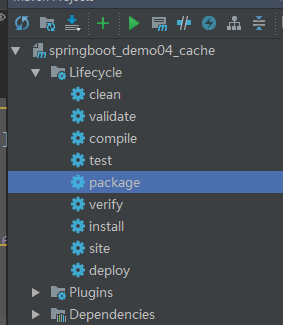
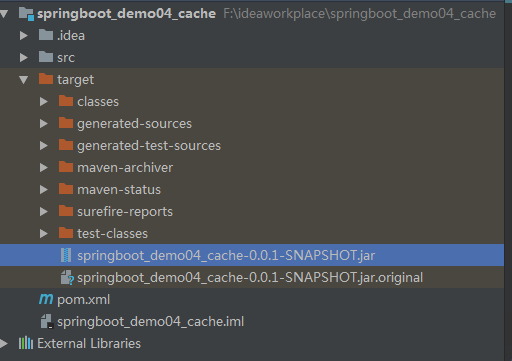
在dos窗口运行springboot文件

1. 第一步：将springboot编写的代码打包（先配置maven的环境变量，然后在工程文件夹下打开dos窗口，执行mvn clean package -Dmaven.test.skip=true即可）



1. 第二步：将项目中的jar复制到桌面



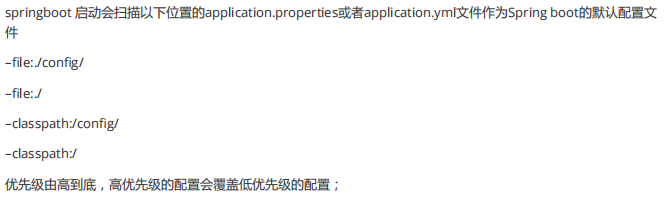
1. cd到jar包所在的文件夹
2. 运行命令启动springboot



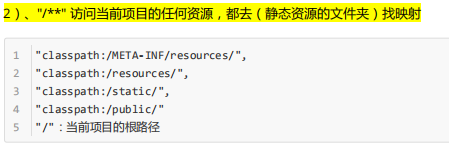
注意：可以在后面加端口号，按自己设置的端口号启动



1配置文件的默认加载位置



2静态文件的默认加载位置



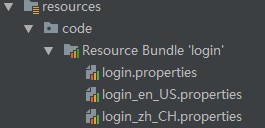
3thymeleaf表达式的写法

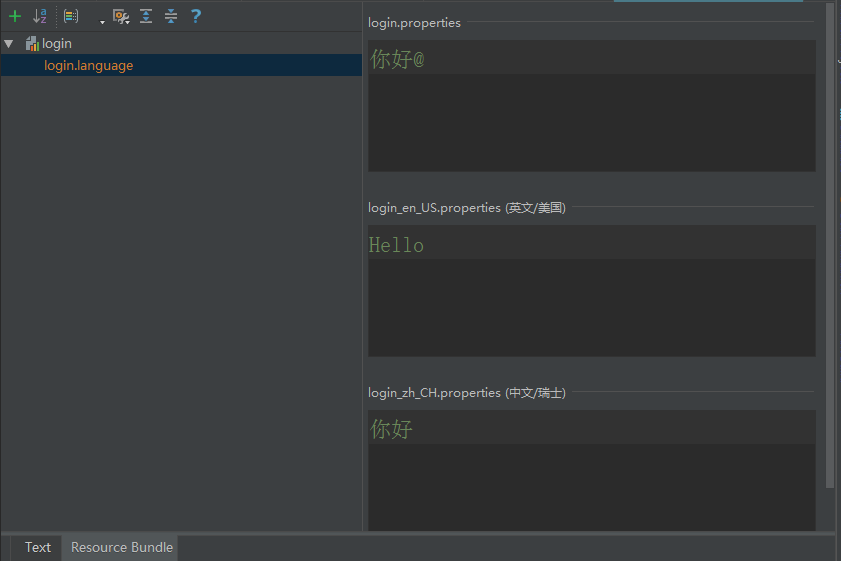
<!DOCTYPE html>  
<!--xmlns:th="http://www.thymeleaf.org"加这个写表达式有了提示-->  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
<table>  
 <!--users是传过来的集合，user是中的每一个值-->  
 <tr th:each="user:${users}">  
 <!--下面这个两个写法结果是一致的都是文本显示（会转义字符）-->  
 <td th:utext="${user}"></td>  
 <td>[(${user})]</td>  
 </tr>  
 <tr>  
 <!--这个两个不会转义字符-->  
 <td th:text="${number}"></td>  
 <td>[[${number}]]</td>  
 </tr>  
</table>  
</body>  
</html>

1. 国际化信息

第一步：编写国际化配置文件，抽取页面要显示的国际化消息

在resources下新建一个文件夹code，编写几个properties类型的文件





第二步:在application.properties中配置国际化配置的当前地址

spring.messages.basename=code.login

第三步：编写实现国际化的LocaleResolver接口的自定义类。

package com.example.demo01.config;  
  
import org.springframework.util.StringUtils;  
import org.springframework.web.servlet.LocaleResolver;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.util.Locale;  
  
*/\*\*  
 \* 自定义的类  
 \*/*public class MyLocaleResolver implements LocaleResolver {  
 @Override  
 public Locale resolveLocale(HttpServletRequest request) {  
 //判断请求传过来的参数里面有没有自定义设置国际化信息  
 String l = request.getParameter("l");  
 //先把默认的国际化信息赋给它  
 Locale locale = Locale.*getDefault*();  
 //如果人家的请求参数中有设置  
 if (!StringUtils.*isEmpty*(l)) {  
 //就把国际化信息参数换成人家的  
 String[] split = l.split("\_");  
 locale = new Locale(split[0], split[1]);  
 }  
 return locale;  
 }  
 @Override  
 public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale) {  
  
 }  
}

第四步：编写扩展SpringMVC的配置类（将上面的自定义类注入到其中）

package com.example.demo01.config;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.LocaleResolver;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;  
  
*/\*\*  
 \* 扩展SpringMVC的配置类  
 \*/*//声明为配置类  
@Configuration  
public class MyMvcConfig extends WebMvcConfigurerAdapter {  
 //注入自定义类  
 @Bean  
 public LocaleResolver localeResolver() {  
 return new MyLocaleResolver();  
 }  
}

第五步：前端页面

1. 将index.html首页放在templates文件夹下以保证可以用thymeleaf表达式
2. 在controller层中配置为默认显示

@RequestMapping({"/", "/index.html"})  
public String index() {  
 return "index";  
}

1. 首页页面

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
<!--给超链接拼接国际化参数l，（链接都为@开头，参数用小框号即可）-->  
<a th:href="@{/success(l='en\_US')}">英文</a><br/>  
<a th:href="@{/success(l='zh\_CH')}">中文</a>  
</body>  
</html>

1. 跳转的页面success.html

<!DOCTYPE html>  
<!--xmlns:th="http://www.thymeleaf.org"加这个写表达式有了提示-->  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
<!--#{}取的值为国际化信息配置的可变值-->  
<h1 th:text="#{login.language}"></h1>  
</body>  
</html>

1. 静态页面的不重启项目修改（小技巧）

第一步：在application.properties配置文件中设置禁用模板缓存

#禁用模板引擎的缓存  
spring.thymeleaf.cache=**false**

第二步：修改完html后，在idea中按Ctrl+F9重新编译页面，然后刷新页面即可。

1. thymeleaf的常用表达式

首先在前面要引入名称空间并且放在templates文件夹下才可以被渲染

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

1. 判断这个msg是否为空，为空前面的text就等于空

<p style="color: red" th:text="${msg}" th:if="${not #strings.isEmpty(msg)}"></p>

2.抽取公共片段替换可以减少重复代码

三种引入公共片段的th属性：

**th:insert**：将公共片段整个插入到声明引入的元素中

**th:replace**：将声明引入的元素替换为公共片段

**th:include**：将被引入的片段的内容包含进这个标签中

抽取方式

<footer th:fragment="copy">

© 2011 The Good Thymes Virtual Grocery

</footer>

引入方式

1<div th:insert="footer :: copy"></div>

2<div th:replace="footer :: copy"></div>

3<div th:include="footer :: copy"></div>

效果

1<div>

<footer>

© 2011 The Good Thymes Virtual Grocery

</footer>

</div>

2<footer>

© 2011 The Good Thymes Virtual Grocery

</footer>

3<div>

© 2011 The Good Thymes Virtual Grocery

</div>

3.三元表达式

th:value="${emp!=null}?${emp.lastName}"

例：

<input name="lastName" type="text" class="form‐control" placeholder="zhangsan"

th:value="${emp!=null}?${emp.lastName}">

4.判断这个标签是否存在

th:if="${emp!=null}"

例：

<input type="hidden" name="\_method" value="put" th:if="${emp!=null}"/>

5.遍历

<tr th:each="emp:${emps}">

例：

<tr th:each="emp:${emps}">

<td th:text="${emp.id}"></td>

<td>[[${emp.lastName}]]</td> //另一种取值的方法

<td th:text="${emp.email}"></td>

<td th:text="${emp.gender}==0?'女':'男'"></td> //三元表达式

<td th:text="${emp.department.departmentName}"></td> //对象中对象的一个属性

<td th:text="${#dates.format(emp.birth, 'yyyy‐MM‐dd HH:mm')}"></td>//内置的时间格式器

<td>

<a class="btn btn‐sm btn‐primary" th:href="@{/emp/}+${emp.id}">编辑</a>

<button th:attr="del\_uri=@{/emp/}+${emp.id}" class="btn btn‐sm btn‐danger

deleteBtn">删除</button> //自定义的属性del\_uri

</td>

</tr>

<script>

$(".deleteBtn").click(function(){

//删除当前员工的 （拿到这个表单改变他的属性"action"的值为上面的".deleteBtn"中的自定义属性，然后//触发它的submit事件）

$("#deleteEmpForm").attr("action",$(this).attr("del\_uri")).submit();

return false;

});

</script>

1. 登录功能的实现

思想原理：先写一个登录的页面，里面有form表单和错误提示栏，在编写controller处理登录信息，登录成功在session中保存用户的信息，跳转到成功页面。登录错误在session中存入错误信息，返回登录页面。（这样如果在成功页面刷新会导致表单重复提交）所以在controller的成功判断后，重定向到一个接口，然后跳转到成功页面。（这样又会有个问题，这个重定向的接口可以直接访问，不需要登录）所以在里面在配置个自定的拦截器，在请求之前这个方法中编写判断存在session中的用户信息代码，有则放行，无则拦截，然后在request域中存一个错误提示，转发到登录页面，在这个自定义的拦截器配置到SpringMVC的扩展类中设置拦截所有请求，后面再排除登录首页的请求和登录按钮的提交请求。

第一步：

登录页面的代码

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
<!--给超链接拼接国际化参数l，（链接都为@开头，参数用小框号即可）-->  
<form action="/user/login" method="post">  
 <table>  
 <tr>  
 <td>[[#{login.username}]]：<input name="username" type="text"/></td>  
 </tr>  
 <tr>  
 <td>[[#{login.password}]]：<input name="password" type="password"/></td>  
 </tr>  
 <tr>  
 <td><input type="submit" th:value="#{login.submit}"/></td>  
 </tr>  
 </table>  
</form>  
<p style="color: red" th:text="${msg}" th:if="${not #strings.isEmpty(msg)}"></p>  
<a th:href="@{/index.html(l='en\_US')}">英文</a>&ensp;&ensp;  
<a th:href="@{/index.html(l='zh\_CH')}">中文</a><br/>  
</body>  
</html>

自定义拦截器的代码

package com.example.demo01.config;  
  
import org.springframework.web.servlet.HandlerInterceptor;  
import org.springframework.web.servlet.ModelAndView;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
*/\*\*  
 \* 配置自定义拦截器（登录检查）  
 \*/*public class LoginHandlerInterceptor implements HandlerInterceptor {  
 //目标方法执行之前  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object o) throws Exception {  
 //拿到session中的登录标识loginUser判断是否登录过  
 Object user = request.getSession().getAttribute("loginUser");  
 if (null == user) {  
 //未登录，转发到登录页面，在session中存入错误代码，返回false为不放行。  
 request.setAttribute("msg", "没有权限，请先登录");  
 request.getRequestDispatcher("/index.html").forward(request, response);  
 return false;  
 } else {  
 //已登录，放行。  
 return true;  
 }  
 }  
  
 @Override  
 public void postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView) throws Exception {  
  
 }  
  
 @Override  
 public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) throws Exception {  
  
 }  
}

springmvc扩展类的代码

package com.example.demo01.config;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.LocaleResolver;  
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;  
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;  
  
*/\*\*  
 \* 扩展SpringMVC的配置类  
 \*/*//声明为配置类  
@Configuration  
public class MyMvcConfig extends WebMvcConfigurerAdapter {  
 //注入自定义类  
 @Bean  
 public LocaleResolver localeResolver() {  
 return new MyLocaleResolver();  
 }  
  
 //所有的WebMvcConfigurerAdapter组件都会一起起作用  
 @Bean //将组件注册在容器  
 public WebMvcConfigurerAdapter webMvcConfigurerAdapter() {  
 //匿名内部类  
 WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter() {  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController("/").setViewName("login");  
 registry.addViewController("/index.html").setViewName("login");  
 registry.addViewController("/main.html").setViewName("dashboard");  
 }  
  
 //注册拦截器  
 @Override  
 public void addInterceptors(InterceptorRegistry registry) {  
 //静态资源； \*.css , \*.js  
 //SpringBoot已经做好了静态资源映射  
 */\*\*  
 \* 将自定义拦截器加进去，/\*\*把所有的请求拦截后再里面在排除下三个请求。  
 \* （包括访问首页的两个和登录按钮的一个）  
 \*/* registry.addInterceptor(new LoginHandlerInterceptor()).addPathPatterns("/\*\*")  
 .excludePathPatterns("/index.html", "/", "/user/login");  
 }  
 };  
 return adapter;  
 }  
  
}

controller控制层的代码

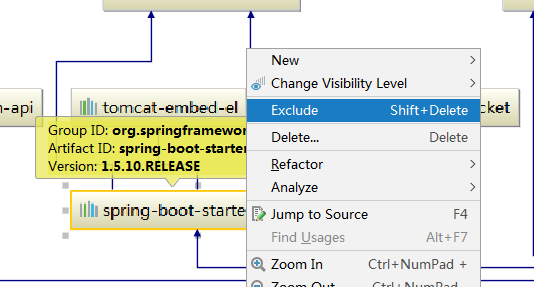
package com.example.demo01.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.util.StringUtils;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
  
import javax.servlet.http.HttpSession;  
import java.util.Map;  
  
@Controller  
public class LoginController {  
 @PostMapping("/user/login")  
 public String login(@RequestParam("username") String username,  
 @RequestParam("password") String password,  
 Map<String, Object> map, HttpSession session) {  
 if (!StringUtils.*isEmpty*(username) && username.equals("admin")  
 && !StringUtils.*isEmpty*(username) && password.equals("123456")) {  
 //登录成功在session中存入username，便于页面显示和拦截器不拦截后端页面的访问  
 session.setAttribute("loginUser", username);  
 //登录成功，防止表单重复提交，可以重定向到一个中间页面让其跳转成功页面  
 return "redirect:/main.html";  
 } else {  
 map.put("msg", "用户名或者密码错误！");  
 return "login";  
 }  
 }  
}

1. 替换为其他嵌入式Servlet容器（默认为tomcat）

第一步：在pom文件页面右击点击下面查看pom依赖的结构



第二步：tomcat的配置上右击排除它



第三步：在pom中加入新的依赖（jetty或者undertow）

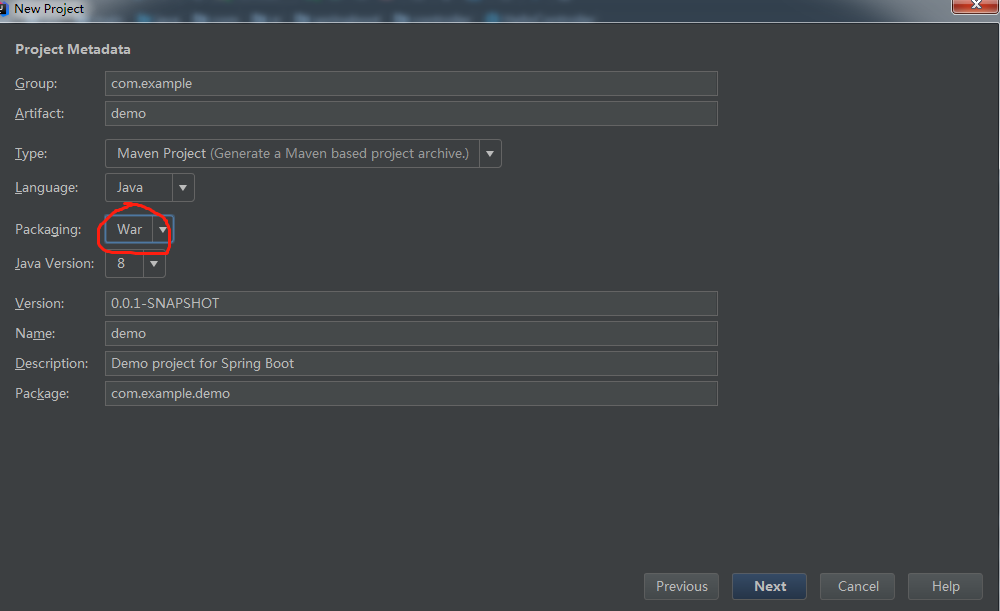
jetty：适合长连接（比如：聊天）

undertow：并发好（不支持jsp）

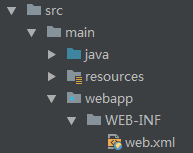
<**dependency**>  
 <**artifactId**>spring-boot-starter-jetty</**artifactId**>  
 <**groupId**>org.springframework.boot</**groupId**>  
</**dependency**>

9.使用外部servlet容器

第一步：在idea中创建一个war的springboot的项目



第二步：补充完整的项目结构



第三步：在controller层编写请求代码

@Controller  
public class HelloController {  
 @GetMapping("/hello")  
 public String hello(Model model) {  
 model.addAttribute("msg", "成功了");  
 return "success";  
 }  
}

第四步：编写application中的配置

#配置视图解析器的前缀  
spring.mvc.view.prefix=/WEB-INF/  
#配置视图解析器的后缀  
spring.mvc.view.suffix=.jsp

第五步：编写前端的jsp页面



hello.jsp

<%@ **page** contentType="text/html;charset=UTF-8" language="java" %>  
<html>  
<head>  
 <title>Title</title>  
</head>  
<body>  
<h1>Hello</h1>  
<a href="/hello">点击我！</a>  
</body>  
</html>

success.jsp

<%@ **page** contentType="text/html;charset=UTF-8" language="java" %>  
<html>  
<head>  
 <title>Title</title>  
</head>  
<body>  
<h1>success</h1>  
<h1>**${**msg**}**</h1>  
</body>  
</html>

第六步：配置tomcat，启动tomcat会自动启动springboot

小技巧：在一个类或接口的源码页面按ctrl+H可以看类的继承结构

整合Druid数据源

1.导入maven依赖

<!--整合druid数据源-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.1.8</version>  
</dependency>

2.配置yml文件

**spring:  
 datasource:** *# 数据源基本配置* **driver-class-name:** com.mysql.jdbc.Driver  
 **url:** jdbc:mysql://localhost:3306/customer  
 **username:** root  
 **password:** 123456  
 **type:** com.alibaba.druid.pool.DruidDataSource  
*# 数据源其他配置* **initialSize:** 5  
 **minIdle:** 5  
 **maxActive:** 20  
 **maxWait:** 60000  
 **timeBetweenEvictionRunsMillis:** 60000  
 **minEvictableIdleTimeMillis:** 300000  
 **validationQuery:** SELECT 1 FROM DUAL  
 **testWhileIdle:** true  
 **testOnBorrow:** false  
 **testOnReturn:** false  
 **poolPreparedStatements:** true  
*# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙* **filters:** stat,wall,log4j  
 **maxPoolPreparedStatementPerConnectionSize:** 20  
 **useGlobalDataSourceStat:** true  
 **connectionProperties:** druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500

3.编写配置类DruidConfig

package com.zr.springboot.config;  
  
import com.alibaba.druid.pool.DruidDataSource;  
import com.alibaba.druid.support.http.StatViewServlet;  
import com.alibaba.druid.support.http.WebStatFilter;  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.boot.web.servlet.FilterRegistrationBean;  
import org.springframework.boot.web.servlet.ServletRegistrationBean;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
import javax.sql.DataSource;  
import java.util.Arrays;  
import java.util.HashMap;  
import java.util.Map;  
  
@Configuration  
public class DruidConfig {  
 @ConfigurationProperties(prefix = "spring.datasource")  
 @Bean  
 public DataSource druid() {  
 return new DruidDataSource();  
 }  
  
 //配置Druid的监控  
 //1、配置一个管理后台的Servlet  
 @Bean  
 public ServletRegistrationBean statViewServlet() {  
 ServletRegistrationBean bean = new ServletRegistrationBean(new StatViewServlet(), "/druid/\*");  
 Map<String, String> initParams = new HashMap<>();  
 initParams.put("loginUsername", "admin");  
 initParams.put("loginPassword", "123456");  
 initParams.put("allow", "");//默认就是允许所有访问  
 initParams.put("deny", "192.168.15.21");  
 bean.setInitParameters(initParams);  
 return bean;  
 }  
  
 //2、配置一个web监控的filter  
 @Bean  
 public FilterRegistrationBean webStatFilter() {  
 FilterRegistrationBean bean = new FilterRegistrationBean();  
 bean.setFilter(new WebStatFilter());  
 Map<String, String> initParams = new HashMap<>();  
 initParams.put("exclusions", "\*.js,\*.css,/druid/\*");  
 bean.setInitParameters(initParams);  
 bean.setUrlPatterns(Arrays.*asList*("/\*"));  
 return bean;  
 }  
  
  
}

整合Mybatis

在Druid的基础上创建项目的时候添加Mybatis的依赖

mapper层代码

package com.zr.springboot.mapper;  
  
  
import com.zr.springboot.bean.Department;  
import org.apache.ibatis.annotations.\*;  
  
  
public interface IDeptMapper {  
 @Select("select \* from department where id=#{id}")  
 Department checkById(Integer id);  
 @Options(useGeneratedKeys = true, keyProperty = "id")  
 @Insert("insert into department(departmentName) VALUES (#{departmentName})")  
 int insertDepartment(Department department);  
  
 @Delete("DELETE FROM department WHERE id=#{id}")  
 int deleteById(Integer id);  
  
 @Update("update department set departmentName=#{departmentName} WHERE id=#{id}")  
 int updateDepartment(Department department);  
  
  
}

controller层代码

package com.zr.springboot.controller;  
  
  
import com.zr.springboot.bean.Department;  
import com.zr.springboot.mapper.IDeptMapper;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class DeptController {  
 @Autowired  
 private IDeptMapper deptMapper;  
  
 @GetMapping("/dept/{id}")  
 public Department checkById(@PathVariable("id") Integer id) {  
 return deptMapper.checkById(id);  
 }  
  
 @GetMapping("/dept")  
 public Department insertDepartment(Department department) {  
 deptMapper.insertDepartment(department);  
 return department;  
 }  
  
}

整合JPA

1. 在创建项目的时候选JPA的依赖
2. 编写一个是实体类

package com.zr.springboot.entity;  
  
  
import javax.persistence.\*;  
  
//使用JPA注解配置映射关系  
@Entity //告诉JPA这是一个实体类（和数据表映射的类）  
@Table(name = "tbl\_user") //@Table来指定和哪个数据表对应;如果省略默认表名就是user；  
public class User {  
  
 @Id //这是一个主键  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)//自增主键  
 private Integer id;  
  
 @Column(name = "last\_name",length = 50) //这是和数据表对应的一个列  
 private String lastName;  
 @Column //省略默认列名就是属性名  
 private String email;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
}

1. 编写一个类继承JpaRepository类

package com.zr.springboot.repository;  
  
  
import com.zr.springboot.entity.User;  
import org.springframework.data.jpa.repository.JpaRepository;  
  
*/\*\*  
 \* 继承JpaRepository来完成对数据库的操作  
 \* User为实体类，Integer为主键的类型  
 \*/*public interface UserRepository extends JpaRepository<User, Integer> {  
}

1. 编写控制层

package com.zr.springboot.controller;  
  
  
import com.zr.springboot.entity.User;  
import com.zr.springboot.repository.UserRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class UserController {  
  
 @Autowired  
 UserRepository userRepository;  
  
 @GetMapping("/user/{id}")  
 public User getUser(@PathVariable("id") Integer id) {  
 User user = userRepository.findOne(id);  
 return user;  
 }  
  
 @GetMapping("/user")  
 public User insertUser(User user) {  
 User save = userRepository.save(user);  
 return save;  
 }  
  
}

1. 编写yml配置文件

**spring:  
 datasource:  
 url:** jdbc:mysql://localhost:3306/jpa  
 **username:** root  
 **password:** 123456  
 **driver-class-name:** com.mysql.jdbc.Driver  
 **jpa:  
 hibernate:***# 更新或者创建数据表结构* **ddl-auto:** update  
*# 控制台显示SQL* **show-sql: true**

高级缓存cache（创建项目的时候要选cache，mybatis，jdbc，mysql，web）

第一步：在springboot的主启动类上加注解。

package com.zr.springboot;  
  
import org.mybatis.spring.annotation.MapperScan;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cache.annotation.EnableCaching;  
  
@SpringBootApplication  
@MapperScan("com.zr.springboot.mapper")  
@EnableCaching//开启基于注解的缓存  
public class SpringbootDemo04CacheApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootDemo04CacheApplication.class, args);  
   
 }  
}

第二步：在service层对要缓存的方法加注解。

package com.zr.springboot.service;  
  
import com.zr.springboot.bean.Employee;  
import com.zr.springboot.mapper.EmployeeMapper;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cache.annotation.\*;  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \** ***@CacheConfig*** *抽取缓存的公共配置  
 \* cacheNames = "emp"为每个缓存都有的  
 \*/*@CacheConfig(cacheNames = "emp")  
@Service  
public class EmployeeService {  
 @Autowired  
 private EmployeeMapper employeeMapper;  
  
 */\*\*  
 \** ***@Cacheable*** *注解  
 \* 标注缓存的方法 缓存中的是key和value的键值对  
 \* 默认key为传过来的参数（此处为id），value为  
 \* 返回的对象（此处为employee）  
 \* cacheNames指定缓存的区域名称  
 \* condition（符合缓存）后面的为判断，当id>1是为true才会缓存  
 \* unless（否定缓存）后面的也是判断，当id==2是false是才缓存  
 \* 流程：请求过来先去缓存中查如果有：就直接返回结果。如果没有：  
 \* 就去数据库中查，查到之后放入到缓存中，然后把结果返回  
 \*/* @Cacheable(cacheNames = {"emp"}, condition = "#id>1", unless = "#id==2")  
 public Employee getEmpById(Integer id) {  
 Employee employee = employeeMapper.getEmpById(id);  
 System.*out*.println("查了" + id + "号员工");  
 return employee;  
 }  
  
 */\*\*  
 \** ***@CachePut*** *注解  
 \* 流程：先去执行方法，去查数据库，修改数据库的值，  
 \* 然后将修改后的值查出来，放入到缓存区中。  
 \* key = "#employee.id" 设置缓存的key为参数employee中的id  
 \*/* @CachePut(cacheNames = "emp", key = "#employee.id")  
 public Employee updateEmp(Employee employee) {  
 employeeMapper.updateEmp(employee);  
 return employee;  
 }  
  
 */\*\*\*  
 \** ***@CacheEvict*** *清除缓存  
 \* allEntries = true 默认为false,当为true的时候会清空所有的缓存  
 \* beforeInvocation = true 默认为false，当为true的时候会在此方法  
 \* 之前清除缓存，（false的话会在方法执行之后清除缓存）（在方法中有  
 \* 异常的时候区别明显）  
 \*/* @CacheEvict(cacheNames = "emp", key = "#id")  
 public void deleteEmp(Integer id) {  
 //employeeMapper.deleteEmp(id);  
 //int i = 10 / 0;  
 System.*out*.println("清除缓存执行了");  
 }  
  
 */\*\*  
 \** ***@Caching*** *组合注解  
 \* 用lastName查完数据库后，按照key=lastName把查询结果Employee对象存进去缓存  
 \* 然后方法执行完，再把返回值的结果Employee对象的email属性和id属性当做key，在缓存中在存  
 \* 入Employee对象。(后续在用上述的方法查的时候，缓存中就已存在了id和email对应的同一个对象)  
 \* 注意:此处的result代表的为返回值对象Employee实例  
 \*/* @Caching(  
 cacheable =  
 {@Cacheable(cacheNames = "emp", key = "#lastName")},  
 put =  
 {@CachePut(cacheNames = "emp", key = "#result.email"),  
 @CachePut(cacheNames = "emp", key = "#result.id")  
 })  
 public Employee getEmpByLastName(String lastName) {  
 return employeeMapper.getEmpByLastName(lastName);  
 }  
  
  
}

controller层的代码

package com.zr.springboot.controller;  
  
  
import com.zr.springboot.bean.Employee;  
import com.zr.springboot.service.EmployeeService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class EmployeeController {  
  
 @Autowired  
 private EmployeeService employeeService;  
  
 @GetMapping("/emp/{id}")  
 public Employee getEmpById(@PathVariable("id") Integer id) {  
 return employeeService.getEmpById(id);  
 }  
  
 @GetMapping("/emp")  
 public Employee updateEmp(Employee employee) {  
 return employeeService.updateEmp(employee);  
 }  
  
 @GetMapping("/delemp/{id}")  
 public void deleteEmp(@PathVariable("id") Integer id) {  
 employeeService.deleteEmp(id);  
 }  
  
 @GetMapping("/emp/lastName/{lastName}")  
 public Employee getEmpByLastName(@PathVariable("lastName") String lastName) {  
 return employeeService.getEmpByLastName(lastName);  
 }  
}

javabean类的代码（Employee 实体类）

package com.zr.springboot.bean;  
  
public class Employee {  
   
 private Integer id;  
 private String lastName;  
 private String email;  
 private Integer gender; //性别 1男 0女  
 private Integer dId;  
   
   
 public Employee() {  
 super();  
 }  
  
   
 public Employee(Integer id, String lastName, String email, Integer gender, Integer dId) {  
 super();  
 this.id = id;  
 this.lastName = lastName;  
 this.email = email;  
 this.gender = gender;  
 this.dId = dId;  
 }  
   
 public Integer getId() {  
 return id;  
 }  
 public void setId(Integer id) {  
 this.id = id;  
 }  
 public String getLastName() {  
 return lastName;  
 }  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
 public String getEmail() {  
 return email;  
 }  
 public void setEmail(String email) {  
 this.email = email;  
 }  
 public Integer getGender() {  
 return gender;  
 }  
 public void setGender(Integer gender) {  
 this.gender = gender;  
 }  
 public Integer getdId() {  
 return dId;  
 }  
 public void setdId(Integer dId) {  
 this.dId = dId;  
 }  
 @Override  
 public String toString() {  
 return "Employee [id=" + id + ", lastName=" + lastName + ", email=" + email + ", gender=" + gender + ", dId="  
 + dId + "]";  
 }  
   
   
  
}

配置文件application.properties

spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/cache  
spring.datasource.username=root  
spring.datasource.password=123456  
#开启驼峰  
mybatis.configuration.map-underscore-to-camel-case=**true**#开启日志（可以在控制台看sql语句）  
logging.level.com.zr.springboot.mapper=debug

redis（缓存中间件）

使用docker拉取redis镜像

第一步：启动docker ：systemctl start docker

第二步: 搜索redis镜像：docker search redis

第三步: 拉取redis镜像：docker pull redis

第四步： 启动： docker run -d -p 6379:6379 --name myredis docker.io/redis

第五步： 查看启动的镜像：docker ps

第六步： 关闭服务：docker stop +容器id

（docker images：查看所有的镜像）（docker rmi +镜像id：删除镜像）

自定义Json数据缓存在redis中

第一步：导入依赖

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

第二步：在application.properties配置redis

#redis的地址  
spring.redis.host=localhost

第三步：编写配置类（自定义把对象Json成数据化存到redis数据库中）前提：对象要实现序列化接口

package com.zr.springboot.config;  
  
import com.zr.springboot.bean.Employee;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.redis.connection.RedisConnectionFactory;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;  
  
import java.net.UnknownHostException;  
  
@Configuration  
public class MyRedisConfig {  
 */\*\*  
 \* 自定义RedisTemplate模板序列化器  
 \*/* @Bean  
 public RedisTemplate<Object, Employee> empRedisTemplate(  
 RedisConnectionFactory redisConnectionFactory)  
 throws UnknownHostException {  
 RedisTemplate<Object, Employee> template = new RedisTemplate<Object, Employee>();  
 template.setConnectionFactory(redisConnectionFactory);  
 */\*\*  
 \* 将自定义的对象用自己想序列化的方式设置进去  
 \*/* Jackson2JsonRedisSerializer<Employee> jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer<Employee>(Employee.class);  
 template.setDefaultSerializer(jackson2JsonRedisSerializer);  
 return template;  
 }  
  
  
}

第四步：在Test类test

//将配合的类注入  
@Autowired  
private RedisTemplate<Object, Employee> empRedisTemplate;  
  
@Test  
public void test01() {  
 //调用类存入对象redis数据库  
 empRedisTemplate.opsForValue().set("employee", employeeMapper.getEmpById(1));  
}

消息队列（消息中间件）

作用：

1. 异步处理：

订单管理可以把下单信息放入消息队列中，物流管理和库存管理异步去找消息队列中的信息。

1. 应用解耦

订单管理把下单信息放到消息队列中，物流管理和库存管理订阅消息队列，自动去执行微服务模块，不需要订单管理去掉接口。

1. 流量削峰

秒杀请求过来，设置一个定长的消息队列，保存消息（剩下的消息抛弃），避免直接访问服务器，造成巨大压力。

消息服务包括两部分：消息代理和目的地。

消息队列有两种形式的目的地

1. 消息队列：点对点消息通信：只有一个发送者和接受者，不一定有一个接收者。
2. 主题：发布/订阅消息通信：发送者发布到主题，订阅这个主题的应用同时受到消息。

JMS（java消息服务）

基于java消息代理的规范。ActiveMQ是JMS的实现。

AMQP（高级消息队列）

是一个消息代理的规范，兼容JMS。RabbitMQ是AMQP的实现。



RabbitMQ

Message： 消息

Publisher： 消息的生产者

Exchange： 交换器

Queue：消息队列

Binding：绑定 用于消息队列和交换器的绑定。（多对多）

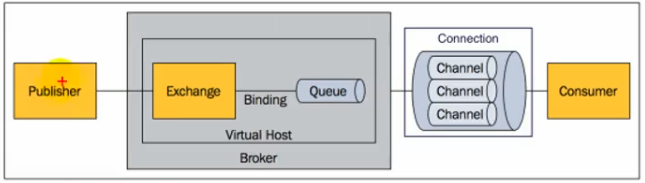
Connection：网络连接

Channel：信道

Consumer：消费者

Virtual Host：虚拟主机

Broker：消息队列服务器实体



交换器

1.Direct（点对点式）

2.Fanout（广播模式）

3.Topic（对路由件实施模糊匹配）

SpringBoot整合RabbitMQ

第一步：

启动Linux虚拟机，启动docker，启动rabbitmq服务

systemctl start docker

docker run -d -p 5672:5672 -p15672:15672 --name rabbitmq +镜像ID

第二步：

在浏览器中访问192.168.0.105:15672（虚拟机的ip）  
登录账号密码：guest

创建交换器（exchange）创建消息队列（queue）绑定连接

第三步：

创建一个springboot项目，选种web和rabbitmq的模板

配置application.properties文件

#配置rabbitmq的IP，用户名，密码，端口一般默认是5672  
spring.rabbitmq.host=192.168.0.105  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest

第四步：

编写Test代码

package com.zr.springboot;  
  
  
import com.zr.springboot.bean.Book;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.amqp.core.AmqpAdmin;  
import org.springframework.amqp.core.Binding;  
import org.springframework.amqp.core.DirectExchange;  
import org.springframework.amqp.core.Queue;  
import org.springframework.amqp.rabbit.core.RabbitTemplate;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.junit4.SpringRunner;  
  
@RunWith(SpringRunner.class)  
@SpringBootTest  
public class SpringbootDemo05RabbitmqApplicationTests {  
 */\*\*  
 \* 注入消息发送模板  
 \*/* @Autowired  
 RabbitTemplate rabbitTemplate;  
 @Autowired  
 AmqpAdmin amqpAdmin;  
  
 @Test  
 public void contextLoads() {  
 */\*\*  
 \* 在消息队列中放入消息‘  
 \* 参数一：交换器的名字  
 \* 参数二：队列名  
 \* 参数三：需要放入的对象  
 \*/* /\*Map<String, Object> map = new HashMap<>();  
 map.put("msg", "这是一个信息！");  
 map.put("data", Arrays.asList("aaa", 111, true, 'k'));  
 rabbitTemplate.convertAndSend("exchange.direct", "atguigu.news", map);  
 rabbitTemplate.convertAndSend("exchange.direct", "atguigu.news", new Book("水浒传", "施耐庵"));  
 rabbitTemplate.convertAndSend("exchange.fanout", "", new Book("三国", "曹操"));\*/  
 rabbitTemplate.convertAndSend("exchange.fanout", "", new Book("水浒传", "施耐庵"));  
 }  
  
 */\*\*  
 \* 从消息队列中取出一个消息  
 \*/* @Test  
 public void receive() {  
 Object o = rabbitTemplate.receiveAndConvert("atguigu.news");  
 System.*out*.println(o.getClass());  
 System.*out*.println(o);  
 }  
 */\*\*  
 \* 在代码中设置交换器和队列以及绑定  
 \*/* @Test  
 public void createExchange() {  
 //创建一个交换器  
 amqpAdmin.declareExchange(new DirectExchange("amqpadmin.exchange"));  
 //创建一个队列  
 amqpAdmin.declareQueue(new Queue("amqpadmin.queue"));  
 //声明一个绑定  
 amqpAdmin.declareBinding(new Binding("amqpadmin.queue",  
 Binding.DestinationType.*QUEUE*, "amqpadmin.exchange",  
 "haha", null));  
  
 }  
}

config下的配置类代码

package com.zr.springboot.config;  
  
  
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;  
import org.springframework.amqp.support.converter.MessageConverter;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
*/\*\*  
 \* 自定义配置类  
 \* MessageConverter注入一个json的数据序列化器  
 \*/*@Configuration  
public class MyAMQPConfig {  
 @Bean  
 public MessageConverter messageConverter() {  
 return new Jackson2JsonMessageConverter();  
 }  
}

第五步：

service层代码

package com.zr.springboot.service;  
  
import com.zr.springboot.bean.Book;  
import org.springframework.amqp.core.Message;  
import org.springframework.amqp.rabbit.annotation.RabbitListener;  
import org.springframework.stereotype.Service;  
  
  
*/\*\*  
 \** ***@RabbitListener*** *注释的方法会监控在自己配置的消息队列中有没有消息的  
 \* 产生，一但监听到就会执行自己的方法  
 \*/*@Service  
public class BookService {  
 */\*\*  
 \* 配置监听哪个消息队列里面的内容  
 \* queues后面可以写队列数组  
 \* atguigu.news是消息队列  
 \*/* @RabbitListener(queues = "atguigu.news")  
 public void receive(Book book) {  
 System.*out*.println("Book:" + book);  
 }  
 @RabbitListener(queues = "atguigu")  
 public void receiveMessage(Message message) {  
 //消息的头信息  
 System.*out*.println(message.getMessageProperties());  
 //消息的内容  
 System.*out*.println(message.getBody());  
 }  
}

在启动类上加注解

package com.zr.springboot;  
  
import org.springframework.amqp.rabbit.annotation.EnableRabbit;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@EnableRabbit //开启基于注解消息队列(RabbitMQ)  
@SpringBootApplication  
public class SpringbootDemo05RabbitmqApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootDemo05RabbitmqApplication.class, args);  
 }  
}

Elasticsearch的安装和启动

在docker中

docker run -e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300 --name elasticsearch +镜像ID （注意：每个字段的中间只能有一个空格键）

SpringBoot异步任务

第一步：

新建一个web的项目

第二步：

在主启动的类上加注解

package com.zr.springboot;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
@EnableAsync//开启对异步注解的支持  
@SpringBootApplication  
public class SpringbootDemo07TaskApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootDemo07TaskApplication.class, args);  
 }  
}

第三步：

在需要异步执行的方法上加注解

package com.zr.springboot.service;  
  
  
import org.springframework.scheduling.annotation.Async;  
import org.springframework.stereotype.Service;  
  
import java.util.concurrent.TimeUnit;  
  
@Service  
public class AsyncService {  
 */\*\*  
 \* 对这个方法加异步的注解，Spring会在线程池中开启一个线程  
 \* 去异步执行这个方法。  
 \*/* @Async  
 public void hello() {  
 try {  
 TimeUnit.*SECONDS*.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("Hello ....");  
 }  
}

SpringBoot定时任务

第一步：

新建一个web的项目

第二步：

在主启动的类上加注解

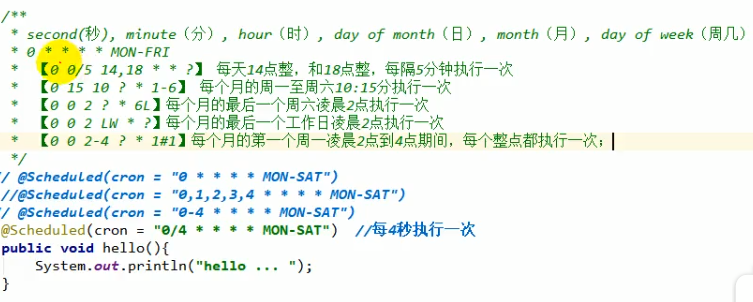
package com.zr.springboot;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.scheduling.annotation.EnableScheduling;  
  
@EnableScheduling//开启SpringBoot对定时注解的支持  
@SpringBootApplication  
public class SpringbootDemo07TaskApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootDemo07TaskApplication.class, args);  
 }  
}

第三步：

在需要定时执行的方法上加注解

package com.zr.springboot.service;  
  
  
import org.springframework.scheduling.annotation.Scheduled;  
import org.springframework.stereotype.Service;  
  
@Service  
public class AsyncService {  
 @Scheduled(cron = "0/5 \* \* \* \* 0-7")  
 public void hello() {  
 System.*out*.println("Hello ....");  
 }  
}

一些cron的表达式



SpringBoot发送邮件

第一步：导入发送邮件的依赖

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

第二步：配置application.properties文件

#发送的账号  
spring.mail.username=randylee1130@163.com  
#发送的密码  
spring.mail.password=li1520633673  
#@163网易服务器的地址  
spring.mail.host=smtp.163.com

第三步：编写测试类

package com.zr.springboot;  
  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.mail.SimpleMailMessage;  
import org.springframework.mail.javamail.JavaMailSenderImpl;  
import org.springframework.mail.javamail.MimeMessageHelper;  
import org.springframework.test.context.junit4.SpringRunner;  
  
import javax.mail.internet.MimeMessage;  
import java.io.File;  
  
@RunWith(SpringRunner.class)  
@SpringBootTest  
public class SpringbootDemo07TaskApplicationTests {  
 @Autowired  
 JavaMailSenderImpl javaMailSender;  
  
 */\*\*  
 \* 发送文本邮件  
 \*/* @Test  
 public void contextLoads() {  
 SimpleMailMessage message = new SimpleMailMessage();  
 message.setSubject("通知-今晚开会");  
 message.setText("今晚7:30开会");  
 message.setTo("1520633673@qq.com");  
 message.setFrom("randylee1130@163.com");  
 javaMailSender.send(message);  
 }  
  
 */\*\*  
 \* 发送带附件的邮件  
 \*/* @Test  
 public void test() throws Exception {  
 //创建一个复杂的邮件  
 MimeMessage mimeMessage = javaMailSender.createMimeMessage();  
 //第二个参数是上传邮件  
 MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, true);  
 helper.setSubject("通知-今晚开会");  
 //第二个参数是否解析为html  
 helper.setText("<p style='color:red'>今晚7:30开会</p>", true);  
 //收件人  
 helper.setTo("1520633673@qq.com");  
 //发件人  
 helper.setFrom("randylee1130@163.com");  
 //第一个参数：发送过去的文件名。第二个参数：本地的文件地址  
 helper.addAttachment("1.jpg", new File("F:\\1.jpg"));  
 helper.addAttachment("2.jpg", new File("F:\\2.jpg"));  
 javaMailSender.send(mimeMessage);  
 }  
}

SpringSecurity权限控制（认证和授权）

第一步：创建springboot项目,选择 thymeleaf和web的模板

第二步：导入依赖

<!--thymeleaf的自定义版本适配-->  
<thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>  
<thymeleaf-layout-dialect.version>2.3.0</thymeleaf-layout-dialect.version>  
<thymeleaf-extras-springsecurity4.version>3.0.2.RELEASE</thymeleaf-extras-springsecurity4.version>

<!--SpringSecurity-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<!--SpringSecurity和thymeleaf的整合的依赖-->  
<dependency>  
 <groupId>org.thymeleaf.extras</groupId>  
 <artifactId>thymeleaf-extras-springsecurity4</artifactId>  
</dependency>

第三步：在config包下编写security的配置类

package com.zr.springboot.config;  
  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
  
*/\*\*  
 \* 声明Security的配置类  
 \*/*@EnableWebSecurity  
public class MySecurityConfig extends WebSecurityConfigurerAdapter {  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 */\*\*  
 \* 定制请求的授权规则  
 \*/* http.authorizeRequests()  
 //所有的用户都可以访问/  
 .antMatchers("/").permitAll()  
 */\*\*  
 \*VIP1可以访问level1下的所有，以下相同  
 \* 相当于资源和角色的映射  
 \*/* .antMatchers("/level1/\*\*").hasRole("VIP1")  
 .antMatchers("/level2/\*\*").hasRole("VIP2")  
 .antMatchers("/level3/\*\*").hasRole("VIP3");  
 */\*\*  
 \*开启自动配置的登录功能，如果么有登录，没有授权就会来到登录页面  
 \* 默认访问/login会来到登录页面  
 \* 可以自定义登录后访问的页面此处的是请求/userlogin会跳转到login.html  
 \* 自定义账号的属性名为user，默认为username  
 \* 自定义密码的属性名为pwd，默认为password  
 \*/* http.formLogin().usernameParameter("user").passwordParameter("pwd").loginPage("/userlogin");  
 */\*\*  
 \* 开启自动配置的注销功能  
 \* 注销功能，默认访问/logout注销，清空session  
 \* logoutSuccessUrl注销完后去的地址  
 \*/* http.logout().logoutSuccessUrl("/");  
 */\*\*  
 \* 开启记住我的功能,设置记住我的参数  
 \*/* http.rememberMe().rememberMeParameter("remeber");  
  
 }  
  
 */\*\*  
 \* 定义认证规则  
 \*/* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 */\*\*  
 \* 设置不同的用户名和密码并授予角色信息（可以多角色）  
 \* 实际的生产中这些的信息要放在数据库中  
 \*/* auth.inMemoryAuthentication()  
 .withUser("zhangsan").password("123456").roles("VIP1")  
 .and()  
 .withUser("lisi").password("123456").roles("VIP1", "VIP2")  
 .and()  
 .withUser("wangwu").password("123456").roles("VIP1", "VIP2", "VIP3");  
 }  
}

第四步：编写登录页面

<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>Insert title here</title>  
</head>  
<body>  
<h1 align="center">欢迎登陆武林秘籍管理系统</h1>  
<hr>  
<div align="center">  
 <form th:action="@{/userlogin}" method="post">  
 用户名:<input name="user"/><br>  
 密码:<input name="pwd"><br/>  
 <input type="checkbox" name="remeber"> 记住我<br/>  
 <input type="submit" value="登陆">  
 </form>  
</div>  
</body>  
</html>

第五步：编写首页

<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org"  
 xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">  
<head>  
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
 <title>Insert title here</title>  
</head>  
<body>  
<h1 align="center">欢迎光临武林秘籍管理系统</h1>  
<!--authorize判断是否登录没有登录就显示以下-->  
<div sec:authorize="!isAuthenticated()">  
 <h2 align="center">游客您好，如果想查看武林秘籍 <a th:href="@{/userlogin}">请登录</a></h2>  
</div>  
<!--authorize判断是否登录登录就显示以下-->  
<div sec:authorize="isAuthenticated()">  
 <h2><span sec:authentication="name"></span>，您好,您的角色有：  
 <span sec:authentication="principal.authorities"></span></h2>  
 <form th:action="@{/logout}" method="post">  
 <input type="submit" value="注销"/>  
 </form>  
</div>  
<hr>  
<!--authorize判断是否拥有特定的角色，有就显示对应的内容-->  
<div sec:authorize="hasRole('VIP1')">  
 <h3>普通武功秘籍</h3>  
 <ul>  
 <li><a th:href="@{/level1/1}">罗汉拳</a></li>  
 <li><a th:href="@{/level1/2}">武当长拳</a></li>  
 <li><a th:href="@{/level1/3}">全真剑法</a></li>  
 </ul>  
  
</div>  
  
<div sec:authorize="hasRole('VIP2')">  
 <h3>高级武功秘籍</h3>  
 <ul>  
 <li><a th:href="@{/level2/1}">太极拳</a></li>  
 <li><a th:href="@{/level2/2}">七伤拳</a></li>  
 <li><a th:href="@{/level2/3}">梯云纵</a></li>  
 </ul>  
</div>  
<div sec:authorize="hasRole('VIP3')">  
 <h3>绝世武功秘籍</h3>  
 <ul>  
 <li><a th:href="@{/level3/1}">葵花宝典</a></li>  
 <li><a th:href="@{/level3/2}">龟派气功</a></li>  
 <li><a th:href="@{/level3/3}">独孤九剑</a></li>  
 </ul>  
</div>  
</body>  
</html>

docker安装zookeeper

第一步：拉取镜像

docker pull zookeeper

第二步：运行镜像

docker run --name zookeeper -p 2181:2181 --restart always -d +镜像id

第三步：查看运行的镜像

docker ps