### AOP编程

**什么是面向切面编程AOP?**

在软件业，AOP为Aspect Oriented Programming的缩写，意为：[面向切面编程](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%88%87%E9%9D%A2%E7%BC%96%E7%A8%8B/6016335" \t "https://baike.baidu.com/item/AOP/_blank)，通过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91/3191547" \t "https://baike.baidu.com/item/AOP/_blank)方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是[OOP](https://baike.baidu.com/item/OOP" \t "https://baike.baidu.com/item/AOP/_blank)的延续，是软件开发中的一个热点，也是[Spring](https://baike.baidu.com/item/Spring" \t "https://baike.baidu.com/item/AOP/_blank)框架中的一个重要内容，是[函数式编程](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B/4035031" \t "https://baike.baidu.com/item/AOP/_blank)的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的[耦合度](https://baike.baidu.com/item/%E8%80%A6%E5%90%88%E5%BA%A6/2603938" \t "https://baike.baidu.com/item/AOP/_blank)降低，提高程序的可重用性，同时提高了开发的效率。

为什么学习AOP?

对程序进行增强:不修改源码的情况下,AOP可以进行权限的校验,日志记录,性能监控,事务控制

Spring的AOP的由来:

AOP最早由AOP联盟的组织提出的,指定了一套规范,Spring将Aop思想引入到框架中,必须遵守AOP联盟的规范

Aop图解:

|  |
| --- |
|  |
| **public interface** UserService {  **public void** select();//查找  **public void** save();//添加  **public void** update();//修改  **public void** delete();//删除 } |

### Spring底层的AOP原理:

动态代理(静态代理)

JDK动态代理: 面向接口的,只能对实现了接口的类产生代理

Cglib动态代理(类似于JavaSsit第三方代理技术):对没有实现接口的类产生代理对象(生成子类对象)

类实现了接口,Spring就用JDK动态代理,没有实现接口的,用Cglib动态代理,Spring底层可以自动切换

### Spring的AOP的简介

▪AOP思想最早是有AOP联盟组织提出的,Spring使用这种思想最好的框架.

▪Spring的AOP有自己的实现方法(非常繁琐)

▪使用第三方的AspectJ是一个AOP的扩建,Spring引入AspectJ作为自身AOP的开发.

▪Spring的AOP开发方式

▪Spring传统的方式(弃用)

▪Spring基于AspentJ的AOP开发(使用广泛)

### Spring的AOP开发的相关术语

## 1.通知（Advice）

　　就是你想要的功能，也就是上面说的 安全，事物，日志等。你给先定义好把，然后在想用的地方用一下。

## 2.连接点（JoinPoint）

　　 spring允许你使用通知的地方，基本每个方法的前，后（两者都有也行），或抛出异常时都可以是连接点，spring只支持方法连接点.和方法有关的前后（抛出异常），都是连接点。

## 3.切入点（Pointcut）

　　上面说的连接点的基础上，来定义切入点，你的一个类里，有5个方法，那就有5个连接点了，但是你并不想在所有方法附近都使用通知（使用叫织入），你只想让其中的几个，在调用这几个方法之前，之后或者抛出异常时干点什么，那么就用切点来定义这几个方法，让切点来筛选连接点，选中那几个你想要的方法。

## 4.切面（Aspect）

　　切面是通知和切入点的结合。通知说明了干什么和什么时候干（什么时候通过方法名中的before,after，around等就能知道），而切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。

## 5.引入（introduction）

　　允许我们向现有的类添加新方法属性。就是把切面（也就是新方法属性：通知定义的）用到目标类中

## 6.目标（target）

　　引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑，他可以在毫不知情的情况下，被咱们织入切面。而自己专注于业务本身的逻辑。

## 7.代理(proxy)

　　怎么实现整套aop机制的. 就是通过代理.织入增强功能后产生的代理对象

## 8.织入(weaving)

把切面应用到目标对象来创建新的代理对象的过程。有3种方式，spring采用的是运行时。

**图形解析:**

|  |
| --- |
|  |

### Spring的AOP开发入门(基于XML开发)

**▪ 引入基本开发包**

**▪ 引入AOP开发的相关包**

|  |
| --- |
|  |
| 第一个aop联盟  第二个aspectj的依赖  第三个AOP核心包  第四个Spring与aspects整合的包 |

**引入xml配置约束文件**

|  |
| --- |
| **官方在线api文档**  **https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#xsd-schemas-aop** |
|  |

|  |
| --- |
| <**aop:config**>  *<!-- 切入点: 指定要被增强哪个类里面哪个方法;\*:表示任何的修饰符或返回值类型 ; ..表示匹配add()方法中里面可以有任意参数 -->* <**aop:pointcut expression="execution(\* com.spring.service.UserService.add(..))" id="p1"**/>  *<!-- 配置切面 -->* <**aop:aspect ref="myAspectj"**>  *<!-- 把check通知的织入切入点p1(目标类的方法之前) -->* <**aop:before method="check" pointcut-ref="p1"**/>  </**aop:aspect**> </**aop:config**> |

**配置web项目**

**编写一个切面类**

配置切面类

|  |
| --- |
|  |

|  |
| --- |
|  |

### 通知的类型

**▪ 前置通知**before：在目标方法执行之前的操作

可以获取切入点的信息

**▪ 后置通知**after-returning**：**在目标方法执行之后的操作

**▪ 带返回值的后置通知**after-returning**：**在目标方法执行之后的操作,并获取目标方法的返回值

**▪ 环绕通知**around：在目标方法执行之前和之后进行操作

**▪ 扩展带参数: 环绕通知**around：在目标方法执行之前和之后进行操作

**▪ 异常抛出通知**after-throwing：在目标方法抛出异常时的操作(比如出现异常后进行回滚)。   
**▪ 最终通知**after：无论是否有异常都会执行的,相当于try-catch-finally中的finally块

|  |
| --- |
| *<?***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" xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**> *<!-- bean definitions here -->  <!--配置aop-->* <**aop:config**>  *<!--切入点: 指定某一个方法要作为切入点,\*表示save方法的返回值类型(包含void), .. 表示,参数列表;id表示切入点的别名-->* <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.save(..))" id="p1"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.select(..))" id="p2"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.selectRetrun(..))" id="p3"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.update(..))" id="p4"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.updateParam(..))" id="p5"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.delete(..))" id="p6"**/>   *<!--配置切面类: ref表示指定当前的切面类-->* <**aop:aspect ref="myAspect"**>  *<!--把check方法(通知)织入切入点p1之前(前置通知);-->* <**aop:before method="check" pointcut-ref="p1"**></**aop:before**>  *<!--把check方法(通知)织入切入点p2之后(后置通知);-->* <**aop:after-returning method="back" pointcut-ref="p2"**></**aop:after-returning**>  *<!--returning="obj"的obj一定要和通知中的参数一致,因为这里通过这个参数把切入点的返回值传给通知-->* <**aop:after-returning method="backReturn" pointcut-ref="p3" returning="obj"**></**aop:after-returning**>  *<!--环绕通知-->* <**aop:around method="around" pointcut-ref="p4"**></**aop:around**>  <**aop:around method="around2" pointcut-ref="p5"**></**aop:around**>  <**aop:after-throwing method="doException" pointcut-ref="p6" throwing="e"**></**aop:after-throwing**>  *<!--最终通知-->* <**aop:after method="finallyDo" pointcut-ref="p6"**></**aop:after**>  </**aop:aspect**>   </**aop:config**>  </**beans**> |
| *<?***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" xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**> *<!-- bean definitions here -->  <!--配置aop-->* <**aop:config**>  *<!--切入点: 指定某一个方法要作为切入点,\*表示save方法的返回值类型(包含void), .. 表示,参数列表;id表示切入点的别名-->* <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.save(..))" id="p1"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.select(..))" id="p2"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.selectRetrun(..))" id="p3"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.update(..))" id="p4"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.updateParam(..))" id="p5"**/>  <**aop:pointcut expression="execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.delete(..))" id="p6"**/>   *<!--配置切面类: ref表示指定当前的切面类-->* <**aop:aspect ref="myAspect"**>  *<!--把check方法(通知)织入切入点p1之前(前置通知);-->* <**aop:before method="check" pointcut-ref="p1"**></**aop:before**>  *<!--把check方法(通知)织入切入点p2之后(后置通知);-->* <**aop:after-returning method="back" pointcut-ref="p2"**></**aop:after-returning**>  *<!--returning="obj"的obj一定要和通知中的参数一致,因为这里通过这个参数把切入点的返回值传给通知-->* <**aop:after-returning method="backReturn" pointcut-ref="p3" returning="obj"**></**aop:after-returning**>  *<!--环绕通知-->* <**aop:around method="around" pointcut-ref="p4"**></**aop:around**>  <**aop:around method="around2" pointcut-ref="p5"**></**aop:around**>  <**aop:after-throwing method="doException" pointcut-ref="p6" throwing="e"**></**aop:after-throwing**>  *<!--最终通知-->* <**aop:after method="finallyDo" pointcut-ref="p6"**></**aop:after**>  </**aop:aspect**>   </**aop:config**>  </**beans**> |

### 切入点表达式语法

基于execution的函数完成的

语法

**execution(**[访问修饰符] 方法返回值 包名.类名.方法名(参数)**)**

com.spring.service.UserServiceImpl.add(..)

com.spring.service.UserServiceImpl.\*(..) 开发中用的最多的是这种,对当前类下所有的方法做增强处理(场景:事务处理)

com.spring.service.impl.add(..)

com.spring.service.impl.\*(..) 表示: com.spring.service.impl类下所有方法被增强

\* \*.\*.service.impl.add(..)没有包可以用\*代替

\* com.spring.\*.\*(..)表示com.spring包下所有的类,所有方法都被增强

### Spring的AOP基于注解的开发

### Spring注解的通知类型

开启注解的aop开发

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

前置通知:@Before(value = **"execution(\* com.spring.service.UserService.add(..))"**)

后置通知:@AfterReturning(value="execution()",returning="result")

环绕通知:@Around

异常抛出通知:@AfterThrowing(value="execution()",throwing="e")

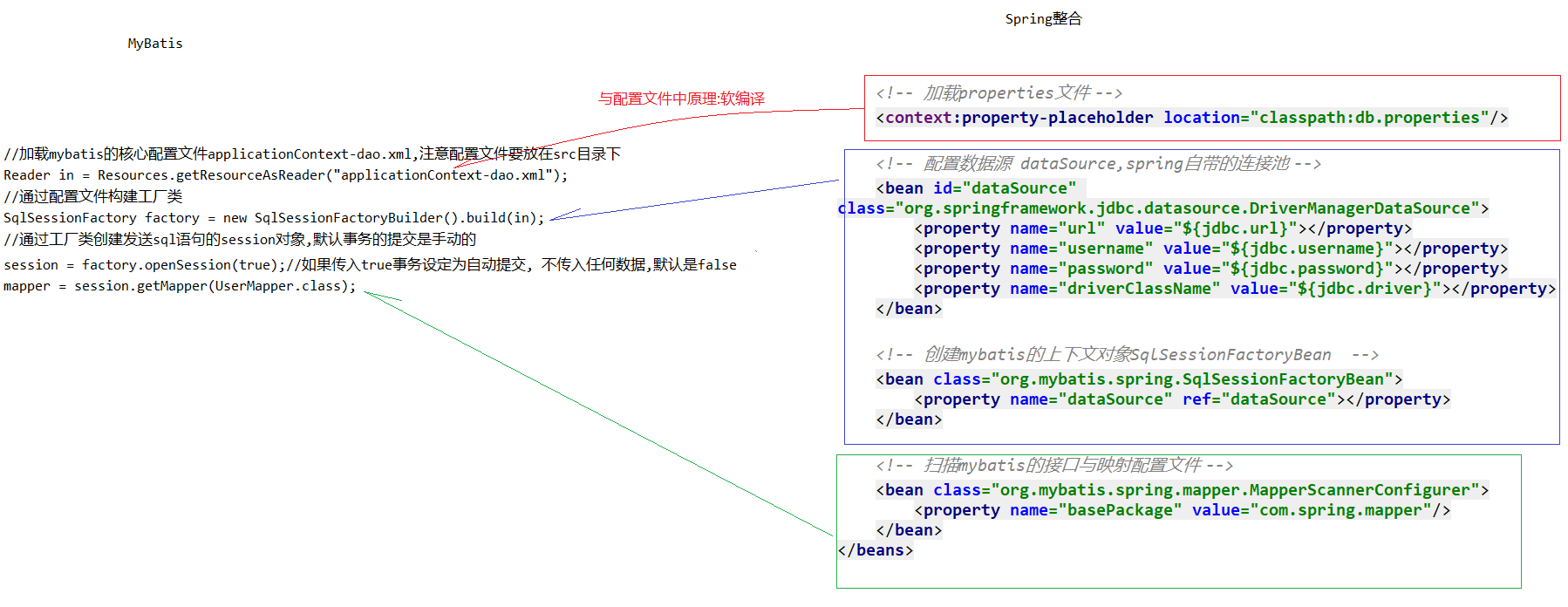
最终通知:@After

|  |
| --- |
| **package** com.xiangxue.aspect.aspectj;  **import** org.aspectj.lang.ProceedingJoinPoint; **import** org.aspectj.lang.annotation.\*; **import** org.springframework.stereotype.Component;  */\*\*  \** ***@Description*** *公共增强类(切面类)  \** ***@Author*** *SaiLing  \*/* @Component @Aspect */\*表示告诉spring当前类是切面类\*/* **public class** MyAspect{  *//校验身份* @Before(value = **"execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.save(..))"**) *//织入的动作,把当前通知织入到save之前* **public void** check(){ *//有增强功能的方法,称为: 通知* System.***out***.println(**"----之前,校验身份"**);   }  @AfterReturning(value = **"execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.select(..))"**)  **public void** back(){  System.***out***.println(**"------之后操作"**);  }  @AfterReturning(value = **"execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.selectRetrun(..))"**,returning =**"obj"**)  **public void** backReturn(Object obj){  System.***out***.println(**"带返回值的后置通知:"**+(String)obj);  }  @Around(value = **"execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.update(..))"**)  **public void** around(ProceedingJoinPoint point) **throws** Throwable { *//point接收切入点的对象* System.***out***.println(**"\*\*\*之前操作"**);  point.proceed();*//执行或进行操作的意思* System.***out***.println(**"\*\*\*之后操作"**);  }  **public void** around2(ProceedingJoinPoint point) **throws** Throwable { *//point接收切入点的对象* System.***out***.println(**"\*\*\*之前操作"**);  *//获取切入点的参数* Object[] args = point.getArgs();  System.***out***.println((String)args[0]);  System.***out***.println((String)args[1]);  point.proceed(args);*//执行了参数列表* System.***out***.println(**"\*\*\*之后操作"**);  }  @AfterThrowing(value = **"execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.delete(..))"**,throwing = **"e"**)  **public void** doException(Exception e){  System.***out***.println(**"有异常:"**+e);*//接收切入点抛出的异常信息* }  @After(value = **"execution(\* com.xiangxue.aspect.service.impl.UserServiceImpl.delete(..))"**)  **public void** finallyDo(){  System.***out***.println(**"最终通知,总会执行"**);  } } |

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"  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/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**> *<!-- bean definitions here -->  <!--开启aop注解-->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**> </**beans**> |

spring整合mybatis开发实现登录



3、何时使用JDK还是CGLIB？

1）如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。

2）如果目标对象实现了接口，也可以强制使用CGLIB实现AOP。

3）如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。

怎么强制使用Cglib代理?

方式一: 在配置文件中加

<**aop:aspectj-autoproxy proxy-target-class="true"**/> *<!--强制使用Cglib代理-->*

|  |
| --- |
| *<?***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" xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**> *<!-- bean definitions here -->  <!--开启aop注解-->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**>  *<!--强制使用Cglib的动态代理-->* <**aop:aspectj-autoproxy proxy-target-class="true"**/> *<!--强制使用Cglib代理-->* </**beans**> |

方式二: 注解的方式(默认开启的)，不需要特意加

@EnableAspectJAutoProxy(proxyTargetClass = **true**)