# Spring Boot入门

## Spring Boot简介

### 简介

spring boot 简化了spring应用开发，约定大于配置，去繁从简，just run 就能创建一个独立的，产品级别的应用

### 背景

J2EE笨重的开发、繁多的配置、底下的开发效率、复杂的部署流程、第三方继承难度太大

### 解决方案

Spring 全家桶时代

Spring Boot ：J2EE一站式解决方案

Spring Could：分布式整体解决方案

### Spring Boot 的优点

1.快速创建独立运行的Spring项目以及与主流框架集成

2.使用嵌入式Servlet容器，应用无需打成WAR包，直接可以打成jar包用java命令就可以运行，无需servlet容器

3.有很多starters负责自动依赖管理和版本控制

4.大量的自动配置，简化开发，也可以修改默认值

5.无需配置XML，无代码生成，开箱即用

6.准生产环境的运行时应用监控

7.与云计算的天然集成

### 缺点

入门容易，精通难

Spring Boot 是基于Spring 等框架的再封装，所以要深入理解了spring等框架才能更好地理解SpringBoot的原理

## 微服务简介

微服务是一种架构风格

一个应用应该由多个小型服务组成，服务之间使用http进行轻量级通信

### 单体应用

与微服务对应的，以前采用的是单体应用架构，一个应用中所有的代码都在一个项目下，统一打成war包部署到应用服务器上。

优点：

开发测试简单，不涉及到各个模块间的互联互通，所有的模块都在一个项目下

部署简单，只需要将整个应用打成war包部署就可以了，后期运维也不复杂

拓展简单，只需要将相同的应用部署到多个服务器上通过负载均衡访问即可

缺点：

1.修改一个模块就需要整个项目重新部署

2.项目维护和扩展复杂，且不利于分工合作开发

### 微服务

将每个功能元素分成一个独立的服务，后期的扩展维护只需要修改某一块

并发量高的模块可以多部署一些，可以按需分配资源

将服务细化后，每一个模块都是一个可替换的、可独立升级维护的软件单元

详细参照微服务文档

## holle world

### 创建一个maven工程

### 导入依赖spring boot 的相关依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.8.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

### 3.编写一个主程序，用于启动spring boot 应用

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

### 4.编写相关的Controller、service等等

### 5.直接运行主程序进行测试

### 6.简化部署

导入maven插件

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

可以将这个应用打成可执行的jar包，用java -jar命令执行

## HolleWorld探究

### POM文件

#### 1.父项目

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.8.RELEASE</version>  </parent>  他的父项目是：  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.5.8.RELEASE</version>  <relativePath>../../spring-boot-dependencies</relativePath>  </parent>  spring-boot-dependencies这个父项目来真正管理Spring Boot应用的所有依赖版本。在他的<properties>标签下。  也可以称之为Spring Boot的版本仲裁中心，以后导入依赖默认是不需要写版本的。  但是，没有在dependencies里面管理的依赖需要声明版本号 |

#### 2.导入的依赖

|  |
| --- |
| <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将所有功能场景都抽取出来了，做成了一个个starters（启动器）。

只需要在项目中引用这些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启动器。

### 主程序类，主入口类

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

**@SpringBootApplication**： 标注在Spring Boot 应用的某个类上，说明这个类是Spring Boot的主程序类。Spring Boot就应该运行这个类的main方法来启动Spring Boot应用。

**@SpringBootApplication注解类部分源码：**

|  |
| --- |
| @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 {  //代码  } |
|  |

**@SpringBootConfiguration：**Spring Boot的配置类，这个注解标注在某个类上，表示这是一个Spring Boot的配置类。

**@SpringBootConfiguration源码：**

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Configuration  **public** **@interface** SpringBootConfiguration {  } |

**@Configuration**：配置类上来标注这个注解

配置类 相当于 配置文件

@Configuration源码

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Component  **public** **@interface** Configuration {  } |

配置类也是容器中的一个组件，用@Component注解标注

**@EnableAutoConfiguration**：开启自动配置功能。

以前需要我们自己配置的东西，Spring Boot帮我们自动配置。@EnableAutoConfiguration告诉Spring Boot开启自动配置功能，这样自动配置才能生效。

@EnableAutoConfiguration注解类：

|  |
| --- |
| @SuppressWarnings("deprecation")  @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @AutoConfigurationPackage  @Import(~~EnableAutoConfigurationImportSelector~~.**class**)  **public** **@interface** EnableAutoConfiguration {  } |

@AutoConfigurationPackage：自动配置包

@AutoConfigurationPackage：

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @Import(**AutoConfigurationPackages**.Registrar.**class**)  **public** **@interface** AutoConfigurationPackage {  } |

@Import(AutoConfigurationPackages.Registrar.class)：

Spring的底层注解@Import，给容器中导入一个组件。

将主程序类（**@SpringBootApplication标注的类**）所在的包及下面所有的子包里面所有的组件扫描到Spring容器。

@Import(EnableAutoConfigurationImportSelector.class)

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（xxxxAutoConfiguration），就是给容器中导入这个场景的所有组件，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；

## 快速创建Spring Boot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。

# 配置文件

# 配置文件

Spring Boot默认支持两种全局配置文件(配置文件名是固定的)：

application.properties

application.yml

配置文件的作用：SpringBoot在底层都已经自动配置好了，配置文件就是用来修改SpringBoot的自动配置的默认值。

yml文件用的是YAML语言（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：

以前我们的配置文件，大多都是XML文件

YAML:以数据为中心，比json、xml等更适合做配置文件

配置例子：

YAML:

|  |
| --- |
| Server:  Port: 8081 |

XML

|  |
| --- |
| <SERVER>  <port>8081</port>  </SERVER> |

# YAML语法

### 基本语法

K：（空格）V 表示一个键值对（value前面必须有一个空格）

以空格缩进来控制层级关系。只要是左对齐的一列数据，都是同一层级的。

属性和值都是大小写敏感的

### 值的写法

#### 字面量：普通的值（数字、字符串、布尔变量等）

字面量直接写： K: V

字符串默认不用加上单引号和双引号。

双引号：不会转义字符串里面的特殊字符，特殊字符作为本身想表示的意思

单引号：会转义特殊字符，特殊字符最终只作为普通的字符输出。

例如：

“zhangsan /n lisi” 输出：

|  |
| --- |
| zhangsan  lisi |

‘zhangsan /n lisi’ 输出：

|  |
| --- |
| zhangsan /n lisi |

#### 对象、Map（键值对）

还是以k: v表示，在下一行来写对象的属性和值的关系，注意缩进

如：friend对象

Friend:

lastName: zhangsan

age: 20

行内写法：

Friend: {lastName: zhangsan,age: 18}

#### 数组或集合（List、Set）

用- 值表示数组的一个元素

|  |
| --- |
| Pets:  - cat  - dog  - pig |

行内写法

|  |
| --- |
| Pets: [cat,dog,pig] |

# 配置文件值的注入

### 配置文件

配置文件：

|  |
| --- |
| **Person:  lastName:** zhangsan  **age:** 18  **boss: false  birth:** 2017/12/13  **maps:** {**k1:** v1,**k2:** v2}  **list:** - lisi  - wangwu  **dog:  name:** 小狗  **age:** 2 |

### 实体类

JavaBean：

|  |
| --- |
| */\*\*  \* 将配置文件中配置的每一个属性值，映射到这个组件中  \** ***@ConfigurationProperties:*** *告诉springboot将本类中所有的属性和配置文件中相关的配置进行绑定  \* perfix = "person" ：与配置文件中哪个下面的所有属性一一对应  \*  \* 只有这个组件是容器中的组件时，才能使用容器提供的功能，所以要加上@Component注解  \*/* @Component @ConfigurationProperties(prefix = **"person"**) **public class** Person {   **private** String **lastName**;  **private** Integer **age**;  **private** Boolean **boss**;  **private** Date **birth**;   **private** Map<String,Object> **maps**;  **private** List<Object> **lists**;  **private** Dog **dog**;   @Override  **public** String toString() {  **return "Person{"** +  **"lastName='"** + **lastName** + **'\''** +  **", age="** + **age** +  **", boss="** + **boss** +  **", birth="** + **birth** +  **", maps="** + **maps** +  **", lists="** + **lists** +  **", dog="** + **dog** +  **'}'**;  }   **public** String getLastName() {  **return lastName**;  }   **public void** setLastName(String lastName) {  **this**.**lastName** = lastName;  }   **public** Integer getAge() {  **return age**;  }   **public void** setAge(Integer age) {  **this**.**age** = age;  }   **public** Boolean getBoss() {  **return boss**;  }   **public void** setBoss(Boolean boss) {  **this**.**boss** = boss;  }   **public** Date getBirth() {  **return birth**;  }   **public void** setBirth(Date birth) {  **this**.**birth** = birth;  }   **public** Map<String, Object> getMaps() {  **return maps**;  }   **public void** setMaps(Map<String, Object> maps) {  **this**.**maps** = maps;  }   **public** List<Object> getLists() {  **return lists**;  }   **public void** setLists(List<Object> lists) {  **this**.**lists** = lists;  }   **public** Dog getDog() {  **return dog**;  }   **public void** setDog(Dog dog) {  **this**.**dog** = dog;  } } |

### 配置文件处理器

可以导入配置文件处理器，以后编写配置文件就有javabean对应的提示了

在pom.xml文件中添加依赖：

|  |
| --- |
| *<!--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**> |

### 单元测试

使用springboot的test 进行单元测试

|  |
| --- |
| */\*\*  \* SpringBoot单元测试  \** ***@SpringBootTest:*** *标注这是一个springBoot测试类  \*  \** ***@RunWith(SpringRunner.class)：*** *标注这个单元测试用SpringRunner这个驱动器来跑，而不是原来的junit  \*  \* 可以再测试期间很方便的类似编码一样进行自动注入到容器的功能  \*  \*/* @RunWith(SpringRunner.**class**) @SpringBootTest **public class** SpringBoot001ApplicationTests {   @Autowired  **private** Person **person**;   @Test  **public void** contextLoads() {  System.***out***.print(**person**);  } |

### @Value注解读取配置文件

|  |
| --- |
| */\*\*  \** ***@Value*** *注解：Spring的注解，对应xml配置文件的<bean>标签  \* <bean class="Person">  \* <property name="lastName" value="?"></property>  \* <bean/>  \*  \* value的值可以是：  \* 1.字面量  \* 2.${key} 从环境变量、配置文件中获取值  \* 3.#{SpEL} SpEL表达式  \*  \*/  //从配置文件取值* @Value(**"${person.last-name}"**) **private** String **lastName**;  *//SpEL表达式* @Value(**"#{11\*2}"**) **private** Integer **age**;  *//字面量* @Value(**"true"**) **private** Boolean **boss**; |

### @Value和@ConfigurationProperties的区别

|  |  |  |
| --- | --- | --- |
|  | @ConfigurationProperties | @Value |
| 功能 | 批量注入配置文件中的属性 | 需要一个个指定 |
| 松散绑定（松散语法） | 如果实体类的字段为lastName。  配置文件中可以写成：  person.lastName  Person.last-name  Person.last\_name  PERSON\_LAST\_NAME(系统属性推荐使用，不区分大小写) | 必须一一对应，且大小写敏感 |
| SpEL | 不支持 | 支持 |
| JSR303数据校验 | 支持 | 不支持 |
| 复杂类型封装（map、对象等） | 支持 | 不支持 |

配置文件yml还是properties文件他们都能取到值

如果我们只是在某个业务逻辑中获取配置文件中某一项的值，使用@value

如果我们专门编写一个javaBean映射配置文件，我们就使用@ConfigurationProperties

### JSR303数据校验

|  |
| --- |
| @Component @ConfigurationProperties(prefix = **"person"**) @Validated **public class** Person {  *// lastName必须是邮箱格式*  @Email  **private** String **lastName**;  } |

### @PropertySource与@ImportResource

#### @PropertuSource

@PropertuSource：加载指定的配置文件

|  |
| --- |
| */\*\*  \** ***@ConfigurationProperties：*** *默认读取全局配置文件application.properties下的配置  \* 要读取其他配置文件中的配置，需要用到@PropertySource注解指定配置文件  \*/* @PropertySource(value = {**"classpath:person.properties"**}) @Component @ConfigurationProperties(prefix = **"person"**) *//@Validated* **public class** Person {**private** String **lastName**; **private** Integer **age**; **private** Boolean **boss**;  **private** Date **birth**;   **private** Map<String,Object> **maps**;  **private** List<Object> **lists**;  **private** Dog **dog**;  } |

#### @ImportResource

@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，我们自己创建的配置也不能自动识别。

想让Spring的配置文件生效，加载进来。把@ImportResource注解标注到一个配置类上

HolleService

|  |
| --- |
| **public class** HolleService { } |

配置文件：beans.xml 配置holleService组件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>   <**bean id="holleService" class="com.example.springboot001.service.HolleService"**></**bean**> </**beans**> |

主类：用@ImportResource将配置文件导入

|  |
| --- |
| @ImportResource(locations = {**"classpath:beans.xml"**}) @SpringBootApplication **public class** SpringBoot001Application {   **public static void** main(String[] args) {  SpringApplication.*run*(SpringBoot001Application.**class**, args);  } } |

测试类：

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** SpringBoot001ApplicationTests {  @Autowired  **private** ApplicationContext **ioc**;   @Test  **public void** testHolleService(){  **boolean** b = **ioc**.containsBean(**"holleService"**);  System.***out***.print(b);  }  } |

#### SpringBoot推荐给容器添加组件的方式

SpringBoot不推荐使用配置文件，而是使用全注解的方式

推荐使用配置类，相当于Spring的配置类，使用@Configuration标注配置类

使用@Bean给容器添加组件

|  |
| --- |
| */\*\*  \** ***@Configuration:*** *指明当前类是一个配置类，用于替代之前的Spring配置文件  \* 配置文件中使用<bean>标签来添加组件的  \*  \*/* @Configuration **public class** MyAppConfig {   */\*\*  \** ***@Bean:*** *对应配置文件中的*<*bean*>*标签  \* 将方法的返回值添加到容器中，容器中这个组件的默认id就是方法名  \*  \** ***@return*** *\*/* @Bean  **public** HolleService holleService(){  System.***out***.print(**"配置类给容器添加组件：holleService"**);  **return new** HolleService();  } } |

### 配置文件占位符

#### 随机数

RandomValuePropertySource:

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

#### 占位符

通过占位符获取之前配置的值，如果没有值可以使用：指定默认值

|  |
| --- |
| app.name=MyApp  app.description=${app.name:MyApp} is a Spring Boot application |

# Profile

Profile是Spring提供多环境支持的功能，对不同环境不同配置功能的支持，可以通过激活、指定参数等方式快速切换环境

### 多profile文件

多profile文件方式实现

我们在主配置文件编写的时候，文件名可以是application-{profile}.properties/yml

如：application-dev.properties、application-prod.properties分别为开发环境和生产环境的配置文件。

默认使用application.properties的配置文件

### 激活指定的profile

1. 在application.properties文件中指定：

spring.profiles.active=dev 指定激活开发环境的profile

1. 命令行方式：

--spring.profiles.active=dev

如java -jar springboot.jar --spring.profiles.active=dev

指定使用项目的dev环境

1. 虚拟机参数

-Dspring.profiles.active=dev

### yml支持多文档块方式

|  |
| --- |
| **spring:  profile:  active:** dev  **server:  port:** 8080  --- **server:  port:** 8090  **spring:  profiles:** dev --- **server:  port:** 8082  **spring:  profiles:** prod |

yml可以用--- 划分文档块，每个文档块用spring.profiles指定profile

然后在第一个文档块用spring.profile.active指定要激活的profile

# 配置文件加载位置

SpringBoot启动时会扫描以下位置的application.properties或application.yml配置文件作为主配置文件：

file:./config/ 项目根目录的config文件夹下

file:./ 项目的根目录下

Classpath:/config/ 类路径下的config文件夹下

Classpath:/ 类路径的根目录下

优先级由高到低，所有位置上的配置文件都会被加载，如果出现重复的内容，高优先级的配置文件会覆盖低优先级的配置文件

项目打包完成后，可以通过命令行参数的形式，使用spring.config.location指定新的配置文件的位置，这个新的配置文件会和项目下的配置文件形成互补配置

# SpringBoot外部配置加载顺序

Spring Boot除了从配置文件加载配置，还可以从以下位置加载配置。高优先级配置会覆盖低优先级配置。

1. [Devtools](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "using-boot-devtools-globalsettings" \o "20.4 Global Settings) 主目录上的[全局设置属性](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "using-boot-devtools-globalsettings" \o "20.4 Global Settings)（~/.spring-boot-devtools.properties当devtools处于活动状态时）。
2. [@TestPropertySource](https://docs.spring.io/spring/docs/5.0.7.RELEASE/javadoc-api/org/springframework/test/context/TestPropertySource.html" \t "https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/_top) 测试上的注释。
3. [@SpringBootTest#properties](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/api/org/springframework/boot/test/context/SpringBootTest.html" \t "https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/_top) 测试中的注释属性。
4. 命令行参数。
5. 来自SPRING\_APPLICATION\_JSON（嵌入在环境变量或系统属性中的内联JSON）的属性。
6. ServletConfig init参数。
7. ServletContext init参数。
8. JNDI属性来自java:comp/env。
9. Java系统属性（System.getProperties()）。
10. OS环境变量。
11. 一RandomValuePropertySource，只有在拥有性能random.\*。
12. [特定于配置文件的应用程序属性](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "boot-features-external-config-profile-specific-properties" \o "24.4 Profile-specific Properties)在打包的jar（application-{profile}.properties和YAML变体）之外。
13. 打包在jar中[的特定于配置文件的应用程序属性](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "boot-features-external-config-profile-specific-properties" \o "24.4 Profile-specific Properties)（application-{profile}.properties 以及YAML变体）。
14. 应用程序属性在打包的jar之外（application.properties和YAML变体）。
15. 打包在jar中的应用程序属性（application.properties和YAML变体）。
16. [@PropertySource](https://docs.spring.io/spring/docs/5.0.7.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html" \t "https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/_top) 你的@Configuration课上的注释。
17. 默认属性（由设置指定SpringApplication.setDefaultProperties）。

# SpringBoot自动配置原理

### 自动配置原理

可配置的内容参考官方文档附录中的Common application properties

自动配置原理：

1. SrpingBoot启动的时候加载了主配置类，开启了自动配置功能@EnableAutoConfiguration
2. @EnableAutoConfiguration作用：
   1. 利用AutoConfigurationImportSelector给容器导入组件。
   2. 可以查看selectImports()方法的相关内容。
   3. List<String> configurations = **this**.getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置
      1. SpringFactoriesLoader.loadFactoryNames()
      2. 扫描所有jar包类路径下 META/spring.factories
      3. 把扫描到的这些文件的内容包装成properties对象
      4. 从properties中获取到的EnableAutoConfiguration.class对应的值，把他们添加到容器中

一句话来说，就是@EnableAutoConfiguration就是把所有jar包类路径下 META/spring.fa ctories里面配置的所有EnableAutoConfiguration的值加入到容器中。

如：

|  |
| --- |
| **org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\ org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\ org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\**  .......等等，参考AutoConfiguration jar包下的 META/spring.factories |

每一个xxxxAutoConfiguration这样的类都是容器的一个组件，加入到容器中。容器用他们来做自动配置。

1. 每一个自动配置类分别进行自动配置功能
2. 以HttpEncodingAutoConfiguration为例 解释自动配置原理

|  |
| --- |
| @Configuration  //表示这是一个配置类，相当于以前的配置文件，给容器添加组件 @EnableConfigurationProperties({HttpEncodingProperties.**class**})  //启动指定类的ConfigurationProperties功能，将配置文件的内容与HttpEncodingProperties绑定起来，并把HttpEncodingProperties添加到ioc容器中 @ConditionalOnWebApplication(  type = Type.SERVLET )  //spring底层的@Conditional注解，根据不条件判断配置类是否生效  //判断当前项目是不是web应用，是配置类才生效 @ConditionalOnClass({CharacterEncodingFilter.**class**})  //判断有没有CharacterEncodingFilter这个类，CharacterEncodingFilter是springmvc中解决乱码问题的过滤器 @ConditionalOnProperty(  prefix = **"spring.http.encoding"**,  value = {**"enabled"**},  matchIfMissing = **true** )  //判断配置文件中是否存在spring.http.encoding.enabled=true这个配置  //matchIfMissing ：如果不存在，默认为true **public class** HttpEncodingAutoConfiguration {  //已经通过@ConfigurationProperties注解和SpringBoot配置文件映射了  **private final** HttpEncodingProperties properties;  //只有一个有参构造方法，参数从会从容器中去取  **public** HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {  **this**.properties = properties;  }  @Bean  //给容器添加一个组件，这个组件的某些值需要从properties里面获取 @ConditionalOnMissingBean **public** CharacterEncodingFilter characterEncodingFilter() {  CharacterEncodingFilter filter = **new** OrderedCharacterEncodingFilter();  filter.setEncoding(**this**.properties.getCharset().name());  filter.setForceRequestEncoding(**this**.properties.shouldForce(org.springframework.boot.autoconfigure.http.HttpEncodingProperties.Type.REQUEST));  filter.setForceResponseEncoding(**this**.properties.shouldForce(org.springframework.boot.autoconfigure.http.HttpEncodingProperties.Type.RESPONSE));  **return** filter; } |

所有能配置的属性都在xxxxxProperties类中，这些都是实体类，用@ConfigurationProperties注解将配置文件的信息和实体类的属性进行绑定，所有能在配置文件中配置的属性都在这些类中

HttpEncodingProperties 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 { |

### SpringBoot的精髓

SpringBoot的精髓就是自动配置。

1. SpringBoot启动的时候会加载大量的自动配置类
2. 我们先看我们需要的功能有没有SpringBoot已有的自动配置类
3. 再看这个自动配置类到底配置了哪些组件（只要有我们要用的组件，我们就不用再配置了）
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，我们就可以再配置文件中指定这些属性的值。

XxxxxxAutoConfiguration：自动配置类，给容器中添加组件

XxxxxxProperties：封装配置文件中相关的属性

### @Conditional派生注解

作用：必须是@Conditional指定的条件成立，注解下的代码才生效

|  |  |
| --- | --- |
| @Conditional扩展注解 | 作用（判断是否满足当前指定条件） |
| @ConditionalOnJav | 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 |
| @ConditionalOnExpressio | 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 |

可以通过启用debug=true属性，来让控制台打印自动配置报告。

# 修改pom文件默认版本

在pom.xml的properties标签下加入版本信息即可覆盖默认版本配置

如<thymeleaf.version>3.0.9.RELEASE</thymeleaf.version> 将版本修改为3.0.9

# 日志

## 日志框架

常用日志框架：

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

|  |  |
| --- | --- |
| 日志门面 （日志的抽象层） | 日志实现 |
| JCL（Jakarta Commons Logging） SLF4j（Simple Logging  Facade for Java） jboss-logging | Log4j JUL（java.util.logging）  Log4j2 Logback |

用的时候左边选一个门面，右边选一个实现

SpringBoot底层使用的是Spring，Spring默认用的是JCL

SpringBoot选用slf4j和logback

## slf4j的使用

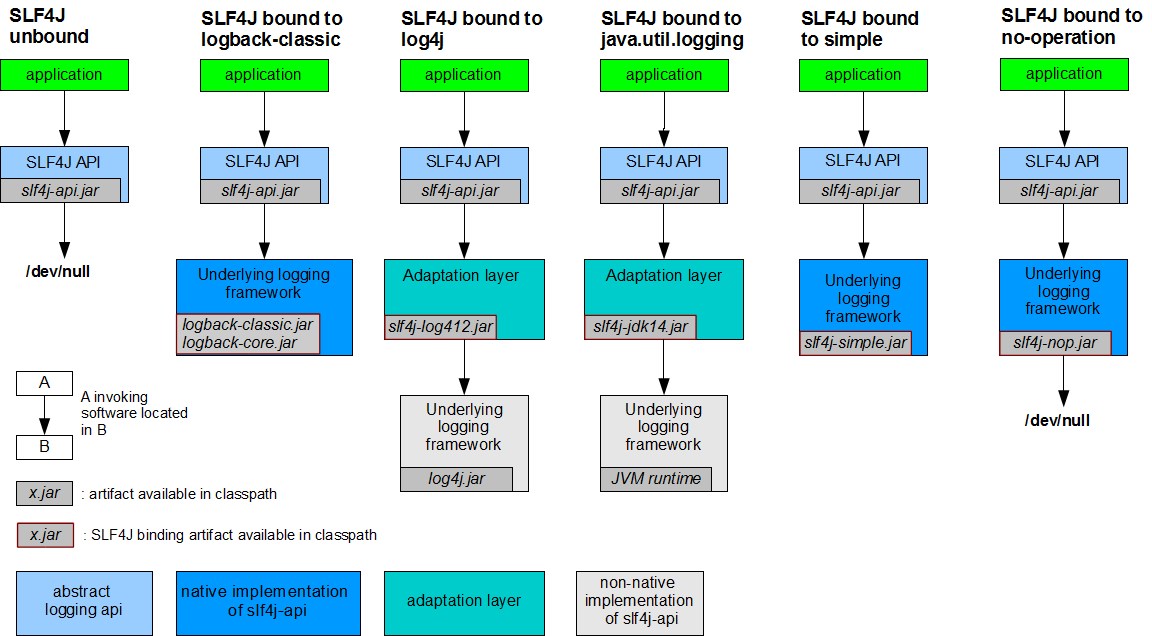
### 如何在系统中使用slf4j

以后在开发的时候，日志调用的方法，不应该直接调用实现类，而是调用日志抽象层里面的方法。

给项目导入slf4j的jar和logback的实现jar

|  |
| --- |
| import org.slf4j.Logger; import org.slf4j.LoggerFactory;  public class HelloWorld {   public static void main(String[] args) {     Logger logger = LoggerFactory.getLogger(HelloWorld.class);     logger.info("Hello World");   } } |

使用不同的日志实现，需要依赖的jar包：



各个日志实现框架的配置文件保持不变，使用slf4j只是提供一个抽象层的接口，配置文件还是写实现框架的配置文件

### 遗留问题

项目中用的是：slf4j+logback

而项目中还要集成一些其他的框架，这些框架内部都有自己的日志模块

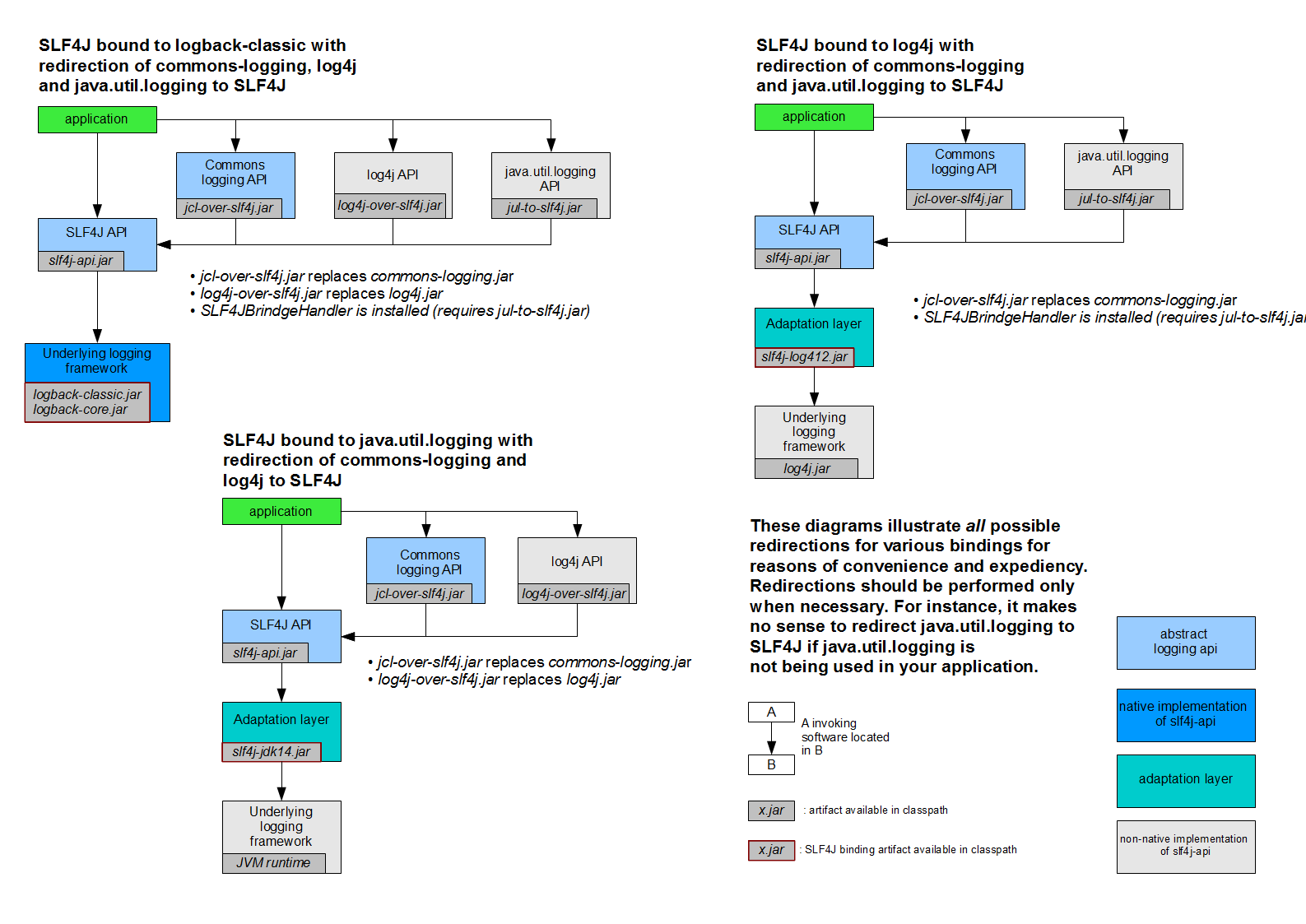
如：

Spring：commons-logging

Hibernate：jboss-logging

等等

所以要统一日志记录，即使是别的框架也要用slf4j+logback

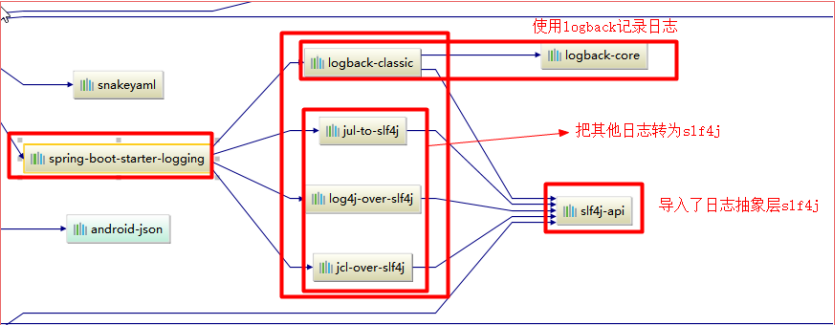


要统一日志记录，可以用对应的中间jar包替换掉原来的依赖包

### springBoot的日志依赖关系

SpringBoot依赖他来做日志功能

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  <**version**>2.0.3.RELEASE</**version**>  <**scope**>compile</**scope**> </**dependency**> |



总结：

1. SpringBoot底层也是使用Slf4j+logback的方式进行日志记录
2. SpringBoot也把其他的日志记录方式替换成了slf4j
3. 替换的中间包下的类名和被替换的实现包一样，只是内部实现代码中用的是slf4j的api调用系统统一的日志记录实现
4. 如果我们要引入的其他框架有默认的日志依赖，一定要移除掉

如,spring框架的commons-logging被springboot移除掉

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring‐core</artifactId>  <exclusions>  <exclusion>  <groupId>commons‐logging</groupId>  <artifactId>commons‐logging</artifactId>  </exclusion>  </exclusions>  </dependency> |

SpringBoot能自动适配所有的日志，而且底层使用slf4j+logback的方式记录日志，我们唯一需要做的是引入其他框架的时候，把这个框架依赖的日志框架排除掉

### 日志的使用

#### 默认配置

SpringBoot默认帮我们配置好了日志

|  |
| --- |
| *//日志记录器* Logger **logger** = LoggerFactory.*getLogger*(getClass());  @Test **public void** contextLoads() {    *//日志的级别，由低到高 trace<debug<info<warn<error  //可以调整需要输出的日志级别,只打印指定级别及更高级别的日志  //跟踪信息* **logger**.trace(**"这是trace日志..."**);   *//调试信息* **logger**.debug(**"这是debug日志..."**);   *//springboot默认的日志输出级别为info  //自定义的一些输出信息* **logger**.info(**"这是info日志..."**);   *//警告日志* **logger**.warn(**"这是warn日志..."**);   *//错误日志* **logger**.error(**"这是error日志..."**);  } |

修改日志默认配置的常用配置

|  |
| --- |
| *#指定com.example.springboot002包下的日志输出级别为trace #如果没有指定那么就使用springboot的默认级别，info* **logging.level.com.example.springboot002**=**trace** *#不指定路径，只有文件名，在当前项目的根目录下生成日志文件springboot.log #logging.file=springboot.log  #指定路径下生成一个日志文件D:/springboot.log #logging.file=D:/springboot.log  #指定日志文件目录，在当前磁盘下生成一个spring/log文件夹，日志文件名为springboot默认的spring.log #logging.path=/spring/log  #logging.file和logging.path都不指定，日志只在控制台打印 #logging.file和logging.path同时指定，只有logging.file生效  #指定在控制台输出的日志格式* **logging.pattern.console**=**%d{yyyy-MM-dd HH:mm:ss.SSS} [ %thread ] - [ %-5level ] [ %logger{50} : %line ] - %msg%n** *#%d指定日期 #%thread 线程名 #%-5level -5：从左边开始最多显示5个字符 %level 日志级别 #%logger{50} logger所在的全类名，最多50个字符 #%msg 消息 #%n 换行  #指定在文件中输出的日志格式* **logging.pattern.file**= |

#### 指定配置

Springboot要使用自己的日志文件，只需要把指定的配置文件名放到类路径下

Logback ：logback-spring.xml logback-spring.grooyv logback.xml logback.grooyv

log4j2：log4j2-spring.xml log4j2.xml

JDK(Java Util Logging) logging.properties

推荐使用logback-spring.xml

logback.xml 直接被日志框架识别了，不能使用一些特殊的配置

Logback-spring.xml 日志框架识别不了，由springBoot来解析日志的配置文件，就可以使用springProfile标签

|  |
| --- |
| <springProfile name=”dev”>  //里面的配置只在指定的环境中生效  </springProfile> |

### 切换日志框架

可以按照slf4j日志适配图进行相关的切换

#### 自己进行依赖切换

slf4j+log4j

1. 去除logback的依赖
2. 去除适配log4j的依赖------将log4j转为slf4j
3. 添加使用log4j的适配包

#### 使用starter进行切换

Springboot提供了spring-boot-starter-logging和spring-boot-starter-log4j2两个启动器

spring-boot-starter-logging：slf4j+logback

spring-boot-starter-log4j2：slf4j+log4j2

# Web开发

使用springboot步骤

1. 创建springboot应用，选择我们需要的模块
2. Springboot已经默认配置好了这些场景，我们只需要进行少量的配置就可以运行起来
3. 自己编写业务代码

## springboot对静态资源的映射规则

WebMvcAutoConfiguration类部分源码 静态资源映射

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

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

Webjars：以jar包的方式引入静态资源

例如要引入jquery，就在pom文件中添加jquery的webjar依赖

可以再webjar官网找到对应的maven依赖

1. 、**/\*\*** 访问当前项目的任何资源

|  |
| --- |
| **"classpath:/META-INF/resources/"**  **"classpath:/resources/"**  **"classpath:/static/"**  **"classpath:/public/"**  **"/" 根路径** |

默认在以上路径下寻找静态资源，以上路径称为静态资源文件夹

1. WebMvcAutoConfiguration类部分源码 欢迎页映射

|  |
| --- |
| @Bean **public** WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext) {  **return new** WelcomePageHandlerMapping(**new** TemplateAvailabilityProviders(applicationContext), applicationContext, **this**.getWelcomePage(), **this**.mvcProperties.getStaticPathPattern()); } |

静态资源文件夹下的index.html文件 被”/\*\*”映射

如果没有欢迎页，会出现404页面

4）WebMvcAutoConfiguration类部分源码 配置标签图标

|  |
| --- |
| @Configuration @ConditionalOnProperty(  value = {**"spring.mvc.favicon.enabled"**},  matchIfMissing = **true** ) **public static class** FaviconConfiguration **implements** ResourceLoaderAware {  **private final** ResourceProperties resourceProperties;  **private** ResourceLoader resourceLoader;   **public** FaviconConfiguration(ResourceProperties resourceProperties) {  **this**.resourceProperties = resourceProperties;  }   **public void** setResourceLoader(ResourceLoader resourceLoader) {  **this**.resourceLoader = resourceLoader;  }   @Bean  **public** SimpleUrlHandlerMapping faviconHandlerMapping() {  SimpleUrlHandlerMapping mapping = **new** SimpleUrlHandlerMapping();  mapping.setOrder(-2147483647);  mapping.setUrlMap(Collections.singletonMap(**"\*\*/favicon.ico"**, **this**.faviconRequestHandler()));  **return** mapping;  }   @Bean  **public** ResourceHttpRequestHandler faviconRequestHandler() {  ResourceHttpRequestHandler requestHandler = **new** ResourceHttpRequestHandler();  requestHandler.setLocations(**this**.resolveFaviconLocations());  **return** requestHandler;  }   **private** List<Resource> resolveFaviconLocations() {  String[] staticLocations = WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter.getResourceLocations(**this**.resourceProperties.getStaticLocations());  List<Resource> locations = **new** ArrayList(staticLocations.length + 1);  Stream var10000 = Arrays.stream(staticLocations);  ResourceLoader var10001 = **this**.resourceLoader;  **this**.resourceLoader.getClass();  var10000.map(var10001::getResource).forEach(locations::add);  locations.add(**new** ClassPathResource(**"/"**));  **return** Collections.unmodifiableList(locations);  } } |

所有的**\*\*/favicon.ico** 都是在静态资源文件夹下寻找

所有的静态映射配置都在ResourceProperties类中

|  |
| --- |
| @ConfigurationProperties(  prefix = **"spring.resources"**,  ignoreUnknownFields = **false** ) **public class** ResourceProperties {  } |

所以可以再配置文件中修改spring.resources修改相关的配置

## 模板引擎

数据和页面通过模板引擎绑定起来，生成一个我们想要展示的页面

常用的有：JSP、Velocity、Freemarker、Thymeleaf等

SpringBoot默认不支持JSP，推荐使用Thymeleaf

语法更简单，功能更强大

### 引入Thymeleaf

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

### Thymeleaf使用&语法

ThymeleafProperties类 部分代码：

|  |
| --- |
| **public class** ThymeleafProperties {  **private static final** Charset DEFAULT\_ENCODING;  **public static final** String DEFAULT\_PREFIX = **"classpath:/templates/"**;  **public static final** String DEFAULT\_SUFFIX = **".html"**; |

只要将html页面放到classpath:/templates/下面，thymeleaf就会自动渲染

1. 导入thymeleaf的名称空间

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

1. 使用thymeleaf

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Title</**title**> </**head**> <**body**>  <**h1**>成功！</**h1**>   *<!-- th:text 将div的文本内容设置为指定内容 -->* <**div th:text="${name}"**></**div**> </**body**> </**html**> |

### Thymeleaf语法规则

#### th: + html任意属性

th: + 任意html属性： 替换原来属性的值

如：

th:id=”id” 替换元素id

th:class=”class” 替换元素class

th:text=”张三” 替换元素的文本

1. Fragment inclusion 片段包含

th:insert th:replace 相当于jsp的jsp:include

1. fragment iteration 遍历

th:each 相当于jsp的c:forEach

1. Conditional evaluation 条件判断

th:if th:unless th:switch th:case 相当于jsp的c:if

1. Local variable definition 声明变量

th:object th:with 相当于c:set

1. General attribute modification 任意属性修改 支持prepend、append添加属性

th:attr th:attrprepend th:attrappend

1. Specific attribute modification 修改指定属性

th:value th:src th:href th:id th:class 等等

1. Text(tag body modification) 修改标签体文本内容的

th:text--转义特殊字符，当普通字符串处理 th:utext--不转义特殊字符

1. Fragment specification 声明片段

th:fragment

1. Fragment removal 移除片段

th:remove

#### 表达式

1. ${...} 获取变量值 就是OGNL表达式
   1. 获取对象、集合的值，调用对象的方法
   2. 使用内置的基本对象 $(#ctx.....) 通过#获取内置基本对象
      1. ctx 当前的上下文对象
      2. vars 当前上下文的变量值
      3. locale 区域信息
      4. request ---HttpServletRequest
      5. response ---HttpServletResponse
      6. Session ----HttpSession
      7. ServletContext ----ServletContext
   3. 使用内置的一些工具对象

如String number 等等一些工具类，详细参考thymeleaf文档

1. \*{...} 变量选择表达式，与${}功能上是一样的

特殊用法，可以配合th:object使用

|  |
| --- |
| <div th:object="${session.user}">  <p>Name: <span th:text="\*{firstName}">Sebastian</span>.</p>  <p>Surname: <span th:text="\*{lastName}">Pepper</span>.</p>  <p>Nationality: <span th:text="\*{nationality}">Saturn</span>.</p>  </div> |

1. #{...} 获取国际化内容
2. @{...}定义url链接

@{/order/process(execId=${execId},execType='FAST')}

直接用/就代表当前路径下，所有参数直接用小括号括起来，多个参数用逗号分隔，参数可以用${...}获取变量传参

1. ~{...} 片段引用表达式

以上是thymeleaf的五种表达式

在表达式中支持：

1.Literals 字面量：字符串、数字、布尔值、null、多个值用逗号分隔等

Text literals: 'one text' , 'Another one!' ,…

Number literals: 0 , 34 , 3.0 , 12.3 ,…

Boolean literals: true , false

Null literal: null

Literal tokens: one , sometext , main ,…

Text operations: 文本操作 +号连接符、用${...}选择变量

String concatenation: +

Literal substitutions: |The name is ${name}|

Arithmetic operations: 数学运算

Binary operators: + , - , \* , / , %

Minus sign (unary operator): -

Boolean operations: 布尔运算

Binary operators: and , or

Boolean negation (unary operator): ! , not

Comparisons and equality: 比较运算

Comparators: > , < , >= , <= ( gt , lt , ge , le )

Equality operators: == , != ( eq , ne )

Conditional operators: 条件运算 支持三元运算符

If-then: (if) ? (then)

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

Default: (value) ?: (defaultvalue)

Special tokens: 特殊标志 用\_表示没有操作

No-Operation: \_

[[${....}]] 相当于th:text

[(${....})] 相当于th:utext

## SpringMVC 自动配置

#### 自动配置

springBoot 自动配置好了springMVC使用的默认配置

以下是springBoot对springMVC的配置：

* Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.

自动配置了ViewResolver（视图解析器）根据方法的返回值的到视图对象（View）

视图对象决定如何渲染（转发？重定向？）

 ContentNegotiatingViewResolver：组合所有的视图解析器，

通过getCandidateViews方法将容器中所有的视图解析器都获取到，

然后通过getBestView选择最适合的一个视图解析器

我们要自定义一个视图解析器，只需要将我们的ViewResolver添加到容器中即可

* Support for serving static resources, including support for WebJars (covered [later in this document](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-static-content" \o "27.1.5 Static Content))).
* Static index.html support.
* Custom Favicon support (covered [later in this document](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-favicon" \o "27.1.7 Custom Favicon)).

支持静态资源的配置，静态资源文件夹、webjars、首页及图标favicon.ico

* Automatic registration of Converter, GenericConverter, and Formatter beans.

自动注册了Converter：转换器，自动进行类型转换

Formatter：格式化器，如格式化日期等

* Support for HttpMessageConverters (covered [later in this document](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-message-converters" \o "27.1.2 HttpMessageConverters)).

HttpMessageConverters：SpringMVC用来转换http请求和响应的转换器

* Automatic registration of MessageCodesResolver (covered [later in this document](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-message-codes" \o "27.1.4 MessageCodesResolver)).

MessageCodesResolver：定义错误代码生成规则

* Automatic use of a ConfigurableWebBindingInitializer bean (covered [later in this document](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/" \l "boot-features-spring-mvc-web-binding-initializer" \o "27.1.9 ConfigurableWebBindingInitializer)).

ConfigurableWebBindingInitializer：初始化WebDataBinder

WebDataBinder：用来将请求参数绑定到javabean的

#### 扩展SpringMVC

|  |
| --- |
| If you want to keep Spring Boot MVC features and you want to add additional [MVC configuration](https://docs.spring.io/spring/docs/5.1.0.BUILD-SNAPSHOT/spring-framework-reference/web.html" \l "mvc" \t "https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/_top) (interceptors, formatters, view controllers, and other features), you can add your own @Configuration class of type WebMvcConfigurer but **without** @EnableWebMvc. If you wish to provide custom instances of RequestMappingHandlerMapping, RequestMappingHandlerAdapter, or ExceptionHandlerExceptionResolver, you can declare a WebMvcRegistrationsAdapter instance to provide such components. |

扩展方式：

编写一个配置类（用@configuration注解标注），是WebMvcConfigurer类型。不能标注@EnableWebMvc注解

例如：

|  |
| --- |
| *//通过实现WebMvcConfigurer接口来实现springMVC扩展 //注意：WebMvcConfigurerAdapter抽象类已过时 // WebMvcConfigurationSupport类一旦存在，对springMVC的默认配置全部失效 //所以只需要实现WebMvcConfigurer接口接口即可，该接口中的方法都使用了JDK1.8的default关键字 //不需要实现所有方法，需要用哪个方法就实现哪个方法* @Configuration **public class** MyMvcConfig **implements** WebMvcConfigurer {   @Override  **public void** addViewControllers(ViewControllerRegistry registry){   *//super.addViewControllers(registry);   //浏览器发送/demo请求，回来到success页面* registry.addViewController(**"/demo"**).setViewName(**"success"**);  } } |

SpringBoot既保留了默认配置，也能使用我们的扩展配置

原理：

1. WebMvcAutoConfiguration SpringMVC的自动配置类
2. 在做其他自动配置的时候会导入 EnableWebMvcConfiguration
3. EnableWebMvcConfiguration 的父类DelegatingWebMvcConfiguration

|  |
| --- |
| **public class** DelegatingWebMvcConfiguration **extends** WebMvcConfigurationSupport {  **private final** WebMvcConfigurerComposite configurers = **new** WebMvcConfigurerComposite();   **public** DelegatingWebMvcConfiguration() {  }   @Autowired(  required = **false** )  //从容器中获取所有的WebMvcConfigurer  **public void** setConfigurers(List<WebMvcConfigurer> configurers) {  **if** (!CollectionUtils.isEmpty(configurers)) {  **this**.configurers.addWebMvcConfigurers(configurers);  }  } |

通过一个WebMvcConfigurerComposite configurers 调用他的方法

|  |
| --- |
| **public void** addInterceptors(InterceptorRegistry registry) {  Iterator var2 = **this**.delegates.iterator();   **while**(var2.hasNext()) {  WebMvcConfigurer delegate = (WebMvcConfigurer)var2.next();  delegate.addInterceptors(registry);  } } |

在WebMvcConfigurerComposite 中是遍历所有的WebMvcConfigurer，每一个都调用一次该方法，把扩展组件全部加载进来

容器中所有的WebMvcConfigurer都会起作用，包括我们自定义的WebMvcConfigurer

#### 全面接管SpringMVC

|  |
| --- |
| If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc. |

在我们自己的配置类上加上@EnableWebMvc注解，那么所有的SpringMvc自动配置都不会生效了，所有的配置都需要我们自己设置

## 如何修改SpringBoot的默认配置

1. SpringBoot在自动配置很多组件的时候，都会先看容器中是否有用户自定义的组件，如果有，就用用户自定义的。如果没有在自动配置。有些组件可以有多个，如ViewResolver，SpringBoot会将用户自定义的和默认配置的组合起来。用户通过@Bean、@component注解给容器添加组件。
2. 在SpringBoot会有非常多的xxxxConfigurer，帮助我们进行扩展配置，如WebMvcConfigurer
3. 还会有非常多的xxxxCustomizer，帮助我们定制配置，如WebServerFactoryCustomizer用来定制嵌入式servlet容器的相关配置

## RestfulCRUD

#### 设置默认首页

|  |
| --- |
| *//自己向容器添加一个WebMvcConfigurer，将“/”请求映射到index.html上* @Bean **public** WebMvcConfigurer myWebMvcConfigurer(){  WebMvcConfigurer config = **new** WebMvcConfigurer() {   @Override  **public void** addViewControllers(ViewControllerRegistry registry){  registry.addViewController(**"/"**).setViewName(**"index"**);  registry.addViewController(**"/index.html"**).setViewName(**"index"**);  }  };  **return** config; } |

#### 添加前端框架

如果要添加bootstrap、jquery等资源，通过webjar的方式用maven依赖导入

然后通过默认的webjar映射路径/webjar/\*\*路径映射到对应的资源

#### 设置项目名

修改application.properties文件

|  |
| --- |
| *#设置端口号* **server.port**=**8080** *#设置项目名* **server.servlet.context-path**=**/demo** |

#### 国际化

原来的步骤：

1. 编写国际化配置文件
2. 使用ResourceBundleMessageSource来管理国际化资源文件
3. 在JSP页面使用fmt:message去除国际化内容

springBoot使用：

1. 编写国际化配置文件，抽取页面需要国际化的内容
   1. 创建index.html页面的国际化配置文件index.properties
   2. 中文和英文的index\_zh\_CN.properties index\_en\_US.properties
   3. 这时idea会自动识别在进行国际化配置，可进入ResourceBundle视图进行国际化配置
2. SpringBoot自动配置好了ResourceBundleMessageSource

在MessageSourceAutoConfiguration类中的部分源码

|  |
| --- |
| @Bean @ConfigurationProperties(  prefix = **"spring.messages"** ) **public** MessageSourceProperties messageSourceProperties() {  **return new** MessageSourceProperties(); }  @Bean **public** MessageSource messageSource() {  MessageSourceProperties properties = **this**.messageSourceProperties();  ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();  **if** (StringUtils.hasText(properties.getBasename())) {  messageSource.setBasenames(StringUtils.commaDelimitedListToStringArray(StringUtils.trimAllWhitespace(properties.getBasename())));  }   **if** (properties.getEncoding() != **null**) {  messageSource.setDefaultEncoding(properties.getEncoding().name());  }   messageSource.setFallbackToSystemLocale(properties.isFallbackToSystemLocale());  Duration cacheDuration = properties.getCacheDuration();  **if** (cacheDuration != **null**) {  messageSource.setCacheMillis(cacheDuration.toMillis());  }   messageSource.setAlwaysUseMessageFormat(properties.isAlwaysUseMessageFormat());  messageSource.setUseCodeAsDefaultMessage(properties.isUseCodeAsDefaultMessage());  **return** messageSource; } |

通过setBasenames设置国际化资源文件的基础名（去掉语言国家代码后的名字）

在MessageSourceProperties有默认的基础名：

**private** String basename = **"messages"**;

也就是说如果我们将国际化配置在类路径下的messages.properties文件中，将不需要做任何配置直接使用

如果基础名不是messages，那么就需要在application.properties文件中进行配置

|  |
| --- |
| *#配置资源文件的基础名 在类路径下的i18n文件夹下的index.properties* **spring.messages.basename**=**i18n.index** |

1. 去页面获取国际化的值

在thymeleaf中有#{...}表达式专门用来处理国际化的

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Title</**title**> </**head**> <**body**>  <**p**>[[#{login.tip}]]</**p**> </**body**> </**html**> |

这样就会根据浏览器语言信息显示不同的文本内容

1. 实现按钮切换语言

原理：

国际化Locale（区域信息对象）；LocaleResolver（获取区域信息对象）

WebMvcAutoConfiguration中关于LocaleResolver的自动配置源码

|  |
| --- |
| @Bean @ConditionalOnMissingBean @ConditionalOnProperty(  prefix = **"spring.mvc"**,  name = {**"locale"**} ) **public** LocaleResolver localeResolver() {  **if** (**this**.mvcProperties.getLocaleResolver() == org.springframework.boot.autoconfigure.web.servlet.WebMvcProperties.LocaleResolver.FIXED) {  **return new** FixedLocaleResolver(**this**.mvcProperties.getLocale());  } **else** {  AcceptHeaderLocaleResolver localeResolver = **new** AcceptHeaderLocaleResolver();  localeResolver.setDefaultLocale(**this**.mvcProperties.getLocale());  **return** localeResolver;  } } |

如果设置成了FIXED那么将会固定区域化信息，否则将会从请求头中获取

可以自己写一个区域信息解析器不从请求头中获取，而从请求连接中获取

|  |
| --- |
| **public class** MyLocaleResolver **implements** LocaleResolver {  @Override  **public** Locale resolveLocale(HttpServletRequest httpServletRequest) {  String l = httpServletRequest.getParameter(**"l"**);  Locale locale = Locale.*getDefault*();  **if**(!StringUtils.*isEmpty*(l)){  String[] split = l.split(**"\_"**);  locale = **new** Locale(split[0],split[1]);  }  **return** locale;  }   @Override  **public void** setLocale(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Locale locale) {  } } |

将自己的LocaleResolver 通过@Bean注解添加到容器中取代自动配置的LocaleResolver 即可

#### 登录

开发期间模板引擎页面修改后，要实时生效：

1. 禁用thymeleaf模板引擎缓存

|  |
| --- |
| *#禁用thymeleaf缓存* **spring.thymeleaf.cache**=**false** |

1. 修改完成后重新编译，就能够刷新页面了

#### 用拦截器进行登录检查

1. 通过实现HandlerInterceptor接口定义一个登录拦截器

|  |
| --- |
| */\*\*  \* 通过实现HandlerInterceptor来定义一个拦截器，进行登录检查  \*/* **public class** LoginHandlerIntercepter **implements** HandlerInterceptor {   *//在目标方法前执行* @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {   Object user = request.getSession().getAttribute(request.getParameter(**"sessionId"**));  **if**(user == **null**){  *//未登录* **return false**;  }**else**{  *//已登录* **return true**;  }  } } |

1. 在自己的WebMvcConfigurer中重写addInterceptors方法注册拦截器

|  |
| --- |
| *//注册拦截器* @Override **public void** addInterceptors(InterceptorRegistry registry) {   *//addPathPatterns添加要拦截的映射路径 "/\*\*"表示拦截所有请求  //excludePathPatterns用来排除部分请求  //静态资源不用排除，因为springBoot已经做好了资源映射* registry.addInterceptor(**new** LoginHandlerIntercepter()).addPathPatterns(**"/\*\*"**).excludePathPatterns(**"/index.html"**); } |

#### Restful与普通请求的区别

Restful请求通过http请求方法来区分增删改查，而普通请求通过url区分增删改查

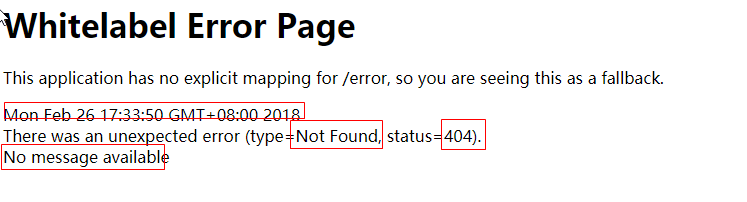
|  |  |  |
| --- | --- | --- |
|  | Restful | 普通 |
| 查找 | emp/{id}---GET | getEmp?id=xxx |
| 修改 | emp---PUT | updateEmp?id=xxx&xxx=xxx |
| 添加 | emp---POST | addEmp?id=xxx&xxx=xxx |
| 删除 | emp/{id}---DELETE | deleteEmp?id=xxx |

## 错误处理机制

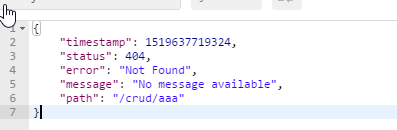
#### SpringBoot默认错误处理机制

默认效果：

1. 返回一个默认的错误页面



1. 如果不是浏览器而是其他客户端，返回一个默认的json数据



原理：可以参照ErrorMvcAutoConfiguration；错误处理的自动配置

在这个自动配置类中，主要给容器添加了以下组件：

1. DefaultErrorAttributes：

默认的共享的错误响应内容

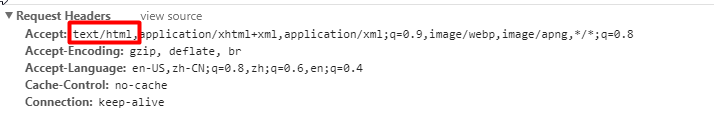
|  |
| --- |
| **public** Map<String, Object> getErrorAttributes(WebRequest webRequest, **boolean** includeStackTrace) {  Map<String, Object> errorAttributes = **new** LinkedHashMap();  errorAttributes.put(**"timestamp"**, **new** Date());  **this**.addStatus(errorAttributes, webRequest);  **this**.addErrorDetails(errorAttributes, webRequest, includeStackTrace);  **this**.addPath(errorAttributes, webRequest);  **return** errorAttributes; }  //**timestamp 时间戳**  **//status 状态码**  **//error 错误信息**  **//exception 异常对象**  **//message 异常消息**  **//errors JSR303数据校验的错误都在这里** |

1. BasicErrorController：//就是一个controller，如果配置文件没配置默认处理/error请求

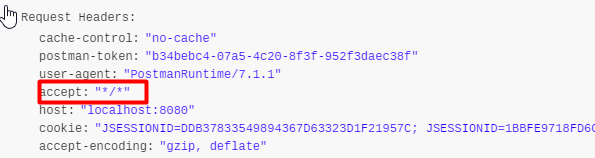
|  |
| --- |
| @Controller @RequestMapping({**"${server.error.path:${error.path:/error}}"**}) **public class** BasicErrorController **extends** AbstractErrorController {  //1.产生html页面数据  @RequestMapping(  produces = {**"text/html"**} ) **public** ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {  HttpStatus status = **this**.getStatus(request);  Map<String, Object> model = Collections.unmodifiableMap(**this**.getErrorAttributes(request, **this**.isIncludeStackTrace(request, MediaType.TEXT\_HTML)));  response.setStatus(status.value()); //去哪个页面作为错误页面，是通过resolveErrorView获得的modelAndView ，包含页面地址和页面//内容  ModelAndView modelAndView = **this**.resolveErrorView(request, response, status, model);  **return** modelAndView != **null** ? modelAndView : **new** ModelAndView(**"error"**, model); }  //产生json数据 @RequestMapping @ResponseBody **public** ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {  Map<String, Object> body = **this**.getErrorAttributes(request, **this**.isIncludeStackTrace(request, MediaType.ALL));  HttpStatus status = **this**.getStatus(request);  **return new** ResponseEntity(body, status); }  } |

两个方法返回错误数据，是根据请求头的accept来选择的

浏览器请求优先接收text/html数据：



而其他客户端没有指明优先接收什么数据



1. ErrorPageCustomizer：

|  |
| --- |
| @Value(**"${error.path:/error}"**) **private** String path = **"/error"**;  //系统出现错误以后来到error请求进行处理；（相当于web.xml注册的错误页面规则）  //如果配置文件没有配置默认为/error |

1. DefaultErrorViewResolver：容器默认的错误视图解析器

|  |
| --- |
| **public** ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, Object> model) {  ModelAndView modelAndView = **this**.resolve(String.valueOf(status), model);  **if** (modelAndView == **null** && SERIES\_VIEWS.containsKey(status.series())) {  modelAndView = **this**.resolve((String)SERIES\_VIEWS.get(status.series()), model);  }   **return** modelAndView; }  **private** ModelAndView resolve(String viewName, Map<String, Object> model) { //默认会去找”error/” + 状态码 页面  String errorViewName = **"error/"** + viewName; //如果有模板引擎，就用模板引擎来解析这个映射路径  TemplateAvailabilityProvider provider = **this**.templateAvailabilityProviders.getProvider(errorViewName, **this**.applicationContext); //如果没有模板引擎，就在静态资源文件夹下寻找对应的资源  **return** provider != **null** ? **new** ModelAndView(errorViewName, model) : **this**.resolveResource(errorViewName, model); } |

步骤：

一但系统出现4xx或者5xx之类的错误；ErrorPageCustomizer就会生效（定制错误的响应规则）；就会来到/error请求；就会被BasicErrorController处理；

BasicErrorController根据accept请求头判断是返回html还是json数据

1. Html

通过resolveErrorView方法获取ModelAndView 对象，包含响应的页面和页面数据

resolveErrorView代码：

|  |
| --- |
| **protected** ModelAndView resolveErrorView(HttpServletRequest request, HttpServletResponse response, HttpStatus status, Map<String, Object> model) {  //通过迭代所有的ErrorViewResolver得到modelAndView，如果没有就返回null  //容器默认注册了一个DefaultErrorViewResolver，如果没有自定义的ErrorViewResolver  //那么就是用这个默认的错误视图解析器解析得到具体返回哪个错误页面  Iterator var5 = **this**.errorViewResolvers.iterator();  ModelAndView modelAndView;  **do** {  **if** (!var5.hasNext()) {  **return null**;  }  ErrorViewResolver resolver = (ErrorViewResolver)var5.next();  modelAndView = resolver.resolveErrorView(request, status, model);  } **while**(modelAndView == **null**);  **return** modelAndView; } |

#### 定制错误响应

1. 定制错误页面
2. 、有模板引擎的情况下；error/状态码; 【将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的error文件夹下】，发生此状态码的错误就会来到 对应的页面；

我们可以使用4xx和5xx作为错误页面的文件名来匹配这种类型的所有错误，精确优先（优先寻找精确的状态码.html）；

1. 、没有模板引擎的情况（模板引擎找不到这个错误页面），静态资源文件夹下找；
2. 以上都没有，就来到SpringBoot默认的错误页面
3. 定制错误数据
4. 处理自定义异常&返回自定义json数据

|  |
| --- |
| @ControllerAdvice  public class MyExceptionHandler {      @ResponseBody      @ExceptionHandler(UserNotExistException.class)      public Map<String,Object> handleException(Exception e){          Map<String,Object> map = new HashMap<>();          map.put("code","user.notexist");          map.put("message",e.getMessage());          return map;      }  }  //没有自适应效果，浏览器和客户端都返回的是json数据 |

1. 转发到/error进行自适应响应效果处理

|  |
| --- |
| @ExceptionHandler(UserNotExistException.class)      public String handleException(Exception e, HttpServletRequest request){           Map<String,Object> map = new HashMap<>();          //传入我们自己的错误状态码  4xx 5xx，否则就不会进入定制错误页面的解析流程          /\*\*           \* Integer statusCode = (Integer) request           .getAttribute("javax.servlet.error.status\_code");           \*/          request.setAttribute("javax.servlet.error.status\_code",500);          map.put("code","user.notexist");          map.put("message",e.getMessage());          //转发到/error          return "forward:/error";  }  //有自适应效果但是不能将我们自定义的数据携带出去 |

1. 、将我们的定制数据携带出去；

出现错误以后，会来到/error请求，会被BasicErrorController处理，响应出去可以获取的数据是由getErrorAttributes得到的（是AbstractErrorController（ErrorController）规定的方法）；

所以有两种解决办法：

1. 完全来编写一个ErrorController的实现类【或者是编写AbstractErrorController的子类】，放在容器中；
2. 自定义ErrorAttributes

页面上能用的数据，或者是json返回能用的数据都是通过errorAttributes.getErrorAttributes得到；

容器中DefaultErrorAttributes.getErrorAttributes()；默认进行数据处理的；

|  |
| --- |
| //给容器中加入我们自己定义的ErrorAttributes  @Component  public class MyErrorAttributes extends DefaultErrorAttributes {      @Override      public Map<String, Object> getErrorAttributes(RequestAttributes requestAttributes,  boolean includeStackTrace) {          Map<String, Object> map = super.getErrorAttributes(requestAttributes,  includeStackTrace);          map.put("company","atguigu");          return map;      }  } |

最终的效果：响应是自适应的，可以通过定制ErrorAttributes改变需要返回的内容

# 配置嵌入式Servlet容器

SpringBoot默认使用的是tomcat作为嵌入式的servlet容器

## 定制和修改servlet容器的相关配置

### application.properties

在application.properties中修改和server有关的配置

如：

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

### EmbeddedServletContainerCustomizer

在springBoot 1.x版本中可以编写一个EmbeddedServletContainerCustomizer：嵌入式的Servlet容器的定制器，来修改servlet容器的相关配置

|  |
| --- |
| @Bean  //一定要将这个定制器加入到容器中  public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer(){      return new EmbeddedServletContainerCustomizer() {          //定制嵌入式的Servlet容器相关的规则          @Override          public void customize(ConfigurableEmbeddedServletContainer container) {              container.setPort(8083);          }      };  } |

### WebServerFactoryCustomizer

在springBoot 2.x版本中使用WebServerFactoryCustomizer接口来定制tomcat配置

|  |
| --- |
| @Bean **public** WebServerFactoryCustomizer webServerFactoryCustomizer(){  **return new** WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>(){  @Override  **public void** customize(ConfigurableServletWebServerFactory server) {  server.setPort(5000);  }  }; } |

## 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。

### springBoot的Registration

springBoot提供了ServletRegistrationBean、FilterRegistrationBean、ServletListenerRegistrationBean 分别用于注册三大组件

注册servlet ServletRegistrationBean

|  |
| --- |
| @Bean **public** ServletRegistrationBean myServlet(){  ServletRegistrationBean servletRegistrationBean = **new** ServletRegistrationBean(**new** MyServlet(),**"/myServlet"**);  **return** servletRegistrationBean; } |

注册Listener ServletListenerRegistrationBean

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

注册filter FilterRegistrationBean

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

### 使用servlet3.0的注解

@WebServlet 、@WebListener、@WebFilter

Servlet

|  |
| --- |
| @WebServlet(name = **"IndexServlet"**,urlPatterns = **"/hello"**) **public class** IndexServlet **extends** HttpServlet {  @Override  **public void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().print(**"hello word"**);  resp.getWriter().flush();  resp.getWriter().close();  }   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **this**.doGet(req, resp);  } } |

Listener

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

Filter

|  |
| --- |
| @WebFilter(urlPatterns = **"/\*"**, filterName = **"indexFilter"**) **public class** IndexFilter **implements** Filter {   @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  System.***out***.println(**"init IndexFilter"**);  }   @Override  **public void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws** IOException, ServletException {  System.***out***.println(**"doFilter IndexFilter"**);  filterChain.doFilter(servletRequest,servletResponse);   }   @Override  **public void** destroy() {   } } |

需要配置一个核心的注解@ServletComponentScan,具体配置项如下，可以配置扫描的路径，将该注解添加到主程序即可

|  |
| --- |
| @SpringBootApplication @ServletComponentScan(**"com.example.springboot002.servlet"**) **public class** SpringBoot002Application {  } |

### 自动配置springMVC的DispatcherServlet

SpringBoot帮我们自动配置SpringMVC时，自动注册了DispatcherServlet

DispatcherServletAutoConfiguration中注册DispatcherServlet部分源码：

|  |
| --- |
| @Bean(  name = {**"dispatcherServletRegistration"**} ) @ConditionalOnBean(  value = {DispatcherServlet.**class**},  name = {**"dispatcherServlet"**} ) **public** ServletRegistrationBean<DispatcherServlet> dispatcherServletRegistration(DispatcherServlet dispatcherServlet) {  ServletRegistrationBean<DispatcherServlet> registration = **new** ServletRegistrationBean(dispatcherServlet, **new** String[]{**this**.serverProperties.getServlet().getServletMapping()});  //在 getServletMapping()方法中可以看到默认的拦截路径为”/”  //拦截所有请求，包括静态资源，但是不包括JSP “/\*”会拦截jsp  //在serverProperties中可以看到可以通过server.Servlet.path修改拦截路径  registration.setName(**"dispatcherServlet"**);  registration.setLoadOnStartup(**this**.webMvcProperties.getServlet().getLoadOnStartup());  **if** (**this**.multipartConfig != **null**) {  registration.setMultipartConfig(**this**.multipartConfig);  }   **return** registration; } |

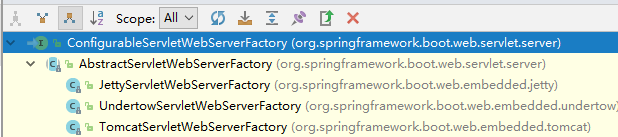
## 使用其他嵌入式Servlet容器

springBoot除了tomcat之外，还支持Jetty和Undertow两种servlet容器

Undertow不支持JSP,是一个高性能的非阻塞的servlet容器，并发性能很好

Jetty更适合开发长连接应用

从SpringBoot的容器配置定制器的继承树就可以看出springBoot支持的嵌入式servlet容器



使用其他容器的步骤

1. 在pom.xml的web启动器模块去掉默认tomcat的依赖

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

1. 添加其他容器的依赖

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

## 嵌入式Servlet容器的自动配置原理