# SpringBoot

# 1SpringBoot入门

## 1.1了解为什么要学习Spring Boot

Spring Boot是一个快速的开发框架,能够快速的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。

spring boot特性:

1.帮助开发者快速整合第三方框架(原理maven依赖特性)

2.完全不需要第三方服务器即可运行, 内置了第三方容器(tomcat/jetty/undertom) (原理:tomcat容器使用java开发的)

3.使用注解的方式来简化xml书写

## 1.2清楚课程怎么学

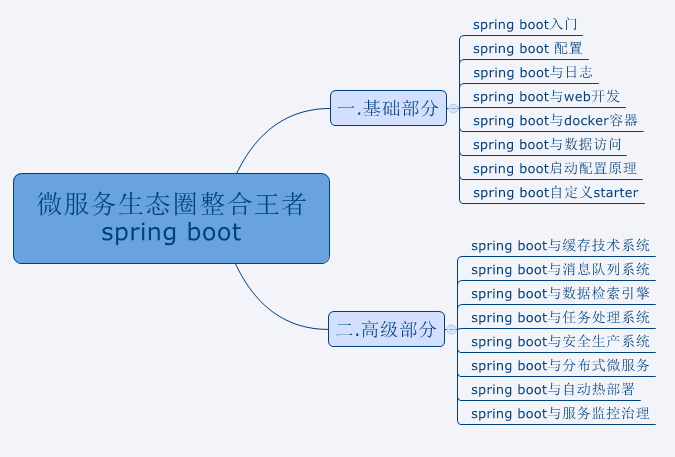
一.基础知识积累

spring框架知识（特别注解方式使用）

熟练使用maven进行项目构建和依赖管理

熟练使用开发工具（eclipse、idea）

二.Spring Boot知识体系



三.统一开发环境

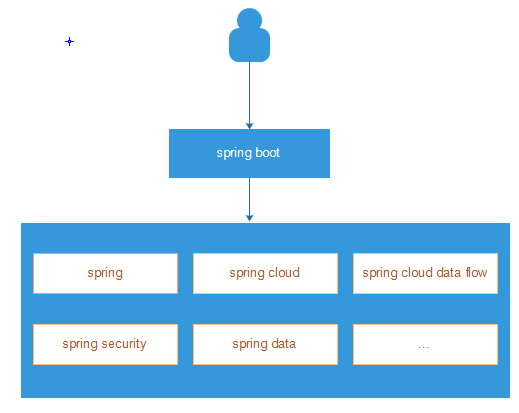
JDK1.8

Maven版本3.x

Idea2018.2.2

## 1.3了解J2EE技术栈与springboot优缺点

J2EE笨重的开发技术，繁重的配置，低下的开发效率，复杂的流程处理，对第三方框架整合困难。Spring Boot 用于简化Spring的应用开发的，约定大于配置，简化spring技术栈的开发。



Spring Boot优点：

Create stand-alone Spring applications

//创建运行的spring项目

Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)

//兼容web容器，无需打成war包部署

Provide opinionated 'starter' dependencies to simplify your build configuration

//使用starter进行简单的配置就可以使用spring全套框架

Automatically configure Spring and 3rd party libraries whenever possible

//自动配置spring以及第三方框架，并且提供自定义配置功能

Provide production-ready features such as metrics, health checks and externalized configuration

//提供生产环境健康功能，比如运行状态的检查和外部配置

Absolutely no code generation and no requirement for XML configuration

/ /不需要配置xml，无须生成代码，开箱即用。

Spring Boot缺点：

由于Spring Boot封装了大量框架，入门比较容易，精通比较困难，要对封装的第三方框架底层API有深入了解，才能定制高级的服务。

## 1.4Springboot快速入门

### 1.4.1创建maven工程

### 1.4.2导入依赖

|  |
| --- |
| *<!-- 导入springboot的版本及其框架依赖 -->* <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.2.8.RELEASE</version> </parent>  *<!-- 导入动态web工程的依赖 -->* <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> </dependencies>  *<!-- 项目的打包工具，会把当前工程打成一个可执行的jar包-->* <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins> </build> |

### 1.4.3创建springboot启动类

创建包com.icss.chapter01

创建启动类

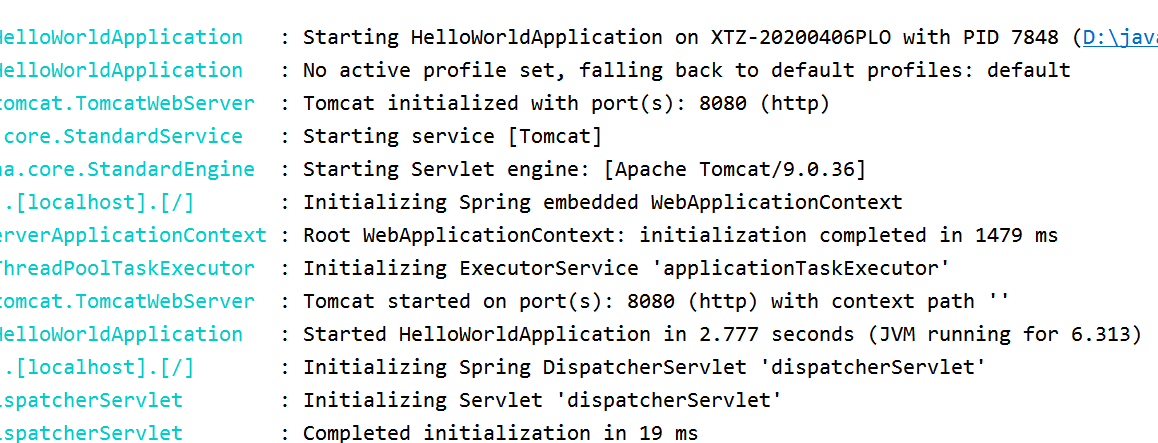
|  |
| --- |
| package com.icss.chapter01;   import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication public class HelloWorldApplication {  public static void main(String[] args) {  SpringApplication.*run*(HelloWorldApplication.class,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

### 1.4.4创建controller

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

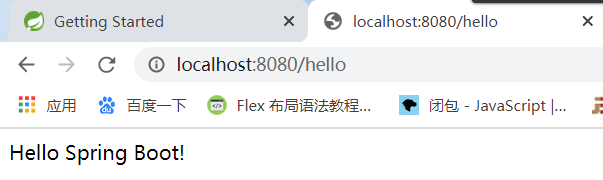
### 1.4.5启动springBoot

运行启动类的main方法

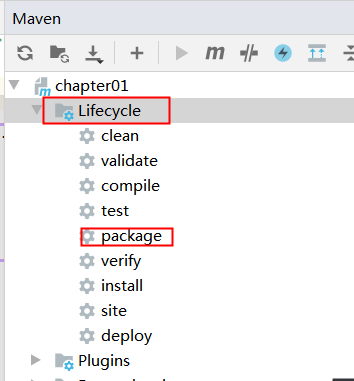


启动成功，端口号8080

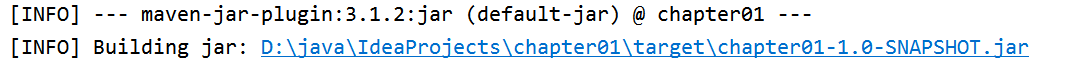
### 1.4.6访问controller



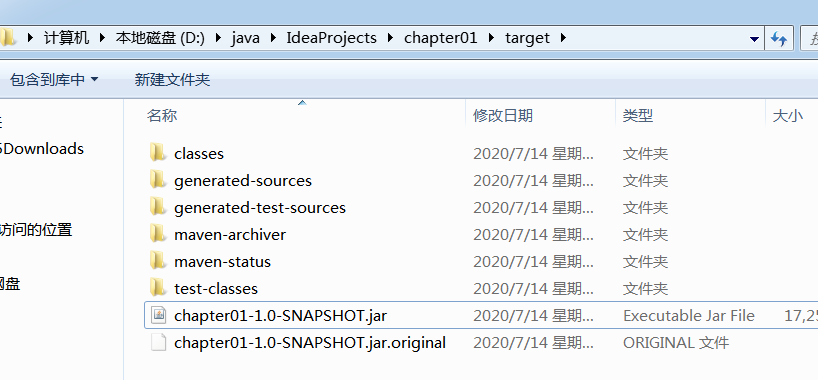
### 1.4.7打包运行



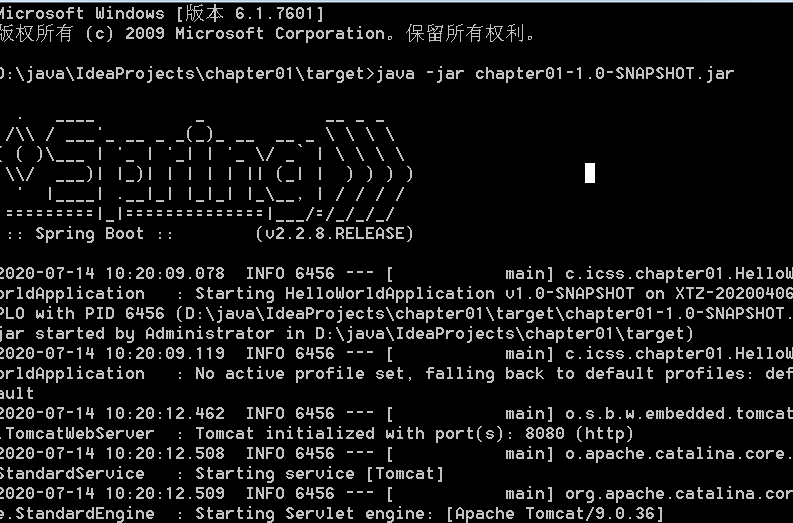
将当前的工程打成jar包



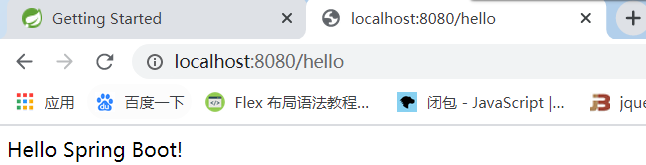
放在target下面。



在当前的路径下打开cmd窗口。运行jar包：

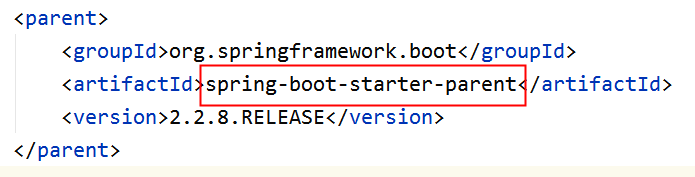


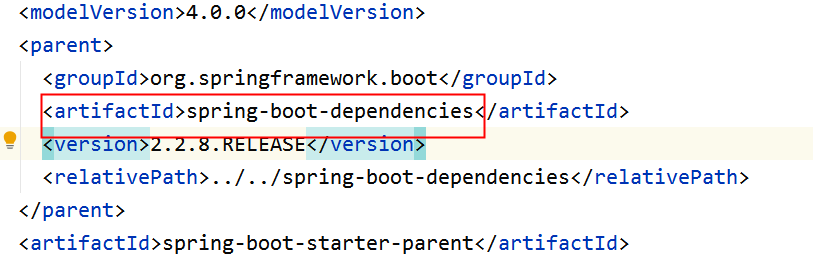
访问controller



## 1.5理解Pom文件的依赖与starter启动器的作用

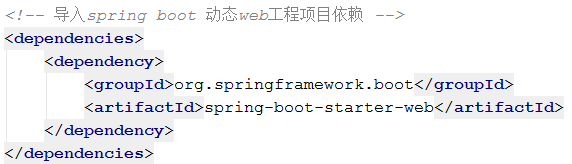
Pom文件中的parent是Spring Boot的框架版本控制中心







starter：spring-boot场景启动器



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

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。

## 1.6理解主程序@SpringBootApplication

@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: 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。

@AutoConfigurationPackage：自动配置包

@Import(AutoConfigurationPackages.Registrar.class)：

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

将主配置类（标注@SpringBootApplication注解的类）的所在目录的包及下面所有子包里面的所有组件扫描到Spring容器。

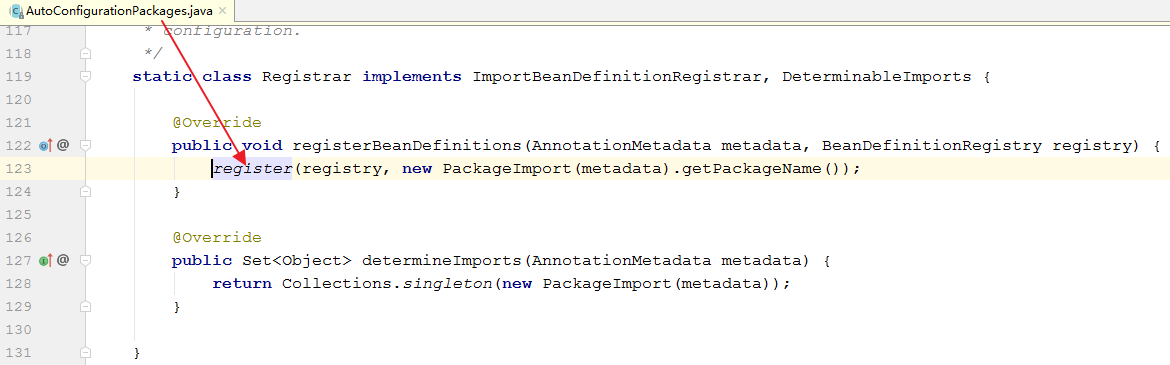
@Import(EnableAutoConfigurationImportSelector.class) 给容器中导入组件。

EnableAutoConfigurationImportSelector：组件的选择器。

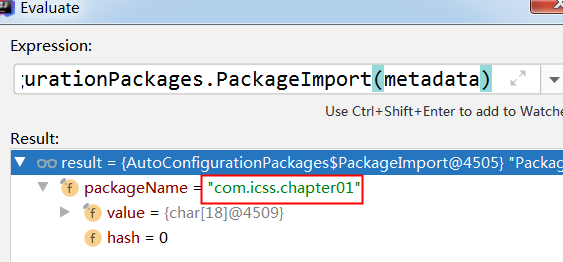
将所有需要导入的组件以全类名的方式返回，这些组件就会被添加到容器中。

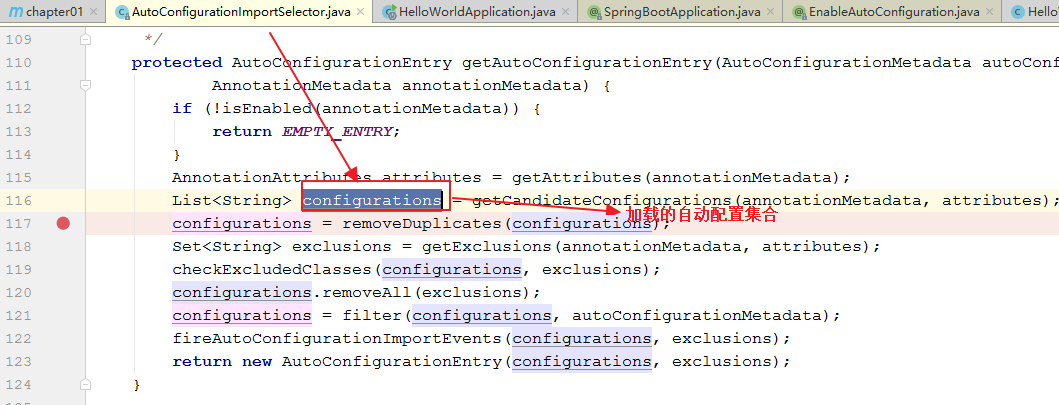
组件的选择器给容器中导入非常多的自动配置类（xxxAutoConfiguration），给容器中导入这个场景需要的所有组件，并配置好这些组件。

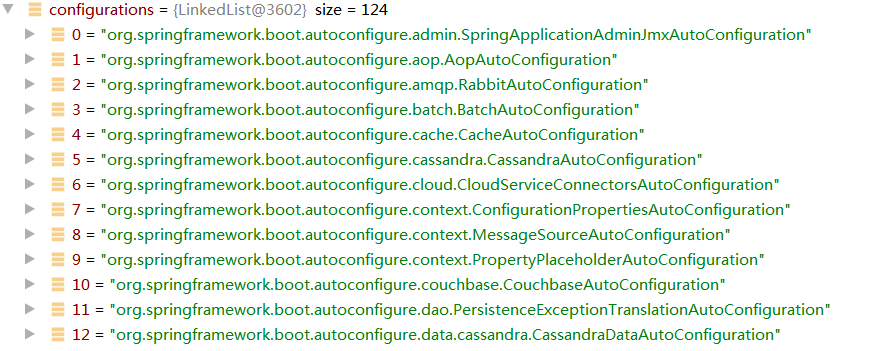
## 1.7理解\*AutoConfiguration自动配置精髓



通过查看包的名称 :启动类所在的包，所以其他所要执行的内容，需要放在启动类所在的包下或者子包下。







总结提问：

为什么要使用spring boot?

Spring boot 有哪些优点？

Pom文件中导入的parent有什么作用？

@SpringBootApplication 有什么作用？

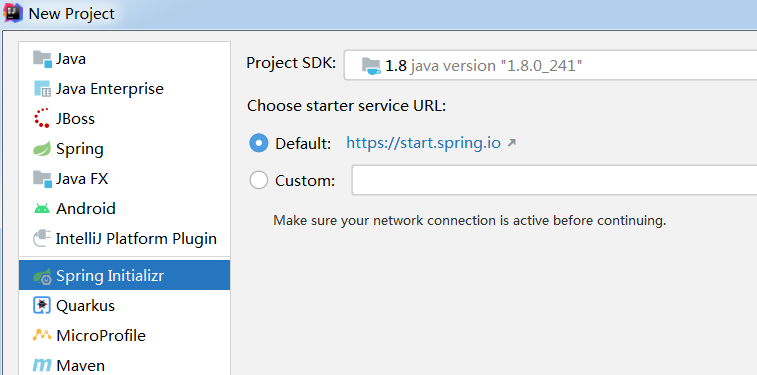
@xxxAutoConfiguration有什么作用？

AutoConfigurationImportSelector 有什么作用？

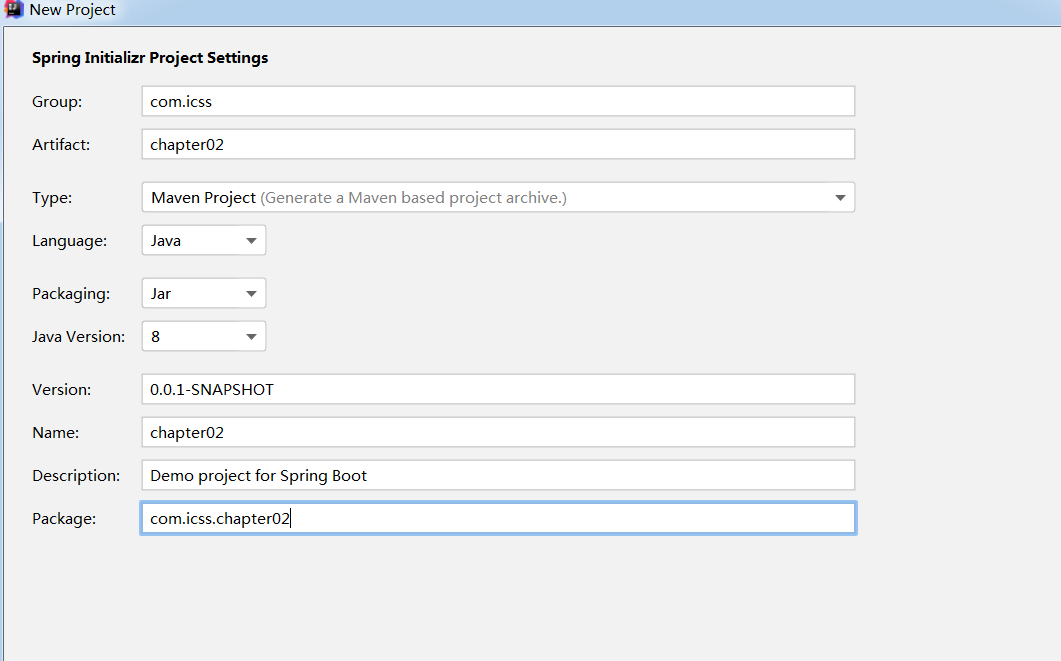
# 2SpringBoot配置

## 2.1了解引导器快速创建Spring Boot应用程序

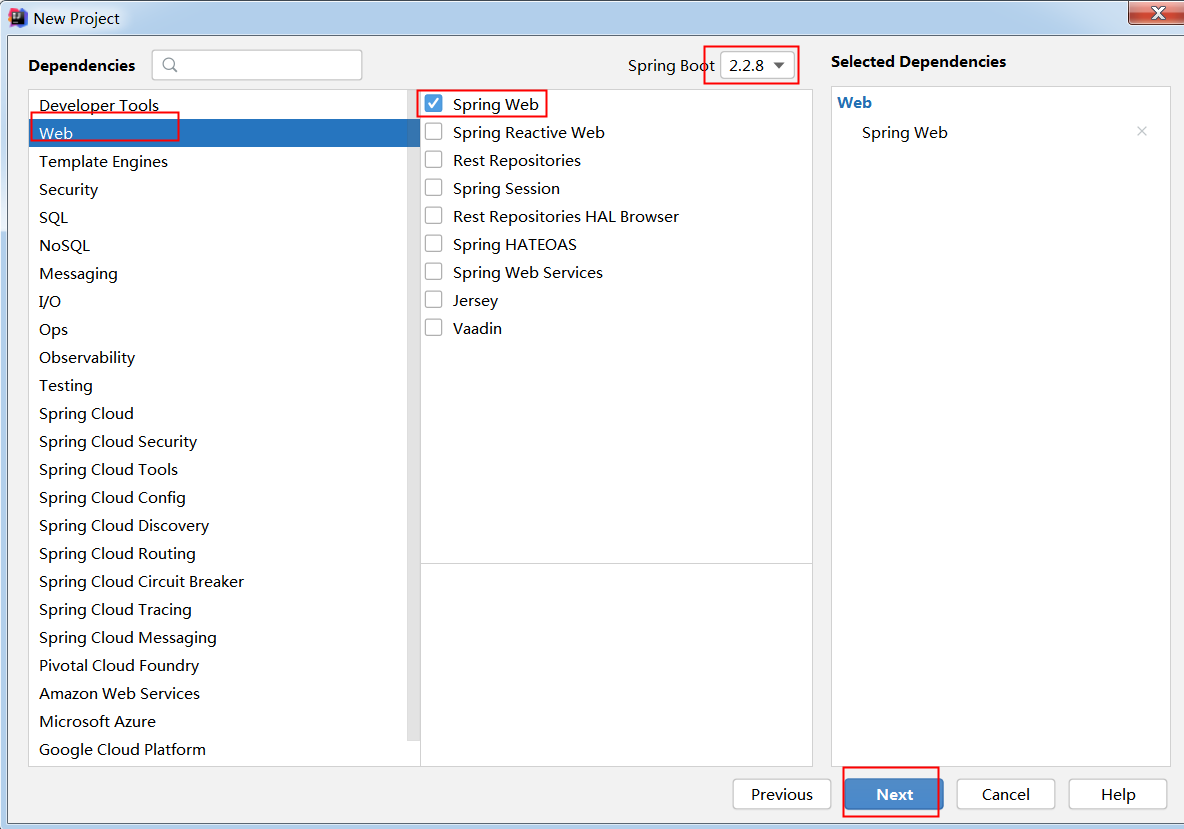
创建project



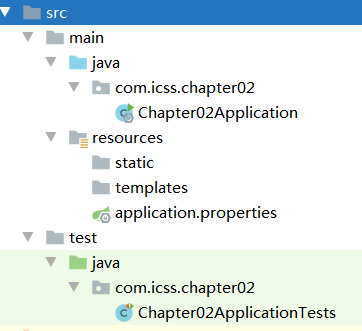
填写group和Artifact



选择依赖：创建web工程，只要选择web依赖即可，选择springboot版本



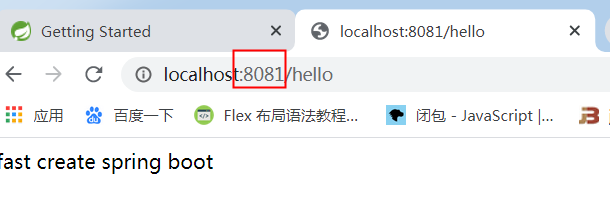
创建工程以后，查看工程的目录结构：



自动创建了测试类，和启动类，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

指定端口号：在application.properties中添加如下代码：

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



pom.xml中导入了parent，web依赖和单元测试，以及打包工具。

## 2.2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

### 2.2.1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

### 2.2.2yml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，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] |

## 2.3Spring Boot获取配置文件的值及配置文件编码设置

### 2.3.1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* private Date 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 Date getBirth() {  return birth;  }   public void setBirth(Date 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 +  '}';  } } |

### 2.3.2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" |

### 2.3.3编写测试类

|  |
| --- |
| *//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);  }  } |

### 2.3.4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.

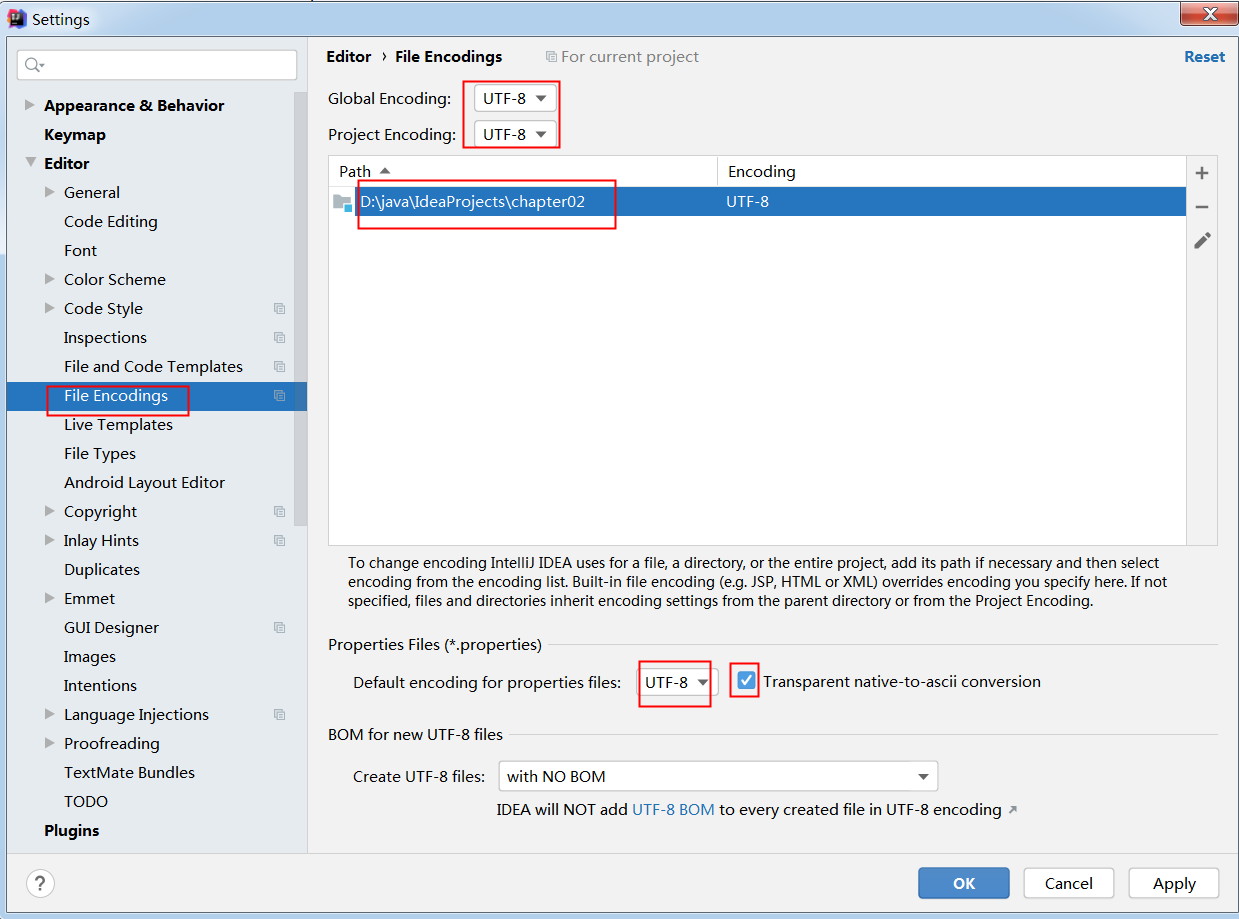
在application.properties中配置student

|  |
| --- |
| *#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

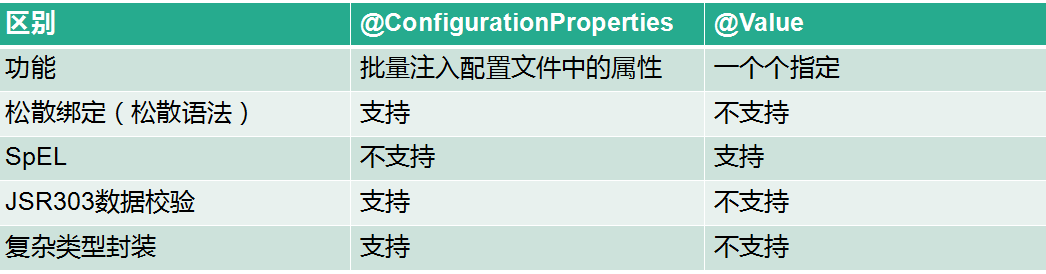


都改好以后，点击OK。

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"'}} |

## 2.4@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：大写用-

student. student\_name：大写用\_

STUDENT\_STUDENT\_NAME：推荐系统属性使用这种写法

### 2.4.1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* |

### 2.4.2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

### 2.4.3SpEL

@ConfigurationProperties不支持SpEL，比如：

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

@Value支持SpEL。比如：

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

Age是可以正常注入的。

### 2.4.4数据校验

@ConfigurationProperties支持数据校验：

|  |
| --- |
| @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* @Validated*//给类加上校验* public class Student {  *//@Value：注意一个容器已经存在的对象 @Value("${student.student-name}") // @Value("${student.student-name}")* @Email*//studentName必须为邮箱格式才可以注入* private String studentName; *// @Value("#{18+8}")* private Integer age; *// @Value("true")* private Boolean sex;*//男代表true* private Date birth;  private Map<String, Object> maps;  private List<Object> lists;  private Book book;  } |

校验结果

|  |
| --- |
| Property: student.studentName  Value: jerry  Origin: class path resource [application.properties]:5:22  Reason: 不是一个合法的电子邮件地址 |

而@Value不支持校验。即使注入不符合要求的studentname，也是可以成功的。

|  |
| --- |
| @Validated*//给类加上校验* public class Student {  *//@Value：注意一个容器已经存在的对象 @Value("${student.student-name}")* @Value("${student.student-name}")  @Email*//studentName必须为邮箱格式才可以注入* private String studentName;  } |

这时，studentName是可以正常注入的，不检查是不是邮箱类型

### 2.4.5复杂类型封装

@ConfigurationProperties支持复杂类型封装

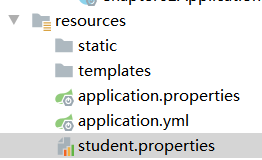
@Value不支持复杂类型，只支持字符串，和基本数据类型及其包装类。

## 2.5 @PropertySource @ImportResource @Bean@Configuration@Import和占位符使用

### 2.5.1@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 {} |

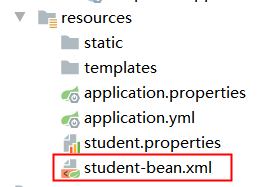
### 2.5.2@ImportResource

导入Spring的配置文件，让配置文件里面的内容生效

创建一个StudentService

|  |
| --- |
| *//按照道理来说，需要给一个@Service注解，才能将studentservice放入容器 //但是现在不给，创建spring的配置文件，在配置文件中，通过bean标签也可以放在容器中* public class StudentService {   public void dosomething(){  System.*out*.println("do something........");  } } |

创建一个spring的配置文件，定义studentservice



|  |
| --- |
| *<?*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="studentService" class="com.icss.chapter02.service.StudentService"></bean> </beans> |

读取spring的配置文件。在启动类上加注解：

|  |
| --- |
| *//导入配置文件，读取加载配置文件,给配置文件的路径* @ImportResource("classpath:student-bean.xml") @SpringBootApplication public class Chapter02Application {   public static void main(String[] args) {  SpringApplication.*run*(Chapter02Application.class, args);  }  } |

测试：

|  |
| --- |
| @Autowired private StudentService studentService;  @Test public void test1(){  System.*out*.println(studentService);  studentService.dosomething(); } |

结果：

|  |
| --- |
| com.icss.chapter02.service.StudentService@37303f12  do something........  这种方式还是需要编写xml配置，所以跟springboot的优点不太符合。在springboot中很少使用xml的配置，可以使用java类的配置来替代。 |

### [2.5.3@Bean、@Configuration](mailto:2.5.3@Bean、@Configuration)

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studentservice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 StudentService studentService(){  return new StudentService();  }  } |

测试：

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

结果：

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

### [2.5.4@Import](mailto:2.5.4@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：

|  |
| --- |
|  |

### 2.5.5占位符

RandomValuePropertySource（配置文件中注入随机值）与占位符

属性配置占位符:

entry.attr=myvalue

${entry.attr:defaultValue}

可以在配置文件中引用前面配置过的属性。

${entry.attr:defaultValue}来指定找不到属性时的默认值

|  |
| --- |
| student.lists=lisi,wangwu *#可以通过${}占位符使用前面已经定义好的变量，${student.student-name} #如果没有值，可以给默认值* student.book.bookName=${student.student-name:tom}中软国际java典型案例 student.book.author=中软卓越研究院 |

还可以给随机数：

|  |
| --- |
| *#配置student* student.student-name=jerry student.age=${random.int} student.sex=true student.birth=1998/10/08 student.maps.k1=v1 student.maps.k2=v2 student.lists=lisi,wangwu *#可以通过${}占位符使用前面已经定义好的变量，${student.student-name} #如果没有值，可以给默认值* student.book.bookName=${student.student-name:tom}中软国际java典型案例 student.book.author=中软卓越研究院 |

执行结果：

|  |
| --- |
| Student{studentName='jerry', age=-1169695155, sex=true, birth=Thu Oct 08 00:00:00 CST 1998, maps={k2=v2, k1=v1}, lists=[lisi, wangwu], book=Book{bookName='jerry中软国际java典型案例', author='中软卓越研究院'}} |

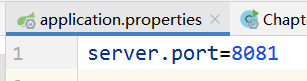
## 2.6多文件配置,加载顺序与位置

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!

### 2.6.1.多文件的形式

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

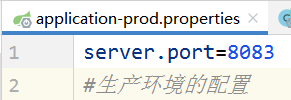
Application.properties:



Application-dev.properties:



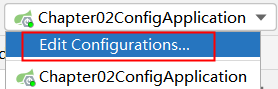
Application-prod.properties:

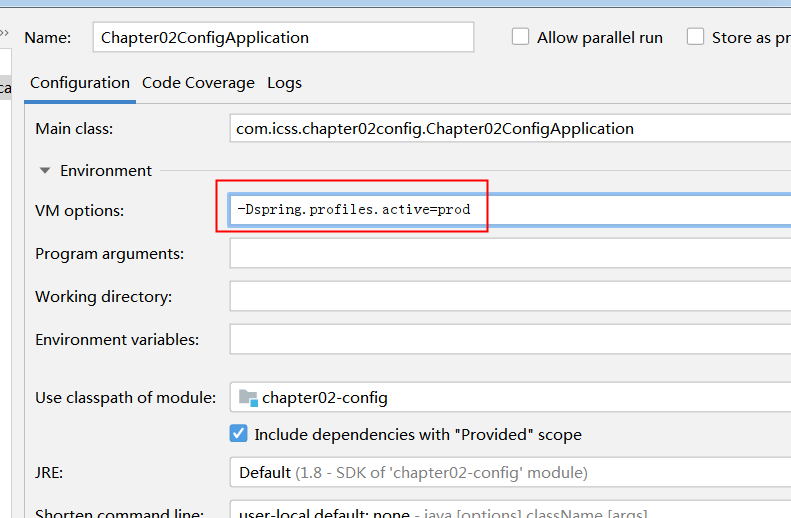


Spring boot提供几种的激活配置方式

### 2.6.2Spring boot提供几种的激活配置方式

一．JVM参数 -Dspring.profiles.active=dev



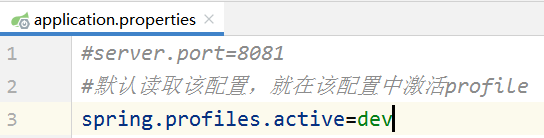


确定，启动springboot程序即可。

这时激活的是prod，端口号8083

1. 配置文件 spring.profiles.active=prod

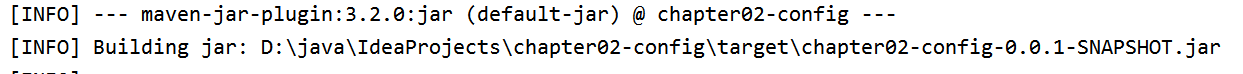
在全局配置文件application.properties中激活相应的profile即可



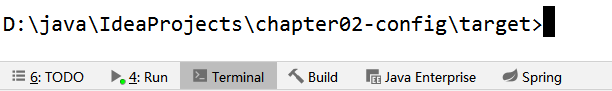
此时，dev被激活，端口号为8082

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

### 2.6.3yml文档块的模式

Yml可以实现与properties同样的功能，只需定义配置文件

Application.yml, application-dev.yml, application-prod.yml

并在全局配置以文件中激活profile即可：

|  |
| --- |
| *#激活profile==prod* spring:  profiles:  active: prod |

如果想减少配置文件的数量，也可以使用文档块：全局配置文件如下：

|  |
| --- |
| server:  port: 8081  *#激活profile==prod* spring:  profiles:  active: prod --- server:  port: 8082 spring:  profiles: dev --- server:  port: 8083 spring:  profiles: prod |

### 2.6.4加载顺序

SpringApplication将从以下位置加载application.properties文件， 并把它们添加到Spring环境上下文中：

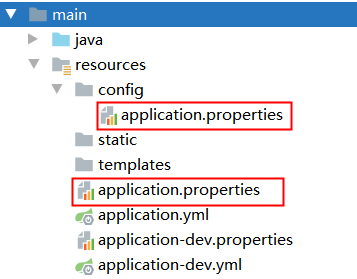
当前目录下的/config子目录

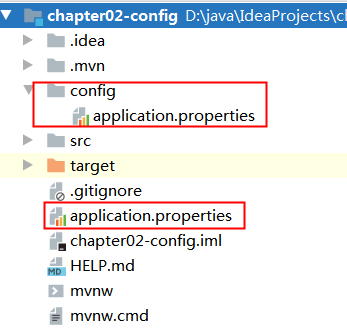
当前目录

classpath下的/config包

classpath根路径（项目root）

这个列表是按优先级排序的（列表中位置高的将覆盖位置低的），同时也可以使用YAML文件替代properties文件!





优先级：当前目录下的/config子目录>当前目录>classpath下的/config包>classpath根路径

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

Spring boot加载规则:优先级从高到低，高优先级的配置覆盖低优先级的配置，不同的配置混合一起使用。所有的配置由jar包外向jar内查找，优先加载带profile的，再加载不带profile的。

命令行参数

来自于java:comp/env的JNDI属性

Java系统属性（System.getProperties()）

操作系统环境变量

只有在random.\*里包含的属性会产生一个RandomValuePropertySource

在打包的jar外的应用程序配置文件（application.properties， 包含YAML和profile变量）

在打包的jar内的应用程序配置文件（application.properties， 包含YAML和profile变量）

在@Configuration类上的@PropertySource注解

默认属性（使用SpringApplication.setDefaultProperties指定）

## 2.7自动配置原理精髓

可以查看DataSourceAutoConfiguration

通用模式

xxxAutoConfiguration：自动配置类

xxxProperties：属性配置类

yml/properties文件中能配置的值就来源于xxxProperties

精髓：

SpringBoot启动会加载大量的自动配置类

先需要看整合组件有没有SpringBoot默认写好的自动配置类；

再看自动配置类中配置了哪些组件；（要是有我们要用的组件，我们就不需要配置了）

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；

## 2.8@Conditional和自动配置报告

@Conditional(Spring的注解):当指定的条件成立，返回为true,代表组件可以使用! 否则,组件不可以使用!

--debug=true查看详细的自动配置报告

Positive matches(正匹配): 代表可以使用的

Negative matches(负匹配): 代表不可以使用的

# 3SpringBoot与日志

## 3.1日志框架设计思想

市面上存在的大量日志框架:

JUL、JCL、Jboss-logging、logback、log4j、log4j2、slf4j......

日志框架分为两类：

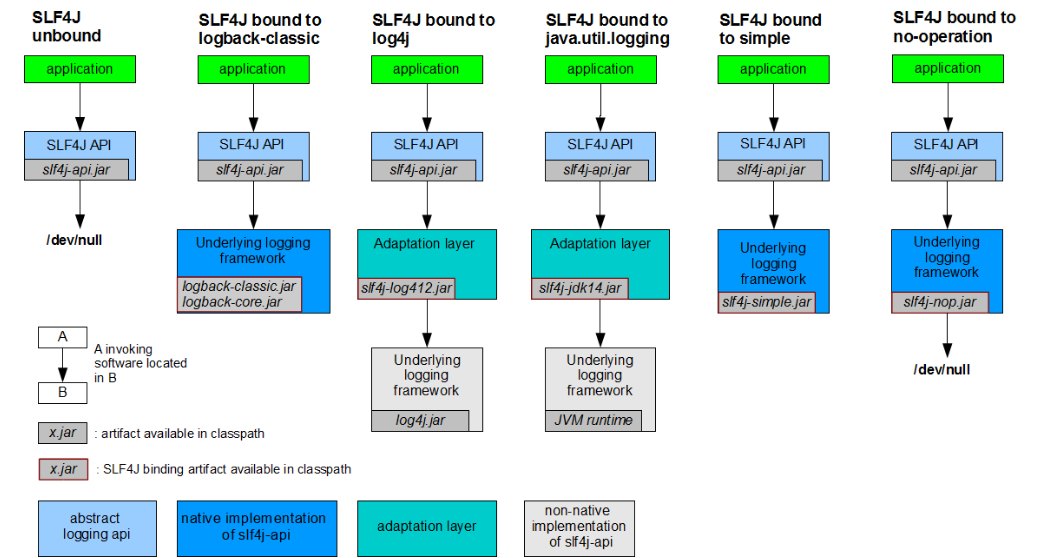
1、抽象类日志：

JCL、 slf4j、 Jboss-logging……

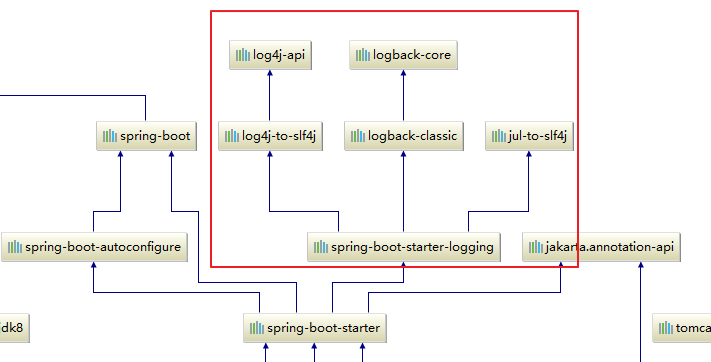
2、实现类日志

log4j、JUL、log4j2、logback……

Spring Boot使用的是SLF4j抽象和logback实现。

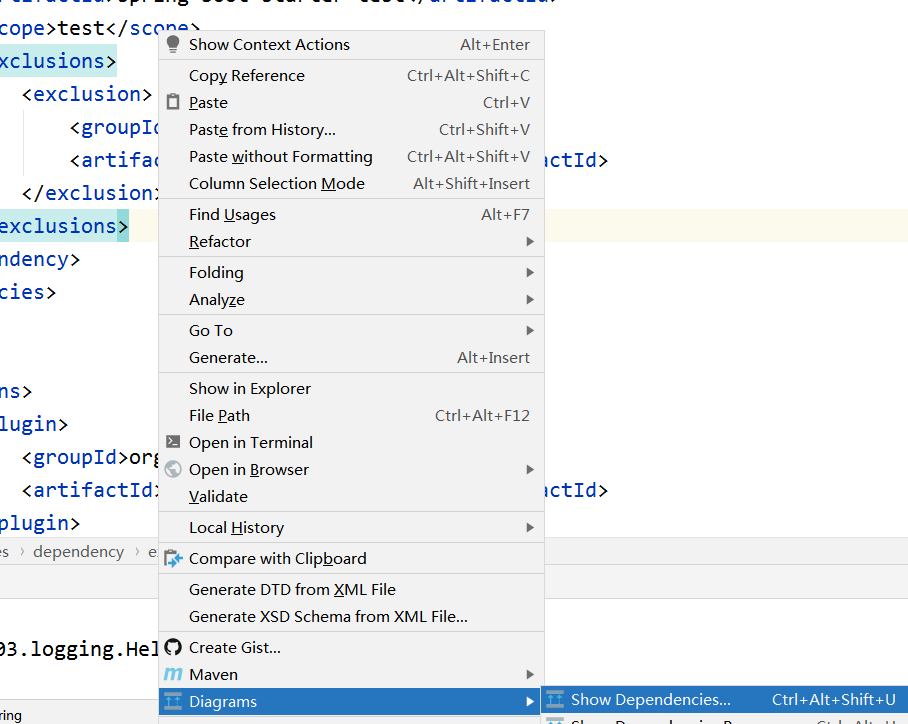


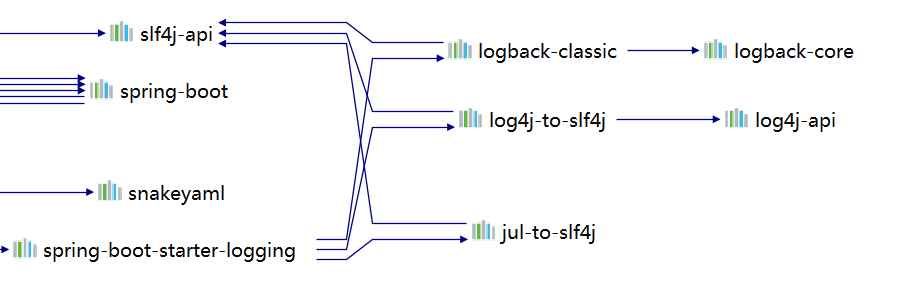
## 3.2Spring Boot日志依赖原理

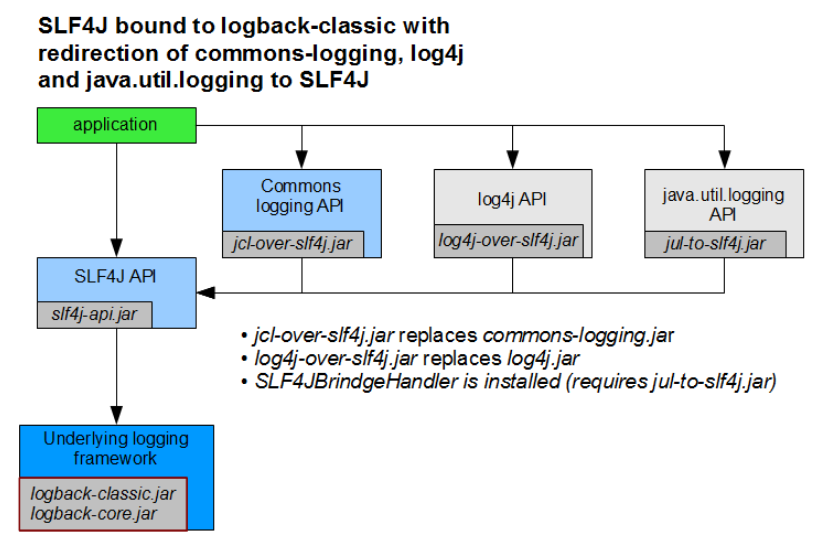


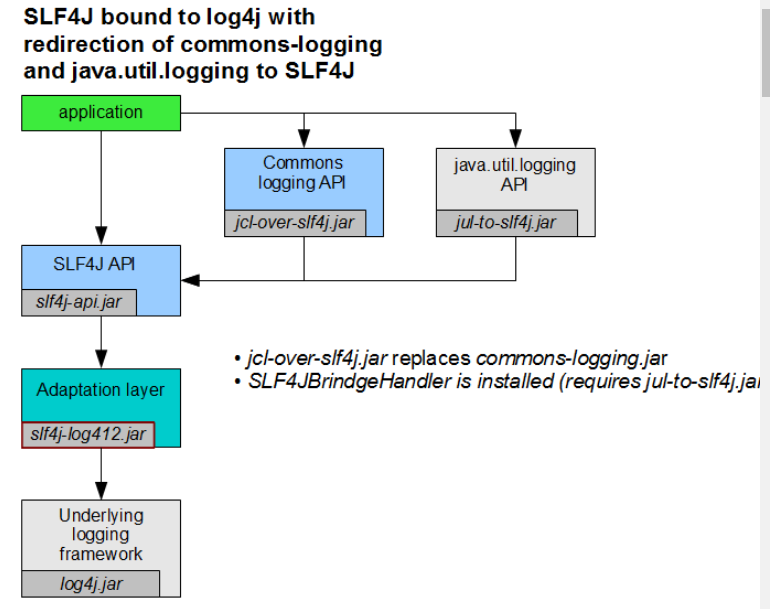
生成依赖关系图：

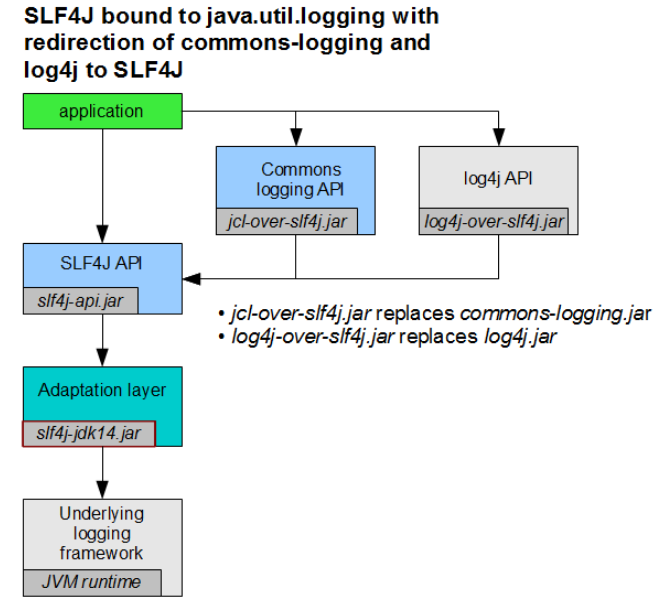
在pom.xml中任意位置点击右键：











## 3.3Spring Boot日志默认配置

日志级别配置：级别从低到高（trace<debug<info<warn<error）

Logging.level.包名=级别

设置了日志的级别信息以后，大于等于该级别的日志信息都会打印，不是只打印当前级别，默认info级别

日志文件与路径配置：

Logging.file.name= 文件完全名称

logging.path=文件输出路径

编写测试类：

|  |
| --- |
| @RunWith(SpringRunner.class) @SpringBootTest class Chapter03ApplicationTests {   @Test  void contextLoads() {  *//获取logger对象,当前类的class对象* Logger logger = LoggerFactory.*getLogger*(Chapter03ApplicationTests.class);  logger.trace("这是trace级别的日志");  logger.debug("这是debug级别的日志");  logger.info("这是info级别的日志");  logger.warn("这是warn级别的日志");  logger.error("这是error级别的日志");  *//为什么有些打印，有些不打印?springboot有默认的日志级别  //默认的日志级别是info，也就info，比info级别高的warn，error，这样级别的会打印  //比info级别低的日志信息不会打印* }  } |

执行结果：

|  |
| --- |
| 2020-07-15 15:10:15.204 INFO 24300 --- [ main] c.i.chapter03.Chapter03ApplicationTests : 这是info级别的日志  2020-07-15 15:10:15.204 WARN 24300 --- [ main] c.i.chapter03.Chapter03ApplicationTests : 这是warn级别的日志  2020-07-15 15:10:15.204 ERROR 24300 --- [ main] c.i.chapter03.Chapter03ApplicationTests : 这是error级别的日志 |

我们发现这里，打印了3条日志信息，包括info，warn和error。说明springboot默认的日志级别是info级别。在哪里定义的呢？

Springboot包下：

Logging包下的logback包下，有一个base.xml文件和defaults.xml文件。

Base.xml文件的内容：

|  |
| --- |
| <included>  <include resource="org/springframework/boot/logging/logback/defaults.xml" />  <property name="LOG\_FILE" value="${LOG\_FILE:-${LOG\_PATH:-${LOG\_TEMP:-${java.io.tmpdir:-/tmp}}}/spring.log}"/>  <include resource="org/springframework/boot/logging/logback/console-appender.xml" />  <include resource="org/springframework/boot/logging/logback/file-appender.xml" />  <root level="INFO">  <appender-ref ref="CONSOLE" />  <appender-ref ref="FILE" />  </root> </included> |

Level：设置日志的级别为info。可以在控制台打印，也可以写入文件。

在全局配置文件中设置日志的级别：

|  |
| --- |
| *#设置日志的级别* logging.level.com.icss=debug |

这是打印的日志内容会从debug级别开始。

还可以将日志写入文件中：

|  |
| --- |
| *#将日志写入文件,在当前目录下生成一个logging.log文件* logging.file.name=logging.log |

上面的文件名可以是全路径名。比如c:/logging.log

还可以定义日志文件的目录

|  |
| --- |
| *#指定日志输出的路径，，日志的名字spring.log* logging.file.path=d:/xyz/log |

如果不写盘符，直接写/log:*在工程所在的盘符下创建一个log文件夹*

## 3.4Spring Boot自定义日志

日志的格式配置：

logging.pattern.console=%d{yyyy‐MM‐dd} [%thread] %‐5level %logger{50} ‐ %msg%n

设置日志文件的输出格式

logging.pattern.file=%d{yyyy‐MM‐dd} === [%thread] === %‐5level === %logger{50} === %msg%n

解释：%d表示日期时间，%thread表示线程名，%‐5level:级别从左显示5个字符宽度 %logger{50} 表示logger名字最长50个字符，否则按照句点分割。 %msg:日志消息，%n是换行符

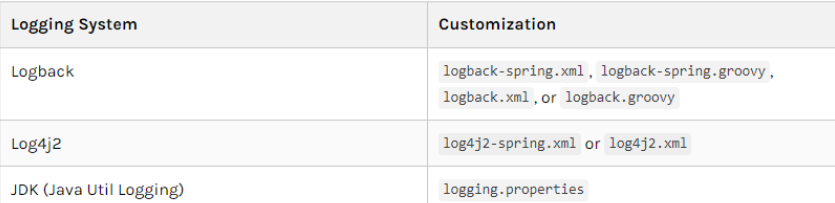
在全局配置文件中，设置控制台日志的格式：

|  |
| --- |
| *#配置控制台日志的格式：* logging.pattern.console=%d{yyyy-MM-dd} \*\*\*\* [%thread] \*\*\*\* %-5level %logger{50} - %msg%n |

还可以设置文件中日志的格式：

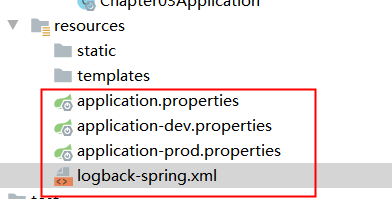
|  |
| --- |
| *#日志文件的输出的格式* logging.pattern.file=%d{yyyy-MM-dd} ### [%thread] ### %-5level ### %logger{50} ### %msg%n |

在类路径下放日志框架自已的配置文件，Spring Boot就会加载它的配置并生效。



logback-spring.xml:日志框架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Profile功能。

如果没有配置profile日志框架，强制使用profile功能，则系统报错。



Logback-spring.xml部分代码，配置profile，在不同的profile环境下，日志的格式不同

|  |
| --- |
| <layout class="ch.qos.logback.classic.PatternLayout">  <springProfile name="dev">  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} \*\*\*\* [%thread] \*\*\*\* %-5level %logger{50} - %msg%n</pattern>  </springProfile>  <springProfile name="!dev">  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} #### [%thread] #### %-5level %logger{50} - %msg%n</pattern>  </springProfile> </layout> |

## 3.5SpringBoot切换日志框架

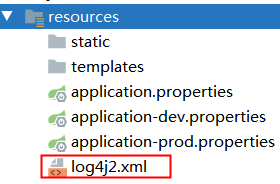
1. 先排除当前使用的实现依赖

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

2.导入自已要实现日志方式的依赖

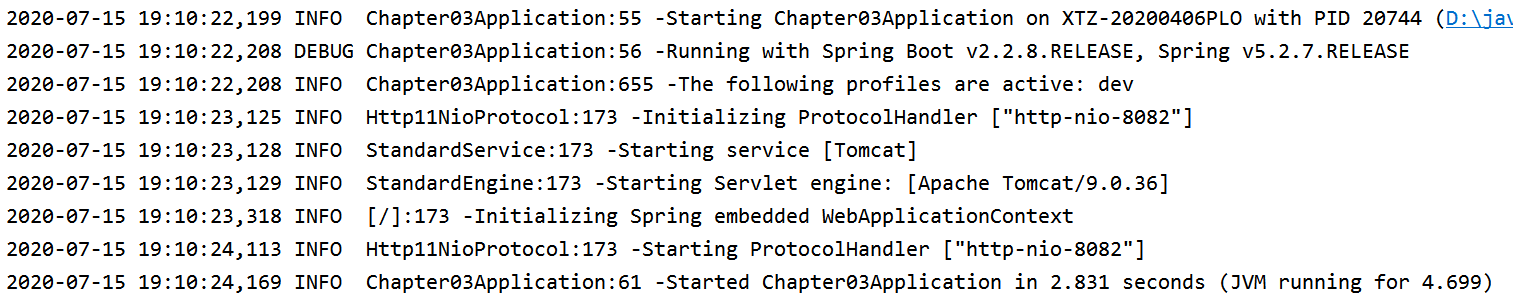
|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-log4j2</artifactId> </dependency> |

3.编写log4j2.xml配置文件 log4j2的配置xml，yml，json，不能再写成log4j2.properties



|  |
| --- |
| *<?*xml version="1.0" encoding="UTF-8"*?> <!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL --> <!-- status log4j2内部日志级别 -->* <configuration status="INFO">  *<!-- 全局参数 -->* <Properties>  <Property name="pattern">%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %c{1}:%L -%m%n</Property>  <Property name="displayName">EurekaServer</Property>  </Properties>  <Appenders>  <Console name="console" target="SYSTEM\_OUT" follow="true">  <PatternLayout>  <pattern>${pattern}</pattern>  </PatternLayout>  </Console>  *<!-- 文件 每次运行程序会自动清空，由append属性决定 -->* <File name="error" fileName="${displayName}\_error.log" append="false">  *<!-- 指定error 级别的日志 -->* <ThresholdFilter level="ERROR" onMatch="ACCEPT"  onMismatch="DENY" />  <PatternLayout>  <pattern>${pattern}</pattern>  </PatternLayout>  </File>  *<!-- 滚动文件 -->* <RollingFile name="rollingFile" fileName="${displayName}.log"  filePattern="${displayName}\_%d{yyyy-MM-dd}.log">  <PatternLayout>  <pattern>${pattern}</pattern>  </PatternLayout>  *<!-- 按大小划分 -->* <SizeBasedTriggeringPolicy size="50 MB" />  </RollingFile>  </Appenders>  <Loggers>  *<!-- <Logger name="org.apache.catalina.util.LifecycleBase" level="ERROR"  /> <Logger name="org.apache.coyote.http11.Http11NioProtocol" level="WARN"  /> <Logger name="org.apache.tomcat.util.net.NioSelectorPool" level="WARN"  /> -->* <Logger name="org.springframework" level="WARN" />  <Logger name="com.icss" level="DEBUG" />  <Root level="INFO">  <AppenderRef ref="console"></AppenderRef>  <AppenderRef ref="error"></AppenderRef>  <AppenderRef ref="rollingFile"></AppenderRef>  </Root>  </Loggers> </configuration> |

执行结果：



# 4SpringBoot与web开发

## 4.1静态资源映射规则

/aaa/\* 与 /aaa/\*\*区别？

/aaa/\* :aaa直接子目录

/aaa/\*\*：aaa的子目录及其子目录的子目录，就是aaa下面的所有目录，不管多少级

1.所有“/webjars/\*\*”找资源文件classpath:/META-INF/resources/webjars/

百度webjars，点击进入webjars官网，就可以获取各种前端框架的maven的坐标

比如jquery



引入坐标：

|  |
| --- |
| <dependency>  <groupId>org.webjars</groupId>  <artifactId>jquery</artifactId>  <version>3.5.1</version> </dependency> |

启动服务器，输入jquery访问路径，就可以获取访问jquery了。

|  |
| --- |
|  |

Jquery的存储位置：

|  |
| --- |
|  |

其实在路径上写的/webjars/jquery/3.5.1/jquery.js就映射到了/META-INF/resources/webjars/jquery/3.5.1/jquery.js

2.“/\*\*” 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:

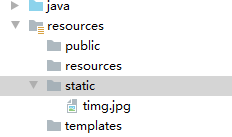
"classpath:/META-INF/resources/",

"classpath:/resources/",

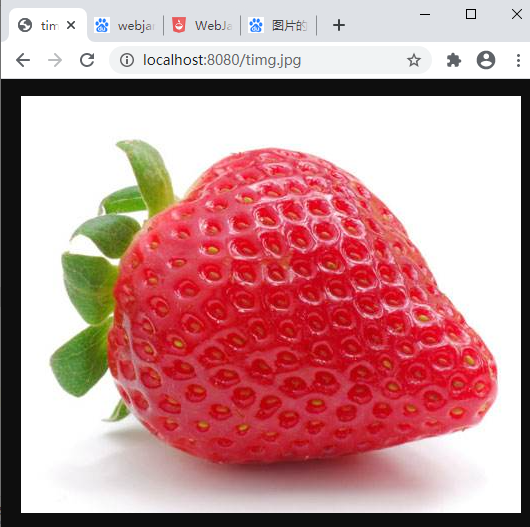
"classpath:/static/",

"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

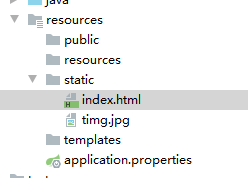


可以直接在地址栏中访问：



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”

|  |
| --- |
|  |

如果要引用webjars下面的jquery，直接引入jquery路径即可

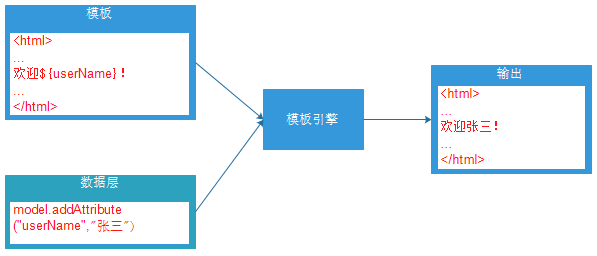
## 4.2静态模板引擎使用

### 4.2.1概述

常见的模板引擎有：

JSP、Freemarker、Thymeleaf 等

模板引擎原理如下：



Thymeleaf 模板引擎功能强大，使用简单，Spring Boot推荐使用。

### 4.2.2thymeleaf使用

1. 导入依赖

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

1. 配置thymeleaf

|  |
| --- |
| *#设置thymeleaf不缓存* spring:  thymeleaf:  cache: false |

1. 将静态页面放入templates



1. 在静态页面中导入thymeleaf的名称空间

|  |
| --- |
| <html lang="en" xmlns:th="http://www.thymeleaf.org"> |

1. 创建controller：

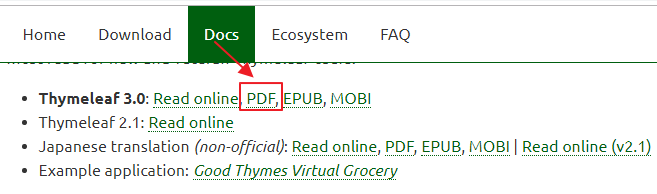
|  |
| --- |
| @Controller public class PageController {   @RequestMapping({"/","/index.html"})  public String toIndex(Model model){  model.addAttribute("username","rose");  return "index";  } } |

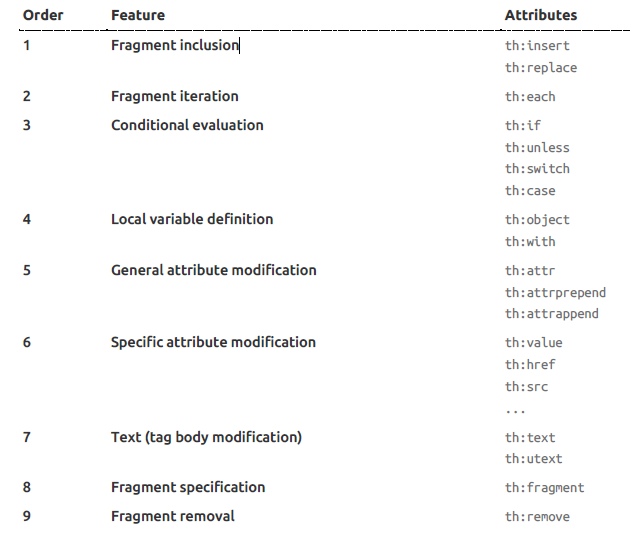
1. 渲染页面

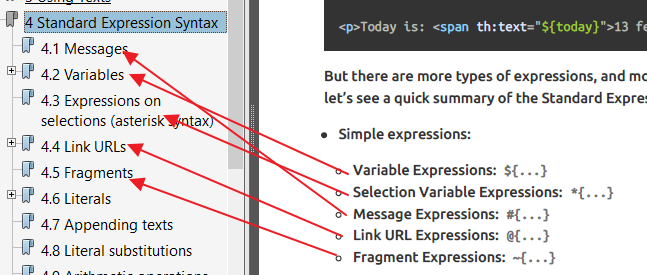
|  |
| --- |
| <!DOCTYPE html> <html lang="en" xmlns:th="http://www.thymeleaf.org"> <head>  <meta charset="UTF-8">  <title>index</title> </head> <body>  <h1>index</h1>  <span th:text="${username}">XXX</span>,欢迎你！ </body> </html> |

### 4.2.3Thymeleaf语法

参考官方文档: https://www.thymeleaf.org/

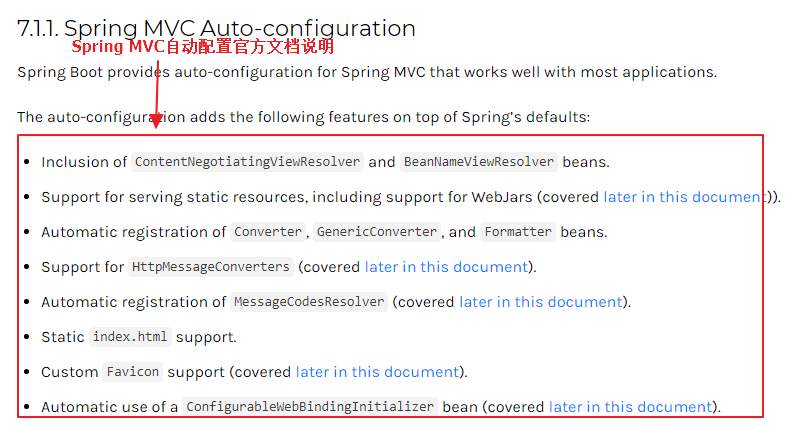




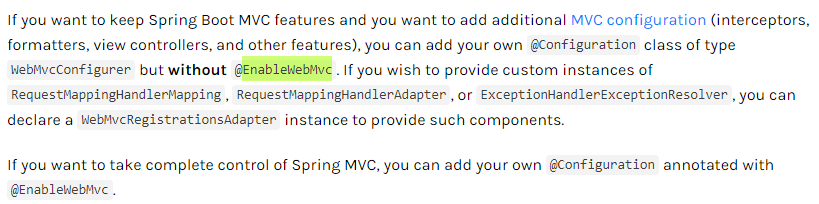


## 4.3SpringMVC自动配置原理

### 4.3.1自动配置原理



### 4.3.2SpringMVC自动配置原理-扩展与全面接管



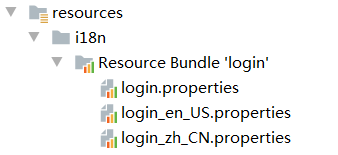
如果想全面控制Spring MVC， 可以添加自己的@Configuration， 并使用@EnableWebMvc对其注解。 如果想保留Spring Boot MVC的特性， 并只是添加其他的MVC配置(拦截器， formatters， 视图控制器等)， 可以添加自己的WebMvcConfigurerAdapter类型的@Bean（不使用@EnableWebMvc注解） 。

|  |
| --- |
| **import** org.springframework.context.annotation.Configuration; **import** org.springframework.web.servlet.config.annotation.\*; */\*\*  \* 相当于 写了 xxx.xml配置文件、  \* 实例化对象到spring容器 <bean> @Bean  \* 扩展springmvc的内容，实现一个接口  \*/* @EnableWebMvc*//使springmvc的自动配置失效，全面接管，如果不需要全面接管，就不要加该注解* @Configuration **public class** AppConfig **implements** WebMvcConfigurer {  @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  *//ddddd* }  @Override  **public void** configureViewResolvers(ViewResolverRegistry registry) {  *//dddddd* } } |

## 4.4Restful WebService风格web实践

### 4.4.1国际化

#### 4.4.1.1编写国际化配置文件



login.properties

|  |
| --- |
| login.tip=请登录 login.username=用户名 login.password=密码 login.inputlg=登录 |

login\_en\_US.properties

|  |
| --- |
| login.tip=please login login.username=username login.password=password login.inputlg=login |

login\_zh\_CN.properties

|  |
| --- |
| login.tip=请登录 login.username=用户名 login.password=密码 login.inputlg=登录 |

#### 4.4.1.2使用ResourceBundleMessageSource管理国际化资源文件

在全局配置文件中：

|  |
| --- |
| spring:  *#配置国际化资源* messages:  basename: i18n/login |

#### 4.4.1.3自定义localeResolver

|  |
| --- |
| package com.icss.chapter04crud0716.component;  import org.springframework.util.StringUtils; import org.springframework.web.servlet.LocaleResolver;  import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; import java.util.Locale;  public class MyLocaleResolver implements LocaleResolver {   @Override  public Locale resolveLocale(HttpServletRequest request) {  *//从request中获取locale信息* String parameter = request.getParameter("locale");*//zh\_CN  //获取当前的locale  //自动获取跟本地相关的locale数据* Locale locale = Locale.*getDefault*();  *//如果浏览器传过来的不是null，说明我们需要自己切换语言* if (!StringUtils.*isEmpty*(parameter)){  String[] split = parameter.split("\_");  locale= new Locale(split[0],split[1]);  }  return locale;  }   @Override  public void setLocale(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Locale locale) {   } } |

#### 4.4.1.4将自定义LocaleResolver实例化到容器

|  |
| --- |
| @Configuration public class AppConfig implements WebMvcConfigurer {   *//统一的视图控制器* @Override  public void addViewControllers(ViewControllerRegistry registry) {  *//一个路径对应一个视图名称* registry.addViewController("/").setViewName("login");  registry.addViewController("/index.html").setViewName("login");  registry.addViewController("/main.html").setViewName("main");  registry.addViewController("/welcome.html").setViewName("welcome");  }   *//将localResolver实例化到容器当中* @Bean  public LocaleResolver localeResolver(){  return new MyLocaleResolver();  } } |

### 4.4.2登录拦截器

#### 4.4.2.1编写controller

|  |
| --- |
| @Controller @RequestMapping("/user") public class UserController {   @Autowired  private UserService userService;   @PostMapping("/login")  public String login(String username, String password, HttpSession session, Model model){  User user = userService.login(username, password);  System.*out*.println(user);  if(user !=null){  *//登录成功* session.setAttribute("userLogin",user);  return "redirect:/main.html";  }else{  *//登录失败，设置失败信息，返回登录页面* model.addAttribute("msg","用户名或密码错误");  return "login";  }  } } |



#### 4.4.2.2编写登录拦截器

|  |
| --- |
| public class LoginInterceptor implements HandlerInterceptor {   *//重写前置拦截* @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  *//获取session，从session中取出user对象* User userLogin = (User) request.getSession().getAttribute("userLogin");  *//判断user是否为null* if(userLogin == null){  *//没有登录，回到登录页面* request.setAttribute("msg", "没有权限，请登录");  *//回到登录页面* request.getRequestDispatcher("/index.html").forward(request,response);  return false;  }  *//已经登录* return true;  } } |

#### 4.4.2.3配置拦截器

|  |
| --- |
| *//将登录的拦截器实例化到容器 //配置拦截器哪些路径拦截，哪些路径不拦截* @Override public void addInterceptors(InterceptorRegistry registry) {  *//创建拦截器对象，配置拦截的路径* registry.addInterceptor(new LoginInterceptor())  .addPathPatterns("/\*\*")  .excludePathPatterns("/","/index.html","/user/login",  "/css/\*\*","/js/\*\*","/images/\*\*","/lib/layui/\*\*","/fonts/\*\*");*//配置不拦截的路径* } |

### 4.4.3会员的CRUD操作

#### 4.4.3.1创建会员表和会员实体类

|  |
| --- |
| import java.util.Date;  public class Member {  private Integer id;  private String mbName;  private String email;  private String password;  private String phoneNumber;  private Date birthday;   public Integer getId() {  return id;  }   public void setId(Integer id) {  this.id = id;  }   public String getMbName() {  return mbName;  }   public void setMbName(String mbName) {  this.mbName = mbName;  }   public String getEmail() {  return email;  }   public void setEmail(String email) {  this.email = email;  }   public String getPassword() {  return password;  }   public void setPassword(String password) {  this.password = password;  }   public String getPhoneNumber() {  return phoneNumber;  }   public void setPhoneNumber(String phoneNumber) {  this.phoneNumber = phoneNumber;  }   public Date getBirthday() {  return birthday;  }   public void setBirthday(Date birthday) {  this.birthday = birthday;  }   public Member() {  }   public Member(Integer id, String mbName, String email, String password, String phoneNumber, Date birthday) {  this.id = id;  this.mbName = mbName;  this.email = email;  this.password = password;  this.phoneNumber = phoneNumber;  this.birthday = birthday;  }   @Override  public String toString() {  return "Member{" +  "id=" + id +  ", mbName='" + mbName + '\'' +  ", email='" + email + '\'' +  ", password='" + password + '\'' +  ", phoneNumber='" + phoneNumber + '\'' +  ", birthday=" + birthday +  '}';  } } |

#### 4.4.3.2查询所有会员

Controller：

|  |
| --- |
| */\*\*  \* 查询所有  \** ***@return*** *\*/* @GetMapping(**"/member/list"**) **public** String memberList(Model model){  **try** {  List<Member> members = **memberService**.findAll();  model.addAttribute(**"members"**,members);  } **catch** (Exception e) {  e.printStackTrace();  }  **return "/member/member-list"**; } |

Service：

|  |
| --- |
| @Override **public** List<Member> findAll() **throws** Exception {  List<Member> members = **null**;  **try** {  members = **memberMapper**.findAll();  } **catch** (Exception e) {  e.printStackTrace();  *//继续向上抛出* **throw** e;  }  **return** members; } |

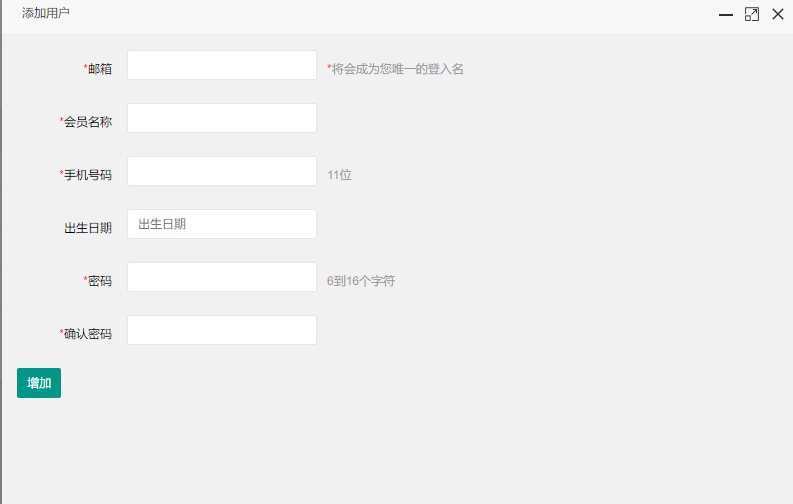
Mapper映射：

|  |
| --- |
| <**resultMap id="memberMap" type="Member"**>  <**id column="id" property="id"**></**id**>  <**result column="member\_name" property="mbName"**/>  <**result column="email" property="email"**/>  <**result column="password" property="password"**/>  <**result column="phone\_number" property="phoneNumber"**/>  <**result column="birthday" property="birthday"**/>  </**resultMap**> *<!-- 查询所有会员列表-->* <**select id="findAll" resultMap="memberMap"**>  select id,member\_name,email,password,phone\_number,birthday from member  </**select**> |

#### 4.4.3.3增加会员

设置日期格式：由于springboot默认的日期格式为dd/MM/yyyy,而我们从前端获取的日期格式为yyyy-MM-dd，所以需要重新配置

|  |
| --- |
| **spring**:*#设置日期格式* **mvc**:  **date-format**: yyyy-MM-dd |



Controller：

|  |
| --- |
| */\*\*  \* 跳转到添加的页面  \** ***@return*** *\*/* @GetMapping(**"/member"**) **public** String toAddPage(){  **return "/member/member-add"**; }  */\*\*  \* 添加会员  \** ***@return*** *\*/* @PostMapping(**"/member"**) **public** ResponseEntity<Boolean> save(Member member){  **try** {  **memberService**.save(member);  *//插入成功* **return** ResponseEntity.*ok*().body(**true**);  } **catch** (Exception e) {  e.printStackTrace();  *//插入失败* **return** ResponseEntity.*status*(HttpStatus.***INTERNAL\_SERVER\_ERROR***).body(**false**);  } } |

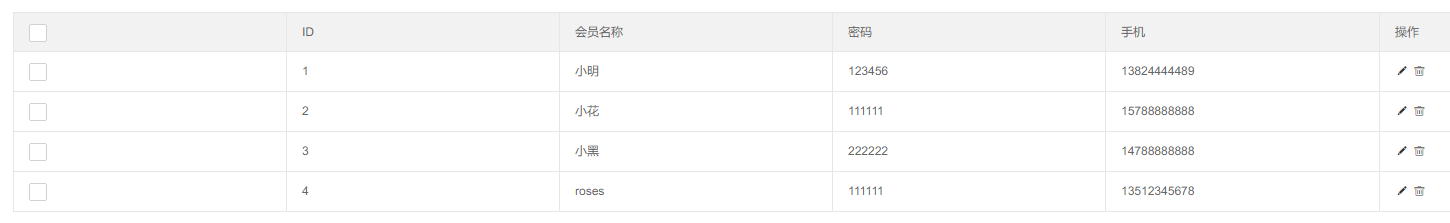
Service：

|  |
| --- |
| @Override **public void** save(Member member) **throws** Exception {  **try** {  **memberMapper**.save(member);  } **catch** (Exception e) {  e.printStackTrace();  **throw** e;  } } |

Mapper映射：

|  |
| --- |
| *<!--插入会员-->* <**insert id="save" parameterType="member"**>  insert into member (id,member\_name,email,password,birthday,phone\_number)  values(null,#{mbName},#{email},#{password},#{birthday},#{phoneNumber})  </**insert**> |

结果：



#### 4.4.3.4修改会员

修改使用put提交方式，需要开启隐藏方法过滤器：

|  |
| --- |
| **spring**:  *#设置put提交方式生效* **hiddenmethod**:  **filter**:  **enabled**: **true** |

Controller：

|  |
| --- |
| */\*\*  \* 通过id获取会员对象  \* @param id  \* @param model  \* @return  \*/* @GetMapping("/member/{id}") public String toEditPage(@PathVariable("id") Integer id,Model model){  *//通过id获取会员对象* Member member = null;  try {  member = memberService.findById(id);  } catch (Exception e) {  e.printStackTrace();  }  model.addAttribute("member",member);  return "/member/member-edit"; }  */\*\*  \* 更新会员信息  \* @param member  \* @return  \*/* @PutMapping("/member") public ResponseEntity<Boolean> doEditMember(Member member){  try {  memberService.updateMember(member);  return ResponseEntity.*status*(HttpStatus.*OK*).body(true);  } catch (Exception e) {  e.printStackTrace();  return ResponseEntity.*status*(HttpStatus.*INTERNAL\_SERVER\_ERROR*).body(false);  } } |

Service：

|  |
| --- |
| @Override public Member findById(Integer id) throws Exception {  try {  Member member = memberMapper.findById(id);  return member;  } catch (Exception e) {  e.printStackTrace();  throw e;  } }  @Override public void updateMember(Member member) throws Exception {  try {  memberMapper.updateMember(member);  } catch (Exception e) {  e.printStackTrace();  throw e;  } } |

Mapper映射：

|  |
| --- |
| <select id="findById" resultMap="memberMap">  select id,member\_name,email,password,phone\_number,birthday  from member  where id=#{id} </select>  <update id="updateMember" parameterType="Member">  update member set member\_name=#{mbName},email=#{email},  password=#{password},birthday=#{birthday},phone\_number=#{phoneNumber}  where id=#{id} </update> |

#### 4.4.3.5删除会员

删除使用delete请求。

Controller：

|  |
| --- |
| */\*\*  \* 删除会员  \* @param id  \* @return  \*/* @DeleteMapping("/member/{id}") public ResponseEntity<Boolean> deleteMember(@PathVariable("id") Integer id){  try {  memberService.deleteMember(id);  return ResponseEntity.*status*(HttpStatus.*OK*).body(true);  } catch (Exception e) {  e.printStackTrace();  return ResponseEntity.*status*(HttpStatus.*INTERNAL\_SERVER\_ERROR*).body(false);  } } |

Service：

|  |
| --- |
| @Override public void deleteMember(Integer id) throws Exception {  try {  memberMapper.deleteMember(id);  } catch (Exception e) {  e.printStackTrace();  throw e;  } } |

Mapper映射：

|  |
| --- |
| <delete id="deleteMember">  delete from member where id=#{id} </delete> |

### 4.4.4异常处理

#### 4.4.4.1默认的异常处理

ErrorMvcAutoConfiguration：错误处理自动配置类

DefaultErrorAttributes：主要用于页面共享

BasicErrorController：处理默认/error请求，并根据请求信息选择返回的数据类型

ErrorPageCustomizer ：用户自定义错误处理

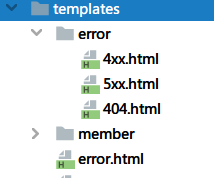
DefaultErrorViewResolver：默认的错误视图解释器

错误处理规则：

系统出现4xx或5xx的错误，ErrorPageCustomizer就会生效(定制错误的响应规则)，就会来到/error请求，会被BasicErrorController处理，最终由DefaultErrorViewResolver解析得到错误页面。

#### 4.4.4.2自定义异常的处理

如何定制错误的页面：

1. 有模板引擎的情况下，error/状态码; 【将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的error文件夹下】，发生此状态码的错误就会来到对应的页面。我们可以使用4xx和5xx作为错误页面的文件名来匹配这种类型的所有错误，精确优先(优先寻找精确的状态码.html)。
2. 

页面能获取的信息：

timestamp:时间戳

status:状态码

error:错误提示

exception:异常对象

message:异常消息

errors:JSR303数据校验的错误

在异常处理的页面，获取异常信息：

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>404</**title**> </**head**> <**body**>  404.html<**br**>  [[${status}]]<**br**>  [[${message}]]<**br**>  [[${path}]]<**br**> </**body**> </**html**> |

结果：

|  |
| --- |
|  |

2、没有模板引擎(模板引擎找不到错误页面)，静态资源文件夹下找。

3、以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页面。

如何定制错误的json数据：

1、自定义异常处理&返回定制json数据

2、转发到/error进行自适应响应效果处理

3、将我们的定制数据携带出去

自定义异常：

|  |
| --- |
| */\*\*  \* 自定义异常，编译期异常，继承Exception  \* 运行时异常，继承RuntimeException  \*/* **public class** MyException **extends** RuntimeException{   **public** MyException() {  **super**(**"运行时异常"**);  } } |

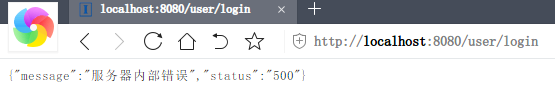
创建全局异常处理器：

|  |
| --- |
| @ControllerAdvice **public class** MyExceptionHanlder {   @ExceptionHandler(MyException.**class**)  @ResponseBody  **public** Map<String,Object> handle(){  *//将异常的信息封装到map中返回,这里只匹配了异常的类型，返回的并不是当前异常的信息  //更近一步的自定义可以返回当前异常的信息* Map<String,Object> map = **new** HashMap<String,Object>();  map.put(**"status"**,**"500"**);  map.put(**"message"**,**"服务器内部错误"**);  **return** map;  } } |

在controller中抛出异常：

|  |
| --- |
| @RequestMapping(**"/user/login"**) **public** String login(@RequestParam String username, @RequestParam String password, HttpServletRequest request){  *//判断用户名是否为admin，如果用户名为admin，就抛一个运行时的异常* **if**(**"admin"**.equals(username)){  *//抛出自定义异常* **throw new** MyException();  }  ……  } |

运行结果：



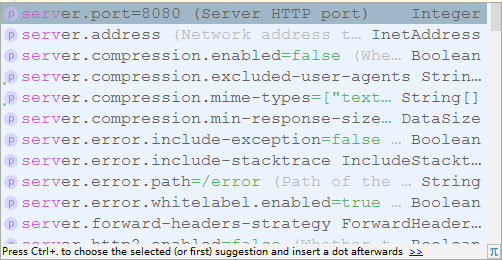
### 4.4.5内嵌容器

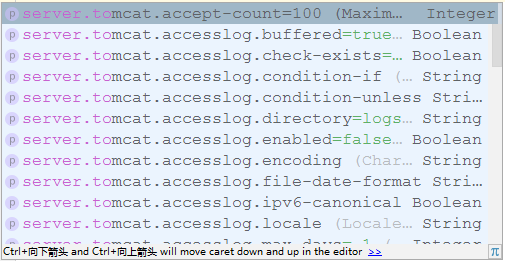
spring boot内置了三种servlet容器：tomcat，jetty，undertow

Spring Boot提供两种方式修改容器配置：

方法一：配置文件修改：通用配置为server.xxx进行设置，当然还可以使用server.容器名（如tomcat）对特定容器进行设置。

方法二：通过WebServerFactoryCustomizer进行程序注入。(XXXCustomizer)





通过配置文件配置服务器端口号：

|  |
| --- |
| *#配置端口号* **server**:  **port**: 8082 |

通过java配置端口号：

|  |
| --- |
| @Configuration **public class** MyServerConfig {   @Bean  **public** WebServerFactoryCustomizer<ConfigurableWebServerFactory> webServerFactoryCustomizer(){   WebServerFactoryCustomizer<ConfigurableWebServerFactory> factoryCustomizer = **new** WebServerFactoryCustomizer<ConfigurableWebServerFactory>() {  @Override  **public void** customize(ConfigurableWebServerFactory factory) {  factory.setPort(8082);  }  };  **return** factoryCustomizer;  } } |

启动：

|  |
| --- |
|  |

如果想替换默认的tomcat服务器，通过以下方法

1. 将Tomcat排除

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> *<!-- 排除tomcat-->* <**exclusions**>  <**exclusion**>  <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  <**groupId**>org.springframework.boot</**groupId**>  </**exclusion**>  </**exclusions**>  </**dependency**> |

1. 导入需要的服务器依赖

|  |
| --- |
| *<!-- 导入jetty依赖 -->* <**dependency**>  <**artifactId**>spring-boot-starter-jetty</**artifactId**>  <**groupId**>org.springframework.boot</**groupId**>  </**dependency**> |

启动：

|  |
| --- |
|  |

### 4.4.6注册web容器三大组件

|  |
| --- |
| 注册Servlet、Filter、Listener的web容器三大组件：  ServletRegistrationBean  FilterRegistrationBean  ServletListenerRegistrationBean |

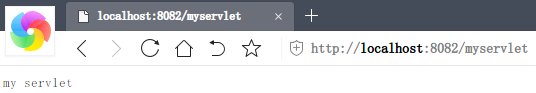
#### 4.4.6.1创建servlet

|  |
| --- |
| **public class** MyServlet **extends** HttpServlet {   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **super**.doPost(req, resp);  }   @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().write(**"my servlet"**);  } } |

#### 4.4.6.2注册servlet

|  |
| --- |
| @Bean  **public** ServletRegistrationBean servletRegistrationBean(){  *//创建Bean的对象,通过该对象将myServlet对象注册到容器中  //启动服务器，myservlet就被注册到spring容器中了。 // ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new MyServlet(),"/myservlet");* ServletRegistrationBean servletRegistrationBean = **new** ServletRegistrationBean();  *//调用对象的方法设置sevlet对象和映射的路径* servletRegistrationBean.setServlet(**new** MyServlet());  servletRegistrationBean.addUrlMappings(**"/myservlet"**);  **return** servletRegistrationBean;  } |

#### 4.4.6.3测试servlet



#### 4.4.6.4创建filter

|  |
| --- |
| **public class** MyFilter **implements** Filter {  @Override  **public void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws** IOException, ServletException {  System.***out***.println(**"do Filter"**);  *//放行* filterChain.doFilter(servletRequest,servletResponse);  } } |

#### 4.4.6.5注册filter

|  |
| --- |
| @Bean **public** FilterRegistrationBean filterRegistrationBean(){  *//创建注册对象* FilterRegistrationBean filterRegistrationBean = **new** FilterRegistrationBean();  filterRegistrationBean.setFilter(**new** MyFilter());  filterRegistrationBean.addUrlPatterns(**"/aaa"**,**"/myservlet"**);  **return** filterRegistrationBean; } |

#### 4.4.6.6测试filter

|  |
| --- |
|  |

#### 4.4.6.7创建listener

|  |
| --- |
| **public class** MyListener **implements** ServletContextListener {   @Override  **public void** contextInitialized(ServletContextEvent sce) {  System.***out***.println(**"contextInitialized"**);  }   @Override  **public void** contextDestroyed(ServletContextEvent sce) {  System.***out***.println(**"contextDestroyed"**);  } } |

#### 4.4.6.8注册listener

|  |
| --- |
| @Bean **public** ServletListenerRegistrationBean servletListenerRegistrationBean(){  *//创建注册对象* ServletListenerRegistrationBean servletListenerRegistrationBean = **new** ServletListenerRegistrationBean();  servletListenerRegistrationBean.setListener(**new** MyListener());  **return** servletListenerRegistrationBean; } |

#### 4.4.6.9测试listener

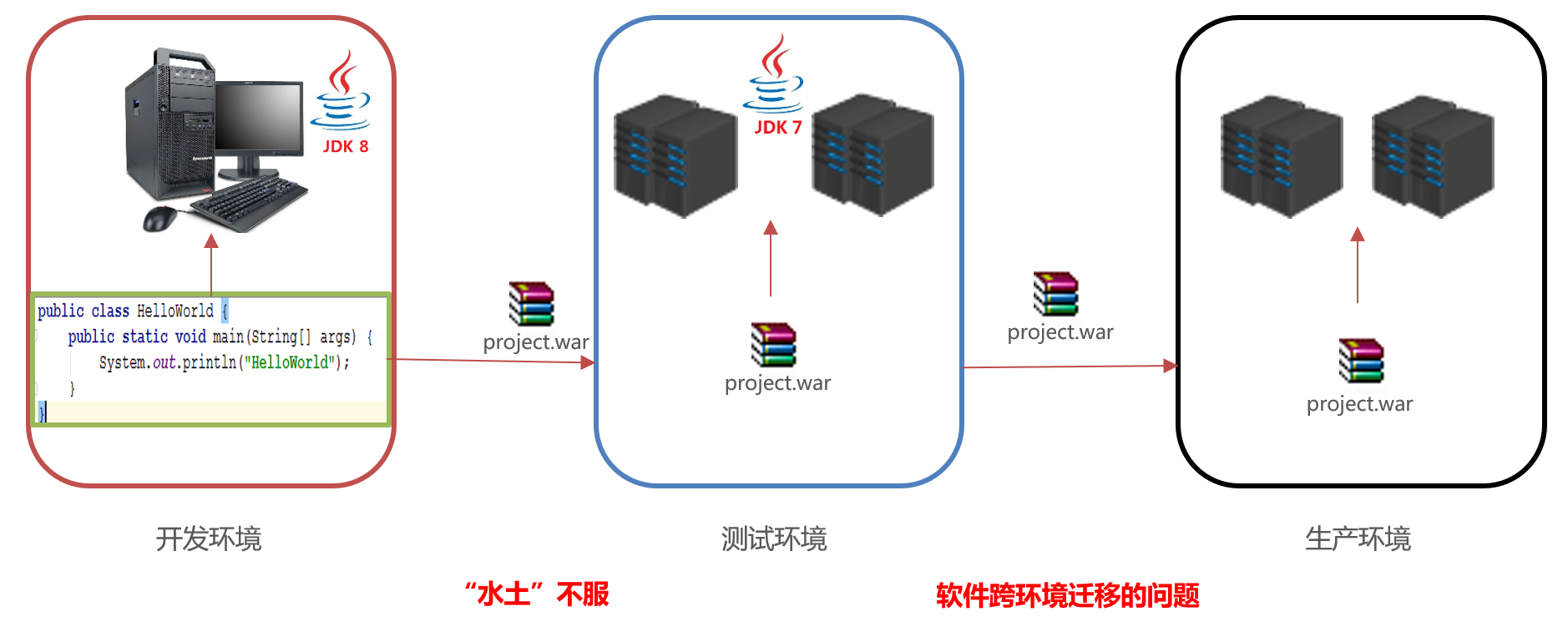
启动服务器，servletcontext初始化，正常退出服务器，servletcontext对象销毁

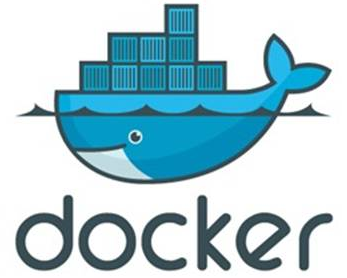
|  |
| --- |
|  |

# 5SpringBoot与docker容器

## 5.1什么是docker容器

我们写代码会接触到好几个环境:开发环境，测试环境，生产环境





Docker 是一个开源的应用容器引擎

诞生于 2013 年初，基于 Go 语言实现， dotCloud 公司出品（后改名为Docker Inc）

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上。

容器是完全使用沙箱机制，相互隔离

容器性能开销极低。

Docker 从 17.03 版本之后分为 CE（Community Edition: 社区版） 和 EE（Enterprise Edition: 企业版

Docker是一个开源的引擎，可以轻松的为任何应用创建一个轻量级的、可移植的、自给自足的容器。开发者在开发环境中编译测试通过的容器可以批量地在生产环境中部署，包括VMs（虚拟机）、bare metal、OpenStack 集群和其他的基础应用平台。

Docker容器完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是**容器性能开销极低**

Docker通常用于如下场景：

web应用的自动化打包和发布

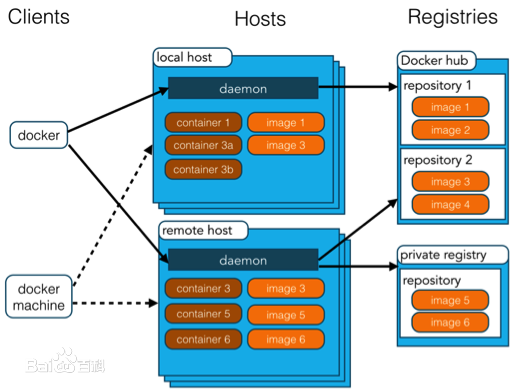
自动化测试和持续集成、发布

在服务型环境中部署和调整数据库或其他的后台应用

从头编译或者扩展现有的OpenShift或Cloud Foundry平台来搭建自己的PaaS环境

## 5.2docker核心组件

* docker主机(Host):安装了Docker程序的机器(Docker直接安装在操作系统之上)
* docker客户端(Client):连接docker主机进行操作
* docker仓库(Registry):用来保存各种打包好的软件镜像。仓库可看成一个代码控制中心，用来保存镜像。
* docker镜像(Images):软件打包好的镜像;放在docker仓库中。Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统。
* docker容器(Container):镜像启动后的实例称为一个容器;容器是独立运行的一个或一组应用。镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和对象一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。



## 5.3安装docker

Docker可以运行在MAC、Windows、CentOS、UBUNTU等操作系统上，本课程基于CentOS 7 安装Docker。官网：<https://www.docker.com>

**在Linux中使用yum命令进行安装时出现如下提示：**

*另外一个程序锁定了 yum；等待它退出……*

**出现原因：**　　  
yum命令一次只能安装一个软件，所以当下载安装第二个软件包时，系统进程锁会锁定yum，这时，即使关闭进程甚至关闭虚拟机重启后，再执行yum命令还是会出现同样情况。

**解决方法：**  
强制采用rm -f /var/run/yum.pid命令关闭yum进程

1、yum 包更新到最新

yum update

2、安装需要的软件包， yum-util 提供yum-config-manager功能，另外两个是devicemapper驱动依赖的

yum install -y yum-utils device-mapper-persistent-data lvm2

3、 设置yum源

yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

4、 安装docker，出现输入的界面都按 y

yum install -y docker-ce

5、 查看docker版本，验证是否验证成功

docker -v

1、检查内核版本，必须是3.10及以上

[root@localhost~]# uname ‐r

2、安装docker

[root@localhost~]# yum install docker

3、输入y确认安装

## 5.4配置Docker镜像加速器

默认情况下，将来从docker hub（<https://hub.docker.com/>）上下载docker镜像，太慢。一般都会配置镜像加速器：

* USTC：中科大镜像加速器（https://docker.mirrors.ustc.edu.cn）
* 阿里云
* 网易云
* 腾讯云

设置阿里云镜像加速器：

### 5.4.1注册阿里云账号

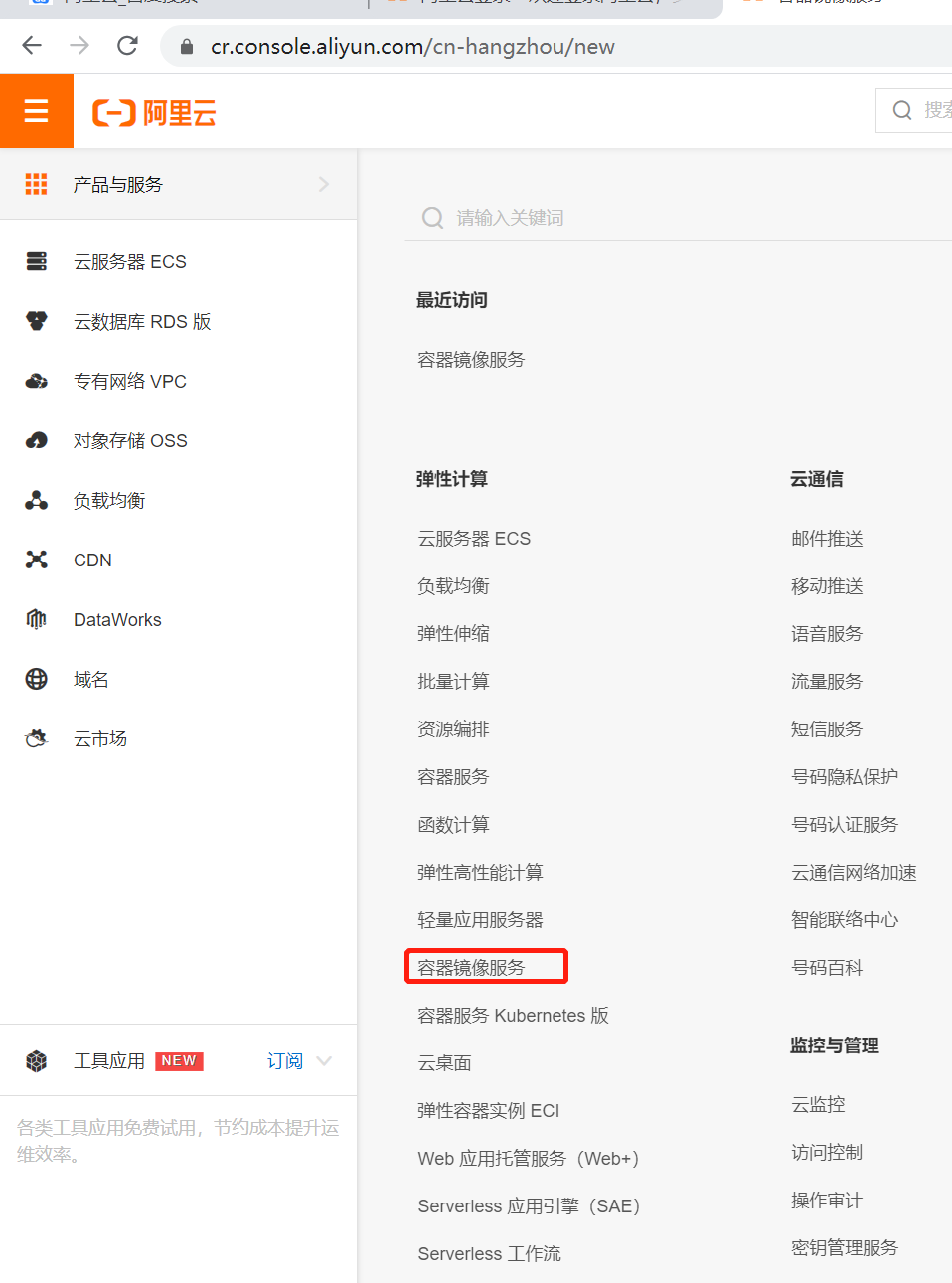


### 5.4.2使用支付宝扫描登录

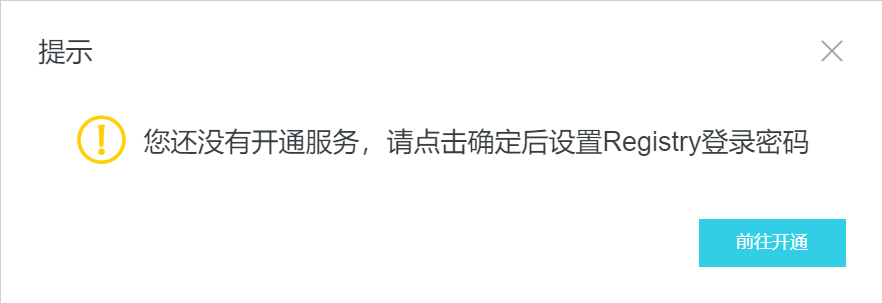


借助手机，完成一系列注册，登录和验证等工作。

### 5.4.3开启容器镜像服务



第一次使用该服务，需要授权，即开通密码。







### 5.4.4设置镜像加速器



上面命令的意思是：在etc/docker文件夹下创建daemon.json文件，在文件中写入如下内容：

{

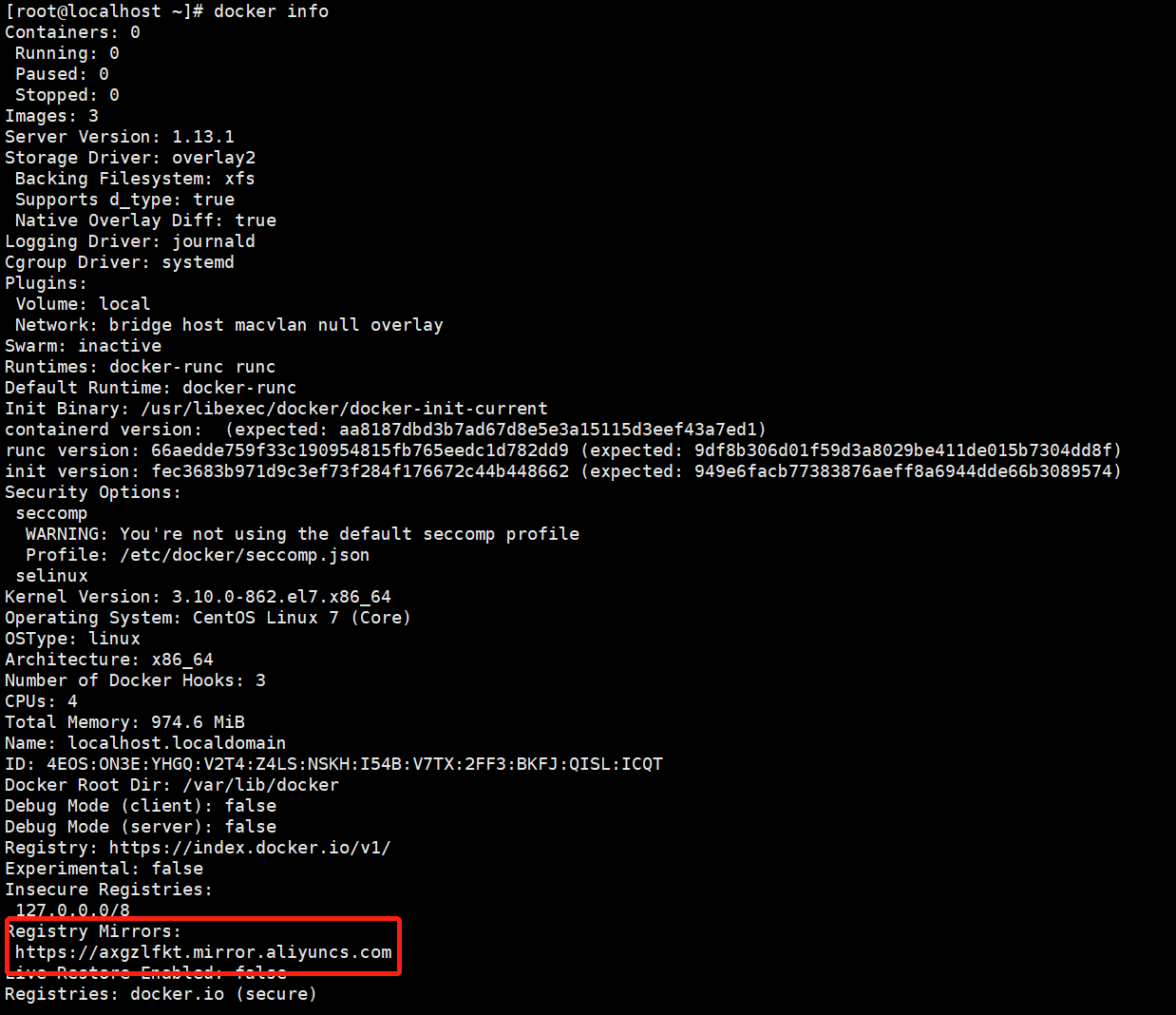
"registry-mirrors": ["https://axgzlfkt.mirror.aliyuncs.com"]

}

然后重新加载daemon文件，重启docker。

实际操作上，只需将上面所有的命令复制，粘贴，运行即可。

5.4.5查看docker信息，确认镜像配置成功



## 5.5 Docker命令

### 5.5.1进程相关命令

1、启动docker服务

[root@localhost~]# systemctl start docker

2、查看docker版本

[root@localhost~]# docker -v

3、停止docker服务

[root@localhost~]# systemctl stop docker

4、重启docker服务

[root@localhost~]# systemctl restart docker

5、查看docker服务的状态

[root@localhost~]# systemctl status docker

6、开机启动docker服务

[root@localhost~]# systemctl enable docker

### 5.5.2镜像相关命令

docker镜像库网址： https://hub.docker.com/

1、搜索镜像

* docker search 关键字: 从仓库搜索关键字的镜像
* docker search mysql

1. 拉取镜像

* docker pull 镜像名:tag

从Docker仓库下载镜像到本地，镜像名称格式为 名称:版本号，如果版本号不指定则是最新的版本。如果不知道镜像版本，可以去docker hub 搜索对应镜像查看。

docker pull mysql

1. 查看镜像：查看本地所有的镜像

docker images

docker images –q # 查看所用镜像的id

1. 删除镜像：删除本地镜像

docker rmi 镜像id # 删除指定本地镜像

docker rmi `docker images -q` # 删除所有本地镜像

### 5.5.3容器相关命令

1、查看容器

docker ps # 查看正在运行的容器

docker ps –a # 查看所有容器

2、创建并启动容器

docker run 参数

参数说明：

-i：保持容器运行。通常与 -t 同时使用。加入it这两个参数后，容器创建后自动进入容器中，退出容器后，容器自动关闭。

-t：为容器重新分配一个伪输入终端，通常与 -i 同时使用。

-d：以守护（后台）模式运行容器。创建一个容器在后台运行，需要使用docker exec 进入容器。退出后，容器不会关闭。

-it 创建的容器一般称为交互式容器，-id 创建的容器一般称为守护式容器

--name：为创建的容器命名。

3、进入容器

docker exec 参数 # 退出容器，容器不会关闭

4、停止容器

docker stop 容器名称

1. 启动容器

docker start 容器名称

1. 删除容器：如果容器是运行状态，需要停止容器才能删除

docker rm 容器名称

1. 查看容器信息

docker inspect 容器名称

## 5.5学会docker容器操作（mysql实践）

# 6SpringBoot与数据访问

## 6.1Spring Data JDBC

Spring Boot默认支持多种数据源：

spring.datasource.tomcat、spring.datasource.hikari、spring.datasource.dbcp2、自定义数据源

提示：根据配置创建数据源，默认使用hikari连接，使用 spring.datasource.type指定自定义的数据源类型

DataSourceInitializer：定义了运行建表语句和运行插入数据的sql语句

Spring Boot自动配置了JdbcTemplate操作数据库

### 6.1.1导入mysql驱动

|  |
| --- |
| *<!--导入mysql驱动-->* <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

### 6.1.2配置数据源

|  |
| --- |
| spring:  datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://localhost:3306/mybatis?serverTimezone=UTC  username: root  password: 123456 |

### 6.1.3测试数据源

在配置文件中书写了datasource的配置后，springboot即可自动配置数据源，直接注入使用即可，测试如下：

|  |
| --- |
| @RunWith(SpringRunner.class) @SpringBootTest class Spring07ApplicationTests {   @Autowired  private DataSource dataSource;   @Test  void contextLoads() throws SQLException {  System.*out*.println(dataSource.getClass());  Connection connection = dataSource.getConnection();  System.*out*.println(connection);  connection.close();  } |

执行结果如下：

|  |
| --- |
|  |

### 6.1.4注入JdbcTemplate

有了数据源，springboot可以自动配置JdbcTemplate，只需使用时注入即可。

在UserService中注入JdbcTemplate：

|  |
| --- |
| @Service public class UserServiceImpl implements UserService {   @Autowiredprivate JdbcTemplate jdbcTemplate;   @Override  public User login(String username, String password) {  String sql = "select *\** from user where username=? and password=?";  return jdbcTemplate.queryForObject(sql,new BeanPropertyRowMapper<User>(User.class),username,password);  } |

### 6.1.5测试

|  |
| --- |
| @RunWith(SpringRunner.class) @SpringBootTest class Spring07ApplicationTests {   @Autowired  private UserService userService;   @Test  void testService(){  User user = userService.login("tom", "xyz");  System.*out*.println(user);  }  } |

执行结果：



## 6.2Spring Data JPA

### 6.2.1什么是JPA

#### 6.2.1.1什么是ORM

ORM（Object-Relational Mapping） 表示对象关系映射。在面向对象的软件开发中，通过ORM，就可以把对象映射到关系型数据库中。只要有一套程序能够做到建立对象与数据库的关联，操作对象就可以直接操作数据库数据，就可以说这套程序实现了ORM对象关系映射。简单的说：ORM就是建立实体类和数据库表之间的关系，从而达到操作实体类就相当于操作数据库表的目的。ORM是一种解决问题的思想。

#### 6.2.1.2JPA的概念

JPA是Java Persistence API的简称，中文名Java持久层API，是JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到数据库中。

JPA由EJB 3.0软件专家组开发，作为JSR-220实现的一部分。但它又不限于EJB 3.0，你可以在Web应用、甚至桌面应用中使用。JPA的宗旨是为POJO提供持久化标准规范。总的来说，JPA包括以下3方面的技术：

* ORM映射元数据，JPA支持XML和JDK 5.0注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中；
* JPA 的API，用来操作实体对象，执行CRUD操作，框架在后台替我们完成所有的事情，开发者从繁琐的JDBC和SQL代码中解脱出来。

查询语言，这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序的SQL语句紧密耦合。

#### 6.2.1.3JPA的优势

* **标准化**

JPA 是 JCP 组织发布的 Java EE 标准之一，因此任何声称符合 JPA 标准的框架都遵循同样的架构，提供相同的访问API，这保证了基于JPA开发的企业应用能够经过少量的修改就能够在不同的JPA框架下运行。

* **容器级特性的支持**

JPA框架中支持大数据集、事务、并发等容器级事务，这使得 JPA 超越了简单持久化框架的局限，在企业应用发挥更大的作用。

* **简单方便**

JPA的主要目标之一就是提供更加简单的编程模型：在JPA框架下创建实体和创建Java 类一样简单，没有任何的约束和限制，只需要使用 javax.persistence.Entity进行注释，JPA的框架和接口也都非常简单，没有太多特别的规则和设计模式的要求，开发者可以很容易的掌握。JPA基于非侵入式原则设计，因此可以很容易的和其它框架或者容器集成

* **查询能力**

JPA的查询语言是面向对象而非面向数据库的，它以面向对象的自然语法构造查询语句，可以看成是Hibernate HQL的等价物。JPA定义了独特的JPQL（Java Persistence Query Language），JPQL是EJB QL的一种扩展，它是针对实体的一种查询语言，操作对象是实体，而不是关系数据库的表，而且能够支持批量更新和修改、JOIN、GROUP BY、HAVING 等通常只有 SQL 才能够提供的高级查询特性，甚至还能够支持子查询。

* **高级特性**

JPA 中能够支持面向对象的高级特性，如类之间的继承、多态和类之间的复杂关系，这样的支持能够让开发者最大限度的使用面向对象的模型设计企业应用，而不需要自行处理这些特性在关系数据库的持久化。

#### 6.2.1.4JPA的供应商

JPA 的目标之一是制定一个可以由很多供应商实现的API，并且开发人员可以编码来实现该API，而不是使用私有供应商特有的API。因此开发人员只需使用供应商特有的API来获得JPA规范没有解决但应用程序中需要的功能。尽可能地使用JPA API，但是当需要供应商公开但是规范中没有提供的功能时，则使用供应商特有的API。

* **Hibernate**

JPA是需要Provider来实现其功能的，Hibernate就是JPA Provider中很强的一个，应该说无人能出其右。从功能上来说，JPA就是Hibernate功能的一个子集。Hibernate 从3.2开始，就开始兼容JPA。Hibernate3.2获得了Sun TCK的JPA(Java Persistence API) 兼容认证。

只要熟悉Hibernate或者其他ORM框架，在使用JPA时会发现其实非常容易上手。

* **OpenJPA**

OpenJPA 是 Apache 组织提供的开源项目，它实现了 EJB 3.0 中的 JPA 标准，为开发者提供功能强大、使用简单的持久化数据管理框架。OpenJPA 封装了和关系型数据库交互的操作，让开发者把注意力集中在编写业务逻辑上。OpenJPA 可以作为独立的持久层框架发挥作用，也可以轻松的与其它 Java EE 应用框架或者符合 EJB 3.0 标准的容器集成。

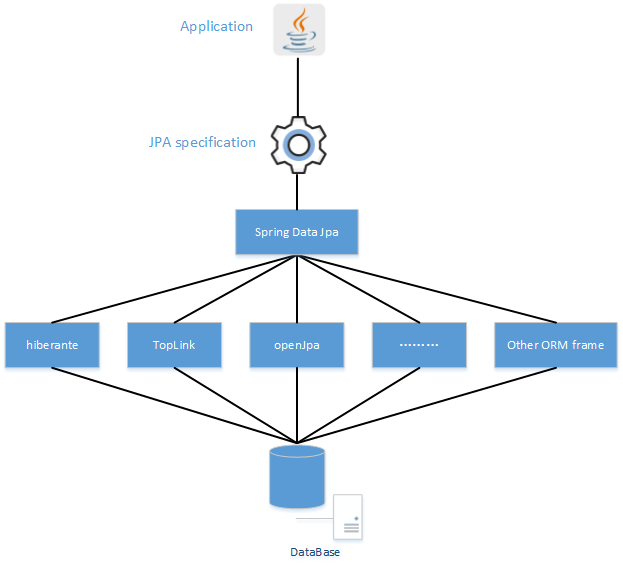
* **TopLink**

TopLink，是位居第一的Java对象关系可持续性体系结构，原署WebGain公司的产品，后被Oracle收购，并重新包装为Oracle AS TopLink。TOPLink为在关系数据库表中存储 Java 对象和企业 Java 组件 (EJB) 提供高度灵活和高效的机制。TopLink 为开发人员提供极佳的性能和选择，可以与任何数据库、任何应用服务器、任何开发工具集和过程以及任何 J2EE 体系结构协同工作。

### 6.2.2什么是SpringDataJPA

Spring Data JPA 是 Spring 基于 ORM 框架、JPA 规范的基础上封装的一套JPA应用框架，可使开发者用极简的代码即可实现对数据库的访问和操作。它提供了包括增删改查等在内的常用功能，且易于扩展！学习并使用 Spring Data JPA 可以极大提高开发效率！

Spring Data JPA 让我们解脱了DAO层的操作，基本上所有CRUD都可以依赖于它来实现,在实际的工作工程中，推荐使用Spring Data JPA + ORM（如：hibernate）完成操作，这样在切换不同的ORM框架时提供了极大的方便，同时也使数据库层操作更加简单。



spring data jpa、jpa以及ORM框架之间的关系

### 6.2.3SpringDataJPA与JPA和Hibernate之间的关系

JPA是一套规范，内部是有接口和抽象类组成的。hibernate是一套成熟的ORM框架，而且Hibernate实现了JPA规范，所以也可以称hibernate为JPA的一种实现方式，我们使用JPA的API编程，意味着站在更高的角度上看待问题（面向接口编程）

Spring Data JPA是Spring提供的一套对JPA操作更加高级的封装，是在JPA规范下的专门用来进行数据持久化的解决方案。

### 6.2.4SpringDataJPA入门案例

导入maven坐标：

|  |
| --- |
| <dependencies> *<!-- mysql驱动-->* <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.45</version>  </dependency> *<!-- SpringDataJPA-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  <version>2.3.4.RELEASE</version>  </dependency> *<!-- 测试-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <version>2.3.4.RELEASE</version>  </dependency>   </dependencies> |

编写application.yml文件

|  |
| --- |
| spring:  datasource:  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://localhost:3306/springdata?serverTimezone=UTC  username: root  password: root  jpa:  hibernate:  ddl-auto: update  show-sql: true |

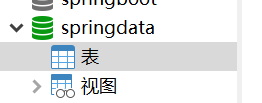
Jpa.hibernate.ddl-auto:取值有4种：

create:  
每次应用启动的时候会重新根据实体建立表，之前的表和数据都会被删除。  
create-drop:  
和上面的功能一样，但是多了一样，就是在应用关闭的时候，也就是sessionFactory一关闭，会把表删除。  
update:  
最常用的，第一次启动根据实体建立表结构，之后启动会根据实体的改变更新表结构，之前的数据都在。  
validate:  
会验证创建数据库表结构，只会和数据库中的表进行比较，不会创建新表，但是会插入新值，运行程序会校验实体字段与数据库已有的表的字段类型是否相同，不同会报错。

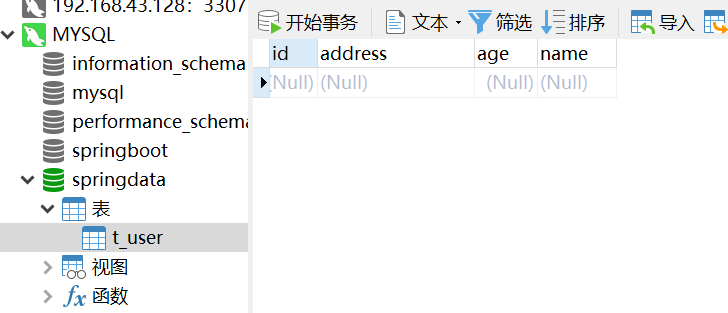
编写实体类：

|  |
| --- |
| import javax.persistence.\*; import java.io.Serializable;  @Entity @Table(name = "t\_user") public class User implements Serializable {   @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  @Column(name = "id")  private Integer id;  @Column(name = "name")  private String name;  @Column(name = "age")  private Integer age;  @Column(name = "address")  private String address;   public Integer getId() {  return id;  }   public void setId(Integer id) {  this.id = id;  }   public String getName() {  return name;  }   public void setName(String name) {  this.name = name;  }   public Integer getAge() {  return age;  }   public void setAge(Integer age) {  this.age = age;  }   public String getAddress() {  return address;  }   public void setAddress(String address) {  this.address = address;  }   @Override  public String toString() {  return "User{" +  "id=" + id +  ", name='" + name + '\'' +  ", age=" + age +  ", address='" + address + '\'' +  '}';  } } |

此时库中是没有表的。



如果这是启动springboot程序，则会生成相应的表：



创建Dao

|  |
| --- |
| @Repository public interface UserRepository extends JpaRepository<User, Integer> { } |

创建测试类：

|  |
| --- |
| import com.icss.dao.UserRepository; import com.icss.domain.User; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;  @RunWith(SpringRunner.class) @SpringBootTest public class TestJpa {   @Autowired  private UserRepository userRepository;   @Test  public void testSave(){  User user = new User();  user.setName("rose");  user.setAge(22);  user.setAddress("天津市西青区");  userRepository.save(user);  }  } |

6.2.5JPA中的接口

6.2.5.1Repository接口

## 6.2整合第三方数据源-阿里Druid

如果想要集成第三方数据源，只需导入依赖数据源依赖即可。

### 6.2.1导入依赖

|  |
| --- |
| *<!-- 导入阿里druid数据源-->* <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.1.8</version>  </dependency> |

### 6.2.2书写配置

Application-druid.yml

|  |
| --- |
| spring:  *#配置连接池* datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://localhost:3306/springboot?serverTimezone=UTC  username: root  password: ROOT  type: com.alibaba.druid.pool.DruidDataSource //配置数据源类型，不写也可以  initialSize: 5  minIdle: 5  maxActive: 20  maxWait: 60000  timeBetweenEvictionRunsMillis: 60000  minEvictableIdleTimeMillis: 300000  validationQuery: SELECT 1 FROM DUAL  testWhileIdle: true  testOnBorrow: false  testOnReturn: false  poolPreparedStatements: true |

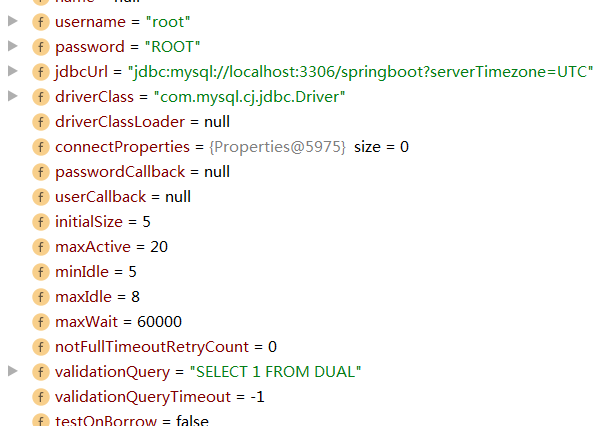
### 6.2.3在主配置中激活druid配置

|  |
| --- |
| spring:  *#激活druid* profiles:  active: druid |

### 6.2.4测试

|  |
| --- |
| @SpringBootTest class Chapter04Crud0716ApplicationTests {  *//注入DataSource* @Autowired  private DataSource dataSource;  @Test  void contextLoads() throws SQLException {  *//数据源的类型* System.*out*.println(dataSource.getClass());  *//获取连接* Connection connection = dataSource.getConnection();  System.*out*.println(connection);  *//关闭连接* connection.close();  } } |
|  |

### 6.2.5debug调试



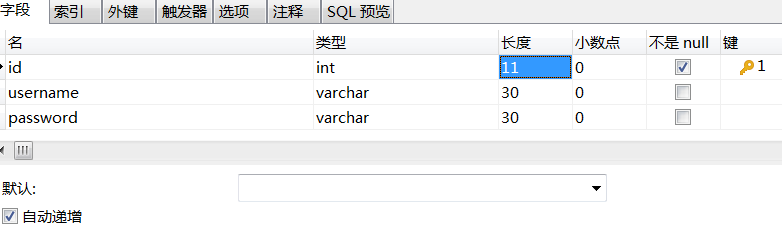
属性都正常注入了。

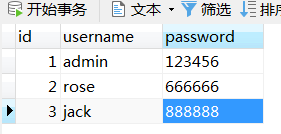
## 6.3整合Mybatis注解开发

### 6.3.1导入依赖

|  |
| --- |
| *<!-- 导入jdbc启动器-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency> *<!-- 导入mybatis启动器-->* <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>2.1.3</version>  </dependency> |

### 6.3.2创建user表





### 6.3.3创建Mapper

|  |
| --- |
| @Mapper public interface UserMapper {   @Select("select \* from user where username=#{username} and password= #{password}")  User selectByUsernamePassword(String username, String password) throws Exception; } |

### 6.3.4创建Service

|  |
| --- |
| @Service public class UserServiceImpl implements UserService {   *//注入mapper* @Autowired  private UserMapper userMapper;   @Override  public User login(String username, String password) throws Exception {  try {  return userMapper.selectByUsernamePassword(username, password);  } catch (Exception e) {  e.printStackTrace();  throw e;  }  } } |

### 6.3.5修改Controller

|  |
| --- |
| @Controller public class LoginController {    *//注入service* @Autowired  private UserService userService;   @RequestMapping("/user/login")  public String login(@RequestParam String username, @RequestParam String password, HttpServletRequest request){  User user = null;  try {  user = userService.login(username, password);  } catch (Exception e) {  e.printStackTrace();  }  if(user != null){  *//登录成功，将user存入session* request.getSession().setAttribute("userLogin", user);  *//跳到main.html* return "redirect:/main.html";  }else{  *//登录失败* request.setAttribute("msg","用户名或密码错误");  *//转发到登录页面* return "login";  }  }  } |

## 6.4基于xml整合mybatis

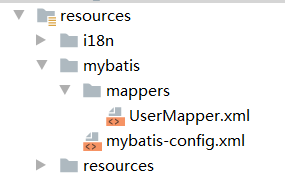
### 6.4.1编写UserMapper

|  |
| --- |
| public interface UserMapper { User selectByUsernamePassword(@Param("username") String username, @Param("password") String password) throws Exception; } |

### 6.4.2在启动类上添加Mapper扫描

|  |
| --- |
| @SpringBootApplication @MapperScan(basePackages = {"com.icss.chapter04crud0716.mapper"}) public class Chapter04Crud0716Application {   public static void main(String[] args) {  SpringApplication.*run*(Chapter04Crud0716Application.class, args);  }  } |

### 6.4.3书写Mybatis核心配置文件



mybatis-config.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>  </configuration> |

### 6.4.4编写mapper映射

UserMapper.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.icss.chapter04crud0716.mapper.UserMapper">   <select id="selectByUsernamePassword" resultType="User">  select *\** from user  where username=#{username} and password=#{password}  </select>  </mapper> |

### 6.4.5编写application-mybatis.yml

|  |
| --- |
| *#配置mybatis* mybatis:  *#mybatis的核心配置文件的路径* config-location: classpath:mybatis/mybatis-config.xml  *#配置mapper映射的路径* mapper-locations: classpath:mybatis/mappers/\*Mapper.xml  *#配置别名扫描* type-aliases-package: com.icss.chapter04crud0716.domain |

### 6.4.6在application.yml中激活mybatis配置

|  |
| --- |
| spring:  *#激活druid,mybatis* profiles:  active: druid,mybatis |