开课吧 kaikeba.com

快速开发框架 Spring Boot

课程讲义

主讲：Reythor 雷

2019

快速开发框架 Spring Boot

Spring Boot 工作原理解析

1.1 自动配置源码解析

使用 Spring Boot 开发较之以前的基于 xml 配置式的开发，要简捷方便快速的多。而这完全得益于 Spring Boot 的自动配置。下面就通过源码阅读方式来分析自动配置的运行原理。

1.1.1 解析@SpringBootApplication

打开启动类的@SpringBootApplication 注解源码。

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

我们发现@SpringBootApplication 注解其实就是一个组合注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes =
        @Filter(type = FilterType.CUSTOM, classes =
public @interface SpringBootApplication {
```

（1） 元注解

前四个是专门（即只能）用于对注解进行注解的，称为元注解。

（2） @SpringBootConfiguration

查看该注解的源码注解可知，该注解与@Configuration 注解功能相同，仅表示当前类为一个 JavaConfig 类，其就是为 Spring Boot 创建的一个注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

（3） @ComponentScan

用于指定当前应用所要扫描的包。注意，其仅仅是指定包，而并没有扫描这些包，更没有装配其中的类，这个真正扫描并装配这些类是@EnableAutoConfiguration 完成的。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {
```

该注解有三个重要属性：

- basePackages 属性：用于指定要扫描的组件包，若没有指定则扫描当前注解所标的类所在的包及其子孙包。
- includeFilters 属性：用于进一步缩小要扫描的基本包中的类，通过指定过滤器的方式进行缩小范围。
- excludeFilters 属于：用于过滤掉那些不适合做组件的类。

（4） @EnableXxx

@EnableXxx 注解一般用于开启某一项功能，是为了简化配置代码的引入。其是组合注

解，一般情况下@EnableXxx 注解中都会组合一个@Import 注解，而该@Import 注解用于导入指定的类，而该被导入的类一般为配置类。其导入配置类的方式常见的有三种：

A、 直接引入配置类

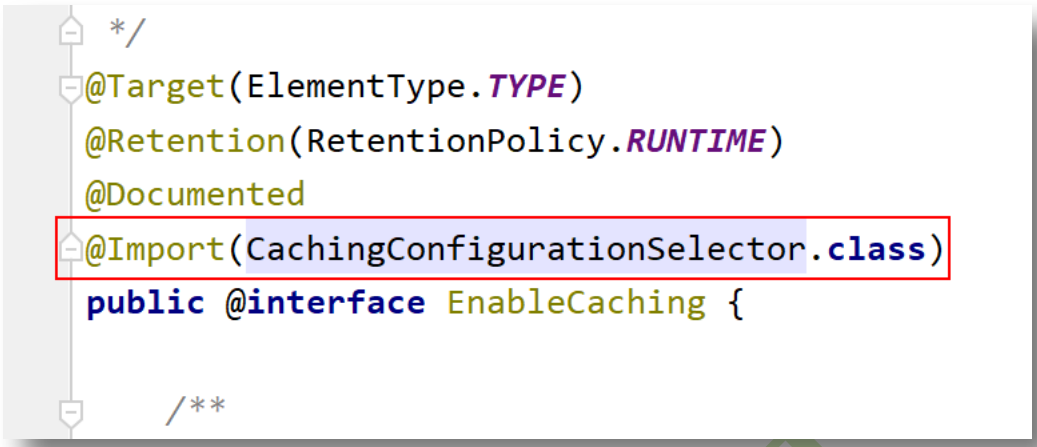
@Import 中指定的类一般为 Configuration 结尾，且该类上会注解@Configuration，表示当前类为 JavaConfig 类。

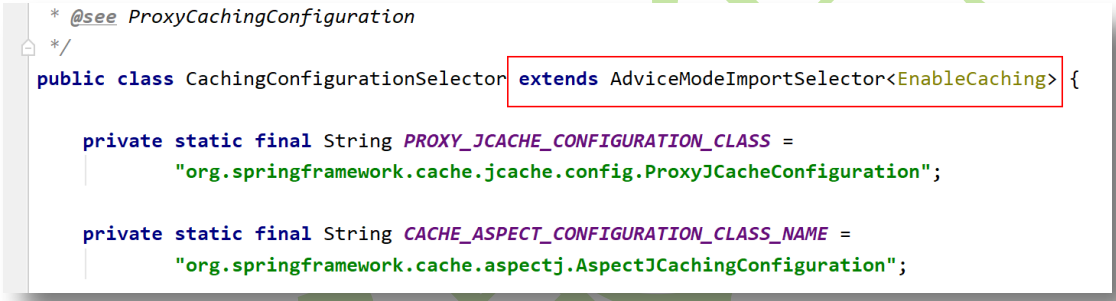
```
204      */
205     @Target(ElementType.TYPE)
206     @Retention(RetentionPolicy.RUNTIME)
207     @Import(SchedulingConfiguration.class)
208     @Documented
209     public @interface EnableScheduling {
210
211     }
212
```

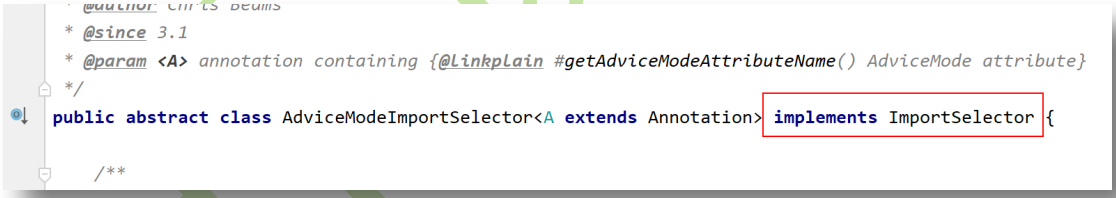
```
*/
@Configuration
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public class SchedulingConfiguration {

    @Bean(name = TaskManagementConfigUtils.SCHEDULED_ANNOTATION_PROCESSOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public ScheduledAnnotationBeanPostProcessor scheduledAnnotationProcessor() {
        return new ScheduledAnnotationBeanPostProcessor();
    }

}
```

B、 根据条件选择配置类

@Import 中指定的类一般以 ConfigurationSelector 结尾，且该类实现了 ImportSelector 接口，表示当前类会根据条件选择不同的配置类导入。







C、 动态注册 Bean

@Import 中指定的类一般以 Registrar 结尾，且该类实现了 ImportBeanDefinitionRegistrar 接口，用于表示在代码运行时若使用了到该配置类，则系统会自动将其导入。

```
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {
```

```
 * @see EnableAspectJAutoProxy
 */
class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    /**
```

```
@Override
public void registerBeanDefinitions(
        AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

    AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

    AnnotationAttributes enableAspectJAutoProxy =
            AnnotationConfigUtils.attributesFor(importingClassMetadata, EnableAspectJAutoProx
    if (enableAspectJAutoProxy != null) {
        if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
            AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
        }
        if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
            AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
        }
    }
}
```

1.1.2 解析@EnableAutoConfiguration

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

该注解用于完成自动配置，是 Spring Boot 的核心注解，是一个组合注解。所谓自动配置是指，将用户自定义的类及框架本身用到的类进行装配。其中最重要的注解有两个：

- @AutoConfigurationPackage：用于导入并装配用户自定义类，即自动扫描包中的类
- @Import：用于导入并装配框架本身的类

（1） @Import

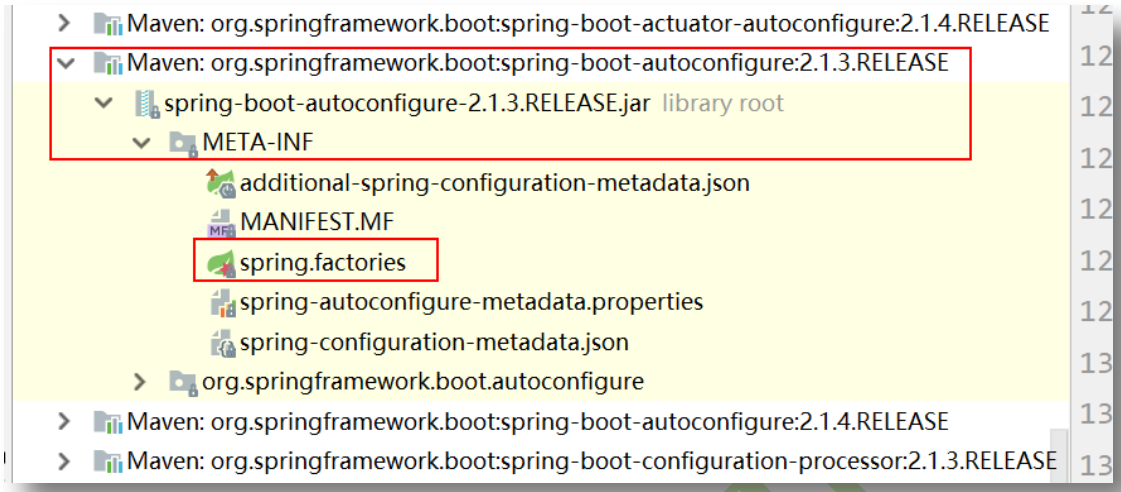
该注解用于导入指定的类。其参数 AutoConfigurationImportSelector 类，该类用于导入自动配置的类。

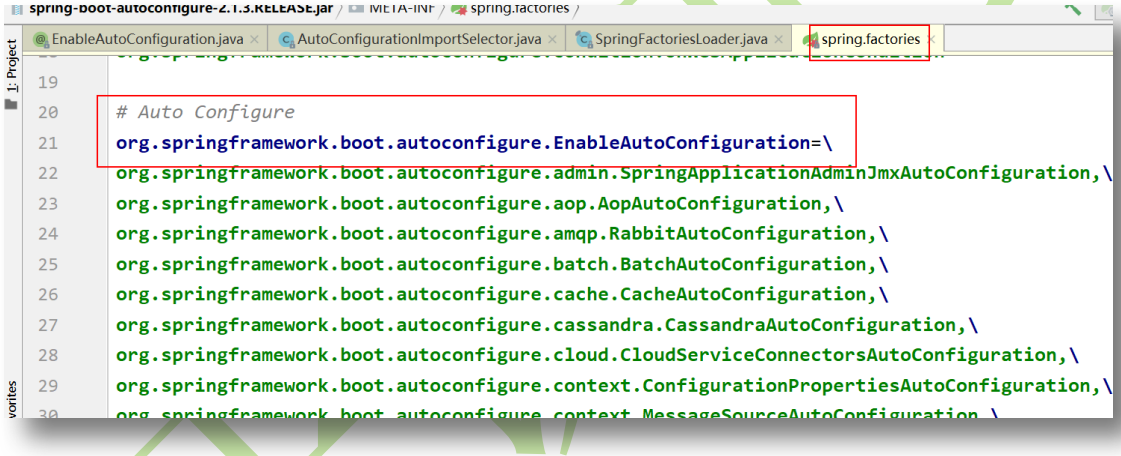
```
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.ena

    /**
```

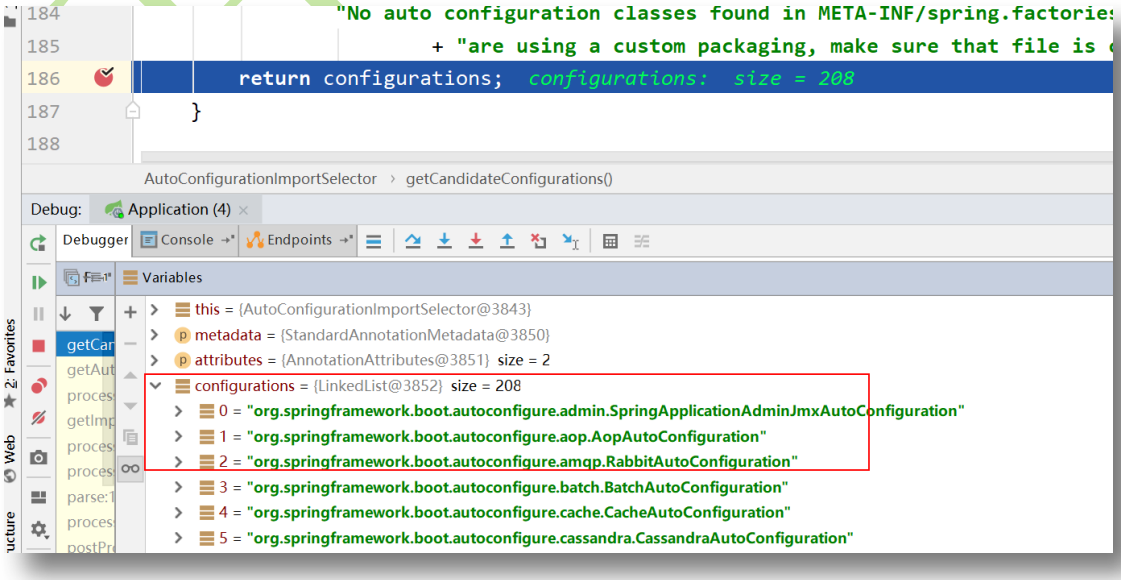
```
*/
public class AutoConfigurationImportSelector
        implements DeferredImportSelector, BeanClassLoaderAware, ResourceLoaderAware,
        BeanFactoryAware, EnvironmentAware, Ordered {

    private static final AutoConfigurationEntry EMPTY_ENTRY = new AutoConfigurationEntry
```

```
170     /**
171      * Return the auto-configuration class names that should be considered. By default
172      * this method will load candidates using {@Link SpringFactoriesLoader} with
173      * {@Link #getSpringFactoriesLoaderFactoryClass()}.
174      * @param metadata the source metadata
175      * @param attributes the {@Link #getAttributes(AnnotationMetadata) annotation
176      * attributes}
177      * @return a list of candidate configurations
178      */
179     protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
180             AnnotationAttributes attributes) {
181         List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
182                 getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
183         Assert.notEmpty(configurations,
184                 "No auto configuration classes found in META-INF/spring.factories. If you "
185                         + "are using a custom packaging, make sure that file is correct.");
186         return configurations;
187     }
```

```
120     public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoa
121         String factoryClassName = factoryClass.getName();
122         return loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collectior
123     }
124
125     private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader cl
126         MultiValueMap<String, String> result = cache.get(classLoader);
127         if (result != null) {
128             return result;
129         }
130
131         try {
132             Enumeration<URL> urls = (classLoader != null ?
133                     classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
134                     ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
135             result = new LinkedMultiValueMap<>();
136             while (urls.hasMoreElements()) {
```


Maven: org.springframework.boot:spring-boot-actuator-autoconfigure:2.1.4.RELEASE
Maven: org.springframework.boot:spring-boot-autoconfigure:2.1.3.RELEASE
spring-boot-autoconfigure-2.1.3.RELEASE.jar library root
META-INF
additional-spring-configuration-metadata.json
MANIFEST.MF
spring.factories
spring-autoconfigure-metadata.properties
spring-configuration-metadata.json
org.springframework.boot.autoconfigure
Maven: org.springframework.boot:spring-boot-autoconfigure:2.1.4.RELEASE
Maven: org.springframework.boot:spring-boot-configuration-processor:2.1.3.RELEASE


EnableAutoConfiguration.java
AutoConfigurationImportSelector.java
SpringFactoriesLoader.java
spring.factories
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudServiceConnectorsAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\


"No auto configuration classes found in META-INF/spring.factories
+ "are using a custom packaging, make sure that file is
return configurations; configurations: size = 208
}
AutoConfigurationImportSelector › getCandidateConfigurations()
Debug: Application (4)
Debugger
Console
Endpoints
Variables
this = {AutoConfigurationImportSelector@3843}
metadata = {StandardAnnotationMetadata@3850}
attributes = {AnnotationAttributes@3851} size = 2
configurations = {LinkedList@3852} size = 208
0 = "org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration"
1 = "org.springframework.boot.autoconfigure.aop.AopAutoConfiguration"
2 = "org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration"
3 = "org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration"
4 = "org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration"
5 = "org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration"

（2） @AutoConfigurationPackage

再打开@AutoConfigurationPackage 的源码，其也包含一个@Import 注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {

}
```

该注解是要将 AutoConfigurationPackages 类的内部静态类 Registrar 导入。从前面的学习可知，其是一个动态注册 Bean。