

springboot自动配置（根据启动流程分析）

@SpringBootApplication内部结构：

1. @SpringBootApplication->@SpringBootConfiguration讲解：
2. @SpringBootApplication->@EnableAutoConfiguration讲解：
 - 2.1 @SpringBootApplication->@EnableAutoConfiguration->@AutoConfigurationPackage讲解（含实现细节总结，2.1.1是细节）：
 - 2.1.1 @SpringBootApplication->@EnableAutoConfiguration->@AutoConfigurationPackage->@Import(AutoConfigurationPackages.Registrar.class)讲解：registerBeanDefinition()??? 将什么组件注册???
 - 2.2 @SpringBootApplication->@EnableAutoConfiguration->@Import(AutoConfigurationImportSelector.class)讲解（含实现细节总结，2.2.1、2.2.2是细节）：
 - 2.2.1 @SpringBootApplication->@EnableAutoConfiguration->@Import(AutoConfigurationImportSelector.class)->AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader)讲解：
 - 2.2.2 @SpringBootApplication->@EnableAutoConfiguration->@Import(AutoConfigurationImportSelector.class)->getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata)讲解：
3. @SpringBootApplication->@ComponentScan讲解：

自定义starter

自定义starter介绍：

自定义starter搭建：

自定义starter使用：

springboot实例化SpringApplication

SpringBootApplication->实例化SpringApplication对象：

SpringBootApplication->实例化SpringApplication对象->WebApplicationType.deduceFromClasspath()：

SpringBootApplication->实例化SpringApplication对象->getSpringFactoriesInstances(ApplicationContextInitializer.class)：

SpringBootApplication执行run方法：

@EnableAutoConfiguration讲解：

dependencyManagement使用简介

springboot源码总结：

springboot源码分两部分：

@SpringBootApplication讲解：

SpringApplication.run(SpringBootDemoApplication.class, args)讲解：

springboot自动配置（根据启动流程分析）

```
1 | @SpringBootApplication//能够扫描Spring组件并自动配置Spring boot
2 | public class Springboot01DemoApplication {
3 |     public static void main(String[] args) {
4 |         SpringApplication.run(Springboot01DemoApplication.class, args);
5 |     }
6 | }
```

@SpringBootApplication内部结构:

```
1 @Target(ElementType.TYPE)    //注解的适用范围,Type表示注解可以描述在类、接口、注解或枚举中
2 @Retention(RetentionPolicy.RUNTIME) ///表示注解的生命周期, Runtime运行时
3 @Documented    ///表示注解可以记录在javadoc中
4 @Inherited    //表示可以被子类继承该注解 (以上这些了解即可)
5
6 @SpringBootConfiguration // 标明该类为配置类
7 @EnableAutoConfiguration // 启动自动配置功能
8 @ComponentScan(excludeFilters = { // 包扫描器 <context:component-scan base-
    package="com.xxx.xxx"/>
9     @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
10    @Filter(type = FilterType.CUSTOM, classes =
    AutoConfigurationExcludeFilter.class) })
11 public @interface SpringBootApplication {
12     @AliasFor(annotation = EnableAutoConfiguration.class)
13     Class<?>[] exclude() default {};
14
15     @AliasFor(annotation = EnableAutoConfiguration.class)
16     String[] excludeName() default {};
17
18     @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
19     String[] scanBasePackages() default {};
20
21     @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
22     Class<?>[] scanBasePackageClasses() default {};
23
24 }
```

1. @SpringBootApplication->@SpringBootConfiguration讲解:

该注解只是对@Configuration的封装, 等同于@Configuration。

```
1 @Configuration //配置类
2 public @interface SpringBootConfiguration {
3 }
```

2. @SpringBootApplication->@EnableAutoConfiguration讲解:

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5
6 @AutoConfigurationPackage    //2.1讲解
7 @Import(AutoConfigurationImportSelector.class) //2.2讲解
8 public @interface EnableAutoConfiguration {
9
10     String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
11 }
```

```

12     Class<?>[] exclude() default {};
13
14     String[] excludeName() default {};
15
16 }

```

2.1@SpringBootApplication->@EnableAutoConfiguration-

>@AutoConfigurationPackage讲解（含实现细节总结，2.1.1是细节）：

作用：会把@springBootApplication注解标注的类所在包名拿到，并且对该包及其子包进行扫描，将组件添加到容器中，这就是为什么把Springboot01DemoApplication放在外层的原因，因为它要扫描dao、service等包。

实现细节总结：将@springBootApplication注解标注的类所在包名拿到，并且对该包及其子包进行扫描，将组件添加到容器中。

```

1
2 //spring框架的底层注解，它的作用就是给容器中导入某个组件类，
3 //例如@Import(AutoConfigurationPackages.Registrar.class)，它就是将Registrar这个组件类
  导入到容器中
4 @Import(AutoConfigurationPackages.Registrar.class) // 默认将主配置类
  (@SpringBootApplication)所在的包及其子包里面的所有组件扫描到Spring容器中
5 public @interface AutoConfigurationPackage {
6 }

```

2.1.1@SpringBootApplication->@EnableAutoConfiguration-

>@AutoConfigurationPackage->@Import(AutoConfigurationPackages.Registrar.class)

讲解：registerBeanDefinition()?? 将什么组件注册??

```

1 public abstract class AutoConfigurationPackages {
2
3     static class Registrar implements ImportBeanDefinitionRegistrar,
  DeterminableImports {
4
5         // 获取的是项目主程序启动类所在的目录
6         //metadata:注解标注的元数据信息
7         @Override
8         public void registerBeanDefinitions(AnnotationMetadata metadata,
  BeanDefinitionRegistry registry) {
9             //默认将会扫描@SpringBootApplication标注的主配置类所在的包及其子包下所有组
  件
10             register(registry, new PackageImport(metadata).getPackageName());
11         }
12     }
13
14     //register(registry, new PackageImport(metadata).getPackageName())讲解
15     public static void register(BeanDefinitionRegistry registry, String...
  packageNames) {
16         GenericBeanDefinition beanDefinition = new GenericBeanDefinition();
17
18         beanDefinition.setBeanClass(AutoConfigurationPackages.BasePackages.class);
19
20         beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue(0,
  packageNames);
21         beanDefinition.setRole(2);
22         registry.registerBeanDefinition(BEAN, beanDefinition);
23     }
24 }

```

2.2@SpringBootApplication->@EnableAutoConfiguration-

>@Import(AutoConfigurationImportSelector.class)讲解（含实现细节总结，2.2.1、2.2.2是细节）：

作用：可以帮助springboot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器(ApplicationContext)中

实现细节总结：见下面

```
1 public class AutoConfigurationImportSelector
2     implements DeferredImportSelector, BeanClassLoaderAware,
   ResourceLoaderAware,
3     BeanFactoryAware, EnvironmentAware, Ordered {
4
5     // selectImports ()：这个方法告诉springboot都需要导入那些组件
6     @Override
7     public String[] selectImports(AnnotationMetadata annotationMetadata) {
8         //判断 enableautoconfiguration注解有没有开启，默认开启（是否进行自动装
   配），没有开启就不运行下面方法
9         if (!isEnabled(annotationMetadata)) {
10             return NO_IMPORTS; //空数组
11         }
12         //加载配置文件META-INF/spring-autoconfigure-metadata.properties，从中获
   取所有支持自动配置类的条件
13         //作用：SpringBoot使用一个Annotation的处理器来收集一些自动装配的条件，那么
   这些条件可以在META-INF/spring-autoconfigure-metadata.properties进行配置。
14         // SpringBoot会将收集好的@Configuration进行一次过滤进而剔除不满足条件的配
   置类
15         // 自动配置的全名.条件=值
16         AutoConfigurationMetadata autoConfigurationMetadata =
   AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
17
18         AutoConfigurationEntry autoConfigurationEntry =
   getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata);
19         return
   StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
20     }
21
22 }
```

2.2.1@SpringBootApplication->@EnableAutoConfiguration-

>@Import(AutoConfigurationImportSelector.class)-

>AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader)讲解：

```
1 //该方法properties文件如下图所示，主要是将properties文件的内容遍历封装成
   AutoConfigurationMetadata对象返回
2 public final class AutoConfigurationMetadataLoader {
3     protected static final String PATH = "META-INF/" + "spring-autoconfigure-
   metadata.properties"
4
5     static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader, String
   path) {
6         try {
7             //1.读取spring-boot-autoconfigure.jar包中spring-autoconfigure-
   metadata.properties的信息生成urls枚举对象
8             // 获得 PATH 对应的 URL 们
9             Enumeration<URL> urls = (classLoader != null)
```

```

10     classLoader.getResources(path) : ClassLoader.getSystemResources(path);
11     // 遍历 URL 数组，读取到 properties 中
12     Properties properties = new Properties();
13
14     //2.解析urls枚举对象中的信息封装成properties对象并加载
15     while (urls.hasMoreElements()) {
16         properties.putAll(PropertiesLoaderUtils.loadProperties(new
17             UrlResource(urls.nextElement())));
18     }
19     // 将 properties 转换成 PropertiesAutoConfigurationMetadata 对象
20     // 根据封装好的properties对象生成AutoConfigurationMetadata对象返回
21     return loadMetadata(properties);
22 } catch (IOException ex) {
23     throw new IllegalArgumentException("Unable to load @ConditionalOnClass
24     location [" + path + "]", ex);
25 }

```

The screenshot shows the 'spring-autoconfigure-metadata.properties' file in an IDE. The file lists various Spring Boot auto-configuration classes and their associated conditional annotations. A red arrow points to the entry 'org.springframework.boot.autoconfigure.web.reactive.HandlerAutoConfiguration:ConditionalOnWebApplication=REACTIVE', with a note '由类全名+条件组成' (composed of class full name + conditions).

2.2.2@SpringBootApplication->@EnableAutoConfiguration-

>@Import(AutoConfigurationImportSelector.class)-

>getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata)讲解:

```

1  protected AutoConfigurationEntry
2  getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigurationMetadata,
3  AnnotationMetadata annotationMetadata) {
4      // 1. 判断是否开启注解。如未开启，返回空串
5      if (!isEnabled(annotationMetadata)) {
6          return EMPTY_ENTRY;
7      }
8      // 2. 获得注解的属性
9      AnnotationAttributes attributes = getAttributes(annotationMetadata);
10
11     // 3. getCandidateConfigurations()用来获取默认支持的自动配置类名列表
12     // spring Boot在启动的时候，使用内部工具类SpringFactoriesLoader，查找
13     classpath上所有jar包中的META-INF/spring.factories，
14     // 找出其中key为
15     org.springframework.boot.autoconfigure.EnableAutoConfiguration的属性定义的工厂类名
16     称，
17     // 将这些值作为自动配置类导入到容器中，自动配置类就生效了
18     List<String> configurations =
19     getCandidateConfigurations(annotationMetadata, attributes);
20
21 }

```

```

15
16 // 3.1 //去除重复的配置类，若我们自己写的starter 可能存在重复的
17 configurations = removeDuplicates(configurations);
18 // 4. 如果项目中某些自动配置类，我们不希望其自动配置，我们可以通过
EnableAutoConfiguration的exclude或excludeName属性进行配置，
19 // 或者也可以在配置文件里通过配置项“spring.autoconfigure.exclude”进行配置。
20 //找到不希望自动配置的配置类（根据EnableAutoConfiguration注解的一个exclusions
属性）
21 Set<String> exclusions = getExclusions(annotationMetadata, attributes);
22 // 4.1 校验排除类（exclusions指定的类必须是自动配置类，否则抛出异常）
23 checkExcludedClasses(configurations, exclusions);
24 // 4.2 从 configurations 中，移除所有不希望自动配置的配置类
25 configurations.removeAll(exclusions);
26
27 // 5. 对所有候选的自动配置类进行筛选，根据项目pom.xml文件中加入的依赖文件筛选出
最终符合当前项目运行环境对应的自动配置类
28
29 //@ConditionalOnClass : 某个class位于类路径上，才会实例化这个Bean。
30 //@ConditionalOnMissingClass : classpath中不存在该类时起效
31 //@ConditionalOnBean : DI容器中存在该类型Bean时起效
32 //@ConditionalOnMissingBean : DI容器中不存在该类型Bean时起效
33 //@ConditionalOnSingleCandidate : DI容器中该类型Bean只有一个或@Primary的只有
一个时起效
34 //@ConditionalOnExpression : SpEL表达式结果为true时
35 //@ConditionalOnProperty : 参数设置或者值一致时起效
36 //@ConditionalOnResource : 指定的文件存在时起效
37 //@ConditionalOnJndi : 指定的JNDI存在时起效
38 //@ConditionalOnJava : 指定的Java版本存在时起效
39 //@ConditionalOnWebApplication : Web应用环境下起效
40 //@ConditionalOnNotWebApplication : 非Web应用环境下起效
41
42 //总结一下判断是否要加载某个类的两种方式：
43 //根据spring-autoconfigure-metadata.properties进行判断。
44 //要判断@Conditional是否满足
45 // 如@ConditionalOnClass({ SqlSessionFactory.class,
SqlSessionFactoryBean.class })表示需要在类路径中存在SqlSessionFactory.class、
SqlSessionFactoryBean.class这两个类才能完成自动注册。
46 configurations = filter(configurations, autoConfigurationMetadata);
47
48
49 // 6. 将自动配置导入事件通知监听器
50 //当AutoConfigurationImportSelector过滤完成后会自动加载类路径下Jar包中META-
INF/spring.factories文件中 AutoConfigurationImportListener的实现类，
51 // 并触发fireAutoConfigurationImportEvents事件。
52 fireAutoConfigurationImportEvents(configurations, exclusions);
53 // 7. 创建 AutoConfigurationEntry 对象
54 return new AutoConfigurationEntry(configurations, exclusions);
55 }
56

```

该方法有两个重要部分：

57 1.getCandidateConfigurations(annotationMetadata, attributes):
58 getCandidateConfigurations()用来获取默认支持的自动配置类名列表（spring Boot在启动
59 的时候，使用内部工具类SpringFactoriesLoader，查找classpath上所有jar包中的META-
INF/spring.factories，找出其中key为
org.springframework.boot.autoconfigure.EnableAutoConfiguration的属性定义的工厂类名
称，将这些值作为自动配置类导入到容器中，自动配置类就生效了）spring.factories见下图。

60 2.filter(configurations, autoConfigurationMetadata):

61 条件类型：

62 @ConditionalOnClass : 某个class位于类路径上，才会实例化这个Bean。
63 @ConditionalOnMissingClass : classpath中不存在该类时起效


```

65 @ConditionalOnBean : DI容器中存在该类型Bean时起效
66 @ConditionalOnMissingBean : DI容器中不存在该类型Bean时起效
67 @ConditionalOnSingleCandidate : DI容器中该类型Bean只有一个或@Primary的只有一个时起效
68 @ConditionalOnExpression : SpEL表达式结果为true时
69 @ConditionalOnProperty : 参数设置或者值一致时起效
70 @ConditionalOnResource : 指定的文件存在时起效
71 @ConditionalOnJndi : 指定的JNDI存在时起效
72 @ConditionalOnJava : 指定的Java版本存在时起效
73 @ConditionalOnWebApplication : Web应用环境下起效
74 @ConditionalOnNotWebApplication : 非Web应用环境下起效
75 对自动配置类名列表进行条件判断, 比如: 1.1、1.2、1.3图,
76 要判断org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration类需满足
77 @ConditionalOnClass(DSLContext.class)才能完成自动注册。//里面也可能是数组

```

如下图可以看到所有需要配置的全路径都在文件中, 每行一个配置, 多个类名逗号分隔, 而\表示忽略换行

```

# 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,\
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.jooq.JooqAutoConfiguration=\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,ConditionalOnClass=javax.sql.DataSource,org.springframework
org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration=
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration=
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration.AutoConfigureBefore=org.springframework
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration.ConditionalOnClass=org.spring
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration.AutoConfigureAfter=
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration.AutoConfigureAfter=org.springframework
org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration.ConditionalOnClass=org.springframework.boot.autoconfigure
org.springframework.boot.autoconfigure.session.JdbcSessionConfiguration=
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration.ConditionalOnClass=org.jooq.DSLContext
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration.ConditionalOnClass=org.springframework.boot.autoconfigure
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration.AutoConfigureAfter=
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseConfigurerAdapterConfiguration.ConditionalOnClass=org.spring
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure
org.springframework.boot.autoconfigure.data.redis.LettuceConnectionConfiguration=
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration.AutoConfigureAfter=org.springframework
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration.AutoConfigureAfter=org.spring
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration=
org.springframework.boot.autoconfigure.transaction.jta.AtomikosJtaConfiguration=
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration=
org.springframework.boot.autoconfigure.kafka.KafkaStreamsAnnotationDrivenConfiguration.ConditionalOnClass=org.apache.kafka
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration=
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration=
org.springframework.boot.autoconfigure.data.couchbase.SpringBootCouchbaseReactiveDataConfiguration.ConditionalOnBean=org

```

```
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\norg.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\norg.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\norg.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\norg.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\norg.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\norg.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\norg.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\norg.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\norg.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\norg.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\norg.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\norg.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\norg.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\norg.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\norg.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\norg.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\norg.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\n
```

1.2

```
/* @author Michael Simons\n * @author Dmytro Nosan\n * @since 1.3.0\n */\n@Configuration\n@ConditionalOnClass(DSLContext.class)\n@ConditionalOnBean(DataSource.class)\n@AutoConfigureAfter({ DataSourceAutoConfiguration.class,\n    TransactionAutoConfiguration.class })\npublic class JooqAutoConfiguration {\n\n    @Bean\n    @ConditionalOnMissingBean\n    public DataSourceConnectionProvider dataSourceConnectionProvider(\n        DataSource dataSource) {\n        return new DataSourceConnectionProvider(\n            new TransactionAwareDataSourceProxy(dataSource));\n    }\n\n    @Bean\n    @ConditionalOnBean(PlatformTransactionManager.class)\n    public SpringTransactionProvider transactionProvider(\n        PlatformTransactionManager txManager) {\n        return new SpringTransactionProvider(txManager);\n    }\n}
```

1.3

3. @SpringBootApplication->@ComponentScan讲解:

包扫描器，相当于ssm框架的<context:component-scan base-package="com.xxx.xxx"/>。扫描的包的路径由 @EnableAutoConfiguration-@AutoConfigurationPackage 得到，该注解进行扫描。

自定义starter

自定义starter介绍:

实际相当于导入autoconfigure依赖，该项目就会自动加载引入的功能jar包的配置，但是后面会有相应的约束来决定是否需要配置。

SpringBoot starter机制

SpringBoot由众多Starter组成（一系列的自动化配置的starter插件），SpringBoot之所以流行，也是因为starter。

starter是SpringBoot非常重要的一部分，可以理解为一个可拔插式的插件，正是这些starter使得使用某个功能的开发者不需要关注各种依赖库的处理，不需要具体的配置信息，由Spring Boot自动通过classpath路径下的类发现需要的Bean，并织入相应的Bean。

例如，你想使用Reids插件，那么可以使用spring-boot-starter-redis；如果想使用MongoDB，可以使用spring-boot-starter-data-mongodb

为什么要自定义starter

开发过程中，经常会有一些独立于业务之外的配置模块。如果我们将这些可独立于业务代码之外的功能配置模块封装成一个个starter，复用的时候只需要将其在pom中引用依赖即可，SpringBoot为我们完成自动装配

自定义starter的命名规则

SpringBoot提供的starter以 `spring-boot-starter-xxx` 的方式命名的。官方建议自定义的starter使用 `xxx-spring-boot-starter` 命名规则。以区分SpringBoot生态提供的starter

整个过程分为两部分：

- 自定义starter
- 使用starter

自定义starter搭建:

导入依赖：

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-autoconfigure</artifactId>
5     <version>2.2.2.RELEASE</version>
6   </dependency>
7 </dependencies>
```

编写javaBean：

```
1 @EnableConfigurationProperties(SimpleBean.class)
2 @ConfigurationProperties(prefix = "simplebean")
3 public class SimpleBean {
4     private int id;
5     private String name;
6     public int getId() {
7         return id;
8     }
9     public void setId(int id) {
```

```

10         this.id = id;
11     }
12     public String getName() {
13         return name;
14     }
15     public void setName(String name) {
16         this.name = name;
17     }
18     @Override
19     public String toString() {
20         return "SimpleBean{" +
21             "id=" + id +
22             ", name='" + name + '\'' +
23             '}';
24     }
25 }

```

编写配置类MyAutoConfiguration:

```

1  @Configuration
2  @ConditionalOnClass
3  public class MyAutoConfiguration {
4      static {
5          System.out.println("MyAutoConfiguration init....");
6      }
7      @Bean
8      public SimpleBean simpleBean() {
9          return new SimpleBean();
10     }
11
12 }

```

resources下创建/META-INF/spring.factories:

注意: META-INF是自己手动创建的目录, spring.factories也是手动创建的文件,在该文件中配置自己的自动配置类

```

1  org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2  com.lagou.config.MyAutoConfiguration

```

自定义starter使用:

导入自定义starter的依赖:

```

1  <dependency>
2      <groupId>com.lagou</groupId>
3      <artifactId>zdy-spring-boot-starter</artifactId>
4      <version>1.0-SNAPSHOT</version>
5  </dependency>

```

全局配置文件配置属性值:

```

1 simplebean.id=1
2 simplebean.name=自定义starter

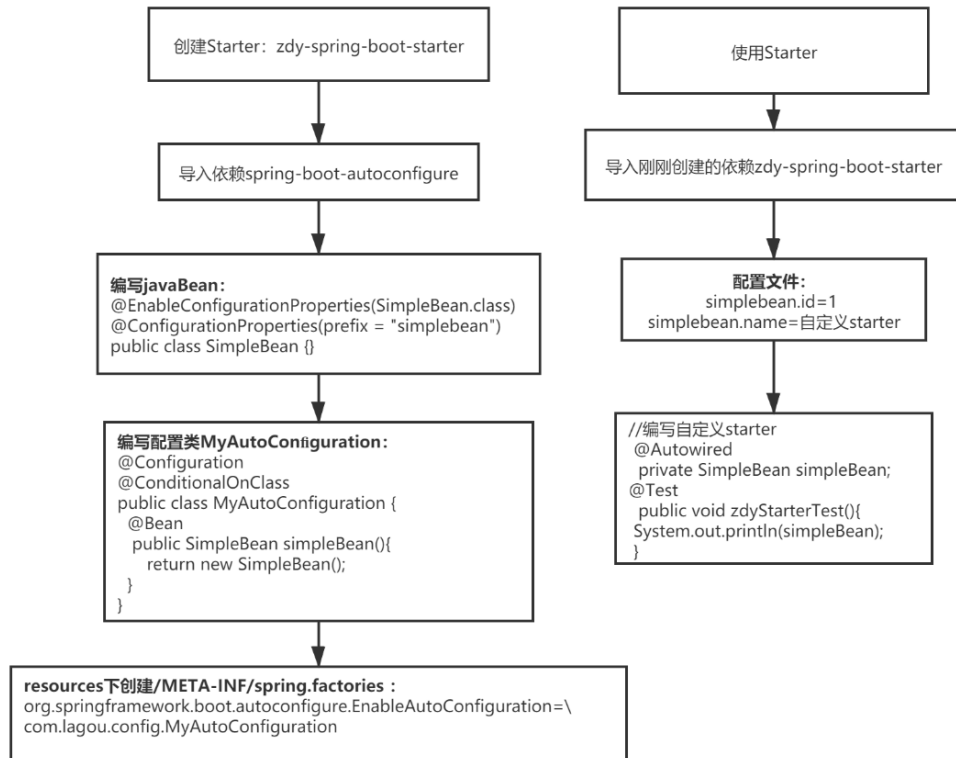
```

编写测试方法：

```

1 @Autowired
2 private SimpleBean simpleBean;
3
4 @Test
5 public void zdyStarterTest(){
6     System.out.println(simpleBean);
7 }

```



springboot实例化SpringApplication

```

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

```

```

1 public static ConfigurableApplicationContext run(Class<?>[] primarySources,
2 String[] args) {
3     //SpringApplication的启动由两部分组成:
4     //1. 实例化SpringApplication对象
5     //2. run(args): 调用run方法
6     return new SpringApplication(primarySources).run(args);
7 }

```

SpringBootApplication->实例化SpringApplication对象:

```

1 public SpringApplication(ResourceLoader resourceLoader, Class<?>...
2 primarySources) {
3     ...
4     //1.项目启动类 SpringbootDemoApplication.class设置为属性存储起来
5     this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
6
7     //2.设置应用类型是SERVLET应用（Spring 5之前的传统MVC应用）还是REACTIVE应用（Spring
8     5开始出现的WebFlux交互式应用）
9     this.webApplicationType = WebApplicationType.deduceFromClasspath();
10
11    //3.设置初始化器(Initializer),最后会调用这些初始化器
12    //所谓的初始化器就是org.springframework.context.ApplicationContextInitializer的
13    实现类,在Spring上下文被刷新之前进行初始化的操作
14    setInitializers((Collection)
15    getSpringFactoriesInstances(ApplicationContextInitializer.class));
16
17    //4.设置监听器(Listener)
18    setListeners((Collection)
19    getSpringFactoriesInstances(ApplicationListener.class));
20
21    //5.初始化 mainApplicationClass 属性:用于推断并设置项目main()方法启动的主程序启动
22    类
23    this.mainApplicationClass = deduceMainApplicationClass();
24 }

```

SpringBootApplication->实例化SpringApplication对象- >WebApplicationType.deduceFromClasspath():

```

1 static WebApplicationType deduceFromClasspath() {
2     // WebApplicationType.REACTIVE 类型 通过类加载器判断REACTIVE相关的Class是否
3     存在
4     if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, null) // 判断REACTIVE相关
5     的Class是否存在
6         && !ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS, null)
7         && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, null)) {
8         return WebApplicationType.REACTIVE;
9     }
10    // WebApplicationType.NONE 类型
11    for (String className : SERVLET_INDICATOR_CLASSES) {
12        if (!ClassUtils.isPresent(className, null)) { // 不存在 Servlet 的类
13            return WebApplicationType.NONE;
14        }
15    }
16 }

```

```

11         return WebApplicationType.NONE;
12     }
13 }
14 // WebApplicationType.SERVLET 类型。可以这样的判断的原因是，引入 Spring MVC
    时，是内嵌的 Web 应用，会引入 Servlet 类。
15     return WebApplicationType.SERVLET;
16 }

```

*SpringBootApplication->实例化SpringApplication对象->
getSpringFactoriesInstances(ApplicationContextInitializer.class) :*

```

1 private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
2               Class<?>[] parameterTypes, Object... args) {
3     ClassLoader classLoader = getClassLoader();
4
5     // 加载指定类型对应的，在 `META-INF/spring.factories` 里的类名的数组
6     Set<String> names = new LinkedHashSet<>
7     (SpringFactoriesLoader.loadFactoryNames(type, classLoader));
8
9     //org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitial
10    izer,\
11    //org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingL
12    istener
13     // 根据names来进行实例化
14     List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
15     classLoader, args, names);
16     // 对实例进行排序
17     AnnotationAwareOrderComparator.sort(instances);
18     return instances;
19 }
20 该方法就是通过ApplicationContextInitializer的类路径找到spring.factories对应的指定类
    型，如下图

```

```

# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener

# Auto Configuration Import Filters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
org.springframework.boot.autoconfigure.condition.OnClassCondition,\
org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition

# 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,\

```

SpringBootApplication执行run方法：

```
1 public ConfigurableApplicationContext run(String... args) {
2     // (1) 获取并启动监听器
3     SpringApplicationRunListeners listeners = getRunListeners(args);
4     listeners.starting();
5     try {
6         // 创建 ApplicationArguments 对象 初始化默认应用参数类
7         // args是启动Spring应用的命令行参数，该参数可以在Spring应用中被访问。
8         // 如: --server.port=9000
9         ApplicationArguments applicationArguments = new
10            DefaultApplicationArguments(args);
11
12         // (2) 项目运行环境Environment的预配置
13         // 创建并配置当前SpringBoot应用将要使用的Environment
14         // 并遍历调用所有的SpringApplicationRunListener的environmentPrepared()
15         // 方法
16         ConfigurableEnvironment environment = prepareEnvironment(listeners,
17            applicationArguments);
18
19         configureIgnoreBeanInfo(environment);
20         // 准备Banner打印机 - 就是启动Spring Boot的时候打印在console上的ASCII艺
21         // 术字体
22         Banner printedBanner = printBanner(environment);
23
24         // (3) 创建Spring容器
25         context = createApplicationContext();
26         // 获得异常报告器 SpringBootExceptionHandler 数组
27         // 这一步的逻辑和实例化初始化和监听器的一样，
28         // 都是通过调用 getSpringFactoriesInstances 方法来获取配置的异常类名称并
29         // 实例化所有的异常处理类。
30         exceptionReporters = getSpringFactoriesInstances(
31            SpringBootExceptionHandler.class,
32            new Class[] { ConfigurableApplicationContext.class },
33            context);
34
35         // (4) Spring容器前置处理
36         // 这一步主要是在容器刷新之前的准备动作。包含一个非常关键的操作：将启动类注
37         // 入容器，为后续开启自动化配置奠定基础。
38         prepareContext(context, environment, listeners, applicationArguments,
39            printedBanner);
40
41         // (5)：刷新容器
42         refreshContext(context);
43
44         // (6)：Spring容器后置处理
45         // 扩展接口，设计模式中的模板方法，默认为空实现。
46         // 如果有自定义需求，可以重写该方法。比如打印一些启动结束log，或者一些其它
47         // 后置处理
48         afterRefresh(context, applicationArguments);
49         // 停止 Stopwatch 统计时长
50         stopwatch.stop();
51         // 打印 Spring Boot 启动的时长日志。
52         if (this.logStartupInfo) {
```



```

44         new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopWatch);
45     }
46     // (7) 发出结束执行的事件通知
47     listeners.started(context);
48
49     // (8): 执行Runners
50     //用于调用项目中自定义的执行器XxxRunner类, 使得在项目启动完成后立即执行一
些特定程序
51     //Runner 运行器用于在服务启动时进行一些业务初始化操作, 这些操作只在服务启动
后执行一次。
52     //Spring Boot提供了ApplicationRunner和CommandLineRunner两种服务接口
53     callRunners(context, applicationArguments);
54     } catch (Throwable ex) {
55         // 如果发生异常, 则进行处理, 并抛出 IllegalStateException 异常
56         handleRunFailure(context, ex, exceptionReporters, listeners);
57         throw new IllegalStateException(ex);
58     }
59
60     // (9)发布应用上下文就绪事件
61     //表示在前面一切初始化启动都没有问题的情况下, 使用运行监听器
SpringApplicationRunListener持续运行配置好的应用上下文ApplicationContext,
62     // 这样整个Spring Boot项目就正式启动完成了。
63     try {
64         listeners.running(context);
65     } catch (Throwable ex) {
66         // 如果发生异常, 则进行处理, 并抛出 IllegalStateException 异常
67         handleRunFailure(context, ex, exceptionReporters, null);
68         throw new IllegalStateException(ex);
69     }
70     //返回容器
71     return context;
72 }
73

```

共九步:

1. 获取并启动监听器:

依然使用getSpringFactoriesInstances()方法来获取实例, 从META-INF/spring.factories中读取Key为org.springframework.boot.SpringApplicationRunListener的Values使用SpringApplicationRunListeners封装对象, 调用starting()启动监听器。

2. 项目运行环境Environment的预配置:

加载外部化配置资源到environment, 包括命令行参数、servletConfigInitParams、servletContextInitParams、systemProperties、systemEnvironment、random、application.yml(.yaml/.xml/.properties)等; 初始化日志系统。

3. 创建Spring容器:

先判断有没有指定的实现类(使用Class<? extends ConfigurableApplicationContext>, 如果为null抛错), 获取获得 ApplicationContext 类型, 反射创建ApplicationContext 对象

再使用getSpringFactoriesInstances()获得异常报告器。

4. Spring容器前置处理:

这块会对整个上下文进行一个预处理, 比如触发监听器的响应事件、加载资源、设置上下文环境等等

包含一个非常关键的操作: 将启动类注入容器, 为后续开启自动化配置奠定基础。

5. 刷新容器:

开启(刷新) Spring 容器, 通过refresh方法对整个IoC容器的初始化(包括Bean资源的定位、解析、注册等等)

94 向JVM运行时注册一个关机钩子,在JVM关机时关闭这个上下文,除非它当时已经关闭(如果JVM
95 关机时,容器也要关闭)

96 6.Spring容器后置处理:
97 该方法没有实现,可以根据需要做一些定制化的操作。

98

99 7.发出结束执行的事件通知:
100 执行所有SpringApplicationRunListener实现的started方法。

101

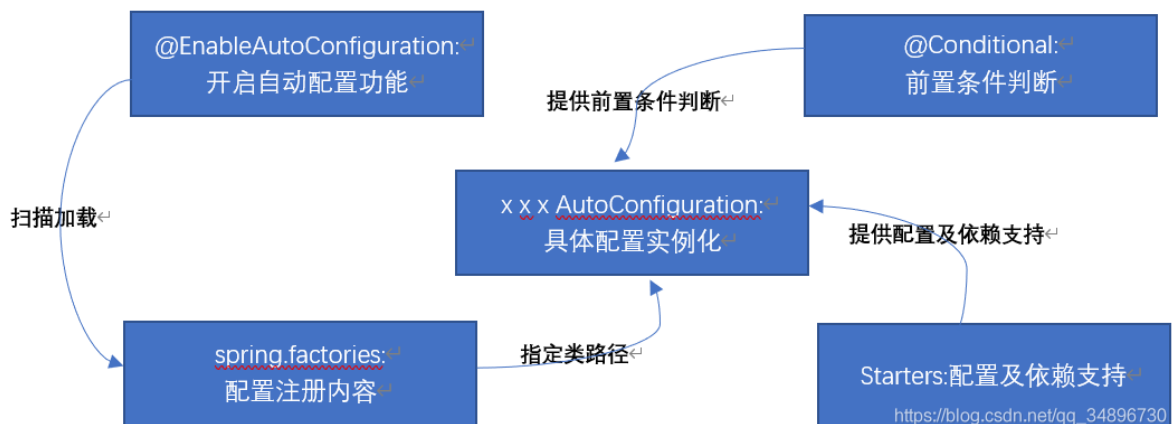
102 8.执行Runners:
103 调用ApplicationRunner、CommandLineRunner实现类的run方法。

104

105 9.发布应用上下文就绪事件:
106 触发所有 SpringApplicationRunListener 监听器的 running 事件方法。(表示在前面
一切初始化启动都没有问题的情况下,使用运行监听器SpringApplicationRunListener持续运行配
置好的应用上下文ApplicationContext,这样整个Spring Boot项目就正式启动完成了)

@EnableAutoConfiguration讲解:

使用Spring Boot时,我们只需引入对应的Starters, Spring Boot启动时便会自动加载相关依赖,配置相应的初始化参数,以最快捷、简单的形式对第三方软件进行集成,这便是Spring Boot的自动配置功能。我们先从整体上看一下Spring Boot实现该运作机制涉及的核心部分,如下图所示。



上图描述了Spring Boot自动配置功能运作过程中涉及的几个核心功能及其相互之间的关系包括@EnableAutoConfiguration、spring.factories、各组件对应的AutoConfiguration类、@Conditional注解以及各种Starters。

可以用一句话来描述整个过程：Spring Boot通过@EnableAutoConfiguration注解开启自动配置，加载spring.factories中注册的各种AutoConfiguration类，当某个AutoConfiguration类满足其注解@Conditional指定的生效条件（Starters提供的依赖、配置或Spring容器中是否存在某个Bean等）时，实例化该AutoConfiguration类中定义的Bean（组件等），并注入Spring容器，就可以完成依赖框架的自动配置。

概念:

- **@EnableAutoConfiguration**: 该注解由组合注解@SpringBootApplication引入,完成自动配置开启,扫描各个jar包下的spring.factories文件,并加载文件中注册的AutoConfiguration类等。
- **spring.factories**: 配置文件,位于jar包的META-INF目录下,按照指定格式注册了自动配置的AutoConfiguration类。spring.factories也可以包含其他类型待注册的类。该配置文件不仅存在于Spring Boot项目中,也可以存在于自定义的自动配置(或Starter)项目中。
- **AutoConfiguration类**: 自动配置类,代表了Spring Boot中一类以XXAutoConfiguration命名的自动配置类。其中定义了三方组件集成Spring所需初始化的Bean和条件。
- **@Conditional**: 条件注解及其衍生注解,在AutoConfiguration类上使用,当满足该条件注解时才会实例化AutoConfiguration类。

- Starters：三方组件的依赖及配置，Spring Boot已经预置的组件。Spring Boot默认的Starters项目往往只包含了一个pom依赖的项目。如果是自定义的starter，该项目还需包含spring.factories文件、AutoConfiguration类和其他配置类。

总结：

```
1 @SpringBootApplication
2     |- @SpringBootConfiguration
3         |- @Configuration //通过javaConfig的方式添加组件到IOC容器中
4     |- @EnableAutoConfiguration
5         |- @AutoConfigurationPackage //自动配置包，与@ComponentScan扫描到的添加到IOC
6         |- @Import(AutoConfigurationImportSelector.class)//到META-INF/spring.factories
           中定义的bean添加到IOC容器中
7     |- @ComponentScan //包扫描
```

dependencyManagement使用简介

在dependencyManagement元素中声明所依赖的jar包的版本号等信息，那么所有子项目再次引入此依赖jar包时则无需显式的列出版本号。Maven会沿着父子层级向上寻找拥有dependencyManagement元素的项目，然后使用它指定的版本号。

举例：

在父项目的POM.xml中配置：

```
1 <dependencyManagement>
2     <dependencies>
3         <dependency>
4             <groupId>org.springframework.boot</groupId>
5             <artifactId>spring-boot-starter-web</artifactId>
6             <version>1.2.3.RELEASE</version>
7         </dependency>
8     </dependencies>
9 </dependencyManagement>
```

此配置即生命了spring-boot的版本信息。

子项目则无需指定版本信息：

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-web</artifactId>
4 </dependency>
```

springboot源码总结：

springboot源码分两部分:

1. @SpringBootApplication讲解。
2. SpringApplication.run(Springboot01DemoApplication.class, args)讲解。

```
1 | @SpringBootApplication//能够扫描Spring组件并自动配置Spring boot
2 | public class Springboot01DemoApplication {
3 |     public static void main(String[] args) {
4 |         SpringApplication.run(Springboot01DemoApplication.class, args);//1. 实例化
        SpringApplication对象; run(args): 调用run方法
5 |     }
6 | }
```

@SpringBootApplication讲解:

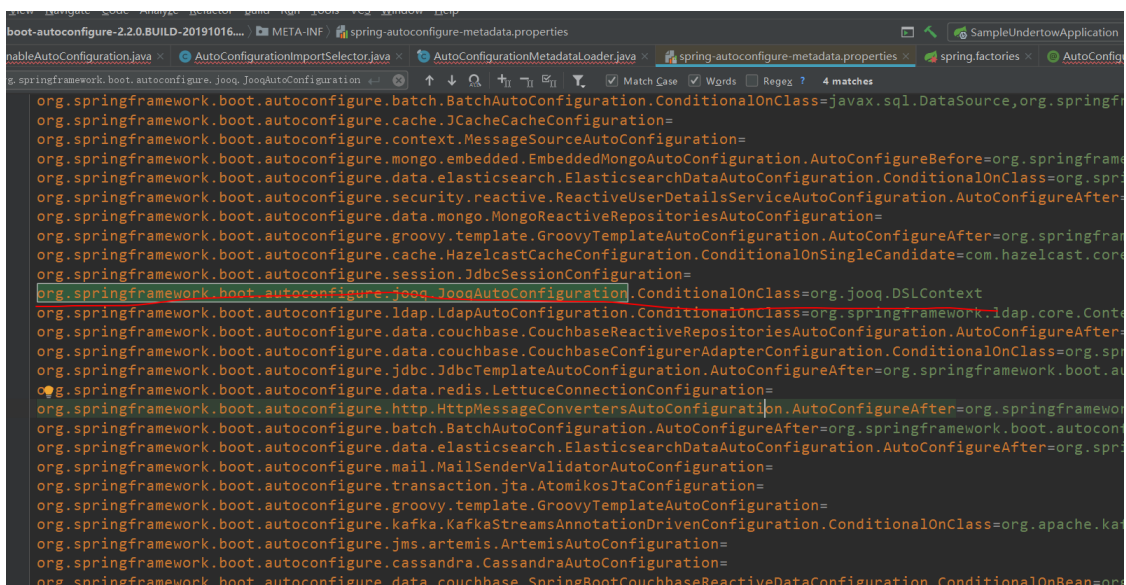
```
1 | @SpringBootApplication
2 |     |-@SpringBootConfiguration
3 |         |- @Configuration //通过javaConfig的方式添加组件到IOC容器中
4 |     |-@EnableAutoConfiguration
5 |         |-@AutoConfigurationPackage //自动配置包, 与@ComponentScan扫描到的添加到IOC
6 |         |-@Import(AutoConfigurationImportSelector.class)//到META-INF/spring.factories
        中定义的bean添加到IOC容器中
7 |     |- @ComponentScan //包扫描
```

springboot实现自动配置@EnableAutoConfiguration:

1. 通过registry.registerBeanDefinition(BEAN, beanDefinition)将ComponentScan扫描到的bean添加到IOC
2. 过滤META-INF/spring.factories中定义的bean添加到IOC容器中:

实现细节:

- 加载配置文件META-INF/spring-autoconfigure-metadata.properties, 从中获取所有支持自动配置类的条件 (将properties文件的内容遍历封装成AutoConfigurationMetadata对象返回)。



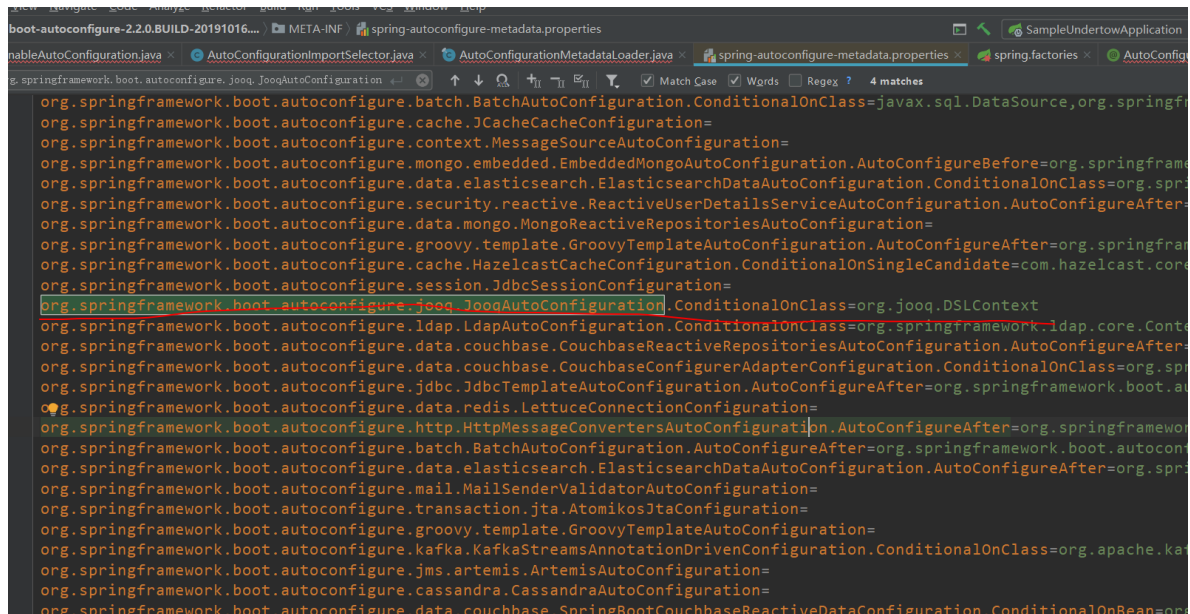
```
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration.ConditionalOnClass=javax.sql.DataSource,org.springframework
org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration=
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration=
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration.AutoConfigureBefore=org.springframework
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration.ConditionalOnClass=org.spr
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration.AutoConfigureAfter=
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration=
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration.AutoConfigureAfter=org.springfram
org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration.ConditionalOnSingleCandidate=com.hazelcast.core
org.springframework.boot.autoconfigure.session.JdbcSessionConfiguration=
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration.ConditionalOnClass=org.jooq.DSLContext
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration.ConditionalOnClass=org.springframework.ldap.core.Cont
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration.AutoConfigureAfter=
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseConfigurerAdapterConfiguration.ConditionalOnClass=org.spr
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration.AutoConfigureAfter=org.springframework.boot.au
org.springframework.boot.autoconfigure.data.redis.LettuceConnectionConfiguration=
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration.AutoConfigureAfter=org.springframework
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconf
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration.AutoConfigureAfter=org.spr
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration=
org.springframework.boot.autoconfigure.transaction.jta.AtomikosJtaConfiguration=
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration=
org.springframework.boot.autoconfigure.kafka.KafkaStreamsAnnotationDrivenConfiguration.ConditionalOnClass=org.apache.kaf
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration=
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration=
org.springframework.boot.autoconfigure.data.couchbase.SpringBootCouchbaseReactiveDataConfiguration.ConditionalOnBean=org
```

- springboot在启动的时候, 使用内部工具类SpringFactoriesLoader, 查找classpath上所有jar包中的META-INF/spring.factories, 找出其中key为org.springframework.boot.autoconfigure.EnableAutoConfiguration的属性定义的工厂类名称, 将这些值作为自动配置类导入到容器中, 自动配置类就生效了

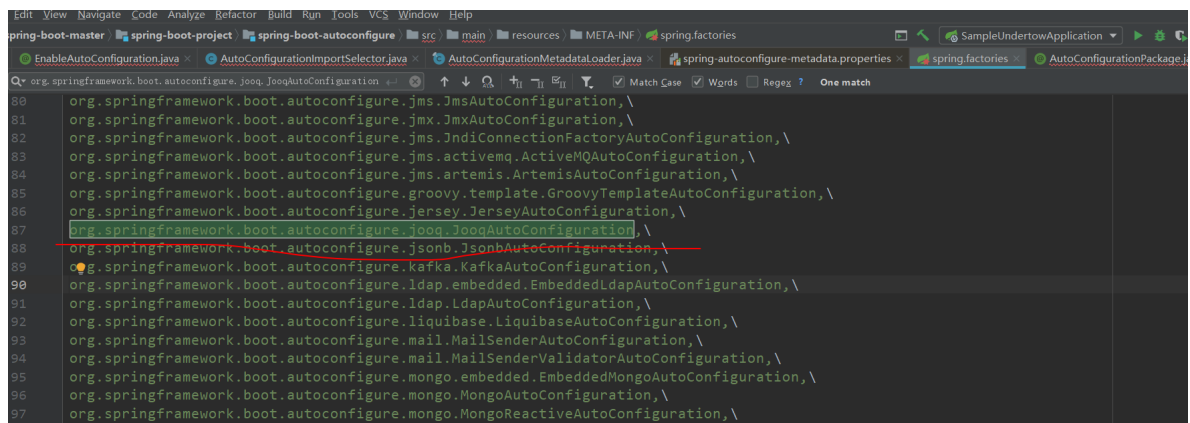
```
1 List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
```

- filter(configurations, autoConfigurationMetadata): 过滤自动配置类

例：如下图，只有JooqAutoConfiguration类上有@ConditionalOnClass({DSLContext.class})时才会自动加载，其他也差不多这个意思。



```
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration.ConditionalOnClass=javax.sql.DataSource,org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration=org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration=org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration.AutoConfigureBefore=org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration.ConditionalOnClass=org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration=org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration.ConditionalOnSingleCandidate=com.hazelcast.core.HazelcastInstance,org.springframework.boot.autoconfigure.session.JdbcSessionConfiguration=org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration.ConditionalOnClass=org.jooq.DSLContext,org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration.ConditionalOnClass=org.springframework.ldap.core.Context,org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration=org.springframework.boot.autoconfigure.data.couchbase.CouchbaseConfigurerAdapterConfiguration.ConditionalOnClass=org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.data.redis.LettuceConnectionFactoryConfiguration=org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration=org.springframework.boot.autoconfigure.transaction.jta.AtomikosJtaConfiguration=org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration=org.springframework.boot.autoconfigure.kafka.KafkaStreamsAnnotationDrivenConfiguration.ConditionalOnClass=org.apache.kafka.clients.producer.ProducerConfig,org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration=org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration=org.springframework.boot.autoconfigure.data.couchbase.SpringBootCouchbaseReactiveDataConfiguration.ConditionalOnBean=org.springframework.boot.autoconfigure.data.couchbase.SpringBootCouchbaseReactiveDataConfiguration
```



```
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\norg.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\norg.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\norg.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\norg.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\norg.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\norg.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\norg.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\norg.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\norg.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\norg.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\norg.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\norg.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\norg.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\norg.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\norg.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\norg.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\norg.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\n
```

```

* @author Michael Simons
* @author Dmytro Nosan
* @since 1.3.0
*/
@Configuration
@ConditionalOnClass(DSLContext.class)
@ConditionalOnBean(DataSource.class)
@AutoConfigureAfter({ DataSourceAutoConfiguration.class,
    TransactionAutoConfiguration.class })
public class JooqAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public DataSourceConnectionProvider dataSourceConnectionProvider(
        DataSource dataSource) {
        return new DataSourceConnectionProvider(
            new TransactionAwareDataSourceProxy(dataSource));
    }

    @Bean
    @ConditionalOnBean(PlatformTransactionManager.class)
    public SpringTransactionProvider transactionProvider(
        PlatformTransactionManager txManager) {
        return new SpringTransactionProvider(txManager);
    }
}

```

SpringApplication.run(SpringBootDemoApplication.class, args) 讲解:

```

1 public static ConfigurableApplicationContext run(Class<?>[] primarySources,
2 String[] args) {
3     //SpringApplication的启动由两部分组成:
4     //1. 实例化SpringApplication对象
5     //2. run(args): 调用run方法
6     return new SpringApplication(primarySources).run(args);
7 }

```