1、代码问题分析

1）**代码缺陷：附加功能**代码**重复，分散在**各个**业务功能方法中**，冗余！且不方便统一维护！

2）**解决思路：解耦**。把附加功能从业务功能代码中抽取出来。

将重复的代码**统一提取**，并且【**动态插入**】到每个业务方法！

3）**技术：代理模式**

2、代理模式

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。

1）**静态代理：可以**实现**解耦**，但是**代码都写死了**，完全不具备灵活性；

2）**动态代理：**1> **JDK 动态代理：**JDK 原生的实现方式，需要被代理的**目标类必须实现接口！**他会**根据**目标类的接口**动态生成一个代理对象**。代理对象和目标对象**有相同的接口**。（拜把子）2> **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。（认干爹）

3、面向切面编程思维（AOP）

AOP 可以说是 OOP（面向对象编程）的补充和完善。**OOP** 引入封装、继承、多态等概念来**建立一种对象层次结构**，用于**模拟公共行为**的一个集合。不过 OOP 允许开发者**定义纵向的关系**，但**并不适合定义横向的关系**，例如日志功能。

日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，**这种散布在各处的无关的代码**被称为**横切**（cross cutting），在 OOP 设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP 技术利用一种称为**“横切”的技术**，剖解开封装的对象内部，并将那些**影响了多个类的公共行为封装到一个可重用模块**，并将其命名为**“Aspect”**，即切面。所谓"切面"，简单说就是**那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来**，便于减少系统的重复代码，降低模块之间的耦合度，使用 AOP，**可以在不修改原来代码的基础上添加新功能**。

4、AOP 思想主要的应用场景

AOP（面向切面编程）是一种编程范式，它**通过将通用的横切关注点（如日志、事务、权限控制等）与业务逻辑分离**，使得代码更加清晰、简洁、易于维护。AOP 可应用于各种场景，以下是一些常见的 AOP 应用场景：

1）**日志记录**：在系统中记录日志是非常重要的，可以使用 AOP 来实现日志记录的功能，可以在**方法执行前**、**执行后**或**异常抛出时**记录日志。

2）**事务处理**：在数据库操作中使用事务可以保证数据的一致性，可以使用 AOP 来实现事务处理的功能，可以在**方法开始前**开启事务，在**方法执行完毕后**提交或回滚事务。

3）**安全控制**：在系统中包含某些需要安全控制的操作，如登录、修改密码、授权等，可以使用 AOP 来实现安全控制的功能。可以在**方法执行前**进行权限判断，**如果用户没有权限，则抛出异常或转向到错误页面**，以防止未经授权的访问。

4）**性能监控**：在系统运行过程中，有时需要对某些方法的性能进行监控，以找到系统的瓶颈并进行优化。可以使用 AOP 来实现性能监控的功能，可以在**方法执行前**记录时间戳，**在方法执行完毕后**计算方法执行时间并输出到日志中。

5）**异常处理**：系统中可能出现各种异常情况，如空指针异常、数据库连接异常等，可以使用 AOP 来实现异常处理的功能，在**方法执行过程中**，如果出现异常，则进行异常处理（如记录日志、发送邮件等）。

6）**缓存控制**：在系统中有些数据可以缓存起来以提高访问速度，可以使用 AOP 来实现缓存控制的功能，可以在**方法执行前**查询缓存中是否有数据，如果有则返回，**否则**执行方法并**将方法返回值存入缓存中**。

7）**动态代理**：AOP 的实现方式之一是通过动态代理，可以代理某个类的所有方法，用于实现各种功能。

5、AOP 术语名词介绍

1）**横切关注点**

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**。

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

2）**通知(增强)**

每个横切关注点上**要做的事情都需要写一个方法（通知方法）**来实现。

- **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）**连接点 joinpoint**（目标方法）

指**那些被拦截到的点**。在 Spring 中可以被动态代理拦截目标类的方法。

4）**切入点 pointcut**（表达式定位目标方法）

指**定位连接点的方式**，或者可以理解成**被选中的连接点**。是一个表达式，比如：execution(*com.spring.service.impl..(..))。**符合条件的每个方法**都是一个具体的**连接点**。

5）**切面 aspect：切入点**和**通知**的结合**组成**一个**切面类**。（是一个类）

6）**目标 target**：被代理的目标对象。

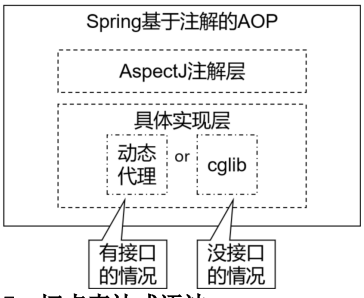
7）**代理 proxy**：向目标对象应用通知之后创建的代理对象。

8）**织入 weave**：指**把通知应用到目标上，生成代理对象**的过程。可以在**编译期**织入，也可以在**运行期**织入，**Spring 采用后者**（运行期织入）。

【注】Spring AOP 框架，**基于 AOP 编程思维，封装动态代理技术，简化动态代理技术**实现的框架。**SpringAOP 内部**帮助我们**实现动态代理**，我们只需写少量的配置，指定生效范围即可，即可完成面向切面思维编程的实现。

6、Spring AOP 底层技术组成

AspectJ 是早期的 AOP 实现框架，SpringAOP 借用了 AspectJ 中的 AOP 注解。



7、切点表达式语法

AOP 切点表达式作用是一种用于指定切点的语言，它可以**通过定义匹配规则，来选择需要被切入的目标对象**。

1）**第一位：*execution( )*** 固定开头；

2）**第二位：**方法***访问修饰符***；

3）**第三位：**方法***返回值***；

【注】：不考虑访问修饰符和返回值时，**不是~~execution(**)~~**，而是 **execution(*)** == 你只要不考虑返回值或者不考虑访问修饰符就相当于全部不考虑了。

4）**第四位：**指定***包的地址***；

1>**固定的包：**com.atguigu.api|service|dao；

2>**单层的任意命名：** com.atguigu.*， 包括 com.atguigu.api com.atguigu.dao；（* 指任意一层的任意命名）

3> **任意层任意命名：** com..， 包括 com.atguigu.api.erdaye； com.a.a.a.a.a；（..指任意层任意命名用在包上）

【注】**..不能用作包开头**，即 ~~public int ..~~ **找任何包下**：*..；**找 com 包下任何层**：com..；

5）**第五位：**指定***类名称***

1> **固定**名称：UserService

2> **任意**类名：*

3> **部分任意：**com..service.impl.*Impl

4> **任意包任意类**：*..*

6）**第六位：**指定***方法名称***

语法和类名一致；

任意访问修饰符，任意类的任意方法：* *.*.*

7)**第七位：*方法参数***

1> **具体值：**(String,int) != (int,String)

2> **没有参数** ()

3> **模糊值：**任意参数/有/没有 (..)

..任意参数的意思

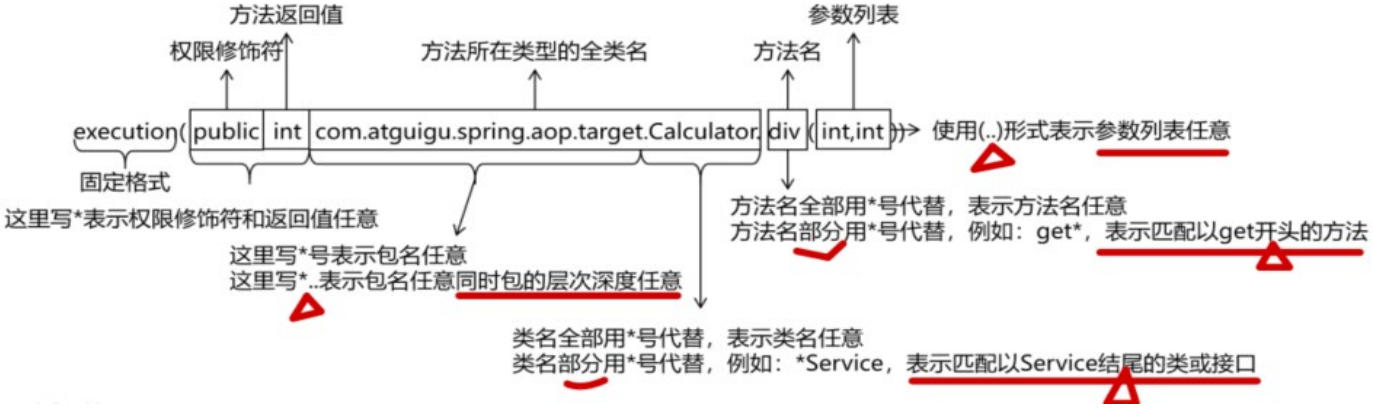
4> **部分具体和模糊：**

> **第一个**参数是字符串的方法 (String..)

> **最后一个**参数是字符串 (..String)

> 字符串**开头**，int **结尾** (String..int)

> **包含** int 类型(..int..)



8、获取通知细节信息

1）JointPoint 接口

通知方法声明 JoinPoint **类型形参**，获取方法签名、传入实参等信息。

要点 1：JoinPoint 接口通过 **getSignature()方法**获取**目标方法的签名**（方法声明时的完整信息）；

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);
}
```

2）方法返回值

在**返回通知中**，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）异常对象捕捉

在**异常通知中**，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());
}
```

9、重用（提取）切点表达式

原因：增强方法的切点表达式相同，出现了冗余，如果需要切换也不方便统一维护，我们可以将切点提取，**在增强上进行引用即可**。

1）同一类内部引用提取

```
// 切入点表达式重用
@Pointcut("execution(public int com.atguigu.aop.api.Calculator.add(int,int)))")
public void declarPointCut() {}
```

注意：提取切点注解使用@Pointcut(切点表达式)，需要添加到一个无参数无返回值方法上即可！

引用

```
@Before(value = "declarPointCut()")
public void printLogBeforeCoreOperation(JoinPoint joinPoint) {
```

2）不同类中引用

不同类在引用切点，只需要添加**类的全限定符+方法名**即可。

```
@Before(value = "com.atguigu.spring.aop.aspect.LogAspect.declarPointCut()")
public Object roundAdvice(ProceedingJoinPoint joinPoint) {
```

3）切点统一管理

建议：将切点表达式统一存储到一个类中进行集中管理和维护！

```
@Component
public class AtguiguPointCut {

    @Pointcut(value = "execution(public int *..Calculator.sub(int,int))")
    public void atguiguGlobalPointCut(){}

    @Pointcut(value = "execution(public int *..Calculator.add(int,int))")
    public void atguiguSecondPointCut(){}

    @Pointcut(value = "execution(* *..*Service.*(..))")
    public void transactionPointCut(){}
}
```

10、环绕通知

环绕通知对应整个 try...catch...finally 结构，包括前面四种通知的所有功能。

```
// 使用@Around注解标明环绕通知方法
@Around(value = "com.atguigu.aop.aspect.AtguiguPointCut.transactionPointCut()")

public Object manageTransaction(

        // 通过在通知方法形参位置声明ProceedingJoinPoint类型的形参，
        // Spring会将这个类型的对象传给我们
        ProceedingJoinPoint joinPoint) {

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

    // 通过ProceedingJoinPoint对象获取目标方法的签名对象
    Signature signature = joinPoint.getSignature();

    // 通过签名对象获取目标方法的方法名
    String methodName = signature.getName();

    // 声明变量用来存储目标方法的返回值
    Object targetMethodReturnValue = null;

    try {

        // 在目标方法执行前：开启事务（模拟）
        log.debug("[AOP 环绕通知] 开启事务，方法名：" + methodName + "，参数列表：" +
Arrays.asList(args));

        // 过ProceedingJoinPoint对象调用目标方法
        // 目标方法的返回值一定要返回给外界调用者
        targetMethodReturnValue = joinPoint.proceed(args);

        // 在目标方法成功返回后：提交事务（模拟）
        log.debug("[AOP 环绕通知] 提交事务，方法名：" + methodName + "，方法返回值：" +
targetMethodReturnValue);

    }catch (Throwable e){

        // 在目标方法抛异常后：回滚事务（模拟）
        log.debug("[AOP 环绕通知] 回滚事务，方法名：" + methodName + "，异常：" +
e.getClass().getName());

    }finally {

        // 在目标方法最终结束后：释放数据库连接
        log.debug("[AOP 环绕通知] 释放数据库连接，方法名：" + methodName);

    }

    return targetMethodReturnValue;
}
```

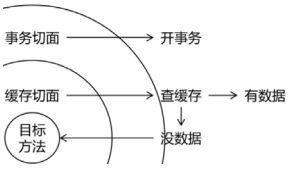
11、切面优先级设置

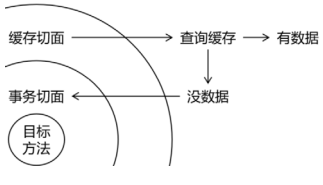
1）相同目标方法上同时存在多个切面时，**切面的优先级**控制**切面的内外嵌套顺序**。（优先级**高**的切面：**外面**；优先级**低**的切面：**里面**）

2）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。

【例】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。





Spring AOP 基于注解方式实现

1）加入依赖

```
<!-- spring-aspects会帮我们传递过来aspectjweaver -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>6.0.6</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>6.0.6</version>
</dependency>
```

2、准备接口

```
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);

}
```

3）纯净接口实现类

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

实现计算接口,单纯添加 + - * / 实现！不掺杂其他功能！

```
@Component
public class CalculatorPureImpl implements Calculator {

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

        int result = i + j;

        return result;
    }
```

4）声明切面类（切入点 + 通知方法）

```
package com.atguigu.advice;

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

// @Aspect表示这个类是一个切面类
@Aspect
// @Component注解保证这个切面类能够放入IOC容器
@Component
public class LogAspect {

    // @Before注解：声明当前方法是前置通知方法
    // value属性：指定切入点表达式，由切入点表达式控制当前通知方法
    @Before(value = "execution(public int com.atguigu.proxy.CalculatorPureImpl.add(int,int))")
    public void printLogBeforeCore() {
        System.out.println("[AOP前置通知] 方法开始了");
    }

    @AfterReturning(value = "execution(public int com.atguigu.proxy.CalculatorPureImpl.add(int,int))")
    public void printLogAfterSuccess() {
        System.out.println("[AOP返回通知] 方法成功返回了");
    }

    @AfterThrowing(value = "execution(public int com.atguigu.proxy.CalculatorPureImpl.add(int,int))")
    public void printLogAfterException() {
        System.out.println("[AOP异常通知] 方法抛异常了");
    }

    @After(value = "execution(public int com.atguigu.proxy.CalculatorPureImpl.add(int,int))")
    public void printLogFinallyEnd() {
        System.out.println("[AOP后置通知] 方法最终结束了");
    }

}
```

5）开启 aspectj 注解支持

1> xml 方式

```
<?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">

    <!-- 进行包扫描-->
    <context:component-scan base-package="com.atguigu" />
    <!-- 开启aspectj框架注解支持-->
    <aop:aspectj-autoproxy />
</beans>
```

2> 配置类方式

```
@Configuration
@ComponentScan(basePackages = "com.atguigu")
//作用等于 <aop:aspectj-autoproxy /> 配置类上开启 Aspectj注解支持!
@EnableAspectJAutoProxy
public class MyConfig {
}
```

6）测试效果

```
//@SpringJUnitConfig(locations = "classpath:spring-aop.xml")
@SpringJUnitConfig(value = {MyConfig.class})
public class AopTest {

    @Autowired
    private Calculator calculator;

    @Test
    public void testCalculator(){
        calculator.add(1,1);
    }
}
```

【注意】springboot 中 AOP 整合配置

导入依赖 spring-boot-starter-aop，直接使用 aop 注解

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

```
@Component
@Aspect
public class LogAdvice {

    @Before("execution(* com..service.*.*(..))")
    public void before(JoinPoint joinPoint){
        System.out.println("LogAdvice.before");
        System.out.println("joinPoint = " + joinPoint);
    }

}
```