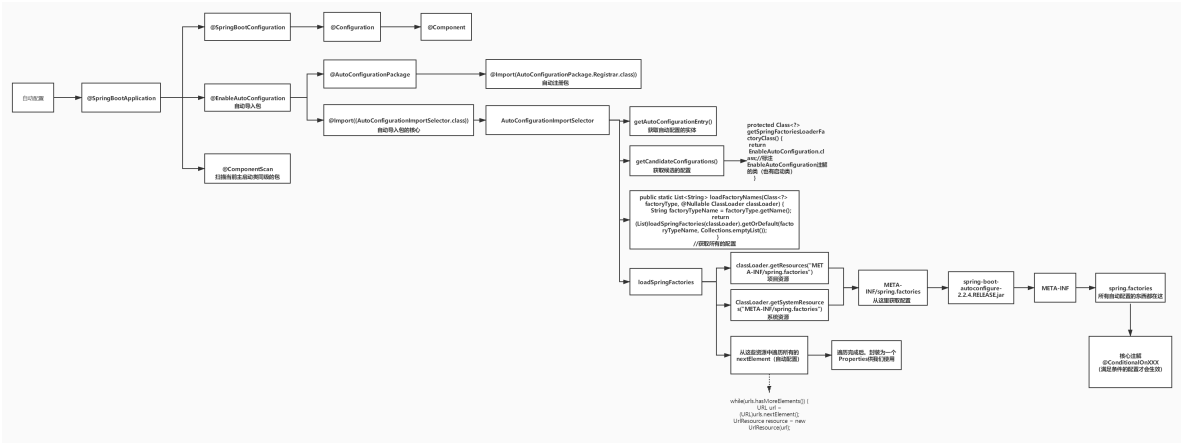
原理初识——自动配置

1 自动配置分析流程图



2 pom.xml

2.1 spring-boot-dependencies

核心依赖在**父工程**里===》依赖一个父项目。

- 我们在写入或者引入一些SpringBoot依赖的时候，不需要指定版本，因为有这些版本仓库（我们可以点parant下的artifactId进去看）

  所有依赖：https://docs.spring.io/spring-boot/docs/2.2.4.RELEASE/reference/html/appendix-dependency-versions.html#appendix-dependency-versions

```
<properties>
  <activemq.version>5.15.11</activemq.version>
  <antlr2.version>2.7.7</antlr2.version>
  <appengine-sdk.version>1.9.77</appengine-sdk.version>
  <artemis.version>2.10.1</artemis.version>
  <aspectj.version>1.9.5</aspectj.version>
  <assertj.version>3.13.2</assertj.version>
  <atomikos.version>4.0.6</atomikos.version>
  <awaitility.version>4.0.2</awaitility.version>
  <bitronix.version>2.1.4</bitronix.version>
  <build-helper-maven-plugin.version>3.0.0</build-helper-maven-plugin.vers
  <byte-buddy.version>1.10.6</byte-buddy.version>
  <caffeine.version>2.8.0</caffeine.version>
  <cassandra-driver.version>3.7.2</cassandra-driver.version>
  <classmate.version>1.5.1</classmate.version>
  <commons-codec.version>1.13</commons-codec.version>
  <commons-dbcp2.version>2.7.0</commons-dbcp2.version>
  <commons-lang3.version>3.9</commons-lang3.version>
```

2.2 启动器starter

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

- 启动器===》就是SpringBoot的启动场景
- 比如spring-boot-starter-web，就会帮我们自动导入web环境需要的所有依赖
- SpringBoot会将所有的功能场景，都变成一个个的启动器
  - 我们要使用什么功能，只需要引入对应的启动器即可。
  - 所有starter：https://docs.spring.io/spring-boot/docs/2.2.4.RELEASE/reference/html/using-spring-boot.html#using-boot-starter

3 主程序（启动类）

```java
/**
 * @SpringBootApplication 标注这个类是一个SpringBoot的应用
 *                        启动类下的所有资源被导入（@EnableAutoConfiguration）
 */
@SpringBootApplication
public class Springboot01HellowordApplication {
    public static void main(String[] args) {
        //将SpringBoot应用启动（使用反射）
        SpringApplication.run(Springboot01HellowordApplication.class, args);
    }
}
```

- 注解（点击查看@SpringBootApplication的源码）

  有如下两个比较重要的注解：

  ```
  @SpringBootConfiguration：SpringBoot的配置
      @Configuration：Spring配置类
          @Component：说明这也是一个Spring的组件

  @EnableAutoConfiguration ：自动配置
      @AutoConfigurationPackage ：  自动配置包（自动扫描包下的所有配置）
          @Import(AutoConfigurationPackage.Registrar.class) ：  自动配置“包注册”（包的位
  置严格）
      @Import({AutoConfigurationImportSelector.class})：自动配置导入选择
  ```

  - 源码中有一句话： 用于获取所有的配置

  ```
  List<String> configurations =
  this.getCandidateConfigurations(annotationMetadata, attributes);
  ```

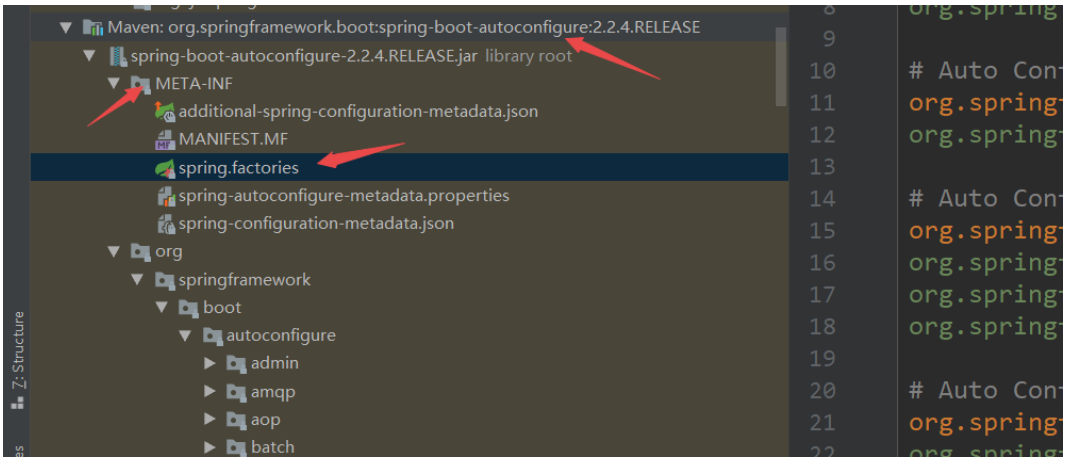
  - 获取候选的配置：

```
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;
}
```

- 自动配置的核心文件：META-INF/spring.factories

```
 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.");
```

查看一下位置：



【注意：这个配置文件十分重要！】

```
Properties properties = PropertiesLoaderUtils.loadProperties(resource);
```

将所有的资源加载到配置文件类中。

```
Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") : ClassLoader.getSystemResources("META-INF/spring.factories");
```

3.1 @SpringBootApplication

SpringBoot应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；

进入源码：

```
@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 {
```

3.2 @ComponentScan

这个注解在Spring中很重要， 它**对应XML配置中的元素。**@ComponentScan的功能就是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；

3.3 @SpringBootConfiguration

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

点进源码:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
    @AliasFor(
        annotation = Configuration.class
    )
    boolean proxyBeanMethods() default true;
}
```

- @Configuration：配置类上来标注这个注解，说明这是一个配置类 ， 配置类---即----配置文件;
-

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";

    boolean proxyBeanMethods() default true;
}
```

我们继续点进去，发现配置类也是**容器中的一个组件**（@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！

3.4 @EnableAutoConfiguration

- 功能：开启自动配置
- 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；
- 点进去可以发现两个注解：
  - @AutoConfigurationPackage —— 自动配置包
    - 继续点进去

      ```java
      @Target({ElementType.TYPE})
      @Retention(RetentionPolicy.RUNTIME)
      @Documented
      @Inherited
      @Import({Registrar.class})
      public @interface AutoConfigurationPackage {
      }
      ```

      可以看到有个@Import({Registrar.class})——Spring底层注解@import， 给容器中导入一个组件

      Registrar.class 将主配置类 【即@SpringBootApplication标注的类】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
  - @Import({AutoConfigurationImportSelector.class}) —— 导入哪些组件的选择器
- 看AutoConfigurationImportSelector的源码（重要）
  - 有个方法

```
// 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;
}
```

- 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());
}
```

- 继续点击查看 loadSpringFactories 方法

```
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
    //获得classLoader ， 我们返回可以看到这里得到的就是EnableAutoConfiguration标注的类本身
    MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);
    if (result != null) {
        return result;
    } else {
        try {
            //去获取一个资源 "META-INF/spring.factories"
            Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") : ClassLoader.getSystemResources("META-INF/spring.factories");
            LinkedMultiValueMap result = new LinkedMultiValueMap();

            //将读取到的资源遍历，封装成为一个Properties
            while(urls.hasMoreElements()) {
                URL url = (URL)urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                Iterator var6 = properties.entrySet().iterator();

                while(var6.hasNext()) {
                    Entry<?, ?> entry = (Entry)var6.next();
```

```
                    String factoryClassName =
((String)entry.getKey()).trim();
                    String[] var9 =
StringUtils.commaDelimitedListToStringArray((String)entry.getValue(
));
                    int var10 = var9.length;

                    for(int var11 = 0; var11 < var10; ++var11) {
                        String factoryName = var9[var11];
                        result.add(factoryClassName,
factoryName.trim());
                    }
                }
            }

            cache.put(classLoader, result);
            return result;
        } catch (IOException var13) {
            throw new IllegalArgumentException("Unable to load
factories from location [META-INF/spring.factories]", var13);
        }
    }
}
```

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

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。

3.5 run方法

```
//将SpringBoot应用启动（使用反射）
//该方法返回一个ConfigurableApplicationContext对象
//参数一：应用入口的类      参数类：命令行参数
SpringApplication.run(Springboot01HellowordApplication.class, args);
```

3.5.1 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

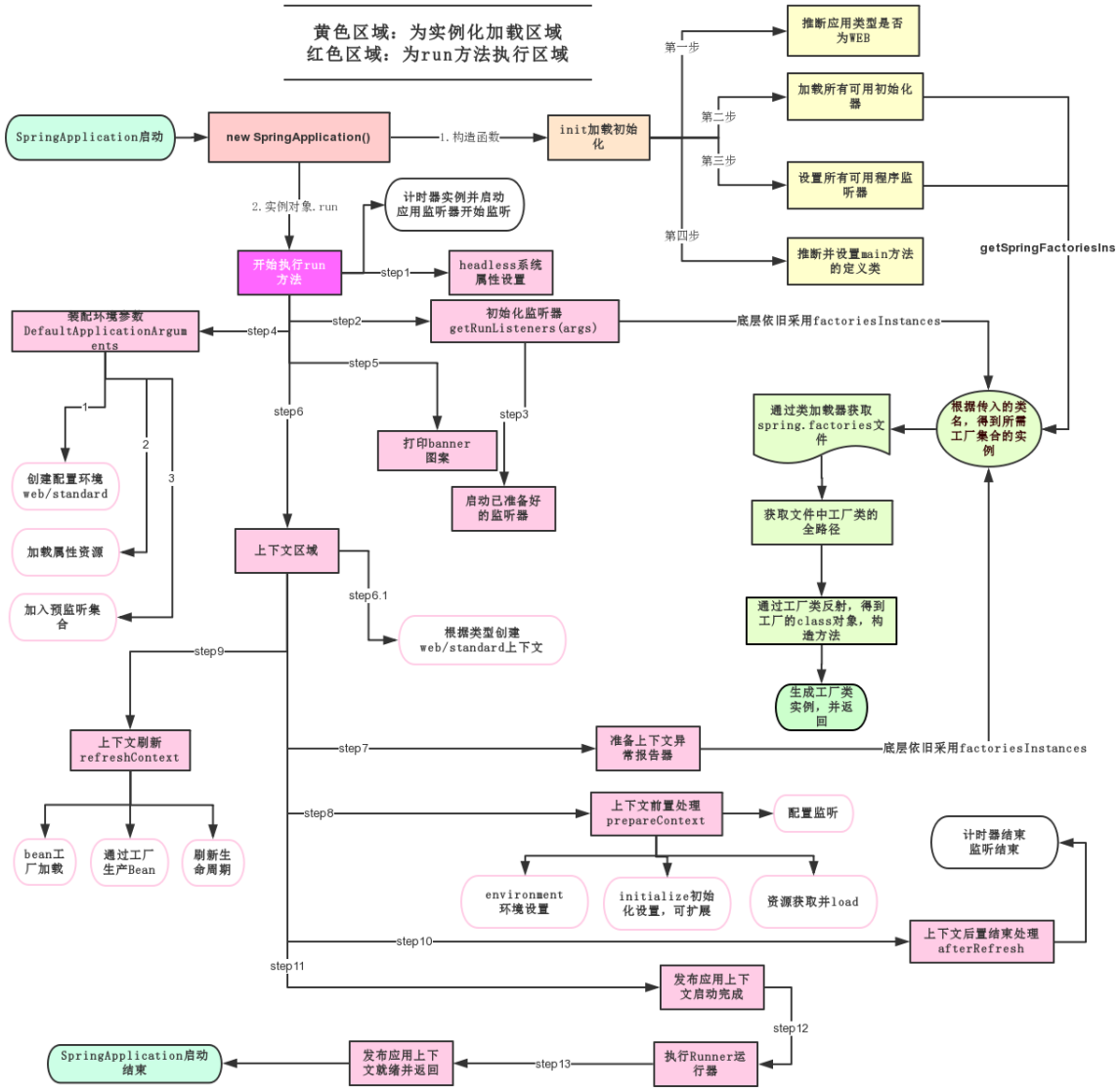
看一下构造器：

```
public SpringApplication(ResourceLoader resourceLoader, Class...
primarySources) {
    this.sources = new LinkedHashSet();
    this.bannerMode = Mode.CONSOLE;
    this.logStartupInfo = true;
    this.addCommandLineProperties = true;
    this.addConversionService = true;
    this.headless = true;
    this.registerShutdownHook = true;
    this.additionalProfiles = new HashSet();
    this.isCustomEnvironment = false;
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();

 this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextIn
itializer.class));

 this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.cla
ss));
    this.mainApplicationClass = this.deduceMainApplicationClass();
}
```

3.5.2 run方法执行流程图



4 结论

SpringBoot的所有自动配置都在启动类中被扫描并加载（spring.factories所有的自动配置都在这里面），但是不一定全部生效：要判断条件是否成立，只要导入了对应的start，就有对应的启动器，有了启动器，我们自动装配就会生效，然后才配置成功。

1. SpringBoot在启动的时候，从类路径下/META-INF/spring.factories获取指定的值
2. 将这些自动配置的类导入容器，自动配置就会生效，帮我们进行自动配置
3. 以前我们需要自动配置的东西，现在SpringBoot帮我们做了
4. 整个JavaEE、解决方案和自动配置的东西都在spring-boot-autoconfigure-2.2.4.RELEASE.jar这个包下

   它会把所有需要导入的组件，以类名的方式返回，这些组件就会被添加到容器
5. 容器中也会存在非常多的xxxAutoConfiguration的文件（@Bean），就是这些类给容器中导入了这个场景需要的所有组件，并自动配置@Configuration，javaConfig
6. 有了自动配置类，免去了我们手动编写配置文件的工作。