# 3.Spring AOP入门

## 3.1.AOP:面向切面编程

在程序运行时，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。

核心思想：代理。

## 3.2.代理的入门案例

角色：客户，中介，房东。（可以画图举例）

代理模式特点：

1. 代理者要实现与被代理者相同的接口。
2. 代理必须完成自己的本质工作。
3. 在完成本质工作的同时，要完成额外的工作。

## 3.3.静态代理

以事务控制为例（直接复制day02-3，添加一个tx包，TransactionManager类）

|  |
| --- |
| @Component  public class TransactionManager {  public void begin(){  System.out.println("事务开启");  }  public void commit(){  System.out.println("事务提交");  }  public void rollBack(){  System.out.println("事务回滚");  }  } |

Service层加入事务代码

|  |
| --- |
| @Service  public class UserServiceImpl implements UserService {  @Autowired  private UserDao userDao;  @Autowired  private TransactionManager tx;  @Override  public void addUser() {  try {  tx.begin();  userDao.addUser();  tx.commit();  } catch (Exception e) {  tx.rollBack();  }  } |

传统的事务控制：

1. 事务控制的代码和业务代码紧紧的耦合在一起
2. 代码大量重复，不能复用。

解决办法：1.将事务层代码和业务层代码分离，各干各的事

2.代码尽可能不重复。

静态代理模式：

添加proxy包并加入包扫描，添加StaticProxy类，实现事务代码和业务代码分离。

|  |
| --- |
| package proxy;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Component;  import service.UserService;  import tx.TransactionManager;  @Component("userService")  public class StaticProxy implements UserService{  @Autowired  private UserService target;  @Autowired  private TransactionManager tx;  @Override  public void addUser() {  try {  tx.begin();  target.addUser();  tx.commit();  } catch (Exception e) {  tx.rollBack();  }  }  } |

修改UserServiceImpl，单纯做业务

|  |
| --- |
| package service;  @Service("target")  public class UserServiceImpl implements UserService {  @Autowired  private UserDao userDao;    @Override  public void addUser() {  userDao.addUser();  }  } |

静态代理模式优点：可以实现业务层和事务层的分离

缺点：大量代码的重复。

一个静态代理只能代理一个类，做法不通用。

说明： 静态代理只是一种思想，实际开发中不使用。

## 3.4.动态代理。

JDK动态代理：要与被代理者实现相同的接口。

CGlib动态代理：对指定类生成一个子类，覆盖其中的所有方法。

4.1.JDK动态代理：

动态代理使用了反射，一个代理可以为任意类提供代理。

步骤：首先，创建代理类

|  |
| --- |
| //创建代理对象  public static Object getProxy(final Object target,final TransactionManager tx){  /\*\*  \* loader:真实对象的类加载器。  \* interfaces：真实对象的接口。  \*/  Object proxy = Proxy.newProxyInstance(target.getClass().getClassLoader(),  target.getClass().getInterfaces(), new InvocationHandler() {  /\*  \* InvocationHandler:动态代理必须实现这个接口，这个接口只定义了一个invoke方法。  \* invoke：其实就是用来反射我们真正要执行的方法。  \*/  @Override  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {  tx.begin();  Object object = method.invoke(target, args);  tx.commit();  return object;  }  });  return proxy;  } |

测试

|  |
| --- |
| //测试JDK动态代理  @Test  public void test01(){  ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  UserService userService = (UserService) context.getBean("target");  TransactionManager tx = (TransactionManager) context.getBean("transactionManager");  UserService proxy = (UserService) DynamicProxy.getProxy(userService, tx);  System.out.println(proxy.getClass());  proxy.addUser();  } |

JDK动态代理的特点：可以继承静态代理的全部特点，并且可以实现代码的复用。

可以处理一类事务

缺点：动态代理的灵活性不强，如果需要指定方法，则比较麻烦。

被代理者必须实现接口，否则不能代理。

4.2.CGlib动态代理

步骤：创建代理类

|  |
| --- |
| package proxy;  public class DynamicProxy{  //创建代理对象  public static Object getProxy(final Object target,final TransactionManager tx){  //1.创建增强器，用来动态创建目标对象的子类，拦截所有的方法，执行方法  Enhancer enhancer = new Enhancer();  //2.设置父类  enhancer.setSuperclass(target.getClass());  //3.完成目标方法:设置回调。  enhancer.setCallback(new MethodInterceptor() {  @Override  public Object intercept(Object proxy, Method method, Object[] args,  MethodProxy methodProxy) throws Throwable {  tx.begin();  Object result = method.invoke(target, args);  tx.commit();  return result;  }  });  //获取代理对象  return enhancer.create();  }  } |

测试，与JDK测试一样。

特点：不管有无接口，都能为目标对象生成代理

生成的代理对象是目标对象的子类。

## 3.5.Spring中的AOP

名词解释：

1. 切面（Aspect）：一个关注点的模块化，这个关注点可能会横切多个对象。

类

1. 连接点（Joinpoint）：在程序执行过程中某个特定的点。

目标方法

1. 通知（Advice）：在切面的某个特定的连接点上执行的动作。

执行目标方法之外的动作，写在切面中

1. 切入点（Pointcut）：匹配连接点的断言。

执行目标方法的匹配规则

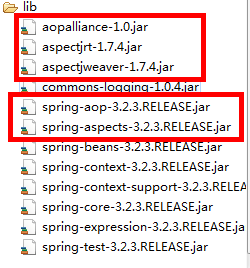
1. 目标对象（Target Object）： 被一个或者多个切面所通知的对象。

被代理对象

## 3.6.AOP入门案例

先创建一个传统项目（只有service层即可）

步骤：6.1.导包



6.2.创建一个切面类

|  |
| --- |
| package aspect;  import org.springframework.stereotype.Component;  @Component  public class TxAspect {    } |

6.3.定义一个通知

|  |
| --- |
| public void before(){  System.out.println("我是一个通知");  } |

6.4.配置切入点

导入aop头文件(自己导入约束，加入模板)

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.2.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-3.2.xsd "> |

配置切入点和切面

|  |
| --- |
| <!-- 配置切面 -->  <aop:config>  <!-- expression:切入点表达式，匹配规则  id：唯一标识-->  <aop:pointcut expression="within(service.UserServiceImpl)" id="pc"/>  <aop:aspect ref="txAspect">  <aop:before method="before" pointcut-ref="pc"/>  </aop:aspect>  </aop:config> |

测试，发现使用的是JDK动态代理。

|  |
| --- |
| @Test  public void test01(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  UserService userService = (UserService) context.getBean("userService");  System.out.println(userService.getClass());  userService.addUser();//连接点  } |

满足切入点表达式则执行通知，如果在service包下创建一个ProductService，是否也执行通知呢？

可以测试一下。发现并没有执行。

Spring自己集成了动态代理。那如果没有接口，spring是不是也集成了CGlib呢？做个试验：

|  |
| --- |
| @Component("userService")  public class UserServiceImpl {  //@Override  public void addUser() {  System.out.println("添加一个用户");  }  } |

测试：

|  |
| --- |
| @Test  public void test01(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  UserServiceImpl userService = (UserServiceImpl) context.getBean("userService");  System.out.println(userService.getClass());  userService.addUser();//连接点  } |

测试发现，输出的class类型为CGlib的动态代理

6.5.说明：

6.5.1.代理对象生成策略：在spring中，如果目标对象有接口，则使用JDK动态代理，如果目标对象没有接口，则使用cglib动态代理。

6.5.2.当从容器中获取对象的时候，如果获取的对象满足切入点表达式，那么就会为其创建代理对象。代理对象执行指定的方法，就会执行与之绑定的通知。

## 3.7.切入点表达式

7.1.within()表达式：根据类进行匹配，控制粒度较粗。

语法：

within(service.UserServiceImpl)：匹配UserServiceImpl这个类

within(service.\*)：匹配service包下第一层类。（如果再写一个包，发现不能匹配）

within(service.\*.\*)：匹配service包下的第二层包（但是不能匹配其他层）

within(service..\*)：匹配service包下所有层的所有类。

7.2.execution()表达式：根据方法名、方法参数类型、返回值等进行匹配，控制粒度细。

语法：execution(返回值类型 包名.类名.方法名(参数列表))

例子1：

<aop:pointcut expression=

*"execution(int service.UserServiceImpl.add())"* id=*"txPointcut"*/>

改切点表达式表示

返回值为int 包名类名serviceUserServiceImpl 方法为add()的匹配规则

例子2：

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

规则：返回值值任意， 包名service下子类的add(),只能包含一层，子孙类不行。

例子3：

<aop:pointcut

expression=*"execution(\* service..\*.add())"* id=*"txPointcut"*/>

规则：方法返回值任意， service包下的所有子孙类的add()

例子4：

<aop:pointcut expression=

*"execution(\* service..\*.add(int,String))"* id=*"txPointcut"*/>

规则：返回值的类型任意 service子孙包下的add方法参数类型为int,String

例子5:

<aop:pointcut expression=

*"execution(\* service..\*.add(..))"*

id=*"txPointcut"*/>

规则：返回值类型任意 service下的所有子孙类.add方法() (参数任意)

说明：一般除了有特殊约定，很少会定义返回值类型。

例子:在UserService中再加入一个updateUser方法，用execution（）表达式只匹配addUser而不匹配updateUser

|  |
| --- |
| <aop:pointcut expression="execution(\* service.UserServiceImpl.addUser())"  id="pc"/> |

如果想匹配service下的所有类的所有方法，应该如何配置。

|  |
| --- |
| <aop:pointcut expression="execution(\* service..\*.\*(..))" id="pc"/> |

当然，也可以简写，\* service..\*(..)，不过一般也不这么写。还是按正规的写。

补充：强制使用cglib来创建代理对象：

|  |
| --- |
| <!-- 配置切面 -->  <aop:config proxy-target-class="true"> |

## 3.8.spring中的五大通知。

前置通知（Before advice）：目标方法执行之前执行。

后置通知（After returning advice）：目标方法执行之后执行。

异常通知（After throwing advice）：目标方法抛出异常后执行。

最终通知（After (finally) advice）：目标方法执行之后都会执行。

前四大通知不能控制目标方法是否执行。

环绕通知（Around Advice）：目标方法执行之前、执行之后都会执行。

只有环绕通知可以控制目标方法执行，功能最强大。

之前我们的例子就是一个前置通知，接下来，在切面中配通知

|  |
| --- |
| @Component  public class TxAspect {  //前置通知  public void before(){  System.out.println("我是一个前置通知");  }  //后置通知  public void afterReturn(){  System.out.println("我是一个后置通知");  }  //异常通知  public void afterThrow(){  System.out.println("我是一个异常通知");  }  //最终通知，用的很少  public void after(){  System.out.println("我是一个最终通知");  }  //环绕通知,要求必须添加ProceedingJoinPoint  public void around(ProceedingJoinPoint joinPoint) throws Throwable{  System.out.println("环绕通知开始");  //目标方法执行  joinPoint.proceed();    System.out.println("环绕通知结束");  }  } |

配置文件中配置通知

|  |
| --- |
| <!-- 配置切面 -->  <aop:config proxy-target-class="true">  <!-- expression:切入点表达式，匹配规则  id：唯一标识-->  <!-- <aop:pointcut expression="within(service..\*)" id="pc"/> -->  <aop:pointcut expression="execution(\* service..\*(..))" id="pc"/>  <aop:aspect ref="txAspect">  <!-- <aop:before method="before" pointcut-ref="pc"/>  <aop:after-returning method="afterReturn" pointcut-ref="pc"/>  <aop:after-throwing method="afterThrow" pointcut-ref="pc"/>  <aop:after method="after" pointcut-ref="pc"/> -->  <aop:around method="around" pointcut-ref="pc"/>  </aop:aspect>  </aop:config> |

## 3.9.通知的具体用法

9.1.如果要控制目标方法的执行（权限、事务）

首选环绕通知

9.2.日志的收集

前置通知和后置通知

9.3.收集异常信息

异常通知

9.4.如果想获取目标对象，或者目标方法，如何获取。

|  |
| --- |
| //前置通知,要求，其他通知中禁止添加ProceedingJoinPoint  public void before(JoinPoint joinPoint){  System.out.println("我是一个前置通知");  //获取目标对象  Class targetClass = joinPoint.getTarget().getClass();  //获取目标对象的类型  String methodName = joinPoint.getSignature().getName();  System.out.println("目标对象："+targetClass);  System.out.println("目标方法："+methodName);  } |

ProceedingJoinPoint只有一个方法，就是控制目标对象执行，如果把这里的JoinPoint换做ProceedingJoinPoint，则会报错。

ProceedingJoinPoint只能用在环绕通知中。

9.5.如何获取异常信息

|  |
| --- |
| //异常通知,通常需要加入JoinPoint，获取目标对象和目标方法  public void afterThrow(JoinPoint joinPoint,Throwable throwable){  System.out.println("我是一个异常通知");  System.out.println("目标方法："+joinPoint.getSignature().getName());  System.out.println("异常信息："+throwable.getMessage());  System.out.println("异常类型："+throwable.getClass());  } |

配置文件

|  |
| --- |
| <aop:after-throwing method="afterThrow" pointcut-ref="pc"  throwing="throwable"/> |

注意：配置文件中的变量名必须与切面中的参数名称相同，否则报错

JoinPoint参数必须放在第一位，否则报错



一般看到如上报错信息，都是参数问题导致。