# 【死磕 Spring】—— IOC 之 bean 的初始化 (http://cmsblogs.com/?p=2890)

2018-11-05 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(7413) 评论(0)

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

一个 bean 经历了 `createBeanInstance()` 被创建出来，然后又经过一番属性注入，依赖处理，历经千辛万苦，千锤百炼，终于有点儿 bean 实例的样子，能堪大任了，只需要经历最后一步就破茧成蝶了。这最后一步就是初始化，也就是 `initializeBean()`，所以这篇文章我们分析 `doCreateBean()` 中最后一步：初始化 bean。

```
    protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinitio
n mbd) {
        if (System.getSecurityManager() != null) {
            AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                // 激活 Aware 方法
                invokeAwareMethods(beanName, bean);
                return null;
            }, getAccessControlContext());
        }
        else {
            // 对特殊的 bean 处理：Aware、BeanClassLoaderAware、BeanFactoryAware
            invokeAwareMethods(beanName, bean);
        }

        Object wrappedBean = bean;
        if (mbd == null || !mbd.isSynthetic()) {
            // 遍历所有的后置处理器，执行后置处理器的postProcessBeforeInitialization(Object bean, String beanName)
            wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
        }

        try {
            // 激活用户自定义的 init-method 方法
            invokeInitMethods(beanName, wrappedBean, mbd);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(
                    (mbd != null ? mbd.getResourceDescription() : null),
                    beanName, "Invocation of init method failed", ex);
        }
        if (mbd == null || !mbd.isSynthetic()) {
            // 遍历所有的后置处理器，执行后置处理器的postProcessAfterInitialization(Object bean, String beanName)
            wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
        }
        return wrappedBean;
    }
```

初始化 bean 的方法其实就是三个步骤的处理，而这三个步骤主要还是根据用户设定的来进行初始化，这三个过程为：

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

**激活 Aware 方法** Aware ,英文翻译是意识到的，感知的，Spring 提供了诸多 **Aware 接口用于辅助 Spring Bean 以编程的方式调用 Spring 容器，通过实现这些接口，可以增强 Spring Bean 的功能。 Spring 提供了如下系列的 Aware 接口：

- 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

Aware主要是提供Bean一个感受到ioc容器的功能
Spring的依赖注入的最大亮点是所有的Bean对Spring容器的存在是没有意识的，我们可以将Spring容器换成其他的容器，Spring容器中的Bean的耦合度因此也是极低的。
但是我们在实际的开发中，我们却经常要用到Spring容器本身的功能资源，所以Spring容器中的Bean此时就要意识到Spring容器的存在才能调用Spring所提供的资源。我们通过Spring提供的一系列接口Spring Aware来实现具体的功能

`invokeAwareMethods()` 源码如下：

```
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);
        }
    }
}
```

这里代码就没有什么好说的，主要是处理 BeanNameAware 、 BeanClassLoaderAware 、 BeanFactoryAware。关于 Aware 接口，后面会专门出篇文章对其进行详细分析说明的。 **后置处理器的应用**
BeanPostProcessor 在前面介绍 bean 加载的过程曾多次遇到，相信各位不陌生，这是 Spring 中开放式框架中必不可少的一个亮点。 BeanPostProcessor 的作用是：如果我们想要在 Spring 容器完成 Bean 的实例化，配置和其他的初始化后添加一些自己的逻辑处理，那么请使用该接口，这个接口给与了用户充足的权限去更改或者扩展 Spring，是我们对 Spring 进行扩展和增强处理一个必不可少的接口。

```
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
        throws BeansException {

    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        Object current = beanProcessor.postProcessBeforeInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}

@Override
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
        throws BeansException {

    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        Object current = beanProcessor.postProcessAfterInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}
```

其实逻辑就是通过 `getBeanPostProcessors()` 获取定义的 BeanPostProcessor ，然后分别调用其 `postProcessBeforeInitialization()` 、 `postProcessAfterInitialization()` 进行业务处理。 **激活自定义的 init 方法** 如果熟悉 `<bean>` 标签的配置，一定不会忘记 `init-method` 方法，该方法的执行就是在这里执行的。

```
protected void invokeInitMethods(String beanName, final Object bean, @Nullable RootBeanDefinition mbd)
        throws Throwable {
    // 首先会检查bean是否实现了InitializingBean接口 ，如果是的话需要调用 bean的afterPropertiesSet()方法
    boolean isInitializingBean = (bean instanceof InitializingBean);
      if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"
))) {
        if (logger.isDebugEnabled()) {
            logger.debug("Invoking afterPropertiesSet() on bean with name '" + beanName + "'");
        }
        if (System.getSecurityManager() != null) {
            try {
                AccessController.doPrivileged((PrivilegedExceptionAction<Object>) () -> {
                    ((InitializingBean) bean).afterPropertiesSet();
                    return null;
                }, getAccessControlContext());
            }
            catch (PrivilegedActionException pae) {
                throw pae.getException();
            }
        }
        else {
            // 属性初始化的处理 执行bean的afterPropertiesSet方法
            ((InitializingBean) bean).afterPropertiesSet();
        }
    }

    if (mbd != null && bean.getClass() != NullBean.class) {
        String initMethodName = mbd.getInitMethodName();
        if (StringUtils.hasLength(initMethodName) &&
                !(isInitializingBean && "afterPropertiesSet".equals(initMethodName)) &&
                !mbd.isExternallyManagedInitMethod(initMethodName)) {
            // 激活用户自定义的 初始化方法init-method
            invokeCustomInitMethod(beanName, bean, mbd);
        }
    }
}
```

首先检查是否为 InitializingBean ，如果是的话需要执行 `afterPropertiesSet()` ，因为我们除了可以使用 `init-method` 来自定初始化方法外，还可以实现 InitializingBean 接口，该接口仅有一个 `afterPropertiesSet()` 方法，而两者的执行先后顺序是先 `afterPropertiesSet()` 后 `init-method` 。关于这篇博客的三个问题，LZ 后面会单独写博客来进行分析说明。 经过六篇博客终于把 Spring 创建 bean 的过程进行详细说明了，过程是艰辛的，但是收获很大，关键还是要耐着性子看。 **更多阅读**

- 【死磕 Spring】—— IOC 之开启 bean 的实例化进程 (http://cmsblogs.com/?p=2846)
- 【死磕 Spring】—— IOC 之 Factory 实例化 bean (http://cmsblogs.com/?p=2848)
- 【死磕 Spring】—— IOC 之构造函数实例化 bean (http://cmsblogs.com/?p=2850)
- 【死磕 Spring】----- IOC 之 属性填充 (http://cmsblogs.com/?p=2885)
- 【死磕 Spring】----- IOC 之循环依赖处理 (http://cmsblogs.com/?p=2887)

## 【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)

死磕Java (http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java)

死磕Spring (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
【死磕 Spring】—— IOC 之循环依赖处理 (http://cmsblogs.com/?p=2887)

下一篇
科普：String hashCode 方法为什么选择数字31作为乘子 (http://cmsblogs.com/?p=2897)