# 概述

Spring是一个开源框架

Spring为简化企业级开发而生，使用Spring开发可以将Bean对象，Dao组件对象，Service组件对象等交给Spring容器来管理，这样使得很多复杂的代码在Spring中开发却变得非常的优雅和简洁，有效的降低代码的耦合度，极大的方便项目的后期维护、升级和扩展。

Spring是一个**IOC**(**DI**)和**AOP**容器框架。

Spring的优良特性

**非侵入式**：基于Spring开发的应用中的对象可以不依赖于Spring的API

**控制反转**：IOC（**Inversion of Control**），指的是将对象的创建权交给Spring去创建。使用Spring之前，对象的创建都是由我们自己在代码中new创建。而使用Spring之后。对象的创建都是由给了Spring框架。

**依赖注入**：DI（**Dependency Injection**）是指依赖的对象不需要手动调用setXX方法去设置，而是通过配置赋值。

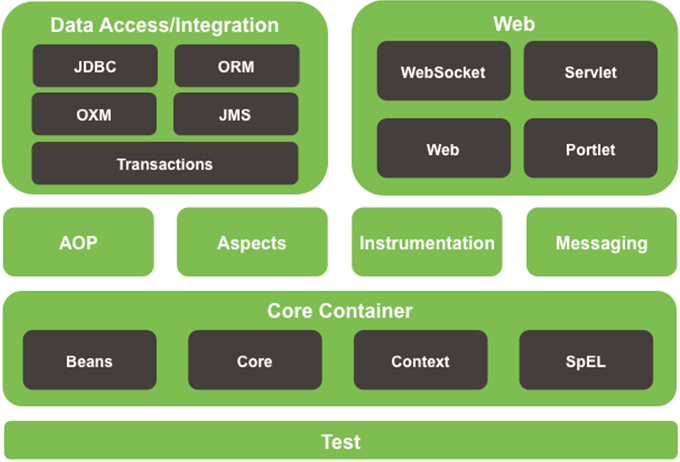
**面向切面编程**AOP（**Aspect Oriented Programming**）

**容器**：Spring是一个容器，因为它包含并且管理应用对象的生命周期

**组件化**：Spring实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用XML和Java注解组合这些对象。

**一站式**：在IOC和AOP的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上Spring 自身也提供了表述层的SpringMVC和持久层的Spring JDBC）。

## Spring模块



# IOC和DI

IOC(**Inverse Of Control**)指的是控制反转。使用spring之前，对象创建是代码中使用new去创建；使用spring之后，对象创建都交给spring去处理。

DI(**Dependency Injection**)指的是依赖注入。使用spring之前通过setXxx方法或构造器传入需要的对象引用。使用spring之后，依赖对象的赋值操作，都通过配置文件来完成。

**创建applicationContext.xml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">  </beans> |

**bean**标签 配置JavaBean对象，一个**bean**标签配置一个JavaBean对象。

**class**属性 设置配置的JavaBean类或工厂类的全类名

**id**属性 设置唯一的标识

**factory-method**属性 配置调用工厂类的方法

**factory-bean**属性 配置工厂类的实例对象

**parent**属性 配置继承的**bean**的id

**abstract**属性 表示当前的配置不能被实例化，只能用来继承

**depends-on**属性 表示依赖关系，其值为其他bean的id，表示要先创建该id的bean对象

**scope**属性 配置对象的作用域

**autowire**属性 配置自动注入

**init-method**属性 配置初始化方法

**destroy-method**属性 配置销毁方法

**property**标签 设置调用空参构造器创建对象，然后调用SetXxx方法注入属性

**name**属性 JavaBean类的属性名

**value**属性 JavaBean类的属性值

**ref**属性 表示引用一个对象，其值为引用的对象的**id**属性值

**constructor-arg**标签 设置构造器注入，此时调用有参构造器创建对象并注入属性

**name**属性 JavaBean构造器中参数名

**value**属性 JavaBean构造器中参数值

**index**属性 JavaBean构造器中参数索引

**type**属性 JavaBean构造器中参数类型

## 获取对象方式

|  |
| --- |
| public class Person {  Integer id;  String name;  Integer age;  } |

### 通过id

|  |
| --- |
| <bean id="p1" class="com.spring.pojo.Person">  <property name="id" value="1"/>  <property name="name" value="Tom"/>  <property name="phone" value="123456"/> </bean> |

|  |
| --- |
| @Test public void getBeanTestP1() {  // 在spring开发中，要先通过配置文件得到spring容器对象，ApplicationContext表示spring容器。  // ClassPathXmlApplicationContext是从Classpath类路径下加载xml配置文件，生成Spring容器对象  ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");  // getBean从容器中获取bean对象(通过id值获取)  Person person = (Person) applicationContext.getBean("p1"); } |

JavaBean对象是随着spring容器对象的创建而创建，而不是在调用getBean方法时创建的。如果多次调用getBean方法，默认获取的是同一个JavaBean对象，可以通过将bean标签中的**scope**属性设置为**prototype**使多次调用getBean方法获取的是不同的JavaBean对象。

获取Spring容器对象方式

获取Spring容器对象的方式有两种，一种是通过ClassPathXmlApplicationContext的构造器获取，其构造器中传入的路径参数是类路径参数，即从工程名开始算起；另一种是通过FileSystemXmlApplicationContext的构造器获取，其构造器中传入的路径参数是文件系统路径参数，包括了源码目录在内。

### 通过Class类型

|  |
| --- |
| <bean id="p1" class="com.spring.pojo.Person">  <property name="id" value="1"/>  <property name="name" value="Tom"/>  <property name="phone" value="123456"/> </bean> <bean id="p2" class="com.spring.pojo.Person"></bean> |

|  |
| --- |
| @Test public void getBeanTestP2(){  // FileSystemXmlApplicationContext是从文件系统中加载指定的xml文本，生成Spring容器对象  //其路径是从Module目录开始，不推荐使用这种方式，因为部署的工程是从类路径开始的  ApplicationContext applicationContext = new FileSystemXmlApplicationContext("src/main/resources/applicationContext.xml");  //通过Class类型获取bean对象，找到一个就直接返回。找不到就报错，找到多个也报错  Person person = applicationContext.getBean(Person.class);  System.*out*.println(person); } |

## 注入属性值方式

### 通过setXxx方法注入

通过<property/>注入属性是通过JavaBean中的setXxx方法注入的，注入的属性名为xxx，如果<property/>的name属性的值与xxx不一致，则会抛出BeanCreationException异常。此时是先通过无参构造器创建JavaBean对象，再通过SetXxx方法注入属性，和SetXxx方法中的参数名无关。需要注意的是SetXxx方法中的参数类型必须要和注入的属性类型一致，否则也会抛出BeanCreationException异常。

每次创建Spring容器对象都会调用对应的所有JavaBean的setXxx方法。

### 通过构造器注入

通过 <constructor-arg/>注入属性是通过JavaBean中的有参构造器注入的，通过这种方式注入时<constructor-arg/>中name的值必须和JavaBean中有参构造器的参数名一一对应，否则会抛出UnsatisfiedDependencyException异常。

而且<constructor-arg/>中value的值也要和JavaBean中有参构造器的参数类型相对应，必须要能互相转换，否则会抛出UnsatisfiedDependencyException异常。

**通过构造器参数名**

**通过构造器参数索引注入**

通过索引注入时index的值表示有参构造器中参数的索引位置，且索引值从0开始计算。

**通过构造器参数类型注入**

通过构造器参数类型注入时，如果注入的属性是基本数据类型，则type的值为int | double等等；如果注入的属性是引用数据类型，则type的值为该数据类型的全类名，如java.lang.Integer | java.lang.String等等。

|  |
| --- |
| <bean id="p2" class="com.spring.pojo.Person">  <constructor-arg name="id" value="2"/>  <constructor-arg index="1" value="Bom"/>  <constructor-arg type="java.lang.String" value="654321"/> </bean> |

### P名称空间注入

使用P名称空间时要在配置文件的beans中添加xmlns:p="http://www.springframework.org/schema/p"后才能使用。

|  |
| --- |
| <bean id="p3" class="com.spring.pojo.Person" p:id="3" p:name="Tim" p:phone="333"/> |

## 对象属性注入方式

### 子对象的注入

**ref属性注入**

Person类中有属性Car，通过ref属性设置id为car的bean来为其注入值，ref属性表示引用一个对象。

|  |
| --- |
| <bean id="car" class="com.spring.pojo.Car">  <property name="name" value="法拉第"/>  <property name="carNo" value="444"/> </bean> <bean id="p4" class="com.spring.pojo.Person">  <property name="name" value="Tom"/>  <property name="phone" value="4444"/>  <property name="id" value="4"/>  <property name="car" ref="car"/> </bean> |

**内部Bean注入**

在property标签内使用bean标签创建一个对象赋值于属性，这种方式称之为内部bean，内部bean不能通过spring容器获取到（不能供外部使用），只能用于赋值。

|  |
| --- |
| <bean id="p5" class="com.spring.pojo.Person">  <property name="name" value="Tom"/>  <property name="phone" value="4444"/>  <property name="id" value="4"/>  <property name="car">  <!--内部bean可以不用指定id，因为其无法被外部bean引用-->  <bean class="com.spring.pojo.Car">  <property name="name" value="法拉第"/>  <property name="carNo" value="444"/>  </bean>  </property> </bean> |

**子对象的属性注入**

子对象的属性的注入又称之为级联注入，其注入方式是通过子对象.子对象的属性名。这种注入方式有一个前提就是注入属性的子对象不能为null，如果为null就会抛出空指针异常。

|  |
| --- |
| <bean id="car6" class="com.spring.pojo.Car"/> <bean id="p6" class="com.spring.pojo.Person">  <property name="name" value="Tom"/>  <property name="phone" value="4444"/>  <property name="id" value="4"/>  <property name="car" ref="car6"/>  <!--级联属性赋值，要求级联的属性car不能为空，如果原先的car属性已有值则会被覆盖-->  <property name="car.name" value="not 法拉第"/>  <property name="car.carNo" value="666"/> </bean> |

### 集合类型属性注入

**List属性注入**

JavaBean中的List类型的属性可以使用<list/>来注入，其list集合中的值可以使用<value/>注入。

|  |
| --- |
| <bean id="p7" class="com.spring.pojo.Person">  <property name="id" value="7"/>  <property name="list">  <list>  <value>item1</value>  <value>item2</value>  <value>item3</value>  </list>  </property> </bean> |

**Map属性注入**

JavaBean中的Map类型的属性可以使用<**map**/>来注入，其map集合中的键值对的值可以用<**entry**/>来注入。

|  |
| --- |
| <bean id="p8" class="com.spring.pojo.Person">  <property name="id" value="8"/>  <property name="map">  <map>  <entry key="key1" value="value1"/>  <entry key="key2" value="value2"/>  <entry key="key3" value="value3"/>  </map>  </property> </bean> |

**Properties属性注入**

JavaBean中的Properties类型的属性可以使用<props/>来注入，其中键值对的值可以使用<prop/>来注入。

|  |
| --- |
| <bean id="p9" class="com.spring.pojo.Person">  <property name="id" value="9"/>  <property name="properties">  <props>  <prop key="pro1">value1</prop>  <prop key="pro2">value2</prop>  <prop key="pro3">value3</prop>  </props>  </property> </bean> |

### util名称空间

使用util名称空间要在beans中添加xmlns:util="http://www.springframework.org/schema/util"才能使用。

|  |
| --- |
| <util:list id="list">  <value>item1</value>  <value>item2</value>  <value>item3</value> </util:list> <bean id="p10" class="com.spring.pojo.Person">  <property name="id" value="8"/>  <property name="list" ref="list"/> </bean> |

## DI和工厂类

### 工厂实例对象注入

**factory-method**属性值表示需要调用的工厂类中的方法，对于非静态方法则需要先创建该工厂类的实例才能调用该方法，可以通过**factory-bean**属性来引入该工厂类的对象实例。

|  |
| --- |
| public Person createPerson(){  return new Person(1,"factory","123456"); } |

|  |
| --- |
| <bean id="factory" class="com.spring.factory.PersonFactory"/> <bean id="p11" factory-bean="factory" factory-method="createPerson"/> |

### 静态工厂注入

对于静态方法注入则不需要先创建该工厂类的实例就能直接调用。

|  |
| --- |
| public static Person createStaticPerson(){  return new Person(1,"static factory","123456"); } |

|  |
| --- |
| <bean id="p12" class="com.spring.factory.PersonFactory" factory-method="createStaticPerson"/> |

### FactoryBean接口注入

Spring中可以通过实现FactoryBean接口来完成接口注入，实现该接口后重写getObject、getObjectType 和isSingleton三个方法来确定注入的对象、对象类型和是否是单例模式。

|  |
| --- |
| //通过实现FactoryBean接口来完成FactoryBean接口注入 public class PersonFactoryBean implements FactoryBean {  @Override  public Object getObject() throws Exception {  return new Person(1,"FactoryBean","123");  }  @Override  public Class<?> getObjectType() {  return Person.class;  }  @Override  public boolean isSingleton() {  return true;  } } |

|  |
| --- |
| <bean id="p13" class="com.spring.factory.PersonFactoryBean"/> |

## 继承注入

可以通过parent属性的值来确定引用继承的父类的属性，如果子类的name属性中有和父类的name属性值相同的，则子类的值会覆盖父类的值。

abstract属性设置为true表示该JavaBean对象不能被实例化，只能用于其他JavaBean对象的parent属性来引用继承。

|  |
| --- |
| <bean id="parent" class="com.spring.pojo.Person" abstract="true">  <property name="name" value="father"/>  <property name="id" value="14"/>  <property name="phone" value="123456"/> </bean> <bean id="p14" class="com.spring.pojo.Person" parent="parent">  <property name="name" value="son"/> </bean> |

## 注入顺序

在配置文件中从上到下配置的顺序，就是他们默认创建的顺序，depends-on="b" 表示要创建a就先创建b，depends-on="b,c" 表示要创建a就要先创建b和c。

|  |
| --- |
| <bean id="a" class="com.spring.pojo.A" depends-on="b"/> <bean id="b" class="com.spring.pojo.B" depends-on="c"/> <bean id="c" class="com.spring.pojo.C"/> |

## 单例和多例

bean标签的scope属性可以用来设置对象的作用域，其值有四种，如下所示：

singleton 默认值，表示只有一个实例对象(单例)

1、单例在spring容器创建的时候一起被创建。

2、每次调用getBean方法都会返回容器中创建的那个对象。

prototype 多例，表示多次创建会创建多个不同对象

1、Spring容器创建的时候，bean对象不会跟着创建

2、每次调用getBean方法都会创建一个新对象实例返回

|  |
| --- |
| <bean id="p15" class="com.spring.pojo.Person" scope="singleton">  <property name="id" value="15"/> </bean> |

**单例管理的对象**

1.默认情况下,spring在读取xml文件的时候,就会创建对象。

2.在创建的对象的时候(先调用构造器),会去调用init-method=".."属性值中所指定的方法。

3.对象在被销毁的时候,会调用destroy-method="..."属性值中所指定的方法。(例如调用container.destroy()方法的时候)

4.lazy-init="true",可以让这个对象在第一次被访问的时候创建。

**非单例管理的对象**

1.spring读取xml文件的时候,不会创建对象.

2.在每一次访问这个对象的时候,spring容器都会创建这个对象,并且调用init-method=".."属性值中所指定的方法.

3.对象销毁的时候,spring容器不会帮我们调用任何方法,因为是非单例,这个类型的对象有很多个,spring容器一旦把这个对象交给你之后,就不再管理这个对象了。

## 配置文件自动注入

bean标签中的autowire属性表示自动注入，自动注入是指spring框架会自动的查找spring容器中存在的对象，通过某个匹配规则找到需要的对象并自动赋值。）

default和no 都表示不要自动注入。不手动设置就没值。就是null。

byName 表示会按照JavaBean中的setter方法去寻找对应的bean去赋值，例如setter方法为setCar234时则会寻找id为car234的bean进行赋值。

1、没有找到，就null值，

2、找到就赋值

byType 表示按照对象的类型进行查找并赋值。

1、找到一个就赋值。

2、没找到就null值。

3、找到多个就报错

constructor 表示按照构造器参数来查找并注入，且构造器中每一个参数都要按此规则查找并注入

1、先按照参数类型查找并注入

2、如果按参数类型查找到多个，接着按参数名做为id继续查找并赋值

|  |
| --- |
| <bean id="car" class="com.spring.pojo.Car">  <property name="name" value="法拉第" />  <property name="carNo" value="11111" /> </bean> <bean id="car2" class="com.spring.pojo.Car">  <property name="name" value="奔驰" />  <property name="carNo" value="22222" /> </bean> <bean id="p16" class="com.spring.pojo.Person" autowire="constructor">  <property name="id" value="16"/>  <property name="name" value="xml自动注入对象"/> </bean> |

## 注入null

可以通过<property/>标签下的<null/>来给该<property/>中的属性赋null。但是如果不针对该属性进行赋值时，该属性也是null值，因此<null/>显得有些多余。

|  |
| --- |
| <bean id="p17" class="com.spring.pojo.Person">  <property name="id" value="17"/>  <property name="name" value="null值测试"/>  <property name="phone">  <!--标识赋空值-->  <null/>  </property> </bean> |

## 对象的生命周期

### 生命周期

对于单例的bean而言，生命周期有11个步骤。

1、instantiate bean对象实例化，是在加载配置文件的时候实例化的。即启动spring容器加载配置文件，此时就实例化bean了。

2、populate properties 封装属性，spring对bean进行依赖注入。

3、如果bean实现了BeanNameAware接口,spring将bean的id传给setBeanName()方法。

4、如果bean实现了BeanFactoryAware接口,spring将调用setBeanFactory方法,将BeanFactory实例传进来。

5、如果bean实现了ApplicationContextAware()接口,spring将调用setApplicationContext()方法将应用上下文的引用传入

6、如果bean实现了BeanPostProcessor接口,spring将调用它们的postProcessBeforeInitialization接口方法

7、如果bean实现了InitializingBean接口,spring将调用它们的afterPropertiesSet接口方法。

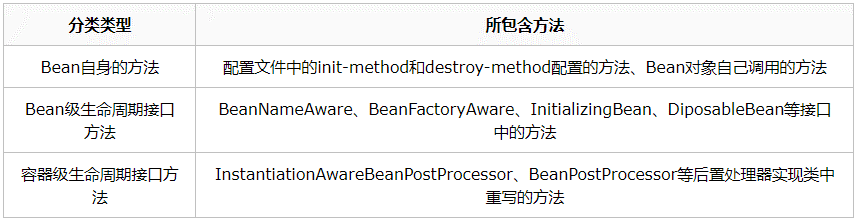
8、调用<bean init-method="init">指定初始化方法 init

9、如果bean实现了BeanPostProcessor接口,spring将调用它们的postProcessAfterInitialization接口方法

10、执行业务处理，此时bean已经准备就绪,可以被应用程序使用了,他们将一直驻留在应用上下文中,直到该应用上下文被销毁。

11、如果bean实现了DisposableBean接口,spring将调用它的distroy()接口方法。

12、调用<bean destroy-method="customerDestroy">指定销毁方法。



### 初始化和销毁

Person类中的初始化和销毁方法

|  |
| --- |
| public void init() {  System.*out*.println("对象被创建了，正在初始化"); } public void destroy() {  System.*out*.println("对象被销毁了。"); } |

bean标签中的init-method用于配置初始化方法；destroy-method用于配置销毁方法，该方法在spring容器关闭时（即ClassPathXmlApplicationContext对象调用close方法时）才会被调用，而且只对单例模式有效，即scope属性值为singleton时有效，对多例无效。

|  |
| --- |
| <bean id="p18" class="com.spring.pojo.Person" init-method="init" destroy-method="destroy">  <property name="id" value="18" />  <property name="name" value="演示生命周期方法"/> </bean> |

### 后置处理器

bean的后置处理器即BeanPostProcessor接口，其提供两个方法，postProcessBeforeInitialization在初始化之前调用，postProcessAfterInitialization在初始化之后调用。

|  |
| --- |
| public class MyBeanPostProcessor implements BeanPostProcessor {  /\* 初始化之前调用，第一个参数是当前正在初始化的对象，第二个参数是当前正在初始化对象的id值\*/  @Override  public Object postProcessBeforeInitialization(Object bean, String id) throws BeansException {  System.*out*.println("初始化之前== id=" + id + " , bean => " + bean);  if ("p18".equals(id)) {  Person person = (Person) bean;  person.setCar( new Car("中奖观众送一车", 222222) );  }  return bean;  }  /\* 初始化之后调用，需要注意两个方法都需要将初始化对象作为返回值返回 \*/  @Override  public Object postProcessAfterInitialization(Object bean, String id) throws BeansException {  System.*out*.println("初始化之后== id=" + id + " , bean => " + bean);  return bean;  } } |

后置处理器可以在对象初始化前后进行一些操作，如果要使用后置处理器，首先要实现BeanPostProcessor接口，然后在applicationContext.xml配置文件中配置。

|  |
| --- |
| <!-- 配置bean的后置处理器 --> <bean class="com.spring.processor.MyBeanPostProcessor"/> |

## 数据库连接池

### XML配置数据库连接池

|  |
| --- |
| <!-- 配置德鲁伊数据库连接池 --> <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="username" value="root" />  <property name="password" value="root" />  <property name="driverClassName" value="com.mysql.jdbc.Driver" />  <property name="url" value="jdbc:mysql://localhost:3306/test" />  <property name="initialSize" value="5" />  <property name="maxActive" value="10" /> </bean> |

|  |
| --- |
| public void test1() throws Exception {  ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");  DataSource dataSource = (DataSource) applicationContext.getBean("dataSource");  System.*out*.println( dataSource.getConnection() ); } |

### properties文件配置数据库连接池

jdbc.properties

|  |
| --- |
| username=root password=root url=jdbc:mysql://localhost:3306/test driverClassName=com.mysql.jdbc.Driver initialSize=5 maxActive=10 |

|  |
| --- |
| <!-- 这个类用来加载properties属性配置文件 --> <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <!-- location属性，设置你要加载的属性配置文件路径  classpath 表示从类路径下开始查找  classpath:jdbc.properties 表示从类路径下找jdbc.properties属性配置文件-->  <property name="location" value="classpath:jdbc.properties" /> </bean> <!-- 配置德鲁伊数据库连接池 --> <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="username" value="${username}" />  <property name="password" value="${password}" />  <property name="driverClassName" value="${driverClassName}" />  <property name="url" value="${url}" />  <property name="initialSize" value="${initialSize}" />  <property name="maxActive" value="${maxActive}" /> </bean> |

#### context名称空间加载properties文件

使用context名称空间加载properties文件时需要将其内部的username这个key的值改为其他的值，例如user，否则该key的值会被系统登陆用户名覆盖而导致可能无法访问数据库。

|  |
| --- |
| <!-- 这个标签就是去加载属性配置文件 --> <context:property-placeholder location="classpath:jdbc.properties"/> <!-- 配置德鲁伊数据库连接池 --> <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="username" value="${user}" />  <property name="password" value="${password}" />  <property name="driverClassName" value="${driverClassName}" />  <property name="url" value="${url}" />  <property name="initialSize" value="${initialSize}" />  <property name="maxActive" value="${maxActive}" /> </bean> |

## SpEL表达式

表达式语言(**Spring Expression LanguageSpring**)，简称SpEL，支持运行时查询并可以操作对象图。和JSP页面上的EL表达式、Struts2中用到的OGNL表达式一样，SpEL根据JavaBean风格的getXxx()、setXxx()方法定义的属性访问对象图，完全符合我们熟悉的操作习惯。

### 基本语法

SpEL使用#{}作为定界符，所有在大框号中的字符都将被认为是SpEL表达式。

### 使用字面量

|  |
| --- |
| <!--整数：--> <property name="count" value="#{5}"/> <!--小数：--> <property name="frequency" value="#{89.7}"/> <!--科学计数法：--> <property name="capacity" value="#{1e4}"/> <!--String类型的字面量可以使用单引号或者双引号作为字符串的定界符号--> <property name="name1" value="#{'Chuck'}"/> <property name='name2' value='#{"Chuck"}'/> <property name="enabled" value="#{false}"/> |

### 运算符

算术运算符：+、-、\*、/、%、^

字符串连接：+

比较运算符：<、>、==、<=、>=、lt、gt、eq、le、ge

逻辑运算符：and, or, not, |

三目运算符：判断条件?判断结果为true时的取值:判断结果为false时的取值

正则表达式：matches

|  |
| --- |
| <property name="salary" value="#{30000\*12}"></property> |

### 引用属性

#{}可以用于引用其他Bean作为属性值或该Bean的属性的属性值作为属性值。

|  |
| --- |
| <bean id="car" class="com.SpEL.pojo.Car">  <property name="name" value="吐优塔"/>  <property name="carNo" value="京B123456" /> </bean>  <bean id="pel" class="com.SpEL.pojo.Man">  <property name="car" value="#{car}" /> </bean>  <bean id="pel" class="com.SpEL.pojo.Man">  <property name="name" value="#{car.name}"/> </bean> |

### 调用方法

|  |
| --- |
| public String getPhoneNum() {  return "13900009999"; } public static String phoneNum2() {  return "13976300276"; } |

|  |
| --- |
| <!--调用非静态方法--> <bean id="pel" class="com.SpEL.pojo.Man">  <property name="phone" value="#{car.getPhoneNum()}"/> </bean> <!--调用静态方法--> <bean id="pel" class="com.SpEL.pojo.Man">  <property name="phone" value="#{T(com.SpEL.pojo.Car).phoneNum2()}"/> </bean> |

## 注解功能

### 常用注解

Spring配置bean的常用注解有

@Controller 配置Controller控制器

@Service 配置Service层的组件

@Repository 配置Dao层的组件

@Component 除了Controller、Service、Dao之外需要配置的bean对象都使用@Component

@Scope 配置作用域，单例或多例

|  |
| --- |
| /\*@Component 注解等价于：<bean id="book" class="com.last.pojo.Book" /> \*/ @Component public class Book { } /\*@Repository 注解等价于：<bean id="bookDao" class="com.last.dao.BookDao" /> \*/ @Repository public class BookDao { } /\*@Service 注解等价于：<bean id="bookService" class="com.last.service.BookService" />\*/ @Service public class BookService { } /\*@Controller 注解等价于：<bean id="bookController" class="com.last.controller.BookController" />\*/ @Controller public class BookController { } |

如果要使这些注解生效还需要在applicationContext.xml文件中进行配置，设定扫描的包，Spring框架就会根据设置的包名去扫描包内含有注解的类。

|  |
| --- |
| <!-- 配置包扫描 base-package 设置你要扫描的包名(扫描该包及其子包） --> <context:component-scan base-package="com.last"/> |

### 扫描过滤

Spring中可以在<context:component-scan>标签中设置扫描时不要包含的类和扫描时要包含的类，从而达到一种过滤的作用。可以使用如下两个标签来达到这个功能：

<context:include-filter /> 设置包含的内容

注意：通常需要与use-default-filters属性配合使用才能够达到“仅包含某些组件”这样的效果，即通过将use-default-filters属性设置为false，

<context:exclude-filter /> 设置排除的内容

|  |  |  |
| --- | --- | --- |
| 类别 | 示例 | **说明** |
| annotation | com.xxx.XxxAnnotation | 过滤所有标注了XxxAnnotation的类。这个规则根据目标组件是否标注了指定类型的注解进行过滤 |
| assignable | com.xxx.BaseXxx | 过滤所有BaseXxx类的子类。这个规则根据目标组件是否是指定类型的子类的方式进行过滤。 |
| aspectj | com.xxx.\*Service+ | 所有类名是以Service结束的，或这样的类的子类。这个规则根据AspectJ表达式进行过滤。 |
| regex | com\.xxx\.anno\.\* | 所有com.xxx.anno包下的类。这个规则根据正则表达式匹配到的类名进行过滤。 |
| custom | com.xxx.XxxTypeFilter | 使用XxxTypeFilter类通过编码的方式自定义过滤规则。该类要实现org.springframework.core.type.filter.TypeFilter  接口 |

|  |
| --- |
| <!-- use-default-filters="false" 设置取消默认包含规则 --> <context:component-scan base-package="com.last" use-default-filters="false">  <!-- context:include-filter 设置包含的内容 -->  <context:include-filter type="annotation" expression="org.springframework.stereotype.Service"/>  <!-- context:exclude-filter 设置排除的内容 -->  <context:exclude-filter type="assignable" expression="com.last.service.BookService"/> </context:component-scan> |

### @Autowired和@Qualifier

#### 标记属性

@Autowired注解会自动根据标注的对象类型在Spring容器中查找相对应的类，如果找到就自动装配。使用@Autowired注解不需要get | set方法。

**单个bean**

|  |
| --- |
| @Service public class BookService {  /\*@Autowired 实现了自动注入，按照类型查找，找到就赋值（注入）\*/  @Autowired  private BookDao bookDao; } |

**多个bean**

如果同类型（本类及其子类）的bean不止一个，那么就会默认根据@Autowired注解标记的属性名作为id来查找bean进行自动装配。

|  |
| --- |
| @Repository public class BookDao { } @Repository public class BookDaoExt extends BookDao{ } @Service public class BookService {  /\*@Autowired 实现了自动注入<br/>  \* 1、先按照类型查找（包括该类型的子类），找到就赋值（注入）  \* 2、按类型查找到多个，接着按属性名做为id继续查找并注入\*/  @Autowired  private BookDao bookDao; } |

此时按类型查找会找到两个BookDao，然后会按属性名作为id来查找，如果属性名为bookDao则查找到的是BookDao的对象，如果属性名是bookDaoExt则查找到的是BookDaoExt的对象。

**指定装配**

如果根据属性名作为id还是找不到bean，可以使用@Qualifier注解指定目标bean的id。

|  |
| --- |
| @Service public class BookService {  /\* @Autowired 实现了自动注入<br/>  \*1、先按照类型查找，找到就赋值（注入）  \*2、按类型查找到多个，接着按属性名做为id继续查找并注入  \*3、使用@Qualifier("bookDaoExt")指定id查找并注入,指定id值后原属性名即bookDao无效\*/  @Autowired  @Qualifier("bookDaoExt")  private BookDao bookDao; } |

如果id冲突可以在注解上加括号写上不同的id，如@Service(“新id”)，则调用时以“新id”为准。

**@Autowired注解的required属性作用**

该属性的值默认为true，即不允许@Autowired注解标记的属性值为null，可以通过将其设置为false来允许@Autowired注解标记的属性值为null。

#### 标记方法

|  |
| --- |
| /\*@Autowired 注解标注在方法上，此方法会在bean对象初始化的时候自动调用，一般用于初始化对象  \* 1、方法中传入的参数先按照类型查找，找到就赋值（注入）  \* 2、按类型查找到多个，接着按属性名做为id继续查找并注入  \* 3、使用@Qualifier("bookDaoExt")指定id查找并注入,指定id值之后。原属性名即dao无效  \* 4、使用required属性设置值为false,允许不调用此方法 \*/ @Autowired(required=false) public void method(@Qualifier("bookDaoExt") BookDao dao) {  System.*out*.println("abc调用了 ====>>>>" + dao);  this.bookDao = dao; } |

### 泛型注入

**泛型代码**

|  |
| --- |
| @Repository public abstract class BaseDao<T> {  public abstract void save(T entity); } @Repository public class BookDao extends BaseDao<Book> {  @Override  public void save(Book entity) {  System.*out*.println("BookDao 保存 book对象 --->>>" + entity);  } } @Repository public class UserDao extends BaseDao<User> {  @Override  public void save(User entity) {  System.*out*.println("UserDao 保存 User对象 --->>>" + entity);  } } public abstract class BaseService<T> {  @Autowired  private BaseDao<T> dao;  public void save(T entity) {  dao.save(entity);  } } @Service public class BookService extends BaseService<Book> { } @Service public class UserService extends BaseService<User> { } |

**测试方法**

|  |
| --- |
| public void test() throws Exception {  ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");  //执行完下面代码得到BookService对象，从BookService类中可以确定泛型是Book  BookService bookService = (BookService) applicationContext.getBean("bookService");  //从Spring容器中查找BookService类的属性baseDao得到UserDao和BookDao  //比较Service和Dao的泛型，因为两者泛型一致，所以最后得到BookDao  bookService.save(new Book());  System.*out*.println("=================================================");  UserService userService = (UserService) applicationContext.getBean("userService");  userService.save(new User()); } |

### Spring专有测试

#### 一般测试类

|  |
| --- |
| public class SpringTest {  @Test  public void test1() throws Exception {  ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");  BookService bookService = (BookService) applicationContext.getBean("bookService");  bookService.save(new Book());  UserService userService = (UserService) applicationContext.getBean("userService");  userService.save(new User());  } } |

#### Spring专有测试类

|  |
| --- |
| //@ContextConfiguration告诉Spring的测试，上哪去找spring容器需要的配置文件。 @ContextConfiguration(locations = "classpath:applicationContext.xml") //@RunWith告诉junit使用Spring扩展好的junit运行器去跑测试 @RunWith(SpringJUnit4ClassRunner.class) public class SpringJunitTest {  @Autowired  private BookService bookService;  @Autowired  private UserService userService;   @Test  public void test1() throws Exception {  userService.save(new User());  bookService.save(new Book());  } } |

# AOP

AOP（**Aspect Oriented Programming**）是面向切面编程，指的是程序运行期间，动态地将某段代码插入到原来方法代码的某些位置中，即动态代理。

Spring底层实现动态代理的原理是使用了jdkProxy代理和cglib字节码增强代理，可以在aop:config标签种使用proxy-target-class属性并将其设置为true来强制使用cglib动态代理。

Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

AOP在事务管理方面，Spring使用AOP来完成声明式的事务管理有annotation和xml两种形式。开发中，方便代码编写，很多时候都是在spring配置文件中配置事务管理器并开启事务控制注解。在业务类或业务类方法中添加@Transactional实现事务控制。

## AOP术语

通知(**Advice**)： 扩展增强的代码，比如前置增强代码。后置增强代码。异常增强代码。

切面(**Aspect**)： 切面就是包含有通知代码的类叫切面。

横切关注点： 可以添加增强代码的位置，如前置位置，后置位置，异常位置和返回值位置。

目标(**Target**)： 目标对象就是被关注的对象或者被代理的对象。

代理(**Proxy**)： 为了拦截目标对象方法而被创建出来的那个对象。

连接点(**Joinpoint**)： 连接点指的是横切关注点和程序代码的连接。

切入点(**pointcut**)： 切入点指真正处理的连接点，在Spring中切入点通过aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条。

横切关注点

前置通知位置

后置通知位置

异常通知位置

返回通知位置

切面类

前置通知logBefore()

后置通知logAfter()

异常通知logAfterThrowing()

返回通知logAfterReturning()

被代理对象（类）

method1 method2 method3

链接点：横切关注点和代码的连接

切入点：真正直接处理的链接点

## AOP简单实现

### LogUtils工具类

|  |
| --- |
| /\* @Aspect 表示这个页面是个切面类 \*/ @Aspect @Component public class LogUtils {  //@Before 注解表示前置通知，每个通知都要通过切入点表达式来告诉Spring框架该通知对哪些方法生效  @Before(value = "execution(int com.aop.pojo.Calculator.add(int, int))")  public static void logBefore(JoinPoint joinPoint) {  System.*out*.println("前置通知");  }  } |

### applicationContext.xml

|  |
| --- |
| <!-- 配置包扫描 --> <context:component-scan base-package="com"/> <!-- 下述标签表示支持注解，会给切面方法所有目标对象添加代理 --> <aop:aspectj-autoproxy/> |

### 测试类

|  |
| --- |
| @ContextConfiguration(locations = "classpath:applicationContext.xml")//指定配置文件 @RunWith(SpringJUnit4ClassRunner.class)//Spring测试类运行器 public class SpringTest {  //如果Calculator实现了Calculate接口，就要使用Calculate接收，否则就要使用Calculator  @Autowired  Calculate calculate;  @Test  public void test1() {  calculate.add(100,100);  } } |

## 切入点表达式

切入点表达式是告诉Spring当前通知对哪些方法有效。

@PointCut切入点表达式语法格式是：execution (访问权限 返回值类型 方法全限定名(参数类型列表))

开发中一般只会用到 execution(public \* com.xxx.service..\*Service\*.\*(..)) 这种格式。

### “\*”限定符

\* 表示任意的意思

1)匹配某全类名下，任意或多个方法。

execution( public int com.xxx.pojo.Calculator.\*(int, int) )

以上的星表示任意方法都有效(参数类型必须是两个int)

execution( public int com.xxx.pojo.Calculator.save\*(int, int) )

以上的星表示方法名必须以save打头(参数类型必须是两个int)。

2)在Spring中只有public权限能拦截到，访问权限可以省略（访问权限不能写\*）。

execution( int com.xxx.pojo.Calculator.add(int, int) )

以上的切入点表达式，可以省略public关键字

3)匹配任意类型的返回值，可以使用 \* 表示

execution(public \* com.xxx.pojo.Calculator.add(int, int) )

以上的星，表示任意的返回值类型都可以

4)匹配任意子包。

execution(public int com.\*.pojo.Calculator.add(int, int) )

以上的星表示 com和pojo中间有任意子包名都可以（一层）。

com.xxx.pojo 匹配

com.a.pojo 匹配

com.at.pojo 匹配

com.a.b.pojo 不匹配

5)任意类型参数

execution(public int com.xxx.pojo.Calculator.add(int, \*) )

以上的星，表示第二个参数类型可以是任意的类型。

### “..”限定符

.. 可以匹配多层路径或任意多个任意类型参数

1)任意层级的包

execution(public int com..pojo.Calculator.add(int, int) )

以上点点.. 限定符，表示com和pojo中间，可以是任意层级的子包

com.a.pojo 匹配

com.a.b.pojo 匹配

com.xxx.a.b.c.d.pojo 匹配

2)任意个数，任意类型的参数

execution(public int com.xxx.pojo.Calculator.add(..) )

以上点点.. 的作用是，不限定参数个数，不限定参数类型

### 模糊匹配

// 表示任意返回值，任意方法全限定符，任意参数

execution(\* \*(..))

// 表示任意返回值，任意包名+任意方法名，任意参数

execution(\* \*.\*(..))

### 精确匹配

execution( public int com.xxx.pojo.Calculator.add(int, int) )

public 是访问权限

int 返回值类型

com.xxx.pojo.Calculator.add 包名+类名+方法名

(int, int) 参数类型列表

### 切入点表达式连接

&& 表示需要同时满足两个表达式

@Before("execution(public int com.xxx.aop.Calculator.add(int, int))"

+ " && "+ "execution(public \* com.xxx.aop.Calculator.add(..))")

|| 表示两个条件只需要满足一个，就会被匹配到

@Before("execution(public int com.xxx.aop.Calculator.add(int, int))"

+ " || "+ "execution(public \* com.xxx.aop.Calculator.a\*(int))")

### 切入点表达式复用

切入点表达式在Spring中提供了复用的功能，避免了多次写同样的切入点表达式。

首先定义一个空方法，然后在方法上使用@Pointcut注解定义一个切入点表达式，最后在需要复用此切入点表达式处使用方法调用替代。

|  |
| --- |
| @Pointcut("execution(int com.aop.pojo.Calculator.add(int, int)))") public static void pointCut(){} @After(value = "pointCut()") public static void logAfter() {  System.*out*.println("后置通知"); } |

## 切面代理对象

在Spring中，可以对有接口的对象和无接口的对象分别进行代理，在使用上有些细微的差别。在Spring AOP功能上，Spring的底层就是使用Jdk动态代理和Cglib动态代理。

1、被代理的对象有接口，则Spring底层默认使用Jdk动态代理，使用接口接收代理对象。

2、被代理的对象没有接口，则Spring底层默认使用Cglib动态代理，使用类来接收被代理的对象。

## 通知和切面执行顺序

### 通知执行顺序

Spring通知的执行顺序是:

正常情况：前置通知🡪目标方法🡪后置通知🡪返回值通知

异常情况：前置通知🡪目标方法🡪后置通知🡪异常通知

**前置通知**

|  |
| --- |
| //@Before 注解表示前置通知 @Before(value = "execution(int com.aop.pojo.Calculator.add(int, int))") public static void logBefore() {  System.*out*.println("前置通知"); } |

**后置通知**

|  |
| --- |
| //@After 注解表示后置通知 @After(value = "execution(int com.aop.pojo.Calculator.add(int, int))") public static void logAfter() {  System.*out*.println("后置通知"); } |

**异常通知**

|  |
| --- |
| //@AfterThrowing 注解表示异常通知 @AfterThrowing(value = "execution(int com.aop.pojo.Calculator.div(int, int))") public static void logAfterThrowing() {  System.*out*.println("异常通知"); } |

**返回值通知**

|  |
| --- |
| //@AfterReturning 注解表示返回通知，在代理对象返回值之前接收返回值使用 @AfterReturning(value="execution( public int com.aop.pojo.Calculator.add(int, int) )") public static void logAfterReturning() {  System.*out*.println("返回通知，获取返回值："); } |

### 多个切面执行顺序

当有多个切面，且每个切面有多个通知时，通知的执行顺序默认是由切面类的类名首字母先后顺序决定的。也可以通过在切面类上使用@Order注解来决定通知执行的顺序，其值越小则表示越先执行。

**Target.java**

|  |
| --- |
| @Component public class Target {  public void target(){  System.*out*.println("this is target" );  int i = 1/0;  } } |

**AspectA.java**

|  |
| --- |
| @Aspect @Component public class AspectA {  @Before(value = "execution(void com.proxy.target.Target.target())")  public void before(){ System.*out*.println("A before"); }  @After(value = "execution(void com.proxy.target.Target.target())")  public void after(){ System.*out*.println("A after"); }  @AfterThrowing(value = "execution(void com.proxy.target.Target.target())")  public void throwing(){ System.*out*.println("A throwing"); } } |

**AspectB.java**

|  |
| --- |
| @Aspect @Component public class AspectB {  @Before(value = "execution(void com.proxy.target.Target.target())")  public void before(){ System.*out*.println("B before"); }  @After(value = "execution(void com.proxy.target.Target.target())")  public void after(){ System.*out*.println("B after"); }  @AfterThrowing(value = "execution(void com.proxy.target.Target.target())")  public void throwing(){ System.*out*.println("B throwing"); } } |

**AspectC.java**

|  |
| --- |
| @Aspect  @Order(1) @Component public class AspectC {  @Before(value = "execution(void com.proxy.target.Target.target())")  public void before(){ System.*out*.println("C before"); }  @After(value = "execution(void com.proxy.target.Target.target())")  public void after(){ System.*out*.println("C after"); }  @AfterThrowing(value = "execution(void com.proxy.target.Target.target())")  public void throwing(){ System.*out*.println("C throwing"); } } |

**TargetTest.java**

|  |
| --- |
| @ContextConfiguration(locations = "classpath:applicationContext.xml") @RunWith(SpringJUnit4ClassRunner.class) public class TargetTest {  @Autowired  Target target;  @Test  public void target(){ target.target(); } } |

该示例切面类执行顺序为CBA，因为C上有@order注解标注，因此优先执行。具体通知执行顺序如下所示：

C before🡪A before🡪B before🡪this is target🡪B after🡪B throwing🡪A after🡪A throwing🡪C after🡪C throwing

## 获取拦截方法信息

### 获取连接点信息

JoinPoint是连接点信息，只要在通知方法的参数中加入JoinPoint参数就可以获取到拦截方法的方法名和方法参数。

|  |
| --- |
| @Before(value = "execution(int com.aop.pojo.Calculator.add(int, int))") public static void logBefore(JoinPoint jp) {  //joinPoint.getSignature().getName()用于获取方法名，joinPoint.getArgs()用于获取方法参数  System.*out*.println("前置通知方法名" + jp.getSignature().getName() +  "参数是" + Arrays.*asList*(jp.getArgs())); } |

### 获取拦截方法的返回值和异常信息

获取返回值的步骤分为两步，首先在返回值通知的方法中添加参数Object result；然后在@AfterReturning注解中添加参数returning=”参数名”。

|  |
| --- |
| //returning属性设置目标方法的返回值由哪个参数来接收 @AfterReturning(value="execution(public int com.aop.pojo.Calculator.add(int, int))",returning = "result") public static void logAfterReturning(JoinPoint jp, Object result) {  System.*out*.println("返回通知方法是"+jp.getSignature().getName()+"获取返回值" + result); } |

获取异常信息的步骤也是两步，首先在异常通知的方法中添加参数Exception exception，该参数的类型越大越好，以便于接收方法抛出的异常；然后在@AfterThrowing注解中添加参数throwing=“参数名“。

|  |
| --- |
| //throwing属性设置目标方法抛出的异常由哪个参数来接收 @AfterThrowing(value = "execution(int com.aop.pojo.Calculator.div(int, int))",throwing = "e") public static void logAfterThrowing(JoinPoint jp, Exception e) {  System.*out*.println("异常代码方法是" + jp.getSignature().getName() + "出现的异常是"+ e); } |

## 环绕通知

1、环绕通知使用@Around注解。

2、环绕通知如果和其他通知同时执行，环绕通知会优先于其他通知之前执行。

3、绕通知一定要有返回值（如果没有返回值，后面的其他通知就无法接收到目标方法执行的结果）。

4、环绕通知中，如果拦截异常，一定要往外抛，否则其他的异常通知无法捕获到异常的。

|  |
| --- |
| @Around(value = "execution( public int com.aop.pojo.Calculator.\*(int, int) )") public static Object around(ProceedingJoinPoint pjp) throws Throwable {  Object result;  try {  try {  System.*out*.println("环绕前置通知");  result = pjp.proceed();// 环绕通知需要自己调用目标方法执行  } finally {  System.*out*.println("环绕后置通知");  }  System.*out*.println("环绕返回通知，返回值是：" + result);  } catch (Throwable e) {  e.printStackTrace();  System.*out*.println("环绕异常通知：" + e);  throw e;  }  return result; } |

## 使用XML文件配置AOP程序

|  |
| --- |
| <bean id="target" class="com.proxy.target.Target"></bean><!-- 配置目标对象 --> <bean id="aspectA" class="com.proxy.aspect.AspectA"></bean><!-- 配置切面类对象 --> <aop:config><!-- aop:config 标签配置aop功能 -->  <!-- aop:pointcut定义可复用的切入点表达式,expression 切入点表达式,id 给切入点表达式定义一个唯一的标识 -->  <aop:pointcut expression="execution( public int com.proxy.target.Target.\*(int,int) )" id="pointcut1"/>  <!-- 配置切面 ref="logUtils" 表示使用id为logUtils对象实例做为切面类.相当于@Aspect注解作用 order配置执行顺序-->  <aop:aspect ref="aspectA" order="1">  <!-- aop:before 标签配置前置通知,method 配置前置通知的方法,pointcut 配置切入点表达式 -->  <aop:before method="before" pointcut="execution( public int com.proxy.target.Target.\*(int, int) )"/>  <!-- aop:after 配置的是后置通知,pointcut-ref 属性设置引用哪个切入点表达式 -->  <aop:after method="after" pointcut-ref="pointcut1"/>  <!-- aop:after-throwing 标签配置异常通知,throwing配置通知方法中哪个参数接收异常 -->  <aop:after-throwing method="throwing" pointcut-ref="pointcut1" throwing="e"/>  <!-- aop:after-returning 配置返回通知,returning配置返回值通知方法中哪个参数接收返回值 -->  <aop:after-returning method="returning" pointcut-ref="pointcut1" returning="result"/>  </aop:aspect> </aop:config> |

|  |
| --- |
| public class AspectA {  public void before(){ System.*out*.println("A before"); }  public void after(){ System.*out*.println("A after"); }  public void throwing(Exception e){ System.*out*.println("A throwing:" + e); }  public void returning(Object result){ System.*out*.println("A returning:" + result); } } |

|  |
| --- |
| @Component public class Target {  public int target(int i,int j){  System.*out*.println("this is target");  return i/j;  } } |

# DataAccess

Spring框架中DataAccess模块支持对数据库的访问操作，提供了JdbcTemplate类来操作数据库，其作用类似于DbUtils中的QueryRunner，可以执行SQL语句，操作数据库。

JdbcTemplate类的update方法用于执行update、insert和delete类型的SQL语句，query方法执行查询返回多行记录的select类型的SQL语句，queryForObject方法执行查询返回一行记录的select类型的SQL语句。

## 配置文件

在进行数据库访问操作前需要对相关配置文件进行配置，jdbc.properties和applicationContext.xml如下所示。

详细数据库表见Appendix的Spring中table\_1

|  |
| --- |
| user=root password=root url=jdbc:mysql://localhost:3306/jdbctemplate driverClassName=com.mysql.jdbc.Driver initialSize=5 maxActive=10 |

|  |
| --- |
| <?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">  <!-- 加载jdbc.properties属性配置文件 -->  <context:property-placeholder location="classpath:jdbc.properties"/>  <!-- 配置数据库连接池 -->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="username" value="${user}" />  <property name="password" value="${password}" />  <property name="url" value="${url}" />  <property name="driverClassName" value="${driverClassName}" />  <property name="initialSize" value="${initialSize}" />  <property name="maxActive" value="${maxActive}" />  </bean>  </beans> |

Spring框架中提供的JdbcTemplate类可以执行SQL语句，其使用前需要在applicationContext.xml配置文件中进行配置。

|  |
| --- |
| <!-- jdbcTemplate专门用来操作数据库执行sql语句-->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource"/> </bean> |

使用@Autowired注解标注JdbcTemplate属性时，为了实现自动装配功能还需要在配置文件中配置包扫描器。

|  |
| --- |
| <!-- 配置包扫描 -->  <context:component-scan base-package="com"></context:component-scan> |

## 数据访问

测试数据源

|  |
| --- |
| @ContextConfiguration(locations = "classpath:applicationContext.xml") @RunWith(SpringJUnit4ClassRunner.class) public class SpringTest {  @Autowired  private DataSource dataSource;  @Test  public void test1() throws SQLException {  System.*out*.println(dataSource.getConnection());  }  } |

### update

插入单条数据

|  |
| --- |
| @Autowired private JdbcTemplate jdbcTemplate;  @Test public void test2() {  String sql = "update employee set salary = ? where id = ?";  /\* update方法执行insert、update、delete的sql语句 \*/  System.*out*.println(jdbcTemplate.update(sql, new BigDecimal(1300), 5)); } |

批量插入数据

|  |
| --- |
| @Test public void test3() {  String sql = "insert into employee(`name`,`salary`) values(?,?)";  /\* 插入一条记录。参数是一个一维 数组<br/>  \* 一次插入多条记录。那么 参数就是多个一维数组<br/> \*/  // jdbcTemplate.update(sql, "国哥我爱你！", new BigDecimal(19999));  // 插入多条记录，需要的多个一维数组  List<Object[]> batchArgs = new ArrayList<>();  batchArgs.add(new Object[]{"插一条", new BigDecimal(12341)});  batchArgs.add(new Object[]{"插二条", new BigDecimal(12341)});  batchArgs.add(new Object[]{"插三条", new BigDecimal(12341)});  jdbcTemplate.batchUpdate(sql, batchArgs); } |

使用带有具名参数的SQL语句插入一条员工记录，并以Map形式传入参数值

|  |
| --- |
| @Test public void test7() {  /\* :name 是占位符 ?（占位符就是传递的参数）<br/>  \* :name name就是这个参数的名称,:salary salary就是参数的名称 \*/  String sql = "insert into employee(`name`,`salary`) values(:name,:salary)";  /\* update方法执行insert语句<br/>  \* 第一个参数是sql语句<br/>  \* 第二个参数是sql对应的参数值，以Map形式传递 \*/  Map<String, Object> paramMap = new HashMap<>();  /\* map中的key必须要和sql语句的参数名值 \*/  paramMap.put("name", "国哥帅一次");  paramMap.put("salary", new BigDecimal(30000));  namedParameterJdbcTemplate.update(sql, paramMap); } |

以SqlParameterSource形式传入参数值

|  |
| --- |
| <!-- NamedParameterJdbcTemplate 类是专门用来解析执行 具体参数 的sql语句<br/> --> <bean id="namedParameterJdbcTemplate"  class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">  <!-- constructor-arg 是通过构造器参数赋值 -->  <constructor-arg index="0" ref="dataSource" /> </bean> |

|  |
| --- |
| @Autowired private NamedParameterJdbcTemplate namedParameterJdbcTemplate;  @Test public void test8() {  /\*:name 是占位符 ?（占位符就是传递的参数）  :name name就是这个参数的名称,:salary salary就是参数的名称\*/  String sql = "insert into employee(`name`,`salary`) values(:name,:salary)";  //update方法执行insert语句，第一个参数是sql语句，个参数是sql对应的参数值，以Map形式传递  Employee employee = new Employee(null, "国哥又帅一次", new BigDecimal(40000));  //BeanPropertySqlParameterSource类将javaBean中的属性和sql语句中的参数做一个赋值操作。  namedParameterJdbcTemplate.update(sql, new BeanPropertySqlParameterSource(employee)); } |

### queryObject

查询一条数据库记录，并封装为一个Java对象返回

|  |
| --- |
| public class Employee {  private Integer id;  private String name;  private BigDecimal salary; } |

|  |
| --- |
| @Test public void test4() {  String sql = "select id,name,salary from employee where id = ?";  /\* 第一个参数是sql语句，第二个参数是RowMapper，功能跟BeanHandler功能一样<br/>  \* RowMapper是每查询回来每一行记录转化为javaBean对象<br/>  \* BeanPropertyRowMapper类将查询回来的列和javaBean的属性建立赋值关系 \*/  Employee employee = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>(Employee.class), 5);  System.*out*.println(employee); } |

查询最大salary，因为返回的也是一条记录，所以使用的是queryForObject方法。

|  |
| --- |
| @Test public void test6() {  String sql = "select *max*(salary) from employee";  /\* 第一个参数是sql语句，第二个参数是返回的类型 \*/  BigDecimal maxSalary = jdbcTemplate.queryForObject(sql, BigDecimal.class);  System.*out*.println(maxSalary); } |

### query

查询数据库返回多条记录，并封装为List集合返回

|  |
| --- |
| @Test public void test5() {  String sql = "select id,name,salary from employee where salary > ?";  /\* update方法执行insert、delete、update的sql语句<br/>  \* queryForObject方法执行查询一行记录的sql语句<br/>  \* query方法执行查询返回多行记录的方法（多个javaBean） \*/  List<Employee> list = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>(Employee.class), new BigDecimal(4000));  for (Employee employee : list) {  System.*out*.println(employee);  } } |

创建Dao，自动装配JdbcTemplate对象

|  |
| --- |
| @Repository public class EmployeeDao {  @Autowired  JdbcTemplate jdbcTemplate;  public int saveEmployee(Employee employee) {  String sql = "insert into employee(`name`,`salary`) values(?,?)";  return jdbcTemplate.update(sql, employee.getName(), employee.getSalary());  } } |

|  |
| --- |
| @Autowired private EmployeeDao employeeDao;  @Test public void test9() {  employeeDao.saveEmployee(new Employee(null, "国哥帅第三次", new BigDecimal(50000))); } |

通过继承JdbcDaoSupport创建JdbcTemplateDao

|  |
| --- |
| @Repository public class EmployeeDao extends JdbcDaoSupport {  */\*****@Autowired*** *标注在方法上，此方法就会在bean对象创建之后马上调用做初始化工作  \* 并且这个方法中的参数spring会自动的从容器对象中查找到对应的对象实例传递赋值调用  \*/* @Autowired  public void init(DataSource dataSource) {  setDataSource(dataSource);  }  public int saveEmployee(Employee employee) {  String sql = "insert into employee(`name`,`salary`) values(?,?)";  return getJdbcTemplate().update(sql, employee.getName(), employee.getSalary());  } } |

|  |
| --- |
| @Autowired private EmployeeDao employeeDao;  @Test public void test10() {  employeeDao.saveEmployee(new Employee(null, "第三次", new BigDecimal(50000))); } |

## 声明式事务

事务分为声明式事务和编码式事务两种，声明式事务是指通过注解或XML配置文件的形式对事务的各种特性进行控制和管理；编程式事务是指通过编码的方式实现事务的声明。

**编码式事务实现实例**

|  |
| --- |
| Connection connection = JdbcUtils.getConnection();//获取数据库连接 try {  connection.setAutoCommit(false);//设置手动管理事务  //一系列数据库操作  connection.commit();//提交事务 } catch (Exception e) {  connection.rollback();//回滚事务  e.printStackTrace(); }finally {  connection.close();//关闭连接释放资源 } |

### 环境搭建

#### 数据库及数据库表

创建完数据库还要在applicationContext.xml配置文件中配置相关信息，否西Spring容器无法找到数据库和操作数据库的对象。

[详细数据库表见Appendix](#_环境搭建)的Spring的table\_2

#### Dao

|  |
| --- |
| @Repository public class BookDao {  @Autowired  JdbcTemplate jdbcTemplate;  public int updateBook() {  return jdbcTemplate.update("update `book` set `name` = '图书表被更新的';");  } }  @Repository public class UserDao {  @Autowired  JdbcTemplate jdbcTemplate;  public int updateUser() {  return jdbcTemplate.update("update `user` set `username` = '用户表被更新的';");  } } |

#### Service默认事务

|  |
| --- |
| @Service public class TransactionService {  @Autowired  private UserDao userDao;  @Autowired  private BookDao bookDao;  //在没有加入任何事务管理的情况下,默认的事务情况上执行一个sql一个事务  public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook();  } } |

#### 测试

|  |
| --- |
| @ContextConfiguration(locations = "classpath:applicationContext.xml") @RunWith(SpringJUnit4ClassRunner.class) public class SpringTest {  @Autowired  TransactionService transactionService;  @Test  public void testMultiUpdate() throws Exception {  transactionService.multiUpdate();  } } |

### 事务管理接口

PlatformTransactionManager是Spring框架提供用来统一管理事务的接口，其实现类DataSourceTransactionManger是使用DataSource数据库连接池技术访问操作数据库时用来管理事务的事务管理器。

DataSourceTransactionManger的方法与事务对应关系：

doBegin方法 前置通知 获取数据库连接和设置手动管理事务

doCommit方法 返回通知 提交事务

doRollback方法 异常通知 回滚事务

### 注解声明事务管理

使用Spring注解式事务管理时，可以简单的分为三个步骤：

1、配置事务管理器类DatasourceTransactionManager（数据库连接池版本）

2、配置事务的自动代理

3、在需要添加事务的方法上，使用@Transaction注解标注

在需要进行事务管理的方法上标注@Transactional注解声明对该方法进行管理，Spring框架底层设置@Transactional注解默认对运行时异常或其子异常进行回滚事务。配置事务管理器的唯一标识id时应将其设置为transactionManager，使之恰好与tx:annotation-driven的transaction-manager属性的默认值对应。

|  |
| --- |
| <!-- 配置事务管理器DataSourceTransactionManager  一般情况下，事务管理器的id值是：transactionManager --> <bean id="transactionManager"  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!-- DataSourceTransactionManager事务管理器里需要访问数据库使用的数据库连接池对象 -->  <property name="dataSource" ref="dataSource" /> </bean> <!-- tx:annotation-driven 标签表示支持注解@Transactional,并且会自动的实现代理功能 --> <!-- 有多个Spring的jar包中有该标签，注意导入tx的xmlns:tx="http://www.springframework.org/schema/tx"--> <!--配置事务管理器时设置的id为transactionManager，使之恰好与transaction-manager属性的默认值对应--> <tx:annotation-driven transaction-manager="transactionManager"/> |

|  |
| --- |
| @Transactional public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

#### 注解属性

Spring底层默认是对RuntimeException运行时异常或运行时异常的子异常进行回滚事务，可以通过设置属性来指定异常类型不再进行事务回滚。

noRollbackFor属性值为异常类的.class类型，指定异常类型不再默认事务回滚

|  |
| --- |
| @Transactional(noRollbackFor = ArithmeticException.class) public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

noRollbackForClassName属性值为异常类的全类名，指定异常类型不再默认事务回滚

|  |
| --- |
| @Transactional(noRollbackForClassName="java.lang.ArithmeticException") public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

rollbackFor属性值为异常类的.class类型，抛出指定异常类型时，进行事务回滚

|  |
| --- |
| @Transactional(rollbackFor = FileNotFoundException.class) public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

rollbackForClassName属性值为异常类的全类名，抛出指定异常类型时，进行事务回滚

|  |
| --- |
| @Transactional(rollbackForClassName="java.io.FileNotFoundException") public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

readOnly设置当前标注的方法是否允许之星SQL的写操作，如insert | delete | update等

timeout设置事务超时属性，指定事务在多长时间内必须完成SQL语句，以秒为单位

propagation设置事务的传播特性

### XML配置声明事务管理

使用XML配置声明式事务时就不需要和@Transactional注解和其相关的XML配置了。

|  |
| --- |
| <!-- 事务属性 --> <tx:advice id="tx\_advice">  <tx:attributes>  <!-- tx:method 给方法配置事务属性   name="updateUser" 精确匹配方法名  propagation="REQUIRED" 事物传播特性为必须REQUIRED。 -->  <tx:method name="multiTransaction" propagation="REQUIRED"/>  <tx:method name="updateUser" propagation="REQUIRED"/>  <!-- name="save\*" 半模糊匹配方法名，表示方法名必须以save打头  propagation="REQUIRED" 事物传播特性为必须REQUIRED。 -->  <tx:method name="save\*" propagation="REQUIRED"/>  <tx:method name="update\*" propagation="REQUIRES\_NEW"/>  <tx:method name="delete\*" propagation="REQUIRED"/>  <tx:method name="modify\*" propagation="REQUIRED"/>  <!-- name="\*"全模糊匹配方法名，表示以上匹配完之后剩下的所有方法   read-only="true" 表示当前方法只能是select查询   当有多个事务属性匹配上同一个方法名时，spring事务底层会做一些优化处理，匹配的优先规则是：  spring事务底层会做一些优化处理精确匹配 ====>>>> 半模糊 ===>>>> 全模糊 -->  <tx:method name="\*" read-only="true"/>  </tx:attributes> </tx:advice> <!-- 代理AOP，配置切面的配置信息(第一种方式)--> <aop:config>  <aop:pointcut id="pointcut1" expression="execution(public \* com.tx.service..\*Service\*.\*(..))"/>  <aop:advisor advice-ref="tx\_advice" pointcut-ref="pointcut1"/> </aop:config> <!-- 代理AOP，配置切面的配置信息(第二种方式)--> <aop:config>  <aop:advisor advice-ref="tx\_advice" pointcut="execution(public \* com.tx.service..\*Service\*.\*(..))" /> </aop:config> |

### 事务的传播特性

#### 概述

事务方法被另一个事务方法调用时，必须指定事务应该如何传播。方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。事务的传播行为可以由传播属性指定。Spring定义了7种类传播行为，其默认的事务传播特性值为REQUIRED。

|  |  |
| --- | --- |
| **传播属性** | **描述** |
| REQUIRED | 如果有事务在运行，就在这个事务内运行；否则就启动一个新的事务运行 |
| REQUIRED\_NEW | 必须启动新事务并运行，如果有事务正在运行，则将其挂起。 |
| SUPPORTS | 如果有事务在运行，就在这个事务内运行，否则不运行在事务中 |
| NOT\_SUPPORTS | 不运行在事务中，如果有事务正在运行，则将其挂起 |
| MANDATORY | 必须运行在事务内部，如果没有正在运行的事务，则抛出异常 |
| NEVER | 不运行在事务中，如果有事务正在运行，就抛出异常 |
| NESTED | 如果有事务在运行，就在这个事务的嵌套事务内运行，否则启动新事务并运行 |

#### 实例

大小事务传播特性都是REQUIRED

所有操作要么都成功，要么都失败，因为使用的是同一个事务。

|  |
| --- |
| @Transactional(propagation = Propagation.*REQUIRED*) public int updateUser() {。。。}  @Transactional(propagation = Propagation.*REQUIRED*) public int updateBook() {。。。}  @Transactional(propagation = Propagation.*REQUIRED*) public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

执行multiUpdate方法时开启新事务，获取连接；

执行updateUser方法时加入multiUpdate方法的事务，执行SQL语句；

执行updateBook方法时加入multiUpdate方法的事务，执行SQL语句；

multiUpdate方法结束时，提交事务，关闭连接，释放资源。

大小事务传播特性都是REQUIRES\_NEW

正常情况下，三个事务都成功；异常情况下，在异常产生之前提交的事务就成功，异常产生之后提交的事务都失败。

|  |
| --- |
| @Transactional(propagation = Propagation.*REQUIRES\_NEW*) public int updateUser() {。。。}  @Transactional(propagation = Propagation.*REQUIRES\_NEW*) public int updateBook() {。。。}  @Transactional(propagation = Propagation.*REQUIRES\_NEW*) public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

执行multiUpdate方法时开启新事务，获取连接；

执行updateUser方法时挂起multiUpdate方法的事务并开启新事务，获取连接执行SQL语句，提交事务关闭连接释放资源，恢复挂起的multiUpdate方法的事务；

执行updateBook方法时挂起multiUpdate方法的事务并开启新事务，获取连接执行SQL语句，提交事务关闭连接释放资源，恢复挂起的multiUpdate方法的事务；

multiUpdate方法结束时，提交事务，关闭连接，释放资源。

大事务是REQUIRED，小事务都是REQUIRES\_NEW

正常情况下，三个事务都成功；异常情况下，在异常产生之前提交的事务就成功，异常产生之后提交的事务都失败。

|  |
| --- |
| @Transactional(propagation = Propagation.*REQUIRES\_NEW*) public int updateUser() {。。。}  @Transactional(propagation = Propagation.*REQUIRES\_NEW*) public int updateBook() {。。。}  @Transactional(propagation = Propagation.*REQUIRED*) public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

执行multiUpdate方法时开启新事务，获取连接；

执行updateUser方法时挂起multiUpdate方法的事务并开启新事务，获取连接执行SQL语句，提交事务关闭连接释放资源，恢复挂起的multiUpdate方法的事务；

执行updateBook方法时挂起multiUpdate方法的事务并开启新事务，获取连接执行SQL语句，提交事务关闭连接释放资源，恢复挂起的multiUpdate方法的事务；

multiUpdate方法结束时，提交事务，关闭连接，释放资源。

大事务是REQUIRED，小1REQUIRED，小2REQUIRES\_NEW

正常情况下，三个事务都成功；异常情况下，在异常开始处，提交了的事务成功，没有提交的事务失败。

|  |
| --- |
| @Transactional(propagation = Propagation.*REQUIRED*) public int updateUser() {。。。}  @Transactional(propagation = Propagation.*REQUIRES\_NEW*) public int updateBook() {。。。}  @Transactional(propagation = Propagation.*REQUIRED*) public void multiUpdate() {  userDao.updateUser();  bookDao.updateBook(); } |

执行multiUpdate方法时开启新事务，获取连接；

执行updateUser方法时加入multiUpdate方法的事务，执行SQL语句；

执行updateBook方法时挂起multiUpdate方法的事务并开启新事务，获取连接执行SQL语句，提交事务关闭连接释放资源，恢复挂起的multiUpdate方法的事务；

multiUpdate方法结束时，提交事务，关闭连接，释放资源。

# Spring名称空间

## XML概述

　　XML 命名空间是由国际化资源标识符 (IRI) 标识的 XML 元素和属性集合；该集合通常称作 XML“词汇”。在XML中，元素名称是由开发者定义的，当两个不同的文档使用相同的元素名时，就会发生命名冲突。举个简单的栗子，命名空间很像 Java 中的包，不同的包下面可以存放相同的类名，只要在引入类时前面加上类的包就可以避免同名类的冲突。

命名空间被声明为元素的属性。并不一定只在根元素声明命名空间；而是可以在 XML 文档中的任何元素中进行声明。声明的命名空间的范围起始于声明该命名空间的元素，并应用于该元素的所有内容，直到被具有相同前缀名称的其他命名空间声明覆盖，其中，元素内容是指该元素的 <opening-tag> 和 </closing-tag> 之间的内容。

xmlns:类似于一个保留字，它只用于声明命名空间。换言之，xmlns 用于绑定命名空间，但其本身并不绑定到任何命名空间。

## Spring的XML配置

下述是spring的xml配置的最简单的格式

|  |
| --- |
| <?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"> </beans> |

1、在XML中，用xmlns或者xmlns：prefix指定命名空间

2、xmlns指定的命名空间是默认的命名空间，xmlns=”http://www.springframework.org/schema/beans”表示默认命名空间

3、xmlns：xsi这个命名空间的prefix为xsi，因此这个命名空间里面的元素或者属性就必须要以xsi:这种方式来写，比如schemaLocation就是他的一个属性，所以写成xsi:schemaLocation

4、xmlns是该XML文件默认的命名空间；

xmlns:xsi是指该XML文件遵守xml规范；

xsi:schemaLocation是指具体用到的schema资源

5、一般地，xml文件都需要写xmlns, xmlns:xsi, xsi:schemaLocation这3项。

## 引入名称空间

引入aop、util、context和tx名称空间，此时xsi:schemaLocation中xsd中的版本号可以省略。

|  |
| --- |
| <?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:util="http://www.springframework.org/schema/util"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans-4.0.xsd  http://www.springframework.org/schema/aop   http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/util   http://www.springframework.org/schema/util/spring-util-4.0.xsd  http://www.springframework.org/schema/context   http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/tx   http://www.springframework.org/schema/tx/spring-tx-4.0.xsd"> </beans> |

http://www.springframework.org/schema/aop表示的是名称空间

http://www.springframework.org/schema/aop/spring-aop-4.0.xsd表示的是该名称空间的位置（Spring源码包下，schema下，aop下的spring-aop-4.0.xsd文件），在aop下有不同版本的xsd文件。

aop:这里实际上是将前缀“aop”与命名空间"http://www.springframework.org/schema/aop"(这个URI包含关于命名空间的信息)绑定在一起。通常我们会用一个比较简短或约定俗成的名字来作为命名空间的前缀（例如这里的aop），但具体使用什么前缀完全取决于个人.自定义命名空间的前缀是合法的。使用有意义的命名空间前缀增强了XML档的清晰性。所以可以看到我们平时在配置Spring配置文件的时候，前缀名都是aop（切面）、tx（事务）等命名方式。

### 单个默认名称空间

我们看到，在配置文件中，beans，bean等元素我们是没有使用命名空间前缀的。重复限定一个要在命名空间中使用的元素或属性可能会非常麻烦。

这种情况下，可以声明一个 默认命名空间。无论在任何时候都只能存在一个默认命名空间。

声明一个 默认命名空间 意味着，如果 默认命名空间 声明范围内的任何元素未使用前缀显式限定，则该元素将被隐式限定。与带前缀的命名空间一样，

默认命名空间 也可以被覆盖。也就是说，在命名空间范围内，不带有前缀的元素都是在这个命名空间内的，例如这里的<beans> <bean>等，因为比较常用所以就让声明他们的命名空间为默认的啦，不用每次写都带前缀。

### 配置名称空间

spring 整合了各种工具，并且spring提供了对各种工具的xml scheme 的配置方式，简化了开发。对于各种工具的xml命名空间的引入，我们也应该有一个比较清楚的认识。Spring在启动时是要检验XML文件的。如果xml空间存在命名空间内没有的元素是要报错的。通常情况下，命名空间对应的URI是一个存放XSD的地址，尽管规范没有这么要求。如果没有提供schemaLocation，那么Spring的XML解析器会从命名空间的URI里加载XSD文件。

xmlns="http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"则默认的命名空间就是加载指定的xsd文件。

schemaLocation提供了一个xml 命名空间到对应的XSD(Xml Schema Definition)文件的一个映射，它的值由一个或多个URI引用对组成，在xsi:schemaLocation后面配置的字符串都是成对的，前面的是命名空间的URI，后面是xsd文件的URI；

两个URI之间以空白符分隔（空格和换行均可）。第一个URI是定义的 XML命名空间的值，第二个URI给出Schema文档的实际位置，

Schema处理器将从这个位置读取Schema文档，该文档的targetNamespace必须与第一个URI(XML命名空间的值)相匹配。

### 校验XML配置文件

Spring默认在启动时是要从配置的命名空间的位置加载XSD文件来验证xml文件的，所以如果有的时候断网了，或者一些开源软件切换域名，那么就很容易碰到应用启动不了。

为了防止这种情况，Spring提供了一种机制，即默认从本地加载XSD文件，当本地没有时才根据实际的URI去联网获得。

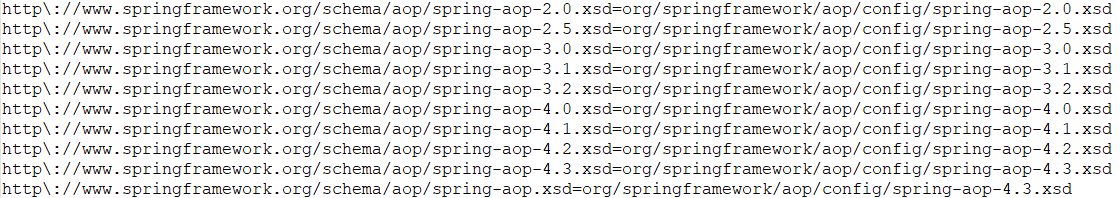
我们打开Spring-aop-4.1.6RELEASE.jar （这是我本地的版本），这个包下有一个META\_INF文件夹，其中有两个文件：spring.handlers和spring.schemas。



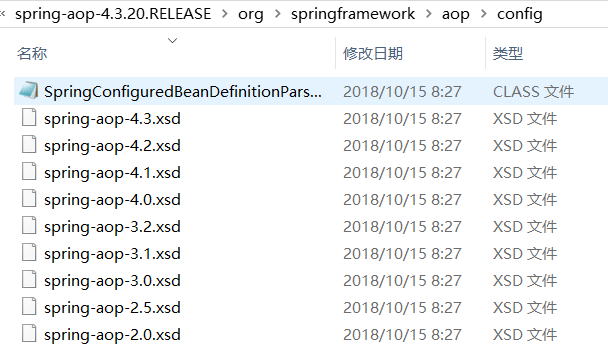
**spring.handlers**



**spring.schemas**



在org/springframework/aop/config目录下可以看到对应的本地xsd文件



这就很明显，Spring是把XSD文件放到本地了，再在spring.schemas里做了一个映射，优先从本地里加载XSD文件。

并且把spring旧版本的XSD文件也全放了。这样可以防止升级了Spring版本，而配置文件里用的还是旧版本的XSD文件，然后断网了，应用启动不了。

注意我在spring.schemas中标红的最后一行，说明我们在写命名空间值对应的xsd文件位置时，可以不用写版本号，它默认的是本地spring相关版本的对应xsd版本，我这里是4.1。

在xsi:schemaLocation中这样写：http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd

# Spring常用类

## Core包

### StopWatch

 StopWatch是位于org.springframework.util包下的一个工具类，通过它可方便的对程序部分代码进行计时(ms级别)，适用于同步单线程代码块。

**优点：**

spring自带工具类，可直接使用；代码实现简单，使用更简单；统一归纳，展示每项任务耗时与占用总时间的百分比，展示结果直观；性能消耗相对较小，并且最大程度的保证了start与stop之间的时间记录的准确性；可在start时直接指定任务名字，从而更加直观的显示记录结果。

**缺点：**

一个StopWatch实例一次只能开启一个task，不能同时start多个task，并且在该task未stop之前不能start一个新的task，必须在该task stop之后才能开启新的task，若要一次开启多个，需要new不同的StopWatch实例；代码侵入式使用，需要改动多处代码。

|  |
| --- |
| public class NormalTest {  @Test  public void test(){  StopWatch stopWatch = new StopWatch();  stopWatch.start("task1");  doSomething();  stopWatch.stop();  stopWatch.start("task2");  doSomething();  stopWatch.stop();  String print = stopWatch.prettyPrint();  System.*out*.println(print);  }  public void doSomething(){  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  } } |

# Spring和Web整合

Spring和Web工程的整合分两步

1、导入相关的jar包

2、配置web.xml配置文件

**web.xml配置文件**

1、在web.xml配置文件中配置org.springframework.web.context.ContextLoaderListener监听器监听ServletContext的初始化

2、在web.xml配置文件中配置contextConfigLocation上下文参数。配置Spring配置文件的位置，以用于初始化Spring容器

|  |
| --- |
| <servlet>  <servlet-name>SpringTestServlet</servlet-name>  <servlet-class>com.xmm.servlet.SpringTestServlet</servlet-class> </servlet> <servlet-mapping>  <servlet-name>SpringTestServlet</servlet-name>  <url-pattern>/springTestServlet</url-pattern> </servlet-mapping> <!-- 配置上下文参数， 提供Spring配置文件的路径 --> <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value> </context-param> <!-- 配置监听器，监听ServletContext对象的创建，和销毁。  在创建ServletContext对象的时候，创建spring的容器对象  在销毁ServletContext对象的时候，关闭Spring容器。 --> <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener> |

## 获取WebApplicationContext上下文对象

### 方法一（推荐）

方法一本质上是对方法二的封装，提高了程序的兼容性，避免了因为Spring框架升级导致常量ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE的修改而导致旧程序错误。

|  |
| --- |
| ApplicationContext applicationContext = WebApplicationContextUtils.*getWebApplicationContext*(getServletContext()); |

### 方法二（不推荐）

ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE是接口WebApplicationContext中的String类型的常量，用于获取ApplicationContext的对象。

|  |
| --- |
| ApplicationContext applicationContext = (ApplicationContext) getServletContext().getAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*); |

# SSM

## SSM配置

### 导入jar包

首先创建动态web工程，然后导入ssm需要的jar包，如下所示：

**Spring的核心包**

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

commons-logging-1.1.3.jar

**Spring的切面包**

com.springsource.org.aopalliance-1.0.0.jar

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

spring-aop-4.0.0.RELEASE.jar

spring-aspects-4.0.0.RELEASE.jar

**log4j日记包**

log4j-1.2.17.jar

**mysql驱动和数据库连接池包**

druid-1.19.jar

mysql-connector-java-5.1.37-bin.jar

**Spring的数据库及事务包**

spring-jdbc-4.0.0.RELEASE.jar

spring-orm-4.0.0.RELEASE.jar

spring-tx-4.0.0.RELEASE.jar

**SpringMVC的包**

spring-web-4.0.0.RELEASE.jar

spring-webmvc-4.0.0.RELEASE.jar

**文件上传包**

commons-fileupload-1.2.1.jar

commons-io-1.4.jar

**Spring中的Json处理包**

jackson-annotations-2.1.5.jar

jackson-core-2.1.5.jar

jackson-databind-2.1.5.jar

**MyBatis以及整合包**

mybatis-3.4.1.jar

mybatis-spring-1.3.0.jar

**JSTL标签库**

taglibs-standard-impl-1.2.1.jar

taglibs-standard-spec-1.2.1.jar

### 创建数据库

drop database if exists springmvc;  
create database springmvc;  
use springmvc;  
##创建图书表  
create table t\_book  
(  
 `id` int(11) primary key auto\_increment, ## 主键  
 `name` varchar(50) not null, ## 书名   
 `author` varchar(50) not null, ## 作者  
 `price` decimal(11, 2) not null, ## 价格  
 `sales` int(11) not null, ## 销量  
 `stock` int(11) not null ## 库存  
);  
## 插入初始化测试数据  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, 'java从入门到放弃', '国哥', 80, 9999, 9);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '数据结构与算法', '严敏君', 78.5, 6, 13);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, 'C语言程序设计', '谭浩强', 28, 52, 74);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, 'Lua语言程序设计', '雷丰阳', 51.5, 48, 82);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '西游记', '罗贯中', 12, 19, 9999);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '水浒传', '施耐庵', 33.05, 22, 88);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '操作系统原理', '刘优', 133.05, 122, 188);  
## 查看表内容  
select id, name, author, price, sales, stock  
from t\_book;

配置数据库配置文件jdbc.properties

|  |
| --- |
| user=root password=root url=jdbc:mysql://localhost:3306/ssm driverClassName=com.mysql.jdbc.Driver initialSize=5 maxActive=10 |

### 配置文件

配置Mybatis核心配置文件mybatis-config.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration>  <settings>  <!-- 打开延迟加载的开关 -->  <setting name="lazyLoadingEnabled" value="true" />  <!-- 将积极加载改为消极加载 按需加载 -->  <setting name="aggressiveLazyLoading" value="false" />  <!-- 表示允许使用二级缓存，打开二缓存。 -->  <setting name="cacheEnabled" value="true"/>  </settings>  <!-- mappers标签用来引入sql的配置文件 -->  <mappers>  <package name="com.ssm.mapper" />  </mappers> </configuration> |

配置SpringMVC配置文件springmvc.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd  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">  <!-- SpringMVC的配置文件，只配置SpringMVC需要的组件。  它不用的组件都交给Spring配置文件去配置管理 -->  <context:component-scan base-package="com" use-default-filters="false">  <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  <context:include-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan>  <!-- SpringMVC标配 -->  <mvc:default-servlet-handler/>  <mvc:annotation-driven/> </beans> |

配置Spring核心配置文件applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:mybatis-spring="http://mybatis.org/schema/mybatis-spring"  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-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd http://mybatis.org/schema/mybatis-spring http://mybatis.org/schema/mybatis-spring.xsd">  <!-- 配置包扫描，只扫描除了SpringMVC之外的组件 -->  <context:component-scan base-package="com">  <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  <context:exclude-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan>   <!-- 加载jdbc.properties属性配置文件 -->  <context:property-placeholder location="classpath:jdbc.properties" />  <!-- 配置数据库连接池对象 -->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="username" value="${user}" />  <property name="password" value="${password}" />  <property name="url" value="${url}" />  <property name="driverClassName" value="${driverClassName}" />  <property name="initialSize" value="${initialSize}" />  <property name="maxActive" value="${maxActive}" />  </bean>   <!-- 配置事务管理器 -->  <bean id="transactionManager"  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource" />  </bean>  <!-- 配置事务属性 -->  <tx:advice id="tx\_ssm" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="add\*" propagation="REQUIRED" />  <tx:method name="save\*" propagation="REQUIRED" />  <tx:method name="update\*" propagation="REQUIRED" />  <tx:method name="delete\*" propagation="REQUIRED" />  <tx:method name="\*" read-only="true" />  </tx:attributes>  </tx:advice>  <!-- 配置事务切面 -->  <aop:config>  <aop:advisor advice-ref="tx\_ssm"  pointcut="execution(public \* com..service..\*.\*(..))" />  </aop:config>   <!-- Mybatis整合Spring的核心配置之一 -->  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  <property name="dataSource" ref="dataSource" />  <property name="configLocation" value="classpath:mybatis-config.xml" />  </bean>  <!-- Mybatis整合Spring的核心配置 扫描并将Mapper接口注入到SpringIOC容器中 -->  <mybatis-spring:scan base-package="com.ssm.mapper"/> </beans> |

# 数据库表

ex\_1

|  |
| --- |
| drop database if exists jdbctemplate; create database jdbctemplate; use jdbctemplate; create table `employee` (`id` int(11) primary key auto\_increment,  `name` varchar(100) default null,  `salary` decimal(11, 2) default null); insert into `employee`(`id`, `name`, `salary`) values (1, '李三', 5000.23),(2, '李四', 4234.77),  (3, '王五', 9034.51),(4, '赵六', 8054.33),  (5, '孔七', 6039.11), (6, '曹八', 7714.11); select \*from employee; |

ex\_2

|  |
| --- |
| drop database if exists `tx`; CREATE database `tx`; USE `tx`; DROP TABLE IF EXISTS `user`; CREATE TABLE `user`(`id` int primary key auto\_increment, `username` varchar(50) NOT NULL,`money` int(11) DEFAULT NULL); insert into `user`(`username`, `money`) values ('张三', 1000),('李四', 1000); drop table if exists `book`; create table `book`(`id` int primary key auto\_increment,`name` varchar(500) not null,`stock` int); insert into book(`name`, `stock`) values ('java编程思想', 100),('C++编程思想', 100); select \* from `book`; select \* from `user`; |