1. AOP面向切面编程

1.1 AOP介绍

OOP(Object Oriented Programming) 面向对象编程，万物皆对象!

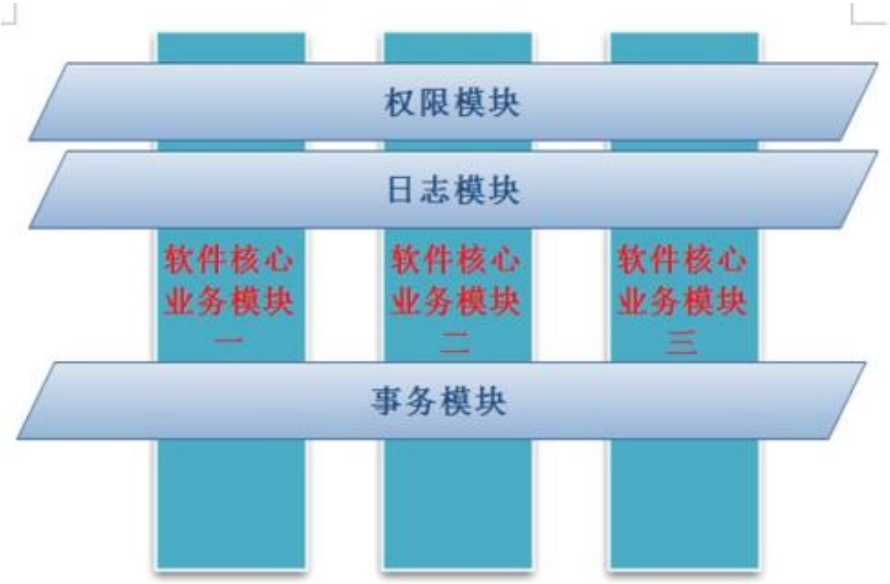
AOP（Aspect Oriented Programming），即**面向切面编程**，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。

OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义**纵向**的关系，但并不适合定义**横向**的关系，例如**日志功能**。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系；对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

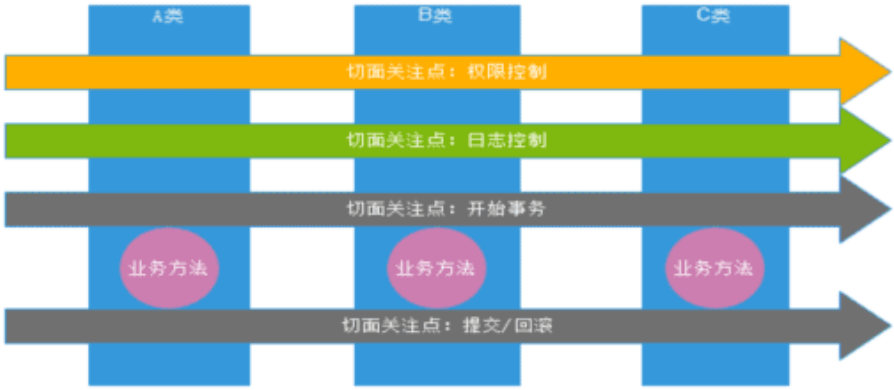
AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"\"，即**切面**。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

> *虽然使用OOP可以通过组合或继承的方式来达到代码的重用，但是如果要实现某个功能（如：日志记录等），同样的代码仍然可会分散到各个方法中。这样如果想要关闭某个功能，或者对其进行修改，就必须要修改所有的相关方法，这不但增加了开发人员的工作量，而且提高了代码的出错率；*
>
> *为了解决这一问题，AOP思想随之产生。AOP采取横向抽取机制，将分散在各个方法中的额重复代码抽取出来，然后再程序编程或运行时，再将这些提取出来的代码引用到需要执行的地方。这种采用横向抽取的方式，采用传统的OOP思想显然是无法办到的，因为OOP只能实现父子关系的纵向宠用。虽然AOP是一种新的编程思想，但却不是OOP的替代品，它只是OOP的延申和补充。*
>
> *AOP的使用使开发人员在编写业务逻辑时可以专心于核心业务，而不必关注于其他业务逻，这不但提高了开发效率而且增强了代码的可维护性。*

使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志打印、事务处理。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

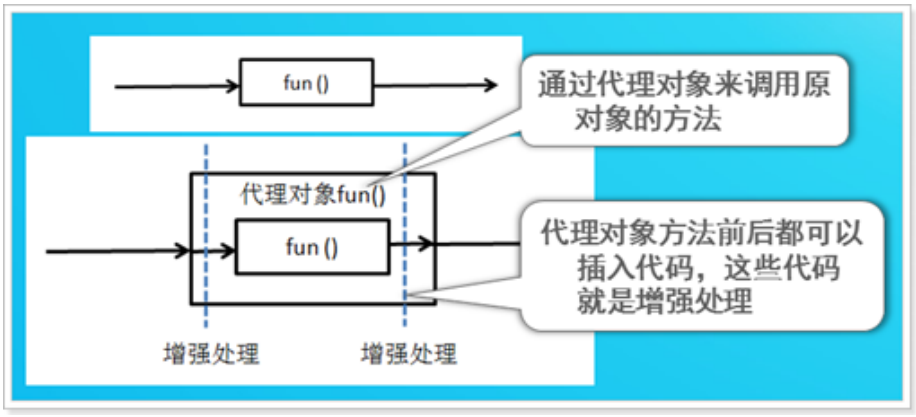


软件纵向与横向结构

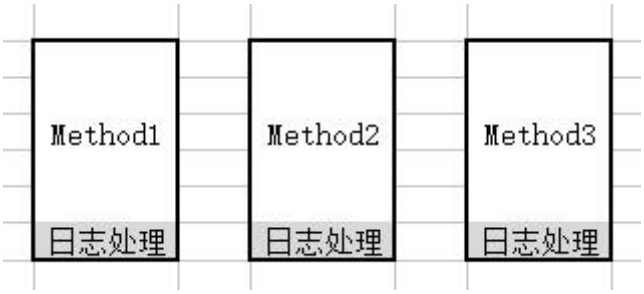


1.2 AOP图解

AOP编程底层代理设计模式！Spring框架底层使用的代理设计模式来完成AOP！

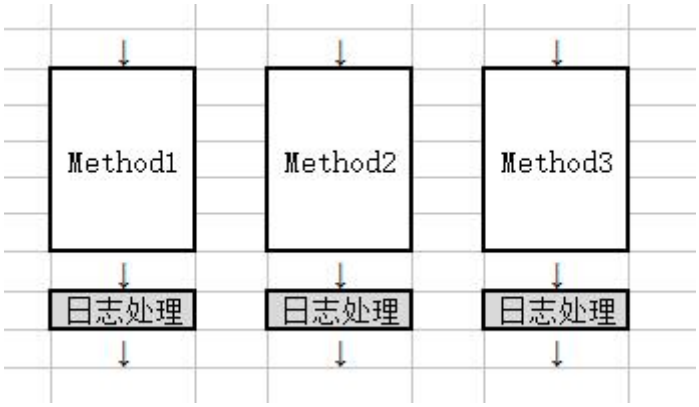


假如没有aop，在做日志处理的时候，我们会在每个方法中添加日志处理，比如



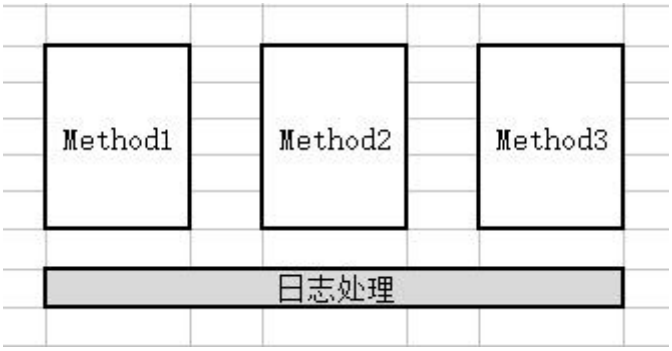
单一职责！

但大多数的日子处理代码是相同的，为了实现代码复用，我们可能把日志处理抽离成一个新的方法。但是这样我们仍然必须手动插入这些方法。



但这样两个方法就是强耦合的，假如此时我们不需要这个功能了，或者想换成其他功能，那么就必须一个个修改。

通过**动态代理**，可以在指定位置执行对应流程。这样就可以将一些横向的功能抽离出来形成一个独立的模块，然后在指定位置插入这些功能。这样的思想，被称为**面向切面编程**，亦即AOP。



1.3 Spring中代理的使用

具体内容可以去看《Spring框架专题(五)-Spring框架之Proxydaili.md》中的6.4节内

2. AOP框架

2.1 AOP框架

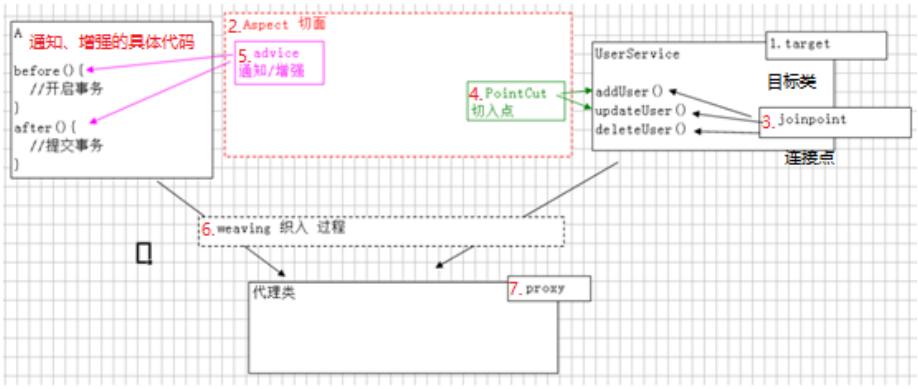
目前流行的AOP框架有两个：Spring AOP、AspectJ。

1. Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期间通过代理方式向目标类织入增强的代码。
2. AspectJ是一个基于Java语言的AOP框架（第3节中介绍的就是AspectJ的使用），从Spring2.0开始，Spring AOP引入了AspectJ的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码织入。

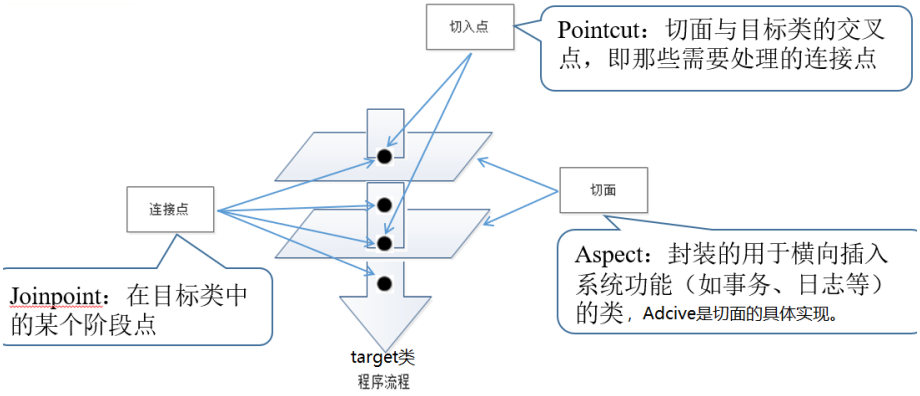
2.2 AOP术语

1. target：**目标类**，需要被代理的类。例如：UserServiceImpl（这个类是代理中的一个类）
2. Aspect(切面): 是切入点pointcut和通知advice的结合；在实际使用中，切面通常是指封装的用于横向插入系统功能（如：事务、日志等）的类，这种类被Spring容器识别为切面，需要在配置文件中通过bean元素指定。
3. Joinpoint(连接点)：所谓连接点是指程序执行过程中（target中）可能被处理的方法。例如：所有的方法
4. PointCut(切入点)：**target中被切面做增强操作的连接点**，例如：UserServiceImpl类中的addUser()方法。是指切面与程序流程（target类）的交叉点，即那些需要处理的连接点
5. advice 通知/增强，增强代码。例如：after、before。AOP框架在特定的切入点执行的增强处理，即在定义好的切入点处所要执行的程序代码。可以将其理解为切面类中的方法，它是切面的具体实现。
6. Weaving(织入)：是指把增强advice（切面代码）应用到目标对象target来创建新的代理对象proxy的过程.
7. proxy 代理类：通知+切入点(由动态代理自动生成的类)
8. 引介（Introduction）
   引介是一种特殊的增强（是通知的一种），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实现类。

具体可以根据下面这张图来理解：



教材中的一张图：



- Proxy（代理）：将通知（advice）应用到目标对象之后，被动态创建的对象。
- Weaving（织入）：将切面代码插入到目标对象上，从而生成代理对象的过程。

2.3 Spring的通知类型

Spring AOP中的通知按照在目标类方法的连接点位置，可以分为以下5种类型：

1. **org.springframework.aop.MethodBeforeAdvice**（前置通知）
   在目标方法执行前实施增强，可以应用于权限管理等功能。
2. **org.springframework.aop.AfterReturningAdvice**（后置通知）
   在目标方法执行后实施增强，可以应用于关闭流、上传文件、删除临时文件等功能。
3. **org.aopalliance.intercept.MethodInterceptor**（环绕通知）
   在目标方法执行前后实施增强，可以应用于日志、事务管理等功能。
4. **org.springframework.aop.ThrowsAdvice**（异常抛出通知）
   在方法抛出异常后实施增强，可以应用于处理异常记录日志等功能。
5. **org.springframework.aop.IntroductionInterceptor**（引介通知）
   在目标类中添加一些新的方法和属性，可以应用于修改老版本程序。

AspectJ中的通知类型：

1. 前置通知：在目标方法运行之前实施增强
2. 后置通知：在目标方法成功返回后实施增强
3. 环绕通知：在目标方法之前和之后实施增强
4. 异常通知：用于处理程序发生异常，如果程序没有异常，将不会执行增强

5. 最终通知：无论目标方法发生任何事情（无论方法是否成功执行），都会实施增强

最终通知和后置通知都是在方法执行之后实施增强

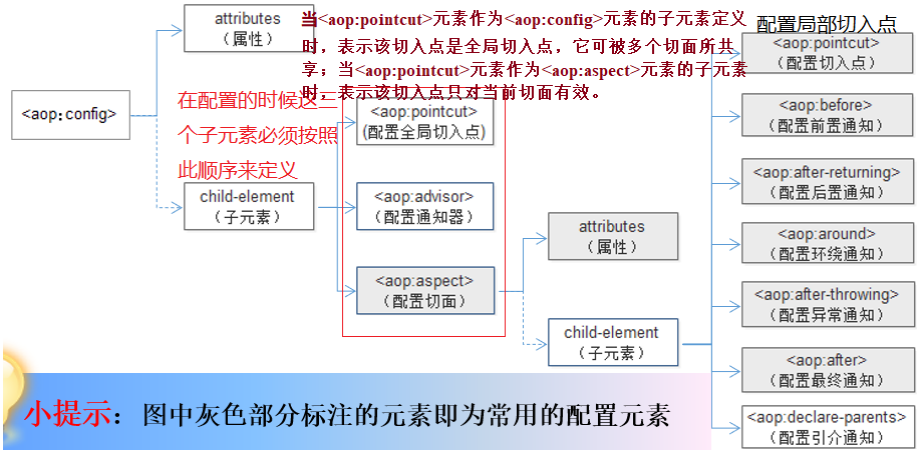
如果再同一个连接带你有多个通知需要执行，那么在同一个切面 中，目标反复噶之前的前置通知和环绕通知的执行顺序时位置的，目标方法之后的后置通知和环绕通知的执行顺序也是未知的。

3. AOP框架的AspectJ

AspectJ时一个基于Java语言的AOP框架，它提供了强大的AOP功能。Spring 2.0之后，Spring AOP引入了AspectJ的支持，并允许直接使用AspectJ进行编程，而Spring自身的AOP API也尽量与AspectJ保持一致。新版本的Spring框架，也建议使用AspectJ来开发AOP。使用AspectJ实现AOP有两种方式：一种是基于XML的声明式AspectJ，另一种是基于注解的声明式AspectJ。

3.1 AspectJ基于xml

配置文件中>元素及其子元素结构：



XML文件中常用的元素的配置方式如下：

1. 配置切面

配置<aop:aspect>元素时，通常会指定id和ref两个属性。

属性名称	描述
id	用于定义该切面的唯一标识名称
ref	用于引用普通的 Spring Bean

2. 配置切入点表达式

在定义<aop:pointcut>元素时，通常会指定id和expression两个属性。

属性名称	描述
id	用于指定切入点的唯一标识名称
expression	用于指定切入点关联的切入点表达式

切入点表达式：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)
/**
modifiers-pattern?：表示定义的目标方法的范围跟修饰符，如public、private等
ret-type-pattern：表示定义的目标方法的返回值类型，如void、String等；可以使用*表示任意类型
declaring-type-pattern?：表示定义的目标方法的类路径；再这个路径中可以使用；可以在类路径中使用*代表所有类
name-pattern：表示具体需要被代理的目标方法（方法名和类的全限定名之间没有空格，使用"."分割）；可以使用*代表所有方法
param-pattern：表示需要被dialing的目标方法包含的参数；在括号内可以使用..代表任意参数
throws-pattern?：表示需要被dialing的目标方法抛出的异常类型
*/
```

3. 配置通知

使用>的子元素可以配置5种常用通知，这5个子元素不支持使用子元素，但在使用时可以指定一些属性，其常用属性及其描述如下：

属性名称	描述
pointcut	该属性用于指定一个切入点表达式，Spring 将在匹配该表达式的连接点时织入该通知。
pointcut-ref	该属性指定一个已经存在的切入点名称，如配置代码中的 myPointCut。通常 pointcut 和 pointcut-ref 两个属性只需要使用其中之一。
method	该属性指定一个方法名，指定将切面 Bean 中的该方法转换为增强处理。
throwing	该属性只对<after-throwing>元素有效，它用于指定一个形参名，异常通知方法可以通过该形参访问目标方法所抛出的异常。
returning	该属性只对<after-returning>元素有效，它用于指定一个形参名，后置通知方法可以通过该形参访问目标方法的返回值。

上面的表达式中：带有问号（？）的部分（修饰符、类路径、异常）表示可配置项，而其他部分属于必须配置项。

3.1.1 项目准备

- 项目名: spring-03-aop

- 导入jar包:

- pom文件添加

```
<dependencies>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-
support</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-
expression</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <!--面向切面编程需要用到的包-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.2.8.RELEASE</version>
    </dependency>

    <!--LomBok插件-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
        <scope>provided</scope>
    </dependency>

</dependencies>
```

3.1.2 准备操作对象

- 先创建UserService接口：

```
public interface UserService {

    // 添加 user
    public void addUser(User user);

    // 删除 user
    public void deleteUser(int uid);
}
```

- 实现类（目标类，target）

```
public class UserServiceImpl implements
UserService {

    public void addUser(User user) {
        System.out.println("增加 User");
    }

    public void deleteUser(int uid) {
        System.out.println("删除 User");
    }
}
```

3.1.3 增强类（切面+advice）

注意：

1. 在切面的Advice（这里说的是before、after.....方法）的参数中可以使用JoinPoint类型的参数来获得目标类中目标方法的信息，可以根据这个参数获得目标类中目标方法的参数、签名.....；
2. 异常通知的参数中使用Throwable类型的参数，在增强方法（通知）中获得异常，注意这个参数的名字要和XML中配置的>标签中的属性throwing的值一致
3. 环绕通知必须接受一个类型为ProceedingJoinPoint类型的参，返回值必须是Object类型，且必须抛出异常
4. 后置通知中可以使用在XML中配置的>标签中的属性returning的值作为参数的名称，在后置通知中可以使用这个参数获得目标方法的返回值

```
public class MyAdivce {

    /**
    //前置通知：目标方法运行之前调用
    //后置通知(如果出现异常不会调用)：在目标方法运行之后调用
    //环绕通知：在目标方法之前和之后都调用
    //异常拦截通知：如果出现异常，就会调用
    //后置通知(无论是否出现 异常都会调用)：在目标方法运行之后调用
    */
```

```
    // 前置通知
    public void before() {
        System.out.println("这是前置通知");
    }

    // 后置通知
    public void afterReturning() {
        System.out.println("这是后置通知(方法不出现异常)");
    }

/**
     * 环绕通知
     */
    public Object arround(ProceedingJoinPoint point){
        Object o=null;
        try {
            System.out.println("==前置通知==>开启事务......"
);
            o=point.proceed();//调用目标方法，即使目标方法有参
数，这里也不用传递，因为底层知道
            System.out.println("==后置通知==>后置通知,提交事
务......");
        } catch (Throwable e) {
            System.out.println("==异常通知==>回滚事务,目标方法
出现了异常......异常类型是:"+e.getMessage());
            e.printStackTrace();
        }finally{
            System.out.println("==最终通知==>......");
        }
        return o;
    }
    public void afterException() {
        System.out.println("异常通知!");
    }

    public void after() {
        System.out.println("这也是后置通知,就算方法发生异常也会
调用!");
    }
}
```

3.1.4 织入目标对象(xml)

在applicationContext-aop.xml 中配置

注意:添加了 aop命名空间

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/conte
xt"
    xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/
beans
       http://www.springframework.org/schema/beans/spring-
beans-4.2.xsd
       http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-
context-4.2.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-
aop-4.2.xsd">

    <!--target对象，再Java代码中获得动态代理类的实例对象也是通过这
个Bean来获得-->
    <bean name="userService"
class="com.bruceliu.service.UserServiceImpl" />

    <!--  切面+advice（通知类，增强类） -->
    <bean name="myAdvice"
class="com.bruceliu.dao.MyAdivce" />

    <!-- 将增强类织入目标对象 -->
    <aop:config>
          <!--
             com.itqf.spring.service.UserServiceImpl
             1 2 3 4
             1: 修饰符  public/private/*  可忽略
             2: 返回值  String/../*
             3: 全限定类名  类名  ..代表不限层数  *ServiceImpl
             4: 参数  (..)  ..表示任意参数
             5: throws 异常
            -->

        <!--切入点：哪个目标类中的哪个方法要被增强
            切入点表达式：expression(1 2 3 4 5)
        -->
        <aop:pointcut id="pc" expression="execution(*
com.bruceliu.service.*ServiceImpl.*(..))" />

        <!--在以下面这种方式配置好之后，idea会在目标类被增强的切入
点（方法）前显示一个标志-->
        <!--织入切面的advice-->
        <aop:aspect ref="myAdvice">
            <!--这行标签的意思是：把切面的myAdvice通知中的before
方法织入到pc指向的切入点指代
```

```
            的方法执行之前-->
            <aop:before method="before" pointcut-ref="pc"
/>
            <aop:after-returning method="afterReturning"
pointcut-ref="pc" />
            <aop:around method="around" pointcut-ref="pc"
/>
            <aop:after-throwing method="afterException"
pointcut-ref="pc" />
            <aop:after method="after" pointcut-ref="pc" />
        </aop:aspect>
    </aop:config>
</beans>
```

3.1.5 测试

```
public class TestUser4 {

    @Test
    public void test1() {
        // TODO 测试切面引入
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext-
aop.xml");
        //这里从容器中获取动态代理类的实例对象时，可以使用xml中配置
的目标类所在的bean的id，因为容器中用动态代理类的实例对象替换掉了目标
类对象
        UserService userServiceImpl =
context.getBean("userService", UserService.class);
        userServiceImpl.addUser(null);
        userServiceImpl.deleteUser(11);
    }
}
```

3.2 AspectJ基于注解

在使用@Aspect之前,首先必须保证所使用的Java是 5.0 以上版本,否则无法使用注解技术.

Spring 在处理@Aspect注解表达式时,需要将Spring的asm模块添加到类路径中,asm是轻量级的字节码处理框架,因为Java的反射机制无法获取入参名,Spring利用asm处理@Aspect中所描述的方法入参名.

此外,Spring采用AspectJ提供的@Aspect注解类库及相应的解析类库,需要在pom.xml文件中添加aspectj.weaver和aspectj.tools类包的依赖.

注解名称	描述
@Aspect	用于定义一个切面。
@Pointcut	用于定义切入点表达式。在使用时还需定义一个包含名字和任意参数的方法签名来表示切入点名称。实际上，这个方法签名就是一个返回值为 void，且方法体为空的普通的方法。
@Before	用于定义前置通知，相当于 BeforeAdvice。在使用时，通常需要指定一个 value 属性值，该属性值用于指定一个切入点表达式（可以是已有的切入点，也可以直接定义切入点表达式）。
@AfterReturning	用于定义后置通知，相当于 AfterReturningAdvice。在使用时可以指定 pointcut/value 和 returning 属性，其中 pointcut/value 这两个属性的作用一样，都用于指定切入点表达式。returning 属性值用于表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法的返回值。
@Around	用于定义环绕通知，相当于 MethodInterceptor。在使用时需要指定一个 value 属性，该属性用于指定该通知被织入的切入点。
@AfterThrowing	用于定义异常通知来处理程序中未处理的异常，相当于 ThrowAdvice。在使用时可指定 pointcut/value 和 throwing 属性。其中 pointcut/value 用于指定切入点表达式，而 throwing 属性值用于指定一个形参名来表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法抛出的异常。
@After	用于定义最终 final 通知，不管是否异常，该通知都会执行。使用时需要指定一个 value 属性，该属性用于指定该通知被织入的切入点。
@DeclareParents	用于定义引介通知，相当于 IntroductionInterceptor（不要求掌握）。

3.2.1 创建配置文件

class-path路径下创建applicationContext-aop-annotation.xml

配置目标对象,配置通知对象,开启注解

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/conte
xt"
    xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/
beans
       http://www.springframework.org/schema/beans/spring-
beans-4.2.xsd
       http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-
context-4.2.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-
aop-4.2.xsd ">
    <!--配置目标类-->
    <bean name="userService"
class="com.bruceliu.service.UserServiceImpl" />

    <!--配置切面类 -->
    <bean name="myAdvice"
class="com.bruceliu.dao.MyAdivce1" />
```

```
    <!-- 配置将增强织入目标对象 使用注解的方式 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

    <!--注意：配置文件中的这三个配置完全可以使用注解配置在类中，
             切面类的注解根据情况选择@Service、@Controller……；
             配置切面类的Bean使用@Componment注解；
             配置最后一个可以使用注解
@EableAspectJAutoProcy(proxyTargetClass=false)。
    -->


</beans>
```

3.2.2 修改增强类

MyAdvice

1. 在切面类上添加 @Aspect 注解
2. 配置通知的两种方式：
    a. 可以将 execution 直接定义在通知方法名上，如下：

```
// 后置通知
@AfterReturning("execution(* com..service.*ServiceImpl.*(..))")
public void afterReturning() {
    System.out.println("这是后置通知(如果出现异常不会调用)!!");
}
```

    b. 但是在注解中的切入点表达式太长，多处使用的话会造成冗余；这个时候可以定义一个方法，将切入点表达式抽取出来， 如下面的 pc() 方法，然后在通知方法上指定 @Before("MyAdvice.pc()")

```
@Pointcut("execution(* com.bruceliu.service.*ServiceImpl.*(..))")
public void pc() {

}

// 前置通知
@Before("MyAdivce.pc()")
public void before() {
    System.out.println("这是前置通知");
}
```

- 完整定义增强类

```
@Aspect
public class MyAdivce1 {

    /**
     * //前置通知：目标方法运行之前调用 //后置通知(如果出现异常不会调用)：在目标方法运行之后调用 //环绕通知：在目标方法之前和之后都调用
     * //异常拦截通知：如果出现异常，就会调用 //后置通知(无论是否出现 异常都会调用)：在目标方法运行之后调用
     */

    @Pointcut("execution(* com..*ServiceImpl.*(..))")
```

```
    public void pc() {

    }

    // 前置通知
    @Before("MyAdivce1.pc()")
    public void before() {
        System.out.println("这是前置通知");
    }

    // 后置通知
    @AfterReturning("MyAdivce1.pc()")
    public void afterReturning() {
        System.out.println("这是后置通知(方法不出现异常)");
    }

    @Around("execution(* com..*ServiceImpl.*(..))")
    public Object around(ProceedingJoinPoint point) throws
Throwable {
        System.out.println("这是环绕通知之前部分!!");
        Object object = point.proceed(); // 调用目标方法
        System.out.println("这是环绕通知之后的部分!!");
        return object;
    }

    @AfterThrowing("execution(* com..*ServiceImpl.*(..))")
    public void afterException() {
        System.out.println("异常通知!");
    }

    @After("execution(* com..*ServiceImpl.*(..))")
    public void after() {
        System.out.println("这也是后置通知,就算方法发生异常也会
调用!");
    }

}
```

- 测试类

```java
public class TestUser5 {

    @Test
    public void test1() {
        // TODO 测试切面引入
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext-aop-
annotation.xml");
        UserService userServiceImpl =
context.getBean("userService", UserService.class);
        userServiceImpl.addUser(null);
        userServiceImpl.deleteUser(11);
    }
}
```