

@Import注解：导入配置类的四种方式&源码解析


码农StayUp
[关注](#)
2021.03.08 08:04:39
字数 1,702
阅读 267



微信搜索：码农StayUp
主页地址：<https://gozhuyinglong.github.io>
源码分享：<https://github.com/gozhuyinglong/blog-demos>

平时喜欢看源码的小伙伴，应该知道Spring中大量使用了 `@Import` 注解。该注解是Spring用来导入配置类的，等价于 `Spring XML` 中的 `<import/>` 元素。

本文将对该注解进行介绍，并通过实例演示它导入配置类的四种方式，最后对该注解进行源码解析。

话不多说，走起~

简介

`@Import` 注解的全类名是 `org.springframework.context.annotation.Import` 。其只有一个默认的 `value` 属性，该属性类型为 `Class<?>[]` ，表示可以传入一个或多个 `Class` 对象。

通过注释可以看出，该注解有如下作用：

- 可以导入一个或多个组件类（通常是 `@Configuration` 配置类）
- 该注解的功能与 `Spring XML` 中的 `<import/>` 元素相同。可以导入 `@Configuration` 配置类、`ImportSelect` 和 `ImportBeanDefinitionRegistrar` 的实现类。从4.2版本开始，还可以引用常规组件类（普通类），该功能类似于 `AnnotationConfigApplicationContext.register` 方法。
- 该注解可以在类中声明，也可以在元注解中声明。
- 如果需要导入 `XML` 或其他非 `@Configuration` 定义的资源，可以使用 `@ImportResource` 注释。


导入配置类的四种方式

源码注释写得很清楚，该注解有四种导入方式：

- 普通类
- `@Configuration` 配置类
- `ImportSelector`的实现类
- `ImportBeanDefinitionRegistrar`的实现类



便宜的云电脑


码农StayUp
总资产8
[关注](#)

一文搞懂单向散列函数
阅读 12

浅聊Linux的五种IO模型
阅读 152

JDK动态代理：不仅要学会用，更要掌握其原理
阅读 195

推荐阅读

Mybatis实现原理
阅读 812

Android之注解的使用介绍
阅读 474

Spring中@Component和@Bean的区别
阅读 513

Spring全家桶--单数据源的配置
阅读 263

Dart：通过注解生成代码
阅读 237



进销存ERP系统



下面我们逐个来介绍~

准备工作

创建四个配置类：ConfigA、ConfigB、ConfigC、ConfigD。其中ConfigB中增加 `@Configuration` 注解，表示为配置类，其余三个均为普通类。

ConfigA:

```
1 public class ConfigA {
2
3     public void print() {
4         System.out.println("输出: ConfigA.class");
5     }
6 }
```

ConfigB:

```
1 @Configuration
2 public class ConfigB {
3
4     public void print() {
5         System.out.println("输出: ConfigB.class");
6     }
7
8 }
```

ConfigC:

```
1 public class ConfigC {
2
3     public void print() {
4         System.out.println("输出: ConfigC.class");
5     }
6
7 }
```

ConfigD:

```
1 public class ConfigD {
2
3     public void print() {
4         System.out.println("输出: ConfigD.class");
5     }
6
7 }
```

再创建一个主配置类Config，并试图通过 `@Resource` 注解将上面四个配置类进行注入。当然，这样是不成功的，还需要将它们进行导入。

```
1 @Configuration
2 public class Config {
3
4     @Resource
5     ConfigA configA;
6
7     @Resource
8     ConfigB configB;
9
10    @Resource
11    ConfigC configC;
12
13    @Resource
14    ConfigD configD;
15 }
```



赞



赞赏



更多好文





赞



赞赏



更多好文

```
20
17     public void print() {
18         configA.print();
19         configB.print();
20         configC.print();
21         configD.print();
22     }
23 }
```

```
public class Car1 {
    public void say(){
        System.out.println("i am car1");
    }
}
```

方式一：导入普通类

导入普通类非常简单，只需在 `@Import` 传入类的 `Class` 对象即可。

```
1 @Configuration
2 @Import(ConfigA.class) //相当于@Bean，就是将一个类放入容器中，在没有加@ComponentScan，@Component注解的情况下，该类进入了容器
3 public class Config {
4     ...
5 }
```

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
Car1 car1 = context.getBean(Car1.class);
car1.say();
```

方式二：导入 @Configuration 配置类

导入配置类与导入普通类一样，在 `@Import` 注解中传入目标类的 `Class` 对象。

```
1 @Configuration
2 @Import({ConfigA.class,
3         ConfigB.class})
4 public class Config {
5     ...
6 }
```

方式三：导入 ImportSelector 的实现类

`ImportSelector` 接口的全类名为 `org.springframework.context.annotation.ImportSelector`。其主要作用的是收集需要导入的配置类，并根据条件来确定哪些配置类需要被导入。实现该接口，重写其中的 `selectImports` 方法，返回由需要导入的Java类的全限定性名称组成的字符串数组

该接口的实现类同时还可以实现以下任意一个 `Aware` 接口，它们各自的方法将在 `selectImport` 之前被调用：

- `EnvironmentAware`
- `BeanFactoryAware`
- `BeanClassLoaderAware`
- `ResourceLoaderAware`

另外，该接口实现类可以提供一个或多个具有以下形参类型的构造函数：

- `Environment`
- `BeanFactory`
- `ClassLoader`
- `ResourceLoader`

如果你想要推迟导入配置类，直到处理完所有的 `@Configuration`。那么你可以使用 `DeferredImportSelector`

下面我们创建一个实现该接口的类 `MyImportSelector`。





赞



赞赏



更多好文

```
1 public class MyImportSelector implements ImportSelector {
2
3     @Override
4     public String[] selectImports(AnnotationMetadata importingClassMetadata) {
5         return new String[]{"io.github.gozhuyinglong.importanalysis.config.ConfigC"};
6     }
7 }
```

在配置类 Config 中导入 MyImportSelector 类。

```
1 @Configuration
2 @Import({ConfigA.class,
3         ConfigB.class,
4         MyImportSelector.class})
5 public class Config {
6     ...
7 }
```

方式四：导入 ImportBeanDefinitionRegistrar 的实现类

该接口的目的是有选择性的进行注册 Bean，注册时可以指定 Bean 名称，并且可以定义bean的级别。其他功能与 ImportSelector 类似，这里就不再赘述。

下面来看示例：

创建一个实现 ImportBeanDefinitionRegistrar 接口的类 MyImportBeanDefinitionRegistrar，并在 registerBeanDefinitions 方法中注册 configD 类。

入参 AnnotationMetadata 为主配置类 Config 的注解元数据； BeanDefinitionRegistry 参数可以注册 Bean 的定义信息。

```
1 public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
2
3     @Override
4     public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
5         registry.registerBeanDefinition("configD", new RootBeanDefinition(ConfigD.class));
6     }
7 }
```

在配置类 Config 中导入 MyImportBeanDefinitionRegistrar 类。

```
1 @Configuration
2 @Import({ConfigA.class,
3         ConfigB.class,
4         MyImportSelector.class,
5         MyImportBeanDefinitionRegistrar.class})
6 public class Config {
7     ...
8 }
```

测试结果

创建一个测试类 ImportDemo，看上面四个配置类是否被注入。

```
1 public class ImportDemo {
2
3     public static void main(String[] args) {
4         AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigA.class, ConfigB.class, ConfigC.class, ConfigD.class, MyImportBeanDefinitionRegistrar.class);
5     }
6 }
```

```
public class Car3 {
    public void say(){
        System.out.println("i am car3");
    }
}
-----
public class CustomImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar{
    @Override
    public void registerBeanDefinitions(AnnotationMetadata annotationMetadata, BeanDefinitionRegistry beanDefinitionRegistry) {

        RootBeanDefinition beanDefinition = new RootBeanDefinition(Car3.class);
        beanDefinitionRegistry.registerBeanDefinition("car3",beanDefinition);
    }
}
-----
@Configuration
@Import(value = {Car1.class, CustomImportSelector.class, CustomImportBeanDefinitionRegistrar.class})
public class AutoConfig {
}
-----
public class Test {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AutoConfig.class);
        Car3 car3 = context.getBean(Car3.class);
        car3.say();
    }
}
```





赞



赞赏



更多好文

输出结果：

```
1 | 输出: ConfigA.class
2 | 输出: ConfigB.class
3 | 输出: ConfigC.class
4 | 输出: ConfigD.class
```

通过输出结果可以看出，这四个配置类被导入到主配置类中，并成功注入。

源码解析

`ConfigurationClassParser` 类为Spring的工具类，主要用于分析配置类，并产生一组 `ConfigurationClass` 对象（因为一个配置类中可能会通过 `@Import` 注解来导入其它配置类）。也就是说，其会递归的处理所有配置类。

doProcessConfigurationClass

其中的 `doProcessConfigurationClass` 方法是处理所有配置类的过程，其按下面步骤来处理：

1. `@Component`注解
2. `@PropertySource`注解
3. `@ComponentScan`注解
4. `@Import`注解
5. `@ImportResource`注解
6. `@Bean`注解
7. 配置类的接口上的默认方法
8. 配置类的超类

```
1 | @Nullable
2 | protected final SourceClass doProcessConfigurationClass(
3 |     ConfigurationClass configClass, SourceClass sourceClass, Predicate<String> filter)
4 |     throws IOException {
5 |
6 |     if (configClass.getMetadata().isAnnotated(Component.class.getName())) {
7 |         // 1. 首先会递归的处理所有成员类，即@Component注解
8 |         processMemberClasses(configClass, sourceClass, filter);
9 |     }
10 |
11 |     // 2. 处理所有@PropertySource注解
12 |     for (AnnotationAttributes propertySource : AnnotationConfigUtils.attributesForRepeatable(
13 |         sourceClass.getMetadata(), PropertySources.class,
14 |         org.springframework.context.annotation.PropertySource.class)) {
15 |         if (this.environment instanceof ConfigurableEnvironment) {
16 |             processPropertySource(propertySource);
17 |         }
18 |         else {
19 |             logger.info("Ignoring @PropertySource annotation on [" + sourceClass.getMetadata()
20 |                 + "]. Reason: Environment must implement ConfigurableEnvironment");
21 |         }
22 |     }
23 |
24 |     // 3. 处理所有@ComponentScan注解
25 |     Set<AnnotationAttributes> componentScans = AnnotationConfigUtils.attributesForRepeatable(
26 |         sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);
27 |     if (!componentScans.isEmpty() &&
28 |         !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(), ConfigurationPhase.REGI
29 |         for (AnnotationAttributes componentScan : componentScans) {
30 |             // 配置类的注解为@ComponentScan-> 立即执行扫描
31 |             Set<BeanDefinitionHolder> scannedBeanDefinitions =
32 |                 this.componentScanParser.parse(componentScan, sourceClass.getMetadata().getCla
33 |             // 检查扫描过的BeanDefinition集合，看看是否有其他配置类，如果有里面，递归扫描
```





赞



赞赏



更多好文

```
39         }
40         parse(bdCand.getBeanClassName(), holder.getBeanName());
41     }
42 }
43 }
44 }
45
46 // 4. 处理所有@Import注解
47 processImports(configClass, sourceClass, getImports(sourceClass), filter, true);
48
49 // 5. 处理所有@ImportResource注解
50 AnnotationConfigUtils.importResource =
51     AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(), ImportResource.class);
52 if (importResource != null) {
53     String[] resources = importResource.getStringArray("locations");
54     Class<? extends BeanDefinitionReader> readerClass = importResource.getClass("reader");
55     for (String resource : resources) {
56         String resolvedResource = this.environment.resolveRequiredPlaceholders(resource);
57         configClass.addImportedResource(resolvedResource, readerClass);
58     }
59 }
60
61 // 6. 处理标注为@Bean注解的方法
62 Set<MethodMetadata> beanMethods = retrieveBeanMethodMetadata(sourceClass);
63 for (MethodMetadata methodMetadata : beanMethods) {
64     configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));
65 }
66
67 // 7. 处理配置类的接口上的默认方法
68 processInterfaces(configClass, sourceClass);
69
70 // 8. 处理配置类的超类（如果有的话）
71 if (sourceClass.getMetadata().hasSuperClass()) {
72     String superclass = sourceClass.getMetadata().getSuperClassName();
73     if (superclass != null && !superclass.startsWith("java") &&
74         !this.knownSuperclasses.containsKey(superclass)) {
75         this.knownSuperclasses.put(superclass, configClass);
76         // Superclass found, return its annotation metadata and recurse
77         return sourceClass.getSuperClass();
78     }
79 }
80
81 // 处理完成
82 return null;
83 }
```

processImports

`processImports` 方法为处理 `@Import` 注解导入的配置类，是我们本篇的主题。

该方法会循环处理每一个由 `@Import` 导入的类：

1. `ImportSelector`类的处理
2. `ImportBeanDefinitionRegistrar`类的处理
3. 其它类统一按照 `@Configuration` 类来处理，所以加不加 `@Configuration` 注解都能被导入

```
1 /**
2  * 处理配置类上的@Import注解引入的类
3  *
4  * @param configClass 配置类，这里是Config类
5  * @param currentSourceClass 当前资源类
6  * @param importCandidates 该配置类中的@Import注解导入的候选类列表
7  * @param exclusionFilter 排除过滤器
8  * @param checkForCircularImports 是否循环检查导入
9  */
10 private void processImports(ConfigurationClass configClass, SourceClass currentSourceClass,
11                             Collection<SourceClass> importCandidates, Predicate<String> exclus
12                             boolean checkForCircularImports) {
13     // 如果该@Import注解导入的列表为空，直接返回
14     if (importCandidates.isEmpty()) {
15         return;
```





赞



赞赏



更多好文

```
22         this.importStack.push(configClass);
23     try {
24         // 循环处理每一个由@Import导入的类
25         for (SourceClass candidate : importCandidates) {
26             if (candidate.isAssignable(ImportSelector.class)) {
27                 // 1. ImportSelector类的处理
28                 Class<?> candidateClass = candidate.loadClass();
29                 ImportSelector selector = ParserStrategyUtils.instantiateClass(candidateCl
30                                                                 this.enviro
31
32                 Predicate<String> selectorFilter = selector.getExclusionFilter();
33                 if (selectorFilter != null) {
34                     exclusionFilter = exclusionFilter.or(selectorFilter);
35                 }
36                 if (selector instanceof DeferredImportSelector) {
37                     // 1.1 若是DeferredImportSelector接口的实现，则延时处理
38                     this.deferredImportSelectorHandler.handle(configClass, (DeferredImport
39                 }
40                 else {
41                     // 1.2 在这里调用我们的ImportSelector实现类的selectImports方法
42                     String[] importClassNames = selector.selectImports(currentSourceClass.);
43                     Collection<SourceClass> importSourceClasses = asSourceClasses(importCl
44                     // 1.3 递归处理每一个selectImports方法返回的配置类
45                     processImports(configClass, currentSourceClass, importSourceClasses, e
46                 }
47             }
48             else if (candidate.isAssignable(ImportBeanDefinitionRegistrar.class)) {
49                 // 2. ImportBeanDefinitionRegistrar类的处理
50                 Class<?> candidateClass = candidate.loadClass();
51                 ImportBeanDefinitionRegistrar registrar =
52                     ParserStrategyUtils.instantiateClass(candidateClass, ImportBeanDefinit
53                                                                 this.environment, this.resourceLo
54                 configClass.addImportBeanDefinitionRegistrar(registrar, currentSourceClass
55             }
56             else {
57                 // 3. 其它类统一按照@Configuration类来处理，所以加不加@Configuration注解都能被
58                 this.importStack.registerImport(
59                     currentSourceClass.getMetadata(), candidate.getMetadata().getClassName
60                     processConfigurationClass(candidate.asConfigClass(configClass), exclusionF
61                 }
62             }
63         }
64     } catch (BeanDefinitionStoreException ex) {
65         throw ex;
66     }
67     catch (Throwable ex) {
68         throw new BeanDefinitionStoreException(
69             "Failed to process import candidates for configuration class [" +
70             configClass.getMetadata().getClassName() + "]", ex);
71     }
72     finally {
73         this.importStack.pop();
74     }
75 }
```

总结

通过上面源码的解析可以看出， `@Import` 注解主要作用是导入外部类的，并且普通类也会按照 `@Configuration` 类来处理。这大大方便了我们将自己的组件类注入到容器中了（无需修改自己的组件类）。

源码分享

完整代码请访问我的Github，若对你有帮助，欢迎给个☆，感谢~~🐼🐼🐼

<https://github.com/gozhuyinglong/blog-demos/tree/main/spring-source-analysis/src/main/java/io/github/gozhuyinglong/importanalysis>



参与评论

写下你的评论...

评论0 赞