朱晔和你聊 Spring 系列 S1E3：Spring 咖啡罐里的豆子

标题中的咖啡罐指的是 Spring 容器，容器里装的当然就是被称作 Bean 的豆子。本文我们会以一个最基本的例子来熟悉 Spring 的容器管理和扩展点。

为什么要让容器来管理对象？

首先我们来聊聊这个问题，为什么我们要用 Spring 来管理对象（的生命周期和对象之间的关系）而不是自己 new 一个对象呢？大家可能会回答是方便，为了解耦。我个人觉得除了这两个原因之外，还有就是给予了我们更多可能性。如果我们以容器为依托来管理所有的框架、业务对象，那么不仅仅我们可以无侵入调整对象的关系，还有可能无侵入随时调整对象的属性甚至悄悄进行对象的替换。这就给了我们无限多的可能性，大大方便了框架的开发者在程序背后实现一些扩展。不仅仅 Spring Core 本身以及 Spring Boot 大量依赖 Spring 这套容器体系，一些外部框架也因为这个原因可以和 Spring 进行无缝整合。

Spring 可以有三种方式来配置 Bean，分别是最早期的 XML 方式、后来的注解方式以及现在最流行的 Java 代码配置方式。

Bean 的回调事件

在前文 parent 模块的基础上，我们先来创建一个 beans 模块：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>me.josephzhu</groupId>
    <artifactId>spring101-beans</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring101-beans</name>
    <description></description>
```

```
    <parent>
        <groupId>me.josephzhu</groupId>
        <artifactId>spring101</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

然后来创建我们的豆子：

```
package me.josephzhu.spring101beans;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class MyService implements InitializingBean, DisposableBean {

    public int increaseCounter() {
        this.counter++;
        return counter;
    }

    public int getCounter() {
        return counter;
    }

    public void setCounter(int counter) {
        this.counter = counter;
    }
```

```
    private int counter=0;

    public MyService(){
        counter++;
        System.out.println(this + "#constructor:" + counter);
    }

    public String hello(){
        return this + "#hello:" + counter;
    }

    @PreDestroy
    public void preDestroy() {
        System.out.println(this + "#preDestroy:" + counter);

    }

    @Override
    public void afterPropertiesSet() {
        counter++;
        System.out.println(this + "#afterPropertiesSet:" + counter);
    }

    @PostConstruct
    public void postConstruct(){
        counter++;
        System.out.println(this + "#postConstruct:" + counter);
    }

    @Override
    public void destroy() {
        System.out.println(this + "#destroy:" + counter);

    }
}
```

这里可以看到，我们的服务中有一个 counter 字段，默认是 0。这个类我们实现了 InitializingBean 接口和 DisposableBean 接口，同时还创建了两个方法分别加上了 @PostConstruct 和@PreDestroy 注解。这两套实现方式都可以在对象的额外初始化功能和释放功能，注解的实现不依赖 Spring 的接口，侵入性弱一点。

接下去，我们创建一个 Main 类来测试一下：

```
package me.josephzhu.spring101beans;

import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

import javax.annotation.Resource;

@SpringBootApplication
public class Spring101BeansApplication implements CommandLineRunner {

    @Autowired
    private ApplicationContext applicationContext;
    @Resource
    private MyService helloService;
    @Autowired
    private MyService service;

    public static void main(String[] args) {
        SpringApplication.run(Spring101BeansApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("====================");
        applicationContext.getBeansOfType(MyService.class).forEach((name, service)->{
            System.out.println(name + ":" + service);
        });

        System.out.println("====================");
        System.out.println(helloService.hello());
        System.out.println(service.hello());

    }
}
```

ApplicationContext 直接注入即可，不一定需要用 ApplicationContextAware 方式来获取。

执行程序后可以看到输出如下：

```
me.josephzhu.spring101beans.MyService@7fb4f2a9#constructor:1
me.josephzhu.spring101beans.MyService@7fb4f2a9#postConstruct:2
me.josephzhu.spring101beans.MyService@7fb4f2a9#afterPropertiesSet:3
====================
myService:me.josephzhu.spring101beans.MyService@7fb4f2a9
====================
me.josephzhu.spring101beans.MyService@7fb4f2a9#hello:3
me.josephzhu.spring101beans.MyService@7fb4f2a9#hello:3
```

```
me.josephzhu.spring101beans.MyService@7fb4f2a9#preDestroy:3
me.josephzhu.spring101beans.MyService@7fb4f2a9#destroy:3
```

这里我们使用@Resource 注解和@Autowired 注解分别引用了两次对象，可以看到由于 Bean 默认配置为 singleton 单例，所以容器中 MyService 类型的对象只有一份，代码输出也可以证明这点。此外，我们也通过输出看到了构造方法以及两套 Bean 回调的次序是：

1. 类自己的构造方法
2. @PostConstruct 注释的方法
3. InitializingBean 接口实现的方法
4. @PreDestroy 注释的方法
5. DisposableBean 接口实现的方法

Java 代码方式创建 Bean

从刚才的输出中可以看到，在刚才的例子中，我们为 Bean 打上了@Component 注解，容器为我们创建了名为 myService 的 MyService 类型的 Bean。现在我们再来用 Java 代码方式来创建相同类型的 Bean，创建如下的文件：

```
package me.josephzhu.spring101beans;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
public class ApplicationConfig {

    @Bean(initMethod = "init")
    public MyService helloService(){
        MyService myService = new MyService();
        myService.increaseCounter();
        return myService;
    }

}
```

这里可以看到在定义 Bean 的时候我们关联了一个 initMethod，因此我们需要修改 Bean 加上这个方法：

```
public void init() {
    counter++;
    System.out.println(this + "#init:" + counter);

}
```

现在我们运行代码看看结果，得到了如下错误：

Field service in me.josephzhu.spring101beans.Spring101BeansApplication required a single bean, but 2 were found:

- myService: defined in file [/Users/zyhome/IdeaProjects/spring101/spring101-beans/target/classes/me/josephzhu/spring101beans/MyService.class]

- helloService: defined by method 'helloService' in class path resource [me/josephzhu/spring101beans/ApplicationConfig.class]

出现错误的原因是@Autowired 了一个 MyService，@Resource 注解因为使用 Bean 的名称来查找 Bean，所以并不会出错，而@Autowired 因为根据 Bean 的类型来查抄 Bean 找到了两个匹配所有出错了，解决方式很简单，我们在多个 Bean 里选一个作为主 Bean。我们修改一下 MyService 加上注解：

```
@Component
@Primary
public class MyService implements InitializingBean, DisposableBean
```

这样，我们的@Resource 根据名字匹配到的是我们@Configuration 出来的 Bean，而@Autowired 根据类型+Primary 匹配到了@Component 注解定义的 Bean，重新运行代码来看看是不是这样：

```
me.josephzhu.spring101beans.MyService@6cd24612#constructor:1

me.josephzhu.spring101beans.MyService@6cd24612#postConstruct:3

me.josephzhu.spring101beans.MyService@6cd24612#afterPropertiesSet:4

me.josephzhu.spring101beans.MyService@6cd24612#init:5

me.josephzhu.spring101beans.MyService@7486b455#constructor:1

me.josephzhu.spring101beans.MyService@7486b455#postConstruct:2

me.josephzhu.spring101beans.MyService@7486b455#afterPropertiesSet:3

====================

myService:me.josephzhu.spring101beans.MyService@7486b455

helloService:me.josephzhu.spring101beans.MyService@6cd24612

====================

me.josephzhu.spring101beans.MyService@6cd24612#hello:5

me.josephzhu.spring101beans.MyService@7486b455#hello:3

me.josephzhu.spring101beans.MyService@7486b455#preDestroy:3
```

```
me.josephzhu.spring101beans.MyService@7486b455#destroy:3

me.josephzhu.spring101beans.MyService@6cd24612#preDestroy:5

me.josephzhu.spring101beans.MyService@6cd24612#destroy:5
```

从输出中我们注意到几点：

1. 先输出的的确是 helloService，说明@Resource 引入的是我们 Java 代码配置的 MyService，helloService 由于在我们配置的多调用了一次 increaseCounter()以及关联的 initMethod，所以 counter 的值是 5
2. initMethod 执行的顺序在@PostConstruct 注释的方法和 InitializingBean 接口实现的方法之后
3. 虽然我们的 MySerive 的两种 Bean 的定义都是单例，但是这不代表我们的 Bean 就是一套，在这里我们通过代码配置和注解方式在容器内创建了两套 MyService 类型的 Bean，它们都经历了自己的初始化过程。通过@Resource 和@Autowired 引入到了是不同的 Bean，当然也就是不同的对象

你还可以试试在使用@Autowired 引入 MyService 的时候直接指定需要的 Bean：

```
@Autowired
@Qualifier("helloService")
private MyService service;
```

两个重要的扩展点

我们来继续探索 Spring 容器提供给我们的两个有关 Bean 的重要扩展点。

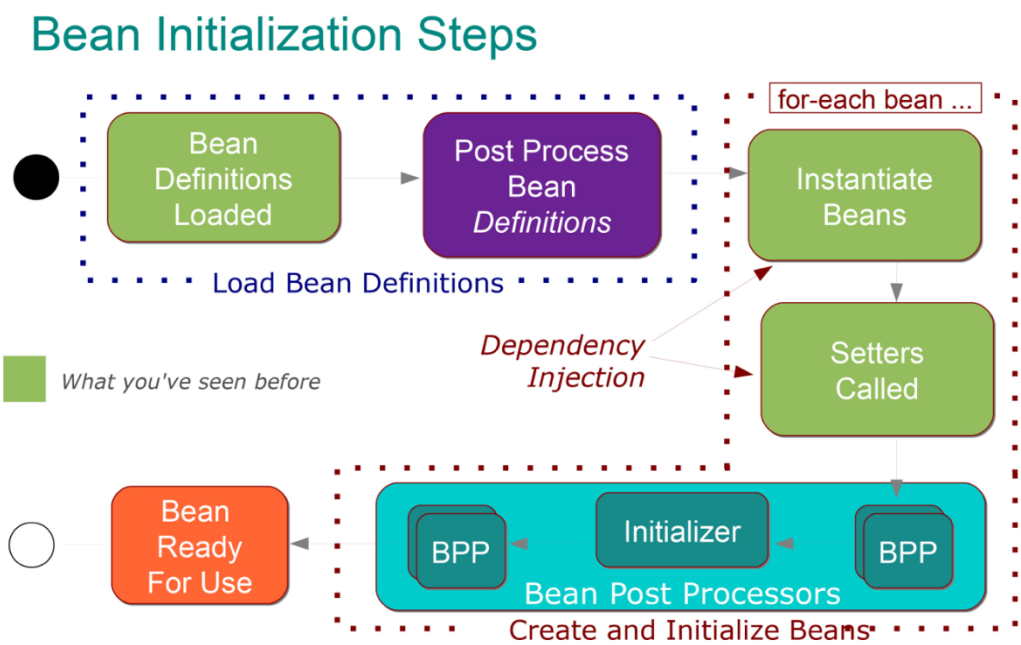
- 用于修改 Bean **定义**的 BeanFactoryPostProcessor。所谓修改定义就是修改 Bean 的元数据，元数据有哪些呢？如下图所示，类型、名字、实例化方式、构造参数、属性、Autowire 模式、懒初始化模式、初始析构方法。实现了这个接口后，我们就可以修改这些已经定义的元数据，实现真正的动态配置。这里需要注意，我们不应该在这个接口的实现中去实例化 Bean，否则这相当于提前进行了实例化会破坏 Bean 的生命周期。

Table 1. The bean definition

Property	Explained in…
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

- 用于修改 Bean **实例**的 BeanPostProcessor。在这个阶段其实 Bean 已经实例化了，我们可以进行一些额外的操作对 Bean 进行修改。如下图，我们可以清晰的看到 Bean 的生命周期如下（BeanPostProcessor 缩写为 BPP）：

1. Bean 定义加载
2. BeanFactoryPostProcessor 来修改 Bean 定义
3. Bean 逐一实例化
4. BeanPostProcessor 预处理
5. Bean 初始化
6. BeanPostProcessor 后处理



好，我们现在来实现这两种类型的处理器，首先是用于修改 Bean 定义的处理器：

```
package me.josephzhu.spring101beans;
```

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.stereotype.Component;

@Component
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
configurableListableBeanFactory) throws BeansException {
        BeanDefinition beanDefinition =
configurableListableBeanFactory.getBeanDefinition("helloService");
        if (beanDefinition != null) {
            beanDefinition.setScope("prototype");
            beanDefinition.getPropertyValues().add("counter", 10);
        }
        System.out.println("MyBeanFactoryPostProcessor");
    }
}
```

这里，我们首先找到了我们的 helloService（Java 代码配置的那个 Bean），然后修改了它的属性和 Scope（还记得吗，在之前的图中我们可以看到，这两项都是 Bean 的定义，定义相当于类描述，实例当然就是类实例了）。

然后，我们再来创建一个修改 Bean 实例的处理器：

```
package me.josephzhu.spring101beans;

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

@Component
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        if (bean instanceof MyService) {
            System.out.println(bean + "#postProcessAfterInitialization:" +
((MyService)bean).increaseCounter());
        }
        return bean;
    }

    @Override
```

```
   public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
       if (bean instanceof MyService) {
          System.out.println(bean + "#postProcessBeforeInitialization:" +
((MyService)bean).increaseCounter());
       }
       return bean;
   }
}
```

实现比较简单，在这个处理器的两个接口我们都调用了一次增加计数器的操作。我们运行代码来看一下这两个处理器执行的顺序是否符合刚才那个图的预期：

```
MyBeanFactoryPostProcessor

me.josephzhu.spring101beans.MyService@41330d4f#constructor:1

me.josephzhu.spring101beans.MyService@41330d4f#postProcessBeforeInitialization:11

me.josephzhu.spring101beans.MyService@41330d4f#postConstruct:12

me.josephzhu.spring101beans.MyService@41330d4f#afterPropertiesSet:13

me.josephzhu.spring101beans.MyService@41330d4f#init:14

me.josephzhu.spring101beans.MyService@41330d4f#postProcessAfterInitialization:15

me.josephzhu.spring101beans.MyService@6f36c2f0#constructor:1

me.josephzhu.spring101beans.MyService@6f36c2f0#postProcessBeforeInitialization:11

me.josephzhu.spring101beans.MyService@6f36c2f0#postConstruct:12

me.josephzhu.spring101beans.MyService@6f36c2f0#afterPropertiesSet:13

me.josephzhu.spring101beans.MyService@6f36c2f0#init:14

me.josephzhu.spring101beans.MyService@6f36c2f0#postProcessAfterInitialization:15

me.josephzhu.spring101beans.MyService@3b35a229#constructor:1

me.josephzhu.spring101beans.MyService@3b35a229#postProcessBeforeInitialization:2

me.josephzhu.spring101beans.MyService@3b35a229#postConstruct:3

me.josephzhu.spring101beans.MyService@3b35a229#afterPropertiesSet:4

me.josephzhu.spring101beans.MyService@3b35a229#postProcessAfterInitialization:5

====================
```

```
me.josephzhu.spring101beans.MyService@6692b6c6#constructor:1

me.josephzhu.spring101beans.MyService@6692b6c6#postProcessBeforeInitialization:11

me.josephzhu.spring101beans.MyService@6692b6c6#postConstruct:12

me.josephzhu.spring101beans.MyService@6692b6c6#afterPropertiesSet:13

me.josephzhu.spring101beans.MyService@6692b6c6#init:14

me.josephzhu.spring101beans.MyService@6692b6c6#postProcessAfterInitialization:15

myService:me.josephzhu.spring101beans.MyService@3b35a229

helloService:me.josephzhu.spring101beans.MyService@6692b6c6

====================

me.josephzhu.spring101beans.MyService@41330d4f#hello:15

me.josephzhu.spring101beans.MyService@6f36c2f0#hello:15

me.josephzhu.spring101beans.MyService@3b35a229#preDestroy:5

me.josephzhu.spring101beans.MyService@3b35a229#destroy:5
```

这个输出结果有点长，第一行就输出了 MyBeanFactoryPostProcessor 这是预料之中，Bean 定义的修改肯定是最先发生的。我们看下输出的规律，1、11、12、13、14、15 出现了三次，之所以从 1 跳到了 11 是因为我们的 BeanFactoryPostProcessor 修改了其中的 counter 属性的值为 10。这说明了，我们的 helloService 的初始化进行了三次：

第一套指针地址是 5a7fe64f，对应输出第一个 hello()，这是我们@Resource 引入的

```
MyBeanFactoryPostProcessor
me.josephzhu.spring101beans.MyService@5a7fe64f#constructor:1
me.josephzhu.spring101beans.MyService@5a7fe64f#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@5a7fe64f#postConstruct:12
me.josephzhu.spring101beans.MyService@5a7fe64f#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@5a7fe64f#init:14
me.josephzhu.spring101beans.MyService@5a7fe64f#postProcessAfterInitialization:15
me.josephzhu.spring101beans.MyService@69ee81fc#constructor:1
me.josephzhu.spring101beans.MyService@69ee81fc#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@69ee81fc#postConstruct:12
me.josephzhu.spring101beans.MyService@69ee81fc#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@69ee81fc#init:14
me.josephzhu.spring101beans.MyService@69ee81fc#postProcessAfterInitialization:15
me.josephzhu.spring101beans.MyService@29f7cefd#constructor:1
me.josephzhu.spring101beans.MyService@29f7cefd#postProcessBeforeInitialization:2
me.josephzhu.spring101beans.MyService@29f7cefd#postConstruct:3
me.josephzhu.spring101beans.MyService@29f7cefd#afterPropertiesSet:4
me.josephzhu.spring101beans.MyService@29f7cefd#postProcessAfterInitialization:5
2018-10-01 14:36:03.539  INFO 15461 --- [           main] o.s.j.e.a.AnnotationMBeanExporter
2018-10-01 14:36:03.579  INFO 15461 --- [           main] m.j.s.Spring101BeansApplication
====================
me.josephzhu.spring101beans.MyService@bb9e6dc#constructor:1
me.josephzhu.spring101beans.MyService@bb9e6dc#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@bb9e6dc#postConstruct:12
me.josephzhu.spring101beans.MyService@bb9e6dc#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@bb9e6dc#init:14
me.josephzhu.spring101beans.MyService@bb9e6dc#postProcessAfterInitialization:15
myService:me.josephzhu.spring101beans.MyService@29f7cefd
helloService:me.josephzhu.spring101beans.MyService@bb9e6dc
====================
me.josephzhu.spring101beans.MyService@5a7fe64f#hello:15
me.josephzhu.spring101beans.MyService@69ee81fc#hello:15
```

第二套指针地址是 69ee81fc，对应输出第二个 hello()，这是我们@Autowird+@Qualifier 引入的（刚才一节最后我们指定了 helloService）

```
me.josephzhu.spring101beans.MyService@5a7fe64f#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@5a7fe64f#postConstruct:12
me.josephzhu.spring101beans.MyService@5a7fe64f#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@5a7fe64f#init:14
me.josephzhu.spring101beans.MyService@5a7fe64f#postProcessAfterInitialization:15
me.josephzhu.spring101beans.MyService@69ee81fc#constructor:1
me.josephzhu.spring101beans.MyService@69ee81fc#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@69ee81fc#postConstruct:12
me.josephzhu.spring101beans.MyService@69ee81fc#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@69ee81fc#init:14
me.josephzhu.spring101beans.MyService@69ee81fc#postProcessAfterInitialization:15
me.josephzhu.spring101beans.MyService@29f7cefd#constructor:1
me.josephzhu.spring101beans.MyService@29f7cefd#postProcessBeforeInitialization:2
me.josephzhu.spring101beans.MyService@29f7cefd#postConstruct:3
me.josephzhu.spring101beans.MyService@29f7cefd#afterPropertiesSet:4
me.josephzhu.spring101beans.MyService@29f7cefd#postProcessAfterInitialization:5
2018-10-01 14:36:03.539  INFO 15461 --- [           main] o.s.j.e.a.AnnotationMBeanExporter
2018-10-01 14:36:03.579  INFO 15461 --- [           main] m.j.s.Spring101BeansApplication
====================
me.josephzhu.spring101beans.MyService@bb9e6dc#constructor:1
me.josephzhu.spring101beans.MyService@bb9e6dc#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@bb9e6dc#postConstruct:12
me.josephzhu.spring101beans.MyService@bb9e6dc#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@bb9e6dc#init:14
me.josephzhu.spring101beans.MyService@bb9e6dc#postProcessAfterInitialization:15
myService:me.josephzhu.spring101beans.MyService@29f7cefd
helloService:me.josephzhu.spring101beans.MyService@bb9e6dc
====================
me.josephzhu.spring101beans.MyService@5a7fe64f#hello:15
me.josephzhu.spring101beans.MyService@69ee81fc#hello:15
2018-10-01 14:36:03.594  INFO 15461 --- [       Thread-5] s.c.a.AnnotationConfigApplicationC
2018-10-01 14:36:03.600  INFO 15461 --- [       Thread-5] o.s.j.e.a.AnnotationMBeanExporter
```

第三套指针地址是 29f7cefd，这是我们 getBeansOfType 的时候创建的，对应下面 Key-Value 的输出：

```
me.josephzhu.spring101beans.MyService@29f7cefd#postProcessBeforeInitialization:2
me.josephzhu.spring101beans.MyService@29f7cefd#postConstruct:3
me.josephzhu.spring101beans.MyService@29f7cefd#afterPropertiesSet:4
me.josephzhu.spring101beans.MyService@29f7cefd#postProcessAfterInitialization:5
2018-10-01 14:36:03.539  INFO 15461 --- [           main] o.s.j.e.a.AnnotationMBeanEx
2018-10-01 14:36:03.579  INFO 15461 --- [           main] m.j.s.Spring101BeansApplica
====================
me.josephzhu.spring101beans.MyService@bb9e6dc#constructor:1
me.josephzhu.spring101beans.MyService@bb9e6dc#postProcessBeforeInitialization:11
me.josephzhu.spring101beans.MyService@bb9e6dc#postConstruct:12
me.josephzhu.spring101beans.MyService@bb9e6dc#afterPropertiesSet:13
me.josephzhu.spring101beans.MyService@bb9e6dc#init:14
me.josephzhu.spring101beans.MyService@bb9e6dc#postProcessAfterInitialization:15
myService:me.josephzhu.spring101beans.MyService@29f7cefd
helloService:me.josephzhu.spring101beans.MyService@bb9e6dc
====================
me.josephzhu.spring101beans.MyService@5a7fe64f#hello:15
me.josephzhu.spring101beans.MyService@69ee81fc#hello:15
2018-10-01 14:36:03.594  INFO 15461 --- [       Thread-5] s.c.a.AnnotationConfigAppli
2018-10-01 14:36:03.600  INFO 15461 --- [       Thread-5] o.s.j.e.a.AnnotationMBeanEx
me.josephzhu.spring101beans.MyService@29f7cefd#preDestroy:5
me.josephzhu.spring101beans.MyService@29f7cefd#destroy:5

Process finished with exit code 0
```

这里的输出说明了几点：

- 我们的 BeanFactoryPostProcessor 生效了，不但修改了 helloService 的 Scope 为 prototype 而且修改了它的 counter 属性
- 对于 Scope=ptototype 的 Bean，显然在每次使用 Bean 的时候都会新建一个实例
- BeanPostProcessor 两个方法的顺序结合一开始说的 Bean 事件回调的顺序整体如下：
  1. 类自己的构造方法
  2. **BeanFactoryPostProcessor 接口实现的 postProcessBeforeInitialization()方法**
  3. @PostConstruct 注释的方法
  4. InitializingBean 接口实现的 afterPropertiesSet()方法
  5. Init-method 定义的方法
  6. **BeanFactoryPostProcessor 接口实现的 postProcessAfterInitialization()方法**
  7. @PreDestroy 注释的方法
  8. DisposableBean 接口实现的 destroy()方法

最后，我们可以修改 BeanFactoryPostProcessor 中的代码把 prototype 修改为 singleton 看看是否我们的 helloService 这个 Bean 恢复为了单例：

```
MyBeanFactoryPostProcessor

me.josephzhu.spring101beans.MyService@51891008#constructor:1

me.josephzhu.spring101beans.MyService@51891008#postProcessBeforeInitialization:11

me.josephzhu.spring101beans.MyService@51891008#postConstruct:12
```

```
me.josephzhu.spring101beans.MyService@51891008#afterPropertiesSet:13

me.josephzhu.spring101beans.MyService@51891008#init:14

me.josephzhu.spring101beans.MyService@51891008#postProcessAfterInitialization:15

me.josephzhu.spring101beans.MyService@49c90a9c#constructor:1

me.josephzhu.spring101beans.MyService@49c90a9c#postProcessBeforeInitialization:2

me.josephzhu.spring101beans.MyService@49c90a9c#postConstruct:3

me.josephzhu.spring101beans.MyService@49c90a9c#afterPropertiesSet:4

me.josephzhu.spring101beans.MyService@49c90a9c#postProcessAfterInitialization:5

====================

myService:me.josephzhu.spring101beans.MyService@49c90a9c

helloService:me.josephzhu.spring101beans.MyService@51891008

====================

me.josephzhu.spring101beans.MyService@51891008#hello:15

me.josephzhu.spring101beans.MyService@51891008#hello:15

me.josephzhu.spring101beans.MyService@49c90a9c#preDestroy:5

me.josephzhu.spring101beans.MyService@49c90a9c#destroy:5

me.josephzhu.spring101beans.MyService@51891008#preDestroy:15

me.josephzhu.spring101beans.MyService@51891008#destroy:15
```

本次输出结果的 hello()方法明显是同一个 bean，结果中也没出现三次 1、11、12、13、14、15。

总结

本文以探索的形式讨论了下面的一些知识点：

1. 容器管理对象的意义是什么
2. Bean 的生命周期回调事件
3. Spring 提供的 Bean 的两个重要扩展点
4. @Resource 和@Autowired 的区别
5. 注解方式和代码方式配置 Bean

6. @Primary 和@Qualifier 注解的作用
7. Bean 的不同类型的 Scope