# Spring AOP 简介

## Spring AOP 概述

### Spring AOP 是什么

AOP 是Aspect-Oriented Programming的缩写，被译成面向切面编程。面向切面编程是通过预编译的方式或运行时动态代理方法实现业务中的一个扩展功能。可以说是OOP

（Object-Oriented Programing，面向对象编程）的补充和完善。实际项目中我们通常会将OOP与AOP结合使用。

Spring中依托于@AspectJ框架，底层通过动态代理方式实现了面向切面编程，我们称之为Spring的AOP。

Spring 框架核心功能模块之一

Spring 中的面向切面编程(Aspect Oriented Programming)

Spring 中业务对象的横切面

提示：AOP把软件系统分为两个部分：核心关注点和横切关注点。

业务处理的主要流程是核心关注点，非核心流程是横切关注点。

### Spring AOP 应用特点

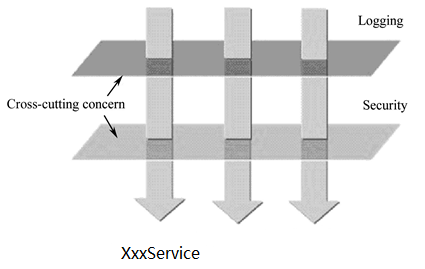
Spring AOP基于OCP原则，在不改变原有功能代码的基础上扩展新的功能，这样可以很好的实现业务代码解耦，提高代码的可重用性，以简化代码编写，提高开发效率。

### Spring AOP 应用场景

作为面向对象编程的一种补充，广泛应用于处理一些具有横切性质的系统级服务，如

系统日志处理，系统事务处理，系统安全验证，系统数据缓存等。

例如



## Spring AOP 核心概念及原理

### Spring AOP 核心概念

切面(aspect): 横切面对象,一般为一个具体类对象(可以借助@Aspect声明)

连接点(joinpoint):程序执行过程中某个特定的点，一般指被拦截到的的方法

切入点(pointcut):对连接点拦截内容的一种定义

通知(Advice):在切面的某个特定连接点上执行的动作，例如before,after等

很晦涩难懂，多做例子，做完就会清晰。先可以按白话去理解。

### Spring AOP 应用原理

Spring AOP 底层依赖于代理机制，在对象运行时可以动态织入新的扩展功能。

Spring创建代理的规则：

1、默认使用Java动态代理来创建AOP代理，这样就可以为任何接口实例创建代理了

2、当需要代理的类不是代理接口的时候，Spring会切换为使用CGLIB代理。

# Spring AOP 编程实现

## Spring AOP 编程基本关注点

纵观SpringAOP编程，程序员只需要参与三个部分：

1. 定义普通业务切面（Aspect）
2. 定义切入点（Pointcut），一个切入点可能横切多个业务组件
3. 定义增强处理（Advice），增强处理就是在AOP框架为普通业务组件织入的处理动作（从属于上讲就是通知）

对于以上AOP编程的三个部分最关键就是定义切入点和定义增强处理，一旦定义了合适的切入点和增强处理

## Spring AOP 编程基本步骤

1. 创建maven项目并添加依赖
2. 添加spring配置文件启用注解配置
3. 创建核心业务接口及实现类
4. 创建业务切面类，定义切入点及增强处理
5. 编写测试类进行测试

## Spring AOP 项目创建及依赖

创建maven的java或web项目，并添加AOP依赖。

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.9.RELEASE</version>

</dependency>

<!-- 1.8.5有问题 -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>1.8.9</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.8.9</version>

</dependency>

</dependencies>

添加单元测试依赖

<dependency>

<groupId>Junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

## Spring AOP 基于XML方式实现

### Spring AOP 核心业务组件定义

定义SayHelloService接口

**public** **interface** HelloService {

**void** sayHello(String msg);

}

定义SayHelloServiceImpl实现类实现SayHelloService接口

**@Service**

**public** **class** HelloServiceImpl **implements** HelloService {

**public** **void** sayHello(String msg) {

System.***out***.println("hello "+msg);

}

}

### Spring AOP 切面业务组件定义

@Component

**public** **class** LoggingAspect {

**public** **void** beforeMethod() {

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

}

**public** **void** afterMethod() {

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

}

}

### Spring AOP 基于xml方式配置

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

<beans

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

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

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

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

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

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

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

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

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

xsi:schemaLocation=*"*

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

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd*

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

*http://www.springframework.org/schema/tx/spring-tx-4.3.xsd*

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

*http://www.springframework.org/schema/aop/spring-aop-4.3.xsd*

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

*http://www.springframework.org/schema/util/spring-util-4.3.xsd*

*http://www.springframework.org/schema/data/jpa*

*http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"*>

<!-- 自动扫描该包 -->

<context:component-scan base-package=*"com.company.spring"* />

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

<aop:pointcut

expression=*"within(com.company.spring.service.HelloServiceImpl)"*

id=*"pc"*/>

<!-- 切面触发的执行类 -->

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

<!-- 切面触发的执行类的before方法 -->

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

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

</aop:aspect>

</aop:config>

</beans>

基于xml方式实现AOP的配置，代码的侵入比较小，但灵活性不足。

### Spring AOP 测试

ClassPathXmlApplicationContext ctx=

**new** ClassPathXmlApplicationContext("aop.xml");

HelloService helloService=

ctx.getBean("helloServiceImpl",HelloService.**class**);

helloService.sayHello("hello");

ctx.close();

## Spring AOP 基于注解方式实现

### Spring AOP 核心业务组件定义

定义SayHelloService接口

**public** **interface** MessageService {

**void** sendMsg(String msg);

}

定义SayHelloServiceImpl实现类实现SayHelloService接口

@Service

**public** **class** MessageServiceImpl **implements** MessageService{

**public** **void** sendMsg(String msg) {

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

}

}

### Spring AOP 切面业务组件定义

@Component

@Aspect

**public** **class** TxManager {

@Pointcut("within(com.company.spring.service.MessageServiceImpl)")

**public** **void** pointCut() {}

@Before("pointCut()")

**public** **void** beginTx() {

System.***out***.println("begin transaction");

}

@After("pointCut()")

**public** **void** endTx() {

System.***out***.println("end transaction");

}

}

其中：

@Component 表示此组件由Spring对象管理

@Aspect 用于定义切面（封装扩展功能）

@Pointcut 用于定义切入点（用于织入扩展功能的点）

@Before 用于定义前置通知（业务方法之前执行）

@After 用于定义最终通知（业务方法执行完成以后执行）

### Spring AOP 配置文件启用AOP功能

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

<beans default-lazy-init=*"true"*

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

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

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

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

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

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

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

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

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

xsi:schemaLocation=*"*

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

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd*

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

*http://www.springframework.org/schema/tx/spring-tx-4.3.xsd*

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

*http://www.springframework.org/schema/aop/spring-aop-4.3.xsd*

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

*http://www.springframework.org/schema/util/spring-util-4.3.xsd*

*http://www.springframework.org/schema/data/jpa*

*http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"*>

<!-- 自动扫描该包 -->

<context:component-scan base-package=*"cn.tedu.aop"* />

<!-- 使aspectj注解生效，自动为目标对象生成代理对象 -->

<aop:aspectj-autoproxy/>

</beans>

说明：基于AOP注解的方式配置AOP时，因为注解要应用在类或方法的上面，所以会由一定的侵入性。但这种侵入性属于声明式侵入性，所以范畴同样也是属于弱耦合范围之内（强耦合一般是在业务代码内部的直接耦合）。

### Spring AOP 测试

ClassPathXmlApplicationContext ctx=

**new** ClassPathXmlApplicationContext("aop.xml");

MessageService msgService=

ctx.getBean("messageServiceImpl",MessageService.**class**);

msgService.sendMsg("hello");

ctx.close();

# Spring AOP 编程增强

## Spring AOP 切入点表达式增强

Spring中通过切入点表达式定义具体切入点，其常用AOP切入点表达式定义及说明：

|  |  |  |
| --- | --- | --- |
|  | 指示符 | 作用 |
|  | bean | 用于匹配指定类型内的方法执行 |
|  | within | 用于匹配指定类型内的方法执行 |
|  | execution | 用于匹配方法执行的连接点 |

### bean 表达式

bean应用于类级别，实现粗粒度的控制：

|  |  |
| --- | --- |
| bean(UserServiceImpl)) | 指定一个类 |
| bean(\*Service) | 指定所有的后缀为service的类 |

例如：

@Aspect

@Component

**public** **class** LoggingAspect {

@Before("bean(xxxService)")

**public** **void** beforeMethod() {

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

}

@After("bean(xxxService)")

**public** **void** afterMethod() {

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

}

}

### within 表达式

within应用于类级别，实现粗粒度的控制：

|  |  |
| --- | --- |
| within(aop.service.UserServiceImpl)) | 指定类，只能指定一个类 |
| within(aop.service.\*)) | 只包括当前目录下的类 |
| within(aop.service..\*)) | 指定当前目录包含所有子目录中的类 |

@Aspect

@Component

**public** **class** LoggingAspect {

@Before("within(aop.XxxService.\*)")

**public** **void** beforeMethod() {

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

}

@After("within(aop.XxxService.\*)")

**public** **void** afterMethod() {

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

}

}

### execution 表达式

execution方法级别，细粒度的控制：

语法：execution(返回值类型 包名.类名.方法名(参数列表))

|  |  |
| --- | --- |
| execution(void aop.service.UserServiceImpl.addUser()) | 匹配方法 |
| execution(void aop.service.PersonServiceImpl.addUser(String)) | 方法参数必须为字符串 |
| execution(\* aop.service..\*.\*(..)) | 万能配置 |

例如：

@Aspect

@Component

**public** **class** TimingAspect {

@Before("execution(\* cn.tedu..\*Service.\*(..))")

**public** **void** timeBeforeMethod(JoinPoint point) {

String method=point.getSignature().getName();

System.***out***.println("timeBeforeMethod:"+method);

}

/\*\*方法切入点 (execution: 执行)\*/

@After("execution(\* cn.tedu..\*Service.update(..))")

**public** **void** timeAfterMethod(JoinPoint point) {

String method=point.getSignature().getName();

Object arg=point.getArgs()[0];

System.***out***.println("timeAfterMethod:"+method+"("+arg+")");

}

}

## Spring AOP 配置增强

### Spring AOP xml 方式配置增强

切入点及前置通知，后置通知，返回通知，异常通知，环绕通知的配置

<aop:config>

<aop:pointcut id=*"pc"*expression=*"execution(\**

*com.company.spring.service..\*.\*(..))"* >

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

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

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

<aop:after-returning method=*"returnMethod"*

pointcut-ref=*"pc"*/>

<aop:after-throwing method=*"throwMethod"*

pointcut-ref=*"pc"*/>

</aop:aspect>

</aop:config>

切入点及环绕通知的配置

<aop:config>

<aop:pointcut id=*"pc"*expression=*"execution(\**

*com.company.spring.service..\*.\*(..))"* >

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

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

</aop:aspect>

</aop:config>

### Spring AOP 注解方式配置增强

切入点及前置通知，后置通知，返回通知，异常通知，环绕通知的配置

@Aspect

@Component

**public** **class** LoggingAspect {

@Before("bean(xxxService)")

**public** **void** beforeMethod() {

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

}

@After("bean(xxxService)")

**public** **void** afterMethod() {

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

}

@AfterReturning(pointcut="bean(xxxService)",returning="result")

**public** **void** afterReturningMethod(Object result) {

System.***out***.println("afterReturnMethod.result="+result);

}

@AfterThrowing("bean(xxxService)")

**public** **void** afterThrowingMethod() {

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

}

}

切入点及环绕通知的配置

@Component

@Aspect

**public** **class** TxManager {

@Pointcut("execution(com.company.spring.service..\*.\*(..))")

**public** **void** pointCut() {}

@Around("pointCut()")

**public** Object around(ProceedingJoinPoint joinPoint)

**throws** Throwable{

System.***out***.println("事务开始");

Object result = joinPoint.proceed();

System.***out***.println("事务结束");

**return** result;

}

}

## Spring AOP 切面顺序配置

当业务系统中存在多个切面时，哪个切面优先执行，哪个切面要延后执行，我们可以通过配置切面 执行顺序实现，序号小的会优先执行。

### 基于xml方式配置

<!-- 自动扫描该包 -->

<context:component-scan base-package=*"com.company.spring"* />

<!-- AOP配置 -->

<aop:config>

<aop:pointcut id=*"pc"*

expression=*"execution(\**

*com.company.spring.service..\*.\*(..))"*/>

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

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

</aop:aspect>

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

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

</aop:aspect>

</aop:config>

### 基于注解方式配置

事务切面

@Order(1)

@Aspect

@Component

**public** **class** TxManager {

@Pointcut("execution(\* com.company.spring.service..\*.(..))")

**public** **void** pointCut() {}

@Around("pointCut()")

**public** Object around(ProceedingJoinPoint joinPoint)

**throws** Throwable{

System.***out***.println("事务开始");

Object result = joinPoint.proceed();

System.***out***.println("事务结束");

**return** result;

}

}

日志切面

@Order(2)

@Aspect

@Component

**public** **class** LoggingAspect {

@Pointcut("execution(\* com.company.spring.service..\*.(..))")

**public** **void** pointCut() {}

@Before("pointCut()")

**public** **void** beforeMethod() {

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

}

@Before("pointCut()")

**public** **void** afterMethod() {

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

}

}

# 总结

## 重点和难点分析

|  |
| --- |
|  |

## 常见FAQ

|  |
| --- |
|  |