**Spring\_02**

1. spring表达式语言（简称SpEL）:是一个支持运行时查询和操作对象图的强大的表达式语言
2. 语法类似于EL，SpEL使用#{...}作为定界符，所有在大括号内的字符都将被认为是SpEL
3. SpEL为bean的属性进行动态赋值提供了便利。
4. 通过SpEL可以实现：

通过bean的id对bean进行引用

调用方法以及引用对象中的属性

计算表达式的值

正则表达式的匹配

范例：

**<?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:p=*"http://www.springframework.org/schema/p"***

**xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>**

**<bean id=*"address"* class=*"com.wp.spring.beans.spel.Address"*>**

**<!-- 使用spel为属性赋某一个字面量 -->**

**<property name=*"city"* value=*"#{'chengdu'}"*></property>**

**<property name=*"street"* value=*"高兴区"*></property>**

**</bean>**

**<bean id=*"car"* class=*"com.wp.spring.beans.spel.Car"*>**

**<property name=*"brand"* value=*"大众"*></property>**

**<property name=*"price"* value = *"200000"*></property>**

**<!-- 使用SpEL引用类的静态属性PI -->**

**<property name=*"tyrePerimeter"* value=*"#{T(java.lang.Math).PI \* 80}"*></property>**

**</bean>**

**<bean id=*"person"* class=*"com.wp.spring.beans.spel.Person"*>**

**<!-- 使用SpEL来引用其他的bean -->**

**<property name=*"car"* value=*"#{car}"*></property>**

**<!-- 使用SpEL来引用其他bean的属性 -->**

**<property name=*"city"* value=*"#{address.city}"*></property>**

**<!-- 使用SpEL使用三目运算符 -->**

**<property name=*"info"* value=*"#{car.price > 300000 ? '金领':'白领'}"*></property>**

**<property name=*"name"* value=*"文攀"*></property>**

**</bean>**

**</beans>**

**Spring\_管理bean的生命周期：**

SpringIOC容器可以管理bean的生命周期，Spring允许在bean生命周期的特定点执行定制的任务。

SpringIOC容器对bean的生命周期进行管理的过程：

**通过构造器或工厂方法创建bean实例**

**为bean的属性设置值和对其他bean的引用**

**调用bean的初始化方法**

**Bean可以使用了**

**当容器关闭时，调用bean的销毁方法**

**在bean的声明里面设置init-method和destroy-method属性为bean指定初始化和销毁方法**

**Bean后置处理器允许在调用初始化方法前后对bean进行额外的处理**

**Bean后置处理器对IOC容器里的所有bean实例进行处理，而非单一实例，其典型的应用是检查bean的属性的正确性或根据特定的标准更改bean的属性。**

**对bean后置处理而言，需要实现interface BeanPostProcessor接口，在初始化方法被调用前后spring将把每个bean实例分别传递给上述接口的以下两个方法：**

**加完BeanPostProcessor 之后生命周期：**

**通过构造器或工厂方法创建bean实例**

**为bean的属性设置值和对其他bean的引用**

**将bean实例传递给后置处理器的postProcessBeforeInitialization(Object bean, String beanName)方法**

**调用bean的初始化方法**

**将bean实例传递给后置处理器的postProcessAfterInitialization(Object bean, String beanName) 方法**

**Bean可以使用了**

**当容器关闭时，调用bean的销毁方法**

**在bean的声明里面设置init-method和destroy-method属性为bean指定初始化和销毁方法**

范例：

package com.wp.spring.beans.cycle;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.config.BeanPostProcessor;

public class MyBeanPostProcessor implements BeanPostProcessor {

@Override

public Object postProcessAfterInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("初始化方法之后。。。。。" + bean + "," + beanName);

Car car = new Car();

car.setBrand("福特");

return car;

}

@Override

public Object postProcessBeforeInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("初始化方法之前。。。。。" + bean + "," + beanName);

if("car".equals(beanName)) {

//.....

}

return bean;

}

}

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"car"* class=*"com.wp.spring.beans.cycle.Car"*

init-method=*"init"*

destroy-method=*"destroy"*>

<property name=*"brand"* value=*"奥迪"*></property>

</bean>

<!-- 配置bean的后置处理器 -->

**<bean class=*"com.wp.spring.beans.cycle.MyBeanPostProcessor"*>**

</bean>

<!-- 实现BeanPostProcessor接口，并具体提供

**public Object postProcessAfterInitialization(Object bean, String beanName) 在初始化bean的方法之后调用**

**public Object postProcessBeforeInitialization(Object bean, String beanName) 在初始化bean的方法之前调用**

**两个方法的实现**

**bean:bean实例本身**

**beanName:ioc容器配置的bean的名字**

**返回值：是实际上返回给用户的哪个bean， 注意：可以在以上两个方法中修改返回的bean，甚至可以返回一个新的bean**

-->

<!-- **配置bean的后置处理器：不需要配置id，IOC容器自动识别是一个BeanPostProcessor** -->

</beans>

**public** **class** Main {

**public** **static** **void** main(String[] args) {

ClassPathXmlApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans-cycle.xml");

Car car = (Car)ctx.getBean("car");

System.***out***.println(car);

//关闭IOC容器

ctx.close();

}

}

**通过工厂方法配置bean：**

范例：

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<!-- 通过静态工厂方法来配置bean，注意：不是配置工厂方法实例，而是配置bean实例 -->

<!--

class属性：执行静态工厂方法的全类名

factory-method:执行静态工厂方法的名字

constructor-arg：如果工厂方法里需要传入参数，则使用constructor-arg来配置传入参数

-->

<bean id=*"car1"*

**class=*"com.wp.spring.beans.factory.StaticCarFactory"***

**factory-method=*"getCar"*>**

<!-- 给静态工厂里面的getCar方法传递一个name参数 -->

**<constructor-arg value=*"audi"*></constructor-arg>**

</bean>

</beans>

静态工厂：

**package** com.wp.spring.beans.factory;

/\*\*

\* 静态工厂方法：直接调用某一个类的静态方法就可以返回bean的实例

\* \*/

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** StaticCarFactory {

**private** **static** Map<String, Car> *cars* = **new** HashMap<>();

**static** {

*cars*.put("audi", **new** Car("audi", 300000));

*cars*.put("dazong", **new** Car("dazong", 500000));

}

//静态工厂方法

**public** **static** Car getCar(String name) {

**return** *cars*.get(name);

}

}

**通过实例工厂方法来创建bean**：

<!-- 配置实例工厂本身 -->

<bean id=*"carFactory"* class=*"com.wp.spring.beans.factory.InstanceCarFactory"*></bean>

<!-- 通过实例工厂来配置bean -->

<bean id=*"car2"* factory-bean=*"carFactory"* factory-method=*"getCar"*>

<constructor-arg value=*"dazong"*></constructor-arg>

</bean>

实例工厂：

**package** com.wp.spring.beans.factory;

**import** java.util.HashMap;

**import** java.util.Map;

/\*\*

\* 实例工厂方法：实例工厂的方法，即需要先创建工厂本身再调用工厂的实例方法来返回bean的本身

\* \*/

**public** **class** InstanceCarFactory {

**private** Map<String, Car> cars = **null**;

**public** InstanceCarFactory() {

cars = **new** HashMap<>();

cars.put("audi", **new** Car("audi", 300000));

cars.put("dazong", **new** Car("dazong", 100000));

}

**public** Car getCar(String name) { //通过传入的参数名称，取得相应的汽车

**return** cars.get(name);

}

}

**通过FactoryBean来配置bean**：

自定义的FactoryBean：

**package** com.wp.spring.beans.factorybean;

**import** org.springframework.beans.factory.FactoryBean;

/\*\*

\* 自定义的FactoryBean,需要实现FactoryBean接口

\* \*/

**public** **class** CarFactoryBean **implements** FactoryBean<Car> {

**private** String brand;

**public** **void** setBrand(String brand) {

**this**.brand = brand;

}

//返回bean的对象

@Override

**public** Car getObject() **throws** Exception {

System.***out***.println("getObject....");

**return** **new** Car(brand, 800000);

}

//返回bean的类型

@Override

**public** Class<?> getObjectType() {

System.***out***.println("getObjectType....");

**return** Car.**class**;

}

//是不是单实例的

@Override

**public** **boolean** isSingleton() {

System.***out***.println("isSingleton...");

**return** **true**;

}

}

**配置文件：**

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<!--

**通过FactoryBean来配置bean的实例**

**class：指向FactoryBean的全类名**

**Property：配置FactoryBean的属性值**

**但实际返回的实例却是FactoryBean的getObject方法返回的实例**

-->

<bean id=*"car1"* **class=*"com.wp.spring.beans.factorybean.CarFactoryBean"***>

<property name=*"brand"* value=*"BMW"*></property>

</bean>

</beans>

**通过注解配置bean：**

组件扫描：spring能够从classpath下自动扫描，侦测和实例化具有特定注解的组件。

特定组件包括：

1. **@Component:基本注解，标识了一个受Spring管理的组件**
2. **@Resposition:标识持久层组件**
3. **@Service:标识服务层（业务层）组件**
4. **Controller:标识表现层组件**

对于扫描到的组件，Spring有默认的命名策略，使用非限定类名，第一个字母小写，也可以在注解中通过value属性值标识组件的名称

在组件类上使用了特定的注解之后，还需要在spring的配置文件中声明**<context:component-scan>**

**Base-package属性指定一个需要扫描的基类包，Spring容器将会扫描这个基类包里及其子包中的所有类。**

**当需要扫描多个包时，可以使用逗号分割**

**如果仅仅希望扫描特定的类而非基包下的所有类，可以首映resource-pattern属性过滤特定的类。实例：**

**<contextinclude-filter>:子节点表示要包含的目标类**

**<contextexclude-filter>子节点表示要清除在外的目标类**

**<context:component-scan>下可以拥有若干个<contextinclude-filter>和<contextexclude-filter>子节点。**

**范例：**

<?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-4.3.xsd"*>

<!-- 指定Spring IOC容器扫描的包-->

<!-- 可以通过resource-pattern指定扫描的资源 -->

<!-- <context:component-scan

base-package="com.wp.spring.beans.annotation"

**resource-pattern="repository/\*.class">**

<!--扫描实现repository接口的所有类-->

</context:component-scan> -->

<!--context:exclude-filter：子节点指定排除那些指定表达式的组件 -->

<!-- context:include-filter: 子节点表示指定包含那些表达式的组件，该子节点需要

**use-default-filters="false"**配合使用-->

<context:component-scan **use-default-filters=*"false"***

base-package=*"com.wp.spring.beans.annotation"*>

<!-- 根据指定的包进行指定的过滤 -->

<!-- <context:exclude-filter type="**annotation**"

expression="org.springframework.stereotype.Repository"/>-->

<!-- <context:include-filter type="**annotation"**

expression="org.springframework.stereotype.Repository"/> -->

<!-- <context:exclude-filter type="**assignable**"

expression="com.wp.spring.beans.annotation.repository.UserRepository"/> -->

<!-- 根据指定的类进行指定过滤 -->

<context:include-filter type=*"assignable"*

expression=*"com.wp.spring.beans.annotation.repository.UserRepository"*/>

</context:component-scan>

</beans>

**在注解中配置的名字就相当于在配置文件中配置的实例的id属性。**

利用注解实现bean之间的相互引用的配置：

范例：

**package** com.wp.spring.beans.annotation.repository;

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

**import** org.springframework.stereotype.Repository;

**import** com.wp.spring.beans.annotation.TestObject;

**@Repository("userRepository1")**

**public** **class** UserRepositoryImpl **implements** UserRepository {

**@Autowired(required=false) //假如容器里面没有这个对象**

**private** TestObject testObject;

@Override

**public** **void** save() {

System.***out***.println("UserRepositoryImpl save()....");

System.***out***.println("=====" + testObject);

}

}

**package** com.wp.spring.beans.annotation.service;

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

**import** org.springframework.stereotype.Service;

**import** com.wp.spring.beans.annotation.repository.UserRepository;

**@Service**

**package** com.wp.spring.beans.annotation.service;

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

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

**import** org.springframework.stereotype.Service;

**import** com.wp.spring.beans.annotation.repository.UserRepository;

@Service

**public** **class** UserService {

**private UserRepository userRepository;**  //接口类型。如果容器里有两个实现该接口的实例，该怎么自动装配？

**@Autowired**

**public** **void** setUserRepository(**@Qualifier("userRepositoryImpl") //若容器里不止一个该类型的bean实例** UserRepository userRepository) {

**this**.userRepository = userRepository;

}

**public** **void** add() {

System.***out***.println("UserService add().....");

userRepository.save();

}

}

**package** com.wp.spring.beans.annotation.repository;

**import** org.springframework.stereotype.Repository;

**@Repository("userRepository")**

**public** **class** UserjdbcRepository **implements** UserRepository {

@Override

**public** **void** save() {

System.***out***.println("UserjdbcRepository save()....");

}

}

**假如在容器里面有两个*private UserRepository userRepository;* 接口的实例，那么就查找有没有注解中配置的值与属性名称*userRepository* 相同的注解。如果有则加载这个bean。如果没有则抛出异常。**

@Autowired:为自动装配属性的注解标志。（即若容器中已经存在这个bean，那么就自动装配上）

**@Autowired(required=false)** 用来指定，该自动装配不是必须的（即如果IOC容器中没有这个bean，那么就不装配，而不是抛出异常）

**@Autowired注解自动装配具有兼容类型的单个Bean属性**

**--构造器，普通字段（即使是非public），一切具有参数的方法都可以应用@Autowired注解**

**默认情况下，所有使用@Autowired注解的属性都需要被设置，当spring找不到匹配的bean装配时，会弹出异常。当某属性允许不被设置时，可以设置@Autowired注解的required属性为false。**

**默认情况下当IOC容器里存在多个类型兼容的bean时，通过类型自动匹配则无法工作。此时可以在@Qualifiler（要引用的bean的类名称将第一个字母小写）注解里提供bean的名称，spring允许对方法的入参标注@Qualifiler已制定注入bean的名称。**

**@Autowired注解也可以应用在数组类型上，此时spring会把所有匹配的bean进行自动装配。**

**@Autowired注解也可以应用在集合属性上，此时spring读取该集合类型信息，然后自动装配所有与之兼容的bean。**

**@Autowired注解用在java.util.map上时，若该map的键值为stringnamespring将自动装配与之map值类型兼容的bean，此时bean的名称作为键值。**

**Spring面向切面编程（AOP）：**

1. 使用动态代理实现日志和执行函数的结合输出：

**package** com.wp.aop.helloworld;

**public** **class** ArithmeticCalculatorImpl **implements** ArithmeticCalculator{

@Override

**public** **int** add(**int** i, **int** j) {

**int** result = i+j;

**return** result;

}

@Override

**public** **int** sub(**int** i, **int** j) {

**int** result = i-j;

**return** result;

}

@Override

**public** **int** mul(**int** i, **int** j) {

**int** result = i\*j;

**return** result;

}

@Override

**public** **int** div(**int** i, **int** j) {

**int** result = i/j;

**return** result;

}

}

**package** com.wp.aop.helloworld;

**import** java.lang.reflect.InvocationHandler;

**import** java.lang.reflect.Method;

**import** java.lang.reflect.Proxy;

**import** java.util.Arrays;

**public** **class** ArithmeticCalculatorLoggingProxy {

//要代理的对象

**private** ArithmeticCalculator target;

**public** ArithmeticCalculatorLoggingProxy(ArithmeticCalculator target) {

**this**.target = target;

}

**public** ArithmeticCalculator getLoggingProxy() {

ArithmeticCalculator proxy = **null**;

//代理对象由哪一个类加载器负责加载

ClassLoader loader = target.getClass().getClassLoader();

//代理对象的类型，即其中有哪些方法

Class[] interfaces = **new** Class[] {ArithmeticCalculator.**class**};

//当调用代理对象其中的方法时，该执行的代码

InvocationHandler h = **new** InvocationHandler() { //内部类？？

/\*\*

\* proxy:正在返回的哪个代理对象，一般情况下，在invoke方法中都不使用该对象

\* method:正在被调用的方法。

\* args:调用方法时传入的参数

\* \*/

@Override

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

**throws** Throwable {

String methodName = method.getName(); //取得正在被调用的方法的名称

//日志

System.***out***.println("wenpan ---> The method " + methodName + " begins with " + Arrays.*asList*(args));

//执行方法

Object result = method.invoke(target, args); //仅仅用于执行方法？？？

//日志

System.***out***.println("wenpan ---> The method " + methodName + " ends width " + result);

**return** 0;

}

};

proxy = (ArithmeticCalculator) Proxy.*newProxyInstance*(loader, interfaces, h);

**return** proxy;

}

}

**package** com.wp.aop.helloworld;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

ArithmeticCalculator target = **new** ArithmeticCalculatorImpl(); //真实操作类

ArithmeticCalculator proxy = **new** ArithmeticCalculatorLoggingProxy(target).getLoggingProxy(); //取得代理类的实例

**int** result = proxy.add(1, 2);

System.***out***.println("--->" + result);

result = proxy.div(4, 2);

System.***out***.println("--->" + result);

}

}

Spring前置通知：

AspectJ:是java社区最流行的AOP框架。

1.SpringAOP

**前置通知：在目标方法（连接点）执行之前执行的通知**

**后置通知：在目标方法执行之后的通知（无论是否发生异常都会执行）。注意在后置通知中还不能访问目标方法执行后返回的结果。**

**步骤总结：**

1.加入jar包：

com.springsource.net.sf.cglib-2.2.0.jar

com.springsource.org.aopalliance-1.0.0.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

commons-logging-1.1.1.jar

spring-aop-4.0.0.RELEASE.jar

spring-aspects-4.0.0.RELEASE.jar

spring-beans-4.0.0.RELEASE.jar

spring-context-4.0.0.RELEASE.jar

spring-core-4.0.0.RELEASE.jar

spring-expression-4.0.0.RELEASE.jar

2.在配置文件中加入aop的命名空间

3.基于注解的方式

①.在配置文件中加入如下配置：

<!-- 使AspectJ 注解起作用：自动为匹配的类生成代理对象 -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

②.把横切关注点的代码抽象到切面的类中。

i.切面首先是一个IOC中的bean，即加入@Component

ii.切面还需要加入@Aspect注解

③.在类中声明各种通知：

@Berfore:前置通知

i.声明一个方法

ii.在方法前加入一个@Before注解

eg:@Before("execution(public int com.wp.aop.impl.ArithmeticCalculator.\*(int, int))")

iii:如果想要在方法中访问到方法的一些细节（如正在调用的方法名称，方法参数）可以在通知方法中声明一个类型为JoinPoint的参数。

范例：

配置文件：

<!-- 配置自动扫描的包 -->

<context:component-scan base-package=*"com.wp.aop.impl"*></context:component-scan>

<!-- 使AspectJ 注解起作用：自动为匹配的类生成代理对象 -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

**package** com.wp.aop.impl;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

**import** org.springframework.stereotype.Component;

//把这个类申明为一个切面：需要把该类放入到IOC容器中,再声明为一个切面

@Aspect

@Component

**public** **class** LoggingAspect { //日志切面

//声明该方法是一个前置通知，在目标方法执行之前执行

@Before("execution(public int com.wp.aop.impl.ArithmeticCalculator.\*(int, int))")

**public** **void** beforeMethod(**JoinPoint joinpoint**) {

String methodName = **joinpoint.getSignature().getName();**  //获取正在调用的方法名字

List<Object> args = Arrays.*asList*(**joinpoint.getArgs()**); //获取正在调用的方法的参数

System.***out***.println("The method " + methodName + " begins with " + args);

}

}

**package** com.wp.aop.impl;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

//1.创建IOC容器

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//2.从IOC容器中获取bean的实例

ArithmeticCalculator arithmeticCalculator = ctx.getBean(ArithmeticCalculator.**class**);

//3.使用bean

**int** result = arithmeticCalculator.add(3, 6);

System.***out***.println("result----->" + result);

result = arithmeticCalculator.div(12, 6);

System.***out***.println("result----->" + result);

}

}

**Spring切面的优先级**：（可以指定到底是哪个切面先执行）

比如指定日志切面和验证切面的执行顺序。

可以在切面类中使用**@order（优先级的值）**注解来配置切面的优先级。优先级值越小，越先执行。

**Spring重用切点表达式：**

范例：

**@Order(2)**

**@Aspect**

**@Component**

**public** **class** LoggingAspect { //该类相当于一个代理类

**/\*\***

**\* 定义一个方法，用于声明切入点表达式。一般的该方法中不需要填入其他的代码，**

**\* 使用@pointcut来声明切入点表达式**

**\* 后面的其他通知使用方法名来引入当前的切入点表达式**

**\* \*/**

**//指定切入点**

**@Pointcut("execution(public int com.wp.aop.ArithmeticCalculator.\*(..))")**

**public void declareJointpointExpression() {}**

/\*\*

\* 在com.wp.aop.ArithmeticCalculator接口的每一个实现类的每一个方法开始之前执行一段代码

\* \*/

@Before("declareJointpointExpression()")

**public** **void** beforeMethod(JoinPoint joinPoint) { //前置通知

String methodName = joinPoint.getSignature().getName(); //取得正在执行的方法名

Object[] args = joinPoint.getArgs(); //取得正在执行的方法的参数

System.***out***.println("The method " + methodName + " begins with " + Arrays.*asList*(args));

}

}

**基于配置文件来配置AOP:**

<?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-2.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>

<bean id=*"arithmeticCalculator"*

class=*"com.wp.aop.xml.ArithmeticCalculatorImpl"*></bean>

<!-- 配置切面的bean -->

<bean id=*"loggingAspect"*

class=*"com.wp.aop.xml.LoggingAspect"*></bean>

<bean id=*"ValidationAspect"*

class=*"com.wp.aop.xml.ValidationAspect"*></bean>

<!-- 配置AOP -->

<aop:config>

<!-- 配置切点表达式 -->

<aop:pointcut expression=*"execution(\* com.wp.aop.xml.ArithmeticCalculator.\*(int,int))"*

id=*"pointcut"*/>

<!-- 配置切面及通知 -->

<aop:aspect ref=*"loggingAspect"* order=*"2"*>

<aop:before method=*"beforeMethod"* pointcut-ref=*"pointcut"*/>

<aop:after method=*"afterMethod"* pointcut-ref=*"pointcut"*/>

<aop:after-returning method=*"afterReturning"* pointcut-ref=*"pointcut"* returning=*"result"*/>

<aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pointcut"* throwing=*"ex"*/>

<!--配置 环绕通知 -->

<aop:around method=*"aroundMethod"* pointcut-ref=*"pointcut"*/>

</aop:aspect>

<aop:aspect ref=*"ValidationAspect"* order=*"1"*>

<aop:before method=*"validateArgs"* pointcut-ref=*"pointcut"*/>

</aop:aspect>

</aop:config>

</beans>

**使用JdbcTemplate和JdbcDaoSupport:**

见下章节