

SpringBoot

1.HelloSpringBoot

1.建立是SpringBoot文件

方式一：

在 <https://start.spring.io/> 这个网站上建立项目

方式二：

在idea中建立新项目，选择 `spring initializr`，这里的操作界面是和官网的界面是一样的

统一选择Maven项目，java8，组件选择Web

Server URL: start.spring.io ⚙️

Name:

Location: 📁

Language: ☒ Java ☐ Kotlin ☐ Groovy

Type: ☒ Maven ☐ Gradle

Group:

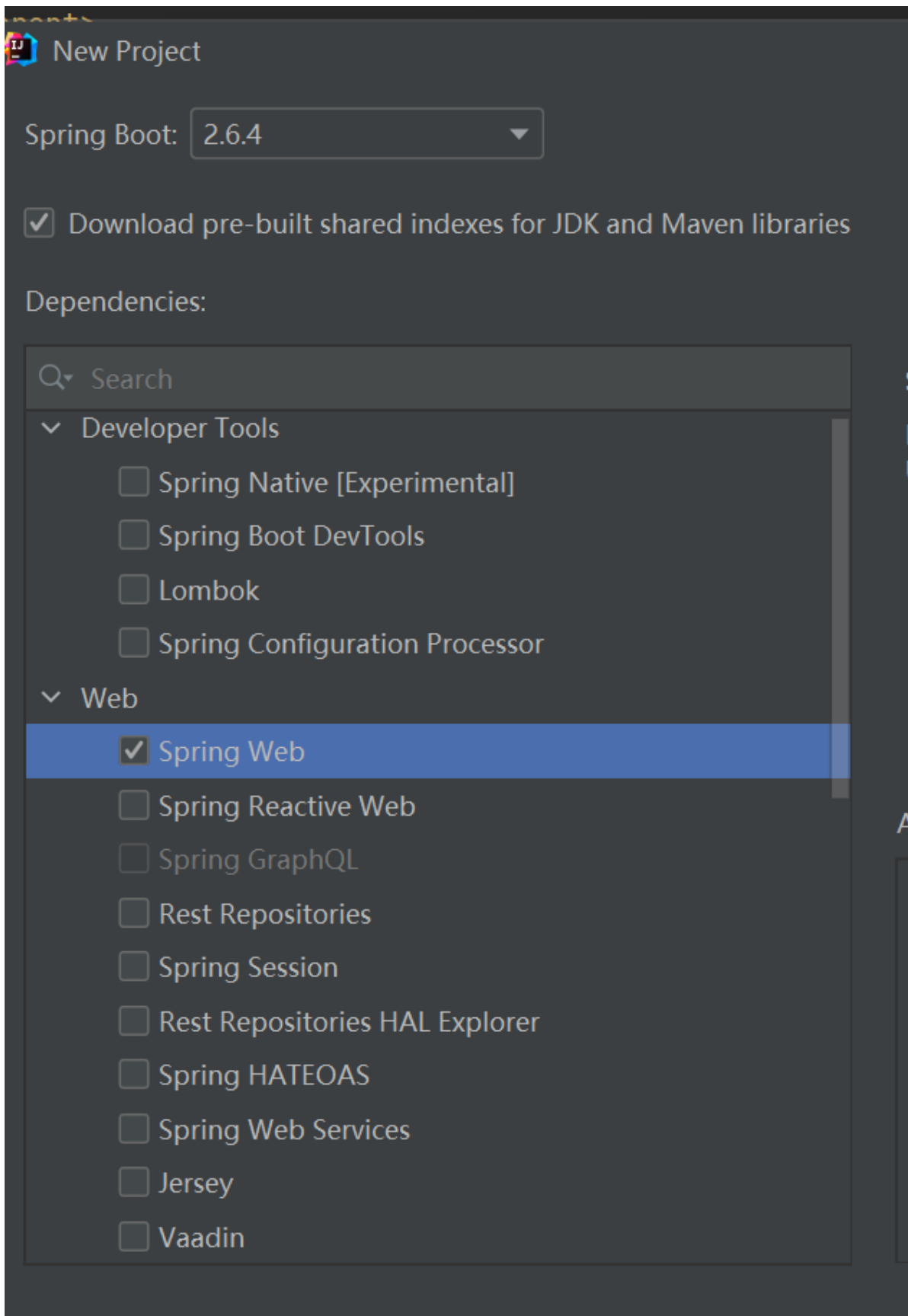
Artifact:

Package name:

Project SDK: ▼

Java: ▼

Packaging: ☒ Jar ☐ War



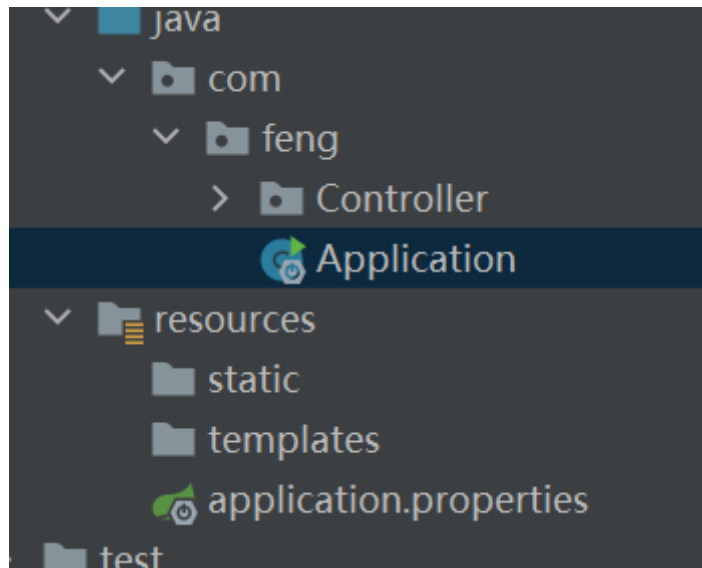
2.pom.xml分析

SpringBoot的初始依赖

```
<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>
```

3.注意观察初始类的位置

初始类也就是启动类，这个初始类在所有文件的最外面，也就是管理这些文件



4.编写Controller类

这个和Spring是一样的，只是不需要再配xml环境了，SpringBoot直接集成了

```
@Controller
public class controller {
    @RequestMapping(value = "/t1", method = RequestMethod.GET)
    @ResponseBody
    public String test1() { return "HelloSpringBoot"; }
}
```

注意

这里的application.properties的文件名不一定是application，但是一定是带有application的字样

2.yaml配置注入

1.对比properties和yaml文件

- application.properties的语法
 - key=value
- application.yaml的语法
 - key: value(注意这之间有一个空格)

可以在这样的配置文件中修改服务器的端口号

下面的是properties的语法

```
server.port=7878
```

下面的是yaml的语法

```
server:  
  port: 7878
```

yaml基础语法

- 空格不能省略
- 以缩进来控制层级关系，只要是左边对齐的一列数据都是同一个层级的
- 属性和值的大小写都是十分敏感的

下面的是yaml的行内写法

键值对(对象和值)

```
server: {port: 7878}
```

数组

```
pet:  
  - cat  
  - dog  
  - pig
```

行内写法

```
pets: [cat,dog,pig]
```

2.测试yaml注册实体类

1.建立相关的实体类

person类

```

import java.util.Map;
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
    private String name;
    private int age;
    private boolean happy;
    private Date birth;
    private Map<String, Object> map;
    private List<Object> list;
    private Dog dog;
}

```

dog类

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
public class Dog {
    private String name;
    private int age;
}

```

2.对应yaml文件的配置

这里要十分的注意空格

```

person:
  name: feng
  age: 3
  happy: true
  birth: 2022/03/12
  map: {"k1": 1, "k2": 2}
  list:
    - 1
    - 2
    - 3
dog:
  name: a
  age: 3

```

注意这里使用了一个注解

`@ConfigurationProperties(prefix = "person")`，这个是yaml注册的配套注解，这个注解是使yaml中配置person的实体类起作用

需要这个注解起作用要加入相关的依赖

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>

```

3.测试类的配置

这个测试的注解都是SpringBoot帮忙集成的，无需手写，注意 `@Autowired`，自动注入

```

@SpringBootTest
class ApplicationTests {
    @Autowired
    Person person;

    @Test
    void contextLoads() {
        System.out.println(person);
    }
}

```

拓展

yaml文件中可以新添加属性，还可使用相应的模块，实现模块下的方法，与el表达式灵活运用

```

person:
  name: feng
  age: 3
  happy: true
  birth: 2022/03/12
  map: {"k1": 1, "k2": 2}
  hello: feng
list:
  - 1
  - 2
  - 3
dog:
  name: ${person.hello:hello}a
  age: 3

```

3.通过properties文件注册实体类

1.建立相对应的person.properties文件

```

ApplicationTests
name=feng

```

2.建立和上面一样的实体类

这里用到一个新的注解 `@PropertySource("classpath:文件名.properties")`，这个是直接定位对应的properties文件，再通过 `@Value("${属性名}")` 来取得值

```
//@ConfigurationProperties(prefix = "person")
@PropertySource("classpath:person.properties")
public class Person {
    @Value("${name}")
    private String name;
```

3.JSR303校验

这个就是通过加载一个maven依赖

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

在类的定义上加上一个 `@Validated`，再在类的属性上加入相对应想要检验的注解就行，还可以自定义错误的消息，这个的注解有很多

```
@Validated
//@PropertySource("classpath:person.properties")
public class Person {
    //    @Value("feng")
    @Email(message = "你错了哦")
    private String name;
    //    @Value("${person.age}")
    private int age;
    private boolean happy;
    private Date birth;
    private Map<String, Object> map;
```

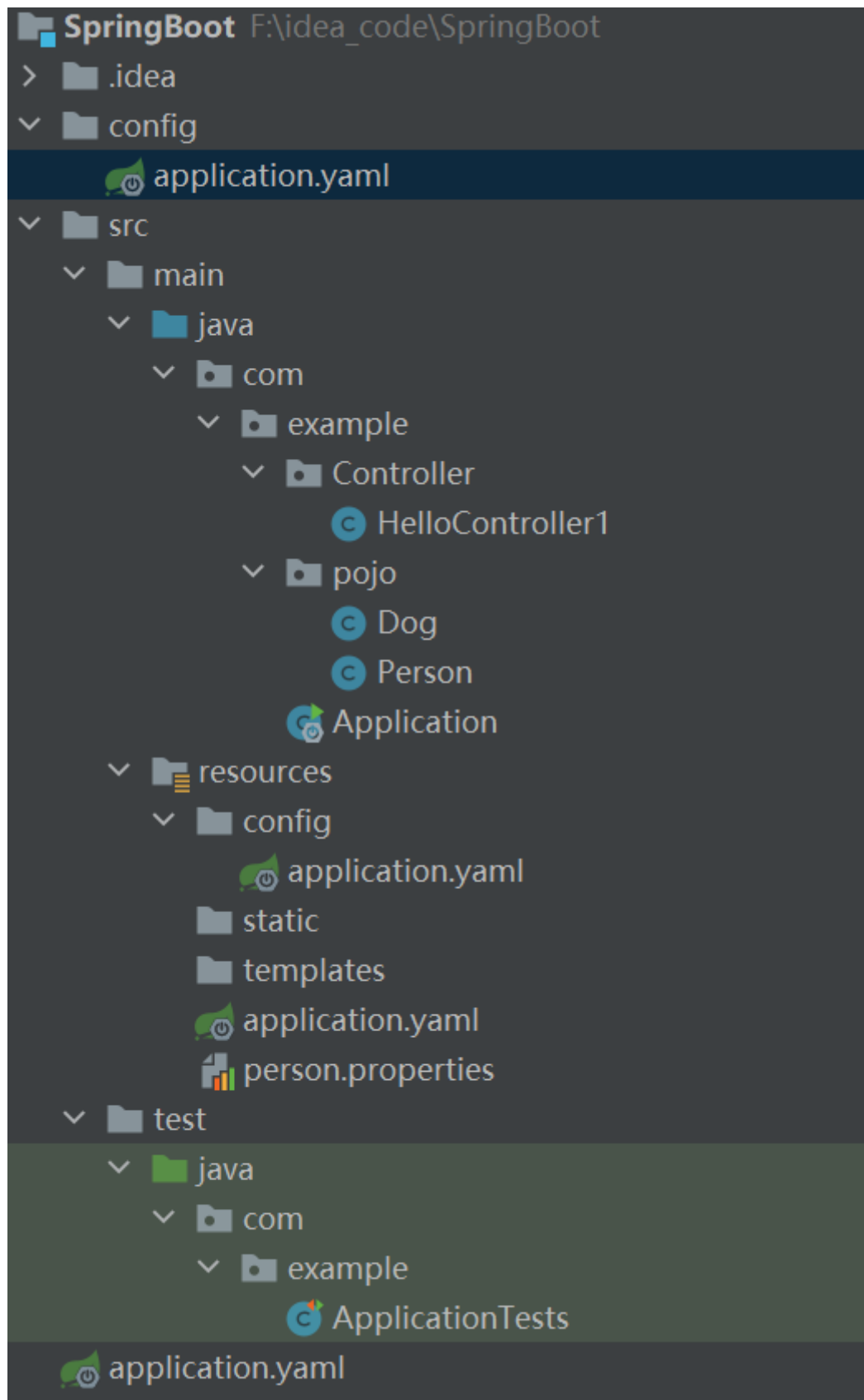
4.多环境配置

application.yml或application.properties可存在的位置有很多

- file:./config/
- file:./
- classpath:/config/
- classpath:/

优先级：

- 项目下的config文件夹中的配置文件
- 项目路径下的配置文件
- 资源路径下的config文件配置文件
- 资源路径下的配置文件

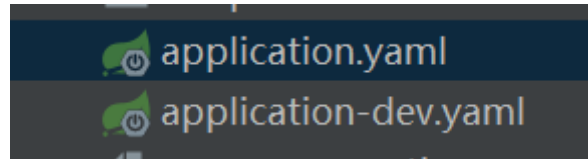


yml多环境配置

通过总的application.yaml来指定yaml文件生效

```
spring:
  profiles:
    active: dev
```

文件序列，这个文件名的设置要学习



application.yaml
application-dev.yaml

多环境，这个还可以直接在一个yaml文件中写

```
server:
  port: 7878
spring:
  profiles:
    active: test
---
server:
  port: 7880
spring:
  profiles: dev
---
server:
  port: 7881
spring:
  profiles: test
```

5.SpringBoot Web开发

1.静态资源

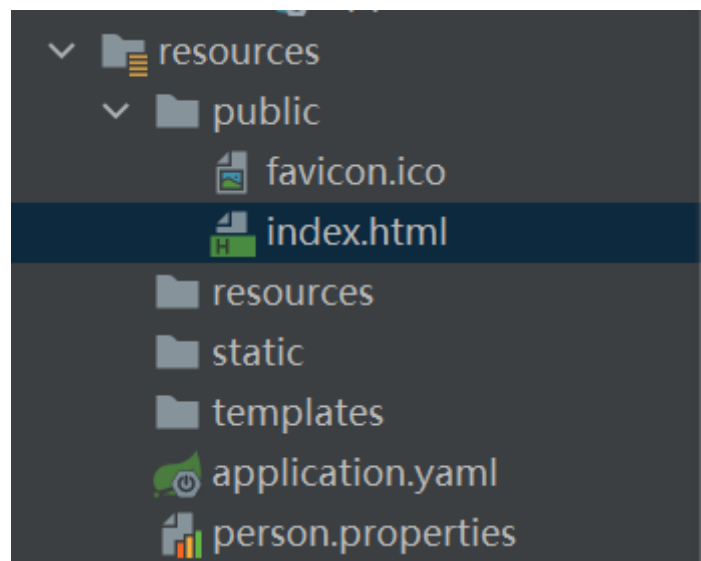
通过读源码，发现有放置静态资源路径的设置

- classpath:/META-INF/resources/;
- classpath:/resources/
- classpath:/static/
- classpath:/public/

这几个有个优先级 **resources>static>public**

通过读源码发现，SpringBootWeb首页是上面静态资源目录下的index.html文件，通过设置首页就可以每次访问该网站的时候直接进入index.html，通过读源码知道可以改变**网页的图标**，通过设置相关的属性就可以实现

下面是静态资源的放置目录



下面是设置icon的语句，这个是在application.yaml中配置的

```
# properties: test
spring:
  mvc:
    favicon:
      enable: false
```

6.Thymeleaf

1.导入maven依赖

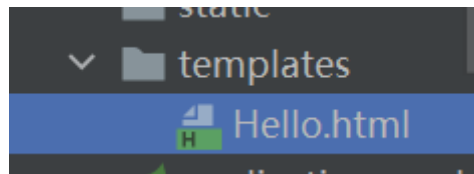
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
  <version>2.6.4</version>
</dependency>
```

还有要在html界面中加入这样的一句话

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

2.资源放置的位置

通过读取源码，知道了静态网页界面放在 `classpath:/templates/`，controller中会自动调用视图解析器进行拼接，之后进入这个静态页面，并且template下的页面只能通过controller来跳转



3.相关的css,js,img等资源路径:

`th:href=classpath:/static/` 下的资源，还要设置application.yaml的配置，这个就是将thymeleaf的资源缓存取消掉

实例：这个包括th:href, th:src等资源

```
th:href="@{/css/bootstrap.min.css}"
```

```
spring:
  mvc:
    favicon:
      enable: false
  thymeleaf:
    cache: false
```

4.Thymeleaf语法

- th:id 替换id `<input th:id="${user.id}">`
- th:text 文本替换 `<p text="${user.name}"></p>`
- th:utext 支持html的文本替换 `<p utext="${htmlcode}"></p>`
- th:object 替换对象 `<div th:object="${user}"></div>`
- th:value 替换值 `<input th:value="${user.name}">`
- th:each 迭代 `<tr th:each="user:${users}" th:text="${user}"></tr>`
- th:href 替换超链接 `<a th:href="@{index.html}">`
- th:src 替换资源 `<script type="text/javascript" th:src="@{index.js}"></script>`
导入js
- th:if 逻辑判断 ` 24">老油条`
- th:switch switch语句

```
<div th:switch="${user.role}">
  <p th:case="'admin'">用户是管理员</p>
  <p th:case="'manager'">用户是经理</p>
  <p th:case="*">用户是别的玩意</p>
</div>
```

链接表达式: `@{...}`

实例: `<link rel="stylesheet" th:href="@{index.css}">`

变量表达式: `${...}`

实例: `<div th:text="${name}"></div>`

- 取对象

```
<table bgcolor="#ffe4c4" border="1">
  <tr>
    <td>介绍</td>
    <td th:text="${user.name}"></td>
  </tr>
  <tr>
    <td>年龄</td>
    <td th:text="${user['age']}"></td>
  </tr>
  <tr>
    <td>介绍</td>
    <td th:text="${user.getDetail()}"></td>
  </tr>
</table>
```

- List的取值

```
<table bgcolor="#ffe4c4" border="1">
  <tr th:each="item:${userlist}">
    <td th:text="${item}"></td>
  </tr>
</table>
```

- Map的取值

```
<table bgcolor="#8fbc8f" border="1">
  <tr>
    <td>place:</td>
    <td th:text="${map.get('place')}"></td>
  </tr>
  <tr>
    <td>feeling:</td>
    <td th:text="${map['feeling']}"></td>
  </tr>
</table>
```

- 变量表达式 `*{...}`

```
<div th:object="${user}">
  <p>Name: <span th:text="*{name}">赛</span>.</p>
  <p>Age: <span th:text="*{age}">18</span>.</p>
  <p>Detail: <span th:text="*{detail}">好好学习</span>.</p>
</div>
```

- 信息表达式 `#{...}`

可以通过这个表达式, 来读取配置文件中的数据, 通过配合properties文件的一起使用来读取template文件夹下的东西, 国际化需要的注解

1.controller

这里会自动拼接域名地址

```
@RequestMapping("/t3")
public String test3(){
    return "test";
}
```

2.application.yml

下面就把原来的默认地址，这里修改了basename的属性来到template下的home.properties，将这个文件作为配置文件

```
spring:
  mvc:
    favicon:
      enable: false
    message:
      basename: template/home
```

3.html页面

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>枫</title>
</head>
<body>
    <h2>消息表达</h2>
    <table bgcolor="#ffe4c4" border="1">
        <tr>
            <td>name</td>
            <td th:text="#{bigsai.name}"></td>
        </tr>
        <tr>
            <td>年龄</td>
            <td th:text="#{bigsai.age}"></td>
        </tr>
        <tr>
            <td>province</td>
            <td th:text="#{province}"></td>
        </tr>
    </table>
</body>
</html>

```

7.MVC配置原理

1.拓展SpringMVC的配置

注意

这里的 `WebMvcConfigurer` 其实是 `WebMvcConfigurerAdapter`，这里要全部改成 `WebMvcConfigurerAdapter`

实例：拓展自己的视图解析器

通过加上注解 `@Configuration` 和 `@Bean` 的标签来实现自己配置的视图解析器生效

注意这里不要加上 `@EnableWebMvc` 注解，因为加上了之后就会将之前的原生的配置失效，这个是通过读取源码获得的

```

@Configuration
public class MyMVCConfig implements WebMvcConfigurer {
    @Bean
    public ViewResolver myViewResolver(){
        return new MyViewResolver();
    }
    public static class MyViewResolver implements ViewResolver{

        @Override
        public View resolveViewName(String viewName, Locale locale) throws Exception {
            return null;
        }
    }
}

```

实例二: 重写WebMvcConfigurerAdapter中的方法

实现这个接口中的方法，可以实现自定义SpringMVC的配置

```

@Configuration
public class MyMVCConfig implements WebMvcConfigurer {
    // @Bean

```

通过这个方法可以实现当访问 /feng 的时候会自动跳转到 test.html，转发的实现

```

@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController( urlPathOrPattern: "/feng").setViewName("test")
}

```

重定向的实现

```

@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addRedirectViewController( urlPath: "/feng", redirectUrl: "/t2");
}

```

插曲

通过设置可以改变首页的路径

```

server:
  port: 7878
servlet:
  context-path: /feng

```

结果为:

8.员工管理系统

1.实体类

1.学校部门

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Department {
    private Integer id;
    private String DepartmentName;
}
```

2.学校职工

```
import java.util.Date;

@Data
@NoArgsConstructor
public class Employee {
    private Integer id;
    private String name;
    private String email;
    private Integer gender; // 0 为女 1 为男
    private Department department;
    private Date birth;

    public Employee(int id, String name, String email, int gender, Department department) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.gender = gender;
        this.department = department;
        this.birth = new Date();
    }
}
```

2.模拟数据库

这两个要加上 `@Component` 或者 `@Repository` 声明这个是个组件类型，后面就可以直接 `@Autowired` 了，一般加的是 `@Repository`，因为这个是将接口的一个实现类交给Spring管理

1.部门数据库

```

public class DepartmentMapper {
    private static Map<Integer, Department> departments;
    static {
        departments = new HashMap<Integer, Department>();
        departments.put(101, new Department( id: 101, DepartmentName: "教学部"));
        departments.put(102, new Department( id: 102, DepartmentName: "市场部"));
        departments.put(103, new Department( id: 103, DepartmentName: "校研部"));
        departments.put(104, new Department( id: 104, DepartmentName: "运营部"));
        departments.put(105, new Department( id: 105, DepartmentName: "后勤部"));
    }
    public Collection<Department> getAllDepartments(){
        return departments.values();
    }
    public Department getDepartmentById(int id){
        return departments.get(id);
    }
}

```

2.职工数据库

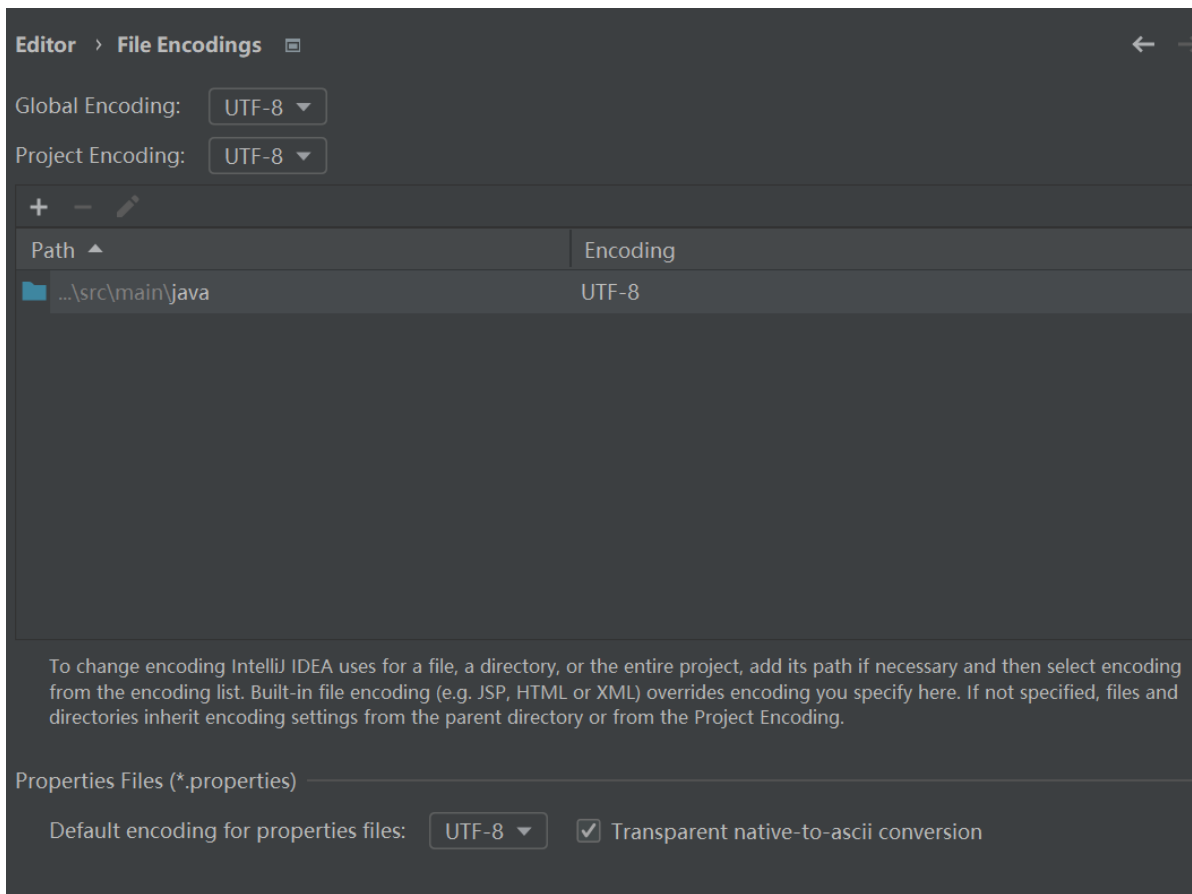
```

public class EmployeeMapper {
    private static Map<Integer, Employee> employees;
    static {
        employees = new HashMap<>();
        employees.put(101, new Employee( id: 101, name: "AA", email: "A2560965468@qq.com", gender: 0, new Department( id: 101, DepartmentName: "教学部")));
        employees.put(102, new Employee( id: 102, name: "BB", email: "B2560965468@qq.com", gender: 1, new Department( id: 102, DepartmentName: "市场部")));
        employees.put(103, new Employee( id: 103, name: "CC", email: "C2560965468@qq.com", gender: 0, new Department( id: 103, DepartmentName: "校研部")));
        employees.put(104, new Employee( id: 104, name: "DD", email: "D2560965468@qq.com", gender: 1, new Department( id: 104, DepartmentName: "运营部")));
        employees.put(105, new Employee( id: 105, name: "EE", email: "E2560965468@qq.com", gender: 0, new Department( id: 105, DepartmentName: "后勤部")));
    }
    private static int initId = 106;
    public void addEmployee(Employee employee){
        if(employee.getId()==null){
            employee.setId(initId++);
        }
        employees.put(employee.getId(), employee);
    }
    public Collection<Employee> getAllEmployees(){
        return employees.values();
    }
    public Employee getEmployeeById(Integer id){
        return employees.get(id);
    }
    public void deleteEmployeeById(Integer id){
        employees.remove(id);
    }
}

```

2.页面国际化

0.软件的支持 **这个是十分重要的**

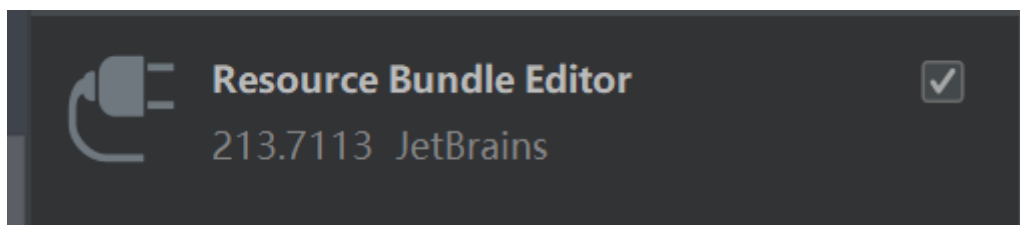


1.设置yaml文件的配置

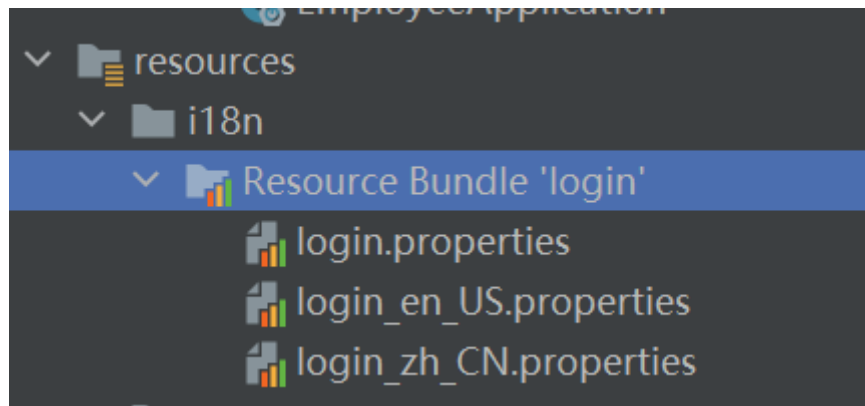
```
spring:
  messages:
    encoding: UTF-8
    basename: i18n/login
  thymeleaf:
    cache: false
  mvc:
    favicon:
      enable: false
```

2.配置对应的properties文件

在resources的目录下建立i18n文件，在加入properties文件，这里自动生成Bundle，注意idea这里要下载一个插件

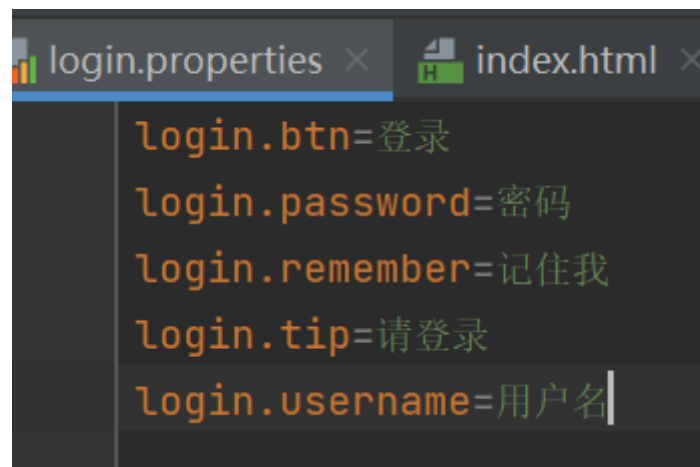


properties文件的配置

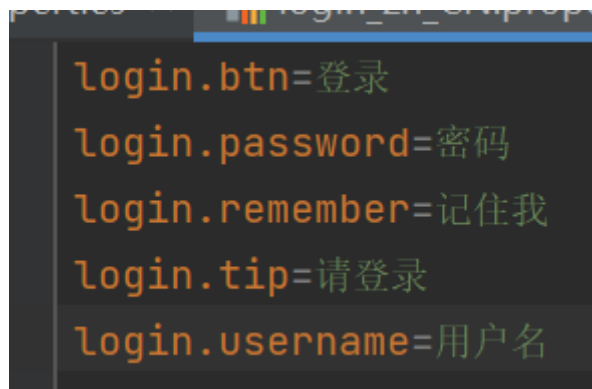


3.properties文件中的配置

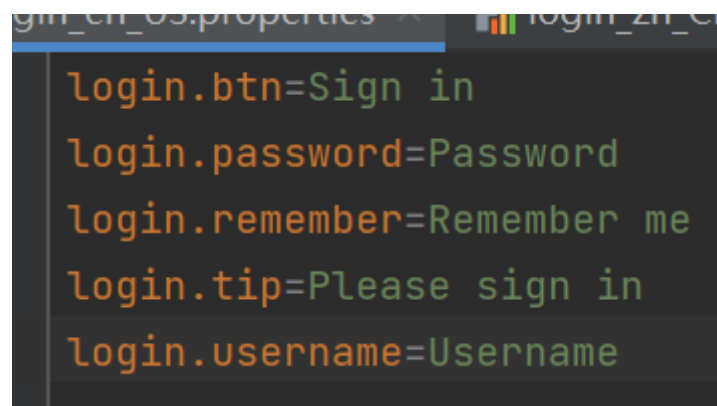
1.总文件的配置



2.中文文件的配置



3.英文文件的配置



4.对应页面的设置

在thymeleaf渲染下的a标签传递参数时和传统的是不一样的，下面的就是例子

```

<h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please sign in</h1>
<input type="text" class="form-control" th:placeholder="#{login.username}" required="" autofocus="">
<input type="password" class="form-control" th:placeholder="#{login.password}" required="">
<div class="checkbox mb-3">
    <label>
        <input type="checkbox" th:text="#{login.remember}">
    </label>
</div>
<button class="btn btn-lg btn-primary btn-block" type="submit" th:text="#{login.btn}"></button>
<p class="mt-5 mb-3 text-muted">© 2017-2018</p>
<a class="btn btn-sm" th:href="@{/login(lang='zh_CN')}">中文</a>
<a class="btn btn-sm" th:href="@{/login(lang='en_US')}">English</a>
```

5. 路径的设置

这里用到了 `@GetMapping` 关键词，其实就是 `@RequestMapping` 中加入method来实现

```
@Controller
public class indexController {

    @GetMapping("/login")
    public String login(){
        return "index";
    }
}
```

6. 语言视图解析器的设置 **Important**

这里建立了一个继承 `LocaleResolver` 类的实体类，通过这个类可以实现解析前端发送过来的请求
这里的Locale是区域的意思

```
package com.feng.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;
public class MyLanguageResolver implements LocaleResolver {
    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        String language = request.getParameter("lang");
        Locale locale = Locale.getDefault();
        if (!StringUtils.isEmpty(language)){
            String[] split = language.split("_");
            locale = new Locale(split[0],split[1]);
        }
        return locale;
    }
    @Override
    public void setLocale(HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse, Locale locale) {
    }
}
```

```
}
```

这里通过一个判断来设置是否是默认的，如果有前端请求来到后端，就启用语言视图解析器，并判断是否携带有关语言设置的信息，这个的写法在SpringBoot中有很多，要多读源码

7.语言视图解析器的注册

方式一：

直接在主启动类中注册视图解析器

```
@SpringBootApplication
public class EmployeeApplication {

    public static void main(String[] args) { SpringApplication.run(EmployeeApplication.class, args); }

    @Bean
    public LocaleResolver localeResolver(){
        return new MyLanguageResolver();
    }
}
```

方式二：

在自己配置SpringMvcConfig类中注册，这里注意继承类的名字

```
package com.feng.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.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
@Configuration
public class MySpringMvcConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
        registry.addViewController("/index.html").setViewName("index");
    }
    @Bean
    public LocaleResolver localeResolver(){
        return new MyLanguageResolver();
    }
}
```

通过上面的步骤就可实现页面国际化了

3.显示所有的员工

1.先配置对应的controller类

这里先将之前模拟数据库里的数据取了出来，也就是之前定义的static的元素，这里配置一个网页的路径，将这些数据放在model中，之后反应在html文件中

```

@Controller
public class empController {
    @Autowired
    EmployeeMapper employeeMapper;
    @Autowired
    DepartmentMapper departmentMapper;
    @RequestMapping("/emps")
    public String getAllEmployees(Model model){
        Collection<Employee> allEmployees = employeeMapper.getAllEmployees();
        model.addAttribute("emps",allEmployees);
        return "emps/list";
    }
    @GetMapping("/emp")
    public String toAddEmp(Model model){
        Collection<Department> department = departmentMapper.getAllDepartments();
        model.addAttribute("department",department);
        return "emps/addEmp";
    }
    @PostMapping("/emp")

```

2.配置对应的html页面

这里用了一种新的thymeleaf标签，`th:fragment`，`th:insert`和`th:replace`等

这里先说明一下的insert和replace的区别

- insert就是将公共片段插入到声明引入的标签中，这里加了一层渲染
- replace就是公共片段替换掉声明引入的标签，而这里没有，推荐使用这个

`th:fragment` 的用法

这里是声明一个相同的片段

```

<th:fragment="topBar">

```

`th:insert` 的用法

这里是引入commons文件夹下的commons.html中的fragment为topBar的片段

```

<th:insert="~{commons/commons::topBar}">

```

`th:replace` 的用法

这里的和上面的是一样的

```

<th:replace="~{commons/commons::topBar}">

```

这里还要实现a标签的高亮的操作

通过insert的时候，传递参数，就像上面的一样，使用括号进行传递参数，而在commons.html中要判断是否是对应的页面，从而来选择是否需要高亮

这个是传递参数

```
th:replace="~{commons/commons::sideBar(active='main.html')}"></div>
```

这个是接受参数，这里还使用了一个三元运算符来判断是否需要点击高亮显示

```
class="nav-item">
a th:class="{active=='main.html'? 'nav-link active': 'nav-link'}"
```

显示员工的时候，使用 `th:each` 来进行遍历员工

```
<tr th:each="emp:${emps}">
  <td th:text="${emp.getId()}"></td>
  <td th:text="${emp.getName()}"></td>
  <td th:text="${emp.getEmail()}"></td>
  <td th:text="${emp.getGender() == 0? '女': '男'}"></td>
  <td th:text="${emp.getDepartment().getDepartmentName()}"></td>
  <td th:text="${#dates.format(emp.getBirth(), 'yyyy-MM-dd HH:mm:ss')}"></td>
  <td class="btn btn-sm btn-primary">编辑</td>
  <td class="btn btn-sm btn-danger">删除</td>
</tr>
```

4.增加职工

1.配置controller层

下面的是这个是Restful风格的，这里的Get和Post是使用的同一个域名，但是通过提交的方式不同来实现不同的功能的实现

```
@GetMapping("/emp")
public String toAddEmp(Model model){
    Collection<Department> department = departmentMapper.getAllDepartments();
    model.addAttribute("departments", department);
    return "emps/addEmp";
}

@PostMapping("/emp")
public String addEmp(Employee employee){
    employee.setDepartment(departmentMapper.getDepartmentById(employee.getDepartment().getId()));
    employeeMapper.addEmployee(employee);
    return "redirect:/emps";
}
```

2.配置对应的html层

这里的操作就是建立一个form表单，里面放上对应的数据字段，不过要input标签中的name属性的字段名要和对应的实体类的属性名相同，这里form表单的提交的方式是post方法，这里的编号是自增长的，无需担心

这里有个select，option和th:each的结合要注意，对了还要注意这里Department的传递参数的属性名，这里是先传递编号，在后台进行操作，把对应的Department的属性完善了

```
<select class="form-control" name="department.id">
  <option th:each="dept:${departments}" th:text="${dept.getDepartmentName()}" th:value="${dept.getId()}"></option>
</select>
```


5.修改职工

0.跳转到修改页面

这里使用了一个字符串拼接的功能，这里爆红是因为idea可能不支持，但是功能实现了

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

1.建立对应的controller层

这个的操作和上面是一样的，都是Restful风格的，但是这里通过 @PathVariable 来拼接路径，前端有个操作要注意

```
@GetMapping("/toUpdate/{id}")
public String toUpdateEmp(@PathVariable("id") Integer id, Model model){
    Employee employee = employeeMapper.getEmployeeById(id);
    model.addAttribute("emp", employee);
    Collection<Department> allDepartments = departmentMapper.getAllDepartments();
    model.addAttribute("departments", allDepartments);
    return "emps/updateEmp";
}

@PostMapping("/updateEmp")
public String updateEmp(Employee employee){
    employee.setDepartment(departmentMapper.getDepartmentById(employee.getDepartment().getId()));
    employeeMapper.addEmployee(employee);
    return "redirect:/emps";
}
```

2.建立对应的html界面

这里的html界面和上面的添加的页面是差不多的，只是这里提前将数据给显示出来的，方便修改，所以这里需要后端将数据传递上来，也就是Model传递参数

还有对应的html标签，其对应确定选值的方式也不同，这里需要html方面的知识

select和option

```
<select class="form-control" name="department.id">
    <option th:selected="${emp.getDepartment().getId()==dept.getId()}" th:each="dept:${departments}" th:text="${dept.getDepartmentName()}"
</select>
```

input的radio

```
<input th:checked="${emp.gender==1}" class="form-check-input" type="radio" name="gender" value="1">
```

input的其他属性

```
<input th:value="${emp.email}" type="email" name="email" class="form-control" placeholder="2560965468@qq.com">
```

其他的就要自己寻找了

6.删除职工

0.跳转到删除网址

这里和上面一样，有个字符串拼接的功能实现

```
<a class="btn btn-sm btn-danger" th:href="@{/deleteEmp/}+${emp.id}">删除</a>
```

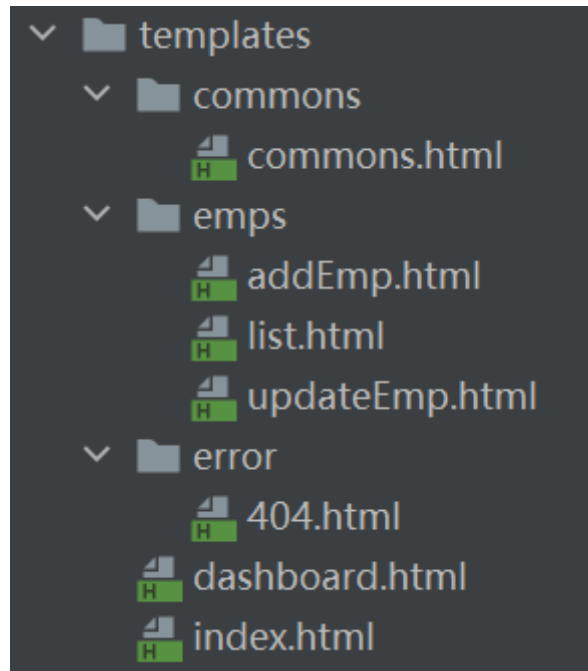
1.建立controller层

这个和上面的一样

```
@GetMapping("/deleteEmp/{id}")
public String delete(@PathVariable("id") Integer id){
    employeeMapper.deleteEmployeeById(id);
    return "redirect:/emps";
}
```

7.404界面

这里再resources下的template文件中建立一个error文件夹，将需要的404界面放进去，就可以实现自动注册404界面的功能，其他500等都是这要操作的



8.注销用户

这个的操作就将用户的Session给消除掉

```
@RequestMapping("/logout")
public String logout(HttpSession session){
    session.removeAttribute(name: "loginSession");
    return "index";
}
```

9.登录操作

1.前端的界面

通过前端的index.html来实现登录操作，这里的name的名字是十分重要的，要和后端传递的参数要相同，但是也可以不同，可以通过 `@RequestParam("参数名")` 来修改参数的名字

```

<form class="form-signin" th:action="@{/login}" method="post">
  
  <h1 class="h3 mb-3 font-weight-normal" th:text="{login.tip}">Please sign in</h1>
  <p style="color: red" th:text="{msg}" th:if="{not #strings.isEmpty(msg)}"></p>
  <input type="text" name="username" class="form-control" th:placeholder="{login.username}" required="" autofocus="">
  <input type="password" name="password" class="form-control" th:placeholder="{login.password}" required="">
  <div class="checkbox mb-3">
    <label>
      <input type="checkbox" th:text="{login.remember}">
    </label>
  </div>
  <button class="btn btn-lg btn-primary btn-block" type="submit" th:text="{login.btn}"></button>
  <p class="mt-5 mb-3 text-muted">© 2017-2018</p>
  <a class="btn btn-sm" th:href="@{/toLogin(lang='zh_CN')}">中文</a>
  <a class="btn btn-sm" th:href="@{/toLogin(lang='en_US')}">English</a>
</form>

```

2.controller层

这里只要登录了之后就会在浏览器中留下一个Session来验证是否已经登录了

```

@Controller
public class indexController {
    @GetMapping("/toLogin")
    public String toLogin() { return "index"; }
    @PostMapping("/login")
    public String login(@RequestParam("username") String username, @RequestParam("password") String password, Model model) {
        if("feng".equals(username)&&"123456".equals(password)){
            session.setAttribute( name: "loginSession",username);
            return "redirect:main.html";
        }else{
            model.addAttribute( attributeName: "msg", attributeValue: "密码或用户名出现错误");
        }
        return "index";
    }
}

```

9.拦截器 Important

SpringBoot的拦截器和上面的那个实现国际的Locale一样需要自定义并且注册

这里的拦截器和SpringMVC中的是一样的操作，只是这里的功能很多，也更加人性化一点

同样这里的拦截器实现继承了 `HandlerInterceptor`，并重写其中的方法就行了

这里的拦截器是用来拦截没有登录时，直接进入主页的操作

```

package com.feng.config;
import org.springframework.web.servlet.HandlerInterceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class MyInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        HttpSession session = request.getSession();
        Object loginSession = session.getAttribute("loginSession");
        if(loginSession==null){
            request.setAttribute("msg","你没有权限，请先登录");
        }
        request.getRequestDispatcher("index.html").forward(request, response);
        return false;
    }else{
        return true;
    }
}
}

```

之前登录的操作给浏览器留下了一个Session，通过这个来判断是否已经登录过，来实现拦截的操作之后再MySpringMvcConfig中注册

这里的MySpringMvcConfig文件中需要重写一个方法，也就是 `addInterceptors` 方法，这里要将自己的interceptor加入到Config中

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
    MyInterceptor()).addPathPatterns("/*").excludePathPatterns("/toLogin","/index.html","/login","/","/css/*","/js/*","/img/*");
}
```

这个方法还贴心的帮我们实现了增加拦截的请求网址路径，还删除了特定的请求网址路径，十分的完美
以上就是职工管理系统的完整步骤开发了，结束了，好耶！

9.JDBC

1.SpringBoot的CRUD

SpringBoot中集成了 `JdbcTemplate`，这个类十分的nb，这个类中集成了spl的很多方法，可以直接使用，里面有很多的重写的方法

```
@RestController
public class JDBCController {
    @Autowired
    JdbcTemplate jdbcTemplate;
    @GetMapping("/getUsers")
    public List<Map<String, Object>> getUsers(){
        String sql = "select * from user";
        List<Map<String, Object>> maps = jdbcTemplate.queryForList(sql);
        return maps;
    }
    @GetMapping("/addUser/{id}/{username}/{age}")
    public String addUser(@PathVariable("id")Integer id,@PathVariable("username")String name,@PathVariable("age") Integer age){
        String sql = "insert into user(id,name,age)
values("+id+", '"+name+"',"+age+")";
        jdbcTemplate.update(sql);
        return "addUserok";
    }
    @GetMapping("/updateUser/{id}/{username}/{age}")
    public String updateUser(@PathVariable("id")Integer id,@PathVariable("username")String name,@PathVariable("age") Integer age){
        String sql = "update user set name='"+name+"',"+ " age="+age+"
where id="+id;
        jdbcTemplate.update(sql);
        return "updateUserok";
    }
    @GetMapping("/deleteUser/{id}")
    public String deleteUser(@PathVariable("id") Integer id){
        String sql = "delete from user where id=?";
        jdbcTemplate.update(sql,id);
        return "deleteok";
    }
}
```

```
}  
}
```

这个代码使用了Restful风格来添加数据等，sql语句的拼接十分重要，这一步是关键，在这个代码中有两种方法来实现sql功能，一是：字符串的拼接，二是：经典的jdbc的操作

10.整合Druid数据源

1.先导入maven依赖

```
<dependency>  
    <groupId>com.alibaba</groupId>  
    <artifactId>druid</artifactId>  
    <version>1.2.8</version>  
</dependency>  
<dependency>  
    <groupId>com.alibaba</groupId>  
    <artifactId>druid-spring-boot-starter</artifactId>  
    <version>1.2.8</version>  
</dependency>
```

2.建立对应的config类

```
@Configuration  
public class DruidConfig{  
    @Bean  
    @ConfigurationProperties(prefix = "spring.datasource")  
    public DataSource druidDataSource(){  
        return new DruidDataSource();  
    }  
    @Bean  
    public ServletRegistrationBean statViewServlet(){  
        ServletRegistrationBean<StatViewServlet> bean = new  
ServletRegistrationBean<>(new StatViewServlet(), "/druid/*");  
        LinkedHashMap<String, String> initParameters = new LinkedHashMap<>();  
        initParameters.put("loginUsername", "feng");  
        initParameters.put("loginPassword", "123456");  
        initParameters.put("allow", "");  
        //initParameters.put("deny", "localhost");  
        initParameters.put("resetEnable", "false");  
        bean.setInitParameters(initParameters);  
        return bean;  
    }  
    @Bean  
    public FilterRegistrationBean statViewFilter(){  
        FilterRegistrationBean<StatViewFilter> bean = new  
FilterRegistrationBean<>(new StatViewFilter());  
        bean.addUrlPatterns("/");  
  
        bean.addInitParameter("exclusions", "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*  
,"");  
        return bean;  
    }  
}
```

这里的参数有些是固定的，不能更改，这个是从源代码中得到的

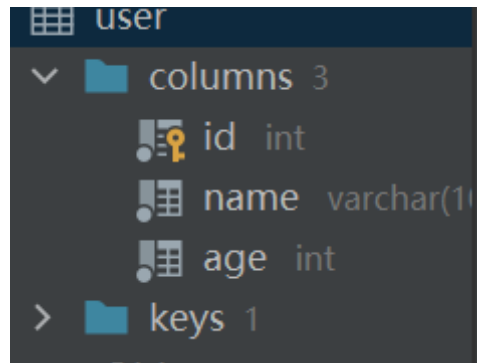
3.网址访问

这个是直接访问的界面

) localhost:8080/druid/login.html

11.整合Mybatis框架

0.建立相关的数据库



1.建立相关的实体类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int id;
    private String name;
    private int age;
}
```

2.建立相关的Mapper接口

这里的@Mapper是注解这个是一个Mapper接口类，还可以不通过这个方法来实现这个功能

在主启动类中加入 @MapperScan("com.feng.mapper") 这句话，就会扫描这个包下的所有注解

```

@Mapper
@Repository
public interface UserMapper {
    List<User> getAllUsers();
    User getUser(int id);
    void addUser(User user);
    void updateUser(User user);
    void deleteUser(int id);
}

```

3.建立对应的xml文件

这个xml文件没有什么大的改变

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.mapper.UserMapper">
    <select id="getAllUsers" resultType="com.feng.pojo.User">
        select * from user;
    </select>
    <select id="getUser" resultType="com.feng.pojo.User">
        select * from USER where id=#{id};
    </select>
    <insert id="addUser" parameterType="com.feng.pojo.User">
        insert into USER(id,name,age) values("#{id},#{name},#{age});
    </insert>
    <update id="updateUser" parameterType="com.feng.pojo.User">
        update user set name=#{name},age=#{age} where id=#{id};
    </update>
    <delete id="deleteUser">
        delete from USER where id=#{id};
    </delete>
</mapper>

```

4.注册对应的mapper位置

在yaml文件中表明文件的位置，当然这里还可以配置多个属性，和上面的Druid一样

```
-mybatis:
-  mapper-locations: classpath:Mybatis/Mapper/*.xml
```

5.对应的controller层

其实这个应该是service层的工作，但是是练手就直接写在了controller层了

这个就相比上面拼接字符串好的多

```
@RestController
public class MyBatisController {
    @Autowired
    UserMapper userMapper;
    @GetMapping("/getAllUsers")
    public List<User> getAllUsers(){
        List<User> allUsers = userMapper.getAllUsers();
        return allUsers;
    }
    @GetMapping("/getUser/{id}")
    public User getUser(@PathVariable("id") int id){
        User user = userMapper.getUser(id);
        return user;
    }
    @GetMapping("/addUser/{id}/{name}/{age}")
    public void addUser(@PathVariable("id") int id,@PathVariable("name") String
name,@PathVariable("age") int age){
        User user = new User(id, name, age);
        userMapper.addUser(user);
    }
    @GetMapping("/updateUser/{id}/{name}/{age}")
    public void updateUser(@PathVariable("id") int id,@PathVariable("name")
String name,@PathVariable("age") int age){
        User user = new User(id, name, age);
        userMapper.updateUser(user);
    }
    @GetMapping("/deleteUser/{id}")
    public void deleteUser(@PathVariable("id") int id){
        userMapper.deleteUser(id);
    }
}
```

这里依旧是Restful风格的实现，但是不需要拼接字符串了，直接调用方法

12.SpringSecurity

1.建立对应的controller层

这个就是建立一个链接，这个也是Restful风格的，字符串拼接功能


```

@Controller
public class SecurityController {
    @RequestMapping({"/", "/index.html", "index"})
    public String index() { return "index"; }
    @RequestMapping("/toLogin")
    public String login() { return "views/login"; }
    @RequestMapping("/level1/{id}")
    public String level1(@PathVariable("id") int id){
        return "views/level1/"+id;
    }
    @RequestMapping("/level2/{id}")
    public String level2(@PathVariable("id") int id){
        return "views/level2/"+id;
    }
    @RequestMapping("/level3/{id}")
    public String level3(@PathVariable("id") int id){
        return "views/level3/"+id;
    }
}

```

2.建立对应的security层，这是个配置层

这个要继承 `WebSecurityConfigurerAdapter`，然后重写其中的方法，这个是链式编程，这其中的很多方法有很多不知道的，要理解，这里的 `@EnableWebSecurity` 是承认这是一个安全层的配置

这里改了原来的name属性，之后再config层上也要改动

```

<div class="ui left icon input">
    <input type="text" placeholder="Username" name="name">
    <i class="user icon"></i>
</div>
div>
iv class="field">
    <label>Password</label>
    <div class="ui left icon input">
        <input type="password" name="pwd">

```

```

@EnableWebSecurity
public class UserSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .antMatchers("/").permitAll()
            .antMatchers("/level1/**").hasRole("vip1")
            .antMatchers("/level2/**").hasRole("vip2")
            .antMatchers("/level3/**").hasRole("vip3");
    }
}

```

```

http.formLogin().loginPage("/toLogin").usernameParameter("name").passwordParameter("pwd").loginProcessingUrl("/login");
//自定义属性的名字和登录界面
http.csrf().disable();
//SpringSecurity的防御措施
http.logout().logoutSuccessUrl("/");
//退出后的进的界面自定义
http.rememberMe().rememberMeParameter("remember");
//设置remember的属性，来记住之前登录的账号
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth.inMemoryAuthentication().passwordEncoder(new
BCryptPasswordEncoder())//编码
        .withUser("feng").password(new
BCryptPasswordEncoder().encode("123456")).roles("vip1", "vip2", "vip3").and()
        .withUser("wrq").password(new
BCryptPasswordEncoder().encode("123456")).roles("vip1");
}
}

```

3.对应的html文件

要使用这些注解的前提要导入包

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
    <version>3.0.4.RELEASE</version>
</dependency>

```

注解提示的语句

```
xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
```

其中有很多的注解，方法

这个是否认证了，就是可以控制元素的可见或不可见

```
sec:authorize="!isAuthenticated()">
```

这个也是一样的，就是你登陆的账号是否有这个身份，来显示你的界面

```
sec:authorize="hasRole('vip1')">
```

13.Shiro

1.导入相关的依赖

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.8.0</version>
</dependency>
```

2.建立相关的配置类

1.ShiroConfig

```
@Configuration
public class ShiroConfig {
    @Bean
    public ShiroFilterFactoryBean
getShiroFilterFactoryBean(@Qualifier("SecurityManager")
DefaultWebSecurityManager defaultManager){
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
ShiroFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(defaultManager);
        Map<String, String> LinkedHashMap = new LinkedHashMap<>();
        //授权，正常情况下，没有授权会跳转到为授权页面
        LinkedHashMap.put("/user/addUser", "perms[add]");
        LinkedHashMap.put("/user/updateUser", "perms[update]");
        LinkedHashMap.put("/user/*", "authc");
        shiroFilterFactoryBean.setFilterChainDefinitionMap(LinkedHashMap);
        shiroFilterFactoryBean.setLoginUrl("/toLogin");
        shiroFilterFactoryBean.setUnauthorizedUrl("/noauth");
        return shiroFilterFactoryBean;
    }
    @Bean("SecurityManager")
    public DefaultWebSecurityManager
getDefaultWebSecurityManager(@Qualifier("realm") UseRealm useRealm){
        DefaultWebSecurityManager SecurityManager = new
DefaultWebSecurityManager();
        SecurityManager.setRealm(useRealm);
        return SecurityManager;
    }
    @Bean(name="realm")
    public UseRealm useRealm(){
        return new UseRealm();
    }
    @Bean
    public ShiroDialect getShiroDialect(){
        return new ShiroDialect();
    }
}
```

在这个类中，建立的subject，realm和SecurityManager的实现体，通过一层一层的向上传递来实现Shiro的搭建

首先注意这个@Bean的参数，这个@Bean可以指定参数的名字，在其他类使用的时候就可以直接调用这个参数来实现调用这个Spring托管的类，当然这个还要配合@Qualifier的指定特定的名字来进行注册来使用

其次注意ShiroFilterFactoryBean中的LinkedHashMap的参数

- anno: 无需认证就可以访问
- authc: 必须认证了才能访问
- user: 必须拥有记住我功能才能使用
- perms: 拥有对某个资源的权限才能使用
- role: 拥有某个角色权限
- logout: 推出拦截器, 也就是注销的意思, 但是我在实验的过程中, 没有实现页面跳转的功能, 所以不使用这样的方式注销账户

```
LinkedHashMap.put("/user/addUser", "perms[add]");
LinkedHashMap.put("/user/updateUser", "perms[update]");
LinkedHashMap.put("/user/*", "authc");
```

这个还有先后顺序之分, 我在实验的时候件认证放在前面直接不走授权的功能, 所以要注意这里的顺序

2.UserRealm

```
public class UserRealm extends AuthorizingRealm {
    @Autowired
    UserServiceImpl userService;
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
        SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
        Subject subject = SecurityUtils.getSubject();
        User user = (User) subject.getPrincipal();
        info.addStringPermission(user.getPerms());
        return info;
    }

    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
authenticationToken) throws AuthenticationException {
        UsernamePasswordToken token = (UsernamePasswordToken)
authenticationToken;
        User user = userService.queryUserByName(token.getUsername());
        //        Subject subject = SecurityUtils.getSubject();
        //        Session session = subject.getSession();
        //        session.setAttribute("loginUser", user);
        if (user == null) {
            return null;
        }
        return new SimpleAuthenticationInfo(user, user.getPassword(), "");
    }
}
```

这里操作都是死的, 这里的密码是没有通过加密的, 而且这里的密码的判断是通过Shiro判断的, 所以是不需要手动判断, 这里的判断密码是直接通过最后的一个语句段实现的, 当然这里还连接了数据库的查询功能, 所以要实现数据库的链接, 下面的return null代表的是这个是有误的, 通过Shiro的调度传递到了controller层, 将对应的信息传递出去

3.数据库的链接

1.导入依赖

导入Druid和mybatis的依赖包来实现这个功能

```

<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.2.8</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-starter</artifactId>
  <version>1.2.8</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.2.2</version>
</dependency>

```

2.编写驱动

驱动的连接和Druid的配置，当然还有mapper的地址指明

```

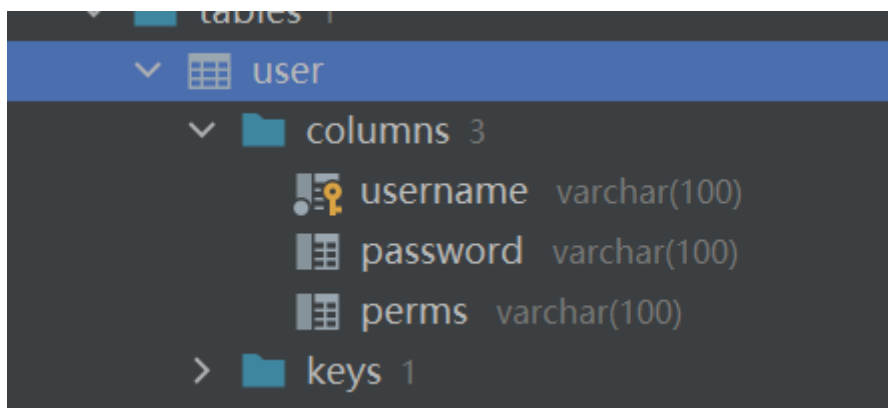
spring:
  mvc:
    favicon:
      enable: false
  datasource:
    username: root
    password: root
    url: jdbc:mysql://localhost:3306/user?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    driver-class-name: com.mysql.cj.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
    #Spring Boot 默认是不注入这些属性值的，需要自己绑定
    #druid 数据源专有配置
    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，stat:监控统计、log4j: 日志记录、wall: 防御sql注入
    #如果允许时报错 java.lang.ClassNotFoundException: org.apache.log4j.Priority
    #则导入 log4j 依赖即可，Maven 地址：
https://mvnrepository.com/artifact/log4j/log4j
    filters: stat,wall
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
mybatis:

```

```
mapper-locations: classpath:mybatis/mapper/*.xml
```

2.编写对应的数据库类

这里username是主键，perms是授权管理的信息



3.编写对应的pojo类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private String username;
    private String password;
    private String perms;
}
```

4.编写对应的mapper类

```
@Mapper
@Repository
public interface UserMapper {
    User queryUserByName(String name);
}
```

5.编写对应的service层

service层可以加入其他的功能，在这层上加人相应的功能可以避免改动底层的代码，直接实现相应的功能

接口service层

```
public interface UserService {
    User queryUserByName(String name);
}
```

接口service层的实现层

这里的@Service是和@Component的功能是一样的，只是术业有专攻

```

@Service
public class UserServiceImpl implements UserService{
    @Autowired
    UserMapper userMapper;
    @Override
    public User queryUserByName(String name) {
        return userMapper.queryUserByName(name);
    }
}

```

6.编写对应的mapper.xml文件

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.mapper.UserMapper">
    <select id="queryUserByName" resultType="com.feng.pojo.User">
        select * from user where username=#{name};
    </select>
</mapper>

```

通过上述的步骤之后就可以实现通过数据库存值的功能来实现验证用户是否已经存在了

3.建立对应的controller层

在这个controller层中，实现了shiro的注销功能，就和SpringSecurity的功能是一样的

```

@Controller
public class ShiroController {
    @RequestMapping({"/", "/index", "/index.html"})
    public String index(){
        return "index";
    }
    @RequestMapping("/user/addUser")
    public String addUser(){
        return "User/add";
    }
    @RequestMapping("/user/updateUser")
    public String updateUser(){
        return "User/update";
    }
    @RequestMapping("/toLogin")
    public String toLogin(){
        return "login";
    }
    @RequestMapping("/login")
    public String login(String name, String pwd, Model model){
        Subject subject = SecurityUtils.getSubject();
        UsernamePasswordToken token = new UsernamePasswordToken(name, pwd);
        try{
            subject.login(token);
            return "index";
        }catch (UnknownAccountException e){

```

```

        model.addAttribute("msg", "用户名出错");
        return "login";
    } catch (IncorrectCredentialsException e) {
        model.addAttribute("msg", "密码错误");
        return "login";
    }
}

@RequestMapping("/noauth")
@ResponseBody
public String NoAuth() {
    return "该界面未经授权";
}

@RequestMapping("/logout")
public String logout() {
    Subject subject = SecurityUtils.getSubject();
    subject.logout();
    return "index";
}
}

```

4.对应的前端画面

0.导入依赖

这个是thymeleaf和shiro的结合依赖包

```

<dependency>
    <groupId>com.github.theborakompanioni</groupId>
    <artifactId>thymeleaf-extras-shiro</artifactId>
    <version>2.1.0</version>
</dependency>

```

1.界面的配置

这个页面是index.html，也就是首页，这里集成了Shiro的命名空间和thymeleaf的命名空间，这里还是实现了对应的展示显示界面的设置，就是登录之后，登录键消失，不同的账号，有不同的显示权限等

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
    xmlns:shiro="http://www.pollix.at/thymeleaf/shiro">
    <head>
        <meta charset="UTF-8">
        <title>Title</title>
    </head>
    <body>
        <h1>首页</h1>
        <div shiro:notAuthenticated>
            <a th:href="@{/toLogin}">登录</a>
        </div>
        <!--<div th:if="${session.loginUser==null}">-->
        <!--    <a th:href="@{/toLogin}">登录</a>-->
        <!--</div>-->
        <div shiro:hasPermission="add">
            <a th:href="@{/user/addUser}">增加</a>
        </div>
        <div shiro:hasPermission="update">
            <a th:href="@{/user/updateUser}">更新</a>
        </div>
    </body>
</html>

```



```

    </div>
    <br>
    <a th:href="@{/logout}">注销</a>
  </body>
</html>

```

之后就是对应的add, update, login界面了

```

<h1>增加</h1>
<a th:href="@{/logout}">注销</a>

```

```

<h1>更新</h1>
<a th:href="@{/logout}">注销</a>

```

```

<p th:text="${msg}" style="color: red"></p>
<form th:action="@{/login}" method="post">
  用户名: <input type="text" name="name">
  密码: <input type="text" name="pwd">
  <input type="submit" value="提交">
</form>

```

这节课的详细解释: [Shiro](#)

14. Swagger Swagger

1. 导入Swagger的依赖

```

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

```

注意 yaml或者properties文件中要配置这么一句话来开启Swagger

```
spring.mvc.pathmatch.matching-strategy=ant_path_matcher
```

```

spring:
  mvc:
    pathmatch:
      matching-strategy: ant_path_matcher

```

还有这里的网址是：

```
http://localhost:8080/swagger-ui.html
```

2.配置对应的config类

```
@Configuration
@EnableSwagger2    //开启Swagger类
public class SwaggerConfig {
    //配置文档信息
    private ApiInfo apiInfo() {
        Contact contact = new Contact("feng", "https://www.baidu.com",
"2560965468@qq.com");
        return new ApiInfo(
            "Swagger学习", // 标题
            "学习演示如何配置Swagger", // 描述
            "v1.0", // 版本
            "http://terms.service.url/组织链接", // 组织链接
            contact, // 联系人信息
            "Apach 2.0 许可", // 许可
            "许可链接", // 许可连接
            new ArrayList<>()// 扩展
        );
    }
    @Bean
    public Docket docket(Environment environment) {
        Profiles of = Profiles.of("dev", "test");
        boolean b = environment.acceptsProfiles(of);
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .enable(false)
            .groupName("feng")
            .select()

        .apis(RequestHandlerSelectors.withClassAnnotation(Configuration.class))
            .paths(PathSelectors.ant("/feng/**"))
            .build();
    }
    @Bean
    public Docket docket1(){
        return new Docket(DocumentationType.SWAGGER_2)
            .enable(true)
            .groupName("A");
    }
    @Bean
    public Docket docket2(){
        return new Docket(DocumentationType.SWAGGER_2)
            .groupName("B");
    }
    @Bean
    public Docket docket3(){
        return new Docket(DocumentationType.SWAGGER_2)
            .groupName("C");
    }
}
```

这里的函数

- apiinfo是配置用户信息的
- enable是配置swagger是否开启的，这个可以通过开发环境的选择来实现
- groupname是分组名，也就是多人开发时的接口，这里要实现分组，只需要多建几个Docket就行了，加上不同的名字
- select是配置下面路径扫描的函数开启头
- apis里面是配置扫描的controller接口的，就是使什么接口控制放在swagger上
- paths是配置路径请求的，就是只允许里面参数中的请求允许放在swagger上
- build就是建立select之后的结束语

其中路径配置的函数还有一些

下面的这些参数都是在RequestHandlerSelectors的参数

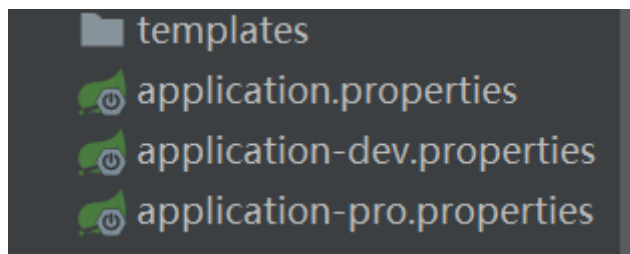
- basePackage(final String basePackage) 根据包路径扫描对应的接口
- any() 扫描所有接口，项目中的所有接口控制都会被扫描到
- none() 不扫描接口
- withMethodAnnotation(final Class<? extends Annotation> annotation) 这个只扫描请求方式的注解
- withClassAnnotation(final Class<? extends Annotation> annotation) 这个只扫描类上的注解

接下来的就是配置paths的函数

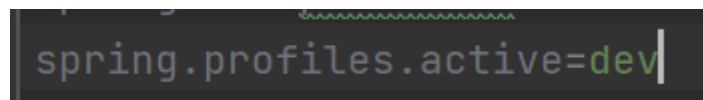
- any() 任何请求都识别
- none() 任何请求都不识别
- regex(final String pathRegex) 通过正则表达式控制
- ant(final String antPattern) 通过ant控制，这个控制的url请求地址

接下来就是配置开发环境了，来间接控制swagger的开关

application.properties的配置



主文件中配置的是dev，再通过上面的enable来开启或关闭，通过这个还可以控制那些人可以使用SwaggerAPI，就是通过enable的来控制



module的配置

这里只需要在controller中加上对应的请求就行了

这里配置对应的实体类和对应的controller

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@ApiModel("用户类")
public class User {
    @ApiModelProperty("用户名")
    private String username;
    @ApiModelProperty("密码")
    private String password;
}

```

```

@ApiOperation("User返回类")
@RequestMapping(value = "/user")
public User User(){
    return new User();
}

```

这样之后module中就会出现user类

Swagger的注释

- @Api(tags={"用户接口"}) 这个可以标记controller类作为Swagger的文档资源
- @ApiModel(value description) 这个作用在实体类上，起解释的作用
- @ApiModelProperty(value description) 这标记实体类中参数的含义
- @ApiParam(value required) 这个标记controller中参数的含义
- @ApiOperation(value notes) 这个是标记controller中方法的含义
- @ApiResponses({ @ApiResponse(code = 404, message = "错误了", response = User.class) })
这个是配置状态码的信息的

15.异步，定时，邮件任务 异步、定时、邮件任务

1.异步

这个不需要导入包，直接进行

2.建立对应的service层

这里的注解@service的作用是和@Component的作用是一样的，给Spring托管的，然后对应的@Async是使这个类成为异步托管类

```

@Async
@Service
public class AsyncService {
    public void Hello(){
        try {
            Thread.sleep( millis: 3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("你被执行了");
    }
}

```

在这个基础上，还要在SpringBoot启动层上加上 `@EnableAsync`

3.建立对应的controller层进行测试

这里的@Autowired直接引入对应的服务层中的@Bean实例，这里的运行后果是页面等了3秒后出现了，经过上面的操作，成功使得任务异步了，就是页面直接出现，输出在后面出现，就不需要在等了

```

@RestController
public class AsyncController {
    @Autowired
    AsyncService asyncService;
    @RequestMapping("/hello")
    public String Hello(){
        asyncService.Hello();
        return "hello";
    }
}

```

2.邮件传输

1.导入相关的依赖

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

在这个文件中里有javax.mail文件，也就是mail发送的依赖

2.配置对应的properties或者yaml文件

```
spring.mail.username=2560965468@qq.com
spring.mail.password=授权码
spring.mail.host=smtp.qq.com
spring.mail.properties.mail.smtp.ssl.enable=true
spring.mail.properties.mail.smtp.starttls.enable=false
spring.mail.properties.mail.smtp.starttls.required=false
```

yaml文件直接这样对着写

注意 这里的密码由于腾讯的原因，需要开启smtp和pop3的协议，所以这里的授权码需要自己去找

3.配置对应的测试类

```
@SpringBootTest
class AsyncApplicationTests {
    @Autowired
    JavaMailSenderImpl javaMailSender;
    @Test
    void contextLoads() {
        SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
        simpleMailMessage.setSubject("你好");
        simpleMailMessage.setText("你好");
        simpleMailMessage.setTo("2560965468@qq.com");
        simpleMailMessage.setFrom("2560965468@qq.com");
        javaMailSender.send(simpleMailMessage);
    }
    @Test
    void contextLoads1() throws MessagingException {
        MimeMessage mimeMessage = javaMailSender.createMimeMessage();
        MimeMessageHelper mimeMessageHelper = new MimeMessageHelper(mimeMessage,
true);
        mimeMessageHelper.setSubject("测试");
        mimeMessageHelper.setText("<p style='color:red;font-size:30px;'>大佬你的数  
模作业写了没</p>",true);
        mimeMessageHelper.addAttachment("1.png",new
File("C:\\Users\\feng\\Desktop\\2.png"));
        mimeMessageHelper.setTo("2560965468@qq.com");
        mimeMessageHelper.setFrom("2560965468@qq.com");
        javaMailSender.send(mimeMessage);
    }
}
```

在这个测试类中，实现了简单邮件发送和复杂邮件发送，方法都是可以直接抄的，还可以封装成service层的代码，用controller来控制发送

当然还可实现邮件轰炸了，嘿嘿嘿！！！！

3.定时任务

1.开启定时任务

- `@Scheduled(cron = "0/10 * * * * ?")` 在相关的service类中加上的注解
- `@EnableScheduling` 在SpringBoot的启动类上加的注解

2.学习cron表达式 [在线Cron表达式生成器](#)

corn表达式

| 字段 | 允许值 | 允许特殊字符 |
|------|---------------------|-----------------|
| • 秒 | 0-59 | , - * / |
| • 分 | 0-59 | , - * / |
| • 小时 | 0-23 | , - * / |
| • 日期 | 1-31 | , - * / ? L W C |
| • 月份 | 1-12 | , - * / |
| • 星期 | 0-7或SUN-SAT 0,7是SUN | , - * / ? L W C |

特殊字符

| | |
|-----|------------|
| • , | 枚举 |
| • - | 区间 |
| • * | 任意 |
| • / | 步长 |
| • ? | 日/星期冲突匹配 |
| • L | 最后 |
| • W | 工作日 |
| • C | 和日志联系后计算的值 |
| • # | 星期 |

corn表达式对应的参数

```
corn("秒 分 时 日 月 周几")
```