# 1.框架概述:

struts2:基于MVC模式应用层框架

主要是作为控制层组件,javabean,jsp

Hibernate:持久化层组件,简化jdbc操作

Spring:主要有6大模块功能:

事务管理

与其他框架整合

专业术语:

高内聚,低耦合:

类内部的关系越紧密越好,类与类的关系越少越好

侵入式设计:

引入的组件对现有的类的结构会有影响.

侵入式设计的例子:struts2

非侵入式设计:

引入的组件对现有的类的结构没有影响.

非侵入式设计:Hibernate,Spring

IOC容器:

Inversion of Control 控制反转的意思,控制反转容器

User user = new User();自己控制对象的创建

现在需要对象,自己不创建,交给外部的容器创建,叫控制反转

IOC容器=bean.xml配置 + ApplicationContext容器类

DL;

dependency injection 依赖注入

创建对象后,处理对象的依赖关系

User user = new User();

user.setAddress(..);//需要DL(依赖注入)

6大模块

SpringCore:核心功能

主要提供了IOC容器创建对象/依赖关系

Spring Aop:面向切面编程

Spring Web:对web开发的支持,可以与Struts2无缝整合

Spring Jdbc:对jdbc操作的支持

Spring ORM:对hibernate提供支持

Spring JEE:对javaee开发其他模块的支持(ejb)

# 2.加载applicationContext.xml文件的方式

方式1: ClassPathXmlApplicationContext,从类路径加载,只需放在 src/main/resources/下就可以了

方式2: FileSystemXmlApplicationContext,从绝对路径加载或者相对路径,如果是相对路径,则根目录为项目目录,当然不会选择绝对路径,一般都是相对路径

方式3: BeanFactory bf = new XmlBeanFactory(new ClassPathResource(“xxx”))

ApplicationContext和BeanFactory容器的区别:

1. **ApplicationContext**容器在进行初始化时,会将其所有Bean对象进行创建

优点:响应速度快

缺点:占用系统资源大

1. **BeanFactory**容器在进行初始化时,不会将其所有Bean对象进行创建,而是在调用getBean()时创建.

优点:占用系统资源小

缺点:响应速度满

# 3. Bean的装配

Bean的装配概念:容器根据代码要求创建Bean对象后再传递给代码的过程,称为Bean的装配

## 3.1默认装配方式

默认装配方式:通过getBean()方式从容器获取指定的Bean实例,容器会调用Bean类的无参构造函数

|  |
| --- |
| <bean id="bean1" class="lw.pers.bean.MyBean"></bean> |

|  |
| --- |
| ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  MyBean bean = (MyBean) context.getBean("bean1");  System.out.println(bean); |

## 3.2动态工厂Bean

概念:将动态工厂Bean作为普通Bean来使用,通过创建的工厂bean调用工厂的方法,创建需要的bean对象

实例Bean:

|  |
| --- |
| Public clas MyBean(){  Private int age;  Private String name;  Setter…..  } |

工厂Bean:

|  |
| --- |
| package lw.pers.factory;  import lw.pers.bean.MyBean;  public class MyFactory {  public MyBean getInstance(){  return new MyBean();  }  } |

配置:

|  |
| --- |
| <!--注册动态工厂-->  <bean id="factory" class="lw.pers.factory.MyFactory" ></bean>  <!--普通bean-->  <bean id="bean" factory-bean="factory" factory-method="getInstance"></bean> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  MyBean bean = (MyBean) context.getBean("bean");  System.out.println(bean); |

## 3.3静态工厂Bean

静态工厂类无需要配置工厂的bean,只需要在普通的bean指定由某个工厂的某个静态方法创建.  
工厂类:

|  |
| --- |
| package lw.pers.factory;  import lw.pers.bean.MyBean;  public class MyFactory {  public static MyBean getInstance(){  return new MyBean();  }  } |

配置:

|  |
| --- |
| <!--普通bean-->  <bean id="bean" class="lw.pers.factory.MyFactory" factory-method="getInstance"></bean> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  MyBean bean = (MyBean) context.getBean("bean");  System.out.println(bean); |

## 3.4容器中Bean的作用域

bean标签的scope即bean的作用域属性.

其值可以为5种:

1. singleton:单例模式(默认)
2. prototype:原型模式,每次通过getBean获取的都是新的对象
3. request:对于http请求,都将会产生一个不同的Bean对象
4. session:对每个不同的http session，都将产生一个不同的Bean实例
5. global session:每个全局的http session对应一个Bean对象，仅在使用protlet集群时有效,多个web应用共享一个session,一般应用中,global-session和session是等同的。

注意:

1. 对于scope的值request,session,global session,只有在web应用中使用spring时才有效
2. 对于scope为sinleton的单例模式,该Bean是在容器被创建时被装配好了
3. 对于scope为prototype的原型模式,该Bean是调用getBean时被装配的

## 3.5Bean后处理器

bean后处理器是一种特殊的bean,这个bean不需要设置id,当创建好容器时,会自动创建bena后处理器

web层

(表现层)

dao层

(数据访问层)

service层

(业务逻辑层)

作用:被其他普通的bean来调用bean后处理器的两个方法。

bean后处理器:

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.BeansException;  import org.springframework.beans.factory.config.BeanPostProcessor;  public class MyBeanPostProcessor implements BeanPostProcessor{  @Override  public Object postProcessBeforeInitialization(Object o, String s) throws BeansException {  System.out.println("执行before 方法,s:" + s);  return o;  }  public MyBeanPostProcessor(){  System.out.println("MyBeanPostPRocessor构造方法");  }  @Override  public Object postProcessAfterInitialization(Object o, String s) throws BeansException {  System.out.println("执行after 方法");  return o;  }  } |

普通的bean:

|  |
| --- |
| package lw.pers.bean;  public class MyBean {  public MyBean(){  System.out.println("MyBean的无参构造函数");  }  private String name;  private int age;  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  } |

配置:

|  |
| --- |
| <!--普通的bean-->  <bean id="bean1" class="lw.pers.bean.MyBean"></bean>  <!--bean后处理器:不写id-->  <bean class="lw.pers.bean.MyBeanPostProcessor"></bean> |

测试:

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

结果:

|  |
| --- |
| MyBeanPostPRocessor构造方法  MyBean的无参构造函数  执行before 方法,s:bean1  执行after 方法 |

## 3.6Bean后处理器应用

某个bean里有一个方法,我想增强这个方法,所以用Bean后处理器进行增强.由于是增强方法,则使用代理设计模式,因为代理设计模式的函数newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) 的第二个参数表示生成哪个对象的代理,这个对象以接口表示,所以bean必须要实现接口.

例子,增强某个bena的某个方法.

接口:

|  |
| --- |
| package lw.pers.bean;  public interface Bean {  String doSome();  String other();  } |

普通bean有两个：

|  |
| --- |
| package lw.pers.bean;  public class MyBean implements Bean{  public MyBean(){  System.out.println("MyBean的无参构造函数");  }  private String name;  private int age;  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  public String doSome(){  System.out.println("doSome()");  return "hello";  }  @Override  public String other() {  return "world";  }  } |

|  |
| --- |
| package lw.pers.bean;  public class School implements Bean{  private String name;  public void setName(String name) {  this.name = name;  }  public School(){  System.out.println("School无参构造函数");  }  @Override  public String other() {  return "world";  }  @Override  public String doSome() {  return "hello";  }  } |

bean后处理器:

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.BeansException;  import org.springframework.beans.factory.config.BeanPostProcessor;  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  import java.lang.reflect.Proxy;  public class MyBeanPostProcessor implements BeanPostProcessor{  @Override  public Object postProcessBeforeInitialization(Object o, String s) throws BeansException {  System.out.println("执行before 方法,s:" + s);  return o;  }  public MyBeanPostProcessor(){  System.out.println("MyBeanPostPRocessor构造方法");  }  @Override  public Object postProcessAfterInitialization(final Object o, String s) throws BeansException {  System.out.println("执行after 方法");  //s为bean配置的id  if("bean1".equals(s)) {  Object obj = Proxy.newProxyInstance(  o.getClass().getClassLoader(),  o.getClass().getInterfaces(),  new InvocationHandler() {  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  Object invoke = method.invoke(o, args);//这里匿名内部类使用外部的变量,外部变量必须使用final,所以  if("doSome".equals(method.getName())) {  //postProcessAfterInitialization的第一个参数为要加final  return ((String) invoke).toUpperCase();  }  return invoke;  }  }  );  return obj;  }  return o;  }  } |

配置:

|  |
| --- |
| <!--普通的bean-->  <bean id="bean1" class="lw.pers.bean.MyBean"></bean>  <bean id="bean2" class="lw.pers.bean.School"></bean>  <!--bean后处理器:不写id-->  <bean class="lw.pers.bean.MyBeanPostProcessor"></bean> |

测试:

|  |
| --- |
| ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  Bean obj1 = (Bean) ac.getBean("bean1");  Bean obj2 = (Bean) ac.getBean("bean2");  System.out.println(obj1.doSome());  System.out.println(obj1.other());  System.out.println(obj2.doSome()); |

结果:

|  |
| --- |
| MyBeanPostPRocessor构造方法  MyBean的无参构造函数  执行before 方法,s:bean1  执行after 方法  School无参构造函数  执行before 方法,s:bean2  执行after 方法  doSome()  HELLO  world  hello |

## 3.7定制Bean的生命始末

bean:

|  |
| --- |
| package lw.pers.bean;  public class MyBean {  public MyBean(){  System.out.println("MyBean的无参构造函数");  }  private String name;  private int age;  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  public String doSome(){  System.out.println("doSome()");  return "hello";  }  public void start(){  System.out.println("初始化完毕,正在调用start函数");  }  public void end(){  System.out.println("对象将要销毁,正在调用end函数");  }  } |

配置:

|  |
| --- |
| <bean id="bean1" class="lw.pers.bean.MyBean" init-method="start" destroy-method="end"></bean> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  MyBean obj1 = (MyBean) ac.getBean("bean1");  obj1.doSome();  ac.close(); |

结果:

|  |
| --- |
| MyBeanPostPRocessor构造方法  MyBean的无参构造函数  执行before 方法,s:bean1  初始化完毕,正在调用start函数  执行after 方法  doSome()  对象将要销毁,正在调用end函数 |

## 3.8Bean的生命周期

step1:调用无参构造函数,创建bean对象

step2:调用参数的setter,为属性注入值

step3:若Bean实现了BeanNameAware接口,则执行接口方法:setBeanName(String beanId),beanId为bean的id

step4:若Bean实现了BeanFactoryAware接口,则执行接口方法:setBeanFactory(BeanFactory factory),使用Bean类可以获取到BeanFactory对象

step5:定义了Bean后处理器,则执行接口方法postProcessBeforeInitialization

step6:若Bean实现了InitializingBean接口,则执行接口方法:afterPropertiesSet()

step7:若设置了init-method方法,则执行

step8:定义了Bean后处理器,则执行接口方法: postProcessAfterInitialization

step9:执行业务方法

step10:若Bean实现了DisposableBean接口,则执行接口方法,destory()

step11:若设置了destory-method方法,则执行

Bean后处理器:

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.BeansException;  import org.springframework.beans.factory.config.BeanPostProcessor;  public class MyBeanPostProcessor implements BeanPostProcessor{  @Override  public Object postProcessBeforeInitialization(Object o, String s) throws BeansException {  System.out.println("step5:执行before 方法,s:" + s);  return o;  }  public MyBeanPostProcessor(){  System.out.println("MyBeanPostPRocessor构造方法");  }  @Override  public Object postProcessAfterInitialization(final Object o, String s) throws BeansException {  System.out.println("step8:执行after 方法");  return o;  }  } |

Bean:

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.BeansException;  import org.springframework.beans.factory.\*;  public class MyBean implements BeanNameAware,BeanFactoryAware,InitializingBean,DisposableBean{  public MyBean(){  System.out.println("step1:MyBean的无参构造函数");  }  private String name;  private int age;  public void setName(String name) {  this.name = name;  System.out.println("step2:setter");  }  public void setAge(int age) {  this.age = age;  System.out.println("step2:setter");  }  public String doSome(){  System.out.println("step9:doSome()");  return "hello";  }  public void start(){  System.out.println("step7:初始化完毕后,正在调用start函数");  }  public void end(){  System.out.println("step11:对象将要销毁,正在调用end函数");  }  @Override  public void setBeanName(String s) {  System.out.println("step3:获取到bean的id:" + s);  }  @Override  public void setBeanFactory(BeanFactory beanFactory) throws BeansException {  System.out.println("step4:获取到BeanFactory容器");  }  @Override  public void afterPropertiesSet() throws Exception {  //这个方法执行后表示初始化执行完毕了  System.out.println("step6:Bean已经初始化完毕");  }  @Override  public void destroy() throws Exception {  System.out.println("step10:实现接口,销毁之前执行此方法");  }  } |

配置:

|  |
| --- |
| <!--普通的bean-->  <bean id="bean1" class="lw.pers.bean.MyBean" init-method="start" destroy-method="end">  <property name="name" value="小明"></property>  <property name="age" value="27"></property>  </bean>  <!--bean后处理器:不写id-->  <bean class="lw.pers.bean.MyBeanPostProcessor"></bean> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  MyBean obj1 = (MyBean) ac.getBean("bean1");  obj1.doSome();  ac.close(); |

# 4.基于xml的DI

## 4.1注入分类

有三种注入方式:设置注入,构造注入,实现特定接口的注入(采用的是侵入式编程,不用)

**设置注入用的多**

**构造注入用的少**

### 4.1.1设置注入

bean:

|  |
| --- |
| package lw.pers.bean;  public class School{  private String name;  public void setName(String name) {  this.name = name;  }  public School(){  System.out.println("School无参构造函数");  }  @Override  public String toString() {  return "School{" +  "name='" + name + '\'' +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  public class Student {  private String name;  private int age;  private School school;  public void setSchool(School school) {  this.school = school;  }  @Override  public String toString() {  return "Student{" +  "name='" + name + '\'' +  ", age=" + age +  ", school=" + school +  '}';  }  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  } |

配置:

|  |
| --- |
| <bean id="student" class="lw.pers.bean.Student">  <property name="age" value="27"></property>  <property name="name" value="lw"></property>  <property name="school" ref="school"></property>  </bean>  <bean id="school" class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean> |

### 4.1.2构造注入

bean:

|  |
| --- |
| package lw.pers.bean;  public class Student {  private String name;  private int age;  private School school;  public void setSchool(School school) {  this.school = school;  }  public Student(){  System.out.println("student无参构造");  }  public Student(String name, int age, School school) {  this.name = name;  this.age = age;  this.school = school;  System.out.println("student有参构造");  }  @Override  public String toString() {  return "Student{" +  "name='" + name + '\'' +  ", age=" + age +  ", school=" + school +  '}';  }  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  } |

|  |
| --- |
| package lw.pers.bean;  public class School{  private String name;  public void setName(String name) {  this.name = name;  }  public School(){  System.out.println("School无参构造函数");  }  @Override  public String toString() {  return "School{" +  "name='" + name + '\'' +  '}';  }  } |

配置:

|  |
| --- |
| <bean id="student" class="lw.pers.bean.Student">  <constructor-arg index="0" value="lw"></constructor-arg>  <constructor-arg index="1" value="27"></constructor-arg>  <constructor-arg index="2" ref="school"></constructor-arg>  </bean>  <bean id="school" class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean> |

### 4.1.3集合属性赋值

|  |
| --- |
| package lw.pers.bean;  public class School{  private String name;  public void setName(String name) {  this.name = name;  }  public School(){  System.out.println("School无参构造函数");  }  @Override  public String toString() {  return "School{" +  "name='" + name + '\'' +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  import java.util.\*;  public class Student {  private School[] schools;  private String[] mystrs;  private List<String> myList;  private Set<String> mySet;  private Map<String,Object> myMap;  private Properties myPros;  public void setSchools(School[] schools) {  this.schools = schools;  }  public void setMystrs(String[] mystrs) {  this.mystrs = mystrs;  }  public void setMyList(List<String> myList) {  this.myList = myList;  }  public void setMySet(Set<String> mySet) {  this.mySet = mySet;  }  public void setMyMap(Map<String, Object> myMap) {  this.myMap = myMap;  }  public void setMyPros(Properties myPros) {  this.myPros = myPros;  }  @Override  public String toString() {  return "Student{" +  "schools=" + Arrays.toString(schools) +  ", mystrs=" + Arrays.toString(mystrs) +  ", myList=" + myList +  ", mySet=" + mySet +  ", myMap=" + myMap +  ", myPros=" + myPros +  '}';  }  } |

配置:

|  |
| --- |
| <bean id="student" class="lw.pers.bean.Student">  <property name="schools">  <array>  <ref bean="school1"></ref>  <ref bean="school2"></ref>  </array>  </property>  <property name="mystrs">  <array>  <value>中国</value>  <value>北京</value>  </array>  </property>  <property name="myList">  <list>  <value>大兴</value>  <value>亦庄</value>  </list>  </property>  <property name="mySet">  <set>  <value>大族企业湾</value>  <value>10号楼</value>  </set>  </property>  <property name="myMap">  <map>  <entry key="mobile" value="1234567"></entry>  <entry key="QQ" value="7654321"></entry>  </map>  </property>  <property name="myPros">  <props>  <prop key="education">大学</prop>  <prop key="gender">男</prop>  </props>  </property>  </bean>  <bean id="school1" class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean>  <bean id="school2" class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean> |

**配置的简写方式:**

|  |
| --- |
| <bean id="student" class="lw.pers.bean.Student">  <property name="schools">  <array>  <ref bean="school1"></ref>  <ref bean="school2"></ref>  </array>  </property>  <property name="mystrs" value="中国,北京"></property>  <property name="myList" value="大兴,亦庄"></property>  <property name="mySet" value="大族企业湾,10号楼"></property>  <property name="myMap">  <map>  <entry key="mobile" value="1234567"></entry>  <entry key="QQ" value="7654321"></entry>  </map>  </property>  <property name="myPros">  <props>  <prop key="education">大学</prop>  <prop key="gender">男</prop>  </props>  </property>  </bean>  <bean id="school1" class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean>  <bean id="school2" class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean> |

### 4.1.4自动注入-byName

|  |
| --- |
| package lw.pers.bean;  public class Student {  private String name;  private int age;  private School school; //这里的变量名称和bean School的id一样.  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  public void setSchool(School school) {  this.school = school;  }  @Override  public String toString() {  return "Student{" +  "name='" + name + '\'' +  ", age=" + age +  ", school=" + school +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  public class School{  private String name;  public void setName(String name) {  this.name = name;  }  public School(){  System.out.println("School无参构造函数");  }  @Override  public String toString() {  return "School{" +  "name='" + name + '\'' +  '}';  }  } |

配置:

|  |
| --- |
| <bean id="student" class="lw.pers.bean.Student" **autowire="byName">**  <property name="name" value="lw"></property>  <property name="age" value="27"></property>  </bean>  <bean id="school" class="lw.pers.bean.School" >  <property name="name" value="清华大学"></property>  </bean> |

### 4.1.5自动注入-byType

接口:

|  |
| --- |
| package lw.pers.bean;  public interface Animal {  String getName();  } |

实现:

|  |
| --- |
| package lw.pers.bean;  public class Cat implements Animal {  private String name;  @Override  public String getName() {  return this.name;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "Cat{" +  "name='" + name + '\'' +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  public class Dog implements Animal {  private String name;  @Override  public String getName() {  return this.name;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "Dog{" +  "name='" + name + '\'' +  '}';  }  } |

AnimalManager有的属性是Animal类型的.

|  |
| --- |
| package lw.pers.bean;  public class AnimalManager {  private String name;  private Animal animal;  public void setName(String name) {  this.name = name;  }  public void setAnimal(Animal animal) {  this.animal = animal;  }  @Override  public String toString() {  return "AnimalManager{" +  "name='" + name + '\'' +  ", animal=" + animal +  '}';  }  } |

配置:

|  |
| --- |
| <bean id="manager1" class="lw.pers.bean.AnimalManager" **autowire="byType"**>  <property name="name" value="管理员1"></property>  </bean>  <!—注意由于属性是Animal且是个接口,则这里只能为实现类,并且只能有一种,当然如果是继承关系,则还可以有Animal类型-->  <bean id="cat" class="lw.pers.bean.Cat">  <property name="name" value="猫1"></property>  </bean>  <!--<bean id="dog" class="lw.pers.bean.Dog">-->  <!--<property name="name" value="狗1"></property>-->  <!--</bean>--> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  AnimalManager manager1 = (AnimalManager) ac.getBean("manager1");  System.out.println(manager1); |

### 4.1.6内部bean

即内部bean不想让人使用,所以放在内部.

配置:

|  |
| --- |
| <bean id="student" class="lw.pers.bean.Student">  <property name="name" value="lw"></property>  <property name="age" value="27"></property>  <property name="school">  <bean class="lw.pers.bean.School">  <property name="name" value="清华大学"></property>  </bean>  </property>  </bean> |

### 4.1.7同类抽象Bean

当若干Bean实例同属于一个类,且这写实例的值又有相同的值时.

bean:

|  |
| --- |
| package lw.pers.bean;  public class MyBean{  private String name;  @Override  public String toString() {  return "MyBean{" +  "name='" + name + '\'' +  ", age=" + age +  '}';  }  public void setAge(int age) {  this.age = age;  }  private int age;  public void setName(String name) {  this.name = name;  }  } |

配置:

|  |
| --- |
| <!--如果是abstract=ture,则通过getBean获取不到-->  <bean id="baseBean" class="lw.pers.bean.MyBean" abstract="true">  <property name="name" value="hello,world"></property>  </bean>  <bean id="bean1" parent="baseBean">  <property name="age" value="20"></property>  </bean>  <bean id="bean2" parent="baseBean">  <property name="age" value="22"></property>  </bean> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  MyBean b1 = (MyBean) ac.getBean("bean1");  MyBean b2 = (MyBean) ac.getBean("bean2");  System.out.println(b1);  System.out.println(b2); |

### 4.1.8异类抽象Bean

当若干不同的类对象具有相同的属性,且其值也相同时.

bean:

|  |
| --- |
| package lw.pers.bean;  public class Cat {  private int age;  private String name;  public void setAge(int age) {  this.age = age;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "Cat{" +  "age=" + age +  ", name='" + name + '\'' +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  public class Dog{  private String name;  @Override  public String toString() {  return "Dog{" +  "name='" + name + '\'' +  ", age=" + age +  '}';  }  public void setAge(int age) {  this.age = age;  }  private int age;  public void setName(String name) {  this.name = name;  }  } |

配置:

|  |
| --- |
| <bean id="myBase" abstract="true">  <property name="name" value="hello,world"></property>  </bean>  <bean id="cat" class="lw.pers.bean.Cat" parent="myBase">  <property name="age" value="20"></property>  </bean>  <bean id="dog" class="lw.pers.bean.Dog" parent="myBase">  <property name="age" value="22"></property>  </bean> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  Cat cat = (Cat) ac.getBean("cat");  Dog dog = (Dog) ac.getBean("dog");  System.out.println(cat);  System.out.println(dog); |

## 4.2配置文件的分配

### 4.2.1平等关系

比如有spring-1.xml 和spring-2.xml两个文件,

可以这么写

ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("spring-1.xml","spring-2.xml");

还可以String[] a = {“spring-1.xml”,”spring-2”};

在把数组a传到ClassPathXmlApplicationContext的构造函数中.

还可以这么写:

ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("spring-\*.xml");

### 4.2.2包含关系

有两个配置文件spring-1.xml spring-2.xml

spring-1.xml包含spring-2.xml则在spring-1.xml中这么写:

<import resource="classpath:spring-1.xml"></import>

## 4.3基于注解的DI

|  |
| --- |
| <?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.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  <context:component-scan base-package="lw.pers.bean"></context:component-scan>  </beans> |

base-package=”lw.pers” 是扫描lw.pers这个包及其子包

base-package=”lw.pers.\*” 是扫描这个包的子包

用@Value进行基本属性的注解,类中不必含有setter方法

@Autowired用来进行对象属性的注解的

例子:

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.stereotype.Component;  //组件,表示当前类被spring容器所管理  @Component("bean1")  public class MyBean{  @Value("hellow,world")  private String name;  @Override  public String toString() {  return "MyBean{" +  "name='" + name + '\'' +  ", age=" + age +  '}';  }  public void setAge(int age) {  this.age = age;  }  private int age;  public void setName(String name) {  this.name = name;  }  } |

### 4.3.1各种注解类

与@Component注解功能相同,但是意义不同的注解还有三个:

1)@Repository:注解在Dao实现类上

2)@Service:注解在Service实现类上

3)@Controller:注解在SrpingMVC的处理器上

@Scope(“prototype”):指定scope属性

### 4.3.2自动注入-byType

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.stereotype.Component;  @Component("school")  public class School{  @Value("清华大学")  private String name;  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "School{" +  "name='" + name + '\'' +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.stereotype.Component;  @Component("student")  public class Student {  @Value("lw")  private String name;  @Value("27")  private int age;  @Autowired //也可以用@Resource代替  private School mySchool;  public void setMySchool(School mySchool) {  this.mySchool = mySchool;  }  public void setName(String name) {  this.name = name;  }  public void setAge(int age) {  this.age = age;  }  @Override  public String toString() {  return "Student{" +  "name='" + name + '\'' +  ", age=" + age +  ", mySchool=" + mySchool +  '}';  }  } |

### 4.3.3自动注入-byName

和4.3.2差不多

|  |
| --- |
| @Autowired  @Qualifier("school") //当然也可以用@Resource(name=”school”)代替  private School school; |

注意:这两个注解要联合一起写

### 4.3.4bean的生命始末

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.stereotype.Component;  import javax.annotation.PostConstruct;  import javax.annotation.PreDestroy;  //组件,表示当前类被spring容器所管理  @Component("bean1")  public class MyBean{  @Value("hello,world")  private String name;  @Override  public String toString() {  return "MyBean{" +  "name='" + name + '\'' +  '}';  }  @PostConstruct  public void initAfter(){  System.out.println("当前bean初始化完毕....");  }  @PreDestroy  public void preDestroy(){  System.out.println("当前bean将被销毁....");  }  } |

### 4.3.5自定义spring容器

|  |
| --- |
| package lw.pers.bean;  public class Student {  private String name;  private int age;  private School school;  //这里一定要写setter方法,不知道为什么  public void setSchool(School school) {  this.school = school;  }  public Student(String name, int age) {  this.name = name;  this.age = age;  }  @Override  public String toString() {  return "Student{" +  "name='" + name + '\'' +  ", age=" + age +  ", school=" + school +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  public class School{  private String name;  public School(String name) {  this.name = name;  }  public School() {  }  @Override  public String toString() {  return "School{" +  "name='" + name + '\'' +  '}';  }  } |

|  |
| --- |
| package lw.pers.bean;  import org.springframework.beans.factory.annotation.Autowire;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  @Configuration //表示当前类充当spring容器  public class MyJavaConfig {  @Bean(name="school") //如果下面Autowire是by\_name则这里的name的值和Student的属性的对象属性要一致  public School mySchoolCreator(){  return new School("北京大学");  }  // @Bean(name="myStudent",autowire = Autowire.BY\_TYPE)  @Bean(name="myStudent",autowire = Autowire.BY\_NAME)  public Student myStudentCreator(){  return new Student("lw",27);  }  } |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  Object mystudent = ac.getBean("myStudent");  System.out.println(mystudent); |

### 4.3.6使用spring junit4测试

依赖:

|  |
| --- |
| <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>4.3.16.RELEASE</version>  </dependency> |

|  |
| --- |
| package lw.pers.service;  import lw.pers.bean.MyBean;  import lw.pers.bean.MyBean2;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.test.context.ContextConfiguration;  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations = "classpath:applicationContext.xml")  public class MyTest {  @Autowired  private MyBean bean1 ;  @Autowired  private MyBean2 bean2;  @Test  public void test01(){  System.out.println(bean1);  }  @Test  public void test02(){  System.out.println(bean2);  }  } |

# 5.Aop

Aop底层采用动态代理模式,有两种代理:JDK的动态代理,和CGLIB的动态代理

## 5.1Aop术语

1. 切面(Aspect)

切面指的是交叉业务逻辑,如日志处理等,就是对主业务逻辑增强的逻辑

1. 织如(Weaving)

将切面代码插入到目标对象的过程

1. 连接点(JoinPoint)

指的是可以被切面织如的方法,通常业务接口中的方法均为连接点

1. 切入点(Pointcut)

指的是具体织如的方法

1. 目标对象(Target)

指的是将要被增强的对象

1. 通知(Advice)

是切面的一种实现,可以完成简单的织入功能,无法完成将切面织入到目标的指定方法中,而是所有的方法都会被增强.

通知增强代码到目标代码的时间点

1. 顾问(Advisor)

顾问是切面的另一种实现,能够将通知以更为复杂方式织入到目标对象中

包装为更为复杂的切面装配器

## 5.2Aop编程环境

需要引入两个包:spring-aop和aopalliance-sources两个jar包

## 5.3通知

### 5.3.1前置通知

|  |
| --- |
| package lw.pers.service;  public interface ISomeService {  void doFirst();  void doSecond();  } |

|  |
| --- |
| package lw.pers.service;  public class ISomeServiceImpl implements ISomeService {  @Override  public void doFirst() {  System.out.println("doFirst");  }  @Override  public void doSecond() {  System.out.println("doSecond");  }  } |

|  |
| --- |
| package lw.pers.advice;  import org.springframework.aop.MethodBeforeAdvice;  import java.lang.reflect.Method;  public class MyMethodBeforeAdvice implements MethodBeforeAdvice{  @Override  public void before(Method method, Object[] objects, Object o) throws Throwable {  //当前方法在目标方法执行之前执行  //method:目标方法  //args:目标方法的参数列表  //target:目标对象  System.out.println("执行前置通知方法");  }  } |

配置:

|  |
| --- |
| <?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.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  <context:component-scan base-package="lw.pers.bean"></context:component-scan>  <context:component-scan base-package="lw.pers.service"></context:component-scan>  <context:component-scan base-package="lw.pers.advice"></context:component-scan>  <!--目标bean-->  <bean id="someService" class="lw.pers.service.ISomeServiceImpl"></bean>  <!--切面-->  <bean id="myAdvice" class="lw.pers.advice.MyMethodBeforeAdvice"></bean>  <!--代理-->  <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <!--<property name="target" ref="someService"></property>-->  <property name="targetName" value="someService"></property>  <!--指定切面-->  <property name="interceptorNames" value="myAdvice"></property>  </bean>  </beans> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  ISomeService bean1 = (ISomeService)ac.getBean("serviceProxy");  bean1.doFirst(); |

### 5.3.2后置通知

和前置通知差不多,只是实现的接口不一样.

|  |
| --- |
| package lw.pers.advice;  import org.springframework.aop.AfterReturningAdvice;  import java.lang.reflect.Method;  public class MyAfterReturningAdvice implements AfterReturningAdvice{  //在目标方法之后执行  //returnValue:目标方法的返回值,改变这个值并不能影响到目标返回结果  @Override  public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {  System.out.println("后置方法");  }  } |

### 5.3.3环绕通知

在目标方法之前和之后执行,返回值会取代目标方法的返回值

|  |
| --- |
| public class MyMethodInterceptor implements MethodInterceptor{  @Override  public Object invoke(MethodInvocation invocation) throws Throwable {  System.out.println("执行环绕通知:目标方法执行之前");  //执行目标方法  Object result = invocation.proceed();  System.out.println("执行环绕通知:目标方法执行之后");  if(result!=null){  result = ((String)result).toUpperCase();  }  return result;  }  } |

### 5.3.4异常通知

需要实现ThrowsAdvice接口,作用是在目标方法抛出异常后,根据不同的异常做出不同的处理,也就是在这个通知里会定义很多方法,参数类型是目标方法抛出的异常类型.

|  |
| --- |
|  |

该接口虽然不需要实现任何接口,但是文档中确实是要实现这几个接口,总共4个.

|  |
| --- |
| public class ISomeServiceImpl implements ISomeService {  @Override  public void doFirst() {  System.out.println("doFirst" + 1/0);  }  @Override  public String doSecond() {  return "acbd";  }  } |

|  |
| --- |
| package lw.pers.advice;  import org.springframework.aop.ThrowsAdvice;  public class MyThrowAdvice implements ThrowsAdvice{  public void afterThrowing(Exception ex){  System.out.println("执行异常通知方法");  }  } |

### 5.3.5多个通知

|  |
| --- |
| <!--目标bean-->  <bean id="someService" class="lw.pers.service.ISomeServiceImpl"></bean>  <!--切面-->  <bean id="myBeforeAdvice" class="lw.pers.advice.MyMethodBeforeAdvice"></bean>  <bean id="myAfterAdvice" class="lw.pers.advice.MyAfterReturningAdvice"></bean>  <!--代理-->  <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <!--<property name="target" ref="someService"></property>-->  <property name="targetName" value="someService"></property>  <!--指定切面-->  <property name="interceptorNames">  <array>  <value>myBeforeAdvice</value>  <value>myAfterAdvice</value>  </array>  </property>  </bean> |

简单的写法:

|  |
| --- |
| <property name="interceptorNames" value="myAfterAdvice,myBeforeAdvice"></property> |

## 5.4顾问

### 5.4.1名称匹配方法切入点顾问

业务接口:

|  |
| --- |
| package lw.pers.service;  public interface ISomeService {  void doFirst();  String doSecond();  void doThird();  } |

业务实现类:

|  |
| --- |
| package lw.pers.service;  public class ISomeServiceImpl implements ISomeService {  @Override  public void doFirst() {  System.out.println("doFirst");  }  @Override  public String doSecond() {  return "acbd";  }  @Override  public void doThird() {  System.out.println("doThird");  }  } |

通知1：

|  |
| --- |
| package lw.pers.advice;  import org.springframework.aop.MethodBeforeAdvice;  import java.lang.reflect.Method;  public class MyMethodBeforeAdvice implements MethodBeforeAdvice{  @Override  public void before(Method method, Object[] objects, Object o) throws Throwable {  System.out.println("执行前置通知方法");  }  } |

通知2:

|  |
| --- |
| package lw.pers.advice;  import org.springframework.aop.AfterReturningAdvice;  import java.lang.reflect.Method;  public class MyAfterReturningAdvice implements AfterReturningAdvice{  //在目标方法之后执行  //returnValue:目标方法的返回值,改变这个值并不能影响到目标返回结果  @Override  public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {  System.out.println("后置方法");  }  } |

配置:

|  |
| --- |
| <?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.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  <context:component-scan base-package="lw.pers.bean"></context:component-scan>  <context:component-scan base-package="lw.pers.service"></context:component-scan>  <context:component-scan base-package="lw.pers.advice"></context:component-scan>  <!--目标bean-->  <bean id="someService" class="lw.pers.service.ISomeServiceImpl"></bean>  <!--切面:通知-->  <bean id="myBeforeAdvice" class="lw.pers.advice.MyMethodBeforeAdvice"></bean>  <bean id="myAfterAdvice" class="lw.pers.advice.MyAfterReturningAdvice"></bean>  <!--切面:顾问-->  <bean id="myAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  <!--指定通知-->  <property name="advice" ref="myAfterAdvice"></property>  <!--指定切入点:即指定方法-->  <!--<property name="mappedNames" value="doFirst,doThird"></property>-->  <property name="mappedNames" value="\*ir\*"></property>  </bean>  <!--代理-->  <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <!--<property name="target" ref="someService"></property>-->  <property name="targetName" value="someService"></property>  <!--指定切面-->  <property name="interceptorNames" value="myAdvisor"></property>  </bean>  </beans> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  ISomeService bean1 = (ISomeService)ac.getBean("serviceProxy");  bean1.doFirst();  // bean1.doSecond();  bean1.doThird(); |

### 5.4.2正则表达式切入点顾问

|  |
| --- |
| <!--切面:顾问-->  <bean id="myAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  <!--指定通知-->  <property name="advice" ref="myAfterAdvice"></property>  <!--指定切入点:即指定方法-->  <property name="mappedNames" value="doFirst,doThird"></property>  </bean>  <bean id="myAdvisor2" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  <!--指定通知-->  <property name="advice" ref="myAfterAdvice"></property>  <!--指定切入点:即指定方法-->  <!--全限定方法名称-->  <!--<property name="pattern" value="lw.pers.service.ISomeServiceImpl.doFirst"></property>-->  <!--<property name="patterns" value=".\*doFirst,.\*doThird"></property>-->  <property name="pattern" value=".\*doFirst|.\*doThird"></property>  </bean>  <!--代理-->  <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <!--<property name="target" ref="someService"></property>-->  <property name="targetName" value="someService"></property>  <!--指定切面-->  <property name="interceptorNames" value="myAdvisor2"></property>  </bean> |

## 5.5自动代理生成器

### 5.5.1默认advisor自动代理生成器

把代理配置：

|  |
| --- |
| <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <property name="target" ref="someService"></property>  <property name="targetName" value="someService"></property>  <!--指定切面-->  <property name="interceptorNames" value="myAdvisor2"></property>  </bean> |

替换成:

|  |
| --- |
| <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></bean> |

它会根据配置的顾问进行自动生成代理,即有多少个顾问就会生成多少个代理.

测试的时候:getBean参数为业务Bean的id而不是以前的代理的id

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");  ISomeService bean1 = (ISomeService)ac.getBean("someService");  bean1.doFirst(); |

DefaultAdvisorAutoProxyCreator的缺点:

1. 不能选择目标对象
2. 不能选择切面类型,只能是advisor
3. 不能选择advisor,所以advisor均将被作为切面织入到目标方法

为了解决这三个问题,可以使用Bean名称自动代理生成器

### 5.5.2 Bean名称自动代理生成器

|  |
| --- |
| <?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.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  <context:component-scan base-package="lw.pers.bean"></context:component-scan>  <context:component-scan base-package="lw.pers.service"></context:component-scan>  <context:component-scan base-package="lw.pers.advice"></context:component-scan>  <!--目标bean-->  <bean id="someService" class="lw.pers.service.ISomeServiceImpl"></bean>  <!--切面:通知-->  <bean id="myBeforeAdvice" class="lw.pers.advice.MyMethodBeforeAdvice"></bean>  <bean id="myAfterAdvice" class="lw.pers.advice.MyAfterReturningAdvice"></bean>  <!--切面:顾问-->  <bean id="myAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  <!--指定通知-->  <property name="advice" ref="myAfterAdvice"></property>  <!--指定切入点:即指定方法-->  <property name="mappedNames" value="doFirst,doThird"></property>  </bean>  <bean id="myAdvisor2" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  <!--指定通知-->  <property name="advice" ref="myAfterAdvice"></property>  <!--<property name="pattern" value="lw.pers.service.ISomeServiceImpl.doFirst"></property>-->  <!--<property name="patterns" value=".\*doFirst,.\*doThird"></property>-->  <property name="pattern" value=".\*doFirst|.\*doThird"></property>  </bean>  <!--默认自动代理生成器-->  <!--<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></bean>-->  <!--bean名称自动代理生成器-->  <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">  <!--指定目标对象-->  <property name="beanNames" value="someService"></property>  <!--指定切面-->  <property name="interceptorNames" value="myAdvisor2"></property>  </bean>  </beans> |

# 6.AspectJ对AOP的实现

对于AOP这种编程思想,很多框架都实现了,Spring就是其中之一,可以完成面向切面的编程,

然而,AspectJ也实现了AOP的功能,其实现方式更为简洁,而且支持注解式开发,所以Spring将AspectJ的对于AOP的实现也引入到了自己的框架中

配置环境:

1. 需要Aop的配置环境(两个jar包)
2. aspectj相关的两个包

|  |
| --- |
| <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.6.8</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring-version}</version>  </dependency> |

AspectJ的通知类型有5种:

1. 前置通知
2. 后置通知
3. 环绕通知
4. 异常通知
5. 最终通知

最终通知是指无论程序是否正常执行,该通知都会执行,类似try ... catch 中的finally代码块

AspectJ的切入点表达式:

execution(

[modifiers-pattern] 访问权限类型

ret-type-pattern 返回值类型

[declaring-type-pattern] 权限定类名

name-pattern(param-pattern) 方法名(参数名)

[throws-pattern] 抛出异常类型

)

表达式的红色部分不可省略

表达式符号:

“\*” :0个或者任意多个字符

“..” :用在方法参数中,表示任意多个参数,用在包名后,表示当前包及其子包路径

“+” :用在类名后,表示当前类及其子类,用在接口后,表示当前接口及其实现类

## 6.1aspectj基于注解的aop

业务接口:

|  |
| --- |
| package lw.pers.service;  public interface ISomeService {  void doFirst();  String doSecond();  void doThird();  } |

业务实现类:

|  |
| --- |
| package lw.pers.service;  public class ISomeServiceImpl implements ISomeService {  @Override  public void doFirst() {  System.out.println("doFirst");  }  @Override  public String doSecond() {  return "acbd";  }  @Override  public void doThird() {  System.out.println("doThird" + 1/0);  }  } |

切面类:

|  |
| --- |
| package lw.pers.annotation;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.\*;  @Aspect //表示当前类为切面  public class MyAspect {  @Before("execution(\* \*..ISomeService.doFirst(..))")  public void mybefore(){  System.out.println("执行前置通知方法");  }  //jp就是那个匹配到目标方法的全路径  @Before("execution(\* \*..ISomeService.doFirst(..))")  public void mybefore(JoinPoint jp){  System.out.println("执行前置通知方法 jp=" + jp);  }  @AfterReturning("execution(\* \*..ISomeService.doSecond(..))")  public void myafterReturning(){  System.out.println("执行后置通知方法");  }  @AfterReturning(value = "execution(\* \*..ISomeService.doSecond(..))",returning = "result")  public void myafterReturning(Object result){  System.out.println("执行后置通知方法 result=" + result);  }  @Around("execution(\* \*..ISomeService.doSecond(..))")  public Object myAround(ProceedingJoinPoint pjp) throws Throwable {  System.out.println("执行环绕通知方法,目标方法执行之前");  //执行目标方法  Object result = pjp.proceed();  System.out.println("执行环绕通知方法,目标方法执行之后");  if(result!=null){  result = ((String)result).toUpperCase();  }  return result;  }  //异常通知  @AfterThrowing(value = "execution(\* \*..ISomeService.doThird(..))")  public void myAfterThrowing(){  System.out.println("执行异常通知方法");  }  // @AfterThrowing(value = "execution(\* \*..ISomeService.doThird(..))",throwing = "ex")  @AfterThrowing(value = "doThirdPointcut()",throwing = "ex")  public void myAfterThrowing(Exception ex){  System.out.println("执行异常通知方法 ex=" + ex.getMessage());  }  //最终通知  // @After(value = "execution(\* \*..ISomeService.doThird(..))")  @After("doThirdPointcut()")  public void myAfter(){  System.out.println("执行最终通知");  }  //定义了一个切入点,叫doThirdPointcut()  @Pointcut("execution(\* \*..ISomeService.doThird(..))")  public void doThirdPointcut(){}  } |

配置:

|  |
| --- |
| <?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:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">  <!--注册切面-->  <bean id="myAspect" class="lw.pers.annotation.MyAspect"></bean>  <!--注册目标对象-->  <bean id="someService" class="lw.pers.service.ISomeServiceImpl"></bean>  <!--注册AspectJ的自动代理-->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>  </beans> |

测试:

|  |
| --- |
| ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("spring.xml");  ISomeService bean1 = (ISomeService)ac.getBean("someService");  bean1.doFirst();  bean1.doSecond();  bean1.doThird(); |

## 6.2aspectj基于xml的aop

接口和接口实现类都是一样的

切面类:

|  |
| --- |
| package lw.pers.xml;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.ProceedingJoinPoint;  //切面  public class MyAspect {  public void mybefore(){  System.out.println("执行前置通知方法");  }  public void mybefore(JoinPoint jp){  System.out.println("执行前置通知方法 jp=" + jp);  }  public void myafterReturning(){  System.out.println("执行后置通知方法");  }  public void myafterReturning(Object result){  System.out.println("执行后置通知方法 result=" + result);  }  public Object myAround(ProceedingJoinPoint pjp) throws Throwable {  System.out.println("执行环绕通知方法,目标方法执行之前");  //执行目标方法  Object result = pjp.proceed();  System.out.println("执行环绕通知方法,目标方法执行之后");  if(result!=null){  result = ((String)result).toUpperCase();  }  return result;  }  //异常通知  public void myAfterThrowing(){  System.out.println("执行异常通知方法");  }  public void myAfterThrowing(Exception ex){  System.out.println("执行异常通知方法 ex=" + ex.getMessage());  }  //最终通知  public void myAfter(){  System.out.println("执行最终通知");  }  } |

配置:

|  |
| --- |
| <?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:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">  <!--注册切面-->  <bean id="myAspect" class="lw.pers.xml.MyAspect"></bean>  <!--注册目标对象-->  <bean id="someService" class="lw.pers.service.ISomeServiceImpl"></bean>  <!--AOP配置-->  <aop:config>  <aop:pointcut id="doFirstPointcut" expression="execution(\* \*..ISomeService.doFirst(..))"></aop:pointcut>  <aop:pointcut id="doSecondPointcut" expression="execution(\* \*..ISomeService.doSecond(..))"></aop:pointcut>  <aop:pointcut id="doThirdPointcut" expression="execution(\* \*..ISomeService.doThird(..))"></aop:pointcut>  <aop:aspect ref="myAspect">  <!--前置通知-->  <!--<aop:before method="mybefore" pointcut="execution(\* \*..ISomeService.doFirst(..))"></aop:before>-->  <aop:before method="mybefore" pointcut-ref="doFirstPointcut"></aop:before>  <aop:before method="mybefore(org.aspectj.lang.JoinPoint)" pointcut="execution(\* \*..ISomeService.doFirst(..))"></aop:before>  <!--后置通知-->  <aop:after-returning method="myafterReturning" pointcut-ref="doSecondPointcut"></aop:after-returning>  <aop:after-returning method="myafterReturning(java.lang.Object)" returning="result" pointcut-ref="doSecondPointcut"></aop:after-returning>  <!--环绕通知-->  <aop:around method="myAround" pointcut-ref="doSecondPointcut"></aop:around>  <!--异常通知-->  <aop:after-throwing method="myAfterThrowing" pointcut-ref="doThirdPointcut"></aop:after-throwing>  <aop:after-throwing method="myAfterThrowing(java.lang.Exception)" throwing="ex" pointcut-ref="doThirdPointcut"></aop:after-throwing>  <!--最终通知-->  <aop:after method="myAfter" pointcut-ref="doThirdPointcut"></aop:after>  </aop:aspect>  </aop:config>  </beans> |

测试和6.1一样的

# 7.spring和dao

spring和dao部分,是spring的两个核心技术LOC和AOP的典型应用

对于JDBC模板的使用,似乎LOC的应用,是将JDBC模板对象注入给了dao层的实现类

对于spring的事务管理,是AOP的应用,将事务作为切面织入到了Service层的业务方法中

需要的jar包:

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring-version}</version>  </dependency>  <dependency>  <groupId>com.mchange</groupId>  <artifactId>c3p0</artifactId>  <version>0.9.5.2</version>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-dbcp2</artifactId>  <version>2.1.1</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.44</version>  </dependency> |

## 7.1spring与JDBC模板

实体类:

|  |
| --- |
| package lw.pers.entity;  public class Student {  private Integer id;  private String name;  private int age;  public Student(){  }  public Student(String name, int age) {  this.name = name;  this.age = age;  }  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 int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  @Override  public String toString() {  return "Student{" +  "id=" + id +  ", name='" + name + '\'' +  ", age=" + age +  '}';  }  } |

RowMapper的实现类

|  |
| --- |
| package lw.pers.entity;  import org.springframework.jdbc.core.RowMapper;  import java.sql.ResultSet;  import java.sql.SQLException;  public class StudentRowMapper implements RowMapper<Student> {  @Override  //这里的rs:不是查询出来的总的结果集，而是代表一行数据  public Student mapRow(ResultSet rs, int rowNum) throws SQLException {  Student student = new Student();  student.setId(rs.getInt("id"));  student.setName(rs.getString("name"));  student.setAge(rs.getInt("age"));;  return student;  }  } |

dao层:

dao接口

|  |
| --- |
| package lw.pers.dao;  import lw.pers.entity.Student;  import java.util.List;  public interface IStudentDao {  void insertStudent(Student student);  void deleteById(int id);  void updateStudent(Student student);  List<String> selectAllStudentsNames();  String selectStudentNameById(int id);  List<Student> selectAllStudents();  Student selectStudentById(int id);  } |

dao实现类:

|  |
| --- |
| package lw.pers.dao;  import lw.pers.entity.Student;  import lw.pers.entity.StudentRowMapper;  import org.springframework.jdbc.core.support.JdbcDaoSupport;  import java.util.List;  //注意:JDBC模板对象是多例的,每次使用模板时都需要通过this. getJdbcTemplate()获取  public class IStudentDaoImpl extends JdbcDaoSupport implements IStudentDao {  @Override  public void insertStudent(Student student) {  String sql = "insert into student(name,age) values(?,?)";  this.getJdbcTemplate().update(sql,student.getName(),student.getAge());  }  @Override  public void deleteById(int id) {  String sql = "delete from student where id=?";  this.getJdbcTemplate().update(sql,id);  }  @Override  public void updateStudent(Student student) {  String sql = "update student set name=?,age=? where id=?";  this.getJdbcTemplate().update(sql,student.getName(),student.getAge(),student.getId());  }  @Override  public List<String> selectAllStudentsNames() {  String sql = "SELECT name FROM student";  //queryForList的第二个参数为返回的单个元素的类型  return this.getJdbcTemplate().queryForList(sql,String.class);  }  @Override  public String selectStudentNameById(int id) {  String sql = "select name from student where id=?";  return this.getJdbcTemplate().queryForObject(sql,String.class,id);  }  @Override  public List<Student> selectAllStudents() {  String sql = "select id,name,age from student";  return this.getJdbcTemplate().query(sql,new StudentRowMapper());  }  @Override  public Student selectStudentById(int id) {  String sql = "select id,name,age from student where id=?";  return this.getJdbcTemplate().queryForObject(sql,new StudentRowMapper(),1);  }  } |

Service层:

service接口:

|  |
| --- |
| package lw.pers.service;  import lw.pers.entity.Student;  import java.util.List;  public interface IStudentService {  void addStudent(Student student);  void removeById(int id);  void modifyStudent(Student student);  List<String> findAllStudentsNames();  String findStudentNameById(int id);  List<Student> findAllStudents();  Student findStudentById(int id);  } |

service实现类:

|  |
| --- |
| package lw.pers.service;  import lw.pers.dao.IStudentDao;  import lw.pers.entity.Student;  import java.util.List;  public class IStudentServiceImpl implements IStudentService {  private IStudentDao dao;  public void setDao(IStudentDao dao) {  this.dao = dao;  }  @Override  public void addStudent(Student student) {  dao.insertStudent(student);  }  @Override  public void removeById(int id) {  dao.deleteById(id);  }  @Override  public void modifyStudent(Student student) {  dao.updateStudent(student);  }  @Override  public List<String> findAllStudentsNames() {  return dao.selectAllStudentsNames();  }  @Override  public String findStudentNameById(int id) {  return dao.selectStudentNameById(id);  }  @Override  public List<Student> findAllStudents() {  return dao.selectAllStudents();  }  @Override  public Student findStudentById(int id) {  return dao.selectStudentById(id);  }  } |

配置:

spring.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:aop="http://www.springframework.org/schema/aop"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  <!--注册数据源-->  <!--spring默认的数据源-->  <bean id="myDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>  <property name="url" value="jdbc:mysql://47.96.27.61:3306/test"></property>  <property name="username" value="lw"></property>  <property name="password" value="linwei"></property>  </bean>  <!--注册数据源:dbcp2-->  <!--<bean id="myDataSource" class="org.apache.commons.dbcp2.BasicDataSource">-->  <!--<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>-->  <!--<property name="url" value="jdbc:mysql://47.96.27.61:3306/test"></property>-->  <!--<property name="username" value="lw"></property>-->  <!--<property name="password" value="linwei"></property>-->  <!--</bean>-->  <!--注册数据源:c3p0-->  <!--<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">-->  <!--<property name="driverClass" value="com.mysql.jdbc.Driver"></property>-->  <!--<property name="jdbcUrl" value="jdbc:mysql://47.96.27.61:3306/test"></property>-->  <!--<property name="user" value="lw"></property>-->  <!--<property name="password" value="linwei"></property>-->  <!--</bean>-->  <!--注册属性(资源)文件:方式1 要记Class的名称很麻烦,一般用方式2-->  <!--<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">-->  <!--<property name="location" value="classpath:jdbc.properties"></property>-->  <!--</bean>-->  <!--注册属性(资源)文件:方式2-->  <context:property-placeholder location="classpath:jdbc.properties"></context:property-placeholder>  <!--注册数据源:c3p0-->  <!--<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">-->  <!--<property name="driverClass" value="${jdbc.driver}"></property>-->  <!--<property name="jdbcUrl" value="${jdbc.url}"></property>-->  <!--<property name="user" value="${jdbc.user}"></property>-->  <!--<property name="password" value="${jdbc.password}"></property>-->  <!--</bean>-->  <!--注册JdbcTemplate-->  <bean id="myJdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="myDataSource"></property>  </bean>  <!--注册dao-->  <bean id="studentDao" class="lw.pers.dao.IStudentDaoImpl">  <property name="jdbcTemplate" ref="myJdbcTemplate"></property>  <!--默认模板配置,就要注册JdbcTemplate-->  <!--<property name="dataSource" ref="myDataSource"></property>-->  </bean>  <bean id="studentService" class="lw.pers.service.IStudentServiceImpl">  <property name="dao" ref="studentDao"></property>  </bean>  </beans> |

jdbc.properties:

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://47.96.27.61:3306/test  jdbc.user=lw  jdbc.password=linwei |

测试:

|  |
| --- |
| package lw.pers;  import lw.pers.entity.Student;  import lw.pers.service.IStudentService;  import org.junit.Before;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import java.util.List;  public class AppTest  {  IStudentService service;  @Before  public void before()throws Exception{  String resource = "spring.xml";  ApplicationContext ac = new ClassPathXmlApplicationContext(resource);  service = (IStudentService) ac.getBean("studentService");  }  //增  @Test  public void test\_01()throws Exception{  Student student = new Student("张三",34);  service.addStudent(student);  }  //删  @Test  public void test\_02()throws Exception{  service.removeById(2);  }  //改  @Test  public void test\_03()throws Exception{  Student student = new Student("小李",22);  student.setId(3);  service.modifyStudent(student);  }  //查询所有名字  @Test  public void test\_04()throws Exception{  List<String> names = service.findAllStudentsNames();  System.out.println(names);  }  //根据id查询姓名  @Test  public void test\_05()throws Exception{  String name = service.findStudentNameById(1);  System.out.println(name);  }  @Test  public void test\_06()throws Exception{  List<Student> students = service.findAllStudents();  System.out.println(students);  }  @Test  public void test\_07()throws Exception{  Student student = service.findStudentById(1);  System.out.println(student);  }  } |

## 7.2spring与事务管理

事务原本是数据库中的概念,应该在Dao层,但是一般情况下,需要将事务提到业务层,即Service层,目的:为了能够使用事务的特性来管理具体的业务

spring中有三种方式来实现对事务的管理:

1. 使用spring的事务代理工厂管理事务
2. 使用spring的事务注解管理事务
3. 使用AspectJ的Aop配置管理事务

# 8.POM.xml配置

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>lw.pers</groupId>  <artifactId>demo2</artifactId>  <version>1.0-SNAPSHOT</version>  <name>demo2</name>  <!-- FIXME change it to the project's website -->  <url>http://www.example.com</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <maven.compiler.source>1.7</maven.compiler.source>  <maven.compiler.target>1.7</maven.compiler.target>  <spring-version>4.3.16.RELEASE</spring-version>  </properties>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring-version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>${spring-version}</version>  </dependency>  <dependency>  <groupId>aopalliance</groupId>  <artifactId>aopalliance</artifactId>  <version>1.0</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.6.8</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring-version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring-version}</version>  </dependency>  <dependency>  <groupId>com.mchange</groupId>  <artifactId>c3p0</artifactId>  <version>0.9.5.2</version>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-dbcp2</artifactId>  <version>2.1.1</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.44</version>  </dependency>  </dependencies>  <build>  <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to parent pom) -->  <plugins>  <plugin>  <artifactId>maven-clean-plugin</artifactId>  <version>3.0.0</version>  </plugin>  <!-- see http://maven.apache.org/ref/current/maven-core/default-bindings.html#Plugin\_bindings\_for\_jar\_packaging -->  <plugin>  <artifactId>maven-resources-plugin</artifactId>  <version>3.0.2</version>  </plugin>  <plugin>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.7.0</version>  </plugin>  <plugin>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.20.1</version>  </plugin>  <plugin>  <artifactId>maven-jar-plugin</artifactId>  <version>3.0.2</version>  </plugin>  <plugin>  <artifactId>maven-install-plugin</artifactId>  <version>2.5.2</version>  </plugin>  <plugin>  <artifactId>maven-deploy-plugin</artifactId>  <version>2.8.2</version>  </plugin>  </plugins>  </pluginManagement>  </build>  </project> |

# 9.ssm整合

1. xml配置方式

请看demo-xml

1. 注解配置方式

请看demo- annotation