

17 | 动态代理：如何在运行时插入逻辑？

2023-04-19 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。

从这节课开始，我们就要进入 AOP 环节了。在学习之前，我们先来了解一下是 AOP 怎么回事。

AOP，就是面向切面编程（Aspect Orient Programming），这是一种思想，也是对 OOP 面向对象编程的一种补充。你可能会想：既然已经存在 OOP 面向对象编程了，为什么还需要 AOP 面向切面编程呢？

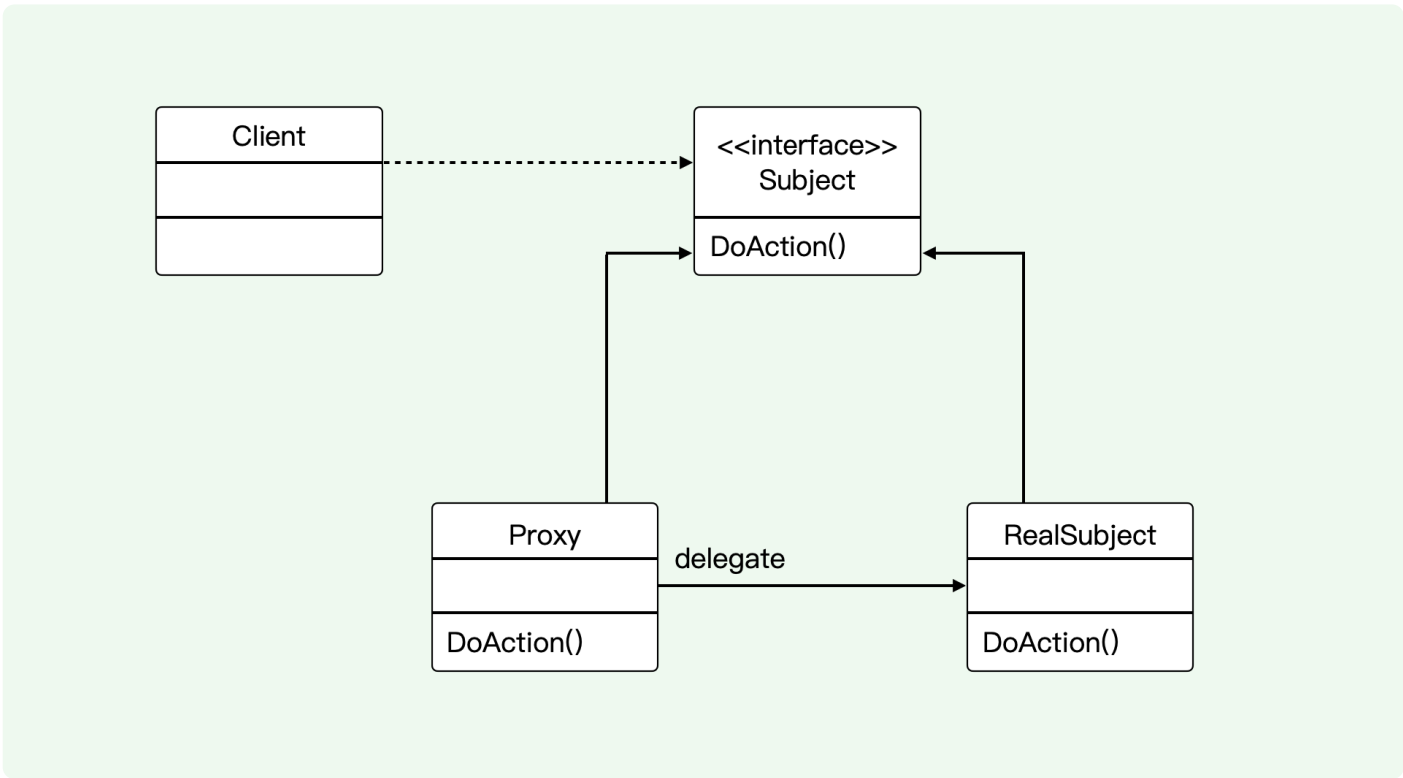
这是因为在许多场景下，一个类的方法中，除了业务逻辑，通常还会包括其他比较重要但又不算主业务逻辑的例行性逻辑代码，比如常见的日志功能，它不影响我们的主业务逻辑，但又能在必要时定位问题，几乎每一个业务方法中都需要。又比如权限检查、事务处理，还有性能监控等等，都是这种情况。

显而易见，日志这类例行性逻辑，在任何一个业务方法实现中都是需要的。如果简单地将这些代码写在业务方法中，会出现两个后果，第一，我们就会将日志之类的代码重复地编写多次；第二，一个业务方法中会包含很多行例行代码，去看源代码会发现方法中多数语句不是在做业务处理。

有专业进取心的程序员就会思考一个问题，**有没有办法将这些例行性逻辑单独抽取出来，然后在程序运行的时候动态插入到业务逻辑中呢？**正是因为这个疑问，AOP 应运而生了。这个问题听起来似乎无解，程序在运行时改变程序本身，似乎有点不可思议。我们研究一下 Java，就会惊奇地发现，Java 里面早就给我们提供了一个手段：**动态代理**。我们可以利用它来开展我们的工作。

代理模式

我们一步步来，先从代理讲起。




GoF的《设计模式》一书中的代理模式类图

看图，我们知道真正干活儿的类是 RealSubject，具体则是由 DoAction() 执行任务。Proxy 作为代理提供一个同样的 DoAction()，然后调用 RealSubject 的 DoAction()。它们都实现 Subject 接口，而 Client 应用程序操作的是 Subject 接口。


简单说来，就是在 Client 应用程序与真正的服务程序 RealSubject 之间增加了一个 Proxy。

我们举例说明，先定义一个服务类接口。

 复制代码


```
1 public interface Subject {  
2     String doAction(String name);  
3 }
```

再定义具体的服务类。

 复制代码

```
1 public class RealSubject implements Subject {  
2     public String doAction(String name) {  
3         System.out.println("real subject do action "+name);  
4         return "SUCCESS";  
5     }  
6 }
```

最后再定义一个代理类。

 复制代码

```
1 public class ProxySubject implements Subject {  
2     Subject realsubject;  
3     public ProxySubject() {  
4         this.realsubject = new RealSubject();  
5     }  
6     public String doAction(String name) {  
7         System.out.println("proxy control");  
8         String rtnValue = realsubject.doAction(name);  
9         return "SUCCESS";  
10    }  
11 }
```

通过代码我们看到，代理类内部包含了一个真正的服务类，而代理类给外部程序提供了和真正的服务类同样的接口。当外部应用程序调用代理类的方法时，代理类内部实际上会转头调用真

正的服务类的相应方法，然后把真正的服务类的返回值直接返回给外部程序。这样做就隐藏了真正的服务类的实现细节。

同时，在调用真正的服务方法之前和之后，我们还可以在代理类里面做一点手脚，加上额外的逻辑，比如上面程序中的 `System.out.println("proxy control");`，这些额外的代码，大部分都是一些例行性的逻辑，如权限和日志等。

最后我们提供一个客户程序使用这个代理类。

 复制代码

```
1 public class Client {
2     public static void main(String[] args) {
3         Subject subject = new ProxySubject();
4         subject.doAction("Test");
5     }
6 }
```


总结一下，代理模式能够让我们在业务处理之外添加例行性逻辑。但是这个经典的模式在我们这里不能直接照搬，因为这个代理是静态的，要事先准备好。而我们需要的是在相关的业务逻辑执行的时候，动态插入例行性逻辑，不需要事先手工静态地准备这些代理类。解决方案就是 **Java 中的动态代理技术**。

动态代理

Java 提供的动态代理可以对接口进行代理，在代理的过程中主要做三件事。


1. 实现 `InvocationHandler` 接口，重写接口内部唯一的方法 `invoke`。
2. 使用 `Proxy` 类，通过 `newProxyInstance`，初始化一个代理对象。
3. 通过代理对象，代理其他类，对该类进行增强处理。

这里我们还是举例说明。首先定义一个 `IAction` 接口。

 复制代码


```
1 package com.test.service;
2 public interface IAction {
3     void doAction();
4 }
```

提供一个具体实现类。

 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 action");
6     }
7 }
```

我们定义了一个 DynamicProxy 类，用来充当代理对象的类。

 复制代码

```
1 package com.test.service;
2 public class DynamicProxy {
3     private Object subject = null;
4
5     public DynamicProxy(Object subject) {
6         this.subject = subject;
7     }
8
9     public Object getProxy() {
10         return Proxy.newProxyInstance(DynamicProxy.class
11             .getClassLoader(), subject.getClass().getInterfaces(),
12             new InvocationHandler() {
13                 public Object invoke(Object proxy, Method method, Object[] args) throws
14                     if (method.getName().equals("doAction")) {
15                         System.out.println("before call real object.....");
16                         return method.invoke(subject, args);
17                     }
18                 return null;
19             }
20         });
21     }
22 }
```

通过这个类的实现代码可以看出，我们使用了 Proxy 类，调用 newProxyInstance 方法构建 IAction 接口的代理对象，而且重写了 InvocationHandler 接口中的 invoke 方法。在重写的方法中我们判断方法名称是否与接口中的 doAction 方法保持一致，随后加上例行性逻辑（print 语句），最后通过反射调用接口 IAction 中的 doAction 方法。


通过这个操作，例行性逻辑就在业务程序运行的时候，动态地添加上去了。

我们编写一个简单的测试程序，就能直观感受到代理的效果了。

 复制代码

```
1 package com.test.controller;
2 public class HelloWorldBean {
3     @Autowired
4     IAction action;
5
6     @RequestMapping("/testaop")
7     public void doTestAop(HttpServletRequest request, HttpServletResponse response) {
8         DynamicProxy proxy = new DynamicProxy(action);
9         IAction p = (IAction)proxy.getProxy();
10        p.doAction();
11
12        String str = "test aop, hello world!";
13        try {
14            response.getWriter().write(str);
15        } catch (IOException e) {
16            e.printStackTrace();
17        }
18    }
19 }
```

运行这个程序，返回内容是 “test aop, hello world! ”。这个时候查看命令行里的内容，你就会发现还有两行输出。

 复制代码

```
1 before call real object.....
2 really do action
```

第一行是代理对象中的输出，第二行是 Action1 中 doAction 方法的实现。

根据这个输出顺序我们发现，这个代理对象达到了代理的效果，在调用 IAction 的具体实现类之前进行了额外的操作，从而增强了代理类。而这个代理是我们动态增加的，而不是事先静态地手工编写一个代理类。

但是读代码，这种方式显然是不美观的，需要在业务逻辑程序中写上，对代码的侵入性太强了。

 复制代码

```
1 DynamicProxy proxy = new DynamicProxy(action);
2 IAction p = (IAction)proxy.getProxy();
```

这个写法跟我们手工写一个代理类实际上相差不多。这种侵入式的做法不是我们推崇的，所以我们要继续前进。


引入 FactoryBean

我们的目标是**非侵入式编程**，也就是应用程序在编程的时候，它不应该手工去创建一个代理，而是使用本来的业务接口，真正的实现类配置在外部，代理类也是配置在外部。

 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 }
```

配置如下：

 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="java.lang.Object" name="target" ref="realaction"/>
4 </bean>
```

业务类中自动注入的是一个 action，也就是上面代码里的 ProxyFactoryBean 类，这个类内部包含了真正干活儿的类 realaction。

这里就有一个初看起来非常奇怪的需求：注册的 action bean 是 ProxyFactoryBean 类，而业务程序使用 `getBean("action")` 的时候，期待返回的又不是这个 Bean 本身，而是内部那个 target。因为只有这样才能让业务程序实际调用 target 中的方法，外面的这个 ProxyFactoryBean 对我们来讲是一个入口，而不是目标。这也就要求，当业务程序使用 `getBean("action")` 方法的时候，这个 ProxyFactoryBean 应该在内部进行进一步地处理，根据 target 再动态生成一个代理返回，达到侵入式编程中下面这两句话的效果。


 复制代码

```
1 DynamicProxy proxy = new DynamicProxy(action);
2 IAction p = (IAction)proxy.getProxy();
```

上面的方案，看起来奇怪，但是确实能解决动态代理的问题。

好，现在我们就按照这个思路动手去实现。首先我们参考 Spring 框架，定义 FactoryBean 接口。


相关代码参考：

 复制代码

```
1 package com.minis.beans.factory;
2 public interface FactoryBean<T> {
3     T getObject() throws Exception;
4     Class<?> getObjectType();
5     default boolean isSingleton() {
6         return true;
7     }
8 }
```


主要的方法就是 getObject(), 从 Factory Bean 中获取内部包含的对象。

接着定义 FactoryBeanRegistrySupport, 提供一部分通用的方法。

 复制代码

```
1 package com.minis.beans.factory.support;
2 public abstract class FactoryBeanRegistrySupport extends DefaultSingletonBeanRegi
3     protected Class<?> getTypeForFactoryBean(final FactoryBean<?> factoryBean) {
4         return factoryBean.getObjectType();
5     }
6     protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String bean
7         Object object = doGetObjectFromFactoryBean(factory, beanName);
8         try {
9             object = postProcessObjectFromFactoryBean(object, beanName);
10        } catch (BeansException e) {
11            e.printStackTrace();
12        }
13        return object;
14    }
15    //从factory bean中获取内部包含的对象
16    private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final
17        Object object = null;
18        try {
19            object = factory.getObject();
20        } catch (Exception e) {
21            e.printStackTrace();
22        }
23        return object;
24    }
25 }
```

最重要的是这个方法: doGetObjectFromFactoryBean(), 从一个 Factory Bean 里面获取内部包含的那个 target 对象。

因为 FactoryBeanRegistrySupport 继承了 DefaultSingletonBeanRegistry, 所以我们接下来可以改写 AbstractBeanFactory, 由原本继承 DefaultSingletonBeanRegistry 改成继承 FactoryBeanRegistrySupport, 保留原有功能的同时增加了功能扩展。

我们重点要修改核心的 getBean() 方法。

```

1 package com.minis.beans.factory.support;
2 public abstract class AbstractBeanFactory extends FactoryBeanRegistrySupport impl
3     public Object getBean(String beanName) throws BeansException{
4         Object singleton = this.getSingleton(beanName);
5         if (singleton == null) {
6             singleton = this.earlySingletonObjects.get(beanName);
7             if (singleton == null) {
8                 System.out.println("get bean null ----- " + beanName);
9                 BeanDefinition bd = beanDefinitionMap.get(beanName);
10                if (bd != null) {
11                    singleton=createBean(bd);
12                    this.registerBean(beanName, singleton);
13                    //beanpostprocessor
14                    //step 1 : postProcessBeforeInitialization
15                    applyBeanPostProcessorsBeforeInitialization(singleton, beanName);
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            }
23            else {
24                return null;
25            }
26        }
27    }
28    else {
29    }
30    //处理factorybean
31    if (singleton instanceof FactoryBean) {
32        return this.getObjectForBeanInstance(singleton, beanName);
33    }
34    else {
35    }
36    return singleton;
37 }
38

```

我们看到在 `getBean()` 这一核心方法中，原有的逻辑处理完毕后，我们新增下面这一段。

```

1 //process Factory Bean
2 if (singleton instanceof FactoryBean) {

```


```
3     return this.getObjectForBeanInstance(singleton, beanName);
4 }
```

根据代码实现可以看出，这里增加了一个判断，如果 Bean 对象是 FactoryBean 类型时，则调用 getObjectForBeanInstance 方法。

 复制代码

```
1 protected Object getObjectForBeanInstance(Object beanInstance, String beanName) {
2     // Now we have the bean instance, which may be a normal bean or a FactoryBean
3     if (!(beanInstance instanceof FactoryBean)) {
4         return beanInstance;
5     }
6     Object object = null;
7     FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
8     object = getObjectFromFactoryBean(factory, beanName);
9     return object;
10 }
```


代码显示，getObjectForBeanInstance 又会调用 doGetObjectFromFactoryBean 方法。

 复制代码

```
1 private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final Str
2     Object object = null;
3     try {
4         object = factory.getObject();
5     } catch (Exception e) {
6         e.printStackTrace();
7     }
8     return object;
9 }
```

最后落实到了 factory.getObject() 里。由此可以看出，我们通过 AbstractBeanFactory 获取 Bean 的时候，对 FactoryBean 进行了特殊处理，获取到的已经不是 FactoryBean 本身了，而是它内部包含的那一个对象。而这个对象，也不是真正底层对应的 Bean。它仍然只是一个代理的对象，我们继续往下看。

我们这个 getObject() 只是 FactoryBean 里的一个接口，接下来我们提供一下它的接口实现——ProxyFactoryBean。

 复制代码


```
1 package com.minis.aop;
2 public class ProxyFactoryBean implements FactoryBean<Object> {
3     private AopProxyFactory aopProxyFactory;
4     private String[] interceptorNames;
5     private String targetName;
6     private Object target;
7     private ClassLoader proxyClassLoader = ClassUtils.getDefaultClassLoader();
8     private Object singletonInstance;
9     public ProxyFactoryBean() {
10         this.aopProxyFactory = new DefaultAopProxyFactory();
11     }
12     public void setAopProxyFactory(AopProxyFactory aopProxyFactory) {
13         this.aopProxyFactory = aopProxyFactory;
14     }
15     public AopProxyFactory getAopProxyFactory() {
16         return this.aopProxyFactory;
17     }
18     protected AopProxy createAopProxy() {
19         return getAopProxyFactory().createAopProxy(target);
20     }
21     public void setInterceptorNames(String... interceptorNames) {
22         this.interceptorNames = interceptorNames;
23     }
24     public void setTargetName(String targetName) {
25         this.targetName = targetName;
26     }
27     public Object getTarget() {
28         return target;
29     }
30     public void setTarget(Object target) {
31         this.target = target;
32     }
33     @Override
34     public Object getObject() throws Exception { //获取内部对象
35         return getSingletonInstance();
36     }
37     private synchronized Object getSingletonInstance() { //获取代理
38         if (this.singletonInstance == null) {
39             this.singletonInstance = getProxy(createAopProxy());
40         }
41         return this.singletonInstance;
42     }
43     protected Object getProxy(AopProxy aopProxy) { //生成代理对象
```

```
44         return aopProxy.getProxy();
45     }
46     @Override
47     public Class<?> getObjectType() {
48         return null;
49     }
50 }
```

这段代码的核心在于，ProxyFactoryBean 在 getObject() 方法中生成了一个代理 getProxy(createAopProxy())，同样也是通过这种方式，拿到了要代理的目标对象。这里的工作就是**创建动态代理**。


基于 JDK 的实现

Spring 作为一个雄心勃勃的框架，自然不会把自己局限于 JDK 提供的动态代理一个技术上，所以，它再次进行了包装，提供了 AopProxy 的概念，JDK 只是其中的一种实现。

 复制代码

```
1 package com.minis.aop;
2 public interface AopProxy {
3     Object getProxy();
4 }
```

还定义了 factory。

 复制代码

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

然后给出了基于 JDK 的实现。

 复制代码

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

```

3     Object target;
4     public JdkDynamicAopProxy(Object target) {
5         this.target = target;
6     }
7     @Override
8     public Object getProxy() {
9         Object obj = Proxy.newProxyInstance(JdkDynamicAopProxy.class.getClassLoad
10         return obj;
11     }
12     @Override
13     public Object invoke(Object proxy, Method method, Object[] args) throws Thrown
14         if (method.getName().equals("doAction")) {
15             System.out.println("-----before call real object, dynamic proxy.....
16             return method.invoke(target, args);
17         }
18         return null;
19     }
20 }

```

 复制代码

```

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

```


在这个实现里，我们终于看到了我们曾经熟悉的 `Proxy.newProxyInstance()` 和 `invoke()`。利用 Java 的动态代理技术代理了目标对象，而这也是 `ProxyFactoryBean` 里真正要返回的 `Object`。

这就是 Spring AOP 的实现原理。

测试


有了上面的工具，我们的测试程序就不需要再手动构建代理对象了，而是交给框架本身处理。而注入的对象，则通过配置文件注入属性值。

applicationContext.xml 配置中新增一段内容。

 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="java.lang.Object" name="target" ref="realaction"/>
4 </bean>
```

通过配置，我们在 HelloWorldBean 里注入的 IAction 对象就纳入了容器管理之中，因此后续测试的时候，直接使用 action.doAction()，就能实现手动初始化 JDK 代理对象的效果。

 复制代码

```
1 package com.test.controller;
2 public class HelloWorldBean {
3     @Autowired
4     IAction action;
5
6     @RequestMapping("/testaop")
7     public void doTestAop(HttpServletRequest request, HttpServletResponse response) {
8         action.doAction();
9
10        String str = "test aop, hello world!";
11        try {
12            response.getWriter().write(str);
13        } catch (IOException e) {
14            e.printStackTrace();
15        }
16    }
17 }
```

我们终于看到了动态代理的结果。

小结

这节课我们**利用 JDK 动态代理技术实现了 AOP 这个概念**。

我们介绍了代理模式实现的静态代理，然后使用了 JDK 的动态代理技术。在使用动态代理技术的程序代码中，我们发现它是侵入式的，不理想，所以我们就想办法把代理配置在 XML 文

件里了。但是如果按照原有的 Bean 的定义，这个配置在外部文件里的代理 Bean 本身不能代理业务类，我们真正需要的是通过这个代理 Bean 来创建一个动态代理，于是引入了 FactoryBean 的概念，不是直接获取这个 Bean 本身，而是通过里面的 getObject() 获取到 Factory Bean 里面包含的对象。

这样将 IoC 容器里的 Bean 分成了两类：一是普通的 Bean，二是 Factory Bean。在 getObject() 的实现中，我们使用 JDK 的动态代理技术创建了一个代理。这样就实现了 AOP。

另外，Spring 中的代理支持 JDK 代理与 Cglib 代理两种，目前 MiniSpring 定义的 DefaultAopProxyFactory 只支持 JDK 代理。另一种方式我留作思考题，你可以先想一想要怎么实现。

AOP 还有别的实现方案，比如 AspectJ，也比较常用，在实际工程实践中，一般采用的就是 AspectJ，而不是 Spring AOP，因为 AspectJ 更加高效，功能更强。比如，AspectJ 是编译时创建的代理，性能高十倍以上，而且切入点不仅仅在方法上，而是可以在类的任何部分。所以 AspectJ 才是完整的 AOP 解决方案，Spring AOP 不是成功的工业级方案。之所以保留 Spring AOP，一个原因是原理简单、利于理解，另一个是 Rod Johnson 不忍抛弃自己的心血。

完整源代码参见 [🔗 https://github.com/YaleGuo/minis](https://github.com/YaleGuo/minis)

课后题

学完这节课，我也给你留一道思考题。如果 MiniSpring 想扩展到支持 Cglib，程序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。



风轻扬

2023-05-29 来自北京

可以在DefaultAopProxyFactory类中，获取AopProxy时改造。新增一个入参，区分是jdk的接口代理还是cglib代理，进而返回不同的代理

作者回复: 后面有参考回答。



peter

2023-04-20 来自北京

Spring与AspectJ是什么关系？AspectJ是一个独立的工具软件，Spring使用该软件完成AOP，这样理解对吗？

作者回复: 对。跟jdk的动态代理等位的一个实现。



不是早晨，就是黄昏

2023-04-19 来自河南

```
DynamicProxy proxy = new DynamicProxy(action);  
IAction p = (IAction)proxy.getProxy();  
action.doAction();  
这里是不是要写成p.doAction();
```

作者回复: 是的是的，后面改过来。这是线下班的时候一边演示一边跟学生说这个地方最后应该是action.doAction顺手就改了。Github上是正确的。

共 2 条评论 >

