# 

**Spring学习笔记**

**Java相关课程系列笔记之十五**

**笔记内容说明**

Spring（梁建全老师主讲，占笔记内容100%）；

**目 录**

[一、 Spring概述](#_Toc18490)

[1.1 Spring框架的作用](#_Toc32354)

[1.2 Spring框架的优点](#_Toc17443)

[1.3 Spring框架的容器](#_Toc20131)

[二、 Spring容器的基本应用](#_Toc7824)

[2.1如何将一个Bean组件交给Spring容器](#_Toc6257)

[2.2如何获取Spring容器对象和Bean对象](#_Toc1539)

[2.3如何控制对象创建的模式](#_Toc8985)

[2.4 Bean对象创建的时机](#_Toc29736)

[2.5为Bean对象执行初始化和销毁方法](#_Toc15098)

[2.6案例：Spring框架的使用以及2.1节-2.5节整合测试](#_Toc24501)

[三、 Spring框架IoC特性](#_Toc20579)

[3.1 IoC概念](#_Toc17000)

[3.2 DI概念](#_Toc15900)

[3.3案例：测试IoC（set注入）](#_Toc15332)

[3.4案例：测试IoC（构造注入）](#_Toc32079)

[3.5案例：不用JDBC访问数据库，而是采用Hibernate访问](#_Toc8311)

[四、 Spring中各种类型的数据注入](#_Toc31157)

[4.1 Bean对象注入](#_Toc31504)

[4.2基本数据的注入](#_Toc4143)

[4.3集合的注入](#_Toc1009)

[4.4案例：各类数据注入](#_Toc21307)

[五、 AOP概念](#_Toc23008)

[5.1什么是AOP](#_Toc19418)

[5.2 AOP和OOP的区别](#_Toc31199)

[5.3 AOP相关术语](#_Toc25319)

[5.4案例：AOP的使用，模拟某些组件需要记录日志的功能](#_Toc4537)

[5.5通知类型](#_Toc15859)

[5.6切入点](#_Toc27622)

[5.7案例：环绕通知，修改5.4案例使之动态显示所执行的操作](#_Toc29721)

[5.8案例：利用AOP实现异常处理，将异常信息写入文件](#_Toc26511)

[六、 Log4j日志记录工具](#_Toc25198)

[6.1 Log4j介绍](#_Toc17854)

[6.2 Log4j的使用](#_Toc25848)

[6.3案例：修改5.8案例，使用Log4j记录日志](#_Toc7419)

[七、 Spring注解配置](#_Toc8271)

[7.1组件扫描功能](#_Toc3814)

[7.2组件扫描的使用方法](#_Toc10584)

[7.3注入注解标记使用方法](#_Toc30606)

[7.4 AOP注解标记使用方法](#_Toc949)

[八、 Spring对数据访问技术的支持](#_Toc19606)

[8.1 Spring提供了统一的异常处理类型](#_Toc29922)

[8.2 Spring提供了编写DAO的支持类](#_Toc21094)

[8.3 Spring提供了声明式事务管理方法](#_Toc22808)

[8.4 Spring框架如何使用JDBC技术](#_Toc16374)

[8.5连接池优点](#_Toc20590)

[8.6 Spring框架如何使用Hibernate技术](#_Toc28158)

[8.7 Spring+Hibernate如何使用Session、Query等对象](#_Toc3535)

[8.8 Spring框架和Struts2整合应用](#_Toc21645)

[8.9案例：采用SSH结构重构资费管理模块](#_Toc7860)

[九、 整合开发包struts-spring-plugin.jar](#_Toc29258)

[9.1 Struts2创建对象的方式](#_Toc31189)

[9.2 struts-spring-pligin.jar创建对象的方式](#_Toc21244)

[9.3 struts-spring-plugin.jar的内部实现](#_Toc21540)

[9.4原理图1](#_Toc22276)

[9.5原理图2](#_Toc30412)

[9.6注意事项](#_Toc11244)

[9.7注入规则](#_Toc26130)

[十、 Spring的事务管理](#_Toc9793)

[10.1声明式事务管理（基于配置方式实现事务控制）](#_Toc26690)

[10.2编程式事务管理（基于Java编程实现事务控制），不推荐用！](#_Toc28485)

[10.3 Spring中常用的事务类型](#_Toc13590)

[十一、 Spring的MVC](#_Toc19783)

[11.1 Spring MVC的体系结构](#_Toc6452)

[11.2 Spring MVC的工作流程](#_Toc17312)

[11.3案例：简易登录（基于XML配置，不推荐使用）](#_Toc30387)

[11.4案例：修改11.3案例（基于注解配置，推荐使用）](#_Toc28578)

[十二、 其他注意事项](#_Toc4816)

[12.1 Spring的核心模块](#_Toc29184)

[12.2表单中action属性的相对、绝对路径问题](#_Toc25142)

[12.3用SSH重构NetCTOSS项目模块的步骤](#_Toc28789)

# Spring概述

我们学习Spring框架的最终目的是用它整合Struts2、Hibernate框架（SSH）。

## 1.1 Spring框架的作用

Spring框架主要负责技术整合（可以整合很多技术），该框架提供IoC和AOP机制，基于这些特性整合，可以降低系统组件之间的耦合度，便于系统组件的维护、扩展和替换。

## 1.2 Spring框架的优点

其实与Spring框架的作用相同：

在SSH中，主要是利用Spring容器管理我们程序中的Action、DAO等组件，通过容器的IoC机制，可以降低Action、DAO之间的耦合度（关联度），利用AOP进行事务管理等共通部分的处理。

在SSH中，Struts2主要是利用它的控制器，而不是标签、表达式；Hibernate主要利用它的数据库访问；Spring主要是利用它的整合。

## 1.3 Spring框架的容器

Spring框架的核心是提供了一个容器（是我们抽象出来的，代指后面的类型）。该容器类型是BeanFactory或ApplicationContext（建议用这个类型，它是BeanFactory的子类，功能更多）。

该容器具有以下功能：

1. 容器可以创建和销毁组件对象，等价于原来“工厂”类的作用。
2. 容器可以采用不同的模式创建对象，如单例模式创建对象。
3. 容器具有IoC机制实现。
4. 容器具有AOP机制实现。

# Spring容器的基本应用

## 2.1如何将一个Bean组件交给Spring容器

1）Bean组件其实就是个普通的Java类！

2）方法：在applicationContext.xml中添加以下定义，见2.6案例中step4。

<bean id="标识符" class="Bean组件类型"></bean>

## 2.2如何获取Spring容器对象和Bean对象

1. 实例化容器：

ApplicationContext ac=new ClassPathXmlApplicationContext("/applicationContext.xml");

//FileSystemXmlApplicationContext("");//去指定的磁盘目录找，上面的为去Class路径找

2）利用getBean("标识符")方法获取容器中的Bean对象。见2.6案例中step5。

## 2.3如何控制对象创建的模式

Spring支持singleton（单例）和prototype（原型，非单例）两种模式。

默认是singleton模式，可以通过<bean>的scope属性修改为prototype模式。以后在Web程序中，通过扩展可以使用request、session等值。见2.6案例中step4、step7。

例如：<bean id="标识符" scope="prototype" class="Bean组件类型"></bean>

* 注意事项：对于NetCTOSS项目，一个请求创建一个Action，所以用Spring时必须指明prototype，否则默认使用singleton会出问题。而DAO则可用singleton模式。

## 2.4 Bean对象创建的时机

1）singleton模式的Bean组件是在容器实例化时创建。

2）prototype模式是在调用getBean()方法时创建。

3）singleton模式可以使用<bean>元素的lazy-init="true"属性将对象的创建时机推迟到调用getBean()方法。也可以在<beans>（根元素）中使用default-lazy-init="false"推迟所有单例Bean组件的创建时机。见2.6案例中step3、step4。

例如：<bean id="标识符" scope="singleton" lazy-init="true" class="Bean组件类型"></bean>

<beans ...... default-lazy-init="false"></beans>

## 2.5为Bean对象执行初始化和销毁方法

1）初始化：①可以利用<bean>元素的init-method="方法名"属性指定初始化方法。

②指定的初始化方法是在构造方法调用后自动执行。若非单例模式，则每创建一个对象，则执行一次初始化方法（单例、非单例模式都可）。见2.6案例中step3、step4。

* 注意事项：
* 初始化的三种方式：写构造方法中；或写{ }中（代码块）；Spring框架中<bean>元素写init-method="方法名"属性。
* 初始化不能用static{ }，它是类加载调用，比创建对象要早。

1. 销毁：①可以利用<bean>元素的destroy-method="方法名"属性执行销毁方法。

②指定的销毁方法是在容器关闭时触发，而且只适用于singleton模式的组件（只能为单例模式）。见2.6案例中step3、step4、step6。

## 2.6案例：Spring框架的使用以及2.1节-2.5节整合测试

step1：导入Spring开发包：spring.jar、commons-logging.jar和配置文件：applicationContext.xml

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

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

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

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

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

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

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd

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

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

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

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

</beans>

step2：在org.tarena.dao包下，创建接口CostDAO，添加两个方法

public void delete(); public void save();

step3：在org.tarena.dao包下，创建JdbcCostDAO类，并实现CostDAO接口

public JdbcCostDAO(){ System.out.println("创建CostDAO对象"); }

public void myinit(){ System.out.println("初始化CostDAO对象"); }

public void mydestroy(){ System.out.println("销毁CostDAO对象"); }

public void delete() { System.out.println("利用JDBC技术实现删除资费记录"); }

public void save() { System.out.println("利用JDBC技术实现保存资费记录"); }

step4：在applicationContext.xml配置文件中，将Bean组件（Java类）交给Spring容器

<bean id="jdbcCostDAO" scope="singleton" lazy-init="true" init-method="myinit"

destroy-method="mydestroy" class="org.tarena.dao.JdbcCostDAO">

</bean>

step5：在org.tarena.test包下，创建TestApplicationContext类，获取Spring容器对象，并测试

@Test

public void test1(){ String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);//实例化容器

CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDAO");//获取Bean对象

costDAO.save(); costDAO.delete(); }

step6：在TestApplicationContext类中添加方法，测试销毁对象

@Test

/\*\*关闭容器才会触发销毁，但关闭容器方法封装在AbstractApplicationContext类中 \*/

public void test2(){ String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);

AbstractApplicationContext ac=new ClassPathXmlApplicationContext(conf);

CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDAO"); ac.close(); }

step7：在TestApplicationContext类中添加方法，测试单例

@Test

public void test3(){ String conf="/applicationContext.xml";

ApplicationContext ac= new ClassPathXmlApplicationContext(conf);

CostDAO costDAO1=(CostDAO)ac.getBean("jdbcCostDAO");

CostDAO costDAO2=(CostDAO)ac.getBean("jdbcCostDAO");

System.out.println(costDAO1==costDAO2);//true，所以Spring默认为单例模式 }

# Spring框架IoC特性

## 3.1 IoC概念

1）Inverse of Controller被称为控制反转或反向控制，其实真正体现的是“控制转移”。

2）所谓的控制指的是负责对象关系的指定、对象创建、初始化和销毁等逻辑。

3）IoC指的是将控制逻辑交给第三方框架或容器负责（即把Action中的控制逻辑提出来，交给第三方负责），当两个组件关系发生改变时，只需要修改框架或容器的配置即可。

4）IoC主要解决的是两个组件对象调用问题，可以以低耦合方式建立使用关系。

## 3.2 DI概念

1）Dependency Injection依赖注入。

2）Spring框架采用DI技术实现了IoC控制思想。

3）Spring提供了两种形式的注入方法：

①setter方式注入（常用）：依靠set方法，将组件对象传入（可注入多个对象）。

A.首先添加属性变量和set方法。

B.在该组件的<bean>定义中采用下面的描述方式：

<property name="属性名" ref="要注入的Bean对象的id值"></property>

* 注意事项：例如CostAction中有costDAO属性，而它的标准set方法名为setCostDAO，那么配置文件中的name就应该写costDAO（去掉set，首字母小写）。如果set方法名为setCost，那么name就应该写cost（去掉set，首字母小写）！确切的说，name不是看定义的属性名，而是set方法名。

②构造方式注入（用的少）：依靠构造方法，将组件对象传入。

A.在需要注入的组件中，添加带参数的构造方法。

B.在该组件的<bean>定义中，使用下面格式描述：

<constructor-arg index="参数索引" ref="要注入的Bean对象的id值"></constructor-arg>

## 3.3案例：测试IoC（set注入）

step1：接2.6案例，在org.tarena.action包下，创建CostAction类，调用save方法

private CostDAO costDAO;//利用Spring的IOC机制使用CostDAO组件对象，set注入

public void setCostDAO(CostDAO costDAO) { this.costDAO = costDAO; }

public String execute(){ System.out.println("处理资费添加操作");

costDAO.save();//调用CostDAO中的save方法 return "success"; }

step2：在applicationContext.xml配置文件中，将CostAction组件交给Spring容器

<bean id="costAction" scope="prototype" class="org.tarena.action.CostAction">

<!-- 利用setCostDAO方法接收jdbcCostDAO对象 -->

<property name="costDAO" ref="jdbcCostDAO"></property>

<!-- name：与CostAction中对应的set方法匹配的名。ref：指明哪个对象 -->

</bean><!--此处用到了2.6案例中step3描述的组件JdbcCostDAo-->

step3：在org.tarena.test包下，创建TestIoc类，用于测试IOC机制

@Test //测试set注入

public void test1(){ String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);

CostAction action=(CostAction)ac.getBean("costAction");//获得CostAction类的对象

action.execute(); }

step4：测试结果为：

创建CostDAO对象 初始化CostDAO对象 处理资费添加操作

利用JDBC技术实现保存资费记录。

## 3.4案例：测试IoC（构造注入）

step1：接3.3案例，在org.tarena.action包下，创建DeleteAction类，调用delete方法

private CostDAO costDAO;

public DeleteAction(CostDAO costDAO){ this.costDAO=costDAO; }//构造注入

public String execute() { System.out.println("处理资费删除操作");

costDAO.delete();//调用CostDAO中的delete方法 return "success"; }

step2：在applicationContext.xml配置文件中，将DeleteAction组件交给Spring容器

<bean id="deleteAction" scope="prototype" class="org.tarena.action.DeleteAction">

<!-- 索引0：给构造方法中第一个参数注入一个jdbcCostDAO对象。

若多个参数则重复追加constructor-arg元素即可 -->

<constructor-arg index="0" ref="jdbcCostDAO"></constructor-arg><!-- 构造注入 -->

</bean>

step3：在TestIoc类中添加方法，测试构造注入

@Test //测试构造注入

public void test2(){ String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);

DeleteAction action=(DeleteAction)ac.getBean("deleteAction");

action.execute(); }

step4：测试结果为：

创建CostDAO对象 初始化CostDAO对象 处理资费删除操作

利用JDBC技术实现删除资费记录。

## 3.5案例：不用JDBC访问数据库，而是采用Hibernate访问

接3.3案例，如果不用JDBC访问数据库，而是采用Hibernate访问，则替换组件过程为：

step1：创建HibernateCostDAO类，并实现CostDAO接口

public void delete() { System.out.println("利用Hibernate技术实现删除资费记录"); }

public void save() { System.out.println("利用Hibernate技术实现保存资费记录"); }

step2：在applicationContext.xml配置文件中，将HibernateCostDAO组件交给Spring容器

<bean id="hibernateCostDAO" class="org.tarena.dao.HibernateCostDAO"></bean>

step3：修改3.3案例中step2中CostAction组件的描述

<bean id="costAction" scope="prototype" class="org.tarena.action.CostAction">

<!-- 修改ref属性的指引 -->

<property name="costDAO" ref="hibernateCostDAO"></property>

<!-- name：与CostAction中添加的属性名相同。ref：指明哪个对象 -->

</bean>

step4：再次执行3.3案例step3中test1方法，测试结果为：

处理资费添加操作 利用Hibernate技术实现保存资费记录

# Spring中各种类型的数据注入

Spring可以为对象注入以下类型的数据。

## 4.1 Bean对象注入

<property name="属性名" ref="要注入的Bean对象的id值"></property>

## 4.2基本数据的注入

1）字符串、数字

<property name="属性名" value="要注入的值"></property>

## 4.3集合的注入

1）List、Set

<property name="集合属性名">

<list><!-- 普通值用<value>标签，对象用<bean>标签 -->

<value>集合中的值1</value><value>集合中的值2</value> …………

</list>

</property>

2）Map

<property name="集合属性名">

<map><!-- 普通值用<value>标签，对象用<bean>标签 -->

<entry key="键1" value="值1"></entry>

<entry key="键2" value="值2"></entry> …………

</map>

</property>

3）Properties

<property name="集合属性名">

<props>

<prop key="键1">值1</prop>

<prop key="键2">值2</prop> …………

</props>

</property>

4）特殊用法：set方法接收字符串，内部进行处理（如分割），再存入集合属性

<property name="集合属性名" value="字符串"></property>

## 4.4案例：各类数据注入

step1：对象注入参考3.3、3.4案例

step2：创建MessageBean类，并定义不同类型的数据以及对应的set方法

private String username;//用户名 private String fileDir;//上传路径

private List<String> hbms; private Set<String> cities;

private Map<Integer, String> books; private Properties props;

private Set<String> types;//允许上传类型

/\*\* 注意set方法的名字！不是看属性名，而是看set方法名去掉set，首字母大写的名 \*/

public void setName(String username) { this.username = username; }//手动更改过名字

public void setDir(String fileDir) { this.fileDir = fileDir; }//手动更改过名字

……其他属性名字没改，其他属性代码略

public void setTypes(String str) {//特殊用法：注入一个字符串，分析之后给set集合赋值

String[] arr=str.split(","); types=new HashSet<String>();

for(String s:arr){ types.add(s); } }

public void show(){

System.out.println("用户名："+username); System.out.println("上传路径："+fileDir);

System.out.println("--hbms文件如下--");

for(String s:hbms){ System.out.println(s); }

System.out.println("--city城市如下--");

for(String c:cities){ System.out.println(c); }

System.out.println("--book图书信息--");

Set<Entry<Integer,String>> ens=books.entrySet();

for(Entry en:ens){ System.out.println(en.getKey()+" "+en.getValue()); }

System.out.println("--props参数如下--");

Set keys=props.keySet();//另一种方式遍历

for(Object key:keys){ System.out.println(key+" "+

props.getProperty(key.toString())); }

System.out.println("--允许上传类型如下--");//特殊用法

for(String type:types){ System.out.println(type); } }

step3：applicationContext.xml配置文件中

<!-- 各种数据类型的注入 -->

<bean id="messageBean" class="org.tarena.dao.MessageBean">

<!-- 注意名字 name指的是set方法名去掉set，首字母大写的名，不看属性名！ -->

<property name="name" value="root"></property><!--手动更改过set方法名 -->

<property name="dir" value="D:\\images"></property><!--手动更改过set方法名 -->

<property name="hbms">

<list><value>/org/tarena/entity/Cost.hbm.xml</value>

<value>/org/tarena/entity/Admin.hbm.xml</value></list></property>

<property name="cities">

<set><value>北京</value><value>上海</value></set></property>

<property name="books">

<map><entry key="1" value="Java语言基础"></entry>

<entry key="2" value="Java Web入门"></entry></map></property>

<property name="props">

<props><prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.dialect\_sql">org.hibernate.dialect.OracleDialect</prop>

</props>

</property>

<!-- 特殊用法，set方法传入字符串，内部进行处理，再存入集合 -->

<property name="types" value="jpg,gif,jpeg"></property>

</bean>

step4：创建TestInjection类用于测试各类数据的注入

@Test

public void test1(){

String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);

MessageBean bean=(MessageBean)ac.getBean("messageBean");

bean.show(); }

# AOP概念

## 5.1什么是AOP

Aspect Oriented Programming，被称为面向方面编程。对单个对象（一对一）的解耦用IOC，而当有个共通组件，它对应多个其他组件（一对多），则解耦用AOP。如，拦截器。这也是为何在程序中大量的用IoC，而AOP却用的很少，因为程序中不可能有很多的共通部分。

## 5.2 AOP和OOP的区别

OOP是面向对象编程，AOP是以OOP为基础的。

OOP主要关注的是对象，如何抽象和封装对象。

AOP主要关注的是方面，方面组件可以以低耦合的方式切入到（作用到）其他某一批目标对象方法中（类似于Struts2中的拦截器）。

AOP主要解决共通处理和目标组件之间解耦。

## 5.3 AOP相关术语

1. 方面（Aspect）：指的是封装了共通处理的功能组件。该组件可以作用到某一批目标组件的方法上。  
    2）连接点（JoinPoint）：指的是方面组件和具体的哪一个目标组件的方法有关系。
2. 切入点（Pointcut）：用于指定目标组件的表达式。指的是方面组件和哪一批目标组件方法有关系。多个连接点组成的集合就是切入点。如：a、b为切入点，1、2为连接点。

a

b

save()

delete()

delete()

save()

方面组件

HibernateCostDAO

JdbcCostDAO

1

2

1

2

4）通知（Advice）：用于指定方面组件和目标组件方法之间的作用时机。例如：先执行方面组件再执行目标方法；或先执行目标方法再执行方面组件。

5）目标（Target）：利用切入点指定的组件和方法。

6）动态代理（AutoProxy）：Spring同样采用了动态代理技术实现了AOP机制。当使用AOP之后，从容器getBean()获取的目标组件，返回的是一个动态生成的代理类。然后通过代理类执行业务方法，代理类负责调用方面组件功能和原目标组件功能。

Spring提供了下面两种动态代理技术实现：

1）采用CGLIB技术实现（目标组件没有接口采用此方法）

例如：public class 代理类 extends 原目标类型 { }

CostAction action=new 代理类();//代理类中有原来类的方法

2）采用JDK Proxy API实现（目标组件有接口采用此方法，即实现了某个接口）

例如：Public class 代理类 implements 原目标接口 { }

CostDAO costDAO=new 代理类();//代理类去实现了原目标接口，所以没有原来类的方法

## 5.4案例：AOP的使用，模拟某些组件需要记录日志的功能

接3.3、3.4案例，想让所有的操作进行日志记录，那么按以前的方式就需要给所有Action或DAO中添加记录日志的代码，如果Action或DAO很多，那么不便于维护。而使用AOP机制，则可以很方便的实现上述功能：

step1：导入AOP需要的包：aopalliance.jar、aspectjrt.jar、aspectjweaver.jar、cglib-nodep-2.1\_3.jar

step2：在org.tarena.aop包下新建LoggerBean类，并添加logger方法用于模拟记录日志功能

public void logger(){ System.out.println("记录了用户的操作日志"); }

step3：在applicationContext.xml配置文件中，添加AOP机制

<bean id="loggerBean" class="org.tarena.aop.LoggerBean"></bean>

<aop:config>

<!--定义切入点，指定目标组件和方法。id：可任意起个名字。expression：指定哪些组件是目标，并作用在这些目标的方法上。下面表示所有Action中的所有方法为切入点-->

<aop:pointcut id="actionPointcut" expression="within(org.tarena.action.\*)" />

<!--定义方面，将loggerBean对象指定为方面组件，loggerBean从普通Bean组件升级为了方面组件-->

<aop:aspect id="loggerAspect" ref="loggerBean">

<!-- aop:before在操作前执行 aop:after操作后执行 -->

<!-- 定义通知，aop:before：指定先执行方面组件的logger方法，再执行切入点指定的目标方法。aop:after：与aop:before相反 -->

<aop:before pointcut-ref="actionPointcut" method="logger"/>

</aop:aspect>

</aop:config>

step4：执行3.3案例step3，则发现添加操作已有了记录日志功能

创建CostDAO对象 初始化CostDAO对象 记录了用户的操作日志

处理资费添加操作 利用JDBC技术实现保存资费记录

step5：执行3.4案例step3，则发现删除操作已有了记录日志功能，记得加无参构造方法！

记录了用户的操作日志 处理资费删除操作

利用Hibernate技术实现删除资费记录

* 注意事项：DeleteAction用的是构造注入，所以此处要把无参构造器再加上！因为AOP底层调用了DeleteAction的无参构造方法。不加则报错：Superclass has no null constructors but no arguments were given

## 5.5通知类型

通知决定方面组件和目标组件作用的关系。主要有以下几种类型通知：

1. 前置通知：方面组件在目标方法之前执行。
2. 后置通知：方面组件在目标方法之后执行，目标方法没有抛出异常才执行方面组件。
3. 最终通知：方面组件在目标方法之后执行，目标方法有没有异常都会执行方面组件。
4. 异常通知：方面组件在目标方法抛出异常后才执行。
5. 环绕通知：方面组件在目标方法之前和之后执行。

try{ //前置通知执行时机<aop:before>

//执行目标方法

//后置通知执行时机<aop:after-returning>

}catch(Exception e){//异常通知执行时机<aop:after-throwing>

}finally{ //最终通知执行时机<aop:after>

}//环绕通知等价于前置+后置<aop:around>

## 5.6切入点

切入点用于指定目标组件和方法，Spring提供了多种表达式写法：

1. 方法限定表达式：指定哪些方法启用方面组件。

①形式：execution(修饰符? 返回类型 方法名(参数列表) throws 异常?)

②示例：

execution(public \* \* (..))，匹配容器中，所有修饰符是public（不写则是无要求的），返回类型、方法名都没要求，参数列表也不要求的方法。

execution(\* set\*(..))，匹配容器中，方法以set开头的所有方法。

execution(\* org.tarena.CostDAO.\*(..))，匹配CostDAO类中的所有方法。

execution(\* org.tarena.dao.\*.\*(..))，匹配dao包下所有类所有方法。

execution(\* org.tarena.dao..\*.\*(..))，匹配dao包及其子包中所有类所有方法。

2）类型限定表达式：指定哪些类型的组件的所有方法启用方面组件（默认就是所有方法都启用，且知道类型，不到方法）。

①形式：within(类型) ②示例：

within(com.xyz.service.\*)，匹配service包下的所有类所有方法

within(com.xyz.service..\*)，匹配service包及其子包中的所有类所有方法

within(org.tarena.dao.CostDAO)，匹配CostDAO所有方法

* 注意事项：within(com.xyz.service..\*.\*)，为错误的，就到方法名！

1. Bean名称限定：按<bean>元素的id值进行匹配。

①形式：Bean(id值) ②示例：

bean(costDAO)，匹配id=costDAO的bean对象。

bean(\*DAO)，匹配所有id值以DAO结尾的bean对象。

1. args参数限定表达式：按方法参数类型限定匹配。

①形式：args(类型) ②示例：

args(java.io.Serializable)，匹配方法只有一个参数，并且类型符合Serializable的方法，public void f1(String s)、public void f2(int i)都能匹配。

* 注意事项：上述表达式可以使用&&、| | 运算符连接使用。

## 5.7案例：环绕通知，修改5.4案例使之动态显示所执行的操作

step1：新建opt.properties文件，自定义格式：包名.类名.方法名=操作名。在高版本MyEclipse中，切换到Properties界面，点击Add直接输入键和值，则中文会自动转为ASCII码。低版本的则需要使用JDK自带的转换工具：native2ascii.exe

#第一个为资费添加，第二个为资费删除

org.tarena.action.CostAction.execute=\u8D44\u8D39\u6DFB\u52A0

org.tarena.action.DeleteAction=\u8D44\u8D39\u5220\u9664

step2：新建PropertiesUtil工具类，用于解析.properties文件

private static Properties props = new Properties();

static{ try{ props.load(PropertiesUtil.class.getClassLoader()

.getResourceAsStream("opt.properties"));

}catch(Exception ex){ ex.printStackTrace(); } }

public static String getValue(String key){

String value = props.getProperty(key);

if(value == null){ return ""; }else{ return value; } }

step3：使用环绕通知，将5.4案例step3中的<aop:before />标签换为<aop:around />

<aop:around pointcut-ref="actionPointcut" method="logger"/>

step4：修改5.4案例step2中的LoggerBean类

public Object logger(ProceedingJoinPoint pjp) throws Throwable{//采用环绕通知，加参数

//前置逻辑

String className=pjp.getTarget().getClass().getName();//获取要执行的目标组件类名

String methodName=pjp.getSignature().getName();//获取要执行的方法名

//根据类名和方法名，给用户提示具体操作信息

String key=className+"."+methodName; System.out.println(key);

//解析opt.properties，根据key获取value

String value=PropertiesUtil.getValue(key);

//XXX用户名可以通过ActionContext.getSession获取

System.out.println("XXX执行了"+value+"操作！操作时间："+

new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(

new Date(System.currentTimeMillis())));

Object obj=pjp.proceed();//执行目标方法

//后置逻辑

return obj; }

step5：分别执行3.3案例step3和3.4案例step3，执行结果动态显示所执行的操作及时间

XXX执行了资费添加操作！操作时间：2013-08-19 20:14:47

XXX执行了资费删除操作！操作时间：2013-08-19 20:15:45

## 5.8案例：利用AOP实现异常处理，将异常信息写入文件

1. 分析：方面：将异常写入文件。切入点：作用到所有Action业务方法上

within(org.tarena.action..\*)。通知：异常通知<aop:after-throwing>。

1. 实现：step1：在org.tarena.aop包中创建ExceptionBean类

public class ExceptionBean {//模拟，将异常信息写入文件

public void exec(Exception ex){//ex代表目标方法抛出的异常

System.out.println("将异常记录文件"+ex); //记录异常信息 } }

step2：在applicationContext.xml配置文件中进行配置

<bean id="exceptionBean" class="org.tarena.aop.ExceptionBean"></bean>

<aop:pointcut id="actionPointcut" expression="within(org.tarena.action.\*)"/>

<!-- 定义方面组件，将exceptionBean指定为方面 -->

<aop:aspect id="exceptionAspect" ref="exceptionBean">

<!-- throwing：和自定的方法中的参数名相同。一定要把异常抛出来才行！

try-catch了则不行！ -->

<aop:after-throwing pointcut-ref="actionPointcut" method="exec" throwing="ex"/>

</aop:aspect>

step3：在DeleteAction的execute方法中添加异常

String str=null; str.length();

step4：执行3.3案例step3则添加操作执行正常；执行3.4案例step3则删除操作报空指针异常！显示结果：将异常记录文件java.lang.NullPointerException

# Log4j日志记录工具

## 6.1 Log4j介绍

Log4j主要用于日志信息的输出。可以将信息分级别（错误、严重、警告、调式信息）按不同方式（控制台、文件、数据库）和格式输出。

Log4j主要有以下3部分组件构成：

1. 日志器（Logger）：负责消息输出，提供了各种不同级别的输出方法。
2. 输出器（Appender）：负责控制消息输出的方式，例如输出到控制台、文件输出等。
3. 布局器（格式器，Layout）：负责控制消息的输出格式。

## 6.2 Log4j的使用

step1：引入log4j.jar

step2：在src下添加log4j.properties（定义了消息输出级别、采用哪种输出器、采用哪种布局器）

#level：大小写都可，myconsole是自己随便起的appender名字，可以写多个appender

log4j.rootLogger=debug,myconsole,myfile

#appender：可在org.apache.log4j中找自带的类

log4j.appender.myconsole=org.apache.log4j.ConsoleAppender

log4j.appender.myfile=org.apache.log4j.FileAppender

#log4j.appender.myfile.File=D:\\error.txt

log4j.appender.myfile.File=D:\\error.html

#layout：可在org.apache.log4j中找自带的类

log4j.appender.myconsole.layout=org.apache.log4j.SimpleLayout

log4j.appender.myfile.layout=org.apache.log4j.HTMLLayout

* 注意事项：级别从小到大为：debug、info、warn、error、fatal

step3：创建TestLog4j类，测试利用日志器不同的方法输出消息。

public class TestLog4j {

public static Logger logger=Logger.getLogger(TestLog4j.class);

public static void main(String[] args) {

//能显示就显示，不显示也不会影响主程序后面的运行，仅是个辅助工具

logger.debug("调试信息"); logger.info("普通信息");

logger.warn("警告信息"); logger.error("错误信息");

logger.fatal("致命信息"); } }

* 注意事项：
* 导包为org.apache.log4j.Logger。
* 若在log4j.properties中指定的级别为debug，则五种信息都会显示；若指定的级别为error，则只显示error和fatal信息。

## 6.3案例：修改5.8案例，使用Log4j记录日志

step1：继续使用6.2节step1和step2

step2：修改5.8案例step1

public class ExceptionBean {//将异常信息写入文件

Logger logger=Logger.getLogger(Exception.class);

public void exec(Exception ex){//ex代表目标方法抛出的异常

logger.error("====异常信息====");//记录异常信息

logger.error("异常类型"+ex);

StackTraceElement[] els=ex.getStackTrace();

for(StackTraceElement el:els){ logger.error(el); } } }

step3：执行3.4案例step3则删除操作报空指针异常（前提：已进行了5.8案例step3操作）！由于log4j.properties配置了两种输出方式，所以两种方式都有效。

控制台的显示结果：

XXX执行了资费删除操作！操作时间：2013-08-20 12:47:54

ERROR - ====异常信息====

ERROR - 异常类型java.lang.NullPointerException

…… …… …… ……

HTML显示结果：



# Spring注解配置

注解技术从JDK5.0推出，之后很多框架开始提供注解配置形式。Spring框架从2.5版本开始支持注解配置。注解配置的优点：简单、快捷。

## 7.1组件扫描功能

Spring可以按指定的包路径扫描内部的组件，当发现组件类定义前有一下的注解标记，会将该组件纳入Spring容器中。

1）@Component（其他组件）

2）@Controller（Action组件，负责调Service）

3）@Service（Service组件，负责调DAO，处理一些额外逻辑）

4）@Repository（DAO组件，负责访问数据库）

* 注意事项：
* 括号中的为推荐用法，上述4个注解任意用也可以，但不符合规范。
* 注解只能用在类定义前、方法定义前、成员变量定义前！

## 7.2组件扫描的使用方法

step1：在applicationContext.xml配置文件中开启组件扫描配置

<!-- 开启组件扫描，base-package指定扫描包路径。使用前提：要有xmlns:context命名空间的引入。base-package="org.tarena"这么写，则dao和action都能被扫描-->

<context:component-scan base-package="org.tarena" />

step2：在要扫描的组件的类定义前使用上述注解标记即可。例如在JdbcCostDAO类前使用

@Repository

public class JdbcCostDAO implements CostDAO {

public JdbcCostDAO(){ System.out.println("创建CostDAO对象"); }

@PostConstruct//等价于设置了init-method="方法名"属性

public void myinit(){ System.out.println("初始化CostDAO对象"); }

@PreDestroy//等价于设置了destroy-method="方法名"属性

public void mydestroy(){ System.out.println("销毁CostDAO对象"); }

…… ……

@Repository等价于原来配置文件中的：

<bean id="jdbcCostDAO" class="org.tarena.dao.JdbcCostDAO"></bean>

加上@Scope("prototype")等价于原配置文件中的：

<bean id="jdbcCostDAO" scope="prototype" class="org.tarena.dao.JdbcCostDAO"></bean>

* 注意事项：
* 上述标记将组件扫描到容器后，id属性默认是类名首字母小写。如果需要自定义id值，可以使用@Repository("自定义id值")，其他注解也同理。
* 类的命名和变量的命名要规范！首字母大写，第二个字母要小写！否则在使用框架时会有冲突或无法识别，如类名为JDBCCostDAO时无法识别它的id值：jDBCCostDAO，此时以它的类名作为id值却可识别：JDBCCostDAO。
* 默认采用singleton模式创建Bean对象，如果需要改变，可以使用

@Scope("prototype")定义。

* lazy-init="true"属性只能在<beans>根元素定义了，没有对应的注解。

step3：创建TestAnnotation类，用于测试注解

@Test //组件扫描

public void test1(){ String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);

//获取扫描到容器的Bean对象

CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDao");

costDAO.delete();//组件扫描，默认的id为类名首字母小写！且默认单例模式 }

## 7.3注入注解标记使用方法

如果容器中两个符合要求可被注入同一个组件的Bean对象，可以采用下面注解标记：

1）@Resource，默认按**类型匹配**注入（JDK自带的）。若有多个符合要求的类型，则报错：匹配不唯一，那么就需要采取按名称注入的方式，它的使用格式为：

@Resource(name="需要注入的Bean对象id值")。

2）@Autowired，默认按**类型匹配**注入（Spring提供的）。若有多个符合要求的类型，则采取按名称注入的方式，它的使用格式为：

@Autowired

@Qualifier("需要注入的Bean对象id值")

* 注意事项：注入标记在成员变量定义前，但@Resource也可以在set方法前使用！

3）案例：id为hibernateCostDao的Bean对象和id为costDao的Bean对象，都符合CostDAO接口，在CostAction组件中注入，那么此时将会报错：匹配不唯一。解决如下：

step1：修改CostActin，添加注入标记

@Controller("costAction")

@Scope("prototype")

public class CostAction {

//@Resource//将costDao注入，按类型匹配注入，JDK自带的

//@Autowired//将costDao注入，按类型匹配注入，Spring提供的

//@Resource(name="hibernateCostDao")//当有多个符合要求的类型，则按名称注入

@Autowired

@Qualifier("hibernateCostDao")//当有多个符合要求的类型，则按名称注入

private CostDAO costDAO;

public void setCostDAO(CostDAO costDAO) { this.costDAO = costDAO; }

…… …… }

step2：在TestAnnotation类，添加方法测试注入标记

@Test //注入标记测试

public void test2(){ String conf="/applicationContext.xml";

ApplicationContext ac=new ClassPathXmlApplicationContext(conf);

CostAction costAction=(CostAction)ac.getBean("costAction");

costAction.execute(); }

step3：可正常执行，如果没写注入标记则报错：NullPointerException

## 7.4 AOP注解标记使用方法

step1：在applicationContext.xml配置文件中开启AOP注解

<aop:aspectj-autoproxy /><!--之前的配置可都删除，只留根元素-->

step2：在方面组件中，使用下面注解标记：

1）首先使用@Component将组件扫描到Spring容器。

2）然后使用@Aspect将组件定义为方面组件。

3）之后定义一个空方法（方法名随便起）在方法前使用@Pointcut定义切入点表达式。

4）最后在方面组件的处理方法前使用@Around、@Before、@AfterReturning、@AfterThrowing、@After

例如：修改5.7案例step4中的LoggerBean类

@Component//将组件扫描到Spring容器

@Aspect//将该组件定义为方面组件

public class LoggerBean {

//定义切入点

@Pointcut("within(org.tarena.action..\*)")

public void mypoint(){}//主要目的是使用@Pointcut标记，id则为它的方法名mypoint

//采用环绕通知

@Around("mypoint()")//方法即为下面的方法名

public Object logger(ProceedingJoinPoint pjp) throws Throwable{

//……方法体内容没变…… } }

* 注意事项：@Pointcut注解在JDK1.7中不能识别，只能把切入点表达式写在通知中：

@Around("within(org.tarena.action..\*)")。而此用法JDK1.6也支持。

step3：再次执行3.3案例step3，则也可正常执行

step4：把6.3案例step2中的ExceptionBean修改为使用AOP注解

@Component//将组件扫描到Spring容器

@Aspect//将该组件定义为方面组件

public class ExceptionBean {

Logger logger=Logger.getLogger(Exception.class);

@Pointcut("within(org.tarena.action..\*)")

public void mypoint(){}

@AfterThrowing(pointcut="mypoint()",throwing="ex")//方法名即为下面方法的名字

public void exec(Exception ex){ //……方法体内容没变…… } }

step5：执行6.3案例step3，则正常执行，控制台显示空指针异常，异常信息也被写入HTML文件。

# Spring对数据访问技术的支持

## 8.1 Spring提供了统一的异常处理类型

1）SQLException是JDBC抛的异常。

2）org.hibernate.XXXException是Hibernate抛出的异常。

3）Spring提供的异常根类：DataAccessException，但是很多异常都被try-catch了，所以出错后看不到提示，因此要用Log4j。

## 8.2 Spring提供了编写DAO的支持类

1）DaoSupport类：JdbcDaoSupport、HibernateDaoSupport，自己写的DAO按使用的访问技术，有选择的继承它们（类似于以前写的BaseDAO类）。

2）Template类：JdbcTemplate、HibernateTemplate，封装了通用操作，如：增删改查。特殊操作，如：分页查询，则仍需要使用Hibernate原来的方式，详见8.6节。

3）继承DaoSupport类后，就可通过getJdbcTemplate()、getHibernateTemplate()方法获得对应的Template类对象，即可进行通用操作：

①update()：实现增删改查。 ②query()：实现查询多行记录。

③queryForObject()：实现查询单行记录。 ④queryForInt()：实现查询单个int值。

4）将一条记录转换成一个实体对象，需要实现Spring提供的RowMapper接口（将实体与记录间的转换写在它的实现类中），因为Spring提供的Template对象中的查询方法query()有RowMapper类型的参数。

## 8.3 Spring提供了声明式事务管理方法

基于AOP配置实现，不需要写Java代码，加注解标签即可。详情见第十章。

## 8.4 Spring框架如何使用JDBC技术

以前的DBUtil则不需要写了，交给Spring配置。此例包含注解配置和xml配置。

step1：新建一个工程，引入Spring开发包（IoC和AOP开发包）和配置文件

spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1\_3.jar

step2：引入JDBC技术相关的开发包（驱动包）

step3：根据要操作的表，编写对应的实体类。此处用Hibernate笔记5.16案例中Cost实体

step4：编写DAO接口CostDAO和实现类JdbcCostDAO（实现类继承JdbcDaoSupport，并使用其提供的getJdbcTemplate对象实现增删改查操作）

1. CostDAO接口

public Cost findById(int id); public void save(Cost cost); public void delete(int id);

public void update(Cost cost); public int count(); public List<Cost> findAll();

1. JdbcCostDAO实现类

public class JdbcCostDAO extends JdbcDaoSupport implements CostDAO{

public void delete(int id) { String sql="delete from COST\_CHANG where ID=?";

Object[] params={id}; getJdbcTemplate().update(sql,params); }

public List<Cost> findAll() { String sql="select \* from COST\_CHANG";

/\*\* 将一条记录转换成一个Cost对象，需要实现Spring提供的RowMapper接口 \*/

RowMapper mapper=new CostRowMapper();

List<Cost> list=getJdbcTemplate().query(sql,mapper); return list; }

public int count(){ String sql="select count(\*) from COST\_CHANG";

int size=getJdbcTemplate().queryForInt(sql); return size; }

public Cost findById(int id) { String sql="select \* from COST\_CHANG where ID=?";

Object[] params={id}; RowMapper mapper=new CostRowMapper();

Cost cost=(Cost)getJdbcTemplate().queryForObject(sql, params,mapper);

return cost; }

public void save(Cost cost) {

String sql="insert into COST\_CHANG(ID,NAME,BASE\_DURATION," +

"BASE\_COST,UNIX\_COST,STATUS,DESCR," +

"STARTTIME) values(COST\_SEQ\_CHANG.nextval,?,?,?,?,?,?,?)";

Object[] params={ cost.getName(), cost.getBaseDuration(), cost.getBaseCost(),

cost.getUnitCost(), cost.getStatus(), cost.getDescr(),

cost.getStartTime() };

getJdbcTemplate().update(sql,params); }

public void update(Cost cost) {

String sql="update COST\_CHANG set NAME=?,BASE\_DURATION=?," +

BASE\_COST=?,UNIX\_COST=?,STATUS=?,DESCR=?," +

"STARTTIME=? where ID=?";

Object[] params={ cost.getName(), cost.getBaseDuration(), cost.getBaseCost(),

cost.getUnitCost(), cost.getStatus(), cost.getDescr(), cost.getStartTime(),

cost.getId() };

getJdbcTemplate().update(sql,params); } }

1. 在org.tarena.entity包中创建CostRowMapper类

public class CostRowMapper implements RowMapper{//实现RowMapper接口

/\*\* rs：结果集。index：当前记录的索引。Object：返回实体对象。 \*/

public Object mapRow(ResultSet rs, int index) throws SQLException {

Cost cost = new Cost(); cost.setId(rs.getInt("ID"));

cost.setName(rs.getString("NAME"));

cost.setBaseDuration(rs.getInt("BASE\_DURATION"));

cost.setBaseCost(rs.getFloat("BASE\_COST"));

cost.setUnitCost(rs.getFloat("UNIT\_COST"));

cost.setStartTime(rs.getDate("STARTIME"));

cost.setStatus(rs.getString("STATUS")); return cost; } }

step5：将DAO组件交给Spring容器，在applicationContext.xml中进行相关配置

1）定义DAO组件的<bean>元素

<bean id="costDao" class="org.tarena.dao.impl.JdbcCostDAO">

<!-- 此处需要注入连接资源给DaoSupport，用于实例化Template对象，否则没有与数据库的连接！dataSource：代表连接池对象。此处实际是给JdbcDaoSupport注入连接，用于实例化JdbcTemplate对象。 -->

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

</bean>

1. 需要DAO的Bean注入一个dataSource对象。dataSource对象采用一个连接池构建（此处使用dbcp连接池），先引入dbcp连接池开发包（commons-pool.jar、commons-dbcp-1.2.2.jar、commons-collections-3.1.jar），再定义dataSource对象的<bean>元素。

<!-- 定义连接池Bean对象 -->

<bean id="MyDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<!-- 注入数据库连接参数 -->

<property name="url" value="jdbc:oracle:thin:@localhost:1521:dbchang"></property>

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>

<property name="username" value="system"></property>

<property name="password" value="chang"></property>

<property name="maxActive" value="20"></property><!-- 设置连接最大数 -->

<!-- 连接池实例化时初始创建的连接数 -->

<property name="initialSize" value="2"></property>

</bean>

step6：创建TestJdbcCostDAO类，用于测试xml配置，可正常执行

@Test

public void testFindAll() {//测试基于xml配置方法

String conf="/applicationContext.xml";

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

CostDAO costDAO=(CostDAO)ac.getBean("costDao");

List<Cost> list=costDAO.findAll(); System.out.println("总数："+costDAO.count());

for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); }

System.out.println("findById："+costDAO.findById(1).getName()); }

step7：使用注解配置，保留step5中的连接资源MyDataSource，而Bean组件costDao删除，并添加开启组件扫描的配置

<context:component-scan base-package="org.tarena" /><!-- 开启组件扫描 -->

step8：在JdbcCostDAO中添加注解

@Repository //id值默认为类名首字母小写

@Scope("prototype") //设置非单例模式

public class JdbcCostDAO extends JdbcDaoSupport implements CostDAO{

@Resource //将容器中的myDataSource按类型匹配注入

/\*\* 虽然继承了JdbcDaoSupport，但我们无法在别人的代码中添加注解，所以我们添加一个set方法，注意名字别用setDataSource！因为JdbcDaoSupport 有同名的set方法，且是final修饰的，所以需要稍微改变一下。 \*/

public void setMyDataSource(DataSource ds){

super.setDataSource(ds);//将注入的dataSource给DaoSupport注入 }

…… ……

step9：在TestJdbcCostDAO类添加方法，用于测试注解配置，可正常执行

@Test

public void testFindAllByAnnotation() {//测试基于注解配置方法

String conf="/applicationContext.xml";

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDAO");

List<Cost> list=costDAO.findAll(); System.out.println("总数："+costDAO.count());

for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); }

System.out.println("findById："+costDAO.findById(1).getName()); }

* 注意事项：此时由于一开始定义的id为costDao的Bean已被删除，所以TestJdbcCostDAO类中的testFindAll方法已无法执行。在测试注解配置时，可以新建applicationContext-annotation.xml文件，把注解开启、连接资源写这里，之后在测试方法中，指定它的名字即可！这样两个方法都可执行。

step10：优化，当大量Bean组件需要采取注解配置时，每个Bean都要写set方法注入dataSource连接资源，所以比较麻烦，可采取继承方式：

1. 新建JdbcBaseDAO类，并继承JdbcDaoSupport，把set方法写入

public class JdbcBaseDAO extends JdbcDaoSupport {

@Resource //将容器中的myDataSource按类型匹配注入

public void setMyDataSource(DataSource ds){//注意名字，详见step8

super.setDataSource(ds); } }

1. 删除step8中JdbcCostDAOset的set方法及其前面的注解，并继承JdbcBaseDAO

public class JdbcCostDAO extends JdbcBaseDAO implements CostDAO{ …… }

## 8.5连接池优点

1. 增强数据访问的稳定性。
2. 连接池可以将连接数控制在安全的范围内。

3）连接池中的连接对象始终与数据库保持联通状态，它的close方法被重写，不是真正的关闭，而是把连接又放回池中，避免了重复的新建连接和释放连接过程。

## 8.6 Spring框架如何使用Hibernate技术

Hibernate的主配置也不需要了，也交给Spring配置。此例包含注解配置和xml配置。

step1：新建一个工程，引入Spring开发包（IoC和AOP开发包）和配置文件

spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1\_3.jar

step2：引入Hibernate相关的开发包（Hibernate开发包+驱动）

step3：编写实体类和hbm.xml映射描述文件，此处用Hibernate笔记5.16案例中Cost实体及其Cost.hbm.xml映射文件

step4：编写DAO接口和HibernateCostDAO实现类（实现类继承HibernateDaoSupport，并使用其提供的HibernateTemplate对象实现增删改查操作）

1. CostDAO接口，与8.4节step4中1）基本相同，但额外添加一个方法

public List<Cost> findPage(int page,int pageSize);//分页查询

2）HibernateCostDAO实现类

public class HibernateCostDAO extends HibernateDaoSupport implements CostDAO {

public int count() { String hql="select count(\*) from Cost";

List list=getHibernateTemplate().find(hql);

int size=Integer.parseInt(list.get(0).toString());

/\*\* 最好先变成string再转成int。不要强转，Spring的不同版本返回值可能是long、Integer类型 \*/ return size; }

public void delete(int id) { Cost cost=findById(id);

getHibernateTemplate().delete(cost); }

public List<Cost> findAll() { String hql="from Cost";

List<Cost> list=getHibernateTemplate().find(hql); return list; }

public Cost findById(int id) {

Cost cost=(Cost)getHibernateTemplate().get(Cost.class, id); return cost; }

public void save(Cost cost) { getHibernateTemplate().save(cost); }

public void update(Cost cost) { getHibernateTemplate().update(cost); }

public List<Cost> findPage(final int page,final int pageSize) {//分页查询，参数为final

List<Cost> list=(List<Cost>)getHibernateTemplate().execute(

new HibernateCallback(){//回调函数及下面的Session如何使用详见8.7节

public Object doInHibernate(Session session)//记得改参数名

throws HibernateException, SQLException {

String hql="from Cost";

Query query=session.createQuery(hql);//使用session对象

int begin=(page-1)\*pageSize;

query.setFirstResult(begin);

query.setMaxResults(pageSize);

return query.list();

}

}

);

return list; } }

step5：将DAO组件交给Spring容器管理，在applicationContext.xml中进行相关配置

<bean id="hibernateCostDao" scope="prototype"

class="org.tarena.dao.impl.HibernateCostDAO">

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

</bean><!-- name：代表Hibernate的连接资源，该资源要么是hibernateTemplate类型，要么是sessionFactory类型（按Alt+/就会显示），我们用sessionFactory类型。ref：名字任意起，是我们配置的sessionFactory连接资源，有了该资源，getHibernateTemplate()方法才能执行 -->

step6：在applicationContext.xml中配置sessionFactory资源（相当于Hibernate的主配置）

<bean id="MySessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<!-- LocalSessionFactoryBean是Hibernate中SessionFactory的子类，我们不用写 -->

<!-- 注入数据库连接信息，此处要再配置一下dataSource数据库连接资源 -->

<property name="dataSource" ref="MyDataSource"></property><!--ref：名字任意起-->

<!-- 注入Hibernate配置参数 -->

<property name="hibernateProperties">

<props><!-- 放Spring中属性要加个前缀hibernate -->

<prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.format\_sql">true</prop>

</props>

</property>

<!-- 注入映射描述 -->

<property name="mappingResources">

<list><value>org/tarena/entity/Cost.hbm.xml</value></list>

</property>

</bean>

step7：在applicationContext.xml中配置dataSource数据库连接资源，与8.4案例step5中2）相同，dataSource对象采用一个连接池构建（此处使用dbcp连接池），先引入dbcp连接池开发包（commons-pool.jar、commons-dbcp-1.2.2.jar、commons-collections-3.1.jar）

<!-- 定义连接池Bean对象 -->

<bean id="MyDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<!-- 注入数据库连接参数 -->

<property name="url" value="jdbc:oracle:thin:@localhost:1521:dbchang"></property>

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>

<property name="username" value="system"></property>

<property name="password" value="chang"></property>

<property name="maxActive" value="20"></property><!-- 设置连接最大数 -->

<!-- 连接池实例化时初始创建的连接数 -->

<property name="initialSize" value="2"></property>

</bean>

* 注意事项：此案例的step5-7的注入关系为：dataSource对象（dbcp连接池构建的）注入给sessionFactory对象，sessionFactory对象注入给DAO（Bean）对象。

step8：创建TestHibernateCostDAO类，用于测试xml配置，可正常执行

@Test //测试基于xml配置方法

public void testFindAll() { String conf="/applicationContext.xml";

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

CostDAO costDAO=(CostDAO)ac.getBean("hibernateCostDao");

List<Cost> list=costDAO.findAll(); System.out.println("总数："+costDAO.count());

for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); }

System.out.println("findById："+costDAO.findById(1).getName()); }

step9：使用注解配置，新建applicationContext-annotation.xml配置文件（此处使用的为8.4案例step9中注意事项说的方式），在其中添加step6中的sessionFactory资源，添加step7中的dataSource数据库连接资源（不要step5中的Bean组件hibernateCostDao），并添加开启组件扫描的配置

<context:component-scan base-package="org.tarena" /><!-- 开启组件扫描 -->

step10：在HibernateCostDAO中添加注解

@Repository("hibernateCostDao")

@Scope("prototype")

public class HibernateCostDAO extends HibernateDaoSupport implements CostDAO {

@Resource //setSessionFactory名字用不了是final的，同理说明见8.4案例step8

public void setMySessionFactory(SessionFactory sf){

super.setSessionFactory(sf);//将注入的sessionFactory给HibernateDaoSupport传入

}

…… …… ……

step11：在TestHibernateCostDAO类添加方法，用于测试注解配置，可正常执行

@Test //测试基于注解配置方法

public void testFindAllByAnnotation() { String conf="/applicationContext-annotation.xml";

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

CostDAO costDAO=(CostDAO)ac.getBean("hibernateCostDao");

List<Cost> list=costDAO.findAll(); System.out.println("总数："+costDAO.count());

for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); }

System.out.println("findById："+costDAO.findById(1).getName()); }

step12：在TestHibernateCostDAO类添加方法，用于测试分页查询

@Test //分页查询

public void testFindByPage() { String conf="/applicationContext-annotation.xml";

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

CostDAO costDAO=(CostDAO)ac.getBean("hibernateCostDao");

List<Cost> list=costDAO.findPage(2, 4);System.out.println("总数："+costDAO.count());

for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); } }

## 8.7 Spring+Hibernate如何使用Session、Query等对象

1）方式一：利用HibernateDaoSupport提供的getSession()方法：

没用到延迟加载的API，那么用这个方式简单些。但是有延迟加载的API，则会出现问题：session关闭早了，页面可能获取不到数据；不关闭吧，一但出了方法体则关不上了！而多次访问数据库后，就发现没结果，因为连接数用完了。

Session session=getSession();

……利用session进行一些操作……

session.close();//注意，一定要释放！

* 注意事项：getHibernateTemplate中的API都有释放操作，所以自己不用再写。

2）方式二：利用HibernateTemplate.execute()方法，以回调函数方式使用。这种方式不用担心方式一出现的问题，session的关闭由HibernateTemplate统一管理。

getHibernateTemplate().execute(

new HibernateCallback(){//实现该接口的doInHibernate方法

public Object doInHibernate(Session session)

throws HibernateException, SQLException{

//回调函数中使用session，可见8.6案例step4

}

}

);

## 8.8 Spring框架和Struts2整合应用

step1：新建一个工程，引入Spring开发包（IoC和AOP开发包）、配置文件和Struts2的五个基本核心包

spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1\_3.jar/xwork-core.jar/struts-core.jar/ognl.jar/freemarker.jar/commons-fileupload.jar

step2：在org.tarena.action包中新建HelloAction

private String name;//output

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String execute(){ name="Chang"; return "success"; }

step3：新建applicationContext.xml文件并配置，将Action或DAO等组件交给Spring容器

<bean id="helloAction" scope="prototype" class="org.tarena.action.HelloAction"></bean>

step4：引入Struts2和Spring整合的开发包：struts-spring-plugin.jar。该开发包的作用为：当Struts2请求过来时，Action对象将交给整合包去Spring容器获取，即Struts2不再产生Action对象，详解介绍可看第九章。

step5：新建struts.xml文件，并配置<action>，将class属性与Spring容器中<bean>元素的id属性保持一致。（整合包利用class值当作id标识去Spring容器获取Bean对象）

<struts>

<package name="spring05" extends="struts-default">

<!--关键：利用struts-spring-plugin.jar去Spring容器寻找Bean对象

利用class属性当作id值去Spring容器获取，详细介绍看第九章 -->

<!-- 原来为class="org.tarena.action.HelloAction"，现在改为id值：helloAction -->

<action name="hello" class="helloAction"><result>/hello.jsp</result></action>

</package>

</struts>

step6：在WebRoot目录中，新建hello.jsp和index.jsp

1. hello.jsp

<body style="font-size:30px;font-style:italic;">${name }你好！ </body>

1. index.jsp

<a href="hello.action">Hello示例</a>

step7：在web.xml中添加前端控制器

<filter>

<filter-name>StrutsFilter</filter-name>

<filter-class>

org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter

</filter-class>

</filter><!-- 前端控制器 -->

<filter-mapping>

<filter-name>StrutsFilter</filter-name>

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

</filter-mapping>

step8：部署，启动服务器，准备测试。但是发现报错：

You might need to add the following to web.xml:

<listener>

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

</listener>

step9：在web.xml中添加ContextLoaderListener组件配置（可以在启动服务器时，实例化Spring容器），但它的默认查找路径为：/WEB-INF/applicationContext.xml，所以要指定路径（src中），否则还是会报错：java.io.FileNotFoundException

<!-- 指定Spring配置文件位置和名称 -->

<context-param><!-- 下面的名字是ContextLoaderListener里约定好的，必须这么写 -->

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

<param-value>classpath:applicationContext.xml</param-value><!--classpath代表src-->

</context-param>

<!-- 在服务器启动时，实例化Spring容器，在中间过程无法插入实例化代码，因为都是自动的，所以没地方写！因此要配置listener，启动服务器则创建Spring容器。 -->

<listener>

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

</listener>

* 注意事项：在前端控制器前添加！！否则无法生效。

step10：在浏览器中输入[http://localhost:8080/Spring05\_Struts2t/index.jsp](http://localhost:8080/Spring05_test/index.jsp)进行测试

## 8.9案例：采用SSH结构重构资费管理模块

修改Hibernate笔记5.16案例中的资费管理模块（原功能采用Struts2+Hibernate结构）。

采用SSH结构需要追加以下步骤：

step1：引入Spring开发包（IoC和AOP）和applicationContext.xml配置文件

spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1\_3.jar

====================重构Spring+Hibernate====================

step2：编写DAO组件，采用Spring+Hibernate方式实现，如：新建SpringHibernateCostDAOImpl

public class SpringHibernateCostDAOImpl extends HibernateDaoSupport implements CostDAO{ public void delete(int id) throws DAOException { Cost cost=findById(id);

getHibernateTemplate().delete(cost); }

public List<Cost> findAll() throws DAOException { String hql="from Cost";

List<Cost> list=getHibernateTemplate().find(hql); return list; }

public List<Cost> findAll(final int page,final int rowsPerPage) throws DAOException {

List list=(List)getHibernateTemplate().execute(

new HibernateCallback(){ @Override

public Object doInHibernate(Session session)

throws HibernateException, SQLException {

String hql="from Cost";

Query query=session.createQuery(hql);

int start=(page-1)\*rowsPerPage;//设置分页查询参数

query.setFirstResult(start);//设置抓取记录的起点，从0开始（第一条记录）

query.setMaxResults(rowsPerPage);//设置抓取多少条记录

return query.list();//按分页参数查询 } }

);

return list; }

public Cost findById(Integer id) throws DAOException {

Cost cost=(Cost)getHibernateTemplate().load(Cost.class, id);

return cost;//有延迟问题！要配置web.xml ，详见step10 }

public Cost findByName(String name) throws DAOException {

String hql = "from Cost where name=?"; Object[] params = {name};

List<Cost> list = getHibernateTemplate().find(hql,params);

if(!list.isEmpty()){ return list.get(0);

}else{ return null; } }

public int getTotalPages(int rowsPerPage) throws DAOException {

String hql="select count(\*) from Cost";//类名

List list=getHibernateTemplate().find(hql);

int totalRows=Integer.parseInt(list.get(0).toString());

if(totalRows%rowsPerPage==0){ return totalRows/rowsPerPage;

}else { return (totalRows/rowsPerPage)+1; } }

public void save(Cost cost) throws DAOException { cost.setStatus("1");

cost.setCreaTime(new Date(System.currentTimeMillis()));

getHibernateTemplate().save(cost); }

public void update(Cost cost) throws DAOException {

getHibernateTemplate().update(cost); } }

step3：在Spring容器的配置文件中定义配置

1）将DAO扫描到Spring容器

<context:component-scan base-package="com.tarena.netctoss" /><!--开启组件扫描-->

2）配置sessionFactory

<!-- 注意：这里的id值不能自定义了！必须写sessionFactory，因为配置了

OpenSessionInViewFilter后（session关闭延迟到JSP解析后，详见step10），默认找的就是这个名字 -->

<bean id="sessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<!-- LocalSessionFactoryBean是Hibernate中SessionFactory的子类，我们不用写-->

<!-- 注入数据库连接信息 -->

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

<!-- 注入Hibernate配置参数 -->

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect"><!-- 放Spring中属性加个前缀hibernate -->

org.hibernate.dialect.OracleDialect

</prop>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.format\_sql">true</prop>

</props>

</property>

<!-- 注入映射描述 -->

<property name="mappingResources">

<list>

<value>com/tarena/netctoss/entity/Cost.hbm.xml</value>

<value>com/tarena/netctoss/entity/Admin.hbm.xml</value>

</list>

</property>

</bean>

3）引入dbcp开发包（连接池），定义DataSource

<!-- 定义连接池Bean对象 -->

<bean id="MyDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<!-- 注入数据库连接参数 -->

<property name="url" value="jdbc:oracle:thin:@localhost:1521:dbchang"></property>

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>

<property name="username" value="system"></property>

<property name="password" value="chang"></property>

<!-- 设置连接最大数 -->

<property name="maxActive" value="20"></property>

<!-- 设置连接池实例化时初始创建的连接数 -->

<property name="initialSize" value="1"></property>

</bean>

4）将sessionFactory给DAO注入

@Repository @Scope("prototype")//排版需要，这里写一行了

public class SpringHibernateCostDAOImpl extends HibernateDaoSupport implements CostDAO{

@Resource

public void setMySessionFactory(SessionFactory sf){//注意名字问题

super.setSessionFactory(sf); }

…… …… ……

step4：测试DAO组件，在TestCostDAO中添加方法，可正常执行

@Test //Spring+Hibernate

public void testFindAllByPageBySpring() throws Exception{

String conf="applicationContext.xml";

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

CostDAO costDao = (CostDAO)ac.getBean("springHibernateCostDAOImpl");

List<Cost> list = costDao.findAll(1,5);

for(Cost c : list){ System.out.println(c.getId()+" "+c.getName()); } }

====================重构Spring+Struts2====================

step5：（\*）将资费模块中所有Action定义到Spring容器（step3已开启组件扫描），如果使用了DAO，采用注入方式使用（此处是新建了ListCostActionSpring类，与原来的ListCostAction相同）

@Controller @Scope("prototype")//排版需要，这里写一行了

public class ListCostActionSpring { @Resource

private CostDAO costDAO; … … }

step6：引入struts-spring-plugin.jar插件包

step7：（\*）修改Action的struts配置，将class属性修改为容器中Action组件的id值

<action name="list" class="listCostActionSpring">

<result name="success">/WEB-INF/jsp/cost/cost\_list.jsp</result>

</action><!-- class可写包名.类名，即和以前的方式相同，详情见9.3节-->

step8：在web.xml中添加ContextLoaderListener配置，用于实例化Spring容器，详细说明见8.8案例step9

<!-- 指定Spring配置文件位置和名称 -->

<context-param><!--默认在WEB-INF下查找applicationContext.xml的，不指定则报错-->

<param-name>contextConfigLocation</param-name><!--名字必须这么写 -->

<param-value>classpath:applicationContext.xml</param-value><!--classpath表示src-->

</context-param>

<!-- 在服务器启动时，实例化Spring容器对象 -->

<listener>

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

</listener>

step9：测试SSH，发现修改资费时，无法回显数据！这是因为Hibernate中的load方法为延迟加载，而session被早关闭，所以JSP页面获取不到数据。为了支持Hibernate延迟加载的使用，在web.xml中可以配置Spring提供的OpenSessoinInViewFilter。将Session关闭动作推迟到JSP解析之后。

step10：在web.xml中添加OpenSessoinInViewFilter，利用它控制Session关闭

<!-- 追加OpenSessionInViewFilter，不需要自己写了，Spring提供了！作用：将Template中的session关闭动作推迟到jsp解析之后 -->

<filter><!--注意：配置的Filter顺序，要在Struts控制器Filter之前配置才能生效！-->

<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>/\*</url-pattern>

</filter-mapping>

step11：测试SSH，发现增、删、改会报错，这是由于配置OpenSessionInViewInterceptor后产生的（查询正常），详细说明见下面的注意事项

org.springframework.dao.InvalidDataAccessApiUsageException:

Write operations are not allowed in read-only mode (FlushMode.NEVER/MANUAL):

Turn your Session into FlushMode.COMMIT/AUTO or remove 'readOnly' marker from transaction definition.

step12：在applicationContext.xml配置文件中添加AOP事务控制，使用注解的方式，代码及详情见10.3节2）

step13：分别给Action添加事务注解（可类定义前，也可execute或自定义方法前）

1）AddCostAction、DeleteCostAction、UpdateCostAction中添加，事务类型详见10.3节

@Transactional(propagation=Propagation.REQUIRED)

2）DetailCostAction、ListCostActionSpring、ValidateCostNameAction中添加

@Transactional(readOnly=true,propagation=Propagation.REQUIRED)

* 注意事项：
* 注意配置的Filter顺序！该Filter需要在Struts控制器Filter之前配置才能生效！！
* 当配置OpenSessionInViewFilter后，Spring容器中SessionFactory组件的id值必须为sessionFactory，不能再随便起了。
* 当配置OpenSessionInViewFilter后，会默认将session操作置成readOnly状态，此时需要添加AOP事务才能执行增、删、改，否则报错（查询没问题）。
* 由于Hibernate笔记5.16案例中用到了5.17节4）step2的自定义拦截器OpenSessionInViewInterceptor，并按step3修改了struts-cost.xml，此时会有冲突：资费模块的权限验证会失效（无权限的也能进入）。当项目为SSH结构时，用Spring提供的OpenSessionInViewFilter就行了，之前定义的OpenSessionInViewInterceptor不用了！把struts-cost.xml中自定义的拦截器的声明、引用、默认全局都删掉！包cost继承netctoss-default，权限验证即可生效。其他配置也都继承netctoss-default（之前为了方便测试所以都继承了json-default）。

# 整合开发包struts-spring-plugin.jar

## 9.1 Struts2创建对象的方式

Struts2-->ObjectFactory-->StrutsObjectFactory

## 9.2 struts-spring-pligin.jar创建对象的方式

在它的包中有个StrutsSpringObjectFactory，而它的配置文件struts-pligin.xml，则把Struts2中的ObjectFactory指定成了StrutsSpringObjectFactory，这样一来，当Struts2请求过来时，Action对象将交给整合包去Spring容器获取，即Struts2不再产生Action对象，可在该包的struts-plugin.xml中查看。

<bean type="com.opensymphony.xwork2.ObjectFactory" name="spring"

class="org.apache.struts2.spring.StrutsSpringObjectFactory" />

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

## 9.3 struts-spring-plugin.jar的内部实现

通过它的内部代码，可以发现该整合开发包获取Bean对象时，有两种方式：

try{//第一种 见原理图1

Object action = ac.getBean(class属性);//id值

}catch(){//第二种 见原理图2

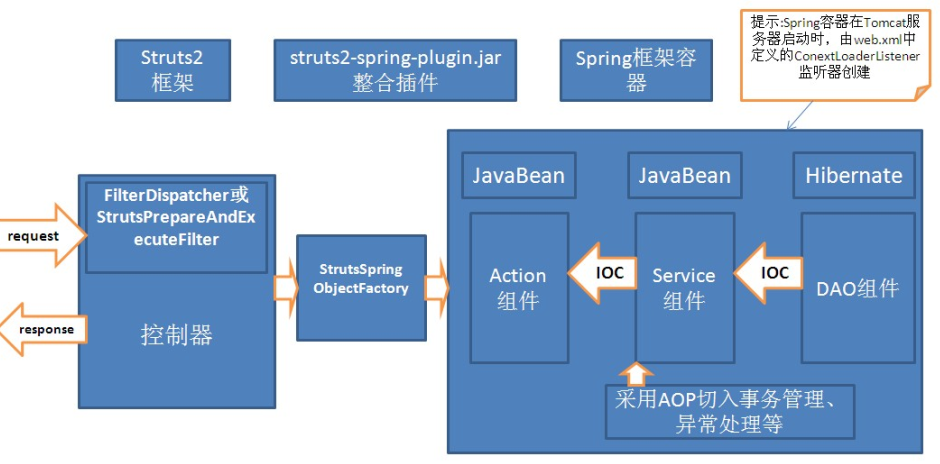
Class class = Class.forName(class属性);//step1：包名.类名，利用反射机制生成Action

Object action = class.newInstance();

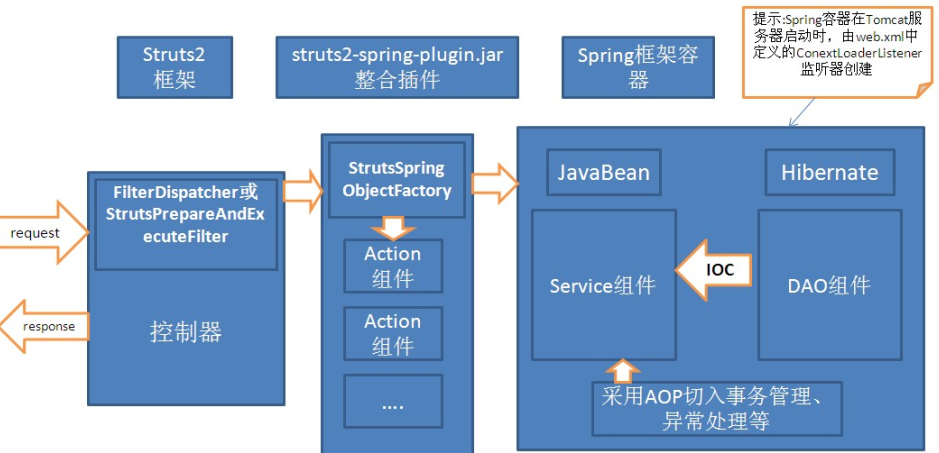
//step2：然后自动注入，注入规则见9.7节！

}

## 9.4原理图1



## 9.5原理图2



## 9.6注意事项

第一种方式：创建Action由Spring容器负责。

第二种方式：创建Action由插件负责，与第一种相比Action脱离了Spring容器，所以不能用AOP机制了！也不能用事务管理了！所以，为了使Action只负责调用而不涉及业务逻辑，开发中一般会有个Service组件，把业务逻辑、AOP机制、事务管理都给Service组件。

## 9.7注入规则

9.3节中catch块的注入规则：

1）组件中，不加@Resource（默认）是按名称匹配，即属性名和id值一致才可以。

2）组件中，如果添加了@Resource，则按类型匹配。

# Spring的事务管理

Spring提供以下两种方式管理事务。

## 10.1声明式事务管理（基于配置方式实现事务控制）

1）以8.9案例为例，在applicationContext.xml配置文件中使用xml方式配置事务：

<!--事务管理配置-->

<!--定义事务管理Bean（用于管理事务），不用我们写了，直接用Spring提供的类-->

<bean id="txManager"

class="org.springframework.orm.hibernate3.HibernateTransactionManager">

<!-- 注入session资源 -->

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

</bean>

<!-- 定义方面和通知，直接使用Spring提供的命名空间：xmlns:tx=http://.../schema/tx -->

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

<!--可以指定目标对象中不同方法采用不同的事务管理机制。注意：name要根据目标对象中的方法名写。read-only="true"：只读事务，只能查询。propagation="REQUIRED"为默认值其他可取的值见10.3节 -->

<tx:attributes> <!--注释中的是给DAO添加事务，但不好！没注释的是给Action加-->

<!-- <tx:method name="save" propagation="REQUIRED"/>

<tx:method name="update" propagation="REQUIRED"/>

<tx:method name="delete" propagation="REQUIRED"/>

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

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

<tx:method name="\*" propagation="REQUIRED"/> -->

<!-- action中execute方法都采取REQUIRED，而REQUIRED把默认的只读操作模式改了，此时可增、删、改、查。由于NetCTOSS项目没加Service组件，所以把事务加到Action上，否则就是给Service加的。而不加在DAO中是因为有些操作可能要执行多个DAO共同完成一项功能，如果加在DAO中，那么里面的一个方法就是一个事务（会提交），那么当该功能中途出现问题，则之前的操作将无法回滚了！ -->

<tx:method name="execute" propagation="REQUIRED"/>

<!--其他没有考虑到的按默认的事务管理机制值-->

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

</tx:attributes>

</tx:advice>

<!--定义切入点，AOP切入--><!--proxy-target-class="true"：表示强制采用CGLIB方式生成代理类，若不写则容器会根据是否有接口，而去选择用哪种技术产生代理类，但这样就会有问题！如：Action继承BaseAction，而BaseAction实现某个接口，那么容器选择的代理类技术而产生的代理Action是没有原来Action中的方法的！若是作用在DAO上，则可不写 -->

<aop:config proxy-target-class="true"><!-- 类型匹配 -->

<aop:pointcut id="actionPointcut" expression="within(com.tarena.netctoss.action..\*)" />

<!-- 将切入点和通知结合 -->

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

</aop:config>

2）以8.9案例为例，在applicationContext.xml配置文件中使用注解方式配置：

step1：定义HibernateTransactionManager（事务管理）Bean组件

<bean id="txManager"

class="org.springframework.orm.hibernate3.HibernateTransactionManager">

<!-- 注入session资源 -->

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

</bean>

step2：开启事务的注解配置

<!-- 开启事务注解配置 --> <!-- 指明让txManager来管理事务 -->

<tx:annotation-driven proxy-target-class="true" transaction-manager="txManager"/>

step3：然后在业务组件的类定义前或方法中使用@Transactional注解即可，例如：

①AddCostAction，在类定义前使用（对类中除了get/set方法外的所有方法，都使用相同的事务管理）

@Transactional(propagation=Propagation.REQUIRED)

public class AddCostAction extends BaseAction{ …… }

②DeleteCostAction，在方法前使用

@Transactional(propagation=Propagation.REQUIRED)

public String execute() throws DAOException{ …… }

③UpdateCostAction，在类定义前使用

@Transactional(propagation=Propagation.REQUIRED)

public class UpdateCostAction extends BaseAction { …… }

④ListCostAction，在方法前使用

@Transactional(readOnly=true,propagation=Propagation.REQUIRED)

public String execute() throws Exception { …… }

* 注意事项：如果将Action当作目标，需要在<tx:annotation-driven>添加proxy-target-class="true"属性，表示不管有没有接口，都采用CGLIB方式生成代理类。

## 10.2编程式事务管理（基于Java编程实现事务控制），不推荐用！

主要是利用transactionTemplate的execute()方法，以回调方式将多个操作封装在一个事务中。

## 10.3 Spring中常用的事务类型

1）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，也是默认值。

2）SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。

3）MANDATORY：支持当前事务，如果当前没有事务，就抛出异常。

4）REQUIRES\_NEW：新建事务，如果当前存在事务，把当前事务挂起。

5）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。

6）NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

7）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REQUIRED类似的操作。拥有多个可以回滚的保存点，内部回滚不会对外部事务产生影响。只对DataSourceTransactionManager有效。

# Spring的MVC

## 11.1 Spring MVC的体系结构

1. 控制器（两种）：①DispatcherServlet（等价于Struts2中的Filter）

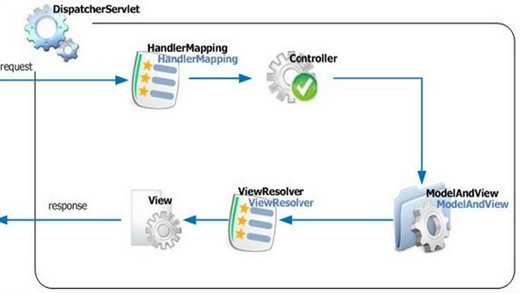
②Controller（等价于Struts2中的Action）

2）映射处理器：HandlerMapping（完成请求和Controller之间的调用，等价于Struts2中的ActionMapping）

3）模型视图组件：ModelAndView（封装了模型数据和视图标识）

4）视图解析器：ViewResolver（等价于Struts2中的Result）

5）视图组件：主要用JSP



## 11.2 Spring MVC的工作流程

1. 客户端发送请求，请求到达DispatcherServlet主控制器。
2. DispatcherServlet控制器调用HandlerMapping处理。
3. HandlerMapping负责维护请求和Controller组件对应关系。HandlerMapping根据请求调用对应的Controller组件处理。
4. 执行Controller组件的业务处理，需要访问数据库，可以调用DAO等组件。
5. Controller业务方法处理完毕后，会返回一个ModelAndView对象。该组件封装了模型数据和视图标识。
6. Servlet主控制器调用ViewResolver组件，根据ModelAndView信息处理。定位视图资源，生成视图响应信息。
7. 控制器将响应信息给用户输出。

## 11.3案例：简易登录（基于XML配置，不推荐使用）

由于此案例没有用到JDBC、Hibernate等访问数据库的技术，所以AOP包可以不用导入！

step1：导入spring-webmvc.jar包

step2：导入Spring的IoC开发包（spring.jar、commons-logging.jar）

step3：配置web.xml

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

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

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

</init-param><!-- 让容器从指定的src目录下查找applicationContext.xml文件-->

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>\*.do</url-pattern><!-- 不能写/\*了，影响太广，自定义一种请求形式 -->

</servlet-mapping>

step4：在/WEB-INF/jsp中新建login.jsp和ok.jsp

1. login.jsp

<h1>Spring MVC 登录</h1><font size="5" color="red">${error }</font>

<form action="login.do" method="post">

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

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

<input type="submit" value="登录"/></form>

1. ok.jsp

<h1>欢迎${user }，登录成功！</h1>

step5：在WebRoot下新建index.jsp

<a href="toLogin.do">spring mvc(xml)</a>

step6：在org.tarena.controller包中新建ToLoginController类，并实现Controller接口

public class ToLoginController implements Controller {//必须实现Controller，并重写方法

@Override //记得改改参数名字

public ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response) throws Exception {//默认执行的业务方法

ModelAndView mv=new ModelAndView("login");//调用login.jsp，指定视图名称

return mv; } }

step7：在org.tarena.controller包中新建LoginController类，并实现Controller接口

public class LoginController implements Controller {//必须实现Controller，并重写方法

@Override //记得改改参数名字

public ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response) throws Exception {//默认执行的业务方法

String name=request.getParameter("username");

String pwd=request.getParameter("password");

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

if("chang".equals(name) && "123".equals(pwd)){//简单模拟，不访问数据库了

map.put("user", name); return new ModelAndView("ok",map); }

map.put("error", "用户名或密码错误");

return new ModelAndView("login",map);//指定视图名称 } }

step8：新建applicationContext.xml文件，并进行配置

<!-- 定义handlermapping，即定义请求和Controller的映射信息 -->

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="toLogin.do">toLoginController</prop>

<prop key="login.do">loginController</prop>

</props>

</property>

</bean>

<!-- 定义视图解析器，负责根据ModelAndView信息调用View组件 -->

<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"></property><!-- 声明前缀 -->

<!-- 因返回的ModelAndView对象仅有个名字，所以要定义前后缀 -->

<property name="suffix" value=".jsp"></property><!-- 声明后缀 -->

</bean>

<!-- 定义Controller -->

<bean id="toLoginController" class="org.tarena.controller.ToLoginController"></bean>

<bean id="loginController" class="org.tarena.controller.LoginController"></bean>

step9：部署，测试

## 11.4案例：修改11.3案例（基于注解配置，推荐使用）

由于此案例没有用到JDBC、Hibernate等访问数据库的技术，所以AOP包可以不用导入！

step1：导入spring-webmvc.jar包

step2：导入Spring的IoC开发包（spring.jar、commons-logging.jar）

step3：配置web.xml，与9.3案例step3相同

step4：在/WEB-INF/jsp中新建login.jsp和ok.jsp

1. login.jsp

<h1>Spring MVC 登录</h1><font size="5" color="red">${error }</font>

<!-- 模拟请求有多级，即user/login.do，action写user/login.do为相对路径，出现叠加问题；写绝对路径需要加“/”，同时也要写应用名。详细说明见10.2节-->

<form action="/Spring06\_MVC2/user/login.do" method="post">

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

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

<input type="submit" value="登录"/></form>

2）ok.jsp，与9.3案例step4中2）相同

step5：在WebRoot下新建index.jsp

<a href="toLogin.do">spring mvc(annotation)</a>

step6：新建applicationContext.xml文件，并进行配置

<!-- 开启组件扫描 -->

<context:component-scan base-package="org.tarena" />

<!-- 定义映射处理器，采用注解AnnotationMethodHandlerAdapter指定映射 -->

<bean id="annotationMapping"

class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

</bean>

<!-- 定义视图解析器，负责根据ModelAndView信息调用View组件（JSP）-->

<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"></property><!-- 声明一个前缀 -->

<property name="suffix" value=".jsp"></property><!-- 声明一个后缀 -->

</bean>

step7：在org.tarena.controller包中新建ToLoginController类，并使用注解

@Controller //将组件扫描到Spring容器

@Scope("prototype")

public class ToLoginController {//不用再实现接口了，就是写个普通的类

@RequestMapping(value="/toLogin.do",method=RequestMethod.GET)

public String execute(){//写个最简单的方法,返回类型也可写ModelAndView

return "login";//返回视图名称 } }

step8：在org.tarena.controller包中新建LoginController类，并使用注解

@Controller //将组件扫描到Spring容器

@Scope("prototype")

@RequestMapping("/user/\*")//当请求有多级，即有共同前缀，可写类定义前

public class LoginController { //不用去实现Controller接口了，就是写个普通的类

//@RequestMapping(value="/login.do",method=RequestMethod.POST)//没有共同前缀

@RequestMapping(value="login.do",method=RequestMethod.POST)//有共同前缀

public String execute(User user,Model model){//Model是可以传到下一个页面的

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

if("chang".equals(user.getUsername()) && "123".equals(user.getPassword())){

map.put("user", user.getUsername());

model.addAttribute("user",user.getUsername()); return "ok"; }

model.addAttribute("error", "用户名或密码错误");

return "login"; } }

* 注意事项：
* @RequestMapping注解：value属性指定请求；method属性指定请求提交方式。
* Controller中业务方法可以定义成以下格式：

1）public String f1(){}

2）public ModelAndView f1(){}

3）public String f1(HttpServletRequest request){}//需要request就加上，不需要可不写

4）public String f1(HttpServletRequest request,HttpServletResponse response){}

5）public String f1(User user){} //自定义实体类，属性与表单中的提交名一致  
6）public String f1(Model model){} //org.springframework.ui.Model中的

7）public String f1(User user,Model model){}

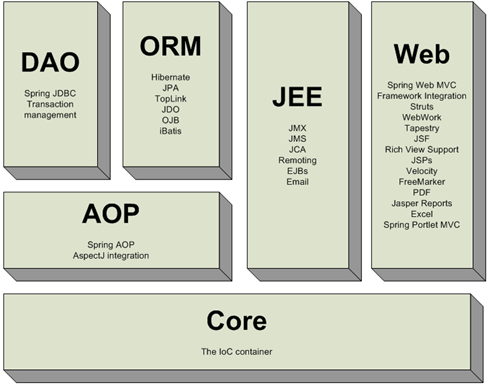
step9：step8中execute方法用到了实体User，所以在org.tarena.entity包下创建User实体

private String username; private String password;//和表单提交名一样 ……get/set方法

step10：部署，测试

# 其他注意事项

## 12.1 Spring的核心模块



## 12.2表单中action属性的相对、绝对路径问题

例如9.4案例step4中的login.jsp，当请求有多级时（项目中不需要创建什么user目录，仅是通过请求实现的分类而已）：

<form>标签中的action属性，如果写“user/login.do”，即相对路径，会出现叠加问题：

<form action="user/login.do" method="post">

因为它相对的是当前请求地址，比如当前请求页面地址为：

http://localhost:8080/Spring06\_MVC2/user/login.do

那么，第一次向user/login.do发请求没问题，但第二次发请求则出现叠加问题！（点两次登录就能出现该问题）

http://localhost:8080/Spring06\_MVC2/user/user/login.do

即，第二次请求把后面的login.do又替换为了user/login.do，而之前的user仍在，则出现叠加问题。

解决方式：写绝对路径，但要加“/”，同时也要写应用名，即：

<form action="/Spring06\_MVC2/user/login.do" method="post">

## 12.3用SSH重构NetCTOSS项目模块的步骤

1. 了解原功能的处理流程，例如资费模块：/cost/add.action-->AddCostAction.execute

-->CostDAO.save-->list.action

2）重构CostDAO（Spring+Hibernate）

①追加Spring开发包和配置文件。

②追加Cost类（已存在）和Cost.hbm.xml。

③在Spring的sessionFactory中加载hbm.xml。

④基于HibernateDaoSupport和HibernateTemplate编写CostDAO实现组件。

1. CostDAO的Spring配置

①将CostDAO扫描到Spring容器。

②将容器中dataSource注入到sessionFactory，然后sessionFactory注入到DAO组件。

③测试DAO。

4）修改AddCostAction

①将Action扫描到Spring容器。

②采用注入方式使用DAO组件对象。

③在业务方法上定义@Transactional事务注解。

5）修改AddCostAction的struts配置

①将class属性改成与扫描到Spring容器后的Action组件id值。

1. 检查共通的操作是否完成

①是否引入struts-spring-plugin.jar。

②是否在web.xml中添加ContextLoaderListener。

③是否开启了组件扫描配置。

④是否开启了注解事务配置。

⑤是否配置dataSource、sessionFactory。