Demo 介绍

本项目是搭建一个 `用户管理系统` 。采用 springboot 作为框架开发，并且以 Mysql 作为数据库进行项目的搭建。

分为外部网站和内部网站。当用户未进行登陆时，则可以访问外部网站'index, Login 和 Register'。当用

户登陆成功后，会添加 cookie 并且进入内部网站。可以对数据库执行 `增删改查` 等基础功能。

数据库

数据库源码解释

使用 `jdbctemplate` 进行数据库相关操作。所有在Query.java里的方法都以 `user 对象` 的形式返回。

```
import org.springframework.jdbc.core.JdbcTemplate;

//以多个对象的形式 返回所有消息
public List<UserInfo> AllInfo() {
    String sql = "SELECT * FROM User_Info";
    return jdbcTemplate.query(sql, new UserRowMapper());
}

//将查询到的信息，打包成对象。
public class UserRowMapper implements RowMapper<UserInfo> {
    public UserInfo mapRow(ResultSet rs, int rowNum) throws
SQLException {
        UserInfo user = new UserInfo();
        user.setId(rs.getInt("Id"));
        user.setPassword(rs.getString("Password"));
```

```
            user.setUsername(rs.getString("Name"));
            user.setRole(rs.getString("Role"));
            return user;
        }
    }
```

当需要调用 Query.java 里的方法时，只需要进行

```
@Autowired
private Query query;

//在某个方法里执行
List<UserInfo> userInfos = query.AllInfo()
```

Nvicat 基础配置

application.properties 基础配置为

```
spring.datasource.url=jdbc:mysql://localhost:3306/scca_user_tes
spring.datasource.username=root
```

表的初始化 与 `UserInfo.java 里的对象` 对应

```
CREATE TABLE User_Info (
    Id INT PRIMARY KEY AUTO_INCREMENT,
    Name TEXT,
    Password TEXT,
    Role TEXT
);
```

用户登陆与注册

登陆认证

1: 后端通过 `@RequestMapping("/Login_Check")` 从前端表单里获取对应参数

```
<form method="post" th:action="@{/Login_Check}">
```

2: 参数传递与获取

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

public String Acquire(@RequestParam("username") String username,
                      @RequestParam ("password") String password,
                      HttpServletResponse response,
                      Model md)
```

3: 首先从数据库里查询所有用户信息

```
List<UserInfo> UserInfos = query.AllInfo()
```

4: 然后再将获取到的 Parameters 与数据库中的数据进行对比， 并进行判断

```
Boolean result = false; //默认结果为 false
Integer index = 0;      //为 cookie 的创建做准备
for (int i = 0; i < UserInfos.size(); i++) {        //进行判断循环
  if(UserInfos.get(i).getUsername().equals(username) &&
UserInfos.get(i).getPassword().equals(password)){       //开始进行对比，是
否存在匹配结果
    result = true;          //(如果)匹配成功，设置 result 为正确，index 为当
前 索引。
    index = i;
    break;
  }
}
```

5: 最终根据 result 跳转不同页面

用户注册

1: 用户注册的页面首先会被下面代码渲染

```
@PostMapping("/Register")
    public String Register(){
        return "Register";
    }
```

2: 由于html 是由thymeleaf 进行渲染。所以第一次页面 ${result}不显示，后续根据后端判断返回相关错误描述。

```
<h5 th:text="${result}">禁止任何特殊符号输入 <br> 长度小于 10!</h5>
```

3: 与 登陆认证 原理一样，前后端执行数据传递后，进行判断与近一步操作。在后端判断中，加入了 `长度以及特殊字符` 的判断, 来防止对数据库的潜在威胁。当用户触发相关错误时，错误信息会被传递到 ${result}中，并显示.

Cookie 的创建

代码部分释义

当用户进行登陆成功后，后端会先在数据库总寻找用户相对应的Id(Unique). 更具ID进行 `Cookie的添加` 。

并且内部网站的地址都位于 `/Dashboard` 之后，所以将Cookie的 path 设置为 /Dashboard.

```
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletResponse;

if (result) {              //当用户信息确认后
            try {
                Integer targetId = UserInfos.get(index).getId();
                String ConId = Integer.toString(targetId);
                Cookie cookie = new Cookie(ConId,
UserInfos.get(index).getUsername());
                cookie.setPath("/Dashboard");
                cookie.setMaxAge(180);
                response.addCookie(cookie);
            }catch (Exception e){
```

```
                System.out.printf(String.valueOf(e));
            }

            System.out.printf("Cookie 已经添加! \n");

            return "/Dashboard/Homepage";
```

判断当前用户

CheckCurrentUser 是一个以封装的方法。

1: `查看` 目前 `所有` 存在的cookie.

2: 进入循环，获取每一个 cookie 的 `Name & Value` 。在这里 Name 是用户的 ID

3: 从数据库获取所有用户信息，并且寻找 用户ID 对应的 Username.

4: 如果数据库查到的 ID 和 username 与 Cookie 获取的 ID 与 Username 匹配，则确认该用户身份信息。

```
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

//方法调用
List<String> CookieFeedback = new ArrayList<>();
CookieFeedback = CheckCurrentUser(request);
System.out.printf("现在登陆的用户 \n用户ID: "+CookieFeedback.get(0)+", 用户
名: "+CookieFeedback.get(1)+"\n");

//封装方法
private List<String> CheckCurrentUser(HttpServletRequest request){
  //---------------------------- 1 ----------------------------
        Cookie[] cookies = request.getCookies();
        List<String> FinalResult = new ArrayList<>();
        if(cookies != null) {
            for (int i = 0; i < cookies.length; i++) {
              //---------------------------- 2 ----------------------
------
                String cookieID = cookies[i].getName();
                String cookieValue = cookies[i].getValue();
                // 获取所有用户 关注ID
```

```
                List<UserInfo> userInfos = query.AllInfo();
                for (int j = 0; j < userInfos.size(); j++) {
                    //先 转换 ID
                    try {
                      //----------------------------- 3 ---------------
--------------
                        int Id=Integer.parseInt(cookieID);
                        if (userInfos.get(j).getId().equals(Id) &&
userInfos.get(j).getUsername().equals(cookieValue)){
                            FinalResult.add(cookieID);
                            FinalResult.add(cookieValue);
                          //----------------------------- 4 -----------
-----------------
                            // 返回 ID 和 name
                            return FinalResult;
                        }
                    }catch (Exception e){
                        continue;
                    }
                }
            }
        }
        return FinalResult;
    }
```

删除用户 Cookie

当用户点击 Logout 页面的按钮时，将会触发该方法。

1: 确认当前操作用户（详细原理点击这里）

2: 更具确认信息中的 user ID 来进行cookie操作

3: 将该cookie的 value 设置为 null

4: 将有效期设置为0

5: 更新该cookie

```
@RequestMapping("/Dashboard/LogoutButton")
    public String Logout(HttpServletResponse
response,HttpServletRequest request) {
```

```
        List<String> CookieFeedback = new ArrayList<>();
        CookieFeedback = CheckCurrentUser(request);
        System.out.printf("现在登陆的用户 \n用户ID:
"+CookieFeedback.get(0)+", 用户名: "+CookieFeedback.get(1)+"\n");

        Cookie cookie = new Cookie(CookieFeedback.get(0), null);
        //将`Max-Age`设置为0
        cookie.setMaxAge(0);
        response.addCookie(cookie);

        System.out.printf("已删除该用户 cookie\n");

        return "redirect:/Login";
    }
```

增删改查

用户列表

用户列表的底层逻辑，则是 `调用` 数据库中`所有的用户信息。通过model.addAttribute 传递至前端网页。

Java/ html 代码：

```
List<UserInfo> userInfos = query.AllInfo();
model.addAttribute("usersList", userInfos);    //对应表格里的
${usersList}
return "/Dashboard/UserList";
```

```
<table class="table table-striped">
            <thead>
            <tr>
                <th>用户 ID</th>
                <th>用户名</th>
                <th>用户密码</th>
                <th>用户身份类型</th>
            </tr>
            </thead>
```

```
            <tbody>
            <tr th:each="user : ${usersList}">
                <td th:text="${user.id}">用户 ID</td>
                <td th:text="${user.username}">用户名</td>
                <td th:text="${user.password}">用户密码</td>
                <td th:text="${user.role}">用户身份类型</td>
            </tr>
            </tbody>
        </table>
```

页面展示：

添加用户

逻辑和用户注册是一样的，唯一区别则是：在该界面创建用户时，可以创建Admin 权限的用户。

Java 代码：

```
@RequestMapping ("/Dashboard/AddUser")
public String AcquireInfom(@RequestParam("username") String username,
                           @RequestParam ("password") String password,
                           @RequestParam ("role") String role,
                           Model md) {
```

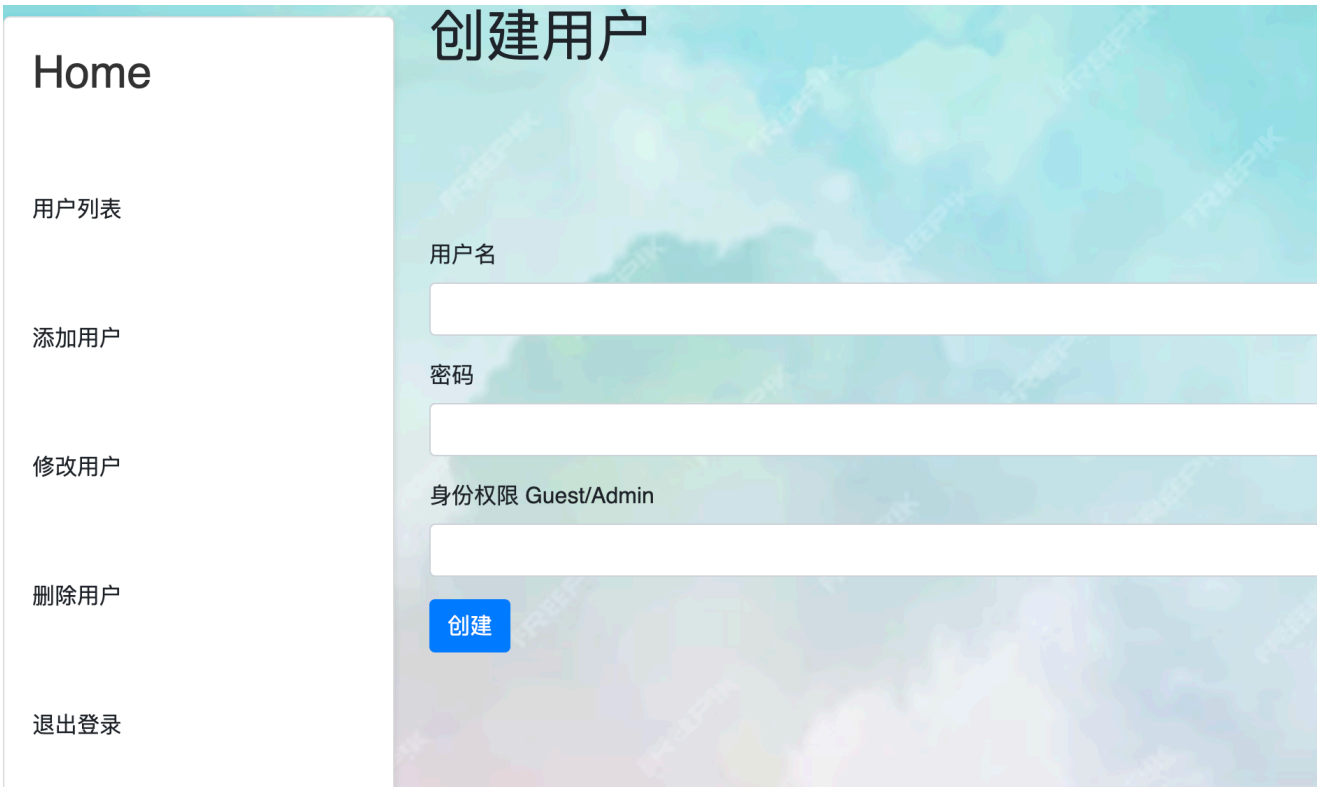
```
    // 检查是否有特殊符号 和 长度
    if (check_input(username) || check_input(password)) {
      md.addAttribute("Alert", "您的输入有特殊符号! ");
      return "/Dashboard/AddUser";
    } else if (username.length() > 10 || password.length() > 10) {
      md.addAttribute("Alert", "您输入的密码或账户名过长! ");
      return "/Dashboard/AddUser";
    } else if (!(role.equals("Admin") || role.equals("Guest"))) {
      md.addAttribute("Alert", "只支持 Admin 以及 Guest! ");
      return "/Dashboard/AddUser";
    }

    //数据库开始插入
    UserInfo newUser = new UserInfo(0, username, password, role);
    Boolean RegisterResult = query.registerUser(newUser);

    //跳转网页
    if (RegisterResult) {
      return "redirect:/Dashboard/UserList";
    } else {
      md.addAttribute("result", "Error, Something wrong here.");
      return "/Dashboard/AddUser";
    }
  }
```

html:

```
<form method="post" th:action="@{/AddUser}">
    <div class="form-group">
      <label for="username">用户名</label>
      <input name="username" type="text" class="form-control"
id="username" placeholder="">
    </div>
    <div class="form-group">
      <label for="password">密码</label>
      <input name="password" type="password" class="form-control"
id="password" placeholder="">
    </div>
    <div class="form-group">
      <label for="role">身份权限 Guest/Admin</label>
      <input name="role" type="text" class="form-control" id="role"
placeholder="">
    </div>
```

```
  <button type="submit" class="btn btn-primary">创建</button>
</form>
```

页面展示：



修改用户

用户可以进行 `Id, username, password, role` 的修改。即使只修改其中一栏。

Java:

```
@RequestMapping ("/Dashboard/updateUser")
public String updateUser(@RequestParam("MyId") String myIdStr,
                         @RequestParam("username") String username,
                         @RequestParam("password") String password,
                         @RequestParam("role") String role) {
  int Id = 0;
  try {
    Id = Integer.parseInt(myIdStr); // 将字符串转换为整数

  } catch (NumberFormatException e) {
    System.out.println("转换错误: " + e.getMessage());
    // 处理转换异常，可能需要返回一个错误页面或消息
```

```
    }
    // 更新用户信息的逻辑
    query.updateUserById(Id,username, password, role);
    return "redirect:/Dashboard/UserList";
}
```

Html:

```
<form id="edit-form" th:action="@{/updateUser}" method="post">
  <div class="form-group">
    <label for="edit-id">用户ID:</label>
    <input type="text" class="form-control" id="edit-id" name="MyId"/>
  </div>
  <div class="form-group">
    <label for="edit-username">用户名:</label>
    <input type="text" class="form-control" id="edit-username"
name="username"/>
  </div>
  <div class="form-group">
    <label for="edit-password">密码:</label>
    <input type="password" class="form-control" id="edit-password"
name="password"/>
  </div>
  <div class="form-group">
    <label for="edit-role">权限:</label>
    <input type="text" class="form-control" id="edit-role"
name="role"/>
  </div>
  <button type="submit" class="btn btn-primary">提交</button>
</form>
```

页面展示：

删除用户

Java:

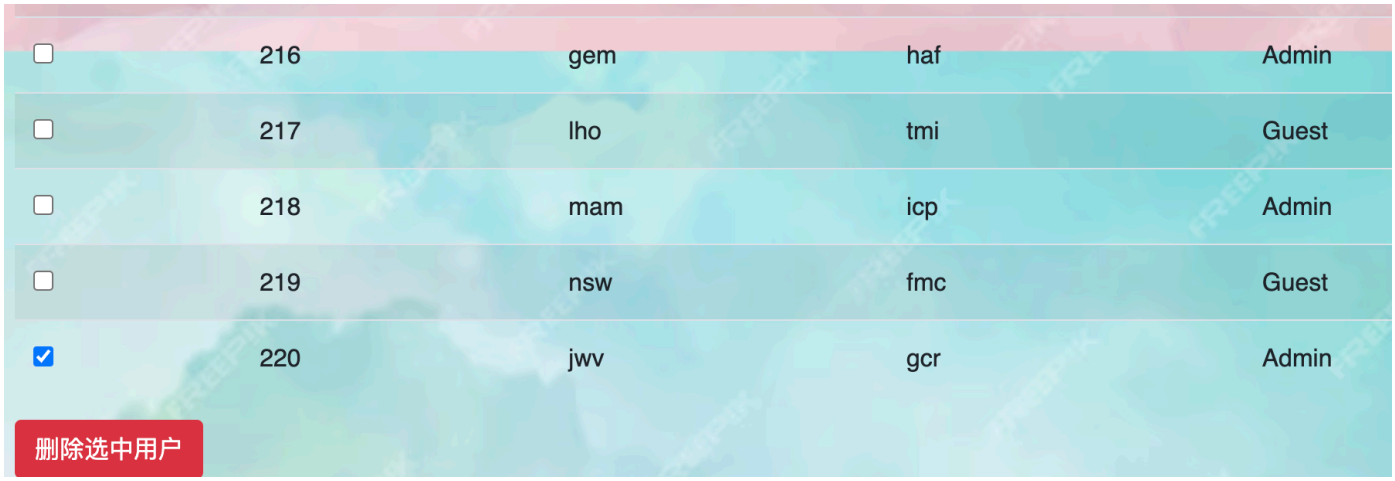
```
    @GetMapping("/Dashboard/DeleteUser") //获取网页所以用 Get
    public String showUsersDelete(Model model) {
        List<UserInfo> userInfos = query.AllInfo();
        model.addAttribute("usersList_delete", userInfos);  //先显示素有用户信息
        return "/Dashboard/DeleteUser";
    }

    @PostMapping("/Dashboard/DeleteUser")   //对应 html Post
    public String deleteSelectedUsers() {return "redirect:/Dashboard/UserList";}
```

Html:

```
<form th:action="@{/DeleteUser}" method="post">
        <table class="table table-striped">
            <thead>
            <tr>
```

```
                <th>选择</th>
                <th>用户 ID</th>
                <th>用户名</th>
                <th>用户密码</th>
                <th>用户身份类型</th>
            </tr>
            </thead>
            <tbody>
            <tr th:each="user : ${usersList_delete}">
                <td><input type="checkbox" th:name="'user_' +
${user.id}" /></td>
                <td th:text="${user.id}">用户 ID</td>
                <td th:text="${user.username}">用户名</td>
                <td th:text="${user.password}">用户密码</td>
                <td th:text="${user.role}">用户身份类型</td>
            </tr>
            </tbo
```

页面展示：



Spring Boot 注解

@Controller 类

负责处理 `通过HTTP请求(request)` 到达的 `用户请求` ，并返回相应的视图，确保 `数据模型和视图之间的交互`

@Autowired

是Spring的 `自动装配注解` 。它可以用于字段、构造函数、方法等，用于自动注入依赖

当Spring创建一个包含 `@Autowired` 注解的类的实例时，它会尝试通过类型匹配方式来注入所需的依赖

@GetMapping

用于 `将HTTP GET request` map 到特定的处理方法上。它是一个组合注解，是 `@RequestMapping(method = RequestMethod.GET)` 的简写。通常用于读取数据的操作。

@PostMapping

类似于 `@GetMapping` ，它将HTTP POST请求映射到特定的处理方法上。通常 `用于提交表单数据` 或 `上传文件`

@RequestMapping

用于 `映射HTTP请求到 Controller Function` 。它可以注解到类或方法上。在类上使用时，表示类中所有响应方法的公共路径前缀；在方法上使用时，指定与该方法相对应的具体请求路径

@Service

这是一个用于 `标注业务层组件` 的注解，在Spring框架中，它用于标注一个服务层类，表明这个类 `提供了业务功能`

@Entity

用于标注实体类，表示它是一个JPA实体。这个注解将 class map 到数据库表

@Id

这是JPA的注解，用于标识实体类中的 Primary Key 字段

@GeneratedValue

(strategy = GenerationType.IDENTITY)

这个注解与 `@Id` 一起使用，用于指定主键的生成策略。在这里，`GenerationType.IDENTITY` 指的是主键由数据库自动生成（通常是自增）。

Thymeleaf

本 demo 中各个例子解释

1. `<td th:text="${user.id}">用户 ID</td>`
    - 这个例子中， `th:text` 属性用于动态显示变量 `user.id` 的值。在Thymeleaf中，`${...}` 用于表达式求值。如果 `user.id` 的值存在，它会替换掉 `<td>` 标签内的默认文本“用户 ID”。

2. `<tr th:each="user : ${usersList_delete}">`
   - 这里的 `th:each` 属性用于遍历 `usersList_delete` 集合。对于集合中的每个元素，都会生成一个 `<tr>` 元素。在这个循环中，每个元素都被称为 `user`，可以在循环体内部使用。
3. `<form th:action="@{/DeleteUser}" method="post">`
   - 在这个例子中，`th:action` 定义了表单提交时的URL。`@{...}` 是Thymeleaf中的URL表达式，这里它指向了 `/DeleteUser` 路径。这意味着当表单提交时，请求会被发送到 `/DeleteUser`。
4. `<h5 th:text="${result}">禁止任何特殊符号输入 <br> 长度小于 10!</h5>`
   - 类似于第一个例子，这里的 `th:text` 用于动态显示 `result` 变量的值。如果 `result` 变量存在且有值，它将替换掉 `<h5>` 标签内的默认文本。
5. `<a th:href="@{/user/{id}(id=${user.id})}">View User</a>`
   - 这个例子展示了如何在Thymeleaf中创建动态URL。`th:href` 属性用于设置超链接的 `href` 属性。`@{...}` 里的 `{id}` 是URL模板变量，它会被 `id=${user.id}` 中的值替换。
6. `<span th:if="${not #lists.isEmpty(usersList)}">用户列表不为空</span>`
   - 这里使用了 `th:if` 条件属性。该属性检查表达式的结果是否为 `true`，如果为 `true`，则渲染 `<span>` 标签。在这个例子中，它检查 `usersList` 是否不为空。

所使用的依赖

源码

```
<dependencies>

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

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

```
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>

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

    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>

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

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

</dependencies>
```

用途解释

1. **spring-boot-starter-data-jpa**
    - 这个依赖提供了Spring Data JPA的支持，用于简化数据库操作。它整合了Spring Data和JPA（Java Persistence API），让你可以更容易地通过Java对象来访问和操作关系型数据库。
2. **spring-boot-starter-thymeleaf**

- 该依赖是Thymeleaf的Spring Boot启动器。Thymeleaf是一个现代的服务器端Java模板引擎，用于Web和独立环境。这个依赖使得Thymeleaf与Spring Boot应用程序轻松集成，用于生成动态HTML视图。

3. **spring-boot-starter-web**
   - 这是构建Web应用程序（包括RESTful应用程序）的基础依赖。它包括Spring MVC和Tomcat作为默认的嵌入式容器，使得构建web应用更加方便。
4. **mysql-connector-j**
   - 这个依赖是MySQL数据库的JDBC驱动，用于连接MySQL数据库。`<scope>runtime</scope>` 表示这个依赖只在运行时被使用，不会在编译时被包含。
5. **javax.servlet-api** (这里我们用于处理 Cookie)
   - 这个依赖包含了Java Servlet API，用于编写Servlets。Servlet是运行在服务器端的Java程序，用于扩展服务器功能，处理请求和生成响应。这里的`<version>3.1.0</version>` 指定了使用的API版本。
6. **spring-boot-starter-test**
   - 这个依赖提供了测试Spring Boot应用所需的库，包括JUnit、Spring Test和Mockito等。`<scope>test</scope>` 表示这个依赖仅在测试编译和执行阶段被使用。
7. **spring-boot-starter-jdbc**
   - 该依赖提供了对JDBC（Java数据库连接）的支持。它使得在Spring Boot应用中连接和操作数据库变得简单，包括了如连接池等常用的数据库操作功能