

# 一、BeanDefinition

## 1.BeanDefinition是什么？

在Spring框架中，BeanDefinition是描述和定义Spring容器中的Bean的元数据对象。它包含了定义Bean的相关信息，例如Bean的类名、作用域、生命周期等。

BeanDefinition对象通常由Spring容器在启动过程中根据配置信息或注解生成。是Spring IoC容器管理的核心数据结构之一，用于保存Bean的配置和属性。

解释案例

通过一个简单Java类的案例，去理解什么是BeanDefinition

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person() {}  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // getters and setters  
  
}
```

我们可以用XML配置或者Java配置的方式来定义一个Person类型的Bean，同时这个Bean的配置信息会被封装在BeanDefinition中。

```
<bean id="person" class="com.example.Person">  
    <constructor-arg name="name" value="John"/>  
    <constructor-arg name="age" value="25"/>  
</bean>
```

此BeanDefinition的信息包括了class属性（全限定类名）以及构造函数参数的名称和值

再定义一个Person Bean的配置类

```
@Configuration  
public class AppConfig {  
    @Bean  
    public Person person() {  
        return new Person("John", 25);  
    }  
}
```

在这个案例中，BeanDefinition的信息包括class属性（全限定类名）以及构造函数参数。

应该能理解到一点“什么是BeanDefinition了吧

ok 懵逼的同学可以继续向下看 还会有解释

## 2.BeanDefinition关键方法解剖

BeanDefinition接口定义了Bean的所有元信息，主要包含以下：

- get/setBeanClassName() - 获取/设置Bean的类名
- get/setScope() - 获取/设置Bean的作用域
- isSingleton() / isPrototype() - 判断是否单例/原型作用域
- get/setInitMethodName() - 获取/设置初始化方法名
- get/setDestroyMethodName() - 获取/设置销毁方法名
- get/setLazyInit() - 获取/设置是否延迟初始化
- get/setDependsOn() - 获取/设置依赖的Bean
- get/setPropertyValue() - 获取/设置属性值
- get/setAutowiredCandidate() - 获取/设置是否可以自动装配
- get/setPrimary() - 获取/设置是否首选的自动装配Bean

BeanDefinition是Spring框架中用来描述Bean的元数据对象，这个元数据包含了关于Bean的一些基本信息，总的概括为以下几个方面：

- Bean的类信息：这是Bean的全限定类名，即这个Bean实例化后的具体类型。
- Bean的属性信息：包括了Bean的作用域（单例or原型）；是否为主要（primary）、描述信息等等
- Bean的行为特性：Bean是否支持延迟加载；是否可以作为自动装配的候选者、以及Bean的初始化方法和销毁方法
- Bean的依赖关系：Bean所依赖的其他Bean，以及这个Bean是否有父Bean
- Bean的配置信息：Bean的构造器函数、属性值等

## 3.BeanDefinition方法

接下来用一个详细的代码示例来说明BeanDefinition接口中各个方法的使用，并结合实际的代码示例说明这些方法的实际含义。下面，我会针对BeanDefinition的几个重要方面提供代码示例。

这里直接展示所有代码，不一一解读，注意阅读代码注释

- AppConfig配置类

```
package com.example.demo.configuration;

import com.example.demo.bean.Person;
import org.springframework.context.annotation.*;

@Configuration
public class AppConfig {

    @Bean(initMethod = "init", destroyMethod = "cleanup")
    @Scope("singleton")
    @Lazy
    @Primary
    @Description("A bean for person")
    public Person person() {
```

```

        return new Person("John", 25);
    }
}

```

- 人员对象类

```

package com.example.demo.bean;

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // getters and setters

    public void init() {
        System.out.println("Initializing Person bean");
    }

    public void cleanup() {
        System.out.println("Cleaning up Person bean");
    }

}

```

注意看注释：

```

package com.example.demo;

import com.example.demo.configuration.AppConfig;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.Arrays;

public class DemoApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        String personBeanName = "person";
        BeanDefinition personBeanDefinition =
context.getBeanFactory().getBeanDefinition(personBeanName);

        // 获取Bean的类信息
        System.out.println("Bean Class Name: " +
context.getBean(personBeanName).getClass().getName());
    }
}

```

```

// 获取Bean的属性
System.out.println("Scope: " + personBeanDefinition.getScope());
System.out.println("Is primary: " + personBeanDefinition.isPrimary());
System.out.println("Description: " + personBeanDefinition.getDescription());

// 获取Bean的行为特征
System.out.println("Is lazy init: " + personBeanDefinition.isLazyInit());
System.out.println("Init method: " + personBeanDefinition.getInitMethodName());
System.out.println("Destroy method: " +
personBeanDefinition.getDestroyMethodName());

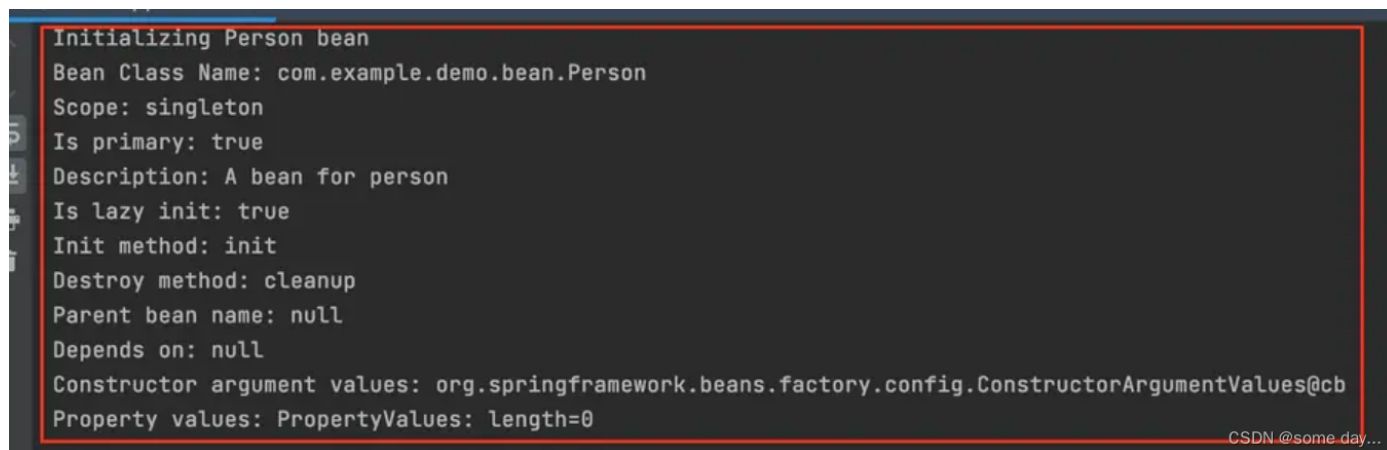
// 获取Bean的关系
System.out.println("Parent bean name: " + personBeanDefinition.getParentName());
System.out.println("Depends on: " +
Arrays.toString(personBeanDefinition.getDependsOn()));

// 获取Bean的配置属性
System.out.println("Constructor argument values: " +
personBeanDefinition.getConstructorArgumentValues());
System.out.println("Property values: " +
personBeanDefinition.getPropertyValues());
}

}

```

运行结果：



```

Initializing Person bean
Bean Class Name: com.example.demo.bean.Person
Scope: singleton
Is primary: true
Description: A bean for person
Is lazy init: true
Init method: init
Destroy method: cleanup
Parent bean name: null
Depends on: null
Constructor argument values: org.springframework.beans.factory.config.ConstructorArgumentValues@cb
Property values: PropertyValues: length=0

```

这个例子包含了BeanDefinition的大部分方法，展示了它们的作用。请注意，在这个例子中，一些方法如getDependsOn()、getParentName()、getConstructorArgumentValues()、getPropertyValues()的返回结果可能不会显示出任何实质内容，因为我们的person Bean并没有设置这些值。如果在实际应用中设置了这些值，那么这些方法将返回相应的结果。

## 4.BeanDefinition梳理

在 Spring 中，BeanDefinition 包含了以下主要信息：

- Class：这是全限定类名，Spring使用这个信息通过反射创建Bean实例。例如：  
com.example.demo.bean.xxx，当Spring需要创建Book bean的实例时，他讲根据这个类名通过反射创建Book类实例
- Name：这是Bean的名称。我们通常使用这个名称来获取 Bean 的实例。例如，我们可能有一个名称为“bookService”的 Bean，我们可以通过 context.getBean(“bookService”) 来获取这个 Bean 的实例。
- Scope：这定义了Bean的作用域，有两个值分别为singleton或prototype。如果scope是singleton，那么Spring容器将只创建一个bean实例并在每次请求时返回这个实例。如果scope是prototyoe，那么每次请求bean时，Spring都将创建一个新的bean实例。
- Constructor arguments：这是用于bean实例化的构造函数参数。如果我们有一个 Book 类，它的构造函数需要一个 String 类型的参数 title，那么我们可以在 BeanDefinition 中设置 constructor arguments 来提供这个参数。
- Properties：这些是需要注入到bean的属性值。例如，我们可能有一个 Book 类，它有一个 title 属性，我们可以在 BeanDefinition 中设置 properties 来提供这个属性的值。这些值也可以通过 标签或 @Value 注解在配置文件或类中注入。
- Autowiring Mode：这是自动装配的模式。如果byType，那么Spring容器将自动装配Bean的属性，它将查找容器中类型相匹配的bean并注入。byName，那么容器将查找容器中名称与属性名相匹配的bean并注入。还有一个选项是contructor，它是指通过bean构造函数的参数类型来自动装配依赖。
- Lazy Initialization：如果设置为true，bean将在首次请求时创建，而不是在应用启动时。这可以提高应用的启动速度，但可能会在首次请求bean时引入一些延迟。
- Initialization Method and Destroy Method：这些是bean的初始化和销毁方法。例如，我们可能有一个 BookService 类，它有一个名为 init 的初始化方法和一个名为 cleanup 的销毁方法，我们可以在 BeanDefinition 中设置这两个方法，那么 Spring 容器将在创建 Bean 后调用 init 方法，而在销毁 Bean 之前调用 cleanup 方法。
- Dependency beans：这些是 Bean 的依赖关系。例如，我们可能有一个 BookService Bean，它依赖于一个 BookRepository Bean，那么我们可以在 BookService 的 BeanDefinition 中设置 dependency beans 为“bookRepository”，那么在创建 BookService Bean 之前，Spring 容器将首先创建 BookRepository Bean。

以上就是BeanDefinition中主要包含的信息，这些信息将会告诉Spring容器如何创建和配置Bean。

不同的BeanDefinition实现可能会有不同的配置信息，例如 RootBeanDefinition ChildBeanDefinition GenericBeanDefinition 等都是BeanDefinition接口的具体实现类，它们包含更多的配置选项。

## 5.生成BeanDefinition的原理解读

首先：BeanDefinition对象是在Spring启动过程中，由各种BeanDefinitionReader实现类读取配置并生成的。在Spring中主要有三种方式来创建BeanDefiniton：

- XML配置方式

```

<bean id="bookService" class="com.example.demo.service.BookService">
    <property name="bookRepository" ref="bookRepository"/>
</bean>

<bean id="bookRepository" class="com.example.demo.repository.BookRepository"/>

```

如上代码，在这种情况下Spring启动的时候，XmlBeanDefinitionReader会读取XML文件，并解析其中的标签，并为每一个元素创建一个BeanDefinition对象。

简述为：XmlBeanDefinitionReader 读取XML文件，解析标签生成相对应的BeanDefinition。

- 注解配置方式

```

@Repository
public class BookRepository {
    // ... repository methods
}

@Service
public class BookService {
    private final BookRepository bookRepository;

    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    // ... service methods
}

```

如上代码，当Spring启动时，ClassPathBeanDefinitionScanner会扫描指定的包路径，找到所有带有特定注解的类，并为这些类创建BeanDefinition对象。

简述为：由ClassPathBeanDefinitionScanner扫描指定路径下带注解的类，并生成相对应的BeanDefinition

- Java配置方式

```

@Configuration
public class AppConfig {
    @Bean
    public BookRepository bookRepository() {
        return new BookRepository();
    }

    @Bean
    public BookService bookService(BookRepository bookRepository) {
        return new BookService(bookRepository);
    }
}

```

如上代码，当Spring启动时，ConfigurationClassPostProcessor就会处理这些配置类，并交给ConfigurationClassParser来解析。对于配置类中每一个标记了 @Bean 的方法，都会创建一个 BeanDefinition 对象。这种方式下生成的 BeanDefinition 通常是 ConfigurationClassBeanDefinition 类型。  
简述为：由ConfigurationClassPostProcessor处理标记了@Configuration的类，解析其中的@Bean方法并生成BeanDefinition。

总的来说：我们用任一哪种方式，Spring都是解析这些配置，并生成相对应的BeanDefinition对象，以此来指导Spring容器创建和管理Bean实例。

## 二、BeanFactory

### 1.什么是BeanFactory

BeanFactory是Spring框架中的一个接口，它是一个工厂类，用来创建和管理Spring中的Bean对象。BeanFactory接口定义了Spring容器的基本规范和行为，它提供了一种机制来将配置文件中定义的Bean实例化、配置和管理起来。

### 2.BeanFactory的作用

BeanFactory的主要作用是提供Bean的创建、配置、初始化和销毁等基本操作，它可以根据配置文件或注解来创建并管理Bean实例，并提供了各种方法来获取和操作Bean实例。

### 3.BeanFactory的实现类

BeanFactory接口有多个实现类，其中最常用的是XmlBeanFactory和DefaultListableBeanFactory。XmlBeanFactory是通过XML文件来配置Bean的实例化、配置和管理的，而DefaultListableBeanFactory则是通过Java代码来配置Bean的实例化、配置和管理的。下面我们将详细介绍BeanFactory的使用。

## 三、BeanFactory的使用

### 1.BeanFactory的创建

BeanFactory的创建有三种方式：XML配置方式、Java配置方式和注解配置方式。

- XML配置方式

在使用XML配置方式时，需要在配置文件中定义Bean的实例化、配置和管理信息。下面是一个简单的XML配置文件示例：

```
<?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
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userService" class="com.example.UserService">
        <property name="userDao" ref="userDao" />
    </bean>

    <bean id="userDao" class="com.example.UserDaoImpl">
```

```

        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/test"/>
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
    </bean>

</beans>

```

在上面的示例中，定义了3个Bean：userService、userDao和dataSource。其中，userService和userDao之间存在依赖关系，userService依赖于userDao，而userDao又依赖于dataSource。

- Java配置方式

在使用Java配置方式时，需要编写Java代码来定义Bean的实例化、配置和管理信息。下面是一个简单的Java配置类示例：

```

@Configuration
public class AppConfig {

    @Bean
    public UserService userService() {
        UserService userService = new UserService();
        userService.setUserDao(userDao());
        return userService;
    }

    @Bean
    public UserDao userDao() {
        UserDao userDao = new UserDaoImpl();
        userDao.setDataSource(dataSource());
        return userDao;
    }

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/test");
        dataSource.setUsername("root");
        dataSource.setPassword("123456");
        return dataSource;
    }

}

```



在上面的示例中，使用了@Configuration注解来标识该类是一个配置类，并使用@Bean注解来定义Bean的实例化、配置和管理信息。在AppConfig类中，定义了3个Bean：userService、userDao和dataSource。其中，userService和userDao之间存在依赖关系，userService依赖于userDao，而userDao又依赖于dataSource。

- 注解配置方式

在使用注解配置方式时，需要在Bean类上添加相应的注解来标识该类是一个Bean，并进行相应的配置信息。下面是一个简单的注解配置类示例：

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/test");
        dataSource.setUsername("root");
        dataSource.setPassword("123456");
        return dataSource;
    }

}
```

在上面的示例中，使用了@Configuration注解来标识该类是一个配置类，并使用@ComponentScan注解来指定需要扫描的包。在AppConfig类中，定义了一个Bean：dataSource。在该Bean中，使用了@Bean注解来标识该方法返回一个Bean实例，并进行了相应的配置信息。需要注意的是，在使用注解配置方式时，需要保证被注解标识的类在扫描范围内，否则该类将不会被Spring容器管理。

## 2.BeanFactory的配置

在BeanFactory的配置中，主要包括Bean的定义、依赖和属性等方面。

- Bean的定义

在Bean的定义中，主要包括Bean的类型、ID和作用域等方面。下面是一个简单的Bean定义示例：

```
<bean id="userService" class="com.example.UserService" scope="singleton"/>
```

在上面的示例中，定义了一个ID为userService，类型为com.example.UserService，作用域为singleton的Bean。

- Bean的依赖

在Bean的依赖中，主要包括Bean之间的依赖关系和依赖注入方式等方面。下面是一个简单的Bean依赖示例：

```
<bean id="userService" class="com.example.UserService">
    <property name="userDao" ref="userDao"/>
</bean>

<bean id="userDao" class="com.example.UserDaoImpl">
```

```
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/test"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</bean>
```

在上面的示例中，userService依赖于userDao，而userDao又依赖于dataSource。在userService中，使用了标签来注入userDao的实例，而在userDao中，同样使用了标签来注入dataSource的实例。

- Bean的属性

在Bean的属性中，主要包括Bean的各种属性信息，如普通属性、集合属性和引用属性等。下面是一个简单的Bean属性示例：

```
<bean id="userService" class="com.example.UserService">
    <property name="name" value="John"/>
    <property name="age" value="30"/>
    <property name="hobbies">
        <list>
            <value>reading</value>
            <value>writing</value>
            <value>traveling</value>
        </list>
    </property>
    <property name="userDao" ref="userDao"/>
</bean>
```

在上面的示例中，定义了一个userService的Bean，并设置了name、age、hobbies和userDao等属性。其中，name和age是普通属性，而hobbies是集合属性，它包含了三个值：reading、writing和traveling。userDao是引用属性，它依赖于另一个Bean实例。

### 3.BeanFactory的初始化

在BeanFactory的初始化中，主要包括BeanFactoryAware接口、InitializingBean接口和init-method属性等方面。

- BeanFactoryAware接口

如果一个Bean实现了BeanFactoryAware接口，那么它将能够获取到当前Bean所在的BeanFactory实例。下面是一个简单的BeanFactoryAware接口示例：

```
public class MyBean implements BeanFactoryAware {

    private BeanFactory beanFactory;

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = beanFactory;
    }

}
```

在上面的示例中，MyBean实现了BeanFactoryAware接口，并重写了setBeanFactory()方法。该方法将传入的BeanFactory实例保存到了类成员变量beanFactory中。

- InitializingBean接口

如果一个Bean实现了InitializingBean接口，那么它将能够在Bean实例化后、依赖注入后、属性设置后进行一些初始化操作。下面是一个简单的InitializingBean接口示例：

```
public class MyBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        // do something after bean initialization
    }

}
```

在上面的示例中，MyBean实现了InitializingBean接口，并重写了afterPropertiesSet()方法。该方法将在Bean实例化、依赖注入和属性设置后被调用，可以在该方法中进行一些初始化操作。

- init-method属性

如果一个Bean配置了init-method属性，那么在Bean实例化、依赖注入和属性设置后，该方法将被调用来进行一些初始化操作。下面是一个简单的init-method属性示例：

```
<bean id="myBean" class="com.example.MyBean" init-method="init">
```

在上面的示例中，定义了一个id为myBean的Bean，并配置了init-method属性为init。在Bean实例化、依赖注入和属性设置后，Spring容器将调用MyBean类中的init()方法进行初始化操作。

## 4.BeanFactory的销毁

在BeanFactory的销毁中，主要包括DisposableBean接口和destroy-method属性等方面。

- DisposableBean接口

如果一个Bean实现了DisposableBean接口，那么它将能够在Bean销毁前进行一些操作。下面是一个简单的DisposableBean接口示例：

```
public class MyBean implements DisposableBean {

    @Override
    public void destroy() throws Exception {
        // do something before bean destruction
    }

}
```

在上面的示例中，MyBean实现了DisposableBean接口，并重写了destroy()方法。该方法将在Bean销毁前被调用，可以在该方法中进行一些清理操作。

- destroy-method属性

如果一个Bean配置了destroy-method属性，那么在Bean销毁前，该方法将被调用来进行一些清理操作。下面是一个简单的destroy-method属性示例：

```
<bean id="myBean" class="com.example.MyBean" destroy-method="cleanup">
```

在上面的示例中，定义了一个id为myBean的Bean，并配置了destroy-method属性为cleanup。在Bean销毁前，Spring容器将调用MyBean类中的cleanup()方法进行清理操作。

## 四、BeanFactoryPostProcessor

BeanFactoryPostProcessor是BeanFactory的扩展点之一，它允许在BeanFactory实例化、配置和初始化Bean之前对BeanFactory进行一些修改。下面是一个简单的BeanFactoryPostProcessor示例：

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
    throws BeansException {
        // do something to modify the beanFactory
    }

}
```

在上面的示例中，定义了一个MyBeanFactoryPostProcessor类，它实现了BeanFactoryPostProcessor接口，并重写了postProcessBeanFactory()方法。在该方法中，可以对传入的beanFactory进行一些修改操作。

## 五、BeanPostProcessor

BeanPostProcessor是BeanFactory的扩展点之一，它允许在Bean实例化、依赖注入和属性设置之后对Bean进行一些修改。下面是一个简单的BeanPostProcessor示例：

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
```

```

public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
    // do something before bean initialization
    return bean;
}

@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
    // do something after bean initialization
    return bean;
}
}

```

在上面的示例中，定义了一个MyBeanPostProcessor类，它实现了BeanPostProcessor接口，并重写了postProcessBeforeInitialization()和postProcessAfterInitialization()方法。在这两个方法中，分别可以在Bean实例化、依赖注入和属性设置之前和之后对Bean进行一些修改操作。

## 六、FactoryBean

### 1.FactoryBean介绍

- FactoryBean
 

首先是个 bean，也存放在 BeanFactory 中。它具有工厂方法的功能，在程序运行中产生指定(一种)类型的 bean，并添加到了 IOC容器中的 factoryBeanObjectCache 属性中。
- BeanFactory
 

Spring提供的存放Bean的工厂，FactoryBean是一个可生产Bean的工厂Bean

### 2.FactoryBean的源码：

```

public interface FactoryBean<T> {

    String OBJECT_TYPE_ATTRIBUTE = "factoryBeanObjectType";

    //自定义创建bean方式
    @Nullable
    T getObject() throws Exception;

    // 从 beanFactory 中获取bean的类型。
    @Nullable
    Class<?> getObjectType();

    //默认单例
    default boolean isSingleton() {
        return true;
    }
}

```

### 3.如何使用FactoryBean

```
@Service
public class MyFactoryBean implements FactoryBean {

    @Override
    public Object getObject() throws Exception {
        //手动创建bean
        return new PersonServiceImpl();
    }

    @Override
    public Class<?> getObjectType() {
        return PersonServiceImpl.class;
    }

    //这个方法可以不用覆写，父类默认是单例，如果想设置成非单例可以覆写方法返回false
    public boolean isSingleton() {
        return false;
    }
}
```

创建PersonServiceImpl类，这里我们没有使用@Service注解，说明应用启动时不会放入IOS容器中

```
public class PersonServiceImpl implements PersonService{

    public void test() {
        System.out.println("3333333");
    }
}

@SpringBootTest
class DemoApplicationTests {

    @Autowired
    ApplicationContext applicationContext;

    @Test
    void test() {
        PersonService bean = applicationContext.getBean(PersonService.class);
        System.out.println(bean);
        bean.test();
        PersonService bean2 = applicationContext.getBean(PersonService.class);
        System.out.println(bean2);
    }
}
```

编写测试类通过applicationContext的getBean()方法从容器中获取PersonService的bean，发现可以获取到，说明获取bean时发现容器中存在FactoryBean 的实现类MyFactoryBean 并判断传入的类型是否是PersonService如果一样，则返回getObject()方法对象。这里我们调用了两次bean是不一样的，说明我们设置成了非单例模式

## 4.FactoryBean的使用场景

说了这么说，为什么需要 FactoryBean ，它的特殊功能是什么呢？

Spring 中 FactoryBean 最大的应用场景是用在 AOP 中。我们都知道，AOP 实际上是 Spring 在运行是创建出来的代理对象，这个对象是在运行时才被创建的，而不是在启动时定义的，这与工厂方法模式是一致的。更生动地说，AOP 代理对象通过 java 反射机制在运行时创建代理对象，并根据业务需求将相应的方法编织到代理对象的目标方法中。Spring 中的 ProxyFactoryBean 就是干这事的。

因此，FactoryBean 提供了更灵活的实例化 bean 的方法。通过 FactoryBean 我们可以创建更复杂的 bean。

- 创建一个生成随机数的FactoryBean

```
import java.util.Random;
import org.springframework.beans.factory.FactoryBean;

public class RandomNumberFactoryBean implements FactoryBean<Random> {

    private int seed;

    public void setSeed(int seed) {
        this.seed = seed;
    }

    @Override
    public Random getObject() throws Exception {
        return new Random(seed);
    }

    @Override
    public Class<?> getObjectType() {
        return Random.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

- 配置类中配置FactoryBean

```
import java.util.Random;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
```

```

@Bean
public RandomNumberFactoryBean myRandom() {
    RandomNumberFactoryBean factoryBean = new RandomNumberFactoryBean();
    factoryBean.setSeed(12345);
    return factoryBean;
}
}

```

- 获得FactoryBean，实际上获得的是工厂中的Bean

```

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MyApp {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Random random = context.getBean("myRandom", Random.class);
        int number = random.nextInt();
        System.out.println("Random number: " + number);
        context.close();
    }
}

```