13 我的人生我做主揭秘Bean的生命周期

更新时间：2020-08-04 14:00:44

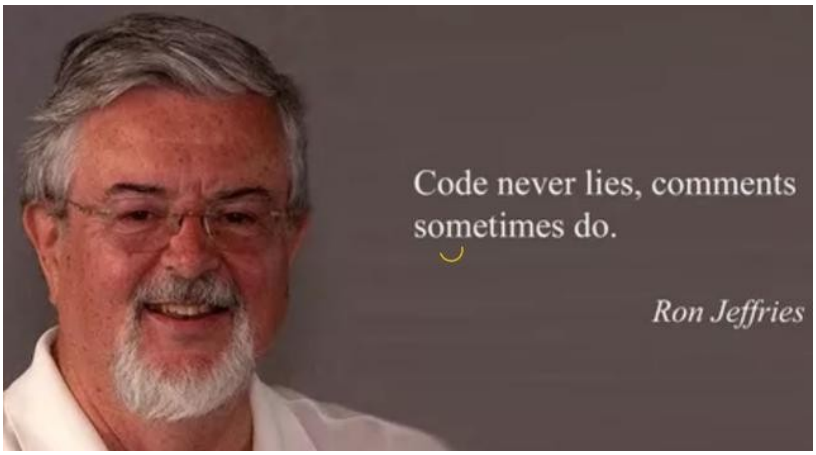
> 成功的奥秘在于目标的坚定。——迪斯雷利

背景

Spring Bean 的生命周期是 Spring 面试热点问题。这个问题即考察对 Spring 的微观了解，又考察对 Spring 的宏观认识，想要答好并不容易！Spring Bean 的生命周期贯穿了 Spring IoC 的整个过程，掌握了 Spring Bean 的生命周期等同于掌握了 Spring IoC 本身。

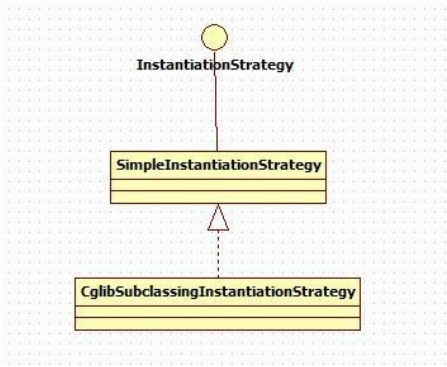
Spring IoC 容器 Bean 生命周期概述

代码不会说谎，但人会。网上看到的资料很多是人云亦云或者是资料已经过期，所以阅读源码成了必然选择。通过简单的示例，抽出 Spring BeanFactory 中 Bean 的生命周期大大超出了我们的预期，原来我们以前看到的或者记住的只是真实中的一部分！



Bean 生成策略

BeanFactory 中 Bean 的生成有两种策略：构造器方式和 CGLIB。



实现逻辑在 SimpleInstantiationStrategy.java#instantiate() 如下：

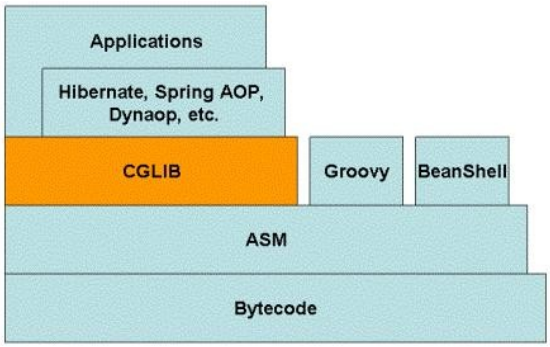
```
@Override
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner) {
    // Don't override the class with CGLIB if no overrides.
    if (!bd.hasMethodOverrides()) {
        Constructor<?> constructorToUse;
        synchronized (bd.constructorArgumentLock) {
            constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;
            if (constructorToUse == null) {
                final Class<?> clazz = bd.getBeanClass();
                if (clazz.isInterface()) {
                    throw new BeanInstantiationException(clazz, "Specified class is an interface");
                }
                try {
                    if (System.getSecurityManager() != null) {
                        constructorToUse = AccessController.doPrivileged(
                                (PrivilegedExceptionAction<Constructor<?>>) clazz::getDeclaredConstructor);
                    }
                    else {
                        constructorToUse = clazz.getDeclaredConstructor();
                    }
                    bd.resolvedConstructorOrFactoryMethod = constructorToUse;
                }
                catch (Throwable ex) {
                    throw new BeanInstantiationException(clazz, "No default constructor found", ex);
                }
            }
        }
        return BeanUtils.instantiateClass(constructorToUse);
    }
    else {
        // Must generate CGLIB subclass.
        return instantiateWithMethodInjection(bd, beanName, owner);
    }
}
```

1 constructor

2 CGLib

1. CGLIB

CGLIB（Code Generator Library）是一个强大的、高性能的代码生成库。CGLIB 代理主要通过对字节码的操作，为对象引入间接级别，以控制对象的访问。我们知道 Java 中有一个动态代理也是做这个事情的，那我们为什么不直接使用 Java 动态代理，而要使用 CGLIB 呢？答案是 CGLIB 相比于 JDK 动态代理更加强大，JDK 动态代理虽然简单易用，但是其有一个致命缺陷是，只能对接口进行代理。如果要代理的类为一个普通类、没有接口，那么 Java 动态代理就没法使用了。



2. 构造器方式

```
BeanUtils.instantiateClass(constructorToUse);
```

其中，BeanUtils 是一个工具栏，它提供了对 javaBean 进行操作的很多便利方法，例如初始化 Bean，检查 Bean 属性类型，复制 Bean 属性。BeanUtils#instantiateClass() 最终通过使用 Constructor.newInstance 方法来实例化 Bean。

注意：BeanUtils 的最著名的两个方法：

copyProperties

instantiateClass

Bean 生命周期实例程序

Debug 测试程序：

```
19 import java.util.Date;
20
21 public class MyTestBean {
22       private String name;
23
24       private Date date;
25
26       public void doSomething () {
27           System.out.println("create my bean, date: " + date+" , with name : "+name);
28       }
29
30       public void setName(String name) {
31           this.name=name;
32       }
33
34       public void setDate (Date date) {
35           this.date = date;
36       }
37   }
38
```

使用最简单的测试类：

```java
import java.util.Date;

import org.springframework.beans.MutablePropertyValues;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.GenericBeanDefinition;

public class GenericBeanDefinitionExample {

    public static void main (String[] args) {
        DefaultListableBeanFactory context =new DefaultListableBeanFactory();

        GenericBeanDefinition gbd = new GenericBeanDefinition();
        gbd.setBeanClass(MyTestBean.class);

        MutablePropertyValues mpv = new MutablePropertyValues();
        mpv.add("date", new Date());
        mpv.add("name", "myTestBean");

        gbd.setPropertyValues(mpv);

        context.registerBeanDefinition("myTestBean", gbd);

        MyTestBean bean = context.getBean(MyTestBean.class);
        bean.doSomething();
    }
}

```

说明：

GenericBeanDefinition 是一站式定义标准 Bean 的类；

MutablePropertyValues 用来操作属性，支持深度从 Map 从拷贝属性。

通过 GenericBeanDefinition 构造一个 Bean，然后通过 MutablePropertyValues 来设置该 Bean 的属性，最后注册该 Bean 到 IoC 容器 DefaultListableBeanFactory，构成一个最小完整的生命周期过程。

深入 Spring IoC 容器 Bean 生命周期源码

放大招了，调试源码的技巧：

SimpleInstantiationStrategy#instantiate() 方法，抛出异常调用链：

```java
@Override
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner) {
    // Don't override the class with CGLIB if no overrides.
    if (!bd.hasMethodOverrides()) {
        Constructor<?> constructorToUse;
        synchronized (bd.constructorArgumentLock) {
            constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;
            if (constructorToUse == null) {
                final Class<?> clazz = bd.getBeanClass();
                if (clazz.isInterface()) {
                    throw new BeanInstantiationException(clazz, "Specified class is an interface");
                }
                try {
                    if (System.getSecurityManager() != null) {
                        constructorToUse = AccessController.doPrivileged(
                                (PrivilegedExceptionAction<Constructor<?>>) clazz::getDeclaredConstructor);
                    }
                    else {
                        constructorToUse = clazz.getDeclaredConstructor();
                    }
                    bd.resolvedConstructorOrFactoryMethod = constructorToUse;
                }
                catch (Throwable ex) {
                    throw new BeanInstantiationException(clazz, "No default constructor found", ex);
                }
            }
        }
        throw new NullPointerException("use constructor to create instance");
        //return BeanUtils.instantiateClass(constructorToUse);
    }
    else {
        // Must generate CGLIB subclass.
        throw new NullPointerException("use constructor to create instance");
        //return instantiateWithMethodInjection(bd, beanName, owner);
    }
}
```

上面两处打上断点（可选），Debug：

```
25 public class GenericBeanDefinitionExample {
26
27     public static void main (String[] args) {
28         DefaultListableBeanFactory context =new DefaultListableBeanFactc
29
30         GenericBeanDefinition gbd = new GenericBeanDefinition();
31         gbd.setBeanClass(MyTestBean.class);
32
33         MutablePropertyValues mpv = new MutablePropertyValues();
34         mpv.add("date", new Date());
35         mpv.add("name", "myTestBean");
36
37         gbd.setPropertyValues(mpv);
38
39         context.registerBeanDefinition("myTestBean", gbd);
40
41         MyTestBean bean = context.getBean(MyTestBean.class);
42         bean.doSomething();
43     }
44 }
```

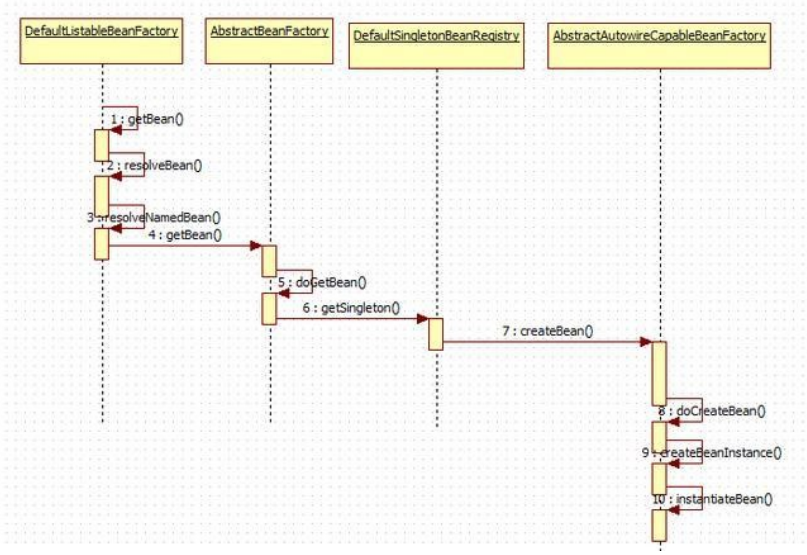
在 registerBeanDefinition 的时候并没有生成实例 instance，而是在 getBean 的时候才生成 instance，即延迟加载。

```
Exception in thread "main" org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'myTestBean': Instantiation of bean failed; nested exception is java.lang.NullPointerException: use constructor to create instance
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.instantiateBean(AbstractAutowireCapableBeanFactory.java:1319)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBeanInstance(AbstractAutowireCapableBeanFactory.java:1214)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:557)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:517)
        at org.springframework.beans.factory.support.AbstractBeanFactory.lambda$0(AbstractBeanFactory.java:323)
        at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:223)
        at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:321)
        at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:227)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveNamedBean(DefaultListableBeanFactory.java:1155)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveBean(DefaultListableBeanFactory.java:416)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:349)
        at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
        at com.davidwang456.beans.GenericBeanDefinitionExample.main(GenericBeanDefinitionExample.java:41)
Caused by: java.lang.NullPointerException: use constructor to create instance
        at org.springframework.beans.factory.support.SimpleInstantiationStrategy.instantiate(SimpleInstantiationStrategy.java:87)
        at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.instantiateBean(AbstractAutowireCapableBeanFactory.java:1312)
        ... 12 more
```

> Instantiation of bean failed; nested exception is java.lang.NullPointerException: use constructor to create instance

我们修改源码后抛出的异常，走的是构造器生成实例。

时序图如下：



AbstractAutowireCapableBeanFactory#createBean()。

创建 **Bean** 之前的处理器

AbstractAutowireCapableBeanFactory#resolveBeforeInstantiation。

```java
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
        throws BeanCreationException {

    if (logger.isTraceEnabled()) {
        logger.trace("Creating instance of bean '" + beanName + "'");
    }
    RootBeanDefinition mbdToUse = mbd;

    // Make sure bean class is actually resolved at this point, and
    // clone the bean definition in case of a dynamically resolved Class
    // which cannot be stored in the shared merged bean definition.
    Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
    if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
        mbdToUse = new RootBeanDefinition(mbd);
        mbdToUse.setBeanClass(resolvedClass);
    }

    // Prepare method overrides.
    try {
        mbdToUse.prepareMethodOverrides();
    }
    catch (BeanDefinitionValidationException ex) {
        throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
                beanName, "Validation of method overrides failed", ex);
    }

    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
                "BeanPostProcessor before instantiation of bean failed", ex);
    }

    try {
        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isTraceEnabled()) {
            logger.trace("Finished creating instance of bean '" + beanName + "'");
        }
        return beanInstance;
    }
    catch (BeanCreationException | ImplicitlyAppearedSingletonException ex) {
        // A previously detected exception with proper bean creation context already,
        // or illegal singleton state to be communicated up to DefaultSingletonBeanRegistry.
        throw ex;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
                mbdToUse.getResourceDescription(), beanName, "Unexpected exception during bean creation", ex);
    }
}
```

InstantiationAwareBeanPostProcessor#postProcessBeforeInitialization。

InstantiationAwareBeanPostProcessor#postProcessAfterInitialization。

创建 **Bean** 的处理逻辑

创建 Bean 的过程在 AbstractAutowireCapableBeanFactory#doCreateBean() 方法，主要有

创建 Bean 的实例，注册 PropertyEditorRegistry；

BeanPostProcessor 处理，这一阶段常用的注解被处理如 @Autowired、@Resource、@WebServiceRef、@EJB、@Required、@Scheduled 等。

将要用的 Bean 提前注入到容器 BeanFactory 中，如创建 AOP 代理的类 DefaultAdvisorAutoProxyCreator，BeanNameAutoProxyCreator 等。

设置 Bean 属性；

注册 DisposableBean。

创建 **Bean** 实例

AbstractAutowireCapableBeanFactory#createBeanInstance。

1. 创建 Bean：

```
beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
```

2. 使用 BeanWrapper 包装：registerCustomEditors() 注册 PropertyEditorRegistry。

BeanPostProcessor 处理

AbstractAutowireCapableBeanFactory#applyMergedBeanDefinitionPostProcessors：

MergedBeanDefinitionPostProcessor#postProcessMergedBeanDefinition。

```
// Allow post-processors to modify the merged bean definition.
synchronized (mbd.postProcessingLock) {
    if (!mbd.postProcessed) {
        try {
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Post-processing of merged bean definition failed", ex);
        }
        mbd.postProcessed = true;
    }
}
```

将要用到的 Bean 提前注册到 BeanFactory

```
// Eagerly cache singletons to be able to resolve circular references
// even when triggered by lifecycle interfaces like BeanFactoryAware.
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
        isSingletonCurrentlyInCreation(beanName));
if (earlySingletonExposure) {
    if (logger.isTraceEnabled()) {
        logger.trace("Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
    }
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}
```

初始化 Bean

AbstractAutowireCapableBeanFactory#initializeBean：

```
// Initialize the bean instance.
Object exposedObject = bean;
try {
    populateBean(beanName, mbd, instanceWrapper);
    exposedObject = initializeBean(beanName, exposedObject, mbd);
}
catch (Throwable ex) {
    if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
        throw (BeanCreationException) ex;
    }
    else {
        throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
    }
}

if (earlySingletonExposure) {
    Object earlySingletonReference = getSingleton(beanName, false);
    if (earlySingletonReference != null) {
        if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
        }
        else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);
            for (String dependentBean : dependentBeans) {
                if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                    actualDependentBeans.add(dependentBean);
                }
            }
            if (!actualDependentBeans.isEmpty()) {
                throw new BeanCurrentlyInCreationException(beanName,
                        "Bean with name '" + beanName + "' has been injected into other beans [" +
                        StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                        "] in its raw version as part of a circular reference, but has eventually been " +
                        "wrapped. This means that said other beans do not use the final version of the " +
                        "bean. This is often the result of over-eager type matching - consider using " +
                        "'getBeanNamesForType' with the 'allowEagerInit' flag turned off, for example.");
            }
        }
    }
}
```

1. populateBean

第一阶段： InstantiationAwareBeanPostProcessor#postProcessAfterInstantiation；

第二阶段：InstantiationAwareBeanPostProcessor#postProcessProperties；

第三阶段：设置属性，BeanDefinitionValueResolver。

2. initializeBean

第一阶段：BeanNameAware，BeanClassLoaderAware，BeanFactoryAware；

第二阶段：BeanPostProcessor#postProcessBeforeInitialization；

第三阶段：invokeInitMethods，InitializingBean.afterPropertiesSet()，执行自定义的 init() 方法；

第四阶段：BeanPostProcessor#applyBeanPostProcessorsAfterInitialization。

注册 DisposableBean

```
/**
 * Add the given bean to the list of disposable beans in this factory,
 * registering its DisposableBean interface and/or the given destroy method
 * to be called on factory shutdown (if applicable). Only applies to singletons.
 * @param beanName the name of the bean
 * @param bean the bean instance
 * @param mbd the bean definition for the bean
 * @see RootBeanDefinition#isSingleton
 * @see RootBeanDefinition#getDependsOn
 * @see #registerDisposableBean
 * @see #registerDependentBean
 */
protected void registerDisposableBeanIfNecessary(String beanName, Object bean, RootBeanDefinition mbd) {
    AccessControlContext acc = (System.getSecurityManager() != null ? getAccessControlContext() : null);
    if (!mbd.isPrototype() && requiresDestruction(bean, mbd)) {
        if (mbd.isSingleton()) {
            // Register a DisposableBean implementation that performs all destruction
            // work for the given bean: DestructionAwareBeanPostProcessors,
            // DisposableBean interface, custom destroy method.
            registerDisposableBean(beanName,
                    new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(), acc));
        }
        else {
            // A bean with a custom scope...
            Scope scope = this.scopes.get(mbd.getScope());
            if (scope == null) {
                throw new IllegalStateException("No Scope registered for scope name '" + mbd.getScope() + "'");
            }
            scope.registerDestructionCallback(beanName,
                    new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(), acc));
        }
    }
}
```

总结

Bean 的生命周期是 Spring 的重点难点，这方面的材料网上汗牛充栋，我从网上拿到一张比较接近的图片。

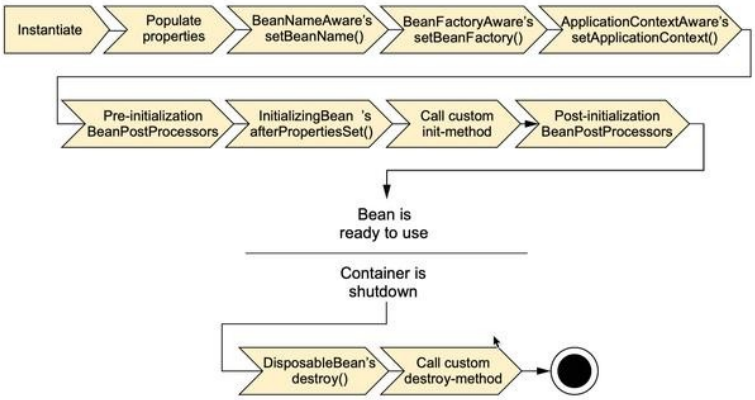


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

从上图可以看到，Spring Bean 的完整生命周期从创建 Spring 容器开始，直到最终 Spring 容器销毁 Bean，这其中包含了一系列关键点。

注意：FactoryBean 和 BeanFactory

> “factory bean” refers to a bean that is configured in the Spring container and that creates objects through an instance or static factory method.

简单的说，FactoryBean 是一个特殊的 Bean，它由 Spring 容器（如 BeanFactory 或者 ApplicationContext）管理，它通过静态工厂方法和实例来创建一个对象。

}

12 @Autowired是如何工作的？ -- Spring注解源码深度揭秘

14 离开了Spring，IoC容器还可以怎么做？