# **[SPRINGBOOT启动流程及其原理](https://www.cnblogs.com/theRhyme/p/11057233.html)**

* [SpringBoot启动原理精简版](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label0)
* [Spring Boot、Spring MVC 和 Spring 有什么区别？](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label1)
* [一 springboot启动原理及相关流程概览](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label2)
* [二  springboot的启动类入口](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label3)
* [三  单单是SpringBootApplication接口用到了这些注解](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label4)
  + [1）@Configuration注解](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_lab2_4_0)
  + [2） @ComponentScan注解](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_lab2_4_1)
  + [3) @EnableAutoConfiguration](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_lab2_4_2)
    - [AutoConfigurationPackage注解：](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label3_4_2_0)
    - [Import(AutoConfigurationImportSelector.class)注解](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label3_4_2_1)
  + [SpringFactoriesLoader详解](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_lab2_4_3)
* [四  springboot启动流程概览图](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label5)
* [五 深入探索SpringApplication执行流程](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label6)
* [简单了解下Bean的生命周期](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label7)
* [BeanFactory 和ApplicationContext的区别](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label8)
* [SpringMVC处理请求的流程](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label9)
* [BEANFACTORY和FACTORYBEAN的区别与联系](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label10)
* [Bean的循环依赖](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label11)
* [@Import注解介绍](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label12)
* [同一个类中调用 @Transaction注解的方法会有事务效果吗？](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label13)
* [来源](https://www.cnblogs.com/theRhyme/p/11057233.html" \l "_label14)

# **SpringBoot启动原理精简版**

[https://www.cnblogs.com/theRhyme/p/how-does-springboot-start.html](https://www.cnblogs.com/theRhyme/p/how-does-springboot-start.html" \t "https://www.cnblogs.com/theRhyme/p/_blank)

# **Spring Boot、Spring MVC 和 Spring 有什么区别？**

分别描述各自的特征：

Spring 框架就像一个家族，有众多衍生产品例如 boot、security、jpa等等；但他们的基础都是Spring 的ioc和 aop，ioc 提供了依赖注入的容器， aop解决了面向切面编程，然后在此两者的基础上实现了其他延伸产品的高级功能。

Spring MVC提供了一种轻度耦合的方式来开发web应用；它是Spring的一个模块，是一个web框架；通过DispatcherServlet, ModelAndView 和 View Resolver，开发web应用变得很容易；解决的问题领域是网站应用程序或者服务开发——URL路由、Session、模板引擎、静态Web资源等等。

Spring Boot实现了auto-configuration**自动配置**（另外三大神器actuator监控，cli命令行接口，starter依赖），降低了项目搭建的复杂度。它主要是为了解决使用Spring框架需要进行大量的配置太麻烦的问题，所以它并不是用来替代Spring的解决方案，而是和Spring框架紧密结合用于提升Spring开发者体验的工具；同时它集成了大量常用的第三方库配置(例如Jackson, JDBC, Mongo, Redis, Mail等等)，Spring Boot应用中这些第三方库几乎可以零配置的开箱即用(out-of-the-box)。

所以，用最简练的语言概括就是:

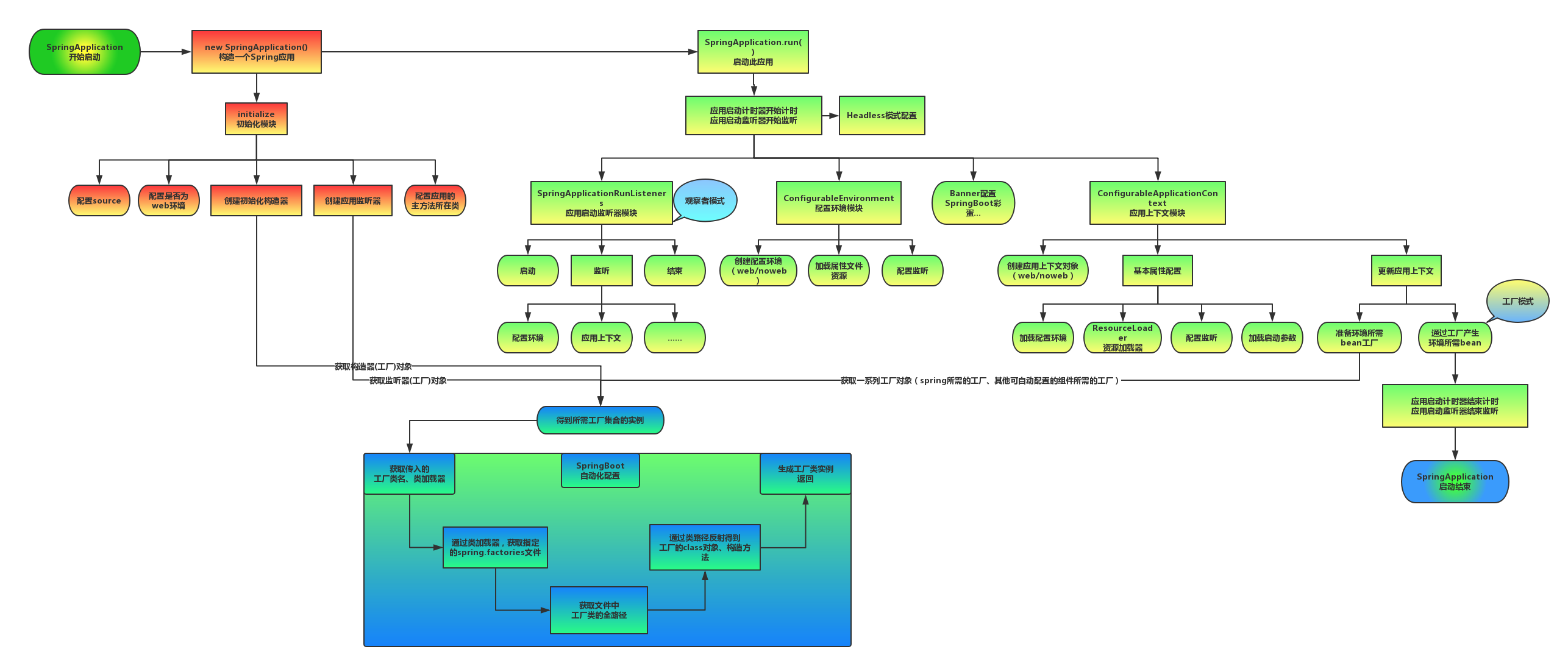
Spring 是一个“引擎”;

Spring MVC 是基于Spring的一个 MVC 框架;

Spring Boot 是基于Spring4的条件注册的一套快速开发整合包。

# **一 springboot启动原理及相关流程概览**

　　springboot是基于spring的新型的轻量级框架，最厉害的地方当属**自动配置。**那我们就可以根据启动流程和相关原理来看看，如何实现传奇的自动配置。



# **二  springboot的启动类入口**

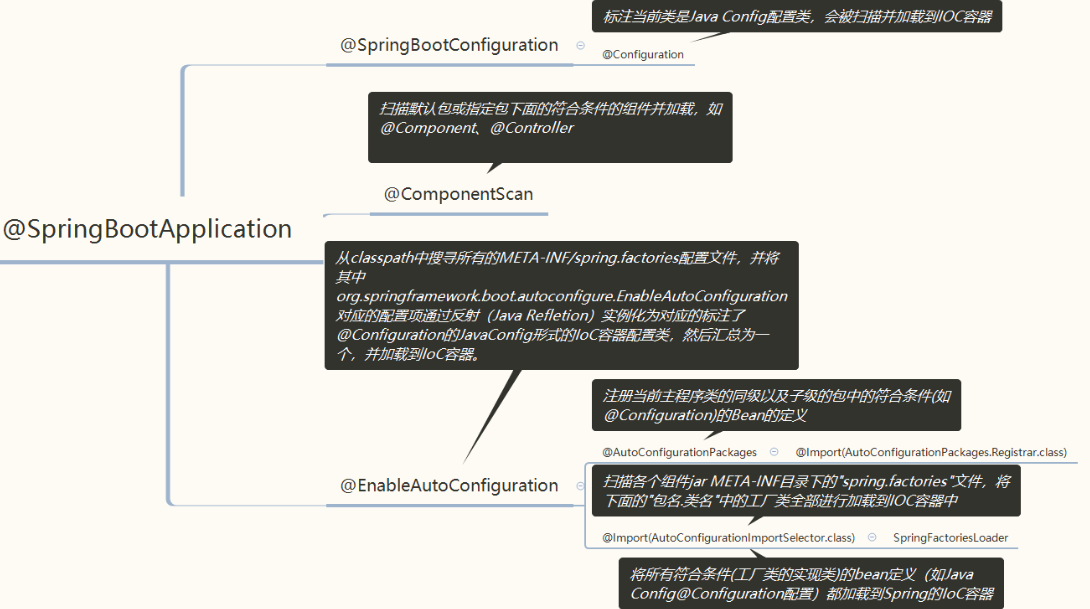
用过springboot的技术人员很显而易见的两者之间的差别就是视觉上很直观的：springboot有自己独立的启动类（独立程序）

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @SpringBootApplication  public class Application {      public static void main(String[] args) {          SpringApplication.run(Application.class, args);      }  } |

从上面代码可以看出，Annotation定义（@SpringBootApplication）和类定义（SpringApplication.run）最为耀眼，所以要揭开SpringBoot的神秘面纱，我们要从这两位开始就可以了。

# **三  单单是SpringBootApplication接口用到了这些注解**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Target(ElementType.TYPE) // 注解的适用范围，其中TYPE用于描述类、接口（包括包注解类型）或enum声明  @Retention(RetentionPolicy.RUNTIME) // 注解的生命周期，保留到class文件中（三个生命周期）  @Documented // 表明这个注解应该被javadoc记录  @Inherited // 子类可以继承该注解  @SpringBootConfiguration // 继承了Configuration，表示当前是注解类  @EnableAutoConfiguration // 开启springboot的注解功能，springboot的四大神器之一，其借助@import的帮助  @ComponentScan(excludeFilters = { // 扫描路径设置  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })  public @interface SpringBootApplication {  ...  } |



在其中比较重要的有三个注解，分别是：

　　1）@SpringBootConfiguration // 继承了Configuration，表示当前是注解类

　　2）@EnableAutoConfiguration // 开启springboot的注解功能，springboot的四大神器之一，其借助@import的帮助

　　3）@ComponentScan(excludeFilters = { // 扫描路径设置（具体使用待确认）

 　　接下来对三个注解一一详解，增加对springbootApplication的理解：

## **1）@Configuration注解**

　　按照原来xml配置文件的形式，在springboot中我们大多用配置类来解决配置问题

　　　配置bean方式的不同：

　　　　a）xml配置文件的形式配置bean

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"  default-lazy-init="true">  <!--bean定义-->  </beans> |

　　b）java configuration的配置形式配置bean

|  |  |
| --- | --- |
| 1  2  3  4 | @Configuration  public class MockConfiguration{      //bean定义  } |

　　注入bean方式的不同：

　　　　a）xml配置文件的形式注入bean

|  |  |
| --- | --- |
| 1  2  3 | <bean id="mockService" class="..MockServiceImpl">  ...  </bean> |

　　b）java configuration的配置形式注入bean

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Configuration  public class MockConfiguration{      @Bean      public MockService mockService(){          return new MockServiceImpl();      }  } |

任何一个标注了@Bean的方法，其返回值将作为一个bean定义注册到Spring的IoC容器，方法名将默认成该bean定义的id。

　　表达**bean之间依赖关系**的不同：

　　　　a）xml配置文件的形式表达依赖关系

|  |  |
| --- | --- |
| 1  2  3  4 | <bean id="mockService" class="..MockServiceImpl">  　　<propery name ="dependencyService" ref="dependencyService" />  </bean>  <bean id="dependencyService" class="DependencyServiceImpl"></bean> |

　　　　b）java configuration配置的形式表达**依赖关系（重点）**

**如果一个bean A的定义依赖其他bean B,则直接调用对应的JavaConfig类中依赖bean B的创建方法就可以了。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Configuration  public class MockConfiguration{  　　@Bean  　　public MockService mockService(){      　　return new MockServiceImpl(dependencyService());  　　}  　　@Bean  　　public DependencyService dependencyService(){      　　return new DependencyServiceImpl();  　　}  } |

## **2） @ComponentScan注解**

　　作用：a）对应xml配置中的元素；

　　　　　b） **（重点）**ComponentScan的功能其实就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义;

　　　　　c) 将这些bean定义加载到**IoC**容器中.

　　 我们可以通过basePackages等属性来**细粒度**的定制@ComponentScan自动扫描的范围，如果不指定，则**默认**Spring框架实现会从声明@ComponentScan所在类的package进行扫描。

　　注：所以SpringBoot的启动类最好是放在root package下，因为默认不指定basePackages

## **3) **@EnableAutoConfiguration****

　　　　此注解顾名思义是可以自动配置，所以应该是springboot中最为重要的注解。

　　　　在spring框架中就提供了各种以@Enable开头的注解，例如： @EnableScheduling、@EnableCaching、@EnableMBeanExport等； @EnableAutoConfiguration的理念和做事方式其实一脉相承简单概括一下就是，借助@Import的支持，收集和注册特定场景相关的bean定义。

* + @EnableScheduling是通过@Import将Spring调度框架相关的bean定义都加载到IoC容器【定时任务、时间调度任务】
  + @EnableMBeanExport是通过@Import将JMX相关的bean定义加载到IoC容器【监控JVM运行时状态】

　　　　 @EnableAutoConfiguration也是借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器。

　　　 　@EnableAutoConfiguration作为一个复合Annotation,其自身定义关键信息如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @SuppressWarnings("deprecation")  @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @AutoConfigurationPackage【重点注解】  @Import(AutoConfigurationImportSelector.class)【重点注解】  public @interface EnableAutoConfiguration {  ...  } |

其中最重要的两个注解已经标注：1、@AutoConfigurationPackage【重点注解】2、@Import(AutoConfigurationImportSelector.class)【重点注解】

 　  当然还有其中比较重要的一个类就是：AutoConfigurationImportSelector.class

### **AutoConfigurationPackage注解：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @Import(AutoConfigurationPackages.Registrar.class)  public @interface AutoConfigurationPackage {    } |

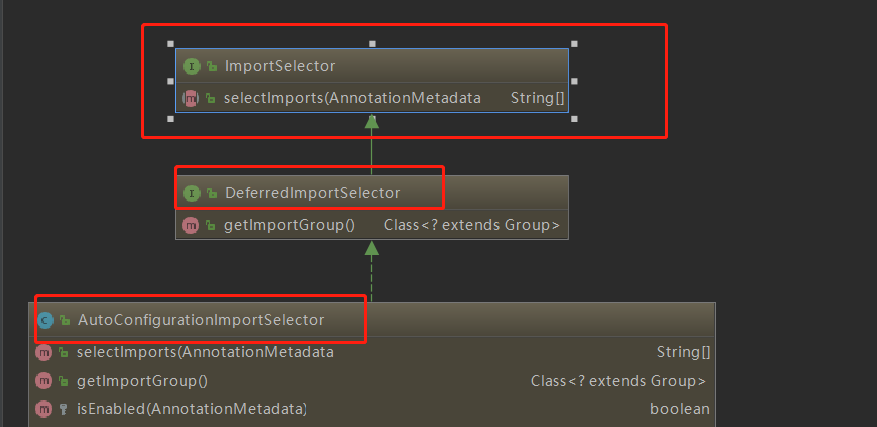
通过@Import(AutoConfigurationPackages.**Registrar**.class)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {            @Override          public void registerBeanDefinitions(AnnotationMetadata metadata,                  BeanDefinitionRegistry registry) {              register(registry, new PackageImport(metadata).getPackageName());          }            ……        } |

注册**当前启动类的根package**；

注册org.springframework.boot.autoconfigure.**AutoConfigurationPackages**的**BeanDefinition**。

### **Import(AutoConfigurationImportSelector.class)注解**



（重点）可以从图中看出  **AutoConfigurationImportSelector 实现**了 DeferredImportSelector 从 ImportSelector继承的方法：**selectImports**。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Override      public String[] selectImports(AnnotationMetadata annotationMetadata) {          if (!isEnabled(annotationMetadata)) {              return NO\_IMPORTS;          }          AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader                  .loadMetadata(this.beanClassLoader);          AnnotationAttributes attributes = getAttributes(annotationMetadata);          List<String> configurations = getCandidateConfigurations(annotationMetadata,                  attributes);          configurations = removeDuplicates(configurations);          Set<String> exclusions = getExclusions(annotationMetadata, attributes);          checkExcludedClasses(configurations, exclusions);          configurations.removeAll(exclusions);          configurations = filter(configurations, autoConfigurationMetadata);          fireAutoConfigurationImportEvents(configurations, exclusions);          return StringUtils.toStringArray(configurations);      } |

第9行List<String> configurations = getCandidateConfigurations(annotationMetadata,attributes);其实是去加载各个组件jar下的   public static final String FACTORIES\_RESOURCE\_LOCATION = "META-INF/spring.factories";外部文件。

该方法在springboot启动流程——bean实例化前被执行，返回要实例化的类信息列表；

如果获取到类信息，spring可以通过类加载器将类加载到jvm中，现在我们已经通过spring-boot的starter依赖方式依赖了我们需要的组件，那么这些组件的类信息在select方法中就可以被获取到。

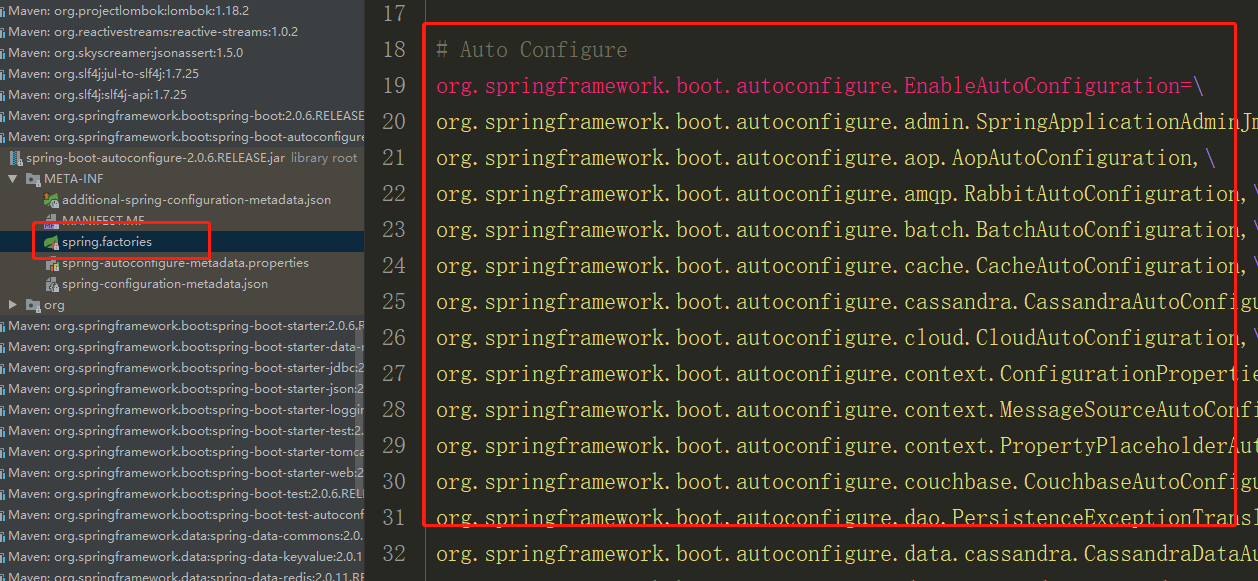
|  |  |
| --- | --- |
| 1  2  3  4  5 | protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {   List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());   Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you are using a custom packaging, make sure that file is correct.");   return configurations;   } |

其返回一个自动配置类的类名列表，方法调用了loadFactoryNames方法，查看该方法

|  |  |
| --- | --- |
| 1  2  3  4 | public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {   String factoryClassName = factoryClass.getName();   return (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());   } |

自动配置器会跟根据传入的factoryClass.getName()到项目系统路径下所有的spring.factories文件中找到相应的key，从而加载里面的类。

这个外部文件，有很多自动配置的类。如下：



**（重点）**其中，最关键的要属@Import(AutoConfigurationImportSelector.class)，借助AutoConfigurationImportSelector，@EnableAutoConfiguration可以帮助SpringBoot应用将所有符合条件(spring.factories)的**bean定义**（如Java Config@Configuration配置）都加载到当前SpringBoot创建并使用的IoC容器。

## **SpringFactoriesLoader详解**

借助于Spring框架**原有**的一个工具类：**SpringFactoriesLoader**的支持，@EnableAutoConfiguration可以智能的自动配置功效才得以大功告成！

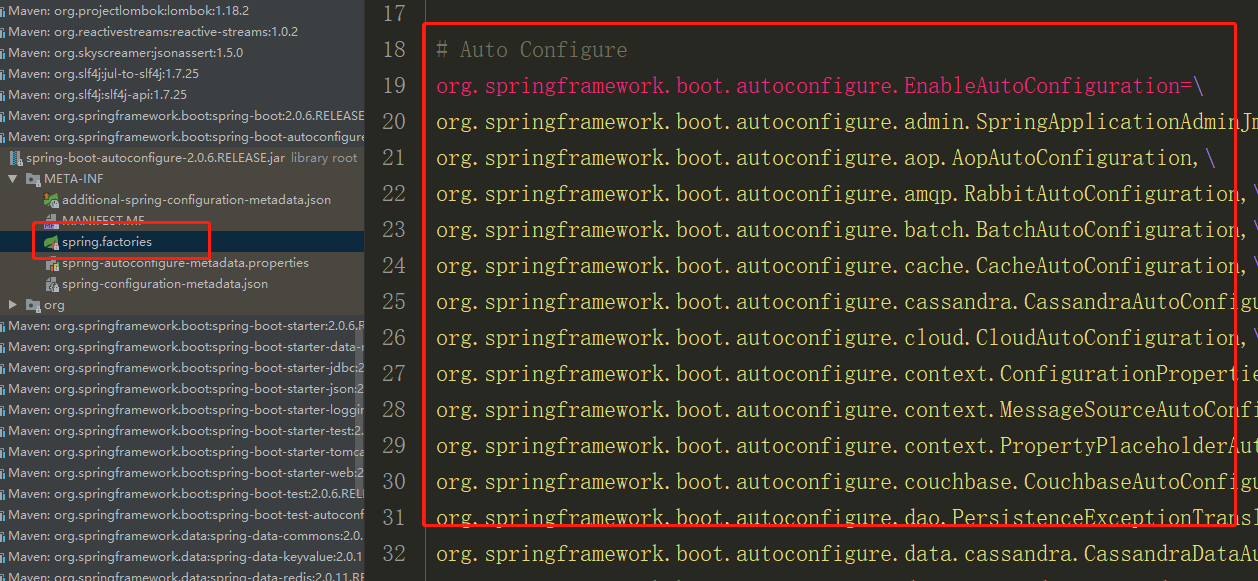
SpringFactoriesLoader属于Spring框架私有的一种扩展方案，其主要功能就是从**指定的配置文件META-INF/spring.factories**加载配置,**加载工厂类**。

SpringFactoriesLoader为Spring工厂加载器，该对象提供了loadFactoryNames方法，入参为factoryClass和classLoader即需要传入**工厂类**名称和对应的类加载器，方法会根据指定的classLoader，加载该类加器搜索路径下的指定文件，即spring.factories文件；

传入的工厂类为接口，而文件中对应的类则是接口的实现类，或最终作为实现类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public abstract class SpringFactoriesLoader {  //...  　　public static <T> List<T> loadFactories(Class<T> factoryClass, ClassLoader classLoader) {  　　　　...  　　}      　　public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {  　　　　....  　　}  } |

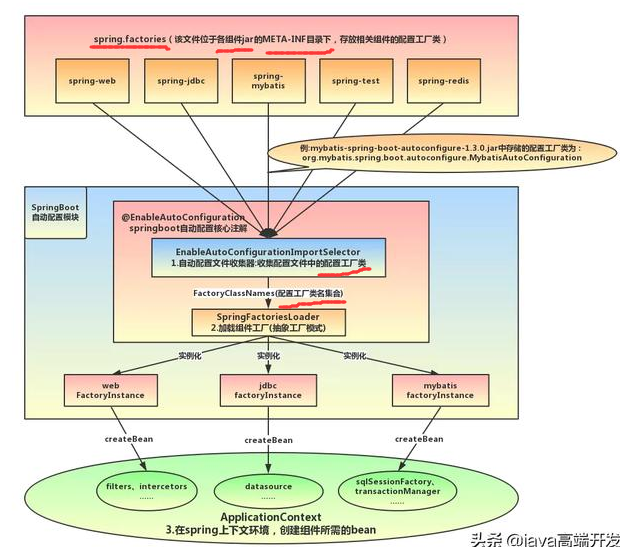
配合@EnableAutoConfiguration使用的话，它更多是提供一种配置查找的功能支持，即根据@EnableAutoConfiguration的完整类名org.springframework.boot.autoconfigure.EnableAutoConfiguration作为查找的Key,获取对应的一组@Configuration类



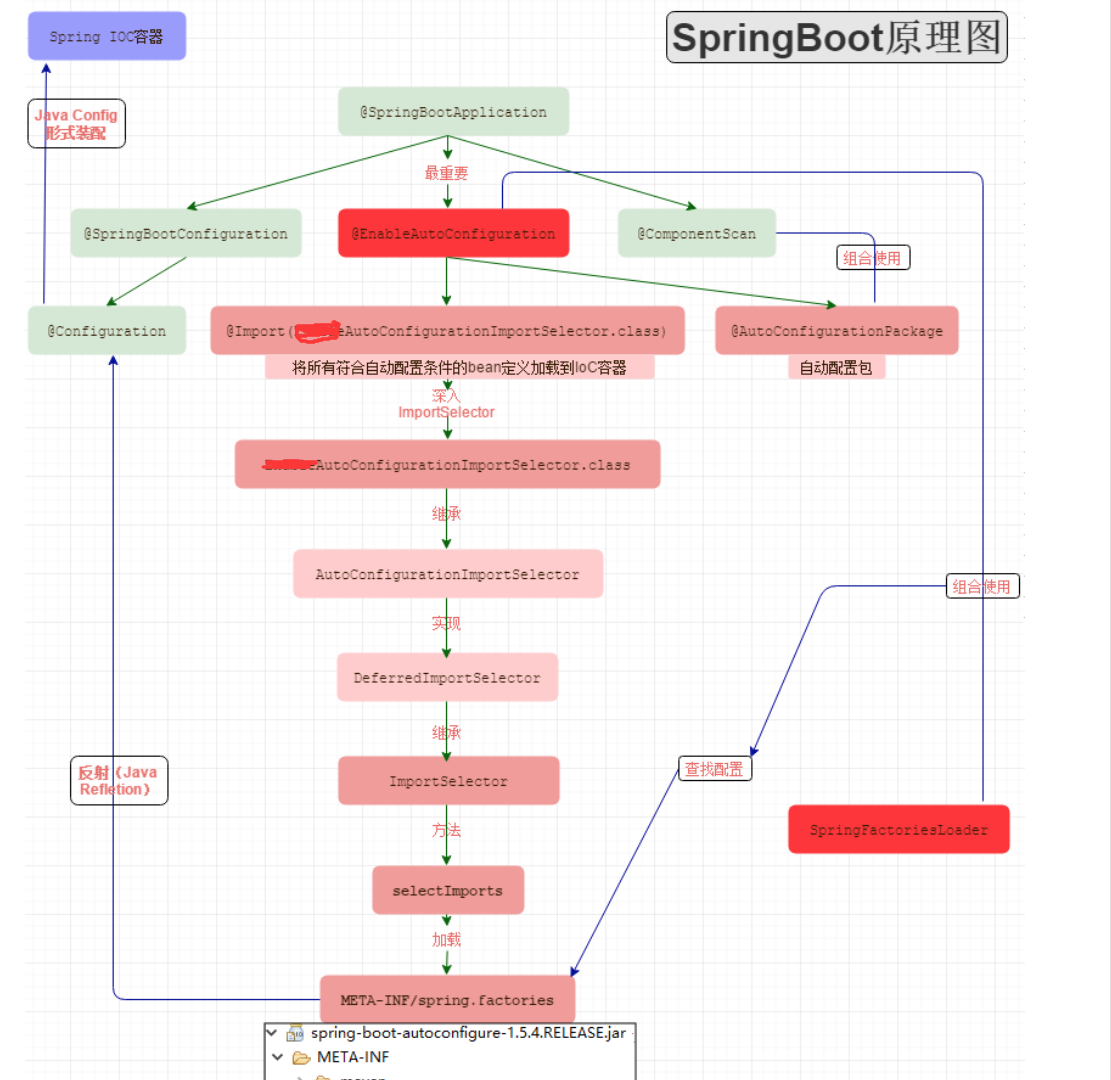
上图就是从SpringBoot的autoconfigure依赖包中的META-INF/spring.factories配置文件中摘录的一段内容，可以很好地说明问题。

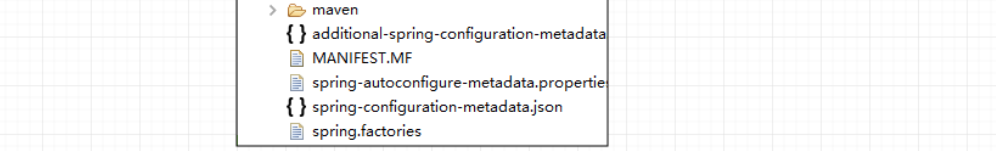
（重点）所以，@EnableAutoConfiguration自动配置的魔法其实就变成了：

从classpath中搜寻所有的META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的**配置项**通过**反射（Java Refletion）**实例化为对应的标注了**@Configuration**的JavaConfig形式的IoC容器配置类，然后汇总为一个并加载到IoC容器。

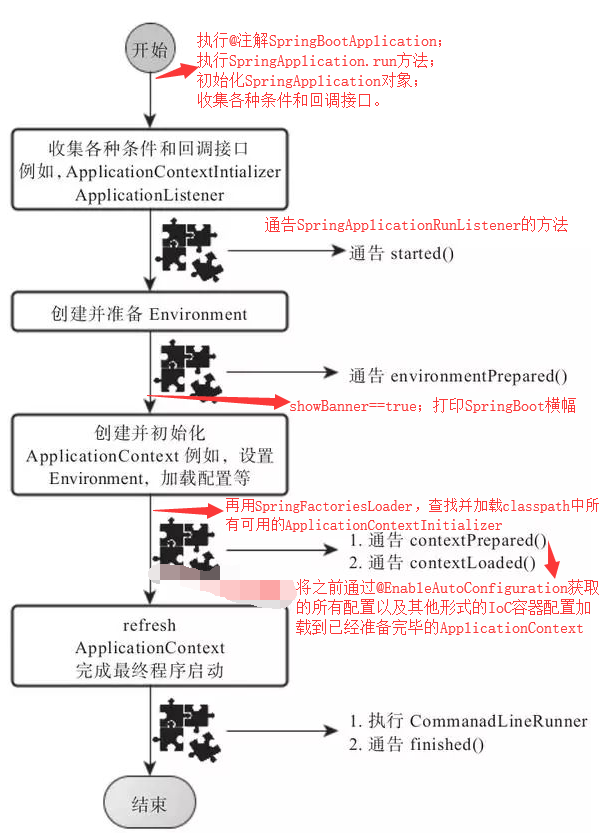


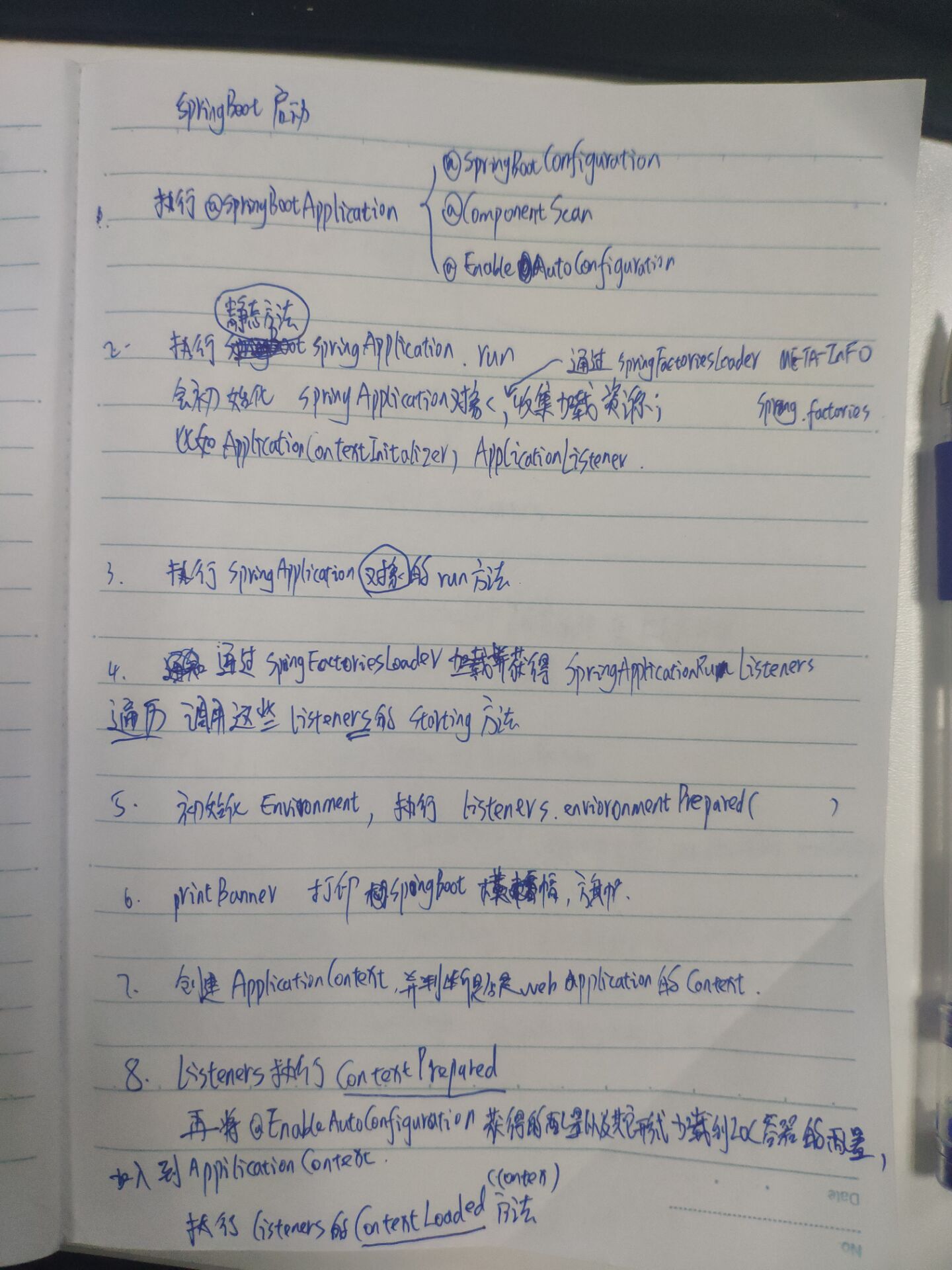
# **四  springboot启动流程概览图**

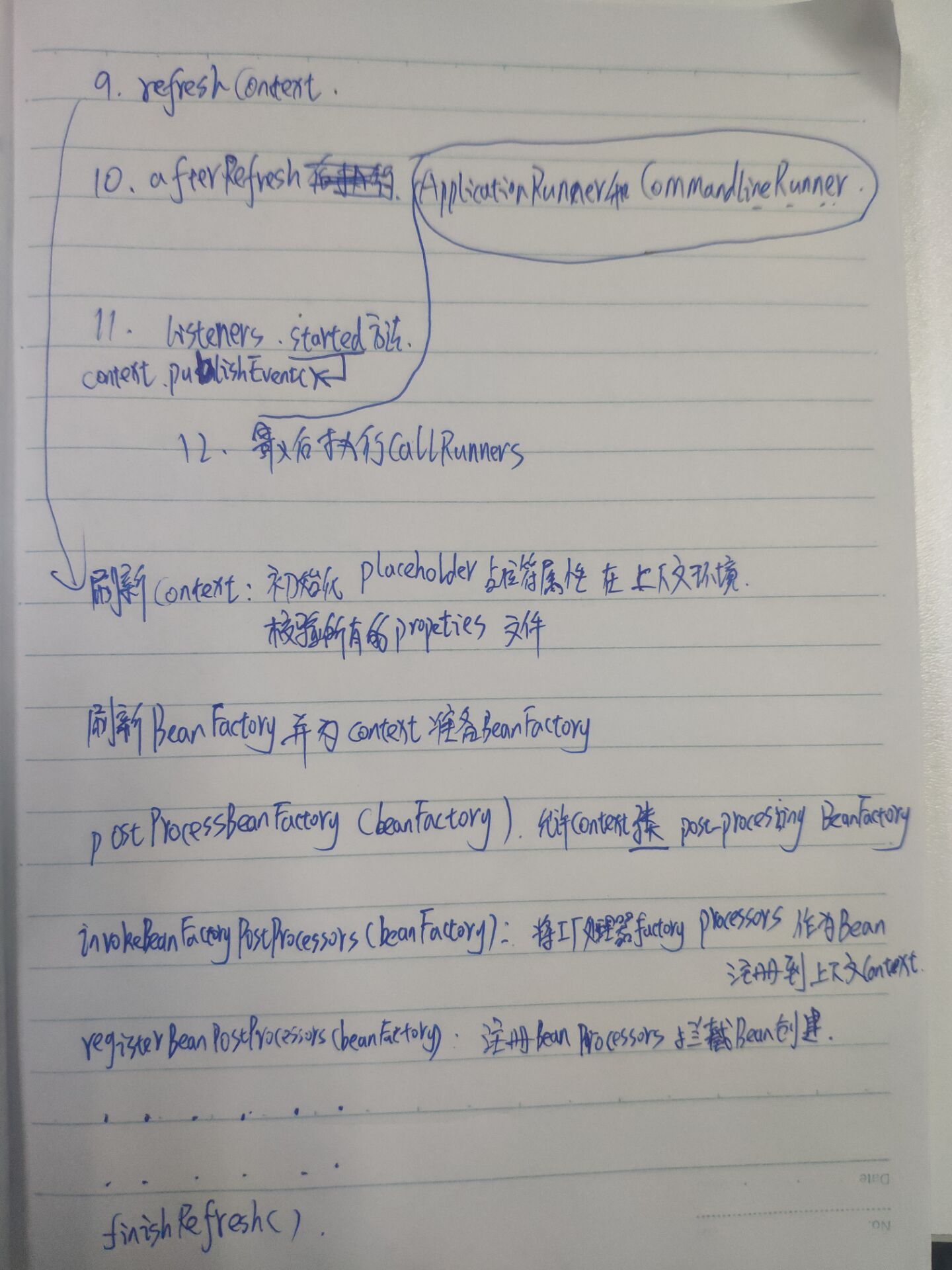




# **五 深入探索SpringApplication执行流程**







|  |  |
| --- | --- |
| 1 | public class EventPublishingRunListener implements SpringApplicationRunListener, Ordered |

SpringApplicationRunListener接口的唯一实现是EventPublishingRunListener；

实现了方法，下面是部分方法源码：

[IMG_267](https://www.cnblogs.com/theRhyme/p/javascript:void(0);)

public class EventPublishingRunListener implements SpringApplicationRunListener, Ordered {

private final SpringApplication application;

private final String[] args;

private final SimpleApplicationEventMulticaster initialMulticaster;

public EventPublishingRunListener(SpringApplication application, String[] args) {

this.application = application;

this.args = args;

this.initialMulticaster = new SimpleApplicationEventMulticaster();

for (ApplicationListener<?> listener : application.getListeners()) {

this.initialMulticaster.addApplicationListener(listener);

}

}

@Override

public int getOrder() {

return 0;

}

// SpringBoot启动事件 @Override

public void starting() {

this.initialMulticaster.multicastEvent(new ApplicationStartingEvent(this.application, this.args));

}

// 创建和配置环境 @Override

public void environmentPrepared(ConfigurableEnvironment environment) {

this.initialMulticaster

.multicastEvent(new ApplicationEnvironmentPreparedEvent(this.application, this.args, environment));

}

// 准备ApplicationContext @Override

public void contextPrepared(ConfigurableApplicationContext context) {

this.initialMulticaster

.multicastEvent(new ApplicationContextInitializedEvent(this.application, this.args, context));

}

// 发布ApplicationContext已经refresh事件，标志着ApplicationContext初始化完成 @Override

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));

}

// SpringBoot已启动事件 @Override

public void started(ConfigurableApplicationContext context) {

context.publishEvent(new ApplicationStartedEvent(this.application, this.args, context));

AvailabilityChangeEvent.publish(context, LivenessState.CORRECT);

}

// "SpringBoot现在可以处理接受的请求"事件 @Override

public void running(ConfigurableApplicationContext context) {

context.publishEvent(new ApplicationReadyEvent(this.application, this.args, context));

AvailabilityChangeEvent.publish(context, ReadinessState.ACCEPTING\_TRAFFIC);

}

@Override

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 final Log logger = LogFactory.getLog(EventPublishingRunListener.class);

@Override

public void handleError(Throwable throwable) {

logger.warn("Error calling ApplicationEventListener", throwable);

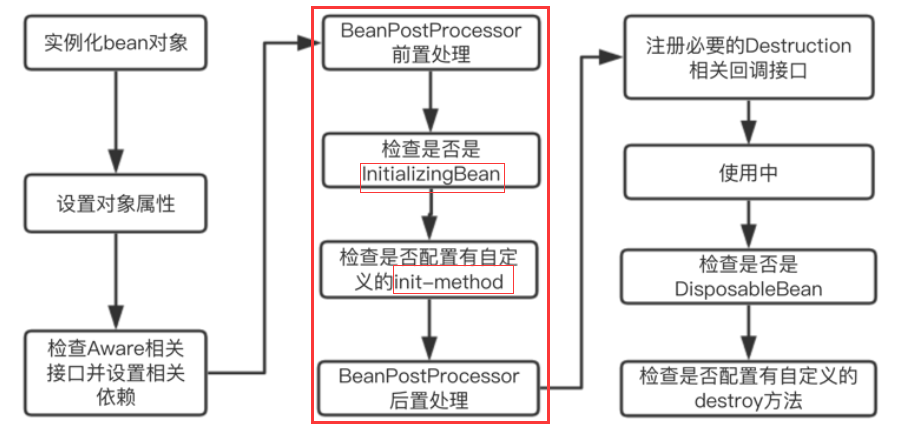
}

}

}

[IMG_268](https://www.cnblogs.com/theRhyme/p/javascript:void(0);)

# **简单了解下Bean的生命周期**



一个Bean的构造函数初始化时是最先执行的，这个时候，**bean属性还没有被注入**；

@PostConstruct注解的方法优先于InitializingBean的afterPropertiesSet执行，这时Bean的属性竟然被注入了；

spring很多组件的初始化都放在afterPropertiesSet做,想和spring一起启动，可以放在这里启动;

spring为bean提供了两种初始化bean的方式，实现InitializingBean接口，实现afterPropertiesSet方法，或者在配置文件中同过init-method指定，两种方式可以同时使用；

实现InitializingBean接口是直接调用afterPropertiesSet方法，比通过反射调用init-method指定的方法效率相对来说要高点；但是init-method方式消除了对spring的依赖；

如果调用afterPropertiesSet方法时出错，则不调用init-method指定的方法。

Bean在实例化的过程中：

Constructor > @PostConstruct > InitializingBean > init-method

# ****BeanFactory 和ApplicationContext的区别****

BeanFactory和ApplicationContext都是接口，并且ApplicationContext间接继承了BeanFactory。

BeanFactory是Spring中最底层的接口，提供了最简单的容器的功能，只提供了实例化对象和获取对象的功能，而ApplicationContext是Spring的一个更高级的容器，提供了更多的有用的功能。

ApplicationContext提供的额外的功能：获取Bean的详细信息(如定义、类型)、国际化的功能、统一加载资源的功能、强大的事件机制、对Web应用的支持等等。

加载方式的区别：BeanFactory采用的是**延迟加载的**形式来注入Bean；ApplicationContext则相反的，它是在Ioc**启动**时就一次性创建**所有的Bean,**好处是可以马上发现Spring配置文件中的错误，坏处是造成浪费。

|  |  |
| --- | --- |
| 1  2 | public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, HierarchicalBeanFactory,          MessageSource, ApplicationEventPublisher, ResourcePatternResolver |

# ****SpringMVC处理请求的流程****

1、用户发送请求至前端控制器DispatcherServlet

2、DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、处理器映射器根据请求url找到具体的处理器，生成处理器对象Handler及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、DispatcherServlet通过HandlerAdapter（让Handler实现更加灵活）处理器适配器调用处理器

5、执行处理器(Controller，也叫后端控制器)。

6、Controller执行完成返回ModelAndView（连接业务逻辑层和展示层的桥梁，持有一个ModelMap对象和一个View对象）。

7、HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

8、DispatcherServlet将ModelAndView传给ViewReslover视图解析器

9、ViewReslover解析后返回具体View

10、DispatcherServlet对View进行渲染视图（将ModelMap模型数据填充至视图中）。

11、DispatcherServlet响应用户

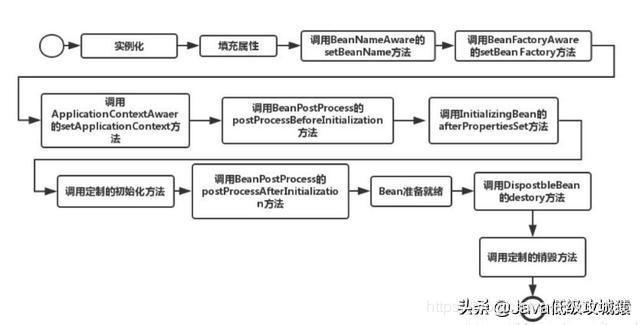
# **[BEANFACTORY和FACTORYBEAN的区别与联系](https://www.cnblogs.com/theRhyme/p/9139673.html)**

1. 两者都是接口；
2. BeanFactory主要是用来创建Bean和获得Bean的；
3. FactoryBean跟普通Bean不同，其返回的对象不是指定类的一个实例，而是该FactoryBean的**getObject**方法所返回的对象；
4. 通过BeanFactory和beanName获取bean时，如果beanName不加&则获取到对应bean的实例；如果beanName加上&，则获取到FactoryBean本身的实例
5. FactoryBean 通常是用来**创建比较复杂的bean(如创建mybatis的SqlSessionFactory很复杂)**，一般的bean 直接用xml配置即可，但如果创建一个bean的创建过程中涉及到很多**其他的bean**和复杂的逻辑，用xml配置比较困难，这时可以考虑用FactoryBean。

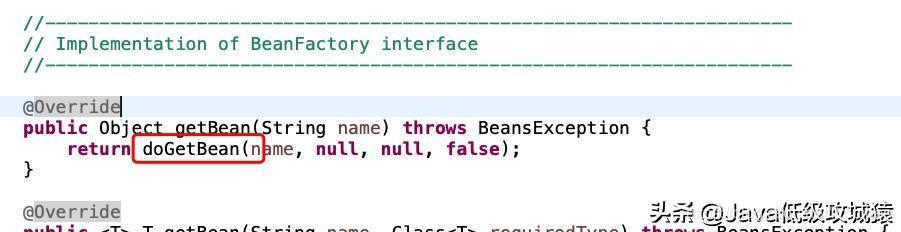
# **Bean的循环依赖**

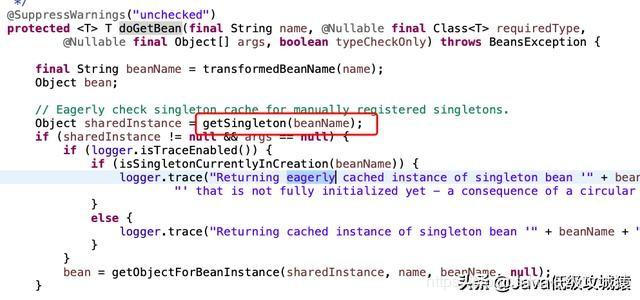
<https://blog.csdn.net/itmrchen/article/details/90201279>

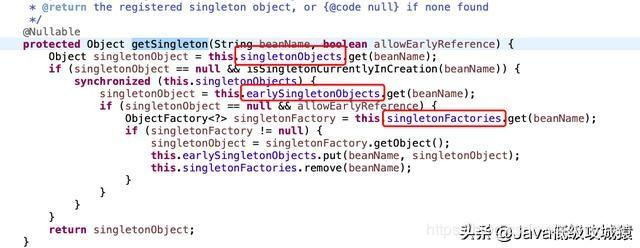
对于Spring中Bean的管理，下图一目了然：

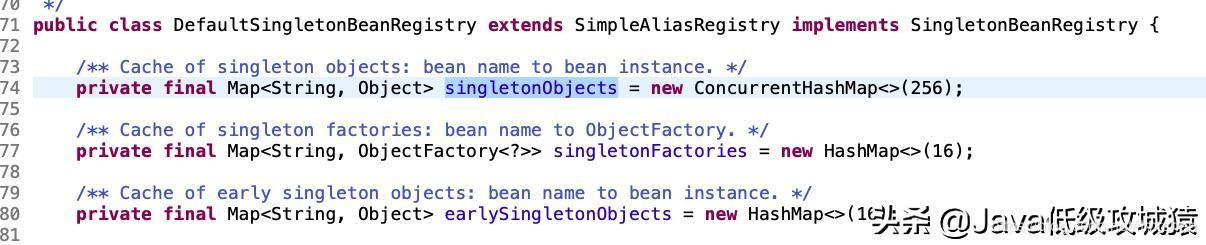


先调用构造函数进行实例化，然后填充属性，再接着进行其他附加操作和初始化，正是这样的生命周期，才有了Spring的解决循环依赖，这样的解决机制是根据Spring框架内定义的三级缓存来实现的，也就是说：三级缓存解决了Bean之间的循环依赖。我们从源码中来说明。

先来看Spring中Bean工厂是怎么获取Bean的（AbstractBeanFactory中）：  








 一级一级向下寻找，找出了前面提到的三级缓存，也就是三个Map集合类：

singletonObjects：第一级缓存，里面放置的是已经实例化好的单例对象；

earlySingletonObjects：第二级缓存，里面存放的是提前曝光的单例对象；

singletonFactories：第三级缓存，里面存放的是将要被实例化的对象的对象工厂。

所以当一个Bean调用构造函数进行实例化后，即使set属性还未填充，就可以通过**三级缓存**向外暴露依赖的引用值进行set（所以循环依赖问题的解决也是基于Java的引用传递），这也说明了另外一点，基于构造函数的注入，如果有循环依赖，Spring是不能够解决的。

还要说明一点，Spring默认的Bean Scope是**单例**的，而三级缓存中都包含singleton，可见是对于单例Bean之间的循环依赖的解决，Spring是通过三级缓存来实现的。

# **@Import注解介绍**

[https://www.cnblogs.com/theRhyme/p/13887223.html](https://www.cnblogs.com/theRhyme/p/13887223.html" \t "https://www.cnblogs.com/theRhyme/p/_blank)

# **同一个类中调用 @Transaction注解的方法会有事务效果吗？**

没有，可以Autowired注入自己，然后再调用注入的类中的方法，即自己依赖自己，循环依赖；

这里在一个内部调用应该是相当于单纯的调用方法this.methodName()，并没有AOP代理。

# **Spring Boot启动过程分析**

[](https://www.jianshu.com/u/de5f5d2331ee)

[看不见的BUG](https://www.jianshu.com/u/de5f5d2331ee" \t "https://www.jianshu.com/p/_blank)关注

22019.06.08 18:04:32字数 1,570阅读 125,958

首先贴一张很不错的图，[SpringBoot启动结构图](https://links.jianshu.com/go?to=https://www.processon.com/view/link/59812124e4b0de2518b32b6e" \t "https://www.jianshu.com/p/_blank)，图片出自[SpringBoot启动流程解析](https://www.jianshu.com/p/87f101d8ec41" \t "https://www.jianshu.com/p/_blank)。  
本文的分析基于Spring Boot 2.1.5，非Spring的代码只有下面这个启动main函数：

@SpringBootApplicationpublic class App {

public static void main(String[] args) {

SpringApplication application = new SpringApplication(AppServer.class);

application.run(args);

}}

# **构造函数**

SpringApplication的构造函数实例化了 初始化上下文的各种接口--ApplicationContextInitializer以及监听器--ApplicationListener，要注意的是这里的实例化，并不像平时的Spring Components一样通过注解和扫包完成，而是通过一种不依赖Spring上下文的加载方法，这样才能在Spring完成启动前做各种配置。Spring的解决方法是以接口的全限定名作为key，实现类的全限定名作为value记录在项目的META-INF/spring.factories文件中，然后通过SpringFactoriesLoader工具类提供静态方法进行类加载并缓存下来，spring.factories是Spring Boot的核心配置文件，后面会继续说明。另外比较有意思的是两个deduce方法，Spring Boot项目主要的目标之一就是自动化配置，通过这两个deduce方法可以看出，Spring Boot的判断方法之一是检查系统中是否存在的核心类。

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();//通过核心类判断是否开启、开启什么web容器

//实例化初始器

setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));

//实例化监听器

setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));

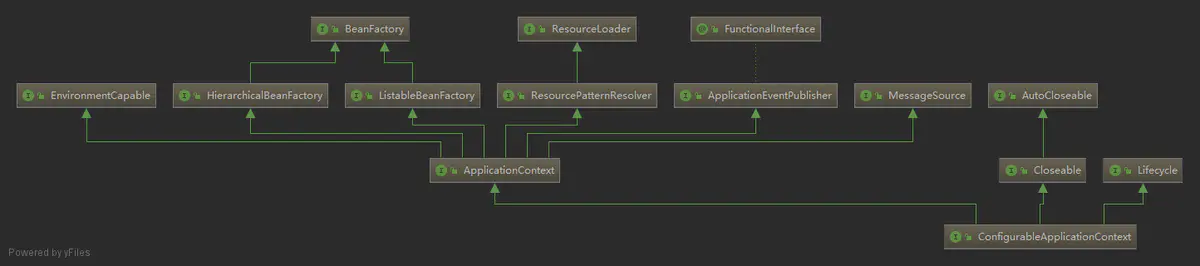
this.mainApplicationClass = deduceMainApplicationClass();}

# **Run**

初始化完成之后就进到了run方法，run方法完成了所有Spring的整个启动过程：准备Environment——发布事件——创建上下文、bean——刷新上下文——结束，其中穿插了很多监听器的动作，并且很多逻辑都是靠各种监听器的实现类执行的，所以在分析run方法之前，先看下各种核心监听器、接口的作用。

## **ConfigurableApplicationContext**

不得不说，用IDEA分析源码真的很方便，直接生成接口的UML类图：



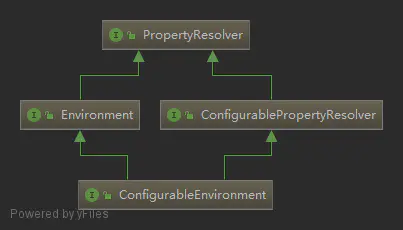
ConfigurableApplicationContext.png

相对于只读的ApplicationContext而言，ConfigurableApplicationContext提供了配置上下文的接口，如设置Environment、监听器、切面类、关闭上下文的钩子等，还有刷新上下文的接口。默认是只读的接口，接口名前面加Configurable对应是一个提供可配置接口的新接口——在Spring很多配置相关的接口中都有这样的继承形式，例如ConfigurableEnvironment和Environment、ConfigurablePropertyResolver和PropertyResolver、ConfigurableBeanFactory和BeanFactory等等。  
继承的三个父类接口里，Closeable提供了关闭时资源释放的接口，Lifecycle是提供对生命周期控制的接口(start\stop)以及查询当前运行状态的接口，ApplicationContext则是配置上下文的中心配置接口，继承了其他很多配置接口，其本身提供查询诸如id、应用程序名等上下文档案信息的只读接口，以及构建自动装配bean的工厂（注释上官方说该接口提供的工厂是用于注册上下文外部的bean的，但调试发现和在程序内@Autowired获取到的工厂是同一个对象...）。简单写下ApplicationContext继承的父类接口。

* EnvironmentCapable  
  提供Environment接口。
* MessageSource  
  国际化资源接口。
* ApplicationEventPublisher  
  事件发布器。
* ResourcePatternResolver  
  资源加载器。
* HierarchicalBeanFactory、ListableBeanFactory  
  这两个都继承了bean容器的根接口BeanFactory，具体在另一篇博客[Spring的bean工厂分析](https://links.jianshu.com/go?to=%5Bhttps://www.jianshu.com/p/5e944d79c62c%5D(https://www.jianshu.com/p/5e944d79c62c)" \t "https://www.jianshu.com/p/_blank)分析。

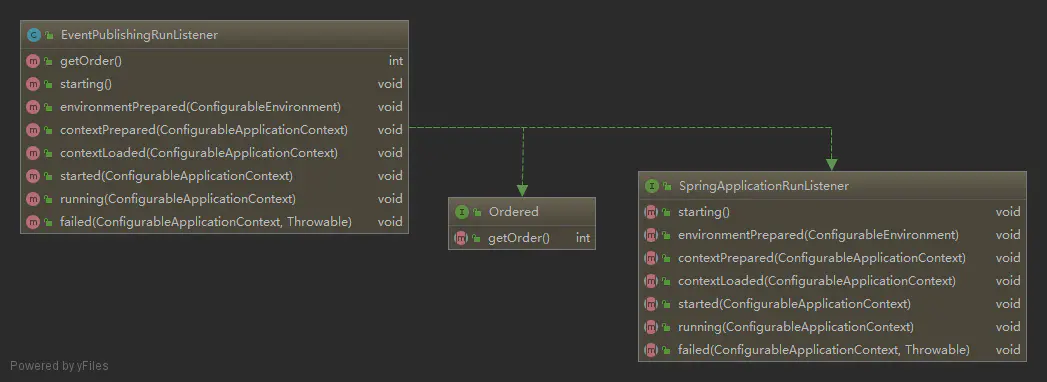
## **ConfigurableEnvironment**

一般在写业务代码时使用的都是只读类型的接口Environment，该接口是对运行程序环境的抽象，是保存系统配置的中心，而在启动过程中使用的则是可编辑的ConfigurableEnvironment。接口的UML类图如下，提供了合并父环境、添加active profile以及一些设置解析配置文件方式的接口。  
其中一个比较重要的方法MutablePropertySources getPropertySources();，该方法返回一个可编辑的PropertySources，如果有在启动阶段自定义环境的PropertySources的需求，就可以通过该方法设置。



ConfigurableEnvironment.png

## **EventPublishingRunListener**



EventPublishingRunListener.png

该监听器实际上是一个用于广播Spring事件的广播器，实现SpringApplicationRunListener接口的方法都是包装一个Spring事件并进行广播，例如：

@Overridepublic void contextPrepared(ConfigurableApplicationContext context) {

this.initialMulticaster.multicastEvent(new ApplicationContextInitializedEvent(this.application, this.args, context));}@Overridepublic void running(ConfigurableApplicationContext context) {

context.publishEvent(new ApplicationReadyEvent(this.application, this.args, context));}

可以看到有两种广播方式，一种是当Spring还在启动的时候，通过监听器内部的SimpleApplicationEventMulticaster广播器进行广播；一种是当Spring启动完成内部的广播器可用时，直接调用上下文提供的接口进行广播。

## **继续分析Run**

了解了一些核心的接口后，就可以启动Debug模式运行Run方法了，由于涉及的方法调用很多，以下代码将拆分源码，并将方法签名记在前面。  
首先开启了一个秒表用来统计启动时间并在日志打印（如果开启控制字），声明了一些在后面需要用到的变量，然后开始初始化SpringApplicationRunListener类型的监听器，SpringApplicationRunListeners对监听器List进行了封装，例如调用.starting()时会遍历内部所有监听器调用其.starting()方法。

public ConfigurableApplicationContext run(String... args){

StopWatch stopWatch = new StopWatch();

stopWatch.start();

ConfigurableApplicationContext context = null;

Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();

configureHeadlessProperty();//开启设置，让系统模拟不存在io设备，略。。

SpringApplicationRunListeners listeners = getRunListeners(args);//初始化监听器

listeners.starting();

...

private SpringApplicationRunListeners getRunListeners(String[] args) {

Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class };//SpringApplicationRunListener的构造函数参数类型

return new SpringApplicationRunListeners(logger, getSpringFactoriesInstances(SpringApplicationRunListener.class, types, this, args));}

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, Object... args) {

ClassLoader classLoader = getClassLoader();//从当前线程获取类加载器

//Spring的类加载工具会从注册文件META-INF/spring.factories用指定的类加载器加载类，这里返回相应类型的实现类全限定名

Set<String> names = new LinkedHashSet<>(SpringFactoriesLoader.loadFactoryNames(type, classLoader));

List<T> instances = createSpringFactoriesInstances(type, parameterTypes, classLoader, args, names);//实例化

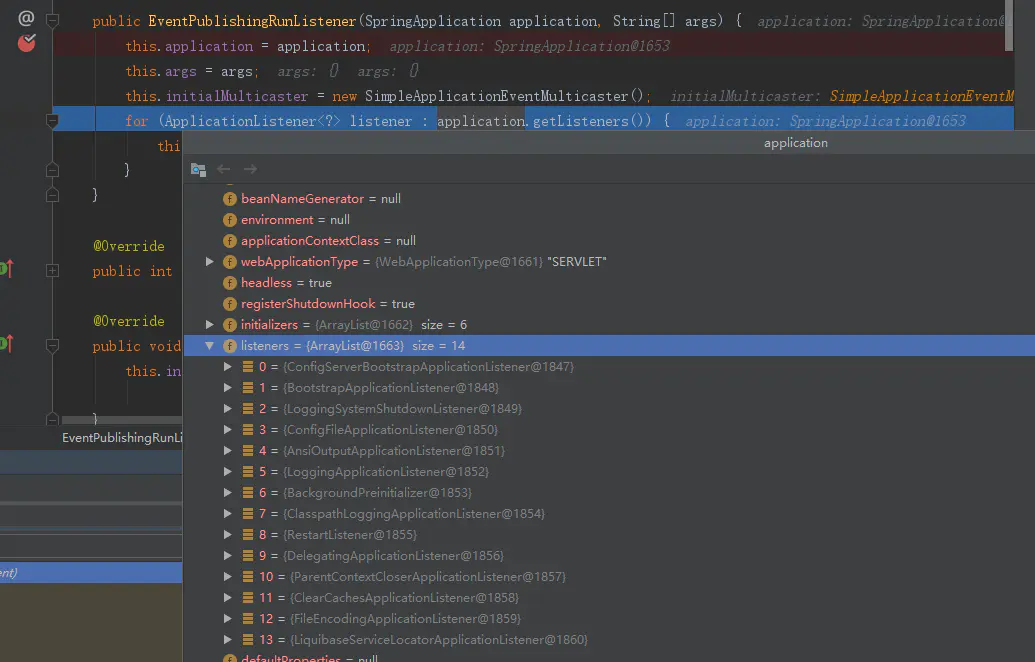
//Spring的排序工具，对继承了Ordered接口或者@Priority标记的类进行排序

AnnotationAwareOrderComparator.sort(instances);

return instances;

}

调试发现，注册为SpringApplicationRunListener的实现类只有EventPublishingRunListener，之前说过该注册器是一个用于广播Spring事件的广播器，进到构造函数中可以看到都有哪些监听器被绑定到了这个广播器中，这里每个监听器的作用就不再深入了，需要说的是，如果在项目中有什么需要集成到Spring的框架，可以注册SpringApplicationRunListener\ApplicationListener的实现类，监听Spring的不同启动事件并执行集成的逻辑。当然也有别的方法，例如：[Creating a Custom Starter with Spring Boot](https://links.jianshu.com/go?to=https://www.baeldung.com/spring-boot-custom-starter" \t "https://www.jianshu.com/p/_blank)。



EventPublishingRunListener构造函数.png

继续往下看run方法，这里重点是准备Environment的逻辑。首先Spring会根据web容器的类型新建一个ConfigurableEnvironment，不同的web容器类型的Environment会重载customizePropertySources方法，该方法会注入不同的propertySources，例如如果开启内嵌的Servlet容器，就会注入servlet context init params等相关的参数。接下来会对新建的Environment执行配置写入的逻辑，主要是把main方法中设置到SpringApplication的参数写入到Environment中，然后发布ApplicationEnvironmentPreparedEvent事件，做一些绑定后返回Environment。吐槽下Spring对Environment的处理这块的代码写得很深奥，看不懂~

try {

ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);//封装main方法的参数

//初始化填充Environment的参数

ConfigurableEnvironment environment = prepareEnvironment(listeners,applicationArguments);

configureIgnoreBeanInfo(environment);//设置获取BeanInfo的一个参数，有兴趣的可以去了解下Introspector.getBeanInfo(Class**<?**> beanClass, int flags)这个方法

...

private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners, ApplicationArguments applicationArguments) {

//新建\获取当前Environment实例

ConfigurableEnvironment environment = getOrCreateEnvironment();

configureEnvironment(environment, applicationArguments.getSourceArgs());//配置参数

listeners.environmentPrepared(environment);//发布事件

bindToSpringApplication(environment);//绑定"spring.main"为当前的application，做SpEL用

if (!this.isCustomEnvironment) {//转换environment的类型，但这里应该类型和deduce的相同不用转换

environment = new EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment, deduceEnvironmentClass());

}

//将现有的配置封装成ConfigurationPropertySourcesPropertySource，看起来是为了做SpEL的，看不懂~

ConfigurationPropertySources.attach(environment);

return environment;}protected void configureEnvironment(ConfigurableEnvironment environment, String[] args) {

if (this.addConversionService) {//默认开启，会注入一组转换工具，例如StringToDurationConverter

ConversionService conversionService = ApplicationConversionService.getSharedInstance();

environment.setConversionService(ConfigurableConversionService) conversionService);

}

configurePropertySources(environment, args);//如果main启动时设置了默认参数或者有命令行参数，则写入到environment中

configureProfiles(environment, args);//如果main启动时设置了profile，则写入到environment的ActiveProfiles中}

继续往下看run方法，这里会创建Spring的上下文实例，详情请看另一篇博客[Spring Boot Context分析](https://www.jianshu.com/p/578bb0906c18" \t "https://www.jianshu.com/p/_blank)，简而言之就是根据Web容器类型的不同来创建不用的上下文实例。

Banner printedBanner = printBanner(environment);//打应标语

context = createApplicationContext();//创建上下文实例//异常播报器，默认有org.springframework.boot.diagnostics.FailureAnalyzers

exceptionReporters = getSpringFactoriesInstances(SpringBootExceptionReporter.class, new Class[] { ConfigurableApplicationContext.class }, context);...

继续往下看run方法，接下来是对刚创建的上下文完成加载。加载过程先填充Environment以及设置的参数，然后执行注册到spring.factories的ApplicationContextInitializer切面，如果自己实现切面的话要注意这时context已经有的信息是什么。接着发布ApplicationContextInitializedEvent事件，然后加载bean，最后发布ApplicationPreparedEvent事件。

prepareContext(context, environment, listeners, applicationArguments,printedBanner);

...

private void prepareContext(ConfigurableApplicationContext context, ConfigurableEnvironment environment, SpringApplicationRunListeners listeners, ApplicationArguments applicationArguments, Banner printedBanner) {

context.setEnvironment(environment);

//如果application有设置beanNameGenerator、resourceLoader就将其注入到上下文中，并将转换工具也注入到上下文中

postProcessApplicationContext(context);

applyInitializers(context);//调用初始化的切面

listeners.contextPrepared(context);//发布ApplicationContextInitializedEvent事件

if (this.logStartupInfo) {//日志

logStartupInfo(context.getParent() == null);

logStartupProfileInfo(context);

}

// Add boot specific singleton beans

ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();

beanFactory.registerSingleton("springApplicationArguments", applicationArguments);//注入main方法的参数

if (printedBanner != null) {

beanFactory.registerSingleton("springBootBanner", printedBanner);

}

if (beanFactory instanceof DefaultListableBeanFactory) {

//如果bean名相同的话是否允许覆盖，默认为false，相同会抛出异常

((DefaultListableBeanFactory) beanFactory).setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);

}

// 这里获取到的是BootstrapImportSelectorConfiguration这个class，而不是自己写的启动来，这个class是在之前注册的BootstrapApplicationListener的监听方法中注入的

Set<Object> sources = getAllSources();

Assert.notEmpty(sources, "Sources must not be empty");

load(context, sources.toArray(new Object[0]));//加载sources 到上下文中

listeners.contextLoaded(context);//发布ApplicationPreparedEvent事件

}

回到run方法，在实例化上下文并完成相关配置后，会刷新上下文。

refreshContext(context);

...

AbstractApplicationContext

public void refresh() throws BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {

//记录启动时间、状态，web容器初始化其property，复制listener

prepareRefresh();

//这里返回的是context的BeanFactory

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

//beanFactory注入一些标准组件，例如ApplicationContextAwareProcessor，ClassLoader等

prepareBeanFactory(beanFactory);

try {

//给实现类留的一个钩子，例如注入BeanPostProcessors，这里是个空方法

postProcessBeanFactory(beanFactory);

// 调用切面方法

invokeBeanFactoryPostProcessors(beanFactory);

// 注册切面bean

registerBeanPostProcessors(beanFactory);

// Initialize message source for this context.

initMessageSource();

// bean工厂注册一个key为applicationEventMulticaster的广播器

initApplicationEventMulticaster();

// 给实现类留的一钩子，可以执行其他refresh的工作，这里是个空方法

onRefresh();

// 将listener注册到广播器中

registerListeners();

// 实例化未实例化的bean

finishBeanFactoryInitialization(beanFactory);

// 清理缓存，注入DefaultLifecycleProcessor，发布ContextRefreshedEvent

finishRefresh();

}

catch (BeansException ex) {

if (logger.isWarnEnabled()) {

logger.warn("Exception encountered during context initialization - " +

"cancelling refresh attempt: " + ex);

}

// Destroy already created singletons to avoid dangling resources.

destroyBeans();

// Reset 'active' flag.

cancelRefresh(ex);

// Propagate exception to caller.

throw ex;

}

finally {

// Reset common introspection caches in Spring's core, since we

// might not ever need metadata for singleton beans anymore...

resetCommonCaches();

}

}}

回到run方法，最后的逻辑就是发布启动完成的事件，并调用监听者的方法。

...

afterRefresh(context, applicationArguments);//给实现类留的钩子，这里是一个空方法。

stopWatch.stop();

if (this.logStartupInfo) {

new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);

}

listeners.started(context);//发布ApplicationStartedEvent事件

callRunners(context, applicationArguments);

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, listeners);

throw new IllegalStateException(ex);

}

try {

listeners.running(context);//发布ApplicationReadyEvent事件

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, null);

throw new IllegalStateException(ex);

}

return context;