# 尚马教育 JAVA 课程

# Spring切面

文档编号：C04

创建日期： 2017-07-07

最后修改日期：2021-01-21

版 本 号：V3.5

电子版文件名：尚马教育-第三阶段-4.spring切面.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2017-07-30 | 张元林 | 初始版本 | V1.0 |
| 2018-08-01 | 王绍成 | Mybatis版本更新 | V2.0 |
| 2019-08-09 | 冯勇涛 | 课件格式以及课程深度加深 | V3.0 |
| 2019-11-06 | 冯勇涛 | 格式调整 | V3.1 |
| 2021-01-21 | 冯勇涛 | 优化aop课程，提供案例 | V3.5 |

目录

[尚马教育 JAVA 课程 1](#_Toc13542)

[Spring切面 1](#_Toc19004)

[1. AOP介绍 3](#_Toc23018)

[2. 业务分析 4](#_Toc21636)

[3. Aop中的概念 8](#_Toc23992)

[4. Spring中aop 9](#_Toc13638)

[4.1. 环境准备 9](#_Toc31372)

[4.2. 基于aspectj注解方式进行AOP开发 9](#_Toc5785)

[4.2.1. 开启注解支持 9](#_Toc20419)

[4.2.2. 定义切面类 11](#_Toc18817)

[4.2.10. 测试类 15](#_Toc17670)

[4.3. 基于XML配置方式进行AOP开发 16](#_Toc20203)

[4.4. 基于自定义注解定义切入点表达式 16](#_Toc22474)

[4.5. 通知总结 17](#_Toc13749)

[4.6. 切入点表达式实例 19](#_Toc11723)

[4.7. Aop任务 19](#_Toc23987)

知识点：

|  |
| --- |
| 知识点1：认识aop思想  知识点2：认识spring中的aop两种实现方式 |

## AOP介绍

Aop：面向切面编程，是一种在程序运行期间通过动态代理实现在不修改源代码的情况下给程序动态统一的添加新功能的一种技术。是GOF设计模式的一种延续。

|  |
| --- |
| AOP是什么(Aspect Oriented Programming)  AOP是一种编程范式，提供从另一个角度来考虑程序结构以完善面向对象编程（OOP）。  AOP为开发者提供了一种描述横切关注点的机制，并能够自动将横切关注点织入到面向对象的软件系统中，从而实现了横切关注点的模块化。  AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如日志记录，性能统计，安全控制，事务处理等封装起来，将以上这些代码从业务逻辑代码中划分出来，通过对这些行为的分离，达到改变这些行为而不影响核心业务逻辑的代码目的。  在应用 AOP 编程时, 仍然需要定义公共功能, 但可以明确的定义这个功能在哪里, 以什么方式应用, 并且不必修改受影响的类. 这样一来横切关注点就被模块化到特殊的对象(切面)里。  AOP 的好处:  每个事物逻辑位于一个位置, 代码不分散, 便于维护和升级  业务模块更简洁, 只包含核心业务代码 |

## 业务分析

设已有如下的程序实现:

|  |
| --- |
|  |

考虑这样一个问题:

要对系统中的支付方法添加日志记录、性能监测、安全控制等功能.面对这样的需求我们应该怎么办呢?

一个很自然的实现方式,可能如下：

|  |
| --- |
|  |

很快大家会发现里面有很多重复性的代码,一个自然改进的方式是:

把公共部分提取出来,做成公共模块,然后让应用调用这些模块，代码如下：

|  |
| --- |
|  |

仍然存在问题：  
 大家会发现，需要修改的地方分散在很多个文件中，如果需要修改的文件多那么修改的量会很大，这无疑会增加出错的几率，并且加大系统维护的难度。  
而且，如果添加功能的需求是在软件开发的后期才提出的话，这样大量修改已有的文件，也不符合基本的“开-闭原则”。

按照章节1中aop思想，对代码进行重新设计，需要通过动态代理模式，实现pay方法中的代码动态的织入，而不是手工添加到核心付款方法中。

代码如下：

|  |
| --- |
| 1. 自定义异常对象：   **public class** PayException **extends** Exception {  **public** PayException(String message) {  **super**(message);  } } |
| 1. **定义付款接口**   **public interface** IPay {  **public** String pay(**Double** money, String customUser, String busnessUser) **throws** PayException ; } |
| 1. 定义付款接口的阿里付款与微信付款实现类 **public class** AliPay **implements** IPay {  @Override  **public** String pay(D**ouble** money, String customUser, String busnessUser) **throws** PayException {  **if**(payMoney<0)  **throw new** PayException(**"金额小于0"**);  System.***out***.println(**"模拟调用阿里付款接口,耗时1050毫秒"**);  **try** {  Thread.*sleep*(1050);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return "aliSuc"**;  } } |
| **public class** WechatPay **implements** IPay{  @Override  **public** String pay(**Double** money, String customUser, String busnessUser) {  **if**(payMoney<0)  **throw new** NullPointerException(**"金额小于0"**);  System.***out***.println(**"调用wechat付款接口,模拟耗时1300毫秒"**);  **try** {  Thread.*sleep*(1300);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return "wechatSuc"**;  } } |
| 1. **测试方法**   **public static void** main(String[] args) **throws** PayException {  IPay wechatPay = **new** WechatPay();  IPay aliPay = **new** AliPay();   IPay wechatPayProxy =*getProxy*(wechatPay);  IPay aliPayProxy = *getProxy*(aliPay);   System.***out***.println(**"输入付款方式(wechat|ali)："**);  Scanner s = **new** Scanner(System.***in***);  String payWay= s.next();  System.***out***.println(**"输入付款金额："**);  **double** rmb = s.nextDouble();  String result = **null**;  **if**(**"wechat"**.equals(payWay)){  result = wechatPayProxy.pay(rmb,**"三儿"** , **"老板"**);  }**else if**(**"ali"**.equals(payWay)){  result= aliPayProxy.pay(rmb,**"三儿"** , **"老板**);  }  System.***out***.println(**"付款结果："**+result); }  **public static** IPay getProxy(IPay source){  IPay wechatPayProxy = (IPay)Proxy.*newProxyInstance*(source.getClass().getClassLoader(), **new** Class[]{IPay.**class**}, **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  Object result= **null**;  **try**{  System.***out***.println(**"前置增强通知，取得method与args信息:"**+method.getName()+**"--"**+ Arrays.*toString*(args));  result= method.invoke(source, args);  System.***out***.println(**"返回增强通知,取得执行结果："**+result);  }**catch**(Exception e){  System.***out***.println(**"异常通知，取得异常信息："**+e.getMessage());  result=**"error"**;  }**finally** {  System.***out***.println(**"最终通知"**);  }  **return** result;  }  });  **return** wechatPayProxy; } |

## Aop中的概念

|  |
| --- |
| * Aspect(切面):指横切性关注点的抽象即为切面,它与类相似,只是两者的关注点不一样,类是对物体特征的抽象,而切面是横切性关注点的抽象. |
| * Advice(增强通知):所谓通知是指拦截到joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知。 |
| * Joinpoint(连接点):所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点,实际上joinpoint还可以是field或类构造器) |
| * Pointcut(切入点):所谓切入点是指我们要对那些joinpoint进行拦截的定义。 |
| * Target(目标对象):代理的目标对象。 |
| * Weave(织入):指将aspects应用到target对象并导致proxy对象创建的过程称为织入。 |

## Spring中aop

### 环境准备

使用AOP编程，除了原来Spring导入的包以外还需要导入的包：

aopalliance-1.0.jar

Aspectjrt-xx.jar

Aspectjweaver-xx.jar

cglib-nodep-2.1\_3.jar

spring-aop.jar

### 基于aspectj注解方式进行AOP开发

AspectJ是Java 社区里最完整最流行的 AOP 框架，提供了几种常见的通知类型参考：

前置通知（Before advice）

返回后通知（After returning advice）

抛出异常后通知（After throwing advice）

后通知（After (finally) advice）

环绕通知（Around Advice）

在 Spring2.0 以上版本中, 可以使用基于 AspectJ 注解配置的 AOP

#### 开启注解支持

在spring配置文件中配置**aop:aspectj-autoproxy标签。**

|  |
| --- |
| <**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 https://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd"**>  <**context:component-scan base-package="com.javasm"**></**context:component-scan**>  *<!--用来识别aop相关的注解:@Aspect,@Pointcut,@Before,@After,@AfterReturning,@AfterThrowsing,@Around-->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**>  </**beans**> |

#### 定义切面类

|  |
| --- |
| @Component  @Aspect  **public** **class** LogAspect{  } |

注意点：

注意点1：该类必须注册进spring容器，被spring管理。

注意点2：该类使用Aspect注解标注是一个切面bean对象。

#### 定义切入点

|  |
| --- |
| @Pointcut(**"execution(\* com.javasm.\*.service.\*.\*(..))"**) **public void** servicePointcut(){} |

注意点：

注意点1：切入点的定义通过Pointcut注解定义，注解到方法上，该方法是空方法；

注意点2：切入点表达式的定义有很多种方法，这里写的是最常用的一种，通过executrion定义，规则：execution(返回值类型 包.类.方法(形参))，其中包与类可以省略。

#### 定义前置通知

通过Before注解来定义前置通知方法。

|  |
| --- |
| @Before(**"servicePointcut()"**) **public void** log(JoinPoint jp){  *//连接点方法的实参*  Object[] args = jp.getArgs();  *//目标对象* Object target = jp.getTarget(); *//连接点方法名*  String name = jp.getSignature().getName();System.***out***.println(**"log前置通知"**+target.getClass().getName()+**"."**+name+**"--"**+ Arrays.*toString*(args)); } |

注意点：

注意点1：Before注解内必须指定切入点名称。表示前置通知织入的位置。

注意点2：前置通知中可以得到连接点的信息。添加JoinPoint形参即可。

#### 定义返回通知

使用AfterReturning注解定义返回通知。

|  |
| --- |
| @AfterReturning(value=**"servicePointcut()"**,returning = **"obj"**) **public void** afterReturning(JoinPoint jp,Object obj){  Object[] args = jp.getArgs();*//实参* Object target = jp.getTarget();*//目标对象* String name = jp.getSignature().getName();*//连接点方法名* System.***out***.println(**"返回通知"**+name+**",返回值:"**+obj); } |

注意点：

注意点1：返回通知中能获取连接点信息与连接点方法执行返回值。

注意点2：AfterReturning注解的returning属性值与方法形参名一致。

#### 定义异常通知

通过AfterThrowing注解定义异常通知方法。

|  |
| --- |
| @AfterThrowing(value = **"servicePointcut()"**,throwing = **"e"**) **public void** afterThrowing(JoinPoint jp,Exception e){  String name = jp.getSignature().getName();*//连接点方法名* System.out.println(**"异常通知"**+name+**",异常信息："**+e.getMessage()); } |

注意点1：异常通知中能够获取异常对象，throwing属性值与异常通知方法形参名一致。

#### 定义最终通知

通过After注解定义最终通知

|  |
| --- |
| @After(**"servicePointcut()"**) **public void** after(JoinPoint jp){  String name = jp.getSignature().getName();*//连接点方法名* System.out.println(**"最终通知"**+name); } |

注意点1：最终通知中只能获取连接点信息。

以上4类增强通知，在实际使用中，一般只使用其中的某一个或两个，比如定义日志切面，仅需定义最终通知；定义异常处理切面，仅需定义异常通知；如果定义某个切面需要4类通知都定义，则建议使用环绕通知，如下。

#### 定义环绕通知

通过Around定义环绕通知，环绕通知是最强大的一个通知。

|  |
| --- |
| @Around(**"servicePointcut()"**) **public** Object aroundMethod(ProceedingJoinPoint jp){  **try** {  Object[] args = jp.getArgs();*//实参* Object target = jp.getTarget();*//目标对象* String name = jp.getSignature().getName();*//连接点方法名* System.out.println(**"前置"**+target.getClass().getName()+**"."**+name+**"--"**+args);  Object result = jp.proceed();*//执行连接点方法,并得到连接点方法的返回值* System.out.println(**"返回"**+result);  **return** result;  } **catch** (Throwable throwable) {  System.out.println(**"异常"**+throwable.getMessage());  }**finally**{  System.out.println(**"最终"**);  }  **return null**; } |

注意点：

注意点1：环绕通知能够取代前4类通知，很方便的再连接点方法的前后异常最终等各个位置动态织入代码。

注意点2：环绕通知方法返回值必须是Object，把连接点方法的执行结果继续返回上级。

注意点3：环绕通知方法的形参必须是ProceedingJoinPoint 对象，该对象中有proceed方法执行连接点方法，并得到方法返回值。

注意点4：proceed方法抛出受检异常，必须try-catch-finally处理。

注意点5：如果连接点方法内部已经对异常try-catch后，则如果连接点方法内产生异常，不能切面类中的异常通知代码。

注意点6：不要把环绕通知与普通增强通知混合使用。

#### Service层代码

|  |
| --- |
| @Service **public class** AliPayServiceImpl **implements** IPayService {  @Override  **public boolean** pay(**double** money, String customUser, String busnessUser) **throws** PayException {  **try** {  Thread.*sleep*(1500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if**(money>1000)  **throw new** PayException(**"付款不允许超过1000"**);  System.***out***.println(**"支付宝付款："**+customUser+**"付款给"**+busnessUser+**"，金额："**+money);  **return true**;  } } |

注意点：该类必须定义在切入点表达式execution(\* com.javasm.\*.service.\*.\*(..))定义的包范围内。

#### 测试类

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(**"classpath:spring.xml"**) **public class** TestAop {  @Resource  **private** IPayService **payService**;  @Test  **public void** test1\_testSysuserController() **throws** PayException {  **boolean** result = **payService**.pay(100.0,**"三儿"**,**"老板"**);  System.***out***.println(result);  } } |

注意点：

注意点1：观察注入的payService对象，是原生的AliPayServiceImpl类型吗？Dubug观察结果如下：

|  |
| --- |
|  |

注意点2：注意观察执行结果，可以测试付款金额大于1000，抛出异常进行测试。

### 基于XML配置方式进行AOP开发

|  |
| --- |
| *<!--注册Log切面-->* <**bean id="logAspect" class="com.javasm.sys.aspect.LogAspect"**></**bean**> *<!--进行aop配置-->* <**aop:config**>  <**aop:aspect ref="logAspect"**>  <**aop:pointcut id="servicePointcut" expression="execution(\* com.javasm.\*.service.\*.\*(..))"**></**aop:pointcut**>  <**aop:before method="before" pointcut-ref="servicePointcut"**></**aop:before**>  <**aop:after-returning method="afterReturning" pointcut-ref="servicePointcut" returning="obj"**></**aop:after-returning**>  <**aop:after-throwing method="afterThrowing" pointcut-ref="servicePointcut" throwing="e"**></**aop:after-throwing**>  <**aop:after method="after" pointcut-ref="servicePointcut"**></**aop:after**>  *<!--<aop:around method="aroundMethod" pointcut-ref="servicePointcut"></aop:around>-->* </**aop:aspect**> </**aop:config**> |

### 基于自定义注解定义切入点表达式

#### 自定义日志注解

|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***) @Target({ElementType.***METHOD***}) **public** @**interface** Log {  String value() **default ""**; } |

#### 修改切入点表达式

|  |
| --- |
| @Pointcut(**"@annotation(com.javasm.sys.aspect.Log)"**) **public void** servicePointcut(){} |

注意点：**"@annotation(com.javasm.sys.aspect.Log)"表示带有Log注解的方法即连接点方法。**

#### 修改服务层pay方法实现

|  |
| --- |
| @Log(**"阿里付款"**) @Override **public boolean** pay(**double** money, String customUser, String busnessUser) **throws** PayException {  **try** {  Thread.*sleep*(1500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if**(money>1000)  **throw new** PayException(**"付款不允许超过1000"**);  System.***out***.println(**"支付宝付款："**+customUser+**"付款给"**+busnessUser+**"，金额："**+money);  **return true**; } |

注意点：方法上添加Log注解，表示该方法被LogAspect切面织入增强通知。

### 通知总结

前置通知:在切入点选择的连接点处的方法之前执行的通知，该通知不影响正常程序执行流程（除非该通知抛出异常，该异常将中断当前方法链的执行而返回）。

后置返回通知:在切入点选择的连接点处的方法正常执行完毕时执行的通知，必须是连接点处的方法没抛出任何异常正常返回时才调用后置通知。

后置异常通知:在切入点选择的连接点处的方法抛出异常返回时执行的通知，必须是连接点处的方法抛出任何异常返回时才调用异常通知。

后置最终通知:在切入点选择的连接点处的方法返回时执行的通知，不管抛没抛出异常都执行，类似于Java中的finally块。

环绕通知：环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知可以在方法调用之前和之后自定义任何行为，并且可以决定是否执行连接点处的方法、替换返回值、抛出异常等等。

### 切入点表达式实例

expression-匹配方法执行的连接点，是Spring最主要的切入点。

除了类型模式、名字模式和参数模式以外，其余的都是可选的，\*为任意通配符

|  |
| --- |
| expression=“execution(\* com.javasm.\*.service.\*.\*(..))”  表示作用在com.javasm.service包及子包下所有的类的所有方法上 |
| expression=“execution(\* com.javasm.service.\*.\*(java.lang.String,..))”  表示作用在com.javasm.service包及子包下所有的类的第一个参数为String类型的方法上 |
| expression=“execution(java.lang.String com.javasm.service.\*.\*(..))”  表示作用在com.javasm.service包及子包下所有的类的返回值类型为String的方法上 |
| expression=“execution(!java.lang.String com.javasm.service.\*.\*(..))”  表示作用在com.javasm.service包及子包下所有的类的返回值类型不是String的所有方法上 |
| expression=“execution(\* set\*(..))”  表示作用在任意以set开始的方法上 |

### Aop任务

以aop思想，实现事务管理切面。

0.DataSource注册容器

1. 定义事务管理器MyTransactionManager对象，注册进spring容器。(依赖Druid连接池,获取连接,开始事务,提交事务,回滚事务,关闭连接)
2. 定义dao接口，使用jdbc定义dao实现，不要使用mybatis，把dao实现注册spring容器(curd)
3. 定义service接口，定义service实现，注册容器(curd)。
4. 定义事务切面(依赖MyTransactionManager)。