**SpringBoot**

上海千锋 文威

## springboot入门

### 1.1.1 springboot的基本概述

#### 1.1.1.1诞生背景

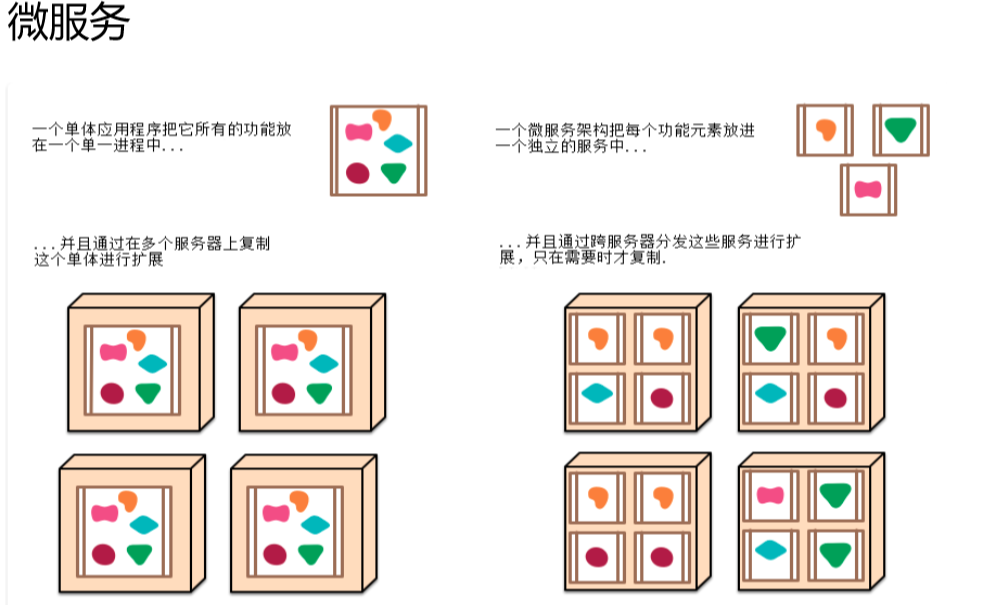
在普通的java项目中，大量的XML文件配置起来是很繁琐就会导致**开发效率低**，整合第三方框架的配置可能存在**冲突问题**导致**部署效率低**，还有其它的问题,传统java项目的打包方式：打包成一个war放入到tomcatwebapps目录下进行执行,也就是说需要依赖外部的tomcat服务器才能执行。

#### 1.1.1.2springboot的优点

* 快速创建独立运行的Spring项目以及与主流框架集成
* 使用嵌入式的Servlet容器，应用无需打成WAR包
* starters自动依赖与版本控制
* 大量的自动配置，简化开发，也可修改默认值
* 无需配置XML，无代码生成，开箱即用
* 准生产环境的运行时应用监控
* 与云计算的天然集成

#### 1.1.1.3微服务

微服务其实是一种架构风格，它提倡我们在开发的时候，一个应用应该是一组小型服务，每一个小服务都运行在自己的进程内，每一个小服务都通过HTTP的方式进行互通。



使用微服务架构，每一个功能元素都是可以独立替换和独立升级的软件单元。

### 1.1.2第一个springboot项目

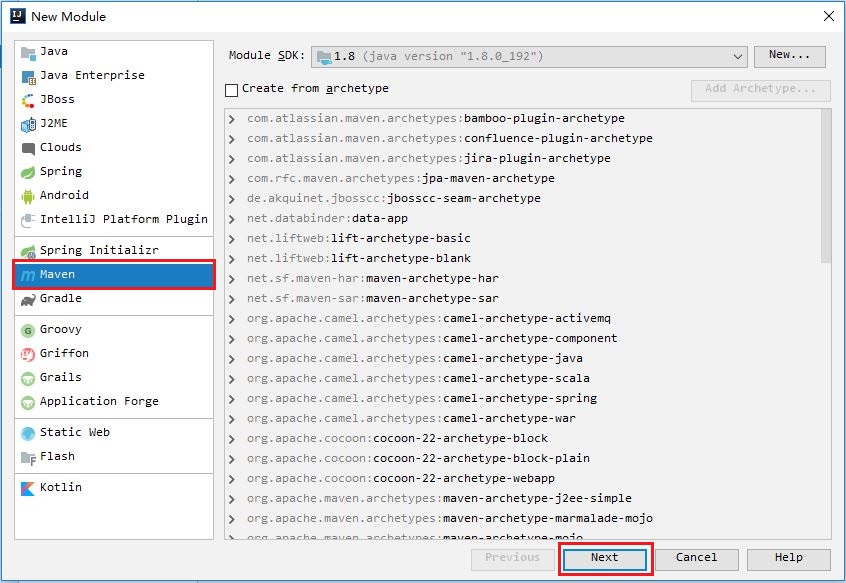
* 入门环境准备：
* jdk1.8
* maven3.x
* IntelliJ IDEA 2018

需求: 浏览器发送hello请求，服务器接收并处理请求，向浏览器页面返回一个helloworld字符串

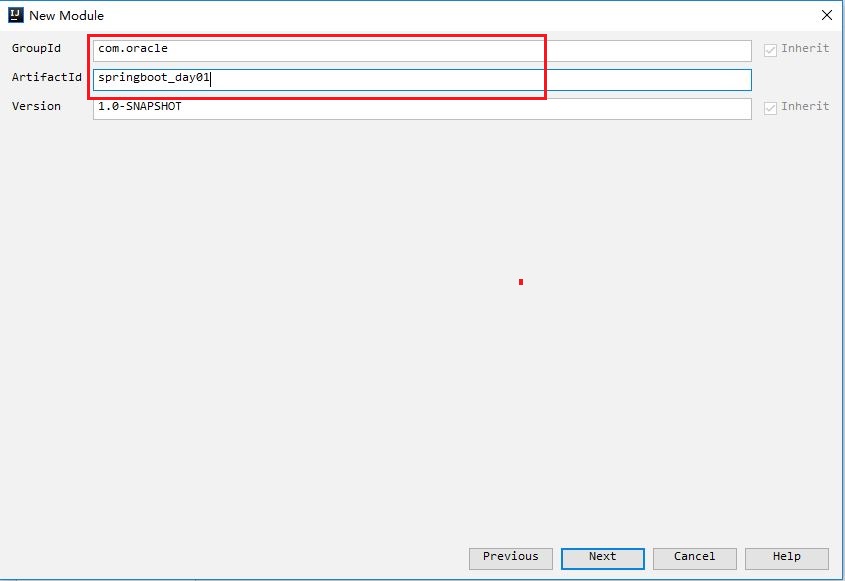
#### 1.1.2.1 创建maven工程

步骤:

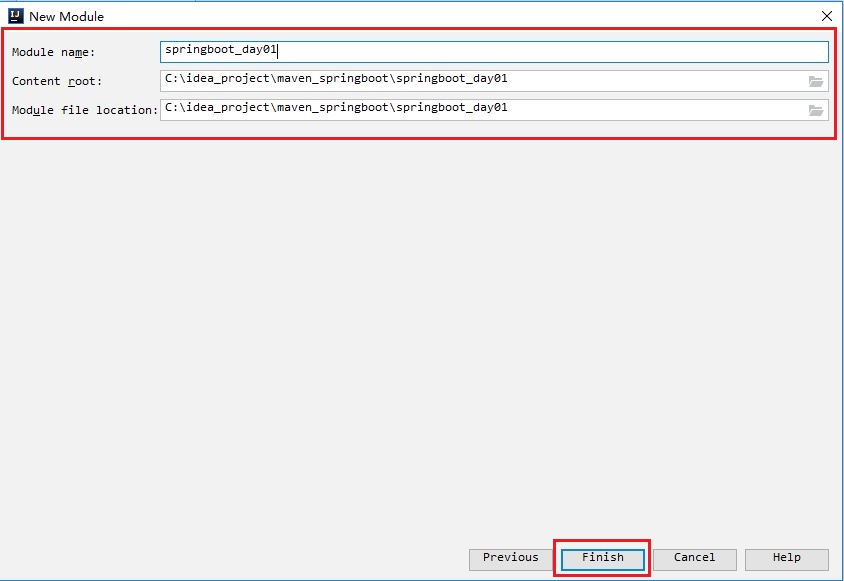
使用idea工具创建一个maven工程，该工程为普通的java工程即可



点击Next



点击Next



点击Finish

#### 1.1.2.2 添加依赖

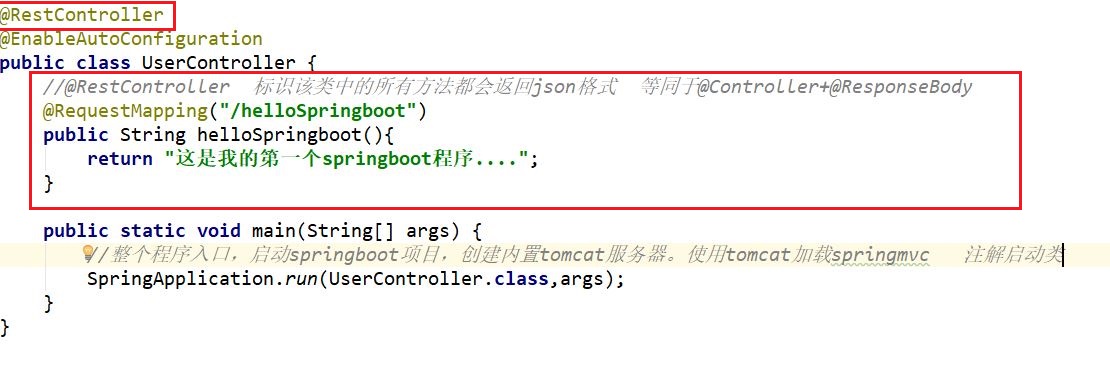
在pom.xml里面导入起步依赖

|  |
| --- |
| *<!--spring-boot-starter-parent整合第三方常用框架的依赖信息-->*  <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.0.1.RELEASE</**version**> </**parent**> |

SpringBoot要集成SpringMVC进行Controller的开发，所以项目要导入web的启动依赖

|  |
| --- |
| *<!--spring-boot-starter-web 是springboot整合springMVC是maven的依赖继承关系 -->*  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**> </**dependencies**> |

#### 1.1.2.3 编写controller



#### 1.2.2.4编写启动类



@EnableAutoConfiguration注解:作用在于让 Spring Boot   根据应用所声明的依赖来对 Spring 框架进行自动配置  
这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于spring-boot-starter-web添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。

**@EnableAutoConfiguration扫描的时候，只能扫描到当前类**

@ComponentScan注解

由于@EnableAutoConfiguration注解只能扫描当前的类，这样对Controller里面的类进行管理很不方便，这个时候，我们可以用@ComponentScan注解来配置扫描包的范围。

我们可以将启动器抽取成一个单独的类

|  |
| --- |
| @EnableAutoConfiguration @ComponentScan(**"com.qf.controller"**) **public class** App {  **public static void** main(String[] args) {  SpringApplication.*run*(App.**class**,args);  } } |

但是使用@ComponentScan进行扫包的时候，包比较多的情况下，写起来比较麻烦。比如我要扫码多个包:

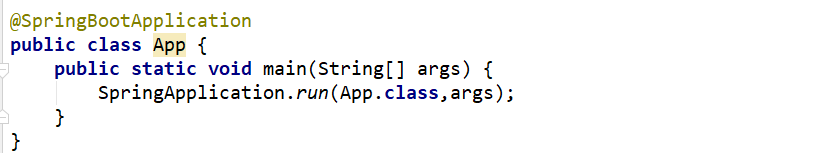
@ComponentScan(basePackages = {**"com.qf.user.controller","com.qf.order.controller"**})

我们可以使用@SpringBootApplication，一劳永逸的解决以上问题

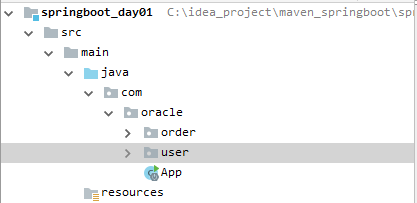
@SpringBootApplication 被 @Configuration、@EnableAutoConfiguration、@ComponentScan 注解所修饰，换言之 Springboot 提供了统一的注解来替代以上三个注解

扫包范围：在启动类上加上@SpringBootApplication注解,当前包下或者子包下所有的类都可以扫到。

#### 1.2.2.5 使用@SpringBootApplication注解编写启动类



但是此时需要注意App类所在的包中的位置



疑问: 如果把App 放置在order包中或者放置在user包中可以吗?

### springboot项目中的细节

#### 1.1.3.1 父项目

|  |
| --- |
| *<!--spring-boot-starter-parent整合第三方常用框架的依赖信息-->* <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.0.1.RELEASE</**version**> </**parent**>  他的父项目是  <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-dependencies</**artifactId**>  <**version**>2.0.1.RELEASE</**version**>  <**relativePath**>../../spring-boot-dependencies</**relativePath**> </**parent**>  它真正管理springboot项目里面的所有依赖版本，**以后我们导入依赖默认是不需要写版本；**（没有在dependencies里面管理的依赖自然需要声明版本号） |

#### 1.1.3.2 启动器

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

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

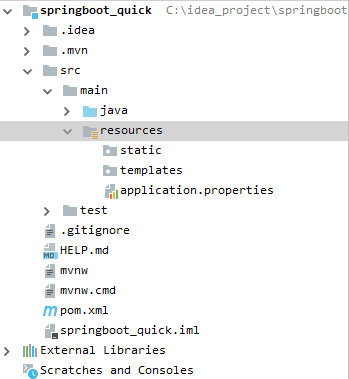
Spring Boot将所有的功能场景都抽取出来，做成一个个的starters（启动器），只需要在项目里面引入这些starter 相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动

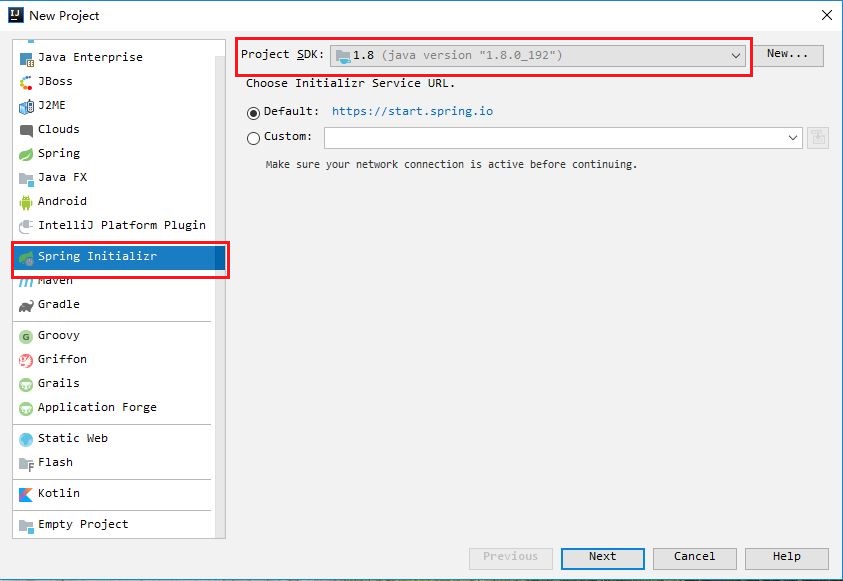
### 1.1.4使用Spring Initializr快速创建springboot项目

IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目； 选择我们需要的模块；向导会联网创建Spring Boot项目； 默认生成的Spring Boot项目；

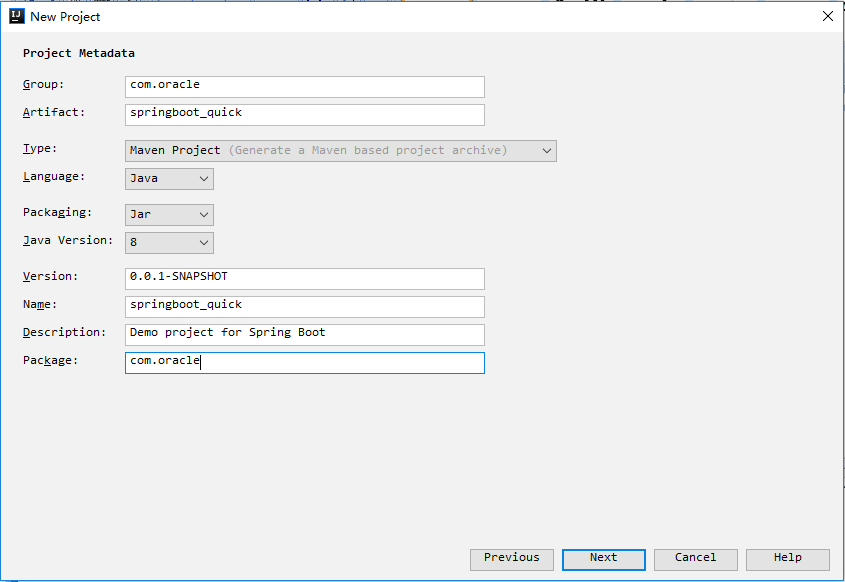
主程序已经生成好了，我们只需要我们自己的逻辑 resources文件夹中目录结构 static：保存所有的静态资源； js css images； templates：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页 面）；可以使用模板引擎（freemarker、thymeleaf）； application.properties：Spring Boot应用的配置文件；可以修改一些默认设置

生成的结构目录如下:

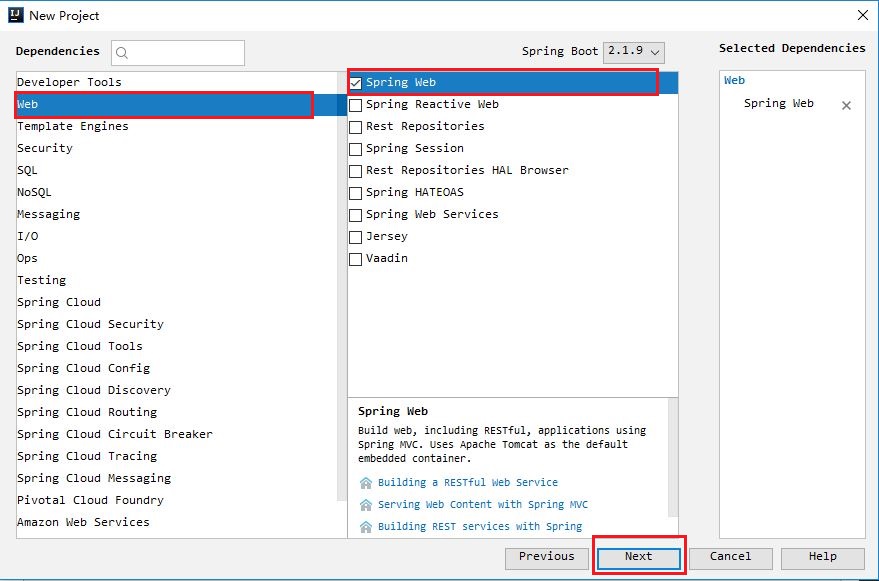




点击Next



点击Next,选中需要的启动器



点击Next完成即可。

### 1.1.5springboot的配置

SpringBoot使用一个**全局的配置文件**，配置文件名是固定的；

•application.properties

•application.yml

YAML：配置例子

|  |
| --- |
| **server**:  **port**: 8081 |

#### 1.1.5.1 YAML的基本语法

k:(空格)v：表示一对键值对（**空格必须有**）；

以缩进来控制层级关系；只要是左对齐的一列数据，都是同一个层级的

属性和值也是大小写敏感；

|  |
| --- |
| **server**:  **port**: 8888 *# 设置端口号* **servlet**:  **path**: /day01 *# 配置虚拟路径* |

此时port和path就是在同一个层级

#### 1.1.5.2 值的写法

* 字面量：普通的值(字符串，数字，布尔值)

k: v：字面直接来写；

字符串默认不用加上单引号或者双引号；

""：双引号；会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思

name: "zhangsan \n lisi"：输出；zhangsan 换行 lisi

''：单引号；不会转义特殊字符，特殊字符终只是一个普通的字符串数据

name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi

* 对象

k: v：在下一行来写对象的属性和值的关系；注意缩进

对象还是k: v的方式

|  |
| --- |
| **animal**:  **name**: Sunny  **age**: 12 |

行内写法:

|  |
| --- |
| **friend**: {**fname**: Oscar,**age**: 20} |

* 数组(List Set)

用- 值表示数组中的一个元素

|  |
| --- |
| **pets**:  - cat  - dog  - fish |

行内写法:

|  |
| --- |
| **names**: [cat,dog,fish] |

#### 1.1.5.3 实例：将JavaBean注入到配置文件中

* 分别提供实体类 Person Dog

Person:

|  |
| --- |
| **public class** Person {  **private** String **lastName**;  **private** Integer **age**;  **private** Boolean **boss**;  **private** Date **birth**;  **private** Map<String,Object> **maps**;  **private** List<Object> **lists**;  **private** Dog **dog**;  //提供 Get Set方法 toString方法  } |

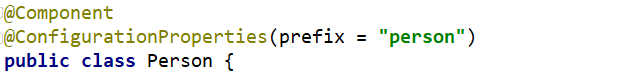
Dog:

|  |
| --- |
| **public class** Dog {  **private** String **name**;  **private** Integer **age**;  //提供 Get Set方法 toString方法  } |

* 在application.ymal中注入属性

|  |
| --- |
| **person**:  **lastName**: Oscar  **age**: 19  **boss**: **false  birth**: 2019/10/10  **maps**: {**k1**: v1,**k2**: v2}  **lists**:  - eric  - kobe  **dog**:  **name**: 大黄  **age**: 2 |

* 在注入的Person的类上面添加注解



注:

@ConfigurationProperties:告诉springboot 将当前类的所有属性和配置文件中的相关信息进行绑定  
\* prefix：配置文件中的哪个属性进行一一映射

@Component：只有将这个类配置在Spring容器中 才能使用@ConfigurationProperties的功能

使用@Autowired即可访问该对象

* 以导入配置文件处理器，以后编写配置就有提示了

|  |
| --- |
| *<!--配置文件处理器-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**> </**dependency**> |

* 测试。

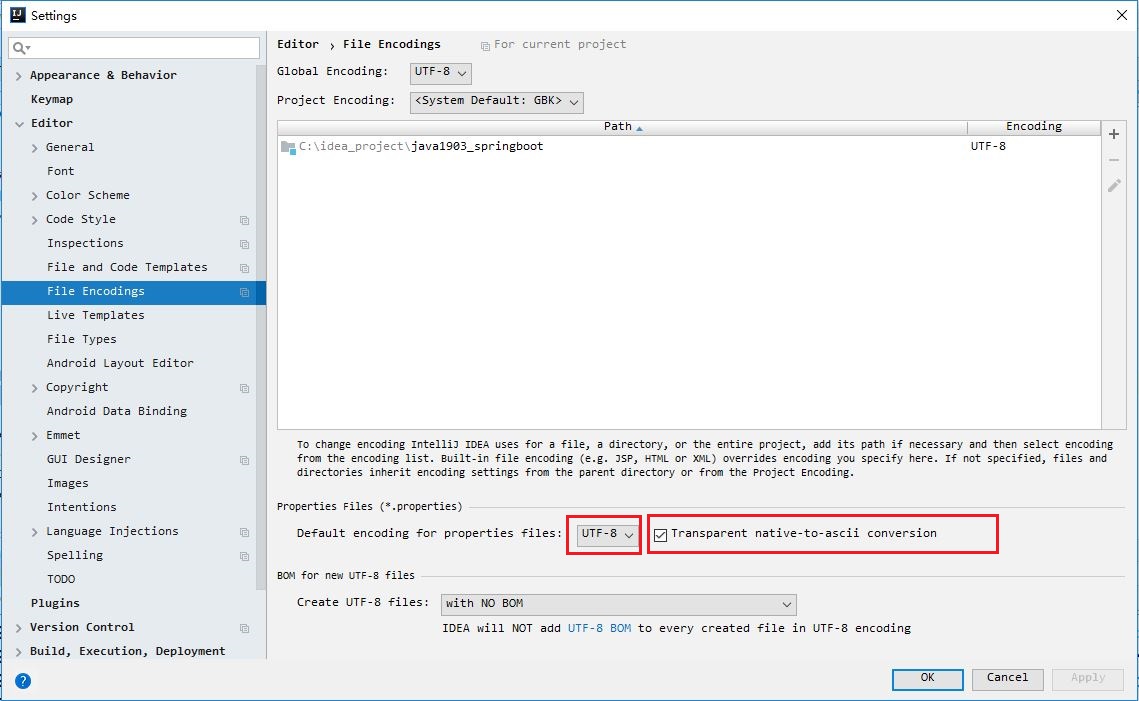
|  |
| --- |
| @AutoWired Person person;  @RequestMapping(**"/hello"**) **public** String hello(){  System.***out***.println(**person**);  **return "这是我的第一个springboot程序"**; } |

#### 1.1.5.4 Properties文件的配置

在application.properties文件中配置

|  |
| --- |
| **server.port**=**8081 person.last-name**=**张三 person.age**=**12 person.birth**=**2019/12/30 person.boss**=**false person.lists**=**a,b,c person.maps.k1**=**value1 person.maps.k2**=**value2 person.dog.name**=**大黄 person.dog.age**=**1** |

解决在application.properties文件中的中文乱码问题:



#### 1.1.5.5 @Value获取值和@ConﬁgurationProperties获取值比校

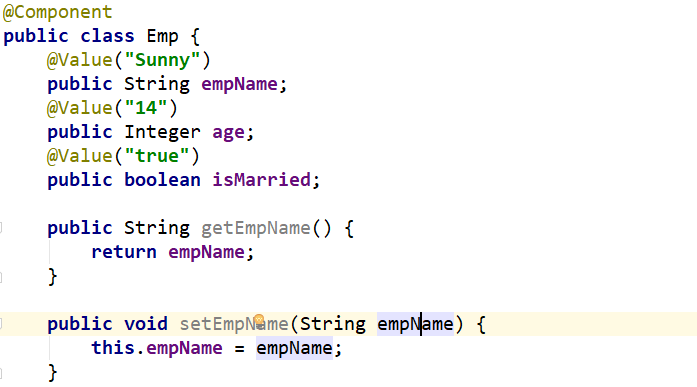
在spring中@Value相当于在配置文件中的

<bean id=” ” class=” ”>

<property name=” ” value=” ”></ property>

</bean>

* @Value注解支持字面量的注解方式

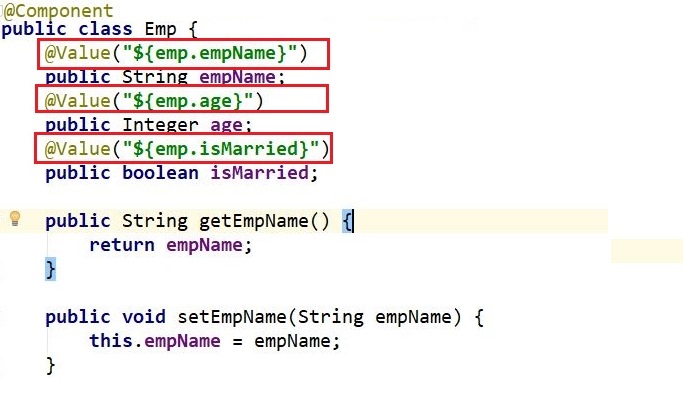


* 从配置文件中获取

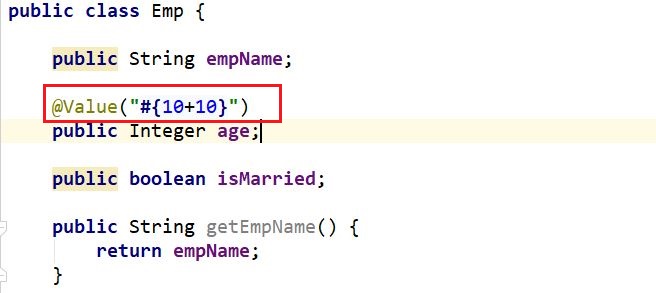
application.yml配置文件中定义:

|  |
| --- |
| **emp**:  **empName**: kobe  **age**: 19  **isMarried**: true |

Emp中使用@Value注解中获取属性



* 使用SPEL表达式获取



通过比较@ConﬁgurationProperties和@Value的区别：



最大的区别就是@ConﬁgurationProperties支持数据校验，而@Value则不支持。



在application.yml文件中 email的值只能是邮箱类型，否则启动报错。



总结:

配置文件yml还是properties他们都能获取到值；

如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value； 如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConﬁgurationProperties；

#### 1.1.5.6 @PropertySource

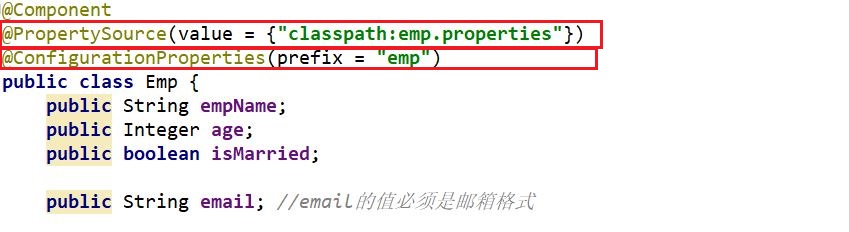
在前面的配置中，使用@ConﬁgurationProperties注解可以获取配置文件中的属性值。**但是@ConﬁgurationProperties读取的配置文件是全局的，**也就是说这个注解只能加载全局的配置文件(application.yml/application.properties).但是全局配置文件一般定义的都是有关spring配置的信息。如果我们需要定义一些与spring配置无关的信息，那么该如何呢?

我们可以定义局部的配置文件，然后使用@PropertySource注解进行获取。

* 定义emp.properties配置文件

|  |
| --- |
| **emp.empName**=**kobe emp.age**=**19 emp.email**=**risswen@sina.com emp.isMarried**=**true** |

* 定义实体类Emp



#### 1.1.5.7 @ImportResource

导入Spring的配置文件，让配置文件里面的内容生效；

**Spring Boot里面没有Spring的配置文件，我们自己编写的配置文件，也不能自动识别**； 想让Spring的配置文件生效，加载进来；@ImportResource标注在一个配置类上

实例：

定义一个UserSercice方法，使用xml配置文件的方式管理UserService类。如何在springboot项目中获取?

* 定义UserService

|  |
| --- |
| **public class** UserService {  **public void** test(){  System.***out***.println(**"这是UserService里面的方法"**);  } } |

* 定义bean.xml配置文件，管理bean

|  |
| --- |
| <**bean id="userService" class="com.qf.service.UserService"**></**bean**> |

* 在启动类上添加@importResource注解

|  |
| --- |
| @SpringBootApplication **@ImportResource(locations = {"classpath:bean.xml"})**//加载类路径下面的配置文件 **public class** App {  **public static void** main(String[] args) {  SpringApplication.*run*(App.**class**,args);  } } |

* 测试

|  |
| --- |
| @RestController **public class** controller {  @Autowired  ApplicationContext **applicationContext**;  @RequestMapping(**"/hello"**)  **public** String hello(){  UserService userService = (UserService) **applicationContext**.getBean(**"userService"**);  userService.test();  **return "这是我的第一个springboot程序"**;  } } |

**注意！！！**:在springboot项目中，我们在spring容器中添加组件，我们一般不用以上使用xml的方式。Springboot推荐我们使用**全注解**的方式在spring容器添加组件。

1、配置类@Conﬁguration------>Spring配置文件

2、使用@Bean给容器中添加组件

* 定义OrderService

|  |
| --- |
| **public class** OrderService {  **public void** test(){  System.***out***.println(**"这是OrderService里面的方法"**);  } } |

* 定义配置类

|  |
| --- |
| **package** com.qf.config; **import** com.qf.service.OrderService; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; @Configuration ***/\*\*  \* 指明当前类就是一个配置类，用来替代之前的配置文件<bean class=""></bean>  \*/*****public class** MyAppConfig {   ***//将方法的返回值添加到spring容器中，组件的默认id就是当前方法名*** @Bean  **public** OrderService orderService(){  **return new** OrderService();  } } |

* 测试调用

|  |
| --- |
| @RestController **public class** controller {  @Autowired  ApplicationContext **applicationContext**;   @RequestMapping(**"/hello"**)  **public** String hello(){  OrderService userService = (OrderService) **applicationContext**.getBean(**"orderService"**);  userService.test();  **return "这是我的第一个springboot程序"**;  } } |

#### 1.1.5.8 配置文件的占位符

* 随机数

|  |
| --- |
| ${**random.uuid**}  ${**random.int(18,60)**}  ${**random.value**}  …… |

|  |
| --- |
| **randomdemo**: #名字不能以驼峰命名 否则启动报错  **id**: ${**random.uuid**} 取随机的uuid  **age**: ${**random.int(18,60)**} 取指定范围的随机数  **address**: ${**random.value**} 获取随机字符串 |

* 默认值

|  |
| --- |
| **randomdemo**:  **id**: ${**random.uuid**}  **age**: ${**random.int(18,60)**}  **address**: ${**random.value**}  **name**: ${**:铁蛋**} 如果不写 默认值就是铁蛋 |

#### 1.1.5.9 springboot配置文件的加载位置

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

–ﬁle:./conﬁg/ 项目根目录下面的config文件夹 优先级最高

–ﬁle:./ 项目根目录下面

–classpath:/conﬁg/ resources文件夹下面的config文件夹

–classpath:/ resources文件夹下面 优先级最低

优先级由**高到底**，高优先级的配置会覆盖低优先级的配置；

SpringBoot会从这四个位置全部加载主配置文件；互补配置；

测试: 在以上各级文件夹下面定义application.properties文件，里面定义server.port端口号。看看到底以哪一个端口号为准。

小细节: 如果需要给访问路径加上虚拟目录，那么在配置文件中可以配置:

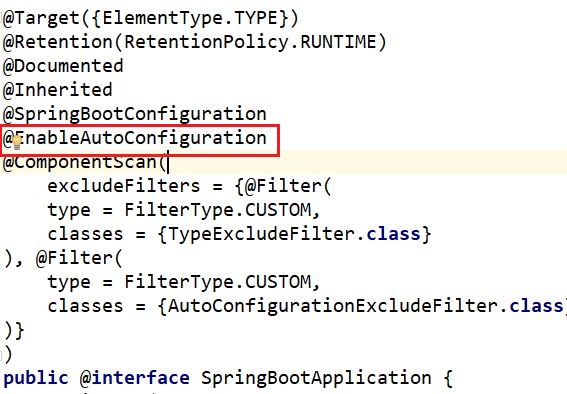
|  |
| --- |
| **server**:  **port**: 8081  **servlet:  context-path: /springboot *#指定虚拟路径*** |

### 1.1.6 springboot的自动配置原理

(1) SpringBoot启动的时候加载主配置类,开启了自动配置功能 @EnableAutoConﬁguration

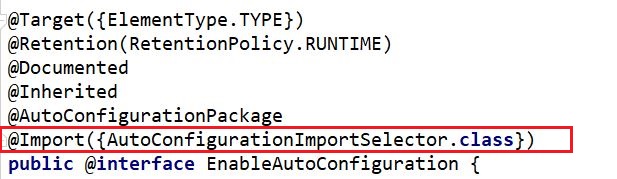
|  |
| --- |
| **@SpringBootApplication** @ImportResource(locations = {**"classpath:bean.xml"**}) **public class** App {  **public static void** main(String[] args) {  SpringApplication.*run*(App.**class**,args);  } } |

(2)点击进入@SpringBootApplication注解的源码



@EnableAutoConfiguration注解的作用?

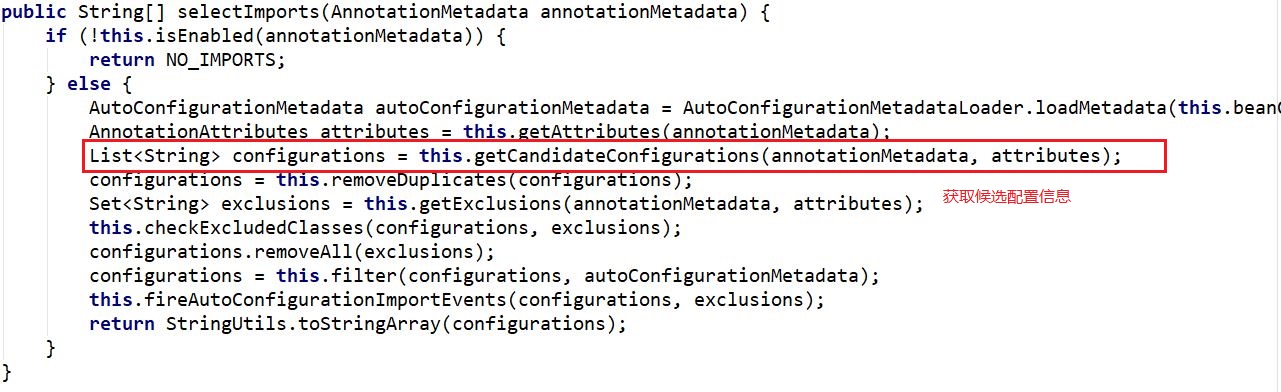
(3)点击进入@EnableAutoConfiguration注解的源码



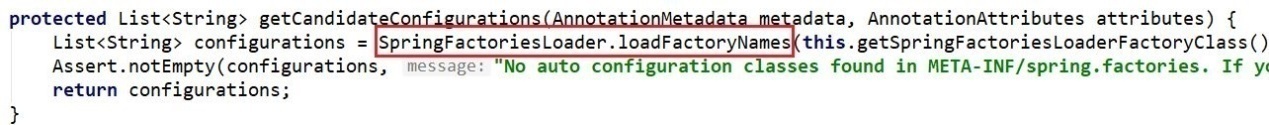
利用EnableAutoConﬁgurationImportSelector给容器中导入一些组件

(4)点击进入AutoConfigurationImportSelector这个类里面

这个类里面有一个selectImports方法



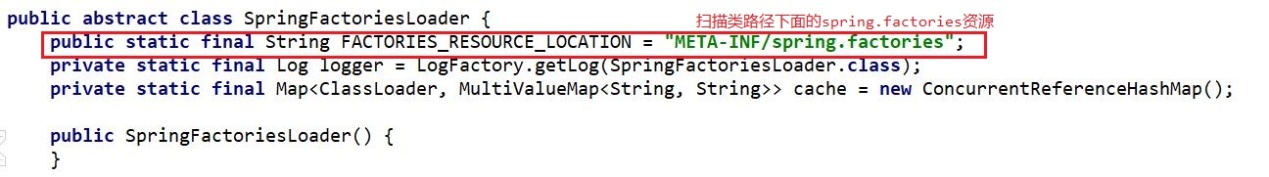
(5)点击进入getCandidateConfigurations方法



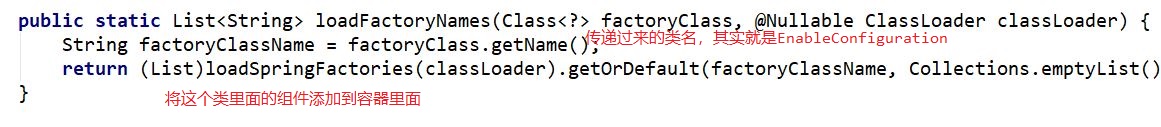
(6)点击进入getSpringFactoriesLoaderFactoryClass()



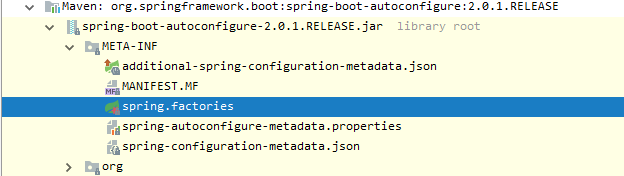
(7)点击进入loadFcatoryNames方法，进入SpringFactoriesLoader类



(8)查看SpringFactoriesLoader里面的loadFactoryNames方法



(9)加载的组件





### 1.1.7 springboot与日志

#### 1.1.7.1 springboot整合log4j日志记录

* 在resources目录下面创建日志文件，并引入：

|  |
| --- |
| *#log4j.rootLogger=CONSOLE,info,error,DEBUG* **log4j.rootLogger**=**info,error,CONSOLE,DEBUG log4j.appender.CONSOLE**=**org.apache.log4j.ConsoleAppender  log4j.appender.CONSOLE.layout**=**org.apache.log4j.PatternLayout  log4j.appender.CONSOLE.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.logger.info**=**info log4j.appender.info**=**org.apache.log4j.DailyRollingFileAppender log4j.appender.info.layout**=**org.apache.log4j.PatternLayout  log4j.appender.info.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.appender.info.datePattern**=**'.'yyyy-MM-dd log4j.appender.info.Threshold** = **info  log4j.appender.info.append**=**true** *#log4j.appender.info.File=/home/admin/pms-api-services/logs/info/api\_services\_info* **log4j.appender.info.File**=**/Users/dddd/Documents/testspace/pms-api-services/logs/info/api\_services\_info log4j.logger.error**=**error  log4j.appender.error**=**org.apache.log4j.DailyRollingFileAppender log4j.appender.error.layout**=**org.apache.log4j.PatternLayout  log4j.appender.error.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.appender.error.datePattern**=**'.'yyyy-MM-dd log4j.appender.error.Threshold** = **error  log4j.appender.error.append**=**true** *#log4j.appender.error.File=/home/admin/pms-api-services/logs/error/api\_services\_error* **log4j.appender.error.File**=**/Users/dddd/Documents/testspace/pms-api-services/logs/error/api\_services\_error log4j.logger.DEBUG**=**DEBUG log4j.appender.DEBUG**=**org.apache.log4j.DailyRollingFileAppender log4j.appender.DEBUG.layout**=**org.apache.log4j.PatternLayout  log4j.appender.DEBUG.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.appender.DEBUG.datePattern**=**'.'yyyy-MM-dd log4j.appender.DEBUG.Threshold** = **DEBUG  log4j.appender.DEBUG.append**=**true** *#log4j.appender.DEBUG.File=/home/admin/pms-api-services/logs/debug/api\_services\_debug* **log4j.appender.DEBUG.File**=**/Users/dddd/Documents/testspace/pms-api-services/logs/debug/api\_services\_debug  log4j\u4EE3\u7801 private static final Logger logger = LoggerFactory.getLogger(IndexController.class);** |

* 引入log4j依赖

|  |
| --- |
| *<!-- springboot-log4j -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j</**artifactId**>  <**version**>1.3.8.RELEASE</**version**> </**dependency**> |

* Controller

|  |
| --- |
| @RestController **public class** LoggController {  **private static final Logger *logger* = LoggerFactory.*getLogger*(LoggController.class);**  @RequestMapping(**"/printLog"**)  **public** String printLog(){  ***logger***.info(**"日志打印输出了......"**);  **return "Hello World....."**;  } } |

#### 1.1.7.2使用Aop统一处理Web请求日志

* 导入依赖

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

* Aop处理日志的类

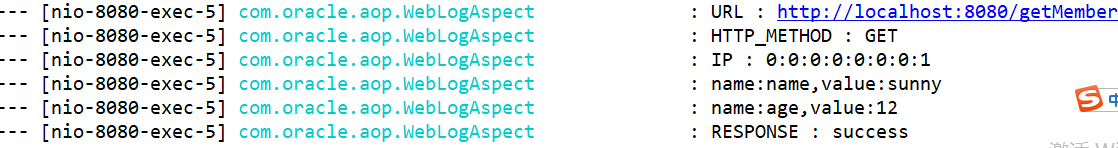
|  |
| --- |
| **package** com.qf.aop; **import** org.aspectj.lang.JoinPoint; **import** org.aspectj.lang.annotation.AfterReturning; **import** org.aspectj.lang.annotation.Aspect; **import** org.aspectj.lang.annotation.Before; **import** org.aspectj.lang.annotation.Pointcut; **import** org.slf4j.Logger; **import** org.slf4j.LoggerFactory; **import** org.springframework.stereotype.Component; **import** org.springframework.web.context.request.RequestContextHolder; **import** org.springframework.web.context.request.ServletRequestAttributes; **import** javax.servlet.http.HttpServletRequest; **import** java.util.Enumeration; @Aspect @Component **public class** WebLogAspect {  **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(WebLogAspect.**class**);  @Pointcut(**"execution(public \* com.qf.order.controller.\*.\*(..))"**)  **public void** webLog() {  }  @Before(**"webLog()"**)  **public void** doBefore(JoinPoint joinPoint) **throws** Throwable {  *// 接收到请求，记录请求内容* ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  HttpServletRequest request = attributes.getRequest();  *// 记录下请求内容* ***logger***.info(**"URL : "** + request.getRequestURL().toString());  ***logger***.info(**"HTTP\_METHOD : "** + request.getMethod());  ***logger***.info(**"IP : "** + request.getRemoteAddr());  Enumeration<String> enu = request.getParameterNames();  **while** (enu.hasMoreElements()) {  String name = (String) enu.nextElement();  ***logger***.info(**"name:{},value:{}"**, name, request.getParameter(name));  }  }  @AfterReturning(returning = **"ret"**, pointcut = **"webLog()"**)  **public void** doAfterReturning(Object ret) **throws** Throwable {  *// 处理完请求，返回内容* ***logger***.info(**"RESPONSE : "** + ret);  }  } |

* 控制类

|  |
| --- |
| @RequestMapping(**"/getMember"**) **public** String getMember(String name,Integer age){  **return "success"**; } |

请求路径: <http://localhost:8080/getMember?name=sunny&age=12>

控制台输出打印结果:

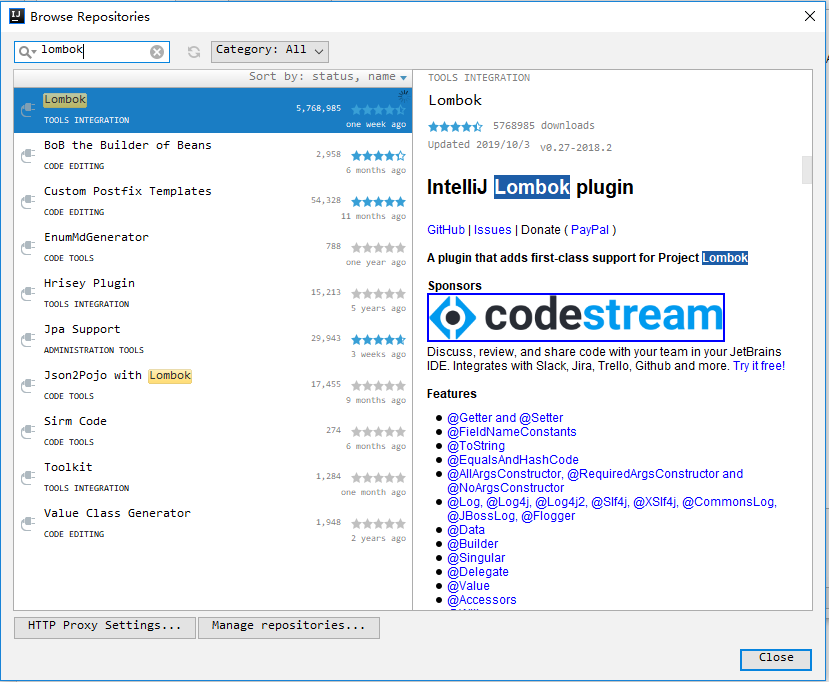


#### 1.1.7.3 lombok插件的使用

* 导入依赖

|  |
| --- |
| *<!--lombok插件-->* <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**> </**dependency**> |

* 下载lombok插件



* 使用lombok插件的注解

|  |
| --- |
| @Slf4j *// 可以直接使用log打印输出 不需要创建对象* @Data *//集成了Get Set toString的功能* **public class** Emp {  String **username**;  String **password**;  **public static void** main(String[] args) {  Emp emp = **new** Emp();  emp.setUsername(**"sunny"**);  emp.setPassword(**"123"**);  ***log***.info(emp.toString());  } } |

## 2.1 springboot web开发

使用SpringBoot；

**1）、创建SpringBoot应用，选中我们需要的模块；**

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

**3）、自己编写业务代码；**

### 2.1.1 springboot对静态资源的映射规则

* "/\*\*" 访问当前项目的任何资源，都去（静态资源的文件夹）找映射

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

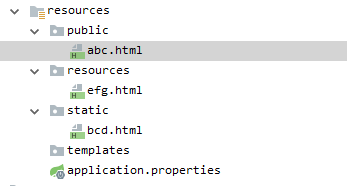
**"classpath:/resources/",**

**"classpath:/static/",**

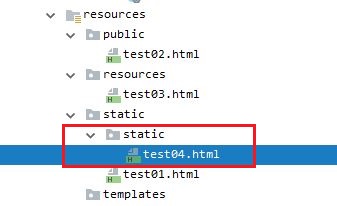
**"classpath:/public/"**

"/"：当前项目的根路径

测试效果:



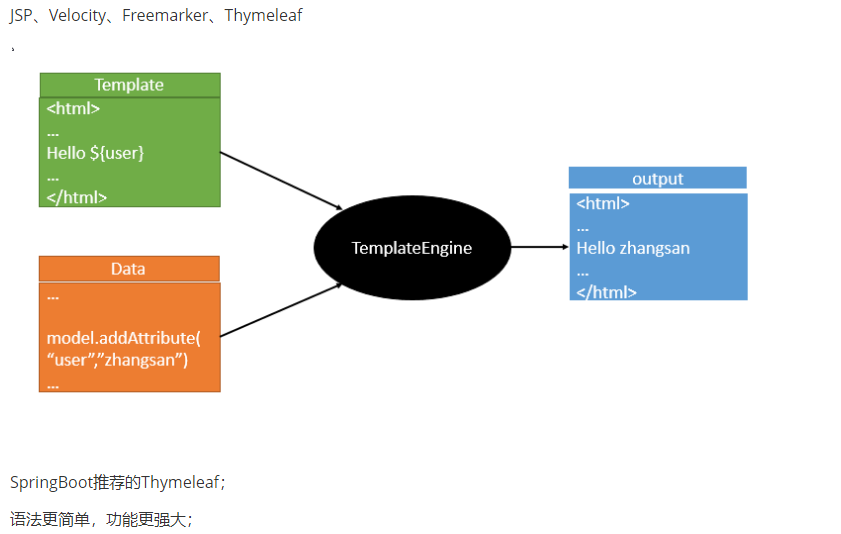
如果要实现[http://localhost:8081/springboot\_day03/**static**/test04.html](http://localhost:8081/springboot_day03/static/test04.html)



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

localhost:8080/ 找index.html页面

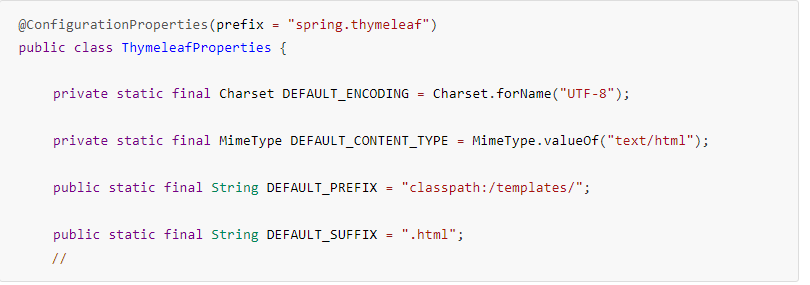
### 2.1.2 模板引擎



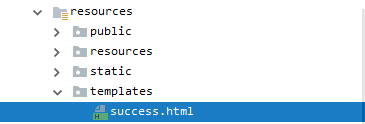
#### 2.1.2.1 引入Thymeleaf依赖

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

#### 2.1.2.2 Thymeleaf使用



只要我们把HTML页面放在classpath:/**templates**/，thymeleaf就能自动渲染；



编写控制器类:

|  |
| --- |
| @Controller **public class** HelloController {  @RequestMapping(**"/hello"**)  **public** String success(){  **return "success"**;//跳转到templates下面的success.html页面  } } |

使用<http://localhost:8080/hello>即可访问

#### 2.1.2.3 Thymeleaf语法初体验

需求：通过控制器实现页面跳转，并将数据传递给前台html页面

* 在HTML页面导入thymeleaf的名称空间

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

* 编写控制器类

|  |
| --- |
| @Controller **public class** HelloController {  @RequestMapping(**"/hello"**)  **public** String success(**Map<String,Object> map**){   **map.put("name","Sunny");**  **return "success"**;  } } |

* 页面取值

|  |
| --- |
| <**body**>  <**h3**>这是一个成功页面</**h3**>  欢迎你,<**span th:text="${name}"**></**span**> </**body**> |

#### 2.1.2.4 thyemleaf的语法规则

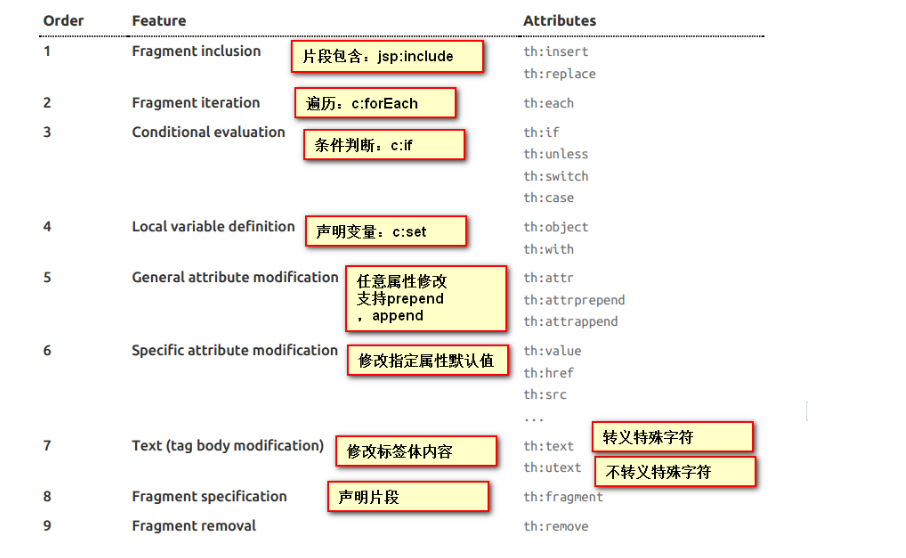
* th语法

th:text；改变当前元素里面的文本内容；

th:任意html属性；来替换原生属性的值

|  |
| --- |
| <**body**>  <**h3**>这是一个成功页面</**h3**>  欢迎你,<**span th:text="${name}" id="s1" class="span1" th:id="${name}" th:class="${name}"**></**span**> </**body**> |

以上的th:id、th:class的值就替换了原来的id和class属性的值。



* 表达式语法

具体可以参照官方文档

${}

\*{}

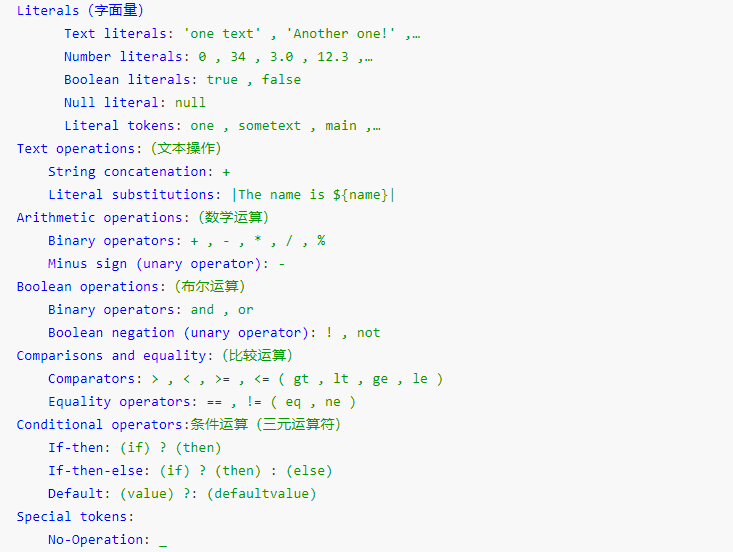
#{}

@{}

~{}







* 小实验

控制器:

|  |
| --- |
| @RequestMapping(**"/hello"**) **public** String success(Map<String,Object> map){  map.put(**"name"**,**"Oscar"**);  map.put(**"content"**,**"<h2>大家好</h2>"**);  List<String> list = **new** ArrayList<>();  list.add(**"Sunny"**);  list.add(**"Kobe"**);  list.add(**"Ketty"**);  map.put(**"list"**,list);  **return "success"**; } |

HTML页面展示

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Title</**title**> </**head**> <**body**> <**h3**>这是一个成功页面</**h3**> 欢迎你,<**span th:text="${name}" id="s1" class="span1" th:id="${name}" th:class="${name}"**></**span**> <**hr**> *<!--循环展示-->* <**h2**>  <**div th:each="list:${list}" th:text="${list}"**></**div**> </**h2**> *<!--th:utext-->* <**div th:utext="${content}"**></**div**> *<!--转义-->* <**div th:text="${content}"**></**div**> *<!--不转义--> <!--循环展示的另一种写法-->* <**div id="box" th:each="names:${list}"**>[[${names}]]</**div**> </**body**> </**html**> |

#### 2.1.3 springboot中springMVC

* Springmvc的扩展配置

在springboot中，提供了很多springMVC的自动配置，但是这些自动配置还不满足我们的开发需求，所以我们可以在默认的配置上进行扩展。

如何扩展?

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

|  |
| --- |
| @Configuration **public class** MyWebConfig **implements** WebMvcConfigurer {  */\*\*  \* 自定义跳转视图  \** ***@param registry*** *\*/* @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/login"**).setViewName(**"success"**);  registry.addViewController(**"/admin"**).setViewName(**"success"**);  } } |

以上代码：指定了我们访问路径跳转到的执行页面。

* 全面接管springMVC的配置

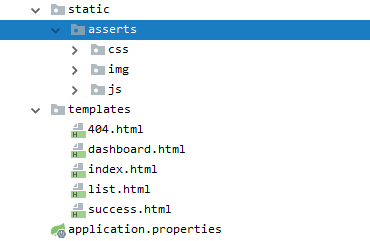
SpringBoot对SpringMVC的自动配置不需要了，所有都是我们自己配置;所有的SpringMVC的自动配置都失效了**我们需要在配置类中添加@EnableWebMvc即可。**



测试:这样我们直接访问项目的静态页面就会访问不到

### 2.1.3 springBoot实现增删改查的操作

* 前期准备:将静态资源导入到项目中

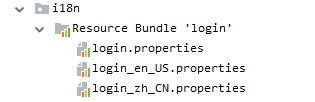


* 默认访问首页的配置

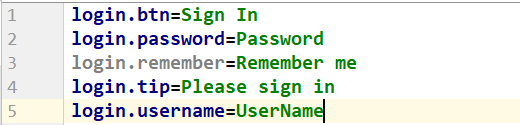
|  |
| --- |
| @Configuration *//@EnableWebMvc //相当于导入DelegatingWebMvcConfiguration.class 从容器中获取所有webmcvconfig* **public class** MyMvcConfig **implements** WebMvcConfigurer {  *// 视图跳转* @Override  **public void** addViewControllers(ViewControllerRegistry registry) {   **registry.addViewController("/").setViewName("index");  registry.addViewController("/index.html").setViewName("index");  registry.addViewController("/main.html").setViewName("dashboard");**  }   @Override  **public void** addInterceptors(InterceptorRegistry registry) { *// 添加拦截器* registry.addInterceptor(**new** LoginHandleInterceptor()).addPathPatterns(**"/\*\*"**).excludePathPatterns(**"/index.html"**, **"/"**, **"/user/login"**, **"/css/\*\*"**, **"/js/\*\*"**, **"/img/\*\*"**);  } } |

#### 2.1.3.1 标签国际化操作

* 编写配置文件



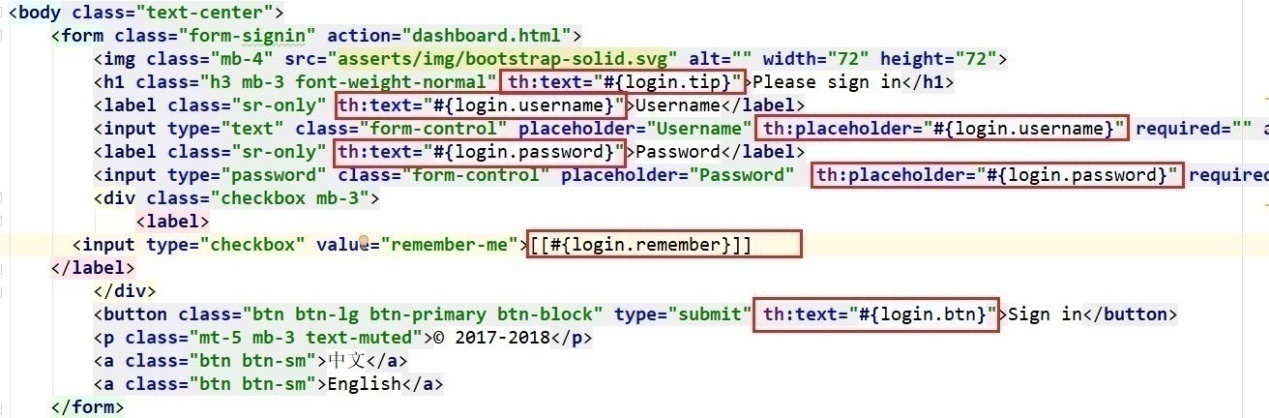




* 在application.properties文件中引入login.properties文件

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

* 在login.html使用国际化标签取值



#### 2.1.3.2 登录功能

* 修改login.html表单中的表单提交地址

|  |
| --- |
| <**form class="form-signin" action="dashboard.html"** th:action="@{/user/login}" method="post"> |

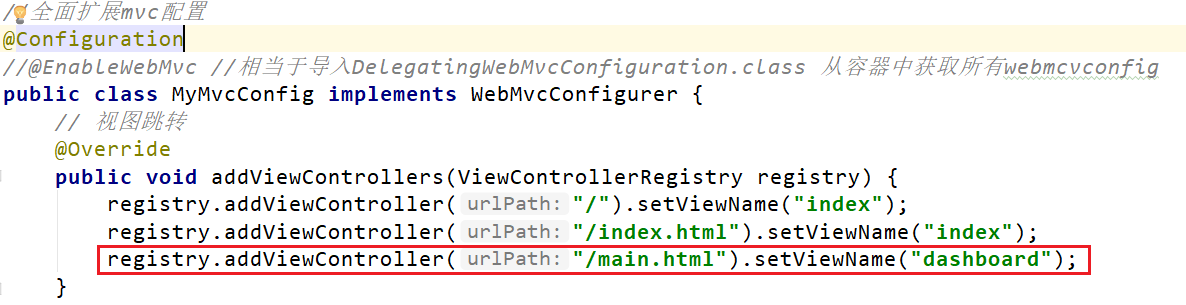
* 禁用浏览器缓存

|  |
| --- |
| **spring.thymeleaf.cache**=**false** |

* 编写登录的controller

|  |
| --- |
| @Controller **public class** LoginController {  *//@RequestMapping(value = "/user/login",method = RequestMethod.POST)* @PostMapping(value = **"/user/login"**)  **public** String login(@RequestParam(**"username"**) String username,  @RequestParam(**"password"**) String password, HttpSession session, Map<String, Object> map){  **if**(!StringUtils.*isEmpty*(username)&&password.equals(**"123456"**)){  *//登录成功,防止表单重复提交，可以重定向到主页* session.setAttribute(**"loginUser"**,username);  **return "redirect:/main.html"**;  }**else**{  *//登陆失败* map.put(**"msg"**,**"用户名或密码错误"**);  **return "index"**;  }   } } |

* 添加视图映射 /main.html的视图映射



* login.html添加登录失败的提示信息

|  |
| --- |
| <**h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}"**>Please sign in</**h1**> *<!--判断-->* **<p style="color: red" th:text="${msg}" th:if="${not #strings.isEmpty(msg)}"></p>** |

以上代码存在的问题:

<http://localhost:8080/main.html> 可以绕过登录直接访问主页。

解决方案： 配置拦截器

* 编写拦截器

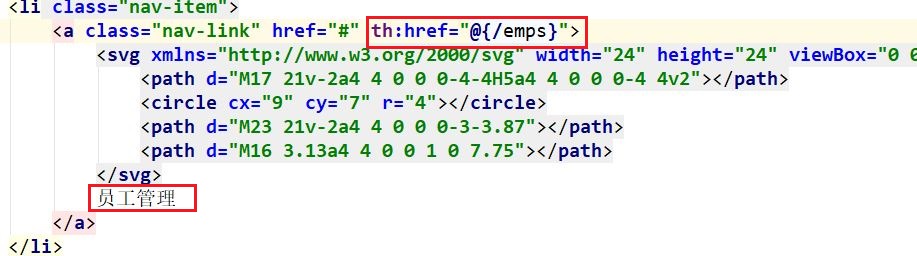
|  |
| --- |
| **public class** LoginHandleInterceptor **implements** HandlerInterceptor {  *// 登录前拦截 true 放行* @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  *// 登录成功存session* HttpSession session = request.getSession();  Object loginuser = session.getAttribute(**"loginuser"**);  **if** (loginuser == **null**) {  *// 未登录* request.setAttribute(**"msg"**, **"没有权限，请先登录"**);  request.getRequestDispatcher(**"/index.html"**).forward(request, response);  **return false**;  } **else** {  **return true**;  }  } } |

* 注册拦截器

|  |
| --- |
| *//全面扩展mvc配置* @Configuration *//@EnableWebMvc //相当于导入DelegatingWebMvcConfiguration.class 从容器中获取所有webmcvconfig* **public class** MyMvcConfig **implements** WebMvcConfigurer {  *// 视图跳转* @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/"**).setViewName(**"index"**);  registry.addViewController(**"/index.html"**).setViewName(**"index"**);  registry.addViewController(**"/main.html"**).setViewName(**"dashboard"**);  }   @Bean  **public** LocaleResolver localeResolver() {  **return new** MylocaleResolver();  }   @Override   **public void addInterceptors(InterceptorRegistry registry) {  *//添加拦截器* registry.addInterceptor(new LoginHandleInterceptor()).addPathPatterns("/\*\*").excludePathPatterns("/index.html", "/", "/user/login", "/css/\*\*", "/js/\*\*", "/img/\*\*");  }** } |

#### 2.1.3.3 员工查询

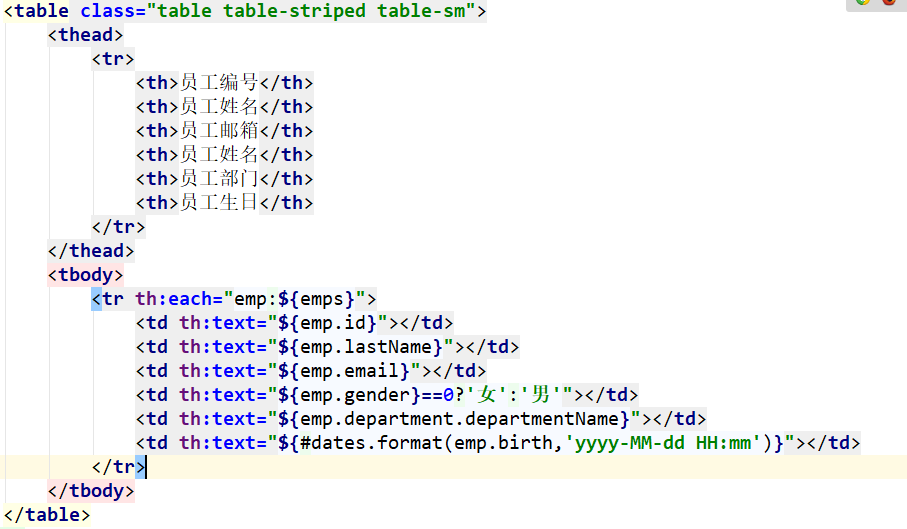
* 员工管理的链接上添加跳转controller路径 dashboard.html上



* 控制器方法

|  |
| --- |
| @Controller **public class** EmployeeController {  @Autowired  EmployeeDao **employeeDao**;  @Autowired  DepartmentDao **departmentDao**;  *//添加员工信息*  @RequestMapping(**"/emps"**)  **public** String list(Model model) {  Collection<Employee> employees = **employeeDao**.getAll();  model.addAttribute(**"emps"**, employees);  **return "emp/list"**;  }  } |

* 在list.html循环遍历数据



#### 2.1.3.4 员工添加

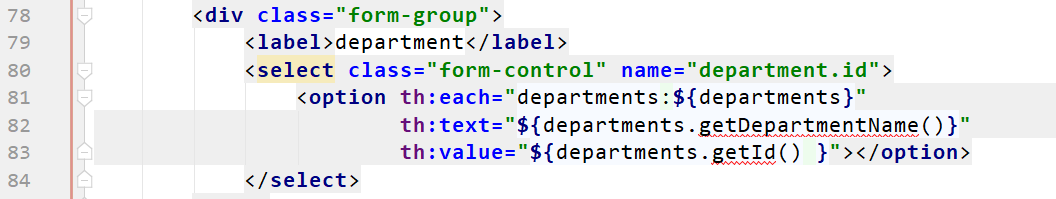
* 在员工添加按钮上添加 路径

|  |
| --- |
| <**h2**><**a class="btn btn-sm btn-success" th:href="@{/emp}"**>员工添加</**a**></**h2**> |

* 后台controller

|  |
| --- |
| *//点击添加员工，跳转到添加员工的页面* @GetMapping(**"/emp"**) **public** String toAddpage(Model model) {  *//添加数据之前查询部门信息的数据并渲染到页面* Collection<Department> departments = **departmentDao**.getDepartments();  model.addAttribute(**"departments"**, departments);  **return "emp/add"**; } |

* 在add.html循环展示下拉框的部门id



* 在添加员工的form表单上添加提交的地址 add.html

|  |
| --- |
| <**form th:action="@{/emp}" method="post"**> |

* 添加员工的controller

|  |
| --- |
| @PostMapping(**"/emp"**) **public** String addEmp(Employee employee){  System.***out***.println(employee);  **employeeDao**.save(employee);  **return "redirect:/emps"**; } |

* 注意细节:

再添加日期数据类型的时候，默认只支持yyyy/MM/dd类型，如果添加yyyy-MM-dd会报错。如果想添加yyyy-MM-dd的数据，需要在application.properties文件里面添加:

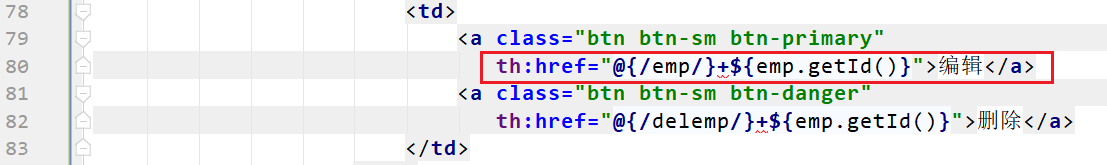
|  |
| --- |
| **spring.mvc.date-format**=**yyyy-MM-dd** |

#### 2.1.3.5 修改员工

* 定义修改页面，可以将add.html复制一份修改为update.html
* 点击编辑按钮，跳转到后台的修改的controller

后台通过员工id查询出需要修改的指定员工对象

* 在list.html编辑按钮的url



* 编写控制器方法

|  |
| --- |
| @GetMapping(**"/emp/{id}"**) **public** String toUpdateEmp(@PathVariable(**"id"**) Integer id, Model model) {  *//查出原来的数据* Employee employeeById = **employeeDao**.getEmployeeById(id);  model.addAttribute(**"emp"**, employeeById);  Collection<Department> departments = **departmentDao**.getDepartments();  model.addAttribute(**"departments"**, departments);  **return "emp/update"**; } |

* 由于要根据员工修改员工，所以将员工id也需要显示在update.html页面上

通过隐藏域存储员工的id值

|  |
| --- |
| <**input type="hidden" th:value="${emp.getId()}" name="id"**> |

* 定义修改的表单的action

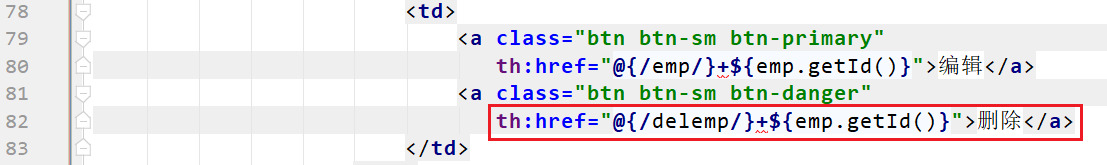
|  |
| --- |
| <**form th:action="@{/updateEmp}" method="post"**> |

* 后台编写修改的controller

|  |
| --- |
| *//修改员工信息* @PostMapping(**"/updateEmp"**) **public** String updateEmp(Employee employee) {  **employeeDao**.save(employee);  **return "redirect:/emps"**; } |

#### 2.1.3.6 删除员工

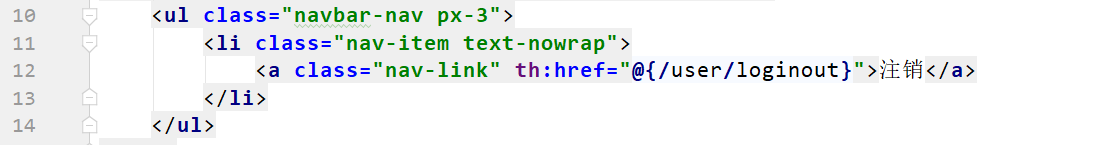
* 修改删除按钮的url



* 编写删除的controller

|  |
| --- |
| *//删除员工* @GetMapping(**"/delemp/{id}"**) **public** String updateEmp(@PathVariable(**"id"**) Integer id) {  **employeeDao**.delete(id);  **return "redirect:/emps"**; } |

#### 2.1.3.7 退出登录



|  |
| --- |
| *//退出登录* @GetMapping(**"/user/loginout"**) **public** String userLogin(HttpSession session) {  *//*session.invalidate();  session.removeAttribute(“loginUser”);  **return "redirect:/index"**; } |

### 2.1.4 springboot异常处理

#### 2.1.4.1 springboot默认处理异常的方式

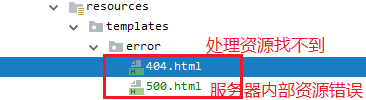
BasicErrorController：处理默认/error请求

DefaultErrorViewResolver：响应页面；去哪个页面是由DefaultErrorViewResolver解析得到的

（1） springboot默认处理异常的方式

* 在templates目录下面，创建一个error.html页面。只要后台资源发生了异常(不管是什么异常)，都统统的交给error页面来处理
* 如果我们需要对不同的异常做不同的处理，我们将错误页面命名为 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）；



页面能获取的信息；

timestamp：时间戳

status：状态码

error：错误提示

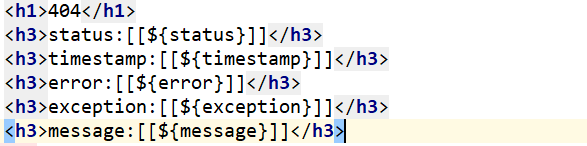
exception：异常对象

message：异常消息

errors：JSR303数据校验的错误都在这里

定义4xx.html或者5xx.html页面。在页面输出一些错误处理信息





#### 2.1.4.2 使用@ExceptionHandler注解处理

|  |
| --- |
| @Controller **public class** HelloController {   @RequestMapping(**"hello"**)  **public** String hello(){  **int** i = 10 / 0;  **return "success"**;  }   @RequestMapping(**"show1"**)  **public** String show1(){  String str = **null**;  System.***out***.println(str.length());  **return "success"**;  }   @RequestMapping(**"show2"**)  **public** String show2(){  **int**[] arr = **new int**[3];  arr[100] = 90;  **return "success"**;  }   *// 处理算术异常* @ExceptionHandler(value = {java.lang.ArithmeticException.**class**})  **public** ModelAndView handleArithmeticException(Exception e){  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"error"**,e.toString());*//向页面展示错误信息* mv.setViewName(**"error1"**);  **return** mv;  }   *//处理空指针异常* @ExceptionHandler(value = {java.lang.NullPointerException.**class**})  **public** ModelAndView handleNullPointerException(Exception e){  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"error"**,e.toString());*//向页面展示错误信息* mv.setViewName(**"error1"**);  **return** mv;  }   *//处理数组下标越界异常* @ExceptionHandler(value = {java.lang.ArrayIndexOutOfBoundsException.**class**})  **public** ModelAndView handleArrayIndexOutOfBoundsException(Exception e){  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"error"**,e.toString());*//向页面展示错误信息* mv.setViewName(**"error1"**);  **return** mv;  } } |

优点：可以自定义异常信息存储的key,自定义跳转视图的名称

缺点：需要编写大量的异常处理方法,不能跨controller，如果两个controller中出现同样的异常，需要重新编写异常处理的方法

#### 2.1.4.3 使用@ControllerAdvice+@ExceptionHandler全局异常处理

* 编写异常处理类

|  |
| --- |
| @ControllerAdvice *//标识当前类是一个全局异常处理类* **public class** GlobalException {  *// 处理算术异常* @ExceptionHandler(value = {java.lang.ArithmeticException.**class**})  **public** ModelAndView handleArithmeticException(Exception e){  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"error"**,e.toString());*//向页面展示错误信息* mv.setViewName(**"error1"**);  **return** mv;  }   *//处理空指针异常* @ExceptionHandler(value = {java.lang.NullPointerException.**class**})  **public** ModelAndView handleNullPointerException(Exception e){  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"error"**,e.toString());*//向页面展示错误信息* mv.setViewName(**"error1"**);  **return** mv;  }   *//处理数组下标越界异常* @ExceptionHandler(value = {java.lang.ArrayIndexOutOfBoundsException.**class**})  **public** ModelAndView handleArrayIndexOutOfBoundsException(Exception e){  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"error"**,e.toString());*//向页面展示错误信息* mv.setViewName(**"error1"**);  **return** mv;  } |

* 编写处理错误的页面 error1.html
* 编写controller 测试

#### 2.1.4.4 SimpleMappingExceptionResolver处理全局异常

* 需要将SimpleMappingExceptionResolver注入到spring容器里面

|  |
| --- |
| */\*\*  \* 全局异常处理器  \*/* @Configuration **public class** GlobalExceptionHandler {   @Bean  **public** SimpleMappingExceptionResolver getExceptionResolver(){  SimpleMappingExceptionResolver resolver = **new** SimpleMappingExceptionResolver();  Properties properties = **new** Properties();  */\*\*  \* 参数1：处理的具体的异常的类型  \* 参数2：处理异常需要跳转到的视图  \*/* properties.put(**"java.lang.NullPointerException"**,**"error3"**);  properties.put(**"java.lang.ArithmeticException"**,**"error3"**);  properties.put(**"java.lang.ArrayIndexOutOfBoundsException"**,**"error3"**);  resolver.setExceptionMappings(properties);  **return** resolver;  } } |

* 定义处理的页面 error3.html

确定:不显示具体的异常信息

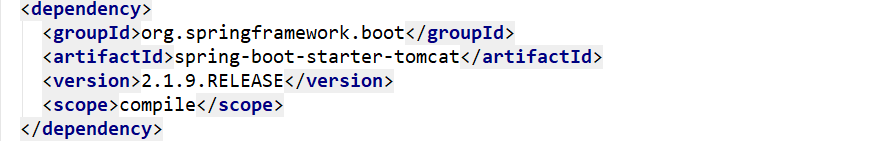
#### 2.1.4.5 自定义 HandlerExceptionResolver 类处理异常

|  |
| --- |
| @Configuration **public class** MyExceptionReslover **implements** HandlerExceptionResolver {   */\*\*  \* 处理异常的方法  \** ***@param httpServletRequest*** *\** ***@param httpServletResponse*** *\** ***@param o*** *\** ***@param ex*** *\** ***@return*** *\*/* @Override  **public** ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse,  Object o, Exception ex) {  ModelAndView mv = **new** ModelAndView();  *// 判断不同异常类型，做不同视图跳转* **if** (ex **instanceof** ArithmeticException) {  mv.setViewName(**"error4"**);  }  **if** (ex **instanceof** NullPointerException) {  mv.setViewName(**"error5"**);  }  mv.addObject(**"error"**, ex.toString());  **return** mv;  } } |

<https://blog.csdn.net/qq_24598601/article/details/89243914>

### 2.1.5 配置嵌入式Servlet容器

SpringBoot默认使用Tomcat作为嵌入式的Servlet容器；



#### 2.1.5.1 如何定制和修改Servlet容器的相关配置；

|  |
| --- |
| server.port=8081  server.servlet.context-path=/crud  server.tomcat.uri-encoding=UTF-8  //通用的Servlet容器设置  server.xxx  //Tomcat的设置  server.tomcat.xxx |

#### 2.1.5.2 注册Web三大组件

由于SpringBoot默认是以jar包的方式启动嵌入式的Servlet容器来启动SpringBoot的web应用，没有web.xml文件。

注册三大组件用以下方式:

* ServletRegistrationBean 注册Servlet组件
* FilterRegistrationBean 注册Filter组件
* ServletListenerRegistrationBean 注册监听器组件

1. 注册Servlet组件

* 编写一个Servlet

|  |
| --- |
| @WebServlet(**"/myServlet"**) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  System.***out***.println(**"Servlet执行了...."**);  } } |

* 在配置类里面注册Servlet组件

|  |
| --- |
| *//注册Servlet组件* @Bean **public** ServletRegistrationBean registerServlet(){  ServletRegistrationBean registrationBean = **new** ServletRegistrationBean(**new** MyServlet(),**"/myServlet"**);  **return** registrationBean; } |

1. 注册过滤器

* 定义过滤器类

|  |
| --- |
| **public class** MyFilter **implements** Filter {  @Override  **public void** doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) **throws** IOException, ServletException {  System.***out***.println(**"Filter执行了....."**);  filterChain.doFilter(request,response);  } } |

* 注册过滤器组件

|  |
| --- |
| *//注册Filter组件* @Bean **public** FilterRegistrationBean filterRegistrationBean(){  FilterRegistrationBean filterBean = **new** FilterRegistrationBean();  filterBean.setFilter(**new** MyFilter());  filterBean.setUrlPatterns(Arrays.*asList*(**"/myServlet"**));  **return** filterBean; } |

1. 注册监听器

* 定义监听器(监听ServContext对象的创建和销毁)

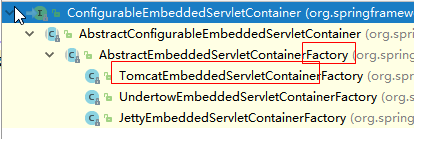
|  |
| --- |
| **public class** MyListener **implements** ServletContextListener {  *//监听SerlvetContext对象的创建* @Override  **public void** contextInitialized(ServletContextEvent sce) {  System.***out***.println(**"ServletContext对象创建了....."**);  }  *//监听SerlvetContext对象的销毁* @Override  **public void** contextDestroyed(ServletContextEvent sce) {  System.***out***.println(**"ServletContext对象销毁了....."**);  } } |

* 注册监听器

|  |
| --- |
| *//注册Listener组件* @Bean **public** ServletListenerRegistrationBean servletListenerRegistrationBean(){  ServletListenerRegistrationBean<MyListener> registrationBean = **new** ServletListenerRegistrationBean<>(**new** MyListener());  **return** registrationBean; } |

注意:正常退出才可以看出ServletContext对象销毁的效果 

#### 2.1.5.3 springboot中使用其他的Servlet容器



只需要在springboot的pom.xml中引入相关容器的依赖即可

* Jetty容器

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

* Undertow容器

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

## 3.1 springboot与Docker

### 3.1.1 Docker简介

**Docker**是一个开源的应用容器引擎；是一个轻量级容器技术；

Docker支持将软件编译成一个镜像；然后在镜像中各种软件做好配置，将镜像发布出去，其他使用者可以直接使用这个镜像；

运行中的这个镜像称为容器，容器启动是非常快速的。



### 3.1.2 Docker的核心概念

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；

docker镜像(Images)：软件打包好的镜像；放在docker仓库中

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

### 3.1.3 在linux环境下安装Docker

注意：Docker安装Docker要求linux内核必须高于3.10.可以在命令行里面查询当前linux的内核版本。

|  |
| --- |
| uname -r |

如果内核版本低于3.10.需要升级当前内核版本。命令为:

|  |
| --- |
| yum update |

**这里建议使用centOS7.0及以上的版本。这里给大家提供:**



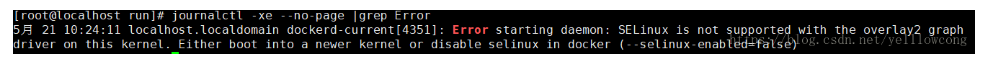
Docker的安装

|  |
| --- |
| yum install docker |

启动docker

|  |
| --- |
| systemctl start docker |

注意：如果出现



这个错误，是因为此linux的内核中的SELinux不支持 overlay2 graph driver ，解决方法有两个，要么启动一个新内核，要么就在docker里禁用selinux，–selinux-enabled=false。**在这里我们设置selinux-enabled=false**

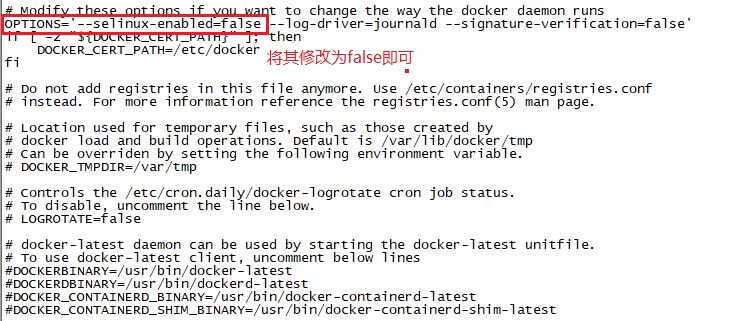
**设置方法如下:**

**编辑docker的配置文件**

**在命令行输入:**

|  |
| --- |
| **vi /etc/sysconfig/docker** |

**编辑效果如下:**

****

**然后按esc退出命令行插入模式，按Shift+:,然后WQ+回车键保存即可。**

查看按照的docker版本号:

|  |
| --- |
| docker -v |

设置docker开启启动

|  |
| --- |
| systemctl enable docker |

停止docker

|  |
| --- |
| stop docker |

### 3.1.4 docker的常用操作

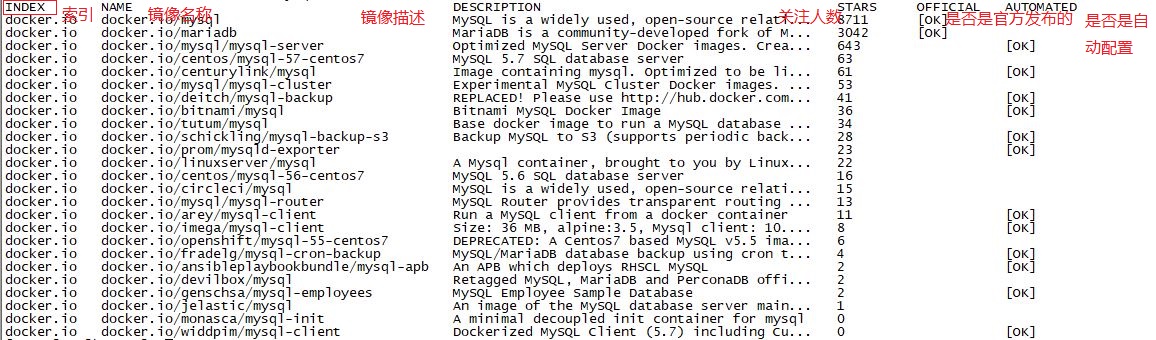
#### 3.1.4.1 镜像操作



* 操作mysql镜像
* 搜索mysql镜像

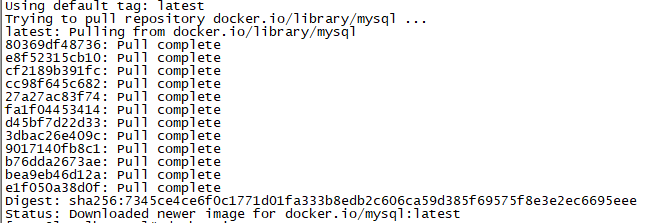
|  |
| --- |
| docker search mysql |

效果如下:



* 拉取镜像

|  |
| --- |
| docker pull mysql |



出现以上信息说明下载成功。

* 查看下载好的镜像

|  |
| --- |
| docker images |



* 删除镜像

|  |
| --- |
| docker rmi image-id |

#### 3.1.4.2 容器操作

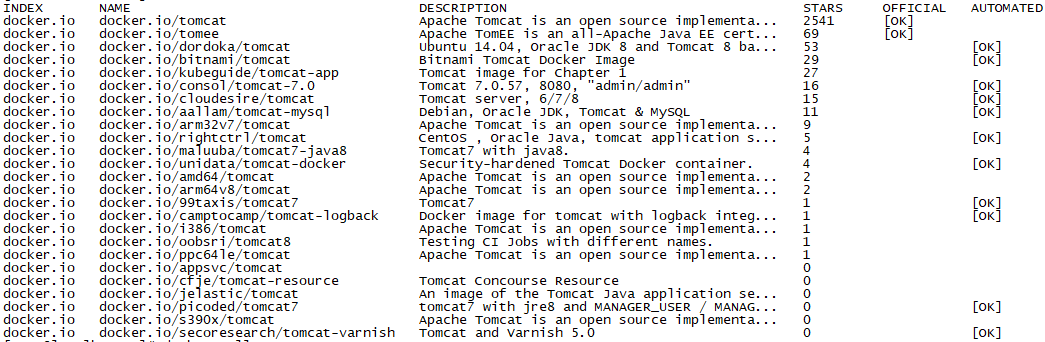
**(1) 安装tomcat容器**

软件镜像（QQ安装程序）----运行镜像----产生一个容器（正在运行的软件，运行的QQ）；

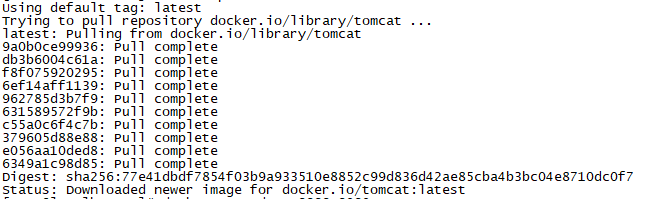
在这里以tomcat镜像为例。

步骤:

* 1. ：搜索镜像 docker search tomcat



* 1. ：拉取镜像 docker pull tomcat:8.0.52



出现以上信息说明镜像拉取成功

③： 根据镜像启动容器 docker run -d -p 8888:8080 tomcat

-d：后台运行

-p: 将主机的端口映射到容器的一个端口 主机端口:容器内部的端口

④： 查看运行中的容器 docker ps



⑤：查看能否访问tomcat 当前的ip地址:8888

<http://192.168.2.215:8888/>

注意:

为了演示简单,关闭了linux的防火墙

service firewalld status ；查看防火墙状态

service firewalld stop：关闭防火墙

查看容器日志 docker logs 容器id

查看tomcat容器的日志:docker logs b169b410a074

一个镜像可以启动多个容器，所以我可以启动多个tomcat容器

docker run -d -p 8889:8080 tomcat

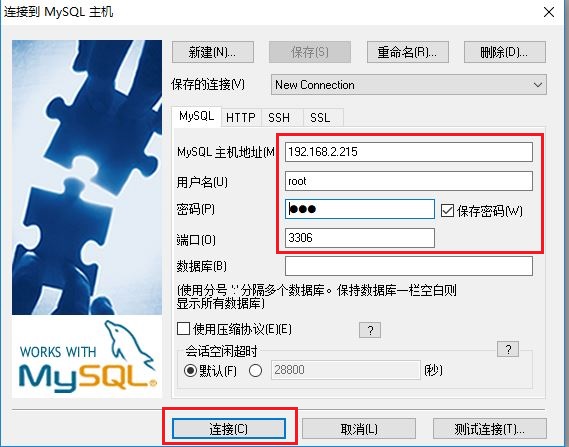
docker run -d -p **9090**:8080 tomcat

**(2) 安装mysql容器**

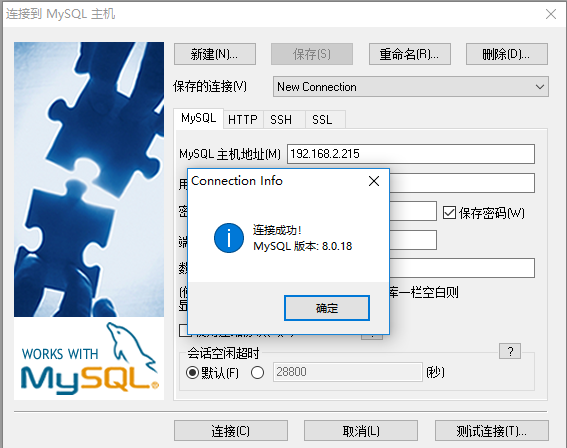
①：启动容器

|  |
| --- |
| docker run -p 3306:3306 --name mysql01 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql |

1. :外部使用sqlyog或者Navicat客户端工具测试连接



出现：



表示连接成功

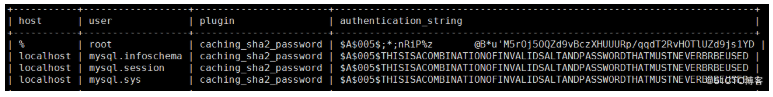
注意:如果连接不成功，出现



解决方案如下:

1. 在虚拟机中登录到mysql容器，然后进入mysql

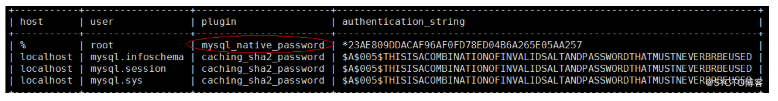
|  |
| --- |
| docker exec -it mysql02 /bin/bash  mysql -uroot -p  Enter password:  mysql> **select** host,user,plugin,authentication\_string **from** mysql.user; |



备注：host为 % 表示不限制ip localhost表示本机使用 plugin非mysql\_native\_password 则需要修改密码

1. 修改密码

|  |
| --- |
| **mysql>** use mysql;  **mysql>** alter user 'root'@'%' identified with mysql\_native\_password by '123';  **mysql>** flush privileges;  **mysql>** select host,user,plugin,authentication\_string from mysql.user; |



出现上述图片，再次用navicat远程连接mysql就成功了

## 4.1springboot与数据访问

### 4.1.1 springboot整合jdbc

使用脚手架快速创建一个springboot项目，注意我们需要使用jdbc进行数据持久层操作，所以需要勾选jdbc和mysql数据库的启动器。



* 在pom.xml中，添加的依赖如下:

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

* 在application.yml文件中配置:

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **password**: 123  **url**: jdbc:mysql://192.168.2.215:3306/springboot\_mysql?useSSL=false  **driver-class-name**: com.mysql.jdbc.Driver **#如果报红将pom.xml里面的mysql依赖的scope runtime注释掉即可** |

* 测试

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** SpringbootJdbcApplicationTests {  @Autowired  DataSource **dataSource**;  @Test  **public void** contextLoads() {  **try** {  Connection connection = **dataSource**.getConnection();  System.***out***.println(connection);  } **catch** (SQLException e) {  e.printStackTrace();  }  } } |

获取的连接对象:

|  |
| --- |
| HikariProxyConnection@1758876146 wrapping com.mysql.cj.jdbc.ConnectionImpl@3ad4a7d6 |

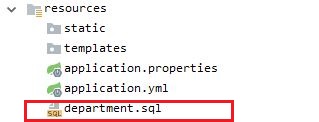
测试连接对象是否可用(使用java代码创建数据表)



* springboot支持在类路径下面放置sql脚本文件，那么在springboot项目启动的时候，会加载这些脚本文件的。

需求: 在springboot项目启动的时候，自动执行建表语句department.sql

1. ： 在类路径下面，也就是resources目录下面放置department.sql脚本文件



②：在application.yml文件中配置

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **password**: 123  **url**: jdbc:mysql://192.168.2.215:3306/springboot\_mysql  **driver-class-name**: com.mysql.jdbc.Driver   **schema:  - classpath:department.sql  initialization-mode: *always*** *# springboot项目加载的时候，自动执行这些脚本文件* |

* springboot还自动配置了JdbcTemplate用来操作数据库

需求：将test数据表中的数据查询出来。

|  |
| --- |
| @RestController **public class** DepartmentController {  @Autowired  JdbcTemplate **jdbcTemplate**;  @ResponseBody  @RequestMapping(**"/query"**)  **public** Map<String, Object> test(){  List<Map<String, Object>> list = **jdbcTemplate**.queryForList(**"SELECT** *\** **FROM test"**);  **return** list.get(0);  } } |

### 4.1.2 springboot整合druid数据源

#### 4.1.2.1 整合步骤

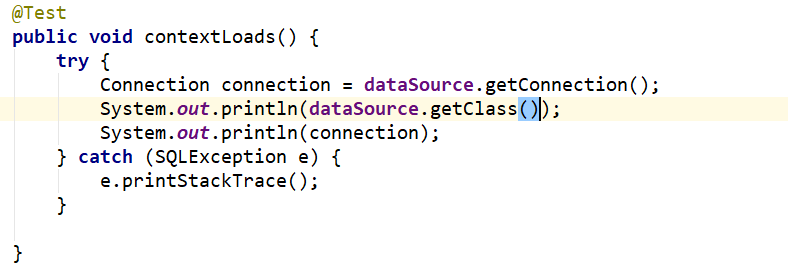
* 导入druid数据源依赖

|  |
| --- |
| *<!--引入Druid数据源-->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.14</**version**> </**dependency**> |

* Application.yml配置文件



* 测试



测试结果:



#### 4.1.2.2 druid数据源的相关配置

* 引入log4j依赖

|  |
| --- |
| *<!-- springboot-log4j -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j</**artifactId**>  <**version**>1.3.8.RELEASE</**version**> </**dependency**> |

* 在application.yml配置文件中添加配置信息(红色部分)

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **password**: 123  **url**: jdbc:mysql://192.168.2.215:3306/springboot\_mysql  **driver-class-name**: com.mysql.cj.jdbc.Driver *#注意：这里不推荐使用com.mysql.jdbc.Driver* **type**: com.alibaba.druid.pool.DruidDataSource *#指定数据源类型* **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  filters: stat,wall,log4j  maxPoolPreparedStatementPerConnectionSize: 20  useGlobalDataSourceStat: true  connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500**  *# schema:  # - classpath:department.sql* **initialization-mode**: *always # springboot项目加载的时候，自动执行这些脚本文件* |

* 通过配置类将配置信息与数据源进行绑定

|  |
| --- |
| @Configuration **public class** DruidConfig {  @ConfigurationProperties(prefix = **"spring.datasource"**)  @Bean  **public** DataSource druid(){  **return new** DruidDataSource();  } } |

#### 4.1.2.3 配置数据源监控

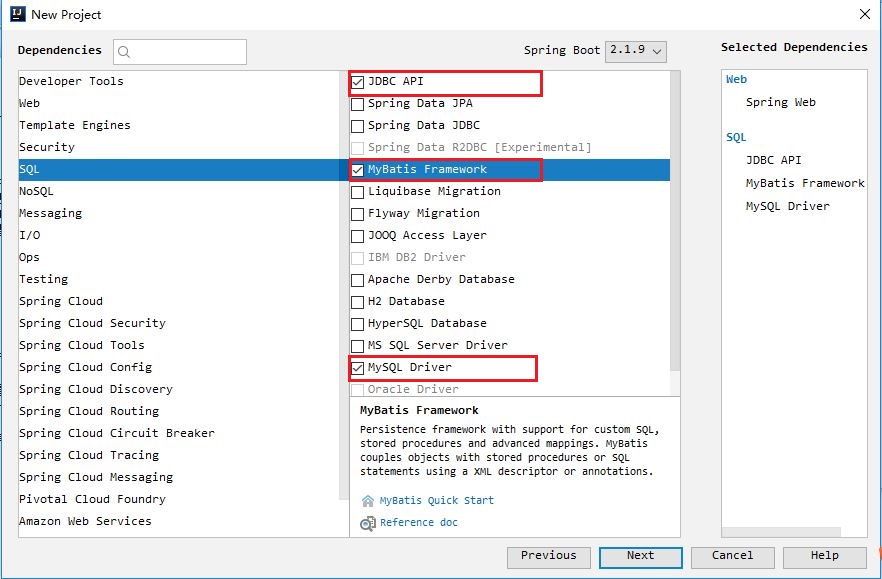
配置数据源监控的步骤:

1. 配置一个管理后台的Servlet。
2. 配置一个web监控的filter。

|  |
| --- |
| *//配置Druid的监控 //1、配置一个管理后台的Servlet* @Bean **public** ServletRegistrationBean statViewServlet(){  ServletRegistrationBean bean = **new** ServletRegistrationBean(**new** StatViewServlet(), **"/druid/\*"**);  Map<String,String> initParams = **new** HashMap<>();  initParams.put(**"loginUsername"**,**"admin"**);  initParams.put(**"loginPassword"**,**"123456"**);  initParams.put(**"allow"**,**""**);*//默认就是允许所有访问* initParams.put(**"deny"**,**"192.168.15.21"**);  bean.setInitParameters(initParams);  **return** bean; } *//2、配置一个web监控的filter* @Bean **public** FilterRegistrationBean webStatFilter(){  FilterRegistrationBean bean = **new** FilterRegistrationBean();  bean.setFilter(**new** WebStatFilter());  Map<String,String> initParams = **new** HashMap<>();  initParams.put(**"exclusions"**,**"\*.js,\*.css,/druid/\*"**);  bean.setInitParameters(initParams);  bean.setUrlPatterns(Arrays.*asList*(**"/\*"**));  **return** bean; } |

### 4.1.3 springboot整合mybatis

使用脚手架创建springboot项目的时候需要勾选以下信息:



#### 4.1.3.1 搭建整合mybatis的环境

* 引入依赖

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.2.0.RELEASE</**version**> </**parent**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.2</**version**>  </**dependency**> </**dependencies**> |

* 编写启动类

|  |
| --- |
| @SpringBootApplication @MapperScan(basePackages = **"com.qf.mapper"**) *//扫描接口 生成代理对象* **public class** MybatisApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(MybatisApplication.**class**,args);  } } |

* 编写配置文件

|  |
| --- |
| **server**:  **port**: 8888 **spring**:  **datasource**:  **url**: jdbc:mysql://192.168.10.137:3306/ssm  **username**: root  **password**: Admin123!  **driver-class-name**: com.mysql.jdbc.Driver *#mybatis的相关配置* **mybatis**:  *# mapper配置文件* **mapper-locations**: classpath:mapper/\*.xml  *# 配置别名* **type-aliases-package**: com.qf.pojo |

* 编写接口

|  |
| --- |
| **public interface** AccountDao {  **public** Account findAccountById(Integer id); } |

* 编写映射文件

|  |
| --- |
| *<?***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.qf.mapper.AccountDao"**>  <**select id="findAccountById" parameterType="int" resultType="account"**>  select *\** from account where id=#{id}  </**select**> </**mapper**> |

* 编写Service

略

* 编写Controller

|  |
| --- |
| @RestController @RequestMapping(**"account"**) **public class** AccountController {   @Autowired  AccountService **accountService**;   @RequestMapping(**"findById"**)  **public** Account account(@RequestParam(**"id"**) Integer id){  **return accountService**.findAccountById(id);  } } |