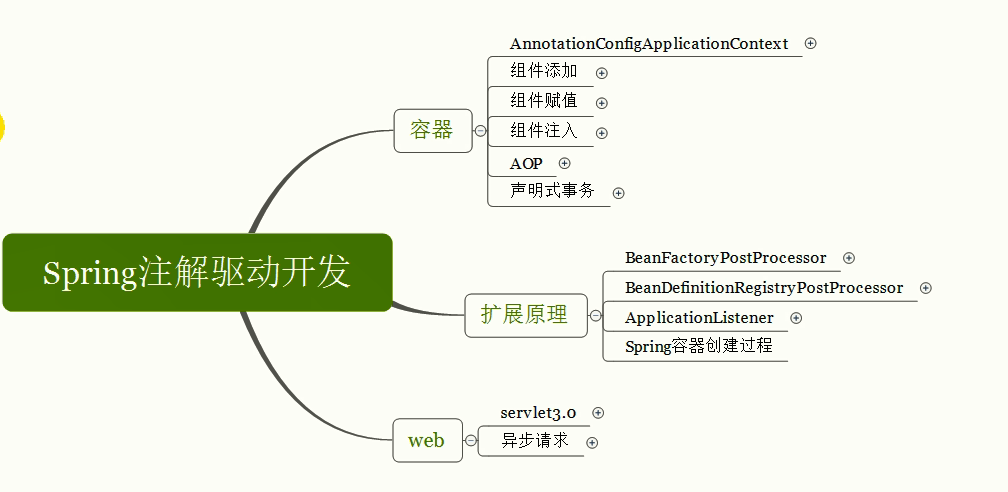
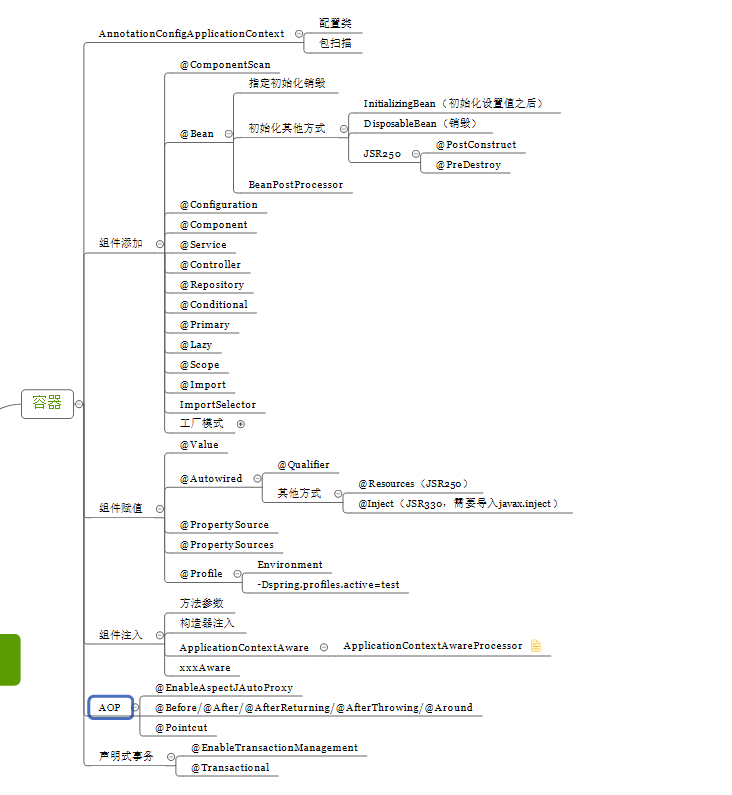
1. 注解驱动开发





1. 使用配置文件

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 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>

1. 使用@Configuration&@Bean给容器中注册组件

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  
 \*/*@Configurationpublic class MainConfig {  
  
 @Bean(value = "person")//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，如果指定value，那么id就是value  
 public Person getPerson(){  
 return new Person("lisi",20);  
 }  
}

@ComponentScan自动扫描组件&扫描规则

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  
 \*/*@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 {};  
 \* ComponentScan.Filter[] excludeFilters() default {};  
 \* 这两个方法可以指定要扫描或不扫描哪些。但是要配合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 = "person")//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，如果指定value，那么id就是value  
 public Person getPerson(){  
 return new Person("lisi",20);  
 }  
}

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;  
 }  
}

@Scope @Lazy

package com.cc.config;  
  
import com.cc.bean.Color;  
import com.cc.bean.Person;  
import com.cc.bean.Red;  
import com.cc.condition.LinuxCondition;  
import com.cc.condition.WindowsCondition;  
import org.springframework.beans.factory.config.ConfigurableBeanFactory;  
import org.springframework.context.annotation.\*;  
  
*/\*\*  
 \** ***@ClassName*** *MainConfig2  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/9 21:26  
 \** ***@Version*** *1.0  
 \*\*/*@Configuration  
*/\*\*  
 \* 使用@Import快速导入组件，id默认是组件的全类名  
 \*  
 \*/*  
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("想容器中添加person");  
 return new Person("zhangsan",70);  
 }

@Conditional按照条件注册bean

package com.cc.config;  
  
import com.cc.bean.Color;  
import com.cc.bean.Person;  
import com.cc.bean.Red;  
import com.cc.condition.LinuxCondition;  
import com.cc.condition.WindowsCondition;  
import org.springframework.beans.factory.config.ConfigurableBeanFactory;  
import org.springframework.context.annotation.\*;  
  
*/\*\*  
 \** ***@ClassName*** *MainConfig2  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2019/9/9 21:26  
 \** ***@Version*** *1.0  
 \*\*/*@Configuration  
*/\*\*  
 \* 使用@Import快速导入组件，id默认是组件的全类名  
 \*  
 \*/*  
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  
 \*/* @Conditional({WindowsCondition.class})  
 @Bean("cc")  
 public Person person01(){  
 return new Person("cc",22);  
 }  
 @Conditional({LinuxCondition.class})  
 @Bean("ht")  
 public Person person02(){  
 return new Person("ht",22);  
 }  
  
}

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;  
 }  
}

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&

@Import({Color.class, Red.class})  
public class MainConfig2 {

*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&*

}

@Import

@Import({Color.class, Red.class,MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})  
public class MainConfig2 {

}

1. 直接使用过类
2. 使用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"};  
 }  
}

1. 使用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

package com.cc.bean;  
  
import org.springframework.beans.factory.FactoryBean;  
import org.springframework.cglib.proxy.Factory;  
  
*/\*\*  
 \* 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;  
 }  
}

测试

@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);  
}