Spring注解驱动开发第33讲——AOP原理总结

往期精选

- Spring注解驱动开发第25讲——你敢信？面试官竟然让我现场搭建一个AOP测试环境！
- Spring注解驱动开发第26讲——总有人让我给他讲讲@EnableAspectJAutoProxy注解
- Spring注解驱动开发第27讲——为AnnotationAwareAspectJAutoProxyCreator组件里面和后置处理器以及Aware接口有关的方法打上断点
- Spring注解驱动开发第28讲——为你呕心沥血分析创建和注册AnnotationAwareAspectJAutoProxyCreator的过程，这应该是全网分析的最详细的了！
- Spring注解驱动开发第29讲——注册完AnnotationAwareAspectJAutoProxyCreator后置处理器之后，就得完成BeanFactory的初始化工作了
- Spring注解驱动开发第30讲——AnnotationAwareAspectJAutoProxyCreator作为后置处理器，你知道它都做了些什么吗？
- Spring注解驱动开发第31讲——目标方法的拦截逻辑
- Spring注解驱动开发第32讲——拦截器链的执行过程

经过前面这8讲的学习，相信你对AOP的原理有了一个更深入的认识，而且还是从源码角度去认识的。

AOP原理总结

最后，我们还需要对 AOP原理　做一个简单的总结，完美结束对其研究的旅程。

1. 利用@EnableAspectJAutoProxy注解来开启AOP功能
2. 这个AOP功能是怎么开启的呢？主要是通过@EnableAspectJAutoProxy注解向 IOC容器　中注册一个AnnotationAwareAspectJAutoProxyCreator组件来做到这点的
3. AnnotationAwareAspectJAutoProxyCreator组件是一个后置处理器
4. 该后置处理器是怎么工作的呢？在IOC容器创建的过程中，我们就能清楚地看到这个后置处理器是如何创建以及注册的，以及它的工作流程。
    1. 首先，在创建IOC容器的过程中，会调用refresh　()方法来刷新容器，而在刷新容器的过程中有一步是来注册后置处理器的，如下所示：

        ```
        1 registerBeanPostProcessors(beanFactory); // 注册后置处理器，在这一步会创建AnnotationAwareAspectJAutoProxyCreator对象
        ```
        AI写代码java运行

        其实，这一步会为所有后置处理器都创建对象。
    2. 在刷新容器的过程中还有一步是来完成BeanFactory的初始化工作的，如下所示：

        ```
        1 finishBeanFactoryInitialization(beanFactory); // 完成BeanFactory的初始化工作。所谓的完成BeanFactory的初始化工作，其实就是来创建剩下的单实
        ```
        AI写代码java运行

        很显然，剩下的单实例bean自然就包括MathCalculator（业务逻辑类）和LogAspects（切面类）这两个bean，因此这两个bean就是在这儿被创建的。
        1. 创建业务逻辑组件和切面组件
        2. 在这两个组件创建的过程中，最核心的一点就是AnnotationAwareAspectJAutoProxyCreator（后置处理器）会来拦截这俩组件的创建过程
        3. 怎么拦截呢？主要就是在组件创建完成之后，判断组件是否需要增强。如需要，则会把切面里面的通知方法包装成增强器，然后再为业务逻辑组件创建一个代理对象。我们也认真仔细探究过了，在为业务逻辑组件创建代理对象的时候，使用的是cglib来创建动态代理的。当然了，如果业务逻辑类有实现接口，那么就使用jdk来创建动态代理。一旦这个代理对象创建出来了，那么它里面就会有所有的增强器。

            这个代理对象创建完以后，IOC容器也就创建完了。接下来，便要来执行目标方法了。
5. 执行目标方法
    1. 此时，其实是代理对象来执行目标方法
    2. 使用CglibAopProxy类的intercept()方法来拦截目标方法的执行，拦截的过程如下：
        1. 得到目标方法的拦截器链，所谓的拦截器链其实就是每一个通知方法又被包装为了方法拦截器，即MethodInterceptor
        2. 利用拦截器的链式机制，依次进入每一个拦截器中进行执行
        3. 最终，整个的执行效果就会有两套：
            - 目标方法正常执行：前置通知→目标方法→后置通知→返回通知
            - 目标方法出现异常：前置通知→目标方法→后置通知→异常通知