# Springboot

目录

[Springboot 1](#_Toc8539)

[一、 SpringBoot入门 2](#_Toc1279)

[1、 springboot简介 2](#_Toc32008)

[2、 Springboot快速入门 3](#_Toc13698)

[二、 SpringBoot基础知识 5](#_Toc19971)

[1、 建立统一父pom 5](#_Toc5022)

[2、 SpringBoot代码测试 7](#_Toc22178)

[3、 SpringBoot启动注解分析 7](#_Toc20105)

[4、 路径访问控制 8](#_Toc28331)

[5、 使用内置对象 10](#_Toc22192)

[6、 项目打包 11](#_Toc26001)

[三、 SpringBoot 配置深入 12](#_Toc17292)

[1、 配置环境属性 12](#_Toc11127)

[2、 读取资源文件 13](#_Toc18712)

[3、 Bean配置 14](#_Toc7998)

[4、 模板渲染 15](#_Toc16925)

[5、 Profile配置 18](#_Toc4665)

[四、 springboot错误处理 20](#_Toc15787)

[1、 数据验证错误处理 20](#_Toc22066)

[2、 配置错误页 23](#_Toc12658)

[3、 全局异常处理 24](#_Toc4327)

[五、 springboot整合tomcat 26](#_Toc16349)

[1、 spring配置tomcat运行 26](#_Toc10284)

[2、 https访问控制 27](#_Toc14368)

[六、 thymeleaf页面模板 28](#_Toc29773)

[1、 信息输出 28](#_Toc26521)

[2、 路径处理 29](#_Toc4720)

[3、 处理内置对象 29](#_Toc20263)

[4、 对象输出 30](#_Toc16127)

[5、 逻辑判断 31](#_Toc9328)

[6、 迭代输出 31](#_Toc8289)

[7、 包含处理 32](#_Toc1583)

[8、 数据处理 33](#_Toc27794)

[七、 文件上传 33](#_Toc17398)

[1、 基础文件上传 34](#_Toc26706)

[2、 上传限制 35](#_Toc23825)

[3、 上传多个文件 35](#_Toc29963)

[4、 使用图片服务器 37](#_Toc14185)

[八、 拦截器 40](#_Toc19098)

[1、 基础拦截器 40](#_Toc22169)

[2、 aop拦截器 42](#_Toc9368)

[九、 springboot整合Mybatis 43](#_Toc23700)

[1、配置druid数据库连接池 43](#_Toc9958)

[2、整合mybatis开发框架 45](#_Toc10379)

[3、事务控制 48](#_Toc7038)

[4、druid监控 51](#_Toc19955)

[十、 springboot整合消息服务组件 53](#_Toc31700)

[1、 整合ActiveMQ组件 53](#_Toc11491)

[2、 整合RabbitMQ组件 56](#_Toc2643)

[3、整合kafka组件 60](#_Toc32280)

[十一、 整合其他服务 61](#_Toc9405)

[1、 邮件服务 61](#_Toc30817)

[2、 定时调度 62](#_Toc32365)

[3、Actuator监控 63](#_Toc8781)

[十二、 springboot整合redis数据库 66](#_Toc10705)

[1、 redisTemplate操作 66](#_Toc14917)

[2、 redis序列化处理 67](#_Toc21146)

[3、 配置多个redis连接 69](#_Toc12411)

[十三、 springboot整合restful 73](#_Toc22122)

[1、 使用RestTemplate调用Rest服务 73](#_Toc18776)

[2、 配置swagger服务 78](#_Toc6161)

[3、动态修改日志级别 80](#_Toc29771)

[十四、 springboot整合shiro验证框架 82](#_Toc2458)

[1、 整合说明与环境配置 82](#_Toc555)

[2、 定义用户认证授权微服务 86](#_Toc2985)

[3、 shiro整合处理 91](#_Toc32650)

[4、 使用redis进行缓存处理 100](#_Toc29958)

[5、thymeleaf整合shiro标签 111](#_Toc23203)

## 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。

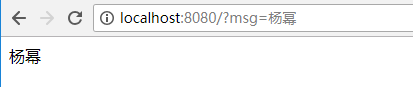
但是一个项目里面可能控制器之中返回的全部是是restful信息，这样分别定义就态麻烦了，为此在SpringBoot里面又提供一个符合注解：@RestController。

@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。

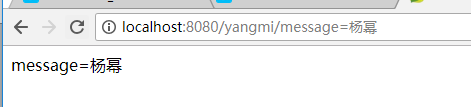
2、现在所给出的控制器类里面只是负责了简单的信息返回，那么实际上也可以进行参数的接收处理，传递参数到控制器中最简单的做法是使用地址重写传递“xx?参数名称=内容”。

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



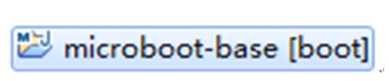
3、由于Springboot支持rest分割处理，所以此时对于参数的接收可以采用路径参数的形式完成。

@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>

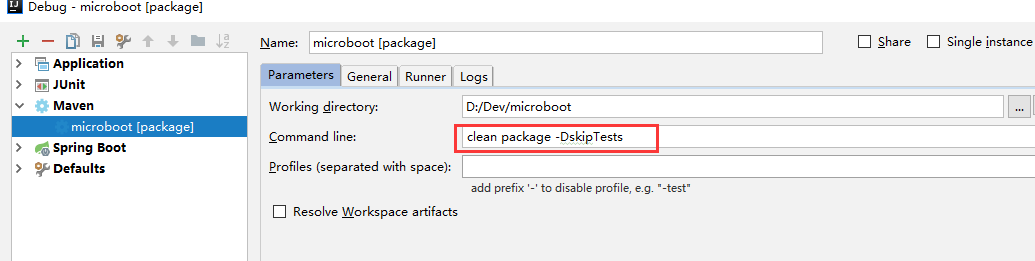
</executions>

</plugin>

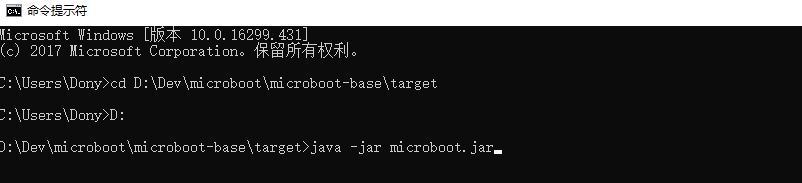
</plugins>

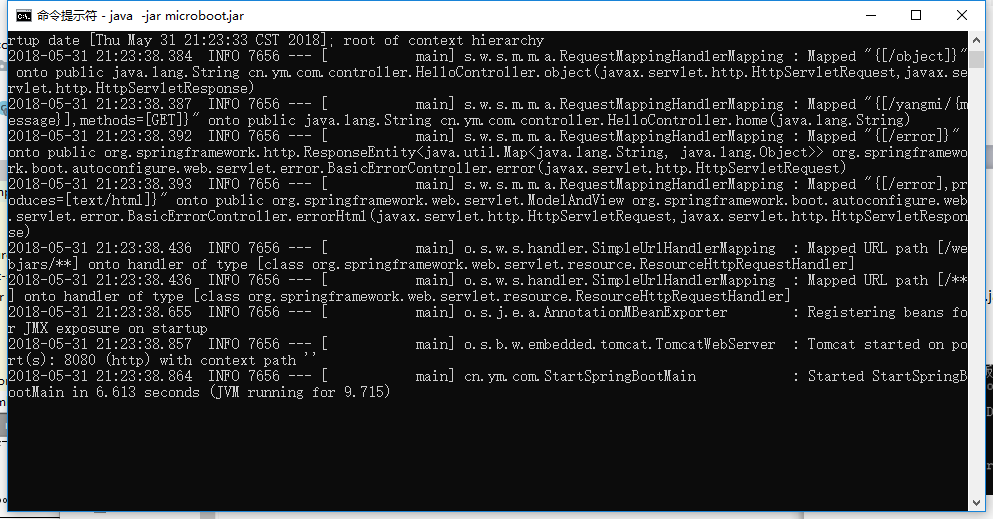
这样每一个模块都可以使用这个配置，如果每个模块不一样则单独配置即可

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，那么会优先进行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：\*.xml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Rest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)。

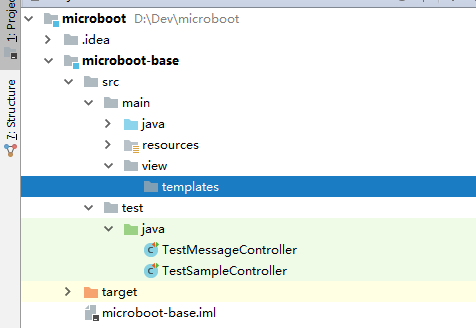
1. 如果要想在项目中使用thymeleaf模板，那么应该首先进行相关支持依赖库的导入：

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

1. 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；

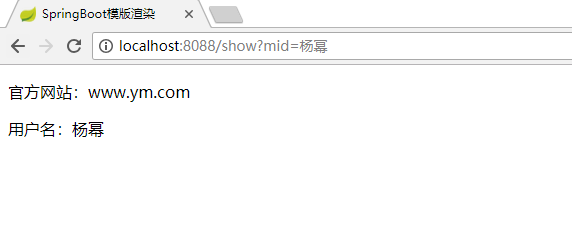
* 建立一个源代码目录：src/main/view



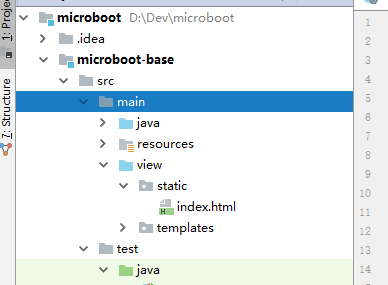
1. 编写控制器：  
    @RequestMapping(**"/show"**)  
    **public** String show(String mid, Model model){  
    model.addAttribute(**"url"**,**"www.ym.com"**);  
    model.addAttribute(**"mid"**,mid);  
    **return "message/message\_show"**;  
    }  
   }
2. 编写message\_show.html页面（编写时所有的元素一定要完结）

<!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或者java -jar -Dspring.profiles.active=beta microboot.jar

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icon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="qqq@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

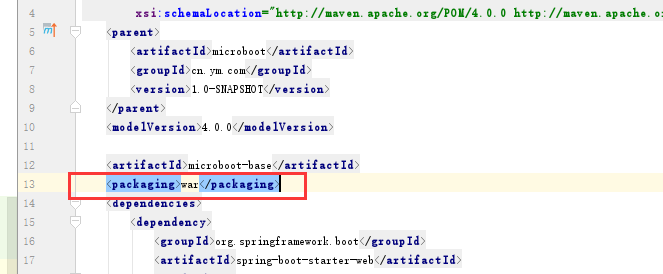
Springboot本身支持两类web容器：默认tomcat、jetty

### 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>

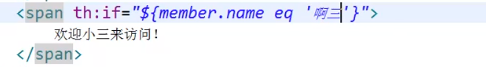
区别：关于“${属性}”和“\*{属性}”区别？

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）。

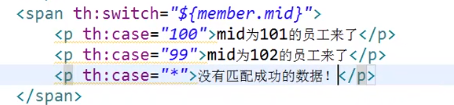




除了这种做法之外还可以实现不满足条件的判断：



使用switch语句进行多内容的判断：

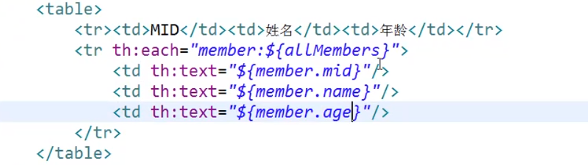


thymeleaf之中实现的switch语句没有default的存在，只能通过“\*”实现。

### 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：

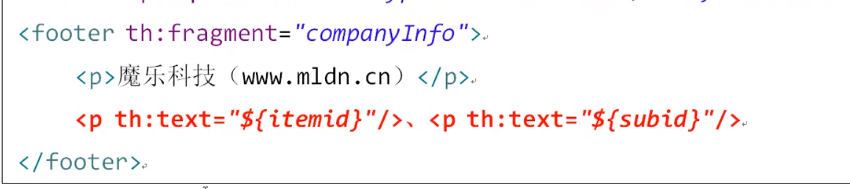
1. th：replace:使用标签替换，原始的宿主标签还在，但是包含标签不在。
2. th:include：进行包含，原始宿主消失，而保留包含的标签。
3. 新建一个脚本呢页面：

<**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

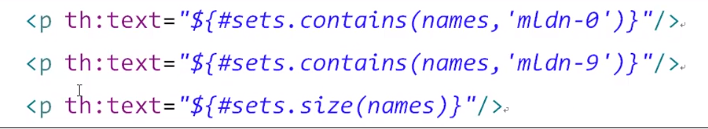
判断key是否存在：



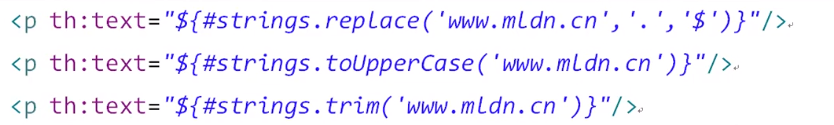
获取指定值：



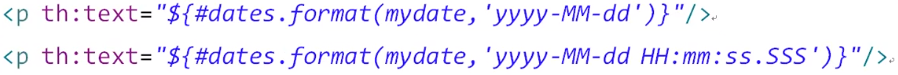
1. Set判断某个数据是否存在



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

在spring中文件上传使用的是Apache的fileupload组件，在springboot里面也同样要继续使用此组件。

### 基础文件上传

1. 如果要进行上传处理，则首先需要整备出相应的控制器：

@Controller(**"upload"**)  
**public class** UploadController **extends** AbstractController {  
  
 @RequestMapping(value = **"/uploadPre"**, method = RequestMethod.***GET***)  
 **public** String uploadPre() {  
  
 **return "upload\_page"**;  
 }  
  
 @RequestMapping(value = **"/upload"**, method = RequestMethod.***POST***)  
 @ResponseBody  
 **public** String upload(String name, MultipartFile photo) {  
 **if** (photo != **null**) {*// 现在有文件上传* System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】name="** + name);  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】photoName = "** + photo.getName());  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】photoContentType ="** + photo.getContentType());  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】photoSize = "**+photo.getSize());  
 **try** {  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】 "**+photo.getInputStream());  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **return "upload-file"**;  
 }  
}

1. 建立一个上传的编辑页面：

<!DOCTYPE **html**>  
<**html xmlns:th="http://www.thymeleaf.org"**>  
<**head**>  
 <**meta charset="UTF-8"/**>  
 <**title**>Title</**title**>  
</**head**>  
<**body**>  
<**form th:action="@{/upload/doUpload}" method="post" enctype="multipart/form-data"**>  
 姓名：<**input type="text" name="name"**/><**br**/>  
 照片：<**input type="file" name="photo"**/><**br**/>  
 <**input type="submit" value="上传"**/>  
</**form**>  
</**body**>  
</**html**>

### 上传限制

如果要想去改变上传的限制，那么最好的做法是进行application.yml配置文件的变更。

**spring:  
 profiles:  
 active:** dev  
 **http:  
 multipart:  
 enabled:** true *# 可以启用http上传* **max-file-size:** 2MB *# 设置支持的单个上传文件的大小限制* **max-request-size:** 100MB *# 设置最大的请求的文件大小，设置总体大小请求* **file-size-threshold:** 512KB *# 当上传文件达到指定配置量的时候会将文件内容写入磁盘* **location:** / *# 设置上传的临时目录*

除了采用以上的配置文件编写之外，也可以单独编写一个配置程序类。

@Configuration  
**public class** UploadConfig {  
  
 @Bean  
 **public** MultipartConfigElement getMultipartConfig() {  
 MultipartConfigFactory configFactory = **new** MultipartConfigFactory();  
 configFactory.setMaxFileSize(**"2MB"**); *// 设置上传文件的单个大小限制* configFactory.setMaxRequestSize(**"100MB"**); *// 设置总的上传的大小限制* configFactory.setLocation(**"/"**); *//设置临时保存目录* **return** configFactory.createMultipartConfig(); *// 创建一个上传配置* }  
}

### 上传多个文件

1. 修改上传表单：

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

1. 修改控制器之中的参数接收，这个地方需要一点点特殊处理；

@Controller  
@RequestMapping(**"/upload"**)  
**public class** UploadController **extends** AbstractController {  
  
 @RequestMapping(value = **"/uploadPre"**, method = RequestMethod.***GET***)  
 **public** String uploadPre() {  
  
 **return "upload\_page"**;  
 }  
  
 @RequestMapping(value = **"/doUpload"**, method = RequestMethod.***POST***)  
 @ResponseBody  
 **public** String upload(String name, HttpServletRequest request) {  
 *// 如果是MultipartHttpServletRequest对象* **if** (request **instanceof** MultipartHttpServletRequest) {  
 MultipartHttpServletRequest multipartHttpServletRequest = (MultipartHttpServletRequest) request;  
 List<MultipartFile> files = multipartHttpServletRequest.getFiles(**"photo"**);  
 Iterator<MultipartFile> iter = files.iterator();  
 **while** (iter.hasNext()) {  
 MultipartFile photo = iter.next();  
 **if** (photo != **null**) {*// 现在有文件上传* System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】name="** + name);  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】photoName = "** + photo.getName());  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】photoContentType ="** + photo.getContentType());  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】photoSize = "** + photo.getSize());  
 **try** {  
 System.***out***.println(**"【\*\*\* 文件上传 \*\*\*\*】 "** + photo.getInputStream());  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 **return "upload-file"**;  
 }  
  
}

多个上传文件的信息要通过Spring包裹的request对象获得。

### 使用图片服务器

在微架构系统之中，如果要保存上传的图片内容，肯定要保存到图片服务器中，本次使用fastdfs图片服务器完成整体的图片保存。

1. 如果要想使用图片服务器，则一定要进行依赖包的配置。

<dependency>

<groupId>com.github.kischang</groupId>

<artifactId>fastdfs-client</artifactId>

<version>0.1</version>

</dependency>

1. 由于该图片服务器有token认证处理过程，所以一定要保证有认证的配置文件，在项目类路径下创建fastdfs\_client.conf文件。

tracker\_server=192.168.1.225:22122  
http.anti\_steal\_token=true  
http.anti\_steal.secret\_key=yangmi1234567890

1. 修改控制器中的上传处理操作：

@Controller  
@RequestMapping(**"/upload"**)  
**public class** UploadController **extends** AbstractController {  
  
 @RequestMapping(value = **"/uploadPre"**, method = RequestMethod.***GET***)  
 **public** String uploadPre() {  
  
 **return "upload\_page"**;  
 }  
  
 @RequestMapping(value = **"/show"**, method = RequestMethod.***GET***)  
 **public** String show(String groupId, String fileId, Model model) **throws** Exception {  
 *//通过ClassPath获取要使用的配置文件* ClassPathResource classPathResource = **new** ClassPathResource(**"fastdfs\_client.conf"**);  
 *//进行客户端访问的整体配置。需要知道配置文件的完成路径* ClientGlobal.*init*(classPathResource.getClassLoader().getResource(**"fastdfs\_client.conf"**).getPath());  
 *//fastdfs的核心操作在于tracker处理上，所以此时需要定义tracker客户端* TrackerClient trackerClient = **new** TrackerClient();  
 *//定义trackerServer富人配置信息* TrackerServer trackerServer = trackerClient.getConnection();  
 **int** ts = (**int**) (System.*currentTimeMillis*() / 1000); *//时间参考* StringBuffer fileUrl = **new** StringBuffer();  
 fileUrl.append(**"http://"**);  
 fileUrl.append(**"fastdfs-tracker"**);  
 fileUrl.append(**"/"** + groupId + **"/"**).append(fileId);  
 fileUrl.append(**"?token="**).append(  
 ProtoCommon.*getToken*(fileId, ts, ClientGlobal.*g\_secret\_key*));  
 fileUrl.append(**"&ts="**).append(ts);  
 System.***out***.println(fileUrl);  
 trackerServer.close();  
 model.addAttribute(**"image"**, fileUrl);  
 **return "upload\_show"**;  
 }  
  
 @RequestMapping(value = **"/doUpload"**, method = RequestMethod.***POST***)  
 @ResponseBody  
 **public** String upload(String name, HttpServletRequest request) **throws** Exception {  
 *// 如果是MultipartHttpServletRequest对象* **if** (request **instanceof** MultipartHttpServletRequest) {  
 MultipartHttpServletRequest multipartHttpServletRequest = (MultipartHttpServletRequest) request;  
 List<MultipartFile> files = multipartHttpServletRequest.getFiles(**"photo"**);  
 Iterator<MultipartFile> iter = files.iterator();  
 **while** (iter.hasNext()) {  
 MultipartFile photo = iter.next();  
 **if** (photo != **null**) {*// 现在有文件上传  
 //如果想进行上传则一定需要获取到文件的扩展名称* String fileExtName = photo.getContentType().substring(photo.getContentType().lastIndexOf(**"/"**) + 1);  
 *//通过ClassPath获取要使用的配置文件* ClassPathResource classPathResource = **new** ClassPathResource(**"fastdfs\_client.conf"**);  
 *//进行客户端访问的整体配置。需要知道配置文件的完成路径* ClientGlobal.*init*(classPathResource.getClassLoader().getResource(**"fastdfs\_client.conf"**).getPath());  
 *//fastdfs的核心操作在于tracker处理上，所以此时需要定义tracker客户端* TrackerClient trackerClient = **new** TrackerClient();  
 *//定义trackerServer富人配置信息* TrackerServer trackerServer = trackerClient.getConnection();  
 *//在整个fastdfs之中真正负责干活的就是storage* StorageServer storageServer = **null**;  
 StorageClient1 storageClient1 = **new** StorageClient1(trackerServer, storageServer);  
 *//定义上传文件的元数据* NameValuePair[] metaList = **new** NameValuePair[3];  
 metaList[0] = **new** NameValuePair(**"fileName"**, photo.getOriginalFilename());  
 metaList[1] = **new** NameValuePair(**"fileExtName"**, fileExtName);  
 metaList[2] = **new** NameValuePair(**"fileLength"**, String.*valueOf*(photo.getSize()));  
 *//如果要上传则使用trackClient对象完成* String upload\_file = storageClient1.upload\_file1(photo.getBytes(), fileExtName, metaList);  
 System.***out***.println(upload\_file);  
 trackerServer.close();  
 }  
 }  
 }  
 **return "upload-file"**;  
 }  
}

1. 修改父项目pom.xml文件，追加编译是的资源文件拷贝（conf）：

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

1. 创建图片显示页面

upload\_show.html文件：

<body>

<img th:src="${image}"/>

</body>

upload\_page.html:

## 拦截器

在springboot中，使用的拦截器与spring中的拦截器完全一样。

### 基础拦截器

拦截器是一种aop的操作实现，用户不用关注其存在。

1. 创建一个程序的控制器类：

@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 = **"/member\_add"**, method = RequestMethod.***POST***)  
 @ResponseBody  
 **public** Object memberAdd(Member member) {  
 **return** member;  
 }  
}

1. 定义一个member\_add.html的页面实现表单定义：

<!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**>  
<**form th:action="@{/member\_add}" method="post"**>  
 用户编号：<**input type="text" name="mid" value="101"**/><**br**/>  
 用户姓名：<**input type="text" name="name" value="SMITH"**/><**br**/>  
 <**input type="submit" value="表单提交"**/>  
</**form**>  
</**body**>  
</**html**>

1. 此时一个正常的mvc的代码就实现了，随后需要编写一个拦截器，对其进行控制。

(1)为了更好的说明问题，现在将拦截器定义在外包（即不在子包下）中

**public class** MyInterceptor **implements** HandlerInterceptor {  
 **private** Logger **log** = LoggerFactory.*getLogger*(MyInterceptor.**class**);  
  
 @Override  
 **public boolean** preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o) **throws** Exception {  
 HandlerMethod handlerMethod = (HandlerMethod) o;  
 **this**.**log**.info(**"【\*\*\* MyInterceptor.preHandle() \*\*\*】"** + handlerMethod.getBean().getClass().getSimpleName());  
 *// 如果返回false表示不继续请求，如果返回true表示继续请求* **return true**;  
 }  
  
 @Override  
 **public void** postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView) **throws** Exception {  
 HandlerMethod handlerMethod = (HandlerMethod) o;  
 **this**.**log**.info(**"【\*\*\* MyInterceptor.postHandle() \*\*\*】"** + handlerMethod.getBean().getClass().getSimpleName());  
 **this**.**log**.info(**"【\*\*\* MyInterceptor.postHandle() \*\*\*】"** + modelAndView);  
 }  
  
 @Override  
 **public void** afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) **throws** Exception {  
 **this**.**log**.info(**"【\*\*\* MyInterceptor.afterCompletion() \*\*\*】拦截处理完毕"** );  
 }  
}

1. 如果想要使用拦截器则必须有一个拦截器的配置类。

@Configuration  
**public class** MyWebApplicationConfig **extends** WebMvcConfigurerAdapter {*// 定义MVC配置* @Override  
 **public void** addInterceptors(InterceptorRegistry registry) {*// 进行拦截器的注册处理操作* registry.addInterceptor(**new** MyInterceptor()).addPathPatterns(**"/\*\*"**);*// 匹配路径* **super**.addInterceptors(registry);  
 }  
}

以上为web中拦截器。

### aop拦截器

spring中还提供有一种aop拦截器配置，不过大部分的aop拦截器都是围绕着业务层进行拦截处理的。

1. 建立一个普通的业务操作接口和它的子类：

**public interface** IMemberService {  
 **public** Member get(**long** mid) ;  
}

@Service  
**public class** MemberServiceImpl **implements** IMemberService {  
 @Override  
 **public** Member get(**long** mid) {  
 Member vo = **new** Member();  
 vo.setMid(mid);  
 vo.setName(**"KING"**);  
 vo.setSalary(50000.00);  
 **return** vo;  
 }  
}

1. 修改控制层：

@Controller  
**public class** MemberController **extends** AbstractController {  
 @Resource  
 **private** IMemberService **memberService**;  
  
 @RequestMapping(value = **"/addPre"**, method = RequestMethod.***GET***)  
 **public** String addPre() { *// 增加前的准备操作路径* **return "member\_add"**;  
 }  
  
 @RequestMapping(value = **"/member\_add"**, method = RequestMethod.***POST***)  
 @ResponseBody  
 **public** Object memberAdd(Member member) {  
 **return** member;  
 }  
  
 @RequestMapping(value = **"/member\_get"**, method = RequestMethod.***GET***)  
 @ResponseBody  
 **public** Object get(**long** mid) {  
 **return this**.**memberService**.get(mid);  
 }  
}

1. 希望在调用的过程中进行拦截处理，所以想要实现这样的处理，那么就需要引入新的开发包：

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

1. 编写一个aop拦截的控制程序类。

@Aspect  
@Component  
**public class** ServiceAspect {*// 此时定义有一个业务层的拦截处理* **private** Logger **log** = LoggerFactory.*getLogger*(ServiceAspect.**class**);  
  
 @Around(**"execution(\* cn.ym.com..service..\*.\*(..))"**)  
 **public** Object arroundInvoke(ProceedingJoinPoint point) **throws** Throwable {  
 **this**.**log**.info(**"【\*\*\* Service-Before \*\*\*】执行参数："** + Arrays.*toString*(point.getArgs()));  
 Object obj = point.proceed(point.getArgs()); *// 进行具体业务调用* **this**.**log**.info(**"【\*\*\* Service-After \*\*\*】返回结果："** + obj);  
 **return** obj;  
 }  
}

## springboot整合Mybatis

### 1、配置druid数据库连接池

这个数据库连接池的配置是有阿里提供的，并且由于其性能很高，同时具备有很好的监控性。

1. 首先编写一个数据库创建脚本：

DROP DATBASE IF EXISTS mldn ;

CREATE DATBASE mldn CHARACTER SET UTF8 ;

USE mldn ;

CREATE TABLE dept (

deptno BIGINT AUTO\_INCREMENT ,

dname VARCHAR(50) ,

CONSTRAINT pk\_deptno PRIMARY KEY(deptno)

) ;

INSERT INTO dept(dname) VALUES ('开发部') ;

INSERT INTO dept(dname) VALUES ('财务部') ;

INSERT INTO dept(dname) VALUES ('市场部') ;

INSERT INTO dept(dname) VALUES ('后勤部') ;

INSERT INTO dept(dname) VALUES ('公关部') ;

1. 配置druid数据源，追加开发包：

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.0.4</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>druid</**artifactId**>  
 <**version**>1.0.31</**version**>  
</**dependency**>

1. 修改application.yml配置文件即可：

**spring:  
 profiles:  
 active:** dev  
 **datasource:  
 type:** com.alibaba.druid.pool.DruidDataSource *# 配置当前要使用的数据源的操作类型* **driver-class-name:** com.mysql.jdbc.Driver *# 配置MySQL的驱动程序类* **url:** jdbc:mysql://localhost:3306/mldn?SSL=false  
 **username:** root  
 **password:** 12345  
 **dbcp2:** *# 进行数据库连接池的配置* **min-idle:** 5 *# 数据库连接池的最小维持连接数* **initial-size:** 5 *# 初始化提供的连接数* **max-total:** 5 *# 最大的连接数* **max-wait-millis:** 200 *# 等待连接获取的最大超时时间*

1. 要进行Junit代码测试，则一定要将mybatis开发包配置过来，因为只有在mybatis开发包里面，才会把druid配置的数据库连接池变为所需要的dataSource数据源对象。

<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>1.3.0</**version**>  
</**dependency**>

1. 测试当前的连接池是否可用：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestDataSource {  
 @Resource  
 **private** DataSource **dataSource**;  
  
 @Test  
 **public void** testConnection() **throws** Exception {  
 System.***out***.println(**this**.**dataSource**);  
 }  
}

### 2、整合mybatis开发框架

如果要进行mybatis的配置，一定要导入spring-boot嗦支持的mybatis开发包。

<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>1.3.0</**version**>  
</**dependency**>

1、修改application.yml配置文件，追加mybatis的相关配置项：

**server:  
 port:** 8080  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **datasource:  
 type:** com.alibaba.druid.pool.DruidDataSource *# 配置当前要使用的数据源的操作类型* **driver-class-name:** org.gjt.mm.mysql.Driver *# 配置MySQL的驱动程序类* **url:** jdbc:mysql://localhost:3306/mldn *# 数据库连接地址* **username:** root *# 数据库用户名* **password: root** *# 数据库连接密码* **dbcp2:** *# 进行数据库连接池的配置* **min-idle:** 5 *# 数据库连接池的最小维持连接数* **initial-size:** 5 *# 初始化提供的连接数* **max-total:** 5 *# 最大的连接数* **max-wait-millis:** 200 *# 等待连接获取的最大超时时间***mybatis:  
 config-location:** classpath:mybatis/mybatis.config.xml *# mybatis配置文件所在路径* **type-aliases-package:** cn.ym.com.vo *# 定义所有操作类的别名所在包* **mapper-locations:** *# 所有的mapper映射文件* - **classpath:** mybatis/mapper/\*\*/\*.xml

2、建立Dept的VO类(包名要和application.yml中一致)：

**public class** Dept **implements** Serializable {  
 **private** Long **deptno**;  
 **private** String **dname**;  
  
 **public** Long getDeptno() {  
 **return deptno**;  
 }  
  
 **public void** setDeptno(Long deptno) {  
 **this**.**deptno** = deptno;  
 }  
  
 **public** String getDname() {  
 **return dname**;  
 }  
  
 **public void** setDname(String dname) {  
 **this**.**dname** = dname;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Dept [deptno="** + **deptno** + **", dname="** + **dname** + **"]"**;  
 }  
}

1. 建立src/main/resources/mybatis/mybatis.config.xml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**>  
 *<!-- 进行Mybatis的相应的环境的属性定义 -->* <**settings**> *<!-- 在本项目之中开启二级缓存 -->* <**setting name="cacheEnabled" value="true"**/>  
 </**settings**>  
</**configuration**>

4、在src/main/resources/mybatis下建立一个mapper子目录，而后在里面定义有Dept.xml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="cn.ym.com.dao.IDeptDAO"**>  
 <**select id="findAll" resultType="Dept"**>  
 SELECT deptno,dname FROM dept ;  
 </**select**>  
</**mapper**>

1. 建立IDeptDao接口，注意所在的包：

@Component

@Mapper  
**public interface** IDeptDAO {  
 List<Dept> findAll();  
}

注意：一定要添加“@Mapper”，否则无法扫描dao

1. 建立IDeptService接口：

**public interface** IDeptService {  
 **public** List<Dept> list() ;  
}

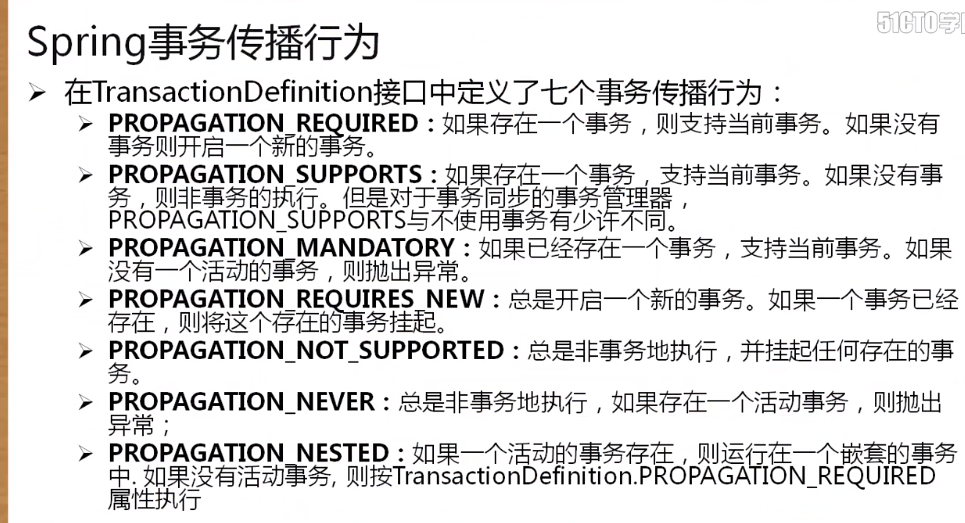
@Service  
**public class** DeptServiceImpl **implements** IDeptService {  
 @Resource  
 **private** IDeptDAO **deptDAO**;  
  
 @Override  
 **public** List<Dept> list() {  
 **return this**.**deptDAO**.findAll();  
 }  
}

1. 编写测试类：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestDeptService {  
 @Resource  
 **private** IDeptService **deptService**;  
  
 @Test  
 **public void** testList() **throws** Exception {  
 System.***out***.println(**this**.**deptService**.list());  
 }  
}

### 3、事务控制

在Spring之中提供的事务控制里面有以下几种数据级别：



如果要在springboot里面去启用mybatis事务（数据库的事务），则可以通过如下的方式完成：

1. 修改IDeptService接口，追加一个只读事务控制：

**public interface** IDeptService {  
 @Transactional(readOnly = **true**)  
 **public** List<Dept> list();  
}

此时配置了一个只读的事务操作，那么也就是说在这个业务方法只能够采用读的模式来进行操作。

1. 配置了一个注解并不表示当前已经合理的支持了事务，如果要进行事务的启用，还需要在启动类上追加一个新的注解配置：

@SpringBootApplication *// 启动SpringBoot程序，而后自带子包扫描*@EnableTransactionManagement  
**public class** StartSpringBootMain {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(StartSpringBootMain.**class**, args);  
 }  
}

1. 编写一个数据增加操作，而后为这个方法设置只读配置：
2. 修改IDeptDAO接口，追加方法：

**boolean** doCreate(Dept vo) ;

1. 修改Dept.xml配置文件：

<insert id="doCreate" parameterType="Dept">

INSERT INTO dept(dname) VALUES (#{dname}) ;

</insert>

1. 在IDeptService接口之中追加有一个业务方法：

@Transactional(readOnly = **true**)  
**public boolean** add(Dept vo);

子类：

@Override  
**public boolean** add(Dept vo) {  
 **return deptDAO**.doCreate(vo);  
}

1. 编写测试类：

@Test  
**public void** testAd() **throws** Exception{  
 Dept dept = **new** Dept();  
 dept.setDname(**"测试部"**);  
 System.***out***.println(**this**.**deptService**.add(dept));  
}

1. 在实际的工作中，对于更新操作应该强制性的启动一个事务控制：

@Transactional(propagation = Propagation.***REQUIRED***)  
**public boolean** add(Dept vo);

1. 在使用spring+mybatis里面处理的时候应该考虑到信息显示问题，所以建议使用logback日志组件来进行日志信息的配置：
2. 将logback.xml配置文件拷贝到src/main/resources目录之中;

*<?***xml version="1.0" encoding="UTF-8"***?>*<**configuration scan="true"**>  
 <**property name="APP" value="${project.artifactId}"** />  
 <**property name="LOG\_HOME" value="/data/www/log/${APP}"** />  
 <**appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender"**>  
 <**encoder**>  
 <**pattern**>%d{yy-MM-dd.HH:mm:ss.SSS} [%-16t] %-5p %-22c{0} %X{ServiceId} - %m%n</**pattern**>  
 </**encoder**>  
 </**appender**>  
 <**appender name="DETAIL"  
 class="ch.qos.logback.core.rolling.RollingFileAppender" additivity="false"**>  
 <**File**>${LOG\_HOME}/${APP}\_detail.log</**File**>  
 <**encoder**>  
 <**pattern**>%d{yy-MM-dd.HH:mm:ss.SSS} [%-16t] %-5p %-22c{0} %X{ServiceId} - %m%n</**pattern**>  
 </**encoder**>  
 <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  
 <**fileNamePattern**>${LOG\_HOME}/${APP}\_detail.log.%d{yyyyMMdd}</**fileNamePattern**>  
 </**rollingPolicy**>  
 </**appender**>  
 <**appender name="ACCESS"  
 class="ch.qos.logback.core.rolling.RollingFileAppender" additivity="false"**>  
 <**File**>${LOG\_HOME}/${APP}\_access.log</**File**>  
 <**encoder**>  
 <**pattern**>%d{yy-MM-dd.HH:mm:ss.SSS};%X{ServiceId};%m%n</**pattern**>  
 </**encoder**>  
 <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  
 <**fileNamePattern**>${LOG\_HOME}/${APP}\_access.log.%d{yyyyMMdd}</**fileNamePattern**>  
 </**rollingPolicy**>  
 </**appender**>  
  
  
 <**logger name="ACCESS"**>  
 <**appender-ref ref="ACCESS"** />  
 </**logger**>  
 <**logger name="druid.sql.Statement" level="DEBUG"** />  
 <**logger name="cn.ym.com.dao" level="TRACE"** />  
  
 <**root level="INFO"**>  
 <**appender-ref ref="DETAIL"** />  
 <**appender-ref ref="CONSOLE"** />  
 </**root**>  
</**configuration**>

1. 在项目中引入logback的依赖程序文件：

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-core</artifactId>

<version>1.2.3</version>

</dependency>

1. 在正常情况下mybatis中的日志信息的输出操作必须设置对应的命名空间，在logback.xml中追加如下信息：

<**logger name="cn.ym.com.dao" level="TRACE"** />

### 4、druid监控

druid数据库连接池之所以使用非常广泛，其最主要的原因在于他可以直接提供性能监控。

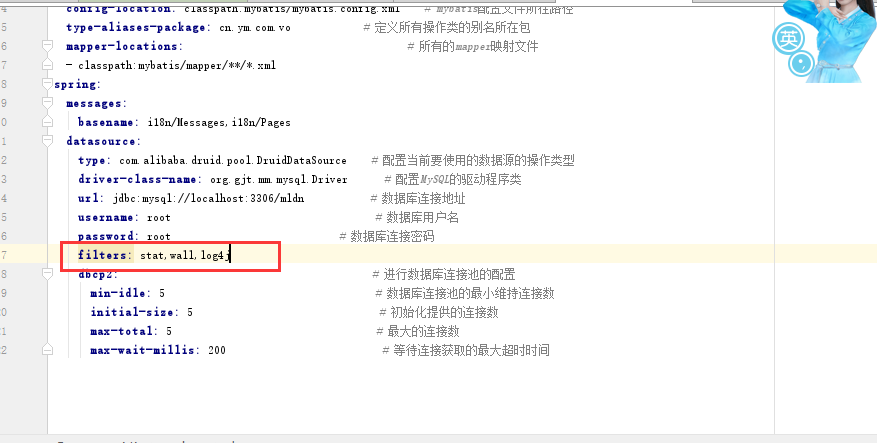
1、要想使用druid性能监控操作，则需要做一些基础配置，例如：访问的IP地址是否是白名单。

@Configuration  
**public class** DruidConfig {  
 @Bean  
 @ConfigurationProperties(prefix = **"spring.datasource"**) *//表示将在spring.datasource配置项后面追加属性配置* **public** DataSource druidDataSource() {  
  
 **return new** DruidDataSource();  
 }  
  
 */\*\*  
 \* // 主要实现WEB监控的配置处理  
 \*  
 \** ***@return*** *\*/* @Bean  
 **public** ServletRegistrationBean druidServlet() {  
 *// 现在要进行druid监控的配置处理操作* ServletRegistrationBean servletRegistrationBean = **new** ServletRegistrationBean(**new** StatViewServlet(), **"/druid/\*"**);  
 servletRegistrationBean.addInitParameter(**"allow"**, **"192.168.1.250"**); *//白名单* servletRegistrationBean.addInitParameter(**"deny"**, **"192.168.1.251"**); *//黑名单* servletRegistrationBean.addInitParameter(**"loginUsername"**, **"yangmi"**); *//控制台登录用户名* servletRegistrationBean.addInitParameter(**"loginPassword"**, **"ym"**);  
 servletRegistrationBean.addInitParameter(**"restEnable"**, **"false"**); *// 是否可以重置数据源* **return** servletRegistrationBean;  
 }  
  
 @Bean  
 **public** FilterRegistrationBean filterRegistrationBean() {  
 FilterRegistrationBean filterRegistrationBean = **new** FilterRegistrationBean();  
 filterRegistrationBean.setFilter(**new** WebStatFilter());  
 filterRegistrationBean.addUrlPatterns(**"/\*"**); *//所有请求进行监控处理  
 //访问静态资源文件和控制台不监控* filterRegistrationBean.addInitParameter(**"exclusions"**, **"\*.js,\*.gif,\*.jpg,\*.css,/druid/"**);  
 **return** filterRegistrationBean;  
 }  
}

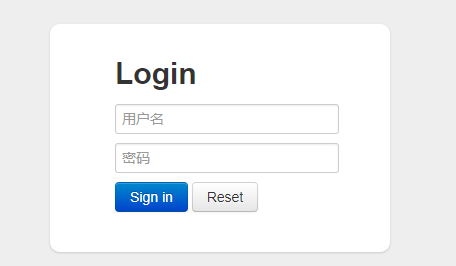
2、为了更好的说明问题，建议建立一个控制器进行业务层的调用：

@RestController  
**public class** DeptController {  
 @Resource  
 **private** IDeptService **deptService**;  
  
 @RequestMapping(value = **"/list"**,method = RequestMethod.***GET***)  
 **public** Object list(){  
 **return this**.**deptService**.list();  
 }  
}

3、开启一个过滤配置，这个过滤配置的开启需要通过application.yml文件配置;



1. 启动springboot项目，访问后台：<http://localhost:8080/druid/login.html>



6、访问http://localhost:8080/list,然后查看监控



## springboot整合消息服务组件

对于异步组件在实际的应用中会有两类：

1. JMS:代表作就是ActiveMQ，但是其性能不高，因为其是用java程序实现的；
2. AMQP:直接利用协议实现的消息组件，其代表作是rabbitmq,高性能代表作kafka。

### 整合ActiveMQ组件

1. 如果要想在项目中使用activeMQ组件，则应该为项目添加依赖包：

<dependency>

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

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

<version>1.5.14.RELEASE</version>

</dependency>

1. 修改application.yml配置文件进行activemq配置：

**server:  
 port:** 8080  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **jms:  
 pub-sub-domain:** false *# 配置消息的类型，如果是true则表示为topic消息，如果为false表示Queue消息* **activemq:  
 user:** admin *# 连接用户名* **password:** admin *# 连接密码* **broker-url:** tcp://172.20.10.5:61616 *# 消息组件的连接主机信息*

1. 随后定义一个消息的消费者，消费者主要是进行一个监听控制，在springboot里面可以利用注解进行监听：

@Service  
**public class** MessageConsumerService {  
 @JmsListener(destination = **"yangmi.msg.queue"**)  
 **public void** receiveMessage(String text) {*// 进行消息接收处理* System.***out***.println(**"【\*\*\*接收消息\*\*\*】"** + text);  
 }  
}

1. 随后建立消息的发送者服务，一般而言如果进行消息的发送，会准备出一个业务接口来：

**public interface** IMessageProducerService {  
 **void** sendMessage(String msg) ;  
}

1. 建立一个配置程序类，定义activeMQ的消息发送模板处理类：

@Configuration  
@EnableJms  
**public class** ActiveMQConfig {  
 @Bean  
 **public** Queue queue(){  
   
 **return new** ActiveMQQueue(**"yangmi.msg,queue"**);  
 }  
}

1. 创建消息发送的子类实现消息发送处理：

@Service  
**public class** MessageProducerServiceImpl **implements** IMessageProducerService {  
  
 *//此模板由spring自动配置注入* @Autowired  
 **private** JmsMessagingTemplate **jmsMessagingTemplate**;  
  
 *//注入ActiveMQConfig中配置的消息queue* @Resource  
 **private** Queue **queue**;  
  
 @Override  
 **public void** sendMessage(String msg) {  
 **this**.**jmsMessagingTemplate**.convertAndSend(**this**.**queue**, msg);  
 }  
}

1. 编写测试类来观察消息的处理：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestActiveMQ {  
 @Resource  
 **private** IMessageProducerService **messageProducerService**;  
  
 @Test  
 **public void** testSend() {  
 **for** (**int** x = 0; x < 100; x++) {  
 **this**.**messageProducerService**.sendMessage(**"杨幂"** + x);  
 }  
 }  
  
}

先启动项目，然后在执行测试类：

2018-06-21 13:29:13.900 INFO 5888 --- [ main] TestActiveMQ : Started TestActiveMQ in 7.629 seconds (JVM running for 9.159)

【\*\*\*接收消息\*\*\*】杨幂1

【\*\*\*接收消息\*\*\*】杨幂3

【\*\*\*接收消息\*\*\*】杨幂5

【\*\*\*接收消息\*\*\*】杨幂7

【\*\*\*接收消息\*\*\*】杨幂9

【\*\*\*接收消息\*\*\*】杨幂11

【\*\*\*接收消息\*\*\*】杨幂13

【\*\*\*接收消息\*\*\*】杨幂15

【\*\*\*接收消息\*\*\*】杨幂17

【\*\*\*接收消息\*\*\*】杨幂19

【\*\*\*接收消息\*\*\*】杨幂21

【\*\*\*接收消息\*\*\*】杨幂23

【\*\*\*接收消息\*\*\*】杨幂25

【\*\*\*接收消息\*\*\*】杨幂27

【\*\*\*接收消息\*\*\*】杨幂29

【\*\*\*接收消息\*\*\*】杨幂31

【\*\*\*接收消息\*\*\*】杨幂33

【\*\*\*接收消息\*\*\*】杨幂35

【\*\*\*接收消息\*\*\*】杨幂37

【\*\*\*接收消息\*\*\*】杨幂39

【\*\*\*接收消息\*\*\*】杨幂41

【\*\*\*接收消息\*\*\*】杨幂43

【\*\*\*接收消息\*\*\*】杨幂45

【\*\*\*接收消息\*\*\*】杨幂47

【\*\*\*接收消息\*\*\*】杨幂49

【\*\*\*接收消息\*\*\*】杨幂51

【\*\*\*接收消息\*\*\*】杨幂53

【\*\*\*接收消息\*\*\*】杨幂55

【\*\*\*接收消息\*\*\*】杨幂57

【\*\*\*接收消息\*\*\*】杨幂59

【\*\*\*接收消息\*\*\*】杨幂61

【\*\*\*接收消息\*\*\*】杨幂63

【\*\*\*接收消息\*\*\*】杨幂65

【\*\*\*接收消息\*\*\*】杨幂67

【\*\*\*接收消息\*\*\*】杨幂69

【\*\*\*接收消息\*\*\*】杨幂71

【\*\*\*接收消息\*\*\*】杨幂73

【\*\*\*接收消息\*\*\*】杨幂75

【\*\*\*接收消息\*\*\*】杨幂77

【\*\*\*接收消息\*\*\*】杨幂79

【\*\*\*接收消息\*\*\*】杨幂81

【\*\*\*接收消息\*\*\*】杨幂83

【\*\*\*接收消息\*\*\*】杨幂85

【\*\*\*接收消息\*\*\*】杨幂87

【\*\*\*接收消息\*\*\*】杨幂89

【\*\*\*接收消息\*\*\*】杨幂91

【\*\*\*接收消息\*\*\*】杨幂93

【\*\*\*接收消息\*\*\*】杨幂95

【\*\*\*接收消息\*\*\*】杨幂97

【\*\*\*接收消息\*\*\*】杨幂99

2018-06-21 13:29:16.355 INFO 5888 --- [ Thread-5] o.s.w.c.s.GenericWebApplicationContext : Closing org.springframework.web.context.support.GenericWebApplicationContext@43dac38f: startup date [Thu Jun 21 13:29:08 CST 2018]; root of context hierarchy

2018-06-21 13:29:16.357 INFO 5888 --- [ Thread-5] o.s.c.support.DefaultLifecycleProcessor : Stopping beans in phase 2147483647

### 整合RabbitMQ组件

在与RabbitMQ整合的时候需要注意一下几个概念：交换空间、虚拟主机、队列信息。为方便起见，本次将项目分为两个：RabbitMQ-Consumer、Rabbit-Producer。

1. 【microboot-rabbit-producer、consumer】将rabbitmq依赖支持包拷贝到项目之中；

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-amqp</**artifactId**>  
 <**version**>1.5.14.RELEASE</**version**>  
</**dependency**>

1. 【microboot-rabbit-producer、consumer】修改application.yml配置文件，追加rabbit的相关配置项：

**server:  
 port:** 8080  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **rabbitmq:  
 addresses:** 192.168.1.216  
 **username:** yangmi  
 **password:** ym  
 **virtual-host:** /

1. 【microboot-rabbit-producer】建立一个消息的发送接口：

**public interface** IMessageProducerService {  
 **public void** sendMessage(String msg) ;  
}

1. 【microboot-rabbit-producer】为了可以正常使用rabbitmq进行消息处理，还需要做一个消息生产的配置类：

@Configuration  
**public class** ProducerConfig {  
 **public static final** String ***EXCHANGE*** = **"yangmi.microboot.exchange"**; *// 交换空间名称* **public static final** String ***ROUTINGKEY*** = **"yangmi.microboot.routingkey"**; *// 设置路由key* **public static final** String ***QUEUE\_NAME*** = **"yangmi.microboot.queue"**; *// 队列名称* @Bean  
 **public** Binding bindingExchangeQueue(DirectExchange exchange, Queue queue) {  
 **return** BindingBuilder.*bind*(queue).to(exchange).with(***ROUTINGKEY***);  
 }  
  
 */\*\*  
 \* 配置交换机模式  
 \*  
 \** ***@return*** *\*/* @Bean  
 **public** DirectExchange getDirectExchange() {*// 使用直连的模式* **return new** DirectExchange(***EXCHANGE***, **true**, **true**);*//持久化，可自动删除* }  
  
 */\*\*  
 \* 配置队列信息  
 \*  
 \** ***@return*** *\*/* @Bean  
 **public** Queue queue() {  
 **return new** Queue(***QUEUE\_NAME***);  
 }  
}

1. 【microboot-rabbit-producer】创建消息服务实现子类：

@Service

**public class** MessageProducerServiceImpl **implements** IMessageProducerService {  
 @Resource  
 **private** RabbitTemplate **rabbitTemplate**;  
  
 @Override  
 **public void** sendMessage(String msg) {  
 **this**.**rabbitTemplate**.convertAndSend(ProducerConfig.***EXCHANGE***, ProducerConfig.***ROUTINGKEY***, msg);  
 }  
}

1. 【microboot-rabbit-consumer】依然需要做消费者的配置程序类，而这个程序类里面主要目的依然是设置交换空间、路由Key等信息。

@Configuration  
**public class** ConsumerConfig{  
 **public static final** String ***EXCHANGE*** = **"yangmi.microboot.exchange"**; *// 交换空间名称* **public static final** String ***ROUTINGKEY*** = **"yangmi.microboot.routingkey"**; *// 设置路由key* **public static final** String ***QUEUE\_NAME*** = **"yangmi.microboot.queue"**; *// 队列名称* @Bean  
 **public** Binding bindingExchangeQueue(DirectExchange exchange, Queue queue) {  
 **return** BindingBuilder.*bind*(queue).to(exchange).with(***ROUTINGKEY***);  
 }  
  
 */\*\*  
 \* 配置交换机模式  
 \*  
 \** ***@return*** *\*/* @Bean  
 **public** DirectExchange getDirectExchange() {*// 使用直连的模式* **return new** DirectExchange(***EXCHANGE***, **true**, **true**);*//持久化，可自动删除* }  
  
 */\*\*  
 \* 配置队列信息  
 \*  
 \** ***@return*** *\*/* @Bean  
 **public** Queue queue() {  
 **return new** Queue(***QUEUE\_NAME***);  
 }  
}

7、【microboot-rabbit-consumer】配置application.yml。

**server:  
 port:** 8080  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **rabbitmq:  
 addresses:** 192.168.1.216  
 **username:** yangmi  
 **password:** ym  
 **virtual-host:** /

8、【microboot-rabbit-consumer】实现监听处理类:

@Service  
**public class** MessageConsumerService {  
 @RabbitListener(queues = **"yangmi.microboot.queue"**)  
 **public void** receiveMessage(String text){*//进行消息接收处理* System.***err***.println(**"【\*\*\* 接收消息 \*\*\*】"** + text);  
 }  
}

9、【microboot-rabbit-producer】创建一个测试类实现消息的发送处理。

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestRabbitMQ {  
 @Resource  
 **private** IMessageProducerService messageProducerService;  
 @Test  
 **public void** testSend() **throws** Exception {  
 **for** (**int** x = 0; x < 100; x++) {  
 **this**.messageProducerService.sendMessage(**"yangmi - "** + x);  
 }  
 }  
}

10、【microboot-rabbit-consumer】编写消息接收测试类

### 3、整合kafka组件

Kafka是现在最好的开源消息组件，其仿照AMQP协议操作，而且处理的性能也是最高的。而本次使用已经配置好的kafka服务器，而且这台服务器上使用了kerberos认证，所以应该首先准备好一个jass配置文件；

1. 定义“kafka\_client\_jass.conf”配置文件：
2. 为方便观察，本次准备两个项目：生产者（microboot-kafka-producer）、消费者（microboot-kafka-consumer）。为这两个项目添加kafka配置支持：
3. 【microboot-kafka-consumer】修改application.yml配置文件，进行kafka配置编写：
4. 【microboot-kafka-consumer】建立一个Kafka的消息的消费程序类：
5. 【microboot-kafka-consumer】随后还需要修改Springboot的启动程序类，追加kerberos配置：
6. 【microboot-kafka-producer】修改application.yml配置文件：
7. 【microboot-kafka-producer】定义消息发送的服务接口：
8. 【microboot-kafka-producer】修改程序启动类：
9. 【microboot-kafka-producer】编写消息发送的测试类：
10. 启动消费者测试程序类，并通过生产者程序发送消息：

在使用kafka进行数据处理的时候一定要记住，它速度快的主要原因是采用的协议、处理的模式、零拷贝。

## 整合其他服务

### 邮件服务

要进行邮件的整合处理，那么硬要有一个邮件服务器，本次申请QQ的邮件服务，在QQ邮箱里面设置开启的服务。

1. 要实现邮件发送服务，则首先要导入一个依赖的支持库：

<dependency>

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

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

<version>1.5.14.RELEASE</version>

</dependency>

1. 修改application.yml配置文件，实现邮件的配置:

**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **mail:  
 host:** smtp.qq.com  
 **username:** 你自己的用户名，110@qq.com  
 **password:** 生成的授权码  
 **properties:  
 mail.smtp.auth:** true  
 **mail.smtp.starttls.enable:** true  
 **mail.smtp.starttls.requ**

1. 编写测试类，直接调用mail的服务：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestMail {  
 @Resource  
 **private** JavaMailSender **javaMailSender**;  
  
 @Test  
 **public void** testSendMail() {  
 SimpleMailMessage message = **new** SimpleMailMessage();*// 要发送的消息内容* message.setFrom(**"963390654@qq.com"**);  
 message.setTo(**"1423678754@qq.com"**);  
 message.setSubject(**"测试！"**);  
 message.setText(**"1111"**);  
 **this**.**javaMailSender**.send(message);  
 }  
}

### 定时调度

对于定时调度，在实际实际开发中可以使用：timertask、quartz、springtask配置，实际上这里面最简单的配置就是Spring自己提供的task处理。

1. 如果要想实现定时调度，只需要配置一个定时调度组件类即可：

@Component  
**public class** MyScheduler {  
 @Scheduled(fixedRate = 2000) *// 采用间隔调度，每2秒执行一次* **public void** runJobA() {  
 *// 定义一个要执行的任务* **try** {  
 Thread.*sleep*(5000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"【\*\*\* MyTaskA - 间隔调度 \*\*\*】"** + **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**)  
 .format(**new** Date()));  
 }  
  
 @Scheduled(cron = **"\* \* \* \* \* ?"**) *// 每秒调用一次* **public void** runJobB() {  
 System.***err***.println(**"【\*\*\* MyTaskB - 间隔调度 \*\*\*】"** + **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**)  
 .format(**new** Date()));  
 }  
}

1. 如果现在想要执行此任务，那么还需要有一个基本前提：你的程序启动类上一定要启用调度处理。

@SpringBootApplication *// 启动SpringBoot程序，而后自带子包扫描*@EnableScheduling *// 启用定时调度***public class** StartSpringBootMain {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(StartSpringBootMain.**class**, args);  
 }  
}

这个时候所实现的任务调度只是串行任务调度，也就是说所有的任务一个一个的执行。

1. 如果要进行并行的调度处理，则一定要准备出一个线程调度池：

@Configuration *// 定时调度的配置类一定要实现指定的父接口***public class** SchedulerConfig **implements** SchedulingConfigurer {  
 @Override  
 **public void** configureTasks(ScheduledTaskRegistrar scheduledTaskRegistrar) {  
 *// 开启一个线程调度池* scheduledTaskRegistrar.setScheduler(Executors.*newScheduledThreadPool*(200));  
 }  
}

### 3、Actuator监控

对于springboot中服务的监控在springboot开发框架之中提供有一个actuator监控程序，但是这个监控程序在好多时候使用有些矛盾，对于此类的信息服务，一般而言有两类信息：

1. 第一类是由系统直接提供好的信息，这个需要关闭安全控制；
2. 第二类是由用户定义的，希望可以返回固定的一些信息，例如：版本、服务名称等。



1. 如果现在想要在代码之中启用actuator服务，那么首先一定要进行开发包的配置；

<dependency>

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

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

<version>2.0.3.RELEASE</version>

</dependency>

1. 此时配置此依赖包就具有了监控的操作能力，但是这个能力无法直接体现。如果想打开系统的这些默认信息，则需要修改application.yml配置文件，关闭掉当前的安全配置项：

**server:  
 port:** 8080  
**management:  
 security:  
 enabled:** false *#关闭系统安全配置*

<http://localhost:8080/health>

1. 但是此时将系统的安全配置关闭之后，那么此时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，同时也希望所有人可以看见一些能够看见的提示信息，所以在这样的状态下，就需要去考虑自定义一些服务信息出现。例如现在先进行健康信息的配置；

定义一个健康配置类：

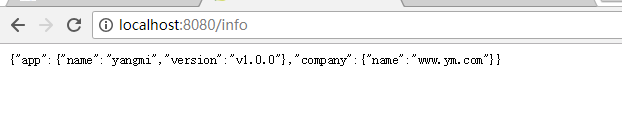
@Component  
**public class** MyHealthIndicator **implements** HealthIndicator {  
 @Override  
 **public** Health health() {  
 **return** Health.*down*().withDetail(**"info"**, **"老子还活着"**).build();  
 }  
}

但是这种健康信息的配置是需要关闭安全策略后才能够发现一些更加详细的内容，而如果开启了安全策略后可以看见的信息只能够是一些基础的状态：UP(存活)、DOWN(关闭)。

@Component  
**public class** MyHealthIndicator **implements** HealthIndicator {  
 @Override  
 **public** Health health() {  
 **int** errorCode = 100; *//这个错误码，可以通过其他程序获得* **if** (errorCode != 0) {  
 **return** Health.*down*().withDetail(**"error code "**, errorCode).build();  
 }  
 **return** Health.*up*().withDetail(**"info"**, **"老子还活着"**).build();  
 }  
}

1. 除了健康之外，还有一项数据很重要，就是一些服务信息（/info）,而这些服务信息可以直接通过application.yml文件配置；

**server:  
 port:** 8080  
**management:  
 security:  
 enabled:** false *#关闭系统安全配置***info:  
 app.name:** yangmi  
 **app.version:** v1.0.0  
 **company.name:** www.ym.com



1. 对于这样的配置信息最好的方式就是进行pom.xml文件的读取，所以如果要想实现这种关联操作，则必须修改microboot项目中的pom.xml文件，追加有一个访问的插件：

<**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-resources-plugin</**artifactId**>  
 <**version**>3.1.0</**version**>  
 <**configuration**>  
 <**delimiters**>  
 <**delimiter**>$</**delimiter**>  
 </**delimiters**>  
 </**configuration**>  
</**plugin**>

用$分割信息。

1. 修改filtering为true:

<**resource**>  
 <**directory**>src/main/resources</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.properties</**include**>  
 <**include**>\*\*/\*.yml</**include**>  
 <**include**>\*\*/\*.xml</**include**>  
 <**include**>\*\*/\*.tld</**include**>  
 <**include**>\*\*/\*.conf</**include**>  
 </**includes**>  
 <**filtering**>true</**filtering**>  
</**resource**>

1. 修改application.yml配置文件，追加详细信息

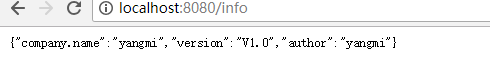
**server:  
 port:** 8080  
**management:  
 security:  
 enabled:** false *#关闭系统安全配置***info:  
 app.name:** yangmi  
 **app.version:** v1.0.0  
 **company.name:** www.ym.com  
 **pom.artifactId:** $project.artifactId$  
 **pom.version:** $project.version$

用$分割信息。



1. 由于在开发中这中提示信息会成为微服务的重要组成部分，所以很多时候如果重复配置定义，那就比较麻烦，最简单的做法是直接做一个配置程序类进行信息的配置。

@Component  
**public class** ProjectInfoContributor **implements** InfoContributor{  
 @Override  
 **public void** contribute(Info.Builder builder) {  
 builder.withDetail(**"company.name"**, **"yangmi"**) ;  
 builder.withDetail(**"version"**, **"V1.0"**) ;  
 builder.withDetail(**"author"**, **"yangmi"**) ;  
 }  
}



## springboot整合redis数据库

### redisTemplate操作

在spring支持的redis操作中提供有一个redisTemplate处理程序类，利用此类可以轻松的完成redis的各种数据的基本操作。

1. 修改项目的pom.xml配置文件，追加redis的依赖引用：

<dependency>

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

<artifactId>spring-boot-starter-data-redis</artifactId>

<version>2.0.3.RELEASE</version>

</dependency>

1. 修改application.yml配置文件，进行redis各种连接处理：

**server:  
 port:** 8080  
**spring:  
 redis:  
 host:** 192.168.1.241  
 **port:** 6379  
 **password:** 12345  
 **database:** 0  
 **timeout:** 3000  
 **pool:  
 max-active:** 10  
 **max-idle:** 8  
 **min-idle:** 2  
 **max-wait:** 100

1. 下面就可以通过程序来利用redisTemplate模板进行数据处理了，因为以上的配置一旦完成会在spring内部帮助用户直接获得一个redisTemplate模板处理对象。

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestRedis {  
 @Resource  
 **private** RedisTemplate<String, String> **redisTemplate**;  
  
 @Test  
 **public void** testSet() {  
 **this**.**redisTemplate**.opsForValue().set(**"yangmi"**, **"杨幂"**);  
 System.***out***.println(**this**.**redisTemplate**.opsForValue().get(**"yangmi"**));  
 }  
}

此时就可以利用redis实现springboot中的数据存储操作了。

### redis序列化处理

1. 准备一个序列化处理程序类，这个程序类有实现要求：

*//此时定义的序列化操作表示所有类的对象，当然，这个对象所在的类一定要实现序列化接口***public class** RedisObjectSerializer **implements** RedisSerializer<Object> {  
 *// 为了方便进行对象与字节数组的转换，所以应该首先准备出两个转换器* **private** Converter<Object, **byte**[]> **serializingConverter** = **new** SerializingConverter();  
 **private** Converter<**byte**[], Object> **deserializingConverter** = **new** DeserializingConverter();  
  
 @Override  
 **public byte**[] serialize(Object o) **throws** SerializationException {  
 **if** (**null** != o) {  
 *// 将对象变为字节数组* **this**.**serializingConverter**.convert(o);  
 }  
 **return new byte**[0];  
 }  
  
 @Override  
 **public** Object deserialize(**byte**[] bytes) **throws** SerializationException {  
 *// 此时没有对象的内容信息* **if** (bytes == **null** || bytes.**length** == 0) {  
 **return null**;  
 }  
 **return this**.**deserializingConverter**.convert(bytes);  
 }  
}

1. 此时如果想要RedisTemplate模板知道存在有这样的一个序列化类存在，就不能再采用redistemplate默认配置形式，需要准备一个单独的配置类进行处理：

@Configuration  
**public class** RedisConfig {  
 **public** RedisTemplate<String, Object> getRedisTemplate(RedisConnectionFactory factory) {  
 RedisTemplate<String, Object> redisTemplate = **new** RedisTemplate<String, Object>();  
 redisTemplate.setConnectionFactory(factory);  
 redisTemplate.setKeySerializer(**new** StringRedisSerializer()); *// key的序列化类型* redisTemplate.setValueSerializer(**new** RedisObjectSerializer()); *// value的序列化类型* **return** redisTemplate;  
 }  
}

1. 准备一个Member类

**public class** Member **implements** Serializable {  
 **private** String **mid**;  
 **private** Integer **age**;  
  
 **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;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Member [mid="** + **mid** + **", age="** + **age** + **"]"**;  
 }  
}

1. 编写测试类

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestRedis {  
 @Resource  
 **private** RedisTemplate<String, Object> **redisTemplate**;  
  
 @Test  
 **public void** testSet() {  
 Member vo = **new** Member();  
 vo.setMid(**"1"**);  
 vo.setAge(30);  
 **this**.**redisTemplate**.opsForValue().set(**"yangmi"**, vo);  
 }  
  
 @Test  
 **public void** testGet() {  
 System.***out***.println(**this**.**redisTemplate**.opsForValue().get(**"yangmi"**));  
 }  
}

### 配置多个redis连接

由于在项目开发中redis的使用会非常的频繁，那么就有可能出现这样的情况：在项目里面要求连接两个redis数据库。springboot里面针对于redis的连接配置本质上只提供有一个连接配置项，那么如果要实现多个redis连接配置，那么就需要进行redis的创建管理。

1. 非正规的配置修改application.yml配置文件：

**server:  
 port:** 8080  
**spring:  
 redis:  
 host:** 192.168.1.241  
 **port:** 6379  
 **password:** 12345  
 **database:** 0  
 **timeout:** 3000  
 **pool:  
 max-active:** 10  
 **max-idle:** 8  
 **min-idle:** 2  
 **max-wait:** 100  
 **redis-6380:  
 host:** 192.168.1.241  
 **port:** 6380  
 **password:** 12345  
 **timeout:** 3000  
 **database:** 0  
 **pool:  
 max-active:** 10  
 **max-idle:** 8  
 **min-idle:** 2  
 **max-wait:** 100

1. 建立一个redis的程序配置类，进行第二个redis连接的配置处理：

@Configuration  
**public class** RedisTwoConfig {  
  
 */\*\*  
 \* 获取自定义redis连接工厂  
 \*  
 \** ***@param hostName*** *\** ***@param password*** *\** ***@param port*** *\** ***@param maxActive*** *\** ***@param maxIdle*** *\** ***@param minIdle*** *\** ***@param maxWait*** *\** ***@param database*** *\** ***@return*** *\*/* **public** RedisConnectionFactory getRedisConnectionFactory(String hostName, String password, **int** port,  
 **int** maxActive, **int** maxIdle, **int** minIdle,  
 **long** maxWait, **int** database) {  
 *// 是负责建立Factory的连接工厂类* JedisConnectionFactory jedisConnectionFactory = **new** JedisConnectionFactory();  
 jedisConnectionFactory.setHostName(hostName);  
 jedisConnectionFactory.setPort(port);  
 jedisConnectionFactory.setPassword(password);  
 jedisConnectionFactory.setDatabase(database);  
  
 *// 进行连接池配置* JedisPoolConfig jedisPoolConfig = **new** JedisPoolConfig();  
 JedisPoolConfig poolConfig = **new** JedisPoolConfig(); *// 进行连接池配置* poolConfig.setMaxTotal(maxActive);  
 poolConfig.setMaxIdle(maxIdle);  
 poolConfig.setMinIdle(minIdle);  
 poolConfig.setMaxWaitMillis(maxWait);  
 jedisConnectionFactory.setPoolConfig(poolConfig);  
  
 *// 初始化连接池配置* jedisConnectionFactory.afterPropertiesSet();  
 **return** jedisConnectionFactory;  
 }  
  
 */\*\*  
 \* 获取spring配置文件中属性信息  
 \*  
 \** ***@param hostName*** *\** ***@param password*** *\** ***@param port*** *\** ***@param database*** *\** ***@param maxActive*** *\** ***@param maxIdle*** *\** ***@param minIdle*** *\** ***@param maxWait*** *\** ***@return*** *\*/* @Bean(**"redis-6380"**) *//按名称注入* **public** RedisTemplate<String, Object> getRedisTemplate(@Value(**"${spring.redis-6380.host}"**) String hostName,  
 @Value(**"${spring.redis-6380.password}"**) String password,  
 @Value(**"${spring.redis-6380.port}"**) **int** port,  
 @Value(**"${spring.redis-6380.database}"**) **int** database,  
 @Value(**"${spring.redis-6380.pool.max-active}"**) **int** maxActive,  
 @Value(**"${spring.redis-6380.pool.max-idle}"**) **int** maxIdle,  
 @Value(**"${spring.redis-6380.pool.min-idle}"**) **int** minIdle,  
 @Value(**"${}spring.redis-6380.pool.max-wait}"**) **long** maxWait) {  
 *// 建立Redis的连接* RedisConnectionFactory factory = **this**.getRedisConnectionFactory(hostName, password, port, maxActive, maxIdle, minIdle, maxWait,  
 database);  
 RedisTemplate<String, Object> redisTemplate = **new** RedisTemplate<String, Object>();  
 redisTemplate.setConnectionFactory(factory);  
 redisTemplate.setKeySerializer(**new** StringRedisSerializer()); *// key的序列化类型* redisTemplate.setValueSerializer(**new** RedisObjectSerializer()); *//value的序列化类型* **return** redisTemplate;  
 }  
}

1. 编写测试类

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestRedisTwo {  
 @Resource(name = **"redis-6380"**)  
 **private** RedisTemplate<String, Object> **redisTemplate**;  
  
 @Test  
 **public void** testGet() {  
 System.***out***.println(**this**.**redisTemplate**.opsForValue().get(**"yangmi"**));  
 }  
  
 @Test  
 **public void** testSet() {  
 Member vo = **new** Member();  
 vo.setMid(**"yangmi"**);  
 vo.setAge(29);  
 **this**.**redisTemplate**.opsForValue().set(**"yangmi"**, vo);  
 }  
}

## springboot整合restful

spring与restful整合才是微架构的核心，虽然在整个springboot（springCloud也一样）之中提供有大量的服务方便整合，但是这些整合都不如rest重要，因为整个微架构中，进行通讯的基础模式。对于rest首先必须对其有一个最为核心的解释：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，利用此模板可以非常轻松的实现rest的json数据与各种对象间的自动转换。

在默认情况下Spring里面针对于Rest的处理模式使用的都是jackson开发支持。

### 使用RestTemplate调用Rest服务

由于Rest属于分布式的项目开发环境，所以本次进行项目建立的时候一共建立有三个子模块：

1. microboot-restful-api:作为公共的类定义，例如：可以将所有的vo类定义子在此项目
2. microboot-restful-provider:作为服务提供者，这次的服务提供者提供有两个服务（获得对象、增加对象）
3. microboot-restful-consumer:作为服务的消费者，消费者就是利用RestTemplate实现Rest服务的调用以及对象的转换。
4. 【microboot-restful-api】建立一个公共的VO类对象：

**public class** Member **implements** Serializable {  
 **private** Long **mid**;  
 **private** String **name**;  
 **private** Integer **age**;  
 **private** Double **salary**;  
 **private** Date **birthday**;  
  
 **public** Long getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(Long mid) {  
 **this**.**mid** = mid;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **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** + **", name="** + **name** + **", age="** + **age** + **", salary="** + **salary** + **", birthday="** + **birthday** + **"]"**;  
 }  
}

2、【microboot-restful-provider】修改pom.xml配置文件，去引用api模块：

<**dependency**>  
 <**artifactId**>microboot-restful-api</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>

3、【microboot-restful-provider】建立一个控制器实现rest服务的处理：

@RestController  
**public class** MemberController {  
 @RequestMapping(value = **"/member/add"**, method = RequestMethod.***POST***)  
 **public** Object add(Member member) {*// 表示当前的配置可以直接将参数变为VO对象* System.***out***.println(**"【MemberController.add()接收对象】"** + member);  
 **return true**;  
 }  
  
 @RequestMapping(value = **"/member/get"**, method = RequestMethod.***GET***)  
 **public** Member get(**long** mid) {  
 Member vo = **new** Member();  
 vo.setMid(mid);  
 vo.setName(**"yangmi"** + mid);  
 vo.setBirthday(**new** Date());  
 vo.setSalary(99999.99);  
 vo.setAge(16);  
 **return** vo;  
 }  
}

1. 【microboot-restful-provider】定义程序启动类，启动服务，而后测试当前程序是否可用：
2. 获取对象信息：<http://localhost:8080/member/get?mid=100>
3. 增加对象信息：

http://localhost:8080/member/add?mid=100&name=杨幂&age=16&salary=1000000&birthday=2018-10-10

1. 【microboot-restful-consumer】修改pom.xml配置文件，去引用api模块：

<**dependency**>  
 <**artifactId**>microboot-restful-api</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>

1. 【microboot-restful-consumer】如果要进行rest操作，那么一定要注意使用一个RestTemplate模板完成处理。建立一个程序配置类，进行RedisTemplate模板对象创建：

@Configuration  
**public class** RestConfig {  
 @Bean  
 **public** RestTemplate getRestTemplate() {  
  
 **return new** RestTemplate();  
 }  
}

1. 【microboot-restful-consumer】修改application.yml配置端口：

**server:  
 port:** 8088

1. 【microboot-restful-consumer】编写测试程序类测试远程Rest服务是否可用：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestMemberRestful {  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @Test  
 **public void** testAdd() {  
 **boolean** flag = **this**.**restTemplate**.getForObject(**"http://localhost:8080/member/add?mid=100&name=杨幂&age=16&salary=1000000&birthday=2018-10-10"**, Boolean.**class**);  
 System.***out***.println(**"【ConsumerTest.add()】"** + flag);  
 }  
  
 @Test  
 **public void** testGet() {  
 Member member = **this**.**restTemplate**.getForObject(**"http://localhost:8080/member/get?mid=110"**, Member.**class**);  
 System.***out***.println(**"【ConsumerTest.get()】"** + member);  
 }  
}

1. 【microboot-restful-provider】为了更方便的进行内容的传输，此时rest服务的提供方一定要做出修改：

@RestController  
**public class** MemberController **extends** AbstractBaseController {  
 @RequestMapping(value = **"/member/add"**, method = RequestMethod.***POST***)  
 **public** Object add(@RequestBody Member member) {*// 表示当前的配置可以直接将参数变为VO对象* System.***out***.println(**"【MemberController.add()接收对象】"** + member);  
 **return true**;  
 }  
  
 @RequestMapping(value = **"/member/get/{mid}"**, method = RequestMethod.***GET***)  
 **public** Member get(@PathVariable(**"mid"**) **long** mid) {  
 Member vo = **new** Member();  
 vo.setMid(mid);  
 vo.setName(**"yangmi"** + mid);  
 vo.setBirthday(**new** Date());  
 vo.setSalary(99999.99);  
 vo.setAge(16);  
 **return** vo;  
 }  
}

1. 【microboot-restful-consumer】编写一个调用的控制器进行处理：

@Controller

**public class** MemberConsumerController **extends** AbstractBaseController {  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @RequestMapping(value = **"/consumer/get"**, method = RequestMethod.***GET***)  
 **public** String getMember(**long** mid, Model model) {  
 Member member = **this**.**restTemplate**.getForObject(**"http://localhost:8080/member/get/"** + mid, Member.**class**);  
 model.addAttribute(**"member"**, member);  
 **return "member\_show"**;  
 }  
  
 @RequestMapping(value = **"/consumer/add"**, method = RequestMethod.***GET***)  
 @ResponseBody  
 **public** Object addMember(Member member) {  
 Boolean flag = **this**.**restTemplate**.postForObject(**"http://localhost:8080/member/add"**, member, Boolean.**class**);  
 **return false**;  
 }  
}

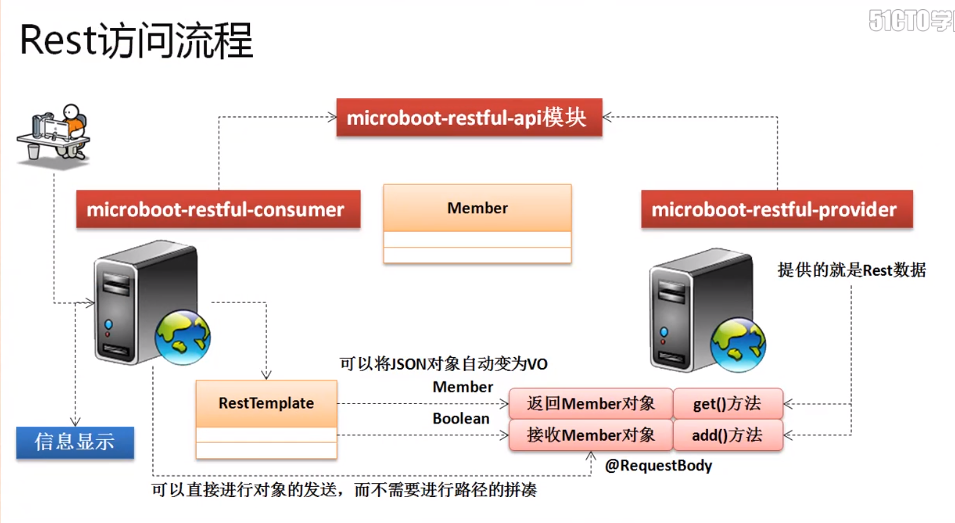
1. 【microboot-restful-consumer】为了方便进行接收数据的演示，建立一个普通的thymeleaf页面：

<!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="'用户编号：' + ${member.mid}"**/>  
 <**p th:text="'用户姓名：' + ${member.name}"**/>  
 <**p th:text="'用户年龄：' + ${member.age}"**/>  
 <**p th:text="'用户工资：' + ${member.salary}"**/>  
 <**p th:text="'用户生日：' + ${member.birthday}"**/>  
</**body**>  
</**html**>

1. 【microboot-restful-consumer】访问消费端服务：

<http://localhost:8088/consumer/get?mid=100>

1. 现在在整个的项目之中会发现以下几个特点：
2. rest服务的生产者只是按照自己返回的内容进行JSON数据的输出
3. 消费者利用RestTemplate进行JSON数据的获得以及自动向指定类型的对象转换
4. 为了达到这种歌转换的操作标准，特意准备了一个api项目保存的VO类型



而对于rest服务的更多考虑应该包含如下几点：

1. 既然服务提供者只能够被消费者访问，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. 在服务端访问非常繁忙的时候，消费端（客户端）执行时有可能需要进行短期的熔断处理；
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. 消费端进行服务端的调用操作，如果所有的调用都写上明确地址，态麻烦了；
6. 消费端进行处理的时候都是自己来直接采用RestTemplate做处理，代码结构太差，因为毕竟服务端是远程业务端，远程业务端的调用应该就采用接口完成。

### 配置swagger服务

当建立一些公共的rest服务的时候就可以利用Swagger进行所有rest服务的描述。它提供的只是一个说明工具的概念。

1. 要使用swagger说明操作，则必须引入相应的依赖支持包：

<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger2</**artifactId**>  
 <**version**>2.9.2</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger-ui</**artifactId**>  
 <**version**>2.9.2</**version**>  
</**dependency**>

1. 定义一个swagger2的配置程序类：

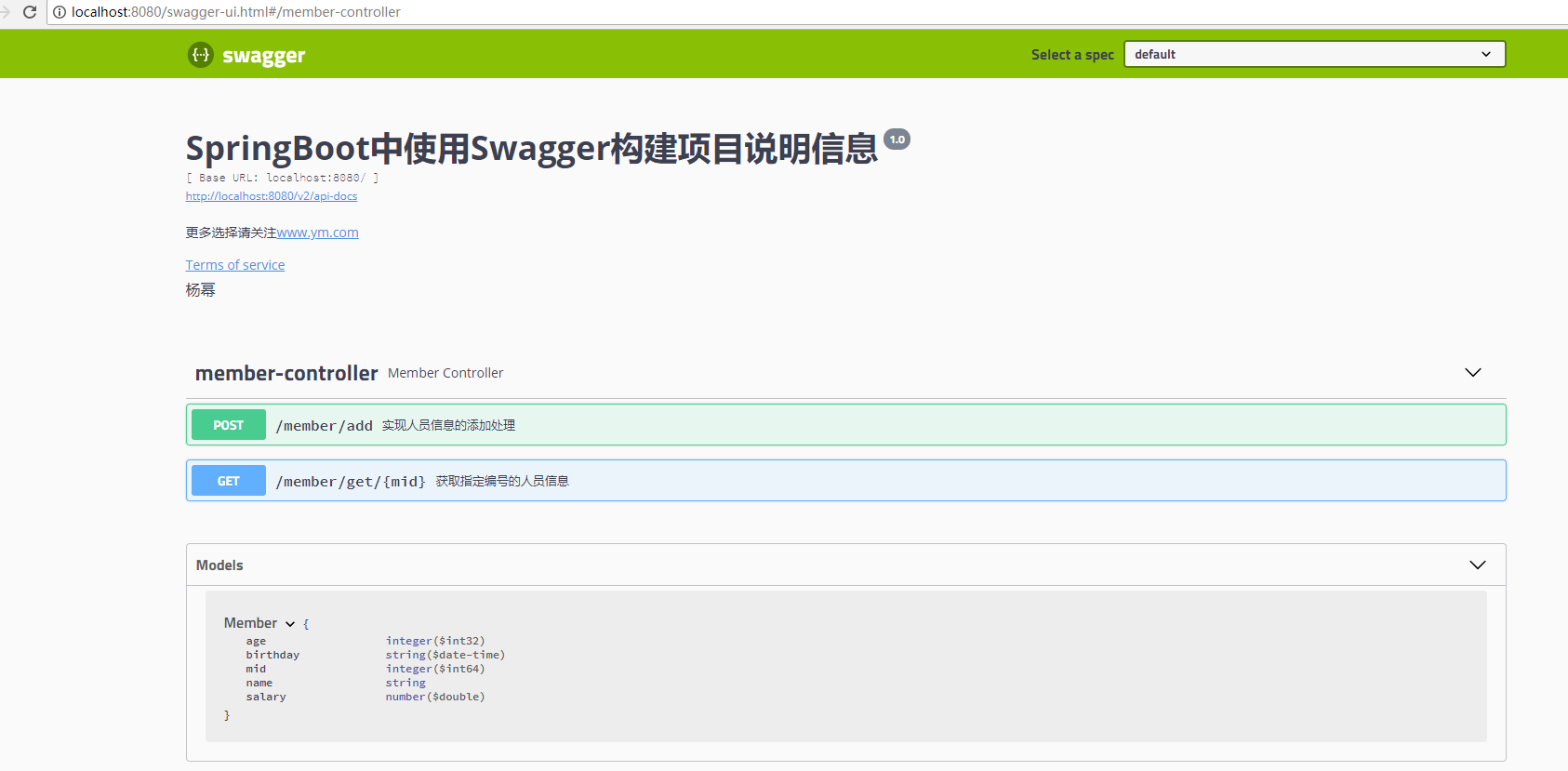
@Configuration  
@EnableSwagger2  
**public class** Swagger2Config { *// 此类主要是整个的Swagger配置项，利用这个类需要来指派扫描包* @Bean  
 **public** Docket getDocket() {  
  
 **return new** Docket(DocumentationType.***SWAGGER\_2***).apiInfo(**this**.getApiInfo()).select()  
 .apis(RequestHandlerSelectors.*basePackage*(**"cn.ym.com.controller"**))  
 .paths(PathSelectors.*any*()).build();*// 设置文档的显示类型* }  
  
 **private** ApiInfo getApiInfo() {  
 **return new** ApiInfoBuilder().title(**"SpringBoot中使用Swagger构建项目说明信息"**).description(**"更多选择请关注www.ym.com"**)  
 .termsOfServiceUrl(**"http://www.ym.com"**).contact(**"杨幂小姐姐"**)  
 .license(**"杨幂"**).version(**"1.0"**).build();  
 }  
}

1. 修改MemberControllter程序类：

@RestController  
**public class** MemberController {  
  
 @ApiOperation(value = **"实现人员信息的添加处理"**, notes = **"就是加人的，多么的简单"**)  
 @ApiImplicitParams({  
 @ApiImplicitParam(name = **"member"**, value = **"用户描述的详细实体信息"**, required = **true**, dataType = **"MemberClass"**)  
 })  
 @RequestMapping(value = **"/member/add"**, method = RequestMethod.***POST***)  
 **public** Object add(@RequestBody Member member) { *// 表示当前的配置可以直接将参数变为VO对象* System.***err***.println(**"【MemberController.add()接收对象】"** + member);  
 **return true**;  
 }  
  
 @ApiOperation(value = **"获取指定编号的人员信息"**, notes = **"只需要设置mid的信息就可以获取Member的完整内容"**)  
 @ApiImplicitParams({  
 @ApiImplicitParam(name = **"mid"**, value = **"用户编号"**, required = **true**, dataType = **"String"**)})  
 @RequestMapping(value = **"/member/get/{mid}"**, method = RequestMethod.***GET***)  
 **public** Member get(@PathVariable(**"mid"**) **long** mid) {  
 Member vo = **new** Member();  
 vo.setMid(mid);  
 vo.setName(**"mldnjava - "** + mid);  
 vo.setBirthday(**new** Date());  
 vo.setSalary(99999.99);  
 vo.setAge(16);  
 **return** vo;  
 }  
}

1. 正常进行程序的启动配置处理，然后进入浏览器界面：

<http://localhost:8080/swagger-ui.html>



### 3、动态修改日志级别

在项目开发中日志可以使用info()、error()进行输出，在springboot中用户可以通过远程的控制追加日志的显示级别的操作。

1. 定义一个简单的控制器程序：

@RestController  
**public class** MessageController {  
 **private** Logger **log** = LoggerFactory.*getLogger*(MessageController.**class**);  
  
 @RequestMapping(value = **"/test"**)  
 **public** Object test() {  
 **this**.**log**.info(**"【\*\*\* INFO \*\*\*】日志输出"**);  
 **this**.**log**.error(**"【\*\*\* ERROR \*\*\*】日志输出"**);  
 **return true**;  
 }  
}

访问：<http://localhost:8080/test>



1. 如果现在希望只进行error级别的日志输出，则修改application.yml的配置文件：

**logging:  
 level:  
 cn.ym.com.controller:** ERROR

再次访问：<http://localhost:8080/test>



1. 如果希望可以在以后的运行的时候这个日志的输出级别可以动态的做一个扩充，则必须进行安全的关闭操作：
2. 追加依赖包：

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

1. application.yml追加安全关闭配置：

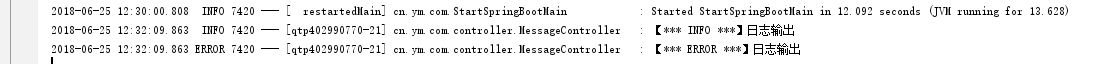
**management:  
 security:  
 enabled:** false

1. 随后在一个客户端上进行修改，直接利用测试类完成。
2. 新建日志类：

**public class** LogInfo {  
 **private** String **level**;  
  
 **public** String getLevel() {  
 **return level**;  
 }  
  
 **public void** setLevel(String level) {  
 **this**.**level** = level;  
 }  
}

1. 编写测试类：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestLogger {  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @Test  
 **public void** testLevel() {  
 LogInfo log = **new** LogInfo();  
 log.setLevel(**"INFO"**); *// 新的日志级别* **this**.**restTemplate**.postForLocation(**"http://localhost:8080/loggers/cn.ym.com.controller"**, log);  
 }  
}



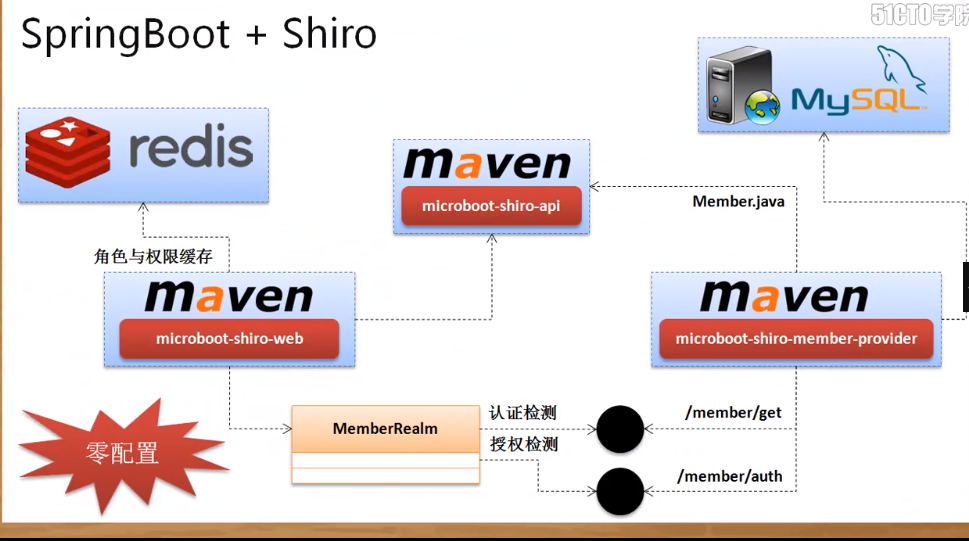
## springboot整合shiro验证框架

### 整合说明与环境配置

在整个的shiro之中最为重要的部分：认证以及授权处理（Realm）,在realm里面实际上在开发中所需要调用的业务只有两类：根据用户编号取得用户完整信息，在认证通过之后根据用户编号获得用户对应的所有角色以及权限信息，而且到了微架构阶段，不得不去面对一个问题，对于这种用户的业务操作是放在WEB端还是单独提出来做出一个rest服务？很明显，应该作为一个服务进行抽象出来，也就是说在整体的调用处理之中，realm需要进行rest服务调用（RestTemplate可以让整个调用更加容易）。

按照如上的设计方案，现在的整体的项目里面认为应该包含有如下的几个开发模块：

1. microboot-shiro-api：应该共有服务的vo类、各种加密处理的工具类；
2. microboot-shiro-member-provider：进行用户认证与授权REST服务的提供，要暴露两个接口：用户信息获得、角色与权限信息获得；
3. microboot-shiro-web：主要进行shiro的认证与授权检测处理



1. 创建数据库脚本：

-- 删除数据库

DROP DATABASE IF EXISTS mldn ;

-- 创建数据库

CREATE DATABASE mldn CHARACTER SET UTF8 ;

-- 使用数据库

USE mldn ;

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

password VARCHAR(32) ,

locked INT ,

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

CREATE TABLE role (

rid VARCHAR(50) ,

title VARCHAR(50) ,

CONSTRAINT pk\_rid PRIMARY KEY(rid)

) ;

CREATE TABLE action (

actid VARCHAR(50) ,

title VARCHAR(50) ,

rid VARCHAR(50) ,

CONSTRAINT pk\_actid PRIMARY KEY(actid)

) ;

CREATE TABLE member\_role (

mid VARCHAR(50) ,

rid VARCHAR(50)

) ;

INSERT INTO member(mid,name,password,locked) VALUES ('mldnjava','mldn','2E866BF58289E01583AD418F486A69DF',0) ;

INSERT INTO member(mid,name,password,locked) VALUES ('admin','admin','2E866BF58289E01583AD418F486A69DF',0) ;

INSERT INTO role(rid,title) VALUES ('emp','雇员管理') ;

INSERT INTO role(rid,title) VALUES ('dept','部门管理') ;

INSERT INTO action(actid,title,rid) VALUES ('emp:add','雇员入职','emp') ;

INSERT INTO action(actid,title,rid) VALUES ('emp:remove','雇员离职','emp') ;

INSERT INTO action(actid,title,rid) VALUES ('emp:list','雇员列表','emp') ;

INSERT INTO action(actid,title,rid) VALUES ('emp:edit','雇员编辑','emp') ;

INSERT INTO action(actid,title,rid) VALUES ('dept:list','部门列表','dept') ;

INSERT INTO action(actid,title,rid) VALUES ('dept:edit','部门编辑','dept') ;

INSERT INTO member\_role(mid,rid) VALUES ('mldnjava','emp') ;

INSERT INTO member\_role(mid,rid) VALUES ('admin','emp') ;

INSERT INTO member\_role(mid,rid) VALUES ('admin','dept') ;

1. 【microboot-shiro-api】建立一个Member程序类，保存认证返回的信息：
2. shuro进行认证处理的时候要求根据一个用户的编号获得对应的完整信息，而后在进行用户是否存在的判断、密码是否正确的判断、是否被锁定的判断。

**public class** Member **implements** Serializable {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** String **password**;  
 **private** Integer **locked**;  
  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** String getPassword() {  
 **return password**;  
 }  
  
 **public void** setPassword(String password) {  
 **this**.**password** = password;  
 }  
  
 **public** Integer getLocked() {  
 **return locked**;  
 }  
  
 **public void** setLocked(Integer locked) {  
 **this**.**locked** = locked;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Member [mid="** + **mid** + **", name="** + **name** + **", password="** + **password** + **", locked="** + **locked** + **"]"**;  
 }  
}

3、【microboot-shiro-api】密码加密处理，将base64和MD5程序发到程序中：

**package** cn.ym.com.util.enctype;  
  
**import** java.util.Base64;  
  
**public class** PasswordUtil {  
 **private static final** String ***SEED*** = **"mldnjava"** ; *// 该数据为种子数，如果要加密则需要使用Base64做多次迭代* **private static final int *NE\_NUM*** = 3 ; *// 密码迭代处理3次* **private** PasswordUtil() {}  
 **private static** String createSeed() { *// 创建一个基于Base64的种子数* String str = ***SEED*** ;  
 **for** (**int** x = 0 ; x < ***NE\_NUM*** ; x ++) {  
 str = Base64.*getEncoder*().encodeToString(str.getBytes()) ;  
 }  
 **return** str ;  
 }  
 */\*\*  
 \* 进行密码的处理操作  
 \** ***@param password*** *用户输入的真实密码  
 \** ***@return*** *与数据库保存匹配的加密的处理密码  
 \*/* **public static** String getPassword(String password) {  
 MD5Code md5 = **new** MD5Code() ;  
 String pass = **"{"** + password + **":"** + *createSeed*() + **"}"**;  
 **for** (**int** x = 0 ; x < ***NE\_NUM*** ; x ++) {  
 pass = md5.getMD5ofStr(pass) ;  
 }  
 **return** pass ;   
 }  
}

### 定义用户认证授权微服务

需要考虑如下几点：

1. 该服务需要进行数据库的开发，所以一定要配置数据库连接池
2. 既然要进行微服务的编写，那么就一定需要提供业务接口以及Dao实现子类，现在的实现依靠mybatis完成。
3. 所有的微服务最终要通过控制器的rest进行发布处理。
4. 【microboot-shiro-member-provider】配置Druid数据库连接池
5. 需要修改pom.xml配置文件，为项目整合添加相关依赖

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>druid</**artifactId**>  
</**dependency**>

<**dependency**>  
 <**groupId**>ch.qos.logback</**groupId**>  
 <**artifactId**>logback-core</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
</**dependency**>

1. 【microboot-shiro-member-provider】建立几个DAO接口：
2. 提供用户认证的DAO接口：IMemberDAO

@Mapper  
**public interface** IMemberDAO {  
 Member findById(String mid);  
}

1. 提供角色检测的IRoleDAO接口：

@Mapper  
**public interface** IRoleDAO {  
 Set<String> findAllRoleByMember(String mid);  
}

1. 权限检测接口IActionDAO：

@Mapper  
**public interface** IActionDAO {  
 Set<String> findAllActionByMember(String mid);  
}

1. 【microboot-shiro-member-provider】配置mybatis.config.xml

*<?***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**> *<!-- 进行Mybatis的相应的环境的属性定义 -->* <**settings**> *<!-- 在本项目之中开启二级缓存 -->* <**setting name="cacheEnabled" value="true"**/>  
 </**settings**>  
</**configuration**>

* 配置src/main/resources/mybatis/mapper/cn/ym/com/Member.xml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="cn.ym.com.dao.IMemberDAO"**>  
 <**select id="findById" parameterType="String" resultType="Member"**>  
 SELECT mid,name,password,locked FROM member WHERE mid=#{mid} ;  
 </**select**>  
</**mapper**>

* 配置src/main/resources/mybatis/mapper/cn/ym/com/Role.xml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="cn.ym.com.dao.IRoleDAO"**>  
 <**select id="findAllRoleByMember" parameterType="String" resultType="String"**>  
 SELECT rid FROM role WHERE rid IN (  
 SELECT rid FROM member\_role WHERE mid=#{mid}) ;  
 </**select**>  
</**mapper**>

* 配置src/main/resources/mybatis/mapper/cn/ym/com/Action.xml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="cn.ym.com.dao.IActionDAO"**>  
<**select id="findAllActionByMember" parameterType="String" resultType="String"**>  
 SELECT actid FROM action WHERE rid IN (  
 SELECT rid FROM member\_role WHERE mid=#{mid}) ;  
</**select**>  
</**mapper**>

1. 【microboot-shiro-member-provider】修改application.yml配置文件：

**server:  
 port:** 8001  
**mybatis:  
 config-location:** classpath:mybatis/mybatis.config.xml *# mybatis配置文件所在路径* **type-aliases-package:** cn.ym.com.vo *# 定义所有操作类的别名所在包* **mapper-locations:** *# 所有的mapper映射文件* - classpath:mybatis/mapper/\*\*/\*.xml  
**spring:  
 messages:  
 basename:** i18n/Messages,i18n/Pages  
 **datasource:  
 type:** com.alibaba.druid.pool.DruidDataSource *# 配置当前要使用的数据源的操作类型* **driver-class-name:** org.gjt.mm.mysql.Driver *# 配置MySQL的驱动程序类* **url:** jdbc:mysql://localhost:3306/mldn *# 数据库连接地址* **username:** root *# 数据库用户名* **password:** root *# 数据库连接密码* **dbcp2:** *# 进行数据库连接池的配置* **min-idle:** 5 *# 数据库连接池的最小维持连接数* **initial-size:** 5 *# 初始化提供的连接数* **max-total:** 5 *# 最大的连接数* **max-wait-millis:** 200 *# 等待连接获取的最大超时时间*

1. 【microboot-shiro-member-provider】定义IMemberService业务接口：

public interface IMemberService {

public Member get(String mid) ;

public Map<String,Set<String>> listAuthByMember(String mid) ;

}

@Service  
**public class** MemberServiceImpl **implements** IMemberService {  
  
 @Resource  
 **private** IMemberDAO **memberDAO**;  
  
 @Resource  
 **private** IRoleDAO **roleDAO**;  
  
 @Resource  
 **private** IActionDAO **actionDAO**;  
  
 @Override  
 **public** Member get(String mid) {  
  
 **return this**.**memberDAO**.findById(mid);  
 }  
  
 @Override  
 **public** Map<String, Set<String>> listAuthByMember(String mid) {  
 Map<String, Set<String>> map = **new** HashMap<String, Set<String>>();  
 map.put(**"allRoles"**, **this**.**roleDAO**.findAllRoleByMember(mid));  
 map.put(**"allActions"**, **this**.**actionDAO**.findAllActionByMember(mid));  
 **return** map;  
 }  
}

1. 【microboot-shiro-member-provider】编写业务层测试类：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestMemberService {  
 @Resource  
 **private** IMemberService **memberService**;  
  
 @Test  
 **public void** testGet() {  
 System.***out***.println(**this**.**memberService**.get(**"admin"**));  
 }  
  
 @Test  
 **public void** testAuth() {  
 System.***out***.println(**this**.**memberService**.listAuthByMember(**"admin"**));  
 }  
}

1. 【microboot-shiro-member-provider】进行控制层编写，控制层现在给出的一定是rest服务：

@RestController  
**public class** MemberController {  
 @Resource  
 **private** IMemberService **memberService**;  
  
 @RequestMapping(value=**"/member/get"**,method= post)  
 **public** Object get(String mid, RequestMethod post) {  
 **return this**.**memberService**.get(mid) ;  
 }  
   
 @RequestMapping(value=**"/member/auth"**,method=RequestMethod.***POST***)  
 **public** Object auth(String mid) {  
 **return this**.**memberService**.listAuthByMember(mid) ;  
 }  
}

1. 【microboot-shiro-member-provider】编写控制层测试，如果要访问Rest服务肯定要使用RestTemplate模板完成，这个类现在为了简单起见，直接进行实例化处理：

@SpringBootTest(classes = StartSpringBootMain.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@WebAppConfiguration  
**public class** TestMemberController {  
 **private** RestTemplate **restTemplate** = **new** RestTemplate();  
  
 @Test  
 **public void** testGet() {  
 Member vo = **this**.**restTemplate**.postForObject(**"http://localhost:8001/member/get?mid=admin"**, **null**, Member.**class**);  
 }  
  
 @Test  
 **public void** testAuth() {  
 Map<String, Object> map = **this**.**restTemplate**.postForObject(**"http://localhost:8001/member/auth?mid=admin"**, **null**, Map.**class**);  
 Set<String> allRoles = **new** HashSet<String>();  
 Set<String> allActions = **new** HashSet<>();  
 allRoles.addAll((List) map.get(**"allRoles"**));  
 allRoles.addAll((List<String>) map.get(**"allRoles"**));  
 System.***out***.println(**"【角色】"** + allRoles);  
 System.***out***.println(**"【权限】"** + allActions);  
 }  
}

### shiro整合处理

在web模块中要求必须调用用户的认证与授权为服务（Realm）,而后进行各种依赖包的配置（shiro）、考虑到各种缓存的问题、认证与授权检测问题。

1. 【microboot-shiro-web】修改pom.xml配置文件，追加shiro的相关依赖程序包

**dependencies**>  
 <**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microboot-shiro-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-spring</**artifactId**>  
 <**version**>1.3.2</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-core</**artifactId**>  
 <**version**>1.3.1</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-ehcache</**artifactId**>  
 <**version**>1.3.1</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-quartz</**artifactId**>  
 <**version**>1.3.1</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-web</**artifactId**>  
 <**version**>1.3.1</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-mail</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>ch.qos.logback</**groupId**>  
 <**artifactId**>logback-core</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.kafka</**groupId**>  
 <**artifactId**>spring-kafka</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-jetty</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>springloaded</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
 </**dependency**>  
  
</**dependencies**>

1. 【microboot-shiro-web】建立一个RestTemplate的配置类对象

@Configuration  
**public class** RestConfig {  
 @Bean  
 **public** RestTemplate getRestTemplate() {  
 **return new** RestTemplate();  
 }  
}

1. 【microboot-shiro-web】Shiro中所有的认证与授权的处理都在Realm中

**public class** MemberRealm **extends** AuthorizingRealm {  
  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @Override  
 **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  
 System.***out***.println(**"++++++++++++++ 2、进行授权操作处理 ++++++++++++++"**);  
 *// 该操作的主要目的是取得授权信息，说的直白一点就是角色和权限数据* SimpleAuthorizationInfo auth = **new** SimpleAuthorizationInfo();  
 *// 执行到此方法的时候一定是已经进行过用户认证处理了（用户名和密码一定是正确的）* String mid = (String) principalCollection.getPrimaryPrincipal(); *//取得用户名* String url = **"http://localhost:8001/member/auth?mid="** + mid;  
 Map<String, Object> map = **this**.**restTemplate**.postForObject(url, **null**, Map.**class**);  
 Set<String> allRoles = **new** HashSet<>();  
 Set<String> allActions = **new** HashSet<String>();  
 allRoles.addAll((List<String>) map.get(**"allRoles"**));  
 allActions.addAll((List<String>) map.get(**"allActions"**));  
 auth.setRoles(allRoles); *// 保存所有的角色* auth.setStringPermissions(allActions); *// 保存所有的权限* **return** auth;  
 }  
  
 @Override  
 **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) **throws** AuthenticationException {  
 System.***out***.println(**"============== 1、进行认证操作处理 =============="**);  
 String mid = authenticationToken.getPrincipal().toString();  
 *// 取得用户名之后就需要通过业务层获取用户对象以确定改用户名是否可用* String url = **"http://localhost:8001/member/get?mid="** + mid;  
 Member member = **this**.**restTemplate**.postForObject(url, **null**, Member.**class**); *//通过用户名获取用户信息* **if** (member == **null**) {*// 表示该用户信息不存在，不存在则应该抛出一个异常* **throw new** UnknownAccountException(**"搞什么搞，用户名不存在！"**);  
 }  
  
 *// 用户名如果存在了，那么就需要确定密码是否正确* String password = PasswordUtil.*getPassword*(**new** String((**char**[]) authenticationToken.getCredentials()));  
 **if** (!password.equals(member.getPassword())) {  
 **throw new** IncorrectCredentialsException(**"密码都记不住，去死吧！"**);  
 }  
  
 *// 随后还需要考虑用户被锁定的问题* **if** (member.getLocked().equals(1)) { *//1表示被锁定，非0 表示未锁定* **throw new** LockedAccountException(**"被锁了，求解锁去吧！"**);  
 }  
  
 *// 定义需要进行返回的操作数据信息项，返回的认证信息使用应该是密文* SimpleAuthenticationInfo auth = **new** SimpleAuthenticationInfo(authenticationToken.getPrincipal(), password, **"memberRealm"**);  
 *// 在认证完成之后可以直接取得用户所需要的信息内容，保存在Session之中* SecurityUtils.*getSubject*().getSession().setAttribute(**"name"**, **"我的名字"**);  
 **return** auth;  
 }  
}

**package** cn.ym.com.realm;  
  
**import** cn.ym.com.util.enctype.PasswordUtil;  
**import** org.apache.shiro.authc.AuthenticationInfo;  
**import** org.apache.shiro.authc.AuthenticationToken;  
**import** org.apache.shiro.authc.credential.SimpleCredentialsMatcher;  
  
*/\*\*  
 \* 密码加密处理  
 \*/***public class** CustomerCredentialsMatcher **extends** SimpleCredentialsMatcher {  
 @Override  
 **public boolean** doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {  
 *// 取得原始的输入数据信息* Object tokenCredentials = PasswordUtil.*getPassword*(**super**.toString(token.getCredentials())).getBytes();  
 *// 取得认证数据库中的数据* Object accountCredentials = **super**.getCredentials(info);  
 **return super**.equals(tokenCredentials, accountCredentials);  
 }  
}

1. 【microboot-shiro-web】现在虽然准备好了realm程序类，但是整个shiro进行整合处理的时候实际上需要编写大量的配置程序类。这时候虽然可以直接使用xml配置文件，虽然可以，但是不标准，最好的做法是将xml的配置项变为Bean配置。

**package** cn.ym.com.config;  
  
**import** cn.ym.com.realm.CustomerCredentialsMatcher;  
**import** cn.ym.com.realm.MemberRealm;  
**import** org.apache.shiro.cache.ehcache.EhCacheManager;  
**import** org.apache.shiro.mgt.RememberMeManager;  
**import** org.apache.shiro.realm.Realm;  
**import** org.apache.shiro.session.mgt.SessionManager;  
**import** org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO;  
**import** org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator;  
**import** org.apache.shiro.session.mgt.eis.SessionDAO;  
**import** org.apache.shiro.session.mgt.eis.SessionIdGenerator;  
**import** org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler;  
**import** org.apache.shiro.spring.LifecycleBeanPostProcessor;  
**import** org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor;  
**import** org.apache.shiro.spring.web.ShiroFilterFactoryBean;  
**import** org.apache.shiro.web.filter.authc.FormAuthenticationFilter;  
**import** org.apache.shiro.web.filter.authc.LogoutFilter;  
**import** org.apache.shiro.web.mgt.CookieRememberMeManager;  
**import** org.apache.shiro.web.mgt.DefaultWebSecurityManager;  
**import** org.apache.shiro.web.servlet.SimpleCookie;  
**import** org.apache.shiro.web.session.mgt.DefaultWebSessionManager;  
**import** org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator;  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.context.annotation.DependsOn;  
  
**import** javax.servlet.Filter;  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
@Configuration  
**public class** ShiroConfig {  
  
 @Bean  
 **public** MemberRealm getRealm() {*// 1、获取配置的Realm，之所以没使用注解配置，是因为此处需要考虑到加密处理* MemberRealm realm = **new** MemberRealm();  
 realm.setCredentialsMatcher(**new** CustomerCredentialsMatcher());  
 **return** realm;  
 }  
  
 @Bean(name = **"lifecycleBeanPostProcessor"**)  
 **public** LifecycleBeanPostProcessor getLifecycleBeanPostProcessor() {  
 **return new** LifecycleBeanPostProcessor();  
 }  
  
 @Bean  
 @DependsOn(**"lifecycleBeanPostProcessor"**)  
 **public** DefaultAdvisorAutoProxyCreator getDefaultAdvisorAutoProxyCreator() {  
 DefaultAdvisorAutoProxyCreator daap = **new** DefaultAdvisorAutoProxyCreator();  
 daap.setProxyTargetClass(**true**);  
 **return** daap;  
 }  
  
 @Bean  
 **public** EhCacheManager getCacheManager() {*// 2、缓存配置* EhCacheManager cacheManager = **new** EhCacheManager();  
 cacheManager.setCacheManagerConfigFile(**"classpath:ehcache.xml"**);  
 **return** cacheManager;  
 }  
  
 @Bean  
 **public** SessionIdGenerator getSessionIdGenerator() { *// 3* **return new** JavaUuidSessionIdGenerator();  
 }  
  
 @Bean  
 **public** SessionDAO getSessionDAO(SessionIdGenerator sessionIdGenerator) { *// 4* EnterpriseCacheSessionDAO sessionDAO = **new** EnterpriseCacheSessionDAO();  
 sessionDAO.setActiveSessionsCacheName(**"shiro-activeSessionCache"**);  
 sessionDAO.setSessionIdGenerator(sessionIdGenerator);  
 **return** sessionDAO;  
 }  
  
 @Bean  
 **public** RememberMeManager getRememberManager() { *// 5* CookieRememberMeManager rememberMeManager = **new** CookieRememberMeManager();  
 SimpleCookie cookie = **new** SimpleCookie(**"MLDNJAVA-RememberMe"**);  
 cookie.setHttpOnly(**true**);  
 cookie.setMaxAge(3600);  
 rememberMeManager.setCookie(cookie);  
 **return** rememberMeManager;  
 }  
  
 @Bean  
 **public** QuartzSessionValidationScheduler getQuartzSessionValidationScheduler() {  
 QuartzSessionValidationScheduler sessionValidationScheduler = **new** QuartzSessionValidationScheduler();  
 sessionValidationScheduler.setSessionValidationInterval(100000);  
 **return** sessionValidationScheduler;  
 }  
  
 @Bean  
 **public** AuthorizationAttributeSourceAdvisor getAuthorizationAttributeSourceAdvisor(  
 DefaultWebSecurityManager securityManager) {  
 AuthorizationAttributeSourceAdvisor aasa = **new** AuthorizationAttributeSourceAdvisor();  
 aasa.setSecurityManager(securityManager);  
 **return** aasa;  
 }  
  
 @Bean  
 **public** DefaultWebSessionManager getSessionManager(SessionDAO sessionDAO,  
 QuartzSessionValidationScheduler sessionValidationScheduler) { *// 6* DefaultWebSessionManager sessionManager = **new** DefaultWebSessionManager();  
 sessionManager.setGlobalSessionTimeout(1000000);  
 sessionManager.setDeleteInvalidSessions(**true**);  
 sessionManager.setSessionValidationScheduler(sessionValidationScheduler);  
 sessionManager.setSessionValidationSchedulerEnabled(**true**);  
 sessionManager.setSessionDAO(sessionDAO);  
 SimpleCookie sessionIdCookie = **new** SimpleCookie(**"mldn-session-id"**);  
 sessionIdCookie.setHttpOnly(**true**);  
 sessionIdCookie.setMaxAge(-1);  
 sessionManager.setSessionIdCookie(sessionIdCookie);  
 sessionManager.setSessionIdCookieEnabled(**true**);  
 **return** sessionManager;  
 }  
  
 @Bean  
 **public** DefaultWebSecurityManager getSecurityManager(Realm memberRealm, EhCacheManager cacheManager,  
 SessionManager sessionManager, RememberMeManager rememberMeManager) {*// 7* DefaultWebSecurityManager securityManager = **new** DefaultWebSecurityManager();  
 securityManager.setRealm(memberRealm);  
 securityManager.setCacheManager(cacheManager);  
 securityManager.setSessionManager(sessionManager);  
 securityManager.setRememberMeManager(rememberMeManager);  
 **return** securityManager;  
 }  
  
 **public** FormAuthenticationFilter getLoginFilter() { *// 在ShiroFilterFactoryBean中使用* FormAuthenticationFilter filter = **new** FormAuthenticationFilter();  
 filter.setUsernameParam(**"mid"**);  
 filter.setPasswordParam(**"password"**);  
 filter.setRememberMeParam(**"rememberMe"**);  
 filter.setLoginUrl(**"/loginPage"**); *// 登录提交页面* filter.setFailureKeyAttribute(**"error"**);  
 **return** filter;  
 }  
  
 **public** LogoutFilter getLogoutFilter() { *// 在ShiroFilterFactoryBean中使用* LogoutFilter logoutFilter = **new** LogoutFilter();  
 logoutFilter.setRedirectUrl(**"/"**); *// 首页路径，登录注销后回到的页面* **return** logoutFilter;  
 }  
  
 @Bean  
 **public** ShiroFilterFactoryBean getShiroFilterFactoryBean(DefaultWebSecurityManager securityManager) {  
 ShiroFilterFactoryBean shiroFilterFactoryBean = **new** ShiroFilterFactoryBean();  
 *// 必须设置 SecurityManager* shiroFilterFactoryBean.setSecurityManager(securityManager);  
 shiroFilterFactoryBean.setLoginUrl(**"/loginPage"**); *// 设置登录页路径* shiroFilterFactoryBean.setSuccessUrl(**"/pages/hello"**); *// 设置跳转成功页* shiroFilterFactoryBean.setUnauthorizedUrl(**"/pages/unauthUrl"**); *// 授权错误页* Map<String, Filter> filters = **new** HashMap<String, Filter>();  
 filters.put(**"authc"**, **this**.getLoginFilter());  
 filters.put(**"logout"**, **this**.getLogoutFilter());  
 shiroFilterFactoryBean.setFilters(filters);  
 Map<String, String> filterChainDefinitionMap = **new** HashMap<String, String>();  
 filterChainDefinitionMap.put(**"/logout"**, **"logout"**);  
 filterChainDefinitionMap.put(**"/loginPage"**, **"authc"**); *// 定义内置登录处理* filterChainDefinitionMap.put(**"/pages/back/\*\*"**, **"authc"**);  
 filterChainDefinitionMap.put(**"/\*"**, **"anon"**);  
 shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);  
 **return** shiroFilterFactoryBean;  
 }  
}

1. 【microboot-shiro-web】建立一个控制器

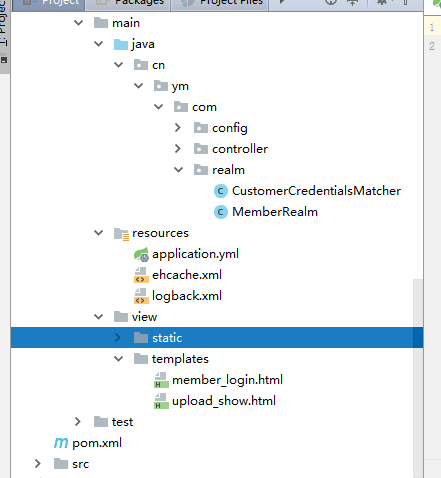
@RestController  
**public class** DeptController {  
  
 @RequiresRoles(**"dept"**)  
 @RequestMapping(**"/pages/back/dept/get"**)  
 **public** String get() {  
  
 **return "部门信息"**;  
 }  
}

1. 【microboot-shiro-web】登录出错后应该跑到表单上，所以建立一个MemberController，负责此跳转处理

@Controller  
**public class** MemberController {  
  
 @RequestMapping(**"/loginPage"**)  
 **public** String get() {  
 **return "member\_login"**;  
 }  
}

1. 【microboot-shiro-web】建立有个member\_login.html的页面；

<!DOCTYPE **HTML**>  
<**html xmlns:th="http://www.thymeleaf.org"**>  
<**head**>  
 <**title**>SpringBoot模版渲染</**title**>  
 <**script type="text/javascript" th:src="@{/js/main.js}"**></**script**>   
 <**link rel="icon" type="image/x-icon" href="/images/mldn.ico"**/>  
 <**meta http-equiv="Content-Type" content="text/html;charset=UTF-8"**/>  
</**head**>  
<**body**>  
 <**h1**>用户登录表单、<**span th:text="${errors}"**/></**h1**>  
   
 <**form th:action="@{/loginPage}" method="post"**>  
 登录名：<**input type="text" name="mid" value="mldnjava"**/><**br**/>  
 密**&nbsp;**码：<**input type="text" name="password" value="hello"**/><**br**/>  
 <**input type="submit" value="登录"**/>  
 </**form**>  
</**body**>   
</**html**>



### 使用redis进行缓存处理

现在使用了ehcache缓存组件进行了缓存处理，而实际开发中将使用redis实现缓存配置。

1. 【microboot-shiro-web】如果要使用缓存的使用，则首先一定要配置缓存处理类；

**package** cn.ym.com.cache;  
  
**import** java.util.Collection;  
**import** java.util.HashSet;  
**import** java.util.Iterator;  
**import** java.util.Set;  
  
**import** org.apache.commons.logging.Log;  
**import** org.apache.commons.logging.LogFactory;  
**import** org.apache.shiro.cache.Cache;  
**import** org.apache.shiro.cache.CacheException;  
**import** org.springframework.dao.DataAccessException;  
**import** org.springframework.data.redis.connection.RedisConnection;  
**import** org.springframework.data.redis.core.RedisCallback;  
**import** org.springframework.data.redis.core.RedisTemplate;  
  
**public class** RedisCache<K, V> **implements** Cache<K, V> {  
 **private** Log **log** = LogFactory.*getLog*(RedisCache.**class**);  
 **private** RedisTemplate<String, Object> **redisTempate**; *// 要提供有Redis处理工具类* **public** RedisCache(RedisTemplate<String, Object> redisTempate) {  
 **this**.**redisTempate** = redisTempate;  
 }  
 @Override  
 **public** V get(K key) **throws** CacheException {  
 **log**.info(**"### get() : K = "** + key);  
 **return** (V) **this**.**redisTempate**.opsForValue().get(key.toString());  
 }  
 @Override  
 **public** V put(K key, V value) **throws** CacheException {  
 **log**.info(**"### put() : K = "** + key + **"、V = "** + value);  
 **this**.**redisTempate**.opsForValue().set(key.toString(), value);  
 **return** value;  
 }  
  
 @Override  
 **public** V remove(K key) **throws** CacheException {  
 **log**.info(**"### remove() : K = "** + key);  
 V val = **this**.get(key);  
 **this**.**redisTempate**.delete(key.toString());  
 **return** val;  
 }  
  
 @Override  
 **public void** clear() **throws** CacheException {  
 **log**.info(**"### clear()"**);  
 **this**.**redisTempate**.execute(**new** RedisCallback<Boolean>() {  
 @Override  
 **public** Boolean doInRedis(RedisConnection connection)  
 **throws** DataAccessException {  
 connection.flushDb(); *// 清空数据库* **return true**;  
 }  
 });  
 }  
  
 @Override  
 **public int** size() {  
 **log**.info(**"### size()"**);  
 **return this**.**redisTempate**.execute(**new** RedisCallback<Integer>() {  
 @Override  
 **public** Integer doInRedis(RedisConnection connection)  
 **throws** DataAccessException {  
 **return** connection.keys(**"\*"**.getBytes()).size();  
 }  
 });  
 }  
  
 @Override  
 **public** Set<K> keys() {  
 **log**.info(**"### keys()"**);  
 **return this**.**redisTempate**.execute(**new** RedisCallback<Set<K>>() {  
 @Override  
 **public** Set<K> doInRedis(RedisConnection connection)  
 **throws** DataAccessException {  
 Set<K> set = **new** HashSet<K>();  
 Set<**byte**[]> keys = connection.keys(**"\*"**.getBytes());  
 Iterator<**byte**[]> iter = keys.iterator();  
 **while** (iter.hasNext()) {  
 set.add((K) iter.next());  
 }  
 **return** set;  
 }  
 });  
 }  
  
 @Override  
 **public** Collection<V> values() {  
 **log**.info(**"### values()"**);  
 **return this**.**redisTempate**.execute(**new** RedisCallback<Set<V>>() {  
 @Override  
 **public** Set<V> doInRedis(RedisConnection connection)  
 **throws** DataAccessException {  
 Set<V> set = **new** HashSet<V>();  
 Set<**byte**[]> keys = connection.keys(**"\*"**.getBytes());  
 Iterator<**byte**[]> iter = keys.iterator();  
 **while** (iter.hasNext()) {  
 set.add((V) connection.get(iter.next()));  
 }  
 **return** set;  
 }  
 });  
 }  
}

1. 【microboot-shiro-web】进行redis缓存管理类的配置

**package** cn.ym.com.cache;  
  
**import** org.apache.shiro.cache.Cache;  
**import** org.apache.shiro.cache.CacheException;  
**import** org.apache.shiro.cache.CacheManager;  
**import** org.springframework.data.redis.core.RedisTemplate;  
**import** org.springframework.stereotype.Component;  
  
**import** javax.annotation.Resource;  
**import** java.util.concurrent.ConcurrentHashMap;  
**import** java.util.concurrent.ConcurrentMap;  
  
@Component  
**public class** RedisCacheManager **implements** CacheManager {  
 *// CacheManager负责所有数据的缓存，那么对于数据而言，应该保存在缓存里面* **private final** ConcurrentMap<String, Cache> **caches** = **new** ConcurrentHashMap<String, Cache>();  
 @Resource  
 **private** RedisTemplate<String, Object> **redisTemplate**;  
  
 @Override  
 **public** Cache<Object, Object> getCache(String name) **throws** CacheException {  
 Cache<Object, Object> cache = **this**.**caches**.get(name); *// 通过Map取得cache数据* **if** (cache == **null**) { *// 当前的集合里面没有Cache的数据* cache = **new** RedisCache(**this**.**redisTemplate**); *// 实例化一个新的Cache对象* **this**.**caches**.put(name, cache);  
 }  
 **return** cache;  
 }  
  
}

3、【microboot-shiro-web】配置一个shiro中的一个Session管理操作

**package** cn.ym.com.session;  
  
**import** org.apache.commons.logging.Log;  
**import** org.apache.commons.logging.LogFactory;  
**import** org.apache.shiro.session.Session;  
**import** org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO;  
**import** org.springframework.data.redis.core.RedisTemplate;  
  
**import** javax.annotation.Resource;  
**import** java.io.Serializable;  
  
*// 此时的类将实现SessionDAO的改写***public class** RedisSessionDAO **extends** EnterpriseCacheSessionDAO {  
 **private** Log **log** = LogFactory.*getLog*(RedisSessionDAO.**class**);  
 @Resource  
 **private** RedisTemplate<String, Object> **redisTempate**; *// 要提供有Redis处理工具类* @Override  
 **protected** Serializable doCreate(Session session) { *// 创建Session，返回session id* **log**.info(**"\*\*\* doCreate : "** + session);  
 Serializable sessionId = **super**.doCreate(session); *// 创建sessionid  
 // 将当前创建好的Session的数据保存在Redis数据库里面* **this**.**redisTempate**.opsForValue().set(sessionId.toString(), session,  
 1800);  
 **return** sessionId;  
 }  
  
 @Override  
 **protected** Session doReadSession(Serializable sessionId) { *// 根据session  
 // id读取session数据* **log**.info(**"\*\*\* doReadSession : "** + sessionId);  
 Session session = **super**.doReadSession(sessionId); *// 读取Session数据* **if** (session == **null**) { *// 现在没有读取到session数据，通过Redis读取* **return** (Session) **this**.**redisTempate**.opsForValue()  
 .get(sessionId.toString());  
 }  
 **return null**;  
 }  
  
 @Override  
 **protected void** doUpdate(Session session) { *// 实现Session更新，每次操作都要更新* **log**.info(**"\*\*\* doUpdate : "** + session);  
 **super**.doUpdate(session);  
 **if** (session != **null**) {  
 **this**.**redisTempate**.opsForValue().set(session.getId().toString(),  
 session, 1800);  
 }  
 }  
  
 @Override  
 **protected void** doDelete(Session session) { *// session的删除处理* **log**.info(**"\*\*\* doDelete : "** + session);  
 **super**.doDelete(session);  
 **this**.**redisTempate**.delete(session.getId().toString());  
 }  
}

4、【microboot-shiro-web】在当前的项目开发中，配置shiro的Bean配置里面使用的还是EHCache缓存组件，所以需要进行更换处理。

(1)更换sessionDao实现子类:

@Bean  
**public** SessionDAO getSessionDAO(SessionIdGenerator sessionIdGenerator) { *// 4* RedisSessionDAO sessionDAO = **new** RedisSessionDAO(); *//使用redis进行Session管理* sessionDAO.setActiveSessionsCacheName(**"shiro-activeSessionCache"**);  
 sessionDAO.setSessionIdGenerator(sessionIdGenerator);  
 **return** sessionDAO;  
}

1. 更换使用的缓存组件：

@Bean  
**public** DefaultWebSecurityManager getSecurityManager(Realm memberRealm, RedisCacheManager cacheManager,  
 SessionManager sessionManager, RememberMeManager rememberMeManager) {*// 7* DefaultWebSecurityManager securityManager = **new** DefaultWebSecurityManager();  
 securityManager.setRealm(memberRealm);  
 securityManager.setCacheManager(cacheManager);  
 securityManager.setSessionManager(sessionManager);  
 securityManager.setRememberMeManager(rememberMeManager);  
 **return** securityManager;  
}

完整的shiro配置：

**package** cn.ym.com.config;  
  
**import** cn.ym.com.cache.RedisCacheManager;  
**import** cn.ym.com.realm.CustomerCredentialsMatcher;  
**import** cn.ym.com.realm.MemberRealm;  
**import** cn.ym.com.session.RedisSessionDAO;  
**import** org.apache.shiro.cache.ehcache.EhCacheManager;  
**import** org.apache.shiro.mgt.RememberMeManager;  
**import** org.apache.shiro.realm.Realm;  
**import** org.apache.shiro.session.mgt.SessionManager;  
**import** org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO;  
**import** org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator;  
**import** org.apache.shiro.session.mgt.eis.SessionDAO;  
**import** org.apache.shiro.session.mgt.eis.SessionIdGenerator;  
**import** org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler;  
**import** org.apache.shiro.spring.LifecycleBeanPostProcessor;  
**import** org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor;  
**import** org.apache.shiro.spring.web.ShiroFilterFactoryBean;  
**import** org.apache.shiro.web.filter.authc.FormAuthenticationFilter;  
**import** org.apache.shiro.web.filter.authc.LogoutFilter;  
**import** org.apache.shiro.web.mgt.CookieRememberMeManager;  
**import** org.apache.shiro.web.mgt.DefaultWebSecurityManager;  
**import** org.apache.shiro.web.servlet.SimpleCookie;  
**import** org.apache.shiro.web.session.mgt.DefaultWebSessionManager;  
**import** org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator;  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.context.annotation.DependsOn;  
  
**import** javax.servlet.Filter;  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
@Configuration  
**public class** ShiroConfig {  
  
 @Bean  
 **public** MemberRealm getRealm() {*// 1、获取配置的Realm，之所以没使用注解配置，是因为此处需要考虑到加密处理* MemberRealm realm = **new** MemberRealm();  
 realm.setCredentialsMatcher(**new** CustomerCredentialsMatcher());  
 **return** realm;  
 }  
  
 @Bean(name = **"lifecycleBeanPostProcessor"**)  
 **public** LifecycleBeanPostProcessor getLifecycleBeanPostProcessor() {  
 **return new** LifecycleBeanPostProcessor();  
 }  
  
 @Bean  
 @DependsOn(**"lifecycleBeanPostProcessor"**)  
 **public** DefaultAdvisorAutoProxyCreator getDefaultAdvisorAutoProxyCreator() {  
 DefaultAdvisorAutoProxyCreator daap = **new** DefaultAdvisorAutoProxyCreator();  
 daap.setProxyTargetClass(**true**);  
 **return** daap;  
 }  
*/\*  
 @Bean  
 public EhCacheManager getCacheManager() {// 2、缓存配置  
 EhCacheManager cacheManager = new EhCacheManager();  
 cacheManager.setCacheManagerConfigFile("classpath:ehcache.xml");  
 return cacheManager;  
 }\*/* @Bean  
 **public** SessionIdGenerator getSessionIdGenerator() { *// 3* **return new** JavaUuidSessionIdGenerator();  
 }  
  
 @Bean  
 **public** SessionDAO getSessionDAO(SessionIdGenerator sessionIdGenerator) { *// 4* RedisSessionDAO sessionDAO = **new** RedisSessionDAO(); *//使用redis进行Session管理* sessionDAO.setActiveSessionsCacheName(**"shiro-activeSessionCache"**);  
 sessionDAO.setSessionIdGenerator(sessionIdGenerator);  
 **return** sessionDAO;  
 }  
  
 @Bean  
 **public** RememberMeManager getRememberManager() { *// 5* CookieRememberMeManager rememberMeManager = **new** CookieRememberMeManager();  
 SimpleCookie cookie = **new** SimpleCookie(**"MLDNJAVA-RememberMe"**);  
 cookie.setHttpOnly(**true**);  
 cookie.setMaxAge(3600);  
 rememberMeManager.setCookie(cookie);  
 **return** rememberMeManager;  
 }  
  
 @Bean  
 **public** QuartzSessionValidationScheduler getQuartzSessionValidationScheduler() {  
 QuartzSessionValidationScheduler sessionValidationScheduler = **new** QuartzSessionValidationScheduler();  
 sessionValidationScheduler.setSessionValidationInterval(100000);  
 **return** sessionValidationScheduler;  
 }  
  
 @Bean  
 **public** AuthorizationAttributeSourceAdvisor getAuthorizationAttributeSourceAdvisor(  
 DefaultWebSecurityManager securityManager) {  
 AuthorizationAttributeSourceAdvisor aasa = **new** AuthorizationAttributeSourceAdvisor();  
 aasa.setSecurityManager(securityManager);  
 **return** aasa;  
 }  
  
 @Bean  
 **public** DefaultWebSessionManager getSessionManager(SessionDAO sessionDAO,  
 QuartzSessionValidationScheduler sessionValidationScheduler) { *// 6* DefaultWebSessionManager sessionManager = **new** DefaultWebSessionManager();  
 sessionManager.setGlobalSessionTimeout(1000000);  
 sessionManager.setDeleteInvalidSessions(**true**);  
 sessionManager.setSessionValidationScheduler(sessionValidationScheduler);  
 sessionManager.setSessionValidationSchedulerEnabled(**true**);  
 sessionManager.setSessionDAO(sessionDAO);  
 SimpleCookie sessionIdCookie = **new** SimpleCookie(**"mldn-session-id"**);  
 sessionIdCookie.setHttpOnly(**true**);  
 sessionIdCookie.setMaxAge(-1);  
 sessionManager.setSessionIdCookie(sessionIdCookie);  
 sessionManager.setSessionIdCookieEnabled(**true**);  
 **return** sessionManager;  
 }  
  
 @Bean  
 **public** DefaultWebSecurityManager getSecurityManager(Realm memberRealm, RedisCacheManager cacheManager,  
 SessionManager sessionManager, RememberMeManager rememberMeManager) {*// 7* DefaultWebSecurityManager securityManager = **new** DefaultWebSecurityManager();  
 securityManager.setRealm(memberRealm);  
 securityManager.setCacheManager(cacheManager);  
 securityManager.setSessionManager(sessionManager);  
 securityManager.setRememberMeManager(rememberMeManager);  
 **return** securityManager;  
 }  
  
 **public** FormAuthenticationFilter getLoginFilter() { *// 在ShiroFilterFactoryBean中使用* FormAuthenticationFilter filter = **new** FormAuthenticationFilter();  
 filter.setUsernameParam(**"mid"**);  
 filter.setPasswordParam(**"password"**);  
 filter.setRememberMeParam(**"rememberMe"**);  
 filter.setLoginUrl(**"/loginPage"**); *// 登录提交页面* filter.setFailureKeyAttribute(**"error"**);  
 **return** filter;  
 }  
  
 **public** LogoutFilter getLogoutFilter() { *// 在ShiroFilterFactoryBean中使用* LogoutFilter logoutFilter = **new** LogoutFilter();  
 logoutFilter.setRedirectUrl(**"/"**); *// 首页路径，登录注销后回到的页面* **return** logoutFilter;  
 }  
  
 @Bean  
 **public** ShiroFilterFactoryBean getShiroFilterFactoryBean(DefaultWebSecurityManager securityManager) {  
 ShiroFilterFactoryBean shiroFilterFactoryBean = **new** ShiroFilterFactoryBean();  
 *// 必须设置 SecurityManager* shiroFilterFactoryBean.setSecurityManager(securityManager);  
 shiroFilterFactoryBean.setLoginUrl(**"/loginPage"**); *// 设置登录页路径* shiroFilterFactoryBean.setSuccessUrl(**"/pages/hello"**); *// 设置跳转成功页* shiroFilterFactoryBean.setUnauthorizedUrl(**"/pages/unauthUrl"**); *// 授权错误页* Map<String, Filter> filters = **new** HashMap<String, Filter>();  
 filters.put(**"authc"**, **this**.getLoginFilter());  
 filters.put(**"logout"**, **this**.getLogoutFilter());  
 shiroFilterFactoryBean.setFilters(filters);  
 Map<String, String> filterChainDefinitionMap = **new** HashMap<String, String>();  
 filterChainDefinitionMap.put(**"/logout"**, **"logout"**);  
 filterChainDefinitionMap.put(**"/loginPage"**, **"authc"**); *// 定义内置登录处理* filterChainDefinitionMap.put(**"/pages/back/\*\*"**, **"authc"**);  
 filterChainDefinitionMap.put(**"/\*"**, **"anon"**);  
 shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);  
 **return** shiroFilterFactoryBean;  
 }  
}

1. 【microboot-shiro-web】修改application.yml配置文件进行redis配置

**server:  
 port:** 8080  
**spring:  
 redis:  
 host:** 192.168.1.241  
 **port:** 6379  
 **password:** 12345  
 **timeout:** 1000  
 **database:** 0  
 **pool:  
 max-active:** 10  
 **max-idle:** 8  
 **min-idle:** 2  
 **max-wait:** 100

1. 【microboot-shiro-web】追加redis项目依赖：

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

7、【microboot-shiro-web】创建RedisTemplate的配置程序类

**package** cn.ym.com.config;  
  
**import** cn.ym.com.util.RedisObjectSerializer;  
**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** javax.annotation.Resource;  
  
@Configuration  
**public class** RedisConfig {  
 @Resource  
 **private** JedisConnectionFactory **jedisConnectionFactory**;  
  
 @Bean(**"shiroRedis"**)  
 **public** RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {  
 RedisTemplate<String, Object> template = **new** RedisTemplate<String, Object>();  
 template.setConnectionFactory(**this**.**jedisConnectionFactory**);  
 template.setKeySerializer(**new** StringRedisSerializer());  
 template.setValueSerializer(**new** RedisObjectSerializer());  
 **return** template;  
 }  
}

**package** cn.ym.com.util;  
  
**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;  
  
**public class** RedisObjectSerializer **implements** RedisSerializer<Object> {  
 **private** Converter<Object, **byte**[]> **serializer** = **new** SerializingConverter();  
 **private** Converter<**byte**[], Object> **deserializer** = **new** DeserializingConverter();  
 **private static final byte**[] ***EMPTY\_ARRAY*** = **new byte**[0];  
  
 @Override  
 **public byte**[] serialize(Object object) **throws** SerializationException {  
 **if** (object == **null**) {  
 **return *EMPTY\_ARRAY***;  
 }  
 **try** {  
 **return serializer**.convert(object);  
 } **catch** (Exception ex) {  
 **return *EMPTY\_ARRAY***;  
 }  
 }  
  
 @Override  
 **public** Object deserialize(**byte**[] bytes) **throws** SerializationException {  
 **if** (**this**.isEmpty(bytes)) {  
 **return null**;  
 }  
 **try** {  
 **return deserializer**.convert(bytes);  
 } **catch** (Exception ex) {  
 **throw new** SerializationException(**"序列化对象出错！"**, ex);  
 }  
 }  
  
 **private boolean** isEmpty(**byte**[] data) {  
 **return** (data == **null** || data.**length** == 0);  
 }  
}

### 5、thymeleaf整合shiro标签

1、【microboot-shiro-web】修改pom.xml配置文件，追加thymeleaf与shiro的整合依赖：

<**dependency**>  
 <**groupId**>com.github.theborakompanioni</**groupId**>  
 <**artifactId**>thymeleaf-extras-shiro</**artifactId**>  
 <**version**>1.2.1</**version**>  
</**dependency**>

2、修改shiro配置类，启用shiro页面支持：

@Bean  
**public** ShiroDialect getShiroDialect() {*// 必须配置此操作才可以使用thymeleaf-extras-shiro开发包* **return new** ShiroDialect();  
}

1. 建立一个新的页面：
2. 修改DeptController控制器，进行跳转的配置：

@Controller  
**public class** DeptController {  
  
 @RequiresAuthentication  
 @RequestMapping(**"/pages/back/dept/get"**)  
 @ResponseBody  
 **public** String get() {  
  
 **return "部门信息"**;  
 }  
  
 @RequestMapping(**"/pages/back/dept/show"**)  
 **public** String show() {  
 **return "dept\_show"**;  
 }  
}

(2建立页面：

<!DOCTYPE HTML>

<html xmlns:shiro="http://www.pollix.at/thymeleaf/shiro">

<head>

<title>SpringBoot模版渲染</title>

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

</head>

<body>

<h1>显示部门信息的内容</h1>

<h2>欢迎：<shiro:principal/></h2>

<p><a shiro:hasRole="emp">雇员管理</a></p>

<p><a shiro:hasRole="dept">部门管理</a></p>

<p><a shiro:hasPermission="emp:add">雇员增加</a></p>

<p><a shiro:hasPermission="dept:edit">部门修改</a></p>

<p shiro:notAuthenticated="">您还未登录，请先登录!</p>

<p shiro:authenticated="">欢迎光临!</p>

</body>

</html>