





# 注解说明

## 注册注解处理器

• 方式一:bean

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

• 方式二: 命名空间<context:annotation-config />

<context:annotationconfig /> 将隐式地向Spring 容器注册AutowiredAnnotationBeanPostProcessor、CommonAnnotationBeanPostProcessor 、 PersistenceAnnotationBeanPostProcessor 以及RequiredAnnotationBeanPostProcessor 这4 个BeanPostProcessor 。

• 方式三: 命名空间<context:component-scan />

如果要使注解工作，则必须配置component-scan ，实际上不需要再配置annotation-config。base-package 属性指定了需要扫描的类包，类包及其递归子包中所有的类都会被处理。还允许定义过滤器将基包下的某些类纳入或排除。

• Spring 支持以下4 种类型的过滤方式：

• 注解 org.example.SomeAnnotation 将所有使用SomeAnnotation 注解的类过滤出来

• 类名指定 org.example.SomeClass 过滤指定的类

• 正则表达式 com.kedacom.spring.annotation.web..\* 通过正则表达式过滤一些类

• AspectJ 表达式 org.example..\*Service+ 通过AspectJ 表达式过滤一些类

• 正则表达式的过滤方式举例：

<context:component-scan base-package="com.casheen.spring.annotation">

<context:exclude-filtertype="regex" expression="com.casheen.spring.annotation.web..\*"/>  
 </context:component-scan>

• 注解的过滤方式举例：

<context:component-scan base-package="com.netqin" >

<context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  
 <context:include-filter type="annotation" expression="org.springframework.stereotype.Service"/>  
 <context:include-filter type="annotation" expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

## 启用Spring MVC 注解

• 启动Spring MVC 的注解功能，完成请求和注解POJO 的映射

• <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"/>

注解举例：

@Controller

       • 例如

         @Controller  
         public class SoftCreateController extends SimpleBaseController {}

       • 或者  
         @Controller("softCreateController")

       • 说明

         @Controller 负责注册一个bean 到spring 上下文中，bean 的ID 默认为类名称开头字母小写

@Service

      • 例如

        @Service  
        public class SoftCreateServiceImpl implements ISoftCreateService {}

      • 或者  
        @Service("softCreateServiceImpl")

      • 说明

        @Service 负责注册一个bean 到spring 上下文中，bean 的ID 默认为类名称开头字母小写

@Autowired

      • 例如

         @Autowired  
         private ISoftPMService softPMService;

       • 或者

 　　    @Autowired(required=false)

     　　private ISoftPMService softPMService = new SoftPMServiceImpl();

      • 说明

   　　@Autowired 根据bean 类型从spring 上线文中进行查找，注册类型必须唯一，否则报异常。与@Resource 的区别在于，@Resource 允许通过bean 名称或bean 类型两种方式进行查找@Autowired(required=false) 表示，如果spring 上下文中没有找到该类型的bean 时， 才会使用new SoftPMServiceImpl();

@Autowired 标注作用于 Map 类型时，如果 Map 的 key 为 String 类型，则 Spring 会将容器中所有类型符合 Map 的 value 对应的类型的 Bean 增加进来，用 Bean 的 id 或 name 作为 Map 的 key。

　　@Autowired 还有一个作用就是，如果将其标注在 BeanFactory 类型、ApplicationContext 类型、ResourceLoader 类型、ApplicationEventPublisher 类型、MessageSource 类型上，那么 Spring 会自动注入这些实现类的实例，不需要额外的操作。

@RequestMapping

• 类

　　@Controller   
　　@RequestMapping("/bbtForum.do")

　　public class BbtForumController {  
 @RequestMapping(params = "method=listBoardTopic")

　　 public String listBoardTopic(int topicId,User user) {}  
　　}

• 方法

@RequestMapping("/softpg/downSoftPg.do")  
 @RequestMapping(value="/softpg/ajaxLoadSoftId.do",method = POST)  
 @RequestMapping(value = "/osu/product/detail.do", params = { "modify=false" }, method =POST)

• 说明  
    @RequestMapping 可以声明到类或方法上

• 参数绑定说明  
　　如果我们使用以下的 URL 请求：

<http://localhost/bbtForum.do?method=listBoardTopic&topicId=1&userId=10&userName=tom>

　　topicId URL 参数将绑定到 topicId 入参上，而 userId 和 userName URL 参数将绑定到 user 对象的 userId 和 userName 属性中。和 URL 请求中不允许没有 topicId 参数不同，虽然 User 的 userId 属性的类型是基本数据类型，但如果 URL 中不存在 userId 参数，Spring 也不会报错，此时 user.userId 值为 0 。如果 User 对象拥有一　　个 dept.deptId 的级联属性，那么它将和 dept.deptId URL 参数绑定。

@RequestParam

• 参数绑定说明  
　　@RequestParam("id")

<http://localhost/bbtForum.do?method=listBoardTopic&id=1&userId=10&userName=tom>

　　listBoardTopic(@RequestParam("id")int topicId,User user) 中的 topicId 绑定到 id 这个 URL 参数， 那么可以通过对入参使用 @RequestParam 注解来达到目的

　　@RequestParam(required=false)：参数不是必须的，默认为true

　　@RequestParam(value="id",required=false)

　　请求处理方法入参的可选类型

• Java 基本数据类型和 String

默认情况下将按名称匹配的方式绑定到 URL 参数上，可以通过 @RequestParam 注解改变默认的绑定规则

• request/response/session

既可以是 Servlet API 的也可以是 Portlet API 对应的对象，Spring 会将它们绑定到Servlet 和 Portlet 容器的相应对象上

  • org.springframework.web.context.request.WebRequest

　　 内部包含了 request 对象

  • java.util.Locale

　　绑定到 request 对应的 Locale 对象上

• java.io.InputStream/java.io.Reader

           可以借此访问 request 的内容

• java.io.OutputStream / java.io.Writer

可以借此操作 response 的内容

• 任何标注了 @RequestParam 注解的入参

被标注 @RequestParam 注解的入参将绑定到特定的 request 参数上。

• java.util.Map / org.springframework.ui.ModelMap

它绑定 Spring MVC 框架中每个请求所创建的潜在的模型对象，它们可以被 Web 视图对象访问（如 JSP ）

• 命令/ 表单对象（注：一般称绑定使用 HTTP GET 发送的 URL 参数的对象为命令对象，而称绑定使用HTTP POST 发送的 URL 参数的对象为表单对象）

它们的属性将以名称匹配的规则绑定到 URL 参数上，同时完成类型的转换。

而类型转换的规则可以通过 @InitBinder 注解或通过 HandlerAdapter 的配置进行调 整

• org.springframework.validation.Errors / org.springframework.validation.BindingResult

为属性列表中的命令/ 表单对象的校验结果，注意检验结果参数必须紧跟在命令/ 表单对象的后面

• org.springframework.web.bind.support.SessionStatus

可以通过该类型 status 对象显式结束表单的处理，这相当于触发 session 清除其中的通过@SessionAttributes 定义的属性

请求处理方法返回值的可选类型

• void

此时逻辑视图名由请求处理方法对应的 URL 确定，如以下的方法：  
　　 @RequestMapping("/welcome.do")  
　　 public void welcomeHandler() {}  
　　 对应的逻辑视图名为 “ welcome ”

• String

此时逻辑视图名为返回的字符，如以下的方法：  
　　 @RequestMapping(method = RequestMethod.GET)  
　　 public String setupForm(@RequestParam("ownerId") int ownerId, ModelMap model) {

　　 Owner owner = this.clinic.loadOwner(ownerId);  
　　 model.addAttribute(owner);  
　　 return "ownerForm";

}  
　　 对应的逻辑视图名为 “ ownerForm ”

• org.springframework.ui.ModelMap

　　和返回类型为 void 一样，逻辑视图名取决于对应请求的 URL ，如下面的例子：  
　　@RequestMapping("/vets.do")  
　　public ModelMap vetsHandler() {

　　 return new ModelMap(this.clinic.getVets());

　　}

对应的逻辑视图名为 “ vets ” ，返回的 ModelMap 将被作为请求对应的模型对象，可以在 JSP 视图页面中访问到。

　　• ModelAndView  
　　当然还可以是传统的 ModelAndView 。

@ModelAttribute

• 作用域：request

• 例如  
 @RequestMapping("/base/userManageCooper/init.do")

public String handleInit(@ModelAttribute("queryBean") ManagedUser sUser,Model model,){

• 或者

@ModelAttribute("coopMap")// 将coopMap 返回到页 面  
 public Map<Long,CooperatorInfo> coopMapItems(){}

• 说明

@ModelAttribute 声明在属性上，表示该属性的value 来源于model 里"queryBean" ，并被保存到model 里@ModelAttribute声明在方法上，表示该方法的返回值被保存到model 里

@Cacheable 和@CacheFlush

• @Cacheable ：声明一个方法的返回值应该被缓存。例如：@Cacheable(modelId = "testCaching")

• @CacheFlush ：声明一个方法是清空缓存的触发器。例如：@CacheFlush(modelId = "testCaching")

• 说明

要配合缓存处理器使用，参考： [http://hanqunfeng.iteye.com/blog/603719](http://www.iteye.com/blog/603719)

spring3.0没有对缓存提供支持，不过3.1之后就有了，可以参考：[Spring3.1 Cache注解](http://hanqunfeng.iteye.com/blog/1158824)

@Resource

• 例如

@Resource  
 private DataSource dataSource; // inject the bean named 'dataSource'

• 或者

　　　　@Resource(name="dataSource")  
　　　　@Resource(type=DataSource.class)

　　• 说明

　　 @Resource 默认按bean 的name 进行查找，如果没有找到会按type 进行查找，  
　　 此时与@Autowired 类 似

在没有为 @Resource 注解显式指定 name 属性的前提下，如果将其标注在 BeanFactory 类型、ApplicationContext 类型、ResourceLoader 类型、　　ApplicationEventPublisher 类型、MessageSource 类型上，那么 Spring 会自动注入这些实现类的实例，不需要额外的操作。此时 name 属性不需要指定 ( 或者指定为"")，否则注入失败；

@PostConstruct 和@PreDestroy

　　• @PostConstruct

在方法上加上注解@PostConstruct ，这个方法就会在Bean 初始化之后被Spring 容器执 行  
（注：Bean 初始化包括，实例化Bean ，并装配Bean 的属性（依赖注入））。

• @PreDestroy  
在方法上加上注解@PreDestroy ，这个方法就会在Bean 被销毁前被Spring 容器执行。

@Repository

• 与@Controller 、@Service 类似，都是向spring 上下文中注册bean ，不在赘述。

@Component （不推荐使用）

　　• @Component

@Component 是所有受Spring 管理组件的通用形式，Spring 还提供了更加细化的注解形式：  @Repository 、@Service、@Controller ，它们分别对应存储层Bean ，业务层Bean ，和展示层Bean 。

目前版本（2.5 ）中，这些注解与@Component 的语义是一样的，完全通用， 在Spring 以后的版本中可能会给它们追加更多的语义。 所以，我们推荐使用@Repository 、@Service 、@Controller 来替代@Component 。

@Scope

• 例如

@Scope("session")

@Repository()

public class UserSessionBean implementsSerializable {}

       • 说明

　　　　 在使用XML 定义Bean 时，可以通过bean 的scope 属性来定义一个Bean 的作用范围，  
　　　　 同样可以通过@Scope 注解来完成

        @Scope中可以指定如下值：

       singleton:定义bean的范围为每个spring容器一个实例（默认值）

       prototype:定义bean可以被多次实例化（使用一次就创建一次）

       request:定义bean的范围是http请求（springMVC中有效）

       session:定义bean的范围是http会话（springMVC中有效）

       global-session:定义bean的范围是全局http会话（portlet中有效）

@SessionAttributes

　　• 说明

　　 Spring 允许我们有选择地指定 ModelMap 中的哪些属性需要转存到 session 中，  
　　 以便下一个请求属对应的 ModelMap 的属性列表中还能访问到这些属性。  
　　 这一功能是通过类定义处标注 @SessionAttributes 注解来实现的。  
　　 @SessionAttributes 只能声明在类上，而不能声明在方法上。

　　• 例如

　　 @SessionAttributes("currUser") // 将ModelMap 中属性名为currUser 的属性  
　　 @SessionAttributes({"attr1","attr2"})  
　　 @SessionAttributes(types = User.class)  
　　 @SessionAttributes(types = {User.class,Dept.class})  
　　 @SessionAttributes(types = {User.class,Dept.class},value={"attr1","attr2"})

@InitBinder

　　• 说明

　　 如果希望某个属性编辑器仅作用于特定的 Controller ，  
　　 可以在 Controller 中定义一个标注 @InitBinder 注解的方法，  
　　 可以在该方法中向 Controller 了注册若干个属性编辑器

　　• 例如

　　@InitBinder  
　　public void initBinder(WebDataBinder binder) {

　　 SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
　　 dateFormat.setLenient(false);  
　　 binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));

}

@Required

• 例如

@Required

public  setName(String name){}

• 说明

@ required 负责检查一个bean在初始化时其声明的 set方法是否被执行， 当某个被标注了 @Required 的 Setter 方法没有被调用，则 Spring 在解析的时候会抛出异常，以提醒开发者对相应属性进行设置。 @Required 注解只能标注在 Setter 方法之上。因为依赖注入的本质是检查 Setter 方法是否被调用了，而不是真的去检查属性是否赋值了以及赋了什么样的值。如果将该注解标注在非 setXxxx() 类型的方法则被忽略。

@Qualifier

• 例如

@Autowired

@Qualifier("softService")  
 private ISoftPMService softPMService;

• 说明

使用@Autowired 时，如果找到多个同一类型的bean，则会抛异常，此时可以使用 @Qualifier("beanName")，明确指定bean的名称进行注入，此时与 @Resource指定name属性作用相同。

# [Spring开启Annotation <context:annotation-config> 和 <context:component-scan>诠释及区别](https://www.cnblogs.com/leiOOlei/p/3713989.html)

<context:annotation-config> 和 <context:component-scan>的区别

<context:annotation-config> **是用于激活那些已经在spring容器里注册过的bean**（无论是通过xml的方式还是通过package sanning的方式）上面的注解。

<context:component-scan>除了具有<context:annotation-config>的功能之外，<context:component-scan>还可以在指定的package下扫描以及注册javabean 。

# spring中<tx:advice></tx:advice>是什么意思？作用是什么？谁能简单说下

<tx:advice id="tv" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="save\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

意思是这个事物advice的管理者为transactionManager，你从配置文件中应该能够找到一个[ID](https://www.baidu.com/s?wd=ID&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujubm1FbmHbLuH01uHDz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R1PHcvP1n3Pf)为transactionManager的bean，而这个建议中规定了save方法的传输方式为required,也就是说没有sessionfactory的时候会自动建立，有的时候就不会建立了。

当然了这只是规定了一个advice，你还需要指定一个aop:pointcut去引用他，例如

<aop:config>

<aop:pointcut [id](https://www.baidu.com/s?wd=id&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujubm1FbmHbLuH01uHDz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R1PHcvP1n3Pf)="bussinessService" expression="execution(public \* x.y..\*.\*(..))" />

<aop:advisor pointcut-ref="bussinessService" advice-ref="tv" />

</aop:config>

这样这个advice就被联系到了bussinessService这个pointcut上了

# spring tx：advice ...aop：config配置事物

spring来配置事务管理，记住三个参数就可以了

<aop:pointcut>切入点

<aop:advice>需要注入的方法

<aop:advisor> 适配器，把注入的方法注入切入点的位置，是连接pointcut跟advice的工具

下面上实例

1. <aop:config>
2. <aop:pointcut expression="execution(public \* com.test.registration.service.\*.\*(..))" id="logPointcut" />
3. <aop:advisor pointcut-ref="logPointcut" advice-ref="txAdvice" />
4. </aop:config>
5. <tx:advice id="txAdvice" transaction-manager="txManager">
6. <tx:attributes>
7. <tx:method name="exists" read-only="true"/>
8. <tx:method name="add\*" propagation="REQUIRED"/>
9. <tx:method name="modify\*" propagation="REQUIRED"/>
10. <tx:method name="del\*" propagation="REQUIRED"/>
11. </tx:attributes>
12. </tx:advice>

<aop:pointcut expression="execution(public \* com.test.registration.service.\*.\*(..))"

这句话对切入点执行方法的解析public \* com.test.registration.service.\*.\*(..)

第一个\*表示返回值为任意值

第二个\*表示在 com.test.registration.service这个包中的任意类

第三个\*表示当前类的任意方法（..）表示0个或多个参数

对于上边的实例

tx：advice的配置解析事物管理来自txManager 而sessionfactory具体就不在详述了主要是配置数据源跟实体类之类的

tx：attributes里的方法，这里主要对增删改查进行了配置，

有的初学者会问，什么是事物呢，事物简单讲就是对多个方法之类的同时操作，遇到错误同时回滚，在这里是为了避免数据提交，

比如一个方法里写了好多个insert，有一个突然由于一些故障，insert不了了，那么直接插入的话，其结果肯定不是自己想要的了，

数据库里出现了一些脏数据。

对于里边还有的参数解析，详见spring api.

# Spring aop:aspect与aop:advisor的区别

在开发过程中，不少有Spring Aop的使用，

在面向切面编程时，我们会使用< aop:aspect>；

在进行事务管理时，我们会使用< aop:advisor>。

那么，对于< aop:aspect>与< aop:advisor>的区别，具体是怎样的呢？

至于两者的区别，网上有很多资料，但是似乎都不能说清楚。   
首先，我们需要明确两者的概念。

<aop:aspect>：定义切面（切面包括通知和切点）

<aop:advisor>：定义通知器（通知器跟切面一样，也包括通知和切点）

下面，我们列举两者的几个区别。

## 1、实现方式不同

<aop:aspect>定义切面时，只需要定义一般的bean就行，

而定义< aop:advisor>中引用的通知时，通知必须实现Advice接口。

下面我们举例说明。   
首先，我们定义一个接口Sleepable和这个接口的实现Human，代码如下：

1. public interface Sleepable {
2. public void sleep();
3. }
5. public class Human implements Sleepable {
6. @Override
7. public void sleep() {
8. System.out.println("我要睡觉了！");
9. }
10. }

### 下面是< aop:advisor>的实现方式：

1. *//定义通知*
2. public class SleepHelper implements MethodBeforeAdvice,AfterReturningAdvice{
3. @Override
4. public void before(Method arg0, Object[] arg1, Object arg2) throws Throwable {
5. System.out.println("睡觉前要脱衣服！");
6. }
8. @Override
9. public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3) throws Throwable {
10. System.out.println("起床后要穿衣服！");
11. }
12. }
14. *//aop配置*
15. <bean id="sleepHelper" class="com.ghs.aop.SleepHelper"></bean>
17. <aop:config>
18. <aop:pointcut expression="execution(\* \*.sleep(..))" id="sleepPointcut"/>
19. <aop:advisor advice-ref="sleepHelper" pointcut-ref="sleepPointcut"/>
20. </aop:config>
22. <bean id="human" class="com.ghs.aop.Human"/>

### 下面是< aop:aspect>的实现方式：

1. //定义切面
2. public class SleepHelperAspect{
3. public void beforeSleep(){
4. System.out.println("睡觉前要脱衣服！");
5. }
7. public void afterSleep(){
8. System.out.println("起床后要穿衣服！");
9. }
10. }
12. //aop配置
13. <bean id="sleepHelperAspect" class="com.ghs.aop.SleepHelperAspect"></bean>
15. <aop:config>
16. <aop:pointcut expression="execution(\* \*.sleep(..))" id="sleepPointcut"/>
17. <aop:aspect ref="sleepHelperAspect">
18. *<!--前置通知-->*
19. <aop:before method="beforeSleep" pointcut-ref="sleepPointcut"/>
20. *<!--后置通知-->*
21. <aop:after method="afterSleep" pointcut-ref="sleepPointcut"/>
22. </aop:aspect>
23. </aop:config>
25. <bean id="human" class="com.ghs.aop.Human"/>

测试代码如下：

1. public class TestAOP {
2. public static void main(String[] args) {
3. method1();
4. *// method2();*
5. }
7. private static void method1() {
8. ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext1.xml");
9. Sleepable sleeper = (Sleepable) context.getBean("human");
10. sleeper.sleep();
11. }
13. private static void method2() {
14. ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext2.xml");
15. Sleepable sleeper = (Sleepable) context.getBean("human");
16. sleeper.sleep();
17. }
19. *//执行结果*
20. 睡觉前要脱衣服！
21. 我要睡觉了！
22. 起床后要穿衣服！
23. }

## 2、使用场景不同

<aop:advisor>大多用于事务管理。   
例如：

1. *<!-- 会重复读，不会脏读事务 -->*
2. <tx:advice id="txAdvice" transaction-manager="transactionManager">
3. <tx:attributes>
4. <tx:method name="\*" timeout="120" propagation="REQUIRED" rollback-for="Exception" />
5. </tx:attributes>
6. </tx:advice>
8. <aop:config proxy-target-class="true">
9. <aop:pointcut id="txPointCut" expression="..."/>
10. <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut" />
11. </aop:config>

其实，不管是< aop:advisor>还是< aop:aspect>最终的实现逻辑是一样的。

小结：   
可以看出，< aop:advisor>和< aop:aspect>其实都是将通知和切面进行了封装，原理基本上是一样的，只是使用的方式不同而已。

附：   
对于这个问题的理解，其实尚未有一个深入的认识，下面推荐大家几篇博文，希望共同交流。   
<http://www.iteye.com/problems/69785>   
<http://blog.sina.com.cn/s/blog_5198c7370100hw1p.html>   
<http://blog.csdn.net/huitoukest/article/details/46469177>   
<http://www.tz365.cn/ask/shenghuo/2016/0804/739237.html>   
<https://zhidao.baidu.com/question/371238289198208804.html>

# advisor标签

advisor标签是需要定义在aspect标签里面的，其作用与aspect类似，可以简单的把它理解为一个特殊的切面，用于把一个Advice和一个Pointcut组合起来。一个advisor标签对应的就是一个Advisor接口的实现类，默认是DefaultBeanFactoryPointcutAdvisor实现。其使用的基本语法类似如下这样。

<aop:config>

<aop:advisor advice-ref="" pointcut-ref=""/>

</aop:config>

上面的advice-ref属性用于指定一个org.aopalliance.aop.Advice实现，该接口没有任何内容，只是起到标记作用，用于标记某个类是Advice。pointcut-ref用于指定一个通过已经存在的Pointcut定义，当然也可以直接通过pointcut属性指定对应的Pointcut表达式。如果在一个config元素下既定义了aspect，又定义了advisor，那advisor必须定义在aspect之前。接下来看一下如何通过advisor标签应用常用的5种Advice，本文旨在介绍advisor是如何用的，以及如何使用5种Advice，至于每种Advice的功能、区别啥的已经在之前的文章中已经介绍过了，就不再赘述了，有兴趣的读者请参考以前的博文。

# jdk的动态代理和CGLIB的区别

先说一下我对代理的理解：  
    我们租房或者买房的时候，我们只关注房子多大，多少钱，还有一些其他需求，但是这个过程有时候需要你自己  
    去找资源，要商量价格，要自己拟定合同的内容，我们上班哪有那么多时间去管这些事呢，因此中介油然而生，  
    中介帮你完成我们与自己需求不关注的点，这就跟代理类是一个意思。  
jdk代理是基于接口实现类来说的，如果你没有实现接口，则不给你代理：  
    当一个实现类实现一个接口时，实现类实现了接口的方法。在实际开发过程中我们最重要的是实现业务逻辑，  
    但是有一些像日志之类的功能跟业务没有多大关系，所以就需要让别的类来现实日志功能，这时候就需要代理类来  
    帮助我们完成这个任务。  
    代理的大概原理就是：代理类实现接口的方法，在实现的方法里，自己可以想怎么玩怎么玩，玩完之后再调取  
    甲方（也就是谁让我们代理那个实现类）对应的方法，执行完我还可以想干什么干什么。  
CGLIB与jdk代理不同的是 CGLIB是继承类方式来实现的，这种缺点就是不能代理final修饰的类

一、原理区别：

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP   
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP 

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

如何强制使用CGLIB实现AOP？  
 （1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar  
 （2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>  
  
JDK动态代理和CGLIB字节码生成的区别？  
 （1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类  
 （2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法  
   因为是继承，所以该类或方法最好不要声明成final

# [Spring aop的实现原理](https://www.cnblogs.com/lcngu/p/5339555.html)

* **简介**

　　前段时间写的java设计模式--代理模式，最近在看Spring Aop的时候，觉得于代理模式应该有密切的联系，于是决定了解下Spring Aop的实现原理。

　　说起AOP就不得不说下OOP了，OOP中引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。但是，如果我们需要为部分对象引入公共部分的时候，OOP就会引入大量重复的代码。例如：日志功能。

　　AOP技术利用一种称为“横切”的技术，解剖封装的对象内部，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，这样就能减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。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，而各处都基本相似。比如权限认证、日志、事务处理。

* **实现原理**

　　前面在学习代理模式的时候，了解到代理模式分为动态代理和静态代理。现在我们就以代理模式为基础先实现我们自己的AOP框架，再来研究Spring的AOP的实现原理。

　　先以静态代理实现，静态代理关键是在代理对象和目标对象实现共同的接口，并且代理对象持有目标对象的引用。

　　公共接口代码：

[复制代码](javascript:void(0);)

1 public interface IHello {

2 /\*\*

3 \* 业务方法

4 \* @param str

5 \*/

6 void sayHello(String str);

7 }

[复制代码](javascript:void(0);)

　　目标类代码：

[复制代码](javascript:void(0);)

1 public class Hello implements IHello{

2 @Override

3 public void sayHello(String str) {

4 System.out.println("hello "+str);

5 }

6

7 }

[复制代码](javascript:void(0);)

　　代理类代码，我们给它添加日志记录功能，在方法开始前后执行特定的方法，是不是和AOP特别像呢？

[复制代码](javascript:void(0);)

public class ProxyHello implements IHello{

private IHello hello;

public ProxyHello(IHello hello) {

super();

this.hello = hello;

}

@Override

public void sayHello(String str) {

Logger.start();//添加特定的方法

hello.sayHello(str);

Logger.end();

}

}

[复制代码](javascript:void(0);)

　　日志类代码：

[复制代码](javascript:void(0);)

1 public class Logger {

2 public static void start(){

3 System.out.println(new Date()+ " say hello start...");

4 }

5

6 public static void end(){

7 System.out.println(new Date()+ " say hello end");

8 }

9 }

[复制代码](javascript:void(0);)

　　测试代码：

[复制代码](javascript:void(0);)

1 public class Test {

2 public static void main(String[] args) {

3 IHello hello = new ProxyHello(new Hello());//如果我们需要日志功能，则使用代理类

4 //IHello hello = new Hello();//如果我们不需要日志功能则使用目标类

5 hello.sayHello("明天");

6 }

7 }

[复制代码](javascript:void(0);)

　　这样我们就实现了一个最简单的AOP，但是这样会存在一个问题：如果我们像Hello这样的类很多，那么，我们是不是要去写很多个HelloProxy这样的类呢。其实也是一种很麻烦的事。在jdk1.3以后，jdk跟我们提供了一个API java.lang.reflect.InvocationHandler的类， 这个类可以让我们在JVM调用某个类的方法时动态的为些方法做些什么事。下面我们就来实现动态代理的实现。

　　动态代理实现主要是实现InvocationHandler，并且将目标对象注入到代理对象中，利用反射机制来执行目标对象的方法。

　　接口实现与静态代理相同，代理类代码：

[复制代码](javascript:void(0);)

1 public class DynaProxyHello implements InvocationHandler{

2

3 private Object target;//目标对象

4 /\*\*

5 \* 通过反射来实例化目标对象

6 \* @param object

7 \* @return

8 \*/

9 public Object bind(Object object){

10 this.target = object;

11 return Proxy.newProxyInstance(this.target.getClass().getClassLoader(), this.target.getClass().getInterfaces(), this);

12 }

13

14 @Override

15 public Object invoke(Object proxy, Method method, Object[] args)

16 throws Throwable {

17 Object result = null;

18 Logger.start();//添加额外的方法

19 //通过反射机制来运行目标对象的方法

20 result = method.invoke(this.target, args);

21 Logger.end();

22 return result;

23 }

24

25 }

[复制代码](javascript:void(0);)

　　测试类代码：

[复制代码](javascript:void(0);)

1 public class DynaTest {

2 public static void main(String[] args) {

3 IHello hello = (IHello) new DynaProxyHello().bind(new Hello());//如果我们需要日志功能，则使用代理类

4 //IHello hello = new Hello();//如果我们不需要日志功能则使用目标类

5 hello.sayHello("明天");

6 }

7 }

[复制代码](javascript:void(0);)

　　看完上面的代码可能和Spring AOP相比有一个问题，日志类只能在方法前后打印，但是AOP应该是可以在满足条件就可以执行，所有是否可以将DynaPoxyHello对象和日志操作对象(Logger)解耦呢？

　　看下面代码实现，将将DynaPoxyHello对象和日志操作对象(Logger)解耦：

　　我们要在被代理对象的方法前面或者后面去加上日志操作代码(或者是其它操作的代码)，那么，我们可以抽象出一个接口，这个接口里就只有两个方法：一个是在被代理对象要执行方法之前执行的方法,我们取名为start，第二个方法就是在被代理对象执行方法之后执行的方法,我们取名为end。

　　Logger的接口：

1 public interface ILogger {

2 void start(Method method);

3 void end(Method method);

4 }

　　Logger的接口实现：

[复制代码](javascript:void(0);)

1 public class DLogger implements ILogger{

2 @Override

3 public void start(Method method) {

4 System.out.println(new Date()+ method.getName() + " say hello start...");

5 }

6 @Override

7 public void end(Method method) {

8 System.out.println(new Date()+ method.getName() + " say hello end");

9 }

10 }

[复制代码](javascript:void(0);)

　　动态代理类：

[复制代码](javascript:void(0);)

1 public class DynaProxyHello1 implements InvocationHandler{

2 //调用对象

3 private Object proxy;

4 //目标对象

5 private Object target;

6

7 public Object bind(Object target,Object proxy){

8 this.target=target;

9 this.proxy=proxy;

10 return Proxy.newProxyInstance(this.target.getClass().getClassLoader(), this.target.getClass().getInterfaces(), this);

11 }

12

13

14 @Override

15 public Object invoke(Object proxy, Method method, Object[] args)

16 throws Throwable {

17 Object result = null;

18 //反射得到操作者的实例

19 Class clazz = this.proxy.getClass();

20 //反射得到操作者的Start方法

21 Method start = clazz.getDeclaredMethod("start", new Class[]{Method.class});

22 //反射执行start方法

23 start.invoke(this.proxy, new Object[]{this.proxy.getClass()});

24 //执行要处理对象的原本方法

25 method.invoke(this.target, args);

26 //反射得到操作者的end方法

27 Method end = clazz.getDeclaredMethod("end", new Class[]{Method.class});

28 //反射执行end方法

29 end.invoke(this.proxy, new Object[]{method});

30 return result;

31 }

32

33 }

[复制代码](javascript:void(0);)

　　测试代码：

[复制代码](javascript:void(0);)

1 public class DynaTest1 {

2 public static void main(String[] args) {

3 IHello hello = (IHello) new DynaProxyHello1().bind(new Hello(),new DLogger());//如果我们需要日志功能，则使用代理类

4 //IHello hello = new Hello();//如果我们不需要日志功能则使用目标类

5 hello.sayHello("明天");

6 }

7 }

[复制代码](javascript:void(0);)

　　通过上面例子，可以发现通过动态代理和发射技术，已经基本实现了AOP的功能，如果我们只需要在方法执行前打印日志，则可以不实现end()方法，这样就可以控制打印的时机了。如果我们想让指定的方法打印日志，我们只需要在invoke（）方法中加一个对method名字的判断，method的名字可以写在xml文件中，这样我们就可以实现以配置文件进行解耦了，这样我们就实现了一个简单的spring aop框架。