目录

[一、容器 2](#_Toc32174064)

[1、AnnotationConfigApplicationContext 2](#_Toc32174065)

[2、组件注册 2](#_Toc32174066)

[1、自定义过滤规则 5](#_Toc32174067)

[2、@Scope设置组件的作用域 @Lazy 懒加载 8](#_Toc32174068)

[3、Conditional按照条件注册bean 9](#_Toc32174069)

[4、@Import 12](#_Toc32174070)

[5、生命周期 16](#_Toc32174071)

[1、指定初始化、销毁方法； 18](#_Toc32174072)

[2、让bean实现InitializingBean 19](#_Toc32174073)

[3、使用JSR250 19](#_Toc32174074)

[4、BeanPostProcessor：bean的后置处理器 20](#_Toc32174075)

[5. BeanPostProcessor原理 21](#_Toc32174076)

[3、组件赋值 22](#_Toc32174077)

[1、@Value 22](#_Toc32174078)

[2、PropertySource 23](#_Toc32174079)

[3、自动装配 25](#_Toc32174080)

[1、@Autowrite 27](#_Toc32174081)

[2、@Resource @Inject 29](#_Toc32174082)

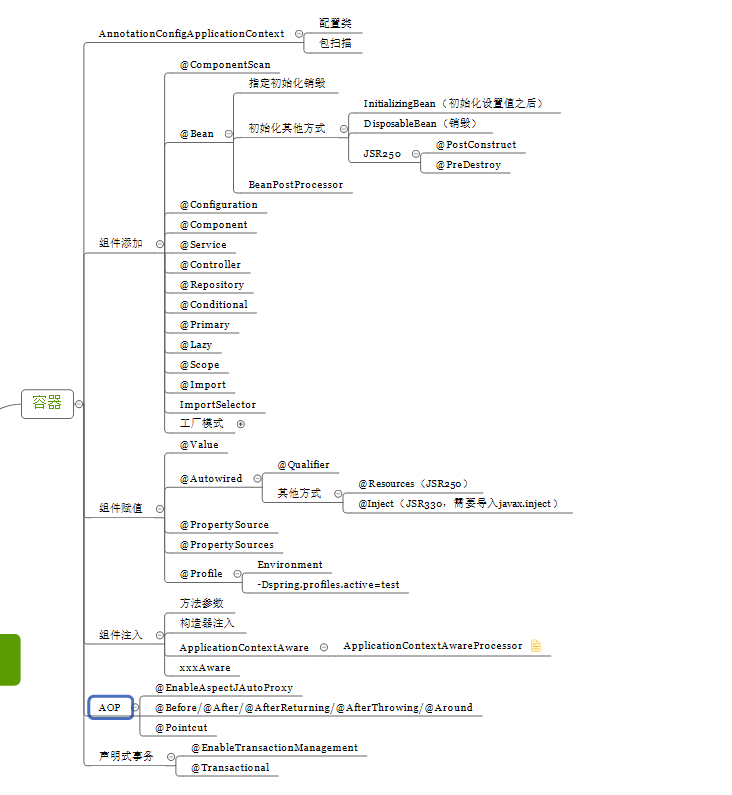
[3、自定义组件想使用Spring容器底层的一些组件 29](#_Toc32174083)

[4、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 31](#_Toc32174084)

[4、AOP 34](#_Toc32174085)

[5、、声明式事务 34](#_Toc32174086)

# 一、容器



## 1、AnnotationConfigApplicationContext

## 2、组件注册

之前使用使用配置文件，向ioc容器中注册bean。并使用包扫描的形式来扫描

要注册的bean

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  
 <!--使用context:component-scan来指定我们要扫描那些包下的组件-->  
 <!--只要标注了@Controller、@Service、@Repository、@Compontent的类都会被扫描到-->  
 <!--use-default-filters="false"禁用默认的过滤规则-->  
 <context:component-scan base-package="com.cc" use-default-filters="false" ></context:component-scan>  
 <!--我们之前使用bean标签的方式注册组件-->  
 <bean id="person" class="com.cc.bean.Person">  
 <property name="age" value="18"></property>  
 <property name="name" value="zhangsan"></property>  
 </bean>  
</beans>

使用配置文件的形式。

package com.cc.config;  
  
import com.cc.bean.Person;  
import com.cc.service.BookService;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.FilterType;  
import org.springframework.stereotype.Controller;  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \* 配置类等同于之前的配置文件  
 \** ***@ClassName*** *Mainconfig  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/8 22:26  
 \** ***@Version*** *1.0  
 \*\*/  
  
/\*\*  
 \* 此注解标志这是一个配置类  
 \*/*@Configuration  
*/\*\*  
 \** ***@ComponentScan(value*** *= "com.cc")//  
 \* 作用等同于配置文件中的context:component-scan并且在jdk1.8中支持多个@ComponentScan（@Repeatable(ComponentScans.class)）  
 \* 例如：@ComponentScan(value = "com.cc.dao") @ComponentScan(value = "com.cc.bean")这两个和在一起就是扫描dao包下和bean包下*

*\* 不是jdk1.8也可以使用ComponentScans指定多个规则  
\* ComponentScans(value = {})*

*\*/*@ComponentScan(value = "com.cc.bean",includeFilters = {/\*@ComponentScan.Filter(type= FilterType.ANNOTATION,classes = {Controller.class, Service.class}),  
 @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE,classes = BookService.class),\*/  
 @ComponentScan.Filter(type=FilterType.*CUSTOM*,classes = MyTypeFilter.class)})  
*/\*\*  
\* ComponentScan中有 ComponentScan.Filter[] includeFilters() default {};扫描的时候只包含那些组件  
\* ComponentScan.Filter[] excludeFilters() default {};扫描的时候按照什么规则排除哪些组件*

*\* 这两个方法可以指定要扫描或不扫描哪些。*

*但是includeFilters要配合useDefaultFilters;false-->相当于配置文件中use-default-filters="false"  
 \* 使用@Filter指定扫描规则，规则而的方式有;  
 \* FilterType.ANNOTATION：按照注解过滤  
 \* FilterType.ASSIGNABLE\_TYPE：按照给定的类型  
 \* FilterType.ASPECTJ：使用ASPECTJ表达式  
 \* FilterType.REGEX：正则  
 \* FilterType.CUSTOM：自定义规则，创建类实现TypeFilter接口 @ComponentScan.Filter(type=FilterType.CUSTOM,classes = MyTypeFilter.class)}  
 \* basePackages与value: 用于指定包的路径，进行扫描  
 \* basePackageClasses: 用于指定某个类的包的路径进行扫描  
 \*/  
  
/\*\*  
 \* ComponentScans其实和ComponentScan一样，相当于一个ComponentScan数组  
 \*/*public class MainConfig {  
  
 @Bean(value = "person01")//给容器中注册bean 类型就是返回值类型 id默认用方法名作为id（为getPerson），如果指定value，那么id就是value（person01）  
 public Person getPerson(){  
 return new Person("lisi",20,"lisi");  
 }  
}

### 1、自定义过滤规则

package com.cc.config;  
  
import org.springframework.core.io.Resource;  
import org.springframework.core.type.AnnotationMetadata;  
import org.springframework.core.type.ClassMetadata;  
import org.springframework.core.type.classreading.MetadataReader;  
import org.springframework.core.type.classreading.MetadataReaderFactory;  
import org.springframework.core.type.filter.TypeFilter;  
  
import java.io.IOException;  
  
*/\*\*  
 \** ***@ClassName*** *MyTypeFilter  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/9 21:14  
 \** ***@Version*** *1.0  
 \*\*/*public class MyTypeFilter implements TypeFilter {  
 */\*\*  
 \*  
 \** ***@param*** *metadataReader 读取到的当前正在扫描的信息  
 \** ***@param*** *metadataReaderFactory 获取到其他任何类的信息  
 \** ***@return*** *\** ***@throws*** *IOException  
 \*/* @Override  
 public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {  
 //获取当前类注解的信息  
 AnnotationMetadata annotationMetadata=metadataReader.getAnnotationMetadata();  
 //获取当前正在扫描的类的类信息  
 ClassMetadata classMetadata=metadataReader.getClassMetadata();  
 //获取当前类资源（类的路径）  
 Resource resource = metadataReader.getResource();  
 String className = classMetadata.getClassName();  
 System.*out*.println("----------->"+className);  
 if(className.contains("er")){  
 return true;  
 }  
 return false;  
 }  
}

使用测试类来进行测试

package com.cc.test;  
  
import com.cc.bean.Blue;  
import com.cc.bean.Person;  
import com.cc.config.MainConfig;  
import com.cc.config.MainConfig2;  
import org.junit.jupiter.api.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
import org.springframework.core.env.ConfigurableEnvironment;  
  
import java.util.Map;  
  
*/\*\*  
 \** ***@ClassName*** *Test  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/8 22:38  
 \** ***@Version*** *1.0  
 \*\*/*public class IocTest {  
 AnnotationConfigApplicationContext annotationConfigApplicationContext = new AnnotationConfigApplicationContext(MainConfig2.class);  
 @SuppressWarnings("resource")  
 @Test  
 public void testImport(){  
 printBeans(annotationConfigApplicationContext);  
 Blue bean = annotationConfigApplicationContext.getBean(Blue.class);  
 System.*out*.println(bean);  
 //工厂bean获取的是调用getObject创建的对象  
 Object colorFactory = annotationConfigApplicationContext.getBean("colorFactory");  
 Object colorFactory2 = annotationConfigApplicationContext.getBean("colorFactory");  
 System.*out*.println("bean2"+colorFactory.getClass());  
 System.*out*.println(colorFactory==colorFactory2);  
  
 */\*\*  
 \* 使用&获取FactoryBean本身  
 \*/* Object colorFactory3 = annotationConfigApplicationContext.getBean("&colorFactory");  
 System.*out*.println("3"+colorFactory3);  
 }  
 private void printBeans(AnnotationConfigApplicationContext annotationConfigApplicationContext){  
 String [] names=annotationConfigApplicationContext.getBeanDefinitionNames();  
 for (String name:names  
 ) {  
 System.*out*.println(name);  
 }  
 }  
 @Test  
 public void test1(){  
  
 String[] beanDefinitionNames = annotationConfigApplicationContext.getBeanDefinitionNames();  
 for (String name:beanDefinitionNames) {  
 System.*out*.println(name);  
 }  
 }  
  
 @Test  
 public void test2(){  
  
  
 /\* String[] beanDefinitionNames = annotationConfigApplicationContext.getBeanDefinitionNames();  
 for (String name:beanDefinitionNames) {  
 System.out.println(name);  
 }  
  
 Object bean=annotationConfigApplicationContext.getBean("person");  
 Object bean2=annotationConfigApplicationContext.getBean("person");  
 System.out.println(bean==bean2);\*/  
 System.*out*.println("ioc容器创建完成");  
 }  
  
 @Test  
 public void test3(){  
  
 ConfigurableEnvironment configurableEnvironment=annotationConfigApplicationContext.getEnvironment();  
 String property = configurableEnvironment.getProperty("os.name");  
 System.*out*.println(property);  
 String[] beanNamesForType = annotationConfigApplicationContext.getBeanNamesForType(Person.class);  
  
 for (String name:beanNamesForType  
 ) {  
 System.*out*.println(name);  
 }  
 Map<String, Person> beansOfType = annotationConfigApplicationContext.getBeansOfType(Person.class);  
 System.*out*.println(beansOfType);  
 }  
}

### 2、@Scope设置组件的作用域 @Lazy 懒加载

@Configuration  
*/\*\*  
 \* 使用@Import快速导入组件，id默认是组件的全类名  
 \*  
 \*/*@Import({Color.class, Red.class,MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})  
public class MainConfig2 {  
 */\*\*  
 \* bean在ioc容器中默认是单实例的可以通过@Scope来调整作用域  
 \** ***@return*** *\*/* @Scope("singleton")  
 */\*\*  
 \* ConfigurableBeanFactory#SCOPE\_PROTOTYPE prototype  
 \* ConfigurableBeanFactory#SCOPE\_SINGLETON singleton  
 \* org.springframework.web.context.WebApplicationContext#SCOPE\_REQUEST request  
 \* org.springframework.web.context.WebApplicationContext#SCOPE\_SESSION session  
 \*  
 \* prototype 多实例的 ioc容器启动不会调用方法创建对象放在容器中，而是每次获取时才会调用方法创建对象  
 \* singleton 单实例的（默认）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中拿  
 \* request 同一次请求创建一个实例  
 \* 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对象  
 \* 懒加载：容器启动不创建对象。第一次使用（获取）bean创建对象，并初始化  
 \*/* @Lazy  
 @Bean("person")  
 public Person person(){  
 System.*out*.println("向容器中添加person");  
 return new Person("zhangsan",70,"lisi");  
 }

}

### 3、Conditional按照条件注册bean

@Configuration  
*/\*\*  
 \* 使用@Import快速导入组件，id默认是组件的全类名  
 \*  
 \*/*@Import({Color.class, Red.class,MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})  
public class MainConfig2 {

*/\*\*  
 \** ***@Conditional({Condition})*** *按照一定的条件进行判断，满足条件给容器注册bean(可以用在类上，也可以用在方法上)  
 \* 如果是windows注册cc  
 \* 如果是linux注册ht*

*\*可以放在属性和方法上*

*\*/*@Conditional({WindowsCondition.class})  
@Bean("cc")  
public Person person01(){  
 return new Person("cc",22,"lis");  
}  
@Conditional({LinuxCondition.class})  
@Bean("ht")  
public Person person02(){  
 return new Person("ht",22,"lisi ");  
}

}

package com.cc.condition;  
  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.core.env.Environment;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
*/\*\*  
 \** ***@ClassName*** *LinuxCondition  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/9 23:36  
 \** ***@Version*** *1.0  
 \*\*/*public class LinuxCondition implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 */\*\*  
 \* 获取当前环境信息  
 \*/* Environment environment = context.getEnvironment();  
 String property = environment.getProperty("os.name");  
 if(property.contains("linux")){  
 return true;  
 }  
 return false;  
 }  
}

package com.cc.condition;  
  
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;  
import org.springframework.beans.factory.support.BeanDefinitionRegistry;  
import org.springframework.context.annotation.Condition;  
import org.springframework.context.annotation.ConditionContext;  
import org.springframework.core.env.Environment;  
import org.springframework.core.type.AnnotatedTypeMetadata;  
  
*/\*\*  
 \** ***@ClassName*** *WindowsCondition  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/9 23:36  
 \** ***@Version*** *1.0  
 \* 判断是否为windows  
 \*\*/*public class WindowsCondition implements Condition {  
 */\*\*  
 \*  
 \** ***@param*** *context p判断条件能使用的上下文（环境）  
 \** ***@param*** *metadata 注释信息  
 \** ***@return*** *\*/* @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 //是否windows系统  
 */\*\*  
 \* 获取ioc使用的beanfactory  
 \*/* ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();  
 */\*\*  
 \* 获取类加载器  
 \*/* ClassLoader classLoader = context.getClassLoader();  
 */\*\*  
 \* 获取当前环境信息  
 \*/* Environment environment = context.getEnvironment();  
 */\*\*  
 \* 获取bean定义 的注册类  
 \*/* BeanDefinitionRegistry registry = context.getRegistry();  
  
 String property = environment.getProperty("os.name");  
 if(property.contains("Windows")){  
 return true;  
 }  
 return false;  
 }  
}

测试

@Test  
public void test3(){  
  
 ConfigurableEnvironment configurableEnvironment=annotationConfigApplicationContext.getEnvironment();  
 String property = configurableEnvironment.getProperty("os.name");  
 System.*out*.println(property);  
 String[] beanNamesForType = annotationConfigApplicationContext.getBeanNamesForType(Person.class);  
  
 for (String name:beanNamesForType  
 ) {  
 System.*out*.println(name);  
 }  
 Map<String, Person> beansOfType = annotationConfigApplicationContext.getBeansOfType(Person.class);  
 System.*out*.println(beansOfType);  
}

### 4、@Import

*/\*\*  
 \* 给容器中注册组件：  
 \* 1、包扫描+组件标注注解（@Controller、@Service、@Repository、@Component）【自己写的类】  
 \* 2、@Bean【导入的第三方包里面的组件】  
 \* 3、@Import【快速给容器中倒入一个组件】  
 \* 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，id默认为全类名  
 \* 2、ImportSelector：返回需要导入的组件的全类名数组 需要手写类实现ImportSelector接口  
 \* 3、ImportBeanDefinitionRegistrar 手动注册bean 需要手写类实现ImportBeanDefinitionRegistrar接口  
 \* 4、使用spring提供的FactoryBean（工厂bean）  
 \* 1、默认获取到的是工厂bean调用getObject创建的对象  
 \* 2、要获取gongchangbean本身需要在id前加&  
 \*/*

*ImportSelector*

package com.cc.config;  
  
import com.cc.bean.Blue;  
import org.springframework.context.annotation.ImportSelector;  
import org.springframework.core.type.AnnotationMetadata;  
  
*/\*\*  
 \** ***@ClassName*** *MyImportSelector  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/10 21:44  
 \** ***@Version*** *1.0  
 \* 自定义逻辑返回需要导入的组件  
 \*\*/*public class MyImportSelector implements ImportSelector {  
 */\*\*  
 \* 返回值就是导入到容器中的组件全类名  
 \** ***@param*** *importingClassMetadata 当前标注@Import注解的类的所有注解信息  
 \** ***@return*** *\*/* @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 //方法不能返回null值  
 return new String[] {"com.cc.bean.Blue","com.cc.bean.Yellow"};  
 }  
}

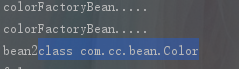
*ImportBeanDefinitionRegistrar*

package com.cc.config;  
  
import com.cc.bean.Pink;  
import org.springframework.beans.factory.support.BeanDefinitionRegistry;  
import org.springframework.beans.factory.support.RootBeanDefinition;  
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;  
import org.springframework.core.type.AnnotationMetadata;  
  
*/\*\*  
 \** ***@ClassName*** *MyImportBeanDefinitionRegistrar  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/10 22:32  
 \** ***@Version*** *1.0  
 \*\*/*public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {  
 */\*\*  
 \*  
 \** ***@param*** *importingClassMetadata 当前类的注解信息  
 \** ***@param*** *registry BeanDefinition注册类  
 \* 把所有需要添加到容器中的bean；调用  
 \* registry.registerBeanDefinition()手动注册  
 \*/* @Override  
 public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  
 boolean red = registry.containsBeanDefinition("com.cc.bean.Red");  
 boolean blue = registry.containsBeanDefinition("com.cc.bean.Blue");  
 if(red&&blue){  
 //指定Bean定义信息（Bean的类型等）  
 RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(Pink.class);  
 //注册一个bean，指定bean名  
 registry.registerBeanDefinition("pink",rootBeanDefinition);  
 }  
  
 }  
}

*使用spring提供的FactoryBean（工厂bean）*

package com.cc.bean;  
  
import org.springframework.beans.factory.FactoryBean;  
  
*/\*\*  
 \* 创建Spring定义的FactoryBean  
 \** ***@ClassName*** *ColorFactory  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/10 22:46  
 \** ***@Version*** *1.0  
 \*\*/*public class ColorFactory implements FactoryBean<Color> {  
 */\*\*  
 \* 返回一个Color对象，这个对象会添加到容器中  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Override  
 public Color getObject() throws Exception {  
 System.*out*.println("colorFactoryBean.....");  
 return new Color();  
 }  
  
 @Override  
 public Class<?> getObjectType() {  
 return Color.class;  
 }  
  
 */\*\*  
 \* 控制是否单例 true：单例  
 \** ***@return*** *\*/* @Override  
 public boolean isSingleton() {  
 return false;  
 }  
}

@Test  
public void testImport(){  
 printBeans(annotationConfigApplicationContext);  
 Blue bean = annotationConfigApplicationContext.getBean(Blue.class);  
 System.*out*.println(bean);  
 //工厂bean获取的是调用getObject创建的对象（其实就是color）  
 Object colorFactory = annotationConfigApplicationContext.getBean("colorFactory");  
 Object colorFactory2 = annotationConfigApplicationContext.getBean("colorFactory");  
 System.*out*.println("bean2"+colorFactory.getClass());  
 System.*out*.println(colorFactory==colorFactory2);  
  
 */\*\*  
 \* 使用&获取FactoryBean本身  
 \*/* Object colorFactory3 = annotationConfigApplicationContext.getBean("&colorFactory");  
 System.*out*.println("3"+colorFactory3);  
}



### 5、生命周期

package com.cc.config;  
  
*/\*\*  
 \** ***@ClassName*** *MainConfigOfLifeCycle  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/11 20:26  
 \** ***@Version*** *1.0  
 \*\*/*import com.cc.bean.Car;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Scope;  
  
*/\*\*  
 \* bean的生命周期：  
 \* bean创建---初始化---销毁的过程  
 \* 容器管理bean的生命周期  
 \* 我们可以自定义初始化和销毁方法；容器在bean进行到当前生命周期的时候来调用我们自定义的初始化和销毁方法。  
 \* 构造（对象创建）  
 \* 单实例：在容器启动的时候创建对象  
 \* 多实例：在每次获取的时候创建对象  
 \* 初始化：  
 \* 在对象创建完成，并赋值好，调用初始化方法。  
 \* 销毁：  
 \* 单实例：在容器关闭的时候  
 \* 多实例：容器不会管理这个bean，容器不会调用销毁方法。  
 \*  
 \* 遍历得到容器中所有的BeanPostProcessor；挨个执行BeforeInitialization，一旦返回null，跳出for循环，不会执行后面的BeanPostProcessor  
  
 \*  
 \*  
 \* 1、指定初始化、销毁方法；  
 \* 1、以前基于xml配置，我们有init-method 和 destroy-method方法这些方法必须没有参数但可以抛出任何异常；现在我们可以通过@Bean(initMethod = "init",destroyMethod = "detory")  
 \* 2、让bean实现InitializingBean（定义初始化逻辑）  
 \* 实现DisposableBean（定义销毁逻辑）这两个接口是Spring提供  
 \* 3、使用JSR250  
 \** ***@PostConstruct:在bean创建完成并且属性赋值完成，来执行初始化方法*** *\** ***@PreDestory:在容器销毁bean之前，通知我们进行销毁方法*** *\* 4、BeanPostProcessor：bean的后置处理器  
 \* 在bean初始化前后进行一些处理工作  
 \* postProcessBeforeInitialization：在初始化之前工作  
 \* postProcessAfterInitialization：在初始化之后工作  
 \*  
 \* BeanPostProcessor原理  
 \* populateBean(beanName, mbd, instanceWrapper)给bean进行属性赋值  
 \* initializeBean(beanName, exposedObject, mbd);初始化bean  
 \* {  
 \* applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  
 \* invokeInitMethods(beanName, wrappedBean, mbd);执行自定义初始化方法  
 \* applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  
 \* }  
 \* 首先会对bean进行属性赋值---》初始化bean---》applyBeanPostProcessorsBeforeInitialization---》invokeInitMethods---》applyBeanPostProcessorsAfterInitialization  
 \*  
 \* Spring底层对BeanPostProcessor使用  
 \* bean赋值，注入其他组件@Autowirte，生命周期注解功能，@Async，xxxBEanPostProcessor  
 \*  
 \*  
 \*/*@ComponentScan("com.cc.bean")  
@Configuration  
public class MainConfigOfLifeCycle {  
 @Scope("prototype")  
 @Bean(initMethod = "init",destroyMethod = "detory")  
 public Car car(){  
 return new Car();  
 }  
  
  
}

#### 1、指定初始化、销毁方法；

package com.cc.bean;  
  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \** ***@ClassName*** *Car  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/11 20:31  
 \** ***@Version*** *1.0  
 \*\*/*@Component  
public class Car {  
 public Car(){  
 System.*out*.println("car 创建");  
 }  
 public void init(){  
 //进行一些初始化赋值  
 System.*out*.println("car init");  
 }  
 public void detory(){  
 System.*out*.println("car detory");  
 }  
}

#### 2、让bean实现InitializingBean

定义初始化逻辑、实现DisposableBean（定义销毁逻辑）这两个接口是Spring提供

package com.cc.bean;  
  
import org.springframework.beans.factory.DisposableBean;  
import org.springframework.beans.factory.InitializingBean;  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \** ***@ClassName*** *Cat  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/11 20:43  
 \** ***@Version*** *1.0  
 \*\*/*@Component  
public class Cat implements InitializingBean, DisposableBean {  
 public Cat(){  
 System.*out*.println("cat 创建");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.*out*.println("cat destroy");  
 }  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 System.*out*.println("cat init");  
 }  
}

#### 3、使用JSR250

@PostConstruct:在bean创建完成并且属性赋值完成，来执行初始化方法@PreDestory:在容器销毁bean之前，通知我们进行销毁方法

package com.cc.bean;  
  
import org.springframework.stereotype.Component;  
  
import javax.annotation.PostConstruct;  
import javax.annotation.PreDestroy;  
  
*/\*\*  
 \** ***@ClassName*** *Dog  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/11 21:24  
 \** ***@Version*** *1.0  
 \*\*/*@Component  
public class Dog {  
 public Dog(){  
 System.*out*.println("dog construct");  
 }  
 //对象创建并赋值之后创建  
 @PostConstruct  
 public void init(){  
 System.*out*.println("dog。。。。@PostConstruct");  
 }  
 @PreDestroy  
 public void detory(){  
 System.*out*.println("dog.....@PreDestroy");  
 }  
}

#### 4、BeanPostProcessor：bean的后置处理器

在bean初始化前后进行一些处理

package com.cc.bean;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \** ***@ClassName*** *MYBeanPostProcessor  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/11 21:29  
 \** ***@Version*** *1.0  
 \* 初始化前后进行处理  
 \*\*/*@Component  
public class MyBeanPostProcessor implements BeanPostProcessor {  
 */\*\*  
 \*  
 \** ***@param*** *bean 容器创建的bean实例  
 \** ***@param*** *beanName bean的名称  
 \** ***@return*** *可以直接返回bean，也可以进行包装后返回  
 \** ***@throws*** *BeansException  
 \*/* @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 System.*out*.println("postProcessBeforeInitialization..."+beanName+"=>"+bean);  
 return bean;  
 }  
  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 System.*out*.println("postProcessAfterInitialization..."+beanName+"=>"+bean);  
 return bean;  
 }  
}

其实就是：

先执行BeanPostProcessor.postProcessBeforeInitialization

然后初始化： 在执行BeanPostProcessor.postProcessAfterInitialization

#### 5. BeanPostProcessor原理

遍历得到容器中所有的BeanPostProcessor；挨个执行BeforeInitialization，一旦返回null，跳出for循环，不会执行后面的BeanPostProcessor

\* populateBean(beanName, mbd, instanceWrapper)给bean进行属性赋值\* initializeBean(beanName, exposedObject, mbd);初始化bean\* {\* applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);\* invokeInitMethods(beanName, wrappedBean, mbd);执行自定义初始化方法\* applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);\* }\* 首先会对bean进行属性赋值---》初始化bean---》applyBeanPostProcessorsBeforeInitialization---》invokeInitMethods---》applyBeanPostProcessorsAfterInitialization\*\* Spring底层对BeanPostProcessor使用bean赋值，注入其他组件@Autowirte，生命周期注解功能，@Async，xxxBEanPostProcessor（异步方法） 都是使用 BeanPostProcessor完成的\*

## 3、组件赋值

package com.cc.config;  
  
import com.cc.bean.Person;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.PropertySource;  
  
*/\*\*  
 \** ***@ClassName*** *MainConfigOfPropertyValues  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/18 22:33  
 \** ***@Version*** *1.0  
 \*\*/  
  
/\*\*  
 \*使用PropertySource读取外部配置文件中的k/v保存到运行的环境变量中;加载完外部的配置文件以后使用${}取出配置文件的值  
 \** ***@PropertySource可以写多个*** *\* 当然作用等同于@PropertySources  
 \*/*@PropertySource(value={"classpath:person.properties"})  
@Configuration  
public class MainConfigOfPropertyValues {  
 @Bean  
 public Person person(){  
 return new Person();  
 }  
}

### 1、@Value

package com.cc.bean;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import org.springframework.beans.factory.annotation.Value;  
  
*/\*\*  
 \** ***@ClassName*** *Person  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/8 22:11  
 \** ***@Version*** *1.0  
 \*\*/*@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Person {  
 */\*\*  
 \* 使用@Value  
 \* 1、几本书只  
 \* 2、使用SpEL #{}  
 \* 3、${}取配置文件中的值（在运行环境变量中的值）  
 \*/* @Value("chenchen")  
 private String name;  
 @Value("#{20-2}")  
 private Integer age;  
 @Value("${person.EnglistName}")  
 private String EnglishName;  
}

### 2、PropertySource

person.EnglistName=lisi

/\*\* \*使用PropertySource读取外部配置文件中的k/v保存到运行的环境变量中;加载完外部的配置文件以后使用${}取出配置文件的值 \* @PropertySource可以写多个 \* 当然作用等同于@PropertySources \*/@PropertySource(value={"classpath:person.properties"})‘

测试

package com.cc.test;  
  
import com.cc.bean.Person;  
import com.cc.config.MainConfigOfLifeCycle;  
import com.cc.config.MainConfigOfPropertyValues;  
import org.junit.jupiter.api.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
import org.springframework.core.env.ConfigurableEnvironment;  
  
*/\*\*  
 \** ***@ClassName*** *IOCTest\_PropertyValue  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/18 22:34  
 \** ***@Version*** *1.0  
 \*\*/*public class IOCTest\_PropertyValue {  
 //创建ioc容器  
 AnnotationConfigApplicationContext annotationConfigApplicationContext = new AnnotationConfigApplicationContext(MainConfigOfPropertyValues.class);  
 @Test  
 public void test1(){  
 printBeans(annotationConfigApplicationContext);  
 System.*out*.println("====================");  
 Person p=(Person)annotationConfigApplicationContext.getBean("person");  
 System.*out*.println(p);  
 System.*out*.println("======================");  
 ConfigurableEnvironment environment = annotationConfigApplicationContext.getEnvironment();  
 String property = environment.getProperty("person.EnglistName");  
 System.*out*.println(property);  
 annotationConfigApplicationContext.close();  
 }  
 private void printBeans(AnnotationConfigApplicationContext annotationConfigApplicationContext){  
 String [] names=annotationConfigApplicationContext.getBeanDefinitionNames();  
 for (String name:names  
 ) {  
 System.*out*.println(name);  
 }  
 }  
}

### 3、自动装配

package com.cc.config;  
  
*/\*\*  
 \** ***@ClassName*** *MyConfigOfAutowried  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/18 23:03  
 \** ***@Version*** *1.0  
 \*\*/*import com.cc.dao.BookDao;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
  
import javax.annotation.Resource;  
  
*/\*\*  
 \* 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、@Autowrite:自动注入  
 \* 1、有限按照类型去容器中找到对应的组件annotationConfigApplicationContext.getBean(BookDao.class)找到就赋值  
 \* 2、如果找到多个相同的组件，则将属性的名称作为组件的id去容器中查找 applicatioContext.getBean("bookDao")*

*\* 3、@Qualifier("bookDao")使用@Qualifier指定需要装配的组件的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  
 \* 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bena  
 \**  *也可以继续使用@Qualifier指定需要装配的bean的名字(@Qualifier的优先级要高于@Primary)*

*\* BookService{  
 \** ***@Autowrite*** *\* BookDao bookDao；  
 \*  
 \* }  
 \* 2、Spring还支持试用@Resouurce(JSR250)和Inject（JSR330）【java规范的注解】  
 \** ***@Resource:*** *\* 可以和@Autowirted一样实现自动装配功能；默认是按照组件名称进行装配的  
 \* 没有能支持@Primary功能没有支持@Autowrited（reqiured=false）  
 \** ***@Inject:*** *\* 需要导入javax.inject包 和Autowrited的功能一样。没有required=false的功能  
 \** ***@Autowrite是Spring定义的*** *@Resource和@Inject都是java规范  
 \* AutowiredAnnotationBeanPostProcessor：解析完成自动装配功能  
 \* 3、@Autowrited：构造器、属性、方法、参数 都是从容器中获取组件的值  
 \* 1）、标注在方法上 //标注在方法上，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（ @Bean标注的方法创建对象时，方法参数的值从容器中获取）  
 \* 2）、标注在构造器上 构造器需要的组件，都是从容器中获取（如果组件只有一个有参构造器，这个有参构造器的@Autowrited可以省略，参数位置的组件还是可以从容器中获取）  
 \* 3）、放在参数位置  
 \* 4、自定义组件想使用Spring容器底层的一些组件：ApplicationContext，BeanFactory  
 \* 自定义组件实现xxxAware 在创建对象的时候，会调用接口规定的方法注入相关组件  
 \* 把Spring底层一些组件注入到自定义的Bean中  
 \* xxxAware:使用xxxProcessor  
 \*  
 \*  
 \*/*@Configuration  
@ComponentScan({"com.cc.controller","com.cc.service","com.cc.dao","com.cc.bean"})  
public class MyConfigOfAutowried {  
  
 @Primary  
 @Bean(value = "bookDao")  
 @Resource  
 public BookDao bookDao(){  
 BookDao bookDao2=new BookDao();  
 bookDao2.setLabel("2");  
 return bookDao2;  
 }  
}

package com.cc.service;  
  
import com.cc.dao.BookDao;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
import javax.annotation.Resource;  
import javax.inject.Inject;  
  
*/\*\*  
 \** ***@ClassName*** *BookService  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/8 22:37  
 \** ***@Version*** *1.0  
 \*\*/*@Service  
public class BookService {  
 /\*@Autowired(required = false)//required = false当容器中没有时则不装配，默认是必须装配（true）。

@Qualifier("bookDao2")\*/  
 /\*@Resource\*/  
 @Inject  
 private BookDao bookDao;  
 public void print(){  
 System.*out*.println(bookDao);  
 }  
  
 @Override  
 public String toString() {  
 return "BookService{" +  
 "bookDao=" + bookDao +  
 '}';  
 }  
}

#### 1、@Autowrite

1、默认优先按照类型去容器中找到对应的组件annotationConfigApplicationContext.getBean(BookDao.class)找到就赋值2、如果找到多个相同的组件，在将属性的名称作为组件的id去容器中查找applicatioContext.getBean("bookDao")3、@Qualifier("bookDao")使用@Qualifier指定需要装配的组件的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

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bena也可以继续使用@Qualifier指定需要装配的bean的名字(@Qualifier的优先级要高于@Primary) BookService{

@Autowrite BookDao bookDao； }

标注的位置

@Autowrited：构造器、属性、方法、参数 都是从容器中获取组件的值1）、标注在方法上

@Bean+方法参数;参数从从容器中获取；默认不写@Autowrited效果是一样的；都能自动装配

2）、标注在构造器上

构造器需要的组件，都是从容器中获取

（如果组件只有一个有参构造器，这个有参构造器的@Autowrited可以省略，参数位置的组件还是可以从容器中获取）3）、放在参数位置

package com.cc.bean;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \** ***@ClassName*** *Boss  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/11/5 22:20  
 \** ***@Version*** *1.0  
 \*\*/*//默认加载ioc容器中的组件，容器启动会调用无参构造器创建对象，在进行初始化赋值等操作  
@Component  
public class Boss {  
 private Car car;  
 //构造器需要的组件，都是从容器中获取  
 //@Autowired  
 //放在参数位置  
 public Boss(@Autowired Car car){  
 this.car=car;  
 }  
  
 public Car getCar() {  
 return car;  
 }  
 //@Autowired  
 //标注在方法上，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;  
 }  
  
 @Override  
 public String toString() {  
 return "Boss{" +  
 "car=" + car +  
 '}';  
 }  
}

#### 2、@Resource @Inject

Spring还支持试用@Resouurce(JSR250)和Inject（JSR330）【java规范的注解】

@Resource:可以和@Autowirted一样实现自动装配功能；默认是按照组件名称进行装配的 没有能支持@Primary功能

没有支持@Autowrited（reqiured=false）@Inject:需要导入javax.inject包 和Autowrited的功能一样。

支持@Primary但没有required=false的功能@Autowrite是Spring定义的 @Resource和@Inject都是java规范

#### 3、自定义组件想使用Spring容器底层的一些组件

比如：ApplicationContext，BeanFactory

自定义组件实现xxxAware 在创建对象的时候，会调用接口规定的方法注入相关组件把Spring底层一些组件注入到自定义的Bean中xxxAware的功能使用xxxProcessor进行处理

ApplicationContextAware===》ApplicationContextAwareProcessor

package com.cc.bean;  
  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.BeanNameAware;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ApplicationContextAware;  
import org.springframework.context.EmbeddedValueResolverAware;  
import org.springframework.stereotype.Component;  
import org.springframework.util.StringValueResolver;  
  
*/\*\*  
 \** ***@ClassName*** *Red  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/10 21:24  
 \** ***@Version*** *1.0  
 \*\*/*@Component  
public class Red implements ApplicationContextAware, BeanNameAware, EmbeddedValueResolverAware {  
 private ApplicationContext applicationContext;  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 System.*out*.println("传入的ioc："+applicationContext);  
 this.applicationContext=applicationContext;  
 }  
  
 @Override  
 public void setBeanName(String name) {  
 System.*out*.println("当前bean的名称:"+name);  
 }  
  
 @Override  
 public void setEmbeddedValueResolver(StringValueResolver resolver) {  
  
  
 String s = resolver.resolveStringValue("你好${os.name} 我是#{20\*18}");  
 System.*out*.println("解析站位符:"+s);  
 }  
}

4、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；

package com.cc.config;  
  
import com.cc.bean.Yellow;  
/\*import com.mchange.v2.c3p0.ComboPooledDataSource;\*/  
import com.mchange.v2.c3p0.ComboPooledDataSource;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.EmbeddedValueResolverAware;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Profile;  
import org.springframework.context.annotation.PropertySource;  
import org.springframework.util.StringValueResolver;  
  
import javax.sql.DataSource;  
import java.beans.PropertyVetoException;  
  
*/\*\*  
 \** ***@ClassName*** *MainconfigOfProfile  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/11/7 22:26  
 \** ***@Version*** *1.0  
 \* 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  
 \* 比如数据源（/A：开发环境/B：测试环境/C：生产环境）  
 \** ***@profile:指定组件在那个环境的情况下才能被注册到容器中，不指定，任何环境都能注册这个组件*** *\* 1)、加了环境标识的bean 只有在这个环境才会被注册到容器中 默认是default环境  
 \* 2）、写在配置类上、只有是指定的环境的时候，整个配置类里面的所有配置才会生效。  
 \* 3）、没有标注环境表示的bean在任何环境下都是加载的  
 \*\*/*@PropertySource("classpath:/dbconfig.properties")  
@Configuration  
public class MainconfigOfProfile implements EmbeddedValueResolverAware {  
 @Value("${user}")  
 private String user;  
 private String driverClass;  
 private StringValueResolver valueResolver;  
 @Profile("test")  
 @Bean  
 public Yellow yellow(){  
 return new Yellow();  
 }  
  
  
 @Profile("test")  
 @Bean("testDataSource")  
 public DataSource dataSourceTest(@Value("${db.password}") String pwd) throws PropertyVetoException {  
 ComboPooledDataSource dataSource=new ComboPooledDataSource();  
 dataSource.setUser(user);  
 dataSource.setPassword(pwd);  
 dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/depot");  
 dataSource.setDriverClass(driverClass);  
 return dataSource;  
 }  
 @Profile("dev")  
 @Bean("DevDataSource")  
 public DataSource dataSourceDevelop(@Value("${db.password}") String pwd) throws PropertyVetoException {  
 ComboPooledDataSource dataSource=new ComboPooledDataSource();  
 dataSource.setUser(user);  
 dataSource.setPassword(pwd);  
 dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/prjdb");  
 dataSource.setDriverClass(driverClass);  
 return dataSource;  
 }  
 @Profile("prod")  
 @Bean("ProdDataSource")  
 public DataSource dataSourceProduct(@Value("${db.password}") String pwd) throws PropertyVetoException {  
 ComboPooledDataSource dataSource=new ComboPooledDataSource();  
 dataSource.setUser(user);  
 dataSource.setPassword(pwd);  
 dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/twenty");  
 dataSource.setDriverClass(driverClass);  
 return dataSource;  
 }  
  
 @Override  
 public void setEmbeddedValueResolver(StringValueResolver resolver) {  
 this.valueResolver=resolver;  
 driverClass=valueResolver.resolveStringValue("${db.driverClass}");  
 }  
}

测试

package com.cc.test;  
  
import com.cc.bean.Boss;  
import com.cc.config.MainconfigOfProfile;  
import com.cc.config.MyConfigOfAutowried;  
import com.cc.dao.BookDao;  
import com.cc.service.BookService;  
import org.junit.jupiter.api.Test;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import javax.sql.DataSource;  
  
*/\*\*  
 \** ***@ClassName*** *IOC\_test\_profile  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/11/7 22:51  
 \** ***@Version*** *1.0  
 \*\*/*public class IOC\_test\_profile {  
 //创建ioc容器  
  
 //1、使用命令行动态参数:在虚拟机参数位置加载 -Dspring.profiles.active=test  
 //2、代码的方式激活某种环境  
 @Test  
 public void test1(){  
 AnnotationConfigApplicationContext annotationConfigApplicationContext = new AnnotationConfigApplicationContext();  
 //1、创建ioc容器  
  
 //2、设置需要激活的环境  
 annotationConfigApplicationContext.getEnvironment().setActiveProfiles("test","dev");  
 //3、注册配置类  
 annotationConfigApplicationContext.register(MainconfigOfProfile.class);  
 //4、启动容器刷新  
 annotationConfigApplicationContext.refresh();  
  
 String[] beanNamesForType = annotationConfigApplicationContext.getBeanNamesForType(DataSource.class);  
 for (String x:beanNamesForType  
 ) {  
 System.*out*.println(x);  
 }  
 }  
}

## 4、AOP

## 5、、声明式事务