Spring Boot 运行机制源码剖析

一、SpringBoot入口程序

SpringApplication运行

```
@SpringBootApplication
public class Application {

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

自定义SpringApplication

1. 通过SpringApplicationAPI调整

```
SpringApplication.run(Application.class, args)
SpringApplication springApplication = new SpringApplication(Application.class);
springApplication.setBannerMode(Banner.Mode.CONSOLE);
springApplication.setWebApplicationType(WebApplicationType.NONE);
springApplication.setAdditionalProfiles("prod");
springApplication.setHeadless(true);
```

2. 通过SpringApplicationBuilderAPI 调整

```
new SpringApplicationBuilder(Application.class)
    .bannerMode(Banner.Mode.CONSOLE)
    .web(WebApplicationType.NONE)
    .profiles("prod")
    .headless(true) .run(args);
```

二、SpringBoot准备阶段

当程序开始执行之后，会调用SpringApplication的构造方法，进行某些初始参数的设置

```
//创建一个新的实例，这个应用程序的上下文将要从指定的来源加载Bean
public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
    //资源初始化资源加载器，默认为null
    this.resourceLoader = resourceLoader;
    //断言主要加载资源类不能为 null，否则报错
    Assert.notNull(primarySources, "PrimarySources must not be null");
    //初始化主要加载资源类集合并去重
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
```

```
    //推断当前 WEB 应用类型，一共有三种：NONE,SERVLET,REACTIVE
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    //设置应用上线文初始化器,从"META-INF/spring.factories"读取
ApplicationContextInitializer类的实例名称集合并去重，并进行set去重。（一共7个）
    setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
    //设置监听器,从"META-INF/spring.factories"读取ApplicationListener类的实例名称集合
并去重，并进行set去重。（一共11个）
    setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
    //推断主入口应用类，通过当前调用栈，获取Main方法所在类，并赋值给mainApplicationClass
    this.mainApplicationClass = deduceMainApplicationClass();
    }
```

配置Spring Boot Bean源

Java 配置 Class 或 XML 上下文配置文件集合，用于 Spring Boot `BeanDefinitionLoader` 读取，并且将配置源解析加载为Spring Bean 定义

- 数量：一个或多个以上

```
//加载关系
SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources)
   -------> ConfigurableApplicationContext run(String... args)
        -------> prepareContext(ConfigurableApplicationContext context)
            ------->load(context, sources.toArray(new Object[0]));

protected void load(ApplicationContext context, Object[] sources) {
   if (logger.isDebugEnabled()) {
      logger.debug(
           "Loading source " +
StringUtils.arrayToCommaDelimitedString(sources));
   }
   BeanDefinitionLoader loader = createBeanDefinitionLoader(
         getBeanDefinitionRegistry(context), sources);
   if (this.beanNameGenerator != null) {
      loader.setBeanNameGenerator(this.beanNameGenerator);
   }
   if (this.resourceLoader != null) {
      loader.setResourceLoader(this.resourceLoader);
   }
   if (this.environment != null) {
      loader.setEnvironment(this.environment);
   }
   loader.load();
}
```

Java配置Class

用于 Spring 注解驱动中 Java 配置类，大多数情况是 Spring 模式注解所标注的类，如 @Configuration 。

XML上下文配置文件

用于 Spring 传统配置驱动中的 XML 文件

判断当前应用类型

在上述构造方法中，有一个判断应用类型的方法，用来判断当前应用程序的类型：

- Web Reactive：`WebApplicationType.REACTIVE`
- Web Servlet：`WebApplicationType.SERVLET`
- 非 Web：`WebApplicationType.NONE`

```
    static WebApplicationType deduceFromClasspath() {
        //如果存在reactive环境类，并且不存在Servlet环境类(jersey、mvc)，则推断当前环境为
REACTIVE
        if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, null) &&
!ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS, null)
                && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, null)) {
            return WebApplicationType.REACTIVE;
        }
        //不存在Servlet相关类，则推断当前环境为NONE
        for (String className : SERVLET_INDICATOR_CLASSES) {
            if (!ClassUtils.isPresent(className, null)) {
                return WebApplicationType.NONE;
            }
        }
        //默认推断当前环境为SERVLET
        return WebApplicationType.SERVLET;
    }
//WebApplicationType的类型
public enum WebApplicationType {

    /**
     * The application should not run as a web application and should not start
an
     * embedded web server.
     * 非web项目
     */
    NONE,

    /**
     * The application should run as a servlet-based web application and should
start an
     * embedded servlet web server.
     * servlet web 项目
     */
    SERVLET,

    /**
     * The application should run as a reactive web application and should start
an
     * embedded reactive web server.
     * 响应式 web 项目
     */
    REACTIVE;
```

加载应用上下文初始器 （ApplicationContextInitializer）

利用 Spring 工厂加载机制，实例化 ApplicationContextInitializer 实现类，并排序对象集合。

- 实现

```
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type, Class<?>[]
parameterTypes, Object... args) {
        /**
         * 利用 Spring 工厂加载机制，实例化type相关类，并排序对象集合。
         */
        ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
        // Use names and ensure unique to protect against duplicates Set<String>
names = new LinkedHashSet<>(
        SpringFactoriesLoader.loadFactoryNames(type, classLoader));
        List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
                classLoader, args, names);
        AnnotationAwareOrderComparator.sort(instances);
        return instances;
    }
```

- 技术

  实现类： org.springframework.core.io.support.SpringFactoriesLoader

  配置资源： META-INF/spring.factories

  排序： AnnotationAwareOrderComparator#sort

加载应用事件监听器（ApplicationListener）

利用 Spring 工厂加载机制，实例化 ApplicationListener 实现类，并排序对象集合

找到程序运行的主类（Main Class）

根据 Main 线程执行抛异常操作，获取堆栈判断实际的引导类

参考方法： `org.springframework.boot.SpringApplication#deduceMainApplicationClass`

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

```
    }
    return null;
}
```

三、SpringBoot运行阶段

springboot启动的run方法，可以看到主要是各种运行环境的准备工作

```
public ConfigurableApplicationContext run(String... args) {
    //1、创建并启动计时监控类
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    //2、初始化应用上下文和异常报告集合
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>
();
    //3、设置系统属性“java.awt.headless”的值，默认为true，用于运行headless服务器，进行简单的图像处理，多用于在缺少显示屏、键盘或者鼠标时的系统配置，很多监控工具如jconsole 需要将该值设置为true
    configureHeadlessProperty();
    //4、创建所有spring运行监听器并发布应用启动事件，简单说的话就是获取SpringApplicationRunListener类型的实例（EventPublishingRunListener对象），并封装进SpringApplicationRunListeners对象，然后返回这个SpringApplicationRunListeners对象。说的再简单点，getRunListeners就是准备好了运行时监听器EventPublishingRunListener。
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        //5、初始化默认应用参数类
        ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
        //6、根据运行监听器和应用参数来准备spring环境
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
        //将要忽略的bean的参数打开
        configureIgnoreBeanInfo(environment);
        //7、创建banner打印类
        Banner printedBanner = printBanner(environment);
        //8、创建应用上下文，可以理解为创建一个容器
        context = createApplicationContext();
        //9、准备异常报告器，用来支持报告关于启动的错误
        exceptionReporters =
getSpringFactoriesInstances(SpringBootExceptionReporter.class,
                new Class[] { ConfigurableApplicationContext.class },
context);
        //10、准备应用上下文，该步骤包含一个非常关键的操作，将启动类注入容器，为后续开启自动化提供基础
        prepareContext(context, environment, listeners, applicationArguments,
printedBanner);
        //11、刷新应用上下文
        refreshContext(context);
        //12、应用上下文刷新后置处理，做一些扩展功能
        afterRefresh(context, applicationArguments);
        //13、停止计时监控类
        stopWatch.stop();
```

```
        //14、输出日志记录执行主类名、时间信息
        if (this.logStartupInfo) {
                new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopWatch);
        }
        //15、发布应用上下文启动监听事件
        listeners.started(context);
        //16、执行所有的Runner运行器
        callRunners(context, applicationArguments);
    }catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, listeners);
        throw new IllegalStateException(ex);
    }
    try {
        //17、发布应用上下文就绪事件
        listeners.running(context);
    }catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, null);
        throw new IllegalStateException(ex);
    }
    //18、返回应用上下文
    return context;
}
```

下面详细介绍各个启动的环节：

1、创建并启动计时监控类

创建并启动计时监控类，可以看到记录当前任务的名称，默认是空字符串，然后记录当前springboot应用启动的开始时间。

```
StopWatch stopWatch = new StopWatch();
stopWatch.start();
//详细源代码
public void start() throws IllegalStateException {
    start("");
}
public void start(String taskName) throws IllegalStateException {
    if (this.currentTaskName != null) {
        throw new IllegalStateException("Can't start StopWatch: it's already
running");
    }
    this.currentTaskName = taskName;
    this.startTimeNanos = System.nanoTime();
}
```

2、初始化应用上下文和异常报告集合

```
ConfigurableApplicationContext context = null;
Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
```

3、设置系统属性java.awt.headless的值

```
/*
java.awt.headless模式是在缺少显示屏、键盘或者鼠标的系统配置
当配置了如下属性之后，应用程序可以执行如下操作：
    1、创建轻量级组件
    2、收集关于可用的字体、字体指标和字体设置的信息
    3、设置颜色来渲染准备图片
    4、创造和获取图像，为渲染准备图片
    5、使用java.awt.PrintJob,java.awt.print.*和javax.print.*类里的方法进行打印
*/
private void configureHeadlessProperty() {
        System.setProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS,
                System.getProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS,
Boolean.toString(this.headless)));
}
```

4、创建所有spring运行监听器并发布应用启动事件

加载SpringApplication 运行监听器（ SpringApplicationRunListeners)

利用 Spring 工厂加载机制，读取 SpringApplicationRunListener 对象集合，并且封装到组合类 SpringApplicationRunListeners

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

//创建spring监听器
private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class
};
    return new SpringApplicationRunListeners(logger,
                getSpringFactoriesInstances(SpringApplicationRunListener.class,
types, this, args));
}
SpringApplicationRunListeners(Log log, Collection<? extends
SpringApplicationRunListener> listeners) {
    this.log = log;
    this.listeners = new ArrayList<>(listeners);
}
//循环遍历获取监听器
void starting() {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.starting();
    }
}
//此处的监听器可以看出是事件发布监听器，主要用来发布启动事件
@Override
public void starting() {
    //这里是创建application事件‘applicationStartingEvent’
    this.initialMulticaster.multicastEvent(new
ApplicationStartingEvent(this.application, this.args));
}
//applicationStartingEvent是springboot框架最早执行的监听器，在该监听器执行started方法
时，会继续发布事件，主要是基于spring的事件机制
    @Override
    public void multicastEvent(final ApplicationEvent event, @Nullable
ResolvableType eventType) {
```

```
        ResolvableType type = (eventType != null ? eventType :
resolveDefaultEventType(event));
        //获取线程池，如果为空则同步处理。这里线程池为空，还未初始化
        Executor executor = getTaskExecutor();
        for (ApplicationListener<?> listener : getApplicationListeners(event,
type)) {
            if (executor != null) {
                //异步发送事件
                executor.execute(() -> invokeListener(listener, event));
            }
            else {
                //同步发送事件
                invokeListener(listener, event);
            }
        }
    }
```

运行 SpringApplication运行监听器（SpringApplicationRunListeners）

SpringApplicationRunListener 监听多个运行状态方法：

监听方法	阶段说明	SpringBoot 起始版本
starting()	Spring 应用刚启动	1.0
environmentPrepared(ConfigurableEnvironment)	ConfigurableEnvironment 准备妥当，允许将其调整	1.0
contextPrepared(ConfigurableApplicationContext)	ConfigurableApplicationContext 准备妥当，允许将其调整	1.0
contextLoaded(ConfigurableApplicationContext)	ConfigurableApplicationContext 已装载，但仍未启动	1.0
started(ConfigurableApplicationContext)	ConfigurableApplicationContext 已启动，此时 Spring Bean 已初始化完成	2.0
running(ConfigurableApplicationContext)	Spring 应用正在运行	2.0
failed(ConfigurableApplicationContext,Throwable)	Spring 应用运行失败	2.0

监听 Spring Boot事件/ Spring 事件

Spring Boot 通过 `SpringApplicationRunListener` 的实现类 `EventPublishingRunListener`

利用 Spring Framework 事件API，广播 Spring Boot 事件。

Spring Framework事件/监听器编程模型

- Spring 应用事件

  普通应用事件： ApplicationEvent
  - 应用上下文事件： ApplicationContextEvent
- Spring 应用监听器
  - 接口编程模型： ApplicationListener
  - 注解编程模型： @EventListener
- Spring 应用事件广播器
  - 接口： ApplicationEventMulticaster

- 实现类： SimpleApplicationEventMulticaster
  - 执行模式：同步或异步

EventPublishingRunListener监听方法与 Spring Boot 事件对应关系

监听方法	Spring Boot事件	Spring Boot起始版本
starting()	ApplicationStartingEvent	1.5
environmentPrepared(ConfigurableEnvironment)	ApplicationEnvironmentPreparedEvent	1.0
contextPrepared(ConfigurableApplicationContext)		
contextLoaded(ConfigurableApplicationContext)	ApplicationPreparedEvent	1.0
started(ConfigurableApplicationContext)	ApplicationStartedEvent	2.0
running(ConfigurableApplicationContext)	ApplicationReadyEvent	2.0
failed(ConfigurableApplicationContext,Throwable)	ApplicationFailedEvent	1.0

5、初始化默认应用参数类

```
ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
public DefaultApplicationArguments(String... args) {
    Assert.notNull(args, "Args must not be null");
    this.source = new Source(args);
    this.args = args;
}
```

6、根据运行监听器和应用参数来准备spring环境

创建Environment

根据准备阶段的推断 Web 应用类型创建对应的 ConfigurableEnvironment 实例：

Web Reactive： `StandardEn vironment`

Web Servlet： `StandardServletEnvironment`

非 Web： `StandardEnvironment`

```
ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
//详细环境的准备
private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners
listeners,
    ApplicationArguments applicationArguments) {
    // 获取或者创建应用环境
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    // 配置应用环境，配置propertySource和activeProfiles
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    //listeners环境准备，广播ApplicationEnvironmentPreparedEvent
    ConfigurationPropertySources.attach(environment);
    listeners.environmentPrepared(environment);
```

```
    //将环境绑定给当前应用程序
    bindToSpringApplication(environment);
    //对当前的环境类型进行判断，如果不一致进行转换
    if (!this.isCustomEnvironment) {
        environment = new
EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment
,
                    deduceEnvironmentClass());
    }
    //配置propertySource对它自己的递归依赖
    ConfigurationPropertySources.attach(environment);
    return environment;
}
// 获取或者创建应用环境，根据应用程序的类型可以分为servlet环境、标准环境(特殊的非web环境)和响
应式环境
private ConfigurableEnvironment getOrCreateEnvironment() {
    //存在则直接返回
        if (this.environment != null) {
            return this.environment;
        }
    //根据webApplicationType创建对应的Environment
        switch (this.webApplicationType) {
        case SERVLET:
            return new StandardServletEnvironment();
        case REACTIVE:
            return new StandardReactiveWebEnvironment();
        default:
            return new StandardEnvironment();
        }
    }
//配置应用环境
protected void configureEnvironment(ConfigurableEnvironment environment,
String[] args) {
    if (this.addConversionService) {
        ConversionService conversionService =
ApplicationConversionService.getSharedInstance();
        environment.setConversionService((ConfigurableConversionService)
conversionService);
    }
    //配置property sources
    configurePropertySources(environment, args);
    //配置profiles
    configureProfiles(environment, args);
}
```

7、创建banner的打印类

```
Banner printedBanner = printBanner(environment);
//打印类的详细操作过程
private Banner printBanner(ConfigurableEnvironment environment) {
    if (this.bannerMode == Banner.Mode.OFF) {
        return null;
    }
    ResourceLoader resourceLoader = (this.resourceLoader != null) ?
this.resourceLoader
```

```
        : new DefaultResourceLoader(getClassLoader());
    SpringApplicationBannerPrinter bannerPrinter = new
SpringApplicationBannerPrinter(resourceLoader, this.banner);
    if (this.bannerMode == Mode.LOG) {
        return bannerPrinter.print(environment, this.mainApplicationClass,
logger);
    }
    return bannerPrinter.print(environment, this.mainApplicationClass,
System.out);
}
```

8、创建应用的上下文:根据不同应用类型初始化不同的上下文应用类

创建Spring 应用上下文（ ConfigurableApplicationContext)

根据准备阶段的推断 Web 应用类型创建对应的 `ConfigurableApplicationContext` 实例：

- Web Reactive： `AnnotationConfigReactiveWebServerApplicationContext`
- Web Servlet： `AnnotationConfigServletWebServerApplicationContext`
- 非 Web： `AnnotationConfigApplicationContext`

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

9、准备异常报告器

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

10、准备应用上下文

```
prepareContext(context, environment, listeners, applicationArguments,
printedBanner);

private void prepareContext(ConfigurableApplicationContext context,
ConfigurableEnvironment environment,
            SpringApplicationRunListeners listeners, ApplicationArguments
applicationArguments, Banner printedBanner) {
        //应用上下文的environment
        context.setEnvironment(environment);
        //应用上下文后处理
        postProcessApplicationContext(context);
        //为上下文应用所有初始化器，执行容器中的
applicationContextInitializer(spring.factories的实例)，将所有的初始化对象放置到
context对象中
        applyInitializers(context);
        //触发所有SpringApplicationRunListener监听器的ContextPrepared事件方法。添加所
有的事件监听器
        listeners.contextPrepared(context);
        //记录启动日志
        if (this.logStartupInfo) {
            logStartupInfo(context.getParent() == null);
            logStartupProfileInfo(context);
        }
        // 注册启动参数bean，将容器指定的参数封装成bean，注入容器
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        beanFactory.registerSingleton("springApplicationArguments",
applicationArguments);
        //设置banner
        if (printedBanner != null) {
            beanFactory.registerSingleton("springBootBanner", printedBanner);
        }
        if (beanFactory instanceof DefaultListableBeanFactory) {
            ((DefaultListableBeanFactory) beanFactory)

.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
        }
        if (this.lazyInitialization) {
```

```
            context.addBeanFactoryPostProcessor(new
LazyInitializationBeanFactoryPostProcessor());
        }
        // 加载所有资源，指的是启动器指定的参数
        Set<Object> sources = getAllSources();
        Assert.notEmpty(sources, "Sources must not be empty");
        //将bean加载到上下文中
        load(context, sources.toArray(new Object[0]));
        //触发所有springapplicationRunListener监听器的contextLoaded事件方法，
        listeners.contextLoaded(context);
    }
--------------------
    //这里没有做任何的处理过程，因为beanNameGenerator和resourceLoader默认为空，可以方便后
续做扩展处理
    protected void postProcessApplicationContext(ConfigurableApplicationContext
context) {
        if (this.beanNameGenerator != null) {

context.getBeanFactory().registerSingleton(AnnotationConfigUtils.CONFIGURATION_B
EAN_NAME_GENERATOR,
                    this.beanNameGenerator);
        }
        if (this.resourceLoader != null) {
            if (context instanceof GenericApplicationContext) {
                ((GenericApplicationContext)
context).setResourceLoader(this.resourceLoader);
            }
            if (context instanceof DefaultResourceLoader) {
                ((DefaultResourceLoader)
context).setClassLoader(this.resourceLoader.getClassLoader());
            }
        }
        if (this.addConversionService) {

context.getBeanFactory().setConversionService(ApplicationConversionService.getSh
aredInstance());
        }
    }
---------------------
    //将启动器类加载到spring容器中，为后续的自动化配置奠定基础，之前看到的很多注解也与此相关
    protected void load(ApplicationContext context, Object[] sources) {
        if (logger.isDebugEnabled()) {
            logger.debug("Loading source " +
StringUtils.arrayToCommaDelimitedString(sources));
        }
        BeanDefinitionLoader loader =
createBeanDefinitionLoader(getBeanDefinitionRegistry(context), sources);
        if (this.beanNameGenerator != null) {
            loader.setBeanNameGenerator(this.beanNameGenerator);
        }
        if (this.resourceLoader != null) {
            loader.setResourceLoader(this.resourceLoader);
        }
        if (this.environment != null) {
            loader.setEnvironment(this.environment);
        }
        loader.load();
    }
```

```
--------------------
    //springboot会优先选择groovy加载方式，找不到在选择java方式
    private int load(Class<?> source) {
        if (isGroovyPresent() &&
GroovyBeanDefinitionSource.class.isAssignableFrom(source)) {
            // Any GroovyLoaders added in beans{} DSL can contribute beans here
            GroovyBeanDefinitionSource loader =
BeanUtils.instantiateClass(source, GroovyBeanDefinitionSource.class);
            load(loader);
        }
        if (isComponent(source)) {
            this.annotatedReader.register(source);
            return 1;
        }
        return 0;
    }
```

11、刷新应用上下文

```
refreshContext(context);

private void refreshContext(ConfigurableApplicationContext context) {
        refresh(context);
        if (this.registerShutdownHook) {
            try {
                context.registerShutdownHook();
            }
            catch (AccessControlException ex) {
                // Not allowed in some environments.
            }
        }
    }

------------
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            //刷新上下文环境，初始化上下文环境，对系统的环境变量或者系统属性进行准备和校验
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            //初始化beanfactory，解析xml，相当于之前的xmlBeanfactory操作
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            //为上下文准备beanfactory，对beanFactory的各种功能进行填充，如@autowired，
设置spel表达式解析器，设置编辑注册器，添加applicationContextAwareprocessor处理器等等
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context
subclasses.
                //提供子类覆盖的额外处理，即子类处理自定义的beanfactorypostProcess
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
```

```
                //激活各种beanfactory处理器
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                //注册拦截bean创建的bean处理器，即注册beanPostProcessor
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                //初始化上下文中的资源文件如国际化文件的处理
                initMessageSource();

                // Initialize event multicaster for this context.
                //初始化上下文事件广播器
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context
subclasses.
                //给子类扩展初始化其他bean
                onRefresh();

                // Check for listener beans and register them.
                //在所有的bean中查找listener bean,然后 注册到广播器中
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                //初始化剩余的非懒惰的bean，即初始化非延迟加载的bean
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                //发完成刷新过程，通知声明周期处理器刷新过程，同时发出ContextRefreshEvent
通知别人
                finishRefresh();
            }

            catch (BeansException ex) {
                if (logger.isWarnEnabled()) {
                    logger.warn("Exception encountered during context
initialization - " +
                            "cancelling refresh attempt: " + ex);
                }

                // Destroy already created singletons to avoid dangling
resources.
                destroyBeans();

                // Reset 'active' flag.
                cancelRefresh(ex);

                // Propagate exception to caller.
                throw ex;
            }

            finally {
                // Reset common introspection caches in Spring's core, since we
                // might not ever need metadata for singleton beans anymore...
                resetCommonCaches();
            }
        }
```

```
    }
```

12、应用上下文刷新后置处理

```
afterRefresh(context, applicationArguments);
//当前方法的代码是空的，可以做一些自定义的后置处理操作
protected void afterRefresh(ConfigurableApplicationContext context,
ApplicationArguments args) {
    }
```

13、停止计时监控类：计时监听器停止，并统计一些任务执行信息

```
stopWatch.stop();
public void stop() throws IllegalStateException {
        if (this.currentTaskName == null) {
            throw new IllegalStateException("Can't stop StopWatch: it's not
running");
        }
        long lastTime = System.nanoTime() - this.startTimeNanos;
        this.totalTimeNanos += lastTime;
        this.lastTaskInfo = new TaskInfo(this.currentTaskName, lastTime);
        if (this.keepTaskList) {
            this.taskList.add(this.lastTaskInfo);
        }
        ++this.taskCount;
        this.currentTaskName = null;
    }
```

14、输出日志记录执行主类名、时间信息

```
if (this.logStartupInfo) {
    new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopWatch);
}
```

15、发布应用上下文启动完成事件

触发所有SpringapplicationRunListener监听器的started事件方法

```
listeners.started(context);
    void started(ConfigurableApplicationContext context) {
        for (SpringApplicationRunListener listener : this.listeners) {
            listener.started(context);
        }
    }
```

16、执行所有Runner执行器

执行所有applicationRunner和CommandLineRunner两种运行器

```
callRunners(context, applicationArguments);
```

```
private void callRunners(ApplicationContext context, ApplicationArguments args)
{
        List<Object> runners = new ArrayList<>();

runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());

runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
        AnnotationAwareOrderComparator.sort(runners);
        for (Object runner : new LinkedHashSet<>(runners)) {
            if (runner instanceof ApplicationRunner) {
                callRunner((ApplicationRunner) runner, args);
            }
            if (runner instanceof CommandLineRunner) {
                callRunner((CommandLineRunner) runner, args);
            }
        }
    }
```

17、发布应用上下文就绪事件

触发所有springapplicationRunnListener将挺起的running事件方法

```
listeners.running(context);
void running(ConfigurableApplicationContext context) {
        for (SpringApplicationRunListener listener : this.listeners) {
            listener.running(context);
        }
    }
```

18、返回应用上下文

```
return context;
```

四、SpringBoot中获取factorys文件

注意：整个springboot框架中获取factorys文件的方式统一如下：

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

```
}
-----------------------------
    public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable
ClassLoader classLoader) {
        String factoryTypeName = factoryType.getName();
        return loadSpringFactories(classLoader).getOrDefault(factoryTypeName,
Collections.emptyList());
    }

    private static Map<String, List<String>> loadSpringFactories(@Nullable
ClassLoader classLoader) {
        MultiValueMap<String, String> result = cache.get(classLoader);
        if (result != null) {
            return result;
        }

        try {
            Enumeration<URL> urls = (classLoader != null ?
                    classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :

ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
            result = new LinkedMultiValueMap<>();
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties =
PropertiesLoaderUtils.loadProperties(resource);
                for (Map.Entry<?, ?> entry : properties.entrySet()) {
                    String factoryTypeName = ((String) entry.getKey()).trim();
                    for (String factoryImplementationName :
StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                        result.add(factoryTypeName,
factoryImplementationName.trim());
                    }
                }
            }
            cache.put(classLoader, result);
            return result;
        }
        catch (IOException ex) {
            throw new IllegalArgumentException("Unable to load factories from
location [" +
                    FACTORIES_RESOURCE_LOCATION + "]", ex);
        }
    }
-------------------------
    private <T> List<T> createSpringFactoriesInstances(Class<T> type, Class<?>[]
parameterTypes,
            ClassLoader classLoader, Object[] args, Set<String> names) {
        List<T> instances = new ArrayList<>(names.size());
        for (String name : names) {
            try {
                //装载class文件到内存
                Class<?> instanceClass = ClassUtils.forName(name, classLoader);
                Assert.isAssignable(type, instanceClass);
                Constructor<?> constructor =
instanceClass.getDeclaredConstructor(parameterTypes);
                //通过反射创建实例
```

```
                T instance = (T) BeanUtils.instantiateClass(constructor, args);
                instances.add(instance);
            }
            catch (Throwable ex) {
                throw new IllegalArgumentException("Cannot instantiate " + type
+ " : " + name, ex);
            }
        }
        return instances;
    }
```

spring.factory文件中的类的作用：

```
# PropertySource Loaders 属性文件加载器
org.springframework.boot.env.PropertySourceLoader=\
# properties文件加载器
org.springframework.boot.env.PropertiesPropertySourceLoader,\
# yaml文件加载器
org.springframework.boot.env.YamlPropertySourceLoader

# Run Listeners 运行时的监听器
org.springframework.boot.SpringApplicationRunListener=\
# 程序运行过程中所有监听通知都是通过此类来进行回调
org.springframework.boot.context.event.EventPublishingRunListener

# Error Reporters   错误报告器
org.springframework.boot.SpringBootExceptionReporter=\
org.springframework.boot.diagnostics.FailureAnalyzers

# Application Context Initializers
org.springframework.context.ApplicationContextInitializer=\
# 报告spring容器的一些常见的错误配置
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitiali
zer,\
# 设置spring应用上下文的ID
org.springframework.boot.context.ContextIdApplicationContextInitializer,\
# 使用环境属性context.initializer.classes指定初始化器进行初始化规则
org.springframework.boot.context.config.DelegatingApplicationContextInitializer,
\
org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitia
lizer,\
# 将内置servlet容器实际使用的监听端口写入到environment环境属性中
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer

# Application Listeners
org.springframework.context.ApplicationListener=\
# 应用上下文加载完成后对缓存做清除工作，响应事件ContextRefreshEvent
org.springframework.boot.ClearCachesApplicationListener,\
# 监听双亲应用上下文的关闭事件并往自己的孩子应用上下文中传播，相关事件
ParentContextAvailableEvent/ContextClosedEvent
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor,\
# 如果系统文件编码和环境变量中指定的不同则终止应用启动。具体的方法是比较系统属性file.encoding
和环境变量spring.mandatory-file-encoding是否相等(大小写不敏感)。
org.springframework.boot.context.FileEncodingApplicationListener,\
# 根据spring.output.ansi.enabled参数配置AnsiOutput
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
```

```
# EnvironmentPostProcessor，从常见的那些约定的位置读取配置文件，比如从以下目录读取
#application.properties,application.yml等配置文件：
# classpath:
# file:.
# classpath:config
# file:./config/:
# 也可以配置成从其他指定的位置读取配置文件
org.springframework.boot.context.config.ConfigFileApplicationListener,\
# 监听到事件后转发给环境变量context.listener.classes指定的那些事件监听器
org.springframework.boot.context.config.DelegatingApplicationListener,\
# 一个SmartApplicationListener,对环境就绪事件ApplicationEnvironmentPreparedEvent/应
用失败事件ApplicationFailedEvent做出响应，往日志DEBUG级别输出TCCL(thread context class
loader)的classpath。
org.springframework.boot.context.logging.ClasspathLoggingApplicationListener,\
# 检测正在使用的日志系统，默认时logback，，此时日志系统还没有初始化
org.springframework.boot.context.logging.LoggingApplicationListener,\
# 使用一个可以和Spring Boot可执行jar包配合工作的版本替换liquibase ServiceLocator
org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener

# Environment Post Processors
org.springframework.boot.env.EnvironmentPostProcessor=\
org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor,\
org.springframework.boot.env.SpringApplicationJsonEnvironmentPostProcessor,\
org.springframework.boot.env.SystemEnvironmentPropertySourceEnvironmentPostProce
ssor,\
org.springframework.boot.reactor.DebugAgentEnvironmentPostProcessor

# Failure Analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.diagnostics.analyzer.BeanCurrentlyInCreationFailureAnal
yzer,\
org.springframework.boot.diagnostics.analyzer.BeanDefinitionOverrideFailureAnaly
zer,\
org.springframework.boot.diagnostics.analyzer.BeanNotOfRequiredTypeFailureAnalyz
er,\
org.springframework.boot.diagnostics.analyzer.BindFailureAnalyzer,\
org.springframework.boot.diagnostics.analyzer.BindValidationFailureAnalyzer,\
org.springframework.boot.diagnostics.analyzer.UnboundConfigurationPropertyFailur
eAnalyzer,\
org.springframework.boot.diagnostics.analyzer.ConnectorStartFailureAnalyzer,\
org.springframework.boot.diagnostics.analyzer.NoSuchMethodFailureAnalyzer,\
org.springframework.boot.diagnostics.analyzer.NoUniqueBeanDefinitionFailureAnaly
zer,\
org.springframework.boot.diagnostics.analyzer.PortInUseFailureAnalyzer,\
org.springframework.boot.diagnostics.analyzer.ValidationExceptionFailureAnalyzer
,\
org.springframework.boot.diagnostics.analyzer.InvalidConfigurationPropertyNameFa
ilureAnalyzer,\
org.springframework.boot.diagnostics.analyzer.InvalidConfigurationPropertyValueF
ailureAnalyzer

# FailureAnalysisReporters
org.springframework.boot.diagnostics.FailureAnalysisReporter=\
org.springframework.boot.diagnostics.LoggingFailureAnalysisReporter

# Initializers
org.springframework.context.ApplicationContextInitializer=\
```

```
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitial
izer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingL
istener

# Application Listeners
org.springframework.context.ApplicationListener=\
# 另外单独启动一个线程实例化并调用run方法，包括验证器、消息转换器等
org.springframework.boot.autoconfigure.BackgroundPreinitializer

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

# 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.SpringApplicationAdminJmxAutoConfig
uration,\
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.CloudServiceConnectorsAutoConfigura
tion,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfig
uration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfigurat
ion,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoCo
nfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoC
onfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositor
iesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoC
onfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoC
onfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositor
iesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoC
onfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfi
guration,\
```

```
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoC
onfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositor
iesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchR
epositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveRestClientAuto
Configuration,\
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfigurati
on,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfigurati
on,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoC
onfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfigurati
on,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfigurat
ion,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,
\
org.springframework.boot.autoconfigure.elasticsearch.rest.RestClientAutoConfigur
ation,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfi
guration,\
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfigurati
on,\
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.DataSourceTransactionManagerAutoConf
iguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfigu
ration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfigurati
on,\
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.EmbeddedMongoAutoConfigura
tion,\
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.rsocket.RSocketMessagingAutoConfiguration
,\
org.springframework.boot.autoconfigure.rsocket.RSocketRequesterAutoConfiguration
,\
org.springframework.boot.autoconfigure.rsocket.RSocketServerAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketStrategiesAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoCo
nfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfig
uration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoCon
figuration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServ
iceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.rsocket.RSocketSecurityAutoConfi
guration,\
org.springframework.boot.autoconfigure.security.saml2.Saml2RelyingPartyAutoConfi
guration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2Clie
ntAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveO
Auth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2Re
sourceServerAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration

# Failure analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefinitionFailureAnalyzer,\
org.springframework.boot.autoconfigure.flyway.FlywayMigrationScriptMissingFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailureAnalyzer,\
org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFailureAnalyzer
```

```
# Template availability providers
org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider=\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerTemplateAvailability
Provider,\
org.springframework.boot.autoconfigure.mustache.MustacheTemplateAvailabilityProv
ider,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAvailabilit
yProvider,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafTemplateAvailabilityPr
ovider,\
org.springframework.boot.autoconfigure.web.servlet.JspTemplateAvailabilityProvid
er
```