



[回到首页](#)

## 芋道源码 —— 知识星球

我是一段不羁的公告！

记得给芳芳这 3 个项目加油，添加一个 STAR 噢。

<https://github.com/YunaiV/SpringBoot-Labs>

<https://github.com/YunaiV/onemall>

<https://github.com/YunaiV/ruoyi-vue-pro>

2019-07-10

[Spring](#)

# 【死磕 Spring】—— IoC 之深入分析 Bean 的生命周期

本文主要基于 Spring 5.0.6.RELEASE

摘要：原创出处 <http://cmsblogs.com/?p=todo> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老芳芳」略作修改，记录在理解过程中，参考的资料。

在分析 Spring Bean 实例化过程中提到 Spring 并不是一启动容器就开启 bean 的实例化进程，只有当客户端通过显示或者隐式的方式调用 BeanFactory 的 `#getBean(...)` 方法来请求某个实例对象的时候，它才会触发相应 bean 的实例化进程。当然，也可以选择直接使用 `ApplicationContext` 容器，因为该容器启动的时候会立刻调用注册到该容器所有 bean 定义的实例化方法。当然，对于 `BeanFactory` 容器而言，并不是所有的 `#getBean(...)` 方法都会触发实例化进程，比如 `singleton` 类型的 bean，该类型的 bean 只会在第一次调用 `getBean()` 的时候才会触发，而后续的调用则会直接返回容器缓存中的实例对象。

`#getBean(...)` 方法，只是 bean 实例化进程的入口，真正的实现逻辑其实是在 `AbstractAutowireCapableBeanFactory` 的 `#doCreateBean(...)` 中实现，实例化过程如下图：



原来我们采用 new 的方式创建一个对象，用完该对象在其脱离作用域后就会被回收，对于后续操作我们无权也没法干涉，但是采用 Spring 容器后，我们完全摆脱了这种命运，Spring 容器将会对其所有管理的 Bean 对象全部给予一个统一的生命周期管理，同时在这个阶段我们也可以对其进行干涉（比如对 bean 进行增强处理，对 bean 进行篡改），如上图。

## 1. bean 实例化

在 #doCreateBean(...) 方法中，首先进行 bean 实例化工作，主要由 #createBeanInstance(...) 方法实现，该方法返回一个 BeanWrapper 对象。BeanWrapper 对象是 Spring 的一个低级 Bean 基础结构的核心接口，为什么说是低级呢？因为这个时候的 Bean 还不能够被我们使用，连最基本的属性都没有设置。而且在我们实际开发过程中，一般都不会直接使用该类，而是通过 BeanFactory 隐式使用。

BeanWrapper 接口有一个默认实现类 BeanWrapperImpl，其主要作用是对 Bean 进行“包裹”，然后对这个包裹的 bean 进行操作，比如后续注入 bean 属性。

在实例化 bean 过程中，Spring 采用“策略模式”来决定采用哪种方式来实例化 bean，一般有反射和 CGLIB 动态字节码两种方式。

InstantiationStrategy 定义了 Bean 实例化策略的抽象接口，其子类 SimpleInstantiationStrategy 提供了基于反射来实例化对象的功能，但是不支持方法注入方式的对象实例化。CglibSubclassingInstantiationStrategy 继承 SimpleInstantiationStrategy，他除了拥有父类以反射实例化对象的功能外，还提供了通过 CGLIB 的动态字节码的功能进而支持方法注入所需的对象实例化需求。默认情况下，Spring 采用 CglibSubclassingInstantiationStrategy。

关于 Bean 实例化的详细过程，请参考以下几篇文章：

1. [【死磕 Spring】—— IoC 之加载 bean: 创建 bean（一）之主流程](#)
2. [【死磕 Spring】—— IoC 之加载 bean: 创建 bean（二）之实例化 Bean 对象\(1\)](#)
3. [【死磕 Spring】—— IoC 之加载 bean: 创建 bean（三）之实例化 Bean 对象\(2\)](#)
4. [【死磕 Spring】—— IoC 之加载 bean: 创建 bean（四）之属性填充](#)
5. [【死磕 Spring】—— IoC 之加载 bean: 创建 bean（五）之循环依赖处理](#)
6. [【死磕 Spring】—— IoC 之加载 bean: 创建 bean（六）之初始化 Bean 对象](#)

对于 BeanWrapper 和 具体的实例化策略，LZ 在后面会专门写文章来进行详细说明。

## 2. 激活 Aware

当 Spring 完成 bean 对象实例化并且设置完相关属性和依赖后，则会开始 bean 的初始化进程（#initializeBean(...)），初始化第一个阶段是检查当前 bean 对象是否实现了一系列以 Aware 结尾的接口。

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。

在初始化阶段主要是感知 BeanNameAware、BeanClassLoaderAware、BeanFactoryAware。代码如下：

```
// AbstractAutowireCapableBeanFactory.java
```

```

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

```

BeanNameAware: 对该 bean 对象定义的 beanName 设置到当前对象实例中

BeanClassLoaderAware: 将当前 bean 对象相应的 ClassLoader 注入到当前对象实例中

BeanFactoryAware: BeanFactory 容器会将自身注入到当前对象实例中，这样当前对象就会拥有一个 BeanFactory 容器的引用。

当然，Spring 不仅仅只是提供了上面三个 Aware 接口，而是一系列：

LoadTimeWeaverAware: 加载Spring Bean时织入第三方模块，如AspectJ

BootstrapContextAware: 资源适配器BootstrapContext，如JCA, CCI

ResourceLoaderAware: 底层访问资源的加载器

PortletConfigAware: PortletConfig

PortletContextAware: PortletContext

ServletConfigAware: ServletConfig

ServletContextAware: ServletContext

MessageSourceAware: 国际化

ApplicationEventPublisherAware: 应用事件

NotificationPublisherAware: JMX通知

更多关于 Aware 的请关注: [《【死磕 Spring】—— IoC 之深入分析 Aware 接口》](#)。

### 3. BeanPostProcessor

初始化第二个阶段则是 BeanPostProcessor 增强处理，在该阶段 BeanPostProcessor 会处理当前容器内所有符合条件的实例化后的 bean 对象。它主要是对 Spring 容器提供的 bean 实例对象进行有效的扩展，允许 Spring 在初始化 bean 阶段对其进行定制化修改，如处理标记接口或者为其提供代理实现。

BeanPostProcessor 接口提供了两个方法，在不同的时机执行，分别对应上图的前置处理和后置处理。代码如下：

```

public interface BeanPostProcessor {

```

```

@Nullable
default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
    return bean;
}

@Nullable
default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    return bean;
}
}

```

更多关于 BeanPostProcessor 的请关注: [《【死磕 Spring】—— IoC 之深入分析 BeanPostProcessor》](#)。

## 4. InitializingBean 和 init-method

InitializingBean 是一个接口，它为 Spring Bean 的初始化提供了一种方式，它有一个 #afterPropertiesSet() 方法，在 bean 的初始化进程中会判断当前 bean 是否实现了 InitializingBean，如果实现了则调用 #afterPropertiesSet() 方法，进行初始化工作。然后再检查是否也指定了 init-method，如果指定了则通过反射机制调用指定的 init-method 方法。代码如下：

```

// AbstractAutowireCapableBeanFactory.java

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

    if (mbd != null && bean.getClass() != NullBean.class) {
        // 判断是否指定了 init-method(),
        // 如果指定了 init-method(), 则再调用指定的 init-method
        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
        // 利用反射机制执行
        invokeCustomInitMethod(beanName, bean, mbd);
    }
}
}

```

对于 Spring 而言，虽然上面两种方式都可以实现初始化定制化，但是更加推崇 `init-method` 方式，因为对于 `InitializingBean` 接口而言，他需要 `bean` 去实现接口，这样就会污染我们的应用程序，显得 Spring 具有一定的侵入性。但是由于 `init-method` 是采用反射的方式，所以执行效率上相对于 `InitializingBean` 接口回调的方式可能会低一些。

更多关于 `init` 的请关注：[《【死磕 Spring】—— IoC 之 深入分析 InitializingBean 和 init-method》](#)

## 5. DisposableBean 和 destroy-method

与 `InitializingBean` 和 `init-method` 用于对象的自定义初始化工作相似，`DisposableBean` 和 `destroy-method` 则用于对象的自定义销毁工作。

当一个 `bean` 对象经历了实例化、设置属性、初始化阶段，那么该 `bean` 对象就可以供容器使用了（调用的过程）。当完成调用后，如果是 `singleton` 类型的 `bean`，则会看当前 `bean` 是否应实现了 `DisposableBean` 接口或者配置了 `destroy-method` 属性，如果是的话，则会为该实例注册一个用于对象销毁的回调方法，便于在这些 `singleton` 类型的 `bean` 对象销毁之前执行销毁逻辑。

但是，并不是对象完成调用后就会立刻执行销毁方法，因为这个时候 Spring 容器还处于运行阶段，只有当 Spring 容器关闭的时候才会去调用。但是，Spring 容器不会这么聪明会自动去调用这些销毁方法，而是需要我们主动去告知 Spring 容器。

对于 `BeanFactory` 容器而言，我们需要主动调用 `#destroySingletons()` 方法，通知 `BeanFactory` 容器去执行相应的销毁方法。

对于 `ApplicationContext` 容器而言，调用 `#registerShutdownHook()` 方法。

## 6. 实践验证

下面用一个实例来真实看看上面执行的逻辑，毕竟理论是不能缺少实践的：

```

public class LifecycleBean implements BeanNameAware, BeanFactoryAware, BeanClassLoaderAware, BeanPostProcessor,
    InitializingBean, DisposableBean {

    private String test;

    public String getTest() {
        return test;
    }

    public void setTest(String test) {
        System.out.println("属性注入...");
        this.test = test;
    }
}

```

```

public LifeCycleBean() { // 构造方法
    System.out.println("构造函数调用...");
}

public void display() {
    System.out.println("方法调用...");
}

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

@Override
public void setBeanName(String name) {
    System.out.println("BeanNameAware 被调用...");
}

@Override
public void setBeanClassLoader(ClassLoader classLoader) {
    System.out.println("BeanClassLoaderAware 被调用...");
}

@Override
public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
    System.out.println("BeanPostProcessor postProcessBeforeInitialization 被调用...");
    return bean;
}

@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    System.out.println("BeanPostProcessor postProcessAfterInitialization 被调用...");
    return bean;
}

@Override
public void destroy() throws Exception {
    System.out.println("DisposableBean destroy 被调用...");
}

@Override
public void afterPropertiesSet() throws Exception {
    System.out.println("InitializingBean afterPropertiesSet 被调用...");
}

public void initMethod() {
    System.out.println("init-method 被调用...");
}

public void destroyMethod() {
    System.out.println("destroy-method 被调用...");
}
}

```

**LifeCycleBean** 继承了 `BeanNameAware` , `BeanFactoryAware` , `BeanClassLoaderAware` , `BeanPostProcessor` , `InitializingBean` , `DisposableBean` 六个接口, 同时定义了一个 `test` 属性用于验证属性注入和提供一个 `#display()` 方法用于模拟调用。

配置如下：

```
<bean id="lifeCycle" class="org.springframework.core.test.lifeCycleBean"
      init-method="initMethod" destroy-method="destroyMethod">
    <property name="test" value="test"/>
</bean>
```

配置 init-method 和 destroy-method。

测试方法如下：

```
// BeanFactory 容器一定要调用该方法进行 BeanPostProcessor 注册
factory.addBeanPostProcessor(new LifeCycleBean()); // <1>

LifeCycleBean lifeCycleBean = (LifeCycleBean) factory.getBean("lifeCycle");
lifeCycleBean.display();

System.out.println("方法调用完成，容器开始关闭...");
// 关闭容器
factory.destroySingletons();
```

运行结果：

```
构造函数调用...
构造函数调用...
属性注入...
BeanNameAware 被调用...
BeanClassLoaderAware 被调用...
BeanFactoryAware 被调用...
BeanPostProcessor postProcessBeforeInitialization 被调用...
InitializingBean afterPropertiesSet 被调用...
init-method 被调用...
BeanPostProcessor postProcessAfterInitialization 被调用...
方法调用...
方法调用完成，容器开始关闭...
DisposableBean destroy 被调用...
destroy-method 被调用...
```

有两个构造函数调用，是因为要 <1> 处注入一个 BeanPostProcessor（你也可以另外提供一个 BeanPostProcessor 实例）。

根据执行的结果已经上面的分析，我们就可以对 Spring Bean 的声明周期过程如下（方法级别）：

1. Spring 容器根据实例化策略对 Bean 进行实例化。
2. 实例化完成后，如果该 bean 设置了一些属性的话，则利用 set 方法设置一些属性。
3. 如果该 Bean 实现了 BeanNameAware 接口，则调用 #setBeanName(String beanName) 方法。
4. 如果该 bean 实现了 BeanClassLoaderAware 接口，则调用 setBeanClassLoader(ClassLoader classLoader) 方法。
5. 如果该 bean 实现了 BeanFactoryAware接口，则调用 setBeanFactory(BeaFactory beanFactory) 方法。
6. 如果该容器注册了 BeanPostProcessor，则会调用#postProcessBeforeInitialization(Object bean, String beanName) 方法, 完成 bean 前置处理
7. 如果该 bean 实现了 InitializingBean 接口，则调用#afterPropertiesSet() 方法。

8. 如果该 bean 配置了 `init-method` 方法，则调用其指定的方法。
9. 初始化完成后，如果该容器注册了 `BeanPostProcessor` 则会调用 `#postProcessAfterInitialization(Object bean, String beanName)` 方法, 完成 bean 的后置处理。
10. 对象完成初始化，开始方法调用。
11. 在容器进行关闭之前，如果该 bean 实现了 `DisposableBean` 接口，则调用 `#destroy()` 方法。
12. 在容器进行关闭之前，如果该 bean 配置了 `destroy-method` ，则调用其指定的方法。
13. 到这里一个 bean 也就完成了它的一生。

## 文章目录

1. [1. 1. bean 实例化](#)
2. [2. 2. 激活 Aware](#)
3. [3. 3. BeanPostProcessor](#)
4. [4. 4. InitializingBean 和 init-method](#)
5. [5. 5. DisposableBean 和 destroy-method](#)
6. [6. 6. 实践验证](#)

2014 - 2023 芋道源码 |  
总访客数 次 && 总访问量 次  
[回到首页](#)