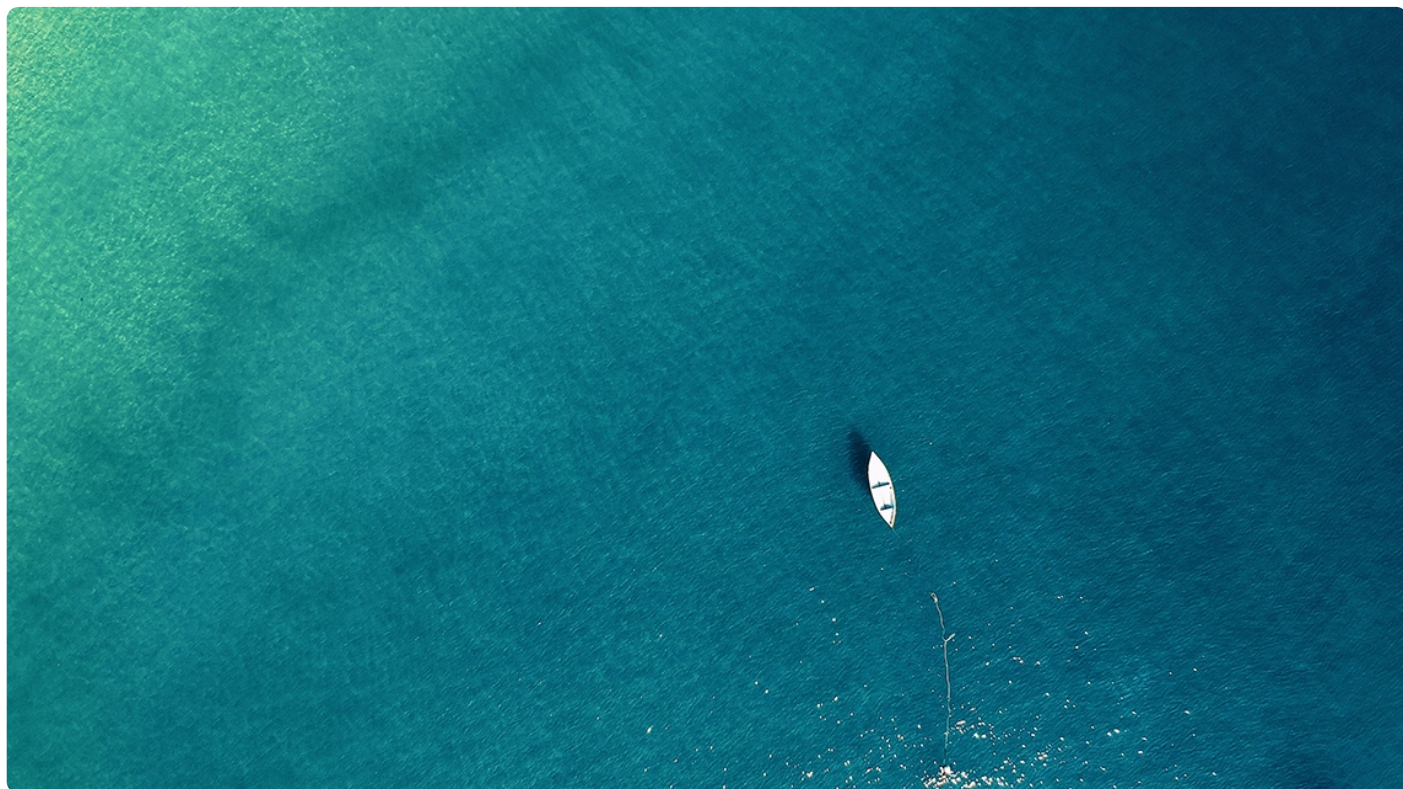


20 | AutoProxyCreator：如何自动添加动态代理？

2023-04-26 郭屹 来自北京


《手把手带你写一个MiniSpring》



你好，我是郭屹，今天我们继续手写 MiniSpring，这也是 AOP 正文部分的最后一节。今天我们将完成一个有模有样的 AOP 解决方案。

问题的引出

前面，我们已经实现了通过动态代理技术在运行时进行逻辑增强，并引入了 Pointcut，实现了代理方法的通配形式。到现在，AOP 的功能貌似已经基本实现了，但目前还有一个较大的问题，具体是什么问题呢？我们查看 applicationContext.xml 里的这段配置文件来一探究竟。

 复制代码

```
1 <bean id="realaction" class="com.test.service.Action1" />
2 <bean id="action" class="com.minis.aop.ProxyFactoryBean">
3     <property type="String" name="interceptorName" value="advisor" />
4     <property type="java.lang.Object" name="target" ref="realaction"/>
5 </bean>
```

看这个配置文件可以发现，在 ProxyFactoryBean 的配置中，有个 Object 类型的属性：target。在这里我们的赋值 ref 是 realactionbean，对应 Action1 这个类。也就是说，给 Action1 这个 Bean 动态地插入逻辑，达成 AOP 的目标。

在这里，一次 AOP 的配置对应一个目标对象，如果整个系统就只需要为一个对象进行增强操作，这自然没有问题，配置一下倒也不会很麻烦，但在一个稍微有规模的系统中，我们有成百上千的目标对象，在这种情况下一个个地去配置则无异于一场灾难。


一个实用的 AOP 解决方案，应该可以**用一个简单的匹配规则代理多个目标对象**。这是我们这节课需要解决的问题。

匹配多个目标对象的思路

在上节课，我们其实处理过类似的问题，就是当时我们的目标方法只能是一个固定的方法名 doAction()，我们就提出了 Pointcut 这个概念，用一个模式来通配方法名，如 do*、do*Action 之类的字符串模式。

Pointcut 这个概念解决了一个目标对象内部多个方法的匹配问题。这个办法也能给我们灵感，我们就借鉴这个思路，用类似的手段来解决匹配多个目标对象的问题。

因此，我们想象中当解决方案实现之后，应该是这么配置的。

 复制代码

```
1 <bean id="genaralProxy" class="GeneralProxy" >
2     <property type="String" name="pattern" value="action*" />
3     <property type="String" name="interceptorName" value="advisor" />
4 </bean>
```

上面的配置里有一个通用的 ProxyBean，它用一个模式串 pattern 来匹配目标对象，作为例子这里就是 action*，表示所有名字以 action 开头的对象都是目标对象。

这个想法好像成立，但是我们知道，IoC 容器内部所有的 Bean 是相互独立且平等的，这个 GeneralProxy 也就是一个普通的 Bean。那么作为一个普通的 Bean，它怎么能影响到别的

Bean 呢？它如何能做到给别的 Bean 动态创建代理呢？这个办法有这样一个关键的难点。

我们反过来思考，如果能找个办法让这个 General Proxy 影响到别的 Bean，再根据规则决定给这些 Bean 加上动态代理（这一点我们之前就实现过了），是不是就可以了？

那么在哪个时序点能做这个事情呢？我们再回顾一下 Bean 的创建过程：第一步，IoC 容器扫描配置文件，加载 Bean 的定义。第二步，通过 `getBean()` 这个方法创建 Bean 实例，这一步又分成几个子步骤：

1. 创建 Bean 的毛坯实例；
2. 填充 Properties；
3. 执行 `postProcessBeforeInitialization`；
4. 调用 `init-method` 方法；
5. 执行 `postProcessAfterInitialization`。

后三个子步骤，实际上都是在每一个 Bean 实例创建好之后可以进行的后期处理。那么我们就可以利用这个时序，把自动生成代理这件事情交给后期处理来完成。在我们的 IoC 容器里，有一个现成的机制，叫 **BeanPostProcessor**，它能在每一个 Bean 创建的时候进行后期修饰，也就是上面的 3 和 5 两个子步骤其实都是调用的 `BeanPostProcessor` 里面的方法。所以现在就比较清晰了，我们考虑用 `BeanPostProcessor` 实现自动生成目标对象代理。

利用 `BeanPostProcessor` 自动创建代理

创建动态代理的核心是**把传进来的 Bean 包装成一个 `ProxyFactoryBean`**，改头换面变成一个动态的代理，里面包含了真正的业务对象，这一点我们已经在前面的工作中做好了。现在是要自动创建这个动态代理，它的核心就是通过 `BeanPostProcessor` 来为每一个 Bean 自动完成创建动态代理的工作。

我们用一个 `BeanNameAutoProxyCreator` 类实现这个功能，顾名思义，这个类就是根据 Bean 的名字匹配来自动创建动态代理的，你可以看一下相关代码。

```

1 package com.minis.aop.framework.autoproxy;
2 public class BeanNameAutoProxyCreator implements BeanPostProcessor{
3     String pattern; //代理对象名称模式, 如action*
4     private BeanFactory beanFactory;
5     private AopProxyFactory aopProxyFactory;
6     private String interceptorName;
7     private PointcutAdvisor advisor;
8     public BeanNameAutoProxyCreator() {
9         this.aopProxyFactory = new DefaultAopProxyFactory();
10    }
11    //核心方法。在bean实例化之后, init-method调用之前执行这个步骤。
12    @Override
13    public Object postProcessBeforeInitialization(Object bean, String beanName) {
14        if (isMatch(beanName, this.pattern)) {
15            ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean(); //创建以恶
16            proxyFactoryBean.setTarget(bean);
17            proxyFactoryBean.setBeanFactory(beanFactory);
18            proxyFactoryBean.setAopProxyFactory(aopProxyFactory);
19            proxyFactoryBean.setInterceptorName(interceptorName);
20            return proxyFactoryBean;
21        }
22        else {
23            return bean;
24        }
25    }
26    protected boolean isMatch(String beanName, String mappedName) {
27        return PatternMatchUtils.simpleMatch(mappedName, beanName);
28    }
29 }

```

通过代码可以知道, 在 `postProcessBeforeInitialization` 方法中, 判断了 Bean 的名称是否符合给定的规则, 也就是 `isMatch(beanName, this.pattern)` 这个方法。往下追究一下, 发现这个 `isMatch()` 就是直接调用的 `PatternMatchUtils.simpleMatch()`, 跟上一节课的通配方法名一样。所以如果 Bean 的名称匹配上了, 我们就用和以前创建动态代理一样的办法来自动生成代理。


```

1 ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
2 proxyFactoryBean.setTarget(bean);
3 proxyFactoryBean.setBeanFactory(beanFactory);
4 proxyFactoryBean.setAopProxyFactory(aopProxyFactory);
5 proxyFactoryBean.setInterceptorName(interceptorName);

```

这里我们还是用到了 ProxyFactoryBean，跟以前一样，只不过这里是经过了 BeanPostProcessor。因此，按照 IoC 容器的规则，这一切不再是手工的了，而是对每一个符合规则 Bean 都会这样做一次动态代理，就可以完成我们的工作了。

现在我们只要把这个 BeanPostProcessor 配置到 XML 文件里就可以了。

 复制代码

```
1 <bean id="autoProxyCreator" class="com.minis.aop.framework.autoproxy.BeanNameAuto
2     <property type="String" name="pattern" value="action*" />
3     <property type="String" name="interceptorName" value="advisor" />
4 </bean>
```

IoC 容器扫描配置文件的时候，会把所有的 BeanPostProcessor 对象加载到 Factory 中生效，每一个 Bean 都会过一遍手。

getBean 方法的修改

工具准备好了，这个 BeanPostProcessor 会自动创建动态代理。为了使用这个 Processor，对应的 AbstractBeanFactory 类里的 getBean() 方法需要同步修改。你可以看一下修改后 getBean 的实现。

 复制代码

```
1 public Object getBean(String beanName) throws BeansException{
2     Object singleton = this.getSingleton(beanName);
3     if (singleton == null) {
4         singleton = this.earlySingletonObjects.get(beanName);
5         if (singleton == null) {
6             BeanDefinition bd = beanDefinitionMap.get(beanName);
7             if (bd != null) {
8                 singleton=createBean(bd);
9                 this.registerBean(beanName, singleton);
10                if (singleton instanceof BeanFactoryAware) {
11                    ((BeanFactoryAware) singleton).setBeanFactory(this);
12                }
13                //用beanpostprocessor进行后期处理
14                //step 1 : postProcessBeforeInitialization调用processor相关方法
```




```

15         singleton = applyBeanPostProcessorsBeforeInitialization(singleton,
16         //step 2 : init-method
17         if (bd.getInitMethodName() != null && !bd.getInitMethodName().equa
18             invokeInitMethod(bd, singleton);
19         }
20         //step 3 : postProcessAfterInitialization
21         applyBeanPostProcessorsAfterInitialization(singleton, beanName);
22         this.removeSingleton(beanName);
23         this.registerBean(beanName, singleton);
24     }
25     else {
26         return null;
27     }
28 }
29 }
30 else {
31 }
32 //process Factory Bean
33 if (singleton instanceof FactoryBean) {
34     return this.getObjectForBeanInstance(singleton, beanName);
35 }
36 else {
37 }
38 return singleton;
39 }

```

上述代码中主要修改这一行：

 复制代码

```
1 singleton = applyBeanPostProcessorsBeforeInitialization(singleton, beanName);
```


代码里会调用 Processor 的 `postProcessBeforeInitialization` 方法，并返回 `singleton`。这一段代码的功能是如果这个 Bean 的名称符合某种规则，就会自动创建 Factory Bean，这个 Factory Bean 里面会包含一个动态代理对象用来返回自定义的实例。

于是，`getBean` 的时候，除了创建 Bean 实例，还会用 `BeanPostProcessor` 进行后期处理，对满足规则的 Bean 进行包装，改头换面成为一个 Factory Bean。

测试

到这里，我们就完成自动创建动态代理的工作了，简单测试一下。


修改 applicationContext.xml 配置文件，增加一些配置。

 复制代码

```
1 <bean id="autoProxyCreator" class="com.minis.aop.framework.autoproxy.BeanNameAuto
2     <property type="String" name="pattern" value="action*" />
3     <property type="String" name="interceptorName" value="advisor" />
4 </bean>
5
6 <bean id="action" class="com.test.service.Action1" />
7 <bean id="action2" class="com.test.service.Action2" />
8
9 <bean id="beforeAdvice" class="com.test.service.MyBeforeAdvice" />
10 <bean id="advisor" class="com.minis.aop.NameMatchMethodPointcutAdvisor">
11     <property type="com.minis.aop.Advice" name="advice" ref="beforeAdvice"/>
12     <property type="String" name="mappedName" value="do*" />
13 </bean>
```

这里我们配置了两个 Bean，BeanPostProcessor 和 Advisor。

相应地，controller 层的 HelloWorldBean 增加一段代码。

 复制代码

```
1 @Autowired
2 IAction action;
3
4 @RequestMapping("/testaop")
5 public void doTestAop(HttpServletRequest request, HttpServletResponse response) {
6     action.doAction();
7 }
8 @RequestMapping("/testaop2")
9 public void doTestAop2(HttpServletRequest request, HttpServletResponse response)
10     action.doSomething();
11 }
12
13 @Autowired
14 IAction action2;
15
16 @RequestMapping("/testaop3")
17 public void doTestAop3(HttpServletRequest request, HttpServletResponse response)
18     action2.doAction();
```

```
19 }  
20 @RequestMapping("/testaop4")  
21 public void doTestAop4(HttpServletRequest request, HttpServletResponse response)  
22     action2.doSomething();  
23 }
```

这里，我们用到了这两个 Bean，action 和 action2，每个 Bean 里面都有 doAction() 和 doSomething() 两个方法。

通过配置文件可以看到，在 Processor 的 Pattern 配置里，通配 action* 可以匹配所有以 action 开头的 Bean。在 Advisor 的 MappedName 配置里，通配 do*，就可以匹配所有以 do 开头的方法。

运行一下，就可以看到效果了。这两个 Bean 里的两个方法都加上了增强，说明系统在调用这些 Bean 的方法时自动插入了逻辑。

小结

这节课，我们对匹配 Bean 的办法进行了扩展，使系统可以按照某个规则来匹配某些 Bean，这样就不用一个 Bean 一个 Bean 地配置动态代理了。

实现的思路是利用 Bean 的时序，使用一个 BeanPostProcessor 进行后期处理。这个 Processor 接收一个模式串，而这个模式也是可以由用户配置在外部文件里的，然后提供 isMatch() 方法，支持根据名称进行模式匹配。具体的字符串匹配工作，和上节课一样，也是采用从前到后的扫描技术，分节段进行校验。匹配上之后，还是利用以前的 ProxyFactoryBean 创建动态代理。这里要理解一点，就是系统会自动把应用程序员配置的业务 Bean 改头换面，让它变成一个 Factory Bean，里面包含的是业务 Bean 的动态代理。

这个方案能用是因为之前 IoC 容器里提供的这个 BeanPostProcessor 机制，所以这里我们再次看到了 IoC 容器的强大之处。

到这里，我们的 AOP 方案就完成了。这是基于 JDK 的方案，对于理解 AOP 原理很有帮助。

完整源代码参见 <https://github.com/YaleGuo/minis>。

课后题

学完这节课，我也给你留一道思考题。AOP 经常用来处理数据库事务，如何用我们现在的 AOP 架构实现简单的事务处理呢？欢迎你在留言区与我交流讨论，也欢迎你把这节课分享给需要的朋友。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (6)



马儿 置顶

2023-04-26 来自四川

这节课的代码能够做到不需要在beans.xml中额外专门配置来生成代理对象，已经接近spring的雏形了。但是按照之前的代码是跑不起来的，需要对之前的BeanPostProcessor的逻辑修改一下，应该是老师之前讲漏了的部分。主要工作在修改AbstractAutowireCapableBeanFactory类将属性中的beanPostProcessors改为面向接口的列表，其次是修改ClassPathXmlApplicationContext#registerBeanPostProcessors让其可以在配置文件中读到注册的BeanPostProcessors并注册到容器中。最后将我们之前用到的AutowiredAnnotationBeanPostProcessor注册到容器管理中就能够自动发现了。

代码修改可以参考：<https://github.com/horseLk/mini-spring/commit/7186afebeaf30d622d79b4111970945abca97701>

作者回复：github上每一节的代码都是可运行的。文稿是写重点，会有漏掉细节。感谢你的补充。



peter

2023-04-27 来自北京

问题放在第20课，但问题是关于第19课的：

Q1：Join Point和Pointcut的区别是什么？两个看起来是一回事啊。

Q2：流程方面，Interceptor先拦截，拦截以后再进行增强操作。换一种说法，先是Interceptor工作，然后是Join Point、Pointcut、advice这些登场，对吗？

Q3：19课的总结部分，是这样说的：“Advisor：通知者，它实现了 Advice”。19课的留言解答

有这样一句话“advisor则是一个管理类，它包了一个advice，还能寻找到符合条件的方法名进行增强”。留言的解释很不错，但总结部分，“实现了 Advice”，个人感觉这个措辞不是很合理啊，怎么是“实现”？这个词容易让人理解为接口与实现类的关系。

作者回复: Peter总是很用心细致。

Join point跟Pointcut不是一回事,它是指在类中的位置,如是方法上? 是构造器上? 是属性上? pointcut是条件, 如哪些符合条件的方法上加上增强。

流程你得理一下源代码, 这些概念之间并没有流程。

口头讲课确实有一些随意, 不那么精确, “实现”一词呢, 因为advisor里面包了一个advice, 就那么说了, 不是代码意义上的“实现”。形象地说, advice是饭(真正的业务增强逻辑), advisor是碗筷(装饭给人吃的工具), 人不能直接用嘴巴啃饭, 要用一个工具把饭吃到嘴里。



___Alucard

2023-05-29 来自浙江

完结撒花, 谢谢指导

作者回复: 恭喜, 多谢。



___Alucard

2023-05-29 来自浙江

动态代理失效的根本原因是@Autowired注解解析的时候, 取到的spring bean还是没代理的对象; 我的临时解决方案是 在BeanNameAutoProxyCreator的postProcessBeforeInitialization手动把创建后的动态代理对象注入进spring ioc中, beanFactory.registerBean(beanName,proxyFactoryBean); 但是这样会导致BeanFactory又对外暴露了注册bean的接口, void registerBean(String beanName, Object obj);明显不合适, 这个有没有更好的办法

作者回复: 你讲的问题讲到点上了。我只能说Spring目前的方案就是挺合适的方案, 我自己想不出别的合适的办法。



曾泽浩

2023-05-02 来自广东

老师你好，怎么github上面这块framework这个包下面有部分代码是重复的呢？建议可以来个代码结构的总结

作者回复：我检查一下，多谢提醒。



Geek_320730

2023-04-26 来自北京

可以定义一个注解@Transaction并实现一个MethodMatcher，根据有没有这个注解来判断方法是否匹配，匹配的话，在方法执行前，手动开启事务，方法结束后，手动提交事务，有异常的话回滚事务。那事务方法调用事务方法的时候不知道会不会报错。。。

另外遇到Bean可以一直嵌套代理的问题，比如上一章手动配置的action,本身就是一个ProxyFactoryBean了，但是他的名字依然符合本章的action*的匹配规则，这样就又加了一层代理，注入的时候就会失败。需要在获取类的时候判断一下类型递归返回，或者在bean匹配规则的时候做一下类型判断，如果本身是个ProxyFactoryBean了，就不做操作返回。

作者回复：你这个bean嵌套代理的补充很好。

