<https://mp.weixin.qq.com/s/yjd9zsmlPPUMDpTzAlRMTw>

# SpringBoot：认认真真梳理一遍自动装配原理

## 前言

Spring翻译为中文是“春天”，的确，在某段时间内，它给Java开发人员带来过春天，但是随着我们项目规模的扩大，Spring需要配置的地方就越来越多，夸张点说，“配置两小时，Coding五分钟”。这种纷繁复杂的xml配置随着软件行业一步步地发展，必将逐步退出历史舞台。

## SpringBoot介绍

来自：百度百科

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

### SpringBoot所具备的特征有：

可以创建独立的Spring应用程序，并且基于其Maven或Gradle插件，可以创建可执行的JARs和WARs；

内嵌Tomcat或Jetty等Servlet容器；

提供自动配置的“starter”项目对象模型（POMS）以简化Maven配置；

尽可能自动配置Spring容器；

提供准备好的特性，如指标、健康检查和外部化配置；

绝对没有代码生成，不需要XML配置。

### 自己的理解：

SpringBoot，顾名思义，给人的感觉就是让Spring启动的这么一个项目。在过去，我们要让一个Spring项目启动，往往需要配置很多的xml配置文件，但是在使用SpringBoot之后，我们甚至无需写一行xml，就可以直接将整个项目启动，这种“零配置”的做法减轻了开发人员很多的工作量，可以让开发人员一心扑在业务逻辑的设计上，使项目的逻辑更加完善。

除此之外，其采用了JavaConfig的配置风格，导入组件的方式也由原来的直接配置改为@EnableXXXX，这种纯Java代码的配置和导入组件的方式，使代码看上去更加的优雅，所以SpringBoot如今受到大小公司和大多数程序员的青睐，不是没有原因的。

SpringBoot之所以可以做到简化配置文件直接启动，无外乎是其内部的两种设计策略：**开箱即用和约定大于配置**。

开箱即用：**在开发过程中，通过maven项目的pom文件中添加相关依赖包，然后通过相应的注解来代替繁琐的XML配置以管理对象的生命周期。**

约定大于配置：**由SpringBoot本身来配置目标结构，由开发者在结构中添加信息的软件设计范式。这一特点虽降低了部分灵活性，增加了BUG定位的复杂性，但减少了开发人员需要做出决定的数量，同时减少了大量的XML配置，并且可以将代码编译、测试和打包等工作自动化。**

那么在这篇博客中，我们需要了解的所有东西，就应该从这两个特点出发，一步一步深入SpringBoot自动装配的原理。

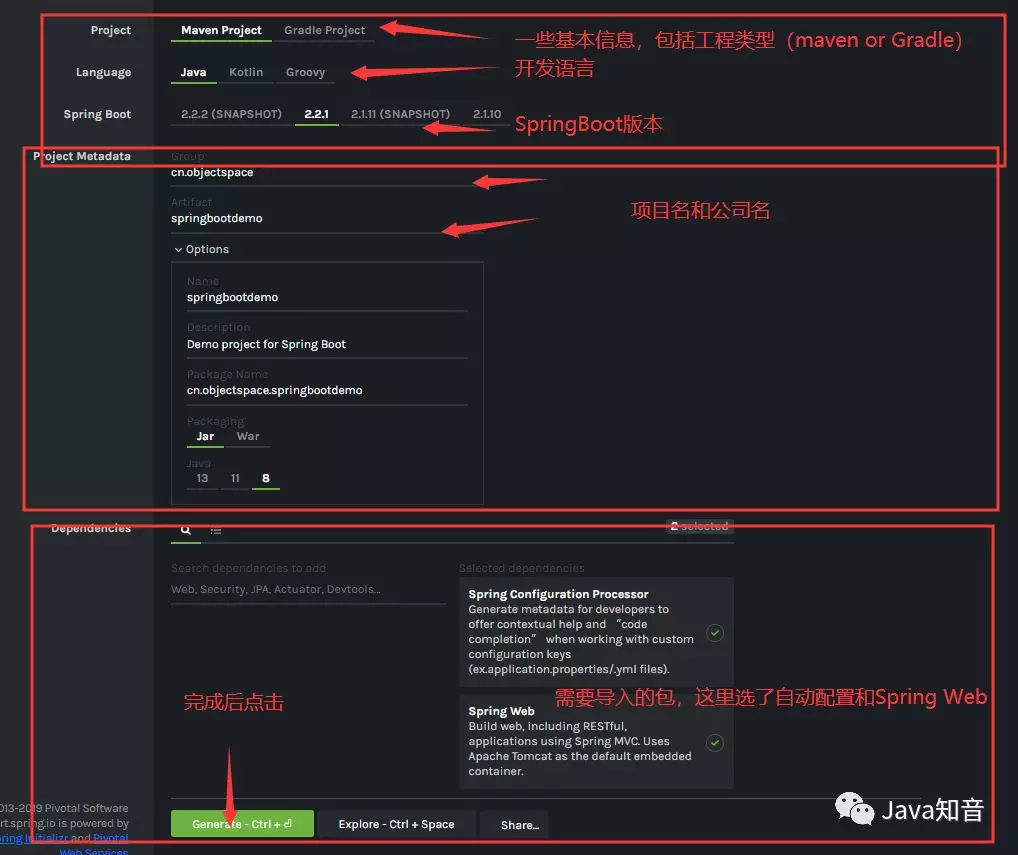
## 开箱即用原理

要理解这一特点，首先要先自己体会开箱即用的整个过程带来的便利。

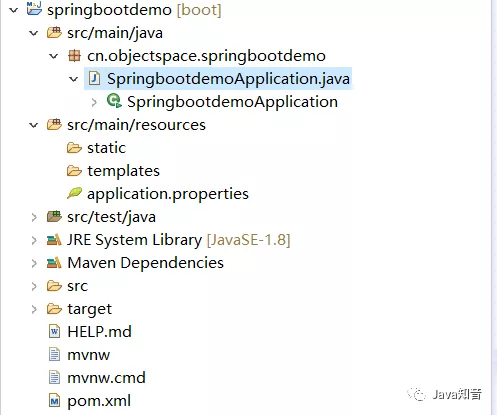
### 体验开箱即用

SpringBoot提供了我们快速创建SpringBoot项目的地方：https://start.spring.io/

我们只需要在这个网页中把整个项目起好名字，然后选好我们需要的组件，就可以直接获得一个可以跑起来的SpringBoot项目。



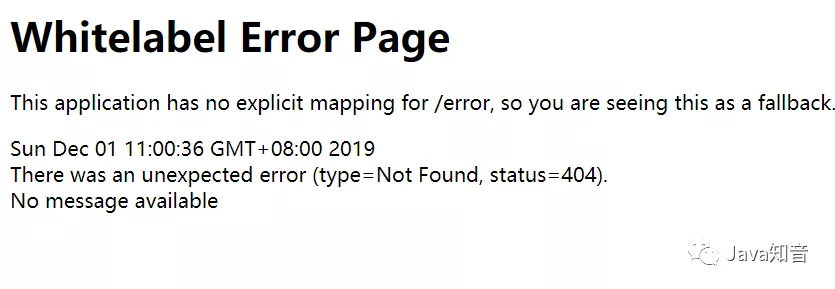
我们只需要填完上述信息，点击Generate，就可以直接将一个SpringBoot项目下载下来，然后导入我们的IDE，Eclipse或者IDEA都可，之后就可以直接将它运行起来。



启动：



访问：<http://localhost:8080/>



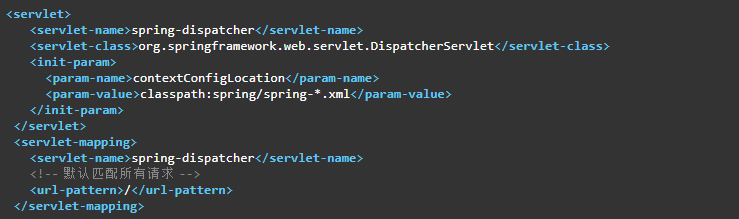
#### 对比SSM配置

其实在上文的开箱即用中，我们相当于引入了一个SpringMVC的组件，但是大家可以看到，我们没有经过任何的配置就将项目启动了。反观过去SSM框架的SpringMVC配置，我这里有一份留存的大家可以对比一下。

spring-web.xml:



web.xml:

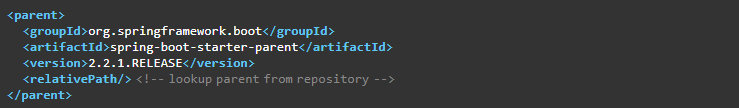


可以看到，这里需要配置两个文件，web.xml和spring-web.xml，配置可以说是相当繁重。

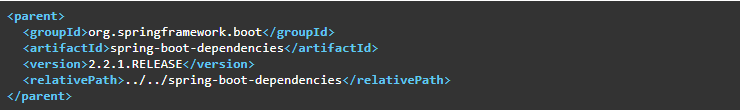
那么相对于这个，SpringBoot的开箱即用就显得特别方便，那么我们着重聊聊SpringBoot开箱即用的原理。

#### 从pom.xml开始

SpringBoot的项目都会存在一个父依赖，按住Ctrl+鼠标左键，可以点进去。

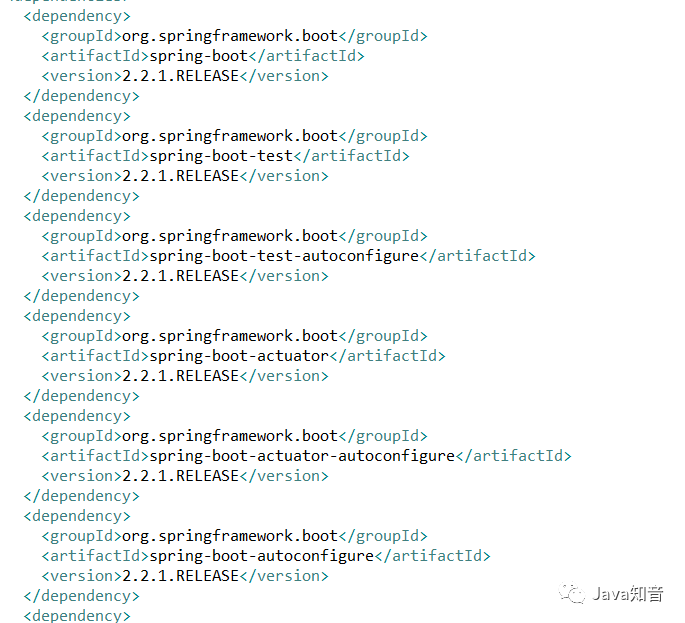


点进去之后发现里面除了一些插件和配置文件的格式之外，还存在一个依赖。



于是再点进去，可以发现里面放了很多的依赖和依赖的版本号。由于这个文件实在太长了，所以这里只展示一部分。

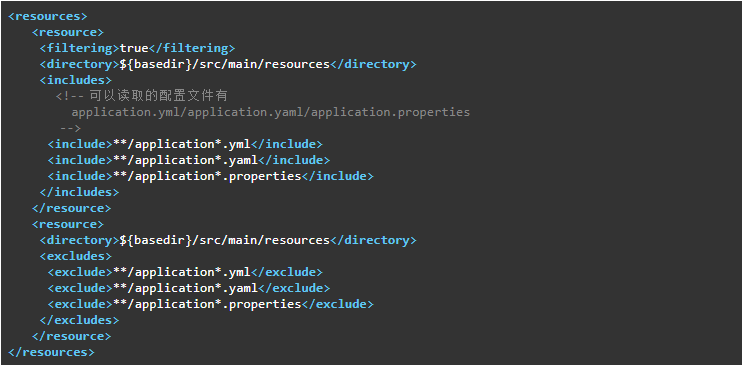




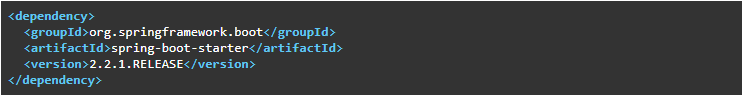
**所以我们可以得出第一个结论：**

spring-boot-dependencies:作为父工程，存放了SpringBoot的核心依赖。我们在写或者引入一些SpringBoot依赖的时候，不需要指定版本，正是因为SpringBoot的父依赖已经帮我们维护了一套版本。

另外我们还可以看到，在父依赖中也帮我们写好了资源库，不用我们自己再去配置了。



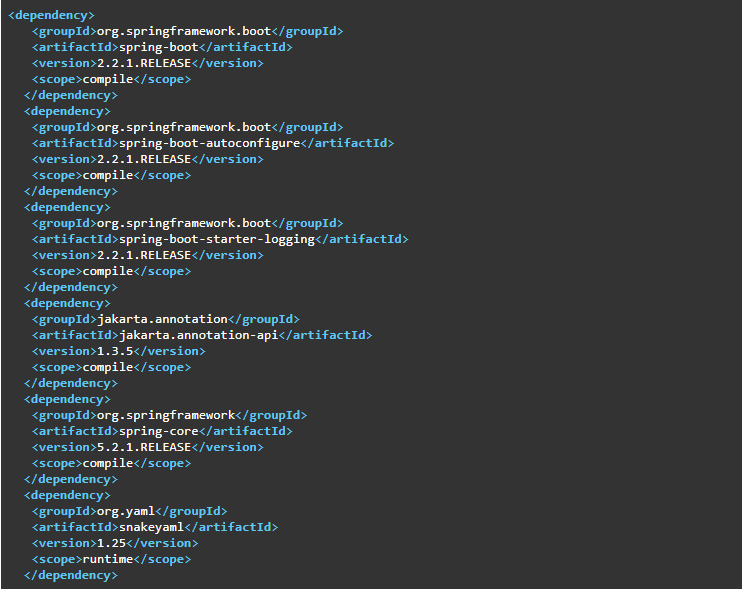
#### 启动器



启动器就是SpringBoot的启动场景，比如我们要使用web相关的，那么就直接引入spring-boor-starter-web，那么他就会帮我们自动导入web环境下所有必需的依赖。

我们来看看启动器中存放了一些什么内容：

以spring-boot-starter为例：

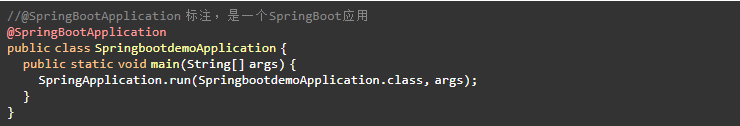


其中存放了自动配置相关的依赖、日志相关依赖、还有Spring-core等依赖，这些依赖我们只需要导入一个spring-boor-starter就可以直接将其全部引入，而不需要再像以前那样逐个导入了。

**SpringBoot会将所有的功能场景都封装成一个一个的启动器，供开发人员使用。**

我们在使用的时候也可以直接去官网上找我们所需的启动器，直接将其引入。

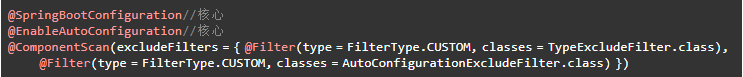
#### 主程序（重要）



再写SpringBoot项目的时候，总要写这么一个主程序，这个主程序最大的特点就是其类上放了一个@SpringBootApplication注解，这也正是SpringBoot项目启动的核心，也是我们要研究的重点。

**注意：之后的分析可能会深入源码，源码是一层一层嵌套的，所以光靠文字说明会比较难以理解，最好是自己在IDE环境下跟着一步一步跟着点下去。当然也可以绕过这一部分直接看结论。**

点开@SpringBootApplication，可以发现它是一个组合注解，主要是由这么几个注解构成的。



我们首先要研究的就是核心的两个注解 **@SpringBootConfiguration**和**@EnableAutoConfiguration**，逐个进行分析。

@SpringBootConfiguration



可以看到SpringBootConfiguration其实就携带了一个@Configuration注解，这个注解我们再熟悉不过了，他就代表自己是一个Spring的配置类。所以我们可以认为：@SpringBootConfiguration = @Configuration

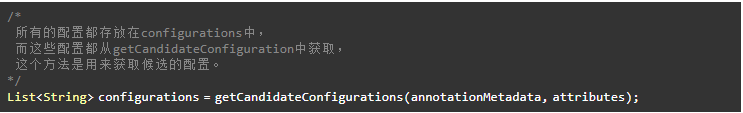
@EnableAutoConfiguration

顾名思义，这个注解一定和自动配置相关，点进去看源代码之后可以发现，其内部就包含了这么两个注解。顾名思义，这个注解一定和自动配置相关，点进去看源代码之后可以发现，其内部就包含了这么两个注解。



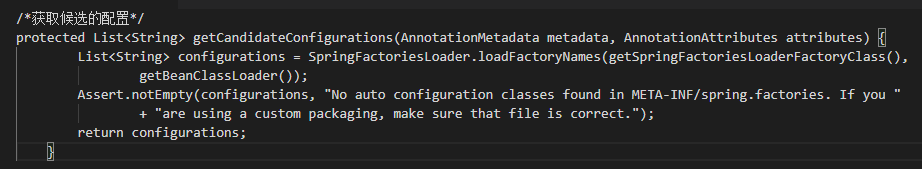
来看看@Import(AutoConfigurationImportSelector.class)中的内容：

它帮我们导入了AutoConfigurationImportSelector，这个类中存在一个方法可以帮我们获取所有的配置，代码如下。



**getCandidateConfigurations():**

这个方法可以用来获取所有候选的配置，那么这些候选的配置又是从哪来的呢？



实际上它返回了一个List，这个List是由**loadFactoryNames()**方法返回的,其中传入了一个getSpringFactoriesLoaderFactoryClass()，我们可以看看这个方法的内容。

我们看到了一个眼熟的词 —— **EnableAutoConfiguration**，也就是说，它实际上返回的就是标注了这个类的所有包。标注了这个类的包不就是@SpringBootApplication吗？

所以我们可以得出结论：它兜兜转转绕了这么多地方，就是为了将启动类所需的所有资源导入。

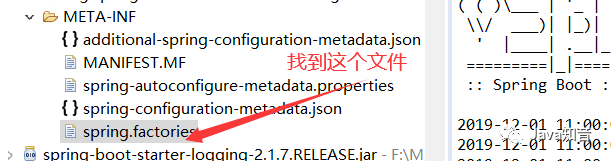
我们接着往下看，它其中还有这么一条语句，是一条断言：

Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you "

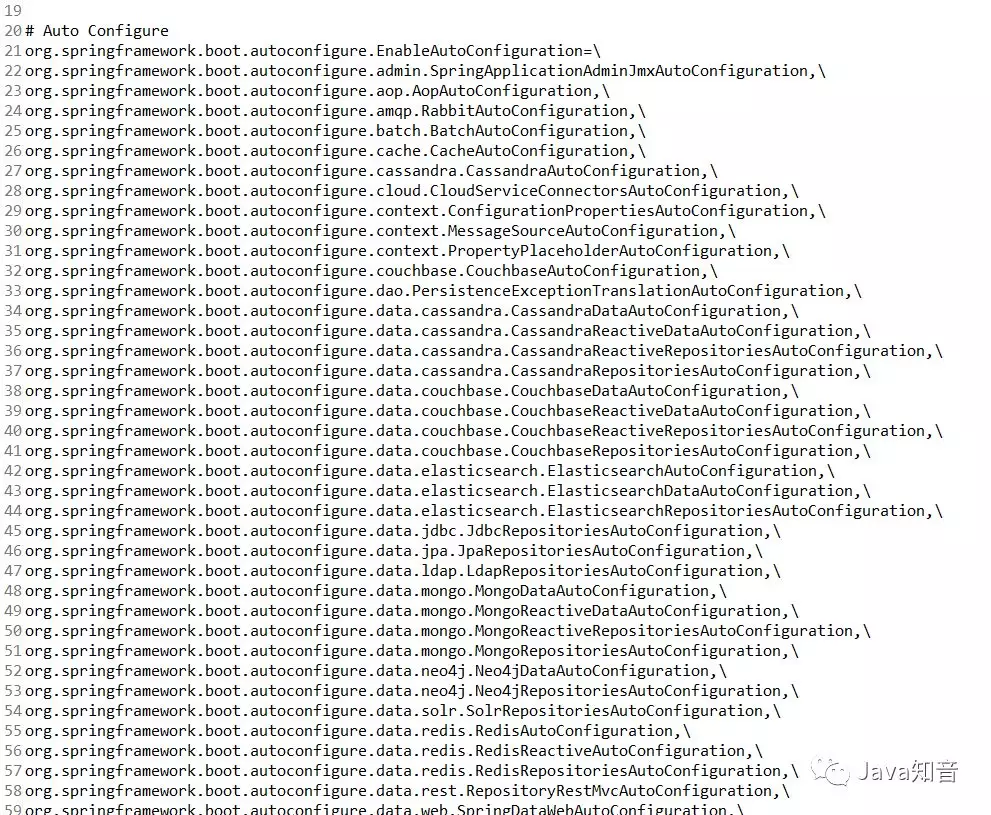
+ "are using a custom packaging, make sure that file is correct.");

这个断言的意思是，configurations必须非空，否则就打印一段话，No auto configuration classes found in META-INF/spring.factories，我们把这个逻辑反过来想想。如果这个集合不为空，是不是就代表找到了这个spring.factories并且会去加载这个文件中的内容呢？

带着这个疑问，我们首先找到spring.factories这个文件：



可以看到里面包含了很多自动配置属性：



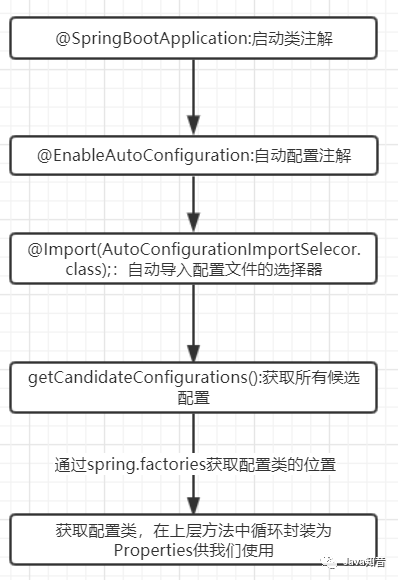
我们可以随便找一个自动配置点进去,比如WebMvcAutoConfiguration：



这里放了所有关于WebMvc的配置，如视图解析器、国际化等等。

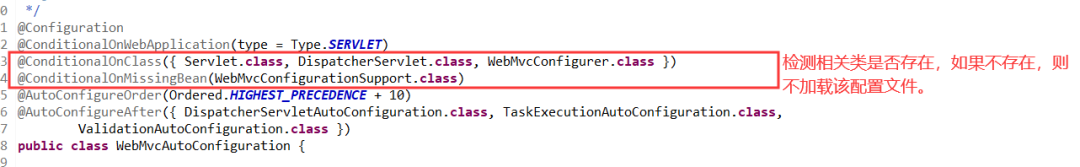
分析到这里，我们就可以得出一个完整的结论了：

**当我们的SpringBoot项目启动的时候，会先导入AutoConfigurationImportSelector，这个类会帮我们选择所有候选的配置，我们需要导入的配置都是SpringBoot帮我们写好的一个一个的配置类，那么这些配置类的位置，存在与META-INF/spring.factories文件中，通过这个文件，Spring可以找到这些配置类的位置，于是去加载其中的配置。**



看到这里，可能有些同学会存在疑问，spring.factories中存在那么多的配置，每次启动时都是把它们全量加载吗？这显然是不现实的。

这其实也是我在看源码的时候存在疑问的地方，因为其中有一个注解并不常用，我们点开一个配置类就可以看到。



@ConditionalOnXXX:如果其中的条件都满足，该类才会生效。

所以在加载自动配置类的时候，并不是将spring.factories的配置全量加载进来，而是通过这个注解的判断，如果注解中的类都存在，才会进行加载。

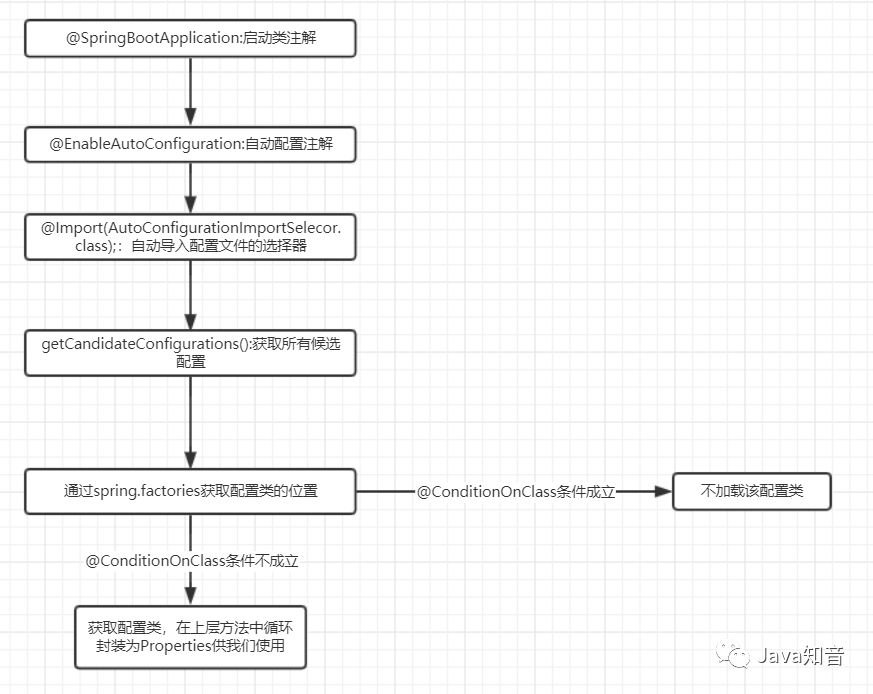
所以就实现了：我们在pom.xml文件中加入stater启动器，SpringBoot自动进行配置。完成开箱即用。

#### 结论

SpringBoot所有自动配置类都是在启动的时候进行扫描并加载，通过spring.factories可以找到自动配置类的路径，但是不是所有存在于spring,factories中的配置都进行加载，而是通过@ConditionalOnClass注解进行判断条件是否成立（只要导入相应的stater，条件就能成立），如果条件成立则加载配置类，否则不加载该配置类。

在这里贴一个我认为的比较容易理解的过程：

* 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
* 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；
* 以前我们需要自己配置的东西 ， 自动配置类都帮我们解决了
* 整个J2EE的整体解决方案和自动配置都在springboot-autoconfigure的jar包中；
* 它将所有需要导入的组件以全类名的方式返回 ， 这些组件就会被添加到容器中 ；
* 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 ；
* 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；



## 约定大于配置

开箱即用的原理说完了，约定大于配置就比较好理解了。其实约定大于配置就是开箱即用中那些自动配置的细节。说的具体点就是：**我们的配置文件（.yml）应该放在哪个目录下，配置文件的命名规范，项目启动时扫描的Bean，组件的默认配置是什么样的（比如SpringMVC的视图解析器）等等等等这一系列的东西，都可以被称为约定**，下面就来一点一点地说一下SpringBoot中的“约定”。

### maven目录结构的约定

我们可以去Spring的官网查看一下官方文档，看看文档中描述的目录结构是怎样的。

Config locations are searched in reverse order. By default, the configured locations are classpath:/,classpath:/config/,file:./,file:./config/. The resulting search order is the following:

* file:./config/
* file:./
* classpath:/config/
* classpath:/

也就是说，spring的配置文件目录可以放在

* /config
* /(根目录)
* resource/config/
* resource/

这四个路径从上到下存在优先级关系。

### SpringBoot默认配置文件的约定

SpringBoot默认可以加载以下三种配置文件：

* application.yml
* application.yaml
* application.properties

建议使用前两种作为项目的配置文件。

### 项目启动时扫描包范围的约定

SpringBoot的注解扫描的默认规则是SpringBoot的入口类所在包及其子包。

若入口类所在的包是cn.objectspace.demo那么自动扫描包的范围是cn.objectspace.demo包及其下面的子包，如果service包和dao包不在此范围，则不会自动扫描。

## SpringBoot自动配置类如何读取yml配置

### 从更细节的角度去理解自动配置

上文中我们阐述了一些SpringBoot自动配置的原理，我们是从全局的角度去看自动配置的整个过程。比如从哪个地方开始进行装配流程、如何找到装配的包等。

那么现在将自己的视角贴近SpringBoot，来聊聊application.yml中我们配置的东西，是如何配置到一个个的配置类中的。

### yml配置文件中可以配置那些东西

首先要知道这个问题的答案，我们应该习惯springboot的配置方式。在上文中我们阐述了SpringBoot总是将所有的配置都用JavaConfig的形式去呈现出来，这样能够使代码更加优雅。

那么yml中配置的东西，必然是要和这种配置模式去进行联系的，我们在application.yml中配置的东西，通常是一些存在与自动配置类中的属性，那么这些自动配置类，在启动的时候是怎么找到的呢？

如果你还记得上文的描述，那么你可以很明确地知道：spring.factories！没错，就是它，所以这个问题我们似乎得到了答案——只要存在于spring.factories中的，我们都可以在application.yml中进行配置。

当然，这并不意味着不存在其中的我们就不能配置，这些配置类我们是可以进行自定义的，只要我们写了配置类，我们就可以在yml中配置我们需要的属性值，然后在配置类中直接读取这个配置文件，将其映射到配置类的属性上。那么就牵扯出我们的问题了：配置类是如何去读取yml配置文件中的信息的呢？

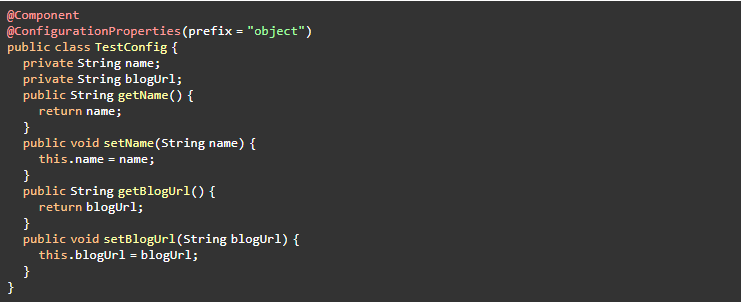
### @ConfigurationProperties

要明白这个问题。我们就首先要去了解这个注解有什么作用。

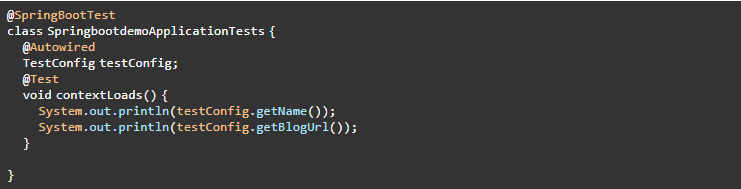
我们可以自己尝试在application.yml中去定义一些属性，如下：

object:   
  name: Object  
  blogurl: blog.objectspace.cn

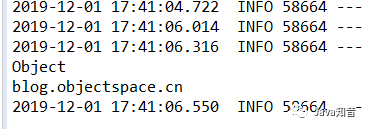
我们现在自己定义一个类去读取这个文件：



然后我们在测试类中输出一下这个对象：



测试结果：

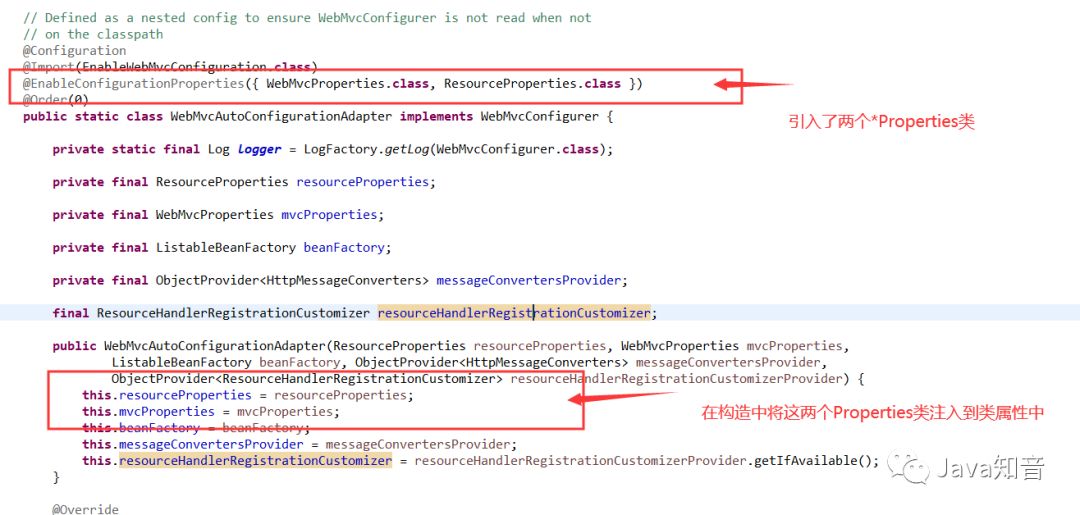


我们可以看到，在控制台中输出了我们在yml中配置的属性值，但是这些值我们没有在任何地方显式地对这个对象进行注入。

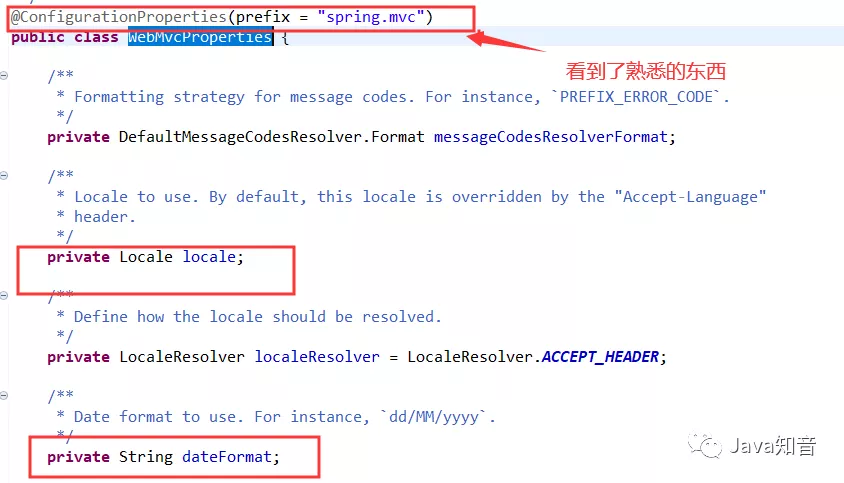
所以@ConfigurationProperties这个注解，可以将yml文件中写好的值注入到我们类的属性中。

**明白了它的作用，就能明白自动配置类工作的原理了。**

我们依旧是选取SpringMVC的自动配置类，我们来看看其中有些什么东西。



点击任意一个\*Properties类中，look一下其中的内容：



看到这里相信所有人都明白了，我们就拿mvc配置来举例。



我们在yml中配置的date-format,就可以通过@ConfigurationProperties映射到类中的dateFormat中,然后在通过自动配置类，将这些属性配置到配置类中。