Spring Application 课程（Spring Boot-2.1.2.RELEASE）

1. 自定义SpringApplication应用启动

Spring Application 是Sping Boot 驱动Spring应用上下文的引导类。

@SpringBootApplication 标注当前的一些功能

等同于：@SpringBootApplication

-> @SpringBootConfiguration

-> @Configuration

->@Component

如：

|  |
| --- |
| @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(  excludeFilters = {@Filter(  type = FilterType.CUSTOM,  classes = {TypeExcludeFilter.**class**} ), @Filter(  type = FilterType.CUSTOM,  classes = {AutoConfigurationExcludeFilter.**class**} )} ) **public** @**interface** SpringBootApplication {  …  } |

@ComponentScan ：Spring Framewokr 3.1版本引入。

@EnableAutoConfiguration ：激活自动装配。@Enable开头

* @EnableWebMvc
* @EnableTransactionManager
* @EnableAspectJAutoProxy
* @EnableAsync

@SpringBootConfiguration：等价于@Configuration->Configuration Class 注解

XML配置文件驱动：ClasspathXmlApplicationContext

注解驱动：AnnotationConfigApplicationContext->查找Bean Definition 如：@Bean、@Configuration

zhouxuejing10975

@Component 的“派生性”等同于注解的继承

@Component ->@ComponentScan

处理类：ConfigurationClassParser

扫描类：ClassPathBeanDefinitionScanner

* ClassPathScanningCandidateComponentProvider

|  |
| --- |
| **protected void** registerDefaultFilters() {  **this**.includeFilters.add(**new** AnnotationTypeFilter(Component.**class**));  …  } |

Dubbo @Service->2.5.7->new AnnotationTypeFilter(Service.class)

Spring 注解编程模型

* @Service

|  |
| --- |
| @Component **public** @**interface** Service {  …  } |

* @Repository

|  |
| --- |
| @Component **public** @**interface** Repository {  …  } |

* @Controller

|  |
| --- |
| @Component **public** @**interface** Controller {  …  } |

* @Configuration

|  |
| --- |
| @Component **public** @**interface** Configuration {  …  } |

* @RestController

|  |
| --- |
| @Component **public** @**interface** RestController {  …  } |

Spring 模式注解：Stereotype Annotation

SpringApplication

|  |
| --- |
| *//SpringApplication.run(MicroserviceProjectApplication.class, args);*  SpringApplication springApplication = **new** SpringApplication(MicroserviceProjectApplication.**class**); Map<String, Object> properties = **new** LinkedHashMap<String, Object>(); properties.put(**"server.port"**, 0); springApplication.setDefaultProperties(properties); springApplication.run(args); |

SpringApplicationBuilder

|  |
| --- |
| **new** SpringApplicationBuilder(MicroserviceProjectApplication.**class**)*// Fluent API  //单元测试是 PORT=随机端口* .properties(**"server.port=0"**) *//随机向OS要可用端口* .run(args); |

Spring Boot引导实例

|  |
| --- |
| @SpringBootApplication **public class** MicroserviceProjectApplication {   **public static void** main(String[] args) {SpringApplication springApplication = **new** SpringApplication(MicroserviceProjectApplication.**class**);  Map<String, Object> properties = **new** LinkedHashMap<String, Object>();  properties.put(**"server.port"**, 0);  springApplication.setDefaultProperties(properties);  ConfigurableApplicationContext context = springApplication.run(args);   System.***out***.println(context.getBean(MicroserviceProjectApplication.**class**));}  } |

Spring注解驱动实例

|  |
| --- |
| @SpringBootApplication **public class** SpringAnnotationDemo {   **public static void** main(String[] args) {  AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext();  context.register(SpringAnnotationDemo.**class**);  *//上下文启动* context.refresh();   System.***out***.println(context.getBean(MicroserviceProjectApplication.**class**));  } } |

调整实例为非Web程序：

|  |
| --- |
| @SpringBootApplication **public class** MicroserviceProjectApplication {   **public static void** main(String[] args) { *//设置为非Web应用* SpringApplication springApplication = **new** SpringApplication(MicroserviceProjectApplication.**class**);  Map<String, Object> properties = **new** LinkedHashMap<String, Object>();  properties.put(**"server.port"**, 0);  springApplication.setDefaultProperties(properties);  *//设置为非Web应用* springApplication.setWebApplicationType(WebApplicationType.***NONE***);  ConfigurableApplicationContext context = springApplication.run(args);   *//输出当前Spring Boot应用的ApplicationContext的类名* System.***out***.println(**"当前Spring应用上下文的类"**+context.getClass().getName());  }  } |

输出结果：

当前Spring应用上下文的类org.springframework.context.annotation.AnnotationConfigApplicationContext

1. 配置Spring Boot源
2. Spring Application 类型推断

|  |
| --- |
| SpringApplication -> WebApplicationType.deduceFromClasspath() |

WebApplicationType. NONE ：非web应用

判断依据：

* **Servlet 不存在**
* **Spring 应用上下文ConfigurableWebApplicationContext** 不存在

**->**spring-boot-starter-web 不存在

**-**> spring-boot-starter-webflux 不存在

WebApplicationType. REACTIVE：WebFlux

判断依据：

* **DispatcherHandler 存在**

->spring-boot-starter-webflux 存在

* **Servlet 不存在**

-> spring-boot-starter-web 不存在

WebApplicationType. SERVLET：Spring MVC

* spring-boot-starter-web 存在

人工干预类型：

|  |
| --- |
| springApplication.setWebApplicationType(WebApplicationType.***NONE***); |

根据WebApplicationType 创建不同的上下文

SpringApplication

|  |
| --- |
| **protected** ConfigurableApplicationContext createApplicationContext() {  Class<?> contextClass = **this**.applicationContextClass;  **if** (contextClass == **null**) {  **try** {  **switch**(**this**.webApplicationType) {  **case** SERVLET:  contextClass = Class.forName(**"org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext"**);  **break**;  **case** REACTIVE:  contextClass = Class.forName(**"org.springframework.boot.web.reactive.context.AnnotationConfigReactiveWebServerApplicationContext"**);  **break**;  **default**:  contextClass = Class.forName(**"org.springframework.context.annotation.AnnotationConfigApplicationContext"**);  }  } **catch** (ClassNotFoundException var3) {  **throw new** IllegalStateException(**"Unable create a default ApplicationContext, please specify an ApplicationContextClass"**, var3);  }  }   **return** (ConfigurableApplicationContext)BeanUtils.instantiateClass(contextClass); } |

引导Spring Boot 应用：

|  |
| --- |
| **public** ConfigurableApplicationContext run(String... args) {  …  ConfigurableApplicationContext context = **null**;  …  context = **this**.createApplicationContext();  exceptionReporters = **this**.getSpringFactoriesInstances(SpringBootExceptionReporter.**class**, **new** Class[]{ConfigurableApplicationContext.**class**}, context);  **this**.prepareContext(context, environment, listeners, applicationArguments, printedBanner);  **this**.refreshContext(context);  **this**.afterRefresh(context, applicationArguments);  …  } |

1. Spring Boot 事件
2. Spring 内部事件

ContextRefreshedEvent事件

->ApplicationContextEvent

->ApplicationEvent

refresh()->finishRefresh()->**this**.publishEvent((ApplicationEvent)(**new** ContextRefreshedEvent(**this**)));

ContextClosedEvent 事件

->ApplicationContextEvent

->ApplicationEvent

close()->doClose()->**this**.publishEvent((ApplicationEvent)(**new** ContextClosedEvent(**this**)));

1. 自定义事件：

PayloadApplicationEvent事件

Spring 事件都是ApplicationEvent 类型。

1. 发送事件

ApplicationEventMulticaster.multicastEvent((ApplicationEvent)applicationEvent, eventType);

事件广播器初始化：

refresh()->initApplicationEventMulticaster()

|  |
| --- |
| **protected void** initApplicationEventMulticaster() {  ConfigurableListableBeanFactory beanFactory = **this**.getBeanFactory();  **if** (beanFactory.containsLocalBean(**"applicationEventMulticaster"**)) {  **this**.applicationEventMulticaster = (ApplicationEventMulticaster)beanFactory.getBean(**"applicationEventMulticaster"**, ApplicationEventMulticaster.**class**);  **if** (**this**.logger.isTraceEnabled()) {  **this**.logger.trace(**"Using ApplicationEventMulticaster ["** + **this**.applicationEventMulticaster + **"]"**);  }  } **else** {  **this**.applicationEventMulticaster = **new** SimpleApplicationEventMulticaster(beanFactory);  beanFactory.registerSingleton(**"applicationEventMulticaster"**, **this**.applicationEventMulticaster);  **if** (**this**.logger.isTraceEnabled()) {  **this**.logger.trace(**"No 'applicationEventMulticaster' bean, using ["** + **this**.applicationEventMulticaster.getClass().getSimpleName() + **"]"**);  }  }  } |

总结：

Spring 的事件类型：ApplicationEvent

Spring 的事件监听器：ApplicationListener

Spring的事件广播器：ApplicationEventMulticaster

实现：SimpleApplicationEventMulticaster

Sprng Boot 事件演示：

|  |
| --- |
| @SpringBootApplication **public class** SpringBootEventDemo {   **public static void** main(String[] args) {  **new** SpringApplicationBuilder(SpringBootEventDemo.**class**)  .listeners(**new** ApplicationListener<ApplicationEvent>() {  @Override  **public void** onApplicationEvent(ApplicationEvent applicationEvent) {  System.***out***.println(**"监听事件："** + applicationEvent);  }  })  .run(args).close(); } } |

相关事件： ApplicationStartingEvent(1)  
 ApplicationEnvironmentPreparedEvent(2)  
 ApplicationContextInitializedEvent(3)  
 ApplicationPreparedEvent(4)  
 ContextRefreshedEvent  
 ServletWebServerInitializedEvent  
 ApplicationStartedEvent(5)  
 ApplicationReadyEvent(6)  
 ContextClosedEvent

ApplicationFailedEvent(启动失败事件)(7)

EventPublishingRunListener 发布以上事件

Spring Boot事件监听：

|  |
| --- |
| *# Application Listeners* **org.springframework.context.ApplicationListener**=**\ org.springframework.boot.ClearCachesApplicationListener,\ org.springframework.boot.builder.ParentContextCloserApplicationListener,\ org.springframework.boot.context.FileEncodingApplicationListener,\ org.springframework.boot.context.config.AnsiOutputApplicationListener,\ org.springframework.boot.context.config.ConfigFileApplicationListener,\ org.springframework.boot.context.config.DelegatingApplicationListener,\ org.springframework.boot.context.logging.ClasspathLoggingApplicationListener,\ org.springframework.boot.context.logging.****LoggingApplicationListener,\ org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener** |

ConfigFileApplicationListener 监听ApplicationEnvironmentPreparedEvent事件进而加载

application.properties 和application.yml配置文件

Spring Boot 很多组件依赖于Spring Boot 事件监听器实现，本质是Spring Framework 事件/监听

机制。

Spring Application 应用：

* Spring应用上下文生命周期控制注解驱动Bean
* Spring事件/监听机制加载或者初始化组件

@EnableAutoConfiguration 自动配置

ConfigurationClassParser-> parse()->processConfigurationClass()->doProcessConfigurationClass

()->ComponentScanAnnotationParser. parse()->ClassPathBeanDefinitionScanner. doScan()->processImports()->AutoConfigurationImportSelector-> selectImports

-> AutoConfigurationMetadataLoader.loadMetadata(加载**META-INF/spring-autoconfigure-metadata.properties中配置的配置类**

)

.