

狂神说SpringBoot02：运行原理初探

狂神说 狂神说 2020-03-09

狂神说SpringBoot系列连载课程，通俗易懂，基于SpringBoot2.2.5版本，欢迎各位狂粉转发关注学习。未经作者授权，禁止转载

运行原理探究

我们之前写的HelloSpringBoot，到底是怎么运行的呢，Maven项目，我们一般从pom.xml文件探究起：

pom.xml

父依赖

其中它主要是依赖一个父项目，主要是管理项目的资源过滤及插件！

```
<parent>    <groupId>org.springframework.boot</groupId>    <artifactId>spring-boot-starter-parent</artifactId>    <version>2.2.5.RELEASE</version>    <relativePath/>    <!-- lookup parent from repository --></parent>
```

点进去，发现还有一个父依赖

```
<parent>    <groupId>org.springframework.boot</groupId>    <artifactId>spring-boot-dependencies</artifactId>    <version>2.2.5.RELEASE</version>    <relativePath>../../spring-boot-dependencies</relativePath></parent>
```

这里才是真正管理SpringBoot应用里面所有依赖版本的地方，SpringBoot的版本控制中心；

以后我们导入依赖默认是不需要写版本；但是如果导入的包没有在依赖中管理着就需要手动配置版本了；

启动器 spring-boot-starter

```
<dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-web</artifactId></dependency>
```

springboot-boot-starter-xxx: 就是spring-boot的场景启动器

spring-boot-starter-web: 帮我们导入了web模块正常运行所依赖的组件;

SpringBoot将所有的功能场景都抽取出来，做成一个个的**starter**（启动器），只需要在项目中引入这些**starter**即可，所有相关的依赖都会导入进来，我们要用什么功能就导入什么样的场景启动器即可；我们未来也可以自己自定义 **starter**;

主启动类

分析完了 pom.xml 来看看这个启动类

默认的主启动类

```
//@SpringBootApplication 来标注一个主程序类//说明这是一个Spring Boot应用@SpringBootApplicationpublic class SpringbootApplication {  
    public static void main(String[] args) {        //以为是启动了一个方法，没想到启动了一个服务        SpringApplication.run(SpringbootApplication.class, args);  
    }  
}
```

但是一个简单的启动类并不简单！我们来分析一下这些注解都干了什么

@SpringBootApplication

作用：标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用；

进入这个注解：可以看到上面还有很多其他注解！

```
@SpringBootApplication@EnableAutoConfiguration@ComponentScan(excludeFilters = {@Filter(type = FilterType.CUSTOM, classes = {TypeExcludeFilter.class}), @Filter(type = FilterType.CUSTOM, classes = {AutoConfigurationExcludeFilter.class})})public @interface SpringBootApplication {  
    // .....  
}
```

| @ComponentScan

这个注解在Spring中很重要,它对应XML配置中的元素。

作用：自动扫描并加载符合条件的组件或者bean，将这个bean定义加载到IOC容器中

| @SpringBootApplication

作用：SpringBoot的配置类，标注在某个类上，表示这是一个SpringBoot的配置类；

我们继续进去这个注解查看

```
// 点进去得到下面的 @Component@Configurationpublic @interface SpringBootConfiguration {}  
  
@Componentpublic @interface Configuration {}
```

这里的 `@Configuration`，说明这是一个配置类，配置类就是对应Spring的xml 配置文件；

里面的 `@Component` 这就说明，启动类本身也是Spring中的一个组件而已，负责启动应用！

我们回到 `SpringBootApplication` 注解中继续看。

! @EnableAutoConfiguration

@EnableAutoConfiguration：开启自动配置功能

以前我们需要自己配置的东西，而现在SpringBoot可以自动帮我们配置；`@EnableAutoConfiguration`告诉SpringBoot开启自动配置功能，这样自动配置才能生效；

点进注解接续查看：

@AutoConfigurationPackage：自动配置包

```
@Import({Registrar.class})public @interface AutoConfigurationPackage {}
```

@import：Spring底层注解**@import**，给容器中导入一个组件

`Registrar.class` 作用：将主启动类的所在包及包下面所有子包里面的所有组件扫描到Spring容器；

这个分析完了，退到上一步，继续看

@Import({AutoConfigurationImportSelector.class})：给容器导入组件；

`AutoConfigurationImportSelector`：自动配置导入选择器，那么它会导入哪些组件的选择器呢？我们点击去这个类看源码：

1、这个类中有一个这样的方法

```
// 获得候选的配置protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {    //这里的getSpringFactoriesLoaderFactoryClass()方法    //返回的就是我们最开始看的启动自动导入配置文件的注解类; EnableAutoConfiguration    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;}
```

2、这个方法又调用了 **SpringFactoriesLoader** 类的静态方法！我们进入**SpringFactoriesLoader**类 **loadFactoryNames()** 方法

```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable
    ClassLoader classLoader) {    String factoryClassName = factoryClass.getName();    //这里它又调用了 loadSpringFactories 方法    return (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList())
    ;}
```

3、我们继续点击查看 **loadSpringFactories** 方法

```

private static Map<String, List<String>> loadSpringFactories(@Nullable Class
sLoader classLoader) {    //获得classLoader , 我们返回可以看到这里得到的就是Enabl
eAutoConfiguration标注的类本身    MultiValueMap<String, String> result = (Mul
tiValueMap)cache.get(classLoader);    if (result != null) {        return r
esult;    } else {        try {            //去获取一个资源 "META-INF/spring.f
actories"            Enumeration<URL> urls = classLoader != null ? classLoa
der.getResources("META-INF/spring.factories") : ClassLoader.getSystemResour
ces("META-INF/spring.factories");            LinkedMultiValueMap result = n
ew LinkedMultiValueMap();

            //将读取到的资源遍历, 封装成为一个Properties            while(urls.ha
sMoreElements()) {                URL url = (URL)urls.nextElement();

                UrlResource resource = new UrlResource(url);                Proper
ties properties = PropertiesLoaderUtils.loadProperties(resource);

                Iterator var6 = properties.entrySet().iterator();

                while(var6.hasNext()) {                    Entry<?, ?> entr
y = (Entry)var6.next();                    String factoryClassName = ((Stri
ng)entry.getKey()).trim();                    String[] var9 = StringUtils.c
ommaDelimitedListToStringArray((String)entry.getValue());

                    int var10 = var9.length;

                    for(int var11 = 0; var11 < var10; ++var11) {

                        String factoryName = var9[var11];                        resul
t.add(factoryClassName, factoryName.trim());                    }

                }

                cache.put(classLoader, result);                return result;

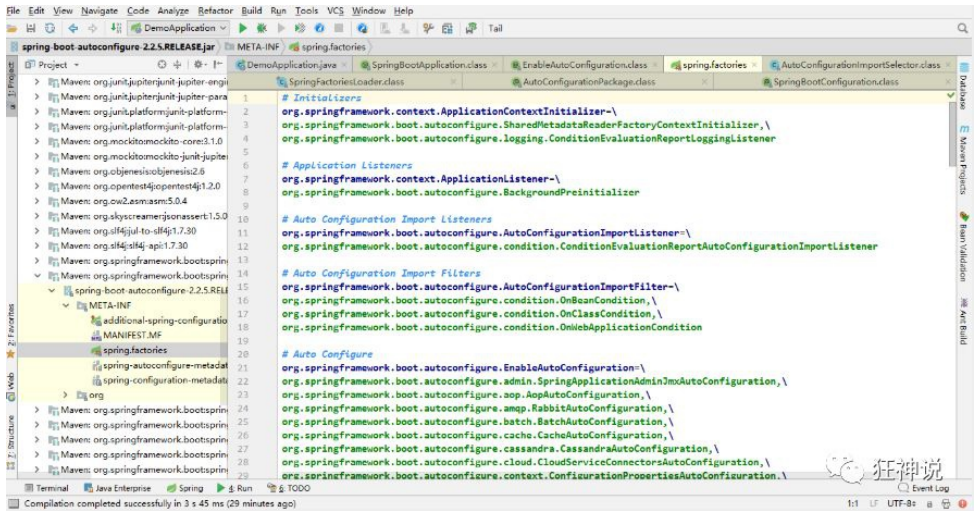
            } catch (IOException var13) {                throw new IllegalArgumentExcepti
on("Unable to load factories from location [META-INF/spring.factories]", va
r13);            }        }
}

```

4、发现一个多次出现的文件：**spring.factories**，全局搜索它

spring.factories

我们根据源头打开**spring.factories**，看到了很多自动配置的文件；这就是自动配置根源所在！



WebMvcAutoConfiguration

我们在上面的自动配置类随便找一个打开看看，比如：WebMvcAutoConfiguration



可以看到这些一个个的都是JavaConfig配置类，而且都注入了一些Bean，可以找一些自己认识的类，看着熟悉一下！

所以，自动配置真正实现是从classpath中搜寻所有的META-INF/spring.factories配置文件，并将其中对应的org.springframework.boot.autoconfigure.包下的配置项，通过反射实例化为对应标注了@Configuration的JavaConfig形式的IOC容器配置类，然后将这些都汇总成为一个实例并加载到IOC容器中。

结论：

1. SpringBoot在启动的时候从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值

2. 将这些值作为自动配置类导入容器，自动配置类就生效，帮我们进行自动配置工作；
3. 整个J2EE的整体解决方案和自动配置都在springboot-autoconfigure的jar包中；
4. 它会给容器中导入非常多的自动配置类（xxxAutoConfiguration），就是给容器中导入这个场景需要的所有组件，并配置好这些组件；
5. 有了自动配置类，免去了我们手动编写配置注入功能组件等的工作；

现在大家应该大概的了解了下，**SpringBoot**的运行原理，后面我们还会深化一次！

SpringApplication

不简单的方法

我最初以为就是运行了一个main方法，没想到却开启了一个服务；

```
@SpringBootApplicationpublic class SpringbootApplication {    public static    void main(String[] args) {        SpringApplication.run(SpringbootApplication.class, args);    }}
```

SpringApplication.run分析

分析该方法主要分两部分，一部分是SpringApplication的实例化，二是run方法的执行；

SpringApplication

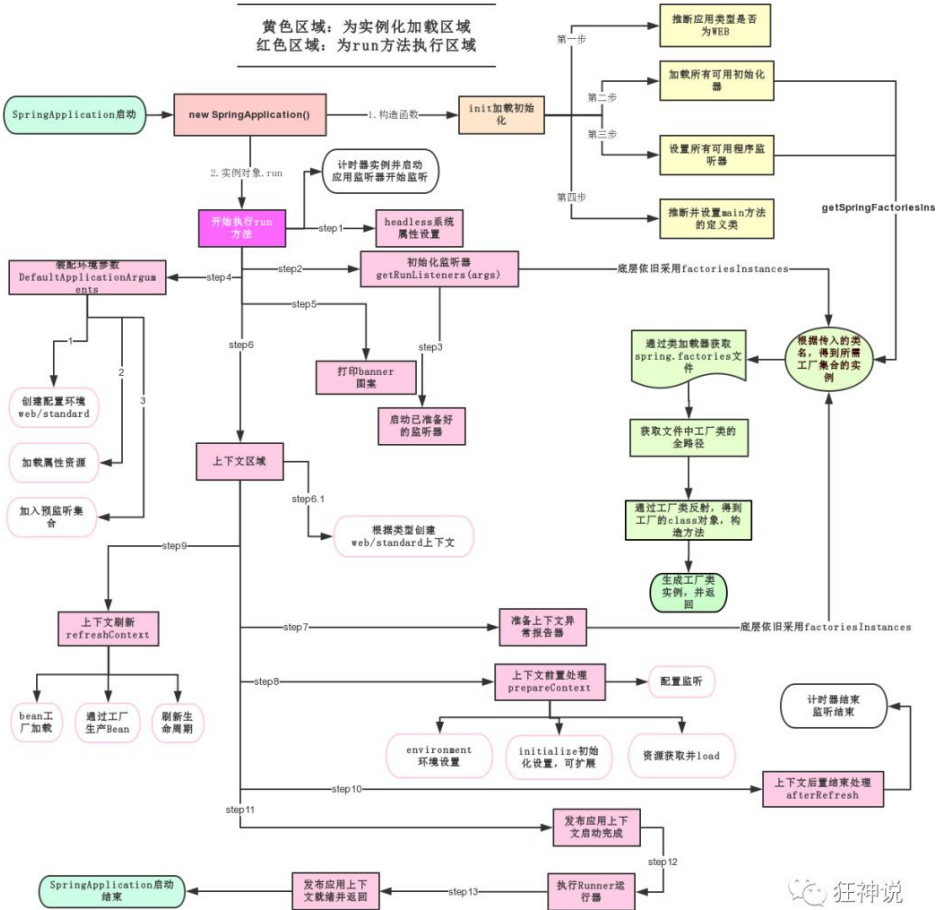
这个类主要做了以下四件事情：

- 1、推断应用的类型是普通的项目还是Web项目
- 2、查找并加载所有可用初始化器，设置到initializers属性中
- 3、找出所有的应用程序监听器，设置到listeners属性中
- 4、推断并设置main方法的定义类，找到运行的主类

查看构造器：


```
public SpringApplication(ResourceLoader resourceLoader, Class... primarySources) {    // .....    this.webApplicationType = WebApplicationType.deduceFromClasspath();    this.setInitializers(this.getSpringFactoriesInstances());    this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));    this.mainApplicationClass = this.deduceMainApplicationClass();}
```

run方法流程分析



跟着源码和这幅图就可以一探究竟了！



🔊 长按关注

据说关注小狂神的人都
走向人生巅峰了，还不
长按关注一下？

×

×

👤 狂神说

