

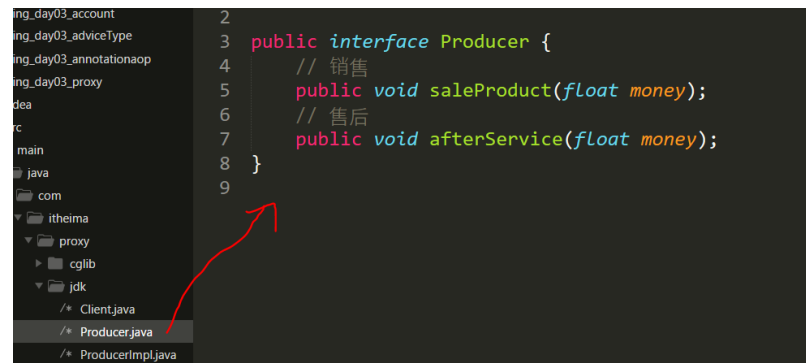
## Spring03

### 代理: jdk 和 cglib

Jdk 代理（对接口代理，spring 的 aop 默认代理）

在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

如果对接口，默认使用 JDK 代理。



```
2
3 public interface Producer {
4     // 销售
5     public void saleProduct(float money);
6     // 售后
7     public void afterService(float money);
8 }
9
```

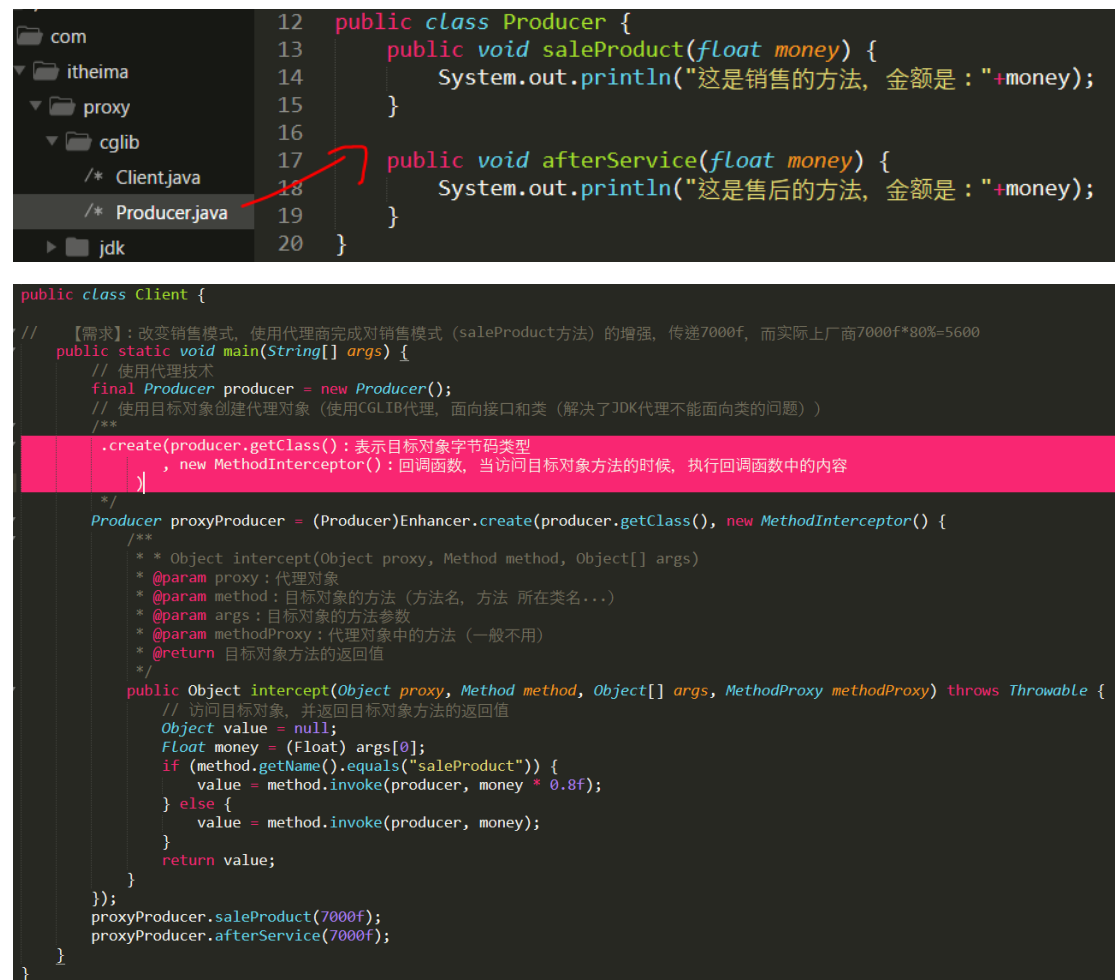
```
public class ProducerImpl implements Producer {
    public void saleProduct(float money) {
        System.out.println("这是销售的方法, 金额是: "+money);
    }

    public void afterService(float money) {
        System.out.println("这是售后的方法, 金额是: "+money);
    }
}
```

```
public class Client {

    // 【需求】：改变销售模式，使用代理商完成对销售模式（saleProduct方法）的增强，传递7000f，而实际上厂商7000f*80%=5600
    public static void main(String[] args) {
        // 使用代理技术
        final Producer producer = new ProducerImpl();
        // 使用目标对象创建代理对象（使用JDK代理，面向接口）
        /**
         * newProxyInstance(ClassLoader loader, 类加载器，代理对象和目标对象要使用同一个类加载器
         * Class<?>[] interfaces,代理对象和目标对象要使用同一个接口
         * InvocationHandler, 回调函数，当访问目标对象方法的时候，执行回调函数中的内容
         */
        Producer proxyProducer = (Producer) Proxy.newProxyInstance(producer.getClass().getClassLoader(),
            producer.getClass().getInterfaces(),
            new InvocationHandler() {
                /**
                 * Object invoke(Object proxy, Method method, Object[] args)
                 * @param proxy : 代理对象
                 * @param method : 目标对象的方法（方法名，方法 所在类名...）
                 * @param args : 目标对象的方法参数
                 * @return 目标对象方法的返回值
                 */
                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                    // 访问目标对象，并返回目标对象方法的返回值
                    Object value = null;
                    Float money = (Float)args[0];
                    if(method.getName().equals("saleProduct")){
                        value = method.invoke(producer, money*0.8f);
                    }
                    else{
                        value = method.invoke(producer, money);
                    }
                    return value;
                }
            });
        proxyProducer.saleProduct(7000f);
        proxyProducer.afterService(7000f);
    }
}
```

## Cglib 代理（对类和接口的代理）



### AOP 相关术语

**Joinpoint(连接点):** (方法) (重点) (重点)

所谓连接点是指那些被拦截到的点。在 spring 中, 这些点指的是方法, 因为 spring 只支持方法类型的连接点。

**Pointcut(切入点):** (方法) (重点)

所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义。

**Advice(通知/增强):** (方法) (重点)

所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。

通知的类型: 前置通知, 后置通知, 异常通知, 最终通知, 环绕通知。

**\*Target(目标对象):** \*

代理的目标对象。

**Weaving(织入):** (了解)

是指把增强应用到目标对象来创建新的代理对象的过程。

spring 采用动态代理织入, 而 AspectJ 采用编译期织入和类装载期织入。

**\*Proxy (代理):** \*

一个类被 AOP 织入增强后, 就产生一个结果代理类。

**Aspect(切面):** (类) (重点)

是切入点和通知 (引介) 的结合。

**Introduction(引介):** (不了解)

引介是一种特殊的通知在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方法或 Field。

## 切入点表达式的写法（重点）

### execution(表达式)

标准的表达式写法：

```
public void com.itheima.service.impl.AccountServiceImpl.saveAccount()
```

访问修饰符可以省略

```
void com.itheima.service.impl.AccountServiceImpl.saveAccount()
```

返回值可以使用通配符（\*：表示任意），表示任意返回值

```
* com.itheima.service.impl.AccountServiceImpl.saveAccount()
```

包名可以使用通配符，表示任意包。但是有几级包，就需要写几个\*。

```
* *.*.*.*.AccountServiceImpl.saveAccount()
```

包名可以使用..表示当前包及其子包

```
* *..AccountServiceImpl.saveAccount()
```

类名和方法名都可以使用\*来实现通配（一般情况下，不会这样配置）

```
* *..*.*() == * *()
```

参数列表：

可以直接写数据类型：

基本类型直接写名称                      int

引用类型写包名.类名的方式              java.lang.String

可以使用通配符表示任意类型，但是必须有参数

可以使用..表示有无参数均可，有参数可以是任意类型

全通配写法：\* \*.\*.\*.\*(..)

## Aop 入门案例

```
public interface AccountService {
    /**
     * 模拟保存账户（无返回值，无参数）
     */
    void saveAccount();
    /**
     * 模拟更新账户（无返回值，有参数）
     * @param i
     */
    void updateAccount(int i);
    /**
     * 删除账户（有返回值，无参数）
     * @return
     */
    int deleteAccount();
}

public class AccountServiceImpl implements AccountService {
    public void saveAccount() {
        System.out.println("执行了保存");
    }
    public void updateAccount(int i) {
        System.out.println("执行了更新"+i);
    }
    public int deleteAccount() {
        System.out.println("执行了删除");
        return 0;
    }
}
```

```
public class Logger {
    /**
     * 用于打印日志：计划让其在切入点方法执行之前执行（切入点方法就是业务层方法）
     */
    public void printLog(){
        System.out.println("Logger类中的pringLog方法开始记录日志了。。。");
    }
}
```

## 配置 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">
    <!-- 配置spring的Ioc,把service对象配置进来-->
    <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
    </bean>

    <!--spring中基于XML的AOP配置步骤
    1、把通知Bean也交给spring来管理
    2、使用aop:config标签表明开始AOP的配置
    3、使用aop:aspect标签表明配置切面
        id属性: 是给切面提供一个唯一标识
        ref属性: 是指定通知类bean的Id。
    4、在aop:aspect标签的内部使用对应标签来配置通知的类型
    我们现在示例是让printLog方法在切入点方法执行之前执行: 所以是前置通知
    aop:before: 表示配置前置通知
        method属性: 用于指定Logger类中哪个方法是前置通知
        pointcut属性: 用于指定切入点表达式, 该表达式的含义指的是对业务层中哪些方法
    增强
    -->
    <!-- 配置Logger类, 声明切面 (创建对象, 不是真正aop的切面) -->
    <bean id="logger" class="com.itheima.utils.Logger"></bean>

    <!--配置AOP-->
    <aop:config>
        <!--配置切面 -->
        <aop:aspect id="logAdvice" ref="logger">
            <!-- 配置通知的类型, 并且建立通知方法和切入点方法的关联-->
            <aop:before method="printLog" pointcut="execution(void
com.itheima.service.impl.AccountServiceImpl.saveAccount())"></aop:before>
            <aop:before method="printLog" pointcut="execution(void
com.itheima.service.impl.AccountServiceImpl.updateAccount(int))"></aop:before>
            <aop:before method="printLog" pointcut="execution(int
com.itheima.service.impl.AccountServiceImpl.deleteAccount())"></aop:before>
        </aop:aspect>
    </aop:config>
</beans>
```

## Spring AOP 的五种通知类型（使用XML）

```
public class Logger {  
    /**  
     * 前置通知  
     */  
    public void beforePrintLog(JoinPoint jp){  
        System.out.println("前置通知Logger类中的beforePrintLog方法开始记录日志了。。。");  
    }  
    /**  
     * 后置通知  
     */  
    public void afterReturningPrintLog(JoinPoint jp){  
        System.out.println("后置通知Logger类中的afterReturningPrintLog方法开始记录日志了。。。");  
    }  
    /**  
     * 异常通知  
     */  
    public void afterThrowingPrintLog(JoinPoint jp){  
        System.out.println("异常通知Logger类中的afterThrowingPrintLog方法开始记录日志了。。。");  
    }  
    /**  
     * 最终通知  
     */  
    public void afterPrintLog(JoinPoint jp){  
        System.out.println("最终通知Logger类中的afterPrintLog方法开始记录日志了。。。");  
    }  
    /**  
     * 环绕通知  
     * 问题：  
     *      当我们配置了环绕通知之后，切入点方法没有执行，而通知方法执行了。  
     * 分析：  
     *      通过对比动态代理中的环绕通知代码，发现动态代理的环绕通知有明确的切入点方法调用，而我们的代码中没有。  
     * 解决：  
     *      Spring框架为我们提供了一个接口：ProceedingJoinPoint。该接口有一个方法proceed()，此方法就相当于明确调用切入点方法。  
     *      该接口可以作为环绕通知的方法参数，在程序执行时，spring框架会为我们提供该接口的实现类供我们使用。  
     *  
     * spring中的环绕通知：  
     *      它是spring框架为我们提供的一种可以在代码中手动控制增强方法何时执行的方式。  
     */  
    public Object aroundPrintLog(ProceedingJoinPoint pjp){  
        Object rtValue = null;  
        try{  
            Object[] args = pjp.getArgs();//得到方法执行所需的参数  
            System.out.println("Logger类中的aroundPrintLog方法开始记录日志了。。。前置");  
            rtValue = pjp.proceed(args);//明确调用业务层方法（切入点方法）  
            System.out.println("Logger类中的aroundPrintLog方法开始记录日志了。。。后置");  
            return rtValue;  
        }catch (Throwable t){  
            System.out.println("Logger类中的aroundPrintLog方法开始记录日志了。。。异常");  
            throw new RuntimeException(t);  
        }finally {  
            System.out.println("Logger类中的aroundPrintLog方法开始记录日志了。。。最终");  
        }  
    }  
}
```

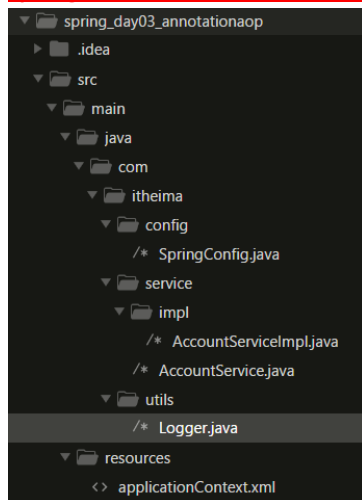
```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:aop="http://www.springframework.org/schema/aop"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans.xsd  
        http://www.springframework.org/schema/aop  
        http://www.springframework.org/schema/aop/spring-aop.xsd">  
    <!-- 配置spring的Ioc,把Service对象配置进来-->  
    <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"></bean>  
    <!-- 配置Logger类 -->  
    <bean id="logger" class="com.itheima.utils.Logger"></bean>  
    <!-- 配置AOP-->  
    <aop:config>  
        <!-- 配置切入点表达式 id属性用于指定表达式的唯一标识。expression属性用于指定表达式内容  
            此标签写在aop:aspect标签内部只能当前切面使用。  
            它还可以写在aop:aspect外面，此时就变成了所有切面可用  
            -->  
        <aop:pointcut id="pt1" expression="execution(* com.itheima.service..*(..))"></aop:pointcut>  
        <!-- 配置切面 -->  
        <aop:aspect id="logAdvice" ref="logger">  
            <!-- 配置前置通知：在切入点方法执行之前执行  
            <aop:before method="beforePrintLog" pointcut-ref="pt1"></aop:before-->  
            <!-- 配置后置通知：在切入点方法正常执行之后值。它和异常通知永远只能执行一个  
            <aop:after-returning method="afterReturningPrintLog" pointcut-ref="pt1"></aop:after-returning-->  
            <!-- 配置异常通知：在切入点方法执行产生异常之后执行。它和后置通知永远只能执行一个  
            <aop:after-throwing method="afterThrowingPrintLog" pointcut-ref="pt1"></aop:after-throwing-->  
            <!-- 配置最终通知：无论切入点方法是否正常运行它都会在其后面执行  
            <aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after-->  
            <!-- 配置环绕通知 详细的注释请看Logger类中-->  
            <aop:around method="aroundPrintLog" pointcut-ref="pt1"></aop:around>  
        </aop:aspect>  
    </aop:config>  
</beans>
```

```

public class AccountServiceImpl implements AccountService {
    public void saveAccount() {
        System.out.println("执行了保存");
    }
    public void updateAccount(int i) {
        System.out.println("执行了更新"+i);
    }
    public int deleteAccount() {
        System.out.println("执行了删除");
        return 0;
    }
}

```

### Spring AOP 的注解方式配置五种通知类型



```

@Configuration
@ComponentScan(value = "com.itheima")<context:component-scan base-package="com.itheima"></context:component-scan>
@EnableAspectJAutoProxy //<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
public class SpringConfig {
}

```

```

public interface AccountService {
    // 目标对象：无参数无返回值，有参数无返回值，无参数有返回值3个方法
    // 保存
    public void saveAccount();
    // 更新
    public void updateAccount(int i);
    // 删除
    public int deleteAccount();
}

```

```

@Service("accountService")
public class AccountServiceImpl implements AccountService {

    public void saveAccount() {
        System.out.println("【保存】...");
    }

    public void updateAccount(int i) {
        System.out.println("【更新】..., 传递的参数i="+i);
        int ii = 10/0;
    }

    public int deleteAccount() {
        System.out.println("【删除】..., 返回值="+10);
        return 10;
    }
}

```

```

// 切面
@Component("logger")
@Aspect
public class Logger {
    @Pointcut(value = "execution(* com.itheima.service..*(..))")
    public void myPointcut();
    // 前置通知
    //@Before(value = "myPointcut()")
    public void printLogBefore(JoinPoint jp){
        System.out.println("[前置通知] : 打印日志..." + jp.getSignature().getDeclaringType().getName() + "." + jp.getSignature().getName());
    }
    // 后置通知
    //@AfterReturning(value = "myPointcut()")
    public void printLogAfterReturning(JoinPoint jp){
        System.out.println("[后置通知] : 打印日志..." + jp.getSignature().getDeclaringType().getName() + "." + jp.getSignature().getName());
    }
    // 异常通知
    //@AfterThrowing(value = "myPointcut()")
    public void printLogAfterThrowing(JoinPoint jp){
        System.out.println("[异常通知] : 打印日志..." + jp.getSignature().getDeclaringType().getName() + "." + jp.getSignature().getName());
    }
    // 最终通知
    //@After(value = "myPointcut()")
    public void printLogAfter(JoinPoint jp){
        System.out.println("[最终通知] : 打印日志..." + jp.getSignature().getDeclaringType().getName() + "." + jp.getSignature().getName());
    }
}
// 环绕通知 (1: 参数必须使用ProceedingJoinPoint; 2: 必须具有Object返回值)
@Around(value = "myPointcut()")
public Object printLogAround(ProceedingJoinPoint jp){
    //System.out.println("[环绕通知] : 打印日志..." + jp.getSignature().getDeclaringType().getName() + "." + jp.getSignature().getName());
    Object value = null;
    try {
        this.printLogBefore(jp);
        value = jp.proceed(jp.getArgs()); // 相当于jdk代理中的 method.invoke(producer, money*0.8f);
        this.printLogAfterReturning(jp);
    } catch (Throwable throwable) {
        throwable.printStackTrace();
        this.printLogAfterThrowing(jp);
    } finally {
        this.printLogAfter(jp);
    }
    return value;
}
}

```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:p="http://www.springframework.org/schema/p"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xmlns:aop="http://www.springframework.org/schema/aop"
6     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7     xsi:schemaLocation="http://www.springframework.org/schema/beans
8         http://www.springframework.org/schema/beans/spring-beans.xsd
9         http://www.springframework.org/schema/context
10        http://www.springframework.org/schema/context/spring-context.xsd
11        http://www.springframework.org/schema/aop
12        http://www.springframework.org/schema/aop/spring-aop.xsd">
13
14     <!--1: 开启组件的注解支持-->
15     <context:component-scan base-package="com.itheima"></context:component-scan>
16     <!--2: 开启aop的注解支持-->
17     <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
18 </beans>

```