目录

[Spring 注解 2](#_Toc56845483)

[@Configuration 2](#_Toc56845484)

# Spring 注解

## @Configuration

**将实例注册到Spring容器中，相当于在spring-beans.xml中定义**

<beans>

<context:annotation-config/>

<bean class="com.acme.AppConfig"/>

</beans>

**在配置类中使用@Configuration后，会将@Bean注解的Java类实例注册到Spring容器**

@Configuration

public class AppConfig {

@Bean

public MyBean myBean() {

// instantiate, configure and return bean ...

}

}

**通过@Configuration注解注册的实例，可以通过AnnotationConfigApplictionContext获取**

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class);

ctx.refresh();

MyBean myBean = ctx.getBean(MyBean.class);

// use myBean ...

**@Configuration类不仅可以使用组件扫描来引导，还可以自己使用@ComponentScan注释配置组件扫描**

@Configuration

@ComponentScan("com.acme.app.services")

public class AppConfig {

// various @Bean definitions ...

}

**@Confirguration配置类可以用@Import注解导入其他配置类**

@Configuration

public class DatabaseConfig {

@Bean

public DataSource dataSource() {

// instantiate, configure and return DataSource

}

}

@Configuration

@Import(DatabaseConfig.class)

public class AppConfig {

private final DatabaseConfig dataConfig;

public AppConfig(DatabaseConfig dataConfig) {

this.dataConfig = dataConfig;

}

@Bean

public MyBean myBean() {

// reference the dataSource() bean method

return new MyBean(dataConfig.dataSource());

}

}

## @Scope

**当与@Component一起用作类型级别的注释时，@ Scope表示要用于带注释类型的实例的范围。**

**当与@Bean一起用作方法级注释时，@ Scope表示要用于从方法返回的实例的范围。**

/\*\*

\* Scope value:

\* prototype 多实例

\* singleton （默认）单实例，IOC容器启动时会调用方法创建对象放到IOC容器，之后每次都从容器中拿

\* request 同一次请求创建一个实例

\* session 同一个session创建一个实例

\*

\*/

*@Scope*("prototype")

*@Bean*

public User user() {

return new User(8888l, userName);

}

## @Conditional

**仅当所有指定条件都匹配时，组件才有资格注册**

**@Conditional注解在以下方式中使用：**

* **直接或间接作用在@Component注释的类，包括@Configuration**
* **作为元注释，目的是组成自定义构造型注释**
* **作为任何@Bean方法上的方法级注释**

public class OverCondition implements Condition {

*@Override*

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

// TODO Auto-generated method stub

//获取到IOC使用的beanFactory

ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();

//获取到类加载器

ClassLoader classLoader = context.getClassLoader();

//获取到当前环境信息

Environment environment = context.getEnvironment();

//获取到当前bean定义的注册类

BeanDefinitionRegistry registry = context.getRegistry();

return false;

}

}

## @Import

**快速给容器导入组件**

Color是通过类名直接注册到容器

*@Configuration*

*@ComponentScan*(value = "com.zh.spring")

*@Import*({Color.class, ColorImportSelector.class, ColorBeanDefinitionRegistrar.class})

public class MainConfig {

……

}

Blue是通过ImportSelector返回的全路径类名注册到容器

public class ColorImportSelector implements ImportSelector{

*@Override*

public String[] selectImports(AnnotationMetadata importingClassMetadata) {

// **TODO** Auto-generated method stub

return new String[] {"com.zh.spring.bean.Blue"};

}

}

Red是通过BeanDefinitionRegistrar手动注册到容器

public class ColorBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar{

*@Override*

public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

// **TODO** Auto-generated method stub

if (!registry.containsBeanDefinition("com.zh.spring.bean.Red")) {

RootBeanDefinition beanDefinition = new RootBeanDefinition(Red.class);

registry.registerBeanDefinition("red", beanDefinition);

}

}

}

## @PostConstruct

# Spring 组件注册

## 包扫描+组件注解

**@ComponentScan + @Controller/@Service/@@Repository/@Component**

## **@Bean**

**导入第三方包里的组件**

## @Import

**快速给容器导入一个组件**

* *@Import*({Color.class}) 容器会自动注册这个组件；
* *@Import*({ColorImportSelector.class }) ImportSelector会返回需要注册的组件的全类名数组，容器会将这些组件注册；
* *@Import*({ColorBeanDefinitionRegistrar.class}) 在BeanDefinitionRegistrar写代码手动注册组件到容器中

## FactoryBean

# Spring IOC

# Spring Aop

## @EnableAspectJAutoProxy

在配置类上启用@EnableAspectJAutoProxy注解，那么这个注解主要起什么作用呢？

阅读源码可知,@EnbableAspectJAutoProxy注解通过@Import注解导入AspectJAutoProxyRegistrar的组件

package com.zh.spring.test;

}

## AspectJAutoProxyRegistrar

AspectJAutoProxyRegistrar实现了ImportBeanDefinitionRegistrar接口，可以手动注册组件到容器，而AspectJAutoProxyRegistrar通过registerBeanDefinitions方法向容器中注册了AnnotationAwareAspectJAutoProxyCreator的类定义。

package com.zh.spring.test;

}

## AnnotationAwareAspectJAutoProxyCreator

关于AOP切面生成代理类的后处理器，实现了InstantiationBeanPostPorcessor，BeanPostProcessor以及BeanFactory接口。

前者在bean实例化前会调用postProcessorBeforeInitiation方法尝试创建代理类对象；后者在bean初始化完成后会调用postProcessorAfterInitialization方法尝试返回代理对象。

package com.zh.spring.test;

}

## AspectJAwareAdvisorAutoProxyCreator

package com.zh.spring.test;

}

## AbstractAdvisorAutoProxyCreator

package com.zh.spring.test;

}

## AbstractAutoProxyCreator

package com.zh.spring.test;

}

## DefaultAopProxyFactory

package com.zh.spring.test;

}

## JdkDynamicAopProxy

package com.zh.spring.test;

}

## ObjenessisCglibAopProxy

package com.zh.spring.test;

}

## Spring AOP的原理分析

### 容器初始化

package com.zh.spring.test;

}

### AbstractApplication.refresh()

package com.zh.spring.test;

}

### invokeBeanFactoryPostProcessor()

package com.zh.spring.test;

}

### registerBeanPostProcessors()

package com.zh.spring.test;

}

### finishBeanFactoryInitialization()

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

package com.zh.spring.test;

}

# 设计模式

## 单例模式

### 饿汉式单例模式

**饿汉式单例模式是在类加载时就立即初始化，并且创建单例对象。它绝对线程安全，在线程还没有出现前就实例化了，不可能存在访问安全问题。饿汉式单例模式适用于单例对象较少的情况**

package com.zh.spring.test;

public class HungrySingleton {

private static final HungrySingleton ***hungrySingleton*** = new HungrySingleton();

private HungrySingleton() {}

public static HungrySingleton getInstance() {

return ***hungrySingleton***;

}

}

### 懒汉式单例模式

**类被调用时才会加载**

package com.zh.spring.test;

public class HungrySingleton {

private static final HungrySingleton ***hungrySingleton*** = new HungrySingleton();

private HungrySingleton() {}

public static HungrySingleton getInstance() {

return ***hungrySingleton***;

}

}