**前言**

前面几章我们见识了SpringBoot为我们做的自动配置，确实方便快捷，但是对于新手来说，如果不大懂SpringBoot内部启动原理，以后难免会吃亏。所以这次博主就跟你们一起一步步揭开SpringBoot的神秘面纱，让它不在神秘。

**正文**

我们开发任何一个Spring Boot项目，都会用到如下的启动类

|  |  |
| --- | --- |
| 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。

**SpringBootApplication背后的秘密**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | @Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherited @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(excludeFilters = {  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) }) public @interface SpringBootApplication { ... } |

虽然定义使用了多个Annotation进行了原信息标注，但实际上重要的只有三个Annotation：

* @Configuration（@SpringBootConfiguration点开查看发现里面还是应用了@Configuration）
* @EnableAutoConfiguration
* @ComponentScan

所以，如果我们使用如下的SpringBoot启动类，整个SpringBoot应用依然可以与之前的启动类功能对等：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | @Configuration @EnableAutoConfiguration @ComponentScan public class Application {  public static void main(String[] args) {  SpringApplication.run(Application.class, args);  } } |

每次写这3个比较累，所以写一个@SpringBootApplication方便点。接下来分别介绍这3个Annotation。

**@Configuration**

这里的@Configuration对我们来说不陌生，它就是JavaConfig形式的Spring Ioc容器的配置类使用的那个@Configuration，SpringBoot社区推荐使用基于JavaConfig的配置形式，所以，这里的启动类标注了@Configuration之后，本身其实也是一个IoC容器的配置类。  
举几个简单例子回顾下，XML跟config配置方式的区别：

* 表达形式层面  
  基于XML配置的方式是这样：

|  |  |
| --- | --- |
| 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> |

而基于JavaConfig的配置方式是这样：

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

任何一个标注了@Configuration的Java类定义都是一个JavaConfig配置类。

* 注册bean定义层面  
  基于XML的配置形式是这样：

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

而基于JavaConfig的配置形式是这样的：

|  |  |
| --- | --- |
| 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。

* 表达依赖注入关系层面  
  为了表达bean与bean之间的依赖关系，在XML形式中一般是这样：

|  |  |
| --- | --- |
| 1 2 3 4 5 | <bean id="mockService" class="..MockServiceImpl">  <propery name ="dependencyService" ref="dependencyService" /> </bean>  <bean id="dependencyService" class="DependencyServiceImpl"></bean> |

而基于JavaConfig的配置形式是这样的：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | @Configuration public class MockConfiguration{  @Bean  public MockService mockService(){  return new MockServiceImpl(dependencyService());  }    @Bean  public DependencyService dependencyService(){  return new DependencyServiceImpl();  } } |

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

**@ComponentScan**

@ComponentScan这个注解在Spring中很重要，它对应XML配置中的元素，@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，最终将这些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。

**@EnableAutoConfiguration**

个人感觉@EnableAutoConfiguration这个Annotation最为重要，所以放在最后来解读，大家是否还记得Spring框架提供的各种名字为@Enable开头的Annotation定义？比如@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。
* @EnableMBeanExport是通过@Import将JMX相关的bean定义加载到IoC容器。

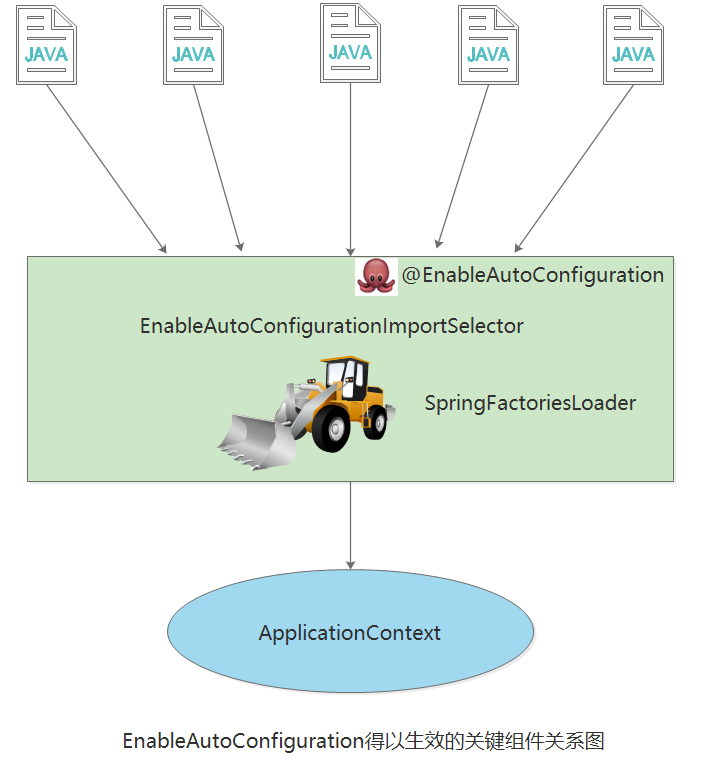
而@EnableAutoConfiguration也是借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器，仅此而已！

@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 @Import(EnableAutoConfigurationImportSelector.class) public @interface EnableAutoConfiguration {  ... } |

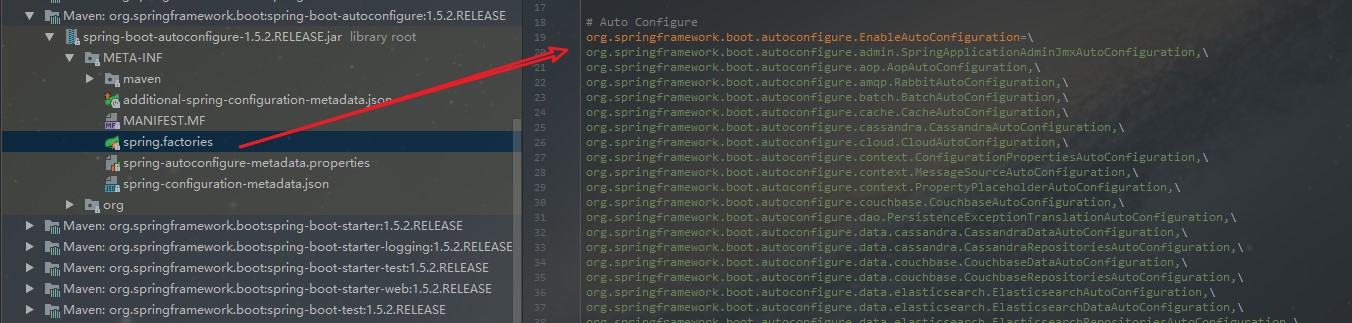
其中，最关键的要属@Import(EnableAutoConfigurationImportSelector.class)，借助EnableAutoConfigurationImportSelector，@EnableAutoConfiguration可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。就像一只“八爪鱼”一样

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！

[](http://7xqch5.com1.z0.glb.clouddn.com/springboot3-1.png)

**自动配置幕后英雄：SpringFactoriesLoader详解**  
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。

|  |  |
| --- | --- |
| 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  
[](http://7xqch5.com1.z0.glb.clouddn.com/springboot3-2.jpg)  
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。

所以，@EnableAutoConfiguration自动配置的魔法骑士就变成了：**从classpath中搜寻所有的META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableutoConfiguration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。**

**深入探索SpringApplication执行流程**

SpringApplication的run方法的实现是我们本次旅程的主要线路，该方法的主要流程大体可以归纳如下：

1） 如果我们使用的是SpringApplication的静态run方法，那么，这个方法里面首先要创建一个SpringApplication对象实例，然后调用这个创建好的SpringApplication的实例方法。在SpringApplication实例初始化的时候，它会提前做几件事情：

* 根据classpath里面是否存在某个特征类（org.springframework.web.context.ConfigurableWebApplicationContext）来决定是否应该创建一个为Web应用使用的ApplicationContext类型。
* 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationContextInitializer。
* 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationListener。
* 推断并设置main方法的定义类。

2） SpringApplication实例初始化完成并且完成设置后，就开始执行run方法的逻辑了，方法执行伊始，首先遍历执行所有通过SpringFactoriesLoader可以查找到并加载的SpringApplicationRunListener。调用它们的started()方法，告诉这些SpringApplicationRunListener，“嘿，SpringBoot应用要开始执行咯！”。

3） 创建并配置当前Spring Boot应用将要使用的Environment（包括配置要使用的PropertySource以及Profile）。

4） 遍历调用所有SpringApplicationRunListener的environmentPrepared()的方法，告诉他们：“当前SpringBoot应用使用的Environment准备好了咯！”。

5） 如果SpringApplication的showBanner属性被设置为true，则打印banner。

6） 根据用户是否明确设置了applicationContextClass类型以及初始化阶段的推断结果，决定该为当前SpringBoot应用创建什么类型的ApplicationContext并创建完成，然后根据条件决定是否添加ShutdownHook，决定是否使用自定义的BeanNameGenerator，决定是否使用自定义的ResourceLoader，当然，最重要的，将之前准备好的Environment设置给创建好的ApplicationContext使用。

7） ApplicationContext创建好之后，SpringApplication会再次借助Spring-FactoriesLoader，查找并加载classpath中所有可用的ApplicationContext-Initializer，然后遍历调用这些ApplicationContextInitializer的initialize（applicationContext）方法来对已经创建好的ApplicationContext进行进一步的处理。

8） 遍历调用所有SpringApplicationRunListener的contextPrepared()方法。

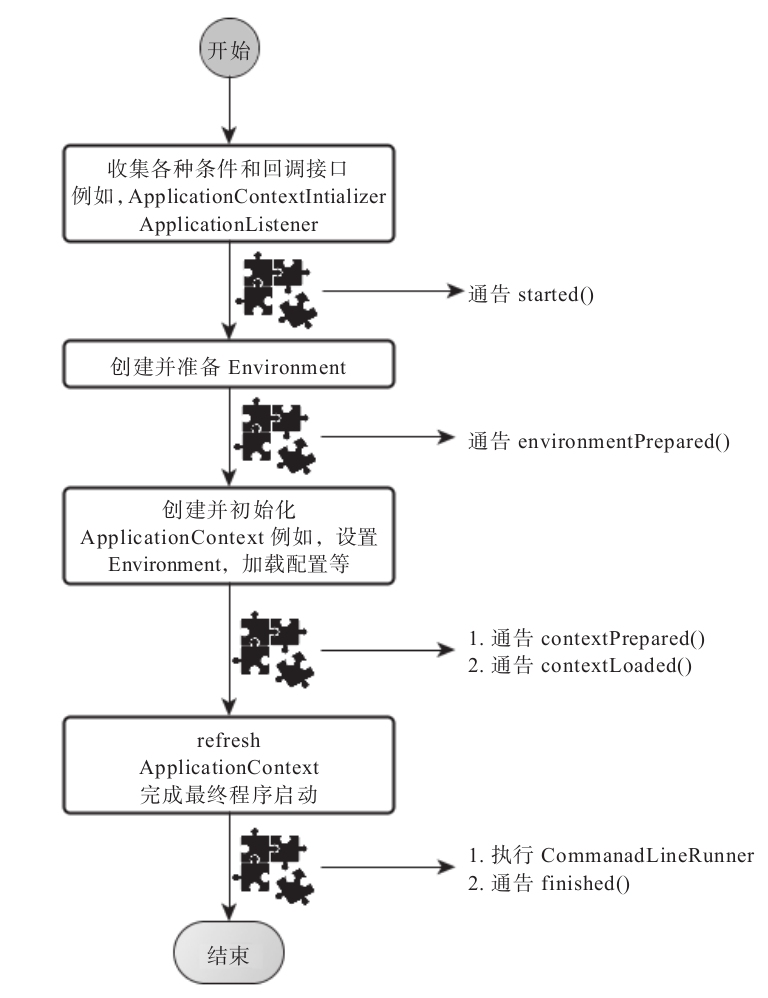
9） 最核心的一步，将之前通过@EnableAutoConfiguration获取的所有配置以及其他形式的IoC容器配置加载到已经准备完毕的ApplicationContext。

10） 遍历调用所有SpringApplicationRunListener的contextLoaded()方法。

11） 调用ApplicationContext的refresh()方法，完成IoC容器可用的最后一道工序。

12） 查找当前ApplicationContext中是否注册有CommandLineRunner，如果有，则遍历执行它们。

13） 正常情况下，遍历执行SpringApplicationRunListener的finished()方法、（如果整个过程出现异常，则依然调用所有SpringApplicationRunListener的finished()方法，只不过这种情况下会将异常信息一并传入处理）  
去除事件通知点后，整个流程如下：

[](http://7xqch5.com1.z0.glb.clouddn.com/springboot3-3.jpg)

**总结**

到此，SpringBoot的核心组件完成了基本的解析，综合来看，大部分都是Spring框架背后的一些概念和实践方式，SpringBoot只是在这些概念和实践上对特定的场景事先进行了固化和升华，而也恰恰是这些固化让我们开发基于Sping框架的应用更加方便高效。

想要查看更多Spring Boot干货教程,可前往：[Spring Boot干货系列总纲](http://tengj.top/2017/04/24/springboot0/)

**参考**

本章节大部分参考了《SpringBoot揭秘快速构建为服务体系》这本书的第三章，个人看过的几本书就感觉这本书介绍原理的章节最为透彻，本章也算这本书最精华的部分。所以我没忍住就分享出来给大家学习。当然主要是我也没有这本书的电子版，无法分享给大家了，深表遗憾(。・＿・。)ﾉ