# spring-day03

# 第一章 AOP面向切面编程

## 第一节 AOP的概述

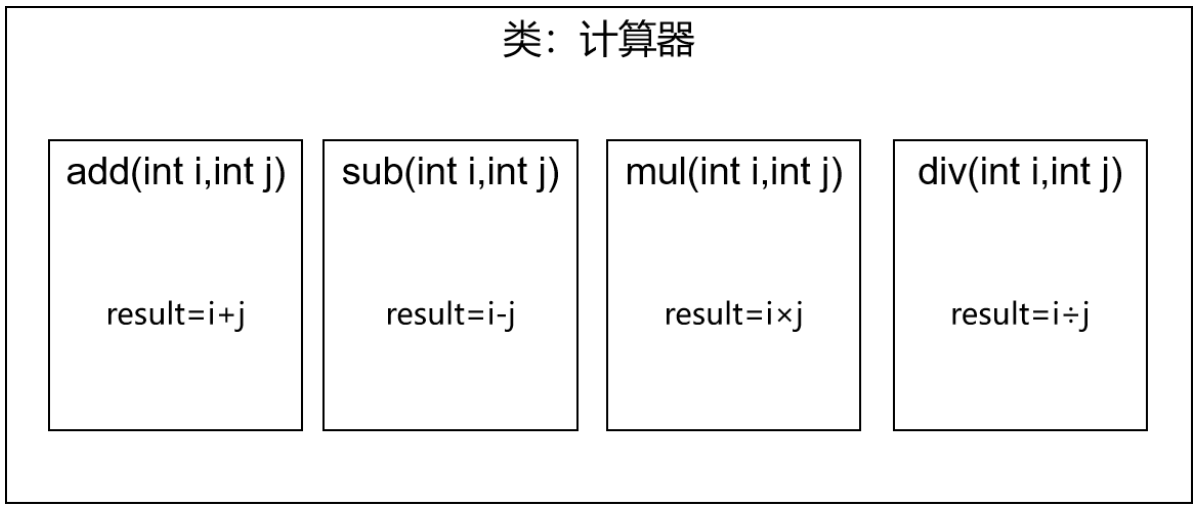
### 1. 为什么需要AOP

#### 1.1 情景设定

##### 1.1.1 声明一个计算器接口

```
public interface Calculator {

    int add(int i, int j);

    int sub(int i, int j);

    int mul(int i, int j);

    int div(int i, int j);

}
```

##### 1.1.2 给接口声明一个纯净版实现类

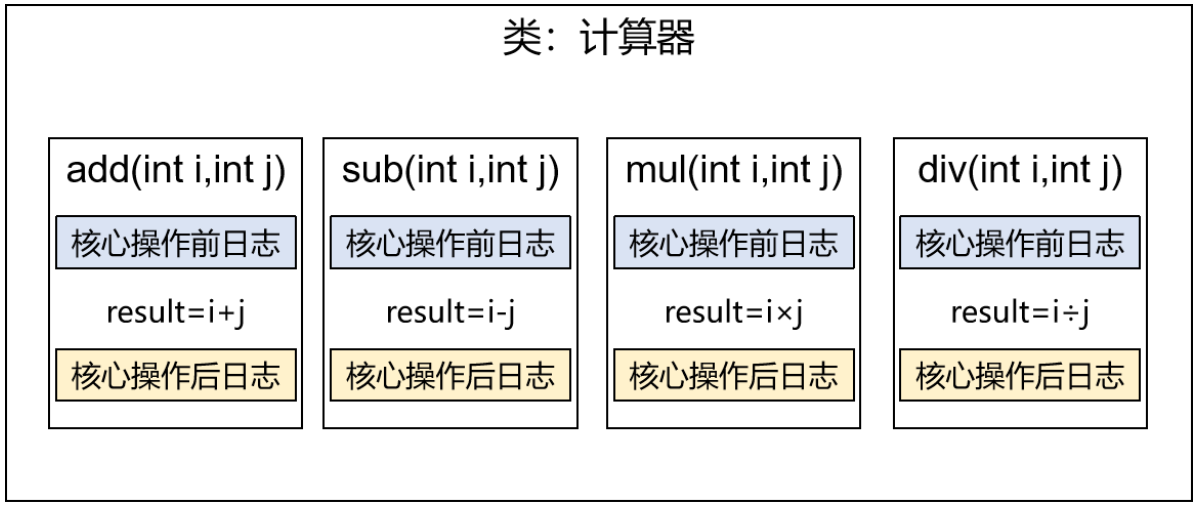



```
package com.atguigu.proxy.imp;

import com.atguigu.proxy.api.Calculator;

public class CalculatorPureImpl implements Calculator {

    @Override
    public int add(int i, int j) {
        int result = i + j;
```

```
        return result;
    }

    @Override
    public int sub(int i, int j) {

        int result = i - j;

        return result;
    }

    @Override
    public int mul(int i, int j) {

        int result = i * j;

        return result;
    }

    @Override
    public int div(int i, int j) {

        int result = i / j;

        return result;
    }
}
```

#### 1.1.3 需求

在计算器的每个方法执行前后加入日志打印:




#### 1.1.4 实现方案探讨

方案一: 在每个方法的前后都加上日志打印的代码

方案二: 创建一个工具类，将日志打印的代码写在工具类中，然后在每个方法的前后直接调用工具类中的方法打印日志

方案三: 创建一个父类，在父类的方法中打印日志，子类重写父类的方法(对目前功能不适用)

**方案四: 动态代理**

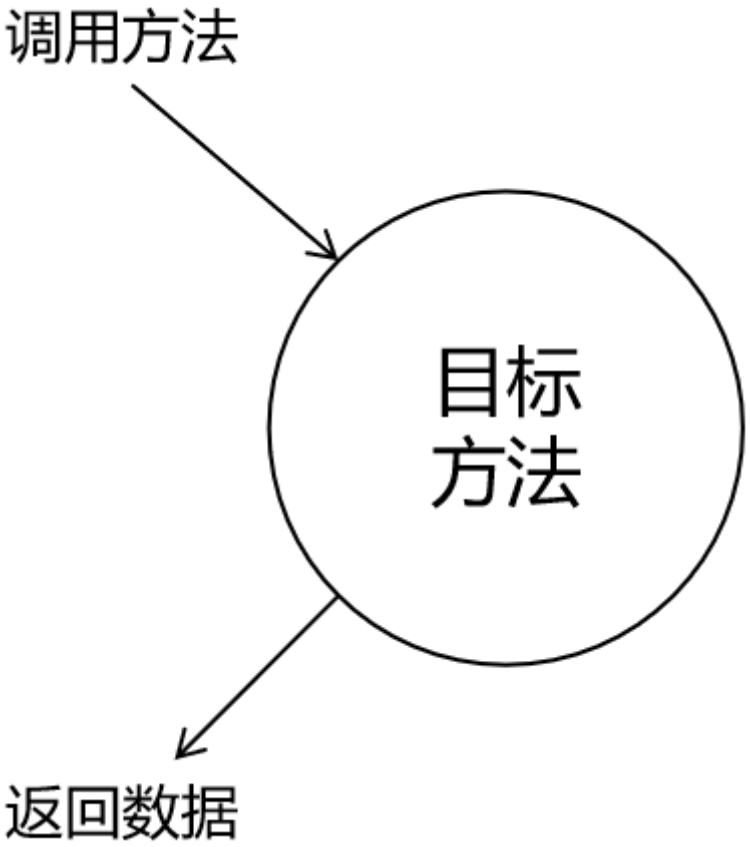
**方案五: AOP**

## 2. 代理模式

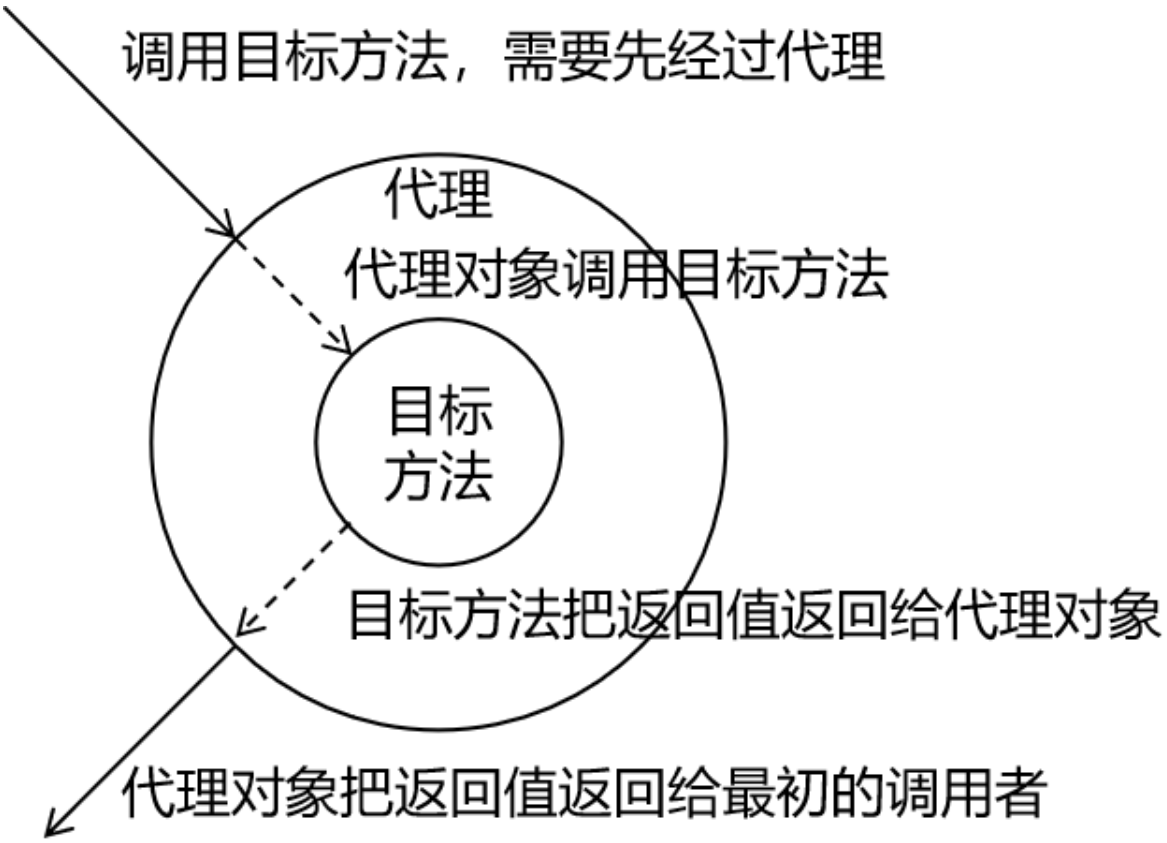
### 2.1 概念

二十三种设计模式中的一种，属于结构型模式。它的作用就是通过提供一个代理类，让我们在调用目标方法的时候，不再是直接对目标方法进行调用，而是通过代理类**间接**调用。让不属于目标方法核心逻辑的代码从目标方法中剥离出来——**解耦**。调用目标方法时先调用代理对象的方法，减少对目标方法的调用和打扰，同时让附加功能能够集中在一起也有利于统一维护。

未经过代理的情况:




使用了代理模式的情况:




### 2.2 相关术语

1. 代理: 又称之为代理者，用于将非核心逻辑剥离出来以后，封装这些非核心逻辑的类、对象、方法
2. 目标: 又称之为被代理者，用于执行核心逻辑，并且将代理者的非核心逻辑代码**套用**在目标类、对象、方法上

### 2.3 静态代理

#### 2.3.1 创建静态代理类：

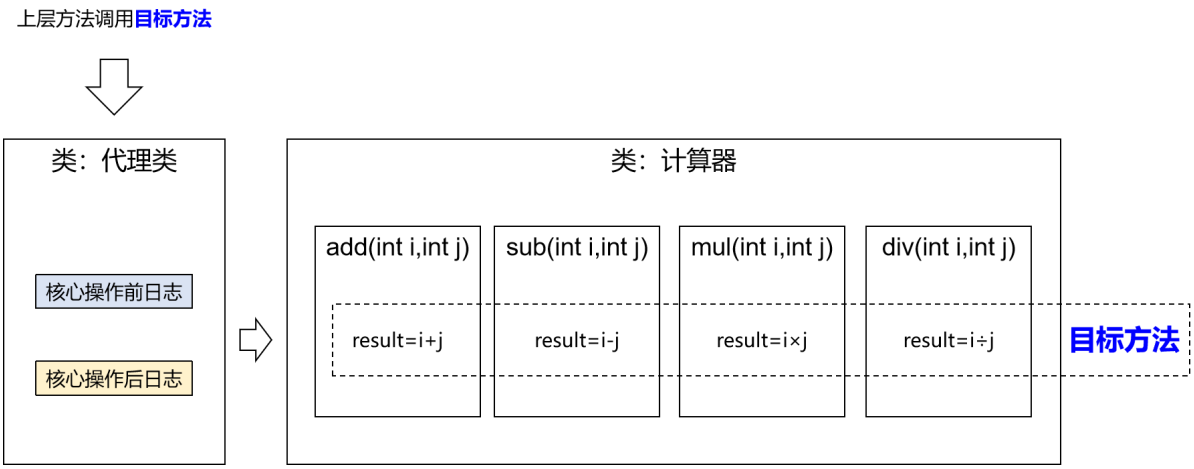
```
public class CalculatorStaticProxy implements Calculator {
    // 将被代理的目标对象声明为成员变量
    private Calculator target;

    public CalculatorStaticProxy(Calculator target) {
        this.target = target;
    }

    @Override
    public int add(int i, int j) {

        // 附加功能由代理类中的代理方法来实现
        System.out.println("[日志] add 方法开始了，参数是：" + i + "," + j);

        // 通过目标对象来实现核心业务逻辑
        int addResult = target.add(i, j);

        System.out.println("[日志] add 方法结束了，结果是：" + addResult);

        return addResult;
    }
    ……
```

#### 2.3.2 问题思考

静态代理确实实现了解耦，但是由于代码都写死了，完全不具备任何的灵活性。就拿日志功能来说，将来其他地方也需要附加日志，那还得再声明更多个静态代理类，那就产生了大量重复的代码，日志功能还是分散的，没有统一管理。

提出进一步的需求：将日志功能集中到一个代理类中，将来有任何日志需求，都通过这一个代理类来实现。这就需要使用动态代理技术了。

### 2.4 动态代理




#### 2.4.1 创建生产代理对象的工厂类

JDK本身就支持动态代理，这是反射技术的一部分。下面我们还是创建一个代理类（生产代理对象的工厂类）：

```
package com.atguigu.factory;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 包名:com.atguigu.factory
 *
 * @author Leevi
 * 日期2021-09-01  10:00
 */
public class LogDynamicProxyFactory<T> {
    private T target;

    public LogDynamicProxyFactory(T target) {
        this.target = target;
    }

    /**
     * 创建动态代理对象
     * @return
     */
    public T getProxy(){
        //使用JDK的动态代理技术
        //1. 获取被代理者的字节码对象
        Class<?> clazz = target.getClass();
```

```
        //2. 使用JDK中的Proxy.newProxyInstance(classLoader, interfaces, new
InvocationHandler()方法创建动态代理对象
        // 该方法的返回值就是动态代理对象
        //2.1 类加载器对象
        ClassLoader classLoader = clazz.getClassLoader();
        //2.2 需要代理的接口:如果明确要代理什么接口，那么就可以直接写new Class[]{要代理的接
口.class}
        //如果不明确要代理什么接口，那么就获取被代理者实现的所有接口: 被代理者的字节码对
象.getInterfaces();
        Class<?>[] interfaces = clazz.getInterfaces();
        //2.3 InvocationHandler接口的对象: 使用匿名内部类
        T t = (T) Proxy.newProxyInstance(classLoader, interfaces, new
InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                //invoke()方法:该方法会在代理对象调用任意方法的时候执行
                //所以我们就在这个方法中编写代理逻辑
                //参数一:proxy对象表示代理对象本身
                //参数二:method表示代理对象所调用的方法本身
                //参数三:args表示代理对像所调用的方法中传入的参数

                //编写代理逻辑:我的想法是在执行被代理对象的add()、sub()、mul()、div()这四
个方法的前后添加日志打印
                //判断方法：是否是这四个方法
                String proxyMethodName = method.getName();
                //如果是: 则添加前后日志打印
                if (proxyMethodName.equals("add") ||
proxyMethodName.equals("sub") || proxyMethodName.equals("mul") ||
proxyMethodName.equals("div")) {
                    //核心逻辑之前打印日志
                    System.out.println("[日志] "+proxyMethodName+"方法开始了，参数
是：" + args[0] + "," + args[1]);
                    //执行被代理者的当前方法
                    Object result = method.invoke(target, args);
                    //核心逻辑之后打印日志
                    System.out.println("[日志] "+proxyMethodName+" 方法结束了，结果
是：" + result);
                    //返回执行结果
                    return result;
                }
                //如果不是: 就按照被代理者原本的方法执行
                return method.invoke(target,args);
            }
        });
        return t;
    }
}
```

### 2.4.2 测试

```
@Test
public void testDynamicProxy(){
    //创建被代理者
    Calculator calculator = new CalculatorPureImpl();
    //1. 创建动态代理工厂类的对象
    LogDynamicProxyFactory<Calculator> proxyFactory = new
LogDynamicProxyFactory<Calculator>(calculator);
    //2. 使用工厂对象创建动态代理对象
    Calculator calculatorProxy = proxyFactory.getProxy();
    //3. 使用代理对象调用方法
    System.out.println(calculatorProxy.sub(3, 4));
}
```

## 3. AOP的相关概念

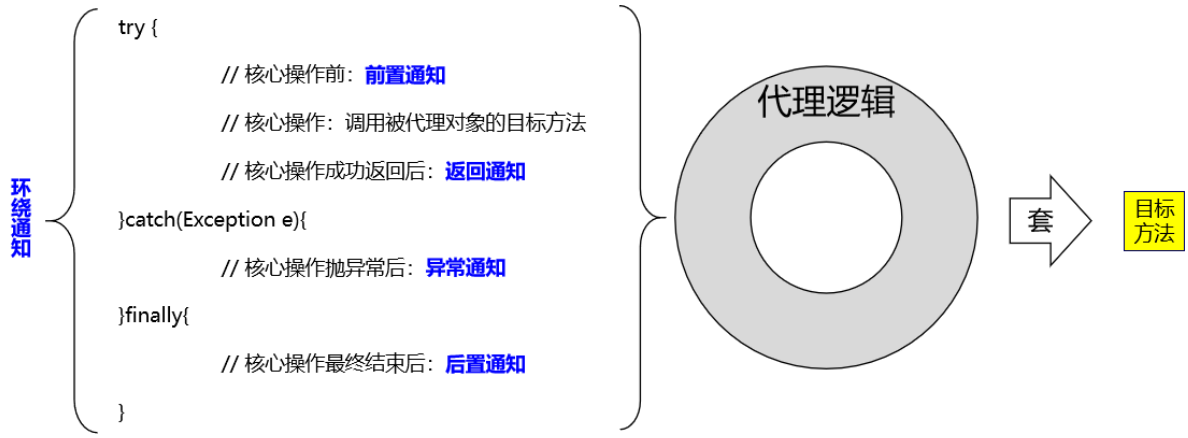
### 3.1 概念

AOP：Aspect Oriented Programming面向切面编程

### 3.2 作用

1. 简化代码：把方法中固定位置的重复的代码**抽取**出来，让被抽取的方法更专注于自己的核心功能，提高内聚性。
2. 代码增强：把抽取出来的特定的功能封装到切面类中，看哪里有需要，就往上套，被**套用**了切面逻辑的方法就被切面给增强了。

### 3.3 AOP的核心思路

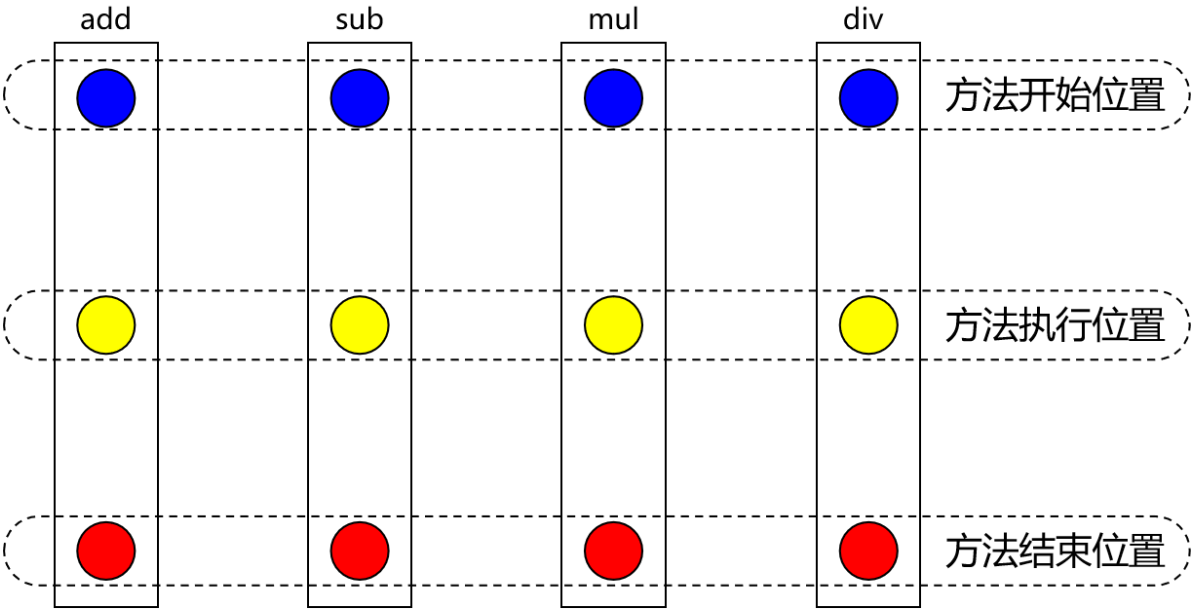



### 3.4 AOP的相关术语

#### 3.4.1 横切关注点(了解)

横切关注点是从每个方法中抽取出来的同一类非核心业务。在同一个项目中，我们可以使用多个横切关注点对相关方法进行多个不同方面的增强。

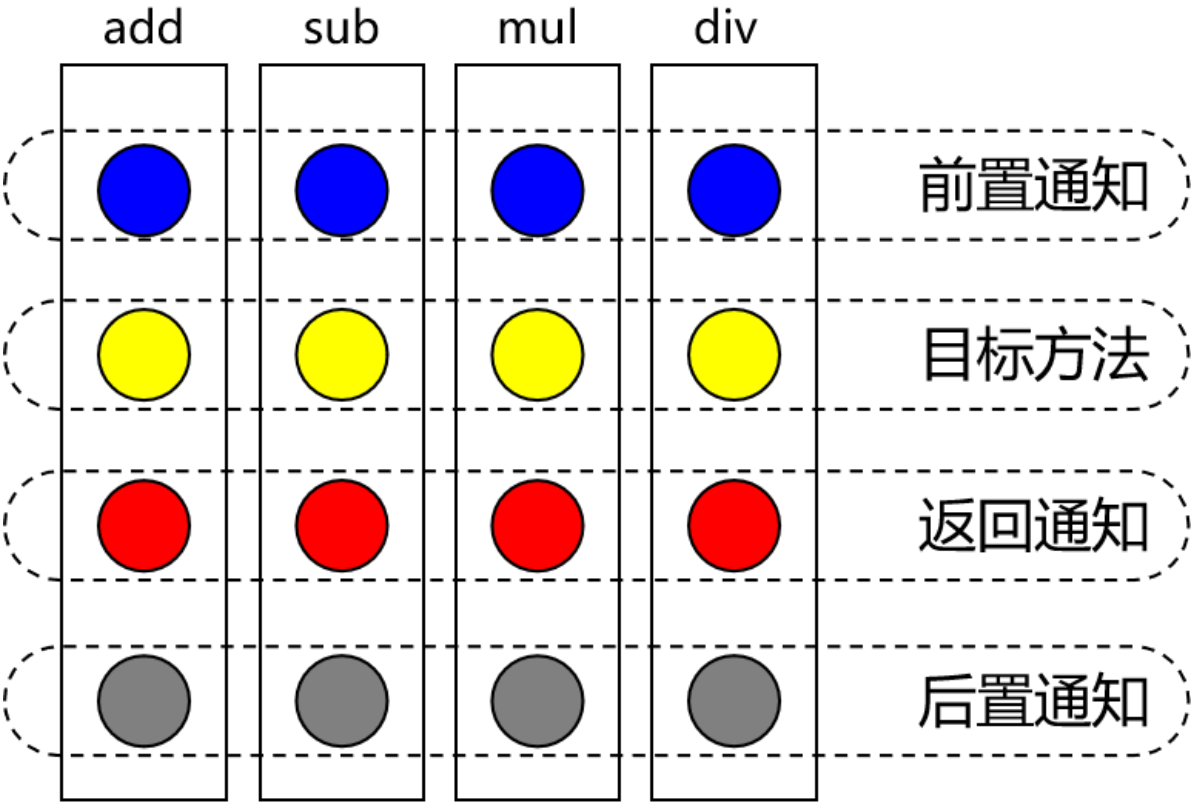
这个概念不是语法层面天然存在的，而是根据附加功能的逻辑上的需要：有十个附加功能，就有十个横切关注点。




### 3.4.2 通知(重要)

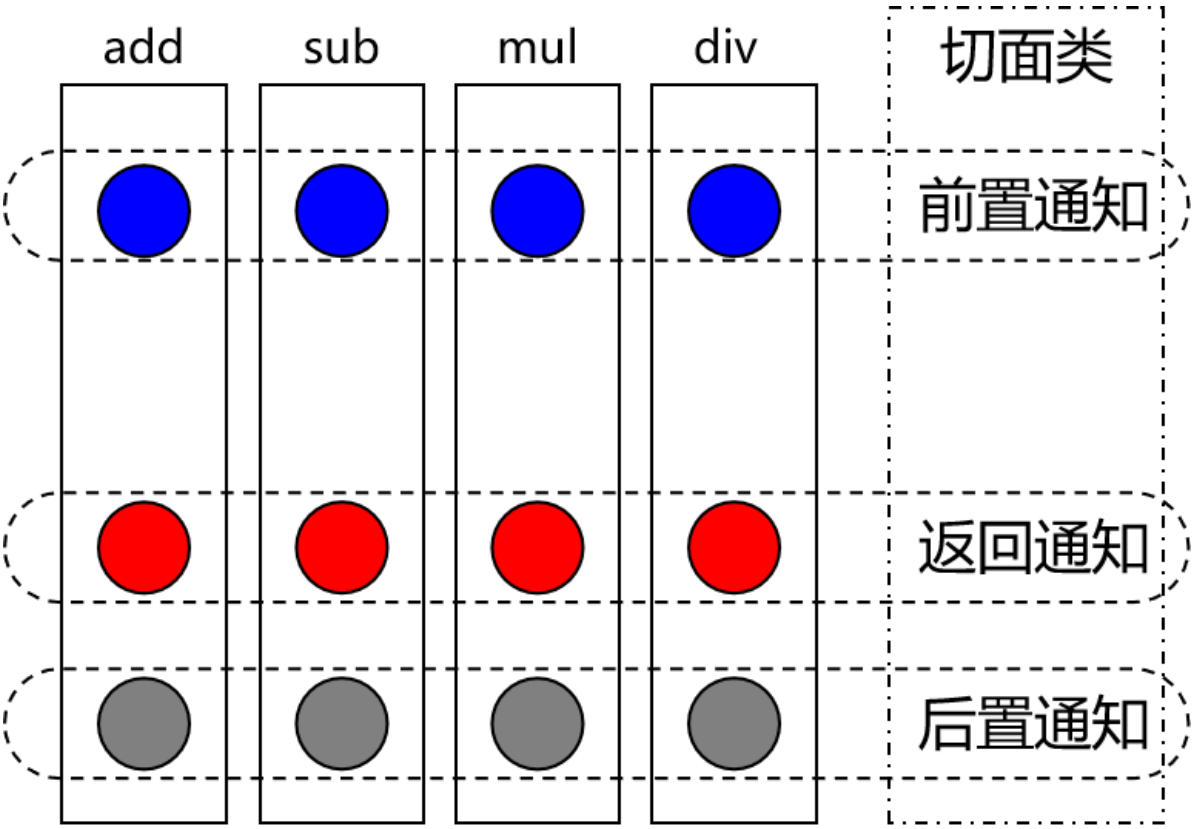
每一个横切关注点上要做的事情都需要写一个方法来实现，这样的方法就叫通知方法。

- 前置通知：在被代理的目标方法**前**执行
- 返回通知：在被代理的目标方法**成功结束**后执行（**寿终正寝**）
- 异常通知：在被代理的目标方法**异常结束**后执行（**死于非命**）
- 后置通知：在被代理的目标方法**最终结束**后执行（**盖棺定论**）
- 环绕通知：使用try...catch...finally结构围绕**整个**被代理的目标方法，包括上面四种通知对应的所有位置



#### 3.4.3 切面(重要)

封装通知方法的类。




#### 3.4.4 目标(重要)

被代理的目标对象,执行核心业务代码的那个对象

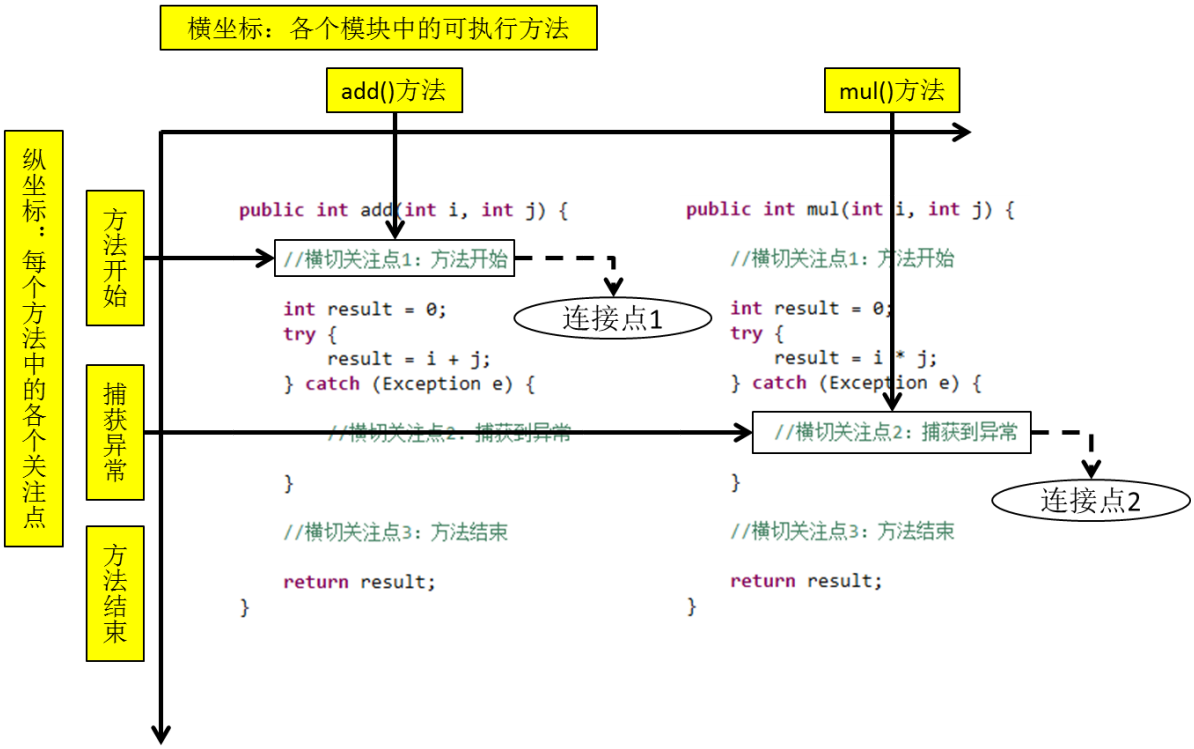
#### 3.4.5 代理(了解)

向目标对象应用通知之后创建的代理对象。

#### 3.4.6 连接点(了解)

这也是一个纯逻辑概念，不是语法定义的。

把方法排成一排，每一个横切位置看成x轴方向，把方法从上到下执行的顺序看成y轴，x轴和y轴的交叉点就是连接点。连接点其实就是各个方法中可以被增强或修改的点

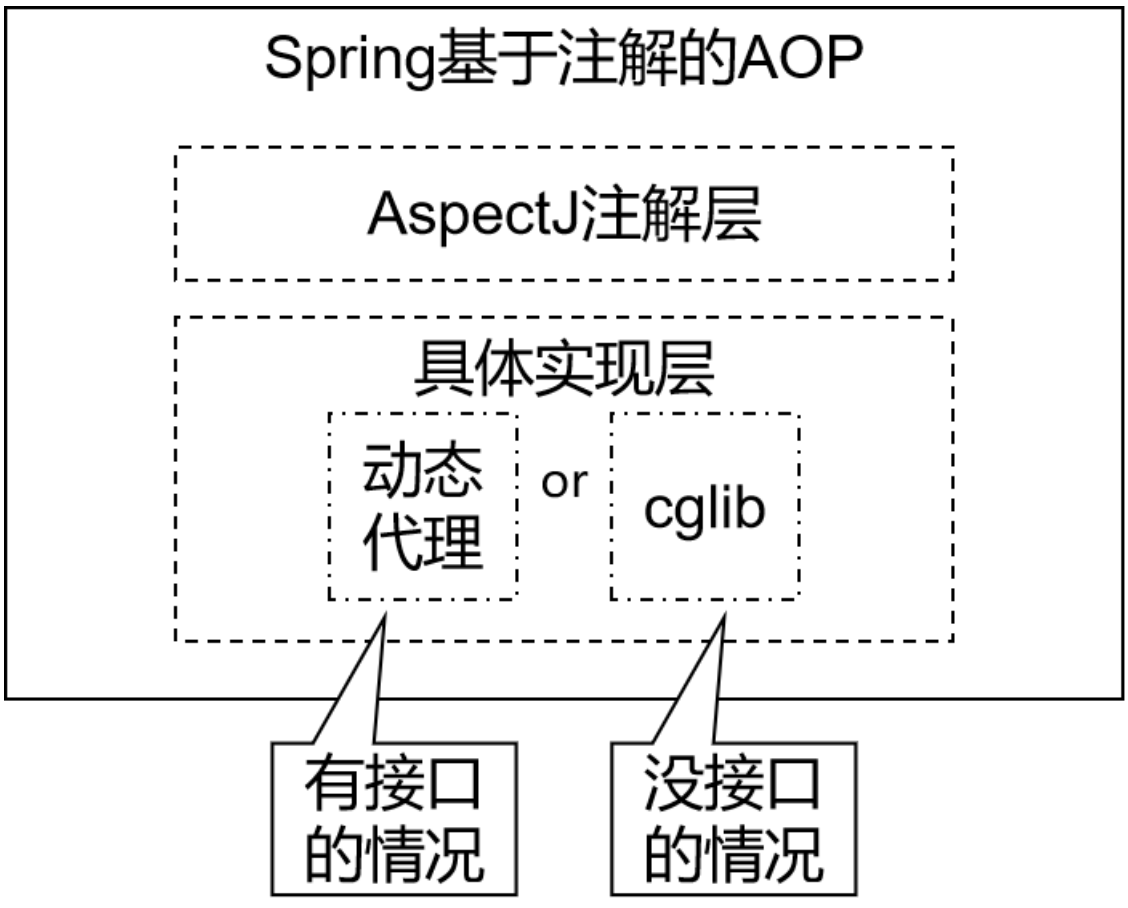



#### 3.4.7 切入点(重点)

每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物（从逻辑上来说）。而切入点指的则是方法中真正要去配置增强或者配置修改的地方

## 第二节 基于注解方式配置AOP

### 1. 基于注解的AOP用到的技术



- 动态代理（InvocationHandler）：JDK原生的实现方式，需要被代理的目标类必须实现接口。因为这个技术要求**代理对象和目标对象实现同样的接口。**
- cglib：通过**继承被代理的目标类**实现代理，所以不需要目标类实现接口。
- AspectJ：本质上是静态代理，**将代理逻辑"织入"被代理的目标类编译得到的字节码文件**，所以最终效果是动态的。weaver就是织入器。Spring只是借用了AspectJ中的注解。

## 2. 实现基于注解的AOP

### 2.1 加入依赖

在IOC所需依赖基础上再加入下面依赖即可：

```
<dependencies>
    <!-- 基于Maven依赖传递性，导入spring-context依赖即可导入当前所需所有jar包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.1</version>
    </dependency>
    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!-- spring-aspects会帮我们传递过来aspectjweaver -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.3.1</version>
    </dependency>
    <!--spring整合Junit-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.3.1</version>
    </dependency>

</dependencies>
```

### 2.2 准备被代理的目标资源

#### 2.2.1 接口

```
public interface Calculator {

    int add(int i, int j);

    int sub(int i, int j);

    int mul(int i, int j);

    int div(int i, int j);

}
```

#### 2.2.2 接口的实现类

在Spring环境下工作，所有的一切都必须放在IOC容器中。现在接口的实现类是AOP要代理的目标类，所以它也必须放入IOC容器。

```
package com.atguigu.component;

import org.springframework.stereotype.Component;

/**
 * 包名:com.atguigu
 *
 * @author Leevi
 * 日期2021-09-01  09:21
 */
@Component
public class CalculatorPureImpl implements Calculator {
    @Override
    public int add(int i, int j) {
        int result = i + j;
        //int num = 10 / 0;
        return result;
    }

    @Override
    public int sub(int i, int j) {
        int result = i - j;
        return result;
    }

    @Override
    public int mul(int i, int j) {

        int result = i * j;
        return result;
    }

    @Override
    public int div(int i, int j) {

        int result = i / j;
        return result;
    }
}
```

#### 2.2.3 创建切面类

```
package com.atguigu.aspect;

import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

/**
 * 包名:com.atguigu.aspect
 *
 * @author Leevi
 * 日期2021-09-01  11:32
```

```
 * Aspect注解:指定一个切面类
 * Component注解: 对这个切面类进行IOC
 *
 * 注解AOP的关键点:
 * 1. 一定要在配置文件中加上<aop:aspectj-autoproxy />表示允许自动代理
 * 2. 切面类一定要加上Aspect注解，并且切面类一定要进行IOC
 * 3. 其它的类该进行IOC和依赖注入的就一定要进行IOC和依赖注入
 * 4. 通知上一定要指定切入点(怎么使用切入点表达式描述切入点又是一个难点)
 */
@Aspect
@Component
public class LogAspect {
    @Before("execution(int com.atguigu.component.CalculatorPureImpl.*
(int,int))")
    public void printLogBeforeCore(){
        System.out.println("[前置通知]在方法执行之前打印日志...");
    }

    @AfterReturning("execution(int com.atguigu.component.CalculatorPureImpl.*
(int,int))")
    public void printLogAfterReturning(){
        System.out.println("[返回通知]在方法执行成功之后打印日志...");
    }

    @AfterThrowing("execution(int com.atguigu.component.CalculatorPureImpl.*
(int,int))")
    public void printLogAfterThrowing(){
        System.out.println("[AOP异常通知]在方法抛出异常之后打印日志...");
    }

    @After("execution(int com.atguigu.component.CalculatorPureImpl.*(int,int))")
    public void printLogFinallyEnd(){
        System.out.println("[AOP后置通知]在方法最终结束之后打印日志...");
    }
}
```

#### 2.2.4 创建Spring的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--包扫描-->
    <context:component-scan base-package="com.atguigu"/>

    <!--允许注解AOP-->
    <aop:aspectj-autoproxy />
</beans>
```

#### 2.2.5 测试

```
package com.atguigu;

import com.atguigu.component.Calculator;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * 包名:com.atguigu
 *
 * @author Leevi
 * 日期2021-09-01  11:34
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-application.xml")
public class TestAop {
    @Autowired
    private Calculator calculator;
    @Test
    public void testAdd(){
        //调用CalculatorPureImpl对象的add()方法
        System.out.println("返回值是:"+calculator.add(1, 2));
    }
}
```

打印效果如下：

> [AOP前置通知] 方法开始了 方法内部 result = 12 [AOP返回通知] 方法成功返回了 [AOP后置通知] 方法最终结束了 方法外部 add = 12

### 2.3 通知执行顺序

- Spring版本5.3.x以前：
  - 前置通知
  - 目标操作
  - 后置通知
  - 返回通知或异常通知
- Spring版本5.3.x以后：
  - 前置通知
  - 目标操作
  - 返回通知或异常通知
  - 后置通知

## 3. 在通知内部获取细节信息

### 3.1 JoinPoint接口

org.aspectj.lang.JoinPoint

- 要点1：JoinPoint接口通过getSignature()方法获取目标方法的签名
- 要点2：通过目标方法签名对象获取方法名
- 要点3：通过JoinPoint对象获取外界调用目标方法时传入的实参列表组成的数组

```
// @Before注解标记前置通知方法
// value属性：切入点表达式，告诉Spring当前通知方法要套用到哪个目标方法上
// 在前置通知方法形参位置声明一个JoinPoint类型的参数，Spring就会将这个对象传入
// 根据JoinPoint对象就可以获取目标方法名称、实际参数列表
@Before(value = "execution(public int
com.atguigu.aop.api.Calculator.add(int,int))")
public void printLogBeforeCore(JoinPoint joinPoint) {
    // 1.通过JoinPoint对象获取目标方法签名对象
    // 方法的签名：一个方法的全部声明信息
    Signature signature = joinPoint.getSignature();

    // 2.通过方法的签名对象获取目标方法的详细信息
    String methodName = signature.getName();
    System.out.println("methodName = " + methodName);

    int modifiers = signature.getModifiers();
    System.out.println("modifiers = " + modifiers);

    String declaringTypeName = signature.getDeclaringTypeName();
    System.out.println("declaringTypeName = " + declaringTypeName);

    // 3.通过JoinPoint对象获取外界调用目标方法时传入的实参列表
    Object[] args = joinPoint.getArgs();

    // 4.由于数组直接打印看不到具体数据，所以转换为List集合
    List<Object> argList = Arrays.asList(args);

    System.out.println("[AOP前置通知] " + methodName + "方法开始了，参数列表：" +
argList);
}
```

需要获取方法签名、传入的实参等信息时，可以在通知方法声明JoinPoint类型的形参。

### 3.2 获取目标方法的方法返回值

只有在AfterReturning返回通知中才能够获取目标方法的返回值

```
// @AfterReturning注解标记返回通知方法
// 在返回通知中获取目标方法返回值分两步：
// 第一步：在@AfterReturning注解中通过returning属性设置一个名称
// 第二步：使用returning属性设置的名称在通知方法中声明一个对应的形参
@AfterReturning(
        value = "execution(public int com.atguigu.aop.api.Calculator.add(int,int))",
        returning = "targetMethodReturnValue"
)
public void printLogAfterCoreSuccess(JoinPoint joinPoint, Object targetMethodReturnValue) {
```

目标方法返回值

定义形参名称

通过@AfterReturning注解的returning属性获取目标方法的返回值

```
// @AfterReturning注解标记返回通知方法
// 在返回通知中获取目标方法返回值分两步：
// 第一步：在@AfterReturning注解中通过returning属性设置一个名称
// 第二步：使用returning属性设置的名称在通知方法中声明一个对应的形参
@AfterReturning(
```

```
        value = "execution(public int
com.atguigu.aop.api.Calculator.add(int,int))",
        returning = "targetMethodReturnValue"
)
public void printLogAfterCoreSuccess(JoinPoint joinPoint, Object
targetMethodReturnValue) {

    String methodName = joinPoint.getSignature().getName();

    System.out.println("[AOP返回通知] "+methodName+"方法成功结束了，返回值是：" +
targetMethodReturnValue);
}
```

### 3.3 获取目标方法抛出的异常

只有在 AfterThrowing 异常通知中才能获取到目标方法抛出的异常


通过@AfterThrowing注解的throwing属性获取目标方法抛出的异常对象

```
// @AfterThrowing注解标记异常通知方法
// 在异常通知中获取目标方法抛出的异常分两步：
// 第一步：在@AfterThrowing注解中声明一个throwing属性设定形参名称
// 第二步：使用throwing属性指定的名称在通知方法声明形参，Spring会将目标方法抛出的异常对象从这
里传给我们
@AfterThrowing(
        value = "execution(public int
com.atguigu.aop.api.Calculator.add(int,int))",
        throwing = "targetMethodException"
)
public void printLogAfterCoreException(JoinPoint joinPoint, Throwable
targetMethodException) {

    String methodName = joinPoint.getSignature().getName();

    System.out.println("[AOP异常通知] "+methodName+"方法抛异常了，异常类型是：" +
targetMethodException.getClass().getName());
}
```

打印效果局部如下：

> [AOP异常通知] div方法抛异常了，异常类型是：java.lang.ArithmeticException
>
> java.lang.ArithmeticException: / by zero

> at com.atguigu.aop.imp.CalculatorPureImpl.div(CalculatorPureImpl.java:42) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.Method.invoke(Method.java:498) at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:344)

## 4. 切入点

### 4.1 重用切入点

#### 4.1.1 声明切入点

在一处声明切入点表达式之后，其他有需要的地方引用这个切入点表达式。易于维护，一处修改，处处生效。声明方式如下：

```
@Pointcut("execution(int com.atguigu.component.CalculatorPureImpl.*(int,int))")
public void calculatorPointCut(){
}
```

#### 4.1.2 同一个类内部引用切入点

通过方法名引入

```
@Before("calculatorPointCut()")
public void printLogBeforeCore(JoinPoint joinPoint){
```

#### 4.1.3 在其它类中引用切入点

通过全限定名引入

```
@Before("com.atguigu.pointcut.AtguiguPointCut.calculatorPointCut()")
public void printLogBeforeCore(JoinPoint joinPoint){}
```

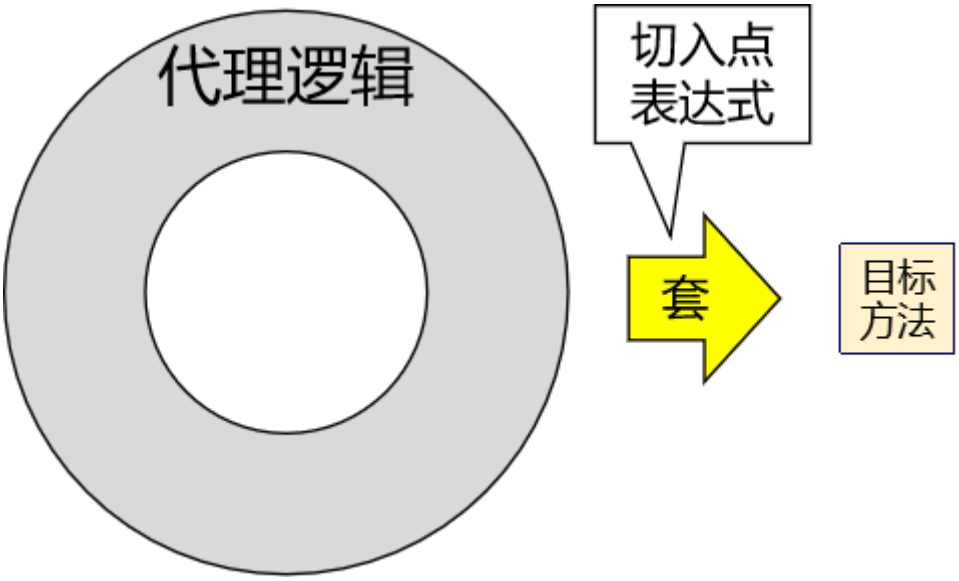
#### 4.1.4 对项目中的所有切入点进行统一管理

而作为存放切入点表达式的类，可以把整个项目中所有切入点表达式全部集中过来，便于统一管理：

```
package com.atguigu.pointcut;

import org.aspectj.lang.annotation.Pointcut;

/**
 * 包名:com.atguigu.pointcut
 *
 * @author Leevi
 * 日期2021-09-01  15:30
 */
public class AtguiguPointCut {
    @Pointcut("execution(int com.atguigu.component.CalculatorPureImpl.*
(int,int))")
    public void calculatorPointCut(){
    }
}
```

### 4.2 切入点表达式语法

#### 4.2.1 切入点表达式的作用




切入点表达式的作用是用于描述将代理逻辑套用在哪些目标方法上

#### 4.2.2 语法细节

- 用*号代替“权限修饰符”和“返回值”部分表示“权限修饰符”和“返回值”不限
- 在包名的部分，一个“*”号只能代表包的层次结构中的一层，表示这一层是任意的。
  - 例如：*.Hello匹配com.Hello，不匹配com.atguigu.Hello
- 在包名的部分，使用“*..”表示包名任意、包的层次深度任意
- 在类名的部分，类名部分整体用*号代替，表示类名任意
- 在类名的部分，可以使用*号代替类名的一部分

```
*Service
```

上面例子表示匹配所有名称以Service结尾的类或接口

- 在方法名部分，可以使用*号表示方法名任意
- 在方法名部分，可以使用*号代替方法名的一部分

```
*Operation
```

上面例子表示匹配所有方法名以Operation结尾的方法

- 在方法参数列表部分，使用(..)表示参数列表任意
- 在方法参数列表部分，使用(int,..)表示参数列表以一个int类型的参数开头
- 在方法参数列表部分，基本数据类型和对应的包装类型是不一样的
  - 切入点表达式中使用 int 和实际方法中 Integer 是不匹配的
- 在方法返回值部分，如果想要明确指定一个返回值类型，那么权限修饰符不能使用 *

```
execution(public int *..*Service.*(.., int))
```

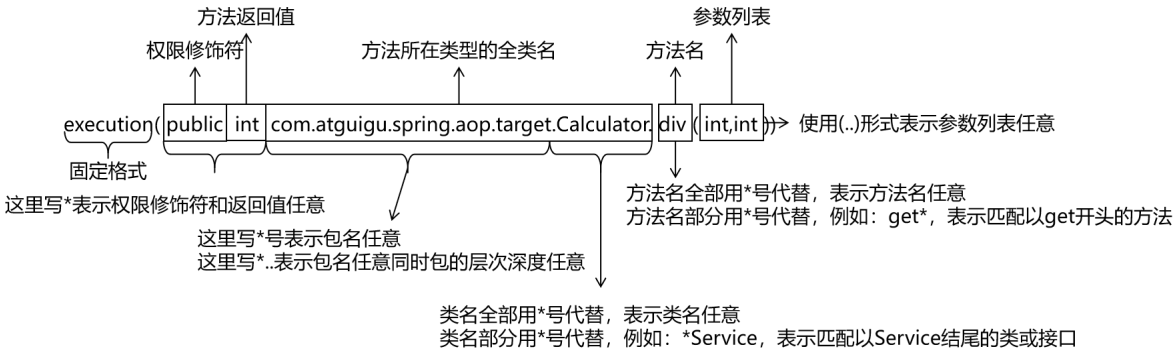
上面例子是对的，下面例子是错的：

```
execution(* int *..*Service.*(.., int))
```

但是public *表示权限修饰符明确，返回值任意是可以的。

- 对于execution()表达式整体可以使用三个逻辑运算符号
  - execution() || execution()表示满足两个execution()中的任何一个即可
  - execution() && execution()表示两个execution()表达式必须都满足
  - !execution()表示不满足表达式的其他方法

#### 4.2.3 总结




## 5. 环绕通知

环绕通知对应整个try...catch...finally结构，可以在目标方法的各个部位进行套用代理逻辑，它能够真正介入并改变目标方法的执行

```
@Around("com.atguigu.pointcut.AtguiguPointCut.calculatorPointCut()")
public Object around(ProceedingJoinPoint proceedingJoinPoint){
    try {
        System.out.println("开启事务....");
        //环绕通知是可以介入到目标方法执行之前、返回值之前、出现异常之后、finally中等等各个部位执行

        //环绕通知可以在目标方法执行之前做一些事情：就相当于前置通知
        //获取目标方法的参数
        Object[] args = proceedingJoinPoint.getArgs();
        //改变目标方法的参数：例如要做一些统一的参数的处理逻辑
        //args[0] = 2;
        //args[1] = 4;

        //这句代码就是执行目标方法:也就是在这里开始你就能介入目标方法
        Object result = proceedingJoinPoint.proceed(args);
        //改变目标方法的返回值:
        //环绕通知可以在目标方法执行成功之后，做一些事情：就相当于返回通知
        //return 1000;

        System.out.println("提交事务...");
        return result;
    } catch (Throwable throwable) {
        throwable.printStackTrace();
        System.out.println("回滚事务...");
        //环绕通知可以在目标方法执行出现异常之后，做一些事情：就相当于异常通知
```

```
            throw new RuntimeException(throwable.getMessage());
        } finally {
            //环绕通知可以在目标方法执行成功或者出现异常之后，做一些事情：就相当于后置通知
            System.out.println("将连接恢复默认状态，归还连接...");
        }
    }
```

## 6. 切面的优先级(了解)

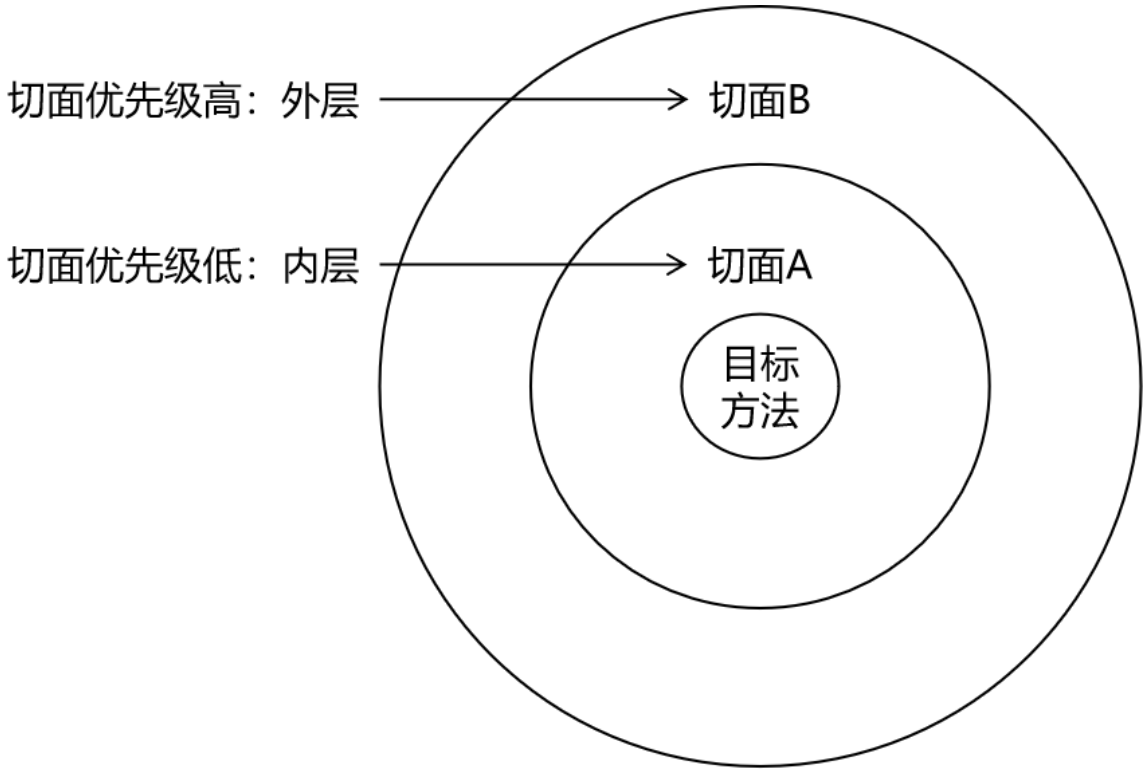
### 6.1 优先级的规则

相同目标方法上同时存在多个切面时，切面的优先级控制切面的**内外嵌套**顺序。

- 优先级高的切面：外面
- 优先级低的切面：里面

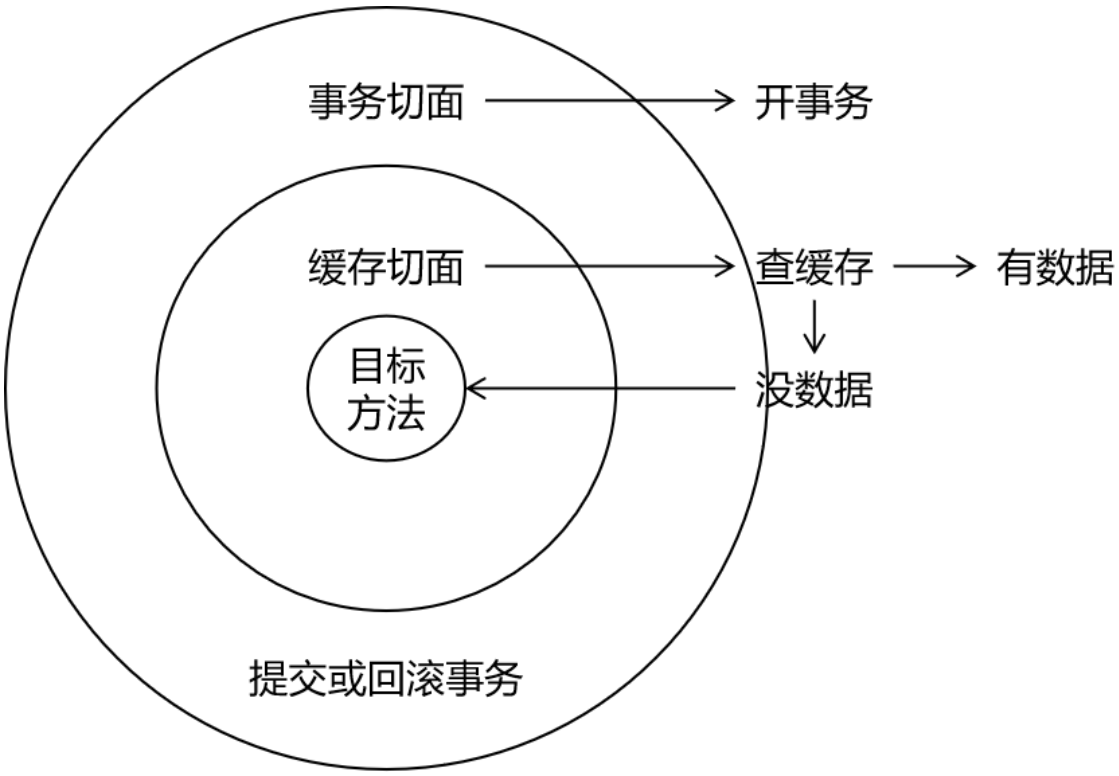
使用@Order注解可以控制切面的优先级：

- @Order(较小的数)：优先级高
- @Order(较大的数)：优先级低

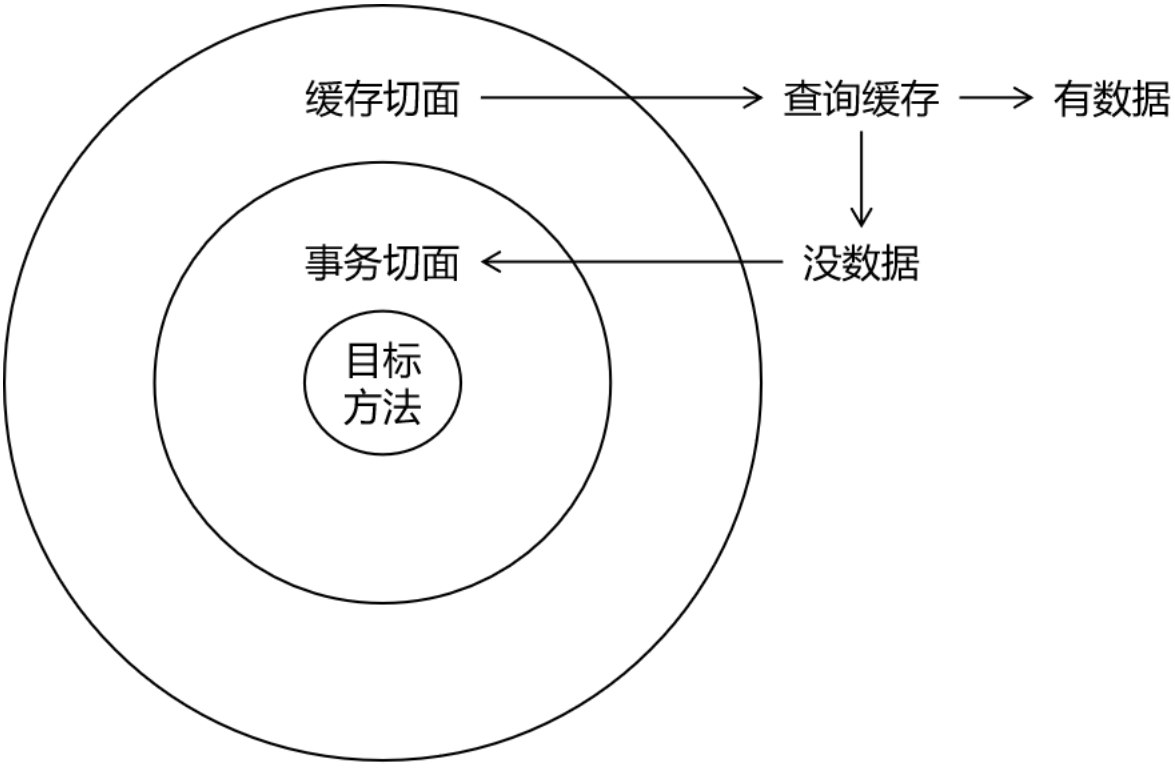



### 6.2 实际意义

实际开发时，如果有多个切面嵌套的情况，要慎重考虑。例如：如果事务切面优先级高，那么在缓存中命中数据的情况下，事务切面的操作都浪费了。




此时应该将缓存切面的优先级提高，在事务操作之前先检查缓存中是否存在目标数据。




## 7. CGLIB的动态代理

### 7.1 动态代理的分类

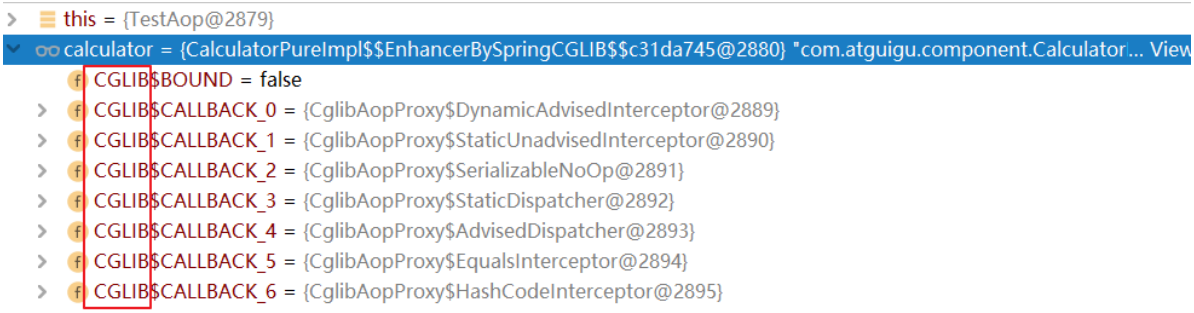
动态代理分成两种:

第一种是JDK内置的动态代理，这种动态代理需要被代理者实现接口，如果被代理者没有实现接口，那么则无法使用JDK的动态代理

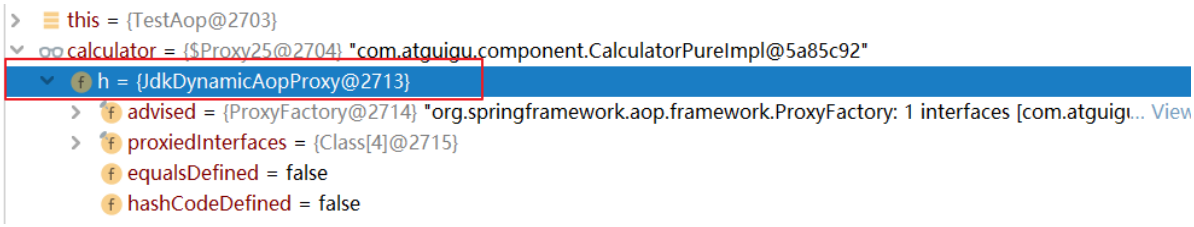
第二种是CGLIB的动态代理，在被代理类没有实现任何接口的情况下，Spring会自动使用cglib技术实现代理。

### 7.2 Debug查看

#### 7.2.1 没有实现接口情况




#### 7.2.2 有实现接口的情况




### 7.3 Spring中到底使用哪种动态代理

如果要创建代理对象的类实现了接口，那么就使用JDK的动态代理；如果要创建代理对象的类没有实现接口，那么就使用CGLIB的动态代理

# 第三节 基于XML方式配置AOP(了解)

## 1. 准备工作

### 1.1 加入依赖

和基于注解的AOP时一样。

### 1.2 准备代码

把基于注解的Module复制一份，修改Module名，并导入到工程中，然后去除所有AOP注解。

## 2. 配置Spring配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://www.springframework.org/schema/context
                           https://www.springframework.org/schema/context/spring-
context.xsd
                           http://www.springframework.org/schema/aop
                           https://www.springframework.org/schema/aop/spring-
aop.xsd">
    <!--包扫描-->
    <context:component-scan base-package="com.atguigu"/>
```

```
    <!--
        使用xml方式配置AOP:
            1. 切面: 封装非核心逻辑的那个类，非核心逻辑就是封装在切面的方法中
            2. 通知: 将非核心逻辑套在核心逻辑上进行执行
            3. 切入点: 核心逻辑
    -->
    <aop:config>
        <!--
            1. 切面: ref属性就是指定作为切面的那个对象的id，order属性表示切面的优先级
        -->
        <aop:aspect id="myAspect" ref="logAspect">
            <!--2. 通知-->
            <!--配置前置通知-->
            <aop:before method="printLogBeforeCore" pointcut-
ref="calculatorPoint"/>
            <!--配置返回通知-->
            <aop:after-returning method="printLogAfterReturning" pointcut-
ref="calculatorPoint" returning="result"/>
            <!--配置异常通知-->
            <aop:after-throwing method="printLogAfterThrowing" pointcut-
ref="calculatorPoint" throwing="throwable"/>
            <!--配置后置通知-->
            <aop:after method="printLogFinallyEnd" pointcut-
ref="calculatorPoint"/>
            <!--配置环绕通知-->
            <aop:around method="printLogAround" pointcut-ref="calculatorPoint"/>
            <!--3. 切入点-->
            <aop:pointcut id="calculatorPoint"
                          expression="execution(*
com.atguigu.component.CalculatorPureImpl.*(..))"/>
        </aop:aspect>
    </aop:config>
</beans>
```

### 3. 测试

```
package com.atguigu;

import com.atguigu.component.Calculator;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * 包名:com.atguigu
 *
 * @author Leevi
 * 日期2021-09-01  11:34
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-application.xml")
public class TestAop {
    @Autowired
    private Calculator calculator;
```

```
    @Test
    public void testAdd(){
        //调用CalculatorPureImpl对象的add()方法
        System.out.println("调用完目标方法之后获取返回值是:"+calculator.sub(5, 3));
    }
}
```

## 第四节 AOP总结

目标:

1. 将目标方法中的非核心业务抽取出来制作成通知
2. 在调用目标方法的核心业务的时候，底层动态代理自动将非核心业务套在核心业务上执行

实现目标:

1. 具备一双慧眼:能识别出来哪里可以抽取
2. 准备一个切面类:
   2.1 IOC: @Component
   2.2 让它成为切面类: @Aspect
3. 将抽取出来的代码封装成方法(通知)，方法放在切面类中
4. 让切面类中的方法成为通知
    1. 前置通知: Before
    2. 返回通知:AfterReturning
    3. 异常通知:AfterThrowing
    4. 后置通知:After
    5. 环绕通知Around
5. 指定通知的作用位置(切入点): 就是引用切入点
    1. 如果切入点和和通知在同一个类中：根据方法名引用
    2. 如果切入点跟通知不在同一个类中: 根据类的全限定名.方法名来引用
6. 声明切入点以及切入点的语法