# SpringBoot

# 1SpringBoot简介

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式进行配置，从而使开发人员不再需要定义样板化的配置。用我的话来理解，就是 Spring Boot 其实不是什么新的框架，它默认配置了很多框架的使用方式，就像 Maven 整合了所有的 Jar 包，Spring Boot 整合了所有的框架。

Spring Boot是一个快速的开发框架,能够帮助程序员快速整合第三方框架，内置了第三方容器(tomcat/jetty/undertom),完全简化编写xml,采用是注解方式。

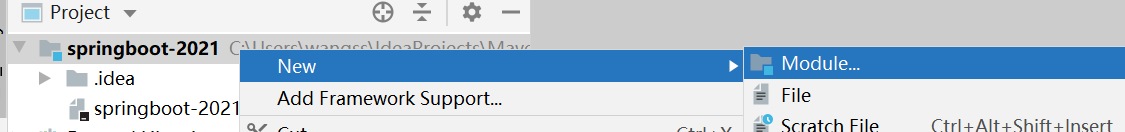
优势：

* 快速构建项目
* 对主流开发框架的无配置集成
* 项目可独立运行，无须外部依赖Servlet容器（例如 tomcat，jar包方式）
* 提供运行时的应用监控
* 极大的提高了开发、部署效率
* 与云计算的天然集成（服务编排技术：k8s）

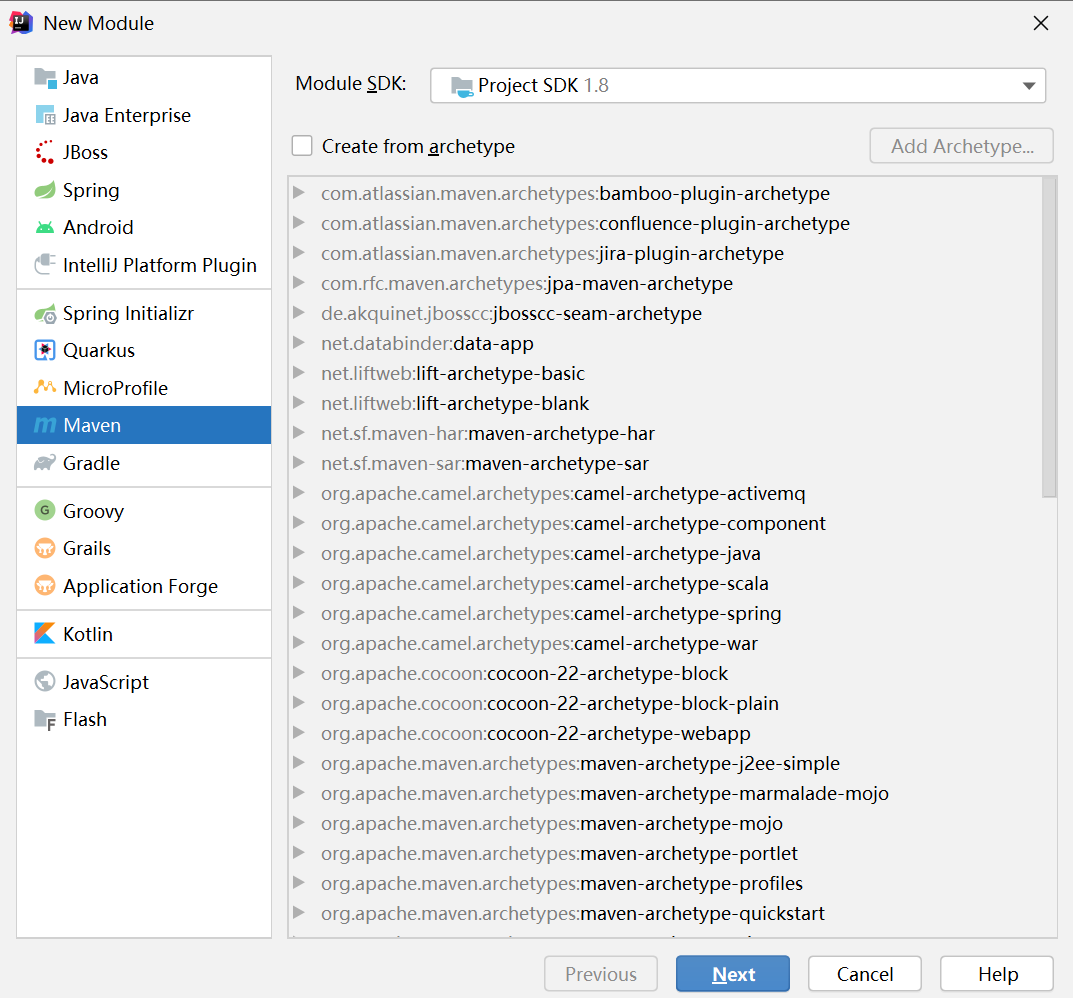
# 2第一个SpringBoot项目

## 2.1maven形式创建

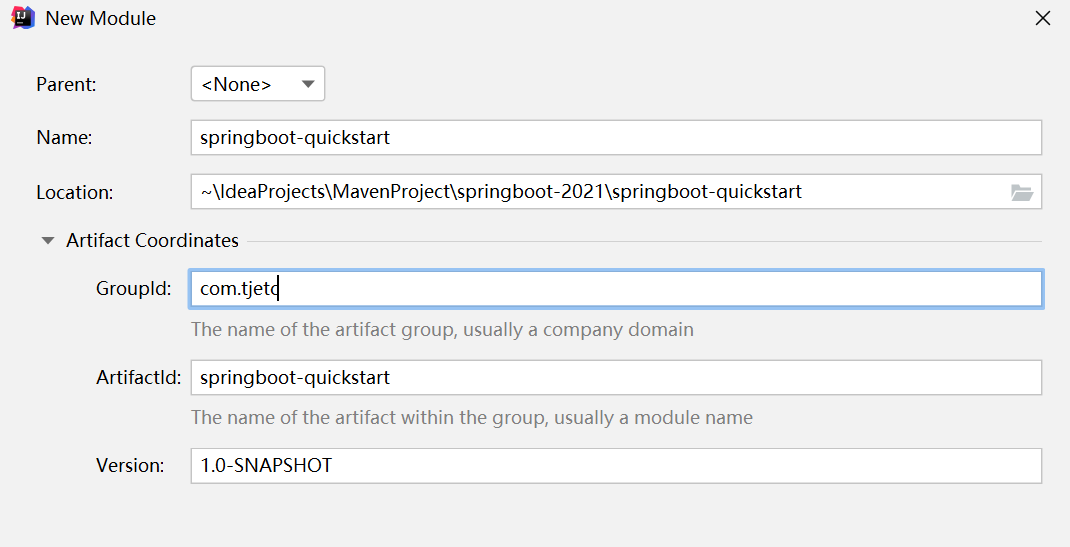
### 2.1.1创建maven工程



选择maven，直接next



点击next，填写相应信息：



点击finish。

### 2.1.2导入依赖

|  |
| --- |
| *<!-- 导入springboot版本和框架依赖 -->* <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.6.3</version>  <relativePath></relativePath> </parent> *<!-- 导入动态web场景启动器 -->* <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> </dependencies>  *<!--添加maven插件，项目的打包工具，打成jar包，否则在打包运行时报错 -->* <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins> </build> |

### 2.1.3创建springboot启动类

创建包com.tjetc，在包下创建启动类

创建启动类

|  |
| --- |
| package com.icss.chapter01;   import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication public class HelloWorldApplication {  public static void main(String[] args) {  SpringApplication.*run*(HelloWorldApplication.class, args);  } } |

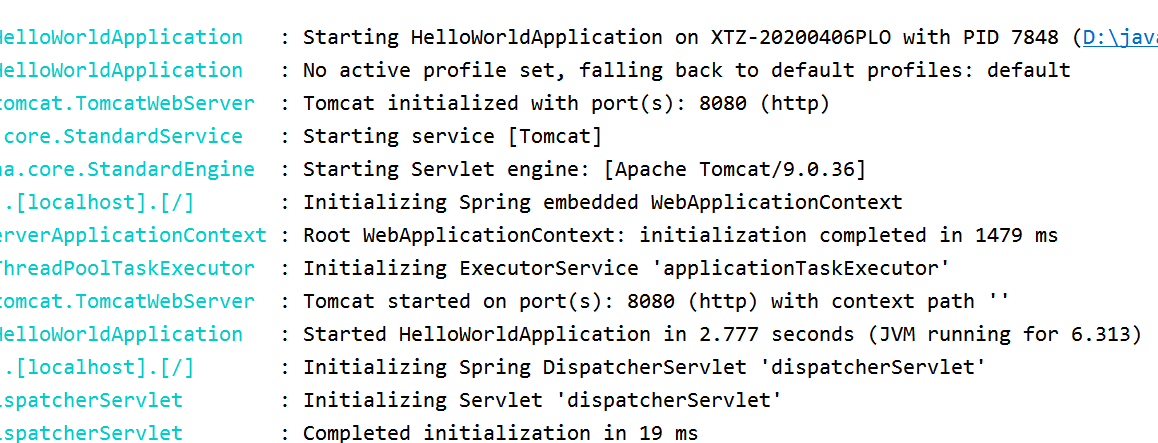
### 2.1.4创建controller

创建com.tjetc.controller包，在包下创建controller类

|  |
| --- |
| @Controller public class HelloWorldController {   @RequestMapping("/hello")  @ResponseBody  public String hello(){  return "Hello Spring Boot!";  } } |

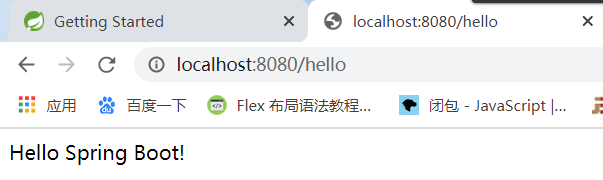
### 2.1.5启动springBoot

运行启动类的main方法

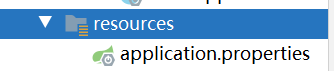


启动成功，端口号8080

### 2.1.6访问controller

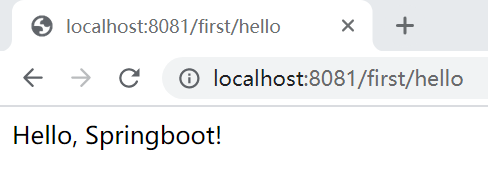


在resources下创建配置文件，application.properties,springboot启动以后，默认读取该配置文件。配置端口号，contextpath



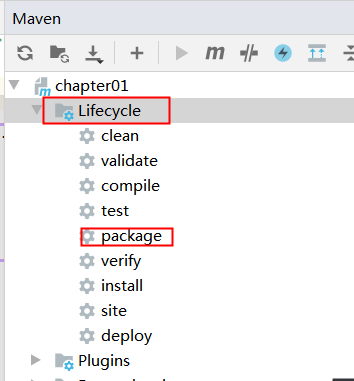
|  |
| --- |
| server.port=8081 server.servlet.context-path=/first |

访问：

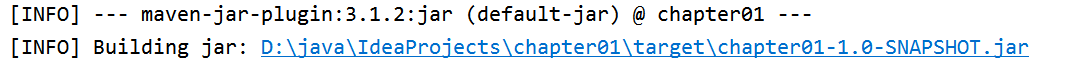


### 2.1.7打包部署

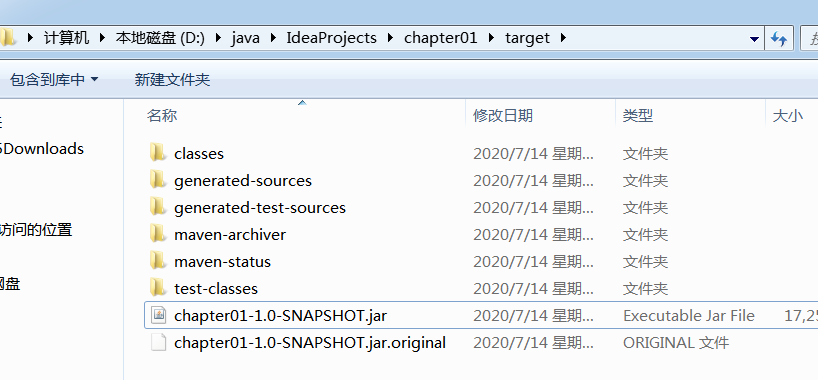
Springboot工程被打包成jar包，通过package命令打成jar包。



将当前的工程打成jar包



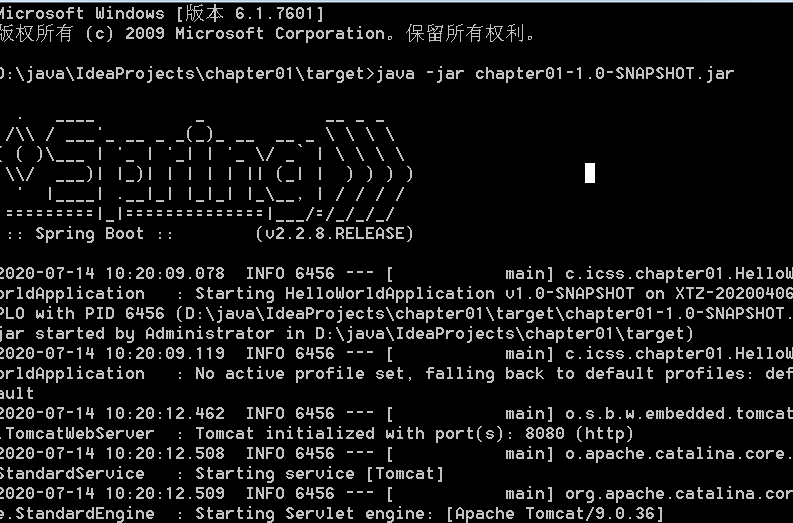
放在target下面。



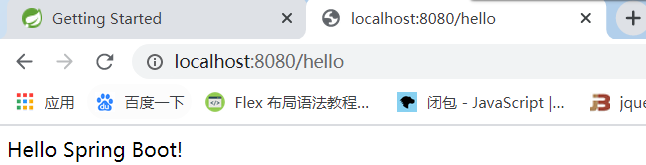
/usr/local >java -jar jar包名称

Dfsd> java -jar /usr/local/jar包的名称

在当前的路径下打开cmd窗口。运行jar包：

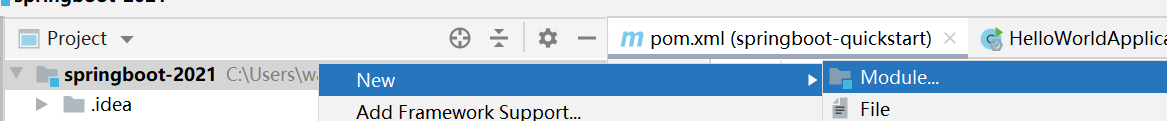


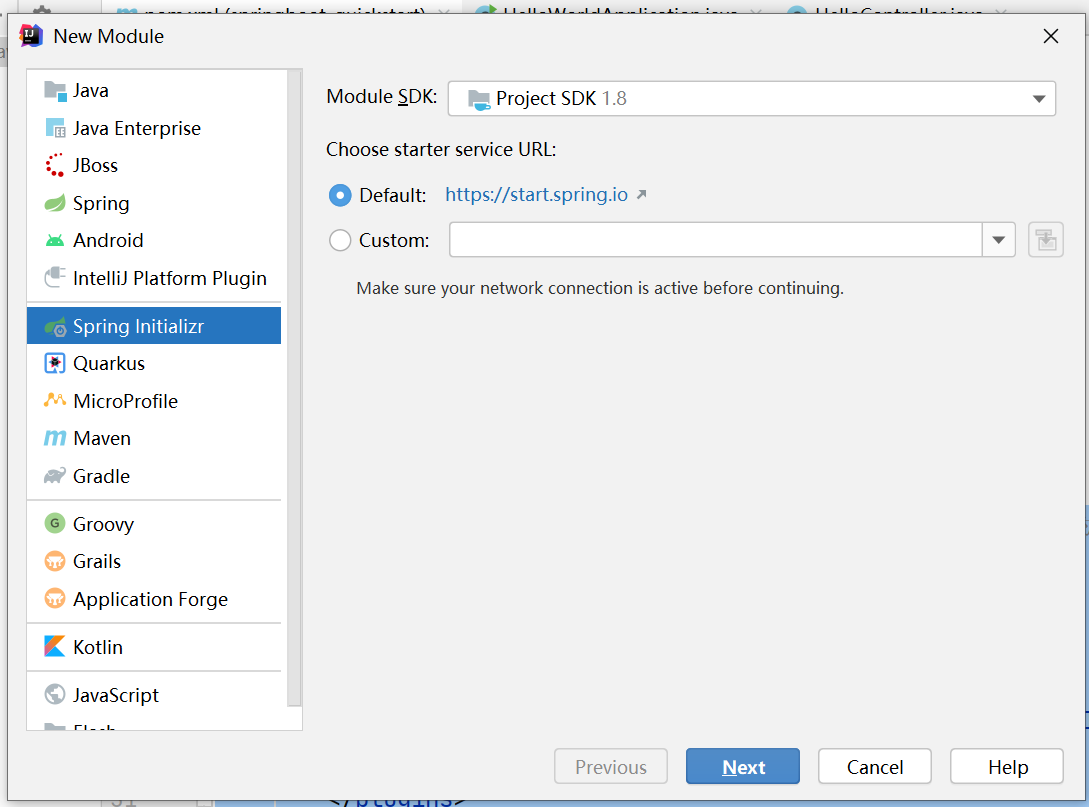
访问controller



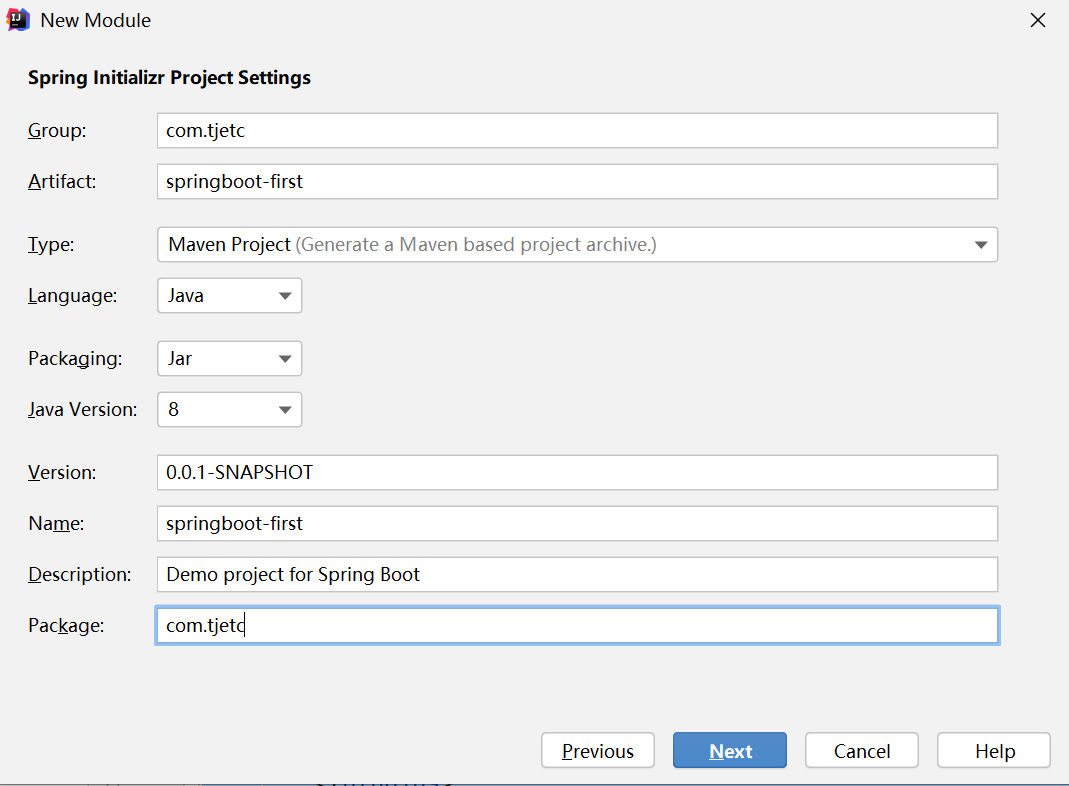
## 2.2引导器快速创建

创建project

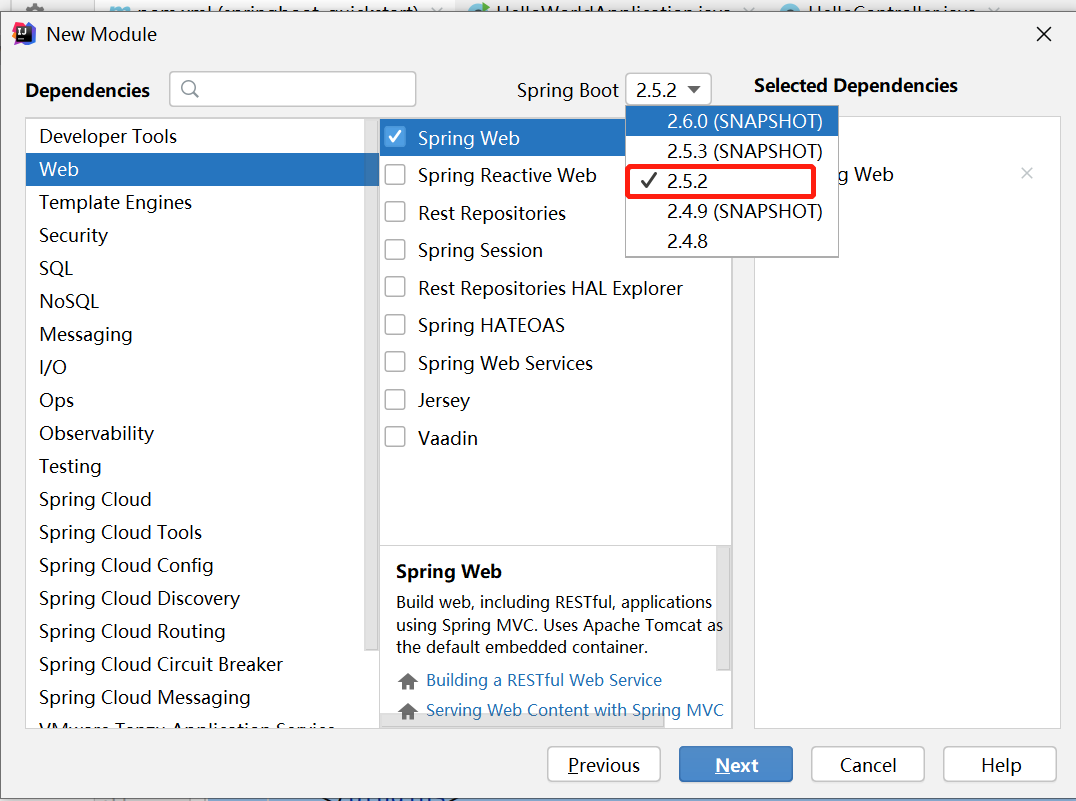




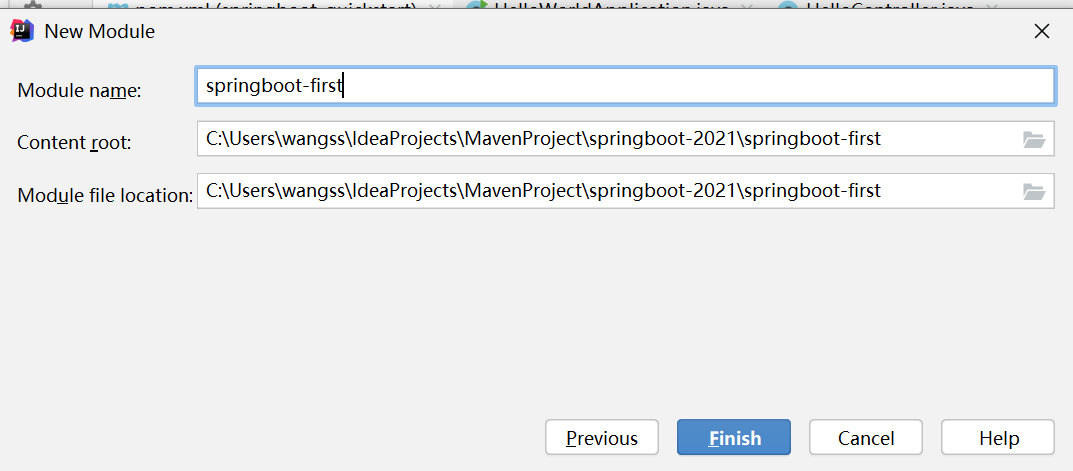
点击next：填写group和Artifact，确定打包方式，jdk版本，和package表示的启动类所在的包名,**如果连接失败，可以使用Custom:阿里的镜像服务Custom:https://start.aliyun.com**



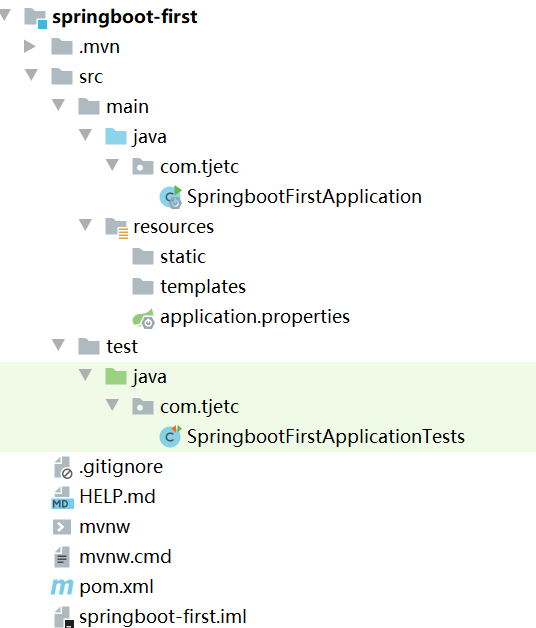
没有问题，点击next。选择依赖：创建web工程，只要选择web依赖即可，选择springboot版本，选择当前稳定版本。点击next。



确定module信息以后，点击finish



创建工程以后，查看工程的目录结构：



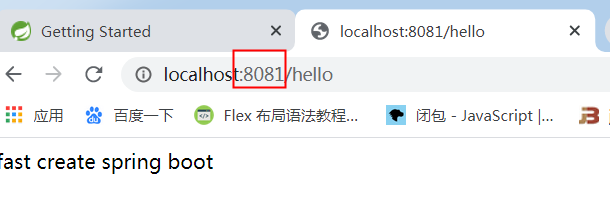
自动创建了包：com.tjetc，并在包下创建了启动类和测试类。同时在resources下创建了static，templates和springboot的主配置文件application.properties，在启动类启动时，会自动读取主配置文件中的数据。Springboot内部集成了tomcat，默认端口号8080，可以在主配置文件中指定端口号：在application.properties中添加如下代码：

|  |
| --- |
| *#指定端口号* server.port=8081 |

在pom.xml中导入了parent，web依赖和单元测试，以及打包工具。

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.6.3</version>  <relativePath/> *<!-- lookup parent from repository -->* </parent>  。。。。。  <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> </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins> </build> |

启动springboot：此时端口号为8081。



## 2.3理解Pom文件的依赖与starter启动器的作用

### 2.3.1Pom文件的依赖

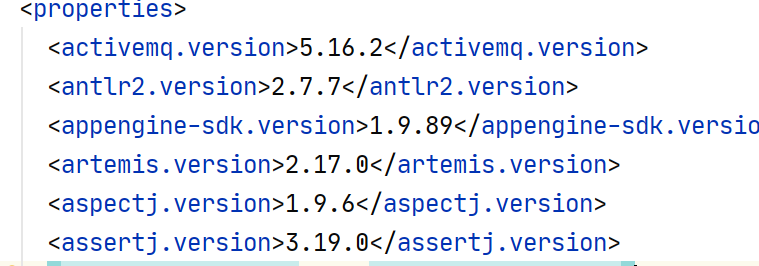
Pom文件中的parent是Spring Boot的框架版本控制中心

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.6.3</version>  <relativePath/> *<!-- lookup parent from repository -->* </parent> |
|  |

点进去看一下parent是如何控制版本的。点进去之后，也有一个父工程

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>2.6.3</version> </parent> |

父工程为spring-boot-dependencies，看到名字应该想到，是springboot的依赖。同时还有其他东西。再点进去，看到里面有<dependencyManagement>，管理各种依赖的版本。





到这里就理解了parent是如何控制版本的。我们创建的springboot工程是spring-boot-start-parent的子工程，spring-boot-starter-parent是spring-boot-denpendencies的子工程，父工程通过dependencyManagement控制了各种依赖的版本。所以当子工程导入依赖时，可以不写版本，自动使用父工程规定的版本。以此来进行版本的控制。

### 2.3.2spring-boot场景启动器

starter：spring-boot场景启动器，以web启动器为例：

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

点进去看到spring-boot-starter-web的内容：

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter</artifactId>  <version>2.6.3</version>  <scope>compile</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-json</artifactId>  <version>2.6.3</version>  <scope>compile</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  <version>2.6.3</version>  <scope>compile</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>5.3.15</version>  <scope>compile</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>5.3.15</version>  <scope>compile</scope>  </dependency> </dependencies> |

Spring-boot-starter-web导入了web环境所有的依赖，只需导入starter，可自动导入web模块正常运行所依赖的组件。其他的starter也是一样的。

springboot出厂默认就写好了很多starter,如：

spring-boot-starter-activemq，spring-boot-starter-aop，spring-boot-starter-data-redis，spring-boot-starter-data-solr等

重要提示：Spring Boot将所有的绝大部分框架整合场景都进行了抽取，做成一个个的starters（启动器），只需要在项目里面引入这些starter相关整合所需的依赖都会导入进来。

## 2.4理解主程序@SpringBootApplication

@SpringBootApplication 用于标识spring boot应用程序，代表该类是一个spring boot启动类

Spring boot运行这个类的main方法时启动SpringBoot应用。

|  |
| --- |
| @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan public @interface SpringBootApplication {} |

@SpringBootConfiguration: Spring Boot的配置类。标注在类上表示是一个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。配置类也是容器中的一个组件。

@Component把组件实例化到spring容器中。

@EnableAutoConfiguration：开启自动配置功能；

当我们需要Spring Boot帮我们自动配置所需要的配置，@EnableAutoConfiguration告诉Spring Boot开启自动配置功能，这样Spring Boot会自动配置好并使之生效。

# 3SpringBoot配置文件

## 3.1学会Spring Boot全局配置和yaml的语法

Spring Boot全局配置文件（在src/main/resources目录或者类路径/config下），名称如下：

application.properties

application.yaml/yml

也许作者认为properties或json 的写法不爽，于是发明了yml这种以数据为中心写法的配置文件。

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

properties:*#指定端口号*server.port=8081

yml:

*#指定端口号*server:  
 port: 8082

### 3.1.1语法

参考语法规范：http://www.yaml.org

语法校验 : https://nodeca.github.io/js-yaml

YAML基本语法

A.使用缩进表示层级关系

B.缩进时不允许使用Tab键，只允许使用空格。

C.缩进的空格数目不重要，只要相同层级的元素左侧对齐即可

D.大小写敏感

**E.键与值之间一定要有空格**

### 3.1.2yml支持的三种数据的结构

YAML 支持的三种数据结构

常见普通值：单个的、不可再分的值

对象：键值对的集合

数组：一组按次序排列的值

1. 单个的，不能再分割的值

|  |
| --- |
| #第一种，不能再分割的值  name: rose |

2.对象：键值对的集合

|  |
| --- |
| #第二种，对象，键值对的集合  person:  name: zhangsan  age: 20  sex: male |

将对象写在一行上，键与值之间也要有空格

|  |
| --- |
| #一行  user: {name: zhangsan, age: 20, sex: male} |

3.数组：一组按次序排列的值,-与值之间也要空格

|  |
| --- |
| #第三种，数组或集合，一组按次序排列的值  names:  - zhangsan  - lisi  - wangwu |

names: [ 'zhangsan', 'lisi', 'wangwu' ]

数组中存的是对象

users: [ { name: 'tom', age: 2 }, { name: 'jerry', age: 3 } ]

|  |
| --- |
| users:  - {name: tom, age: 2}  - {name: jerry, age: 3} |

name和age的左侧要对齐

|  |
| --- |
| users:  - name: tom  age: 2  - name: jerry  age: 3 |

数组如果放在一行：

|  |
| --- |
| lists: [zhangsan,lisi,wangwu] |

## 3.2Spring Boot获取配置文件的值及配置文件编码设置

### 3.2.1创建实体类

|  |
| --- |
| public class Book {  private String bookName;  private String author;   public String getBookName() {  return bookName;  }   public void setBookName(String bookName) {  this.bookName = bookName;  }   public String getAuthor() {  return author;  }   public void setAuthor(String author) {  this.author = author;  }   @Override  public String toString() {  return "Book{" +  "bookName='" + bookName + '\'' +  ", author='" + author + '\'' +  '}';  } } |
| @Component*//将student对象放入容器 //读取主配置文件中的数据，赋值给student对象* @ConfigurationProperties(prefix = "student")*//需要属性，指定主配置文件中的数据* public class Student {   private String studentName;  private Integer age;  private Boolean sex;*//男代表true* @DateTimeFormat(pattern = "yyyy-MM-dd")  private LocalDate birth;  private Map<String, Object> maps;  private List<Object> lists;  private Book book;   public String getStudentName() {  return studentName;  }   public void setStudentName(String studentName) {  this.studentName = studentName;  }   public Integer getAge() {  return age;  }   public void setAge(Integer age) {  this.age = age;  }   public Boolean getSex() {  return sex;  }   public void setSex(Boolean sex) {  this.sex = sex;  }   public LocalDate getBirth() {  return birth;  }   public void setBirth(LocalDate birth) {  this.birth = birth;  }   public Map<String, Object> getMaps() {  return maps;  }   public void setMaps(Map<String, Object> maps) {  this.maps = maps;  }   public List<Object> getLists() {  return lists;  }   public void setLists(List<Object> lists) {  this.lists = lists;  }   public Book getBook() {  return book;  }   public void setBook(Book book) {  this.book = book;  }   @Override  public String toString() {  return "Student{" +  "studentName='" + studentName + '\'' +  ", age=" + age +  ", sex=" + sex +  ", birth=" + birth +  ", maps=" + maps +  ", lists=" + lists +  ", book=" + book +  '}';  } } |

### 3.2.2在主配置文件中配置student

|  |
| --- |
| *#配置student studentName--> student-name* student:  student-name: rose  age: 23  sex: false  birth: 2000-01-08  maps:  k1: v1  k2: v2  lists:  - lisi  - zhangsan  book:  book-name: "中软国际java典型案例"  author: "中软卓越研究院" |

### 3.2.3编写测试类

单元测试依赖

|  |
| --- |
| *<!--单元测试-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope> </dependency> <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter</artifactId>  <scope>test</scope> </dependency> <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope> </dependency> |

|  |
| --- |
| *//调用spring底层的单元测试* @RunWith(SpringRunner.class) *//使用springboot环境测试* @SpringBootTest class Chapter02ApplicationTests {   @Autowired  private Student student;   @Test  void contextLoads() {  System.*out*.println(student);  }  } |

### 3.2.4执行结果

Student{studentName='rose', age=23, sex=false, birth=Sat Jan 08 00:00:00 CST 2000, maps={k1=v1, k2=v2}, lists=[lisi, zhangsan], book=Book{bookName='中软国际java典型案例', author='中软卓越研究院'}}

启动springboot测试类，会自动加载主配置文件，@ConfigurationProperties负责将主配置文件中的数据赋值给student对象。Student对象已经通过@Component注解注册到容器中，所以在测试类中可以直接注入。

@ConfigurationProperties 映射实体的属性值

可以为实体对读入配置文件的值,支持所有类型值的读取;

前提是实体类需要提供一个setter或使用可变的值初始化它.

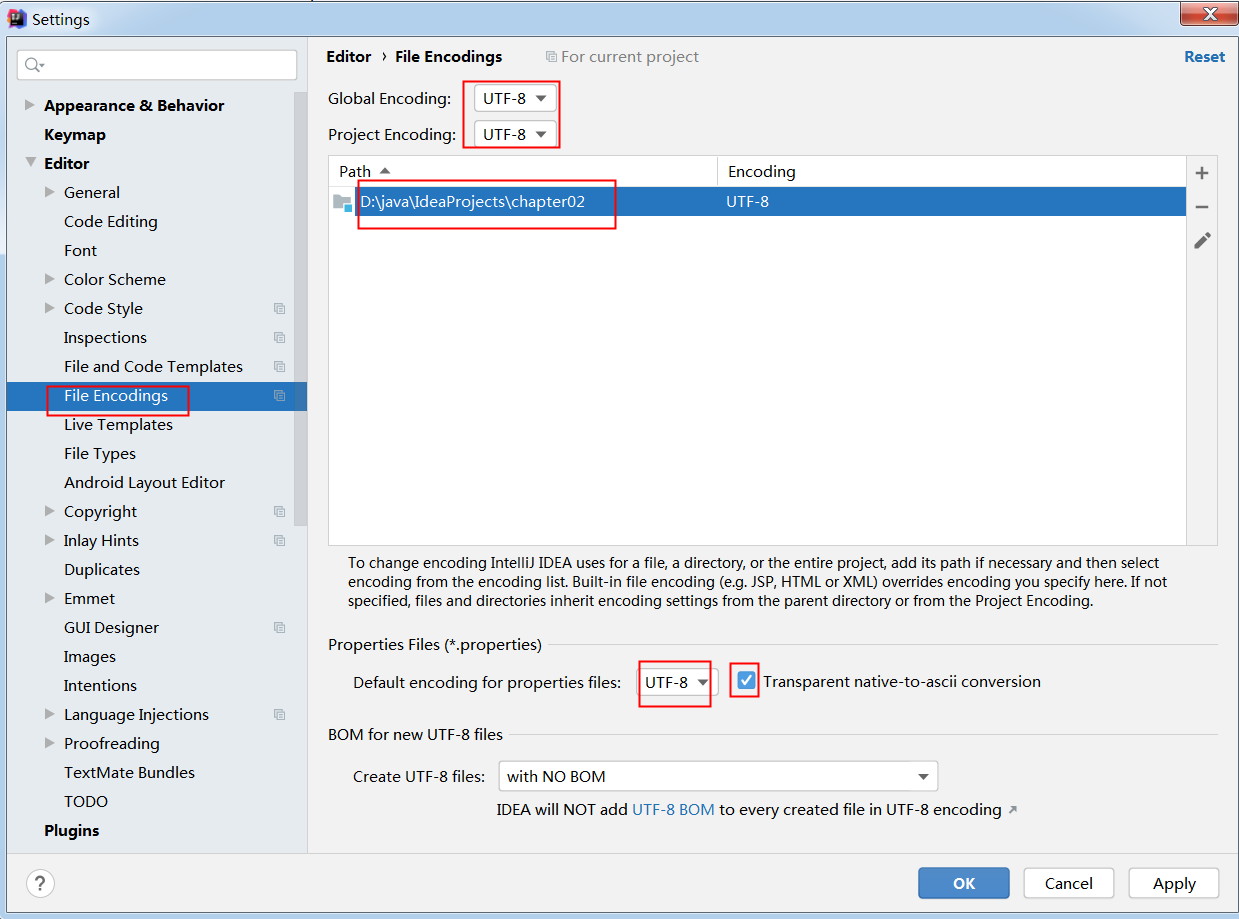
在application.properties中配置student

|  |
| --- |
| *#配置student* student.student-name=jerry student.age=25 student.sex=true student.birth=1998-10-08 student.maps.k1=v1 student.maps.k2=v2 student.lists=lisi,wangwu student.book.bookName="中软国际java典型案例" student.book.author="中软卓越研究院" |

最终执行结果：

|  |
| --- |
| Student{studentName='jerry', age=25, sex=true, birth=Thu Oct 08 00:00:00 CST 1998, maps={k1=v1, k2=v2}, lists=[lisi, wangwu], book=Book{bookName='"ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½javaï¿½ï¿½ï¿½Í°ï¿½ï¿½ï¿½"', author='"ï¿½ï¿½ï¿½ï¿½×¿Ô½ï¿½Ð¾ï¿½Ôº"'}} |

在使用properties文件的时候，中文乱码。如何解决?修改properties文件的编码，将所有编码类型都设置为utf-8

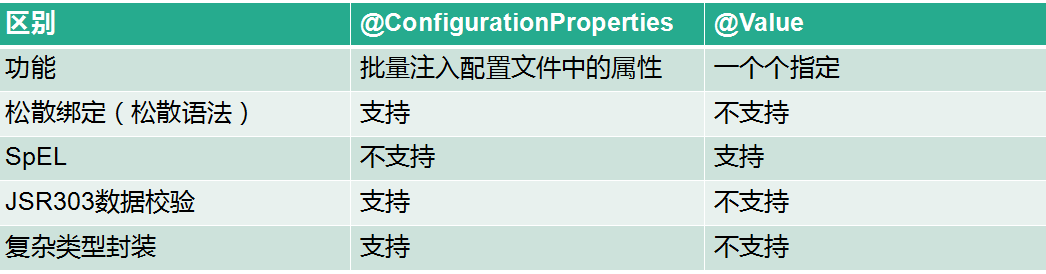


都改好以后，点击OK。

原来的application.properties就乱码了，修改乱码，重新运行，乱码问题解决了

|  |
| --- |
| Student{studentName='jerry', age=25, sex=true, birth=Thu Oct 08 00:00:00 CST 1998, maps={k1=v1, k2=v2}, lists=[lisi, wangwu], book=Book{bookName='"中软国际java典型案例"', author='"中软卓越研究院"'}} |

## 3.3@ConfigurationProperties与@Value区别



松散语法属性名匹配规则:

student.studentName：使用标准方式

student.student-name：小写写用-

student. STUDENT\_NAME：大写用\_

3.3.1功能不同

通过前面的例子我们知道，@ConfigurationProperties是可以批量注入的，一次搞定

而@Value只能一个一个注入，如下：

|  |
| --- |
| *//@Value：注意一个容器已经存在的对象 @Value("${student.student-name}")* @Value("${student.student-name}") private String studentName; @Value("${student.age}") private Integer age; @Value("true") private Boolean sex;*//男代表true* |

### 3.3.1松散绑定

@ConfigurationProperties支持松散绑定studentName可以写为student-name都没有问题。而@Value不可以。@Value进行绑定时，名称只能与application.properties中属性的名称相同

### 3.3.2SpEL

@ConfigurationProperties不支持SpEL，比如：

|  |
| --- |
| *#配置student* student.student-name=jerry student.age=#{18+5} //这里是直接报错 student.sex=true |

Yml文件配置不支持SpEL

|  |
| --- |
| student:  student-name: rose@qq.com  age: *#{18+8} #不会报错，但是不能注入到student的age中* sex: false |

@Value支持SpEL。比如：

|  |
| --- |
| @Value("#{18+8}") private Integer age; |

Age是可以正常注入的。

### 3.3.3复杂类型封装

@ConfigurationProperties支持复杂类型封装

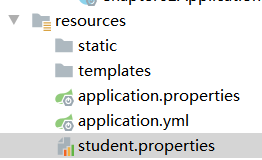
@Value不支持复杂类型，只支持字符串，和基本数据类型及其包装类。

## 3.4 @PropertySource@Bean@Configuration@Import

### 3.4.1@PropertySource

读取指定的properties配置文件

如果我们将student的数据写在任意一个配置文件中，比如。student.properties文件中，如何读取配置中信息，注入到student中。可以使用该注解



|  |
| --- |
| */\*\*  \* 创建student对象，注册到spring容器中  \*/* @Component */\*\*  \* 只要springboot启动，就会读取application.yml这个配置文件  \* 该注解的作用：读取配置文件中以student开头的属性，注入到Student对象中  \*/* @ConfigurationProperties(prefix = "student")*//复杂属性文件内部的属性的获取，如果不通过@PropertySource指定属性 //文件，默认读取的就是application.properties或者application.yml文件* @PropertySource("classpath:student.properties")*//指定属性文件*  public class Student {    } |

### [3.4.2@Bean、@Configuration](mailto:3.4.3@Bean、@Configuration)

Spring Boot 由于没有XML文件，所以所有的Bean管理都放入在一个配置类中实现。

配置类就是类上具有@Configuration的类。这个类就相当于之前的applicationContext.xml

配置类中的bean注入到容器

写一个config包，用来存放配置类：

创建配置类：

|  |
| --- |
| import com.icss.chapter02.service.StudentService; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; *//代表当前类是一个配置类，可以达到跟配置文件相同的效果 //创建一个*userService*的对象，放在容器中* @Configuration public class MyAppConfig {   *//将方法的返回值存入spring容器，bean对象的id是方法名*  *//作用，通过java代码将组建注册到spring容器中  //相当于<bean>的作用,* @Bean  public UserService userService (){  return new StudentService();  }  } |

测试：

|  |
| --- |
| @Autowired private UserService userService;  @Test public void testService(){  userService.doSomething(); } |

结果：

|  |
| --- |
| do something........ |

|  |
| --- |
| *//@Configuration 相当于xml配置文件中 <beans> 管理bean的* @Configuration public class MyAppConfig {    *//@Configuration+@Bean 方法的执行时机，是springboot启动的时候，执行@Bean对应的方法  //如果@Bean对应的方法是非静态方法，那么只执行一次，bean按照单例处理，如果是静态方法，执行多次，生成的bean是多例的  //@Bean 相当于xml配置文件<bean>标签，生成实例对象交给spring管理，默认是单例* @Bean  public static UserService userService(){  System.*out*.println("MyAppConfig.userService().....");  *//生成对象实例* return new UserService();  }   @Bean  public String abc(){  System.*out*.println("MyAppConfig.abc()...");  *userService*();  *userService*();  *userService*();  *userService*();  return "aaa";  } } |

### [3.4.3@Import](mailto:3.4.4@Import)

@Import可以引入一个或多个类型，代表将该类实例化到IOC容器中。可以用在启动类上，或者配置类上，只能在程序启动的时候，能够读取该注解即可，这样该注解表示的类型就会被实例化到容器中。

创建db.properties属性文件

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver jdbc.url=jdbc:mysql://localhost:3306/springboot jdbc.user=root jdbc.password=123456 |

创建MyDataSource类，读取db.properties属性文件，并为属性赋值。

|  |
| --- |
| package com.icss.config;   import org.springframework.beans.factory.annotation.Value; import org.springframework.context.annotation.PropertySource;  */\*\*  \* 指定读取的属性文件，这里仅是读取属性文件，赋值，并不会把MyDataSource实例化到容器当中  \*/* @PropertySource("classpath:db.properties") public class MyDataSource {   *//指定属性文件中的相应属性* @Value("${jdbc.driver}")  private String driver;  @Value("${jdbc.url}")  private String url;  @Value("${jdbc.user}")  private String user;  @Value("${jdbc.password}")  private String password;   public String getDriver() {  return driver;  }   public void setDriver(String driver) {  this.driver = driver;  }   public String getUrl() {  return url;  }   public void setUrl(String url) {  this.url = url;  }   public String getUser() {  return user;  }   public void setUser(String user) {  this.user = user;  }   public String getPassword() {  return password;  }   public void setPassword(String password) {  this.password = password;  }   @Override  public String toString() {  return "DataSource{" +  "driver='" + driver + '\'' +  ", url='" + url + '\'' +  ", user='" + user + '\'' +  ", password='" + password + '\'' +  '}';  } } |

在配置类上，通过@Import注解引入MyDataSource，当配置类加载时，就会加载MyDataSource

|  |
| --- |
| package com.icss.config;  import com.icss.service.UserService; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; import org.springframework.context.annotation.Import;  @Configuration*//代表这是一个配置类，该类的方法的返回值就是要实例化到容器中的bean* @Import(MyDataSource.class)*//引入MyDataSource，这样MyDataSrouce就会实例化到容器中了。 //除了@Import也可以i使用@Bean来实现* public class AppConfig {   */\*\*  \* @Bean的作用就相当于<bean>的作用，如果@Bean不指定名称，方法名就是对象的id  \* 如果指定名称，指定的是多少就是多少  \* @return  \*/* @Bean  public UserService userService(){  return new UserService();  } } |

测试：

|  |
| --- |
| @RunWith(SpringRunner.class) @SpringBootTest public class TestDataSource {   @Autowired  private MyDataSource myDataSource;   @Test  public void testDataSource(){  System.*out*.println(myDataSource);  } } |

结果：

|  |
| --- |
|  |

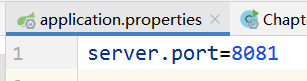
## 3.5多文件配置,加载顺序与位置

多文件配置:spring boot为了适应开发环境或生产环境的变化,专门打造profile通过指定参数来快速切换环境!

### 3.5.1.多文件的形式

格式: application-{profile}.properties 或 application-{profile}.yml

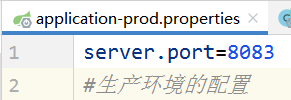
application.properties:



application-dev.properties:



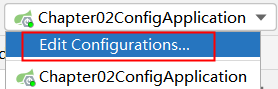
application-prod.properties:

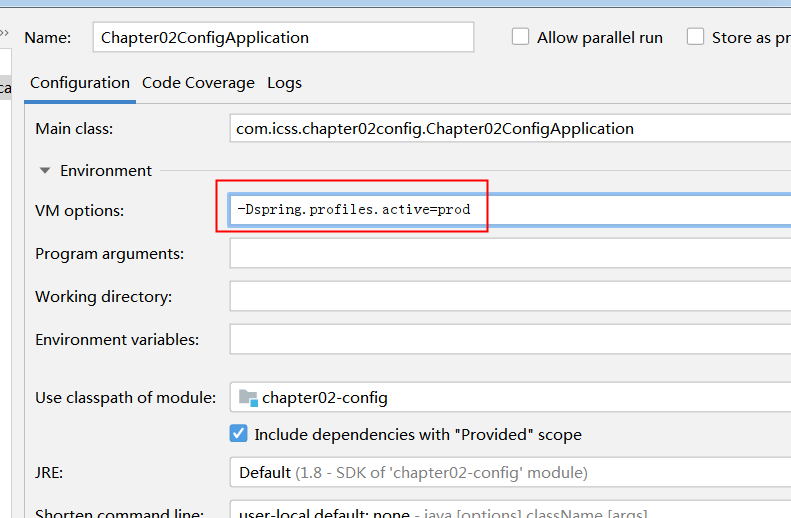


Spring boot提供几种的激活配置方式

### 3.5.2Spring boot提供几种的激活配置方式

一．JVM参数 -Dspring.profiles.active=dev



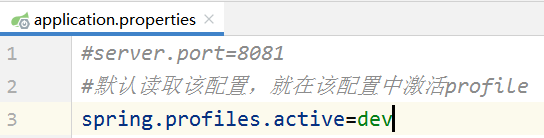


确定，启动springboot程序即可。

这时激活的是prod，端口号8083

1. 配置文件 spring.profiles.active=prod

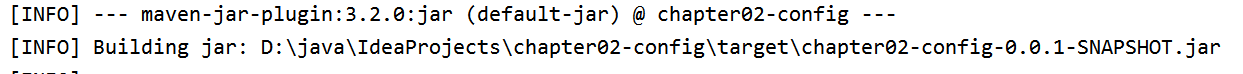
在全局配置文件application.properties中激活相应的profile即可



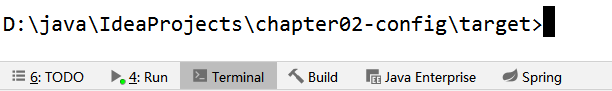
此时，dev被激活，端口号为8082

1. 命令行 : --spring.profiles.active=prod

先打包应用。



打开terminal，进入命令行，进入target目录：



运行jar包，激活profile



此时，prod被激活，端口号为8083

### 3.5.3 启动SpringBoot加载指定配置文件

如果不希望使用默认的application.properties作为配置文件名，可以通过指定spring.config.name环境属性来切换其他的名称。 也可以使用spring.config.location环境属性来引用一个明确的路径（目录位置或文件路径列表以逗号分割）

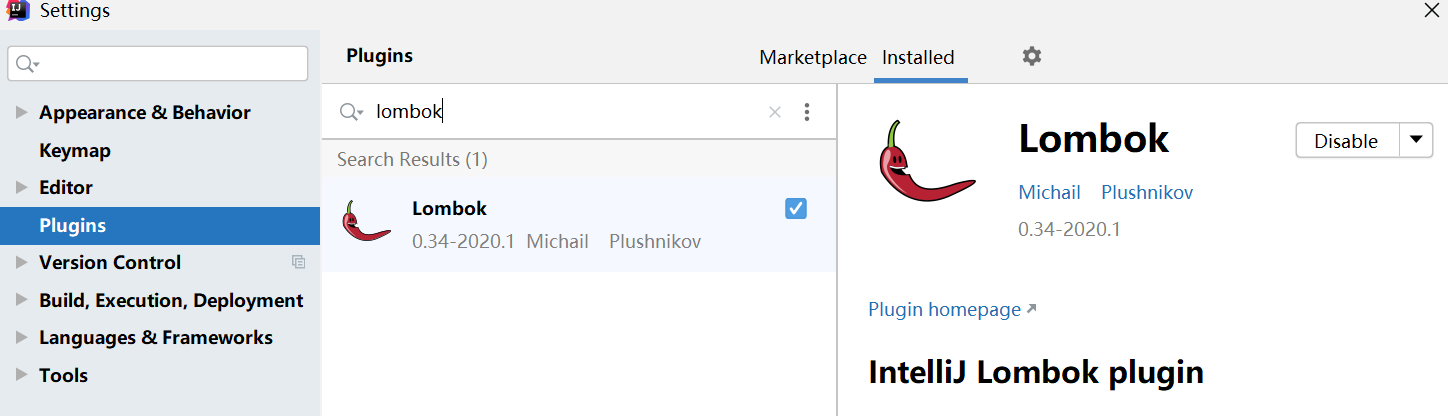
java -jar myproject.jar --spring.config.location=c:/application.properties

# 4SpringBoot整合MyBatis

## 4.1lombok

为了简化实体类的写法，可以使用lombok

### 4.1.1安装Lombok插件



### 4.1.2导入lombok依赖

|  |
| --- |
| <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>1.18.20</version> </dependency> |

### 4.1.3使用lombok注解

|  |
| --- |
| import lombok.Data;  @Data public class User {   private Long id;  private String name;  private String password; } |

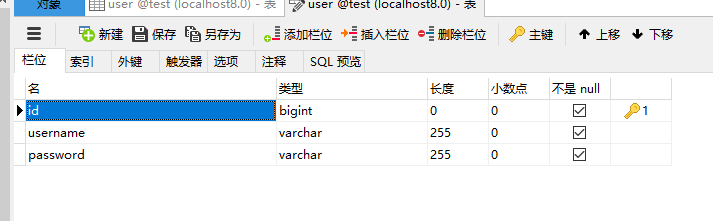
在User上加上@Data注解，会自动拥有set方法，get方法，equals和hoshcode方法，toString方法。

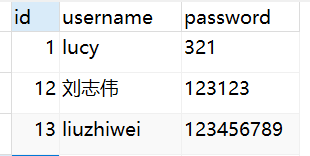
## 4.2整合Mybatis注解开发

### 4.2.1导入依赖

|  |
| --- |
| <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>2.2.0</version> </dependency> <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId> </dependency> |

### 4.2.2创建user表





### 4.2.3创建实体类

|  |
| --- |
| @Data public class User {  private Long id;  private String username;  private String password; } |

### 4.2.4创建Mapper

|  |
| --- |
| public interface UserMapper {  @Select("select \* from user")  List<User> selectList(); } |

### 4.2.5创建Service

|  |
| --- |
| public interface UserService {   List<User> findAll(); } |

### 4.2.6创建service实现类

|  |
| --- |
| @Service public class UserServiceImpl implements UserService {   @Autowired  private UserMapper userMapper;   @Override  public List<User> findAll() {  return userMapper.selectAll();  } } |

### 4.2.7编写controller

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

### 4.2.8添加mapper扫描

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

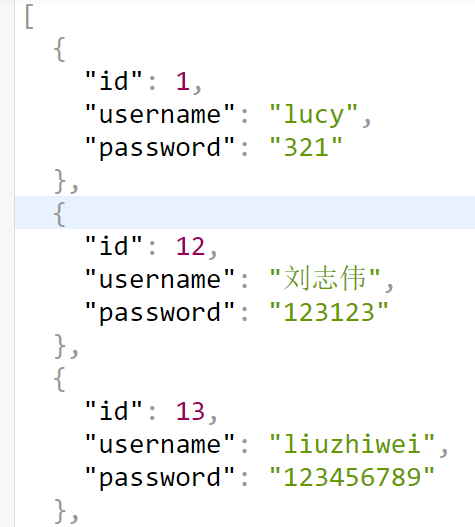
如果在UserMapper接口上添加@Mapper注解，则不需要添加mapper扫描。二者用一种就可以。

### 4.2.9添加配置

在application.properties中配置datasource

|  |
| --- |
| spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver spring.datasource.url=jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai  spring.datasource.username=root spring.datasource.password=123456 |

### 4.2.10启动测试



## 4.3基于xml整合mybatis

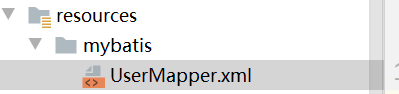
以查询所有用户为例。

### 4.3.1编写UserMapper

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

### 4.3.2编写mapper映射

在resources下创建mybatis目录，专门用来存放mybatis相关配置。



|  |
| --- |
| *<?*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.tjetc.dao.UserMapper">   <select id="selectList" resultType="User">  select *\** from user  </select>  </mapper> |

### 4.3.3添加mybatis配置

在springboot的配置文件，接管mybatis核心配置：

|  |
| --- |
| spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver spring.datasource.url= jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai spring.datasource.username=root spring.datasource.password=123456  #配置mapper映射文件的位置 mybatis.mapper-locations=classpath:mybatis/\*Mapper.xml  #别名扫描 mybatis.type-aliases-package=com.tjetc.entity |

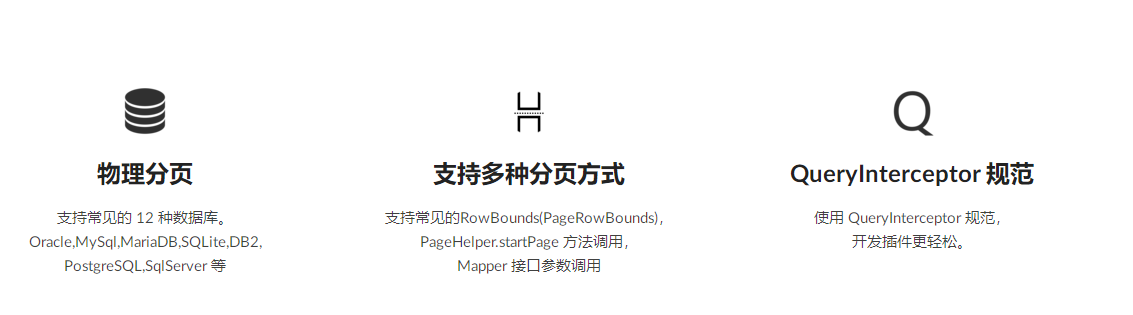
### 4.3.4启动测试

实现跟注解相同的效果。

## 4.4整合PageHelper



PageHelper是一款犀利的Mybatis分页插件，使用了这个插件之后，分页开发起来更加简单容易。



Spring Boot整合PageHelper不需要做任何配置文件的配置，添加依赖后就可以直接使用。

### 4.4.1导入依赖

|  |
| --- |
| <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper-spring-boot-starter</artifactId>  <version>1.4.3</version>  </dependency> |

### 4.4.2准备数据

在service中实现分页，以查询所有用户为例：



### 4.4.3编写分页service

|  |
| --- |
| */\*\*  \* 查询分页数据  \* @param pageNum  \* @param pageSize  \* @return  \*/* PageInfo<User> findPage(int pageNum, int pageSize); |
| @Override public PageInfo<User> findPage(int pageNum, int pageSize) {  *//第一页，每页显示5条数据* PageHelper.*startPage*(pageNum,pageSize);  *//查询所有用户* List<User> all = userMapper.findAll();  *//获取Page* PageInfo<User> page = new PageInfo<>(all);  return page; } |

### 4.4.4编写测试

|  |
| --- |
| @Autowired private UserService userService;  @Test public void testPage(){  *//每页5条数据，获取第一页的数据* PageInfo<User> page = userService.findPage(1, 5);  *//获取当前页* System.*out*.println(page.getPageNum());  *//获取每页显示几条* System.*out*.println(page.getPageSize());  *//获取每页的数据* List<User> list = page.getList();  for(User user: list){  System.*out*.println(user);  }  } |

### 4.4.5结果

|  |
| --- |
| 1  5  User(id=1, name=rose, password=123456)  User(id=2, name=jack, password=123456)  User(id=3, name=tom, password=123456)  User(id=4, name=张三, password=123456)  User(id=5, name=李四, password=123456) |

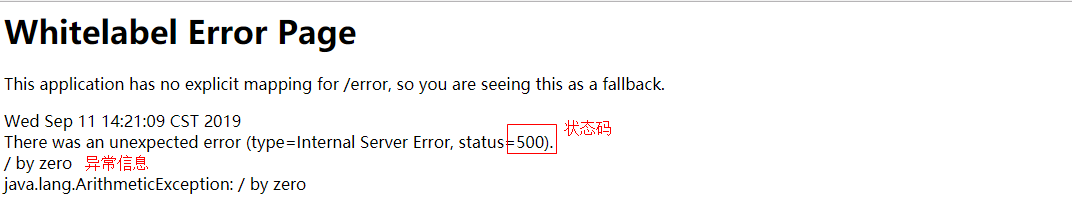
# 5SpringBoot异常处理

## 5.1异常显示的页面

加入thymeleaf依赖

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

默认情况下，SpringBoot 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更加友好的错误页面

下面我们来看具体是如何实现的。

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

|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>500</title> </head> <body>  您的代码出现了内部错误，请检查！ </body> </html> |

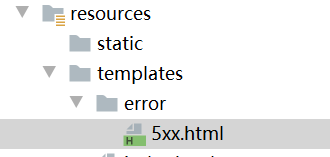
1. 创建controller

|  |
| --- |
| @Controller public class ErrorController {   @RequestMapping("/test")  public String testError(){  System.*out*.println(5/0);  return "index";  } } |

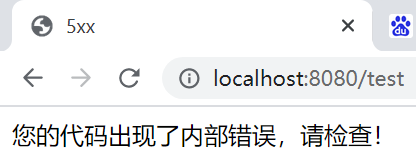
我们知道，在运行上面代码的时候发生算术异常。错误码为500。

1. 使用X进行模糊匹配
   1. 当出现5开头状态码的错误时，显示页面可以命名为5xx.html
   2. 如果500.html和5xx.html同时存在，则会精确匹配相应页面。

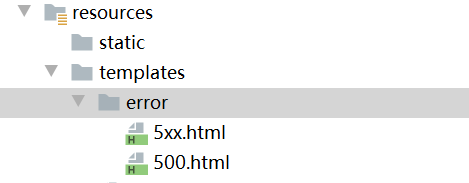
我们把刚才的500.html改为5xx.html，也是可以的。因为发生了5开头的异常，会走5xx.html。



执行结果：



如果500.html和5xx.html同时存在的话，可以会根据状态码精确匹配，如果没有相关状态码的html，则还是执行5xx.html



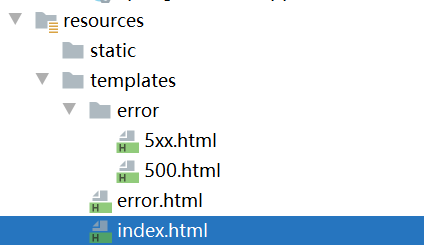
500.html和5xx.html同时存在，则会根据状态码精确匹配500.html。

4. 统一错误页面显示

在templates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。

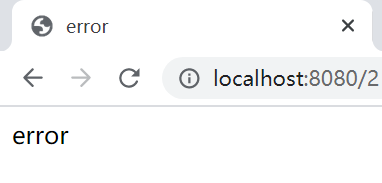
比如我们在error下没有设置4xx相关的错误页面，当发生状态码为4xx的错误时，找不到相关的错误处理页面，这时会走一个统一的错误处理页面。一般会在templates下创建error.html，作为统一的错误处理页面。

在templates下创建error.html



|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>error</title> </head> <body>  error </body> </html> |

当发生404错误时，会自动走error.html



## 5.2异常处理

在Spring Boot项目中除了设置错误页面，还可以通过注解实现错误处理。

### 5.2.1局部异常

在Spring Boot项目中除了设置错误页面，还可以通过注解实现错误处理。

局部异常：

在控制器类中添加一个方法，结合@ExceptionHandler。但是只能对当前控制器中方法出现异常进行解决。

1. 创建异常信息类

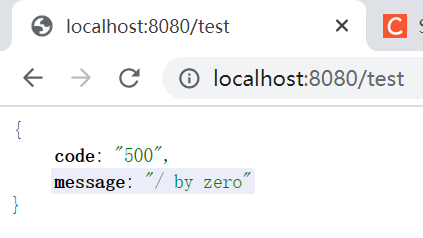
|  |
| --- |
| @Data public class ExceptionMessage {   private String code;  private String message; } |

1. 在controller中设置异常处理

|  |
| --- |
| @Controller public class ErrorController {   @RequestMapping("/test")  public String testError(){  System.*out*.println(5/0);  return "index";  }   @ExceptionHandler(ArithmeticException.class)  @ResponseBody  public ExceptionMessage arithmeticException(Exception e){  ExceptionMessage exceptionMessage = new ExceptionMessage();  exceptionMessage.setCode("500");  exceptionMessage.setMessage(e.getMessage());  return exceptionMessage;  } } |

@ExceptionHandler的参数为发生异常的类型。如果controller的方法中捕获到了这种异常，就会走@ExceptionHandler表示的方法arithmeticException(),在方法参数中，可以获取异常对象。

最终执行结果：



当访问test的controller方法时，会出现除0异常，就会走异常处理方法，封装异常信息，返回。

### 5.2.2全局异常

新建全局异常类，通过@ControllerAdvice结合@ExceptionHandler。当全局异常处理和局部处理同时存在时，局部生效（就近原则）

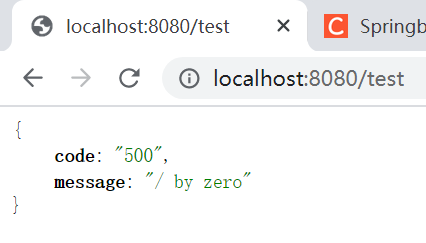
1. 编写异常处理controller

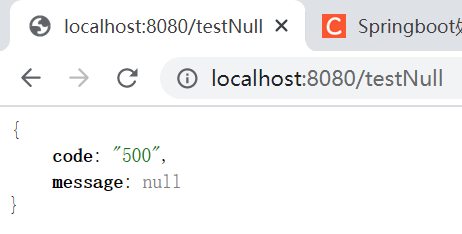
|  |
| --- |
| @ControllerAdvice public class GlobalExceptionHandleController {   *//发生除0异常时，会执行该方法* @ExceptionHandler(ArithmeticException.class)  @ResponseBody  public ExceptionMessage arithmeticException(Exception e){  ExceptionMessage exceptionMessage = new ExceptionMessage();  exceptionMessage.setCode("500");  exceptionMessage.setMessage(e.getMessage());  return exceptionMessage;  }   *//发生空指针异常时会执行该方法* @ExceptionHandler(NullPointerException.class)  @ResponseBody  public ExceptionMessage nullPointerException(NullPointerException  e){  ExceptionMessage exceptionMessage = new ExceptionMessage();  exceptionMessage.setCode("500");  exceptionMessage.setMessage(e.getMessage());  return exceptionMessage;  }   *//发生其他未知异常时，会执行该方法* @ExceptionHandler(Exception.class)  @ResponseBody  public ExceptionMessage otherException(){  ExceptionMessage exceptionMessage = new ExceptionMessage();  exceptionMessage.setCode("500");  exceptionMessage.setMessage("发生了其他未知异常！");  return exceptionMessage;  }  } |

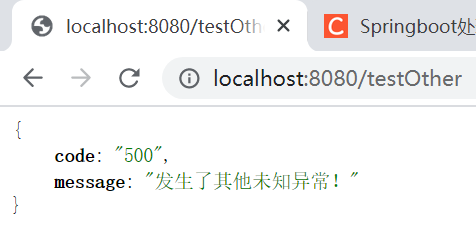
1. 编写controller

|  |
| --- |
| @Controller public class ErrorController {   @RequestMapping("/test")  public String testError(){  System.*out*.println(5/0);  return "index";  }  @RequestMapping("/testNull")  public String testNull(){  String s = null;  System.*out*.println(s.length());  return null;  }   @RequestMapping("testOther")  public String testOther(){  int[] arr = new int[3];  System.*out*.println(arr[5]);  return null;  } } |

1. 测试结果







# 6SpringBoot定时任务

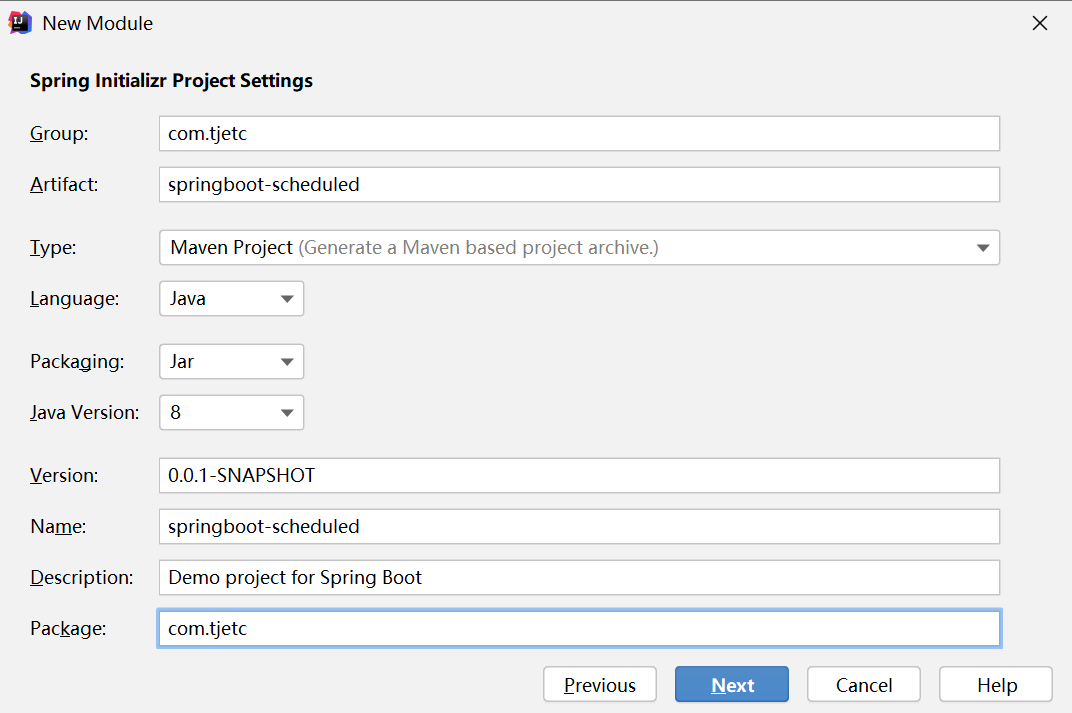
## 6.1 Scheduled简介

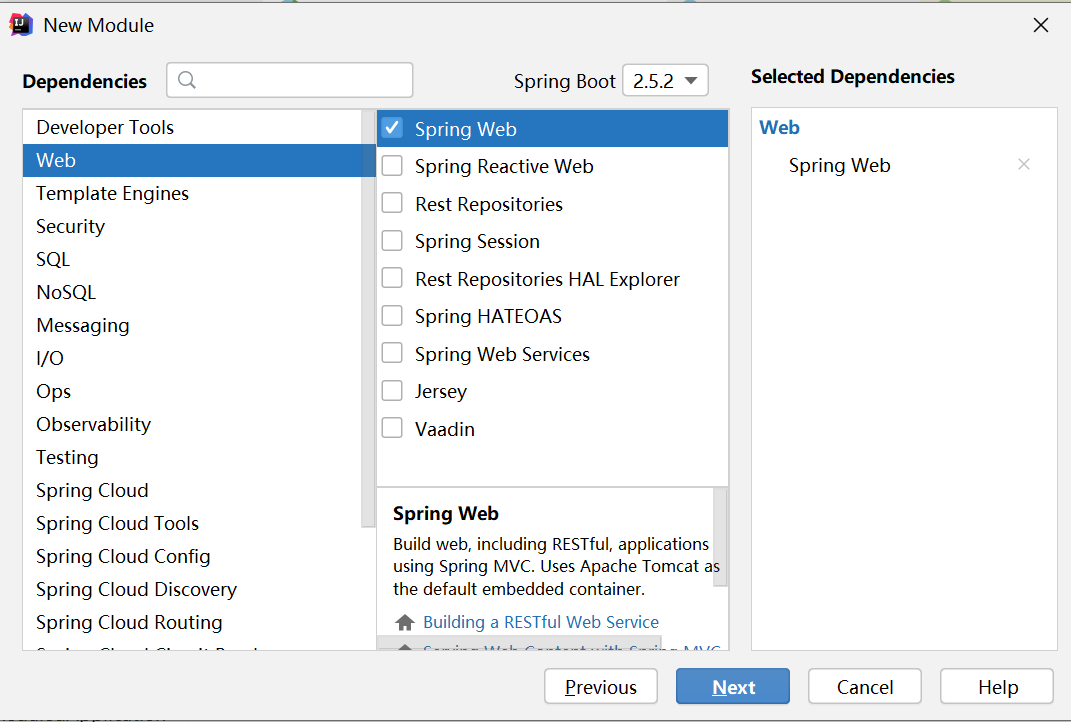
Scheduled是Spring3.0后内置的定时任务器。通过Scheduled可以完成周期的执行一些功能。存在于spring-conext-support.jar中。

在SpringBoot中使用Scheduled非常简单，只需要在对应的方法上添加@Scheduled注解，再配置对应的参数就可以完成。

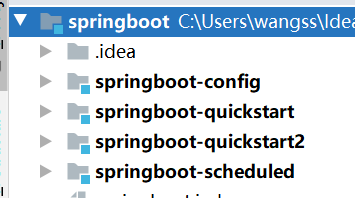
### 6.1.1创建工程







next，finish。



### 6.1.2添加依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId> </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId> </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope> </dependency> |

### 6.1.3在启动类上添加注解

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

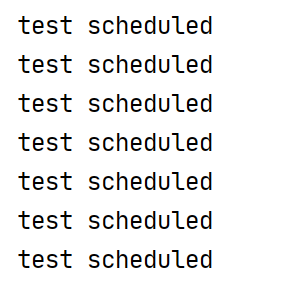
在启动类上添加@EnableScheduling注解

### 6.1.4创建定时任务

|  |
| --- |
| import org.springframework.scheduling.annotation.Scheduled; import org.springframework.stereotype.Component;  @Component public class DemoScheduled {  *//调度注解 cron填写表达式 每个参数之间有空格*  @Scheduled(cron = "0/2 \* \* \* \* \*")  public void testScheduled(){  System.*out*.println("test scheduled");  } } |

### 6.1.5启动服务

启动spring服务，运行结果如下，每隔两秒执行一次任务。



## 6.2Cron表达式

Cron表达式是一个字符串，分为6或7个域，每一个域代表一个含义

Cron有如下两种语法格式：

* Seconds Minutes Hours Day Month Week Year
* Seconds Minutes Hours Day Month Week Year

corn从左到右（用空格隔开）：

秒 分 小时 月份中的日期 月份 星期中的日期 年份

|  |  |  |  |
| --- | --- | --- | --- |
| 位置 | 时间域名 | 允许值 | 允许的特殊字符 |
| 1 | 秒 | 0-59 | , - \* / |
| 2 | 分钟 | 0-59 | , - \* / |
| 3 | 小时 | 0-23 | , - \* / |
| 4 | 日 | 1-31 | , - \* / L W C |
| 5 | 月 | 1-12 | , - \* / |
| 6 | 星期 | 1-7（周日到周六） | , - \* ? / L C # |
| 7 | 年(可选) | 1970-2099 | , - \* / |

Cron表达式的时间字段除允许设置数值外，还可使用一些特殊的字符，提供列表、范围、通配符等功能，细说如下：

* 星号(\*)：可用在所有字段中，表示对应时间域的每一个时刻，例如，\*在分钟字段时，表示“每分钟”；
* 问号（?）：该字符只在星期字段中使用，它通常指定为“无意义的值”，相当于占位符；
* 减号(-)：表达一个范围，如在小时字段中使用“10-12”，则表示从10到12点，即10,11,12；
* 逗号(,)：表达一个列表值，如在星期字段中使用“MON,WED,FRI”，则表示星期一，星期三和星期五；
* 斜杠(/)：x/y表达一个等步长序列，x为起始值，y为增量步长值。如在秒字段中使用0/15，则表示为0,15,30和45秒，而5/15在分钟字段中表示5,20,35,50，你也可以使用\*/y，它等同于0/y；
* L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中意思不同。L在日期字段中，表示这个月份的最后一天，如一月的31号，非闰年二月的28号；如果L用在星期中，则表示星期六，等同于7。但是，如果L出现在星期字段里，而且在前面有一个数值X，则表示“这个月的最后X天”，例如，6L表示该月的最后星期五；
* W：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如15W表示离该月15号最近的工作日，如果该月15号是星期六，则匹配14号星期五；如果15日是星期日，则匹配16号星期一；如果15号是星期二，那结果就是15号星期二。但必须注意关联的匹配日期不能够跨月，如你指定1W，如果1号是星期六，结果匹配的是3号星期一，而非上个月最后的那天。W字符串只能指定单一日期，而不能指定日期范围
* LW组合：在日期字段可以组合使用LW，它的意思是当月的最后一个工作日；
* 井号(#)：该字符只能在星期字段中使用，表示当月某个工作日。如6#3表示当月的第三个星期五(6表示星期五，#3表示当前的第三个)，而4#5表示当月的第五个星期三，假设当月没有第五个星期三，忽略不触发；
* C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中所有日期。例如5C在日期字段中就相当于日历5日以后的第一天。1C在星期字段中相当于星期日后的第一天。

Cron表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。

例子:

* + @Scheduled(cron = "0 0 1 1 1 ?")//每年一月的一号的1:00:00 执行一次
  + @Scheduled(cron = "0 0 1 1 1,6 ?") //一月和六月的一号的1:00:00 执行一次
  + @Scheduled(cron = "0 0 1  1 1,4,7,10 ?") //每个季度的第一个月的一号的1:00:00 执行一次
  + @Scheduled(cron = “0 0 0 11 11 ?”) 11月11日0点执行

Cron表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。

# 7SpringBoot与web开发

## 7.1静态资源映射规则

“/\*\*” 访问当前项目任何资源，全部找静态资源的文件夹进行映射

静态资源的文件夹:

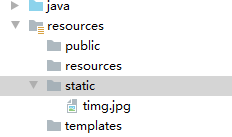
"classpath:/META-INF/resources/",

"classpath:/resources/",

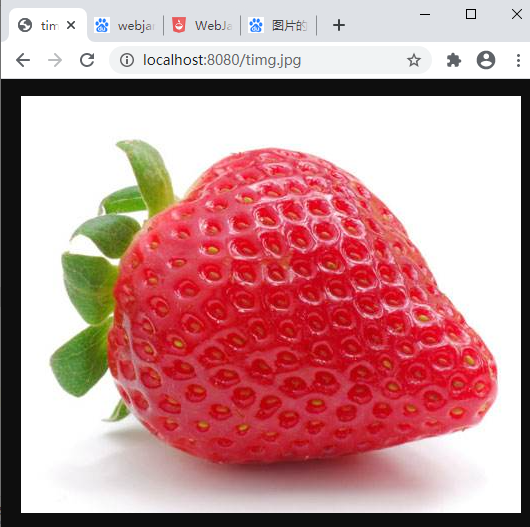
"classpath:/static/",

"classpath:/public/"

静态资源路径下的文件，可以通过地址栏直接访问。在static，public，resources或者META-INF/resources/下放图片timg.jpg

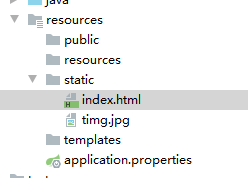


可以直接在地址栏中访问：



3.“/\*\*” 访问静态资源文件夹下的所有index.html页面

在static下创建index.html



|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>index</title> </head> <body>  <h1>INDEX</h1>  <img src="timg.jpg" width="100px"> </body> </html> |

通过地址栏直接访问index.html

|  |
| --- |
|  |

如果index.html的位置在/static/aaa/index.html，则相应的访问路径也要为/aaa/index.html

Img标签的src属性为”../timg.jpg”

|  |
| --- |
|  |

## 7.2自定义静态资源映射规则

自定义配置类

|  |
| --- |
| import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  */\*\*  \* 资源映射路径  \*/* @Configuration public class MyWebAppConfigurer implements WebMvcConfigurer {  *//添加自定义资源映射路径* @Override  public void addResourceHandlers(ResourceHandlerRegistry registry) {  *//addResourceHandler 添加资源处理url路径  //addResourceLocations 添加url对应的磁盘物理路径* registry.addResourceHandler("/\*\*").  addResourceLocations("classpath:/META-INF/resources/",  "classpath:/resources/",  "classpath:/static/",  "classpath:/public/",  "file:D:/image/");  } } |

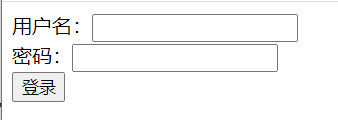
## 7.3登录拦截器

### 7.3.1编写controller

|  |
| --- |
| @Controller @RequestMapping("user") public class UserController {  */\*\*  \* 登录验证  \*  \* @param username  \* @param password  \* @param session  \* @param model  \* @return  \*/* @RequestMapping("login")  public String login(@RequestParam("username") String username,  @RequestParam("password") String password,  HttpSession session, Model model) {  *//模拟数据库查询* if (username.equals("zhangsan") && password.equals("111")) {  session.setAttribute("username", "zhangsan");  return "redirect:/user/welcome";  } else {  model.addAttribute("msg", "用户名或者密码错误");  return "login";  }  }  */\*\*  \* 登录的页面  \*  \* @param model  \* @return  \*/* @RequestMapping("login-html")  public String loginHtml(Model model) {  model.addAttribute("msg", "");  return "login";  }  */\*\*  \* 欢迎首页  \*  \* @return  \*/* @RequestMapping("welcome")  public String welcome() {  return "welcome";  } } |

登录页面

|  |
| --- |
| <!DOCTYPE html> <html lang="en" xmlns:th="http://www.thymeleaf.org"> <head>  <meta charset="UTF-8">  <title>Title</title> </head> <body>  <form action="/user/login" method="post">  用户名：<input type="text" name="username"><br>  密码：<input type="password" name="password"><br>  <input type="submit" value="登录"><br>  <span th:if="${msg}!=''" th:text="${msg}"></span>  </form> </body> </html> |



### 7.3.2编写登录拦截器

|  |
| --- |
| public class LoginInterceptor implements HandlerInterceptor {  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  *//获取session* HttpSession session = request.getSession();  *//判断session总是否存在用户名* String username = (String) session.getAttribute("username");  if (username == null) {*//没有登录或者登录过期跳转登录页面* request.setAttribute("msg", "");  *//重定向到请求/user/login-html* response.sendRedirect("/user/login-html");  *//返回false，不执行后面的拦截器和Controller对应的方法* return false;  }  *//放行* return true;  } } |

### 7.3.3配置拦截器

|  |
| --- |
| @Configuration public class LoginInterceptorConfig implements WebMvcConfigurer {  */\*\*  \* 注册登录拦截器  \*  \* @param registry  \*/* @Override  public void addInterceptors(InterceptorRegistry registry) {  *//注册拦截器* registry.addInterceptor(new LoginInterceptor())  *//拦截器要拦截url* .addPathPatterns("/\*\*")  *//排除拦截器要拦截的url* .excludePathPatterns("/user/login-html", "/user/login", "/css/\*\*", "/js/\*\*", "/img/\*\*");  } } |