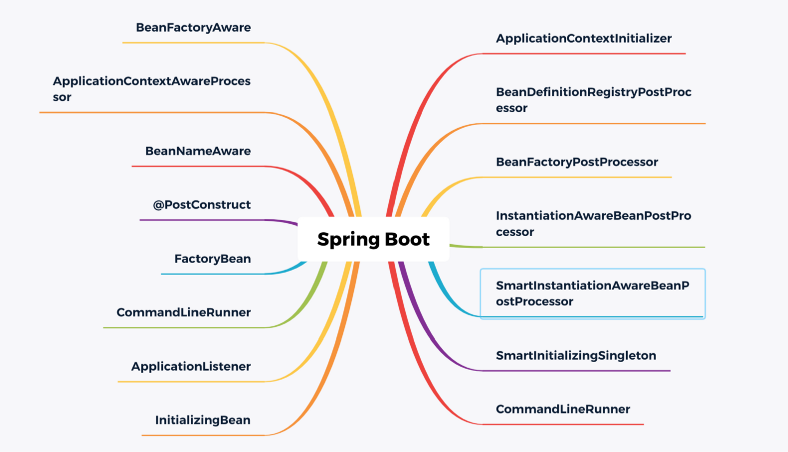
[[](https://javarevisited.blogspot.com/2018/05/top-5-courses-to-learn-spring-boot-in.html)](https://javarevisited.blogspot.com/2018/05/top-5-courses-to-learn-spring-boot-in.html)

As senior programmers know, the core idea of [Spring](https://medium.com/javarevisited/top-10-free-courses-to-learn-spring-framework-for-java-developers-639db9348d25)is the container, so in the project, if the container is refreshed, it will be a stormy sea. For [Spring boot,](https://medium.com/javarevisited/10-free-spring-boot-tutorials-and-courses-for-java-developers-53dfe084587e) it encapsulates Spring, following conventions rather than configuration, plus the mechanism of automatic assembly.

Personally, I really like the mechanics of [autowiring](https://javarevisited.blogspot.com/2022/03/how-autowiring-of-beans-works-in-spring.html" \t "_blank). Of course, if you want to be familiar with automatic assembly, you must understand spring’s bean construction life cycle and various extension interfaces. This is not easy to imagine  
In this article, I would like to share with you some of the extension interfaces of [Spring](https://javarevisited.blogspot.com/2020/06/10-advanced-spring-framework-courses.html) & [Springboot](https://medium.com/javarevisited/top-10-courses-to-learn-spring-boot-in-2020-best-of-lot-6ffce88a1b6e), as well as the usage scenarios of these extension points.

Because there are too many interfaces, I will use 5 examples to illustrate. For the specific extension interface, you can take a look at the map I drew. For the other remaining interfaces, you can supplement and exchange in the comment area.



Photo by [Thought Catalog](https://unsplash.com/@thoughtcatalog?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**Extension point (1): ApplicationContextInitializer**

First, let’s briefly understand the callback interface of the entire spring container to initialize the ConfigurableApplicationContext before refreshing, that is, to call the initialize method of this class before the container is refreshed.

This is better because the user can do something before the entire spring container has been initialized, because it allows the user to extend by himself.  
**【scenes to be used】**  
(1) Activate some configurations at the beginning  
(2) Or use the time when the class has not been loaded by the class loader to perform dynamic bytecode injection and other operations  
***The expansion method is:***

public class TestApplicationContextInitializer implements ApplicationContextInitializer {  
@Override  
public void initialize(ConfigurableApplicationContext applicationContext) {  
System.out.println("[ApplicationContextInitializer]");  
}  
}

You can see that the spring container has not been initialized at this time. If your extension takes effect, there are two ways:  
(1) In the startup class, use the springApplication.addInitializers(new TestApplicationContextInitializer()) statement to add  
(2) Configuration file configuration context.initializer.classes=com.example.demo.TestApplicationContextInitialize



Photo by [Sandy Millar](https://unsplash.com/@sandym10?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**Extension point (2): BeanFactoryPostProcessor**

This interface is the extension interface of [beanFactory](https://javarevisited.blogspot.com/2012/11/difference-between-beanfactory-vs-applicationcontext-spring-framework.html" \t "_blank). The call timing is after spring reads the beanDefinition information and before instantiating the bean.

At this time, the user can handle some things by implementing this extension interface, such as modifying the meta information of the registered beanDefinition.  
***The expansion method is:***

public class TestBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
@Override  
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
System.out.println("[BeanFactoryPostProcessor]");  
}  
}

**Extension point (3): BeanFactoryAware**

There is only one trigger point for this class, which happens after the instantiation of the bean and before the property is injected, that is, before the Setter. The extension point method of this class is setBeanFactory, which can get the property of BeanFactory.  
**【scenes to be used】**  
(1) You can get the BeanFactory after the bean is instantiated, but before it is initialized. At this time, you can make special customizations for each bean.  
(2) You can also get the [BeanFactory](https://javarevisited.blogspot.com/2012/11/difference-between-beanfactory-vs-applicationcontext-spring-framework.html" \t "_blank)for caching and use it in the future.  
***The expansion method is:***

public class TestBeanFactoryAware implements BeanFactoryAware {  
@Override  
public void setBeanFactory(BeanFactory beanFactory) throws BeansException {  
System.out.println("[TestBeanFactoryAware] " + beanFactory.getBean(TestBeanFactoryAware.class).getClass().getSimpleName());  
}  
}



Photo by [Simon Kuznetsov](https://unsplash.com/@simonkuznetsovphoto?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**Extension point (4): @PostConstruct**

Does it count as an extension point for this? In my opinion, this is not actually an extension point, it is a label. Its role is that in the initialization phase of the bean, if a method is marked with [@PostConstruc](https://javarevisited.blogspot.com/2022/05/50-free-spring-professional-certification-questions.html)t, this method will be called first.

The point here is to pay attention to the trigger point of this standard, which is after postProcessBeforeInitialization and before InitializingBean.afterPropertiesSet.  
**【scenes to be used】:**  
Users can annotate a method to initialize a property  
***The expansion method is:***

public class NormalBeanA {  
public NormalBeanA() {  
System.out.println("NormalBean constructor");  
}  
@PostConstruct  
public void init(){  
System.out.println("[PostConstruct] NormalBeanA");  
}  
}

**Extension point (5): SmartInitializingSingleton**

There is only one method afterSingletonsInstantiated in this interface, which is a callback interface called after all singleton objects managed by the spring container are initialized. The trigger timing is after postProcessAfterInitialization.

**【scenes to be used】**  
Users can extend this interface to do some post-processing after initializing all singleton objects.  
***The expansion method is:***

public class TestSmartInitializingSingleton implements SmartInitializingSingleton {  
@Override  
public void afterSingletonsInstantiated() {  
System.out.println("[TestSmartInitializingSingleton]");  
}  
}