# 框架高级课程系列之Spring Boot

## 尚硅谷JavaEE教研组

## 版本：V3.4

# 一. Spring Boot 概述与入门(掌握)

## 1.1 Spring Boot 概述

### 1.1.1 什么是Spring Boot

Spring是一个非常成功的框架，最核心的理念是IoC（控制反转）和AOP（面向切面编程）。随着Spring的出现、SSH、SSH2，SSM称为目前主流的企业级开发技术。但是SSM肯定不是企业级开发技术的终点，本身也有不足之处的。

Spring开发使用了大量的XML配置文件，配置繁琐，整合不易，开发和部署效率低下（前几天还在夸奖SSM呢，今天就开始贬低了，因为简化软件开发技术无止境），这时急切的需要一种能解决这些问题的快速开发框架，SpringBoot就是这种快速开发框架。



经常遇到这样的场景：一个初学者花了半个月的时间，看了几本Spring书籍，掌握了最基本的理论知识，但是在实际开发中，往往被初始环境配置挡住了去路，非常痛苦，怀疑自己是否适合学习，甚至在起点就选择了放弃，坚持下来的依然面临着大量繁琐的配置工作。而使用SpringBoot不会存在这些问题。

SpringBoot的设计初衷是解决Spring配置工作过于繁琐的问题，简化初始搭建流程，降低开发难度，使开发人员只需要专注应用程序的功能和业务逻辑实现，而不用在配置上花费太多时间。SpringBoot使用“默认大于配置”的理解，提供了很多已经集成好的方案，以便开发者在开发时可以零配置或者极简配置。

总之，十多年前，Spring颠覆了传统的JavaEE技术，迎来了Java企业级应用开发的春天。然而今天的Spring Boot却站在Spring巨人的肩膀上，让我们可以更高效地开发与交付，**SpringBoot是Spring的扩展和自动化。SpringBoot是当前后台开发极佳框架**，表现亮眼突出。**SpringBoot将逐渐替代SSM开发，更简单更自动化**。SpringBoot整合Spring技术栈的一站式框架，推荐将原生的spring开发升级为springboot开发

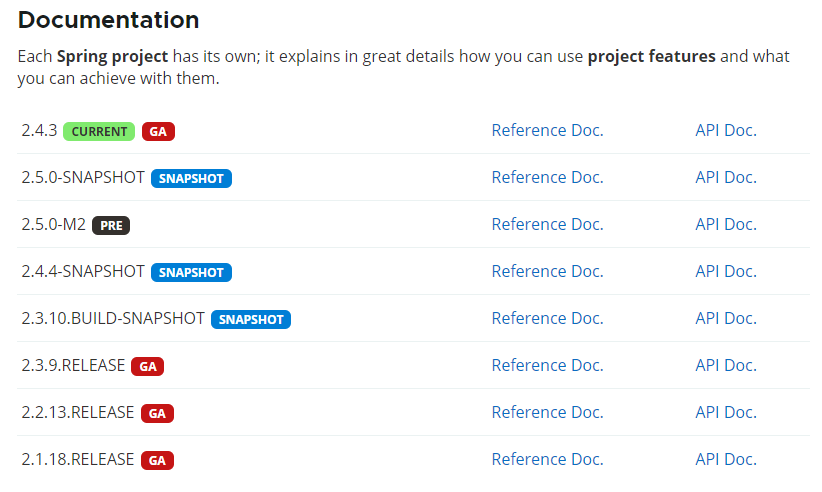
### 1.1.2 SpringBoot的发展

|  |  |  |
| --- | --- | --- |
| 时间 | 版本 | 特征 |
| 2013年初 |  | 开始的 Spring Boot 项目的研发 |
| 2014年4月 | 1.0 |  |
| 2014年6月 | 1.1 | 改进的模板支持，gemfire 支持，elasticsearch 和 apache solr 的自动配置 |
| 2015年3月 | 1.2 | 升级到 servlet 3.1 / tomcat 8 / jetty 9，spring 4.1 升级，支持 banner / jms / SpringBootApplication 注解 |
| 2016年12月 | 1.3 | Spring 4.2 升级，新的 spring-boot-devtools，用于缓存技术的自动配置以及完全可执行的 jar 支持 |
| 2017年1月 | 1.4 | Spring 4.3 升级，支持 couchbase / neo4j，分析启动失败和RestTemplateBuilder |
| 2017年2月 | 1.5 | 支持 kafka / ldap，第三方库升级，弃用 CRaSH 支持和执行器记录器端点以动态修改应用程序日志级别 |
| 2018年3月 | 2.0 | **基于 Java 8**，支持 Java 9，支持 Quartz，调度程序大大简化了安全自动配置，支持嵌入式 Netty |
| 2018年11月 | 2.1 |  |
| 2019年10月 | 2.2 |  |
| 2020年5月 | 2.3 |  |
| 2020年11月 | 2.4 |  |

### 1.1.3 SpringBoot官网学习资料

SpringBoot官网地址：<https://spring.io/projects/spring-boot>

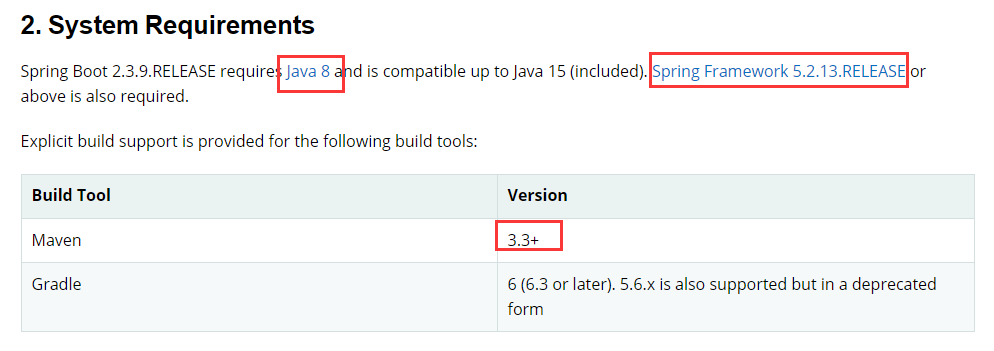
SpringBoot文档下载地址：<https://spring.io/projects/spring-boot#learn> 提供了线上参考文档和线上API。



SpringBoot版本

* GA：General Availability，正式发布的版本，官方推荐使用版本
* Snapshot：快照版，可以稳定使用，且仍在继续改进版本
* PRE：预览版,内部测试版. 主要是给开发人员和测试人员测试和找BUG用的，不建议使用；

官网文档明确说明了不同版本的SpringBoot需要的JDK、Spring、Maven等版本要求。



### 1.1.4 SpringBoot特色

SpringBoot最主要作用就是帮助开发人员快速的构建庞大的spring项目，并且尽可能的减少一切xml配置，做到**开箱即用**，迅速上手，让开发人员关注业务而非配置。

SpringBoot的特色用一个词汇表示，就是简单。当然，简单就会带来快速和方便。当然，简单并强大着。

* 使用简单：支持**使用注解**轻松实现类定义和功能开发，无代码生成和XML配置
* 配置简单：能自**动完成大量配置**；还提供**了大量的Starter来简化依赖配置**，大大减少了依赖的jar的数量
* 部署简单：**内置了嵌入式的Tomca**t、Jetty、Netty等Servlet容器，项目不需要被打包成WAR格式，**可以直接以JAR包的方式**运行
* 监控简单：提供了一整套的监控、管理应用程序状态的功能模块
* 提供了**极其强大的整合功能**：SpringMVC、MyBatis、JSP、Thymeleaf、FreeMaker、数据源、日志、缓存、安全、搜索、消息队列、测试、定时器等等，SpringBoot都提供了集成/整合功能，从而简化操作。
* 和**微服务天然集成**：微服务开发框架SpringCloud也是基于SpringBoot实现。





**主要特点:**

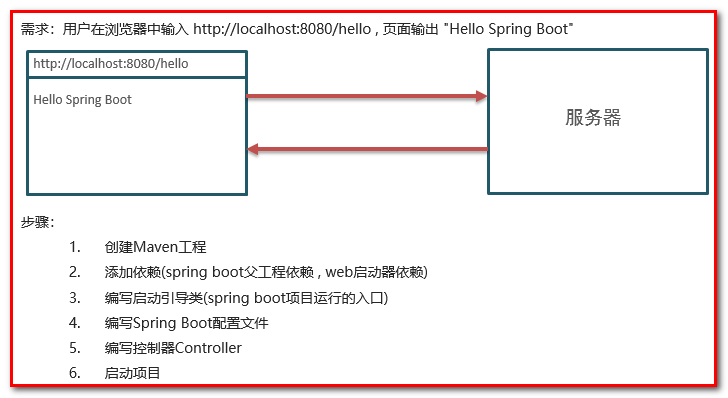
1. 自动配置 : 不需要再关注各个框架的整合配置, springboot全部已经配置好了
2. 起步依赖 : 我们在需要使用某个框架的时候, 直接添加这个框架的启动器starter依赖即可 , 不需要在关注jar包的冲突和整合

## 1.2 Spring Boot 入门1-从创建Maven项目开始

下面，我们就利用Spring Boot搭建一个web工程，体会一下Spring Boot的魅力所在！

### 1.2.1 需求

需求：访问 http://localhost:8080/hello输出 “Hello Spring Boot”



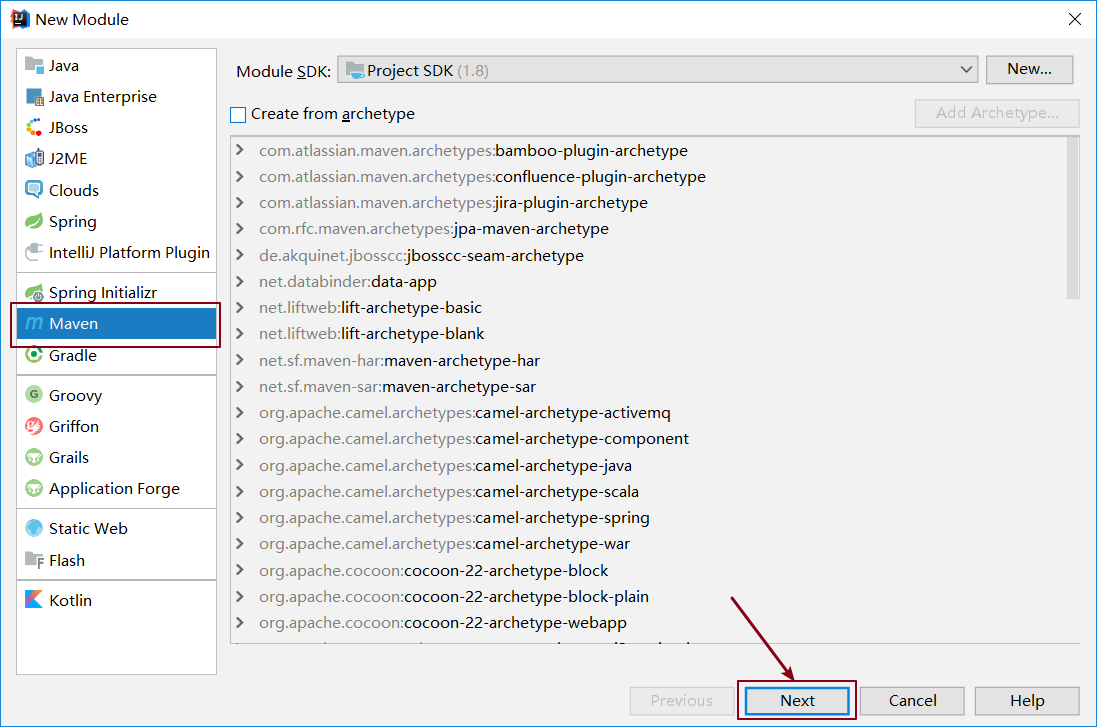
### 1.2.2 步骤

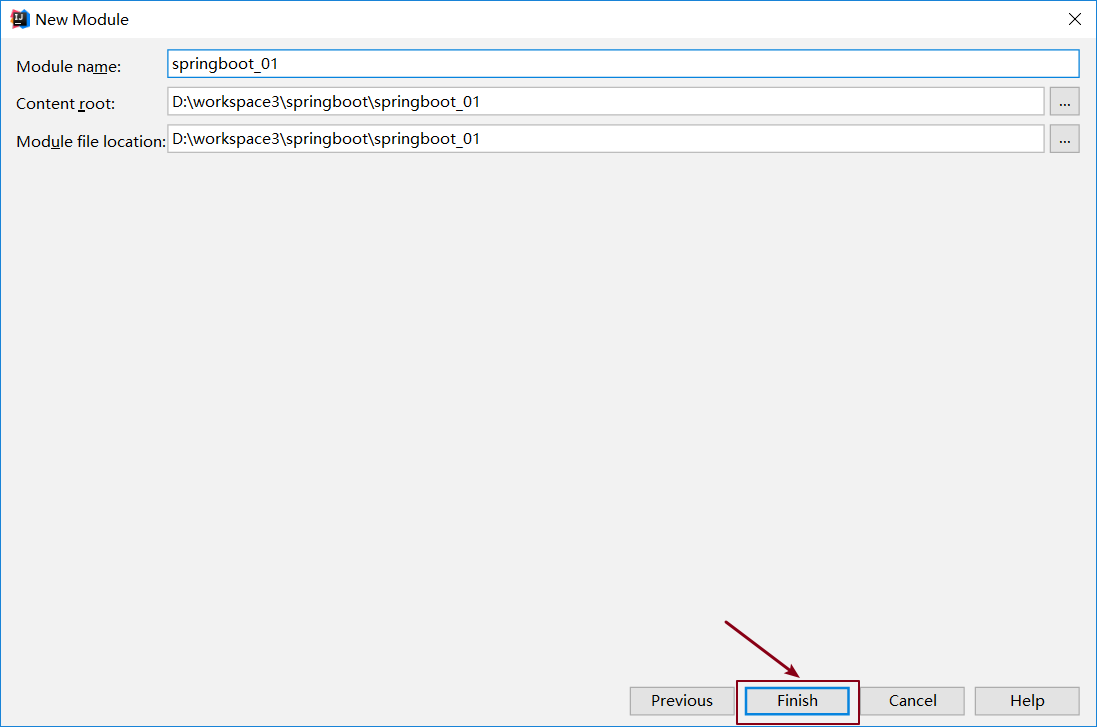
1. 创建Maven工程
2. 添加依赖(springboot父工程依赖 , web启动器依赖)
3. 编写启动引导类(springboot项目运行的入口)
4. 编写处理器Controller
5. 启动项目

### 1.2.3 实现

创建项目: springboot\_01

#### 1.2.3.1 创建工程





#### 1.2.3.2 添加依赖

SpringBoot可以帮我们方便的管理项目依赖 , 在Spring Boot提供了一个名为\*\***spring-boot-starter-parent**\*\*的工程，里面已经对各种常用依赖的版本进行了管理，我们的项目需要以这个项目为父工程，这样我们就不用操心依赖的版本问题了，**需要什么依赖，直接引入坐标(不需要添加版本)即可**！

1. **添加父工程坐标**

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>**spring-boot-starter-parent**</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  </**parent**> |

1. **添加web启动器**

为了让Spring Boot帮我们完成各种自动配置，我们必须引入Spring Boot提供的**自动配置依赖**，我们称为**启动器**。因为我们是web项目，这里我们引入web启动器，在 pom.xml 文件中加入如下依赖：

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  </**dependencies**> |

需要注意的是，我们并没有在这里指定版本信息 , 当我们添加好启动器之后我们发现项目中已经依赖了大量的Jar包

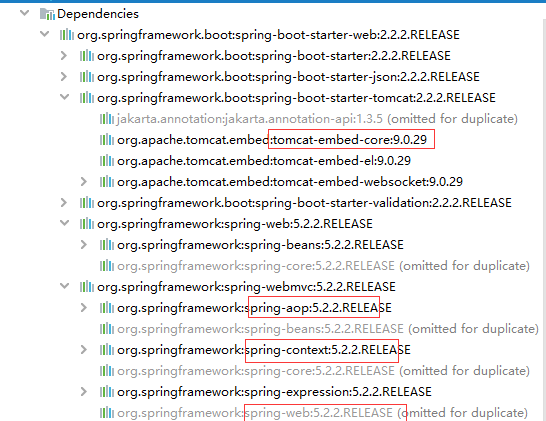
1. **配置JDK版本(可选，默认1.8)**

|  |
| --- |
| <**properties**>  <**java.version**>1.8</**java.version**>  </**properties**> |

思考: 为什么我们这里仅仅配置了这么一个变量 , 项目的JDK版本就会改变呢 ?

因为jdk插件已经在父工程中定义好了 , 默认会读取${java.version}变量值

查看依赖关系，发现一个>spring-boot-starter-web背后已经加载了Spring、SpringMVC的很多依赖，甚至包含了一个内嵌的Tomcat。



#### 1.2.3.3 编写controller

|  |
| --- |
| **package** com.atguigu.controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** **HelloController** {  @RequestMapping("/hello")  **public** String **sayHello**(){  **return** "hello spring boot!!" ;  }  } |

#### 1.2.3.4 创建启动类

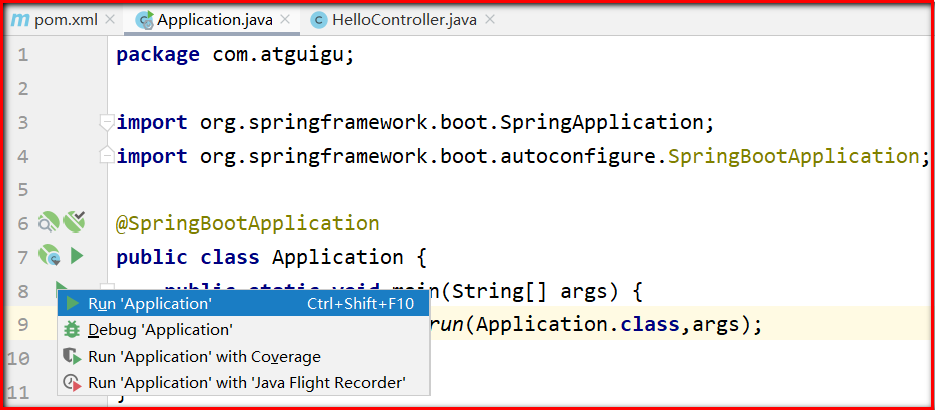
Spring Boot项目通过main函数即可启动，我们需要创建一个启动类：

|  |
| --- |
| **package** com.atguigu;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(Application.class,args);  }  } |

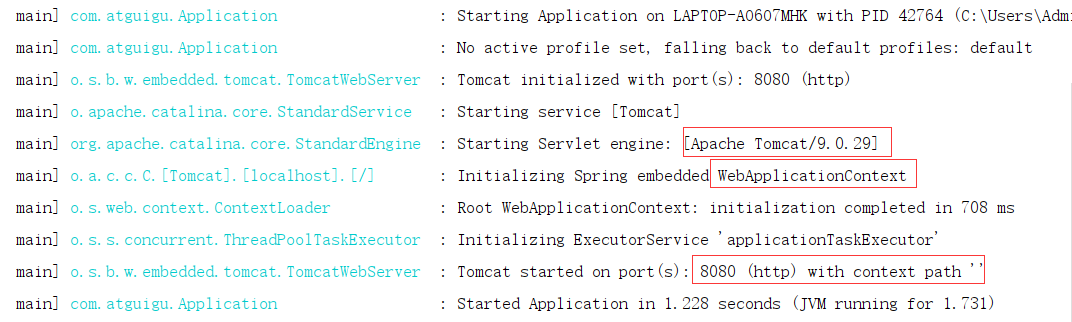
* SpringBoot启动类是SpringBoot项目执行的入口，含有main方法
* main方法中只写一条语句即可：SpringApplication.run(MyApplication.class);
* 必须给启动类使用@SpringBootApplication注解，说明该类是SpringBoot启动类
* 启动类在启动时会做注解扫描(@Controller、@Service、@Repository......)，扫描位置为同包或者子包下的注解，所以要注意启动类的位置，比如com.spring。

#### 1.2.3.5 启动测试

运行启动类的main方法 :



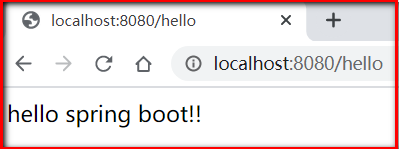
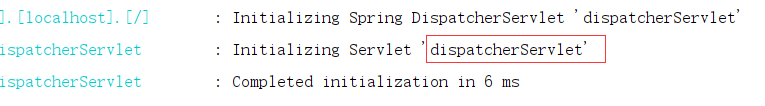
控制台会输出如下信息 :



通过输出的日志我们知道了以下信息 :

* 内嵌了Tomcat9.0.29，端口8080，上下文路径为””
* 加载了WebApplicationContext，这是Spring中针对Web环境的ApplicationContext
* 访问controller时还会加载dispatcherServlet，是SpringMVC核心类

打开浏览器,访问：<http://localhost:8080/hello>

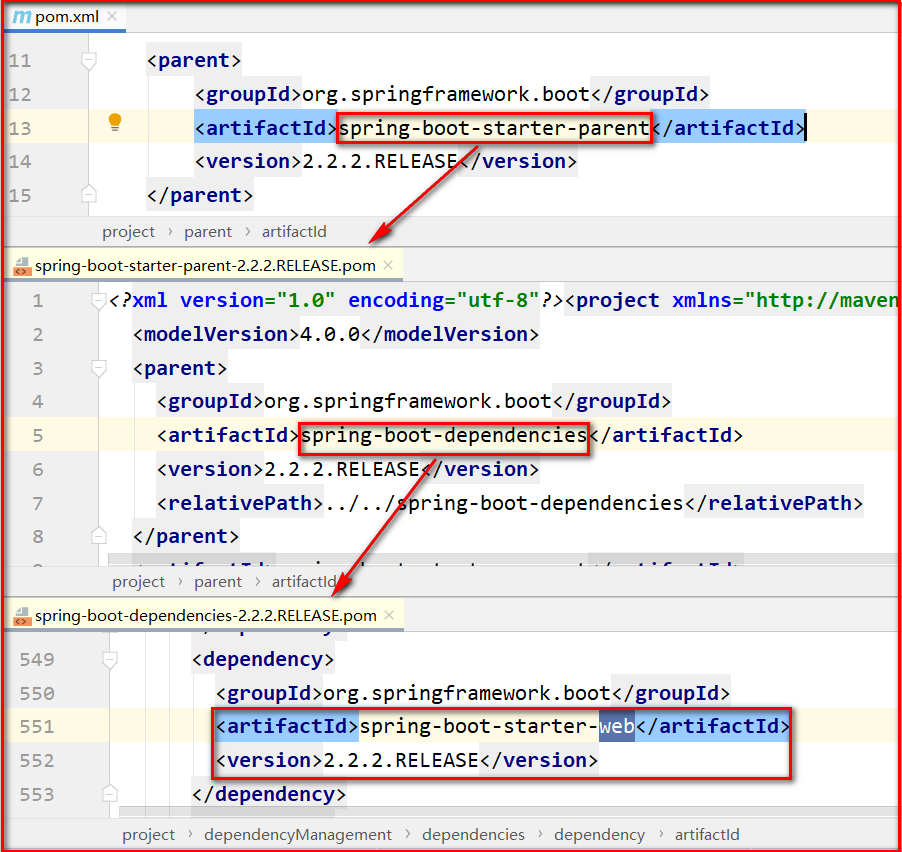
#### 1.2.3.6 SpringBoot简单在哪里了

* 没有web.xml
* 没有spring.xml
* 没有springmvc.xml
* 没有使用外部Tomcat或maven-tomcat插件，不需要安装和部署
* 有pom.xml，但是使用了启动器，dependency少多了

#### 1.2.3.7 常见问题

**问题1：为什么我们在添加启动器的时候不需要在启动器的坐标中指定版本？**

答案：因为我们指定了项目的父工程，在spring-boot-starter-parent中已经通过Maven的版本锁定了Jar包的版本，所以就不需要再指定了。



**问题2：在公司中可能会出现必须继承某个项目，如果Spring Boot用了继承就不能继承别的项目了。怎么办呢？**

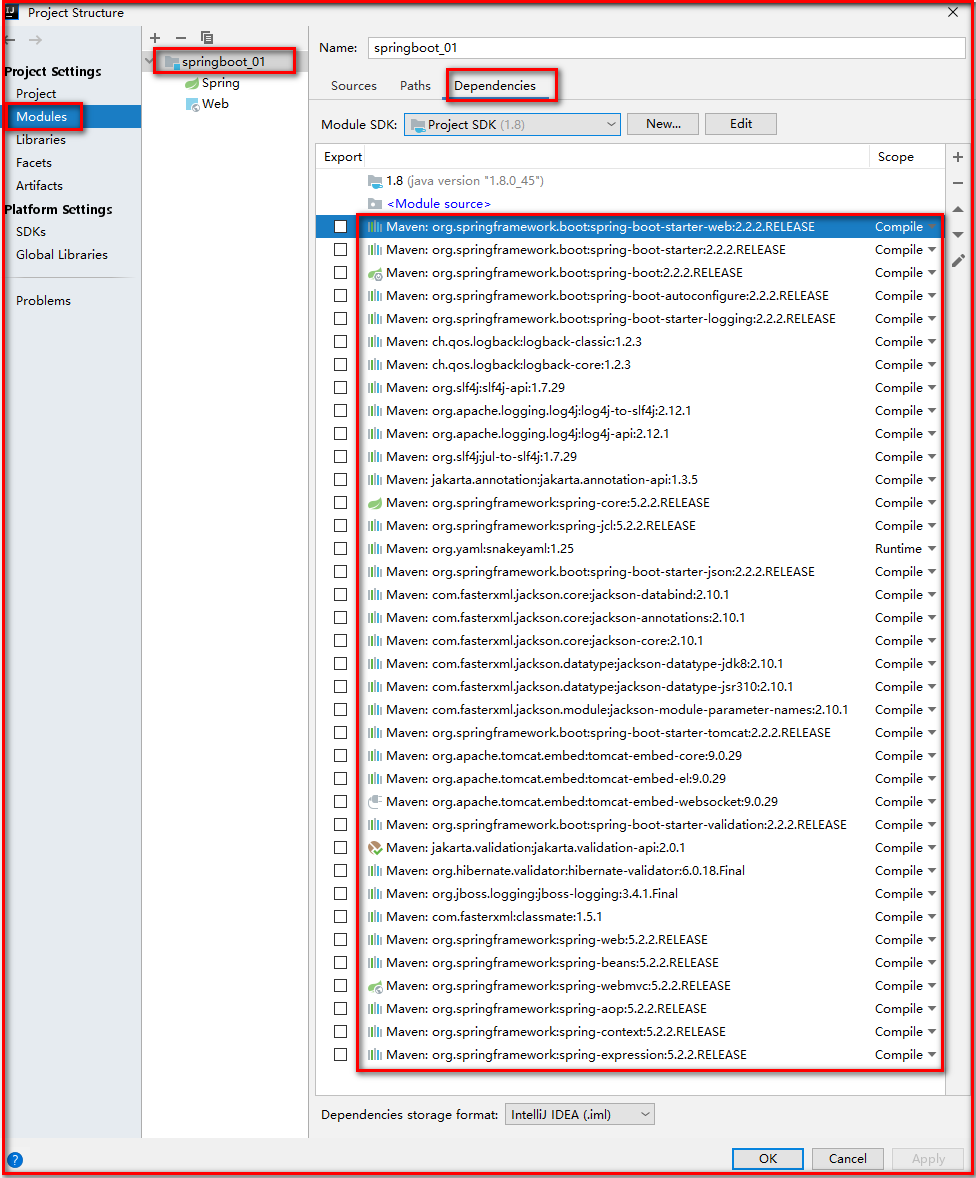
Spring Boot还提供了依赖的方式。需要先注释掉<**parent**>标签。

|  |
| --- |
| <**dependencyManagement**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-dependencies</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  <**type**>pom</**type**>  <**scope**>import</**scope**>  </**dependency**>  </**dependencies**> </**dependencyManagement**> |

**问题3：为什么我们就添加一个启动器依赖，项目就可以运行起来了，运行项目所需要的Jar包从何而来？**

答案：因为我们添加了这个启动器的依赖，它已经把自己运行所需要的必要包集成在这个启动器中，通过Maven的依赖传递性，将这些包都依赖到咱们的项目里了。

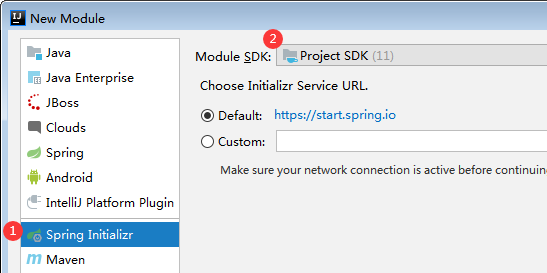
点击项目右键 open module settings



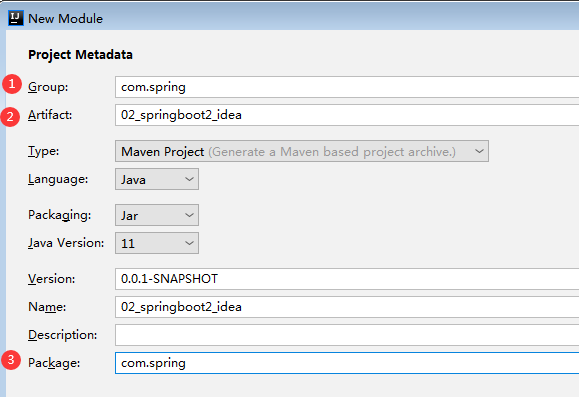
## 1.3 Spring Boot 入门2 – 使用spring initializr创建项目

在idea中，提供了创建SpringBoot项目的向导Spring Initializr，可以简化创建项目的操作。具体步骤如下：

#### 1.3.1 new Module---->Spring Initializr

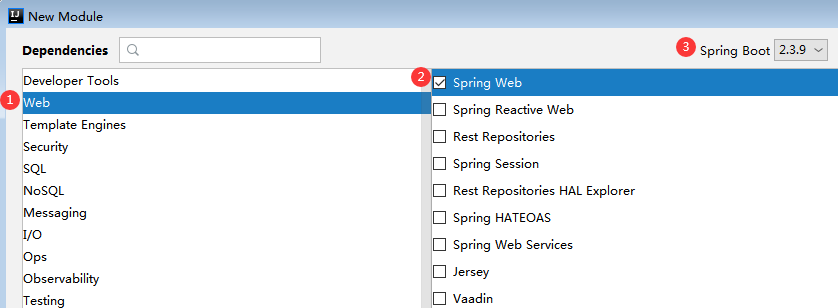


#### 1.3.2 添加项目的POM坐标信息



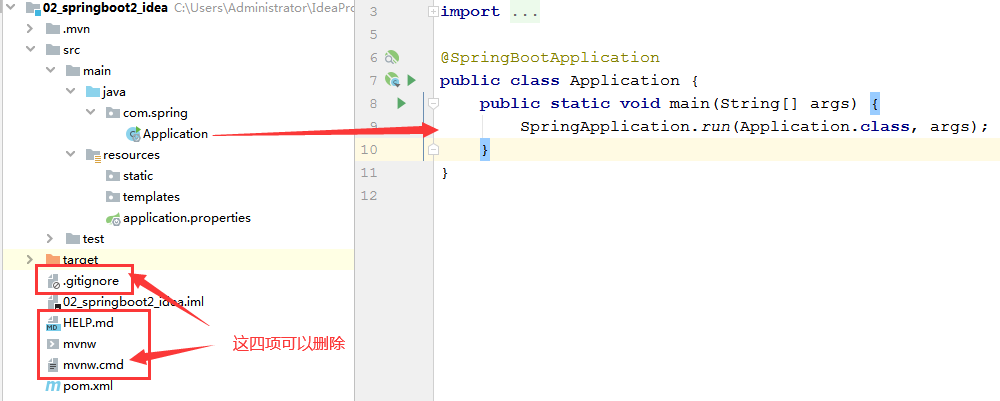
#### 1.3.3 选择要整合的技术

整合SpringMVC，需在Web大项中选择Spring Web小项。同时选择SpringBoot的版本。



#### 1.3.4 执行向导后结果如下

* pom.xml已经自动生成
* 启动类已经自动生成
* 项目中自动生成了一些额外项，可以进行删除
* resources中自动生成了三个目录
  + static：静态资源，比如HTML、CSS、JS、IMAGES
  + templates: 模板资源，主要是动态网页资源，比如freemarker、thymeleaf等，不建议放JSP
  + application.properties:配置文件



#### 1.3.5 准备资源和修改配置

* 添加一个HTML文件：static/html/index.html
* 添加一张图片：static/images/img1.jpg
* 在application.properties中修改Tomcat端口：server.port=8888
* 在application.properties中修改项目上下文路径：server.servlet.context-path=/test

#### 1.3.6 启动服务器并进行测试



<http://127.0.0.1:8888/test/html/index.html>

<http://127.0.0.1:8888/test/images/img1.jpg>

#### 1.3.7 修改启动类代码

|  |
| --- |
| @SpringBootApplication **public class** Application {  **public static void** main(String[] args) {  //获取容器的引用  ConfigurableApplicationContext applicationContext  = SpringApplication.run(Application.**class**,args);  //获取并输出容器创建的Bean的数量  **int** beanDefinitionCount = applicationContext.getBeanDefinitionCount();  System.**out**.println(beanDefinitionCount);  //获取并输出容器创建的每个Bean的名称  String[] beanDefinitionNames = applicationContext.getBeanDefinitionNames();  **for** (String beanDefinitionName : beanDefinitionNames) {  System.**out**.println(beanDefinitionName);  }  } } |

运行代码，发现SpringBoot底层居然创建了132个Bean之多，其中有我们熟悉的和SpringMVC有关的Bean。通过这些，大家也直观的感觉到SpringBoot的简单并强大。

* dispatcherServlet xxxHandlerMapping xxxHandlerAdapter
* viewResolver multipartResolver
* characterEncodingFilter JacksonXXXX

# 二、Spring Boot核心功能(应用)

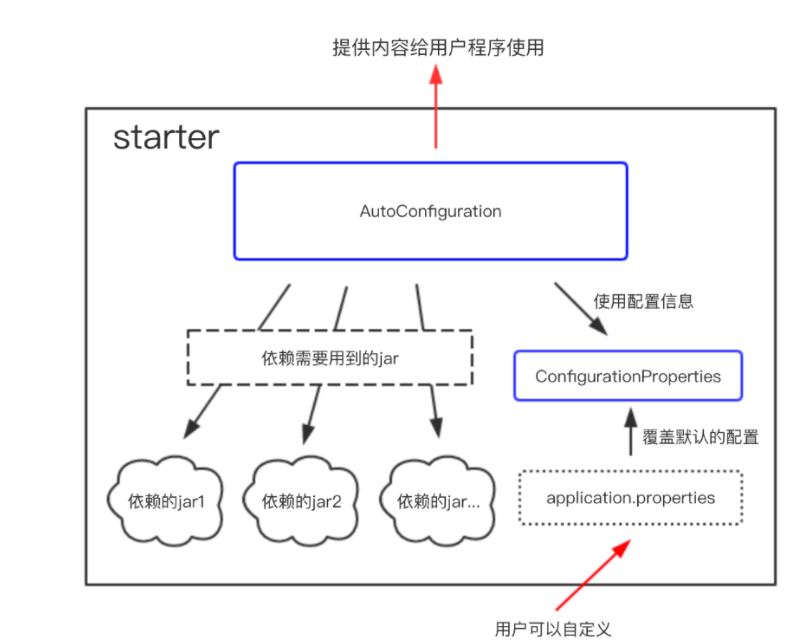
## 2.1启动器Starter

在我们的日常开发工作中，经常会有一些独立于业务之外的配置模块，我们经常将其放到一个特定的包下，然后如果另一个工程需要复用这块功能的时候，需要将代码硬拷贝到另一个工程，重新集成一遍，麻烦至极。如果我们将这些可独立于业务代码之外的功配置模块封装成一个个starter，复用的时候只需要将其在pom中引用依赖即可，SpringBoot为我们完成自动装配。

SpringBoot中的starter是一种非常重要的机制，能够抛弃以前繁杂的配置，将其统一集成进starter，应用者只需要在maven中引入starter依赖，SpringBoot就能自动扫描到要加载的信息并启动相应的默认配置。starter让我们摆脱了各种依赖库的处理，需要配置各种信息的困扰。Spring Boot实际上是利用Spring Framework 4 自动配置特性完成。

发展到现在，Spring Boot已经具有很大的生态圈，各种主流技术已经都提供了Spring Boot的启动器。所有这些依赖模块都遵循着约定成俗的默认配置，并允许我们调整这些配置，即遵循“约定大于配置”的理念。

Spring Boot的启动器实际上就是一个依赖。这个依赖中包含了整个这个技术的相关jar包，还包含了这个技术的自动配置，以前绝大多数XML配置都不需要配置了。当然了，启动器中自动配置无法实现所有内容的自动配置，在使用Spring Boot时还需要进行少量的配置（这个配置不是在xml中了，而是在properties或yml中即可）。



如果是Spring自己封装的启动器的artifact id名字满足：spring-boot-starter-xxxx，如果是第三方公司提供的自定义启动器满足：xxxx-spring-boot-starter。

|  |  |
| --- | --- |
| **启动器** | **作用** |
| spring-boot-starter | Spring Boot的核心启动器，包含了自动配置、日志和YAML |
| spring-boot-starter-web | 支持全栈式Web开发，包括Tomcat和spring-webmvc |
| spring-boot-starter-test | 支持常规的测试依赖，包括JUnit、Hamcrest、Mockito以及spring-test模块。 |
| spring-boot-starter-aop | 支持面向方面的编程即AOP，包括spring-aop和AspectJ |
| spring-boot-starter-freemarker | 支持FreeMarker模板引擎 |
| spring-boot-starter-thymeleaf | 支持Thymeleaf模板引擎 |
| spring-boot-starter-data-elasticsearch | 支持ElasticSearch搜索和分析引擎 |
| spring-boot-starter-data-solr | 支持Apache Solr搜索平台 |
| spring-boot-starter-data-mongodb | 支持MongoDB数据 |
| spring-boot-starter-jdbc | 支持JDBC数据库 |
| spring-boot-starter-redis | 支持Redis键值存储数据库， |
| spring-boot-starter-security | 支持spring-security |
| spring-boot-starter-jetty | 引入了Jetty HTTP引擎（用于替换Tomcat） |
| spring-boot-starter-logging | 引入了Spring Boot默认的日志框架Logback |
| mybatis-spring-boot-starter | 支持MyBatis |
| druid-spring-boot-starter | 支持druid数据源 |

SpringBoot所有场景启动器的网址：<https://docs.spring.io/spring-boot/docs/2.3.9.RELEASE/reference/html/using-spring-boot.html#using-boot-starter>

以后每次使用Spring Boot整合其他技术时首先需要考虑导入启动器。也就是说，SpringBoot要整合某一个技术，比如SpringMVC、MyBatis、Thymeleaf、第一步怎么办，先在pom.xml中加入对应的starter依赖。

关于如何自定义Starter，属于SpringBoot高级内容，我们在本章内容后面进行扩展。

## 2.2 项目结构

SpringBoot的项目结构已经有了初步认识，这里进行更详细说明。

-- 项目名

--src

--main

--java java代码

--resources

--public 公共资源。所有共享的内容。对外公开的内容。

--static 静态资源。图片、js、css。不会被服务器解析。

--js

-- vue.js

注意:该目录是SpringBoot可以直接识别的目录，会将其中的

静态资源编译到web项目中，并放到tomcat中使用。静态资源的

访问路径中无需声明static 例如:localhost:8080/a.png

--templates thymeleaf、freemarker等模板页面目录。

-- application.properties 配置文件

-- application.yml 配置文件

--test 单元测试目录

**上面是SpringBoot的核心项目结构，某些情况还可以加入其它目录，**

* 比如整合MyBatis时可以添加resources/mapper/XXXXMapper.xml
* 使用日志时可以添加日志配置文件resources/logback.xml
* 后面也会讲解application.properties和application.yml可以放在多个不同的位置

|  |
| --- |
| **扩展：静态资源的位置**   * 默认的静态资源的位置有四个，具体情况源码。 * 优先级顺序/META-INF/resources>resources>static>public * 注意：这个resources指的是和static并列的，即resources/resources。 |

## 2.3 配置文件详解

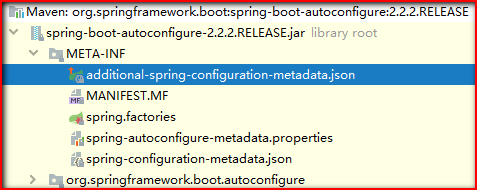
springboot支持二种类型的配置文件

* properties属性配置文件
* yaml/yml配置文件

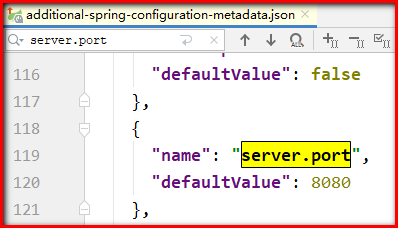
配置文件必须放置在项目的类加载目录下, 并且名字必须是application

springboot项目在运行的时候会自动加载这些配置文件

同级目录下打开：spring-configuration-metadata.json



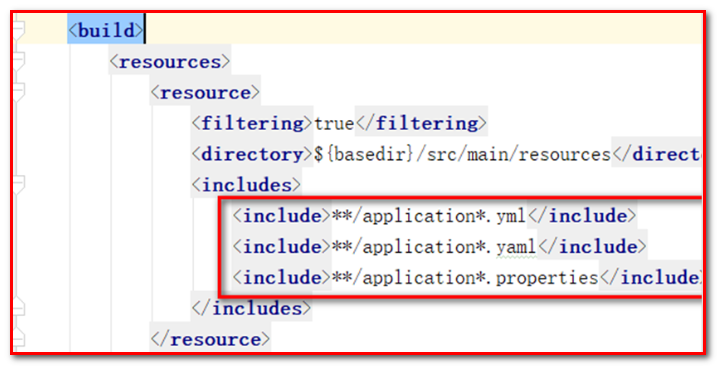
搜素：server.port



为什么可以在resources下创建application.properties文件呢？我们查看springboot的启动依赖：



点击spring-boot-starter-parent



加载顺序从上到下，最下面的会将之前已有的相同属性进行覆盖，比如 application.yml 和 application.properties中都配置了服务的启动端口，最终是 application.properties中的起左右。

### 2.3.1 属性配置文件



在 resource 文件夹下面新建 application.properties 配置文件

|  |
| --- |
| **spring**.jdbc.datasource.driverClassName=com.mysql.jdbc.driver  **spring**.jdbc.datasource.url=jdbc:mysql:///springboot  # spring.jdbc.datasource.url=jdbc:mysql://127.0.0.1:3306/springboot  **spring**.jdbc.datasource.username=root  **spring**.jdbc.datasource.password=root |

如何获取属性文件内容呢

**方式1：使用@Value注解一个个注入属性文件内容**

新建 properties 包，创建类 DataSourceProperties

|  |
| --- |
| **package** com.atguigu.properties;  **import** org.springframework.beans.factory.annotation.Value;  **import** org.springframework.stereotype.Component;  @Component  **public** **class** **DataSourceProperties** {  @Value("${spring.jdbc.datasource.driverClassName}")  **private** String driverClassName;  @Value("${spring.jdbc.datasource.url}")  **private** String url;  @Value("${spring.jdbc.datasource.username}")  **private** String username;  @Value("${spring.jdbc.datasource.password}")  **private** String password;  // 生成get set 和 toString方法  } |

这种注入方式，如果属性特别多，一个一个注入太麻烦啦o(╥﹏╥)o

在 controller 添加 sayHello2 方法

|  |
| --- |
| **package** com.atguigu.controller;  **import** com.atguigu.properties.DataSourceProperties;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** **HelloController** {  @Autowired  **private** DataSourceProperties dataSourceProperties ;    @RequestMapping(path = "/hello")  **public** String **sayHello**() {  System.out.println(dataSourceProperties);  **return** "Hello Spring Boot ! " ;  }  } |

请求地址：http://localhost:8080/hello

控制台打印：

|  |
| --- |
| DataSourceProperties{driverClassName='com.mysql.jdbc.Driver', url='jdbc:mysql:///springboot\_01', username='root', password='root'} |

**方式2：使用@ConfigurationProperties批量注入**

这种注入方式，属性再多，只要按照规则就可以一次性自动注入。方便的很哦\(^o^)/~

|  |
| --- |
| **package** com.atguigu.properties;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  @Component  @ConfigurationProperties(prefix = "spring.jdbc.datasource")  **public** **class** **DataSourceProperties2** {  **private** String driverClassName;  **private** String url;  **private** String username;  **private** String password;    // 省略getter和setter.....  } |

* 在类上通过@ConfigurationProperties注解声明该类要读取属性配置
* prefix="spring.jdbc.datasource" 读取属性文件中前缀为spring.jdbc.datasource的值。前缀和属性名称和配置文件中的key必须要保持一致才可以注入成功

开启@ConfigurationProperties注解使用

|  |
| --- |
| @Controller  @EnableConfigurationProperties(DataSourceProperties2.class)  **public** **class** **HelloController** {    @Autowired  **private** DataSourceProperties2 dataSourceProperties2 ;  @RequestMapping(path = "/hello")  @ResponseBody  **public** String **sayHello**(){  System.out.println(dataSourceProperties2);  **return** "hello spring boot";  }  } |

使用@EnableConfigurationProperties(DataSourceProperties2.class)，开启DataSourceProperties2身上的@ConfigurationProperties注解 , 他就会生效了, 就会帮助我们注入数据了。

注意：如果DataSourceProperties2同时使用了@ConfigurationProperties和@Component注解，则无需使用@EnableConfigurationProperties注解

请求地址：http://localhost:8080/hello

打印结果：

|  |
| --- |
| DataSourceProperties2{driverClassName='com.mysql.jdbc.Driver', url='jdbc:mysql:///springboot\_01', username='root', password='root'} |

报错提示，请在pom文件添加配置信息



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

### 2.3.2 YAML配置文件

YAML（YAML Ain't Markup Language），扩展名可以是yaml或者yml。

* YAML A Markup Language：是一个标记语言
* YAML isn't Markup Language：不是一个标记语言

语法要求:

* YAML配置文件的扩展名是yaml 或 yml
* 数据结构用树形结构呈现
* 配置项之间如果有层级关系的话需要分行并缩进
* 同一级别缩进必须相同，建议缩进2个空格
* 键值结构里面的key/value对用冒号 ” : ” 来分隔，并且” : ”之后空一格再写配置项值。
* 大小写敏感
* # 表示注释，从这个字符一直到行尾，都会被解析器忽略

YAML 支持的数据结构有三种

* 字面量：单个的、不可再分的值类型可为字符串，布尔，整数，浮点，时间日期。 字符串默认不用加上单引号或者双引号。

|  |
| --- |
| **empno**: 7839 **ename**: SMITH **hiredate**: 1982-12-13 **sal**: 3000.0 **ismgr**: false |

* 对象：键值对的集合。比如如上数据库的四个连接参数集合。

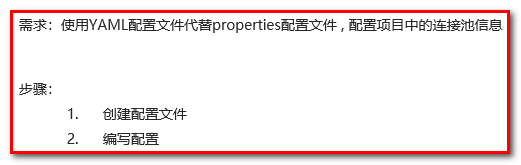
|  |
| --- |
| *#缩进样式* **emp**:  **empno**: 7839  **ename**: SMITH  **hiredate**: 1982-12-13  **sal**: 3000.0  **ismgr**: false *#行内样式* **emp2**: {**empno**: 7839,**ename**: SMITH,**hiredate**: 1982-12-13,**sal**: 3000.0,**ismgr**: false} |

* 数组：一组按次序排列的值，又称为序列（sequence） / 列表（list）

|  |
| --- |
| *#缩进样式* **city**:  - beijing  - tianjin  - shanghai  - chongqing  *# 行内方式*  **hobbyArr**: [game,program,travel] |

**yaml与properties配置文件除了展示形式不相同以外，其它功能和作用都是一样的**

**如果properties和yml文件都存在，不存在spring.profiles.active设置，如果有重叠属性，默认以properties优先。**



在 resource 文件夹下面新建 application.yml 配置文件，修改 application.properties 配置文件名字为 application.properties.bak

|  |
| --- |
| spring:  jdbc:  datasource:  driverClassName: com.mysql.jdbc.Driver  url: jdbc:mysql:///springboot\_01  username: root  password: root |

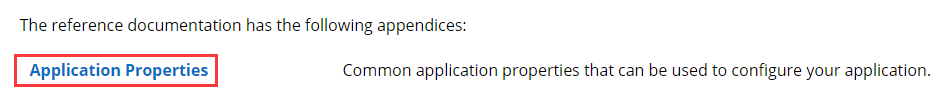
如何获取yml文件内容呢？同样可以通过@Value实现单个注入，使用@ConfigurationProperties来实现批量注入

运行项目，重新请求 http://localhost:8080/hello

问题：像**server.port**=**8888，server.servlet.context-path**=**/test**这样的配置项**，**如何知道有它，还有哪些配置项。

官网参考文档的附录中有专门提供。

<https://docs.spring.io/spring-boot/docs/2.4.3/reference/html/>



点击进入即可。



### 2.3.3 多环境profile切换配置

我们刚刚说过在Spring Boot项目中配置文件的名称只能是\*\*application\*\* , 如果我们把所有的配置全都写在一个配置文件中如果配置项比较多, 配置文件就会显得比较复杂和臃肿 ! 不利于后期的项目维护和开发

例如下面几个场景 :

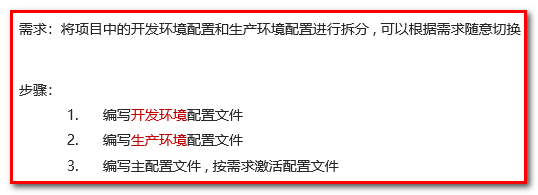
1.因为开发环境的变化, 我们需要修改配置文件中某一个配置项的值（比如之前是mysql数据库，切换成oracle数据库）

2.项目开发完成需要上线了 , 需要把一些环境修改成正式环境（**开发**，**测试**，**上线**，多环境切换）

**解决方案 :使用profiles拆分配置**

spring boot项目中允许使用多个YAML配置文件。

这些文件名称必须为application-\*\*\*.yml，并且在application.yml中激活。



创建**application-dev.yml**文件如下：

# 配置数据库连接池信息 ，开发环境

|  |
| --- |
| spring:  jdbc:  datasource:  driverClassName: com.mysql.jdbc.Driver  url: jdbc:mysql:///springboot  username: root  password: root |

创建**application-pro.yml**文件如下：

# 配置数据库连接池信息，上线环境

|  |
| --- |
| spring:  jdbc:  datasource:  driverClassName: com.mysql.jdbc.Driver  url: jdbc:mysql:///business  username: business  password: business |

在 **application.yml** 文件中添加如下配置：

# 激活配置文件

|  |
| --- |
| spring:  profiles:  active: **dev** |

直接运行项目：http://localhost:8080/hello

打印结果：

|  |
| --- |
| DataSourceProperties{driverClassName='com.mysql.jdbc.Driver', url='jdbc:mysql:///springboot', username='root', password='root'} |

修改 application.yml 配置文件：

# 激活配置文件

|  |
| --- |
| spring:  profiles:  active: **pro** |

打印结果：

|  |
| --- |
| DataSourceProperties{driverClassName='com.mysql.jdbc.Driver', url='jdbc:mysql:///business', username='business', password='business'} |

**注意 :**

**如果properties和yml文件都存在，不存在spring.profiles.active设置，如果有重叠属性，默认以properties优先。**

**如果设置了spring.profiles.active，并且有重叠属性，以active设置优先。**

**可以在两种文件中分别增加server.port属性指定不同的端口，启动项目查看控制台端口号进行测试。**

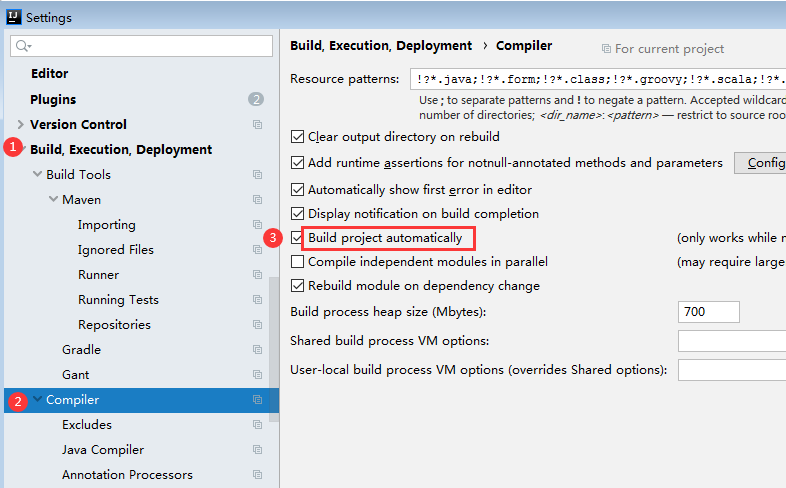
## 2.4使用开发者工具包实现热部署

SpringBoot提供了spring-boot-devtools，可以实现热部署。当前每次classpath下的文件（包括类文件、属性文件、页面等）时，会重新启动应用，提高开发者体验。

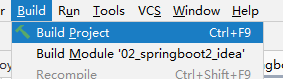
1. 添加依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**optional**>true</**optional**> </**dependency**> |

1. 设置自动构建项目

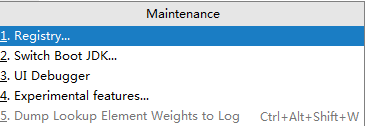


如果不设置此项，需要每次手动构建项目，快捷键为:ctrl+F9。

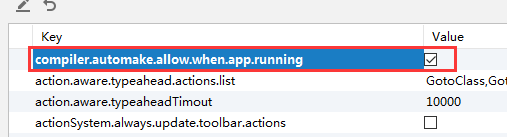


1. 注册项目运行是自动编译

Ctrl+Shift+Alt+/ 点击弹出框中Registry...



选中：compiler.automake.allow.when.app.running。



|  |
| --- |
| **扩展：开发者工具devtools的原理**   * spring-boot-devtools使用了两个类加载器来实现重启（restart）机制：base类加载器（base ClassLoader）, restart类加载器（restart ClassLoader）。 * base ClassLoader：用于加载不会改变的jar（eg.第三方依赖的jar）。restart ClassLoader：用于加载我们正在开发的jar（eg.整个项目里我们自己编写的类）。当应用重启后，原先的restart ClassLoader被丢掉、重新new一个restart ClassLoader来加载这些修改过的东西，而base ClassLoader却不需要动一下。这就是devtools重启速度快的原因。 * 如果发现重新启动对于应用程序来说不够快，或者遇到类加载问题，则可以考虑安装热部署插件，比如JRebel等。当程序员在开发环境中对任何一个类或者资源作出修改的时候，这个变化会直接反应在部署好的应用程序上，从而跳过了构建和部署的过程，可以省去大量的部署用的时间。 |

## 2.5 项目打包和运行

项目开发完毕后，需要打包部署运行。对于SpringBoot项目，内嵌Tomcat，可以直接打成jar包而不是war包，直接运行（使用内嵌Tomcat）而不需要部署到特定Tomcat中。

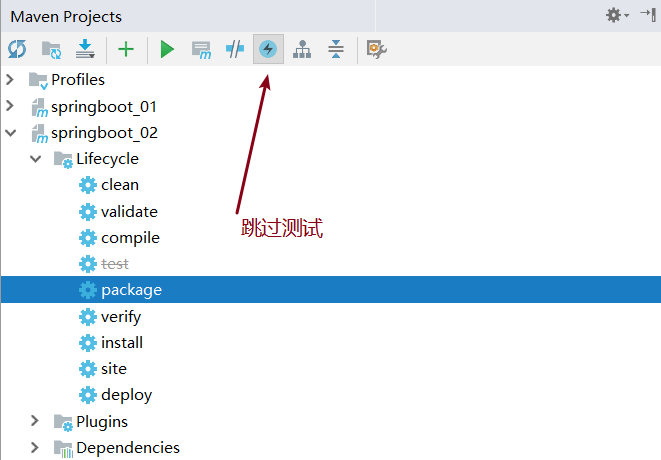
1. 在pom.xml中配置Spring Boot项目的maven插件

|  |
| --- |
| <**build**>  <**plugins**>  <!-- 打jar包时如果不配置该插件，打出来的jar包没有清单文件 -->  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**> |

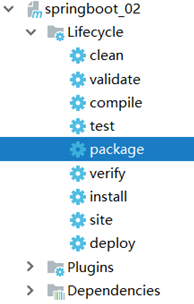
如果是采用<dependencyManagement>依赖SpringBoot而不是<parent>方式继承SpringBoot，需要采用下面的方式

|  |
| --- |
| <**plugins**>  *<!-- 打jar包时如果不配置该插件，打出来的jar包没有清单文件 -->* <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  <**executions**>  <**execution**>  <**goals**>  <**goal**>repackage</**goal**>  </**goals**>  </**execution**>  </**executions**>  </**plugin**> </**plugins**> |

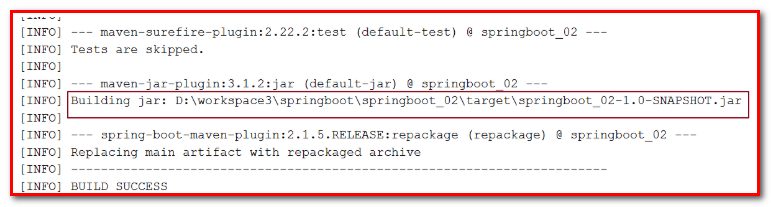
1. 打包之前我们需要跳过测试 , 如果不跳过测试那么我们编写的测试类都会被maven自动执行, 可能会出现错误,导致打包不成功



1. 运行maven的打包命令 : package



1. 执行之后可以在控制台看到打包的日志信息, 其中有生成的包的位置



打开指定目录就可以发现有一个jar包存在 , 仔细观察其实我们会发现 , 在target目录下其实会存在二个jar包 , 一个是springboot\_02-1.0-SNAPSHOT.jar一个是springboot\_02-1.0-SNAPSHOT.jar.original , 那么这两个jar包有什么区别呢?

我们如果是普通项目打包那么就只会得到一个jar包 , 这个jar包中不包含项目的一些依赖jar包

但现在是一个Spring Boot项目 , 我们希望打完的包能够直接运行, 所以项目中就必须包含他的依赖jar包 , 我们之前在pom.xml中配置一个Spring Boot的maven插件可以在普通包的基础上将我们项目的一些运行及依赖信息打进jar包里面 , 打完包之后将原来的普通包改名为xxx.jar.original , 新打的包为xxx.jar .

简单总结一下 :

* .jar.original 是普通jar包，不包含依赖
* .jar 是可执行jar包，包含了pom中的所有依赖，可以直接用java -jar 命令执行
* 如果是部署就用.jar , 如果是给别的项目用，就要给.jar.original这个包

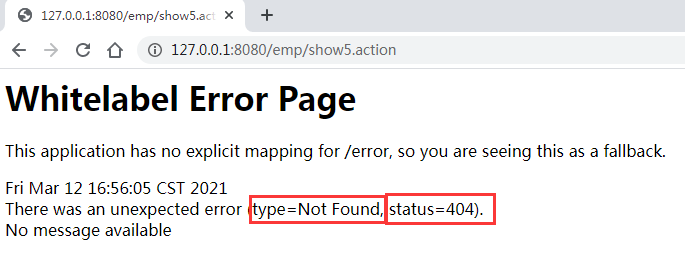
1. 项目运行:

打开命令行运行打出来的包；使用命令：java –jar 包全名

|  |
| --- |
| **java** -jar springboot\_02-1.0-SNAPSHOT.jar |

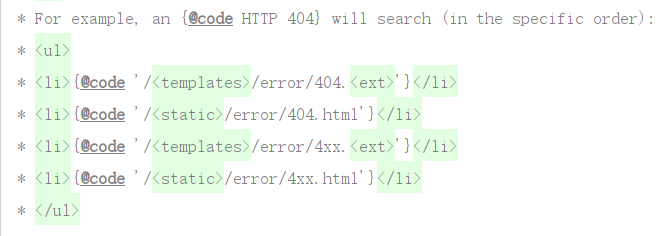
## 2.6 异常显示页面

默认情况，Spring Boot项目错误页面如下。



当项目实际上线，如果给用户显示这个页面就不是很友好。当系统出现异常时应该给用户显示更加友好的错误页面。

查看DefaultErrorViewResolver源码注释，可以看到查询错误页面的顺序和位置。



#### 设置具体的状态码页面

在static/下新建error文件夹，在error中新建：状态码.html的页面。例如当出现500时显示的页面为500.html

#### 使用x进行模糊匹配

在static/下新建error文件夹，

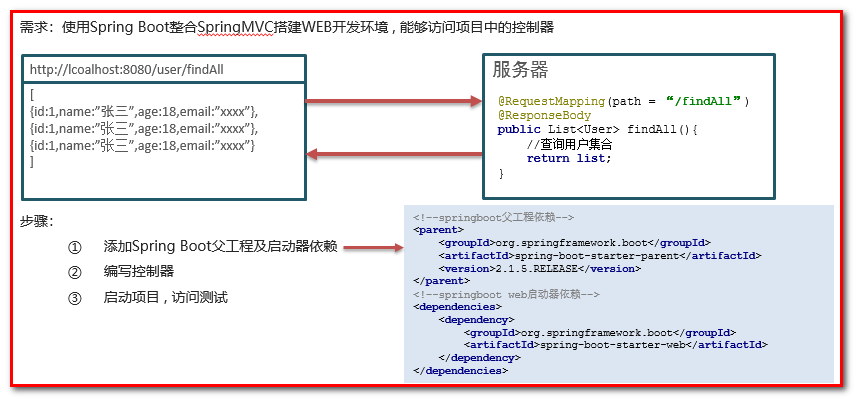
当出现5开头状态码的错误时，显示页面可以命名为5xx.html

当出现50开头状态码的错误时，显示页面可以命名为50x.html

#### 统一错误显示页面

在static下新建error.html。如果项目中不存在具体状态码的页面或没有使用x成功匹配的页面时，显示error.html作为错误显示页面。

# 三. Spring Boot整合其他技术



## 3.1 SpringBoot整合MVC

创建项目 springboot\_02\_mvc

### 3.1.1 起步依赖

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  </**parent**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  **<optional>**true<**/optional>**  </**dependency**>  </**dependencies**> |

① 新建入口程序类 Application

|  |
| --- |
| **package** com.atguigu;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(Application.class,args);  }  } |

② 新建 javabean

|  |
| --- |
| **package** com.atguigu.pojo;  **public** **class** **User** {  **private** String username ;  **private** String password ;  **private** Integer age ;  **private** String sex ;  … |

③ 新建 UserController

|  |
| --- |
| **package** com.atguigu.controller;  **import** com.atguigu.pojo.User;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  **import** java.util.ArrayList;  **import** java.util.List;  @Controller  @RequestMapping(path = "/user")  **public** **class** **UserController** {  @RequestMapping(path = "/findAll")  @ResponseBody  **public** List<User> **findAll**(){  //查询所有  List<User> users = **new** ArrayList<User>();  User user1 = **new** User();  user1.setUsername("杨过");  user1.setPassword("123456");  user1.setAge(18);  user1.setSex("男");  User user2 = **new** User();  user2.setUsername("杨过");  user2.setPassword("123456");  user2.setAge(18);  user2.setSex("男");  User user3 = **new** User();  user3.setUsername("杨过");  user3.setPassword("123456");  user3.setAge(18);  user3.setSex("男");  users.add(user1);  users.add(user2);  users.add(user3);  **return** users ;  }  } |

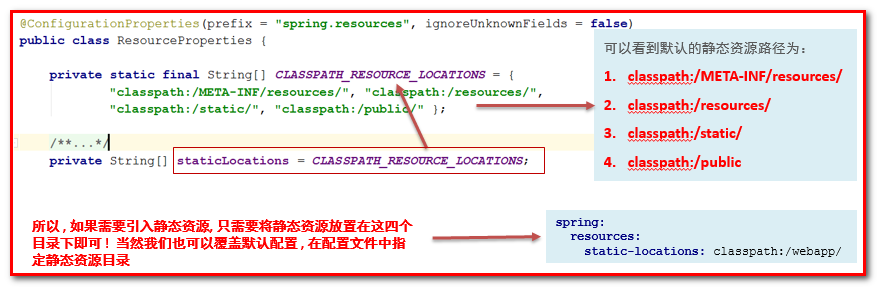
运行程序

### 3.1.2 静态资源目录

在WEB开发中我们经常需要引入一些静态资源 , 例如 : HTML , CSS , JS , 图片等 , 如果是普通的项目静态资源可以放在项目的webapp目录下

现在使用Spring Boot做开发 , 项目中没有webapp目录 , 我们的项目是一个jar工程，那么就没有webapp，我们的静态资源该放哪里呢？

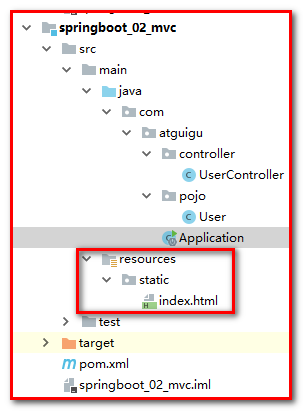
在springboot中有一个叫做ResourceProperties的类，里面就定义了静态资源的默认查找路径：



默认的静态资源路径为：

* classpath:/META-INF/resources/
* classpath:/resources/
* classpath:/static/
* classpath:/public

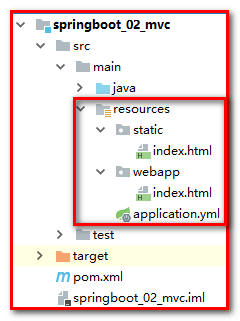
我们只要静态资源放在这些目录中任何一个，SpringMVC都会帮我们处理。 我们习惯会把静态资源放在classpath:/static/ 目录下。在resources目录下static目录下创建index.html文件



打开浏览器输入 : http://localhost:8080/index.html

覆盖路径

如果想要修改默认的静态资源路径, 配置如下 :



新建 application.yml

|  |
| --- |
| spring:  resources:  **static**-locations: classpath:/webapp/ |

请求地址 http://localhost:8080/index.html

### 3.1.3 自定义拦截器

web开发中的拦截器也是我们经常需要使用的组件,可以帮我们完成一些日志记录 , 数据过滤 , 请求过滤等等很多功能，那么在SpringBoot中该如何配置呢？

回顾一下SpringMVC中配置拦截器的步骤 :

1. 编写一个拦截器(实现HandlerInterceptor接口)
2. 注册拦截器(mvc:interceptors)

xml

|  |
| --- |
| <!--配置拦截器-->  <**mvc:interceptors**>  <**mvc:interceptor**>  <!--配置拦截路径-->  <**mvc:mapping** **path**="/user/\*\*"/>  <!--配置不拦截路径:不拦截路径是指从拦截路径中排除-->  <**mvc:exclude-mapping** **path**="/user/sayByby"></**mvc:exclude-mapping**>  <!--配置拦截器bean-->  <**bean** **class**="com.atguigu.interceptor.LogInterceptor2"></**bean**>  </**mvc:interceptor**>  </**mvc:interceptors**> |

因为SpringBoot没有XML配置文件了 , 所以在SpringBoot中使用拦截器的注册拦截器的方式就不太一样了, 需要借助一个WebMvcConfigurer类帮助我们注册拦截器 , 实现拦截器的具体步骤如下 :

1. 编写一个拦截器
2. 通过WebMvcConfigurer注册拦截器

**编写拦截器**

|  |
| --- |
| **package** com.atguigu.interceptor;  **import** org.springframework.stereotype.Component;  **import** org.springframework.web.servlet.HandlerInterceptor;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  @Component  **public** **class** **MyInterceptor** **implements** **HandlerInterceptor** {  @Override  **public** **boolean** **preHandle**(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  System.out.println("MyInterceptor拦截器的preHandle方法执行....");  **return** **false**;  }  @Override  **public** **void** **postHandle**(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  System.out.println("MyInterceptor拦截器的postHandle方法执行....");  }  @Override  **public** **void** **afterCompletion**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  System.out.println("MyInterceptor拦截器的afterCompletion方法执行....");  }  } |

**注册拦截器**

|  |
| --- |
| **package** com.atguigu.config;  **import** com.atguigu.interceptor.MyInterceptor;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  @Configuration  **public** **class** **MvcConfig** **implements** **WebMvcConfigurer** {  @Autowired  **private** MyInterceptor myInterceptor ;  /\*\*  \* /\*\* 拦截当前目录及子目录下的所有路径 /user/\*\* /user/findAll /user/order/findAll  \* /\* 拦截当前目录下的以及子路径 /user/\* /user/findAll  \* @param registry  \*/  @Override  **public** **void** **addInterceptors**(InterceptorRegistry registry) {  registry.addInterceptor(myInterceptor).addPathPatterns("/\*\*");  }  } |

打开浏览器,输入 : <http://localhost:8888/user/findAll>

## 3.2 SpringBoot整合Spring Data JPA

Spring  Data项目是从2010年发展起来的，从创立之初SpringData就想提供一个大家熟悉的、一致的、基于Spring的数据访问编程模型，同时仍然保留底层数据存储的特殊特性。它可以轻松地让开发者使用数据访问技术，包括关系数据库、非关系数据库（NoSQL）和基于云的数据服务。Spring Data包括多个子项目，可以访问 <https://spring.io/projects/spring-data>来查看。不仅对传统的数据库访问技术JDBC、Hibernate、JDO、TopLick、JPA、Mybitas做了很好的支持、扩展、抽象、提供方便的API，还对NoSQL等非关系数据做了很好的支持，包括MongoDB、Redis、Apache Solr等

JPA是Java Persistence API的简称，中文名Java持久层API，是JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到数据库中。 Sun引入新的JPA ORM规范出于两个原因：其一，简化现有Java EE和Java SE应用开发工作；其二，Sun希望整合ORM技术，实现天下归一。Hibernate3.2+、TopLink 10.1.3以及OpenJPA都提供了JPA的实现。

Spring Data JPA可以理解为JPA规范的再次封装抽象，底层还是使用了Hibernate的JPA技术实现，引用JPQL（Java Persistence Query Language）查询语言，属于Spring整个生态体系的一部分。

新建项目 springboot\_jpa

添加Spring Data JPA的起步依赖

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  <**relativePath**/>  </**parent**>  <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  <**java.version**>1.8</**java.version**>  </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**>  <!-- springBoot JPA的起步依赖 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  </**dependency**>  <!-- MySQL连接驱动 -->  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  </**dependency**>  <!-- 配置使用redis启动器 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  </**dependency**>  </**dependencies**> |

在application.yml中配置数据库和jpa的相关属性

|  |
| --- |
| logging:  level:  com.atguigu.dao: debug # 配置日志  spring:  datasource:  username: root  password: root  url: jdbc:mysql://127.0.0.1:3306/springboot?useUnicode=**true**&characterEncoding=utf8&serverTimezone=Asia/Shanghai  driver-**class**-**name**: **com**.**mysql**.**cj**.**jdbc**.**Driver**  jpa:  database: mysql  show-sql: **true**  generate-ddl: **true**  hibernate:  ddl-auto: update  naming\_strategy: org.hibernate.cfg.ImprovedNamingStrategy  server:  port: 18081 |

注意：Mysql8.x版本，连接时url需要指定时区，并且驱动类包名发生了变化。

创建实体配置实体

|  |
| --- |
| **package** com.atguigu.domain;  **import** javax.persistence.\*;  @Entity  @Table(name = "user")  **public** **class** **User**{  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "id")  **private** Long id;  @Column(name = "username")  **private** String username;  @Column(name = "password")  **private** String password;  @Column(name = "name")  **private** String name;    //此处省略setter和getter方法... ...  } |

编写UserRepository

|  |
| --- |
| **package** com.atguigu.dao;  **import** com.atguigu.domain.User;  **import** org.springframework.data.jpa.repository.JpaRepository;  **public** **interface** **UserDao** **extends** **JpaRepository**<**User**,**Integer**> {  } |

编写service类

|  |
| --- |
| **package** com.atguigu.service;  **import** com.atguigu.domain.User;  **import** java.util.List;  **public** **interface** **UserService** {  List<User> **findUsers**();  User **findUserById**(Integer id);  **void** **saveUser**(User user);  **void** **updateUser**(User user);  **void** **deleteUserById**(Integer id);  } |

|  |
| --- |
| **package** com.atguigu.service.impl;  **import** com.atguigu.dao.UserDao;  **import** com.atguigu.domain.User;  **import** com.atguigu.service.UserService;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Service;  **import** java.util.List;  @Service  **public** **class** **UserServiceImpl** **implements** **UserService** {  @Autowired  **private** UserDao userDao;  /\*\*  \* 查询所有  \* @return  \*/  @Override  **public** List<User> **findUsers**() {  **return** userDao.findAll();  }  /\*\*  \* 根据id查询  \* @return  \*/  @Override  **public** User **findUserById**(Integer id) {  **return** userDao.findById(id).get();  }  /\*\*  \* 保存  \* @return  \*/  @Override  **public** **void** **saveUser**(User user) {  userDao.save(user);  }  /\*\*  \* 更新  \* @return  \*/  @Override  **public** **void** **updateUser**(User user) {  userDao.save(user);  }  /\*\*  \* 根据id删除  \* @return  \*/  @Override  **public** **void** **deleteUserById**(Integer id) {  userDao.deleteById(id);  }  } |

编写controller类

|  |
| --- |
| **package** com.atguigu.controller;  **import** com.atguigu.domain.User;  **import** com.atguigu.service.UserService;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** java.util.List;  @RestController  @RequestMapping("/user")  **public** **class** **UserController** {  @Autowired  **private** UserService userService;  @RequestMapping("/findAll")  **public** List<User> **findAll**(){  **return** userService.findUsers();  }  } |

编写主程序类

测试运行，控制台打印信息



# 四. SpringBoot综合案例(应用)

## 4.1 需求说明

功能：实现用户信息列表查询



技能

* SpringBoot整合SpringMVC
* SpringBoot整合MyBatis（tkmybatis）
* SpringBoot整合Druid数据源
* SpringBoot整合Redis
* SpringBoot整合Junit进行DAO层、Service测试
* 使用Postman进行controller层测试

## 4.2 环境搭建

### 4.2.1 数据库准备

|  |
| --- |
| **create** **database** springboot character **set** utf8 ;  **use** springboot ;  **CREATE** **TABLE** `tb\_user` (  `id` int(11) **NOT** NULL AUTO\_INCREMENT,  `name` varchar(20) **NOT** NULL,  `gender` varchar(5) **DEFAULT** NULL,  `age` int(11) **DEFAULT** NULL,  `address` varchar(32) **DEFAULT** NULL,  `qq` varchar(20) **DEFAULT** NULL,  `email` varchar(50) **DEFAULT** NULL,  `username` varchar(20) **NOT** NULL,  `phone` varchar(11) **DEFAULT** NULL,  **PRIMARY** **KEY** (`id`),  **UNIQUE** **KEY** `user\_username\_uindex` (`username`)  ) **ENGINE**=**InnoDB** AUTO\_INCREMENT=8 **DEFAULT** **CHARSET**=utf8;  **INSERT** **INTO** `tb\_user` **VALUES** (1,'黄蓉','女',38,'桃花岛','212223390222','huangrong222@qq.com','huangrong','15600003333'),(2,'黄老邪','男',58,'湖北省武汉市','212223390','huanglaoxie@qq.com','huanglaoxie','15872320405'),(3,'小龙女','男',18,'湖北省荆门市','212223390','xiaolongnv@qq.com','xiaolongnv','15600004444'),(7,'杨过','男',30,'扬州','212223390','yangguo@qq.com','yangguo','15600005555'); |

### 4.2.2 创建项目及包结构

创建项目 springboot\_case

### 4.2.3 导入依赖

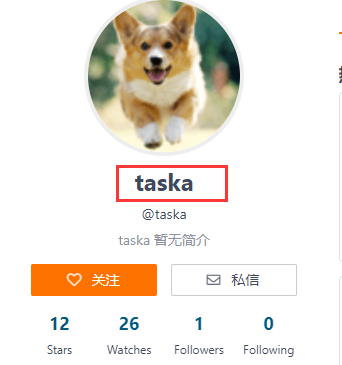
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <**project** **xmlns**="http://maven.apache.org/POM/4.0.0"  **xmlns:xsi**="http://www.w3.org/2001/XMLSchema-instance"  **xsi:schemaLocation**="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <**modelVersion**>4.0.0</**modelVersion**>  <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  </**parent**>  <**dependencies**>  <!--单元测试启动器-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  </**dependency**>  <!--通用mapper启动器依赖-->  <**dependency**>  <**groupId**>tk.mybatis</**groupId**>  <**artifactId**>mapper-spring-boot-starter</**artifactId**>  <**version**>2.1.5</**version**>  </**dependency**>  <!--JDBC启动器依赖-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**>  </**dependency**>  <!--mysql驱动-->  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.47</**version**>  </dependency>  <!--druid启动器依赖-->  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid-spring-boot-starter</**artifactId**>  <**version**>1.1.10</**version**>  </**dependency**>  <!--web启动器依赖-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <!--spring boot actuator依赖-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**>  <!--编码工具包-->  <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-lang3</**artifactId**>  </**dependency**>  <!—热部署 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**scope**>runtime</**scope**>  <**optional**>true</**optional**>  </**dependency**>  </**dependencies**>  <**build**>  <**plugins**>  <!--spring boot maven插件 , 可以将项目运行依赖的jar包打到我们的项目中-->  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**>  </**project**> |

### 4.2.4 创建启动类

|  |
| --- |
| **package** com.atguigu;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** tk.mybatis.spring.annotation.MapperScan;  @SpringBootApplication  @MapperScan(basePackages = "com.atguigu.dao")  @EnableTransactionManagement  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(Application.class,args);  }  } |

## 4.3 数据访问层

Tkmybatis 是基于 Mybatis 框架开发的一个工具，通过调用它提供的方法实现对单表的数据操作，不需要写任何 sql 语句，这极大地提高了项目开发效率。



可以在一些并发量不大、对性能要求不高的项目中尝试下 Tkmybatis ，对一些比较大的项目来说，还是希望自己对 SQL 的控制更强一点。

### 4.3.1 编写配置文件application.yml

|  |
| --- |
| server:  port: 10001  spring:  datasource:  driver-**class-name**: com.mysql.jdbc.Driver  url: jdbc:mysql:///springboot  username: root  password: root  type: com.alibaba.druid.pool.DruidDataSource  **druid**:  *# 连接池的配置信息  # 初始化大小，最小，最大* **initial-size**: 5  **min-idle**: 5  **maxActive**: 20  *# 配置获取连接等待超时的时间* **maxWait**: 60000  *# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒* **timeBetweenEvictionRunsMillis**: 60000  mybatis:  **type-aliases-package**: com.atguigu.pojo |

### 4.3.2 编写实体类User

|  |
| --- |
| package com.atguigu.pojo; import java.io.Serializable;  @Entity @Table(name = "tb\_user")  public class User implements Serializable {  private Integer id;  private String name;  private String gender;  private Integer age;  private String address;  private String qq;  private String email;  private String username;  private String phone;  … |

### 4.3.3 Mapper接口和映射配置

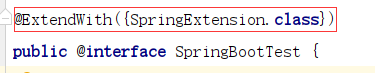
|  |
| --- |
| **package** com.atguigu.dao;  **import** com.atguigu.pojo.User;  **import** tk.mybatis.mapper.common.Mapper;  **import** java.util.List;  **public** **interface** **UserMapper** **extends** **Mapper**<**User**> {  } |

### 4.3.4 编写测试代码

|  |
| --- |
| @SpringBootTest  **public** **class** **UserMapperTest** {  @Autowired  **private** UserMapper userMapper ;  @Test  **public** **void** **findAll**() {  List<User> users = userMapper.selectAll();  System.out.println(users);  }  } |

**Spring Boot整合单元测试 ：**

1. 类上添加@SpringBootTest 注解，内部整合了Junit5，指定这是一个Spring Boot的测试类, 运行时会自动加载Spring Boot运行环境



1. 测试方法上使用@Test注解，主要要写org.junit.jupiter.api.Test。

## 4.4 业务层

### 4.4.1编****写接口****

|  |
| --- |
| **public** **interface** **UserService** {  /\*\*  \* 查询所有用户信息  \* @return  \*/  **public** List<User> **findAll**();  } |

### ****4.4.2 编写实现类****

|  |
| --- |
| **package** com.atguigu.service.impl;  **import** com.atguigu.dao.UserMapper;  **import** com.atguigu.pojo.User;  **import** com.atguigu.service.UserService;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Service;  **import** java.util.List;  @Service  **public** **class** **UserServiceImpl** **implements** **UserService** {  @Autowired  **private** UserMapper userMapper;  @Override  @Transactional(readOnly = true ,propagation = Propagation.SUPPORTS)  **public** List<User> **findAll**() {  **return** userMapper.selectAll();  }  } |

### ****4.4.3 编写测试代码****

|  |
| --- |
| @SpringBootTest  **public** **class** **UserServiceTest** {  @Autowired  **private** UserService userService;  @Test  **public** **void** **findAll**() {  List<User> users = userService.findAll();  System.out.println(users);  }  } |

## 4.5 表现层

### 4.5.1 引入起步依赖

|  |
| --- |
| <!--Web起步依赖-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <!--编码工具包-->  <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-lang3</**artifactId**>  </**dependency**> |

### 4.5.2 新建工具类

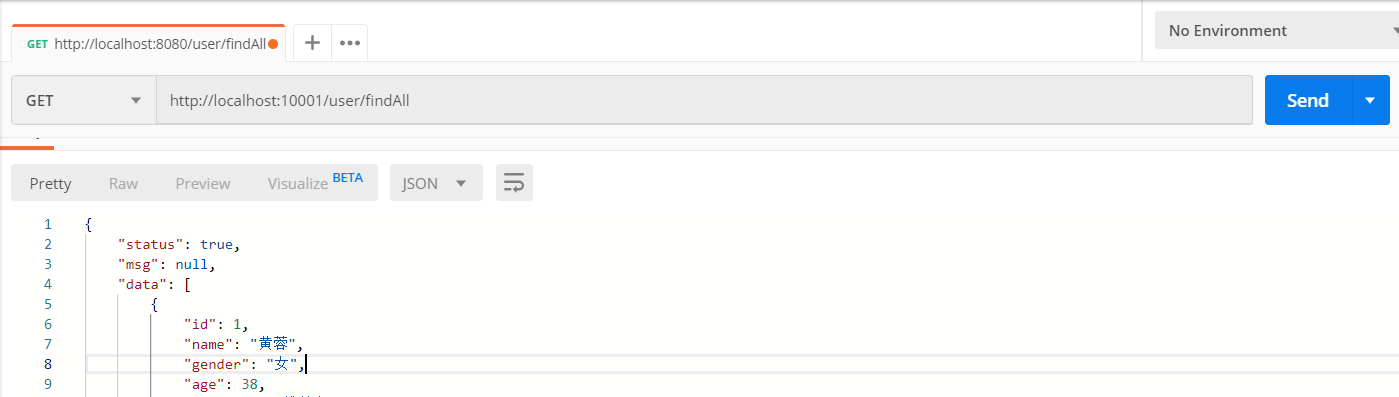
|  |
| --- |
| **package** com.atguigu.utils;  **import** java.io.Serializable;  **public** **class** **Result** **implements** **Serializable** {  **private** **boolean** status ; //响应状态 true false  **private** String msg ; // 响应信息  **private** Object data ; //处理成功的响应数据  **public** **static** Result **ok**(Object data){  Result result = **new** Result();  result.setStatus(**true**);  result.setData(data);  **return** result ;  }  **public** **static** Result **error**(String msg){  Result result = **new** Result();  result.setStatus(**false**);  result.setMsg(msg);  **return** result ;  }  // 生成set get tostring方法  } |

### 4.5.3 编写表现层代码

|  |
| --- |
| @Controller  @RequestMapping(path = "/user")  **public** **class** **UserController** {  @Autowired  **private** UserService userService;  @RequestMapping(path = "/findAll")  @ResponseBody  **public** Result **findAll**() {  List<User> users = userService.findAll();  **return** Result.ok(users);  }  } |

### 4.5.4 代码测试

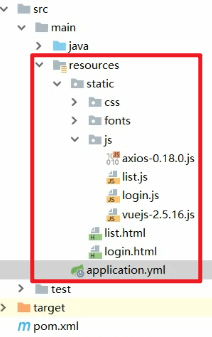
使用postman进行测试



## 4.6 页面展示

在resources目录下创建static目录 , 将提供的页面复制进来 , 修改即可 :

* 页面异步请求的端口和服务器端口一致
* 页面异步请求访问的路径和对应的表现层控制方法路径要致
* 页面异步请求参数名称和和对应的表现层控制方法参数一致

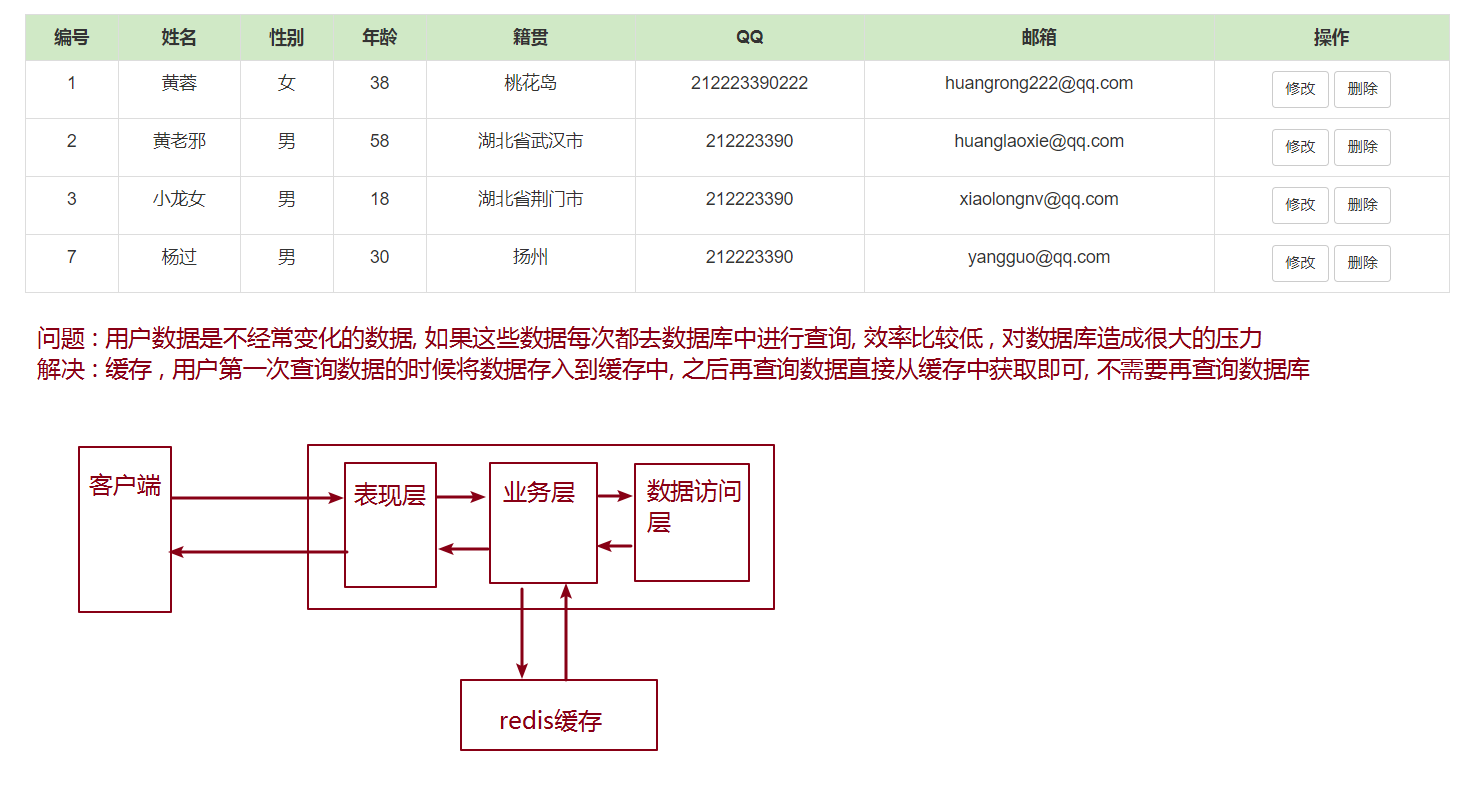
****

修改之后, 访问页面即可 : localhost:10001/list.html

****

## 4.7 缓存优化

### 4.7.1 缓存需求



### 4.7.2 引入起步依赖

|  |
| --- |
| <!--springboot整合redis启动器-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  </**dependency**> |

### 4.7.3 编写配置文件

|  |
| --- |
| spring:  redis: # 配置redis  host: localhost  port: 6379 |

### 4.7.4 修改业务层实现类代码

|  |
| --- |
| **package** com.atguigu.service.impl;  **import** com.atguigu.mapper.UserMapper;  **import** com.atguigu.pojo.User;  **import** com.atguigu.service.UserService;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.data.redis.core.RedisTemplate;  **import** org.springframework.stereotype.Service;  **import** org.springframework.transaction.annotation.\*;  **import** java.util.List;  @Service  **public** **class** **UserServiceImpl** **implements** **UserService** {  @Autowired  **private** UserMapper userMapper ;  @Autowired  **private** RedisTemplate redisTemplate ;  @Override  @Transactional(readOnly = **true** ,propagation = Propagation.SUPPORTS)  **public** List<User> **findAll**() {  //从缓存中查询数据 规定存储用户信息使用string类型进行存储, 存储的key就是userList  List<User> userList = (List<User>) redisTemplate.boundValueOps("userList").get();  //如果缓存中没有数据, 查询数据库 , 将查询到的数据放入缓存  **if**(userList==**null**){  userList = userMapper.findAll();  redisTemplate.boundValueOps("userList").set(userList);  System.out.println("从数据库中查询...");  }**else** {  System.out.println("从缓存中查询.....");  }  //如果缓存中有数据, 直接返回  **return** userList ;  }  } |

# 五. SpringBoot其他组件（了解）

## 5.1 SpringBoot Actuator组件

**Spring Boot Actuator**是SpringBoot自带的一个组件 , 可以帮助我们监控和管理Spring Boot应用，比如健康检查、审计、统计和HTTP追踪等。

**引入SpringBoot Actuator起步依赖**

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**> |

**配置SpringBoot Actuator参数**

|  |
| --- |
| management:  endpoints:  web:  exposure:  include: '\*' # 对外暴露的访问入口 , 默认是/health和/info  base-path: /monitor # 默认是actuator  endpoint:  health:  show-details: ALWAYS # 显示所有健康状态  server:  port: 9999 |

**启动项目获取系统信息**

项目启动之后就可以通过发送http请求获取系统健康数据了 , 例如 : http://localhost:9999/monitor/health , 返回数据如下 :

|  |
| --- |
| {  "**status**": "UP",  "**details**": {  "**db**": {  "**status**": "UP",  "**details**": {  "**database**": "MySQL",  "**hello**": 1  }  },  "**diskSpace**": {  "**status**": "UP",  "**details**": {  "**total**": 355816562688,  "**free**": 129251151872,  "**threshold**": 10485760  }  },  "**redis**": {  "**status**": "UP",  "**details**": {  "**version**": "2.8.9"  }  }  }  } |

常用的访问路径如下 :

| HTTP 方法 | 路径 | 描述 |
| --- | --- | --- |
| GET | /autoconfig | 提供了一份自动配置报告，记录哪些自动配置条件通过了，哪些没通过 |
| GET | /**configprops** | 描述配置属性(包含默认值)如何注入Bean |
| GET | /**beans** | 描述应用程序上下文里全部的Bean，以及它们的关系 |
| GET | /dump | 获取线程活动的快照 |
| GET | /**env** | 获取全部环境属性 |
| GET | /env/{name} | 根据名称获取特定的环境属性值 |
| GET | /**health** | 报告应用程序的健康指标，这些值由HealthIndicator的实现类提供 |
| GET | /info | 获取应用程序的定制信息，这些信息由info打头的属性提供 |
| GET | /**mappings** | 描述全部的URI路径，以及它们和控制器(包含Actuator端点)的映射关系 |
| GET | /metrics | 报告各种应用程序度量信息，比如内存用量和HTTP请求计数 |
| GET | /metrics/{name} | 报告指定名称的应用程序度量值 |
| POST | /shutdown | 关闭应用程序，要求endpoints.shutdown.enabled设置为true |
| GET | /trace | 提供基本的HTTP请求跟踪信息(时间戳、HTTP头等) |

我们可以通过发送这些请求, 获取系统状态信息

## 5.2 SpringBoot Admin组件

上面我们讲了Spring Boot Actuator , 可以通过http协议获取系统状态信息 , 但是返回的是JSON格式数据, 看起来不太方面, 而且还需要记忆路径, 比较麻烦 , Spring Boot Admin给我们提供了更加友好的可视化界面来查看这些信息 !

Spring Boot Admin是一个开源社区项目，用于管理和监控SpringBoot应用程序。 应用程序作为Spring Boot Admin Client向Spring Boot Admin Server注册 , Client会定时向Server发送数据, Server使用友好界面展示数据。



### 5.2.1 SpringBoot Admin服务端

1. **创建项目**springboot-admin-server
2. **起步依赖**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <**project** **xmlns**="http://maven.apache.org/POM/4.0.0"  **xmlns:xsi**="http://www.w3.org/2001/XMLSchema-instance"  **xsi:schemaLocation**="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <**modelVersion**>4.0.0</**modelVersion**>  <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  </**parent**>  <**groupId**>com.atguigu</**groupId**>  <**artifactId**>springboot-admin-server</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  <**properties**>  <**java.version**>1.8</**java.version**>  </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>de.codecentric</**groupId**>  <**artifactId**>spring-boot-admin-starter-server</**artifactId**>  <**version**>2.2.0</**version**>  </**dependency**>  </**dependencies**>  </**project**> |

1. **配置application.yml**

|  |
| --- |
| spring:  application:  name: admin-server  server:  port: 8769 |

Spring Boot Admin 端口号 8769

1. **启动类**

|  |
| --- |
| @SpringBootApplication  **@EnableAdminServer**  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(Application.class,args);  }  } |

@EnableAdminServer , 开启管理服务

### 5.2.2 SpringBoot Admin客户端

1. **起步依赖**

|  |
| --- |
| <**dependency**>  <**groupId**>de.codecentric</**groupId**>  <**artifactId**>spring-boot-admin-starter-client</**artifactId**>  <**version**>2.2.0</**version**>  </**dependency**> |

1. **配置**

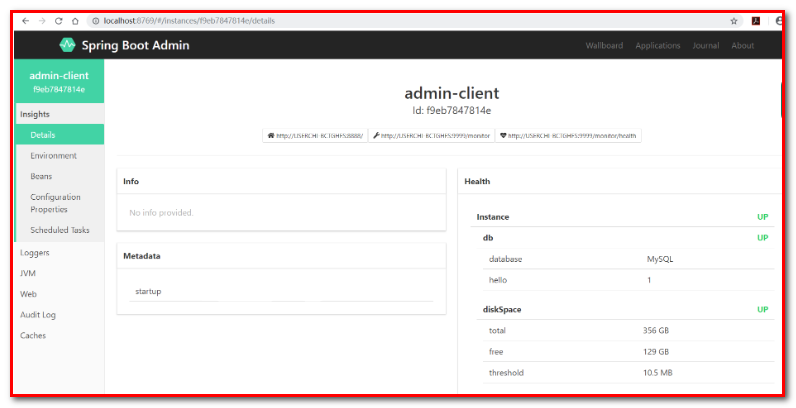
向admin-server注册的地址为http://localhost:8769，最后暴露自己的actuator的所有端口信息，具体配置如下：

|  |
| --- |
| server:  port: 9999  spring:  application:  name: admin-client  boot:  admin:  client:  url: http://localhost:8769 # 指定注册地址 , Spring Boot Admin Server地址  management:  endpoints:  web:  exposure:  include: '\*'  endpoint:  health:  show-details: ALWAYS |

注册地址一定要和Spring Boot Admin Server地址匹配

### 5.3.3 启动测试

分别开启客户端(应用程序)和服务端 ,访问http://localhost:8769 , 可以看到如下界面



# 六. Spring Boot 自动配置(理解)

## 6.1 @SpringBootApplication注解

* @SpringBootConfiguration : 代表这个类就是一个配置类 , 本质上就是一个@Configuration注解
* @ComponentScan : 组件扫描, 默认扫描启动类所在包及子包下的类身上的注解
* @EnableAutoConfiguration : 自动配置注解 , 添加了此注解会自动去读取spring.factories配置文件中的自动配置类

## 6.2 条件化配置注解

我们看到**自动配置类**上有一些ConditionXxxx注解 , 这些注解的作用就是进行条件化选择

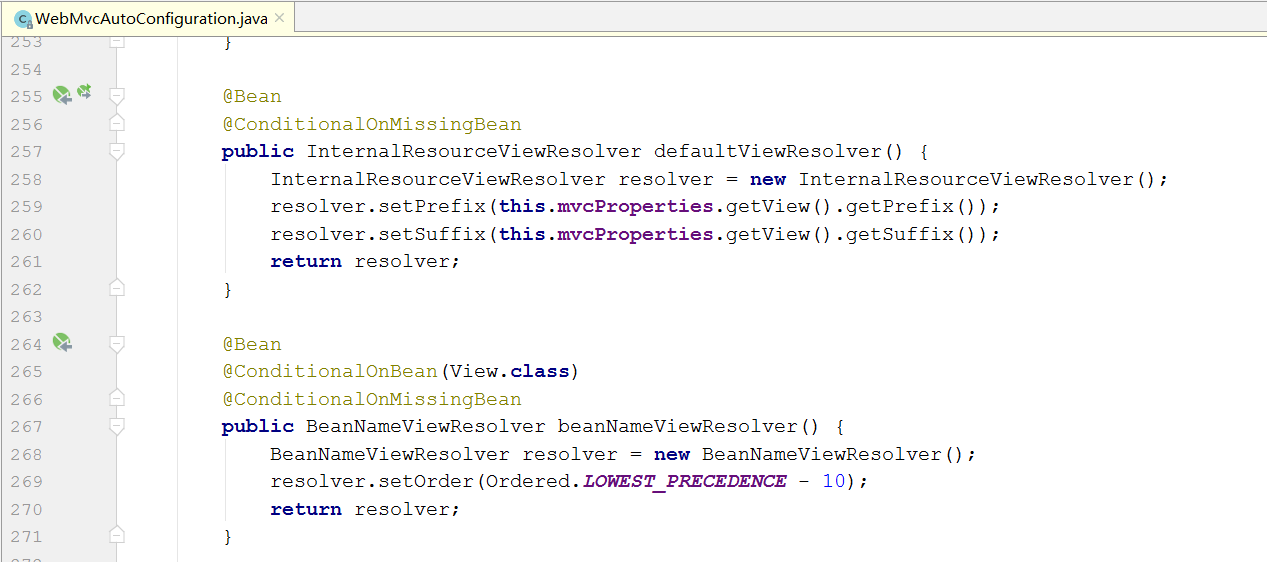
所谓条件化选择就是如果满足条件, 该配置类就生效, 如果不满足该配置类就不生效

常用的条件化选择注解如下 :

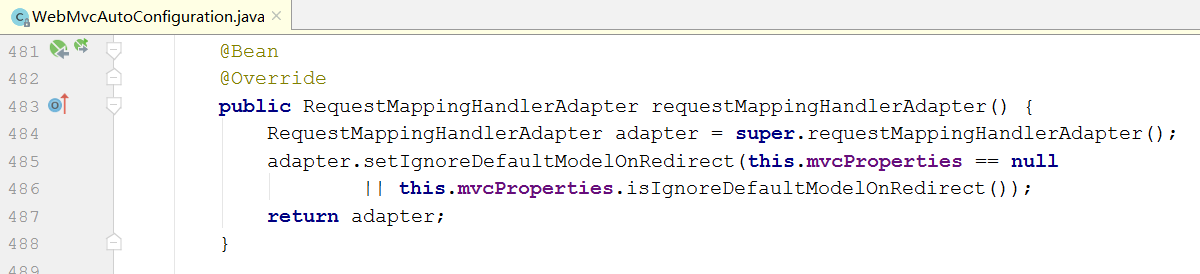
| 注解 | 作用 |
| --- | --- |
| @ConditionalOnBean | 如果存在某个Bean, 配置类生效 |
| @ConditionalOnMissingBean | 如果不存在某个Bean, 配置类生效 |
| @ConditionalOnClass | 如果存在某个类, 配置类生效 |
| @ConditionalOnMissingClass | 如果不存在某个类, 配置类生效 |
| @ConditionalOnProperty | 如果存在某个属性配置, 配置类生效 |
| @ConditionalOnWebApplication | 如果是一个web应用, 配置类生效 |
| @ConditionalOnNotWebApplication | 如果不是一个web应用, 配置类生效 |

因为我们配置了DispatcherServlet 满足上面定义的条件, 所以WebMvcAutoConfiguration会生效 , 那么WebMvcAutoConfiguration自动配置类中帮我们配置了什么呢 ?

**视图解析器**



**处理器适配器（HandlerAdapter）**

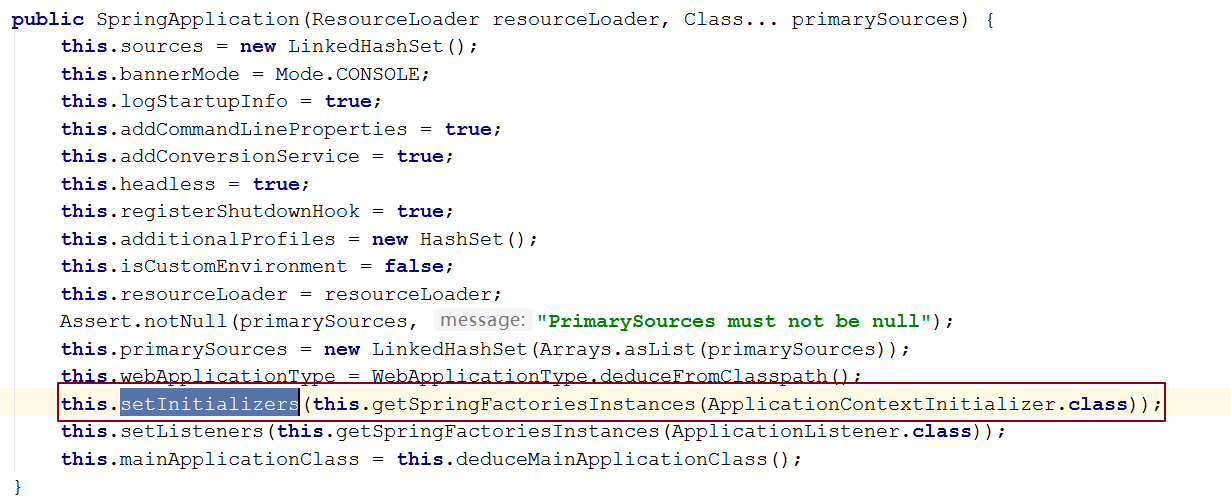


这些配置都是我们之前在学习SpringMVC时需要自己配置的 , 现在Spring Boot框架都已经提前帮我们配置好了 , 所以我们才能使用的那么方便

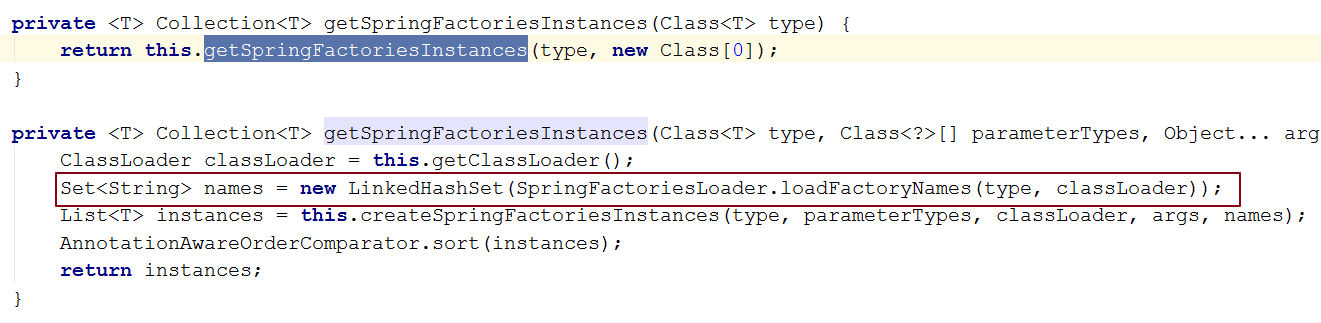
## 6.3 自动配置原理

### 6.3.1 加载spring.factories

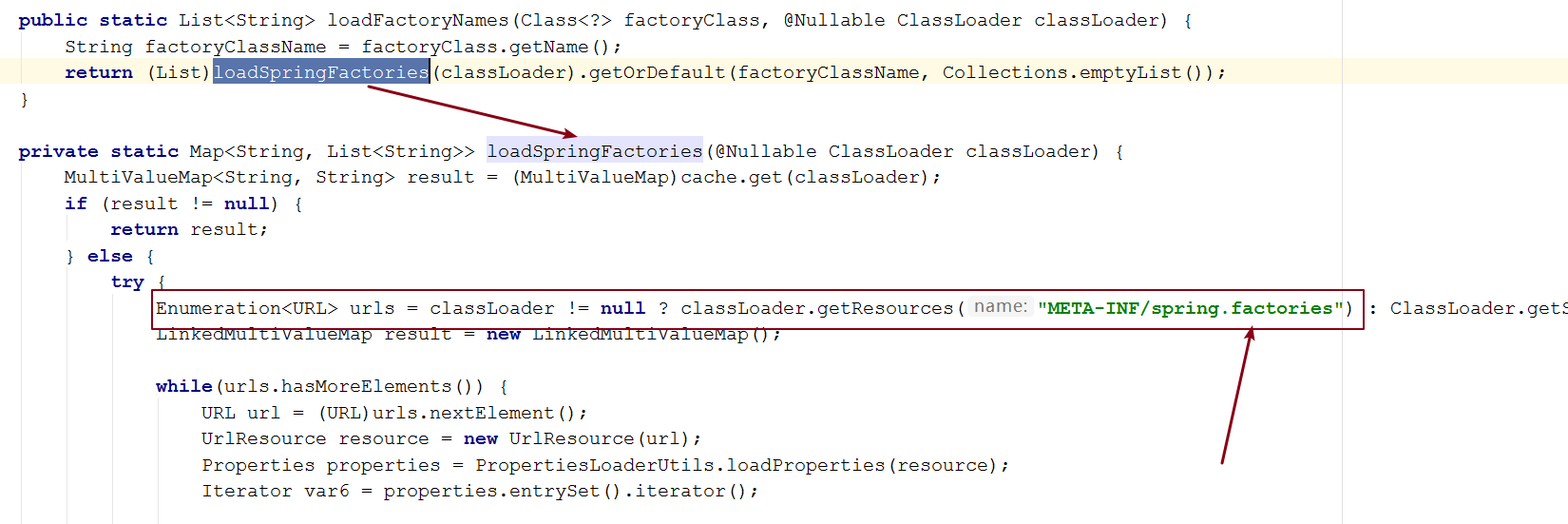
在\*\*SpringApplication\*\*类构建的时候，有这样一段初始化代码：



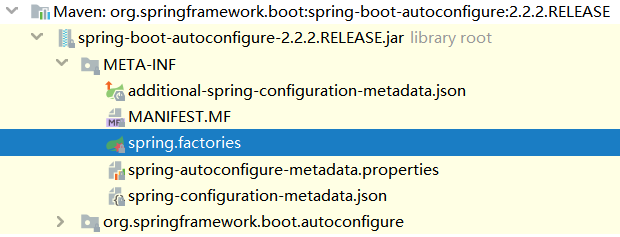
跟进去往下走



这里发现会通过loadFactoryNames尝试加载一些FactoryName，然后利用createSpringFactoriesInstances将这些加载到的类名进行实例化。 继续跟进loadFactoryNames方法：



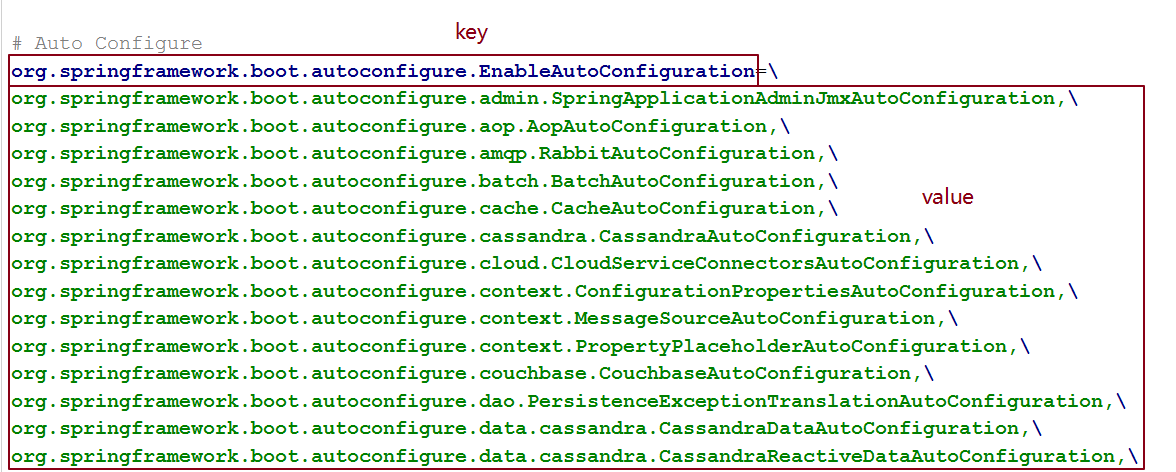
发现此处会利用类加载器加载一个文件： META-INF/spring.factories 。我们知道，ClassLoader默认是从classpath下读取文件，因此，SpringBoot会在初始化的时候，加载所有classpath:META-INF/spring.factories文件，包括jar包当中的。而在Spring的一个依赖包：spring-boot-autoconfigure中，就有这样的文件：



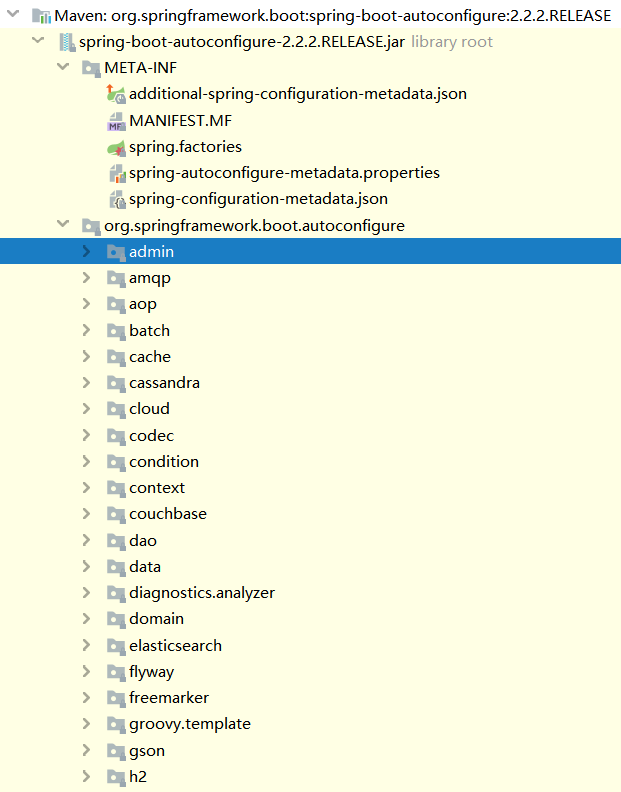
我们引入的任何第三方启动器，只要实现自动配置，也都会有类似文件。

### 6.3.2 读取自动配置类

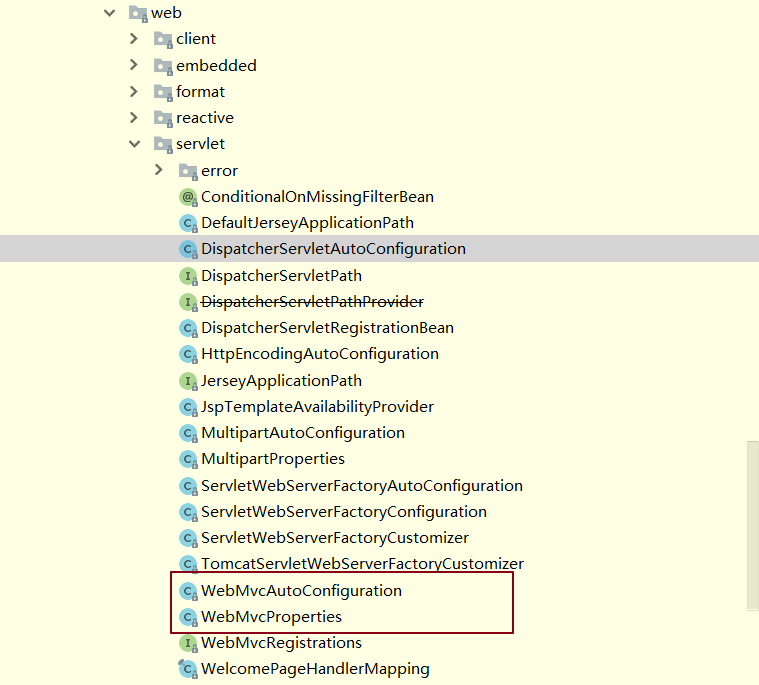
我们打开\*\*spring.factories\*\*文件



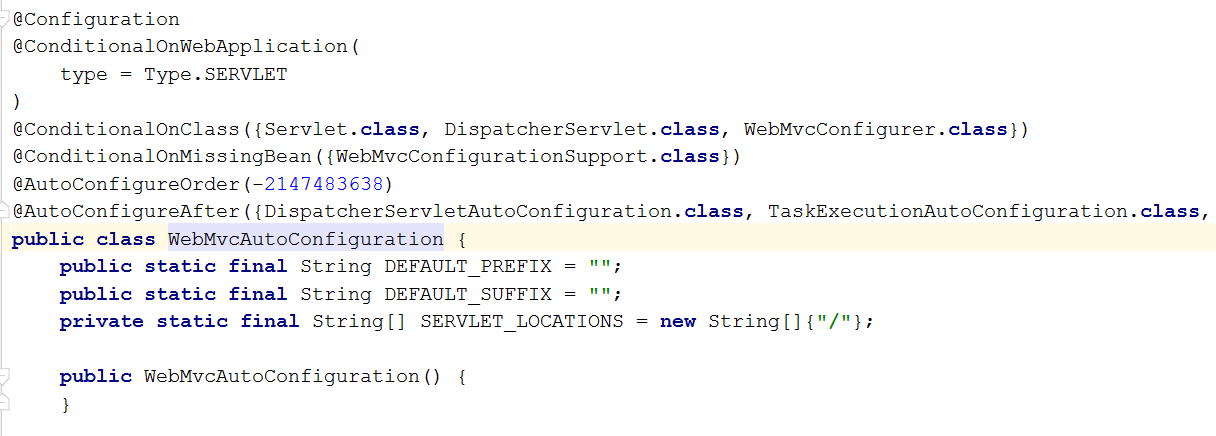
可以发现以EnableAutoConfiguration接口为key的一系列配置，key所对应的值，就是所有的自动配置类，可以在当前的jar包中找到这些自动配置类：



几乎涵盖了现在主流的开源框架 , 我们来看一个我们熟悉的，例如SpringMVC，查看mvc 的自动配置类：



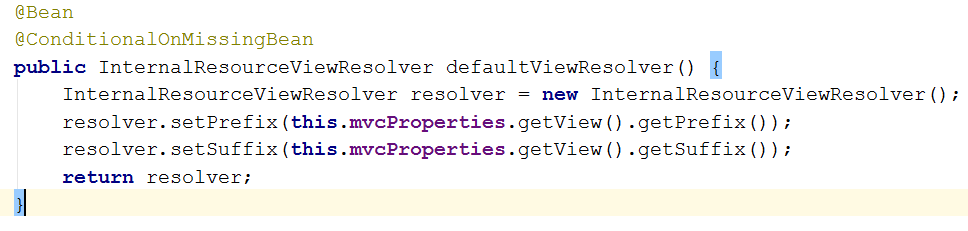
打开WebMvcAutoConfiguration



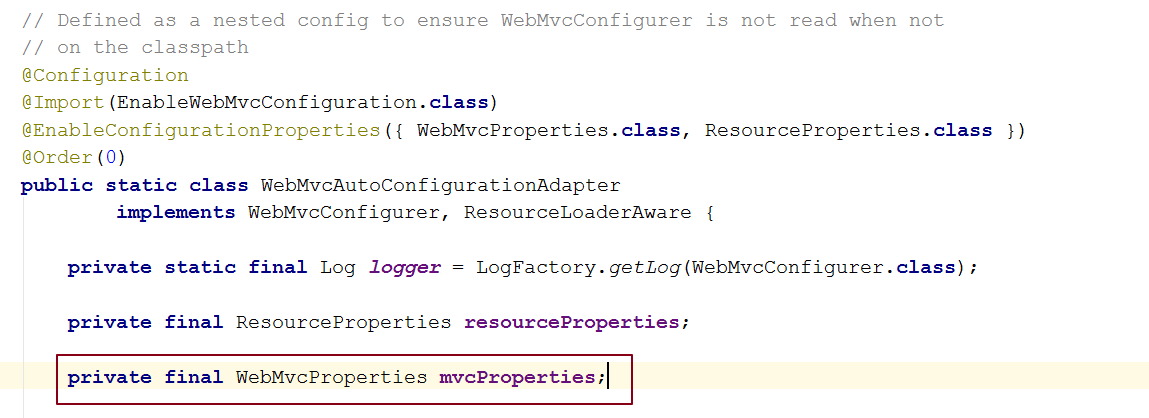
### 6.3.3 默认属性配置

配置类我们找到了 , 那么这些默认配置的属性来自哪里呢？

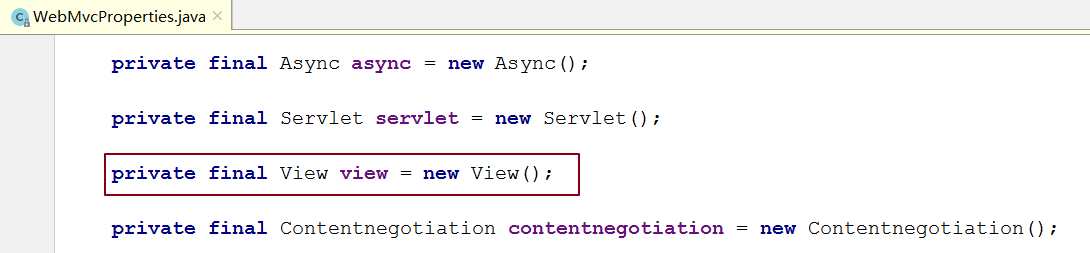
例如 : 我们配置视图解析器的时候需要配置前缀和后缀 , 那么这些配置在哪配置的呢 ?



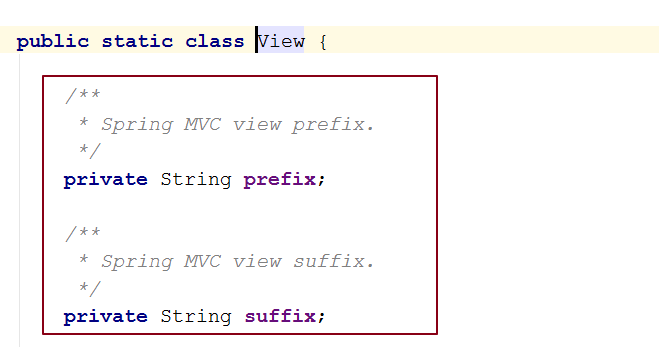
通过源码发现, 这个配置是从this.mvcProperties.getView()中读取的 ,this.mvcProperties又是什么呢 ? 我们继续跟踪,发现其实就是定义的一个变量



这个变量中又有一个View类型的变量 , 这个变量中配置的就是前缀和后缀



View中的源码如下 :

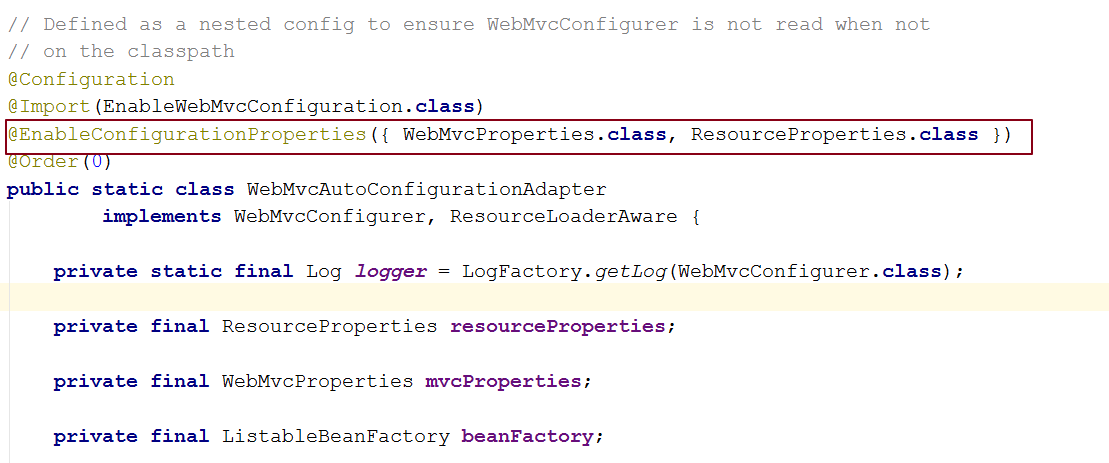


可以看到, 默认读取就是这里的前缀和后缀变量的值 , 默认就是null .

### 6.3.4 覆盖默认属性配置

如果我们想自己指定视图的前缀和后缀该如何去做呢 ?

我们再看WebMvcAutoConfiguration这个类的声明, 发现这个类身上有这么一个注解, 我们之前也使用过 , 引入了二个配置对象 , 所以我们知道下面成员位置的WebMvcProperties这个成员变量 , 就是从这里来的



我们再进去看看这两个配置类 , 配置类身上使用ConfigurationProperties读取配置,前缀是spring.mvc , 所以如果我们在配置文件中配置spring.mvc前缀开头的配置 , 是不是就可以将自己配置的数据注入到这个对象的属性中 !



所以如果想要自己设置视图前缀和后缀就可以这么配置

# 激活配置文件

|  |
| --- |
| spring:  mvc:  view:  prefix: /WEB-INF/  suffix: .jsp |

修改服务器端口 :

|  |
| --- |
| server:  port: 10000 |

## 6.4 自定义启动器

### 6.4.1 需求介绍

定义一个连接池启动器 , 当用户引入了连接池启动依赖之后 , 项目中就已经自动配置了连接池

### 6.4.2 步骤分析

1. 创建启动器项目
2. 添加启动器相关依赖
3. 创建属性配置类
4. 创建自动配置类
5. 编写自动配置文件(\*\*spring.factories\*\*)
6. 使用自定义的启动器

### 6.4.3 代码实现

#### 6.4.3.1 创建项目并引入依赖

**创建项目** spring-boot-jdbc-starter

**引入依赖**

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.2.2.RELEASE</**version**>  </**parent**>  <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  <**java.version**>1.8</**java.version**>  </**properties**>  <**dependencies**>  <!--引入spring‐boot‐starter；所有starter的基本配置-->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter</**artifactId**>  </**dependency**>  <!--自动配置连接池-->  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>c3p0</**groupId**>  <**artifactId**>c3p0</**artifactId**>  <**version**>0.9.1.2</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**>  </**dependency**>  </**dependencies**> |

#### 6.4.3.2 [创建属性配置类](file:///D:\\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\8.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_SpringBoot_1%E5%A4%A9_V3.0\\springboot%E9%AB%98%E7%BA%A7.html" \l "创建属性配置类)

|  |
| --- |
| **package** com.atguigu.autoconfig;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  @Component  @ConfigurationProperties(prefix = "spring.jdbc.datasource")  **public** **class** **DataSourceProperties** {  **private** String driverClassName ;  **private** String url;  **private** String username;  **private** String password;  // 生成set get toString方法  } |

#### 6.4.3.3 [创建自动配置类](file:///D:\\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\8.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_SpringBoot_1%E5%A4%A9_V3.0\\springboot%E9%AB%98%E7%BA%A7.html" \l "创建自动配置类)

|  |
| --- |
| **package** com.atguigu.autoconfig;  **import** com.alibaba.druid.pool.DruidDataSource;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.context.properties.EnableConfigurationProperties;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** javax.sql.DataSource;  @SpringBootConfiguration  @EnableConfigurationProperties(DataSourceProperties.class)  **public** **class** **DataSourceAutoConfiguration** {  @Autowired  **private** DataSourceProperties dataSourceProperties ;  @Bean  **public** DataSource **createDataSource**(){  DruidDataSource dataSource = **new** DruidDataSource();  dataSource.setDriverClassName(dataSourceProperties.getDriverClassName());  dataSource.setUrl(dataSourceProperties.getUrl());  dataSource.setUsername(dataSourceProperties.getUsername());  dataSource.setPassword(dataSourceProperties.getPassword());  **return** dataSource;  }  } |

#### 6.4.3.4 [编写自动配置属性文件](file:///D:\\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\8.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_SpringBoot_1%E5%A4%A9_V3.0\\springboot%E9%AB%98%E7%BA%A7.html" \l "编写自动配置属性文件)

在 resource 文件夹下面新建 META-INF/spring.factories

|  |
| --- |
| # Auto Configure  org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.atguigu.autoconfig.DataSourceAutoConfiguration |

做完了之后注意要执行install , 安装项目

#### 6.4.3.5 [使用自定义启动器](file:///D:\\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\\8.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_SpringBoot_1%E5%A4%A9_V3.0\\springboot%E9%AB%98%E7%BA%A7.html" \l "使用自定义启动器)

在 springboot\_01 项目当中引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>com.atguigu</**groupId**>  <**artifactId**>spring-boot-jdbc-starter</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**> |

**配置连接池信息**

新建 application-datasource.yml

|  |
| --- |
| spring:  jdbc:  datasource:  driverClassName: com.mysql.jdbc.Driver  url: jdbc:mysql:///springboot\_01  username: root  password: root |

激活配置文件 application.yml

# 激活配置文件

|  |
| --- |
| spring:  profiles:  active: datasource |

**注入连接池, 查看连接池属性**

|  |
| --- |
| **package** com.atguigu.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.context.properties.EnableConfigurationProperties;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** javax.sql.DataSource;  @RestController  **public** **class** **HelloController** {  @Autowired  **private** DataSource dataSource ;  @RequestMapping(path = "/hello")  **public** String **sayHello**() {  System.out.println(dataSource.getClass());//打印DruidDataSource数据源  **return** "Hello Spring Boot ! " ;  }  } |

### 6.4.4 多种数据源

如果想让我们的启动器支持多种数据源, 例如 : C3P0和Druid , 根据配置进行选择 , 就可以使用条件选择进行实现。例如 : 如下配置中, 有二个创建连接池的配置, 一个是C3P0 , 一个是Druid ,如何能够根据配置文件自动选择呢 ?

修改配置文件 application-datasource.yml

|  |
| --- |
| spring:  jdbc:  datasource:  driverClassName: com.mysql.jdbc.Driver  url: jdbc:mysql:///springboot\_01  username: root  password: root  type: druid # 数据源类型 |

* 如果配置文件中配置了spring.jdbc.datasource.type=c3p0使用c3p0数据源
* 如果配置文件中配置了spring.jdbc.datasource.type=druid使用druid数据源

在项目 spring-boot-jdbc-starter , 添加创建 c3p0 方法

|  |
| --- |
| **package** com.atguigu.autoconfig;  **import** com.alibaba.druid.pool.DruidDataSource;  **import** com.mchange.v2.c3p0.ComboPooledDataSource;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.context.properties.EnableConfigurationProperties;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** javax.sql.DataSource;  @Configuration  @EnableConfigurationProperties(DataSourceProperties.class)  **public** **class** **DataSourceAutoConfiguratioin** {  @Autowired  **private** DataSourceProperties dataSourceProperties ;  @Bean  @ConditionalOnProperty(value = "spring.jdbc.datasource.type",havingValue = "druid")  **public** DataSource **createDataSource**(){  DruidDataSource dataSource = **new** DruidDataSource();  dataSource.setDriverClassName(dataSourceProperties.getDriverClassName());  dataSource.setUrl(dataSourceProperties.getUrl());  dataSource.setUsername(dataSourceProperties.getUsername());  dataSource.setPassword(dataSourceProperties.getPassword());  **return** dataSource;  }  @Bean  @ConditionalOnProperty(value = "spring.jdbc.datasource.type",havingValue = "c3p0")  **public** DataSource **createC3P0DataSource**() **throws** Exception{  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setDriverClass(dataSourceProperties.getDriverClassName());  dataSource.setJdbcUrl(dataSourceProperties.getUrl());  dataSource.setUser(dataSourceProperties.getUsername());  dataSource.setPassword(dataSourceProperties.getPassword());  **return** dataSource;  }  } |

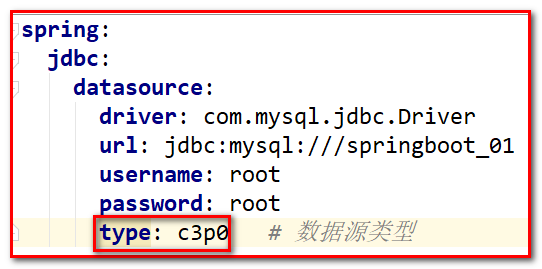
我们可以使用条件选择实现 , 如下图所示

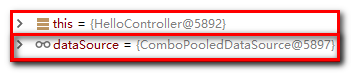
@ConditionalOnProperty(value = "spring.jdbc.datasource.type",havingValue = "druid")

install 安装 spring-boot-jdbc-starter , 运行 springboot\_01



修改配置文件 ，重新安装，再次请求





# 附录：SpringBoot原理

到现在为止，我们已经系统讲解了SpringBoot的技能点，能够使用SpringBoot进行开发了，对SpringBoot的一些核心概念也有了初步的认识，比如启动器、启动类、自动配置等，也能够亲身体会到使用SpringBoot的确大大简化了传统Spring项目的开发。

那么，SpringBoot为什么这么神奇？引入的依赖变少了，配置文件也不见了，但项目却可以正常运行。下面我们就来研究一下SpringBoot的核心原理和源码。

我们开发一个Spring Boot项目，都会用到如下的启动类

|  |
| --- |
| @SpringBootApplication **public class** MyApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(MyApplication.**class**);  } } |

其中其实只包含

* 一个注解：@SpringBootApplication
* 一条语句：SpringApplication.run(MyApplication.class);

要揭开SpringBoot的神秘面纱，我们从这两位开始就可以了。

### 4.1 @SpringBootApplication注解

进入@SpringBootApplication注解，发现虽然使用了多个原注解来标记，但最重要的却是三个：@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan。掌握这三个注解，也就理解了@SpringBootApplication的作用。

|  |
| --- |
| @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 {  } |

#### @SpringBootConfiguration注解

进入@SpringBootConfiguration，发现它其实就是一个@Configuration，而这个注解就是标记一个配置类。所以这个注解不是重点。

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

#### @ComponentScan注解

@ComponentScan注解在之前Spring中已用到，这是一个组件扫描类，可通过basePackages来指定扫描的一个或者多个基准路径，如果没有指定，就扫描当前类所在包及其子包中的注解（@Component、@Service、@Controller、@Mapper等），也可以通过excludeFilters 指定排除在扫描范围之外的Filters。所以这个注解不是重点。

#### @EnableAutoConfiguration注解

这个注解才是最重要的注解，是理解SpringBoot原理的一个核心注解。从字面含义是启用自动配置。进入该注解：发现理解该注解的关键理解两个注解：@AutoConfigurationPackage、@Import({AutoConfigurationImportSelector.class})。

|  |
| --- |
| @Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherited @AutoConfigurationPackage @Import({AutoConfigurationImportSelector.**class**}) **public** @**interface** EnableAutoConfiguration {  String ENABLED\_OVERRIDE\_PROPERTY = **"spring.boot.enableautoconfiguration"**;  Class<?>[] exclude() **default** {};  String[] excludeName() **default** {}; } |

#### @AutoConfigurationPackage注解

进入@AutoConfigurationPackage注解，发现其关键也是一个@Import注解：@Import({Registrar.class})。因为没有指定basePackages和basePackageClasses，所以自动扫描的包就是启动类所在包及其子包。

|  |
| --- |
| @Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherited @Import({Registrar.**class**}) **public** @**interface** AutoConfigurationPackage {  String[] basePackages() **default** {};   Class<?>[] basePackageClasses() **default** {}; } |

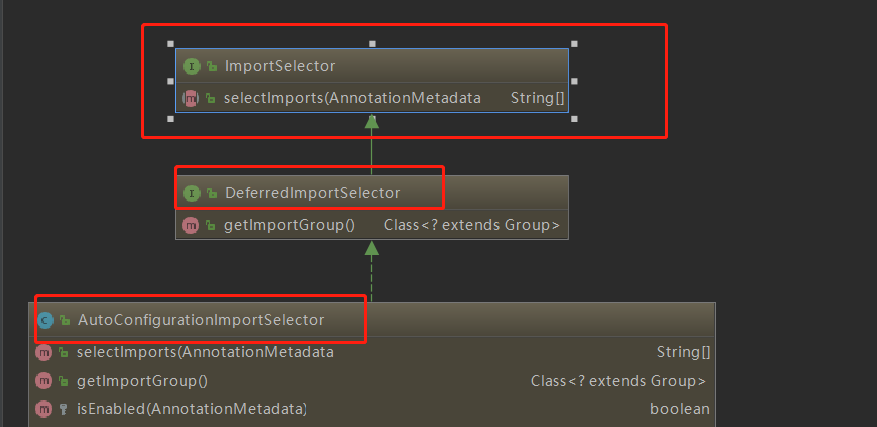
像@EnableAutoConfiguration类似的注解还有：@EnableScheduling、@EnableCaching、@EnableMBeanExport，其实一脉相承，简单概括一下就是，借助@Import的支持，收集和注册特定场景相关的bean定义加入到IoC容器中。

#### @Import({AutoConfigurationImportSelector.class})注解

@Import注解只能用在类上，作用是通过快速导入的方式实现把实例加入Spring的IOC容器中，完整路径org.springframework.context.annotation.Import，从路径上可以看到，这是Spring提供的注解。可以指定要快速导入的一个或者多个类。

|  |
| --- |
| @Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME) @Documented **public** @**interface** Import {  Class<?>[] value(); } |

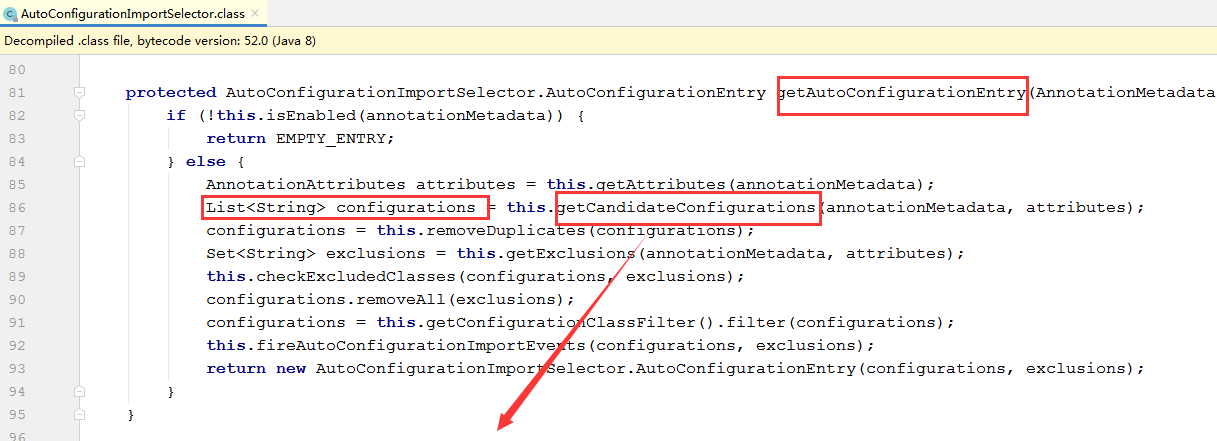
@Import({AutoConfigurationImportSelector.class})、@Import({Registrar.class})的作用就是要创建AutoConfigurationImportSelector、Registrar类的实例并加入到IoC容器中。



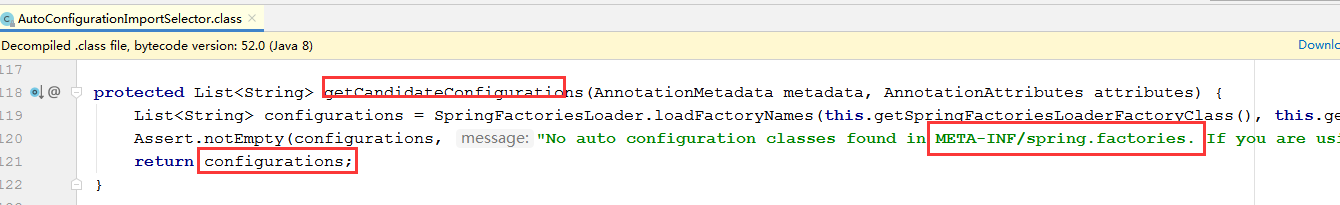
AutoConfigurationImportSelector类实现了IpmortSelector接口，实现了selectImports()接口。其中调用了getAutoConfigurationEntry()方法。



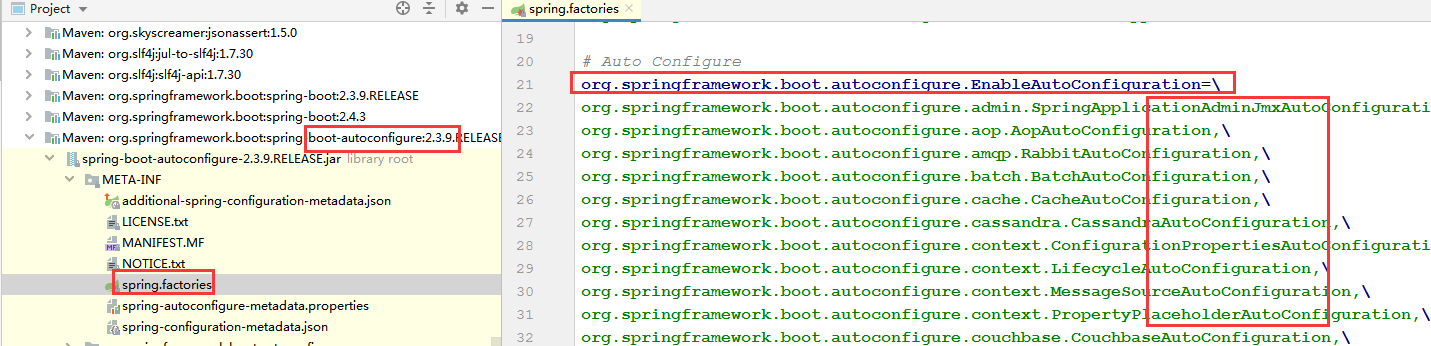
getAutoConfigurationEntry()方法中调用了getCandidateConfigurations()，该方法会获取所有的自动配置项列表。



该方法会借助SpringFactoriesLoader工具类的支持，默认加载该类所在spring-boot-autoconfigure-2.3.9.RELEASE.jar的META-INF/ spring.factories的信息。其中特别关注的是包括127个自动配置类的路径，会全部读取进List(然后会进行过滤，只保留符合条件的路径信息并进行实例化加载带IoC容器中)。



以下是spring.factories的信息。

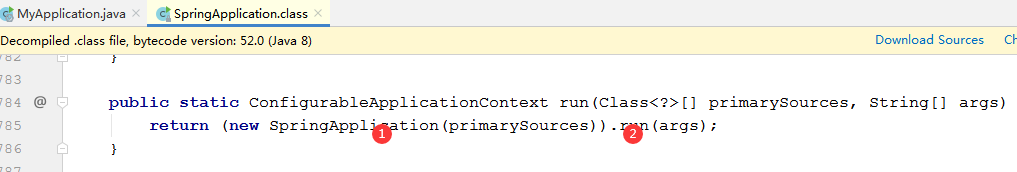


### 4.2 SpringApplication.run(MyApplication.class)；语句

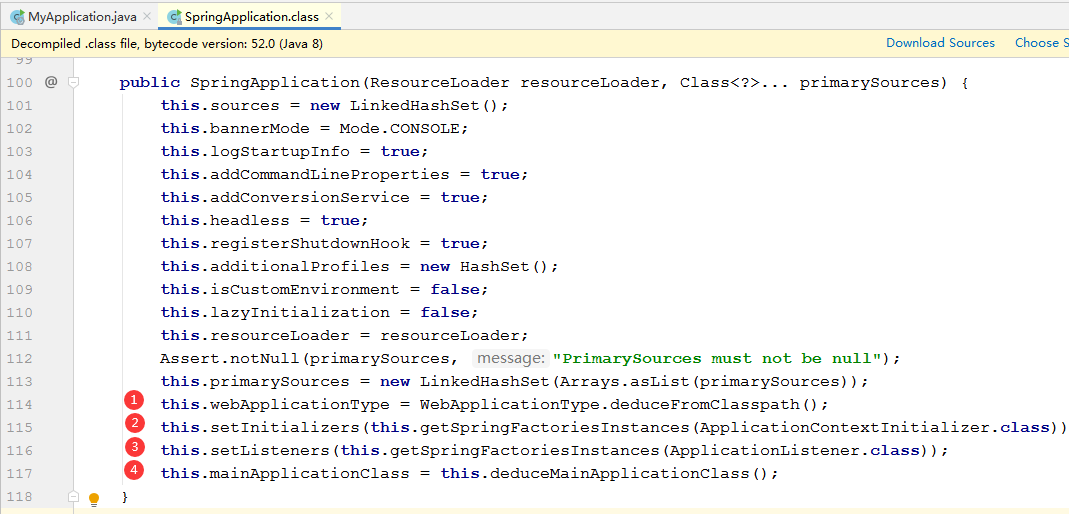
下面跟踪一下SpringApplication.run(MyApplication.class);的执行过程

|  |
| --- |
| @SpringBootApplication **public class** MyApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(MyApplication.**class**);  } } |

跟踪代码，关键点在于两个位置

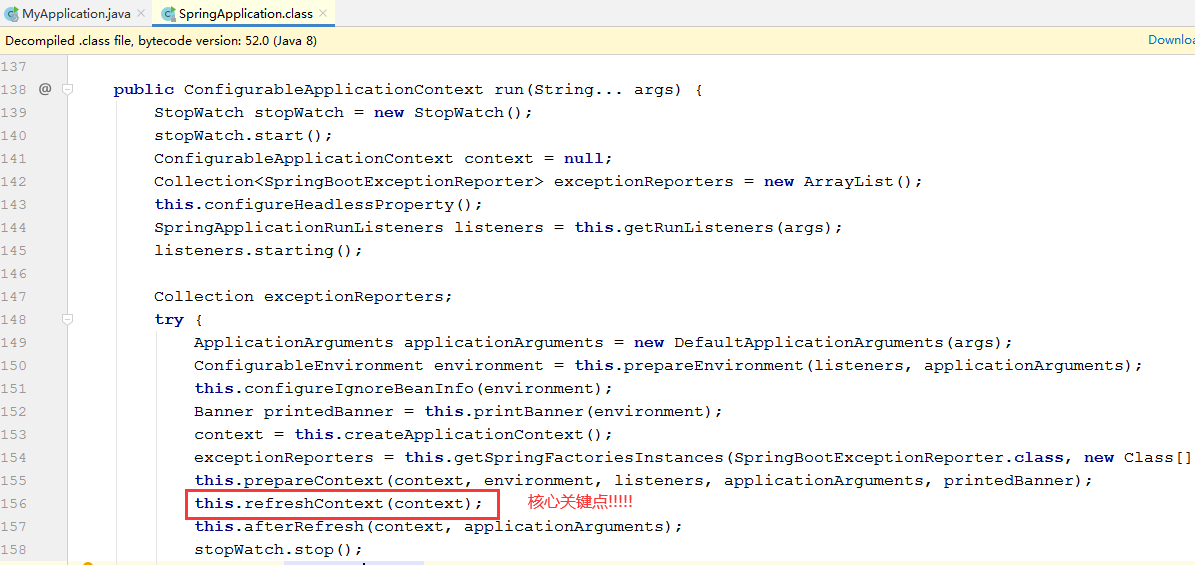


#### new SpringApplication(primarySources)

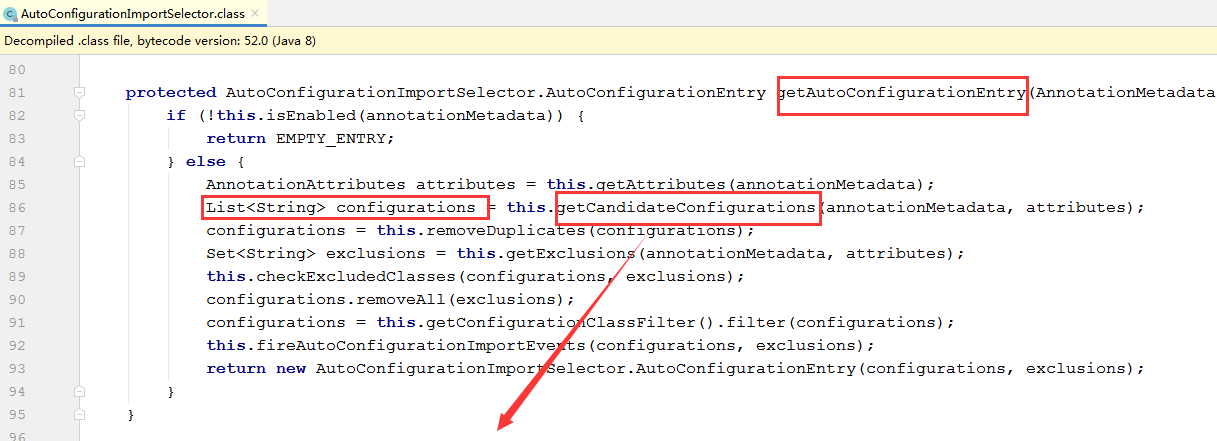


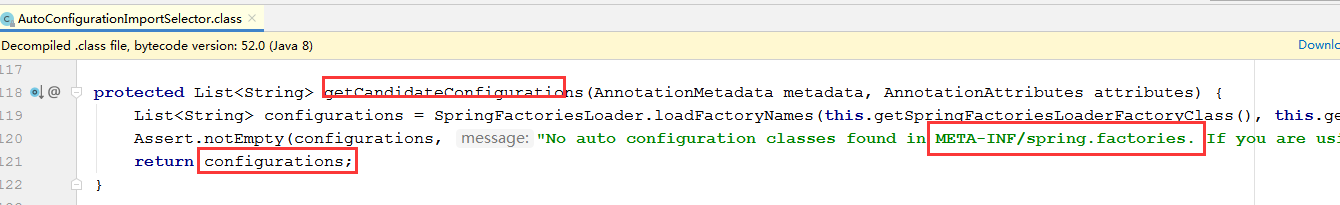
1. 根据classpath里面是否存在某个特征类来决定是否应该创建一个为Web应用类型：NONE、SERVLET、REACTIVE。
2. 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationContextInitializer。
3. 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationListener。
4. 推断并设置main方法的定义类。

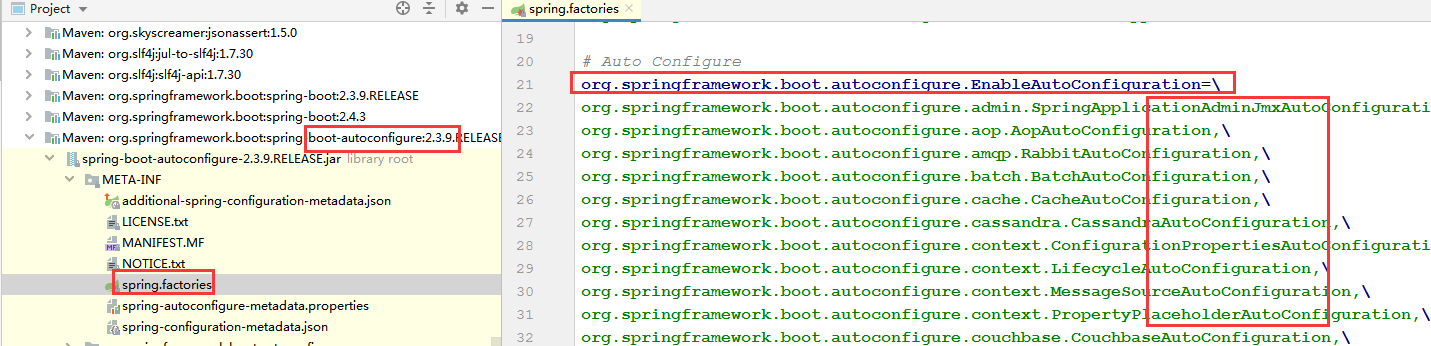
#### .run(args)



跟踪this.refreshContext(context);调用过程，最终调用AutoConfigurationImportSelector的getAutoConfigurationEntry()，其中会加载符合条件的自动配置类的实例并加载到IoC容器。







参考网址：<https://www.cnblogs.com/shamo89/p/8184960.html>