# springboot 的简介

## springboot

### 优点

优点：

-快速创建独立运行的Spring项目以及与主流框架集成

-使用嵌入式的Servlet容器，应用无需打成WAR包

-starters自动依赖与版本控制

-大量的自动配置，简化开发，也可修改默认值

-无需配置XML，无代码生成，开箱即用

-准生产环境的运行时应用监控

-与云计算的天然集成

## springboot的pom.xml

### 父项目

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.1.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  他的父项目是:  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>2.1.1.RELEASE</version>  <relativePath>../../spring-boot-dependencies</relativePath>  </parent> |

Spring Boot的版本仲裁中心:以后我们导入依赖默认是不需要写版本；（没有在dependencies里面管理的依赖自然需要声明版本号）

### 导入的依赖

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

spring-boot-starter:spring-boot场景启动器；帮我们导入了web模块正常运行所依赖的组件；

Spring Boot将所有的功能场景都抽取出来，做成一个个的starters(启动器），只需要在项目里面引入这些starter相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动器。

## 主程序类

|  |
| --- |
| @SpringBootApplication //来标注一个主程序类，说明这是一个Spring Boot应用  **public** **class** Springboot01helloApplication {  **public** **static** **void** main(String[] args) {  //spring应用启动  SpringApplication.*run*(Springboot01helloApplication.**class**, args);  }  } |

@SpringBootApplication:Spring Boot应用标注在某类上说明这个类是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；

### SpringBootApplication注解

|  |
| --- |
| @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:Spring Boot的配置类；标注在某个类上，表示这是一个Spring Boot的配置类；

@Configuration:配置类上来标注这个注解；

配置类相当于配置文件配置文件；配置类也是容器中的一个组件；@Component

@EnableAutoConfiguration：开启自动配置功能；

以前我们需要配置的东西，Spring Boot帮我们自动配置；@EnableAutoConfiguration告诉SpringBoot开启自动配置功能；这样自动配置才能生效；

|  |
| --- |
| @Target(ElementType.***TYPE***)  @Retention(RetentionPolicy.***RUNTIME***)  @Documented  @Inherited  @AutoConfigurationPackage  @Import(AutoConfigurationImportSelector.**class**)  **public** **@interface** EnableAutoConfiguration {  String ***ENABLED\_OVERRIDE\_PROPERTY*** = "spring.boot.enableautoconfiguration"; |

@AutoConfigurationPackage 自动配置包

|  |
| --- |
| @Target(ElementType.***TYPE***)  @Retention(RetentionPolicy.***RUNTIME***)  @Documented  @Inherited  @Import(AutoConfigurationPackages.Registrar.**class**)  **public** **@interface** AutoConfigurationPackage {  } |

@Import(AutoConfigurationPackages.Registrar.class):

Spring的底层注解@lmport，给容器中导入一个组件；导入的组件由AutoConfigurationPackages.Registrar.class;

|  |
| --- |
| @Override  **public** **void** registerBeanDefinitions(AnnotationMetadata metadata,  BeanDefinitionRegistry registry) {  *register*(registry, **new** PackageImport(metadata).getPackageName());  } |

将主配置类（@SpringBootApplication标注的类）的所在包及下面所有子包里面的所有组件扫描到Spring容器；

@lmport(EnableAutoConfigurationlmportSelector.class)；给容器中导入组件？

EnableAutoConfigurationlmportSelector:导入哪些组件的选择器；将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中；会给容器中导入非常多的自动配置类（xxxAutoConfiguration)；就是给容器中导入这个场景需要的所有组件，并配置好这些组件；

有了自动配置类，免去了我们手动编写配置注入功能组件等的工作；

SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,classloader);从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值；

将这些值作为自动配置类导入到容器中，自动配置类就生效，帮我们进行自动配置工作；以前我们需要自己配置的东西，自动配置类都帮我们；

## 创建项目

IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目；

选择我们需要的模块；向导会联网创建Spring Boot项目；

默认生成的Spring Boot项目；

·主程序已经生成好了，我们只需要我们自己的逻辑

·resources文件夹中目录结构T

static：保存所有的静态资源；js css images；

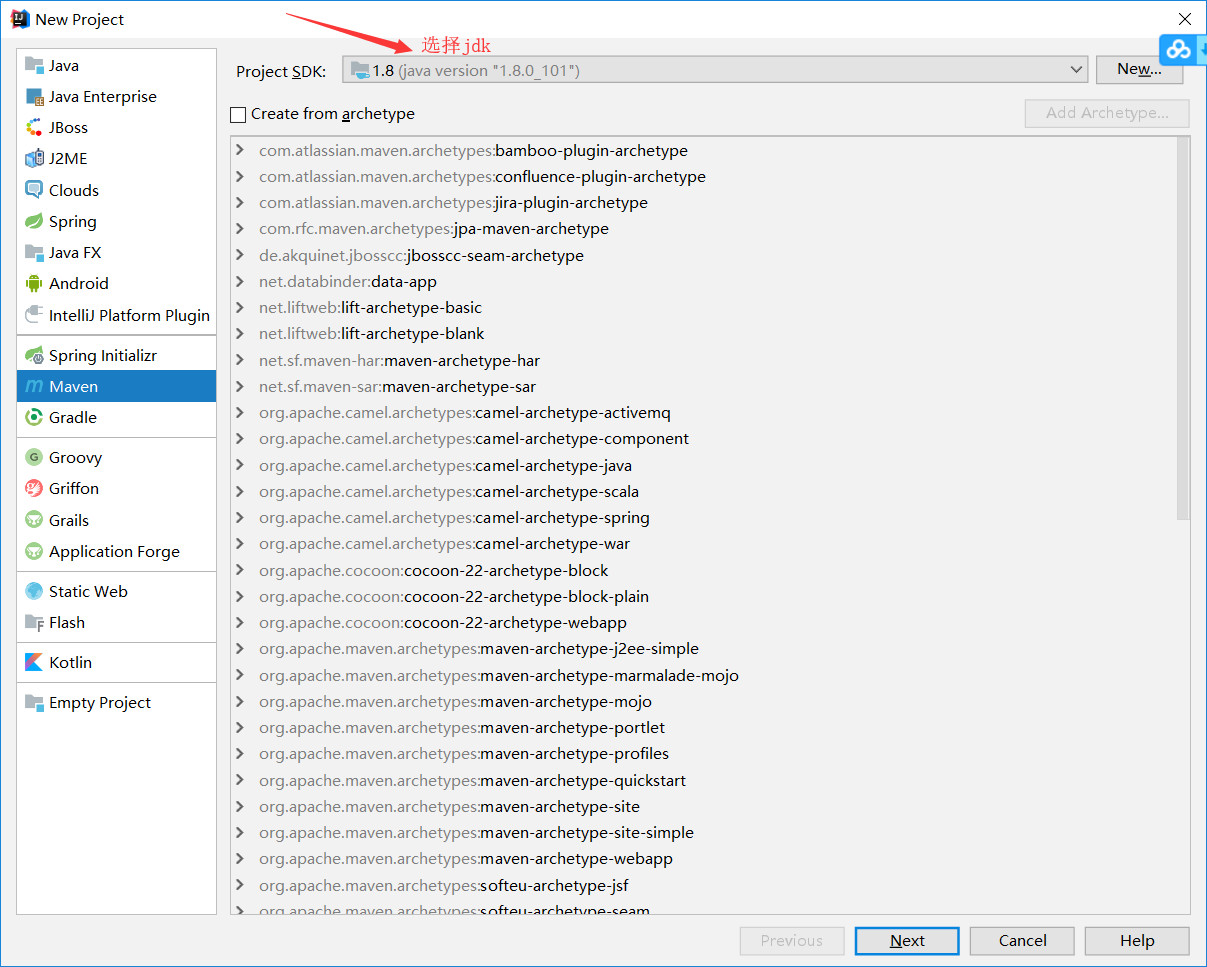
templates：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页面）；可以使用模板引擎（freemarker、thymeleaf）；

oapplication.properties:Spring Boot应用的配置文件；

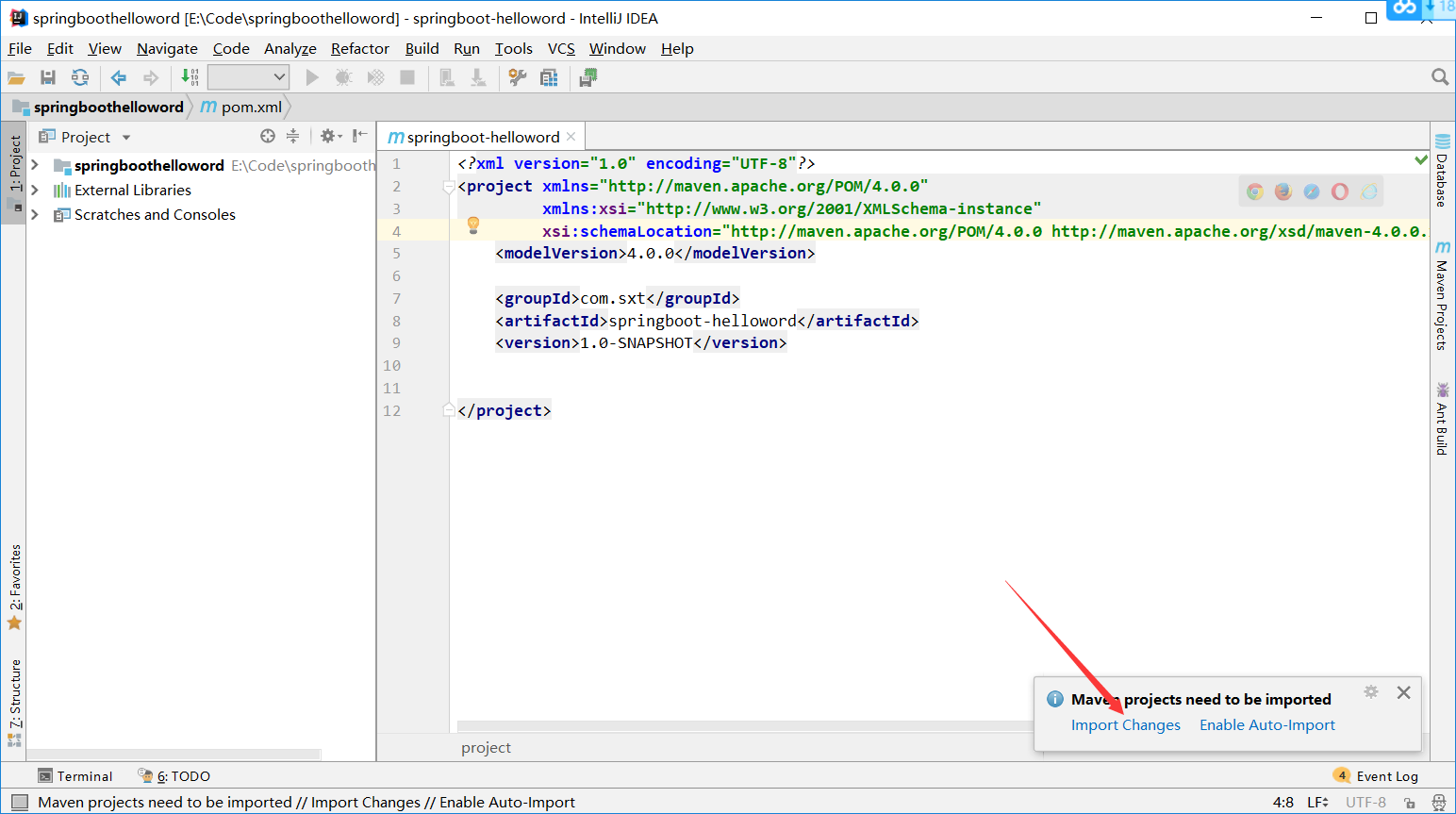
# springboot的helloworld

## 创建项目

### 创建maven项目



### 让idea自动导入依赖



### 导入依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  <**version**>1.5.9.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.9.RELEASE</**version**>  </**dependency**> </**dependencies**> |

### 编写主程序(启动springboot)

|  |
| --- |
| *//来标注一个主程序类，说明这是一个springbooty应用* @SpringBootApplication **public class** SpringBootAPP {  **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootAPP.**class**,args);  } } |

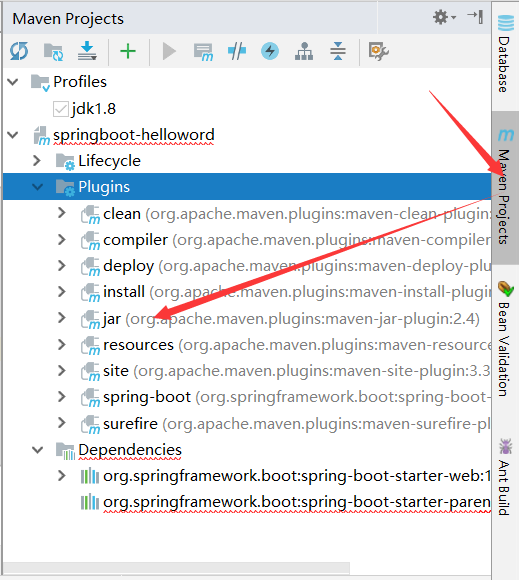
### controller

|  |
| --- |
| @Controller **public class** HelloController {  @RequestMapping(**"/hello"**)  @ResponseBody  **public** String hello(){  **return "hello world"**;  } } |

### 简化部署

|  |
| --- |
| *<!--可以应用程序打包成一个jar包-->* <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**> |

### 打jar包



## helloworld分析

### pom.xml

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.19.RELEASE</**version**> </**parent**> |

依赖的父项目

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-dependencies</**artifactId**>  <**version**>1.5.19.RELEASE</**version**>  <**relativePath**>../../spring-boot-dependencies</**relativePath**> </**parent**> |

他的父项目中：管理所有的依赖，所有的版本都在这个依赖中

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-dependencies</**artifactId**>  <**version**>1.5.19.RELEASE</**version**>  <**relativePath**>../../spring-boot-dependencies</**relativePath**> </**parent**> |

### 导入的依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  <**version**>1.5.9.RELEASE</**version**>  </**dependency**> </**dependencies**> |

spring-boot-starter-web：

spring-boot-starter:spring-boot场景启动器；帮我们导入了web模块正常运行所依赖的组件；而依赖的版本由父项目仲裁。

Spring Boot将所有的功能场景都抽取出来，做成一个个的starters（启动器，只需要在项目里面引入这些starter相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动器。

### 编写主程序(启动springboot)

|  |
| --- |
| *//来标注一个主程序类，说明这是一个springbooty应用* @SpringBootApplication **public class** SpringBootAPP {  **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootAPP.**class**,args);  } } |

@SpringBootApplication:Spring Boot应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；

|  |
| --- |
| @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:Spring Boot的配置类；

标注在某个类上，表示这是一个Spring Boot的配置类；

@Configuration:配置类上来标注这个注解；

配置类---配置文件；配置类也是容器中的一个组件；@Component

@EnableAutoConfiguration:开启自动配置功能；

以前我们需要配置的东西，Spring Boot帮我们自动配置；@EnableAutoConfiguration告诉SpringBoot开启自动配置功能；这样自动配置才能生效；

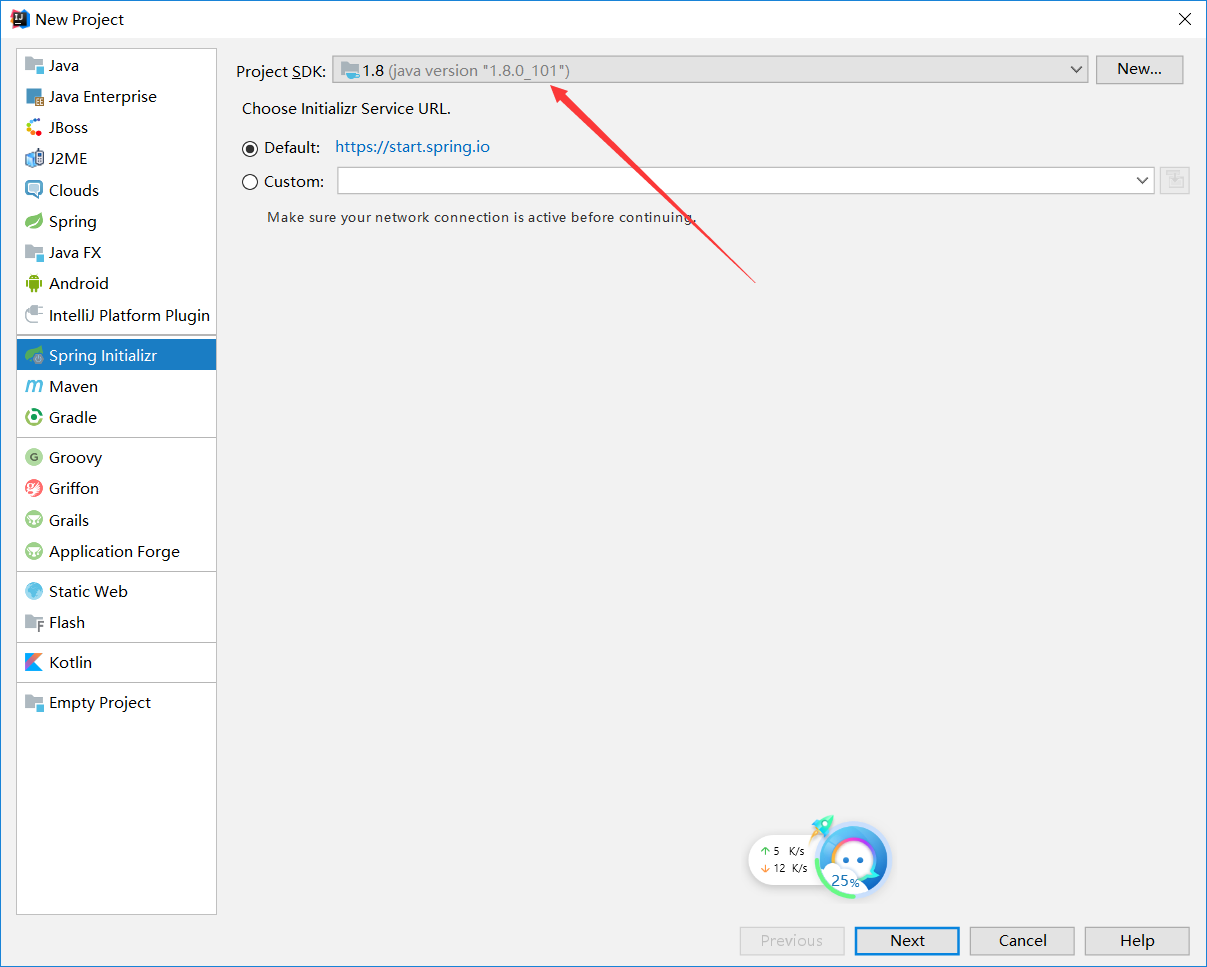
|  |
| --- |
| @AutoConfigurationPackage @Import(EnableAutoConfigurationImportSelector.**class**) **public** @**interface** EnableAutoConfiguration { |

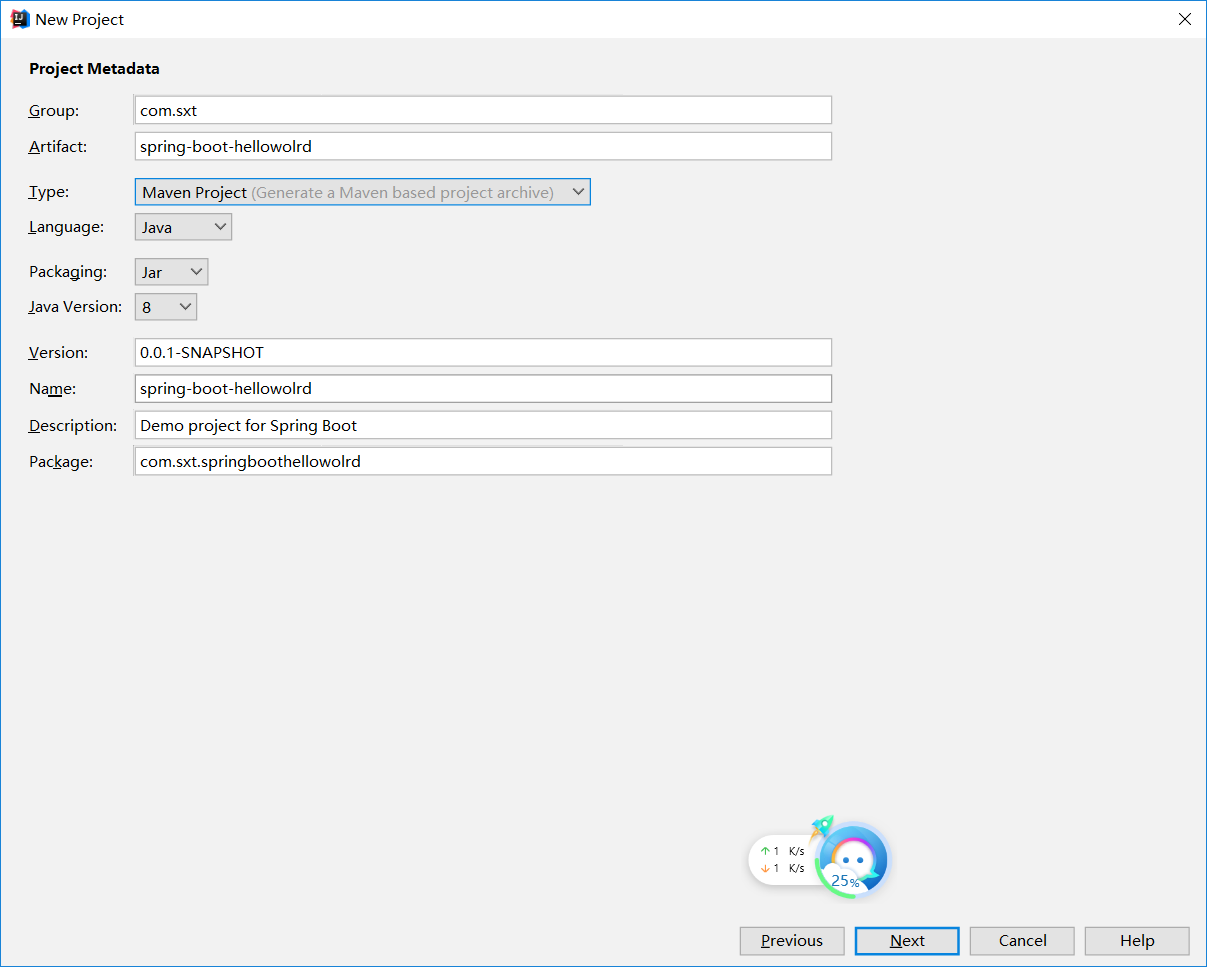
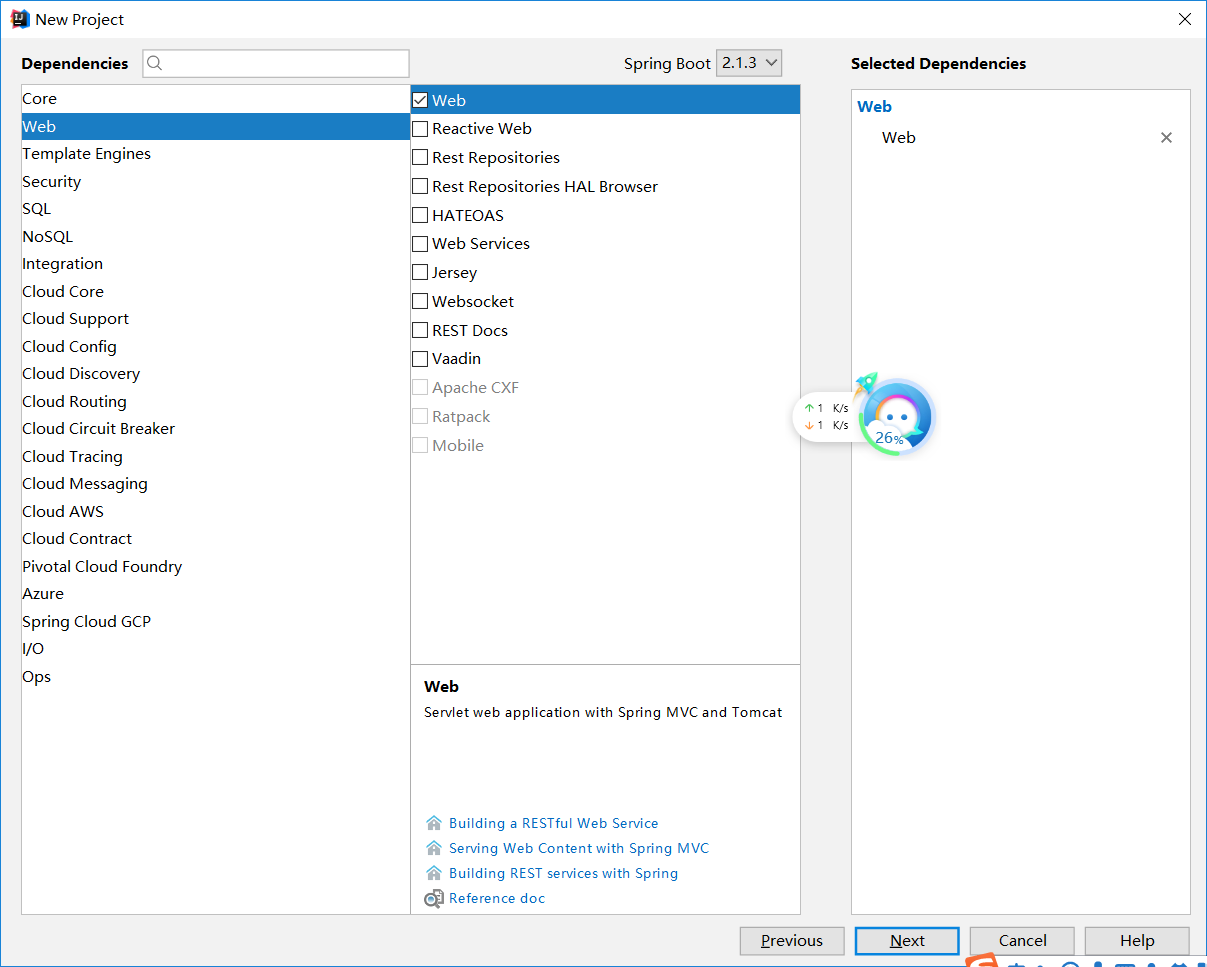
@AutoConfigurationPackage：自动配置包

@lmportAutoConfigurationPackages.Registrar.class):

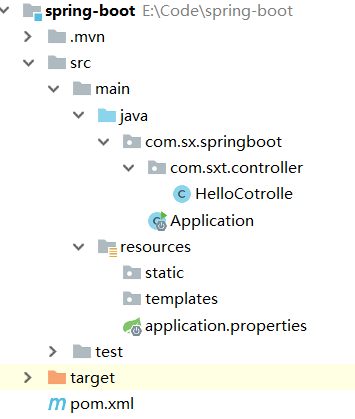
Spring的底层注解@lmport，给容器中导入一个组件；

## 快速创建springboot项目



### 创建好的项目



默认生成的Spring Boot项目；

·主程序已经生成好了，我们只需要我们自己的逻辑

·resources文件夹中目录结构

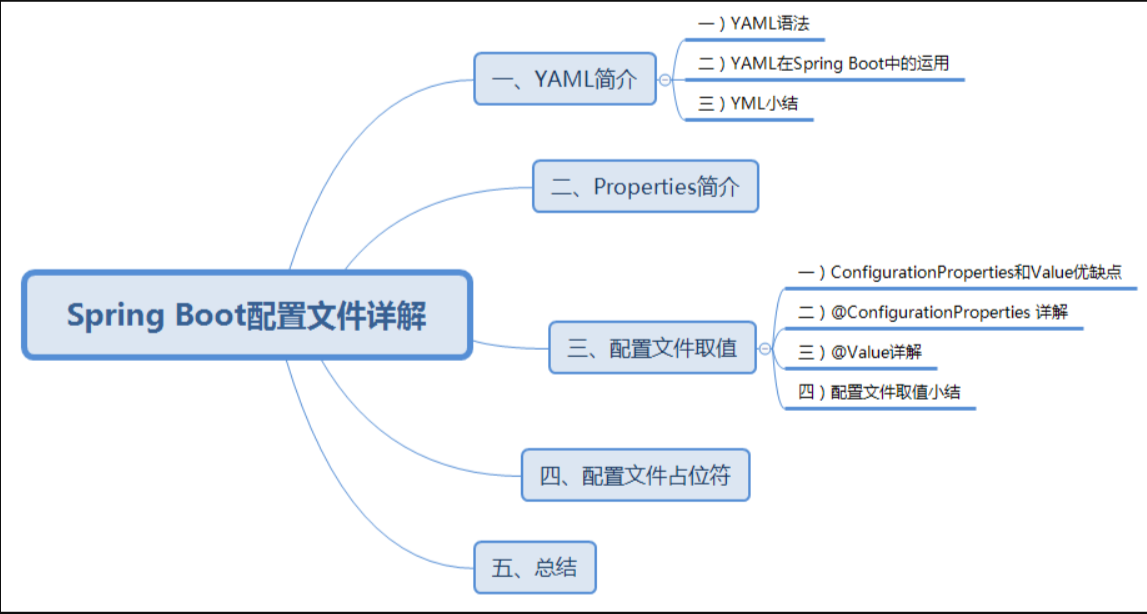
static：保存所有的静态资源；jscss images；

templates：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页面）；可以使用模板引擎（freemarker、thymeleaf）；

application.properties:Spring Boot应用的配置文件；

# springboot的配置文件

Spring Boot提供了两种常用的配置文件，分别是properties文件和yml文件。他们的作用都是修改Spring Boot自动配置的默认值。相对于properties文件而言，yml文件更年轻，也有很多的坑。可谓成也萧何败萧何，yml通过空格来确定层级关系，是配置文件结构跟清晰，但也会因为微不足道的空格而破坏了层级关系



### springboot的配置文件

SpringBoot使用一个全局的配置文件，配置文件名是固定的；

·application.properties

·application:yml

配置文件的作用：修改SpringBoot自动配置的默认值；SpringBoot在底层都给我们自动配置好；

YAML(YAML Ain't Markup Language)

YAML A Markup Language:是一个标记语言

YAML isn't Markup Language:不是一个标记语言；

标记语言：以前的配置文件；大多都使用的是xxxx.xml文件；

YAML：以数据为中心，比json、xml等更适合做配置文件；

### 导入配置文件处理器

导入之后配置文件有提示

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**version**>2.0.8.RELEASE</**version**> </**dependency**> |

## YAML

YAML

yml是YAML（YAML Ain't Markup Language）语言的文件，以数据为中心，比json、xml等更适合做配置文件。yml和xml相比，少了一些结构化的代码，使数据更直接，一目了然。yml和json呢？没有谁好谁坏，合适才是最好的。yml的语法比json优雅，注释更标准，适合做配置文件。json作为一种机器交换格式比yml强，更适合做api调用的数据交换。

### YAML语法

以空格的缩进程度来控制层级关系。空格的个数并不重要，只要左边空格对齐则视为同一个层级。注意不能用tab代替空格。且大小写敏感。支持字面值，对象，数组三种数据结构，也支持复合结构。

例如：**server:  
 port:** 8081  
 **servlet:  
 context-path:** /hello

字面值：字符串，布尔类型，数值，日期。字符串默认不加引号，单引号会转义特殊字符。日期格式支持yyyy/MM/dd HH:mm:ss

对象：由键值对组成，形如 key:(空格)value 的数据组成。冒号后面的空格是必须要有的，每组键值对占用一行，且缩进的程度要一致，也可以使用行内写法：{k1: v1, ....kn: vn}

数组：由形如 -(空格)value 的数据组成。短横线后面的空格是必须要有的，每组数据占用一行，且缩进的程度要一致，也可以使用行内写法： [1,2,...n]

复合结构：上面三种数据结构任意组合

### YAML的运用

创建一个Spring Boot 的全局配置文件 application.yml，配置属性参数。主要有字符串，带特殊字符的字符串，布尔类型，数值，集合，行内集合，行内对象，集合对象这几种常用的数据格式。

|  |
| --- |
| yaml:  str: 字符串可以不加引号  specialStr: "双引号直接输出\n特殊字符"，双引不会转义特殊字符  specialStr2: '单引号可以转义\n特殊字符' 单引号不会转义特殊字符  flag: false  num: 666  Dnum: 88.88  list:  - one  - two  - two  set: [1,2,2,3]  map: {k1: v1, k2: v2}  positions:  - name: ITDragon  salary: 15000.00  - name: ITDragonBlog  salary: 18888.88 |

创建实体类YamlEntity.java 获取配置文件中的属性值， 通过注解@ConfigurationProperties获取配置文件中的指定值并注入到实体类中 。其具体的测试方法和获取值的原理，请继续往后看！

|  |
| --- |
| **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Set;  /\*\*  \* YAML 语法实体类  \* 切记点：  \* 一、冒号后面加空格，即 key:(空格)value  \* 二、每行参数左边空格数量决定了该参数的层级，不可乱输入。  \*/  //将配置文件中配置的每一个属性，映射到这个组件，且这个组件必须在容器中  //prefix=“yaml” 配置文件中那个下面的属性一一映射。  @Component  @ConfigurationProperties(prefix = "yaml")  **public** **class** YamlEntity {    // 字面值，字符串，布尔，数值  **private** String str; // 普通字符串  **private** String specialStr; // 转义特殊字符串  **private** String specialStr2;// 输出特殊字符串  **private** Boolean flag; // 布尔类型  **private** Integer num; // 整数  **private** Double dNum; // 小数    // 数组，List和Set，两种写法： 第一种：-空格value，每个值占一行，需缩进对齐；第二种：[1,2,...n] 行内写法  **private** List<Object> list; // list可重复集合  **private** Set<Object> set; // set不可重复集合    // Map和实体类，两种写法：第一种：key空格value，每个值占一行，需缩进对齐；第二种：{key: value,....} 行内写法  **private** Map<String, Object> map; // Map K-V  **private** List<Position> positions; // 复合结构，集合对象    // 省略getter，setter，toString方法  } |

### 例如：配置端口号

传统的xml配置

<server>

<port>8081</port>

</server>

使用yaml配置

server:

port: 8081

### 校验

|  |
| --- |
| @Component @Validated **public class** User {  *//从配置文件中获取值* @Value(**"${persion.name}"**)  **private** String **name**;   @Value(**"${persion.id}"**)  **private** Integer **id**;  @Email  @Value(**"${persion. gender}"**)  **private String gender**; } |

yml支持校验，如果给**gender写的是非emain值会报错**

### YML小结

一、字符串可以不加引号，若加双引号则输出特殊字符，若不加或加单引号则转义特殊字符；

二、数组类型，短横线后面要有空格；对象类型，冒号后面要有空格；

三、YAML是以空格缩进的程度来控制层级关系，但不能用tab键代替空格，大小写敏感；

四、如何让一个程序员崩溃？在yml文件中加几个空格！(〃＞皿＜)

## Properties

### properties

properties文件大家经常用，这里就简单介绍一下。其语法结构形如：key=value。注意中文乱码问题，需要转码成ASCII。具体如下所示：

|  |
| --- |
| userinfo.account=itdragonBlog  userinfo.age=25  userinfo.active=true  userinfo.created-date=2018/03/31 16:54:30  userinfo.map.k1=v1  userinfo.map.k2=v2  userinfo.list=one,two,three  userinfo.position.name=Java架构师  userinfo.position.salary=19999.99 |

从配置文件中取值注入到实体类中，和YAML是一样的。

|  |
| --- |
| **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  **import** java.util.Date;  **import** java.util.List;  **import** java.util.Map;  /\*\*  \* 用户信息  \* **@ConfigurationProperties** : 被修饰类中的所有属性会和配置文件中的指定值（该值通过prefix找到）进行绑定  \*/  @Component  @ConfigurationProperties(prefix = "userinfo")  **public** **class** UserInfo {    **private** String account;  **private** Integer age;  **private** Boolean active;  **private** Date createdDate;  **private** Map<String, Object> map;  **private** List<Object> list;  **private** Position position;    // 省略getter，setter，toString方法  } |

### 配置文件的占位符

1、随机数

RandomValuePropertySource:配置文件中可以使用随机数

${random.value}、${random.int)、${random.long}

${random.int(10)}、${random.int[1024,65536]}

2、占位符获取之前配置的值，如果没有可以是用：指定默认值

person.name=张三${random.int} #加一个随机数

person.age=${random.uuid} #加一个uuid

person.dog.name=${person.name} #引用其他的变量

person.dog.age=${person:haha} //如果没有persion，则选择haha

## 配置文件取值

Spring Boot通过ConfigurationProperties注解从配置文件中获取属性。从上面的例子可以看出ConfigurationProperties注解可以通过设置prefix指定需要批量导入的数据。支持获取字面值，集合，Map，对象等复杂数据。ConfigurationProperties注解还有其他特么呢？

### ConfigurationProperties和Value优缺点

ConfigurationProperties注解的优缺点

一、可以从配置文件中批量注入属性；

二、支持获取复杂的数据类型；

三、对属性名匹配的要求较低，比如user-name，user\_name，userName，USER\_NAME都可以取值；

四、支持JAVA的JSR303数据校验；

五、缺点是不支持强大的SpEL表达式

Value注解的优缺点正好相反，它只能一个个配置注入值；不支持数组、集合等复杂的数据类型；不支持数据校验；对属性名匹配有严格的要求。最大的特点是支持SpEL表达式，使其拥有更丰富的功能。

### @ConfigurationProperties详解

第一步：导入依赖。若要使用ConfigurationProperties注解，需要导入依赖 spring-boot-configuration-processor；

第二步：配置数据。在application.yml配置文件中，配置属性参数，其前缀为itdragon，参数有字面值和数组，用来判断是否支持获取复杂属性的能力；

第三步：匹配数据。在类上添加注解ConfigurationProperties，并设置prefix属性值为itdragon。并把该类添加到Spring的IOC容器中。

第四步：校验数据。添加数据校验Validated注解，开启数据校验，测试其是否支持数据校验的功能；

第五步：测试ConfigurationProperties注解是否支持SpEL表达式；

导入依赖：pom.xml 添加 spring-boot-configuration-processor依赖

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

配置数据：application.yml 配置属性参数，nick-name是用来判断匹配属性的松散性，若换成nick\_name依然可以获取值。

|  |
| --- |
| itdragon:   nick-name: ITDragonBlog   email: 1234567890@qq.com   iphone: 1234567890   abilities: [java, sql, html]  created\_date: 2018/03/31 15:27:30 |

匹配和校验数据：

|  |
| --- |
| **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  **import** org.springframework.validation.annotation.Validated;  **import** javax.validation.constraints.Email;  **import** java.util.Date;  **import** java.util.List;  /\*\*  \* ConfigurationProperties 注解语法类  \* 第一步：导入依赖 spring-boot-configuration-processor；  \* 第二步：把ConfigurationProperties注解修饰的类添加到Spring的IOC容器中；  \* 第三步：设置prefix属性，指定需要注入属性的前缀；  \* 第四步：添加数据校验注解，开启数据校验；  \*  \* 注意点：  \* 一、nickName和createdDate在yml配置文件中，对应参数分别是中划线和下划线，用于测试其对属性名匹配的松散性  \* 二、email和iphone 测试其支持JSR303数据校验  \* 三、abilities 测试其支持复杂的数据结构  \*/  @Component  @ConfigurationProperties(prefix = "itdragon")  @Validated  **public** **class** ConfigurationPropertiesEntity {    **private** String nickName; // 解析成功，支持松散匹配属性  **private** String email;  // @Email // 解析失败，数据校验成功：BindValidationException: Binding validation errors on itdragon  **private** String iphone;  **private** List<String> abilities;  **private** Date createdDate; // 解析成功，支持松散匹配属性    // @ConfigurationProperties("#{(1+2-3)/4\*5}")  **private** String operator; // 语法报错，不支持SpEL表达式：not applicable to field    // 省略getter，setter，toString方法  } |

### 配置文件取值小结

一、ConfigurationProperties注解支持批量注入，而Value注解适合单个注入；

二、ConfigurationProperties注解支持数据校验，而Value注解不支持；

三、ConfigurationProperties注解支持松散匹配属性，而Value注解必须严格匹配属性；

四、ConfigurationProperties不支持强大的SpEL表达式，而Value支持；

## 配置文件占位符

占位符和随机数比较简单，这里就直接贴出代码。需要注意的是：

一、占位符的值必须是完整路径

二、占位符设置默认值，冒号后面不能有空格

|  |
| --- |
| ran: # 这里的prefix不能是random，   ran-value: ${random.value}   ran-int: ${random.int}   ran-long: ${random.long}   ran-int-num: ${random.int(10)}   ran-int-range: ${random.int[10,20]}  ran-placeholder: placeholder\_${ran.ran-value:此处不能有空格，且key为完整路径} |

SpringBootYmlApplicationTests

|  |
| --- |
| **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;    /\*\*  \* 随机数和占位符语法类  \*/  @Component  @ConfigurationProperties(prefix = "ran")  **public** **class** RandomEntity {    **private** String ranValue; // 随机生成一个字符串  **private** Integer ranInt; // 随机生成一个整数  **private** Long ranLong; // 随机生成一个长整数  **private** Integer ranIntNum; // 在指定范围内随机生成一个整数  **private** Integer ranIntRange;// 在指定区间内随机生成一个整数  **private** String ranPlaceholder;// 占位符    // 省略getter，setter，toString方法e  }  测试代码：    @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** SpringBootYmlApplicationTests {    @Autowired  **private** UserInfo userInfo;  @Autowired  **private** YamlEntity yamlEntity;  @Autowired  **private** ConfigurationPropertiesEntity configurationPropertiesEntity;  @Autowired  **private** ValueEntity valueEntity;  @Autowired  **private** RandomEntity randomEntity;    @Test  **public** **void** contextLoads() {  // System.out.println("YAML Grammar : " + yamlEntity);  // System.out.println("UserInfo : " + userInfo);  // System.out.println("ConfigurationProperties Grammar : " + configurationPropertiesEntity);  // System.out.println("Value Grammar : " + valueEntity);  System.out.println("Random Grammar : " + randomEntity);  }  } |

## 总结

一、Spring Boot 支持两种格式的配置文件，其中YAML的数据结构比properties更清晰。

二、YAML 是专门用来写配置文件的语言，非常简洁和强大。

三、YAML 对空格的要求很严格，且不能用Tab键代替。

四、YAML 通过空格缩进的程度确定层级，冒号后面有空格，短横线后面有空格。

五、ConfigurationProperties注解适合批量注入配置文件中的属性，Value注解适合获取配置文件中的某一项。

六、ConfigurationProperties注解支持数据校验和获取复杂的数据，Value注解支持SpEL表达式。

以上就是本文的全部内容，希望对大家的学习有所帮助，也希望大家多多支持脚本之家。

## 配置文件的加载顺序

### 配置文件的加载顺序

spring boot 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

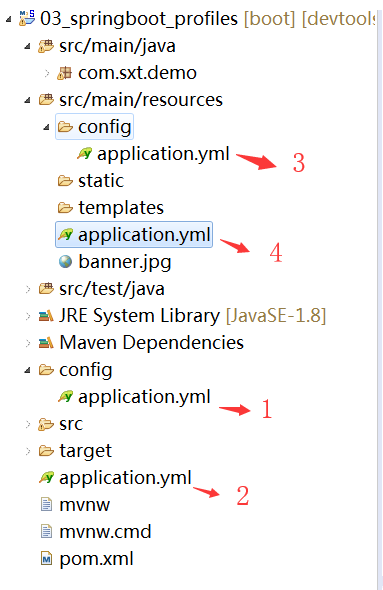
其中同一目标下的properties文件的优先级大于yml文件

file:./config/ (当前项目路径config目录下);

file:./ (当前项目路径下);

classpath:/config/ (类路径config目录下);

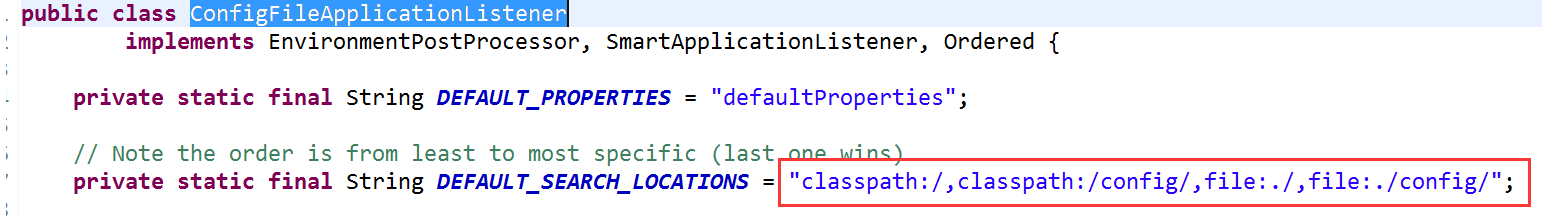
classpath:/ (类路径config下).



以上是按照优先级从高到低的顺序，所有位置的文件都会被加载，高优先级配置内容会覆盖低优先级配置内容。

SpringBoot会从这四个位置全部加载主配置文件，如果高优先级中配置文件属性与低优先级配置文件不冲突的属性，则会共同存在---互补配置。

我们可以从ConfigFileApplicationListener这类便可看出，其中DEFAULT\_SEARCH\_LOCATIONS属性设置了加载的目录：



### 外部配置加载顺序

SpringBoot也可以从以下位置加载配置；优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置，会形成互补配置

1.命令行参数

例如：java -jar -jar包 --server.port=8081 –server.context-path=/项目名

2.来自java:comp/envJNDI属性

3Java系统属性（System.getProperties0)

4.操作系统环境变量

5.RandomValuePropertySource配置的random.\*属性值

1、有jar包外向jar包内寻找

2、优先加载带profile

优先加载带Profile的

6.jar包外部的application-(profile}.properties或application.yml(带spring.profile)配置文件。

7.jar包内部的application-(profile}-properties或application.yml(带spring.profile)配置文件

再来加载不带prifile的

8.jar包外部的application.properties或application.yml不带spring.profile)配置文件

9.jar包内部的application.properties或application.yml不带spring.profile)配置文件

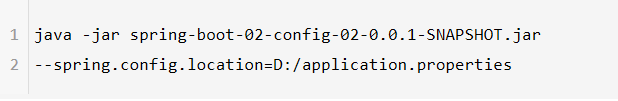
10.@Configuration注解类上的@PropertySource

11.通过SpringApplication.setDefaultProperties指定的默认属性

### 外部的配置文件

1，使用配置文件的路径

我们也可以通过配置spring.config.location来改变默认配置。

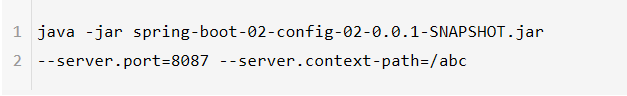


项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置。指定配置文件和默认加载的这些配置文件共同起作用形成互补配置。

2，使用命令行参数

所有的配置都可以在命令行上进行指定；

多个配置用空格分开； –配置项=值



## spingboot自动配置原理（重点）

自动配置原理：

1、SpringBoot启动的时候加载主配置类，开启了自动配置功能

2、@EnableAutoConfiguration作用：

利用EnableAutoconfigurationlmportSelector给容器中导入一些组件？

可以插件selectlmports()方法的内容；

获取候选的配置：

|  |
| --- |
| List<String> configurations = **this**.getCandidateConfigurations(annotationMetadata, attributes); |

扫描所有jar包类路径下的META-INF/spring.factories,将扫描到这些文件的内容包装成properties对象。从properties获取到enableautoConfiguration.class对应的值添加到容器中。

|  |
| --- |
| List<String> configurations = SpringFactoriesLoader.loadFactoryNames(**this**.getSpringFactoriesLoaderFactoryClass(), **this**.getBeanClassLoader()); |

结：将雷路惊吓META-IINF/spring.factoryies里面配置的所有enableautoconfiguration的值加入到了容器中。

例如：

|  |
| --- |
| *# Test Execution Listeners* **org.springframework.test.context.TestExecutionListener**=**\ org.springframework.boot.test.autoconfigure.restdocs.RestDocsTestExecutionListener,\ org.springframework.boot.test.autoconfigure.web.client.MockRestServiceServerResetTestExecutionListener,\ org.springframework.boot.test.autoconfigure.web.servlet.MockMvcPrintOnlyOnFailureTestExecutionListener,\ org.springframework.boot.test.autoconfigure.web.servlet.WebDriverTestExecutionListener** |

每一个这样的xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；

3)每一个自动配置类进行自动配置功能

以HttpEncodingAutoConfiguration解释原理（http编码自动配置）

|  |
| --- |
| @Configuration //表示这是 一个配置类，以前编写的配置文件一样，也可以为容器添加组件。 @EnableConfigurationProperties({HttpProperties.**class**})//启动指定类的ConfigurationProperties的功能，将配置文件中对应的值和httpencodingproperties绑定起来。并把 @ConditionalOnWebApplication(//spring底层@conditionnal注解，根据不同的条件，如果满足某个条件，整个配置类里面的配置就全部生效。判断当前应用是否是个web应用。如果是，当前配置生效。  type = Type.SERVLET ) @ConditionalOnClass({CharacterEncodingFilter.**class**})//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。 @ConditionalOnProperty(//判断配置文件中是否存在**"spring.http.encoding. enabled "**，若果不存在，判断也是成立的。即使配置文件中不配置**"spring.http.encoding. enabled "，也是默认生效的。**  prefix = **"spring.http.encoding"**,  value = {**"enabled"**},  matchIfMissing = **true** ) **public class** HttpEncodingAutoConfiguration {  **//这个值已经和springboot的配置文件映射了。使用**  **private final** Encoding properties;  //只有一个构造器的情况下，参数的值就会从容器中拿  **public** HttpEncodingAutoConfiguration(HttpProperties properties) {  **this**.properties = properties.getEncoding();  }  @Bean //跟容器添加一个组件。这个组件的某些值需要从某properties中获取 @ConditionalOnMissingBean **public** CharacterEncodingFilter characterEncodingFilter() {  CharacterEncodingFilter filter = **new** OrderedCharacterEncodingFilter();  filter.setEncoding(**this**.properties.getCharset().name());  filter.setForceRequestEncoding(**this**.properties.shouldForce(org.springframework.boot.autoconfigure.http.HttpProperties.Encoding.Type.REQUEST));  filter.setForceResponseEncoding(**this**.properties.shouldForce(org.springframework.boot.autoconfigure.http.HttpProperties.Encoding.Type.RESPONSE));  **return** filter; } |

总结：根据当前不同的调价你判断，决定这个配置类是否生效。一旦这个配置类生效，这个配置类就会个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。

properties：这是能配置的属性都是来来自于这个功能的properties类

|  |
| --- |
| **spring.http.encoding.charset**=**utf-8 spring.http.encoding.force-request**=**true spring.http.encoding.enabled**=**true** |

5、HttpProperties:所有在配置文件中能够配置的属性xxxxproperties类中封装着，配置文件中能够配置什么，就可以参照某个对应的这个属性类。

|  |
| --- |
| @ConfigurationProperties( //  prefix = **"spring.http"** )从配置问价中获取指定的和bean的属性进行绑定。例如 **public class** HttpProperties { |

### 总结

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SpringBoot默认写好的自动配置类；

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有，我们就不需要再来配置

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。我们就可以在配置文件中指定这些属性的值：

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；

## @conditional的派生



自动配置类必须在一定的条件下才能生效；

我们怎么知道哪些自动配置类生效；

我们可以通过启用debig=true 打印启用配置类是否开启。

# springboot与日志

## 日志框架

### 引入

开发一个大型系统：

1、system.out.printfln(“”) 将关数据打印子在控制台，写到一个文件中。

2、框架来记录系统的一些运行信息；日志框架；xxx.jar

3、异步模式、自动归档？xxx？

4、将以前的框架卸下来，换上新的框架，重新修改之前相关的apixxx.jar

5、写了一个统一的接口层：入职门面(日志的一个抽象层)xxx.jar

给项目中添加入职实现就行了。

### 常见的日志框架

市场上存在非常多的日志框架。JUL（java.util.logging)，JCL(ApacheCommons Logging)，Log4j，Log4j2,Logback、SLF4j、jboss-logging等。Spring Boot在框架内容部使用JCL，spring-boot-starter-logging采用了sif4j+logback的形式，Spring Boot也能自动适配（jul、log4j2、logback)并简化配置



左边选一个接口，右边选一个实现。

springboot选择

日志门面SLF4J+

日志实现：logback

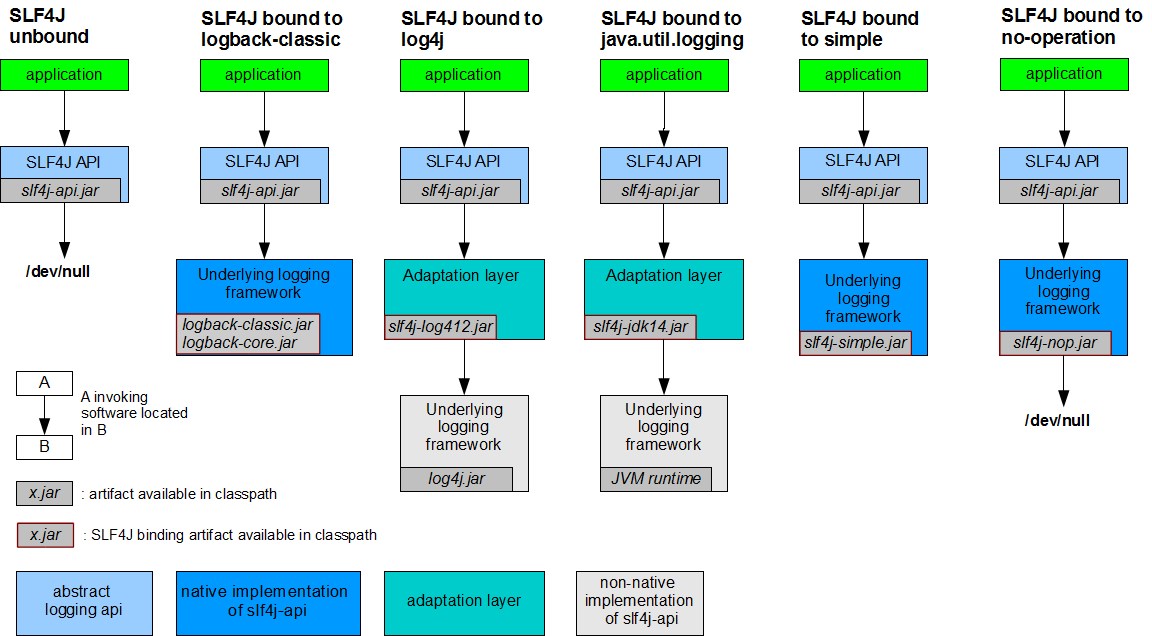
## SLF4J使用

### SLF4J的使用

以后开发的时候，日志记录方法的调用，不应该来直接调用日志的实现类，而是调用日志抽象层里面的方法；给系统里面导入slf4j的jar和logback的实现jar

|  |
| --- |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class HelloWorld {  public static void main(String[] args) {  Logger logger = LoggerFactory.getLogger(HelloWorld.class);  logger.info("Hello World");  }  } |

使用slf4j引用jar包：每一个日志的实现框架都有自己的配置文件。使用slf4j以后，配置文件还是做成日志实现框架的配置文件；



### 问题

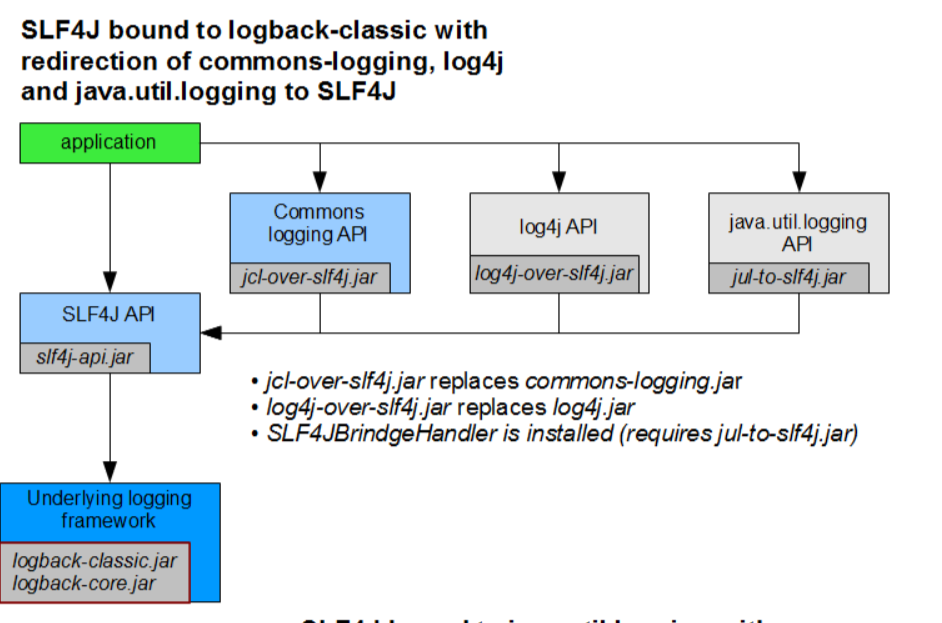
a（slf4j+logback):Spring(commons-logging)、Hibernate(jboss-logging)、MyBatis、xxxx

统一日志记录，即使是别的框架和我一起统一使用slf4j框架输出?

1、将系统中其他日志框架排除出去。

2、用中间包来替换所有的日志框架。

3、再导入slf4j其他的实现。



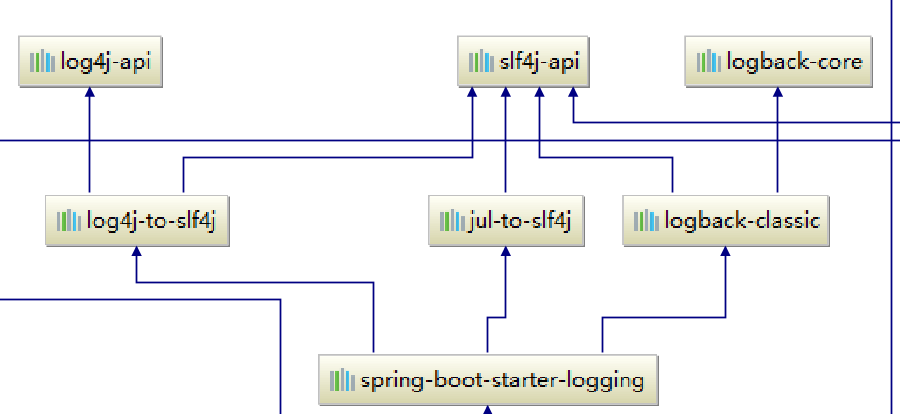
## springboot的日志

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter</**artifactId**>  <**version**>2.1.3.RELEASE</**version**>  <**scope**>compile</**scope**> </**dependency**> |

springboot使用他来做日志功能

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  <**version**>2.1.3.RELEASE</**version**>  <**scope**>compile</**scope**> </**dependency**> |

springboot底层依赖关系： 底层使用slf4j



总结：

1）、SpringBoot底层也是使用slf4j+logback的方式进行日志记录

2）、SpringBoot也把其他的日志都替换成了slf4j；

3）、中间替换包？

4）、如果我们要引入其他框架？一定要把这个框架的默认日志依赖移除掉？

Spring框架用的是commons-logging；

总结：SpringBoot能自动适配所有的日志，而且底层使用slf4j+logback的方式记录日志，引入其他框架的时候，只需要把这个框架依赖的日志框架排除掉；

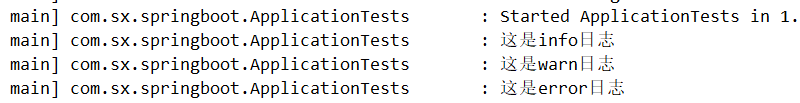


## springboot的使用

### 测试

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** ApplicationTests {  *//日志记录记录器* Logger **logger** = LoggerFactory.*getLogger*(getClass());  @Test  **public void** contextLoads() {  *//日志级别，由低到高，trace<debug<info<warn<error  //可以调整输出日志的级别;日志就会在这个级别以后的高级别生效  //springboot默认给我们调整的是info级别，可以在配置文件中调整日志的级别。  //跟踪轨迹* **logger**.trace(**"这是trace日志"**);  *//debug* **logger**.debug(**"这是dubug信息"**);  *//info日志* **logger**.info(**"这是info日志"**);  *//警告日志* **logger**.warn(**"这是warn日志"**);  *//error日志* **logger**.error(**"这是error日志"**);  } } |

结果：只会输出info级别以上的



### 配置文件中设置日志级别



注:logging.file和logging.path如果同时存在，有显示用logging.file

|  |
| --- |
| **logging:** *#在当前的项目下生成spring.log日志文件  #file: spring.log  #在指定位置下生成日志文件* **file:** D:/spring.log  *#在当前磁盘的根路径下创建spring文件夹和和里面的log文件夹，使用spring.log作为默认文件* **path:** /spring/log  **pattern:** *#可以自动以日志格式  #在可控制台输出的日志格式* **console:** *#在指定的文件中输出的格式* **file:** |

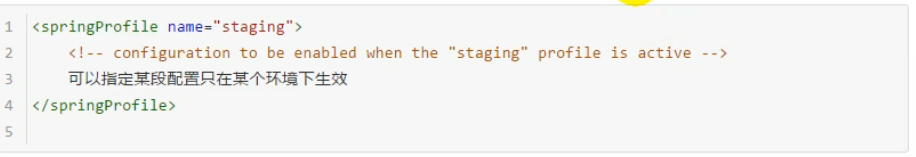
### 使用自己的日志配置文件

给类路径下放上每个日志框架自己的配置文件即可；SpringBoot就不使用他默认配置的了



logback.xml：直接就被日志框架识别了；

logback-spring.xml：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用springboot的高级功能。



# profile环境支持

## 什么是profile

举个例子。一般在开发项目的时候要有多个环境，如开发环境、测试环境、生产环境，他们的配置文件一般不同。当我们要向各个环境发布程序时，需要人工处理这些配置文件，这显然麻烦且易错。有了profile，一切问题就简单了

简单讲profile就是一组配置，不同profile提供不同组合的配置，程序运行时可以选择使用哪些profile来适应环境。

对于多环境的配置，各种项目构建工具或是框架的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot也不例外，或者说更加简单。

## 多环境配置文件

在Spring Boot中多环境配置文件名需要满足application-{profile}.properties的格式，其中{profile}对应你的环境标识，比如：

- application-dev.properties：开发环境

- application-test.properties：测试环境

- application-pro.properties：生产环境

至于哪个具体的配置文件会被加载，需要在application.properties文件中通过spring.profiles.active属性来设置，其值对应{profile}值。

如：spring.profiles.active=test就会加载application-test.properties配置文件内容

## 普通方式

### 创建application-dev.properties

|  |
| --- |
| **server.port**=**8081** |

### 创建application-pro.properties

|  |
| --- |
| **server.port**=**8082** |

### 修改application.properties

|  |
| --- |
| **spring.profiles.active**=**dev //指定激活** |

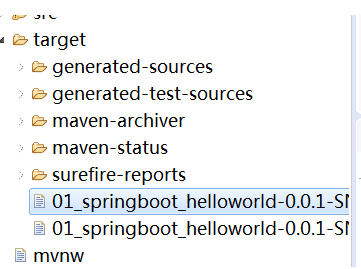
### 运行测试

运行时指定使用的是8081

|  |
| --- |
| @SpringBootApplication **public class** SpringBoot01HelloworldApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(SpringBoot01HelloworldApplication.**class**, args);  } } |

## 去掉application.properties的jar包运行方式

### 打包



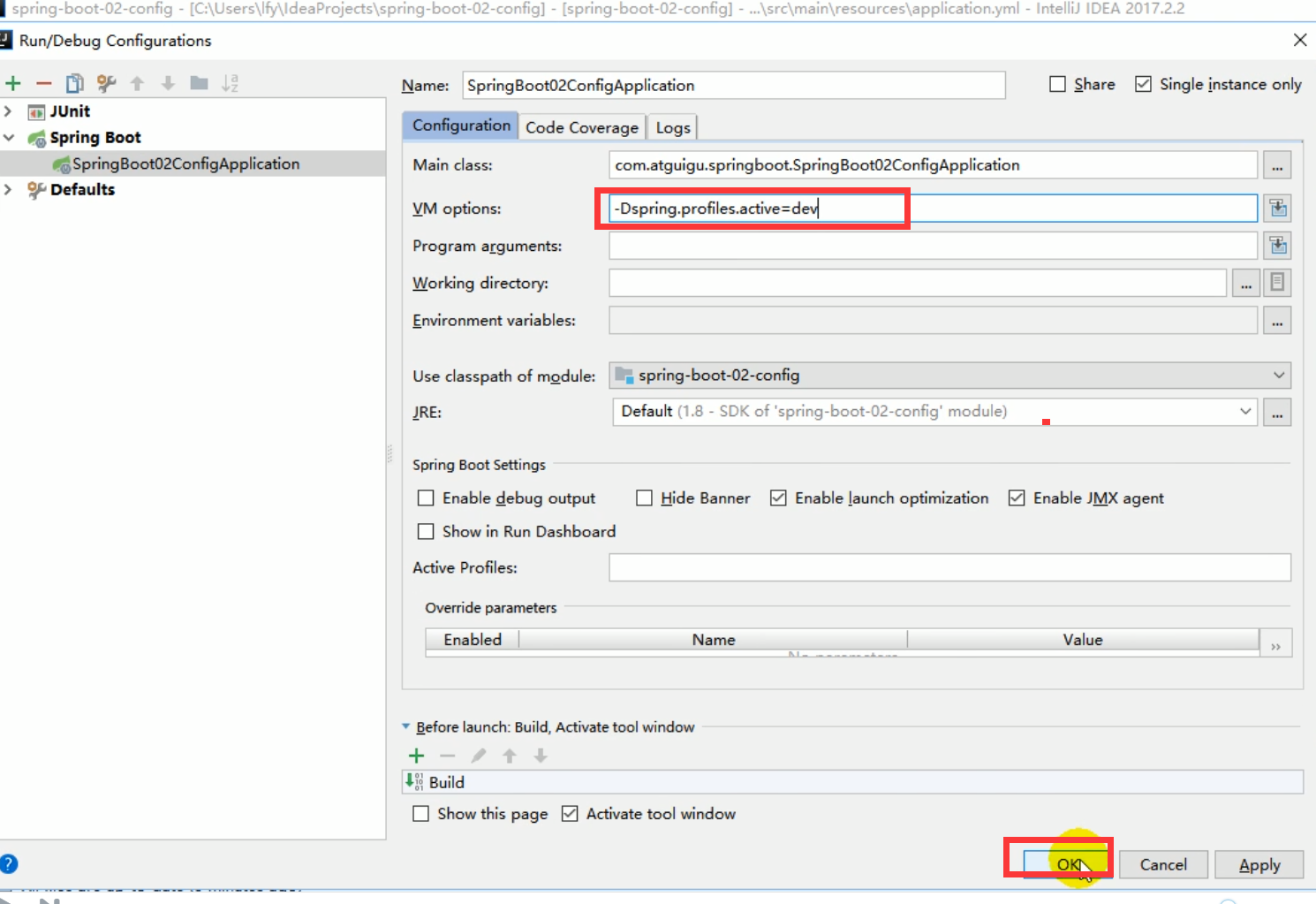
D:\>java -jar 01\_springboot\_helloworld-0.0.1-SNAPSHOT.jar --spring.profiles.acti值ve=pro

注：这种方式的优先级最高

### java -jar运行



## 虚拟机参数



## yml文件的多配置块的集成切换

在yml中只需要使用一个yml文件

|  |
| --- |
| spring:  profiles:  active: pro 激生产环境环境，激活之后使用的该对应文档快中的内容  ---  server:  port: 8081  spring:  profiles: dev  ---  server:  port: 8082  spring:  profiles: pro |

# 注解

## 配置文件取值

Spring Boot通过ConfigurationProperties注解从配置文件中获取属性。从上面的例子可以看出ConfigurationProperties注解可以通过设置prefix指定需要批量导入的数据。支持获取字面值，集合，Map，对象等复杂数据。ConfigurationProperties注解还有其他特么呢？

### ConfigurationProperties和Value优缺点

ConfigurationProperties注解的优缺点

一、可以从配置文件中批量注入属性；

二、支持获取复杂的数据类型；

三、对属性名匹配的要求较低，比如user-name，user\_name，userName，USER\_NAME都可以取值；

四、支持JAVA的JSR303数据校验；

五、缺点是不支持强大的SpEL表达式

Value注解的优缺点正好相反，它只能一个个配置注入值；不支持数组、集合等复杂的数据类型；不支持数据校验；对属性名匹配有严格的要求。最大的特点是支持SpEL表达式，使其拥有更丰富的功能。

### @ConfigurationProperties详解

第一步：导入依赖。若要使用ConfigurationProperties注解，需要导入依赖 spring-boot-configuration-processor；

第二步：配置数据。在application.yml配置文件中，配置属性参数，其前缀为itdragon，参数有字面值和数组，用来判断是否支持获取复杂属性的能力；

第三步：匹配数据。在类上添加注解ConfigurationProperties，并设置prefix属性值为itdragon。并把该类添加到Spring的IOC容器中。

第四步：校验数据。添加数据校验Validated注解，开启数据校验，测试其是否支持数据校验的功能；

第五步：测试ConfigurationProperties注解是否支持SpEL表达式；

导入依赖：pom.xml 添加 spring-boot-configuration-processor依赖

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

配置数据：application.yml 配置属性参数，nick-name是用来判断匹配属性的松散性，若换成nick\_name依然可以获取值。

|  |
| --- |
| itdragon:   nick-name: ITDragonBlog   email: 1234567890@qq.com   iphone: 1234567890   abilities: [java, sql, html]  created\_date: 2018/03/31 15:27:30 |

匹配和校验数据：

|  |
| --- |
| **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  **import** org.springframework.validation.annotation.Validated;  **import** javax.validation.constraints.Email;  **import** java.util.Date;  **import** java.util.List;  /\*\*  \* ConfigurationProperties 注解语法类  \* 第一步：导入依赖 spring-boot-configuration-processor；  \* 第二步：把ConfigurationProperties注解修饰的类添加到Spring的IOC容器中；  \* 第三步：设置prefix属性，指定需要注入属性的前缀；  \* 第四步：添加数据校验注解，开启数据校验；  \*  \* 注意点：  \* 一、nickName和createdDate在yml配置文件中，对应参数分别是中划线和下划线，用于测试其对属性名匹配的松散性  \* 二、email和iphone 测试其支持JSR303数据校验  \* 三、abilities 测试其支持复杂的数据结构  \*/  @Component  @ConfigurationProperties(prefix = "itdragon")  @Validated  **public** **class** ConfigurationPropertiesEntity {    **private** String nickName; // 解析成功，支持松散匹配属性  **private** String email;  // @Email // 解析失败，数据校验成功：BindValidationException: Binding validation errors on itdragon  **private** String iphone;  **private** List<String> abilities;  **private** Date createdDate; // 解析成功，支持松散匹配属性    // @ConfigurationProperties("#{(1+2-3)/4\*5}")  **private** String operator; // 语法报错，不支持SpEL表达式：not applicable to field    // 省略getter，setter，toString方法  } |

### @value

例如在xml中

|  |
| --- |
| <**bean id="persion" class="com.sx.springboot.com.sxt.controller.User"**>  <**property name="name" value="字面量/${key}从环境变量或者是配置文件中获取"**></**property**> </**bean**> |

|  |
| --- |
| *//将user对象放到容器中* @Component **public class** User {  *//从配置文件中获取值* @Value(**"${persion.name}"**)  **private** String **name**;   @Value(**"${persion.id}"**)  **private** Integer **id**;   @Value(**"${persion.id}"**)  **private** Boolean **gender**; } |

### @ PropertySource

@PropertySource(value ={"classpath:properties.propertes"})

作用;加载指定文件的配置文件，可以为多个文件。

### 配置文件取值小结

一、ConfigurationProperties注解支持批量注入，而Value注解适合单个注入；

二、ConfigurationProperties注解支持数据校验，而Value注解不支持；

三、ConfigurationProperties注解支持松散匹配属性，而Value注解必须严格匹配属性；

四、ConfigurationProperties不支持强大的SpEL表达式，而Value支持；

总结：如果值是在业务逻辑中获取一下配置文件中的值，用@value。如果专门写了个javabean和配置文件映射，就是用ConfigurationProperties。

## @ImportResource

导入Spring的配置文件，让配置文件里面的内容生效；Spring Boot里面没有Spring的配置文件，我们自己编写的配置文件，也不能自动识别；想让Spring的配置文件生效，加载进来。

**spring.xml：自己定义的spring的配置文件。**

|  |
| --- |
| @SpringBootApplication @ImportResource(locations ={**"classpath:spring.xml"**}) **public class** Application {  **public static void** main(String[] args) {  SpringApplication.*run*(Application.**class**, args);  } } |

## 配置类

@ImportResource方式加载配置文件是不推荐的。

### 以前的方式

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  <**bean id="persion" class="com.sx.springboot.com.sxt.controller.User"**>  <**property name="name" value="字面量/${key}从环境变量或者是配置文件中获取"**></**property**>  </**bean**> </**beans**> |

### 全注解方式

SpringBoot推荐给容器中添加组件的方式；

@Configuration:指明当前的类是一个配置类，就是来替代之前的spring配置文件  
 在配置文件中<bean></bean>标签添加组件

1、配置类======Spring配置文件

|  |
| --- |
| *//@Configuration:指明当前的类是一个配置类，就是来替代之前的spring配置文件 //在配置文件中<bean></bean>标签添加组件* @Configuration **public class** MyAppConfig {  *//将容法的返回只添加到容器中；容器中这个组件的默认的id就是方法名。* @Bean  **public** MessageService getMessageService(){  System.***out***.println(**"创建组件并添加到容器中"**);  **return new** MessageService();  } } |

# springboot的WEB开发

## 简单使用springboot

使用SpringBoot；

1）、创建SpringBoot应用，选中我们需要的模块；

2）、SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来

3）、自己编写业务代码；

# 静态资源的映射规则

webmvc的读取配置类

|  |
| --- |
| @ConfigurationProperties(  prefix = **"spring.resources"**,  ignoreUnknownFields = **false** ) **public class** ResourceProperties { |

可以设置与

1、设置缓存时间

## webjars

### 自动配置类WebMvcAutoConfiguration

WebMvcAutoConfiguration

|  |
| --- |
| **public void** addResourceHandlers(ResourceHandlerRegistry registry) {  **if** (!**this**.resourceProperties.isAddMappings()) {  logger.debug(**"Default resource handling disabled"**);  } **else** {  Duration cachePeriod = **this**.resourceProperties.getCache().getPeriod();  CacheControl cacheControl = **this**.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();  **if** (!registry.hasMappingForPattern(**"/webjars/\*\*"**)) {  **this**.customizeResourceHandlerRegistration(registry.addResourceHandler(**new** String[]{**"/webjars/\*\*"**}).addResourceLocations(**new** String[]{**"classpath:/META-INF/resources/webjars/"**}).setCachePeriod(**this**.getSeconds(cachePeriod)).setCacheControl(cacheControl));  }  String staticPathPattern = **this**.mvcProperties.getStaticPathPattern();  **if** (!registry.hasMappingForPattern(staticPathPattern)) {  **this**.customizeResourceHandlerRegistration(registry.addResourceHandler(**new** String[]{staticPathPattern}).addResourceLocations(getResourceLocations(**this**.resourceProperties.getStaticLocations())).setCachePeriod(**this**.getSeconds(cachePeriod)).setCacheControl(cacheControl));  }  } } |

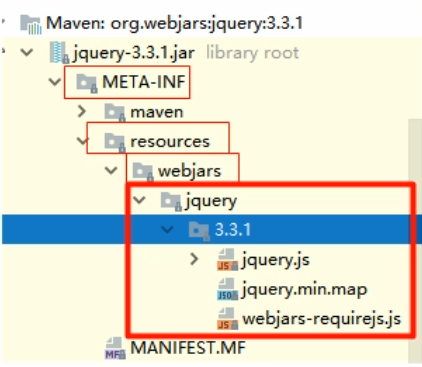
1）、所有/webjars/\*\*，都去classpath:/META-INF/resources/webjars/找资源；

webjars:以jar包的方式引入静态资源；

|  |
| --- |
| <**dependency**>   <**groupId**>org.webjars</**groupId**>  <**artifactId**>jquery</**artifactId**>  <**version**>3.3.0</**version**> </**dependency**> |

访问webjars下的资源只需要写webjars下面的资源就可以了。

例如：http://localhost:8080/webjars/jquery/3.3.1/jquery.js



## /\*\*访问当前项目的任何资源

WebMvcAutoConfiguration

|  |
| --- |
| **public void** addResourceHandlers(ResourceHandlerRegistry registry) {  **if** (!**this**.resourceProperties.isAddMappings()) {  logger.debug(**"Default resource handling disabled"**);  } **else** {  Duration cachePeriod = **this**.resourceProperties.getCache().getPeriod();  CacheControl cacheControl = **this**.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();  **if** (!registry.hasMappingForPattern(**"/webjars/\*\*"**)) {  **this**.customizeResourceHandlerRegistration(registry.addResourceHandler(**new** String[]{**"/webjars/\*\*"**}).addResourceLocations(**new** String[]{**"classpath:/META-INF/resources/webjars/"**}).setCachePeriod(**this**.getSeconds(cachePeriod)).setCacheControl(cacheControl));  }  String staticPathPattern = **this**.mvcProperties.getStaticPathPattern();  **if** (!registry.hasMappingForPattern(staticPathPattern)) {  **this**.customizeResourceHandlerRegistration(registry.addResourceHandler(**new** String[]{staticPathPattern}).addResourceLocations(getResourceLocations(**this**.resourceProperties.getStaticLocations())).setCachePeriod(**this**.getSeconds(cachePeriod)).setCacheControl(cacheControl));   }  } } |

会从ResourceProperties 的getStaticLocations()方法中去获取。

|  |
| --- |
| **public class** ResourceProperties {  **private static final** String[] CLASSPATH\_RESOURCE\_LOCATIONS = **new** String[]{**"classpath:/META-INF/resources/"**, **"classpath:/resources/"**, **"classpath:/static/"**, **"classpath:/public/"**};  **private** String[] staticLocations; //将上边的常量赋给了这个数组  **private boolean** addMappings;  **private final** ResourceProperties.Chain chain;  **private final** ResourceProperties.Cache cache;   **public** ResourceProperties() {  **this**.staticLocations = CLASSPATH\_RESOURCE\_LOCATIONS;  **this**.addMappings = **true**;  **this**.chain = **new** ResourceProperties.Chain();  **this**.cache = **new** ResourceProperties.Cache();  }  //获取属性的方法  **public** String[] getStaticLocations() {  **return this**.staticLocations;  } |

### 访问规则

用户：http://localhost:8080/abc, 如果没有类来处理，将会去下面的文件夹去寻找。

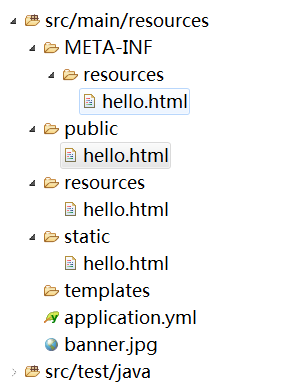
（1）在src/main/resources/目录下创建static文件夹

（2）在src/main/resources/目录下创建resources文件夹

（3）在src/main/resources/目录下创建public文件夹

（4）在src/main/resources/目录下创建META-INF/resources文件夹 “/” 当前项目的根路径

注:访问的优先级为：META-INF>resources>static>public



### 自动以访问路径

自定义的发静态资源文件夹

spring.resources.static-locations=classpath:/hello/,classpath:/haha/

## 欢迎页的配置

欢迎页；静态资源文件夹下的所有index.html页面；被/\*\*“映射；

如果访问localhost:8080/ 会找到资源文文件夹下的indel.html

WebMvcAutoConfiguration

|  |
| --- |
| @Bean **public** WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext) {  **return new** WelcomePageHandlerMapping (**new** TemplateAvailabilityProviders(applicationContext), applicationContext, **this**.getWelcomePage(), **this**.mvcProperties.getStaticPathPattern()); } |

返回的WelcomePageHandlerMapping类

|  |
| --- |
| WelcomePageHandlerMapping(TemplateAvailabilityProviders templateAvailabilityProviders, ApplicationContext applicationContext, Optional<Resource> welcomePage, String staticPathPattern) {  **if** (welcomePage.isPresent() && **"/\*\*"**.equals(staticPathPattern)) {  logger.info(**"Adding welcome page: "** + welcomePage.get());  **this**.setRootViewName(**"forward:index.html"**);  } **else if** (**this**.welcomeTemplateExists(templateAvailabilityProviders, applicationContext)) {  logger.info(**"Adding welcome page template: index"**);  **this**.setRootViewName(**"index"**);  } } |

## 网页图标的设定

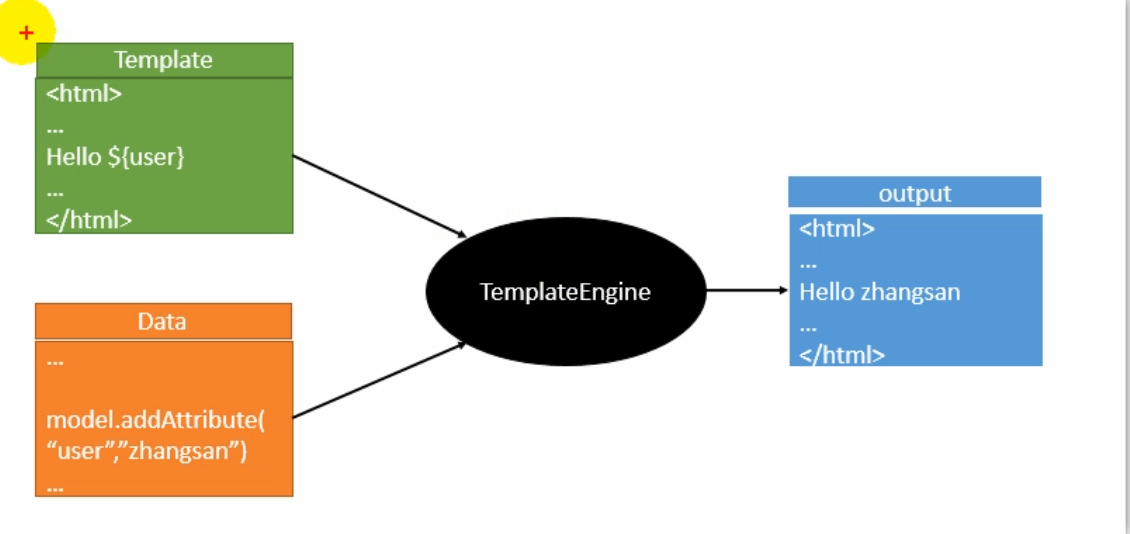
开启图标：**spring.mvc.favicon.enabled：默认开启**

**图标配置：\*\*/favicon.ico 在静态资源文件夹下找。**

|  |
| --- |
| @Configuration  @ConditionalOnProperty(  value = {**"spring.mvc.favicon.enabled"**},  matchIfMissing = **true** )  **public static class** FaviconConfiguration **implements** ResourceLoaderAware {  **private final** ResourceProperties resourceProperties;  **private** ResourceLoader resourceLoader;   **public** FaviconConfiguration(ResourceProperties resourceProperties) {  **this**.resourceProperties = resourceProperties;  }   **public void** setResourceLoader(ResourceLoader resourceLoader) {  **this**.resourceLoader = resourceLoader;  }   @Bean  **public** SimpleUrlHandlerMapping faviconHandlerMapping() {  SimpleUrlHandlerMapping mapping = **new** SimpleUrlHandlerMapping();  mapping.setOrder(-2147483647);  mapping.setUrlMap(Collections.singletonMap(**"\*\*/favicon.ico"**, **this**.faviconRequestHandler()));  **return** mapping;  }   @Bean  **public** ResourceHttpRequestHandler faviconRequestHandler() {  ResourceHttpRequestHandler requestHandler = **new** ResourceHttpRequestHandler();  requestHandler.setLocations(**this**.resolveFaviconLocations());  **return** requestHandler;  }   **private** List<Resource> resolveFaviconLocations() {  String[] staticLocations = WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter.getResourceLocations(**this**.resourceProperties.getStaticLocations());  List<Resource> locations = **new** ArrayList(staticLocations.length + 1);  Stream var10000 = Arrays.stream(staticLocations);  ResourceLoader var10001 = **this**.resourceLoader;  **this**.resourceLoader.getClass();  var10000.map(var10001::getResource).forEach(locations::add);  locations.add(**new** ClassPathResource(**"/"**));  **return** Collections.unmodifiableList(locations);  }  } } |

# thymeleaf

## 模板引擎



## thymeleaf使用

### 导入依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  <version>2.1.0.RELEASE</version>  </dependency>  <**properties**>  <**thymeleaf.version**>3.0.9.RELEASE</**thymeleaf.version**>  *<!--布局功能支持程序，thymeleaf3主程序，layout2以上版本-->* <**thymeleaf-layout-dialect.version**>2.1.1</**thymeleaf-layout-dialect.version**> </**properties**> |

### thymeleaf原理

|  |
| --- |
| @ConfigurationProperties(  prefix = **"spring.thymeleaf"** ) **public class** ThymeleafProperties {  **private static final** Charset DEFAULT\_ENCODING;  **public static final** String DEFAULT\_PREFIX = **"classpath:/templates/"**;  **public static final** String DEFAULT\_SUFFIX = **".html"**; |

1、只要我们把HTML页面放在c1asspath:/templates/，thymeleaf就能自动渲染；

# Springmvc的配置

## springmvc的自动配置

|  |
| --- |
| * 以下是默认的配置      * Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans. * Support for serving static resources, including support for WebJars (see below). * Automatic registration of Converter, GenericConverter, Formatter beans. * Support for HttpMessageConverters (see below). * Automatic registration of MessageCodesResolver (see below). * Static index.html support. * Custom Favicon support (see below). * Automatic use of a ConfigurableWebBindingInitializer bean (see below).   If you want to keep Spring Boot MVC features, and you just want to add additional [MVC configuration](https://docs.spring.io/spring/docs/4.3.22.RELEASE/spring-framework-reference/htmlsingle" \l "mvc" \t "_top) (interceptors, formatters, view controllers etc.) you can add your own @Configuration class of type WebMvcConfigurerAdapter, but **without** @EnableWebMvc. If you wish to provide custom instances of RequestMappingHandlerMapping, RequestMappingHandlerAdapter or ExceptionHandlerExceptionResolver you can declare a WebMvcRegistrationsAdapter instance providing such components.  If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc. |

## 视图解析器

1、自动配置了视图解析器：自动配置了ViewResolvef（视图解析器：根据方法的返回值得到视图对象（View)，视图对象决定如何渲染（转发？重定向？|））

Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans

ContentNegotiatingViewResolver:组合所有的视图解析器的；从容器中获取的视图解析器。

如何定利：我们可以自己给容器中添加一个视图解析器；自动的将其组合进来：、

## Converter, GenericConverter, Formatter

Converter：转换器；public String hello（User user)：类型转换使用Converter

例如：将String转为Integer

Formatter：格式化器：页面带来的数据时2018-1-1，要将该字符串转为Date类型。

## HttpMessageConverters

HttpMessageConverters：Springmvc用来转换http请求和响应的。

例如：想要将User对象变为Json

HttpMessageConverters是从容器中获取的。

自己给容器中添加HttpMessageConverters，只需要将自己的组件注册到容器中。

|  |
| --- |
| **public** WebMvcAutoConfigurationAdapter(ResourceProperties resourceProperties, WebMvcProperties mvcProperties, ListableBeanFactory beanFactory, ObjectProvider<HttpMessageConverters> messageConvertersProvider, ObjectProvider<WebMvcAutoConfiguration.ResourceHandlerRegistrationCustomizer> resourceHandlerRegistrationCustomizerProvider) {  **this**.resourceProperties = resourceProperties;  **this**.mvcProperties = mvcProperties;  **this**.beanFactory = beanFactory;  **this**.messageConvertersProvider = messageConvertersProvider;  **this**.resourceHandlerRegistrationCustomizer = (WebMvcAutoConfiguration.ResourceHandlerRegistrationCustomizer)resourceHandlerRegistrationCustomizerProvider.getIfAvailable(); } |

## 拓展springmvc

编写一个配置类（@Configuration），是WebMvcConfigurerAdapter类型；不能标注@EnableWebMvc

既保留了所有的自动配置，也能用我们扩展的配置；

## 如何修改SpringBoot的默认配置

1）、SpringBoot在自动配置很多组件的时候，先看容器中有没有用户自己配置的（@Bean、

@Component)如果有就用用户配置的，如果没有，才自动配置；如果有些组件可以有多个（ViewResolver)将用户配置的和自己默认的组合起来；

|  |
| --- |
| *//使用WebMvcConfigurerAdapter可以拓展springmvc的功能* @Configuration **public class** MyMvcConfig **extends** WebMvcConfigurerAdapter {   @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  *//1、浏览器发送/hello，直接映射到index页面* registry.addViewController(**"/hehe"**).setViewName(**"hello.html"**);  } } |

原理：

1）、WebMvcAutoConfiguration是SpringMVC的自动配置类

2）、在做其他自动配置时会导入；@lmport（EnableWebMvcConfiguration.class)



3）、容器中所有的WebMvcConfigurer都会一起起作用；

4）、我们的配置类也会被调用；

效果：SpringMVC的自动配置和我们的扩展配置都会起作用；

## @EnableWebmvc

SpringBoot对SpringMVC的自动配置不需要了，所有都是我们自己配置；所有的自动配置都实现了。我们需要在配置类中添加@EnableWebMvc即可；

|  |
| --- |
| *//使用WebMvcConfigurerAdapter可以拓展springmvc的功* @EnableWebMvc @Configuration **public class** MyMvcConfig **extends** WebMvcConfigurerAdapter {   @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  *//1、浏览器发送/hello，直接映射到index页面* registry.addViewController(**"/hehe"**).setViewName(**"hello.html"**);  } } |

原理：

@EnableWebMvc

|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***) @Target(ElementType.***TYPE***) @Documented @Import(DelegatingWebMvcConfiguration.**class**) **public** @**interface** EnableWebMvc { } |

DelegatingWebMvcConfiguration

|  |
| --- |
| @Configuration **public class** DelegatingWebMvcConfiguration **extends** WebMvcConfigurationSupport { |

WebMvcAutoConfiguration

|  |
| --- |
| @Configuration @ConditionalOnWebApplication @ConditionalOnClass({Servlet.**class**, DispatcherServlet.**class**, WebMvcConfigurerAdapter.**class**})  //当webmvc中没有这个组件时，自动配置才生效 @ConditionalOnMissingBean({WebMvcConfigurationSupport.**class**}) @AutoConfigureOrder(-2147483638) @AutoConfigureAfter({DispatcherServletAutoConfiguration.**class**, ValidationAutoConfiguration.**class**}) **public class** WebMvcAutoConfiguration { |

总结：

1、EnableWebMvc将WebMvcConfigurationSupport导入进来了，只是springmvc导入的基本功能。

2、在SpringBoot中会有非常多的xxxConfigurer帮助我们进行扩展配置

# Restful crud

## 配置

### 依赖

|  |
| --- |
| <**properties**>  <**thymeleaf.version**>3.0.9.RELEASE</**thymeleaf.version**>  *<!--布局功能支持程序，thymeleaf3主程序，layout2以上版本-->* <**thymeleaf-layout-dialect.version**>2.1.1</**thymeleaf-layout-dialect.version**> </**properties**> |
| *<!--引入thymeleaf模板引擎-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> </**dependency**> *<!--webjar-->* <**dependency**>  <**groupId**>org.webjars</**groupId**>  <**artifactId**>jquery</**artifactId**>  <**version**>3.3.1-2</**version**> </**dependency**> <**dependency**>  <**groupId**>org.webjars</**groupId**>  <**artifactId**>bootstrap</**artifactId**>  <**version**>4.3.1</**version**> </**dependency**> |

### 修改页面

导入webjar资源

<**link th:href="@{/webjars/bootstrap/4.3.1/css/bootstrap.css}" rel="stylesheet"**>

导入css资源

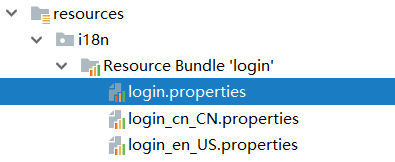
<**link th:href="@{asserts/css/signin.css}" rel="stylesheet"**>

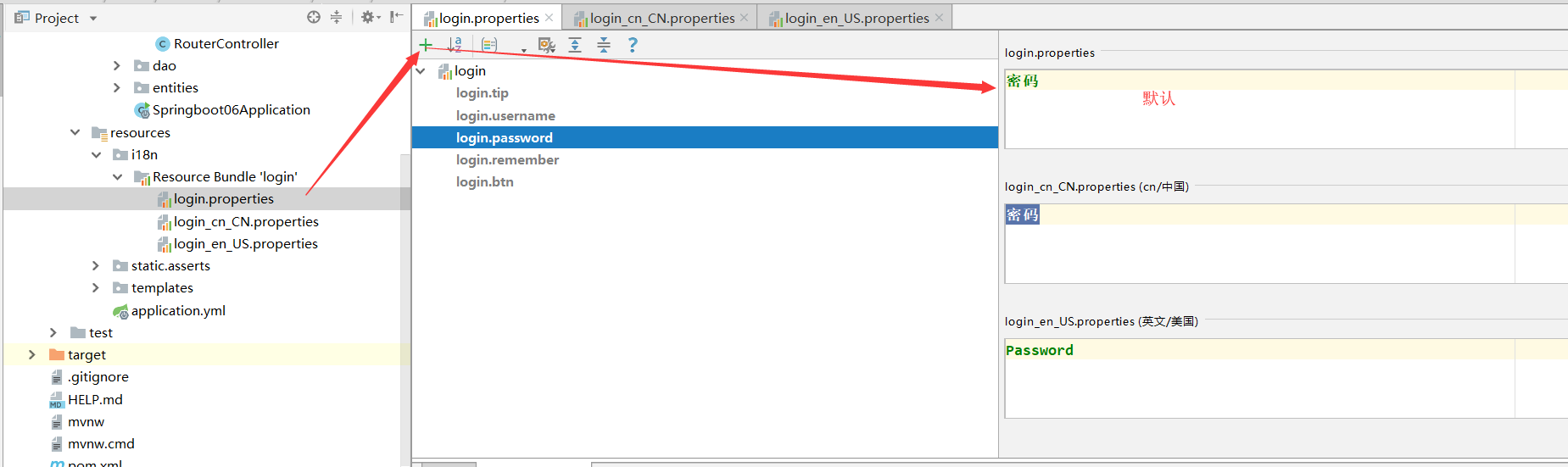
图片资源：

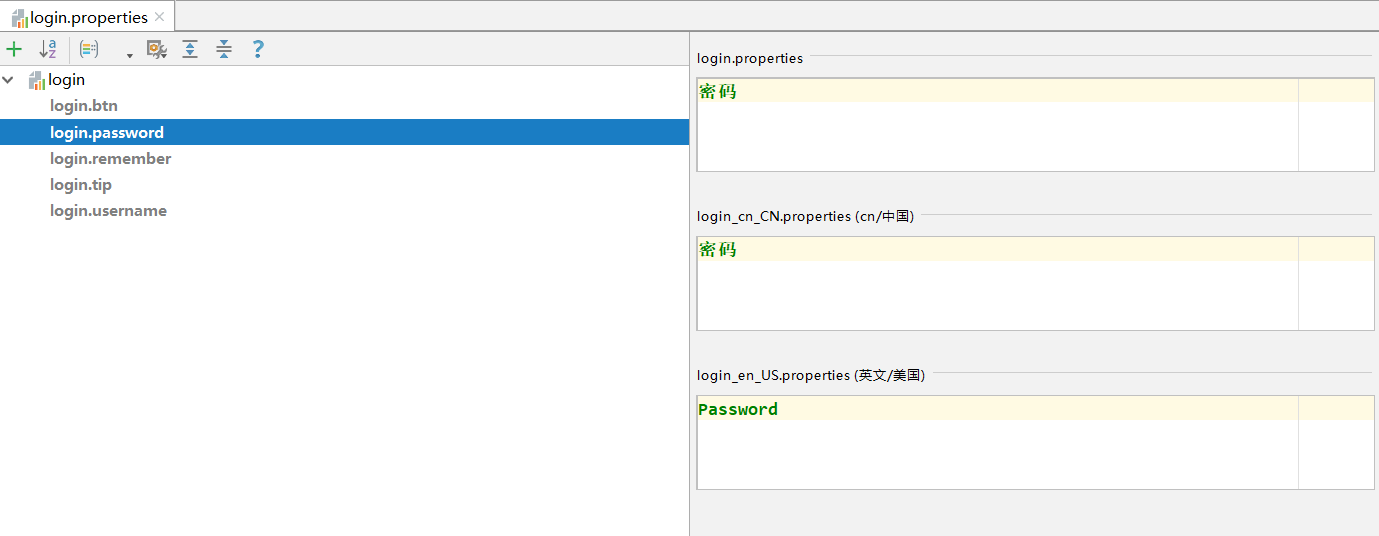
<**img class="mb-4" th:src="@{asserts/img/bootstrap-solid.svg}" alt="" width="72" height="72"**>

## 国际化

### properties配置文件

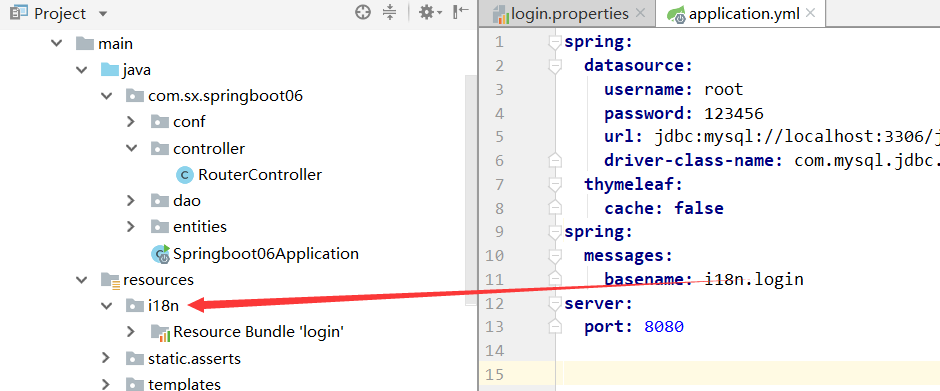






### yml配置文件

|  |
| --- |
| **spring:  messages:  basename:** i18n.login |



### 国际化原理

MessageSourceAutoConfiguration

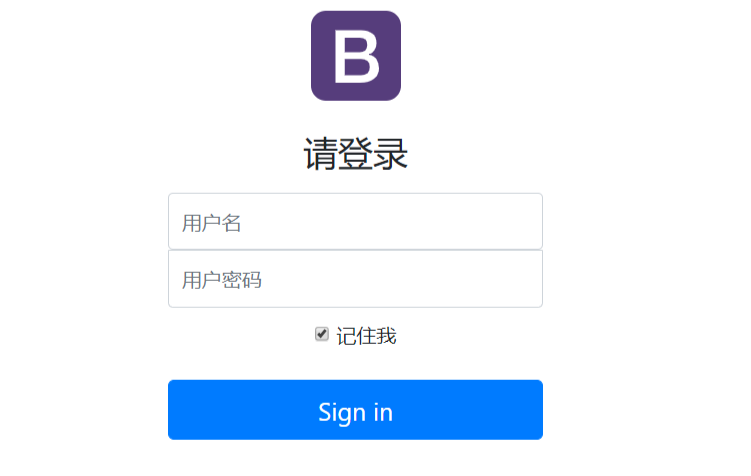
|  |
| --- |
| @Configuration @ConditionalOnMissingBean(value = MessageSource.**class**, search = SearchStrategy.***CURRENT***) @AutoConfigureOrder(Ordered.***HIGHEST\_PRECEDENCE***) @Conditional(ResourceBundleCondition.**class**) @EnableConfigurationProperties @ConfigurationProperties(prefix = **"spring.messages"**) **public class** MessageSourceAutoConfiguration {   **private static final** Resource[] ***NO\_RESOURCES*** = {};  **private** String **basename** = **"messages"**;  //我们的配置文件的名字可以叫做**messages.properties，可以不用做任何配置。**  @Bean **public** MessageSource messageSource() {  ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();  **if** (StringUtils.*hasText*(**this**.**basename**)) {  //设置国际化资源的基础名（去掉语言国家代码的。  messageSource.setBasenames(StringUtils.*commaDelimitedListToStringArray*(  StringUtils.*trimAllWhitespace*(**this**.**basename**)));  }  **if** (**this**.**encoding** != **null**) {  messageSource.setDefaultEncoding(**this**.**encoding**.name());  }  messageSource.setFallbackToSystemLocale(**this**.**fallbackToSystemLocale**);  messageSource.setCacheSeconds(**this**.**cacheSeconds**);  messageSource.setAlwaysUseMessageFormat(**this**.**alwaysUseMessageFormat**);  **return** messageSource; } |

### 去页面获取国际化的值

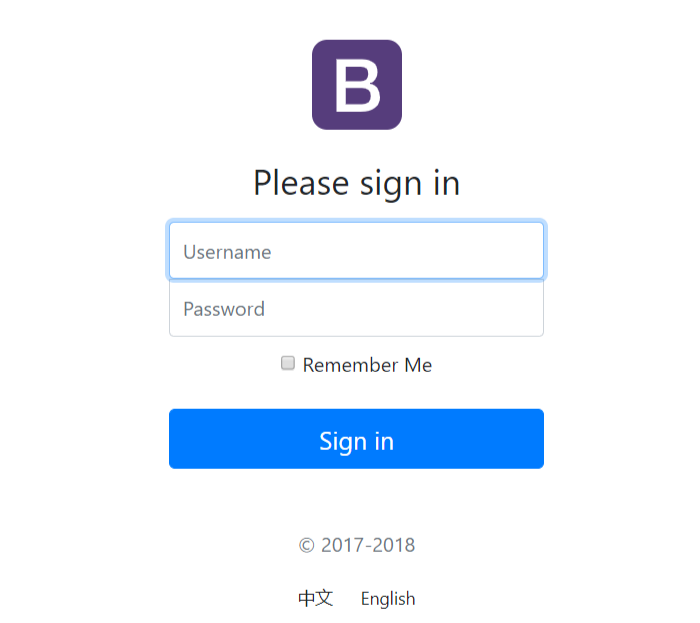
注：使用#{获取国际化的值}

|  |
| --- |
| <**form class="form-signin" action="dashboard.html"**>  <**img class="mb-4" th:src="@{asserts/img/bootstrap-solid.svg}" src="asserts/img/bootstrap-solid.svg" alt="" width="72" height="72"**>  <**h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}"**>Please sign in</**h1**>  <**label class="sr-only" th:text="#{login.username}"**>Username</**label**>  <**input type="text" class="form-control" placeholder="Username" th:placeholder="#{login.username}" required="" autofocus=""**>  <**label class="sr-only" th:text="#{login.password}"**>Password</**label**>  <**input type="password" class="form-control" placeholder="Password"th:placeholder="#{login.password}" required=""**>  <**div class="checkbox mb-3"**>  <**label**>  <**input type="checkbox" value="remember-me"**> [[#{login.remember}]]  </**label**> |

### 效果



切换浏览器语言



### 点击按钮实现切换国际化

原理：

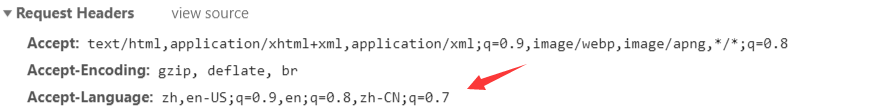
际化Locale（区域信息对象）；LocaleResolver（获取区域信息对象）；

WebMvcAutoConfiguration中配置了

|  |
| --- |
| @Bean @ConditionalOnMissingBean @ConditionalOnProperty(prefix = **"spring.mvc"**, name = **"locale"**) **public** LocaleResolver localeResolver() {  **if** (**this**.**mvcProperties** .getLocaleResolver() == WebMvcProperties.LocaleResolver.***FIXED***) {  **return new** FixedLocaleResolver(**this**.**mvcProperties**.getLocale());  }  AcceptHeaderLocaleResolver localeResolver = **new** AcceptHeaderLocaleResolver();  localeResolver.setDefaultLocale(**this**.**mvcProperties**.getLocale());  **return** localeResolver; } |

|  |
| --- |
| @Override **public** Locale resolveLocale(HttpServletRequest request) {  Locale defaultLocale = getDefaultLocale();  //从请求头中获取信息  **if** (defaultLocale != **null** && request.getHeader(**"Accept-Language"**) == **null**) {  **return** defaultLocale;  }  Locale requestLocale = request.getLocale();  List<Locale> supportedLocales = getSupportedLocales();  **if** (supportedLocales.isEmpty() || supportedLocales.contains(requestLocale)) {  **return** requestLocale;  }  Locale supportedLocale = findSupportedLocale(request, supportedLocales);  **if** (supportedLocale != **null**) {  **return** supportedLocale;  }  **return** (defaultLocale != **null** ? defaultLocale : requestLocale); } |

从request中取出对应的local信息



解决方法：

点击中文/英文切换

|  |
| --- |
| <**a class="btn btn-sm" th:href="@{/index.html('zh\_CN')}"**>中文</**a**> <**a class="btn btn-sm" th:href="@{/index.html('en\_US')}"**>English</**a**> |

配置

|  |
| --- |
| */\*\*  \** ***@author*** *pad  \** ***@description*** *${description}  \** ***@date*** *2019/3/4  \*/* **public class** MyLocalResolver **implements** LocaleResolver {  @Override  **public** Locale resolveLocale(HttpServletRequest request) {  String acceptLanguage = request.getParameter(**"acceptLanguage"**);  *//如果不存在，则使用默认的语言* Locale locale = Locale.*getDefault*();  **if**(!StringUtils.*isEmpty*(acceptLanguage)){  *//分成语言代码和国家代码* String[] split = acceptLanguage.split(**"\_"**);  locale = **new** Locale(split[0], split[1]);  }  **return** locale;  }  @Override  **public void** setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale) {   } } |

# aop开发

# 配置Web

## 拦截器和跨域请求

|  |
| --- |
| **package** com.sxt.cloud.conf;  **import** java.io.PrintWriter;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.web.servlet.HandlerInterceptor;  **import** org.springframework.web.servlet.ModelAndView;  **import** org.springframework.web.servlet.config.annotation.CorsRegistry;  **import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;  @Configuration  **public** **class** WebConfig **extends** WebMvcConfigurerAdapter{      //注册一个拦截器  **public** **void** addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(**new** HandlerInterceptor() {  @Override  **public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  String token = request.getHeader("token");  **if**(token==**null**||!token.equals("123456")) {  PrintWriter writer = response.getWriter();  writer.println("{'code':'403','msg':'token错误'}");  writer.close();  **return** **false**;  }  **return** **true**;  }  @Override  **public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  ModelAndView modelAndView) **throws** Exception {    }  @Override  **public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)  **throws** Exception {    }//拦截所有请求  }).addPathPatterns("/\*\*").excludePathPatterns(//排除swagger的依赖  "/\*.html",  "/\*\*/\*\*.css",  "/\*\*/\*\*.js",  "/favicon.ico",  "/swagger-resources/\*\*",  "/v2/api-docs/\*\*",  "/user/code",  "/user/add"  );  }    //Cors 添加跨域请求  @Override  **public** **void** addCorsMappings(CorsRegistry registry) {  registry.addMapping("/\*\*").  //跨域请求方法  allowedMethods("\*").  //  allowedOrigins("\*").  //跨域是否允许cookie  allowCredentials(**true**).  //跨域是否允许那些头  allowedHeaders("\*").  //请求成功后，距离下次验证的时间，一般两个小时  maxAge(7200);  **super**.addCorsMappings(registry);    }  } |

## 异常配置类

目的：将抛出的异常拦截下来。

|  |
| --- |
| /\*\*以后所有的异常都会进该类  \* **@author** Padingpading  \*  \*/  //web层的切面  @RestControllerAdvice  **public** **class** WebErrorLog {    //记录日志  **private** **static** Logger *logger* = LoggerFactory.*getLogger*(WebErrorLog.**class**);    //拦截所有的异常，异常后会进入该类  //括号中为要处理的异常类型  @ExceptionHandler(RuntimeException.**class**)  **public** Result handlerRuntime(Exception exception) {  //1、程序员  *logger*.error(exception.getMessage());  //2、用户  **return** Result.*error*(500, "服务器错误");    }  } |

## 日志的配置

### yml文件

|  |
| --- |
| #log日志的配置  logging:  file: log/log.log #日志文件放到哪里,相对路径  path: #日志文件放到哪里 绝对路径 /etc/user/service/log.log linux  level: #日志的级别，通过报名配置  root: error #在没有配置包的级别，默认是这个级别,默认对error都不打印  com.sxt.cloud.service.impl: info #级别很低，信息也多 |

### 日志打印

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

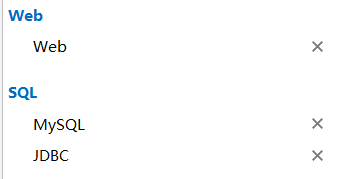
|  |
| --- |
| **private** **static** Logger *logger* = LoggerFactory.*getLogger*(MessageServiceImpl.**class**);  *logger*.info("记录日志"); |

### 日志注解

@log

# springBoot整合jdbc

## 创建项目



## 配置

# springBoot与数据访问

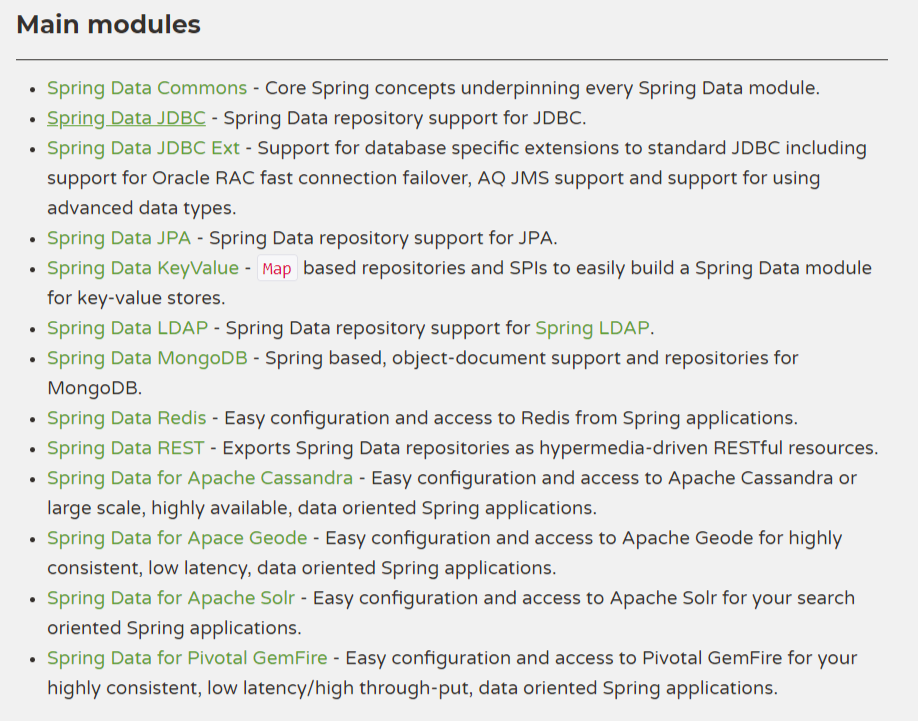
## 简介

对于数据访问层，无论是SQL还是NOSQL，Spring Boot默认采用整合Spring Data的方式进行统一处理，添加大量自动配置，屏蔽了很多设置。引入各种xxxTemplate，xxxRepository来简化我们对数据访问层的操作。对我们来说只需要进行简单的设置即可。我们将在数据访问章节测试使用SQL相关、NOSQL在缓存、消息、检索等章节测试。

-JDBC

-MyBatis

-JPA



spring data 与数据库相关操作的。

## 整合基本的JDBC与数据源

### 步骤

1、引入starter

-spring-boot-starter-jdbc

2、配置application.yml

3、测试

4、高级配置：使用druid数据源

-引入druid

-配置属性

5、配置druid数据源监控

### 导入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> |

### yml配置

|  |
| --- |
| **spring:  datasource:  username:** root  **password:** 123456  **url:** jdbc:mysql://localhost:3306/jdbc  **driver-class-name:** com.mysql.jdbc.Driver |

默认使用的数据源：class org.apache.tomcat.jdbc.pool.DataSource

数据源的相关配置都在DataSourceProperties里面

### 自动配置原理

org.springframework.boot.autoconfigure.jdbc包下

1、参考DataSourceConfiguration，根据配置创建数据源，默认使用Tomcat连接池；

可以使用spring.datasource.type指定自定义的数据源类型;

2、默认支持的数据源：

org.apache.tomcat.jdbc.pool.DataSource

HikariDataSource

org.apache.commons.dbcp.BasicDataSource

org.apache.commons.dbcp2.BasicDataSource

3、自定义数据源

如果不是上面的数据源

|  |
| --- |
| */\*\*  \* Generic DataSource configuration.  \*/* @ConditionalOnMissingBean(DataSource.**class**) @ConditionalOnProperty(name = **"spring.datasource.type"**) **static class** Generic {  @Bean////使用DataSourceBuilder创建数据源，利用反射创建响应type的数据源，并且绑定相关属性  **public** DataSource dataSource(DataSourceProperties properties) {  **return** properties.initializeDataSourceBuilder().build();  } } |

4、DataSourcelnitializer:ApplicationListener;

作用

1）、runschemaScripts()；运行建表语句；

2）、runDataScripts()；运行插入数据的sql语句；

默认只需要将文件命名为：schema.sq、data-.sql

## 整合Druid

### 引入jar包

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.10</**version**> </**dependency**> |

### yml

|  |
| --- |
| **spring:  datasource:  username:** root  **password:** 123456  **url:** jdbc:mysql://127.0.0.1:3306/car?serverTimezone=GMT%2B8  **driver-class-name:** com.mysql.jdbc.Driver  **type:** com.alibaba.druid.pool.DruidDataSource |

### config

|  |
| --- |
| @Configuration **public class** DuridDatasource {  @ConfigurationProperties(prefix =**"spring.dataSource"**)  @Bean  **public** DataSource druid(){  **return new** DruidDataSource();  }   *//配置Druid的监控台  //1、配置一个管理后台的servlet* @Bean  **public** ServletRegistrationBean statViewServlet(){  *//处理druid的所有请求* ServletRegistrationBean bean = **new** ServletRegistrationBean(**new** StatViewServlet(), **"/druid/\*"**);  *//配置参数* Map<String,String> initParams = **new** HashMap<>();  *//登录后台时的账号和密码* initParams.put(**"loginUsername"**,**"admin"**);  initParams.put(**"loginPassword"**,**"123456"**);  *//默认允许访问(白名单)* initParams.put(**"allow"**,**""**);  *//禁止访问的ip(黑名单)* initParams.put(**"deny"**,**"192.168.10.136"**);  bean.setInitParameters(initParams);  **return** bean;  }  *//配置一个web监控的的filter* **public** FilterRegistrationBean webStatFilter(){  FilterRegistrationBean bean = **new** FilterRegistrationBean();  bean.setFilter(**new** WebStatFilter());  Map<String,String> initParams = **new** HashMap<>();  initParams.put(**"exclusions"**,**"\*.js,\*.css,/druid/\*"**);  bean.setInitParameters(initParams);  bean.setUrlPatterns(Arrays.*asList*(**"/\*"**));  **return** bean;  } } |

# 集成数据库

## 添加依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**scope**>runtime</**scope**>  <**optional**>true</**optional**> </**dependency**> <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**> </**dependency**> <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**optional**>true</**optional**> </**dependency**> |

### yml文件

|  |
| --- |
|  |

## 数据源的配置

### DriverManagerDataSource

yml文件

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **url**: jdbc:mysql://localhost:3306/car  **driver-class-name**: com.mysql.cj.jdbc.Driver  **password**: 123456  *#如果不指定数据源的类型，默认使用HikariDataSource数据源* **type**: org.springframework.jdbc.datasource.DriverManagerDataSource |

### dbcp

添加依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-dbcp2</**artifactId**> </**dependency**> |

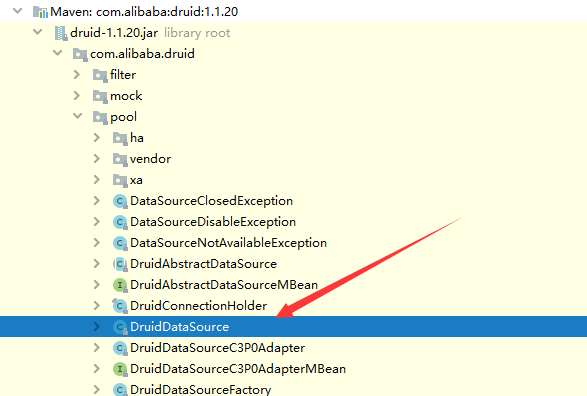
修改yml

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **url**: jdbc:mysql://localhost:3306/car  **driver-class-name**: com.mysql.cj.jdbc.Driver  **password**: 123456  *#如果不指定数据源的类型，默认使用HikariDataSource数据源* **type**: org.apache.commons.dbcp2.BasicDataSource |

## Druid

### druid配置

复制类名字



### 配置yml

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **url**: jdbc:mysql://localhost:3306/car  **driver-class-name**: com.mysql.cj.jdbc.Driver  **password**: 123456  *#如果不指定数据源的类型，默认使用HikariDataSource数据源* **type**: com.alibaba.druid.pool.DruidDataSource  **max-wait**: 5000  **max-active**: 20  **initila-size**: 1 |

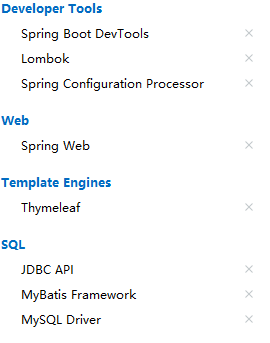
### java类配置

|  |
| --- |
| @Configuration @ConditionalOnClass(value = {DataSource.**class**, DruidDataSource.**class**}) **public class** MyDuridAutoConfiguration {  @Bean  *//该注解注入将配置文件注入到当前对象中* @ConfigurationProperties(prefix =**"spring.datasource.druid"**)  **public** DruidDataSource getDataSource(){  DruidDataSource druidDataSource = **new** DruidDataSource();  druidDataSource.setMaxActive(20);  druidDataSource.setInitialSize(10);  druidDataSource.setMaxWait(10);  **return** druidDataSource;  } } |

### 监控页面的配置

# 集成mybatis

## 导入依赖



## 使用注解方式完成

### yml配置

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **url**: jdbc:mysql://localhost:3306/car?characterEncoding=utf8&useSSL=false&serverTimezone=GMT  **driver-class-name**: com.mysql.cj.jdbc.Driver  **password**: 123456  *#如果不指定数据源的类型，默认使用HikariDataSource数据源* **type**: com.alibaba.druid.pool.DruidDataSource  **max-wait**: 5000  **max-active**: 20  **initila-size**: 1 *##mybatis日志的配置* **mybatis**:  **configuration**:  **log-impl**: org.apache.ibatis.logging.stdout.StdOutImpl |

### domain

|  |
| --- |
| @Data @AllArgsConstructor @NoArgsConstructor @ToString **public class** User {  **private** Integer **id**;   **private** String **name**;   **private** String **address**;   **private** Date **birth**; } |

### mapper/ UserMapper

@Mapper创建代理类

|  |
| --- |
| @Mapper **public interface** UserMapper {  @Insert(value = **"insert into user(id,name,address,birth) values(#{id},#{name},#{address},#{birth})"**)  **public void** addUser(User user);   @Update(value = **"update user set name=#{name} where id=#{id}"**)  **public void** updateUser(User user);   @Delete(value = **"delete from user where id=#{id}"**)  **public void** deleteUser(@Param(**"id"**) Integer id);   @Select(value = **"select \* from user where id=#{id}"**)  **public** User queryUserById(@Param(**"id"**) Integer id);   @Select(value = **"select \* from user"**)  **public** List<User> queryUsers(); } |

### 测试

|  |
| --- |
| @SpringBootTest **class** MybatisApplicationTests {  @Autowired  **private** UserMapper **userMapper**;   @Test  **void** contextLoads() {  **userMapper**.addUser(**new** User(1,**"dfsd"**,**"df"**,**new** Date()));  } } |

### 扫描全部的mapper

|  |
| --- |
| @SpringBootApplication @MapperScan(basePackages = {**"com.sxt.mapper"**}) //添加该注解 **public class** MybatisApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(MybatisApplication.**class**, args);  } } |

## 使用xml配置

### mapper

|  |
| --- |
| **public interface** UserMapper {   **public void** addUser(User user);   **public void** updateUser(User user);   **public void** deleteUser(@Param(**"id"**) Integer id);   **public** User queryUserById(@Param(**"id"**) Integer id);   **public** List<User> queryUsers(); } |

### mapper.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" *>*** <**mapper namespace="com.example.mybatis.mapper.UserMapper"**>   <**resultMap id="BaseResMap" type="com.example.mybatis.domaindo.User"**>  <**id property="id" column="id"**></**id**>  <**result property="name" column="name"**></**result**>  <**result property="address" column="address"**></**result**>  <**result property="birth" column="birth"**></**result**>  </**resultMap**>   <**insert id="addUser"** >  insert into user(id,name,address,birth) values(#{id},#{name},#{address},#{birth})  </**insert**>   <**update id="updateUser"**>  update user set name=#{name},address=#{address} where id=#{id}  </**update**>   <**delete id="deleteUser"**>  delete from user where id=#{id}  </**delete**>   <**select id="queryUserById" resultMap="BaseResMap"**>  select *\** from user where id=#{id}  </**select**>   <**select id="queryUsers" resultMap="BaseResMap"**>  select *\** from user  </**select**>  </**mapper**> |

### 配置yml

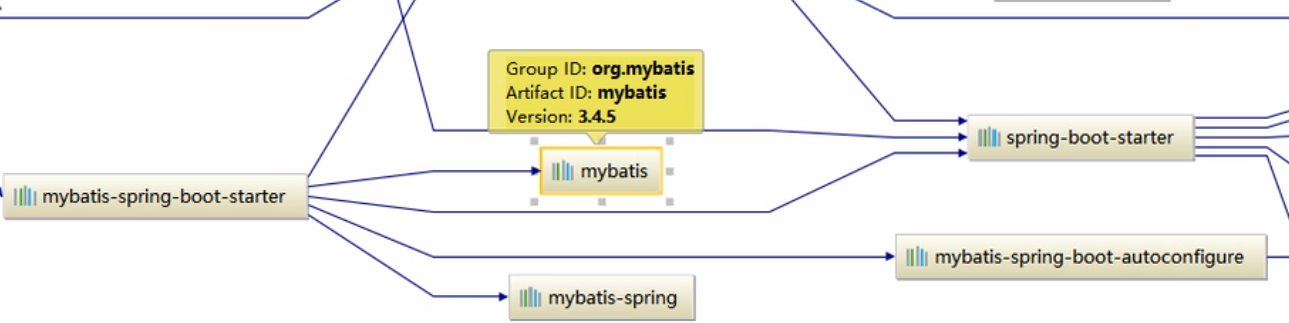
|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **url**: jdbc:mysql://localhost:3306/car?characterEncoding=utf8&useSSL=false&serverTimezone=GMT  **driver-class-name**: com.mysql.cj.jdbc.Driver  **password**: 123456  **type**: com.alibaba.druid.pool.DruidDataSource  **max-wait**: 5000  **max-active**: 20  **initila-size**: 1 *##mybatis配置* **mybatis**:  **configuration**:##日志  **log-impl**: org.apache.ibatis.logging.stdout.StdOutImpl  **mapper-locations**: ##mapper位置  - classpath:mapper/\*.xml *#mapper的xml的路径* |

### 启动类

MapperScan:扫描mapper位置

|  |
| --- |
| @SpringBootApplication  @MapperScan(basePackages = {**"com.example.mybatis.mapper"**}) **public class** MybatisApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(MybatisApplication.**class**, args);  } } |

mybatis stater的内容



## mybatis外部config文件加载方式

住：实际开发中会将该文件省掉

### mybatis.cfg.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd" *>*** <**configuration**>  <**mappers**>  <**mapper resource="classpath:mapper/UserMapper.xml"**></**mapper**>  </**mappers**> </**configuration**> |

### yml文件

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **url**: jdbc:mysql://localhost:3306/car?characterEncoding=utf8&useSSL=false&serverTimezone=GMT  **driver-class-name**: com.mysql.cj.jdbc.Driver  **password**: 123456  *#\u5982\u679C\u4E0D\u6307\u5B9A\u6570\u636E\u6E90\u7684\u7C7B\u578B\uFF0C\u9ED8\u8BA4\u4F7F\u7528HikariDataSource\u6570\u636E\u6E90* **type**: com.alibaba.druid.pool.DruidDataSource  **max-wait**: 5000  **max-active**: 20  **initila-size**: 1 *##mybatis\u65E5\u5FD7\u7684\u914D\u7F6E* **mybatis**: *# configuration: ##必须注掉，会和 config-location冲突 # log-impl: org.apache.ibatis.logging.stdout.StdOutImpl* **mapper-locations**:  **config-location**: classpath:mybatis.cfg.xml |

## 第一种集成pageHelper

### 配置

|  |
| --- |
| @Configuration **public class** PageHelperAutoConfigrution {  @Bean  **public** PageInterceptor pageInterceptor(){  **return new** PageInterceptor();  } } |

## 第二种集成pageHelper

### 引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper</**artifactId**>  <**version**>5.1.10</**version**> </**dependency**> |

### 配置

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd" *>*** <**configuration**>  <**plugins**>  <**plugin interceptor="com.github.pagehelper.PageInterceptor"**></**plugin**>  </**plugins**>  <**mappers**>  <**mapper resource="mapper/UserMapper.xml"**></**mapper**>  </**mappers**> </**configuration**> |

## 使用pageHelperStarter

### 直接添加依赖即可

|  |
| --- |
| <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>2.1.1</**version**> </**dependency**> |

# 集成lombok

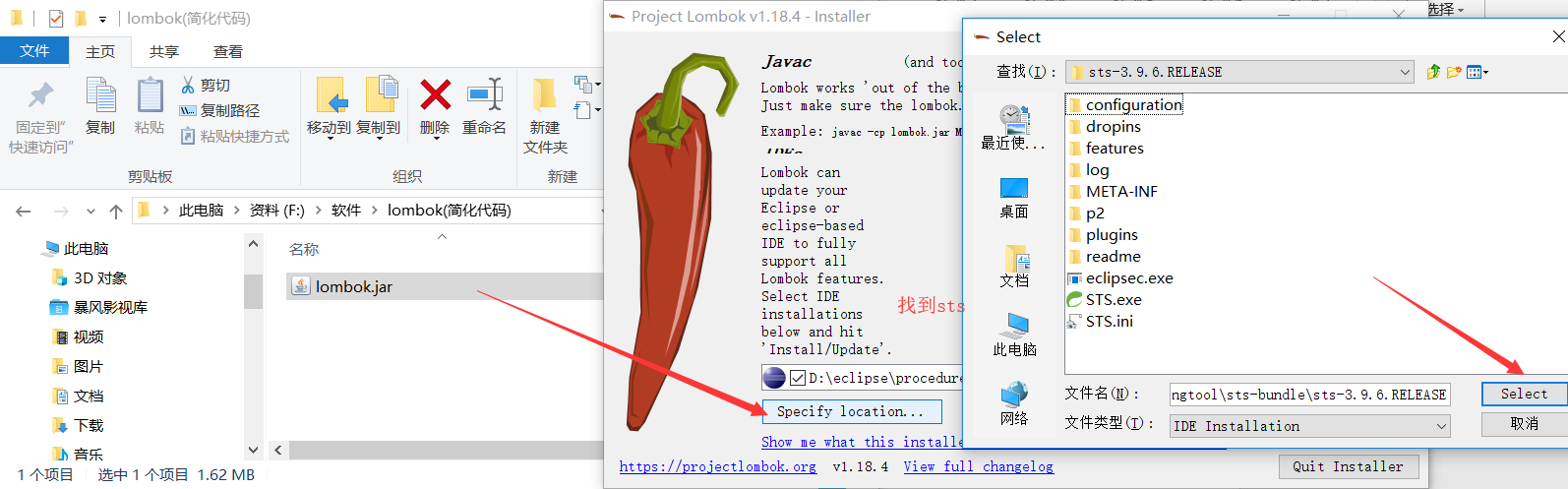
## lombok

目的，简化代码

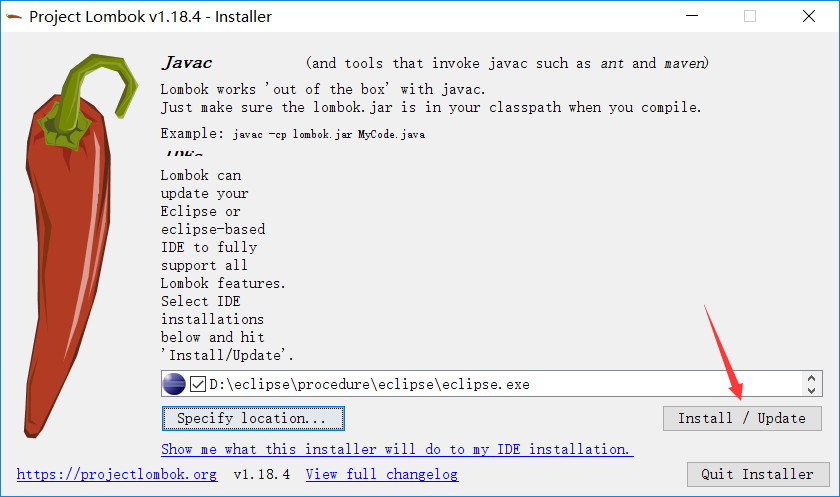
功能：自动生成，getter和setter代码

## lombok安装

### 选择sts的目录



### 确定



### 安装成功标志



### 引入依赖pom

springboot中已经集成，所以不需要版本

|  |
| --- |
| <!--lombok -->  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <scope>provided</scope>  </dependency> |

## 功能

@Data ：自动生成getter和setter方法，和无参构造器

@AllArgsConstructor:有参构造器

@EqualsAndHashCode equals和hashcode方法

@

# 集成Swagger

## 加入swagger2的依赖

|  |
| --- |
| <!-- swagger -->  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.5.0</version>  </dependency>  <!-- swagger-ui -->  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.5.0</version>  </dependency> |

SwaggerConfig

## 配置类

## SwaggerConfig配置类

|  |
| --- |
| @Configuration  @EnableSwagger2  **public** **class** SwaggerConfig {  @Bean  **public** Docket swaggerSpringMvcPlugin() {  **return** **new** Docket(DocumentationType.***SWAGGER\_2***).select()  .apis(RequestHandlerSelectors.*withMethodAnnotation*(ApiOperation.**class**)).build();  }  } |

## controller

|  |
| --- |
| @Api(~~description~~="用户接口")  @RestController  @RequestMapping("user")  **public** **class** UserController {    **public** **static** List<User> *userList* = **new** ArrayList<>();  **static** {  **for**(**int** i = 0;i<10;i++) {  *userList*.add(**new** User(i, "sx"+i, "北京"+i));  }  }    //用户添加  @ApiOperation(value="新增用户",notes="添加一名用户")  @RequestMapping(value="addUser",method= {RequestMethod.***POST***})  **public** ResultObj addUser(@RequestBody User user) {  *userList*.add(user);  pring();  **return** **new** ResultObj("添加成功", HttpStatus.***OK***.value());  }    //用户修改  @ApiOperation(value="修改用户",notes="修改一名用户")  @RequestMapping(value="updateUser",method= {RequestMethod.***POST***})  **public** ResultObj updateUser(@RequestBody User user) {  **for**(User user1 :*userList*) {  **if**(user1.getId()==user.getId()) {  BeanUtils.*copyProperties*(user, user1);  **break**;  }  }  pring();  **return** **new** ResultObj("修改成功", HttpStatus.***OK***.value());  }    //用户删除  @ApiOperation(value="删除用户",notes="删除一名用户")  @RequestMapping(value="deleteUser",method= {RequestMethod.***POST***})  **public** ResultObj deleteUser(@RequestParam("id") Integer id) {  **for**(User user1 :*userList*) {  **if**(user1.getId()==id) {  *userList*.remove(user1);  **break**;  }  }  pring();  **return** **new** ResultObj("删除成功", HttpStatus.***OK***.value());  }    //查询用户  @ApiOperation(value="查询用户",notes="查询一名用户")  @RequestMapping(value="queryUser",method= {RequestMethod.***POST***})  **public** User queryUser(@RequestParam("id") Integer id) {  User u = **new** User();  **for**(User user1 :*userList*) {  **if**(user1.getId()==id) {  u = user1;  **break**;  }  }  **return** u;  }    //查询用户  @ApiOperation(value="查询全部用户",notes="查询全部用户")  @RequestMapping(value="queryAllUser",method= {RequestMethod.***POST***})  **public** List<User> queryAllUser() {  **return** *userList*;  }  **public** **void** pring() {  **for** (User user : *userList*) {  System.***out***.println(user);  }  }  } |

## 访问地址

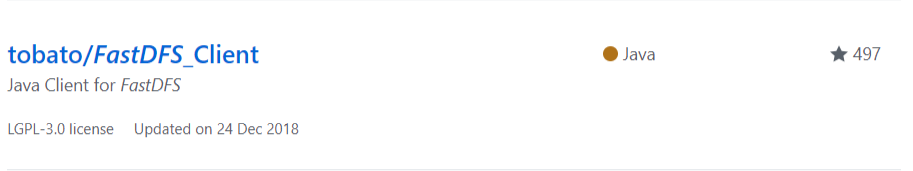
http://127.0.0.1:8080/swagger-ui.html



# 整合shiro

# 整合FastDfs

## 依赖



|  |
| --- |
| <!--fastdfs -->  <dependency>  <groupId>com.github.tobato</groupId>  <artifactId>fastdfs-client</artifactId>  <version>1.26.5</version>  </dependency> |

## yml配置

|  |
| --- |
| fdfs:  tracker-list: #TrackerList参数,支持多个  - 192.168.10.136:22122  pool:  max-total:  max-idle-per-key:  min-idle-per-key: |

## 测试

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** UserServiceApplicationTests {  @Autowired  **private** FastFileStorageClient fastClient;    @Test  **public** **void** contextLoads() {  File file = **new** File("D://1.jpg");  **try** {  StorePath uploadFile = fastClient.uploadFile(**new** FileInputStream(file), file.length(), "jpg", **null**);  System.***out***.println(uploadFile.getFullPath());  } **catch** (FileNotFoundException e) {  e.printStackTrace();  }  }  } |

## 访问文件

从nginx里面访问

<http://192.168.10.136:8080/g1/M00/00/00/wKgKiFxyhs2AXm4SAABrcJlW7YU676.jpg>

# 附录

## springboot转为springcloud

### 在springboot中加入依赖

client

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  </dependency> |

### 依赖管理

|  |
| --- |
| <dependencyManagement>  <!--管理spring-cloud -->  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

### 并定义依赖管理的版本

|  |
| --- |
| <properties>  <spring-cloud.version>Edgware.SR5</spring-cloud.version>  </properties> |

### 配置yml

|  |
| --- |
| #eureka注册中心地址  eureka:  client:  service-url:  defaultZone: ${EUREKA.SERVER\_URL:http://localhost:8761/eureka}  spring  application:  name: user-service |