

手写SpringBean注册器

[简介](#)

[定义元注解](#)

[定义扫描注解](#)

[扫描类](#)

[Bean自动注册类](#)

[测试](#)

简介

在进行web开发的时候，我们经常使用@Component、@Services、@Controller等注解去声明Bean，然后Spring就会帮我们注册。

如果我们想要自己写一个组件（方便后期使用），进行无感引入的时候，就需要自己去注册Bean，所以实现自写组件第一步，先整个Bean注册器。

定义元注解

这里的元注解的意思就是类似于Controller、Service这样的注解，直接在类上加入，Spring就会帮我们自动创建。



Java

复制代码

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  public @interface Meta {
5      /**
6       * 元注解：类似于Component
7       */
8  }
```

定义扫描注解

定义完元注解后，我们还需要定义一个扫描注解，告诉Spring我要扫描哪些包里面的类。就像MapperScan一样，你里面可以没有值，但是你得加上这个注解。

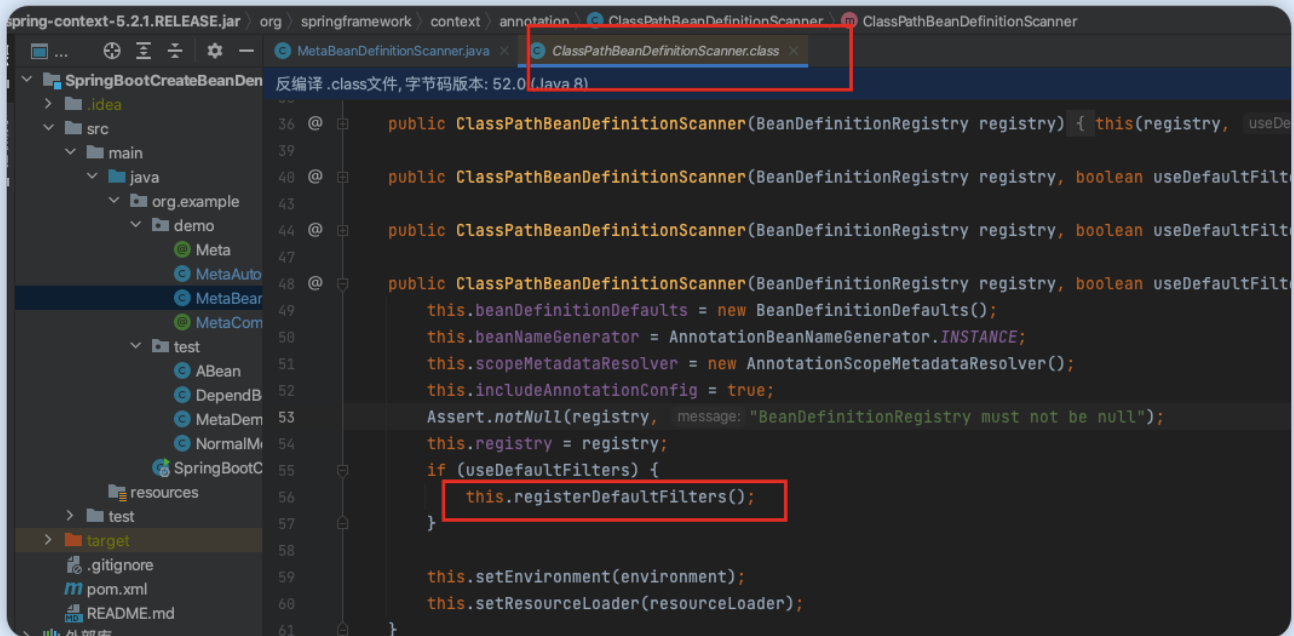
Java | 复制代码

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Import({MetaAutoConfigureRegistrar.class})
5  public @interface MetaComponentScan {
6
7      /**
8       * 当指定了值的时候，主要加载这些包路径下，包含@Meta注解的类；
9       * 如果全是默认值（即为空），则扫描这个注解所在类对应的包路径下所有包含@Meta的类
10      * @return
11      */
12      @AliasFor("basePackages") String[] value() default {};
13
14      @AliasFor("value") String[] basePackages() default {};
15
16      Class<?>[] basePackageClasses() default {};
17  }
```

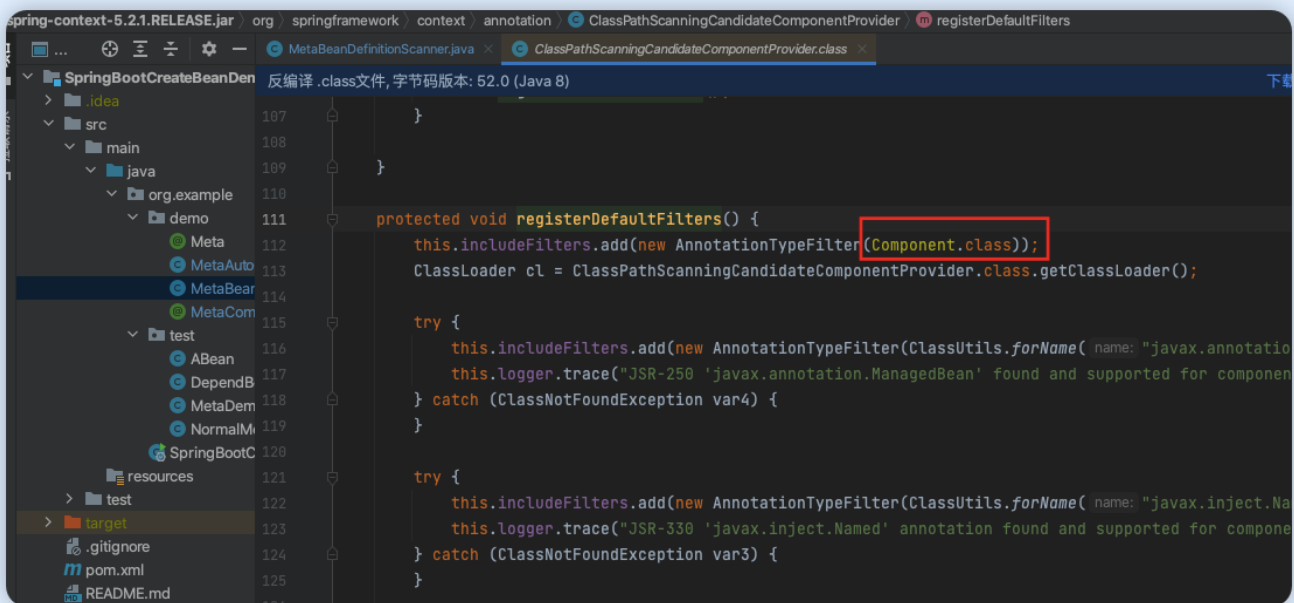
扫描类

元注解，扫描注解现在都有了。现在还需要一个扫描类，这个类的作用就是过滤类，如果类上面有元注解，那么就把类拦下来。具体怎么实现的呢？

使用ClassPathBeanDefinitionScanner类:



可以看到他原来的scanner的构造方法中有个注册默认过滤器方法registerDefaultFilters



可以看到Component注解的过滤就是在此生效的，没看到Service等注解，可能是因为这是Spring的jar包，这个没深入了解，以后再研究。

可以看到Spring原生的就是一个ClassPathScanningCandidateComponentProvider类，然后设置扫描的注解，就可以过滤出相应的类。我们按照这个逻辑去仿写。

首先继承ClassPathScanningCandidateComponentProvider，仿照他的构造函数全CV一遍。

因为它的构造方法中只有四个参数的那个有注册过滤器方法，所以我们就写这一个。

```

1
2  /**
3   * 扫描类
4   */
5  public class MetaBeanDefinitionScanner extends ClassPathBeanDefinitionScanner {
6      /**
7       * 仿照原生的构造方法全部来一遍
8       * @param registry Bean定义注册中心
9       * @param useDefaultFilters 是否使用用户默认的filter
10      * @param environment 整个spring应用运行时的环境信息：类似于application.yml 里面就是。
11      * @param resourceLoader 对资源封装的加载器：可以从文件中、网络中、流中加载资源。
12      *                      资源加载器是个接口，默认从类路径下加载。
13      *                      通过getResource方法从字符串中查找路径。
14      *                      可以从容器中获取ResourceLoader，调用getResource方法。
15      */
16      public MetaBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useDefaultFilters,
17                                      Environment environment, ResourceLoader resourceLoader) {
18          super(registry, useDefaultFilters, environment, resourceLoader);
19          registerFilters();
20      }
21
22      /**
23       * 注册过滤器 在此类的构造函数中引用
24       */
25      protected void registerFilters() {
26          //注册一个AnnotationTypeFilter，确保过滤获取所有@Meta注解的类
27          addIncludeFilter(new AnnotationTypeFilter(Meta.class));
28      }
29  }
30

```

Bean自动注册类

相关的准备工作完成了，就可以进行完成自动注册类了。

自动注册类我们需要实现三个接口。

ImportBeanDefinitionRegistrar接口：

会执行ImportBeanDefinitionRegistrar的registerBeanDefinitions方法，然后生成BeanDefinition对象，并最终注册到BeanDefinitionRegistry中，为后续实例化bean做准备。

ResourceLoaderAware接口：

获取资源加载器,可以获得外部资源文件。在Java类中可以通过文件流等的形式加载外部资源文件，并能读取资源文件中的配置，通过下面的get属性可以获取Properties中的配置属性。

▼ ResourceLoaderAware使用示例

Java

📄 复制代码

```
1 //TestBean必须实现ResourceLoaderAware接口，才能加载资源
2 public static void main(String[] args)
3 {
4     // 创建ApplicationContext容器
5     ApplicationContext ctx = new
6         ClassPathXmlApplicationContext("beans.xml");
7     // 获取容器中名为test的Bean实例
8     TestBean tb = ctx.getBean("test" , TestBean.class);
9     // 通过tb实例来获取ResourceLoader对象
10    ResourceLoader rl = tb.getResourceLoader();
11    // 判断程序获得ResourceLoader和容器是否相同
12    System.out.println(rl == ctx);
13 }
```

EnvironmentAware接口：

凡注册到Spring容器内的bean，实现了EnvironmentAware接口重写setEnvironment方法后，在工程启动时可以获得application.properties的配置文件配置的属性值。

▼ EnvironmentAware接口使用示例

Java

📄 复制代码

```
1 @Configuration
2 public class MyProjectc implements EnvironmentAware {
3     @Override
4     public void setEnvironment(Environment environment) {
5         String projectName = environment.getProperty("project.name");
6         System.out.println(projectName);
7     }
8 }
```

然后这是最终自动注册类代码，必要注释已经写进去了。

```
1 package org.example.demo;
2
3 import org.springframework.beans.factory.support.BeanDefinitionRegistry;
4 import org.springframework.context.EnvironmentAware;
5 import org.springframework.context.ResourceLoaderAware;
6 import org.springframework.context.annotation.ClassPathBeanDefinitionScanner;
7 import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;
8 import org.springframework.core.annotation.AnnotationAttributes;
9 import org.springframework.core.env.Environment;
10 import org.springframework.core.io.ResourceLoader;
11 import org.springframework.core.type.AnnotationMetadata;
12 import org.springframework.core.type.filter.AnnotationTypeFilter;
13 import org.springframework.util.ClassUtils;
14
15 import java.util.Arrays;
16 import java.util.LinkedHashSet;
17 import java.util.Set;
18
19 /**
20  * 核心类
21  * 流程就是：因为我们这里的目标是注册所有带 @Meta 注解的类，
22  * 扫描所有的类，判断是否有@Meta注解，有则通过 registry 手动注册
23  */
24
25 public class MetaAutoConfigureRegistrar
26     implements ImportBeanDefinitionRegistrar, ResourceLoaderAware, EnvironmentAware {
27
28     /**
29      *
30      */
31     private ResourceLoader resourceLoader;
32
33     private Environment environment;
34
35     /**
36      * 实现ResourceLoaderAware接口方法，可以获得外部资源xml、txt等。
37      * @param resourceLoader
38      */
39     @Override
40     public void setResourceLoader(ResourceLoader resourceLoader) {
41         this.resourceLoader = resourceLoader;
42     }
```

```

43
44  /**
45   * 实现EnvironmentAware接口方法，此属性可以获得application.properties中的属
    性值
46   * @param environment
47   */
48   @Override
49   public void setEnvironment(Environment environment) {
50       this.environment = environment;
51   }
52
53  /**
54   * 核心!! 注册bean
55   * @param importingClassMetadata 注解元数据，多半是用来获取注解的属性
56   * @param registry bean 定义注册器
57   */
58   @Override
59   public void registerBeanDefinitions(AnnotationMetadata importingClassM
60   etadata, BeanDefinitionRegistry registry) {
61       //新建一个元注解扫描类，过滤出来要进行新建的Bean
62       MetaBeanDefinitionScanner scanner =
63           new MetaBeanDefinitionScanner(registry, true, this.environ
64   ment, this.resourceLoader);
65       //设置扫描包路径
66       Set<String> packagesToScan = this.getPackagesToScan(importingClass
67   Metadata);
68       //根据扫描包路径，元注解扫描类去扫描对应的包，过滤出对应的类
69       //找到以后就会注册到Bean容器中，当然我们的代码到这里就结束了，再向下的步骤就是
70   Spring自己干了
71       scanner.scan(packagesToScan.toArray(new String[]{}));
72   }
73
74  /**
75   * 找到要扫描的包路径，set存储
76   * @param metadata 注解元数据
77   * @return
78   */
79   private Set<String> getPackagesToScan(AnnotationMetadata metadata) {
80       //获取使用元数据扫描注解的类的注解属性值
81       AnnotationAttributes attributes =
82           AnnotationAttributes.fromMap(metadata.getAnnotationAttribu
83   tes(MetaComponentScan.class.getName()));
84       //取出注解中的basePackages值
85       String[] basePackages = attributes.getStringArray("basePackages");
86       //取出注解中的basePackageClasses值
87       Class<?>[] basePackageClasses = attributes.getClassArray("basePack
88   ageClasses");
89       //新建双向链表存储 包扫描路径

```

```

84         Set<String> packagesToScan = new LinkedHashSet<>(Arrays.asList(bas
85     ePackages));
86         for (Class clz : basePackageClasses) {
87             //添加扫描路径
88             packagesToScan.add(ClassUtils.getPackageName(clz));
89         }
90     }
91     if (packagesToScan.isEmpty()) {
92         //如果到这里包扫描路径为空，说明使用MetaComponentScan注解的时候没有设定
93         值
94         //那么就按照这个注解使用地的包路径作为扫描路径
95         packagesToScan.add(ClassUtils.getPackageName(metadata.getClass
96         Name()));
97     }
98     // 返回包路径
99     return packagesToScan;
100 }

```

测试

完成代码后，开始写测试了。测试类的思路也很简单，就是整一个SpringApplication，上面加上元注解扫描注解，然后整四个普通类，对照实验加上注解。在这四个类的构造函数中，输出string，这样就可以看出来顺序和是否实现注解了。

下面是四个类

▼

Java | 复制代码

```

1  @Meta
2  public class MetaDemo {
3      public MetaDemo() {
4          System.out.println("MetaDemo类 注册成功!");
5      }
6  }

```



```
1 @Component
2 public class NormalMeta {
3     public NormalMeta() {
4         System.out.println("component注解的普通bean");
5     }
6 }
7
```

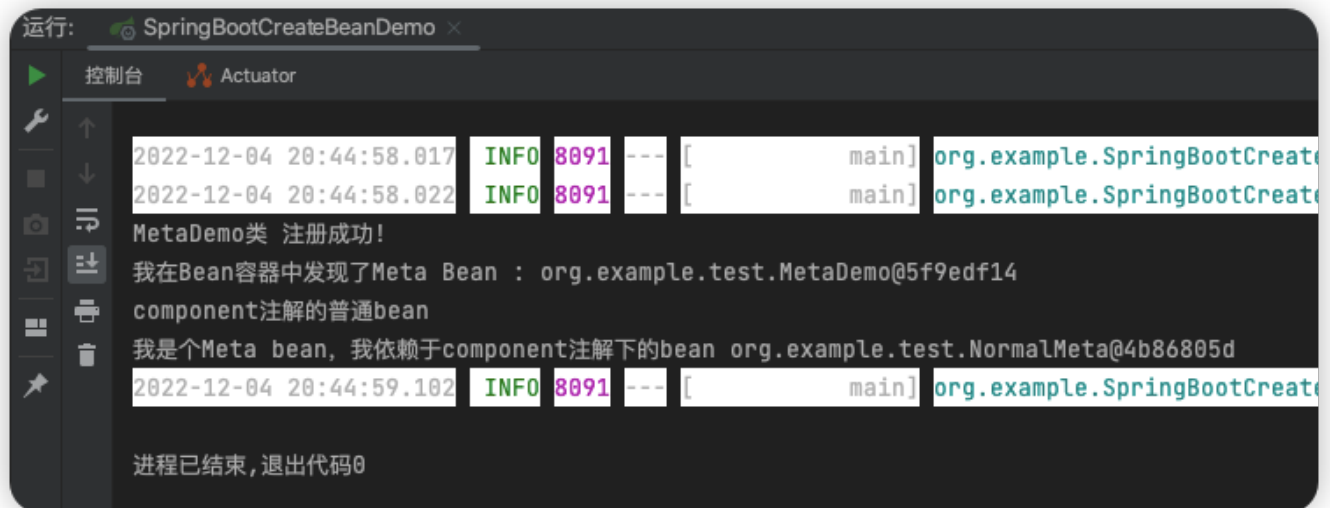
```
1 @Meta
2 public class DependBean {
3     public DependBean(NormalMeta normalBean) {
4         System.out.println("我是个Meta bean, 我依赖于component注解下的bean " +
5             normalBean);
6     }
7 }
```

```
1 @Component
2 public class ABean {
3     /**
4      * 这里的构造函数用到MetaDemo, 你的IDE可能会报错, 因为无法自动装配
5      * 因为Component等注解IDEA已经知道了, 我们自定义的注解他还不认识, 所以报错
6      * 这里如果ABean也能成功构造, 说明Bean已经注册成功了。
7      * @param metaDemo
8      */
9     public ABean(MetaDemo metaDemo) {
10         System.out.println("我在Bean容器中发现了Meta Bean : " + metaDemo);
11     }
12 }
```

Main主类

```
1 @SpringBootApplication
2 @MetaComponentScan
3 public class SpringBootCreateBeanDemo {
4     public static void main(String[] args) {
5         SpringApplication.run(SpringBootCreateBeanDemo.class, args);
6     }
7 }
```

输出结果



```
运行: SpringBootCreateBeanDemo x
控制台 Actuator
2022-12-04 20:44:58.017 INFO 8091 --- [main] org.example.SpringBootCreateBeanDemo: MetaDemo类 注册成功!
2022-12-04 20:44:58.022 INFO 8091 --- [main] org.example.SpringBootCreateBeanDemo: 我在Bean容器中发现了Meta Bean : org.example.test.MetaDemo@5f9edf14
component注解的普通bean
我是个Meta bean, 我依赖于component注解下的bean org.example.test.NormalMeta@4b86805d
2022-12-04 20:44:59.102 INFO 8091 --- [main] org.example.SpringBootCreateBeanDemo: 进程已结束,退出代码0
```

可以看到顺序，单纯的使用元注解Meta的类先注册，然后Component注解注册，最后才是依赖于普通类的Meta类。至于为什么自定义的先注册这个我还没有搞懂顺序。

至此结束。