
# SpringSecurity

本章我们会一边讲解SpringSecurity框架，一边从头开始编写图书管理系统。

SpringSecurity是一个基于Spring开发的非常强大的权限验证框架，其核心功能包括：

- 认证 （用户登录）
- 授权 （此用户能够做哪些事情）
- 攻击防护 （防止伪造身份攻击）

我们为什么需要使用更加专业的全新验证框架，还要从CSRF说起。

## CSRF跨站请求伪造攻击

我们时常会在QQ上收到别人发送的钓鱼网站链接，只要你在上面登陆了你的QQ账号，那么不出意外，你的号已经在别人手中了。实际上这一类网站都属于恶意网站，专门用于盗取他人信息，执行非法操作，甚至获取他人账户中的财产，非法转账等。而这里，我们需要了解一种比较容易发生的恶意操作，从不法分子的角度去了解整个流程。

我们在JavaWeb阶段已经了解了Session和Cookie的机制，在一开始的时候，服务端会给浏览器一个名为JSESSION的Cookie信息作为会话的唯一凭据，只要用户携带此Cookie访问我们的网站，那么我们就可以认定此会话属于哪个浏览器。因此，只要此会话的用户执行了登录操作，那么就可以随意访问个人信息等内容。

比如现在，我们的服务器新增了一个转账的接口，用户登录之后，只需要使用POST请求携带需要转账的金额和转账人访问此接口就可以进行转账操作：

```
@RequestMapping("/index")
public String index(HttpSession session){
    session.setAttribute("login", true);    //这里就正常访问一下index表示登陆
    return "index";
}
```

```
@RequestMapping(value = "/pay", method = RequestMethod.POST, produces =
"text/html;charset=utf-8") //这里要设置一下produces不然会乱码
@ResponseBody
public String pay(String account,
                  int amount,
                  @SessionAttribute("login") Boolean isLogin){
    if (isLogin) return "成功转账 ￥"+amount+" 给: "+account;
    else return "转账失败，您没有登陆！";
}
```

那么，大家有没有想过这样一个问题，我们为了搜索学习资料时可能一不小心访问了一个恶意网站，而此网站携带了这样一段内容：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>我是(恶)学(意)习网站</title>
</head>
<body>
    <div>
        <div>对不起，您还没有充值本站的学习会员，请先充值后再观看学习视频</div>
        <form action="http://localhost:8080/mvc/pay" method="post">
            <input type="text" name="account" value="hacker" hidden>
            <input type="number" name="amount" value="666666" hidden>
            <input type="submit" value="点我充值会员，观看完整视频">
        </form>
    </div>
</body>
</html>
```

注意这个页面并不是我们官方提供的页面，而是不法分子搭建的恶意网站。我们发现此页面中有一个表单，但是表单中的两个输入框被隐藏了，而我们看到的只有一个按钮，我们不知道这是一个表单，也不知道表单会提交给那个地址，这时整个页面就非常有迷惑性了。如果我们点击此按钮，那么整个表单的数据会以POST的形式发送给我们的服务端（会携带之前登陆我们网站的Cookie信息），但是这里很明显是另一个网站跳转，通过这样的方式，恶意网站就成功地在我们毫不知情的情况下引导我们执行了转账操作，当你发现上当受骗时，钱已经被转走了。

而这种构建恶意页面，引导用户访问对应网站执行操作的方式称为：**跨站请求伪造**（CSRF，Cross Site Request Forgery）

显然，我们之前编写的图书管理系统就存在这样的安全漏洞，而SpringSecurity就很好地解决了这样的问题。

---

# 开发环境搭建

我们依然使用之前的模板来搭建图书管理系统项目。

导入以下依赖：

```
<!-- 建议为各个依赖进行分类，到后期我们的项目可能会导入很多依赖，添加注释会大幅度提高阅读效率 -
->
<dependencies>
```

```
    <!--  Spring框架依赖  -->
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>5.5.3</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>5.5.3</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.14</version>
    </dependency>

    <!--  持久层框架依赖  -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.27</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>2.0.6</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.7</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.3.14</version>
        </dependency>
        <dependency>
            <groupId>com.zaxxer</groupId>
            <artifactId>HikariCP</artifactId>
            <version>3.4.5</version>
        </dependency>

    <!--  其他工具框架依赖：Lombok、Slf4j  -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.22</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-jdk14</artifactId>
        <version>1.7.32</version>
    </dependency>
```

```
    <!--  ServletAPI  -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>

    <!--  JUnit依赖  -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

接着创建Initializer来配置Web应用程序：

```
public class MvcInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{RootConfiguration.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{MvcConfiguration.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

创建Mvc配置类：

```
@ComponentScan("book.manager.controller")
@Configuration
@EnableWebMvc
public class MvcConfiguration implements WebMvcConfigurer {

    //我们需要使用ThymeleafViewResolver作为视图解析器，并解析我们的HTML页面
    @Bean
```

```
    public ThymeleafViewResolver thymeleafViewResolver(@Autowired
SpringTemplateEngine springTemplateEngine){
        ThymeleafViewResolver resolver = new ThymeleafViewResolver();
        resolver.setOrder(1);
        resolver.setCharacterEncoding("UTF-8");
        resolver.setTemplateEngine(springTemplateEngine);
        return resolver;
    }

    //配置模板解析器
    @Bean
    public SpringResourceTemplateResolver templateResolver(){
        SpringResourceTemplateResolver resolver = new
SpringResourceTemplateResolver();
        resolver.setSuffix(".html");
        resolver.setPrefix("/WEB-INF/template/");
        return resolver;
    }

    //配置模板引擎Bean
    @Bean
    public SpringTemplateEngine springTemplateEngine(@Autowired
ITemplateResolver resolver){
        SpringTemplateEngine engine = new SpringTemplateEngine();
        engine.setTemplateResolver(resolver);
        return engine;
    }

    //开启静态资源处理
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }

    //静态资源路径配置
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/static/**").addResourceLocations("/WEB-
INF/static/");
    }
}
```

创建Root配置类：

```
@Configuration
public class RootConfiguration {

}
```

最后创建一个专用于响应页面的Controller即可：

```
/**
 * 专用于处理页面响应的控制器
 */
@Controller
public class PageController {

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

接着我们需要将前端页面放到对应的文件夹中，然后开启服务器并通过浏览器，成功访问。

接着我们需要配置SpringSecurity，与Mvc一样，需要一个初始化器：

```
public class SecurityInitializer extends
AbstractSecurityWebApplicationInitializer {
    //不用重写任何内容
    //这里实际上会自动注册一个Filter，SpringSecurity底层就是依靠N个过滤器实现的，我们之后再
探讨
}
```

接着我们需要再创建一个配置类用于配置SpringSecurity：

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
        //继承WebSecurityConfigurerAdapter，之后会进行配置
}
```

接着在根容器中添加此配置文件即可：

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[]{RootConfiguration.class, SecurityConfiguration.class};
}
```

这样，SpringSecurity的配置就完成了，我们再次运行项目，会发现无法进入的我们的页面中，无论我们访问哪个页面，都会进入到SpringSecurity为我们提供的一个默认登录页面，之后我们会讲解如何进行配置。

至此，项目环境搭建完成。

---

# 认证

## 直接认证

既然我们的图书管理系统要求用户登录之后才能使用，所以我们首先要做的就是实现用户验证，要实现用户验证，我们需要进行一些配置：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  //这里使用
SpringSecurity提供的BCryptPasswordEncoder
    auth
            .inMemoryAuthentication() //直接验证方式，之后会讲解使用数据库验证
            .passwordEncoder(encoder) //密码加密器
            .withUser("test")   //用户名
            .password(encoder.encode("123456"))   //这里需要填写加密后的密码
            .roles("user");   //用户的角色（之后讲解）
}
```

SpringSecurity的密码校验并不是直接使用原文进行比较，而是使用加密算法将密码进行加密（更准确地说应该进行Hash处理，此过程是不可逆的，无法解密），最后将用户提供的密码以同样的方式加密后与密文进行比较。对于我们来说，用户提供的密码属于隐私信息，直接明文存储并不好，而且如果数据库内容被窃取，那么所有用户的密码将全部泄露，这是我们不希望看到的结果，我们需要一种既能隐藏用户密码也能完成认证的机制，而Hash处理就是一种很好的解决方案，通过将用户的密码进行Hash值计算，计算出来的结果无法还原为原文，如果需要验证是否与此密码一致，那么需要以同样的方式加密再比较两个Hash值是否一致，这样就很好的保证了用户密码的安全性。

我们这里使用的是SpringSecurity提供的BCryptPasswordEncoder，至于加密过程，这里不做深入讲解。

现在，我们可以尝试使用此账号登录，在登录后，就可以随意访问我们的网站内容了。

## 使用数据库认证

前面我们已经实现了直接认证的方式，那么如何将其连接到数据库，通过查询数据库中的内容来进行用户登录呢?

首先我们需要将加密后的密码添加到数据库中作为用户密码:

```
public class MainTest {

    @Test
    public void test(){
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        System.out.println(encoder.encode("123456"));
    }
}
```

这里编写一个测试来完成。

然后我们需要创建一个Service实现，实现的是UserDetailsService，它支持我们自己返回一个UserDetails对象，我们只需直接返回一个包含数据库中的用户名、密码等信息的UserDetails即可，SpringSecurity会自动进行比对。

```
@Service
public class UserAuthService implements UserDetailsService {

    @Resource
    UserMapper mapper;
```

```
    @Override
    public UserDetails loadUserByUsername(String s) throws
UsernameNotFoundException {
        String password = mapper.getPasswordByUsername(s);  //从数据库根据用户名获取
密码
        if(password == null)
            throw new UsernameNotFoundException("登录失败，用户名或密码错误！");
        return User   //这里需要返回UserDetails，SpringSecurity会根据给定的信息进行比对
                .withUsername(s)
                .password(password)   //直接从数据库取的密码
                .roles("user")   //用户角色
                .build();
    }
}
```

别忘了在配置类中进行扫描，将其注册为Bean，接着我们需要编写一个Mapper用于和数据库交互：

```
@Mapper
public interface UserMapper {

    @Select("select password from users where username = #{username}")
    String getPasswordByUsername(String username);
}
```

和之前一样，配置一下Mybatis和数据源：

```
@ComponentScans({
        @ComponentScan("book.manager.service")
})
@MapperScan("book.manager.mapper")
@Configuration
public class RootConfiguration {
    @Bean
    public DataSource dataSource(){
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/study");
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword("123456");
        return dataSource;
    }

    @Bean
    public SqlSessionFactoryBean sqlSessionFactoryBean(@Autowired DataSource
dataSource){
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        return bean;
    }
}
```

最后再修改一下Security配置：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
      .userDetailsService(service)   //使用自定义的Service实现类进行验证
      .passwordEncoder(new BCryptPasswordEncoder());   //依然使用
BCryptPasswordEncoder
}
```

这样，登陆就会从数据库中进行查询。

## 自定义登录界面

前面我们已经了解了如何实现数据库权限验证，那么现在我们接着来看看，如何将登陆页面修改为我们自定义的样式。

首先我们要了解一下SpringSecurity是如何进行登陆验证的，我们可以观察一下默认的登陆界面中，表单内有哪些内容：

```
<div class="container">
      <form class="form-signin" method="post" action="/book_manager/login">
        <h2 class="form-signin-heading">Please sign in</h2>
        <p>
          <label for="username" class="sr-only">Username</label>
          <input type="text" id="username" name="username" class="form-control"
placeholder="Username" required="" autofocus="">
        </p>
        <p>
          <label for="password" class="sr-only">Password</label>
          <input type="password" id="password" name="password" class="form-
control" placeholder="Password" required="">
        </p>
<input name="_csrf" type="hidden" value="83421936-b84b-44e3-be47-58bb2c14571a">
        <button class="btn btn-lg btn-primary btn-block" type="submit">Sign
in</button>
      </form>
</div>
```

我们发现，首先有一个用户名的输入框和一个密码的输入框，我们需要在其中填写用户名和密码，但是我们发现，除了这两个输入框以外，还有一个input标签，它是隐藏的，并且它存储了一串类似于Hash值的东西，名称为"_csrf"，其实看名字就知道，这玩意八成都是为了防止CSRF攻击而存在的。

从Spring Security 4.0开始，默认情况下会启用CSRF保护，以防止CSRF攻击应用程序，Spring Security CSRF会针对PATCH，POST，PUT和DELETE方法的请求（不仅仅只是登陆请求，这里指的是任何请求路径）进行防护，而这里的登陆表单正好是一个POST类型的请求。在默认配置下，无论是否登陆，页面中只要发起了PATCH，POST，PUT和DELETE请求一定会被拒绝，并返回**403**错误**（注意，这里是个究极大坑）**，需要在请求的时候加入csrfToken才行，也就是"83421936-b84b-44e3-be47-58bb2c14571a"，正是csrfToken，如果提交的是表单类型的数据，那么表单中必须包含此Token字符串，键名称为"_csrf"；如果是JSON数据格式发送的，那么就需要在请求头中包含此Token字符串。

综上所述，我们最后提交的登陆表单，除了必须的用户名和密码，还包含了一个csrfToken字符串用于验证，防止攻击。

因此，我们在编写自己的登陆页面时，需要添加这样一个输入框：

```
<input type="text" th:name="${_csrf.getParameterName()}"
th:value="${_csrf.token}" hidden>
```

隐藏即可，但是必须要有，而Token的键名称和Token字符串可以通过Thymeleaf从Model中获取，SpringSecurity会自动将Token信息添加到Model中。

接着我们就可以将我们自己的页面替换掉默认的页面了，我们需要重写另一个方法来实现：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .authorizeRequests()    //首先需要配置哪些请求会被拦截，哪些请求必须具有什么角色才能访问
            .antMatchers("/static/**").permitAll()     //静态资源，使用permitAll来运行任何人访问（注意一定要放在前面）
            .antMatchers("/**").hasRole("user")      //所有请求必须登陆并且是user角色才可以访问（不包含上面的静态资源）
}
```

首先我们需要配置拦截规则，也就是当用户未登录时，哪些路径可以访问，哪些路径不可以访问，如果不可以访问，那么会被自动重定向到登陆页面。

接着我们需要配置表单登陆和登录页面：

```
.formLogin()        //配置Form表单登陆
.loginPage("/login")        //登陆页面地址（GET）
.loginProcessingUrl("/doLogin")     //form表单提交地址（POST）
.defaultSuccessUrl("/index")     //登陆成功后跳转的页面，也可以通过Handler实现高度自定义
.permitAll()     //登陆页面也需要允许所有人访问
```

需要配置登陆页面的地址和登陆请求发送的地址，这里登陆页面填写为`/login`，登陆请求地址为`/doLogin`，登陆页面需要我们自己去编写Controller来实现，登陆请求提交处理由SpringSecurity提供，只需要写路径就可以了。

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

配置好后，我们还需要配置一下退出登陆操作：

```
.and()
.logout()
.logoutUrl("/logout")     //退出登陆的请求地址
.logoutSuccessUrl("/login");     //退出后重定向的地址
```

注意这里的退出登陆请求也必须是POST请求方式（因为开启了CSFR防护，需要添加Token），否则无法访问，这里主页面就这样写：

```
<body>
    <form action="logout" method="post">
        <input type="text" th:name="${_csrf.getParameterName()}"
th:value="${_csrf.token}" hidden>
        <button>退出登陆</button>
    </form>
</body>
</html>
```

登陆成功后，点击退出登陆按钮，就可以成功退出并回到登陆界面了。

由于我们在学习的过程中暂时用不到CSFR防护，因此可以将其关闭，这样直接使用get请求也可以退出登陆，并且登陆请求中无需再携带Token了，推荐关闭，因为不关闭后面可能会因为没考虑CSRF防护而遇到一连串的问题：

```
.and()
.csrf().disable();
```

这样就可以直接关闭此功能了，但是注意，这样将会导致您的Web网站存在安全漏洞。（这里为了之后省事，就关闭保护了，但是一定要记得在不关闭的情况下需要携带Token访问）

---

# 授权

用户登录后，可能会根据用户当前是身份进行角色划分，比如我们最常用的QQ，一个QQ群里面，有群主、管理员和普通群成员三种角色，其中群主具有最高权限，群主可以管理整个群的任何板块，并且具有解散和升级群的资格，而管理员只有群主的一部分权限，只能用于日常管理，普通群成员则只能进行最基本的聊天操作。

对于我们来说，用户的一个操作实际上就是在访问我们提供的`接口`(编写的对应访问路径的Servlet），比如登陆，就需要调用`/login`接口，退出登陆就要调用/`logout`接口，而我们之前的图书管理系统中，新增图书、删除图书，所有的操作都有着对应的Servlet来进行处理。因此，从我们开发者的角度来说，决定用户能否使用某个功能，只需要决定用户是否能够访问对应的Servlet即可。

我们可以大致像下面这样进行划分：

- 群主：`/login`、`/logout`、`/chat`、`/edit`、`/delete`、`/upgrade`
- 管理员：`/login`、`/logout`、`/chat`、`/edit`
- 普通群成员：`/login`、`/logout`、`/chat`

也就是说，我们需要做的就是指定哪些请求可以由哪些用户发起。

SpringSecurity为我们提供了两种授权方式：

- 基于权限的授权：只要拥有某权限的用户，就可以访问某个路径
- 基于角色的授权：根据用户属于哪个角色来决定是否可以访问某个路径

两者只是概念上的不同，实际上使用起来效果差不多。这里我们就先演示以角色方式来进行授权。

## 基于角色的授权

现在我们希望创建两个角色，普通用户和管理员，普通用户只能访问index页面，而管理员可以访问任何页面。

首先我们需要对数据库中的角色表进行一些修改，添加一个用户角色字段，并创建一个新的用户，Test用户的角色为user，而Admin用户的角色为admin。

接着我们需要配置SpringSecurity，决定哪些角色可以访问哪些页面：

```
http
        .authorizeRequests()
        .antMatchers("/static/**").permitAll()
            .antMatchers("/index").hasAnyRole("user", "admin")   //index页面可以由user或admin访问
        .anyRequest().hasRole("admin")   //除了上面以外的所有内容，只能是admin访问
```

接着我们需要稍微修改一下验证逻辑，首先创建一个实体类用于表示数据库中的用户名、密码和角色：

```
@Data
public class AuthUser {
    String username;
    String password;
    String role;
}
```

接着修改一下Mapper：

```
@Mapper
public interface UserMapper {

    @Select("select * from users where username = #{username}")
    AuthUser getPasswordByUsername(String username);
}
```

最后再修改一下Service：

```
@Override
public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException
{
    AuthUser user = mapper.getPasswordByUsername(s);
    if(user == null)
        throw new UsernameNotFoundException("登录失败，用户名或密码错误！");
    return User
            .withUsername(user.getUsername())
            .password(user.getPassword())
            .roles(user.getRole())
            .build();
}
```

现在我们可以尝试登陆，接着访问一下`/index`和`/admin`两个页面。

## 基于权限的授权

基于权限的授权与角色类似，需要以`hasAnyAuthority`或`hasAuthority`进行判断：

```
.anyRequest().hasAnyAuthority("page:index")
```

```
@Override
public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException
{
    AuthUser user = mapper.getPasswordByUsername(s);
    if(user == null)
        throw new UsernameNotFoundException("登录失败，用户名或密码错误！");
    return User
            .withUsername(user.getUsername())
            .password(user.getPassword())
            .authorities("page:index")
            .build();
}
```

## 使用注解判断权限

除了直接配置以外，我们还可以以注解形式直接配置，首先需要在配置类（注意这里是在Mvc的配置类上添加，因为这里只针对Controller进行过滤，所有的Controller是由Mvc配置类进行注册的，如果需要为Service或其他Bean也启用权限判断，则需要在Security的配置类上添加）上开启：

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
```

接着我们可以直接在需要添加权限验证的请求映射上添加注解：

```
@PreAuthorize("hasRole('user')")   //判断是否为user角色，只有此角色才可以访问
@RequestMapping("/index")
public String index(){
    return "index";
}
```

通过添加`@PreAuthorize`注解，在执行之前判断判断权限，如果没有对应的权限或是对应的角色，将无法访问页面。

这里其实是使用了SpEL表达式，相当于可以执行一些逻辑再得到结果，而不是直接传值，官方文档地址：https://docs.spring.io/spring-framework/docs/5.2.13.RELEASE/spring-framework-reference/core.html#expressions，内容比较多，不是重点，这里就不再详细介绍了。

同样的还有`@PostAuthorize`注解，但是它是在方法执行之后再进行拦截：

```
@PostAuthorize("hasRole('user')")
@RequestMapping("/index")
public String index(){
    System.out.println("执行了");
    return "index";
}
```

除了Controller以外，只要是由Spring管理的Bean都可以使用注解形式来控制权限，只要不具备访问权限，那么就无法执行方法并且会返回403页面。

```
@Service
public class UserService {

    @PreAuthorize("hasAnyRole('user')")
    public void test(){
        System.out.println("成功执行");
    }
}
```

注意Service是由根容器进行注册，需要在Security配置类上添加`@EnableGlobalMethodSecurity`注解才可以生效。与具有相同功能的还有`@Secure`但是它不支持SpEL表达式的权限表示形式，并且需要添加"ROLE_"前缀，这里就不做演示了。

我们还可以使用`@PreFilter`和`@PostFilter`对集合类型的参数或返回值进行过滤。

比如：

```
@PreFilter("filterObject.equals('lbwnb')")   //filterObject代表集合中每个元素，只要满足条件的元素才会留下
public void test(List<String> list){
    System.out.println("成功执行"+list);
}
```

```
@RequestMapping("/index")
public String index(){
    List<String> list = new LinkedList<>();
    list.add("lbwnb");
    list.add("yyds");
    service.test(list);
    return "index";
}
```

与`@PreFilter`类似的`@PostFilter`这里就不做演示了，它用于处理返回值，使用方法是一样的。

当有多个集合时，需要使用`filterTarget`进行指定：

```
@PreFilter(value = "filterObject.equals('lbwnb')", filterTarget = "list2")
public void test(List<String> list, List<String> list2){
    System.out.println("成功执行"+list);
}
```

# 记住我

我们的网站还有一个重要的功能，就是记住我，也就是说我们可以在登陆之后的一段时间内，无需再次输入账号和密码进行登陆，相当于服务端已经记住当前用户，再次访问时就可以免登陆进入，这是一个非常常用的功能。

我们之前在JavaWeb阶段，使用本地Cookie存储的方式实现了记住我功能，但是这种方式并不安全，同时在代码编写上也比较麻烦，那么能否有一种更加高效的记住我功能实现呢？

SpringSecurity为我们提供了一种优秀的实现，它为每个已经登陆的浏览器分配一个携带Token的Cookie，并且此Cookie默认会被保留14天，只要我们不清理浏览器的Cookie，那么下次携带此Cookie访问服务器将无需登陆，直接继续使用之前登陆的身份，这样显然比我们之前的写法更加简便。并且我们需要进行简单配置，即可开启记住我功能：

```
.and()
.rememberMe()    //开启记住我功能
.rememberMeParameter("remember")  //登陆请求表单中需要携带的参数，如果携带，那么本次登陆会被记住
.tokenRepository(new InMemoryTokenRepositoryImpl())  //这里使用的是直接在内存中保存的TokenRepository实现
  //TokenRepository有很多种实现，InMemoryTokenRepositoryImpl直接基于Map实现的，缺点就是占内存、服务器重启后记住我功能将失效
  //后面我们还会讲解如何使用数据库来持久化保存Token信息
```

接着我们需要在前端修改一下记住我勾选框的名称，将名称修改与上面一致，如果上面没有配置名称，那么默认使用"remember-me"作为名称：

```
<input type="checkbox" name="remember" class="ad-checkbox">
```

现在我们启动服务器，在登陆时勾选记住我勾选框，观察Cookie的变化。

虽然现在已经可以实现记住我功能了，但是还有一定的缺陷，如果服务器重新启动（因为Token信息全部存在HashMap中，也就是存在内存中），那么所有记录的Token信息将全部丢失，这时即使浏览器携带了之前的Token也无法恢复之前登陆的身份。

我们可以将Token信息记录全部存放到数据库中（学习了Redis之后还可以放到Redis服务器中）利用数据库的持久化存储机制，即使服务器重新启动，所有的Token信息也不会丢失，配置数据库存储也很简单：

```
@Resource
PersistentTokenRepository repository;

@Bean
public PersistentTokenRepository jdbcRepository(@Autowired DataSource dataSource){
    JdbcTokenRepositoryImpl repository = new JdbcTokenRepositoryImpl();  //使用基于JDBC的实现
    repository.setDataSource(dataSource);   //配置数据源
    repository.setCreateTableOnStartup(true);   //启动时自动创建用于存储Token的表（建议第一次启动之后删除该行）
    return repository;
}
```

```
.and()
.rememberMe()
.rememberMeParameter("remember")
.tokenRepository(repository)
.tokenValiditySeconds(60 * 60 * 24 * 7)  //Token的有效时间（秒）默认为14天
```

稍后服务器启动我们可以观察一下数据库，如果出现名为`persistent_logins`的表，那么证明配置没有问题。

当我们登陆并勾选了记住我之后，那么数据库中会新增一条Token记录。

---

# SecurityContext

用户登录之后，怎么获取当前已经登录用户的信息呢？通过使用SecurityContextHolder就可以很方便地得到SecurityContext对象了，我们可以直接使用SecurityContext对象来获取当前的认证信息：

```
@RequestMapping("/index")
    public String index(){
        SecurityContext context = SecurityContextHolder.getContext();
        Authentication authentication = context.getAuthentication();
        User user = (User) authentication.getPrincipal();
        System.out.println(user.getUsername());
        System.out.println(user.getAuthorities());
        return "index";
    }
```

通过SecurityContext我们就可以快速获取当前用户的名称和授权信息等。

除了这种方式以外，我们还可以直接从Session中获取：

```
@RequestMapping("/index")
public String index(@SessionAttribute("SPRING_SECURITY_CONTEXT") SecurityContext
context){
    Authentication authentication = context.getAuthentication();
    User user = (User) authentication.getPrincipal();
    System.out.println(user.getUsername());
    System.out.println(user.getAuthorities());
    return "index";
}
```

注意SecurityContextHolder是有一定的存储策略的，SecurityContextHolder中的SecurityContext对象会在一开始请求到来时被设定，至于存储方式其实是由存储策略决定的，如果我们这样编写，那么在默认情况下是无法获取到认证信息的：

```
@RequestMapping("/index")
public String index(){
    new Thread(() -> {   //创建一个子线程去获取
        SecurityContext context = SecurityContextHolder.getContext();
        Authentication authentication = context.getAuthentication();
        User user = (User) authentication.getPrincipal();   //NPE
        System.out.println(user.getUsername());
        System.out.println(user.getAuthorities());
    });
    return "index";
}
```

这是因为SecurityContextHolder的存储策略默认是`MODE_THREADLOCAL`，它是基于ThreadLocal实现的，`getContext()`方法本质上调用的是对应的存储策略实现的方法：

```
public static SecurityContext getContext() {
    return strategy.getContext();
}
```

SecurityContextHolderStrategy有三个实现类：

- GlobalSecurityContextHolderStrategy：全局模式，不常用
- ThreadLocalSecurityContextHolderStrategy：基于ThreadLocal实现，线程内可见
- InheritableThreadLocalSecurityContextHolderStrategy：基于InheritableThreadLocal实现，线程和子线程可见

因此，如果上述情况需要在子线程中获取，那么需要修改SecurityContextHolder的存储策略，在初始化的时候设置：

```
@PostConstruct
public void init(){

 SecurityContextHolder.setStrategyName(SecurityContextHolder.MODE_INHERITABLETHR
EADLOCAL);
}
```

这样在子线程中也可以获取认证信息了。

因为用户的验证信息是基于SecurityContext进行判断的，我们可以直接修改SecurityContext的内容，来手动为用户进行登陆：

```
@RequestMapping("/auth")
@ResponseBody
public String auth(){
    SecurityContext context = SecurityContextHolder.getContext();  //获取
SecurityContext对象（当前会话肯定是没有登陆的）
    UsernamePasswordAuthenticationToken token = new
UsernamePasswordAuthenticationToken("Test", null,
            AuthorityUtils.commaSeparatedStringToAuthorityList("ROLE_user"));
 //手动创建一个UsernamePasswordAuthenticationToken对象，也就是用户的认证信息，角色需要添
加ROLE_前缀，权限直接写
    context.setAuthentication(token);  //手动为SecurityContext设定认证信息
    return "Login success! ";
}
```

在未登陆的情况下，访问此地址将直接进行手动登陆，再次访问`/index`页面，可以直接访问，说明手动设置认证信息成功。

**疑惑：**SecurityContext这玩意不是默认线程独占吗，那每次请求都是一个新的线程，按理说上一次的SecurityContext对象应该没了才对啊，为什么再次请求依然能够继续使用上一次SecurityContext中的认证信息呢?

SecurityContext的生命周期：请求到来时从Session中取出，放入SecurityContextHolder中，请求结束时从SecurityContextHolder取出，并放到Session中，实际上就是依靠Session来存储的，一旦会话过期验证信息也跟着消失。

---

# SpringSecurity原理

**注意：**本小节内容作为选学内容，但是难度比前两章的源码部分简单得多。

最后我们再来聊一下SpringSecurity的实现原理，它本质上是依靠N个Filter实现的，也就是一个完整的过滤链（注意这里是过滤器，不是拦截器）

我们就从`AbstractSecurityWebApplicationInitializer`开始下手，我们来看看它配置了什么：

```
//此方法会在启动时被调用
public final void onStartup(ServletContext servletContext) {
    this.beforeSpringSecurityFilterChain(servletContext);
    if (this.configurationClasses != null) {
        AnnotationConfigWebApplicationContext rootAppContext = new
AnnotationConfigWebApplicationContext();
        rootAppContext.register(this.configurationClasses);
        servletContext.addListener(new ContextLoaderListener(rootAppContext));
    }

    if (this.enableHttpSessionEventPublisher()) {

 servletContext.addListener("org.springframework.security.web.session.HttpSessio
nEventPublisher");
    }

    servletContext.setSessionTrackingModes(this.getSessionTrackingModes());
    //重点在这里，这里插入了关键的FilterChain
    this.insertSpringSecurityFilterChain(servletContext);
    this.afterSpringSecurityFilterChain(servletContext);
}
```

```
private void insertSpringSecurityFilterChain(ServletContext servletContext) {
    String filterName = "springSecurityFilterChain";
    //创建了一个DelegatingFilterProxy对象，它本质上也是一个Filter
    DelegatingFilterProxy springSecurityFilterChain = new
DelegatingFilterProxy(filterName);
    String contextAttribute = this.getWebApplicationContextAttribute();
    if (contextAttribute != null) {
        springSecurityFilterChain.setContextAttribute(contextAttribute);
    }
        //通过ServletContext注册DelegatingFilterProxy这个Filter
    this.registerFilter(servletContext, true, filterName,
springSecurityFilterChain);
}
```

我们接着来看看，`DelegatingFilterProxy`在做什么：

```
//这个是初始化方法，它由GenericFilterBean（父类）定义，在afterPropertiesSet方法中被调用
protected void initFilterBean() throws ServletException {
    synchronized(this.delegateMonitor) {
        if (this.delegate == null) {
            if (this.targetBeanName == null) {
                this.targetBeanName = this.getFilterName();
            }

            WebApplicationContext wac = this.findWebApplicationContext();
```

```
            if (wac != null) {
                //耐心点，套娃很正常
                this.delegate = this.initDelegate(wac);
            }
        }

    }
}
```

```
protected Filter initDelegate(WebApplicationContext wac) throws ServletException
{
    String targetBeanName = this.getTargetBeanName();
    Assert.state(targetBeanName != null, "No target bean name set");
    //这里通过WebApplicationContext获取了一个Bean
    Filter delegate = (Filter)wac.getBean(targetBeanName, Filter.class);
    if (this.isTargetFilterLifecycle()) {
        delegate.init(this.getFilterConfig());
    }

    //返回Filter
    return delegate;
}
```

这里我们需要添加一个断点来查看到底获取到了什么Bean。

通过断点调试，我们发现这里放回的对象是一个FilterChainProxy类型的，并且调用了它的初始化方法，但是FilterChainProxy类中并没有重写`init`方法或是`initFilterBean`方法。

我们倒回去看，当Filter返回之后，`DelegatingFilterProxy`的一个成员变量`delegate`被赋值为得到的Filter，也就是FilterChainProxy对象，接着我们来看看，`DelegatingFilterProxy`是如何执行doFilter方法的。

```
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain filterChain) throws ServletException, IOException {
    Filter delegateToUse = this.delegate;
    if (delegateToUse == null) {
        //非正常情况，这里省略...
    }
        //这里才是真正的调用，别忘了delegateToUse就是初始化的FilterChainProxy对象
    this.invokeDelegate(delegateToUse, request, response, filterChain);
}
```

```
protected void invokeDelegate(Filter delegate, ServletRequest request,
ServletResponse response, FilterChain filterChain) throws ServletException,
IOException {
  //最后实际上调用的是FilterChainProxy的doFilter方法
    delegate.doFilter(request, response, filterChain);
}
```

所以我们接着来看，`FilterChainProxy`的doFilter方法又在干什么：

```
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
    boolean clearContext = request.getAttribute(FILTER_APPLIED) == null;
    if (!clearContext) {
        //真正的过滤在这里执行
        this.doFilterInternal(request, response, chain);
    } else {
        //...
    }
}
```

```
private void doFilterInternal(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
    FirewalledRequest firewallRequest =
this.firewall.getFirewalledRequest((HttpServletRequest)request);
    HttpServletResponse firewallResponse =
this.firewall.getFirewalledResponse((HttpServletResponse)response);
    //这里获取了一个Filter列表，实际上SpringSecurity就是由N个过滤器实现的，这里获取的都是
SpringSecurity提供的过滤器
    //但是请注意，经过我们之前的分析，实际上真正注册的Filter只有DelegatingFilterProxy
    //而这里的Filter列表中的所有Filter并没有被注册，而是在这里进行内部调用
    List<Filter> filters = this.getFilters((HttpServletRequest)firewallRequest);
    //只要Filter列表不是空，就依次执行内置的Filter
    if (filters != null && filters.size() != 0) {
        if (logger.isDebugEnabled()) {
            logger.debug(LogMessage.of(() -> {
                return "Securing " + requestLine(firewallRequest);
            }));
        }
                //这里创建一个虚拟的过滤链，过滤流程是由SpringSecurity自己实现的
        FilterChainProxy.VirtualFilterChain virtualFilterChain = new
FilterChainProxy.VirtualFilterChain(firewallRequest, chain, filters);
        //调用虚拟过滤链的doFilter
        virtualFilterChain.doFilter(firewallRequest, firewallResponse);
    } else {
        if (logger.isTraceEnabled()) {
            logger.trace(LogMessage.of(() -> {
                return "No security for " + requestLine(firewallRequest);
            }));
        }

        firewallRequest.reset();
        chain.doFilter(firewallRequest, firewallResponse);
    }
}
```

我们来看一下虚拟过滤链的doFilter是怎么处理的：

```
//看似没有任何循环，实际上就是一个循环，是一个递归调用
public void doFilter(ServletRequest request, ServletResponse response) throws
IOException, ServletException {
    //判断是否已经通过全部的内置过滤器，定位是否等于当前大小
    if (this.currentPosition == this.size) {
        if (FilterChainProxy.logger.isDebugEnabled()) {
```

```
            FilterChainProxy.logger.debug(LogMessage.of(() -> {
                return "Secured " +
FilterChainProxy.requestLine(this.firewalledRequest);
            }));
        }

        this.firewalledRequest.reset();
        //所有的内置过滤器已经完成，按照正常流程走DelegatingFilterProxy的下一个Filter
        //也就是说这里之后就与DelegatingFilterProxy没有任何关系了，该走其他过滤器就走其他地
方配置的过滤器，SpringSecurity的过滤操作已经结束
        this.originalChain.doFilter(request, response);
    } else {
        //定位自增
        ++this.currentPosition;
        //获取当前定位的Filter
        Filter nextFilter =
(Filter)this.additionalFilters.get(this.currentPosition - 1);
        if (FilterChainProxy.logger.isTraceEnabled()) {
            FilterChainProxy.logger.trace(LogMessage.format("Invoking %s
(%d/%d)", nextFilter.getClass().getSimpleName(), this.currentPosition,
this.size));
        }
                //执行内部过滤器的doFilter方法，传入当前对象本身作为Filter，执行如果成功，
那么一定会再次调用当前对象的doFilter方法
        //可能最不理解的就是这里，执行的难道不是内部其他Filter的doFilter方法吗，怎么会让当前
对象的doFilter方法递归调用呢？
        //没关系，了解了其中一个内部过滤器就明白了
        nextFilter.doFilter(request, response, this);
    }
}
```

因此，我们差不多已经了解了整个SpringSecurity的实现机制了，那么我们来看几个内部的过滤器分别在做什么。

比如用于处理登陆的过滤器`UsernamePasswordAuthenticationFilter`，它继承自`AbstractAuthenticationProcessingFilter`，我们来看看它是怎么进行过滤的：

```
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
    this.doFilter((HttpServletRequest)request, (HttpServletResponse)response,
chain);
}

private void doFilter(HttpServletRequest request, HttpServletResponse response,
FilterChain chain) throws IOException, ServletException {
    //如果不是登陆请求，那么根本不会理这个请求
    if (!this.requiresAuthentication(request, response)) {
        //直接调用传入的FilterChain的doFilter方法
        //而这里传入的正好是VirtualFilterChain对象
        //这下知道为什么上面说是递归了吧
        chain.doFilter(request, response);
    } else {
        //如果是登陆请求，那么会执行登陆请求的相关逻辑，注意执行过程中出现任何问题都会抛出异常
        //比如用户名和密码错误，我们之前也已经测试过了，会得到一个BadCredentialsException
        try {
```

```
            //进行认证
            Authentication authenticationResult =
this.attemptAuthentication(request, response);
            if (authenticationResult == null) {
                return;
            }

            this.sessionStrategy.onAuthentication(authenticationResult, request,
response);
            if (this.continueChainBeforeSuccessfulAuthentication) {
                chain.doFilter(request, response);
            }

            //如果一路绿灯，没有报错，那么验证成功，执行successfulAuthentication
            this.successfulAuthentication(request, response, chain,
authenticationResult);
        } catch (InternalAuthenticationServiceException var5) {
            this.logger.error("An internal error occurred while trying to
authenticate the user.", var5);
            //验证失败，会执行unsuccessfulAuthentication
            this.unsuccessfulAuthentication(request, response, var5);
        } catch (AuthenticationException var6) {
            this.unsuccessfulAuthentication(request, response, var6);
        }

    }
}
```

那么我们来看看successfulAuthentication和unsuccessfulAuthentication分别做了什么：

```
protected void successfulAuthentication(HttpServletRequest request,
HttpServletResponse response, FilterChain chain, Authentication authResult)
throws IOException, ServletException {
    //向SecurityContextHolder添加认证信息，我们可以通过SecurityContextHolder对象获取当
前登陆的用户
    SecurityContextHolder.getContext().setAuthentication(authResult);
    if (this.logger.isDebugEnabled()) {
        this.logger.debug(LogMessage.format("Set SecurityContextHolder to %s",
authResult));
    }

    //记住我实现
    this.rememberMeServices.loginSuccess(request, response, authResult);
    if (this.eventPublisher != null) {
        this.eventPublisher.publishEvent(new
InteractiveAuthenticationSuccessEvent(authResult, this.getClass()));
    }

    //调用默认的或是我们自己定义的AuthenticationSuccessHandler的
onAuthenticationSuccess方法
    //这个根据我们配置文件决定
    //到这里其实页面就已经直接跳转了
    this.successHandler.onAuthenticationSuccess(request, response, authResult);
}
```

```
protected void unsuccessfulAuthentication(HttpServletRequest request,
HttpServletResponse response, AuthenticationException failed) throws
IOException, ServletException {
    //登陆失败会直接清理掉SecurityContextHolder中的认证信息
    SecurityContextHolder.clearContext();
    this.logger.trace("Failed to process authentication request", failed);
    this.logger.trace("Cleared SecurityContextHolder");
    this.logger.trace("Handling authentication failure");
    //登陆失败的记住我处理
    this.rememberMeServices.loginFail(request, response);
    //同上，调用默认或是我们自己定义的AuthenticationFailureHandler
    this.failureHandler.onAuthenticationFailure(request, response, failed);
}
```

了解了整个用户验证实现流程，其实其它的过滤器是如何实现的也就很容易联想到了，SpringSecurity的过滤器从某种意义上来说，更像是一个处理业务的Servlet，它做的事情不像是拦截，更像是完成自己对应的职责，只不过是使用了过滤器机制进行实现罢了。

SecurityContextPersistenceFilter也是内置的Filter，可以尝试阅读一下其源码，了解整个SecurityContextHolder的运作原理，这里先说一下大致流程，各位可以依照整个流程按照源码进行推导：

当过滤器链执行到SecurityContextPersistenceFilter时，它会从HttpSession中把SecurityContext对象取出来（是存在Session中的，跟随会话的消失而消失），然后放入SecurityContextHolder对象中。请求结束后，再把SecurityContext存入HttpSession中，并清除SecurityContextHolder内的SecurityContext对象。

---

# 完善功能

在了解了SpringSecurity的大部分功能后，我们就来将整个网站的内容进行完善，登陆目前已经实现了，我们还需要实现以下功能：

- 注册功能（仅针对于学生）
- 角色分为同学和管理员
    - 管理员负责上架、删除、更新书籍，查看所有同学的借阅列表
    - 同学可以借阅和归还书籍，以及查看自己的借阅列表

开始之前我们需要先配置一下Thymeleaf的SpringSecurity扩展，它针对SpringSecurity提供了更多额外的解析：

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
    <version>3.0.4.RELEASE</version>
</dependency>
```

```
//配置模板引擎Bean
@Bean
public SpringTemplateEngine springTemplateEngine(@Autowired ITemplateResolver
resolver){
    SpringTemplateEngine engine = new SpringTemplateEngine();
    engine.setTemplateResolver(resolver);
    engine.addDialect(new SpringSecurityDialect());   //添加针对于SpringSecurity的
方言
    return engine;
}
```

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

下一章就是最后一章了，我们会深入讲解MySQL的高级部分，包括函数、存储过程、锁机制、索引以及存储引擎。