SpringBoot入门

# 引言

**应用场景：**

1. web项目；
2. 微服务；
3. Spring应用。

**所需知识点：**

1. java 1.8
2. spring 4.0
3. maven

# 具体内容

## SpringBoot的主要特性

1. Spring Boot是伴随着Spring 4.0诞生的，继承了原有Spring框架的优秀基因，并非提升Spring；
2. 遵循“习惯优于配置”的原则，使用Spring Boot只需要很少的配置，大部分的时候我们直接使用默认的配置即可；
3. 对主流开发框架的无配置集成，自动整合第三方框架；
4. 独立运行的Spring项目Spring Boot可以以jar包的形式独立的运行，使用：java –jar xxx.jar就可以成功的运行项目，或者在应用项目的主程序中运行main函数即可；
5. 内嵌的Servlet容器，嵌入式Tomcat，Jetty容器；
6. 提供starter简化Maven配置，基本上可以做到自动化配置，高度封装开箱即用。提供了一系列的starter pom用来简化我们的Maven依赖；
7. 应用监控，Spring Boot提供了基于http、ssh、telnet对运行时项目进行监控；
8. 无代码生成和XML配置，纯Java的配置方式，很简单，很方便；
9. 分布式开发与Spring Cloud的天然微服务；
10. 带来了脚本语言开发的效率，但是Spring Boot并没有让我们以为这是一个新技术，都是Java EE开发者常见的技术。

## SpringBoot的不足

1. 坑有些多，无从下手，文档略少；
2. 版本迭代速度很快；
3. 习惯问题，spring boot应用喜欢把配置都写到代码里，有时会带来混乱；
4. 对spring技术不认可。

## SpringBoot-HelloWorld

1、@RestController = @Controller + @RquestMapping；

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.5.RELEASE</version>  </parent> |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> |
| **package** org.dougin.demo;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** HelloWorldController {  @RequestMapping("/hello")  **public** String hello() {  **return** "Hello World SpringBoot !";  }  } |
| **package** org.dougin.demo;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |

## SpringBoot核心

### 基本配置

#### 入口类

一般是以\*Application开头的入库类，并且包含一个main()方法，使用SpringApplication.run(App.class, args)进行启动项目。在入口类中使用核心注解@SpringBootApplication。

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @Configuration  @EnableAutoConfiguration  @ComponentScan(excludeFilters = @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class))  public @interface SpringBootApplication {  /\*\*  \*Exclude specific auto-configuration classes such that they will never be applied.  \*@return the classes to exclude  \*/  Class<?>[] exclude() default {};  …  } |
| @SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan |
| **package** org.dougin.demo;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;  **import** org.springframework.context.annotation.ComponentScan;  **import** org.springframework.context.annotation.Configuration;  //@SpringBootApplication  @Configuration  @EnableAutoConfiguration  @ComponentScan  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |

@Configuration：提到@Configuration就要提到它的搭档@Bean，使用者两个注解就可以创建一个简单的Spring配置类，可以用来代替相应的XML配置文件。

|  |
| --- |
| XML文件定义 |
| <beans>  <bean id=”car” class=”com.test.Car”>  <property name=”wheel” ref=”wheel”/>  </bean>  <bean id=”wheel” class=”com.test.Wheel”/>  </beans> |
| Class类定义 |
| @Configuration  Public class Conf {  @Bean  public Car car() {  Car car = new Car();  car.setWheel(wheel());  return car;  }  @Bean  public Wheel wheel() {  return new Wheel();  }  } |

@EnableAutoConfiguration：能够自动配置Spring的上下文，试图猜测和配置你想要的bean类，通常会自动根据你的类路径和你的bean定义自动配置。

比如：添加了spring-boot-starter-web依赖，就会自动添加springmvc和tomcat的依赖，那么spring boot就会对springmvc和tomcat进行默认配置。

@ComponentScan：会自动扫描指定包下的全部标有@Component的类，并注册成bean，当然包括@Component下的子注解@Service，@Repository，@Controller。

#### 关闭特定配置

spring-boot-autoconfigure.jar这个包里的配置是springboot的所有自动配置项。这个注解可以禁用springboot自带的配置，关闭默认的配置（后面跟上类名，可多个使用）：

* @SpringBootApplication(exclude={DataSourceAutoConfiguration.class,DataSourceTransactionManagerAutoConfiguration.class})
* @SpringBootApplication(exclude=MyBatisAutoConfiguration.class)
* @SpringBootApplication(exclude=MongoAutoConfiguration.class)

#### 配置文件

Spring Boot的默认属性配置文件是resources下的application.properties;

也可以使用application.yml来配置默认参数。

如果是使用Spring Boot自带的配置的话，配置即使用，无需再单独去声明内容。

|  |
| --- |
| application.properties: |
| server.port=8081 |

|  |
| --- |
| application.yml: |
| server:  port:8081 |

也可以使用自己习惯的配置，但是这种方式不太安全，有可能会和自带的配置或第三方配置冲突，例如：

|  |
| --- |
| test.name=zhangsan  test.age=30  Controller:  @Value(“${test.name}”)  private String name;  @Value(“${test.age}”)  private String age; |

|  |
| --- |
| **package** org.dougin.demo;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  @Component  @ConfigurationProperties(prefix="user",~~locations~~={"classpath:user.properties"})  **public** **class** UserConfig {  **private** String name;  **private** **int** age;  **public** String getName() { **return** name; }  **public** **void** setName(String name) { **this**.name = name; }  **public** **int** getAge() { **return** age; }  **public** **void** setAge(**int** age) { **this**.age = age; }  } |
| user.properties:  user.name=Dougin.Wei  user.age=25 |

Spring boot默认使用properties文件、yml文件或者命令行参数作为外部配置。

比如：java –jar a.jar --server.port=8081

#### 定制启动图案

关闭banner

Banner是什么？其实就是启动时显示的图案，如果不想显示它，直接在配置文件里配置即可关闭，如下：

|  |
| --- |
| spring.main.banner-mode=off |

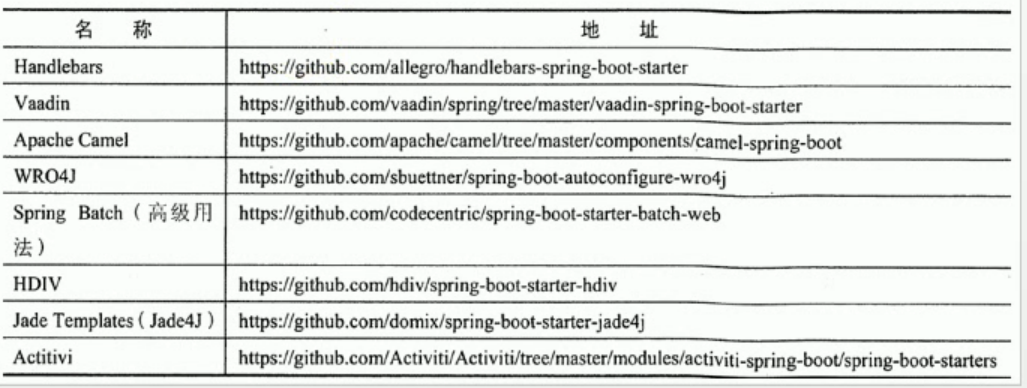
修改banner

我们也可以定制自己的启动图案，通过字符生成网站生成好看的字母或图案字符来显示自己的启动图案。通过网站<http://patorjk.com/software/taag/>进行定制显示字符图案。

1. 在src/main/resources下新建一个banner.txt文件；
2. 将生成好的字符复杂到banner.txt文件中；
3. 启动即可显示自定义图案了。

#### starter

|  |  |
| --- | --- |
| spring-boot-starter-hateoas | 通过spring-hatcoas对基于HATEOAS的REST形式的网络服务的支持 |
| spring-boot-starter-hornetq | 通过HornetQ对JMS的支持 |
| spring-boot-starter-integration | 对系统集成框架spring-integration的支持 |
| spring-boot-starter-jdbc | 对JDBC数据库的支持 |
| spring-boot-starter-jersey | 对Jersery REST形式的网络服务的支持 |
| spring-boot-starter-jta-atomikos | 通过Atomikos对分布式事务的支持 |
| spring-boot-starter-jta-bitronix | 通过Bitronix对分布式事务的支持 |
| spring-boot-starter-mail | 对javax.mail的支持 |
| spring-boot-starter-mobile | 对spring-mobile的支持 |
| spring-boot-starter-mustache | 对Mustache模板引擎的支持 |
| spring-boot-starter-redis | 对键值对内存数据库Redis的支持，包含spring-redis |
| spring-boot-starter-security | 对spring-security的支持 |
| spring-boot-starter-social-facebook | 通过spring-social-facebook对Facebook的支持 |
| spring-boot-starter-social-linkedin | 通过spring-social-linkedin对Linkedin的支持 |
| spring-boot-starter-social-twitter | 通过spring-social-twitter对Twitter的支持 |
| spring-boot-starter-test | 对常用的测试框架JUnit、Hamcrest和Mockito的支持，包含spring-test模块 |
| spring-boot-starter-thymeleaf | 对Thymeleaf模块引擎的支持，包含spring整合的配置 |
| spring-boot-starter-velocity | 对Velocity模板引擎的支持 |
| spring-boot-starter-web | 对Web项目开发的支持，包含Tomcat和spring-webmvc |
| spring-boot-starter-Tomcat | Spring Boot默认的Servlet容器Tomcat |
| spring-boot-starter | Spring Boot核心starter，包含自动配置、日志、yml配置文件的支持 |
| spring-boot-starter-actuator | 准生产特性，用来监控和管理应用 |
| spring-boot-starter-remote-shell | 提供基于ssh协议的监控和管理 |
| spring-boot-starter-amqp | 使用spring-rabbit来支持AMQP |
| spring-boot-starter-aop | 使用spring-aop和AsectJ支持面向切面编程 |
| spring-boot-starter-batch | 对Spring Batch的支持 |
| spring-boot-starter-cache | 对Spring Cache抽象的支持 |
| spring-boot-starter-cloud-connectors | 对云平台（Cloud Foundry、Heroku）提供的服务提供简化的链接方式 |
| spring-boot-starter-data-elasticsearch | 通过spring-data-elasticsearch对Elasticsearch的支持 |
| spring-boot-starter-data-gemfire | 通过spring-data-gemfire对分布式存储GemFire的支持 |
| spring-boot-starter-data-jpa | 对JPA的支持，包含spring-data-jpa、spring-orm和Hibernate |
| spring-boot-starter-data-mongodb | 通过spring-data-mongodb，对MongoDB的支持 |
| spring-boot-starter-data-rest | 通过spring-data-rest-webmvc将Spring Data repository暴露为REST形式的服务 |
| spring-boot-starter-data-solr | 通过spring-data-solr对Apache Solr数据检索平台的支持 |
| spring-boot-starter-freemarker | 对FreeMarker模板引擎的支持 |
| spring-boot-starter-groovy-templates | 对Groovy模板引擎的支持 |
| spring-boot-starter-Jetty | 使用Jetty作为Servlet容器替换Tomcat |
| spring-boot-starter-undertow | 使用Undertow作为Servlet容器替换Tomcat |
| spring-boot-starter-logging | Spring Boot默认的日子框架Logback |
| spring-boot-starter-log4j | 支持使用Log4j日志框架 |
| spring-boot-starter-websocket | 对WebSocket开发的支持 |
| spring-boot-starter-ws | 对Spring Web Services的支持 |
| 第三方的支持 | |
| Handlebars | https://github.com/allegro/handlebars-spring-boot-starter |



### 日志框架

#### LogBack

无论使用哪种框架，Spring Boot都为当前使用日子框架及文件输出做好了配置。而默认情况下，Spring Boot使用LogBack作为默认日志框架。

配置日志文件保存路径：

|  |
| --- |
| logging.file=c:/springboot.log |

配置日志文件，按照格式loggin.lever.包名=级别：

|  |
| --- |
| logging.level.org,springframework.web=DEBUG |

或者直接配置root所有日志级别：

|  |
| --- |
| logging.level.root=DEBUG |

#### Log4j

Parent指定到低版本，并排除starter-logging，添加log4j依赖配置

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <aritfactId>spring-boot-starter-parent</aritfactId>  <version>1.2.5.RELEASE</version>  </parent>  <dependency>  <groupId>org.springframework.boot</groupId>  <aritfactId>spring-boot-starter-starter</aritfactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <aritfactId>spring-boot-starter-starter-logging</aritfactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <aritfactId>spring-boot-starter-starter-log4j</aritfactId>  </dependency> |
| log4j.properties |
| log4J.rootLogger=info.ServerDailyRollingFile.stdout  log4J.appender.ServerDailyRollingFile=org.apache.log4j.DailyRolingFileAppender  log4J.appender.ServerDailyRollingFile.DatePattern=’.’yyyy-MM-dd  log4J.appender.ServerDailyRollingFile.File=C://SSMM/springboot\_test.log  log4J.appender.ServerDailyRollingFile.layout=org.apache.log4j.PatternLayout  log4J.appender.ServerDailyRollingFile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%t] %-5p [%c] - %m%n  log4J.appender.ServerDailyRollingFile.Append=true  log4J.appender.stdout=org.apache.log4j.ConsoleAppender  log4J.appender.stdout.layout=org.apache.log4j.PatternLayout  log4J.appender.stdout.layout.ConversionPattern=%d yyyy-MM-dd HH:mm:ss %p [%c] %m%n |

### Profile配置

Profile是Spring针对不同的环境提供不同的配置支持的一种功能，一般应用在生产环境和测试环境的配置区分上（比如，测试环境一般需要打印更多日志信息或者http响应中会由很多测试标示信息等等，生产环境则不会有）。

Spring Boot中通过类似格式application-{profile\_name}.properties的配置形式指定当前活动得Profile。

例如：首先在application.properties中配置如下：

|  |
| --- |
| spring.profiles.active=dev |

新建application-dev.properties和application-prod.properties两个配置文件，内容如下：

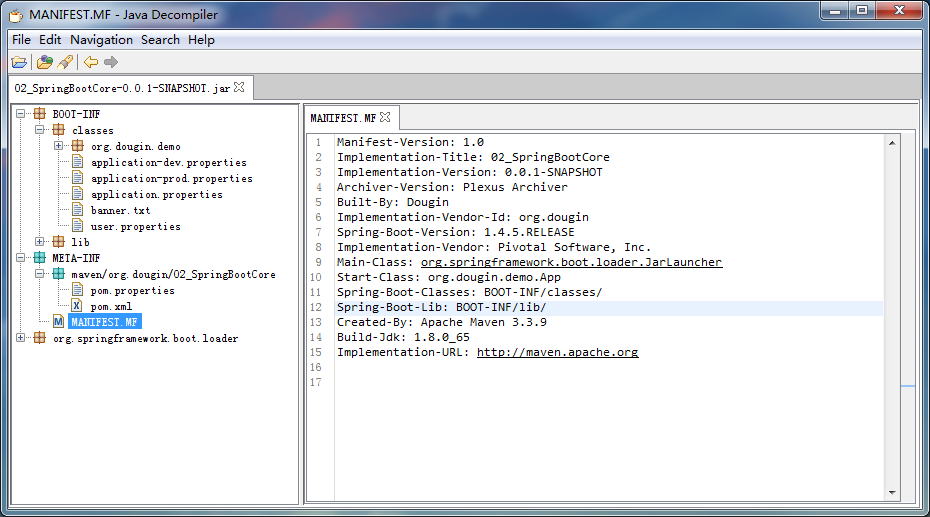
application-dev.properties:

|  |
| --- |
| server.port=8181 |

application-prod.properties:

|  |
| --- |
| server.port=8080 |

### 启动原理



1. Main-Class是org.springframework.boot.loader.JarLauncher，这个是jar启动的Main函数。还有一个Start-Class是org.dougin.demo.App，这个是我们应用自己的Main函数。
2. Archive的概念：archive即归档文件，这个概念在linux下比较常见，通常就是一个tar/zip格式的压缩包，jar是zip格式。
3. Spring Boot扩展了这个协议，让它支持多个‘ !/’，就可以表示jar in jar，jar in directory的资源了。

Spring的启动流程：

Spring Boot应用打包之后，生成一个jar，里面包含了应用依赖的jar包，还有Spring Boot Loader相关的类。jar的启动Main函数是JarLauncher，它负责创建一个LaunchedURLClassLoader来加载/lib下面的jar，并以一个新线程启动应用的Main函数。

### 运行原理

查看自动配置，通过日志跟踪：

1. 运行jar是增加 –debug参数
2. 在application.properties中设置debug=true
3. 在eclipse工具中启动右键 VM arguments: -debug

Spring Boot并没有任何新的技术，全都是基于Spring4.0提供的计算，用优秀的设计，为Web开发提供了一套新的方式。

在HelloWorld中，我们没有运行任何显示的配置，但是程序还是运行起来了，那么Spring Boot是怎么做到的呢？那就得从启动类说起。

## 热部署

首先确保eclipse打开了Project-Build Automatically，才能正常使用热部署。

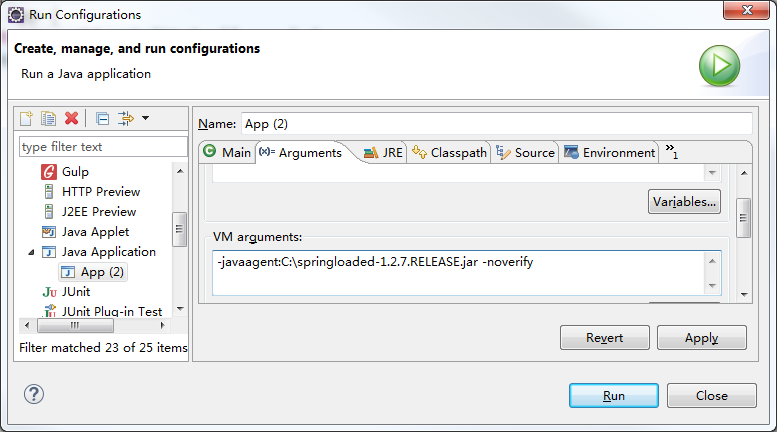
### SpringLoaded

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  <version>1.2.7.RELEASE</version>  </dependency>  </dependencies>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  <configuration>  <classifier>exec</classifier>  </configuration>  </execution>  </executions>  </plugin>  </plugins>  </build> |

手动结束java.exe线程

或把springloaded.jar包下载到本地；

在运行项目的时候，添加参数：-javaagent:C:\springloaded-1.2.7.RELEASE.jar -noverify



### Devtools

spring-boot-devtools是一个为开发者服务的一个模块，其中最重要的功能就是自动应用代码更改到最新的App上面去。原理是在发现代码有更改之后，重新启动应用，但是速度比手动停止后再启动还要更快，更快指的不是节省出来的手工操作的时间。

其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。

注意：

1. 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），因为其采用的虚拟机机制，该项重启是很快的。
2. devtools可以实现页面热部署（即页面修改后悔立即生效，这个可以直接在application.properties文件中配置）

|  |
| --- |
| spring.thymeleaf.cache=false  spring.freemarker.cache=false  spring.velocity.cache=false |

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <optional>true</optional>  <scope>true</scope>  </dependency>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <fork>true</fork>  </configuration>  </plugin>  </plugins>  </build> |

## Web开发

### JSON数据处理

默认jackjson解析:

1. 编写bean实体类；
2. 编写controller类，将bean对象返回页面；
3. 测试。

|  |
| --- |
| @RequestMapping("getUser.json")  **public** User getUser() {  User user = **new** User();  user.setId("10001");  user.setName("Dougin");  user.setAge(25);  user.setCreateDate(**new** Date());  **return** user;  } |
| {"id":"10001","name":"Dougin","age":25,"createDate":1493300739248} |

fastjson

1. 引入fastjson依赖；
2. 配置fastjson（支持两种方法）：
   1. 启动类继承extends WebMvcConfigurerAdapter；
   2. 启动类注入Bean：HttpMessageConverters；
3. 测试。

|  |
| --- |
| <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.30</version>  </dependency> |
| **public** **class** User {  …  @JSONField(format="yyyy-MM-dd HH:mm:ss")  **private** Date createDate;  …  } |
| 第一种方式：启动类注入Bean：HttpMessageConverters； |
| @SpringBootApplication  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }    @Bean  **public** HttpMessageConverters fastJsonConverters() {  FastJsonHttpMessageConverter fastConverter =  **new** FastJsonHttpMessageConverter();  FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  fastConverter.setFastJsonConfig(fastJsonConfig);  HttpMessageConverter<?> converter = fastConverter;  **return** **new** HttpMessageConverters(converter);  }  } |
| 第二种方式：启动类继承extends WebMvcConfigurerAdapter； |
| @SpringBootApplication  **public** **class** App **extends** WebMvcConfigurerAdapter {  **public** **void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  **super**.configureMessageConverters(converters);  FastJsonHttpMessageConverter fastConverter =  **new** FastJsonHttpMessageConverter();  FastJsonConfig fastJsonConfig = **new** FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  fastConverter.setFastJsonConfig(fastJsonConfig);  converters.add(fastConverter);  }  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |
| {  "age":25,  "createDate":"2017-04-27 22:02:01",  "id":"10001",  "name":"Dougin"  } |

### 使用thymeleaf

使用方法：

1. 在pom.xml中引入thymeleaf；
2. 关闭thymelea缓存；
3. 编写thyemelea模板文件（默认在src/main/resources/templates目录下）；
4. 编写模板请求controller。

|  |
| --- |
| <!-- thymelea默认是使用的2.0，如果要使用其他版本的话需要指定。-->  <properties>  <thymeleaf.version>3.0.0.RELEASE</thymeleaf.version>  <thymeleaf-layout-dialect.version>2.0.0</thymeleaf-layout-dialect.version>  </properties>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency> |
| spring.thymeleaf.prefix=classpath:/templates/  spring.thymeleaf.suffix=.html  spring.thymeleaf.mode=HTML5  spring.thymeleaf.encoding=UTF-8  spring.thymeleaf.content-type=text/html  spring.thymeleaf.cache=false |
| <!DOCTYPE html>  <html xmlns=*"http://www.w3.org/1999/xhtml"*  xmlns:th=*"http://www.thymeleaf.org"*  xmlns:sec=*"http://thymeleaf.org/thymeleaf-extras-springsecurity3"*>  <head>  <meta charset=*"UTF-8"*>  <title>Insert title here</title>  </head>  <body>  <h1 th:inline=*"text"*>Hello.v.1</h1>  <p th:text=*"${hello}"*></p>  </body>  </html> |
| @Controller  **public** **class** ThymeleafController {  @RequestMapping("/index.html")  **public** String indexHtml(Map<String, Object> map) {  map.put("hello", "welcome");  **return** "/NewFile";  }  } |

使用模板的时候Controller的注解就不能使用@RestController了，所以就单独使用@Controller就可以了。

### 使用freemarker

使用方法：

1. 在pom.xml中引入freemarker；
2. 关闭freemarker缓存；
3. 编写模板文件.ftl（默认在src/main/resources/templates目录下）；
4. 编写访问文件的controller。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-freemarker</artifactId>  </dependency> |
| spring.freemarker.allow-request-override=false  spring.freemarker.cache=false  spring.freemarker.check-template-location=true  spring.freemarker.charset=UTF-8  spring.freemarker.content-type=text/html  spring.freemarker.expose-request-attributes=false  spring.freemarker.expose-session-attributes=false  #spring.freemarker.expose-request-context-attributes=  spring.freemarker.expose-spring-macro-helpers=false  #spring.freemarker.prefix=  #spring.freemarker.suffix=.ftl  #spring.freemarker.template-loader-path=classpath:/templates/#comma-separated list  #spring.freemarker.view-names=# whitelist of view names that can be resolved |
| <!DOCTYPE html>  <html>  <head>  <meta charset="UTF-8">  <title>Insert title here</title>  </head>  <body>  <h1>Index By Freemarker</h1>  <p>${hello}</p>  </body>  </html> |
| @Controller  **public** **class** FreemarkerController {  @RequestMapping("/index")  **public** String index(Map<String, Object> map) {  map.put("hello", "Thi is freemarker page.");  **return** "index";  }  } |

### 使用JSP

1. 创建maven web project
2. 在pom.xml文件配置依赖
3. 配置application.properties支持jsp
4. 编写测试controller
5. 编写JSP页面

Spring Boot官方其实并不推荐使用JSP作为展示页面，STS创建的项目会在src/main/resources下有个templates目录，这里就是让我们放模板文件的，然后并没有生成诸如SpringMVC中的webapp目录。

因此使用eclipse新建一个Maven Web Project项目。

|  |
| --- |
| 配置application.properties |
| spring.mvc.view.prefix=/WEB-INF/jsp/  spring.mvc.view.suffix=.jsp |

### web相关的配置

* Servlet:引用HttpServlet接口，采用原生的Servlet进行请求响应；
* Listener:引用ServletContextListener，常用于Web缓存；
* Filter:引用Filter接口，常用于认证、日志、令牌等。

方案一：采用原生Servlet 3.0的注解进行配置，@WebServlet、@WebListener、@WebFilter是Servlet 3.0 api中提供的注解，通过注解可以完全代替web.xml中的配置。

方案二：是采用自己SpringBoot配置bean的方式进行配置的，SpringBoot提供了三种bean：FilterRegistrationBean、ServletRegistrationBean、ServletListenerRegistrationBean分别对应配置原生的Filter、Servlet、Listener，这三个配置和方案一采用的方式能够达到统一的效果。

|  |
| --- |
|  |

### 文件上传

|  |
| --- |
|  |

### 邮件发送

|  |
| --- |
|  |

### tomcat配置

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# 总结

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |