



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

【死磕 Spring】—— IOC 之 深入分析 Aware 接口 (<http://cmsblogs.com/?p=3335>)

2018-12-09 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(8966) 评论(0)

原文出自: <http://cmsblogs.com> (<http://cmsblogs.com>)

doCreateBean() 方法主要干三件事情:

1. 实例化 bean 对象: `createBeanInstance()`
2. 属性注入: `populateBean()`
3. 初始化 bean 对象: `initializeBean()`

而初始化 bean 对象时也是干了三件事情:

1. 激活 Aware 方法
2. 后置处理器的应用
3. 激活自定义的 init 方法

接下来三篇文章将会详细分析这三件事情, 这篇主要分析 Aware 接口。

Aware 接口定义如下:






```
/**
 * 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 {

}
```

Aware 接口为 Spring 容器的核心接口，是一个具有标识作用的超级接口，实现了该接口的 bean 是具有被 Spring 容器通知的能力，通知的方式是采用回调的方式。

Aware 接口是一个空接口，实际的方法签名由各个子接口来确定，且该接口通常只会有一个接收单参数的 set 方法，该 set 方法的命名方式为 set + 去掉接口名中的 Aware 后缀，即 XxxAware 接口，则方法定义为 setXxx() ， 例如 BeanNameAware (setBeanName) ， ApplicationContextAware (setApplicationContext) 。

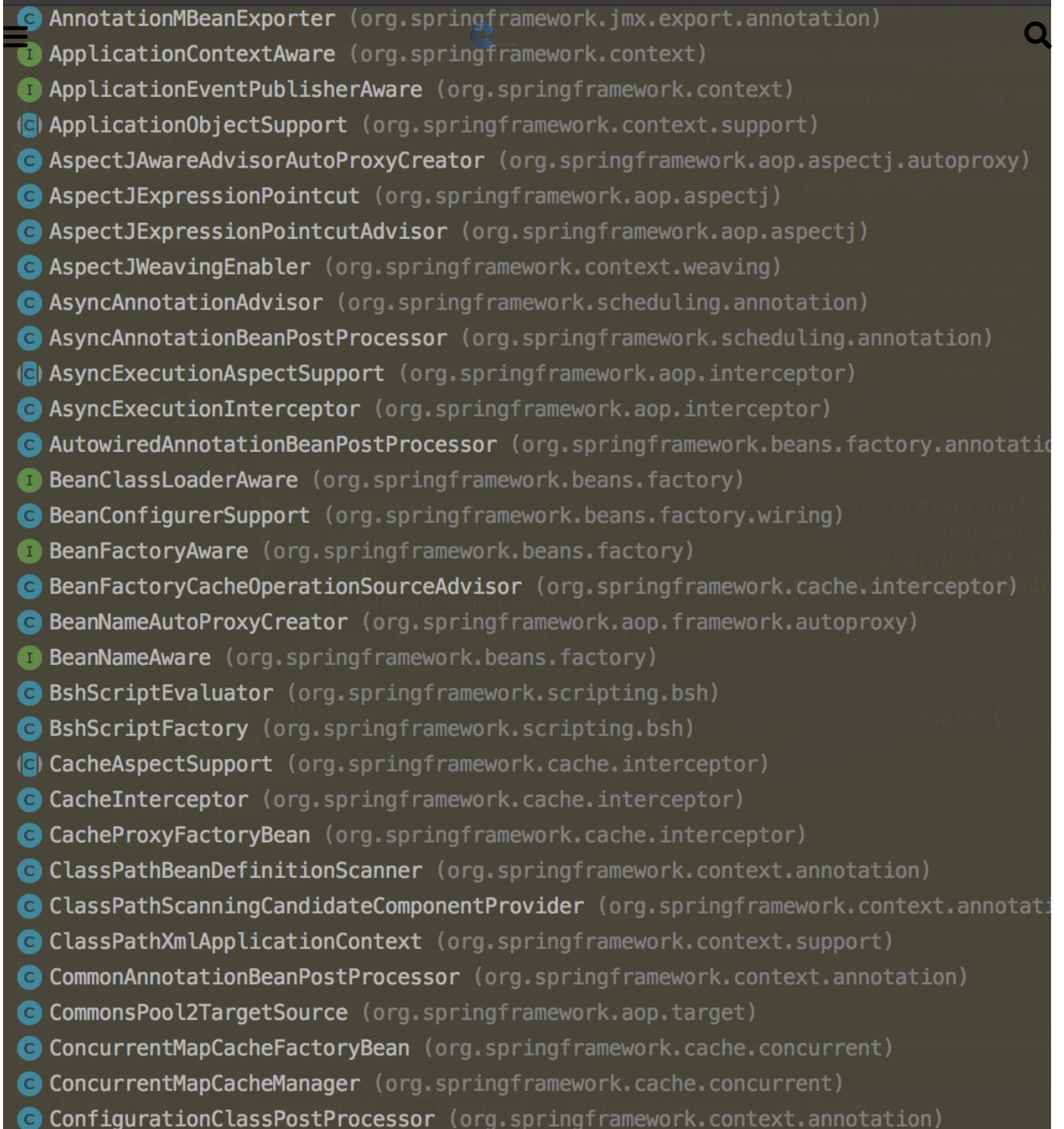
Aware 的子接口需要提供一个 setXxx 方法，我们知道 set 是设置属性值的方法，即 Aware 类接口的 setXxx 方法其实就是设置 xxx 属性值的。Aware 的含义是感知的、感应的，那么在 Spring 容器中是如何实现感知并设置属性值得呢？我们可以从初始化 bean 中的激活 Aware 的方法 invokeAwareMethods() 中看到一点点，如下：



```
private void invokeAwareMethods(final String beanName, final Object bean) {  
    if (bean instanceof Aware) {  
        if (bean instanceof BeanNameAware) {  
            ((BeanNameAware) bean).setBeanName(beanName);  
        }  
        if (bean instanceof BeanClassLoaderAware) {  
            ClassLoader bcl = getBeanClassLoader();  
            if (bcl != null) {  
                ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);  
            }  
        }  
        if (bean instanceof BeanFactoryAware) {  
            ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);  
        }  
    }  
}
```

首先判断 bean 实例是否属于 Aware 接口的范畴，如果是的话，则调用实例的 setXxx() 方法给实例设置 xxx 属性值，在 invokeAwareMethods() 方法主要是设置 beanName、beanClassLoader、BeanFactory 中三个属性值。

Spring 提供了一系列的 Aware 接口，如下图（部分）：



A screenshot of a list of Spring Aware interfaces and their packages. The list is displayed in a dark-themed editor with a search bar in the top right corner. The interfaces are listed with their package names in parentheses. The list includes:

- AnnotationMBeanExporter (org.springframework.jmx.export.annotation)
- ApplicationContextAware (org.springframework.context)
- ApplicationEventPublisherAware (org.springframework.context)
- ApplicationObjectSupport (org.springframework.context.support)
- AspectJAwareAdvisorAutoProxyCreator (org.springframework.aop.aspectj.autoproxy)
- AspectJExpressionPointcut (org.springframework.aop.aspectj)
- AspectJExpressionPointcutAdvisor (org.springframework.aop.aspectj)
- AspectJWeavingEnabler (org.springframework.context.weaving)
- AsyncAnnotationAdvisor (org.springframework.scheduling.annotation)
- AsyncAnnotationBeanPostProcessor (org.springframework.scheduling.annotation)
- AsyncExecutionAspectSupport (org.springframework.aop.interceptor)
- AsyncExecutionInterceptor (org.springframework.aop.interceptor)
- AutowiredAnnotationBeanPostProcessor (org.springframework.beans.factory.annotation)
- BeanClassLoaderAware (org.springframework.beans.factory)
- BeanConfigurerSupport (org.springframework.beans.factory.wiring)
- BeanFactoryAware (org.springframework.beans.factory)
- BeanFactoryCacheOperationSourceAdvisor (org.springframework.cache.interceptor)
- BeanNameAutoProxyCreator (org.springframework.aop.framework.autoproxy)
- BeanNameAware (org.springframework.beans.factory)
- BshScriptEvaluator (org.springframework.scripting.bsh)
- BshScriptFactory (org.springframework.scripting.bsh)
- CacheAspectSupport (org.springframework.cache.interceptor)
- CacheInterceptor (org.springframework.cache.interceptor)
- CacheProxyFactoryBean (org.springframework.cache.interceptor)
- ClassPathBeanDefinitionScanner (org.springframework.context.annotation)
- ClassPathScanningCandidateComponentProvider (org.springframework.context.annotation)
- ClassPathXmlApplicationContext (org.springframework.context.support)
- CommonAnnotationBeanPostProcessor (org.springframework.context.annotation)
- CommonsPool2TargetSource (org.springframework.aop.target)
- ConcurrentMapCacheFactoryBean (org.springframework.cache.concurrent)
- ConcurrentMapCacheManager (org.springframework.cache.concurrent)
- ConfigurationClassPostProcessor (org.springframework.context.annotation)

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808210001.png>)

上面只是一部分子类，从这里我们可以看到 Spring 提供的 Aware 接口是是何其多。同时从上图我们也看到了几个比较熟悉的接口，如 **BeanClassLoaderAware**、**BeanFactoryAware**、**BeanNameAware**，下面就这三个接口来做一个简单的演示，先看各自的定义：

public interface BeanClassLoaderAware extends Aware {

```
/**
 * 将 BeanClassLoader 提供给 bean 实例回调
 * 在 bean 属性填充之后、初始化回调之前回调，
 * 例如InitializingBean的InitializingBean.afterPropertiesSet () 方法或自定义init方法
 */
void setBeanClassLoader(ClassLoader classLoader);
}
```

```
public interface BeanFactoryAware extends Aware {
    /**
     * 将 BeanFactory 提供给 bean 实例回调
     * 调用时机和 setBeanClassLoader 一样
     */
    void setBeanFactory(BeansFactory beanFactory) throws BeansException;
}
```

```
public interface BeanNameAware extends Aware {
    /**
     * 在创建此 bean 的 bean工厂中设置 beanName
     */
    void setBeanName(String name);
}
```

```
public interface ApplicationContextAware extends Aware {
    /**
     * 设置此 bean 对象的 ApplicationContext，通常，该方法用于初始化对象
     */
    void setApplicationContext(ApplicationContext applicationContext) throws BeansException;
}
```

下面简单演示下上面四个接口的使用方法：

```
public class MyApplicationAware implements BeanNameAware, BeanFactoryAware, BeanClassLoaderAware, ApplicationContextAware {

    private String beanName;

    private BeanFactory beanFactory;

    private ClassLoader classLoader;

    private ApplicationContext applicationContext;

    @Override
    public void setBeanClassLoader(ClassLoader classLoader) {
        System.out.println("调用了 BeanClassLoaderAware 的 setBeanClassLoader 方法");

        this.classLoader = classLoader;
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("调用了 BeanFactoryAware 的 setBeanFactory 方法");

        this.beanFactory = beanFactory;
    }

    @Override
    public void setBeanName(String name) {
        System.out.println("调用了 BeanNameAware 的 setBeanName 方法");

        this.beanName = name;
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("调用了 ApplicationContextAware 的 setApplicationContext 方法");

        this.applicationContext = applicationContext;
    }

    public void display(){
        System.out.println("beanName:" + beanName);

        System.out.println("是否为单例: " + beanFactory.isSingleton(beanName));

        System.out.println("系统环境为: " + applicationContext.getEnvironment());
    }
}
```

测试方法如下:

```

public static void main(String[] args) {
    ClassPathResource resource = new ClassPathResource("spring.xml");
    DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
    reader.loadBeanDefinitions(resource);

    MyApplicationAware applicationAware = (MyApplicationAware) factory.getBean("myApplicationAware");
    applicationAware.display();
}

```

运行结果:

```

调用了 BeanNameAware 的 setBeanName 方法
调用了 BeanClassLoaderAware 的 setBeanClassLoader 方法
调用了 BeanFactoryAware 的 setBeanFactory 方法
beanName:myApplicationAware
是否为单例: true
Exception in thread "main" java.lang.NullPointerException
    at org.springframework.core.test1.MyApplicationAware.display(MyApplicationAware.java:54)
    at org.springframework.core.LoadBeanDefinitionTest.main(LoadBeanDefinitionTest.java:20)

```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808210002.png>)

从该运行结果可以看出, 这里只执行了三个 Aware 接口的 set 方法, 原因就是调用 `getBean()` 调用时在激活 Aware 接口时只检测了 `BeanNameAware`、`BeanClassLoaderAware`、`BeanFactoryAware` 三个 Aware 接口。如果将测试方法调整为下面:

```

public static void main(String[] args) {
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
    MyApplicationAware applicationAware = (MyApplicationAware) applicationContext.getBean("myApplicationAware");
    applicationAware.display();
}

```

如果一个类实现了上面的 Aware 接口, 则通过 `context.getBean("myAware")` 时候, 创建 bean 实例对象, 执行实例对象初始化之前, 执行实现的 Aware 接口, 把 ioc 的环境信息设置到实例对象中, 让实例对象感知到 ioc 容器环境。

则运行结果如下:

```

调用了 BeanNameAware 的 setBeanName 方法
调用了 BeanClassLoaderAware 的 setBeanClassLoader 方法
调用了 BeanFactoryAware 的 setBeanFactory 方法
调用了 ApplicationContextAware 的 setApplicationContext 方法
beanName:myApplicationAware
是否为单例: true
系统环境为: StandardEnvironment {activeProfiles=[], defaultProfiles=[default], propertySources=[MapPropertySource {name='systemProperties'}, SystemEnvironmentPropertySource {name='systemEnvironment'}]}


```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808210003.png>)

从这了我们基本上就可以 Aware 真正的含义是什么了? 感知, 其实是 Spring 容器在初始化主动检测当前 bean 是否实现了 Aware 接口, 如果实现了则回调其 set 方法将相应的参数设置给该 bean, 这个时候该 bean 就从 Spring 容器中取得相应的资源。最后文章末尾列出部分常用的 Aware 子接口, 便于日后查询:

- LoadTimeWeaverAware: 加载 Spring Bean 时织入第三方模块, 如 AspectJ
- BeanClassLoaderAware: 加载 Spring Bean 的类加载器
- BootstrapContextAware: 资源适配器 BootstrapContext, 如 JCA, CCI
- ResourceLoaderAware: 底层访问资源的加载器

- BeanFactoryAware: 声明BeanFactory
- PortletConfigAware: PortletConfig
- PortletContextAware: PortletContext
- ServletConfigAware: ServletConfig
- ServletContextAware: ServletContext
- MessageSourceAware: 国际化
- ApplicationEventPublisherAware: 应用事件
- NotificationPublisherAware: JMX通知
- BeanNameAware: 声明Spring Bean的名字

 赞(6) 打赏

【公告】版权声明 (http://cmsblogs.com/?page_id=1908)

标签: [Spring源码解析](http://cmsblogs.com/?tag=spring%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) (<http://cmsblogs.com/?tag=spring%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90>)

[死磕 Java](http://cmsblogs.com/?tag=%E6%AD%BB%E7%A3%95-java) (<http://cmsblogs.com/?tag=%E6%AD%BB%E7%A3%95-java>)

[死磕Spring](http://cmsblogs.com/?tag=%E6%AD%BB%E7%A3%95spring) (<http://cmsblogs.com/?tag=%E6%AD%BB%E7%A3%95spring>)

 **chenssy** (<http://cmsblogs.com/?author=1>)

不想当厨师的程序员不是好的架构师....

上一篇

双十一瞬间点击量过万, Redis 热点 Key 问题发现与5种
解决方案 (<http://cmsblogs.com/?p=3279>)

下一篇

【死磕 Spring】—— IOC 之 深入分析
BeanPostProcessor (<http://cmsblogs.com/?p=3338>)

- 【死磕 Spring】—— IOC 总结 (<http://cmsblogs.com/?p=4047>)
- 【死磕 Spring】—— 4 张图带你读懂 Spring IOC 的世界 (<http://cmsblogs.com/?p=4045>)
- 【死磕 Spring】—— 深入分析 ApplicationContext 的 refresh() (<http://cmsblogs.com/?p=4043>)
- 【死磕 Spring】—— ApplicationContext 相关接口架构分析 (<http://cmsblogs.com/?p=4036>)
- 【死磕 Spring】—— IOC 之 分析 bean 的生命周期 (<http://cmsblogs.com/?p=4034>)
- 【死磕 Spring】—— Spring 的环境&属性: PropertySource、Environment、Profile (<http://cmsblogs.com/?p=4032>)
- 【死磕 Spring】—— IOC 之 BeanDefinition 注册机: BeanDefinitionRegistry (<http://cmsblogs.com/?p=4026>)
- 【死磕 Spring】—— IOC 之 bean 的实例化策略: InstantiationStrategy (<http://cmsblogs.com/?p=4022>)
- 【死磕 Spring】—— IOC 之分析 BeanWrapper (<http://cmsblogs.com/?p=4020>)
- 【死磕 Spring】—— IOC 之自定义类型转换器 (<http://cmsblogs.com/?p=3985>)