# 本篇是本人看完所有spring教程视频后的总结笔记

# Spring学习大纲

第一部分： spring框架的概述以及spring中基于XML的IOC（反转控制）配置

第二部分：Spring中基于注解的IOC和IOC案例

第三部分：spring中的AOP和基于XML以及注解的AOP（面向切面编程）配置

第四部分：spring中的JDBC Temlate以及Spring事物控制

### Spring学习第三天——Spring AOP及其实现

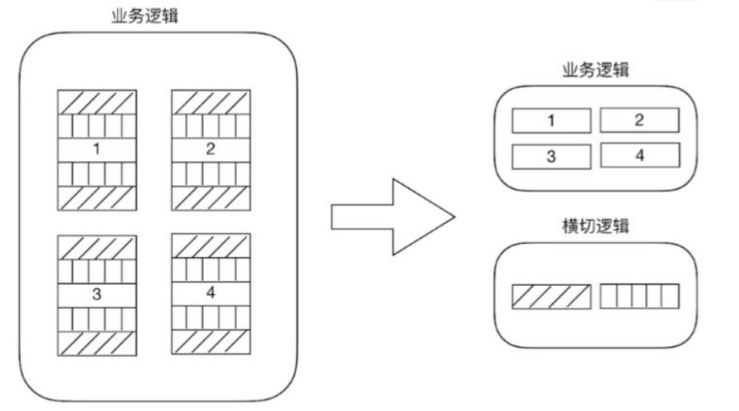
在这一部分学习内容中，主要的内容是理解Spring AOP的概念以及原理，同时学习SpringAOP的实现。主要实现方式有xml配置文件的实现以及注解实现AOP。

**1、AOP及动态代理**

* 1. **AOP是什么**

AOP：全称是Aspect Oriented Programming，即**面向切面编程**。就是在多个纵向的流程中将相同的横切逻辑代码抽取出来，将横切逻辑代码与业务逻辑代码分离，解决代码重复和耦合的问题，提高程序的可重用性。

简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用**动态代理**的技术，在不修改源码的基础上，对我们的已有方法进行增强。



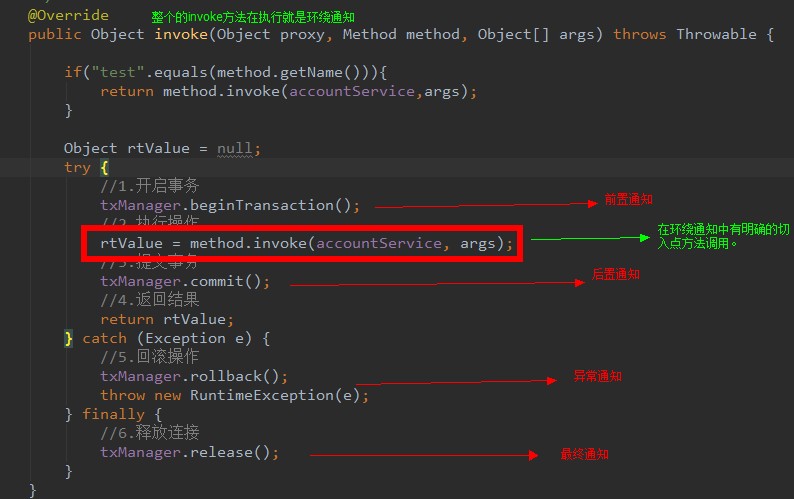
作用：在程序运行期间，不修改源码对已有方法进行增强。  
优势：减少重复代码、提高开发效率、维护方便。

**AOP术语：**

* Joinpoint(连接点): 被拦截到的方法.
* Pointcut(切入点): 我们对其进行增强的方法.
* Advice(通知/增强): 对切入点进行的增强操作

包括前置通知,后置通知,异常通知,最终通知,环绕通知

* Weaving(织入): 是指把增强应用到目标对象来创建新的代理对象的过程。
* Aspect(切面): 是切入点和通知的结合



* 1. **AOP解决问题实例**

1. @Override
2. public void saveAccount(Account account) {
3. try {
4. TransactionManager.beginTransaction();
5. accountDao.save(account); // 唯一的一行业务代码
6. TransactionManager.commit();
7. } catch (Exception e) {
8. TransactionManager.rollback();
9. e.printStackTrace();
10. }finally {
11. TransactionManager.release();
12. }
13. }
14. @Override
15. public void updateAccount(Account account) {
16. try {
17. TransactionManager.beginTransaction();
18. accountDao.update(account); // 唯一的一行业务代码
19. TransactionManager.commit();
20. } catch (Exception e) {
21. TransactionManager.rollback();
22. e.printStackTrace();
23. }finally {
24. TransactionManager.release();
25. }
26. }
27. @Override
28. public void deleteAccount(Integer accountId) {
29. try {
30. TransactionManager.beginTransaction();
31. accountDao.delete(accountId); // 唯一的一行业务代码
32. TransactionManager.commit();
33. } catch (Exception e) {
34. TransactionManager.rollback();
35. e.printStackTrace();
36. }finally {
37. TransactionManager.release();
38. }
39. }

**在上述代码中有**两个问题:

1、业务层方法变得臃肿了,里面充斥着很多重复代码.

2、业务层方法和事务控制方法耦合了. 若提交,回滚,释放资源中任何一个方法名变更,都需要修改业务层的代码。

可以用动态代理来解决这些问题。

**1.3 动态代理**

动态代理：

* 特点：字节码随用随创建，随用随加载。它与静态代理的区别也在于此。因为静态代理是字节码一上来就 创建好，并完成加载。 装饰者模式就是静态代理的一种体现。
* 作用：在不修改源码的基础上对方法增强

常用的动态代理分为两种

1. 基于接口的动态代理,使用JDK 官方的 Proxy 类,**要求被代理者至少实现一个接口**
2. 基于子类的动态代理,使用第三方的 CGLib库,**要求被代理类不能是final类.**

**1.3.1**基于接口的动态代理

* 如何创建代理对象：使用Proxy类中的newProxyInstance方法

1）涉及的类：jdk官方提供的 Proxy；

​ 2）要求：被代理类必须至少实现一个接口，如果没有则不能使用

​ 3）newProxyInstance方法的参数：

​ classLoader类加载器：它是用于加载代理对象字节码的，和被加载对象使用相同的类加载器

​ Class[] 字节码数组

​ InvocationHandler 用于提供增强的代码，通常情况下时匿名内部类，但不必须

**写法：**

接口名 新对象名 = (接口名)Proxy.newProxyInstance(

被代理的对象.getClass().getClassLoader(), // 被代理对象的类加载器,固定写法

被代理的对象.getClass().getInterfaces(), // 被代理对象实现的所有接口,固定写法

new InvocationHandler() { // 匿名内部类,通过拦截被代理对象的方法来增强被代理对象

/\* 被代理对象的任何方法执行时,都会被此方法拦截到

其参数如下:

proxy: 代理对象的引用,不一定每次都用得到

method: 被拦截到的方法对象

args: 被拦截到的方法对象的参数

返回值:

被增强后的返回值

\*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if("方法名".equals(method.getName())) {

// 增强方法的操作

rtValue = method.invoke(被代理的对象, args);

// 增强方法的操作

return rtValue;

}

}

});

1.3.2 基于子类的动态代理

* 基于子类的动态代理：

如何创建代理对象：使用Enhancer类中的create方法

1）涉及的类：Enhancer；提供者：cglib库

2）要求：被代理类不能是最终类

3）create()方法的参数：

​ class：字节码

​ 类加载器：它是用于加载代理对象字节码的，和被加载对象使用相同的类加载器

​ Callback： 用于提供增强的代码，通常情况下时匿名内部类，但不必须。我们一般写的是该接口的子 接口实现类

**写法：**

public class Client {

public static void main(String[] args) {

final Producer producer = new Producer();

Producer cglibProducer = (Producer) Enhancer.create(producer.getClass(), new MethodInterceptor() {

/\*\*

\* 执行被代理对象的任何方法都会经过该方法

\* @param proxy

\* @param method

\* @param args

\* @param methodProxy

\* @return

\* @throws Throwable

\*/

@Override

public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {

Object returnValue = null;

Float money = (Float)args[0];

if("saleProduct".equals(method.getName())) {

returnValue = method.invoke(producer, money\*0.8f);

}

return returnValue;

}

});

**1.4 小结**

在这一部分中，首先学习了什么是AOP,讨论了当前代码会出现的各种问题，并给出了非SpringAOP的解决方法，用动态代理来解决。同时，动态代理分为基于接口的动态代理与基于子类的动态代理。

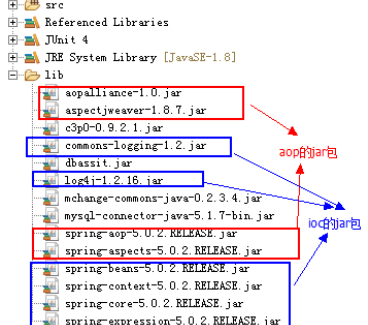
**2、基于xml的AOP配置**

在这一部分中，将会学习使用Spring的AOP来完成代理的功能。

**2.1 必要准备**

使用了与springIOC例子中相同的实体类，业务层和持久层代码。

**同时，导入AOP的jar包。**



**2.2配置通知类**

public class Logger {  
public void printLog(){  
 System.*out*.println("Logger类中的pringLog方法开始记录日志了。。。");  
 }  
}

该通知类用于打印日志：计划让其在切入点方法执行之前执行（切入点方法就是业务层方法）

**2.3 配置xml文件**

**1、首先，**在bean.xml中引入约束并将通知类注入Spring容器中

<?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: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/aop

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

<!--通知类-->

<bean id="logger" class=" com.itheima.utils.Logger"></bean>

</beans>

2、使用<aop:config>标签声明AOP配置,所有关于AOP配置的代码都写在<aop:config>标签内

<aop:config>

<!-- AOP配置的代码都写在此处 -->

</aop:config>

3、使用<aop:aspect>标签配置切面,其属性如下

1、id: 指定切面的id

2、ref: 引用通知类的id

<aop:config>

<aop:aspect id="logAdvice" ref="logger">

<!--配置通知的类型要写在此处-->

</aop:aspect>

</aop:config>

**4、**使用<aop:pointcut>标签配置切入点表达式,指定对哪些方法进行增强,其属性如下

id: 指定切入点表达式的id

expression: 指定切入点表达式

<aop:pointcut id="pt1" expression="execution(\* com.itheima.service.impl.\*.\*(..))"></aop:pointcut>

**5、**使用<aop:xxx>标签配置对应类型的通知方法

* 其属性如下:
  + method: 指定通知类中的增强方法名.
  + ponitcut-ref: 指定切入点的表达式的id
  + poinitcut: 指定切入点表达式

其中pointcut-ref和pointref属性只能有其中一个

* 具体的通知类型:
  + <aop:before>: 配置前置通知,指定的增强方法在切入点方法之前执行.
  + <aop:after-returning>: 配置后置通知,指定的增强方法在切入点方法正常执行之后执行.
  + <aop:after-throwing>: 配置异常通知,指定的增强方法在切入点方法产生异常后执行.
  + <aop:after>: 配置最终通知,无论切入点方法执行时是否发生异常,指定的增强方法都会最后执行.
  + <aop:around>: 配置环绕通知,可以在代码中手动控制增强代码的执行时机.

代码:

* <aop:config>
* <aop:aspect id="logAdvice" ref="logger">
* <!--指定切入点表达式-->
* <aop:pointcut expression="execution(\* com.itheima.service.impl.\*.\*(..))" id="pt1"></aop:pointcut>
* <!--配置各种类型的通知-->
* <aop:before method="printLogBefore" pointcut-ref="pt1"></aop:before>
* <aop:after-returning method="printLogAfterReturning" pointcut-ref="pt1"></aop:after-returning>
* <aop:after-throwing method="printLogAfterThrowing" pointcut-ref="pt1"></aop:after-throwing>
* <aop:after method="printLogAfter" pointcut-ref="pt1"></aop:after>
* </aop:aspect>
* </aop:config>

**2.4 环绕通知的配置**

环绕通知是spring框架为我们提供的一种可以在代码中手动控制增强方法何时执行的方式。

Spring框架为我们提供了一个接口：ProceedingJoinPoint。该接口有一个方法proceed()，此方法就相当于明确调用切入点方法。该接口可以作为环绕通知的方法参数，在程序执行时，spring框架会为我们提供该接口的实现类供我们使用。该方法与自己写Handler类的方法类似，**较为灵活**。

* ProceedingJoinPoint对象的getArgs()方法返回被拦截的参数

2、ProceedingJoinPoint对象的proceed()方法执行被拦截的方法

代码：

// 环绕通知方法,返回Object类型

public Object printLogAround(ProceedingJoinPoint pjp) {

Object rtValue = null;

try {

Object[] args = pjp.getArgs(); //获取被拦截方法参数

printLogBefore(); // 执行前置通知

rtValue = pjp.proceed(args);// 执行被拦截方法

printLogAfterReturn(); // 执行后置通知

}catch(Throwable e) {

printLogAfterThrowing(); // 执行异常通知

}finally {

printLogAfter(); // 执行最终通知

}

return rtValue;

}

同时在xml中，使用<aop:around>:配置环绕通知,可以在代码中手动控制增强代码的执行时机。

<aop:aspect id="logAdvice" ref="logger">  
 <aop:around method="aroundPrintLog" pointcut-ref="pt1"></aop:around>  
</aop:aspect>

**注意：环绕通知一般单独使用**

**2.5切入点表达式**

execution:匹配方法的执行(常用)

execution(表达式)

表达式语法：execution([修饰符] 返回值类型 包名.类名.方法名(参数))

写法说明：

全匹配方式： public void com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

访问修饰符可以省略 void com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

返回值可以使用\*号，表示任意返回值 \* com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

包名可以使用\*号，表示任意包，但是有几级包，需要写几个\* \* \*.\*.\*.\*.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

使用..来表示当前包，及其子包 \* com..AccountServiceImpl.saveAccount(com.itheima.domain.Account)

类名可以使用\*号，表示任意类 \* com..\*.saveAccount(com.itheima.domain.Account)

方法名可以使用\*号，表示任意方法 \* com..\*.\*( com.itheima.domain.Account)

参数列表可以使用\*，表示参数可以是任意数据类型，但是必须有参数 \* com..\*.\*(\*)

参数列表可以使用..表示有无参数均可，有参数可以是任意类型 \* com..\*.\*(..)

全通配方式： \* \*..\*.\*(..)

**注意： 通常情况下，我们都是对业务层的方法进行增强，所以切入点表达式都是切到业务层实现类。 execution(\* com.itheima.service.impl.\*.\*(..))**

**3、基于注解的 AOP 配置**

**3.1 基本准备**

**基于注解的AOP仍然是使用IOC的实体类与注解，不同的是需要在配置文件中先导入context的名称空间。同时，在xml中指定创建容器要扫描的包。**

**<** context:component-scan base-package="com.itheima"></context:component-scan>

**3.2 通过注解进行通知配置**

**3.2.1用于声明切面的注解**

@Aspect: 声明当前类为通知类,该类定义了一个切面.相当于xml配置中的<aop:aspect>标签

@Component("logger")

@Aspect

public class Logger {

// ...

}

3.2.2用于声明通知的注解

* @Before: 声明该方法为前置通知.相当于xml配置中的<aop:before>标签
* @AfterReturning: 声明该方法为后置通知.相当于xml配置中的<aop:after-returning>标签
* @AfterThrowing: 声明该方法为异常通知.相当于xml配置中的<aop:after-throwing>标签
* @After: 声明该方法为最终通知.相当于xml配置中的<aop:after>标签

属性:

* value: 用于指定切入点表达式或切入点表达式的引用
* @Component("logger")
* @Aspect //表示当前类是一个通知类
* public class Logger {
* // 配置前置通知
* @Before("execution(\* cn. itheima.service.impl.\*.\*(..))")
* public void printLogBefore(){
* System.out.println("前置通知Logger类中的printLogBefore方法开始记录日志了。。。");
* }
* // 配置后置通知
* @AfterReturning("execution(\* cn.maoritian.service.impl.\*.\*(..))")
* public void printLogAfterReturning(){
* System.out.println("后置通知Logger类中的printLogAfterReturning方法开始记录日志了。。。");
* }
* // 配置异常通知
* @AfterThrowing("execution(\* cn. itheima.service.impl.\*.\*(..))")
* public void printLogAfterThrowing(){
* System.out.println("异常通知Logger类中的printLogAfterThrowing方法开始记录日志了。。。");
* }
* // 配置最终通知
* @After("execution(\* cn. itheima.service.impl.\*.\*(..))")
* public void printLogAfter(){
* System.out.println("最终通知Logger类中的printLogAfter方法开始记录日志了。。。");
* }
* }

#### 3.2.3用于指定切入点表达式的注解

@Pointcut: 指定切入点表达式,其属性如下:

* value: 指定表达式的内容

@Pointcut注解没有id属性,通过调用被注解的方法获取切入点表达式.

// 配置切入点表达式

@Pointcut("execution(\* cn.itheima.service.impl.\*.\*(..))")

private void pt1(){}

// 通过调用被注解的方法获取切入点表达式

@Before("pt1()")

public void printLogBefore(){

System.out.println("前置通知Logger类中的printLogBefore方法开始记录日志了。。。");

}

3.2.5 环绕通知的注解

@Around: 声明该方法为环绕通知.相当于xml配置中的<aop:around>标签。

代码

// 配置环绕通知

@Around("execution(\* cn. itheima.service.impl.\*.\*(..))")

public Object aroundPringLog(ProceedingJoinPoint pjp){

Object rtValue = null;

try{

Object[] args = pjp.getArgs();

printLogBefore(); // 执行前置通知

rtValue = pjp.proceed(args); // 执行切入点方法

printLogAfterReturning(); // 执行后置通知

return rtValue;

}catch (Throwable t){

printLogAfterThrowing(); // 执行异常通知

throw new RuntimeException(t);

}finally {

printLogAfter(); // 执行最终通知

}

}

## 3.3 纯注解配置AOP

在Spring配置类前添加@EnableAspectJAutoProxy注解,可以使用纯注解方式配置AOP

@Configuration

@ComponentScan(basePackages="cn.itheima")

@EnableAspectJAutoProxy // 允许AOP

public class SpringConfiguration {

// 具体配置

//...

}

**注意：在使用注解配置AOP时,会出现一个bug. 四个通知的调用顺序依次是:前置通知,最终通知,后置通知. 这会导致一些资源在执行最终通知时提前被释放掉了,而执行后置通知时就会出错.**

**4、总结**

在这一部分学习内容中，主要的内容是理解Spring AOP的概念以及原理，同时学习SpringAOP的实现。主要实现方式有xml配置文件的实现以及注解实现AOP。通知可以用前置、后置等等通知来配置切入点方法，也可以通过直接使用环绕通知来一次性配置所有通知。同时，可以使用注解方法更方便的配置通知，但是需要注意的一点是使用全注解方法配置通知会产生通知顺序出错的bug。