# 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

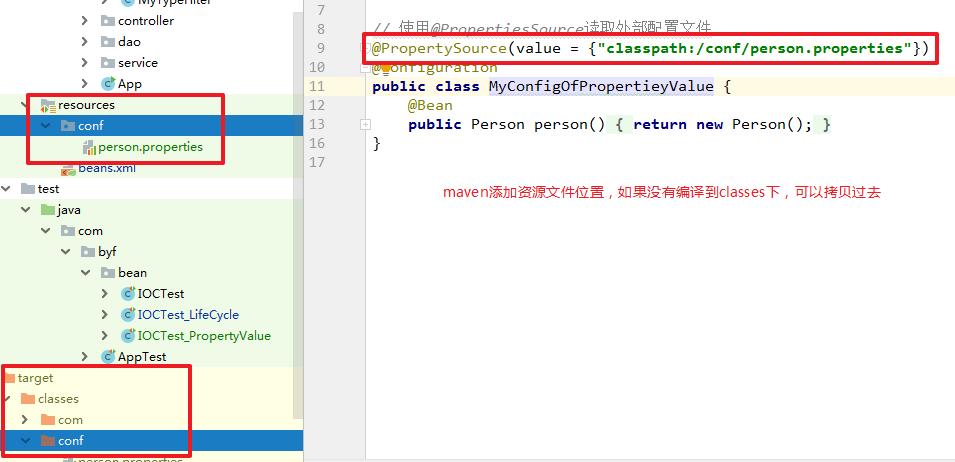
### 4.1@Value-赋值

|  |
| --- |
| *// 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 |

### 5.5@Profile环境搭建

|  |
| --- |
| */\*\*  \* Profile:  \* Spring提供的可以根据当前环境，动态的激活和切换一系列组件的功能  \*  \* 开发环境、测试环境、生产环境  \* 数据源（/A）（/B）（/C）  \*  \** ***@Profile*** *\* 指定组件在哪个环境下，才能被注册到容器中，不指定，任何环境下都能注册组件  \* 1）加载了环境标识的bean，只有这个环境被激活的时候才能注册到容器中，默认是default环境  \* 2）写在配置类上，只有指定了环境的时候，整个配置类里面的所有配置生效  \* 3）没有标注环境标识的bean，任何环境都是加载的  \*/* @Profile(**"test"**) @PropertySource(**"classpath:/conf/dbconfig.properties"**) @Configuration **public class** MyConfigOfProfile **implements** EmbeddedValueResolverAware {   @Value(**"${db.user}"**)  **private** String **user**;   **private** StringValueResolver **valueResolver**;   @Profile(**"test"**)  @Bean(**"testDataSource"**)  **public** DataSource dataSourceTest(@Value(**"${db.password}"**) String pwd) **throws** PropertyVetoException {  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser(**user**);  dataSource.setPassword(pwd);  dataSource.setJdbcUrl(**"jdbc:mysql://localhost:3306/user\_test"**);  dataSource.setDriverClass(**"com.mysql.jdbc.Driver"**);  **return** dataSource;  }   @Profile(**"dev"**)  @Bean(**"devDataSource"**)  **public** DataSource dataSourceDev(@Value(**"${db.password}"**) String pwd) **throws** PropertyVetoException {  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser(**user**);  dataSource.setPassword(**"1234"**);  dataSource.setJdbcUrl(**"jdbc:mysql://localhost:3306/user"**);  dataSource.setDriverClass(**"com.mysql.jdbc.Driver"**);  **return** dataSource;  }   @Profile(**"prod"**)  @Bean(**"prodDataSource"**)  **public** DataSource dataSourceProd(@Value(**"${db.password}"**) String pwd) **throws** PropertyVetoException {  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser(**user**);  dataSource.setPassword(**"1234"**);  String driverClass = **valueResolver**.resolveStringValue(**"${db.driverClass}"**);  dataSource.setJdbcUrl(**"jdbc:mysql://localhost:3306/user\_prod"**);  dataSource.setDriverClass(driverClass);  **return** dataSource;  }   *// 不单独指定Profile，类上有标注，默认加载* @Bean  **public** Red red(){  **return new** Red();  }   @Override  **public void** setEmbeddedValueResolver(StringValueResolver resolver) {  **this**.**valueResolver** = resolver;  } } |

### 5.6@Profile根据环境注册bean

|  |
| --- |
| **public class** IOCTest\_Profile {   *// 1、使用命令行动态设置环境变量：在虚拟器参数位置加载 -Dspring.profiles.active=dev  // 2、创建IOC容器时设置环境变量* @Test  **public void** test(){  *// 1、创建IOC容器* AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext();  *// 2、设置需要激活的环境* applicationContext.getEnvironment().setActiveProfiles(**"test"**);  *// 3、注册主配置类* applicationContext.register(MyConfigOfProfile.**class**);  *// 4、启动刷新容器* applicationContext.refresh();  System.***out***.println(**"容器创建完成..."**);   printBeans(applicationContext);  applicationContext.close();  }   **public void** printBeans(AnnotationConfigApplicationContext applicationContext){  String[] definitionNames = applicationContext.getBeanDefinitionNames();  **for** (String name : definitionNames)  {  System.***out***.println(name);  }  } } |
| **......**  **myConfigOfProfile**  **testDataSource**  **red** |

### 5.7IOC总结

## 第6节AOP

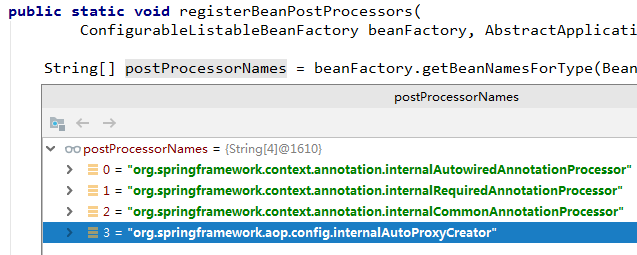
### 6.1AOP功能测试

|  |
| --- |
| **public class** MathCalculator {  **public int** div(**int** i,**int** j){  System.***out***.println(**"Calculator ... "** + i + **"/"** + j + **" = ?"**);  **return** i / j;  } } |
| @Aspect **public class** LogAspects {  *// 抽取公共的切入点表达式  // 1、本类引用  // 2、其他的却面引用com.byf.aop.MathCalculator.div* @Pointcut(**"execution(public int com.byf.aop.MathCalculator.\*(..))"**)  **public void** pointCut(){};   *// @Before：在目标方法之前切入：切入点表达式（指定在哪个方法切入）  //@Before("public int com.byf.aop.MathCalculator.\*(..)")* @Before(**"com.byf.aop.LogAspects.pointCut()"**)  **public void** logStart(JoinPoint joinPoint){  Object[] args = joinPoint.getArgs();  System.***out***.println(**""**+ joinPoint.getSignature().getName() +**"运行：参数列表是{"**+ Arrays.*asList*(args)+**"}"**);  }  @After(**"pointCut()"**)  **public void** logEnd(JoinPoint joinPoint){  System.***out***.println(**""**+ joinPoint.getSignature().getName() +**"结束！"**);  }  @AfterReturning(value = **"pointCut()"**,returning = **"result"**)  **public void** logReturn(JoinPoint joinPoint, Object result){  System.***out***.println(**""**+ joinPoint.getSignature().getName() +**"正常返回：运行结果是："** + result);  }  @AfterThrowing(value = **"pointCut()"**,throwing = **"exception"**)  **public void** logException(JoinPoint joinPoint, Exception exception){  System.***out***.println(**""**+ joinPoint.getSignature().getName() +**"异常：异常结果："** + exception);  } } |
| */\*\*  \* AOP：【动态代理】  \* 只在程序运行期间动态的将某段代码注入到指定方法指定位置进行运行的编程  \*  \* 1、导入aop模块依赖：Spring AOP（spring-aspects）  \* 2、定义一个业务逻辑类（MathCalculator）：在业务逻辑运行的时候  \* 3、定义一个日志切面（LogAspects）：切面里的方法需要动态感知MathCalculator.div运行到  \* 通知方法：  \* 前置通知(@Before)：logStart：在目标方法(div)运行之前运行  \* 后置通知(@After)：logEnd：在目标方法(div)运行结束之后运行  \* 返回通知(@AfterReturning)：logReturn：在目标方法(div)正常返回之后运行  \* 异常通知(@AfterThrowing)：logException：在目标方法(div)出现异常以后运行  \* 环绕通知(@Around)：动态代理，手动推进目标方法运行（joinPoint.proceed())  \* 4、给切面类的目标方法标注何时何地运行（通知注解）  \* 5、将切面类和业务逻辑类（目标方法所在类）加入到容器  \* 6、必须告诉Spring切面类是谁，@Aspect  \* 7、给配置类加@EnableAspectJAutoProxy【开启基于注解的aop模式】  \* 在Spring中很多的@EnableXXX  \*  \* 三步：  \* 1）将业务逻辑组件和切面类都加入到容器中：告诉Spring哪个是切面类（@Aspect）  \* 2）在切面类的每一个通知方法上标注通知注解，告诉Spring何时何地执行（切入点表达式）  \* 3）开启基于注解的aop模式：  \** ***@EnableAspectJAutoProxy*** *\*/* @EnableAspectJAutoProxy @Configuration **public class** MyConfigOfAOP {  @Bean  **public** MathCalculator calculator(){  **return new** MathCalculator();  }   @Bean  **public** LogAspects logAspects(){  **return new** LogAspects();  } } |
| @Test **public void** test(){  *// 1、创建IOC容器* AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MyConfigOfAOP.**class**);  System.***out***.println(**"容器创建完成..."**);  **final** MathCalculator calculator = applicationContext.getBean(MathCalculator.**class**);  **try**{  calculator.div(5,0);  } **catch** (Exception e){  System.***out***.println(**"java.lang.ArithmeticException: / by zero"**);  }  applicationContext.close(); } |
| 容器创建完成...  div运行：参数列表是{[5, 0]}  Calculator ... 5/0 = ?  div结束！  div异常：异常结果：java.lang.ArithmeticException: / by zero  java.lang.ArithmeticException: / by zero |

### 6.2@EnableAspectJAutoProxy

|  |
| --- |
| @Import(AspectJAutoProxyRegistrar.**class**) **public** @**interface** EnableAspectJAutoProxy { |
| *\* AOP原理：【给容器注册了哪些组件，组件是么时候工作，组件的功能是什么】*  *\* 1、@EnableAspectJAutoProxy是什么？ \** ***@Import（AspectJAutoProxyRegistrar.class)：给容器导入AspectJAutoProxyRegistrar*** *\* 利用AspectJAutoProxyRegistrar自定义给容器中注册bean \* internalAutoProxyCreator = AnnotationAwareAspectJAutoProxyCreator \* 给容器中注册一个容器代理创建器：AnnotationAwareAspectJAutoProxyCreator（AspectJ自动代理创建器） \* \* 2、AnnotationAwareAspectJAutoProxyCreator \* ->AspectJAwareAdvisorAutoProxyCreator \* ->AbstractAdvisorAutoProxyCreator \* ->AbstractAutoProxyCreator \* implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware \* 关注后置处理器（在bean初始化完成前后做事情）、自动装配BeanFactory \** |

### 6.3注册AnnotationAwareAspectJAutoProxyCreator



|  |
| --- |
| *\* 流程： \* 1）传入配置类：创建IOC容器 \* 2）注册配置类：调用refresh()刷新容器 \* 3）invokeBeanFactoryPostProcessors(beanFactory);注册bean的后置处理器来方便连接bean的创建 \* 1）先获取ioc容器已经定义了的需要创建对象的所有BeanPostProcessor \* 2）给容器中添加别的BeanPostProcessor \* 3）优先注册实现了PriorityOrdered接口的BeanPostProcessor \* 4）再给容器注册实现了Ordered接口的BeanPostProcessor \* 5）注册没实现优先级接口的BeanPostProcessor \* 6）注册BeanPostProcessor，实际上就是创建BeanPostProcessor对象，保存在容器中 \* 创建internalAutoProxyCreator的BeanPostProcessor【AnnotationAwareAspectJAutoProxyCreator】 \* 1）创建bean实例 \* 2）populateBean：给bean的各种属性赋值 \* 3）initializeBean：初始化bean； \* 1）invokeAwareMethods()：处理Aware接口的方法回调 \* 2）applyBeanPostProcessorsBeforeInitialization()：应用后置处理器的processor.postProcessBeforeInitialization \* 3）invokeInitMethods()：执行自定义初始化方法 \* 4）applyBeanPostProcessorsAfterInitialization()：应用后置处理器的processor.postProcessAfterInitialization \* 4）BeanPostProcessor(AnnotationAwareAspectJAutoProxyCreator)：创建成功：aspectJAdvisorFactory、aspectJAdvisorsBuilder \* 7）把beanPostProcessor注册到beanFactory中： \* beanFactory.addBeanPostProcessor(beanPostProcessor) \* -----------------以上是创建和注册AnnotationAwareAspectJAutoProxyCreator的过程--------------* |

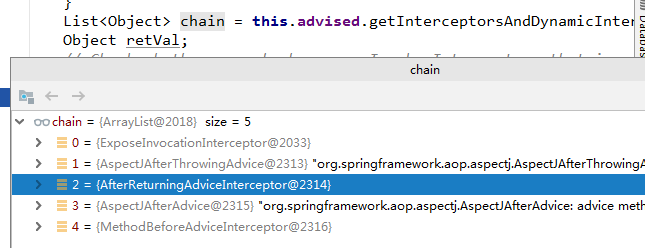
### 6.4AnnotationAwareAspectJAutoProxyCreator执行时机

|  |
| --- |
| *\* 4）finishBeanFactoryInitialization(beanFactory)：完成beanFactory初始化工作，创建剩下的单实例bean \* 1）遍历获取容器中所有的bean，依次创建对象getBean(beanName); \* getBean->doGetBean->getSingleton(beanName) \* 2）创建bean： \* 【AnnotationAwareAspectJAutoProxyCreator在所有bean创建之前会拦截，InstantiationAwareBeanPostProcessor， \* 会调用AbstractAutoProxyCreator的postProcessBeforeInstantiation()】 \* 1）先从缓存中获取当前bean，如果能获取到，说明bean是之前被创建的bean，直接调用，否则创建 \* 只要创建好的Bean会被缓存起来 \* 2）createBean()：创建bean：AnnotationAwareAspectJAutoProxyCreator在任何创建Bean之前返回bean的实例 \* 【BeanPostProcessor：在Bean对象创建完成，初始化前后调用】 \* 【InstantiationAwareBeanPostProcessor：在创建Bean实例之前先尝试调用后置处理器返回对象的】 \* resolveBeforeInstantiation(beanName, mbdToUse);解析BeforeInstantiation \* 希望后置处理器能再次返回一个代理对象，如果能返回代理对象就返回，如果不能继续 \* 1）后置处理器先尝试返回对象 \* bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName); \* 拿到所有后置处理器：如果是InstantiationAwareBeanPostProcessor； \* 就执行postProcessBeforeInstantiation* |
|  |

### 6.5创建AOP代理

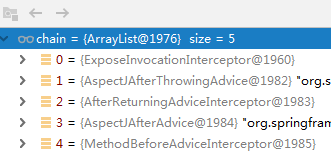
|  |
| --- |
| *\* AnnotationAwareAspectJAutoProxyCreator \* 1）每一个bean创建之前，调用postProcessBeforeInstantiation： \* 关心MathCalculator和LogAspects的场景 \* 1）判断当前bean是否在advisedBeans中（保存了所有需要增强bean） \* 2）判断当前bean是否是基础类型Advice/Pointcut/Advisor/AopInfrastructureBean \* 或者是否为切面（@Aspect） \* 3）是否需要跳过 \* 1）获取候选的增强（切面里面的通知方法）【List<Advisor> candidateAdvisors】 \* 每一个封装的通知方法的增强器是InstantiationModelAwarePointcutAdvisor类型 \* 判断每一个增强器是否是AspectJPointcutAdvisor类型：返回true \* 2）永远返回false \* \* 2）创建对象 \* postProcessAfterInitialization \* return wrapIfNecessary(bean, beanName, cacheKey); \* 1）获取当前bean的所有增强器（通知方法） \* 1）找到能在当前bean使用的增强器（找那些通知方法时需要切入当前bean方法的） \* 2）获取能够在bean使用的增强器 \* 3）给增强器排序 \* 2）保存当前bean在advisedBeans中 \* 3）如果当前bean需要增强，创建当前bean的代理对象 \* 1）获取所有增强器（通知方法） \* 2）保存到proxyFactory \* 3）创建代理对象：Spring自动决定 \* if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) { \* return new JdkDynamicAopProxy(config); // jdk动态代理 \* } \* return new ObjenesisCglibAopProxy(config); // cglib的动态代理 \* 4）给容器中返回当前组件使用cglib增强了的代理对象 \* 5）以后容器中获取到的这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程 \* \* \*/* |

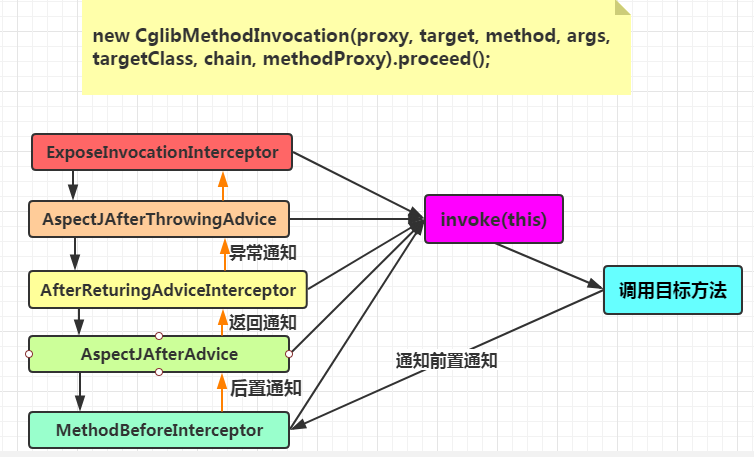
### 6.6获取拦截器链-MethodInterceptor



|  |
| --- |
| *\* 3）目标方法执行： \* 容器中保存了组件的代理对象（cglib增强后的对象），这个对象里面保存了详细信息（比如增强器，目标对象：xxx） \* 1）CglibAopProxy.intercept();拦截目标方法的执行 \* 2）根据ProxyFactory对象获取将要执行的目标方法拦截器 \* List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass); \* 1）List<Object> interceptorList保存所有拦截器 5 \* 一个默认的ExposeInvocationInterceptor 和 4个增强器 \* 2）遍历所有的增强器，将其转换为Interceptor： \* registry.getInterceptors(advisor)； \* 3）将增强器转换为List<MethodInterceptor>; \* 如果是MethodInterceptor：直接加入到集合中； \* 如果不是，使用AdvisorAdapter将增强器转换为MethodInterceptor \* 转换完成，返回MethodInterceptor数组 \* 3）如果没有拦截器链，直接执行目标方法： \* 拦截器链（每一个通知方法又被包装为方法拦截器，利用MethodInterceptor机制） \* 4）如果有拦截器链：把需要执行的目标对象，目标方法，拦截器链等信息传入一个CglibMethodInvocation对象 \* 并调用Object retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy).proceed(); \** |

### 6.7链式调用通知方法





|  |
| --- |
| *5）拦截器链的除法过程： \* 1）如果没有拦截器执行目标方法，或者拦截器的所有和拦截器数组-1大小一样（指定了最后一个拦截器）执行目标方法； \* 2）链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成返回以后再来执行： \* 拦截器机制：保证通知方法与目标方法的执行顺序* |

### 6.8AOP原理总结

|  |
| --- |
| *\* 总结： \* 1）@EnableAspectJAutoProxy 开启AOP功能 \* 2）@EnableAspectJAutoProxy 给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreater \* 3）AnnotationAwareAspectJAutoProxyCreater是一个后置处理器； \* 4）容器的创建流程： \* 1）registerBeanPostProcessors（）注册后置处理器：创建 AnnotationAwareAspectJAutoProxyCreator 对象 \* 2）finishBeanFactoryInitialization(beanFactory); 初始化剩下的单实例bean \* 1）创建业务逻辑组件和切面组件 \* 2）AnnotationAwareAspectJAutoProxyCreator 拦截组件的创建过程 \* 3）组件创建完成之后，判断组件是否需要增强； \* 是：切面的通知方法，包装成增强器（Advisor）；给业务逻辑创建一个代理对象（CGlib）； \* 5）执行目标方法： \* 1）代理对象执行目标方法 \* 2）CglibA拦截器opProxy.intercept() \* 1）得到目标方法的拦截器（增强器包装成MethodInterceptor） \* 2）利用拦截器的链式机制，一次进入每一个拦截器进行执行 \* 3）效果： \* 正常执行：前置通知-》目标方法-》后置通知-》返回通知 \* 异常执行：前置通知-》目标方法-》后置通知-》异常通知 \** |

## 第7节声明式事务

### 7.1环境搭建

|  |
| --- |
| */\*\*  \* 声明式事务：  \*  \* 环境搭建：  \* 1）导入相关依赖  \* 数据源、数据库驱动、Spring-jdbc模块  \* 2）配置数据源、JdbcTemplate（Spring提供的简化数据库操作的工具）操作数据源  \* 3）给方法上标注@Transactional 标识当前方法时一个事务方法；  \* 4）@EnableTransactionManagement 开启基于注解的事务管理功能  \** ***@Enablexxx*** *\* 5）配置事务管理器来控制事务；  \** ***@Bean*** *\* public PlatformTransactionManager transactionManager()  \*  \*/* @EnableTransactionManagement @ComponentScan(value = **"com.byf.tx"**) @Configuration **public class** TxConfig {  @Bean  **public** DataSource dataSource() **throws** PropertyVetoException {  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser(**"root"**);  dataSource.setPassword(**"1234"**);  dataSource.setDriverClass(**"com.mysql.cj.jdbc.Driver"**);  dataSource.setJdbcUrl(**"jdbc:mysql://localhost:3306/user?useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=UTC"**);  **return** dataSource;  }   @Bean  **public** JdbcTemplate jdbcTemplate() **throws** PropertyVetoException {  *// Spring对@Configuration类会特殊处理：给容器中加组件的方法，多次调用都只是从容器中找组件* JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource());  **return** jdbcTemplate;  }   *// 注册事务管理器到容器中* @Bean  **public** PlatformTransactionManager transactionManager() **throws** PropertyVetoException {  **return new** DataSourceTransactionManager(dataSource());  } } |
| @Repository **public class** UserDao {  @Autowired  **private** JdbcTemplate **jdbcTemplate**;   @Transactional  **public void** insert(){  String sql = **"insert into user(name,phone) values(?,?)"**;  String user = UUID.*randomUUID*().toString().substring(0,5);  **jdbcTemplate**.update(sql,user, 18705193697L);  System.***out***.println(**"插入完成"**);  **int** i = 1 /0;  } } |
| **public class** IOCTest\_tx {   @Test  **public void** test(){  ApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(TxConfig.**class**);  UserDao userDao = applicationContext.getBean(UserDao.**class**);  printBeans(applicationContext);  userDao.insert();  }   **public void** printBeans(ApplicationContext applicationContext){  String[] definitionNames = applicationContext.getBeanDefinitionNames();  **for** (String name : definitionNames)  {  System.***out***.println(name);  }  } } |
| **txConfig**  **userDao**  **userService**  **org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration**  **org.springframework.transaction.config.internalTransactionAdvisor**  **transactionAttributeSource**  **transactionInterceptor**  **org.springframework.transaction.config.internalTransactionalEventListenerFactory**  **dataSource**  **jdbcTemplate**  **transactionManager**  **org.springframework.aop.config.internalAutoProxyCreator**  **......**  **插入完成**  **java.lang.ArithmeticException: / by zero** |

### 7.2声明式事务原理

|  |
| --- |
| *\* 原理： \* 1）@EnableTransactionManagement \* 利用TransactionManagementConfigurationSelector给容器中导入组件 \* 给容器中导入两个组件： \* AutoProxyRegistrar、ProxyTransactionManagementConfiguration \* 2）AutoProxyRegistrar：给容器中注册一个 InfrastructureAdvisorAutoProxyCreator 组件； \* InfrastructureAdvisorAutoProxyCreator： \* 利用后置处理器机制在对象创建后，包装对象，返回一个代理对象（增强器），代理对象执行方法利用拦截器链进行调用 \* 3）ProxyTransactionManagementConfiguration： \* 1）给容器中注册事务增器： \* 1）事务增强器要用事务注解的信息，AnnotationTransactionAttributeSource 解析事务 \* 2）事务拦截器： \* TransactionInterceptor：保存了事务属性信息，事务管理器 \* 他是一个 MethodInterceptor： \* 在目标方法执行的时候： \* 执行拦截方法； \* 事务拦截器： \* 1）先获取事务相关属性 \* 2）再获取 PlatformTransactionManager ，如果事务没有添加指定的tr则使用平台默认的事务管理器 \* 最终会从容器中按照类型 PlatformTransactionManager \* 3）执行目标方法： \* 如果异常，获取事务管理器，利用事务管理回滚操作： \* txInfo.getTransactionManager().rollback \* 如果正常，利用事务管理的提交操作： \* txInfo.getTransactionManager().commit \* \** |

## 第8节扩展原理

### 8.1BeanFactoryPostProcessor Bean工厂后置处理

|  |
| --- |
| */\*\*  \* 扩展原理：  \* BeanPostProcessor： bean后置处理器，bean创建对象初始化前后进行拦截工作的  \* BeanFactoryPostProcessor：beanFactory 的后置处理器  \* 在 beanFactory 标准初始化之后调用：所有的bean定义已经保存加载到 beanFactory  \* 1）ioc容器创建对象  \* 2）invokeBeanFactoryPostProcessors(beanFactory); 执行BeanFactoryPostProcessors  \* 如何找到所有BeanFactoryPostProcessors并执行他们的方法：  \* 1）直接在BeanFactory中找到所有类型是BeanFactoryPostProcessor的组件，并执行他们的方法  \* 2）在初始化创建其他组件前面执行  \*/* @ComponentScan(value = **"com.byf.ext"**) @Configuration **public class** ExtConfig {  @Bean  **public** Car red(){  **return new** Car();  } } |
| @Component **public class** MyBeanFactoryPostProcessor **implements** BeanFactoryPostProcessor {  @Override  **public void** postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {  System.***out***.println(**"MyBeanFactoryPostProcessor ... postProcessBeanFactory"**);  **int** count = beanFactory.getBeanDefinitionCount();  String[] names = beanFactory.getBeanDefinitionNames();  System.***out***.println(**"当前BeanFactory中有"** + count + **"个Bean"**);  System.***out***.println(Arrays.*asList*(names));  } } |
| MyBeanFactoryPostProcessor ... postProcessBeanFactory  当前BeanFactory中有9个Bean  [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, extConfig, myBeanFactoryPostProcessor, car]  ......  Car ... Constructor |

### 8.2BeanDefinitionRegistryPostProcessor Bean注册中心后置处理

|  |
| --- |
| *\* 2、BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor \* postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry); \* 在所有bean定义信息将要加载，bean实例还未创建 \* \* 优先于BeanFactoryPostProcessor执行， \* 利用BeanDefinitionRegistryPostProcessor给容器中再额外添加一些bean \* \* 原理： \* 1）ioc创建容器 \* 2）refresh() -> invokeBeanFactoryPostProcessors(beanFactory) \* 3）从容器中获取到所有 BeanDefinitionRegistryPostProcessor 组件 \* 1、依次触发所有的 postProcessBeanDefinitionRegistry() 方法 \* 2、再来触发 postProcessBeanFactory() 方法 BeanFactoryPostProcessor \* 4）再来从容器找到BeanFactoryPostProcessor组件，依次触发触发postProcessBeanFactory()方法 \* \*/* |
| @Component **public class** MyBeanDefinitionRegistryPostProcessor **implements** BeanDefinitionRegistryPostProcessor {  @Override  **public void** postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) **throws** BeansException {  System.***out***.println(**"postProcessBeanDefinitionRegistry...bean的数量："** + registry.getBeanDefinitionCount());  *// RootBeanDefinition beanDefinition = new RootBeanDefinition(Red.class);* AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.*rootBeanDefinition*(Red.**class**).getBeanDefinition();  registry.registerBeanDefinition(**"hello"**,beanDefinition);  }  *// BeanDefinitionRegistry Bean定义信息的保存中心 ，  // 以后BeanFactory 就是按照BeanDefinitionRegistry里面保存的每一个bean的定义信息创建Bean* @Override  **public void** postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {  System.***out***.println(**"MyBeanDefinitionRegistryPostProcessor...bean的数量："**+beanFactory.getBeanDefinitionCount());  } } |
|  |

### 8.3ApplicationListener事件监听

|  |
| --- |
| *\* 3、ApplicationListener：建ring容器中发布的时间。事件驱动模型开发 \* public interface ApplicationListener<E extends ApplicationEvent> extends EventListener \* 监听ApplicationEvent 及其下面的子事件； \* \* 步骤： \* 1）写一个事件监听器来监听某个事件（ApplicationEvent及其子类） \* 2）把监听器加入容器 \* 3）只要容器中有相关事件的发布，我们就能监听到这个事件： \* ContextRefreshedEvent：容器刷新完成（所有bean都完全创建）会发布这个事件； \* ContextClosedEvent：关闭容器会发布这个事件； \* 4）发布一个事件 \** |

|  |
| --- |
| @Component **public class** MyApplicationListener **implements** ApplicationListener<ApplicationEvent> {  @Override  **public void** onApplicationEvent(ApplicationEvent event) {  System.***out***.println(**"收到事件："** + event);  } } |
| @Test **public void** test(){  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(ExtConfig.**class**);  *// 发布事件：* applicationContext.publishEvent(**new** ApplicationEvent(**new** String(**"我发布了一个事件"**)) {  });  applicationContext.close(); } |
| 传入的IOC容器：org.springframework.context.annotation.AnnotationConfigApplicationContext@2a098129: startup date [Tue Jul 30 22:34:06 CST 2019]; root of context hierarchy  收到事件：org.springframework.context.event.ContextRefreshedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationContext@2a098129: startup date [Tue Jul 30 22:34:06 CST 2019]; root of context hierarchy]  收到事件：com.byf.bean.IOCTest\_Ext$1[source=我发布了一个事件]  七月 30, 2019 10:34:07 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose  信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a098129: startup date [Tue Jul 30 22:34:06 CST 2019]; root of context hierarchy  收到事件：org.springframework.context.event.ContextClosedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationContext@2a098129: startup date [Tue Jul 30 22:34:06 CST 2019]; root of context hierarchy] |

|  |
| --- |
| *\* 原理： \* ContextRefreshedEvent、com.byf.bean.IOCTest\_Ext$1、ContextClosedEvent \* 1）ContextRefreshedEvent事件： \* 1）容器创建对象：refresh() \* 2）finishRefresh();容器刷新完成就会发布ContextRefreshedEvent事件 \* 2）自己发布事件 \* 3）容器关闭会发布ContextClosedEvent事件： \* \* 【事件发布流程】： \* 3）publishEvent(new ContextRefreshedEvent(this)); \* 1）获取事件多播器派发事件（派发器）：getApplicationEventMulticaster() \* 2）multicastEvent派发事件 \* 3）获取到所有ApplicationListener：getApplicationListeners(); \* for (final ApplicationListener*<?> *listener : getApplicationListeners(event, type)) { \* 1）如果有Executor：可以使用Executor异步进行派发； \* Executor executor = getTaskExecutor(); \* 2）否则同步的方式直接执行listener方法：invokeListener(listener, event); \* 拿到listener回调 listener.onApplicationEvent(event); \* 【事件多播器(派发器)】 \* 1）容器创建对象：refresh(); \* 2）initApplicationEventMulticaster();初始化多播器ApplicationEventMulticaster \* 1）先去容器中找有没有id="applicationEventMulticaster"的组件 \* if (beanFactory.containsLocalBean(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME)) \* 2）如果没有：this.applicationEventMulticaster = \* beanFactory.getBean(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME, ApplicationEventMulticaster.class); \* 并加入到容器中：我们就可以在其他组件要派发事件，自动注入这个applicationEventMulticaster \* 【容器中有哪些监听器】 \* 1）容器创建对象：refresh(); \* 2）registerListener(); \* for (ApplicationListener*<?> *listener : getApplicationListeners()) { \* getApplicationEventMulticaster().addApplicationListener(listener); \* } \** |

### 8.4@EventListener && SmartInitializingSingleton

|  |
| --- |
| *\* 1）写一个事件监听器(实现ApplicationListener<ApplicationEvent>接口)来监听某个事件（ApplicationEvent及其子类） \* #@EventListener； \* 原理：使用EventListenerMethodProcessor处理器解析方法上的@EventListener； \**  *\* SmartInitializingSingleton 原理： \* 1）ioc容器创建对象并refresh(); \* 2）finishBeanFactoryInitialization(beanFactory);初始化剩下的单实例bean \* 1）先创建所有的单实例bean：getBean(); \* 2）获取所有创建好的单实例bean，判断是否是SmartInitializingSingleton类型： \* if (singletonInstance instanceof SmartInitializingSingleton) { \* 如果是就调用：smartSingleton.afterSingletonsInstantiated(); \* } \** |

## 第9节Spring容器创建

### 9.1BeanFactory预准备

|  |
| --- |
| Spring容器refresh【创建刷新】 1、prepareRefresh()刷新前的预处理；  1）initPropertySources();初始化一些属性配置；子类自定义个性化的属性设置方法；  2）getEnvironment().validateRequiredProperties();验证属性的合法性等  3）this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();  保存容器中的一些早起事件 2、obtainFreshBeanFactory();获取BeanFactory  1）refreshBeanFactory();刷新【创建】BeanFactory；  创建了一个this.beanFactory = new DefaultListableBeanFactory();  设置id；this.beanFactory.setSerializationId(getId());  2）getBeanFactory();返回刚才GenericApplicationContext创建的BeanFactory对象  3）将创建的BeanFactory【DefaultListableBeanFactory】返回 3、prepareBeanFactory(beanFactory);BeanFactory的预准备工作（BeanFactory进行一些设置）；  1）设置BeanFactory的类加载器，支持表达式解析器...  2）添加BeanPostProcessor【ApplicationContextAwareProcessor】  3）设置忽略的自动装配的接口：EnvironmentAware、EmbeddedValueResolverAware...  接口的实现类，不能通过接口类型自动注入，默认自动忽略  4）注册可以解析的自动装配：能直接在任何组件中自动注入BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext  5）添加后置处理器：addBeanPostProcessor(new ApplicationListenerDetector(this))  6）添加编译时的AspectJ支持；  7）给BeanFactory注册一些能用的组件：（可以通过@Autowired注入）  environment【ConfigurableEnvironment】；  systemProperties【Map<String, Object>】；  systemEnvironment【Map<String, Object>】; 4、postProcessBeanFactory(beanFactory);BeanFactory准备工作完成后进行的后置处理工作；  1）子类通过重写这个方法来在BeanFactory创建预准备完成后做进一步的设置 ======================以上是BeanFactory的创建及预准备工作=============================  5、invokeBeanFactoryPostProcessors(beanFactory);执行BeanFactoryPostProcessor  BeanFactoryPostProcessor ：BeanFactory的后置处理器。在BeanFactory准备  两个接口：BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor  1）执行 BeanFactoryPostProcessor 的方法：  1）获取所有BeanDefinitionRegistryPostProcessor；  2）看先执行实现了PriorityOrdered优先级的BeanDefinitionRegistryPostProcessor  postProcessor.postProcessBeanDefinitionRegistry(registry);  3）再执行实现了Ordered殊勋接口的BeanDefinitionRegistryPostProcessor;  postProcessor.postProcessBeanDefinitionRegistry(registry);  4）最后执行没有实现任何优先级或者顺序接口的BeanDefinitionRegistryPostProcessor；  postProcessor.postProcessBeanFactory(beanFactory);  6、registerBeanPostProcessors(beanFactory);注册 BeanPostProcessor （Bean的后置处理器）【拦截bean创建：intercept bean creation.】  不同接口类型的BeanPostProcessor：在Bean创建前后的执行时机不一样的  BeanPostProcessor、  DestructionAwareBeanPostProcessor、  InstantiationAwareBeanPostProcessor、  SmartInstantiationAwareBeanPostProcessor、  MergedBeanDefinitionPostProcessor、   1）获取所有的 BeanPostProcessor；后置处理器都默认可以通过PriorityOrdered、Ordered 接口来指定优先级  2）先注册PriorityOrdered优先级接口的BeanPostProcessor；  把每一个BeanPostProcessor添加到BeanFactory中  beanFactory.addBeanPostProcessor(postProcessor);  3）再注册Ordered接口的BeanPostProcessor；  4）再注册没有实现任何优先级接口的BeanPostProcessor；  5）最终注册MergedBeanDefinitionPostProcessor；  6）注册一个ApplicationListenerDetector：来在Bean创建完成后检查是否是ApplicationListener，  如果是，添加到容器：  applicationContext.addApplicationListener((ApplicationListener<?>) bean); 7、initMessageSource();初始化MessageSource组件（国际化、消息绑定、消息解析）；  1）获取BeanFactory  2）看容器中是否有id为messageSource，类型是MessageSource的组件；  如果有赋值给messageSource、如果没有自己创建一个DelegatingMessageSource；  MessageSource：取出国际化配置文件中的某个key的值，能够按照区域信息获取；  3）把创建好的MessageSource注册到容器中，以后去国际化配置文件值的时候，可以自动注入到MessageSource；  beanFactory.registerSingleton(MESSAGE\_SOURCE\_BEAN\_NAME, this.messageSource);  String getMessage(String code, Object[] args, String defaultMessage, Locale locale); 8、initApplicationEventMulticaster();初始化事件派发器  1）获取BeanFactory  2）从BeanFactory获取applicationEventMulticaster的 ApplicationEventMulticaster  this.applicationEventMulticaster =  beanFactory.getBean(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME, ApplicationEventMulticaster.class);  3）如果上一步没有配置：默认创建一个SimpleApplicationEventMulticaster  4）将创建的ApplicationEventMulticaster添加到BeanFactory，以后其他组件直接自动注入 9、onRefresh();留给子容器（子类）  1）子类重写这个方法，在容器刷新的时候可以钉钉一逻辑； 10、registerListeners();给容器中将所有项目里面的ApplicationListener注册进来  1）从容器中拿到所有ApplicationListener；  2）将每个监听器添加到事件派发器中：  getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);  3）派发之前步骤产生的事件  getApplicationEventMulticaster().multicastEvent(earlyEvent); 11、finishBeanFactoryInitialization(beanFactory);初始化所有剩下的单实例bean；  1）beanFactory.preInstantiateSingletons();初始化后剩下的单实例bean  2）获取Bean的定义信息：RootBeanDefinition  3）Bean不是抽象的，是单实例的，不是懒加载的  if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {}  1）判断是否是FactoryBean：是否是实现FactoryBean接口的Bean  2）不是工厂Bean：利用getBean创建对象  0、getBean(beanName);  1、doGetBean(name, null, null, false);  2、先获取缓存中保存的单例Bean。如果能获取到说明这个Bean之前被创建过（所有创建过的单实例Bean都会被缓存起来）  从private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256);获取  3、缓存中获取不到，开始Bean的创建对象流程；  4、标记当前bean已经被创建  5、获取bean的定义信息  6、【获取当前Bean依赖的其他Bean，如果有按照getBean()把依赖的Bean先创建出来】  7、启动单实例Bean的单实例创建；  1）createBean(beanName, mbd, args);  2）Object bean = resolveBeforeInstantiation(beanName, mbdToUse);  InstantiationAwareBeanPostProcessor:提前执行  先触发：postProcessBeforeInstantiation();  如果有返回值：触发postProcessAfterInitialization  3）如果前边的  4）Object beanInstance = doCreateBean(beanName, mbdToUse, args);  1）【创建Bean实例】：doCreateBean(beanName, mbdToUse, args);  利用工厂方法或者对象的构造器创建对象  2）applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  调用MergedBeanDefinitionPostProcessor的bdp.postProcessMergedBeanDefinition(mbd, beanType, beanName);  3）【Bean的属性赋值】populateBean(beanName, mbd, instanceWrapper);  赋值之前：  1）拿到InstantiationAwareBeanPostProcessor 后置处理器；  postProcessAfterInstantiation  2）拿到InstantiationAwareBeanPostProcessor 后置处理器；  postProcessPropertyValues  3）应用Bean属性的值；为属性利用setter方法等进行赋值；  applyPropertyValues(beanName, mbd, bw, pvs);  4）【Bean】初始化：initializeBean(beanName, exposedObject, mbd);  1）invokeAwareMethods(beanName, bean); 执行xxxAware接口的方法：public class Red implements ApplicationContextAware,  BeanNameAware/BeanClassLoaderAware/BeanFactoryAware  2）【执行后置处理器之前的方法】：applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  processor.postProcessBeforeInitialization(result, beanName);  3）invokeInitMethods(beanName, wrappedBean, mbd);执行  1）【执行初始化方法】是否是InitializingBean接口的Bean，执行自定义初始化方法  public class Cat implements InitializingBean,  2）是否自定义初始化方法  4）【执行后置处理器之后的方法】applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  1）processor.postProcessAfterInitialization(result, beanName);  5）【注册Bean的销毁方法】registerDisposableBeanIfNecessary(beanName, bean, mbd);  在容器关闭时调用  5）将创建的Bean添加到缓存中  this.singletonObjects.put(beanName, (singletonObject != null ? singletonObject : NULL\_OBJECT));  ioc容器就是这些Map；很多Map里边保存了单实例Bean，环境信息  所有Bean都利用getBean创建完成以后：  检查所有Bean是否是SmartInitializingSingleton接口  12、finishRefresh();完成BeanFactory的初始化创建工作；IOC容器创建完成；  1）initLifecycleProcessor();初始化和生命周期有关的后置处理器：lifecycleProcessor   写一个LifeCycleProcessor的实现类，可以在BeanFactory  void onRefresh();  void onClose();  2）getLifecycleProcessor().onRefresh();  拿到前边定义的生命周期处理器（BeanFactory），回调onRefresh()  3）publishEvent(new ContextRefreshedEvent(this));发布容器刷新完成事件；  4）LiveBeansView.registerApplicationContext(this); |

### 9.2总结

|  |
| --- |
| ========总结============= 1）Spring容器在启动的时候，会先保存所有注册进来的Bean的定义信息；  1）XML注册Bean：<bean></bean>  2）注解注册Bean：@Service、@Component、@Bean 2）Spring容器会在合适的时机创建这些Bean  1）用到这个Bean的时候：利用getBean创建这个Bean；创建好以后保存在容器中；  2）统一创建所有Bean的时候；finishBeanFactoryInitialization() 3）后置处理器：  1）每一个bean创建完成，都会使用各种后置处理器进行处理，增强bean的功能：  AutowiredAnnotationBeanPostProcessor：处理自动注入  AnnotationAwareAspectJAutoProxyCreator：来做AOP  xxx...  增强的功能注解  AsyncAnnotationBeanPostProcessor 4）事件驱动模型：  ApplicationListener：事件监听  ApplicationEventPublisher：事件派发 |