



[回到首页](#)

芋道源码 —— 知识星球

我是一段不羁的公告！

记得给芬芳这 3 个项目加油，添加一个 STAR 噢。

<https://github.com/YunaiV/SpringBoot-Labs>

<https://github.com/YunaiV/onemall>

<https://github.com/YunaiV/ruoyi-vue-pro>

[2021-01-07](#)

[Spring Boot](#)

精尽 Spring Boot 源码分析 —— SpringApplication

1. 概述

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // <1>
public class MVCApplication {

    public static void main(String[] args) {
        SpringApplication.run(MVCApplication.class, args); // <2>
    }

}
```

<1> 处，使用 `@SpringBootApplication` 注解，标明是 Spring Boot 应用。通过它，可以开启自动配置的功能。

<2> 处，调用 `SpringApplication#run(Class<?>... primarySources)` 方法，启动 Spring Boot 应用。

上述的代码，是我们使用 Spring Boot 时，最最最常用的代码。而本文，我们先来分析 Spring Boot 应用的启动过程。

关于 `@SpringBootApplication` 注解，我们会后面单独开文章，详细解析。

2. SpringApplication

`org.springframework.boot.SpringApplication`，Spring 应用启动器。正如其代码上所添加的注释，它来提供启动 Spring 应用的功能。

```
Class that can be used to bootstrap and launch a Spring application from a Java main method.
```

大多数情况下，我们都是使用它提供的静态方法：

```
// SpringApplication.java

public static void main(String[] args) throws Exception {
    SpringApplication.run(new Class<?>[0], args);
}

public static ConfigurableApplicationContext run(Class<?> primarySource, String... args) {
    return run(new Class<?>[] { primarySource }, args);
}

public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {
    // 创建 SpringApplication 对象，并执行运行。
    return new SpringApplication(primarySources).run(args);
}
```

前两个静态方法，最终调用的是第 3 个静态方法。而第 3 个静态方法，实现的逻辑就是：

- 首先，创建一个 SpringApplication 对象。详细的解析，见 [\[2.1 构造方法\]](#)。
- 然后，调用 SpringApplication#run(Class<?> primarySource, String... args) 方法，运行 Spring 应用。详细解析，见 [\[2.2 run\]](#)。

2.1 构造方法

```
// SpringApplication.java

/**
 * 资源加载器
 */
private ResourceLoader resourceLoader;

/**
 * 主要的 Java Config 类的数组
 */
private Set<Class<?>> primarySources;

/**
 * Web 应用类型
 */
private WebApplicationType webApplicationType;

/**
 * ApplicationContextInitializer 数组
 */
private List<ApplicationContextInitializer<?>> initializers;

/**
 * ApplicationListener 数组
 */
private List<ApplicationListener<?>> listeners;

public SpringApplication(Class<?>... primarySources) {
    this(null, primarySources);
}

public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    // 初始化 initializers 属性
}
```

```

setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));
// 初始化 listeners 属性
setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
this.mainApplicationClass = deduceMainApplicationClass();
}

```

SpringApplication 的变量比较多，我们先只看构造方法提到的几个。

resourceLoader 属性，资源加载器。可以暂时不理解，感兴趣的胖友，可以看看 [《【死磕 Spring】—— IoC 之 Spring 统一资源加载策略》](#) 文章。

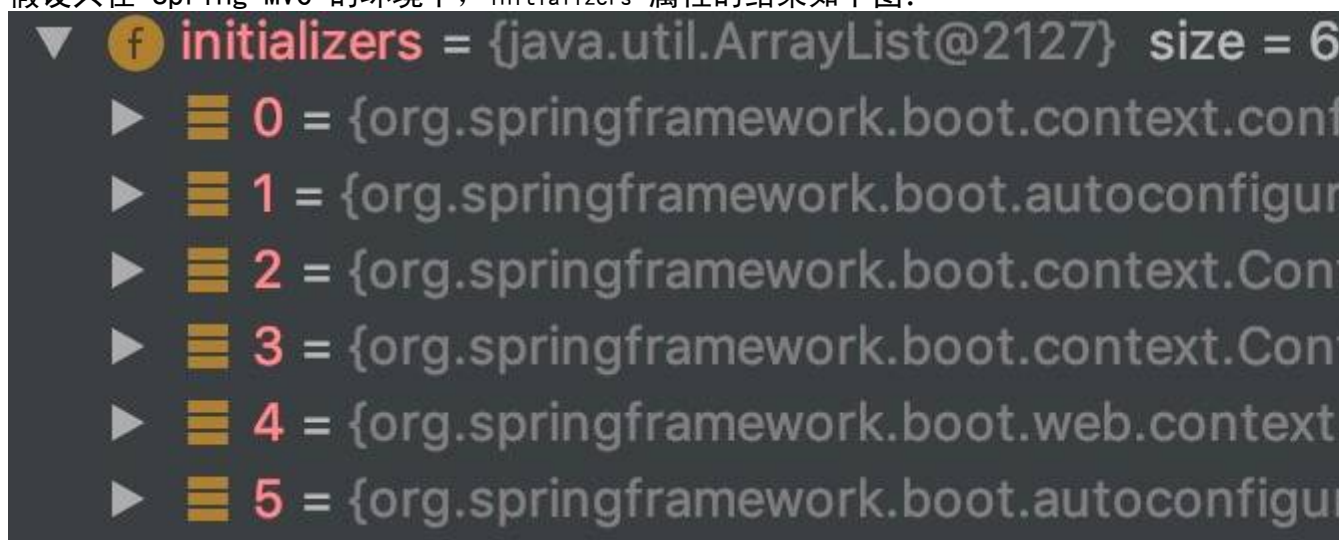
primarySources 属性，主要的 Java Config 类的数组。在文初提供的示例，就是 MVCApplication 类。

webApplicationType 属性，调用 WebApplicationType#deduceFromClasspath() 方法，通过 classpath，判断 Web 应用类型。

- 具体的原理是，是否存在指定的类，芳芳已经在 [WebApplicationType](#) 上的方法添加了注释，直接瞅一眼就明白了。
- 这个属性，在下面的 #createApplicationContext() 方法，将根据它的值（类型），创建不同类型的 ApplicationContext 对象，即 Spring 容器的类型不同。

initializers 属性，ApplicationContextInitializer 数组。

- 通过 #getSpringFactoriesInstances(Class<T> type) 方法，进行获得 ApplicationContextInitializer 类型的对象数组，详细的解析，见 [「2.1.1 getSpringFactoriesInstances」](#) 方法。
- 假设只在 Spring MVC 的环境下，initializers 属性的结果如下图：



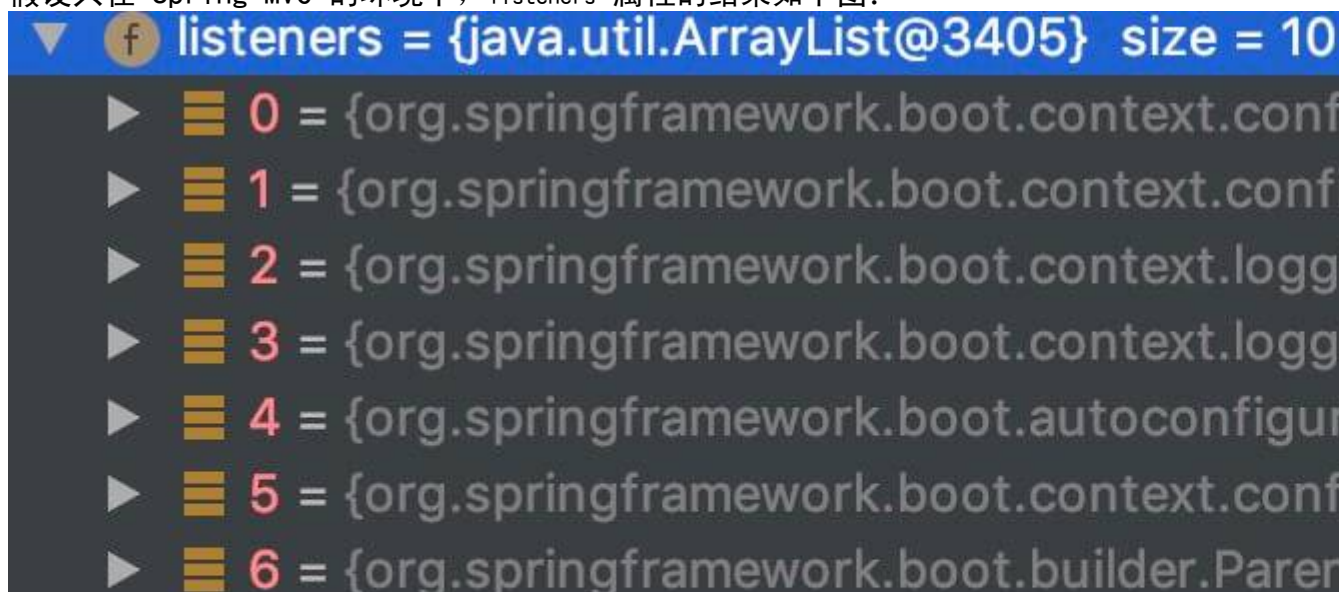
```

▼ f initializers = {java.util.ArrayList@2127} size = 6
  ▶ 0 = {org.springframework.boot.context.conf...
  ▶ 1 = {org.springframework.boot.autoconfigure...
  ▶ 2 = {org.springframework.boot.context.Con...
  ▶ 3 = {org.springframework.boot.context.Con...
  ▶ 4 = {org.springframework.boot.web.context...
  ▶ 5 = {org.springframework.boot.autoconfigure...

```

listeners 属性，ApplicationListener 数组。

- 也是通过 #getSpringFactoriesInstances(Class<T> type) 方法，进行获得 ApplicationListener 类型的对象数组。
- 假设只在 Spring MVC 的环境下，listeners 属性的结果如下图：



```

▼ f listeners = {java.util.ArrayList@3405} size = 10
  ▶ 0 = {org.springframework.boot.context.conf...
  ▶ 1 = {org.springframework.boot.context.conf...
  ▶ 2 = {org.springframework.boot.context.logg...
  ▶ 3 = {org.springframework.boot.context.logg...
  ▶ 4 = {org.springframework.boot.autoconfigure...
  ▶ 5 = {org.springframework.boot.context.conf...
  ▶ 6 = {org.springframework.boot.builder.Paren...

```

`mainApplicationClass` 属性，调用 `#deduceMainApplicationClass()` 方法，获得是调用了哪个 `#main(String[] args)` 方法，代码如下：

```
// SpringApplication.java

private Class<?> deduceMainApplicationClass() {
    try {
        // 获得当前 StackTraceElement 数组
        StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
        // 判断哪个执行了 main 方法
        for (StackTraceElement stackTraceElement : stackTrace) {
            if ("main".equals(stackTraceElement.getMethodName())) {
                return Class.forName(stackTraceElement.getClassName());
            }
        }
    } catch (ClassNotFoundException ex) {
        // Swallow and continue
    }
    return null;
}
```

- 在文初的例子中，就是 `MVCApplication` 类。
- 这个 `mainApplicationClass` 属性，没有什么逻辑上的用途，主要就是用来打印下日志，说明是通过这个类启动 Spring 应用的。

2.1.1 getSpringFactoriesInstances

`#getSpringFactoriesInstances(Class<T> type)` 方法，获得指定类类对应的对象们。代码如下：

```
// SpringApplication.java

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
    // <1> 加载指定类型对应的，在 `META-INF/spring.factories` 里的类名的数组
    Set<String> names = new LinkedHashSet<>() {
        SpringFactoriesLoader.loadFactoryNames(type, classLoader);
    };
    // <2> 创建对象们
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
        classLoader, args, names);
    // <3> 排序对象们
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

<1> 处，调用 `SpringFactoriesLoader#loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader)` 方法，加载指定类型对应的，在 `META-INF/spring.factories` 里的类名的数组。

- 在 [META-INF/spring.factories](#) 文件中，会以 KEY-VALUE 的格式，配置每个类对应的实现类们。
- 关于 `SpringFactoriesLoader` 的该方法，我们就不去细看了。很多时候，我们看源

码的时候，不需要陷入到每个方法的细节中。非关键的方法，猜测到具体的用途后，跳过也是没问题的。

<2> 处，调用 `#createSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, ClassLoader classLoader, Object[] args, Set<String> names)` 方法，创建对象们。代码如下：

```
// SpringApplication.java

/**
 * 创建对象的数组
 *
 * @param type 父类
 * @param parameterTypes 构造方法的参数类型
 * @param classLoader 类加载器
 * @param args 参数
 * @param names 类名的数组
 * @param <T> 泛型
 * @return 对象的数组
 */
private <T> List<T> createSpringFactoriesInstances(Class<T> type,
    Class<?>[] parameterTypes, ClassLoader classLoader, Object[] args,
    Set<String> names) {
    List<T> instances = new ArrayList<>(names.size()); // 数组大小，细节~
    // 遍历 names 数组
    for (String name : names) {
        try {
            // 获得 name 对应的类
            Class<?> instanceClass = ClassUtils.forName(name, classLoader);
            // 判断类是否实现自 type 类
            Assert.isAssignable(type, instanceClass);
            // 获得构造方法
            Constructor<?> constructor = instanceClass.getDeclaredConstructor(parameterTypes);
            // 创建对象
            T instance = (T) BeanUtils.instantiateClass(constructor, args);
            instances.add(instance);
        } catch (Throwable ex) {
            throw new IllegalArgumentException("Cannot instantiate " + type + " : " + name, ex);
        }
    }
    return instances;
}
```

。比较简单，就不多做解释了。

<3> 处，调用 `AnnotationAwareOrderComparator#sort(List<?> list)` 方法，排序对象们。例如说，类上有 [@Order](#) 注解。

2.2 run

`#run(String... args)` 方法，运行 Spring 应用。代码如下：

芳芳：这是一个饱满的方法，所以逻辑比较多哈。

```
// SpringApplication.java

public ConfigurableApplicationContext run(String... args) {
```

```

// <1> 创建 Stopwatch 对象，并启动。StopWatch 主要用于简单统计 run 启动过程的时长。
StopWatch stopWatch = new StopWatch();
stopWatch.start();
//
ConfigurableApplicationContext context = null;
Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
// <2> 配置 headless 属性
configureHeadlessProperty();
// 获得 SpringApplicationRunListener 的数组，并启动监听
SpringApplicationRunListeners listeners = getRunListeners(args);
listeners.starting();
try {
    // <3> 创建 ApplicationArguments 对象
    ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
    // <4> 加载属性配置。执行完成后，所有的 environment 的属性都会加载进来，包括 application.properties 和外部的属性文件
    ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
    configureIgnoreBeanInfo(environment);
    // <5> 打印 Spring Banner
    Banner printedBanner = printBanner(environment);
    // <6> 创建 Spring 容器。
    context = createApplicationContext();
    // <7> 异常报告器
    exceptionReporters = getSpringFactoriesInstances(
        SpringBootExceptionReporter.class,
        new Class[] { ConfigurableApplicationContext.class }, context);
    // <8> 主要是调用所有初始化类的 initialize 方法
    prepareContext(context, environment, listeners, applicationArguments,
        printedBanner);
    // <9> 初始化 Spring 容器。
    refreshContext(context);
    // <10> 执行 Spring 容器的初始化的后置逻辑。默认实现为空。
    afterRefresh(context, applicationArguments);
    // <11> 停止 Stopwatch 统计时长
    stopWatch.stop();
    // <12> 打印 Spring Boot 启动的时长日志。
    if (this.logStartupInfo) {
        new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
    }
    // <13> 通知 SpringApplicationRunListener 的数组，Spring 容器启动完成。
    listeners.started(context);
    // <14> 调用 ApplicationRunner 或者 CommandLineRunner 的运行方法。
    callRunners(context, applicationArguments);
} catch (Throwable ex) {
    // <14.1> 如果发生异常，则进行处理，并抛出 IllegalStateException 异常
    handleRunFailure(context, ex, exceptionReporters, listeners);
    throw new IllegalStateException(ex);
}

// <15> 通知 SpringApplicationRunListener 的数组，Spring 容器运行中。
try {
    listeners.running(context);
} catch (Throwable ex) {
    // <15.1> 如果发生异常，则进行处理，并抛出 IllegalStateException 异常
    handleRunFailure(context, ex, exceptionReporters, null);
    throw new IllegalStateException(ex);
}
return context;
}

```

<1> 处，创建 Stopwatch 对象，并调用 StopWatch#run() 方法来启动。StopWatch 主要用于简单

⟨2⟩ 处，配置 headless 属性。这个逻辑，可以无视，和 AWT 相关。

```
// SpringApplication.java

private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class };
    return new SpringApplicationRunListeners(logger, getSpringFactoriesInstances(
        SpringApplicationRunListener.class, types, this, args));
}
```

- ▼ **listeners** = {org.springframework.boot.SpringApplication}
- ▶ **log** = {org.apache.commons.logging.LogAdapter}
- ▼ **listeners** = {java.util.ArrayList@2375} size = 0
 - ▼ **0** = {org.springframework.boot.context.event.ApplicationEventMulticaster}
 - ▶ **application** = {org.springframework.boot.SpringApplication}
 - ▶ **args** = {java.lang.String[0]@2360}
 - ▶ **initialMulticaster** = {org.springframework.boot.context.event.ApplicationEventMulticaster}

- <4> 处，调用 `#prepareEnvironment(SpringApplicationRunListeners listeners, ApplicationArguments applicationArguments)` 方法，加载属性配置。执行完成后，所有的 `environment` 的属性都会加载进来，包括 `application.properties` 和外部的属性配置。详细的，胖友先一起跳到 [\[2.2.1 prepareEnvironment\]](#) 中。

```

      .   _       _       _       _       _
     /\ / _ \   - - - - - (\) - - - - - \ \ \ \ \
    ( ( ) \_ | ' | ' | ' | ' \_ | \ \ \ \
     \|/_ )|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
     |_||_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
=====|_|=====|_|=/ / / / /
:: Spring Boot ::

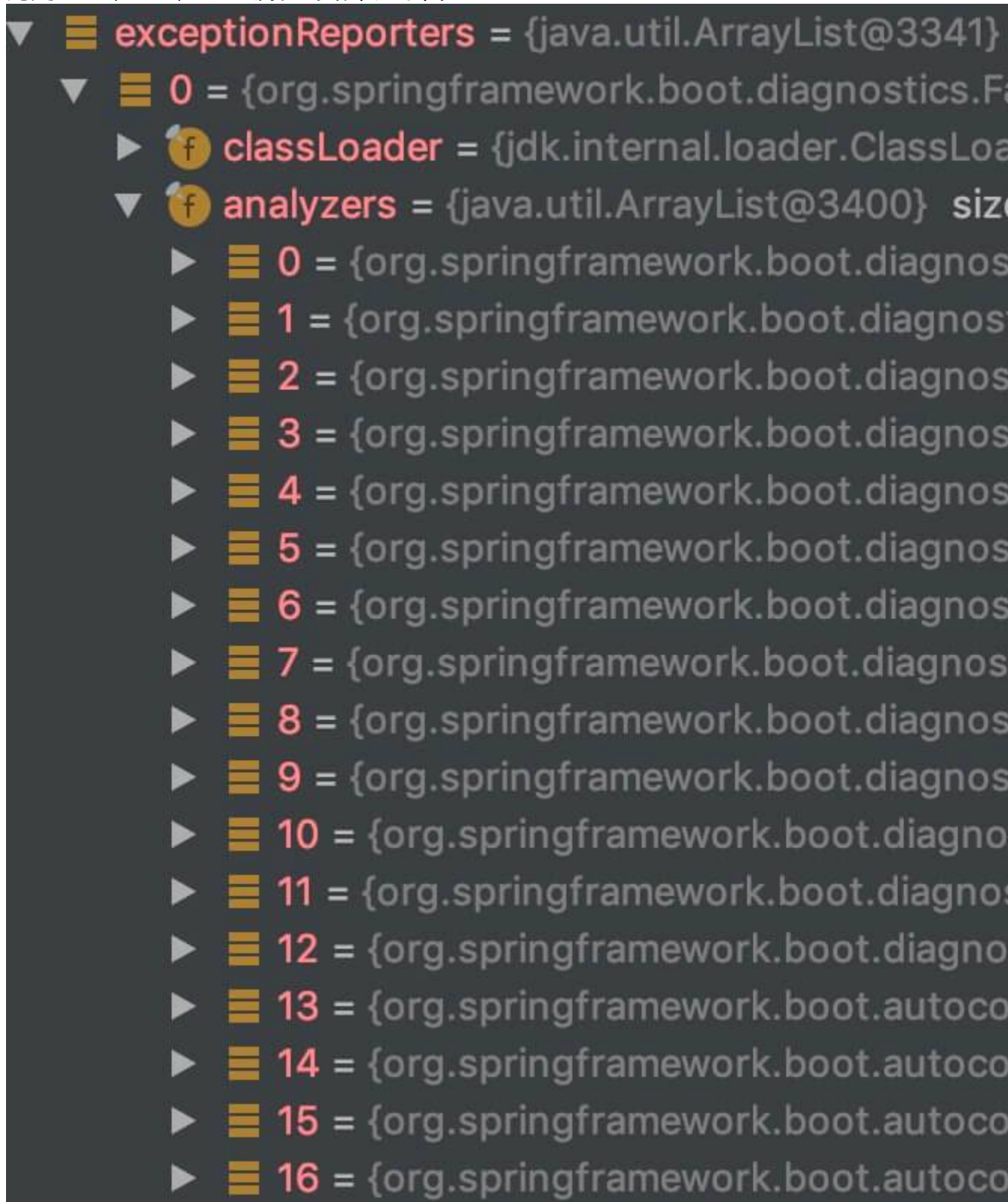
```

- <6> 处，调用 `#createApplicationContext()` 方法，创建 Spring 容器。详细解析，见 [2.2.2

[createApplicationContext](#) 。

<7> 处，通过 `#getSpringFactoriesInstances(Class<T> type)` 方法，进行获得 `SpringBootExceptionReporter` 类型的对象数组。`SpringBootExceptionReporter`，记录启动过程中的异常信息。

- 此处，`exceptionReporters` 属性的结果如下图：



- 关于 `SpringBootExceptionReporter`，感兴趣的胖友，自己研究先。

<8> 处，调用 `#prepareContext(...)` 方法，主要是调用所有初始化类的 `#initialize(...)` 方法。详细解析，见 [\[2.2.3 prepareContext\]](#) 。

<9> 处，调用 `#refreshContext(ConfigurableApplicationContext context)` 方法，启动

（刷新） Spring 容器。详细解析，见 [\[2.2.4 refreshContext\]](#) 。

<10> 处，调用 `#afterRefresh(ConfigurableApplicationContext context, ApplicationArguments args)` 方法，执行 Spring 容器的初始化的后置逻辑。默认实现为空。代码如下：

```
// SpringApplication.java

protected void afterRefresh(ConfigurableApplicationContext context, ApplicationArguments args) {
}
```

<11> 处，停止 `StopWatch` 统计时长。

<12> 处，打印 Spring Boot 启动的时长日志。效果如下：

```
2019-01-28 20:42:03.338 INFO 53001 --- [           main] c.iocoder.springboot.mvc.MVCApplication : Started MV
```

<13> 处，调用 `SpringApplicationRunListeners#started(ConfigurableApplicationContext context)` 方法，通知 `SpringApplicationRunListener` 的数组，Spring 容器启动完成。

<14> 处，调用 `#callRunners(ApplicationContext context, ApplicationArguments args)` 方法，调用 `ApplicationRunner` 或者 `CommandLineRunner` 的运行方法。详细解析，见 [\[2.2.5 callRunners\]](#) 。

- <14.1> 处，如果发生异常，则调用 `#handleRunFailure(...)` 方法，交给 `SpringBootExceptionHandler` 进行处理，并抛出 `IllegalStateException` 异常。

<15> 处，调用 `SpringApplicationRunListeners#running(ConfigurableApplicationContext context)` 方法，通知 `SpringApplicationRunListener` 的数组，Spring 容器运行中。

- <15.1> 处，如果发生异常，则调用 `#handleRunFailure(...)` 方法，交给 `SpringBootExceptionHandler` 进行处理，并抛出 `IllegalStateException` 异常。

2.2.1 prepareEnvironment

芴芴：这个方法，大体看下即可。

`#prepareEnvironment(SpringApplicationRunListeners listeners, ApplicationArguments applicationArguments)` 方法，加载属性配置。代码如下：

```
// SpringApplication.java

private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners, ApplicationArguments appl
// Create and configure the environment
// <1> 创建 ConfigurableEnvironment 对象，并进行配置
ConfigurableEnvironment environment = getOrCreateEnvironment();
configureEnvironment(environment, applicationArguments.getSourceArgs());
// <2> 通知 SpringApplicationRunListener 的数组，环境变量已经准备完成。
listeners.environmentPrepared(environment);
// <3> 绑定 environment 到 SpringApplication 上
bindToSpringApplication(environment);
// <4> 如果非自定义 environment，则根据条件转换
if (!this.isCustomEnvironment) {
    environment = new EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment, deduceEnv
}
// <5> 如果有 attach 到 environment 上的 MutablePropertySources，则添加到 environment 的 PropertySource 中。
ConfigurationPropertySources.attach(environment);
```

```

return environment;
}

```

<1> 处，调用 `#getOrCreateEnvironment()` 方法，创建 `ConfigurableEnvironment` 对象。代码如下：

```

// SpringApplication.java

private ConfigurableEnvironment getOrCreateEnvironment() {
    // 已经存在，则进行返回
    if (this.environment != null) {
        return this.environment;
    }
    // 不存在，则根据 webApplicationType 类型，进行创建。
    switch (this.webApplicationType) {
        case SERVLET:
            return new StandardServletEnvironment();
        case REACTIVE:
            return new StandardReactiveWebEnvironment();
        default:
            return new StandardEnvironment();
    }
}

```

- 根据 `webApplicationType` 类型，会创建不同类型的 `ConfigurableEnvironment` 对象。
- 例如说，`Servlet` 需要考虑 `<servletContextInitParams />` 和 `<servletConfigInitParams />` 等配置参数。

<1> 处，调用 `#configureEnvironment(ConfigurableEnvironment environment, String[] args)` 方法，配置 `environment` 变量。代码如下：

```

// SpringApplication.java

/**
 * 是否添加共享的 ConversionService
 */
private boolean addConversionService = true;

protected void configureEnvironment(ConfigurableEnvironment environment, String[] args) {
    // <1.1> 设置 environment 的 conversionService 属性
    if (this.addConversionService) {
        ConversionService conversionService = ApplicationConversionService.getSharedInstance();
        environment.setConversionService((ConfigurableConversionService) conversionService);
    }
    // <1.2> 增加 environment 的 PropertySource 属性源
    configurePropertySources(environment, args);
    // <1.3> 配置 environment 的 activeProfiles 属性
    configureProfiles(environment, args);
}

```

- <1.1> 处，设置 `environment` 的 `conversionService` 属性。可以暂时无视。感兴趣的胖友，可以看看 [《【死磕 Spring】—— 环境 & 属性: PropertySource、Environment、Profile》](#)
- <1.2>

处，增加 `environment` 的 `PropertySource` 属性源。代码如下：

```
// SpringApplication.java

/**
 * 是否添加 JVM 启动参数
 */
private boolean addCommandLineProperties = true;

/**
 * 默认的属性集合
 */
private Map<String, Object> defaultProperties;

protected void configurePropertySources(ConfigurableEnvironment environment,
    String[] args) {
    MutablePropertySources sources = environment.getPropertySources();
    // 配置的 defaultProperties
    if (this.defaultProperties != null && !this.defaultProperties.isEmpty()) {
        sources.addLast(new MapPropertySource("defaultProperties", this.defaultProperties));
    }
    // 来自启动参数的
    if (this.addCommandLineProperties && args.length > 0) {
        String name = CommandLinePropertySource.COMMAND_LINE_PROPERTY_SOURCE_NAME;
        if (sources.contains(name)) { // 已存在，就进行替换
            PropertySource<?> source = sources.get(name);
            CompositePropertySource composite = new CompositePropertySource(name);
            composite.addPropertySource(new SimpleCommandLinePropertySource(
                "springApplicationCommandLineArgs", args));
            composite.addPropertySource(source);
            sources.replace(name, composite);
        } else { // 不存在，就进行添加
            sources.addFirst(new SimpleCommandLinePropertySource(args));
        }
    }
}
```

- 代码上可以看出，可以根据配置的 `defaultProperties`、或者 JVM 启动参数，作为附加的 `PropertySource` 属性源。

- <1.3> 处，配置 `environment` 的 `activeProfiles` 属性。代码如下：

```
// SpringApplication.java

/**
 * 附加的 profiles 的数组
 */
private Set<String> additionalProfiles = new HashSet<>();

protected void configureProfiles(ConfigurableEnvironment environment, String[] args) {
    environment.getActiveProfiles(); // ensure they are initialized 保证已经被初始化
    // But these ones should go first (last wins in a property key clash)
    Set<String> profiles = new LinkedHashSet<>(this.additionalProfiles);
    profiles.addAll(Arrays.asList(environment.getActiveProfiles()));
    // 设置 activeProfiles
    environment.setActiveProfiles(StringUtils.toStringArray(profiles));
}
```

- 。不了解 Profile 的胖友，可以看看 [《Spring Boot 激活 profile 的几种方式》](#) 文章。

<2> 处，调用 `SpringApplicationRunListeners#environmentPrepared(ConfigurableEnvironment environment)` 方法，通知 `SpringApplicationRunListener` 的数组，环境变量已经准备完成。

<3> 处，调用 `#bindToSpringApplication(ConfigurableEnvironment environment)` 方法，绑定 `environment` 到 `SpringApplication` 上。暂时不太知道用途。

<4> 处，如果非自定义 `environment`，则根据条件转换。默认情况下，`isCustomEnvironment` 为 `false`，所以会执行这块逻辑。但是，一般情况下，返回的还是 `environment` 自身，所以可以无视这块逻辑先。

<5> 处，调用 `ConfigurationPropertySources#attach(Environment environment)` 静态方法，如果有 `attach` 到 `environment` 上的 `MutablePropertySources`，则添加到 `environment` 的 `PropertySource` 中。这块逻辑，也可以先无视。

2.2.2 createApplicationContext

`#createApplicationContext()` 方法，创建 Spring 容器。代码如下：

```
// SpringApplication.java

/**
 * The class name of application context that will be used by default for non-web
 * environments.
 */
public static final String DEFAULT_CONTEXT_CLASS = "org.springframework.context."
    + "AnnotationConfigApplicationContext";

/**
 * The class name of application context that will be used by default for web
 * environments.
 */
public static final String DEFAULT_SERVLET_WEB_CONTEXT_CLASS = "org.springframework.boot."
    + "web.servlet.context.AnnotationConfigServletWebServerApplicationContext";

/**
 * The class name of application context that will be used by default for reactive web
 * environments.
 */
public static final String DEFAULT_REACTIVE_WEB_CONTEXT_CLASS = "org.springframework."
    + "boot.web.reactive.context.AnnotationConfigReactiveWebServerApplicationContext";

protected ConfigurableApplicationContext createApplicationContext() {
    // 根据 webApplicationType 类型，获得 ApplicationContext 类型
    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 Applicat
        }
    }
}
// 创建 ApplicationContext 对象
return (ConfigurableApplicationContext) BeanUtils.instantiateClass(contextClass);
}

```

根据 `webApplicationType` 类型，获得对应的 `ApplicationContext` 对象。

2.2.3 prepareContext

`#prepareContext(ConfigurableApplicationContext context, ConfigurableEnvironment environment, SpringApplicationRunListeners listeners, ApplicationArguments applicationArguments, Banner printedBanner)` 方法，准备 `ApplicationContext` 对象，主要是初始化它的一些属性。代码如下：

```

// SpringApplication.java

private void prepareContext(ConfigurableApplicationContext context,
    ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments, Banner printedBanner) {
    // <1> 设置 context 的 environment 属性
    context.setEnvironment(environment);
    // <2> 设置 context 的一些属性
    postProcessApplicationContext(context);
    // <3> 初始化 ApplicationContextInitializer
    applyInitializers(context);
    // <4> 通知 SpringApplicationRunListener 的数组，Spring 容器准备完成。
    listeners.contextPrepared(context);
    // <5> 打印日志
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }
    // Add boot specific singleton beans
    // <6> 设置 beanFactory 的属性
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    beanFactory.registerSingleton("springApplicationArguments", applicationArguments);
    if (printedBanner != null) {
        beanFactory.registerSingleton("springBootBanner", printedBanner);
    }
    if (beanFactory instanceof DefaultListableBeanFactory) {
        ((DefaultListableBeanFactory) beanFactory).setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
    }
    // Load the sources
    // <7> 加载 BeanDefinition 们
    Set<Object> sources = getAllSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    load(context, sources.toArray(new Object[0]));
    // <8> 通知 SpringApplicationRunListener 的数组，Spring 容器加载完成。
    listeners.contextLoaded(context);
}

```

这个方法，还是蛮长的，主要是给 `context` 的属性做赋值，以及 `ApplicationContextInitializer` 的初始化。

<1> 处，设置 `context` 的 `environment` 属性。

<2> 处，调用 `#postProcessApplicationContext(ConfigurableApplicationContext context)` 方法，设置 `context` 的一些属性。代码如下：

```
// SpringApplication.java

protected void postProcessApplicationContext(ConfigurableApplicationContext context) {
    if (this.beanNameGenerator != null) {
        context.getBeanFactory().registerSingleton(AnnotationConfigUtils.CONFIGURATION_BEAN_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.getSharedInstance());
    }
}
```

<3> 处，调用 `#applyInitializers(ConfigurableApplicationContext context)` 方法，初始化 `ApplicationContextInitializer`。代码如下：

```
// SpringApplication.java

protected void applyInitializers(ConfigurableApplicationContext context) {
    // 遍历 ApplicationContextInitializer 数组
    for (ApplicationContextInitializer initializer : getInitializers()) {
        // 校验 ApplicationContextInitializer 的泛型非空
        Class<?> requiredType = GenericTypeResolver.resolveTypeArgument(
            initializer.getClass(), ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, "Unable to call initializer.");
        // 初始化 ApplicationContextInitializer
        initializer.initialize(context);
    }
}
```

- 遍历 `ApplicationContextInitializer` 数组，逐个调用 `ApplicationContextInitializer#initialize(context)` 方法，进行初始化。

<4> 处，调用 `SpringApplicationRunListeners#contextPrepared(ConfigurableApplicationContext context)` 方法，通知 `SpringApplicationRunListener` 的数组，Spring 容器准备完成。

<5> 处，打印日志。效果如下：

```
2019-01-28 17:53:31.600 INFO 21846 --- [main] c.iocoder.springboot.mvc.MVCApplication : Starting MVCApplication
2019-01-28 17:53:40.028 INFO 21846 --- [main] c.iocoder.springboot.mvc.MVCApplication : The following profiles are active:
```

- 具体的方法逻辑，胖友自己瞅瞅哈。

<6> 处，设置 `beanFactory` 的属性。

<7> 处，调用 `#load(ApplicationContext context, Object[] sources)` 方法，加载 `BeanDefinition` 们。

代码如下：

```
// SpringApplication.java

protected void load(ApplicationContext context, Object[] sources) {
    if (logger.isDebugEnabled()) {
        logger.debug("Loading source " + StringUtils.arrayToCommaDelimitedString(sources));
    }
    // <1> 创建 BeanDefinitionLoader 对象
    BeanDefinitionLoader loader = createBeanDefinitionLoader(getBeanDefinitionRegistry(context), sources);
    // <2> 设置 loader 的属性
    if (this.beanNameGenerator != null) {
        loader.setBeanNameGenerator(this.beanNameGenerator);
    }
    if (this.resourceLoader != null) {
        loader.setResourceLoader(this.resourceLoader);
    }
    if (this.environment != null) {
        loader.setEnvironment(this.environment);
    }
    // <3> 执行 BeanDefinition 加载
    loader.load();
}
```

- <1> 处，调用 #getBeanDefinitionRegistry(ApplicationContext context) 方法，创建 BeanDefinitionRegistry 对象。代码如下：

```
// SpringApplication.java

private BeanDefinitionRegistry getBeanDefinitionRegistry(ApplicationContext context) {
    if (context instanceof BeanDefinitionRegistry) {
        return (BeanDefinitionRegistry) context;
    }
    if (context instanceof AbstractApplicationContext) {
        return (BeanDefinitionRegistry) ((AbstractApplicationContext) context)
            .getBeanFactory();
    }
    throw new IllegalStateException("Could not locate BeanDefinitionRegistry");
}
```

- 关于 BeanDefinitionRegistry 类，暂时不需要深入了解。感兴趣的胖友，可以看看 [《【死磕 Spring】—— IoC 之 BeanDefinition 注册表：BeanDefinitionRegistry》](#) 文章。
- <1> 处，调用 #createBeanDefinitionLoader(BeanDefinitionRegistry registry, Object[] sources) 方法，创建 org.springframework.boot.BeanDefinitionLoader 对象。关于它，后续的文章，详细解析。
- <2> 处，设置 loader 的属性。
- <3> 处，调用 BeanDefinitionLoader#load() 方法，执行 BeanDefinition 加载。关于这一块，胖友感兴趣，先简单看看 [《【死磕 Spring】—— IoC 之加载 BeanDefinition》](#) 文章。

<8> 处，调用 SpringApplicationRunListeners#contextLoaded(ConfigurableApplicationContext context) 方法，通知 SpringApplicationRunListener 的数组，Spring 容器加载完成。

2.2.4 refreshContext

#refreshContext(ConfigurableApplicationContext context) 方法，启动（刷新） Spring 容器。代码如下：

```
// SpringApplication.java

/**
 * 是否注册 ShutdownHook 钩子
 */
private boolean registerShutdownHook = true;

private void refreshContext(ConfigurableApplicationContext context) {
    // <1> 开启（刷新） Spring 容器
    refresh(context);
    // <2> 注册 ShutdownHook 钩子
    if (this.registerShutdownHook) {
        try {
            context.registerShutdownHook();
        } catch (AccessControlException ex) {
            // Not allowed in some environments.
        }
    }
}
```

<1> 处，调用 #refresh(ApplicationContext applicationContext) 方法，开启（刷新） Spring 容器。代码如下：

```
// SpringApplication.java

protected void refresh(ApplicationContext applicationContext) {
    // 断言，判断 applicationContext 是 AbstractApplicationContext 的子类
    Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);
    // 启动（刷新） AbstractApplicationContext
    ((AbstractApplicationContext) applicationContext).refresh();
}
```

- 调用 AbstractApplicationContext#refresh() 方法，启动（刷新） Spring 容器。
 - AbstractApplicationContext#refresh() 方法，胖友可以看看 [《【死磕 Spring】——ApplicationContext 相关接口架构分析》](#) 文章。
 - 这里，可以触发 Spring Boot 的自动配置的功能。关于这一块，我们会在下一篇文章，详细解析。

<2> 处，调用 ConfigurableApplicationContext#registerShutdownHook() 方法，注册 ShutdownHook 钩子。这个钩子，主要用于 Spring 应用的关闭时，销毁相应的 Bean 们。

2.2.5 callRunners

#callRunners(ApplicationContext context, ApplicationArguments args) 方法，调用 ApplicationRunner 或者 CommandLineRunner 的运行方法。代码如下：

```
// SpringApplication.java

private void callRunners(ApplicationContext context, ApplicationArguments args) {
    // <1> 获得所有 Runner 们
```

```

List<Object> runners = new ArrayList<>();
// <1.1> 获得所有 ApplicationRunner Bean 们
runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
// <1.2> 获得所有 CommandLineRunner Bean 们
runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
// <1.3> 排序 runners
AnnotationAwareOrderComparator.sort(runners);
// <2> 遍历 Runner 数组，执行逻辑
for (Object runner : new LinkedHashSet<>(runners)) {
    if (runner instanceof ApplicationRunner) {
        callRunner((ApplicationRunner) runner, args);
    }
    if (runner instanceof CommandLineRunner) {
        callRunner((CommandLineRunner) runner, args);
    }
}
}

```

<1> 处，获得所有 Runner 们，并进行排序。

<2> 处，遍历 Runner 数组，执行逻辑。代码如下：

```

// SpringApplication.java

private void callRunner(ApplicationRunner runner, ApplicationArguments args) {
    try {
        (runner).run(args);
    } catch (Exception ex) {
        throw new IllegalStateException("Failed to execute ApplicationRunner", ex);
    }
}

private void callRunner(CommandLineRunner runner, ApplicationArguments args) {
    try {
        (runner).run(args.getSourceArgs());
    } catch (Exception ex) {
        throw new IllegalStateException("Failed to execute CommandLineRunner", ex);
    }
}
}

```

关于 Runner 功能的使用，可以看看 [《ApplicationRunner 接口》](#) 和 [《CommandLineRunner 接口》](#) 文档。

3. SpringApplicationRunListeners

org.springframework.boot.SpringApplicationRunListeners，SpringApplicationRunListener 数组的封装。代码如下：

```

// SpringApplicationRunListeners.java

class SpringApplicationRunListeners {

    private final Log log;
}

```

```

/**
 * SpringApplicationRunListener 数组
 */
private final List<SpringApplicationRunListener> listeners;

SpringApplicationRunListeners(Log log,
    Collection<? extends SpringApplicationRunListener> listeners) {
    this.log = log;
    this.listeners = new ArrayList<>(listeners);
}

public void starting() {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.starting();
    }
}

public void environmentPrepared(ConfigurableEnvironment environment) {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.environmentPrepared(environment);
    }
}

public void contextPrepared(ConfigurableApplicationContext context) {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.contextPrepared(context);
    }
}

public void contextLoaded(ConfigurableApplicationContext context) {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.contextLoaded(context);
    }
}

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

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

public void failed(ConfigurableApplicationContext context, Throwable exception) {
    for (SpringApplicationRunListener listener : this.listeners) {
        callFailedListener(listener, context, exception);
    }
}

private void callFailedListener(SpringApplicationRunListener listener, ConfigurableApplicationContext context, Th
    try {
        listener.failed(context, exception);
    } catch (Throwable ex) {
        if (exception == null) {
            ReflectionUtils.rethrowRuntimeException(ex);

```

```

    }
    if (this.log.isDebugEnabled()) {
        this.log.error("Error handling failed", ex);
    } else {
        String message = ex.getMessage();
        message = (message != null) ? message : "no error message";
        this.log.warn("Error handling failed (" + message + ")");
    }
}
}
}
}
}

```

3.1 SpringApplicationRunListener

`org.springframework.boot.SpringApplicationRunListener` ， `SpringApplication` 运行的监听器接口。代码如下：

```

// SpringApplicationRunListener.java

public interface SpringApplicationRunListener {

    /**
     * Called immediately when the run method has first started. Can be used for very
     * early initialization.
     */
    void starting();

    /**
     * Called once the environment has been prepared, but before the
     * {@link ApplicationContext} has been created.
     * @param environment the environment
     */
    void environmentPrepared(ConfigurableEnvironment environment);

    /**
     * Called once the {@link ApplicationContext} has been created and prepared, but
     * before sources have been loaded.
     * @param context the application context
     */
    void contextPrepared(ConfigurableApplicationContext context);

    /**
     * Called once the application context has been loaded but before it has been
     * refreshed.
     * @param context the application context
     */
    void contextLoaded(ConfigurableApplicationContext context);

    /**
     * The context has been refreshed and the application has started but
     * {@link CommandLineRunner CommandLineRunners} and {@link ApplicationRunner
     * ApplicationRunners} have not been called.
     * @param context the application context.
     * @since 2.0.0
     */
    void started(ConfigurableApplicationContext context);
}

```

```

/**
 * Called immediately before the run method finishes, when the application context has
 * been refreshed and all {@link CommandLineRunner CommandLineRunners} and
 * {@link ApplicationRunner ApplicationRunners} have been called.
 * @param context the application context.
 * @since 2.0.0
 */
void running(ConfigurableApplicationContext context);

/**
 * Called when a failure occurs when running the application.
 * @param context the application context or {@code null} if a failure occurred before
 * the context was created
 * @param exception the failure
 * @since 2.0.0
 */
void failed(ConfigurableApplicationContext context, Throwable exception);
}

```

目前，SpringApplicationRunListener 的实现类，只有 EventPublishingRunListener 类。

3.2 EventPublishingRunListener

org.springframework.boot.context.event.EventPublishingRunListener，实现 SpringApplicationRunListener、Ordered 接口，将 SpringApplicationRunListener 监听到的事件，转换成对应的 SpringApplicationEvent 事件，发布到监听器们。

代码如下：

```

// EventPublishingRunListener.java

public class EventPublishingRunListener implements SpringApplicationRunListener, Ordered {

    /**
     * Spring 应用
     */
    private final SpringApplication application;

    /**
     * 参数集合
     */
    private final String[] args;

    /**
     * 事件广播器
     */
    private final SimpleApplicationEventMulticaster initialMulticaster;

    public EventPublishingRunListener(SpringApplication application, String[] args) {
        this.application = application;
        this.args = args;
        // 创建 SimpleApplicationEventMulticaster 对象
        this.initialMulticaster = new SimpleApplicationEventMulticaster();
        // 添加应用的监听器们，到 initialMulticaster 中
        for (ApplicationListener<?> listener : application.getListeners()) {
            this.initialMulticaster.addApplicationListener(listener);
        }
    }
}

```



```

    }
}

@Override
public int getOrder() {
    return 0;
}

@Override
public void starting() {
    this.initialMulticaster.multicastEvent(new ApplicationStartingEvent(this.application, this.args));
}

@Override // ApplicationEnvironmentPreparedEvent
public void environmentPrepared(ConfigurableEnvironment environment) {
    this.initialMulticaster.multicastEvent(new ApplicationEnvironmentPreparedEvent(this.application, this.args, environment));
}

@Override // ApplicationContextInitializedEvent
public void contextPrepared(ConfigurableApplicationContext context) {
    this.initialMulticaster.multicastEvent(new ApplicationContextInitializedEvent(this.application, this.args, context));
}

@Override // ApplicationPreparedEvent
public void contextLoaded(ConfigurableApplicationContext context) {
    for (ApplicationListener<?> listener : this.application.getListeners()) {
        if (listener instanceof ApplicationContextAware) {
            ((ApplicationContextAware) listener).setApplicationContext(context);
        }
        context.addApplicationListener(listener);
    }
    this.initialMulticaster.multicastEvent(new ApplicationPreparedEvent(this.application, this.args, context));
}

@Override // ApplicationStartedEvent
public void started(ConfigurableApplicationContext context) {
    context.publishEvent(new ApplicationStartedEvent(this.application, this.args, context));
}

@Override
public void running(ConfigurableApplicationContext context) {
    context.publishEvent(new ApplicationReadyEvent(this.application, this.args, context));
}

@Override // ApplicationFailedEvent
public void failed(ConfigurableApplicationContext context, Throwable exception) {
    ApplicationFailedEvent event = new ApplicationFailedEvent(this.application, this.args, context, exception);
    if (context != null && context.isActive()) {
        // Listeners have been registered to the application context so we should
        // use it at this point if we can
        context.publishEvent(event);
    } else {
        // An inactive context may not have a multicaster so we use our multicaster to
        // call all of the context's listeners instead
        if (context instanceof AbstractApplicationContext) {
            for (ApplicationListener<?> listener : ((AbstractApplicationContext) context)
                .getApplicationListeners()) {
                this.initialMulticaster.addApplicationListener(listener);
            }
        }
    }
}

```

```

        this.initialMulticaster.setErrorHandler(new LoggingErrorHandler());
        this.initialMulticaster.multicastEvent(event);
    }
}

private static class LoggingErrorHandler implements ErrorHandler {

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

    @Override
    public void handleError(Throwable throwable) {
        logger.warn("Error calling ApplicationEventListener", throwable);
    }

}
}

```

代码比较简单，胖友自己瞅瞅就明白了。

通过这样的方式，可以很方便的将 `SpringApplication` 启动的各种事件，方便的修改成对应的 `SpringApplicationEvent` 事件。这样，我们就可以不需要修改 `SpringApplication` 的代码。或者说，我们认为 `EventPublishingRunListener` 是一个“转换器”。

关于 `Spring Boot` 的事件，可以看看 [《事件监听与发布》](#) 文章。

666. 彩蛋

整块代码略微有点长，胖友一定一定一定自己调试下。总的来说，逻辑并不复杂，是吧？是吧！

参考和推荐如下文章：

快乐崇拜 [《Spring Boot 源码深入分析》](#)

需要付费 3 块钱。你看，芳芳为了写好源码解析，还是去学习了下别人的博客。

老田 [《Spring Boot 2.0 系列文章\(七\)：SpringApplication 深入探索》](#)

oldflame-Jm [《Spring boot源码分析-SpringApplication启动（1）》](#)

dm_vincent

- [《\[Spring Boot\] 1. Spring Boot 启动过程源码分析》](#)
- [《\[Spring Boot\] 2. Spring Boot 启动过程定制化》](#)

一个努力的码农

设计 `Spring Framework` 的部分，他也写了一些~

- [《spring boot 源码解析2-SpringApplication初始化》](#)
- [《spring boot 源码解析3-SpringApplication#run》](#)
- [《spring boot 源码解析4-SpringApplication#run第4步》](#)
- [《spring boot 源码解析5-SpringApplication#run第5步》](#)
- [《spring boot 源码解析6-SpringApplication#run第6步》](#)
- [《spring boot 源码解析7-SpringApplication#run第7步》](#)
- [《spring boot 源码解析8-SpringApplication#run第8步》](#)
- [《spring boot 源码解析9-SpringApplication#run第9步》](#)
- [《spring boot 源码解析10-SpringApplication#run第10-13步》](#)

文章目录

1. [1. 1. 概述](#)
2. [2. 2. SpringApplication](#)
 1. [2. 1. 2. 1 构造方法](#)
 1. [2. 1. 1. 2. 1. 1 getSpringFactoriesInstances](#)
 2. [2. 2. 2. 2 run](#)
 1. [2. 2. 1. 2. 2. 1 prepareEnvironment](#)
 2. [2. 2. 2. 2. 2 createApplicationContext](#)
 3. [2. 2. 3. 2. 2. 3 prepareContext](#)
 4. [2. 2. 4. 2. 2. 4 refreshContext](#)
 5. [2. 2. 5. 2. 2. 5 callRunners](#)
3. [3. 3. SpringApplicationRunListeners](#)
 1. [3. 1. 3. 1 SpringApplicationRunListener](#)
 2. [3. 2. 3. 2 EventPublishingRunListener](#)
4. [4. 666. 彩蛋](#)

2014 - 2023 芋道源码 |
总访客数 次 && 总访问量 次
[回到首页](#)