**动态代理：**

1. **JDK Proxy**
2. **CGLIB Enhancer**

**JDK Proxy :**

**java.lang.reflect.Proxy类动态的根据您指定的所有接口生成一个class字节码**

**该class会继承Proxy类，并实现所有你指定的接口（您在参数中传入的接口数组）；**

**然后再利用您指定的classloader将class字节码加载进系统，最后生成类的对象，**

**并初始化该对象的一些值，即所有的接口对应的Method成员。**

**初始化之后将该对象返回给调用的客户端。**

**这样客户端拿到的就是一个实现你所有的接口的Proxy对象。**

**也就是说：**

**父类java.lang.reflect.Proxy使用newProxyInstance()方法时有3个参数：**

**(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)**

**其中第二个参数：interfaces 子类$Prox将继承业务类的接口，并重写其中的方法**

**也就是说，实现了业务类继承的所有接口，但是不对业务类本身进行继承，因此在拦截上，不会去拦截业务类本身自带的(非实现接口而来的)方法，**

**甚至可以说，不光是不拦截，子类$proxy0中就没有去重写业务类本身的方法，在newProxyInstance后获取到的返回对象，想调用都没门。**

**其中第一个参数：loader 将子类字节码加载进系统**

**其中第三个参数：InvocationHandler 用于拦截子类重写的接口方法**

**之后将这个子类$Proxy0对象返回给客户端。**

**注意：**

1. **子类$Proxy0在重写接口方法时,同样重写了toString,equal,hashcode这三个**

**而这些方法包括整个子类$Proxy0都会设置成final,不可继承，不可重写**

**其中这个子类会使用static代码块，设置好这些方法的各个method引用。**

**即调用toString,equal,hashcode，也会被拦截。**

1. **重写方法的核心在于：super.invocationHandler.invoke(this,method,args)**

**也就是说：**

**在使用子类$Proxy0对象的接口方法时，会使用invocationHandler的invoke(Object o, Method method, Object[] objects)**

**而这个方法的三个参数，**

**其中o:代表子类$Proxy0对象，即代理类对象**

**Method：代表子类已经设置好的method引用**

**Objects: 代表调用子类方法时传入的参数**

**也就是调用方法，但是在携带了method,args转向了其他方法的调用，**

**这也就是代理，拦截了方法的调用。**

1. **在拦截下来之后应该如果放行这个方法呢？**

**需要一个target目标类对象，将这些被拦截下来的方法再利用反射重新调用**

**在新建一个实现了invocationHandler的类时，可以添加一个target用来设置业务类对象作为成员对象。**

**在invocationHandler.invoke(…)中需要使用：method.invoke(target,args)**

**这样就将方法放行了，要接收method.invoke的返回值，并将其返回出来**

**这里可能引起爆栈(死循环)：在invokeHandler.invoke中，再对$Proxy0对象的方法调用（如：toString,equal,hashcode.），因为这个方法再一次的进入了：**

**super.invocationHandler.invoke(this,method,args)，再其中又进入了此处，也就是无限递归了。**

1. **客户端在使用Proxy.newProxyInstance(…)后接收这个方法的返回值，作为代理对象，由他来执行真实对象(内部设置的target成员变量)的方法**
2. **想要看到生成的子类$Proxy0,可以在main的开头使用：**
   1. **System.getProperties().put("jdk.proxy.ProxyGenerator.saveGeneratedFiles", "true"); -- jdk 1.8可以**
   2. **System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true"); -- jdk11可以**

**proxy class :class com.sun.proxy.$Proxy0**

**proxy super class :class java.lang.reflect.Proxy**

**CGLIB Enhancer**

**使用字节码处理框架ASM，来转换字节码并动态生成业务类的子类，且重写了非final父类的所有非final方法(也就是这是一个继承关系，不可以有final，不可继承，不可重写的关键字，理所当然的是private也是不可以的)。在子类中拦截所有父类方法的调用。**

**创建Enhancer对象后，执行setSuperClass(业务类.class)和setCallback(…)后, 就可以使用create制作代理对象了并返回出来，客户端就能获取到这个代理对象。**

**注意：**

1. **与Proxy区别：业务类不必强调是非实现接口，有也可以，没有也可以，均可使用cglib代理,而proxy只能对实现了接口的业务类进行代理。**
2. **MethodInterceptor具有足够的通用性，可以满足任何拦截需求。在setCallBack中就可以使用实现了MethodInterceptor的类，需要重写其intercept()方法。**
3. **类似于Proxy，它也是一种代理方式，生成了新的代理类，再调用代理类的方法时，这些方法会携带相关参数回到Enhancer增强器的intercept回调函数，再由intercept中将这些方法放行。Intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy)**

**其中：**

**o : 操作了字节码的业务子类对象，即代理类对象**

**method : 子类调用的方法的引用**

**objects : 方法的传入参数**

**MethodProxy : 子类类对方法的代理引用。**

**方法的放行：**

**MethodProxy.invokeSuper(o,objects)**

**这里传递的是业务子类对象，即代理类对象**

**如果是使用Method.invoke(target,objects)**

**那么传递的就应该是目标对象，即真实对象。**

1. **想要看到生成的代理业务子类，可以在main的开头使用：**
   1. **System.setProperty(DebuggingClassWriter.DEBUG\_LOCATION\_PROPERTY, "target/cglib");**

**object class: class cn.cxd.beans.StudentSeviceImpl$$EnhancerByCGLIB$$c820b0d2**

**super class :class cn.cxd.beans.StudentSeviceImpl**

**AOP切面的xml配置：**

1. **创建target真实业务类，创建advice类/aspect类**

**创建advice类：(半自动)**

**可分为多个advice类，但是要成为advice类需要实现一些接口中的至少一个：**

**在半自动配置中将在interceptorNames 的<list>中添加value用于拦截**

MethodInterceptor

[invoke(MethodInvocation methodInvocation)]  ,

AfterReturningAdvice

[afterReturning(Object o, Method method, Object[] objects, Object o1)],

ThrowsAdvice [afterThrowing(Exception ex)],

MethodBeforeAdvice [before(Method method, Object[] objects, Object o)]

1. **半自动：依赖于生成ProxyFactoryBean Bean对象来实现代理**

**jar: aopalliance-1.0.0**

**xml中不必添加新的xsd约束等，仅仅是使用bean标签生成初始化成员对象的ProxyFactoryBean 对象，并借此完成代理，这种代理下，拦截的方法对于使用的方式有差异：**

1. **如果没有设置Optimize为“true”,那么使用的是jdkProxy，原理是实现接口，那么只能拦截下真实业务类中重写了的接口下的方法，而且想直接转成业务类也是不可行的，故而想执行业务类中自己的方法是不成功的，更无拦截可言。**
2. **如果设置了cglib，那么业务类中自己的方法可以被拦截。**

**在设置时有几个重要的property要设置：**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **类** | **类型** | **Bean中配置** |
| **Target**  **真实业务对象** | **AdvisedSupport** | **Object** | **<property name="target" ref="beanID"></property>** |
| **Interfaces**  **业务类实现的接口，list<class>设置时，以value设置class** | **AdvisedSupport** | **List<Class<?>>** | **<property name="interfaces">**  **<list> <value>cn.cxd.springAopXmlConfig.**  **service.IUserService</value>**  **</list>**  **</property>** |
| **interceptorNames**  **用于拦截的类的名称，String[]设置时，以value设置beanID,注意是字符串value，而不是引用ref** | **ProxyFactoryBean** | **private String[] interceptorNames;** | **<property name="interceptorNames">**  **<list> <value>myAdvice</value> <value>**  **myMethodBeforeAdvice**  **</value>**  **</list>**  **</property>** |
| **Optimize**  **默认为false，使用jdkProxy，如果设置为true，则代表使用cglib** | **ProxyConfig** | **private boolean optimize = false;** | **<property name="optimize" value="true"></property>** |

**完整的xml配置：**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="
5. http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans.xsd
7. ">
8. <bean id="userService" **class**="cn.cxd.springAopXmlConfig.service.UserService"></bean>
9. <bean id="myAdvice" **class**="cn.cxd.springAopXmlConfig.aspect.MyAdvice"></bean>
10. <bean id="myMethodBeforeAdvice" **class**="cn.cxd.springAopXmlConfig.aspect.MyMethodBeforeAdvice"></bean>
11. <bean id="myMethodAfterReturning" **class**="cn.cxd.springAopXmlConfig.aspect.MyMethodAfterReturning"></bean>
12. <bean id="proxyOrCglib" **class**="org.springframework.aop.framework.ProxyFactoryBean">
13. <property name="target" ref="userService"></property>
14. <property name="interfaces">
15. <list>
16. <value>cn.cxd.springAopXmlConfig.service.IUserService</value>
17. </list>
18. </property>
19. <property name="interceptorNames">
20. <list>
21. <value>myMethodBeforeAdvice</value>
22. <value>myAdvice</value>
23. <value>myMethodAfterReturning</value>
24. </list>
25. </property>
27. <property name="optimize" value="false"></property>
29. </bean>
30. </beans>

顺序：与<list>中的interceptorNames的顺序有关联

* 1. 若实现了MethodInterceptor接口的位置较先于MethodBeforeAdvice，则invoke中proceed()放行前的语句也会较先被执行，随后再是before方法。
  2. 若实现了 AfterReturningAdvice 接口的位置较先于 MethodInterceptor，则 invoke 中proceed()放行后的语句会较先被执行，随后再是AfterReturningMethod方法

可以是以下顺序：

1. Invoke中proceed()放行前的语句，
2. before，o代表target对象,method代表方法引用,objects代表参数.
3. invoke中放行的业务真实方法，
4. afterReturning, o代表method被放行得到的返回值,o1代表target对象，method,objects
5. Invoke中proceed()放行后的语句

**客户端：获取的是在xml中初始化好的ProxyFactoryBean对象。**

1. **package** test;
3. **import** cn.cxd.springAopXmlConfig.service.IUserService;
4. **import** org.springframework.context.ApplicationContext;
5. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **public** **class** SpringAopHalfTest {
8. **public** **static** **void** main(String[] args) {
9. ApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");
10. IUserService userProxy = (IUserService) context.getBean("proxyOrCglib");
11. userProxy.addUser();
12. System.out.println(userProxy.getClass().getSuperclass());
13. }
14. }

* 1. **全自动：**

**jars: aspectjweaver-1.8.10.jar aopalliance-1.0.0**

**spring-aspects-4.3.0.RELEASE.jar**

**xml中要添加新的xsd约束等，使用<aop:config> 完成配置**

**可以使用自定义aspect类<aop:aspect>，**

**也可以使用继承了一些接口的advice类<aop:advice>**

**（aspect不需继承什么类，也不需实现什么接口，但是aspect类的方法或许需要一些参数）**

**Xml的配置：**

1. <?xml version="1.0" encoding="UTF-8"?>
3. <!-- 1. 配置注解需要xmlns + xsi:schemaLocation -->
4. <beans xmlns="http://www.springframework.org/schema/beans"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xmlns:aop="http://www.springframework.org/schema/aop"
7. xsi:schemaLocation="
8. http://www.springframework.org/schema/beans
9. http://www.springframework.org/schema/beans/spring-beans.xsd
10. http://www.springframework.org/schema/aop
11. http://www.springframework.org/schema/aop/spring-aop.xsd"
12. >


16. <bean id="proxyOrCglib" **class**="cn.cxd.springAopXmlConfig.service.UserService"></bean>
17. <bean id="myAdvice" **class**="cn.cxd.springAopXmlConfig.aspect.MyAdvice"></bean>
18. <bean id="myMethodBeforeAdvice" **class**="cn.cxd.springAopXmlConfig.aspect.MyMethodBeforeAdvice"></bean>
19. <bean id="myAspect" **class**="cn.cxd.springAopXmlConfig.aspect.MyAspect"></bean>
20. <bean id="myBookService" **class**="cn.cxd.springAopXmlConfig.service.BookService"></bean>
21. <bean id="myMethodAfterReturning" **class**="cn.cxd.springAopXmlConfig.aspect.MyMethodAfterReturning"></bean>
22. <aop:config proxy-target-**class**="true">
23. <aop:pointcut id="myPointCut" expression="execution(\* cn.cxd.springAopXmlConfig..\*.\*add\*(..))"/>
24. <aop:advisor advice-ref="myAdvice" pointcut-ref="myPointCut"></aop:advisor>
25. <aop:advisor advice-ref="myMethodBeforeAdvice" pointcut-ref="myPointCut"></aop:advisor>
26. <aop:advisor advice-ref="myMethodAfterReturning" pointcut-ref="myPointCut"></aop:advisor>
27. <aop:aspect ref="myAspect">
28. <aop:before method="before" pointcut-ref="myPointCut"></aop:before>
29. <aop:around method="around"  pointcut-ref="myPointCut"/>
30. <aop:after-returning method="afterReturning" pointcut-ref="myPointCut" returning="reObj"></aop:after-returning>
31. <aop:after method="after" pointcut-ref="myPointCut"></aop:after>
32. <aop:after-throwing method="afterThrowing" pointcut-ref="myPointCut" throwing="e"></aop:after-throwing>
33. </aop:aspect>
34. </aop:config>
35. </beans>

**推荐配置：**

**先配置<aop:advisor> ,按序声明：**

**MethodBeforeAdvice**

**MethodInterceptor**

**AfterReturningAdvice**

**ThrowsAdvice**

**再配置<aop:aspect>,按序声明：**

**<aop:before>**

**<aop:around>**

**<aop: after-returning>**

**<aop:after>**

**<aop:after-throwing>**

**这种配置下的执行顺序是：**

**1.接口methodInterceptor invoke 在放行前**

**2.接口MethodBeforeAdvice before**

**3.aspect <aop:before>**

**4.aspect <aop:around> 在放行前**

**5.方法的放行**

**6.aspect <aop:around> 在放行后**

**7.aspect <aop:after-returning>**

**8.aspect <aop:after>**

**9.接口AfterReturningMethod after**

**10.接口methodInterceptor invoke 在放行后**

1. **<aop:config proxy-target-class="true">：**

**proxy-target-class="true"：使用cglib代理方式**

1. **<aop:pointcut id="myPointCut" expression="execution(\* cn.cxd.springAopXmlConfig..\*.\*add\*(..))"/>**

**表达式expression = “execution(xxxx)”:**

**表达式结构：(未试过) 权限 返回值 包 类 方法(..)**

**但是\*(..)代表方法应保留**

**aaa\* 代表aaa开头,一般是用于方法/类/包上,也可 \*aa\* 代表含有aa的类/方法/包**

**bbb.. 代表bbb本包及子包上**

**bb.\*.\*(..)代表bb包下的任意类的任意方法（不含bb包）**

**bb.\*.\*.\*(..)代表bb包下的任意包的任意类的任意方法(不含bb包)**

**例子：execution(public \* \*(..))**

**execution(\* add\*(..)) : add开始的方法(\*空格add\*(..))**

1. **配置<aop:aspect>的重点：  
   1. around方法,需要ProceedingJoinPoint参数，用于放行**

**2. after方法，需要JoinPoint，Object参数，Object用于接收方法调用的返回值。**

**在xml中：returning 即为Object参数的变量名称**

**<aop:after-returning method="afterReturning" pointcut-ref="myPointCut" returning="reObj"></aop:after-returning>**

**3. afterThrowing方法，需要JoinPoint,Exception参数**

**在xml中：throwing 即为Exception参数的变量名称**

**<aop:after-throwing method="afterThrowing" pointcut-ref="myPointCut" throwing="e"></aop:after-throwing>**

**4.** **after不管有没异常抛出都会被执行，但是如果由于异常的抛出，导致方法没有执行到返回语句，那么afterReturning就不会执行，如果方法是void，正常执行时afterReturning的Object接收到的值为null.**

1. **package** cn.cxd.springAopXmlConfig.aspect;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.ProceedingJoinPoint;
5. **import** org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint;
7. **public** **class** MyAspect {
8. **public** **void** before(JoinPoint joinPoint){
9. System.out.println("方法前使用StuSeviceAspect的before方法 : "+ " args: "+joinPoint.getArgs()+" kind: "+joinPoint.getKind()+" target: "+joinPoint.getTarget()+" signature: "+joinPoint.getSignature()+" staticPart: "+joinPoint.getStaticPart()+" this: "+joinPoint.getThis()+" sourceLocation: "+joinPoint.getSourceLocation());
11. }
12. **public** **void** after(){
13. System.out.println("方法最终StuSeviceAspect的after方法 : ");
14. }
15. **public** **void** afterReturning(JoinPoint joinPoint,Object reObj) {
16. System.out.println("方法后使用StuSeviceAspect的afterReturning方法 返回值为： "+reObj);
17. }
18. **public** Object around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {
19. System.out.println("方法环绕前使用StuSeviceAspect的around方法 ");
20. Object reObj = proceedingJoinPoint.proceed();
21. System.out.println("方法环绕后使用StuSeviceAspect的around方法 ");
22. **return** reObj;
23. }
24. **public** **void** afterThrowing(JoinPoint joinPoint,Exception e){
25. System.out.println("方法异常后出现afterThrowing joinPoint: "+joinPoint.getSignature().getName()+" ex: "+e);
26. }
27. }

**注解配置：**

**Xml配置：**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:aop="http://www.springframework.org/schema/aop"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/aop
10. http://www.springframework.org/schema/aop/spring-aop.xsd
11. http://www.springframework.org/schema/context
12. http://www.springframework.org/schema/context/spring-context.xsd">
14. <context:component-scan base-**package**="cn.cxd"></context:component-scan>
15. <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
17. </beans>

**切面类定义：**

1. **package** cn.cxd.aspects;
3. **import** org.aspectj.lang.JoinPoint;
4. **import** org.aspectj.lang.ProceedingJoinPoint;
5. **import** org.aspectj.lang.annotation.\*;
6. **import** org.springframework.stereotype.Component;
8. @Component
9. @Aspect
10. **public** **class** UserServiceAspect {


14. @Around(value = "pointCut()")
15. **public** Object around(ProceedingJoinPoint joinPoint) **throws** Throwable{
16. System.out.println("around 放行前");
17. Object reObj = joinPoint.proceed();
18. System.out.println("around 放行后");
19. **return** reObj;
20. }
22. @Pointcut("execution(\* cn.cxd.services.\*.add\*(..))")
23. **public** **void** pointCut(){};
25. @Before(value = "pointCut()")
26. **public** **void** before(JoinPoint joinPoint){
27. System.out.println("before 方法: "+joinPoint.getSignature().getName());
28. }
30. @AfterReturning(value = "pointCut()",returning = "reObj")
31. **public** **void** afterReturning(JoinPoint joinPoint,Object reObj){
32. System.out.println("afterReturning 返回值： "+reObj);
33. }
35. @After(value = "pointCut()")
36. **public** **void** after(){
37. System.out.println("after 最终方法");
38. }
40. @AfterThrowing(value = "pointCut()",throwing = "e")
41. **public** **void** afterThrowing(JoinPoint joinPoint,Exception e){
42. System.out.println("afterThrowing 方法："+e.getCause());
43. }
45. }

**顺序：**

**before - around - afterReturning - after - afterThrowing 注解方法的定义顺序**

**虽然执行顺序是固定的，但是还是推荐xml**

**around 放行前**

**before 方法: addUser**

**UserServiceImpl addUser()**

**around 放行后**

**after 最终方法**

**afterReturning 返回值： 1235**