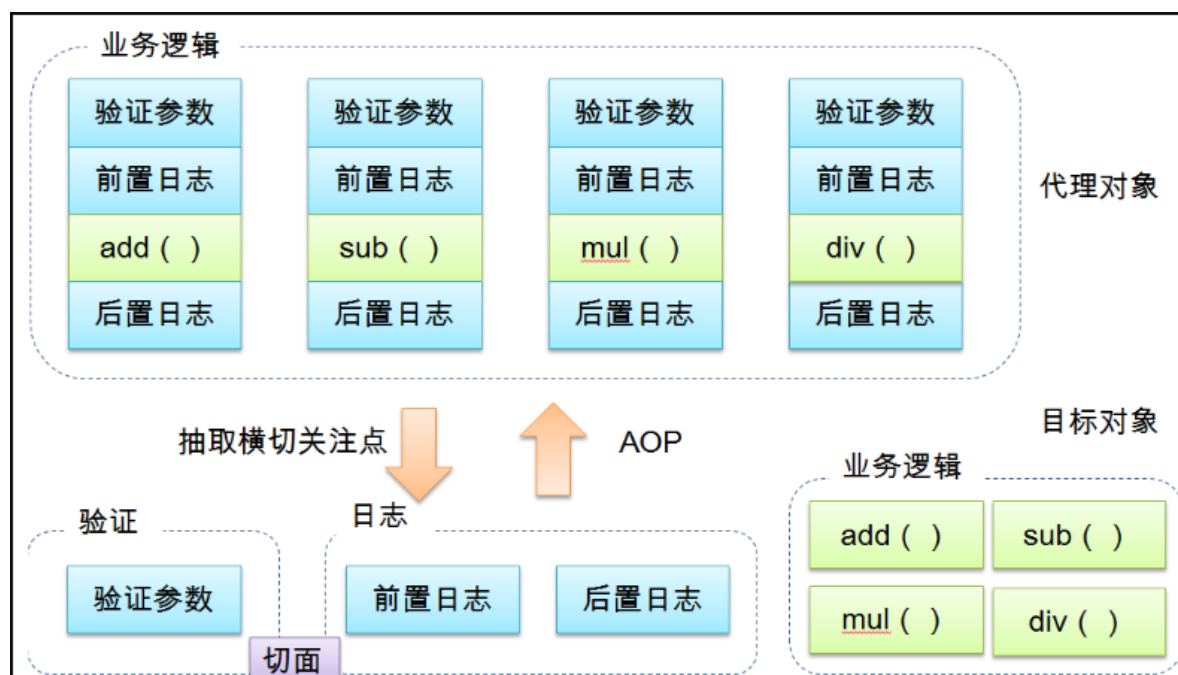


AOP

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。



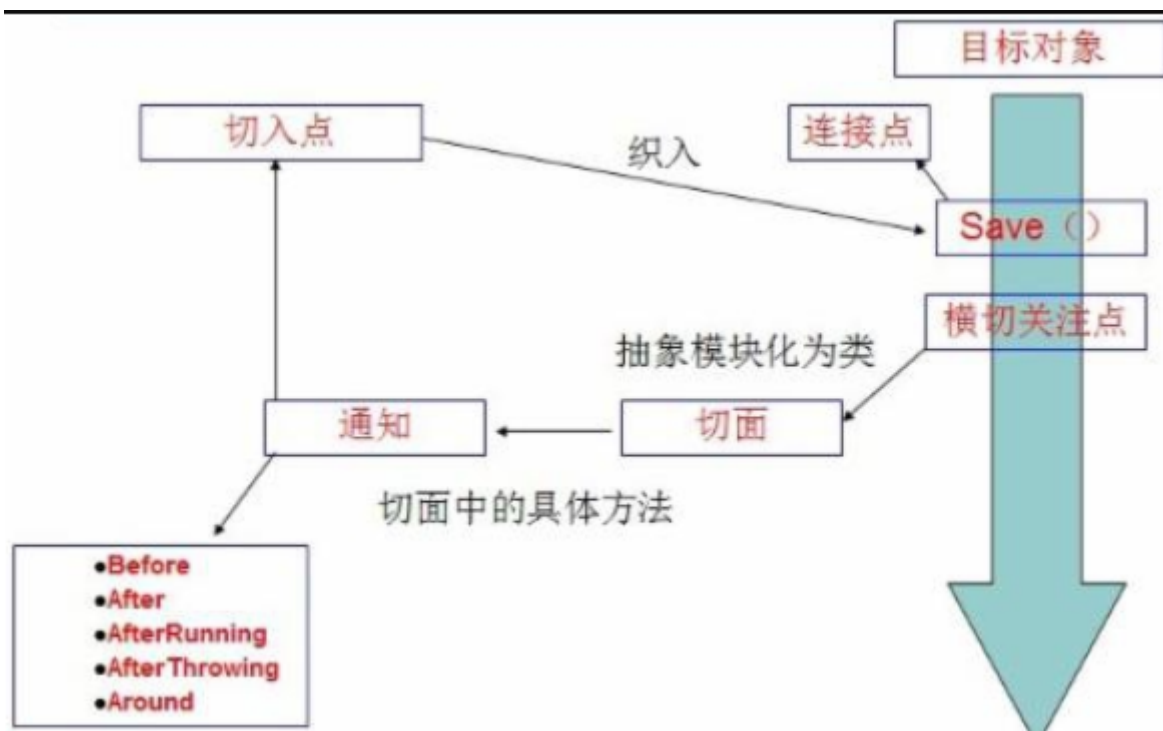
- 代理对象: 经过AOP处理后, 有了前置日志和后置日志 (这些都是自定义的功能)
- 目标对象的业务逻辑: 未经处理, 只有执行方法
- 关注点: Spring的内容, 后续讲解
- 这张图就是, 在原先的业务逻辑上 (目标对象), 经过AOP切面处理, 使用代理的方式, 加上了想要的功能 (日志)。这样子, 不用修改源码, 我们即可添加想要的功能。

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 (JoinPoint): 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
引入通知	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）

```

1 <!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
2 <dependency>
3     <groupId>org.aspectj</groupId>
4     <artifactId>aspectjweaver</artifactId>
5     <version>1.9.4</version>
6 </dependency>

```

- 写一些业务接口

```

1 package com.kuang.service;
2
3 public interface UserService {
4     public void add();
5
6     public void delete();
7
8     public void update();
9
10    public void query();
11
12
13 }

```

- 业务接口的实现类

```

1 package com.kuang.serviceImpl;
2
3 import com.kuang.service.UserService;
4
5 public class UserServiceImpl implements UserService {
6     public void add() {
7         System.out.println("增加了一个用户");
8     }
9
10    public void delete() {
11        System.out.println("删除一个用户");
12    }
13
14    public void update() {
15        System.out.println("更新用户信息");
16    }
17
18    public void query() {
19        System.out.println("查询了用户");
20    }
21 }

```

- 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:

```
1  /**
2   * 前置增强
3   */
4  public class BeforeLog implements MethodBeforeAdvice {
5      /**
6       * @param method 要执行的目标对象的方法
7       * @param args 参数
8       * @param target 目标对象
9       * @throws Throwable
10     */
11     public void before(Method method, Object[] args, Object target)
12     throws Throwable {
13         System.out.println(target.getClass().getName() + "的" +
14         method.getName() + "方法被执行了");
15     }
16 }
```

后置通知:

```
1  /**
2   * 后置增强
3   */
4  public class AfterLog implements AfterReturningAdvice {
5      /**
6       * @param returnValue 返回值
7       * @param method
8       * @param args
9       * @param target
10     * @throws Throwable
11     */
12     public void afterReturning(Object returnValue, Method method,
13     Object[] args, Object target) throws Throwable {
14         System.out.println("执行了" + method.getName() + "方法, 返回结果
15     为: " + returnValue);
16     }
17 }
```

- spring配置文件: 配置bean和aop

```
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:aop="http://www.springframework.org/schema/aop"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
6         https://www.springframework.org/schema/beans/spring-beans.xsd
7         http://www.springframework.org/schema/aop
8         https://www.springframework.org/schema/aop/spring-aop.xsd">
9
10     <!--注册bean-->
11     <bean id="userService"
12     class="com.kuang.serviceImpl.UserServiceImpl"/>
13     <bean id="afterLog" class="com.kuang.Log.AfterLog"/>
14 
```

```

13     <bean id="beforeLog" class="com.kuang.Log.BeforeLog"/>
14
15     <!--配置AOP: 方式一 (使用原生的Spring API接口)
16     需要导入AOP的约束:
17     xmlns:aop="http://www.springframework.org/schema/aop"
18         http://www.springframework.org/schema/aop
19         https://www.springframework.org/schema/aop/spring-aop.xsd
20     -->
21     <aop:config>
22         <!--配置切入点
23         id: 自己定义
24         expression(表达式): execution(要执行的位置)
25             括号里面是【public修饰词 返回值 类名 方法名 参
26             数】，如果是*, 代表任意
27             execution( *
28             com.kuang.serviceImpl.UserServiceImpl.*(..) )
29             类: com.kuang.serviceImpl.UserServiceImpl
30             所有方法: .*
31             所有参数: (..)
32         -->
33         <aop:pointcut id="pointcut" expression="execution(*
34             com.kuang.serviceImpl.UserServiceImpl.*(..)) "/>
35
36         <!--执行环绕增强
37         advice-ref: 使用哪个类
38         pointcut-ref: 使用哪个切入点 (在哪进行环绕增强)
39
40         这里就是: 将afterLog切入到pointcut这个切入点的位置
41         -->
42         <aop:advisor advice-ref="afterLog" pointcut-ref="pointcut"/>
43         <aop:advisor advice-ref="beforeLog" pointcut-ref="pointcut"/>
44     </aop:config>
45 </beans>

```

- 测试运行

```

1  @Test
2  public void test() {
3      ApplicationContext context = new
4      ClassPathXmlApplicationContext("applicationContext.xml");
5
6      //动态代理, 代理的是接口, 不要使用实现类
7      UserService userService = context.getBean("userService",
8      UserService.class);
9
10     userService.add();
11 }

```

```
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:

```
1 public class DiyPointCut {
2
3     public void before(){
4         System.out.println("=====方法执行前=====");
5     }
6
7     public void after(){
8         System.out.println("=====方法执行后=====");
9     }
10
11 }
```

- xml的配置

```
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:aop="http://www.springframework.org/schema/aop"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6                           https://www.springframework.org/schema/beans/spring-beans.xsd
7                           http://www.springframework.org/schema/aop
8                           https://www.springframework.org/schema/aop/spring-aop.xsd">
9
10     <!--注册bean-->
11     <bean id="userService"
12           class="com.kuang.serviceImpl.UserServiceImpl"/>
13     <bean id="afterLog" class="com.kuang.Log.AfterLog"/>
14     <bean id="beforeLog" class="com.kuang.Log.BeforeLog"/>
15     <bean id="diy" class="com.kuang.diy.DiyPointCut"/>
16     <!-- AOP的配置
17         方式二：自定义实现AOP-->
18     <aop:config>
19         <!--自定义切面
20         ref: 要引用的类
21         -->
22         <aop:aspect ref="diy">
23             <!--切入点
24             id: 自定义
25             expression: 表达式，和之前一样
26             -->
27             <aop:pointcut id="pointcut" expression="execution(*
28 com.kuang.serviceImpl.UserServiceImpl.*(..))"/>
```

```

29         <!--通知-->
30         <aop:before method="before" pointcut-ref="pointcut"/>
31         <aop:after method="after" pointcut-ref="pointcut"/>
32
33     </aop:aspect>
34 </aop:config>
35 </beans>

```

- 运行测试

```

1  @Test
2  public void test(){
3      ApplicationContext context = new
4      ClassPathXmlApplicationContext("applicationContextDiy.xml");
5
6      UserService userService = context.getBean("userService",
7      UserService.class);
8      userService.add();
9  }

```

```

=====方法执行前=====
增加了一个用户
=====方法执行后=====

Process finished with exit code 0

```

- 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

```

1  package com.kuang.anno;
2
3  import org.aspectj.lang.ProceedingJoinPoint;
4  import org.aspectj.lang.Signature;
5  import org.aspectj.lang.annotation.After;
6  import org.aspectj.lang.annotation.Around;
7  import org.aspectj.lang.annotation.Aspect;
8  import org.aspectj.lang.annotation.Before;
9
10 /**
11  * 使用注解的方式实现AOP
12  */
13 @Aspect //标注这个类是一个切面
14 public class AnnoPointCut {
15     @Before("execution(* com.kuang.serviceImpl.UserServiceImpl.*(..))")
16     public void before() {
17         System.out.println("=====方法执行前=====");
18     }
19 }

```

```

18     }
19
20     @After("execution(* com.kuang.serviceImpl.UserServiceImpl.*(..))")
21     public void after() {
22         System.out.println("=====方法执行后=====");
23     }
24
25
26     /**
27      * 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
28      * ProceedingJoinPoint 连接点，可以获取到执行时的一些参数
29      */
30     @Around("execution(* com.kuang.serviceImpl.UserServiceImpl.*(..))")
31     public void around(ProceedingJoinPoint pjp) throws Throwable {
32         System.out.println("环绕前");
33
34         //获得签名：类的信息
35         Signature signature = pjp.getSignature();
36         System.out.println("signature:" + signature);
37
38         //执行方法
39         Object proceed = pjp.proceed();
40
41         System.out.println("环绕后");
42     }
43 }
44 }

```

- 在xml中配置

```

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:aop="http://www.springframework.org/schema/aop"
5      xmlns:context="http://www.springframework.org/schema/context"
6      xsi:schemaLocation="http://www.springframework.org/schema/beans
7          https://www.springframework.org/schema/beans/spring-beans.xsd
8          http://www.springframework.org/schema/aop
9          https://www.springframework.org/schema/aop/spring-aop.xsd
10         http://www.springframework.org/schema/context
11         https://www.springframework.org/schema/context/spring-context.xsd">
12
13      <!--注册bean-->
14      <bean id="userService"
15          class="com.kuang.serviceImpl.UserServiceImpl"/>
16      <bean id="afterLog" class="com.kuang.Log.AfterLog"/>
17      <bean id="beforeLog" class="com.kuang.Log.BeforeLog"/>
18
19      <bean id="anno" class="com.kuang.anno.AnnoPointCut"/>
20
21      <!--开启注解支持
22          proxy-target-class 默认为false，就是 JDK代理
23          true : cglib代理
24      -->
25      <aop:aspectj-autoproxy proxy-target-class="false"/>
26
27  </beans>

```


- 测试运行

```
1 public static void main(String[] args) {  
2     ApplicationContext context = new  
    ClassPathXmlApplicationContext("annoAop.xml");  
3  
4     UserService anno = context.getBean("userService",  
        UserService.class);  
5     anno.add();  
6 }
```

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
```

环绕前

signature: void com.kuang.service.UserService.add()

=====方法执行前=====

增加了一个用户

环绕后

=====方法执行后=====

Process finished with exit code 0

|