第 5-8 课：综合实战客户管理系统（二）

客户管理系统需要考虑验证用户的注册邮箱是否正确，使用 Filter 来判断用户的登录状态是否已经启用，以及在项目中缓存的使用，如何使用 Thymeleaf 的最新语法判断表达式对页面布局，最后讲解使用 Docker 部署客户管理系统。

邮箱验证

我们希望用户注册的邮箱信息是正确的，因此会引入邮件验证功能。注册成功后会给用户发送一封邮件，邮件中会有一个关于用户的唯一链接，当单击此链接时更新用户状态，表明此邮箱即为用户真正使用的邮箱。

首先需要定义一个邮件模板，每次用户注册成功后调用模板进行发送。

邮件模板

```
<!DOCTYPE html>
<html lang="zh" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8"/>
        <title>邮件模板</title>
    </head>
    <body>
        您好，感谢您的注册，请您尽快对注册邮件进行验证，请点击下方链接完成，感谢您的支持！<br/>
        <a href="#" th:href="@{http://localhost:8080/verified/{id}(id=${id}) }">激活账
号</a>
    </body>
</html>
```

id 为用户注册成功后生成的唯一标示，每次动态替换。

效果图如下：

发送邮件

```
public void sendRegisterMail(UserEntity user) {
    Context context = new Context();
    context.setVariable("id", user.getId());
    String emailContent = templateEngine.process("emailTemplate", context);
    MimeMessage message = mailSender.createMimeMessage();
    try {
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setFrom(from);
        helper.setTo(user.getEmail());
        helper.setSubject("注册验证邮件");
        helper.setText(emailContent, true);
        mailSender.send(message);
    } catch (Exception e) {
        logger.error("发送注册邮件时异常！", e);
    }
}
```

上面代码封装了邮件发送的内容，注册成功后调用即可。

邮箱验证

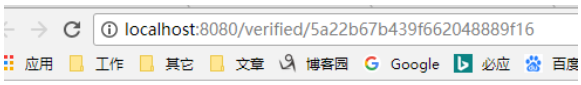
当用户单击链接时请求 verified() 方法，将用户的状态改为：verified，表明邮箱已经得到验证。

```
@RequestMapping("/verified/{id}")
public String verified(@PathVariable("id") String id,ModelMap model) {
    UserEntity user=userRepository.findById(id);
    if (user!=null && "unverified".equals(user.getState())){
        user.setState("verified");
        userRepository.save(user);
        model.put("userName",user.getUserName());
    }
    return "verified";
}
```

验证成功后，在页面中给出提示：

```
<h1>注册邮箱验证</h1>
<br/><br/>
<div class="with:60%">
    <h3 th:if="${userName!=null}">邮箱验证成功，<a href="/toLogin">请登录！</a></h3>
    <h3 th:if="${userName==null}">邮箱已经验证或者参数有误，请重新检查！<a href="/toRegiste
r">去注册</a></h3>
</div>
```

效果图如下：



注册邮箱验证

邮箱验证成功，请登录！

Redis 使用、自定义 Filter

Redis

Session 管理

使用 Redis 管理 Session 非常简单，只需在配置文件中指明 Session 使用 Redis，配置其失效时间。

```
spring.session.store-type=redis
# 设置 session 失效时间
spring.session.timeout=3600
```

数据缓存

为了避免用户列表页每一次请求都会查询数据库，可以使用 Redis 作为数据缓存。只需要在方法头部添加一个注解即可，如下：

```
@RequestMapping("/list")
@Cacheable(value="user_list")
public String list(Model model,@RequestParam(value = "page", defaultValue = "0") Inte
ger page,
                   @RequestParam(value = "size", defaultValue = "6") Integer size) {
       //方法内容
    return "user/list";
}
```

自定义 Filter

我们需要自定义一个 Filter，来判断每次请求的时候 Session 是否失效，同时排除一些不需要验证登录状态的 URL。

启动时初始化白名单 URL 地址，如注册、登录、验证等。

```
// 将 GreenUrlSet 设置为全局变量，在启动时添加 URL 白名单
private static Set<String> GreenUrlSet = new HashSet<String>();
...
//不需要 Session 验证的 URL
@Override
public void init(FilterConfig filterconfig) throws ServletException {
    GreenUrlSet.add("/toRegister");
    GreenUrlSet.add("/toLogin");
    GreenUrlSet.add("/login");
    GreenUrlSet.add("/loginOut");
    GreenUrlSet.add("/register");
    GreenUrlSet.add("/verified");
}
...
//判断如果在白名单内，直接跳过
if (GreenUrlSet.contains(uri) || uri.contains("/verified/")) {
        log.debug("security filter, pass, " + request.getRequestURI());
        filterChain.doFilter(srequest, sresponse);
        return;
    }
...
```

uri.contains("/verified/") 表示 URL 含有 /verified/ 就会跳过验证。

同时 Filter 中也会过滤静态资源：

```
if (uri.endsWith(".js")
        || uri.endsWith(".css")
        || uri.endsWith(".jpg")
        || uri.endsWith(".gif")
        || uri.endsWith(".png")
        || uri.endsWith(".ico")) {
    log.debug("security filter, pass, " + request.getRequestURI());
    filterChain.doFilter(srequest, sresponse);
    return;
}
```

Session 验证：

```
String id=(String)request.getSession().getAttribute(WebConfiguration.LOGIN_KEY);
if(StringUtils.isBlank(id)){
    String html = "<script type=\"text/javascript\">window.location.href=\"/toLogin\"
</script>";
    sresponse.getWriter().write(html);
}else {
    filterChain.doFilter(srequest, sresponse);
}
```

判断 Session 中是否存在用户 ID，如果存在表明用户已经登录，如果不存在跳转到用户登录页面。

这样 Session 验证就完成了。

页面布局

现在需要在用户登录后的所有页面中添加版权信息，部分页面的头部添加一些提示信息，这时候就需要引入页面布局，否则每个页面都需要单独添加，当页面越来越多的时候容出错，使用 Thymeleaf 的片段表达式可以很好的解决这类问题。

我们首先可以抽取出公共的页头和页尾。

页头

```
<header th:fragment="header">
    <div style="float: right;margin-top: 30px">
        <div style="font-size: large">欢迎登录， <text th:text="${session.LOGIN_SESSION
_USER.getUserName()}" ></text>
            !  <a href="/loginOut" th:href="@{/loginOut}" style="font-size: small">退
出</a>
        </div>
        <div th:if="${session.LOGIN_SESSION_USER.getState()=='unverified'}"  style="c
olor: red">请尽快验证您的注册邮件!</div>
    </div>
</header>
```

根据上面代码可以看出页头做了以下几个事情：

- 用户登录后给出欢迎信息
- 提供用户退出链接
- 如果用户邮箱未验证给出提示，让用户尽快验证注册邮箱。

页尾

```
<footer th:fragment="footer">
    <p style="color: green;margin: 60px;float: right">© 2018-2020 版权所有 纯洁的微笑</p
>
</footer>
```

页尾比较简单，只是展示出版权信息。

接下来需要做一个页面模板 layout.html，包含标题、内容和页尾。

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"  th:fragment="common_layout(title
,content)">
<head>
    <meta charset="UTF-8"></meta>
    <title th:replace="${title}">comm title</title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.css}"></link>
    <link rel="stylesheet" th:href="@{/css/main.css}"></link>
</head>
<body>
    <div class="container">
        <th:block th:replace="${content}" />
        <th:block th:insert="layout/footer :: footer" ></th:block>
    </div>
</body>
</html>
```

这里定义了一个片段表达式 common_layout(title,content)，同时在页面可以看到 `<title th:replace="${title}">comm title</title>` 和 `<th:block th:replace="${content}" />` 的两块作为片段表达式的参数，也就是说如果其他页面想使用此页面的布局，只需要传入 title 和 content 两块的页面代码即可。

页面中使用了 th:block，此元素作为页面的自定义使用不会展示到页面中，在页面模板的 head 中引入了两个 css 文件，也意味使用此片段表达式的页面同时会具有这两个 css 文件，在页面的最后将我们抽取的页面做完页面片段引入。此模板页面并没有引入 Header 页面信息，因此我们只希望在列表页面展示用户的登录状态信息。

用户列表页引入模板 layout 示例：

```
<html xmlns:th="http://www.thymeleaf.org"  th:replace="layout :: common_layout(~{::ti
tle},~{::content})">
<head>
    <meta charset="UTF-8"/>
    <title>用户列表</title>
</head>
<body>
    <content>
        <th:block th:if="${users!=null}" th:replace="layout/header :: header"  ></th:
block>
        ...
        用户列表信息
        ...
    </content>
</body>
</html>
```

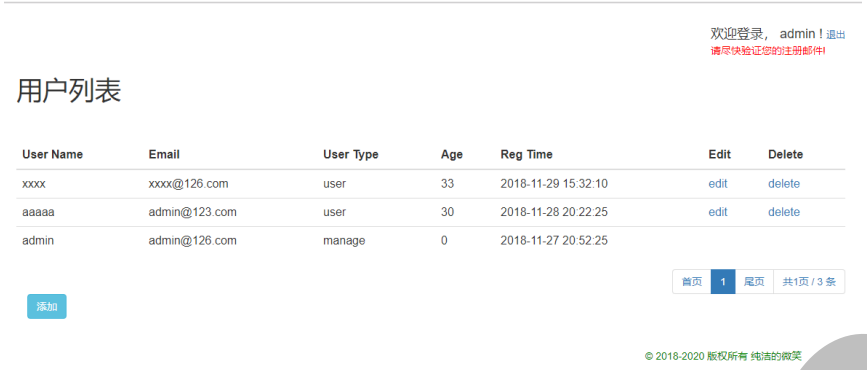
最主要有三块内容需要修改：

- html 头部添加 th:replace="layout :: common_layout(~{::title},~{::content})" 说明只有了 layout.html 页面的

common_layout 片段表达式；

- `<th:block th:if="${users!=null}" th:replace="layout/header :: header"></th:block>` 页面引入了前面定义的 Header 信息，也就是用户登录状态相关内容；
- 提前定义好 title 和 content 标签，这两个页面标签会作为参数和定义的页面模板组合成新的页面。

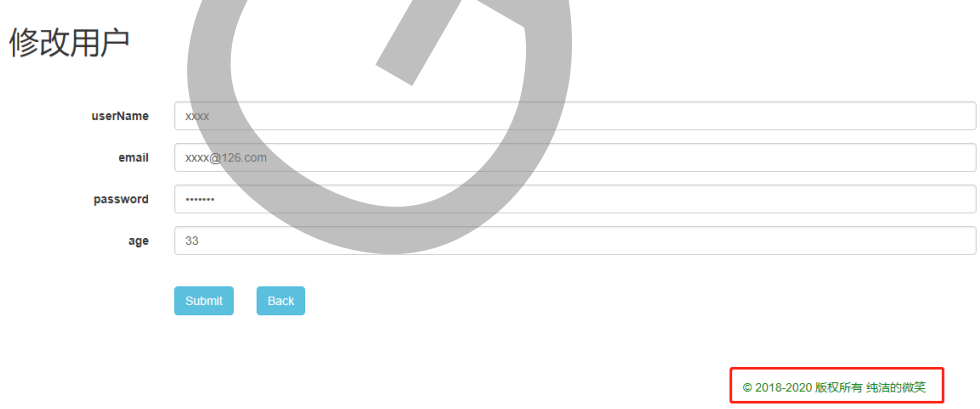
效果图如下：



修改用户页面模板示例：

```
<html xmlns:th="http://www.thymeleaf.org"th:replace="layout :: common_layout(~{::titl
e},~{::content})" >
<head>
    <meta charset="UTF-8"/>
    <title>修改用户</title>
</head>
<body>
    <content >
     ....
     修改页面
     ....
    </div>
</body>
</html>
```

效果图如下：



我们发现修改页面有版权信息，证明使用片段表达式布局成功，添加用户页面类似这里不再展示。

统一异常处理

如果在项目运行中出现了异常，我们一般不希望将这个信息打印到前端，可能会涉及到安全问题，并且对用户不够友好，业内常用的做法是返回一个统一的错误页面提示错误信息，利用 Spring Boot 相关特性很容易实现此功能。

首先来自定义一个错误页面 error.html：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head lang="en" >
    <meta charset="UTF-8" />
    <title>500</title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.css}"></link>

</head>
<body class="container">
    <h1>服务端错误</h1>
    请求地址：<pan th:text="${url}"></pan><br/>
    错误信息：<pan th:text="${exception}"></pan>
</body>
</html>
```

页面有两个变量信息，一个是出现错误的请求地址和异常信息的展示。

创建 GlobalExceptionHandler 类处理全局异常情况。

```
@ControllerAdvice
public class GlobalExceptionHandler {
    protected Logger logger =  LoggerFactory.getLogger(this.getClass());
    public static final String DEFAULT_ERROR_VIEW = "error";

    @ExceptionHandler(value = Exception.class)
    public ModelAndView defaultErrorHandler(Exception e, HttpServletRequest request)
throws Exception {
        logger.info("request url: " + request.getRequestURL());
        ModelAndView mav = new ModelAndView();
        mav.addObject("exception", e);
        mav.addObject("url", request.getRequestURL());
        logger.error("exception: ",e);
        mav.setViewName(DEFAULT_ERROR_VIEW);
        return mav;
    }
}
```

@ControllerAdvice 是一个控制器增强的工具类，可以在项目处理请求的时候去做一些额外的操作，

@ControllerAdvice 注解内部使用 @ExceptionHandler、@InitBinder、@ModelAttribute 注解的方法应用到所有的 @RequestMapping 注解方法。@ExceptionHandler 注解即可监控 Contoller 层代码的相关异常信息。

我们修改代码在登录页面控制器中抛出异常来测试：

```
@RequestMapping("/toLogin")
public String toLogin() {
    if (true)
    throw  new RuntimeException("test");
    return "login";
}
```

启动项目之后，访问地址 http://localhost:8080/，页面即可展示以下信息：

```
服务端错误
请求地址：http://localhost:8080/toLogin
错误信息：java.lang.RuntimeException: test
```

可以看出打印出来出现异常的请求地址和异常信息，表明统一异常处理成功拦截了异常信息。

Docker 部署

我们将用户管理系统 user-manage 复制一份重新命名为 user-manage-plus，在 user-manage-plus 项目上添加 Docer 部署。

（1）项目添加 Docker 插件

在 pom.xml 文件中添加 Docker 镜像名称前缀：

```
<properties>
    <docker.image.prefix>springboot</docker.image.prefix>
</properties>
```

plugins 中添加 Docker 构建插件：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <!-- Docker maven plugin -->
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <version>1.0.0</version>
            <configuration>
                <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
                <dockerDirectory>src/main/docker</dockerDirectory>
                <resources>
                    <resource>
                        <targetPath>/</targetPath>
                        <directory>${project.build.directory}</directory>
                        <include>${project.build.finalName}.jar</include>
                    </resource>
                </resources>
            </configuration>
        </plugin>
        <!-- Docker maven plugin -->
    </plugins>
</build>
```

（2）添加 Dockerfile 文件

在目录 src/main/docker 下创建 Dockerfile 文件：

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD user-manage-plus-1.0.jar app.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

> 注意 ADD 的是我们打包好的项目 Jar 包名称。

（3）部署

将项目 user-manage-plus 复制到安装好 Docker 环境的服务器中，进入项目路径下。

```
#打包
mvn clean package
#启动
java -jar target/user-manage-plus-1.0.jar
```

看到 Spring Boot 的启动日志后表明环境配置没有问题，接下来使用 DockerFile 构建镜像。

```
mvn package docker:build
```

构建成功后，使用 docker images 命令查看构建好的镜像：

```
[root@localhost user-manage-plus]# docker images
REPOSITORY                                  TAG                 IMAGE ID
 CREATED              SIZE
springboot/user-manage-plus                 latest              f5e23ce0ce7d
 4 seconds ago       139 MB
```

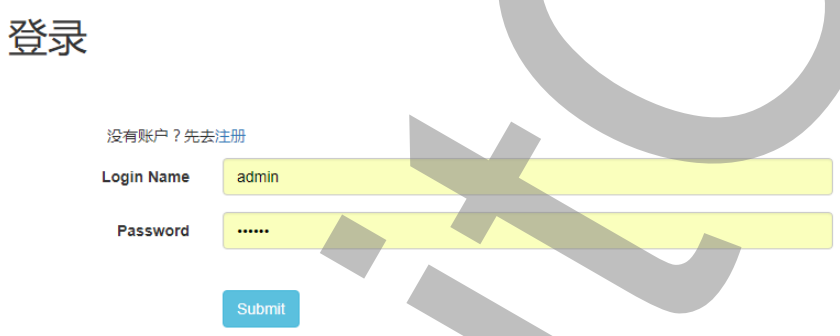
springboot/user-manage-plus 就是我们构建好的镜像，下一步就是运行该镜像：

```
docker run -p 8080:8080 -t springboot/user-manage-plus
```

启动完成之后我们使用 docker ps 查看正在运行的镜像：

```
[root@localhost user-manage-plus]# docker ps
CONTAINER ID        IMAGE                              COMMAND                  CREATED
          STATUS              PORTS                      NAMES
6e0ba131da6d        springboot/user-manage-plus    "java -Djava.secur..."   2 minutes
ago       Up 2 minutes        0.0.0.0:8080->8080/tcp   elastic_bartik
```

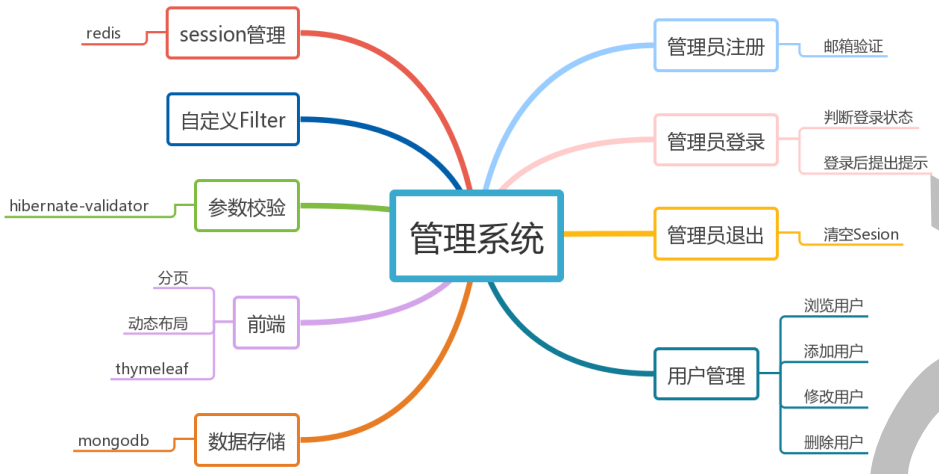
可以看到构建的容器正在在运行，访问浏览器 http://192.168.0.x:8080，跳转到登录页面证明项目启动成功。



说明使用 Docker 部署 user-manage-plus 项目成功！

总结

我们用思维导图来看一下用户管理系统所涉及到的内容：



左边是我们使用的技术栈，右边为用户管理系统所包含的功能，通过这一节课的综合实践，我们了解到如何使用 Spring Boot 去开发一个完整的项目、如何在项目中使用我们前期课程所学习的内容。

> 点击这里下载源码