

SpringApplication to Fire in Detail

• 简介

- 使用版本是 **Spring-Boot 2.1.2.RELEASE**
- 每一个SpringBoot程序都可以通过一个简单的main方法去启动，如：

```
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringBootHowToRunApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringBootHowToRunApplication.class, args);
11     }
12
13 }
```

- 其中最重要的包含了两部分：
 - 第一个是@SpringBootApplication注解
 - 第二个是 SpringApplication.run() 方法，这个方法中封装了整个SpringBoot程序启动的所有细节
- 所以，这篇文章的核心就是深入 SpringApplication.run() 方法，理解整个SpringBoot 是如何启动的
- 在SpringApplication类中定义了整个启动过程分为了2个大步骤：
 - **准备阶段**
 - **运行阶段**

• 准备阶段

- 所谓SpringApplication的准备阶段，其实就是SpringApplication对象的构建过程：

```
252 /**
253  * Create a new {@link SpringApplication} instance. The application context will load
254  * beans from the specified primary sources (see {@link SpringApplication class-level}
255  * documentation for details. The instance can be customized before calling
256  * {@link #run(String...)}).
257  * @param resourceLoader the resource loader to use
258  * @param primarySources the primary bean sources
259  * @see #run(Class, String[])
260  * @see #setSources(Set)
261  */
262 @SuppressWarnings({ "unchecked", "rawtypes" })
263 @public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
264     this.resourceLoader = resourceLoader;
265     Assert.notNull(primarySources, "PrimarySources must not be null");
266     this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
267     this.webApplicationType = WebApplicationType.deduceFromClasspath();
268     setInitializers((Collection) getSpringFactoriesInstances(
269         ApplicationContextInitializer.class));
270     setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
271     this.mainApplicationClass = deduceMainApplicationClass();
272 }
```

- 这里使用了构造器模式，详情参看设计模式部分的构造器模式
- 在准备阶段主要分为以下几个步骤：
 - 1. 保存资源加载器(ResourceLoader)
 - 2. 判断初始资源(primarySources)，确保初始资源不能为空

- 3. 保存初始资源，初始资源中包含了Bean的配置信息，注意此处LinkedHashSet的使用
- 4. 推断SpringBoot应用的类型，在2.x版本中引入了Reactive WebFlux，所以总共有三种类型：WebMVC、WebFlux、None

- 这三种类型封装在了一个WebApplicationType类中
- WebApplicationType类中有一个方法deduceFromClasspath()，通过该方法来推断应用的类型

```

63     static WebApplicationType deduceFromClasspath() {
64         if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, classLoader: null)
65             && !ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS, classLoader: null)
66             && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, classLoader: null)) {
67             return WebApplicationType.REACTIVE;
68         }
69         for (String className : SERVLET_INDICATOR_CLASSES) {
70             if (!ClassUtils.isPresent(className, classLoader: null)) {
71                 return WebApplicationType.NONE;
72             }
73         }
74         return WebApplicationType.SERVLET;
75     }

```

- 此处，注意一下类的职责问题，WebApplicationType封装了应用的类型，所以理所应当的由这个类来推断应用类型
 - 再次提出，类是属性和方法的集合，每个类应该有自己的职责，不能推卸自己责任，也不能抢别的类的任务

- 5. 设置初始化器，这些初始化器都实现了ApplicationContextInitializer接口
- ApplicationContextInitializer接口

```

34     * @author Chris Beams
35     * @since 3.1
36     * @param <C> the application context type
37     * @see org.springframework.web.context.ContextLoader#customizeContext
38     * @see org.springframework.web.context.ContextLoader#CONTEXT_INITIALIZER_CLASSES_PARAM
39     * @see org.springframework.web.servlet.FrameworkServlet#setContextInitializerClasses
40     * @see org.springframework.web.servlet.FrameworkServlet#applyInitializers
41     */
42     public interface ApplicationContextInitializer<C extends ConfigurableApplicationContext> {
43
44         /**
45          * Initialize the given application context.
46          * @param applicationContext the application to configure
47          */
48         void initialize(C applicationContext);
49     }
50

```

- 如何设置的呢？通过一个getSpringFactoriesInstances()方法

```

423     @private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
424         Class<?>[] parameterTypes, Object... args) {
425         ClassLoader classLoader = getClassLoader();
426         // Use names and ensure unique to protect against duplicates
427         Set<String> names = new LinkedHashSet<>();
428         SpringFactoriesLoader.loadFactoryNames(type, classLoader);
429         List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
430             classLoader, args, names);
431         AnnotationAwareOrderComparator.sort(instances);
432         return instances;
433     }

```

- getSpringFactoriesInstances()中最重要的一步：SpringFactoriesLoader.loadFactoryNames()
 - 在loadFactoryNames()方法中，去所有的jar包的类路径下找到META-INF/spring.factories文件，然后保存到一个Set集合中
 - 具体保存哪些类，由传入的类型决定，此处就是保存实现ApplicationContextInitializer接口的所有类的全类名

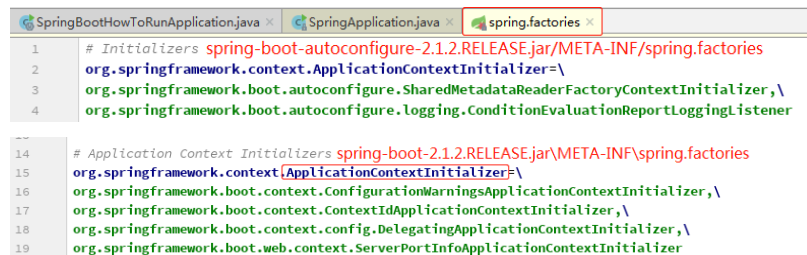
- 然后调用createSpringFactoriesInstances()，这个方法中，通过反射调用的机制创建出符合类型的所有类实例
- 最后对这些创建好的类实例进行排序
 - 这里就涉及到整个Spring框架中比较重要的排序机制，也正是由于这一点，SpringBoot的扩展变得复杂，因为我们在扩展SpringBoot时，必须要知道内置的各个组件的执行顺序，而组件往往很多，我们把自定义的组件放在什么位置去执行，是一件很考究的事情
- 我们此时加载了多少个ApplicationContextInitializer实例呢？如下：

```

▼ ⓘ initializers = {ArrayList@1851} size = 7
  > 0 = {DelegatingApplicationContextInitializer@1854}
  > 1 = {SharedMetadataReaderFactoryContextInitializer@1855}
  > 2 = {ContextIdApplicationContextInitializer@1856}
  > 3 = {HelloApplicationContextInitializer@1857} 自定义
  > 4 = {ConfigurationWarningsApplicationContextInitializer@1858}
  > 5 = {ServerPortInfoApplicationContextInitializer@1859}
  > 6 = {ConditionEvaluationReportLoggingListener@1860}

```

- 我们自定义了一个HelloApplicationContextInitializer，排在第四的位置
- 其他的6个是SpringBoot自动配置的，这些自动配置的描述文件又在何处呢？



```

SpringBootHowToRunApplication.java x SpringApplication.java x spring.factories x
1 # Initializers spring-boot-autoconfigure-2.1.2.RELEASE.jar/META-INF/spring.factories
2 org.springframework.context.ApplicationContextInitializer=\
3 org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
4 org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener
...
14 # Application Context Initializers spring-boot-2.1.2.RELEASE.jar\META-INF\spring.factories
15 org.springframework.context.ApplicationContextInitializer=\
16 org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\
17 org.springframework.boot.context.ContextIdApplicationContextInitializer,\
18 org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\
19 org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer

```

- 6. 设置监听器，这些监听器都实现了ApplicationListener接口
 - ApplicationListener接口

```

36 @FunctionalInterface
37 public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {
38
39     /**
40      * Handle an application event.
41      * @param event the event to respond to
42      */
43     void onApplicationEvent(E event);
44
45 }

```

- 如何设置这些监听器呢？依然是通过getSpringFactoriesInstances()方法
 - 设置的流程就和上面的一模一样了
 - 现在，我们又知道什么了呢？
 - 如果再次遇到getSpringFactoriesInstances()，我们就知道这是框架自动去META-INF/spring.factories寻找并创建加载对应的类
 - 我们此时加载了多少个ApplicationListener实例呢？如下：

```

▼ f listeners = (ArrayList@1881) size = 10
  > 0 = {ConfigFileApplicationListener@1883}
  > 1 = {AnsiOutputApplicationListener@1884}
  > 2 = {LoggingApplicationListener@1885}
  > 3 = {ClasspathLoggingApplicationListener@1886}
  > 4 = {BackgroundPreinitializer@1887}
  > 5 = {DelegatingApplicationListener@1888}
  > 6 = {ParentContextCloserApplicationListener@1889}
  > 7 = {ClearCachesApplicationListener@1890}
  > 8 = {FileEncodingApplicationListener@1891}
  > 9 = {LiquibaseServiceLocatorApplicationListener@1892}

```

- SpringBoot自动配置了10个事件监听器，这些配置文件的描述文件又在什么位置呢？

```

21 # Application Listeners spring-boot-2.1.2.RELEASE.jar\META-INF\spring.factories
22 org.springframework.context.ApplicationListener=\
23 org.springframework.boot.ClearCachesApplicationListener,\
24 org.springframework.boot.builder.ParentContextCloserApplicationListener,\
25 org.springframework.boot.context.FileEncodingApplicationListener,\
26 org.springframework.boot.context.config.AnsiOutputApplicationListener,\
27 org.springframework.boot.context.config.ConfigFileApplicationListener,\
28 org.springframework.boot.context.config.DelegatingApplicationListener,\
29 org.springframework.boot.context.logging.ClasspathLoggingApplicationListener,\
30 org.springframework.boot.context.logging.LoggingApplicationListener,\
31 org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener

```

```

5 spring-boot-autoconfigure-2.1.2.RELEASE.jar\META-INF\spring.factories
6 # Application Listeners
7 org.springframework.context.ApplicationListener=\
8 org.springframework.boot.autoconfigure.BackgroundPreinitializer

```

- 7. 推断应用主程序类
 - 根据栈中的信息匹配main，然后获得主程序类

```

274 private Class<?> deduceMainApplicationClass() {
275     try {
276         StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
277         for (StackTraceElement stackTraceElement : stackTrace) {
278             if ("main".equals(stackTraceElement.getMethodName())) {
279                 return Class.forName(stackTraceElement.getClassName());
280             }
281         }
282     }
283     catch (ClassNotFoundException ex) {
284         // Swallow and continue
285     }
286     return null;
287 }

```

- 最终获得就是SpringBootHowToRunApplication类，因为程序就是从这个类中main方法中进入的

运行阶段

- 所谓SpringApplication的运行阶段，其实就是SpringApplication对象执行run()方法的过程：

```

289  /**
290   * Run the Spring application, creating and refreshing a new
291   * {@link ApplicationContext}.
292   * @param args the application arguments (usually passed from a Java main method)
293   * @return a running {@link ApplicationContext}
294   */
295  @ public ConfigurableApplicationContext run(String... args) {
296      Stopwatch stopWatch = new Stopwatch();
297      stopWatch.start();
298      ConfigurableApplicationContext context = null;
299      Collection<SpringBootExceptionHandler> exceptionReporters = new ArrayList<>();
300      configureHeadlessProperty();
301      SpringApplicationRunListeners listeners = getRunListeners(args);
302      listeners.starting();
303      try {
304          ApplicationArguments applicationArguments = new DefaultApplicationArguments(
305              args);
306          ConfigurableEnvironment environment = prepareEnvironment(listeners,
307              applicationArguments);
308          configureIgnoreBeanInfo(environment);
309          Banner printedBanner = printBanner(environment);
310          context = createApplicationContext();
311          exceptionReporters = getSpringFactoriesInstances(
312              SpringBootExceptionHandler.class,
313              new Class[] { ConfigurableApplicationContext.class }, context);
314          prepareContext(context, environment, listeners, applicationArguments,
315              printedBanner);
316          refreshContext(context);
317          afterRefresh(context, applicationArguments);
318          stopWatch.stop();
319          if (this.logStartupInfo) {
320              new StartupInfoLogger(this.mainApplicationClass)
321                  .logStarted(getApplicationLog(), stopWatch);
322          }
323          listeners.started(context);
324          callRunners(context, applicationArguments);
325      }
326
327      catch (Throwable ex) {
328          handleRunFailure(context, ex, exceptionReporters, listeners);
329          throw new IllegalStateException(ex);
330      }
331
332      try {
333          listeners.running(context);
334      }
335      catch (Throwable ex) {
336          handleRunFailure(context, ex, exceptionReporters, listeners: null);
337          throw new IllegalStateException(ex);
338      }
339      return context;
340  }

```

- 此处使用了模板方法的设计模式，即框架规定了程序运行的骨架，并在骨架中预留了一些供应用开发人员扩展的抽象方法
 - “你不要调我，让我来调用你”是这个设计模式的精髓
- 在运行阶段主要分为以下几个步骤：
 - 1. 源码296-297行，生成一个StopWatch，这就是一个简单的计时器，不属于程序逻辑的一部分，仅仅是为了卡一下时间
 - 查看StopWatch类的源码，可以看到，它仅仅是为了提升代码的易读性和减少时间计算的可能错误，才没有使用Java自带的currentTimeMillis
 - 它在程序最开始的位置，先掐一个时间，stopWatch.start()
 - 2. 源码298行，声明一个ConfigurableApplicationContext，按照字面意思，这是一个可配置的应用容器
 - 3. 声明一个异常报告器，用于通知启动过程中产生的异常，命名为exceptionReporters。
 - 这个接口就是所谓的可以供应用开发人员扩展的接口，我们可以按照官方的操作去实现这个接口，完成启动过程中异常的自定义

- 4. 源码300行是和Java.AWT相关的信息配置
- 5. 通过getRunListeners()获取监听器，封装在SpringApplicationRunListeners对象中，这个对象命名为listeners
 - **这就是典型的观察者模式了**
 - SpringApplicationRunListeners 是一个集合对象，内置了一个List，用于保存所有的SpringApplicationRunListener
 - SpringApplicationRunListener接口很直白，就是用于监听SpringApplication.run()方法的执行，包含了7个方法，对应了7中不同的运行阶段：
 - starting()
 - environmentPrepared(ConfigurableEnvironment environment)
 - contextPrepared(ConfigurableApplicationContext context)
 - contextLoaded(ConfigurableApplicationContext context)
 - started(ConfigurableApplicationContext context)
 - running(ConfigurableApplicationContext context)
 - failed(ConfigurableApplicationContext context, Throwable exception)
 - getRunListeners()是如何获取所有的SpringApplicationRunListener对象的呢？
 - getRunListeners()方法调用了getSpringFactoriesInstances()，传入SpringApplicationRunListener.class
 - 这就很了然了，getSpringFactoriesInstances()去所有的jar包中读取META-INF/spring.factories文件，从中获取相应的全类名
 - 我们这里加载了多少个SpringApplicationRunListener实例呢？


```

              v listeners = {SpringApplicationRunListeners@1819}
              > log = {LogAdapter$Slf4jLocationAwareLog@1841}
              v listeners = {ArrayList@1842} size = 2
              > 0 = {EventPublishingRunListener@1846}
              > 1 = {HelloSpringApplicationRunListener@1847} 自定义
              
```
 - 我们自定义了一个HelloSpringApplicationRunListener，SpringBoot加载了一个EventPublishingRunListener
 - 这个EventPublishingRunListener的配置文件又在什么位置呢？


```

              5 spring-boot-2.1.2.RELEASE.jar/META-INF/spring.factories
              6 # Run Listeners
              7 org.springframework.boot.SpringApplicationRunListener=\
              8 org.springframework.boot.context.event.EventPublishingRunListener
              
```
- 6. listeners调用starting()，会循环地通知集合中所有的SpringApplicationRunListener启动starting()
- 7. 源码304行，对main方法传入的args参数进行包装，包装成一个ApplicationArguments类对象
- 8. prepareEnvironment()，准备环境
 - 在环境准备结束后，listeners循环通知所有的SpringApplicationRunListener启动environmentPrepared()
- 9. configureIgnoreBeanInfo()，将环境中忽略的bean的信息保存起来

- 10. printBanner(), 打印标语, 这是SpringBoot的彩蛋, 我们也可以自定义Banner
- 11. createApplicationContext(), 对上面第2步声明的ConfigurableApplicationContext对象进行赋值
 - 赋值的过程, 其实就是ConfigurableApplicationContext对象创建的过程
 - 先判断当前应用的类型, 是servlet, reactive还是none
 - 再使用spring的BeanUtils实例化对象: BeanUtils.instantiateClass()
- 12. 对第3步声明的exceptionReporters进行赋值
 - 赋值的过程其实就是调用了getSpringFactoriesInstances()方法, 传入SpringBootExceptionHandler.class
 - 轻车熟路, 这次加载了多少个SpringBootExceptionHandler实例呢?

```

v  exceptionReporters = {ArrayList@3449} size = 1
  v  0 = {FailureAnalyzers@3528}
    >  classLoader = {Launcher$AppClassLoader@3529}
    v  analyzers = {ArrayList@3530} size = 17
      >  0 = {BeanCurrentlyInCreationFailureAnalyzer@3536}
      >  1 = {BeanDefinitionOverrideFailureAnalyzer@3537}
      >  2 = {BeanNotOfRequiredTypeFailureAnalyzer@3538}
      >  3 = {BindFailureAnalyzer@3539}
      >  4 = {BindValidationFailureAnalyzer@3540}
      >  5 = {UnboundConfigurationPropertyFailureAnalyzer@3541}
      >  6 = {ConnectorStartFailureAnalyzer@3542}
      >  7 = {NoSuchMethodFailureAnalyzer@3543}
      >  8 = {NoUniqueBeanDefinitionFailureAnalyzer@3544}
      >  9 = {PortInUseFailureAnalyzer@3545}
      >  10 = {ValidationExceptionFailureAnalyzer@3546}
      >  11 = {InvalidConfigurationPropertyNameFailureAnalyzer@3547}
      >  12 = {InvalidConfigurationPropertyValueFailureAnalyzer@3548}
      >  13 = {NoSuchBeanDefinitionFailureAnalyzer@3549}
      >  14 = {DataSourceBeanCreationFailureAnalyzer@3550}
      >  15 = {HikariDriverConfigurationFailureAnalyzer@3551}
      >  16 = {NonUniqueSessionRepositoryFailureAnalyzer@3552}

```

- SpringBootExceptionHandler只有一个实例对象, 即: FailureAnalyzers
- 类似的, FailureAnalyzers中内置了一个List, 用于保存所有的FailureAnalyzer
- 从上面可以看出, SpringBoot一共向容器中加入了17个FailureAnalyzer对象
- 又是在哪里定义了这个SpringBootExceptionHandler和FailureAnalyzer呢?

```

9  spring-boot-2.1.2.RELEASE.jar\META-INF\spring.factories
10  # Error Reporters
11  org.springframework.boot.SpringBootExceptionHandler=\
12  org.springframework.boot.diagnostics.FailureAnalyzers

```

```

38 spring-boot-2.1.2.RELEASE.jar\META-INF\spring.factories
39 # Failure Analyzers
40 org.springframework.boot.diagnostics.FailureAnalyzer=\
41 org.springframework.boot.diagnostics.analyzer.BeanCurrentlyInCreationFailureAnalyzer,\
42 org.springframework.boot.diagnostics.analyzer.BeanDefinitionOverrideFailureAnalyzer,\
43 org.springframework.boot.diagnostics.analyzer.BeanNotOfRequiredTypeFailureAnalyzer,\
44 org.springframework.boot.diagnostics.analyzer.BindFailureAnalyzer,\
45 org.springframework.boot.diagnostics.analyzer.BindValidationFailureAnalyzer,\
46 org.springframework.boot.diagnostics.analyzer.UnboundConfigurationPropertyFailureAnalyzer,\
47 org.springframework.boot.diagnostics.analyzer.ConnectorStartFailureAnalyzer,\
48 org.springframework.boot.diagnostics.analyzer.NoSuchMethodFailureAnalyzer,\
49 org.springframework.boot.diagnostics.analyzer.NoUniqueBeanDefinitionFailureAnalyzer,\
50 org.springframework.boot.diagnostics.analyzer.PortInUseFailureAnalyzer,\
51 org.springframework.boot.diagnostics.analyzer.ValidationExceptionFailureAnalyzer,\
52 org.springframework.boot.diagnostics.analyzer.InvalidConfigurationPropertyNameFailureAnalyzer,\
53 org.springframework.boot.diagnostics.analyzer.InvalidConfigurationPropertyValueFailureAnalyzer

140 spring-boot-autoconfigure-2.1.2.RELEASE.jar\META-INF\spring.factories
141 # Failure analyzers
142 org.springframework.boot.diagnostics.FailureAnalyzer=\
143 org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefinitionFailureAnalyzer,\
144 org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAnalyzer,\
145 org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailureAnalyzer,\
146 org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFailureAnalyzer

```

• 13. prepareContext(), 准备上下文

- 这个方法中进行了大量的配置工作和初始化工作，值得深入研究

```

368 @ private void prepareContext(ConfigurableApplicationContext context,
369     ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
370     ApplicationArguments applicationArguments, Banner printedBanner) {
371     context.setEnvironment(environment);
372     postProcessApplicationContext(context);
373     applyInitializers(context);
374     listeners.contextPrepared(context);
375     if (this.logStartupInfo) {
376         logStartupInfo( isRoot: context.getParent() == null);
377         logStartupProfileInfo(context);
378     }
379     // Add boot specific singleton beans
380     ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
381     beanFactory.registerSingleton( S: "springApplicationArguments", applicationArguments);
382     if (printedBanner != null) {
383         beanFactory.registerSingleton( S: "springBootBanner", printedBanner);
384     }
385     if (beanFactory instanceof DefaultListableBeanFactory) {
386         ((DefaultListableBeanFactory) beanFactory)
387             .setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
388     }
389     // Load the sources
390     Set<Object> sources = getAllSources();
391     Assert.notEmpty(sources, message: "Sources must not be empty");
392     load(context, sources.toArray(new Object[0]));
393     listeners.contextLoaded(context);
394 }
395

```

- 将环境(environment)中的信息，保存到context(IOC容器)中，同时调用了后置处理器，并进行了初始化工作
- 接着回调所有的SpringApplicationRunListener的contextPrepared()方法
- 最后还回调了所有的SpringApplicationRunListener的contextLoaded()方法

• 14. refreshContext(context), 刷新IOC容器

- 这是整个过程中最为重要的一步，正是在这一步中SpringBoot调用了Spring-framework的refresh()方法
- 而这个refresh()方法中包含了大量的步骤，我们自己所声明的Bean都将在这一步得到实例化和初始化
- 其中包含了大量的BeanPostProcess机制，允许我们应用开发人员对Bean的实例化和初始化过程进行细粒度的控制
- 详情参看另外一篇文章《Spring 容器的初始化和创建过程》

• 15. afterRefresh(), 这又是一个空方法

- 再次提出，这就是模板方法设计模式中提到的，供开发人员扩展的地方

- 16. `stopWatch.stop()` 再次掐一次时间，用于记录程序启动过程中使用了多长时间
- 17. `listeners.started()`，回调所有的`SpringApplicationRunListener`的`started()`方法
- 18. `callRunners()`，加载并回调所有的runner

- 源码：

```

787 @ private void callRunners(ApplicationContext context, ApplicationArguments args) {
788     List<Object> runners = new ArrayList<>();
789     runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
790     runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
791     AnnotationAwareOrderComparator.sort(runners);
792     for (Object runner : new LinkedHashSet<>(runners)) {
793         if (runner instanceof ApplicationRunner) {
794             callRunner((ApplicationRunner) runner, args);
795         }
796         if (runner instanceof CommandLineRunner) {
797             callRunner((CommandLineRunner) runner, args);
798         }
799     }
800 }

```

- 第791行，对这些runner对象进行了排序
- 这次运行加载了多少个`ApplicationRunner`、`CommandLineRunner`呢？

```

v  runners = {ArrayList@5743} size = 2
  > 0 = {HelloApplicationRunner@5755} 自定义
  > 1 = {HelloCommandLineRunner@5756} 自定义

```

- 这两个都是我们自定义的。其实这里又是一个可扩展的点
- 这种`ApplicationRunner`、`CommandLineRunner`对象，是在IOC容器已经准备好的情况加入的
- 所以可以像我们普通的Bean一样，使用`@Bean`、`@Component`等注解的方式加入到容器中
- 19. 如果在启动过程中出现了异常，会被catch，然后通过`handleRunFailure()`去处理
 - 在`handleRunFailure()`方法中，回调了`listeners.failed()`方法
- 20. 在一个try-catch中尝试运行`listeners.running(context)`；
 - 同样的，出现异常后，在catch中回调`failed()`方法
- 最终返回一个`ConfigurableApplicationContext`对象，SpringBoot应用正常启动

• 总结

- 至此，大体上将SpringBoot的启动过程记录了一遍
- 值得深入理解的是，SpringBoot和SpringFramework中使用了优秀的设计理念
- 这些优秀的理念值得我们思考并借鉴

