

18 | 拦截器：如何在方法前后进行拦截？

2023-04-21 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。

前面，我们用 JDK 动态代理技术实现了 AOP，并且进行了解耦，采用 IoC 容器来管理代理对象，实现了非侵入式编程。我们现在能在不影响业务代码的前提下，进行逻辑的增强工作，比如打印日志、事务处理、统计接口耗时等等，将这些例行性逻辑作为一种增强放在代理中，运行时动态插入（编织）进去。

有了这个雏形，我们自然就会进一步考虑，在这个代理结构的基础上，将动态添加逻辑这件事情做得更加结构化一点，而不是全部简单地堆在 `invoke()` 方法里。

引入三个概念

我们先来看看 `invoke()` 这个方法的代码在结构方面有什么问题。

[📄 复制代码](#)

```
1 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
2     if (method.getName().equals("doAction")) {
3         System.out.println("-----before call real object, dynamic proxy.....");
4         return method.invoke(target, args);
5     }
6     return null;
7 }
```

我们看到，在实际调用某个方法的时候，是用的反射直接调用 method，对应代码里也就是 `method.invoke(target, args);` 这一句。而增强的例行性代码是直接写在 `method.invoke()` 这个方法前面的，也就是上面代码里的 `System.out.println()`。这么做当然没有错，不过扩展性不好。这里我们还是使用那个老办法，**不同的功能由不同的部件来做**，所以这个增强逻辑我们可以考虑抽取出一个专门的部件来做，实际业务方法的调用也可以包装一下。

所以这节课，我们引入以下几个概念。

Advice：表示这是一个增强操作。

Interceptor：拦截器，它实现的是真正的增强逻辑。

MethodInterceptor：调用方法上的拦截器，也就是它实现在某个方法上的增强。

通过这几个概念，我们就可以把例行性逻辑单独剥离出来了。现在我们要做一个切面，只需要实现某个 Interceptor 就可以了。


对应地，我们定义一下 Advice、Interceptor、MethodInterceptor 这几个接口。

[📄 复制代码](#)

```
1 package com.minis.aop;
2 public interface Advice {
3 }
```

[📄 复制代码](#)


```
1 package com.minis.aop;
2 public interface Interceptor extends Advice{
3 }
}
```

 复制代码

```
1 package com.minis.aop;
2 public interface MethodInterceptor extends Interceptor{
3     Object invoke(MethodInvocation invocation) throws Throwable;
4 }
```


MethodInterceptor 就是方法上的拦截器，对外就是一个 invoke() 方法。拦截器不仅仅会增强逻辑，它内部也会调用业务逻辑方法。因此，对外部程序而言，只需要使用这个 MethodInterceptor 就可以了。

它需要传入一个 MethodInvocation，然后调用 method invocation 的 proceed() 方法，MethodInvocation 实际上就是以前通过反射方法调用业务逻辑的那一段代码的包装。。

 复制代码

```
1 public interface MethodInvocation {
2     Method getMethod();
3     Object[] getArguments();
4     Object getThis();
5     Object proceed() throws Throwable;
6 }
```


我们再来看一下应用程序员的工作，为了插入切面，需要在 invoke() 中实现自己的业务增强代码。

 复制代码

```
1 public class TracingInterceptor implements MethodInterceptor {
2     public Object invoke(MethodInvocation i) throws Throwable {
3         System.out.println("method "+i.getMethod()+" is called on "+
4                             i.getThis()+" with args "+i.getArguments());
5         Object ret=i.proceed();
6         System.out.println("method "+i.getMethod()+" returns "+ret);
7         return ret;
8     }
}
```

```
9 }
```


中间的 `i.proceed()` 才是真正的目标对象的方法调用。

 复制代码

```
1 public Object proceed() throws Throwable {  
2     return this.method.invoke(this.target, this.arguments);  
3 }
```


改造代理类

有了上面准备好的这些部件，我们在动态代理中如何使用它们呢？这里我们再引入一个 `Advisor` 接口。

 复制代码

```
1 public interface Advisor {  
2     MethodInterceptor getMethodInterceptor();  
3     void setMethodInterceptor(MethodInterceptor methodInterceptor);  
4 }
```


在代理类 `ProxyFactoryBean` 里增加 `Advisor` 属性和拦截器。

 复制代码

```
1 private String interceptorName;  
2 private Advisor advisor;
```

这样，我们的代理类里就有跟拦截器关联的点了。

接下来，为了在目标对象调用前进行拦截，我们就需要调整这个 `ProxyFactoryBean`，并设置其 `Advisor` 属性，同时定义这个 `initializeAdvisor` 方法来进行关联。

 复制代码

```
1 package com.minis.aop;
```


```

2 public class ProxyFactoryBean implements FactoryBean<Object> {
3     private BeanFactory beanFactory;
4     private String interceptorName;
5     private Advisor advisor;
6
7     private synchronized void initializeAdvisor() {
8         Object advice = null;
9         MethodInterceptor mi = null;
10        try {
11            advice = (MethodInterceptor) this.beanFactory.getBean(this.intercepto
12        } catch (BeansException e) {
13            e.printStackTrace();
14        }
15        advisor = new DefaultAdvisor();
16        advisor.setMethodInterceptor((MethodInterceptor)advice);
17    }
18 }

```

通过 ProxyFactoryBean 代码实现可以看出，里面新增了 initializeAdvisor 处理，将应用程序自定义的拦截器获取到 Advisor 里。并且，可以在 IoC 容器中配置这个 Interceptor 名字。

在 initializeAdvisor 里，我们把 Advisor 初始化工作交给了 DefaultAdvisor。

 复制代码

```

1 package com.minis.aop;
2 public class DefaultAdvisor implements Advisor{
3     private MethodInterceptor methodInterceptor;
4     public DefaultAdvisor() {
5     }
6     public void setMethodInterceptor(MethodInterceptor methodInterceptor) {
7         this.methodInterceptor = methodInterceptor;
8     }
9     public MethodInterceptor getMethodInterceptor() {
10        return this.methodInterceptor;
11    }
12 }

```

随后，我们修改 AopProxyFactory 中 createAopProxy 接口的方法签名，新增 Advisor 参数。

[📄 复制代码](#)

```
1 package com.minis.aop;
2 public interface AopProxyFactory {
3     AopProxy createAopProxy(Object target, Advisor advisor);
4 }
```

修改接口后，我们需要相应地修改其实现方法。在 ProxyFactoryBean 中，唯一的实现方法就是 createAopProxy()。

[📄 复制代码](#)

```
1 protected AopProxy createAopProxy() {
2     return getAopProxyFactory().createAopProxy(target);
3 }
```

在这个方法中，我们对前面引入的 Advisor 进行了赋值。修改之后，代码变成了这样。

[📄 复制代码](#)

```
1 protected AopProxy createAopProxy() {
2     return getAopProxyFactory().createAopProxy(target, this.advisor);
3 }
```

默认实现是 DefaultAopProxyFactory 与 JdkDynamicAopProxy，这里要一并修改。

[📄 复制代码](#)

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

[📄 复制代码](#)

```
1 package com.minis.aop;
2 public class JdkDynamicAopProxy implements AopProxy, InvocationHandler {
```

```

3     Object target;
4     Advisor advisor;
5     public JdkDynamicAopProxy(Object target, Advisor 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 Thrown
16        if (method.getName().equals("doAction")) {
17            Class<?> targetClass = (target != null ? target.getClass() : null);
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 }
26

```

在 JdkDynamicAopProxy 里我们发现，invoke 方法和之前相比有了不小的变化，在调用某个方法的时候，不再是直接用反射调用方法了，而是先拿到 Advisor 里面的 Interceptor，然后把正常的 method 调用包装成 ReflectiveMethodInvocation，最后调用 interceptor.invoke(invocation)，对需要调用的方法进行了增强处理。

你把这一段和之前的 invoke() 进行比对，可以看出，通过 Interceptor 这个概念，我们就把增强逻辑单独剥离出来了。

你可以看一下实际的 ReflectiveMethodInvocation 类，其实就是对反射调用方法进行了一次包装。

 复制代码

```

1 package com.minis.aop;
2 public class ReflectiveMethodInvocation implements MethodInvocation{
3     protected final Object proxy;
4     protected final Object target;

```

```

5     protected final Method method;
6     protected Object[] arguments;
7     private Class<?> targetClass;
8     protected ReflectiveMethodInvocation(
9         Object proxy, Object target, Method method, Object[] arguments,
10        Class<?> targetClass) {
11        this.proxy = proxy;
12        this.target = target;
13        this.targetClass = targetClass;
14        this.method = method;
15        this.arguments = arguments;
16    }
17
18    //省略getter/setter
19
20    public Object proceed() throws Throwable {
21        return this.method.invoke(this.target, this.arguments);
22    }
23 }

```

测试

我们现在可以来编写一下测试代码，定义 TracingInterceptor 类模拟业务拦截代码。

 复制代码

```

1 package com.test.service;
2 import com.minis.aop.MethodInterceptor;
3 import com.minis.aop.MethodInvocation;
4 public class TracingInterceptor implements MethodInterceptor {
5     @Override
6     public Object invoke(MethodInvocation i) throws Throwable {
7         System.out.println("method "+i.getMethod()+" is called on "+
8             i.getThis()+" with args "+i.getArguments());
9         Object ret=i.proceed();
10        System.out.println("method "+i.getMethod()+" returns "+ret);
11        return ret;
12    }
13 }

```

applicationContext.xml 配置文件：


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

配置文件里，除了原有的 target，我们还增加了一个 interceptorName 属性，让程序员指定需要启用什么样的增强。

到这里，我们就实现了 MethodInterceptor。

在方法前后拦截

我们现在实现的方法拦截，允许程序员自行编写 invoke() 方法，进行任意操作。但是在许多场景下，调用方式实际上是比较固定的，即在某个方法调用之前或之后，允许程序员插入业务上需要的增强。为了满足这种情况，我们可以提供特定的方法拦截，并允许程序员在这些拦截点之前和之后进行业务增强的操作。这种方式就大大简化了程序员的工作。

所以这里我们新增两种 advice：MethodBeforeAdvice 和 AfterReturningAdvice。根据名字也可以看出来，它们分别对应方法调用前处理和返回后的处理。你可以看一下它们的定义。

```
1 package com.minis.aop;
2 public interface BeforeAdvice extends Advice{
3 }
```

复制代码


```
1 package com.minis.aop;
2 public interface AfterAdvice extends Advice{
3 }
```

复制代码

```
1 package com.minis.aop;
2 import java.lang.reflect.Method;
```

复制代码


```
3 public interface MethodBeforeAdvice extends BeforeAdvice {
4     void before(Method method, Object[] args, Object target) throws Throwable;
5 }
```

 复制代码

```
1 package com.minis.aop;
2 import java.lang.reflect.Method;
3 public interface AfterReturningAdvice extends AfterAdvice{
4     void afterReturning(Object returnValue, Method method, Object[] args, Object
5 }
```

首先我们定义通用接口 `BeforeAdvice` 与 `AfterAdvice`，随后定义核心的 `MethodBeforeAdvice` 与 `AfterReturningAdvice` 接口，它们分别内置了 `before` 方法和 `afterReturning` 方法。由方法签名可以看出，这两者的区别在于 `afterReturning` 它内部传入了返回参数，说明是目标方法执行返回后，再调用该方法，在方法里面可以拿到返回的参数。

有了新的 `Advice` 的定义，我们就可以实现新的 `Interceptor` 了。你可以看下实现的代码。

 复制代码

```
1 package com.minis.aop;
2 public class MethodBeforeAdviceInterceptor implements MethodInterceptor, BeforeAd
3     private final MethodBeforeAdvice advice;
4     public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
5         this.advice = advice;
6     }
7     @Override
8     public Object invoke(MethodInvocation mi) throws Throwable {
9         this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());
10        return mi.proceed();
11    }
12 }
```

在这个 `Interceptor` 里，`invoke()` 方法的实现实际上就是限制性地使用 `advice.before()` 方法，然后执行目标方法的调用，也意味着这是在方法调用之前插入的逻辑。由于这是针对 `before` 这种行为的特定 `Interceptor`，因此上层应用程序员无需自己再进行实现，而是可以直接使用这个 `Interceptor`。

```
1 package com.minis.aop;
2 public class AfterReturningAdviceInterceptor implements MethodInterceptor, AfterA
3     private final AfterReturningAdvice advice;
4     public AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {
5         this.advice = advice;
6     }
7     @Override
8     public Object invoke(MethodInvocation mi) throws Throwable {
9         Object retVal = mi.proceed();
10        this.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(), mi.
11        return retVal;
12    }
13 }
14
```

同样，由 `AfterReturningAdviceInterceptor` 类中对 `invoke` 方法的实现可以看出，是先调用 `mi.proceed()` 方法获取到了返回值 `retVal`，再调用 `afterReturning` 方法，实现的是方法调用之后的逻辑增强，这个时序也是固定的。所以注意了，在 `advice.afterReturing()` 方法中，是可以拿到目标方法的返回值的。

在拦截器的使用中，存在一个有意思的问题，同时也是一个有着广泛争议的话题：拦截器是否应该影响业务程序的流程？比如，在 `before()` 拦截器中加入一个返回标志（`true/false`），当其为 `false` 时，我们就中止业务流程并且不再调用目标方法。

不同的开发者对于这个问题有着不同的主张。一方面，这种机制使得开发者能够根据需要对业务逻辑进行精细控制；另一方面，过度使用这种机制也可能会导致代码难度增加、可维护性降低等问题。因此，在使用拦截器的时候，需要在开发效率和程序可维护性之间做出一个平衡，并根据实际情况做出相应的选择。

现在我们手上有三种 `Advice` 类型了，普通的 `MethodInterceptor`，还有特定的 `MethodBeforeAdviceInterceptor` 和 `AfterReturningAdviceInterceptor`，自然在 `ProxyFactoryBean` 中也要对这个 `initializeAdvisor` 方法进行改造，分别支持三种不同类型的 `Advice`。

```

1 package com.minis.aop;
2 public class ProxyFactoryBean implements FactoryBean<Object>, BeanFactoryAware {
3     private synchronized void initializeAdvisor() {
4         Object advice = null;
5         MethodInterceptor mi = null;
6         try {
7             advice = this.beanFactory.getBean(this.interceptorName);
8         } catch (BeansException e) {
9             e.printStackTrace();
10        }
11        if (advice instanceof BeforeAdvice) {
12            mi = new MethodBeforeAdviceInterceptor((MethodBeforeAdvice)advice);
13        }
14        else if (advice instanceof AfterAdvice) {
15            mi = new AfterReturningAdviceInterceptor((AfterReturningAdvice)advice);
16        }
17        else if (advice instanceof MethodInterceptor) {
18            mi = (MethodInterceptor)advice;
19        }
20        advisor = new DefaultAdvisor();
21        advisor.setMethodInterceptor(mi);
22    }
23 }

```

上述实现比较简单，根据不同的 Advice 类型进行判断，最后统一用 MethodInterceptor 来封装。

测试

在这一步改造完毕后，我们测试一下，这里我们提供的是比较简单的实现，实际开发过程中你可以跟据自己的需求定制开发。

我们先提供两个 Advice。

[复制代码](#)

```
1 package com.test.service;
2 public class MyAfterAdvice implements AfterReturningAdvice {
3     @Override
4     public void afterReturning(Object returnValue, Method method, Object[] args,
5         System.out.println("-----my interceptor after method call-----")
6     }
7 }
```

[复制代码](#)

```
1 package com.test.service;
2 public class MyBeforeAdvice implements MethodBeforeAdvice {
3     @Override
4     public void before(Method method, Object[] args, Object target) throws Throwable
5         System.out.println("-----my interceptor before method call-----")
6     }
7 }
```

上述的测试代码都很简单，在此不多赘述。相应的 applicationContext.xml 这个配置文件里面的内容也要发生变化。

[复制代码](#)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans>
3     <bean id="myBeforeAdvice" class="com.test.service.MyBeforeAdvice" />
4     <bean id="realaction" class="com.test.service.Action1" />
5     <bean id="action" class="com.minis.aop.ProxyFactoryBean" >
6         <property type="java.lang.Object" name="target" ref="realaction"/>
7         <property type="String" name="interceptorName" value="myBeforeAdvice"/>
8     </bean>
9 </beans>
```

将 beforeAdvice 或者 afterAdvice 放在配置文件里，除了注册的 Bean 类名有一些修改，其配置是没有发生任何别的变化的，但经过这样一番改造，我们就能使用上述三类 Advice，来对我们的业务代码进行拦截增强处理了。

小结

这节课我们在简单动态代理结构的基础上，**将动态添加的逻辑设计得更加结构化一点，而不是全部简单地堆在 `invoke()` 一个方法中**。为此，我们提出了 Advice 的概念，表示这是一个增强操作。然后提出 Interceptor 拦截器的概念，它实现了真正的增强逻辑并包装了目标方法的调用，应用程序中实际使用的就是这个 Interceptor。我们实际实现的是 MethodInterceptor，它表示的是调用方法上的拦截器。

我们注意到大部分拦截的行为都是比较固定的，或者在方法调用之前，或者在之后，为了方便处理这些常见的场景，我们进一步分离出了 beforeAdvice 和 afterAdvice。通过这些工作，用户希望插入的例行性逻辑现在都单独抽取成一个部件了，应用程序员只要简单地实现 MethodBeforeAdvice 和 AfterReturningAdvice 即可。整个软件结构化很好，完全解耦。

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。如果我们希望 beforeAdvice 能在某种情况下阻止目标方法的调用，应该从哪里下手改造？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 (1)



Geek_5c6106

2023-05-19 来自湖北

老师，请问一下 ProxyFactoryBean bean中的BeanFactory beanFactory属性是在什么时候的设置值的？

这个是要在AbstractBeanFactory中设置值的吗？

```
if (singleton instanceof FactoryBean) {  
    return this.getObjectForBeanInstance(singleton, beanName);  
}
```

getObjectForBeanInstance 方法中将this，将值设置到ProxyFactoryBean 中吗？

作者回复: 它实现了BeanFactoryAware接口, 容器对这个接口会有特殊处理:调用setBeanFactory()方法。

