第5章

Spring Boot安全管理

本章学习目标

- 理解 Spring Security 的基本概念。
- 掌握安全管理效果测试方法。
- 熟悉自定义用户认证过程。
- 熟悉自定义用户授权管理过程。

根据党的二十大报告，网络安全作为网络强国、数字中国的底座，将在未来的发展中承负重担，是我国现代化产业体系中不可或缺的部分。

在实际开发中，一些应用通常要考虑到安全性问题。例如，对于一些重要的操作，有些请求需要用户验明身份后才可以执行，还有一些请求需要用户具有特定权限才可以执行。这样做不仅可以用来保护项目安全，而且可以控制项目访问效果。

本章将针对 Spring Boot 安全管理进行讲解，并进而体会网络安全的重要性。

5.1 Spring Security 简介

5.1.1 什么是 Spring Security

安全可以说是公司的红线，一般项目都有严格的认证和授权操作，在 Java 开发领域常见的安全框架有 Shiro 和 Spring Security。Shiro 是一个轻量级的安全管理框架，提供了认证、授权、会话管理、密码管理和缓存管理等功能。Spring Security 是一个相对复杂的安全管理框架，功能比 Shiro 更加强大，权限控制细粒度更高，对 OAuth2 的支持也更好。由于 Spring Security 源自 Spring 家族，因此可以和 Spring 框架无缝整合，特别是 Spring Boot 中提供的自动化配置方案，可以让 Spring Security 的使用更加便捷。

Spring Security 是 Spring 社区的一个顶级项目，也是 Spring Boot 官方推荐使用的安全框架。除了常规的认证(authentication)和授权(authorization)外，Spring Security 还提供了诸如 ACLs、LDAP、JAAS 和 CAS 等高级特性，以满足复杂场景下的安全需求。

Spring Security 应用级别的安全主要包含两个主要部分，即登录认证(login authentication)和访问授权(access authorization)。首先在用户登录时传入登录信息，登录验证器完成登录认证并将登录认证好的信息存储到请求上下文；然后进行其他操作，如接口访问、方法调用等，权限认证器从上下文中获取登录认证信息；最后根据认证信息获取权限信息，通过权限

信息和特定的授权策略决定是否授权。

简单来说，Spring Security 核心功能只做以下几件事情：

(1) 在系统初始化时，告诉 Spring Security 访问路径所需要的对应权限。

(2) 在登录时，告诉 Spring Security 真实用户名和密码。

(3) 在登录成功时，告诉 Spring Security 当前用户具备的权限。

(4) 在用户访问接口时，Spring Security 已经知道用户具备的权限，也知道访问路径需要的对应权限，可以自动判断该用户能否访问。

5.1.2 为什么要使用 Spring Security

在项目开发中，安全框架多种多样，那么为什么要选择 Spring Security 作为微服务开发的安全框架呢？Java EE 有另一个优秀的安全框架 Apache Shiro，Apache Shiro 框架在企业级的项目开发中非常受欢迎，一般在单体项目中使用；在微服务架构中目前却是无能为力的。

选择 Spring Security 的原因之一是它来自于 Spring Resource 社区，采用了注解的方式来控制权限，熟悉 Spring 框架者很容易上手。另外一个原因是 Spring Security 很容易应用在 Spring Boot 工程中，也易于集成到 Spring Cloud 构建的微服务项目中。

Spring Security 是一个能够为基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。它提供了一组可以在 Spring 应用上下文中配置的 Bean，充分利用了 Spring IoC(Inversion of Control，控制反转)、DI(Dependency Injection，依赖注入)和 AOP(Aspect Oriented Programming，面向切面编程)功能，为应用系统提供声明式的安全访问控制功能，减少了为企业系统安全控制编写大量重复代码的工作。

5.1.3 Spring Security 的核心类

1. SecurityContext

SecurityContext 中包含当前正在访问系统的用户的详细信息，它有以下两种方法。

- getAuthentication()：获取当前经过身份验证的主题或请求令牌。
- setAuthentication()：更改或删除当前已经验证的主体身份验证信息。

SecurityContext 的信息是由 SecurityContextHolder 来处理的。

2. SecurityContextHolder

SecurityContextHolder 是最基本的对象，保存当前会话用户认证、权限和鉴权等核心数据。SecurityContextHolder 默认使用 ThreadLocal 策略来存储认证信息、与线程绑定的策略等。在用户退出时，它会自动清除当前线程的认证信息。

最常用的方法是 getContext()方法，用来获得当前的 SecurityContext。该方法使用 Authentication 对象来描述当前用户的相关信息。SecurityContextHolder 持有的是当前用户的 SecurityContext，而 SecurityContext 持有的是代表当前用户相关信息的 Authentication 的引用。这个 Authentication 对象不需要自己创建，Spring Security 会自动创建，然后赋值给当前的 SecurityContext。

在程序的任何地方，可以通过如下方式获取到当前用户的用户名。

```
public String getCurrentUsername(){
    Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

```
        if(principal instanceof UserDetails){
            return ((UserDetails) principal).getUsername();
        }
        if(principal instanceof Principal){
            return ((Principal) principal).getName();
        }
        return String.valueOf(principal);
    }
```

getAuthentication()方法会返回认证信息。

getPrincipal()方法会返回身份信息,它是 UserDetails 对身份信息的封装。

获取当前用户的用户名,最简单的方式如下:

```
public String getCurrentUsername(){
    return SecurityContextHolder.getContext().getAuthentication().getName();
}
```

3. PrividerManager

PrividerManager 会维护一个认证列表,用来处理不同认证方式的认证,这是因为系统可能会存在多种认证方式,如手机号、用户名密码、邮箱等。如果认证结果不是 null,则说明成功,存在 SecurityContext 中;如果认证结果是 null,则说明不成功,抛出 ProviderNotFoundException 异常。

4. DaoAuthenticationProvider

DaoAuthenticationProvider 是 AuthenticationProvider 最常用的实现,用来获取用户提交的用户名和密码,并进行正确性比对。如果比对正确,则返回数据库中的用户信息。

5. UserDetails

UserDetails 是 Spring Security 的用户实体类,包含用户名、密码、权限等信息。Spring Security 默认实现了内置的 User 类,供 Spring Security 安全认证使用,也可以自己实现。

UserDetails 接口和 Authentication 接口很类似,都拥有 username 和 authorities。一定要区分清楚 Authentication 中的 getCredentials()与 UserDetails 中的 getPassword(),前者是用户提交的密码凭证,不一定正确或数据库不一定存在;后者是用户正确的密码。认证器要进行比对的就是两者是否相同。

UserDetails 的实现代码如下:

```
public interface UserDetails extends Serializable {
  Collection<? extends GrantedAuthority> getAuthorities();
  String getPassword();
  String getUsername();
  boolean isAccountNonExpired();
  boolean isAccountNonLocked();
  boolean isCredentialsNonExpired();
  boolean isEnabled();
}
```

6. UserDetailsService

用户信息通过 UserDetailsService 接口加载。该接口的唯一方法是 loadUserByUsername(String username),用来根据用户名加载相关信息。这个方法返回值是 UserDetails 接口,其中包含了用户的信息,包括用户名、密码、权限、是否启用等。

UserDetailsService 的实现代码如下：

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String var1) throws UsernameNotFoundException;
}
```

7. Authentication

Authentication 是建立系统使用者信息(principal)的过程。用户认证一般要求用户提供用户名和密码，系统通过用户名密码完成认证通过或拒绝。Spring Security 支持主流认证方式，包括 HTTP 基本认证、表单验证、摘要认证、OpenID 和 LDAP 等。除了利用提供的认证外，还可以编写自己的 Filter(过滤器)，以保证并非基于 Spring Security 的验证系统的操作。

用户认证的验证步骤如下：

(1) 用户使用用户名和密码登录。

(2) 过滤器获取到用户名和密码，然后封装成 Authentication。

(3) AuthenticationManager 认证 token(Authentication 的实现类传递)。

(4) AuthenticationManager 认证成功，返回一个封装了用户权限信息的 Authentication 对象，建立用户的上下文信息(如角色列表等)。

(5) Authentication 对象赋值给当前的 SecurityContext，建立这个用户的安全上下文(通过调用 SecurityContextHolder.getContext().setAuthentication()实现)。

(6) 用户进行一些受到访问控制机制保护的操作，访问控制机制会依据当前安全上下文信息检查这个操作所需要的权限。

Authentication 的实现代码如下：

```
public interface Authentication extends Principal, Serializable {
  //权限列表,通常是代表权限的字符串集合
  Collection<? extends GrantedAuthority> getAuthorities();
  //密码,认证之后会移出,用来保证安全性
  Object getCredentials();
  //请求的细节参数
  Object getDetails();
  //核心身份信息,一般返回 UserDetails 的实现类
  Object getPrincipal();
  boolean isAuthenticated();
  void setAuthenticated(boolean var1) throws IllegalArgumentException;
}
```

8. Authorization

在一个系统中，不同用户所具有的权限是不同的。系统会为不同的用户分配不同的角色，而每个角色则对应一系列的权限。

对 Web 资源的保护，最好的办法是使用过滤器。对方法调用的保护，最好的方法是使用 AOP。

5.2 安全管理效果测试


视频讲解

Spring Boot 针对 Spring Security 提供了自动化配置方案，因此可以使 Spring Security 非常容易地整合进 Spring Boot 项目中，这也是在 Spring Boot 项目中使用 Spring Security

的优势。

下面通过示例讲解 Spring Boot 实现安全管理的基本方法。

1. 添加依赖

以项目 springboot0406 为基础,创建项目 springboot0501,在 Maven 中添加依赖,以便启用安全管理。

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

2. 添加图书首页

添加网上图书的首页 index.html,该页面中通过标签分类展示了书店前台和书店后台两个选项,并且这两个选项都通过<a>标签连接到了具体的页面。

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>网上书店</title>
</head>
<body>
      <h1 align="center">欢迎进入网上书店</h1>
      <div>
          <ul>
              <li><a th:href="@{/loginbooks}">书店前台</a></li>
               <li><a th:href="@{/loginadmin}">书店后台</a></li>
          </ul>
      </div>
</body>
</html>
```

3. 启动项目测试

接下来启动项目,启动成功后,执行“http://localhost:8080/”访问项目首页,会自动跳转到一个新的登录链接页面“http://localhost:8080/login”,这说明在项目中添加 spring-boot-starter-security 依赖启动器后,项目实现了 Spring Security 的自动化配置,并且具有了一些默认的安全管理功能。另外,项目会自动跳转到登录页面,这个登录页面是由 Spring Security 提供的,如图 5-1 所示。

图 5-1 项目首页访问效果

当在 Spring Security 提供的默认登录页面"/login"中输入错误的登录信息后,会重定向到"/login?error"页面并显示出错误信息,如图 5-2 所示。

图 5-2 项目登录错误效果

需要说明的是，在 Spring Boot 项目中加入安全依赖启动器后，Security 会默认提供一个可登录的用户信息。该用户信息的用户名为 user，密码会随着项目的每次启动随机生成并打印在控制台上。查看项目启动日志，得到密码如图 5-3 所示。

```
Using generated security password: ee4a1590-8a18-4332-9008-81bc3586b6d4
```

图 5-3 查看密码

在登录页面输入正确的用户名和密码，项目登录成功效果如图 5-4 所示。

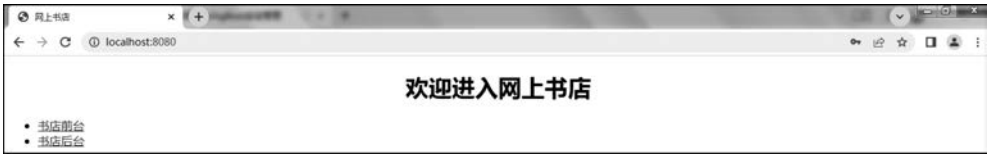

图 5-4 项目登录成功效果

可以发现这种默认安全管理方式存在诸多问题，例如，只有唯一的默认登录用户 user、密码随机生成且过于暴露、登录页面及错误提示页面不是预想的等。

如果开发者对默认的用户名和密码不满意，可以在 application.properties 中配置默认的用户名、密码和用户角色，配置方式如下：

```
spring.security.user.name = zhangsan
spring.security.user.password = 123456
spring.security.user.roles = admin
```

再次启动项目，项目启动日志就不会打印出随机生成的密码了。用户可以直接使用配置好的用户名和密码登录，登录成功后，用户还会具有一个角色——admin。

5.3 自定义用户认证

通过自定义 WebSecurityConfigurerAdapter 类型的 Bean 组件，可以完全关闭 Security 提供的 Web 应用默认安全配置，但是不会关闭 UserDetailsService 用户信息自动配置类。如果要关闭 UserDetailsService 默认用户信息配置，可以自定义 UserDetailsService、AuthenticationProvider 或 AuthenticationManager 类型的 Bean 组件。另外，可以通过自定义 WebSecurityConfigurerAdapter 类型的 Bean 组件覆盖默认访问规则。Spring Boot 提供了很多方便的方法，可用于覆盖请求映射和静态资源的访问规则。

通过重写抽象接口 WebSecurityConfigurerAdapter,再加上注解@Configuration,就可以通过重写 configure 方法配置所需要的安全配置。自定义适配器的代码如下。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  //通常用于设置忽略权限的静态资源
  @Override
  public void configure(WebSecurity web) throws Exception {
   super.configure(web);
  }
  //通过 HTTP 对象的 authorizeRequests()方法定义 URL 访问权限,默认为 formLogin()提供一个简单的登录验证页面
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    super.configure(http);
  }
  //通过 auth 对象的方法添加身份验证
  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
   super.configure(auth);
  }
}
```

自定义适配器的配置方法如下。

- AuthenticationManagerBuilder:认证相关 builder,用来配置全局的认证相关的信息。它包含 AuthenticationProvider 和 UserDetailsService,前者是认证服务提供者,后者是用户详情查询服务。
- HttpSecurity:进行权限控制规则相关配置。
- WebSecurity:进行全局请求忽略规则配置、HttpFirewall 配置、debug 配置和全局 SecurityFilterChain 配置。


视频讲解

5.3.1 内存身份认证

in-Memory Authentication(内存身份认证)是最简单的身份认证方式,主要用于 Security 安全认证体验和测试。在自定义内存身份认证时,只需要在重写的 configure(Authentication ManagerBuilder auth)方法中定义测试用户即可。下面通过 Spring Boot 整合 Spring Security 实现内存身份认证。

(1) 创建内存身份认证。

打开项目 springboot0501,创建 SecurityConfig 类并继承 WebSecurityConfigurerAdapter,在重写的 configure(AuthenticationManagerBuilder auth)方法中使用内存身份认证。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BcryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
         User.UserBuilder builder = User.builder().passwordEncoder(passwordEncoder()::
```

```
encode);
        auth.inMemoryAuthentication().withUser(builder.username("zhangsan").password
                        ("123456").roles("common"));
        auth.inMemoryAuthentication().withUser(builder.username("lisi").password
                        ("654321").authorities("ROLE_vip", "ROLE_common"));
    }
}
```

自定义类继承 WebSecurityConfigurerAdapter 类，重写 configure 方法并在其中增加两个用户，配置用户名、密码和角色。需要注意的是，此处要额外设置密码的加密方式，否则在实际登录时会发现，即便输入了正确的用户名密码，也会提示登录失败。这是因为从 Spring Security 5 开始，自定义用户认证必须设置密码编码器用于保护密码，否则控制台会出现异常错误。Spring Security 提供了多种密码编码器，包括 BcryptPasswordEncoder、Pbkdf2PasswordEncoder 和 ScryptPasswordEncoder 等，密码设置不限于本例中的 BcryptPasswordEncoder 密码编码器。

（2）运行程序，效果如图 5-1 所示。

5.3.2 JDBC 身份认证


视频讲解

JDBC Authentication（JDBC 身份认证）通过 JDBC 连接数据库对已有用户身份进行认证。下面通过 Spring Boot 整合 Spring Security 实现 JDBC 身份认证。

（1）数据准备。

JDBC 身份认证的本质是使用数据库中已有的用户信息在项目中实现用户认证服务，所以需要提前准备好相关数据。这里使用之前创建的名为 book 的数据库，在该数据库中修改之前创建的 3 个表 user、role 和 user_role，并插入几条测试数据，分别如图 5-5～图 5-7 所示。

对象 | user @book (db) - 表 | role @book (db) - 表 | user_role @book (db) - 表

开始事务 | 备注 | 筛选 | 排序 | 导入 | 导出

id	password	username	address	phone	email	idcard	valid
admin	$2a$10$5ooQI8dir8jv0/gCa1Six.GpzAdIPf6pMqdminZ/3ijYzivCyPlfK	张三	天津	13821145201	wuxintouxin@163.com	1	1
123	$2a$10$5ooQI8dir8jv0/gCa1Six.GpzAdIPf6pMqdminZ/3ijYzivCyPlfK	李四	北京	12345678881	ww@123.com	3	1
3454	335	赵四	北京	11123132132	WW@123.COM	2	1

图 5-5 用户表

对象 | role @book (db) - 表

开始事务 | 备注 | 筛选

id	rolename
1	ROLE_common
2	ROLE_vip
3	ROLE_vvipadmin

图 5-6 权限表

对象 | user_role @book (db) - 表

开始事务 | 备注 | 筛选

id	user_id	role_id
1	admin	1
2	admin	2
3	123	2

图 5-7 用户权限表

在使用 JDBC 身份认证创建用户/权限表时，需要注意以下几点。

① 在创建用户表 user 时，用户名 id 必须唯一，因为 Security 在进行用户查询时是通过 id 定位是否存在唯一用户的。

② 在创建用户表 user 时，必须额外定义一个 tinyinit 类型的字段（对应 boolean 类型的

属性,如示例中的 valid),用于校验用户身份是否合法(默认都是合法的)。

③ 在初始化用户表 user 数据时,插入的用户密码 password 必须是对应编码器编码后的密码,如示例中的密码就是加密后的形式(对应的原始密码为 123456)。因此,在自定义配置类中进行用户密码查询时,必须使用与数据库密码统一的密码编码器进行编码。

④ 在初始化角色表 role 数据时,角色 rolename 值必须带有"ROLE_"前缀,而默认的用户角色值则是对应权限值去掉"ROLE_"前缀。这是因为之前的 role 都是通过 springsecurity 的 api 进行赋值的,它会自行加上该前缀。

(2) 添加 JDBC 连接数据库的依赖驱动启动器。

打开项目 springboot0501 中的 pom.xml 文件,在该文件中添加对应的依赖。

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.30</version>
    </dependency>
```

(3) 进行数据库连接配置。

在项目的全局配置文件 application.properties 中编写对应的数据库连接配置。

```
spring.datasource.url=jdbc:mysql://localhost:3306/book
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

(4) 使用 JDBC 进行身份认证。

打开项目 springboot0501,创建 SecurityConfig 类并继承 WebSecurityConfigurerAdapter,在重写的 configure(AuthenticationManagerBuilder auth)方法中使用 JDBC 身份认证。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Autowired
    private DataSource datasource;
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        PasswordEncoder encoder = new BCryptPasswordEncoder();
        String userSql = "select id,password,valid from user where id=?";
        String authoritySql = "SELECT u.id,r.rolename FROM user AS u ,role AS r ,user_role AS
ur WHERE u.id = ? AND u.id = ur.user_id AND r.id = ur.role_id";
        auth.jdbcAuthentication().passwordEncoder(encoder)
                .dataSource(datasource)
                .usersByUsernameQuery(userSql)
                .authoritiesByUsernameQuery(authoritySql);
    }
}
```

需要注意的是，在定义用户查询的 SQL 语句时，必须返回用户名 id、密码 password、是否为有效用户 valid 这 3 个字段信息；在定义权限查询的 SQL 语句时，必须返回用户名 id、角色 rolename 这两个字段信息。否则，登录时输入正确的用户信息会出现 PreparedStatement Callback 的 SQL 异常错误信息。

实际项目中并不会把密码明文存储在数据库中。默认使用的 PasswordEncoder 要求数据库中的密码格式为{id}password，它会根据 id 去判断密码的加密方式，但是一般不会采用这种方式。通常需要替换 PasswordEncoder，使用 Spring Security 提供的 BCryptPasswordEncoder，将 BCryptPasswordEncoder 对象注入 Spring 容器中，Spring Security 就会使用该 PasswordEncoder 进行密码校验。

(5) 运行程序，效果如图 5-1 所示。

5.3.3 UserDetailsService 身份认证


视频讲解

采用配置文件的方式可以从数据库中读取用户进行登录。虽然该方式相较于静态账号密码的方式更加灵活，但是将数据库的结构暴露在明显的位置上，绝对不是一个明智的做法。接下来将通过 UserDetailsService 接口实现身份认证。

Spring Security 中进行身份验证的是 AuthenticationManager 接口，ProviderManager 是该接口的一个默认实现，但它并不用来处理身份认证，而是委托给配置好的 AuthenticationProvider，每个 AuthenticationProvider 会轮流检查身份认证，在检查后返回 Authentication 对象或抛出异常。

验证身份就是加载响应的 UserDetails，比对是否与用户输入的账号、密码、权限等信息匹配。该步骤由实现 AuthenticationProvider 的 DaoAuthenticationProvider（它利用 UserDetailsService 验证用户名、密码和授权）处理，包含 GrantedAuthority 的 UserDetails 对象在构建 Authentication 对象时填入数据，如图 5-8 所示。

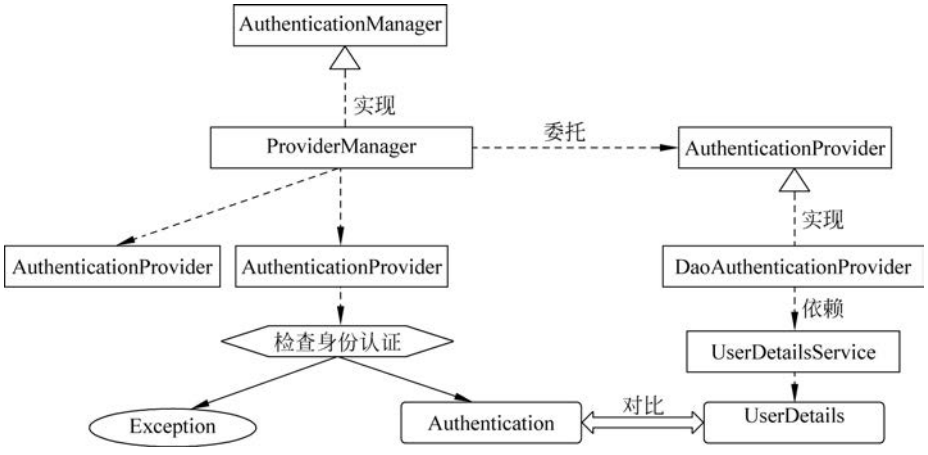

图 5-8 UserDetailsService 身份认证

正常的登录流程可能就是前端传过来账号密码，后端收到账号密码，直接通过账号密码的两个条件进行数据库查询，查询到就登录成功，查询不到就登录失败。SpringSecurity 并非如此，其流程如下：

(1) 前端将账号和密码传送给后端(这里直接以明文举例)。

(2) 后端通过输入账号获取用户信息(获取不到则证明该账号不存在)。

(3) 获取信息后进行密码比对,看看是否正确(该过程称为身份认证)。

UserDetailsService 接口的主要作用就是该流程的第二个步骤。

下面通过 Spring Boot 整合 Spring Security 实现 UserDetailsService 身份认证。

(1) 创建数据库实体类。

User 类:

```
@Entity(name = "user")
@Data
public class User implements Serializable {
  @Id
  private String id;
  private String password, username, address, email, phone;
  @ManyToMany
  @JoinTable(name = "user_role", joinColumns = @JoinColumn(name = "user_id"),
             inverseJoinColumns = @JoinColumn(name = "role_id"))
  private List<Role> roles;
}
```

Role 类:

```
@Data
@Entity(name = "role")
public class Role implements Serializable {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;
  private String rolename;
}
```

(2) 创建 UserRepository。

```
public interface UserRepository extends JpaRepository<User, String> {
}
```

(3) 定义查询用户及角色信息的服务接口。

```
@Service
public class UserServiceImpl implements UserService {
  @Autowired
  private UserRepository dao;
  @Cacheable(cacheNames = "user",key = "#id")
  @Override
  public User findByid(String id) {
      User user = dao.getById(id);
      return user;
  }
}
```

(4) 定义 UserDetailsService 接口,用于封装认证用户信息。

UserDetailsService 接口只有一个方法,该方法返回值为 UserDetails,UserDetails 类是系统默认的用户“主体”。在用户登录时会访问该方法,并将登录传入的用户名以参数形式进行传送。为此需要做的就是实现 UserDetailsService 接口,然后重写这个方法,并返回一个 UserDetails 对象。

定义 UserDetailsService 接口的步骤如下。

① 根据登录用户名查询数据库，判断该用户是否存在。

② 将从数据库查询出的账号密码封装到 UserDetails 对象中，作为方法返回值返回。

```
@Service
@Qualifier
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UserService service;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
      it.com.boot.springboot05_01.po.User user = service.findByid(username);
     if (user!= null) {
      List < SimpleGrantedAuthority > authorities = new ArrayList <>();
      for (Role role : user.getRoles()) {
        SimpleGrantedAuthority simpleGrantedAuthority = new SimpleGrantedAuthority(role.
getRolename());
        authorities.add(simpleGrantedAuthority);
      }
      UserDetails userDetails = new User(username, user.getPassword(), authorities);
      return userDetails;
   }
    else
      throw new UsernameNotFoundException("当前用户不存在!");
 }
}
```

需要注意的是，在 UserDetailsServiceImpl 业务处理类获取 User 实体类时，必须对当前用户进行非空判断。这里使用 throw 进行异常处理，如果查询的用户为空，throw 会抛出 UsernameNotFoundException 的异常。如果没有使用 throw 异常处理，Security 将无法识别，导致程序整体报错。

UserDetails(位于 org.springframework.security.core.userdetails 包下)主要与用户信息有关，该接口是提供用户信息的核心接口。该接口仅能实现存储用户的信息，后续会将该接口提供的用户信息封装到认证对象 Authentication 中。

UserDetails 的子类 User(位于 org.springframework.security.core.userdetails 包下)是用户类，用于存放待认证的用户信息。如从数据库查出的用户信息和从缓存中获取的用户信息都可以封装到此类中，最终交给 SpringSecurity 进行用户信息认证。UserDetails 实体类的结构如图 5-9 所示。

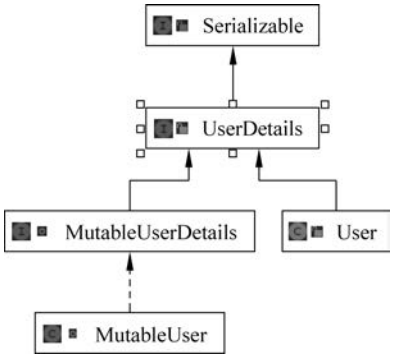

图 5-9 UserDetails 实体类的结构

(5) 使用 UserDetailsService 进行身份认证。

打开项目 springboot0501,创建 SecurityConfig 类并继承 WebSecurityConfigurerAdapter,在重写的 configure(AuthenticationManagerBuilder auth)方法中使用 JDBC 身份认证。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Autowired
    @Qualifier
    private UserDetailsService userDetailsService;
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        PasswordEncoder encoder = new BCryptPasswordEncoder();
        auth.userDetailsService(userDetailsService).passwordEncoder(encoder);
    }
}
```

(6) 运行程序,效果如图 5-1 所示。

5.4 自定义用户授权管理

5.3 节的案例只是做了请求认证,并没有对权限进行控制。下面基于 5.3 节的案例进行权限控制处理。

所谓权限控制,以一个学校图书馆的管理系统为例进行分析。如果是普通学生登录,能看到借书还书相关的功能,但不能看到或使用添加书籍信息、删除书籍信息等功能。如果是图书馆管理员登录,就能看到并使用添加书籍信息、删除书籍信息等功能。

总结起来就是不同的用户可以使用不同的功能,这就是权限系统要实现的效果。

在权限控制的实现过程中,不能只依赖前端判断用户的权限。因为如果只是这样,当有人知道了对应功能的接口地址时,就可以不通过前端而直接发送请求实现相关功能操作。所以还需要在后台进行用户权限的判断,判断当前用户是否有相应的权限,必须具有所需权限才能进行相应的操作。

权限控制系统一般都会分为角色控制和菜单控制,菜单控制又分为页面菜单和按钮控制。所谓按钮控制,就是指定 Java 某个接口必须具备什么角色,或者具备按钮权限才可以访问。

在一些比较早的项目中没用到 Spring Security,当时的做法如下:

前端在登录时访问后端,获取该用户有哪些权限,这个权限包含菜单权限和按钮权限,当不具备某个按钮权限时直接屏蔽。拦截器只有一层拦截,即登录成功的人就能访问所有接口,对于普通人来说该功能只是程序员设置的假象。一旦登录成功,只要知道接口名称,便可以通过接口直接访问。

Spring Security 只需要通过简单的配置即可避免这样的问题。

5.4.1 授权基本流程

在 Spring Security 中,会使用默认的 FilterSecurityInterceptor 进行权限校验。FilterSecurityInterceptor 会获取 SecurityContextHolder 中的 Authentication,然后获取其

中的权限信息，包括当前用户是否拥有访问当前资源所需的权限。所以在项目中只需要把当前登录用户的权限信息存入 Authentication，然后设置资源所需要的权限即可。

在实际生产中，网站访问多是基于 HTTP 请求的，通过重写 WebSecurityConfigurerAdapter 类的 config(HttpSecurity http)方法可以对基于 HTTP 的请求访问进行控制。configure(HttpSecurity http)方法的参数类型是 HttpSecurity 类，HttpSecurity 类提供了 HTTP 请求的限制权限、Session 管理配置、CSRF 跨站请求问题等方法。

下面通过对 configure(HttpSecurity http)方法的剖析，分析自定义用户访问控制的实现过程。

1. 访问控制 URL 匹配

配置类中的 httpSecurity.authorizeRequests()主要是对 URL 进行控制，也就是授权(访问控制)。httpSecurity.authorizeRequests()支持连缀写法，可以有很多 URL 匹配规则和权限控制方法，这些内容进行各种组合就形成了 Spring Security 中的授权。

在所有匹配规则中取所有规则的交集。配置顺序会影响之后的授权效果，越是具体的配置越应该放在前面，越是模糊的配置越应该放到后面。

(1) antMatcher()。

antMatcher()的匹配规则如下：

① ? 匹配一个字符

② * 匹配 0 个或多个字符

③ ** 匹配 0 个或多个目录

在实际项目中经常需要放行所有静态资源，下面的代码表示放行 js 文件夹下的所有脚本文件。

```
.antMatchers("/js/**").permitAll()
```

还有一种配置方式是放行所有.js 文件。

```
antMatchers("/**/*.js").permitAll()
```

(2) anyRequest()。

在之前的认证过程中使用的是 anyRequest()，表示匹配所有的请求。一般情况下都会使用该方法，并设置全部内容都需要进行认证。

代码示例：

```
anyRequest().authenticated();
```

2. 内置访问控制

Spring Security 在匹配 URL 后会调用 permitAll()，表示不需要认证即可随意访问。Spring Security 提供了多种内置控制，其底层都是基于 access 进行实现的。

(1) permitAll()。

permitAll()表示所匹配的 URL 允许任何人访问。

```
public ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry permitAll() {
    return this.access("permitAll");
}
```

(2) authenticated()。

authenticated()表示所匹配的 URL 需要被认证才能访问。

```
public ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
authenticated() {
    return this.access("authenticated");
}
```

(3) anonymous()。

anonymous()表示可以匿名访问匹配的 URL。与 permitAll()效果类似,但设置为 anonymous()的 URL 会执行 filter 链中的语句。

```
public ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
anonymous() {
    return this.access("anonymous");
}
```

(4) denyAll()。

denyAll()表示所匹配的 URL 不允许被访问。

```
public ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry denyAll() {
    return this.access("denyAll");
}
```

(5) rememberMe()。

rememberMe()表示被“记住”的用户允许访问。

```
public ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
rememberMe() {
    return this.access("rememberMe");
}
```

(6) fullyAuthenticated()。

fullyAuthenticated()表示不被“记住”的用户才可以访问。

```
public ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
fullyAuthenticated() {
    return this.access("fullyAuthenticated");
}
```

3. 角色权限判断

除了之前讲解的内置权限控制外,Spring Security 还支持很多其他权限控制。这些方法一般都用于用户已经被认证后,判断用户是否具有特定的要求。

底层也是调用 access(参数),参数正好是调用的方法名。需要注意的是,判断角色会给调用方法参数前面添加 ROLE_,这也是为什么正常调用方法时角色不允许以 ROLE_开头命名的原因。

(1) hasAuthority(String)。

判断用户是否具有特定的权限,用户的权限是在自定义登录逻辑中创建 User 对象时指定的。

```
List<String> listPermission = userMapper.selectPermissionByUsername(username);
List<SimpleGrantedAuthority> listAuthority = new ArrayList<SimpleGrantedAuthority>();
for(String permisssion : listPermission){
    listAuthority.add(new SimpleGrantedAuthority(permisssion));
}
```

在配置类中通过 hasAuthority("admin")设置具有 admin 权限时才能访问,代码如下。

```
.antMatchers("/main1.html").hasAuthority("admin")
```

(2) hasAnyAuthority(String)。

如果用户具备给定权限中的某一个就允许访问。

下面代码中由于大小写和用户的权限不相同,所以用户无权访问/main1.html。

```
.antMatchers("/main1.html").hasAnyAuthority("adMin","admiN")
```

(3) hasRole(String)。

如果用户具备给定角色就允许访问,否则出现 403 错误提示。

参数取值来源于自定义登录逻辑 UserDetailsService 实现类中创建 User 对象时给 User 赋予的授权。在给用户赋予角色时,角色需要以 ROLE_开头,其后添加角色名称。例如,ROLE_abc,其中 abc 是角色名,ROLE_是固定的字符开头。

```
List<String> listPermission = userMapper.selectPermissionByUsername(username);
List<String> listRoles = userMapper.selectRoleByUsername(username);
for(String role: listRoles){
    listAuthority.add(new SimpleGrantedAuthority("ROLE_" + role));
}
```

在配置类中添加代码如下。

```
.antMatchers("/bjsxt").hasRole("abc")
```

(4) hasAnyRole(String …)。

如果用户具备给定角色的任意一个就允许访问。

5.4.2 自定义登录页面


视频讲解

虽然 Spring Security 提供了登录页面,但是用户在实际项目中大多喜欢使用自己的登录页面。因此 Spring Security 不仅提供了登录页面,而且支持用户自定义登录页面。实现过程也比较简单,只需要修改配置类即可,具体步骤如下。

(1) 编写登录页面。

编写登录页面,其中<form>的 action 可以不编写对应控制器。

```
<form class="form-signin" th:action="@{/tologin}" th:method="post">
    <img class="mb-4" th:src="@{/login/img/logo.gif}" width="72px" height="72px">
    <h1 class="h3 mb-3 font-weight-normal">请登录</h1>
    <!-- 用户登录错误信息提示框 -->
    <div th:if="${param.error}" style="color: red;height: 40px;text-align: left;font-size: 1.1em">
        <img th:src="@{/login/img/loginError.jpg}" width="20px">用户名或密码错误,请重新登录!
    </div>
    <input type="text" name="name" class="form-control" placeholder="用户名" required="" autofocus="">
    <input type="password" name="pwd" class="form-control" placeholder="密码" required="">
    <button class="btn btn-lg btn-primary btn-block" type="submit">登录</button>
    <p class="mt-5 mb-3 text-muted">Copyright © 2019-2020</p>
</form>
```

(2) 编写控制器。

在创建的 LoginController 类中添加一个跳转到登录页面 login.html 的方法。

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

在上述添加的 toIndex()方法中,配置了请求路径为/toindex 的 GET 请求,并向静态资源根目录下的 login.html 页面跳转。

Spring Security 默认向登录页面跳转时,采用的请求方式是 GET,请求路径是/login;如果要处理登录后的数据,默认采用的请求方式是 POST,请求路径是 login。

表单处理成功会跳转到一个地址,失败也会跳转到一个地址。在控制器类中添加控制器方法,方法映射路径为/fail。此处需要注意的是,如果是 POST 请求访问/fail,一旦返回值直接转发到 fail.html 中,即使有效果,控制台也会报警提示 fail.html 不支持 POST 访问方式。

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

(3) 创建登录失败页面 fail.html。

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>网上书店</title>
</head>
<body>
<h1 align="center">欢迎进入网上书店</h1>
<div>
    操作失败,请重新登录. <a href="/toindex">跳转</a>
</div>
</body>
</html>
```

(4) 修改配置类。

修改配置类主要是设置哪个页面是登录页面,配置类需要继承 WebSecurityConfigurerAdapter,并重写 configure 方法。

```
@Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/").permitAll()
                .antMatchers("/login/**").permitAll()
                .anyRequest().authenticated();
        http.formLogin()
        //定义登录时用户名的 key,默认为 username
                .usernameParameter("name")
        //定义登录时用户密码的 key,默认为 password
                .passwordParameter("pwd")
        //登录页面表单提交地址,此地址可以不真实存在
```

```
                .loginProcessingUrl("/tologin")
        //登录成功之后跳转到哪个 URL
                .defaultSuccessUrl("/")
        //登录失败之后跳转到哪个 URL
                .failureUrl("/fail").permitAll()
        //登录页面
                .loginPage("/toindex").permitAll();
        /*
        http.formLogin()
            .usernameParameter("name")
            .passwordParameter("pwd")
            .loginProcessingUrl("/tologin")
            .defaultSuccessUrl("/")
            .failureUrl("/toindex?error")
            .loginPage("/toindex").permitAll();
         */
}
```

failureUrl("/toindex?error")方法用来控制用户登录认证失败后的跳转路径,该方法默认参数为"/login?error"。其中,参数中的"/toindex"为向登录页面跳转的映射;error 是一个错误标识,作用是登录失败后在登录页面进行接收判断,如 login.html 实例中的${param.error},这两者必须保持一致。

(5) 运行项目。

自定义登录页面如图 5-10 所示。输入正常的用户名和密码,单击“登录”按钮,跳转到首页,如图 5-11 所示。

图 5-10 自定义登录页面

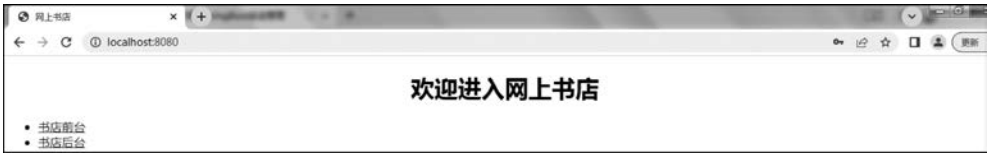

图 5-11 网上书店首页

当输入用户名或密码有误时,如果配置文件中配置的是跳转到 fail.html,单击“登录”按钮,如图 5-12 所示。

图 5-12　登录失败页面

如果配置文件中配置的是跳转到登录页面，且同时携带一个错误标识 error，单击“登录”按钮，如图 5-13 所示。

图 5-13　错误提示

在传统项目中进行用户登录处理时，通常会查询用户是否存在，如果存在则登录成功，同时将当前用户放在 Session 中。

Spring Security 针对拦截的登录用户专门提供了一个 SecurityContextHolder 类，该类存储了当前与系统交互的用户信息。Spring Security 使用一个 Authentication 对象来表示这些信息。一般不需要自己创建这个对象，但是查找这个对象的操作对用户来说却十分常见。

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
```

(6) 修改配置文件，完成登录成功后，在后台获取用户名和权限信息。

```
@Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/").permitAll()
                .antMatchers("/login/**").permitAll()
                .anyRequest().authenticated();
        http.formLogin()
                .usernameParameter("name")
                .passwordParameter("pwd")
                .loginProcessingUrl("/tologin")
                .successHandler(new AuthenticationSuccessHandler() {
                @Override
                 public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse
response, Authentication authentication) throws IOException, ServletException {
                //第一种方法
```

```
                System.out.println(authentication.getName());
                authentication.getAuthorities().forEach(System.out::println);
                System.out.println("------------------------");
                //第二种方法
                UserDetails userDetails = (UserDetails) authentication.getPrincipal();
                System.out.println(userDetails.getUsername());
                userDetails.getAuthorities().forEach(System.out::println);
                response.sendRedirect("/");
                }
            })
                .failureUrl("/toindex?error")
                .loginPage("/toindex").permitAll();
    }
```

(7) 再次运行项目。

再次运行项目并正确登录后,在控制台输出当前登录的用户信息,如图 5-14 所示。

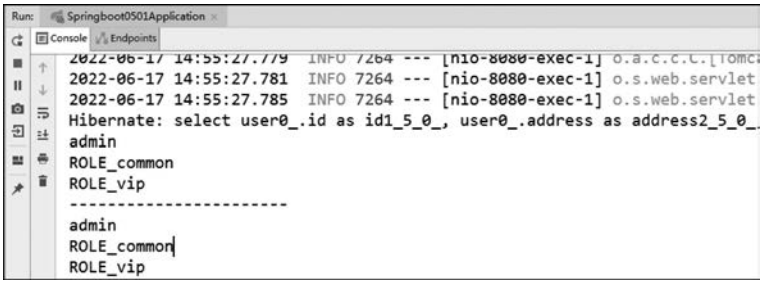

图 5-14 控制台输出登录用户信息

5.4.3 权限控制和注销


视频讲解

在 Spring Security 框架下实现用户的“退出”logout 的功能其实非常简单,具体实现步骤如下。

(1) 修改配置类。

```
@Override
protected void configure(HttpSecurity http) throws Exception {
      http.logout();
}
```

(2) 在 indexhtml 页面中增加一个注销按钮。

```
<form th:action="@{/logout}" method="post">
    <input type="submit" value="注销">
</form>
```

HttpSecurity 类的 logout()方法用来处理用户退出,它默认处理路径为/logout 的 POST 类型请求,同时也会清除 Session 和 Remember Me 等任何默认用户配置。

(3) 运行测试,在登录成功后单击“注销”按钮,发现注销完成后会跳转到登录页面。

(4) 进行个性化配置。

Spring Security 默认使用/logout 作为退出处理请求路径,将登录页面作为退出后的跳转页面。这符合绝大多数的应用的开发逻辑,但有的时候我们需要一些个性化设置。

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.logout().logoutUrl("/mylogout").logoutSuccessUrl("/");
}
```

通过指定 logoutUrl 配置改变退出请求的默认路径，当然 HTML 退出按钮的请求 URL 也要修改，同时通过指定 logoutSuccessUrl 配置显式指定退出后的跳转页面。

(5) 运行项目。

在项目首页上方已经出现了新添加的用户退出链接，如图 5-15 所示。单击“注销”按钮后，发现跳转到首页。

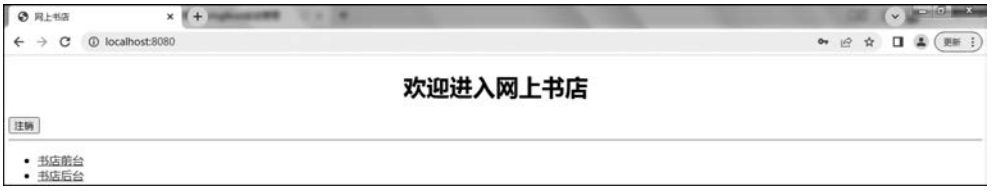

图 5-15　用户注销

现在提出一个需求：对应不同权限的用户应该能够访问不同的页面请求。例如，sili 这个用户拥有 common 权限，所以只能查看书店前台而不能访问书店后台。

(6) 修改配置类，对书店前后台访问的控制器设置不同的权限。

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/").permitAll()
            .antMatchers("/login/**").permitAll()
            .antMatchers("/loginadmin").hasAnyRole("vip","vvip")
            .antMatchers("/loginbooks").hasRole("common")
            .anyRequest().authenticated();
```

(7) 运行项目。

以普通身份 zhaosi 访问书店前台，如图 5-16 所示。然后以该身份访问书店后台，此时由于身份权限的设置，页面会发出 403 拒绝响应，如图 5-17 所示。

图 5-16　以普通身份访问书店前台

接下来再提出几个需求：在用户没有登录时，导航栏上只显示登录按钮；在用户登录后，导航栏可以显示登录的用户信息及注销按钮；当用户只有普通权限时，登录后只能显示出书店前台的功能，而不显示书店后台的功能。这就是真实的网站情况。该如何完成需求

图 5-17 403 拒绝响应

呢？这里需要结合 Thymeleaf 中的一些功能。

(8) 导入 Thymeleaf 和 Security 结合的 Maven 依赖。

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

(9) 修改前端页面 index.html。

① 导入命名空间。

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```

② 修改导航栏,增加认证判断。

```
<body>
<h1 align="center">欢迎进入网上书店</h1>
<div sec:authorize="isAnonymous()">
    <h2 align="center">访问您好,请先<a th:href="@{/toindex}">登录</a></h2>
</div>
<div sec:authorize="isAuthenticated()">
    <h2 align="center"><span sec:authentication="name"></span> 你好,你的权限是<span
sec:authentication="authorities"/>,你能够访问以下内容。
    </h2>
    <h2 align="center"><span sec:authentication="principal.username"></span>你好,你的
权限是<span sec:authentication="principal.authorities"/>,你能够访问以下内容。</h2>
</div>
<form th:action="@{/mylogout}" method="post">
    <input type="submit" value="注销">
</form>
<hr>
<div>
    <ul>
        <li sec:authorize="hasRole('common')"><a th:href="@{/loginbooks}">书店前台</a>
</li>
        <li sec:authorize="hasAnyRole('vip','vvip')"><a th:href="@{/loginadmin}">书店后
台</a></li>
    </ul>
</div>
</body>
```

页面顶部通过"xmlns:sec"引入了 Security 安全标签库。常用的标签库如下。

- sec:authorize 权限
- sec:authentication 认证
- sec:authorize-url 不能直接使用,需要额外的配置(不建议使用该标签,因为该标签不支持 RESTful 风格)

- SecurityExpressionRoot 常用的表达式
- boolean hasAuthority(String authority)
- boolean hasAnyAuthority(String authorities)
- boolean hasRole(String role)
- hasAnyRole(String roles)
- Authentication getAuthentication()
- Object getPrincipal()
- boolean isAnonymous()
- boolean isAuthenticated()
- boolean isRememberMe()

下面对 模块的作用及内部属性进行详细说明。

sec:authorize="isAnonymous()"属性,判断用户是否未登录,只有匿名用户(未登录用户)才会显示登录链接提示。

sec:authorize="isAuthenticated()"属性,判断用户是否已登录,只有认证用户才会显示登录用户信息和注销链接等提示。

sec:authorize="hasRole('common')"属性,定义了只有角色为 common(对应权限 Authority 为 ROLE_common)且登录的用户才会显示书店前台列表信息。

sec:authorize="hasAuthority('ROLE_vip')"属性,定义了只有权限为 ROLE_vip(对应角色 Role 为 vip)且登录的用户才会显示书店后台列表信息。

sec:authorize="hasAnyRole('vip','vvip')"属性,定义了只有权限为 vip 或 vvip,且登录的用户才会显示书店后台列表信息。

Spring Security 首先利用 BeanWrapperImpl 封装了 Authentication 对象,然后调用 BeanWrapperImpl 的 getPropertyValue()方法获取 property 属性的值。而 BeanWrapperImpl 类能够通过 name-value 值对的方式对目标对象(Authentication 对象)进行属性(属性可以为嵌套属性)操作,所以 property 的取值可以是 Authentication 的直接属性或嵌套属性。

sec:authentication="name"和 sec:authentication="principal.username" 这两个属性都是获取当前登录用户的用户名。

sec:authentication="authorities"和 sec:authentication="principal.authorities"这两个属性都是获取当前登录用户的所有角色。

(10) 运行项目。

当用户以匿名身份进行首页登录时,效果如图 5-18 所示。从图 5-18 可以看出,此次访问项目首页时,页面上方出现“请先登录”的链接,页面中不再显示书店前后台的链接,说明页面安全访问控制实现了效果。接着单击“登录”链接,输入正确的用户名和密码,登录成功

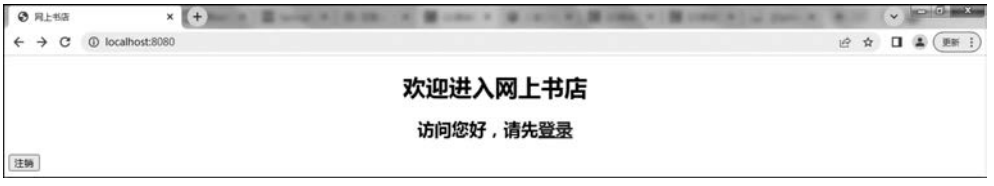

图 5-18 未登录访问首页效果

后跳转到项目首页。由于用户 zhaosi 只拥有 common 权限，所以只能显示书店前台选项，如图 5-19 所示。

图 5-19 登录后访问首页效果

5.4.4 “记住我”及首页定制

现在网上书店的情况是，只要登录后关闭浏览器，再登录时就会要求重新登录。很多网站的有一个记住密码的功能，这个该如何实现呢？记住密码功能“记住我”的实现过程如图 5-20 所示。

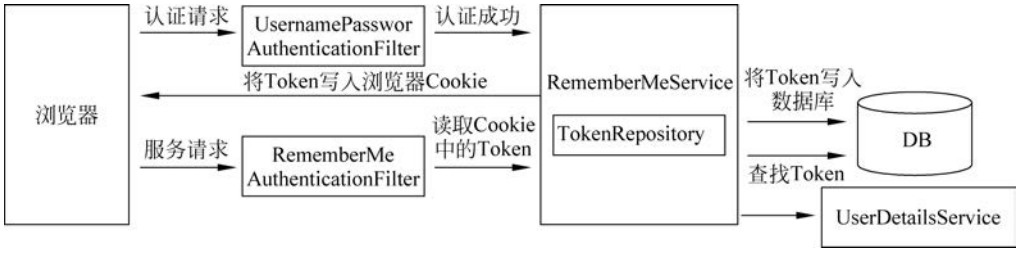

图 5-20 “记住我”的实现过程

(1) 浏览器在发送用户请求时，会到 UsernamePasswordAuthenticationFilter 进行认证，该过滤器认证成功后会调用 RememberMeService 服务。

(2) RememberMeService 会生成一个 Token，并把这个 Token 写入浏览器的 Cookie 里面。注意此时在客户端的 Cookie 中，仅保存一个无意义的加密串(与用户名、密码等敏感数据无关)。同时 RememberMeService 会用 TokenRepository 将 Token 存入数据库中，因为是认证成功后进行存库操作，所以 SpringSecurity 会将用户名也写入库中。用户名与 Token 是一一对应的。

(3) 第二天再访问系统请求时会经过一个 RememberMeAuthenticationFilter 过滤器，这个过滤器的作用就是读取 Cookie 中的 Token 并将其交给 RememberMeService。RememberMeService 会根据 Token 查询数据库中是否有记录，有记录则将用户名取出并调用 UserDetailsServicer 将其放入 SecurityContext 中，此时登录成功。

下面继续完善前面的项目，并添加“记住我”这个功能。

(1) 创建一张记录 RememberMe 的表，用于记录令牌信息。

```
create table persistent_logins (username varchar(64) not null,
                                series varchar(64) primary key,
                                token varchar(64) not null,
                                last_used timestamp not null);
```

这张表的名称和字段都是官方提供的固定模板,可以使用默认的 JDBC,即 JdbcTokenRepositoryImpl 来实现“记住我”功能。这张表也可以完全自定义,还可以使用系统提供的 JDBC 来操作。

(2) 编写登录页面。

在 login.html 登录页面中添加“记住我”选项,其中“记住我”的选择框是 checkbox 类型的多选框,它的 name 属性可以选择默认值 rememberme。

```
<div class = "checkbox mb - 3">
  <label >
    <input type = "checkbox" name = "rememberme"> “记住我”选项
  </label >
</div >
```

(3) 添加依赖。

Spring Security 在实现 RememberMe 功能时,其底层实现依赖 SpringJDBC,所以需要导入 SpringJDBC。由于此项目使用的是 JPA,所以此处导入 JPA 启动器,同时还需要添加 MySQL 驱动。

```
<dependency >
    <groupId > org.springframework.boot </groupId >
    <artifactId > spring - boot - starter - data - jpa </artifactId >
</dependency >
<dependency >
    <groupId > mysql </groupId >
    <artifactId > mysql - connector - java </artifactId >
    <version > 5.1.30 </version >
</dependency >
```

(4) 配置数据源。

在 application.properties 中配置数据源,这在前面已经配置完成,这里就不再赘述了。

(5) 编写配置文件。

配置 TokenRepository 并在 configure 中指定 RememberMe 需要的配置包含 TokenRepository 对象和 Token 过期时间。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private DataSource datasource;
    @Override
    protected void configure(HttpSecurity http) throws Exception {
       http.rememberMe().rememberMeParameter("rememberme")
.tokenValiditySeconds(60).tokenRepository(persistentTokenRepository());
    }
    @Bean
    public PersistentTokenRepository persistentTokenRepository() {
        JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();
        tokenRepository.setDataSource(datasource);
        return tokenRepository;
    }
}
```

其中 rememberMeParameter("rememberme")方法指定了“记住我”勾选框的 name 属性，如果页面中使用了默认的 rememberme，则该方法可以省略。tokenValiditySeconds(60)方法设置了状态有效时间为 60 秒，默认为两周时间。

(6) 重启项目。

登录页面如图 5-21 所示。

图 5-21 “记住我”登录页面

勾选并成功登录后，可以看到网页多了一个 RememberMe 的 Cookie 对象，如图 5-22 所示。

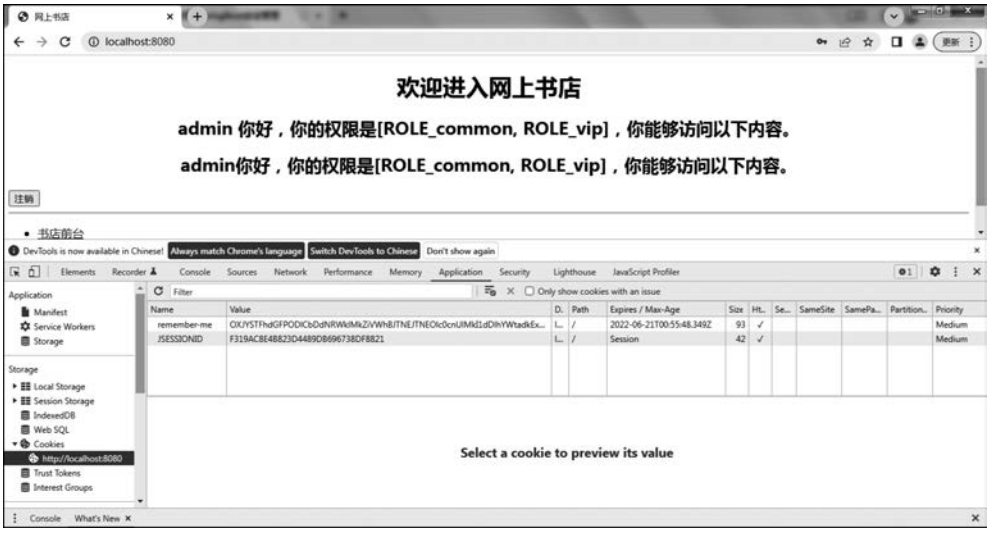

图 5-22 Cookie 对象

查看数据库表 persistent_logins，如图 5-23 所示。

可以看到 Token 信息已经成功持久化，并且浏览器也成功生成了相应的 Cookie。在 Cookie 未失效之前，无论是重开浏览器还是重启项目，用户都无须再次登录就可以访问系统资源。

图 5-23　数据库表 persistent_logins

本章小结

本章主要讲解了 Spring Boot 的 Spring Security 安全管理。首先介绍了 Spring Security 安全框架和 Spring Boot 支持的安全管理，并体验了 Spring Boot 默认的安全管理；然后讲解了 Spring Security 自定义用户认证和授权管理；最后介绍了 Security 与前端的整合实现页面安全管理控制。希望大家通过本章的学习，能够掌握 Spring Boot 的安全管理机制，并灵活运用在实际开发中，提升项目的安全性。

在线测试

习题

一、单选题

1. 以下关于@EnableWebSecurity 注解的说法，正确的是(　　)。
   A. @EnableWebSecurity 注解是一个组合注解，开启基于 WebFluxSecurity 的安全支持
   B. 在安全配置类上使用@EnableWebSecurity 注解后，无须使用@Configuration 注解
   C. 是针对 SpringWebFlux 框架的安全支持，只需要替换使用@EnableWebFluxSecurity 注解即可
   D. 以上说法都错误
2. 以下关于自定义用户退出 logout()方法及其说明，错误的是(　　)。
   A. 它默认处理路径为“/logout”的 POST 类型请求
   B. 自定义用户退出功能，必须使用 POST 方式的 HTTP 请求进行用户注销
   C. logoutUrl()方法指定了用户退出的请求路径，可以省略
   D. 在用户退出后，用户会话信息则会默认清除
3. 以下关于基于简单加密 Token 方式的“记住我”的说法，错误的是(　　)。
   A. 基于简单加密 Token 的方式中的 Token 在指定的时间内有效
   B. 必须保证 Token 中所包含的 username、password 和 key 没有被改变
   C. 任何人获取到该“记住我”功能的 Token 后，都可以无限制进行自动登录
   D. 在 Token 有效期过后再次访问项目时，会发现又需要重新进行登录认证

4. 以下关于 Spring Boot 整合 Security 的说法，错误的是(　　)。

A. Spring Boot 一旦引入 spring-boot-starter-security，无须配置，Spring Security 即可生效

B. Spring Boot 整合 Security 项目启动时会在控制台 Console 中自动生成一个安全密码，每次都不一样

C. 访问 Spring Boot 项目默认首页 index.html，无须登录

D. 在 Spring Security 登录页面"/login"中输入错误登录信息后，会重定向到"/login?error"页面

5. 以下使用 JDBC 身份认证方式创建用户/权限表及初始化数据的说法，错误的是(　　)。

A. 用户表中用户名必须唯一

B. 用户表必须提供一个 tinyint 类型的字段

C. 用户角色值是对应权限值加上"ROLE_"前缀

D. 用户表中插入的用户密码 password 必须是对应编码器编码后的密码

二、多选题

1. 以下关于 Security 中基于持久化 Token 方式的"记住我"的说法，正确的是(　　)。

A. 选择"记住我"并成功登录后，会把 username、随机产生的序列号、生成的 Token 进行持久化存储

B. 当用户再次访问系统时，将重新生成一个新的 Token 替换数据库中旧的 Token

C. 如果再次登录的 Cookie 中的 Token 不匹配，Spring Security 将删除数据库中与当前用户相关的所有 Token 记录

D. 如果用户访问系统时没有携带 Cookie，那么将会引导用户到登录页面

2. Spring Boot 整合 Spring Security 安全框架中包含的安全管理功能包括(　　)。

A. WebFlux Security　　B. MVC Security

C. OAuth2　　D. Actuator Security

3. 以下关于 configure()方法中使用 JDBC 身份认证的方式进行自定义用户认证相关说法，正确的是(　　)。

A. 要引入 DataSource 数据源

B. 使用 JDBC 身份认证时，首先需要对密码进行编码设置

C. 在定义用户查询的 SQL 语句时，必须返回用户名 username 和密码 password 两个字段信息

D. 在定义权限查询的 SQL 语句时，必须返回用户名 username、角色 role、权限 authority 三个字段信息

4. 针对自定义用户认证，SpringSecurity 提供了多种自定义认证方式，包括(　　)。

A. In-Memory Authentication(内存身份认证)

B. JDBC Authentication(JDBC 身份认证)

C. LDAP Authentication(LDAP 身份认证)

D. UserDetailsService(身份详情服务)

5. 以下关于 Security 与 Thymeleaf 整合实现前端页面管理的相关标签及属性的说法，

错误的是(　　)。

A. 页面顶部通过“xmlns:sec”引入了 Security 安全标签

B. 使用 sec:authorize="!isAuthenticated()"属性判断用户是否未登录

C. 使用 sec:authorize="hasRole('common')"属性判断用户是否有 ROLE_common 权限

D. 使用 sec:authentication="principal.authorities"属性可以获取登录用户角色

三、判断题(对的打“√”,错的打“×”)

1. 自定义 WebSecurityConfigurerAdapter 类型的 Bean 组件,会同时关闭 UserDetailsService 用户信息自动配置类。(　　)

2. 自定义的登录页跳转路径必须与数据处理提交路径一致。(　　)

3. rememberMeParameter()方法用于指定“记住我”勾选框的 name 属性值,可以省略。(　　)

4. 持久化 Token 的方式比简单加密 Token 的方式相对更加安全,不会存在安全问题。(　　)

5. Security 会默认提供一个可登录的用户信息,其中用户名为 user,密码为 root。(　　)

四、填空题

1. 在 Spring Security 默认登录页面中输入错误登录信息后会重定向到_____页面。

2. In-Memory Authentication(内存身份认证)是最简单的身份认证方式,主要用于_____。

3. 用户请求控制相关方法中的 permitAll()方法用于表示_____。

4. Security 控制登录的用户信息被封装在_____类对象中。

5. _____是 Security 提供的进行认证用户信息封装的接口。

五、简答题

1. 简述在 Spring Boot 项目中添加 Security 整合 Thymeleaf 进行前端页面管理依赖要注意的问题。

2. 对 WebSecurityConfigurerAdapter 类的两个主要方法进行简要介绍。