springboot启动源码分析和自动配置原理

1、springboot启动原理

1.1、简述

从前面我们搭建一个springboot工程我们知道了，一个springboot项目的启动非常简单，只要启动一个main方法就可以完成整个项目的启动，下面我们来看看一个启动类的代码：

```java
@SpringBootApplication
// Servlet filter listener
@ServletComponentScan(basePackages = "com.gp.wy")
@MapperScan("com.gp.wy.dao")
public class SpringBootApp {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootApp.class,args);
    }
}
```

可以看到这个main方法中只有一行代码

SpringApplication.run(SpringBootApp.class,args);就是这个run方法，执行这个run方法就完成了一个springboot项目的启动，那么这个run方法到底是完成了哪些工作呢？其实在分析源码之前我们大概也可以猜一下，其实这个run方法要大体上完成两个事情：

1、完成spring容器的启动，把需要扫描的类实例化

2、启动Servlet容器，完成Servlet容器的启动

那么接下来我们就从源码的角度来分析一下，这两个步骤是如何完成的

1.2、源码分析

我们先进入SpringApplication.run(SpringBootApp.class,args);的run方法，我们来看看spring容器是如何启动的

1.2.1、spring容器的启动

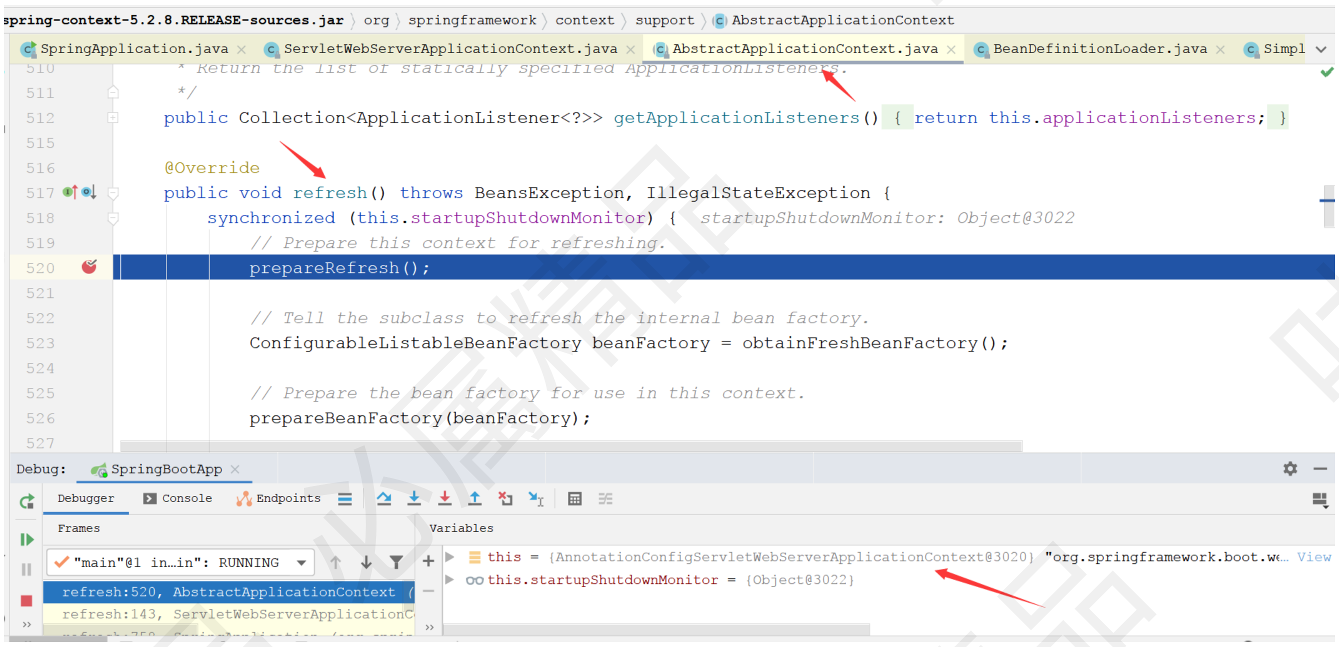
首先我们来看看run方法

```java
public ConfigurableApplicationContext run(String... args) {
   StopWatch stopWatch = new StopWatch();
   stopWatch.start();
   ConfigurableApplicationContext context = null;
   Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
   configureHeadlessProperty();
    //SPI的方式获取SpringApplicationRunListener实例
   SpringApplicationRunListeners listeners = getRunListeners(args);
```

```java
     //调用SpringApplicationRunListener的starting()方法
    listeners.starting();
    try {
       ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
       //生成Environment对象
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
       configureIgnoreBeanInfo(environment);
        //打印banner图
       Banner printedBanner = printBanner(environment);
        //创建springboot的上下文对象AnnotationConfigServletWebServerApplicationContext
       context = createApplicationContext();
       exceptionReporters = getSpringFactoriesInstances(SpringBootExceptionReporter.class,
             new Class[] { ConfigurableApplicationContext.class }, context);
       //初始化上下文对象
        prepareContext(context, environment, listeners, applicationArguments,
printedBanner);
        //spring容器启动的核心代码
       refreshContext(context);
       afterRefresh(context, applicationArguments);
       stopWatch.stop();
       if (this.logStartupInfo) {
          new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopWatch);
       }
       listeners.started(context);
       callRunners(context, applicationArguments);
    }
    catch (Throwable ex) {
       handleRunFailure(context, ex, exceptionReporters, listeners);
       throw new IllegalStateException(ex);
    }

    try {
       //调用SpringApplicationRunListener的running()方法
       listeners.running(context);
    }
    catch (Throwable ex) {
       handleRunFailure(context, ex, exceptionReporters, null);
       throw new IllegalStateException(ex);
    }
    return context;
}
```

从上面的代码分析来看，spring启动的核心代码就是**refreshContext(context);**其上下文对象就是**AnnotationConfigServletWebServerApplicationContext**

接下来我们重点分析一下这个refreshContext(context)方法。首先我们断点看看

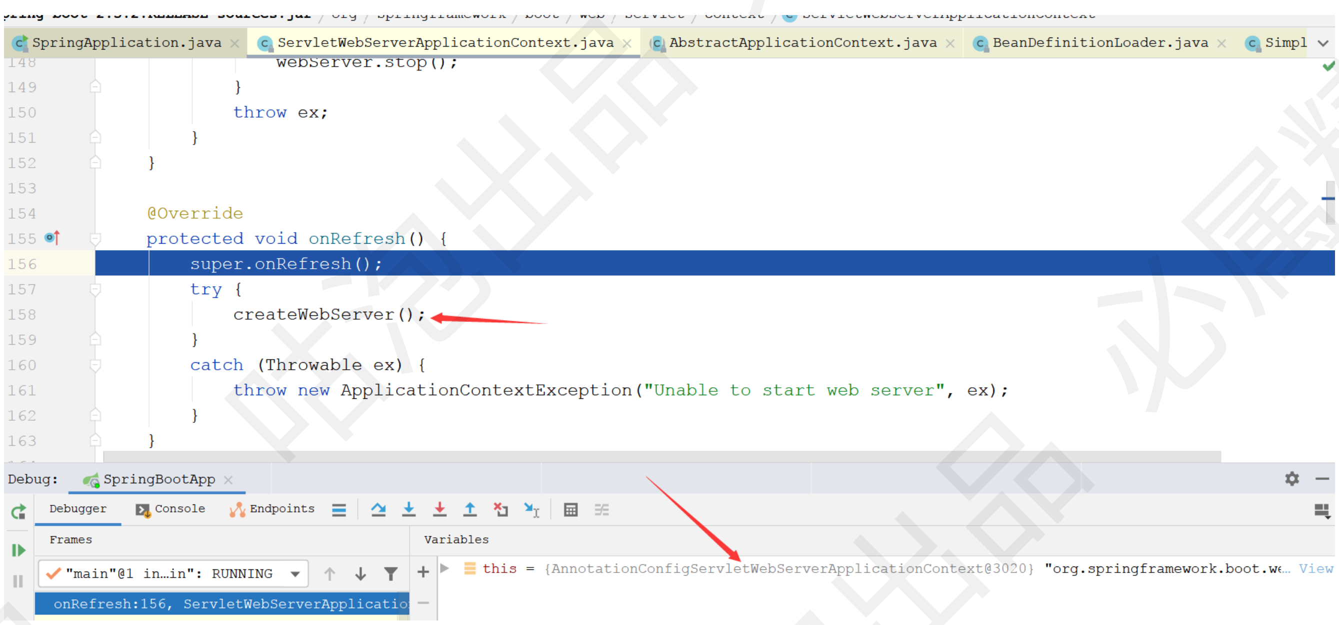

```
spring-context-5.2.8.RELEASE-sources.jar 〉 org 〉 springframework 〉 context 〉 support 〉 AbstractApplicationContext
SpringApplication.java × | ServletWebServerApplicationContext.java × | AbstractApplicationContext.java × | BeanDefinitionLoader.java × | Simpl

	 * Return the list of statically specified ApplicationListeners.
	 */
	public Collection<ApplicationListener<?>> getApplicationListeners() { return this.applicationListeners; }

	@Override
	public void refresh() throws BeansException, IllegalStateException {
		synchronized (this.startupShutdownMonitor) {  startupShutdownMonitor: Object@3022
			// Prepare this context for refreshing.
			prepareRefresh();

			// Tell the subclass to refresh the internal bean factory.
			ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

			// Prepare the bean factory for use in this context.
			prepareBeanFactory(beanFactory);
```
```
Debug: SpringBootApp ×
Debugger  Console  Endpoints
Frames                                   Variables
"main"@1 in…in": RUNNING                 this = {AnnotationConfigServletWebServerApplicationContext@3020} "org.springframework.boot.we… View
refresh:520, AbstractApplicationContext  this.startupShutdownMonitor = {Object@3022}
refresh:143, ServletWebServerApplicationC
```

启动main方法，断点最终来到了**AnnotationConfigServletWebServerApplicationContext**上下文对象的refresh核心方法，而这个方法就是spring容器启动的核心方法，执行了这个方法就可以完成spring容器的启动。

从上述分析来看，springboot的spring容器的启动最终是掉到了上下文对象的refresh方法的。

1.2.2、Servlet容器的启动

前面我们分析过了spring容器的启动了，那么在原先springboot启动类的main方法的时候是如何完成Servlet容器的启动的呢？ 这里以启动tomcat为例，tomcat的启动的地方也是在refresh方法中，具体就是**onRefresh();**方法。


```
SpringApplication.java × | ServletWebServerApplicationContext.java × | AbstractApplicationContext.java × | BeanDefinitionLoader.java × | Simpl

				webServer.stop();
			}
			throw ex;
		}
	}

	@Override
	protected void onRefresh() {
		super.onRefresh();
		try {
			createWebServer();
		}
		catch (Throwable ex) {
			throw new ApplicationContextException("Unable to start web server", ex);
		}
	}
```
```
Debug: SpringBootApp ×
Debugger  Console  Endpoints
Frames                                   Variables
"main"@1 in…in": RUNNING                 this = {AnnotationConfigServletWebServerApplicationContext@3020} "org.springframework.boot.we… View
onRefresh:156, ServletWebServerApplicatio
```

从截图来看，有一个**createWebServer();**方法，就是在这个方法里面完成了tomcat容器的启动，接下来我们重点分析一下这个方法。

```java
private void createWebServer() {
    WebServer webServer = this.webServer;
    ServletContext servletContext = getServletContext();
    if (webServer == null && servletContext == null) {
        //这里获取TomcatServletWebServerFactory对象
       ServletWebServerFactory factory = getWebServerFactory();
        //这里new了Tomcat对象，启动了tomcat容器
       this.webServer = factory.getWebServer(getSelfInitializer());
       getBeanFactory().registerSingleton("webServerGracefulShutdown",
             new WebServerGracefulShutdownLifecycle(this.webServer));
       getBeanFactory().registerSingleton("webServerStartStop",
             new WebServerStartStopLifecycle(this, this.webServer));
    }
    else if (servletContext != null) {
       try {
          getSelfInitializer().onStartup(servletContext);
       }
       catch (ServletException ex) {
          throw new ApplicationContextException("Cannot initialize servlet context", ex);
       }
    }
    initPropertySources();
 }
```

接下来我们来看看this.webServer = factory.getWebServer(getSelfInitializer());这行代码

```
@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
   if (this.disableMBeanRegistry) {
      Registry.disableRegistry();
   }
    //new Tomcat对象
   Tomcat tomcat = new Tomcat();
   File baseDir = (this.baseDirectory != null) ? this.baseDirectory :
createTempDir("tomcat");
   tomcat.setBaseDir(baseDir.getAbsolutePath());
   //HTTP1.1协议
   Connector connector = new Connector(this.protocol);
   connector.setThrowOnFailure(true);
   tomcat.getService().addConnector(connector);
   //设置Http的属性
   customizeConnector(connector);
   tomcat.setConnector(connector);
   tomcat.getHost().setAutoDeploy(false);
   configureEngine(tomcat.getEngine());
   for (Connector additionalConnector : this.additionalTomcatConnectors) {
      tomcat.getService().addConnector(additionalConnector);
   }
   prepareContext(tomcat.getHost(), initializers);
    //开启tomcat服务
   return getTomcatWebServer(tomcat);
}
```

从上面源码分析来看，确实是启动了一个tomcat容器并且把项目部署到了tomcat中了。

2、springboot自动配置

2.1、简述

springboot的自动配置功能是其简化运用的关键技术，比如Aop、事务、缓存等的自动配置功能，就是我们在使用事务时根本就不需要配置有关事务的逻辑，只需要直接在业务代码中使用事务注解就可以了，这个就是springboot的自动配置功能。自动配置的思想就是**约定大于配置**，意思就是一个工程约定必须要有事务功能，要有aop功能，要有mvc功能，所以springboot在创建工程的时候自动把这些功能所需的类实例化并加入到spring容器了，这个就是约定大于配置，约定了必须要有这些功能。

那么springboot如何完成自动配置功能的呢？**首先我们必须掌握springboot中的SPI机制是如何使用和实现的**

2.2、什么是SPI

2.2.1、SPI简介

SPI ，全称为 Service Provider Interface，是一种服务发现机制。它通过在ClassPath路径下的META-INF/services文件夹查找文件，自动加载文件里所定义的类。 这一机制为很多框架扩展提供了可能，比如在Dubbo、JDBC中都使用到了SPI机制。我们先通过一个很简单的例子来看下它是怎么用的。

简单来说，SPI是一种扩展机制，核心就是将服务配置化，在核心代码不用改动的前提下，通过加载配置文件中的服务，然后根据传递的参数来决定到底走什么逻辑，走哪个服务的逻辑。这样就对扩展是开放的，对修改是关闭的。

2.2.2、SPI的运用场景

当你在写核心代码的时候，如果某个点有涉及到会根据参数的不同走不同的逻辑的时候，如果没有SPI，你可能会在代码里面写大量的if else代码，这样代码就非常不灵活，假设有一天又新增了一种逻辑，代码里面也要跟着改，这个就违背了开闭原则，SPI的出现就是解决这种扩展问题的，你可以把实现类全部都配置到配置文件中，然后在核心代码里面就只要加载配置文件，然后根据入参跟加载的这些类进行匹配，如果匹配的就走该逻辑，这样如果有一天新增了逻辑，核心代码是不用变的，唯一变的就是自己工程里面的配置文件和新增类，符合了开闭原则。

2.3、JDK中的SPI机制

前面已经介绍过SPI是什么以及在哪些地方方用了，在这里就重点介绍一下SPI在JDK中的实现，下面我们看看具体的

2.3.1、案例

1、顶层接口

```
public interface GLog {
    boolean support(String type);
    void debug();
    void info();
}
```

2、接口实现

```
public class Log4j implements GLog {
    @Override
    public boolean support(String type) {
        return "log4j".equalsIgnoreCase(type);
    }

    @Override
```

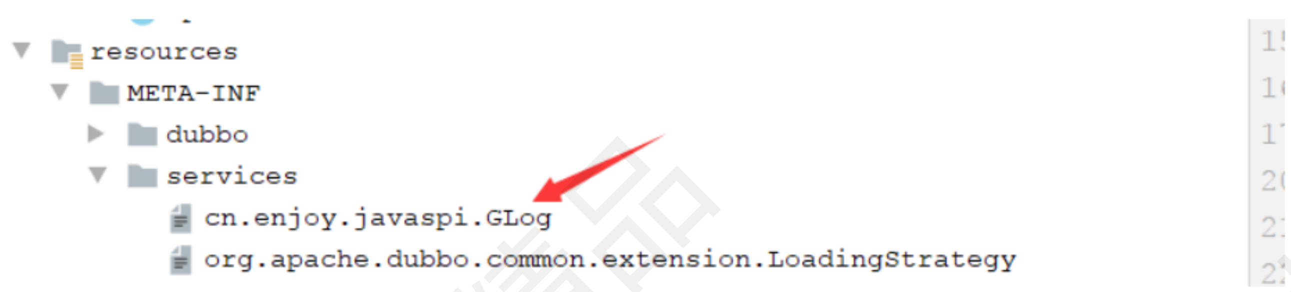
```java
    public void debug() {
        System.out.println("====log4j.debug======");
    }

    @Override
    public void info() {
        System.out.println("====log4j.info======");
    }
}
```

```java
public class Logback implements GLog {
    @Override
    public boolean support(String type) {
        return "Logback".equalsIgnoreCase(type);
    }

    @Override
    public void debug() {
        System.out.println("====Logback.debug======");
    }

    @Override
    public void info() {
        System.out.println("====Logback.info======");
    }
}
```

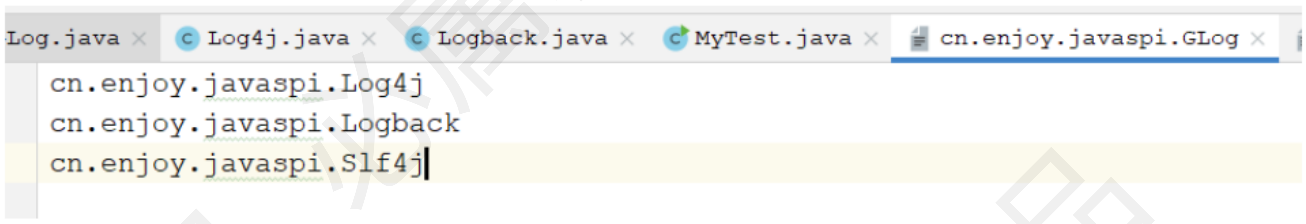
```java
public class Slf4j implements GLog {
    @Override
    public boolean support(String type) {
        return "Slf4j".equalsIgnoreCase(type);
    }

    @Override
    public void debug() {
        System.out.println("====Slf4j.debug======");
    }

    @Override
    public void info() {
        System.out.println("====Slf4j.info======");
    }
}
```

3、文件配置

在resources/META-INF/services目录创建文件，**文件名称必须跟接口的完整限定名相同。**如图：



这个接口文件中配置了该接口的所有实现类的完整限定名，如图：



```
cn.enjoy.javaspi.Log4j
cn.enjoy.javaspi.Logback
cn.enjoy.javaspi.Slf4j
```

现在要根据输入的参数来决定到底是走Log4j的逻辑还是Logback的逻辑。

4、单元测试

```
public class MyTest {

    //不同的入参，对应调用的逻辑是不一样的
    public static void main(String[] args) {
        //这个是我们业务的核心代码，核心代码会根据外部的参数决定要掉哪一个实例
        //可以去读配置文件 properties配置，去决定掉哪个实例
        //jdk api  加载配置文件配置实例
        ServiceLoader<GLog> all = ServiceLoader.load(GLog.class);
        Iterator<GLog> iterator = all.iterator();

        Scanner scanner = new Scanner(System.in);
        String s = scanner.nextLine();

        while (iterator.hasNext()) {
            GLog next = iterator.next();
            //这个实例是不是我们需要掉的
            // 策略模式  当前实例是不是跟入参匹配
            if(next.support(s)) {
                next.debug();
            }
        }
    }
}
```

2.4、springboot中的SPI

2.4.1、案例

接下来我们来看看springboot中的spi如何使用的，其实大体的流程跟jdk中的spi使用流程差不多

1、定义接口

```
public interface Log {
    void debug();
}
```

2、接口实现

```
public class Log4j implements Log {
    @Override
    public void debug() {
        System.out.println("-----Log4j");
    }
}

public class Logback implements Log {
    @Override
    public void debug() {
        System.out.println("-----Logback");
    }
}

public class Slf4j implements Log {
    @Override
    public void debug() {
        System.out.println("-----Slf4j");
    }
}
```

3、spring.factories

在工程的resources下面创建META-INF文件夹，在文件夹下创建spring.factories文件，在文件夹中配置内容如下：

```
com.gp.wy.spi.Log=\
com.gp.wy.spi.Log4j,\
com.gp.wy.spi.Logback,\
com.gp.wy.spi.Slf4j
```

配置的key就是接口完整限定名，value就是接口的各个实现类，用","号隔开。

4、API

loadFactoryNames方法获取实现了接口的所有类的名称

```
@Test
public void test() {
    List<String> strings = SpringFactoriesLoader.loadFactoryNames(Log.class, 
ClassUtils.getDefaultClassLoader());
    for (String string : strings) {
        System.out.println(string);
    }
}
```

loadFactories方法获取实现了接口的所有类的实例

```
@Test
public void test1() {
    List<Log> logs = SpringFactoriesLoader.loadFactories(Log.class,
ClassUtils.getDefaultClassLoader());
    for (Log log : logs) {
        System.out.println(log);
    }
}
```

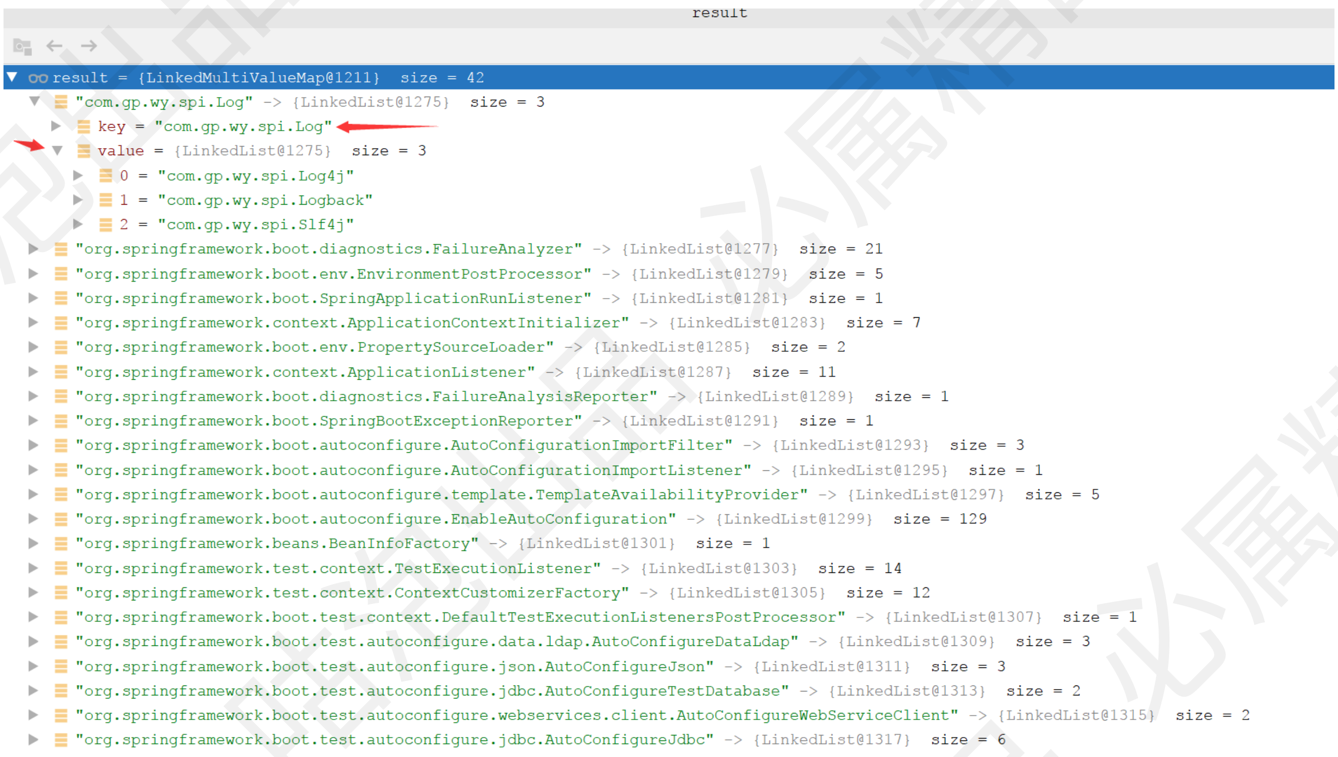
2.4.2、源码分析

我们以SpringFactoriesLoader.loadFactoryNames(Log.class, ClassUtils.getDefaultClassLoader());方法调用为例分析其源码。

```
public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader
classLoader) {
    //获取类型名称
   String factoryTypeName = factoryType.getName();
    //核心代码
   return loadSpringFactories(classLoader).getOrDefault(factoryTypeName,
Collections.emptyList());
}
```

```
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader
classLoader) {
    //先根据classLoader从缓存中拿，如果能拿到就返回
   MultiValueMap<String, String> result = cache.get(classLoader);
   if (result != null) {
      return result;
   }

   try {
       //FACTORIES_RESOURCE_LOCATION
       //public static final String FACTORIES_RESOURCE_LOCATION = "META-
INF/spring.factories";
       //根据类加载器获取该路径下的spring.factories文件
       //获取的文件是所有jar包和自己工程里面的所有spring.factories文件
      Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
      result = new LinkedMultiValueMap<>();
       //循环所有文件
      while (urls.hasMoreElements()) {
         URL url = urls.nextElement();
          //包装成UrlResource对象
         UrlResource resource = new UrlResource(url);
          //核心代码，把文件包装成properties对象
         Properties properties = PropertiesLoaderUtils.loadProperties(resource);
          //循环properties对象
         for (Map.Entry<?, ?> entry : properties.entrySet()) {
             //拿到key
            String factoryTypeName = ((String) entry.getKey()).trim();
            for (String factoryImplementationName :
StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                //把key对应的所有实例加入到list容器中
               result.add(factoryTypeName, factoryImplementationName.trim());
            }
         }
      }
       //建立缓存
      cache.put(classLoader, result);
      return result;
   }
   catch (IOException ex) {
      throw new IllegalArgumentException("Unable to load factories from location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
   }
}
```

我们看看这个result的结果：



```
result
▼ result = {LinkedMultiValueMap@1211}  size = 42
  ▼ "com.gp.wy.spi.Log" -> {LinkedList@1275}  size = 3
    ▶ key = "com.gp.wy.spi.Log"
    ▼ value = {LinkedList@1275}  size = 3
      ▶ 0 = "com.gp.wy.spi.Log4j"
      ▶ 1 = "com.gp.wy.spi.Logback"
      ▶ 2 = "com.gp.wy.spi.Slf4j"
  ▶ "org.springframework.boot.diagnostics.FailureAnalyzer" -> {LinkedList@1277}  size = 21
  ▶ "org.springframework.boot.env.EnvironmentPostProcessor" -> {LinkedList@1279}  size = 5
  ▶ "org.springframework.boot.SpringApplicationRunListener" -> {LinkedList@1281}  size = 1
  ▶ "org.springframework.context.ApplicationContextInitializer" -> {LinkedList@1283}  size = 7
  ▶ "org.springframework.boot.env.PropertySourceLoader" -> {LinkedList@1285}  size = 2
  ▶ "org.springframework.context.ApplicationListener" -> {LinkedList@1287}  size = 11
  ▶ "org.springframework.boot.diagnostics.FailureAnalysisReporter" -> {LinkedList@1289}  size = 1
  ▶ "org.springframework.boot.SpringBootExceptionReporter" -> {LinkedList@1291}  size = 1
  ▶ "org.springframework.boot.autoconfigure.AutoConfigurationImportFilter" -> {LinkedList@1293}  size = 3
  ▶ "org.springframework.boot.autoconfigure.AutoConfigurationImportListener" -> {LinkedList@1295}  size = 1
  ▶ "org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider" -> {LinkedList@1297}  size = 5
  ▶ "org.springframework.boot.autoconfigure.EnableAutoConfiguration" -> {LinkedList@1299}  size = 129
  ▶ "org.springframework.beans.BeanInfoFactory" -> {LinkedList@1301}  size = 1
  ▶ "org.springframework.test.context.TestExecutionListener" -> {LinkedList@1303}  size = 14
  ▶ "org.springframework.test.context.ContextCustomizerFactory" -> {LinkedList@1305}  size = 12
  ▶ "org.springframework.boot.test.context.DefaultTestExecutionListenersPostProcessor" -> {LinkedList@1307}  size = 1
  ▶ "org.springframework.boot.test.autoconfigure.data.ldap.AutoConfigureDataLdap" -> {LinkedList@1309}  size = 3
  ▶ "org.springframework.boot.test.autoconfigure.json.AutoConfigureJson" -> {LinkedList@1311}  size = 3
  ▶ "org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase" -> {LinkedList@1313}  size = 2
  ▶ "org.springframework.boot.test.autoconfigure.webservices.client.AutoConfigureWebServiceClient" -> {LinkedList@1315}  size = 2
  ▶ "org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureJdbc" -> {LinkedList@1317}  size = 6
```

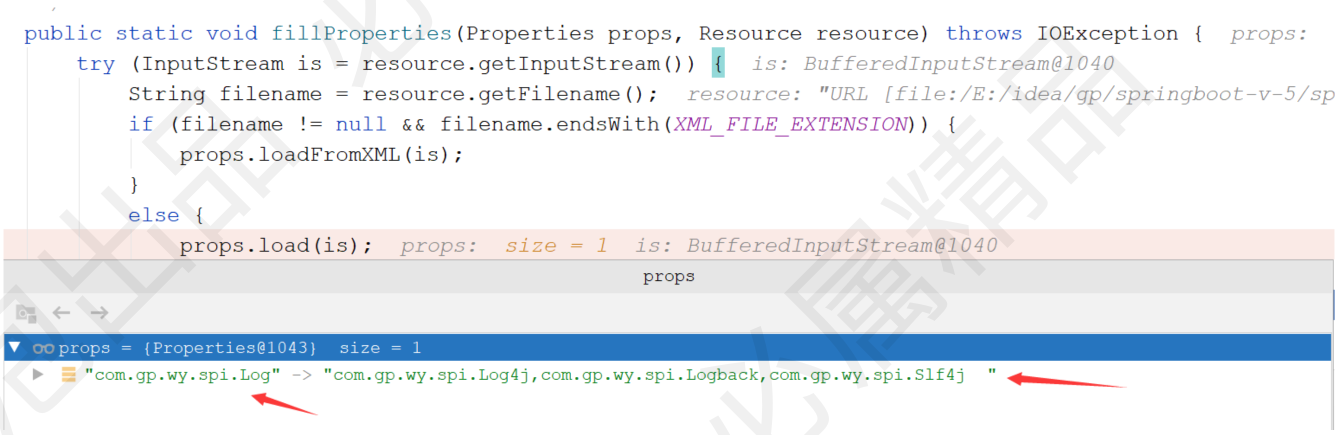
可以看到springboot spi是加载了整个工程的jar包和自己工程定义的spring.factories文件的。

接着我们看看Properties properties = PropertiesLoaderUtils.loadProperties(resource);

```
public static Properties loadProperties(Resource resource) throws IOException {
   Properties props = new Properties();
    //核心代码，把文件包装成properties对象
   fillProperties(props, resource);
   return props;
}
```

```
public static void fillProperties(Properties props, Resource resource) throws IOException {
   try (InputStream is = resource.getInputStream()) {
      String filename = resource.getFilename();
      if (filename != null && filename.endsWith(XML_FILE_EXTENSION)) {
         props.loadFromXML(is);
      }
      else {
         props.load(is);
      }
   }
}
```

properties结构如下:



```
public static void fillProperties(Properties props, Resource resource) throws IOException {  props:
    try (InputStream is = resource.getInputStream()) {   is: BufferedInputStream@1040
        String filename = resource.getFilename();  resource: "URL [file:/E:/idea/gp/springboot-v-5/sp
        if (filename != null && filename.endsWith(XML_FILE_EXTENSION)) {
            props.loadFromXML(is);
        }
        else {
            props.load(is);   props:  size = 1   is: BufferedInputStream@1040
```

```
props
▼ props = {Properties@1043}  size = 1
  ▶ "com.gp.wy.spi.Log" -> "com.gp.wy.spi.Log4j,com.gp.wy.spi.Logback,com.gp.wy.spi.Slf4j  "
```

可以看到就是建立了key和value的关系，我们可以看到springboot中的spi其实就是加载整个工程里面的spring.factories文件，然后把文件里面的内容建立一个key和value的映射关系，**只是这个映射关系是一个类型和list的映射关系**

```
SpringFactoriesLoader.loadFactories(Log.class, ClassUtils.getDefaultClassLoader());
```

该方法的源码其实跟上面方法源码类似，是调用了loadFactoryNames拿到了接口类型对应的所有类的名称，然后反射实例化了而已，这里就不分析了。

2.5、@EnableAutoConfiguration

@EnableAutoConfiguration注解是springboot自动配置的核心注解，就是因为有这个注解存在就会把例如事务，缓存，aop，mvc等功能自动导入到springboot工程中，那它是如何做到自动导入功能的呢？我们慢慢分析，我们来看看这个注解

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

可以看到这个注解@Import了一个类，我们看看这个**AutoConfigurationImportSelector**自动配置类，在看这个类之前我们先看看一个案例；

2.5.1、DeferredImportSelector

DeferredImportSelector该接口是ImportSelector接口的一个子接口，那么它是如何使用的呢？我们来看看案例

2.5.1.1、DeferredImportSelectorDemo

自定义一个类实现DeferredImportSelector接口，**该类必须是@Import导入进来**

```
public class DeferredImportSelectorDemo implements DeferredImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        System.out.println("=====DeferredImportSelectorDemo.selectImports");
        return new String[]{DeferredBean.class.getName()};
    }

    /**
     * 要返回一个实现了Group接口的类
    */
    @Override
    public Class<? extends Group> getImportGroup() {
        return DeferredImportSelectorGroupDemo.class;
    }

    private static class DeferredImportSelectorGroupDemo implements Group {

        List<Entry> list = new ArrayList<>();
        /**
            收集需要实例化的类
        */
        @Override
        public void process(AnnotationMetadata metadata, DeferredImportSelector selector) {
            System.out.println("=====DeferredImportSelectorGroupDemo.process");
            String[] strings = selector.selectImports(metadata);
            for (String string : strings) {
                list.add(new Entry(metadata,string));
            }
        }

        //把收集到的类返回给spring容器
        @Override
        public Iterable<Entry> selectImports() {
            System.out.println("=====DeferredImportSelectorGroupDemo.selectImports");
            return list;
        }
    }
}
```

要实例的bean

```
public class DeferredBean {
}
```

把DeferredImportSelectorDemo import进来

```
@Component
//Import虽然是实例化一个类，Import进来的类可以实现一些接口
@Import({DeferredImportSelectorDemo.class})
public class ImportBean {
}
```

2.5.2、AutoConfigurationImportSelector

接下来我们再来看看AutoConfigurationImportSelector类，这个类就是@EnableAutoConfiguration注解中@Import进来的类，我们来分析一下该类，可以看到该类正是实现了DeferredImportSelector接口的,请同学认真看看我写了注释的代码，这些代码才是关键代码：

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

   private static final AutoConfigurationEntry EMPTY_ENTRY = new AutoConfigurationEntry();

   private static final String[] NO_IMPORTS = {};

   private static final Log logger =
LogFactory.getLog(AutoConfigurationImportSelector.class);

   private static final String PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE =
"spring.autoconfigure.exclude";

   private ConfigurableListableBeanFactory beanFactory;

   private Environment environment;

   private ClassLoader beanClassLoader;

   private ResourceLoader resourceLoader;

   private ConfigurationClassFilter configurationClassFilter;

   @Override
   public String[] selectImports(AnnotationMetadata annotationMetadata) {
      if (!isEnabled(annotationMetadata)) {
         return NO_IMPORTS;
      }
      AutoConfigurationEntry autoConfigurationEntry =
getAutoConfigurationEntry(annotationMetadata);
      return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
   }

   @Override
   public Predicate<String> getExclusionFilter() {
      return this::shouldExclude;
   }

   private boolean shouldExclude(String configurationClassName) {
      return
getConfigurationClassFilter().filter(Collections.singletonList(configurationClassName)).isEm
pty();
   }

   /**
    * Return the {@link AutoConfigurationEntry} based on the {@link AnnotationMetadata}
    * of the importing {@link Configuration @Configuration} class.
    * @param annotationMetadata the annotation metadata of the configuration class
    * @return the auto-configurations that should be imported
    */
   protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata
annotationMetadata) {
      if (!isEnabled(annotationMetadata)) {
         return EMPTY_ENTRY;
      }
       //获取@SpringBootApplication的配置属性
      AnnotationAttributes attributes = getAttributes(annotationMetadata);
       //获取候选的所有类的名称
      List<String> configurations = getCandidateConfigurations(annotationMetadata,
attributes);
      configurations = removeDuplicates(configurations);
       //从注解配置属性中获取需要排除的类
      Set<String> exclusions = getExclusions(annotationMetadata, attributes);
      checkExcludedClasses(configurations, exclusions);
       //从候选的类中删除需要排除的类
      configurations.removeAll(exclusions);
       //SPI的扩展，获取过滤器实例对候选的类过滤
      configurations = getConfigurationClassFilter().filter(configurations);
      fireAutoConfigurationImportEvents(configurations, exclusions);
       //把候选的所有类包装成AutoConfigurationEntry对象
      return new AutoConfigurationEntry(configurations, exclusions);
   }

   //该类实现了getImportGroup()接口，所有启动时就会调用到AutoConfigurationGroup类的方法
   @Override
   public Class<? extends Group> getImportGroup() {
      return AutoConfigurationGroup.class;
   }

   protected boolean isEnabled(AnnotationMetadata metadata) {
      if (getClass() == AutoConfigurationImportSelector.class) {
         return
getEnvironment().getProperty(EnableAutoConfiguration.ENABLED_OVERRIDE_PROPERTY,
Boolean.class, true);
      }
      return true;
   }

   /**
    * Return the appropriate {@link AnnotationAttributes} from the
    * {@link AnnotationMetadata}. By default this method will return attributes for
    * {@link #getAnnotationClass()}.
```

```
     * @param metadata the annotation metadata
     * @return annotation attributes
     */
   protected AnnotationAttributes getAttributes(AnnotationMetadata metadata) {
      String name = getAnnotationClass().getName();
      AnnotationAttributes attributes =
AnnotationAttributes.fromMap(metadata.getAnnotationAttributes(name, true));
      Assert.notNull(attributes, () -> "No auto-configuration attributes found. Is " +
metadata.getClassName()
            + " annotated with " + ClassUtils.getShortName(name) + "?");
      return attributes;
   }

   /**
    * Return the source annotation class used by the selector.
    * @return the annotation class
    */
   protected Class<?> getAnnotationClass() {
      return EnableAutoConfiguration.class;
   }

   /**
    * Return the auto-configuration class names that should be considered. By default
    * this method will load candidates using {@link SpringFactoriesLoader} with
    * {@link #getSpringFactoriesLoaderFactoryClass()}.
    * @param metadata the source metadata
    * @param attributes the {@link #getAttributes(AnnotationMetadata) annotation
    * attributes}
    * @return a list of candidate configurations
    */
   protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
AnnotationAttributes attributes) {
       //SPI的方式获取EnableAutoConfiguration.class类型的所有类的名称
      List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
            getBeanClassLoader());
      Assert.notEmpty(configurations, "No auto configuration classes found in META-
INF/spring.factories. If you "
            + "are using a custom packaging, make sure that file is correct.");
      return configurations;
   }

   /**
    * Return the class used by {@link SpringFactoriesLoader} to load configuration
    * candidates.
    * @return the factory class
    */
   protected Class<?> getSpringFactoriesLoaderFactoryClass() {
      return EnableAutoConfiguration.class;
   }

   private void checkExcludedClasses(List<String> configurations, Set<String> exclusions) {
      List<String> invalidExcludes = new ArrayList<>(exclusions.size());
      for (String exclusion : exclusions) {
         if (ClassUtils.isPresent(exclusion, getClass().getClassLoader()) &&
!configurations.contains(exclusion)) {
            invalidExcludes.add(exclusion);
         }
      }
      if (!invalidExcludes.isEmpty()) {
         handleInvalidExcludes(invalidExcludes);
      }
   }

   /**
    * Handle any invalid excludes that have been specified.
    * @param invalidExcludes the list of invalid excludes (will always have at least one
    * element)
    */
   protected void handleInvalidExcludes(List<String> invalidExcludes) {
      StringBuilder message = new StringBuilder();
      for (String exclude : invalidExcludes) {
         message.append("\t- ").append(exclude).append(String.format("%n"));
      }
      throw new IllegalStateException(String.format(
            "The following classes could not be excluded because they are not auto-
configuration classes:%n%s",
            message));
   }

   /**
    * Return any exclusions that limit the candidate configurations.
    * @param metadata the source metadata
    * @param attributes the {@link #getAttributes(AnnotationMetadata) annotation
    * attributes}
    * @return exclusions or an empty set
    */
   protected Set<String> getExclusions(AnnotationMetadata metadata, AnnotationAttributes
attributes) {
      Set<String> excluded = new LinkedHashSet<>();
      excluded.addAll(asList(attributes, "exclude"));
      excluded.addAll(Arrays.asList(attributes.getStringArray("excludeName")));
      excluded.addAll(getExcludeAutoConfigurationsProperty());
      return excluded;
   }

   /**
    * Returns the auto-configurations excluded by the
    * {@code spring.autoconfigure.exclude} property.
    * @return excluded auto-configurations
    * @since 2.3.2
    */
   protected List<String> getExcludeAutoConfigurationsProperty() {
      Environment environment = getEnvironment();
      if (environment == null) {
         return Collections.emptyList();
      }
      if (environment instanceof ConfigurableEnvironment) {
         Binder binder = Binder.get(environment);
         return binder.bind(PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE,
String[].class).map(Arrays::asList)
               .orElse(Collections.emptyList());
      }
      String[] excludes = environment.getProperty(PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE,
String[].class);
      return (excludes != null) ? Arrays.asList(excludes) : Collections.emptyList();
   }

   protected List<AutoConfigurationImportFilter> getAutoConfigurationImportFilters() {
      return SpringFactoriesLoader.loadFactories(AutoConfigurationImportFilter.class,
this.beanClassLoader);
   }

   private ConfigurationClassFilter getConfigurationClassFilter() {
      if (this.configurationClassFilter == null) {
         List<AutoConfigurationImportFilter> filters = getAutoConfigurationImportFilters();
         for (AutoConfigurationImportFilter filter : filters) {
            invokeAwareMethods(filter);
         }
         this.configurationClassFilter = new ConfigurationClassFilter(this.beanClassLoader,
filters);
      }
      return this.configurationClassFilter;
   }

   protected final <T> List<T> removeDuplicates(List<T> list) {
      return new ArrayList<>(new LinkedHashSet<>(list));
   }

   protected final List<String> asList(AnnotationAttributes attributes, String name) {
      String[] value = attributes.getStringArray(name);
      return Arrays.asList(value);
   }

   private void fireAutoConfigurationImportEvents(List<String> configurations, Set<String>
exclusions) {
      List<AutoConfigurationImportListener> listeners =
getAutoConfigurationImportListeners();
      if (!listeners.isEmpty()) {
         AutoConfigurationImportEvent event = new AutoConfigurationImportEvent(this,
configurations, exclusions);
         for (AutoConfigurationImportListener listener : listeners) {
            invokeAwareMethods(listener);
            listener.onAutoConfigurationImportEvent(event);
         }
      }
   }

   protected List<AutoConfigurationImportListener> getAutoConfigurationImportListeners() {
      return SpringFactoriesLoader.loadFactories(AutoConfigurationImportListener.class,
this.beanClassLoader);
   }

   private void invokeAwareMethods(Object instance) {
      if (instance instanceof Aware) {
         if (instance instanceof BeanClassLoaderAware) {
            ((BeanClassLoaderAware) instance).setBeanClassLoader(this.beanClassLoader);
         }
         if (instance instanceof BeanFactoryAware) {
            ((BeanFactoryAware) instance).setBeanFactory(this.beanFactory);
         }
         if (instance instanceof EnvironmentAware) {
            ((EnvironmentAware) instance).setEnvironment(this.environment);
         }
         if (instance instanceof ResourceLoaderAware) {
            ((ResourceLoaderAware) instance).setResourceLoader(this.resourceLoader);
         }
      }
   }

   @Override
   public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
      Assert.isInstanceOf(ConfigurableListableBeanFactory.class, beanFactory);
      this.beanFactory = (ConfigurableListableBeanFactory) beanFactory;
   }

   protected final ConfigurableListableBeanFactory getBeanFactory() {
      return this.beanFactory;
   }

   @Override
   public void setBeanClassLoader(ClassLoader classLoader) {
      this.beanClassLoader = classLoader;
   }

   protected ClassLoader getBeanClassLoader() {
      return this.beanClassLoader;
   }

   @Override
   public void setEnvironment(Environment environment) {
      this.environment = environment;
   }

   protected final Environment getEnvironment() {
      return this.environment;
   }

   @Override
   public void setResourceLoader(ResourceLoader resourceLoader) {
      this.resourceLoader = resourceLoader;
   }
   protected final ResourceLoader getResourceLoader() {
      return this.resourceLoader;
   }

   @Override
   public int getOrder() {
      return Ordered.LOWEST_PRECEDENCE - 1;
   }

   private static class ConfigurationClassFilter {

      private final AutoConfigurationMetadata autoConfigurationMetadata;

      private final List<AutoConfigurationImportFilter> filters;

      ConfigurationClassFilter(ClassLoader classLoader, List<AutoConfigurationImportFilter>
filters) {
         this.autoConfigurationMetadata =
AutoConfigurationMetadataLoader.loadMetadata(classLoader);
         this.filters = filters;
      }

      List<String> filter(List<String> configurations) {
         long startTime = System.nanoTime();
         String[] candidates = StringUtils.toStringArray(configurations);
         boolean skipped = false;
         for (AutoConfigurationImportFilter filter : this.filters) {
            boolean[] match = filter.match(candidates, this.autoConfigurationMetadata);
            for (int i = 0; i < match.length; i++) {
               if (!match[i]) {
                  candidates[i] = null;
                  skipped = true;
               }
            }
         }
         if (!skipped) {
            return configurations;
         }
         List<String> result = new ArrayList<>(candidates.length);
         for (String candidate : candidates) {
            if (candidate != null) {
               result.add(candidate);
            }
         }
         if (logger.isTraceEnabled()) {
            int numberFiltered = configurations.size() - result.size();
            logger.trace("Filtered " + numberFiltered + " auto configuration class in "
                  + TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - startTime) + " ms");
         }
         return result;
      }
```

```
    }

    //核心逻辑就在该内部类中
    private static class AutoConfigurationGroup
          implements DeferredImportSelector.Group, BeanClassLoaderAware, BeanFactoryAware,
ResourceLoaderAware {

       private final Map<String, AnnotationMetadata> entries = new LinkedHashMap<>();

       private final List<AutoConfigurationEntry> autoConfigurationEntries = new ArrayList<>
();

       private ClassLoader beanClassLoader;

       private BeanFactory beanFactory;

       private ResourceLoader resourceLoader;

       private AutoConfigurationMetadata autoConfigurationMetadata;

       @Override
       public void setBeanClassLoader(ClassLoader classLoader) {
          this.beanClassLoader = classLoader;
       }

       @Override
       public void setBeanFactory(BeanFactory beanFactory) {
          this.beanFactory = beanFactory;
       }

       @Override
       public void setResourceLoader(ResourceLoader resourceLoader) {
          this.resourceLoader = resourceLoader;
       }

        //该方法收集需要实例化的方法
       @Override
       public void process(AnnotationMetadata annotationMetadata, DeferredImportSelector
deferredImportSelector) {
          Assert.state(deferredImportSelector instanceof AutoConfigurationImportSelector,
                () -> String.format("Only %s implementations are supported, got %s",
                      AutoConfigurationImportSelector.class.getSimpleName(),
                      deferredImportSelector.getClass().getName()));
          //核心代码，SPI的方式获取需要实例化的类。。AutoConfigurationEntry类中包装了所有需要实例化的类的
集合
           AutoConfigurationEntry autoConfigurationEntry = ((AutoConfigurationImportSelector)
deferredImportSelector)
                .getAutoConfigurationEntry(annotationMetadata);
          this.autoConfigurationEntries.add(autoConfigurationEntry);
           //循环所有需要实例化的类，并建立类和注解的映射关系
          for (String importClassName : autoConfigurationEntry.getConfigurations()) {
             this.entries.putIfAbsent(importClassName, annotationMetadata);
          }
```

```
      }

      @Override
      public Iterable<Entry> selectImports() {
         if (this.autoConfigurationEntries.isEmpty()) {
            return Collections.emptyList();
         }
          //获取需要排除类的集合
         Set<String> allExclusions = this.autoConfigurationEntries.stream()

.map(AutoConfigurationEntry::getExclusions).flatMap(Collection::stream).collect(Collectors.t
oSet());
          //获取所有需要实例化的类的集合
         Set<String> processedConfigurations = this.autoConfigurationEntries.stream()
               .map(AutoConfigurationEntry::getConfigurations).flatMap(Collection::stream)
               .collect(Collectors.toCollection(LinkedHashSet::new));
         //删除需要排除的类
          processedConfigurations.removeAll(allExclusions);

          //把需要实例化的类包装成Entry的集合，必须这么写，spring要根据entry实例化对象
         return sortAutoConfigurations(processedConfigurations,
getAutoConfigurationMetadata()).stream()
               .map((importClassName) -> new Entry(this.entries.get(importClassName),
importClassName))
               .collect(Collectors.toList());
      }

      private AutoConfigurationMetadata getAutoConfigurationMetadata() {
         if (this.autoConfigurationMetadata == null) {
            this.autoConfigurationMetadata =
AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
         }
         return this.autoConfigurationMetadata;
      }

      private List<String> sortAutoConfigurations(Set<String> configurations,
            AutoConfigurationMetadata autoConfigurationMetadata) {
         return new AutoConfigurationSorter(getMetadataReaderFactory(),
autoConfigurationMetadata)
               .getInPriorityOrder(configurations);
      }

      private MetadataReaderFactory getMetadataReaderFactory() {
         try {
            return
this.beanFactory.getBean(SharedMetadataReaderFactoryContextInitializer.BEAN_NAME,
                  MetadataReaderFactory.class);
         }
         catch (NoSuchBeanDefinitionException ex) {
            return new CachingMetadataReaderFactory(this.resourceLoader);
         }
      }
```

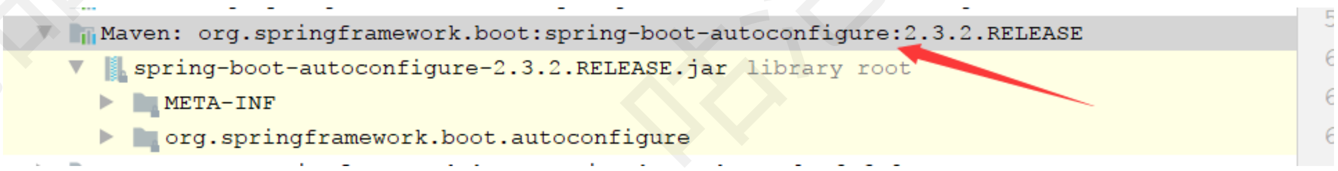
```
    }

    protected static class AutoConfigurationEntry {

        private final List<String> configurations;

        private final Set<String> exclusions;

        private AutoConfigurationEntry() {
            this.configurations = Collections.emptyList();
            this.exclusions = Collections.emptySet();
        }

        /**
         * Create an entry with the configurations that were contributed and their
         * exclusions.
         * @param configurations the configurations that should be imported
         * @param exclusions the exclusions that were applied to the original list
         */
        AutoConfigurationEntry(Collection<String> configurations, Collection<String>
exclusions) {
            this.configurations = new ArrayList<>(configurations);
            this.exclusions = new HashSet<>(exclusions);
        }

        public List<String> getConfigurations() {
            return this.configurations;
        }

        public Set<String> getExclusions() {
            return this.exclusions;
        }

    }

}
```

从上面的分析来看看，该类其实就是收集spring.factories文件中以@EnableAutoConfiguration类型为key的所有的类，然后把这些类交给spring去实例化，而这些类就是我们说的aop、事务、缓存、mvc等功能的支持类，这就是自动配置的加载原理。很简单吧

2.5.3、自动配置包

前面我们分析了springboot自动配置原理，其实就是加载自动配置类的过程，那么我们接下来了解一下自动包

springboot中的自动配置包是：spring-boot-autoconfigure包，这个包就是自动配置包。如下图



```
Maven: org.springframework.boot:spring-boot-autoconfigure:2.3.2.RELEASE
    spring-boot-autoconfigure-2.3.2.RELEASE.jar library root
        META-INF
        org.springframework.boot.autoconfigure
```

我们再来看看该包下的spring.factories文件，内容如下

```
# 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.ConditionEvaluationReportAutoConfigurationI
mportListener

# 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
#核心就是这里，这里导入了 N多功能支持的类，这个地方就是自动配置的精髓
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.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration,\
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.CassandraReactiveRepositoriesAutoConfi
guration,\
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.CouchbaseReactiveRepositoriesAutoConfi
guration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfi
```

```
guration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRepositoriesA
utoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRestClientAut
oConfiguration,\
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.r2dbc.R2dbcDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.r2dbc.R2dbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.r2dbc.R2dbcTransactionManagerAutoConfiguration,\
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.ElasticsearchRestClientAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
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.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration
,\
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.r2dbc.R2dbcAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketRequesterAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketServerAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketStrategiesAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
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.ReactiveUserDetailsServiceAutoConfi
guration,\
org.springframework.boot.autoconfigure.security.rsocket.RSocketSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.saml2.Saml2RelyingPartyAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2ClientAutoConfig
uration,\
org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientA
utoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServer
AutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2Resou
rceServerAutoConfiguration,\
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.EmbeddedWebServerFactoryCustomizerAutoCo
nfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoC
onfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfigurati
on,\
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.WebServiceTemplateAutoConfiguratio
n

# Failure analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.autoconfigure.data.redis.RedisUrlSyntaxFailureAnalyzer,\
org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefinitionFailureAnaly
zer,\
org.springframework.boot.autoconfigure.flyway.FlywayMigrationScriptMissingFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailureAnalyzer,\
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFailureAnalyzer

# Template availability providers
org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider=\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.mustache.MustacheTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.web.servlet.JspTemplateAvailabilityProvider
```

2.5.4、AOP的自动配置

AOP功能的自动配置，aop自动配置类：**AopAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnProperty(prefix = "spring.aop", name = "auto", havingValue = "true",
matchIfMissing = true)
public class AopAutoConfiguration {

   @Configuration(proxyBeanMethods = false)
   @ConditionalOnClass(Advice.class)
   static class AspectJAutoProxyingConfiguration {

      @Configuration(proxyBeanMethods = false)
      @EnableAspectJAutoProxy(proxyTargetClass = false)
      @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue
= "false",
            matchIfMissing = false)
      static class JdkDynamicAutoProxyConfiguration {

      }

      @Configuration(proxyBeanMethods = false)
      @EnableAspectJAutoProxy(proxyTargetClass = true)
      @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue
= "true",
            matchIfMissing = true)
      static class CglibAutoProxyConfiguration {

      }

   }

   @Configuration(proxyBeanMethods = false)
   @ConditionalOnMissingClass("org.aspectj.weaver.Advice")
   @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue =
"true",
         matchIfMissing = true)
   static class ClassProxyingConfiguration {

      ClassProxyingConfiguration(BeanFactory beanFactory) {
         if (beanFactory instanceof BeanDefinitionRegistry) {
            BeanDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory;
            AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
            AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
         }
      }

   }

}
```

2.5.5、数据源的自动配置

数据源的自动配置类：**DataSourceAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@ConditionalOnMissingBean(type = "io.r2dbc.spi.ConnectionFactory")
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

   @Configuration(proxyBeanMethods = false)
   @Conditional(EmbeddedDatabaseCondition.class)
   @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
```

```
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedDatabaseConfiguration {

    }

    @Configuration(proxyBeanMethods = false)
    @Conditional(PooledDataSourceCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
            DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.Generic.class,
            DataSourceJmxConfiguration.class })
    protected static class PooledDataSourceConfiguration {

    }

    /**
     * {@link AnyNestedCondition} that checks that either {@code spring.datasource.type}
     * is set or {@link PooledDataSourceAvailableCondition} applies.
     */
    static class PooledDataSourceCondition extends AnyNestedCondition {

        PooledDataSourceCondition() {
            super(ConfigurationPhase.PARSE_CONFIGURATION);
        }

        @ConditionalOnProperty(prefix = "spring.datasource", name = "type")
        static class ExplicitType {

        }

        @Conditional(PooledDataSourceAvailableCondition.class)
        static class PooledDataSourceAvailable {

        }

    }

    /**
     * {@link Condition} to test if a supported connection pool is available.
     */
    static class PooledDataSourceAvailableCondition extends SpringBootCondition {

        @Override
        public ConditionOutcome getMatchOutcome(ConditionContext context,
AnnotatedTypeMetadata metadata) {
            ConditionMessage.Builder message =
ConditionMessage.forCondition("PooledDataSource");
            if (DataSourceBuilder.findType(context.getClassLoader()) != null) {
                return ConditionOutcome.match(message.foundExactly("supported DataSource"));
            }
            return ConditionOutcome.noMatch(message.didNotFind("supported
DataSource").atAll());
        }
    }

    /**
     * {@link Condition} to detect when an embedded {@link DataSource} type can be used.
     * If a pooled {@link DataSource} is available, it will always be preferred to an
     * {@code EmbeddedDatabase}.
     */
    static class EmbeddedDatabaseCondition extends SpringBootCondition {

        private static final String DATASOURCE_URL_PROPERTY = "spring.datasource.url";

        private final SpringBootCondition pooledCondition = new PooledDataSourceCondition();

        @Override
        public ConditionOutcome getMatchOutcome(ConditionContext context,
AnnotatedTypeMetadata metadata) {
            ConditionMessage.Builder message =
ConditionMessage.forCondition("EmbeddedDataSource");
            if (hasDataSourceUrlProperty(context)) {
                return ConditionOutcome.noMatch(message.because(DATASOURCE_URL_PROPERTY + " is
set"));
            }
            if (anyMatches(context, metadata, this.pooledCondition)) {
                return ConditionOutcome.noMatch(message.foundExactly("supported pooled data
source"));
            }
            EmbeddedDatabaseType type =
EmbeddedDatabaseConnection.get(context.getClassLoader()).getType();
            if (type == null) {
                return ConditionOutcome.noMatch(message.didNotFind("embedded
database").atAll());
            }
            return ConditionOutcome.match(message.found("embedded database").items(type));
        }

        private boolean hasDataSourceUrlProperty(ConditionContext context) {
            Environment environment = context.getEnvironment();
            if (environment.containsProperty(DATASOURCE_URL_PROPERTY)) {
                try {
                    return StringUtils.hasText(environment.getProperty(DATASOURCE_URL_PROPERTY));
                }
                catch (IllegalArgumentException ex) {
                    // Ignore unresolvable placeholder errors
                }
            }
            return false;
        }

    }

}
```

默认数据源

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(HikariDataSource.class)
@ConditionalOnMissingBean(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.type", havingValue =
"com.zaxxer.hikari.HikariDataSource",
      matchIfMissing = true)
static class Hikari {

   @Bean
   @ConfigurationProperties(prefix = "spring.datasource.hikari")
   HikariDataSource dataSource(DataSourceProperties properties) {
      HikariDataSource dataSource = createDataSource(properties, HikariDataSource.class);
      if (StringUtils.hasText(properties.getName())) {
         dataSource.setPoolName(properties.getName());
      }
      return dataSource;
   }

}
```

2.5.6、事务的自动配置

事务管理器自动配置类：**DataSourceTransactionManagerAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ JdbcTemplate.class, PlatformTransactionManager.class })
@AutoConfigureOrder(Ordered.LOWEST_PRECEDENCE)
@EnableConfigurationProperties(DataSourceProperties.class)
public class DataSourceTransactionManagerAutoConfiguration {

   @Configuration(proxyBeanMethods = false)
   @ConditionalOnSingleCandidate(DataSource.class)
   static class DataSourceTransactionManagerConfiguration {

      @Bean
      @ConditionalOnMissingBean(PlatformTransactionManager.class)
      DataSourceTransactionManager transactionManager(DataSource dataSource,
            ObjectProvider<TransactionManagerCustomizers> transactionManagerCustomizers) {
         DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager(dataSource);
         transactionManagerCustomizers.ifAvailable((customizers) ->
customizers.customize(transactionManager));
         return transactionManager;
      }

   }

}
```

事务配置类：**TransactionAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(PlatformTransactionManager.class)
@AutoConfigureAfter({ JtaAutoConfiguration.class, HibernateJpaAutoConfiguration.class,
      DataSourceTransactionManagerAutoConfiguration.class, Neo4jDataAutoConfiguration.class
})
@EnableConfigurationProperties(TransactionProperties.class)
public class TransactionAutoConfiguration {

   @Bean
   @ConditionalOnMissingBean
   public TransactionManagerCustomizers platformTransactionManagerCustomizers(
         ObjectProvider<PlatformTransactionManagerCustomizer<?>> customizers) {
      return new
TransactionManagerCustomizers(customizers.orderedStream().collect(Collectors.toList()));
   }

   @Bean
   @ConditionalOnMissingBean
   @ConditionalOnSingleCandidate(ReactiveTransactionManager.class)
   public TransactionalOperator transactionalOperator(ReactiveTransactionManager
transactionManager) {
      return TransactionalOperator.create(transactionManager);
   }

   @Configuration(proxyBeanMethods = false)
   @ConditionalOnSingleCandidate(PlatformTransactionManager.class)
   public static class TransactionTemplateConfiguration {

      @Bean
      @ConditionalOnMissingBean(TransactionOperations.class)
      public TransactionTemplate transactionTemplate(PlatformTransactionManager
transactionManager) {
         return new TransactionTemplate(transactionManager);
      }

   }

   @Configuration(proxyBeanMethods = false)
   @ConditionalOnBean(TransactionManager.class)
   @ConditionalOnMissingBean(AbstractTransactionManagementConfiguration.class)
   public static class EnableTransactionManagementConfiguration {

      @Configuration(proxyBeanMethods = false)
      @EnableTransactionManagement(proxyTargetClass = false)
      @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue
= "false",
            matchIfMissing = false)
      public static class JdkDynamicAutoProxyConfiguration {

      }

      @Configuration(proxyBeanMethods = false)
      @EnableTransactionManagement(proxyTargetClass = true)
      @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue
= "true",
            matchIfMissing = true)
      public static class CglibAutoProxyConfiguration {

      }

   }

}
```

2.5.7、mvc的自动配置

DispatcherServlet自动配置：**DispatcherServletAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
@Conditional(DefaultDispatcherServletCondition.class)
@ConditionalOnClass(ServletRegistration.class)
@EnableConfigurationProperties(WebMvcProperties.class)
protected static class DispatcherServletConfiguration {

   @Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
   public DispatcherServlet dispatcherServlet(WebMvcProperties webMvcProperties) {
      DispatcherServlet dispatcherServlet = new DispatcherServlet();

dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());
      dispatcherServlet.setDispatchTraceRequest(webMvcProperties.isDispatchTraceRequest());

dispatcherServlet.setThrowExceptionIfNoHandlerFound(webMvcProperties.isThrowExceptionIfNoHan
dlerFound());
      dispatcherServlet.setPublishEvents(webMvcProperties.isPublishRequestHandledEvents());

dispatcherServlet.setEnableLoggingRequestDetails(webMvcProperties.isLogRequestDetails());
      return dispatcherServlet;
   }

   @Bean
   @ConditionalOnBean(MultipartResolver.class)
   @ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
   public MultipartResolver multipartResolver(MultipartResolver resolver) {
      // Detect if the user has created a MultipartResolver but named it incorrectly
      return resolver;
   }

}
```

mvc的自动配置：**WebMvcAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration
implements ResourceLoaderAware {
   private final ResourceProperties resourceProperties;

   private final WebMvcProperties mvcProperties;

   private final ListableBeanFactory beanFactory;

   private final WebMvcRegistrations mvcRegistrations;

   private ResourceLoader resourceLoader;

   public EnableWebMvcConfiguration(ResourceProperties resourceProperties,
         ObjectProvider<WebMvcProperties> mvcPropertiesProvider,
         ObjectProvider<WebMvcRegistrations> mvcRegistrationsProvider, ListableBeanFactory
beanFactory) {
      this.resourceProperties = resourceProperties;
      this.mvcProperties = mvcPropertiesProvider.getIfAvailable();
      this.mvcRegistrations = mvcRegistrationsProvider.getIfUnique();
      this.beanFactory = beanFactory;
   }

   @Bean
   @Override
   public RequestMappingHandlerAdapter requestMappingHandlerAdapter(
         @Qualifier("mvcContentNegotiationManager") ContentNegotiationManager
contentNegotiationManager,
         @Qualifier("mvcConversionService") FormattingConversionService conversionService,
         @Qualifier("mvcValidator") Validator validator) {
      RequestMappingHandlerAdapter adapter =
super.requestMappingHandlerAdapter(contentNegotiationManager,
            conversionService, validator);
      adapter.setIgnoreDefaultModelOnRedirect(
            this.mvcProperties == null ||
this.mvcProperties.isIgnoreDefaultModelOnRedirect());
      return adapter;
   }

   @Override
   protected RequestMappingHandlerAdapter createRequestMappingHandlerAdapter() {
      if (this.mvcRegistrations != null &&
this.mvcRegistrations.getRequestMappingHandlerAdapter() != null) {
         return this.mvcRegistrations.getRequestMappingHandlerAdapter();
      }
      return super.createRequestMappingHandlerAdapter();
   }

   @Bean
   @Primary
   @Override
   public RequestMappingHandlerMapping requestMappingHandlerMapping(
         @Qualifier("mvcContentNegotiationManager") ContentNegotiationManager
contentNegotiationManager,
         @Qualifier("mvcConversionService") FormattingConversionService conversionService,
         @Qualifier("mvcResourceUrlProvider") ResourceUrlProvider resourceUrlProvider) {
```

```
      // Must be @Primary for MvcUriComponentsBuilder to work
      return super.requestMappingHandlerMapping(contentNegotiationManager,
conversionService,
            resourceUrlProvider);
   }

   @Bean
   public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext
applicationContext,
         FormattingConversionService mvcConversionService, ResourceUrlProvider
mvcResourceUrlProvider) {
      WelcomePageHandlerMapping welcomePageHandlerMapping = new WelcomePageHandlerMapping(
            new TemplateAvailabilityProviders(applicationContext), applicationContext,
getWelcomePage(),
            this.mvcProperties.getStaticPathPattern());
      welcomePageHandlerMapping.setInterceptors(getInterceptors(mvcConversionService,
mvcResourceUrlProvider));
      welcomePageHandlerMapping.setCorsConfigurations(getCorsConfigurations());
      return welcomePageHandlerMapping;
   }

   private Optional<Resource> getWelcomePage() {
      String[] locations =
getResourceLocations(this.resourceProperties.getStaticLocations());
      return
Arrays.stream(locations).map(this::getIndexHtml).filter(this::isReadable).findFirst();
   }

   private Resource getIndexHtml(String location) {
      return this.resourceLoader.getResource(location + "index.html");
   }

   private boolean isReadable(Resource resource) {
      try {
         return resource.exists() && (resource.getURL() != null);
      }
      catch (Exception ex) {
         return false;
      }
   }

   @Bean
   @Override
   public FormattingConversionService mvcConversionService() {
      Format format = this.mvcProperties.getFormat();
      WebConversionService conversionService = new WebConversionService(new
DateTimeFormatters()

.dateFormat(format.getDate()).timeFormat(format.getTime()).dateTimeFormat(format.getDateTime
()));
      addFormatters(conversionService);
      return conversionService;
   }
   @Bean
   @Override
   public Validator mvcValidator() {
      if (!ClassUtils.isPresent("javax.validation.Validator", getClass().getClassLoader()))
{
         return super.mvcValidator();
      }
      return ValidatorAdapter.get(getApplicationContext(), getValidator());
   }

   @Override
   protected RequestMappingHandlerMapping createRequestMappingHandlerMapping() {
      if (this.mvcRegistrations != null &&
this.mvcRegistrations.getRequestMappingHandlerMapping() != null) {
         return this.mvcRegistrations.getRequestMappingHandlerMapping();
      }
      return super.createRequestMappingHandlerMapping();
   }

   @Override
   protected ConfigurableWebBindingInitializer getConfigurableWebBindingInitializer(
         FormattingConversionService mvcConversionService, Validator mvcValidator) {
      try {
         return this.beanFactory.getBean(ConfigurableWebBindingInitializer.class);
      }
      catch (NoSuchBeanDefinitionException ex) {
         return super.getConfigurableWebBindingInitializer(mvcConversionService,
mvcValidator);
      }
   }

   @Override
   protected ExceptionHandlerExceptionResolver createExceptionHandlerExceptionResolver() {
      if (this.mvcRegistrations != null &&
this.mvcRegistrations.getExceptionHandlerExceptionResolver() != null) {
         return this.mvcRegistrations.getExceptionHandlerExceptionResolver();
      }
      return super.createExceptionHandlerExceptionResolver();
   }

   @Override
   protected void extendHandlerExceptionResolvers(List<HandlerExceptionResolver>
exceptionResolvers) {
      super.extendHandlerExceptionResolvers(exceptionResolvers);
      if (this.mvcProperties.isLogResolvedException()) {
         for (HandlerExceptionResolver resolver : exceptionResolvers) {
            if (resolver instanceof AbstractHandlerExceptionResolver) {
               ((AbstractHandlerExceptionResolver)
resolver).setWarnLogCategory(resolver.getClass().getName());
            }
         }
      }
```

```
    }

    @Bean
    @Override
    public ContentNegotiationManager mvcContentNegotiationManager() {
        ContentNegotiationManager manager = super.mvcContentNegotiationManager();
        List<ContentNegotiationStrategy> strategies = manager.getStrategies();
        ListIterator<ContentNegotiationStrategy> iterator = strategies.listIterator();
        while (iterator.hasNext()) {
            ContentNegotiationStrategy strategy = iterator.next();
            if (strategy instanceof PathExtensionContentNegotiationStrategy) {
                iterator.set(new OptionalPathExtensionContentNegotiationStrategy(strategy));
            }
        }
        return manager;
    }

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }

}
```