**03、AOP编程**

# AOP基本介绍

## AOP简介

* + AOP(Aspect Oriented Programing)面向切面/方面编程
  + AOP隶属软件工程的范畴，指导开发人员如何制作开发软件，进行结构设计
  + AOP联盟：掌握着AOP的发展方向
  + AOP开发只关注共性功能
  + 主思想：在不惊动原始设计的基础之上，为原始设计追加功能，称之为插拔式组件体系结构

## OOP与AOP区别

* + OOP开发，关注的是层与层之间的关系，及其每层中各个实现类如何制作
  + AOP开发，关注的是共性功能，无论该共性功能是否服务于同一层面的类,将共性功能抽取出来，制作成独立的模块，在程序运行时，动态的为其追加该功能。

## AOP核心思想

AOP关注的是程序中的共性功能，开发时，将共性功能抽取出来制作成独立的功能模块，此时原始功能中将不具有这些被抽取出的共性功能代码。在当具有被抽取的共性功能的模块运行时候，将共性功能，模块进行组装运行，即可完成原始的功能。

* 优点：加强代码的复用性，同时程序开发时可以只考虑个性化的功能，不需要考虑共性功能

## AOP基本概念

* 连接点(Joinpoint)：类中的任意方法的运行时表示，可以简单理解为类中的方法
* 切入点(Pointcut) ：具有共性功能的方法的运行时表示，可以简单理解为具有共性功能的方法

注意：切入点对应的是被挖去了共性功能后的方法

* 通知(Advice) ：共性功能模块化，可以简单理解为将共性功能抽取出来制作成独立的方法
* 切面(Aspect) ：切入点与通知的对应关系，可以简单理解为被抽取的共性功能与共性功能被抽取位置对应的方法之间的关系
* 目标对象(Target Object) ：包含切入点的运行时对象，开发阶段制作的是目标对象对应的类
* AOP代理(AOP Proxy) ：使用AOP的代理机制创建目标对象运行时代理对象，完成原始的功能

注意：原始目标对象已经被挖去了共性功能，此时使用目标对象执行任务无法完成原始任务，使用AOP代理机制，创建一个代理对象来完成原始目标对象的功能

* 织入(Weaving) ：是一个将通知功能加入原始字节码的动态过程，共有三种方式，Spring使用的是运行时织入机制
* 引入(Introduction) ：一种特殊的机制，可以为一个类的字节码动态的加入变量或方法

## AOP工作流程

开发时，制作功能类（目标对象类），将其中方法的通用功能（通知）抽取出来，制作成独立的类（通知类），原始目标对象中的方法（切入点）不再书写通用功能。通用功能被抽取后，原始目标对象无法完成完整的业务逻辑，需要在运行时将通知加入到对应的位置执行。为了完成此操作，必须将切入点与通知进行一对一的对应关系设定（切面）。

## AOP运行流程

运行时，Spring监控切面中所配置切入点对应方法的执行，发现执行了匹配的方法，使用AOP代理机制，创建代理对象（AOP代理），将原始方法（切入点）与通知进行融合，形成完整的业务逻辑并进行运行，此过程称为织入。

## 在使用aop之前的实现方案

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

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-mm-dd HH:mm:ss");

Logger logger = Logger.*getLogger*(**this**.getClass());

**public** **void** sleep() {

logger.info(sdf.format(**new** Date())+",类"+**this**.getClass()+".sleep()被调用了");

System.***out***.println("睡觉...");

}

**public** **void** eat() {

logger.info(sdf.format(**new** Date())+",类"+**this**.getClass()+".eat()被调用了");

System.***out***.println("吃饭...");

}

}

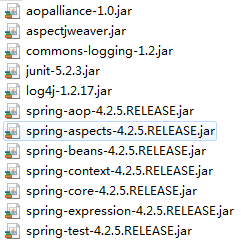
由此可见，在每个业务方法中，均需记录日志，而此功能是在重复实现，并且不属于具体的业务功能，让业务开发人员不能专注于业务的开发上。想想看，若有1000个业务类，每个类中10个业务方法，将是什么一个情况？

# 基于xml配置——使用AOP来实现日志记录

需求：需要将业务层的每个业务方法在什么时间被调用记录到日志文件中，记录的信息包含：什么时间、哪个类、哪个方法被调用。

## 准备工作

### 添加jar包



### 定义log4j的配置文件log4j.properties

该配置文件的名称必须是log4j.properties，且必须位于编译路径的根目录，一般放在resources目录下。

**log4j.rootLogger**=**DEBUG,Console,File***##控制台日志***log4j.appender.Console**=**org.apache.log4j.ConsoleAppender  
log4j.appender.Console.Target**=**System.out  
log4j.appender.Console.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.Console.layout.ConversionPattern**=**[%p][%t][%d{yyyy-MM-dd HH\:mm\:ss}][%C] - %m%n***#  
##普通文件日志***log4j.appender.File**=**org.apache.log4j.RollingFileAppender***#文件存储位置***log4j.appender.File.File**=**d:/log.log  
log4j.appender.File.MaxFileSize**=**10MB***#输出日志，如果换成DEBUG表示输出DEBUG以上级别日志***log4j.appender.File.Threshold**=**ALL  
log4j.appender.File.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.File.layout.ConversionPattern**=**[%p][%t][%d{yyyy-MM-dd HH\:mm\:ss}][%C] - %m%n***#mybatis日志配置信息***log4j.logger.java.sql.ResultSet**=**DEBUG  
log4j.logger.org.apache**=**DEBUG  
log4j.logger.java.sql.Connection**=**DEBUG  
log4j.logger.java.sql.Statement**=**DEBUG  
log4j.logger.java.sql.PreparedStatement**=**DEBUG**

## 使用AOP的方式来实现日志记录

### 定义切面类

将公共部分提取出来作为通知方法的执行部分,该方法的内容为业务方法中的公共部分内容。

//切面类(通知类)

**public** **class** LoggerAdvice {

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-mm-dd HH:mm:ss");

Logger logger = Logger.*getLogger*(**this**.getClass());

**public** **void** logBefore() {

String msg = String.*format*("【%s】方法被调用了", sdf.format(**new** Date()));

logger.info(msg);

}

}

### 创建业务类UserService

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

**public** **void** sleep() {

System.***out***.println("睡觉...");

}

**public** **void** eat() {

System.***out***.println("吃饭...");

}

}

### 在xml配置进行AOP配置装配

#### 首先需要加入aop命名空间，加入后的内容如下：

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

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

*http://www.springframework.org/schema/beans/spring-beans-4.2.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-4.2.xsd"*>

#### 定义AOP配置

<!-- 定义切面(通知)类的bean -->

<bean id=*"loggerAdvice"* class=*"com.lanqiao.aop.LoggerAdvice"*></bean>

<!-- 定义业务层的bean -->

<bean id=*"userService"* class=*"com.lanqiao.service.UserService"*></bean>

<!-- aop的配置 -->

<aop:config>

<!-- 配置切面(通知)，并指定该切面需要使用的通知类的引用 -->

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

<!-- 定义一个切入点，切入点的表达式指定哪些包下的哪些方法需要被织入通知，使用execution表达式 -->

<aop:pointcut expression=*"execution(\* com.lanqiao.service.\*.\*(..))"* id=*"pt"*/>

<!--

定义aop通知

before代表前置通知

pointcut-ref:指定切面引用

method:指定被织入的通知方法是什么 logBefore为LoggerAdvice中的方法名

-->

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

</aop:aspect>

</aop:config>

#### 通过测试类测试结果

**public** **class** Test1 {

**private** UserService userService;

@Before

**public** **void** testBefore() {

ApplicationContext applicationContext =

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

userService =(UserService) applicationContext.getBean(UserService.**class**);

}

@Test

**public** **void** testEat() {

userService.eat();

}

}



### 业务满足度？不满足需求

从执行结果来看，结果并不满足我们的需求，在日志中，并没有体现出目前是哪个类的哪个方法被调用了。

**实现方案：**在通知方法中将第一个参数定义为JoinPoint类型，当该增强处理方法被调用时，该JoinPoint参数就代表了织入增强处理的连接点。JoinPoint里包含了如下几个常用的方法：

* Object[] getArgs：返回目标方法的参数
* Signature getSignature：返回目标方法的签名，可以包含方法名和返回类型等信息
* Object getTarget：返回被织入增强处理的目标对象

    注意：当使用@Around处理时，我们需要将第一个参数定义为ProceedingJoinPoint类型，该类是JoinPoint的子类。

修改通知方法如下：

**public** **class** LoggerAdvice {

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-mm-dd HH:mm:ss");

Logger logger = Logger.*getLogger*(**this**.getClass());

**public** **void** logBefore(JoinPoint jp) {

Object target = jp.getTarget();

String targetClassName = target.getClass().getName();

String methodName = jp.getSignature().getName();

Object[] args = jp.getArgs();

String msg = "【"+sdf.format(**new** Date())+"】-"+targetClassName+"."+methodName+"("+Arrays.*toString*(args)+")";

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

logger.info(msg);

}

}

执行结果如下：



### 扩展说明

#### 通知类型

advice，即通知。通知在 AOP 中代表切面的工作，它定义了切面具体要做什么，何时做。

在 Spring 中共有 5 类通知类型，用来分别表示切面在不同时候要做的工作。

1. **前置通知（Before Advice）**：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**）。通过aop:before配置
2. **后置通知（After Advice）**：在切入点选择的连接点处的方法之后执行的通知（**无论方法执行是否成功都会被调用**）。通过aop:after配置
3. **后置返回通知（After returning Advice）**：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。通过aop: **after-returning**配置
4. **后置异常通知（After throwing Advice）**: 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。通过aop: **after-throwing**配置
5. **环绕通知（Around Advices）**：环绕着在切入点选择的连接点处的方法所执行的通知, 原始操作前后运行，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。通过ProceedingJoinPoint对象调用procee()方法完成对原始操作的调用，若不调用，则不执行原始操作。通过aop: **after-** **around**配置

#### 切入点表达式

格式：execution(切入点表达式)

execution([方法的访问控制修饰符] 方法的返回值 包名.类名/接口名.方法名(参数))

注意：方法的访问控制修饰符可以省略 ，方法的返回值一般用 \* 代表，表示不限定方法的返回类型。

常见符号：

\* 代表一个不限定的返回类型或者是包名或者是方法名或者是参数类型；

.. 代表0个或者多个

常见用法：

1. \* \*..\*.\*(..) 任意包下的任意类中的任意方法
2. \* com.lanqiao.service.\*.\*(..) com.lanqiao.service 包下的所有类的所有方法
3. \* com.lanqiao.service.\*..\*.\*(..) com.lanqiao.service 包及其子包下的类的所有方法

方法参数说明

add() 无参数

add(\*) 一个参数

add(int) 一个int型参数

add(\*,\*) 两个参数

add(\*,int) 两个参数，第一个任意，第二个int

add(..) 任意参数

add(\*,..) 至少一个参数

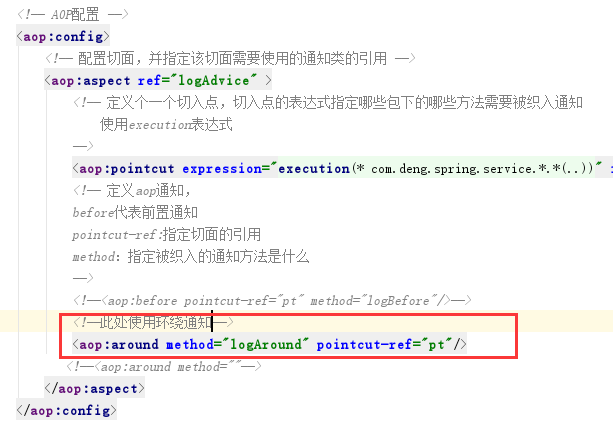
### 对方法访问日志功能升级，要求记录的是某个类的某个方法在什么时候执行，执行总计花费多少时间。

分析：前面的例子中实现了日志中记录哪个类的哪个方法在什么时候被调用，但是无法统计出该方法执行的时间。那么如何统计方法执行时间呢？其实就是方法开始执行到执行结束所花费的时间，涉及到原始方法前后，所以采用环绕通知。

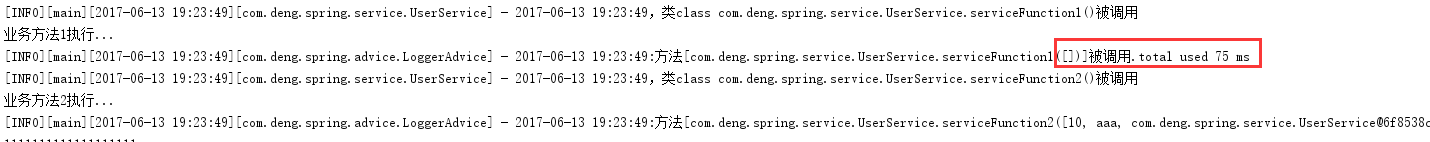
#### 编写通知方法

*/\*\*  
 \** ***@param jp*** *\** ***@return*** *必须定义object返回类型，且应该将原始方法执行的结果返回，若不返回，  
 \* 则原始方法的调用处无法获取方法的执行结果  
 \** ***@throws*** *Throwable  
 \*/***public** Object logAround(ProceedingJoinPoint jp) **throws** Throwable {  
 Object target = jp.getTarget();  
 String targetClassName = target.getClass().getName();  
 String methodName= jp.getSignature().getName();  
 Object []args = jp.getArgs();  
 String msg =String.*format*(**"%s:方法[%s.%s(%s)]被调用."**,**df**.format(**new** Date()),  
 targetClassName,methodName, Arrays.*toString*(args));  
 **long** start = System.*currentTimeMillis*();  
 *//执行原始方法，若没有下面的代码，则不会执行原始方法* **try** {  
 *// o为原始方法执行后的结果，必须返回* Object o = jp.proceed();  
 **return** o;  
 } **catch** (Throwable throwable) {  
 *//若原始方法抛出异常，则此处会捕获  
 //若此处不将该异常抛出，则调用原始方法的地方无法获取到业务方法的异常，所以该异常应该继续抛出  
 //除非此处有对异常的相关处理，处理后不需要再抛出* throwable.printStackTrace();  
 **throw** throwable;  
 }**finally** {  
 **long** end = System.*currentTimeMillis*();  
 **long** used = end-start;  
 msg = String.*format*(msg+**"total used %d ms"**,used);  
 **logger**.info(msg);  
 }  
}

#### 修改配置文件中的通知为采用环绕通知

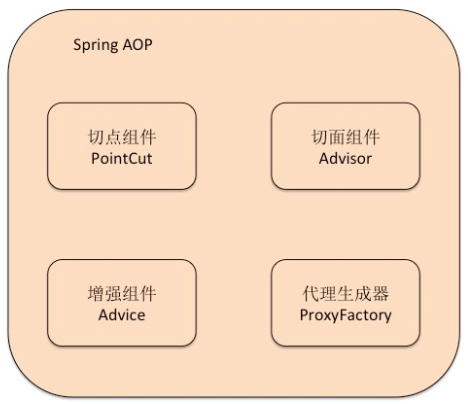


#### 执行看看结果

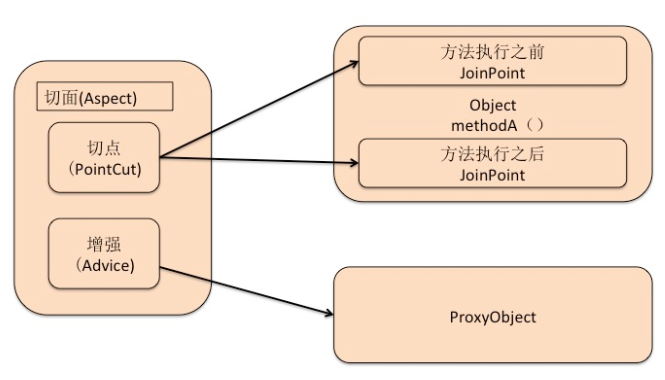


# 基于注解配置——使用AOP来实现日志记录

## Spring AOP的四大组件



## 流程结构



## **AspectJ支持5中类型的通知注解**

1. @Before: 前置通知，在方法执行之前执行
2. @After： 后置通知，在方法执行之后执行 (不管是否有异常，都会执行)
3. @AfterRuning: 返回通知，在方法返回结果之后执行 (只在没有异常时候才执行)
4. @AfterThrowing: 异常通知，在方法抛出异常之后
5. @Around: 环绕通知，围绕着方法执行

## 实现步骤

1. 引入jar包(spring的必要jar包 以及aspectj的jar包)
2. 业务方法HelloworldService (类上加上注解@Component,放入到spring ioc容器中)
3. 切面LogingAop (类上加上注解@Component使其加入到ioc容器中，还需要注解@Aspect，使其成为一个切面)
4. spring配置文件applicationContext.xml (扫描包，以及aop生成代理类的配置<aop:aspectj-autoproxy/>)
5. 测试