# Spring系列 什么是BeanDefinition

# 一、BeanDefinition是什么？（理论）

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Sping 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。

# 二、BeanDefinition关键方法

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/setPropertyValues() - 获取/设置属性值

get/setAutowireCandidate() - 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）；是否为主要的（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

# 三、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：



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。

# 四、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。

**小总结（仔细阅读 很关键）：**

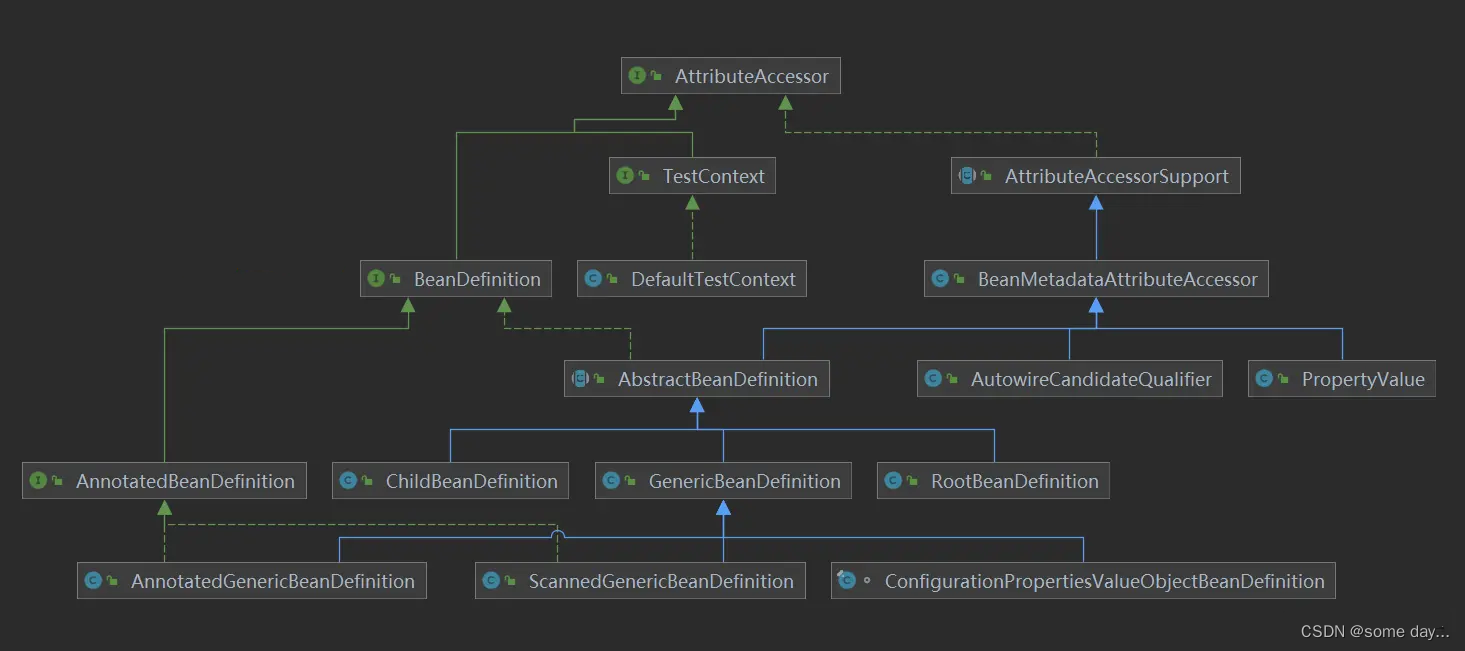
以上就是BeanDefinition中主要包含的信息，这些信息将会告诉Spring容器如何创建和配置Bean。

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。

# 五、BeanDefinition的不同实现

在Spring中，一个bean的配置信息就是由BeanDefinition来保存的。

但是但是但是根据bean配置的不同来源和方式，BeanDefinition又被分为很多种类，因此BeanDefinition的实现也有许多种，如下图：



**RootBeanDefinition**

当我们在XML中定义一个bean时，Spring会为这个bean创建一个RootBeanDefinition对象，这个对象包含了所有用于创建bean的信息，如bean的类名、属性值等

<bean id="exampleBean" class="com.example.ExampleBean">

<property name="stringProperty" value="stringValue"/>

</bean>

当Spring读取这段配置时，会创建一个RootBeanDefinition对象来保存这个bean的所有配置信息。在XML文件中定义一个bean时，Spring就会创建一个RootBeanDefinition实例，这个实例会保存所有配置信息，比如类名、属性值等等。

ChildBeanDefinition

<bean id="parentBean" class="com.example.ParentBean">

<property name="stringProperty" value="stringValue"/>

</bean>

<bean id="childBean" parent="parentBean">

<property name="anotherStringProperty" value="anotherStringValue"/>

</bean>

这段XML配置中，“childBean"继承了"parentBean"的所有配置，同时还添加了一个新的属性"anotherStringProperty”。

当Spring读取这段配置时，会首先为"parentBean"创建一个RootBeanDefinition对象，然后为childBean创建一个ChildBeanDefinition对象，这个对象会引用"parentBean"的BeanDefinition。

总的来说就是：如果现有一个beanA，并想创建一个beanB，且beanB需要继承原有bean的所有配置，但又要添加或修改一些配置信息，Spring就会创建一个ChildBeanDenifition实例去存储相关bean的配置信息。

**GenericBeanDefinition**

这是一种通用的BeanDefinition，可以根据需要转化为RootDefinition或者ChildBeanDefinition。  
如下，在一个配置类中使用@Bean注解定义了一个bean：

@Configuration

public class AppConfig {

@Bean

public MyComponent myComponent() {

return new MyComponent();

}

}

在这段代码中，我们定义了一个名为"myComponent"的bean，它的类是"MyComponent"。当Spring解析这个配置类时，会为myComponent()方法创建一个GenericBeanDefinition对象。这个GenericBeanDefinition对象会保存方法的名字（这也是bean的名字）、返回类型，以及任何需要的构造函数参数或属性。这个GenericBeanDefinition对象之后可以被Spring容器用于生成bean的实例。

总的来说：在Java配置类中使用@Bean注解定义一个bean的时候，Spring就会创建一个GenericBeanDefinition实例。

AnnotatedBeanDefinition

当我们在代码中使用注解（@Component @Servicec @Repository等注解）来定义bean时，Spring会创建一个AnnotatedBeanDefinition接口的实例。

@Component("myComponent")

public class MyComponent {

// some fields and methods

}

在这段代码中，我们定义了一个名为"myComponent"的bean，它的类是"MyComponent"，并且这个类上有一个@Component注解。当Spring解析这个类时，会创建一个AnnotatedBeanDefinition对象。这个AnnotatedBeanDefinition对象会保存类名（这也是bean的名字）、类的类型，以及类上的所有注解信息。在这个例子中，AnnotatedBeanDefinition实例会包含@Component注解及其所有元数据。这个AnnotatedBeanDefinition实例之后可以被Spring容器用于生成bean的实例，同时Spring还可以使用存储在AnnotatedBeanDefinition中的注解信息来进行进一步的处理，如AOP代理、事务管理等。

总的来说：在类上使用注解（如@Component @Service @Repository等）来定义一个bean时，Spring会创建一个实现了AnnotatedBeanDefinition接口的实力，如AnnotatedGenericBeanDefinition或者ScannedGenericBeanDefinition。

GenericBeanDefinition和AnnotatedBeanDefinition的主要区别在于，AnnotatedBeanDefinition保存了类上的注解信息，这使得Spring能够在运行时读取和处理这些注解。

在大多数情况下，我们并不需要关心Spring为bean创建的是哪一种BeanDefinition。Spring会自动管理这些BeanDefinition，并根据它们的类型以及它们所包含的信息来创建和配置bean。

# 六、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);

}

}

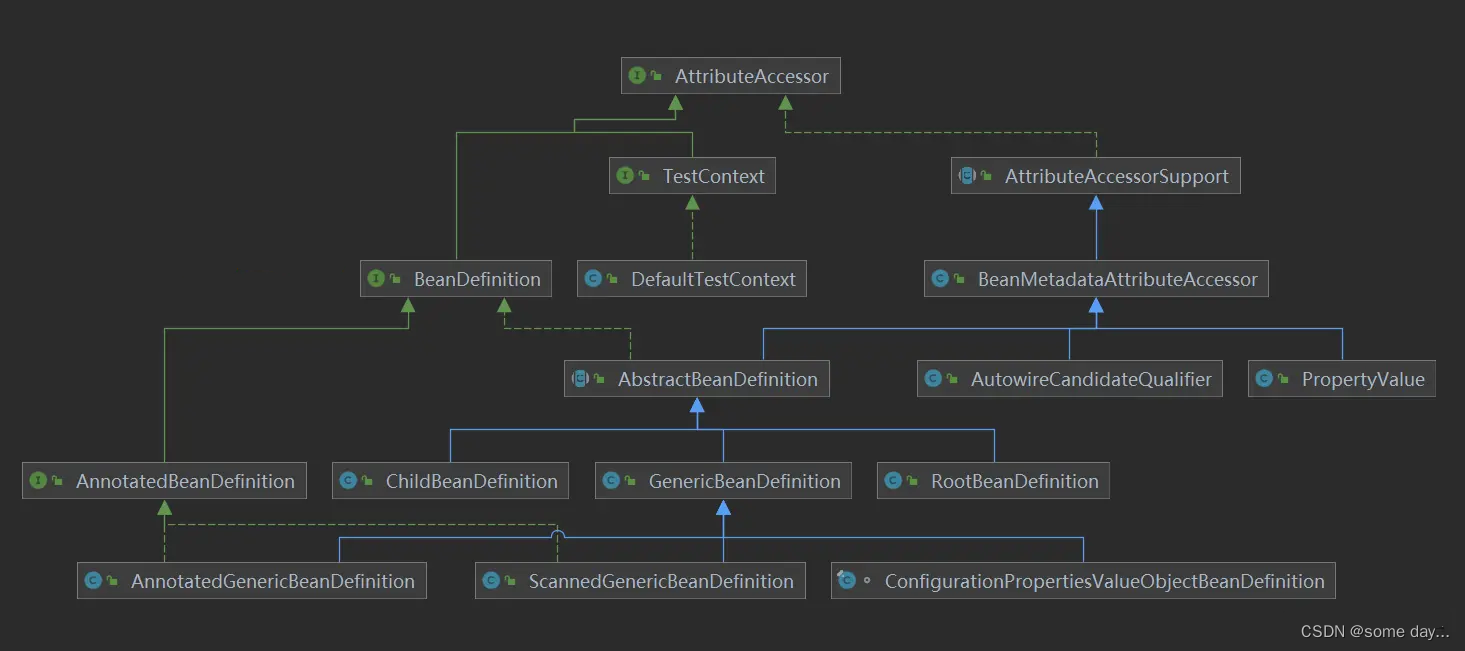
如上代码，当Spring启动时，ConfigurationClassPostProcessor就会处理这些配置类，并交给ConfigurationClassParser来解析。对于配置类中每一个标记了 @Bean 的方法，都会创建一个 BeanDefinition 对象。这种方式下生成的 BeanDefinition 通常是 ConfigurationClassBeanDefinition 类型。

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

# 七、额外补充：AttributeAccessor

BeanDefinition的不同实现的一张图：



从上图中可以看到一个叫做AttributeAccessor的接口

什么是AttributeAccessor（理论）

AttributeAccessor是Spring框架中的一个重要接口，他提供了一种灵活的方式来附加额外的元数据到Spring的核心组件。在Spring中，包括BeanDefinition在内的许多重要类都实现了AttributeAccessor接口，这样就可以动态地添加和获取这些组件的额外属性。这样做的一个显著的好处是：开发人员可以在不改变原有类定义的情况下，灵活的管理这些组件的额外信息。

AttributeAccessor接口定义的方法是为了附加、获取和移出与某个对象相关联的元数据，而不是为了操作对象本身。

简单案例

class Book {

private String title;

private String author;

public Book() {}

public Book(String title, String author) {

this.title = title;

this.author = author;

}

// getter 和 setter 省略...

}

package com.example.demo;

import com.example.demo.bean.Book;

import org.springframework.beans.factory.config.BeanDefinition;

import org.springframework.beans.factory.support.RootBeanDefinition;

public class DemoApplication {

public static void main(String[] args) {

// 创建一个BeanDefinition, BeanDefinition是AttributeAccessor的子接口

BeanDefinition bd = new RootBeanDefinition(Book.class);

// 设置属性

bd.setAttribute("bookAttr", "a value");

// 检查和获取属性

if(bd.hasAttribute("bookAttr")) {

System.out.println("bookAttr: " + bd.getAttribute("bookAttr"));

// 移除属性

bd.removeAttribute("bookAttr");

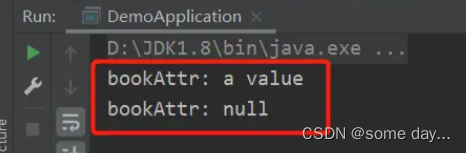
System.out.println("bookAttr: " + bd.getAttribute("bookAttr"));

}

}

}

运行结果：



在这个案例中，我们创建了一个RootBeanDefinition的实例。RootBeanDefinition是BeanDefinition的实现，而BeanDefinition是实现了AttributeAccessor接口，因此RootBeanDefinition也就继承了AttributeAccessor的方法。

探究原理

在RootBeanDefinition实例上调用setAttribute(“bookAttr”, “a value”)方法时，其实并不是在Book实例上设置一个名为bookAttr的字段，而是在RootBeanDefinition实例上附加了一个元数据，这个元数据的键是"bookAttr"，值是"a value"。

简单来说就是：这些元数据是附加在RootBeanDefinition对象上的，而不是附加在由RootBeanDefinition对象描述的实例上的。

# 简单总结

BeanDefinition是实现了AttributeAccessor接口的一个重要类，BeanDefinition对象是Spring框架用来存储bean配置信息的数据结构。

当Spring容器在后续需要创建bean实例时，它会查看BeanDefinition对象，按照其中的元数据（例如：scope 初始化 销毁方法等等）来创建和管理bean实例。且这些元数据不会直接附加在对应的bean实例上，而是存储在BeanDefinition对象中，由Spring容器来管理和使用。

因此：我们在main方法中通过Applicaition获取BeanDefinition并打印其属性时，实际上我们是在查看Spring框架用来管理的内部数据结构，也就是BeanDefinition，而不是直接查看bean实例本身。