理解 `SpringApplication`

`SpringApplication` 基本使用

`SpringApplication` 运行

```
SpringApplication.run(DiveInSpringBootApplication.class, args)
```

自定义 `SpringApplication`

通过 `SpringApplication` API 调整

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

通过 `SpringApplicationBuilder` API 调整

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

`SpringApplication` 准备阶段

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

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。

推断 Web 应用类型

根据当前应用 ClassPath 中是否存在相关实现类来推断 Web 应用的类型，包括：

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

> 参考方法：`org.springframework.boot.SpringApplication#deduceWebApplicationType`

```
    private WebApplicationType deduceWebApplicationType() {
        if (ClassUtils.isPresent(REACTIVE_WEB_ENVIRONMENT_CLASS, null)
                && !ClassUtils.isPresent(MVC_WEB_ENVIRONMENT_CLASS, null)) {
            return WebApplicationType.REACTIVE;
        }
        for (String className : WEB_ENVIRONMENT_CLASSES) {
            if (!ClassUtils.isPresent(className, null)) {
                return WebApplicationType.NONE;
            }
        }
        return WebApplicationType.SERVLET;
    }
```

推断引导类（Main Class）

根据 Main 线程执行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;
    }
```

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) {
   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

SpringApplication 运行阶段

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`

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	Spring Boot 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

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`

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：`StandardEnvironment`
- Web Servlet：`StandardServletEnvironment`
- 非 Web：`StandardEnvironment`