# Spring高手之路21——深入剖析Spring AOP代理对象的创建

## 创建代理对象核心动作的三个步骤

> 本文将详细介绍创建代理对象的三个核心步骤。关于`AOP`的基本调试，可以参考前文介绍的调试代码（任何涉及`AOP`的代码均可，如前置通知），这里不再详细说明。

### 1. 判断 Bean 是否需要增强（源码分析+时序图说明）

本节源码基于 `spring-aop-5.3.16`。

在`Spring AOP`中，这一步骤主要通过检查目标`bean`是否实现了特定接口或已是代理对象来完成。关键的方法是在`AbstractAutoProxyCreator`类的`postProcessBeforeInstantiation`中实现，该方法是 `Spring AOP` 的核心，属于 `Spring Bean` 生命周期的一部分，特别是在后置处理器(`BeanPostProcessor`)机制中起重要作用。

**主要功能：**

* 拦截`Bean`的创建：在实际的 `Bean` 实例化之前拦截创建过程，这使得开发者可以在对象实际创建之前注入特定的行为或逻辑。
* 条件判断：基于特定条件（例如`Bean`的类型或注解）来确定是否需要对该 `Bean` 应用代理或其他增强，特定条件比如(切点表达式)
* 创建代理：如果条件满足，这个方法可以用来创建一个代理实例代替原来的`Bean`，以便在运行时应用如安全、事务、日志等横切点。

**作用流程：**

**1. 获取缓存键**：首先通过 `Bean` 的类和名称构造一个缓存键，用于后续的快速查找和决策。
**2. 初步检查**：检查缓存是否已经有该 `Bean` 的信息，检查`Bean`是否为基础设施类或是否标记为不应代理。如果缓存中未找到对应键且`Bean`需要代

理，将进入代理创建步骤。
**3. 决定是否创建代理**：如果 `Bean` 不在上述类别中，进一步检查是否存在自定义的 `TargetSource`（一个控制如何获取或创建被代理对象的组件）。如果存在，表示这个 `Bean` 需要被增强或代理。
**4. 代理创建**：如果需要，创建一个代理实例，这个代理会封装原始 `Bean`，加入额外的行为如拦截方法调用。
**5. 结果返回**：返回创建的代理实例或者在不需要代理的情况下返回 `null`。

来看看对应的源码：



源码提出来分析：

```
@Override
public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) {
    // 获取beanClass和beanName组合的缓存键
    Object cacheKey = getCacheKey(beanClass, beanName);

    // 检查beanName是否有效，并且当前bean是否已经有一个自定义的TargetSource
    if (!StringUtils.hasLength(beanName) || !this.targetSourcedBeans.contains(beanName)) {
        // 如果当前bean已经被处理过，直接返回null，不再处理
        if (this.advisedBeans.containsKey(cacheKey)) {
            return null;
        }
        // 判断当前类是否是基础设施类或者是否应该跳过代理
        if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
            // 标记为不需要代理，并返回null
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return null;
        }
    }

    // 获取自定义的TargetSource，如果存在，则创建代理
    TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
```

```
    if (targetSource != null) {
        // 如果beanName有效，将其加入到管理自定义TargetSource的集合中
        if (StringUtils.hasLength(beanName)) {
            this.targetSourcedBeans.add(beanName);
        }
        // 获取适用于当前bean的advice和advisor
        Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
        // 创建代理对象
        Object proxy = createProxy(beanClass, beanName,
specificInterceptors, targetSource);
        // 缓存代理对象的类型
        this.proxyTypes.put(cacheKey, proxy.getClass());
        return proxy;
    }

    return null;
}
```

源码分析说明：

* **获取缓存键**：通过 `beanClass` 和 `beanName` 组合生成缓存键。
* **检查 beanName 和 TargetSource**：确保 `beanName` 有效且 `bean` 未被处理过，若已处理则返回 `null`。
* **判断基础设施类或跳过代理**：检查 `bean` 是否为基础设施类或应跳过代理，若是则返回 `null`。
* **获取自定义 TargetSource 并创建代理**：如果存在自定义 `TargetSource`，则创建代理对象并缓存其类型。

**这里为什么要获取自定义的 TargetSource？**

在`Spring AOP`中，创建代理对象时，`TargetSource`起着关键作用。它主要决定了如何获取或创建将被代理的目标对象。默认情况下，`Spring` 使用简单的目标源，即直接引用具体的 `Bean` 实例。但在某些情况下，开发者可能需要通过自定义`TargetSource`来改变目标对象的获取逻辑，以适应特定的增强需求。

在`AbstractAutoProxyCreator`类的`postProcessBeforeInstantiation`方法中，我们可以决定是否使用自定义的`TargetSource`。如果存在，这意味着`Bean`需要特殊的处理或增强。自定义的`TargetSource`还可以实现很多复杂的逻辑，比如：

* `池化目标对象`：为了提高性能，可以使用对象池来管理目标对象的实例。
* `延迟初始化`：只有在真正需要时才创建目标对象，可以减少资源使用和启动时间。
* `远程对象访问`：目标对象可能在远程服务器上，需要通过网络调用。
* `多租户支持`：基于当前用户或会话信息返回不同的目标对象实例。

用时序图表示如下：

**总体流程**
这个时序图描述了`Spring AOP`在创建代理对象时的核心过程。主要涉及的组件包括：调用者、`postProcessBeforeInstantiation`方法、缓存与条件检查、自定义`TargetSource`管理以及代理创建与返回。

**步骤解析**
**1. 调用开始**：

* 调用者调用`postProcessBeforeInstantiation`方法，开始代理对象的创建过程。
* 这个方法主要负责在`bean`实例化之前判断并创建其代理。

**2. 获取缓存键和初步条件检查：**

* 方法首先通过`beanClass`和`beanName`获取一个`cacheKey`。
* 接着检查`beanName`是否有效（非空且长度大于`0`）和当前`bean`是否已经有自定义的`TargetSource`。
* 如果`bean`已经存在于`advisedBeans`缓存中，或者属于基础设施类（如配置类等），或已指定为跳过代理，则不会进行进一步处理。

**3. 尝试获取自定义`TargetSource`：**

* 如果通过了初步的条件检查，将尝试获取一个针对当前`bean`的自定义`TargetSource`。
* 这一步是检查是否有特定于该`bean`的增强配置，如果有，则可以继续创建代理。

**4. 代理对象的创建：**

* 如果存在自定义的`TargetSource`，则使用相关的`advisors`（增强器）和这个`TargetSource`来创建一个代理对象。
* 这个代理对象将能够在运行时拦截对`bean`的调用，并应用定义的增强逻辑（如安全检查、事务管理等）。

**5. 返回结果：**

* 如果成功创建了代理对象，则返回这个对象给调用者。
* 如果没有自定义的`TargetSource`或者不需要创建代理，方法将返回`null`。

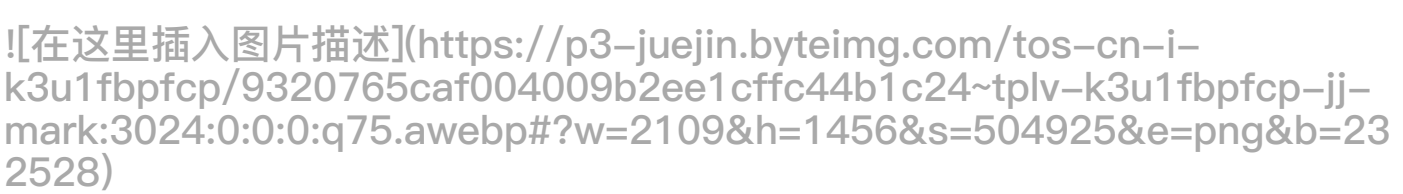
**条件判断**

* 缓存键不存在或`bean`需要代理：这个分支处理创建代理所需的条件检查和配置获取。
* 自定义`TargetSource`存在：只有当自定义的`TargetSource`存在时，才会尝试创建代理对象。
* 自定义`TargetSource`不存在或缓存键存在且`bean`不需要代理：这些情况将导致方法返回`null`，不进行代理的创建。

2. 匹配增强器 Advisors（源码分析+时序图说明）
---

增强器（或称为 "`advisors`"）定义何时以及如何增强目标对象。
源码中的`AbstractAdvisorAutoProxyCreator`类继承自`AbstractAutoProxyCreator`，添加处理Advisor的功能。特别是`findEligibleAdvisors`方法，它用于找出适用于特定`bean`的所有`advisors`。

来看看源码




提取关键代码分析

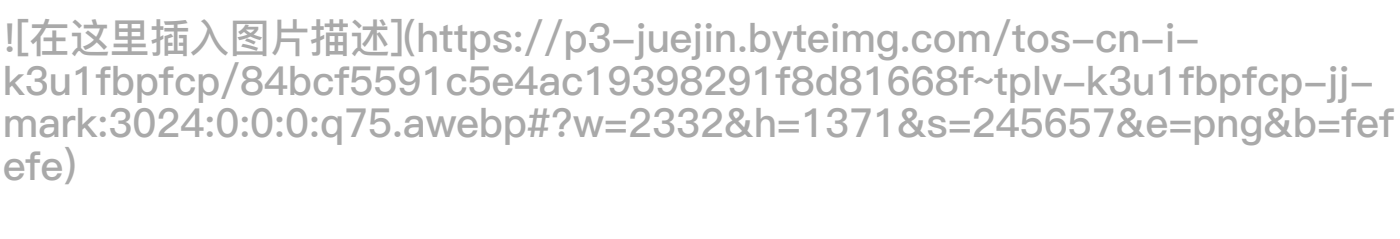
```
// 重写getAdvicesAndAdvisorsForBean方法，用于获取适用于特定bean的advisors和advices
@Override
@Nullable
protected Object[] getAdvicesAndAdvisorsForBean(
        Class<?> beanClass, String beanName, @Nullable TargetSource targetSource) {

    // 调用findEligibleAdvisors方法获取所有适用的advisors
    List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
    // 如果没有找到任何适用的advisor，则返回DO_NOT_PROXY（不进行代理）
    if (advisors.isEmpty()) {
        return DO_NOT_PROXY;
    }
    // 将找到的advisors转换为数组并返回
    return advisors.toArray();
}

/**
 * 查找所有适合自动代理的advisors。
 * @param beanClass 需要查找advisors的bean的类
 * @param beanName 当前被代理的bean的名称
 * @return 如果没有找到适用的pointcuts或interceptors，返回空列表，而不是null
 * @see #findCandidateAdvisors 查找候选的advisors
 * @see #sortAdvisors 对advisors进行排序
 * @see #extendAdvisors 扩展advisors
 */
protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanName) {
    // 查找所有候选的advisors
    List<Advisor> candidateAdvisors = findCandidateAdvisors();
    // 从候选advisors中找出可以应用于当前bean的advisors
    List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);
    // 可选操作：扩展advisors列表
    extendAdvisors(eligibleAdvisors);
    // 如果找到了适用的advisors，则对它们进行排序
    if (!eligibleAdvisors.isEmpty()) {
        eligibleAdvisors = sortAdvisors(eligibleAdvisors);
    }
    // 返回最终确定的适用于当前bean的advisors列表
    return eligibleAdvisors;
}
```

```

用时序图表示如下：




时序图详解

**1. 客户端请求Bean实例:**

* 客户端向`AbstractAutoProxyCreator`发出请求以获取`Bean`实例。这通常发生在`Spring`的应用上下文中，当一个`Bean`被请求时，`Spring`会检查这个`Bean`是否需要代理。

**2. 调用findEligibleAdvisors方法:**

* `AbstractAutoProxyCreator`调用`findEligibleAdvisors`方法，传入`beanClass`和`beanName`。这个方法负责找出所有可能适用于这个特定`Bean`的`Advisors`。

**3. 查找所有候选Advisors:**

* `findEligibleAdvisors`进一步调用`findCandidateAdvisors`，这个方法从`Spring`的应用上下文中检索所有配置的`Advisors`。

**4. 筛选适用于Bean的Advisors:**

* 返回的候选`Advisors`列表会传给`findAdvisorsThatCanApply`，这个方法根据当前`Bean`的类型和名称筛选出适用的`Advisors`。

**5. 对Advisors进行排序:**

* 适用的`Advisors`会传给`sortAdvisors`，以确保在代理中按正确的顺序应用
```

它们。

**6. 返回适用的Advisors列表:**

* `findEligibleAdvisors`将排序后的`Advisors`列表返回给`AbstractAutoProxyCreator`。

**7. 决策点 – 是否存在适用的Advisors:**

* 如果找到适用的`Advisors`，将继续创建`Bean`的代理；如果没有找到，直接返回原始的`Bean`实例。

**8. 创建Bean的代理:**

* 代理工厂根据返回的`Advisors`创建`Bean`的代理实例，并将`Advisors`注入到代理中。

**9. 客户端调用Bean的方法:**

* 客户端通过代理实例调用`Bean`的方法。如果`Bean`被代理，`Advisors`中定义的额外逻辑（例如，拦截、事务管理）会在调用实际`Bean`方法之前或之后执行。

**10. 返回方法执行结果:**

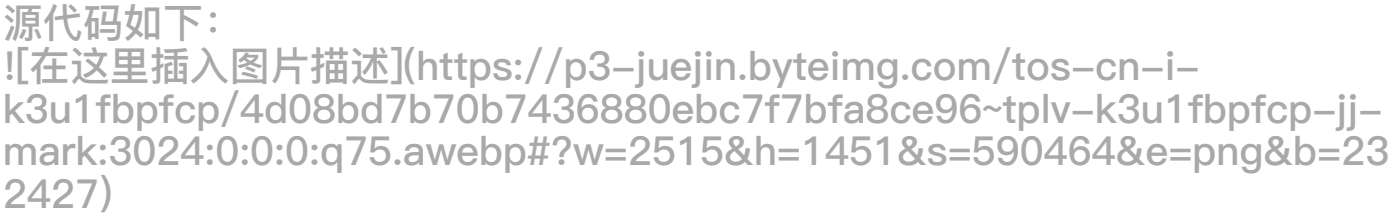
* 方法执行完成后，结果通过代理返回给客户端。

3. 创建代理对象（源码分析+时序图说明）
---

如果发现有合适的`advisors`，`Spring`将使用`AOP`代理工厂来创建代理对象。这部分的处理通常涉及到多种代理的创建策略，如`JDK`动态代理或`CGLIB`代理。

查看`AbstractAutoProxyCreator`中的`createProxy`方法。这个方法负责根据

给定的`bean`、其对应的`bean`名称、匹配到的`advisors`等信息来创建代理对象。

源代码如下：



提出代码分析：

```
/**
 * 创建给定bean的AOP代理。
 * @param beanClass bean的类
 * @param beanName bean的名称
 * @param specificInterceptors 针对这个bean的一组拦截器（可能为空，但不为null）
 * @param targetSource 为代理配置的目标源
 * @return bean的AOP代理
 * @see #buildAdvisors 构建advisor的方法
 */
protected Object createProxy(Class<?> beanClass, @Nullable String beanName,
        @Nullable Object[] specificInterceptors, TargetSource targetSource) {

    // 检查bean工厂是否为ConfigurableListableBeanFactory类型，如果是，则暴露bean的目标类
    if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory) this.beanFactory, beanName, beanClass);
    }

    // 创建ProxyFactory对象，并从当前实例复制配置
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.copyFrom(this);

    // 判断是否应该使用代理目标类，而不是只使用接口代理
    if (proxyFactory.isProxyTargetClass()) {
        // 如果beanClass是一个JDK代理类，处理引入通知场景
        if (Proxy.isProxyClass(beanClass)) {
            // 允许引入，不能仅设置接口为代理的接口
            for (Class<?> ifc : beanClass.getInterfaces()) {
```

```
                proxyFactory.addInterface(ifc);
            }
        }
    } else {
        // 如果没有强制代理目标类标志，则进行默认检查
        if (shouldProxyTargetClass(beanClass, beanName)) {
            // 如果决定代理目标类，则设置相应标志
            proxyFactory.setProxyTargetClass(true);
        } else {
            // 否则，评估并设置需要代理的接口
            evaluateProxyInterfaces(beanClass, proxyFactory);
        }
    }

    // 构建适用于此bean的advisors数组
    Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
    // 将advisors添加到代理工厂
    proxyFactory.addAdvisors(advisors);
    // 设置代理的目标源
    proxyFactory.setTargetSource(targetSource);
    // 根据需要自定义代理工厂
    customizeProxyFactory(proxyFactory);

    // 设置代理是否冻结，即之后是否允许更改代理的配置
    proxyFactory.setFrozen(this.freezeProxy);
    // 如果advisors已经预过滤，设置代理为预过滤状态
    if (advisorsPreFiltered()) {
        proxyFactory.setPreFiltered(true);
    }

    // 获取代理的类加载器
    ClassLoader classLoader = getProxyClassLoader();
    if (classLoader instanceof SmartClassLoader && classLoader !=
beanClass.getClassLoader()) {
        // 如果类加载器为SmartClassLoader且与bean的类加载器不同，使用
原始类加载器
        classLoader = ((SmartClassLoader)
classLoader).getOriginalClassLoader();
    }
    // 获取并返回代理对象
    return proxyFactory.getProxy(classLoader);
}

```

时序图如下：

mark:3024:0:0:0:q75.awebp#?w=1331&h=1566&s=176392&e=png&b=fefefe)

时序图详解：

**1. 客户端请求创建代理**

* 动作：客户端向`Spring`容器请求创建一个代理对象。这通常是因为客户端需要一个被`AOP`增强的`Bean`，比如添加了事务管理、性能监控或安全控制等。
* 背景：这通常发生在`Spring`应用启动时或者第一次请求`Bean`时，`Spring`容器会根据`Bean`的定义和`AOP`配置决定是否需要创建代理。

**2. Spring容器调用createProxy方法**

* 动作：`Spring`容器调用`createProxy`方法开始代理创建过程。
* 背景：`createProxy`是实现代理逻辑的核心方法，它负责集成所有相关配置并生成代理对象。

**3. 检查BeanFactory类型并暴露目标类**

* 动作：`createProxy`方法首先检查`Bean`工厂的类型，如果是`ConfigurableListableBeanFactory`，则调用`AutoProxyUtils.exposeTargetClass`来暴露`Bean`的目标类。
* 目的：这一步骤是为了在需要时能够获取`Bean`的实际类信息，尤其是当代理需要基于类而非接口创建时。

**4. 创建ProxyFactory实例**

* 动作：`createProxy`方法创建一个`ProxyFactory`实例。
* 目的：`ProxyFactory`是用于创建实际代理对象的工具，它支持对代理的各种配置，如代理的类型、拦截器链等。

**5. 判断并处理代理策略**

* 动作：根据是否使用代理目标类来决定代理方式，包括是否为`JDK`动态代理或`CGLIB`代理。
* 条件分支：

* + 如果目标类已是`JDK`代理类，将添加所有实现的接口到代理。
* + 如果不是`JDK`代理类，将根据`shouldProxyTargetClass`的结果决定是否代理目标类或仅代理特定接口。

**6. 构建Advisors并配置ProxyFactory**

* 动作：调用`buildAdvisors`方法构建适用于此`Bean`的`advisors`数组，然后将这些`advisors`添加到`ProxyFactory`。
* 目的：`Advisors`包含了增强的定义，这些增强定义了如何拦截方法调用及在调用前后执行特定的操作。

**7. 自定义ProxyFactory并创建代理对象**

* 动作：设置代理的目标源、自定义配置，冻结配置以确保在运行时不被修改，设置预过滤以优化匹配过程，最后通过`ProxyFactory`获取代理对象。
* 目的：完成所有代理配置后，最终生成代理对象，该对象将在运行时代表原始`Bean`，增加了指定的`AOP`功能。

**8. 返回代理对象**

* 动作：`createProxy`方法将代理对象返回给`Spring`容器，容器再返回给客户端。
* 结果：客户端接收到的`Bean`是一个被代理增强过的对象，具备了额外的`AOP`功能，如事务控制、安全检查等。

欢迎一键三连~

有问题请留言，大家一起探讨学习

----------Talk is cheap, show me the code----------