

【第31话：Spring面试题又来了，Bean的生命周期】

小伙伴们，这节课给大家带来一个“Spring框架中Bean生命周期”这个面试题。

对于这个问题其实并不是特别好记忆，所以我们先上图



既然是生命周期，也就是Bean从实例化一直到销毁的整个流程。上图就是整个Bean生命周期的流程图。

为了能够让小伙伴们更加清晰的理解到Bean的生命周期，我们上代码，来具体演示下Bean的生命周期。

既然是演示Bean的生命周期，最开始肯定要准备一个类。我们就拿万能的People类进行演示。类中随意提供了String name属性，这个属性主要是为了演示设值注入的执行顺序。同时还提供了无参构造方法及Getter/Setter方法。并在里面提供了输出语句，以便观察执行顺序。toString()方法也是不能少的，为了能够查看设置注入是否成功。

```
package com.bjsxt.pojo;

public class People {
    private String name;

    public People() {
        System.out.println("执行构造方法");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("执行setter方法");
        this.name = name;
    }

    @Override
    public String toString() {
        return "People{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

然后新建一个Spring的配置文件applicationContext.xml,并配置People的Bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd" >
    <bean id="peo" class="com.bjsxt.pojo.People">
        <property name="name" value="张三"></property>
    </bean>
</beans>
```

接下来新建测试类，获取peo这个Bean，并打印到控制台上。

```

package com.bjsxt.test;

import com.bjsxt.pojo.People;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        People people = applicationContext.getBean("peo", People.class);
        System.out.println(people);
    }
}

```

运行测试类，可以看到控制台输出的执行顺序是先执行构造方法，后执行setter方法。

执行构造方法

执行setter方法

People{name='张三'}

接下来会调用Spring提供的包装接口Aware。Aware接口有很多子接口。

```

* Aware (org.springframework.beans.factory)
> ApplicationEventPublisherAware (org.springframework.context)
> MessageSourceAware (org.springframework.context)
> ResourceLoaderAware (org.springframework.context)
> ApplicationStartupAware (org.springframework.context)
> NotificationPublisherAware (org.springframework.jmx.export.notification)
> BeanFactoryAware (org.springframework.beans.factory)
> EnvironmentAware (org.springframework.context)
> EmbeddedValueResolverAware (org.springframework.context)
> ImportAware (org.springframework.context.annotation)
> LoadTimeWeaverAware (org.springframework.context.weaving)
> BeanNameAware (org.springframework.beans.factory)
> BeanClassLoaderAware (org.springframework.beans.factory)
> ApplicationContextAware (org.springframework.context)

```

我们带领小伙伴们使用里面偶尔能使用到的几个子接口：BeanNameAware、BeanFactoryAware、ApplicationContextAware

一个一个的演示。先看BeanNameAware。接口中只有一个setBeanName(String)方法，参数表示Bean的id值。

```

package org.springframework.beans.factory;

public interface BeanNameAware extends Aware {
    void setBeanName(String var1);
}

```

使用时只需要让People实现这个接口后重写接口中方法就可以了。为了查看执行顺序，在setBeanName方法中输出一句话。

```
package com.bjsxt.pojo;

import org.springframework.beans.factory.BeanNameAware;

public class People implements BeanNameAware {
    private String name;

    public People() {
        System.out.println("执行构造方法");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("执行setter方法");
        this.name = name;
    }

    @Override
    public String toString() {
        return "People{" +
            "name='" + name + '\'' +
            '}';
    }

    @Override
    public void setBeanName(String s) {
        System.out.println("setBeanName:"+s);
    }
}
```

运行测试类后，根据IDEA控制台输出的结果可以看到BeanNameAware接口中方法setBeanName(String)是在设值注入后被调用。而且是自动被调用的。

IDEA控制台的执行结果如下：

```
执行构造方法
执行setter方法
setBeanName:peo
People{name='张三'}
```

下面继续演示BeanFactoryAware的效果。先看看BeanFactoryAware接口的定义。里面只有一个setBeanFactory的方法，参数为BeanFactory的具体实例DefaultListableBeanFactory对象。

```
public interface BeanFactoryAware extends Aware {
    void setBeanFactory(BeanFactory var1) throws BeansException;
}
```

使用的时候让People实现BeanFactoryAware接口，并重写setBeanFactory方法，为了观察执行顺序，在方法中添加了输出语句

```
package com.bjsxt.pojo;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;

public class People implements BeanNameAware, BeanFactoryAware {
    private String name;

    public People() {
        System.out.println("执行构造方法");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("执行setter方法");
        this.name = name;
    }

    @Override
    public String toString() {
        return "People{" +
            "name='" + name + '\'' +
            '}';
    }

    @Override
    public void setBeanName(String s) {
        System.out.println("setBeanName:"+s);
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("setBeanFactory"+beanFactory);
    }
}
```

运行测试类可以看到先执行BeanNameAware，然后执行BeanFactoryAware的方法。

```
执行构造方法
执行setter方法
setBeanName:peo
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo]; root of factory hierarchy
People{name='张三'}
```

这时有的小伙伴可能有疑问了：BeanNameAware和BeanFactoryAware默认就是按照固定的执行顺序执行，还是和实现接口时的顺序有关系呢？

我们做个小实验，让People实现接口时先实现BeanFactoryAware然后实现BeanNameAware，同时为了测试的更彻底，把setBeanFactory方法放在setBeanName方法上面。

修改后的代码如下：

```
package com.bjsxt.pojo;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;

public class People implements BeanFactoryAware, BeanNameAware {
    private String name;

    public People() {
        System.out.println("执行构造方法");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("执行setter方法");
        this.name = name;
    }

    @Override
    public String toString() {
        return "People{" +
            "name='" + name + '\'' +
            '}';
    }

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

    @Override
    public void setBeanName(String s) {
        System.out.println("setBeanName:"+s);
    }
}
```

再次运行测试类，可以看到IDEA控制台输出的内容和之前一样，执行顺序并没有变化。这也说明了BeanNameAware和BeanFactoryAware就是按照固定的顺序执行，先执行BeanNameAware后执行BeanFactoryAware

```
执行构造方法
执行setter方法
setBeanName: peo
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo]; root of factory hierarchy
People{name='张三'}
```

下面我们测试最后一个Aware接口ApplicationContextAware。还是先看一下接口的定义。里面只有一个setApplicationContext方法。方法参数是ApplicationContext接口的具体实现类ClassPathXmlApplicationContext的实例。这个实例也是我们在测试类中代码创建的实例。

```

public interface ApplicationContextAware extends Aware {
    void setApplicationContext(ApplicationContext applicationContext) throws BeansException;
}

```

使用ApplicationContextAware的方式和上面一样，都是让People实现这个接口，并重写接口中方法。为了看到执行顺序，还是需要在方法中添加输出语句。

```

package com.bjsxt.pojo;

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

public class People implements BeanFactoryAware, BeanNameAware,
ApplicationContextAware {
    private String name;

    public People() {
        System.out.println("执行构造方法");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("执行setter方法");
        this.name = name;
    }

    @Override
    public String toString() {
        return "People{" +
            "name='" + name + '\'' +
            '}';
    }

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

    @Override
    public void setBeanName(String s) {
        System.out.println("setBeanName:"+s);
    }

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

```

```
}  
}
```

运行测试类，观察IDEA控制台可以看到执行完BeanNameAware的方法后执行ApplicationContextAware。这三个接口的执行顺序正好和我们的演示顺序是相同的。

```
执行构造方法  
执行setter方法  
setBeanName:peo  
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo]; root of factory hierarchy  
setApplicationContext:org.springframework.context.support.ClassPathXmlApplicationContext@312b1dae, started on Tue May 24 14:33:43 CST 2022  
People{name='张三'}
```

Aware接口我们重点给小伙伴们展示了三个子接口，有兴趣的小伙伴也可以按照上面的步骤查看其它子接口，套路是一样一样的。

下面我们继续演示，演示下一组对应的接口分别是InitializingBean代表初始化Bean时执行的方法和DisposableBean代表销毁Bean时执行的方法。这两个接口中都包含一个方法。

InitializingBean的定义如下：

```
public interface InitializingBean {  
    void afterPropertiesSet() throws Exception;  
}
```

DisposableBean的定义如下：

```
public interface DisposableBean {  
    void destroy() throws Exception;  
}
```

所以依然是让People实现这两个接口，并重写方法就可以了。为了演示执行顺序，还是在两个方法中都添加了输出语句。People类修改后的代码如下：

```
package com.bjsxt.pojo;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.*;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ApplicationContextAware;  
  
public class People implements BeanFactoryAware, BeanNameAware,  
    ApplicationContextAware, InitializingBean, DisposableBean {  
    private String name;  
  
    public People() {  
        System.out.println("执行构造方法");  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        System.out.println("执行setter方法");  
        this.name = name;  
    }  
}
```



```

@Override
public String toString() {
    return "People{" +
        "name='" + name + '\'' +
        '}';
}

@Override
public void setBeanFactory(BeansFactory beanFactory) throws BeansException {
    System.out.println("setBeanFactory"+beanFactory);
}

@Override
public void setBeanName(String s) {
    System.out.println("setBeanName:"+s);
}

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

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

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

```

运行测试类，查看控制台可以看到InitializingBean的afterPropertiesSet()方法是在Aware接口执行完成后执行的。

```

执行构造方法
执行setter方法
setBeanName:peo
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo]; root of factory hierarchy
setApplicationContext:org.springframework.context.support.ClassPathXmlApplicationContext@312b1dae, started on Tue May 24 14:53:50 CST 2022
afterPropertiesSet
People{name='张三'}

```

但是DisposableBean的destroy()方法并没有看到执行。这又是为什么呢？这是因为scope="singleton"的Bean生命周期随着Spring配置文件加载而创建，只要不销毁容器或者不刷新容器(refresh),Bean是不会被销毁的。我们在测试类的代码是获取到Bean对象后程序结束了，并没有销毁容器，所以不会打印destroy()中的内容。为了演示效果，只需要在测试类中销毁（close()）/刷新(refresh())一下容器就可以了。

小伙伴们在改测试类的代码时需要注意，close()方法在ApplicationContext接口中没有的，根据多态特性。接收容器对象时还是ClassPathXmlApplicationContext进行接收。

```

package com.bjsxt.test;

import com.bjsxt.pojo.People;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        People people = applicationContext.getBean("peo", People.class);
        System.out.println(people);
        applicationContext.close();
    }
}

```

修改后再次运行测试类，可以看到DisposableBean的destroy()方法被执行了。而且是最后执行的。

```

执行构造方法
执行setter方法
setBeanName:peo
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo]; root of factory hierarchy
setApplicationContext:org.springframework.context.support.ClassPathXmlApplicationContext@312b1dae, started on Tue May 24 15:01:22 CST 2022
afterPropertiesSet
People{name='张三'}
destroy

```

在Spring框架之所以被Java程序员所喜爱，就是因为Spring的强大，并且Spring时时刻刻考虑我们开发者。Spring不仅仅提供了实现接口完成初始化/销毁Bean的功能。还提供了一种允许程序员自定义方法的方式。

我们现在People类中添加两个自定义方法，分别是myinit()代表初始化方法，mydestory()代表销毁方法。这两个方法名称没有要求可以随意定义，方法返回值和访问权限修饰符也没有要求。示例代码中都设置为public void的方法了。

```

package com.bjsxt.pojo;

import org.springframework.beans.BeanException;
import org.springframework.beans.factory.*;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class People implements BeanFactoryAware, BeanNameAware,
ApplicationContextAware, InitializingBean, DisposableBean {
    private String name;

    public People() {
        System.out.println("执行构造方法");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("执行setter方法");
        this.name = name;
    }
}

```

```

@Override
public String toString() {
    return "People{" +
        "name='" + name + '\'' +
        '}';
}

@Override
public void setBeanFactory(BeansFactory beanFactory) throws BeansException {
    System.out.println("setBeanFactory"+beanFactory);
}

@Override
public void setBeanName(String s) {
    System.out.println("setBeanName:"+s);
}

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

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

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

public void myinit(){
    System.out.println("myinit");
}

public void mydestroy(){
    System.out.println("mydestroy");
}
}

```

然后在配置文件中通过 <bean> 的 init-method 和 destroy-method 两个属性指定出方法名。其中 init-method 表示初始化的方法，destroy-method 表示销毁的方法

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd" >
    <bean id="peo" class="com.bjsxt.pojo.People" init-method="myinit" destroy-
method="mydestroy">
        <property name="name" value="张三"></property>
    </bean>
</beans>
```

运行测试类可以发现这两个方法都是在基于接口方式后面执行。

```
执行构造方法
执行setter方法
setBeanName:peo
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo]; root of factory hierarchy
setApplicationContext:org.springframework.context.support.ClassPathXmlApplicationContext@312b1dae, started on Tue May 24 15:26:37 CST 2022
afterPropertiesSet
myinit
People{name='张三'}
destroy
mydestroy
```

好了，最后在演示一下Spring框架提供的一个增强处理接口BeanPostProcessor。在这个接口中有两个方法，这两个方法的名称很像，主要的区别是一个方法名中是Before，另一个是After。他们的执行顺序是postProcessBeforeInitialization是在初始化方法之前执行，postProcessAfterInitialization是在销毁方法之前执行。

两个方法的参数分别是：bean代表操作的Bean对象，beanName代表Bean的名称。

```
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;
    }

}
```

使用BeanPostProcessor时，并不是让People实现这个接口。需要单独创建一个类让类实现接口，并在Spring的配置文件中配置这个类的Bean

先新建一个类

```
package com.bjsxt.processor;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("postProcessBeforeInitialization");
        return null;
    }

    @Override
```

```

    public Object postProcessAfterInitialization(Object bean, String beanName)
    throws BeansException {
        System.out.println("postProcessAfterInitialization");
        return null;
    }
}

```

然后在Spring配置文件中配置MyBeanPostProcessor的Bean

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd" >
    <bean id="peo" class="com.bjsxt.pojo.People" init-method="myinit" destroy-
method="mydestroy" >
        <property name="name" value="张三"></property>
    </bean>
    <bean class="com.bjsxt.processor.MyBeanPostProcessor"></bean>
</beans>

```

运行测试类可以发现BeanPostProcessor的postProcessBeforeInitialization在初始化之前执行，postProcessAfterInitialization在销毁之前执行。

```

执行构造方法
执行setter方法
setBeanName:peo
setBeanFactoryorg.springframework.beans.factory.support.DefaultListableBeanFactory@79b4d0f: defining beans [peo,com.bjsxt.processor.MyBeanPostProcessor#0]; root of factory hierarchy
setApplicationContext:org.springframework.context.support.ClassPathXmlApplicationContext@312b1dae, started on Tue May 24 15:47:00 CST 2022
postProcessBeforeInitialization
afterPropertiesSet
myinit
postProcessAfterInitialization
People{name='张三'}
destroy
mydestroy

```

这个输出过程就是Spring Bean的生命周期中涉及方法的执行流程。