面试高频题：springBoot自动装配的原理你能说出来吗?

程序员面试 5天前

以下文章来源于java金融，作者码农二胖

引言

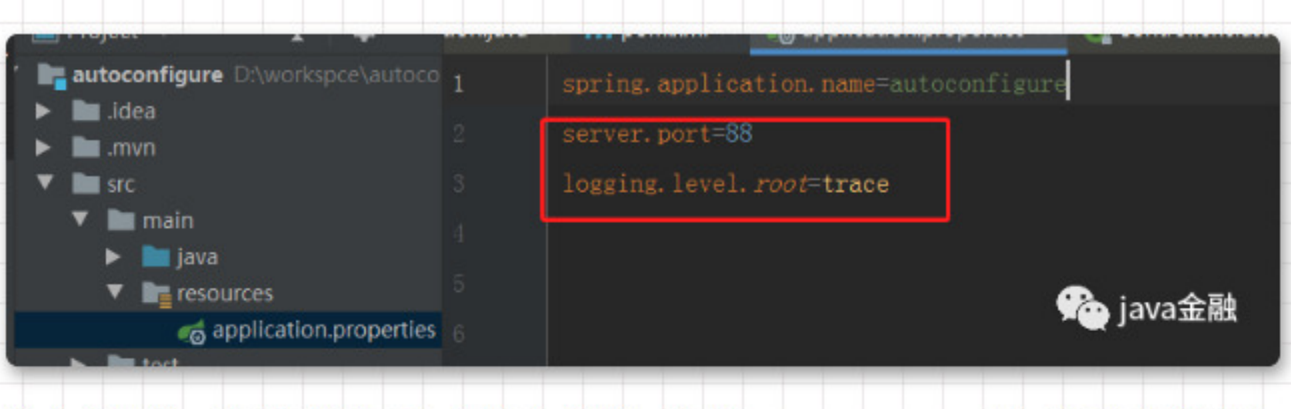
最近有个读者在面试，面试中被问到了这样一个问题“看你项目中用到了 `springboot`，你说下 `springboot` 的自动配置是怎么实现的？”这应该是一个 `springboot` 里面最最常见的一个面试题了。下面我们就来带着这个问题一起解剖下 `springBoot` 的自动配置原理吧。

SpringMvc和SpringBoot对比

首先我们回顾下原来搭建一个 `springmvc` 的 `hello-word` 的 `web` 项目（ `xml` 配置的）我们是不是要在 `pom` 中导入各种依赖，然后各个依赖有可能还会存在版本冲突需要各种排除。当你历尽千辛万苦的把依赖解决了，然后还需要编写 `web.xml、springmvc.xml` 配置文件等。我们只想写 `个hello-word` 项目而已，却把一大把的时间都花在了配置文件和 `jar` 包的依赖上面。大大的影响了我们开发的效率，以及加大了 `web` 开发的难度。为了简化这复杂的配置、以及各个版本的冲突依赖关系，`springBoot` 就应运而生。我们现在通过 `idea` 创建一个 `springboot` 项目只要分分钟就解决了，你不需要关心各种配置（基本实现零配置），让你真正的实现了开箱即用。`SpringBoot` 帮你节约了大量的时间去陪女朋友，不对程序员怎么会有女朋友呢？（**「没有的话也是可以new一个的」**）它的出现不仅可以让你把更多的时间都花在你的业务逻辑开发上，而且还大大的降低了 `web` 开发的门槛。所以 `SpringBoot` 还是比较善解人衣的,错啦错啦是善解人意，知道开发人员的痛点在哪。

SpringBoot自动配置加载

既然 `Springboot` 尽管这么好用，但是作为一个使用者，我们还是比较好奇它是怎么帮我们实现开箱即用的。`Spring Boot` 有一个全局配置文件：`application.properties或application.yml`。在这个全局文件里面可以配置各种各样的参数比如你想改个端口啦 `server.port` 或者想调整下日志的级别啦通通都可以配置。更多其他可以配置的属性可以参照官网。https://docs.spring.io/spring-boot/docs/2.3.0.RELEASE/reference/htmlsingle/#common-application-properties



```
spring.application.name=autoconfigure
server.port=88
logging.level.root=trace
```

这么多属性，这些属性在项目是怎么起作用的呢？ `SpringBoot` 项目看下来啥配置也没有，配置”（ `application.properties或application.yml` 除外），既然从配置上面找不到突破口，那么我们就只能从启动类上面找入口了。启动类也就一个光秃秃的一个 `main` 方法，类上面仅有一个注 `SpringBootApplication` 这个注解是 `Spring Boot` 项目必不可少的注解。那么自动配置原理一定和这个注解有着千丝万缕的联系！我们下面来一起看看这个注解吧。**「@SpringBootApplication注解」**

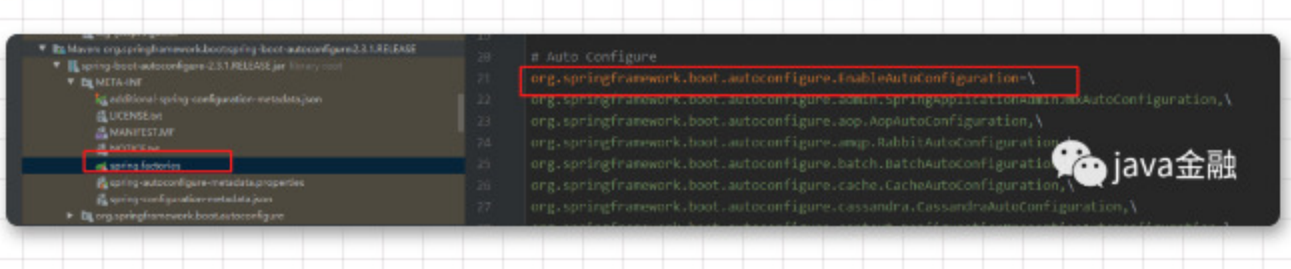
```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilte
  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

这里最上面四个注解的话没啥好说的，基本上自己实现过自定义注解的话，都知道分别是什么意思。

- `@SpringBootConfiguration` 继承自 `@Configuration`，二者功能也一致，标注当前类是配置类。
- `@ComponentScan` 用于类或接口上主要是指定扫描路径，跟Xml里面的 `<context:component-scan base-package="" />` 配置一样。`springboot` 如果不写这个扫描路径的话，默认就是启动类的路径。
- `@EnableAutoConfiguration`

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

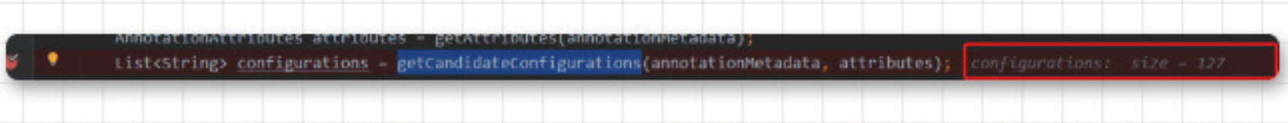
这个注解我们重点看下 `AutoConfigurationImportSelector` 这个类 `getCandidateConfigurations` 这个方法里面通过 `SpringFactoriesLoader.loadFactoryNames()` 扫描所有具有 `META-INF/spring.factories` 的 `jar` 包（ spring.factories 我们可以理解成 `Spring Boot` 自己的 `SPI` 机制）。`spring-boot-autoconfigure-x.x.x.x.jar` 里就有一个spring.factories文件。`spring.factories` 文件由一组一组的 `Key = value` 的形式，其中一个 `key` 是EnableAutoConfiguration类的全类名，而它的value是一个以 `AutoConfiguration` 结尾的类名的列表，有 `redis、mq` 等这些类名以逗号分隔。



```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
```

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotation
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);  attributes:
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attribu
    configurations = removeDuplicates(configurations);  configurations:  size = 127
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
AutoConfigurationImportSelector > getAutoConfigurationEntry()
```

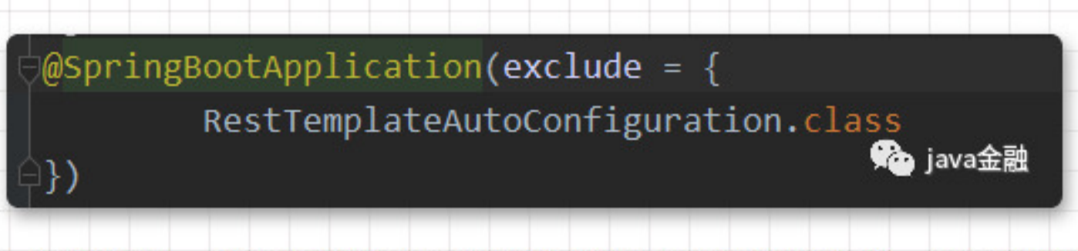
我们在回到 `getAutoConfigurationEntry` 这个方法当执行完 `getCandidateConfigurations` 这个方法的时候我们可以看到此时总共加载了 `127` 个自动配置类。



```
List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);  configurations:  size = 127
```

这些类难道都要加载进去吗？ `springboot` 还是没有那么傻的，它提倡的话是按需加载。

- 它会去掉重复的类
- 过滤掉我们配置了 `exclude` 注解的类下面配置就会过滤掉 `RestTemplateAutoConfiguration` 这个类

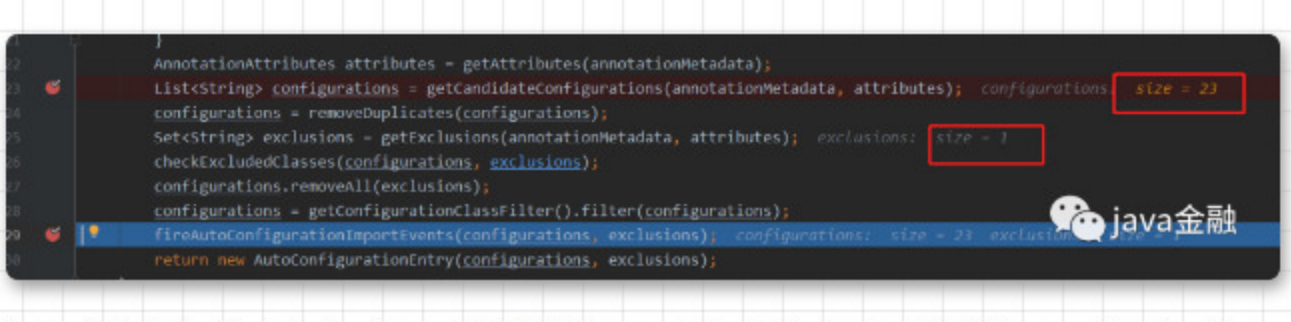


```
@SpringBootApplication(exclude = {
        RestTemplateAutoConfiguration.class
})
```

- 经过上面的处理，剩下的这些自动配置的类如果要起作用的话，是需要满足一定的条件的。这些条件的满足的话 `spring boot` 是通过条件注解来实现的。

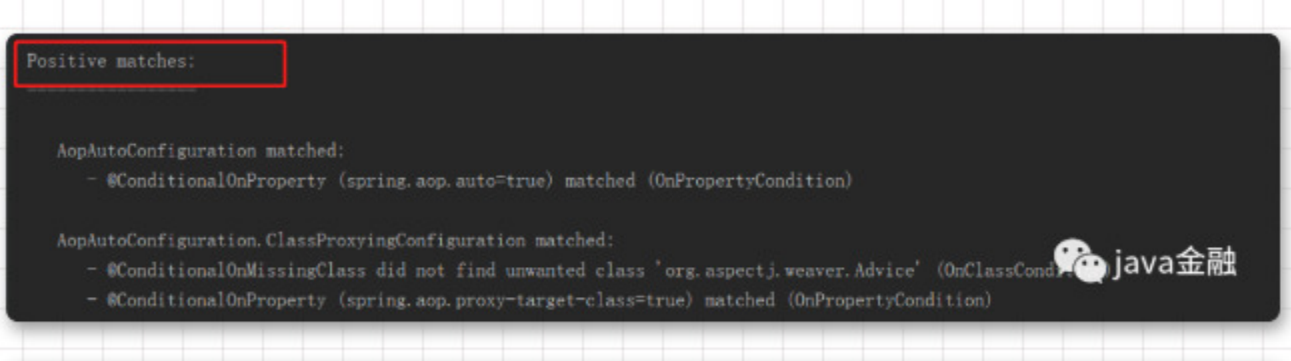
> @ConditionalOnBean：当容器里有指定Bean的条件下
> @ConditionalOnClass：当类路径下有指定的类的条件下
> @ConditionalOnExpression：基于SpEL表达式为true的时候作为判断条件才去实例化
>
> @ConditionalOnJava：基于JVM版本作为判断条件 @ConditionalOnJndi：在JNDI存在的条件下查找指定的位置 @ConditionalOnMissingBean：当容器里没有指定Bean的情况下
>
> @ConditionalOnMissingClass：当容器里没有指定类的情况下
>
> @ConditionalOnWebApplication：当前项目是Web项目的条件下
>
> @ConditionalOnNotWebApplication：当前项目不是Web项目的条件下
>
> @ConditionalOnProperty：指定的属性是否有指定的值
> @ConditionalOnResource：类路径是否有指定的值
> @ConditionalOnOnSingleCandidate：当指定Bean在容器中只有一个，或者有多个但是指定首选的Bean

这些注解都组合了 `@Conditional` 注解，只是使用了不同的条件组合最后为true时才会去实例化需要实例化的类，否则忽略过滤掉。我们在回到代码可以看到经过了条件判断过滤后我们剩下符合条件的自动配置类只剩23个了。其他的都是因为不满足条件注解而被过滤了。



```
AnnotationAttributes attributes = getAttributes(annotationMetadata);
List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);  configurations  size = 23
configurations = removeDuplicates(configurations);
Set<String> exclusions = getExclusions(annotationMetadata, attributes);  exclusions:  size = 1
checkExcludedClasses(configurations, exclusions);
configurations.removeAll(exclusions);
configurations = getConfigurationClassFilter().filter(configurations);
fireAutoConfigurationImportEvents(configurations, exclusions);  configurations:  size = 23  exclusion
return new AutoConfigurationEntry(configurations, exclusions);
```

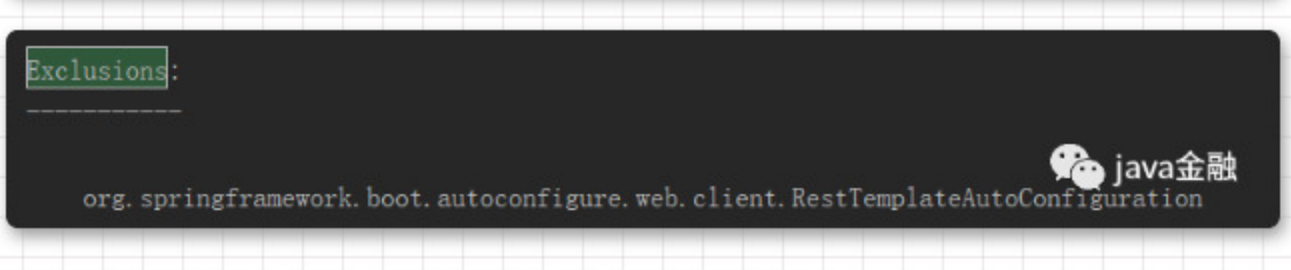
如果我们想知道哪些自动配置类被过滤了，是由于什么原因被过滤了，以及加载了哪些类等。 `spring boot` 都为我们记录了日志。还是非常贴心的。我们可以调整下我们日志的级别改为 `debug`。然后我们就能看到以下日志了



```
Positive matches:
-----------------

   AopAutoConfiguration matched:
      - @ConditionalOnProperty (spring.aop.auto=true) matched (OnPropertyCondition)

   AopAutoConfiguration.ClassProxyingConfiguration matched:
      - @ConditionalOnMissingClass did not find unwanted class 'org.aspectj.weaver.Advice' (OnClassCond
      - @ConditionalOnProperty (spring.aop.proxy-target-class=true) matched (OnPropertyCondition)
```



```
Exclusions:
-----------

    org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration
```

这里就截取了部分日志。总共分别有下面四部分日志：

- `Positive matches`：`@Conditional` 条件为真，配置类被Spring容器加载。
- `Negative matches:` `@Conditional` 条件为假，配置类未被Spring容器加载。
- `Exclusions`：我们明确了不需要加载的类。比如在上面启动类配置的 `RestTemplateAutoConfiguration` 类
- `Unconditional classes`：自动配置类不包含任何类级别的条件，也就是说，类始终会被自动加载。

自动配置生效

我们以 `ServletWebServerFactoryAutoConfiguration` 配置类为例，解释一下全局配置文件中的属性如何生效，比如：`server.port=88`，是如何生效的（当然不配置也会有默认值，这个默认值来自于 `org.apache.catalina.startup.Tomcat`）。

```
// 标记为配置类
@Configuration(proxyBeanMethods = false)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
// 如果有ServletRequest.class 才会生效
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
// 把@ConfigurationProperties注解的类注入为Spring容器的Bean。
@EnableConfigurationProperties(ServerProperties.class)
@Import({ ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
  ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
  ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
  ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {
```

我们可以发现 `EnableConfigurationProperties` 注解里面配置的 `ServerProperties.class`

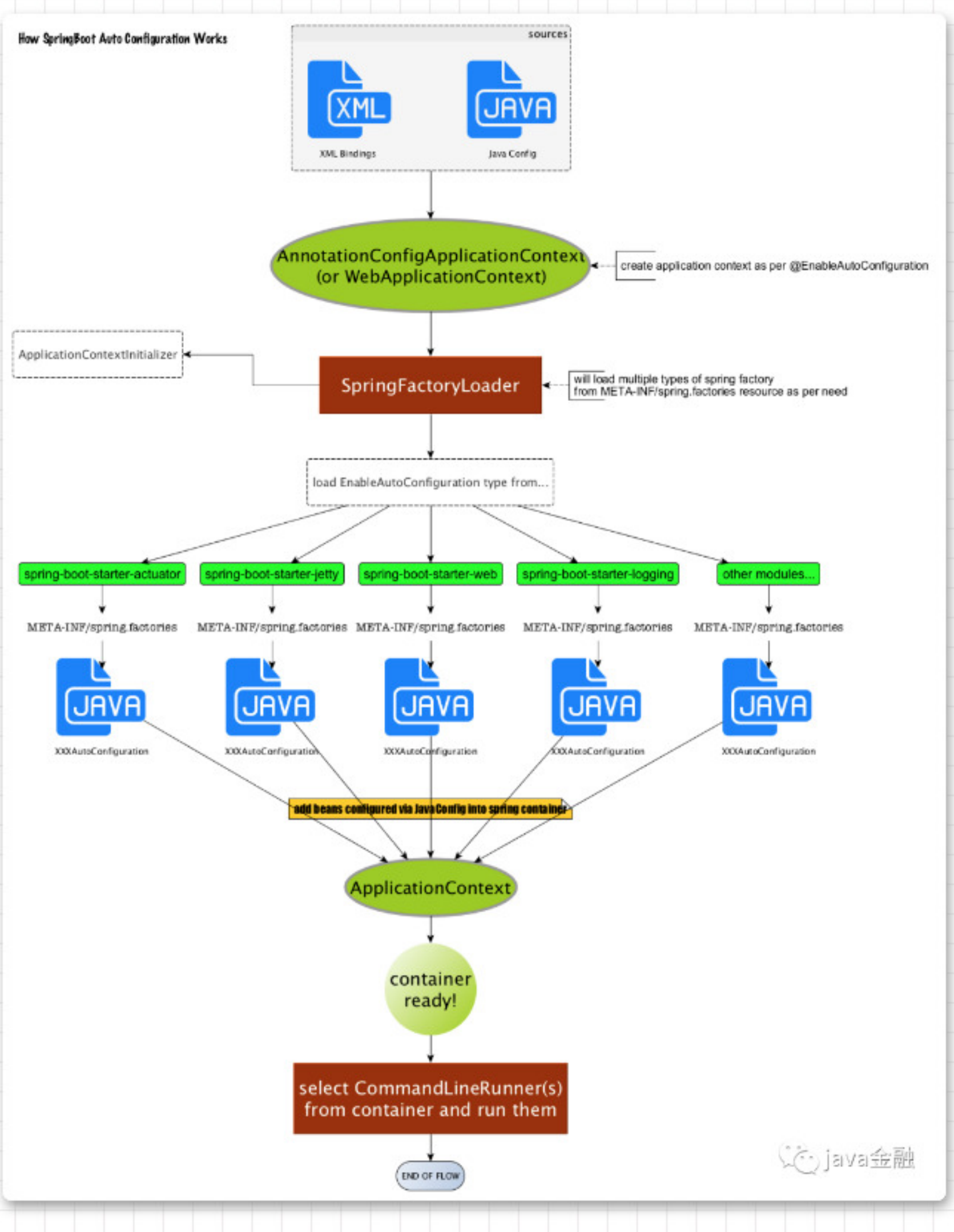
```
@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
public class ServerProperties {

 /**
  * Server HTTP port.
  */
 private Integer port;
```

在这个类上有一个注解：`@ConfigurationProperties`，它的作用就是从配置文件中绑定属性到对应的bean上(也就是把我们 `application.properties` 对应的server.port映射到 `ServerProperties` 类中的 `port` 属性)而@ `EnableConfigurationProperties` 这个注解就是把已经绑定了属性的 `bean`（ `ServerProperties`）注入到 `spring` 容器中（相当于@ `Component` 注解一样）。所有在配置文件中能配置的属性都是在 `xxxxPropertites` 类中封装着，配置文件能配置什么就可以参照某个功能对应的这个属性类。到现在为止应该能回答文章开头的那个问题了，面试的时候应该不需要回答的这么详细可以参考下以下答案：

> Spring Boot启动的时候会通过@EnableAutoConfiguration注解找到META-INF/spring.factories配置文件中的所有自动配置类，并对其进行加载，而这些自动配置类都是以AutoConfiguration结尾来命名的，它实际上就是一个JavaConfig形式的Spring容器配置类，它能通过以Properties结尾命名的类中取得在全局配置文件中配置的属性如：server.port，而XxxxProperties类是通过@ConfigurationProperties注解与全局配置文件中对应的属性进行绑定的。

在网上找了一张图，基本上把自动装配的流程给说清楚了。图片地址



总结

- `SpringBoot` 启动会加载大量的自动配置类（通过“ `SPI` ”的方式），然后会根据条件注解保留一些需要的类。
- 我们新引入一个组件，可以先看看springBoot是否已经有默认的提供。
- `SpringBoot` 基本实现了“零配置”，并且开箱即用。
- 站在巨人的肩膀上摘苹果：

  https://docs.spring.io/spring-boot/docs/2.3.0.RELEASE/reference/htmlsingle https://blog.csdn.net/u014745069/article/details/83820511

  https://afoo.me/posts/2015-07-09-how-spring-boot-works.html