# Springboot

## SpringBoot入门

### springboot简介

长期以来，java的开发一直让人所诟病：

* Java项目开发复杂度极高
* Java项目维护困难
* 在云时代如何实现项目的快速部署以及快速启动
* 即便使用了大量的开发框架，发现我们的开发也没有少多少
* 当所有人认为spring不再前进的时候，spring推出了微架构实现的两个重要开发框架：SpringBoot、SpingCloud。

1. java开发复杂度是最高的？

* Java里面提供的开发支持都属于原生操作的代码，例如：JDBC为例，如果使用java的原生代码，会重复编写大量的内容。例如：prepareStatement操作。
* Java进行Web项目开发的时候，必需要求按照严格的格式进行WEB项目的创建，以及每当修改web程序的时候，有需要进行tomcat的重新启动；
* Java中虽然提供了开发标准，但是所有的公司几乎都有可能有自己的标准，比如最初时代，jvm的标准就有三个，而且许多公司，由于版本的不同，会造成部署环境的不同；
* Java严格要求使用mvc设计模式完成
* 以web开发为例，jsp程序里面不应该包含有任何的Script程序代码。但是要做到这步会非常麻烦，有各种的实现标准，例如：JSTL+EL、SpringTaglib、StrutsTaglib、JSF
* 如果现在使用Node.js、Python开发一个控制器程序类可能只需要几行代码，而java

需要写一堆的代码，而且要部署。

1. java后期发展使用了大量的Maven技术作为开发，会发现传统的开发处理中并没有逃离掉web的身影，所有的项目需要打包成war文件，而后上传到系统中。使用maven最大的痛是：如果是开发框架，那一堆的maven配置依赖库。
2. Rest技术开始在行业中广为流传，而java要想实现Rest架构的开发（基于：Spring），那么也是相当麻烦。
3. 现在行业中，spring已经作为绝对的java架构，但是如果要想在Spring中整合RabbitMQ、KafKa、ActiveMQ、Druid、Redis、Shiro,需要编写一堆的.xml配置文件。

所以在这样的历史背景下，很多人开始寻求更加简便的开发，而遗憾的是这种简便的开发没有被JDK所支持、被JavaEE所支持，因为这些知识平台，平台能够提供的只是最原始的技术支持。这一时刻终于由于Spring框架的升级而得到了提升，SpringBoot的出现，改变了java开发的困境，SpringBoot的最终奉行的宗旨是：废除掉所有复杂的开发，废除掉所有的配置文件。让开发变得更简单纯粹。

### Springboot快速入门

1. 如果要想开发SpringBoot程序，只需要按照管方的要求配置一个父pom即可。

<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>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<groupId>cn.mldn</groupId>

<artifactId>bootfirst</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>bootfirst</name>

<url>http://maven.apache.org</url>

<properties>

<jdk.version>1.8</jdk.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<finalName>bootfirst</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>${jdk.version}</source><!-- 源代码使用的开发版本 -->

<target>${jdk.version}</target><!-- 需要生成的目标class文件的编译版本 -->

<encode>${project.build.sourceEncoding}</encode>

</configuration>

</plugin>

</plugins>

</build>

</project>

1. 编写一个具体的程序：

@Controller  
@EnableAutoConfiguration  
**public class** SampleController {  
 @RequestMapping(**"/"**)  
 @ResponseBody  
 **public** String home(){  
  
 **return "杨幂"**;  
 }  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(SampleController.**class**, args);  
 }  
}

直接运行java程序“Spring Boot APP”。

如果是一个Maven的普通项目，最简单的做法是在maven运行输入“Spring-boot:run”。

## SpringBoot基础知识

### 建立统一父pom

1. 建立一个microboot的maven项目

*<?***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.cn.ym.com</**groupId**>  
 <**artifactId**>microboot</**artifactId**>  
 <**packaging**>pom</**packaging**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**properties**>  
 <**java.version**>1.8</**java.version**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-dependencies</**artifactId**>  
 <**version**>2.0.2.RELEASE</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
 </**dependencyManagement**>  
 <**build**>  
 <**finalName**>microboot</**finalName**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**configuration**>  
 <**source**>${java.version}</**source**>*<!-- 源代码使用的开发版本 -->* <**target**>${java.version}</**target**>*<!-- 需要生成的目标class文件的编译版本 -->* <**encoding**>${project.build.sourceEncoding}</**encoding**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
 </**build**>  
 <**modules**>  
 <**module**>microboot-base</**module**>  
 <**module**>microboot-base</**module**>  
 </**modules**>  
</**project**>

1. 建立microboot-base的子模块，实现之前同样的操作功能

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microboot</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microboot-base</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

### SpringBoot代码测试

1、【修改microboot-base模块】修改pom.xml配置文件，追加SpringBoot测试支持类

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

2、【修改microboot-base模块】建立一个测试程序类

@SpringBootTest(classes = SampleController.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestSampleController {  
 @Autowired  
 **private** SampleController **sampleController**;  
  
 @Test  
 **public void** home() {  
 TestCase.*assertEquals*(**this**.**sampleController**.home(), **"www.ym.cn"**);  
 }  
}

### SpringBoot启动注解分析

|  |  |  |
| --- | --- | --- |
| No | 注解 | 说明 |
| 1 | @Controller | 进行控制器的配置注解，这个注解所在的类就是控制器类 |
| 2 | @EnableAutoConfiguration | 开启自动配置处理 |
| 3 | @RequestMapping("/") | 表示访问路径，此时的路径为“/”,访问地址：“http://localhost:8080/” |
| 4 | @ResponseBody | 在Restful架构中，该注解表示直接将返回的数据以字符串或json的形式获得 |

在给出的几个直接中，“EnableAutoConfiguration”为整个SpringBoot的启动注解配置，也就是说这个注解应该随着程序的主类一起定义。

而对于控制器程序类，由于项目中会有许多的控制器，那么最好将这些类统一保存在一个包中。

下面将所有的程序类保存在“xxx.comtroller”中，是”xxx”的子包，强烈建议（Spring官方建议）：如果要想进行简单方便的开发，所有的程序类一定要在启动类所在包的子包下。

1. 【microboot-base】模块建立一个HelloController程序类：

@Controller

public class HelloController {

@RequestMapping("/")

@ResponseBody

public String home() {

return "www.mldn.cn";

}

}

1. 【microboot-base】模块启动程序主类：

@SpringBootApplication // 启动SpringBoot程序，而后自带子包扫描

public class SampleController {

public static void main(String[] args) throws Exception {

SpringApplication.run(SampleController.class, args);

}

}

1. 【microboot-base】以上的做法只是传统程序的开发做法，因为现在为止毕竟是两个程序类，其彼此之间的联系需要有一个连接点，程序中配置的“@ComponentScan”注解就是负责这个连接处理。但是SpringBoot考虑到了此类的配置问题，提出了一个更简化策略，该策略核心思想：既然程序主类会在所有开发包的父包里面，那么能不能简化点取得配置呢？为此在实际开发中，会使用一个特殊的符合注解：@SpringBootApplication（启动SpringBoot程序，而后自带**子包**扫描）。

如果开发的程序类和启动主类不在一个子包下，那么就启动类必须加上“@ComponentScan("cn.ym.com")”。

@SpringBootApplication=@EnableAutoConfiguration+@ComponentScan+其他配置。正式因为它有这样的特点，所以当以后使用Bean实现配置处理的时候将会非常的容易。

注意：程序类和启动类在一个包下不行，必须是在启动类所在的子包下：

### 路径访问控制

在实际开发中，控制器的路径可能会有许多个，而且在进行控制器编写的时候也会有两种运行模式：跳转配置、Restful显示。那么下面来观察关于路径的详细描述。

1. 在之前所编写的控制器里面有如下两种注解配置使用：

* @Controller在类上定义表示的是一个控制器；
* @ResponseBody：将控制器中方法的返回值变为rest内容。如果不加这个注解，表示发挥到“”内的页面。

@RequestMapping(**"/"**)  
**public** String home() {  
 **return "www.baidu.cn"**;  
}

访问后跳转项目中“www.baidu.com”这个页面，但是项目中是没有这个页面。

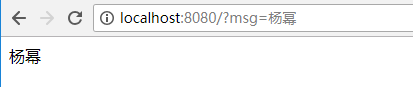
但是一个项目里面可能控制器之中返回的全部是是restful信息，这样分别定义就态麻烦了，为此在SpringBoot里面又提供一个符合注解：@RestController。

@RestController  
**public class** HelloController {  
 @RequestMapping(**"/"**)  
 **public** String home() {  
 **return "www.baidu.cn"**;  
 }  
}

从MVC的实际标准来讲，控制器需要传递一些属性到页面显示，按照这样的原则并不是所有的开发都会以rest的结构返回，但是rest结构是SpringCloud的实现核心技术。

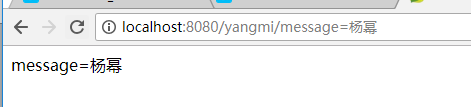
2、现在所给出的控制器类里面只是负责了简单的信息返回，那么实际上也可以进行参数的接收处理，传递参数到控制器中最简单的做法是使用地址重写传递“xx?参数名称=内容”。

@ResponseBody  
@SpringBootApplication  
**public class** SampleController {  
 **public static void** main(String[] args) **throws** Exception {  
 SpringApplication.*run*(SampleController.**class**, args);  
 }  
}



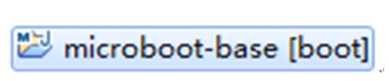
3、由于Springboot支持rest分割处理，所以此时对于参数的接收可以采用路径参数的形式完成。

@ResponseBody  
@SpringBootApplication  
**public class** SampleController {  
 **public static void** main(String[] args) **throws** Exception {  
 SpringApplication.*run*(SampleController.**class**, args);  
 }  
}



4、在springboot里面，可以追加自动加载配置的依赖。

* 如果项目只是一个简单的springBoot的配置项目，则显示的项目信息如下：



* 【mircoo-base模块】如果现在添加有如下的两个依赖库：

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>



这就表示该项目重新启动后以后可以自动加载。

### 使用内置对象

在springboot中控制器的形式和SpringMVC是一样的，所以要想使用jsp内置对象，也可以按照springmvc的方式进行。

@RequestMapping("/object")

public String object(HttpServletRequest request,HttpServletResponse response) {

System.out.println("\*\*\* 客户端IP地址：" + request.getRemoteAddr());

System.out.println("\*\*\* 取得客户端响应编码：" + response.getCharacterEncoding());

System.out.println("\*\*\* 取得SessionID：" + request.getSession().getId());

System.out.println("\*\*\* 取得真实路径：" + request.getServletContext().getRealPath("/upload/"));

return "www.mldn.cn" ;

}

### 项目打包

在项目中配置好插件，以及打包就可以执行了，并且这个执行不需要特别复杂的配置。1、【microo项目】修改Pom.xml,追加新的插件：

<plugin> <!-- 该插件的主要功能是进行项目的打包发布处理 -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration> <!-- 设置程序执行的主类 -->

<mainClass>cn.mldn.microboot.StartSpringBootMain</mainClass>

</configuration>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

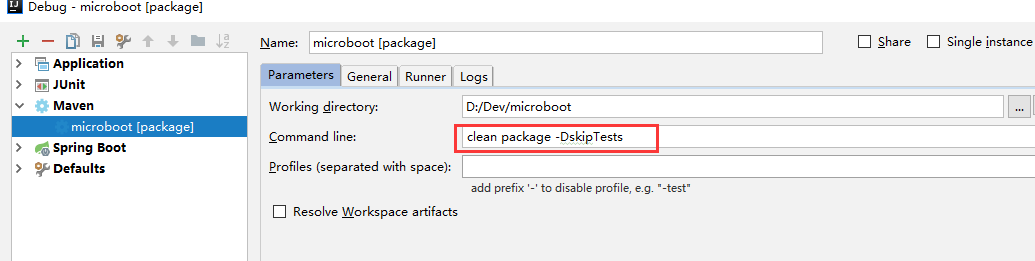
</executions>

</plugin>

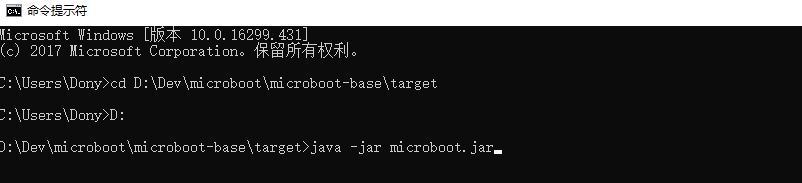
</plugins>

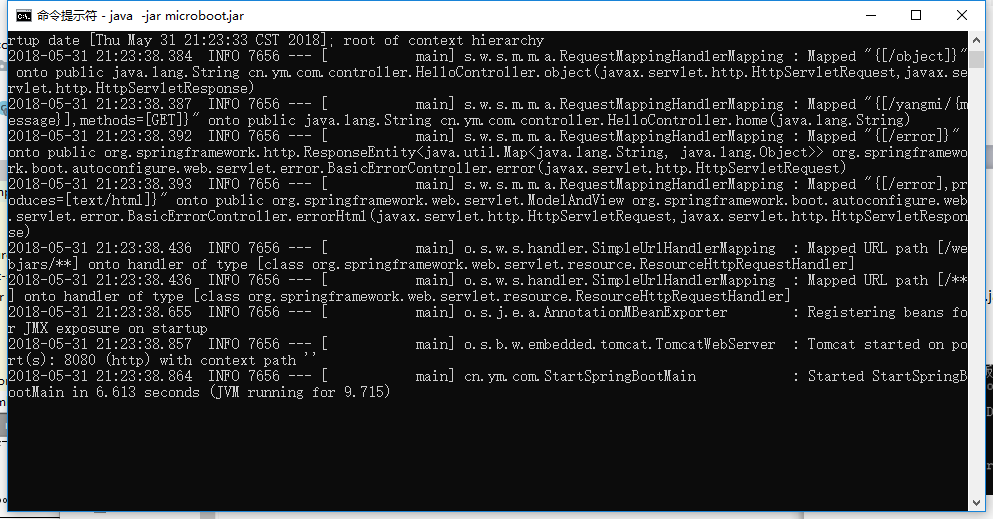
这样每一个模块都可以使用这个配置，如果每个模块不一样则单独配置即可

1. 【microboot-base模块】将当前的项目模块打包处理：clean package;



1. 将“microbbot.jar”文件随意拷贝到一个路径中，在命令行下执行：java -jar xxx.jar（linux环境同理）。





## SpringBoot 配置深入

### 配置环境属性

在java的MVC项目里面，所有的项目都一定需要满足以下几点要求：

* 访问的端口不能是8080，应该使用默认的80端口；
* 在项目中为了方便进行数据的维护，建议建立一序列的\*.properties配置文件
* 所有的控制器都采用rest风格的输出，但是正常来讲，信息的显示应该交给页面（不再是jsp）负责完成；
* 项目打包的时候应该考虑到不同的profile配置。

在默认情况下，在SpringBoot里面启动WEB容器为Tomcat,项目部署时如果单独运行。不可能运行在8080端口上，从正常来讲应该是运行在80端口上。所以要想修改这样的默认环境，则必须编写与之对应的配置文件，改配置文件必须卸载类路径下src/main/resources，可以在该目录下创建application.properties文件（文件名称不要改变）。

#设置tomcat的运行服务所在的端口

server.port=80

#可以配置Contextpath访问路径，但是实际开发中是不能够进行配置的。

Server.context-path=/hello

此时的访问路径要追加ContextPath前缀：http://localhost/hello/

在springboot、springcloud里面可以使用两类配置文件：application.properties、application.yml

yml文件：这是一种结构化的数据文件，在好多地方都是用过，例如：apacheStorm开发组件上进行配置时候使用的就是yml文件，该配置文件的全称（Yet Another Markup Languange、是一种标记语言）。

server:

port:80 #设置服务访问端口

如果同名的properties文件和yml文件同时存在，那么会有限进行proterties配置文件的加载。如果两个配置文件的内容冲突则以properties内容为主。

默认情况下，SpringBoot使用的是tomcat容器，如果需要也可以将其更换成jetty容器。

追加一些依赖（排除tomcat依赖）即可使用这个容器：

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

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

### 读取资源文件

在springboot里面进行资源文件的配置只需要做一些简单的application.yml即可，而且所有注入的资源文件都可以想spring那样直接使用MessageSource进行读取，资源文件一定是properties。

1. 为统一管理资源文件在“src/main/resources”目录下建立有一个i18n的存储目录：
2. 建立两个资源文件：

Message.properties、Page.pproperties

1. 在application.yml中引用资源文件：

**spring:** *#表示该配置交给spring负责* **messages:** *#表示进行资源配置* **basename:** i18n/Message,i18n/Pages *#资源文件名称*

1. 当执行完以上配置之后会自动为用户创建MessageSource对象，在使用的时候直接注入此对象即可。

* 考虑到实际开发的标准，所以现在创建一个父的控制器的抽象类：AbstractController,在此抽象类中进行资源文件的注入配置；

**public class** AbstractController {  
 @Autowired  
 **private** MessageSource **messageSource**;  
  
 **public** String getMessage(String key, String... args) {  
 **return this**.**messageSource**.getMessage(key, args, Locale.*getDefault*());  
 }  
}

1. 在控制器的子类中读取以上的配置信息：

@RestController  
**public class** MessageController **extends** AbstractController {  
 @RequestMapping(**"/message"**)  
 **public** String message() {  
 **return super**.getMessage(**"yangmi"**);  
 }  
}

### Bean配置

在Spring进行开发配置的时候有两类选择：\*.ml配置文件、配置的bean（@Configure）,在Springboot中如果真的需要\*.xml文件编写，但是又不想出现配置文件的话，这个时候最简单的做法就是使用bean的方式来进行类的配置。

前提：该配置程序的bean所在的包必须是程序启动类所在包的子包之中，这样才可以自动扫描到。

1、传统的spring：

@Service  
**public class** MessageServiceImpl **implements** IMessageService {  
 @Override  
 **public void** sayMessage() {  
 System.***out***.println(**"杨幂，我爱你啊，2018"**);  
 }  
}

2、如果不使用注解@services等对象将无法注入，在启动类所在的子包中建立一个配置程序类：ServiceConfig（名字可以随便写）。

@Configuration  
*//类名称可以随便写***public class** MessageServiceConfig {  
  
 @Bean *// 此处返回的是一个Spring的配置Bean，与xml的“<bean>”等价  
 //方法名称随便写* **public** MessageServiceImpl getMessageService() {  
  
 **return new** MessageServiceImpl();  
 }  
}

此时采用了自动扫描Bean的模式来进行了相关对象的配置。

3、SSM或SSH开发框架出现时间比较长，已经有了一个非常完善的xml配置，那么SpringBoot也支持配置文件的读取。

*<?***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="messageService" class="cn.ym.com.service.MessageServiceImpl"** />  
</**beans**>

随后在程序启动类上使用xml进行配置加载：

@ResponseBody  
**@ImportResource(locations = "classpath:spring-common.xml")**  
@SpringBootApplication  
**public class** StartSpringBootMain {  
 **public static void** main(String[] args) **throws** Exception {  
 SpringApplication.*run*(StartSpringBootMain.**class**, args);  
 }  
}

@Configuration  
*//类名称可以随便写***public class** MessageServiceConfig {  
  
 @Bean(name = **"configService"**) *// 此处返回的是一个Spring的配置Bean，与xml的“<bean>”等价  
 //方法名称随便写* **public** MessageServiceImpl getMessageService() {  
  
 **return new** MessageServiceImpl();  
 }  
}

如果此时所配置的两个bean都没有名字那么在进行注入的时候一定出现重复的错误，而这个错误新版本中已经完善，不过开发中要准确的指定对象，则需要使用名字来完成：

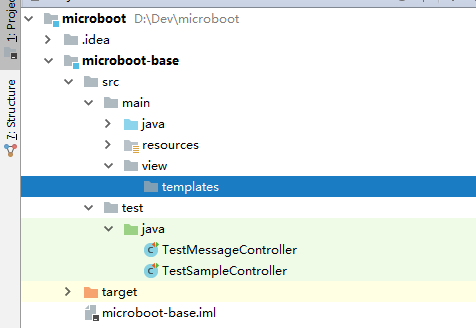
### 模板渲染

在之前所看到的信息显示发现都是以Res风格显示，但是很明显在实际的开发中，所有数据的显示最终都应该交由页面完成，但是这个页面并不是\*.jsp页面，而是普通的\*.html页面，而且最为重要的是，此处所使用的渲染的页面采用的使用模板的方式显示，而在java开发行业，对于前台的显示模板常见的一共有三类技术：FreeMarker、Velocity、thymeleaf(推荐使用)。

1. 如果要想在项目中使用thymeleaf模板，那么应该首先进行相关支持依赖库的导入：
2. 本次的开发依然通过控制层跳转到页面之中进行显示。在springMVC的时代使用的是ModelAndView传递信息，而在SpingBoot中则可以直接在方法中定义一个Model参数即可。此时要将@RestController换成@Controller。

现在控制器使用的是“@Controller”注解，所以此时执行该控制器后会跳转。要跳转页面定义有严格要求：在CLASSPATH路径下：（src/main/resources、src/main/view）必须建立一个templates(名字不能有错)的目录，在此目录下保存有thymeleaf所有相关页面。这些页面可以按照文件目录保存；

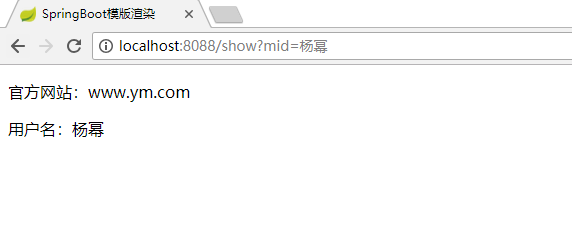
* 建立一个源代码目录：src/main/view



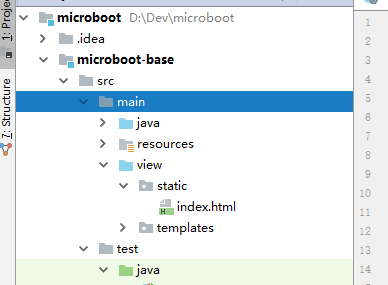
1. 编写控制器：  
    @RequestMapping(**"/show"**)  
    **public** String show(String mid, Model model){  
    model.addAttribute(**"url"**,**"www.ym.com"**);  
    model.addAttribute(**"mid"**,mid);  
    **return "message/message\_show"**;  
    }  
   }
2. 编写message\_show.html页面（编写时所有的元素一定要完结）

<!DOCTYPE **HTML**>  
<**html xmlns:th="http://www.thymeleaf.org"**>  
<**head**>  
 <**title**>SpringBoot模版渲染</**title**>  
 <**meta http-equiv="Content-Type" content="text/html;charset=UTF-8"**/>  
</**head**>  
<**body**>  
<**p th:text="'官方网站：' + ${url}"**/>  
<**p th:text="'用户名：' + ${mid}"**/>  
</**body**>  
</**html**>

1. 启动springboot，访问页面：



5、如果定义的页面访问不是通过控制器跳转的，可以考虑在thymeleaf所在路径中建立一个static的子目录，该目录保存的是所有的静态页面；在以后开发中，静态资源文件要求放在static目录中。



6、thymeleaf默认访问的页面路径的后缀为\*.html，那么通过修改application.ynl配置文件进行变更：



### Profile配置

由于开发（dev）、测试（beta）、运行（produc）的环境不同，有可能需要选择不同的配置文件，所以springboot提供多个profile配置，对于多profile的配置一定要区分出是yml还是properties.

1. 基于yml实现profile的配置处理：

在使用yml配置文件的时候所有可以使用的profile配资项都要求在一个文件中编写：

**spring:  
 profiles:  
 active:** dev  
---  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 profiles:dev  
**server:  
 port:** 8080  
---  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **profiles:** beta  
**server:  
 port:** 8088  
---  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **profiles:** product  
**server:  
 port:** 80

最初进行profile切换的处理可以通过maven的编译工具动态选择，但是在spring里面可以方便的实现这种切换。

1. 此时所有的thymeleaf页面都保存在“src/main/view”目录中，而且所有的springboot的配置文件（\*.yml）也应该一起输出到jar文件里面，那么就需要一个资源文件引用，修改“microboot”中的pom.xml中的配置文件：

<**resources**>  
 <**resource**>  
 <**directory**>src/main/resources</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.properties</**include**>  
 <**include**>\*\*/\*.yml</**include**>  
 <**include**>\*\*/\*.xml</**include**>  
 <**include**>\*\*/\*.tld</**include**>  
 </**includes**>  
 <**filtering**>false</**filtering**>  
 </**resource**>  
 <**resource**>  
 <**directory**>src/main/java</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.properties</**include**>  
 <**include**>\*\*/\*.xml</**include**>  
 <**include**>\*\*/\*.tld</**include**>  
 </**includes**>  
 <**filtering**>false</**filtering**>  
 </**resource**>  
 <**resource**>  
 <**directory**>src/main/view</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.\*</**include**>  
 </**includes**>  
 <**filtering**>false</**filtering**>  
 </**resource**>  
</**resources**>

1. 随后可以将项目进行打包处理，在打包之后也可以动态切换profile配置。

|-maven打包：clean package(clean package -DskipTests);

|-为了实现动态切换profile,可以在程序执行的时候设置一个执行的处理参数：

java -jar microboot.jar --spring.profiles.active=beta

1. 现在运行完成之后发现程序中国默认采用的图标是spring的，可以替换成自己的图标，可以单独准备出一个ico配置文件，可以将这个配置文件拷贝到“src/main/view/static/images”目录之中，随后在页面中进行引用：

<link rel=”icon” type=”image/x-icon” href=”/images/xxx.ico”/>

1. 在进行profile配置的时候，如果使用的是application.properties配置，这样的配置处理过程是不一样的，此时就需要编写多个属性文件内容：

|  |  |  |
| --- | --- | --- |
| 【开发环境】application-dev.properties | 【测试】application-beta.properties | 【生产】application-product.properties |
| **spring.messages.basename**=**i18n/Messages,i18n/Pages server.port**=**8080** | **spring.messages.basename**=**i18n/Messages,i18n/Pages server.port**=**8088** | **spring.messages.basename**=**i18n/Messages,i18n/Pages server.port**=**80** |

1. 随后还是需要有一个公共的application.properties配置文件出现，用于指派可以使用的profile配置。

**spring.profiles.active**=**product**

在执行jar包的时候，也可以执行“--spring.profile.active=beta”动态指派不同的profile配置

## springboot错误处理

Springboot针对于错误处理，一共提供有三种方式：数据验证错误、错误页的指派以及全局异常的处理。

### 数据验证错误处理

1. 建立一个VO类

public class Member implements Serializable {

private String mid ;

private Integer age ;

private Double salary ;

private Date birthday ;

public String getMid() {

return mid;

}

public void setMid(String mid) {

this.mid = mid;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

public Double getSalary() {

return salary;

}

public void setSalary(Double salary) {

this.salary = salary;

}

public Date getBirthday() {

return birthday;

}

public void setBirthday(Date birthday) {

this.birthday = birthday;

}

@Override

public String toString() {

return "Member [mid=" + mid + ", age=" + age + ", salary=" + salary

+ ", birthday=" + birthday + "]";

}

}

1. 建立控制器程序类

@Controller  
**public class** MemberController **extends** AbstractController {  
 @RequestMapping(value = **"/addPre"**, method = RequestMethod.***GET***)  
 **public** String addPre() { *// 增加前的准备操作路径* **return "member\_add"**;  
 }  
  
 @RequestMapping(value = **"/add"**, method = RequestMethod.***POST***)  
 @ResponseBody  
 **public** Object add(Member vo) { *// 增加前的准备操作路径* **return** vo;  
 }  
}

1. 由于此时程序中需要进行日期的转换处理操作，需要做一个转换处理的格式配置，修改AbstractController追加如下配置：

@InitBinder  
**public void** InitBinder(WebDataBinder binder) {  
 SimpleDateFormat df = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  
 *// 明确的描述此时需要注册一个日期格式的转化处理程序类* binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(df, **true**));  
}

1. 编写一个页面进行表单填写：

<!DOCTYPE HTML>

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

<head>

<title>SpringBoot模版渲染</title>

<link rel="icon" type="image/x-icon" href="/images/mldn.ico"/>

<meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>

</head>

<body>

<form action="add" method="post">

用户邮箱：<input type="text" name="mid" value="mldnjava@163.com"/><br/>

用户年龄：<input type="text" name="age" value="16"/><br/>

用户工资：<input type="text" name="salary" value="92389239.23"/><br/>

用户生日：<input type="text" name="birthday" value="2010-10-10"/><br/>

<input type="submit" value="提交"/>

<input type="reset" value="重置"/>

</form>

</body>

</html>

1. 数据验证处理（null以及邮箱验证），springboot里面提供有默认支持，只不过这种支持未必是最好的，可以使使用hibernate-validation.jar工具包，有hibernate开发框架提供。



1. 要想进行验证，那么首先要解决的问题就是错误提示信息的问题，而在springboot里面对于错误信息的保存，都要求保存在ValidationMessages.properties文件，在“/src/main/resources”目录中建立此文件；

**member.mid.notnull.error**=**用户名不允许为空！  
member.mid.email.error**=**用户名的注册必须输入正确的邮箱！  
member.mid.length.error**=**用户名的格式错误！  
member.age.notnull.error**=**年龄不允许为空！  
member.age.digits.error**=**年龄必须输入合法数字！  
member.salary.notnull.error**=**工资不允许为空！  
member.salary.digits.error**=**工资必须是数字！  
member.birthday.notnull.error**=**生日不允许为空！**

提示：一个表单就有这么多的配置项，显然不太合理，所以最好的处理方式还是拦截器。

1. 修改Member.java程序类，追加验证处理方式：

@NotNull(message="{member.mid.notnull.error}")

@Email(message="{member.mid.email.error}")

@Length(min=6,message="{member.mid.length.error}")

private String mid ;

@NotNull(message="{member.age.notnull.error}")

@Digits(integer=3,fraction=0,message="{member.age.digits.error}")

private Integer age ;

@NotNull(message="{member.salary.notnull.error}")

@Digits(integer=20,fraction=2,message="{member.salary.digits.error}")

private Double salary ;

@NotNull(message="{member.birthday.notnull.error}")

private Date birthday ;

1. 修改控制器中的add方法，观察错误信息的显示：

@RequestMapping(value = "/add", method = RequestMethod.POST)

@ResponseBody

public Object add(@Valid Member vo, BindingResult result) { // 增加前的准备操作路径

if (result.hasErrors()) { // 现在表示执行的验证出现错误

Iterator<ObjectError> iterator = result.getAllErrors().iterator(); // 获取全部错误信息

while (iterator.hasNext()) {

ObjectError error = iterator.next() ; // 取出每一个错误

System.out.println("【错误信息】code = " + error.getCode() + "，message = " + error.getDefaultMessage());

}

return result.getAllErrors() ;

} else {

return vo;

}

}

### 配置错误页

1. 所有的错误页都是普通的静态文件，在“src/main/view/static”创建创建错误页（HTTP返回编码：404、500）
2. 添加一个错误页的配置类（子包中），在启动类中编写一个错误页面的配置项：

@Configuration  
**public class** ErrorPageConfig {  
 @Bean  
 **public** EmbeddedServletContainerCustomizer containerCustomizer() {  
 **return new** EmbeddedServletContainerCustomizer() {  
 @Override  
 **public void** customize(ConfigurableEmbeddedServletContainer configurableEmbeddedServletContainer) {  
 ErrorPage errorPage400 = **new** ErrorPage(HttpStatus.***BAD\_REQUEST***, **"/error-400.html"**);  
 ErrorPage errorPage404 = **new** ErrorPage(HttpStatus.***NOT\_FOUND***, **"/404.html"**);  
 ErrorPage errorPage500 = **new** ErrorPage(HttpStatus.***INTERNAL\_SERVER\_ERROR***, **"/error-500.html"**);  
 configurableEmbeddedServletContainer.addErrorPages(errorPage400, errorPage404,  
 errorPage500);  
 }  
 };  
 }  
}

### 全局异常处理

1. 如果定义了500处理页面，在发生异常是会跳转到500页面，如果去掉500配置，那么将会跳转到spring默认的空白页：

这个时候可以单独定义一个页面进行错误的信息显示处理，这个页面可以定义在“src/main/view/templates/error.html”。

1. 定义一个全局的异常处理类：

@ControllerAdvice *//作为一个控制层的切面处理***public class** GlobalExceptionHandler {  
 **public static final** String ***DEFAULT\_ERROR\_VIEW*** = **"error"**; *// 定义错误显示页，error.html* @ExceptionHandler(Exception.**class**)*// 所有的异常都是Exception子类* **public** ModelAndView defaultErrorHandler(HttpServletRequest request, Exception e) {  
 ModelAndView mav = **new** ModelAndView(***DEFAULT\_ERROR\_VIEW***); *// 设置跳转路径* mav.addObject(**"exception"**, e); *// 将异常对象传递过去* mav.addObject(**"url"**, request.getRequestURL()); *// 获得请求的路径* **return** mav;  
 }  
}

1. 定义error页面

<!DOCTYPE **HTML**>  
<**html xmlns:th="http://www.thymeleaf.org"**>  
<**head**>  
 <**title**>SpringBoot模版渲染</**title**>  
 <**link rel="icon" type="image/x-icon" href="/images/mldn.ico"**/>  
 <**meta http-equiv="Content-Type" content="text/html;charset=UTF-8"**/>  
</**head**>  
<**body**>  
 <**p th:text="${url}"**/>  
 <**p th:text="${exception.message}"**/>  
</**body**>  
</**html**>

对于全局异常信息显示除了采用以上的跳转方式，也可以使用rest进行显示。

1. 修改全局的异常处理类：

@RestControllerAdvice *//作为一个控制层的切面处理***public class** GlobalExceptionHandler {  
 **public static final** String ***DEFAULT\_ERROR\_VIEW*** = **"error"**; *// 定义错误显示页，error.html* @ExceptionHandler(Exception.**class**) *// 所有的异常都是Exception子类* **public** Object defaultErrorHandler(HttpServletRequest request, Exception e) {  
 **class** ErrorInfo {  
 **private** Integer **code**;  
 **private** String **message**;  
 **private** String **url**;  
  
 **public** Integer getCode() {  
 **return code**;  
 }  
  
 **public void** setCode(Integer code) {  
 **this**.**code** = code;  
 }  
  
 **public** String getMessage() {  
 **return message**;  
 }  
  
 **public void** setMessage(String message) {  
 **this**.**message** = message;  
 }  
  
 **public** String getUrl() {  
 **return url**;  
 }  
  
 **public void** setUrl(String url) {  
 **this**.**url** = url;  
 }  
 }  
 ErrorInfo info = **new** ErrorInfo();  
 info.setCode(100); *// 标记一个错误信息类型* info.setMessage(e.getMessage());  
 info.setUrl(request.getRequestURL().toString());  
 **return** info;  
 }  
}

## springboot整合tomcat

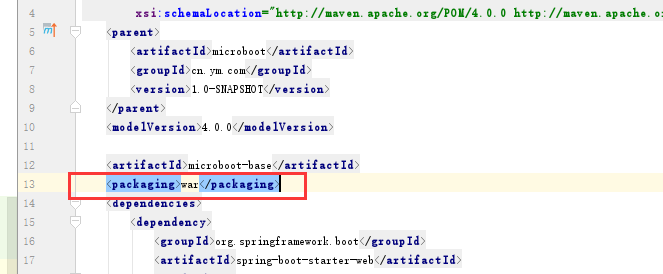
Springboot本身支持两类web容器：默认tomcat、jetty

### spring配置tomcat运行

如果一个程序项目需要发布到tomcat上运行，那么该项目一定要设置为war文件，修改项目的类型为war。

1. 修改pom.xml文件

设置打包类型为war类型



项目最终需要进行打包操作，所以还需设置一个maven的war的打包插件

<**build**>  
 <**finalName**>microboot-base</**finalName**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-war-plugin</**artifactId**>  
 <**configuration**>  
 <**warName**>yangmi</**warName**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

2、由于项目打包为war文件后需要通过tomcat执行，所以注销掉jetty相关配置。

3、更新完maven项目后，会提示缺少web.xml配置文件，可以通过tomcat的安装目录拷贝此文件。

1. 对spring的启动类做配置处理

public class StartSpringBootMain extends SpringBootServletInitializer { // 必须继承指定的父类

@Override

protected SpringApplicationBuilder configure(

SpringApplicationBuilder builder) {

return builder.sources(StartSpringBootMain.class) ;

}

public static void main(String[] args) throws Exception {

SpringApplication.run(StartSpringBootMain.class, args);

}

}

1. 将生成的war文件拷贝到tomcat/webapps下进行热部署；

### https访问控制

正常的https的访问一定需要证书的，为保证这个证书的安全一定要在项目中使用CA进行认证。下面只是在本机做个简单的模拟，利用java提供的keytool命令实现证书的生成。

1. 使用命令行方式进入到一个逻辑下，生成证书：

该服务器端证书生成后名称为“keystore.pl2”,而后别名设置为了“mytomcat”,密码统一设置为了“yangmi”.

1. 将该证书保存到项目中（只要是CLASSPATH路径下即可）。
2. 修改application.yml文件配置SSL安全访问：

需要注意一个问题，如果项目中设置了maven的“<resources>”操作之后一定要保证该配置文件可以正常使用，修改pom.xml文件的resource配置。

1. 如果443的端口没有被其他服务所占用，那么此时一定可以正常使用。

如果想要执行项目的测试不能够使用谷歌浏览器，即使使用火狐也不是能正常访问，因为没有配置CA。

1. 如果用户使用的http请求访问，那么应该跳转到https的443端口上访问。此时需要追加一个配置程序类：

如果用户要访问80端口，则会将其自动跳转到443端口。

6、使用一个已经部署过具有CA认证的tomcat，并且此时的火狐也已经成功的配置上了相应的访问路径，将项目打包放到远程tomcat，启动tomcat。

## thymeleaf页面模板

### 信息输出

1. 显示一个普通的文本信息：

|-现在假设在控制器里面传输了一些简单的信息内容：

而后在message——show.html页面里面进行数据显示时只需要通过“${属性名称}”

“th:text”表示此时只显示一个普通的文本信息。

<p th:text="'官方网站：' + ${url}"/>

1. 如果返回带有html的标签

|  |  |
| --- | --- |
| 转义标签 | <p th:text=”’官方网站：’+${url}”/> |
| 原样显示 | <p  th:utext=”’官方网站：’+${url}”/> |

如果从安全的角度来讲肯定是使用“th:text”的处理方式显示信息才安全，所有的html代码会被过滤掉。

1. “th:text”也可以获取资源文件的内容：

**baiqian**=**我爱杨幂！{0}  
yangmi**=**我爱你**

<**p th:text="#{yangmi}"**/>  
<**p th:text="#{baiqian('仙女')}"**/>

1. “th:text”里面号可以编写基础的运算：

<**p th:text="'算术运算'+(2+6)"**/>

### 路径处理

1. 在src/main/view/static/js目录下创建main.js文件：
2. 后续想要进行方便的访问，可以使用“@{}”进行一个访问的定位处理：

<**script type="text/javascript" th:src="@{/js/main.js}"**/>

1. 在以后进行指定路径访问的时候也可以使用“@{路径}”形式访问Controller。

<**form th:action="@{/add}" method="post"**>

1. 如果是超链接和thymeleaf同时出现，则忽略超链接：

<**a href="www.baidu.com" th:href="www.ym.com"**>杨幂</**a**>

### 处理内置对象

1. 在控制器里面追加一个方法，这个方法将采用内置对象的形式传递属性

@RequestMapping(**"/message/show"**)  
**public** String show(String mid, Model model){  
 model.addAttribute(**"url"**,**"<p style='color:red'>www.ym.com</p>"**);  
 model.addAttribute(**"mid"**,mid);  
 **return "message/message\_shows"**;  
}

其中model传递的本质就是属于request属性范围。

1. 编写页面显示内容

<**p th:text="'官方网站：' + ${url}"**/>  
<**p th:text="'用户名：' + ${mid}"**/>

也可以使用“\*{属性名称}”进行访问；

<**p th:text="'用户名：' + \*{mid}"**/>

1. 不同属性范围的访问

<p th:text="'requestMessage = ' + ${requestMessage}"/>

<p th:text="'sessionMessage = ' + ${session.sessionMessage}"/>

<p th:text="'applicationMessage = ' + ${application.applicationMessage}"/>

1. 实际上thymeleaf也支持有JSP内置对象的获取操作，但一般很少这样使用：

<p th:text="${#httpServletRequest.getRemoteAddr()}"/>

<p th:text="${#httpServletRequest.getAttribute('requestMessage')}"/>

<p th:text="${#httpSession.getId()}"/>

<p th:text="${#httpServletRequest.getServletContext().getRealPath('/')}"/>

### 对象输出

1. 在控制器中输出对象：

@RequestMapping(value = "/message/member\_show", method = RequestMethod.GET)

public String memberShow(Model model) {

Member vo = new Member();

vo.setMid(101L);

vo.setName("啊三");

vo.setAge(9);

vo.setSalary(99999.99);

vo.setBirthday(new Date());

model.addAttribute("member", vo);

return "message/member\_show";

}

1. 页面显示：

<**body**>  
<**p th:text="${member.mid}"**/>  
<**p th:text="${member.name}"**/>  
<**p th:text="${member.age}"**/>  
<**p th:text="${#dates.format(member.birthday,'yyyy-MM-dd')}"**/>  
<**p th:text="${member.salary}"**/>

也可以是使用一种更简单的做法：

<div th:object="${member}">

<p th:text="'用户编号：' + \*{mid}"/>

<p th:text="'用户姓名：' + \*{name}"/>

<p th:text="'用户年龄：' + \*{age}"/>

<p th:text="'用户工资：' + \*{salary}"/>

<p th:text="'出生日期：' + \*{birthday}"/>

<p th:text="'出生日期：' + \*{#dates.format(birthday,'yyyy-MM-dd')}"/>

</div>

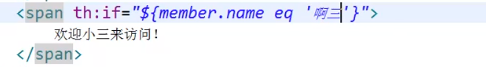
区别：关于“${属性}”和“\*{属性}”区别？

1. 在访问单独或者是完整属性的时候是一样的，而如果要使用“\*{}”形式进行访问往往需要结合“th:object”作为整体对象的声明后才可以直接访问属性。
2. $访问完整信息,而\*访问指定对象中的属性内容，如果访问的只是普通的内容两者没有区别。

### 逻辑判断

在thymeleaf中对于逻辑可以使用如下的运算符完成，例如：and、or、关系比较（>,<,>=,<=,==,!=,lt,gt,le,ge，eq，ne）。

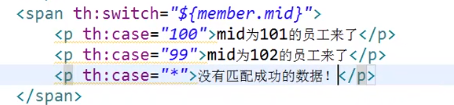




除了这种做法之外还可以实现不满足条件的判断：



使用switch语句进行多内容的判断：

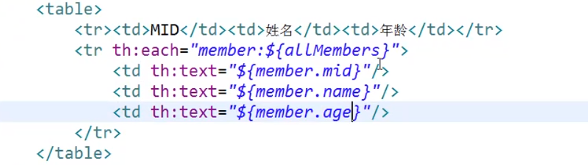


thymeleaf之中实现的switch语句馒头default的存在，只能通过“\*”实现。

### 迭代输出

在开发中数据迭代类型往往会有两类：List、Map。

List迭代：



也可以输出序号：



Map集合迭代：对于Map集合的输出，肯定要使用Map.entry接口完成，对于Map.Entry接口只能够使用getKey()或getValue方法获得。



### 包含处理

数据包含在thymeleaf中提供有两种支持语法：

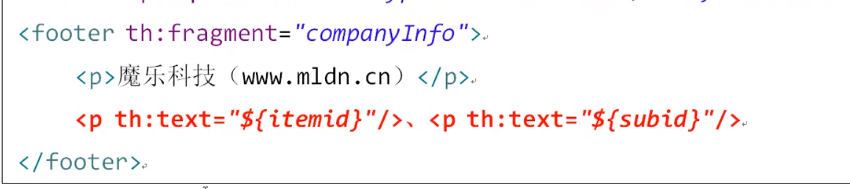
1. th：replace:使用标签替换，原始的宿主标签还在，但是包含标签不在。
2. th:include：进行包含，原始宿主消失，而保留包含的标签。
3. 新建一个脚本呢页面：

<**meta http-equiv="Content-Type" content="text/html;charset=UTF-8"**/>  
<**footer th:fragment="companyInfo"**>  
 <**p**>杨幂小可爱！</**p**>  
</**footer**>

2、页面包含处理：

<**div th:replace="@{/message/footer}::companyInfo"**></**div**>  
<**div th:include="@{/message/footer}::companyInfo"**></**div**>

3、在很多开发中需要向北包含页面传递参数， 使用“th:with”的处理模式。



参数传递：



### 数据处理

在使用list集合的时候会考虑使用get方法获取指定索引的数据，对于set集合则会使用contains来判断某个数据是否存在。对于map集合则使用containsKey判断某个key是否存在，以及使用get根据key获取对应的value，而这些功能在之前并不具备。

1. 通过map集合获取信息

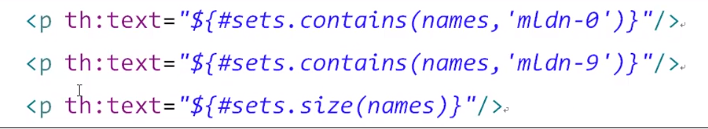
判断key是否存在：



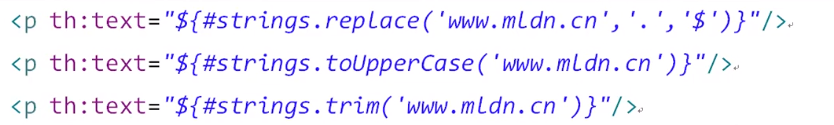
获取指定值：



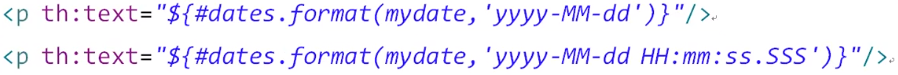
1. Set判断某个数据是否存在



1. 如果此时操作的是List集合只需要把“#sets”换成“#lists”即可。都可以使用索引取得数据，而不关心是set还是list.
2. 处理字符串



1. 日期操作



## 文件上传

### 基础文件上传

### 上传限制

### 上传多个文件

### 使用图片服务器