**AOP技术学习分享**

**——于申明**

**一：概念和术语**

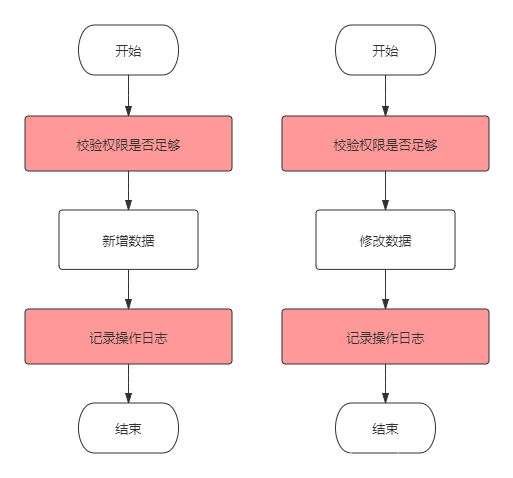
**1.什么是AOP？**

AOP又名Aspect Oriented Programming 意为 ‘面向切面编程’通过预编译和运行期间动态代理来实现程序功能的统一维护的一种技术。AOP思想是OOP(面向对象)的延续 在 OOP 中, 我们以类(class)作为我们的基本单元, 而 AOP中的基本单元是 Aspect(切面)，AOP是软件行业的热点，也是Spring框架中的一个重要内容，是函数式编程的一种延伸范式,

总结：这种在运行时生成代理对象来织入的，还可以在编译期、类加载期织入，动态地将代码在不改变原有的逻辑情况下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。

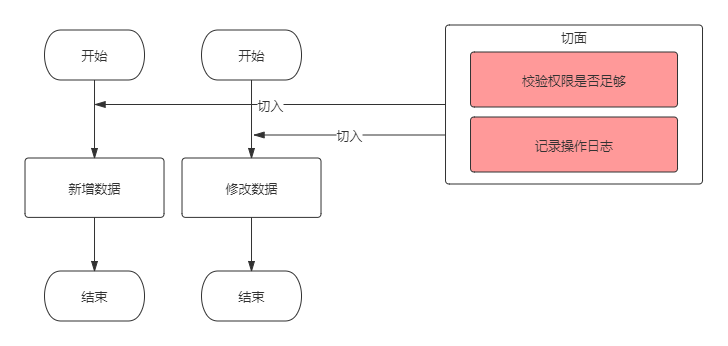
**2.为什么要用AOP？**

假如一个流程分三个步骤，分别是X,A,Y，另一个流程的三个步骤是X,B,Y。X是检验权限，A是新增数据，B是修改数据，C是记录操作日志。写在程序里，两个方法体分别是XAY和XBY。



显然，这出现了重复，违反了DRY原则（不要写重复的代码）。

于是就可以用AOP：

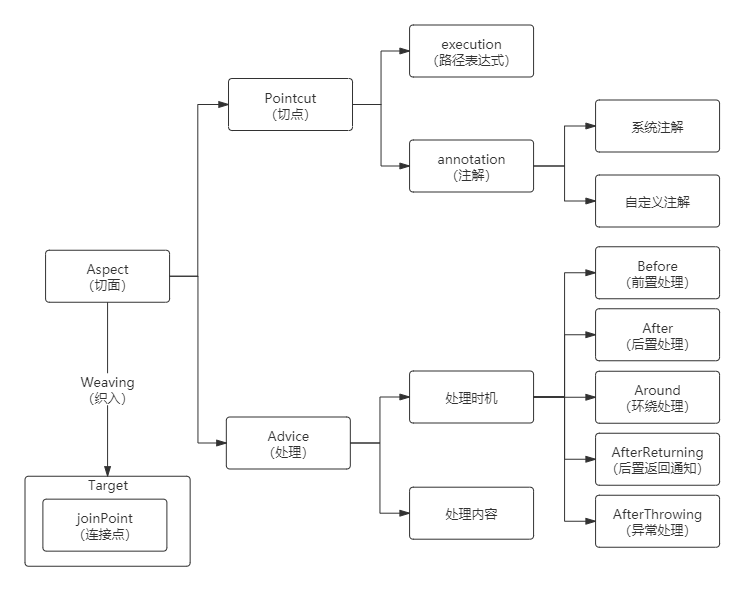


OOP从横向上区分出一个个的类来，而AOP则从纵向上向对象中加入特定的代码。对原有代码毫无入侵性，这就是AOP的好处了，把和主业务无关的事情，放到代码外面去做。从技术上来说，AOP基本上是通过代理机制实现的。

在Spring AOP中业务逻辑仅仅只关注业务本身，将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码。

AOP相当于一个拦截器，去拦截一些处理，例如：当一个方法执行的时候，Spring 能够拦截正在执行的方法，在方法执行的前或者后增加额外的功能和处理，就是我们希望通过使用AOP来对我们的代码进行耦合度的降低 把原本杂乱交错的关系给分离开来 进而改变这些行为的时候不会因为杂乱的关系互相影响。

AOP体系：



**3.术语**

AOP各术语的含义：



**二：AOP原理**

**1.AOP代理**

AOP 代理其实是由 AOP 框架动态生成的一个对象，该对象可作为目标对象使用。

AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异：AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。

Spring 的 AOP 代理由 Spring 的 IoC 容器负责生成、管理，其依赖关系也由 IoC 容器负责管理。因此，AOP 代理可以直接使用容器中的其他 Bean 实例作为目标，这种关系可由 IoC 容器的依赖注入提供。

aop开发时，其中需要参与开发的只有 3 个部分：

1定义普通业务组件。

2定义切入点，一个切入点可能横切多个业务组件。

3定义增强处理，增强处理就是在 AOP 框架为普通业务组件织入的处理动作。

**2.两种动态代理方式**：

Spring默认采取的动态代理机制实现AOP，当动态代理不可用时（代理类无接口）会使用CGlib机制。

Spring提供了两种方式来生成代理对象: JDKProxy和CGlib，具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。

1、JDK 动态代理

JDK动态代理主要涉及到java.lang.reflect包中的两个类：Proxy和InvocationHandler。InvocationHandler是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态将横切逻辑和业务逻辑编制在一起。

Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

2、CGLib动态代理

CGLib全称为Code Generation Library，是一个强大的高性能，高质量的代码生成类库，可以在运行期扩展Java类与实现Java接口，CGLib封装了asm，可以再运行期动态生成新的class。和JDK动态代理相比较：JDK创建代理有一个限制，就是只能为接口创建代理实例，而对于没有通过接口定义业务方法的类，则可以通过CGLib创建动态代理。

**三：AOP应用**

1. **首先导入依赖**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

<scope>test</scope>

</dependency>

2.**定义 aspect(切面)**

@Aspect:作用是把当前类标识为一个切面供容器读取

如果一个类被打上了@Aspect就代表着他是一个切面类

当使用注解 @Aspect 标注一个 Bean 后, 那么 Spring 框架会自动收集这些 Bean, 并添加到 Spring AOP 中。但仅仅使用@Aspect 注解, 并不能将一个 Java 对象转换为 Bean, 因此我们还需要使用类似 @Component 之类的注解。

@Aspect//定义该类为切面

@Component

public class AspectDemo

{

}

注意, 如果一个 类被@Aspect 标注, 则这个类就不能是其他 aspect 的 advised object（目标对象） 了, 因为使用 @Aspect 后, 这个类就会被排除在 auto-proxying（自动代理） 机制之外。

**3.声明pointout切入点**

一个 Pointcut 的声明由两部分组成:

一个方法签名, 包括方法名和相关参数

一个 Pointcut 表达式, 用来指定哪些方法执行是我们感兴趣的(即因此可以织入 Advice ).

**切点标志符(designator)：**

切点表达式由标志符(designator)和操作参数组成. 如 “execution(\* greetTo(..))” 的切点表达式, execution 就是 标志符, 而圆括号里的 \* greetTo(..) 就是操作参数。

**execution(执行):**

execution表达式在切点的应用，\*execution(com.hecl.zhenghe.Controller..\*.\*(..)各参数解释如下表格：



package com.example.javalogframe.text;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Aspect//定义该类为切面

@Component

public class AspectDemo

{

@Pointcut("execution(\* com.example.javalogframe.text.Jui\_LogDemo.\*(..))")//切点表达式

public void aspectPointcut()

{

}

}

上面我们简单地定义了一个 Pointcut , 这个 Pointcut 所描述的是: 匹配所有在包 com.example.javalogframe.text.Jui\_LogDemo 下的所有方法的执行.

**within(内)：**

匹配特定包下的所有 Join Point, 例如 within(com.xys.\*) 表示 com.xys 包中的所有连接点, 即包中的所有类的所有方法. 而 within(com.xys.demo2.\*Service) 表示在 com.xys.demo2包中所有以Service 结尾的类的所有的连接点.

@Pointcut("within(com.xys.demo2.\*)")

public void pointcut2() {

}

**this 与 target**

this 的作用是匹配一个 bean, 这个 bean(Spring AOP proxy) 是一个给定类型的实例(instance of). 而 target 匹配的是一个目标对象(target object, 即需要织入 Advice的原始的类), 此对象是一个给定类型的实例(instance of).

**bean**

匹配 bean 名字为指定值的 bean 下的所有方法, 例如:

bean(\*Service) // 匹配名字后缀为 Service 的 bean 下的所有方法

bean(myService) // 匹配名字为 myService 的 bean 下的所有方法

**args**

匹配参数满足要求的的方法.

**也可以用@annotation**

匹配由指定注解所标注的方法, 例如:

@Pointcut("@annotation(com.xys.demo1.AuthChecker)")

public void pointcut()

{

}

则匹配由注解 AuthChecker 所标注的方法.

**4.声明 Advice**

Advice是和一个 Pointcut 表达式关联在一起的, 并且会在匹配的 Join Point 的方法执行的前/后/周围 运行. Pointcut 表达式可以是简单的一个 Pointcut 名字的引用, 或者是完整的 Pointcut 表达式.

**Advice的5种增强处理的应用：**

**首先以@Before为例：**

@Component

@Aspect

public class BeforeAspectTest {

// 定义一个 Pointcut, 使用 切点表达式函数 来描述对哪些 Join point 使用 advise.

@Pointcut("execution(\* com.xys.service.UserService.\*(..))")

public void dataAccessOperation() {

}

}

@Component

@Aspect

public class AdviseDefine {

// 定义 advise

@Before("com.xys.aspect.PointcutDefine.dataAccessOperation()")

public void doBeforeAccessCheck(JoinPoint joinPoint) {

System.out.println("\*\*\*\*\*Before advise, method: " + joinPoint.getSignature().toShortString() + " \*\*\*\*\*");

}

}

这里, @Before 引用了一个 Pointcut , 即 “com.xys.aspect.PointcutDefine.dataAccessOperation()” 是一个 Pointcut 的名字.

如果我们在 Advice 在内置 Pointcut , 则可以:

@Component

@Aspect

public class AdviseDefine {

// 将 pointcut 和 advice 同时定义

@Before("within(com.xys.service..\*)")

publicvoiddoAccessCheck(JoinPoint joinPoint)

{

System.out.println("\*\*\*\*\*doAccessCheck, Before advise, method: " + joinPoint.getSignature().toShortString() + " \*\*\*\*\*");

}

}

**Around Advice:**

around Advice 比较特别, 它可以在一个方法的之前之前和之后添加不同的操作, 并且甚至可以决定何时, 如何, 是否调用匹配到的方法.

在看环绕增强时我们需要先了解两个对象 joinpoint(连接点) 和 proceedingjoinpoint(进行连接点)

**JoinPoint**

因为我们前面了解到 joinpoint 是连接点的意思 所以 JoinPoint对象则包含了和切入相关的很多信息。比如切入点的对象，方法，属性等。我们可以通过反射的方式获取这些点的状态和信息，用于追踪tracing和记录logging应用信息。

通过JoinPoint对象可以获取到下面信息

# 返回目标对象，即被代理的对象

Object getTarget();

# 返回切入点的参数

Object[] getArgs();

# 返回切入点的Signature

Signature getSignature();

# 返回切入的类型，比如method-call，field-get等等，感觉不重要

String getKind();

**ProceedingJoinPoint**

而proceedingjoinpoint 继承了 JoinPoint。是在JoinPoint的基础上暴露出 proceed 这个方法。proceed很重要，这个是aop代理链执行的方法。

环绕通知=前置+目标方法执行+后置通知，proceed方法就是用于启动目标方法执行的

暴露出这个方法，就能支持 aop:around 这种切面（而其他的几种切面只需要用到JoinPoint，，这也是环绕通知和前置、后置通知方法的一个最大区别。这跟切面类型有关）， 能决定是否走代理链还是走自己拦截的其他逻辑。

例如：

//在使用时，我们传入了 ProceedingJoinPoint 类型的参数，

//这个对象是必须要有的，并且需要调用 ProceedingJoinPoint 的 proceed() 方法

//如果没有调用joinPoint.proceed()则会阻塞目标方法

@Around("LogAspect()")

public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable{

log.info("调用方法前先验证权限是否足够");

Object result = joinPoint.proceed();

log.info("方法结束时间-----------" + System.currentTimeMillis());

return result;

}

**切面日志类具体实例：**

@Aspect

@Component

@Slf4j

public class ALiangXLogAspect {

//配置切入点

@Pointcut(value = "@annotation(com.hecl.zhenghe.annotation.ALiangXLog)")

//在Pointcut我们还可以限制对哪个bean进行增强以及传入参数的限制

//@Pointcut("execution()")

public void LogAspect(){

// 此方法无方法体,目的是为了让同类中其他方法使用此切入点

}

//在Pointcut我们还可以限制对哪个bean进行增强以及传入参数的限制

//还可以搭配正则表达式使用，可定位到特定包下的某些方法,示例

//多个匹配之间我们可以使用链接符 &&(且)，||(或)，!(非)来表示

//下面这个切点的意思就是满足com.hecl.zhenghe.Controller.UserController.getUserInfo(String)

//且参数名为name

//且注入的bean为ALiangX

//该切点才会增强目标方法

@Pointcut("\* execution(com.hecl.zhenghe.Controller.UserController.getUserInfo(String) && args(name) && bean(ALiangX))")

public void LogAspect2(String name){

// 此方法无方法体,目的是为了让同类中其他方法使用此切入点

}

@Before("LogAspect()")

public void doBefore(JoinPoint joinPoint){

ServletRequestAttributes servletRequestAttributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();

HttpServletRequest request = servletRequestAttributes.getRequest();

log.info("请求方法地址-----------" + request.getRequestURI());

log.info("请求方法参数-----------" + Arrays.toString(joinPoint.getArgs()));

log.info("请求类方法名称-----------" + joinPoint.getSignature());

}

@After("LogAspect()")

public void doAfter(JoinPoint joinPoint){

log.info("该方法已执行完毕");

//...这下面还可以执行其它的行为

}

@AfterReturning(pointcut = "LogAspect()",returning = "object")

public void doAfterReturning(Object object){

log.info("返回参数内容-----------" + object);

}

//在使用时，我们传入了 ProceedingJoinPoint 类型的参数，

//这个对象是必须要有的，并且需要调用 ProceedingJoinPoint 的 proceed() 方法

//如果没有调用joinPoint.proceed()则会阻塞目标方法

@Around("LogAspect()")

public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable{

log.info("调用方法前先验证权限是否足够");

Object result = joinPoint.proceed();

log.info("方法结束时间-----------" + System.currentTimeMillis());

return result;

}

@AfterThrowing(pointcut = "LogAspect()",throwing = "e")

public void doAfterThrowing(Exception e){

System.out.println("错误行为-----------" + e);

}

}

总结： Spring实现的AOP是代理模式，给调用者使用的实际是已经过加工的对象，你编程时方法体里只写了A，但调用者拿到的对象的方法体却是xAy。x和y总还是需要你来写的，这就是增强。x和y具体在什么时候被调用总还是需要你来规定的，虽然是基于约定的声明这种简单的规定，这就是切点。