Spring注解驱动开发第23讲——自定义组件中如何注入Spring底层的组件？看了这篇我才真正理解了原理！！

概述

如果我们现在自定义的组件中需要用到Spring底层的一些组件，比如ApplicationContext（IOC容器 ）、底层的BeanFactory等等，那么该怎么办呢？先说说自定义的组件中能不能用Spring底层的一些组件吧？既然都这样说了，那么肯定是能够的。

回到主题，自定义的组件要想使用Spring容器底层的一些组件，比如ApplicationContext（IOC容器）、底层的BeanFactory等等，那么只需要让自定义组件 实现XxxAware接口即可。此时，Spring在创建对象的时候，会调用XxxAware接口中定义的方法注入相关的组件。

XxxAware接口概览

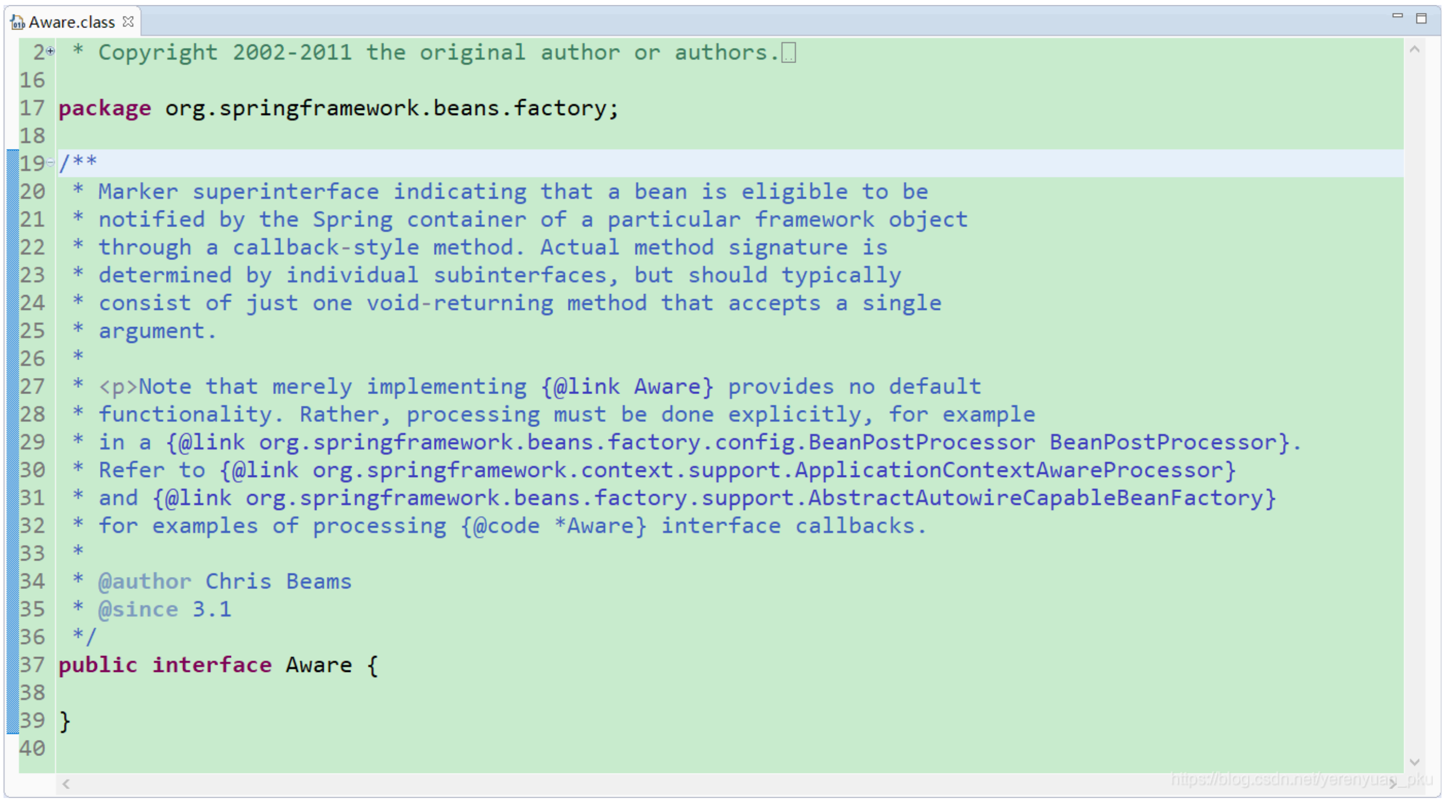
其实，我们之前使用过XxxAware接口，例如，我们之前创建的Dog类，就实现了ApplicationContextAware接口，Dog类的源码如下所示。

```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() { // 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;
    }

}
```

AI写代码java运行

从以上Dog类的源码中可以看出，实现ApplicationContextAware接口的话，需要实现setApplicationContext()方法。在IOC容器启动并创建Dog对象时，Spring会调用setApplicationContext()方法，并且会将ApplicationContext对象传入到setApplicationContext()方法中，我们只需要在Dog类中定义一个ApplicationContext类型的成员变量来接收setApplicationContext()方法中的参数，那么便可以在Dog类的其他方法中使用ApplicationContext对象了。

其实，在Spring中，类似于ApplicationContextAware接口的设计有很多，本质上，Spring中形如XxxAware这样的接口都继承了Aware接口，我们来看下Aware接口的源码，如下所示。

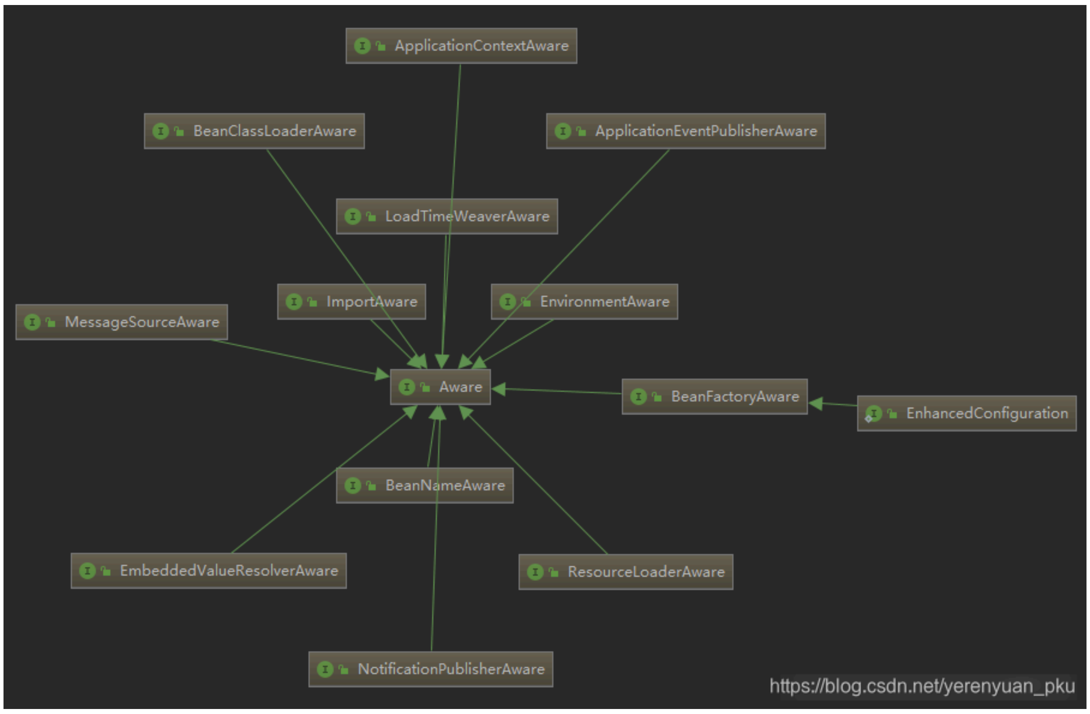


Aware.class

```
 * Copyright 2002-2011 the original author or authors.

package org.springframework.beans.factory;

/**
 * Marker superinterface indicating that a bean is eligible to be
 * notified by the Spring container of a particular framework object
 * through a callback-style method. Actual method signature is
 * determined by individual subinterfaces, but should typically
 * consist of just one void-returning method that accepts a single
 * argument.
 *
 * <p>Note that merely implementing {@link Aware} provides no default
 * functionality. Rather, processing must be done explicitly, for example
 * in a {@link org.springframework.beans.factory.config.BeanPostProcessor BeanPostProcessor}.
 * Refer to {@link org.springframework.context.support.ApplicationContextAwareProcessor}
 * and {@link org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory}
 * for examples of processing {@code *Aware} interface callbacks.
 *
 * @author Chris Beams
 * @since 3.1
 */
public interface Aware {

}

```

可以看到，Aware接口是Spring 3.1版本中引入的接口，在Aware接口中，并未定义任何方法。

接下来，我们看看都有哪些接口继承了Aware接口，如下所示。



哇！真的是有好多接口都实现了这个Aware接口。

XxxAware接口案例

接下来，我们就挑选几个常用的XxxAware接口来简单的说明一下。

ApplicationContextAware接口使用的比较多，我们先来说说这个接口，通过ApplicationContextAware接口我们可以获取到IOC容器。

首先，我们创建一个Red类，它得实现ApplicationContextAware接口，并在实现的setApplicationContext()方法中将ApplicationContext输出，如下所示。

```
package com.meimeixia.bean;

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

/**
```

```java
 * 以Red类为例来讲解ApplicationContextAware接口、BeanNameAware接口以及EmbeddedValueResolverAware接口
 * @author liayun
 *
 */
public class Red implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("传入的IOC：" + applicationContext);
        this.applicationContext = applicationContext;
    }

}
```

AI写代码java运行

其实，我们也可以让Red类同时实现几个XxxAware接口，例如，使Red类再实现一个BeanNameAware接口，我们可以通过BeanNameAware接口获取到当前bean在Spring容器中的名称，如下所示。

```java
package com.meimeixia.bean;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

/**
 * 以Red类为例来讲解ApplicationContextAware接口、BeanNameAware接口以及EmbeddedValueResolverAware接口
 * @author liayun
 *
 */
public class Red implements ApplicationContextAware, BeanNameAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("传入的IOC：" + applicationContext);
        this.applicationContext = applicationContext;
    }

    /**
     * 参数name：IOC容器创建当前对象时，为这个对象起的名字
     */
    @Override
    public void setBeanName(String name) {
        System.out.println("当前bean的名字：" + name);
    }

}
```

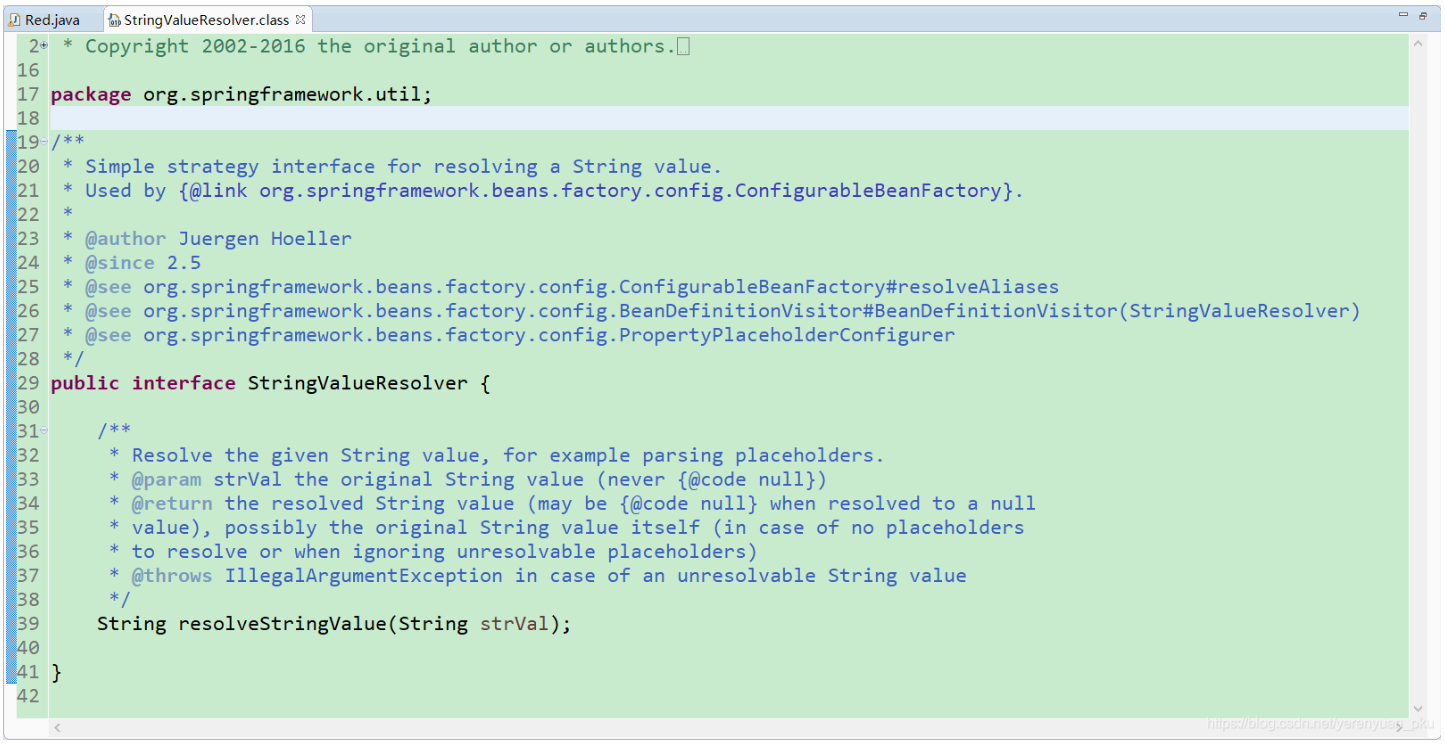
AI写代码java运行

当然了，我们可以再让Red类实现一个EmbeddedValueResolverAware接口，我们通过EmbeddedValueResolverAware接口能够获取到String值解析器，如下所示。

```java
package com.meimeixia.bean;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.EmbeddedValueResolverAware;
import org.springframework.util.StringValueResolver;

/**
 * 以Red类为例来讲解ApplicationContextAware接口、BeanNameAware接口以及EmbeddedValueResolverAware接口
 * @author liayun
 *
 */
public class Red implements ApplicationContextAware, BeanNameAware, EmbeddedValueResolverAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("传入的IOC: " + applicationContext);
        this.applicationContext = applicationContext;
    }

    /**
     * 参数name：IOC容器创建当前对象时，为这个对象起的名字
     */
    @Override
    public void setBeanName(String name) {
        System.out.println("当前bean的名字: " + name);
    }

    /**
     * 参数resolver：IOC容器启动时会自动地将这个String值的解析器传递过来给我们
     */
    @Override
    public void setEmbeddedValueResolver(StringValueResolver resolver) {
        String resolveStringValue = resolver.resolveStringValue("你好, ${os.name}, 我的年龄是#{20*18}");
        System.out.println("解析的字符串: " + resolveStringValue);
    }

}
```

AI写代码java运行

IOC容器启动时会自动地将String值的解析器（即StringValueResolver）传递过来给我们用，咱们可以用它来解析一些字符串，解析哪些字符串呢？比如包含 `#{}` 这样的字符串。我们可以看一下StringValueResolver类的源码，如下所示。



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

package org.springframework.util;

/**
 * Simple strategy interface for resolving a String value.
 * Used by {@link org.springframework.beans.factory.config.ConfigurableBeanFactory}.
 *
 * @author Juergen Hoeller
 * @since 2.5
 * @see org.springframework.beans.factory.config.ConfigurableBeanFactory#resolveAliases
 * @see org.springframework.beans.factory.config.BeanDefinitionVisitor#BeanDefinitionVisitor(StringValueResolver)
 * @see org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
 */
public interface StringValueResolver {

    /**
     * Resolve the given String value, for example parsing placeholders.
     * @param strVal the original String value (never {@code null})
     * @return the resolved String value (may be {@code null} when resolved to a null
     * value), possibly the original String value itself (in case of no placeholders
     * to resolve or when ignoring unresolvable placeholders)
     * @throws IllegalArgumentException in case of an unresolvable String value
     */
    String resolveStringValue(String strVal);

}
```

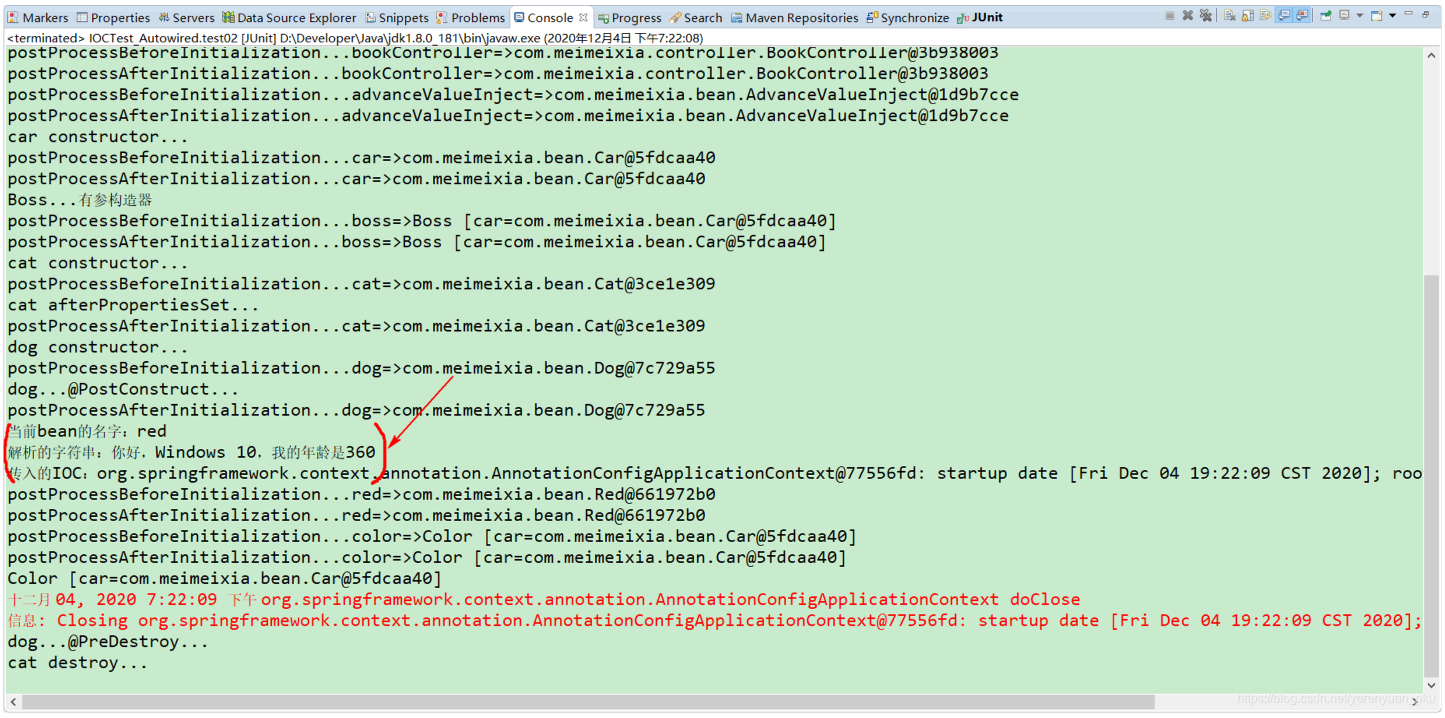
从描述中可以看出，它是用来帮我们解析那些String类型的值的，如果这个String类型的值里面有一些占位符，那么也会帮我们把这些占位符给解析出来，最后返回一个解析后的值。

接着，我们需要在Red类上标注@Component注解将该类添加到IOC容器中，如下所示。

```java
@Component
public class Red implements ApplicationContextAware, BeanNameAware, EmbeddedValueResolverAware {
```

AI写代码java运行

最后，运行IOCTest_Autowired类中的test02()方法，输出的结果信息如下所示。



```
<terminated> IOCTest_Autowired.test02 [JUnit] D:\Developer\Java\jdk1.8.0_181\bin\javaw.exe (2020年12月4日 下午7:22:08)
postProcessBeforeInitialization...bookController=>com.meimeixia.controller.BookController@3b938003
postProcessAfterInitialization...bookController=>com.meimeixia.controller.BookController@3b938003
postProcessBeforeInitialization...advanceValueInject=>com.meimeixia.bean.AdvanceValueInject@1d9b7cce
postProcessAfterInitialization...advanceValueInject=>com.meimeixia.bean.AdvanceValueInject@1d9b7cce
car constructor...
postProcessBeforeInitialization...car=>com.meimeixia.bean.Car@5fdcaa40
postProcessAfterInitialization...car=>com.meimeixia.bean.Car@5fdcaa40
Boss...有参构造器
postProcessBeforeInitialization...boss=>Boss [car=com.meimeixia.bean.Car@5fdcaa40]
postProcessAfterInitialization...boss=>Boss [car=com.meimeixia.bean.Car@5fdcaa40]
cat constructor...
postProcessBeforeInitialization...cat=>com.meimeixia.bean.Cat@3ce1e309
cat afterPropertiesSet...
postProcessAfterInitialization...cat=>com.meimeixia.bean.Cat@3ce1e309
dog constructor...
postProcessBeforeInitialization...dog=>com.meimeixia.bean.Dog@7c729a55
dog...@PostConstruct...
postProcessAfterInitialization...dog=>com.meimeixia.bean.Dog@7c729a55
当前bean的名字：red
解析的字符串：你好，Windows 10，我的年龄是360
传入的IOC：org.springframework.context.annotation.AnnotationConfigApplicationContext@77556fd: startup date [Fri Dec 04 19:22:09 CST 2020]; roo
postProcessBeforeInitialization...red=>com.meimeixia.bean.Red@661972b0
postProcessAfterInitialization...red=>com.meimeixia.bean.Red@661972b0
postProcessBeforeInitialization...color=>Color [car=com.meimeixia.bean.Car@5fdcaa40]
postProcessAfterInitialization...color=>Color [car=com.meimeixia.bean.Car@5fdcaa40]
Color [car=com.meimeixia.bean.Car@5fdcaa40]
十二月 04, 2020 7:22:09 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@77556fd: startup date [Fri Dec 04 19:22:09 CST 2020];
dog...@PreDestroy...
cat destroy...
```

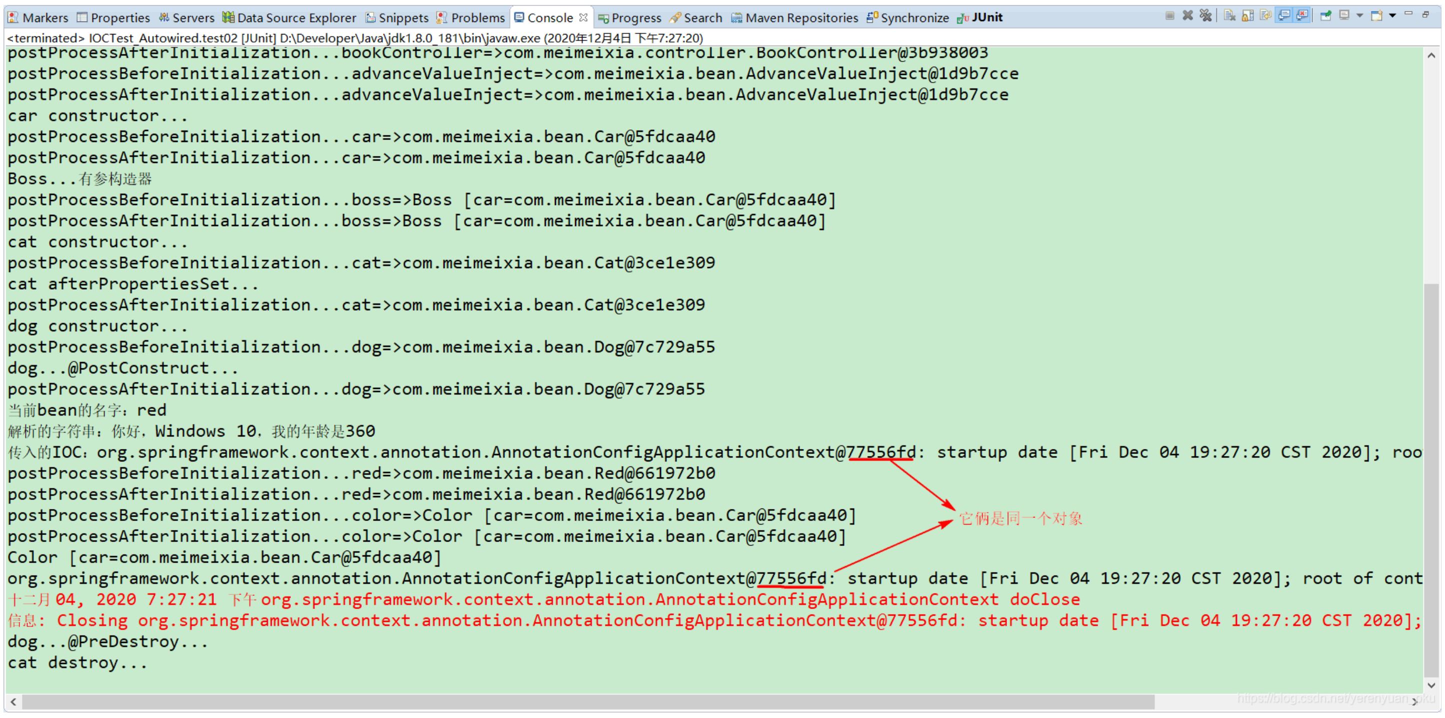
说明正确的输出了结果信息。

你可能会有一个疑问，在咱们自定义的组件中获取到的IOC容器和测试方法中获取到的IOC容器是不是同一个东东呢？带着这样一个疑问，你不妨试试运行一下以下test02()方法。

```java
@Test
public void test02() {
    AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfigOfAutowired.class);

    Color color = applicationContext.getBean(Color.class);
    System.out.println(color);

    System.out.println(applicationContext); // 测试用到的IOC容器

    applicationContext.close();
}
```

AI写代码java运行

相信你会看到输出了如下所示的结果信息。

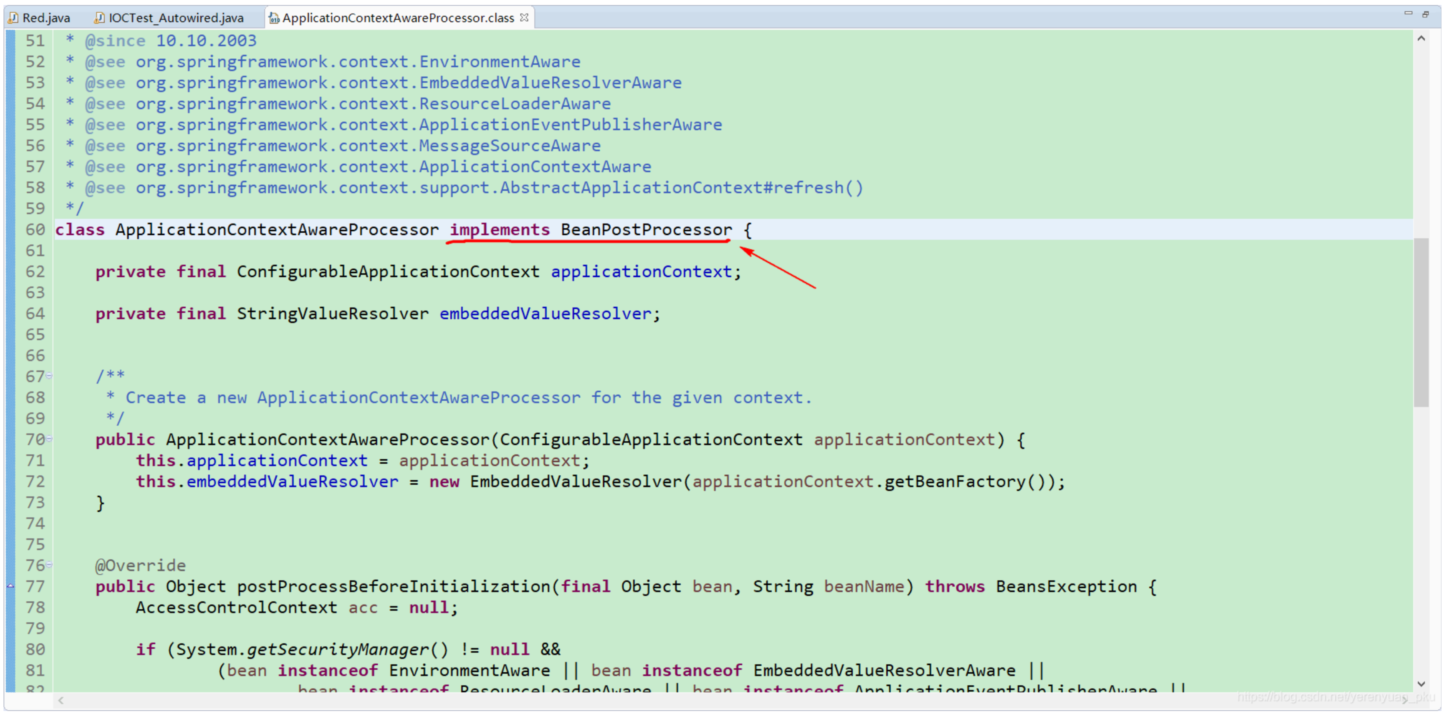


```
<terminated> IOCTest_Autowired.test02 [JUnit] D:\Developer\Java\jdk1.8.0_181\bin\javaw.exe (2020年12月4日 下午7:27:20)
postProcessAfterInitialization...bookController=>com.meimeixia.controller.BookController@3b938003
postProcessBeforeInitialization...advanceValueInject=>com.meimeixia.bean.AdvanceValueInject@1d9b7cce
postProcessAfterInitialization...advanceValueInject=>com.meimeixia.bean.AdvanceValueInject@1d9b7cce
car constructor...
postProcessBeforeInitialization...car=>com.meimeixia.bean.Car@5fdcaa40
postProcessAfterInitialization...car=>com.meimeixia.bean.Car@5fdcaa40
Boss...有参构造器
postProcessBeforeInitialization...boss=>Boss [car=com.meimeixia.bean.Car@5fdcaa40]
postProcessAfterInitialization...boss=>Boss [car=com.meimeixia.bean.Car@5fdcaa40]
cat constructor...
postProcessBeforeInitialization...cat=>com.meimeixia.bean.Cat@3ce1e309
cat afterPropertiesSet...
postProcessAfterInitialization...cat=>com.meimeixia.bean.Cat@3ce1e309
dog constructor...
postProcessBeforeInitialization...dog=>com.meimeixia.bean.Dog@7c729a55
dog...@PostConstruct...
postProcessAfterInitialization...dog=>com.meimeixia.bean.Dog@7c729a55
当前bean的名字：red
解析的字符串：你好，Windows 10，我的年龄是360
传入的IOC：org.springframework.context.annotation.AnnotationConfigApplicationContext@77556fd: startup date [Fri Dec 04 19:27:20 CST 2020]; roo
postProcessBeforeInitialization...red=>com.meimeixia.bean.Red@661972b0
postProcessAfterInitialization...red=>com.meimeixia.bean.Red@661972b0
postProcessBeforeInitialization...color=>Color [car=com.meimeixia.bean.Car@5fdcaa40]
postProcessAfterInitialization...color=>Color [car=com.meimeixia.bean.Car@5fdcaa40]
Color [car=com.meimeixia.bean.Car@5fdcaa40]
org.springframework.context.annotation.AnnotationConfigApplicationContext@77556fd: startup date [Fri Dec 04 19:27:20 CST 2020]; root of cont
十二月 04, 2020 7:27:21 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@77556fd: startup date [Fri Dec 04 19:27:20 CST 2020];
dog...@PreDestroy...
cat destroy...
```

这已然说明了在咱们自定义的组件中获取到的IOC容器和测试方法中获取到的IOC容器是同一个东东。

XxxAware原理

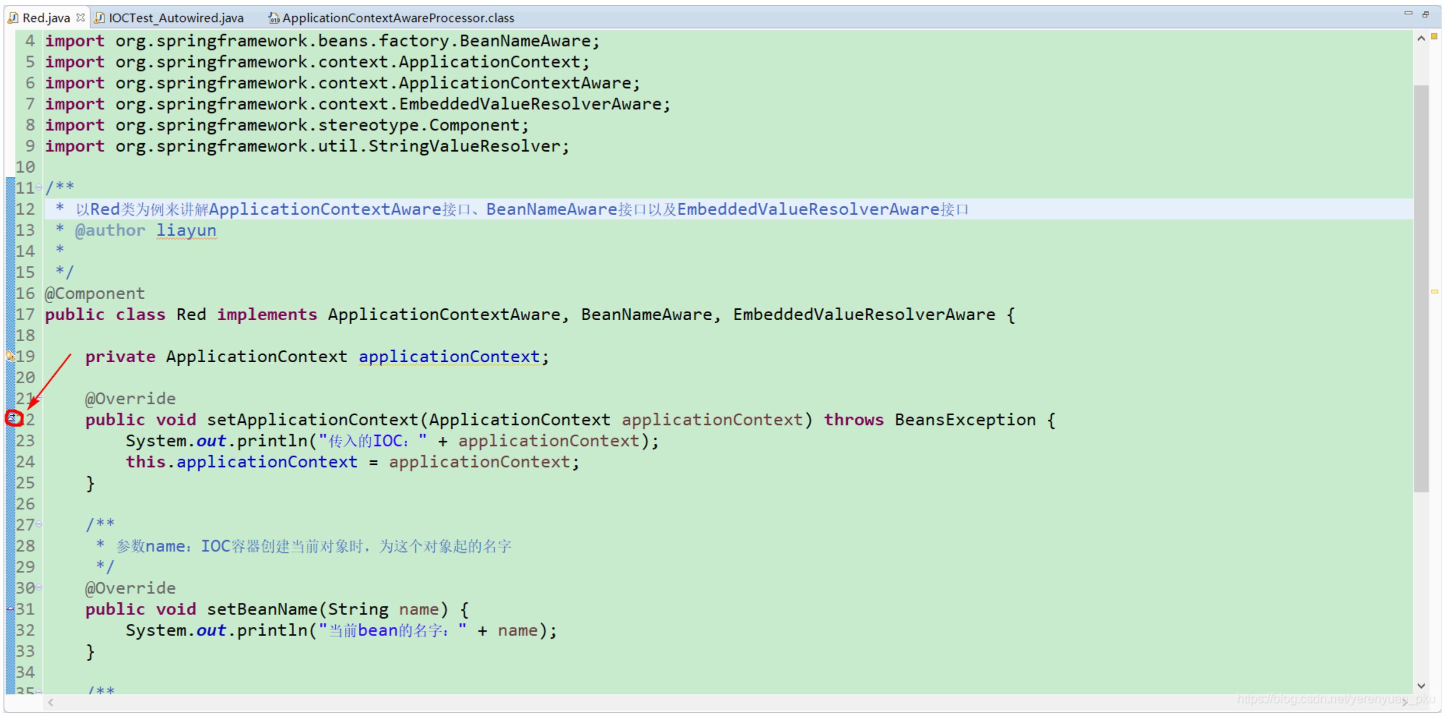
XxxAware接口的底层原理是由XxxAwareProcessor实现类实现的，也就是说每一个XxxAware接口都有它自己对应的XxxAwareProcessor实现类。 例如，我们这里以ApplicationContextAware接口为例，ApplicationContextAware接口的底层原理就是由ApplicationContextAwareProcessor类实现的。从ApplicationContextAwareProcessor类的源码可以看出，其实现了BeanPostProcessor接口，本质上是一个后置处理器。



```java
 * @since 10.10.2003
 * @see org.springframework.context.EnvironmentAware
 * @see org.springframework.context.EmbeddedValueResolverAware
 * @see org.springframework.context.ResourceLoaderAware
 * @see org.springframework.context.ApplicationEventPublisherAware
 * @see org.springframework.context.MessageSourceAware
 * @see org.springframework.context.ApplicationContextAware
 * @see org.springframework.context.support.AbstractApplicationContext#refresh()
 */
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 ||
```

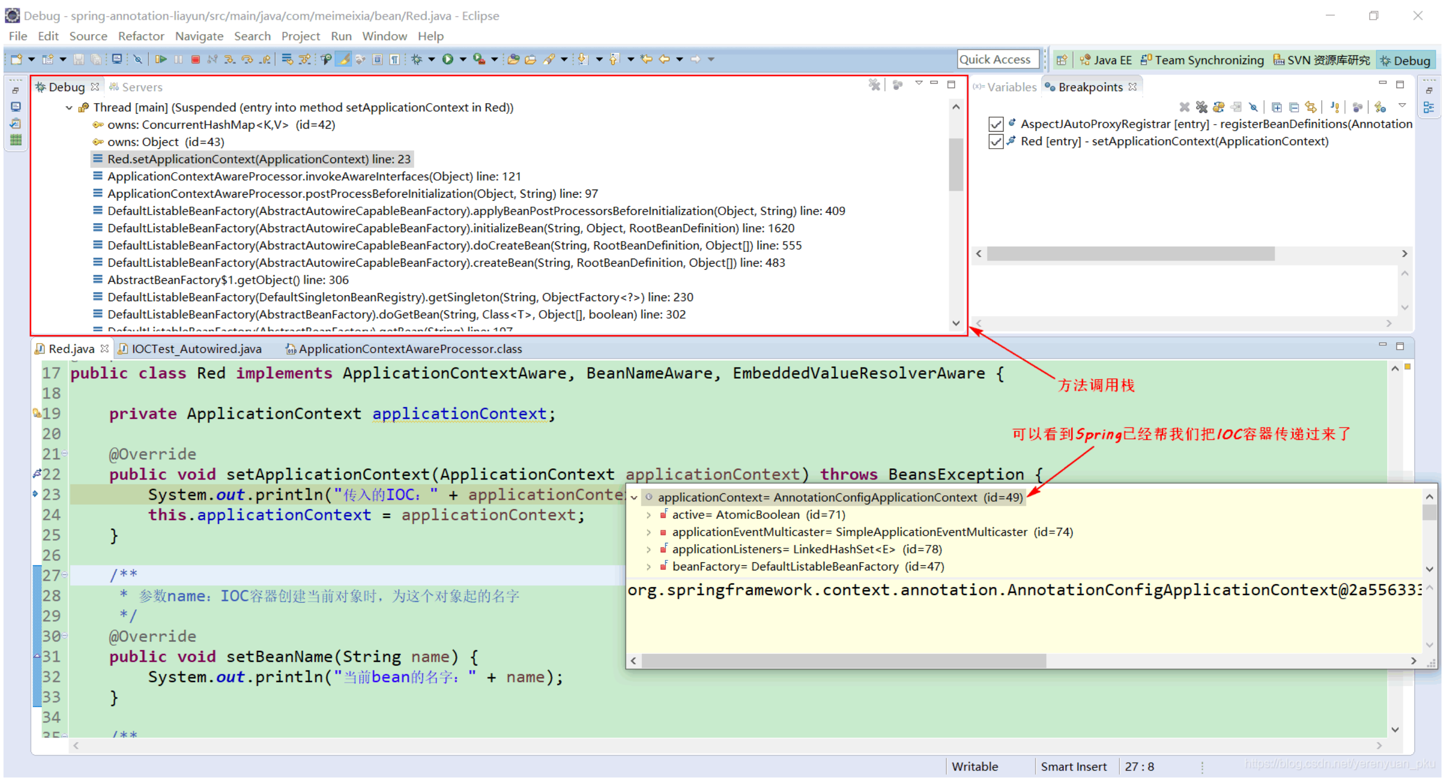
接下来，我们就以分析ApplicationContextAware接口的原理为例，看看Spring是怎么将ApplicationContext对象注入到Red类中的。

首先，我们在Red类的setApplicationContext()方法上打一个断点，如下所示。



```java
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.EmbeddedValueResolverAware;
import org.springframework.stereotype.Component;
import org.springframework.util.StringValueResolver;

/**
 * 以Red类为例来讲解ApplicationContextAware接口、BeanNameAware接口以及EmbeddedValueResolverAware接口
 * @author liayun
 *
 */
@Component
public class Red implements ApplicationContextAware, BeanNameAware, EmbeddedValueResolverAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("传入的IOC：" + applicationContext);
        this.applicationContext = applicationContext;
    }

    /**
     * 参数name：IOC容器创建当前对象时，为这个对象起的名字
     */
    @Override
    public void setBeanName(String name) {
        System.out.println("当前bean的名字：" + name);
    }

    /**
```

然后，我们以debug的方式来运行IOCTest_Autowired类中的test02()方法。


Debug - spring-annotation-liayun/src/main/java/com/meimeixia/bean/Red.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java EE
Team Synchronizing
SVN 资源库研究
Debug
Debug
Servers
Variables
Breakpoints
Red.java
IOCTest_Autowired.java
ApplicationContextAwareProcessor.class
方法调用栈
可以看到Spring已经帮我们把IOC容器传递过来了
Writable
Smart Insert
27 : 8

```
Thread [main] (Suspended (entry into method setApplicationContext in Red))
    owns: ConcurrentHashMap<K,V>  (id=42)
    owns: Object  (id=43)
    Red.setApplicationContext(ApplicationContext) line: 23
    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
    DefaultListableBeanFactory(AbstractBeanFactory).doGetBean(String, Class<T>, Object[], boolean) line: 302
```

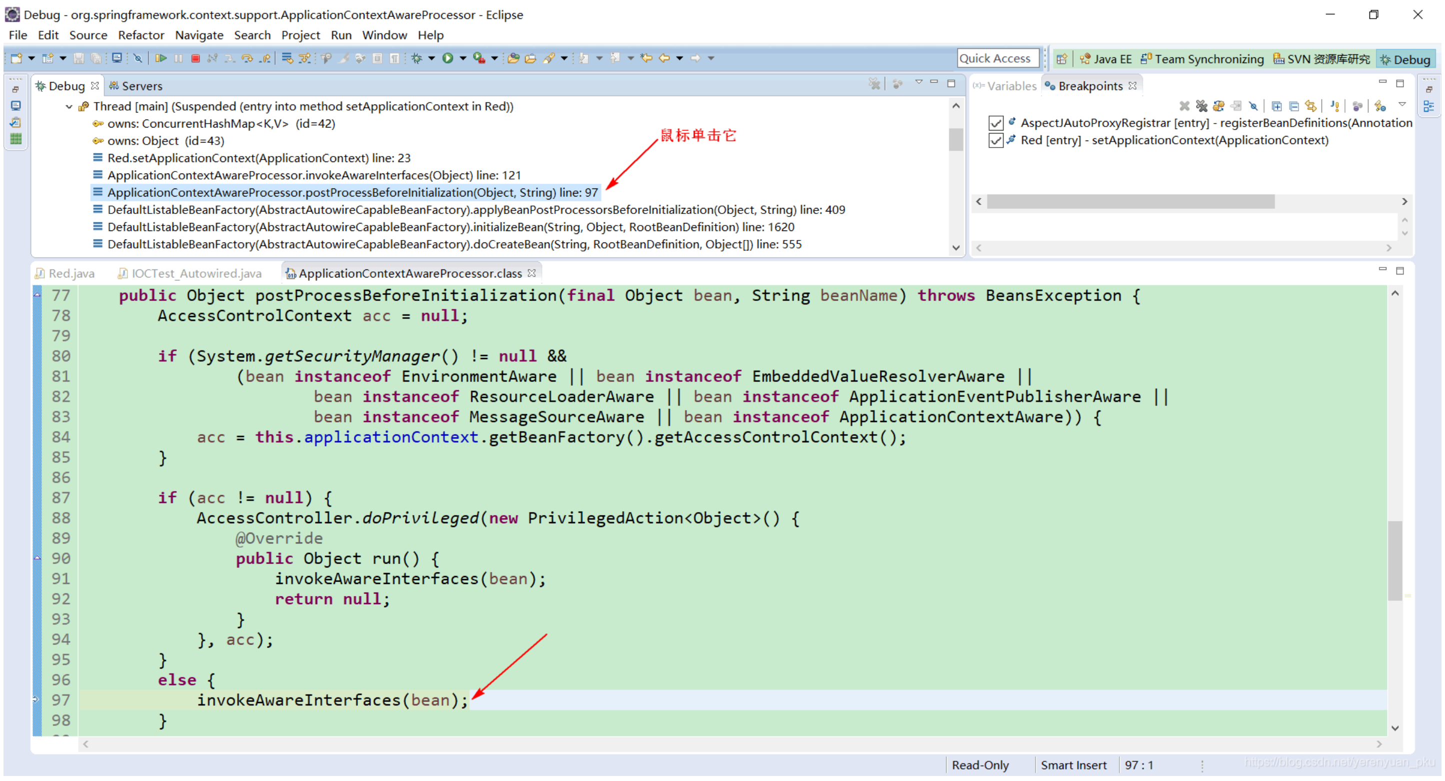
```
AspectJAutoProxyRegistrar [entry] - registerBeanDefinitions(Annotation
Red [entry] - setApplicationContext(ApplicationContext)
```

```java
public class Red implements ApplicationContextAware, BeanNameAware, EmbeddedValueResolverAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("传入的IOC: " + applicationConte
        this.applicationContext = applicationContext;
    }

    /**
     * 参数name：IOC容器创建当前对象时，为这个对象起的名字
     */
    @Override
    public void setBeanName(String name) {
        System.out.println("当前bean的名字：" + name);
    }

    /**
```

```
applicationContext= AnnotationConfigApplicationContext  (id=49)
    active= AtomicBoolean  (id=71)
    applicationEventMulticaster= SimpleApplicationEventMulticaster  (id=74)
    applicationListeners= LinkedHashSet<E>  (id=78)
    beanFactory= DefaultListableBeanFactory  (id=47)
org.springframework.context.annotation.AnnotationConfigApplicationContext@2a556333
```

这里，我们可以看到，实际上ApplicationContext对象已经注入到Red类的setApplicationContext()方法中了。

接着，我们在Eclipse　的方法调用栈中找到postProcessBeforeInitialization()方法并鼠标单击它，如下所示，此时，自动定位到了postProcessBeforeInitialization()方法中。


Debug - org.springframework.context.support.ApplicationContextAwareProcessor - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java EE
Team Synchronizing
SVN 资源库研究
Debug
Debug
Servers
Variables
Breakpoints
鼠标单击它
Red.java
IOCTest_Autowired.java
ApplicationContextAwareProcessor.class
Read-Only
Smart Insert
97 : 1

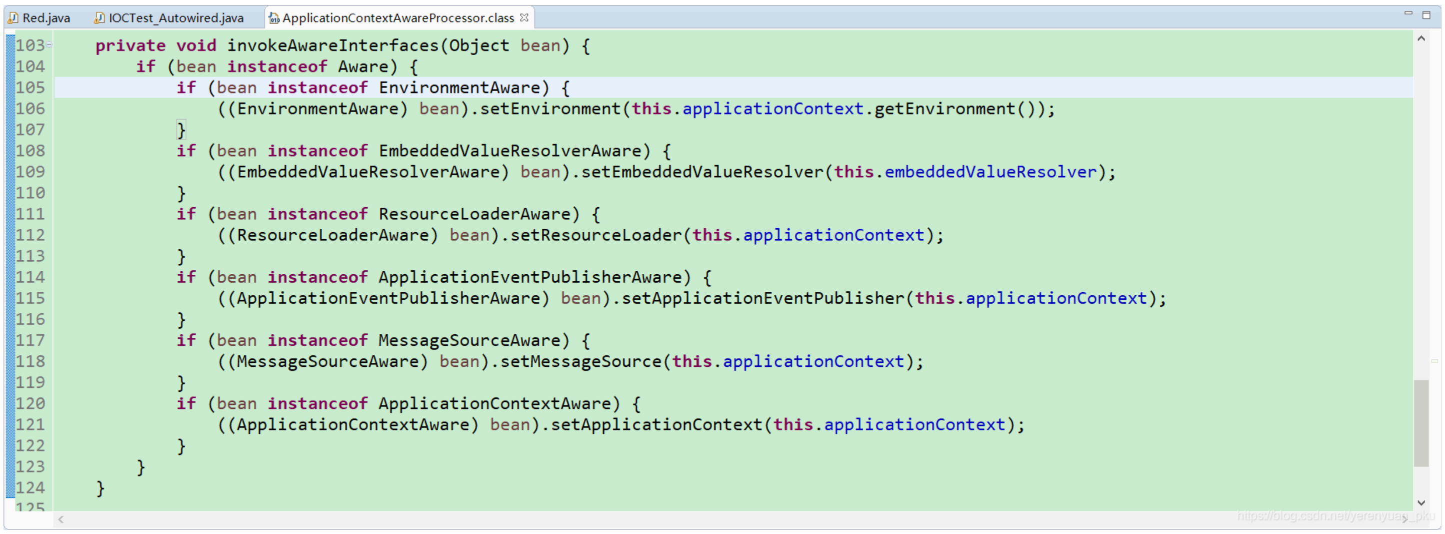
```
Thread [main] (Suspended (entry into method setApplicationContext in Red))
    owns: ConcurrentHashMap<K,V>  (id=42)
    owns: Object  (id=43)
    Red.setApplicationContext(ApplicationContext) line: 23
    ApplicationContextAwareProcessor.invokeAwareInterfaces(Object) line: 121
    ApplicationContextAwareProcessor.postProcessBeforeInitialization(Object, 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
```

```
AspectJAutoProxyRegistrar [entry] - registerBeanDefinitions(Annotation
Red [entry] - setApplicationContext(ApplicationContext)
```

```java
    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);
        }
```

其实，postProcessBeforeInitialization()方法所在的类就是ApplicationContextAwareProcessor。postProcessBeforeInitialization()方法的逻辑还算比较简单。

紧接着，我们来看下在postProcessBeforeInitialization()方法中调用的invokeAwareInterfaces()方法，如下所示。



```
    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);
            }
        }
    }
```

看到这里，大家是不是有种豁然开朗的感觉啊！Red类实现了ApplicationContextAware接口后，Spring为啥会将ApplicationContext对象自动注入到setApplicationContext()方法中就不用我再说了吧！

其实XxxAware接口的原理就是这么简单！