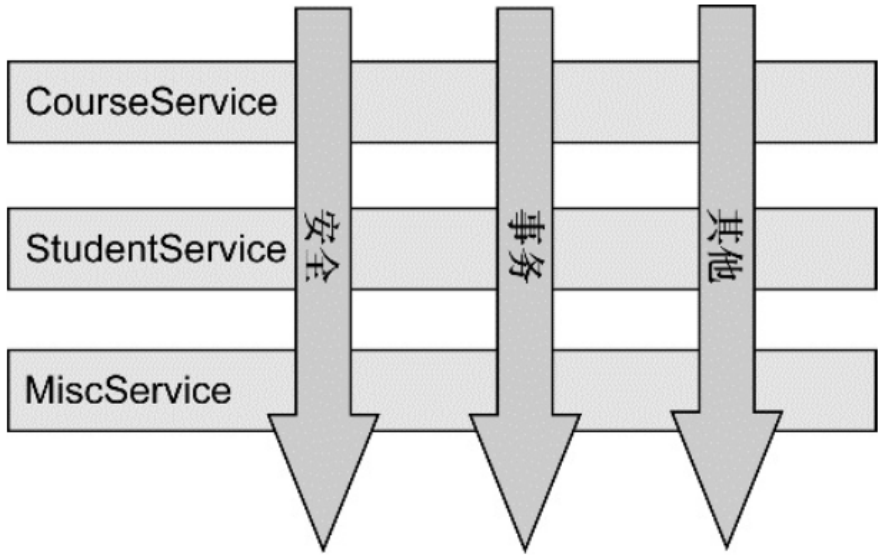
Spring AOP——Spring 中面向切面编程

一、AOP——另一种编程思想

1.1 什么是 AOP

AOP （Aspect Orient Programming）,直译过来就是 面向切面编程。AOP 是一种编程思想，是面向对象编程（OOP）的一种补充。面向对象编程将程序抽象成各个层次的对象，而面向切面编程是将程序抽象成各个切面。
从《Spring实战（第4版）》图书中扒了一张图：



切面实现了横切关注点（跨多个应用对象的逻辑）的模块化

从该图可以很形象地看出，所谓切面，相当于应用对象间的横切点，我们可以将其单独抽象为单独的模块。

1.2 为什么需要 AOP

想象下面的场景，开发中在多个模块间有某段重复的代码，我们通常是怎么处理的？显然，没有人会靠“复制粘贴”吧。在传统的面向过程编程中，我们也会将这段代码，抽象成一个方法，然后在需要的地方分别调用这个方法，这样当这段代码需要修改时，我们只需要改变这个方法就可以了。然而需求总是变化的，有一天，新增了一个需求，需要再多出做修改，我们需要再抽象出一个方法，然后再在需要的地方分别调用这个方法，又或者我们不需要这个方法了，我们还是得删除掉每一处调用该方法的地方。实际上涉及到多个地方具有相同的修改的问题我们都可以通过 AOP 来解决。

1.3 AOP 实现分类

AOP 要达到的效果是，保证开发者不修改源代码的前提下，去为系统中的业务组件添加某种通用功能。AOP 的本质是由 AOP 框架修改业务组件的多个方法的源代码，看到这其实应该明白了，AOP 其实就是前面一篇文章讲的代理模式的典型应用。
按照 AOP 框架修改源代码的时机，可以将其分为两类：

- 静态 AOP 实现， AOP 框架在编译阶段对程序源代码进行修改，生成了静态的 AOP 代理类（生成的 *.class 文件已经被改掉了，需要使用特定的编译器），比如 AspectJ。
- 动态 AOP 实现， AOP 框架在运行阶段对动态生成代理对象（在内存中以 JDK 动态代理，或 CGlib 动态地生成 AOP 代理类），如 SpringAOP。

下面给出常用 AOP 实现比较

类别	机制	原理	优点	缺点
静态 AOP	静态织入	在编译起，切面直接以字节码的形式编译到目标字节码文件中	对系统无性能影响	灵活性不够
动态 AOP	JDK 动态代理	在运行期，目标类加载后，为接口动态生成代理类，将切面织入到代理类中	相对于静态 AOP 更加灵活	切入的关注点需要实现接口。对系统有一点性能影响
动态字节码生成	CGLIB	在运行期，目标类加载后，动态生成目标类的子类，将切面逻辑加入到子类中	没有接口也可以织入	扩展类的实例方法用 final 修饰时，则无法进行织入
自定义类加载器		在运行期，目标类加载前，将切面逻辑加到目标字节码里	可以对绝大部分类进行织入	代码中如果使用了其他类加载器，则这些类将不会织入
字节码转换		在运行期，所有类加载器加载字节码前进行拦截	可以对所有类进行织入	

二、AOP 术语

AOP 领域中的特性术语：

- 通知（Advice）：AOP 框架中的增强处理。切面的工作被称为通知。通知定义了切面是什么以及何时使用。除了描述切面要完成的工作，通知还解决了何时执行这个工作的问题。。
- 连接点（join point）：连接点表示应用执行过程中能够插入切面的一个点，这个点可以是方法的调用、异常的抛出。在 Spring AOP 中，连接点总是方法的调用。
- 切点（PointCut）：可以插入增强处理的连接点。如果说通知定义了切面的“什么”和“何时”的话，那么切点就定义了“何处”。因此，切点其实就是定义了需要执行在哪些连接点上执行通知。
- 切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容——它是什么，在何时和在何处完成其功能。
- 引入（Introduction）：引入允许我们向现有的类添加新的方法或者属性。
- 织入（Weaving）：将增强处理添加到目标对象中，并创建一个被增强的对象，这个过程就是织入。

> **编译期**：切面在目标类编译时被织入。这种方式需要特殊的编译器。AspectJ的织入编译器就是以这种方式织入切面的。
> **类加载期**：切面在目标类加载到JVM时被织入。这种方式需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ 5的加载时织入就支持这种方式织入切面。
> **运行期**：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态的创建一个代理对象。Spring AOP就是以这种方式织入切面的。

概念看起来总是有点懵，并且上述术语，不同的参考书籍上翻译还不一样，所以需要慢慢在应用中理解。

三、初步认识 Spring AOP

3.1 Spring AOP 的特点

AOP 框架有很多种，1.3节中介绍了 AOP 框架的实现方式有可能不同， Spring 中的 AOP 是通过动态代理实现的。不同的 AOP 框架支持的连接点也有所区别，例如，AspectJ 和 JBoss,除了支持方法切点，它们还支持字段和构造器的连接点。而 Spring AOP 不能拦截对对象字段的修改，也不支持构造器连接点,我们无法在 Bean 创建时应用通知。

3.2 Spring AOP 的简单例子

下面先上代码，对着代码说比较好说，看下面这个例子：

`pom` 文件添加依赖：

```
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.3.3</version>
      <scope>compile</scope>
    </dependency>
```

首先创建一个接口 `IBuy.java`

```
package aspectJ.demo.entity;

public interface IBuy {
    String buy();
}
```

`Boy` 和 `Gril` 两个类分别实现了这个接口：
`Boy.java`

```
package aspectJ.demo.entity;



import org.springframework.stereotype.Component;

@Component
public class Boy implements IBuy{
    @Override
    public String buy() {
        System.out.println("男孩买了一个游戏机");
        return "游戏机";
    }
}
```

`Girl.java`

```
package aspectJ.demo.entity;

import org.springframework.stereotype.Component;

@Component
public class Girl implements IBuy {
    public String buy() {
        System.out.println("女孩买了一件漂亮的衣服");
        return "衣服";
    }
}
```

配置文件, `AppConfig.java`

```java
package aspectJ.demo.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan("aspectJ.demo")
public class AppConfig {
}
```

测试类， `App.java`

```java
package aspectJ.demo;

import aspectJ.demo.config.AppConfig;
import aspectJ.demo.entity.Boy;
import aspectJ.demo.entity.Girl;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        Boy boy = context.getBean("boy", Boy.class);
        Girl girl = (Girl) context.getBean("girl");
        boy.buy();
        girl.buy();
    }
}
```

运行结果：

> 男孩买了一个游戏机
> 女孩买了一件漂亮的衣服

这里运用SpringIOC里的自动部署。现在需求改变了，我们需要在男孩和女孩的 buy 方法之前，需要打印出“男孩女孩都买了自己喜欢的东西”。用 Spring AOP 来实现这个需求只需下面几个步骤：

1、 **既然用到 Spring AOP, 首先在 `pom` 文件中引入相关依赖：**

```xml
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
```

2、 **定义一个切面类，** `BuyAspectJ.java`

```java
package aspectJ.demo.config;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class BuyAspectJ {

    @Before("execution(* aspectJ.demo.entity.IBuy.buy(..))")
    public void haha(){
        System.out.println("男孩女孩都买自己喜欢的东西");
    }
}
```

这个类，我们使用了注解 `@Component` 表明它将作为一个Spring Bean 被装配，使用注解 `@Aspect` 表示它是一个切面。
类中只有一个方法 `haha` 我们使用 `@Before` 这个注解，表示他将在方法执行之前执行。关于这个注解后文再作解释。
参数 `("execution(* aspectJ.demo.entity.IBuy.buy(..))")` 声明了切点，表明在该切面的切点是 `aspectJ.demo.entity.IBuy` 这个接口中的 `buy` 方法。至于为什么这么写，下文再解释。

3、 **在配置文件中启用AOP切面功能**

```
package aspectJ.demo.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan("aspectJ.demo")
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class AppConfig {
}
```

我们在配置文件类增加了 `@EnableAspectJAutoProxy` 注解，启用了 AOP 功能，参数 `proxyTargetClass` 的值设为了 true 。默认值是 false，两者的区别下文再解释。
OK，下面只需测试代码,运行结果如下：

```
男孩女孩都买自己喜欢的东西
男孩买了一个游戏机
男孩女孩都买自己喜欢的东西
女孩买了一件漂亮的衣服
```

四、通过注解配置 Spring AOP

4.1 通过注解声明切点指示器

项目类型	描述
arg()	限定连接点方法参数
@args()	通过连接点方法参数上的注解进行限定
execution()	用于匹配是连接点的执行方法
this()	限制连接点匹配AOP代理Bean引用为指定的类型
target	目标对象(即被代理对象)
@target()	限制目标对象的配置了指定的注解
within	限制连接点匹配执行的类型
@windin()	限定连接点带有匹配注解类型
@annotation	限定带有指定注解的连接点

在spring中尝试使用AspectJ其他指示器时，将会抛出IllegalArgumentException异常。

当我们查看上面展示的这些spring支持的指示器时，注意只有execution指示器是唯一的执行匹配，而其他的指示器都是用于限制匹配的。这说明execution指示器是我们在编写切点定义时最主要使用的指示器，在此基础上，我们使用其他指示器来限制所匹配的切点。

下图的切点表达式表示当Instrument的play方法执行时会触发通知。

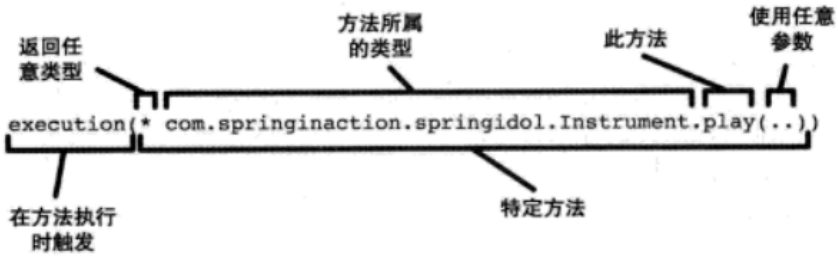


图 4.4　使用 AspectJ 切点表达式来定位

我们使用execution指示器选择Instrument的play方法，方法表达式以 `*` 号开始，标识我们不关心方法的返回值类型。然后我们指定了全限定类名和方法名。对于方法参数列表，我们使用 `..` 标识切点选择任意的play方法，无论该方法的入参是什么。
多个匹配之间我们可以使用链接符 `&&`、`||`、`!` 来表示 “且”、“或”、“非”的关系。但是在使用 XML 文件配置时，这些符号有特殊的含义，所以我们使用 “and”、“or”、“not”来表示。

举例：
限定该切点仅匹配的包是 `aspectJ.demo.entity`,可以使用
`execution(* aspectJ.demo.entity.IBuy.buy(..)) && within(aspectJ.demo.entity.*)`
在切点中选择 bean,可以使用
`execution(* aspectJ.demo.entity.IBuy.buy(..)) && within(aspectJ.demo.entity.*) && bean(girl)`
修改 BuyAspectJ.java

```
@Aspect
@Component
public class BuyAspectJ {

    @Before("execution(* aspectJ.demo.entity.IBuy.buy(..)) && within(aspectJ.demo.entity.*) && bean(girl)")
    public void haha(){
        System.out.println("男孩女孩都买自己喜欢的东西");
    }
}
```

此时，切面只会对 `Girl.java` 这个类生效，执行结果：

```
男孩买了一个游戏机
男孩女孩都买自己喜欢的东西
女孩买了一件漂亮的衣服
```

细心的你，可能发现了，切面中的方法名，已经被我悄悄地从 `haha` 改成了 `hehe`，丝毫没有影响结果，说明方法名没有影响。和 Spring IOC 中用 java 配置文件装配 Bean 时，用 `@Bean` 注解修饰的方法名一样，没有影响。

4.2 通过注解声明 5 种通知类型

Spring AOP 中有 5 中通知类型，分别如下：

注解	通知
@Before	通知方法会在目标方法调用之前执行
@After	通知方法会在目标方法返回或异常后调用
@AfterReturning	通知方法会在目标方法返回后调用
@AfterThrowing	通知方法会在目标方法抛出异常后调用
@Around	通知方法会将目标方法封装起来

修改切面类：

```
package aspectJ.demo.config;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class BuyAspectJ {

    @Before("execution(* aspectJ.demo.entity.IBuy.buy(..))")
    public void hehe(){
        System.out.println("before...");
    }

    @After("execution(* aspectJ.demo.entity.IBuy.buy(..))")
    public void haha() {
        System.out.println("After ...");
    }

    @AfterReturning("execution(* aspectJ.demo.entity.IBuy.buy(..))")
    public void xixi() {
        System.out.println("AfterReturning ...");
    }

    @Around("execution(* aspectJ.demo.entity.IBuy.buy(..))")
    public void xxx(ProceedingJoinPoint pj) {
        try {
            System.out.println("Around aaa ...");
            pj.proceed();
            System.out.println("Around bbb ...");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
    }
}
```

为了方便看效果,我们测试类中，只要 Boy 类:

执行结果如下：

```
Around aaa ...
before...
男孩买了一个游戏机
AfterReturning ...
After ...
Around bbb ...
```

结果显而易见。指的注意的是 `@Around` 修饰的环绕通知类型，是将整个目标方法封装起来了，在使用时，我们传入了 `ProceedingJoinPoint` 类型的参数，这个对象是必须要有的，并且需要调用 `ProceedingJoinPoint` 的 `proceed()` 方法。 如果没有调用 该方法，执行结果为 ：

```
Around aaa ...
Around bbb ...
```

4.3 通过注解声明切点表达式

如你看到的，上面我们写的多个通知使用了相同的切点表达式，对于像这样频繁出现的相同的表达式，我们可以使用 `@Pointcut` 注解声明切点表达式，然后使用表达式，修改代码如下：

```
package aspectJ.demo.config;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class BuyAspectJ {

    @Pointcut("execution(* aspectJ.demo.entity.IBuy.buy(..))")
    public void point() {

    }

    @Before("point()")
    public void hehe(){
        System.out.println("before...");
    }

    @After("point()")
    public void haha() {
        System.out.println("After ...");
    }

    @AfterReturning("point()")
    public void xixi() {
        System.out.println("AfterReturning ...");
    }

    @Around("point()")
    public void xxx(ProceedingJoinPoint pj) {
        try {
            System.out.println("Around aaa ...");
            pj.proceed();
            System.out.println("Around bbb ...");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
    }
}
```

4.4 通过注解处理通知中的参数

上面的例子，我们要进行增强处理的目标方法没有参数，下面我们来说说有参数的情况，并且在增强处理中使用该参数。
下面我们给接口增加一个参数，表示购买所花的金钱。通过AOP 增强处理，如果女孩买衣服超过了 68 元，就可以赠送一双袜子。
更改代码如下：

`IBuy.java`

```
package aspectJ.demo.entity;

public interface IBuy {
    String buy(double price);
}
```

`Girl.java`

```java
package aspectJ.demo.entity;

import org.springframework.stereotype.Component;

@Component
public class Girl implements IBuy {
    public String buy(double price) {
        System.out.printf("女孩买了一件漂亮的衣服,", price);
        return "衣服";
    }
}
```

BuyAspectJ.java

```java
package aspectJ.demo.config;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class BuyAspectJ {

    @Pointcut("execution(* aspectJ.demo.entity.IBuy.buy(..)) && args(price) && bean(girl)")
    public void point(double price) {

    }

    @Around("point(price)")
    public String hehe(ProceedingJoinPoint pj, double price){
        try {
            pj.proceed();
            if (price > 68) {
                System.out.println("女孩买衣服超过了68元，赠送一双袜子");
                return "衣服和袜子";
            }
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        return "衣服";
    }
}
```

测试结果：

> 男孩买了一个游戏机
> 女孩买了一件漂亮的衣服,女孩买衣服超过了68元，赠送一双袜子

4.5 通过注解配置织入的方式

前面还有一个遗留问题，在配置文件中，我们用注解 `@EnableAspectJAutoProxy()` 启用Spring AOP 的时候，我们给参数 `proxyTargetClass` 赋值为 `true`,如果我们不写参数，默认为 false。这个时候运行程序，程序抛出异常

```
Exception in thread "main" java.lang.ClassCastException Create breakpoint : com.sun.proxy.$Proxy18 cannot be cast to aspectJ.demo.entity.Girl
    at aspectJ.demo.App.main(App.java:13)
```

这是一个强制类型转换异常。为什么会抛出这个异常呢？或许已经能够想到，这跟Spring AOP 动态代理的机制有关，这个 `proxyTargetClass` 参数决定了代理的机制。当这个参数为 false 时，
通过jdk的基于接口的方式进行织入，这时候代理生成的是一个接口对象，将这个接口对象强制转换为实现该接口的一个类，自然就抛出了上述类型转换异常。
反之，`proxyTargetClass` 为 `true`，则会使用 cglib 的动态代理方式。这种方式的缺点是拓展类的方法被 `final` 修饰时，无法进行织入。
测试一下，我们将 `proxyTargetClass` 参数设为 `true`，同时将 Girl.java 的 Buy 方法用 `final` 修饰：

```
package aspectJ.demo.entity;

import org.springframework.stereotype.Component;

@Component
public class Girl implements IBuy {
    public final String buy(double price) {
        System.out.printf("女孩买了一件漂亮的衣服,", price);
        return "衣服";
    }
}
```

运行结果：

```
三月 10, 2021 3:11:43 下午 org.springframework.aop.framework.CglibAopProxy doValidateClass
信息: Unable to proxy interface-implementing method [public final java.lang.String aspectJ.demo.entity.Girl.buy(double)] because it is marked as final: Consider using interface-based JDK
男孩买了一个游戏机
女孩买了一件漂亮的衣服,
```

可以看到，我们的切面并没有织入生效。