

19 | Pointcut：如何批量匹配代理方法？

2023-04-24 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。

到目前为止，我们已经初步实现了简单的 AOP，做到了封装 JDK 的动态代理，并且定义了 Advice，实现了调用前、调用时、调用后三个不同位置对代理对象进行增强的效果，而这些切面的定义也是配置在外部文件中的。我们现在在这个基础之上继续前进，引入 Pointcut 这个概念，批量匹配需要代理的方法。

引入 Pointcut

我们再回头看一下代码，前面所有的代理方法，都是同一个名字——doAction。我们用以下代码将该方法名写死了，也就是说我们只认定这一个方法名为代理方法，而且名字是不能改的。

```
1 if (method.getName().equals("doAction")) {
```

复制代码

如果我们需要增加代理方法，或者就算不增加，只是觉得这个方法名不好想换一个，怎么办呢？当前这种方法自然不能满足我们的需求了。而这种对多个方法的代理需求又特别重要，因为业务上有可能会想对某一类方法进行增强，统一加上监控日志什么的，这种情况下，如果要逐个指定方法名就太麻烦了。

进一步考虑，即便我们这里可以支持多个方法名，但是匹配条件仍然是 equals，也就是说，规则仅仅是按照方法名精确匹配的，这样做太不灵活了。

因此这节课我们考虑用方法名匹配规则进行通配，而这个配置则允许应用开发程序员在 XML 文件中自定义。这就是我们常说的**切点 (Pointcut)**，按照规则匹配需要代理的方法。

我们先确定一下，这节课代码改造完毕后，配置文件是什么样子的，我把变动最大的地方放在下面，供你参考。

[复制代码](#)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans>
3     <bean id="realaction" class="com.test.service.Action1" />
4     <bean id="beforeAdvice" class="com.test.service.MyBeforeAdvice" />
5     <bean id="advisor" class="com.minis.aop.NameMatchMethodPointcutAdvisor">
6         <property type="com.minis.aop.Advice" name="advice" ref="beforeAdvice"/>
7         <property type="String" name="mappedName" value="do*"/>
8     </bean>
9     <bean id="action" class="com.minis.aop.ProxyFactoryBean">
10        <property type="String" name="interceptorName" value="advisor" />
11        <property type="java.lang.Object" name="target" ref="realaction"/>
12    </bean>
13 </beans>
```

由上述改动可以看出，我们新定义了一个 NameMatchMethodPointcutAdvisor 类作为 Advisor，其中 property 属性中的 value 值为 do*，这就是我们说的方法规则，也就是匹配所有以 do 开头的方法名称。这里你也可以根据实际的业务情况按照一定的规则配置自定义的代理方法，而不仅仅局限于简单的方法名精确相等匹配。

有了这个 Pointcut，我们就能用一条规则来支持多个代理方法了，这非常有用。如果能实现这个配置，就达到了我们想要的效果。


为了实现这个目标，最后构建出一个合适的 NameMatchMethodPointcutAdvisor，我们定义了 MethodMatcher、Pointcut 与 PointcutAdvisor 三个接口。

MethodMatcher 这个接口代表的是方法的匹配算法，内部的实现就是看某个名是不是符合某个模式。

 复制代码


```
1 package com.minis.aop;
2 public interface MethodMatcher {
3     boolean matches(Method method, Class<?> targetClass);
4 }
```

Pointcut 接口定义了切点，也就是返回一条匹配规则。

 复制代码

```
1 package com.minis.aop;
2 public interface Pointcut {
3     MethodMatcher getMethodMatcher();
4 }
```

PointcutAdvisor 接口扩展了 Advisor，内部可以返回 Pointcut，也就是说这个 Advisor 有一个特性：能支持切点 Pointcut 了。这也是一个常规的 Advisor，所以可以放到我们现有的 AOP 框架中，让它负责来增强。

 复制代码

```
1 package com.minis.aop;
2 public interface PointcutAdvisor extends Advisor{
3     Pointcut getPointcut();
4 }
```

接口定义完毕之后，接下来就要有这些接口对应的实现。实际我们在原理上可以实现一系列不同的规则，但是现在我们只能简单地使用名称进行模式匹配，不过能通过这个搞清楚原理就可以了。


如何匹配？

我们先来看核心问题：**如何匹配到方法？** 我们默认的实现是 `NameMatchMethodPointcut` 和 `NameMatchMethodPointcutAdvisor`。

 复制代码

```
1 package com.minis.aop;
2 public class NameMatchMethodPointcut implements MethodMatcher, Pointcut{
3     private String mappedName = "";
4     public void setMappedName(String mappedName) {
5         this.mappedName = mappedName;
6     }
7     @Override
8     public boolean matches(Method method, Class<?> targetClass) {
9         if (mappedName.equals(method.getName()) || isMatch(method.getName(), mappedName)) {
10             return true;
11         }
12         return false;
13     }
14     //核心方法，判断方法名是否匹配给定的模式
15     protected boolean isMatch(String methodName, String mappedName) {
16         return PatternMatchUtils.simpleMatch(mappedName, methodName);
17     }
18     @Override
19     public MethodMatcher getMethodMatcher() {
20         return null;
21     }
22 }
```

我们看到了，这个类的核心方法就是 **isMatch()**，它用到了一个工具类叫 **PatternMatchUtils**。我们看一下这个工具类是怎么进行字符串匹配的。

 复制代码

```
1 /**
2  * 用给定的模式匹配字符串。
3  * 模式格式："xxx*"，"*xxx"，"*xxx*" 以及 "xxx*yyy"，*代表若干个字符。
```

```

4  */
5  public static boolean simpleMatch( String pattern, String str) {
6      //先判断串或者模式是否为空
7      if (pattern == null || str == null) {
8          return false;
9      }
10     //再判断模式中是否包含*
11     int firstIndex = pattern.indexOf('*');
12     if (firstIndex == -1) {
13         return pattern.equals(str);
14     }
15     //是否首字符就是*,意味着这个是*xxx格式
16     if (firstIndex == 0) {
17         if (pattern.length() == 1) { //模式就是*,通配全部串
18             return true;
19         }
20         //尝试查找下一个*
21         int nextIndex = pattern.indexOf('*', 1);
22         if (nextIndex == -1) { //没有下一个*, 说明后续不需要再模式匹配了, 直接endsWith判断
23             return str.endsWith(pattern.substring(1));
24         }
25         //截取两个*之间的部分
26         String part = pattern.substring(1, nextIndex);
27         if (part.isEmpty()) { //这部分为空, 形如**, 则移到后面的模式进行匹配
28             return simpleMatch(pattern.substring(nextIndex), str);
29         }
30         //两个*之间的部分不为空, 则在串中查找这部分子串
31         int partIndex = str.indexOf(part);
32         while (partIndex != -1) {
33             //模式串移位到第二个*之后, 目标字符串移位到子串之后, 递归再进行匹配
34             if (simpleMatch(pattern.substring(nextIndex), str.substring(partIndex + part
35                 return true;
36         }
37         partIndex = str.indexOf(part, partIndex + 1);
38     }
39     return false;
40 }
41
42 //对不是*开头的模式, 前面部分要精确匹配, 然后后面的子串重新递归匹配
43 return (str.length() >= firstIndex &&
44     pattern.substring(0, firstIndex).equals(str.substring(0, firstIndex)) &&
45     simpleMatch(pattern.substring(firstIndex), str.substring(firstIndex)));
46 }

```

看代码, 整个匹配过程是一种扫描算法, 从前往后扫描, 按照 * 分节段一节一节匹配, 因为长度不定, 所以要用递归, 详细说明代码上有注释。模式格式可以是:"xxx*", "*xxx",

"*xxx*"以及"xxx*yyy"等。

有了上面的实现，我们就有了具体的匹配工具了。下面我们就来使用 PatternMatchUtils 这个工具类来进行字符串的匹配。


NameMatchMethodPointcutAdvisor 的实现也比较简单，就是在内部增加了 NameMatchMethodPointcut 属性和 MappedName 属性。

 复制代码

```
1 package com.minis.aop;
2 public class NameMatchMethodPointcutAdvisor implements PointcutAdvisor{
3     private Advice advice = null;
4     private MethodInterceptor methodInterceptor;
5     private String mappedName;
6     private final NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
7     public NameMatchMethodPointcutAdvisor() {
8     }
9     public NameMatchMethodPointcutAdvisor(Advice advice) {
10         this.advice = advice;
11     }
12     public void setMethodInterceptor(MethodInterceptor methodInterceptor) {
13         this.methodInterceptor = methodInterceptor;
14     }
15     public MethodInterceptor getMethodInterceptor() {
16         return this.methodInterceptor;
17     }
18     public void setAdvice(Advice advice) {
19         this.advice = advice;
20         MethodInterceptor mi = null;
21         if (advice instanceof BeforeAdvice) {
22             mi = new MethodBeforeAdviceInterceptor((MethodBeforeAdvice)advice);
23         }
24         else if (advice instanceof AfterAdvice){
25             mi = new AfterReturningAdviceInterceptor((AfterReturningAdvice)advice);
26         }
27         else if (advice instanceof MethodInterceptor) {
28             mi = (MethodInterceptor)advice;
29         }
30         setMethodInterceptor(mi);
31     }
32     @Override
33     public Advice getAdvice() {
34         return this.advice;
35     }
```

```
36  @Override
37  public Pointcut getPointcut() {
38      return pointcut;
39  }
40  public void setMappedName(String mappedName) {
41      this.mappedName = mappedName;
42      this.pointcut.setMappedName(this.mappedName);
43  }
44 }
```

上述实现代码对新增的 Pointcut 和 MappedName 属性进行了处理，这正好与我们定义的 XML 配置文件保持一致。而匹配的工作，则交给 NameMatchMethodPointcut 中的 matches 方法完成。如配置文件中的 mappedName 设置成了 "do*"，意味着所有 do 开头的方法都会匹配到。


 复制代码

```
1 <bean id="advisor" class="com.minis.aop.NameMatchMethodPointcutAdvisor">
2     <property type="com.minis.aop.Advice" name="advice" ref="beforeAdvice"/>
3     <property type="String" name="mappedName" value="do*" />
4 </bean>
```

另外，我们还要注意 setAdvice() 这个方法，它现在通过 advice 来设置相应的 Interceptor，这一段逻辑以前是放在 ProxyFactoryBean 的 initializeAdvisor() 方法中的，现在移到了这里。现在这个新的 Advisor 就可以支持按照规则匹配方法来进行逻辑增强了。

相关类的改造

在上述工作完成后，相关的一些类也需要改造。JdkDynamicAopProxy 类中的实现，现在我们需要将方法名写死了。你可以看一下改造之后的代码。

 复制代码

```
1 package com.minis.aop;
2 public class JdkDynamicAopProxy implements AopProxy, InvocationHandler {
3     Object target;
4     PointcutAdvisor advisor;
5     public JdkDynamicAopProxy(Object target, PointcutAdvisor advisor) {
6         this.target = target;
7         this.advisor = advisor;
```



```

8      }
9      @Override
10     public Object getProxy() {
11         Object obj = Proxy.newProxyInstance(JdkDynamicAopProxy.class.getClassLoad
12         return obj;
13     }
14     @Override
15     public Object invoke(Object proxy, Method method, Object[] args) throws Thro
16         Class<?> targetClass = (target != null ? target.getClass() : null);
17         if (this.advisor.getPointcut().getMethodMatcher().matches(method, targetC
18             MethodInterceptor interceptor = this.advisor.getMethodInterceptor();
19             MethodInvocation invocation =
20                 new ReflectiveMethodInvocation(proxy, target, method, args, t
21             return interceptor.invoke(invocation);
22     }
23     return null;
24 }
25 }


```

看核心方法 **invoke()**，以前的代码是 `method.getName().equals("doAction")`，即判断名字必须等于"doAction"，现在的判断条件则更具备扩展性了，是用 Pointcut 的 matcher 进行匹配校验。代码是

`this.advisor.getPointcut().getMethodMatcher().matches(method, targetClass))` 这一句。

原本定义的 Advisor 改为了更加具有颗粒度的 PointcutAdvisor，自然连带着其他引用类也要一并修改。

DefaultAopProxyFactory 的 createAopProxy() 方法中，Advisor 参数现在就可以使用 PointcutAdvisor 类型了。


 复制代码

```

1 package com.minis.aop;
2 public class DefaultAopProxyFactory implements AopProxyFactory{
3     @Override
4     public AopProxy createAopProxy(Object target, PointcutAdvisor advisor) {
5         return new JdkDynamicAopProxy(target, advisor);
6     }
7 }

```


而 ProxyFactoryBean 可以简化一下。

 复制代码

```
1 package com.minis.aop;
2 public class ProxyFactoryBean implements FactoryBean<Object>, BeanFactoryAware {
3     private BeanFactory beanFactory;
4     private AopProxyFactory aopProxyFactory;
5     private String interceptorName;
6     private String targetName;
7     private Object target;
8     private ClassLoader proxyClassLoader = ClassUtils.getDefaultClassLoader();
9     private Object singletonInstance;
10    private PointcutAdvisor advisor;
11    public ProxyFactoryBean() {
12        this.aopProxyFactory = new DefaultAopProxyFactory();
13    }
14
15    //省略一些getter/setter
16
17    protected AopProxy createAopProxy() {
18        return getAopProxyFactory().createAopProxy(target, this.advisor);
19    }
20    @Override
21    public Object getObject() throws Exception {
22        initializeAdvisor();
23        return getSingletonInstance();
24    }
25    private synchronized void initializeAdvisor() {
26        Object advice = null;
27        MethodInterceptor mi = null;
28        try {
29            advice = this.beanFactory.getBean(this.interceptorName);
30        } catch (BeansException e) {
31            e.printStackTrace();
32        }
33        this.advisor = (PointcutAdvisor) advice;
34    }
35    private synchronized Object getSingletonInstance() {
36        if (this.singletonInstance == null) {
37            this.singletonInstance = getProxy(createAopProxy());
38        }
39        return this.singletonInstance;
40    }
41 }
```

可以看到，ProxyFactoryBean 中的 initializeAdvisor 方法里，不再需要判断不同的 Interceptor 类型，相关实现被抽取到了 NameMatchMethodPointcutAdvisor 这个类中。


测试

最后，我们还是用以前的 HelloWorldBean 作为测试，现在可以这么写测试程序了。

 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 }
```

配置文件就是我们最早希望达成的样子。

 复制代码

```
1 <bean id="realaction" class="com.test.service.Action1" />
2 <bean id="beforeAdvice" class="com.test.service.MyBeforeAdvice" />
3 <bean id="advisor" class="com.minis.aop.NameMatchMethodPointcutAdvisor">
4     <property type="com.minis.aop.Advice" name="advice" ref="beforeAdvice"/>
5     <property type="String" name="mappedName" value="do*"/>
6 </bean>
7 <bean id="action" class="com.minis.aop.ProxyFactoryBean">
8     <property type="String" name="interceptorName" value="advisor" />
9     <property type="java.lang.Object" name="target" ref="realaction"/>
10 </bean>
```

使用了新的 Advisor，**匹配规则是 "do*"，真正执行的类是 Action1。**

 复制代码

```
1 package com.test.service;
```

```
2 public class Action1 implements IAction {
3     @Override
4     public void doAction() {
5         System.out.println("really do action1");
6     }
7     @Override
8     public void doSomething() {
9         System.out.println("really do something");
10    }
11 }
```

这个 Action1 里面有两个方法，**doAction** 和 **doSomething**，名字都是以 do 开头的。因此，上面的配置规则会使业务程序在调用它们二者的时候，动态插入定义在 MyBeforeAdvice 里的逻辑。

小结

这节课，我们对查找方法名的办法进行了扩展，让系统可以按照某个规则来匹配方法名，这样便于统一处理。这个概念叫做 Pointcut，熟悉数据库操作的人，可以把这个概念类比为 SQL 语句中的 where 条件。

基本的实现思路是使用一个特殊的 Advisor，这个 Advisor 接收一个模式串，而这个模式串也是可以由用户配置在外部文件中的，然后提供 isMatch() 方法，支持按照名称进行模式匹配。具体的字符串匹配工作，采用从前到后的扫描技术，分节段进行校验。

这两节课我们接触到了几个概念，我们再梳理一下。

Join Point：连接点，连接点的含义是指明切面可以插入的地方，这个点可以在函数调用时，或者正常流程中某一行等位置，加入切面的处理逻辑，来实现代码增强的效果。

Advice：通知，表示在特定的连接点采取的操作。

Advisor：通知者，它实现了 Advice。

Interceptor：拦截器，作用是拦截流程，方便处理。

Pointcut：切点。

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

课后题

学完这节课的内容，我也给你留一道思考题。

我们现在实现的匹配规则是按照 * 模式串进行匹配，如果需要支持不同的规则，应该如何改造我们的框架呢？

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 (3)



青莲

2023-05-05 来自浙江

每个匹配模式都可以实现PointcutAdvisor接口，遵循单一职责，如果要同时支持几种能力，可以考虑拐出一个管理类组合几种接口使用

作者回复：赞



欧阳利

2023-04-30 来自广东

为什么Interceptor需要实现Advice接口

作者回复：因为需要把interceptor, beforeadvice和afteradvice几种统一处理



不是早晨，就是黄昏

2023-04-24 来自河南

能不能说明以下Advice接口和Advisor接口之间的关系，更进一步的是设计上的关系。

作者回复: advice是真正要动态插入的业务增强逻辑。advisor则是一个管理类，它包了一个advise，还能寻找到符合条件的方法名进行增强。

