```
Springboot30StartupApplication【10】→SpringApplication.run(Springboot30StartupApplication.class, args);
    SpringApplication【1332】→return run(new Class<?>[] { primarySource }, args);
        SpringApplication【1343】→return new SpringApplication(primarySources).run(args);
            SpringApplication【1343】→SpringApplication(primarySources)
            # 加载各种配置信息，初始化各种配置对象
                SpringApplication【266】→this(null, primarySources);
                    SpringApplication【280】→public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources)
                        SpringApplication【281】→this.resourceLoader = resourceLoader;
                        # 初始化资源加载器
                        SpringApplication【283】→this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
                        # 初始化配置类的类名信息（格式转换）
                        SpringApplication【284】→this.webApplicationType = WebApplicationType.deduceFromClasspath();
                        # 确认当前容器加载的类型
                        SpringApplication【285】→this.bootstrapRegistryInitializers = getBootstrapRegistryInitializersFromSpringFactories();
                        # 获取系统配置引导信息
                        SpringApplication【286】→setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));
                        # 获取 ApplicationContextInitializer.class 对应的实例
                        SpringApplication【287】→setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
                        # 初始化监听器，对初始化过程及运行过程进行干预
                        SpringApplication【288】→this.mainApplicationClass = deduceMainApplicationClass();
                        # 初始化了引导类类名信息，备用
            SpringApplication【1343】→new SpringApplication(primarySources).run(args)
            # 初始化容器，得到 ApplicationContext 对象
                SpringApplication【323】→StopWatch stopWatch = new StopWatch();
                # 设置计时器
                SpringApplication【324】→stopWatch.start();
                # 计时开始
                SpringApplication【325】→DefaultBootstrapContext bootstrapContext = createBootstrapContext();
                # 系统引导信息对应的上下文对象
                SpringApplication【327】→configureHeadlessProperty();
                # 模拟输入输出信号，避免出现因缺少外设导致的信号传输失败，进而引发错误（模拟显示器，键盘，鼠标...）
                    java.awt.headless=true
                SpringApplication【328】→SpringApplicationRunListeners listeners = getRunListeners(args);
```

# 获取当前注册的所有监听器
SpringApplication【329】→listeners.starting(bootstrapContext, this.mainApplicationClass);
# 监听器执行了对应的操作步骤
SpringApplication【331】→ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
# 获取参数
SpringApplication【333】→ConfigurableEnvironment environment = prepareEnvironment(listeners, bootstrapContext, applicationArguments);
# 将前期读取的数据加载成了一个环境对象，用来描述信息
SpringApplication【333】→configureIgnoreBeanInfo(environment);
# 做了一个配置，备用
SpringApplication【334】→Banner printedBanner = printBanner(environment);
# 初始化 logo
SpringApplication【335】→context = createApplicationContext();
# 创建容器对象，根据前期配置的容器类型进行判定并创建
SpringApplication【363】→context.setApplicationStartup(this.applicationStartup);
# 设置启动模式
SpringApplication【337】→prepareContext(bootstrapContext, context, environment, listeners, applicationArguments, printedBanner);
# 对容器进行设置，参数来源于前期的设定
SpringApplication【338】→refreshContext(context);
# 刷新容器环境
SpringApplication【339】→afterRefresh(context, applicationArguments);
# 刷新完毕后做后处理
SpringApplication【340】→stopWatch.stop();
# 计时结束
SpringApplication【341】→if (this.logStartupInfo) {
# 判定是否记录启动时间的日志
SpringApplication【342】→ new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
# 创建日志对应的对象，输出日志信息，包含启动时间
SpringApplication【344】→listeners.started(context);
# 监听器执行了对应的操作步骤
SpringApplication【345】→callRunners(context, applicationArguments);
#
SpringApplication【353】→listeners.running(context);
# 监听器执行了对应的操作步骤