# Spring

**Spring解决对象的创建和对象之间的装配问题。**

自动创建业务层和持久层对象，自动将持久层对象装配到业务层的相关属性上。

Spring是轻量级的框架。

**AOP（面向切面）:**一个对象方法可以根据需求增加功能。采用动态代理技术（面向切面，如 过滤器Filter）

**IOC（控制反转/依赖注入）**：由spring自动创建对象，自动给对象装配关系。

Spring替Hibernate可以自动生成SessionFactory工厂，拿到session自动和线程绑定。自动生成全局性事务。

Spring替struts：通过事务控制并发性。替struts完成action的创建，把业务层都注入。

**Sring的核心技术是IOC。**

**普通示例：（未使用spring）**

**Dao层：**

接口：interface UserDao

实现类：MysqlDaoImpl， OracleDaoImpl (都实现了UserDao接口)

**业务层：** UserManager

private UserDao userDao;//关联属性，持久层接口类型的变量

public UserManager(){}

//通过setUserDao方法，完成装配（也可以通过工厂）

private void setUserDao(UserDao userDao){

this.userDao=userDao;

}

**实现装配的方法：**

1. 通过set方法

public void setUserDao(UserDao userDao){this.userDao=userDao}

1. 通过有参的构造方法

public UserManager(UserDao userDao){this.userDao=userDao}

1. 通过工厂

**控制层：**UserAction（在execute()方法中）

UserManager userManager=new UserManager();

userManager.setUserDao(new MysqlDaoImpl());//调用业务对象的setUserDao()装配MysqlDaoImpl持久对象

**优点：**层与层之间依赖接口，降低业务层和持久层的耦合性。结构型好便于维护。

**缺点：**业务层要给持久层接口类型的变量赋值要修改源代码,必须要查找持久层对象。

如果持久层代码出现问题，业务层也会受到影响出现bug.**业务层依赖持久层**。

依赖关系过多，业务层的装配量过大。（消除依赖关系使用工厂）

**解决办法：**（**不使用工厂，反射， properties**）**使用spring自动创建对象，自动完成装配。**

**Spring的环境搭建：（测试环境）**

1. 建立java项目，
2. 建立userLibrary，加载jar包
3. Spring的主jar,log4j的jar包，commons-logging.jar，aspectj下的jar包（2个），junit测试的jar包，cglib代理的jar包。（共7个，spring2.x）
4. applicationContext.xml、log4j.properties放在src下

**spring的配置信息：**

**applicationContext.xml配置文件：**

创建bean工厂，自动通过**无参的构造方法**创建MysqlDaoImpl持久层对象。放入map中，id是标识，作为key。

<bean id=”**mysqlDaoImpl**” class=”cn.spring.MysqlDaoImpl”/>

创建bean工厂，自动通过**无参的构造方法**创建OracleDaoImpl持久层对象。放入map中，id是标识，作为key。

<bean id=”oracleDaoImpl” class=”cn.spring.OracleDaoImpl”/>

创建bean工厂，自动通过**无参的构造方法**创建UserManager业务层对象。放入map中，id是标识，作为key。

<bean id=”userManager” class=”cn.spring.UserManager”>

**name:**标识属性，是UserManager中的定义的属性的属性名

**ref:**指定为该属性赋值的对象

调用UserManager对象的setUserDao方法，给userDao属性设值

**mysqlDaoImpl**是MysqlDaoImpl的标识，通过这个标识将MysqlDaoImpl对象设到userDao属性中**。**UserManager类中一定要有setUserDao()方法

<property name=”userDao” ref=”**mysqlDaoImpl**”/>

</bean>

**Test:（测试类）:使用Junit测试类中的setUp（）方法进行初始化，在setUp()方法中创建BeanFactory对象。**

factory =new ClassPathXmlApplicationContext(“**applicationContext-\*.xml”**);

**application-\*.xml：匹配所有的application-的xml文件。**

BeanFactory factory =new ClassPathXmlApplicationContext(“applicationContext.xml”);

1. 从src下读取配置文件创建工厂
2. 工厂一创建就会创建三个对象（MysqlDaoImpl，OracleDaoImpl ，UesrManger）
3. 并且装配好了关系(UserManager的userDao属性已经赋值了)
4. BeanFactory是重量级的，非常耗时，建议一个项目初始化一次。
5. 与Hibernate框架集成的时候，会在BeanFactory中创建SessionFactory

UserManager userManager=(UserManager)**factory.getBean**(“userManager”);//拿到userManager标识所关联的对象UserManager对象（在配置文件中，bean标签下：id= userManager）,拿到的Object类型的数据，需要转型。

userManager.save(user);//直接调用UserManager类中的方法

**一旦持久层发生变化，换成oracle数据库只要修改配置文件就可以了，不用再修改代码。**

**通过构造方法注入：<constructor-arg ref=”mysqlDaoImpl”/>**

创建bean工厂，自动通过**无参的构造方法**创建UserManager业务层对象。放入map中，id是标识，作为key。

<bean id=”userManager” class=”cn.spring.UserManager”>

调用UserManager对象的setUserDao方法，给userDao属性设值

**mysqlDaoImpl**是MysqlDaoImpl的标识，通过这个标识将MysqlDaoImpl对象设到userDao属性中**。**UserManager类中一定要有**有参的构造方法。**

**<constructor-arg ref=”mysqlDaoImpl”/>**

</bean>

**使用spring给普通属性赋值：**

public class Bean{

private int num;

private String str;

private List list1;

private Map map1

private String[] arr;

private Date date;

提供公有的set和get方法，无参的构造方法

}

**在applicationContext-bean.xml文件中：**

<bean id=”**bean1**” class=”cn.spring.Bean”>

**整型数据的配置：**

<property name=”num” value=”123”/>

**字符串的配置：**

<property name=”str” value=”hello”/>

**List集合的配置：**

//将list1通过setList1()设到list1属性中。

<property name=”list1” >

<list> //看到list标签就会自动创建List集合，

<value>happy</value>//将<value>标签指定的值自动add到List集合中

<value>happy1</value>

</list>

</property>

**Set集合的配置：**

//将set1通过setSet1()设到set1属性中。

<property name=”set” >

<set> //看到set标签就会自动创建Set集合，

<value>hash</value>

<value>hashOne</value>

<ref bean=”bean1”/>//引用别的bean标签中的对象

<set>

</property>

**String[]的配置：**

//将List集合对象自动转换成字符串数组对象，然后设到arr属性中

<property name=”arr” >

<list> //看到list标签就会自动创建List集合，

<value>happy</value>

<value>happy1</value>

</list>

</property>

**Map集合的配置：**

//将map1通过setMap1()设定到map1属性中

<property name=”map1” >

<map>//看到map标签就会创建Map集合对象

<entry key=”k1” valule=”v1”/>

<entry key=”name” valule=”zhangSan”/>

</map>

</property>

</bean>

**关联属性的配置：**

<property name=”关联属性名” ref=”引用关联属性对象的主标识”>

eg: **<property name=”userDao” ref=”mysqlDaoImpl”>**

也可以用**ref标识设值：**

<property name=”userDao”>

**<ref bean=”mysqlDaoImpl”/>**

</property>

**IOC：自动创建对象并且给对象的属性注入数据**

工厂一创建就已经把对象创建完了，关系也装配好了。

**提取公共属性，整体注入： 使用abstract注入**

<bean id=”**beanAbstract**” **abstract=”true”**>

<property name=”id” value=”10”/>

<property name=”name” value=”zhangSan”/>

</bean>

<bean id=”bean1” class=”cn.spring.Bean” **parent=”beanAbstract”**>

<property name=”id” value=”11”/> //这里重新配置了的按照当前的配置，没有重新配置id就按照**beanAbstract 中的配置赋值。**

</bean>

**Spring获取对象 的单例和多例：**

默认情况下拿到的对象是单例对象。

想要拿到多例对象需要配置，延迟创建。

<bean id=”bean1” class=”cn.spring.Bean” **scope=”singleton”/**> 指定生成的对象为单例，默认配置为singleton

<bean id=”bean1” class=”cn.spring.Bean” **scope=”prototype”/**> 指定生成的对象为多例

**延迟创建：**

Spring创建工厂的时候看到配置scope=prototype时，不会马上创建bean1对象，会等在使用bean1对象时再创建。放入map集合中，key为bean1，value为创建的对象。

**将字符串转换为日期类型：**

1. **创建属性编辑器（转换器）**

public class **UtilDatePropertyEditor** extends PropertyEditorSupport{

private String **format**;

**public void setAsText**(String text)throws Exception{

SimpleDateFormat adf=new SimpleDateFormate(**format**);

Try{

Date d=sdf.parse(text);

This.setValue(d); //自动的找到Bean对象的setDate1()方法，将转换后的日期值设到Bean1对象的date1属性中

}catch(ParseException e){

e.printStackTrace();

}

}

}

1. **配置属性编辑器：（类型转换器）**

方法一：整体配置属性编辑器

<bean id=” customEditorConfigurer”

class=”org.springframework.beans.factory.config.CustomEditorConfigurer”>

// customEditors是CustomEditorConfigurer类的一个私有属性，是一个map集合（非静态也非单例），里面存放类型转换器，**private Map customEditors;**

<property name=” customEditors”>

<map>

<entry key=”java.util.Date”>

<bean class="cn.spring.**UtilDatePropertyEditor**">

<property name="**format**" value="yyyy:MM:dd"/>

</bean>

</entry>

</map>

</property>

</bean>

方法二：分开配置属性编辑器，使用ref引用：（效果和整体配置一样）

<bean id=” customEditorConfigurer”

class=”org.springframework.beans.factory.config.CustomEditorConfigurer”>

// customEditors是CustomEditorConfigurer类的一个私有属性，是一个map集合（非静态也非单例），里面存放类型转换器，**private Map customEditors;**

<property name=” customEditors”>

<map>

<entry key=”java.util.Date” ref=” utilDatePropertyEditor”/>

</map>

</property>

</bean>

<bean id=”utilDatePropertyEditor” class="cn.spring.**UtilDatePropertyEditor**">

<property name="**format**" value="yyyy:MM:dd"/>

</bean>

# 给java.util.Date类型的属性设值：

1、定义Bean1类

public class Bean1{

private Date date1;

public Bean1(){}

public void setDate1(Date date1){

this.date1=date1;

}

public Date getDate1(){

return date1;

}

}

2、创建属性编辑器：(转换器)

public class **UtilDatePropertyEditor** extends PropertyEditorSupport{

private String **format**;

**public void setAsText**(String text)throws Exception{

SimpleDateFormat adf=new SimpleDateFormate(**format**);

Try{

Date d=sdf.parse(text);

This.setValue(d); //自动的找到Bean对象的setDate1()方法，将转换后的日期值设到Bean1对象的date1属性中

}catch(ParseException e){

e.printStackTrace();

}

}

}

3、配置属性编辑器，将自定义转换器注册到CustomEditorConfigurer类中customEditors Map中：

<bean id=” customEditorConfigurer”

class=”org.springframework.beans.factory.config.CustomEditorConfigurer”>

// customEditors是CustomEditorConfigurer类的一个私有属性，是一个map集合（非静态也非单例），里面存放类型转换器，**private Map customEditors;**

<property name=” customEditors”>

<map>

<entry key=”java.util.Date” ref=” utilDatePropertyEditor”/>

</map>

</property>

</bean>

<bean id=”utilDatePropertyEditor” class="cn.spring.**UtilDatePropertyEditor**">

<property name="**format**" value="yyyy:MM:dd"/>

</bean>

4、 使用java.util.Date属性编辑器（转换器）

<bean id=”bean1” class=”cn.spring.Bean1”>

<property name=”date1”>

<value>2017:09:11</value>

</property>

</bean>

1. 自动创建CustomEditorConfigurer对象（spring自带的类），将spring自定义的属性编辑器注入
2. 到CustomEditorConfigurer中找到customEditors Map集合，
3. 到customEditors Map集合中找到key为 java.util.Date 对应的value，---相应的类utilDatePropertyEditor（这就是key对应的value）
4. 到utilDatePropertyEditor类中找到setAsText(text)方法。将字符串转换成日期类型。SimpleDateFormat sdf=new SimpleDateFormat(format);

Date d =sdf.parse(text);

5、 **this.setValue(d);**//自动的找到Bean对象的setDate1(d)方法，将转换后的日期值设到Bean1对象的date1属性中。

# IOC(依赖注入/控制反转)

**注入属性：一定要提供公有的set()方法**

**注入对象：一定要提供公有的无参的构造方法**

**IOC（控制反转/依赖注入）**：由spring自动创建对象，自动给对象装配关系。

Spring替Hibernate可以自动生成SessionFactory工厂，拿到session自动和线程绑定。自动生成全局性事务。

Spring替struts：通过事务控制并发性。替struts完成action的创建，把业务层都注入。

**Sring的核心技术是IOC。**

**applicationContext.xml配置文件：**

创建bean工厂，自动通过**无参的构造方法**创建MysqlDaoImpl持久层对象。放入map中，id是标识，作为key。

<bean id=”**mysqlDaoImpl**” class=”cn.spring.MysqlDaoImpl”/>

创建bean工厂，自动通过**无参的构造方法**创建OracleDaoImpl持久层对象。放入map中，id是标识，作为key。

<bean id=”oracleDaoImpl” class=”cn.spring.OracleDaoImpl”/>

创建bean工厂，自动通过**无参的构造方法**创建UserManager业务层对象。放入map中，id是标识，作为key。

<bean id=”userManager” class=”cn.spring.UserManager”>

**name:**标识属性，是UserManager中的定义的属性的属性名

**ref:**指定为该属性赋值的对象

调用UserManager对象的setUserDao方法，给userDao属性设值

**mysqlDaoImpl**是MysqlDaoImpl的标识，通过这个标识将MysqlDaoImpl对象设到userDao属性中**。**UserManager类中一定要有setUserDao()方法

<property name=”userDao” ref=”**mysqlDaoImpl**”/>

</bean>

**使用spring给普通属性赋值：**

public class Bean{

private int num;

private String str;

private List list1;

private Map map1

private String[] arr;

private Date date;

提供公有的set和get方法，无参的构造方法

}

**在applicationContext-bean.xml文件中：**

<bean id=”**bean1**” class=”cn.spring.Bean”>

**整型数据的配置：**

<property name=”num” value=”123”/>

**字符串的配置：**

<property name=”str” value=”hello”/>

**List集合的配置：**

//将list1通过setList1()设到list1属性中。

<property name=”list1” >

<list> //看到list标签就会自动创建List集合，

<value>happy</value>//将<value>标签指定的值自动add到List集合中

<value>happy1</value>

</list>

</property>

**Set集合的配置：**

//将set1通过setSet1()设到set1属性中。

<property name=”set” >

<set> //看到set标签就会自动创建Set集合，

<value>hash</value>

<value>hashOne</value>

<ref bean=”bean1”/>//引用别的bean标签中的对象

<set>

</property>

**String[]的配置：**

//将List集合对象自动转换成字符串数组对象，然后设到arr属性中

<property name=”arr” >

<list> //看到list标签就会自动创建List集合，

<value>happy</value>

<value>happy1</value>

</list>

</property>

**Map集合的配置：**

//将map1通过setMap1()设定到map1属性中

<property name=”map1” >

<map>//看到map标签就会创建Map集合对象

<entry key=”k1” valule=”v1”/>

<entry key=”name” valule=”zhangSan”/>

</map>

</property>

</bean>

**关联属性的配置：**

<property name=”关联属性名” ref=”引用关联属性对象的主标识”>

eg: **<property name=”userDao” ref=”mysqlDaoImpl”>**

也可以用**ref标识设值：**

<property name=”userDao”>

**<ref bean=”mysqlDaoImpl”/>**

</property>

**IOC：自动创建对象并且给对象的属性注入数据**

工厂一创建就已经把对象创建完了，关系也装配好了。

**提取公共属性，整体注入： 使用abstract注入**

<bean id=”**beanAbstract**” **abstract=”true”**>

<property name=”id” value=”10”/>

<property name=”name” value=”zhangSan”/>

</bean>

<bean id=”bean1” class=”cn.spring.Bean” **parent=”beanAbstract”**>

<property name=”id” value=”11”/> //这里重新配置了的按照当前的配置，没有重新配置id就按照**beanAbstract 中的配置赋值。**

</bean>

# Spring静态代理和JDK动态代理

Aop:解决维护问题，在不修改目标类的基础上增加功能。

**静态代理Spring：**

创建代理类，和目标类实现同一个接口（具有相同的方法声明）

目标类和代理类的不同点：目标类的方法都实现了

代理类中的方法是空实现

接口的回调：使用接口定义变量

**UserManager：目标类**

Public class UserManager implements **UserDao**{

Public UserManager(){}

}

**UserManagerProxy：代理类**

//在代理类中做权限验证，和目标类没有直接关系

Public class UserManagerProxy implements **UserDao**{

private UserManager userManager;//目标类类型的属性

提供公有的set方法和无参的构造方法。

**public void add(){**

**customValidate();**//自定义的权限验证方法

**userManager.add();**//目标类的add()方法

**logValidate();**//执行日志方法

**}**

}

把目标类对象注入代理类中:（Spring的配置------在applicationContext.xml文件中）

<!- - 创建目标类对象- - >

<bean id=”**userManager**” class=”cn.spring.UserManger”/>

<!- - 创建代理类对象，并且为代理类对象装配目标类类型的属性- - >

<bean id=”userManagerProxy” class=”cn.spring.UserMangerProxy”>

<property name=”userManager” ref=”**userManager**” />

</bean>

Test 测试类：（通过代理类，完成对目标类方法的加强）

从工厂中拿到代理对象：

BeanFactory factory=new ClassPathXmlApplicationContext(“applicationContext.xml”)；

//拿到代理类对象

UserManager userManager=( UserManager)factory.getBean(“userManagerProxy”);

userManager.add();//先进行代理类权限验证，再执行目标类add()方法，再执行代理类日志

* **AOP中的术语简介**

\* Cross cutting concern 横切性关注点---检查安全性 (checkSecurity)

\* Aspect 切面（模块化的类--SecurityHandler）

\* Advice 横切性关注点的实现—checkSecurity() （添加的功能点—--方法）

\* Pointcut 应用Advice的过滤条件（添加的验证功能 用在哪些方法上 切点）

\* Joinpoint 连接点—织入的点（如addUser(..)方法，指定的添加功能点的方法）

\* Weave织入--把关注点的方法应用到目标方法的过程

\* Target Object 目标对象

\* Proxy 代理对象

\* Introduction 在类中动态的增加方法

在方法之前执行的叫 BeforeAdvice(安全检查)

在方法之后执行的叫AfterAdvice（日志）

aspect 切面类创建的对象叫切面

advice（添加的验证功能）用在哪些方法上叫 pointcut，切点

把advice应用到方法中的过程叫 Weaver

**动态代理（JDK）：(**JDK的动态代理只对实现接口的类做动态代理**)**

**实现接口： InvocationHandler**

public class SecurityHandler implements **InvocationHandler**{

private Object targetObject;//定义目标对象

public Object newProxy(Object targetObject) {

this.targetObject = targetObject;

/\*\*

\* 第一个参数设置代码使用的类装载器，一般采用和目标

\* 类相同的类装载器

\* 第二个参数设置代理类实现的接口，和目标类实现同一个接口

\* 第三个参数设置回调对象，当代理对象的方法被调用时，

\* 会委派给该参数指定对象的invoke方法

\*/

return **Proxy.newProxyInstance(**

**targetObject.getClass().getClassLoader(),** //目标类的类加载器

**targetObject.getClass().getInterfaces(),** //代理类实现的接口

**this);** //通过this可以自动调用当前对象的invoke()方法

}

//invoke方法为**InvocationHandler**接口中定义的方法

/\*\*

\* invoke:是接口中定义的方法

\* proxy:自动封装的 不用管

\* method:Method方法对象

\* args: method方法对应的实参

\*/

public Object **invoke**(Object proxy, Method **method**, Object[] args)throws Throwable {

checkSecurity();//自定义的权限验证方法

Object ret = null;

try {

// 调用this.targetObject的方法(方法反射)

ret = **method**.invoke(this.targetObject, args);//执行目标类的方法

}catch(Exception e) {

e.printStackTrace();

throw new java.lang.RuntimeException(e);

}

return ret;//返回方法执行的结果

}

public void checkSecurity(){

……验证代码……

}

}

代理类对象和目标类对象实现同一个接口

代理对象拥有this(当前对象的地址)，就会调用当前对象的invoke()方法

测试类：JDK动态代理（和spring无关）

SecurityHandler handler = new SecurityHandler();

**//生成代理对象，可以拿到SecurityHandler对象的地址**

1. 将目标对象设到SecurityHandler类的targetObject属性上
2. 将目标对象的类加载器作为代理对象的类加载器
3. 将目标对象的实现的接口作为代理对象的实现的接口
4. 将SecurityHandler的地址传给代理对象
5. 代理对象通过SecurityHandler的地址找到目标类对象
6. （没有具体代理类，动态生成）

UserManager userManager = (UserManager)handler.newProxy(new UserManagerImpl());

userManager.addUser("张三", "123");

serManager.deleteUser(1);

**动态代理的核心思想：**

用实现InvocationHandler的接口的类

动态创建代理对象，实现invoke方法，在执行invoke方法之前先做权限验证。

**动态代理的机制：**没有明确的代理类，代理对象在程序运行过程中动态生成。

使用invoke方法将动态权限验证加到实现目标对象的方法之前或者之后。（自动调用）

**动态代理（执行过程）**

1. 创建一个类实现InvocationHandler接口
2. 复写InvocationHandler接口中的invoke方法
3. 在类中创建代理对象（用来拿到目标对象的属性）
4. 通过实例方法newProxy()方法创建代理对象一定要传递个实参，目标对象。执行Proxy.new ProxyInstance()方法
5. 创建代理对象：Proxy.newProxyInstance(

targetObject.getClass().getClassLoader(), //目标类的类加载器

targetObject.getClass().getInterfaces(), //代理类实现的接口

this); //通过this可以自动调用当前对象的invoke()方法

1. 通过代理对象找到目标对象
2. 通过this找到当前类的实例方法 invoke,并且自动调用

**在测试类中**

1. 创建SecurityHandler对象
2. 通过SecurityHandler对象调用newProxy()创建代理对象
3. 代理对象调用this所指的接口对象的invoke方法
4. invoke(): 传递封装方法的Method对象,封装实参的Object[ ]
5. 在invoke方法中先执行权限验证，在执行目标对象的相应的方法
6. 如果后面还有方法就继续执行

Advice:新增加的功能 ：cheacSecurity()

包含advice的类叫做切面类：SecurityHandler

切面类的对象叫做切面：securityHandler

把advice应用到目标对象的方法的过程叫Weaver：invoke

Pointcut： 用在目标对象的所有方法上（也可以做判断，对指定的方法做验证。通过Method对象拿到方法名，对方法名做验证。符合验证要求的就做验证 指定pointcut）

Spring的连接点是方法。Joinpoint(只关注方法，不关注类和对象)

**JDK动态代理的问题：**

目标类的所有的方法都要执行一遍权限验证

**解决方法：**

使用Method获得调用的方法名，对方法名做字符串匹配。

# AOP面向切面

1. **使用Spring的aop注解开发**

导入支持aop的jar包。

aspectjrt.jar

aspectjweaver.jar

注解类的编写：

@Aspect //指定SecurityHandler为切面类

Public class SecurityHandler{

/\*所有包下所有类下以add开头的所有参数的方法都支持加advice

\* allAddMethod：表示pointcut的标识 \*/

@Pointcut(“execution(\* add\*(..)”)

private void allAddMethod(){}//此标识没有方法名没有方法体没有返回值。

/\*定义advice，标识在哪个切入点（pointcut）何处织入此方法 \*/

@Before (“allAddMethod()”)

public void checkSeurity(){}

配置：(让spring自动启动注解开发方式)

<aop: aspectj-autoproxy />

<bean id=” securityHandler” class=”cn.spring. SecurityHandler”/>

<!—通过工厂创建的是代理对象。

id所标识的对象是代理对象，不是目标对象。-- >

<bean id=” userManager” class=”cn.spring.UserManagerImpl”/>

**创建切面对象：SecurityHandler类对象**

Spring通过读取配置信息和注解信息，识别pointcut,aspect,advice对应的对象是哪个

**生成目标对象：UserManagerImpl类对象**

<aop: aspectj-autoproxy />配置

会判断目标对象中有没有方法符合pointcut指定的条件(以add开头的方法>=1 个)

如果有就创建一个实现InvocationHandler接口的类的对象。

为目标对象UserManagerImpl创建代理对象

调用代理对象的invoke()方法，（代理对象调用invoke()方法）

在invoke()方法中织入advice,

**如果没有任何方法符合pointcut的话就只创建目标对象，不会创建代理对象。**

**BeanFactory一创建会创建四个对象：(目标对象中有符合条件的方法时)**

创建切面对象：SecurityHandler类对象

生成目标对象：UserManagerImpl类对象

实现InvocationHandler接口的类的对象

目标对象UserManagerImpl的代理对象

**BeanFactory一创建只创建两个对象：(目标对象中没有符合条件的方法时)**

创建切面对象：SecurityHandler类对象

生成目标对象：UserManagerImpl类对象

测试类：

BeanFactory factory=new ClassPathXmlApplicationContext(“applicationContext.xml”);

//创建目标类的代理对象**(一定要用接口类型的变量接收getBean结果)**

UserManager userManager=(UserManager)factory.getBean(“userManager”);

userManager.add();**//将验证checkSeurity()方法织入代理对象的invoke()方法中，在代理对象的invoke方法中执行checkSeurity()，然后执行目标对象的add()方法**

userManager.delete();**//由于delete不符合pointcut指定的条件，代理对象直接执行invoke()方法的时候不会执行checkSeurity()方法。**

使用注解开发的好处：减少配置量，结构清晰

缺点：如果注解复杂就很难维护。

1. **使用配置实现aop开发**

**定义一个类：**

public class SecurityHandler{

Public void add(){}

}

**配置：applicationContext.xml：**

<bean id=” securityHandler “ class=”cn.spring. SecurityHandler”/>

//如果创建了代理对象，这里的userManager一定是代理对象，如果没有创建代理对象，这里的userManager则是目标对象

<bean id=” userManager “ class=”cn.spring. UserManagerImpl”/>

<aop: config>

//指定securityHandler为切面, security为切面标识

<aop:aspect id=”security” ref=”securityHandler”>

// allAddMethod为pointcut标识，expression：指定pointcut匹配条件(作用范围)

<aop:pointcut id=”allAddMethod” expression=”execution(\* \*(..))”/>

// SecurityHandler类中的add()方法作为advice,

<aop: before method=”add” pointcut-ref=”allAddMethod”/>

</aop:aspect>

</aop:config>

如果有方法匹配pointcut标签下的expression指定的条件，创建四个对象（**目标对象，代理对象，实现InvocationHandler接口的类对象，切面对象**）

如果没有有方法匹配pointcut标签下的expression指定的条件，创建两个对象（**目标对象，切面对象**）

**注解方式和配置方式：**

注解：配置量小，适合不经常维护的项目

配置：配置量大但是更加清晰，便于维护。

1. **在advice中得到方法的名字和参数名(了解)**

**切面类的定义：**（其余都和使用配置完成aop开发流程一致，配置也相同）

public class SecurityHandler{

public void security(JointPoint joinPoint){

Object[] args=joinPoint.getArgs();//拿到方法的每一个实参

String name=joinPoint.getSignature().getName();//拿到方法的方法名

}

}

1. **目标对象没有实现接口（JDK的动态代理只能对实现接口的类）**

所以不能使用JDK动态代理，使用Cglib代理。

1. **导入cglib的jar包：**
2. **定义不实现接口的目标类**

public class UserManager{

public void add(){}

}

1. **定义切面类**

public class SecurityHandler{

public void add(){}

}

**配置：applicationContext.xml：**

<bean id=” securityHandler “ class=”cn.spring. SecurityHandler”/>

//如果创建了代理对象，这里的userManager一定是代理对象，如果没有创建代理对象，这里的userManager则是目标对象

<bean id=” userManager “ class=”cn.spring. UserManagerImpl”/>

<aop: config>

//指定securityHandler为切面, security为切面标识

<aop:aspect id=”security” ref=”securityHandler”>

// allAddMethod为pointcut标识，expression：指定pointcut匹配条件

<aop:pointcut id=”allAddMethod” expression=”execution(\* \*(..))”/>

// SecurityHandler类中的add()方法作为advice,

<aop: before method=”add” pointcut-ref=”allAddMethod”/>

</aop:aspect>

</aop:config>

**测试类：**

BeanFactory factory=new ClassPathXmlApplicationContext(“applicationContext.xml”);

//创建目标类的cglib代理对象

UserManager userManager=(UserManager)factory.getBean(“userManager”);

userManager.add();//符合pointcut配置的expression表达式的方法，也会织入

**如果目标类实现接口，默认使用JDK动态代理，如果实现了接口也非要使用cglib代理，就在applicationContext.xml文件中添加配置：**

**<aop : aspectj-autoproxy proxy-target-class=”true”/>**

没有实现接口的目标类，会使用Cglib代理。

**JDK动态代理 和Cglib动态代理**

**相同点：**都为目标对象生成代理对象，都作为代理完成aop

在不修改源码的情况下，增加功能（cglib代理可以实现懒加载）

**不同：**

JDK动态代理：只能对实现接口的对创建代理对象（实现接口：默认使用JDK动态代理）

Cglib动态代理：可以对实现接口的类创建代理对象，也可以对未实现接口的类做动态代理。（未实现接口：默认使用cglib动态代理）

**Cglib代理：（原理）**

代理对象是目标对象的子类，

目标类不能定义为final

目标类一定要有显示的无参的构造方法

**JDK动态代理原理：**

创建代理对象和目标对象实现同一个接口

代理对象在实现InvocationHandler接口的类的newPoxy()方法中通过newProxyInstance（）方法创建

实现InvocationHandler接口的类中要复写invoke方法

invoke方法由当前this指定的类调用。

通过实现 InvocationHandler 接口创建自己的调用处理器；

通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理类；

通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型；

通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。

**Spring和Hibernate集成：**

**日志：（记录用户的操作和 时间）**

日志分为：**操作日志**、安全日志、事件日志

public class Log{

private String type；//日志类型

private String detail; //日志详细

private Date date;//生成日志的时间

}

Hibernate创建两张表：t\_user,t\_log

业务层 实现接口方便实现aop（两个业务：两个接口、两个实现类）

public interface LogDao{

public void add(Log log);

}

public class LogImpl{

public void add(Log log){}

}

Hibernate中：**sessionFactory.getCurrentSession();/自动将session和线程绑定**

需要在hibernate.cfg.xml配置

跨资源事务的配置：

<property name=” hibernate.current\_session\_context\_class”>jta<property>

本地事务的配置：

<property name=” hibernate.current\_session\_context\_class”>thread<property>

public interface UserDao{

}

public class UserImpl{

**在存储用户的时候，同时存储日志信息。放在同一个方法的try{}catch块中。利用同一个线程（同一个session）存。（先存储用户再存储日志）**

**存储用户的session和存储日志的session是同一个session。**

**使用完后session不需要马上关闭，因为和线程绑定了是安全的。**

**Session的关闭时间：线程结束后，会自动关闭session**

}

**请求结束：**jsp把结果页面的数据解析完放入response传给浏览器显示后，请求结束。

请求结束了当前线程也结束。线程结束就会关闭session。

**Hibernate和spring的集成：**

**hibernate只做事务的存储和加载。**

**Spring通过配置：创建sessionFactory，拿到session对象，开启事务，提交事务，发生异常的时候自动回滚，线程结束后自动关闭session。**

1. 创建java项目
2. 导入spring和hibernate的jar包（测试环境没有jar包重复的问题），真实项目要删除重复的包。一般删除低版本的jar包，留下高版本的jar包。
3. 模型层 Model 有实体化类和相应的hbm.xml文件，配置到Hibernate.cfg.xml文件中（自动生成表）
4. Hibernate的配置文件Hibernate.cfg.xml放到src下，Spring的配置文件 applicationContext.xml放到src下
5. 业务层都实现相应的接口（便于使用aop）
6. Spring的主配置文件：applicationContext-common.xml，applicationContext-beans.xml

**通过配置Spring的主配置文件创建Hibernate的sessionFactory**

<bean id=”sessionFactory” class=”org.springframework……**LocationSessionFactoryBean**”>

<!-- 先创建对象(代理对象)，后注入信息，configLocation中包含hibernate.cfg.xml配置文件中的信息 -- >

<property name=”**configLocation**”>

<value>classpath:hibernate.cfg.xml</value>

</property>

</bean>

**创建sessionFactory的作用：**

拿到session对象

管理二级缓存

生成全局性事务

Hibernate是先读取配置信息后创建sessionFactory工厂。

**Spring先生成工厂(proxy----代理工厂)再拿到hibernate的配置信息。**

Spring配置生成的工厂拿到的session是和线程绑定的session对象。是线程安全的。

**configLocation**是LocationSessionFactoryBean类中的一个属性，类中具有setConfigLocation()方法。（hibernate的配置文件信息存在configLocation属性中）

先创建sessionFactory的代理对象，读取主配置文件

在BeanFactory中创建sessionFactory

**配置事务管理器：(通过注入sessionFactory工厂信息生成事务管理器)**

事务管理器的作用：**根据事务的传播特性为相应的方法提供全局性的事务**

<bean id=”**transactionManager**” class=”org.springframework…HibernateTransactionManager>

<property name=”sessionFactory”>

<ref bean=”sessionFactory”>

</property>

</bean>

**配置事务的传播特性：**

**事务传播特性：通过事务管理器配置事务的传播特性决定哪些方法怎么样的使用事务（指导相应的方法如何使用事务）**

**使用事务的方法：自己创建或者使用别的方法创建的事务**

<tx:advice id=”txAdvice” transaction-manager=” **transactionManager**”>

<tx:attributes>

<!--任何线程的任何包下的任何类只要有add开头的方法。执行以add开头的方法时，先判断有没有全局性事务，如果没有全局性事务就为这个方法创建一个新的全局性事务，如果有就直接使用。下一个add开头的方法也是。如果update的事务传播特性也是REQUIRED ，就会直接使用上一个全局性事务（add开头的方法创建的那个）。-- >

<tx:method name=”add\*” propagation=”REQUIRED”>

<tx:method name=”update\*” propagation=”REQUIRED”>

<tx:method name=”delete\*” propagation=”REQUIRED”>

<tx:method name=” \*” read-only=”true”>

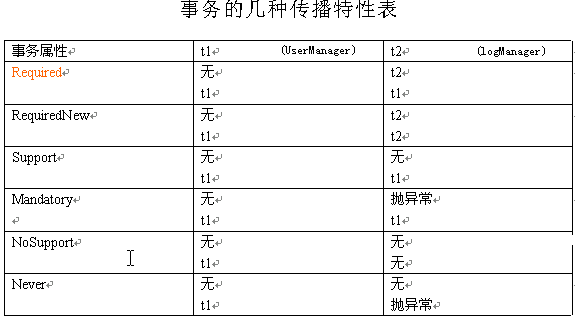
</tx:attributes>

</tx:advice>

**propagation=”REQUIRED”** ：执行以add开头的方法时，先判断有没有全局性事务，如果没有全局性事务就为这个方法创建一个新的全局性事务，如果有就直接使用。下一个方法也是。

propagation=”REQUIREDNew”总是创建新的事务

propagation=”Never” 前一个方法有事务后一个方法就抛出异常；前一个方法没有事务后一个方法也没有事务。

了解事务的几种传播特性：决定哪些方法如何使用事务

\*1.PROPAGATION\_REQUIRED: 如果存在一个事务，则支持当前事务。如果没有事务则开启（使用最普遍）如UserManager在使用事务时发现事务没有开启，则开启事务，这时logManager在使用事务时发现事务已经被UserManager开启，则使用该事务，这样可保证两个业务使用同一个事务

2.PROPAGATION\_SUPPORTS: 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行

3.PROPAGATION\_MANDATORY: 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

4.PROPAGATION\_REQUIRES\_NEW: 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

5.PROPAGATION\_NOT\_SUPPORTED: 总是非事务地执行，并挂起任何存在的事务。

6.PROPAGATION\_NEVER: 总是非事务地执行，如果存在一个活动事务，则抛出异常

7.PROPAGATION\_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TransactionDefinition.PROPAGATION\_REQUIRED 属性执行。

**Aop的配置（哪些方法使用事务）不需要切面**

<aop:config>

**expresssion**：事务的边界定义在业务层（指定具体包名）spring.UserDaoImpl\*.\*(. .))，边界为spring.UserDaoImpl包下的所有类的所有参数类型所有方法

**事务的边界放在业务层的原因：**因为大型项目的开发底层数据库可能不一样，如果把事物的传播特性定义在持久层会导致别的数据库是无法使用的。所以事务边界要定义在业务层。（因为业务层是统一的）

**为什么不定义aop:aspect**：定义切面的目的是找的advice(security()方法),因为txAdvice就是事务的传播特性，不需要再定义了。

**advice-ref** ：给某些功能加事务(事物的传播特性)

<aop:pointcut id=”addManagerMethod” expresssion=”execution(\* spring.UserDaoImpl\*.\*(. .))”>

<aop: advisor pointcut-ref=” addManagerMethod” advice-ref=”txAdvice” />

</aop:config>.

pointcut:指定哪些类的哪些方法加上advice

**目标类的配置：**

**(目标类中的方法符合pointcut，就会织入advice。为目标类创建代理类，调用invoke方法)**

**logManager**：是代理对象

**LogManagerImpl**：继承了HibernateDaoSupport类

<bean id=”logManager” class=”cn.spring.LogManagerImpl”>

**sessionFactory**：HibernateDaoSupport类中有setSessionFactory()和getSessionFactory()方法，当前LogManagerImpl类中没有sessionFactory属性也没有set和get方法。但是通过继承HibernateDaoSupport会有父类的set和get方法。

**<property name=”sessionFactory” ref=”sessionFactory”>//调用setSessionFactory()方法,通过注入sessionFactory，创建hiberanteTemplate**

setSessionFactory( )方法真正的是在给HibernateTemplate类型的hibernateTemplate属性设值，

HibernateTemplate相当于升级版的Session。可以通过SessionFactory创建HibernateTemplate对象。默认和线程绑定。getCurrentSession()。

</bean>

**userManager：**为代理对象

**UserManagerImpl：**继承了HibernateDaoSupport类

<bean id=”userManager” class=”cn.spring.UserManagerImpl”>

<property name=”sessionFactory” ref=”sessionFactory”>

**logManager:**给userManager对象的logManager属性设置的代理logManager对象

<property name=”logManager” ref=”logManager”>

</bean>

怎么手工拿到session对象？

HibernateTemplate hibernateTemplate=new HibernateTemplate (sessionFactory);

创建BeanFactory:（做以下这些事情）

1. sessionFactory工厂创建（代理方式创建）
2. 通过sessionFactory创建事务管理器(对象)
3. 通过事务管理器配置事务传播特性
4. 配置aop指定事务边界，哪些方法使用事务，使用事务的方式是什么样的。通过事务传播特性织入事务（使用事务）
5. 创建目标对象（LogManager）
6. 发现目标对象（LogManager）的方法符合pointcut，创建目标对象的代理对象
7. 给目标对象注入属性 sessionFactory，目标对象所属类继承了HibernateDaoSupport类。创建了HibernateTemplate（Session）对象。

在业务类中：

Public class UserManagerImpl implements UserManager{

private LogManger logManger;

//logManager为代理对象，配置中已经注入代理对象了。

执行代理对象的方法时，会执行代理对象所指的this的invoke方法

logManager.add(log);

}

**this.getHibernateTemplate()**.save(user); HibernateTemplate相当于session

同一个方法中使用的session不一定是同一个，但是使用的是同一个事务。

同一个事务提交和回滚都同步

**Spring管理事物的提交：**自动提交，所有的线程操作完成就自动提交事务，不需要显示的写出事务提交代码。Spring会自动在发生异常的时候回滚。

**Spring 对于异常：**

运行异常：出错也回滚（RuntimException）

编译异常：只出错不回滚(Exception)

**并发性和安全性：**加上悲观锁，使用spring的声明式事务达到安全性的极致（使得拿到同一个事务）。

**spring使用声明式事务，替hibernate管理事务。**

测试类：

通过工厂得到的代理对象

调用invoke()方法时，会先织入advice()。

织入事务会先执行事务传播特性：txAdvice

判断是否方法是否符合pointcut

判断事务传播特性 Requried(有全局性事务就直接使用，没有就创建)

根据事务传播特性找到事务管理器 transactionManager

**userManager.add(user);**

将事务织入add方法中,使得当前add()方法在不修改源代码的情况下增加新功能。

通过invoke将advice先执行。

# Spring和struts的集成

**Spring和struts集成的第一种方案：**

**搭建环境：**

1. 创建web项目
2. 导包--- struts1.0的jar（9个）+jstl的jar包（2个）
3. 配置struts1.0 的默认的国际化资源文件，放到src下<MessageResources.properties>
4. 在WEB-INF下的web.xml中配置struts1.0<ServletAction>
5. 配置struts-config.xml文件 <form-bean><action-mappings><message-resources><plug-in><global-exceptions><global-forwards>
6. 导入spring的jar包（7个）
7. 配置spring的配置文件 测试环境放在 src下：applicationContext.xml，真实环境放到WEB-INF下
8. 显示信息可以配置log4j.properties(也可以不配置)

**登录示例：**

请求login.do,交给struts处理

Struts截取请求路径，找到该请求对应的配置信息

Processpath<截取请求路径>

porcessLocale<国际化配置>

processMapping<根据path找到ActionMapping对象>

processActionForm<创建ActionForm对象>

processPopulate<收集表单数据，一个模块一个表单>

processActionCreate<对象单例且同步，map>

processActionPerform<执行execute方法，返回ActionForward对象>

在execute方法中调用业务层的方法，设定转向信息:

**1、将业务对象交给spring创建，把UserManager对象配置在applicationContext.xml文件中。**

<bean id=”userManager” class=”cn.spring.UserManagerImpl”>

**2、在execute方法中通过new 生成BeanFactory**

BeanFactory factory=new ClassPathXmlApplicationContext(“applicationContext.xml”)

**3、通过工厂beanFactory拿到业务层对象，调用业务层对象的方法**

UserManager userManager=(UserManager)factory.getBean(“userManager”);

userManager.add();

ProcessActionForward<完成跳转>

**缺点：BeanFactory会创建多次（BeanFactory是重量级的不能创建多次）**

**优点：方法简单，效率低，依赖性强**

**Spring和struts集成的第二种方案：**

**将beanFactory放入内置对象application中，保证application内置对象在初始化的时候创建，并且只创建一次**

在**servlet**的init()方法，**Filter**的init()方法,**PlugIn**的init()方法中创建都可以。

在**listener**中创建（保证BeanFactory创建的最早，比ActionServlet还早）

创建一个监听器： （**对application对象的创建和销毁做监听**）

1、实现**ServletContextListener**接口

2、实现**contextInitialized**()方法，在该方法中注入事件对象，

3、通过事件对象拿到application内置对象。

4、在contextInitialized()方法中创建BeanFactory对象

5、通过application.setAttribute(“beanFactory”,beanFactory)方法将BeanFactory对象设置到application内置对象中。

6、使用的时候：

1. request.getSession().getServletContext();//拿到application内置对象
2. application.getAttribute(“beanFactory”);//通过application拿到工厂对象

Web.xml中的配置：

Spring自带的ContextLoaderListener类，实现了ServletContextListener接口以及接口中的方法。

<listener-class>……ContextLoaderListener<listener-class>//配置监听器

<!-- </context-param>:封装公有全局的初始化参数信息，存在application内置对象中，所有组件都可以使用。application.getInitParameter(“contextConfigLocation”);- >

**<context-param>**

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

<!-- Spring的配置文件可以放在src下可以放在WEB-INF目录下- - >

<param-value>classpath\*:application-\*.xml,/WEB-INF/ application-\*.xml </param-value>

**</context-param>**

进行以上配置后的Tomcat启动创建对象的过程：

1、Tomcat一启动就创建ContextLoaderListener监听器对象，（对application的状态变化做监听）

2、在监听器中创建application对象，

3、把<context-param>中的全局初始化参数信息封装到application内置对象。

4、调用ContextLoaderListener的contextInitialized()方法，

5、在contextInitialized()方法中注入事件对象（ServletContextEvent）

6、通过事件对象得到application对象

7、通过application对象从配置信息中拿到初始化参数（BeanFactory的配置信息）

String contextConfigLocation =application.getInitParameter(contextConfigLocation)

1. 判断初始化参数是否null < contextConfigLocation ==null>
2. contextConfigLocation不为null的时候就创建工厂（BeanFactory）

BeanFactory beanFactory=new ClassPathXmlApplicationContext(contextConfigLocation)

1. 创建工厂后返回WebApplicationContext对象，（WebApplicationContext是BeanFactory的子类）
2. 将创建工厂后返回的对象设到application内置对象中

application.setAttribute(“very long 的 属性名”, WebApplicationContext);

手动写监听器创建BeanFactory：

public class **MyBeanFactoryListener** implements ServletContextListener{

public void contextInitialized(ServletContextEvent sce){

ServletContext application=sce.getServletContext();//拿到application对象

//得到web.xml中的全局配置信息，applicationContext.xml文件的位置

String contextLocation=application.getInitParamter(“contextConfigLocation”);

//根据配置文件applicationContext.xml创建BeanFactory对象

BeanFactory factory=new ClassPathXmlApplicationContext(contextLocation);

//将创建的BeanFactory对象存到application内置对象中

application.setAttribute(“beanFactory”,factory);

}

//空实现，因为用不到嘚哈哈哈

public void contextDestroyed(ServletContextEvent sce){}

}

在struts的Action类中使用工厂（在Action的excute()方法中）

1. 通过request内置对象拿到application内置对象

request.getSession().getServletContext();

1. 通过application内置对象拿到工厂对象

**使用spring定义的监听器类**ContextLoaderListener**拿到BeanFactory对象的方法：**

BeanFactory factory = WebApplicationContextUtils.

getRequiredWebApplicationContext(application);

**使用自定义的监听器类**MyBeanFactoryListener**拿到BeanFactory对象的方法：**

BeanFactory factory=application.getAttribute(“beanFactory”);

1. 通过工厂对象拿到业务对象（**业务类配置在applicationContext.xml的中**）

UserManager userManager=(UserManager)factory.getBean(“userManager”);

1. 调用业务对象相应的方法

userManager.add(“zs”,”123”);

**使用监听器listener创建BeanFactory的好处：只创建一次，创建的时间特别早**

**缺点：和Action的依赖大**

**Spring和struts集成的第三种方案：**

**(使用spring替struts生成Action并且为Action注入业务层对象)**

**Spring为struts1.0加了一个代理Action、**

**Web.xml文件中：**

将Spring的主配置文件applicationContext.xml通过<context-param>配置全局变量，存在application内置对象中；

配置Spring的监听器ContextLoaderListener

配置struts的ActionServlet

**Struts-config.xml:中的配置信息：**

使用spring提供的Action类 DelegatingActionProxy(暂时代替自定义的cn.spring\_hibernate.LoginAction类)，把业务对象注入该Action中。

<action path=”/login” name=”loginForm”

type=”…… DelegatingActionProxy” scope=”request”></action>

**applicationContext.xml中的配置信息：**

BeanFactory为struts 创建Action对象，使用name标识，而不是id。并且name标识和struts-config.xml中的path属性一致。

<bean **name**=”/login” class=”cn.spring\_hibernate.LoginAction” scope=”proptotype”>

<property name=”userManager” ref=”userManager”/>

</bean>

**DelegatingActionProxy:** 是一个Action类，继承了struts的Action

**DelegatingActionProxy的execute()方法：**

1. 通过application内置对象拿到BeanFactory
2. 通过ActionMapping找到path属性对应的属性值
3. 通过path到BeanFactory取得Action（多例）对象//因为BeanFactory中的name和ActionMapping中的path属性值相同
4. 调用真正的Action对象的execute()方法

**更改MyEclipse中的项目名**

重新发布：

右击项目—》 properties🡪 myEclipse🡪web—》webContext Root 将项目名改为一致就可以了。

# Struts\_Spring\_Hibernate的集成

**SSH集成示例：**

**搭建环境：**

相同的jar包取高版本的(common-log.jar，log4j.jar，junit.jar)

1. 创建web项目
2. 导入struts1的jar包，ext下的9个jar包，jstl的2个jar包。
3. 导入struts1的国际化资源文件MessageResource.properties(基名+local：MessageResource\_en\_US.properties)
4. 在WEB-INF下的web.xml中配置struts1.0<ServletAction>
5. 配置struts-config.xml文件 <form-bean><action-mappings><message-resources><plug-in><global-exceptions><global-forwards>
6. 导入spring的jar包（7个）Spring的主jar,log4j的jar包，commons-logging.jar，aspectj下的jar包（2个），junit测试的jar包，cglib代理的jar包。（共7个，spring2.x）
7. 配置spring的配置文件 测试环境放在 src下：applicationContext.xml，真实环境放到WEB-INF下
8. 显示信息可以配置log4j.properties(也可以不配置)

**集成原理 ：**

Spring替Hibernate生成工厂，处理事务

Spring替struts生成Action,注入业务对象

**Tomcat启动：**

**Web.xml文件---Listener：**

1.、创建监听器对象（ContextLoaderListener）

2、application创建，将全局初始化参数封装到application

3、监听器调用contextInitialized()方法注入事件对象，通过事件对象拿到application

4、从application中拿到初始化参数application.getInitParameter()

5、通过初始化参数取得spring的配置信息，然后根据配置信息创建BeanFactory对象

**6、一创建BeanFactory，会做以下这些事情：**

6.1、创建SessionFactory生成全局性事务，生成session；

6.2、通过sessionFactory创建事务管理器，

6.3、通过事务管理器创建事务传播特性，

6.4根据事务传播特性调用AOP，根据事务传播特性把事务织入相应的方法里；

6.5、生成业务对象，由于业务类继承了HibernateDaoSupport,所以可以直接在业务对象中注入sessionFactory(HibernateTamplate)

6.6、调用业务对象的方法时，发现方法符合pointcut,生成目标对象的代理对象，在方法中织入advice；

6.7、创建LoginAction对象，这是对象是多例的（延迟创建）并且把业务层的代理对象注入Action,在Action中可以直接使用）

**Web.xml: --Filter**

Spring 的编码拦截过滤器，CharacterEncodingFilter

执行init()方法，注入filterConfig对象，config中封装了编码信息。拿到编码方式，设置到实例全局变量中。然后对请求进行编码拦截chain.doFilter(){ request.setCharacterEncoding(encoding);}

Spring的懒加载拦截器 OpenSessionInview

1、不要创建工厂（在BeanFactory中创建）

2、也不用将session和线程绑定（Bean中线程安全，getCurrentSession()）

3、主要做结果拦截，判断session是否和当前线程绑定，如果绑定就解除绑定。

**Web.xml---Servlet**

创建Servlet对象，执行初始化方法，读取struts的配置文件

数据库中的表什么时候创建？

tomcat一启动就创建, BeanFactory工厂创建后就会创建SessionFactory,然后根据hbm.xml创建对应的表。

**项目执行流程：**

1、发送请求访问struts，截取请求路径到struts-config.xml文件中找到对应的配置信息ActionMapping

2、通过配置信息 找到name，通过name找到对应的FormBeanConfig，找到ActionForm类，通过反射生成对象放在scope中。根据ActionMapping对象创建代理Action对象（DelegatingActionProxy），

3、根据application拿到bean工厂，找到BeanFactory的name，拿到LoginAction对象，调用目标对象的execute方法。（这里的name和struts-config.xml配置文件的<action>标签中的path一致）

4、在LoginAction的execute方法中：

4.1、拿到表单数据

4.2、创建业务对象，调用业务对象的方法（业务对象已经被注入Action中了，并且是代理对象）

4.3、调用代理对象的方法，执行invoke，根据事务传播特性织入advice。

周末任务：

SSH的银行项目

**物料项目;**

**表示层**主要有jsp，组件： struts1的Action

表示层模块：注册登录，物料维护，国际化模块（使用DispatchAction：一个模块一个Action）

配置struts-config.xml文件：action,exception,form-bean,and so on

**业务层**组件： Spring,包含三个配置文件，生成业务对象，注入关联属性

完成持久层SessionFactory的创建,事务管理器的创建，事务传播特性的创建，aop

主要模块是物料维护：物料的添加，查找，删除，

业务层的工具包：定义异常类，一个项目只用一个异常类

Jstl的函数库：为添加和编辑数据 提取数据

pageModel分页技术

HibernateUtil类，测试的时候拿到session对象完成操作。

**持久层**组件：Hibernate ，对持久类完成存储和加载

环境的搭建：

引入struts的jar包和jstl的jar包，主配置文件Struts-config.xml

自定义函数库：引用jstl的tld文件

Spring7个jar包，主配置文件applicationContext.xml文件

Hibernate的jar包，主配置文件hibernate.cfg.xml

log4j.jar,junit.jar,cgllib.jar,有重复的jar包时留下最新的jar包（高版本的）

**实体类分析--DataDict**

public class DataDict{int id ,String name} （两个属性）

public class ItemCategory extends DataDict（没有属性）

public class ItemUnit extends DataDict（没有属性）

**配置hbm.xml文件：**

**单表继承映射：**

<hibernate-mapping package=”cn.spring.hibernate.bean”>

<class name=”DataDict”table=”t\_data\_dict”>

<id name=”id”>//id自动生成

<generator class=”assigned”/>

</id>

<discriminator **column**=”category” type=”string”>//鉴别器字段

<property name=”name” not-null=”true”/>

配置子类：

<**subclass** name=”ItemCategory” discriminator-value=”item\_category”/>

<**subclass** name=”ItemUnit” discriminator-value=”item\_unit”/>

</class>

</hibernate-mapping>

初始数据用于测试：使用HibernateUtil 工具

存储数据的时候会自动往鉴别器字段加鉴别值,查询的时候也会**自动添加鉴别器字段使用多态查询。**

**实体类 ---Item：**

public class Item { itemNo,Itemname.spec,pattern ,//普通属性

ItemCategory category ，ItemUnit unit //关联属性

<hibernate-mapping package=”cn.spring.hibernate.bean”>

<class name=”Item” table=”t\_item”>

<id name=”id”>//id用户手动生成

<generator class=”assigned”/>

</id>

<property name=”name”not-null=”true” />

<property name=”spec” />

<property name=”pattern” />

//配置外键：（管理属性的配置）

<**many-to-one** name=”category” />**参照 t\_data\_dict**

<**many-to-one** name=”unit” />**参照 t\_data\_dict**

</class>

</hibernate-mapping>

三个对象：**Item ,ItemCategory,ItemUnit**

Item 的 category属性关联ItemCategory对象

Item 的 unit属性关联ItemUnit对象

数据库中有两张表： **t\_data-dict，t\_item**

**SSH配置文件解析：**

**Hibernate.cfg.xml：(Hibernate的配置文件)**

配置连接数据库相关信息，显示建表语句信息，

检测数据库中有没有表，没有表就自动生成表的配置，以及映射文件配置信息

**Struts-config.xml：（struts的配置文件）**

配置Action类和ActionForm类，全局跳转以及异常信息，国际化资源文件的配置。

注：这里的Action类统一使用spring中的DelegatingActionProxy代替，真正的Action配置在applicationContext.xml文件中。

**applicationContext-\*.xml：（spring的配置文件）**

**applicationContext-common.xml：**

配置自动创建SessionFactory的bean，配置根据SessionFactory创建的事务管理器的bean，配置事务传播特性的bean，配置AOP指定使用事务的边界的bean。

**applicationContext-beans.xml：**

配置生成业务层实现类对象的bean，并且注入sessionFactory，让spring自动创建hibernateTemplate对象（相当于session），供业务对象直接使用。

**applicationContext-actions.xml：**

配置生成Action对象的bean

**注意：**这里的bean标签 主标识不是id,而是name（与struts 特定的配置),并且**name**指定的内容要和struts-config.xml配置中的对应的<action>标签的**path**指定的吗一致。在配置action的bean中国注入业务层对象，这里注入的对象是代理对象。eg:

<bean **name=”/login”** class=”cn.ssh.action.LoginAction”>

<property name=”userManager” ref=”userManager”/>

</bean>

**Web.xml文件---Listener：**

1.、创建监听器对象（ContextLoaderListener）spring中的类

2、application创建，将全局初始化参数封装到application

3、监听器调用contextInitialized()方法注入事件对象，通过事件对象拿到application

4、从application中拿到初始化参数application.getInitParameter()

5、通过初始化参数取得spring的配置信息，然后根据配置信息创建BeanFactory对象，把工厂存到application内置对象中setAttribtue()

**6、一创建BeanFactory，会做以下这些事情：**

6.1、创建SessionFactory生成全局性事务，生成session；

6.2、通过sessionFactory创建事务管理器，

6.3、根据事务管理器配置事务传播特性，

6.4根据事务传播特性调用AOP，根据事务传播特性把事务织入相应的方法里；（事务边界 -----业务层）

6.5、创建业务对象：ItemManagerImpl对象，由于目标对象所属类ItemManager继承了HibernateDaoSupport, 所以可以直接在业务对象中注入sessionFactory ，拿到Session(HibernateTamplate)

6.6、调用业务对象的方法时，发现业务层对象的方法中有方法符合pointcut,生成目标对象的代理对象，在方法中织入advice；

6.7、创建Action对象（struts-config.xml中配置了几个action标签这里就创建几个Action对象），这里创建的对象是多例的 scope=”prototype”（延迟创建）并且把业务层的代理对象注入Action,在Action中可以直接使用）

**在spring的配置文件applicationContext.xml中：**

**创建自定义函数库的对象,并且注入sessionFactory。**

<bean id="myfunction" class="com.bjsxt.drp.business.util.**Functions**">

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

</bean>

**Functions：**未继承HiberanteDaoSupport类，但是**Functions类中定义了静态的sessionFactory属性，并且提供了公有的set方法**，所以可以注入sessionFactory对象。但是Functions类中所有方法都是静态方法，静态方法没有this指针。

SessionFactory中的**HibernateTemplate对象是一个实例全局变量**，所以即便是注入了sessionFactory对象，但是在Functions类中，无法直接通过this.getHibernateTemplate（）拿到HibernateTemplate 对象。只能在Functoins中手动创建HibernateTemplate对象。

HibernateTemplate ht=new HibernateTemplate(sessionFactory);**//这个sessionFactory属性不能是实例的。因为静态方法中不能使用非静态变量**。

静态全局变量没有this指针，归该类所有对象公有。该类所有对象都可以使用静态全局变量。

**通过注入的方式如何给类的静态全局变量设值？**

**applicationContext.xml中的配置：**

<bean id=”functions” class=”cn.Functions”>

<porperty name=”sessionFactory” ref=” sessionFactory”>

</bean>

**Functions类：**

private static SessionFactory sessionFactory=null;

Set方法：

public void setSessionFactory(SessionFactory sessionFactory){

**Functions.sessionFactory= sessionFactory;**//通过类赋值

}

HibernateTemplate的find()方法

Ht.find(“from ItemCategory a order by a.id”);多条查询

此方法返回的是List集合，内部是callback

**Callback**模式：(**spring内部如何织入事务 advice**)

**Web.xml: --Filter**

Spring 的编码拦截过滤器，CharacterEncodingFilter

执行init()方法，注入filterConfig对象，config中封装了编码信息。拿到编码方式，设置到实例全局变量中。然后对请求进行编码拦截chain.doFilter(){ request.setCharacterEncoding(encoding);}

Spring的懒加载拦截器 OpenSessionInviewFilter

1、不要创建工厂（在BeanFactory中创建）

2、也不用将session和线程绑定（Bean中线程安全,getcurrentSession()）

3、主要做结果拦截，判断session是否和当前线程绑定，如果绑定就解除绑定。

**事务是针对事务的变化而存在的**

**查询是（find）非事务操作，可以不需要事务**

**CallBack详解：**

**HibernateCallback:** 是一个接口

方法 doInHibernate(Session session);

在业务层：ItemManager的findAllItems( )方法中：

this.getHibernateTemplate().executeFind( **new HibernateCallback( ){**

**public Object doInHibernate(Session session){**

**return session.createQuery(“from Item t ”);**

**}**

**}**

);

Hiberante的织入：

// Add：实现HibernateCallback的类

public class **Add** implements **HibernateCallback{**

publc void **doInHibernate** (Session session){

User user=new user();//创建user对象

user.setName(“Tom”);

session.save(user);

}

}

**new CRUD( )**.persistent(**new Add()**);

**}**

public class **CRUD** **{**

**public void persistent(HibernateCallback hibernateCallback){**

**//创建真正的session对象**

**Session session=HibernateUtil.getSession();**

**Try{**

**Session.beginTransaction();**

**//调用new Add()对象的doInHibernate方法，相当于执行了save(user)**

**hibernateCallback.doInHibernate();**

**session.getTransactionCommit();**

**}catch(Exception e){**

**session.getTransaction().rollback();**

**}**

**}**

**}**

在(CRUD)类中调用创建真正的session对象，并且commit()**（匿名类的理解类似，只是缺省了一个定义类的过程，直接在另一个的方法中将匿名类作为参数传入，然后调用）**

Spring后台自动创建真正的对象，完成匿名类中定义的操作。这就是spring织入事务的原理与过程。我们只需要写 save()对象的方法，后面的由spring自动处理。