

11 | 剑走偏锋：面向切面编程

2019-10-04 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 17:00 大小 13.64M



你好，我是四火。

今天我们要接触一个和 MVC 密切相关的，能带来思维模式改变的编程范型——面向切面编程（AOP，Aspect Oriented Programming）。

“给我一把锤子，满世界都是钉子”

我记得曾经有这样一个相当流行的观点，是说，编程语言只需要学习一门就够了，学那么多也没有用，因为技术是一通百通的，别的编程语言可以说是大同小异。我相信至今抱有这种观点的程序员也不在少数。

可惜，事实远没有那么美好。这个观点主要有两处值得商榷：

其一，不同的技术，在一定程度上确实是相通的，可是，技术之间的关联性，远不是“一通百通”这四个简简单单的字能够解释的。妄想仅仅凭借精通一门编程语言，就能够自动打通其它所有编程语言的任督二脉，这是不现实的。

其二，通常来说，说编程语言大同小异其实是很不客观的。编程语言经过了长时间的发展演化，如今已经发展出非常多的类型，用作编程语言分类标准之一的编程范型也可谓是百花齐放。

因此我们要学习多种编程语言，特别是那些能带来新的思维模式的编程语言。现在，把这个观点泛化到普遍的软件技术上，也一样适用。我们都知道要“一切从实际出发”，都知道要“具体问题具体分析”，可是，**在眼界还不够开阔的时候，特别是职业生涯的早期，程序员在武器库里的武器还非常有限的时候，依然无法避免“给我一把锤子，满世界都是钉子”，在技术选择的时候眼光相对局限。**

所以我们要学习全栈技术，尤其是要学习这些不一样，但一定层面上和已掌握知识相通的典型技术。今天我们要学习的这项在 MVC 框架中广泛使用的技术，是和面向对象编程一类层面的编程范型，叫做面向切面编程。

互联网有许多功能，如果使用传统的基于单个请求处理流程的方式来编码，代码就会非常繁琐，而使用 AOP 的方式，代码可以得到很大程度上的简化。希望通过今天的学习，你的武器库里，能够多一把重型机枪。

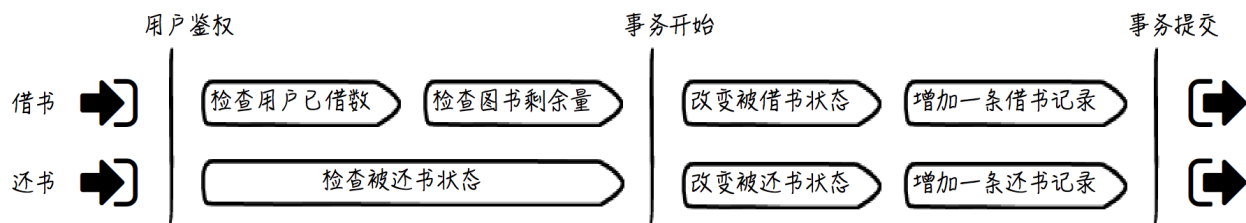
AOP 的概念

面向切面编程是一种通过横切关注点（Cross-cutting Concerns）分离来增强代码模块性的方法，它能够在不修改业务主体代码的情况下，对它添加额外的行为。

不好理解吗？没关系，我们来对它做进一步的说明。

首先需要明确的是，AOP 的目标是增强代码模块性，也就是说，本质上它是一种“解耦”的方法，在这方面它和我们之前介绍的分层等方法类似的，可是，它分离代码的角度与我们传统、自然的模块设计思路截然不同。

我们来看下面这样一个例子，对于图书馆系统来说，有许多业务流程，其中借书和还书是最典型的两条。对于这些业务流程来说，从图书系统接收到请求开始，需要完成若干个步骤，但这些步骤都有一些“共性”，比如鉴权，比如事务控制：



那么，如果我们按照自然的思考方式，我们会把代码按照流程分解成一个一个的步骤，在每个步骤完成的前后添加这些“共性”逻辑。可是这样，这些逻辑就会散落在代码各处了，即便我们把它按照重复代码抽取的原则，抽出来放到单独的方法中，这样的方法的“调用”还是散落在各处，无论是对软件工程上的可维护性，还是代码阅读时对于业务流程的专注度，都是不利的。

藉由 AOP 则可以有效地解决这些问题，对于图中横向的业务流程，我们能够保持它们独立不变，而把鉴权、事务这样的公共功能，彻底拿出去，放到单独的地方，这样整个业务流程就变得纯粹和干净，没有任何代码残留的痕迹，就好像武林高手彻底隐形了一般，但是，功能却没有任何丢失。就好比面条一般顺下来的业务流程，水平地切了几刀，每一刀，都是一个 AOP 的功能实现。

我们可能在 Java 的世界中谈论 AOP 比较多，但请注意，它并不是 Java 范畴的概念，它不依赖于任何框架，也和编程语言本身无关。

Spring 中的应用

[Spring](#) 作为一个应用程序框架，提供了对于 AOP 功能上完整的支持，下面让我们通过例子来学习。还记得我们在 [\[第 08 讲\]](#) 中举例介绍的将图书借出的方法吗？

复制代码

```
1 public class BookService {  
2     public Book lendOut(String bookId, String userId, Date date) { ... (0) }  
3 }
```

现在，我们要给很多的业务方法以 AOP 的方式添加功能，而 `lendOut` 就是其中之一。定义一个 `TransactionAspect` 类：

复制代码

```
1 public class TransactionAspect {
2     public void doBefore(JoinPoint jp) { ... (1) }
3     public void doAfter(JoinPoint jp) { ... (2) }
4     public void doThrowing(JoinPoint jp, Throwable ex) { ... (3) }
5     public void doAround(ProceedingJoinPoint pjp) throws Throwable {
6         ... (4)
7         pjp.proceed();
8         ... (5)
9     }
10 }
```

你看，我给每一处可以实现的代码都用数字做了标记。我们希望在 `doBefore` 方法中添加事务开始逻辑，`doAfter` 方法中添加事务结束的提交逻辑，`doThrowing` 方法中添加事务失败的回滚逻辑，而在 `doAround` 方法中业务执行前后添加日志打印逻辑，其中的 `pjp.proceed()` 方法表示对原方法的调用。

接着，我们需要写一些 XML 配置，目的就是把原方法和 AOP 的切面功能连接起来。配置片段如下：

 复制代码

```
1 <bean id="bookService" class="xxx.BookService"></bean>
2 <bean id="transactionAspect" class="xxx.TransactionAspect"></bean>
3
4 <aop:config>
5     <aop:pointcut expression="execution(* xxx.BookService.*(..))" id="transactionPointcut"
6     <aop:aspect ref="transactionAspect">
7         <aop:before method="doBefore" pointcut-ref="transactionPointcut"/>
8         <aop:after-returning method="doAfter" pointcut-ref="transactionPointcut"/>
9         <aop:after-throwing method="doThrowing" pointcut-ref="transactionPointcut" throwing:
10         <aop:around method="doAround" pointcut-ref="transactionPointcut"/>
11     </aop:aspect>
12 </aop:config>
```

在这段配置中，前两行分别是对 `BookService` 和 `TransactionAspect` 这两个 Bean 的声明，接下来在 `aop:config` 中，我们定义了 `pointcut` 的切面匹配表达式，表示要捕获 `BookService` 的所有方法，并在 `aop:aspect` 标签内定义了我们希望实施的 AOP 功能。

在实际执行的过程中，如果没有异常抛出，上述这些逻辑的执行顺序将是：

1 (1) → (4) → (0) → (5) → (2)

实现原理

讲了 AOP 怎样配置，怎么表现，现在我要来讲讲它的实现原理了。通过这部分内容，希望你搞清楚，为什么不需要对代码做任何改动，就可以在业务逻辑的流水中切一刀，插入我们想要执行的其它逻辑呢？

对于常见的实现，我们根据其作用的不同时间阶段进行分类，有这样两种：

编译期间的静态织入，又称为编译时增强。织入 (Weaving)，指的是将切面代码和源业务代码链接起来的过程。[AspectJ](#) 就是这样一个面向切面的 Java 语言扩展，称呼其为语言的“扩展”，就是因为它扩展了 Java 语言的语法，需要特定的编译器来把 AspectJ 的代码编译成 JVM 可识别的 class 文件。

运行期间的动态代理，又称为运行时增强。这种方式是在程序运行时，依靠预先创建或运行时创建的代理类来完成切面功能的。比如 JDK 基于接口的动态代理技术，或 [CGLib](#) 基于类的代理对象生成技术就属于这一种。

Spring AOP 默认支持的是后者——运行期间的动态代理。至于具体实现，通常来说，我们应该优先考虑使用 JDK 的动态代理技术；但是如果目标类没有实现接口，我们只能退而求其次，使用 CGLib。


动态代理的方式由于在运行时完成代理类或代理对象的创建，需要用到 Java 的拦截、反射和字节码生成等技术，因此运行时的性能表现往往没有静态织入好，功能也有较多限制，但是由于使用起来简便（不需要语言扩展，不需要特殊的编译器等），它的实际应用更为广泛。

控制反转 IoC

通过 AOP 我们知道，某些问题如果我们换个角度来解决，会很大程度地简化代码。现在，让我们来了解在 Spring 中另一个经常和面向切面编程一起出现的概念——控制反转。控制反转是一种设计思想，也是通过“换个角度”来解决问题的。

控制反转，IoC，即 Inversion of Control，言下之意，指的是把原有的控制方向掉转过来了。在我们常规的程序流程中，对象是由主程序流程创建的，例如，在业务流程中使用 new 关键字来创建依赖对象。

但是，当我们使用 Spring 框架的时候，**Spring 把对象创建的工作接管过来，它作为对象容器，来负责对象的查找、匹配、创建、装配，依赖管理，等等。而主程序流程，则不用关心对象是怎么来的，只需要使用对象就可以了。**我们还是拿 BookService 举例子：

 复制代码

```
1 public class BookService {
2     @Autowired
3     private BookDao bookDao;
4     @Autowired
5     private LoanDao loanDao;
6     public Book lendOut(String bookId, String userId, Date date) {
7         bookDao.update( ... );
8         loanDao.insert( ... );
9     }
10 }
```

比如 BookService 的借出方法，假如它的实现中，我们希望：

调用数据访问对象 bookDao 的方法来更新被借书的状态；

调用借阅行为的访问对象 loanDao 来增加一条借阅记录。

在这种情况下，我们可以通过 @Autowired 注解，让容器将实际的数据访问对象注入进来，主程序流程不用关心“下一层”的数据访问对象到底是怎么创建的，怎么初始化的，甚至是怎么注入进来的，而是直接用就可以了，因为这些对象都已经被 Spring 管理起来了。

如果这些注入的对象之间还存在依赖关系，初始化它们的顺序就至关重要了，可是在这种情况下，Service 层依然不用关心，因为 Spring 已经根据代码或配置中声明的依赖关系自动确定了。总之，Service 层的业务代码，只管调用其下的数据访问层的方法就好了。

读到这里，你可能会回想起前文 AOP 的内容，和 IoC 似乎有一个共同的特点：都是**为了尽可能保证主流程的纯粹和简洁**，而将这些不影响主流程的逻辑拿出去，只不过这两种技

术，“拿出去”的是不同的逻辑。值得注意的是，对象之间的依赖关系，各层之间的依赖关系，并没有因为 IoC 而发生任何的改变。

IoC 在实现上包含两种方式，一种叫做依赖查找 (DL, Dependency Lookup)，另一种叫做依赖注入 (DI, Dependency Injection)。二者缺一不可，Spring 容器做到了两者，就如同上面的例子，容器需要先查找到 bookDao 和 loanDao 所对应的对象，再把它注入进来。当然，我们平时听到的更多是第二种。

有了一个大致的感受，那么 IoC 到底能带来什么好处呢？我觉得主要有这样两个方面：

资源统一配置管理。这个方面很好，但并不是 IoC 最大的优势，因为，如果你不把资源交给容器管理，而是自己建立一个资源管理类来管理某项资源，一样可以得到“统一管理”的所有优势。

业务代码不再包含依赖资源的访问逻辑，因此资源访问和业务流程的代码解耦开了。我觉得这里的“解耦”才是 IoC 最核心的优势，它让各层之间的依赖关系变得松散。就如同上面的代码例子一样，如果哪一天我想把它依赖的 bookDao 和 loanDao 替换掉（比如，我想为 Service 层做测试），Service 一行代码都不用改，它压根都不需要知道。

总结思考

今天我们一起学习了面向切面编程，从学习概念，熟悉配置，到了解实现原理，希望你对于 AOP 已经有了一个清晰的认识，在未来设计和开发系统的时候，无论技术怎样演进，框架怎么变化，始终知道什么时候需要它，并能够把它从你的武器库中拿出来使用。

现在我来提两个问题，我们一起讨论吧：

你过去的项目中有没有应用 AOP 的例子，能说说吗？


我介绍了 AOP 的优点，但却没有提到它的缺点，但其实任何技术都是有两面性的，你觉得 AOP 的缺点都有哪些呢？

选修课堂：实践 AOP 的运行时动态代理

我们学习了 AOP 的实现原理，知道其中一种办法是通过 JDK 的动态代理技术来实现的。现在，我们就来写一点代码，用它实现一个小例子。


首先，请你准备好一个项目文件夹，我们会在其中创建一系列文件。你可以使用 Eclipse 来管理项目，也可以自己建立一个独立的文件夹，这都没有关系。

现在建立 BookService.java，这次我们把 BookService 定义为一个接口，包含 lendOut 方法，同时也创建它的实现 BookServiceImpl：

 复制代码


```
1 import java.text.MessageFormat;
2 import java.util.Date;
3
4 interface BookService {
5     void lendOut(String bookId, String userId, Date date);
6 }
7
8 class BookServiceImpl implements BookService {
9     @Override
10    public void lendOut(String bookId, String userId, Date date) {
11        System.out.println(MessageFormat.format("{0}: The book {1} is lent to {2}.", date, bookId, userId));
12    }
13 }
```

然后，我们建立一个 ServiceInvocationHandler.java，在这里我们可以定义代理对象在对原对象的方法调用前后，添加的额外逻辑：

 复制代码

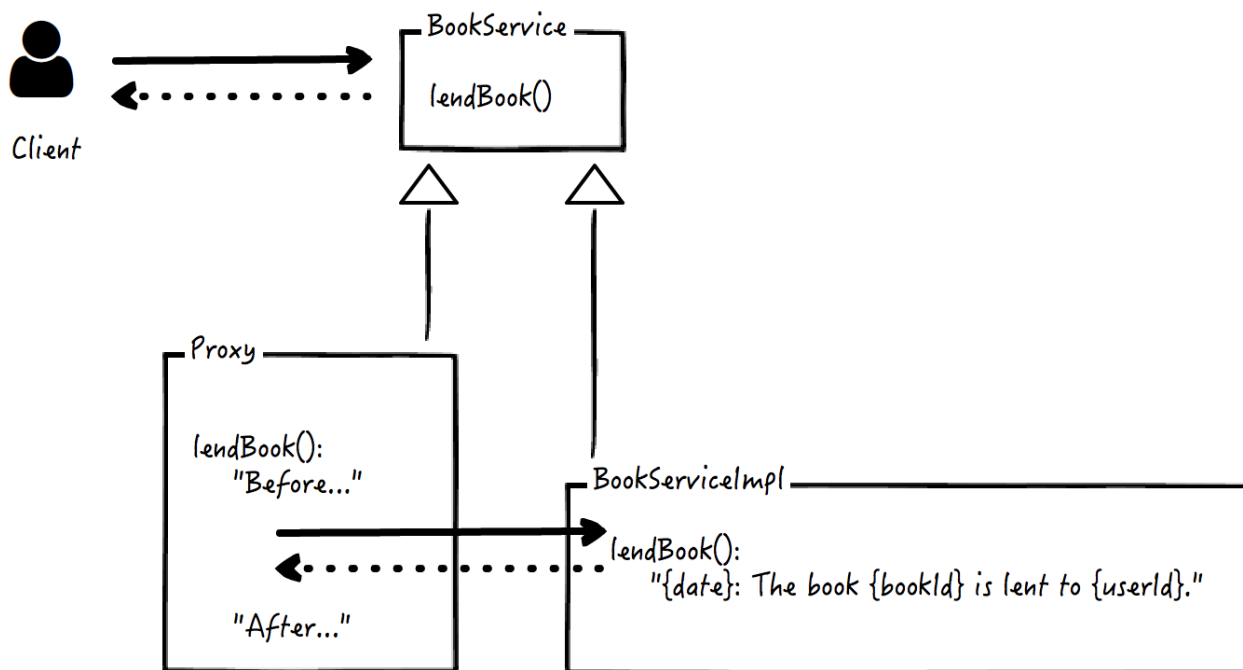
```
1 import java.lang.reflect.InvocationHandler;
2 import java.lang.reflect.Method;
3
4 class ServiceInvocationHandler implements InvocationHandler {
5     private Object target;
6
7     public ServiceInvocationHandler(Object target) {
8         this.target = target;
9     }
10
11    @Override
12    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
13        System.out.println("Before...");
14        Object result = method.invoke(this.target, args);
15        System.out.println("After...");
16        return result;
17    }
18 }
```


接着，我们建立一个 Client.java 类，作为程序的起点，通过动态代理的方式来调用源代码中的 lendOut 方法：

 复制代码

```
1 import java.lang.reflect.Proxy;
2 import java.util.Date;
3
4 public class Client {
5     public static void main(String[] args) throws Exception {
6         BookService bookService = (BookService) Proxy.newProxyInstance(
7             BookService.class.getClassLoader(),
8             new Class[]{ BookService.class },
9             new ServiceInvocationHandler(new BookServiceImpl())
10        );
11        bookService.lendOut("123", "456", new Date());
12    }
13 }
```

你看，我们创建了一个动态代理对象，并赋给 bookService，这个代理对象实际是会调用 BookServiceImpl 的，但调用的前后打印了额外的日志。并且，这个代理对象也实现自 BookService 接口，因此，对于 BookService 的使用者来说，它实际并不知道调用到的是 BookServiceImpl 还是它的代理对象。请看图示：



好，现在我们把这些代码编译一下：

复制代码

```
1 javac BookService.java ServiceInvocationHandler.java Client.java
```

你应该能看到它们的 class 文件分别生成了。

最后，执行 Client 的 main 方法，就能看到相应的执行结果，它显示 `lendBook` 方法前后的 AOP 的逻辑被实际执行了：

复制代码

```
1 java Client
2 Before...
3 8/10/19 11:42 AM: The book 123 is lent to 456.
4 After...
```

扩展阅读

Spring 官方文档中[关于 AOP 的教程](#)，如果你希望看到中文版，那么互联网上有不少对于这部分的翻译，只不过对应的 Spring 版本不同，内容大致是一样的，比如[这一篇](#)。

[Comparing Spring AOP and AspectJ](#)，这是一篇关于静态织入和动态代理这两种 AOP 方式比较的文章。

对于 AspectJ，如果想一瞥其扩展的语法语义，维基百科的[词条](#)就足矣；如果想了解某些细节，请参阅[官方文档](#)。

 极客时间

全栈工程师修炼指南

从全栈入门到技能实战

熊燚
Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | MVC架构解析：控制器（Controller）篇

下一篇 12 | 唯有套路得人心：谈谈Java EE的那些模式

精选留言 (2)

 写留言



sky

2019-10-05

iOS里用的runtime的一些方法也是aop了

展开 ∨



我叫徐小晋

2019-10-05

老师您好。SpringBoot中分层，model层,dao层，service层，controller层。上层通过@Autowired来使用下层的方法，这个就是文中说到的loc吗？

展开 ∨

