12. Spring AOP

本节目标

1. 了解AOP的概念
2. 学习Spring AOP的实现方式以及实现原理, 对代理模式有一定了解

1. AOP概述

学习完Spring的统一功能之后, 我们进入到AOP的学习. AOP是Spring框架的第二大核心(第一大核心是IoC)

什么是AOP?

- Aspect Oriented Programming（面向切面编程）

什么是面向切面编程呢? 切面就是指某一类特定问题, 所以AOP也可以理解为面向特定方法编程.

什么是面向特定方法编程呢? 比如上个章节学习的"登录校验", 就是一类特定问题. 登录校验拦截器, 就是对"登录校验"这类问题的统一处理. 所以, 拦截器也是AOP的一种应用. AOP是一种思想, 拦截器是AOP思想的一种实现. Spring框架实现了这种思想, 提供了拦截器技术的相关接口.

同样的, 统一数据返回格式和统一异常处理, 也是AOP思想的一种实现.

简单来说: **AOP是一种思想, 是对某一类事情的集中处理**.

什么是Spring AOP?

AOP是一种思想, 它的实现方法有很多, 有Spring AOP，也有AspectJ、CGLIB等.

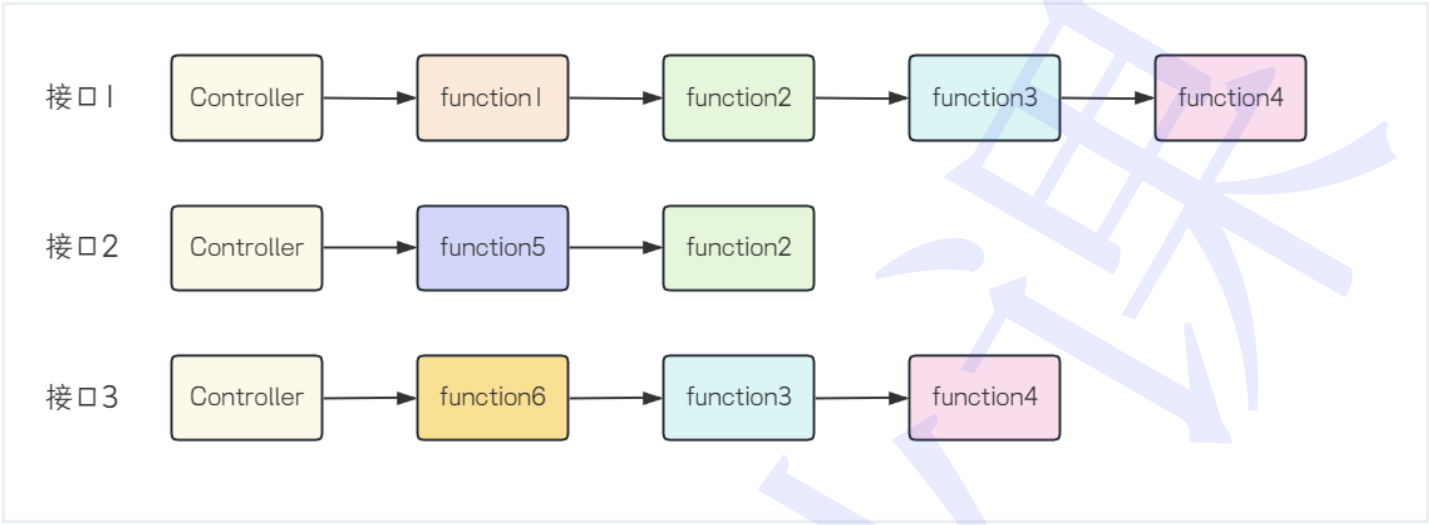
Spring AOP是其中的一种实现方式.

学会了统一功能之后, 是不是就学会了Spring AOP呢, 当然不是.

拦截器作用的维度是URL(一次请求和响应), `@ControllerAdvice` 应用场景主要是全局异常处理(配合自定义异常效果更佳), 数据绑定, 数据预处理. AOP作用的维度更加细致(可以根据包、类、方法名、参数等进行拦截), 能够实现更加复杂的业务逻辑.

举个例子:

我们现在有一个项目, 项目中开发了很多的业务功能



现在有一些业务的执行效率比较低, 耗时较长, 我们需要对接口进行优化.

第一步就需要定位出执行耗时比较长的业务方法, 再针对该业务方法来进行优化

如何定位呢? 我们就需要统计当前项目中每一个业务方法的执行耗时.

如何统计呢? 可以在业务方法运行前和运行后, 记录下方法的开始时间和结束时间, 两者之差就是这个方法的耗时.

```java
public void function1(){
    test1();
}
```

```java
public void function1() {
    long startTime = System.currentTimeMillis();//记录开始时间
    test1();
    long endTime = System.currentTimeMillis();//记录结束时间
    log.info("function1执行耗时:" + (endTime - startTime) + " ms");//记录方法执行耗时
}
```

这种方法是可以解决问题的, 但一个项目中会包含很多业务模块, 每个业务模块又有很多接口, 一个接口又包含很多方法, 如果我们要在每个业务方法中都记录方法的耗时, 对于程序员而言, 会增加很多的工作量.

AOP就可以做到在不改动这些原始方法的基础上, 针对特定的方法进行功能的增强.

> AOP的作用：在程序运行期间在不修改源代码的基础上对已有方法进行增强（无侵入性: 解耦）

接下来我们来看Spring AOP如何来实现

2. Spring AOP快速入门

学习什么是AOP后, 我们先通过下面的程序体验下AOP的开发, 并掌握Spring中AOP的开发步骤.

需求: 统计图书系统各个接口方法的执行时间.

2.1 引入AOP依赖

在pom.xml文件中添加配置

```xml
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

2.2 编写AOP程序

记录Controller中每个方法的执行时间

```java
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Slf4j
@Aspect
@Component
public class TimeAspect {
    /**
     * 记录方法耗时
     */
    @Around("execution(* com.example.demo.controller.*.*(..))")
    public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
        //记录方法执行开始时间
        long begin = System.currentTimeMillis();
        //执行原始方法
        Object result = pjp.proceed();
        //记录方法执行结束时间
        long end = System.currentTimeMillis();
        //记录方法执行耗时
        log.info(pjp.getSignature() + "执行耗时: {}ms", end - begin);
        return result;
    }
}
```

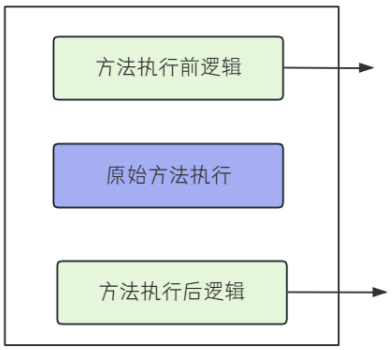
运行程序, 观察日志

```
11:31:03.409 com.example.demo.config.LoginInterceptor preHandle [http-nio-8080-exec-7] 执行拦截器LoginInterceptor...
11:31:03.410 com.example.demo.controller.BookController queryBookById [http-nio-8080-exec-7] 查询图书信息queryBookById, Id:30
11:31:03.413 com.example.demo.aspect.TimeAspect recordTime [http-nio-8080-exec-7] BookInfo com.example.demo.controller.BookController.queryBookById(Integer)执行耗时: 3 ms
11:31:05.632 com.example.demo.config.LoginInterceptor preHandle [http-nio-8080-exec-8] 执行拦截器LoginInterceptor...
11:31:05.632 com.example.demo.controller.BookController queryBookById [http-nio-8080-exec-8] 查询图书信息queryBookById, Id:28
11:31:05.634 com.example.demo.aspect.TimeAspect recordTime [http-nio-8080-exec-8] BookInfo com.example.demo.controller.BookController.queryBookById(Integer)执行耗时: 2 ms
11:31:07.440 com.example.demo.config.LoginInterceptor preHandle [http-nio-8080-exec-9] 执行拦截器LoginInterceptor...
11:31:07.440 com.example.demo.controller.BookController queryBookById [http-nio-8080-exec-9] 查询图书信息queryBookById, Id:17
11:31:07.443 com.example.demo.aspect.TimeAspect recordTime [http-nio-8080-exec-9] BookInfo com.example.demo.controller.BookController.queryBookById(Integer)执行耗时: 3 ms
```

对程序进行简单的讲解:

1. @Aspect: 标识这是一个切面类
2. @Around: 环绕通知, 在目标方法的前后都会被执行. 后面的表达式表示对哪些方法进行增强.
3. ProceedingJoinPoint.proceed() 让原始方法执行

整个代码划分为三部分




```java
public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
    //记录方法执行开始时间
    long begin = System.currentTimeMillis();
    //执行原始方法
    Object result = pjp.proceed();
    //记录方法执行结束时间
    long end = System.currentTimeMillis();
    //记录方法执行耗时
    log.info(pjp.getSignature() + "执行耗时: {} ms", end - begin);
    return result;
}
```

我们通过AOP入门程序完成了业务接口执行耗时的统计.

通过上面的程序, 我们也可以感受到AOP面向切面编程的一些优势:

- 代码无侵入: 不修改原始的业务方法, 就可以对原始的业务方法进行了功能的增强或者是功能的改变
- 减少了重复代码
- 提高开发效率
- 维护方便

3. Spring AOP 详解

下面我们再来详细学习AOP, 主要是以下几部分

- Spring AOP中涉及的核心概念
- Spring AOP通知类型
- 多个AOP程序的执行顺序

3.1 Spring AOP核心概念

3.1.1 切点(Pointcut)

切点(Pointcut), 也称之为"切入点"

Pointcut 的作用就是提供**一组规则** (使用 AspectJ pointcut expression language 来描述), 告诉程序对哪些方法来进行功能增强.

```
@Around("execution(* com.example.demo.controller.*.*(..))")
public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
    //记录方法执行开始时间
    long begin = System.currentTimeMillis();
    //执行原始方法
    Object result = pjp.proceed();
    //记录方法执行结束时间
    long end = System.currentTimeMillis();
    //记录方法执行耗时
    log.info(pjp.getSignature() + "执行耗时: {} ms", end - begin);
    return result;
}
```

上面的表达式 `execution(* com.example.demo.controller.*.*(..))` 就是切点表达式.

3.1.2 连接点(Join Point)

满足切点表达式规则的方法, 就是连接点. 也就是可以被AOP控制的方法

以入门程序举例, 所有 `com.example.demo.controller` 路径下的方法, 都是连接点.

```
1  package com.example.demo.controller;
2
3  @RequestMapping("/book")
4  @RestController
5  public class BookController {
6
7      @RequestMapping("/addBook")
8      public Result addBook(BookInfo bookInfo) {
9          //...代码省略
10     }
11
12     @RequestMapping("/queryBookById")
13     public BookInfo queryBookById(Integer bookId){
14         //...代码省略
```

```
15      }
16
17      @RequestMapping("/updateBook")
18      public Result updateBook(BookInfo bookInfo) {
19          //...代码省略
20      }
21  }
```

上述BookController 中的方法都是连接点

> 切点和连接点的关系
>
> 连接点是满足切点表达式的元素. 切点可以看做是保存了众多连接点的一个集合.
>
> 比如:
>
> 切点表达式: 比特全体教师
>
> 连接点就是: 张三,李四等各个老师

3.1.3 通知(Advice)

通知就是具体要做的工作, 指哪些重复的逻辑，也就是共性功能(最终体现为一个方法)

比如上述程序中记录业务方法的耗时时间, 就是通知.

```
@Around("execution(* com.example.demo.controller.*.*(..))")
public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
    //记录方法执行开始时间
    long begin = System.currentTimeMillis();
    //执行原始方法
    Object result = pjp.proceed();
    //记录方法执行结束时间
    long end = System.currentTimeMillis();
    //记录方法执行耗时
    log.info(pjp.getSignature() + "执行耗时: {} ms", end - begin);
    return result;
}
```

在AOP面向切面编程当中, 我们把这部分重复的代码逻辑抽取出来单独定义, 这部分代码就是通知的内容.

3.1.4 切面(Aspect)

切面(Aspect) = 切点(Pointcut) + 通知(Advice)

通过切面就能够描述当前AOP程序需要针对于哪些方法, 在什么时候执行什么样的操作.

切面既包含了通知逻辑的定义, 也包括了连接点的定义.

```
public class TimeAspect {
    /**
     * 记录方法耗时
     */
    @Around("execution(* com.example.demo.controller.*.*(..))")
    public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
        //记录方法执行开始时间
        long begin = System.currentTimeMillis();
        //执行原始方法
        Object result = pjp.proceed();
        //记录方法执行结束时间
        long end = System.currentTimeMillis();
        //记录方法执行耗时
        log.info(pjp.getSignature() + "执行耗时: {} ms", end - begin);
        return result;
    }
}
```

切面

切面所在的类, 我们一般称为**切面类**(被@Aspect注解标识的类)

3.2 通知类型

上面我们讲了什么是通知, 接下来学习通知的类型. `@Around` 就是其中一种通知类型, 表示环绕通知.

Spring中AOP的通知类型有以下几种:

- @Around: 环绕通知, 此注解标注的通知方法在目标方法前, 后都被执行
- @Before: 前置通知, 此注解标注的通知方法在目标方法前被执行
- @After: 后置通知, 此注解标注的通知方法在目标方法后被执行, 无论是否有异常都会执行
- @AfterReturning: 返回后通知, 此注解标注的通知方法在目标方法后被执行, 有异常不会执行
- @AfterThrowing: 异常后通知, 此注解标注的通知方法发生异常后执行

接下来我们通过代码来加深对这几个通知的理解:

> 为方便学习, 我们可以新建一个项目

```
1 import lombok.extern.slf4j.Slf4j;
2 import org.aspectj.lang.ProceedingJoinPoint;
3 import org.aspectj.lang.annotation.*;
4 import org.springframework.stereotype.Component;
```

```java

@Slf4j
@Aspect
@Component
public class AspectDemo {
    //前置通知
    @Before("execution(* com.example.demo.controller.*.*(..))")
    public void doBefore() {
        log.info("执行 Before 方法");
    }

    //后置通知
    @After("execution(* com.example.demo.controller.*.*(..))")
    public void doAfter() {
        log.info("执行 After 方法");
    }

    //返回后通知
    @AfterReturning("execution(* com.example.demo.controller.*.*(..))")
    public void doAfterReturning() {
        log.info("执行 AfterReturning 方法");
    }

    //抛出异常后通知
    @AfterThrowing("execution(* com.example.demo.controller.*.*(..))")
    public void doAfterThrowing() {
        log.info("执行 doAfterThrowing 方法");
    }

    //添加环绕通知
    @Around("execution(* com.example.demo.controller.*.*(..))")
    public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("Around 方法开始执行");
        Object result = joinPoint.proceed();
        log.info("Around 方法结束执行");
        return result;
    }
}
```

写一些测试程序:

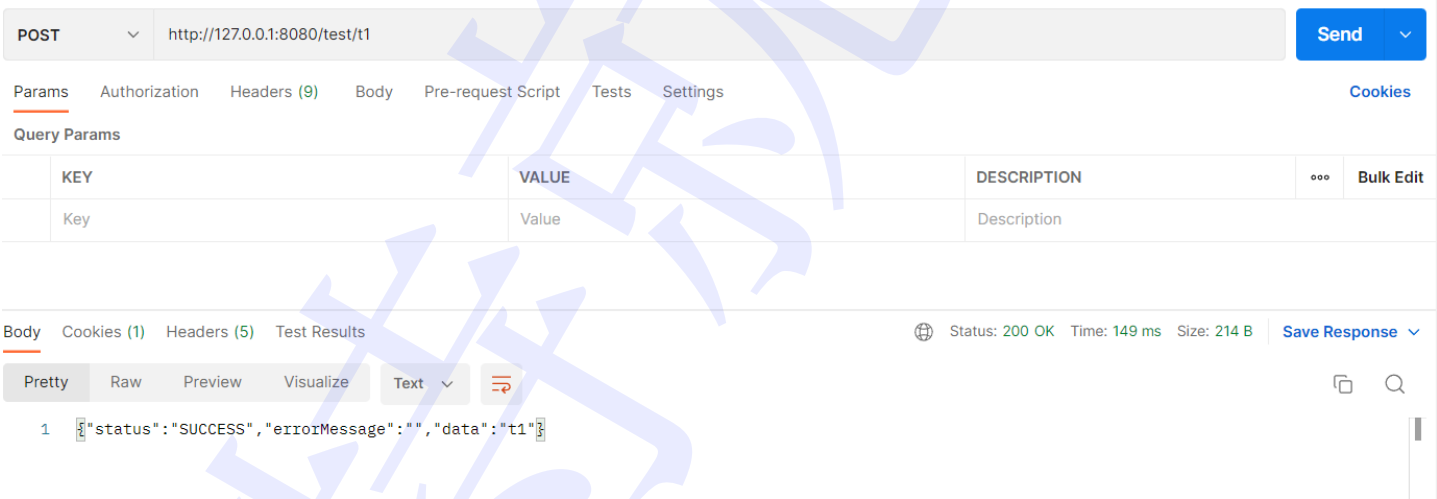
```java
package com.example.demo.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```java

@RequestMapping("/test")
@RestController
public class TestController {
    @RequestMapping("/t1")
    public String t1() {
        return "t1";
    }

    @RequestMapping("/t2")
    public boolean t2() {
        int a = 10 / 0;
        return true;
    }
}
```

运行程序, 观察日志:

1. 正常运行的情况

http://127.0.0.1:8080/test/t1



观察日志

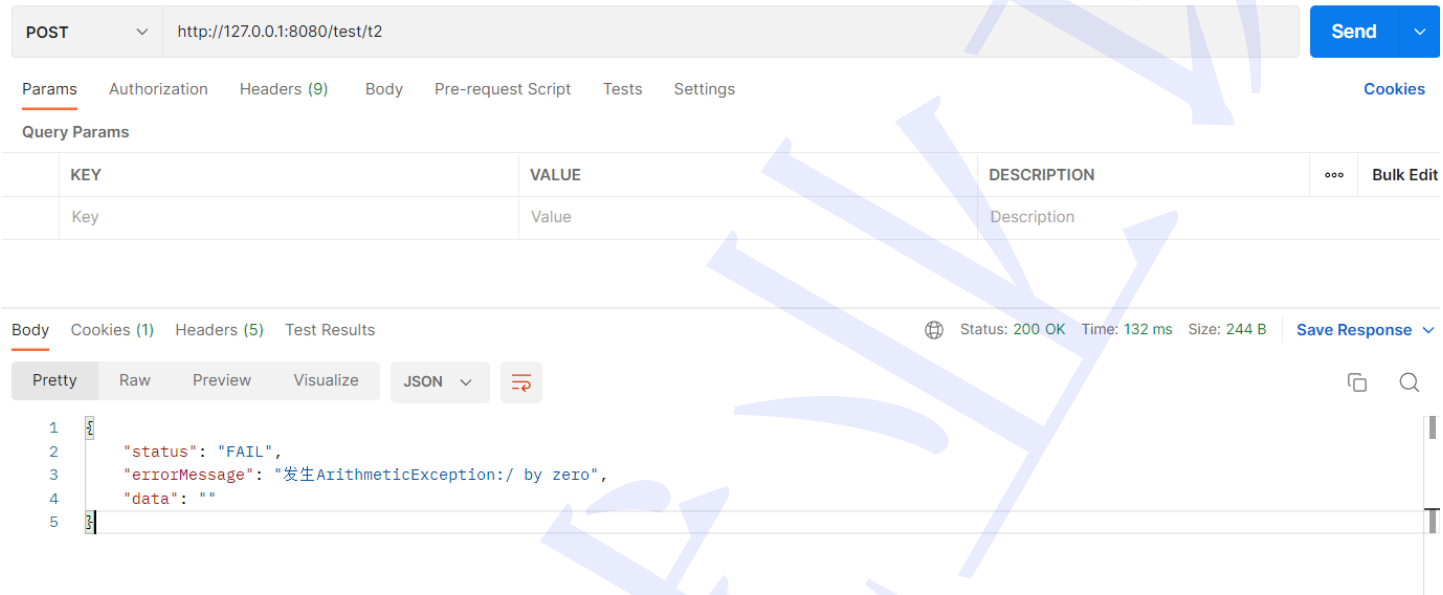
```
15:17:10.978 com.example.demo.aspect.AspectDemo doAround [http-nio-8080-exec-1] Around 方法开始执行
15:17:10.978 com.example.demo.aspect.AspectDemo doBefore [http-nio-8080-exec-1] 执行 Before 方法
15:17:10.983 com.example.demo.aspect.AspectDemo doAfterReturning [http-nio-8080-exec-1] 执行 AfterReturning 方法
15:17:10.983 com.example.demo.aspect.AspectDemo doAfter [http-nio-8080-exec-1] 执行 After 方法
15:17:10.983 com.example.demo.aspect.AspectDemo doAround [http-nio-8080-exec-1] Around 方法结束执行
```

程序正常运行的情况下, `@AfterThrowing` 标识的通知方法不会执行

从上图也可以看出来, `@Around` 标识的通知方法包含两部分, 一个"前置逻辑", 一个"后置逻辑".其中"前置逻辑" 会先于 `@Before` 标识的通知方法执行, "后置逻辑" 会晚于 `@After` 标识的通知方法执行

2. 异常时的情况

http://127.0.0.1:8080/test/t2

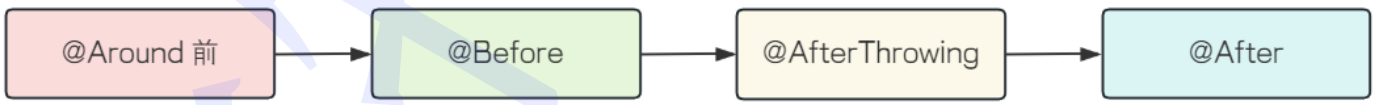


观察日志:

```
15:27:56.419 com.example.demo.aspect.AspectDemo doAround [http-nio-8080-exec-5] Around 方法开始执行
15:27:56.419 com.example.demo.aspect.AspectDemo doBefore [http-nio-8080-exec-5] 执行 Before 方法
15:27:56.420 com.example.demo.aspect.AspectDemo doAfterThrowing [http-nio-8080-exec-5] 执行 doAfterThrowing 方法
15:27:56.421 com.example.demo.aspect.AspectDemo doAfter [http-nio-8080-exec-5] 执行 After 方法
```

程序发生异常的情况下:

- `@AfterReturning` 标识的通知方法不会执行， `@AfterThrowing` 标识的通知方法执行了
- `@Around` 环绕通知中原始方法调用时有异常，通知中的环绕后的代码逻辑也不会在执行了(因为原始方法调用出异常了)



注意事项:

- `@Around` 环绕通知需要调用 `ProceedingJoinPoint.proceed()` 来让原始方法执行, 其他通知不需要考虑目标方法执行.
- `@Around` 环绕通知方法的返回值, 必须指定为Object, 来接收原始方法的返回值, 否则原始方法执行完毕, 是获取不到返回值的.
- 一个切面类可以有多个切点.

3.3 @PointCut

上面代码存在一个问题,就是存在大量重复的切点表达式 `execution(* com.example.demo.controller.*.*(..))` ,Spring提供了 `@PointCut` 注解,把公共的切点表达式提取出来,需要用到时引用该切入点表达式即可.

上述代码就可以修改为:

```java
@Slf4j
@Aspect
@Component
public class AspectDemo {
    //定义切点(公共的切点表达式)
    @Pointcut("execution(* com.example.demo.controller.*.*(..))")
    private void pt(){}
    //前置通知
    @Before("pt()")
    public void doBefore() {
        //...代码省略
    }

    //后置通知
    @After("pt()")
    public void doAfter() {
        //...代码省略
    }

    //返回后通知
    @AfterReturning("pt()")
    public void doAfterReturning() {
        //...代码省略
    }

    //抛出异常后通知
    @AfterThrowing("pt()")
    public void doAfterThrowing() {
        //...代码省略
    }

    //添加环绕通知
    @Around("pt()")
    public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable {
        //...代码省略
    }
}
```

当切点定义使用private修饰时, 仅能在当前切面类中使用, 当其他切面类也要使用当前切点定义时, 就需要把private改为public. 引用方式为: 全限定类名.方法名()

```
@Slf4j
@Aspect
@Component
public class AspectDemo2 {
    //前置通知
    @Before("com.example.demo.aspect.AspectDemo.pt()")
    public void doBefore() {
        log.info("执行 AspectDemo2 -> Before 方法");
    }
}
```

3.4 切面优先级 @Order

当我们在一个项目中, 定义了多个切面类时, 并且这些切面类的多个切入点都匹配到了同一个目标方法.

当目标方法运行的时候, 这些切面类中的通知方法都会执行, 那么这几个通知方法的执行顺序是什么样的呢?

我们还是通过程序来求证:

定义多个切面类:

> 为防止干扰, 我们把AspectDemo这个切面先去掉(把*@Component 注解去掉就可以*)
>
> 为简单化, 只写了 `@Before` 和 `@After` 两个通知

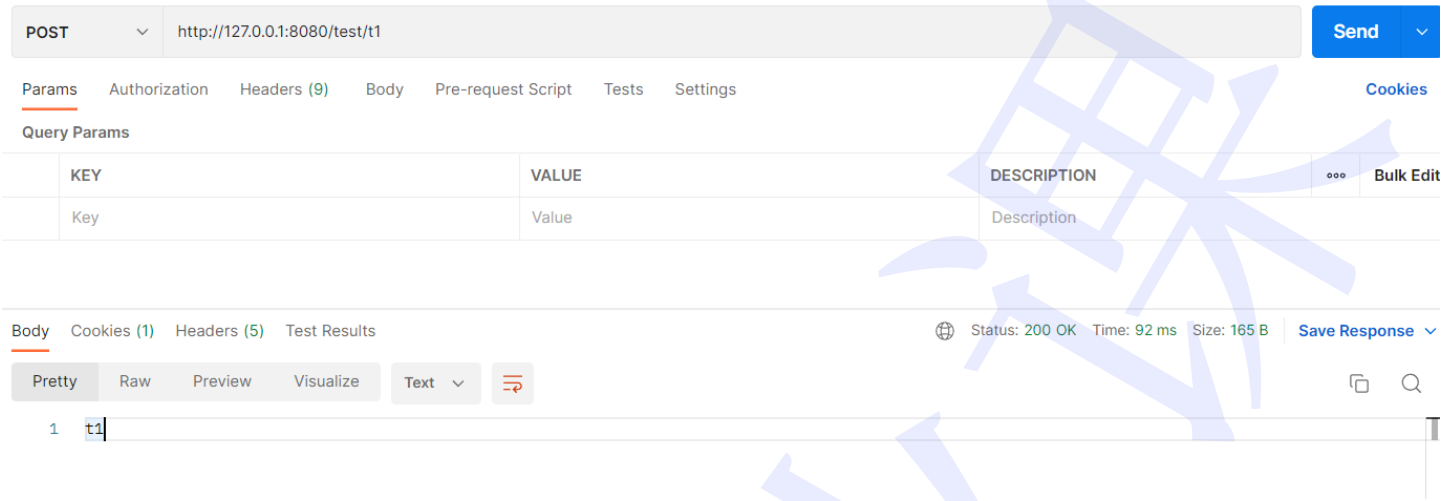
```
@Component
public class AspectDemo2 {
    @Pointcut("execution(* com.example.demo.controller.*.*(..))")
    private void pt(){}

    //前置通知
    @Before("pt()")
    public void doBefore() {
        log.info("执行 AspectDemo2 -> Before 方法");
    }

    //后置通知
    @After("pt()")
    public void doAfter() {
        log.info("执行 AspectDemo2 -> After 方法");
    }
}
```

```java
@Component
public class AspectDemo3 {
    @Pointcut("execution(* com.example.demo.controller.*.*(..))")
    private void pt(){}

    //前置通知
    @Before("pt()")
    public void doBefore() {
        log.info("执行 AspectDemo3 -> Before 方法");
    }

    //后置通知
    @After("pt()")
    public void doAfter() {
        log.info("执行 AspectDemo3 -> After 方法");
    }
}
```

```java
@Component
public class AspectDemo4 {
    @Pointcut("execution(* com.example.demo.controller.*.*(..))")
    private void pt(){}

    //前置通知
    @Before("pt()")
    public void doBefore() {
        log.info("执行 AspectDemo4 -> Before 方法");
    }

    //后置通知
    @After("pt()")
    public void doAfter() {
        log.info("执行 AspectDemo4 -> After 方法");
    }
}
```

运行程序, 访问接口:

http://127.0.0.1:8080/test/t1



观察日志:

```
16:31:27.628 com.example.demo.aspect.AspectDemo2 doBefore [http-nio-8080-exec-1] 执行 AspectDemo2 -> Before 方法
16:31:27.628 com.example.demo.aspect.AspectDemo3 doBefore [http-nio-8080-exec-1] 执行 AspectDemo3 -> Before 方法
16:31:27.628 com.example.demo.aspect.AspectDemo4 doBefore [http-nio-8080-exec-1] 执行 AspectDemo4 -> Before 方法
16:31:27.636 com.example.demo.aspect.AspectDemo4 doAfter [http-nio-8080-exec-1] 执行 AspectDemo4 -> After 方法
16:31:27.636 com.example.demo.aspect.AspectDemo3 doAfter [http-nio-8080-exec-1] 执行 AspectDemo3 -> After 方法
16:31:27.636 com.example.demo.aspect.AspectDemo2 doAfter [http-nio-8080-exec-1] 执行 AspectDemo2 -> After 方法
```

通过上述程序的运行结果, 可以看出:

存在多个切面类时, 默认按照切面类的类名字母排序:

- `@Before` 通知：字母排名靠前的先执行
- `@After` 通知：字母排名靠前的后执行

但这种方式不方便管理, 我们的类名更多还是具备一定含义的.

Spring 给我们提供了一个新的注解, 来控制这些切面通知的执行顺序: `@Order`

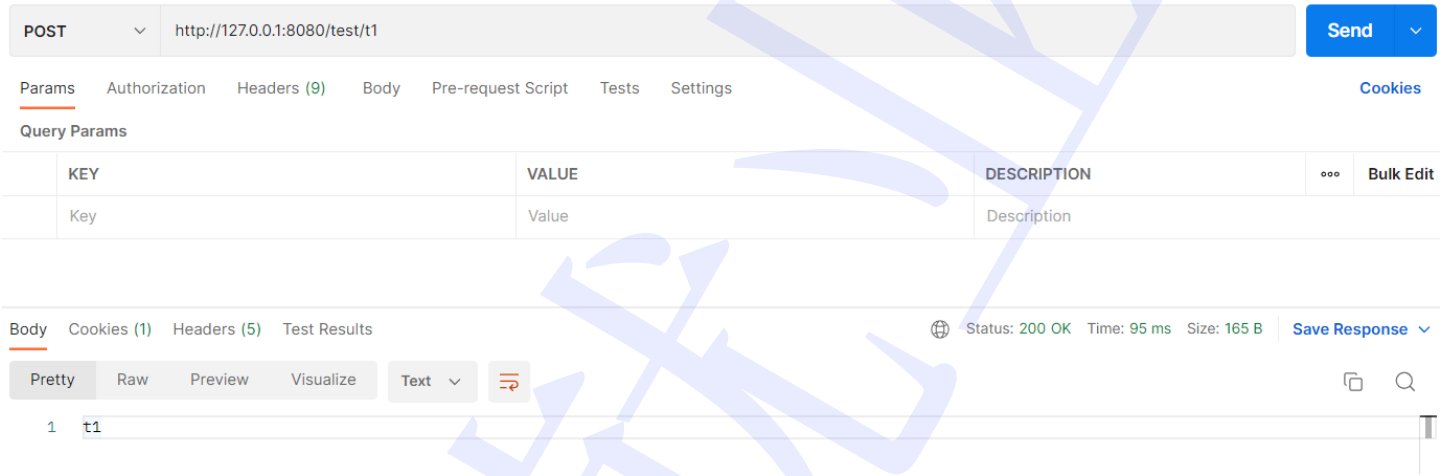
使用方式如下:

```java
@Aspect
@Component
@Order(2)
public class AspectDemo2 {
    //...代码省略
}
```

```java
@Aspect
@Component
@Order(1)
public class AspectDemo3 {
    //...代码省略
}
```

```java
@Aspect
@Component
@Order(3)
public class AspectDemo4 {
    //...代码省略
}
```

重新运行程序, 访问接口 http://127.0.0.1:8080/test/t1



观察日志:

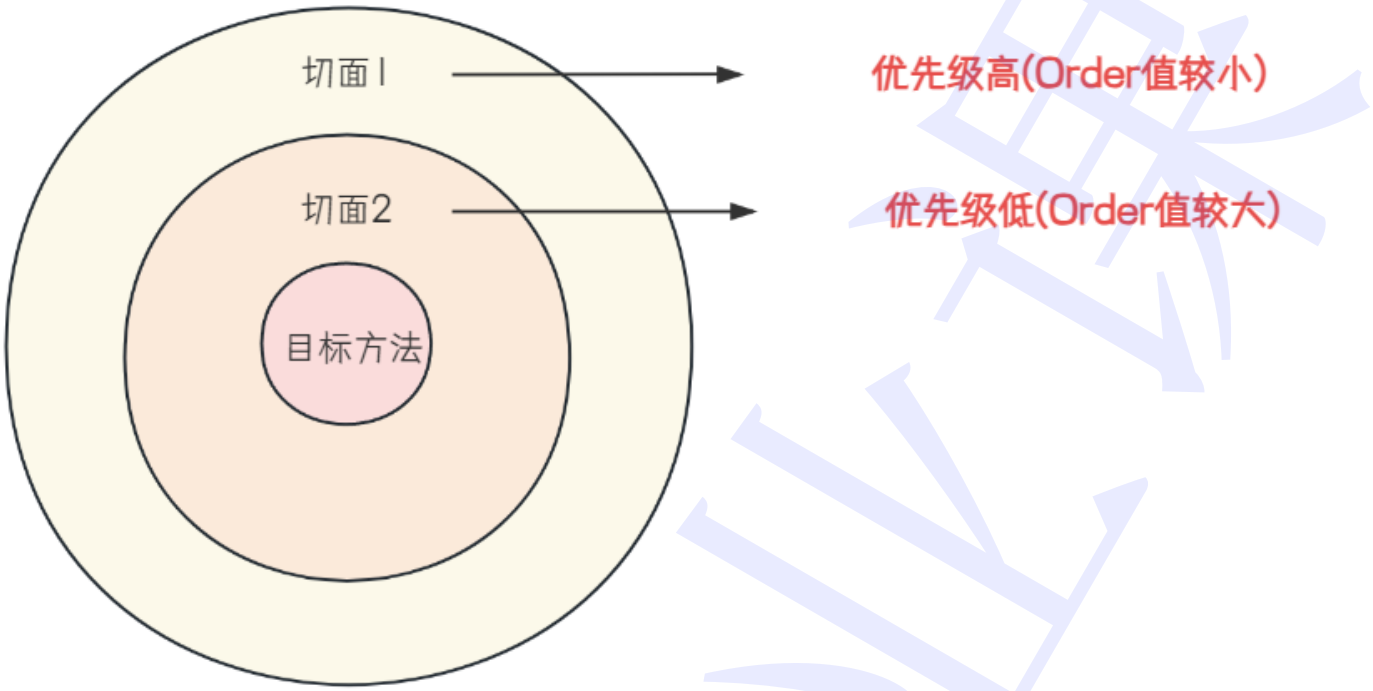
```
16:39:37.633 com.example.demo.aspect.AspectDemo3 doBefore [http-nio-8080-exec-2] 执行 AspectDemo3 -> Before 方法
16:39:37.633 com.example.demo.aspect.AspectDemo2 doBefore [http-nio-8080-exec-2] 执行 AspectDemo2 -> Before 方法
16:39:37.633 com.example.demo.aspect.AspectDemo4 doBefore [http-nio-8080-exec-2] 执行 AspectDemo4 -> Before 方法
16:39:37.641 com.example.demo.aspect.AspectDemo4 doAfter [http-nio-8080-exec-2] 执行 AspectDemo4 -> After 方法
16:39:37.642 com.example.demo.aspect.AspectDemo2 doAfter [http-nio-8080-exec-2] 执行 AspectDemo2 -> After 方法
16:39:37.642 com.example.demo.aspect.AspectDemo3 doAfter [http-nio-8080-exec-2] 执行 AspectDemo3 -> After 方法
```

通过上述程序的运行结果, 得出结论:

@Order 注解标识的切面类, 执行顺序如下:

- `@Before` 通知：数字越小先执行
- `@After` 通知：数字越大先执行

`@Order` 控制切面的**优先级**, 先执行优先级较高的切面, 再执行优先级较低的切面, 最终执行目标方法.



3.5 切点表达式

上面的代码中, 我们一直在使用切点表达式来描述切点. 下面我们来介绍一下切点表达式的语法.

切点表达式常见有两种表达方式

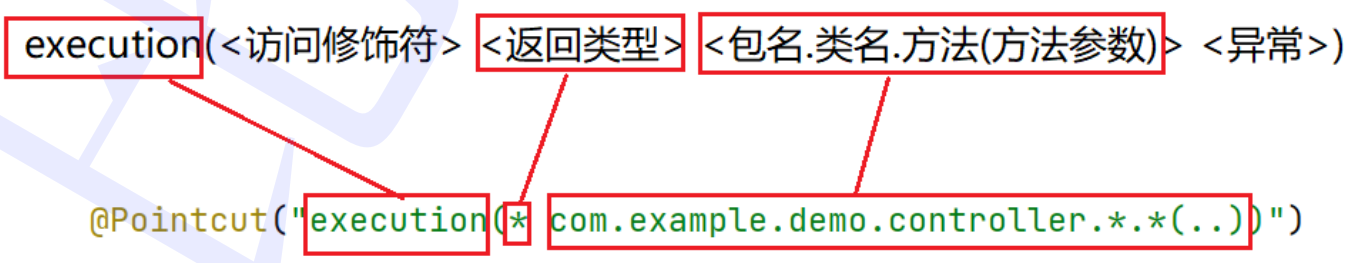
1. execution(……): 根据方法的签名来匹配
2. @annotation(……) : 根据注解匹配

3.5.1 execution表达式

execution() 是最常用的切点表达式, 用来匹配方法, 语法为:

```
execution(<访问修饰符> <返回类型> <包名.类名.方法(方法参数)> <异常>)
```

其中: 访问修饰符和异常可以省略



切点表达式支持通配符表达:

1. `*` ： 匹配任意字符，只匹配一个元素(返回类型, 包, 类名, 方法或者方法参数)
   a. 包名使用 `*` 表示任意包(一层包使用一个*)
   b. 类名使用 `*` 表示任意类
   c. 返回值使用 `*` 表示任意返回值类型
   d. 方法名使用 `*` 表示任意方法
   e. 参数使用 `*` 表示一个任意类型的参数
2. `..` ： 匹配多个连续的任意符号, 可以通配任意层级的包, 或任意类型, 任意个数的参数
   a. 使用 `..` 配置包名，标识此包以及此包下的所有子包
   b. 可以使用 `..` 配置参数，任意个任意类型的参数

切点表达式示例

TestController 下的 public修饰, 返回类型为String 方法名为t1, 无参方法

```
1 execution(public String com.example.demo.controller.TestController.t1())
```

省略访问修饰符

```
1 execution(String com.example.demo.controller.TestController.t1())
```

匹配所有返回类型

```
1 execution(* com.example.demo.controller.TestController.t1())
```

匹配TestController 下的所有无参方法

```
1 execution(* com.example.demo.controller.TestController.*())
```

匹配TestController 下的所有方法

```
execution(* com.example.demo.controller.TestController.*(..))
```

匹配controller包下所有的类的所有方法

```
execution(* com.example.demo.controller.*.*(..))
```

匹配所有包下面的TestController

```
execution(* com..TestController.*(..))
```

匹配com.example.demo包下, 子孙包下的所有类的所有方法

```
execution(* com.example.demo..*(..))
```

3.5.2 @annotation

execution表达式更适用有规则的, 如果我们要匹配多个无规则的方法呢, 比如：TestController中的t1()和UserController中的u1()这两个方法.

这个时候我们使用execution这种切点表达式来描述就不是很方便了.

我们可以借助自定义注解的方式以及另一种切点表达式 `@annotation` 来描述这一类的切点

实现步骤：

1. 编写自定义注解
2. 使用 `@annotation` 表达式来描述切点
3. 在连接点的方法上添加自定义注解

准备测试代码:

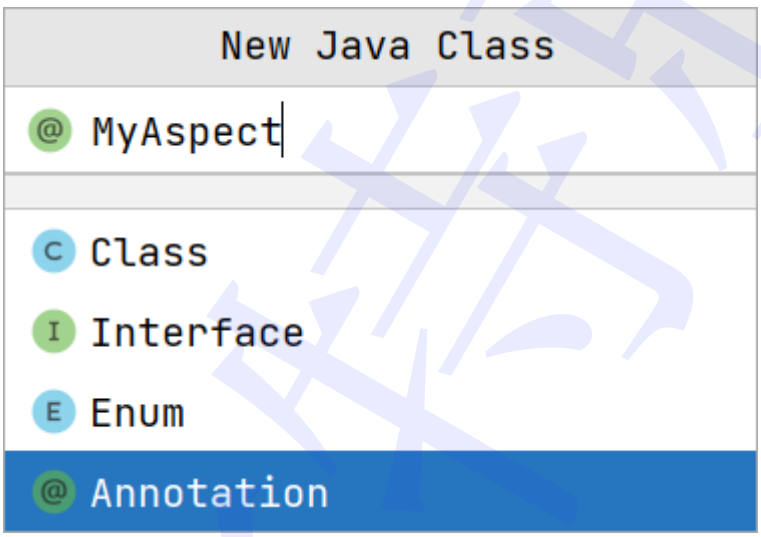
```
@RequestMapping("/test")
@RestController
public class TestController {
    @RequestMapping("/t1")
    public String t1() {
        return "t1";
    }

```

```
    @RequestMapping("/t2")
    public boolean t2() {
        return true;
    }
}
```

```
@RequestMapping("/user")
@RestController
public class UserController {
    @RequestMapping("/u1")
    public String u1(){
        return "u1";
    }
    @RequestMapping("/u2")
    public String u2(){
        return "u2";
    }
}
```

3.5.2.1 自定义注解 `@MyAspect`

创建一个注解类(和创建Class文件一样的流程, 选择Annotation就可以了)




```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAspect {
```

```
9
10 }
```

代码简单说明, 了解即可. 不做过多解释

1. `@Target` 标识了 `Annotation` 所修饰的对象范围, 即该注解可以用在什么地方.

   > 常用取值:
   >
   > ElementType.*TYPE*: 用于描述类、接口(包括注解类型) 或enum声明
   >
   > ElementType.METHOD: 描述方法
   >
   > ElementType.PARAMETER: 描述参数
   >
   > *ElementType.*TYPE_USE*: 可以标注任意类型*

2. `@Retention` 指Annotation被保留的时间长短, 标明注解的生命周期

   > `@Retention` 的取值有三种:
   >
   > 1. RetentionPolicy.SOURCE：表示注解仅存在于源代码中, 编译成字节码后会被丢弃. 这意味着在运行时无法获取到该注解的信息, 只能在编译时使用. 比如 `@SuppressWarnings` , 以及lombok提供的注解 `@Data` , `@Slf4j`
   > 2. RetentionPolicy.CLASS：编译时注解. 表示注解存在于源代码和字节码中, 但在运行时会被丢弃. 这意味着在编译时和字节码中可以通过反射获取到该注解的信息, 但在实际运行时无法获取. 通常用于一些框架和工具的注解.
   > 3. RetentionPolicy.RUNTIME：运行时注解. 表示注解存在于源代码, 字节码和运行时中. 这意味着在编译时, 字节码中和实际运行时都可以通过反射获取到该注解的信息. 通常用于一些需要在运行时处理的注解, 如Spring的 `@Controller` `@ResponseBody`

3.5.2.2 切面类

使用 `@annotation` 切点表达式定义切点, 只对 `@MyAspect` 生效

切面类代码如下:

```
1  @Slf4j
2  @Component
3  @Aspect
4  public class MyAspectDemo {
5      //前置通知
6      @Before("@annotation(com.example.demo.aspect.MyAspect)")
7      public void before(){
8          log.info("MyAspect -> before ...");
9      }
10
11     //后置通知
```

```
    @After("@annotation(com.example.demo.aspect.MyAspect)")
    public void after(){
        log.info("MyAspect -> after ...");
    }
}
```

3.5.2.3 添加自定义注解

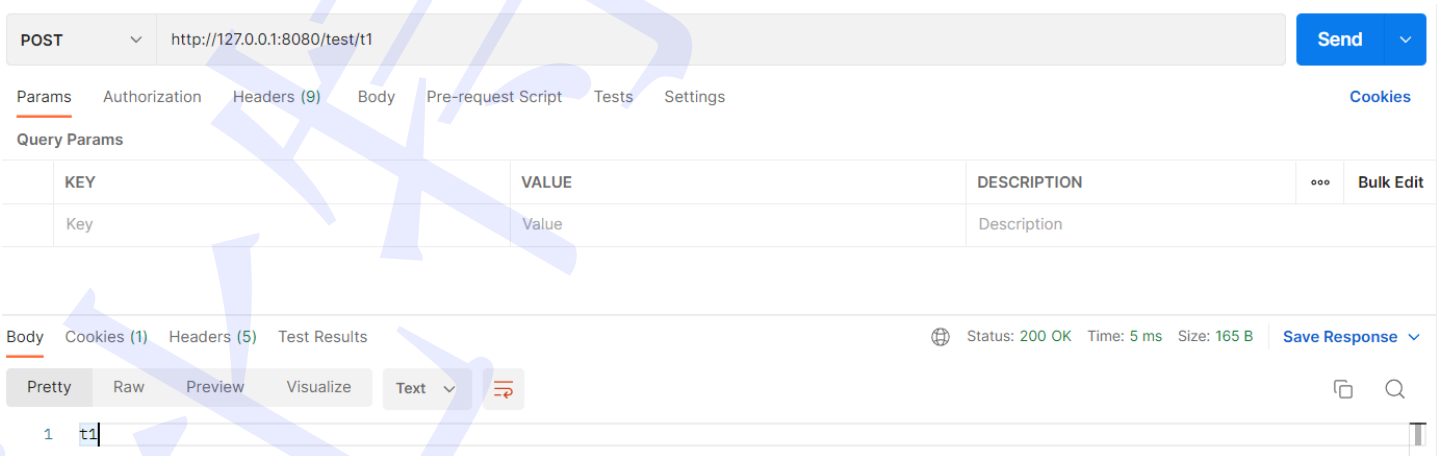
在TestController中的t1()和UserController中的u1()这两个方法上添加自定义注解 `@MyAspect`，其他方法不添加

```
@MyAspect
@RequestMapping("/t1")
public String t1() {
    return "t1";
}
```

```
@MyAspect
@RequestMapping("/u1")
public String u1(){
    return "u1";
}
```

运行程序，测试接口

http://127.0.0.1:8080/test/t1



观察日志：

```
18:54:08.678 com.example.demo.aspect.MyAspectDemo before [http-nio-8080-exec-4] MyAspect -> before ...
18:54:08.678 com.example.demo.aspect.MyAspectDemo after [http-nio-8080-exec-4] MyAspect -> after ...
```

可以看到, 切面通知被执行了.

继续测试:

http://127.0.0.1:8080/test/t2, 切面通知未执行

http://127.0.0.1:8080/user/u1 , 切面通知执行.

> **Spring AOP的实现方式(常见面试题)**
>
> 1. 基于注解 `@Aspect` (参考上述课件内容)
> 2. 基于自定义注解 (参考自定义注解 `@annotation` 部分的内容)
> 3. 基于Spring API (通过xml配置的方式, 自从SpringBoot 广泛使用之后, 这种方法几乎看不到了, 课下自己了解下即可)
> 4. 基于代理来实现(更加久远的一种实现方式, 写法笨重, 不建议使用)
>
> 参考: https://cloud.tencent.com/developer/article/2032268

4. Spring AOP 原理

上面我们主要学习了Spring AOP的应用, 接下来我们来学习Spring AOP的原理, 也就是Spring 是如何实现AOP的.

Spring AOP 是基于**动态代理**来实现AOP的, 咱们学习内容主要分以下两部分

1. 代理模式
2. Spring AOP源码剖析

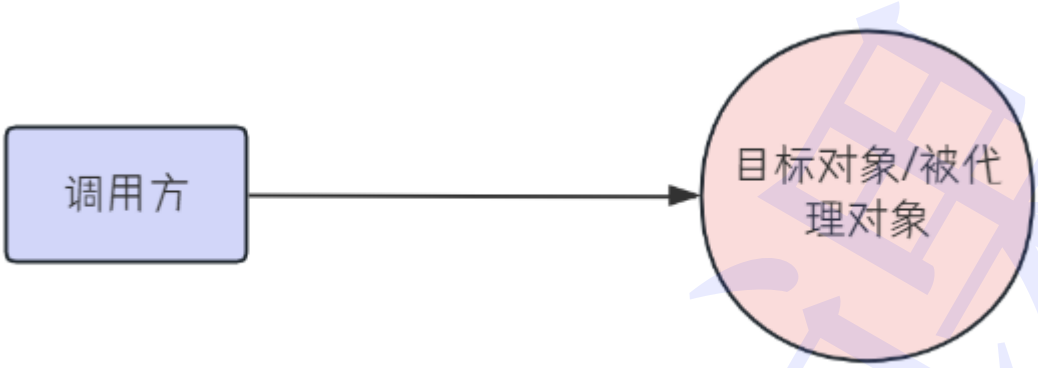
4.1 代理模式

代理模式, 也叫委托模式.

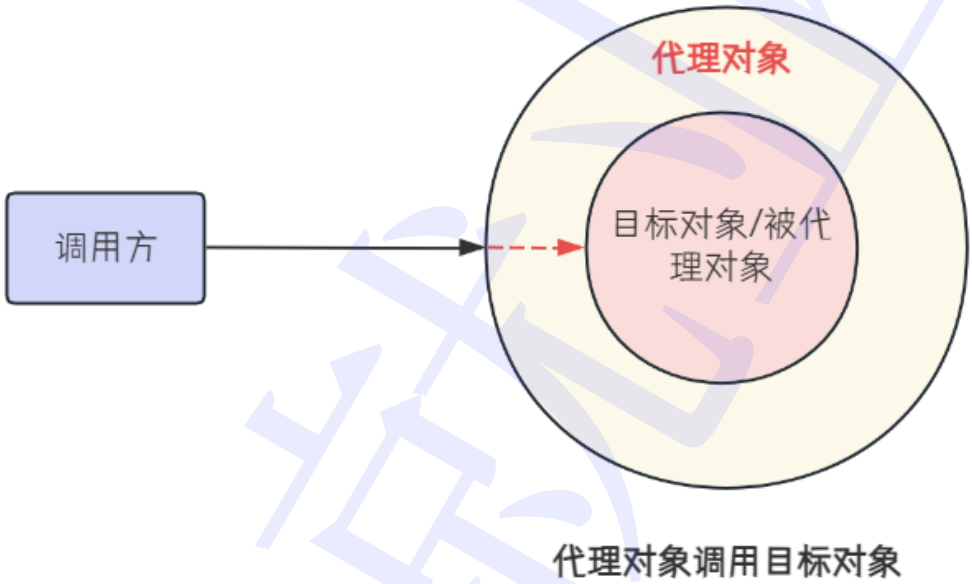
定义：为其他对象提供一种代理以控制对这个对象的访问. 它的作用就是通过提供一个代理类, 让我们在调用目标方法的时候, 不再是直接对目标方法进行调用, 而是通过代理类**间接**调用.

在某些情况下, 一个对象不适合或者不能直接引用另一个对象, 而代理对象可以在客户端和目标对象之间起到中介的作用.

使用代理前:



使用代理后:



代理对象调用目标对象

生活中的代理

- 艺人经纪人: 广告商找艺人拍广告, 需要经过经纪人,由经纪人来和艺人进行沟通.
- 房屋中介: 房屋进行租赁时, 卖方会把房屋授权给中介, 由中介来代理看房, 房屋咨询等服务.
- 经销商: 厂商不直接对外销售产品, 由经销商负责代理销售.
- 秘书/助理: 合作伙伴找老板谈合作, 需要先经过秘书/助理预约.

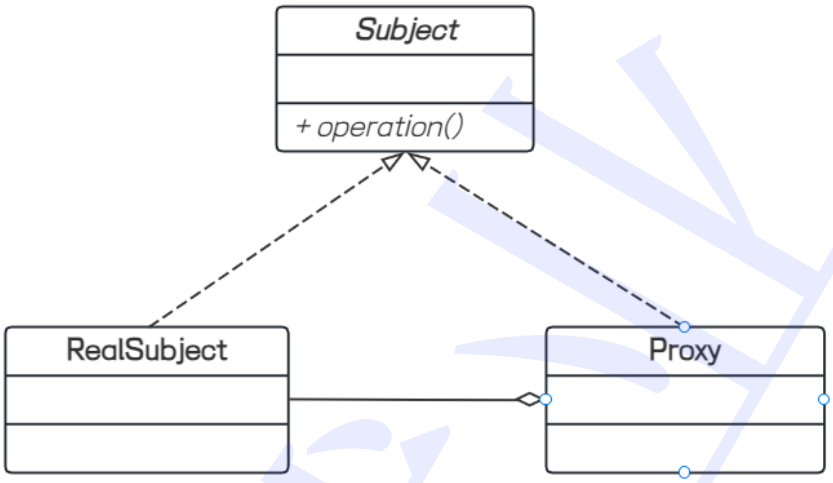
代理模式的主要角色

1. Subject: 业务接口类. 可以是抽象类或者接口(不一定有)
2. RealSubject: 业务实现类. 具体的业务执行, 也就是被代理对象.
3. Proxy: 代理类. RealSubject的代理.

> 比如房屋租赁

> Subject 就是提前定义了房东做的事情, 交给中介代理, 也是中介要做的事情
>
> RealSubject: 房东
>
> Proxy: 中介

UML类图如下:



代理模式可以在不修改被代理对象的基础上, 通过扩展代理类, 进行一些功能的附加与增强.

根据代理的创建时期, 代理模式分为**静态代理**和**动态代理**.

- 静态代理: 由程序员创建代理类或特定工具自动生成源代码再对其编译, 在程序运行前代理类的 .class 文件就已经存在了.
- 动态代理: 在程序运行时, 运用反射机制动态创建而成.

4.1.1 静态代理

静态代理: 在程序运行前, 代理类的 .class文件就已经存在了. (在出租房子之前, 中介已经做好了相关的工作, 就等租户来租房子了)

我们通过代码来加深理解. 以房租租赁为例

1. 定义接口(定义房东要做的事情, 也是中介需要做的事情)

```
1 public interface HouseSubject {
2     void rentHouse();
3 }
```

2. 实现接口(房东出租房子)

```
1 public class RealHouseSubject implements HouseSubject{
```

```
    @Override
    public void rentHouse() {
        System.out.println("我是房东，我出租房子");
    }
}
```

3. 代理(中介, 帮房东出租房子)

```
public class HouseProxy implements HouseSubject{
    //将被代理对象声明为成员变量
    private HouseSubject houseSubject;
    public HouseProxy(HouseSubject houseSubject) {
        this.houseSubject = houseSubject;
    }

    @Override
    public void rentHouse() {
        //开始代理
        System.out.println("我是中介，开始代理");
        //代理房东出租房子
        houseSubject.rentHouse();
        //代理结束
        System.out.println("我是中介，代理结束");
    }
}
```

4. 使用

```
public class StaticMain {
    public static void main(String[] args) {
        HouseSubject subject = new RealHouseSubject();
        //创建代理类
        HouseProxy proxy = new HouseProxy(subject);
        //通过代理类访问目标方法
        proxy.rentHouse();
    }
}
```

运行结果:

```
我是中介，开始代理
我是房东，我出租房子
我是中介，代理结束
```

上面这个代理实现方式就是静态代理(仿佛啥也没干).

从上述程序可以看出, 虽然静态代理也完成了对目标对象的代理, 但是由于代码都写死了, 对目标对象的每个方法的增强都是手动完成的，非常不灵活. 所以日常开发几乎看不到静态代理的场景.

接下来新增需求: 中介又新增了其他业务: 代理房屋出售

我们需要对上述代码进行修改

1. 接口定义修改

```java
public interface HouseSubject {
    void rentHouse();
    void saleHouse();
}
```

2. 接口实现修改

```java
public class RealHouseSubject implements HouseSubject{
    @Override
    public void rentHouse() {
        System.out.println("我是房东，我出租房子");
    }

    @Override
    public void saleHouse() {
        System.out.println("我是房东，我出售房子");
    }
}
```

3. 代理类修改

```java
public class HouseProxy implements HouseSubject{
    //将被代理对象声明为成员变量
    private HouseSubject houseSubject;
```

```
    public HouseProxy(HouseSubject houseSubject) {
        this.houseSubject = houseSubject;
    }

    @Override
    public void rentHouse() {
        //开始代理
        System.out.println("我是中介，开始代理");
        //代理房东出租房子
        houseSubject.rentHouse();
        //代理结束
        System.out.println("我是中介，代理结束");
    }

    @Override
    public void saleHouse() {
        //开始代理
        System.out.println("我是中介，开始代理");
        //代理房东出租房子
        houseSubject.saleHouse();
        //代理结束
        System.out.println("我是中介，代理结束");
    }
}
```

从上述代码可以看出，我们修改接口(Subject)和业务实现类(RealSubject)时，还需要修改代理类(Proxy).

同样的，如果有新增接口(Subject)和业务实现类(RealSubject)，也需要对每一个业务实现类新增代理类(Proxy).

既然代理的流程是一样的，有没有一种办法，让他们通过一个代理类来实现呢？

这就需要用到动态代理技术了.

4.1.2 动态代理

相比于静态代理来说，动态代理更加灵活.

我们不需要针对每个目标对象都单独创建一个代理对象，而是把这个创建代理对象的工作推迟到程序运行时由JVM来实现. 也就是说动态代理在程序运行时，根据需要动态创建生成.

> 比如房屋中介，我不需要提前预测都有哪些业务，而是业务来了我再根据情况创建.

我们还是先看代码再来理解.

Java也对动态代理进行了实现，并给我们提供了一些API，常见的实现方式有两种:

1. JDK动态代理
2. CGLIB动态代理

> 动态代理在我们日常开发中使用的相对较少，但是在框架中几乎是必用的一门技术. 学会了动态代理之后, 对于我们理解和学习各种框架的原理也非常有帮助.

JDK动态代理

JDK 动态代理类实现步骤

1. 定义一个接口及其实现类(静态代理中的 `HouseSubject` 和 `RealHouseSubject`)
2. 自定义 `InvocationHandler` 并重写 `invoke` 方法，在 `invoke` 方法中我们会调用目标方法(被代理类的方法)并自定义一些处理逻辑
3. 通过 `Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)` 方法创建代理对象

定义JDK动态代理类

实现 `InvocationHandler` 接口

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

public class JDKInvocationHandler implements InvocationHandler {
    //目标对象即就是被代理对象
    private Object target;

    public JDKInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 代理增强内容
        System.out.println("我是中介，开始代理");
        //通过反射调用被代理类的方法
        Object retVal = method.invoke(target, args);
        //代理增强内容
        System.out.println("我是中介，代理结束");
        return retVal;
    }
}
```

创建一个代理对象并使用

```
public class DynamicMain {
    public static void main(String[] args) {
        HouseSubject target=  new RealHouseSubject();
        //创建一个代理类：通过被代理类、被代理实现的接口、方法调用处理器来创建
        HouseSubject proxy = (HouseSubject) Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                new Class[]{HouseSubject.class},
                new JDKInvocationHandler(target)
        );
        proxy.rentHouse();
    }
}
```

代码简单讲解

> 主要是学习API的使用, 我们按照Java API的规范来使用即可

1. InvocationHandler

InvocationHandler 接口是Java动态代理的关键接口之一, 它定义了一个单一方法 `invoke()`, 用于处理被代理对象的方法调用.

```
public interface InvocationHandler {
    /**
     * 参数说明
     * proxy：代理对象
     * method：代理对象需要实现的方法，即其中需要重写的方法
     * args：method所对应方法的参数
     */
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

通过实现 `InvocationHandler` 接口, 可以对被代理对象的方法进行功能增强.

2. Proxy

`Proxy` 类中使用频率最高的方法是: **`newProxyInstance()`**, 这个方法主要用来生成一个代理对象

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
```

```
                                          InvocationHandler h)
        throws IllegalArgumentException
    {
         //...代码省略
    }
```

这个方法一共有 3 个参数：

Loader: 类加载器, 用于加载代理对象.

interfaces : 被代理类实现的一些接口(这个参数的定义, 也决定了JDK动态代理只能代理实现了接口的一些类)

h : 实现了 InvocationHandler 接口的对象

CGLIB动态代理

JDK 动态代理有一个最致命的问题是其只能代理实现了接口的类.

有些场景下, 我们的业务代码是直接实现的, 并没有接口定义. 为了解决这个问题, 我们可以用 CGLIB 动态代理机制来解决.

CGLIB(Code Generation Library)是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成. CGLIB 通过继承方式实现代理，很多知名的开源框架都使用到了CGLIB. 例如 Spring 中的 AOP 模块中: 如果目标对象实现了接口，则默认采用 JDK 动态代理, 否则采用 CGLIB 动态代理.

CGLIB 动态代理类实现步骤

1. 定义一个类(被代理类)
2. 自定义 `MethodInterceptor` 并重写 `intercept` 方法， `intercept` 用于增强目标方法，和 JDK 动态代理中的 `invoke` 方法类似
3. 通过 Enhancer 类的 create()创建代理类

接下来看下实现:

添加依赖

> 和JDK 动态代理不同, CGLIB(*Code Generation Library*) 实际是属于一个开源项目，如果你要使用它的话，需要手动添加相关依赖

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>3.3.0</version>
</dependency>
```

自定义 MethodInterceptor（方法拦截器）

实现MethodInterceptor接口

```
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class CGLIBInterceptor implements MethodInterceptor {
    //目标对象，即被代理对象
    private Object target;

    public CGLIBInterceptor(Object target){
        this.target = target;
    }

    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        // 代理增强内容
        System.out.println("我是中介，开始代理");
        //通过反射调用被代理类的方法
        Object retVal = methodProxy.invoke(target, objects);
        //代理增强内容
        System.out.println("我是中介，代理结束");
        return retVal;
    }
}
```

创建代理类, 并使用

```
public class DynamicMain {
    public static void main(String[] args) {
        HouseSubject target=  new RealHouseSubject();
        HouseSubject proxy= (HouseSubject)
Enhancer.create(target.getClass(),new CGLIBInterceptor(target));
        proxy.rentHouse();
    }
}
```

代码简单讲解

1. **MethodInterceptor**

`MethodInterceptor` 和 JDK动态代理中的 `InvocationHandler` 类似, 它只定义了一个方法 `intercept()` ,用于增强目标方法.

```
public interface MethodInterceptor extends Callback {
    /**
     * 参数说明:
     * o: 被代理的对象
     * method: 目标方法(被拦截的方法, 也就是需要增强的方法)
     * objects: 方法入参
     * methodProxy: 用于调用原始方法
     */
    Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable;
}
```

2. Enhancer.create()

Enhancer.create() 用来生成一个代理对象

```
public static Object create(Class type, Callback callback) {
   //...代码省略
}
```

参数说明:

type: 被代理类的类型(类或接口)

callback: 自定义方法拦截器 MethodInterceptor

4.2 Spring AOP 源码剖析(了解)

Spring AOP 主要基于两种方式实现的: JDK 及 CGLIB 的方式

> Spring 源码过于复杂, 我们只摘出一些主要内容, 以了解为主

Spring对于AOP的实现，基本上都是靠 `AnnotationAwareAspectJAutoProxyCreator` 去完成

生成代理对象的逻辑在父类 `AbstractAutoProxyCreator` 中

```
protected Object createProxy(Class<?> beanClass, @Nullable String beanName,
      @Nullable Object[] specificInterceptors, TargetSource targetSource) {

```

```
4     if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
5        AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)
  this.beanFactory, beanName, beanClass);
6     }
7     //创建代理工厂
8     ProxyFactory proxyFactory = new ProxyFactory();
9     proxyFactory.copyFrom(this);
10
11    /**
12     * 检查proxyTargetClass属性值，spring默认为false
13     * proxyTargetClass 检查接口是否对类代理，而不是对接口代理
14     * 如果代理对象为类，设置为true，使用cglib代理
15     */
16    if (!proxyFactory.isProxyTargetClass()) {
17        //是否有设置cglib代理
18        if (shouldProxyTargetClass(beanClass, beanName)) {
19            //设置proxyTargetClass为true,使用cglib代理
20            proxyFactory.setProxyTargetClass(true);
21        } else {
22            /**
23             * 如果beanClass实现了接口，且接口至少有一个自定义方法，则使用JDK代理
24             * 否则CGLIB代理(设置ProxyTargetClass为true )
25             * 即使我们配置了proxyTargetClass=false，经过这里的一些判断还是可能会将其
  设为true
26             */
27            evaluateProxyInterfaces(beanClass, proxyFactory);
28        }
29    }
30
31    Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
32    proxyFactory.addAdvisors(advisors);
33    proxyFactory.setTargetSource(targetSource);
34    customizeProxyFactory(proxyFactory);
35
36    proxyFactory.setFrozen(this.freezeProxy);
37    if (advisorsPreFiltered()) {
38       proxyFactory.setPreFiltered(true);
39    }
40
41    // Use original ClassLoader if bean class not locally loaded in overriding
  class loader
42    ClassLoader classLoader = getProxyClassLoader();
43    if (classLoader instanceof SmartClassLoader && classLoader !=
  beanClass.getClassLoader()) {
44       classLoader = ((SmartClassLoader) classLoader).getOriginalClassLoader();
45    }
46    //从代理工厂中获取代理
```

```
47     return proxyFactory.getProxy(classLoader);
48 }
```

代理工厂有一个重要的属性: proxyTargetClass, 默认值为false. 也可以通过程序设置

proxyTargetClass	目标对象	代理方式
false	实现了接口	jdk代理
false	未实现接口(只有实现类)	cglib代理
true	实现了接口	cglib代理
true	未实现接口(只有实现类)	cglib代理

> 可以通过 `@EnableAspectJAutoProxy(proxyTargetClass = true)` 来设置
>
> 注意:
>
> Spring Boot 2.X开始, 默认使用CGLIB代理
>
> 可以通过配置项 `spring.aop.proxy-target-class=false` 来进行修改,设置默认为jdk代理
>
> SpringBoot设置 `@EnableAspectJAutoProxy` 无效, 因为Spring Boot 默认使用AopAutoConfiguration进行装配

```
1  @SpringBootApplication
2  public class DemoApplication {
3
4     public static void main(String[] args) {
5        ApplicationContext context =
   SpringApplication.run(DemoApplication.class, args);
6        /**
7         * HouseProxy houseProxy = context.getBean(HouseProxy.class);
8         * 设置spring.aop.proxy-target-class=true  cglib代理, 运行成功
9         * 设置spring.aop.proxy-target-class=false  jdk代理, 运行失败, 不能代理类
10        * 因为 HouseProxy 是一个类, 而不是接口, 需要修改为
11        * HouseSubject houseProxy = (HouseSubject)
   context.getBean("realHouseSubject")
12        *
13        */
14       HouseProxy houseProxy = context.getBean(HouseProxy.class);
15       //HouseSubject houseProxy = (HouseSubject)
   context.getBean("realHouseSubject");//正确运行
16       System.out.println(houseProxy.getClass().toString());
17    }
```

```
18 }
```

使用context.getBean() 需要添加注解,使HouseProxy,RealHouseSubject被Spring管理

测试AOP代理, 需要把这些类交给AOP管理(自定义注解或使用@Aspect)

我看点进去看代理工厂的代码

```
1  public class ProxyFactory extends ProxyCreatorSupport {
2      //...代码省略
3      //获取代理
4      public Object getProxy(@Nullable ClassLoader classLoader) {
5         //分两步 先createAopProxy,后getProxy
6         return createAopProxy().getProxy(classLoader);
7      }
8
9      protected final synchronized AopProxy createAopProxy() {
10        if (!this.active) {
11           activate();
12        }
13        return getAopProxyFactory().createAopProxy(this);
14     }
15      //...代码省略
16
17 }
```

createAopProxy的实现在 DefaultAopProxyFactory中

```
1  public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {
2      //...代码省略
3      @Override
4      public AopProxy createAopProxy(AdvisedSupport config) throws
   AopConfigException {
5         /**
6          * 根据proxyTargetClass判断
7          * 如果目标类是接口，使用JDK动态代理
8          * 否则使用cglib动态代理
9          */
10        if (!NativeDetector.inNativeImage() &&
11              (config.isOptimize() || config.isProxyTargetClass() ||
   hasNoUserSuppliedProxyInterfaces(config))) {
12           Class<?> targetClass = config.getTargetClass();
13           if (targetClass == null) {
```

```
            throw new AopConfigException("TargetSource cannot determine target class: " +
                    "Either an interface or a target is required for proxy creation.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass) || ClassUtils.isLambdaClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {

            return new JdkDynamicAopProxy(config);
        }
    }
    //...代码省略
}
```

接下来就是创建代理了

JDK动态代理

```
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {
    //...代码省略
    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isTraceEnabled()) {
            logger.trace("Creating JDK dynamic proxy: " + this.advised.getTargetSource());
        }
        return Proxy.newProxyInstance(determineClassLoader(classLoader), this.proxiedInterfaces, this);
    }
    //...代码省略
}
```

CGLIB动态代理

```
class CglibAopProxy implements AopProxy, Serializable {
    //...代码省略
    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
```

```
            //...代码省略

            // Configure CGLIB Enhancer...
            Enhancer enhancer = createEnhancer();

            // Generate the proxy class and create a proxy instance.
            return createProxyClassAndInstance(enhancer, callbacks);

        }
        //...代码省略
}

```

总结

1. AOP是一种思想, 是对某一类事情的集中处理. Spring框架实现了AOP, 称之为SpringAOP
2. Spring AOP常见实现方式有两种: 1. 基于注解@Aspect来实现 2. 基于自定义注解来实现, 还有一些更原始的方式,比如基于代理, 基于xml配置的方式, 但目标比较少见
3. Spring AOP 是基于动态代理实现的, 有两种方式: 1. 基本JDK动态代理实现 2. 基于CGLIB动态代理实现. 运行时使用哪种方式与项目配置和代理的对象有关.