



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

## 【死磕 Spring】—— IOC 之 深入分析 BeanFactoryPostProcessor (<http://cmsblogs.com/?p=3342>)

2018-12-09 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(8910) 评论(1)

原文出自: <http://cmsblogs.com> (<http://cmsblogs.com>)

在博客【死磕 Spring】----- IOC 之 深入分析 BeanPostProcessor () 深入介绍了 BeanPostProcessor 的实现机制。在这篇文章中提到 BeanPostProcessor 是 Spring 提供一种扩展机制, 该机制允许我们在 Bean 实例化之后初始化之际对 Bean 进行增强处理 (前、后置处理)。同样在 Spring 容器启动阶段, Spring 也提供了一种容器扩展机制: BeanFactoryPostProcessor, 该机制作用于容器启动阶段, 允许我们在容器实例化 Bean 之前对注册到该容器的 BeanDefinition 做出修改。

### BeanFactoryPostProcessor

BeanPostProcessor对bean实例对象初始化方法前后扩展  
BeanFactoryPostProcessor对实例化对象之前对BeanDefintion扩展修改

BeanFactoryPostProcessor 的机制就相当于给了我们在 bean 实例化之前最后一次修改 BeanDefinition 的机会, 我们可以利用这个机会对 BeanDefinition 来进行一些额外的操作, 比如更改某些 bean 的一些属性, 给某些 Bean 增加一些其他的信息等等操作。

定义如下

```
public interface BeanFactoryPostProcessor {

    /**
     * 1、Modify the application context's internal bean factory after its standard initialization.
     *
     * 2、All bean definitions will have been loaded, but no beans will have been instantiated yet. This all
     * ows for overriding or adding properties even to eager-initializing beans.
     *
     * @param beanFactory the bean factory used by the application context
     * @throws org.springframework.beans.BeansException in case of errors
     */
    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;

}
```

BeanFactoryPostProcessor 接口 仅 有一个 postProcessBeanFactory 方法, 该方法接收一个 ConfigurableListableBeanFactory 类型的 beanFactory 参数。上面有两行注释:

- 1、表示了该方法的作用: 在 standard initialization (实在是不知道这个怎么翻译: 标准初始化?) 之后 (已经就是已经完成了 BeanDefinition 的加载) 对 bean factory 容器进行修改。其中参数 beanFactory 应该就是已经完成了 standard initialization 的 BeanFactory。

## 2、表示作用时机：所有的 BeanDefinition 已经完成了加载即加载至 BeanFactory 中，但是还没有完成初始化。




所以这里总结一句话就是：`postProcessBeanFactory()` 工作与 BeanDefinition 加载完成之后，Bean 实例化之前，其主要作用是对加载 BeanDefinition 进行修改。有一点需要需要注意的是在 `postProcessBeanFactory()` 中千万不能进行 Bean 的实例化工作，因为这样会导致 bean 过早实例化，会产生严重后果，我们始终需要注意的是 BeanFactoryPostProcessor 是与 BeanDefinition 打交道的，如果想要与 Bean 打交道，请使用 BeanPostProcessor。

与 BeanPostProcessor 一样，BeanFactoryPostProcessor 同样支持排序，一个容器可以同时拥有多个 BeanFactoryPostProcessor，这个时候如果我们比较在乎他们的顺序的话，可以实现 Ordered 接口。

如果要自定义 BeanFactoryPostProcessor 直接实现该接口即可。

## 实例

---



```
public class BeanFactoryPostProcessor_1 implements BeanFactoryPostProcessor, Ordered{
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException
    {
        System.out.println("调用 BeanFactoryPostProcessor_1 ...");

        System.out.println("容器中有 BeanDefinition 的个数: " + beanFactory.getBeanDefinitionCount());

        // 获取指定的 BeanDefinition , 初始化实例对象之前, 修改BeanDefinition对象的属性值。
        BeanDefinition bd = beanFactory.getBeanDefinition("studentService");

        MutablePropertyValues pvs = bd.getPropertyValues();

        pvs.addPropertyValue("name", "chenssy1");
        pvs.addPropertyValue("age", 15);
    }

    @Override
    public int getOrder() {
        return 1;
    }
}


public class BeanFactoryPostProcessor_2 implements BeanFactoryPostProcessor , Ordered{
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException
    {
        System.out.println("调用 BeanFactoryPostProcessor_2 ...");

        // 获取指定的 BeanDefinition
        BeanDefinition bd = beanFactory.getBeanDefinition("studentService");
        MutablePropertyValues pvs = bd.getPropertyValues();
        pvs.addPropertyValue("age", 18);

    }

    @Override
    public int getOrder() {
        return 2;
    }
}
```

提供了两个自定义的 BeanFactoryPostProcessor , 都继承 BeanFactoryPostProcessor 和 Ordered , 其中 BeanFactoryPostProcessor\_1 改变 name 和 age 的值, BeanFactoryPostProcessor\_2 该变 age 的值。Ordered 分别为 1 和 2。



```
<bean id="studentService" class="org.springframework.core.service.StudentService">
    <property name="name" value="chenssy"/>
    <property name="age" value="10"/>
</bean>

<bean class="org.springframework.core.test.BeanFactoryPostProcessor_1"/>
<bean class="org.springframework.core.test.BeanFactoryPostProcessor_2"/>
```



studentService 设置 name 和 age 分别为 chenss 和 10。

```
ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

StudentService studentService = (StudentService) context.getBean("studentService");
System.out.println("student name:" + studentService.getName() + "-- age:" + studentService.getAge());
```

运行结果：

```
调用 BeanFactoryPostProcessor_1 ...
容器中有 BeanDefinition 的个数: 3
调用 BeanFactoryPostProcessor_2 ...
student name:chenssy1-- age:18
```

看到运行结果，其实对上面的运行流程就已经一清二楚了。这里就不过多阐述了。

在上面测试方法中，我们使用的是 ApplicationContext，对于 ApplicationContext 来说，使用 BeanFactoryPostProcessor 非常方便，因为他会自动识别配置文件中的 BeanFactoryPostProcessor 并且完成注册和调用，我们只需要简单的配置声明即可。而对于 BeanFactory 容器来说则不行，他和 BeanPostProcessor 一样需要容器主动去进行注册调用，方法如下：

```
BeanFactoryPostProcessor_1 beanFactoryPostProcessor1 = new BeanFactoryPostProcessor_1();
beanFactoryPostProcessor1.postProcessBeanFactory(factory);
```

至于 ApplicationContext 是如何自动识别和调用，这个我们后续在分析 ApplicationContext 时会做详细说明的，当然，如果有兴趣的同学可以提前看。

诚然，一般情况下我们是不会主动去自定义 BeanFactoryPostProcessor，其实 Spring 为我们提供了几个常用的 BeanFactoryPostProcessor，他们是 PropertyPlaceholderConfigurer 和 PropertyOverrideConfigurer，其中 PropertyPlaceholderConfigurer 允许我们在 XML 配置文件中使用占位符并将这些占位符所代表的资源单独配置到简单的 properties 文件中来加载，PropertyOverrideConfigurer 则允许我们使用占位符来明确表明 bean 定义中的 property 与 properties 文件中的各配置项之间的对应关系，这两个类在我们大型项目中有非常重要的作用，后续两篇文章将对其进行详细说明分析。