

讲师：任亮

# Spring Boot 基础篇

## Spring Boot 简介

Spring Boot 是所有基于 Spring 开发的项目的起点。SpringBoot其实不是什么新的框架，它默认配置了很多框架的使用方式，就像maven整合了所有的jar包，spring boot整合了很多的技术，提供 了JavaEE的大整合。

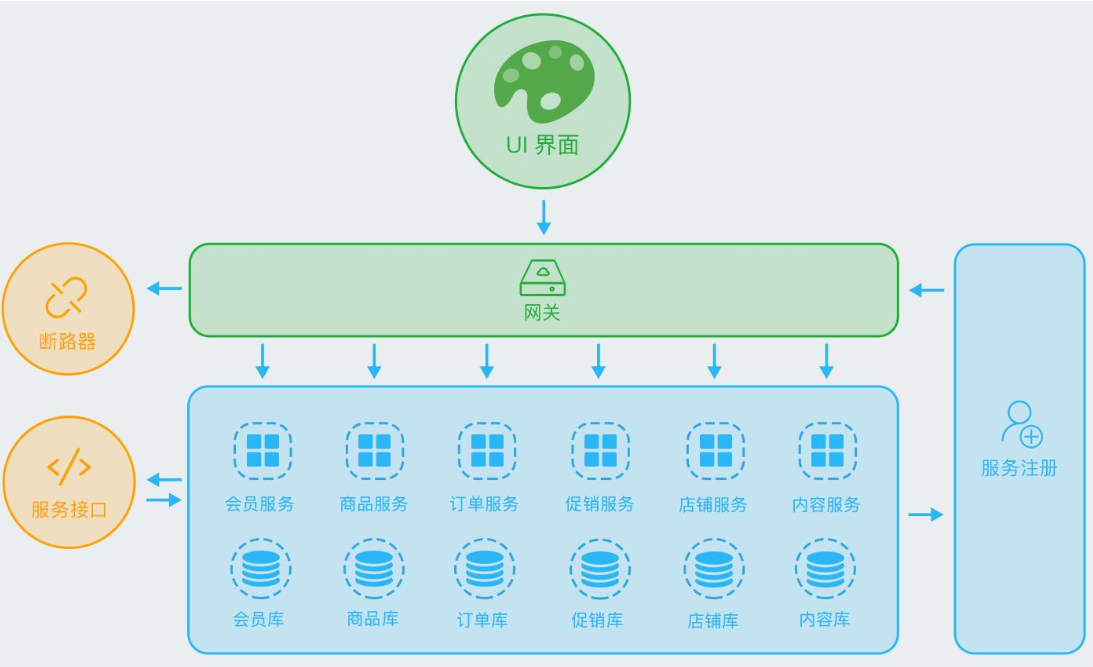
### springboot的学习铺垫和要求

1. 对spring的IOC有深入的理解，springboot是基于spring的。
2. 深入掌握spring的注解开发。传统老程序员对注解认识偏少。
3. 最好有ssm项目的开发经验，因为springboot是整合了javaEE的技术。
4. 有较好的源码学习经验。
5. 有idea和eclipse的使用经验，本次课程使用idea。
6. 在国内IT公司是一个主要的招聘要求。

### 、SpringBoot主要特性

spring官方的网站：<https://spring.io/>

1. SpringBoot Starter：他将常用的依赖分组进行了整合，将其合并到一个依赖中，这样就可以一次性添加到项目的Maven或Gradle构建中；
2. 使编码变得简单，SpringBoot采用 JavaConfig的方式对Spring进行配置，并且提供了大量的注解，极大的提高了工作效率。
3. 自动配置：SpringBoot的自动配置特性利用了Spring对条件化配置的支持，合理地推测应用所需的bean并自动化配置他们；
4. 使部署变得简单，SpringBoot内置了三种Servlet容器，Tomcat，Jetty,undertow.我们只需要一个Java的运行环境就可以跑SpringBoot的项目了，SpringBoot的项目可以打成一个jar包。
5. 现在流行微服务与分布式系统，springboot就是一个非常好的微服务开发框架，你可以使用它快速的搭建起一个系统。同时，你也可以使用spring cloud（Spring Cloud是一个基于Spring Boot实现的云应用开发工具）来搭建一个分布式的架构。



### springboot缺点

1. 将现有或传统的Spring Framework项目转换为Spring Boot应用程序是一个非常困难和耗时的过程。它仅适用于全新Spring项目。
2. 使用简单，学习成本高，精通难。

## 环境准备

jdk1.8：Spring Boot 推荐jdk1.8及以上；

maven3.x：maven 3.3以上版本；

IntelliJIDEA2019，不要使用2017

SpringBoot 2.3.0.RELEASE

### MAVEN设置

1. **配置阿里云镜像**

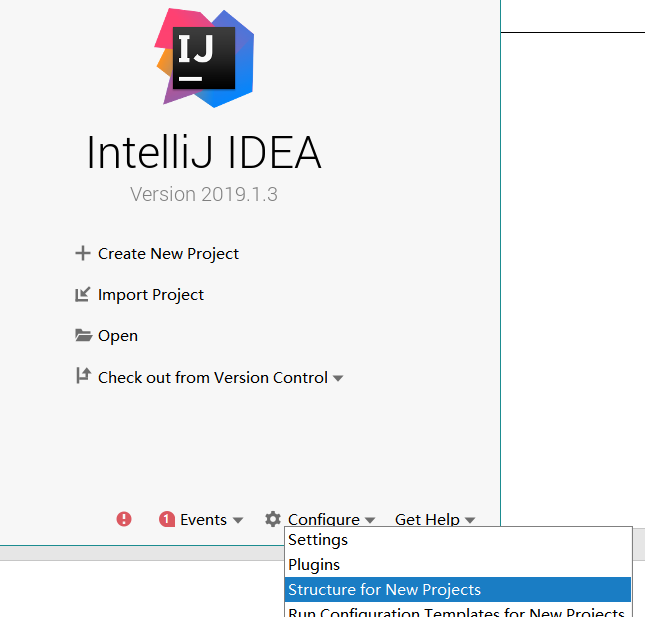
|  |
| --- |
| <mirror>  <id>nexus-aliyun</id>  <mirrorOf>central</mirrorOf>  <name>Nexus aliyun</name>  <url>http://maven.aliyun.com/nexus/content/groups/public</url>  </mirror> |

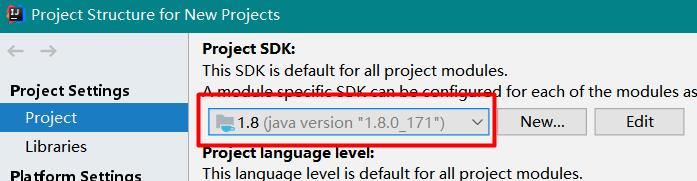
1. **给maven 的settings.xml配置文件的profiles标签添加**

|  |
| --- |
| <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> |

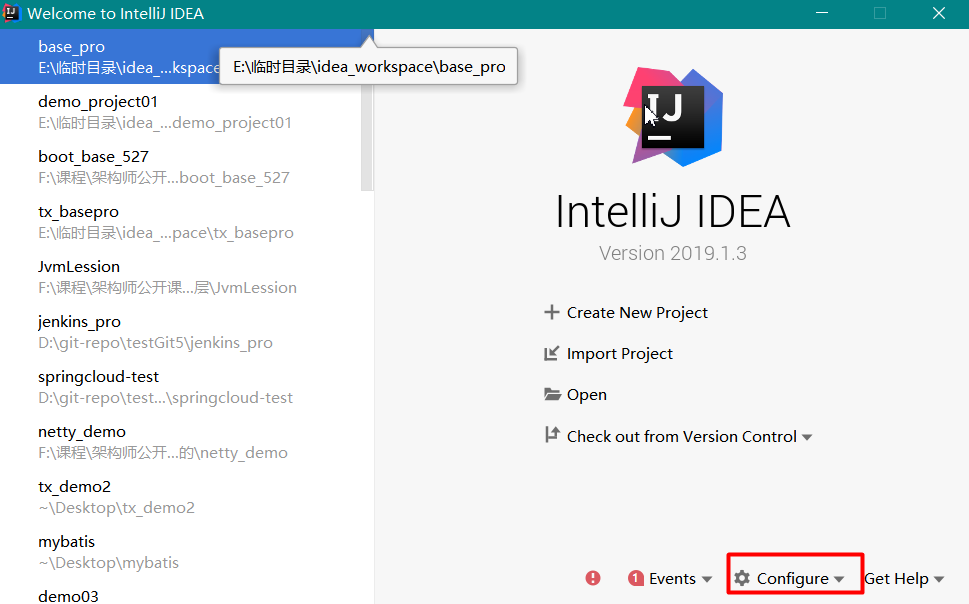
### IDEA设置

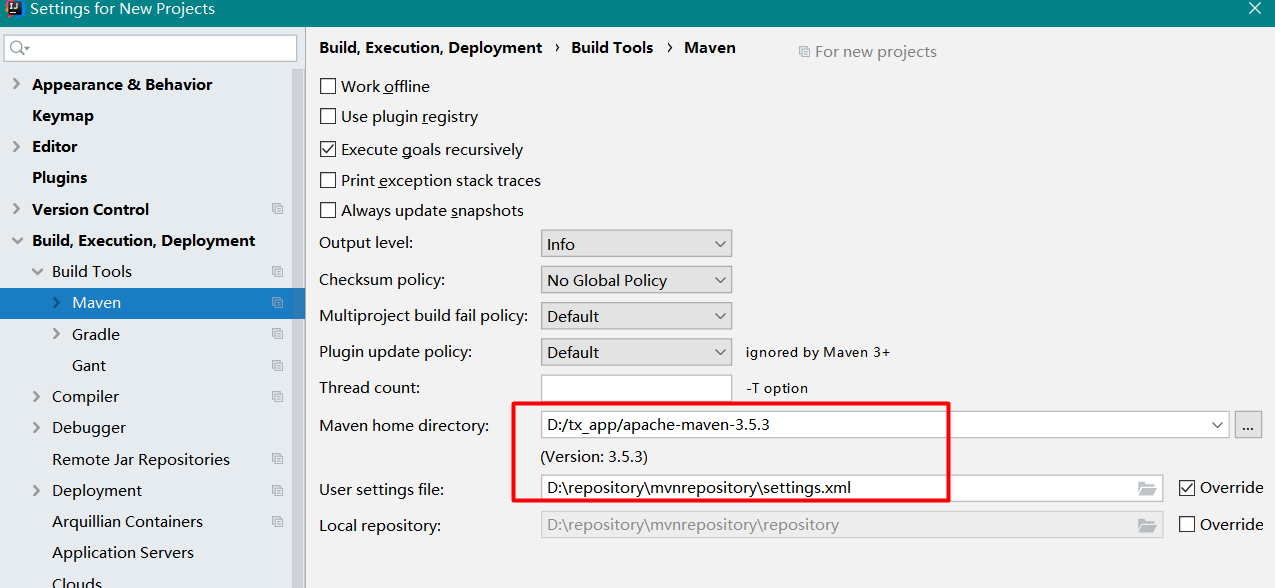
1. 指定jdk环境





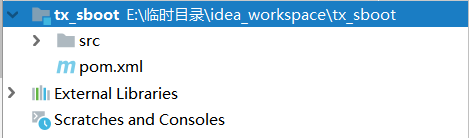
1. 指定maven环境





## Spring Boot HelloWorld

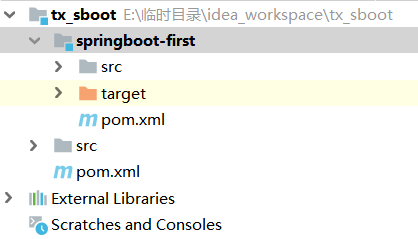
### 创建一个maven父工程tx\_sboot（pom）



pom.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>   <**groupId**>cn.tx.springboot</**groupId**>  <**artifactId**>tx\_sboot</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  <**packaging**>pom</**packaging**>    <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.3.0.RELEASE</**version**>  </**parent**>    <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  </**dependencies**> </**project**> |

### 在父工程下创建springboot-first（jar）



### 创建测试Controller

|  |
| --- |
| @RestController **public class** TestController {   @RequestMapping(**"hello"**)  **public** String hello(){  **return "hello"**;  } } |

### 创建一个springboot启动类

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

启动并且测试：<http://localhost:8080/hello>

成功。

### 在父工程tx\_sboot中加入构建依赖

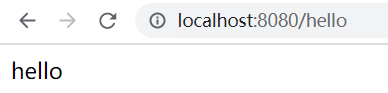
|  |
| --- |
| <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**> </**build**> |

我们可以把springboot工程打成可执行的jar

|  |
| --- |
| mvn -f springboot-first clean package |

打完jar包后，我们切入到对应的jar包里面执行





### 第一个例子做完的疑问。

1. starter是什么？我们何如去使用这些starter？
2. Tomcat的内嵌是如何完成
3. 使用了web对应的starter，springmvc是如何自动装配？
4. 我们如何来配置我们自定义话的相关内容。

## 默认扫描器basepackage

springboot的主启动类所在的package就是扫描器的basepackage

如图所示com.example.myapplication就是我们的扫描器中basepackge

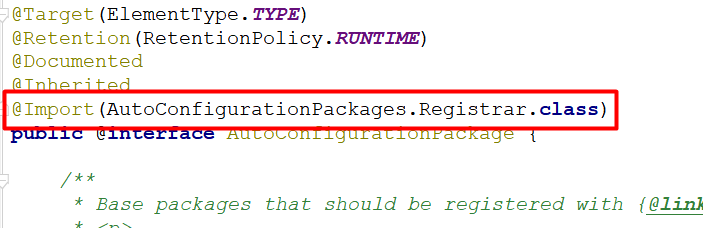
|  |
| --- |
| com  +- example  +- myapplication  +- Application.java  |  +- customer  | +- Customer.java  | +- CustomerController.java  | +- CustomerService.java  | +- CustomerRepository.java  |  +- order  +- Order.java  +- OrderController.java  +- OrderService.java  +- OrderRepository.java |

源码解析

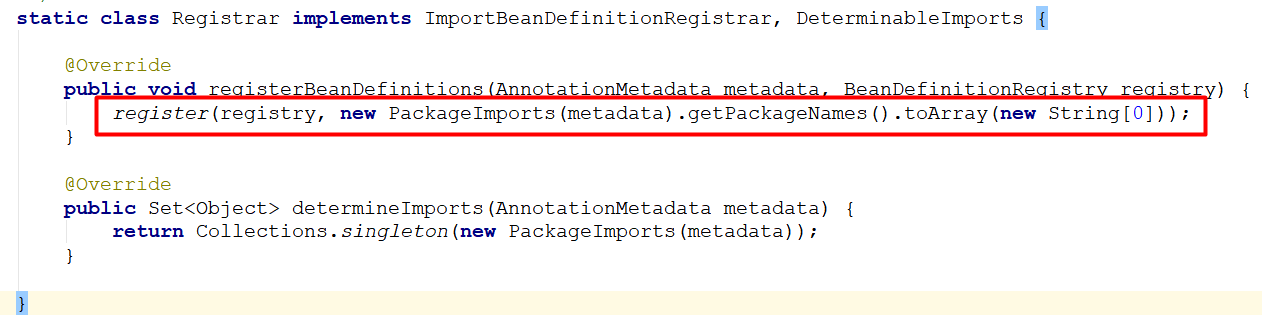
@AutoConfigurationPackage

自动配置包负责basepackge的注册

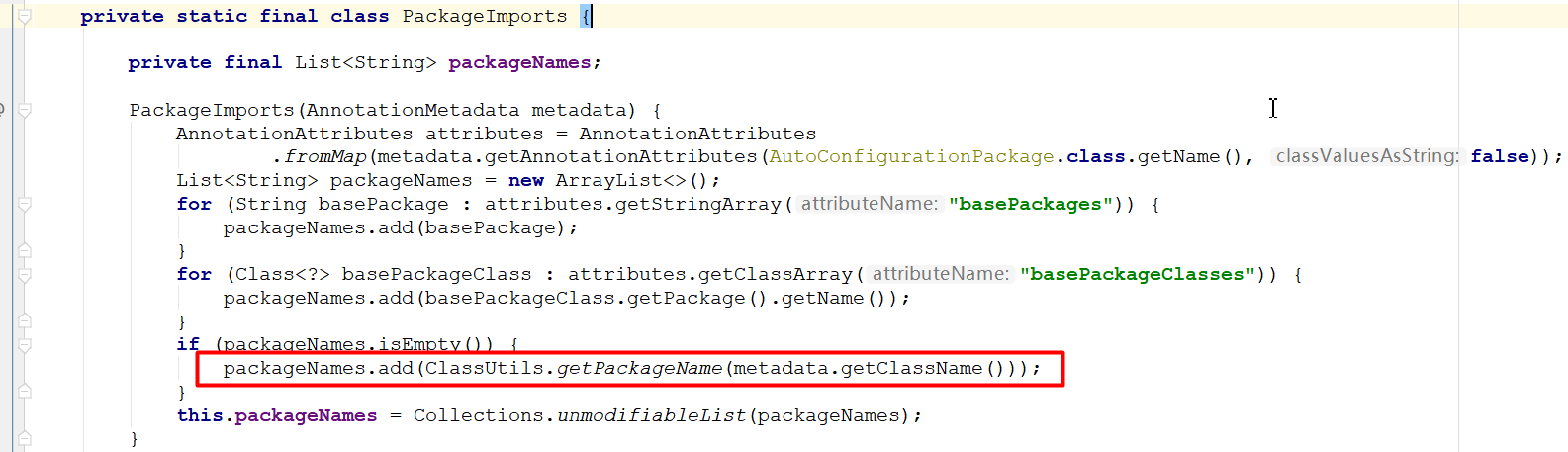
@AutoConfigurationPackage内部使用@Import来做bean的定义的注册



让我们进入AutoConfigurationPackages.Registrar，通过register的调用来注册basepackage的bean定义的

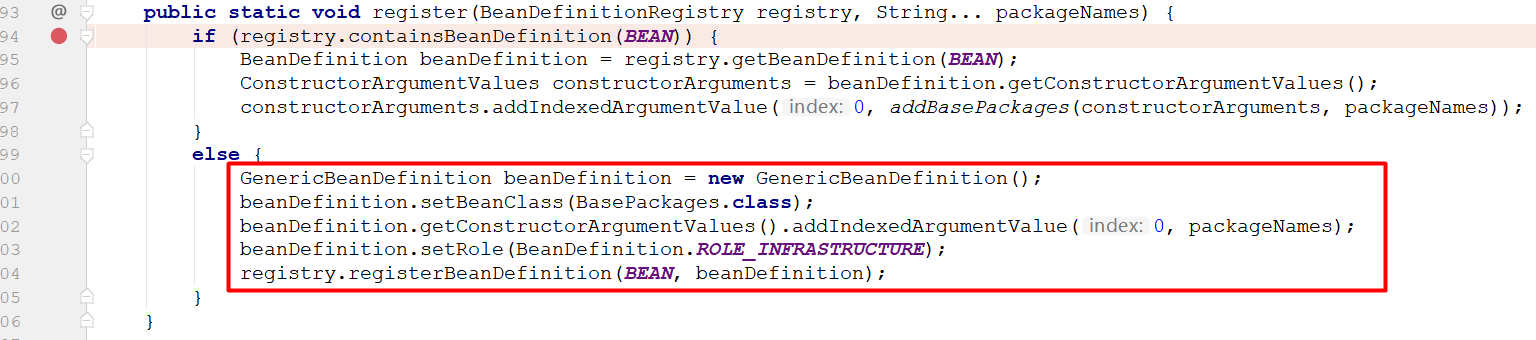


进入到PackageImports，获得basepackge设置给packageNames



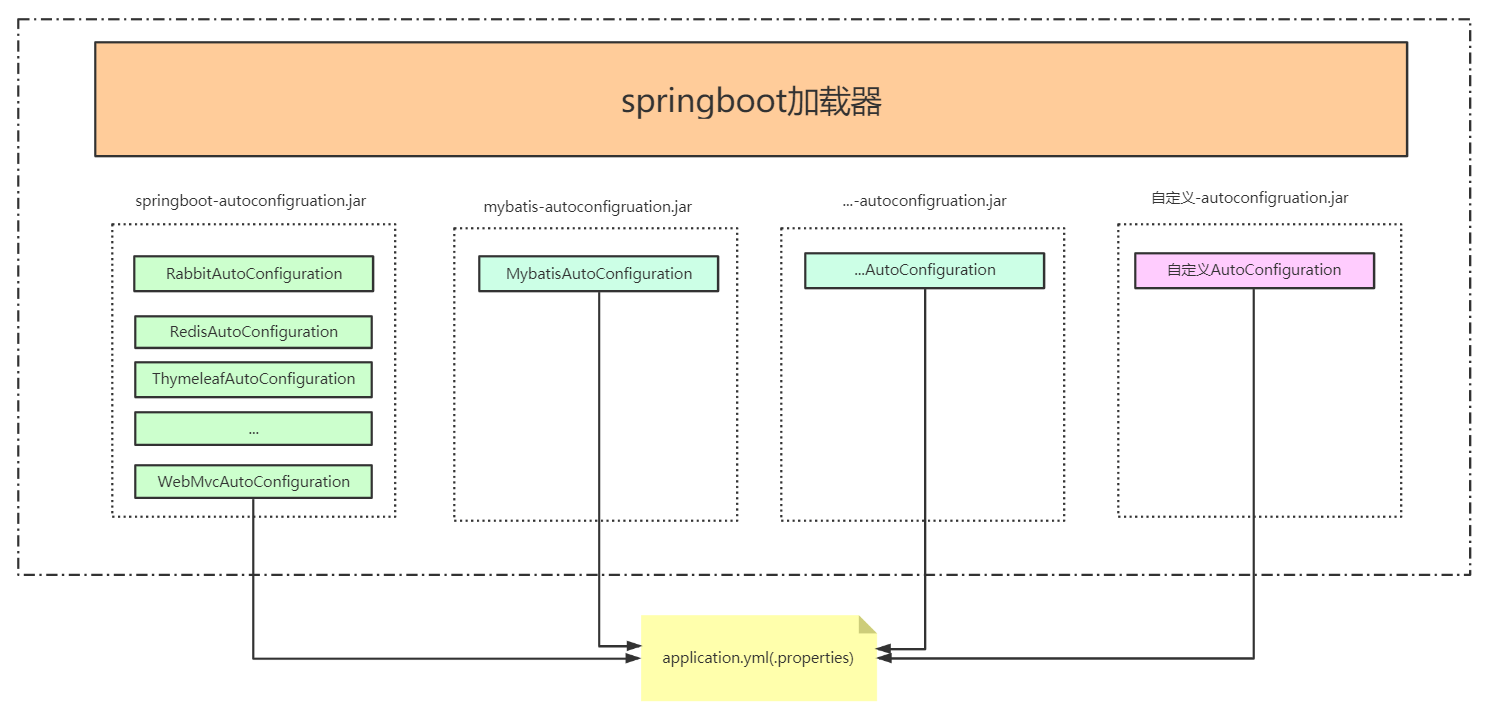
回到上一层进入到register，创建bean的定义并且把packageNames设置给bean定义

然后把bean定义的做注册。



## 自动配置浅析

springboot扫描当前classpath下所有的jar包，筛选出来EnableAutoConfiguration下的所有自动配置类注入到spring容器中，完成自动的bean的配置。



# 热部署

在实际开发过程中，每次修改代码就得将项目重启，重新部署，对于一些大型应用来说，重启时间需要花费大量的时间成本。对于一个后端开发者来说，重启过程确实很难受啊。在 Java 开发领域，热部署一直是一个难以解决的问题，目前的 Java 虚拟机只能实现方法体的修改热部署，对于整个类的结构修改，仍然需要重启虚拟机，对类重新加载才能完成更新操作。下面我们就看看对于简单的类修改的热部署怎么实现。

## 原理

深层原理是使用了两个ClassLoader，一个Classloader加载那些不会改变的类（第三方Jar包），另一个ClassLoader加载会更改的类，称为restart ClassLoader,这样在有代码更改的时候，原来的restart ClassLoader 被丢弃，重新创建一个restart ClassLoader，由于需要加载的类相比较少，所以实现了较快的重启时间。

## devtools工具包

devtools会监听classpath下的文件变动，并且会立即重启应用（发生在保存时机），

java类文件热部署（类文件修改后不会立即生效），实现对属性文件的热部署。

devtools可以实现页面热部署（页面修改后会立即生效，这个可以直接在application.properties文件中配置spring.thymeleaf.cache=false来实现，后面讲到）。

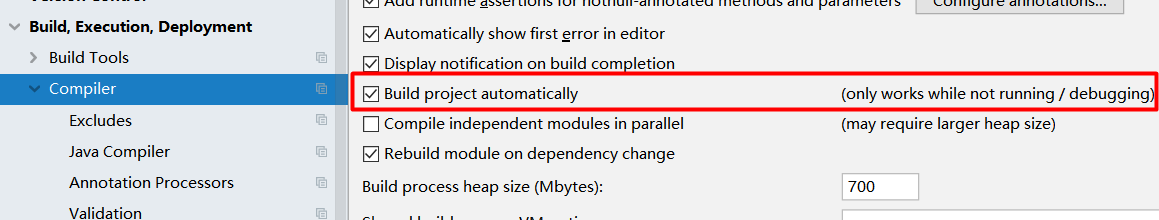
## idea的工具设置

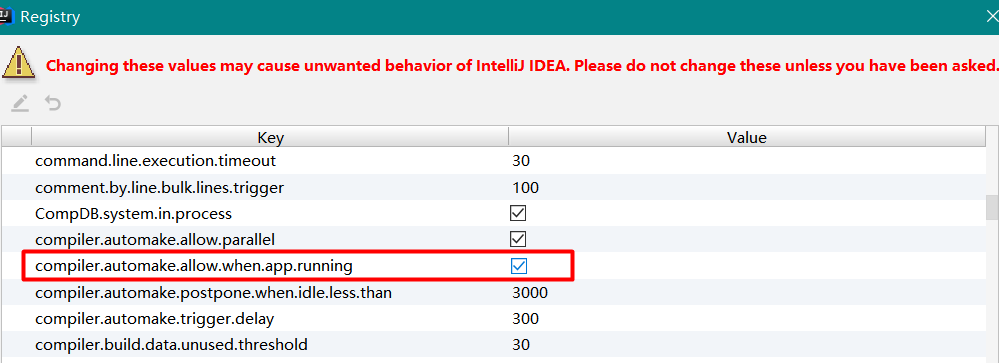
当我们修改了类文件后，idea不会自动编译，需要通过ctrl+F9来触发，

如果想要自动生效得修改idea设置，该功能按着个人的喜好来设置，修改类后，当我们窗口切换时候可以看到热部署的发生

（1）File-Settings-Compiler-Build Project automatically

（2）ctrl + shift + alt + / ,选择Registry,勾上 Compiler autoMake allow when app running





## 热部署的排除

默认情况下，/META-INF/maven，/META-INF/resources，/resources，/static，/templates，/public这些文件夹下的文件修改不会使应用重启，但是会重新加载（devtools内嵌了一个LiveReload server，当资源发生改变时，浏览器刷新）。

1. 我们在resources/static目录下创建tx.js文件每次发生修改后的并不重启，而是采用livereload的方式。
2. 同时我们可以根据自己的意愿来设置想要排除的资源

|  |
| --- |
| spring.devtools.restart.exclude=static/\*\*,public/\*\* |

# boot的属性配置文件

## 配置文件位置

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:./ 项目的跟路径，如果当前的项目有父工程，配置文件要放在父工程 的根路径

–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；

如果我们的配置文件名字不叫application.properties或者application.yml，可以通过以下参数来指定配置文件的名字，myproject是配置文件名

$ java -jar myproject.jar --spring.config.name=myproject

我们同时也可以指定其他位置的配置文件来生效

$ java -jar myproject.jar --spring.config.location=classpath:/default.properties,classpath:/override.properties

## 配置文件

### yaml

yml是YAML（YAML Ain't Markup Language）语言的文件，以数据为中心，比properties、xml等更适合做配置文件

* yml和xml相比，少了一些结构化的代码，使数据更直接，一目了然。
* 相比properties文件更简洁

### yaml语法

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

* 字面值：字符串，布尔类型，数值，日期。字符串默认不加引号，单引号会转义特殊字符。日期格式支持yyyy/MM/dd HH:mm:ss
* 对象：由键值对组成，形如 key:(空格)value 的数据组成。冒号后面的空格是必须要有的，每组键值对占用一行，且缩进的程度要一致，也可以使用行内写法：{k1: v1, ....kn: vn}
* 数组：由形如 -(空格)value 的数据组成。短横线后面的空格是必须要有的，每组数据占用一行，且缩进的程度要一致，也可以使用行内写法： [1,2,...n]
* 复合结构：上面三种数据结构任意组合

### yaml的运用

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

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

创建实体类YamlEntity.java 获取配置文件中的属性值，通过注解@ConfigurationProperties获取配置文件中的指定值并注入到实体类中。

|  |
| --- |
| @Component @ConfigurationProperties(prefix = **"yaml"**) **public class** YamlEntity {   *// 字面值，字符串，布尔，数值* **private** String **str**; *// 普通字符串* **private** String **specialStr**; *// 转义特殊字符串* **private** String **specialStr2**;*// 输出特殊字符串* **private** Boolean **flag**; *// 布尔类型* **private** Integer **num**; *// 整数* **private** Double **dNum**; *// 小数   // 数组，List和Set，两种写法： 第一种：-空格value，每个值占一行，需缩进对齐；第二种：[1,2,...n] 行内写法* **private** List<Object> **list**; *// list可重复集合* **private** Set<Object> **set**; *// set不可重复集合   // Map和实体类，两种写法：第一种：key空格value，每个值占一行，需缩进对齐；第二种：{key: value,....} 行内写法* **private** Map<String, Object> **map**; *// Map K-V* **private** List<Position> **positions**; *// 复合结构，集合对象   // 省略getter，setter，toString方法* |

### 总结

* 字符串可以不加引号，若加双引号则输出特殊字符，若不加或加单引号则转义特殊字符；
* 数组类型，短横线后面要有空格；对象类型，冒号后面要有空格；
* YAML是以空格缩进的程度来控制层级关系，但不能用tab键代替空格，大小写敏感；
* yaml的缺点是可读性比较差

## 属性绑定

前面给大家讲了yaml的语法和绑定注入给实体类，那我们平时在工作中多数是在通过实体类来写yaml的配置。属性的绑定我们必须要提供set方法

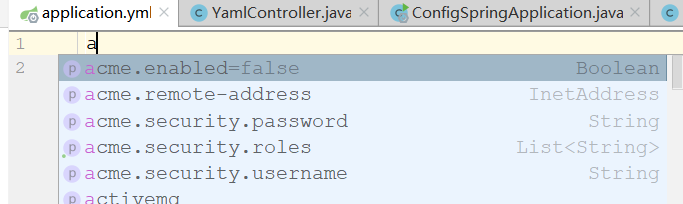
提供一段来自官网的代码：

|  |
| --- |
| @Component @ConfigurationProperties(**"acme"**) **public class** AcmeProperties {   **private boolean enabled**;   **private** InetAddress **remoteAddress**;   **private final** Security **security** = **new** Security();   **public boolean** isEnabled() {  **return enabled**;  }   **public void** setEnabled(**boolean** enabled) {  **this**.**enabled** = enabled;  }   **public** InetAddress getRemoteAddress() {  **return remoteAddress**;  }   **public void** setRemoteAddress(InetAddress remoteAddress) {  **this**.**remoteAddress** = remoteAddress;  }   **public** Security getSecurity() {  **return security**;  }   **public static class** Security {   **private** String **username**;   **private** String **password**;   **private** List<String> **roles** = **new** ArrayList<>(Collections.*singleton*(**"USER"**));   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getPassword() {  **return password**;  }   **public void** setPassword(String password) {  **this**.**password** = password;  }   **public** List<String> getRoles() {  **return roles**;  }   **public void** setRoles(List<String> roles) {  **this**.**roles** = roles;  }  } } |

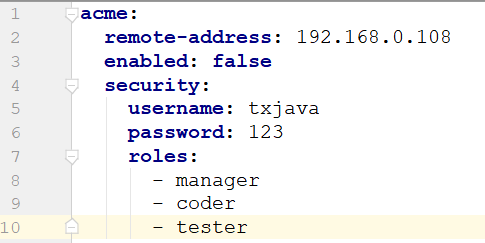
为了让当前的实体类能在配置文件中有对应的提示，我们需要引入如下的依赖，

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

加完依赖后通过Ctrl+F9来使之生效。



在配置文件中加入



然后我们测试绑定的情况。

在属性绑定的方式里，我们是通过set方法来完成的，我们可以借助Lombok来给我们带来方便。

我们在父工程中引入Lombok的依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**version**>1.16.20</**version**> </**dependency**> |

修改属性类：加上@Data注解，然后测试结果相同。

|  |
| --- |
| @Data @Component @ConfigurationProperties(**"acme"**) **public class** AcmeProperties {   **private boolean enabled**;   **private** InetAddress **remoteAddress**;   **private final** Security **security** = **new** Security();   @Data  **public static class** Security {   **private** String **username**;   **private** String **password**;   **private** List<String> **roles** = **new** ArrayList<>(Collections.*singleton*(**"USER"**));   } } |

## 构造器绑定

让我们采用构造器的方式来定义

|  |
| --- |
| @ConfigurationProperties(**"acme"**) @ConstructorBinding **public class** AcmeProperties {   **private boolean enabled**;   **private** InetAddress **remoteAddress**;   **private final** Security **security** ;   **public** AcmeProperties(**boolean** enabled, InetAddress remoteAddress, Security security) {  **this**.**enabled** = enabled;  **this**.**remoteAddress** = remoteAddress;  **this**.**security** = security;  }   **public boolean** isEnabled() {  **return enabled**;  }   **public** InetAddress getRemoteAddress() {  **return remoteAddress**;  }   **public** Security getSecurity() {  **return security**;  }   **public static class** Security {   **private** String **username**;   **private** String **password**;   **private** List<String> **roles**;   **public** Security(String username, String password,  @DefaultValue(**"USER"**) List<String> roles) {  **this**.**username** = username;  **this**.**password** = password;  **this**.**roles** = roles;  }   **public** String getUsername() {  **return username**;  }   **public** String getPassword() {  **return password**;  }   **public** List<String> getRoles() {  **return roles**;  }  } } |

要使用构造函数绑定，必须使用@EnableConfigurationProperties或配置属性扫描启用类。不能对由常规Spring机制创建的Bean使用构造函数绑定（例如@Component Bean、通过@Bean方法创建的Bean或使用@Import加载的Bean）

测试的Controller

|  |
| --- |
| @RestController @EnableConfigurationProperties(AcmeProperties.**class**) **public class** YamlController {   @Autowired  **private** AcmeProperties **acmeProperties**;   @RequestMapping(**"yaml"**)  **public** AcmeProperties yaml(){  System.***out***.println(**acmeProperties**);  **return acmeProperties**;  } } |

测试成功。在属性绑定的案例中我们同样也可以使用@EnableConfigurationProperties，此时不需要提供@Component

如果一个配置类只配置@ConfigurationProperties注解，而没有使用@Component，那么在IOC容器中是获取不到properties 配置文件转化的bean。说白了 @EnableConfigurationProperties 相当于把使用 @ConfigurationProperties 的类进行了启用注入。

在之前的版本我们都是使用@Configuration来进行作为配置类，[从SpringBoot2.2.1.RELEASE版本开始我们不再需要添加@Configuration](mailto:从SpringBoot2.2.1.RELEASE版本开始我们不再需要添加@Configuration)。

我们可以在扫描范围的bean的内部之间定义bean如：

|  |
| --- |
| @RestController **public class** YamlController {  @Bean  **public** Dep getDep(){  **return new** Dep();  }   @Autowired  **private** AcmeProperties **acmeProperties**; } |

## 第三方组件注入

除了使用@ConfigurationProperties注释类之外，还可以在public@Bean方法上使用它。如果要将属性绑定到不在您控制范围内的第三方组件

依然采用之前的案例的yaml配置

创建一个其他组件类

|  |
| --- |
| @Data **public class** AnotherComponent {   **private boolean enabled**;   **private** InetAddress **remoteAddress**; } |

创建MyService

|  |
| --- |
| @Component **public class** MyService {   @ConfigurationProperties(**"acme"**)  @Bean  **public** AnotherComponent anotherComponent(){  **return new** AnotherComponent();  } } |

我们通过测试可以获得AnotherComponent组件的实例对象。

## 松散绑定

Spring Boot使用一些宽松的规则将环境属性绑定到@ConfigurationProperties bean，因此环境属性名和bean属性名之间不需要完全匹配。

例如属性类：

|  |
| --- |
| @Data @Component @ConfigurationProperties(**"acme.my-person.person"**) **public class** OwnerProperties {   **private** String **firstName**;  } |

配置文件：

|  |
| --- |
| **acme**:  **my-person**:  **person**:  **first-name**: 泰森 |

| **属性文件中配置** | **说明** |
| --- | --- |
| acme.my-project.person.first-name | 羊肉串模式case, 推荐使用 |
| acme.myProject.person.firstName | 标准驼峰模式 |
| acme.my\_project.person.first\_name | 下划线模式 |
| ACME\_MYPROJECT\_PERSON\_FIRSTNAME | 大写下划线，如果使用系统环境时候推荐使用 |

## @ConfigurationProperties 校验

每当使用Spring的@Validated注释对@ConfigurationProperties类进行注释时，Spring Boot就会尝试验证它们。你可以用JSR-303 javax.validation直接在配置类上的约束注释。为此，请确保类路径上有一个兼容的JSR-303实现，此处我们用的是hibernate的实现，然后将约束注释添加到字段中

1. 引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.hibernate</**groupId**>  <**artifactId**>hibernate-validator</**artifactId**>  <**version**>5.2.0.Final</**version**> </**dependency**> |

1. 在属性类上加入注解

|  |
| --- |
| @Data @Component @ConfigurationProperties(**"acme.my-person.person"**) @Validated *//spring提供的注解* **public class** OwnerProperties {     @NotNull *//javax.validation.constraints提供* **private** String **firstName**;   @Max(35)  **private int age**;   @Email  **private** String **email**;  } |

1. 配置文件

|  |
| --- |
| **acme**:  **my-person**:  **person**:  **FIRST\_name**: 泰森  **age**: 34  **email**: aaa |

启动主启动类的时候，会自动发生校验。

| **Feature** | @ConfigurationProperties | @Value |
| --- | --- | --- |
| 松散绑定 | Yes | Limit |
| 元数据支持 | Yes | No |
| SpEL 表达式 | No | Yes |
| 复杂类型绑定 | Yes | No |
| 校验 | Yes | No |
| 应用场景 | Boot里面属性多个绑定 | 单个属性的绑定 |

## @ConfigurationProperties vs. @Value

1. [松散绑定在@value](mailto:1.松散绑定在@value)是被限制的

如果您确实想使用@Value，建议引用属性名（kebab case只使用小写字母，既是羊肉串模式）。这允许Spring Boot使用与放松binding@ConfigurationProperties时相同的逻辑。例如，@Value（“${demo.item-price}）将匹配demo.item-price和demo.itemPrice， 其他模式不能匹配。

1. 元数据支持

我们在@ConfigurationProperties方式可以生成元数据，目的是给我们提供提示和属性的描述。但是在@value里面是没有的。@Value适合单个的属性注入

1. spEL在@ConfigurationProperties中是不能支持的。在@Value中可以支持

如：



1. @Value复杂类型不可注入，会有启动报错。

## Profile

我们可以通过多样性的文档来解决多环境的需求。在一个yml中我们可以把文档划分成多个块

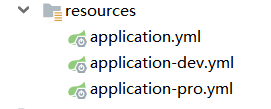
|  |
| --- |
| **acme**:  **enabled**: **true  remote-address**: 192.168.0.108  **spring**:  **profiles**: default --- **spring**:  **profiles**: development **acme**:  **enabled**: **true  remote-address**: 192.168.0.109  --- **spring**:  **profiles**: production **acme**:  **enabled**: **true  remote-address**: 192.168.0.110 |

在启动的时候我们通过**spring.profiles.active: development**来指定开启哪个profile

我们也可以采用多个文件来做。

我们创建application-dev.yml, application-pro.yml

application-{profile}.xml



属性类：

|  |
| --- |
| @Data @Component @ConfigurationProperties(**"acme"**) **public class** AcmeProperties {   **private boolean enabled**;   **private** InetAddress **remoteAddress**;   **private** String **host**;  } |

我们可以通过profile的指定来启用指定的文件。

application-dev.yml

|  |
| --- |
| **acme**:  **enabled**: **true  remote-address**: 192.168.0.109  **host**: ${**acme.remote-address**}:8080 |

application-pro.yml

|  |
| --- |
| **acme**:  **enabled**: **true  remote-address**: 192.168.0.110  **host**: ${**acme.remote-address**}:8080 |

我们依然在启动的时候我们通过**spring.profiles.active: development**来指定。

我们可以通过${}方式获取当前文档中配置和jvm的参数中的配置值。

# springboot自动配置解读

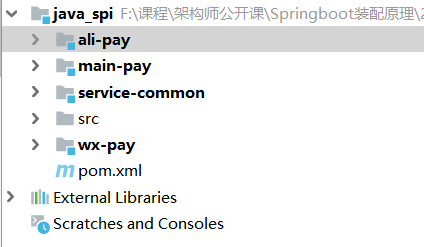
## java 中的spi

SPI的全名为Service Provider Interface.大多数开发人员可能不熟悉，因为这个是针对厂商或者插件的。在java.util.ServiceLoader的文档里有比较详细的介绍。

简单的总结下java SPI机制的思想。我们系统里抽象的各个模块，往往有很多不同的实现方案。面向的对象的设计里，我们一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。

java SPI就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。

详情参考用例代码



Java SPI 规范

要使用Java SPI，需要遵循如下约定：

1、当服务提供者提供了接口的一种具体实现后，在jar包的META-INF/services目录下创建一个以“接口全路径名”为命名的文件，内容为实现类的全限定名；

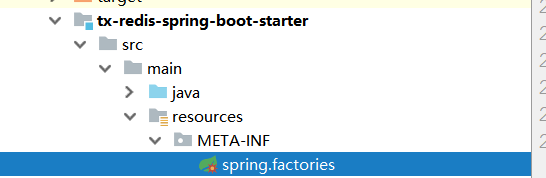
2、接口实现类所在的jar包放在主程序的classpath中；

3、主程序通过java.util.ServiceLoder动态装载实现模块，它通过扫描META-INF/services目录下的配置文件找到实现类的全限定名，把类加载到JVM；

4、SPI的实现类必须携带一个不带参数的构造方法；

## Spring Boot中的SPI机制

在Spring中也有一种类似与Java SPI的加载机制。它在META-INF/spring.factories文件中配置接口的实现类名称，然后在程序中读取这些配置文件并实例化。  
这种自定义的SPI机制是Spring Boot Starter实现的基础。



## 源码的引入

在GitHub上下载源码，解压导入到idea

clean install -DskipTests -Pfast

## Spring Factories实现原理

spring-core包里定义了SpringFactoriesLoader类，这个类实现了检索META-INF/spring.factories文件，并获取指定接口的配置的功能。在这个类中定义了两个对外的方法：

|  |
| --- |
| **public static** List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {  //获得接口名字  String factoryClassName = factoryClass.getName();  //获得所有配置类，并且根据接口名字来获得  **return** *loadSpringFactories*(classLoader).getOrDefault(factoryClassName, Collections.*emptyList*()); } |

|  |
| --- |
| **private static** Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {  //从缓存中获得spring.factories的全量信息  MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);  **if** (result != **null**) {  **return** result;  } **else** {  **try** {  //在classpath下的所有jar包中查找META-INF/spring.factories文件  Enumeration<URL> urls = classLoader != **null** ? classLoader.getResources(**"META-INF/spring.factories"**) : ClassLoader.getSystemResources(**"META-INF/spring.factories"**);  //定义存储全量工厂类的map  LinkedMultiValueMap result = **new** LinkedMultiValueMap();  //遍历urls  **while**(urls.hasMoreElements()) {  URL url = (URL)urls.nextElement();  UrlResource resource = **new** UrlResource(url);  //加载属性集和  Properties properties = PropertiesLoaderUtils.loadProperties(resource);  Iterator var6 = properties.entrySet().iterator();  //遍历属性键值对的键  **while**(var6.hasNext()) {  Entry<?, ?> entry = (Entry)var6.next();  //获得key接口  String factoryClassName = ((String)entry.getKey()).trim();  String[] var9 = StringUtils.commaDelimitedListToStringArray((String)entry.getValue());  **int** var10 = var9.length;  //切分并且遍历接口实现类，加入结果集  **for**(**int** var11 = 0; var11 < var10; ++var11) {  String factoryName = var9[var11];  result.add(factoryClassName, factoryName.trim());  }  }  }   cache.put(classLoader, result);  **return** result;  } **catch** (IOException var13) {  **throw new** IllegalArgumentException(**"Unable to load factories from location [META-INF/spring.factories]"**, var13);  }  } } |

从代码中我们可以知道，在这个方法中会遍历整个ClassLoader中所有jar包下的spring.factories文件。也就是说我们可以在自己的jar中配置spring.factories文件，不会影响到其它地方的配置，也不会被别人的配置覆盖。

spring.factories的是通过Properties解析得到的，所以我们在写文件中的内容都是安装下面这种方式配置的：

|  |
| --- |
| com.xxx.interface=com.xxx.classname |

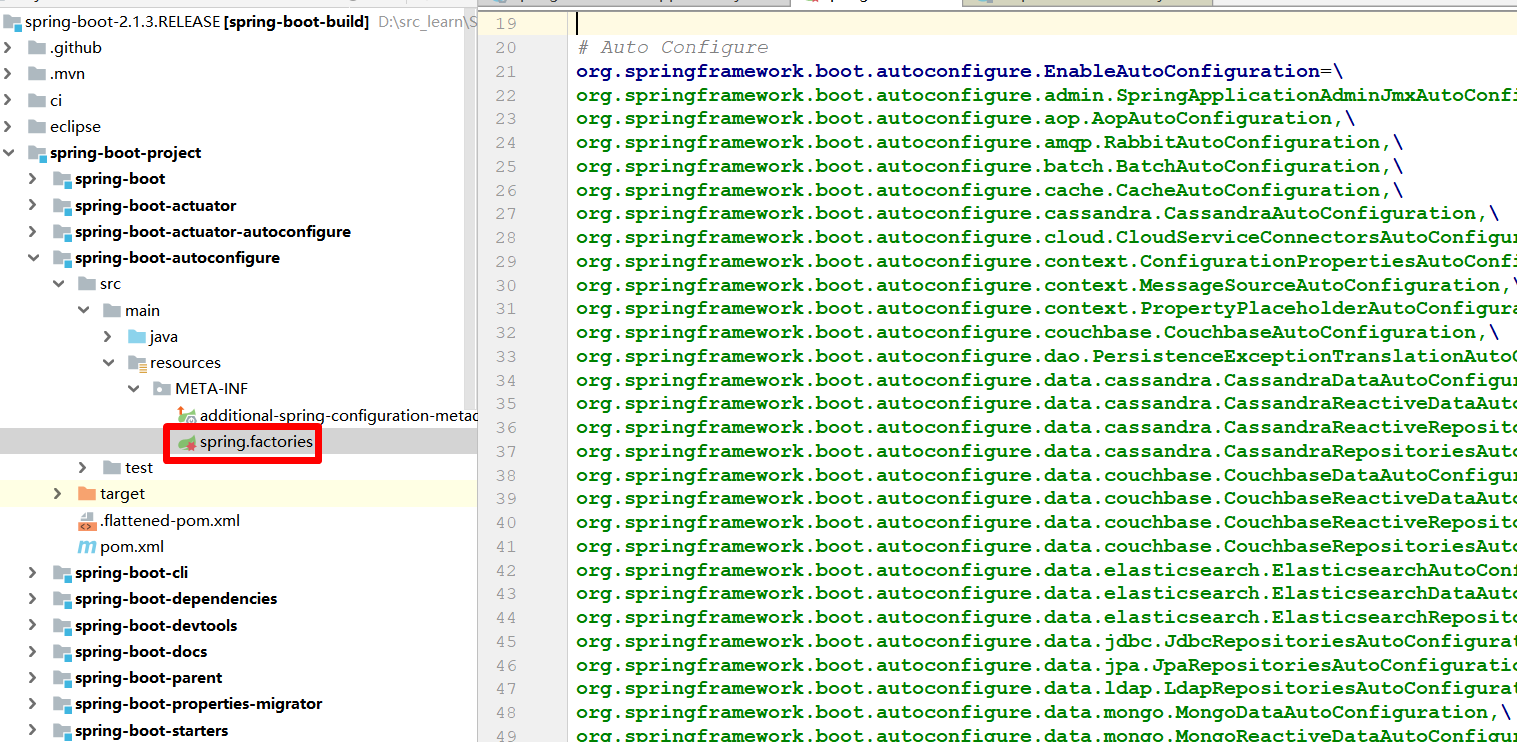
如果一个接口希望配置多个实现类，可以使用’,’进行分割。

在Spring Boot的很多包中都能够找到spring.factories文件，下面就是spring-boot包中的spring.factories文件

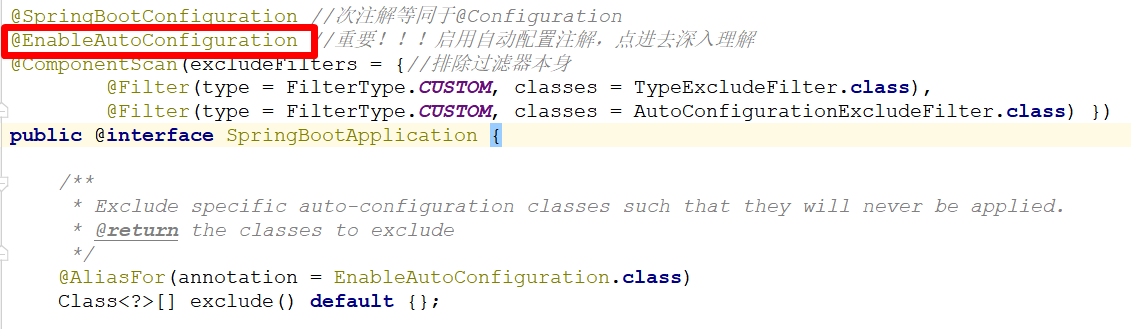
在Spring Boot中，使用的最多的就是starter。starter可以理解为一个可拔插式的插件，例如，你想使用JDBC插件，那么可以使用spring-boot-starter-jdbc；如果想使用MongoDB，可以使用spring-boot-starter-data-mongodb。  
初学的同学可能会说：如果我要使用MongoDB，我直接引入驱动jar包就行了，何必要引入starter包？starter和普通jar包的区别在于，它能够实现自动配置，和Spring Boot无缝衔接，从而节省我们大量开发时间。

## 自动配置类原理

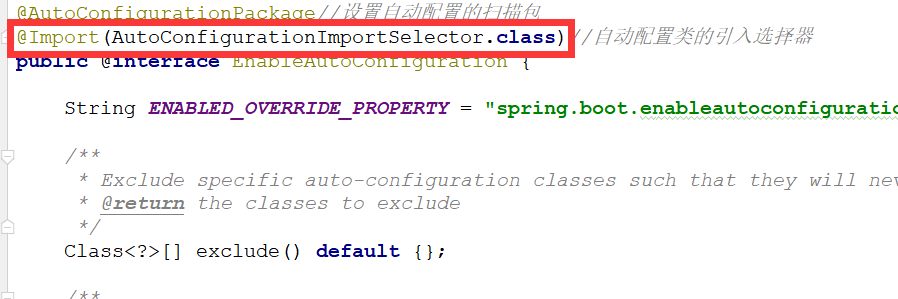
我们可以发现在spring-boot-autoconfigure中的spring.factories里面保存着springboot的默认提供的自动配置类。



让我们看在哪里去创建这些类的。我们可以关注@springbootApplication注解，在boot启动类的bean定义被加载的会执行当前的注解。



进入到@EnableAutoConfiguration注解



@AutoConfigurationImportSelector是引入自动配置类的位置。

|  |
| --- |
| **protected** AutoConfigurationEntry getAutoConfigurationEntry(  AutoConfigurationMetadata autoConfigurationMetadata,  AnnotationMetadata annotationMetadata) {  **if** (!isEnabled(annotationMetadata)) {  **return *EMPTY\_ENTRY***;  }  AnnotationAttributes attributes = getAttributes(annotationMetadata);  *//获得所有的自动配置类* List<String> configurations = getCandidateConfigurations(annotationMetadata,  attributes);  *//排除重复* configurations = removeDuplicates(configurations);  *//排除手动设置的重复* Set<String> exclusions = getExclusions(annotationMetadata, attributes);  checkExcludedClasses(configurations, exclusions);  *//移除排除的自动配置类* configurations.removeAll(exclusions);  *//过滤掉没有引入的自动配置类* configurations = filter(configurations, autoConfigurationMetadata);  fireAutoConfigurationImportEvents(configurations, exclusions);  **return new** AutoConfigurationEntry(configurations, exclusions); } |

# HttpEncodingAutoConfiguration

## HTTP编码自动配置类概览

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;  } |

## 条件判断

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；

|  |
| --- |
| @ConfigurationProperties(prefix = **"spring.http"**) **public class** HttpProperties {   */\*\*  \* Whether logging of (potentially sensitive) request details at DEBUG and TRACE level  \* is allowed.  \*/* **private boolean logRequestDetails**; |

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

# Springboot数据元自动配置

## 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**>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

在默认情况下， 数据库连接可以使用DataSource池进行自动配置

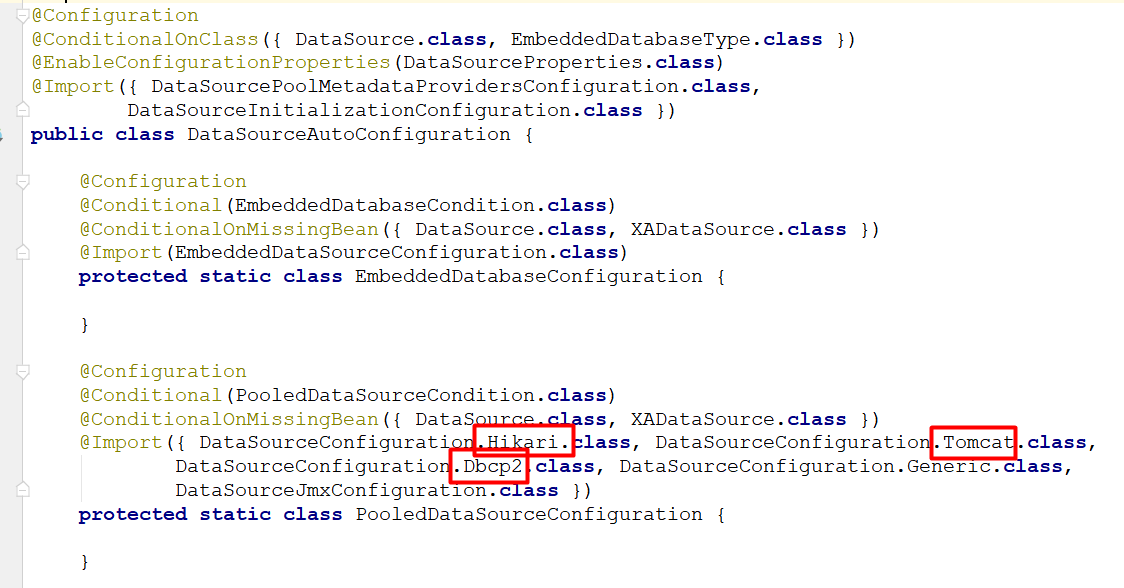
* 默认Hikari可用， Springboot将使用它。

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

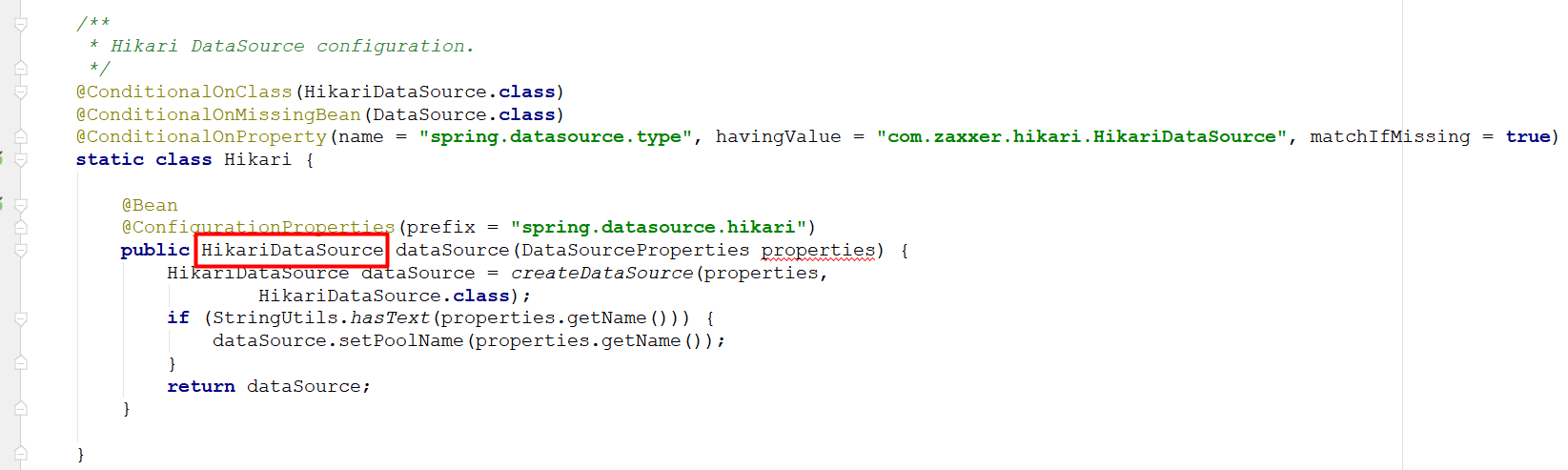
|  |
| --- |
| **spring:  datasource:  username:** root  **password:** root  **url:** jdbc:mysql://localhost:3306/boot\_demo  **driver-class-name:** com.mysql.jdbc.Driver  **type:** com.zaxxer.hikari.HikariDataSource  *# type: org.apache.commons.dbcp2.BasicDataSource* |

## 数据源自动配置原理

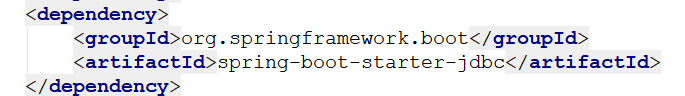
在数据源自动配置类里面我们可以看到默认支持的数据源类型

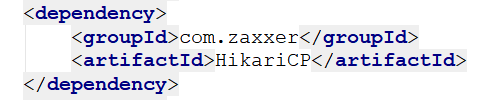


我们可以看到三种数据源的配置



点开starter-jdbc我们可以看到Hikari是默认的数据源





## 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();  } } |

# jdbcTemplate自动配置

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; |

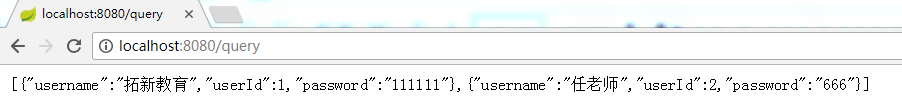


创建Controller

|  |
| --- |
| @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

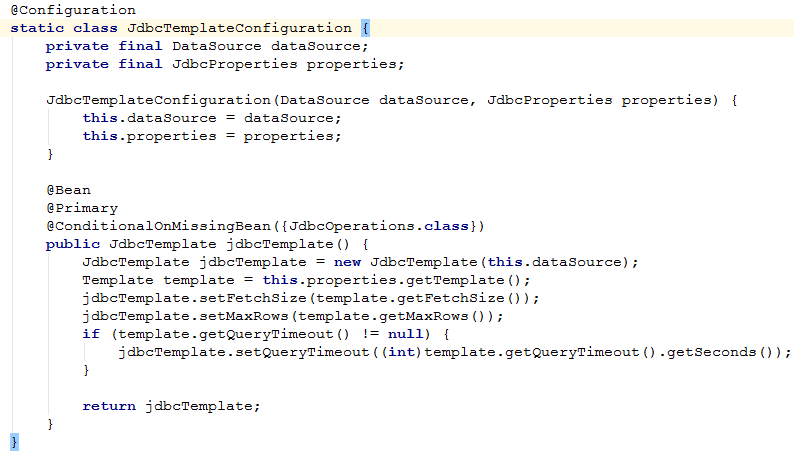
<http://localhost:8080/query>



Springboot中提供了JdbcTemplateAutoConfiguration的自动配置

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

JdbcTemplateAutoConfiguration源码：



注意：url后面加上时区useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC

# mybatis自动配置

## 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

|  |
| --- |
| DROP TABLE IF EXISTS `person`;  CREATE TABLE `person` (  `pid` int(20) NOT NULL AUTO\_INCREMENT,  `pname` varchar(50) DEFAULT NULL,  `addr` varchar(50) DEFAULT NULL,  `gender` int(2) DEFAULT NULL,  `birth` date DEFAULT NULL,  PRIMARY KEY (`pid`)  ) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8;  -- ----------------------------  -- Records of person  -- ----------------------------  INSERT INTO `person` VALUES ('1', '亮哥', '北京', '1', '2020-06-02');  INSERT INTO `person` VALUES ('2', 'zhangsan', 'shanghai', '1', '2022-12-11'); |

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

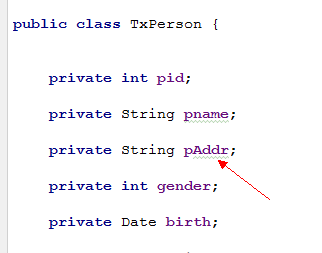
|  |
| --- |
| @Mapper **public interface** TxPersonMapper {    @Select(**"select \* from tx\_person"**)  **public** List<TxPerson> getPersons();    @Select(**"select \* from tx\_person t where t.pid = #{id}"**)  **public** TxPerson getPersonById(**int** id);   @Options(useGeneratedKeys =**true**, keyProperty = **"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);  } |

Mapper上可以不加@Mapper注解，通过@MapperScan扫描器来扫描mapper包下

|  |
| --- |
| @SpringBootApplication @MapperScan(**"cn.tx.sboot.mapper"**) **public class** FirstSpringApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(FirstSpringApplication.**class**, args);  } } |

解决驼峰模式和数据库中下划线不能映射的问题。





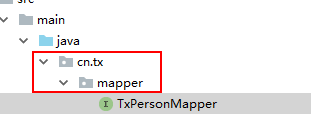
|  |
| --- |
| @Configuration **public class** MybatisConfig {   @Bean  **public** ConfigurationCustomizer getCustomizer(){  **return new** ConfigurationCustomizer() {  @Override  **public void** customize(org.apache.ibatis.session.Configuration configuration) {  configuration.setMapUnderscoreToCamelCase(**true**);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

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，也可以不创建

|  |
| --- |
| *<?***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.sboot.mapper.PersonMapper"**>  <**select id="selectById" resultType="Person"**>  select *\** from person t where t.pid = #{pid}  </**select**>  <**select id="selectAll" resultType="Person"**>  select *\** from person  </**select**>  <**insert id="insert" parameterType="person"**>  <**selectKey keyProperty="pid" resultType="int" order="BEFORE"**>  select last\_insert\_id()  </**selectKey**>  insert into person(pid, pname, addr, gender, birth)values(#{pid},#{pname},#{addr},#{gender},#{birth} )  </**insert**>  <**delete id="delete" parameterType="int"** >  delete from person where pid = #{pid}  </**delete**> </**mapper**> |

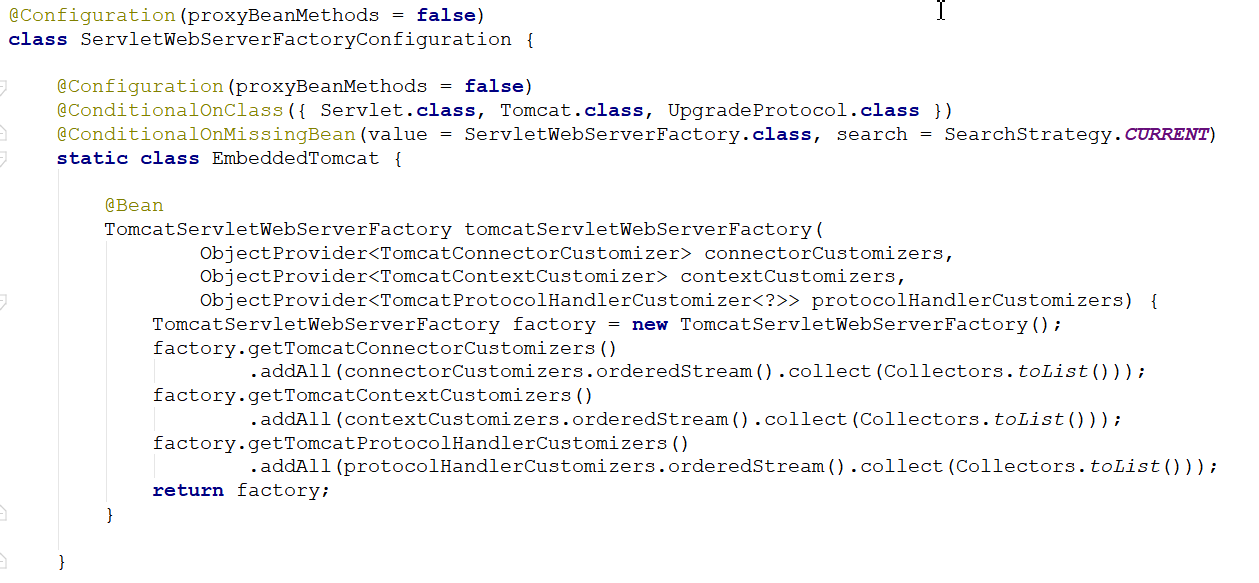
在application.yaml中配置mybatis的信息

|  |
| --- |
| **spring**:  **datasource**:  **url**: jdbc:mysql://localhost:3306/txjava?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC  **driver-class-name**: com.mysql.cj.jdbc.Driver  **username**: root  **password**: root  **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 **mybatis**:  **type-aliases-package**: cn.tx.sboot.model  **configuration**:  **map-underscore-to-camel-case**: **true  mapper-locations**: classpath:mapper/\*.xml |

# mybatis自动配置解析

# 内嵌Tomcat启动流程

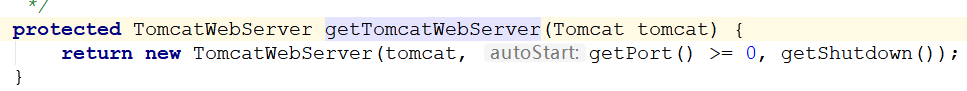
## ServletWebServerFactoryConfiguration自动配置

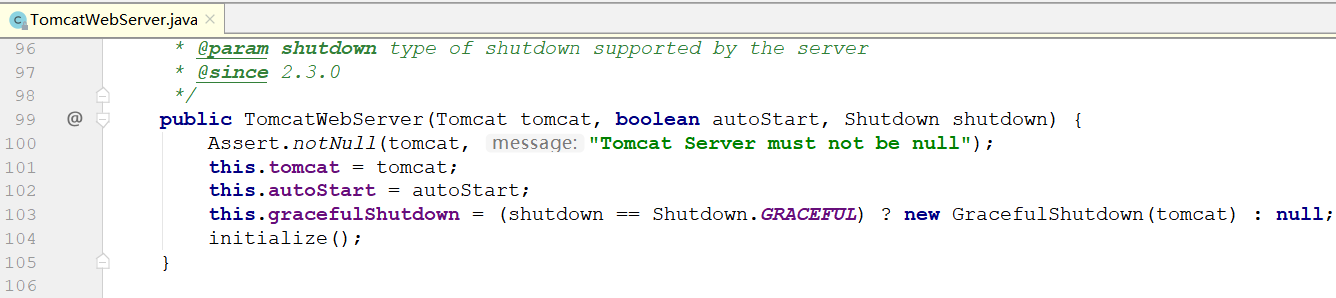


内嵌Tomcat的Bean的定义。

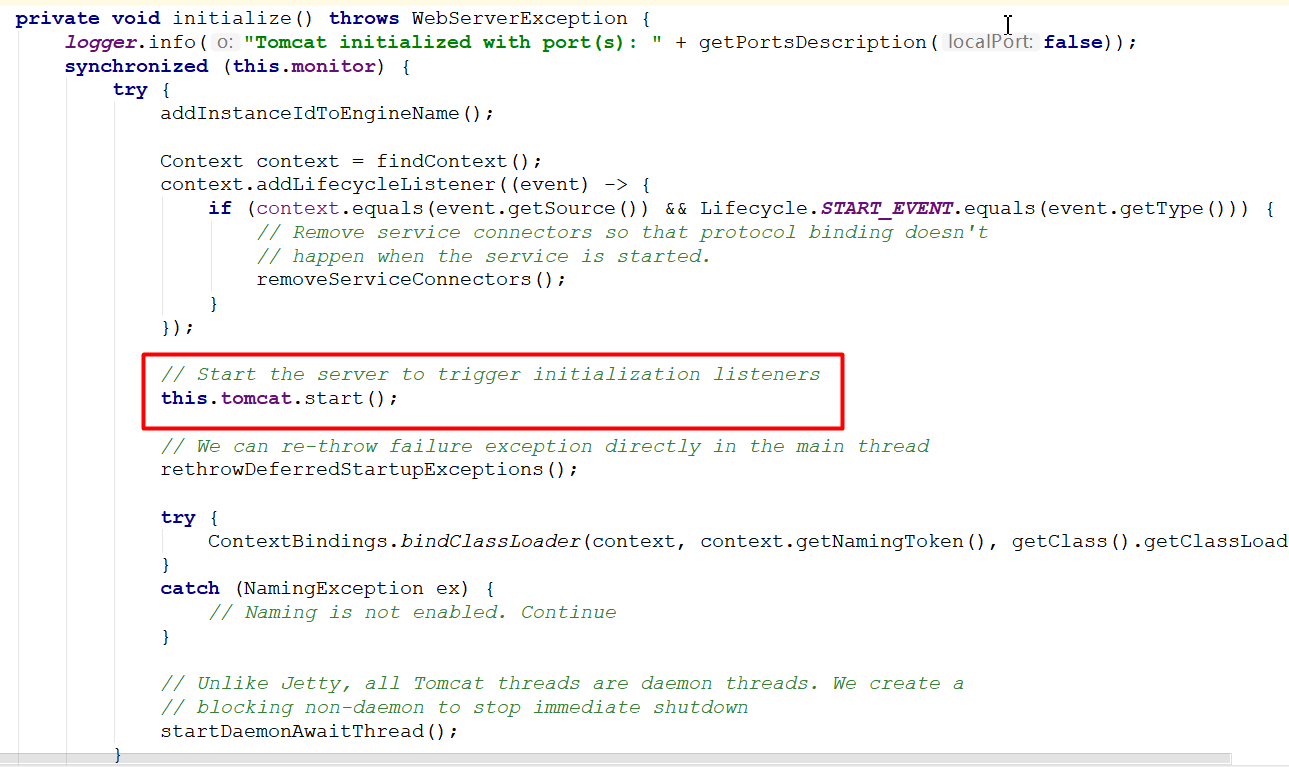


获得TomcatWebServer实例





启动Tomcat



# 11.注册web三大组件【Servlet、Filter、Listener】

## 用法

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

|  |
| --- |
| **public class** MyServlet **extends** HttpServlet {   @Override  **public void** init() **throws** ServletException {  **super**.init();  System.***out***.println(**"MyServlet被初始化"**);  }   @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  System.***out***.println(**"发生请求"**);  }   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  } } |

2. FilterRegistrationBean

|  |
| --- |
| **public class** MyFilter **implements** Filter {  @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  System.***out***.println(**"过滤器被初始化"**);  }   @Override  **public void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  System.***out***.println(**"发生过滤拦截"**);  chain.doFilter(request, response);  }   @Override  **public void** destroy() {  System.***out***.println(**"过滤器被销毁"**);  } } |

3. ServletListenerRegistrationBean

|  |
| --- |
| **public class** MyListener **implements** ServletContextListener {   @Override  **public void** contextInitialized(ServletContextEvent sce) {  System.***out***.println(**"Servlet容器被初始化"**);  }   @Override  **public void** contextDestroyed(ServletContextEvent sce) {  System.***out***.println(**"Servlet容器被销毁"**);  } } |

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

配置类中的配置：

|  |
| --- |
| @Configuration **public class** BootConfig {    @Bean  **public** ServletRegistrationBean myServlet(){  ServletRegistrationBean srb = **new** ServletRegistrationBean(**new** MyServlet(), **"/myServlet"**);  **return** srb;  }   @Bean  **public** FilterRegistrationBean myFilter(){  FilterRegistrationBean frb = **new** FilterRegistrationBean();  frb.setFilter(**new** MyFilter());  frb.setUrlPatterns(Arrays.*asList*(**"/myServlet"**, **"/myFilter"**));  **return** frb;  }   @Bean  **public** ServletListenerRegistrationBean myListener(){  ServletListenerRegistrationBean slrb = **new** ServletListenerRegistrationBean(**new** MyListener());  **return** slrb;  }  } |

## 原理

参见视频讲解。

# Spring MVC自动配置

## springmvc的MVC Config

官网地址：

<https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-config>

WebMvcConfigurer配置类其实是Spring内部的一种配置方式，采用JavaBean的形式来代替传统的xml配置文件形式进行针对框架个性化定制，可以自定义一些Handler，Interceptor，ViewResolver，MessageConverter。基于java-based方式的spring mvc配置，需要创建一个配置类并实现WebMvcConfigurer 接口；

### addInterceptors：拦截器

* addInterceptor：需要一个实现HandlerInterceptor接口的拦截器实例
* addPathPatterns：用于设置拦截器的过滤路径规则；addPathPatterns("/\*\*")对所有请求都拦截
* excludePathPatterns：用于设置不需要拦截的过滤规则
* 拦截器主要用途：进行用户登录状态的拦截，日志的拦截等。

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

### addResourceHandlers：静态资源

* 比如，我们想自定义静态资源映射目录的话，只需重写addResourceHandlers方法即可。

|  |
| --- |
| @Override **public void** addResourceHandlers(ResourceHandlerRegistry registry) {  registry.addResourceHandler(**"/res/\*\*"**).addResourceLocations(**"classpath:/tx\_static/"**); } |

### 配置视图解析器

|  |
| --- |
| @Override **public void** configureViewResolvers(ViewResolverRegistry registry) {  registry.viewResolver(**new** TxViewResolver()); }  @Bean **public** TxViewResolver txViewResolver(){  **return new** TxViewResolver(); } |

视图

|  |
| --- |
| @Component **public class** HelloView **implements** View {   @Override  **public** String getContentType() {  **return "text/html"**;  }   @Override  **public void** render(Map<String, ?> model, HttpServletRequest request,  HttpServletResponse response) **throws** Exception {  response.setHeader(**"Content-type"**, **"text/html;charset=UTF-8"**);  *//2.让servlet用UTF-8转码* response.setCharacterEncoding(**"UTF-8"**);   response.getWriter().print(**"hello view: 自定义视图解析器测试"**);  } } |

### 配置消息转换器

依赖引入

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.31</**version**> </**dependency**> |

配置fastjson消息转换器

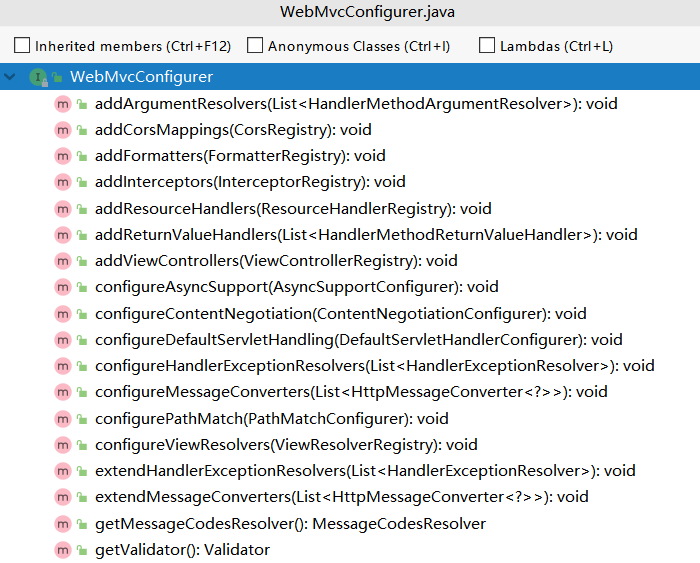
|  |
| --- |
| @Override **public void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  *//1.需要定义一个convert转换消息的对象;* FastJsonHttpMessageConverter fastJsonHttpMessageConverter = **new** FastJsonHttpMessageConverter();  *//2:添加fastJson的配置信息;* FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  *//3处理中文乱码问题* List<MediaType> fastMediaTypes = **new** ArrayList<>();  fastMediaTypes.add(MediaType.***APPLICATION\_JSON\_UTF8***);  *//4.在convert中添加配置信息.* fastJsonHttpMessageConverter.setSupportedMediaTypes(fastMediaTypes);  fastJsonHttpMessageConverter.setFastJsonConfig(fastJsonConfig);  converters.add(fastJsonHttpMessageConverter); } |

### mvc config完整代码

|  |
| --- |
| @Configuration @EnableWebMvc **public class** MVCconfig **implements** WebMvcConfigurer {   @Override  **public void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  *//1.需要定义一个convert转换消息的对象;* FastJsonHttpMessageConverter fastJsonHttpMessageConverter = **new** FastJsonHttpMessageConverter();  *//2:添加fastJson的配置信息;* FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  *//3处理中文乱码问题* List<MediaType> fastMediaTypes = **new** ArrayList<>();  fastMediaTypes.add(MediaType.***APPLICATION\_JSON\_UTF8***);  *//4.在convert中添加配置信息.* fastJsonHttpMessageConverter.setSupportedMediaTypes(fastMediaTypes);  fastJsonHttpMessageConverter.setFastJsonConfig(fastJsonConfig);  converters.add(fastJsonHttpMessageConverter);  }    @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/aaa"**).setViewName(**"index"**);  }    @Override  **public void** addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(**new** TestInterceptor())  .addPathPatterns(**"/\*\*"**)  .excludePathPatterns(**"/listUser"**);  }   @Override  **public void** addResourceHandlers(ResourceHandlerRegistry registry) {  registry.addResourceHandler(**"/res/\*\*"**).addResourceLocations(**"classpath:/tx\_static/"**);  }    @Override  **public void** configureViewResolvers(ViewResolverRegistry registry) {  registry.viewResolver(**new** TxViewResolver());  }   @Bean  **public** TxViewResolver txViewResolver(){  **return new** TxViewResolver();  } } |

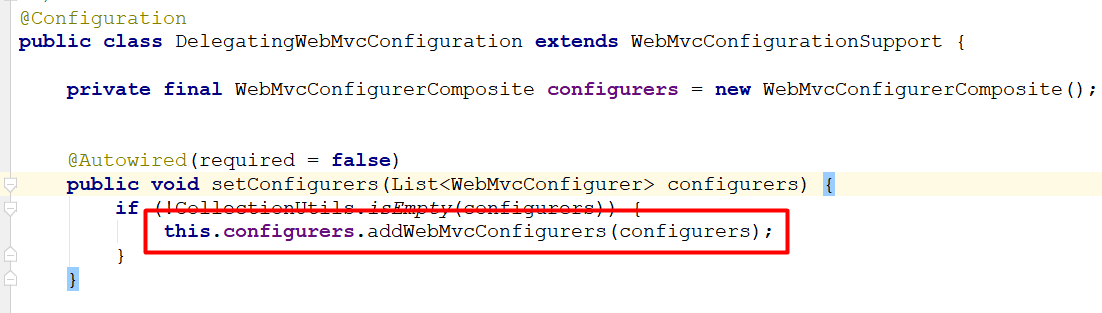
### 原理解析

我们的MVCconfig继承WebMvcConfigurer，实现里面的对应方法



@EnableWebMvc注解中导入DelegatingWebMvcConfiguration

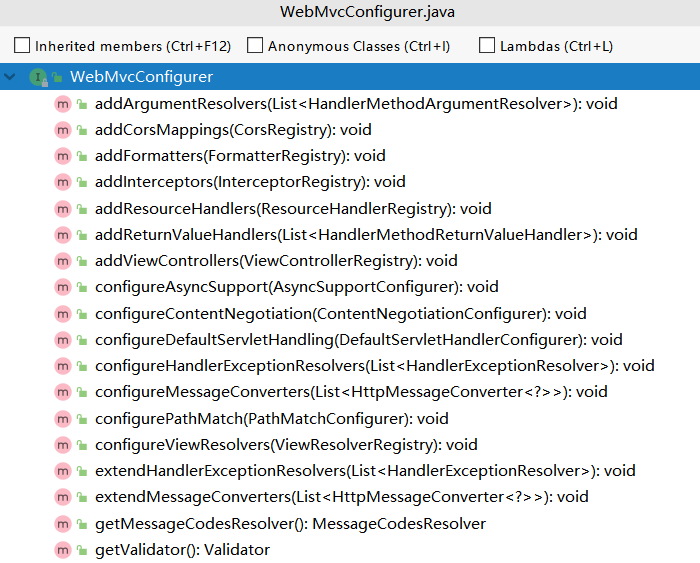
在改配置类中收集所有的WebMvcConfigurer



DelegatingWebMvcConfiguration继承WebMvcConfigurationSupport

在WebMvcConfigurationSupport中同样定义了，下面这些待实现的方法，

在DelegatingWebMvcConfiguration中得以实现





每一个实现的方法的实现，都在遍历所有的WebMvcConfigurer，从而调用么一个WebMvcConfigurer对应的方法完成每一个WebMvcConfigurer的统一注册添加和配置。

这些统一注册添加和配置是在WebMvcConfigurationSupport中以Bean的方式来配置

## springboot自动配置springmvc

让我们打开WebMvcAutoConfiguration自动配置类

|  |
| --- |
| @Configuration(proxyBeanMethods = **false**) @ConditionalOnWebApplication(type = Type.***SERVLET***) @ConditionalOnClass({ Servlet.**class**, DispatcherServlet.**class**, WebMvcConfigurer.**class** }) @ConditionalOnMissingBean(WebMvcConfigurationSupport.**class**) @AutoConfigureOrder(Ordered.***HIGHEST\_PRECEDENCE*** + 10) @AutoConfigureAfter({ DispatcherServletAutoConfiguration.**class**, TaskExecutionAutoConfiguration.**class**,  ValidationAutoConfiguration.**class** }) **public class** WebMvcAutoConfiguration { |

我们发现自动配置springmvc是不需要有WebMvcConfigurationSupport这个bean的，这就意味着我们如果自己来定义springmvc的配置类上不要有@EnableWebMvc

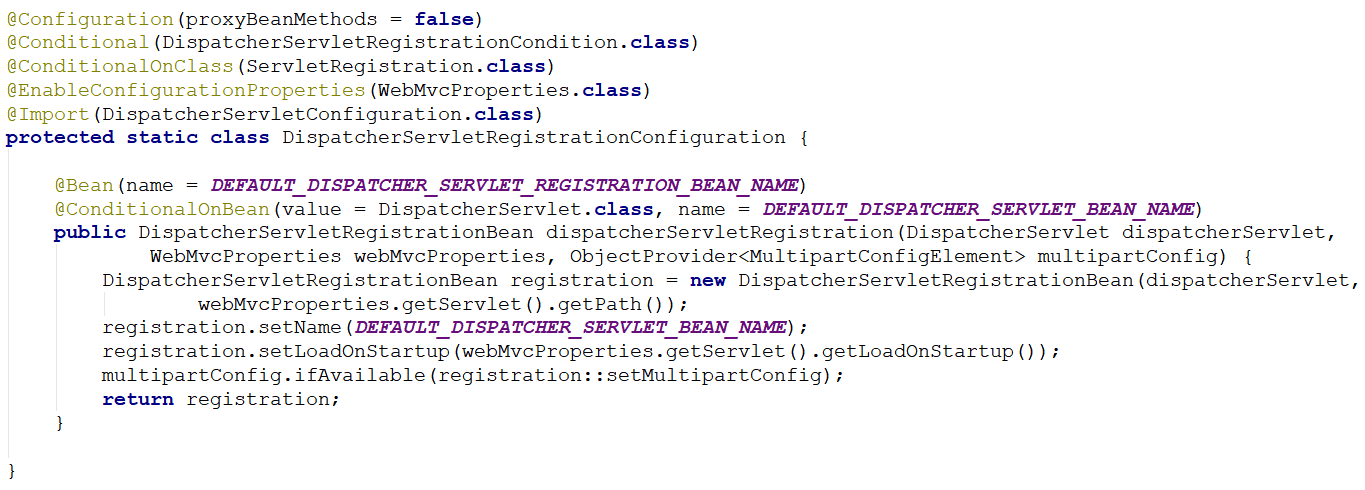
### 中央转发器DispatcherServlet

我们需要把目光投向DispatcherServletAutoConfiguration

DispatcherServlet的bean的定义



DispatcherServletRegistrationBean注册bean的定义



### configureMessageConverters消息转换器

|  |
| --- |
| @Override **public void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  *//1.需要定义一个convert转换消息的对象;* FastJsonHttpMessageConverter fastJsonHttpMessageConverter = **new** FastJsonHttpMessageConverter();  *//2:添加fastJson的配置信息;* FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  *//3处理中文乱码问题* List<MediaType> fastMediaTypes = **new** ArrayList<>();  fastMediaTypes.add(MediaType.***APPLICATION\_JSON\_UTF8***);  *//4.在convert中添加配置信息.* fastJsonHttpMessageConverter.setSupportedMediaTypes(fastMediaTypes);  fastJsonHttpMessageConverter.setFastJsonConfig(fastJsonConfig);  converters.add(fastJsonHttpMessageConverter); } |

### 消息转换器配置

方式1：在配置类中添加消息转换器

|  |
| --- |
| @Configuration **public class** DataSourceConfig {    @ConfigurationProperties(prefix = **"spring.datasource"**)  @Bean  **public** DataSource getCustomDataSource() {  **return new** DruidDataSource();  }    @Bean  **public** HttpMessageConverters fastJsonHttpMessageConverters() {  *//1.需要定义一个convert转换消息的对象;* FastJsonHttpMessageConverter fastJsonHttpMessageConverter = **new** FastJsonHttpMessageConverter();  *//2:添加fastJson的配置信息;* FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  *//3处理中文乱码问题* List<MediaType> fastMediaTypes = **new** ArrayList<>();  fastMediaTypes.add(MediaType.***APPLICATION\_JSON\_UTF8***);  *//4.在convert中添加配置信息.* fastJsonHttpMessageConverter.setSupportedMediaTypes(fastMediaTypes);  fastJsonHttpMessageConverter.setFastJsonConfig(fastJsonConfig);  **return new** HttpMessageConverters(fastJsonHttpMessageConverter);   } } |

### 视图解析器

创建视图解析器

|  |
| --- |
| **public class** TxViewResolver **implements** ViewResolver, Ordered {  @Override  **public** View resolveViewName(String viewName, Locale locale) **throws** Exception {   System.***out***.println(**"解析视图"**+ viewName);   **return new** HelloView();  }   @Override  **public int** getOrder() {  **return** Integer.***MIN\_VALUE***;  } } |

创建对应的试图

|  |
| --- |
| @Component **public class** HelloView **implements** View {   @Override  **public** String getContentType() {  **return "text/html"**;  }   @Override  **public void** render(Map<String, ?> model, HttpServletRequest request,  HttpServletResponse response) **throws** Exception {  response.setHeader(**"Content-type"**, **"text/html;charset=UTF-8"**);  *//2.让servlet用UTF-8转码* response.setCharacterEncoding(**"UTF-8"**);   response.getWriter().print(**"hello view: 自定义视图解析器测试"**);  } } |

### 静态资源加载

springboot对于静态资源的加载主要是在 启动时的 WebMvcAutoConfiguration

这个启动自动配置类中下面可以看下其中部分代码

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

Springboot中静态资源加载的拦截路径是/\*\*, 默认指向的路径是

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

"classpath:/static/"

"classpath:/public/"

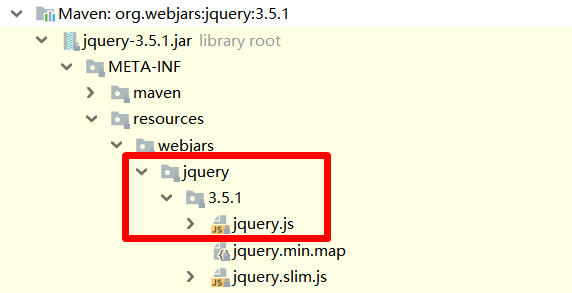
我们一般习惯使用"classpath:/static/"

Springboot也支持webjars方式加载静态资源

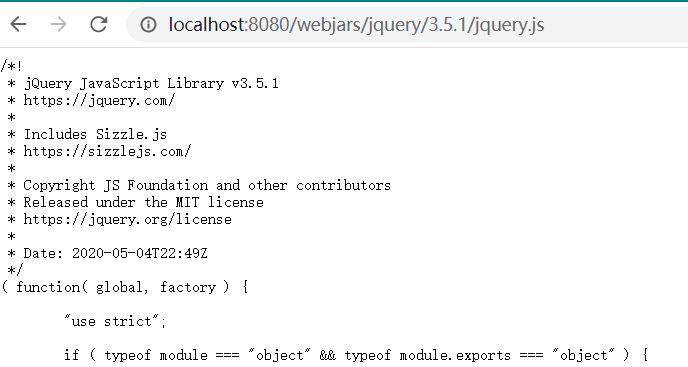
我们访问/webjars/\*\*， 被指向的路径是classpath:/META-INF/resources/webjars/

我们引入jquery的webjars的依赖

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



图中的路径就是我们要引入的路径



### 引入thymeleaf；

在pom.xml中引入

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

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

#### 基础语法

##### 文本标签 th:text/th:utext

用于文本内容的显示操作。

1. **th:text** 进行文本替换 不会解析html
2. **th:utext** 进行文本替换 会解析html

代码演示：

|  |
| --- |
| @RequestMapping("/th")  public String th(Model model){  String msg = "<h1>我是h1</h1>";  model.addAttribute("msg",msg);  return "/course/th";  } |

**th:text** 进行文本替换 不会解析html

<p th:text="text标签： + ${msg}"></p>

结果页面：

<p>text标签：<h1>我是h1</h1></p>

游览器访问的效果：  
在这里插入图片描述

**th:utext** 进行文本替换 会解析html

<p th:utext="utext标签： + ${msg}"></p>

游览器展示效果如下图：  
  
使用 + 和 | | 效果是一样的，如下代码所示：

<p th:utext="utext标签： + ${msg}"></p>

<p th:utext="|utext标签： ${msg}|"></p>

##### 字符串拼接

拼接字符串通过 + 或者 | 进行拼接

代码演示：

|  |
| --- |
| @RequestMapping("/th")  public String th(Model model){  model.addAttribute("a",1);  model.addAttribute("b",2);  return "/course/th";  } |

模版页面：

<p th:text="${a}+${b}"></p>

结果页面：

<p>3</p>

模版页面：

<p th:text="|${a} ${b}|"></p>

结果页面：

<p>1 2</p>

模版页面：

<p th:text="${a} > ${b}"></p>

结果是：

<p>false</p>

java代码：

|  |
| --- |
| @RequestMapping("/th")  public String th(Model model){  model.addAttribute("flag",true);  return "/course/th";  } |

模版页面：

<p th:text="!${flag}"></p>

结果页面：

<p>false</p>

##### \*{...}和 ${...}表达式

正常情况下 \*{...} 和 ${...}是一样的，但是 \*{...} 一般和 **th:object** 进行一起使用来完成对象属性的简写。

代码演示：

|  |
| --- |
| @RequestMapping("/th")  public String th(Model model){  User user = new User("ljk"，18);  model.addAttribute("user",user);  return "/course/th";  } |

**使用 ${...}操作**  
模版代码：

<p th:text="${user.name}"></p>

<p th:text="${user.age}"></p>

结果页面：

<p>ljk</p><p>18</p>

\*使用 {...}操作  
模版代码：

<p th:text="\*{user.name}"></p>

<p th:text="\*{user.age}"></p>

结果页面：

<p>ljk</p><p>18</p>

\*使用 {...}特有操作  
模版代码：

|  |
| --- |
| <div th:object="${user}" >  <p th:text="\*{name}"></p>  <p th:text="\*{age}"></p>  </div> |

结果页面：

<p>ljk</p><p>18</p>

##### @{...}链接网址表达式

一般和 th:href、th:src进行结合使用，用于显示Web 应用中的URL链接。通过@{...}表达式Thymeleaf 可以帮助我们拼接上web应用访问的全路径，同时我们可以通过（）进行参数的拼接

**代码演示：**

模版代码：

<img th:src="@{/images/gtvglogo.png}" />

结果页面：

<img src="/sbe/images/gtvglogo.png">

模版代码：

<a th:href="@{/product/comments(prodId=${prod.id})}" >查看</a>

结果页面：

<a href="/sbe/product/comments?prodId=2">查看</a>

模版代码：

<a th:href="@{/product/comments(prodId=${prod.id},prodId2=${prod.id})}" >查看</a>

结果页面：

<a href="/sbe/product/comments?prodId=2&amp;prodId2=2">查看</a>

##### 条件判断 th:if/th:unless

**th:if** 当条件为true则显示。  
**th:unless** 当条件为false 则显示。

**代码演示：**

java代码：

|  |
| --- |
| @RequestMapping("/thif")  public String thif(Model model){  model.addAttribute("flag",true);  return "/course/thif";  } |

模版页面：

<p th:if="${flag}">if判断</p>

结果页面：

<p>if判断</p>

模版页面：

<p th:unless="!${flag}">unless 判断</p>

结果页面：

<p>unless 判断</p>

##### switch

**th:switch** 我们可以通过switch来完成类似的条件表达式的操作。  
**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thswitch")  public String thswitch(Model model){  User user = new User("ljk",23);  model.addAttribute("user",user);  return "/course/thswitch";  } |

模版页面：

|  |
| --- |
| <div th:switch="${user.name}">  <p th:case="'ljk'">User is ljk</p>  <p th:case="ljk1">User is ljk1</p>  </div> |

结果页面：

<div><p> User is ljk</p></div>

##### for循环

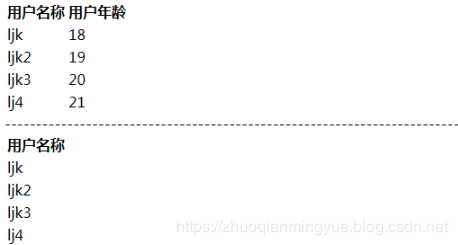
**th:each** 遍历集合

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/theach")  public String theach(Model model){  List<User> userList = new ArrayList<User>();  User user1 = new User("ljk",18);  User user2 = new User("ljk2",19);  User user3 = new User("ljk3",20);  User user4 = new User("lj4",21);  userList.add(user1);  userList.add(user2);  userList.add(user3);  userList.add(user4);  model.addAttribute("userList",userList);    List<String> strList = new ArrayList<String>();  strList.add("ljk");  strList.add("ljk2");  strList.add("ljk3");  strList.add("lj4");  model.addAttribute("strList",strList);  return "/course/theach";  } |

模版页面：

|  |
| --- |
| <table>  <thead>  <tr>  <th>用户名称</th>  <th>用户年龄</th>  </tr>  </thead>  <tbody>  <tr th:each="user : ${userList}" th:class="${userStat.odd}? 'odd'">  <td th:text="${user.name}">Onions</td>  <td th:text="${user.age}">2.41</td>  </tr>  </tbody>  </table>  ----------------------------------------------------------------------  <table>  <thead>  <tr>  <th>用户名称</th>  </tr>  </thead>  <tbody>  <tr th:each="str : ${strList}" th:class="${strStat.odd}? 'odd'">  <td th:text="${str}">Onions</td>  </tr>  </tbody>  </table> |

结果页面：  


我们可以通过便利的变量名+Stat 来获取索引 是否是第一个或最后一个等。  
便利的变量名+Stat称作状态变量，其属性有：

* index:当前迭代对象的迭代索引，从0开始，这是索引属性；
* count:当前迭代对象的迭代索引，从1开始，这个是统计属性；
* size:迭代变量元素的总量，这是被迭代对象的大小属性；
* current:当前迭代变量；
* even/odd:布尔值，当前循环是否是偶数/奇数（从0开始计算）；
* first:布尔值，当前循环是否是第一个；
* last:布尔值，当前循环是否是最后一个

#### 属性设置

##### th:href

用于声明在a 标签上的href属性的链接 该语法会和@{..} 表达式一起使用。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thhref")  public String thhref(Model model){  return "/course/thhref";  } |

模版代码：

<a href="../home.html" th:href="@{/}">返回首页</a>

结果页面：

<a href="/sbe/">返回首页</a>

##### th:class

用于声明在标签上class 属性信息。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thclass")  public String thclass(Model model){  return "/course/thclass";  } |

模版页面：

<p th:class=" 'even'? 'even' : 'odd'" th:text=" 'even'? 'even' : 'odd'"></p>

结果页面：

<p class="even">even</p>

##### th:attr

用于声明html中或自定义属性信息。

**代码演示：**

java代码：

|  |
| --- |
| @RequestMapping("/thattr")  public String thattr(Model model){  return "/course/thattr";  } |

模版页面：

<img th:attr="src=@{/images/gtvglogo.png}" />

结果页面：

<img src="/sbe/images/gtvglogo.png">

##### th:value

用于声明html中value属性信息。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thvalue")  public String thvalue(Model model){  model.addAttribute("name", "ljk");  return "/course/thvalue";  } |

模版页面：

<input type="text" th:value="${name}" />

结果页面：

<input type="text" value="ljk">

##### th:action

用于声明html from标签中action属性信息。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thaction")  public String thaction(Model model){  return "/course/thaction";  } |

模版页面：

<form action="subscribe.html" th:action="@{/subscribe}">

<input type="text" name="name" value="abc"/>

</form>

结果页面：

<form action="/sbe/subscribe">

<input type="text" name="name" value="abc">

</form>

##### th:id

用于声明htm id属性信息。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thid")  public String thid(Model model){  model.addAttribute("id", 123);  return "/course/thid";  } |

模版页面：

<p th:id="${id}"></p>

结果页面：

<p id="123"></p>

##### th:onclick

用于声明htm 中的onclick事件。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thonclick")  public String honclick(Model model){  return "/course/thonclick";  } |

模版页面：

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="UTF-8">  <title>Insert title here</title>  <script type="text/javascript">  function showUserInfo(){  alert("i am zhuoqianmingyue!")  }  </script>  </head>  <body>  <p th:onclick="'showUserInfo()'">点我</p>  </body>  </html> |

结果页面：

<p onclick="showUserInfo()">点我</p>

##### th:selected

用于声明htm 中的selected属性信息。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thselected")  public String thselected(Model model){  model.addAttribute("sex", 1);  return "/course/thselected";  } |

模版页面：

<select>

<option name="sex"></option>

<option th:selected="1 == ${sex}">男</option>

<option th:selected="0 == ${sex}">女</option>

</select>

结果页面：

<select>

<option name="sex"></option>

<option selected="selected">男</option>

<option>女</option>

</select>

##### th:src

用于声明htm 中的img中src属性信息。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thsrc")  public String thsrc(Model model){  return "/course/thsrc";  } |

模版页面：

<img title="GTVG logo" th:src="@{/images/gtvglogo.png}" />

结果页面：

<img title="GTVG logo" src="/sbe/images/gtvglogo.png">

##### th:style

用于声明htm中的标签 css的样式信息。

**代码演示：**  
java代码：

|  |
| --- |
| RequestMapping("/thstyle")  public String thstyle(Model model){  model.addAttribute("isShow", true);  return "/course/thstyle";  } |

模版页面：

<p th:style="'display:' + @{(${isShow} ? 'none' : 'block')} + ''"></p>

结果页面：

<p style="display:none"></p>

##### th:with

用于thymeleaf 模版页面中局部变量定义的使用。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/thwith")  public String thwith(Model model){  model.addAttribute("today", new Date());  return "/course/thwith";  } |

模版页面：

<p th:with="df='dd/MMM/yyyy HH:mm'">

Today is: <span th:text="${#dates.format(today,df)}">13 February 2011</span>

</p>

结果页面：

<span>02/六月/2019 06:52</span>

java代码：

|  |
| --- |
| @RequestMapping("/thwith")  public String thwith(Model model){  List<User> users = new ArrayList<User>();  users.add(new User("ljk",18));  users.add(new User("ljk2",18));  model.addAttribute("users",users);  return "/course/thwith";  } |

模版页面：

<div th:with="firstEle=${users[0]}">

<p>

第一个用户的名称是： <span th:text="${firstEle.name}"></span>.

</p>

</div>

结果页面：

<div>

<p>

第一个用户的名称是： <span>ljk</span>.

</p>

</div>

##### Elvis运算符

Elvis运算可以理解成简单的判断是否为null的三元运算的简写，如果值为nullzhe显示默认值，如果不为null 则显示原有的值。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/elvis")  public String elvis(Model model){  model.addAttribute("age", null);  return "/course/elvis";  } |

模版页面：

<p>Age: <span th:text="${age}?: '年龄为nll'"></span></p>

结果页面：

<p>Age: <span>年龄为nll</span></p>

java代码：

|  |
| --- |
| @RequestMapping("/elvis")  public String elvis(Model model){  model.addAttribute("age2", 18);  return "/course/elvis";  } |

模版页面：

<p>Age2: <span th:text="${age2}?: '年龄为nll'"></span></p>

结果页面：

<p>Age2: <span>18</span></p>

##### 三元表达式

我们可以在thymeleaf 的语法中使用三元表达式 具体使用方法是在th:x 中通过 表达式？1选项：2选项。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/threeElementOperation")  public String threeElementOperation(Model model){  return "/course/threeElementOperation";  } |

模版页面：

<p th:class=" 'even'? 'even' : 'odd'" th:text=" 'even'? 'even' : 'odd'"></p>

结果页面：

<p class="even">even</p>

java代码：

|  |
| --- |
| @RequestMapping("/threeElementOperation")  public String threeElementOperation(Model model){  model.addAttribute("name", "ljk");  return "/course/threeElementOperation";  } |

模版页面：

<p th:value="${name eq 'ljk' ? '帅哥':'丑男'}" th:text="${name eq 'ljk' ? '帅哥':'丑男'}"></p>

结果页面：

<p value="帅哥">帅哥</p>

条件表达式操作字符：  
gt：great than（大于）  
ge：great equal（大于等于）  
eq：equal（等于）  
lt：less than（小于）  
le：less equal（小于等于）  
ne：not equal（不等于）

##### No-Operation（\_）什么都不做

Elvis运算符 的一种特殊简写操作，当显示的值为null 是就什么都不做。

**代码演示：**  
java代码：

|  |
| --- |
| @RequestMapping("/noOperation")  public String noOperation(Model model){  model.addAttribute("name", null);  return "/course/noOperation";  } |

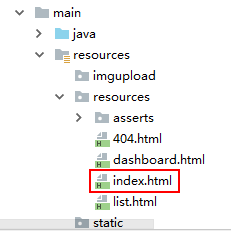
模版页面：

<span th:text="${name} ?: \_">no user authenticated</span>

结果页面：

<span>no user authenticated</span>

<https://www.cnblogs.com/jerry126/p/11531310.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"**);  } } |

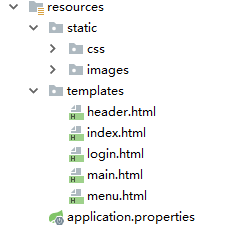
拦截器注册

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

# 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-web</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**optional**>true</**optional**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.webjars</**groupId**>  <**artifactId**>jquery</**artifactId**>  <**version**>3.3.1</**version**> </**dependency**> <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.1</**version**> </**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**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**>  <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**version**>1.16.20</**version**> </**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"**);  }    @Override  **public void** addInterceptors(InterceptorRegistry registry) {   List<String> excludePatterns = **new** ArrayList<String>();  excludePatterns.add(**"/css/\*\*"**);  excludePatterns.add(**"/images/\*\*"**);  excludePatterns.add(**"/toLogin"**);  excludePatterns.add(**"/login"**);  registry.addInterceptor(**new** LoginInterceptor())  .addPathPatterns(**"/\*\*"**)  .excludePathPatterns(excludePatterns);  } } |

## 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**> |