第三阶段 Spring-03-AOP

回顾：

1. 依赖注入的各种配置和属性
2. 使用注解的方式实现依赖注入
3. 在Spring中使用JUnit测试

今天任务

1. AOP基本概念
2. 在Spring中使用AOP

教学目标

1. 掌握AOP基本原理
2. 掌握在Spring中使用AOP

一. AOP介绍

1. Aop介绍

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

AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"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，并有利于未来的可操作性和可维护性。

使用"横切"技术，AOP把软件系统分为两个部分：**核心关注点**和**横切关注点**。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。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. AOP核心概念

- 连接点(Joinpoint):

  特定点是程序执行的某一个特定位置,如类开始初始化前,类初始化后,类某一个方法调用前/调用后,方法抛出异常后,一个类或一段程序代码拥有一些具有边界性质的特定点,这写代码中的特定点就称为"连接点",Spring仅支持方法连接点,即仅能在方法调用前,方法调用后,方法抛出异常时,以及方法调用前后这些程序执行点织入增强.

- 切点(Pointcut)

  每个程序类都拥有对个连接点,如一个拥有两个方法的类,这两个方法都是连接点,即连接点是程序类中客观存在的事物,但在众多连接点中,如何定位某些感兴趣的连接点呢?AOP通过"切点"定位特定的连接点.

- 增强(Advice)

  增强是织入目标类连接点上的一段程序代码,在Spring中,增强不仅可以描述程序代码,还拥有另一个和连接点相关的信息,这便是执行点的方位,结合执行点的方位信息和切点信息,就可以找到特定的连接,正因为增强既包含了用于添加到目标连接点上的一段执行逻辑,又包含用于定位连接点的方位信息,所以Spring所提供的增强接口都是带方位名的.如BeforeAdvice,AfterReturningAdvice,ThrowsAdvice等.BeforeAdvice表示方法调用前的位置.而AfterReturningAdivce表示访问返回后的位置,所以只有结合切点和增强,才能确定特定的连接点并实施增强逻辑.

- 目标对象(Target)

  增强逻辑的织入目标类.如果没有AOP,那么目标业务类需要自己实现所有逻辑,如果使用AOP可以把一些非逻辑性代码通过AOP织入到主程序代码上.

- 引介(Introduction)

引介是一种特殊的增强,它为类添加一些属性和方法.这样,即使一个业务类原本没有实现某一个接口,通过AOP的引介功能,也可以动态地为该业务类添加接口的实现逻辑.让业务类成为这个接口的实现类.

- 织入(Weaving)

  织入是将增强添加到目标类具体链接点上的过程,AOP就像一台织布机,将目标类,增强,或者引介天衣无缝的编织到一起,我们不能不说"织入"这个词太精辟了,根据不同的实现技术,AOP有3种织入方式:

  - 编译期织入,这要求使用特殊的Java编译器.
  - 类装载期织入,这要求使用特殊的类装载器.
  - 动态代理织入,在运行期为目标类添加增强生成子类的方式.

    Spring采用动态代理织入,而AspectJ采用编译期织入和类装载器织入.

- 代理(Proxy)

  一个类被AOP织入增强后,就产生了一个结果类.它是融合了原类和增强逻辑的代理类,根据不同的代理方式,代理类既可能是和原类具有相同接口的类,也可能就是原类的子类,所以可以采用与调用原类相同的方法调用代理类.

- 切面(Aspect)

  切面由切点和增强(介入)组成,它既包括横切逻辑的定义,也包括链接点的定义,也包括链接点的定义,Spring AOP就是负责实施切面的框架,它将切面所定义的横切所定义的横切逻辑织入切面所指定的链接点中.

  AOP的工作重心在于如何将增强应用于目标对象的连接点中上,这里包含两项工作:

  第一,如何通过切点和增强定位到链接点上;

  第二,如何在增强中编写切面代码;

3. Spring AOP 基础知识

Spring的 AOP底层用到两种代理机制:

- JDK 的动态代理：针对实现了接口的类产生代理。
- CGlib 的动态代理：针对没有实现接口的类产生代理，应用的是底层的字节码增强的技术 生成当前类的子类对象

3.1 JDK动态代理实现

1. 创建接口和对应实现类

```
public interface UserService {

   void login();

  void loginOut();

}
```

```
//实现类
public class UserServiceImpl implements  UserService {

  public void login() {

     System.out.println("login方法触发");
  }

  public void loginOut() {

    System.out.println("loginOut方法触发");
  }
}
```

2. 创建动态代理类

```
public class PerformHandler implements InvocationHandler {

    private  Object target; //目标对象

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


    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //本方法中的其他输出输入增强
        System.out.println("方法触发了");
        //执行被代理类 原方法
        Object invoke = method.invoke(target, args);

        System.out.println("执行完毕了");

        return invoke;
    }
}
```

测试

```
 @Test
 public  void  test1(){
     //测试JDK动态代理技术
      UserService userService = new UserServiceImpl();
```

```
        PerformHandler performHandler = new PerformHandler(userService);

        userService = (UserService)
Proxy.newProxyInstance(userService.getClass().getClassLoader(),
                    userService.getClass().getInterfaces(),
                    performHandler
            );

        userService.login();

    }
```

测试结果: 在调用接口方法的前后都会添加代理类的方法!

3.2 CGlib实现代理

> 使用JDK创建代理有一个限制,它只能为接口创建代理实例.这一点可以从Proxy的接口方法newProxyInstance(ClassLoader loader,Class [] interfaces,InvocarionHandler h)中看的很清楚
>
> 第二个入参 interfaces就是需要代理实例实现的接口列表.
>
> 对于没有通过接口定义业务方法的类,如何动态创建代理实例呢? JDK动态代理技术显然已经黔驴技穷,CGLib作为一个替代者,填补了这一空缺.
>
> GCLib采用底层的字节码技术,可以为一个类创建子类,在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势志入横切逻辑.

1. 创建创建CGLib代理器

```
public class CglibProxy  implements MethodInterceptor{

    private Enhancer enhancer = new Enhancer();

    //设置被代理对象
    public  Object getProxy(Class clazz){

        enhancer.setSuperclass(clazz);
        enhancer.setCallback(this);

        return  enhancer.create();
    }


    public Object intercept(Object obj, Method method,
                            Object[] objects,
```

```
                                MethodProxy methodProxy) throws Throwable {

        System.out.println("CGLig代理之前之前");

        Object invoke = methodProxy.invokeSuper(obj,objects);

        System.out.println("CGLig代理之前之后");

        return invoke;
    }
}
```

测试

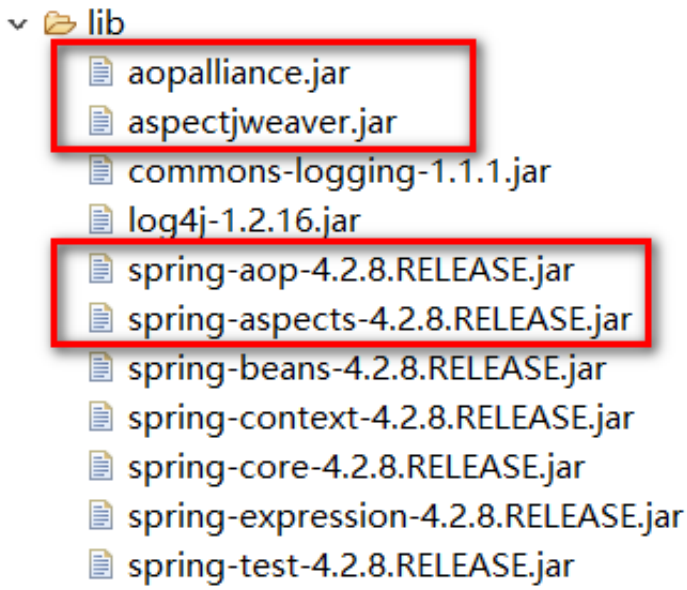
```
 @Test
 public  void test2(){
    //TODO CGlib实现
    CglibProxy cglibProxy = new CglibProxy();
    UserServiceImpl userService= (UserServiceImpl)
cglibProxy.getProxy(UserServiceImpl.class);
    userService.login();
 }
```

二. Spring中AOP开发

1. Spring 中 AOP 基于xml开发

1.1 项目准备

- 项目名: spring-03-aop
- 导入jar包:



pom文件添加

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>aopalliance</groupId>
  <artifactId>aopalliance</artifactId>
  <version>1.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.10</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-aspects -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>4.3.7.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>4.3.10.RELEASE</version>
</dependency>
<dependency>
```

```
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>4.3.11.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.10.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.10.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>4.3.10.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-expression</artifactId>
    <version>4.3.10.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.2</version>
  </dependency>

  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
  </dependency>

  <!-- https://mvnrepository.com/artifact/commons-dbcp/commons-dbcp -->
  <dependency>
    <groupId>commons-dbcp</groupId>
```

```
    <artifactId>commons-dbcp</artifactId>
    <version>1.4</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
    <version>1.5.7.RELEASE</version>
  </dependency>
```

- 引入日志：log4j.properties
- 创建配置文件：`applicationContext-aop.xml`
- 创建测试代码：`AopTest.java`

1.2 准备操作对象

先创建UserService接口:

```
public interface UserService {

    void save();
    void delete();
    void update();
    void select();

}
```

实现类

```
public class UserServiceImpl implements  UserService {

    public void save() {
        System.out.println("保存用户");
    }

    public void delete() {
        System.out.println("删除用户");
    }

    public void update() {
        System.out.println("更新用户");
    }

    public void select() {
        System.out.println("查找用户");
    }
}
```

1.3 增强类

```
public class MyAdivce {

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


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

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

     public  Object around(ProceedingJoinPoint point) throws Throwable {

         System.out.println("这是环绕通知之前部分!!");
         Object object = point.proceed(); //调用目标方法
         System.out.println("这是环绕通知之后的部分!!");

         return  object;
     }

     public  void afterException(){
         System.out.println("异常通知!");
     }

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

}
```

1.4 将增强织入目标对象(xml)

在applicationContext-aop.xml 中配置

注意:添加了 aop命名空间

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/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-4.2.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.2.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.2.xsd ">

    <bean name="userService" class="com.itqf.spring.service.UserServiceImpl" />

    <!--配置通知对象-->
    <bean name="myAdvice" class="com.itqf.spring.aop.MyAdivce" />

    <!-- 配置将增强织入目标对象-->
    <aop:config>

            <!--

             com.itqf.spring.service.UserServiceImpl

             1 2 3 4
             1: 修饰符  public/private/*  可忽略
             2: 返回值  String/../*
             3: 全限定类名  类名  ..代表不限层数  *ServiceImpl
             4: (..)
            -->

        <aop:pointcut id="pc" expression="execution(*
com.itqf.spring.service.*ServiceImpl.*
        (..))"/>

        <aop:aspect ref="myAdvice">
                 <aop:before method="before" pointcut-ref="pc" />
                 <aop:after-returning method="afterReturning" pointcut-ref="pc" />
                 <aop:around method="around" pointcut-ref="pc" />
                 <aop:after-throwing method="afterException" pointcut-ref="pc" />
                 <aop:after method="after" pointcut-ref="pc" />
        </aop:aspect>

    </aop:config>
```

```
</beans>
```

测试:

```
@Test
public  void test1(){
   //TODO 测试切面引入
   ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext-aop.xml");

   UserService userServiceImpl = context.getBean("userService",
UserService.class);

   userServiceImpl.delete();

}
```

2. Spring中的AOP基于注解开发

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

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

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

2.1 创建配置文件

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

配置目标对象,配置通知对象,开启注解

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/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-4.2.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.2.xsd
```

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

    <bean name="userService" class="com.itqf.spring.service.UserServiceImpl" />

    <!--配置通知对象-->
    <bean name="myAdvice" class="com.itqf.spring.aop.MyAdivce" />

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

</beans>
```

2.2 修改增强类MyAdvice

1. 在类上添加 `@Aspect` 注解
2. 可以将 `execution` 直接定义在通知方法名上，如下：

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

3. 也可以定义一个方法， 如下面的 `pc()` 方法，然后在通知方法上指定 `@Before("MyAdvice.pc()")`

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

    //前置通知
    //指定该方法是前置通知,并制定切入点
    @Before("MyAdvice.pc()")
    public void before(){
        System.out.println("这是前置通知!!");
    }
```

完整定义增强类

```
@Aspect
public class MyAdivce {
```

```
    /**
     //前置通知：目标方法运行之前调用
     //后置通知(如果出现异常不会调用)：在目标方法运行之后调用
     //环绕通知：在目标方法之前和之后都调用
     //异常拦截通知：如果出现异常，就会调用
     //后置通知(无论是否出现 异常都会调用)：在目标方法运行之后调用
     */
    @Pointcut("execution(* com.itqf.spring.service.*ServiceImpl.*(..))")
    public  void pc(){

    }


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

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

     public  Object around(ProceedingJoinPoint point) throws Throwable {

         System.out.println("这是环绕通知之前部分!!");
         Object object = point.proceed(); //调用目标方法
         System.out.println("这是环绕通知之后的部分!!");

         return  object;
     }

     public  void afterException(){
         System.out.println("异常通知!");
     }

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

}
```

课前默写

1. 依赖注入的配置和属性的含义
2. 使用注解的方式实现依赖注入
3. 在Spring中使用JUnit测试

作业

1. 使用Hibernate第三天作业的员工部门数据表
2. 完成两表的CRUD操作
3. 使用AOP完成项目的日志功能
4. 使用AOP完成在Hibernate的方法调用时自动开启连接，开启事务、提交事务（回滚事务）、关闭连接，以简化Hibernate业务代码（提示：使用around通知）

面试题

1. 什么是AOP，AOP有什么作用
2. 在Spring中什么地方使用了AOP
3. 在项目中什么时候用到了AOP，举例说明
4. 名词解释：切面、切入点、连接点、织入、通知