# Spring源码深度解析与注解驱动开发

## 文档结构

### 1.1目录视图

**Spring注解驱动开发**

1. 容器

·AnnotationConfigApplicationContext

·组件添加

·组件赋值

·组件注入

·AOP

·声明式事务

1. 扩展原理

·BeanFactoryPostProcessor

·BeanDefinitionRegistryPostProcessor

·ApplicationListener

·Spring容器创建过程

1. Web

·Servlet3.0

·异步请求

## 第2节 组件注册

### 2.1 @Configuration&@Bean给容器中注册组件

|  |
| --- |
| <?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="person" class="com.byf.bean.Person">  <property name="name" value="张三"></property>  <property name="age" value="20"></property>  </bean> </beans> |
| ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml"); Person person = (Person) applicationContext.getBean("person"); System.out.println(person); |

|  |
| --- |
| @Configuration public class PersonConfig {  @Bean("lisi")  public Person person(){  return new Person("李四", 21);  } } |
| ApplicationContext applicationContext =  new AnnotationConfigApplicationContext(PersonConfig.class); Person person = (Person) applicationContext.getBean("lisi"); System.out.println(person); |

### 2.2 @ComponentScan-自动扫描组件&指定扫描规则

|  |
| --- |
| @Configuration @ComponentScan(value = "com.byf", includeFilters = {  @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = {Controller.class}) },useDefaultFilters = false) @ComponentScans(value = {@ComponentScan(value = "com.byf", includeFilters = {  @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = {Service.class}) },useDefaultFilters = false)}) public class PersonConfig {  @Bean("lisi")  public Person person(){  return new Person("李四", 21);  } } |
| public class IOCTest {  @Test  public void testIOC(){  ApplicationContext applicationContext = new AnnotationConfigApplicationContext(PersonConfig.class);  String[] beanNames = applicationContext.getBeanDefinitionNames();  for(String name : beanNames){  System.out.println(name);  }  } } |
| personConfig  bookController  bookService  lisi |

### 2.3自定义TypeFilter指定过滤规则

|  |
| --- |
| @Configuration @ComponentScans(value = {@ComponentScan(value = "com.byf", includeFilters = {  @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = {Controller.class}),  @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, classes = {BookService.class}),  @ComponentScan.Filter(type = FilterType.CUSTOM, value = {MyTypeFilter.class}) },useDefaultFilters = false)}) //FilterType.ANNOTATION：根据注解类注入 //FilterType.ASSIGNABLE\_TYPE: 根据class名注入 //FilterType.ASPECTJ: 根据ASPECTJ表达式 //FilterType.REGEX：根据正则表达式 //FilterType.CUSTOM：根据自定义规则 /\*@ComponentScans(value = {@ComponentScan(value = "com.byf", excludeFilters = {  @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = {Service.class}) })})\*/ public class PersonConfig {  @Bean("lisi")  public Person person(){  return new Person("李四", 21);  } } |

|  |
| --- |
| public class MyTypeFilter implements TypeFilter {  @Override  public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {  AnnotationMetadata annotationMetadata = metadataReader.getAnnotationMetadata();  ClassMetadata classMetadata = metadataReader.getClassMetadata();  Resource resource = metadataReader.getResource();  String name = classMetadata.getClassName();  System.out.println("-->" + name);  if ("er".equals(name.substring(name.length()-2,name.length()))){  return true;  }  return false;  } } |
| -->com.byf.AppTest  -->com.byf.bean.IOCTest  -->com.byf.App  -->com.byf.bean.Person  -->com.byf.config.MyTypeFilter  -->com.byf.controller.BookController  -->com.byf.dao.BookDao  -->com.byf.service.BookService  org.springframework.context.annotation.internalConfigurationAnnotationProcessor  .....  personConfig  myTypeFilter  bookController  lisi |

### 2.4@Scope-设置组件作用域

|  |
| --- |
| @Configuration public class MainConfig2 {  // 默认是单实例的  /\*\*  \* @see ConfigurableBeanFactory#SCOPE\_PROTOTYPE  \* @see ConfigurableBeanFactory#SCOPE\_SINGLETON  \* prototype：多实例  \* singleton：单实例（默认值）：ioc容器启动会调用方法创建对象放到ioc容器中。  \* 以后每次获取就是直接从容器（map.get()）中拿  \* request：同一个请求创建一个实例  \* session：同一个session创建一个实例  \*/  @Scope(value = "prototype")  @Bean("person")  public Person person(){  System.out.println("给容器添加Person...");  return new Person("李四", 21);  } } |

|  |
| --- |
| @Test public void testIOC2(){  ApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfig2.class);  System.out.println("ioc容器创建完成....");  Object p1 = applicationContext.getBean("person");  Object p2 = applicationContext.getBean("person");  System.out.println(p1 == p2);  /\*String[] beanNames = applicationContext.getBeanDefinitionNames();  for(String name : beanNames){  System.out.println(name);  }   Object p1 = applicationContext.getBean("person");  Object p2 = applicationContext.getBean("person");  System.out.println(p1 == p2);\*/ } |
| ioc容器创建完成....  给容器添加Person...  给容器添加Person...  false |

### 2.5@Lazy-bean懒加载

|  |
| --- |
| @Configuration public class MainConfig2 {  // 默认是单实例的  /\*\*  \* @see ConfigurableBeanFactory#SCOPE\_PROTOTYPE  \* @see ConfigurableBeanFactory#SCOPE\_SINGLETON  \* prototype：多实例  \* singleton：单实例（默认值）：ioc容器启动会调用方法创建对象放到ioc容器中。  \* 以后每次获取就是直接从容器（map.get()）中拿  \* request：同一个请求创建一个实例  \* session：同一个session创建一个实例  \*  \* 懒加载：  \* 单实例bean：默认在容器启动的时候创建  \* 懒加载：容器启动的时候不创建对象，第一次使用（获取）Bean创建对象，并初始化  \*/  // @Scope(value = "prototype")  @Bean("person")  @Lazy  public Person person(){  System.out.println("给容器添加Person...");  return new Person("李四", 21);  } } |
| @Test public void testIOC2(){  ApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfig2.class);  System.out.println("ioc容器创建完成....");  Object p1 = applicationContext.getBean("person");  Object p2 = applicationContext.getBean("person");  System.out.println(p1 == p2);  /\*String[] beanNames = applicationContext.getBeanDefinitionNames();  for(String name : beanNames){  System.out.println(name);  }   Object p1 = applicationContext.getBean("person");  Object p2 = applicationContext.getBean("person");  System.out.println(p1 == p2);\*/ } |
| ioc容器创建完成....  给容器添加Person...  true |

### 2.6@Conditional-按照条件注册bean

|  |
| --- |
| public class LinuxCondition implements Condition {  /\*\*  \* ConditionContext：判断条件能使用上下文（环境）  \* AnnotatedTypeMetada：注释信息  \* @param context  \* @param metadata  \* @return  \*/  @Override  public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  Environment environment = context.getEnvironment();  //4、获取bean定义的注册类  BeanDefinitionRegistry beanDefinitionRegistry = context.getRegistry();  String os = environment.getProperty("os.name");  if (os.contains("Linux")){  return true;  }  return false;  } } |
| public class WindowsCondition implements Condition {  /\*\*  \* ConditionContext：判断条件能使用上下文（环境）  \* AnnotatedTypeMetada：注释信息  \* @param context  \* @param metadata  \* @return  \*/  @Override  public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  //1、能获取到ioc使用beanFactory  ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();  //2、获取类加载器  ClassLoader classLoader = context.getClassLoader();  //3、获取当前环境信息  Environment environment = context.getEnvironment();  //4、获取bean定义的注册类  BeanDefinitionRegistry beanDefinitionRegistry = context.getRegistry();  String os = environment.getProperty("os.name");  if (os.contains("Windows")){  return true;  }  return false;  } } |
| @Configuration @ComponentScan(value = "com.byf") public class MainConfig2 {  // 默认是单实例的  /\*\*  \* @see ConfigurableBeanFactory#SCOPE\_PROTOTYPE  \* @see ConfigurableBeanFactory#SCOPE\_SINGLETON  \* prototype：多实例  \* singleton：单实例（默认值）：ioc容器启动会调用方法创建对象放到ioc容器中。  \* 以后每次获取就是直接从容器（map.get()）中拿  \* request：同一个请求创建一个实例  \* session：同一个session创建一个实例  \*  \* 懒加载：  \* 单实例bean：默认在容器启动的时候创建  \* 懒加载：容器启动的时候不创建对象，第一次使用（获取）Bean创建对象，并初始化  \*/  // @Scope(value = "prototype")  @Conditional({WindowsCondition.class})  @Bean  @Lazy  public Person person(){  System.out.println("给容器添加Person...");  return new Person("张三", 21);  }  @Conditional({WindowsCondition.class})  @Bean("bill")  @Lazy  public Person person01(){  System.out.println("给容器添加Person...");  return new Person("bill", 21);  }  @Conditional(LinuxCondition.class)  @Bean("linus")  @Lazy  public Person person02(){  System.out.println("给容器添加Person...");  return new Person("linus", 21);  } } |
| /\*\*  \* @Conditional：按照一定的条件进行判断，满足条件给容器中注册bean  \*  \*/ @Test public void test03(){  ApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfig2.class);   String[] namesForType = applicationContext.getBeanNamesForType(Person.class);  Environment environment = applicationContext.getEnvironment();  // 动态获取环境变量的值：Windows 7  String os = environment.getProperty("os.name");  System.out.println(os);  for (String name : namesForType){  System.out.println(name);  }  Map<String, Person> persons= applicationContext.getBeansOfType(Person.class);  System.out.println(persons.toString());  } |
| Linux  linus  给容器添加Person...  {linus=com.byf.bean.Person@6b4a4e18} |

### 2.7@Import-给容器中快速导入一个组件

|  |
| --- |
| /\*\*  \* 给容器中注册组件：  \* 1）包扫描+组件标注注解（@Controller/@Service/@Repository/@Component）[自己写的]  \* 2）@Bean[导入的第三方包的组件]  \* 3）@Import：导入没有注解标注的Bean  \* 1.@Import(要导入到容器中的组件)：容器中就会自动注册这个组件  \*/  @Configuration @Conditional(WindowsCondition.class) @Import({Color.class, Shape.class}) public class MainConfig2 {  } |
| public class Color { } |
| public class Shape { } |
| ApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfig2.class);  @Test public void testImport(){  printBeans(); }  public void printBeans(){  String[] definitionNames = applicationContext.getBeanDefinitionNames();  for (String name : definitionNames)  {  System.out.println(name);  } } |
| org.springframework.context.annotation.internalConfigurationAnnotationProcessor  org.springframework.context.annotation.internalAutowiredAnnotationProcessor  org.springframework.context.annotation.internalRequiredAnnotationProcessor  org.springframework.context.annotation.internalCommonAnnotationProcessor  org.springframework.context.event.internalEventListenerProcessor  org.springframework.context.event.internalEventListenerFactory  mainConfig2  com.byf.bean.Color  com.byf.bean.Shape  person  bill |

### 2.8@Import-使用ImportBeanDefinitionRegistrar

|  |
| --- |
|  |
| // 自定义需要导入的组件 public class MyImportSelector implements ImportSelector {  // 返回值：就是要导入到容器中的组件全类名  // AnnotationMetaData：当前标注@Import注释类的所有注释信息  @Override  public String[] selectImports(AnnotationMetadata importingClassMetadata) {  return new String[]{"com.byf.bean.Heigh","com.byf.bean.Weight"};  } } |
| public class Heigh { } |
| public class Weight { } |
| @Configuration @Conditional(WindowsCondition.class) @Import({Color.class, Shape.class, MyImportSelector.class}) public class MainConfig2 {  } |
| 信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@161cd475: startup date [Sat Jul 27 10:06:42 CST 2019]; root of context hierarchy  org.springframework.context.annotation.internalConfigurationAnnotationProcessor  org.springframework.context.annotation.internalAutowiredAnnotationProcessor  org.springframework.context.annotation.internalRequiredAnnotationProcessor  org.springframework.context.annotation.internalCommonAnnotationProcessor  org.springframework.context.event.internalEventListenerProcessor  org.springframework.context.event.internalEventListenerFactory  mainConfig2  com.byf.bean.Color  com.byf.bean.Shape  com.byf.bean.Heigh  com.byf.bean.Weight  person  bill |

### 2.9@Import-使用ImportBeanDefinitionRegistrar

|  |
| --- |
|  |
| public class MyImportBeanDefinitionRegistry implements ImportBeanDefinitionRegistrar {  /\*\*  \* @param importingClassMetadata：当前类的注解信息  \* @param registry：BeanDefinition主策类  \* 把所需要添加到容器的bean：调用  \* BeanDefinitionRegistry.registryBeanDefinition手工注册进来  \*/  @Override  public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  boolean definition = registry.containsBeanDefinition("com.byf.bean.Heigh");  boolean definition2 = registry.containsBeanDefinition("com.byf.bean.Weight");  if (definition && definition2)  {  // 指定Bean定义信息：（Bean类型，Scope...）   BeanDefinition beanDefinition = new RootBeanDefinition(Body.class);  registry.registerBeanDefinition("body",beanDefinition);  }  } } |
| @Configuration @Conditional(WindowsCondition.class) @Import({Color.class, Shape.class, MyImportSelector.class, MyImportBeanDefinitionRegistry.class}) public class MainConfig2 {  } |
| ....  com.byf.bean.Heigh  com.byf.bean.Weight  person  bill  body |

### 2.10使用FactoryBean注册组件

|  |
| --- |
| // 创建一个Spring定义的工厂FactoryBean public class ColorFactoryBean implements FactoryBean<Color> {  // 返回一个对象Color，这个对象会添加到容器中  @Override  public Color getObject() throws Exception {  System.out.println("ColorFactoryBean -> getObject()");  return new Color();  }   @Override  public Class<?> getObjectType() {  return Color.class;  }   // 是否单例  // true：这个Bean在容器中保存一份  // false：多实例  @Override  public boolean isSingleton() {  return true;  } } |
| @Bean public ColorFactoryBean colorFactoryBean(){  return new ColorFactoryBean(); } |
| @Test public void testImport(){  printBeans();  // 工厂Bean获取的是调用getObject创建的对象  Object bean = applicationContext.getBean("colorFactoryBean");  Object bean2 = applicationContext.getBean("colorFactoryBean");  System.out.println("bean的类型：" + bean.getClass());  System.out.println("bean的类型：" + bean2.getClass());  // 注意运算符+号的优先级：+ 大于 ==，（bean == bean2）要加括号  System.out.println("bean == bean2 : " + (bean == bean2));  Object bean3 = applicationContext.getBean("&colorFactoryBean");  System.out.println("bean的类型：" + bean3.getClass()); } |
| colorFactoryBean  body  ColorFactoryBean -> getObject()  bean的类型：class com.byf.bean.Color  bean的类型：class com.byf.bean.Color  bean == bean2 : true  bean的类型：class com.byf.bean.ColorFactoryBean |

## 第3节生命周期

### 3.1@Bean指定初始化和销毁方法

|  |
| --- |
| <bean id="person" class="com.byf.bean.Person" init-method="" depends-on="">  <property name="name" value="张三"></property>  <property name="age" value="20"></property> </bean> |

|  |
| --- |
| /\*\*  \* bean的声明周期  \* bean创建--初始化--销毁过程  \* 容器管理bean的声明周期  \* 自定义初始化和销毁方法：容器在bean进行到当前声明舟曲的时候使用自定义初始化和销毁方法  \*  \* 构造（对象创建）  \* 单实例：在容器启动的时候创建对象（指定@Lazy时，在容器创建完成后构造）  \* 多实例：在每次获取Bean的时候创建对象  \* 初始化： \* 对象创建完成，并赋值好，调用初始化方法 \* 销毁： \* 单实例：容器关闭的时候 \* 多实例：容器不会管理这个Bean，容器不会调用销毁方法  \* 1）指定初始化和销毁方法  \* 通过@Bean指定init-method和destroy-method  \*  \*/ @Configuration public class MyConfigOfLifeCycle {  public class Car{  public Car() {  System.out.println("对象Car创建...");  }  public void init(){  System.out.println("Car ... init");  }  public void destory(){  System.out.println("Car ... destroy");  }  }  @Scope(value = "prototype")  @Bean(initMethod = "init", destroyMethod = "destory")  public Car car(){  return new Car();  } } |
| @Test public void test(){  // 1、创建IOC容器  AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(MyConfigOfLifeCycle.class);  System.out.println("容器创建完成...");  Object o = applicationContext.getBean("car");  Object o2 = applicationContext.getBean("car");  // 2、关闭IOC容器  applicationContext.close(); } |
| 容器创建完成...  对象Car创建...  Car ... init  对象Car创建...  Car ... init |

### 3.2InitializingBean和DisposableBean

|  |
| --- |
| @Component **public class** Cat **implements** InitializingBean, DisposableBean {  **public** Cat() {  System.***out***.println(**"cat ... constructor..."**);  }   @Override  **public void** destroy() **throws** Exception {  System.***out***.println(**"cat ... destory..."**);  }   @Override  **public void** afterPropertiesSet() **throws** Exception {  System.***out***.println(**"cat ... afterPropertiesSet"**);  } } |
| *\* 2）通过让Bean实现InitializingBean（定义初始化逻辑）和DisposableBean（定义销毁逻辑）  \*/* @ComponentScan(value = **"com.byf.bean"**) @Configuration **public class** MyConfigOfLifeCycle {  } |
| 七月 27, 2019 11:49:06 上午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh  信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@4141d797: startup date [Sat Jul 27 11:49:06 CST 2019]; root of context hierarchy  cat ... constructor...  cat ... afterPropertiesSet  容器创建完成...  七月 27, 2019 11:49:07 上午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose  信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4141d797: startup date [Sat Jul 27 11:49:06 CST 2019]; root of context hierarchy  cat ... destory... |

### 3.3@PostConstruct&@PreDestroy

|  |
| --- |
| *\* 3）可以使用JSR250： \** ***@PostConstruct*** *：在bean创建完成并属性赋值完成，来执行初始化方法 \** ***@PreDestroy*** *： 在容器销毁bean之前通知我们进行清理工作 \*/* |
| @Component **public class** Dog {  **public** Dog() {  System.***out***.println(**"dog constructor"**);  }  @PostConstruct  **public void** init(){  System.***out***.println(**"Dog ... @PostConstruct"**);  }  @PreDestroy  **public void** destory(){  System.***out***.println(**"Dog ... @OreDestory"**);  } } |
| 七月 27, 2019 11:56:20 上午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh  信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@4141d797: startup date [Sat Jul 27 11:56:20 CST 2019]; root of context hierarchy  dog constructor  Dog ... @PostConstruct  容器创建完成...  七月 27, 2019 11:56:20 上午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose  信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4141d797: startup date [Sat Jul 27 11:56:20 CST 2019]; root of context hierarchy  Dog ... @OreDestory |

### 3.4BeanPostProcessor-后置处理器

|  |
| --- |
| *\* 4）BeanPostProcessor【interface】：bean的后置处理器： \* 在bean的初始化前后进行一些处理工作（例如：包装bean） \* postProcessBeforeInitialization：构造之后，在初始化之前 \* postProcessAfterInitialization：在初始化之后工作 \*/* |
| *\* 构造（对象创建） \* 单实例：在容器启动的时候创建对象（指定@Lazy时，在容器创建完成后构造） \* 多实例：在每次获取Bean的时候创建对象 \* BeanPostProcessor.postProcessBeforeInitialization \* 初始化： \* 对象创建完成，并赋值好，调用初始化方法 \* BeanPostProcessor.postProcessAfterInitialization \* 销毁： \* 单实例：容器关闭的时候 \* 多实例：容器不会管理这个Bean，容器不会调用销毁方法 \** |
| */\*\*  \* 后置处理器：初始化前后  \*/* @Component **public class** MyBeanPostProcessor **implements** BeanPostProcessor {  @Override  **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(**"postProcessBeforeInitialization called..."** + beanName + **"=>"** + bean);  **return** bean;  }   @Override  **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(**"postProcessBeforeInitialization called..."** + beanName + **"=>"** + bean);  **return** bean;  } } |
| *dog constructor*  *postProcessBeforeInitialization called...dog=>com.byf.bean.life.Dog@400cff1a*  *Dog ... @PostConstruct*  *postProcessBeforeInitialization called...dog=>com.byf.bean.life.Dog@400cff1a*  *容器创建完成...*  *七月 27, 2019 12:05:09 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose*  *信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4141d797: startup date [Sat Jul 27 12:05:09 CST 2019]; root of context hierarchy*  *Dog ... @OreDestory* |

### 3.5BeanPostProcessor-原理

|  |
| --- |
| *\* \* 遍历得到容器所有的BeanPostProcessor：挨个执行beforeInitialization， \* 一旦返回null，跳出for循环，不会执行后面的BeanPostProcessor.postProcessBeforeInitialization方法 \* \* beanPostProcessor原理 \* populateBean(beanName, mbd, instanceWrapper); \* initializeBean(beanName, exposedObject, mbd){ // 给bean进行属性赋值 \* applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName); \* invokeInitMethods(beanName, wrappedBean, mbd); 执行自定义初始化 \* applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName); \* } \** |

### 3.6BeanPostProcessor在Spring底层的使用

|  |
| --- |
| @Component **public class** DogWithAppContextAware **implements** ApplicationContextAware {   **private** ApplicationContext **applicationContext**;  **public** DogWithAppContextAware() {  System.***out***.println(**"dog constructor"**);  }  @PostConstruct  **public void** init(){  System.***out***.println(**"Dog ... @PostConstruct"**);  }  @PreDestroy  **public void** destory(){  System.***out***.println(**"Dog ... @OreDestory"**);  }   @Override  **public void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {  **this**.**applicationContext** = applicationContext;  } } |
| 如何调用init方法？ |
|  |
| *\* Spring底层对 BeanPostProcessor 的使用： \* bean赋值，注入其他组件，@Autowired，声明周期注解功能，@Async，xxx 都是通过BeanPostProcessor实现 \** |

## 第4节属性赋值

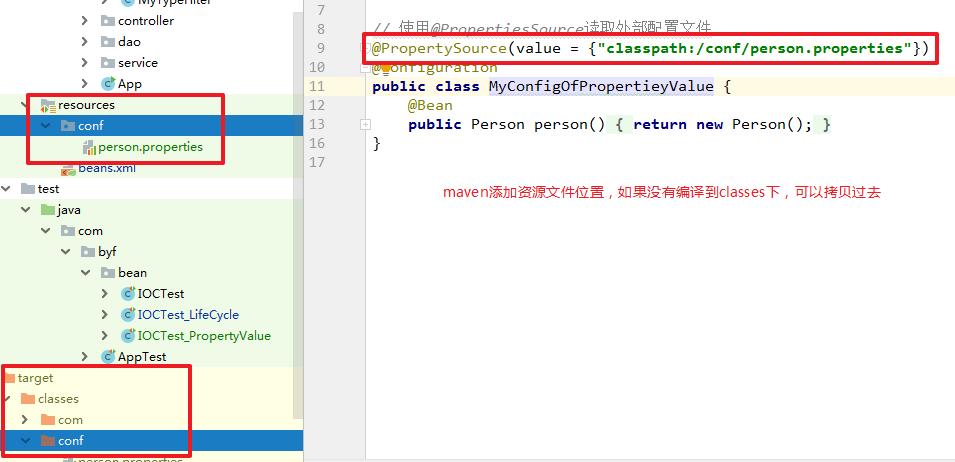
### 4.1@Value-赋值

|  |
| --- |
| *// 使用@Value进行赋值 // 1、基本数值 // 2、可以写SpEL：#{} // 3、可以写${}：取出配置文件【properties文件】中的值（在运行环境变量里面的值）* @Value(**"Tom"**) **private** String **name**; @Value(**"#{20 -2}"**) **private int age**; @Value(**"${person.nickName}"**) **private** String **nickName**; |

### 5.2@PropertySource-读取位置文件

|  |
| --- |
| **person.nickName**=**Tommy** |

|  |
| --- |
| *// 使用@PropertiesSource读取外部配置文件* @PropertySource(value = {**"classpath:/person.properties"**}) @Configuration **public class** MyConfigOfPropertieyValue {  @Bean  **public** Person person(){  **return new** Person();  } } |
| **public class** IOCTest\_PropertyValue {  *// 1、创建IOC容器* AnnotationConfigApplicationContext **applicationContext** = **new** AnnotationConfigApplicationContext(MyConfigOfPropertieyValue.**class**);   @Test  **public void** test(){  System.***out***.println(**"容器创建完成..."**);  printBeans();  Person person = (Person) **applicationContext**.getBean(**"person"**);  System.***out***.println(person);  **applicationContext**.close();  }   **public void** printBeans(){  String[] definitionNames = **applicationContext**.getBeanDefinitionNames();  **for** (String name : definitionNames)  {  System.***out***.println(name);  }  } } |
| **myConfigOfPropertieyValue**  **person**  **Person{name='Tom', age=18, nickName='Tommy'}** |



## 第5节自动装配

### 5.1@Autowired&@Qualifier&@Primary

|  |
| --- |
| */\*\*  \* 自动装配：  \* Spring利用依赖注入（DI），完成对IOC容器中各个组件的依赖关系赋值  \*  \* 1）@Autowired：自动注入  \* 1）默认优先按照类型取容器中找对应的组件：  \* applicationContext.getBean(BookService.class);  \* 2）如果找到多个相同类型的组件，再将属性的名称作为组件的id取容器中查找  \* applicationContext.getBean(BookService.class);  \* 3）@Qualifier("bookDao")：使用@Qualifier指定需要装配的组件id，而不是使用属性名  \* 4）自动装配默认一定要将属性赋值好，没有就会报错；NoSuchBeanDefinition  \* 可以使用@Autowired(required=false);  \* 5）@Primary：让Spring进行自动装配的时候默认使用首选的bean  \* 也可以使用@Qualifier指定要装配的名字  \*  \*  \*/* @Configuration @ComponentScan(value = {**"com.byf.dao"**,**"com.byf.service"**,**"com.byf.controller"**}) **public class** MyConfigOfAutowried {    @Bean(**"bookDao2"**)  **public** BookDao bookDao(){  BookDao bookDao2 = **new** BookDao();  bookDao2.setLable(**"2"**);  **return** bookDao2;  } } |
| @Service **public class** BookService {  @Qualifier(**"bookDao2"**)  @Autowired(required = **false**)  **private** BookDao **bookDao**;   **public void** print(){  System.***out***.println(**bookDao**);  }   @Override  **public** String toString() {  **return "BookService{"** +  **"bookDao="** + **bookDao** +  **'}'**;  } } |
| @Repository @Primary **public class** BookDao {  **private** String **lable** = **"1"**;   **public** String getLable() {  **return lable**;  }   **public void** setLable(String lable) {  **this**.**lable** = lable;  }   @Override  **public** String toString() {  **return "BookDao{"** +  **"lable='"** + **lable** + **'\''** +  **'}'**;  } } |
| BookService{bookDao=BookDao{lable='2'}}  BookDao{lable='2'} |

### 5.2@Resource&@Inject

|  |
| --- |
| *//@Resource(name = "bookDao2")* @Inject **private** BookDao **bookDao**; |
| *BookService{bookDao=BookDao{lable='2'}}*  *BookDao{lable='2'}* |
| *\* 2）Spring还支持使用@Resource(JSR250)和@Inject(JSR330)[java规范的注解] \** ***@Resource：*** *\* 可以和@Autowired一样实现自动装配；默认是按照组件名称进行装配； \* 没有能支持@Primary功能，没有支持@Autowired(required=false); \** ***@Inject：*** *\* 需要导入javax.inject的依赖，和Autowired的功能一样，没有required=false的功能； \** ***@Autowired：*** *\* Spring定义的：@Resource、@Inject都是java规范 \*/* |

### 5.3方法、构造器位置的自动装配

|  |
| --- |
| *\* AutoWiredAnnotationBeanPostProcessor：解析完成自动装配功能 \*  \* 3）@Autowired：构造器，参数，方法，属性，都是从容器中取参数组件的值 \* 1）【标注在方法位置】：@Bean+方法参数：参数从容器中获取参数组件的值 \* 2）【标注在构造器上】：如果组件只有一个有参构造器，这个有参构造器的@Autowired可以省略，参数位置的组件自动从容器中获取 \* 3）【放在参数位置】 \*/* |

|  |
| --- |
| *//构造器要使用的组件，都是从容器中获取* **public** Boss(@Autowired Car car){  **this**.**car** = car; }  *//@Autowired //标注在方法：Spring容器创建当前对象，就会调用方法，完成赋值； // 方法使用的参数，自定义类型的值从ioc容器中获取* **public void** setCar(Car car) {  **this**.**car** = car; }  */\*\*  \** ***@Bean*** *标注的方法创建对象的时候，方法参数的值从容器中获取  \** ***@param car*** *\** ***@return*** *\*/* @Bean **public** Color color(Car car){  Color color = **new** Color();  color.setCar(car);  **return** color; } |
| *Boss{car=com.byf.bean.Car@184cf7cf}*  *com.byf.bean.Car@184cf7cf*  *Color{car=com.byf.bean.Car@184cf7cf}* |

### 5.4Awired注入Spring底层组件

|  |
| --- |
| @Component **public class** Red **implements** ApplicationContextAware,  BeanNameAware, EmbeddedValueResolverAware {  **private** ApplicationContext **applicationContextAware**;  @Override  **public void** setBeanName(String name) {  System.***out***.println(**"当前bean的名字："** + name);  }   @Override  **public void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {  **this**.**applicationContextAware** = applicationContext;  System.***out***.println(**"传入的IOC容器："** + applicationContext);  }   @Override  **public void** setEmbeddedValueResolver(StringValueResolver resolver) {  String str = resolver.resolveStringValue(**"${os.name} + #{20 + 18}"**);  System.***out***.println(**"解析的字符串："** + str);  } } |
| Red red = applicationContext.getBean(Red.**class**); System.***out***.println(red); |
| 当前bean的名字：red  解析的字符串：Windows 7 + 38  传入的IOC容器：org.springframework.context.annotation.AnnotationConfigApplicationContext@424c0bc4: startup date [Sat Jul 27 23:52:52 CST 2019]; root of context hierarchy  com.byf.bean.Red@43195e57 |