Spring注解驱动开发第17讲——BeanPostProcessor在Spring底层是如何使用的？看完这篇我懂了！！

写在前面

在上一讲中，我们详细的介绍了BeanPostProcessor的执行流程。那么，BeanPostProcessor在Spring底层是如何使用的呢？今天，我们就一起来探讨下Spring的源码，一探BeanPostProcessor在Spring底层的使用情况。

BeanPostProcessor接口

我们先来看下BeanPostProcessor接口的源码，如下所示。

BeanPostProcessor.class

```
 * Copyright 2002-2015 the original author or authors.

package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;

 * Factory hook that allows for custom modification of new bean instances,
public interface BeanPostProcessor {

    * Apply this BeanPostProcessor to the given new bean instance <i>before</i> any bean
    Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;

    * Apply this BeanPostProcessor to the given new bean instance <i>after</i> any bean
    Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;

}
```


可以看到，在BeanPostProcessor接口中，提供了两个方法：postProcessBeforeInitialization()方法和postProcessAfterInitialization()方法。postProcessBeforeInitialization()方法会在bean 初始化 之前调用，postProcessAfterInitialization()方法会在bean初始化之后调用。接下来，我们就来分析下BeanPostProcessor接口在Spring中的实现。

注意：这里，我列举几个BeanPostProcessor接口在Spring中的实现类，来让大家更加清晰的理解BeanPostProcessor接口在Spring底层的应用。

ApplicationContextAwareProcessor类

org.springframework.context.support.ApplicationContextAwareProcessor是BeanPostProcessor接口的一个实现类，这个类的作用是可以向组件中注入 IOC容器 ，大致的源码如下所示。

```java
 * Copyright 2002-2016 the original author or authors.

package org.springframework.context.support;

import java.security.AccessControlContext;
import java.security.AccessController;
import java.security.PrivilegedAction;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.Aware;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.factory.config.EmbeddedValueResolver;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.EmbeddedValueResolverAware;
import org.springframework.context.EnvironmentAware;
import org.springframework.context.MessageSourceAware;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.util.StringValueResolver;

 * {@link org.springframework.beans.factory.config.BeanPostProcessor}
class ApplicationContextAwareProcessor implements BeanPostProcessor {

    private final ConfigurableApplicationContext applicationContext;

    private final StringValueResolver embeddedValueResolver;


    /**
     * Create a new ApplicationContextAwareProcessor for the given context.
     */
    public ApplicationContextAwareProcessor(ConfigurableApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
        this.embeddedValueResolver = new EmbeddedValueResolver(applicationContext.getBeanFactory());
    }


    @Override
    public Object postProcessBeforeInitialization(final Object bean, String beanName) throws BeansException {
        AccessControlContext acc = null;

        if (System.getSecurityManager() != null &&
                (bean instanceof EnvironmentAware || bean instanceof EmbeddedValueResolverAware ||
                        bean instanceof ResourceLoaderAware || bean instanceof ApplicationEventPublisherAware ||
                        bean instanceof MessageSourceAware || bean instanceof ApplicationContextAware)) {
            acc = this.applicationContext.getBeanFactory().getAccessControlContext();
        }

        if (acc != null) {
            AccessController.doPrivileged(new PrivilegedAction<Object>() {
                @Override
                public Object run() {
                    invokeAwareInterfaces(bean);
                    return null;
                }
            }, acc);
        }
        else {
            invokeAwareInterfaces(bean);
        }

        return bean;
    }

    private void invokeAwareInterfaces(Object bean) {
        if (bean instanceof Aware) {
            if (bean instanceof EnvironmentAware) {
                ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());
            }
            if (bean instanceof EmbeddedValueResolverAware) {
                ((EmbeddedValueResolverAware) bean).setEmbeddedValueResolver(this.embeddedValueResolver);
            }
            if (bean instanceof ResourceLoaderAware) {
                ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext);
            }
            if (bean instanceof ApplicationEventPublisherAware) {
                ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher(this.applicationContext);
            }
            if (bean instanceof MessageSourceAware) {
                ((MessageSourceAware) bean).setMessageSource(this.applicationContext);
            }
            if (bean instanceof ApplicationContextAware) {
                ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
            }
        }
    }
```

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

}
```


注意：我这里的Spring版本为4.3.12.RELEASE。

那具体如何使用ApplicationContextAwareProcessor类向组件中注入IOC容器呢？别急，我会用一个例子来说明下，相信小伙伴们看完后会有一种豁然开朗的感觉——哦，原来是它啊，我之前在项目中使用过的！

要想使用ApplicationContextAwareProcessor类向组件中注入IOC容器，我们就不得不提Spring中的另一个接口了，即ApplicationContextAware。如果需要向组件中注入IOC容器，那么可以让组件实现ApplicationContextAware接口。

例如，我们创建一个Dog类，使其实现ApplicationContextAware接口，此时，我们需要实现ApplicationContextAware接口中的setApplicationContext()方法，在setApplicationContext()方法中有一个ApplicationContext类型的参数，这个就是IOC容器对象，我们可以在Dog类中定义一个ApplicationContext类型的成员变量，然后在setApplicationContext()方法中为这个成员变量赋值，此时就可以在Dog类中的其他方法中使用ApplicationContext对象了，如下所示。

```java
package com.meimeixia.bean;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

/**
 * ApplicationContextAwareProcessor这个类的作用是可以帮我们在组件里面注入IOC容器，
 * 怎么注入呢？我们想要IOC容器的话，比如我们这个Dog组件，只需要实现ApplicationContextAware接口就行
 *
 * @author liayun
 *
 */
@Component
public class Dog implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Dog() {
        System.out.println("dog constructor...");
    }

    // 在对象创建完成并且属性赋值完成之后调用
    @PostConstruct
    public void init() {
        System.out.println("dog...@PostConstruct...");
    }

    // 在容器销毁（移除）对象之前调用
    @PreDestroy
    public void destory() {
        System.out.println("dog...@PreDestroy...");
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { // 在这儿打个断点调试一下
        // TODO Auto-generated method stub
        this.applicationContext = applicationContext;
    }

}
```

AI写代码java运行

看到这里，相信不少小伙伴们都有一种很熟悉的感觉，没错，我之前也在项目中使用过！是的，这就是BeanPostProcessor在Spring底层的一种使用场景。至于上面的案例代码为何会在setApplicationContext()方法中获取到ApplicationContext对象，这就是ApplicationContextAwareProcessor类的功劳了！

接下来，我们就深入分析下ApplicationContextAwareProcessor类。

我们先来看下ApplicationContextAwareProcessor类中对于postProcessBeforeInitialization()方法的实现，如下所示。

BeanPostProcessor.class | ApplicationContextAwareProcessor.class | Dog.java

```java
    @Override
    public Object postProcessBeforeInitialization(final Object bean, String beanName) throws BeansException {
        AccessControlContext acc = null;

        if (System.getSecurityManager() != null &&
                (bean instanceof EnvironmentAware || bean instanceof EmbeddedValueResolverAware ||
                        bean instanceof ResourceLoaderAware || bean instanceof ApplicationEventPublisherAware ||
                        bean instanceof MessageSourceAware || bean instanceof ApplicationContextAware)) {
            acc = this.applicationContext.getBeanFactory().getAccessControlContext();
        }

        if (acc != null) {
            AccessController.doPrivileged(new PrivilegedAction<Object>() {
                @Override
                public Object run() {
                    invokeAwareInterfaces(bean);
                    return null;
                }
            }, acc);
        }
        else {
            invokeAwareInterfaces(bean);
        }

        return bean;
    }

    private void invokeAwareInterfaces(Object bean) {
        if (bean instanceof Aware) {
```

在bean初始化之前，首先对当前bean的类型进行判断，如果当前bean的类型不是EnvironmentAware，不是EmbeddedValueResolverAware，不是ResourceLoaderAware，不是ApplicationEventPublisherAware，不是MessageSourceAware，也不是ApplicationContextAware，那么直接返回bean。如果是上面类型中的一种类型，那么最终会调用invokeAwareInterfaces()方法，并将bean传递给该方法。

invokeAwareInterfaces()方法又是个什么鬼呢？我们继续看invokeAwareInterfaces()方法的源码，如下所示。

BeanPostProcessor.class | ApplicationContextAwareProcessor.class | Dog.java

```java

    private void invokeAwareInterfaces(Object bean) {
        if (bean instanceof Aware) {
            if (bean instanceof EnvironmentAware) {
                ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());
            }
            if (bean instanceof EmbeddedValueResolverAware) {
                ((EmbeddedValueResolverAware) bean).setEmbeddedValueResolver(this.embeddedValueResolver);
            }
            if (bean instanceof ResourceLoaderAware) {
                ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext);
            }
            if (bean instanceof ApplicationEventPublisherAware) {
                ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher(this.applicationContext);
            }
            if (bean instanceof MessageSourceAware) {
                ((MessageSourceAware) bean).setMessageSource(this.applicationContext);
            }
            if (bean instanceof ApplicationContextAware) {
                ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
            }
        }
    }

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

```

可以看到invokeAwareInterfaces()方法的源码比较简单，就是判断当前bean属于哪种接口类型，然后将bean强转为哪种接口类型的对象，接着调用接口中的方法，将相应的参数传递到接口的方法中。这里，我们在创建Dog类时，实现的是ApplicationContextAware接口，所以，在invokeAwareInterfaces()方法中，会执行如下的逻辑代码。

```java
if (bean instanceof ApplicationContextAware) {
    ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
}
```

AI写代码java运行

我们可以看到，此时会将this.applicationContext传递到ApplicationContextAware接口的setApplicationContext()方法中。所以，我们在Dog类的setApplicationContext()方法中就可以直接接收到ApplicationContext对象了。

当然了，我们也可以在Eclipse 中通过Debug的形式来看一下程序的执行流程，此时我们在Dog类的setApplicationContext()方法上设置一个断点，如下所示。

BeanPostProcessor.class ApplicationContextAwareProcessor.class Dog.java

```java
 *
 */
@Component
public class Dog implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Dog() {
        System.out.println("dog constructor...");
    }

    // 在对象创建完成并且属性赋值完成之后调用
    @PostConstruct
    public void init() { // 在这儿打个断点调试一下
        System.out.println("dog...@PostConstruct...");
    }

    // 在容器销毁（移除）对象之前调用
    @PreDestroy
    public void destory() {
        System.out.println("dog...@PreDestroy...");
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { // 在这儿打个断点调试一下
        // TODO Auto-generated method stub
        this.applicationContext = applicationContext;
    }

}
```

然后，我们以Debug的方式来运行IOCTest_LifeCycle类中的test01()方法，运行后的效果如下图所示。

Debug - spring-annotation-liayun/src/main/java/com/meimeixia/bean/Dog.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Servers

Thread [main] (Suspended (entry into method setApplicationContext in Dog))
owns: ConcurrentHashMap<K,V> (id=42)
owns: Object (id=43)
Dog.setApplicationContext(ApplicationContext) line: 42
ApplicationContextAwareProcessor.invokeAwareInterfaces(Object) line: 121
ApplicationContextAwareProcessor.postProcessBeforeInitialization(Object, String) line: 97
DefaultListableBeanFactory(AbstractAutowireCapableBeanFactory).applyBeanPostProcessorsBeforeInitialization(Object, String) line: 409
DefaultListableBeanFactory(AbstractAutowireCapableBeanFactory).initializeBean(String, Object, RootBeanDefinition) line: 1620
DefaultListableBeanFactory(AbstractAutowireCapableBeanFactory).doCreateBean(String, RootBeanDefinition, Object[]) line: 555
DefaultListableBeanFactory(AbstractAutowireCapableBeanFactory).createBean(String, RootBeanDefinition, Object[]) line: 483
AbstractBeanFactory$1.getObject() line: 306
DefaultListableBeanFactory(DefaultSingletonBeanRegistry).getSingleton(String, ObjectFactory<?>) line: 230

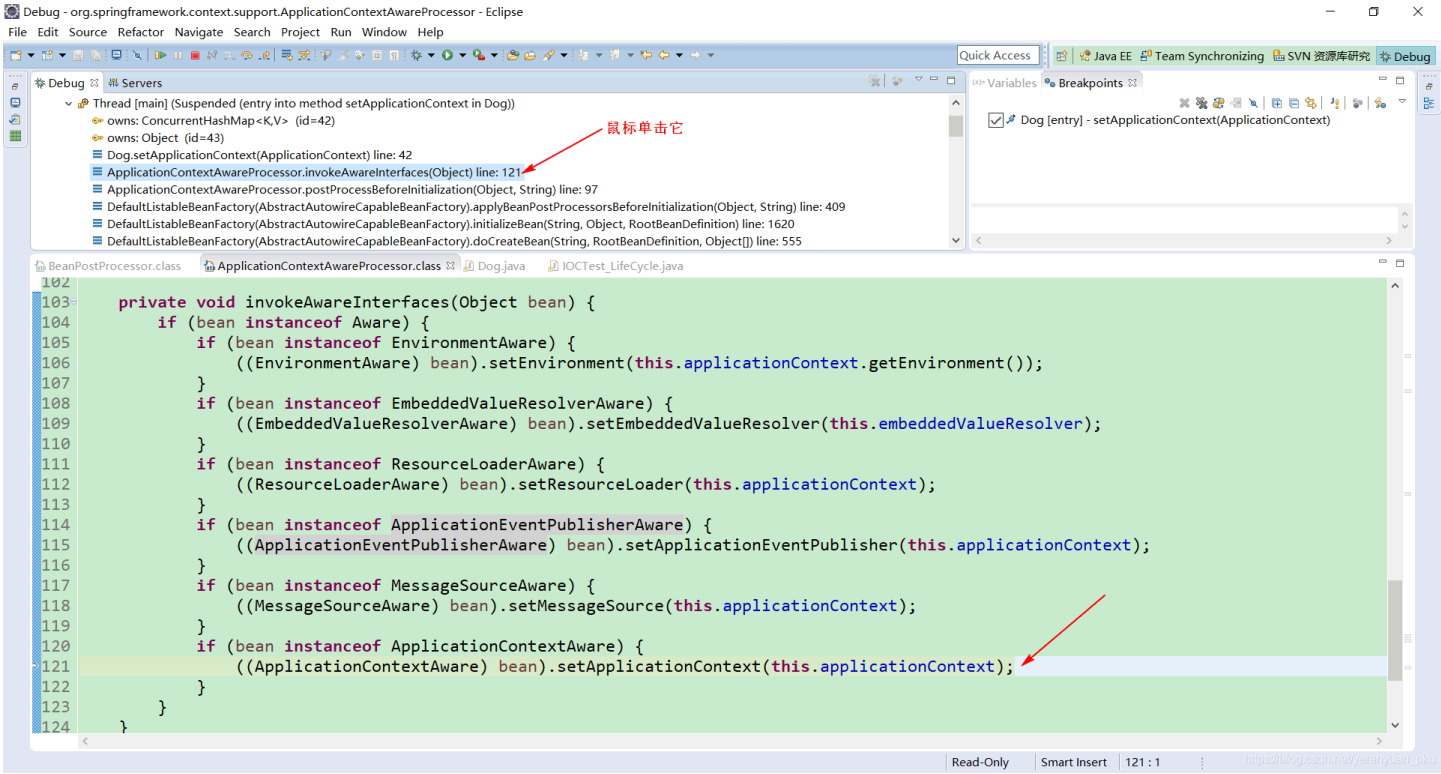
Variables Breakpoints
Dog [entry] - setApplicationContext(ApplicationContext)

BeanPostProcessor.class ApplicationContextAwareProcessor.class Dog.java IOCTest_LifeCycle.java

```java
    @PostConstruct
    public void init() { // 在这儿打个断点调试一下
        System.out.println("dog...@PostConstruct...");
    }

    // 在容器销毁（移除）对象之前调用
    @PreDestroy
    public void destory() {
        System.out.println("dog...@PreDestroy...");
    }

    @Override
    public void setApplicationContext(ApplicationCont
        // TODO Auto-generated method stub
        this.applicationContext = applicationContext
    }

}
```

applicationContext= AnnotationConfigApplicationContext (id=49)
active= AtomicBoolean (id=71)
applicationEventMulticaster= SimpleApplicationEventMulticaster (id=74)
applicationListeners= LinkedHashSet<E> (id=80)
beanFactory= DefaultListableBeanFactory (id=47)
org.springframework.context.annotation.AnnotationConfigApplicationContext@2a556333: s

在Eclipse的左上角可以看到方法的调用堆栈，通过对方法调用栈的分析，我们看到在执行Dog类中的setApplicationContext()方法之前，执行了ApplicationContextAwareProcessor类中的invokeAwareInterfaces方法，如下所示。



当我们点击方法调用栈中的invokeAwareInterfaces()方法时，方法的执行定位到如下这行代码处了。

```
((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
```

AI写代码java运行

这和我们之前分析的逻辑一致。

BeanValidationPostProcessor类

org.springframework.validation.beanvalidation.BeanValidationPostProcessor类主要是用来为bean进行校验操作的，当我们创建bean，并为bean赋值后，我们可以通过BeanValidationPostProcessor类为bean进行校验操作。BeanValidationPostProcessor类的源码如下所示。

```java
 * Copyright 2002-2012 the original author or authors.

package org.springframework.validation.beanvalidation;

import java.util.Iterator;
import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanInitializationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.config.BeanPostProcessor;

 * Simple {@link BeanPostProcessor} that checks JSR-303 constraint annotations
public class BeanValidationPostProcessor implements BeanPostProcessor, InitializingBean {

    private Validator validator;

    private boolean afterInitialization = false;


     * Set the JSR-303 Validator to delegate to for validating beans.
    public void setValidator(Validator validator) {
        this.validator = validator;
    }

     * Set the JSR-303 ValidatorFactory to delegate to for validating beans,
    public void setValidatorFactory(ValidatorFactory validatorFactory) {
        this.validator = validatorFactory.getValidator();
    }

     * Choose whether to perform validation after bean initialization
    public void setAfterInitialization(boolean afterInitialization) {
        this.afterInitialization = afterInitialization;
    }

    @Override
    public void afterPropertiesSet() {
        if (this.validator == null) {
            this.validator = Validation.buildDefaultValidatorFactory().getValidator();
        }
    }


    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        if (!this.afterInitialization) {
            doValidate(bean);
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (this.afterInitialization) {
            doValidate(bean);
        }
        return bean;
    }


     * Perform validation of the given bean.
    protected void doValidate(Object bean) {
        Set<ConstraintViolation<Object>> result = this.validator.validate(bean);
        if (!result.isEmpty()) {
            StringBuilder sb = new StringBuilder("Bean state is invalid: ");
            for (Iterator<ConstraintViolation<Object>> it = result.iterator(); it.hasNext();) {
                ConstraintViolation<Object> violation = it.next();
                sb.append(violation.getPropertyPath()).append(" - ").append(violation.getMessage());
                if (it.hasNext()) {
                    sb.append("; ");
                }
            }
            throw new BeanInitializationException(sb.toString());
        }
    }
```

```
}
```

这里，我们也来看看postProcessBeforeInitialization()方法和postProcessAfterInitialization()方法的实现，如下所示。

BeanPostProcessor.class | Dog.java | IOCTest_LifeCycle.java | BeanValidationPostProcessor.class

```
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        if (!this.afterInitialization) {
            doValidate(bean);
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (this.afterInitialization) {
            doValidate(bean);
        }
        return bean;
    }
```

可以看到，在postProcessBeforeInitialization()方法和postProcessAfterInitialization()方法中的主要逻辑都是调用doValidate()方法对bean进行校验，只不过在这两个方法中都会对afterInitialization这个boolean类型的成员变量进行判断，若afterInitialization的值为false，则在postProcessBeforeInitialization()方法中调用doValidate()方法对bean进行校验；若afterInitialization的值为true，则在postProcessAfterInitialization()方法中调用doValidate()方法对bean进行校验。

InitDestroyAnnotationBeanPostProcessor类

org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor类主要用来处理@PostConstruct注解和@PreDestroy注解。

例如，我们之前创建的Dog类中就使用了@PostConstruct注解和@PreDestroy注解，如下所示。

```java
package com.meimeixia.bean;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

/**
 * ApplicationContextAwareProcessor这个类的作用是可以帮我们在组件里面注入IOC容器，
 * 怎么注入呢？我们想要IOC容器的话，比如我们这个Dog组件，只需要实现ApplicationContextAware接口就行
 *
 * @author liayun
 *
 */
@Component
public class Dog implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Dog() {
        System.out.println("dog constructor...");
    }

    // 在对象创建完成并且属性赋值完成之后调用
    @PostConstruct
    public void init() {
        System.out.println("dog...@PostConstruct...");
    }

    // 在容器销毁（移除）对象之前调用
    @PreDestroy
    public void destory() {
        System.out.println("dog...@PreDestroy...");
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        // TODO Auto-generated method stub
        this.applicationContext = applicationContext;
    }

}
```

AI写代码java运行

那么，在Dog类中使用了@PostConstruct注解和@PreDestroy注解来标注方法，Spring怎么就知道什么时候执行@PostConstruct注解标注的方法，什么时候执行@PreDestroy注解标注的方法呢？这就要归功于InitDestroyAnnotationBeanPostProcessor类了。

接下来，我们也通过Debug的方式来跟进下代码的执行流程。首先，在Dog类的initt()方法上打上一个断点，如下所示。

BeanPostProcessor.class | Dog.java | IOCTest_LifeCycle.java

```
 *
 */
@Component
public class Dog implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Dog() {
        System.out.println("dog constructor...");
    }

    // 在对象创建完成并且属性赋值完成之后调用
    @PostConstruct
    public void init() { // 在这儿打个断点调试一下
        System.out.println("dog...@PostConstruct...");
    }

    // 在容器销毁（移除）对象之前调用
    @PreDestroy
    public void destory() {
        System.out.println("dog...@PreDestroy...");
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { // 在这儿打个断点调试一下
        // TODO Auto-generated method stub
        this.applicationContext = applicationContext;
    }

}
```

然后，我们以Debug的方式运行IOCTest_LifeCycle类中的test01()方法，效果如下所示。

Debug - spring-annotation-liayun/src/main/java/com/meimeixia/bean/Dog.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Debug | Servers

Thread [main] (Suspended (entry into method init in Dog))
- owns: ConcurrentHashMap<K,V> (id=44)
- owns: Object (id=45)
- Dog.init() line: 30
- NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
- NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
- DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
- Method.invoke(Object, Object...) line: 498
- InitDestroyAnnotationBeanPostProcessor$LifecycleElement.invoke(Object) line: 366

Variables | Breakpoints

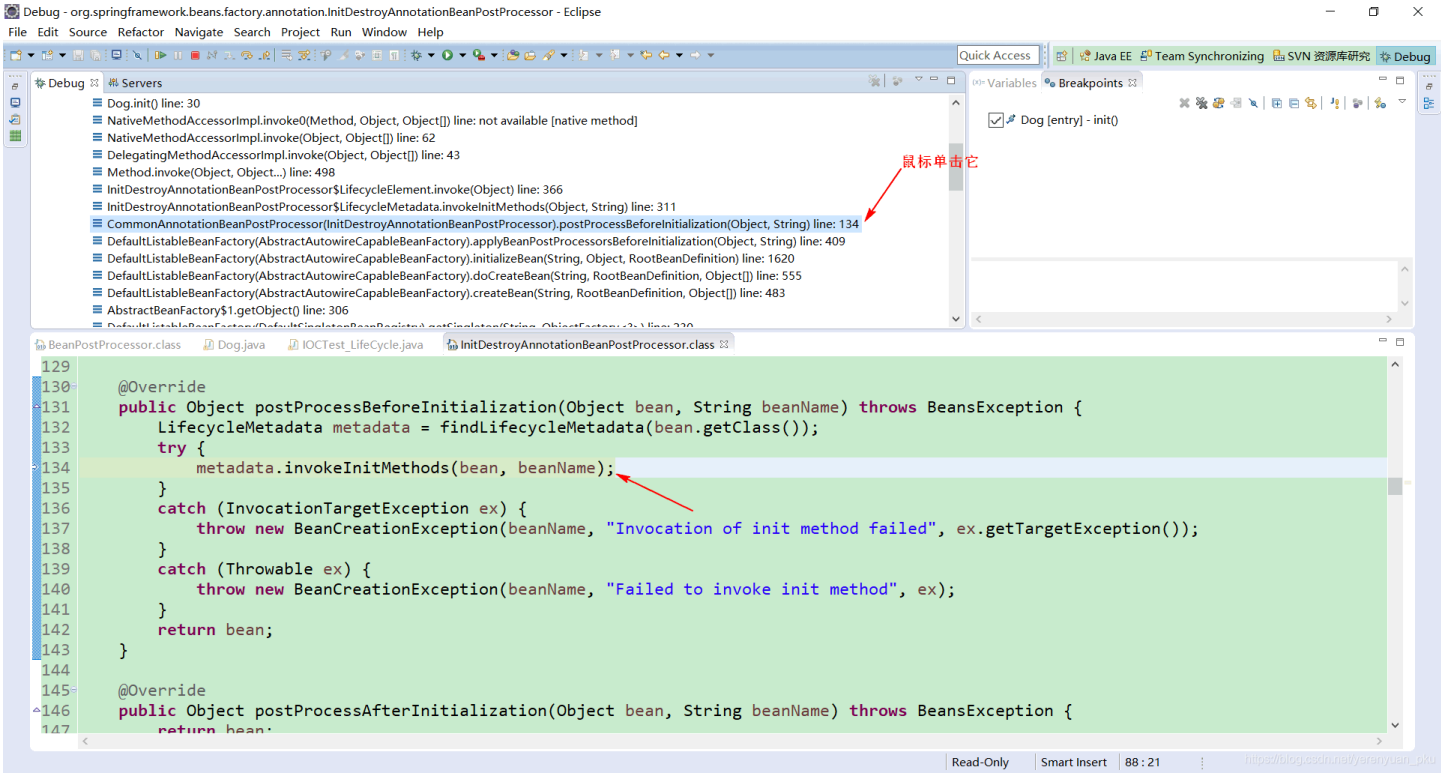
Dog [entry] - init()

BeanPostProcessor.class | Dog.java | IOCTest_LifeCycle.java

```
 *
 */
@Component
public class Dog implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Dog() {
        System.out.println("dog constructor...");
    }

    // 在对象创建完成并且属性赋值完成之后调用
    @PostConstruct
    public void init() { // 在这儿打个断点调试一下
        System.out.println("dog...@PostConstruct...");
    }

    // 在容器销毁（移除）对象之前调用
    @PreDestroy
    public void destory() {
        System.out.println("dog...@PreDestroy...");
    }
```

Writable | Smart Insert | 30 : 1

我们还是带着问题来分析，Spring怎么就能定位到使用@PostConstruct注解标注的方法呢？通过分析方法的调用栈，我们发现在进入使用@PostConstruct注解标注的方法之前，Spring调用了InitDestroyAnnotationBeanPostProcessor类的postProcessBeforeInitialization()方法，如下所示。



在InitDestroyAnnotationBeanPostProcessor类的postProcessBeforeInitialization()方法中，首先会找到bean中有关生命周期的注解，比如@PostConstruct注解等，找到这些注解之后，则将这些信息赋值给LifecycleMetadata类型的变量metadata，之后调用metadata的invokeInitMethods()方法，通过反射来调用标注了@PostConstruct注解的方法。这就是为什么标注了@PostConstruct注解的方法会被Spring执行的原因。

AutowiredAnnotationBeanPostProcessor类

org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor类主要是用于处理标注了@Autowired注解的变量或方法。

Spring为何能够自动处理标注了@Autowired注解的变量或方法，就交给小伙伴们自行分析了。大家可以写一个测试方法并通过方法调用堆栈来分析AutowiredAnnotationBeanPostProcessor类的源码，从而找到自己想要的答案。