

Part II. SpringBoot 原理深入及源码剖析

1. 依赖管理

1.1 为什么在pom.xml中导入dependency时不需要指定版本？

- `spring-boot-starter-parent` 有version，在它的父类 `spring-boot-dependencies` 中，用为该version导入了一系列的依赖文件版本，从而对一些常用技术框架的依赖文件进行了统一版本号管理。
【From 随堂，注意：这里并非为项目提供整合的依赖文件本身（感觉更像是reference的东西）】
- 在自己工程中的pom文件中若引入的依赖文件不归 `spring-boot-starter-parent` 管理，需要写明其version。

1.2 项目本身运行时所依赖的Jar包来自于哪里？

项目含有Controller，所以使用了SpringMVC的jar包，在spring-boot-starter-parent中，有dependency named spring-boot-starter-web，其含有spring-webmvc的dependency，所以，通过依赖传递，自定义pom中就不需要额外引入tomcat服务器及其他web依赖文件等。

Spring-boot-starter-xxx 官方指定了[各种场景依赖启动器](#)，但有的技术框架（比如Mybatis、Druid数据源等）SpringBoot官方并没有提供相应的starter，使用时需要引入第三方自己写的依赖启动器，并且记得配置相应版本号。

2. 自动配置启动流程

自动配置启动流程总结

1. SpringBoot 应用启动
2. @SpringBootApplication，组合注解（包含3个）
3. @SpringBootConfiguration: 标明该类为配置类
 1. @Configuration：通过java config的方式来添加组件的IoC容器中
4. @EnableAutoConfiguration
 1. @AutoConfigurationPackage：使用注解
@Import(AutoConfigurationPackages.Registrar.class) 将 Registrar 类导入容器中。
 1. Registrar类作用：扫描主配置类（即项目启动类，即被 @SpringBootApplication 标示的类）同级目录pkg以及子pkg，与 @ComponentScan 扫描到的 添加到IoC容器中

2. @Import(AutoConfigurationImportSelector.class): 将 AutoConfigurationImportSelector 类导入容器中。
 1. AutoConfigurationImportSelector 类作用:
 1. 通过 selectImports 方法, 使用内部工具类 SpringFactoriesLoader, 查找 classpath 上所有jar包中的 META-INF/spring.factories (这里是在:
/Users/yusihuang/.m2/repository/org/springframework/boot/spring-boot-autoconfigure/2.2.7.RELEASE/spring-boot-autoconfigure-2.2.7.RELEASE.jar!/META-INF/spring.factories) 并加载其中的bean (这些bean就是自动配置类, 他们的类名上都有 @Conditionalxxx 注解)。
 2. 加载时会有过滤操作, 最终获得 a list of 全类名 of 自动配置类。
 3. 将这个list 交给 SpringFactory 加载器从而继续将这些bean 加入 IoC 容器中。
3. @ComponentScan: 包扫描器

SpringBoot到底如何进行自动配置? 都把哪些组件进行了自动配置?

入口 @SpringBootApplication

@SpringBootApplication 能扫描Spring组件并自动配置SpringBoot。

是个组合注解, 包含3个:

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

```
@Target(ElementType.TYPE)    // 注解的适用范围, Type表示注解可以描述在类、接口、注解或枚举中
@Retention(RetentionPolicy.RUNTIME) // 表示注解的生命周期, Runtime运行时
@Documented // 表示注解可以记录在javadoc中
@Inherited // 表示可以被子类继承该注解

@SpringBootApplication // 标明该类为配置类 => 点它
@EnableAutoConfiguration // 启动自动配置功能
@ComponentScan(excludeFilters = { // 包扫描器 <context:component-scan base-package="com.xxx.xxx" />
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =
        AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

2.1 @SpringBootConfiguration

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented

@Configuration // 配置类(核心注解, 由Spring框架提供, 可被组件扫描器识别的配置类)
public @interface SpringBootConfiguration {

}

```

2.2 @EnableAutoConfiguration

相当于自动配置的开关。

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited

@AutoConfigurationPackage //自动配置包 : 会把@SpringbootApplication注解标注的类所在包名拿到, 并且对该包及其子包进行扫描, 将组件添加到容器中
@Import(AutoConfigurationImportSelector.class) //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(ApplicationContext)中
public @interface EnableAutoConfiguration {

```

是个组合注解, 包含2个:

2.2.1 @AutoConfigurationPackage

作用: 把 @SpringbootApplication 注解标注的类所在包名拿到, 并且对该包及其子包进行扫描, 将组件添加到容器中。

```

//spring框架的底层注解, 它的作用就是给容器中导入某个组件类,
//例如@Import(AutoConfigurationPackages.Registrar.class), 它就是将Registrar这个组件类导入到容器中
@Import(AutoConfigurationPackages.Registrar.class) // 默认将主配置类
(@SpringBootApplication)所在的包及其子包里面的所有组件扫描到Spring容器中
public @interface AutoConfigurationPackage {

}

```

2.2.2 @Import(AutoConfigurationImportSelector.class)

作用：给容器中导入某个组件Class，是Spring框架的底层注解

```
public class AutoConfigurationImportSelector implements ...{

...

// 这个方法告诉springboot都需要导入那些组件
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    //判断 enableautoconfiguration注解有没有开启，默认开启（是否进行自动装配）
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    //1. 加载配置文件META-INF/spring-autoconfigure-metadata.properties，从中获取所有支持自动配置类的条件
    //作用：SpringBoot使用一个Annotation的处理器来收集一些自动装配的条件，那么这些条件可以在META-INF/spring-autoconfigure-metadata.properties进行配置。
    // SpringBoot会将收集好的@Configuration进行一次过滤进而剔除不满足条件的配置类
    // 自动配置的全名.条件=值
    AutoConfigurationMetadata autoConfigurationMetadata =
AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
    AutoConfigurationEntry autoConfigurationEntry =
getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}

}
```

- AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader)

`spring-autoconfigure-metadata.properties` 中的 metadata format: 自动配置的全名.条件 = 值

总结一下判断是否要加载某个类的两种方式：

1. 根据spring-autoconfigure-metadata.properties进行判断。

所有自动配置类名在spring.factories文件中 （
/Users/yusihuang/.m2/repository/org/springframework/boot/spring-boot-autoconfigure/2.2.7.RELEASE/spring-boot-autoconfigure-2.2.7.RELEASE.jar!/META-INF/spring.factories)

自动配置的全名.条件=值 ==> 只有满足了相应条件时，才会加载该自动配置类。

2. 要判断@Conditional是否满足。如@ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class }) 表示需要在类路径中存在SqlSessionFactory.class、SqlSessionFactoryBean.class 这两个类才能完成自动注册。

2.3 @ComponentScan

包扫描器

3. 自定义Starter

3.0 Preparation

1. SpringBoot starter 机制

1. 一个可插拔的插件。
2. 在SpringBoot中导入了starter 即相当于导入了其背后的各种dependency。例如，想用Redis，就直接导入它相应的starter （spring-boot-starter-redis）即可。

2. 为什么要自定义Starter?

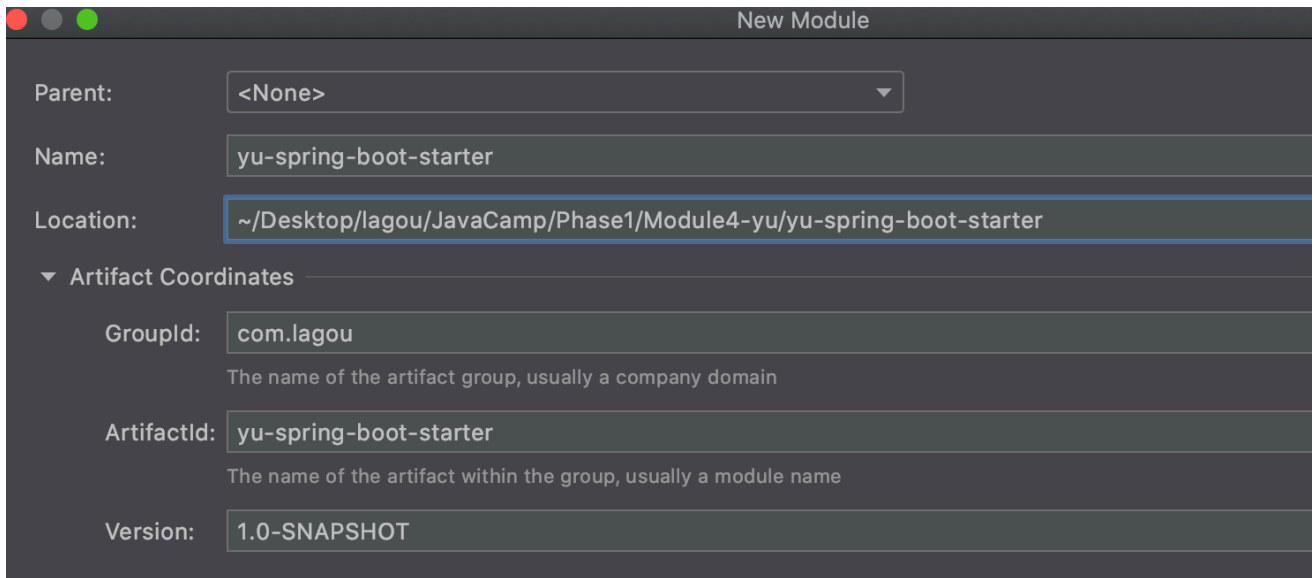
将独立于业务之外的配置模块封装成一个个starter，这样，复用的时候直接在pom中导入相应starter 即可，SpringBoot会完成自动装配。

3. 自定义starter的命名规则

1. 官方提供的starter：spring-boot-starter-xxx
2. 第三方自己写的starter：xxx-spring-boot-starter

3.1 手写自己的starter

Step 1. 新建Module，普通 maven 工程，Parent 为 none，名字为 yu-spring-boot-starter，注意 location:



New Module

Parent: <None>

Name: yu-spring-boot-starter

Location: ~/Desktop/lagou/JavaCamp/Phase1/Module4-yu/yu-spring-boot-starter

▼ Artifact Coordinates

GroupId: com.lagou
The name of the artifact group, usually a company domain

ArtifactId: yu-spring-boot-starter
The name of the artifact within the group, usually a module name

Version: 1.0-SNAPSHOT

Step 2. Add dependency in pom.xml

【注意】version不要用SNAPSHOT，要用RELEASE; 解决了maven compiler的报错，要写1.11 而不是11 【坑】

```
<!-- Refer to https://dev.to/techgirl1908/intellij-error-java-release-version-5-not-supported-376 -->
<properties>
    <maven.compiler.source>1.11</maven.compiler.source>
    <maven.compiler.target>1.11</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
autoconfigure</artifactId>
        <version>2.2.2.RELEASE</version>
    </dependency>
</dependencies>
```

Step 3. Create Java Bean

```

@EnableConfigurationProperties(SimpleBean.class) // 开启下面
@ConfigurationProperties 注解
@ConfigurationProperties(prefix = "simplebean") // 从全局配置文件中获
取"simplebean"开头的属性值，即配置文件转成bean对象
public class SimpleBean {

    private int id;
    private String name;
}

```

Step 4. Create MyAutoConfiguration class and add @Conditionalxxx

```

@Configuration
//@ConditionalOnClass(SimpleBean.class) // 标明 当classpath下存在 指定的类（这里是
SimpleBean类） 时，才会自动配置本配置类
@ConditionalOnClass // 此时不再有限制，本配置类一定会被SpringBoot自动配置
public class MyAutoConfiguration {
    // 加载本类的时候就会print
    static {
        System.out.println("MyAutoConfiguration init ...");
    }

    @Bean
    public SimpleBean simpleBean() {
        return new SimpleBean();
    }
}

```

Step 5. Create resources/META-INF/spring.factories

【注意】META-INF/spring.factories 是手动创建的folder和file，在file中注册上自己的自动配置类。

```

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.lagou.config.MyAutoConfiguration

```

3.2 使用自己的starter

Step 1. 在 `springbootdemo` 的pom文件中，导入自定义starter的dependency

```
<dependency>
  <groupId>com.lagou</groupId>
  <artifactId>yu-spring-boot-starter</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

Step 2. 在 `springbootdemo` 的 `application.properties` 中配置属性值

```
# Assign value to simplebean
simplebean.id=25
simplebean.name=Customized Starter
```

Step 3. Unit Test

```
@RunWith(SpringRunner.class) // 测试启动器, 并且 加载Spring boot测试注解
@SpringBootTest
public class MyCustomizedStarter {

    @Autowired
    private SimpleBean simpleBean;

    @Test
    public void yuStarterTest() {
        System.out.println(simpleBean);
    }
}
```

Result:

SimpleBean{id=25, name='Customized Starter'}

4. 执行原理

Main function 中的 run 方法是如何启动SpringBoot整个项目的?

SpringApplication的启动由两部分组成:

1. 实例化SpringApplication对象
2. run(args): 调用run方法


```
public static ConfigurableApplicationContext run(Class<?>[] primarySources,
String[] args) {
    return new SpringApplication(primarySources).run(args);
}
```

4.1 实例化SpringApplication

Step 1. 项目启动类（SpringbootDemoApplication.class）设置为属性存储起来

```
this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
```

Step 2. 从classpath上，判断 web 应用类型

```
//设置应用类型是SERVLET应用（Spring 5之前的传统MVC应用）还是REACTIVE应用（Spring 5开始出现的WebFlux交互式应用）
this.webApplicationType = WebApplicationType.deduceFromClasspath(); // return
Servlet 类型
```

deduceFromClasspath() 会判断是否是：Reactive 或者 Servlet 或是 None（即不存在Servlet类）。

Step 3. 通过ApplicationContextInitializer找到初始化器并使实例化

```
//所谓的初始化器就是org.springframework.context.ApplicationContextInitializer的实现类，
在Spring上下文被刷新之前进行初始化的操作
setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
```

在META-INF/spring.properties file中，找到ApplicationContextInitializer 以及其初始化器：

org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

Step 4. 通过ApplicationListener 找到监听器并实例化

在META-INF/spring.properties file中，找到ApplicationListener 以及其初始化器，这个监听器会贯穿SpringBoot整个生命周期。

Step 5. 初始化 mainApplicationClass 属性

用于推断并设置项目main() 方法启动的 主程序启动类

4.2 调用 .run(args) 方法

Step 1. 获取并启动 SpringApplicationRunListener

```
private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class };
    // 这里仍然利用了getSpringFactoriesInstances方法来获取Listener实例,
    // 从META-INF/spring.factories中读取Key为
    org.springframework.boot.SpringApplicationRunListener的Values
    return new SpringApplicationRunListeners(logger, getSpringFactoriesInstances(
        SpringApplicationRunListener.class, types, this, args));
}
```

Step 2. 根据SpringApplicationRunListener 以及参数来准备环境

```
// (2) 项目运行环境Environment的预配置
// 创建并配置当前SpringBoot应用将要使用的Environment
// 并遍历调用所有的SpringApplicationRunListener的environmentPrepared()方法
ConfigurableEnvironment environment = prepareEnvironment(listeners,
    applicationArguments);
```

Step 3. 创建Spring容器

1. 根据 webApplicationType 类型，获得 ApplicationContext 类型，
2. 根据webApplicationType进行判断创建容器的类型。该类型为SERVLET类型，所以会通过反射装载对应的字节码，也就是AnnotationConfigServletWebServerApplicationContext。
3. 使用之前初始化设置的context、environment、listeners、applicationArguments 和 printedBanner （在cosole里面打印图标）进行组装配置，并刷新配置。

Step 4. Spring容器前置处理

这一步主要是在容器刷新之前的准备动作，比如触发Listeners的响应事件、加载资源、设置context等。其中包含一个非常关键的操作：**将项目启动类注入容器**，为后续开启自动化配置奠定基础。

Step 5. 刷新容器

1. 通过 refresh 方法，对整个IoC容器进行初始化（包括bean的定位、解析、注册等）
2. 向JVM注册一个 ShutdownHook ，即若JVM被关闭了，context也就关闭了。

Step 6. Spring容器后置处理

1. 扩展接口，设计模式中的模板方法，默认为空实现。如果有自定义需求，可以重写该方法。比如打印一些启动结束log，或者一些其它后置处理。
2. 停止 stopWatch，统计时长
3. 打印 SpringBoot 启动的时长日志

Step 7. 发出结束执行的事件通知

执行所有SpringApplicationRunListener实现的started方法。

Step 8. 执行Runners

用于调用项目中自定义的执行器XxxRunner类，使得在项目启动完成后立即执行一些特定程序，比如一些业务初始化操作，这些操作只在服务启动后执行一次。

Spring Boot提供了ApplicationRunner和CommandLineRunner两种服务接口。若需使用，可自定义一个实现类 implements 其中一个接口并重写 run 方法即可。