Spring第三天

**今天知识点：**

Spring的传统AOP编程的案例（计算方法的运行时间）

AspectJ的集成配置、JdbcTemplate工具类。

**今天的主要内容（AOP）：**

1. Spring的传统AOP编程的案例（计算方法的运行时间）
2. Spring结合AspectJ实现AOP编程（XML和注解）
3. JdbcTemplate编程（连接池的配置(传统连接池)、外部属性文件的引入、实现DAO的CRUD操作（快捷使用模版类的方法dao类））

**学习目标：**

1. aop的编程（传统xml——要会，aspectj注解——掌握）
2. 其他：外部属性文件引入、连接池、使用jdbctemplate操作数据库（jdbctemplateDAO）

# 传统Spring AOP编程案例

传统SpringAOP 是指 1.2版本之后开始支持AOP编程 。

提示：

老的AOP的编程配置过于复杂，这里采用AspectJ的切入点语法来讲解。

面向切面编程开发步骤(动态织入)

1、确定目标对象（target—>bean）

2、编写Advice通知方法 （增强代码）

3、配置切入点和切面

直接使用 CustomerService（需要接口）和ProductService（不需要接口）作为 target目标对象

提示：spring的所有功能，都是基于bean的，下面所说的“目标”，都是bean。

## 传统SpringAOP的Advice编写（了解）

传统Spring AOP的通知（增强）种类：

* AOP联盟为通知Advice定义了org.aopalliance.aop.Interface.Advice
* Spring按照通知Advice在目标类方法的连接点位置，可以分为5类

（1）前置通知 org.springframework.aop.MethodBeforeAdvice

\* 在目标方法执行前实施增强

（2）后置通知 org.springframework.aop.AfterReturningAdvice

\* 在目标方法执行后实施增强

（3）环绕通知 org.aopalliance.intercept.MethodInterceptor

\* 在目标方法执行前后实施增强

（4）异常抛出通知 org.springframework.aop.ThrowsAdvice

\* 在方法抛出异常后实施增强

（5）引介通知 org.springframework.aop.IntroductionInterceptor

\* 在目标类中添加一些新的方法和属性

简单的说：通知就是增强的方式方法

遵循aop联盟规范，传统Spring AOP编程的Advice有五种（前置通知、后置通知、环绕通知、异常通知、引介通知） ，

传统SpringAOP的Advice 必须实现对应的接口！

【需求】：开发一个记录方法运行时间的例子。将目标方法的运行时间，写入到log4j的日志中。

开发步骤：

第一步：引入所需依赖:

<!-- spring核心依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!—- springaspect集成 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<!-- 日志 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<!-- spring集成测试 -->

<dependency>

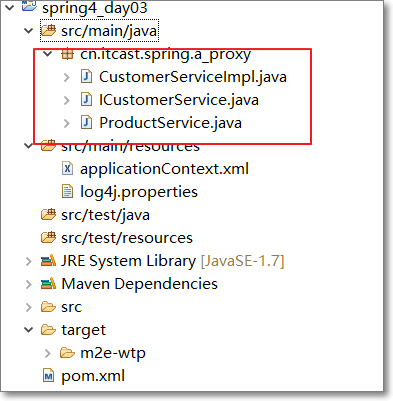
<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.13.RELEASE</version>

</dependency>

使用动态代理的代码:



第二步：编写传统aop的Advice通知类。

创建包：cn.itcast.spring.b\_oldaop，创建类TimeLogInterceptor 。

传统aop的通知，必须实现MethodInterceptor接口

**import** org.aopalliance.intercept.MethodInterceptor;

**import** org.aopalliance.intercept.MethodInvocation;

**import** org.apache.log4j.Logger;

//传统的aop的advice通知，增强类，必须实现org.aopalliance.intercept.MethodInterceptor接口（注意和cglib代理接口区分开）

**public** **class** TimeLogInterceptor **implements** MethodInterceptor {

//回调方法

//参数：目标方法回调函数的包装类，获取调用方法的相关属性、方法名、调用该方法的对象（即封装方法、目标对象，方法的参数）

**public** Object invoke(MethodInvocation methodInvocation) **throws** Throwable {

//业务：记录目标的方法的运行时间

//方法调用之前记录时间

**long** beginTime = System.*currentTimeMillis*();

//目标对象原来的方法的调用，返回目标对象方法的返回值。

Object object = methodInvocation.proceed();//类似于invoke

//方法调用之后记录时间

**long** endTime = System.*currentTimeMillis*();

//计算运行时间

**long** runTime=endTime-beginTime;

//写日志：

*LOG*.info("方法名为："+methodInvocation.getMethod().getName()+"的运行时间为："+runTime+"毫秒");

**return** object;

}

}

第三步：核心配置文件中，创建applicationContext.xml文件：（确定目标和配置通知），仍然使用CustomerServiceImpl和ProductService进行测试。

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类（没有实现接口的类） -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置增强：原则bean能增强bean

Advice:通知，增强

-->

<bean id=*"timeLogAdvice"* class=*"cn.itcast.spring.b\_oldaop.TimeLogInterceptor"*/>

第四步：使用SpringTest进行测试

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

第六步：查看执行结果：



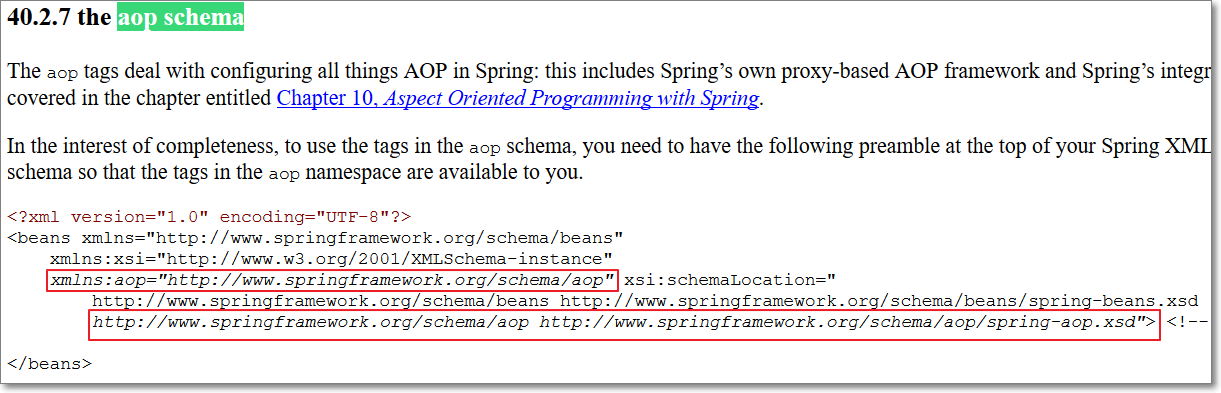
但是发现，此时并没有执行TimeLogInterceptor 类的invoke()方法，也就是说，并没有计算执行Service类的时间，那怎么办呢？我们往下看，需要在spring容器中配置spring的aop。

## 配置切入点和切面

目的：让哪个类（切面）、哪个方法（切入点），进行怎样的增强（通知）。

第一步： 引用aop的名称空间

查看：/课前资料/xsd-configuration.html 搜素aop，发现：



需要我们引入aop的文件约束。

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

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

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

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

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

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

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

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

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

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

第二步：配置切入点，让通知关联切入点

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

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

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

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

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

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

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

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

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

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

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类（没有实现接口的类） -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置增强：

Advice:通知，增强

-->

<bean id=*"timeLogAdvice"* class=*"cn.itcast.spring.b\_oldaop.TimeLogInterceptor"*/>

<!-- 3.配置切入点和切面 :aop:config-->

<aop:config>

<!--

配置切入点：即你要拦截的哪些 连接点（方法）

\* expression：表达式：匹配方法的，语法：使用aspectj的语法，相对简单

\* 表达式：bean（bean的名字），你要对哪些bean中的所有方法增强

\* expression=bean(\*Service)：在spring容器中，所有id/name以Service单词结尾的bean的都能被拦截

\* id="myPointcut"：为切入点定义唯一标识

-->

<aop:pointcut expression="bean(\*Service)" id="myPointcut"/>

<!--

advice-ref="timeLogAdvice"

\* 配置切面:通知（增强的方法）关联切入点（目标对象调用的方法）

pointcut-ref="myPointcut"

\* 告诉：你要对哪些方法（pointcut），进行怎强的增强 （advice）

-->

<aop:advisor advice-ref="timeLogAdvice" pointcut-ref="myPointcut"/>

</aop:config>

</beans>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

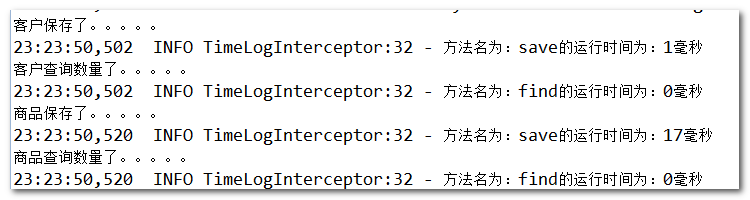
productService.save();

productService.find();

}

}

第四步：查看结果：



切入点表达式的语法整理如下：

* bean(bean Id/bean name)

例如 bean(customerService) 增强spring容器中定义id属性/name属性为customerService的bean中所有方法

* execution(<访问修饰符>?<返回类型>空格<方法名>(<参数>)<异常>?)

例如：

execution(\* cn.itcast.spring.a\_jdkproxy.CustomerServiceImpl.\*(..)) 增强bean对象所有方法

execution(\* cn.itcast.spring..\*.\*(..)) 增强spring包和子包所有bean所有方法

提示：最灵活的

参考文档：spring4\_day1\_课前资料/Spring2.5-中文参考手册.chm



案例应用：

切入点的表达式

execution(void cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

\* 表示：无返回类型，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

表达式的写法

execution（modifiers-pattern? （非必填项）--<访问修饰符>?

ret-type-pattern （必填项）--<返回类型>

declaring-type-pattern? （非必填项）

name-pattern（param-pattern）（必填项）--<方法名>(<参数>)

throws-pattern?（非必填项）<异常>?

）

一共有5个参数

其中的?表示非必填项

文档中写的：

除了返回类型模式（上面代码片断中的ret-type-pattern），名字模式和参数模式以外， 所有的部分都是可选的。

返回类型模式决定了方法的返回类型必须依次匹配一个连接点。 你会使用的最频繁的返回类型模式是\*，它代表了匹配任意的返回类型。

一个全限定的类型名将只会匹配返回给定类型的方法。名字模式匹配的是方法名。 你可以使用\*通配符作为所有或者部分命名模式。

参数模式稍微有点复杂：()匹配了一个不接受任何参数的方法， 而(..)匹配了一个接受任意数量参数的方法（零或者更多）。

模式(\*)匹配了一个接受一个任何类型的参数的方法。 模式(\*,String)匹配了一个接受两个参数的方法，第一个可以是任意类型， 第二个则必须是String类型。更多的信息请参阅AspectJ编程指南中 语言语义的部分。

1：modifiers-pattern? （非必填项）：表示方法的修饰符

execution(public void cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

\* 表示：共有方法，无返回类型，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(private void cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

\* 表示：私有方法，无返回类型，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

2：ret-type-pattern （必填项）：表示方法的返回类型

execution(**void** cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

\* 表示：无返回类型，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(**java.lang.String** cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

\* 表示：返回类型String类型，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(**\*** cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

\* 表示：返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

3：declaring-type-pattern? （非必填项）：表示包，或者子包的，或者类的修饰符

execution(\* **cn.itcast.e\_xml.a\_before.UserServiceImpl**.saveUser(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(\* **cn.itcast.e\_xml.\*.UserServiceImpl**.saveUser(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml包中的所有子包，包中UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(\* **cn.itcast.e\_xml.\***.saveUser(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml包中的所有类，类中的saveUser方法，参数2个，都是String类型

execution(\* **cn.itcast.e\_xml..\***.saveUser(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml包中及其子包中的所有类，类中的saveUser方法，参数2个，都是String类型

execution(\* **\***.saveUser(java.lang.String,java.lang.String))

\* 表示返回类型任意，所有包中的所有类，类中的saveUser方法，参数2个，都是String类型

execution(\* saveUser(java.lang.String,java.lang.String))

\* 表示返回类型任意，所有包中的所有类，类中的saveUser方法，参数2个，都是String类型

4：name-pattern（param-pattern）（必填项）：方法的名称（方法的参数）

（1）方法名称

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.**saveUser**(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.**save\***(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的以save开头的方法，参数2个，都是String类型

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.**\***(java.lang.String,java.lang.String))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的所有方法，参数2个，都是String类型

（2）方法的参数

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(**java.lang.String,java.lang.String**))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，都是String类型

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(**java.lang.String,java.lang.Integer**))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，参数1是String类型，参数二是Integer

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(**java.lang.String,\***))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数2个，参数1是String类型，参数二是任意类型

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(**\***))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数1个，参数是任意类型

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser())

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，没有参数

execution(\* cn.itcast.e\_xml.a\_before.UserServiceImpl.saveUser(**..**))

\* 表示返回类型任意，cn.itcast.e\_xml.a\_before包中的UserServiceImpl类，类中的saveUser方法，参数任意（可以是0个，也可以多个）

5：throws-pattern?（非必填项）：方法上抛出的异常

项目开发中表达式（最多用）

1：execution(\* cn.itcast.procject.service..\*.\*(..))

\* 返回类型任意，cn.itcast.procject.service包及其子包中所有类，类中所有方法，参数任意

2：execution(\* \*..\*.\*(..))

\* 返回类型任意，任意包中及其子包中所有类，类中所有方法，参数任意

3：execution(\* \*(..))

\* 返回类型任意，任意包中及其子包中所有类，类中所有方法，参数任意

下面给出一些通用切入点表达式的例子。

任意公共方法的执行：

execution（public \* \*（..））

任何一个名字以“set”开始的方法的执行：

execution（\* set\*（..））

AccountService接口定义的任意方法的执行：

execution（\* com.xyz.service.AccountService.\*（..））

在service包中定义的任意方法的执行：

execution（\* com.xyz.service.\*.\*（..））

在service包或其子包中定义的任意方法的执行：

execution（\* com.xyz.service..\*.\*（..））

* within(包.类)

例如： within(cn.itcast.spring..\*) 增强spring包和子包所有bean“所有方法 ”

* this(完整类型)/target(完整类型)

范围最小，只针对某个类型。

this对某一个类-（对代理对象有效），target对代理对象无效(只对目标对象有效)

例如： this(cn.itcast.spring.a\_jdkproxy.CustomerServiceImpl) 增强类型所有方法（对代理对象有效）

target(cn.itcast.spring.a\_jdkproxy.CustomerServiceImpl)增强类型所有方法（对目标对象有效）

注意：我们一般都对目标对象进行拦截，很少对代理对象进行拦截

【AspectJ类型匹配的通配符】

\*：匹配任何数量字符（一个）；

..：匹配任何数量字符的重复（多个），如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

【测试】：在applicationContext.xml文件中，测试切入点表达式的写法：

<!-- 3.配置切入点和切面 :aop:config-->

<aop:config>

<!-- <aop:pointcut expression="**bean**(\*Service)" id="myPointcut"/> -->

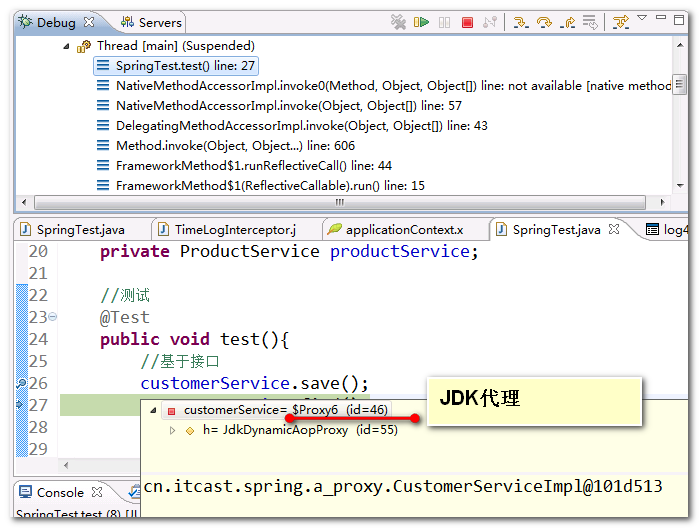
<aop:pointcut expression="**execution**(\* cn.itcast.spring..\*.\*(..))" id="myPointcut"/>

<aop:advisor advice-ref=*"timeLogAdvice"* pointcut-ref=*"myPointcut"*/>

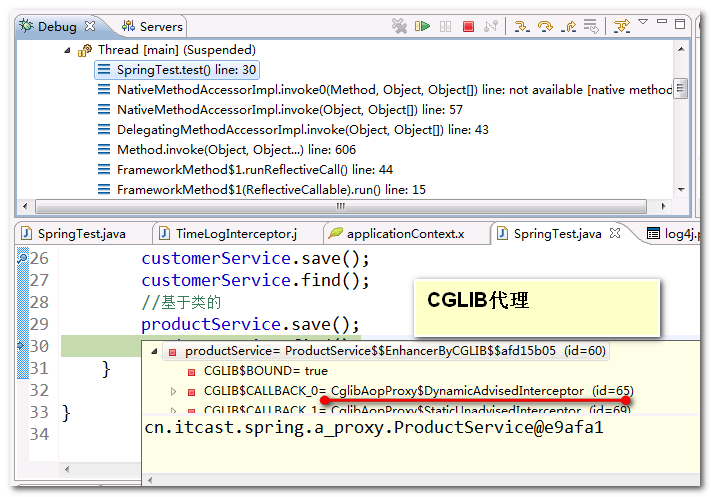
</aop:config>

【补充】：事实上，当运行的时候，两个Service已经是代理对象了。如：使用Debug断点进行调试

（1）有接口的customerService类：（jdk动态代理）



（2）没有接口的productService：（cglib动态代理）



# AspectJ 切面编程（xml方式）

Xml配置aop开发方法还是三步：

1. 确定目标对象（bean）
2. 编写通知，对目标对象增强（advice）
3. 配置切入点（pointcut）、切面（aspect）

## AspectJ 提供Advice类型

普通的pojo即可。（不需要实现接口）

AspectJ提供不同的通知类型：

* Before 前置通知，相当于BeforeAdvice
* AfterReturning 后置通知，相当于AfterReturningAdvice
* Around 环绕通知，相当于MethodInterceptor
* AfterThrowing抛出通知，相当于ThrowAdvice
* After 最终final通知，不管是否异常，该通知都会执行
* DeclareParents 引介通知，相当于IntroductionInterceptor (不要求掌握)

相比传统Spring AOP通知类型多了 After最终通知 （类似 finally ）。

实现步骤：

第一步：确定目标对象，即确定bean对象

第二步：advice通知（编写）

第三步：配置切面（包括切入点），让切入点关联通知

第一步：确定目标对象，即确定bean对象：

在src下，创建applicationContext-aspect.xml。

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

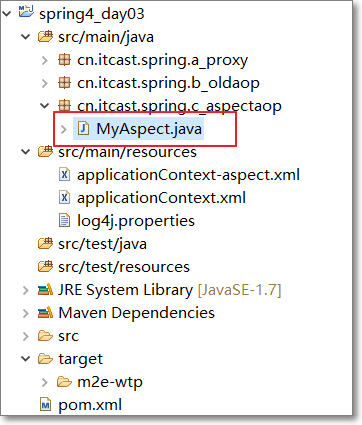
<!-- 基于一般类 -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

第二步：编写Before 前置通知Advice增强 ：

创建包：cn.itcast.spring.c\_aspectaop

创建类：MyAspect.java



编写MyAspect.java

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//前置通知

//普通的方法。方法名随便，但也不能太随便，一会要在applicationContext.xml中配置

**public** **void** firstbefore(){

System.*out*.println("------------第一个前置通知执行了。。。");

}

}

将前置通知配置到spring的容器中

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

## 配置切入点和切面（让切入点关联通知）

第三步：配置切面（包括切入点），让切入点关联通知

核心配置文件applicationContext-aspect.xml中添加：

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<!-- 切入点:拦截哪些bean的方法 -->

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!--

切面：要对哪些方法进行怎样的增强

aop:aspect:aspejctj的方式！

ref:配置通知

-->

<aop:aspect ref=*"myAspectAdvice"*>

<!-- 第一个前置通知 ：在访问目标对象方法之前，先执行通知的方法

method：advice类中的方法名，

pointcut-ref="myPointcut"：注入切入点

目的是让通知关联切入点

-->

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

</aop:aspect>

</aop:config>

使用SpringTest测试代码：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext-aspect.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

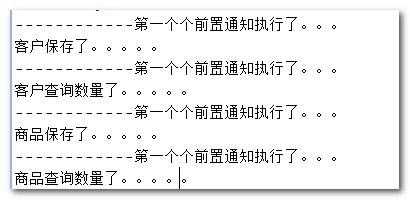
productService.save();

productService.find();

}

}

测试结果：



和传统的aop配置相比，更灵活，advice不需要实现接口，简单的pojo就可以了；一个通知可以增强多个连接点，一个连接点可以被多次增强。

【扩展优化】：

1．将切入点放入aspect标签里面写，同时配置多个通知方法

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<!--

切面：要对哪些方法进行怎样的增强

aop:aspect:aspejctj的方式！

ref:配置通知

-->

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 第一个前置通知 ：在访问目标对象方法之前，先执行通知的方法

method：advice类中的方法名，

pointcut-ref="myPointcut"：注入切入点

目的是让通知关联切入点

-->

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

<aop:before method=*"firstbefore2"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

2.配置多个通知方法：

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//前置通知

//普通的方法。方法名随便，但也不能太随便，一会要配置

**public** **void** firstbefore(){

System.*out*.println("------------第一个前置通知执行了。。。");

}

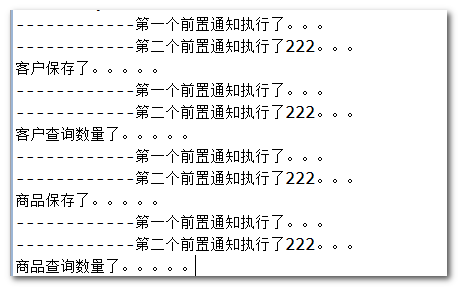
**public** **void** firstbefore2(){

System.*out*.println("------------第二个前置通知执行了222。。。");

}

}

1. 执行结果：表示在执行目标对象方法之前执行



AspectJ切面编程，相比于传统的SpringAOP，定义的通知方法更多。

## 分析各种通知应用

### Before前置通知

案例应用： 实现权限控制 （即：权限不足的时候，抛出异常）、 记录方法调用信息日志

第一步：配置MyAspect类（切面），配置before方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//前置通知的：方法运行之前增强

//应用： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志

//参数：org.aspectj.lang.JoinPoint

//参数：连接点对象（方法的包装对象:方法，参数，目标对象）

**public** **void** before(JoinPoint joinPoint){

//分析：抛出异常拦截的

//当前登录用户

String loginName = "Rose";

System.*out*.println("方法名称："+joinPoint.getSignature().getName());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass().getName());

System.*out*.println("代理对象："+joinPoint.getThis().getClass().getName());

//判断当前用户有没有执行方法权限

**if**(joinPoint.getSignature().getName().equals("save")){

**if**(!loginName.equals("admin")){

//只有超级管理员admin有权限，其他人不能执行某个方法，比如查询方法

**throw** **new** RuntimeException("您没有权限执行方法："+joinPoint.getSignature().getName()+"，类型为："+joinPoint.getTarget().getClass().getName());

}

}

}

}

通过JoinPoint 连接点对象，获取目标对象信息 ！

这里注意：引包不要引错了，使用aspectj中的连接点（org.aspectj.lang.JoinPoint）：



第二步：Spring容器中配置，配置applicationContext-aspect.xml

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类的 -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 前置通知 -->

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

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext-aspect.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

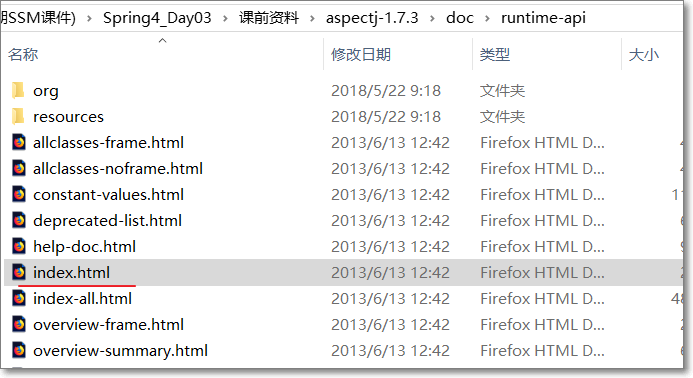
}

}

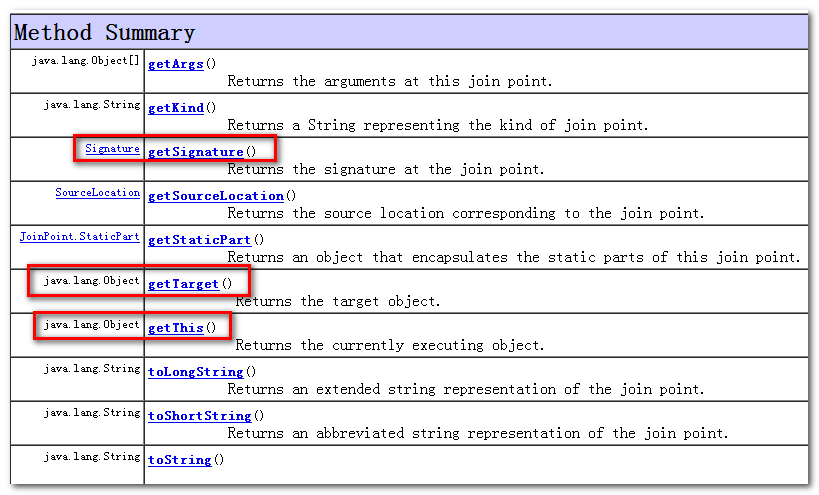
提示：

Aspectj的文档

JoinPoint使用参考文档：\课前资料\aspectj-1.7.3\doc\runtime-api\index.html







### AfterReturing 后置通知

特点：在目标方法运行后，返回值后执行通知增强代码逻辑。

应用场景：与业务相关的，如网上营业厅查询余额后，自动下发短信功能。

分析： 后置通知可以获取到目标方法返回值，如果想对返回值进行操作，使用后置通知（但不能修改目标方法返回 ）

第一步：配置MyAspect类（切面），配置afterReturing方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//应用场景：与业务相关的，如网上营业厅查询余额后，自动下发短信。

//后置通知：会在目标方法执行之后调用通知方法增强。

//参数1：连接点对象（方法的包装对象:方法，参数，目标对象）

//参数2：目标方法执行后的返回值,类型是object，“参数名”随便，但也不能太随便，一会要配置

**public** **void** afterReturing(JoinPoint joinPoint,Object returnVal){

//下发短信:调用运行商的接口，短信猫。。。

System.*out*.println("-++++++++-后置通知-当前下发短信的方法"+"-尊敬的用户，您调用的方法返回余额为："+returnVal);

}

}

第二步：Spring容器中配置，配置applicationContext-aspect.xml

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类的 -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 后置通知

returning:配置方法中的参数名字，与通知方法的第二个参数的名字，名字必须对应。

在运行的时候，spring会自动将返回值传入该参数中。

-->

<aop:after-returning method=*"afterReturing"* returning=*"returnVal"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext-aspect.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

### Around 环绕通知

特点：目标执行前后，都进行增强（控制目标方法执行）

应用场景：日志、缓存、权限、性能监控、**事务管理**

**增强代码的方法要求：**

接受的参数：ProceedingJoinPoint（可执行的连接点）

返回值：Object返回值（即目标对象方法的返回值）

抛出Throwable异常。

【示例】

第一步：配置MyAspect类（切面），配置around方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//应用场景：日志、缓存、权限、性能监控、事务管理

//环绕通知：在目标对象方法的执行前+后，可以增强

//参数：可以执行的连接点对象ProceedingJoinPoint（方法），特点是调用proceed()方法可以随时随地执行目标对象的方法（相当于目标对象的方法执行了）

//必须抛出一个Throwable

**public** **Object** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable{

//目标：事务的控制：

//开启事务：

System.*out*.println("-----开启了事务。。。。。。。。。");

//执行了目标对象的方法

Object resultObject = proceedingJoinPoint.proceed();

//结束事务

System.*out*.println("-----提交了事务。。。。。。。。。");

**return** resultObject;//目标对象执行的结果

}

}

第二步：Spring容器中配置，配置applicationContext-aspect.xml

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类的 -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 环绕通知 -->

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

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext-aspect.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

### AfterThrowing 抛出通知

作用：目标代码出现异常，通知执行。记录异常日志、通知管理员（短信、邮件）

应用场景：处理异常（一般不可预知），记录日志

【示例】

第一步：配置MyAspect类（切面），配置aterThrowing方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//作用：目标代码出现异常，通知执行。记录异常日志、通知管理员（短信、邮件）

//只有目标对象方法抛出异常，通知才会执行

//参数1：静态连接点（方法对象）

//参数2：目标方法抛出的异常，参数名随便，但也不能太随便

**public** **void** afterThrowing(JoinPoint joinPoint,Throwable ex){

//一旦发生异常，发送邮件或者短信给管理员

System.*out*.println("++管理员您好，"+joinPoint.getTarget().getClass().getName()+"的方法："

+joinPoint.getSignature().getName()+"发生了异常，异常为："+ex.getMessage());

}

}

第二步：Spring容器中配置，配置applicationContext-aspect.xml

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类的 -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 抛出通知

throwing:通知中的方法的第二个参数，异常类型的参数的名字，在运行的时候，spring会自动将异常传入该参数中。-->

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

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext-aspect.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

此时发现通知的方法并没有执行。

那我们在目标对象的方法中故意抛出异常，大家看看效果

测试：

在ProductService.java中save的方法中，制造异常：

//没有接口的类

**public** **class** ProductService {

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

System.*out*.println("商品保存了。。。。。");

//故意制造异常

int d = 1/0;

}

**public** **int** find() {

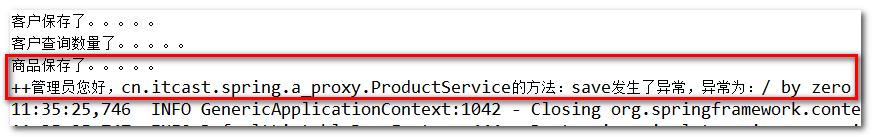
System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

查看测试结果：



### After 最终通知

作用：不管目标方法是否发生异常，最终通知都会执行（类似于finally代码功能）

应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

【示例】

第一步：配置MyAspect类（切面），配置after方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

//最终通知：不管是否有异常都会执行

**public** **void** after(JoinPoint joinPoint){

//释放数据库连接

System.*out*.println("数据库的connection被释放了。。。。。,执行的方法是："+joinPoint.getSignature().getName());

}

}

第二步：Spring容器中配置，配置applicationContext-aspect.xml

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"cn.itcast.spring.a\_proxy.CustomerServiceImpl"*/>

<!-- 基于一般类的 -->

<bean id=*"productService"* class=*"cn.itcast.spring.a\_proxy.ProductService"*/>

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 最终通知 -->

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

<!-- **以上代码也可以写成**：pointcut切入点表达式:只能给一个通知方法来用，相当于省略了<aop:pointcut expression="bean(\*Service)" id="myPointcut"/>

<aop:after method="after" pointcut="bean(\*Service)"/>-->

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext-aspect.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

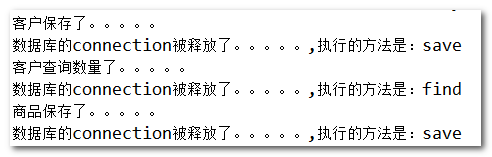
productService.save();

productService.find();

}

}

查看测试结果：



### 通知小结

五种通知小结：

（1）只要掌握Around（环绕通知）通知类型，就可实现其他四种通知效果。

（2）因为你可以在环绕通知的方法中编写如下代码：

try {

//前置通知

Object result = proceedingJoinPoint.proceed();

//后置通知

}catch(Exception){

//抛出通知

}finally{

//最终通知

}

方法格式：

public returnType method (param)

public 返回值类型 方法名 (参数类型 参数名)

返回值类型：void和Object

方法名：任意名称（但是也不能太随意）

参数类型：

\* 参数类型为JoinPoint接口类型，返回值类型为void

\* 参数类型为ProceedingJoinPoint接口类型，返回值类型为Object

具体为：

|  |  |  |  |
| --- | --- | --- | --- |
| 通知类型 | 输入参数(可选) | 返回值类型 | 其他 |
| Before前置通知 | JoinPoint（静态连接点信息） | void |  |
| AfterReturning后置通知 | JoinPoint, Object | void |  |
| Around环绕通知 | ProceedingJoinPoint（可执行的连接点信息） | Object | throws Throwable |
| AfterThrowing抛出通知 | JoinPoint, Throwable | void |  |
| After最终通知 | JoinPoint | void |  |

# @Aspectj注解配置切面编程

## 搭建环境

新建web项目 spring4\_day03\_annotation , 引入依赖

<!-- spring核心依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!-- springaop相关包 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<!-- 日志 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<!-- spring集成测试 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.13.RELEASE</version>

</dependency>

同时导入

applicationContext.xml,

log4j.properties到工程

## 第一步： 编写目标对象 （bean）、spring容器、测试类

创建包：cn.itcast.spring.a\_aspect

（1）：创建接口CustomerService.java

//接口

**public** **interface** CustomerService {

//保存

**public** **void** save();

//查询

**public** **int** find();

}

创建接口的实现类，CustomerServiceImpl

//实现类

/\*\*

\* **@Service**("customerService")

\* 相当于spring容器中定义：

\* <bean id="customerService" class="cn.itcast.spring.a\_aspectj.CustomerServiceImpl">

\*/

@Service("customerService")

**public** **class** CustomerServiceImpl **implements** CustomerService{

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

System.*out*.println("客户保存了。。。。。");

}

**public** **int** find() {

System.*out*.println("客户查询数量了。。。。。");

**return** 100;

}

}

创建类ProductService.java，不需要实现接口

//没有接口的类

/\*\*

\* **@Service**("productService")

\* 相当于spring容器中定义：

\* <bean id="productService" class="cn.itcast.spring.a\_aspectj.ProductService">

\*/

@Service("productService")

**public** **class** ProductService {

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

System.*out*.println("商品保存了。。。。。");

}

**public** **int** find() {

System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

（2）：配置applicationContext.xml

引入几个命名空间：bean、aop、context

<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"*

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

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

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

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

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

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

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

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

使用bean注解的扫描（自动开启注解功能）

<!-- 1。确定目标 -->

<!-- 扫描bean组件 -->

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

（3）：测试代码SpringTest.java

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

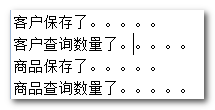
productService.save();

productService.find();

}

}

测试结果：



## 第二步： 编写通知，配置切面

1. 编写通知类，在通知类 添加@Aspect 注解，代表这是一个切面类,并将切面类交给spring管理（能被spring扫描到@Component）。

@Component(“myAspect”)：将增强的类交给spring管理，才可以增强

@Aspect：将该类标识为切面类（这里面有方法进行增强），相当于<aop:aspect ref=”myAspect”>

//advice通知类增强类

@Component("myAspect")//相当于<bean id="myAspect" class="cn.itcast.spring.a\_aspectj.MyAspect"/>

@Aspect//相当于<aop:aspect ref="myAspect">

**public** **class** MyAspect {

}

2） 在切面的类，通知方法上添加

@AspectJ提供不同的通知类型

@Before 前置通知，相当于BeforeAdvice

@AfterReturning 后置通知，相当于AfterReturningAdvice

@Around 环绕通知，相当于MethodInterceptor

@AfterThrowing抛出通知，相当于ThrowAdvice

@After 最终final通知，不管是否异常，该通知都会执行

@DeclareParents 引介通知，相当于IntroductionInterceptor (不要求掌握)

复习回顾：如果是applicationContext.xml中配置通知类型：如下：

<bean **id**="*myAspectAdvice*" **class**="*cn.itcast.spring.c\_aspectaop.MyAspect*"/>

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 前置通知 -->

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

</aop:aspect>

</aop:config>

等同于：以下是简化的写法！可以省略<aop:pointcut>

<bean **id**="*myAspectAdvice*" **class**="*cn.itcast.spring.c\_aspectaop.MyAspect*"/>

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<!-- 前置通知 -->

<aop:before **method**="*before*" **pointcut**="*bean(\*Service)*" />

</aop:aspect>

</aop:config>

3） 在spring容器中开启AspectJ 注解自动代理机制

使用<aop:aspectj-autoproxy/>

作用：能自动扫描带有@Aspect的bean，将其作为增强aop的配置，有点相当于:<aop:config>

<!-- 1。确定目标 -->

<!-- 扫描bean组件 -->

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

<!-- 2:编写通知 -->

<!-- 3：配置aop的aspectj的自动代理：

自动扫描bean组件中，含有@Aspect的bean，将其作为aop管理，开启动态代理 -->

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

### 前置通知

在切面的类MyAspect.java类中添加通知方法@Before()，

方案一：可以直接将切入点的表达式写到@Before()中

//前置通知

//相当于：<aop:before method="before" pointcut="bean(\*Service)"/>

//@Before("bean(\*Service)")：参数值：自动支持切入点表达式或切入点名字

@Before("bean(\*Service)")

**public** **void** before(JoinPoint joinPoint){

System.*out*.println("=======前置通知。。。。。");

}

方案二：可以使用自定义方法，使用@Pointcut 定义切入点

切入点方法的语法要求：

切点方法：private void 无参数、无方法体的方法，方法名为切入点的名称

一个通知方法@Before可以使用多个切入点表达式，中间使用“||”符合分隔，用来表示多个切入点

//自定义切入点

//方法名就是切入点的名字

//相当于<aop:pointcut expression="bean(\*Service)" id="myPointcut"/>

@Pointcut("bean(\*Service)")

**private** **void** myPointcut(){}

//自定义切入点

//方法名就是切入点的名字

//相当于<aop:pointcut expression="bean(\*Service)" id="myPointcut2"/>

@Pointcut("bean(\*Service)")

**private** **void** myPointcut2(){}

//前置通知

//相当于：<aop:before method="before" pointcut-ref="myPointcut"/>

//相当于：<aop:before method="before" pointcut-ref="myPointcut2"/>

@Before("myPointcut()||myPointcut2()")

**public** **void** before(JoinPoint joinPoint){

System.*out*.println("=======前置通知。。。。。");

}

使用SpringTest进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

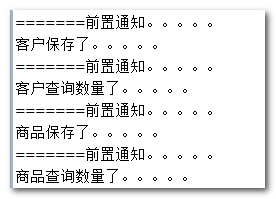
productService.save();

productService.find();

}

}

测试结果：



### 后置通知

在切面的类MyAspect.java类中添加通知方法

//后置通知

@AfterReturning(value="bean(\*Service)",returning="returnVal")

**public** **void** afterReturning(JoinPoint joinPoint,Object returnVal){

System.*out*.println("=======后置通知。。。。。");

}

使用SpringTest进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

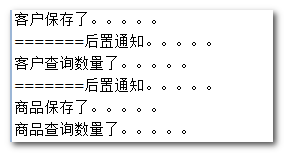
productService.save();

productService.find();

}

}

查看测试结果：



### 环绕通知

在切面的类MyAspect.java类中添加通知方法

//环绕通知：

@Around("bean(\*Service)")

**public** Object around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable{

System.*out*.println("---环绕通知-----前");

Object object = proceedingJoinPoint.proceed();

System.*out*.println("---环绕通知-----后");

**return** object;

}

使用SpringTest进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

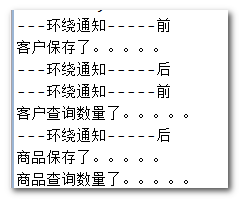
productService.save();

productService.find();

}

}

测试结果：



### 抛出通知

在切面的类MyAspect.java类中添加通知方法

//抛出通知

@AfterThrowing(value="bean(\*Service)",throwing="ex")

**public** **void** afterThrowing(JoinPoint joinPoint ,Throwable ex){

System.*out*.println("---抛出通知。。。。。。"+"抛出的异常信息："+ex.getMessage());

}

使用SpringTest进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

发现没有执行抛出通知，原因是目标对象没有异常，在ProductService添加异常。

@Service("productService")

**public** **class** ProductService {

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

System.*out*.println("商品保存了。。。。。");

**int** d = 1/0;

}

**public** **int** find() {

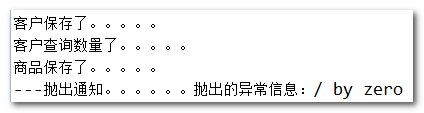
System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

测试结果：



### 最终通知

在切面的类MyAspect.java类中添加通知方法

//最终通知

//拦截所有以ice结尾的bean

@After("bean(\*ice)")

**public** **void** after(JoinPoint joinPoint){

System.*out*.println("+++++++++最终通知。。。。。。。");

}

使用SpringTest进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

测试不管是否抛出异常，都会执行最终通知。

@Service("productService")

**public** **class** ProductService {

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

System.*out*.println("商品保存了。。。。。");

**int** d = 1/0;

}

**public** **int** find() {

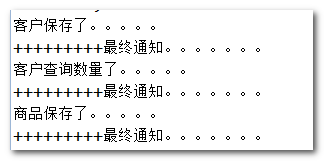
System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

测试结果：



【扩展补充】： 我们的aop代理是使用的Spring的内部代理机制，默认是如果有接口就优先对接口代理（jdk动态代理）。

问题：如果目标对象有接口，能否只对实现类代理，而不对接口进行代理呢？

当然可以了

【测试】

第一步：在CustomerServiceImpl的子类中添加一个新的方法update()，而接口中不要定义update()的方法：

@Service("customerService")

**public** **class** CustomerServiceImpl **implements** CustomerService{

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

System.*out*.println("客户保存了。。。。。");

}

**public** **int** find() {

System.*out*.println("客户查询数量了。。。。。");

**return** 100;

}

//子类扩展方法

**public** **void** update(){

System.*out*.println("客户更新了。。。新增方法。。。");

}

}

第二步：在测试类中调用子类的扩展方法：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** CustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

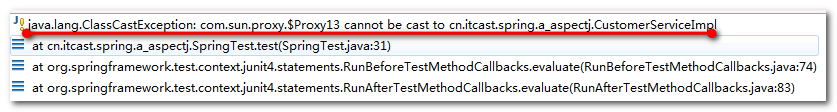
//扩展方法执行:customerService是一个动态代理对象，原因，该对象是接口的子类型的对象

((CustomerServiceImpl)customerService).update();

}

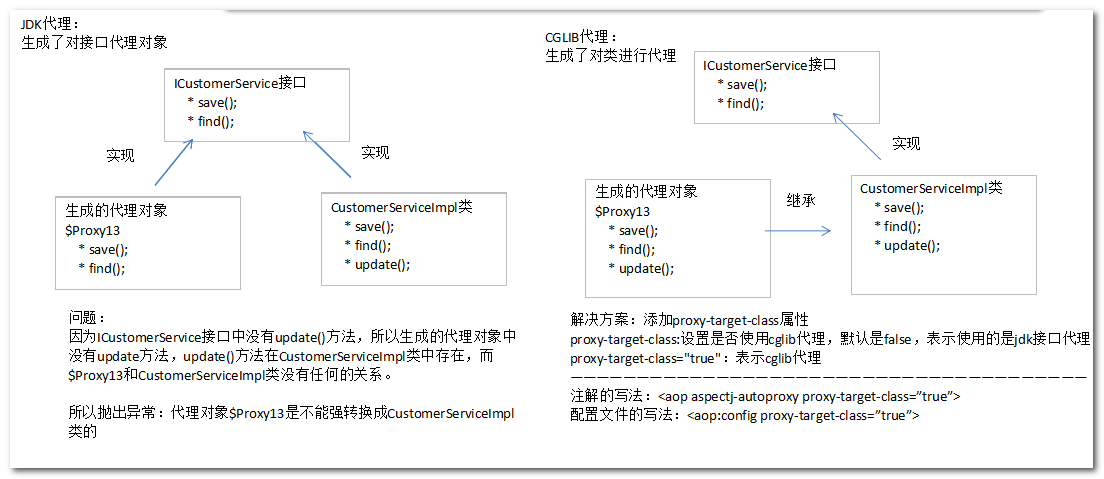
}

结果发现异常：



为什么会抛出异常呢？原因是代理的目标对象是接口，无法转换为子类。





设置 proxy-target-class = true

方案一：注解方式：

<!-- 配置aop的aspectj的自动代理：

自动扫描bean组件中，含有@Aspect的bean，将其作为aop管理，开启动态代理

proxy-target-class:设置是否使用cglib代理，默认是false，表示使用的是jdk接口代理

proxy-target-class="true"：表示cglib代理

-->

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

方案二：配置文件XML的方式

<!-- 3：配置aop -->

<aop:config proxy-target-class=*"true"*>

</aop:config>

# Spring JdbcTemplate的使用

Spring JdbcTemplate 是一个模板工具类，简化Jdbc编程 （类似 Apache DbUtils ）

为了方便Dao中注入JdbcTemplate，Spring为每一个持久化技术都提供了支持类。

Spring对不同持久化技术的支持，Spring为各种支持的持久化技术，都提供了简单操作的模板和回调：

|  |  |
| --- | --- |
| **ORM持久化技术** | **模板类** |
| JDBC | org.springframework.jdbc.core.JdbcTemplate |
| Hibernate5.0 | org.springframework.orm.hibernate5.HibernateTemplate |
| IBatis(MyBatis) | org.springframework.orm.ibatis.SqlMapClientTemplate |
| JPA | org.springframework.orm.jpa.JpaTemplate |

## JdbcTemplate 快速入门

第一步： 基础工程搭建：

新建web项目 spring4\_day3\_jdbctemplate

第二步：引入依赖:

<!-- spring核心依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!-- springaop相关包 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<!-- 日志 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<!-- spring集成测试 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.13.RELEASE</version>

</dependency>

<!-- 操作数据库 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

</dependency>

<!-- MySql -->

<dependency>

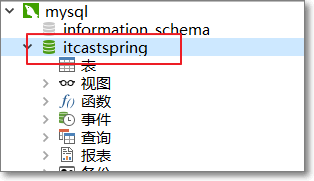
<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

导入配置文件：在src中导入log4j.properties和applicationContiext.xml

第三步： 建立mysql数据库，创建itcastspring



第四步：使用JDBCTemplate编写程序（建表） ,基本步骤如下：

1）构建连接池

2）构建JDBCTemplate

3）调用JDBCTemplate的execute方法

使用mysql数据库，创建包cn.itcast.spring.test，创建测试类JdbcTemplateTest.java进行测试：

**public** **class** JdbcTemplateTest {

@Test

**public** **void** test(){

//目标：使用jdbctemplate执行一段sql

//1.构建数据源

//spring内置了一个数据源

DriverManagerDataSource dataSource = **new** DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///itcastspring");

dataSource.setUsername("root");

dataSource.setPassword("root");

//2.创建jdbctemplate实例

JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

//等同于

// jdbcTemplate.setDataSource(dataSource)

//3.执行sql，创建表test001

jdbcTemplate.execute("create table test001(id int,name varchar(20))");

}

}

## 通过XML配置创建JdbcTemplate对象（多种数据源）

下面将使用几种数据源的方式进行配置。

### Spring内置数据源

目标：将数据源和jdbcTemplate都交给Spring来管理：

在applicationContext.xml中配置dataSource连接池和jdbcTemplate模版对象。编写applicationContext.xml文件

<!--

类似于：DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///itcastspring");

dataSource.setUsername("root");

dataSource.setPassword("root");

-->

<!-- 配置内置的数据源bean -->

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql:///itcastspring"*/>

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

<property name=*"password"* value=*"root"*/>

</bean>

<!-- jdbctemplate对象 -->

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<!-- 注入数据源 -->

<property name=*"dataSource"* ref=*"dataSource"*/>

</bean>

数据源:DriverManagerDataSource是spring内置的连接池，不建议生产环境使用，可以在测试环境使用

编写测试，使用SpringTest.java进行测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** JdbcTemplate jdbcTemplate;

@Test

**public** **void** testCreatetable(){

jdbcTemplate.execute("create table test002(id int,name varchar(20))");

}

}

### C3P0 连接池配置

引入c3p0依赖:

<!-- c3p0数据源 -->

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

配置applicationContext.xml文件

<!-- c3p0连接池 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"jdbcUrl"* value=*"jdbc:mysql:///itcastspring"*/>

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

<property name=*"password"* value=*"root"*/>

</bean>

<!-- jdbctemplate对象 -->

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<!-- 注入数据源 -->

<property name=*"dataSource"* ref=*"dataSource"*/>

</bean>

测试类：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** JdbcTemplate jdbcTemplate;

@Test

**public** **void** testCreatetable(){

jdbcTemplate.execute("create table test003(id int,name varchar(20))");

}

}

## 外部属性文件的配置

模拟需求：

现在数据源的相关参数配置，是测试环境下的。

现在，要将工程搭建在正式的服务器上，因为测试环境和正式环境的数据库肯定不是一个，所以肯定首先要更改数据源相关的配置。

缺点：必须手动修改applicationContext.xml文件，容易造成误操作。

解决方案：不修改。可以将数据源相关配置参数，外置。

目的：可以将xml配置中可能要经常修改内容，抽取到一个properties文件

应用：使用properties文件配置参数，如数据库连接参数等。

第一步： src新建db.properties

将经常需要修改变量抽取出来

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql:///itcastspring

jdbc.username=root

jdbc.password=root

第二步： 配置applicationContext.xml文件，在applicationContext.xml 通过

<context:property-placeholder> 引入外部属性文件

通过${key} 引用属性的值

<!-- 引入外部属性配置文件-->

<context:property-placeholder location=*"classpath:db.properties"*/>

<!-- 配置内置的数据源bean，使用db.properties -->

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

<property name=*"password"* value=*"${jdbc.password}"*/>

</bean>

第三步：使用SpringTest.java进行测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** JdbcTemplate jdbcTemplate;

@Test

**public** **void** testCreatetable(){

jdbcTemplate.execute("create table test004(id int,name varchar(20))");

}

}

知识点：

1. AspectJ AOP 编程 （XML或者注解 重点掌握一套 ）

问题：advice（通知增强）、 advisor（传统aop切面配置标签） 、aspect（aspectj的aop切面配置的标签） 、aspectj（可以进行aop的第三方的开源框架）

Advice 通知 ，增强代码

Advisor 传统SpringAOP切面 ，只包含一个切入点和一个通知

Aspect AspectJ定义切面 ，可以包含多个切入点和多个通知

AspectJ 第三方AOP框架

1. 数据源配置
2. 外部属性文件 （context:placehoder）