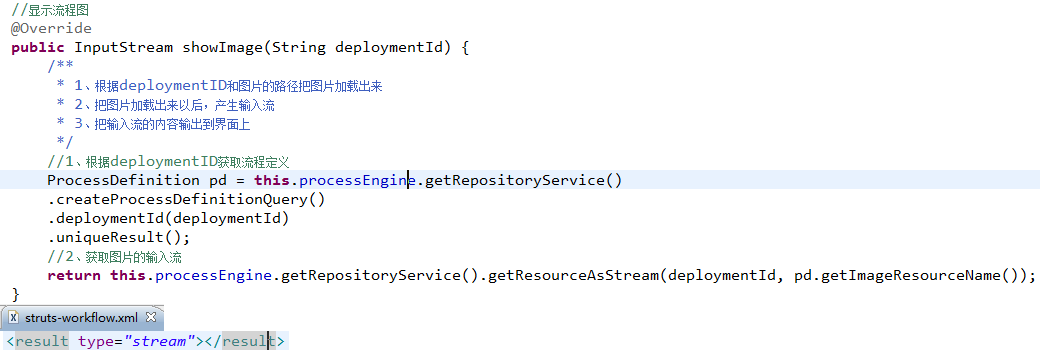
也就是说service层的该方法应该返回一个InputStream

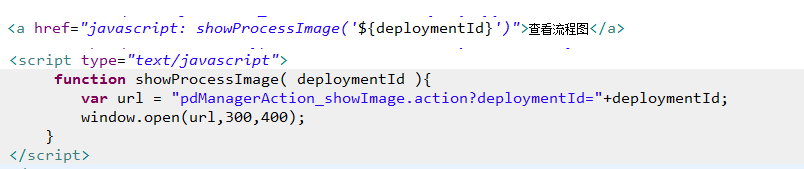
2、在action层

有一个属性：inputStream

service层的方法的返回值赋值给inputStream

3、在页面上利用window.open打开一个窗口，让输出流流到该窗口中





1. **表单模板管理**

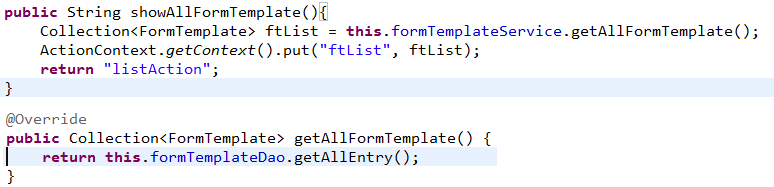
1: DepartmentDaoImpl.java

2: FormTemplateServiceImpl.java

3: FormTemplateAction.java

4: struts-workflow.xml

A:列表页面



B：表单模板添加页面

|  |
| --- |
| /\*\*  \* 跳转到表单模板的添加页面  \* 添加页面需要选择该表单模板所属的流程定义所以要把所有的流程定义查出来  \*/  **public** String addUI(){  Collection<ProcessDefinition> pdList = **this**.pdManager.getLasterVersion();  ActionContext.*getContext*().put("pdList", pdList);  **return** *ADD\_UI*;  } |

C：保存表单模板动作

|  |
| --- |
| **public** String add(){  /\*\*  \* 1、把表单模板上传到服务器上  \* 2、在表单模板中插入一行数据  \*/  String url = **UploadUtils**.*saveUploadFile*(**this**.resource);  FormTemplate formTemplate = **new** FormTemplate();  formTemplate.setName(**this**.getModel().getName());  formTemplate.setProcessKey(**this**.getModel().getProcessKey());  formTemplate.setUrl(url);  **this**.formTemplateService.saveFormTemplate(formTemplate);  **return** *action2action*;  } |

D：表单模板下载

1、根据ftid查找到formTemplate对象

2、根据该对象查找到url

3、根据url创建一个FileInputStream

4、在action类中有一个属性：InputStream,把上面的步骤得到的inputstream赋值给action属性的inputStream

5、在struts2的配置文件中：

结果集类型为"stream"

<result name="download" type="stream">

<!--

输入流的名字

用来获取inputStream

-->

<param name="inputName">inputStream</param>

<!-- 名称的名字 -->

<param name="contentDisposition">

<!--

应用ognl表达式动态显示要下载的文件的名,字

-->

attachment;filename="${#fileName}.doc" **在struts中也可以使用ognl表达式获取值栈中的值**

</param>

</result>

说明：fileName来自于map栈

String url = formTemplate.getUrl();

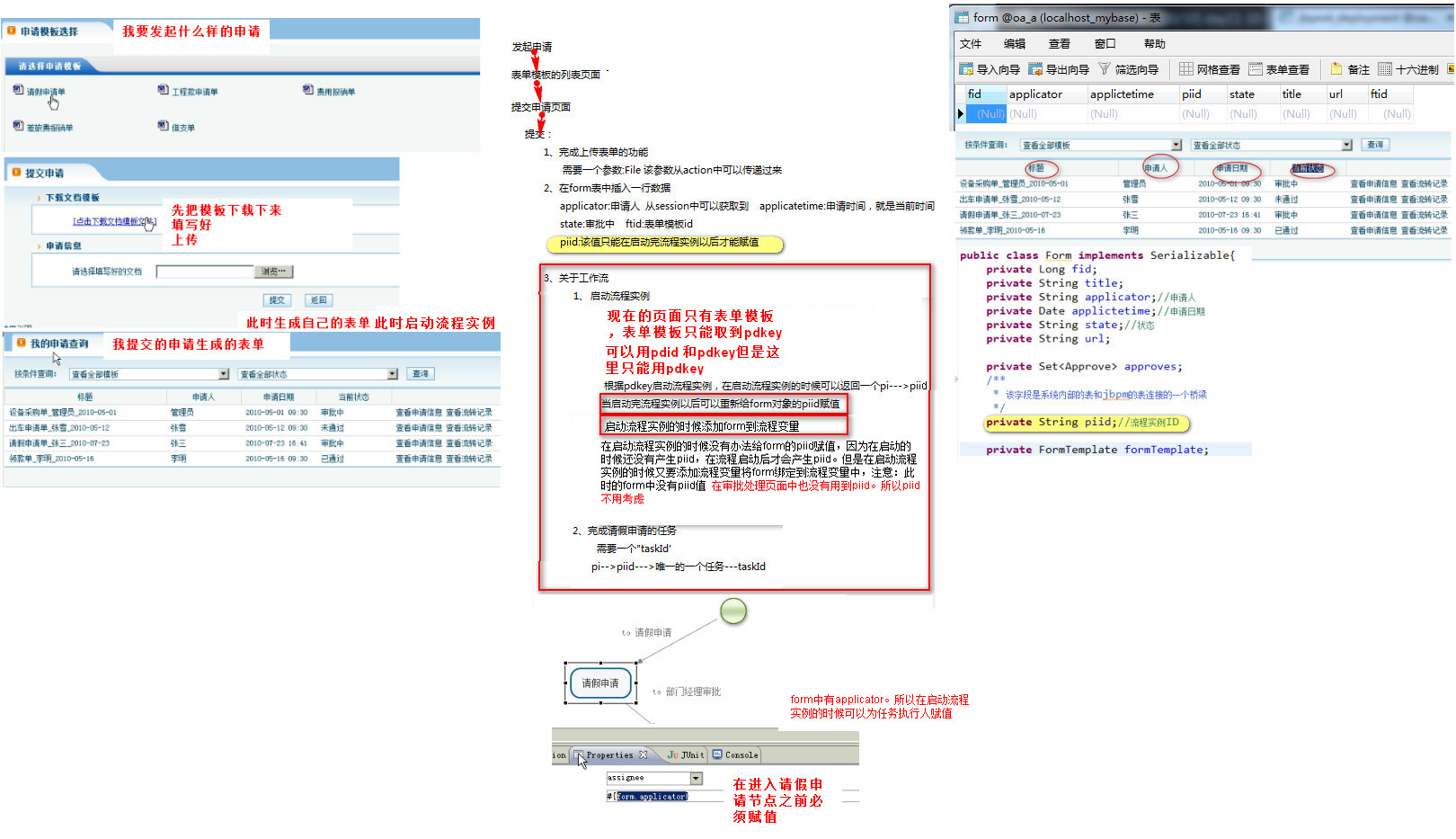
String fileName = formTemplate.getName();

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

ActionContext.getContext().put("fileName", fileName);

这些代码把文件的名称放入到了map栈

1. **发起申请**



1: WorkFlowActionl.java

2: WorkFlowServiceImpl.java

3: struts-workflow.xml

A:列表页面 WorkFlowActionl.java

B:提交申请页面 WorkFlowActionl.java 

C: 提交动作 WorkFlowActionl.java **(测试这一步需要从登陆开始因为需要从session中获取user)**

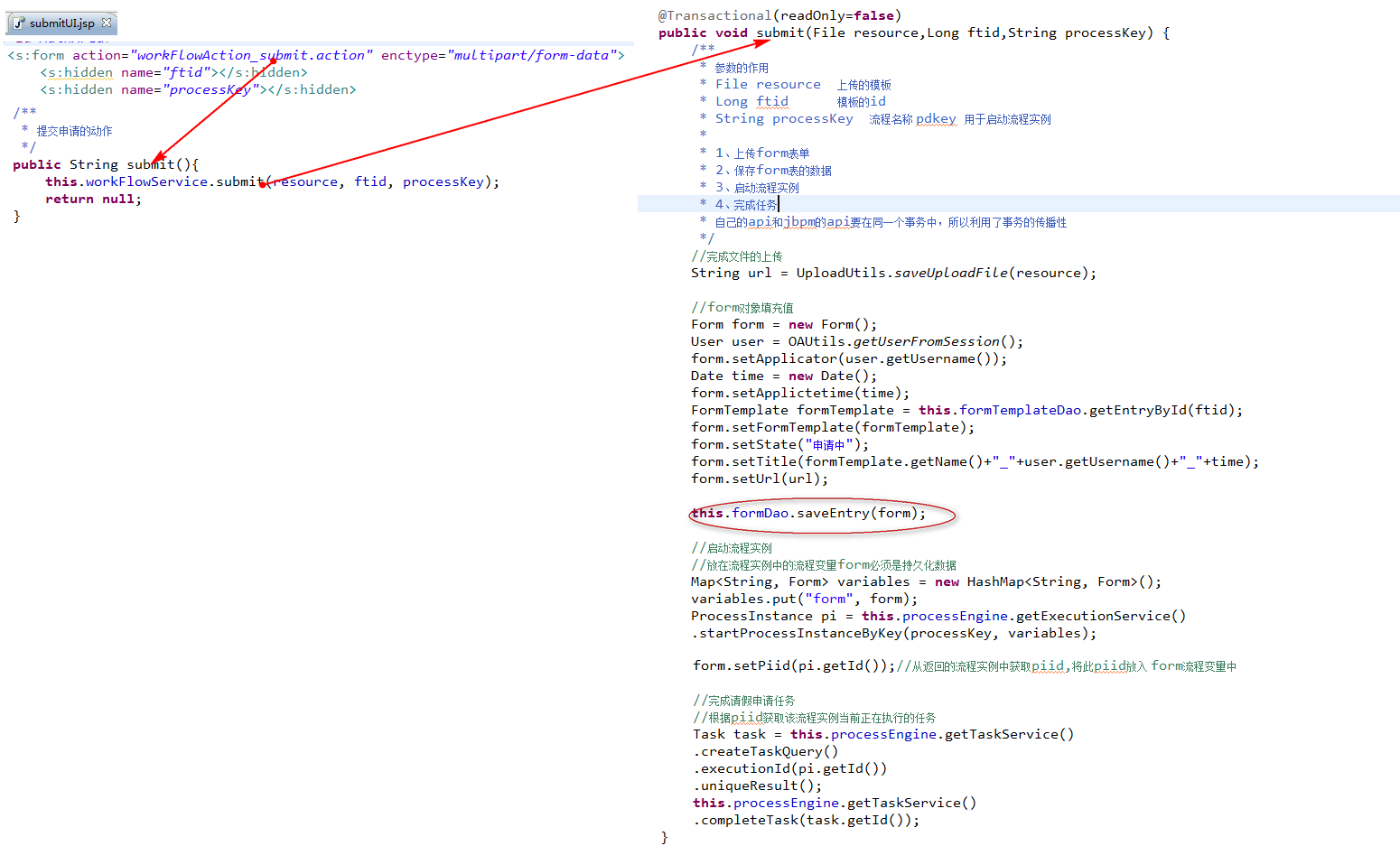
\* 1、上传form表单

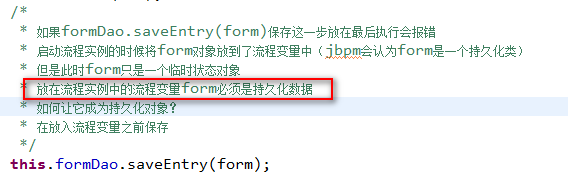
\* 2、保存form表的数据**（需要formDao）**

\* 3、启动流程实例

\* 4、完成任务

自己的api和jbpm的api要在同一个事务中，所以利用了事务的传播性

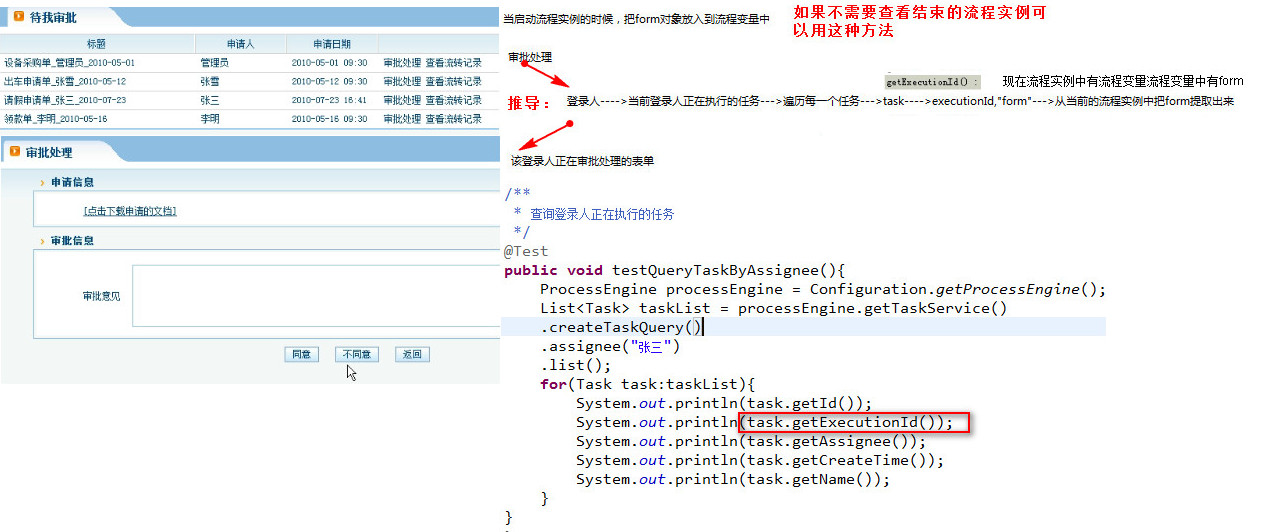




数据库中添加url



**5. 待我审批 和 审批处理**

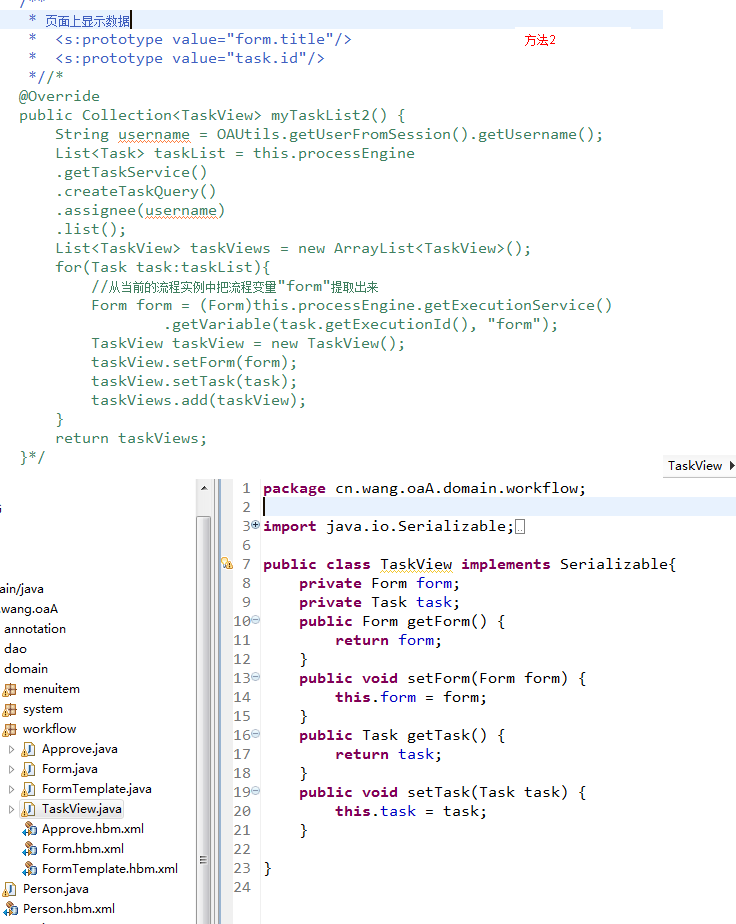


**展示待审批的form表单有两种方式：**

**方式一：**



**方式二**



**审批处理：**

|  |
| --- |
| @Override  **public** **void** approve() {  /\*\*  \* 如果在页面上点击的是不同意，则流程实例直接结束  \* 改变相应的form表中state字段的值为"未通过"  \* 如果页面上点击的是同一  \* 1、完成任务  \* 2、判断当前的流程实例是否已经结束  \* 如果没有结束，则什么都不做  \* 如果结束，改变form表中相应的state字段的值"已通过"  \*  \* 在approve表中插入一行数据  \*/      /\*\*  \* 如果值传递一个fid **（针对第一种展示待审批的form表单的方式）**  \* 1、根据fid把form对象提取出来  \* 2、根据form对象获取到piid  \* 3、根据piid把当前流程实例正在执行的任务提取出来  \* 4、根据任务可以得到taskId  \* 5、完成任务  \* 如果值传递两个参数:fid,taskid **（针对第二种展示待审批的form表单的方式）**  \* 这样可以根据taskId直接完成任务  \*/    /\*\*  \* 页面上传递过来一个字符串：是否同意  \*/  String isapprove = "";  **if**(isapprove.equals("同意")){  //完成任务  //判断流程实例是否结束  //如果结束，则改变form表中相应的行的state状态的值  }**else**{  //直接结束流程实例  //改变form表的状态  }  //往approve表中插入一行数据  /\*\*  \* 在审批处理的页面有两个button,分别是同意和不同意  \* <input type="image" src="../style/blue/images/button/agree.png"/>  <input type="image" src="../style/blue/images/button/disagree.png"/>  1、在页面上创建一个隐藏域  2、在页面上分别对这两个button添加click事件  3、根据不同的click事件，对隐藏域赋值不同的值  4、action端可以根据其值来判断是同意还是不同意  \*/  } |

# 项目回顾

