41 注解 @Aspect 是如何工作的？

更新时间：2020-08-24 10:03:23

> 如果说我比别人看得要远一点，那是因为我站在巨人的肩上。——牛顿

背景

面试官：如何针对某个包下的所有类的方法打印日志？

面试者：可以使用 Spring AOP 的 @Aspect 注解实现，底层由 JDK 动态代理和 CGLib 字节码生产代理实现。

面试官追问：@Aspect 注解是如何工作的？

面试者：

没事，我没哭，真的

Spring AOP @Aspect 注解示例

要拦截的目标

```
package com.davidwang456.test;
public class HelloService {
 public void sayHello(String world) {
  System.out.println("hello "+ world);
 }
}
```

拦截逻辑

```
package com.davidwang456.test;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class LogAspect {
 @Before("execution(* sayHello(..))")
 public void beforeHello() {
  System.out.println("how are you !");
 }
}
```

配置 @Aspect 注解生效

<aop:aspectj-autoproxy />起作用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/aop
 http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

 <aop:aspectj-autoproxy />

 <!-- Target -->
 <bean id="hello" class="com.davidwang456.test.HelloService" />

 <!-- Aspect -->
 <bean id="logAspect" class="com.davidwang456.test.LogAspect" />

</beans>
```


测试类：实现 @Aspect 的功能有 3 种方式：

- 一种方式是使用 XML 配置，利用配置 <aop:aspectj-autoproxy /> 生效注解；
- 一种方式是使用 Java config 方式，使用注解 @EnableAspectJAutoProxy 配置；
- 一种方式是直接利用代码 AspectJProxyFactory（或者 ProxyFactory/ProxyFactoryBean）生成代理。

本篇采用 XML 配置方式，代码如下：

```
package com.davidwang456.test;

import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AspectTest {
 @SuppressWarnings("resource")
 public static void main(String[] args) {
  ApplicationContext context= new ClassPathXmlApplicationContext("com/davidwang456/test/spring.xml");
  HelloService hello=context.getBean(HelloService.class);
  hello.sayHello("world !");
 }

 public static void factory1() {
  HelloService target=new HelloService();
  AspectJProxyFactory factory=new AspectJProxyFactory();
  factory.setTarget(target);
  factory.addAspect(LogAspect.class);

  HelloService proxy=factory.getProxy();
  proxy.sayHello("world !");
 }
}
```

运行程序，控制台输出：

```
how are you !
hello world !
```

Spring AOP @Aspect注解原理分析

<aop:aspectj-autoproxy /> 定义

在 上面的 spring.xml 配置文件中，http://www.springframework.org/schema/aop/spring-aop-3.0.xsd 规范了 <aop:aspectj-autoproxy /> 的定义，如下所示：

```
<xsd:element name="aspectj-autoproxy">
<xsd:annotation>
<xsd:documentation source="java:org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator">
<![CDATA[ Enables the use of the @AspectJ style of Spring AOP. ]]>
</xsd:documentation>
</xsd:annotation>
<xsd:complexType>
<xsd:sequence>
<xsd:element name="include" type="includeType" minOccurs="0" maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>
<![CDATA[ Indicates that only @AspectJ beans with names matched by the (regex) pattern will be considered as defining aspects to use for Spring auto proxying. ]]>
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="proxy-target-class" type="xsd:boolean" default="false">
<xsd:annotation>
<xsd:documentation>
<![CDATA[ Are class-based (CGLIB) proxies to be created? By default, standard Java interface-based proxies are created. ]]>
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="expose-proxy" type="xsd:boolean" default="false">
<xsd:annotation>
<xsd:documentation>
<![CDATA[ Indicate that the proxy should be exposed by the AOP framework as a ThreadLocal for retrieval via the AopContext class. Off by default, i.e. no guarantees that AopContext access will work. ]]>
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
</xsd:complexType>
</xsd:element>
```


从上面可以看到在配置 <aop:aspectj-autoproxy /> 时，org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator 会使 @Aspect 生效。我们来 debug 到内部程序看看。

<aop:aspectj-autoproxy /> 解析

AopNamespaceHandler 解析 <aop:aspectj-autoproxy /> ，注册了 AnnotationAwareAspectJAutoProxyCreator。

```
public class AopNamespaceHandler extends NamespaceHandlerSupport {

 /**
  * Register the {@link BeanDefinitionParser BeanDefinitionParsers} for the
  * '{@code config}', '{@code spring-configured}', '{@code aspectj-autoproxy}'
  * and '{@code scoped-proxy}' tags.
  */
 @Override
 public void init() {
  // In 2.0 XSD as well as in 2.1 XSD.
  registerBeanDefinitionParser("config", new ConfigBeanDefinitionParser());
  registerBeanDefinitionParser("aspectj-autoproxy", new AspectJAutoProxyBeanDefinitionParser());
  registerBeanDefinitionDecorator("scoped-proxy", new ScopedProxyBeanDefinitionDecorator());

  // Only in 2.0 XSD: moved to context namespace as of 2.1
  registerBeanDefinitionParser("spring-configured", new SpringConfiguredBeanDefinitionParser());
 }

}
```

AspectJAutoProxyBeanDefinitionParser 的 parse() 方法调用 AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary() 方法注册 AnnotationAwareAspectJAutoProxyCreator，如下所示：

```
@Override
@Nullable
public BeanDefinition parse(Element element, ParserContext parserContext) {
 AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(parserContext, element);
 extendBeanDefinition(element, parserContext);
 return null;
}
```

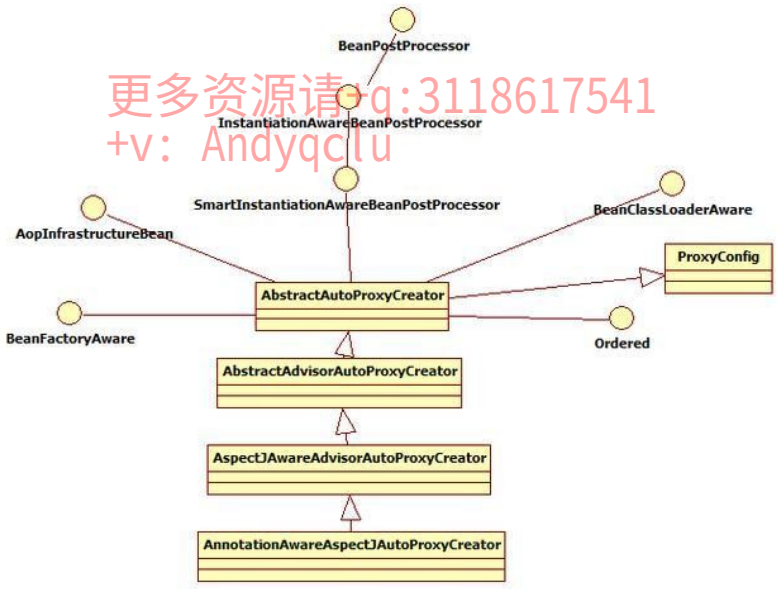
调用实现类：

```
public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
    return registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class,          registry, source);
}
```

那 AnnotationAwareAspectJAutoProxyCreator 如何发现 Aspect 注解，并做处理的呢？

AnnotationAwareAspectJAutoProxyCreator发现Aspect注解原理

首先看看 AnnotationAwareAspectJAutoProxyCreator 是什么？它的继承关系如下：



从上图可以发现，AnnotationAwareAspectJAutoProxyCreator 是一个 BeanPostProcessor，它叫后置处理器，作用是在 Bean 对象在实例化和依赖注入完毕后，在显示调用初始化方法的前后添加我们自己的逻辑。注意是 Bean 实例化完毕后及依赖注入完成后触发的。AnnotationAwareAspectJAutoProxyCreator 的 isInfrastructureClass 判断是否有 @Aspect 注解，如下所示：

```
@Override
protected boolean isInfrastructureClass(Class<?> beanClass) {
 // Previously we setProxyTargetClass(true) in the constructor, but that has too
 // broad an impact. Instead we now override isInfrastructureClass to avoid proxying
 // aspects. I'm not entirely happy with that as there is no good reason not
 // to advise aspects, except that it causes advice invocation to go through a
 // proxy, and if the aspect implements e.g the Ordered interface it will be
 // proxied by that interface and fail at runtime as the advice method is not
 // defined on the interface. We could potentially relax the restriction about
 // not advising aspects in the future.
 return (super.isInfrastructureClass(beanClass) ||
   (this.aspectJAdvisorFactory != null && this.aspectJAdvisorFactory.isAspect(beanClass)));
}
```

AnnotationAwareAspectJAutoProxyCreator生成代理类的过程

可以通过debug程序内部，打印出生成代理的调用链：

调用序号：1 调用类和方法 com.davidwang456.test.AspectTest$main

调用序号：2 调用类和方法 org.springframework.context.support.ClassPathXmlApplicationContext$<init>

调用序号：3 调用类和方法 org.springframework.context.support.ClassPathXmlApplicationContext$<init>

调用序号：4 调用类和方法 org.springframework.context.support.AbstractApplicationContext$refresh

调用序号：5 调用类和方法 org.springframework.context.support.AbstractApplicationContext$finishBeanFactoryInitialization

调用序号：6 调用类和方法 org.springframework.beans.factory.support.DefaultListableBeanFactory$preInstantiateSingletons

调用序号：7 调用类和方法 org.springframework.beans.factory.support.AbstractBeanFactory$getBean

调用序号：8 调用类和方法 org.springframework.beans.factory.support.AbstractBeanFactory$doGetBean

调用序号：9 调用类和方法 org.springframework.beans.factory.support.DefaultSingletonBeanRegistry$getSingleton

调用序号：10 调用类和方法 org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$9/1978869058$getObject

调用序号：11 调用类和方法 org.springframework.beans.factory.support.AbstractBeanFactory$lambda$0

调用序号：12 调用类和方法 org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$createBean

调用序号：13 调用类和方法 org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$doCreateBean

调用序号：14 调用类和方法 org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$initializeBean

调用序号：15 调用类和方法 org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$applyBeanPostProcessorsAfterInitialization

调用序号：16 调用类和方法 org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator$postProcessAfterInitialization

调用序号：17 调用类和方法 org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator$wrapIfNecessary

调用序号：18 调用类和方法 org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator$createProxy

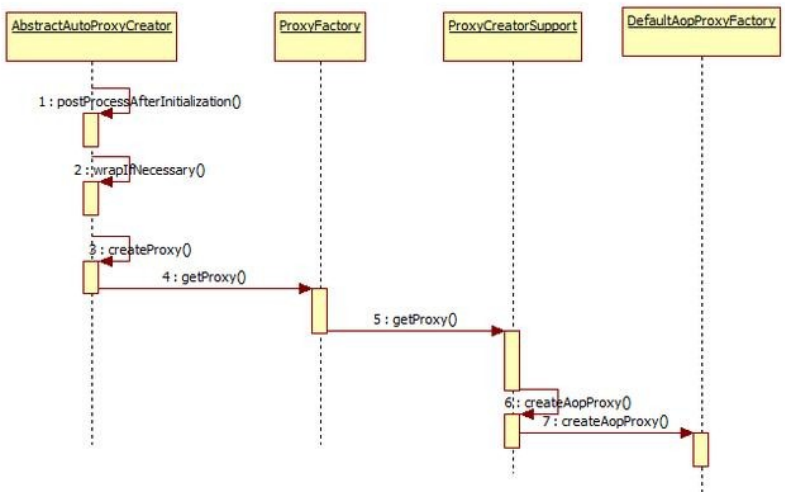
调用序号：19 调用类和方法 org.springframework.aop.framework.ProxyFactory$getProxy

调用序号：20 调用类和方法 org.springframework.aop.framework.ProxyCreatorSupport$createAopProxy

调用序号：21 调用类和方法 org.springframework.aop.framework.DefaultAopProxyFactory$createAopProxy

实现重点为：AbstractAutoProxyCreator 的 postProcessAfterInitialization() 方法

![



DefaultAopProxyFactory#createAopProxy() 方法

```
@Override public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
    if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
        Class<?> targetClass = config.getTargetClass();
        if (targetClass == null) {
            throw new AopConfigException("TargetSource cannot determine target class: " + "Either an interface or a target is required for proxy creation."
);
        }
        if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
            return new JdkDynamicAopProxy(config);
        }
        return new ObjenesisCglibAopProxy(config);
    } else {
        return new JdkDynamicAopProxy(config);
    }
}
```


代理生成默认使用 JDK 自带的代理，使用 CGLIG 的三种情况：

- ProxyConfig 中的 optimize 标识被置为 true；
- ProxyConfig 中的 proxyTargetClass 标识被置为 true；
- 目标类没有可用的代理接口即目标类没有实现接口。

总结

实现 @Aspect 的功能有 3 种方式：

- 一种方式是使用 XML 配置，利用配置 <aop:aspectj-autoproxy /> 生效注解；
- 一种方式是使用 java config 方式，使用注解 EnableAspectJAutoProxy 配置；
- 一种方式是直接利用代码 AspectJProxyFactory（或者ProxyFactory/ProxyFactoryBean）生成代理。

前两种方式都是通过 BeanPostProcessor 的实现类 AnnotationAwareAspectJAutoProxyCreator 完成的，第三种是硬编码形式的。正在的逻辑在父类 AbstractAutoProxyCreator 实现，在 postProcessBeforeInstantiation 方法发现 Aspect 注解，在 postProcessAfterInitialization 方法中创建代理实例。

代理生成默认使用 JDK 自带的代理，使用 CGLIG 的三种情况：

- ProxyConfig 中的 optimize 标识被置为 true；
- ProxyConfig 中的 proxyTargetClass 标识被置为 true；
- 目标类没有可用的代理接口即目标类没有实现接口。

}

40 Spring AOP策略模式使用及示例实战

42 离开Spring AOP，我们如何实现AOP功能？