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

写在前面

在上一讲中，我们只是稍微分析了一下在配置类上添加@EnableAspectJAutoProxy注解之后，会向容器中注册了一个什么样的组件，因为咱们现在是要研究AOP 的原理，而研究AOP的原理就得从@EnableAspectJAutoProxy注解入手研究。

我讲到这里，大家一定会恍然大悟，其实，**要想知道AOP的原理，只需要搞清楚@EnableAspectJAutoProxy注解给容器中注册了什么组件，这个组件什么时候工作以及这个组件工作时候的功能是什么就行了，一旦把这个研究透了，那么AOP的原理我们就清楚了。**

经过上一讲的分析研究，我们知道在配置类上添加@EnableAspectJAutoProxy注解之后，会向容器中注册了这样一个AnnotationAwareAspectJAutoProxyCreator组件。当然了，我们也简单梳理了一下它的核心继承关系 ，如下所示。

```java
AnnotationAwareAspectJAutoProxyCreator
    ->AspectJAwareAdvisorAutoProxyCreator（父类）
        ->AbstractAdvisorAutoProxyCreator（父类）
            ->AbstractAutoProxyCreator（父类）
                implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware（两个接口）
```
AI写代码java运行

通过以上继承关系，我们也知道了，它最终会实现两个接口，分别是：

- BeanPostProcessor：后置处理器，即在bean初始化完成前后做些事情
- BeanFactoryAware：自动注入BeanFactory

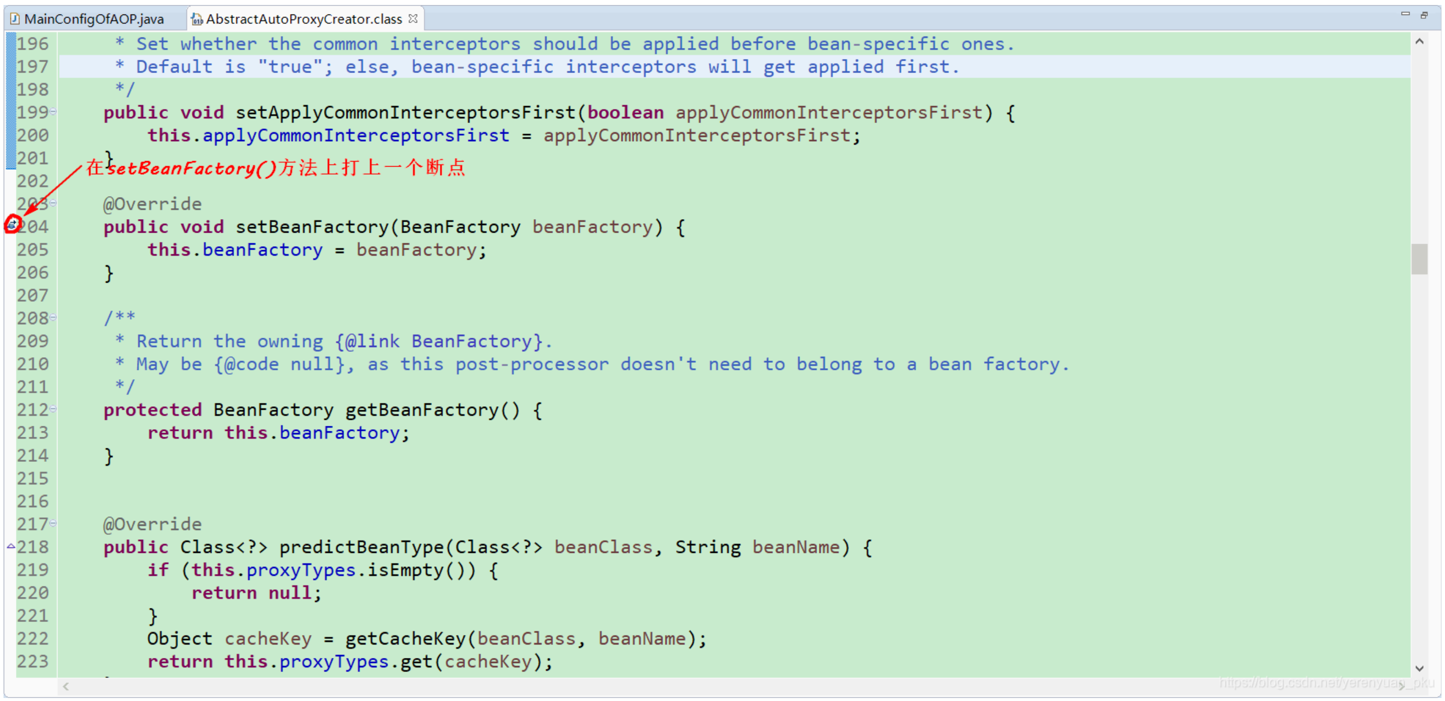
也就是说，AnnotationAwareAspectJAutoProxyCreator不仅是一个后置处理器，还是一个BeanFactoryAware接口的实现类。那么我们就来分析它作为后置处理器，到底做了哪些工作，以及它作为BeanFactoryAware接口的实现类，又做了哪些工作，只要把这个分析清楚，AOP的整个原理就差不多出来了。

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

接下来，我们就要为AnnotationAwareAspectJAutoProxyCreator这个组件里面和后置处理器以及Aware接口有关的方法都打上断点，看一下它们何时运行，以及都做了些什么事。

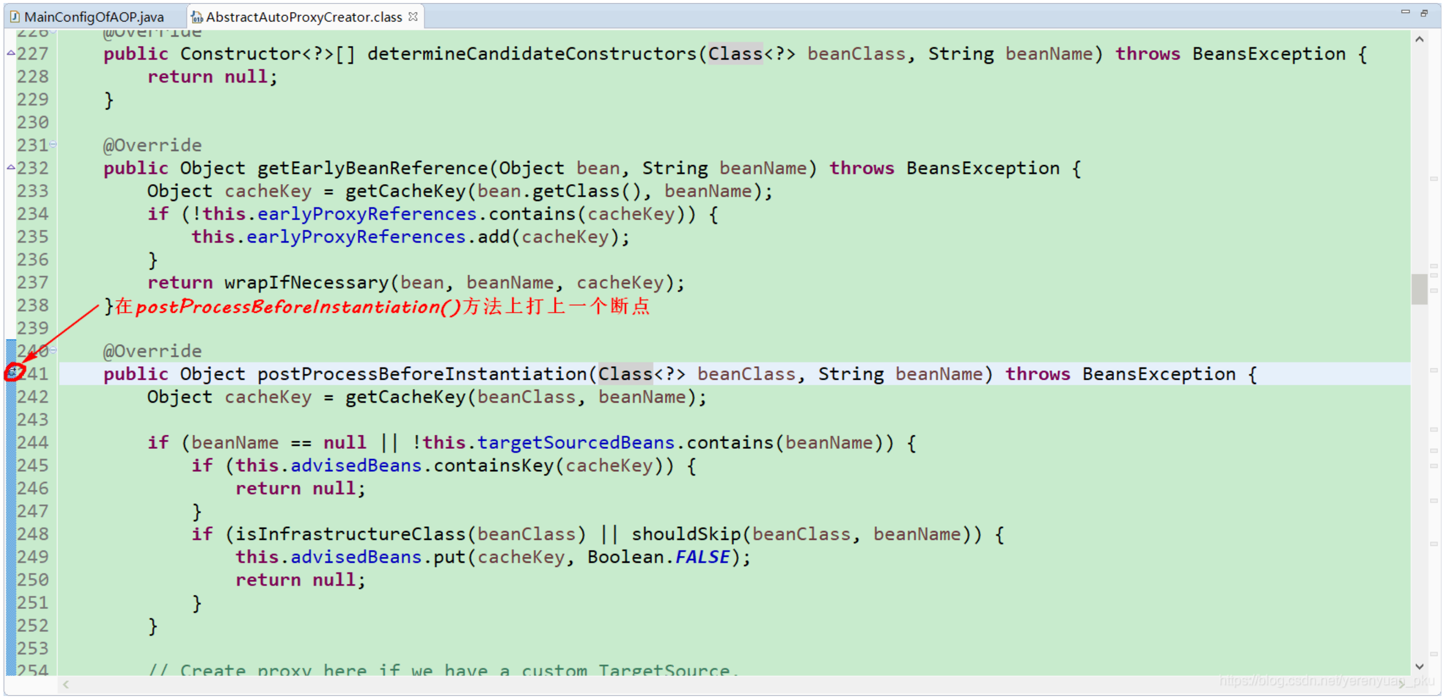
在打断点之前，我们还是得小心分析一下，因为AnnotationAwareAspectJAutoProxyCreator这个组件的继承关系还是蛮复杂的。由于是从AbstractAutoProxyCreator这个抽象类开始实现SmartInstantiationAwareBeanPostProcessor以及BeanFactoryAware这俩接口的，如果我们直接来AnnotationAwareAspectJAutoProxyCreator这个类里面找与Aware接口以及BeanPostProcessor接口有关的方法，是极有可能找不到的，所以我们还是得从它的最开始的父类（即AbstractAutoProxyCreator）开始分析。

我们找到该抽象类，并在里面查找与Aware接口以及BeanPostProcessor接口有关的方法，结果都是可以找到的。该抽象类中的setBeanFactory()方法就是与Aware接口有关的方法，因此我们将断点打在该方法上，如下图所示。


MainConfigOfAOP.java
AbstractAutoProxyCreator.class
在setBeanFactory()方法上打上一个断点

```java
	 * Set whether the common interceptors should be applied before bean-specific ones.
	 * Default is "true"; else, bean-specific interceptors will get applied first.
	 */
	public void setApplyCommonInterceptorsFirst(boolean applyCommonInterceptorsFirst) {
		this.applyCommonInterceptorsFirst = applyCommonInterceptorsFirst;
	}

	@Override
	public void setBeanFactory(BeanFactory beanFactory) {
		this.beanFactory = beanFactory;
	}

	/**
	 * Return the owning {@link BeanFactory}.
	 * May be {@code null}, as this post-processor doesn't need to belong to a bean factory.
	 */
	protected BeanFactory getBeanFactory() {
		return this.beanFactory;
	}


	@Override
	public Class<?> predictBeanType(Class<?> beanClass, String beanName) {
		if (this.proxyTypes.isEmpty()) {
			return null;
		}
		Object cacheKey = getCacheKey(beanClass, beanName);
		return this.proxyTypes.get(cacheKey);
```

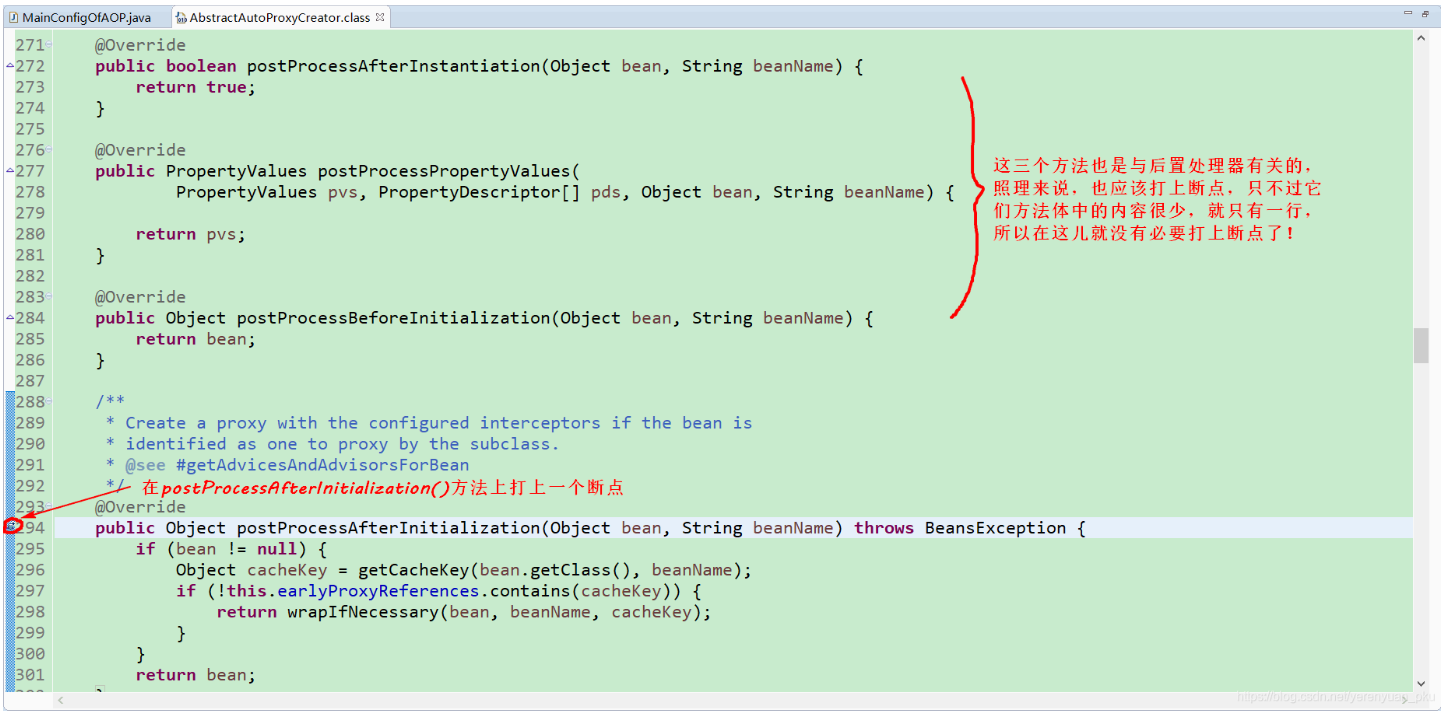
此外，我们还得找到该抽象类中与BeanPostProcessor接口有关的方法，即只要发现有与后置处理器相关的逻辑，就给所有与后置处理器有关的逻辑都打上断点。打的断点有两处，一处是在postProcessBeforeInstantiation()方法上，如下图所示。



```java
	@Override
	public Constructor<?>[] determineCandidateConstructors(Class<?> beanClass, String beanName) throws BeansException {
		return null;
	}

	@Override
	public Object getEarlyBeanReference(Object bean, String beanName) throws BeansException {
		Object cacheKey = getCacheKey(bean.getClass(), beanName);
		if (!this.earlyProxyReferences.contains(cacheKey)) {
			this.earlyProxyReferences.add(cacheKey);
		}
		return wrapIfNecessary(bean, beanName, cacheKey);
	}

	@Override
	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.
```

在*postProcessBeforeInstantiation()*方法上打上一个断点

一处是在postProcessAfterInitialization()方法上，如下图所示。



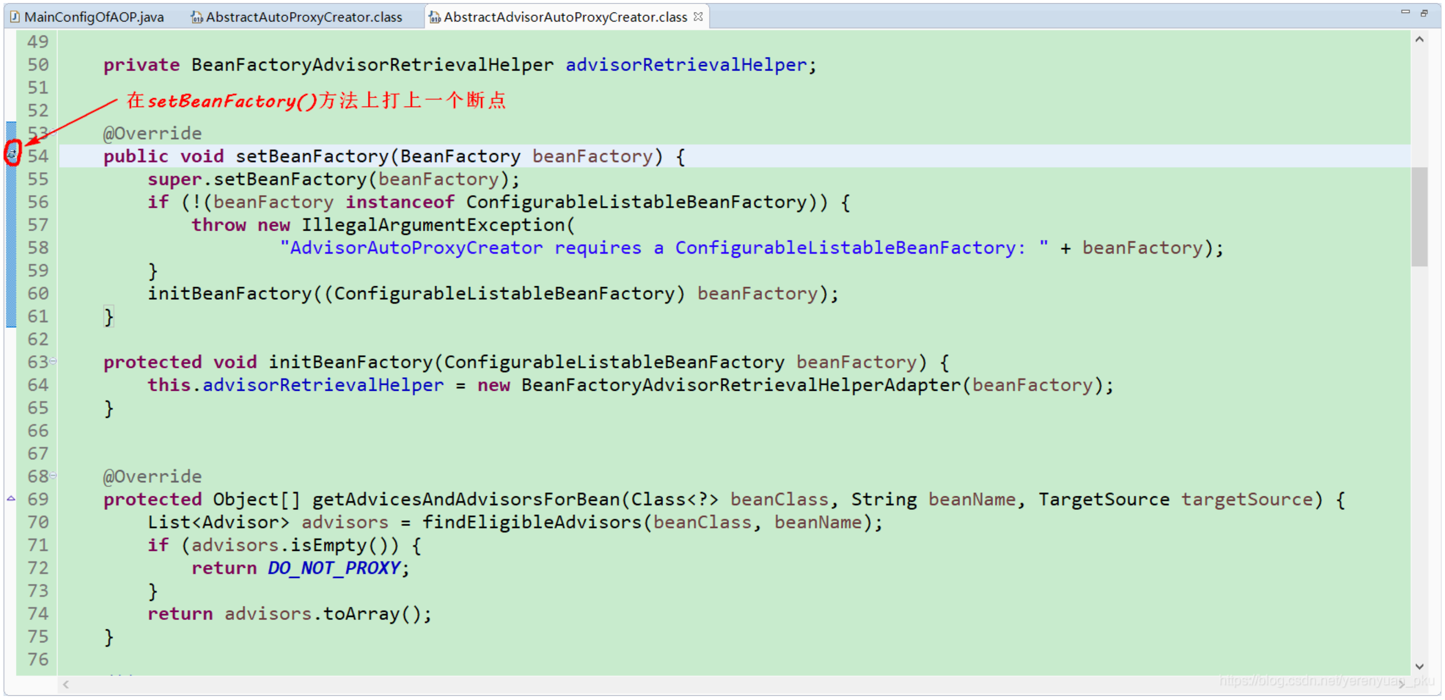
```java
	@Override
	public boolean postProcessAfterInstantiation(Object bean, String beanName) {
		return true;
	}

	@Override
	public PropertyValues postProcessPropertyValues(
			PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) {

		return pvs;
	}

	@Override
	public Object postProcessBeforeInitialization(Object bean, String beanName) {
		return bean;
	}

	/**
	 * Create a proxy with the configured interceptors if the bean is
	 * identified as one to proxy by the subclass.
	 * @see #getAdvicesAndAdvisorsForBean
	 */
	@Override
	public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
		if (bean != null) {
			Object cacheKey = getCacheKey(bean.getClass(), beanName);
			if (!this.earlyProxyReferences.contains(cacheKey)) {
				return wrapIfNecessary(bean, beanName, cacheKey);
			}
		}
		return bean;
```

这三个方法也是与后置处理器有关的，照理来说，也应该打上断点，只不过它们方法体中的内容很少，就只有一行，所以在这儿就没有必要打上断点了！

在*postProcessAfterInitialization()*方法上打上一个断点

接下来，我们再来看它的子类（即AbstractAdvisorAutoProxyCreator），从顶层开始一点一点往上分析。

在该抽象类中，我们只能找到一个与Aware接口有关的方法，即setBeanFactory()方法，虽然父类有setBeanFactory()方法，但是在这个子类里面已经把它重写了，因此最终调用的应该就是它。

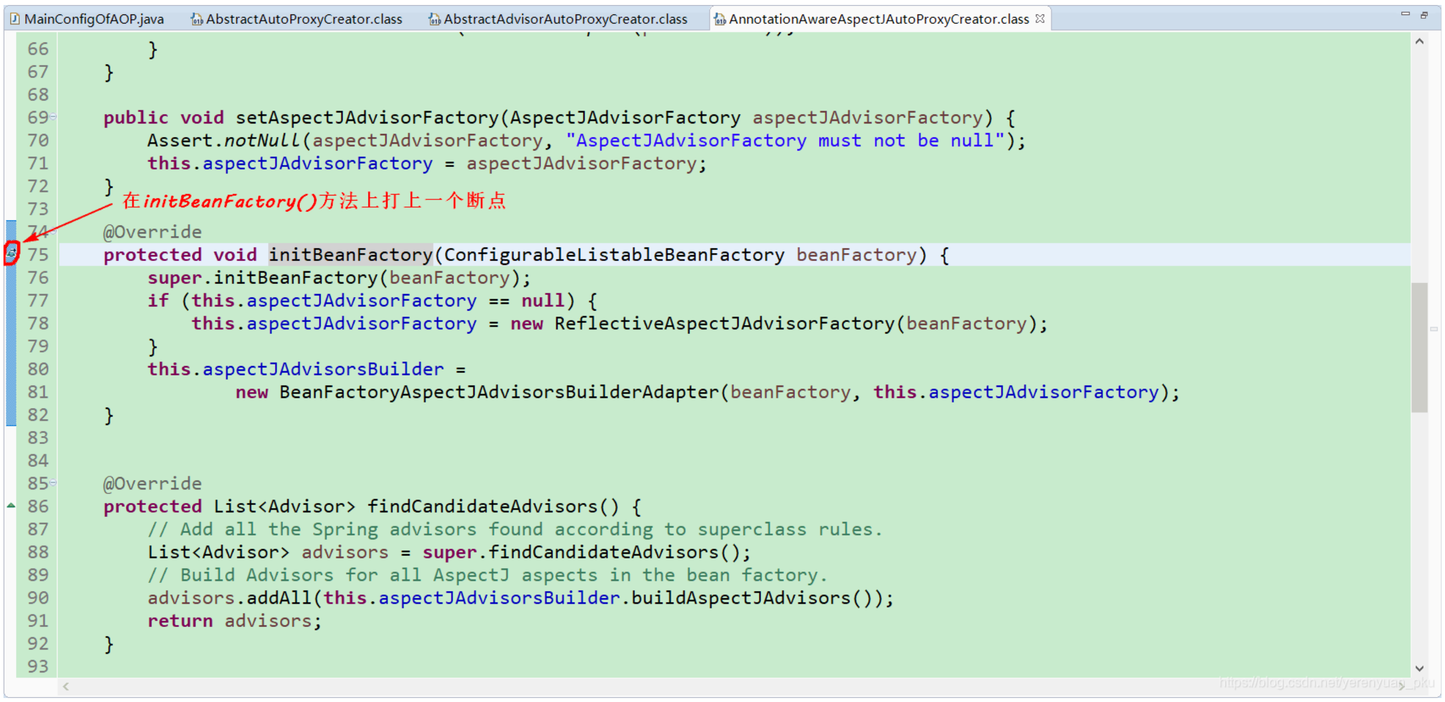

MainConfigOfAOP.java
AbstractAutoProxyCreator.class
AbstractAdvisorAutoProxyCreator.class
在setBeanFactory()方法上打上一个断点

```java
    private BeanFactoryAdvisorRetrievalHelper advisorRetrievalHelper;


    @Override
    public void setBeanFactory(BeanFactory beanFactory) {
        super.setBeanFactory(beanFactory);
        if (!(beanFactory instanceof ConfigurableListableBeanFactory)) {
            throw new IllegalArgumentException(
                    "AdvisorAutoProxyCreator requires a ConfigurableListableBeanFactory: " + beanFactory);
        }
        initBeanFactory((ConfigurableListableBeanFactory) beanFactory);
    }

    protected void initBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        this.advisorRetrievalHelper = new BeanFactoryAdvisorRetrievalHelperAdapter(beanFactory);
    }


    @Override
    protected Object[] getAdvicesAndAdvisorsForBean(Class<?> beanClass, String beanName, TargetSource targetSource) {
        List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
        if (advisors.isEmpty()) {
            return DO_NOT_PROXY;
        }
        return advisors.toArray();
    }
```

大家注意，在重写的时候，在setBeanFactory()方法里面会调用一个initBeanFactory()方法。除此之外，该抽象类中就没有跟后置处理器有关的方法了。

接下来，我们就应该来看AspectJAwareAdvisorAutoProxyCreator这个类了，但由于这个类里面没有跟BeanPostProcessor接口有关的方法，所以我们就不必看这个类了，略过。

接下来，我们就要来看最顶层的类了，即AnnotationAwareAspectJAutoProxyCreator。查看该类时，发现有这样一个initBeanFactory()方法，我们在该方法上打上一个断点就好，如下图所示。


MainConfigOfAOP.java
AbstractAutoProxyCreator.class
AbstractAdvisorAutoProxyCreator.class
AnnotationAwareAspectJAutoProxyCreator.class
在initBeanFactory()方法上打上一个断点

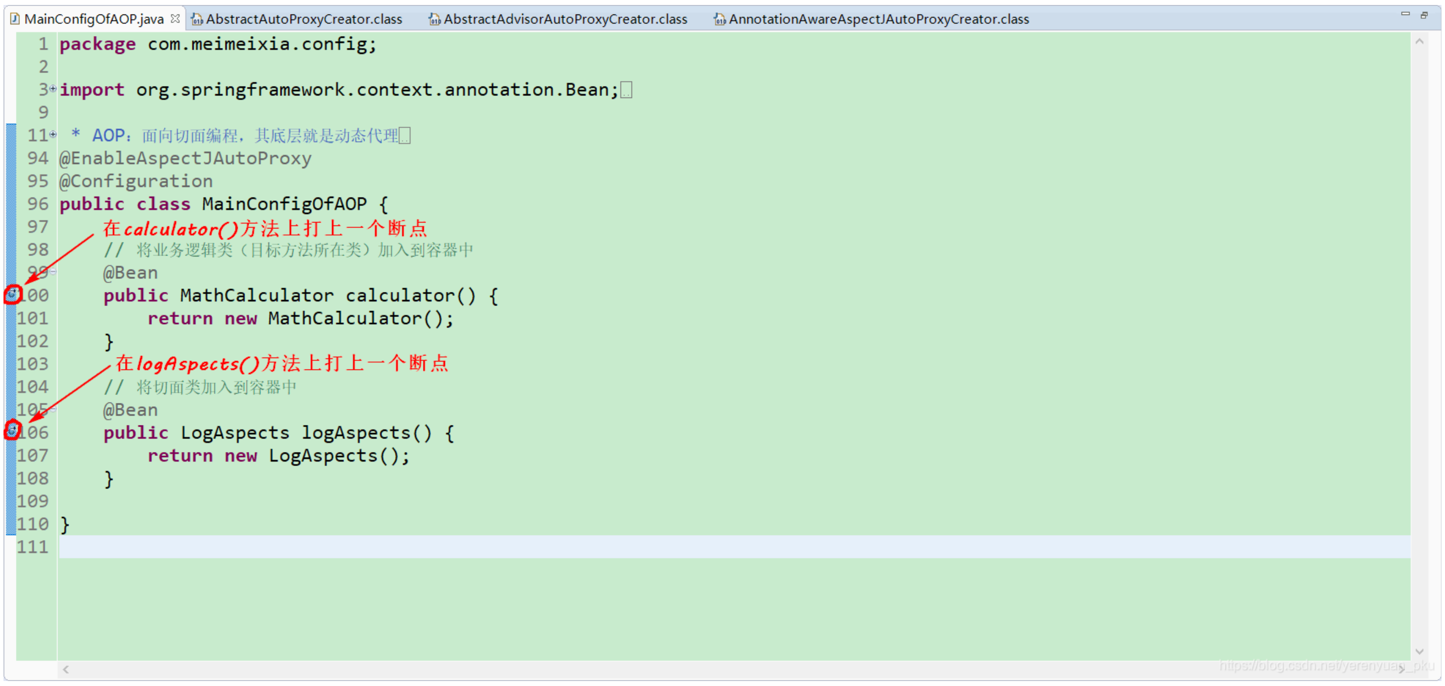
```java
        }
    }

    public void setAspectJAdvisorFactory(AspectJAdvisorFactory aspectJAdvisorFactory) {
        Assert.notNull(aspectJAdvisorFactory, "AspectJAdvisorFactory must not be null");
        this.aspectJAdvisorFactory = aspectJAdvisorFactory;
    }

    @Override
    protected void initBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        super.initBeanFactory(beanFactory);
        if (this.aspectJAdvisorFactory == null) {
            this.aspectJAdvisorFactory = new ReflectiveAspectJAdvisorFactory(beanFactory);
        }
        this.aspectJAdvisorsBuilder =
                new BeanFactoryAspectJAdvisorsBuilderAdapter(beanFactory, this.aspectJAdvisorFactory);
    }


    @Override
    protected List<Advisor> findCandidateAdvisors() {
        // Add all the Spring advisors found according to superclass rules.
        List<Advisor> advisors = super.findCandidateAdvisors();
        // Build Advisors for all AspectJ aspects in the bean factory.
        advisors.addAll(this.aspectJAdvisorsBuilder.buildAspectJAdvisors());
        return advisors;
    }
```

为什么在该类里面会有这个方法呢？因为我们在它的父类里面会调用setBeanFactory()方法，而在该方法里面又会调用initBeanFactory()方法，虽然父类里面有写，但是又被它的子类给重写了，所以说相当于父类中的setBeanFactory()方法还是得调用它。

那在该类中还有没有跟后置处理器有关的方法呢？没有了。

综上，我们通过简单的人工分析，为这个AnnotationAwareAspectJAutoProxyCreator类中有关后置处理器以及自动装配BeanFactoryAware接口的这些方法都打上了一些断点，接下来，我们就要来进行debug调试分析了。

不过在这之前，我们还得为MainConfigOfAOP配置类中的如下两个方法打上断点。


MainConfigOfAOP.java
AbstractAutoProxyCreator.class
AbstractAdvisorAutoProxyCreator.class
AnnotationAwareAspectJAutoProxyCreator.class
在calculator()方法上打上一个断点
在logAspects()方法上打上一个断点

```java
package com.meimeixia.config;

import org.springframework.context.annotation.Bean;

 * AOP：面向切面编程，其底层就是动态代理
@EnableAspectJAutoProxy
@Configuration
public class MainConfigOfAOP {

    // 将业务逻辑类（目标方法所在类）加入到容器中
    @Bean
    public MathCalculator calculator() {
        return new MathCalculator();
    }

    // 将切面类加入到容器中
    @Bean
    public LogAspects logAspects() {
        return new LogAspects();
    }

}
```

然后，我们就可以正式以debug模式来运行IOCTest_AOP测试类了，顺便分析一下整个流程。