

32_源码剖析：@Aspect 注解是怎么运作起来的？

1. 开篇

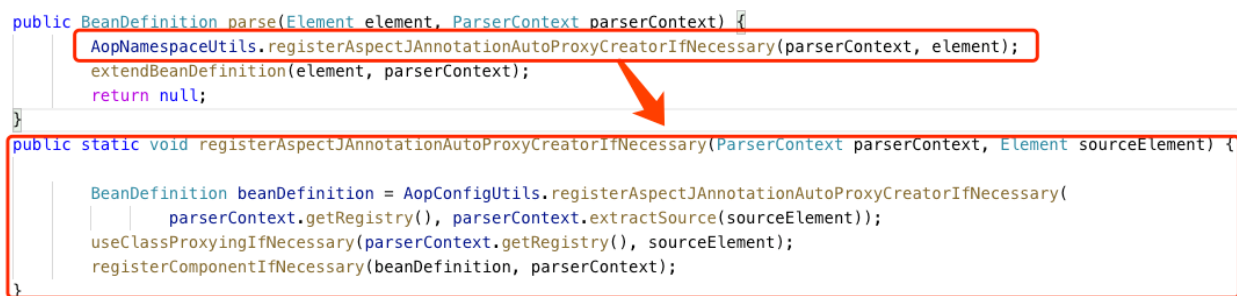
上节课把日志代码通过打包、上传、以及启动服务的方式发布到 ECS，同时登陆服务器查看 logs 目录下面的日志文件，可以看到 ControllerLogAspect 中编写的日志代码产生效果。本节课看看 @Aspect 注解是如何运作起来的，内容包括：

- @Aspect 执行过程

2. @Aspect 执行过程

@Aspect 切面类注解属于 Spring 2.0 以后定义的标签注解，在配置文件 applicationContext.xml 中以 aop:aspectj-autoproxy 的方式开启。其原理是通过代码追踪，在 AopNamespaceHandler 中找到了对这个标签的解析器 AspectJAutoProxyBeanDefinitionParser 类。

先来看看 AspectJAutoProxyBeanDefinitionParser 中的 parse 方法如何对 @Aspect 标签进行解析工作的。如图 1 所示在 parse 方法里面会调用 registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法对解析上下文以及打上注释的元素进行注册。顺势看注册的方法中调用 AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法创建 BeanDefinition，并且把 BeanDefinition 注册到组件中。



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

public static void registerAspectJAnnotationAutoProxyCreatorIfNecessary(ParserContext parserContext, Element sourceElement) {
    BeanDefinition beanDefinition = AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(
        parserContext.getRegistry(), parserContext.extractSource(sourceElement));
    useClassProxyingIfNecessary(parserContext.getRegistry(), sourceElement);
    registerComponentIfNecessary(beanDefinition, parserContext);
}
```

图 1 parse

从前面的课程知道 BeanDefinition 是 Spring IoC 容器对 bean 进行存储和管理的模式，接着看 registerAspectJAnnotationAutoProxyCreatorIfNecessary 是如何生成 BeanDefinition 的。如图 2 所示，

registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法会直接调用 registerOrEscalateApcAsRequired 方法产生 BeanDefinition。其主要操作就是建立一个 RootBeanDefinition 作为 BeanDefinition 的根，然后设置其源、角色以及对 BeanDefinition 的注册。

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

private static BeanDefinition registerOrEscalateApcAsRequired(Class<?> cls, BeansDefinitionRegistry registry, Object source) {
    Assert.notNull(registry, "BeansDefinitionRegistry must not be null");
    if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
        BeanDefinition apcDefinition = registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
        if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
            int currentPriority = findPriorityForClass(apcDefinition.getBeanClassName());
            int requiredPriority = findPriorityForClass(cls);
            if (currentPriority < requiredPriority) {
                apcDefinition.setBeanClassName(cls.getName());
            }
        }
        return null;
    }
    RootBeanDefinition beanDefinition = new RootBeanDefinition(cls); //封装一个代理器;
    beanDefinition.setSource(source);
    beanDefinition.getPropertyValues().add("order", Ordered.HIGHEST_PRECEDENCE);
    beanDefinition.setRole(BeansDefinition.ROLE_INFRASTRUCTURE);
    registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME, beanDefinition);
    return beanDefinition;
}
```

图 2 registerAspectJAnnotationAutoProxyCreatorIfNecessary

经过以上处理 AnnotationAwareAspectJAutoProxyCreator 类就被创建处理了，而这个类的功能就是完成 AOP 增强的，它继承了父类 AbstractAutoProxyCreator，同时其父类实现了 BeanPostProcessor 接口。它在实现该接口后，Spring 加载 bean 时会其实在实例化前调用其 postProcessBeforeInstantiation 方法，接下来看这个方法。如图 3 所示，postProcessBeforeInstantiation 方法中最重要的部分用红框框起来了，先添加目标 bean，然后获取有 @Aspect 注释类中的增强方法，最后根据拦截器 specificInterceptors 对源目标类创建增强代理。

```

public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {
    Object cacheKey = getCacheKey(beanClass, beanName);
    if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {
        if (this.advisedBeans.containsKey(cacheKey)) {
            return null;
        }
        if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return null;
        }
    }

    // Create proxy here if we have a custom TargetSource.
    // Suppresses unnecessary default instantiation of the target bean:
    // The TargetSource will handle target instances in a custom fashion.
    if (beanName != null) {
        TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
        if (targetSource != null) {
            this.targetSourcedBeans.add(beanName);
            //获取存在于aspect注解类中的增强方法;
            Object[] specificInterceptors = getAdvisesAndAdvisorsForBean(beanClass, beanName, targetSource);
            //根据增强器创建代理
            Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }
    }

    return null;
}

```

图 3 postProcessBeforeInstantiation

按照上面的代码描述是由 `getAdvisesAndAdvisorsForBean` 来生成拦截器的，它包括如下工作：

获取 `beanFactory` 中所有 `@Aspect` 注解的类

对标记类进行增强提取，提取之前需求对切点信息（before, after, around 等）进行增强，其中 before 被增强为： `AspectJMethodBeforeAdvice`；after 被增强为： `AspectJAfterAdvice`；around 被增强为： `AspectJAfterThrowingAdvice` 增强器；增强器封装后，被代理接口调用时，会触发增强器中的方法。

接下来看看 `postProcessBeforeInstantiation` 中出现的 `createProxy` 方法，它是用来创建拦截器代理的。如图 4 所示，在 `createProxy` 方法中会接受拦截器 `specificInterceptors` 和目标源。方法体中创建代理工厂，接下来就是通过 `buildAdvisors` 方法生成 `advisors` 也就是增强的方法，对其进行遍历并且加入到代理工厂中。然后设置要代理的类，通过 `proxyFactory` 中的 `getProxy` 方法获取代理类的信息。

```

protected Object createProxy(Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {
    //创建代理工厂
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.copyFrom(this);
    //决定对给定的bean是否应该使用targetClass而不是他的接口代理, 主要看xml中的配置:
    //检查proxyTargetClass设置以及preserveTargetClass属性
    if (!proxyFactory.isProxyTargetClass()) {
        if (shouldProxyTargetClass(beanClass, beanName)) {
            proxyFactory.setProxyTargetClass(true); //类CGLIB代理
        }
        else {
            //添加代理接口, 动态代理
            evaluateProxyInterfaces(beanClass, proxyFactory);
        }
    }
    Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
    for (Advisor advisor : advisors) {
        //加入增强器: AspectJMethodBeforeAdvice等
        proxyFactory.addAdvisor(advisor);
    }
    //设置要代理的类
    proxyFactory.setTargetSource(targetSource);
    customizeProxyFactory(proxyFactory);
    proxyFactory.setFrozen(this.freezeProxy);
    if (advisorsPreFiltered()) {
        proxyFactory.setPreFiltered(true);
    }
    return proxyFactory.getProxy(getProxyClassLoader());
}

```

图 4 createProxy

在 createProxy 方法的最后会调用 getProxy 方法, 如图 5 所示, getProxy 方法中会调用 createAopProxy().getProxy(classLoader)。在被调用的 getProxy 方法是实现 JDK 的动态代理, 这个在上周的课程中 JDK 代理的源码中提到过, 该方法对代理类进行实例化并且返回。

```

public Object getProxy(ClassLoader classLoader) {
    return createAopProxy().getProxy(classLoader);
}

@Override
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());
    }
    Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
    findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}

```

图 5 getProxy

当返回代理类之后会执行该代理类的 invoke 方法, 最后来看看 invoke 方法的源码。如图 6 所示, 这个方法在上周 JDK 动态代理的课中解释过, 主要是对拦截链进行循环增强, 也就是将拦截器一个个地串行执行, 从而达到对目标类方法的增强。

```

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    MethodInvocation invocation;
    Object oldProxy = null;
    boolean setProxyContext = false;
    TargetSource targetSource = this.advised.targetSource;
    Class<?> targetClass = null;
    Object target = null;
    try {
        if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) { ...
        }
        if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) { ...
        }
        if (!this.advised.opaque && method.getDeclaringClass().isInterface() && ...
        }
        Object retVal;
        if (this.advised.exposeProxy) { ...
        }
        target = targetSource.getTarget();
        if (target != null) { ...
        }
        // 获取方法的拦截器链，其实是把增强器封装成拦截器的形式串行调用的；
        List<Object> chain =
            this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
        if (chain.isEmpty()) {
            retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
        }
        else {
            invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
            retVal = invocation.proceed();
        }
        Class<?> returnType = method.getReturnType();
        if (retVal != null && retVal == target && returnType.isInstance(proxy) &&
            !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
            retVal = proxy;
        }
        else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
            throw new AopInvocationException(
                "Null return value from advice does not match primitive return type for: " + method);
        }
        return retVal;
    }
}

```

3. 总结

本节课将对@Aspect 注释的执行过程做了讲解，针对源码部分对其进行了解析注释解析、代理注册、增强器提取、代理获取以及执行等几个步骤。下节课会解决系统内部异常未处理导致返回错误页面的问题，下期见，拜拜。