Spring笔记（2014）

————李耀

【Spring框架】

概念：Spring是一个开源的控制反转(IOC)和面向切面(AOP)的容器框架，它的主要目的是简化企业开发。

1.【控制反转(IOC)】

所谓控制反转，就是应用本身不负责依赖对象的创建和维护，依赖对象的创建和维护是由外部容器负责的。这样控制权就由应用转移到了外部容器，控制权的转移就是所谓控制反转。

2.【依赖注入(DI)】

所谓的依赖注入就是指：在运行期间，由外部容器动态的将依赖对象注入到组件当中。

【依赖注入由两种方式】

1.通过构造器参数，让容器把创建好的依赖对象注入进PersonServiceBean

2.当然也可以使用setter方法进行注入

————>为什么要使用Spring呢？它有哪些好处？

1.降低组件之间的耦合度，实现软件各层之间的解耦。

2.可以使用容器提供的众多服务，比如：事务管理服务，消息服务等等。当我们使用事务管理服务的时候，开发人员就不需要手工控制事务了，也不需要处理复杂的事务传播。

3.容器提供单例模式支持，开发人员不需要再自己编写实现代码。

4.容器提供AOP技术，利用它很容易实现如权限拦截，运行期监控等功能。

5.容器提供的众多辅佐类，使用这些类能够加快应用的开发，如：JdbcTemplate,HibernateTemplate。

6.Spring对于主流的应用框架提供了集成支持，如：集Hibernate,JPA,Struts等，这样更便于应用的开发。

————>【轻量级和重量级】之分：

其实划分一个应用是轻量级框架还是重量级框架，主要看它使用了多少服务，使用的服务越多，容器要为普通java对象做的工作就越多，必然会影响到应用的发布时间或者是运行性能。对于Spring容器，它提供了很多的服务，但是这些服务并不是默认为应用打开的，应用需要某种服务，还需要指明使用该服务，如果应用使用的服务很少，如：只使用了Spring核心服务，那么我们可以认为此时应用属于轻量级的，如果应用使用了Spring的大部分服务，这时应用就属于重量级的。目前EJB容器就因为它默认为应用提供了EJB规范中所有的功能，所以EJB就属于重量级的。

————————————————————————————————

【开发Spring的流程】

1.下载并拷贝jar包必须要有的jar文件；dist/spring.jar

lib/jakarta-commons/commons-logging.jar如果使用到了切面编程，还需要： lib/aspectj/aspectjweaver.jar和aspectjrt.jar和lib/cglib/cglib-nodep-2.1\_3.jar。如果使用到了JSR-250中的注解，如@Resource/@PostConstruct/@PreDestory,还需要

lib/j2ee/common-annotations.jar

2.建项目(Spring既可用在普通java项目中，也可用在web项目中)，顺便将对应jar文件加到构建路径里去。

3.创建并书写xml配置文件模板。

4.实例化Spring容器（测试Spring环境搭建是否成功）

————>运行JUnit实例化Spring容器的方法，控制台没报错及说明环境搭建成功!

【**实例化Spring容**器】：——>【两种方式】

方式1).在类路径下寻找配置文件来实例化容器。——推荐使用

ApplicationContext ctx = new ClassPathXmlApplicationContext

(new String[]{"beans.xml"});

方式2).在文件系统路径下寻找配置文件来实例化容器。——>不常用

ApplicationContext ctx = new FileSystemXmlApplicationContext

(new String[]{"d:\\beans.xml"});

**注意**：ApplicationContext只是一个接口，通过该接口去引用Spring的容器实例ClassPathXmlApplicationContext，Spring的实例提供了很多的接口，我们可以使用该实例的任何一个接口都可以引用Spring的容器实例。当然，除了提供接口，还提供了很多抽象类，这些抽象类也可以引用Spring的容器实例。

当配置文件只有一个的时候，我们可以直接传字符串进去即可，如果是由多个配置文件，就需要传个数组进去。

【Spring的配置文件可以指定多个，可以通过Spring数组传入】

————————————————————————————————

Spring环境搭建成功，我们就可以直接使用。业务bean也就可以直接交给Spring容器管理了。

————————————————————————————————

【Spring的三种实例化bean的方式】

三种实例化Bean的方式：

1.使用类构造器实例化bean【用的比较多】

2.使用静态方法工厂实例化bean

3.使用实例工厂方法实例化

————————————————————————————————

【配置Spring管理的bean作用域】

Bean的作用域：如果没有指定作用域，默认是.Singleton作用域范围，即就是scope=“Singleton”

【.Singleton】在每个Spring IOC的容器中，一个bean定义只有一个对象实例，默认情况下会在容器启动时初始化bean，但我们可以指定bean节点的lazy-init=“true”来延迟初始化bean，这时候，只有第一次获取bean才会初始化bean！

例如：<bean id="xxx" class="xxxx" lazy-init="true"/>如果想要对所有的bean都应用延迟初始化，可以根节点beans设置default-lazy=“true”

如下：<beans default-lazy=“true”/>

【.prototype】每次从容器获取bean都是新的对象。

【.request】

【.session】

【.global session】

——————————————————————————————————

【Spring管理的bean的生命周期】

指定Bean的初始化方法和销毁方法

【注意】在bean的作用域为scope=“Singleton”的时候，该bean会在Spring启动的时候就实例化bean类；而在bean的作用域为scope=“prototype”的时候，bean会在调用getBean()方法的时候实例化bean类。

另外，我们也可以在bean的配置文件中使用lazy-init="false"/"true"来指定bean类的初始化时期。

（前者false代表是在Spring启动时初始化，后者true代表是在调用getBean()方法的时候初始化bean。配置文件中的init-method="xxx"代表在实例化bean类的同时调用bean类的xxx方法进行初始化。

配置文件中的destory-method="xxx"代表在关闭Spring容器的同时调用bean类的xxx方法进行销毁。

——————————————————————————————————

【编码剖析Spring依赖注入的原理】

在Spring使用的过程中，我们通过控制反转将依赖对象交给Spring容器管理，Spring容器将依赖对象注入到组件内部。那么在Spring中如何将依赖对象注入到组件内部呢？内部实现的过程是怎么样的？

【依赖注入由三种方式】

————>1).通过构造器参数，让容器把创建好的依赖对象注入进PersonServiceBean构造器参数方式注入 介绍：

<bean id="orderdao" class="cn.itcast.service.OrderDaoBean" />

<bean id="orderService"class="cn.itcast.service.OrderServiceBean">

<constructor-arg index="0" type="java.lang.String"value="xxx"/> ————>【构造器注入】

<constructor-arg index="1" type="cn.itcast.dao.OrderDao"ref="orderdao"/>

………………

</bean>

————>2).当然也可以使用setter方法(属性的set方法)进行注入。

注意，如果我们使用属性进行依赖对象的注入，那么我们就必须给该属性提供set方法(一般也会提供get方法)。setter方式依赖注入 详细介绍：

【注入依赖对象】

1.基本类型对象注入

<bean id="orderService"class="cn.itcast.service.OrderServiceBean">

<constructor-arg index="0" type="java.lang.String" value="xxx"/> ————>【构造器注入】

<property name="name" value="sssssssssssssss"/> ————>【属性setter方法注入】

<property name="id" value="8932838"/>

</bean>

2.集合类型对象的注入————Spring容器集合类型的装配(集合不一样，装配方式不一样)

<property name="sets">

<set>

<value>第一个</value>

<value>第二个</value>

<value>第三个</value>

</set>

</property>

………………………………

3，注入其他bean

【方式一】——ref方式

<bean id="orderdao" class="cn.itcast.service.OrderDaoBean" />

<bean id="orderService"class="cn.itcast.service.OrderServiceBean">

<property name="orderDao" ref="orderdao" />

</bean>

【方式二】——内部bean方式

使用内部bean，但是该bean不能被其他bean使用(用内部bean方式代替了ref属性引用)

<bean id="orderService"class="cn.itcast.service.OrderServiceBean">

<property name="orderDao">

<bean class="cn.itcast.service.OrderDaoBean" />

</property>

</bean>

——————>方式一和方式二相比，外部bean可以被多个其他的bean所使用，而内部bean不能被其他bean使用。

方式一例子：如何将PersonDaoBean注入进PersonServiceBean当中？(采用属性的set方法)直接采用接口对象，不用去管实现类怎么样，实现了服务层和Dao层彻底的解耦。

【注意】还需要在Spring中为personDao属性注入PersonDaoBean对象——配置xml（见beans.xml文件中依赖对象注入部分）

——>见PersonDaoBean,PersonServiceBean,beans.xml文件

————>3).使用Field注入(用于注解方式进行装配注入)

要想使用注解的方式进行依赖对象的装配：

第一步：

就必须要引入两个命名空间以及它的schema：

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

http://www.springframework.org/schema/context

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

因为这样才可以打开使用注解方式的配置项<context:annotation-config/>

第二步：

打开配置项<context:annotation-config/>，该配置项是在context命名空间下的。

第三步：

同时要使用注解，还需要引入Spring容器相关的jar文件：lib/j2ee/common-annotations.jar

该配置项背后到底做了哪些事情呢？

答：实际上这个配置项背后是【注册了专门针对注解进行解析处理的处理器】，每个处理器都对应于一些注解。注解本身其实和xml文件一样，都是用来配置的，之所以能够干活儿，是因为背后有这些处理器在进行处理操作。

【总结】注入依赖对象可以采用手工装配，也可以采用自动装配，在实际应用开发过程中，建议使用手工装配，因为自动装配可能发生未知情况，开发人员无法预知最终的装配结果。

1.【手工装配依赖对象】————>建议使用

这种方式装配依赖对象又有两种实现方式：

1.在xml配置文件中，通过在bean节点下进行配置(包括1.属性注入，2.构造器参数注入)

2.在java代码中使用注解方式进行装配，但还需要在xml文件中配置相关信息(引入名称空间，打开配置项,导入jar包)

注意：@Autowired和@Resource两种注解方式的区别：

1. @Autowired 默认按照类型装配，也可以通过加上@Qualifier("persondaoxxx")，将其改为按照名称进行装配操作。

@Autowired中有一个属性required，默认值是true，代表该注释作用

的字段属性personDao必须要有注入值，否则会报错若required=false，

则说明该字段值为null.(很少用到)。

2.@Resource 默认按照名称装配，当找不到与名称匹配的bean时，才会按照类型装配。

注意【注解可以放在属性上面，也可以放在属性的set方法上面。都可以注入对象】

对比xml配置文件和注解装配依赖对象：

xml的缺点，需要书写的内容太多了；而注解是我们推荐使用的，可以简化装配依赖对象动作。

2.【自动装配依赖对象】————>不建议使用

这种装配方式是由容器帮我们自动进行装配的，不需要人为的参与，只需要在bean节点中设置一个属性即可：

<bean id=" " class=" " autowire="byType" />

autowire属性取值如下：

byType：按照类型进行装配，可以根据属性的类型，在容器中寻找跟该类型匹配的bean，如果发现多个，那么将会抛出异常，如果没有找到，那么该属性值就为null。

byName：按照名称进行装配，可以根据属性的名称，在容器中寻找跟该名称相同的bean，如果没有找到，那么该属性值就为null。

constructor与byType的方式相似，不同之处在于它应用于构造器参数，如果在容器中没有找到与该构造器参数类型一致的bean，那么就会抛出异常。

autodelect:通过bean类的自省机制(introspection),来决定是使用constructor还是byType方式进行自动装配。如果发现默认构造器，那么将使用byType方式进行装配。

————————————————————————————————

【让Spring自动扫描和管理bean】

在一个稍大的系统中，可能会有成百上千的bean需要交给Spring容器管理，从而就都需要在xml配置文件中进行配置，不言而喻，这样的话，xml显得非常臃肿，不利于查找和管理bean。

【但是】

Spring2.5为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component,@Service,@Controller, @Repository注解的类，并把这些类纳入进Spring容器中进行管理，它的作用和在xml文件中使用bean节点配置组件是一样的。各个注解的作用对象(只是为了方便与Spring容器扫描，目前它们之间没什么特别注意事项，也可以混用，不过不建议混用)

@Component：表示泛指组件，当组件不好归类的时候，我们可以使用该竹节进行标注。

@Service： 用于标注业务层组件。

@Controller： 用于标注控制层组件(如：struts中的action)。

@Repository 用于标注数据访问层组件(即就是DAO组件)。

要想通过在classpath自动扫描方式把组件纳入Spring容器中管理，就必须进行如下配置：

第一步：

就必须要引入两个命名空间以及它的schema：

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

http://www.springframework.org/schema/context

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

第二步：

打开组件扫描的配置项：<context:component-scan base-package=""/>

该配置项背后注册了很多个处理器，这些处理器都会对一些注解进行解析，其中就包括了采用注解方式进行依赖注入的配置项<context:annotation-config/>中的所有处理器，所以，以后只需要加上组件扫描的这个配置项即可。

【注意事项】

1. bean名称问题

此时通过自动扫描机制来将组件纳入到Spring容器中管理，不像之前用bean节点进行配置了，那么怎么才能知道bean的名称是什么呢？

——>当注解没有指定bean的名称时，默认情况下，bean的名称就是注解对应类的简单类名(首字母小写)

2. bean作用域问题

默认注解的类(即就是bean)都是单例范围的scope=“Singleton”，如果想要修改其作用范围，就可以在对类进行注解标注的时候,加上@scope(“prototype”)另一个注解,声明该bean目前处于prototype作用域。

3. bean初始化和销毁的方法问题注解形式，如何像配置bean节点方式那样去指定bean的初始化和销毁方法呢？

——>在对应的初始化和销毁方法上面加上对应的注解即可，初始化方法【@PostConstruct】 销毁方法【@PreDestroy】

————————————————————————————————

【AOP面向切面】

1.使用JDK中的Proxy技术实现AOP功能

2.使用CGLIB实现AOP功能与AOP概念解释

——————>详情请见AOP项目程序及注释介绍

【使用Spring进行面向切面(AOP)编程】

【注意】

Spring提供了【两种切面使用方式】，实际工作中我们可以选用其中一种。

1. **基于注解方式进行AOP开发。**

【配置步骤】

1. 导入AOP编程所需要的jar文件

2. 要进行AOP编程首先我们要在Spring的配置文件中引入AOP命名空间。

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

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

3. 首先要打开(启动)对@Aspectj注解的支持:<aop:aspectj-autoproxy />

4. 开始使用AOP编程……

【开发内容】

1.首先定义一个切面

2.在切面里面定义切入点(切入点,即就是要对业务bean里面的哪些方法进行拦截)

3.定义通知(通知，即就是我们在拦截到这些方法之后要做的工作是哪些)

——————>详情请见SpringAOP项目程序及注释介绍(JUnit测试有点问题)

2. **基于XML配置方式进行AOP开发**。

步骤：

1. 首先把要用作切面的普通java类交给Spring容器管理。(该普通java类没有任何的AOP注解)

2. 再通过<aop:config>aop的配置标签对aop进行配置:

首先配置引用这个切面，然后在这个切面底下可以定义一些切入点(含名称id="mycut")、前置通知等通知(要引用切入点的名称pointcut-ref="mycut"，再交给切面里的某个方法)。

例子：

<bean id="aspetbean"class="cn.itcast.service.MyInterceptor"/>

<aop:config>

<aop:aspect id="asp" ref="aspetbean">

<aop:pointcut id="mycut"expression="execution(\* cn.itcast.service..\*.\*(..))"/>

<aop:before pointcut-ref="mycut"method="doAccessCheck"/>

<aop:after-returning pointcut-ref="mycut"method="doAfterReturning"/>

<aop:after-throwing pointcut-ref="mycut"method="doAfterThrowing"/>

<aop:after pointcut-ref="mycut"method="doAfter"/>

<aop:around pointcut-ref="mycut"method="doBasicProfiling"/>

</aop:aspect>

</aop:config>

【切入点定义里面的AOP的表达式语言语法】

【例子】@Pointcut("execution(\* cn.itcast.Service..\*.\*(..))")

1.execution:代表执行的意思，即就是在执行到业务方法的时候，我要进行拦截。

2.第一个\*：代表返回值的类型，\*代表任意类型。

3.cn.itcast.Service..：代表包名(即就是要对哪个包底下的类进行拦截，后面的".."代表对其子包下的类也要实施拦截)

4.后面的\*：代表要被拦截的那个类，\*代表所有类。

5.最后一个\*：代表要被拦截的方法，\*代表所有的方法。

6.后面的(..)：代表方法参数可任意(可有可无，可多可少)

【表达式语言特殊需求】

要求返回值类型是String类型：execution(java.lang.String cn.itcast.Service..\*.\*(..))要求拦截所有的非void类型的方法：execution(!void cn.itcast.Service..\*.\*(..))要求输入参数的第一个类型是String的，后面不管有没有参数：execution(\* cn.itcast.Service..\*.\*(java.lang.String,..))要求要对Service包底下的所有类，包括子包底下的所有类都进行拦截：execution(\* cn.itcast.Service..\*.\*(..))

【另外】

我们定义完了切入点的表达式之后，它就会为满足这个表达式里面的所有对象都创建代理对象。如果Spring发现被拦截的类实现了接口，那么它就会采用JDK提供的Proxy动态代理技术来创建代理对象;如果Spring发现被拦截的类未实现了接口，那么它就会采用CGlib这种方式来创建代理对象。

——————————————————————————————————

【搭建和配置Spring与jdbc整合的环境】

使用Spring+JDBC集成步骤如下：

第一步：配置数据源（导入数据源jar文件、数据库驱动、Spring相关jar文件）

<!-- 【数据源配置】 -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="org.gjt.mm.mysql.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/itcast?useUnicode=true&amp;characterEncoding=UTF-8" />

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

<property name="password" value="624963768" />

<!-- 连接池启动时的 -->

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

<!-- 连接池的最大值 -->

<property name="maxActive" value="500" />

<!-- 最大空闲值，当经过一个高峰时期后，连接池可以慢慢将已经用不到的链接慢慢释放一部分，一直减少到maxIdle为止 -->

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

<!-- 最小空闲值，当空闲的链接数少于阀值的时候，连接池就会预申请一些链接，以免洪峰到来时来不及申请。 -->

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

</bean>

【注意】

以上数据源相关配置信息可以统一放在类路径底下的一个属性文件里面 例如:jdbc.properties;然后在xml配置文件里面加上<context:property-placeholder location="jdbc.properties"/>来引入该属性文件;再分别在各自的value位置通过${name}来调用属性文件中的对应值即可。

第二步：配置事务，在配置事务时，需要在XML配置文件中引入用于声明事务的命名空间。

【事务的配置方式有两种】：

———>1. **基于注解方式来配置事务**

1).指定jdbc数据源的事务管理器

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/> <!-- 将定义好的数据源注入进来 -->

</bean>

2).引入tx事务的命名空间：

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

http://www.springframework.org/schema/tx

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

3).打开事务的配置项：<tx:annotation-driven transaction-manager="txManager" />

【Spring提供的声明式的事务】

目前这个业务bean并没有交给Spring管理，因为它没有用注解标注。那么上面针对于实体bean的操作语句，都是在各自的事务中运行的，如果某个方法(比如 delete方法)有两条操作语句，那么这两条语句就都会分别在各自的事务中执行，没法儿保证这两个操作会在同一个事务中执行。这时候就会出现一些问题！！！那么为了保证这两条语句在同一个事务中执行，我们应该使用容器给我们提供的声明式的事务：

@Transactional，对该业务bean进行标注，目的是为了明确告诉Spring容器，这个业务bean是要受到Spring容器的事务进行管理的，这样的话，该业务bean底下的所有的方法在执行前都会打开一个事务，在方法执行完成之后，就会结束这个事务。即就是：声明式的事务默认会使得一个方法就是一个事务，同一方法中的不同语句都会在该事务下执行。

【声明式的事务管理中事务是如何被管理的？】

【另外】采用声明式的事务管理，那么事务是如何被管理的呢？在执行一个方法之前打开事务，方法执行完成之后关闭事务，那么什么时候决定事务提交，什么时候决定事务回滚呢？Spring容器在执行事务的时候(即就是一个方法执行的过程中)，默认如果碰到了运行期例外(异常)，即就是在方法内部抛出了异常信息，那么事务就会回滚。——>称其为unchecked

如果碰到了异常，即就是在方法上声明了(例外)异常，那么事务就不会回滚。——>称其为checked

【当然】我们也可以改变Spring这种事务规则，在对应方法头上采用注解的方式进行声明：

@Transactional(rollbackFor = Exception.class) 设置回滚

@Transactional(noRollbackFor = RuntimeException.class)设置不回滚。

【事务的传播行为】

【另外】Spring的声明式事务管理都会默认在每个业务方法执行前都要打开一个事务，而在业务方法结束后关闭该事物。打开和关闭事务本身对系统的性能也是有影响的。对于有些业务方法是不需要事务管理的(比如：获取数据的方法)，那么我们就需要注明一下该方法是不需要事务的，我们采用注解的方式对方法进行声明，即就是：@Transactional(propagation=Propagation.NOT\_SUPPORTED)

【注明：事务的传播行为】

1.REQUESTED(默认)：

业务方法需要在一个事务中运行，如果方法运行时，就已经处在一个事务中，那么直接加入到该事务中即可，否则，为自己新创建一个事务。

2.NOT\_SUPPORTED：

声明方法不需要事务，如果方法没有关联到一个事务，容器就不会为它开启事务，如果方法在一个事务中被调用，该事务会被挂起，在方法调用结束后，原先的事务才会恢复执行。

3.REQUESTMEW：

属性表明;不管是否存在事务，业务方法总会为自己创建一个新的事务，如果方法已经运行在一个事务中，那么原先的事务就会被挂起，新的事务会被创建，直到方法执行结束后，新事物才会算结束，原先的事务不会恢复执行。

4.MANDATORY：

该属性指定业务方法只能在一个已经创建好的事务中执行，业务方法不能发起自己的事务，如果业务方法在没有事务的环境下调用，容器就会抛出例外(异常)。

5.SUPPORTS：

这一事务属性表明，如果业务方法在某个事务的范围内被调用，则方法称为该事务的一部分；如果业务方法在某个事务的范围外被调用，则方法在没有事务的环境下执行。

6.NEVER：

指定业务方法绝对不能在事务范围内执行，如果业务方法在某个事务内执行，容器就会抛出例外(异常)，只有业务方法没有关联到任何事务，才能正常执行。

7.NESTED：

如果一个活动的事务存在，则运行在一个嵌套的事务中，如果没有活动事务，则按照默认的REQUESTED属性执行，它使用了一个单独的事务，这个事务拥有多个可以回滚的保存点，内部事务的回滚不会对外部事务造成影响，它只对事务管理器： DataSourceTransactionManager起效。

———>2. **基于XML配置方式来配置事务**

1).使用XML方式配置事务：就不需要打开事务的配置项：<tx:annotation-driven transaction-manager="txManager" />

2).在配置文件中：1.配置AOP 2.配置事务通知

<!-- 【配置AOP执行拦截】 -->

<aop:config>

<!-- 1.定义切入点 -->

<aop:pointcut expression="execution(\* cn.itcast.Service..\*.\*(..))" id="transactionPointcut"/>

<!-- 2.引入事务通知(调用切入点，引用事务通知) -->

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

</aop:config>

<!-- 【配置事务通知】 -->

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

<tx:attributes>

<!-- 对方法前缀以get开头的：只读且不应用事务 -->

<tx:method name="get\*" read-only="true" propagation="NOT\_SUPPORTED"/>

<!-- 其余所有的方法：均使用默认的事务行为(REQUESTED) -->

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

</tx:attributes>

</tx:advice>

第三步：写业务bean模块，并将其交给Spring管理(XML文件中配置bean)

——————————————————————————————————

【Spring+Hibernate+Struts整合】

SSH整合，建议先整合Spring+Hibernate，再去整合Struts。下边列出了整合开发时，Spring、Hibernate和Struts所需要的jar文件：

第一步：整合Spring2.5 和 Hibernate3.0

1.Hibernate相关jar文件

D:\IdeProjectManager\SSH-jar\hibernate\lib所有jar

(除了commons-logging.jar不要，因为大多框架都有这个jar文件，避免jar冲突)

2.Spring相关jar文件

D:\IdeProjectManager\SSH-jar\spring2.5\lib所有jar

3.另外还需要数据库驱动jar文件

4.在类路径底下建一个Spring的配置文件:beans.xml

(打开组件扫描的配置项、配置数据源、集成Hibernate、打开事务的配置项等等)。

5.编写实体类，并映射其配置文件————>测试（表自动生成）

6.编写业务bean 以及业务方法 ————>测试（对数据库操作是否成功）

第二步：再将Struts2整合进来

1.Struts相关jar文件以及集成Spring的插件jar

D:\IdeProjectManager\SSH-jar\Struts2\lib

(避免jar文件冲突，删掉同名jar文件中版本比较低的)

2.比较重要了【配置web.xml文件】：

1.首先，在web.xml文件中指定Spring的配置文件，并对Spring容器进行实例化。

<!-- 1.【指定Spring的配置文件】默认是从web的根目录寻找配置文件，我们可以通过Spring提供的classpath:前缀指定从类路径寻找 -->

<context-param>

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

<!-- 多个配置文件，使用逗号隔开即可：...,classpath:abc.xml -->

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

</context-param>

<!-- 2.【对Spring容器进行实例化】,实例化后的容器实例放在了application范围 -->

<listener>

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

</listener>

2.在web.xml文件中配置Struts2，启动Struts2框架

<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>

3.编写action类和页面，最后测试。

【特别注意】：

目前action类交给Spring管理，是属于默认的【单例范围】，该范围只在初始化的时候被创建实例，也就是说，action实例只被创建一次！而每当struts2接收一个请求时，会创建一个新的action对象(实例)，如果说将action定义为单例范围，那么就与Struts2的设计理念相违背！要想在即把action交给Spring容器管理，又与Struts2的设计相吻合，那么就需要在创建action的时候，加上注解：@Scope("prototype")将action的作用域设为”原型模式“prototype，如果不设置的话，那么该action就是线程不安全的。因为单例模式下，该action中的参数有可能被其他请求所修改的。当然，如果action中没有指定与用户相关的参数数据等，也可以不用设置action的作用域，不过还是建议大家每次都将其

设置为”原型模式“prototype，以防万一。

+++++++++++++++++++++++++++++++++++++++++

三种实例化bean的方式：

**1.beans.xml**

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

http://www.springframework.org/schema/context

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

<!-- 要想使用注解，就必须要打开配置项,这样也就可以将针对注解的这些处理器注册到Spring容器中 -->

<!-- <context:annotation-config/> -->

<!-- 要想通过在classpath自动扫描方式把组件纳入Spring容器中管理，就必须打开组件扫描的配置项： -->

<!-- base-package代表要扫描哪个包底下的类以及其子包底下的类！ -->

<context:component-scan base-package="cn.itcast"/>

<!-- 【怎么将业务bean交给Spring容器管理呢?】 -->

<!--

该标签用于配置要交给Spring容器管理的bean类。

属性id和name都是为该bean取名称，但是id不能有非法字符(受xml解析)，name可以有。

属性class指要交给Spring管理的bean类

【此时bean类就配置好了，它就可以由Spring容器帮我们来自动创建和维护了

（内部实现方法很复杂，不必深究，包括dom4j读取配置文件，反射技术实例化bean等）

我们要用该bean类，就可以直接从Spring容器中获取就可以了】

-->

<!--+++++++++++++++【通过在classpath自动扫描方式把组件纳入Spring容器中管理】+++++++++++++++-->

<!-- ++++++++++++++++【传统xml文件配置】++++++++++++++++++++ -->

<!-- 【依赖注入】为personDao属性注入PersonDaoBean对象 -->

<!-- 依赖注入位置1 -->

<!-- <bean id="persondaoxxx" class="cn.itcast.dao.impl.PersonDaoBean" /> -->

<!-- 【三种实例化bean的方式】 -->

<!-- 第一种实例化bean的方式【用的比较多】 -->

<!-- <bean id="personService" class="cn.itcast.service.impl.PersonServiceBean"

lazy-init="false" init-method="init" destroy-method="destory"> -->

<!-- 依赖注入位置2(调用位置1的id给ref用) -->

<!--

<property name="personDao" ref="persondao"/>

<property name="name" value="sssssssssssssss"/>

<property name="id" value="90"/>

集合对象的装配、注入

<property name="sets">

<set>

<value>第一个</value>

<value>第二个</value>

<value>第三个</value>

</set>

</property>

<property name="list">

<list>

<value>1</value>

<value>2</value>

<value>3</value>

</list>

</property>

<property name="properties">

<props>

<prop key="key1">value1</prop>

<prop key="key2">value2</prop>

<prop key="key3">value3</prop>

</props>

</property>

<property name="map">

<map>

<entry key="key-1" value="value-1"/>

<entry key="key-2" value="value-2"/>

<entry key="key-3" value="value-3"/>

</map>

</property>

</bean>

-->

<!-- 第二种实例化bean的方式 -->

<!-- <bean id="personService2" class="cn.itcast.service.impl.PersonServiceBeanFactory"

factory-method="createPersonServiceBean"/> -->

<!-- 第三种实例化bean的方式（分两步：1.先实例化工厂2.再通过工厂实例化bean） -->

<!-- <bean id="PersonServiceBeanFactory" class="cn.itcast.service.impl.PersonServiceBeanFactory2"/>

<bean id="personService3" factory-bean="PersonServiceBeanFactory" factory-method="createPersonServiceBean"/> -->

</beans>

**2. PersonDao**

package cn.itcast.dao;

//接口

public interface PersonDao {

public abstract void add();

}

**3. PersonDaoBean**

package cn.itcast.dao.impl;

import org.springframework.stereotype.Repository;

import cn.itcast.dao.PersonDao;

//依赖对象注入的演示（如何将PersonDaoBean注入进PersonServiceBean当中？(采用属性的set方法)）

@Repository("abc") //用于Spring识别自动扫描,括号内容是对应的bean名称

public class PersonDaoBean implements PersonDao {

public void add(){

System.out.println("执行PersonDaoBean中的add()方法");

}

}

**4. PersonServiceBean**

package cn.itcast.service.impl;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

import javax.annotation.Resource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Service;

import cn.itcast.dao.PersonDao;

import cn.itcast.services.PersonService;

//实现类(实例化bean的方式一：使用类构造器实例化bean)

@Service //用于Spring识别自动扫描

public class PersonServiceBean implements PersonService {

/\*将PersonDaoBean注入进PersonServiceBean当中(通过属性的set方法注入)

直接采用接口对象，不用去管实现类怎么样，实现了服务层和Dao层彻底的解耦。

【注意】还需要在Spring中为personDao属性注入PersonDaoBean对象——配置xml

\*/

//使用@Resource注解，默认是按照名称装配的，如果找不到该名称，就寻找该类型进行装配(自动检索类型)

/\*@Resource\*/

/\*使用@Autowired注解，默认按照类型装配,也可以通过@Qualifier("persondaoxxx")，将其改为按照名称进行装配操作。

@Autowired中有一个属性required，默认值是true，代表该注释作用的字段属性personDao必须要有注入值，否则会报错

若required=false，则说明该字段值为null\*/

/\*@Autowired @Qualifier("persondaoxxx")\*/

private PersonDao personDao; //使用注解方式装配依赖对象（注解也可以放在属性的set方法上面）

private String name;

private Integer id;

//集合类型的装配（注入集合类型）

private Set<String> sets = new HashSet<String>();//set集合

private List<String> list = new ArrayList<String>();//list集合

private Properties properties = new Properties();//properties集合

private Map<String,String> map = new HashMap<String,String>();//map集合

public Map<String, String> getMap() {

return map;

}

public void setMap(Map<String, String> map) {

this.map = map;

}

public Properties getProperties() {

return properties;

}

public void setProperties(Properties properties) {

this.properties = properties;

}

public List<String> getList() {

return list;

}

public void setList(List<String> list) {

this.list = list;

}

public Set<String> getSets() {

return sets;

}

public void setSets(Set<String> sets) {

this.sets = sets;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public PersonDao getPersonDao() {

return personDao;

}

public void setPersonDao(PersonDao personDao) {

this.personDao = personDao;

}

@PostConstruct //Ejb3里面初始化方法的注解，在这儿可以用来指定初始化的方法是哪个。

public void init(){

System.out.println("打开资源等操作。。。");

}

public PersonServiceBean(){

System.out.println("我被初始化了。。。");

}

/\* (non-Javadoc)————>面向接口编程，将save方法抽取成接口

\* @see cn.itcast.service.impl.PersonService#save()

\*/

@Override

public void save(){

System.out.println("注进来的值：name="+name+" id="+id);

personDao.add();//直接使用注入进来的对象调用其方法

}

@PreDestroy //Ejb3里面销毁方法的注解，在这儿可以用来指定销毁的方法是哪个。

public void destory(){

System.out.println("关闭资源等操作。。。");

}

}

**5. PersonServiceBeanFactory**

package cn.itcast.service.impl;

//实现类(实例化bean的方式二：使用静态方法工厂实例化bean)

public class PersonServiceBeanFactory {

//用来创建bean对象的工厂方法

public static PersonServiceBean createPersonServiceBean(){

return new PersonServiceBean();

}

}

**6. PersonServiceBeanFactory2**

package cn.itcast.service.impl;

//实现类(实例化bean的方式三：使用实例工厂方法来实例化bean)

public class PersonServiceBeanFactory2 {

//用来创建bean对象的工厂方法

public PersonServiceBean createPersonServiceBean(){

return new PersonServiceBean();

}

}

**7. PersonService**

package cn.itcast.services;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

//接口

public interface PersonService {

public Set<String> getSets();

public List<String> getList();

public Properties getProperties();

public Map<String, String> getMap();

public abstract void save();

}

**8. SpringTest**

package JUint.test;

import org.junit.BeforeClass;

import org.junit.Test;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.itcast.dao.PersonDao;

import cn.itcast.services.PersonService;

public class SpringTest {

@BeforeClass

public static void setUpBeforeClass() throws Exception {

}

//【专门实例化Spring容器（启动Spring容器）的方法】

@Test

public void instanceSpring() {

/\*这里使用第一种方式：在类路径下寻找配置文件的方式来实例化容器

\* ApplicationContext只是一个接口，通过该接口去引用Spring的容器实例ClassPathXmlApplicationContext

\* Spring的实例提供了很多的接口，我们可以使用该实例的任何一个接口都可以引用Spring的容器实例。

\* 当日内，除了提供接口，还提供了很多抽象类，这些抽象类也可以引用Spring的容器实例。

\* 当配置文件只有一个的时候，我们可以直接传字符串进去即可，如果是由多个配置文件，就需要传个数组进去。

\*/

AbstractApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

//获取交给Spring管理的bean类对象(Spring内部已经创建好了该对象，直接获取即可)

//通过该bean类的接口对其进行调用,得到bean对象之后就可以调用器方法了

/\*PersonService personService = (PersonService)ctx.getBean("personService");

personService.save();

System.out.println("===========set==========");

//遍历set集合

for(String value : personService.getSets()){

System.out.println(value);

}

System.out.println("===========list==========");

//遍历list集合

for(String value : personService.getList()){

System.out.println(value);

}

System.out.println("===========properties==========");

//遍历properties集合

for(Object key : personService.getProperties().keySet()){

System.out.println(key+"="+personService.getProperties().getProperty((String)key));

}

System.out.println("===========map==========");

//遍历map集合

for(String key : personService.getMap().keySet()){

System.out.println(key+"="+personService.getMap().get(key));

}\*/

//自动扫描方式(如果注解没有标注bean名称，那么bean名就是注解对应类的简单类名)

PersonService personService = (PersonService)ctx.getBean("personServiceBean"); //注解未标注bean名称

PersonDao personService2 = (PersonDao)ctx.getBean("abc"); //注解已标注bean名称

//打印出来了泪的路径信息，说明获取到了该对象，即就是扫描成功！

System.out.println(personService);

System.out.println(personService2);

ctx.close();//关闭Spring容器

}

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

SpringAndJDBC

**1.beans.xml**

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

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

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-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/aop

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

http://www.springframework.org/schema/tx

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

">

<!-- 【数据源配置】(采用dbcp数据源) -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="org.gjt.mm.mysql.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/SpringAndJDBC?useUnicode=true&amp;characterEncoding=UTF-8" />

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

<property name="password" value="624963768" />

<!-- 连接池启动时的链接数 -->

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

<!-- 连接池中最大的链接数 -->

<property name="maxActive" value="500" />

<!-- 最大空闲值，当经过一个高峰时期后，连接池可以慢慢将已经用不到的链接慢慢释放一部分，一直减少到maxIdle为止 -->

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

<!-- 最小空闲值，当空闲的链接数少于阀值的时候，连接池就会预申请一些链接，以免洪峰到来时来不及申请。 -->

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

</bean>

<!-- 【配置事务】 -->

<!-- 1.指定jdbc数据源的事务管理器 -->

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/> <!-- 将定义好的数据源注入进来 -->

</bean>

<!-- 2.打开事务的配置项(前提要引入命名空间) -->

<tx:annotation-driven transaction-manager="txManager" /> <!-- 将定义好的数据源事务管理器注入进来 -->

<!-- 把业务bean交给Spring管理 -->

<bean id="personServiceBean" class="cn.itcast.service.impl.PersonServiceBean">

<property name="dataSource" ref="dataSource" /> <!-- 将数据源注入到属性中去 -->

</bean>

</beans>

**2. Person**

package cn.itcast.bean;

//实体类

public class Person {

private int id;

private String name;

public Person(String name) {

super();

this.name = name;

}

public Person() {}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**3. PersonService**

package cn.itcast.service;

import java.util.List;

import cn.itcast.bean.Person;

//接口

public interface PersonService {

//保存person

public void save(Person person);

//更新person

public void update(Person person);

//获取person

public Person getPerson(int id);

//获取所有的person

public List<Person> getAllPerson();

//根据id来删除person

public void deletePerson(int id);

}

**4. PersonRowMapper**

package cn.itcast.service.impl;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import cn.itcast.bean.Person;

//【设置回调类】：需要实现回调接口RowMapper，并实现其回调方法

public class PersonRowMapper implements RowMapper {

//if(rs.next()) 【外部在调用毁掉函数的时候已经执行过next判断了，里面直接使用rs即可】

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

Person person = new Person(rs.getString("name"));

person.setId(rs.getInt("id"));

return person;

}

}

**5. PersonServiceBean**

package cn.itcast.service.impl;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.transaction.annotation.Propagation;

import org.springframework.transaction.annotation.Transactional;

import cn.itcast.bean.Person;

import cn.itcast.service.PersonService;

//接口的实现类(业务bean)

//声明式事务，目的是告诉Spring容器这个业务bean是要受到Spring容器的事务进行管理的

@Transactional

public class PersonServiceBean implements PersonService {

/\* 要对数据库的表数据进行：增删改查，就要通过配置好的数据源进行操作。

那么如何【获取数据源】呢？我们采用Spring提供的【依赖注入】功能！

【注意】

我们可以采用Spring提供的辅助类JdbcTemplate来对JDBC进行操作，

因为该类封装了比较多的JDBC代码。只需要调用它的某些方法就可以实

现某些业务逻辑了。

因此目前采用辅助类来操作JDBC，所以就没有必要再去使用数据源属性了,

但是还是需要保留之前通过数据源定义的它的set方法，因为在XML配置文

件中，是通过setter的依赖注入功能注入数据源配置。

\*/

//private DataSource dataSource;//定义数据源

private JdbcTemplate jdbcTemplate;

//为数据源提供set方法(采用属性的setter进行依赖注入)

public void setDataSource(DataSource dataSource) {

//后便直接new出这个辅助类，传入注入进来的dataSource参数

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

//保存person

public void save(Person person) {

//参数1：sql语句 参数2：参数值(考虑位置参数顺序) 参数3：指定参数的类型

jdbcTemplate.update("insert into person(name) values(?)", new Object[]{person.getName()},

new int[]{java.sql.Types.VARCHAR});

}

//根据id来删除person

//设置【回滚】该设置代表计划司出现了check例外，它也会对该事物进行回滚。（也可以设置不回滚）

@Transactional(rollbackFor = Exception.class) //@Transactional(noRollbackFor = RuntimeException.class)

public void deletePerson(int id) {

jdbcTemplate.update("delete from person where id=?", new Object[]{id},

new int[]{java.sql.Types.INTEGER});

}

//更新person

public void update(Person person) {

jdbcTemplate.update("update person set name=? where id=?", new Object[]{person.getName(),person.getId()},

new int[]{java.sql.Types.VARCHAR , java.sql.Types.INTEGER});

}

//获取person

//该注解是在声明该方法是不需要事务的

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

public Person getPerson(int id) {

//参数1：sql语句 参数2：参数值 参数3：指定参数的类型 参数4：代表当查询到这条记录的时候，它会调用对象的回调方法【前提】必须先要实现这个回调方法才可以调用

return (Person)jdbcTemplate.queryForObject("select \* from person where id=?" , new Object[]{id} ,

new int[]{java.sql.Types.INTEGER} , new PersonRowMapper());

}

//获取所有的person

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

@SuppressWarnings("unchecked")

public List<Person> getAllPerson() {

return (List<Person>)jdbcTemplate.query("select \* from person" ,new PersonRowMapper());

}

}

**6. PersonServiceTest**

package JUnit.Test;

import org.junit.BeforeClass;

import org.junit.Test;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.itcast.bean.Person;

import cn.itcast.service.PersonService;

//测试

public class PersonServiceTest {

private static PersonService personService;

//初始化操作

@BeforeClass

public static void setUpBeforeClass() throws Exception {

try {

//得到Spring容器

AbstractApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

//从容器里面去的bean

personService = (PersonService)ctx.getBean("personServiceBean");

} catch (Exception e) {

e.printStackTrace();

}

}

//保存操作(直接new记录并保存)

@Test

public void save() {

personService.save(new Person("李咪"));

}

//获取操作(先获取到记录)

@Test

public void getPerson() {

Person person = personService.getPerson(1);

System.out.println(person.getName());

}

//更新操作(先获取到，再更新同步回数据库)

@Test

public void update() {

Person person = personService.getPerson(1);

person.setName("李四");

personService.update(person);

}

//删除操作(直接根据id删除)

@Test

public void delete() {

personService.deletePerson(1);

}

//获取所有Person(遍历获取)

@Test

public void getAllPerson() {

for(Person person : personService.getAllPerson()){

System.out.println(person.getName());

}

}

}

++++++++++++++++++++++++++++++++++++++++++++++++++++

SpringAOP

**1.beans.xml**

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

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

http://www.springframework.org/schema/context

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

http://www.springframework.org/schema/aop

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

">

<!-- 要进行aop编程,必须要打开(启动)对@Aspectj注解的支持 -->

<aop:aspectj-autoproxy />

<!-- 将切面(MyInterception类)交给Spring管理 -->

<bean id="myInterception" class="cn.itcast.PuTongClass.MyInterception" />

<!-- 将业务bean(PersonServiceBean)也交给Spring管理 -->

<bean id="personServiceBean" class="cn.itcast.Service.impl.PersonServiceBean" />

</beans>

**2. PersonService**

package cn.itcast.Service;

//接口

public interface PersonService {

public void save(String name);

public void update(String name , Integer personid);

public String getPersonName(Integer personid);

}

**3. PersonServiceBean**

package cn.itcast.Service.impl;

import cn.itcast.Service.PersonService;

//实现类（业务bean）

public class PersonServiceBean implements PersonService {

public void save(String name) {

//throw new RuntimeException("出现异常情况了！！！");

System.out.println("我是save方法");

}

public void update(String name, Integer personid) {

System.out.println("我是update方法");

}

public String getPersonName(Integer personid) {

System.out.println("我是getPersonName方法");

return "xxx";

}

}

**4. MyInterception**

package cn.itcast.PuTongClass;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

//这是一个普通java类(要将该类声明为一个【切面】，用注解即可完成)

/\* 第一步：要将该类声明为一个【切面】，用注解@Aspect即可完成.

第二步：在切面里面要【定义切入点】以及声明其名称。

————>在切入点的定义里面使用了AOP的表达式语言

【例子】@Pointcut("execution(\* cn.itcast.Service..\*.\*(..))")

1.execution:代表执行的意思，即就是在执行到业务方法的时候，我要进行拦截。

2.第一个\*：代表返回值的类型，\*代表任意类型。

3.cn.itcast.Service..：代表包名(即就是要对哪个包底下的类进行拦截，后面的".."代表对其子包下的类也要实施拦截)

4.后面的\*：代表要被拦截的那个类，\*代表所有类。

5.最后一个\*：代表要被拦截的方法，\*代表所有的方法。

6.后面的(..)：代表方法参数可任意(可有可无，可多可少)

————>定义切入点的名称(用方法来定义)

第三步：【定义前置通知】(前置通知的注解里面要写入切入点的名称)

【注意】 切面定义完了之后，必须要将其交给Spring管理，否则不会有效果。

做法：在XML配置文件中采用bean方式进行配置。

\*/

@Aspect

public class MyInterception {

//在切面里面要【定义切入点】

@Pointcut("execution (\* cn.itcast.Service.impl.PersonServiceBean.\*(..))")

private void anyMethod() {} //定义切入点的名称(用方法来定义)

//【定义前置通知】(前置通知的注解里面要写入切入点的名称)

//在拦截到的方法之前执行，执行完成之后，被拦截到的方法还会继续执行。

@Before("anyMethod() && args(name)") //获得【输入参数】两个参数名称是对应的

public void doAccessCheck(String name){ //获得【输入参数】

System.out.println("这是前置通知。" + name);

}

//【定义后置通知】

/\*@AfterReturning("anyMethod()")\*/

@AfterReturning(pointcut="anyMethod()",returning="result") //获得【返回结果】

public void doAfterReturning(String result){

System.out.println("这是后置通知。" + result);

}

//【定义最终通知】

@After("anyMethod()")

public void doAfter(){

System.out.println("这是最终通知。");

}

//【定义异常通知】

/\*@AfterThrowing("anyMethod()")\*/

@AfterThrowing(pointcut="anyMethod()",throwing="e") //获得【出现的异常】

public void doAfterThrowing(Exception e){

System.out.println("这是例外(异常)通知。" + e);

}

/\* 【定义环绕通知】===struts2拦截器就是环绕通知

【环绕通知的定义格式固定的;如果使用环绕通知的话，那么就必须确保环绕通知内部必须执行proceed()方法,

如果环绕通知内部没有执行该方法，那么业务bean中被拦截到这个方法是不会执行的。

在执行该方法的时候，如果后面还有切面，先执行后面的切面，如果后面没有切面了，再执行目标对象的业务方法】\*/

@Around("anyMethod()")

public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable{

//if(){} 这块比较适合做权限判断

System.out.println("进入方法");

Object result = pjp.proceed();//【必须要执行该方法】

System.out.println("退出方法");

return result;

}

}

**5. JUnitTest**

package JUnit.test;

import org.junit.BeforeClass;

import org.junit.Test;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.itcast.Service.PersonService;

public class JUnitTest {

@BeforeClass

public static void setUpBeforeClass() throws Exception {

}

@Test

public void InterceptionTest() {

//1.实例化Spring容器

AbstractApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

//2.从Spring中取出PersonServiceBean

PersonService personService = (PersonService)ctx.getBean("personServiceBean");

personService.save("xxx");

ctx.close();//关闭Spring容器

}

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

AOP编程

**1. PersonService**

package cn.itcast.Service;

public interface PersonService {

public void save(String name);

public void update(String name,Integer personid);

public String getPersonName(Integer personid);

}

**2. PersonServiceBean**

package cn.itcast.Service.impl;

import cn.itcast.Service.PersonService;

/\*实现类

需求： 我们要对业务bean里面所有的方法进行拦截，拦截到这个方法之后，我们判断用户是否为null

如果用户为null，那它就没有权限调用该业务方法，如果不为null，那么就代表它有权限调用该

业务方法。

步骤：

第一步：拦截所有的业务方法。

第二步：判断用户是否有权限，有权限就允许它执行业务方法，没权限，就不允许其执行。

(是否有权限是根据用户user是否为null作为判断依据来模拟的)

解决方案：

我们可以通过【设计模式中的代理技术】来实现拦截并进一步处理。

【细节阐述】

客户端应用————>代理对象————>目标对象(业务bean)

客户端应用在调用实际的目标对象(业务bean)之前，要先经过代理对象，即就是客户端调用

的是代理对象。这个代理对象实现了目标对象的所有接口。所以通过代理对象的调用可以把方

法的调用委派给目标对象。等同于（在代理对象的方法内部调用了目标对象的方法）！

这样一来，就不需要在目标对象里面编写臃肿的控制代码来做权限的判断和拦截了。只需要在

处理代理对象里面，在执行目标对象方法之前，对权限做一个判断，有权限则调用目标对象里

面的方法；无权限，则不能调用其方法。那么如何创建代理对象呢？

【创建代理对象】——>两种方式

1. 静态代理(很少使用)

2. 动态代理

我们不需要再去为目标对象去编写静态代理类了，只需要通过JDK或者第三方框架来动态

生成代理对象的字节码就可以了。

【注意】JDK提供的代理类Proxy，是专门为目标对象动态创建代理对象的、

【Proxy使用的前提】

目标对象必须要实现接口，也就是说目标对象必须要面向接口的时候，我们才可以使用JDK

提供的代理类Proxy来创建代理对象。如果目标对象没有实现接口，我们就不能使用Proxy。

++++++++++++++++++++++++++++++++++++++++++++++++

【cglib】创建目标对象的代理对象

在实际应用中，我们的情况可能比较复杂，有可能类没有实现接口，那么就不能用JDK提供的

Proxy来创建代理对象(Proxy有使用的前提：必须实现接口)，那么我们又该如何做呢？

这时候我们就用到了第三方创建代理对象的框架cglib(必须导入cglib的jar文件)来创建目标

对象的代理对象，cglib不要求目标对象必须实现接口。

假设目标对象没有实现任何接口，我们使用cglib来创建目标对象的代理对象

(1.引入jar文件 2.创建cglib的代理工厂来创建目标对象的代理对象)

\*/

//实现接口：通过JDK提供的Proxy类去创建代理对象

//未实现接口：通过cglib去创建代理对象

public class PersonServiceBean /\*implements PersonService\*/ {

//用户

private String user = null;

public String getUser() {

return user;

}

public PersonServiceBean(){}

public PersonServiceBean(String user) {

this.user = user;

}

public void save(String name) {

System.out.println("我是save()方法");

}

public void update(String name, Integer personid) {

System.out.println("我是update()方法");

}

public String getPersonName(Integer personid) {

System.out.println("我是getPersonName()方法");

return "xxx";

}

}

**3. CGlibProxyFactory**

package cn.itcast.AOP;

import java.lang.reflect.Method;

import cn.itcast.Service.impl.PersonServiceBean;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

//cglib的代理工厂(专门用来为目标对象创建代理对象的)

public class CGlibProxyFactory implements MethodInterceptor{

//目标对象的变量

private Object targeObject;

//返回创建好的代理对象(参数是引入目标对象)

public Object CreateProxyIntance(Object targeObject){

//引入传入进来的目标对象

this.targeObject = targeObject;

/\* 采用cglib的Enhancer类来创建目标对象的代理对象。前提：必须要给该代理类对象设置父类(即就是目标类)

当我们把目标类设置为代理对象的父类的时候，那么这个代理技术会产生一个这个目标类的子类，在这个子类里，

它会覆盖目标类里面所有【非final修饰符的方法】。

也就是说cglib创建的代理对象实际上继承了目标类，对目标类里面所有非final修饰的方法进行覆盖，然后在覆

盖的代码里面再添加一些自身的代码。\*/

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(this.targeObject.getClass());

enhancer.setCallback(this);//【设置回调】————>用于将拦截到的方法交由intercept方法去处理。

return enhancer.create();

}

//实现接口中的方法（参数1：代理对象 参数2：被拦截到的方法 参数3：方法的输入参数）

public Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) throws Throwable {

//将目标对象的类型转换为PersonServiceBean业务bean类型。

PersonServiceBean bean = (PersonServiceBean)this.targeObject;

Object result = null;

//有权限,则执行业务方法

if(bean.getUser() != null){

//把方法的调用委派给目标对象

result = methodProxy.invoke(targeObject, args);

}

return null;

}

}

**4. JDKProxyFactory**

package cn.itcast.AOP;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import cn.itcast.Service.impl.PersonServiceBean;

//Proxy的代理工厂(专门用来为目标对象创建代理对象的)

public class JDKProxyFactory implements InvocationHandler{

//目标对象的变量

private Object targeObject;

//返回创建好的代理对象(参数是引入目标对象)

public Object CreateProxyIntance(Object targeObject){

//引入传入进来的目标对象

this.targeObject = targeObject;

/\*【创建代理类】

\* 参数1： 代表类装载器，我们应该使用目标对象的类装载器。

\* 参数2： 代表代理对象要实现的接口是哪些？因为创建的是目标对象的代理类，所以要取得目标对象的所有接口。

\* 那么创建出来的代理对象就会把目标对象里的所有接口都实现。

\* 参数3： 代表回调的过程，这个参数实际上是个接口。回调就是说在拦截到方法的时候，会触发哪个类里面的拦截方法。

\* 写成this，就代表当前类(实例本身),此时当前类就应该实现一个InvocationHandler接口，并实现这个接

\* 口的invoke方法。

\*

\* 【拦截过程】

\* 当我们对代理对象的业务方法进行调用的时候，这个调用操作会被this这个对象所拦截到，拦截到之后它会执

\* 行invoke方法。

\* 也就是说，当客户端应用在调用代理对象的方法的时候，就会调用this对象里面由接口决定的invoke方法。

\* 那么在invoke这个方法里面，如果要访问目标对象的话，我们就必须把方法的调用委派给目标对象。

\* 即就是：Object result = method.invoke(targeObject, args);\*/

return Proxy.newProxyInstance( this.targeObject.getClass().getClassLoader(),

this.targeObject.getClass().getInterfaces(),

this);

}

//实现接口中的方法（参数1：代理对象 参数2：被拦截到的方法 参数3：方法的输入参数）

public Object invoke(Object proxy, Method method, Object[] args) //整个方法可以看做是【环绕通知】

throws Throwable {

//将目标对象的类型转换为PersonServiceBean业务bean类型。

PersonServiceBean bean = (PersonServiceBean)this.targeObject;

Object result = null;

//有权限,则执行业务方法

if(bean.getUser() != null){

//把方法的调用委派给目标对象

//【所谓通知，就是我们要做的事情】

//……………………advice() 【前置通知】

try {

result = method.invoke(targeObject, args);

//……………………afteradvice() 【后置通知】

} catch (Exception e) {

//……………………Exceptionadvice() 【异常通知】

}finally{

//……………………Finallyadvice() 【最终通知】

}

}

return result;

}

}

/\* 【AOP中的概念】

1.Aspect(切面): 指横切性关注点的抽象，即为切面。它与类相似，只是两者的关注点不一样，类是对物体特征的抽象，

而切面是横切性关注点的抽象。横切性关注点的抽象就相当于上面的public Object invoke()方法

2.joinpoint(连接点): 所谓连接点是指那些被拦截到的点，在Spring中，这些点指的是方法，因为Spring只支持方法类型的连接点，

实际上joinpoint还可以是Field或类构造器。

3.Pointcut(切入点): 所谓切入点是指我们要对那些joinpoint进行拦截的定义。

4.Advice(通知): 所谓通知是指拦截到joinpoint之后所要做的事情就是通知，通知分为前置通知、后置通知、异常通知、最终通知、环绕通知。

5.Target(目标对象): 代理的目标对象

6.Weave(织入): 指将aspect(切面)应用到target目标对象并导致proxy对象创建的过程，称为织入。

7.introduction(引入): 在不修改类代码的前提下，introduction可以在运行期为类动态地添加一些方法或Field。

\*/

**5. AOPTest**

package JUnitTest;

import static org.junit.Assert.\*;

import org.junit.BeforeClass;

import org.junit.Test;

import cn.itcast.AOP.CGlibProxyFactory;

import cn.itcast.AOP.JDKProxyFactory;

import cn.itcast.Service.PersonService;

import cn.itcast.Service.impl.PersonServiceBean;

public class AOPTest {

@BeforeClass

public static void setUpBeforeClass() throws Exception {

}

//使用JDK提供的Proxy类来创建代理对象

@Test

public void ProxyTest() {

//创建代理工厂对象

JDKProxyFactory factory = new JDKProxyFactory();

/\* 通过代理工厂来创建目标类(业务bean)的代理对象，并指定输入参数(用户user)，

user不为null，即就是有权限访问目标对象的业务方法。

这块儿的代理对象，我们应该采用接口来引用代理对象，因为代理对象实现了接口。\*/

PersonService service = (PersonService)factory.CreateProxyIntance(new PersonServiceBean("xxx"));

service.save("000");

}

//使用cglib来创建代理对象

@Test

public void CGlibProxyTest() {

//创建代理工厂对象

CGlibProxyFactory factory = new CGlibProxyFactory();

/\* 通过代理工厂来创建目标类(业务bean)的代理对象，并指定输入参数(用户user)，

user不为null，即就是有权限访问目标对象的业务方法。

这块儿的代理对象，我们应该采用目标对象本身来引用代理对象，因为目标对象没有实现接口。\*/

PersonServiceBean service = (PersonServiceBean)factory.CreateProxyIntance(new PersonServiceBean("xxx"));

service.save("999");

}

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

SSH整合

1. **web.xml**

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

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

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

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

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

id="WebApp\_ID" version="2.5">

<display-name>SS2H</display-name>

<!-- ++++++++++【实例化Spring容器】+++++++++++++ -->

<!-- 1.【指定Spring的配置文件】默认是从web的根目录寻找配置文件，

我们可以通过Spring提供的classpath:前缀指定从类路径寻找 -->

<context-param>

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

<param-value>classpath:beans.xml</param-value> <!-- 多个配置文件，使用逗号隔开即可：...,classpath:abc.xml -->

</context-param>

<!-- 2.【对Spring容器进行实例化】,实例化后的容器实例放在了application范围 -->

<listener>

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

</listener>

<!-- ++++++++++【配置Struts2，启动Struts2框架】+++++++++++++ -->

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

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

</filter-mapping>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

1. **beans.xml**

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

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

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-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/aop

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

http://www.springframework.org/schema/tx

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

">

<!-- 通过【注解+自动扫描】方式把组件纳入Spring容器中管理，就必须打开组件扫描的配置项： -->

<!-- base-package代表要扫描哪个包底下的类以及其子包底下的类！ -->

<context:component-scan base-package="cn.itcast" />

<!-- 第一步：配置数据源(采用c3p0数据源) -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">

<property name="driverClass" value="org.gjt.mm.mysql.Driver" />

<property name="jdbcUrl" value="jdbc:mysql://localhost:3306/ss2h?useUnicode=true&amp;characterEncoding=UTF-8" />

<property name="user" value="root" />

<property name="password" value="624963768" />

<!-- 初始化时获取的链接数，取值应在minPoolSize和maxPoolSize之间，Default=3 -->

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

<!-- 连接池中保留的最小连接数 -->

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

<!-- 连接池中保留的最大连接数, Default = 15 -->

<property name="maxPoolSize" value="300" />

<!-- 最大空闲时间，60秒内未使用则链接被丢弃，若为0，则永不丢弃 ，Default = 0 -->

<property name="maxIdleTime" value="60" />

<!-- 当连接池中的链接耗尽的时候，c3p0一次性同时获取的链接数，Default = 3 -->

<property name="acquireIncrement" value="5" />

<!-- 每60秒检查所有连接池中的空闲链接，Default = 0 -->

<property name="idleConnectionTestPeriod" value="60" />

</bean>

<!-- 第二步：集成Hibernate -->

<!-- 1.集成SessionFactory (由Spring提供专门的类来完成) -->

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"/> <!-- 注入数据源 -->

<property name="mappingResources"> <!-- 映射Hibernate映射文件 -->

<list>

<value>cn/itcast/bean/Employee.hbm.xml</value>

</list>

</property>

<!-- 指定Hibernate的一些属性 -->

<property name="hibernateProperties">

<value>

hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

hibernate.hbm2ddl.auto=update

hibernate.show\_sql=false

hibernate.format\_sql=false

</value>

</property>

</bean>

<!-- 2.配置事务管理器 (使用的是关于Hibernate的事务服务，因为事务的开启底层是通过session来做的) -->

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"/> <!-- 注入sessionFactory，来开启事务 -->

</bean>

<!-- 第三步：使用基于注解方式配置事务 (打开事务的配置项)-->

<tx:annotation-driven transaction-manager="txManager" /> <!-- 注入事务管理器 -->

</beans>

1. **Struts2.xml**

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

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"

"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

<!-- struts默认的视图主题 -->

<constant name="struts.ui.theme" value="simple" />

<!-- 【更换Struts2内部的对象工厂类】———>，最好明确指定使用Spring的对象工厂。

Struts2有一个对象工厂这个类，这个类是可以更换的，它主要是用来创建action的。

使用Spring对象工厂类对替换掉Struts2默认的对象工厂类，即就是action现在就

交由Spring去创建和维护了，而不是由Struts2来创建及维护。

value="spring" spring其实就是一个bean工厂的名称，该bean工厂在struts2

集成Spring的插件jar文件(struts2-spring-plugin-2.3.14)中定义。-->

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

<!-- 既然action交给了Spring容器来创建及维护，那么如果struts2要使用到action，怎么做呢？

应该给定一个名称，让struts2在Spring容器中根据这个名称去获取到对应action。

注意：原先我们是根据class属性来指定action类的，但是现在action类都交给Spring来创建维护了，

所以现在我们就不需要再class属性中配置action类了，那么我们又如何从Spring容器中获取到已经创

建好的action类呢？即就是，class="action bean的名称"，在class中给定其action bean名称即可

过程：此时这个对象工厂就会根据class属性中所填写的action bean的名称，到Spring容器中获取到

这个action实例。

-->

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

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

<result name="list">/WEB-INF/page/employee.jsp</result>

</action>

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

<!-- method="{1}"代表:执行的是manager\_\*中的第一个方法名为\*的方法 -->

<result name="add">/WEB-INF/page/AddEmployee.jsp</result>

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

</action>

</package>

</struts>

**4. EmployeeService**

package cn.itcast.Service;

import java.util.List;

import cn.itcast.bean.Employee;

//接口

public interface EmployeeService {

public void save(Employee employee); //增加

public void delete(String... usernames); //删除（可以批量删除）

public void update(Employee employee); //修改

public Employee find(String username);//查找

public List<Employee> list(); //查询所有

}

**5. EmployeeServiceBean**

package cn.itcast.Service.impl;

import java.util.List;

import javax.annotation.Resource;

import org.hibernate.SessionFactory;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Propagation;

import org.springframework.transaction.annotation.Transactional;

import cn.itcast.Service.EmployeeService;

import cn.itcast.bean.Employee;

//实现类(业务bean)

@Service //标注交给Spring管理

@Transactional //标注要使用事务管理

public class EmployeeServiceBean implements EmployeeService{

/\* 要执行对实体对象的增删改查操作，我们需呀使用到Session对象(前提要的到SessionFactory对象)

那么就要注入SessionFactory对象！那要怎么样注入SessionFactory对象呢？

方式：

1.通过提供SessionFactory对象的set方法，利用属性的setter方法进行注入。

2.通过注解的方式进行注入：@Resource SessionFactory factory;

【注意】 现在不能在方法内部直接使用：factory.openSession()来开启事务了，

因为Spring容器已经为该业务bean中的所有方法在执行之前都默认开启了

一个事务,要采用factory.openSession()，那么就相当于开启了新的事务。

那么就不能再使用Spring本身的事务功能了……

【Spring开启事务的过程】

Spring的事务管理器会先从SessionFactory里面获取到一个session对象，

用获取到的session对象再来开启事务，相当于在方法前执行：session.begin();

如果要想使用Spring已经开启好的事务，就必须要得到当前已经开好事务的session

对象，即就是：factory.getCurrentSession();

然后才能通过该session来对实体进行操作。\*/

//利用注解的方式注入SessionFactory对象

@Resource SessionFactory factory;

//保存操作

public void save(Employee employee) {

//首先获得当前已经开好事务的session对象，再调用persist/save方法进行保存

factory.getCurrentSession().persist(employee);

}

//删除操作

public void delete(String... usernames) {

for(String username : usernames){

factory.getCurrentSession().delete(factory.getCurrentSession().load(Employee.class, username));

}

}

//更新操作

public void update(Employee employee) {

factory.getCurrentSession().merge(employee);//merge=saveOrUpdate

}

//查询操作(不需要使用事务)

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

public Employee find(String username) {

return (Employee)factory.getCurrentSession().get(Employee.class, username);

}

//查询所有操作(也不需要使用事务)

@SuppressWarnings("unchecked") //终止警告(不用管)

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

public List<Employee> list() {

return factory.getCurrentSession().createQuery("from Employee").list();

}

}

**6. Employee**

package cn.itcast.bean;

//员工的实体类

public class Employee {

private String username;

private String password;

private Gender gender = Gender.MAN; //枚举类型的：性别

public Employee() {}

public Employee(String username, String password) {

this.username = username;

this.password = password;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public Gender getGender() {

return gender;

}

public void setGender(Gender gender) {

this.gender = gender;

}

}

**7. Gender**

package cn.itcast.bean;

//枚举类型：性别

public enum Gender {

MAN,WOMEN;

}

**8. Employee.hbm**

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

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<!-- hibernate的配置 -->

<hibernate-mapping package="cn.itcast.bean">

<class name="Employee" table="employee">

<id name="username" length="20"/>

<property name="password" length="20" not-null="true"/>

<property name="gender" length="5" not-null="true">

<type name="org.hibernate.type.EnumType">

<param name="enumClass">cn.itcast.bean.Gender</param>

<!-- 【注意】12为java.sql.Types.VARCHAR常量值，即就是：保存枚举的字面

值到数据库，如果不指定type参数，默认则保存枚举的索引值(从0开始)到数据库中 -->

<param name="type">12</param>

</type>

</property>

</class>

</hibernate-mapping>

**9. EmployeeAction**

package cn.itcast.action;

import javax.annotation.Resource;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Controller;

import com.opensymphony.xwork2.ActionContext;

import cn.itcast.Service.EmployeeService;

//action类

@Controller //将action类交给Spring管理，默认bean的名称是：employeeAction

@Scope("prototype") //将action作用域设为原型模式,即就是每次都会创建新的的action实例(线程安全)

public class EmployeeAction {

//既然把action交给了Spring来管理，那就可以使用Spring的依赖注入功能，把业务bean注入进来。

@Resource EmployeeService employeeService;//【注入业务bean】，通过接口来获取

//Struts2的action都默认执行execute方法

public String execute(){

ActionContext.getContext().put("employees", employeeService.list());

return "list";

}

}

**10. EmployeeManagerAction**

package cn.itcast.action;

import javax.annotation.Resource;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Controller;

import com.opensymphony.xwork2.ActionContext;

import cn.itcast.Service.EmployeeService;

import cn.itcast.bean.Employee;

//action类

@Controller //将action类交给Spring管理，默认bean的名称是：employeeManagerAction

@Scope("prototype") //将action作用域设为原型模式,即就是每次都会创建新的的action实例(线程安全)

public class EmployeeManagerAction {

//在action里面通过符合类型去接受参数值（目前得到的employee对象就已经封装好了请求参数的值了）

private Employee employee;

@Resource EmployeeService employeeService;//注入业务bean，对employee(请求参数)进行保存

public Employee getEmployee() {

return employee;

}

public void setEmployee(Employee employee) {

this.employee = employee;

}

//显示界面的方法

public String addUI(){

return "add";

}

//添加用户方法

public String add(){

employeeService.save(employee);//对employee(请求参数)进行保存

ActionContext.getContext().put("message","操作成功");

return "message";

}

}

**11 EmployeeTest**

package JUnit.Test;

import java.util.List;

import org.junit.BeforeClass;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.itcast.Service.EmployeeService;

import cn.itcast.bean.Employee;

import cn.itcast.bean.Gender;

//单元测试

public class EmployeeTest {

//【得到业务bean】(初始化完成之后，从Spring容器中获取)

private static EmployeeService employeeService;//通过接口来获取

@BeforeClass

public static void setUpBeforeClass() throws Exception {

/\*Spring和Hibernate集成好了之后，要想让表生成出来，我们只需要实例化Spring容器即可，

Spring一旦实例化，它就会导致那些对象自动被创建。\*/

try {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");//做初始化行为

//getBean这个bean的名称是业务bean类的简单名称(直接被扫描出来的)

employeeService = (EmployeeService)ctx.getBean("employeeServiceBean");

} catch (Exception e) {

e.printStackTrace();

}

}

//【可以用上面定义的类employeeService，来完成增删改查动作】

@Test

public void save() {

employeeService.save(new Employee("李咪", "limi"));

}

@Test

public void delete() {

employeeService.delete("李咪");

}

@Test

public void update() {

//先获取到记录

Employee em = employeeService.find("李咪");

//再修改

em.setGender(Gender.WOMEN);

//同步回数据库(更新)

employeeService.update(em);

}

@Test

public void find() {

Employee em = employeeService.find("李耀");

System.out.println(em.getPassword());

}

@Test

public void list() {

List<Employee> ems = employeeService.list();

//迭代

for(Employee em : ems){

System.out.println(em.getPassword());

}

}

}

**12. AddEmployee.jsp**

<%@ page language="java" contentType="text/html; charset=utf-8"

pageEncoding="utf-8"%>

<!-- 【标签库】 -->

<%@ taglib uri="/struts-tags" prefix="s" %>

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

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>添加员工</title>

</head>

<body>

<s:form action="manager\_add" namespace="/employee" method="post">

用户名：<s:textfield name="employee.username"/><br/>

密码：<s:password name="employee.password"/><br/>

性别:<s:radio list="#{'MAN':'男','WOMEN':'女'}" listKey="key" listValue="value" name="employee.gender"/>

<input type="submit" value="提交"/>

</s:form>

</body>

</html>

**13. employee.jsp**

<%@ page language="java" contentType="text/html; charset=utf-8"

pageEncoding="utf-8"%>

<!-- 【标签库】 -->

<%@ taglib uri="/struts-tags" prefix="s" %>

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

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>员工列表</title>

</head>

<body>

<a href="<s:url action="manager\_addUI" namespace="/employee"/>">添加员工</a>

采用【ongl表达式】<br/>

<s:iterator value="#request.employees">

<s:property value="username"/>,

<s:property value="password"/>,

<s:property value="gender"/><br/>

</s:iterator>

采用【jstl表达式】<br/>

<c:forEach items="${employees}" var="employee">

${employee.username },

${employee.password },

${employee.gender }<br/>

</c:forEach>

</body>

</html>

**14.** **message.jsp**

<%@ page language="java" contentType="text/html; charset=utf-8"

pageEncoding="utf-8"%>

<!-- 【标签库】 -->

<%@ taglib uri="/struts-tags" prefix="s" %>

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

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>消息提示</title>

</head>

<body>

<h1>

<font color="red">

<s:property value="#request.message"/>

</font>

</h1>

</body>

</html>