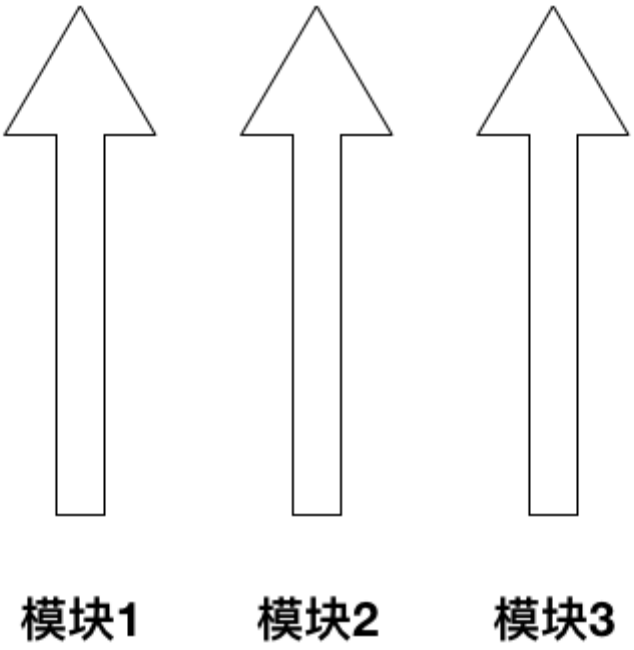
Spring AOP 切点表达式用法总结

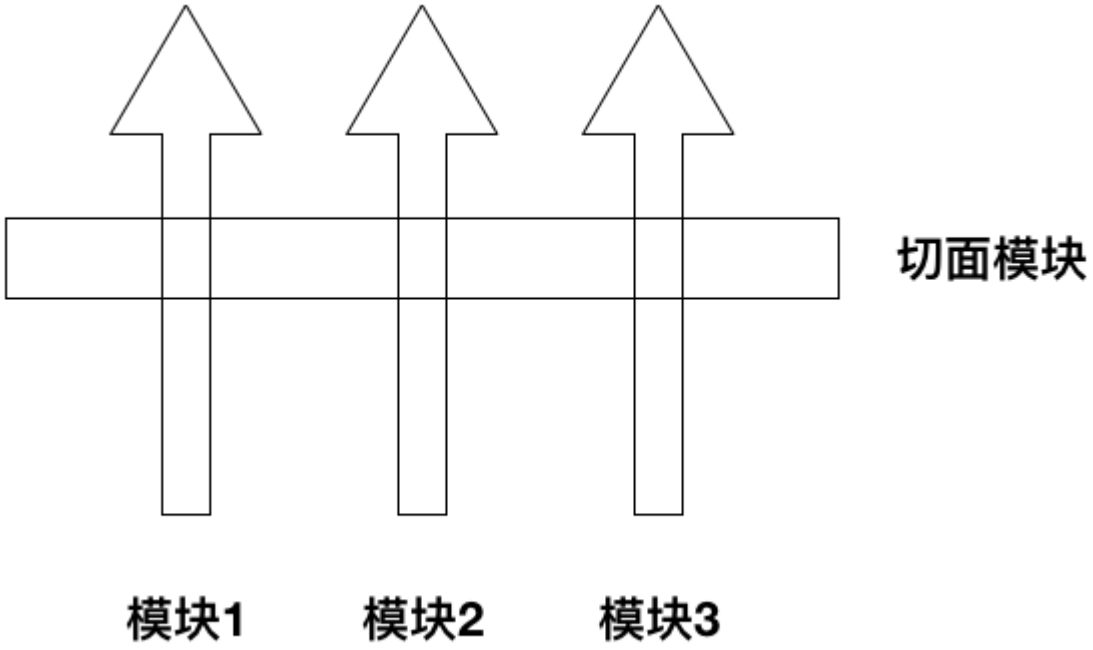
1. 简介

面向对象编程，也称为 OOP（即 Object Oriented Programming）最大的优点在于能够将业务模块进行封装，从而达到功能复用的目的。通过面向对象编程，不同的模板可以相互组装，从而实现更为复杂的业务模块，其结构形式可用下图表示：



面向对象编程解决了业务模块的封装复用的问题，但是对于某些模块，其本身并不独属于摸个业务模块，而是根据不同的情况，贯穿于某几个或全部的模块之间的。例如登录验证，其只开放几个可以不用登录的接口给用户使用（一般登录使用拦截器实现，但是其切面思想是一致的）；再比如性能统计，其需要记录每个业务模块的调用，并且监控器调用时间。可以看到，这些横贯于每个业务模块的模块，如果使用面向对象的方式，那么就需要在已封装的每个模块中添加相应的重复代码，对于这种情况，面向切面编程就可以派上用场了。

面向切面编程，也称为 AOP（即 Aspect Oriented Programming），指的是将一定的切面逻辑按照一定的方式编织到指定的业务模块中，从而将这些业务模块的调用包裹起来。如下是其结构示意图：



2. AOP 的各个扮演者

2.1 AOP 的主要角色

- 切面：使用切点表达式表示，指定了当前切面逻辑所要包裹的业务模块的范围大小；
- Advice：也即切面逻辑，指定了当前用于包裹切面指定的业务模块的逻辑。

2.2 Advice 的主要类型

- @Before：该注解标注的方法在业务模块代码执行之前执行，其不能阻止业务模块的执行，除非抛出异常；
- @AfterReturning：该注解标注的方法在业务模块代码执行之后执行；
- @AfterThrowing：该注解标注的方法在业务模块抛出指定异常后执行；
- @After：该注解标注的方法在所有的 Advice 执行完成后执行，无论业务模块是否抛出异常，类似于 finally 的作用；
- @Around：该注解功能最为强大，其所标注的方法用于编写包裹业务模块执行的代码，其可以传入一个 ProceedingJoinPoint 用于调用业务模块的代码，无论是调用前逻辑还是调用后逻辑，都可以在该方法中编写，甚至其可以根据一定的条件而阻断业务模块的调用；
- @DeclareParents：其是一种 Introduction 类型的模型，在属性声明上使用，主要用于为指定的业务模块添加新的接口和相应的实现。

- @Aspect：严格来说，其不属于一种 Advice，该注解主要用在类声明上，指明当前类是一个组织了切面逻辑的类，并且该注解中可以指定当前类是何种实例化方式，主要有三种：singleton、perthis 和 pertarget，具体的使用方式后面会进行讲解。

这里需要说明的是，@Before 是业务逻辑执行前执行，与其对应的是@AfterReturning，而不是@After，@After 是所有的切面逻辑执行完之后才会执行，无论是否抛出异常。

3. 切点表达式

3.1 execution

由于 Spring 切面粒度最小是达到方法级别，而 execution 表达式可以用于明确指定方法返回类型，类名，方法名和参数名等与方法相关的部件，并且在 Spring 中，大部分需要使用 AOP 的业务场景也只需要达到方法级别即可，因而 execution 表达式的使用是最为广泛的。如下是 execution 表达式的语法：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)
```

这里问号表示当前项可以有也可以没有，其中各项的语义如下：

- modifiers-pattern：方法的可见性，如 public，protected；
- ret-type-pattern：方法的返回值类型，如 int，void 等；
- declaring-type-pattern：方法所在类的全路径名，如 com.spring.Aspect；
- name-pattern：方法名类型，如 buisinessService()；
- param-pattern：方法的参数类型，如 java.lang.String；
- throws-pattern：方法抛出的异常类型，如 java.lang.Exception；

如下是一个使用 execution 表达式的例子：

```
execution(public * com.spring.service.BusinessObject.businessService(java.lang.String,..))
```

上述切点表达式将会匹配使用 public 修饰，返回值为任意类型，并且是 com.spring.BusinessObject 类中名称为 businessService 的方法，方法可以有多个参数，但是第一个参数必须是 java.lang.String 类型的方法。上述示例中我们使用了..通配符，关于通配符的类型，主要有两种：

- *通配符，该通配符主要用于匹配单个单词，或者是以某个词为前缀或后缀的单词。

如下示例表示返回值为任意类型，在 com.spring.service.BusinessObject 类中，并且参数个数为零的方法：

```
execution(* com.spring.service.BusinessObject.*())
```

下述示例表示返回值为任意类型，在 com.spring.service 包中，以 Business 为前缀的类，并且是类中参数个数为零方法：

```
execution(* com.spring.service.Business*.*())
```

- ..通配符，该通配符表示 0 个或多个项，主要用于 declaring-type-pattern 和 param-pattern 中，如果用于 declaring-type-pattern 中，则表示匹配当前包及其子包，如果用于 param-pattern 中，则表示匹配 0 个或多个参数。

如下示例表示匹配返回值为任意类型，并且是 com.spring.service 包及其子包下的任意类的名称为 businessService 的方法，而且该方法不能有任何参数：

```
execution(* com.spring.service..*.businessService())
```

这里需要说明的是，包路径 service..*.businessService()中的..应该理解为延续前面的 service 路径，表示到 service 路径为止，或者继续延续 service 路径，从而包括其子包路径；后面的*.businessService()，这里的*表示匹配一个单词，因为是在方法名前，因而表示匹配任意的类。

如下示例是使用..表示任意个数的参数的示例，需要注意，表示参数的时候可以在括号中事先指定某些类型的参数，而其余的参数则由..进行匹配：

```
execution(* com.spring.service.BusinessObject.businessService(java.lang.String,..))
```

3.2 within

within 表达式的粒度为类，其参数为全路径的类名（可使用通配符），表示匹配当前表达式的所有类都将被当前方法环绕。如下是 within 表达式的语法：

```
within(declaring-type-pattern)
```

within 表达式只能指定到类级别，如下示例表示匹配 com.spring.service.BusinessObject 中的所有方法：

```
within(com.spring.service.BusinessObject)
```

within 表达式路径和类名都可以使用通配符进行匹配，比如如下表达式将匹配 com.spring.service 包下的所有类，不包括子包中的类：

```
within(com.spring.service.*)
```

如下表达式表示匹配 com.spring.service 包及子包下的所有类：

```
within(com.spring.service..*)
```

3.3 args

args 表达式的作用是匹配指定参数类型和指定参数数量的方法，无论其类路径或者是方法名是什么。这里需要注意的是，args 指定的参数必须是全路径的。如下是 args 表达式的语法：

```
args(param-pattern)
```

如下示例表示匹配所有只有一个参数，并且参数类型是 java.lang.String 类型的方法：

```
args(java.lang.String)
```

也可以使用通配符，但这里通配符只能使用..，而不能使用*。如下是使用通配符的实例，该切点表达式将匹配第一个参数为 java.lang.String，最后一个参数为 java.lang.Integer，并且中间可以有任意个数和类型参数的方法：

```
args(java.lang.String,..,java.lang.Integer)
```

3.4 this 和 target

this 和 target 需要放在一起进行讲解，主要目的是对其进行区别。this 和 target 表达式中都只能指定类或者接口，在面向切面编程规范中，this 表示匹配调用当前切点表达式所指代对象方法的对象，target 表示匹配切点表达式指定类型的对象。比如有两个类 A 和 B，并且 A 调用了 B 的某个方法，如果切点表达式为 this(B)，那么 A 的实例将会被匹配，也即其会被使用当前切点表达式的 Advice 环绕；如果这里切点表达式为 target(B)，那么 B 的实例也即被匹配，其将会被使用当前切点表达式的 Advice 环绕。

在讲解 Spring 中的 this 和 target 的使用之前，首先需要讲解一个概念：业务对象（目标对象）和代理对象。对于切面编程，有一个目标对象，也有一个代理对象，目标对象是我们声明的业务逻辑对象，而代理对象是使用切面逻辑对业务逻辑进行包裹之后生成的对象。如果使用的是 Jdk 动态代理，那么业务对象和代理对象将是两个对象，在调用代理对象逻辑时，其切面逻辑中会调用目标对象的逻辑；如果使用的是 Cglib 代理，由于是使用的子类进行切面逻辑织入的，那么只有一个对象，即织入了代理逻辑的业务类的子类对象，此时是不会生成业务类的对象的。

在 Spring 中，其对 this 的语义进行了改写，即如果当前对象生成的代理对象符合 this 指定的类型，那么就为其织入切面逻辑。简单的说就是，this 将匹配代理对象为指定类型的类。target 的语义则没有发生变化，即其将匹配业务对象为指定类型的类。如下是使用 this 和 target 表达式的简单示例：

```
this(com.spring.service.BusinessObject)
```

```
target(com.spring.service.BusinessObject)
```

通过上面的讲解可以看出，this 和 target 的使用区别其实不大，大部分情况下其使用效果是一样的，但其区别也还是有的。Spring 使用的代理方式主要有两种：Jdk 代理和 Cglib 代理（关于这两种代理方式的讲解可以查看本人的文章代理模式实现方式及优缺点对比）。针对这两种代理类型，关于目标对象与代理对象，理解如下两点是非常重要的：

- 如果目标对象被代理的方法是其实现的某个接口的方法，那么将会使用 Jdk 代理生成代理对象，此时代理对象和目标对象是两个对象，并且都实现了该接口；
- 如果目标对象是一个类，并且其没有实现任意接口，那么将会使用 Cglib 代理生成代理对象，并且只会生成一个对象，即 Cglib 生成的代理类的对象。

结合上述两点说明，这里理解 this 和 target 的异同就相对比较简单了。我们这里分三种情况进行说明：

- this(SomeInterface)或 target(SomeInterface)：这种情况下，无论是对于 Jdk 代理还是 Cglib 代理，其目标对象和代理对象都是实现 SomeInterface 接口的（Cglib 生成的目标对象的子类也是实现了 SomeInterface 接口的），因而 this 和 target 语义都是符合的，此时这两个表达式的效果一样；

- this(SomeObject)或 target(SomeObject)，这里 SomeObject 没实现任何接口：这种情况下，Spring 会使用 Cglib 代理生成 SomeObject 的代理类对象，由于代理类是 SomeObject 的子类，子类的对象也是符合 SomeObject 类型的，因而 this 将会被匹配，而对于 target，由于目标对象本身就是 SomeObject 类型，因而这两个表达式的效果一样；

- this(SomeObject)或 target(SomeObject)，这里 SomeObject 实现了某个接口：对于这种情况，虽然表达式中指定的是一种具体的对象类型，但由于其实现了某个接口，因而 Spring 默认会使用 Jdk 代理为其生成代理对象，Jdk 代理生成的代理对象与目标对象实现的是同一个接口，但代理对象与目标对象还是不同的对象，由于代理对象不是 SomeObject 类型的，因而此时是不符合 this 语义的，而由于目标对象就是 SomeObject 类型，因而 target 语义是符合的，此时 this 和 target 的效果就产生了区别；这里如果强制 Spring 使用 Cglib 代理，因而生成的代理对象都是 SomeObject 子类的对象，其是 SomeObject 类型的，因而 this 和 target 的语义都符合，其效果就是一致的。

关于 this 和 target 的异同，我们使用如下示例进行简单演示：

```
// 目标类
public class Apple {
  public void eat() {
    System.out.println("Apple.eat method invoked.");
  }
}
// 切面类
@Aspect
public class MyAspect {
  @Around("this(com.business.Apple)")
  public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("this is before around advice");
    Object result = pjp.proceed();
    System.out.println("this is after around advice");
```

```
    return result;
  }
}
<!-- bean 声明文件 -->
<bean id="apple" class="chapter7.eg1.Apple"/>
<bean id="aspect" class="chapter7.eg6.MyAspect"/>
<aop:aspectj-autoproxy/>
// 驱动类
public class AspectApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("application
Context.xml");
    Apple fruit = (Apple) context.getBean("apple");
    fruit.eat();
  }
}
```

执行驱动类中的 main 方法，结果如下：

```
this is before around advice
Apple.eat method invoked.
this is after around advice
```

上述示例中，Apple 没有实现任何接口，因而使用的是 Cglib 代理，this 表达式会匹配 Apple 对象。这里将切点表达式更改为 target，还是执行上述代码，会发现结果还是一样的：

```
target(com.business.Apple)
```

如果我们对 Apple 的声明进行修改，使其实现一个接口，那么这里就会显示出 this 和 target 的执行区别了：

```
public class Apple implements IApple {
  public void eat() {
    System.out.println("Apple.eat method invoked.");
  }
}
public class AspectApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("application
Context.xml");
    Fruit fruit = (Fruit) context.getBean("apple");
    fruit.eat();
  }
}
```

我们还是执行上述代码，对于 this 表达式，其执行结果如下：

```
Apple.eat method invoked.
```

对于 target 表达式，其执行结果如下：

```
this is before around advice
Apple.eat method invoked.
this is after around advice
```

可以看到，这种情况下 this 和 target 表达式的执行结果是不一样的，这正好符合我们前面讲解的第三种情况。

3.5 @within

前面我们讲解了 within 的语义表示匹配指定类型的类实例，这里的@within 表示匹配带有指定注解的类，其使用语法如下所示：

```
@within(annotation-type)
```

如下所示示例表示匹配使用 com.spring.annotation.BusinessAspect 注解标注的类：

@within(com.spring.annotation.BusinessAspect)

这里我们使用一个例子演示@within 的用法（这里驱动类和 xml 文件配置与 3.4 节使用的一致，这里省略）：

```
// 注解类
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface FruitAspect {
}
// 目标类
@FruitAspect
public class Apple {
  public void eat() {
    System.out.println("Apple.eat method invoked.");
  }
}
// 切面类
@Aspect
public class MyAspect {
  @Around("@within(com.business.annotation.FruitAspect)")
  public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("this is before around advice");
    Object result = pjp.proceed();
    System.out.println("this is after around advice");
    return result;
```

```
  }

}
```

上述切面表示匹配使用 FruitAspect 注解的类，而 Apple 则使用了该注解，因而 Apple 类方法的调用会被切面环绕，执行运行驱动类可得到如下结果，说明 Apple.eat()方法确实被环绕了：

```
this is before around advice

Apple.eat method invoked.

this is after around advice
```

3.6 @annotation

@annotation 的使用方式与@within 的相似，表示匹配使用@annotation 指定注解标注的方法将会被环绕，其使用语法如下：

```
@annotation(annotation-type)
```

如下示例表示匹配使用 com.spring.annotation.BusinessAspect 注解标注的方法：

```
@annotation(com.spring.annotation.BusinessAspect)
```

这里我们继续复用 3.5 节使用的例子进行讲解@annotation 的用法，只是这里需要对 Apple 和 MyAspect 使用和指定注解的方式进行修改，FruitAspect 不用修改的原因是声明该注解时已经指定了其可以使用在类，方法和参数上：

```
// 目标类，将 FruitAspect 移到了方法上

public class Apple {

  @FruitAspect

  public void eat() {

    System.out.println("Apple.eat method invoked.");

  }

}

@Aspect

public class MyAspect {

  @Around("@annotation(com.business.annotation.FruitAspect)")
```

```
  public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("this is before around advice");
    Object result = pjp.proceed();
    System.out.println("this is after around advice");
    return result;
  }
}
```

这里 Apple.eat()方法使用 FruitAspect 注解进行了标注，因而该方法的执行会被切面环绕，其执行结果如下：

```
this is before around advice
Apple.eat method invoked.
this is after around advice
```

3.7 @args

@within 和@annotation 分别表示匹配使用指定注解标注的类和标注的方法将会被匹配，@args 则表示使用指定注解标注的类作为某个方法的参数时该方法将会被匹配。如下是@args 注解的语法：

```
@args(annotation-type)
```

如下示例表示匹配使用了 com.spring.annotation.FruitAspect 注解标注的类作为参数的方法：

```
@args(com.spring.annotation.FruitAspect)
```

这里我们使用如下示例对@args 的用法进行讲解：

```
<!-- xml 配置文件 -->
<bean id="bucket" class="chapter7.eg1.FruitBucket"/>
<bean id="aspect" class="chapter7.eg6.MyAspect"/>
<aop:aspectj-autoproxy/>
// 使用注解标注的参数类
@FruitAspect
```

```
public class Apple {}

// 使用 Apple 参数的目标类

public class FruitBucket {

  public void putIntoBucket(Apple apple) {

    System.out.println("put apple into bucket.");

  }

}

@Aspect

public class MyAspect {

  @Around("@args(chapter7.eg6.FruitAspect)")

  public Object around(ProceedingJoinPoint pjp) throws Throwable {

    System.out.println("this is before around advice");

    Object result = pjp.proceed();

    System.out.println("this is after around advice");

    return result;

  }

}

// 驱动类

public class AspectApp {

  public static void main(String[] args) {

    ApplicationContext context = new ClassPathXmlApplicationContext("application
Context.xml");

    FruitBucket bucket = (FruitBucket) context.getBean("bucket");

    bucket.putIntoBucket(new Apple());
```

```
  }

}
```

这里 FruitBucket.putIntoBucket(Apple)方法的参数 Apple 使用了@args 注解指定的 FruitAspect 进行了标注，因而该方法的调用将会被环绕。执行驱动类，结果如下：

```
this is before around advice

put apple into bucket.

this is after around advice
```

3.8 @DeclareParents

@DeclareParents 也称为 Introduction（引入），表示为指定的目标类引入新的属性和方法。关于@DeclareParents 的原理其实比较好理解，因为无论是 Jdk 代理还是 Cglib 代理，想要引入新的方法，只需要通过一定的方式将新声明的方法织入到代理类中即可，因为代理类都是新生成的类，因而织入过程也比较方便。如下是@DeclareParents 的使用语法：

```
@DeclareParents(value = "TargetType", defaultImpl = WeaverType.class)

private WeaverInterface attribute;
```

这里 TargetType 表示要织入的目标类型（带全路径），WeaverInterface 中声明了要添加的方法，WeaverType 中声明了要织入的方法的具体实现。如下示例表示在 Apple 类中织入 IDescriber 接口声明的方法：

```
@DeclareParents(value = "com.spring.service.Apple", defaultImpl = DescriberImpl.
class)

private IDescriber describer;
```

这里我们使用一个如下实例对@DeclareParents 的使用方式进行讲解，配置文件与 3.4 节的一致，这里略：

```
// 织入方法的目标类

public class Apple {

  public void eat() {

    System.out.println("Apple.eat method invoked.");

  }

}
```

```
// 要织入的接口
public interface IDescriber {
  void desc();
}
// 要织入接口的默认实现
public class DescriberImpl implements IDescriber {
  @Override
  public void desc() {
    System.out.println("this is an introduction describer.");
  }
}
// 切面实例
@Aspect
public class MyAspect {
  @DeclareParents(value = "com.spring.service.Apple", defaultImpl = DescriberImp
l.class)
  private IDescriber describer;
}
// 驱动类
public class AspectApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("application
Context.xml");
    IDescriber describer = (IDescriber) context.getBean("apple");
    describer.desc();
```

```
    }

}
```

在 MyAspect 中声明了我们需要将 IDescriber 的方法织入到 Apple 实例中，在驱动类中我们可以看到，我们获取的是 apple 实例，但是得到的 bean 却可以强转为 IDescriber 类型，因而说明我们的织入操作成功了。

3.9 perthis 和 pertarget

在 Spring AOP 中，切面类的实例只有一个，比如前面我们一直使用的 MyAspect 类，假设我们使用的切面类需要具有某种状态，以适用某些特殊情况的使用，比如多线程环境，此时单例的切面类就不符合我们的要求了。在 Spring AOP 中，切面类默认都是单例的，但其还支持另外两种多例的切面实例的切面，即 perthis 和 pertarget，需要注意的是 perthis 和 pertarget 都是使用在切面类的@Aspect 注解中的。这里 perthis 和 pertarget 表达式中都是指定一个切面表达式，其语义与前面讲解的 this 和 target 非常的相似，perthis 表示如果某个类的代理类符合其指定的切面表达式，那么就会为每个符合条件的目标类都声明一个切面实例；pertarget 表示如果某个目标类符合其指定的切面表达式，那么就会为每个符合条件的类声明一个切面实例。从上面的语义可以看出，perthis 和 pertarget 的含义是非常相似的。如下是 perthis 和 pertarget 的使用语法：

```
perthis(pointcut-expression)

pertarget(pointcut-expression)
```

由于 perthis 和 pertarget 的使用效果大部分情况下都是一致的，我们这里主要讲解 perthis 和 pertarget 的区别。关于 perthis 和 pertarget 的使用，需要注意的一个点是，由于 perthis 和 pertarget 都是为每个符合条件的类声明一个切面实例，因而切面类在配置文件中的声明上一定要加上 prototype，否则 Spring 启动是会报错的。如下是我们使用的示例：

```
<!-- xml 配置文件 -->

<bean id="apple" class="chapter7.eg1.Apple"/>

<bean id="aspect" class="chapter7.eg6.MyAspect" scope="prototype"/>

<aop:aspectj-autoproxy/>

// 目标类实现的接口

public interface Fruit {

  void eat();

}

// 业务类
```

```
public class Apple implements Fruit {
  public void eat() {
    System.out.println("Apple.eat method invoked.");
  }
}
// 切面类
@Aspect("perthis(this(com.spring.service.Apple))")
public class MyAspect {

  public MyAspect() {
    System.out.println("create MyAspect instance, address: " + toString());
  }

  @Around("this(com.spring.service.Apple)")
  public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("this is before around advice");
    Object result = pjp.proceed();
    System.out.println("this is after around advice");
    return result;
  }
}
// 驱动类
public class AspectApp {
  public static void main(String[] args) {
```

```
    ApplicationContext context = new ClassPathXmlApplicationContext("application
Context.xml");

    Fruit fruit = context.getBean(Fruit.class);

    fruit.eat();

  }

}
```

这里我们使用的切面表达式语法为 perthis(this(com.spring.service.Apple))，这里 this 表示匹配代理类是 Apple 类型的类，perthis 则表示会为这些类的每个实例都创建一个切面类。由于 Apple 实现了 Fruit 接口，因而 Spring 使用 Jdk 动态代理为其生成代理类，也就是说代理类与 Apple 都实现了 Fruit 接口，但是代理类不是 Apple 类型，因而这里声明的切面不会匹配到 Apple 类。执行上述驱动类，结果如下：

```
Apple.eat method invoked.
```

结果表明 Apple 类确实没有被环绕。如果我们讲切面类中的 perthis 和 this 修改为 pertarget 和 target，效果如何呢：

```
@Aspect("pertarget(target(com.spring.service.Apple))")
public class MyAspect {


  public MyAspect() {
    System.out.println("create MyAspect instance, address: " + toString());
  }


  @Around("target(com.spring.service.Apple)")
  public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("this is before around advice");
    Object result = pjp.proceed();
    System.out.println("this is after around advice");
    return result;
```

```
  }
}
```

执行结果如下：

```
create MyAspect instance, address: chapter7.eg6.MyAspect@48fa0f47
this is before around advice
Apple.eat method invoked.
this is after around advice
```

可以看到，Apple 类被切面环绕了。这里 target 表示目标类是 Apple 类型，虽然 Spring 使用了 Jdk 动态代理实现切面的环绕，代理类虽不是 Apple 类型，但是目标类却是 Apple 类型，符合 target 的语义，而 pertarget 会为每个符合条件的表达式的类实例创建一个代理类实例，因而这里 Apple 会被环绕。

由于代理类与目标类的差别非常小，因而与 this 和 target 一样，perthis 和 pertarget 的区别也非常小，大部分情况下其使用效果是一致的。关于切面多实例的创建，其演示比较简单，我们可以将 xml 文件中的 Apple 实例修改为 prototype 类型，并且在驱动类中多次获取 Apple 类的实例：

```
<!-- xml 配置文件 -->
<bean id="apple" class="chapter7.eg1.Apple" scope="prototype"/>
<bean id="aspect" class="chapter7.eg6.MyAspect" scope="prototype"/>
<aop:aspectj-autoproxy/>
public class AspectApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("application
Context.xml");
    Fruit fruit = context.getBean(Fruit.class);
    fruit.eat();
    fruit = context.getBean(Fruit.class);
    fruit.eat();
  }
```

```
}
```

执行结果如下：

```
create MyAspect instance, address: chapter7.eg6.MyAspect@48fa0f47

this is before around advice

Apple.eat method invoked.

this is after around advice

create MyAspect instance, address: chapter7.eg6.MyAspect@56528192

this is before around advice

Apple.eat method invoked.

this is after around advice
```

执行结果中两次打印的 create MyAspect instance 表示当前切面实例创建了两次，这也符合我们进行的两次获取 Apple 实例。

4. 小结

本文首先对 AOP 进行了简单介绍，然后介绍了切面中的各个角色，最后详细介绍了切点表达式中各个不同类型表达式的语法。