## 2.6 SpringBoot项目打包

1.【microboot】 项目的pom.xml中追加新的插件：

|  |
| --- |
| <plugin><!--该插值的主要功能是项目的打包处理 -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <!--设置程序执行的主类 -->  <mainClass>com.lxc.intro.microboot.StartSpringBootMain</mainClass>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions>  </plugin> |

2.【micorboot-base】模块项目进行打包：clean package

\*在此打包之前还bulid更新整个项目确保插件下载成功。

打包成功的microboot.jar文件中包含了全部依赖java文件、

3.运行microboot.jar 文件命令：java –jar microboot.jar

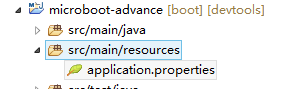
# 3.springBoot配置深入

## 3.1配置环境属性

通过当前的执行可以发现，默认情况下springBoot里面启动WEB容器默认是tomcat

|  |
| --- |
| Tomcat started on port(s): 8080 (http) |

SpringBoot的配置文件必须写在classpath路径下。文件的名称一定要是“application.properties”不能自定的改变其名称。



在springBoot和springColud里面严格来讲可以拥有两种配置文件、

一种是：application.properties.一种是application.yml文件。

(**YML**文件格式是YAML (YAML Aint Markup Language)编写的文件格式，YAML是一种直观的能够被电脑识别的的数据数据序列化格式)

**范例**定义application.yml配置文件：

|  |
| --- |
| server:  port: 80 #设置tomcat的访问端口  context-path: /holle #设置访问的上下文路径 |

如果项目中同时存在application.properties和application.yml两个配置文件的时候。SpringBoot会优先加载.properties里面的配置。如果两个配置文件的配置项冲突的话，会以properties为主。如果不冲突则以为存在的为主。

在之前默认使用的WEB容器为Tomcat容器。如果想切换为jetty容器的话只需要加上一下配置。

|  |
| --- |
| 在pom.xml文件中加入jetty的依赖  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-jetty -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jetty</artifactId>  <version>1.5.6.RELEASE</version>  </dependency> |

如果在以后项目以我们的微服务的方式进行打包发布的时候，基本上使用jetty服务进行发布。因为比较的小巧。

## 3.2读取资源文件

如果要想在SpringBoot里面进行资源文件的配置只需要一些简单的application.yml的配置即可。而且所有注入的资源文件都可以像最初的Spring处理的那样直接使用MessageSource进行读取。

1. 为了统一管理资源文件，在“src/main/resources”目录建立一个i18n的目录存放资源文件。
2. 在“src/main/resources/i18n”目录下建立两个资源文件。

建立Messages.properties.

|  |
| --- |
| welcome.url=www.lxc.com  welcome.msg=\u6B22\u8FCE**{0}**\u5149\u4E34\uFF01 |

建立Pages.properties

|  |
| --- |
| Add=/add.action  Edit=/edit.action |

修改application.yml配置文件

|  |
| --- |
| spring: #表示该配置直接由spring容器处理  messages: #表示资源文件配置  basename: i18n/Messsages,i18n/Pages #配置资源文件名  server:  port: 80 |

1. 当执行完以上的配置后spring容器会自动的创建MessagesSource对象。在使用的时候直接自动注入即可

|  |
| --- |
| /\*\*  \* 读取资源配置类  \*/  **public** **abstract** **class** AbstractBaseController {  @Autowired  **private** MessageSource messageSource;// 有spring自动注入资源对象  **public** String get(String key, String... arg) {  **return** **this**.messageSource.getMessage(key, arg, Locale.*getDefault*());  }  } |

## 3.3Bean配置

1.正常的service的注入方式(将服务类在启动类的子包下，使用@Service注解会自动扫描到MessageServiceImpl类)

|  |
| --- |
| **public** **interface** IMeassageService {  **public** String getMessage();  }  @Service  **public** **class** MessageServiceImpl **implements** IMeassageService {  @Override  **public** String getMessage() {  **return** "www.lxc.com";  }  } |

2．控制器层

|  |
| --- |
| @RestController  **public** **class** MessageController **extends** AbstractBaseController {  @Autowired  **private** IMeassageService messageService;  @RequestMapping("/echo")  **public** String echo(String name) {  System.***out***.println("\*\*读取路径：" + **super**.get("Add"));  **return** **super**.get("Welcome.msg", name);  }  @RequestMapping("/index")  **public** String index() {  **return** **this**.messageService.getMessage();  }  } |

3.测试是否自动注入成功

|  |
| --- |
| import org.junit.Test;  import org.junit.runner.RunWith;  import org.microboot.advance.controller.MessageController;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  import org.springframework.test.context.web.WebAppConfiguration;  @SpringBootTest(classes=StartSpringBootMain.class)  @RunWith(SpringJUnit4ClassRunner.class)  @WebAppConfiguration  public class TestMeassageController {  @Autowired  private MessageController messageController;  @Test  public void testEcho() throws Exception {  System.err.println(messageController.echo("lxc"));  }  @Test  public void testIndex(){  System.out.println(messageController.index());  }  } |

2.使用bean配置的方式注入service,在启动类下的子包中建立一个configuration类型配置Bean

|  |
| --- |
| **package** org.microboot.advance.config;  **import** org.microboot.advance.service.IMessageService;  **import** org.microboot.advance.service.impl.MessageServiceImpl;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  @Configuration  **public** **class** ServiceConfig {  @Bean //此出配置低价与xml的<bean></bean>配置  **public** IMessageService getMessageService(){    **return** **new** MessageServiceImpl();  }  } |

3.当一个项目中已经有了完善的xml的配置，想将项目转为SpringBoot项目。也可以直接在启动类中引入xml的配置，不用转成bean配置的方式

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:jee=*"http://www.springframework.org/schema/jee"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd*  *http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.5.xsd*  *http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *"*>  <bean id=*"messageServiceImpl1"* class=*"org.microboot.advance.service.impl.MessageServiceImpl"*></bean>  </beans> |

启动类型的配置：

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

但是使用springBoot不建议使用xml的配置。一般使用bean的配置方式

## 3.4模板渲染

利用thymeleaf实现一个简单的模板渲染，

1. 首先如果想在项目中使用thymeleaf模板，那么应该首先在项目中导入thymeleaf的依赖库。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  <version>1.5.6.RELEASE</version>  </dependency> |

1. 通过控制层跳转到页面之中进行信息显示。在SrpingMvc使用的是ModelAndView进行的页面跳转，在SpringBoot中直接在参数中注入一个Model即可。

页面定义有严格要求：

在classpath路径下（src/main/resources,src/main/view）必须建立有一个templates的文件夹目录。在这个目录里面保存有thymeleaf所有相关页面。这些页面可以按照文件目录保存。

|  |
| --- |
|  |

3.在编写thymeleaf的页面的时候所有元素一定完结。

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th=*"http://www.thymeleaf.org"*>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"* />  <title>thymeleaf模板渲染</title>  </head>  <body>  <p th:text=*"'springBoot学习：'+${name}"*/>  <p th:text=*"'个人主页：'+${url}"*></p>  </body>  </html> |

4.解决页面内不通过控制器跳转而是直接访问的问题。在thymeleaf的项目之中的父路径下建立一个static的目录。该目录下保存直接通过url访问的页面或者js，cass,img等等

|  |
| --- |
|  |

1. thymeleaf默认的访问页面后缀为.html但是也可以在application.yml里面配置自己的后缀。

|  |
| --- |
| 1. spring: #表示该配置直接由spring容器处理 2. thymeleaf: 3. suffix: .htm 4. messages: #表示资源文件配置 5. basename: i18n/Messages,i18n/Pages #配置资源文件名 6. server: 7. port: 80 |

以上将thymeleaf的后缀更改为.htm。则对应的模板文件也应该是.htm的文件。

## 3.5 profile配置

在实际的开发之中，由于开发（dev），测试(beta)，正式运行(product)的环境不同。所使用的配置文件有可能需要选择不同的配置文件。

SpringBoot里面充分的考虑到了此类问题的存在，那么就提供了多个profile的配置。但是对于我们多个profile的配置我们一定要区分区出，\*.yml和\*.properties，这两种资源的配置是不同的。

### 基于yml实现profile的配置

1.在使用yml配置文件的时候所有可以使用的profile配置项都要求在一个文件之中编写

|  |
| --- |
| #yml实现多个profile配置操作  spring:  profiles:  active: product #表示默认使用dev开发环境的配置信息  ---  spring: #表示该配置直接由spring容器处理  messages: #表示资源文件配置  basename: i18n/Messages,i18n/Pages #配置资源文件名  profiles: dev #开发环境  server: #服务的端口  port: 8080  --- # yml多个profile配置使用 三个“-” 分割  spring:  messages:  basename: i18n/Messages,i18n/Pages  profiles: beta # 测试环境  server:  port: 9090  --- # yml多个profile配置使用 三个“-” 分割  spring:  messages:  basename: i18n/Messages,i18n/Pages  profiles: product # 生产环境  server:  port: 80 |

1. 随后我们可以将项目打包，打包之后我们也可以动态的切换profile配置。

此时我们所有的thymeleaf的页面都保存在“src/main/view”文件夹中，而且所有的springBoot的配置文件也应该都一起输出到\*.jar 文件中。那么就需要配置一个引人资源文件的配置，修改“microboot”项目的pom.xml文件：

|  |
| --- |
| <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  <include>\*\*/\*.tld</include>  </includes>  <filtering>false</filtering>  </resource>  <resource>  <!--配置文件的路径 -->  <directory>src/main/resources</directory>  <!--引入配置文件 -->  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.yml</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> |

Maven 打包命令：clean package

在打包之后在cmd 中执行jar jar –jar microboot.jar

1. 为了动态的切换profile配置的切换。在执行\*.jar文件的时候加入：

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

1. 在页面之中引用小图标

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

### 2.基于application.properties实现多个profile的配置

1.如果使用的是资源文件的配置的方式的话，那么就需要编写多个属性文件：

|  |  |  |
| --- | --- | --- |
| **Application-dev.properties** | **Application-beta.properties** | **Application-product.properties** |
| spring.messages.basename=  i18n/Messages,i18n/Pages  server.port= 8080 | spring.messages.basename=  i18n/Messages,i18n/Pages  server.port= 9090 |  |

2.在需要一个application.properties的配置文件来指定使用那一个配置文件。

|  |
| --- |
| spring.profiles.active=product |

1. 如果将程序也打成\*.jar的话，也可以使用

“--spring.profiles.active=producr”命令动态的切换profile配置。

# 4.SpringBoot错误处理

SpringBoot里对错误的处理提供了三种方式。数据验证错误处理，配置错误页面，全局异常处理。

## 4.1数据验证错误处理、

1.案例表单数据提交

创建表单模型：

|  |
| --- |
| **public** **class** Member {  **private** String email;  **private** Integer age;  **private** Double salary;  **private** Date birthday; |

2.创建MemBer的控制程

|  |
| --- |
| @Controller  @RequestMapping("/member")  **public** **class** MemberController {  @RequestMapping(value = "/add\_pre", method = RequestMethod.***GET***)  **public** String addMemberPre() {  **return** "member/add\_pre";  }  @ResponseBody  @RequestMapping(value="/add",method=RequestMethod.***POST***)  **public** Object add(Member me) {  **return** me;  } |

3.编写MemBer的thymeleaf的表单页面：

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th=*"http://www.thymeleaf.org"*>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"* />  <link rel=*"icon"* type=*"image/x-icon"* href=*"/images/l.ico"*/>  <title>thymeleaf模板渲染</title>  </head>  <body>  <form action=*"add"* method=*"post"*>  成员邮箱：<input type=*"text"* name=*"email"* value=*"921119098@qq.com"* /><br/>  成员年龄：<input type=*"text"* name=*"age"* value=*"18"* /><br/>  成员工资：<input type=*"text"* name=*"salary"* value=*"11000"* /><br/>  成员生日：<input type=*"text"* name=*"birthday"* value=*"2017-08-12"* /><br/>  <input type=*"submit"* value=*"提交"* />  <input type=*"reset"* value=*"重置"* />  </form>  </body>  </html> |

4.由于模型vo中存在日期格式，那么我们就需对日期格式做一个转换处理:对日期的格式加上处理在父类的Controller中加上如下处理

|  |
| --- |
| **public** **abstract** **class** AbstractBaseController {  @InitBinder  **public** **void** initBinder(WebDataBinder binder){  //创建一个将字符串格式化为日期的工具类  SimpleDateFormat format=**new** SimpleDateFormat("yyyy-MM-dd");  //明确此时处理日期格式化的转换类  binder.registerCustomEditor(java.util.Date.**class**, **new** CustomDateEditor(format, **true**));  } |

1. 此时我们代码还存在一些问题
2. 当字段没有输入值则返回null.
3. 某些数据需要验证。比如邮箱。

对于以上的问题,SpringBoot也有提供支持，方便用户编写验证专门提供了一个hibernate-validation.jar工具包。这个工具包有hibernate提供的。

如果要想进行验证，那么首先要解决的问题就是必须是错误的提示信息问。Hibernate-validation。其要求错误的信息保存在

ValidationMessages.properties的配置文件中，在“src/main/resources”下建立配置文件

|  |
| --- |
| member.emali.notnull.error=\u90AE\u7BB1\u4E0D\u80FD\u4E3A\u7A7A\uFF01  member.emali.format.error=\u8BF7\u6B63\u786E\u7684\u8F93\u5165\u90AE\u7BB1\uFF01  member.age.notnull.error=\u5E74\u9F84\u4E0D\u5141\u8BB8\u4E3A\u7A7A\uFF01  member.age.number.error=\u5E74\u9F84\u5FC5\u987B\u662F\u6570\u503C\uFF01  member.salary.notnull.error=\u85AA\u8D44\u4E0D\u80FD\u4E3A\u7A7A\uFF01  member.salary.number.error=\u85AA\u8D44\u5FC5\u987B\u662F\u6570\u7EC4\uFF01  member.birthday.notnull.error=\u751F\u65E5\u4E0D\u80FD\u4E3A\u7A7A\uFF01 |

在molde中属性配置验证：

|  |
| --- |
| @NotBlank(message="{member.emali.notnull.error}")  @Email(message="{member.emali.format.error}")  **private** String email;  @NotNull(message="{member.age.notnull.error}")  @Digits(message="{member.age.number.error}", integer=3,fraction=0)//检查是否是一种数字的整数、分数,小数位数的数字。  **private** Integer age;  @NotNull(message="{member.salary.notnull.error}")  @Digits(message="{member.salary.number.error}", integer=10,fraction=2)//检查是否是一种数字的整数、分数,小数位数的数字。  **private** Double salary;  @NotNull(message="{member.birthday.notnull.error}")  **private** Date birthday; |

修改控制层增加验证：

|  |
| --- |
| @Controller  @RequestMapping("/member")  **public** **class** MemberController **extends** AbstractBaseController {  @RequestMapping(value = "/add\_pre", method = RequestMethod.***GET***)  **public** String addMemberPre() {  **return** "member/add\_pre";  }  @ResponseBody  @RequestMapping(value="/add",method=RequestMethod.***POST***)  **public** Object add(@Valid Member me,BindingResult bindingResult) {  **if**(bindingResult.hasErrors()){//表示提交的数据出现了错误  **return** bindingResult.getAllErrors();  }**else**{  **return** me;  }  }  } |

对此类的验证了解即可，不需要将其作为重点，但是需要清楚，默认情况的验证是通过hibernate的注解和配置文件完成的。

## 4.2错误页面配置处理

错误页是所有WEB项目之中必须具有的一项信息显示处理，但是在普通的WEB项目中错误页面可以在web.xml文件中进行配置的，不过遗憾的是SpringBoot之中没有web.xml文件、如果要想进行错误页面的处理，最好的做法是需要根据每一个错误请求的code创建一个属于自己的错误显示页。

1. 所有的错误页都是静态页面，那么可以在“src/main/view”下专门创建一个error的文件夹来保存所有的错误显示页面。

|  |
| --- |
|  |

1. 添加一个处理错误的配置类

|  |
| --- |
| @Configuration  **public** **class** ErrorPageConfig {  @Bean  **public** EmbeddedServletContainerCustomizer containerCustomizer(){  **return** **new** EmbeddedServletContainerCustomizer(){  @Override  **public** **void** customize(ConfigurableEmbeddedServletContainer container) {  ErrorPage errorPage\_400=**new** ErrorPage(HttpStatus.***BAD\_REQUEST***,"/error/error\_400.html");  ErrorPage errorPage\_404=**new** ErrorPage(HttpStatus.***NOT\_FOUND***,"/error/error\_404.html");  ErrorPage errorPage\_500=**new** ErrorPage(HttpStatus.***INTERNAL\_SERVER\_ERROR***,"/error/error\_500.html");  //加入错误的容器中  container.addErrorPages(errorPage\_400,errorPage\_404,errorPage\_500);  }};  }  } |

此时当出现错误时，就会找到对应的http状态码。跳转到指定的配置页面的路径上。

## 4.3全局异常处理问题

1.定义一个全局的500错误页面详细信息，在temolates下创建一个error.html的文件,这个页面可以输出一些异常的详细信息

|  |
| --- |
| /\*\*  \* 全局异常的处理类  \*/  @ControllerAdvice // 作为一个控制层的切面处理  **public** **class** GlobExceptionHandler {  **public** **static** **final** String ***DEFAULT\_ERROR\_VIEW*** = "error"; // 定义一个全局异常处理页面的路径  @ExceptionHandler(value = Exception.**class**)  **public** ModelAndView defaultErrorHandler(HttpServletRequest request, Exception e) {  ModelAndView mv = **new** ModelAndView(***DEFAULT\_ERROR\_VIEW***);  mv.addObject("e",e);  mv.addObject("url", request.getRequestURL());  **return** mv;  }  } |

2.定义错误信息页面处理：

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th=*"http://www.thymeleaf.org"*>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"* />  <link rel=*"icon"* type=*"image/x-icon"* href=*"/images/l.ico"*/>  <title>thymeleaf模板渲染</title>  </head>  <body>  <h1>HTTP 400 错误 - 请求无效 (Bad request)!</h1>  请求路径：<p th:text=*"${url}"* />  异常信息：<p th:text=*"${e.message}"* />  <img alt=*""* src=*"/images/97.jpg"*/>  </body>  </html> |

1. 对于全局的异常处理可以使用跳转页面的方式进行处理，也可以通过返回json的方式进行处理。

|  |
| --- |
| /\*\*  \* 全局异常的处理类  \*/  @ControllerAdvice // 作为一个控制层的切面处理  **public** **class** GlobExceptionHandler {  **public** **static** **final** String ***DEFAULT\_ERROR\_VIEW*** = "error"; // 定义一个全局异常处理页面的路径  @ResponseBody  @ExceptionHandler(value = Exception.**class**)  **public** Object defaultErrorHandler(HttpServletRequest request, Exception e) {  Map<String,Object> result=**new** HashMap<>();  result.put("url", request.getRequestURL());  result.put("message", e.getMessage());    **return** result;  } |

这种方式主要用于远程调用接口时返回，更加的直观

# 5.springboot整合tomcat

## 5.1spring配置tomcat运行

1.将pom的打包的方式配置成war的方式

|  |
| --- |
| <packaging>war</packaging> |

2.在pom.xml里面配置maven打成war包的插件

|  |
| --- |
| <build>  <finalName>microboot-error</finalName>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-war-plugin</artifactId>  <configuration>  <warName>demo</warName>  </configuration>  </plugin>  </plugins>  </build> |

3.在webapps的目录下加入WEBINF目录下加入web.xml文件

|  |
| --- |
|  |

1. 注释掉pom.xml文件里的jettty服务器的依赖

|  |
| --- |
|  |

1. 在springBoot的启动类继承SpringBoot的tomcat启动类。在覆写配置方法。

|  |
| --- |
| @SpringBootApplication  **public** **class** StartSpringBootMain **extends** SpringBootServletInitializer {  @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. 使用Maven打包命令：clean package 打包项目。在Tomcat的webapps下发布服务。

## 5.2https访问控制

作一个简单的模拟,利用java提供的keytool命令实现证书生成。

1.使用命令行的方式进入一个目录，随后使用keytool命令生成证书

|  |
| --- |
| D:\>keytool -genkey -alias mytomcat -storetype PKCS12 -keyalg RSA -keysize 2048  -keystore keystore.p12 -validity 3650 -dname "CN=Web Server,OU=Unit,O=Organizati  on,L=City,S=State,C=US" -storepass 123456 |

该服务器端证书生成后的名称为“keystore.p12” 别名为：mytomcat 密码为123456

2.将证书放置到classpath的路径下，在修改application.yml文件配置SSL的安全访问。

|  |
| --- |
| server:  port: 443 #https的默认端口为443  ssl:  key-store: classpath:keystore.p12 #keystore配置文件的路径  key-store-type: PKCS12 #keystore的类型  key-alias: mytomcat #keystore的别名  key-password: 123456 #keysore访问密码 |

3.在在父pom.xml文件中的reources标签中加入keysotre文件

|  |
| --- |
| <resource>  <!--配置文件的路径 -->  <directory>src/main/resources</directory>  <!--引入配置文件 -->  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.yml</include>  <include>\*\*/\*.xml</include>  <include>\*\*/\*.tld</include>  <include>\*\*/\*.p12</include>  </includes>  <filtering>false</filtering>  </resource> |

4.在启动项目使用：<https://localhost/>；进行访问。

5.如果用户现在没有通过https访问，你如果在实际的使用之中，也没有输入https，所以最好的的做法是如果现在是进行了http访问，则应该让请求跳转到443端口上面去。要想实现这样的处理需要追加一个程序类：

|  |
| --- |
| **package** com.lxc.intro.microboot.error.config;  **import** org.apache.catalina.Context;  **import** org.apache.catalina.connector.Connector;  **import** org.apache.tomcat.util.descriptor.web.SecurityCollection;  **import** org.apache.tomcat.util.descriptor.web.SecurityConstraint;  **import** org.springframework.boot.context.embedded.EmbeddedServletContainerFactory;  **import** org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainerFactory;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  /\*\*  \* 将http访问请求跳转到https  \*/  @Configuration  **public** **class** HttpConnectorConfig {  **private** Connector initiateHttpConnector() {  Connector connector = **new** Connector("org.apache.coyote.http11.Http11NioProtocol");  connector.setScheme("http");// 如果用户使用的http访问  connector.setPort(8080);// 如果用户访问的是80端口  connector.setSecure(**false**);// 如果该链接为一个跳转则表示不是一个新的连接对象  connector.setRedirectPort(443);// 设置转换端口  **return** connector;  }  @Bean  **public** EmbeddedServletContainerFactory servletContainerFactory() {  TomcatEmbeddedServletContainerFactory tomcat = **new** TomcatEmbeddedServletContainerFactory() {  @Override  **protected** **void** postProcessContext(Context context) {  SecurityConstraint securityConstraint = **new** SecurityConstraint();  securityConstraint.setUserConstraint("CONFIDENTIAL");  SecurityCollection collection = **new** SecurityCollection();  collection.addPattern("/\*");//匹配所有的访问映射路径  securityConstraint.addCollection(collection);  context.addConstraint(securityConstraint);  }  };  tomcat.addAdditionalTomcatConnectors(initiateHttpConnector());  **return** tomcat;  }  } |

# 6.thymeleaf页面模板

## 6.1信息输出

**Th:text** 标签：输出普通文本信息

|  |
| --- |
| <p th:text=*"'欢迎：'+${name}"*/>  <p th:text=*"'主页：'+${url}"*/> |

**Th:utext**标签：输出带html文本

|  |
| --- |
| @RequestMapping("/showStyle")//接受参数，跳转页面  **public** String showStyle(String url,String name,Model model){  model.addAttribute("name","<span style='color: red;'>" +name+"</span>");  model.addAttribute("url", url);    **return** "message/show\_message\_style";  } |

如果从安全的角度出发使用text传递输出信息更加的安全，会过滤掉html的信息，会防止恶意的注入。

**Th:text**标签输出资源文件里面的信息

|  |
| --- |
| Html模板输出：  <p th:text=*"'欢迎：'+#{message.welcome.name(lxc)}"*/>  <p th:text=*"'主页：'+****#****{message.welcome.url}"*/>  注：在模板页面读取直接读取配置文件里面的信息时直接使用”#{}” |

除了以上的操作外，在“th:text”中还可以编写一些基础的运算。

|  |
| --- |
| <p th:text=*"'欢迎：'+${name}+'。除法：'+(20/3)+'乘法：'+(10\*20)+'加法：'+(6+5)+'减法:'+(5-7)+'取余'+(10%2)"*/> |

## 6.2路径处理

**@{}路径标签**

示例：我们在src/main/view/static的目录下建一个.js文件

|  |
| --- |
| window.onload=**function**(){  console.log("\*\*\*\*\*main\*\*\*\*\*\*");  } |

在模板文件的中引入js看路径效果。

|  |
| --- |
| <script type=*"text/javascript"* th:src=*"@{/js/main.js}"*></script>  <scripttype=*"text/javascript"* src=*"${pageContext.request.contextPath}/js/main.js"*></script> |

使用thymeleaf页面中使用@{} 标签对url路径的使用等价于在jsp页面中的

*${pageContext.request.contextPath}*

使用”@{}”跳转Action的请求示例：

|  |
| --- |
| <a th:href=*"@{/message/showStyle(name=lxc,url=www.lxc.com)}"*>点击</a>  注意在thymeleaf的模板中使用“<”小于，“&”的特殊字符是需要特使处理。  在js中的话使用<!<CDATE>>段包裹处理，在路径中传递多个参数时用到&符号是用*/message/showStyle(name=lxc,url=www.lxc.com) 这种方式替代* |

## 6.3内置对象操作

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("/inner")//接受参数，跳转页面  **public** String inner(HttpServletRequest request,Model model){  //1.普通的model传递数据  model.addAttribute("model\_url", "www.lxc.com");  //2.使用request传递对象  request.setAttribute("request\_url", "www.lxc.com");  //3.使用session传递数据  request.getSession().setAttribute("session\_url", "www.lxc.com");  //4.使用application传递数据  request.getServletContext().setAttribute("application\_url", "www.lxc.com");    **return** "message/show\_message\_inner";  } |

2.在页面的不同作用域的标签的取值的方式也是不一样的：

|  |
| --- |
| <p th:text=*"'model\_url:'+${model\_url}"*></p>  <p th:text=*"'request\_url:'+${request\_url}"*></p>  <p th:text=*"'session\_url:'+${session.session\_url}"*></p>  <p th:text=*"'application\_url:'+${application.application\_url}"*></p> |

1. 实际上在thymeleaf里面也支持有jsp内置对象的获取操作

|  |
| --- |
| <p th:text=*"'request对象：'+${#httpServletRequest.getRemoteAddr()}"*></p>  <p th:text=*"'Session对象：'+${#httpSession.getId()}"*></p>  <p th:text=*"'application对象：'+${#httpServletRequest.getServletContext().getRealPath('/')}"*></p> |

## 6.4对象输出

1.首先建立一个domain类型

|  |
| --- |
| **public** **class** Member **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** Long id;  **private** String name;  **private** Integer age;  **private** Double salary;  **private** Date dirthday; |

2.页面正常输出使用“${对象名.属性名的方式}”

|  |
| --- |
| <p th:text=*"'用户ID:'+${member.id}"* />  <p th:text=*"'用户姓名:'+${member.name}"* />  <p th:text=*"'用户年龄:'+${member.age}"* />  <p th:text=*"'用户薪资:'+${member.salary}"* />  <p th:text=*"'用户生日:'+${member.dirthday}"* />  <p th:text=*"'用户生日:'+${#dates.format(member.dirthday,'yyyy-MM-dd')}"* /> |

3.使用 th:object=${对象名} 配合 th:text=\*{属性名}的方式进行输出

|  |
| --- |
| <div th:object=*"${member}"*>  <p th:text=*"'用户ID:'+\*{id}"* />  <p th:text=*"'用户姓名:'+\*{name}"* />  <p th:text=*"'用户年龄:'+\*{age}"* />  <p th:text=*"'用户薪资:'+\*{salary}"* />  <p th:text=*"'用户生日:'+\*{dirthday}"* />  <p th:text=*"'用户生日:'+\*{#dates.format(dirthday,'yyyy-MM-dd')}"* />  </div> |

注：$访问完整信息，而\*访问指定对象中的属性内容，如果访问只是普通内容两者没有区别。

## 6.5逻辑判断

Thymeleaf中的关系比较（> lt,< gt,>= le,<= ge,== eq,!= ne）

标签使用用法：**th:if**

|  |
| --- |
| <p th:if=*"${member.age gt 20}"*>  xxxx  </p>  <p th:if=*"${member.name eq 'xxx'}"*>  欢迎xxx访问！  </p> |

分支标签的用法：**switch case**

|  |
| --- |
| <!--switch 分支语句 -->  <span th:switch=*"${member.id}"*>  <p th:case=*"110"*>拨打110！</p>  <p th:case=*"120"*>拨打120！</p>  <!--default操作 -->  <p th:case=*"\*"*>拨打119！</p>  </span>  在thymeleaf之中实现的switch语句之中并没有default的存在，如果要使用default功能就采用th:case=“\*”的方式。 |

## 6.6迭代输出

|  |
| --- |
| 标签示例：  <table>  <tr>  <th>ID</th>  <th>姓名</th>  <th>年龄</th>  </tr>  <tbody>  <tr th:each=*"member:${list}"*>  <td th:text=*"${member.id}"*></td>  <td th:text=*"${member.name}"*></td>  <td th:text=*"${member.age}"*></td>  </tr>  </tbody>  </table> |

详细的迭代属性输出示例：

|  |
| --- |
| <table>  <tr>  <th>NO.</th>  <th>ID</th>  <th>姓名</th>  <th>年龄</th>  <th>偶数</th>  <th>奇数</th>  </tr>  <tbody>  <tr th:each=*"member,memberState:${list}"*>  <td th:text=*"${memberState.index}"*></td>  <td th:text=*"${member.id}"*></td>  <td th:text=*"${member.name}"*></td>  <td th:text=*"${member.age}"*></td>  <td th:text=*"${memberState.even}"*></td>  <td th:text=*"${memberState.odd}"*></td>  </tr>  </tbody>  </table> |

输出Map集合示例：

|  |
| --- |
| <table>  <tr>  <th>NO.</th>  <th>KEY</th>  <th>ID</th>  <th>姓名</th>  <th>年龄</th>  <th>偶数</th>  <th>奇数</th>  </tr>  <tbody>  <tr th:each=*"member,memberState:${map}"*>  <td th:text=*"${memberState.index}"*></td>  <td th:text=*"${member.key}"*></td>  <td th:text=*"${member.value.id}"*></td>  <td th:text=*"${member.value.name}"*></td>  <td th:text=*"${member.value.age}"*></td>  <td th:text=*"${memberState.even}"*></td>  <td th:text=*"${memberState.odd}"*></td>  </tr>  </tbody>  </table> |

## 6.7包含操作

对于包含操作在thymeleaf模板之中提供有两种支持语法：

Th:replace:是使用标签进行替换

Th:include:是进行包含。

1.首先定义一个包含页面

|  |
| --- |
| <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"* />  <footer th:fragment=*"companyInfo"*>  <p>copyright@www.lxc.com</p>  </footer> |

2.使用替换标签引入页面：

|  |
| --- |
| <span th:replace=*"@{/message/footer}::companyInfo"*></span> |

使用th:replace标签会将原本的span标签所替换掉

3.使用th:include标签引入页面：

|  |
| --- |
| <span th:include=*"@{/message/footer}::companyInfo"*></span> |

4.在很多开发之中都需要向被包含页面进行参数的传递，在thymeleaf之中也可以实现。使用“th:with”语法实现参数传递

修改footer.html页面接受参数：

|  |
| --- |
| <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"* />  <footer th:fragment=*"companyInfo"*>  <p>copyright@www.lxc.com</p>  <p th:text=*"${address}"*></p>  <p th:text=*"${phone}"*></p>  </footer> |

通过th:with语法传递参数

|  |
| --- |
| <span th:include=*"@{/message/footer}::companyInfo"* th:with=*"address=lxc,phone=15208245419"*></span> |

## 6.8数据处理

在thymeleaf之中提供有相应的集合的处理方法，例如：在使用List集合的时候可以考虑采用get()方法获取指定索引的数据，那么在使用Set集合的时候会考虑使用Contains()来判断某个数据是否存在，使用Map集合的时候也希望可以使用containsKey()判断某个key是否存在，以及使用get()根据key获取对应的value.下面我们观察如何使用

|  |
| --- |
| 1. 使用#maps.containsKey()判断Map集合的某个key是否存在   <p th:text=*"${#maps.containsKey(map,'0')}"*></p>   1. 使用Key的方式直接获取map里面的数据   <p th:text=*"${map['0']}"*></p> |

Set集合判断集合里面的内容是否存在：

|  |
| --- |
| <p th:text=*"${#sets.contains(set2,'lxc-1')}"*/> |

根据索引获取指定的数据

|  |
| --- |
| <p th:text=*"${set2[0]}"*/> |

页面获取集合的size

|  |
| --- |
| <p th:text=*"${#sets.size(set2)}"*/><!--获取集合的size --> |

Thymeleaf对字符串的支持操作

|  |
| --- |
| <p th:text=*"${#strings.replace('www.lxc.com','.','$')}"*></p>  <p th:text=*"${#strings.toUpperCase('www.lxc.com')}"*></p>  <p th:text=*"${#strings.trim('www.lxc.com ')}"*></p> |

Thymeleaf的语法支持使用#+java的引用名其实就是调用的java的方法。所以这些java对象中用的方法在Thymeleaf中基本上都可以使用。

对日期格式化操作

|  |
| --- |
| <p th:text=*"${#dates.format(myDate,'yyyy-mm-dd')}"*></p>  <p th:text=*"${#dates.format(myDate,'yyyy-mm-dd HH:MM:SS')}"*></p> |

# 文件上传

## 7.1基础上传

1.上传的form

|  |
| --- |
| <form th:action=*"@{/upload}"* method=*"post"* enctype=*"multipart/form-data"*>  姓名：<input type=*"text"* name=*"name"* /><br/>  文件：<input type=*"file"* name=*"file"*/> <br/>  <input type=*"submit"* value=*"上传"*/>  </form> |

2.后台参数接受

|  |
| --- |
| /\*\*  \* 上传方法  \*/  @ResponseBody  @RequestMapping("/upload")  **public** String upload(String name,MultipartFile file){  System.***out***.println("普通表单字段："+name);  System.***out***.println("注入的类类型："+file.getClass());  System.***out***.println("上传的文件名称1："+file.getName());  System.***out***.println("上传的文件名称："+file.getOriginalFilename());  System.***out***.println("上传的文件类型："+file.getContentType());  System.***out***.println("上传的文件大小："+file.getSize());  **return** "ok";  } |

## 7.2上传限制

1.使用yml文件的配置方式限制上传大小

|  |
| --- |
| server:  port: 80  spring:  messages:  basename: properties/message  http:  multipart:  enabled: **true** #是否支持http上传  location: / #临时文件目录  max-file-size: 2MB #单个上传文件的最大值  max-request-size: 10MB #多个文件上传的最大值  file-size-threshold: 512KB |

1. 使用Bean配置的方式限制文件大小

|  |
| --- |
| @Configuration  **public** **class** UploadConfig {  @Bean  **public** MultipartConfigElement getMultipartConfig(){  MultipartConfigFactory onfigFactory=**new** MultipartConfigFactory();  onfigFactory.setLocation("/");  onfigFactory.setMaxFileSize("2MB");  onfigFactory.setMaxRequestSize("10MB");  **return** onfigFactory.createMultipartConfig();  }  } |

## 7.3多文件上传

1.首先将表单修改成可以上传多个文件的表单

|  |
| --- |
| /\*\*  \* 上传方法  \*/  @ResponseBody  @RequestMapping("/upload")  **public** String upload(String name,HttpServletRequest request){  //判断请示是否是多文件的请求  **if**(request **instanceof** MultipartHttpServletRequest){//使用request获取到 MultipartHttpServletRequest  //强转  MultipartHttpServletRequest mRequest=(MultipartHttpServletRequest)request;  Iterator<String> fileNames = mRequest.getFileNames();//获取到上传的文件name  **while** (fileNames.hasNext()) {  MultipartFile file = mRequest.getFile(fileNames.next());  **if**(file!=**null**){  System.***out***.println("普通表单字段："+name);  System.***out***.println("注入的类类型："+file.getClass());  System.***out***.println("上传的文件名称1："+file.getName());  System.***out***.println("上传的文件名称："+file.getOriginalFilename());  System.***out***.println("上传的文件类型："+file.getContentType());  System.***out***.println("上传的文件大小："+file.getSize());  }  }  }  **return** "ok";  } |

# 拦截器

## 8.1基础拦截器

1.首先建立一个基础的mvc访问控制器

|  |
| --- |
| **public** **class** MyInterceptor **implements** HandlerInterceptor {    **public** MyInterceptor() {  System.***out***.println("拦截器构造方法。。。。。："+MyInterceptor.**class**);  }  @Override  **public** **void** afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, Exception arg3)  **throws** Exception {  System.***out***.println("拦截器处理完成后。。。。。。。。。。。。。。");  }  @Override  **public** **void** postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, ModelAndView arg3)**throws** Exception {  System.***out***.println("拦截器处理中："+arg3.getViewName());  }  @Override  **public** **boolean** preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2) **throws** Exception {  System.***out***.println("进入拦截器进行处理--------------"+arg2.getClass()+"------------------");  HandlerMethod method=(HandlerMethod)arg2;  System.***out***.println(method.getBean().getClass());  System.***out***.println(method.getMethod().getName());  **return** **true**;  } |

## 8.2AOP拦截器

在spring中也通过AOP的拦截器配置，不过大多数的AOP拦截器是处理业务层的拦截器。

1. 首先需要在项目中引入AOP的jar包

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

1. 编写aop代码

|  |
| --- |
| /\*\*  \* 面向切面  \*/  @Aspect  @Component  **public** **class** ServiceAspect {  @Pointcut("execution(\* com.lxc.intro.microboot.upload..\*.\*Service.\*(..))")//配置切入点  **public** **void** pointCut(){}    @Before("pointCut()")  **public** **void** doBefore(JoinPoint joinPoint){  System.***out***.println("AOP Before Advice...");  }  @After("pointCut()")  **public** **void** doAfter(JoinPoint joinPoint){  System.***out***.println("AOP After Advice...");  }  @AfterReturning(pointcut="pointCut()",returning="returnVal")  **public** **void** afterReturn(JoinPoint joinPoint,Object returnVal){  System.***out***.println("AOP AfterReturning Advice:" + returnVal);  }  @AfterThrowing(pointcut="pointCut()",throwing="error")  **public** **void** afterThrowing(JoinPoint joinPoint,Throwable error){  System.***out***.println("AOP AfterThrowing Advice..." + error);  System.***out***.println("AfterThrowing...");  }  @Around("pointCut()")  **public** Object around(ProceedingJoinPoint pjp){  System.***out***.println("AOP Aronud before...");  **try** {  Object proceed = pjp.proceed(pjp.getArgs());  **return** proceed;  } **catch** (Throwable e) {  e.printStackTrace();  }**finally** {  System.***out***.println("AOP Aronud after...");  }  **return** **null**;  } |

# StringBoot 整合Mybatis

## 9.1配置druid数据库连接池

1.druid数据库连接池简介：

该数据连接池是有阿里提供的，其特点是性能很高，同时具备可监控性。在实际的开发之中，已经开始广泛的使用。

2.在使用连接时需要在父pom.xml加入数据库的连接驱动包，druid连接池jar和myBatis的开发包

|  |
| --- |
| <!--orcale 驱动包 -->  <dependency>  <groupId>com.oracle</groupId>  <artifactId>ojdbc14</artifactId>  <version>10.2.0.4.0</version>  </dependency>  <!--druid 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.1.5</version>  </dependency>  <!--mybatis -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.1</version>  </dependency> |

1. 配置项目中的application.yml文件

|  |
| --- |
| server:  port: 80  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: oracle.jdbc.driver.OracleDriver #配置oracle驱动类  url: jdbc:oracle:thin:@localhost:1521:orcl #配置数据库连接地址  username: lxc #配置数据库连接用户名  password: lxc #配置数据库连接用户密码  dbcp2:  min-idle: 5 #配置连接池最小连接数  initial-size: 5 #配置连接池初始化连接数  max-total: 10 #配置连接池最大连接数  max-wait-millis: 200 #配置最大超时断开时间 |

4.测试配置获取数据库连接

|  |
| --- |
| @SpringBootTest(classes=com.lxc.intro.microboot.mybaties.StartSpringBootMain.**class**)  @RunWith(value=SpringJUnit4ClassRunner.**class**)  @WebAppConfiguration  **public** **class** TestDataSource {  @Autowired  **private** DataSource dataSource;  @Test  **public** **void** testConnect() **throws** Exception {  System.***out***.println(dataSource);  }  } |

## 9.2整合Mybatis框架

1.首先得引入Mybatis的开发包

2.在application.yml中配置Mybatis.config.xml

|  |
| --- |
| server:  port: 80  mybatis:  config-location: classpath:mybatis/mybatis-config.xml #mybatis的配置文件路径  type-aliases-package: com.lxc.intro.microboot.mybaties.vo #配置vo下对象的别名  mapper-locations:  - classpath:mybatis/mapper/\*\*/\*.xml #扫面mapper文件下的路径  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: oracle.jdbc.OracleDriver #配置oracle驱动类  url: jdbc:oracle:thin:@localhost:1521:orcl #配置数据库连接地址  username: lxc #配置数据库连接用户名  password: lxc #配置数据库连接用户密码  dbcp2:  min-idle: 5 #配置连接池最小连接数  initial-size: 5 #配置连接池初始化连接数  max-total: 10 #配置连接池最大连接数  max-wait-millis: 200 #配置最大超时断开时间 |

1. 配置Mybatis框架的配置文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <settings>  <!-- 解决数据为NULL报错 -->  <setting name=*"jdbcTypeForNull"* value=*"NULL"* />  <!--开启二级缓存 -->  <setting name=*"cacheEnabled"* value=*"true"*/>  </settings>  </configuration> |

1. 编写Mapper映射配置文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace=*"ths.project.ers.mapper.DispatchDataQueryMapper"*>  <select id=*"findAll"* resultType=*"user"*>  SELECT \* FROM T\_USER  </select>  </mapper> |

1. 分别建立实体类domain类User和service服务IUserService,UserServiceImpl。和Dao类。IUserDao
2. [编写DAO接口整合SpringBoot时一定要加上**@Mapp**r](mailto:7.编写DAO接口整合SpringBoot时一定要加上@Mappr)的注解不如你会扫描不到

|  |
| --- |
| @Mapper  **public** **interface** IUserDao {  List<User> findAll();  } |

## 9.3事物控制

1.首先在Spring-Boot的启动类上加上开启事物注解

|  |
| --- |
| @SpringBootApplication  **@EnableTransactionManagement//开启注解支持**  **public** **class** StartSpringBootMain **extends** SpringBootServletInitializer {    @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** builder.sources(StartSpringBootMain.**class**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  SpringApplication.*run*(StartSpringBootMain.**class**, args);  }  } |

[2.在业务层中使用对应的@Transactional](mailto:2.在业务层中使用对应的@Transactional)来控制事物

|  |
| --- |
| **public** **interface** IUserService {  @Transactional(readOnly=**true**)//只读接口  List<User> findAll();  @Transactional(propagation=Propagation.***REQUIRED***)  **int** insert(User user);  } |

|  |
| --- |
| **在TransactionDefinition接口中定义了七个事务传播行为**。  **PROPAGATION\_REQUIRED**如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。  **PROPAGATION\_SUPPORTS**如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。但是对于事务同步的事务管理器，PROPAGATION\_SUPPORTS与不使用事务有少许不同。  **PROPAGATION\_MANDATORY** 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。  **PROPAGATION\_REQUIRES\_NEW**总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。  **PROPAGATION\_NOT\_SUPPORTED** 总是非事务地执行，并挂起任何存在的事务。  **PROPAGATION\_NEVER** 总是非事务地执行，如果存在一个活动事务，则抛出异常  **PROPAGATION\_NESTED**如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按 |

## 9.4druid数据监控

1.如果要想进行Druid的性能监控操作，则需要做一些基本的配置，如访问的地址是否是白名单。

首先进行sql状态监控的Bean配置

|  |
| --- |
| @Configuration  **public** **class** DruidConfig {  /\*\*  \* 首先配置一个DruidDataSOurce  \*/  @Bean  @ConfigurationProperties(prefix = "spring.datasource") // 使用配置文件的配置  **public** DataSource druidDataSource() {  **return** **new** DruidDataSource();  }  /\*\*  \* 注册Servlet  \*/  @Bean  **public** ServletRegistrationBean servletRegistrationBean() {  ServletRegistrationBean servletRegistrationBean = **new** ServletRegistrationBean(**new** StatViewServlet(),  "/druid/\*");// 配置web访问路径  // 添加初始化参数：initParams  // 白名单：  servletRegistrationBean.addInitParameter("allow", "127.0.0.1");  // IP黑名单 (存在共同时，deny优先于allow) : 如果满足deny的话提示:Sorry, you are not  // permitted to view this page.  servletRegistrationBean.addInitParameter("deny", "192.168.1.73");  // 登录查看信息的账号密码.  servletRegistrationBean.addInitParameter("loginUsername", "lxc");  servletRegistrationBean.addInitParameter("loginPassword", "lxc");  // 是否能够重置数据.  servletRegistrationBean.addInitParameter("resetEnable", "false");  **return** servletRegistrationBean;  }  /\*\*  \* 注册Filet  \*/  @Bean  **public** FilterRegistrationBean filterRegistrationBean() {  FilterRegistrationBean filterRegistrationBean = **new** FilterRegistrationBean();  filterRegistrationBean.setFilter(**new** WebStatFilter());  filterRegistrationBean.addUrlPatterns("/\*");  // 添加不需要忽略的格式信息.  filterRegistrationBean.addInitParameter("exclusions", "\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*");  **return** filterRegistrationBean;  } |

2.第二步非常关键的一点在application.yml中配置druid的过滤器。如果没有配置则状态监控无效

|  |
| --- |
| server:  port: 80  mybatis:  config-location: classpath:mybatis/mybatis-config.xml #mybatis的配置文件路径  type-aliases-package: com.lxc.intro.microboot.mybaties.vo #配置vo下对象的别名  mapper-locations:  - classpath:mybatis/mapper/\*\*/\*.xml #扫面mapper文件下的路径  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: oracle.jdbc.OracleDriver #配置oracle驱动类  url: jdbc:oracle:thin:@localhost:1521:orcl #配置数据库连接地址  username: lxc #配置数据库连接用户名  password: lxc #配置数据库连接用户密码  **filters: stat,wall,log4j**  #druid监控过滤器操作  dbcp2:  min-idle: 5 #配置连接池最小连接数  initial-size: 5 #配置连接池初始化连接数  max-total: 12 #配置连接池最大连接数  max-wait-millis: 200 #配置最大超时断开时间 |

# SringBoot整合消息中间件

### 10.1整合RabbitMQ

进行RabbitMQ整合的时候一定要注意以下几个概念：交换空间、虚拟主机、队列信息。

1.新建两个项目：

microboot-rabbitmq-producer

microboot-rabbitmq-consumer

两个项目分别都需要加入RabbitMQ的支持包

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

2.两个项目的yml文件配置RabbitMQ 连接信息

|  |
| --- |
| **spring**:  **rabbitmq**:  **host**: 192.168.20.85  **port**: 5672  **virtual-host**: /my-virtual-hosts  **username**: java  **password**: java |

3.配置RabbitMQConfig

|  |
| --- |
|  |

# 整合其他服务

## 11.1邮件整合

1.邮件整合服务。邮件整合需要引入email的jar包。

修改pom.xml文件加入maven配置

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

2.修改application.yml文件配置邮件服务

|  |
| --- |
| server:  port: 80  spring:  mail:  host: smtp.qq.com #腾讯邮件服务器地址  username: 921119098@qq.com  password: 921119098@qq.com  properties:  mail.smtp.auth: **true** #必须进行授权认证，它的目的就是阻止他人任意乱发邮件  mail.smtp.starttls.enable: **true** #MTP加密方式:连接到一个TLS保护连接  mail.smtp.starttls.required: **true** |

3.编写邮件发送测试类

|  |
| --- |
| **package** com.lxc.intro.microboot.email;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.mail.SimpleMailMessage;  **import** org.springframework.mail.javamail.JavaMailSender;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  **import** org.springframework.test.context.web.WebAppConfiguration;  @WebAppConfiguration  @SpringBootTest(classes = com.lxc.intro.microboot.email.StartSpringBootMain.**class**)  @RunWith(SpringJUnit4ClassRunner.**class**)  **public** **class** TestSenEmail {  @Autowired  **private** JavaMailSender javaMailSender;  @Test  **public** **void** testEmail() **throws** Exception {  SimpleMailMessage message = **new** SimpleMailMessage();  message.setFrom("921119098@qq.com");// 用户邮箱  message.setTo("839542873@qq.com");// 接受邮箱  message.setSubject("测试邮件发送");  message.setText("测试邮件发送!!!!");  javaMailSender.send(message);  }  } |

## 11.2任务调度

1.使用spring的自带的定时任务配置定时任务调度。

编写定时任务类型

|  |
| --- |
| **package** com.lxc.intro.microboot.email.task;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.scheduling.annotation.Scheduled;  **import** org.springframework.stereotype.Component;  **import** com.lxc.intro.microboot.email.service.IUserService;  @Component  **public** **class** TimeTask {  @Autowired  **private** IUserService userService;  @Scheduled(fixedRate = 1000) // 间隔执行  **public** **void** runJobA() {  userService.println();  }    @Scheduled(cron="\*/3 \* \* \* \* ?")  **public** **void** runJobB() {  userService.println2();  }  } |

2.启动StringBoot的主类加上启动定时任务注解开始定时任务

|  |
| --- |
| @SpringBootApplication  **@EnableScheduling//开始定时任务注解**  **public** **class** StartSpringBootMain **extends** SpringBootServletInitializer {  @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** builder.sources(StartSpringBootMain.**class**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  SpringApplication.*run*(StartSpringBootMain.**class**, args);  }  } |

3.注册线程池，使用多线程的方式分别对每一个定时任务执行。

|  |
| --- |
| **package** com.lxc.intro.microboot.email.config;  **import** java.util.concurrent.Executors;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.scheduling.annotation.SchedulingConfigurer;  **import** org.springframework.scheduling.config.ScheduledTaskRegistrar;  /\*\*  \* 注册定时任务线程池  \*  \* **@author** L  \*/  @Configuration  **public** **class** TaskConfig **implements** SchedulingConfigurer {  @Override  **public** **void** configureTasks(ScheduledTaskRegistrar taskRegistrar) {  taskRegistrar.setScheduler(Executors.*newScheduledThreadPool*(100));// 创建100百个线程池  }  } |

## 11.3Actuator监控

1.如果现在要想在代码之中启动actuator服务，那么首先一定要进行开发包的配置修改我们的POM.xm文件引入jar.

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

2.此时配置此依赖包的项目就有了具有监控的操作能力，但是这个能力无法直接体现，如果需要体现这些信息，则需要修改application.yml配置文件。关闭掉当前的安全配置项。

|  |
| --- |
| server:  port: 80  management:  security:  enabled: **false** #表示关闭了系统的安全配置 |

3.但是如果现在将系统的安全配置关闭了，那么此时你的系统的服务信息就有可能被任何人看见，那么不希望被所有人看见，同时也希望所有的人能够看见一些提示信息。那么就需要自定义监控信息

4.配置info信息，首先我们使用一个读取pom.xml的插件，引入jar

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <configuration>  <delimiters>  <delimiter>$</delimiter>  </delimiters>  </configuration>  </plugin> |

\*在修改miciorboot中的pom.xml配置允许程序访问pom文件

|  |
| --- |
| <resource>  <!--配置文件的路径 -->  <directory>src/main/resources</directory>  <!--引入配置文件 -->  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.yml</include>  <include>\*\*/\*.xml</include>  <include>\*\*/\*.tld</include>  <include>\*\*/\*.keystore</include>  </includes>  <filtering>true</filtering>  </resource> |

在application.yml中配置信息

|  |
| --- |
| server:  port: 80  management:  security:  enabled: **false** #表示关闭了系统的安全配置  info:  pom.name: $project.artifactId$  pom.version: $project.version$ |

5.也可以使用配置类型的方式配置info信息，这种的好处在于，如果有很多的地方都可以引用此模块，不用重复的编写配置文件

|  |
| --- |
| **package** com.lxc.intro.microboot.actuator;  **import** org.springframework.boot.actuate.info.Info.Builder;  **import** org.springframework.boot.actuate.info.InfoContributor;  **import** org.springframework.stereotype.Component;  @Component  **public** **class** ProjectInfo **implements** InfoContributor {  @Override  **public** **void** contribute(Builder arg0) {  arg0.withDetail("version", "1.010.0");  arg0.withDetail("auth", "lxc");  arg0.build();  }  } |

# 整个redis数据库

Redis的数据库的整合在java里面提供的官方工具包：jedis,所以即便你现在使用的是SpringBoot，那么也继续使用此开发包。

## 12.1RedisTemplast操作

1. 在Spring支持的Redis操作之中提供有一个RedisTemplate处理程序，利用这个类可以方便的操作redis基本数据类型。

首先修改pom.xml引用我们的jar包

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  <version>1.5.6.RELEASE</version>  </dependency> |

1. 如果要想使用Redis操作，则一定要修改application.yml的配置文件。在这个配置文件之中要进行Redis的各种配置

|  |
| --- |
| 1. server: 2. port: 80 3. spring: 4. redis: 5. host: 192.168.1.104 #IP 6. port: 6378 #端口 7. database: 0 #数据库 8. password: lxc #连接密码 9. timeout: 1000 #连接超时 10. pool: #连接池配置 11. max-active: 10 #最大的本地连接数 12. max-idle: 8 #最大的保持连接数 13. min-idle: 2 #最小连接数据 14. max-wait: 1000 #最大的等待时间 |

3.下面就可以通过程序类利用RedisTemplate模板进行数据操作。

|  |
| --- |
| **package** com.lxc.intro.redis;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.data.redis.core.RedisTemplate;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  **import** org.springframework.test.context.web.WebAppConfiguration;  @SpringBootTest(classes = com.lxc.intro.microboot.StartSpringBootMain.**class**)  @RunWith(value = SpringJUnit4ClassRunner.**class**)  @WebAppConfiguration  **public** **class** RedisTemolateTest {  @Autowired  RedisTemplate<String, String> template;  @Test  **public** **void** conn() **throws** Exception {  // System.err.println(template);  template.opsForValue().set("key", "hello");  String key = template.opsForValue().get("key");  System.***out***.println(key);  }  } |

## 12.2Redis对象序列化操作

1.如果要想进行Redis对象序列化操作则一定要首先准备一个序列化处理程序类，这个程序类必须实现一个特定的接口。

首先建立一个序列化转换器

|  |
| --- |
| **package** com.lxc.intro.microboot.redis.utils;  **import** org.springframework.core.convert.converter.Converter;  **import** org.springframework.core.serializer.support.DeserializingConverter;  **import** org.springframework.core.serializer.support.SerializingConverter;  **import** org.springframework.data.redis.serializer.RedisSerializer;  **import** org.springframework.data.redis.serializer.SerializationException;  /\*\*  \* redis 实现对象序列化的操作类  \* **@author** L  \*/  **public** **class** RedisObjectSerializer **implements** RedisSerializer<Object> {  // 定义两个序列化对象  **private** Converter<Object, **byte**[]> serializingConverter = **new** SerializingConverter();// 序列化对象  **private** Converter<**byte**[], Object> deSerializingConverter = **new** DeserializingConverter();// 反序列化对象  /\*\*  \* 反序列化  \*/  @Override  **public** Object deserialize(**byte**[] data) **throws** SerializationException {  **if** (data == **null** || data.length == 0) {// 无数据不做操作  **return** **null**;  }  **return** deSerializingConverter.convert(data);  }  /\*\*  \* 序列化  \*/  @Override  **public** **byte**[] serialize(Object object) **throws** SerializationException {  **if** (object == **null**) {  **return** **new** **byte**[0];  }  **return** serializingConverter.convert(object);  }  } |

2.注册自定义的序列化程序

|  |
| --- |
| **package com.lxc.intro.microboot.config;**  **import org.springframework.context.annotation.Bean;**  **import org.springframework.context.annotation.Configuration;**  **import org.springframework.data.redis.connection.RedisConnectionFactory;**  **import org.springframework.data.redis.core.RedisTemplate;**  **import org.springframework.data.redis.serializer.StringRedisSerializer;**  **import com.lxc.intro.microboot.redis.utils.RedisObjectSerializer;**  **@Configuration**  **public class RedisConfig {**  **@Bean**  **public RedisTemplate<String, Object> getRedisTemplate(RedisConnectionFactory connectionFactory) {**  **RedisTemplate<String, Object> redisTemplate = new RedisTemplate<String, Object>();**  **redisTemplate.setConnectionFactory(connectionFactory);**  **redisTemplate.setKeySerializer(new StringRedisSerializer());**  **redisTemplate.setValueSerializer(new RedisObjectSerializer());**  **return redisTemplate;**  **}**  **}** |

3.测试序列

|  |
| --- |
| /\*\*  \* 保存  \*/  @Test  **public** **void** save() **throws** Exception {  User user = **new** User();  user.setAge(24);  user.setName("liuxc");  user.setSex(**false**);  user.setBirthDay(**new** Date());  template.opsForValue().set("user", user);  }  /\*\*  \* 获取  \*/  @Test  **public** **void** get() **throws** Exception {  Object object = template.opsForValue().get("user");  System.***out***.println(object);  } |

## 12.3配置多个Redis数据库

在springBoot中针对Redis的连接配置本质上只提供了一个连接配置项。如果需要更多的连接配置项，那么就需要自己来进行Redis的创建和管理了。

1. 修改application.yml文件，配置两个redis连接配置项

|  |
| --- |
| 1. server: 2. port: 80 3. spring: 4. redis: 5. host: 192.168.1.104 #IP 6. port: 6378 #端口 7. database: 0 #数据库 8. password: lxc #连接密码 9. timeout: 1000 #连接超时 10. pool: #连接池配置 11. max-active: 10 #最大的本地连接数 12. max-idle: 8 #最大的保持连接数 13. min-idle: 2 #最小连接数据 14. max-wait: 1000 #最大的等待时间 15. redis-two: 16. host: localhost #IP 17. port: 6379 #端口 18. database: 0 #数据库 19. timeout: 1000 #连接超时 20. pool: #连接池配置 21. max-active: 10 #最大的本地连接数 22. max-idle: 8 #最大的保持连接数 23. min-idle: 2 #最小连接数据 24. max-wait: 1000 #最大的等待时间 |

2.增加RedisTwo的程序类

|  |
| --- |
| **package** com.lxc.intro.microboot.config;  **import** org.springframework.beans.factory.annotation.Value;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.data.redis.connection.RedisConnectionFactory;  **import** org.springframework.data.redis.connection.jedis.JedisConnectionFactory;  **import** org.springframework.data.redis.core.RedisTemplate;  **import** org.springframework.data.redis.serializer.StringRedisSerializer;  **import** com.lxc.intro.microboot.redis.utils.RedisObjectSerializer;  **import** redis.clients.jedis.JedisPoolConfig;  @Configuration  **public** **class** RedisTwoConfig {  /\*\*  \* 使用jdeisFactory构建连接工厂  \*  \* **@return**  \*/  **private** RedisConnectionFactory getRedisConnectionFactory(String host, **int** port, **int** database, **int** timeout,  **int** maxActive, **int** maxIdle, **int** minIdle, **int** maxWait) {  JedisConnectionFactory jedisConnectionFactory = **new** JedisConnectionFactory();  jedisConnectionFactory.setHostName(host);  jedisConnectionFactory.setPort(port);  jedisConnectionFactory.setDatabase(database);  jedisConnectionFactory.setTimeout(timeout);  JedisPoolConfig poolConfig = **new** JedisPoolConfig();  poolConfig.setMaxTotal(maxActive);  poolConfig.setMaxIdle(maxIdle);  poolConfig.setMinIdle(minIdle);  poolConfig.setMaxWaitMillis(maxWait);  jedisConnectionFactory.setPoolConfig(poolConfig);// 配置连接池  jedisConnectionFactory.afterPropertiesSet();  **return** jedisConnectionFactory;  }  @Bean("redisTwo")  **public** RedisTemplate<String, Object> getRedisTemplate(@Value("${spring.redis-two.host}") String host,  @Value("${spring.redis-two.port}") **int** port, @Value("${spring.redis-two.database}") **int** database,  @Value("${spring.redis-two.timeout}") **int** timeout,  @Value("${spring.redis-two.pool.max-active}") **int** maxActive,  @Value("${spring.redis-two.pool.max-idle}") **int** maxIdle,  @Value("${spring.redis-two.pool.min-idle}") **int** minIdle,  @Value("${spring.redis-two.pool.max-wait}") **int** maxWait) {  RedisTemplate<String, Object> redisTemplate = **new** RedisTemplate<String, Object>();  redisTemplate.setConnectionFactory(  **this**.getRedisConnectionFactory(host, port, database, timeout, maxActive, maxIdle, minIdle, maxWait));// 设置自定义连接池  redisTemplate.setKeySerializer(**new** StringRedisSerializer());  redisTemplate.setValueSerializer(**new** RedisObjectSerializer());  **return** redisTemplate;  }  } |

3.测试RedisTwo连接配置项

|  |
| --- |
| **package** com.lxc.intro.redis;  **import** java.util.Date;  **import** javax.annotation.Resource;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.data.redis.core.RedisTemplate;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  **import** org.springframework.test.context.web.WebAppConfiguration;  **import** com.lxc.intro.microboot.redis.domain.User;  @SpringBootTest(classes = com.lxc.intro.microboot.StartSpringBootMain.**class**)  @RunWith(value = SpringJUnit4ClassRunner.**class**)  @WebAppConfiguration  **public** **class** RedisTwoTest {  @Resource(name = "redisTwo")  **private** RedisTemplate<String, Object> template;  @Test  **public** **void** save() **throws** Exception {  User user = **new** User();  user.setAge(24);  user.setName("liuxc");  user.setSex(**false**);  user.setBirthDay(**new** Date());  template.opsForValue().set("user", user);  }  /\*\*  \* 获取  \*/  @Test  **public** **void** get() **throws** Exception {  Object object = template.opsForValue().get("user");  System.***out***.println(object);  }  } |

# SpringBoot整合RestFul服务

Spring与Restful整合才是微架构的核心，因为Rest是整个在微架构之中进行通信的基础模式。利用JSON实现数据的交互处理。而且Spring里面提供有一个非常强大的RestTemplate操作模板，利用此模板可以轻松实现，Rest的Json数据与各种对象建的自动转换。

## 13.1使用RestTemplate 调用rest服务

使用分布式结构一共建立有三个子模块

Microboot-restfult-api:作为公共类的定义。

Microboot-restfult-provider:作为服务提供者

Microboot-restfult-consumer: 作为服务消费者

1. 在api项目中建立公共的vo类

|  |
| --- |
| 1. package com.lxc.intro.microboot.restful.vo; 2. import java.io.Serializable; 3. import java.util.Date; 4. @SuppressWarnings("serial") 5. public class Member implements Serializable { 6. private String id; 7. private String name; 8. private Integer age; 9. private Double salary; 10. private Date birthDay; 11. public String getId() { 12. return id; 13. } 14. public void setId(String id) { 15. this.id = id; 16. } 17. public String getName() { 18. return name; 19. } 20. public void setName(String name) { 21. this.name = name; 22. } 23. public Integer getAge() { 24. return age; 25. } 26. public void setAge(Integer age) { 27. this.age = age; 28. } 29. public Double getSalary() { 30. return salary; 31. } 32. public void setSalary(Double salary) { 33. this.salary = salary; 34. } 35. public Date getBirthDay() { 36. return birthDay; 37. } 38. public void setBirthDay(Date birthDay) { 39. this.birthDay = birthDay; 40. } 41. @Override 42. public String toString() { 43. return "Member [id=" + id + ", name=" + name + ", age=" + age + ", salary=" + salary + ", birthDay=" + birthDay 44. + "]"; 45. } 46. } |

2.在生产者项目中引用aqi模块，修改pom.xml文件

|  |
| --- |
| <dependency>  <groupId>com.lxc.intro</groupId>  <artifactId>microboot-restful</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency> |

3.在Microboot-restfult-provider 中提供生产Controller

|  |
| --- |
| **package** com.lxc.intro.microboot.restful.provider.controller;  **import** java.util.Date;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestBody;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.lxc.intro.microboot.restful.vo.Member;  @RestController  @RequestMapping("/member")  **public** **class** MemberController {  @RequestMapping(value = "/add", method = RequestMethod.***POST***)  **public** Object add(@RequestBody Member member) {  **return** member;  }  @RequestMapping(value = "/get/{id}", method = RequestMethod.***GET***)  **public** Member add(@PathVariable String id) {  Member member = **new** Member();  member.setAge(24);  member.setName("liuxc1");  member.setSalary(9999.99);  member.setBirthDay(**new** Date());  **return** member;  }  } |

4.【Microboot-restfult-consumer】如果要进行我们resetfult操作，那么一定要注意使用一个RestTemplate模板完成处理，所以我们首先我们要建一个程序配置类，进行模板配置类。

|  |
| --- |
| **package** com.lxc.intro.microboot.restful.consumber.config;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.web.client.RestTemplate;  @Configuration  **public** **class** RestfulConfig {  @Bean  **public** RestTemplate getRestTemplate() {  **return** **new** RestTemplate();  }  } |

5.编写标准的消费者controller

|  |
| --- |
| **package** com.lxc.intro.microboot.restful.consumber.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.RestController;  **import** org.springframework.web.client.RestTemplate;  **import** com.lxc.intro.microboot.restful.vo.Member;  @RestController  @RequestMapping("/consumber")  **public** **class** RestConsumberController {  @Autowired  **private** RestTemplate restTemplate;    @RequestMapping(value="/add",method=RequestMethod.***GET***)  **public** Boolean add(Member member){  Boolean flag = restTemplate.postForObject("http://localhost:8080/member/add", member, Boolean.**class**);  **return** flag;  }  @RequestMapping(value="/get/{id}",method=RequestMethod.***GET***)  **public** Member get(@PathVariable(value="id") String id){  Member member = restTemplate.getForObject("http://localhost:8080/member/get/"+id, Member.**class**);  **return** member;  }  } |

总结：而对于Rest服务的更多考虑，应该包含如下几点

1. 既然服务的提供者只能够被消费者所访问，证明其不可能被所有用户操作，一定需要安全认证。
2. 服务端一定要进行指定业务层和数据层的编写，也就是说每一个服务端都应该具备有一个自己的服务器信息
3. 在服务器端访问非常繁忙的时候，消费端执行时有可能需要短期的熔断机制。
4. 服务端既然是一个独立的组件，那么就必须考虑负载均衡问题。
5. 服务端的调用操作，如果所有的调用都明确的写上消费地址，太麻烦了。
6. 消费者进行处理的时候如果都是自己来直接采用RestTemplate做处理，代码结构太差了。远程业务端最好的调用应该就用接口完成。

## Swagger集成

当你现在建立一些公共的Rest服务的时候就可以利用Swagger进行所有Rest服务描述了。也就是提供的一个说明工具的概念。

1. 如果想要去使用swagger说明操作，则必需的映入一个开发工具包。

|  |
| --- |
| <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.8.0</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.8.0</version>  </dependency> |

1. 定义一个swagger的配置程序类

|  |
| --- |
| 1. **package** com.lxc.intro.microboot.swagger.config; 2. **import** org.springframework.context.annotation.Bean; 3. **import** org.springframework.context.annotation.Configuration; 4. **import** springfox.documentation.builders.ApiInfoBuilder; 5. **import** springfox.documentation.builders.PathSelectors; 6. **import** springfox.documentation.builders.RequestHandlerSelectors; 7. **import** springfox.documentation.service.ApiInfo; 8. **import** springfox.documentation.spi.DocumentationType; 9. **import** springfox.documentation.spring.web.plugins.Docket; 10. **import** springfox.documentation.swagger2.annotations.EnableSwagger2; 11. @Configuration 12. @EnableSwagger2 13. **public** **class** Swagger2Config { 14. @Bean 15. **public** Docket getDocket() { 16. Docket docket = **new** Docket(DocumentationType.***SWAGGER\_2***).apiInfo(**this**.getApiInfo()).select() 17. .apis(RequestHandlerSelectors.*basePackage*("com.lxc.intro.microboot.swagger.controller")) 18. .paths(PathSelectors.*any*()).build(); 19. **return** docket; 20. } 21. **private** ApiInfo getApiInfo() {// 进行API的说明信息配置 22. **return** **new** ApiInfoBuilder().title("SpringBoot中使用swagger构建说明信息").description("更多描述请关注WWW.liuxc.com") 23. .termsOfServiceUrl("http://www.liuxc.com").license("liuxc").version("1.0").build(); 24. } 25. } |

3.修改需要说明的controller程序类，追加相关注解信息

|  |
| --- |
| **package** com.lxc.intro.microboot.swagger.controller;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestBody;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.lxc.intro.microboot.swagger.domain.Member;  **import** io.swagger.annotations.ApiImplicitParam;  **import** io.swagger.annotations.ApiImplicitParams;  **import** io.swagger.annotations.ApiOperation;  @RestController  @RequestMapping("/member")  **public** **class** MemberController {  @ApiOperation(value = "实现成员添加", notes = "就是对人员的添加")  @ApiImplicitParams({  @ApiImplicitParam(name = "member", value = "用户描述的详细实体信息", required = **true**, dataType = "Boolean") })  @RequestMapping("/add")  **public** Object add(@RequestBody Member member) {  System.***out***.println("【member】" + member);  **return** **true**;  }  @ApiOperation(value = "获取成员实体信息", notes = "xxxxx")  @ApiImplicitParams({ @ApiImplicitParam(name = "mid", value = "用户id", required = **true**, dataType = "Member") })  @RequestMapping("/get/{mid}")  **public** Member add(@PathVariable(value = "mid") String mid) {  Member member = **new** Member();  member.setAge(24);  member.setMid(mid);  member.setName("liuxc1");  **return** member;  }  } |
| **访问地址：http://localhost/swagger-ui.html** |

## 13.3动态修改日志级别

用户可以通过远程控制追加日志的显示级别的。

# SpringBoot 整合shiro

SpringBoot没有直接的配置支持，必须自己来进行配置。