SpringBoot SpringApplication底层源码分析与自动装配

目录

抛出问题

```
@SpringBootApplication
public class LearnspringbootApplication {

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

大家可以看到，如上所示是一个很常见的SpringBoot启动类，我们可以看到仅仅使用了一个Main方法就启动了整个SpringBoot项目是不是很神奇，下面我们来仔细剖析以下这一段代码，这段代码中我们可以仔细地观察到最重要的两个部分，分别是`@SpringBootApplication`注解和`SpringApplication`这个类。

- @SpringBootApplication注解
- SpringApplication类

@SpringBootApplication注解剖析

打开这个`@SpringBootApplication`注解，如下所示

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

    @AliasFor(annotation = EnableAutoConfiguration.class)
    Class<?>[] exclude() default {};


    @AliasFor(annotation = EnableAutoConfiguration.class)
    String[] excludeName() default {};


    @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
    String[] scanBasePackages() default {};

    @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
    Class<?>[] scanBasePackageClasses() default {};

}
```

我们可以发现；我们可以在`@SpringBootApplication`注解中使用`exclude()`,`excludeName()`,`scanBasePackages()`,`scanBasePackageClasses()` 这四个方法来进行自定义我们需要排除装配的Bean，扫描包路径，扫描类路径。

搭眼一瞅，在`@SpringBootApplication`注解上还有下面那么多注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

```
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
```

这四个注解不用看，就是关于注解的一些定义

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
```

`@SpringBootConfiguration`注解我们点进去看一下，发现是下面这个样子，@SpringBootConfiguration注解上又标注了@Configuration注解，想必在@Configuration注解上也标注了@Component注解，这不就是我们上一章节说的Spring的模式注解。总的来说嘛，SpringBootConfiguration注解的作用就是把类变成可以被Spring管理的Bean

结论 1 ：也就是说标注`@SpringBootApplication`注解的类会成为Spring的Bean

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

}
```

我们再来看一下`@EnableAutoConfiguration`，从名字上我们就可以看到"启用自动装配"的意思。那我们可要仔细看一下。从下面我们可以看到只有两个我们需要了解的注解，分别是@`AutoConfigurationPackage`和@`Import`注解。

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

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";


    Class<?>[] exclude() default {};


    String[] excludeName() default {};

}
```

当我点进去@`AutoConfigurationPackage`注解中，发现该注解又启用了一个模块装配`@Import(AutoConfigurationPackages.Registrar.class)`（上一章节有讲到）；所以又点进去`AutoConfigurationPackages.Registrar.class`这个类，发现这个类又实现了两个接口`ImportBeanDefinitionRegistrar`和`DeterminableImports` 在该类的实现方法中的`PackageImport()`；最后终于发现了下面这段代码。

```
    PackageImport(AnnotationMetadata metadata) {
            this.packageName = ClassUtils.getPackageName(metadata.getClassName());
        }
```

结论 2 :SpringBoot默认会装配启动类路径的所有包下可装配的Bean；也就是说如果你把SpringBoot启动类放在一个单独的包中，则SpringBoot不会装配到你的其他Bean。这时候你就要使用`@SpringBootApplication`的`scanBasePackages()`方法进行另行配置。

此时`@EnableAutoConfiguration`注解仅仅就剩下`@Import(AutoConfigurationImportSelector.class)`没有看了，不过从注解上我们可以看到使用`@Import`注解，所以可以知道SpringBoot使用的是模块装配的接口实现方式。所以我么针对`AutoConfigurationImportSelector`这个类仔细剖析一下。`AutoConfigurationImportSelector` -> `AutoConfigurationImportSelector.selectImports()` ->`getAutoConfigurationEntry()` -> `getCandidateConfigurations()` ->`SpringFactoriesLoader.loadFactoryNames()` -> `loadFactoryNames()` ->`loadSpringFactories()`；哈哈果然源码不经扒；看下面源码；`META-INF/spring.factories`这不就是配置SpringBoot自动配置的文件嘛。

```
public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";

Enumeration<URL> urls = (classLoader != null ?
                    classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
                    ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
```

结论 3 ：SpringBoot在启动时会自动加载Classpth路径下的`META-INF/spring.factories`文件，所以我们可以将需要自动配置的Bean写入这个文件，SPringBoot会替我们自动装配。这也正是配置SpringBoot自动配置的步骤。

SpringApplication类剖析

```
@SpringBootApplication
public class LearnspringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(LearnspringbootApplication.class, args);
    }
}
```
在Main方法中我们可以看到SpringApplication作为一个启动类来启动SpringBoot应用程序。那么SpringApplication类是如何进行启动整个应用程序的呢？

第一步：配置SpringBoot Bean来源

```
    public SpringApplication(Class<?>... primarySources) {
        this(null, primarySources);
    }
```
从SpringApplication类的构造方法中我们可以看到，这里传入了一个主Bean来源；因为我们将标注了`@SpringBootApplication`注解的`LearnspringbootApplication.class`传递了进来，所以`@SpringBootApplication`扫描到的Bean和自动装配的Bean会作为主Bean来源。当然我们可以调用该类的`setSources()`方法设置自己的SpringXML配置。

第二步 ：自动推断SpringBoot的应用类型

```
this.webApplicationType = WebApplicationType.deduceFromClasspath();
```
从上面一句代码（既SpringApplication初始化方法中一行代码）；我们观察`deduceFromClasspath()`方法可以看到，SpringBoot判断类路径下是否存在下面类进而判断SpringBoot的应用类型。三种类型分别是`NONE`，`SERVLET`，`REACTIVE` 。当然我们也可以调用`setWebApplicationType()`自行设置。

```
    private static final String[] SERVLET_INDICATOR_CLASSES = { "javax.servlet.Servlet",
            "org.springframework.web.context.ConfigurableWebApplicationContext" };

    private static final String WEBMVC_INDICATOR_CLASS = "org.springframework."
            + "web.servlet.DispatcherServlet";

    private static final String WEBFLUX_INDICATOR_CLASS = "org."
            + "springframework.web.reactive.DispatcherHandler";

    private static final String JERSEY_INDICATOR_CLASS = "org.glassfish.jersey.servlet.ServletContainer";

    private static final String SERVLET_APPLICATION_CONTEXT_CLASS =
"org.springframework.web.context.WebApplicationContext";

    private static final String REACTIVE_APPLICATION_CONTEXT_CLASS =
"org.springframework.boot.web.reactive.context.ReactiveWebApplicationContext";
```

第三步：推断SpringBoot的引导类

```
this.mainApplicationClass = deduceMainApplicationClass();
```
从上面一句代码（既SpringApplication初始化方法中一行代码）；我们可以通过`deduceMainApplicationClass()`方法可以看到SpringBoot根据Main 线程执行堆栈判断实际的引导类。（PS:存在一种情况就是标注`@SpringBootApplication`注解的并不是引导类情况）

```
    private Class<?> deduceMainApplicationClass() {
        try {
            StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
            for (StackTraceElement stackTraceElement : stackTrace) {
                if ("main".equals(stackTraceElement.getMethodName())) {
                    return Class.forName(stackTraceElement.getClassName());
                }
            }
        }
        catch (ClassNotFoundException ex) {
            // Swallow and continue
        }
        return null;
    }
```

第四步：加载应用上下文初始化器

```
setInitializers((Collection) getSpringFactoriesInstances(
                ApplicationContextInitializer.class));
```
从上面一句代码（既SpringApplication初始化方法中一行代码）；我们发现`setInitializers()`方法会调用`getSpringFactoriesInstances()` -> `getSpringFactoriesInstances( )` -> `getSpringFactoriesInstances()` 该方法会使用`SpringFactoriesLoader`类记进行加载配置资源既`META-INF/spring.factories`，利用 Spring 工厂加载机制,实例化 ApplicationContextInitializer 实现类,并排序对象集合。并使用`AnnotationAwareOrderComparator`类的`sort( )`方法进行排序。

```
    private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
            Class<?>[] parameterTypes, Object... args) {
        ClassLoader classLoader = getClassLoader();
        // Use names and ensure unique to protect against duplicates
        Set<String> names = new LinkedHashSet<>(
                SpringFactoriesLoader.loadFactoryNames(type, classLoader));
        List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
                classLoader, args, names);
        AnnotationAwareOrderComparator.sort(instances);
        return instances;
    }
```

第五步 ：加载应用事件监听器

```
setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
```

从上面一句代码（既SpringApplication初始化方法中一行代码）；从`setListeners()` 方法中可以看到该方法仍然调用`getSpringFactoriesInstances()`方法，不同的是利用 Spring 工厂加载`META-INF/spring.factories`,实例化 `ApplicationListener` 实现类,并排序对象集合

第六步：启动SpringApplication 运行监听器(SpringApplicationRunListeners)

从`SpringApplication`类的`run()`方法中，我们可以看到下面代码`getRunListeners()`方法同样利用 Spring 工厂加载机制,读取 SpringApplicationRunListener 对象集合,并且封装到组合类 SpringApplicationRunListeners对象中并启动运行监听器。

```
        SpringApplicationRunListeners listeners = getRunListeners(args);
        listeners.starting();
```

第七步：监听SpringBoot/Spring事件

Spring Boot 通过 SpringApplicationRunListener 的实现类 EventPublishingRunListener 利用 Spring Framework 事件 API ,广播 Spring Boot 事件。

第八步：创建SpringBoot的应用上下文

```
context = createApplicationContext();
```

从`SpringApplication`类的`run()`方法中,我们可看到`createApplicationContext()`根据第二步推断的SpringBoot应用类型创建相应的上下文。

```
    protected ConfigurableApplicationContext createApplicationContext() {
        Class<?> contextClass = this.applicationContextClass;
        if (contextClass == null) {
            try {
                switch (this.webApplicationType) {
                case SERVLET:
                    contextClass = Class.forName(DEFAULT_SERVLET_WEB_CONTEXT_CLASS);
                    break;
                case REACTIVE:
                    contextClass = Class.forName(DEFAULT_REACTIVE_WEB_CONTEXT_CLASS);
                    break;
                default:
                    contextClass = Class.forName(DEFAULT_CONTEXT_CLASS);
                }
            }
            catch (ClassNotFoundException ex) {
                throw new IllegalStateException(
                        "Unable create a default ApplicationContext, "
                                + "please specify an ApplicationContextClass",
                        ex);
            }
        }
        return (ConfigurableApplicationContext) BeanUtils.instantiateClass(contextClass);
    }
```

根据推断的SpringBoot应用类型创建下面三种之一的上下文

```
    public static final String DEFAULT_CONTEXT_CLASS = "org.springframework.context."
            + "annotation.AnnotationConfigApplicationContext";


    public static final String DEFAULT_SERVLET_WEB_CONTEXT_CLASS = "org.springframework.boot."
            + "web.servlet.context.AnnotationConfigServletWebServerApplicationContext";


    public static final String DEFAULT_REACTIVE_WEB_CONTEXT_CLASS = "org.springframework."
            + "boot.web.reactive.context.AnnotationConfigReactiveWebServerApplicationContext";
```

第九步：创建 Environment

```
ConfigurableEnvironment environment = prepareEnvironment(listeners,
                    applicationArguments);
            configureIgnoreBeanInfo(environment);
```

从`SpringApplication`类的`run()`方法中,我们可看到`prepareEnvironment()`根据第二步推断的SpringBoot应用类型创建相应的上下文。创建不同的`Environment` 。从下面可以看到他们分别是`StandardServletEnvironment`，`StandardReactiveWebEnvironment`，`StandardEnvironment`

```
    private ConfigurableEnvironment getOrCreateEnvironment() {
        if (this.environment != null) {
            return this.environment;
        }
        switch (this.webApplicationType) {
        case SERVLET:
            return new StandardServletEnvironment();
        case REACTIVE:
            return new StandardReactiveWebEnvironment();
        default:
            return new StandardEnvironment();
        }
    }
```

结论

1. 标注`@SpringBootApplication`注解的类会成为Spring的Bean
2. SpringBoot默认会装配启动类路径的所有包下可装配的Bean；也就是说如果你把SpringBoot启动类放在一个单独的包中，则SpringBoot不会装配到你的其他Bean。这时候你就要使用`@SpringBootApplication`的`scanBasePackages()`方法进行另行配置。
3. SpringBoot在启动时会自动加载Classpth路径下的`META-INF/spring.factories`文件，所以我们可以将需要自动配置的Bean写入这个文件。同样SpringBoot也会扫描Jar包中的`META-INF/spring.factories`文件；例如当导入`spring-boot-starter`起步依赖的时候，并且启用了自动装配注解`@EnableAutoConfiguration`，就将会替我们自动装配如如下类；如果满足条件的话。

```
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,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.rest.RestClientAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.core.ReactorCoreAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityRequestMatcherProviderAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration,\

org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration
```