Spring Boot

拓薪教育

任亮

# Spring Boot 入门

## Spring Boot 简介

简化Spring应用开发的一个框架；  
整个Spring技术栈的一个大整合；  
J2EE开发的一站式解决方案；

## 微服务

2014，martin fowler  
微服务：架构风格（服务微化）

一个应用应该是一组小型服务；可以通过HTTP的方式进行互通；

单体应用：ALL IN ONE

微服务：每一个功能元素最终都是一个可独立替换和独立升级的软件单元；

微服务文档[https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa](https://martinfowler.com/articles/microservices.html" \l "MicroservicesAndSoa)

## 环境准备

环境约束

–jdk1.8：Spring Boot 推荐jdk1.7及以上；java version "1.8.0\_112"

–maven3.x：maven 3.3以上版本；Apache Maven 3.3.9

–IntelliJIDEA2017：IntelliJ IDEA 2017.3.3 x64、STS

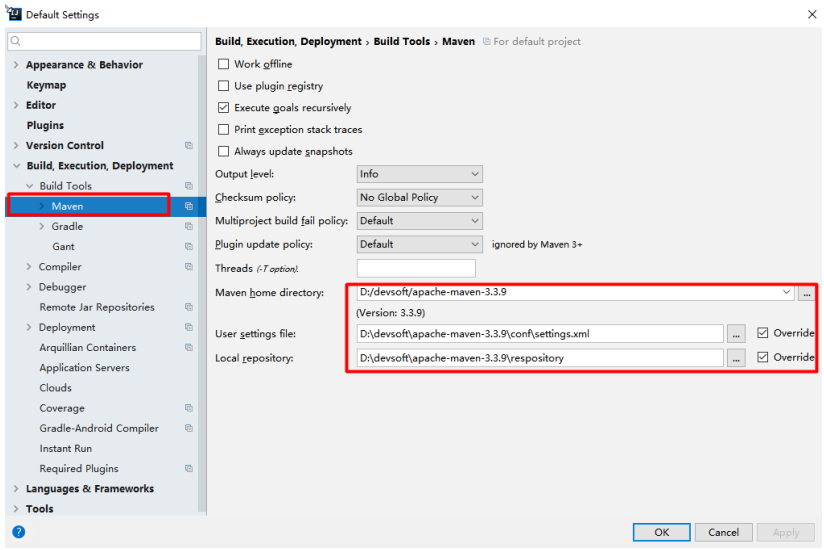
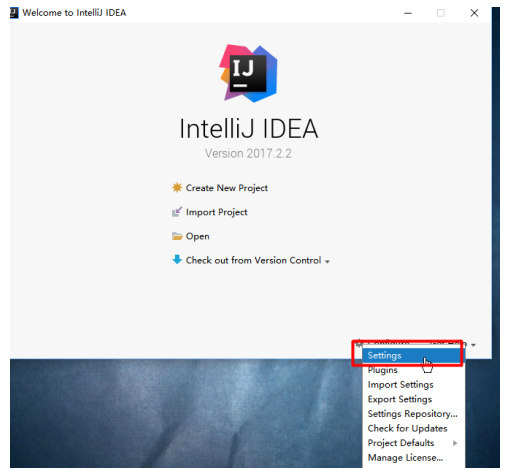
–SpringBoot 2.0.2.RELEASE：统一环境；

### MAVEN设置

给maven 的settings.xml配置文件的profiles标签添加

|  |
| --- |
| <profile> <id>jdk‐1.8</id> <activation>  <activeByDefault>true</activeByDefault> <jdk>1.8</jdk> </activation> <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion> </properties>  </profile> |

### IDEA设置



### Spring Boot HelloWorld

浏览器发送hello请求，服务器接受请求并处理，响应Hello World字符串；

1. 创建一个maven工程；（jar）
2. 导入spring boot相关的依赖

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

1. 编写一个主程序；启动Spring Boot应用

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

1. 编写相关的Controller、Service

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

1. 运行主程序测试
2. 简化部署

|  |
| --- |
| < !‐‐ 这个插件，可以将应用打包成一个可执行的jar包；‐‐> <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring‐boot‐maven‐plugin</artifactId>  </plugin>  </plugins> </build> |

将这个应用打成jar包，直接使用java -jar的命令进行执行；

### Hello World探究

POM文件

1. 父项目

|  |
| --- |
| <parent> <groupId>org.springframework.boot</groupId> <artifactId>spring‐boot‐starter‐parent</artifactId> <version>1.5.9.RELEASE</version>  </parent> 他的父项目是  <parent> <groupId>org.springframework.boot</groupId> <artifactId>spring‐boot‐dependencies</artifactId> <version>1.5.9.RELEASE</version> <relativePath>../../spring‐boot‐dependencies</relativePath>  </parent> 他来真正管理Spring Boot应用里面的所有依赖版本；  Spring Boot的版本仲裁中心； 以后我们导入依赖默认是不需要写版本；（没有在dependencies里面管理的  依赖自然需要声明版本号） |

1. 启动器

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

**spring-boot-starter-web：**spring-boot-starter：spring-boot场景启动器；帮我们导入了web模块正常运行所依赖的组件；  
Spring Boot将所有的功能场景都抽取出来，做成一个个的starters（启动器），只需要在项目里面引入这些starter  
相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动器

### 主程序类，主入口类

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

1. **@SpringBootApplication:**

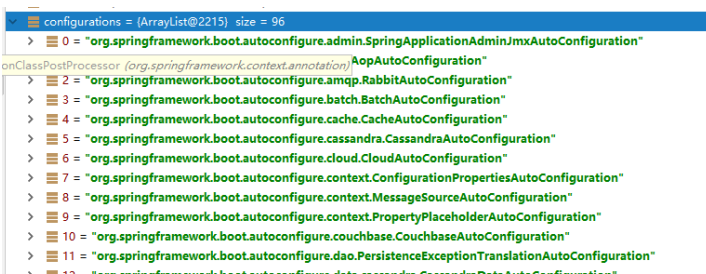
Spring Boot应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot  
就应该运行这个类的main方法来启动SpringBoot应用；

|  |
| --- |
| @Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherited @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(excludeFilters = {  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) }) public @interface SpringBootApplication { |

1. **@SpringBootConfiguration:Spring Boot的配置类；**标注在某个类上，表示这是一个Spring Boot的配置类；
2. **@Configuration:配置类上来标注这个注解；**配置类 ----- 配置文件；配置类也是容器中的一个组件；@Component
3. **@EnableAutoConfiguration：开启自动配置功能；**以前我们需要配置的东西，Spring Boot帮我们自动配置；@EnableAutoConfiguration告诉SpringBoot开启自  
   动配置功能；这样自动配置才能生效；

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

1. **@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（@SpringBootApplication标注的类）的所在包及下面所有子包里面的所有组件扫描到Spring容器；
2. **@Import(EnableAutoConfigurationImportSelector.class)；**给容器中导入组件？  
   EnableAutoConfigurationImportSelector：导入哪些组件的选择器；  
   将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中；  
   会给容器中导入非常多的自动配置类（xxxAutoConfiguration）；就是给容器中导入这个场景需要的所有组件，  
   并配置好这些组件；



有了自动配置类，免去了我们手动编写配置注入功能组件等的工作；  
SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,classLoader)；  
Spring Boot在启动的时候从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将  
这些值作为自动配置类导入到容器中，自动配置类就生效，帮我们进行自动配置工作；以前我们需要自己配置的东  
西，自动配置类都帮我们；  
J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar；

## 使用Spring Initializer快速创建Spring Boot项目

IDEA：使用 Spring Initializer快速创建项目

IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目；  
选择我们需要的模块；向导会联网创建Spring Boot项目；  
默认生成的Spring Boot项目；  
主程序已经生成好了，我们只需要我们自己的逻辑  
resources文件夹中目录结构  
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；

# 配置文件

## Springboot的热部署

spring为开发者提供了一个名为spring-boot-devtools的模块来使Spring Boot应用支持热部署，提高开发者的开发效率，无需手动重启Spring Boot应用。

引入依赖

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

修改java代码或者配置文件模板后可以通过ctrl+f9来实施热部署。

## 配置文件

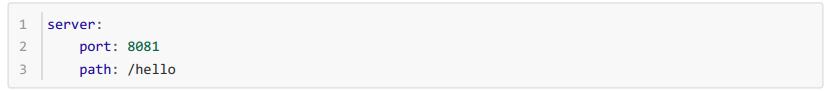
SpringBoot使用一个全局的配置文件，配置文件名是固定的；  
•application.properties  
•application.yml

配置文件的作用：修改SpringBoot自动配置的默认值；SpringBoot在底层都给我们自动配置好；  
YAML（YAML Ain't Markup Language）  
YAML A Markup Language：是一个标记语言  
YAML isn't Markup Language：不是一个标记语言；  
标记语言：  
以前的配置文件；大多都使用的是 xxxx.xml文件；  
YAML：以数据为中心，比json、xml等更适合做配置文件；



## YAML语法

1. 基本语法  
   k:(空格)v：表示一对键值对（空格必须有）；  
   以空格的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；

1. 值的写法

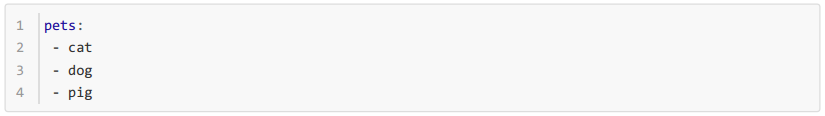
字面量：普通的值（数字，字符串，布尔）  
k: v：字面直接来写；  
字符串默认不用加上单引号或者双引号；  
""：双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思  
name: "zhangsan \n lisi"：输出；zhangsan 换行 lisi  
''：单引号；会转义特殊字符，特殊字符最终只是一个普通的字符串数据  
name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi

1. 对象、Map（属性和值）（键值对）：  
   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



数组（List、Set）：

用- 值表示数组中的一个元素



行内写法



Yml文件可以引用文件中的属性（上下文都可） ，也可以拼接属性

|  |
| --- |
| **person:  name:** max  **age:** 22  **birth:** 2012/3/3  **list:** [dog,cat,hourse]  **maps:** {**key1:** v1,**key2:** v2}  **dog:  name:** ${**person.name**}\_hello #属性引用和拼接  **age:** ${**person.age**} |

## 配置文件值注入



通过注解形式给指定类注入属性值，主要有两个注解，@configuration ,@Value

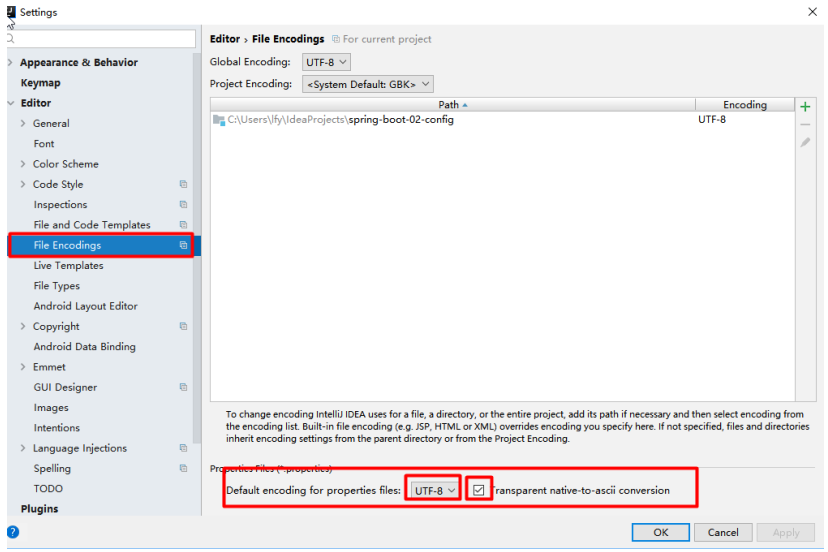
其中@Configuration是批量注入，@Value是单个属性注入

@value注解没法读取yml或properties文件中的三级以上的属性

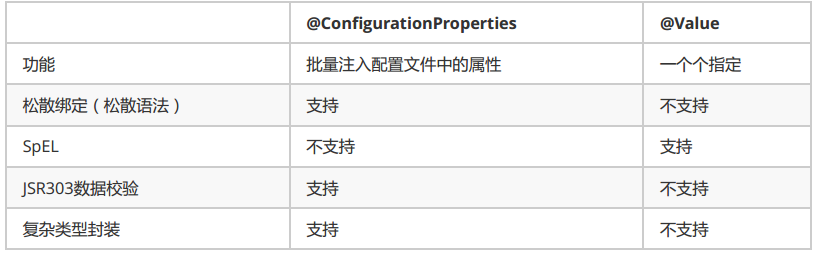
我们可以导入配置文件处理器，以后编写配置就有提示了

|  |
| --- |
| <!‐‐导入配置文件处理器，配置文件进行绑定就会有提示‐‐> <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring‐boot‐configuration‐processor</artifactId>  <optional>true</optional> </dependency> |

1. **properties配置文件在idea中默认utf-8可能会乱码**



2. @Value获取值和@ConfigurationProperties获取值比较



对于@ConfigurationProperties注解来说，他是支持数据校验，对于@Value来说，不支持

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；

### 配置文件注入值数据校验

通过@ConfigurationProperties和@Value注解

|  |
| --- |
| @Component @ConfigurationProperties(prefix = "person") @Validated public class Person {  /\*\*  \* <bean class="Person">  \* <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#  {SpEL}"></property>  \* <bean/>  \*/  //lastName必须是邮箱格式  @Email  //@Value("${person.last‐name}")  private String lastName;  //@Value("#{11\*2}")  private Integer age;  //@Value("true")  private Boolean boss;  private Date birth;  private Map<String,Object> maps;  private List<Object> lists;  private Dog dog; |

### @PropertySource&@ImportResource&@Bean

@PropertySource：加载指定的配置文件；

使用@PropertySource 配置文件进行值得注入时，默认先读取application。Yml/properties里的文件，再通过PropertySource注解加载其他配置文件

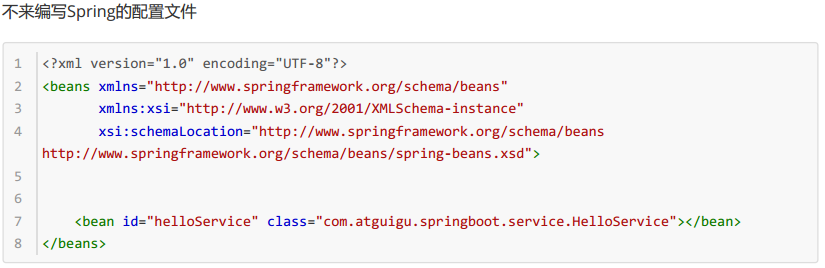
所有的通过文件注入属性值的类都要求该类被容器所管理

|  |
| --- |
| 总结：在boot中，将类添加到boot容器的方式   1. 通过配置文件类@Configuration，@Bean注解方式注入 2. 通过注解@Compenent @service。。。。。等注解注入 3. 通过spring配置文件beans.xml,再将beans.xmlt通过@ImportSource(Location=” classpath:beans.xml”)注入到容器中 |

|  |
| --- |
| /\*\* \* 将配置文件中配置的每一个属性的值，映射到这个组件中 \* @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定； \* prefix = "person"：配置文件中哪个下面的所有属性进行一一映射 \* \* 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能； \* @ConfigurationProperties(prefix = "person")默认从全局配置文件中获取值； \* \* / @PropertySource(value = {"classpath:person.properties"}) @Component @ConfigurationProperties(prefix = "person") //@Validated  public class Person { /\*\* \* <bean class="Person"> \* <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/# {SpEL}"></property> \* <bean/> \*/ //lastName必须是邮箱格式 // @Email //@Value("${person.last‐name}") private String lastName; //@Value("#{11\*2}") private Integer age; //@Value("true") private Boolean boss; |

**@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



SpringBoot推荐给容器中添加组件的方式；推荐使用全注解的方式  
1、配置类@Configuration------>Spring配置文件  
2、使用@Bean给容器中添加组件

|  |
| --- |
| /\*\* \* @Configuration：指明当前类是一个配置类；就是来替代之前的Spring配置文件 \* \* 在配置文件中用<bean><bean/>标签添加组件 \* \* / @Configuration public class MyAppConfig {  //将方法的返回值添加到容器中；容器中这个组件默认的id就是方法名  @Bean  public HelloService helloService02(){  System.out.println("配置类@Bean给容器中添加组件了...");  return new HelloService();  } } |

## 配置文件占位符

1、随机数

|  |
| --- |
| ${random.value}、${random.int}、${random.long} ${random.int(10)}、${random.int[1024,65536]} |

2、占位符获取之前或者之后配置的值，如果没有可以是用:指定默认值

|  |
| --- |
| person.last‐name=张三${random.uuid} person.age=${random.int} person.birth=2017/12/15 person.boss=false person.maps.k1=v1 person.maps.k2=14 person.lists=a,b,c person.dog.name=${person.hello:hello}\_dog person.dog.age=15 |

## Profile

1、多Profile文件  
我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml  
默认使用application.properties的配置；

2、yml支持多文档块方式



3、激活指定profile

1. 在配置文件中指定 spring.profiles.active=dev  
   2、命令行：  
   java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；  
   可以直接在测试的时候，配置传入命令行参数  
   3、虚拟机参数；  
   -Dspring.profiles.active=dev

## 配置文件加载位置

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/  
–file:./  
–classpath:/config/  
–classpath:/  
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；

在-classpath：/的application.properties.中加入

server.servlet.context-path=/boot3访问根目录

在实际开发过程中，若项目大，配偶之文件可以放在resource下的config文件夹，、

项目小，放在resource下即可，

配置文件建议放在resource下

## 外部配置加载顺序

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

但是外部文件优先级高于内部，内部文件的配置会被外部文件屏蔽  
1.命令行参数  
所有的配置都可以在命令行上进行指定  
java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc  
多个配置用空格分开； --配置项=值  
2.来自java:comp/env的JNDI属性  
3.Java系统属性（System.getProperties()）  
4.操作系统环境变量  
5.RandomValuePropertySource配置的random.\*属性值  
由jar包外向jar包内进行寻找；  
优先加载带profile  
6.jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件  
7.jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件  
再来加载不带profile  
8.jar包外部的application.properties或application.yml(不带spring.profile)配置文件  
9.jar包内部的application.properties或application.yml(不带spring.profile)配置文件  
10.@Configuration注解类上的@PropertySource  
11.通过SpringApplication.setDefaultProperties指定的默认属性  
所有支持的配置加载来源；

资料参考：

[https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#boot-features-external-config](https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/" \l "boot-features-external-config)

## 自动配置原理

配置文件到底能写什么？怎么写？自动配置原理；

[https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#common-application-properties](https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/" \l "common-application-properties)

### 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
2. @EnableAutoConfiguration 作用：  
   利用EnableAutoConfigurationImportSelector给容器中导入一些组件？  
   可以查看selectImports()方法的内容；  
   List configurations = getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置

|  |
| --- |
| SpringFactoriesLoader.loadFactoryNames() 扫描所有jar包类路径下 META‐INF/spring.factories 把扫描到的这些文件的内容包装成properties对象 从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器 中 |

将 类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值加入到了容器中；

|  |
| --- |
| # Auto Configure org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\ org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\ org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\ org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\ org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\ org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\ org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\ org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\ org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\ org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\ org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration, \ o rg.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\ org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration ,\ org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\ org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration ,\ org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\ org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration ,\ org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfi guration,\ org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\ org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration,\ org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\ org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\ org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\ org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\ org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\ org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\ org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\ org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\ org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\ org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\ org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\ org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\ org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\ org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\ org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\ org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\ org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\ org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\ org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\ org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\ org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\ org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\ org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\ org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\ org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\ org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\ org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\ org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\ org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\ org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\ org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\ org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\ org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\ org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\ org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\ org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\ org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\ org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\ org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\ org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\ org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\ org.springframework.boot.autoconfigure.mobile.DeviceDelegatingViewResolverAutoConfiguration, \ o rg.springframework.boot.autoconfigure.mobile.SitePreferenceAutoConfiguration,\ org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\ org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\ org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\ org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\ org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\ org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\ org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,\ org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,\ org.springframework.boot.autoconfigure.security.oauth2.OAuth2AutoConfiguration,\ org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\ org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\ org.springframework.boot.autoconfigure.social.SocialWebAutoConfiguration,\ org.springframework.boot.autoconfigure.social.FacebookAutoConfiguration,\ org.springframework.boot.autoconfigure.social.LinkedInAutoConfiguration,\ org.springframework.boot.autoconfigure.social.TwitterAutoConfiguration,\ org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\ org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\ org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\ org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\ org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\ org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\ org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\ org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\ org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration,\ org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\ org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\ org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\ org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration,\ org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\ org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration,\ org.springframework.boot.autoconfigure.websocket.WebSocketMessagingAutoConfiguration,\ org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguratio |

每一个这样的 xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；  
3. 每一个自动配置类进行自动配置功能；  
4. 以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理；

|  |
| --- |
| @Configuration //表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件 @EnableConfigurationProperties(HttpEncodingProperties.class) //启动指定类的 ConfigurationProperties功能；将配置文件中对应的值和HttpEncodingProperties绑定起来；并把 HttpEncodingProperties加入到ioc容器中 @ConditionalOnWebApplication //Spring底层@Conditional注解（Spring注解版），根据不同的条件，如果 满足指定的条件，整个配置类里面的配置就会生效； 判断当前应用是否是web应用，如果是，当前配置类生效 @ConditionalOnClass(CharacterEncodingFilter.class) //判断当前项目有没有这个类 CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器； @ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true) //判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的 //即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的； public class HttpEncodingAutoConfiguration {  //他已经和SpringBoot的配置文件映射了  private final HttpEncodingProperties properties;  //只有一个有参构造器的情况下，参数的值就会从容器中拿  public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {  this.properties = properties;  }  @Bean //给容器中添加一个组件，这个组件的某些值需要从properties中获取 @ConditionalOnMissingBean(CharacterEncodingFilter.class) //判断容器没有这个组件？ public CharacterEncodingFilter characterEncodingFilter() {  CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();  filter.setEncoding(this.properties.getCharset().name());  filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));  filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));  return filter; } |

根据当前不同的条件判断，决定这个配置类是否生效？  
一但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取  
的，这些类里面的每一个属性又是和配置文件绑定的；  
5. 所有在配置文件中能配置的属性都是在xxxxProperties类中封装者‘；配置文件能配置什么就可以参照某个功  
能对应的这个属性类

|  |
| --- |
| @ConfigurationProperties(prefix = "spring.http.encoding") //从配置文件中获取指定的值和bean的属 性进行绑定 public class HttpEncodingProperties { public static final Charset DEFAULT\_CHARSET = Charset.forName("UTF‐8"); |

精髓：  
1）、SpringBoot启动会加载大量的自动配置类  
2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；  
3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）  
4）、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这  
些属性的值；

xxxxAutoConfigurartion：自动配置类；  
给容器中添加组件

### 细节

1. @Conditional派生注解（Spring注解版原生的@Conditional作用）

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效；

|  |  |
| --- | --- |
| @Conditional扩展注解 | 作用（判断是否满足当前指定条件） |
| @ConditionalOnJava | 系统的java版本是否符合要求 |
| @ConditionalOnBean | 容器中存在指定Bean； |
| @ConditionalOnMissingBean | 容器中不存在指定Bean； |
| @ConditionalOnExpression | 满足SpEL表达式指定 |
| @ConditionalOnClass | 系统中有指定的类 |
| @ConditionalOnMissingClass | 系统中没有指定的类 |
| @ConditionalOnSingleCandidate | 容器中只有一个指定的Bean，或者这个Bean是首选Bean |
| @ConditionalOnProperty | 系统中指定的属性是否有指定的值 |
| @ConditionalOnResource | 类路径下是否存在指定资源文件 |
| @ConditionalOnWebApplication | 当前是web环境 |
| @ConditionalOnNotWebApplication | 当前不是web环境 |
| @ConditionalOnJndi | JNDI存在指定项 |

自动配置类必须在一定的条件下才能生效；  
我们怎么知道哪些自动配置类生效；  
我们可以通过启用 debug=true属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置  
类生效；

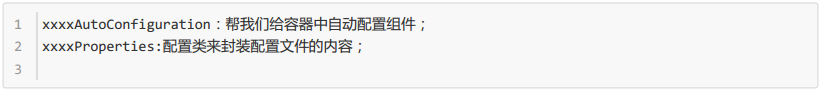
|  |
| --- |
| ======================== AUTO‐CONFIGURATION REPORT ========================= Positive matches:（自动配置类启用的） ‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐ DispatcherServletAutoConfiguration matched: ‐ @ConditionalOnClass found required class 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition) ‐ @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition) Negative matches:（没有启动，没有匹配成功的自动配置类） ‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐ ActiveMQAutoConfiguration: Did not match: ‐ @ConditionalOnClass did not find required classes 'javax.jms.ConnectionFactory', 'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition) AopAutoConfiguration: Did not match: ‐ @ConditionalOnClass did not find required classes 'org.aspectj.lang.annotation.Aspect', 'org.aspectj.lang.reflect.Advice' (OnClassCondition) |

# Web开发

## 简介

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；

自动配置原理？  
这个场景SpringBoot帮我们配置了什么？能不能修改？能修改哪些配置？能不能扩展？xxx



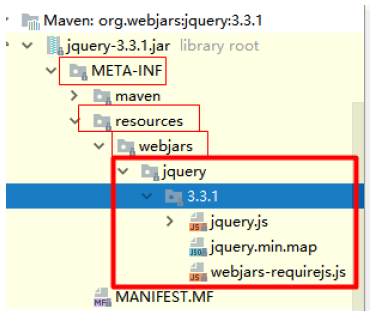
## SpringBoot对静态资源的映射规则

|  |
| --- |
| @ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false) public class ResourceProperties implements ResourceLoaderAware { //可以设置和静态资源有关的参数，缓存时间等 |

|  |
| --- |
| WebMvcAuotConfiguration： @Override public void addResourceHandlers(ResourceHandlerRegistry registry) {  if (!this.resourceProperties.isAddMappings()) {  logger.debug("Default resource handling disabled");  return;  }  Integer cachePeriod = this.resourceProperties.getCachePeriod(); if (!registry.hasMappingForPattern("/webjars/\*\*")) {  customizeResourceHandlerRegistration(registry.addResourceHandler("/webjars/\*\*").addResourceLocations("classpath:/META‐INF/resources/webjars/").setCachePeriod(cachePeriod));  }  String staticPathPattern = this.mvcProperties.getStaticPathPattern(); //静态资源文件夹映射 if (!registry.hasMappingForPattern(staticPathPattern)) {  customizeResourceHandlerRegistration(registry.  addResourceHandler(staticPathPattern).addResourceLocations(this.resourceProperties.getStaticLocations()).setCachePeriod(cachePeriod));  } } / /配置欢迎页映射 @Bean public WelcomePageHandlerMapping welcomePageHandlerMapping(  ResourceProperties resourceProperties) {  return new WelcomePageHandlerMapping(resourceProperties.getWelcomePage(),  this.mvcProperties.getStaticPathPattern()); } //配置喜欢的图标 @Configuration @ConditionalOnProperty(value = "spring.mvc.favicon.enabled", matchIfMissing = true) public static class FaviconConfiguration {  private final ResourceProperties resourceProperties;  public FaviconConfiguration(ResourceProperties resourceProperties) {  this.resourceProperties = resourceProperties; }  @Bean public SimpleUrlHandlerMapping faviconHandlerMapping() {  SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();  mapping.setOrder(Ordered.HIGHEST\_PRECEDENCE + 1);  //所有 \*\*/favicon.ico  mapping.setUrlMap(Collections.singletonMap("\*\*/favicon.ico",  faviconRequestHandler()); return mapping; }  @Bean public ResourceHttpRequestHandler faviconRequestHandler() {  ResourceHttpRequestHandler requestHandler = new ResourceHttpRequestHandler();  requestHandler.setLocations(this.resourceProperties.getFaviconLocations());  return requestHandler;  } } |

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

webjars：以jar包的方式引入静态资源；<http://www.webjars.org/>



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

|  |
| --- |
| <!‐‐引入jquery‐webjar‐‐>在访问的时候只需要写webjars下面资源的名称即可  <dependency>  <groupId>org.webjars</groupId>  <artifactId>jquery</artifactId>  <version>3.3.1</version>  </dependency> |

2）、"/\*\*" 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 |

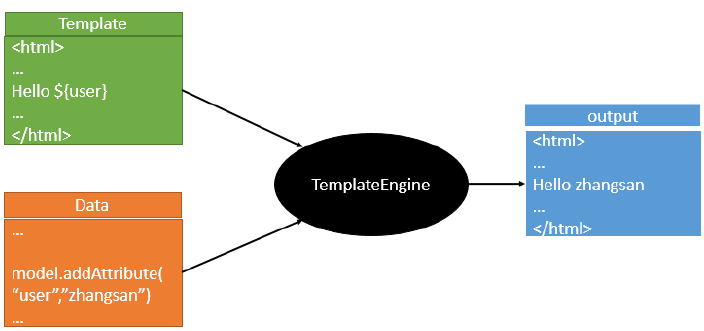
localhost:8080/abc === 去静态资源文件夹里面找abc

3）、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页面

## 模板引擎

JSP、Velocity、Freemarker、Thymeleaf



SpringBoot推荐的Thymeleaf；

语法更简单，功能更强大；

### 引入thymeleaf；

在pom.xml中引入

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

从spring父文件中能看到Springboot2.0.1所使用的thymeleaf版本是3.0.9

springBoot启动的时候会自动配置

**org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration**

从ThymeleafAutoConfiguration的源代码中我们可以得知ThymeleafProperties

中配置了Thymeleaf的规则

|  |
| --- |
| **public class** ThymeleafProperties {  **private static final** Charset DEFAULT\_ENCODING;  **public static final** String DEFAULT\_PREFIX = **"classpath:/templates/"**;  **public static final** String DEFAULT\_SUFFIX = **".html"**;  **private boolean** checkTemplate = **true**;  **private boolean** checkTemplateLocation = **true**;  **private** String prefix = **"classpath:/templates/"**;  **private** String suffix = **".html"**;  **private** String mode = **"HTML"**;  **private** Charset encoding;  **private boolean** cache; |

我们使用html作为模板，而且默认的前缀是放在classpath:/template/下，后缀是.html

当然这些属性我们都可以通过application.properties来修改。我们采用默认即可。

示例：

1. 在template下创建一个success.html
2. 在html中引入thymeleaf的命名空间

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

1. 创建一个Controller提供一个访问的方法

|  |
| --- |
| @RequestMapping(**"/success"**) **public** String hello(Model model){  model.addAttribute(**"hello"**,**"<h1>renliang</h1>"**);  **return "success"**; } |

1. 在thymeleaf模板中取值

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**title**>Title</**title**> </**head**> <**body**> <**div th:text="${hello}"**> </**div**> </**body**> </**html**> |

### Thymeleaf语法

#### **标准表达式语法**

1. **变量表达式**

变量表达式即OGNL表达式或Spring EL表达式(在Spring术语中也叫model attributes)。如下所示： ${session.user.name}

它们将以HTML标签的一个属性来表示：

|  |
| --- |
| <span th:text="${book.author.name}"> |

1. **选择(星号)表达式**

选择表达式很像变量表达式，不过它们用一个预先选择的对象来代替上下文变量容器(map)来执行，如下： \*{customer.name}

被指定的object由th:object属性定义：

|  |
| --- |
| <div th:object="${book}">   ...   <span th:text="\*{title}">...</span>   ...  </div> |

1. **文字国际化表达式**

文字国际化表达式允许我们从一个外部文件获取区域文字信息(.properties)，用Key索引Value，还可以提供一组参数(可选).

|  |
| --- |
| #{main.title} |

1. **URL表达式**

URL表达式指的是把一个有用的上下文或回话信息添加到URL，这个过程经常被叫做URL重写。不需要指定项目名字  
@{/order/list}

URL还可以设置参数：

@{/order/details(id=${orderId})}

让我们看这些表达式：

|  |
| --- |
| <form th:action="@{/createOrder}">  <a href="main.html" rel="external nofollow" th:href="@{/main}" rel="external n |

#### 表达式支持的语法

字面（Literals）

* 文本文字（Text literals）: 'one text', 'Another one!',…
* 数字文本（Number literals）: 0, 34, 3.0, 12.3,…
* 布尔文本（Boolean literals）: true, false
* 空（Null literal）: null
* 文字标记（Literal tokens）: one, sometext, main,…

文本操作（Text operations）

* 字符串连接(String concatenation): +
* 文本替换（Literal substitutions）: |The name is ${name}|

算术运算（Arithmetic operations）

* 二元运算符（Binary operators）: +, -, \*, /, %
* 减号（单目运算符）Minus sign (unary operator): -

布尔操作（Boolean operations）

* 二元运算符（Binary operators）:and, or
* 布尔否定（一元运算符）Boolean negation (unary operator):!, not

比较和等价(Comparisons and equality)

* 比较（Comparators）: >, <, >=, <= (gt, lt, ge, le)
* 等值运算符（Equality operators）:==, != (eq, ne)

条件运算符（Conditional operators）

If-then: (if) ? (then)

If-then-else: (if) ? (then) : (else)

Default: (value) ?: (defaultvalue)

示例代码：

'User is of type ' + (${user.isAdmin()} ? 'Administrator' : (${user.type} ?: 'Unknown'))

#### 常用的thymeleaf标签

| **关键字** | **功能介绍** | **案例** |
| --- | --- | --- |
| th:id | 替换id | <input th:id="'xxx' + ${collect.id}"/> |
| th:text | 文本替换 | <p th:text="${collect.description}">description</p> |
| th:utext | 支持html的文本替换 | <p th:utext="${htmlcontent}">conten</p> |
| th:object | 替换对象 | <div th:object="${session.user}"> |
| th:value | 属性赋值 | <input th:value="${user.name}" /> |
| th:onclick | 点击事件 | th:onclick="'getCollect()'" |
| th:each | 属性赋值 | tr th:each="user,userStat:${users}"> |
| th:if | 判断条件 | <a th:if="${userId == collect.userId}" > |
| th:unless | 和th:if判断相反 | <a th:href="@{/login}" rel="external nofollow" rel="external nofollow" rel="external nofollow" th:unless=${session.user != null}>Login</a> |
| th:href | 链接地址 | <a th:href="@{/login}" rel="external nofollow" rel="external nofollow" rel="external nofollow" th:unless=${session.user != null}>Login</a> /> |
| th:switch | 多路选择 配合th:case 使用 | <div th:switch="${user.role}"> |
| th:case | th:switch的一个分支 | <p th:case="'admin'">User is an administrator</p> |
| th:fragment | 布局标签，定义一个代码片段，方便其它地方引用 | <div th:fragment="alert"> |
| th:include | 布局标签，替换内容到引入的文件 | <head th:include="layout :: htmlhead" th:with="title='xx'"></head> /> |
| th:replace | 布局标签，替换整个标签到引入的文件 | <div th:replace="fragments/header :: title"></div> |
| th:selected | selected选择框 选中 | th:selected="(${xxx.id} == ${configObj.dd})" |
| th:src | 图片类地址引入 | <img class="img-responsive" alt="App Logo" th:src="@{/img/logo.png}" /> |
| th:action | 表单提交的地址 | <form action="subscribe.html" th:action="@{/subscribe}"> |

标签测试

模板：

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**title**>Title</**title**> </**head**> <**body**> <**div th:text="${hello}" th:id="${hello.toUpperCase()}"**>xxxx</**div**> <**input th:value="${user.getUsername()}"**> <**hr**> <**div th:object="${user}"**> <**span th:text="\*{username}"**></**span**>  </**div**>   <**a th:href="" th:if="${user.getAge() == 2}"** >年龄</**a**>  <**a th:class="${user.getAge() > 2}?'class1':'class2'"** >年龄</**a**>  <**p th:if="${user.score >= 60 and user.score < 85}"**>B</**p**> <**p th:if="${user.score < 60}"**>C</**p**> <**p th:if="${user.score > 85}"**>优秀</**p**>   <**span th:switch="${user.gender}"**>  <**p th:case="1"**>男</**p**>  <**p th:case="2"**>女</**p**> </**span**>   <**table**>   <**tr th:each="a,aState:${uList}"**>  <**td th:text="${a.username}"**></**td**>  <**td th:text="${a.password}"**></**td**>  <**td th:text="${aState.index}"**></**td**>  </**tr**> </**table**>  </**body**> </**html**> |

Controller中给数据

|  |
| --- |
| @RequestMapping(**"/success"**) **public** String hello(HttpServletRequest req, HttpSession httpSession, Model model){  model.addAttribute(**"hello"**,**"<h1>renliang</h1>"**);  User user = **new** User();  user.setPassword(**"111"**);  user.setUsername(**"renliang"**);  user.setAge(1);  user.setScore(78);  user.setGender(2);  List<User> uList = **new** ArrayList<>();  **for** (**int** i = 0; i < 10; i++){  User u = **new** User();  u.setUsername(**"renliang"**+i);  u.setPassword(**"111"**+i);   uList.add(u);  }   *// httpSession.setAttribute("user", user);* model.addAttribute(**"user"**, user);  model.addAttribute(**"uList"**, uList);  **return "success"**; } |

## Springboot整合springmvc

[https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/htmlsingle/#boot-features-spring-mvc](https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/htmlsingle/" \l "boot-features-spring-mvc)

学习springmvc和springboot的自动配置我们必须对springmvc的组件足够了解，起码知道怎么用。Springmvc的组件基本都被springboot来做了自动的配置。

### Springmvc的自动解管理

中央转发器

控制器

视图解析器

静态资源访问

消息转换器

格式化

静态资源管理

#### 中央转发器

Xml无需配置

|  |
| --- |
| <**servlet**>  <**servlet-name**>chapter2</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  <**load-on-startup**>1</**load-on-startup**> </**servlet**> <**servlet-mapping**>  <**servlet-name**>chapter2</**servlet-name**>  <**url-pattern**>/</**url-pattern**> </**servlet-mapping**> |

中央转发器被springboot自动接管，不再需要我们在web.xml中配置，我们现在的项目也不是web项目，也不存在web.xml,

**org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\**

#### 控制器

控制器Controller在springboot的注解扫描范围内自动管理。

#### 视图解析器自动管理

Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.

ContentNegotiatingViewResolver：组合所有的视图解析器的；

曾经的配置文件无需再配

|  |
| --- |
| <**bean id="de" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  <**property name="prefix" value="/WEB-INF/jsp/"**></**property**>  <**property name="suffix" value="\*.jsp"**></**property**> </**bean**> |

源码：

|  |
| --- |
| **public** ContentNegotiatingViewResolver viewResolver(BeanFactory beanFactory) {  ContentNegotiatingViewResolver resolver = **new** ContentNegotiatingViewResolver();  resolver.setContentNegotiationManager((ContentNegotiationManager)beanFactory.getBean(ContentNegotiationManager.**class**));  resolver.setOrder(-2147483648);  **return** resolver; } |

当我们做文件上传的时候我们也会发现**multipartResolver**是自动被配置好的

页面

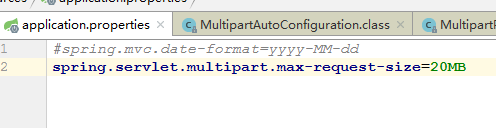
|  |
| --- |
| <**form action="/upload" method="post" enctype="multipart/form-data"**>  <**input name="pic" type="file"**>  <**input type="submit"**> </**form**> |

Controller

|  |
| --- |
| @ResponseBody @RequestMapping(**"/upload"**) **public** String upload(@RequestParam(**"pic"**)MultipartFile file, HttpServletRequest request){  String contentType = file.getContentType();  String fileName = file.getOriginalFilename();  */\*System.out.println("fileName-->" + fileName);  System.out.println("getContentType-->" + contentType);\*/  //String filePath = request.getSession().getServletContext().getRealPath("imgupload/");* String filePath = **"D:/imgup"**;  **try** {  **this**.*uploadFile*(file.getBytes(), filePath, fileName);  } **catch** (Exception e) {  *//* ***TODO: handle exception*** }   **return "success"**; }   **public static void** uploadFile(**byte**[] file, String filePath, String fileName) **throws** Exception {  File targetFile = **new** File(filePath);  **if**(!targetFile.exists()){  targetFile.mkdirs();  }  FileOutputStream out = **new** FileOutputStream(filePath+fileName);  out.write(file);  out.flush();  out.close(); } |

文件上传大小可以通过配置来修改

打开application.properties, 默认限制是10MB，我们可以任意修改

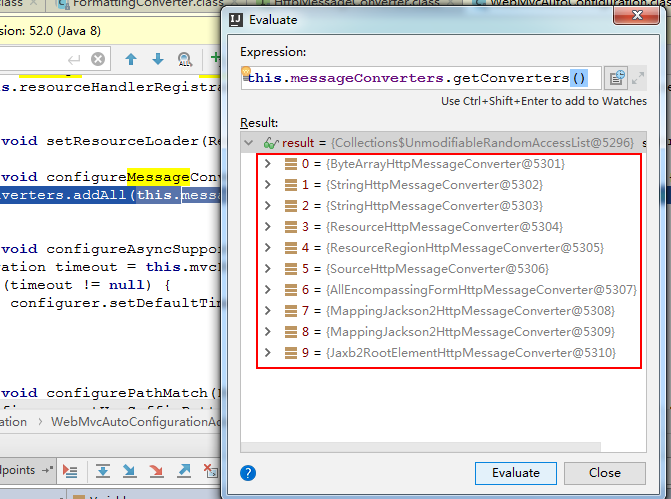


#### 静态资源访问

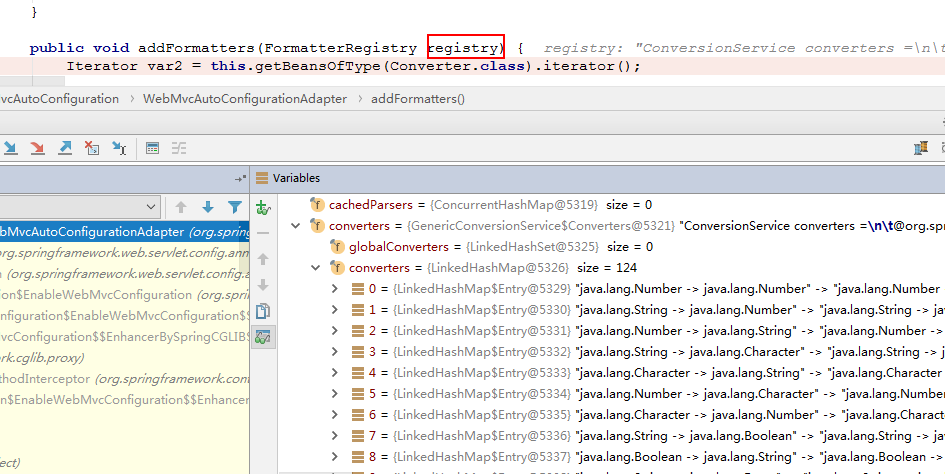
参见4.2

#### 消息转换和格式化

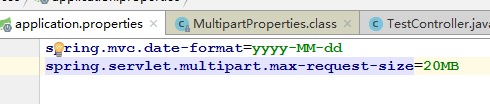
Springboot自动配置了消息转换器



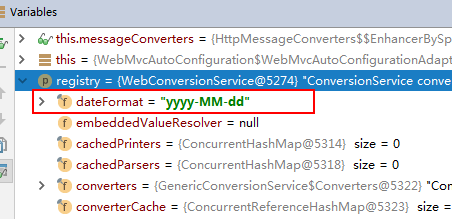
格式化转换器的自动注册



时间类型我们可以在这里修改

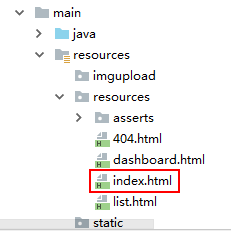


在配置文件中指定好时间的模式我们就可以输入了



#### 欢迎页面的自动配置

Springboot自动指定resources下的index.html



### Springboot扩展springmvc

在实际开发中springboot并非完全自动化，很多跟业务相关我们需要自己扩展，springboot给我提供了接口。

我们可以来通过实现WebMvcConfigurer接口来扩展

|  |
| --- |
| **public interface** WebMvcConfigurer {  **default void** configurePathMatch(PathMatchConfigurer configurer) {  }   **default void** configureContentNegotiation(ContentNegotiationConfigurer configurer) {  }   **default void** configureAsyncSupport(AsyncSupportConfigurer configurer) {  }   **default void** configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {  }   **default void** addFormatters(FormatterRegistry registry) {  }   **default void** addInterceptors(InterceptorRegistry registry) {  }   **default void** addResourceHandlers(ResourceHandlerRegistry registry) {  }   **default void** addCorsMappings(CorsRegistry registry) {  }   **default void** addViewControllers(ViewControllerRegistry registry) {  }   **default void** configureViewResolvers(ViewResolverRegistry registry) {  }   **default void** addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) {  }   **default void** addReturnValueHandlers(List<HandlerMethodReturnValueHandler> handlers) {  }   **default void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  }   **default void** extendMessageConverters(List<HttpMessageConverter<?>> converters) {  }   **default void** configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {  }   **default void** extendHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {  }   @Nullable  **default** Validator getValidator() {  **return null**;  }   @Nullable  **default** MessageCodesResolver getMessageCodesResolver() {  **return null**;  } } |

#### 在容器中注册视图控制器(请求转发)

创建一个MyMVCCofnig实现WebMvcConfigurer接口，实现一下addViewControllers方法，我们完成通过/tx访问，转发到success.html的工作

|  |
| --- |
| @Configuration **public class** MyMVCCofnig **implements** WebMvcConfigurer{    @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/tx"**).setViewName(**"success"**);  } |

#### 注册格式化器

用来可以对请求过来的日期格式化的字符串来做定制化。当然通过application.properties配置也可以办到。

|  |
| --- |
| @Override **public void** addFormatters(FormatterRegistry registry) {  registry.addFormatter(**new** Formatter<Date>() {  @Override  **public** String print(Date date, Locale locale) {  **return null**;  }  @Override  **public** Date parse(String s, Locale locale) **throws** ParseException {  **return new** SimpleDateFormat(**"yyyy-MM-dd"**).parse(s);  }  }); } |

#### 消息转换器扩展fastjson

在pom.xml中引入fastjson

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.47</**version**> </**dependency**> |

配置消息转换器，添加fastjson

|  |
| --- |
| @Override **public void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  FastJsonHttpMessageConverter fc = **new** FastJsonHttpMessageConverter();  FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  fc.setFastJsonConfig(fastJsonConfig);  converters.add(fc); } |

在实体类上可以继续控制

|  |
| --- |
| **public class** User {   **private** String **username**;   **private** String **password**;   **private int age**;   **private int score**;   **private int gender**;   @JSONField(format = **"yyyy-MM-dd"**)  **private** Date **date**; |

#### 拦截器注册

1. 创建拦截器

|  |
| --- |
| **public class** MyInterceptor **implements** HandlerInterceptor {  @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  System.***out***.println(**"前置拦截"**);  **return true**;  }  @Override  **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  System.***out***.println(**"后置拦截"**);  }  @Override  **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  System.***out***.println(**"最终拦截"**);  } } |

拦截器注册

|  |
| --- |
| @Override **public void** addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(**new** MyInterceptor())  .addPathPatterns(**"/\*\*"**)  .excludePathPatterns(**"/hello2"**); } |

## 配置嵌入式服务器

### 如何定制和修改Servlet容器的相关配置；

修改和server有关的配置（ServerProperties）；

|  |
| --- |
| server.port=8081  server.context‐path=/tx  server.tomcat.uri‐encoding=UTF‐8 |

### 注册Servlet三大组件【Servlet、Filter、Listener】

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。

1.servlet

|  |
| --- |
| //注册三大组件  @Bean  public ServletRegistrationBean myServlet(){  ServletRegistrationBean registrationBean = new ServletRegistrationBean(new  MyServlet(),"/myServlet");  return registrationBean;  } |

2. FilterRegistrationBean

|  |
| --- |
| @Bean  public FilterRegistrationBean myFilter(){  FilterRegistrationBean registrationBean = new FilterRegistrationBean();  registrationBean.setFilter(new MyFilter());  registrationBean.setUrlPatterns(Arrays.asList("/hello","/myServlet"));  return registrationBean;  } |

3. ServletListenerRegistrationBean

|  |
| --- |
| @Bean  public ServletListenerRegistrationBean myListener(){  ServletListenerRegistrationBean<MyListener> registrationBean = new  ServletListenerRegistrationBean<>(new MyListener());  return registrationBean;  } |

SpringBoot帮我们自动SpringMVC的时候，自动的注册SpringMVC的前端控制器；DispatcherServlet；

DispatcherServletAutoConfiguration中：

|  |
| --- |
| @Bean(name = DEFAULT\_DISPATCHER\_SERVLET\_REGISTRATION\_BEAN\_NAME)  @ConditionalOnBean(value = DispatcherServlet.class, name =  DEFAULT\_DISPATCHER\_SERVLET\_BEAN\_NAME)  public ServletRegistrationBean dispatcherServletRegistration(  DispatcherServlet dispatcherServlet) {  ServletRegistrationBean registration = new ServletRegistrationBean(  dispatcherServlet, this.serverProperties.getServletMapping());  //默认拦截： / 所有请求；包静态资源，但是不拦截jsp请求； /\*会拦截jsp  //可以通过server.servletPath来修改SpringMVC前端控制器默认拦截的请求路径  registration.setName(DEFAULT\_DISPATCHER\_SERVLET\_BEAN\_NAME);  registration.setLoadOnStartup(  this.webMvcProperties.getServlet().getLoadOnStartup());  if (this.multipartConfig != null) {  registration.setMultipartConfig(this.multipartConfig);  }  return registration;  } |

## 使用外置的Servlet容器

嵌入式Servlet容器：应用打成可执行的jar

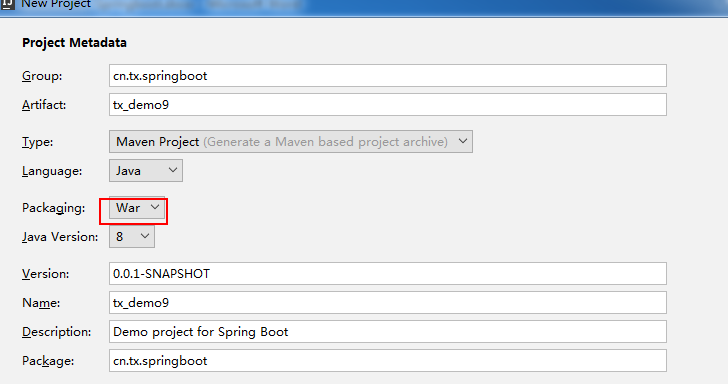
优点：简单、便携；

缺点：默认不支持JSP、优化定制比较复杂.；

外置的Servlet容器：外面安装Tomcat---应用war包的方式打包；

步骤

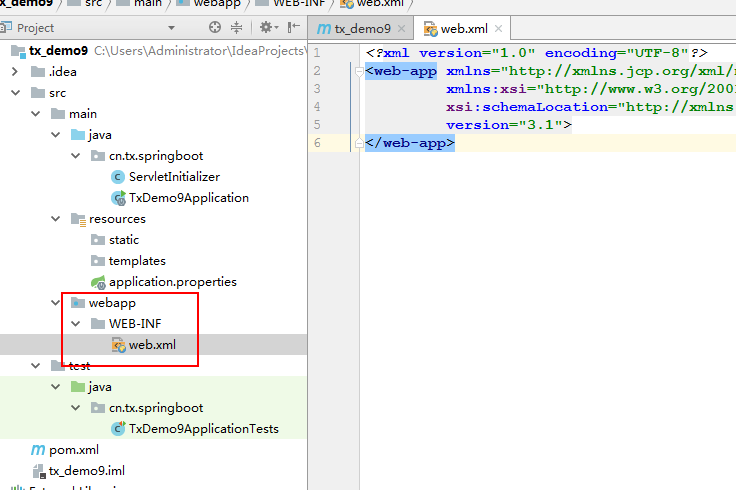
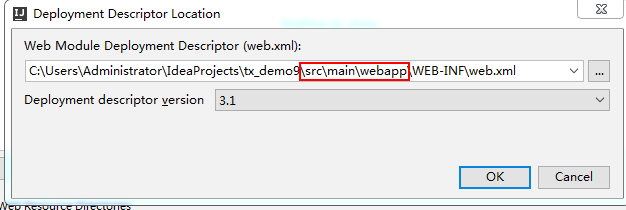
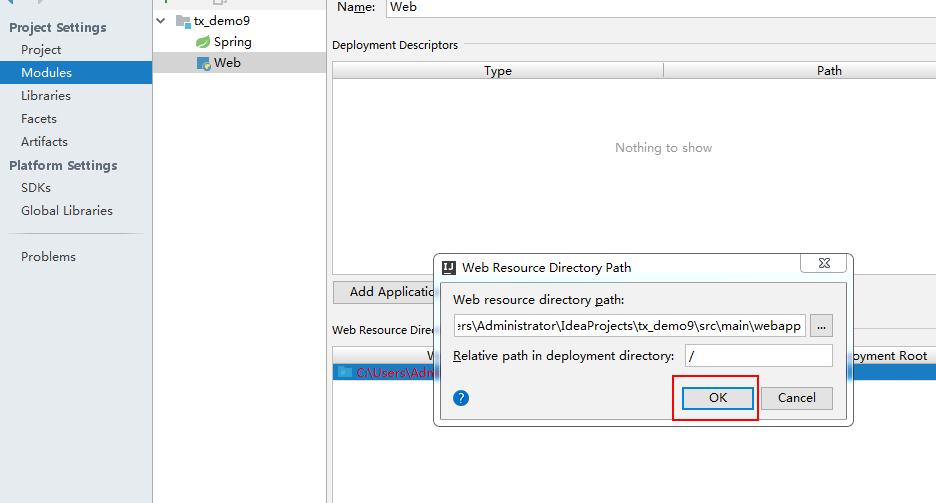
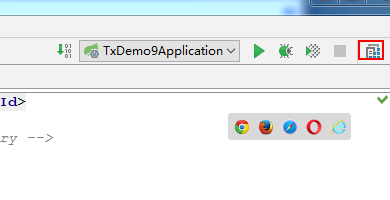
1）、必须创建一个war项目；（利用idea创建好目录结构）



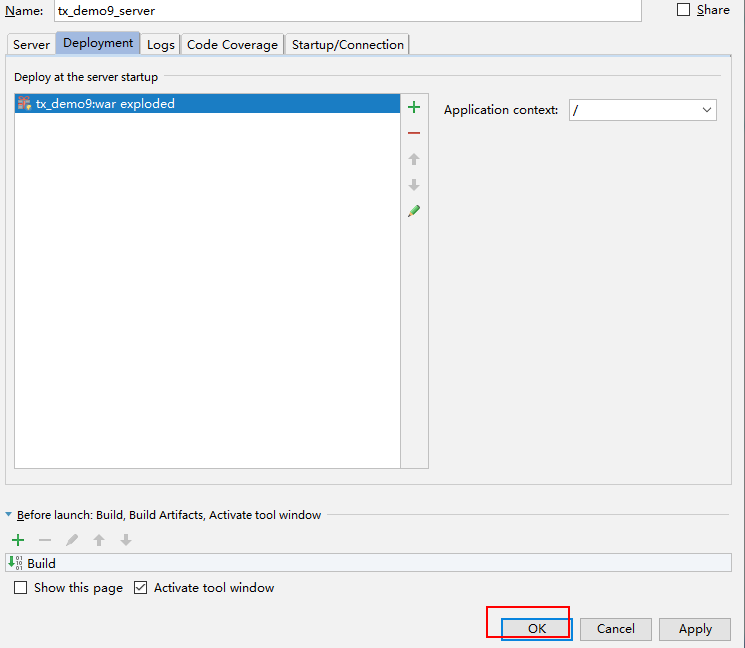
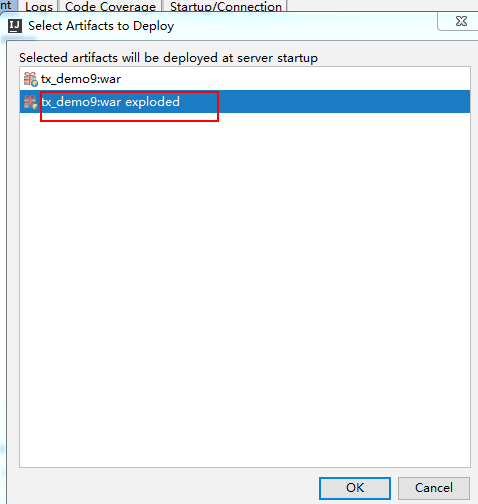
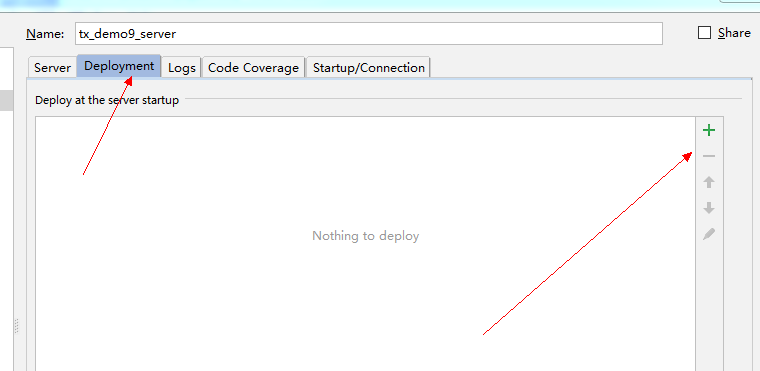
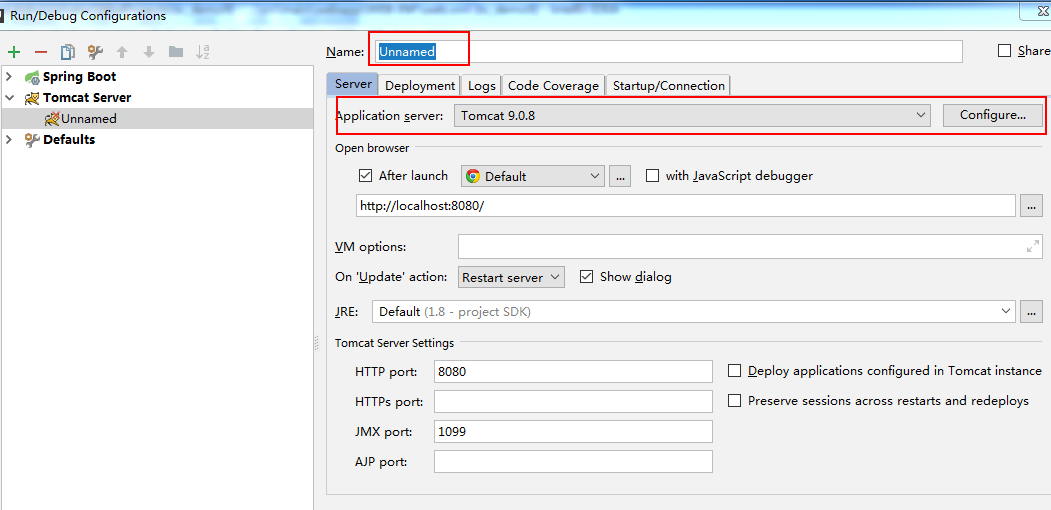
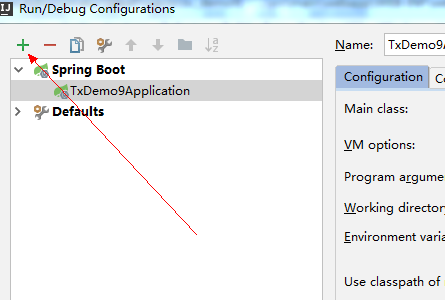
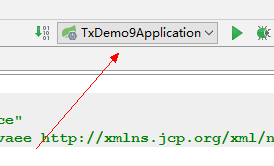
2）、将嵌入式的Tomcat指定为provided；

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring‐boot‐starter‐tomcat</artifactId>  <scope>provided</scope>  </dependency> |

3）配置项目的目录结构



4）部署Tomcat



3）、必须编写一个SpringBootServletInitializer的子类，并调用configure方法

|  |
| --- |
| public class ServletInitializer extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  //传入SpringBoot应用的主程序  return application.sources(SpringBoot04WebJspApplication.class);  }  } |

4）、启动服务器就可以使用；

原理

jar包：执行SpringBoot主类的main方法，启动ioc容器，创建嵌入式的Servlet容器；

war包：启动服务器，服务器启动SpringBoot应用【SpringBootServletInitializer】，启动ioc容器；

# Springboot数据层开发

## 数据源自动管理

引入jdbc的依赖和springboot的应用场景

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-dbcp2</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> |

让我们使用yaml方式配置，创建application.yaml

|  |
| --- |
| **spring:  datasource:  username:** root  **password:** root  **url:** jdbc:mysql://localhost:3306/boot\_demo  **driver-class-name:** com.mysql.jdbc.Driver  **type:** com.zaxxer.hikari.HikariDataSource |

在默认情况下， 数据库连接可以使用DataSource池进行自动配置

* 如果Hikari可用， Springboot将使用它。
* 如果Commons DBCP2可用， 我们将使用它。

我们可以自己指定数据源配置，通过type来选取使用哪种数据源

|  |
| --- |
| **spring:  datasource:  username:** root  **password:** root  **url:** jdbc:mysql://localhost:3306/boot\_demo  **driver-class-name:** com.mysql.jdbc.Driver  **type:** com.zaxxer.hikari.HikariDataSource  *# type: org.apache.commons.dbcp2.BasicDataSource* |

## 配置druid数据源

引入druid的依赖

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.9</**version**> </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.15</**version**> </**dependency**> |

修改spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

在application.yaml中加入

|  |
| --- |
| **spring:  datasource:  username:** root  **password:** root  **url:** jdbc:mysql://localhost:3306/boot\_demo  **driver-class-name:** com.mysql.jdbc.Driver  **type:** com.alibaba.druid.pool.DruidDataSource  **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 |

创建数据源注册类

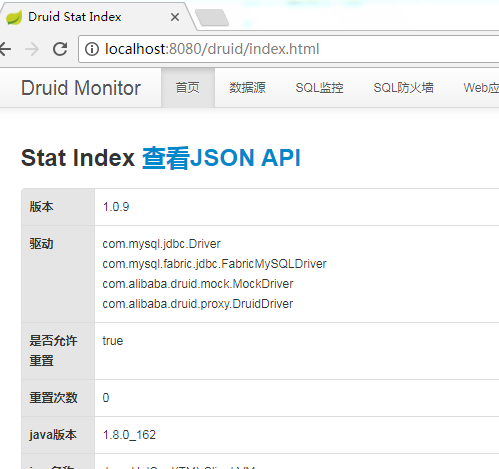
|  |
| --- |
| @Configuration **public class** DruidConfig {   @ConfigurationProperties(prefix = **"spring.datasource"**)  @Bean  **public** DataSource dataSource(){  **return new** DruidDataSource();  } } |

配置druid运行期监控

|  |
| --- |
| @Configuration **public class** DruidConfig {   @ConfigurationProperties(prefix = **"spring.datasource"**)  @Bean  **public** DataSource dataSource(){  **return new** DruidDataSource();  }   @Bean  **public** ServletRegistrationBean statViewServlet(){  ServletRegistrationBean bean = **new** ServletRegistrationBean(**new** StatViewServlet(),  **"/druid/\*"**);  Map<String,String> initParams = **new** HashMap<>();  initParams.put(**"loginUsername"**,**"root"**);  initParams.put(**"loginPassword"**,**"root"**);  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;  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;  } } |

打开监控页面

<http://localhost:8080/druid>



## springboot整合jdbcTemplate

在数据源建表

|  |
| --- |
| SET FOREIGN\_KEY\_CHECKS=0;  -- ----------------------------  -- Table structure for tx\_user  -- ----------------------------  DROP TABLE IF EXISTS `tx\_user`;  CREATE TABLE `tx\_user` (  `username` varchar(10) DEFAULT NULL,  `userId` int(10) NOT NULL,  `password` varchar(10) DEFAULT NULL,  PRIMARY KEY (`userId`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

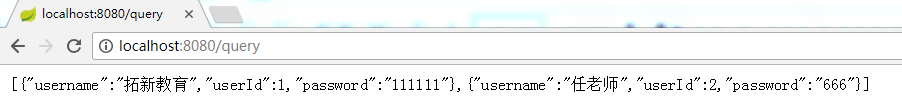


创建Controller

|  |
| --- |
| @Controller **public class** TestController {    @Autowired  JdbcTemplate **jdbcTemplate**;   @ResponseBody  @RequestMapping(**"/query"**)  **public** List<Map<String, Object>> query(){  List<Map<String, Object>> maps = **jdbcTemplate**.queryForList(**"SELECT** *\** **FROM tx\_user"**);  **return** maps;  }  } |

启动springboot访问

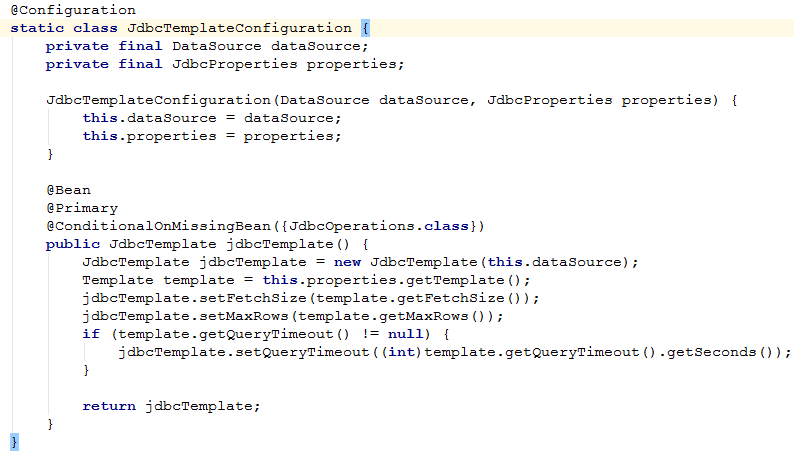
<http://localhost:8080/query>



Springboot中提供了JdbcTemplateAutoConfiguration的自动配置

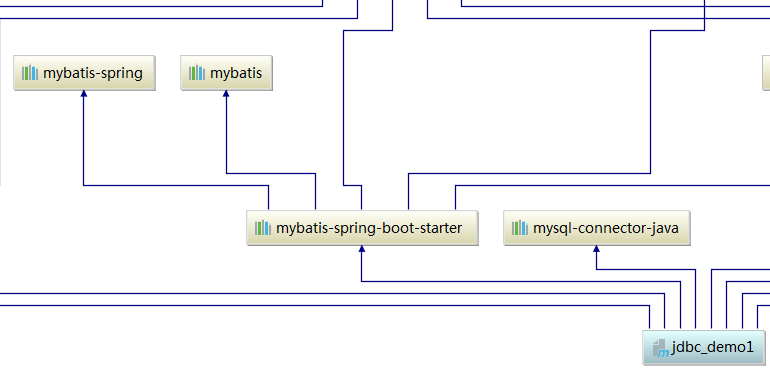
**org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\**

JdbcTemplateAutoConfiguration源码：



## Springboot整合mybatis注解版

|  |
| --- |
| <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.1</**version**> </**dependency**> |



步骤：

1）、配置数据源相关属性（见上一节Druid）

2）、给数据库建表

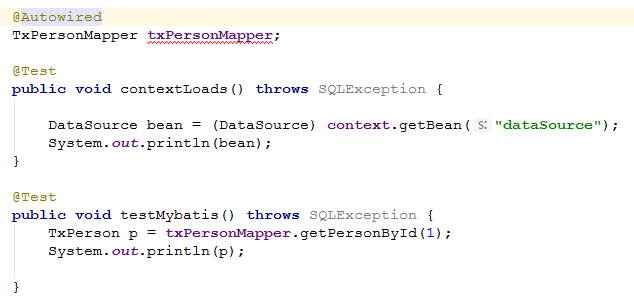
3）、创建JavaBean

|  |
| --- |
| **public class** TxPerson {    **private int pid**;   **private** String **pname**;   **private** String **addr**;   **private int gender**;   **private** Date **birth**; |

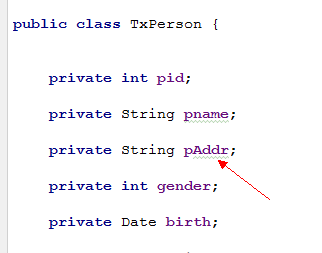
4）创建Mapper

|  |
| --- |
| @Mapper **public interface** TxPersonMapper {    @Select(**"select \* from tx\_person"**)  **public** List<TxPerson> getPersons();    @Select(**"select \* from tx\_person t where t.pid = #{id}"**)  **public** TxPerson getPersonById(**int** id);   @Options(useGeneratedKeys =**true**, keyProperty = **"pid"**)  @Insert(**"insert into tx\_person(pid, pname, addr,gender, birth)"** +  **" values(#{pid}, #{pname}, #{addr},#{gender}, #{birth})"**)  **public void** insert(TxPerson person);   @Delete(**"delete from tx\_person where pid = #{id}"**)  **public void** update(**int** id);  } |

单元测试



解决驼峰模式和数据库中下划线不能映射的问题。

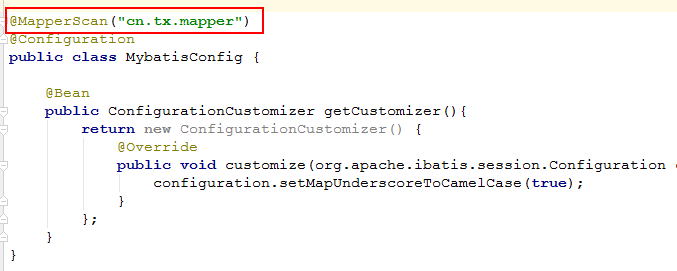


|  |
| --- |
| @Configuration **public class** MybatisConfig {   @Bean  **public** ConfigurationCustomizer getCustomizer(){  **return new** ConfigurationCustomizer() {  @Override  **public void** customize(org.apache.ibatis.session.Configuration configuration) {  configuration.setMapUnderscoreToCamelCase(**true**);  }  };  } } |

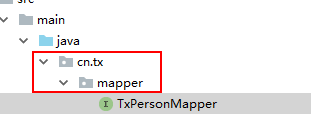
查询结果

|  |
| --- |
| TxPerson{pid=1, pname='张三', pAddr='北京', gender=1, birth=Thu Jun 14 00:00:00 CST 2018} |

我们同样可以在mybatis的接口上不加@Mapper注解，通过扫描器注解来扫描



Mapper接口存放在cn.tx.mapper下



## Springboot整合mybatis配置文件

创建sqlMapConfig.xml配置文件

|  |
| --- |
| *<?***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**> |

创建映射文件PersonMapper.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="cn.tx.mapper.TxPersonMapper"**>  <**select id="getPersons" resultType="TxPerson"**>  select *\** from tx\_person  </**select**> </**mapper**> |

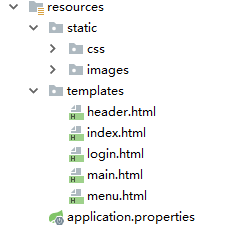
在application.yaml中配置mybatis的信息

|  |
| --- |
| **mybatis:  config-location:** classpath:mybatis/sqlMapConfig.xml  **mapper-locations:** classpath:mybatis/mapper/\*.xml  **type-aliases-package:** cn.tx.springboot.jdbc\_demo1 |

# Springboot开发企业信息管理系统

## 引入资源

**1.创建项目引入页面原型**



**2.引入maven的依赖和starters**

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.2</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.9</version>  </dependency>  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>1.2.15</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-configuration-processor</artifactId>  <optional>true</optional>  </dependency>  <dependency>  <groupId>org.webjars</groupId>  <artifactId>jquery</artifactId>  <version>3.3.1</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <optional>true</optional>  </dependency> |

## thymeleaf模板处理

创建mvc的配置类TxMvcConfig

|  |
| --- |
| @Configuration @MapperScan(**"cn.tx.springboot.mapper"**) **public class** TxMvcConfig **implements** WebMvcConfigurer{   @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/toLogin"**).setViewName(**"login"**);  registry.addViewController(**"/header"**).setViewName(**"header"**);  registry.addViewController(**"/index"**).setViewName(**"index"**);  registry.addViewController(**"/menu"**).setViewName(**"menu"**);  registry.addViewController(**"/add"**).setViewName(**"add"**);  }   } |

## 创建数据库环境

**1.创建数据库执行脚本**

|  |
| --- |
| SET FOREIGN\_KEY\_CHECKS=0;  -- ----------------------------  -- Table structure for my\_user  -- ----------------------------  DROP TABLE IF EXISTS `my\_user`;  CREATE TABLE `my\_user` (  `pid` int(11) NOT NULL AUTO\_INCREMENT,  `username` varchar(255) DEFAULT NULL,  `password` varchar(255) DEFAULT NULL,  `p\_addr` varchar(255) DEFAULT NULL,  `gender` int(11) DEFAULT NULL,  `birth` date DEFAULT NULL,  PRIMARY KEY (`pid`)  ) ENGINE=InnoDB AUTO\_INCREMENT=5 DEFAULT CHARSET=utf8;  -- ----------------------------  -- Records of my\_user  -- ----------------------------  INSERT INTO `my\_user` VALUES ('1', 'zhangsan', '123', '北京', '1', '2020-06-14'); |

1. **配置druid数据源**

|  |
| --- |
| **spring:  datasource:  username:** root  **password:** root  **url:** jdbc:mysql://localhost:3306/boot\_demo  **driver-class-name:** com.mysql.jdbc.Driver  **type:** com.alibaba.druid.pool.DruidDataSource  **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 |

1. **创建数据源的配置类**

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

## 编写数据层

**1.创建mybatis的配置类MybatisConfig**

|  |
| --- |
| @Configuration **public class** MybatisConfig {    @Bean  **public** ConfigurationCustomizer getCustomizer(){  **return new** ConfigurationCustomizer() {  @Override  **public void** customize(org.apache.ibatis.session.Configuration configuration) {  configuration.setMapUnderscoreToCamelCase(**true**);  }  };  }  } |

**2.创建数据层UserMapper**

|  |
| --- |
| **public interface** MyUserMapper {    @Options(useGeneratedKeys = **true**, keyProperty = **"pid"**)  @Insert(**"insert into my\_user(pid, username, password, p\_addr, gender, birth)values("** +  **"#{pid}, #{username}, #{password}, #{pAddr}, #{gender}, #{birth})"**)  **public void** insert(MyUser user) ;   @Select(**"select \* from my\_user"**)  **public** List<MyUser> selectUsers();   @Select(**"select \* from my\_user t where t.username = #{username} and t.password = #{password}"**)  **public** MyUser selectUsersById(Map<String, String> map);    @Select(**"select \* from my\_user t where t.pid = #{pid}"**)  **public** MyUser selectUsersById1(**int** userId);   @Update(**"update my\_user set "** +  **"username = #{username},"** +  **"password=#{password}, "** +  **"p\_addr= #{pAddr}, "** +  **"gender=#{gender}, "** +  **"birth=#{birth} "** +  **"where pid = #{pid}"**)  **public void** update(MyUser user) ;   @Delete(**"delete from my\_user where pid = #{pid}"**)  **public void** delete(**int** pid) ;  } |

## 编写Service

|  |
| --- |
| **public interface** MyUserService {   **public void** insert(MyUser user) ;   **public void** update(MyUser user) ;   **public** List<MyUser> selectUsers();   **public** MyUser selectUsersById(Map<String, String> map);   **public** MyUser selectUsersById1(**int** userId);   **public void** delete(**int** pid) ;   } |

实现类略（参见视频）

## 页面处理

1. main页面处理

|  |
| --- |
| <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**> <**meta http-equiv="Content-Type" content="text/html; charset=utf-8"** /> <**title**>拓新教育首页</**title**> </**head**>  <**frameset cols="\*" rows="136, \*" id="frame\_main" border="0"**>  <**frame src="header" noresize="noresize" name="header"**>    <**frameset cols="240, \*"**>  <**frame th:src="@{/menu}" name="menu"** />  <**frame th:src="@{/main}" name="main"**>  </**frameset**> </**frameset**> |



|  |
| --- |
| <**a th:href="@{/add}"**>添加</**a**> <**table class="mt" align="center"**>  <**tr** >  <**td** >用户名</**td**>  <**td** >密码</**td**>  <**td** >地址</**td**>  <**td** >生日</**td**>  <**td** >操作</**td**>  </**tr**>  <**tr th:each="user:${users}"**>  <**td th:text="${user.username}"**></**td**>  <**td th:text="${user.password}"**></**td**>  <**td th:text="${user.pAddr}"**></**td**>  <**td th:text="${#calendars.format(user.birth,'yyyy-MM-dd')}"**></**td**>  <**td**>  <**a th:href="@{/getUser(pid=${user.pid})}"** >修改</**a**>  <**a th:href="@{/delete(pid=${user.pid})}"** >删除</**a**>  </**td**>  </**tr**> </**table**> |

2.添加页面处理

引入thymeleaf命名空间后

|  |
| --- |
| <**form th:action="@{/save}" method="post"**>  <**table class="mt" align="center"**>  <**tr** >  <**td** >用户名</**td**>  <**td**><**input name="username" type="text"**/></**td**>  </**tr**>  <**tr** >  <**td** >密码</**td**>  <**td**><**input name="password" type="password"**/></**td**>   </**tr**>  <**tr** >  <**td** >地址</**td**>  <**td**><**input name="pAddr" type="text"**/></**td**>  </**tr**>  <**tr** >  <**td** >生日</**td**>  <**td**><**input name="birth" type="text"**/></**td**>   </**tr**>  <**tr** >  <**td** >性别</**td**>  <**td**><**input name="gender" type="text"**/></**td**>  </**tr**>  <**tr** >  <**td** ></**td**>  <**td**><**input type="submit"**/></**td**>  </**tr**>  </**table**>  </**form**> |

2.修改页面处理

|  |
| --- |
| <**form th:action="@{/update}" method="post"**>  <**input type="hidden" name="pid" th:value="${user.pid}"**/>  <**table class="mt" align="center"**>  <**tr** >  <**td** >用户名</**td**>  <**td**><**input name="username" type="text" th:value="${user.username}"**/></**td**>  </**tr**><**tr** >  <**td** >密码</**td**>  <**td**><**input name="password" type="password" th:value="${user.password}"**/></**td**>  </**tr**><**tr** >  <**td** >地址</**td**>  <**td**><**input name="pAddr" type="text" th:value="${user.pAddr}"**/></**td**>  </**tr**>  <**tr** >  <**td** >地址</**td**>  <**td**><**input name="gender" type="text" th:value="${user.gender}"**/></**td**>  </**tr**>  <**tr** >  <**td** >生日</**td**>  <**td**><**input name="birth" type="text" th:value="${#calendars.format(user.birth, 'yyyy-MM-dd')}"**/></**td**>  </**tr**><**tr** >  <**td** ></**td**>  <**td**><**input type="submit"**/></**td**>  </**tr**>  </**table**>  </**form**> |

## 用户登录

|  |
| --- |
| <!DOCTYPE **html** PUBLIC **"-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**> <**meta http-equiv="Content-Type" content="text/html; charset=utf-8"** /> <**title**>用户登录</**title**> *<!--<link href="css/login.css" rel="stylesheet" type="text/css" />-->* <**link th:href="@{/css/login.css}" rel="stylesheet" type="text/css"** /> <**script th:src="@{/webjars/jquery/3.3.1/jquery.js}"**></**script**> <**script**>  **function** *login1*() {  **$**(**"form"**).submit();  }  </**script**> </**head**> <**body**>   <**div id="login"**>    <**div id="top"**>  <**div id="top\_left"**><**img src="images/login\_03.gif"** /></**div**>  <**div id="top\_center"**></**div**>  </**div**>  <**form th:action="@{/login}" method="post"**>  <**div id="center"**>  <**div id="center\_left"**></**div**>  <**div id="center\_middle"**>  <**div style="text-align**: **center**; **color**: **red" th:text="${tip}"**></**div**>  <**div id="user"**>用 户  <**input type="text" name="username"** />  </**div**>  <**div id="password"**>密 码  <**input type="password" name="password"** />  </**div**>  <**div id="btn"**>  <**a href="#" onclick="***login1*()**"**>登录</**a**>  <**a href="#"**>清空</**a**>  </**div**>  </**div**>  <**div id="center\_right"**></**div**>  </**div**>  </**form**>  <**div id="down"**>  <**div id="down\_left"**>  <**div id="inf"**>  <**span class="inf\_text"**>版本信息</**span**>  <**span class="copyright"**>拓薪管理信息系统 v2.0</**span**>  </**div**>  </**div**>  <**div id="down\_center"**></**div**>   </**div**>  </**div**> </**body**> </**html**> |

登录的Controller逻辑

|  |
| --- |
| @PostMapping(**"/login"**) **public** String login(HttpSession session, String username, String password, Model model) {  Map<String, String> map = **new** HashMap<String, String>();  map.put(**"username"**, username);  map.put(**"password"**, password);  MyUser user = **userService**.selectUsersById(map);  **if** (user != **null**){  session.setAttribute(**"user"**, user);  **return "redirect:index"**;  }**else**{  model.addAttribute(**"tip"**,**"用户名或者密码错误"**);  **return "login"**;  } } |

Header模板页面

|  |
| --- |
| <**div id="main"**>  <**div id="welcome"**>欢迎你回来  <**span th:text="${session.user.username}" th:if="${not #strings.isEmpty(session.user)}"**></**span**>  <**a target="\_top" th:text="请登录" th:href="@{/toLogin}" th:if="${#strings.isEmpty(session.user)}"**></**a**>  <**img src="images/clock.gif"** /> 学习是最好的投资</**div**>  <**div id="adminop"**>  <**ul**>  <**li**><**a href="#"**>站点首页</**a**></**li**>  <**li**><**a href="javascript:*parent*.location.reload();"**>管理首页</**a**></**li**>  <**li**><**a href="javascript:*parent*.location.reload();"**>退出管理</**a**></**li**>  <**li**><**a href="#"**>站点首页</**a**></**li**>  </**ul**>  </**div**> </**div**> |

拦截器：

|  |
| --- |
| public class LoginInterceptor implements HandlerInterceptor {  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  String username = (String)request.getSession().getAttribute("username");  if(!StringUtils.isEmpty(username))  {  return true;  }  response.sendRedirect(request.getContextPath()+"/toLogin");  return false;  }  } |

拦截器的注册

|  |
| --- |
| @Override  public void addInterceptors(InterceptorRegistry registry) {  List<String> excludePatterns = new ArrayList<String>();//拦截排除  excludePatterns.add("/css/\*\*");  excludePatterns.add("/images/\*\*");  excludePatterns.add("/toLogin");  excludePatterns.add("/login");  registry.addInterceptor(new LoginInterceptor()).addPathPatterns("/\*\*")//拦截所有  .excludePathPatterns(excludePatterns);  } |