

AOP

1. AOP基础

接下来我们进入到AOP的学习。AOP也是spring框架的第二大核心，我们先来学习AOP的基础。

在AOP基础这个阶段，我们首先介绍一下什么是AOP，再通过一个快速入门程序，让大家快速体验AOP程序的开发。最后再介绍AOP当中所涉及的一些核心的概念。

1.1 AOP概述

什么是AOP？

- **AOP: Aspect Oriented Programming (面向切面编程、面向方面编程)**，其实说白了，面向切面编程就是面向特定方法编程。

那什么又是面向方法编程呢，为什么又需要面向方法编程呢？来我们举个例子做一个说明：

比如，我们这里有一个项目，项目中开发了很多的业务功能。然而有一些业务功能执行效率比较低，执行耗时较长，我们需要针对于这些业务方法进行优化。那首先第一步就需要定位出执行耗时比较长的业务方法，再针对于业务方法再进行优化。

此时我们就需要统计当前这个项目当中每一个业务方法的执行耗时。那么统计每一个业务方法的执行耗时该怎么实现？

可能多数人首先想到的就是在每一个业务方法运行之前，记录这个方法运行的开始时间。在这个方法运行完毕之后，再来记录这个方法运行的结束时间。拿结束时间减去开始时间，不就是这个方法的执行耗时吗？

如下所示：

```

EmpServiceImpl

public List<Dept> list(){
    List<Dept> deptList = deptMapper.list();
    return deptList;
}

public void delete(Integer id) {
    deptMapper.delete(id);
}

public Dept getById(Integer id) {
    Dept dept = deptMapper.getById(id);
    return dept;
}

```



原始方式

```

public List<Dept> list(){
    long beginTime = System.currentTimeMillis();
    List<Dept> deptList = deptMapper.list();
    long endTime = System.currentTimeMillis();
    log.info(" 执行耗时: {} ms", endTime - beginTime);
    return deptList;
}

public void delete(Integer id) {
    long beginTime = System.currentTimeMillis();
    deptMapper.delete(id);
    long endTime = System.currentTimeMillis();
    log.info(" 执行耗时: {} ms", endTime - beginTime);
}

public Dept getById(Integer id){
    long beginTime = System.currentTimeMillis();
    Dept dept = deptMapper.getById(id);
    long endTime = System.currentTimeMillis();
    log.info(" 执行耗时: {} ms", endTime - beginTime);
    return dept;
}

//其他方法省略...

```

以上分析的实现方式是可以解决需求问题的。但是对于一个项目来讲，里面会包含很多的业务模块，每个业务模块又包含很多增删改查的方法，如果我们要在每一个模块下的业务方法中，添加记录开始时间、结束时间、计算执行耗时的代码，就会让程序员的工作变得非常繁琐。

而如果我们基于 AOP 技术来实现，将会变得很简单。在 AOP 中，我们可以把某一些方法共性的一些功能统一进行抽取，定义在 AOP 程序中，程序运行时，动态的为原始业务方法进行增强，更加灵活、高效、简洁。具体代码量如下：

```

EmpServiceImpl

public List<Dept> list(){
    List<Dept> deptList = deptMapper.list();
    return deptList;
}

public void delete(Integer id) {
    deptMapper.delete(id);
}

public Dept getById(Integer id) {
    Dept dept = deptMapper.getById(id);
    return dept;
}

```



AOP方式

```

@Slf4j
@Aspect
@Component
public class RecordTimeAspect {
    @Around("execution(* com.itheima.service.*.*(..))")
    public Object recordTime(ProceedingJoinPoint pjp) {
        long beginTime = System.currentTimeMillis();
        Object result = pjp.proceed();
        long endTime = System.currentTimeMillis();
        log.info("执行耗时: {}ms", endTime - beginTime);
        return result;
    }
}

```

而AOP面向方法编程，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。而且，基于AOP，还可以灵活的控制，需要针对于哪些方法增强这些共性功能。

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）

那面向这样的指定的一个或多个方法进行编程，我们就称之为 面向切面编程。

AOP的优势：

1. 减少重复代码
2. 代码无侵入
3. 提高开发效率
4. 维护方便

AOP 是一种思想，而在Spring框架中，对这种思想进行了实现，而我们要学习的就是 Spring框架中对AOP思想的实现 SpringAOP。

1.2 AOP快速入门

在了解了什么是AOP后，我们下面通过一个快速入门程序，体验下AOP的开发，并掌握Spring中AOP的开发步骤。

1). 需求：统计部门管理各个业务层方法执行耗时。

2). 原始方式：

在原始的实现方式中，我们需要在业务层的也一个方法执行执行，获取方法运行的开始时间；然后运行原始的方法逻辑；最后在每一个方法运行结束时，获取方法运行结束时间，计算执行耗时。

```
原始方式

public List<Dept> list(){
    long beginTime = System.currentTimeMillis();
    List<Dept> deptList = deptMapper.list();
    long endTime = System.currentTimeMillis();
    log.info(" 执行耗时: {} ms", endTime - beginTime);
    return deptList;
}

public void delete(Integer id) {
    long beginTime = System.currentTimeMillis();
    deptMapper.delete(id);
    long endTime = System.currentTimeMillis();
    log.info(" 执行耗时: {} ms", endTime - beginTime);
}

//其他方法省略...
```

```
DeptServiceImpl

public List<Dept> list(){
    List<Dept> deptList = deptMapper.list();
    return deptList;
}

public void delete(Integer id) {
    deptMapper.delete(id);
}

//其他方法省略...
```

- 获取方法运行的开始时间
- 运行原始方法
- 获取方法运行结束时间，计算执行耗时

3). SpringAOP实现步骤:

1. 导入依赖：在pom.xml中导入AOP的依赖
2. 编写AOP程序：针对于特定方法根据业务需要进行编程

为演示方便，可以直接导入资料中提供的 `springboot-aop-quickstart` 项目工程

pom.xml

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

AOP程序：TimeAspect

```
@Component
@Aspect //当前类为切面类
@Slf4j
public class TimeAspect {

    @Around("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
    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();

        //计算方法执行耗时
        log.info("方法执行耗时: {}毫秒", end - begin);
        return result;
    }
}
```

重新启动SpringBoot服务，打开浏览器访问部门管理的功能进行测试：

1). 查询部门列表

首页	部门管理	新增
班级学员管理		
系统信息管理		
部门管理		
员工管理		
数据统计管理		

序号	部门名称	最后修改时间	操作
1	学工部	2023-12-09T10:13:02	修改 删除
2	教研部	2023-10-09T15:17:04	修改 删除
3	咨询部	2023-12-09T10:09:40	修改 删除
4	就业部	2023-09-25T09:47:40	修改 删除
5	人事部	2023-09-25T09:47:40	修改 删除
6	行政部	2023-11-30T20:56:37	修改 删除

可以看到控制台输入执行耗时：

```
JDBC Connection [HikariProxyConnection@1478427214 wrapping com.mysql.cj.jdbc.ConnectionImpl@6ab8e207] will not be managed by Spring
==> Preparing: select * from dept
==> Parameters:
<== Columns: id, name, create_time, update_time
<== Row: 1, 学工部, 2023-09-25 09:47:40, 2023-12-09 10:13:02
<== Row: 2, 教研部, 2023-09-25 09:47:40, 2023-10-09 15:17:04
<== Row: 3, 咨询部, 2023-09-25 09:47:40, 2023-12-09 10:09:40
<== Row: 4, 就业部, 2023-09-25 09:47:40, 2023-09-25 09:47:40
<== Row: 5, 人事部, 2023-09-25 09:47:40, 2023-09-25 09:47:40
<== Row: 15, 行政部, 2023-11-30 20:56:37, 2023-11-30 20:56:37
<== Total: 6
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@106ed045]
2023-12-09T10:15:30.048+08:00 INFO 12436 --- [nio-8080-exec-1] com.itheima.aop.TimeAspect : 方法执行耗时: 6毫秒
```

2). 查询部门数据回显

首页	部门管理	新增
班级学员管理		
系统信息管理		
部门管理		
员工管理		
数据统计管理		

序号	部门名称	最后修改时间	操作
1	学工部	2023-12-09T10:13:02	修改 删除
2	教研部	2023-10-09T15:17:04	修改 删除
3	咨询部	2023-12-09T10:09:40	修改 删除
4	就业部	2023-09-25T09:47:40	修改 删除
5	人事部	2023-09-25T09:47:40	修改 删除
6	行政部	2023-11-30T20:56:37	修改 删除

修改部门

* 部门名称

[取消](#) [保存](#)

可以看到控制台输出的执行耗时：

```
JDBC Connection [HikariProxyConnection@879545607 wrapping com.mysql.cj.jdbc.ConnectionImpl@6ab8e207] will not be managed by Spring
==> Preparing: select * from dept where id = ?
==> Parameters: 15(Integer)
<== Columns: id, name, create_time, update_time
<== Row: 15, 行政部, 2023-11-30 20:56:37, 2023-11-30 20:56:37
<== Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@3d1589ec]
2023-12-09T10:16:34.709+08:00 INFO 12436 --- [nio-8080-exec-2] com.itheima.aop.TimeAspect : 方法执行耗时: 2毫秒
```

我们通过AOP入门程序完成了业务方法执行耗时的统计，那其实AOP的功能远不止于此，常见的应用场景如下：

- 记录系统的操作日志
- 权限控制
- 事务管理：我们前面所讲解的Spring事务管理，底层其实也是通过AOP来实现的，只要添加@Transactional注解之后，AOP程序自动会在原始方法运行前先来开启事务，在原始方法运行完毕之后提交或回滚事务

这些都是AOP应用的典型场景。

通过入门程序，我们也应该感受到了AOP面向切面编程的一些优势：

- 代码无侵入：没有修改原始的业务方法，就已经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

1.3 AOP核心概念

通过SpringAOP的快速入门，感受了一下AOP面向切面编程的开发方式。下面我们再来学习AOP当中涉及到的一些核心概念。

1. 连接点：JoinPoint，可以被AOP控制的方法（暗含方法执行时的相关信息）

连接点指的是可以被aop控制的方法。例如：入门程序当中所有的业务方法都是可以被aop控制的方法。

```

@Service
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptMapper deptMapper;

    @Override
    public List<Dept> list() {
        List<Dept> deptList = deptMapper.list();
        return deptList;
    }

    @Override
    public void delete(Integer id) {
        deptMapper.delete(id);
    }

    @Override
    public void save(Dept dept) {
        dept.setCreateTime(LocalDateTime.now());
        dept.setUpdateTime(LocalDateTime.now());
        deptMapper.save(dept);
    }

    @Override
    public Dept getById(Integer id) {
        return deptMapper.getById(id);
    }
}

```

连接点

在SpringAOP提供的JoinPoint当中，封装了连接点方法在执行时的相关信息。（后面会有具体的讲解）

2. 通知：Advice，指哪些重复的逻辑，也就是共性功能（最终体现为一个方法）

在入门程序中是需要统计各个业务方法的执行耗时的，此时我们就需要在这些业务方法运行开始之前，先记录这个方法运行的开始时间，在每一个业务方法运行结束的时候，再来记录这个方法运行的结束时间。

但是在AOP面向切面编程当中，我们只需要将这部分重复的代码逻辑抽取出来单独定义。抽取出来的这一部分重复的逻辑，也就是共性的功能。

```

@Slf4j
@Aspect
@Component
public class RecordTimeAspect {

    @Around("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
    public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
        long beginTime = System.currentTimeMillis();
        Object result = pjp.proceed();
        long endTime = System.currentTimeMillis();
        log.info(" 执行耗时: {} ms", endTime - beginTime);
        return result;
    }
}

```

通知

3. 切入点: PointCut, 匹配连接点的条件, 通知仅会在切入点方法执行时被应用

在通知当中, 我们所定义的共性功能到底要应用在哪些方法上? 此时就涉及到了切入点pointcut概念。切入点指的是匹配连接点的条件。通知仅会在切入点方法运行时才会被应用。

在aop的开发当中, 我们通常会通过一个切入点表达式来描述切入点(后面会有详解)。

```

@Slf4j
@Aspect
@Component
public class RecordTimeAspect {  切入点表达式

    @Around("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
    public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
        long beginTime = System.currentTimeMillis();
        Object result = pjp.proceed();
        long endTime = System.currentTimeMillis();
        log.info(" 执行耗时: {} ms", endTime - beginTime);
        return result;
    }
}

```

通知

假如: 切入点表达式改为DeptServiceImpl.list(), 此时就代表仅仅只有list这一个方法是切入点。只有list()方法在运行的时候才会应用通知。

4. 切面: Aspect, 描述通知与切入点的对应关系 (通知+切入点)

当通知和切入点结合在一起, 就形成了一个切面。通过切面就能够描述当前aop程序需要针对于哪个原始方法, 在什么时候执行什么样的操作。

```
@Slf4j
@Aspect 切面类
@Component
public class RecordTimeAspect { 切入点表达式

    @Around("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
    public Object recordTime(ProceedingJoinPoint pjp) throws Throwable {
        long beginTime = System.currentTimeMillis();
        Object result = pjp.proceed();
        long endTime = System.currentTimeMillis();
        log.info(" 执行耗时: {} ms", endTime - beginTime);
        return result;
    }
}
```

通知 **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)

5. 目标对象: Target, 通知所应用的对象

目标对象指的就是通知所应用的对象, 我们就称之为目标对象。

```

@Service
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptMapper deptMapper;

    @Override
    public List<Dept> list() {
        List<Dept> deptList = deptMapper.list();
        return deptList;
    }

    @Override
    public void delete(Integer id) {
        deptMapper.delete(id);
    }

    @Override
    public void save(Dept dept) {
        dept.setCreateTime(LocalDate.now());
        dept.setUpdateTime(LocalDate.now());
        deptMapper.save(dept);
    }

    @Override
    public Dept getById(Integer id) {
        return deptMapper.getById(id);
    }
}

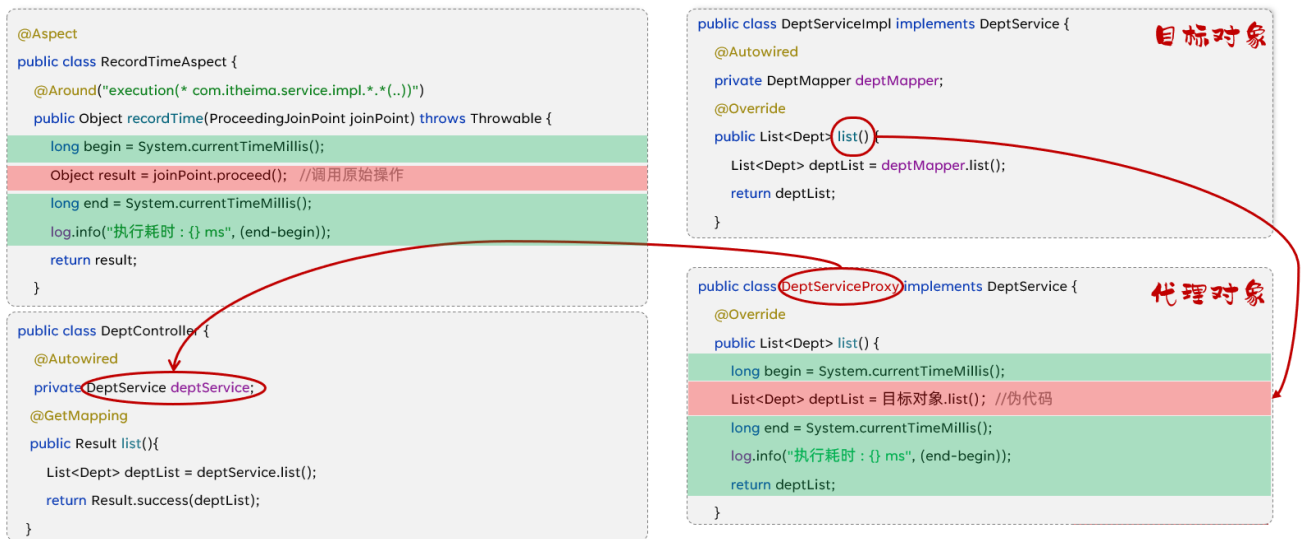
```

目标对象

切入点

连接点

AOP的核心概念我们介绍完毕之后，接下来我们再来分析一下我们所定义的通知是如何与目标对象结合在一起，对目标对象当中的方法进行功能增强的。



Spring的AOP底层是基于动态代理技术来实现的，也就是说在程序运行的时候，会自动的基于动态代理技术为目标对象生成一个对应的代理对象。在代理对象当中就会对目标对象当中的原始方法进行功能的增强。

SpringAOP 旨在管理bean对象的过程中，主要通过底层的动态代理机制，对特定的方法进行编程。

2. AOP进阶

AOP的基础知识学习完之后，下面我们对AOP当中的各个细节进行详细的学习。主要分为4个部分：

1. 通知类型
2. 通知顺序
3. 切入点表达式
4. 连接点

我们先来学习第一部分通知类型。

2.1 通知类型

在入门程序当中，我们已经使用了一种功能最为强大的通知类型：Around环绕通知。

```
@Component
@Aspect //当前类为切面类
@Slf4j
public class TimeAspect {

    @Around("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
    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();
    //计算方法执行耗时
    log.info("方法执行耗时: {}毫秒",end-begin);
    return result;
}
}

```

只要我们在通知方法上加上了@Around注解，就代表当前通知是一个环绕通知。

Spring中AOP的通知类型：

- @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

下面我们通过代码演示，来加深对于不同通知类型的理解：

```

@Slf4j
@Component
@Aspect
public class MyAspect1 {
    //前置通知
    @Before("execution(* com.itheima.service.*.*(..))")
    public void before(JoinPoint joinPoint){
        log.info("before ...");
    }

    //环绕通知
    @Around("execution(* com.itheima.service.*.*(..))")
    public Object around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        log.info("around before ...");

        //调用目标对象的原始方法执行
        Object result = proceedingJoinPoint.proceed();

        //原始方法如果执行时有异常，环绕通知中的后置代码不会在执行了

        log.info("around after ...");
        return result;
    }

    //后置通知
    @After("execution(* com.itheima.service.*.*(..))")
    public void after(JoinPoint joinPoint){

```

```

    log.info("after ...");
}

//返回后通知（程序在正常执行的情况下，会执行的后置通知）
@AfterReturning("execution(* com.itheima.service.*(..))")
public void afterReturning(JoinPoint joinPoint){
    log.info("afterReturning ...");
}

//异常通知（程序在出现异常的情况下，执行的后置通知）
@AfterThrowing("execution(* com.itheima.service.*(..))")
public void afterThrowing(JoinPoint joinPoint){
    log.info("afterThrowing ...");
}
}
}

```

重新启动SpringBoot服务，进行测试：

1. 没有异常情况下：

- 使用 Apifox 测试查询所有部门数据

The screenshot shows the Apifox interface for a GET request to `http://localhost:8080/depts`. The response body is displayed in JSON format:

```

{
  "code": 1,
  "msg": "success",
  "data": [
    {
      "id": 1,
      "name": "学工部",
      "createTime": "2023-09-25T09:47:40",
      "updateTime": "2023-12-09T10:13:02"
    },
    {
      "id": 2,
      "name": "教研部",
      "createTime": "2023-09-25T09:47:40",
      "updateTime": "2023-10-09T15:17:04"
    }
  ]
}

```

The right sidebar indicates a warning: "返回数据结构与接口定义不一致" (Return data structure inconsistent with interface definition). The message states: "1. \$.data 应当是 object,null 类型" (1. \$.data should be object, null type).

- 查看idea中控制台日志输出

```

2023-12-09T10:52:50.998+08:00 INFO 1492 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
2023-12-09T10:52:51.022+08:00 INFO 1492 --- [nio-8080-exec-1] com.itheima.aop.MyAspect1 : around before ...
2023-12-09T10:52:51.023+08:00 INFO 1492 --- [nio-8080-exec-1] com.itheima.aop.MyAspect1 : before ...
2023-12-09T10:52:51.037+08:00 INFO 1492 --- [nio-8080-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2023-12-09T10:52:51.366+08:00 INFO 1492 --- [nio-8080-exec-1] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection com.mysql.cj.jdbc.Conn
2023-12-09T10:52:51.368+08:00 INFO 1492 --- [nio-8080-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2023-12-09T10:52:51.404+08:00 INFO 1492 --- [nio-8080-exec-1] com.itheima.aop.MyAspect1 : afterReturning ...
2023-12-09T10:52:51.405+08:00 INFO 1492 --- [nio-8080-exec-1] com.itheima.aop.MyAspect1 : after ...
2023-12-09T10:52:51.405+08:00 INFO 1492 --- [nio-8080-exec-1] com.itheima.aop.MyAspect1 : around after ...

```

程序没有发生异常的情况下，@AfterThrowing标识的通知方法不会执行。

2. 出现异常情况下：

修改DeptServiceImpl业务实现类中的代码： 添加异常

```

@Slf4j
@Service
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptMapper deptMapper;

    @Override
    public List<Dept> list() {

        List<Dept> deptList = deptMapper.list();
        //模拟异常
        int num = 10/0;
        return deptList;
    }

    //省略其他代码...
}

```

重新启动SpringBoot服务，测试发生异常情况下通知的执行：

- 查看idea中控制台日志输出

```

2023-12-09T10:54:30.611+08:00 INFO 11852 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
2023-12-09T10:54:30.635+08:00 INFO 11852 --- [nio-8080-exec-2] com.itheima.aop.MyAspect1 : around before ...
2023-12-09T10:54:30.635+08:00 INFO 11852 --- [nio-8080-exec-2] com.itheima.aop.MyAspect1 : before ...
2023-12-09T10:54:30.649+08:00 INFO 11852 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2023-12-09T10:54:30.986+08:00 INFO 11852 --- [nio-8080-exec-2] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection com.mysql.cj.jdbc.Conn
2023-12-09T10:54:30.987+08:00 INFO 11852 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2023-12-09T10:54:31.022+08:00 INFO 11852 --- [nio-8080-exec-2] com.itheima.aop.MyAspect1 : afterThrowing ...
2023-12-09T10:54:31.022+08:00 INFO 11852 --- [nio-8080-exec-2] com.itheima.aop.MyAspect1 : after ...
2023-12-09T10:54:31.027+08:00 ERROR 11852 --- [nio-8080-exec-2] o.a.c.c.c.[.][.][dispatcherServlet] : Servlet.service() for servlet [dispatcherServlet] in c

java.lang.ArithmeticException: Create breakpoint : / by zero
at com.itheima.service.impl.DeptServiceImpl.list(DeptServiceImpl.java:22) ~[classes/:na] <2 internal lines>
at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:343) ~[spring-aop-6.0.14.jar:6.0.14]

```

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执行了（因为原始方法调用已经出异常了）

在使用通知时的注意事项：

- @Around环绕通知需要自己调用 `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。

五种常见的通知类型，我们已经测试完毕了，此时我们再来看一下刚才所编写的代码，有什么问题吗？

```
//前置通知
@Before("execution(* com.itheima.service.*(..))")

//环绕通知
@Around("execution(* com.itheima.service.*(..))")

//后置通知
@After("execution(* com.itheima.service.*(..))")

//返回后通知（程序在正常执行的情况下，会执行的后置通知）
@AfterReturning("execution(* com.itheima.service.*(..))")

//异常通知（程序在出现异常的情况下，执行的后置通知）
@AfterThrowing("execution(* com.itheima.service.*(..))")
```

我们发现啊，每一个注解里面都指定了切入点表达式，而且这些切入点表达式都一模一样。此时我们的代码当中就存在了大量的重复性的切入点表达式，假如此时切入点表达式需要变动，就需要将所有的切入点表达式一个一个的来改动，就变得非常繁琐了。

怎么来解决这个切入点表达式重复的问题？答案就是：**抽取**

Spring提供了@PointCut注解，该注解的作用是将公共的切入点表达式抽取出来，需要用到时引用该切入点表达式即可。

```
@Slf4j
@Component
@Aspect
public class MyAspect1 {

    //切入点方法（公共的切入点表达式）
    @Pointcut("execution(* com.itheima.service.*(..))")
    private void pt(){

    }

}
```

```

//前置通知 (引用切入点)
@Before("pt()")
public void before(JoinPoint joinPoint){
    log.info("before ...");
}

//环绕通知
@Around("pt()")
public Object around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    log.info("around before ...");

    //调用目标对象的原始方法执行
    Object result = proceedingJoinPoint.proceed();
    //原始方法在执行时：发生异常
    //后续代码不在执行

    log.info("around after ...");
    return result;
}

//后置通知
@After("pt()")
public void after(JoinPoint joinPoint){
    log.info("after ...");
}

//返回后通知 (程序在正常执行的情况下，会执行的后置通知)
@AfterReturning("pt()")
public void afterReturning(JoinPoint joinPoint){
    log.info("afterReturning ...");
}

//异常通知 (程序在出现异常的情况下，执行的后置通知)
@AfterThrowing("pt()")
public void afterThrowing(JoinPoint joinPoint){
    log.info("afterThrowing ...");
}
}

```

需要注意的是：当切入点方法使用private修饰时，仅能在当前切面类中引用该表达式，当外部其他切面类中也要引用当前类中的切入点表达式，就需要把private改为public，而在引用的时候，具体的语法为：

全类名.方法名()，具体形式如下：


```

@Slf4j
@Component
@Aspect
public class MyAspect2 {
    //引用MyAspect1切面类中的切入点表达式
    @Before("com.itheima.aspect.MyAspect1.pt()")
    public void before(){
        log.info("MyAspect2 -> before ...");
    }
}

```

2.2 通知顺序

讲解完了Spring中AOP所支持的5种通知类型之后，接下来我们再来研究通知的执行顺序。

当在项目开发当中，我们定义了多个切面类，而多个切面类中多个切入点都匹配到了同一个目标方法。此时当目标方法在运行的时候，这多个切面类当中的这些通知方法都会运行。

此时我们就有一个疑问，这多个通知方法到底哪个先运行，哪个后运行？下面我们通过程序来验证（这里呢，我们就定义两种类型的通知进行测试，一种是前置通知@Before，一种是后置通知@After）

定义多个切面类：

```

@Slf4j
@Component
@Aspect
public class MyAspect2 {
    //前置通知
    @Before("execution(* com.itheima.service.*.*(..))")
    public void before(){
        log.info("MyAspect2 -> before ...");
    }

    //后置通知
    @After("execution(* com.itheima.service.*.*(..))")
    public void after(){
        log.info("MyAspect2 -> after ...");
    }
}

```

```

@Slf4j

```

```

@Component
@Aspect
public class MyAspect3 {
    //前置通知
    @Before("execution(* com.itheima.service.*.*(..))")
    public void before(){
        log.info("MyAspect3 -> before ...");
    }

    //后置通知
    @After("execution(* com.itheima.service.*.*(..))")
    public void after(){
        log.info("MyAspect3 -> after ...");
    }
}

```

```

@Slf4j
@Component
@Aspect
public class MyAspect4 {
    //前置通知
    @Before("execution(* com.itheima.service.*.*(..))")
    public void before(){
        log.info("MyAspect4 -> before ...");
    }

    //后置通知
    @After("execution(* com.itheima.service.*.*(..))")
    public void after(){
        log.info("MyAspect4 -> after ...");
    }
}

```

重新启动SpringBoot服务，测试通知的执行顺序：

备注：

1. 把DeptServiceImpl实现类中模拟异常的代码删除或注释掉。
 2. 注释掉其他切面类(把@Aspect注释即可)，仅保留MyAspect2、MyAspect3、MyAspect4，这样就可以清晰看到执行的结果，而不被其他切面类干扰。
- 使用 Apifox 测试查询所有部门数据



• 查看idea中控制台日志输出

```
2023-12-09T10:58:34.798+08:00 INFO 6972 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2023-12-09T10:58:34.798+08:00 INFO 6972 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 0 ms
2023-12-09T10:58:34.829+08:00 INFO 6972 --- [nio-8080-exec-2] com.itheima.aop.MyAspect2 : MyAspect2 -> before ...
2023-12-09T10:58:34.829+08:00 INFO 6972 --- [nio-8080-exec-2] com.itheima.aop.MyAspect3 : MyAspect3 -> before ...
2023-12-09T10:58:34.829+08:00 INFO 6972 --- [nio-8080-exec-2] com.itheima.aop.MyAspect4 : MyAspect4 -> before ...
2023-12-09T10:58:34.843+08:00 INFO 6972 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2023-12-09T10:58:35.152+08:00 INFO 6972 --- [nio-8080-exec-2] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection com.mysql.cj.jdbc.Connection
2023-12-09T10:58:35.154+08:00 INFO 6972 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2023-12-09T10:58:35.190+08:00 INFO 6972 --- [nio-8080-exec-2] com.itheima.aop.MyAspect4 : MyAspect4 -> after ...
2023-12-09T10:58:35.190+08:00 INFO 6972 --- [nio-8080-exec-2] com.itheima.aop.MyAspect3 : MyAspect3 -> after ...
2023-12-09T10:58:35.190+08:00 INFO 6972 --- [nio-8080-exec-2] com.itheima.aop.MyAspect2 : MyAspect2 -> after ...
```

通过以上程序运行可以看出在不同切面类中，默认按照切面类的类名字母排序：

- 目标方法前的通知方法：字母排名靠前的先执行
- 目标方法后的通知方法：字母排名靠前的后执行

如果我们想控制通知的执行顺序有两种方式：

1. 修改切面类的类名（这种方式非常繁琐、而且不便管理）
2. 使用Spring提供的@Order注解

使用@Order注解，控制通知的执行顺序：

```
@Slf4j
@Component
@Aspect
@Order(2) //切面类的执行顺序（前置通知：数字越小先执行；后置通知：数字越小越后执行）

public class MyAspect2 {
```

```

//前置通知
@Before("execution(* com.itheima.service.*.*(..))")
public void before(){
    log.info("MyAspect2 -> before ...");
}

//后置通知
@After("execution(* com.itheima.service.*.*(..))")
public void after(){
    log.info("MyAspect2 -> after ...");
}
}

```

```

@Slf4j
@Component
@Aspect
@Order(3) //切面类的执行顺序（前置通知：数字越小先执行; 后置通知：数字越小越后执行）
public class MyAspect3 {
    //前置通知
    @Before("execution(* com.itheima.service.*.*(..))")
    public void before(){
        log.info("MyAspect3 -> before ...");
    }

    //后置通知
    @After("execution(* com.itheima.service.*.*(..))")
    public void after(){
        log.info("MyAspect3 -> after ...");
    }
}

```

```

@Slf4j
@Component
@Aspect
@Order(1) //切面类的执行顺序（前置通知：数字越小先执行; 后置通知：数字越小越后执行）
public class MyAspect4 {
    //前置通知
    @Before("execution(* com.itheima.service.*.*(..))")
    public void before(){
        log.info("MyAspect4 -> before ...");
    }

    //后置通知
    @After("execution(* com.itheima.service.*.*(..))")
    public void after(){
        log.info("MyAspect4 -> after ...");
    }
}

```

重新启动SpringBoot服务，测试通知执行顺序：

```
2023-12-09T11:00:52.930+08:00 INFO 1384 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2023-12-09T11:00:52.931+08:00 INFO 1384 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
2023-12-09T11:00:52.958+08:00 INFO 1384 --- [nio-8080-exec-2] com.itheima.aop.MyAspect4 : MyAspect4 -> before ...
2023-12-09T11:00:52.958+08:00 INFO 1384 --- [nio-8080-exec-2] com.itheima.aop.MyAspect2 : MyAspect2 -> before ...
2023-12-09T11:00:52.958+08:00 INFO 1384 --- [nio-8080-exec-2] com.itheima.aop.MyAspect3 : MyAspect3 -> before ...
2023-12-09T11:00:52.974+08:00 INFO 1384 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2023-12-09T11:00:53.290+08:00 INFO 1384 --- [nio-8080-exec-2] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection com.mysql.cj.jdbc.Conne
2023-12-09T11:00:53.292+08:00 INFO 1384 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2023-12-09T11:00:53.329+08:00 INFO 1384 --- [nio-8080-exec-2] com.itheima.aop.MyAspect3 : MyAspect3 -> after ...
2023-12-09T11:00:53.330+08:00 INFO 1384 --- [nio-8080-exec-2] com.itheima.aop.MyAspect2 : MyAspect2 -> after ...
2023-12-09T11:00:53.330+08:00 INFO 1384 --- [nio-8080-exec-2] com.itheima.aop.MyAspect4 : MyAspect4 -> after ...
```

通知的执行顺序大家主要知道两点即可：

1. 不同的切面类当中，默认情况下通知的执行顺序是与切面类的类名字母排序是有关系的
2. 可以在切面类上面加上@Order注解，来控制不同的切面类通知的执行顺序

2.3 切入点表达式

从AOP的入门程序到现在，我们一直都在使用切入点表达式来描述切入点。下面我们就来详细的介绍一下切入点表达式的具体写法。

切入点表达式：

- 描述切入点方法的一种表达式
- 作用：主要用来决定项目中的哪些方法需要加入通知
- 常见形式：

1. execution(.....)：根据方法的签名来匹配

```
@Before("execution(public void com.itheima.service.impl.DeptServiceImpl.delete(java.lang.Integer))")
public void before(JoinPoint joinPoint){
```

2. @annotation(.....)：根据注解匹配

```
@Before("@annotation(com.itheima.anno.Log)")
public void before(){
```

首先我们先学习第一种最为常见的execution切入点表达式。

2.3.1 execution

execution主要根据方法的返回值、包名、类名、方法名、方法参数等信息来匹配，语法为：

```
execution(访问修饰符? 返回值 包名.类名.?方法名(方法参数) throws 异常?)
```

其中带?的表示可以省略的部分

- 访问修饰符：可省略（比如：public、protected）
- 包名.类名：可省略
- throws 异常：可省略（注意是方法上声明抛出的异常，不是实际抛出的异常）

示例：

```
@Before("execution(void com.itheima.service.impl.DeptServiceImpl.delete(java.lang.Integer))")
```

可以使用通配符描述切入点

- *：单个独立的任意符号，可以通配任意返回值、包名、类名、方法名、任意类型的一个参数，也可以通配包、类、方法名的一部分
- ..：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

切入点表达式的语法规则：

1. 方法的访问修饰符可以省略
2. 返回值可以使用 * 号代替（任意返回值类型）
3. 包名可以使用 * 号代替，代表任意包（一层包使用一个 *）
4. 使用 .. 配置包名，标识此包以及此包下的所有子包
5. 类名可以使用 * 号代替，标识任意类
6. 方法名可以使用 * 号代替，表示任意方法
7. 可以使用 * 配置参数，一个任意类型的参数
8. 可以使用 .. 配置参数，任意个任意类型的参数

切入点表达式示例

- 省略方法的修饰符号

```
execution(void com.itheima.service.impl.DeptServiceImpl.delete(java.lang.Integer))
```

- 使用 * 代替返回值类型

```
execution(* com.itheima.service.impl.DeptServiceImpl.delete(java.lang.Integer))
```

- 使用 * 代替包名（一层包使用一个 *）

```
execution(* com.itheima.*.*.DeptServiceImpl.delete(java.lang.Integer))
```

- 使用 .. 省略包名

```
execution(* com..DeptServiceImpl.delete(java.lang.Integer))
```

- 使用 * 代替类名

```
execution(* com..*.delete(java.lang.Integer))
```

- 使用 * 代替方法名

```
execution(* com..*.*(java.lang.Integer))
```

- 使用 * 代替参数

```
execution(* com.itheima.service.impl.DeptServiceImpl.delete(*))
```

- 使用 .. 省略参数

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

注意事项:

- 根据业务需要, 可以使用 且 (&&)、或 (||)、非 (!) 来组合比较复杂的切入点表达式。

```
execution(* com.itheima.service.DeptService.list(..)) || execution(* com.itheima.service.DeptService.delete(..))
```

切入点表达式的书写建议:

- 所有业务方法名在命名时尽量规范, 方便切入点表达式快速匹配。如: 查询类方法都是 find 开头, 更新类方法都是update开头

```
//业务类
@Service
public class DeptServiceImpl implements DeptService {

    public List<Dept> findAllDept() {
        //省略代码...
    }

    public Dept findDeptById(Integer id) {
        //省略代码...
    }

    public void updateDeptById(Integer id) {
        //省略代码...
    }

    public void updateDeptByMoreCondition(Dept dept) {
        //省略代码...
    }
    //其他代码...
}
```

```
//匹配DeptServiceImpl类中以find开头的方法
execution(* com.itheima.service.impl.DeptServiceImpl.find*(..))
```

- 描述切入点方法通常基于接口描述，而不是直接描述实现类，增强拓展性

```
execution(* com.itheima.service.DeptService.*(..))
```

- 在满足业务需要的前提下，尽量缩小切入点的匹配范围。如：包名匹配尽量不使用 ..，使用 * 匹配单个包

```
execution(* com.itheima.*.*.DeptServiceImpl.find*(..))
```

切入点表达式书写建议：

- 所有业务方法名在命名时尽量规范，方便切入点表达式快速匹配。如：findXxx，updateXxx。
- 描述切入点方法通常基于接口描述，而不是直接描述实现类，增强拓展性。
- 在满足业务需要的前提下，尽量缩小切入点的匹配范围。如：包名尽量不使用..，使用*匹配单个包。

2.3.2 @annotation

已经学习了execution切入点表达式的语法。那么如果我们要匹配多个无规则的方法，比如：list()和 delete()这两个方法。这个时候我们基于execution这种切入点表达式来描述就不是很方便了。而在之前我们是将两个切入点表达式组合在了一起完成的需求，这个是比较繁琐的。

我们可以借助于另一种切入点表达式annotation来描述这一类的切入点，从而来简化切入点表达式的书写。

实现步骤：

1. 编写自定义注解
2. 在业务类要做为连接点的方法上添加自定义注解

自定义注解： Log

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

业务类： DeptServiceImpl

```
@Slf4j
```



```

@Service
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptMapper deptMapper;

    @Override
    @Log //自定义注解 (表示: 当前方法属于目标方法)
    public List<Dept> list() {
        List<Dept> deptList = deptMapper.list();
        //模拟异常
        //int num = 10/0;
        return deptList;
    }

    @Override
    @Log //自定义注解 (表示: 当前方法属于目标方法)
    public void delete(Integer id) {
        //1. 删除部门
        deptMapper.delete(id);
    }

    @Override
    public void save(Dept dept) {
        dept.setCreateTime(LocalDate.now());
        dept.setUpdateTime(LocalDate.now());
        deptMapper.save(dept);
    }

    @Override
    public Dept getByld(Integer id) {
        return deptMapper.getByld(id);
    }

    @Override
    public void update(Dept dept) {
        dept.setUpdateTime(LocalDate.now());
        deptMapper.update(dept);
    }
}

```

切面类

```

@Slf4j
@Component
@Aspect
public class MyAspect6 {
    //针对list方法、delete方法进行前置通知和后置通知

    //前置通知
    @Before("@annotation(com.itheima.anno.Log)")

```

```

public void before(){
    log.info("MyAspect6 -> before ...");
}

//后置通知
@After("@annotation(com.itheima.anno.Log)")
public void after(){
    log.info("MyAspect6 -> after ...");
}
}

```

重启SpringBoot服务，测试查询所有部门数据，查看控制台日志：

```

2:57.307 INFO 15060 --- [nio-8080-exec-1] com.itheima.aspect.MyAspect6 : MyAspect6 -> before ...
SqlSession
.org.apache.ibatis.session.defaults.DefaultSqlSession@13521226] was not registered for synchronization because synchroni
2:57.321 INFO 15060 --- [nio-8080-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2:57.557 INFO 15060 --- [nio-8080-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed
[HikariProxyConnection@183243451 wrapping com.mysql.cj.jdbc.ConnectionImpl@245aded6] will not be managed by Spring
select * from dept

id, name, create_time, update_time
3, 咨询部, 2022-12-30 13:53:04, 2022-12-30 13:53:04
4, 就业部, 2022-12-30 13:53:04, 2022-12-30 13:53:04
2
nsactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@13521226]
2:57.604 INFO 15060 --- [nio-8080-exec-1] com.itheima.aspect.MyAspect6 : MyAspect6 -> after ...

```

到此我们两种常见的切入点表达式我已经介绍完了。

- execution切入点表达式
 - 根据我们所指定的方法的描述信息来匹配切入点方法，这种方式也是最为常用的一种方式
 - 如果我们要匹配的切入点方法的方法名不规则，或者有一些比较特殊的需求，通过execution切入点表达式描述比较繁琐
- annotation 切入点表达式
 - 基于注解的方式来匹配切入点方法。这种方式虽然多一步操作，我们需要自定义一个注解，但是相对来说比较灵活。我们需要匹配哪个方法，就在方法上加上对应的注解就可以了

根据业务需要，可以使用 && , || , ! 来组合比较复杂的切入点表达式。

2.4 连接点

讲解完了切入点表达式之后，接下来我们再来讲解最后一个部分连接点。我们前面在讲解AOP核心概念的时候，我们提到过什么是连接点，连接点可以简单理解为可以被AOP控制的方法。

我们目标对象当中所有的方法是不是都是可以被AOP控制的方法。而在SpringAOP当中，连接点又特指方法的执行。

在Spring中用JoinPoint抽象了连接点，用它可以获得方法执行时的相关信息，如目标类名、方法名、方法参数等。

- 对于@Around通知，获取连接点信息只能使用ProceedingJoinPoint类型
- 对于其他四种通知，获取连接点信息只能使用JoinPoint，它是ProceedingJoinPoint的父类型

示例代码：

```
package com.itheima.aop;

import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

import java.util.Arrays;

@Slf4j
@Component
@Aspect
public class MyAspect7 {

    @Pointcut("@annotation(com.itheima.anno.Log)")
    private void pt(){}

    //前置通知
    @Before("pt()")
    public void before(JoinPoint joinPoint){
        log.info(joinPoint.getSignature().getName() + " MyAspect7 -> before ...");
    }

    //后置通知
    @After("pt()")
    public void after(JoinPoint joinPoint){
        log.info(joinPoint.getSignature().getName() + " MyAspect7 -> after ...");
    }

    //环绕通知
    @Around("pt()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        //获取目标类名
        String name = pjp.getTarget().getClass().getName();
        log.info("目标类名: {}",name);

        //获取目标方法签名
        Signature signature = pjp.getSignature();
        log.info("目标方法签名: {}",signature);

        //目标方法名
```

```

String methodName = pjp.getSignature().getName();
log.info("目标方法名: {}",methodName);

//获取方法执行时需要的参数
Object[] args = pjp.getArgs();
log.info("目标方法参数: {}", Arrays.toString(args));

//执行原始方法
Object returnValue = pjp.proceed();
log.info("目标方法结果: {}",returnValue);

return returnValue;
}
}

```

重新启动SpringBoot服务，执行查询部门数据的功能：

```

[nio-8080-exec-1] com.itheima.aspect.MyAspect7          : 目标类名: com.itheima.service.impl.DeptServiceImpl
[nio-8080-exec-1] com.itheima.aspect.MyAspect7          : 目标方法名: list
[nio-8080-exec-1] com.itheima.aspect.MyAspect7          : 目标方法参数: []
[nio-8080-exec-1] com.itheima.aspect.MyAspect7          : list MyAspect7 -> after ...
[nio-8080-exec-1] com.itheima.aspect.MyAspect7          : list MyAspect7 -> before ...

```

3. AOP案例

SpringAOP的相关知识我们就已经全部学习完毕了。最后我们要通过一个案例来对AOP进行一个综合的应用。

3.1 需求

需求：将案例（Tlias智能学习辅助系统）中增、删、改相关接口的操作日志记录到数据库表中

- 就是当访问部门管理和员工管理当中的增、删、改相关功能接口时，需要详细的操作日志，并保存在数据表中，便于后期数据追踪。

操作日志信息包含：

- 操作人、操作时间、执行方法的全类名、执行方法名、方法运行时参数、返回值、方法执行时长

所记录的日志信息包括当前接口的操作人是谁操作的，什么时间点操作的，以及访问的是哪个类当中的哪个方法，在访问这个方法的时候传入进来的参数是什么，访问这个方法最终拿到的返回值是什么，以及整个接口方法的运行时长是多长时间。

3.2 分析

问题1：项目当中增删改相关的方法是不是有很多？

- 很多

问题2：我们需要针对每一个功能接口方法进行修改，在每一个功能接口当中都来记录这些操作日志吗？

- 这种做法比较繁琐

以上两个问题的解决方案：可以使用AOP解决(每一个增删改功能接口中要实现的记录操作日志的逻辑代码是相同)。

可以把这部分记录操作日志的通用的、重复性的逻辑代码抽取出来定义在一个通知方法当中，我们通过AOP面向切面编程的方式，在不改动原始功能的基础上来对原始的功能进行增强。目前我们所增强的功能就是来记录操作日志，所以也可以使用AOP的技术来实现。使用AOP的技术来实现也是最为简单，最为方便的。

问题3：既然要基于AOP面向切面编程的方式来完成的功能，那么我们要使用 AOP五种通知类型当中的哪种通知类型？

- 答案：环绕通知 `@Around`

所记录的操作日志当中包括：操作人、操作时间，访问的是哪个类、哪个方法、方法运行时参数、方法的返回值、方法的运行时长。

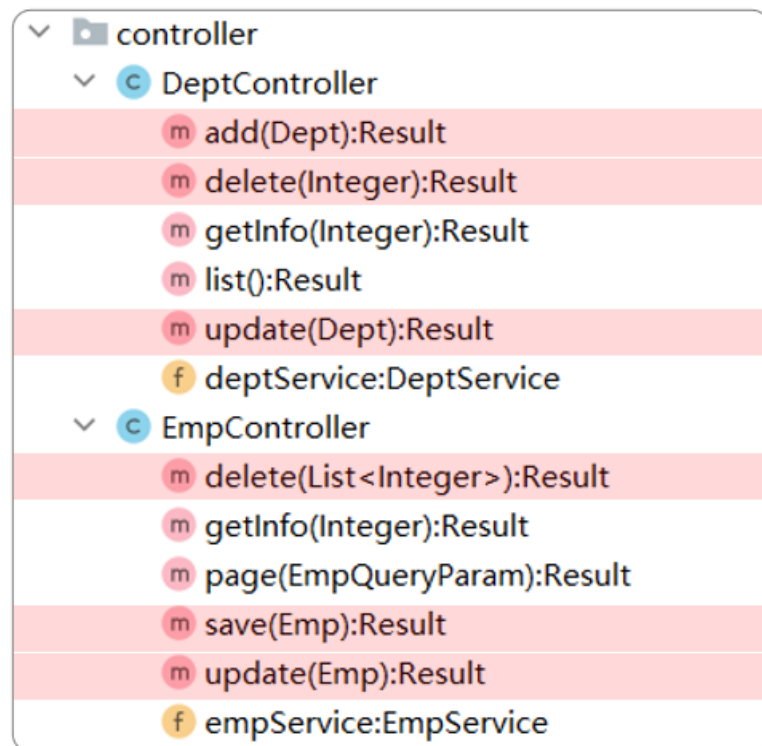
方法返回值，是在原始方法执行后才能获取到的。

方法的运行时长，需要原始方法运行之前记录开始时间，原始方法运行之后记录结束时间。通过计算获得方法的执行耗时。

基于以上的分析我们确定要使用Around环绕通知。

问题4：最后一个问题，切入点表达式我们该怎么写？

- 答案：使用 `@annotation` 来描述切入点表达式



要匹配业务接口当中所有的增删改的方法，而增删改方法在命名上没有共同的前缀或后缀。此时如果使用execution切入点表达式也可以，但是会比较繁琐。当遇到增删改的方法名没有规律时，就可以使用 annotation切入点表达式

3.3 步骤

简单分析了一下大概的实现思路后，接下来我们就要来完成案例了。案例的实现步骤其实就两步：

- 准备工作
 1. 引入AOP的起步依赖
 2. 导入资料中准备好的数据库表结构，并引入对应的实体类
- 编码实现
 1. 自定义注解@Log
 2. 定义切面类，完成记录操作日志的逻辑

3.4 实现

3.4.1 准备工作

1). AOP起步依赖

```
<!--AOP起步依赖-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

2). 导入资料中准备好的数据库表结构，并引入对应的实体类

2.1). 数据表

```
-- 操作日志表
create table operate_log(
  id int unsigned primary key auto_increment comment 'ID',
  operate_user int unsigned comment '操作人',
  operate_time datetime comment '操作时间',
  class_name varchar(100) comment '操作的类名',
  method_name varchar(100) comment '操作的方法名',
  method_params varchar(1000) comment '方法参数',
  return_value varchar(2000) comment '返回值',
  cost_time bigint comment '方法执行耗时, 单位:ms'
) comment '操作日志表';
```

2.2). 实体类

```
//操作日志实体类
@Data
@NoArgsConstructor
@AllArgsConstructor
public class OperateLog {
  private Integer id; //主键ID
  private Integer operateUser; //操作人ID
  private LocalDateTime operateTime; //操作时间
  private String className; //操作类名
  private String methodName; //操作方法名
  private String methodParams; //操作方法参数
  private String returnValue; //操作方法返回值
  private Long costTime; //操作耗时
}
```

2.3). Mapper接口

```

@Mapper
public interface OperateLogMapper {

    //插入日志数据
    @Insert("insert into operate_log (operate_user, operate_time, class_name, method_name, method_params,
return_value, cost_time) " +
        "values ({operateUser}, #{operateTime}, #{className}, #{methodName}, #{methodParams}, #
{returnValue}, #{costTime});")
    public void insert(OperateLog log);

}

```

3.4.2 编码实现

- 自定义注解@Log

```

/**
 * 自定义Log注解
 */
@Target({ElementType.METHOD})
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Log {

}

```

- 修改业务实现类，在增删改业务方法上添加@Log注解

```

/**
 * 部门管理Controller
 */
@Slf4j
@RequestMapping("/depts")
@RestController
public class DeptController {

    @Autowired
    private DeptService deptService;

    @GetMapping
    public Result list() throws Exception {
        List<Dept> deptList = deptService.list();
        return Result.success(deptList);
    }

    /**
     * 根据ID删除部门 - 简单参数接收: 方式三 (请求参数名与方法形参名称一致)
     */
}

```



```

@Log
@DeleteMapping
public Result delete(Integer id){
    log.info("根据ID删除部门: {}", id);
    deptService.delete(id);
    return Result.success();
}

/**
 * 添加部门 - json格式参数接收
 */
@Log
@PostMapping
public Result add(@RequestBody Dept dept){
    log.info("添加部门: {}", dept);
    deptService.add(dept);
    return Result.success();
}

/**
 * 根据ID查询部门数据
 * @return
 */
@GetMapping("/{id}")
public Result getInfo(@PathVariable Integer id){
    log.info("根据ID查询部门数据: {}", id);
    Dept dept = deptService.getInfo(id);
    return Result.success(dept);
}

/**
 * 修改部门数据
 */
@Log
@PutMapping
public Result update(@RequestBody Dept dept){
    log.info("修改部门数据: {}", dept);
    deptService.update(dept);
    return Result.success();
}
}

```

以同样的方式，修改 `EmpController` 业务类

- 定义切面类，完成记录操作日志的逻辑

```

@Slf4j
@Component
@Aspect //切面类

```

```

public class LogAspect {

    @Autowired
    private HttpServletRequest request;

    @Autowired
    private OperateLogMapper operateLogMapper;

    @Around("@annotation(com.itheima.anno.Log)")
    public Object recordLog(ProceedingJoinPoint joinPoint) throws Throwable {
        //操作人ID - 当前登录员工ID
        //获取请求头中的jwt令牌, 解析令牌
        String jwt = request.getHeader("token");
        Claims claims = JwtUtils.parseJWT(jwt);
        Integer operateEmpId = (Integer) claims.get("id");

        //操作时间
        LocalDateTime operateTime = LocalDateTime.now();

        //操作类名
        String className = joinPoint.getTarget().getClass().getName();

        //操作方法名
        String methodName = joinPoint.getSignature().getName();

        //操作方法参数
        Object[] args = joinPoint.getArgs();
        String methodParams = Arrays.toString(args);

        long begin = System.currentTimeMillis();
        //调用原始目标方法运行
        Object result = joinPoint.proceed();
        long end = System.currentTimeMillis();

        //方法返回值
        String returnValue = result.toString();

        //操作耗时
        Long costTime = end - begin;

        //记录操作日志
        OperateLog operateLog = new
        OperateLog(null, operateEmpId, operateTime, className, methodName, methodParams, returnValue, costTime);
        operateLogMapper.insert(operateLog);

        log.info("AOP记录操作日志: {}", operateLog);

        return result;
    }
}

```

代码实现细节：获取request对象，从请求头中获取到jwt令牌，解析令牌获取出当前用户的id。

重启SpringBoot服务，测试操作日志记录功能：

打开浏览器，针对于员工的数据、部门的数据进行增删改之后。我们打开数据库表结构可以来看一下：

operate_log							
WHERE ORDER BY							
	id	operate_user	operate_time	class_name	method_name	method_params	return_value
1	1	2	2023-12-09 14:44:04	com.itheima.controller.EmpController	update	[Emp(id=28, username=lizhong,...	Result(code=1, msg=null, data=
2	2		2023-12-09 14:44:10	com.itheima.controller.EmpController	delete	[[28]]	Result(code=1, msg=null, data=
3	3	2	2023-12-09 14:44:13	com.itheima.controller.DeptControll...	delete	[7]	Result(code=1, msg=null, data=
4	4	2	2023-12-09 14:44:16	com.itheima.controller.DeptControll...	add	[Dept(id=null, name=111, crea...	Result(code=1, msg=null, data=
5	5	2	2023-12-09 14:44:18	com.itheima.controller.DeptControll...	update	[Dept(id=8, name=1112, create...	Result(code=1, msg=null, data=
6	6	2	2023-12-09 14:44:20	com.itheima.controller.DeptControll...	delete	[8]	Result(code=1, msg=null, data=
7	7	2	2023-12-09 14:46:53	com.itheima.controller.DeptControll...	delete	[6]	Result(code=1, msg=null, data=

我们会看到，在数据库表中，就清晰的记录了谁、什么时间点、调用了哪个类的哪个方法、传入了什么参数、返回了什么数据，都清晰的记录在数据库中了。

到此，基于AOP记录操作日志的这个案例，我们已经完成了。