

## 21 | 再回首： 如何实现Spring AOP?

2023-04-28 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。

到这一节课，我们的 Spring AOP 部分也就结束了，你是不是跟随我的这个步骤也实现了自己的 AOP 呢？欢迎你把你的实现代码分享出来，我们一起讨论，共同进步！为了让你对这一章的内容掌握得更加牢固，我们对 AOP 的内容做一个重点回顾。

### 重点回顾

Spring AOP 是 Spring 框架的一个核心组件之一，是 Spring 面向切面编程的探索。面向对象和面向切面，两者一纵一横，编织成一个完整的程序结构。

在 AOP 编程中，Aspect 指的是横切逻辑（cross-cutting concerns），也就是那些和基本业务逻辑无关，但是却是很多不同业务代码共同需要的功能，比如日志记录、安全检查、事务管理，等等。Aspect 能够通过 Join point，Advice 和 Pointcut 来定义，在运行的时候，能够

自动在 Pointcut 范围内的不同类型的 Advice 作用在不同的 Join point 上，实现对横切逻辑的处理。

所以，这个 AOP 编程可以看作是一种以 Aspect 为核心的编程方式，它强调的是将横切逻辑作为一个独立的属性进行处理，而不是直接嵌入到基本业务逻辑中。这样做，可以提高代码的可复用性、可维护性和可扩展性，使得代码更容易理解和设计。

AOP 的实现，是基于 JDK 动态代理的，站在 Java 的角度，这很自然，概念很容易实现，但是效率不高，限制也比较多。可以说 AOP 的实现是 Spring 框架中少数不尽人意的一部分，也可以看出世界顶级高手也有考虑不周到的地方。

那我们在课程中是如何一步步实现 AOP 的呢？

我们是基于 JDK 来实现的，因为比较自然、容易。我们先是引入了 Java 的动态代理技术，探讨如何用这个技术动态插入业务逻辑。然后我们进一步抽取动态业务逻辑，引入 Spring 里的 Interceptor 和 Advice 的概念。之后通过引入 Spring 的 PointCut 概念，进行 advice 作用范围的定义，让系统知道前面定义的 Advice 会对哪些对象产生影响。最后为了免除手工逐个配置 PointCut 和 Interceptor 的工作，我们就通过一个自动化的机制自动生成动态代理。最终实现了一个有模有样的 AOP 解决方案。

好了，回顾完这一章的重点，我们再来看一下我每节课后给你布置的思考题。题目和答案我都放到下面了，不要偷懒，好好思考之后再来看答案。

## 17 | 动态代理：如何在运行时插入逻辑？


### 思考题

如果 MiniSpring 想扩展到支持 Cglib，程序应该从哪里下手改造？

### 参考答案

我们的动态代理包装在 AopProxy 这个接口中，对 JDK 动态代理技术，使用了 JdkDynamicAopProxy 这个类来实现，所以平行的做法，对于 Cglib 技术，我们就可以新增一个 CglibAopProxy 类进行实现。

同时，采用哪一种 AOP Proxy 可以由工厂方法决定，也就是在 ProxyFactoryBean 中所使用的 aopProxyFactory，它在初始化的时候有个默认实现，即 DefaultAopProxyFactory。我们可以将这个类的 createAopProxy() 方法改造一下。

 复制代码

```
1 public class DefaultAopProxyFactory implements AopProxyFactory {
2     public AopProxy createAopProxy(Object target) {
3         if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
4             return new JdkDynamicAopProxy(target);
5         }
6         return new CglibAopProxy(config);
7     }
8 }
```

根据某些条件决定使用 JdkDynamicAopProxy 还是 CglibAopProxy，或者通过配置文件给一个属性来配置也可以。


## 18 | 拦截器：如何在方法前后进行拦截？

### 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改造我们的程序？

### 参考答案

答案在 MethodBeforeAdviceInterceptor 的实现中，看它的 invoke 方法。

 复制代码

```
1 public class MethodBeforeAdviceInterceptor implements MethodInterceptor {
2     public Object invoke(MethodInvocation mi) throws Throwable {
3         this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());
4         return mi.proceed();
5     }
6 }
```

这个方法先调用 `advice.before()`，然后再调用目标方法。所以如果我们希望 `beforeAdvice` 能够阻止流程继续，可以将 `advice.before()` 接口改造成有一个 `boolean` 返回值，规定返回 `false` 则不调用 `mi.proceed()`。


## 19 | Pointcut：如何批量匹配代理方法？

### 思考题

我们现在实现的匹配规则是按照 \* 模式串进行匹配，如果有不同的规则，应该如何改造呢？

### 参考答案


如果仍然按照名字来匹配，那就可以改造 `NameMatchMethodPointcut` 类，它现在的核心代码是：

 复制代码

```
1 public class NameMatchMethodPointcut implements MethodMatcher,Pointcut{
2     private String mappedName = "";
3     protected boolean isMatch(String methodName, String mappedName) {
4         return PatternMatchUtils.simpleMatch(mappedName, methodName);
5     }
6 }
```

默认的实现用的是 `PatternMatchUtils.simpleMatch()`，比较简单的模式串。我们可以给 `PatternMatchUtils` 增加一个方法，如 `regExprMatch()` 正则表达式匹配，在这里接收正则表达式串，进行匹配校验。

如果超出名字匹配的范围，需要用到不一样的匹配规则，就可以并列增加一个 `OtherMatchMethodPointcut` 类 和响应的 `advisor` 类，自己实现。并在配置文件里指定使用这个 `Advisor`。

 复制代码

```
1 <bean id="advisor" class="com.minis.aop.OtherMatchMethodPointcutAdvisor">
2     </bean>
3     <bean id="action" class="com.minis.aop.ProxyFactoryBean">
4         <property type="String" name="interceptorName" value="advisor" />
```

## 20 | AutoProxyCreator：如何自动添加动态代理？

### 思考题

AOP 时常用于数据库事务处理，如何用我们现在的 AOP 架构实现简单的事务处理？

### 参考答案


针对数据库事务，手工代码简化到了极致，就是执行 SQL 之前执行 `conn.setAutoCommit(false)`，在执行完 SQL 之后，再执行 `conn.commit()`。因此，我们用一个 `MethodInterceptor` 就可以简单实现。

假定有了这样一个 `interceptor`。

 复制代码

```
1 <bean id="transactionInterceptor" class="TransactionInterceptor" />
```

这个 `Interceptor` 拦截目标方法后添加事务处理逻辑，因此需要改造一下。

 复制代码

```
1 public class TransactionInterceptor implements MethodInterceptor{
2     @Override
3     public Object invoke(MethodInvocation invocation) throws Throwable {
4         conn.setAutoCommit(false);
5         Object ret=invocation.proceed();
6         conn.commit();
7         return ret;
8     }
9 }
```

从代码里可以看到，这里需要一个 `conn`，因此我们要设法将数据源信息注入到这里。

我们可以抽取出一个 `TransactionManager` 类，大体如下：

[复制代码](#)


```
1 public class TransactionManager {
2     @Autowired
3     private DataSource dataSource;
4     Connection conn = null;
5
6     protected void doBegin() {
7         conn = dataSource.getConnection();
8         if (conn.getAutoCommit()) {
9             conn.setAutoCommit(false);
10        }
11    }
12    protected void doCommit() {
13        conn.commit();
14    }
15 }
```

由这个 transaction manager 负责数据源以及开始和提交事务，然后将这个 transaction manager 作为一个 Bean 注入 Interceptor，因此配置应该是这样的。

[复制代码](#)

```
1 <bean id="transactionInterceptor" class="TransactionInterceptor" >
2     <property type="TransactionManager" name="transactionManager" value="txManage
3 </bean>
4 <bean id="txManager" class="TransactionManager">
5 </bean>
```

所以 Interceptor 最后应该改造成这个样子：

 复制代码

```
1 public class TransactionInterceptor implements MethodInterceptor{
2     TransactionManager transactionManager;
3     @Override
4     public Object invoke(MethodInvocation invocation) throws Throwable {
5         transactionManager.doBegin();
6         Object ret=invocation.proceed();
7         transactionManager.doCommit();
8         return ret;
9     }
10 }
```

© 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)



**peter**

2023-04-29 来自北京

PatternMatchUtils是SDK提供的，怎么增加方法？派生一个类吗？

作者回复: util下的一个工具类，看github

