Spring AOP 详解

作者：valarchie
原文链接：https://ld246.com/article/1570432821684
来源网站：链滴

<h3 id="情景案例">情景案例</h3>
小明辛苦忙了一整年终于完成了包含 300 个接口的业务系统项目。项目圆满上线并稳定运行了段时间了。突然有一天总监说，对于会造成数据变化的所有接口，我们必须记录用户的操作日志。然小明就吭哧吭哧给其中 150 个接口，挨个加上日志代码，累得真够呛。
过了一阵子总监又说，所有变化很少的数据全部都加上缓存，缓存涉及到刷新缓存、获取缓存、除缓存的问题。于是乎，小明就又吭哧吭哧地给其中的 100 个接口加上缓存相关的代码。
又过了一阵子总监说，所有涉及充值退款费用相关的接口，需要生成发票单存入数据库。这时候明又需要吭哧吭哧给涉及到的 50 个接口，挨个加上发票存储操作。
小明天天加班也没在工期内完成任务，并且原本的业务代码已经变得臃肿不堪了。
原本的代码：
<pre><code class="highlight-chroma">/**
 * 业务方法
 */
public static void method() {

    // 业务操作
    doBusiness();

}
</code></pre>
经过硬编码添加各种非业务性代码后的业务代码：
<pre><code class="highlight-chroma">/**
 * 业务方法
 */
public static void method() {
    // 日志操作
    doLog();
    // 业务操作
    doBusiness();
    // 缓存操作
    doLog();
    // 发票操作
    doReceipt();

}
</code></pre>
<blockquote>
读者应该能明显感受到在没有 AOP 代理的情况下的缺点
</blockquote>

业务代码和非业务代码混杂在一起，原本清晰的业务流程淹没在与业务不相关的代码中。
增加非业务性的功能时，都需要手工硬编码去实现，费时费力。
代码变得不好维护，一是代码耦合度高，二是需要通过硬编码的方式去拓展或者修改功能。

<hr>
<h3 id="AOP是什么-">AOP 是什么？</h3>
在软件业，AOP 为 Aspect Oriented Programming 的缩写，意为：面向切面编程，通过预编方式和运行期动态代理实现程序功能的统一维护的一种技术。主要目标还是致力于解耦，我们可以看解耦这一理念贯穿于我们的整个编码工作中。我们通过各种设计模式或者设计原则来使得对象之间解。通过 Spring IOC 容器中利用依赖注入使得对象之间的耦合度更低。而 AOP 的思想解耦得更彻底

通过动态的添加功能来增强实现，并且做到毫无代码的侵入性。利用 AOP 可以对业务逻辑和非业务辑的部分进行隔离，可以提取非业务逻辑的部分，提高程序的可重用性，同时提高了开发的效率。</
>
<blockquote>
如何理解“切面”二字呢？
</blockquote>
<img src="https://ld246.com/images/img-loading.svg" alt="15703790331.png" data-src
"https://b3logfile.com/file/2019/10/15703790331-c45554fb.png?imageView2/2/interlace/1/fo
mat/jpg">
<blockquote>
我们的业务流程方法都是自顶向下垂直的，而当我们需要给这些业务方法统一加上某些非业务功的话，就会发现这些非业务功能方法在图上会连成一条直线，并与原来的业务流程方法垂直横切。</
>
</blockquote>
<h3 id="为什么使用AOP-">为什么使用 AOP？</h3>

核心业务代码与切面代码解耦，切面代码对核心业务代码完全无侵入，遵守单一职责原则，完全离核心业务代码与切面代码。

低耦合带来可维护性高，修改或者新增一个切面代码仅需集中在一处进行更改。低耦合也意味着面代码可复用性高。

Spring IOC 容器天然地为 AOP 的实现提供了便利，IOC 和 AOP 的结合使得 Spring 的解耦能更强。

<h3 id="AOP例子">AOP 例子</h3>
先声明切面类：
<pre><code class="highlight-chroma"><span class="highlight
cl">/**
 * 注解@Aspect
识该类为切面类
 */
@Component
@Aspect
public class Perso
Aspect {

    // 通过表达式
义切入点
    @Pointcut("exe
ution(* com.valarchie.aop.MeetingServiceImpl.meeting(..))")
    public void con
erence() {}

    // 前置通知
    @Before("meet
ng()")
    public void tak
Seats() {
        System.out.pr

```
ntln("开会前，找到位置坐");
    }

    // 前置通知
    @Before("meeting()")
    public void silenceCellPhones() {
        System.out.println("开会前，手机调成静音");
    }

    // 后置通知
    @After("meeting()")
    public void summary() {
        System.out.println("开会后，写总结报告");
    }

}
```

创建要被代理的接口，即 MeetingService 会议服务

```
public interface MeetingService {
    void meeting();
}
```

创建 MeetingService 会议服务的具体实现

```
@Component
public class MeetingServiceImpl implements MeetingService {

    @Override
    public void meeting() {
        System.out.println("会议进行中..");
    }

}
```

定义 AOP 配置

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass = true)
@ComponentScan("com.valarchie")
public class AppC
```

nfig {

}
</code></pre>
<blockquote>

注解 @EnableAspectJAutoProxy 开启代理;
如果属性 proxyTargetClass 默认为 false, 表示使用 jdk 动态代理织入增强;
如果属性 proxyTargetClass 设置为 true，表示使用 Cglib 动态代理技术织入增强;
如果属性 proxyTargetClass 设置为 false，但是目标类没有声明接口， Spring aop 还是会使用 glib 动态代理，也就是说非接口的类要生成代理都用 Cglib。

</blockquote>
测试 AOP：
<pre><code class="highlight-chroma">@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = AppConfig.class)
public class AopTest {

    @Autowired
    private MeetingServiceImpl meetingService;

    @Test
    public void testAopAnnotation() {
        meetingService.meeting();
    }

}

</code></pre>
运行结果：
<pre><code class="highlight-chroma">开会前，手机调成静音
开会前，找到位置

会议进行中..
开会后，写总结报

</code></pre>
在这个 AOP 的例子当中我们没有在会议服务实现类当中硬编码需要添加的切面功能，而是通过外新建一个类来描述切面，以及需要在切面上增强的功能。这样的实现是不是更优雅呢？
关于切入点的表达式稍微解析一下：
<pre><code class="highlight-chroma">例如定义切入点表达式 execution (* com.sample.service.impl..*.*(..))

execution()是最用的切点函数，其语法如下所示：

整个表达式可以

为五个部分：

 1、execution():
达式主体。

 2、第一个*号：
示返回类型，*号表示所有的类型。

 3、包名：表示需
拦截的包名，后面的两个句点表示当前包和当前包的所有子包，
 com.sample.servi
e.impl包、子孙包下所有类的方法。

 4、第二个*号：
示类名，*号表示所有的类。

 5、*(..):最后这个
号表示方法名，*号表示所有的方法，后面括弧里面表示方法
 的参数，两个句
表示任何参数。
</code></pre>
<blockquote>
以上的例子当中涉及不少 AOP 概念，接下来我们针对这些概念进行逐一解释。
</blockquote>
<h3 id="AOP中的概念阐述">AOP 中的概念阐述</h3>

连接点（Joinpoint）
程序执行的某个特定位置：如类开始初始化前、类初始化后、类某个方法调用前、调用后、方法抛出
常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些点中的特定点就称为“连接点”
Spring 仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时以及方法调用前
这些程序执行点织入增强。连接点由两个信息确定：第一是用方法表示的程序执行点；第二是用相对
表示的方位。

切点（Pointcut）
每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序
中客观存在的事物。AOP 通过“切点”定位特定的连接点。连接点相当于数据库中的记录，而切点
当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。在 Spring 中，切
通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件，S
ring AOP 的规则解析引擎负责切点所设定的查询条件，找到对应的连接点。其实确切地说，不能称
为查询连接点，因为连接点是方法执行前、执行后等包括方位信息的具体程序执行点，而切点只定位
某个方法上，所以如果希望定位到具体连接点上，还需要提供方位信息。

增强（Advice）
增强是织入到目标类连接点上的一段程序代码，在 Spring 中，增强除用于描述一段程序代码外，还
有另一个和连接点相关的信息，这便是执行点的方位。结合执行点方位信息和切点信息，我们就可以
到特定的连接点。

目标对象（Target）
增强逻辑的织入目标类。如果没有 AOP，目标业务类需要自己实现所有逻辑，而在 AOP 的帮助下，
标业务类只实现那些非横切逻辑的程序逻辑，而性能监视和事务管理等这些横切逻辑则可以使用 AOP

动态织入到特定的连接点上。

引介（Introduction）
引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口通过 AOP 的引介功能，我们可以动态地为该业务类添加接口的实现逻辑，让业务类成为这个接口的现类。

织入（Weaving）
织入是将增强添加对目标类具体连接点上的过程。AOP 像一台织布机，将目标类、增强或引介通过 AP 这台织布机天衣无缝地编织到一起。根据不同的实现技术，AOP 有三种织入的方式：
a、编译期织入，这要求使用特殊的 Java 编译器。
b、类装载期织入，这要求使用特殊的类装载器。
c、动态代理织入，在运行期为目标类添加增强生成子类的方式。
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。

代理（Proxy）
一个类被 AOP 织入增强后，就产出了一个结果类，它是融合了原类和增强逻辑的代理类。根据不同代理方式，代理类既可能是和原类具有相同接口的类，也可能就是原类的子类，所以我们可以采用调原类相同的方式调用代理类。

切面（Aspect）
切面由切点和增强（引介）组成，它既包括了横切逻辑的定义，也包括了连接点的定义，Spring AOP 就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的连接点中。

