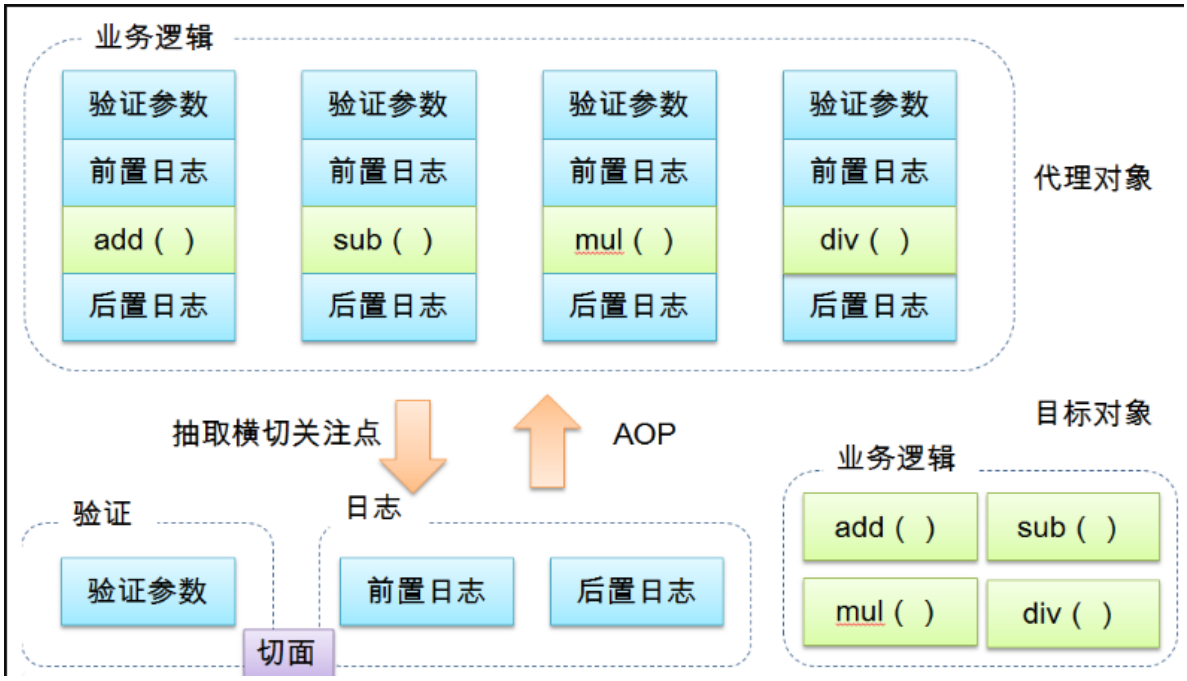
AOP

1 什么是AOP?

AOP（Aspect Oriented Programming），意为：**面向切面编程**，通过**预编译**方式和运行期间**动态代理**实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。



- 代理对象：经过AOP处理后，有了前置日志和后置日志（这些都是自定义的功能）
- 目标对象的业务逻辑：未经处理，只有执行方法
- 关注点：Spring的内容，后续讲解
- 这张图就是，在原先的业务逻辑上（目标对象），经过AOP切面处理，使用代理的方式，加上了想要的功能（日志）。这样子，不用修改源码，我们即可添加想要的功能。

2 AOP在Spring中的作用

①提供声明式事务；

②允许用户自定义切面

- 横切关注点：跨越应用程序多个模块的**方法或功能。**（比如日志Log）即是，与我们业务逻辑无关的，但是我们需要关注的部分，就是横切关注点。如日志 , 安全 , 缓存 , 事务等等 ....
- 切面（ASPECT）：横切关注点 被模块化 的特殊对象。即，它是一个**类。**（比如Log类）
- 通知（Advice）：切面必须要完成的工作。即，它是**类中的一个方法。**（比如Log中的方法）
- 目标（Target）：被通知对象。（接口）
- 代理（Proxy）：向目标对象应用通知之后创建的对象。（代理类）
- 切入点（PointCut）：切面通知 执行的 “地点”的定义。
- 连接点（JointPoint）：与切入点匹配的执行点。

- SpringAOP中，通过Advice定义横切逻辑，Spring中支持5种类型的Advice:

通知类型	连接点	实现接口
前置通知	方法方法前	org.springframework.aop.MethodBeforeAdvice
后置通知	方法后	org.springframework.aop.AfterReturningAdvice
环绕通知	方法前后	org.aopalliance.intercept.MethodInterceptor
异常抛出通知	方法抛出异常	org.springframework.aop.ThrowsAdvice
引介通知	类中增加新的方法属性	org.springframework.aop.IntroductionInterceptor

即 AOP 在 不改变原有代码的情况下，去增加新的功能 .

3 使用Spring实现AOP

案例：给原先的业务代码添加日志。

- 先导入一个包（添加依赖）

```xml
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
```

- 写一些业务接口

```java
package com.kuang.service;

public interface UserService {
    public void add();

    public void delete();

    public void update();

    public void query();


}
```

- 业务接口的实现类

```java
package com.kuang.serviceImpl;

import com.kuang.service.UserService;

public class UserServiceImpl implements UserService {
    public void add() {
        System.out.println("增加了一个用户");
    }

    public void delete() {
        System.out.println("删除一个用户");
    }

    public void update() {
        System.out.println("更新用户信息");
    }

    public void query() {
        System.out.println("查询了用户");
    }
}
```

- 接下来，本该是写动态代理的代理类，但是Spring的AOP不需要。有三种实现方法：
    - Spring的API接口
    - 自定义完成AOP
    - 注解实现

3.1 使用Spring的API接口

- 要添加日志功能的类

这里只列出前置通知和后置通知。

前置通知：

```
/**
 * 前置增强
 */
public class BeforeLog implements MethodBeforeAdvice {
    /**
     * @param method 要执行的目标对象的方法
     * @param args   参数
     * @param target 目标对象
     * @throws Throwable
     */
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println(target.getClass().getName() + "的" + method.getName() + "方法被执行了");
    }
}
```

后置通知：

```
/**
 * 后置增强
 */
public class AfterLog implements AfterReturningAdvice {
    /**
     * @param returnValue 返回值
     * @param method
     * @param args
     * @param target
     * @throws Throwable
     */
    public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {
        System.out.println("执行了" + method.getName() + "方法，返回结果为：" + returnValue);
    }
}
```

- spring配置文件：配置bean和aop

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--注册bean-->
    <bean id="userService" class="com.kuang.serviceImpl.UserServiceImpl"/>
    <bean id="afterLog" class="com.kuang.Log.AfterLog"/>
```

```xml
    <bean id="beforeLog" class="com.kuang.Log.BeforeLog"/>

    <!--配置AOP：方式一 （使用原生的Spring API接口）
    需要导入AOP的约束：
xmlns:aop="http://www.springframework.org/schema/aop"
                       http://www.springframework.org/schema/aop

 https://www.springframework.org/schema/aop/spring-aop.xsd
    -->
    <aop:config>
        <!--配置切入点
        id：自己定义
        expression(表达式)：execution(要执行的位置)
                            括号里面是【public修饰词 返回值 类名 方法名 参数】，如果是*，代表任意
                            execution( *
com.kuang.serviceImpl.UserServiceImpl.*(..) )
                             类：com.kuang.serviceImpl.UserServiceImpl
                             所有方法： .*
                             所有参数：(..)

        -->
        <aop:pointcut id="pointcut" expression="execution(*
com.kuang.serviceImpl.UserServiceImpl.*(..)) "/>


        <!--执行环绕增强
        advice-ref：使用哪个类
        pointcut-ref：使用哪个切入点（在哪进行环绕增强）

        这里就是：将afterLog切入到pointcut这个切入点的位置
        -->
        <aop:advisor advice-ref="afterLog" pointcut-ref="pointcut"/>
        <aop:advisor advice-ref="beforeLog" pointcut-ref="pointcut"/>

    </aop:config>
</beans>
```

- 测试运行

```java
@Test
public void test() {
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    //动态代理，代理的是接口，不要使用实现类
    UserService userService = context.getBean("userService",
UserService.class);

    userService.add();
}
```

```
com.kuang.serviceImpl.UserServiceImpl的add方法被执行了
增加了一个用户
执行了add方法，返回结果为：null

Process finished with exit code 0
```

3.2 自定义实现AOP

- 写一个自定义的切面：

```java
public class DiyPointCut {

    public void before(){
        System.out.println("============方法执行前============");
    }

    public void after(){
        System.out.println("============方法执行后============");
    }

}
```

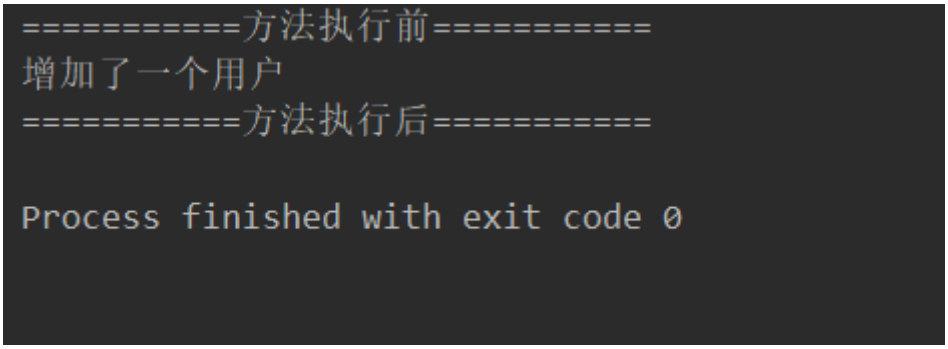
- xml的配置

```xml
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--注册bean-->
    <bean id="userService"
class="com.kuang.serviceImpl.UserServiceImpl"/>
    <bean id="afterLog" class="com.kuang.Log.AfterLog"/>
    <bean id="beforeLog" class="com.kuang.Log.BeforeLog"/>

    <bean id="diy" class="com.kuang.diy.DiyPointCut"/>
    <!-- AOP的配置
      方式二：自定义实现AOP-->
    <aop:config>
        <!--自定义切面
        ref：要引用的类
        -->
        <aop:aspect ref="diy">
            <!--切入点
            id：自定义
            expression：表达式，和之前一样
            -->
            <aop:pointcut id="pointcut" expression="execution(*
com.kuang.serviceImpl.UserServiceImpl.*(..))"/>

```

```
            <!--通知-->
            <aop:before method="before" pointcut-ref="pointcut"/>
            <aop:after method="after" pointcut-ref="pointcut"/>

        </aop:aspect>
    </aop:config>
</beans>
```

- 运行测试

```
@Test
public void test(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContextDiy.xml");

    UserService userService = context.getBean("userService",
UserService.class);
    userService.add();

}
```



- 这种方法看起来固然是简单，而且好理解，但是没有使用前一种方法的强大，毕竟这里增强的方法只是个普通的方法，而之前是接口，强大许多！

3.3 注解实现

- 切面类

```
package com.kuang.anno;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

/**
 * 使用注解的方式实现AOP
 */
@Aspect //标注这个类是一个切面
public class AnnoPointCut {
    @Before("execution(* com.kuang.serviceImpl.UserServiceImpl.*(..))")
    public void before() {
        System.out.println("==========方法执行前==========");
```

```
    }

    @After("execution(* com.kuang.serviceImpl.UserServiceImpl.*(..))")
    public void after() {
        System.out.println("============方法执行后============");
    }


    /**
     * 在环绕增强中，我们可以给定一个参数，代表我们要获取处理切入的点
     * ProceedingJoinPoint 连接点，可以获取到执行时的一些参数
     */
    @Around("execution(* com.kuang.serviceImpl.UserServiceImpl.*(..))")
    public void around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("环绕前");

        //获得签名：类的信息
        Signature signature = pjp.getSignature();
        System.out.println("signature:" + signature);

        //执行方法
        Object proceed = pjp.proceed();

        System.out.println("环绕后");

    }
}
```

- 在xml中配置

```
<?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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <!--注册bean-->
    <bean id="userService"
class="com.kuang.serviceImpl.UserServiceImpl"/>
    <bean id="afterLog" class="com.kuang.Log.AfterLog"/>
    <bean id="beforeLog" class="com.kuang.Log.BeforeLog"/>

    <bean id="anno" class="com.kuang.anno.AnnoPointCut"/>

    <!--开启注解支持
        proxy-target-class 默认为false，就是 JDK代理
                           true ： cglib代理
    -->
    <aop:aspectj-autoproxy proxy-target-class="false"/>

</beans>
```

- 测试运行

```java
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("annoAop.xml");

    UserService anno = context.getBean("userService", UserService.class);
    anno.add();
}
```