

SpringBoot简介

回顾：什么是Spring

Spring是一个开源框架，2003 年兴起的一个轻量级的Java 开发框架，作者：Rod Johnson 。

Spring是为了解决企业级应用开发的复杂性而创建的，简化开发。

Spring是如何简化Java开发的

为了降低Java开发的复杂性，Spring采用了以下4种关键策略：

- 1、基于POJO的轻量级和最小侵入性编程，所有东西都是bean；
- 2、通过IOC，依赖注入（DI）和面向接口实现松耦合；
- 3、基于切面（AOP）和惯例进行声明式编程；
- 4、通过切面和模版减少样式代码，RedisTemplate，xxxTemplate；

什么是SpringBoot

学过javaweb的同学就知道，开发一个web应用，从最初开始接触Servlet结合Tomcat, 跑出一个Hello World程序，是要经历特别多的步骤；后来就用了框架Struts，再后来是SpringMVC，到了现在的SpringBoot，过一两年又会有其他web框架出现；你们有经历过框架不断的演进，然后自己开发项目所有的技术也再不断的变化、改造吗？建议都可以去经历一遍；

言归正传，什么是SpringBoot呢，就是一个javaweb的开发框架，和SpringMVC类似，对比其他javaweb框架的好处，官方说是简化开发，约定大于配置， you can "just run", 能迅速的开发web应用，几行代码开发一个http接口。

所有的技术框架的发展似乎都遵循了一条主线规律：从一个复杂应用场景 衍生 一种规范框架，人们只需要进行各种配置而不需要自己去实现它，这时候强大的配置功能成了优点；发展到一定程度之后，人们根据实际生产应用情况，选取其中实用功能和设计精华，重构出一些轻量级的框架；之后为了提高开发效率，嫌弃原先的各类配置过于麻烦，于是开始提倡“约定大于配置”，进而衍生出一些一站式的解决方案。

是的这就是Java企业级应用->J2EE->spring->springboot的过程。

随着 Spring 不断的发展，涉及的领域越来越多，项目整合开发需要配合各种各样的文件，慢慢变得不那么易用简单，违背了最初的理念，甚至人称配置地狱。Spring Boot 正是在这样的背景下被抽象出来的开发框架，目的为了让家更容易的使用 Spring 、更容易的集成各种常用的中间件、开源软件；

Spring Boot 基于 Spring 开发，Spring Boot 本身并不提供 Spring 框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于 Spring 框架的应用程序。也就是说，它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。Spring Boot 以**约定大于配置的核心思想**，默认帮我们进行了很多设置，多数 Spring Boot 应用只需要很少的 Spring 配置。同时它集成了大量常用的第三方库配置（例如 Redis、MongoDB、Jpa、RabbitMQ、Quartz 等等），Spring Boot 应用中这些第三方库几乎可以零配置的开箱即用。

简单来说就是SpringBoot其实不是什么新的框架，它默认配置了很多框架的使用方式，就像maven整合了所有的jar包，spring boot整合了所有的框架。

Spring Boot 出生名门，从一开始就站在一个比较高的起点，又经过这几年的发展，生态足够完善，Spring Boot 已经当之无愧成为 Java 领域最热门的技术。

Spring Boot的主要优点:

- 为所有Spring开发者更快的入门
- **开箱即用**，提供各种默认配置来简化项目配置
- 内嵌式容器简化Web项目
- 没有冗余代码生成和XML配置的要求

使用 Spring Boot 到底有多爽，用下面这幅图来表达



HelloWorld

准备工作

我们将学习如何快速的创建一个Spring Boot应用，并且实现一个简单的Http请求处理。通过这个例子对Spring Boot有一个初步的了解，并体验其结构简单、开发快速的特性。

我的环境准备：

- java version "1.8.0_181"
- Maven-3.6.1
- SpringBoot 2.x 最新版

开发工具：

- IDEA

创建基础项目说明

Spring官方提供了非常方便的工具让我们快速构建应用，Spring Initializr: <https://start.spring.io/>

项目创建方式一：使用Spring Initializr 的 Web页面创建项目

- 1、打开 <https://start.spring.io/>
- 2、填写项目信息
- 3、点击“Generate Project”按钮生成项目；下载此项目
- 4、解压项目包，并用IDEA以Maven项目导入，一路下一步即可，直到项目导入完毕。

5、如果是第一次使用，可能速度会比较慢，包比较多、需要耐心等待一切就绪。

项目创建方式二：使用 IDEA 直接创建项目

- 1、创建一个新项目
- 2、选择spring initializr， 可以看到默认就是去官网的快速构建工具那里实现
- 3、填写项目信息
- 4、选择初始化的组件（初学勾选 Web 即可）
- 5、填写项目路径
- 6、等待项目构建成功

项目结构分析：

通过上面步骤完成了基础项目的创建。就会自动生成以下文件。

- 1、程序的主启动类
- 2、一个 application.properties 配置文件
- 3、一个 测试类
- 4、一个 pom.xml

pom.xml 分析

打开 `pom.xml`，看看Spring Boot项目的依赖：

```
1  <!-- 父依赖 -->
2  <parent>
3      <groupId>org.springframework.boot</groupId>
4      <artifactId>spring-boot-starter-parent</artifactId>
5      <version>2.2.5.RELEASE</version>
6      <relativePath/>
7  </parent>
8
9  <dependencies>
10     <!-- web场景启动器 -->
11     <dependency>
12         <groupId>org.springframework.boot</groupId>
13         <artifactId>spring-boot-starter-web</artifactId>
14     </dependency>
15     <!-- springboot单元测试 -->
16     <dependency>
17         <groupId>org.springframework.boot</groupId>
18         <artifactId>spring-boot-starter-test</artifactId>
19         <scope>test</scope>
20     <!-- 剔除依赖 -->
21     <exclusions>
22         <exclusion>
23             <groupId>org.junit.vintage</groupId>
24             <artifactId>junit-vintage-engine</artifactId>
```

```

25         </exclusion>
26     </exclusions>
27 </dependency>
28 </dependencies>
29
30 <build>
31     <plugins>
32         <!-- 打包插件 -->
33         <plugin>
34             <groupId>org.springframework.boot</groupId>
35             <artifactId>spring-boot-maven-plugin</artifactId>
36         </plugin>
37     </plugins>
38 </build>

```

编写HTTP接口

- 1、在主程序的同级目录下，新建一个controller包，**一定要在同级目录下，否则识别不到**
- 2、在包中新建一个HelloController类

```

1 @RestController
2 public class HelloController {
3
4     @RequestMapping("/hello")
5     public String hello() {
6         return "Hello world";
7     }
8
9 }

```

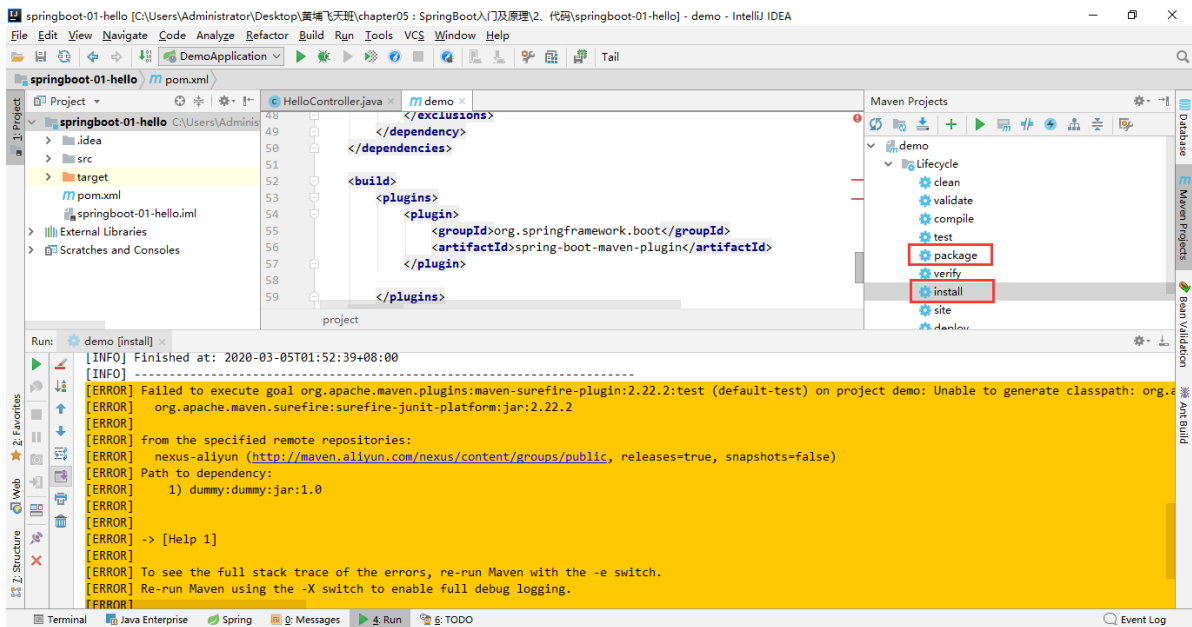
- 3、编写完毕后，从主程序启动项目，浏览器发起请求，看页面返回；控制台输出了 Tomcat 访问的端口号！



Hello World

简单几步，就完成了web接口的开发，SpringBoot就是这么简单。所以我们常用它来建立我们的微服务项目！

将项目打成jar包，点击 maven的 package



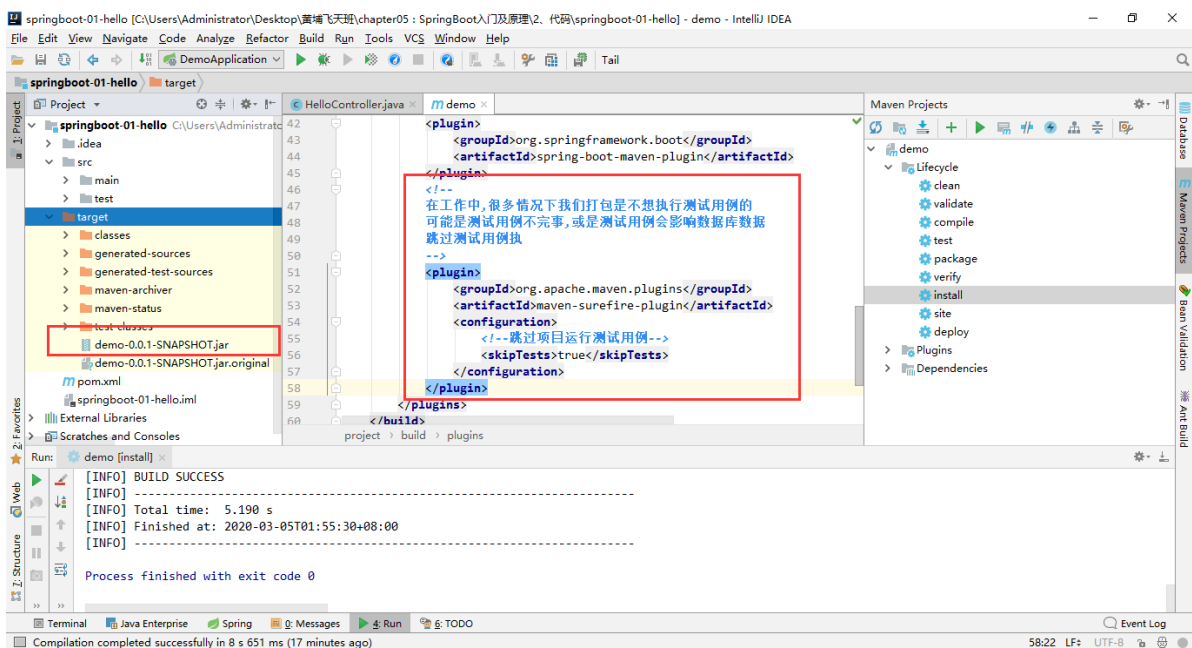
如果遇到以上错误，可以配置打包时 跳过项目运行测试用例

```

1 <!--
2     在工作中,很多情况下我们打包是不想执行测试用例的
3     可能是测试用例不完善,或是测试用例会影响数据库数据
4     跳过测试用例执
5     -->
6 <plugin>
7     <groupId>org.apache.maven.plugins</groupId>
8     <artifactId>maven-surefire-plugin</artifactId>
9     <configuration>
10        <!--跳过项目运行测试用例-->
11        <skipTests>true</skipTests>
12    </configuration>
13 </plugin>

```

如果打包成功，则会在target目录下生成一个 jar 包

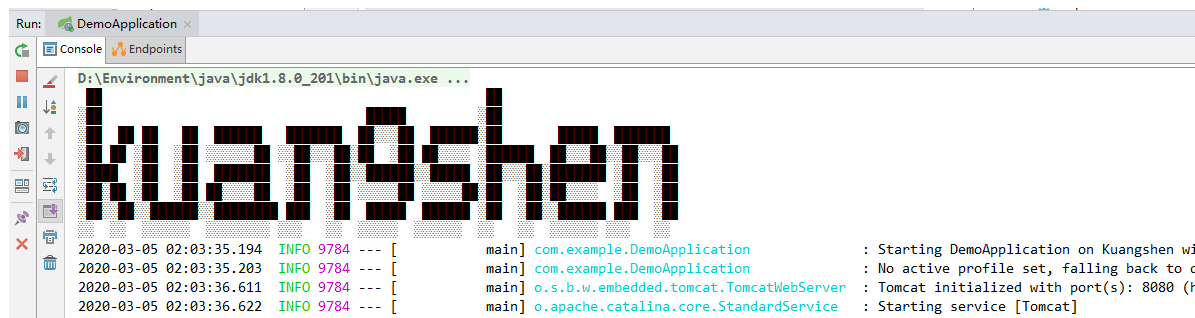


打成了jar包后，可以在任何地方运行了！OK `java -jar xxx.jar`

如何更改启动时显示的字符拼成的字母，SpringBoot呢？也就是 banner 图案；

只需一步：到项目下的 resources 目录下新建一个banner.txt 即可。

图案可以到：<https://www.bootschool.net/ascii> 这个网站生成，然后拷贝到文件中即可！



SpringBoot这么简单的东西背后一定有故事，我们之后去进行一波源码分析！

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！

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>2.2.5.RELEASE</version>
5   <relativePath/> <!-- lookup parent from repository -->
6 </parent>
```

点进去，发现还有一个父依赖

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-dependencies</artifactId>
4   <version>2.2.5.RELEASE</version>
5   <relativePath>../../spring-boot-dependencies</relativePath>
6 </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有在依赖中管理着就需要手动配置版本了；

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4 </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

```
1 // @SpringBootApplication 来标注一个主程序类, 说明这是一个Spring Boot应用
2 @SpringBootApplication
3 public class SpringbootApplication {
4
5     public static void main(String[] args) {
6         // 以为是启动了一个方法, 没想到启动了一个服务
7         SpringApplication.run(SpringbootApplication.class, args);
8     }
9
10 }
```

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!

```
1 @SpringBootConfiguration
2 @EnableAutoConfiguration
3 @ComponentScan(
4     excludeFilters = {@Filter(
5         type = FilterType.CUSTOM,
6         classes = {TypeExcludeFilter.class}
7     )}, @Filter(
8         type = FilterType.CUSTOM,
9         classes = {AutoConfigurationExcludeFilter.class}
10 })
11 )
```

```
12 public @interface SpringBootApplication {
13     // .....
14 }
```

@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

@SpringBootConfiguration

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

```
1 // 点进去得到下面的 @Component
2 @Configuration
3 public @interface SpringBootConfiguration {}
4
5 @Component
6 public @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;

点进注解继续查看:

@AutoConfigurationPackage : 自动配置包

```
1 @Import({Registrar.class})
2 public @interface AutoConfigurationPackage {
3 }
```

@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


```

1 // 获得候选的配置
2 protected List<String> getCandidateConfigurations(AnnotationMetadata
  metadata, AnnotationAttributes attributes) {
3     //这里的getSpringFactoriesLoaderFactoryClass () 方法
4     //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
5     List<String> configurations =
6     SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());
7     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.");
8     return configurations;
9 }

```

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

```

1 public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable
  ClassLoader classLoader) {
2     String factoryClassName = factoryClass.getName();
3     //这里它又调用了 loadSpringFactories 方法
4     return
5     (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName,
  Collections.emptyList());
6 }

```

3、我们继续点击查看 loadSpringFactories 方法

```

1 private static Map<String, List<String>> loadSpringFactories(@Nullable
  ClassLoader classLoader) {
2     //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
3     MultivalueMap<String, String> result =
4     (MultivalueMap)cache.get(classLoader);
5     if (result != null) {
6         return result;
7     } else {
8         try {
9             //去获取一个资源 "META-INF/spring.factories"
10            Enumeration<URL> urls = classLoader != null ?
11            classLoader.getResources("META-INF/spring.factories") :
12            ClassLoader.getSystemResources("META-INF/spring.factories");
13            LinkedMultivalueMap result = new LinkedMultivalueMap();
14
15            //将读取到的资源遍历，封装成为一个Properties
16            while(urls.hasMoreElements()) {
17                URL url = (URL)urls.nextElement();
18                URLResource resource = new URLResource(url);
19                Properties properties =
20                PropertiesLoaderUtils.loadProperties(resource);
21                Iterator var6 = properties.entrySet().iterator();
22
23                while(var6.hasNext()) {
24                    Entry<?, ?> entry = (Entry)var6.next();
25                    String factoryClassName =
26                    ((String)entry.getKey()).trim();

```

```

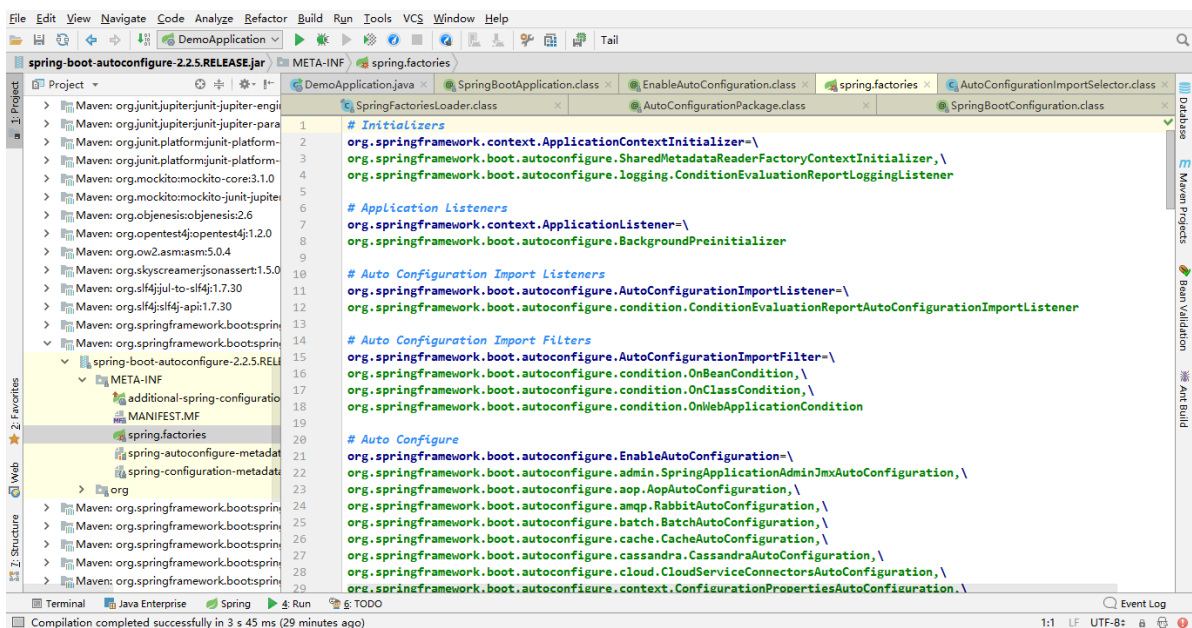
22         String[] var9 =
StringUtils.commaDelimitedListToStringArray((String)entry.getValue());
23         int var10 = var9.length;
24
25         for(int var11 = 0; var11 < var10; ++var11) {
26             String factoryName = var9[var11];
27             result.add(factoryClassName, factoryName.trim());
28         }
29     }
30 }
31
32     cache.put(classLoader, result);
33     return result;
34 } catch (IOException var13) {
35     throw new IllegalArgumentException("Unable to load factories
from location [META-INF/spring.factories]", var13);
36 }
37 }
38 }

```

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；

```
1 @SpringBootApplication
2 public class SpringbootApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(SpringbootApplication.class, args);
5     }
6 }
```

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；

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：

```
1 public SpringApplication(ResourceLoader resourceLoader, Class...  
   primarySources) {  
2     // .....  
3     this.webApplicationType = webApplicationType.deduceFromClasspath();  
4     this.setInitializers(this.getSpringFactoriesInstances());  
5  
   this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class  
6   ));  
7     this.mainApplicationClass = this.deduceMainApplicationClass();  
   }
```


YAML是 "YAML Ain't a Markup Language" (YAML不是一种标记语言) 的递归缩写。

在开发的这种语言时，YAML 的意思其实是: "Yet Another Markup Language" (仍是一种标记语言)

这种语言以数据做为中心，而不是以标记语言为重点！

以前的配置文件，大多数都是使用xml来配置；比如一个简单的端口配置，我们来对比下yaml和xml

传统xml配置：

```
1 <server>
2   <port>8081</port>
3 </server>
```

yaml配置：

```
1 server:
2   prot: 8080
```

yml基础语法

说明：语法要求严格！

- 1、空格不能省略
- 2、以缩进来控制层级关系，只要是左边对齐的一列数据都是同一个层级的。
- 3、属性和值的大小写都是十分敏感的。

字面量：普通的值 [数字，布尔值，字符串]

字面量直接写在后面就可以，字符串默认不用加上双引号或者单引号；

```
1 k: v
```

注意：

- “” 双引号，不会转义字符串里面的特殊字符，特殊字符会作为本身想表示的意思；
比如：name: "kuang \n shen" 输出：kuang 换行 shen
- " 单引号，会转义特殊字符，特殊字符最终会变成和普通字符一样输出
比如：name: 'kuang \n shen' 输出：kuang \n shen

对象、Map (键值对)

```
1 #对象、Map格式
2 k:
3   v1:
4   v2:
```

在下一行来写对象的属性和值得关系，注意缩进；比如：

```
1 student:
2   name: qinjiang
3   age: 3
```

行内写法

```
1 student: {name: qinjiang,age: 3}
```

数组 (List、 set)

用 - 值表示数组中的一个元素,比如:

```
1 pets:
2   - cat
3   - dog
4   - pig
```

行内写法

```
1 pets: [cat,dog,pig]
```

修改SpringBoot的默认端口号

配置文件中添加, 端口号的参数, 就可以切换端口;

```
1 server:
2   port: 8082
```

注入配置文件

yaml文件更强大的地方在于, 他可以给我们的实体类直接注入匹配值!

Yaml注入配置文件

- 1、在springboot项目中的resources目录下新建一个文件 application.yml
- 2、编写一个实体类 Dog;

```
1 package com.kuang.springboot.pojo;
2
3 @Component //注册bean到容器中
4 public class Dog {
5     private String name;
6     private Integer age;
7
8     //有参无参构造、get、set方法、toString()方法
9 }
```

- 3、思考, 我们原来是如何给bean注入属性值的! @Value, 给狗狗类测试一下:

```

1  @Component //注册bean
2  public class Dog {
3      @Value("阿黄")
4      private String name;
5      @Value("18")
6      private Integer age;
7  }

```

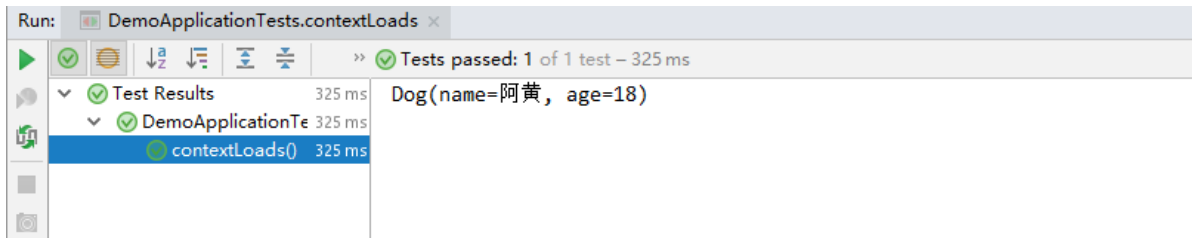
4、在SpringBoot的测试类下注入狗狗输出一下；

```

1  @SpringBootTest
2  class DemoApplicationTests {
3
4      @Autowired //将狗狗自动注入进来
5      Dog dog;
6
7      @Test
8      public void contextLoads() {
9          System.out.println(dog); //打印看下狗狗对象
10     }
11
12 }

```

结果成功输出，@Value注入成功，这是我们原来的办法对吧。



5、我们在编写一个复杂一点的实体类：Person 类

```

1  @Component //注册bean到容器中
2  public class Person {
3      private String name;
4      private Integer age;
5      private Boolean happy;
6      private Date birth;
7      private Map<String,Object> maps;
8      private List<Object> lists;
9      private Dog dog;
10
11      //有参无参构造、get、set方法、toString()方法
12  }

```

6、我们来使用yaml配置的方式进行注入，大家写的时候注意区别和优势，我们编写一个yaml配置


```

1 person:
2   name: qinjiang
3   age: 3
4   happy: false
5   birth: 2000/01/01
6   maps: {k1: v1,k2: v2}
7   lists:
8     - code
9     - girl
10    - music
11 dog:
12   name: 旺财
13   age: 1

```

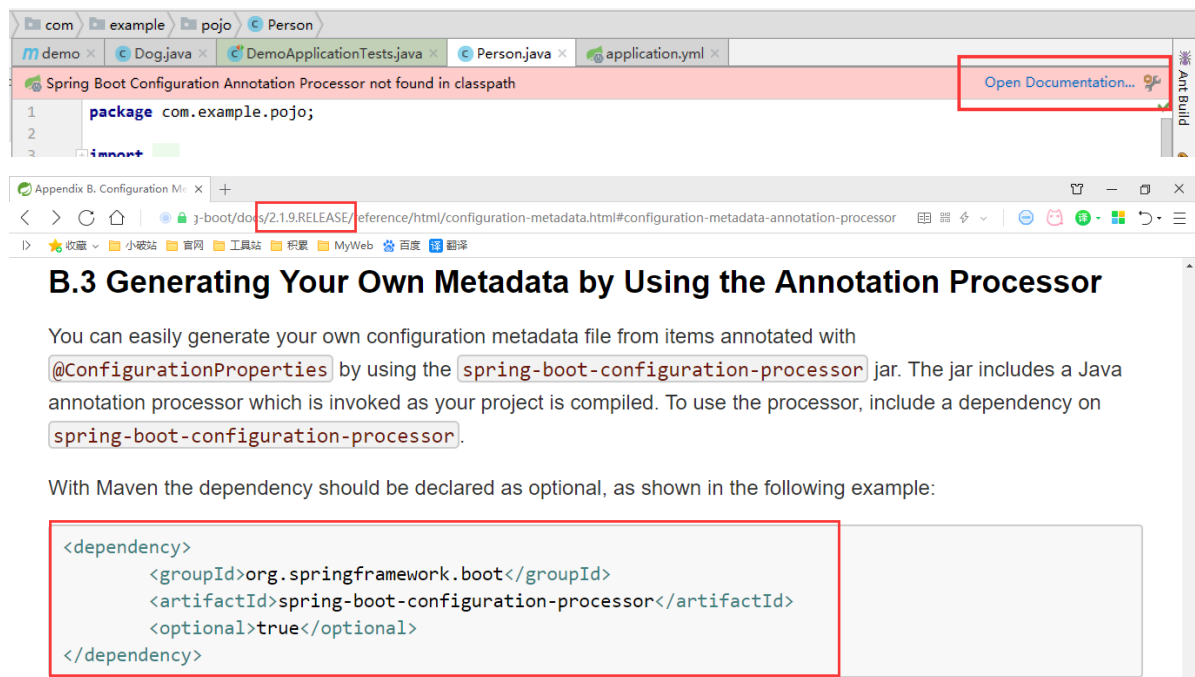
7、我们刚才已经把person这个对象的所有值都写好了，我们现在来注入到我们的类中！

```

1  /*
2   @ConfigurationProperties作用：
3   将配置文件中配置的每一个属性的值，映射到这个组件中；
4   告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定
5   参数 prefix = "person" ： 将配置文件中的person下面的所有属性一一对应
6   */
7  @Component //注册bean
8  @ConfigurationProperties(prefix = "person")
9  public class Person {
10     private String name;
11     private Integer age;
12     private Boolean happy;
13     private Date birth;
14     private Map<String,Object> maps;
15     private List<Object> lists;
16     private Dog dog;
17 }

```

8、IDEA 提示，springboot配置注解处理器没有找到，让我们看文档，我们可以查看文档，找到一个依赖！



The screenshot shows an IDE window with a red error bar: "Spring Boot Configuration Annotation Processor not found in classpath". A red box highlights the "Open Documentation..." link. Below, a web browser shows the Spring Boot documentation page titled "B.3 Generating Your Own Metadata by Using the Annotation Processor". The page explains that the `@ConfigurationProperties` annotation uses the `spring-boot-configuration-processor` jar. A red box highlights the Maven dependency code snippet:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>

```

```

1 <!-- 导入配置文件处理器，配置文件进行绑定就会有提示，需要重启 -->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-configuration-processor</artifactId>
5     <optional>true</optional>
6 </dependency>

```

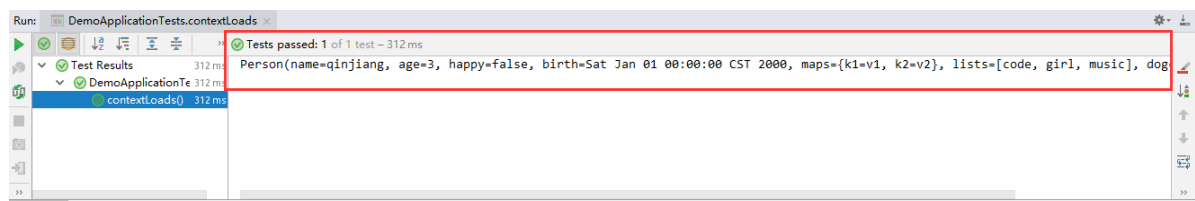
9、确认以上配置都OK之后，我们去测试类中测试一下：

```

1 @SpringBootTest
2 class DemoApplicationTests {
3
4     @Autowired
5     Person person; //将person自动注入进来
6
7     @Test
8     public void contextLoads() {
9         System.out.println(person); //打印person信息
10    }
11
12 }

```

结果：所有值全部注入成功！



yaml配置注入到实体类完全OK!

课堂测试：

- 1、将配置文件的key 值 和 属性的值设置为不一样，则结果输出为null，注入失败
- 2、在配置一个person2，然后将 @ConfigurationProperties(prefix = "person2") 指向我们的 person2;

加载指定配置文件

@PropertySource：加载指定的配置文件；

@configurationProperties：默认从全局配置文件中获取值；

- 1、我们去在resources目录下新建一个**person.properties**文件

```

1 name=kuangshen

```

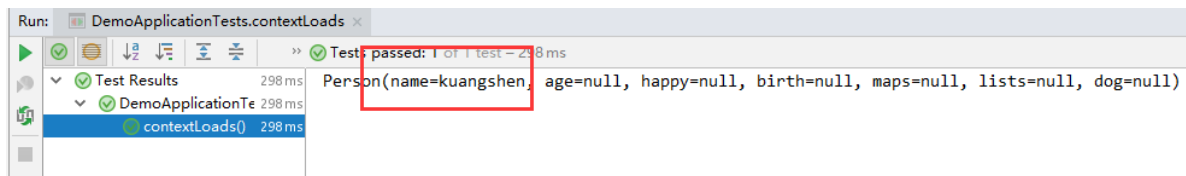
- 2、然后在我们的代码中指定加载person.properties文件

```

1 @PropertySource(value = "classpath:person.properties")
2 @Component //注册bean
3 public class Person {
4
5     @Value("${name}")
6     private String name;
7
8     .....
9 }

```

3、再次输出测试一下：指定配置文件绑定成功！



配置文件占位符

```

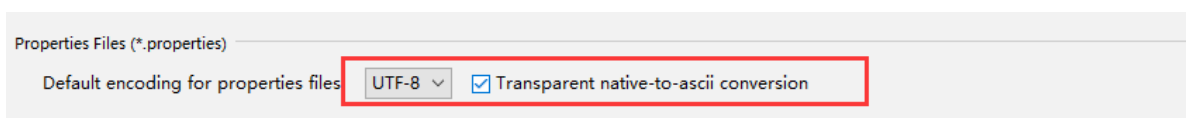
1 person:
2   name: qinjiang${random.uuid} # 随机uuid
3   age: ${random.int} # 随机int
4   happy: false
5   birth: 2000/01/01
6   maps: {k1: v1,k2: v2}
7   lists:
8     - code
9     - girl
10    - music
11  dog:
12    # 引用person.hello 的值，如果不存在就用 : 后面的值，即 other，然后拼接上_旺财
13    name: ${person.hello:other}_旺财
14    age: 1

```

回顾properties配置

我们上面采用的yaml方法都是最简单的方式，开发中最常用的；也是springboot所推荐的！那我们来唠唠其他的实现方式，道理都是相同得；写还是那样写；配置文件除了yaml还有我们之前常用的properties，我们没有讲，我们来唠唠！

【注意】properties配置文件在写中文的时候，会有乱码，我们需要去IDEA中设置编码格式为UTF-8；settings-->FileEncodings 中配置；



测试步骤：

1、新建一个实体类User

```

1 @Component //注册bean
2 public class User {
3     private String name;
4     private int age;
5     private String sex;
6 }

```

2、编辑配置文件 user.properties

```

1 user1.name=kuangshen
2 user1.age=18
3 user1.sex=男

```

3、我们在User类上使用@Value来进行注入！

```

1 @Component //注册bean
2 @PropertySource(value = "classpath:user.properties")
3 public class User {
4     //直接使用@value
5     @Value("${user.name}") //从配置文件中取值
6     private String name;
7     @Value("#{9*2}") // #{SPEL} Spring表达式
8     private int age;
9     @Value("男") // 字面量
10    private String sex;
11 }

```

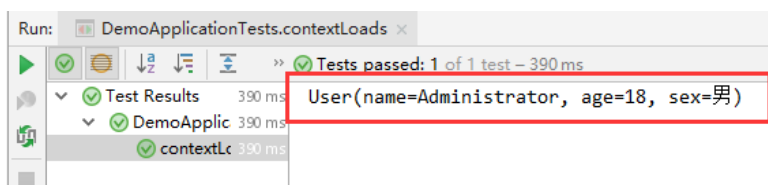
4、Springboot测试

```

1 @SpringBootTest
2 class DemoApplicationTests {
3
4     @Autowired
5     User user;
6
7     @Test
8     public void contextLoads() {
9         System.out.println(user);
10    }
11
12 }

```

结果正常输出：



对比小结

@Value这个使用起来并不友好！我们需要为每个属性单独注解赋值，比较麻烦；我们来看个功能对比图

	@ConfigurationProperties	@Value
功能	批量注入配置文件中的属性	一个个指定
松散绑定（松散语法）	支持	不支持
SpEL	不支持	支持
JSR303数据校验	支持	不支持
复杂类型封装	支持	不支持

1、@ConfigurationProperties只需要写一次即可，@Value则需要每个字段都添加

2、松散绑定：这个什么意思呢？比如我的yml中写的last-name，这个和lastName是一样的，-后面跟着的字母默认是大写的。这就是松散绑定。可以测试一下

3、JSR303数据校验，这个就是我们可以为字段增加一层过滤器验证，可以保证数据的合法性

4、复杂类型封装，yml中可以封装对象，使用value就不支持

结论：

配置yml和配置properties都可以获取到值，强烈推荐 yml；

如果我们在某个业务中，只需要获取配置文件中的某个值，可以使用一下 @value；

如果说，我们专门编写了一个JavaBean来和配置文件进行一一映射，就直接
@configurationProperties，不要犹豫！

JSR303数据校验

Springboot中可以用@validated来校验数据，如果数据异常则会统一抛出异常，方便异常中心统一处理。我们这里来写个注解让我们的name只能支持Email格式；

```

1  @Component //注册bean
2  @ConfigurationProperties(prefix = "person")
3  @Validated //数据校验
4  public class Person {
5
6      @Email(message="邮箱格式错误") //name必须是邮箱格式
7      private String name;
8  }

```

运行结果： default message [不是一个合法的电子邮件地址];

```

org.springframework.boot.context.properties.bind.validation.BindValidationException: Binding validation errors
in object 'person' on field 'name': rejected value [qinjiang]; codes [Email.person.name,Email.name,Email
;framework.boot.context.properties.bind.validation.ValidationBindHandler.getBindValidationException(Valid

```

使用数据校验，可以保证数据的正确性；下面列出一些常见的使用

```

1  @NotNull(message="名字不能为空")
2  private String userName;
3  @Max(value=120,message="年龄最大不能超过120")
4  private int age;
5  @Email(message="邮箱格式错误")
6  private String email;
7
8  空检查

```

```

9  @Null      验证对象是否为null
10 @NotNull   验证对象是否不为null，无法查检长度为0的字符串
11 @NotBlank  检查约束字符串是不是Null还有被Trim的长度是否大于0,只对字符串,且会去掉前后空格.
12 @NotEmpty  检查约束元素是否为NULL或者是EMPTY.
13
14 Boolean检查
15 @AssertTrue    验证 Boolean 对象是否为 true
16 @AssertFalse   验证 Boolean 对象是否为 false
17
18 长度检查
19 @Size(min=, max=) 验证对象（Array,Collection,Map,String）长度是否在给定的范围之内
20
21 @Length(min=, max=) string is between min and max included.
22
23 日期检查
24 @Past          验证 Date 和 Calendar 对象是否在当前时间之前
25 @Future        验证 Date 和 Calendar 对象是否在当前时间之后
26 @Pattern       验证 String 对象是否符合正则表达式的规则
27
28 .....等等
28 除此以外，我们还可以自定义一些数据校验规则

```

多环境切换

profile是Spring对不同环境提供不同配置功能的支持，可以通过激活不同的环境版本，实现快速切换环境；

多配置文件

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml，用来指定多个环境版本；

例如：application-test.properties 代表测试环境配置 application-dev.properties 代表开发环境配置

但是Springboot并不会直接启动这些配置文件，它默认使用application.properties主配置文件；

我们需要通过一个配置来选择需要激活的环境：

```

1  #比如在配置文件中指定使用dev环境，我们可以通过设置不同的端口号进行测试；
2  #我们启动SpringBoot，就可以看到已经切换到dev下的配置了；
3  spring.profiles.active=dev

```

yml的多文档块

和properties配置文件中一样，但是使用yml去实现不需要创建多个配置文件，更加方便了！

```

1  server:
2    port: 8081
3    #选择要激活那个环境块
4  spring:
5    profiles:
6      active: prod

```

```

7
8 ---
9 server:
10   port: 8083
11 spring:
12   profiles: dev #配置环境的名称
13
14
15 ---
16
17 server:
18   port: 8084
19 spring:
20   profiles: prod #配置环境的名称

```

注意：如果yml和properties同时都配置了端口，并且没有激活其他环境，默认会使用properties配置文件的！

配置文件加载位置

外部加载配置文件的方式十分多，我们选择最常用的即可，在开发的资源文件中进行配置！

[官方外部配置文件说明参考文档](#)

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

- 1 优先级1：项目路径下的config文件夹配置文件
- 2 优先级2：项目路径下配置文件
- 3 优先级3：资源路径下的config文件夹配置文件
- 4 优先级4：资源路径下配置文件

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

SpringBoot会从这四个位置全部加载主配置文件；互补配置；

我们在最低级的配置文件中设置一个项目访问路径的配置来测试互补问题；

- 1 #配置项目的访问路径
- 2 server.servlet.context-path=/kuang

扩展：指定位置加载配置文件

我们还可以通过spring.config.location来改变默认的配置文件的配置位置

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；这种情况，一般是后期运维做的多，相同配置，外部指定的配置文件优先级最高

- 1 java -jar spring-boot-config.jar --
spring.config.location=F:/application.properties

自动配置原理

配置文件到底能写什么？怎么写？

[SpringBoot官方文档](#)

分析自动配置原理

我们以**HttpEncodingAutoConfiguration**（Http编码自动配置）为例解释自动配置原理；

```
1 //表示这是一个配置类，和以前编写的配置文件一样，也可以给容器中添加组件；
2 @Configuration
3
4 //启动指定类的ConfigurationProperties功能；
5 //进入这个HttpProperties查看，将配置文件中对应的值和HttpProperties绑定起来；
6 //并把HttpProperties加入到ioc容器中
7 @EnableConfigurationProperties({HttpProperties.class})
8
9 //Spring底层@Conditional注解
10 //根据不同的条件判断，如果满足指定的条件，整个配置类里面的配置就会生效；
11 //这里的意思就是判断当前应用是否是web应用，如果是，当前配置类生效
12 @ConditionalOnWebApplication(
13     type = Type.SERVLET
14 )
15
16 //判断当前项目有没有这个类CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器；
17 @ConditionalOnClass({CharacterEncodingFilter.class})
18
19 //判断配置文件中是否存在某个配置：spring.http.encoding.enabled；
20 //如果不存在，判断也是成立的
21 //即使我们配置文件中不配置spring.http.encoding.enabled=true，也是默认生效的；
22 @ConditionalOnProperty(
23     prefix = "spring.http.encoding",
24     value = {"enabled"},
25     matchIfMissing = true
26 )
27
28 public class HttpEncodingAutoConfiguration {
29     //他已经和SpringBoot的配置文件映射了
30     private final Encoding properties;
31     //只有一个有参构造器的情况下，参数的值就会从容器中拿
32     public HttpEncodingAutoConfiguration(HttpProperties properties) {
33         this.properties = properties.getEncoding();
34     }
35
36     //给容器中添加一个组件，这个组件的某些值需要从properties中获取
37     @Bean
38     @ConditionalOnMissingBean //判断容器没有这个组件？
39     public CharacterEncodingFilter characterEncodingFilter() {
40         CharacterEncodingFilter filter = new
41             OrderedCharacterEncodingFilter();
42         filter.setEncoding(this.properties.getCharset().name());
```



```

42     filter.setForceRequestEncoding(this.properties.shouldForce(org.springframework
    ork.boot.autoconfigure.http.HttpProperties.Encoding.Type.REQUEST));
43
    filter.setForceResponseEncoding(this.properties.shouldForce(org.springframework
    work.boot.autoconfigure.http.HttpProperties.Encoding.Type.RESPONSE));
44     return filter;
45 }
46 //.....
47 }

```

一句话总结：根据当前不同的条件判断，决定这个配置类是否生效！

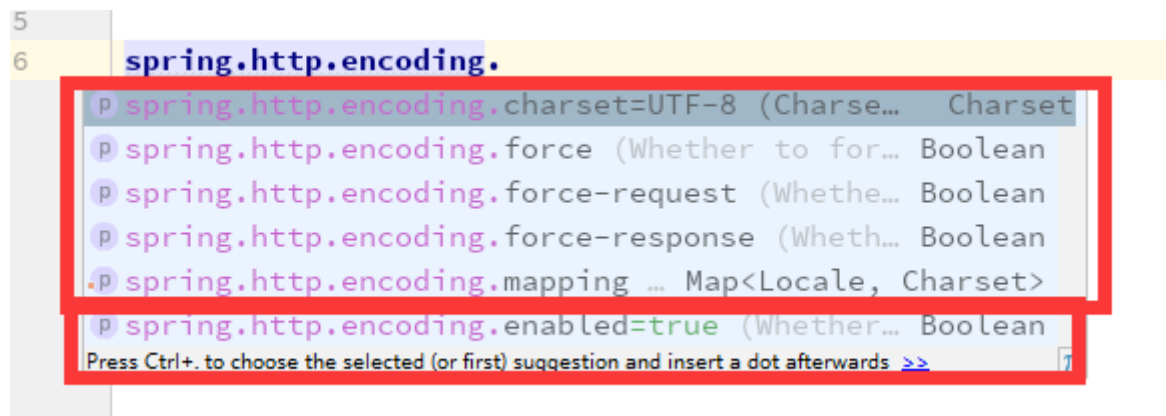
- 一但这个配置类生效；这个配置类就会给容器中添加各种组件；
- 这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的；
- 所有在配置文件中能配置的属性都是在xxxxProperties类中封装着；
- 配置文件能配置什么就可以参照某个功能对应的这个属性类

```

1 //从配置文件中获取指定的值和bean的属性进行绑定
2 @ConfigurationProperties(prefix = "spring.http")
3 public class HttpProperties {
4     // .....
5 }

```

我们去配置文件里面试试前缀，看提示！



```

5
6 spring.http.encoding.
    P spring.http.encoding.charset=UTF-8 (Charset... Charset
    P spring.http.encoding.force (Whether to for... Boolean
    P spring.http.encoding.force-request (Whethe... Boolean
    P spring.http.encoding.force-response (Wheth... Boolean
    P spring.http.encoding.mapping ... Map<Locale, Charset>
    P spring.http.encoding.enabled=true (Whether... Boolean
    Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards >>

```

这就是自动装配的原理！

精髓

- 1、SpringBoot启动会加载大量的自动配置类
- 2、我们看我们需要的功能有没有在SpringBoot默认写好的自动配置类当中；
- 3、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件存在在其中，我们就不需要再手动配置了）
- 4、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们只需要在配置文件中指定这些属性的值即可；

xxxxAutoConfigurartion：自动配置类；给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

了解完自动装配的原理后，我们来关注一个细节问题，**自动配置类必须在一定的条件下才能生效**；

@Conditional派生注解（Spring注解版原生的@Conditional作用）

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效；

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean；
@ConditionalOnMissingBean	容器中不存在指定Bean；
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

那么多的自动配置类，必须在一定的条件下才能生效；也就是说，我们加载了这么多的配置类，但不是所有的都生效了。

我们怎么知道哪些自动配置类生效？

我们可以通过启用 debug=true属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置类生效；

```
1 #开启springboot的调试类
2 debug=true
```

Positive matches:（自动配置类启用的：正匹配）

Negative matches:（没有启动，没有匹配成功的自动配置类：负匹配）

Unconditional classes:（没有条件的类）

【演示：查看输出的日志】

掌握吸收理解原理，即可以不变应万变！

提高：自定义starter

我们分析完毕了源码以及自动装配的过程，我们可以尝试自定义一个启动器来玩玩！

启动器模块是一个空 jar 文件，仅提供辅助性依赖管理，这些依赖可能用于自动装配或者其他类库；

命名归约：

官方命名：

- 前缀：spring-boot-starter-xxx
- 比如：spring-boot-starter-web....

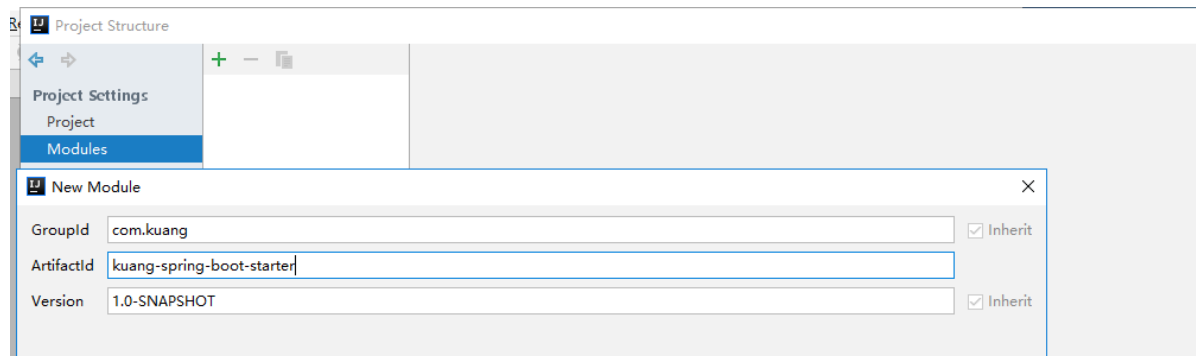
自定义命名：

- xxx-spring-boot-starter
- 比如：mybatis-spring-boot-starter

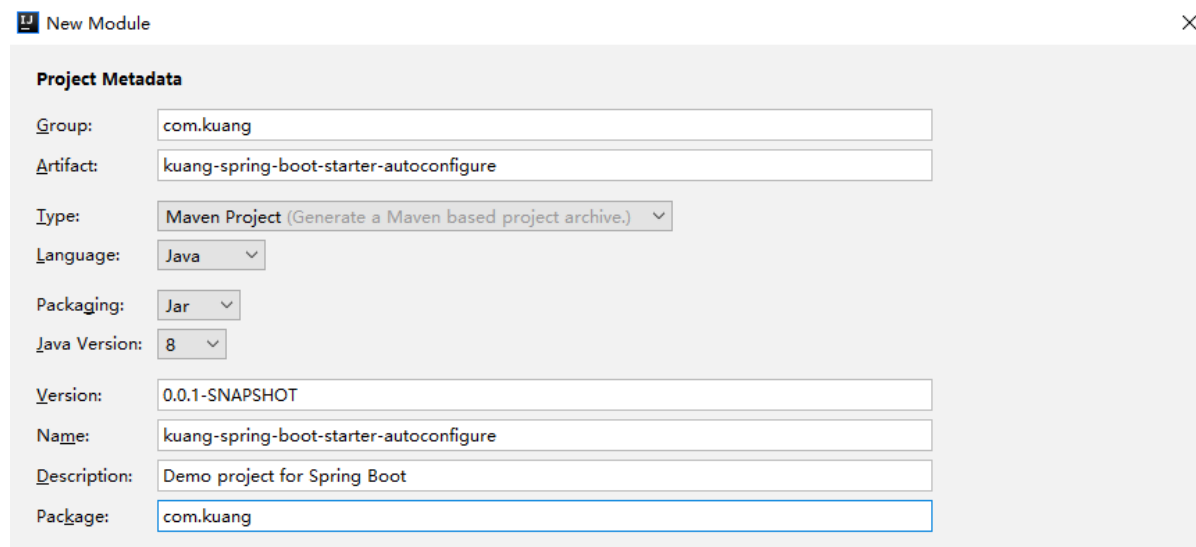
编写启动器

1、在IDEA中新建一个空项目 spring-boot-starter-diy

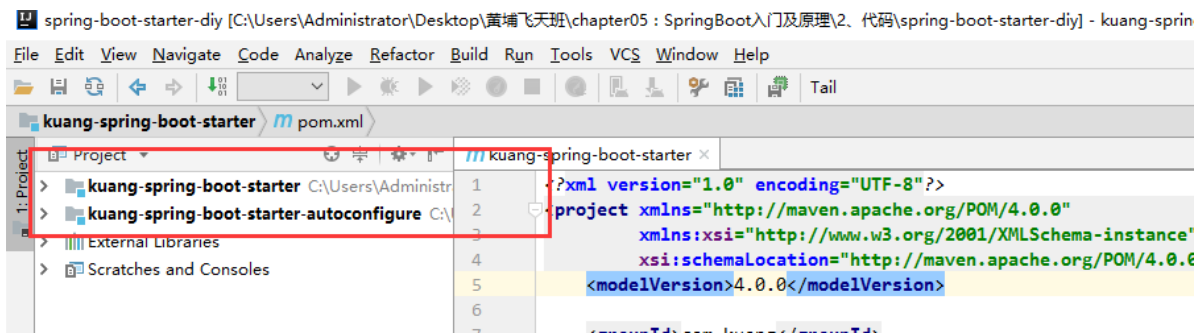
2、新建一个普通Maven模块：kuang-spring-boot-starter



3、新建一个Springboot模块：kuang-spring-boot-starter-autoconfigure



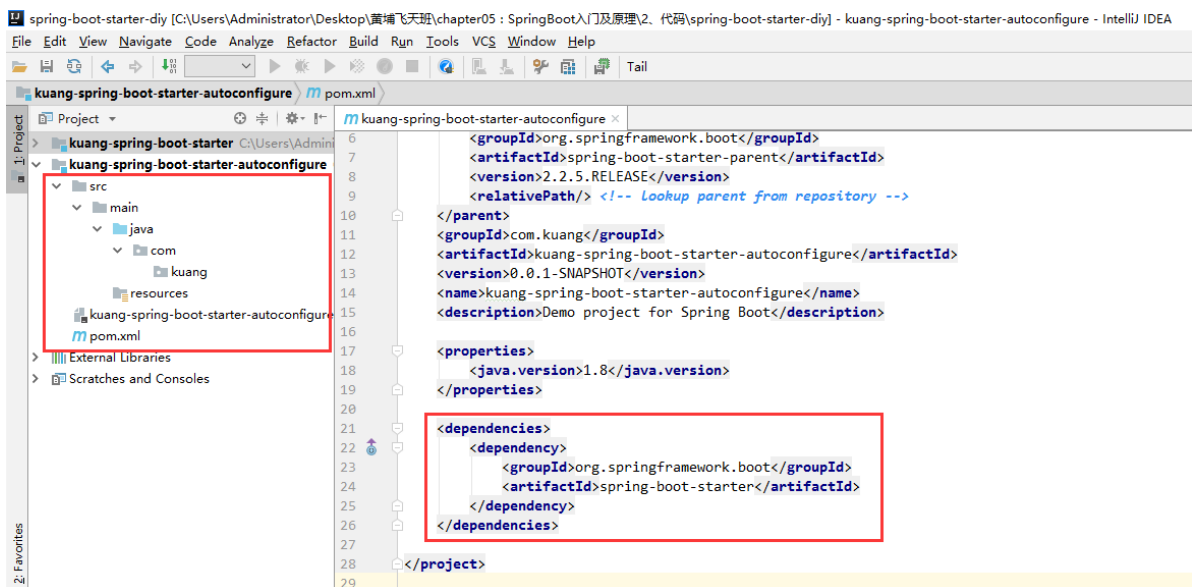
4、点击apply即可，基本结构



5、在我们的 starter 中导入 autoconfigure 的依赖!

```
1 <!-- 启动器 -->
2 <dependencies>
3     <!-- 引入自动配置模块 -->
4     <dependency>
5         <groupId>com.kuang</groupId>
6         <artifactId>kuang-spring-boot-starter-autoconfigure</artifactId>
7         <version>0.0.1-SNAPSHOT</version>
8     </dependency>
9 </dependencies>
```

6、将 autoconfigure 项目下多余的文件都删掉，Pom中只留下一个 starter，这是所有的启动器基本配置



7、我们编写一个自己的服务

```
1 package com.kuang;
2
3 public class HelloService {
4
5     HelloProperties helloProperties;
6
7     public HelloProperties getHelloProperties() {
8         return helloProperties;
9     }
10
11     public void setHelloProperties(HelloProperties helloProperties) {
12         this.helloProperties = helloProperties;
13     }
14 }
```

```

15     public String sayHello(String name){
16         return helloProperties.getPrefix() + name +
helloProperties.getSuffix();
17     }
18
19 }

```

8、编写HelloProperties 配置类

```

1  package com.kuang;
2
3  import org.springframework.boot.context.properties.ConfigurationProperties;
4
5  // 前缀 kuang.hello
6  @ConfigurationProperties(prefix = "kuang.hello")
7  public class HelloProperties {
8
9      private String prefix;
10     private String suffix;
11
12     public String getPrefix() {
13         return prefix;
14     }
15
16     public void setPrefix(String prefix) {
17         this.prefix = prefix;
18     }
19
20     public String getSuffix() {
21         return suffix;
22     }
23
24     public void setSuffix(String suffix) {
25         this.suffix = suffix;
26     }
27 }

```

9、编写我们的自动配置类并注入bean，测试！

```

1  package com.kuang;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import
org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication
;
5  import
org.springframework.boot.context.properties.EnableConfigurationProperties;
6  import org.springframework.context.annotation.Bean;
7  import org.springframework.context.annotation.Configuration;
8
9  @Configuration
10 @ConditionalOnWebApplication //web应用生效
11 @EnableConfigurationProperties(HelloProperties.class)
12 public class HelloServiceAutoConfiguration {
13
14     @Autowired
15     HelloProperties helloProperties;
16

```

```

17     @Bean
18     public HelloService helloService(){
19         HelloService service = new HelloService();
20         service.setHelloProperties(helloProperties);
21         return service;
22     }
23
24 }

```

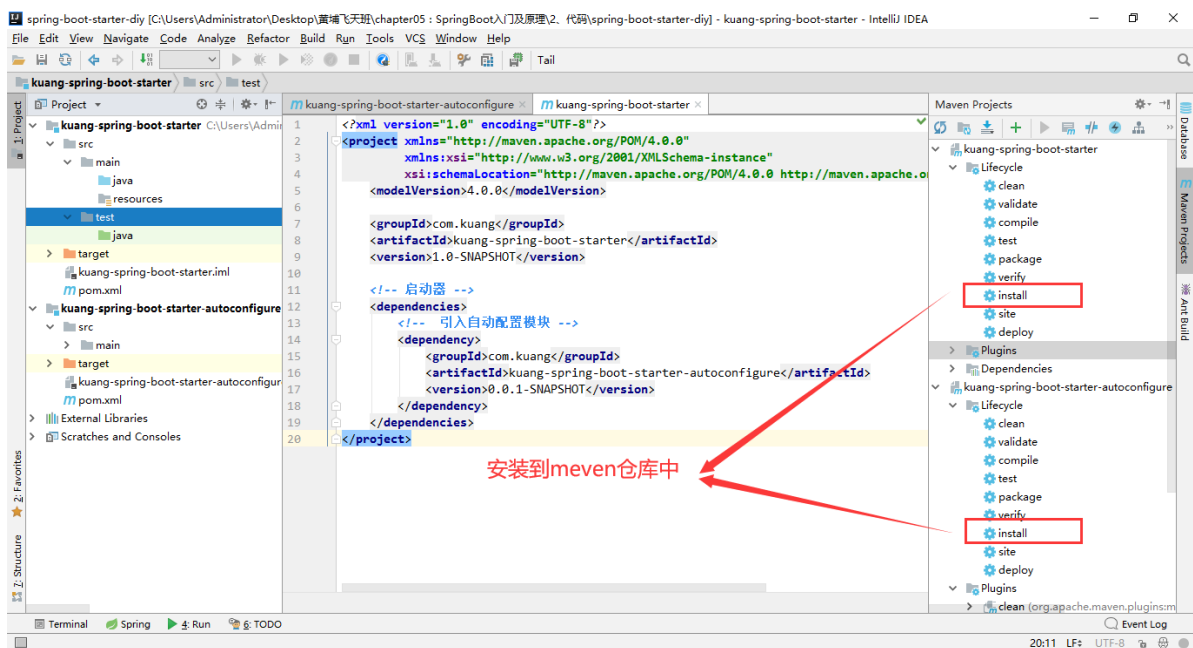
10、在resources编写一个自己的 `META-INF\spring.factories`

```

1 # Auto Configure
2 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
3 com.kuang.HelloServiceAutoConfiguration

```

11、编写完成后，可以安装到maven仓库中！



新建项目测试我们自己的写的启动器

- 1、新建一个SpringBoot 项目
- 2、导入我们自己写的启动器

```

1 <dependency>
2     <groupId>com.kuang</groupId>
3     <artifactId>kuang-spring-boot-starter</artifactId>
4     <version>1.0-SNAPSHOT</version>
5 </dependency>

```

3、编写一个 HelloController 进行测试我们自己的写的接口！

```

1 package com.kuang.controller;
2
3 @RestController
4 public class HelloController {
5
6     @Autowired

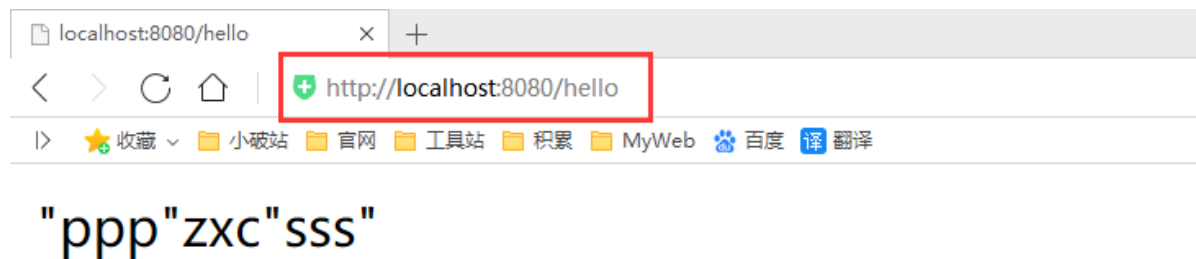
```

```
7     HelloService helloService;  
8  
9     @RequestMapping("/hello")  
10    public String hello(){  
11        return helloService.sayHello("zxc");  
12    }  
13  
14 }
```

4、编写配置文件 application.properties

```
1 kuang.hello.prefix="ppp"  
2 kuang.hello.suffix="sss"
```

5、启动项目进行测试，结果成功！



小狂神温馨提示：学完的东西一定要多下去实践！