# AOP

## AOP理解

AOP，面向切面编程，是面向对象编程(OOP)的一种有益补充。

通俗的理解，就是在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。

  但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志（为了低耦合，保持类的独立性，在每个类中都写上日志方法）。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。

 也许有人会说，那好办啊，我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？**这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。**

  一般而言，我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。有了AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。

## AOP应用

1. 降低耦合；在类中引用通用功能类，会导致类之间耦合，AOP可以在指定位置织入切面，类之间没有引用关系，降低耦合。
2. 减少重复代码；AOP可以把通用的功能分离出来，然后在需要使用通用功能的地方使用切面切入。
3. 功能扩展；AOP的通知本身也被译为增强，可以在方法前后加入代码增强原方法功能，起到扩展方法作用。

## AOP术语

1. 通知（Advice）：在连接点上执行的行为，就是你想要的功能，比如安全，事物，日志等。你需要先定义好，然后在想用的地方用一下。
2. 连接点（JoinPoint）：就是spring允许你使用通知的地方，也就是你在哪个地方想使用日志。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.其他如aspectJ还可以让你在构造器或属性注入时都行，不过那不是咱关注的，只要记住，和方法有关的前前后后（抛出异常），都是连接点。
3. 切入点（PointCut）：切入点是在连接点的基础上定义的。比如一个类有十几个方法，，那就有几十个连接点了对把，但是你并不想在所有方法附近都使用通知（使用叫织入，以后再说），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。然后你可以将这几个连接点通过配置或注解定义为切入点。
4. 切面（Aspect）：切面是横切关注点的模块化，可以认为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。也就是把切面（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，也就是要被切面切入的类，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. 织入（weaving）：织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程。织入可以在编译期、类装载期、运行期进行，spring采用的是运行时。
8. 代理（proxy）：AOP框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是通过代理来对目标对象应用切面。在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。

## 通知类型

1. 前置通知：在目标类中切入点关联的方法执行之前执行
2. 后置通知：在目标类中切入点关联的方法执行之后执行
3. 后置返回通知：在目标类中切入点关联的方法执行之后执行，且要求切入点关联的方法正常执行
4. 后置异常通知：在目标类中切入点关联的方法抛出异常时触发，执行通知的代码
5. 后置最终通知：无论目标类中切入点关联的方法是抛出异常还是正常执行，都会执行。类似于Java中的finally块语句。
6. 环绕通知：在目标类中切入点关联的方法执行之前和执行之后触发，环绕方法首尾

## AOP代理机制

spring用代理类包裹切面，把他们织入到Spring管理的bean中。也就是说代理类伪装成目标类，它会截取对目标类中方法的调用，让调用者对目标类的调用都先变成调用伪装类，伪装类中就先执行了切面，再把调用转发给真正的目标bean。

现在可以自己想一想，怎么搞出来这个伪装类，才不会被调用者发现（过JVM的检查，JAVA是强类型检查，哪里都要检查类型）。

　　1、实现和目标类相同的接口，我也实现和你一样的接口，反正上层都是接口级别的调用，这样我就伪装成了和目标类一样的类（实现了同一接口，咱是兄弟了），也就逃过了类型检查，到java运行期的时候，利用多态的后期绑定（所以spring采用运行时），伪装类（代理类）就变成了接口的真正实现，而他里面包裹了真实的那个目标类，最后实现具体功能的还是目标类，只不过伪装类在之前干了点事情（写日志，安全检查，事物等）。

　　这就好比，一个人让你办件事，每次这个时候，你弟弟就会先出来，当然他分不出来了，以为是你，你这个弟弟虽然办不了这事，但是他知道你能办，所以就答应下来了，并且收了点礼物（写日志），收完礼物了，给把事给人家办了啊，所以你弟弟又找你这个哥哥来了，最后把这是办了的还是你自己。但是你自己并不知道你弟弟已经收礼物了，你只是专心把这件事情做好。

　　顺着这个思路想，要是本身这个类就没实现一个接口呢，你怎么伪装我，我就压根没有机会让你搞出这个双胞胎的弟弟，那么就用第2种代理方式，创建一个目标类的子类，生个儿子，让儿子伪装我

　　2、生成子类调用，这次用子类来做为伪装类，当然这样也能逃过JVM的强类型检查，我继承的吗，当然查不出来了，子类重写了目标类的所有方法，当然在这些重写的方法中，不仅实现了目标类的功能，还在这些功能之前，实现了一些其他的（写日志，安全检查，事物等）。

这次的对比就是，儿子先从爸爸那把本事都学会了，所有人都找儿子办事情，但是儿子每次办和爸爸同样的事之前，都要收点小礼物（写日志），然后才去办真正的事。当然爸爸是不知道儿子这么干的了。这里就有件事情要说，某些本事是爸爸独有的(final的)，儿子学不了，学不了就办不了这件事，办不了这个事情，自然就不能收人家礼了。

前一种兄弟模式，spring会使用JDK的java.lang.reflect.Proxy类，它允许Spring动态生成一个新类来实现必要的接口，织入通知，并且把对这些接口的任何调用都转发到目标类。

　　后一种父子模式，spring使用CGLIB库生成目标类的一个子类，在创建这个子类的时候，spring织入通知，并且把对这个子类的调用委托到目标类。

　　相比之下，还是兄弟模式好些，他能更好的实现松耦合，尤其在今天都高喊着面向接口编程的情况下，父子模式只是在没有实现接口的时候，也能织入通知，应当做一种例外。

## 基于配置文件的AOP

Spring在2.0之后的版本中，提供了在XML配置文件中使用<aop:config>标签定义切面、切入点和通知相关内容的支持。通过配置，我们可以对目标类中切入点关联的方法进行增强。

在Spring配置文件中，所有与AOP相关的定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，且这几个标签的配置顺序不可变。

<aop:pointcut>：用来定义切入点，该切入点可以重用；

<aop:advisor>：用来定义只有一个通知和一个切入点的切面；

<aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和advisor的区别就在此，advisor只包含一个通知和一个切入点。

advisor除了在进行事务控制的情况下，其他情况下一般不建议使用。该方式属于侵入式设计，必须实现通知API

### 切入点表达式

在声明切入点时，需要通过expression表达式匹配指定类中的指定方法作为切人点。

<**aop:pointcut id="mycut" expression="execution(\* \*.\*(..))"**></**aop:pointcut**>

<**aop:pointcut id="mycut" expression="execution(\* com.wei.test.AOPTest.\*.\*(..))"**></**aop:pointcut**>

execution(\* \*.\*(..))

1. execution 声明表达式，不能忽略
2. \* 第一个\*用来匹配任意返回值类型（即切入点的方法返回值为任意类型）
3. 第一个\*后面要有一个空格，否则表达式匹配不到切入点
4. 第二个\*匹配任意包中的任意类
5. 第三个\*表示匹配类中的所有方法
6. (..)表示方法中可以存在任意个参数

以上\*可以用具体的包名、类名、方法名代替

### 切入点声明方式

切入点一般是Java类中的某个方法，在配置文件中有以下3中声明方式：

1. 在<aop:config>标签下使用<aop:pointcut>声明一个切入点。

该切入点可以被多个切面使用，对于需要共享使用的切入点最好使用该方式。该切入点中：

id属性指定切入点所在目标类的名字（应当具有唯一性，在定义通知时通过该ID引用该切入点）；

expression属性指定切入点表达式（expression表达式匹配指定类中的指定方法，也就是决定哪些方法被定义为切入点）

<**aop:config**>  
 <**aop:pointcut id="pointcut" expression="execution(\* cn.javass..\*.\*(..))"**/>  
 <**aop:aspect ref="aspectSupportBean"**>  
 <**aop:before pointcut-ref="pointcut" method="before"**/>  
 </**aop:aspect**>  
</**aop:config**>

1. 在<aop:aspect>标签下使用<aop:pointcut>声明一个切入点。

该切入点也可以被多个切面引用，但一般只用于当前<aop:aspect>标签定义的切面。如果想多个切面共享，应该使用方式1中的切入点声明方式。

<**aop:config**>  
 <**aop:aspect ref="aspectSupportBean"**>  
 <**aop:pointcut id=" pointcut" expression="execution(\* cn.javass..\*.\*(..))"**/>  
 <**aop:before pointcut-ref="pointcut" method="before"**/>  
 </**aop:aspect**>  
</**aop:config**>

1. 在声明通知时通过pointcut属性指定切入点表达式声明一个匿名切入点。

该切入点是匿名切入点，只被声明处的通知中使用。

<**aop:config**>  
 <**aop:aspect ref="aspectSupportBean"**>  
 <**aop:after pointcut="execution(\* cn.javass..\*.\*(..))" method="afterFinallyAdvice"**/>  
 </**aop:aspect**>  
</**aop:config**>

### 前置通知

在切入点关联的方法之前执行，通过<aop:aspect>标签下的<aop:before>标签声明

<**aop:before pointcut="切入点表达式" pointcut-ref="切入点Bean引用"  
 method="前置通知实现方法名"  
 arg-names="前置通知实现方法参数列表参数名字"**/>

pointcut和pointcut-ref：二者选一，指定切入点（pointcut-ref通过已定义切入点的ID引用）

method：指定前置通知关联的方法名，如果是多态需要加上参数类型，多个用“，”隔开，如beforeAdvice(java.lang.String)；

arg-names：指定通知关联方法的参数名字，多个用“，”分隔，可选，类似于构造器注入中的参数名注入限制：在class文件中没生成变量调试信息是获取不到方法参数名字的，因此只有在类没生成变量调试信息时才需要使用arg-names属性来指定参数名，如arg-names="param"表示通知实现方法的参数列表的第一个参数名字为“param”。

<**bean id="targetClass" class="com.wei.test.AOPTest.TargetClass"**></**bean**>  
<**bean id="aspect" class="com.wei.test.AOPTest.HelloWorldAspect"**></**bean**>  
  
<**aop:config**>  
 <**aop:pointcut id="mycut" expression="execution(\* \*.\*(..)) and args(param)"**></**aop:pointcut**>  
 <**aop:aspect ref="aspect"**>  
 <**aop:before method="beforeAdvisor" pointcut-ref="mycut" arg-names="param"**></**aop:before**>  
 </**aop:aspect**>  
</**aop:config**>

1. pointcut表达式后跟的参数名和arg-names后跟的参数名要保持一致；
2. 如果通知方法中有参数，在声明通知时必须在pointcut表达式和arg-names标签中配置参数；
3. 如果目标类中切入点方法中没有参数，而配置文件中声明的切入点中execution表达式带参数时，即使前半部分匹配，通知也不会被调用。可以认为此时execution表达式增加了一个条件，就是切入点关联的方法中必须带有参数；
4. 无论几个参数，只要满足1的条件，切入点方法和通知方法中参数名可以不一致（最好保持一致）
5. 多态？？

### 后置返回通知

在切入点选择的方法正常返回时执行，通过<aop:aspect>标签下的<aop:after-returning>标签声明

<aop:after-returning pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置返回通知实现方法名"** arg-names=**"后置返回通知实现方法参数列表参数名字"** returning=**"返回值对应的后置返回通知实现方法参数名"** />

returning：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法执行正常返回后，将把目标方法返回值传给通知方法；returning限定了只有目标方法返回值匹配与通知方法相应参数类型时才能执行后置返回通知，否则不执行，对于returning对应的通知方法参数为Object类型将匹配任何目标返回值。

### 后置异常通知

在切入点选择的方法抛出异常时执行，通过<aop:aspect>标签下的<aop:after-throwing>标签声明

<aop:after-throwing pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置异常通知实现方法名"** arg-names=**"后置异常通知实现方法参数列表参数名字"** throwing=**"将抛出的异常赋值给的通知实现方法参数名"**/>

throwing：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法抛出异常返回后，将把目标方法抛出的异常传给通知方法；throwing限定了只有目标方法抛出的异常匹配与通知方法相应参数异常类型时才能执行后置异常通知，否则不执行，对于throwing对应的通知方法参数为Throwable类型将匹配任何异常。

### 后置最终通知

在切入点选择的方法返回时执行，不管是正常返回还是抛出异常都执行，通过<aop:aspect>标签下的<aop:after >标签声明

<aop:after pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置最终通知实现方法名"** arg-names=**"后置最终通知实现方法参数列表参数名字"**/>

### 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，环绕通知非常强大，可以决定目标方法是否执行，什么时候执行，执行时是否需要替换方法参数，执行完毕是否需要替换返回值，可通过<aop:aspect>标签下的<aop:around >标签声明

<aop:around pointcut=**"切入点表达式"** pointcut-ref=**"切入点Bean引用"** method=**"后置最终通知实现方法名"** arg-names=**"后置最终通知实现方法参数列表参数名字"**/>

环绕通知第一个参数必须是org.aspectj.lang.ProceedingJoinPoint类型，在通知实现方法内部使用ProceedingJoinPoint的proceed()方法使目标方法执行，proceed 方法可以传入可选的Object[]数组，该数组的值将被作为目标方法执行时的参数。

### 引入

Spring引入允许为目标对象引入新的接口，通过在< aop:aspect>标签内使用< aop:declare-parents>标签进行引入，定义方式如下

<aop:declare-parents  
 types-matching=**"AspectJ语法类型表达式"** implement-**interface**=引入的接口**"   
 default**-impl=**"引入接口的默认实现"** delegate-ref=**"引入接口的默认实现Bean引用"**/>

types-matching：匹配需要引入接口的目标对象的AspectJ语法类型表达式；

implement-interface：定义需要引入的接口；

default-impl和delegate-ref：定义引入接口的默认实现，二者选一，default-impl是接口的默认实现类全限定名，而delegate-ref是默认的实现的委托Bean名；

## AspectJ实现AOP

Spring除了支持Schema方式配置AOP，还支持注解方式：使用@AspectJ风格的切面声明。

### 启动AspectJ注解支持

Spring默认情况下不支持@AspectJ风格的切面声明。如果要使用@AspectJ注解声明切面，需要在Spring的配置文件中启动注解支持：

<aop:aspectj-autoproxy/>

添加如上配置后，Spring才能够映射@AspectJ注解并将切面应用到目标对象。

使用@AspectJ风格的注解声明切面，可以大大减少AOP相关的配置。

### 声明切面

@AspectJ风格声明切面非常简单。如果我们想将一个类声明为切面，直接在这个类上加一个@Aspect注解即可。但要注意，要确保声明为切面的这个类能够被Spring容器加载。

@Aspect()   
Public **class** Aspect{   
……  
}

### 声明切入点

@AspectJ风格声明切入点时，只需要在方法上加@Pointcut注解即可。另外，使用@Pointcut注解修饰的方法返回类型必须是void。

@Pointcut(value="切入点表达式", argNames = "参数名列表")  
public void pointcutName(……) {}

value：指定切入点表达式；

argNames：指定命名切入点方法参数列表参数名字，可以有多个用“，”分隔，这些参数将传递给通知方法同名的参数，同时比如切入点表达式“args(param)”将匹配参数类型为命名切入点方法同名参数指定的参数类型。

pointcutName：切入点名字，可以使用该名字进行引用该切入点表达式。

### 声明通知

@AspectJ和通过XML配置文件声明AOP一样，也支持5种通知类型。

#### 前置通知

前置通知通过@Before注解声明

@Before(value = "切入点表达式或命名切入点", argNames = "参数列表参数名")

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义

#### 后置返回通知

后置返回通知通过@AfterReturning注解声明

@AfterReturning(  
 value="切入点表达式或命名切入点",  
 pointcut="切入点表达式或命名切入点",  
 argNames="参数列表参数名",  
 returning="返回值对应参数名")

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

returning：与Schema方式配置中的同义。

#### 后置异常通知

@AfterThrowing注解声明

@AfterThrowing (  
 value="切入点表达式或命名切入点",  
 pointcut="切入点表达式或命名切入点",  
 argNames="参数列表参数名",  
 throwing="异常对应参数名")

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

throwing：与Schema方式配置中的同义

#### 后置最终通知

@After注解声明

@After (  
 value="切入点表达式或命名切入点",  
 argNames="参数列表参数名")

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

#### 环绕通知

@Around注解声明

@Around (  
 value="切入点表达式或命名切入点",  
 argNames="参数列表参数名")

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

### 声明引入

@DeclareParents声明

@DeclareParents(value=" AspectJ语法类型表达式",

defaultImpl=引入接口的默认实现类)  
private Interface interface;

value：匹配需要引入接口的目标对象的AspectJ语法类型表达式；与Schema方式中的types-matching属性同义；

private Interface interface：指定需要引入的接口；

defaultImpl：指定引入接口的默认实现类，没有与Schema方式中的delegate-ref属性同义的定义方式；

## 通知参数

通知可以对切入点关联的方法进行增强，方法中基本都是有参数的。如何获取方法参数，并将这些参数传给通知进行处理是非常重要的一点。

接下来介绍两种AOP获取切入点关联方法参数的方式：

1. Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

public interface JoinPoint {  
 String toString(); //连接点所在位置的相关信息  
 String toShortString(); //连接点所在位置的简短相关信息  
 String toLongString(); //连接点所在位置的全部相关信息  
 Object getThis(); //返回AOP代理对象  
 Object getTarget(); //返回目标对象  
 Object[] getArgs(); //返回被通知方法参数列表  
 Signature getSignature(); //返回当前连接点签名  
 SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置  
 String getKind(); //连接点类型  
 StaticPart getStaticPart(); //返回连接点静态部分  
}

ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

public interface ProceedingJoinPoint extends JoinPoint {  
 public Object proceed() throws Throwable;  
 public Object proceed(Object[] args) throws Throwable;  
}

JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

public interface StaticPart {  
 Signature getSignature(); //返回当前连接点签名  
 String getKind(); //连接点类型  
 int getId(); //唯一标识  
 String toString(); //连接点所在位置的相关信息  
 String toShortString(); //连接点所在位置的简短相关信息  
 String toLongString(); //连接点所在位置的全部相关信息  
}

1. 自动获取：通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

2.1、如果我们通过“argNames”属性指定了参数名

2.2、如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

2.3、如果“class文件中含有变量调试信息”将使用这些方法签名中的参数名来确定参数名；

2.4、如果没有“class文件中含有变量调试信息”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数

## 通知顺序

如果我们有多个通知想要在同一连接点执行，那执行顺序如何确定呢？Spring AOP使用AspectJ的优先级规则来确定通知执行顺序。总共有两种情况：同一切面中通知执行顺序、不同切面中的通知执行顺序。

### 同一切面中通知执行顺序

### 不同切面中的通知执行顺序

当定义在不同切面的相同类型的通知需要在同一个连接点执行，如果没指定切面的执行顺序，这两个通知的执行顺序将是未知的。

如果需要他们顺序执行，可以通过指定切面的优先级来控制通知的执行顺序。

Spring中可以通过在切面实现类上实现org.springframework.core.Ordered接口或使用Order注解来指定切面优先级。在多个切面中，Ordered.getValue()方法返回值（或者注解值）较小值的那个切面拥有较高优先级

## 切面实例化模型

所谓切面实例化模型指何时实例化切面。

Spring AOP支持AspectJ的singleton、perthis、pertarget实例化模型（目前不支持percflow、percflowbelow 和pertypewithin）。