

# 深入浅出 Spring Security

王松 著

清华大学出版社  
北京

## 内 容 简 介

Spring Security 是 Java 企业级开发中常用的安全管理框架，也能完美支持 OAuth2。同时，Spring Security 作为 Spring 家族的一员，与 Spring Boot、Spring Cloud 等框架整合使用也非常方便。

本书分为 15 章，讲解 Spring Security 框架、认证、认证流程分析、过滤器链分析、密码加密、RememberMe、会话管理、HttpFirewall、漏洞保护、HTTP 认证、跨域问题、异常处理、权限管理、权限模型、OAuth2 等内容。本书致力于让读者在学会 Spring Security 用法的同时，也能通过阅读源码来理解它的实现原理。

本书适合具有 Spring Boot 基础的读者、Java 企业应用开发工程师，也适合作为高等院校和培训机构计算机相关专业师生的教学参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。举报：010-62782989，beiqinquan@tup.tsinghua.edu.cn。

### 图书在版编目（CIP）数据

深入浅出 Spring Security / 王松著. —北京：清华大学出版社，2021.1  
ISBN 978-7-302-57276-3

I. ①深… II. ①王… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字（2021）第 005021 号

责任编辑：夏毓彦

封面设计：王 翔

责任校对：闫秀华

责任印制：丛怀宇

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>，<http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：三河市铭诚印务有限公司

经 销：全国新华书店

开 本：190mm×260mm

印 张：26.25

字 数：672 千字

版 次：2021 年 3 月第 1 版

印 次：2021 年 3 月第 1 次印刷

定 价：99.00 元

---

产品编号：089794-01

# 前言

安全管理是 Java 应用开发中无法避免的问题，目前主流的安全管理框架就是 Spring Security 和 Shiro，其中 Shiro 一直以使用简单和轻量级著称。然而，随着 Spring Boot 和微服务的流行，Spring Security 受到越来越多开发者的重视，因为 Spring Security 在和 Spring Boot 整合时具有先天优势。

目前市面上缺少系统介绍 Spring Security 的书籍，网上的博客内容又比较零散，这为很多初次接触 Spring Security 的 Java 工程师学习这门技术带来诸多不便。

笔者最早于个人博客上连载 Spring Security 系列教程，连载期间有不少读者加笔者微信讨论 Spring Security 的相关技术点，让笔者感受到读者对 Spring Security 的热情，也因此萌生了写一本技术图书来系统介绍 Spring Security 的想法。在朋友和家人的鼓励之下，这一想法逐步付诸实践，最终完成大家现在看到的这本《深入浅出 Spring Security》。

本书以 Spring Security 5.3.4 为基础，详细介绍 Spring Security 的基本用法以及相关原理。得益于 Spring Boot 中的自动化配置，Spring Security 上手非常容易，然而这种自动化配置，也让很多初次接触 Spring Security 的开发者“知其然，而不知其所以然”，仅限于会用，一旦出了漏洞，或者想要定制功能时，就会不知所措。因此，在写作本书过程中，除了基本功能的 Demo 演示外，还对 Spring Security 的相关源码做了深入分析，以便读者“知其然，更知其所以然”。

学习 Spring Security 不仅仅是学习安全管理框架，也是一个学习各种网络攻击与防御策略的过程，Spring Security 对很多常见网络攻击，如计时攻击、CSRF、XSS 等，都提供了相应的防御策略，因此，我们在学习 Spring Security 时，也可以顺便研究一下这些常见的网络攻击，以便设计出更加安全健壮的权限管理系统。

本书分为四部分：

第一部分：第 1 章，这一部分总体介绍 Spring Security 架构，方便读者从整体上把握 Spring Security 的功能。

第二部分：第 2~12 章，这一部分主要介绍 Spring Security 中的认证功能，以及由此衍生出来的会话管理、HTTP 防火墙、跨域管理等。

第三部分：第 13~14 章，这一部分主要介绍 Spring Security 中的授权功能，以及常见的权限模型 ACL 和 RBAC。

第四部分：第 15 章，这一部分主要介绍 OAuth2 协议在 Spring Security 框架中的落地。

## 示例代码约定

为了减少代码冗余和本书篇幅, 书中的所有示例代码片段都省略了 `package` 和 `import` 部分, 像下面这样:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login.html")
            .loginProcessingUrl("/doLogin")
            .defaultSuccessUrl("/index ")
            .failureUrl("/login.html")
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .permitAll()
            .and()
            .csrf().disable();
    }
}
```

有时候为了向读者演示代码的运行效果, 一个案例可能会被反复修改多次, 那么在后面展示代码时, 将不再列出不变的部分, 仅仅列出发生变化的代码片段, 像下面这样:

```
@Autowired
TokenStore tokenStore;
@Autowired
JwtAccessTokenConverter jwtAccessTokenConverter;
@Bean
AuthorizationServerTokenServices tokenServices() {
    DefaultTokenServices services = new DefaultTokenServices();
    services.setClientDetailsService(clientDetailsService);
    services.setSupportRefreshToken(true);
    services.setTokenStore(tokenStore);
    TokenEnhancerChain tokenEnhancerChain = new TokenEnhancerChain();
    tokenEnhancerChain
        .setTokenEnhancers(Arrays.asList(jwtAccessTokenConverter));
    services.setTokenEnhancer(tokenEnhancerChain);
    return services;
}
//省略其他
```

正常情况下, 这样的代码片段并不会影响大家理解本书内容。如果读者想要看到完整的

代码片段，可以下载本书提供的示例代码进行对照理解。

## 源码省略约定

在分析 Spring Security 源码时，为了简化源码和篇幅以便于读者理解，源码中的日志输出、注释以及一些无关紧要的代码会被移除掉，像下面这样：

```
@ConfigurationProperties(prefix = "spring.security")
public class SecurityProperties {
    private User user = new User();
    public User getUser() {
        return this.user;
    }
    public static class User {
        private String name = "user";
        private String password = UUID.randomUUID().toString();
        private List<String> roles = new ArrayList<>();
        //省略 getter/setter
    }
}
```

如果读者觉得这样阅读“不过瘾”，也可以下载 Spring Security 源码对照理解。

## 读者定位

阅读本书需要有一定的 Spring Boot 基础，对于无 Spring Boot 基础的读者，可以先学习 Spring Boot 然后再来阅读本书。学习 Spring Boot，可以参考笔者编写的图书《Spring Boot+Vue 全栈开发实战》或者笔者的教程：<http://springboot.javaboy.org>。

## 源码获取

本书所有的示例代码均存放在 GitHub 上，地址如下：

<https://github.com/lenve/spring-security-book-samples>

所有工程均为标准的 Maven 工程，可以用 IntelliJ IDEA 或者 Eclipse 打开。

## 纠错与勘误

如果读者在阅读本书时发现错误，可以将错误提交到 <https://github.com/lenve/spring-security-book-samples/issues>，笔者将错误内容汇总后同步发布在 <http://www.javaboy.org/spring-security-book> 以及微信公众号“江南一点雨”。修正后的内容将在后续重印的书中得到体现。

## 交流社区

学无止境，笔者将继续对 Spring Security 的发展保持关注。关于 Spring Security 的最新变化，笔者都将发布在微信公众号“江南一点雨”上，读者关注微信公众号后，也可以进入本书微信交流群进行交流。

王松

2021 年 1 月

# 目 录

第 1 章 Spring Security 架构概览 .....	1
1.1 Spring Security 简介 .....	1
1.2 Spring Security 核心功能 .....	2
1.2.1 认证 .....	3
1.2.2 授权 .....	3
1.2.3 其他 .....	3
1.3 Spring Security 整体架构 .....	4
1.3.1 认证和授权 .....	4
1.3.2 Web 安全 .....	6
1.3.3 登录数据保存 .....	9
1.4 小结 .....	9
第 2 章 Spring Security 认证 .....	10
2.1 Spring Security 基本认证 .....	10
2.1.1 快速入门 .....	10
2.1.2 流程分析 .....	11
2.1.3 原理分析 .....	12
2.2 登录表单配置 .....	19
2.2.1 快速入门 .....	19
2.2.2 配置细节 .....	23
2.3 登录用户数据获取 .....	39
2.3.1 从 SecurityContextHolder 中获取 .....	41
2.3.2 从当前请求对象中获取 .....	59
2.4 用户定义 .....	64
2.4.1 基于内存 .....	64
2.4.2 基于 JdbcUserDetailsManager .....	65

2.4.3 基于 MyBatis.....	68
2.4.4 基于 Spring Data JPA.....	74
2.5 小结.....	77
<b>第 3 章 认证流程分析.....</b>	<b>78</b>
3.1 登录流程分析.....	78
3.1.1 AuthenticationManager .....	78
3.1.2 AuthenticationProvider .....	79
3.1.3 ProviderManager .....	86
3.1.4 AbstractAuthenticationProcessingFilter .....	89
3.2 配置多个数据源.....	94
3.3 添加登录验证码.....	95
3.4 小结.....	99
<b>第 4 章 过滤器链分析.....</b>	<b>100</b>
4.1 初始化流程分析.....	100
4.1.1 ObjectPostProcessor .....	101
4.1.2 SecurityFilterChain.....	102
4.1.3 SecurityBuilder.....	103
4.1.4 FilterChainProxy.....	117
4.1.5 SecurityConfigurer.....	120
4.1.6 初始化流程分析 .....	128
4.2 ObjectPostProcessor 使用.....	136
4.3 多种用户定义方式.....	137
4.4 定义多个过滤器链.....	141
4.5 静态资源过滤.....	144
4.6 使用 JSON 格式登录 .....	146
4.7 添加登录验证码.....	150
4.8 小结.....	152
<b>第 5 章 密码加密 .....</b>	<b>153</b>
5.1 密码为什么要加密.....	153
5.2 密码加密方案进化史.....	154



---

5.3 PasswordEncoder 详解 .....	154
5.3.1 PasswordEncoder 常见实现类 .....	155
5.3.2 DelegatingPasswordEncoder .....	156
5.4 实战 .....	159
5.5 加密方案自动升级 .....	161
5.6 是谁的 PasswordEncoder .....	166
5.7 小结 .....	168
 第 6 章 RememberMe .....	 169
6.1 RememberMe 简介 .....	169
6.2 RememberMe 基本用法 .....	170
6.3 持久化令牌 .....	172
6.4 二次校验 .....	174
6.5 原理分析 .....	176
6.6 小结 .....	189
 第 7 章 会话管理 .....	 190
7.1 会话简介 .....	190
7.2 会话并发管理 .....	191
7.2.1 实战 .....	191
7.2.2 原理分析 .....	194
7.3 会话固定攻击与防御 .....	206
7.3.1 什么是会话固定攻击 .....	206
7.3.2 会话固定攻击防御策略 .....	207
7.4 Session 共享 .....	208
7.4.1 集群会话方案 .....	208
7.4.2 实战 .....	210
7.5 小结 .....	212
 第 8 章 HttpFirewall .....	 213
8.1 HttpFirewall 简介 .....	213
8.2 HttpFirewall 严格模式 .....	215
8.2.1 rejectForbiddenHttpMethod .....	216

8.2.2	rejectedBlacklistedUrls .....	217
8.2.3	rejectedUntrustedHosts .....	218
8.2.4	isNormalized .....	219
8.2.5	containsOnlyPrintableAsciiCharacters .....	220
8.3	HttpFirewall 普通模式 .....	220
8.4	小结 .....	221
第 9 章	漏洞保护 .....	222
9.1	CSRF 攻击与防御 .....	222
9.1.1	CSRF 简介 .....	222
9.1.2	CSRF 攻击演示 .....	223
9.1.3	CSRF 防御 .....	224
9.1.4	源码分析 .....	231
9.2	HTTP 响应头处理 .....	237
9.2.1	缓存控制 .....	239
9.2.2	X-Content-Type-Options .....	240
9.2.3	Strict-Transport-Security .....	241
9.2.4	X-Frame-Options .....	244
9.2.5	X-XSS-Protection .....	245
9.2.6	Content-Security-Policy .....	246
9.2.7	Referrer-Policy .....	248
9.2.8	Feature-Policy .....	249
9.2.9	Clear-Site-Data .....	249
9.3	HTTP 通信安全 .....	250
9.3.1	使用 HTTPS .....	250
9.3.2	代理服务器配置 .....	253
9.4	小结 .....	254
第 10 章	HTTP 认证 .....	255
10.1	HTTP Basic authentication .....	255
10.1.1	简介 .....	255
10.1.2	具体用法 .....	257
10.1.3	源码分析 .....	257
10.2	HTTP Digest authentication .....	260

---

10.2.1	简介 .....	260
10.2.2	具体用法 .....	261
10.2.3	源码分析 .....	263
10.3	小结.....	268
第 11 章	跨域问题 .....	269
11.1	什么是 CORS .....	269
11.2	Spring 处理方案 .....	270
11.2.1	@CrossOrigin .....	271
11.2.2	addCorsMappings .....	272
11.2.3	CorsFilter .....	273
11.3	Spring Security 处理方案 .....	274
11.3.1	特殊处理 OPTIONS 请求 .....	275
11.3.2	继续使用 CorsFilter .....	275
11.3.3	专业解决方案 .....	276
11.4	小结.....	279
第 12 章	异常处理 .....	280
12.1	Spring Security 异常体系 .....	280
12.2	ExceptionHandlerFilter 原理分析 .....	281
12.3	自定义异常配置 .....	287
12.4	小结.....	290
第 13 章	权限管理 .....	291
13.1	什么是权限管理 .....	291
13.2	Spring Security 权限管理策略 .....	292
13.3	核心概念 .....	292
13.3.1	角色与权限 .....	292
13.3.2	角色继承 .....	294
13.3.3	两种处理器 .....	295
13.3.4	前置处理器 .....	296
13.3.5	后置处理器 .....	299
13.3.6	权限元数据 .....	300

13.3.7 权限表达式 .....	303
13.4 基于 URL 地址的权限管理 .....	305
13.4.1 基本用法 .....	306
13.4.2 角色继承 .....	308
13.4.3 自定义表达式 .....	309
13.4.4 原理剖析 .....	310
13.4.5 动态管理权限规则 .....	316
13.5 基于方法的权限管理 .....	325
13.5.1 注解介绍 .....	325
13.5.2 基本用法 .....	326
13.5.3 原理剖析 .....	331
13.6 小结 .....	338
<b>第 14 章 权限模型 .....</b>	<b>339</b>
14.1 常见的权限模型 .....	339
14.2 ACL .....	340
14.2.1 ACL 权限模型介绍 .....	340
14.2.2 ACL 核心概念介绍 .....	341
14.2.3 ACL 数据库分析 .....	343
14.2.4 实战 .....	345
14.3 RBAC .....	354
14.3.1 RBAC 权限模型介绍 .....	354
14.3.2 RBAC 权限模型分类 .....	355
14.3.3 RBAC 小结 .....	357
14.4 小结 .....	357
<b>第 15 章 OAuth2 .....</b>	<b>358</b>
15.1 OAuth2 简介 .....	358
15.2 OAuth2 四种授权模式 .....	359
15.2.1 授权码模式 .....	360
15.2.2 简化模式 .....	361
15.2.3 密码模式 .....	363
15.2.4 客户端模式 .....	363
15.3 Spring Security OAuth2 .....	364

---

15.4	GitHub 授权登录.....	365
15.4.1	准备工作 .....	365
15.4.2	项目开发 .....	367
15.4.3	测试 .....	368
15.4.4	原理分析 .....	369
15.4.5	自定义配置 .....	375
15.5	授权服务器与资源服务器.....	379
15.5.1	项目规划 .....	379
15.5.2	项目搭建 .....	380
15.5.3	测试 .....	391
15.5.4	原理分析 .....	393
15.5.5	自定义请求 .....	396
15.6	使用 Redis.....	397
15.7	客户端信息存入数据库.....	399
15.8	使用 JWT.....	401
15.8.1	JWT.....	401
15.8.2	JWT 数据格式.....	402
15.8.3	OAuth2 中使用 JWT.....	403
15.9	小结.....	406



# 第 1 章

---

## Spring Security 架构概览

Spring Security 虽然历史悠久，但是从来没有像今天这样受到开发者这么多的关注。究其原因，还是沾了微服务的光。作为 Spring 家族中的一员，在和 Spring 家族中的其他产品如 Spring Boot、Spring Cloud 等进行整合时，Spring Security 拥有众多同类型框架无可比拟的优势。本章我们就先从整体上了解一下 Spring Security 及其工作原理。

本章涉及的主要知识点有：

- Spring Security 简介。
- Spring Security 整体架构。

### 1.1 Spring Security 简介

Java 企业级开发生态丰富，无论你想做哪方面的功能，都有众多的框架和工具可供选择，以至于 SUN 公司在早些年不得不制定了很多规范，这些规范在今天依然影响着我们的开发，安全领域也是如此。然而，不同于其他领域，在 Java 企业级开发中，安全管理方面的框架非常少，一般来说，主要有三种方案：

- Shiro
- Spring Security
- 开发者自己实现

Shiro 本身是一个老牌的安全管理框架，有着众多的优点，例如轻量、简单、易于集成、可以在 JavaSE 环境中使用等。不过，在微服务时代，Shiro 就显得力不从心了，在微服务面前，

它无法充分展示自己的优势。

也有开发者选择自己实现安全管理，据笔者所知，这一部分人不在少数。但是一个系统的安全，不仅仅是登录和权限控制这么简单，我们还要考虑各种各样可能存在的网络攻击以及防御策略，从这个角度来说，开发者自己实现安全管理也并非是一件容易的事情，只有大公司才有足够的人力物力去支持这件事情。

Spring Security 作为 Spring 家族的一员，在和 Spring 家族的其他成员如 Spring Boot、Spring Cloud 等进行整合时，具有其他框架无可比拟的优势，同时对 OAuth2 有着良好的支持，再加上 Spring Cloud 对 Spring Security 的不断加持（如推出 Spring Cloud Security），让 Spring Security 不知不觉中成为微服务项目的首选安全管理方案。

### 陈年旧事

Spring Security 最早叫 Acegi Security，这个名称并不是说它和 Spring 就没有关系，它依然是为 Spring 框架提供安全支持的。Acegi Security 基于 Spring，可以帮助我们为项目建立丰富的角色与权限管理系统。Acegi Security 虽然好用，但是最为人诟病的则是它臃肿烦琐的配置，这一问题最终也遗传给了 Spring Security。

Acegi Security 最终被并入 Spring Security 项目中，并于 2008 年 4 月发布了改名后的第一个版本 Spring Security 2.0.0，截止本书写作时，Spring Security 的最新版本已经到了 5.3.4。

和 Shiro 相比，Spring Security 重量级并且配置烦琐，直至今今天，依然有人以此为理由而拒绝了解 Spring Security。其实，自从 Spring Boot 推出后，就彻底颠覆了传统了 JavaEE 开发，自动化配置让许多事情变得非常容易，包括 Spring Security 的配置。在一个 Spring Boot 项目中，我们甚至只需要引入一个依赖，不需要任何额外配置，项目的所有接口就会被自动保护起来了。在 Spring Cloud 中，很多涉及安全管理的问题，也是一个 Spring Security 依赖两行配置就能搞定，在和 Spring 家族的产品一起使用时，Spring Security 的优势就非常明显了。

因此，在微服务时代，我们不需要纠结要不要学习 Spring Security，我们要考虑的是如何快速掌握 Spring Security，并且能够使用 Spring Security 实现我们微服务的安全管理。

## 1.2 Spring Security 核心功能

对于一个安全管理框架而言，无论是 Shiro 还是 Spring Security，最核心的功能，无非就是如下两方面：

- 认证
- 授权

通俗点说，认证就是身份验证（你是谁？），授权就是访问控制（你可以做什么？）。



## 1.2.1 认证

Spring Security 支持多种不同的认证方式，这些认证方式有的是 Spring Security 自己提供的认证功能，有的是第三方标准组织制订的。Spring Security 集成的主流认证机制主要有如下几种：

- 表单认证。
- OAuth2.0 认证。
- SAML2.0 认证。
- CAS 认证。
- RememberMe 自动认证。
- JAAS 认证。
- OpenID 去中心化认证。
- Pre-Authentication Scenarios 认证。
- X509 认证。
- HTTP Basic 认证。
- HTTP Digest 认证。

作为一个开放的平台，Spring Security 提供的认证机制不仅仅包括上面这些，我们还可以通过引入第三方依赖来支持更多的认证方式，同时，如果这些认证方式无法满足我们的需求，我们也可以自定义认证逻辑，特别是当我们和一些“老破旧”的系统进行集成时，自定义认证逻辑就显得非常重要了。

## 1.2.2 授权

无论采用了上面哪种认证方式，都不影响在 Spring Security 中使用授权功能。Spring Security 支持基于 URL 的请求授权、支持方法访问授权、支持 SpEL 访问控制、支持域对象安全（ACL），同时也支持动态权限配置、支持 RBAC 权限模型等，总之，我们常见的权限管理需求，Spring Security 基本上都是支持的。

## 1.2.3 其他

在认证和授权这两个核心功能之外，Spring Security 还提供了很多安全管理的“周边功能”，这也是一个非常重要的特色。

大部分 Java 工程师都不是专业的 Web 安全工程师，自己开发的安全管理框架可能会存在大大小小的安全漏洞。而 Spring Security 的强大之处在于，即使你不了解很多网络攻击，只要使用了 Spring Security，它会帮助我们自动防御很多网络攻击，例如 CSRF 攻击、会话固定攻击等，同时 Spring Security 还提供了 HTTP 防火墙来拦截大量的非法请求。由此可见，研究

Spring Security，也是研究常见的网络攻击以及防御策略。

对于大部分的 Java 项目而言，无论是从经济性还是安全性来考虑，使用 Spring Security 无疑是最佳方案。

## 1.3 Spring Security 整体架构

在具体学习 Spring Security 各种用法之前，我们先介绍一下 Spring Security 中常见的概念，以及认证、授权思路，方便读者从整体上把握 Spring Security 架构，这里涉及的所有组件，在后面的章节中还会做详细介绍。

### 1.3.1 认证和授权

#### 1.3.1.1 认证

在 Spring Security 的架构设计中，认证（Authentication）和授权（Authorization）是分开的，在本书后面的章节中读者可以看到，无论使用什么样的认证方式，都不会影响授权，这是两个独立的存在，这种独立带来的好处之一，就是 Spring Security 可以非常方便地整合一些外部的认证方案。

在 Spring Security 中，用户的认证信息主要由 Authentication 的实现类来保存，Authentication 接口定义如下：

```
public interface Authentication extends Principal, Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities();  
    Object getCredentials();  
    Object getDetails();  
    Object getPrincipal();  
    boolean isAuthenticated();  
    void setAuthenticated(boolean isAuthenticated);  
}
```

这里接口中定义的方法如下：

- getAuthorities 方法：用来获取用户的权限。
- getCredentials 方法：用来获取用户凭证，一般来说就是密码。
- getDetails 方法：用来获取用户携带的详细信息，可能是当前请求之类等。
- getPrincipal 方法：用来获取当前用户，例如是一个用户名或者一个用户对象。
- isAuthenticated：当前用户是否认证成功。

当用户使用用户名/密码登录或使用 Remember-me 登录时，都会对应一个不同的 Authentication 实例。

Spring Security 中的认证工作主要由 AuthenticationManager 接口来负责，下面来看一下该

接口的定义：

```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
}
```

`AuthenticationManager` 只有一个 `authenticate` 方法可以用来做认证，该方法有三个不同的返回值：

- 返回 `Authentication`，表示认证成功。
- 抛出 `AuthenticationException` 异常，表示用户输入了无效的凭证。
- 返回 `null`，表示不能断定。

`AuthenticationManager` 最主要的实现类是 `ProviderManager`，`ProviderManager` 管理了众多的 `AuthenticationProvider` 实例，`AuthenticationProvider` 有点类似于 `AuthenticationManager`，但是它多了一个 `supports` 方法用来判断是否支持给定的 `Authentication` 类型。

```
public interface AuthenticationProvider {  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
    boolean supports(Class<?> authentication);  
}
```

由于 `Authentication` 拥有众多不同的实现类，这些不同的实现类又由不同的 `AuthenticationProvider` 来处理，所以 `AuthenticationProvider` 会有一个 `supports` 方法，用来判断当前的 `Authentication Provider` 是否支持对应的 `Authentication`。

在一次完整的认证流程中，可能会同时存在多个 `AuthenticationProvider`（例如，项目同时支持 form 表单登录和短信验证码登录），多个 `AuthenticationProvider` 统一由 `ProviderManager` 来管理。同时，`ProviderManager` 具有一个可选的 `parent`，如果所有的 `AuthenticationProvider` 都认证失败，那么就会调用 `parent` 进行认证。`parent` 相当于一个备用认证方式，即各个 `AuthenticationProvider` 都无法处理认证问题的时候，就由 `parent` 出场收拾残局。

### 1.3.1.2 授权

当完成认证后，接下来就是授权了。在 Spring Security 的授权体系中，有两个关键接口：

- `AccessDecisionManager`
- `AccessDecisionVoter`

`AccessDecisionVoter` 是一个投票器，投票器会检查用户是否具备应有的角色，进而投出赞成、反对或者弃权票；`AccessDecisionManager` 则是一个决策器，来决定此次访问是否被允许。`AccessDecisionVoter` 和 `AccessDecisionManager` 都有众多的实现类，在 `AccessDecisionManager` 中会挨个遍历 `AccessDecisionVoter`，进而决定是否允许用户访问，因而 `AccessDecisionVoter` 和 `AccessDecisionManager` 两者的关系类似于 `AuthenticationProvider` 和 `ProviderManager` 的关系。

在 Spring Security 中，用户请求一个资源（通常是一个网络接口或者一个 Java 方法）所需要的角色会被封装成一个 ConfigAttribute 对象，在 ConfigAttribute 中只有一个 getAttribute 方法，该方法返回一个 String 字符串，就是角色的名称。一般来说，角色名称都带有一个 ROLE\_ 前缀，投票器 AccessDecisionVoter 所做的事情，其实就是比较用户所具备的角色和请求某个资源所需的 ConfigAttribute 之间的关系。

### 1.3.2 Web 安全

在 Spring Security 中，认证、授权等功能都是基于过滤器来完成的。表 1-1 列出了 Spring Security 中常见的过滤器，注意这里说的是否默认加载是指引入 Spring Security 依赖之后，开发者不做任何配置时，会自动加载的过滤器。

表 1-1 Spring Security 中的过滤器

过滤器	过滤器作用	是否默认加载
ChannelProcessingFilter	过滤请求协议，如 HTTPS 和 HTTP	NO
WebAsyncManagerIntegrationFilter	将 WebAsyncManager 与 Spring Security 上下文进行集成	YES
SecurityContextPersistenceFilter	在处理请求之前，将安全信息加载到 SecurityContextHolder 中以便后续使用。请求结束后，再擦除 SecurityContextHolder 中的信息	YES
HeaderWriterFilter	头信息加入到响应中	YES
CorsFilter	处理跨域问题	NO
CsrfFilter	处理 CSRF 攻击	YES
LogoutFilter	处理注销登录	YES
OAuth2AuthorizationRequestRedirectFilter	处理 OAuth2 认证重定向	NO
Saml2WebSsoAuthenticationRequestFilter	处理 SAML 认证	NO
X509AuthenticationFilter	处理 X509 认证	NO
AbstractPreAuthenticatedProcessingFilter	处理预认证问题	NO
CasAuthenticationFilter	处理 CAS 单点登录	NO
OAuth2LoginAuthenticationFilter	处理 OAuth2 认证	NO
Saml2WebSsoAuthenticationFilter	处理 SAML 认证	NO
UsernamePasswordAuthenticationFilter	处理表单登录	YES
OpenIDAuthenticationFilter	处理 OpenID 认证	NO
DefaultLoginPageGeneratingFilter	配置默认登录页面	YES
DefaultLogoutPageGeneratingFilter	配置默认注销页面	YES
ConcurrentSessionFilter	处理 Session 有效期	NO
DigestAuthenticationFilter	处理 HTTP 摘要认证	NO
BearerTokenAuthenticationFilter	处理 OAuth2 认证时的 Access Token	NO
BasicAuthenticationFilter	处理 HttpBasic 登录	YES
RequestCacheAwareFilter	处理请求缓存	YES

(续表)

过滤器	过滤器作用	是否默认加载
SecurityContextHolderAwareRequestFilter	包装原始请求	YES
JaasApiIntegrationFilter	处理 JAAS 认证	NO
RememberMeAuthenticationFilter	处理 RememberMe 登录	NO
AnonymousAuthenticationFilter	配置匿名认证	YES
OAuth2AuthorizationCodeGrantFilter	处理 OAuth2 认证中的授权码	NO
SessionManagementFilter	处理 Session 并发问题	YES
ExceptionTranslationFilter	处理异常认证/授权中的情况	YES
FilterSecurityInterceptor	处理授权	YES
SwitchUserFilter	处理账户切换	NO

开发者所见到的 Spring Security 提供的功能，都是通过这些过滤器来实现的，这些过滤器按照既定的优先级排列，最终形成一个过滤器链。开发者也可以自定义过滤器，并通过 `@Order` 注解去调整自定义过滤器在过滤器链中的位置。

需要注意的是，默认过滤器并不是直接放在 Web 项目的原生过滤器链中，而是通过一个 `FilterChainProxy` 来统一管理。Spring Security 中的过滤器链通过 `FilterChainProxy` 嵌入到 Web 项目的原生过滤器链中，如图 1-1 所示。

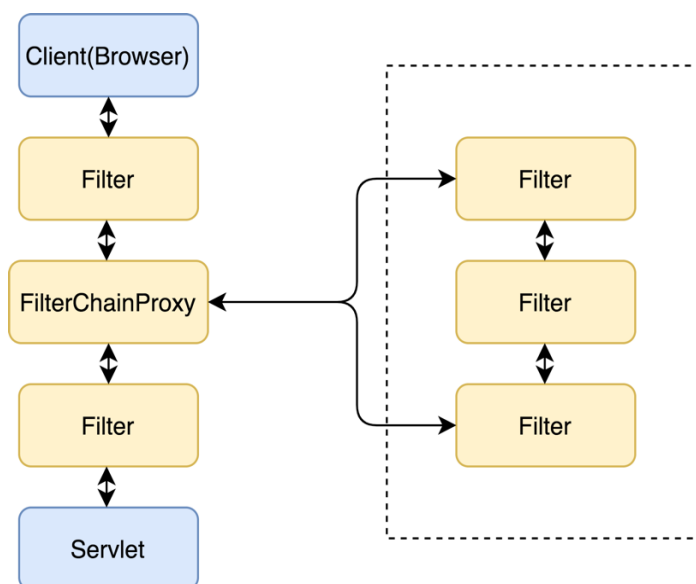


图 1-1 过滤器链通过 `FilterChainProxy` 出现在 Web 容器中

在 Spring Security 中，这样的过滤器链不仅仅只有一个，可能会有多个，如图 1-2 所示。当存在多个过滤器链时，多个过滤器链之间要指定优先级，当请求到达后，会从 `FilterChainProxy` 进行分发，先和哪个过滤器链匹配上，就用哪个过滤器链进行处理。当系统中存在多个不同的认证体系时，那么使用多个过滤器链就非常有效。

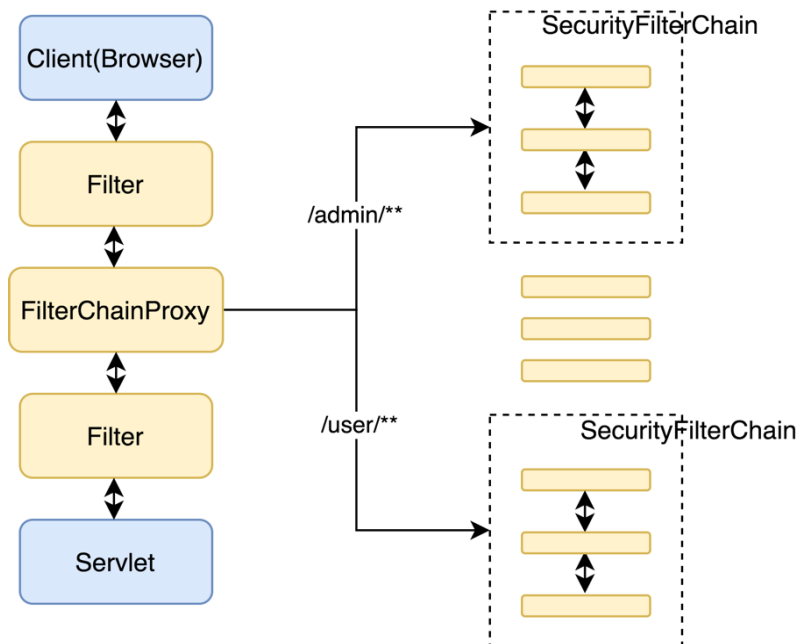


图 1-2 存在多个过滤器链时通过优先级进行匹配

FilterChainProxy 作为一个顶层管理者，将统一管理 Security Filter。FilterChainProxy 本身将通过 Spring 框架提供的 DelegatingFilterProxy 整合到原生过滤器链中，所以图 1-2 还可以做进一步的优化，如图 1-3 所示。

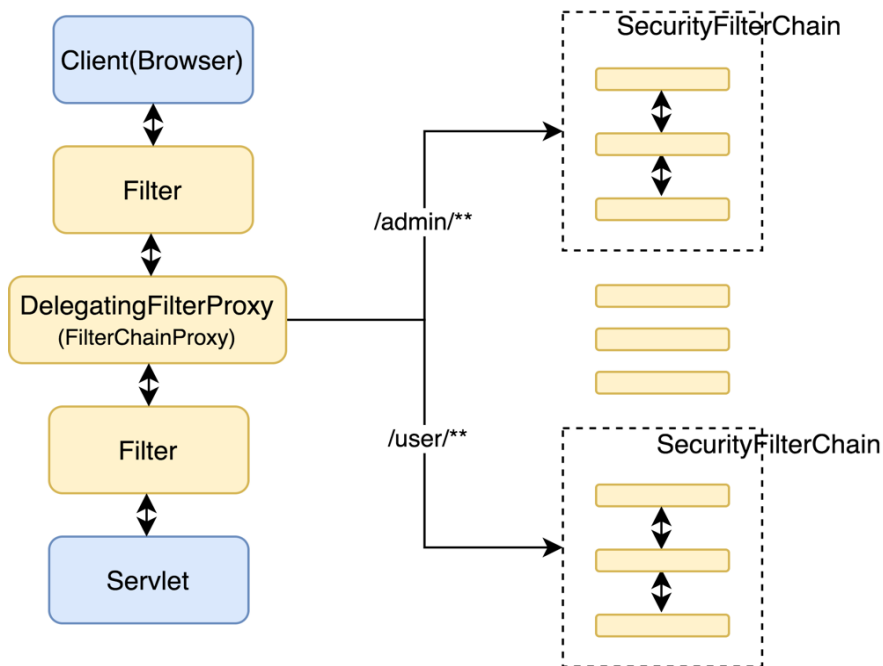


图 1-3 FilterChainProxy 通过 DelegatingFilterProxy 整合进 Web Filter

### 1.3.3 登录数据保存

如果不使用 Spring Security 这一类的安全管理框架，大部分的开发者可能会将登录用户数据保存在 Session 中，事实上，Spring Security 也是这么做的。但是，为了使用方便，Spring Security 在此基础上还做了一些改进，其中最主要的一个变化就是线程绑定。

当用户登录成功后，Spring Security 会将登录成功的用户信息保存到 SecurityContextHolder 中。SecurityContextHolder 中的数据保存默认是通过 ThreadLocal 来实现的，使用 ThreadLocal 创建的变量只能被当前线程访问，不能被其他线程访问和修改，也就是用户数据和请求线程绑定在一起。当登录请求处理完毕后，Spring Security 会将 SecurityContextHolder 中的数据拿出来保存到 Session 中，同时将 SecurityContextHolder 中的数据清空。以后每当有请求到来时，Spring Security 就会先从 Session 中取出用户登录数据，保存到 SecurityContextHolder 中，方便在该请求的后续处理过程中使用，同时在请求结束时将 SecurityContextHolder 中的数据拿出来保存到 Session 中，然后将 SecurityContextHolder 中的数据清空。

这一策略非常方便用户在 Controller 或者 Service 层获取当前登录用户数据，但是带来的另外一个问题就是，在子线程中想要获取用户登录数据就比较麻烦。Spring Security 对此也提供了相应的解决方案，如果开发者使用 @Async 注解来开启异步任务的话，那么只需要添加如下配置，使用 Spring Security 提供的异步任务代理，就可以在异步任务中从 SecurityContextHolder 里边获取当前登录用户的信息：

```
@Configuration
public class ApplicationConfiguration extends AsyncConfigurerSupport {
    @Override
    public Executor getAsyncExecutor() {
        return new DelegatingSecurityContextExecutorService(
            Executors.newFixedThreadPool(5));
    }
}
```

## 1.4 小 结

本章主要介绍了 Spring Security 的基本原理与整体架构，方便读者从整体上把握 Spring Security 认证、授权的实现原理。在接下来的章节中，我们将继续详细介绍 Spring Security 认证与授权的每一个实现细节。

# 第 2 章

---

## Spring Security 认证

对于安全管理框架而言，认证功能可以说是一切的起点，所以我们要研究 Spring Security，就要从最基本的认证开始。在 Spring Security 中，对认证功能做了大量的封装，以至于开发者只需要稍微配置一下就能使用认证功能，然而要深刻理解其源码却并非易事。本章将带领读者从最基本的用法开始讲解，最终再扩展到对源码的理解。

本章涉及的主要知识点有：

- Spring Security 基本认证。
- 登录表单配置。
- 登录用户数据获取。
- 用户的四种定义方式。

### 2.1 Spring Security 基本认证

#### 2.1.1 快速入门

在 Spring Boot 项目中使用 Spring Security 非常方便，创建一个新的 Spring Boot 项目，我们只需要引入 Web 和 Spring Security 依赖即可，具体代码如下：

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



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

然后我们在项目中提供一个用于测试的/hello 接口，代码如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello spring security";
    }
}
```

接下来启动项目，/hello 接口就已经被自动保护起来了。当用户访问/hello 接口时，会自动跳转到登录页面，如图 2-1 所示，用户登录成功后，才能访问到/hello 接口。

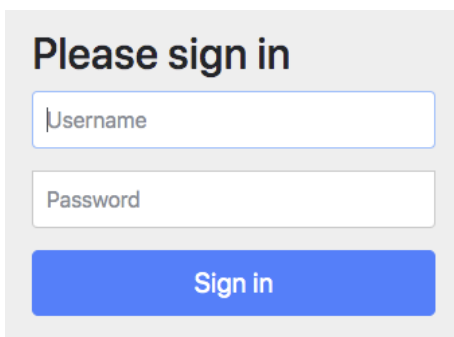
The image shows the default Spring Security login page. It has a light gray background. At the top, the text "Please sign in" is displayed in a bold, dark font. Below this, there are two input fields: the first is labeled "Username" and the second is labeled "Password". Both fields have a light blue border. At the bottom of the form, there is a blue button with the text "Sign in" in white.

图 2-1 Spring Security 默认登录页面

默认的登录用户名是 **user**，登录密码则是一个随机生成的 UUID 字符串，在项目启动日志中可以看到登录密码（这也意味着项目每次启动时，密码都会发生变化）：

```
Using generated security password: 8ef9c800-17cf-47a3-9984-8ff936db6dd8
```

输入默认的用户名和密码，就可以成功登录了。这就是 Spring Security 的强大之处，只需要引入一个依赖，所有的接口就会被自动保护起来。

### 2.1.2 流程分析

通过一个简单的流程图来看一下上面案例中的请求流程，如图 2-2 所示。

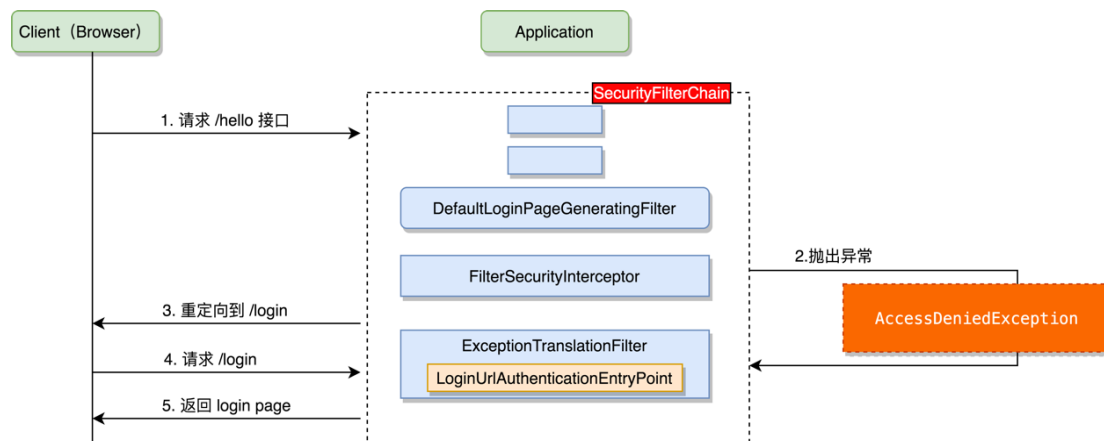


图 2-2 请求流程图

流程图比较清晰地说明了整个请求过程：

（1）客户端（浏览器）发起请求去访问/hello 接口，这个接口默认是需要认证之后才能访问的。

（2）这个请求会走一遍 Spring Security 中的过滤器链，在最后的 FilterSecurityInterceptor 过滤器中被拦截下来，因为系统发现用户未认证。请求拦截下来之后，接下来会抛出 AccessDeniedException 异常。

（3）抛出的 AccessDeniedException 异常在 ExceptionTranslationFilter 过滤器中被捕获，ExceptionTranslationFilter 过滤器通过调用 LoginUrlAuthenticationEntryPoint#commence 方法给客户端返回 302，要求客户端重定向到/login 页面。

（4）客户端发送/login 请求。

（5）/login 请求被 DefaultLoginPageGeneratingFilter 过滤器拦截下来，并在该过滤器中返回登录页面。所以当用户访问/hello 接口时会首先看到登录页面。

在整个过程中，相当于客户端一共发送了两个请求，第一个请求是/hello，服务端收到之后，返回 302，要求客户端重定向到/login，于是客户端又发送了/login 请求。

读者现在去理解上面这一个流程图可能还有些困难，等阅读完本章后面的内容之后，再回过头来看这个流程图，应该就会比较清晰了。

### 2.1.3 原理分析

在 2.1.1 小节中，虽然开发者只是引入了一个依赖，代码不多，但是 Spring Boot 背后却默默做了很多事情：

- 开启 Spring Security 自动化配置，开启后，会自动创建一个名为 springSecurityFilterChain 的过滤器，并注入到 Spring 容器中，这个过滤器将负责所有的安全管理，包括用户的认证、授权、重定向到登录页面等（springSecurityFilterChain 实际上代理了 Spring Security

中的过滤器链)。

- 创建一个 `UserDetailsService` 实例，`UserDetailsService` 负责提供用户数据，默认的用户数据是基于内存的用户，用户名为 `user`，密码则是随机生成的 UUID 字符串。
- 给用户生成一个默认的登录页面。
- 开启 CSRF 攻击防御。
- 开启会话固定攻击防御。
- 集成 X-XSS-Protection。
- 集成 X-Frame-Options 以防止单击劫持。

这里涉及的细节还是非常多的，登录的细节我们会在后面的章节继续详细介绍，这里主要分析一下默认用户的生成以及默认登录页面的生成。

### 2.1.3.1 默认用户生成

Spring Security 中定义了 `UserDetails` 接口来规范开发者自定义的用户对象，这样方便一些旧系统、用户表已经固定的系统集成到 Spring Security 认证体系中。

`UserDetails` 接口定义如下：

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

该接口中一共定义了 7 个方法：

- (1) `getAuthorities` 方法：返回当前账户所具备的权限。
- (2) `getPassword` 方法：返回当前账户的密码。
- (3) `getUsername` 方法：返回当前账户的用户名。
- (4) `isAccountNonExpired` 方法：返回当前账户是否未过期。
- (5) `isAccountNonLocked` 方法：返回当前账户是否未锁定。
- (6) `isCredentialsNonExpired` 方法：返回当前账户凭证（如密码）是否未过期。
- (7) `isEnabled` 方法：返回当前账户是否可用。

这是用户对象的定义，而负责提供用户数据源的接口是 `UserDetailsService`，`UserDetailsService` 中只有一个查询用户的方法，代码如下：

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

`loadUserByUsername` 有一个参数是 `username`，这是用户在认证时传入的用户名，最常见的就是用户在登录表单中输入的用户名（实际开发时还可能存在其他情况，例如使用 CAS 单点登录时，`username` 并非表单输入的用户名，而是 CAS Server 认证成功后回调的用户名参数），开发者在这里拿到用户名之后，再去数据库中查询用户，最终返回一个 `UserDetails` 实例。

在实际项目中，一般需要开发者自定义 `UserDetailsService` 的实现。如果开发者没有自定义 `UserDetailsService` 的实现，Spring Security 也为 `UserDetailsService` 提供了默认实现，如图 2-3 所示。

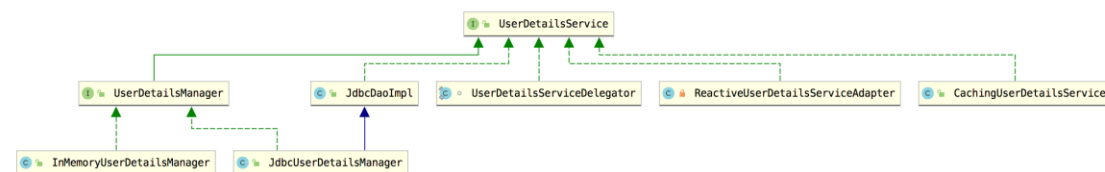


图 2-3 `UserDetailsService` 的默认实现类

- `UserDetailsServiceManager` 在 `UserDetailsService` 的基础上，继续定义了添加用户、更新用户、删除用户、修改密码以及判断用户是否存在共 5 种方法。
- `JdbcDaoImpl` 在 `UserDetailsService` 的基础上，通过 `spring-jdbc` 实现了从数据库中查询用户的方法。
- `InMemoryUserDetailsServiceManager` 实现了 `UserDetailsServiceManager` 中关于用户的增删改查方法，不过都是基于内存的操作，数据并没有持久化。
- `JdbcUserDetailsServiceManager` 继承自 `JdbcDaoImpl` 同时又实现了 `UserDetailsServiceManager` 接口，因此可以通过 `JdbcUserDetailsServiceManager` 实现对用户的增删改查操作，这些操作都会持久化到数据库中。不过 `JdbcUserDetailsServiceManager` 有一个局限性，就是操作数据库中用户的 SQL 都是提前写好的，不够灵活，因此在实际开发中 `JdbcUserDetailsServiceManager` 使用并不多。
- `CachingUserDetailsService` 的特点是会将 `UserDetailsService` 缓存起来。
- `UserDetailsServiceDelegator` 则是提供了 `UserDetailsService` 的懒加载功能。
- `ReactiveUserDetailsServiceAdapter` 是 `webflux-web-security` 模块定义的 `UserDetailsService` 实现。

当我们使用 Spring Security 时，如果仅仅是引入一个 Spring Security 依赖，则默认使用的用户就是由 `InMemoryUserDetailsServiceManager` 提供的。

大家知道，Spring Boot 之所以能够做到零配置使用 Spring Security，就是因为它提供了众多的自动化配置类。其中，针对 `UserDetailsService` 的自动化配置类是 `UserDetailsServiceAutoConfiguration`，这个类的源码并不长，我们一起来看一下：

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(AuthenticationManager.class)
@ConditionalOnBean(ObjectPostProcessor.class)
@ConditionalOnMissingBean(
    value = { AuthenticationManager.class,
              AuthenticationProvider.class,

```

```

        UserDetailsServiceImpl.class },
        type = { "org.springframework.security.oauth2.jwt.JwtDecoder",
            "org.springframework.security.oauth2.server.resource.
                introspection.OpaqueTokenIntrospector" })
public class UserDetailsServiceImplAutoConfiguration {
    private static final String NOOP_PASSWORD_PREFIX = "{noop}";
    private static final Pattern PASSWORD_ALGORITHM_PATTERN
        = Pattern.compile("^\\{.+}\\.*$");

    @Bean
    @ConditionalOnMissingBean(
        type = "org.springframework.security.oauth2.client.
            registration.ClientRegistrationRepository")

    @Lazy
    public InMemoryUserDetailsManager inMemoryUserDetailsManager(
        SecurityProperties properties,
        ObjectProvider<PasswordEncoder> passwordEncoder) {
        SecurityProperties.User user = properties.getUser();
        List<String> roles = user.getRoles();
        return new InMemoryUserDetailsManager(
            User.withUsername(user.getName())
                .password(getOrDeducePassword(user,
                    passwordEncoder.getIfAvailable()))
                .roles(StringUtils.toStringArray(roles)).build());
    }

    private String getOrDeducePassword(
        SecurityProperties.User user, PasswordEncoder encoder) {
        String password = user.getPassword();
        if (user.isPasswordGenerated()) {
            logger.info(String
                .format("%n%nUsing generated security password: %s%n",
                    user.getPassword()));
        }
        if (encoder != null
            || PASSWORD_ALGORITHM_PATTERN.matcher(password).matches()) {
            return password;
        }
        return NOOP_PASSWORD_PREFIX + password;
    }
}

```

从上述代码中可以看到，有两个比较重要的条件促使系统自动提供一个 `InMemoryUserDetailsManager` 的实例：

- (1) 当前 `classpath` 下存在 `AuthenticationManager` 类。
- (2) 当前项目中，系统没有提供 `AuthenticationManager`、`AuthenticationProvider`、`UserDetailsService` 以及 `ClientRegistrationRepository` 实例。

默认情况下，上面的条件都会满足，此时 Spring Security 会提供一个 `InMemoryUser`

DetailsManager 实例。从 inMemoryUserDetailsManager 方法中可以看到，用户数据源自 SecurityProperties#getUser 方法：

```
@ConfigurationProperties(prefix = "spring.security")
public class SecurityProperties {
    private User user = new User();
    public User getUser() {
        return this.user;
    }
    public static class User {
        private String name = "user";
        private String password = UUID.randomUUID().toString();
        private List<String> roles = new ArrayList<>();
        //省略 getter/setter
    }
}
```

从 SecurityProperties.User 类中，我们就可以看到默认的用户名是 user，默认的密码是一个 UUID 字符串。

再回到 inMemoryUserDetailsManager 方法中，构造 InMemoryUserDetailsManager 实例时需要一个 User 对象。这里的 User 对象不是 SecurityProperties.User，而是 org.springframework.security.core.userdetails.User，这是 Spring Security 提供的一个实现了 UserDetails 接口的用户类，该类提供了相应的静态方法，用来构造一个默认的用户实例。同时，默认的用户密码还在 getOrDeducePassword 方法中进行了二次处理，由于默认的 encoder 为 null，所以密码的二次处理只是给密码加了一个前缀{noop}，表示密码是明文存储的（关于 {noop}将在第 5 章密码加密中做详细介绍）。

经过以上的源码梳理，相信大家已经明白了 Spring Security 默认的用户名/密码是来自哪里了！

另外，当看了 SecurityProperties 的源码后，只要对 Spring Boot 中 properties 属性的加载机制有一点了解，就会明白，只要我们在项目的 application.properties 配置文件中添加如下配置，就能定制 SecurityProperties.User 类中各属性的值：

```
spring.security.user.name=javaboy
spring.security.user.password=123
spring.security.user.roles=admin,user
```

配置完成后，重启项目，此时登录的用户名就是 javaboy，登录密码就是 123，登录成功后用户具备 admin 和 user 两个角色。

### 2.1.3.2 默认页面生成

在 2.1.1 小节的案例中，一共存在两个默认页面，一个就是图 2-1 所示的登录页面，另外一个则是注销登录页面。当用户登录成功之后，在浏览器中输入 <http://localhost:8080/logout> 就可以看到注销登录页面，如图 2-4 所示。



```

        boolean logoutSuccess) {
    String errorMsg = "Invalid credentials";
    if (loginError) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            AuthenticationException ex = (AuthenticationException) session
                .getAttribute(WebAttributes.AUTHENTICATION_EXCEPTION);
            errorMsg = ex != null ? ex.getMessage() : "Invalid credentials";
        }
    }
    StringBuilder sb = new StringBuilder();
    String contextPath = request.getContextPath();
    if (this.formLoginEnabled) {
        sb.append("");
    }
    if (openIdEnabled) {
        sb.append("");
    }
    if (oauth2LoginEnabled) {
        sb.append("");
    }
    if (this.saml2LoginEnabled) {
        sb.append("");
    }
    return sb.toString();
}
}

```

`DefaultLoginPageGeneratingFilter` 的源码执行流程还是非常清晰的，我们梳理一下：

(1) 在 `doFilter` 方法中，首先判断出当前请求是否为登录出错请求、注销成功请求或者登录请求。如果是这三种请求中的任意一个，就会在 `DefaultLoginPageGeneratingFilter` 过滤器中生成登录页面并返回，否则请求继续往下走，执行下一个过滤器（这就是一开始的/hello 请求为什么没有被 `DefaultLoginPageGeneratingFilter` 拦截下来的原因）。

(2) 如果当前请求是登录出错请求、注销成功请求或者登录请求中的任意一个，就会调用 `generateLoginPageHtml` 方法去生成登录页面。在该方法中，如果有异常信息就把异常信息取出来一同返回给前端，然后根据不同的登录场景，生成不同的登录页面。生成过程其实就是字符串拼接，拼接出不同的登录表单（由于源码太长，上面没有贴出来具体的字符串拼接源码，读者可以自行查看 `DefaultLoginPageGeneratingFilter` 类的源码）。

(3) 登录页面生成后，接下来通过 `HttpServletResponse` 将登录页面写回到前端，然后调用 `return` 方法跳出过滤器链。

这就是 `DefaultLoginPageGeneratingFilter` 的工作过程。这里重点搞明白为什么/hello 请求没有被拦截，而/login 请求却被拦截了，其他都很好懂。

理解了 `DefaultLoginPageGeneratingFilter`，再来看 `DefaultLogoutPageGeneratingFilter` 就更



容易了，DefaultLogoutPageGeneratingFilter 部分核心源码如下：

```
public class DefaultLogoutPageGeneratingFilter extends OncePerRequestFilter {
    private RequestMatcher matcher = new AntPathRequestMatcher("/logout", "GET");
    @Override
    protected void doFilterInternal(HttpServletRequest request,
        HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        if (this.matcher.matches(request)) {
            renderLogout(request, response);
        } else {
            filterChain.doFilter(request, response);
        }
    }
    private void renderLogout(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException {
        String page = "";
        response.setContentType("text/html;charset=UTF-8");
        response.getWriter().write(page);
    }
}
```

从上述源码中可以看出，请求到来之后，会先判断是否是注销请求/logout，如果是/logout 请求，则渲染一个注销请求的页面返回给客户端，渲染过程和前面登录页面的渲染过程类似，也是字符串拼接（这里省略了字符串拼接，读者可以参考 DefaultLogoutPageGeneratingFilter 的源码）；否则请求继续往下走，执行下一个过滤器。

通过前面的分析，相信大家对这个简单的案例已经有所了解，看似只是加了一个依赖，但实际上 Spring Security 和 Spring Boot 在背后都默默做了很多事情，当然还有很多没有介绍到的，我们将在后面的章节中和大家一起继续深究。

## 2.2 登录表单配置

### 2.2.1 快速入门

理解了入门案例之后，接下来我们再来看一下登录表单的详细配置。

首先创建一个新的 Spring Boot 项目，引入 Web 和 Spring Security 依赖，代码如下：

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

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

项目创建好之后，为了方便测试，需要在 `application.properties` 中添加如下配置，将登录用户名和密码固定下来：

```
spring.security.user.name=javaboy
spring.security.user.password=123
```

接下来，我们在 `resources/static` 目录下创建一个 `login.html` 页面，这个是我们自定义的登录页面：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>登录</title>
  <link
    href="//maxcdn.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css"
    rel="stylesheet" id="bootstrap-css">
  <script
    src="//maxcdn.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js">
  </script>
  <script
    src="//cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js">
  </script>
</head>
<style>
  #login .container #login-row #login-column #login-box {
    border: 1px solid #9C9C9C;
    background-color: #EAEAEA;
  }
</style>
<body>
<div id="login">
  <div class="container">
    <div id="login-row"
      class="row justify-content-center align-items-center">
      <div id="login-column" class="col-md-6">
        <div id="login-box" class="col-md-12">
          <form id="login-form" class="form"
            action="/doLogin" method="post">
            <h3 class="text-center text-info">登录</h3>
            <div class="form-group">
              <label for="username"
                class="text-info">用户名:</label><br>
              <input type="text" name="uname"
                id="username" class="form-control">
```

```

        </div>
        <div class="form-group">
            <label for="password"
                class="text-info">密码:</label><br>
            <input type="text" name="passwd"
                id="password" class="form-control">
        </div>
        <div class="form-group">
            <input type="submit" name="submit"
                class="btn btn-info btn-md" value="登录">
        </div>
    </form>
</div>
</div>
</div>
</div>
</div>
</body>

```

这个 `logint.html` 中的核心内容就是一个登录表单，登录表单中有三个需要注意的地方：

- (1) `form` 的 `action`，这里给出的是 `/doLogin`，表示表单要提交到 `/doLogin` 接口上。
- (2) 用户名输入框的 `name` 属性值为 `uname`，当然这个值是可以自定义的，这里采用了 `uname`。
- (3) 密码输入框的 `name` 属性值为 `passwd`，`passwd` 也是可以自定义的。

`login.html` 定义好之后，接下来定义两个测试接口，作为受保护的资源。当用户登录成功后，就可以访问到受保护的资源。接口定义如下：

```

@RestController
public class LoginController {
    @RequestMapping("/index")
    public String index() {
        return "login success";
    }
    @RequestMapping("/hello")
    public String hello() {
        return "hello spring security";
    }
}

```

最后再提供一个 `Spring Security` 的配置类：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
    }
}

```

```
        .and()  
        .formLogin()  
        .loginPage("/login.html")  
        .loginProcessingUrl("/doLogin")  
        .defaultSuccessUrl("/index ")  
        .failureUrl("/login.html")  
        .usernameParameter("uname")  
        .passwordParameter("passwd")  
        .permitAll()  
        .and()  
        .csrf().disable();  
    }  
}
```

在 Spring Security 中,如果我们需要自定义配置,基本上都是继承自 `WebSecurityConfigurerAdapter` 来实现的,当然 `WebSecurityConfigurerAdapter` 本身的配置还是比较复杂,同时也是比较丰富的,这里先不做过多的展开,仅就结合上面的代码来解释,在下一个小节中我们将会对这里的配置再做更加详细的介绍。

(1) 首先 `configure` 方法中是一个链式配置,当然也可以不用链式配置,每一个属性配置完毕后再从 `http` 重新开始写起。

(2) `authorizeRequests()` 方法表示开启权限配置(该方法的含义其实比较复杂,我们在 13.4.4 小节还会再次介绍该方法), `.anyRequest().authenticated()` 表示所有的请求都要认证之后才能访问。

(3) 有的读者会对 `and()` 方法表示疑惑, `and()` 方法会返回 `HttpSecurityBuilder` 对象的一个子类(实际上就是 `HttpSecurity`), 所以 `and()` 方法相当于又回到 `HttpSecurity` 实例,重新开启新一轮的配置。如果觉得 `and()` 方法很难理解,也可以不用 `and()` 方法,在 `.anyRequest().authenticated()` 配置完成后直接用分号 (;) 结束,然后通过 `http.formLogin()` 继续配置表单登录。

(4) `formLogin()` 表示开启表单登录配置, `loginPage` 用来配置登录页面地址; `loginProcessingUrl` 用来配置登录接口地址; `defaultSuccessUrl` 表示登录成功后的跳转地址; `failureUrl` 表示登录失败后的跳转地址; `usernameParameter` 表示登录用户名的参数名称; `passwordParameter` 表示登录密码的参数名称; `permitAll` 表示跟登录相关的页面和接口不做拦截,直接通过。需要注意的是, `loginProcessingUrl`、`usernameParameter`、`passwordParameter` 需要和 `login.html` 中登录表单的配置一致。

(5) 最后的 `csrf().disable()` 表示禁用 CSRF 防御功能, Spring Security 自带了 CSRF 防御机制,但是我们这里为了测试方便,先将 CSRF 防御机制关闭,本书第 9 章将会详细介绍 CSRF 攻击与防御问题。

配置完成后,启动 Spring Boot 项目,浏览器地址栏中输入 `http://localhost:8080/index`,会自动跳转到 `http://localhost:8080/login.html` 页面,如图 2-5 所示。输入用户名和密码进行登录(用户名为 `javaboy`,密码为 `123`),登录成功之后,就可以访问到 `index` 页面了,如图 2-6 所示。

图 2-5 自定义的登录页面

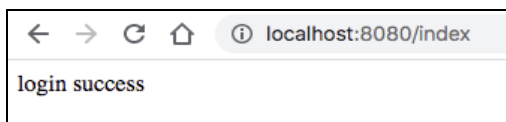


图 2-6 登录成功后的 index 页面

经过上面的配置，我们已经成功自定义了一个登录页面出来，用户在登录成功之后，就可以访问受保护的资源了。

## 2.2.2 配置细节

当然，前面的配置比较粗糙，这里还有一些配置的细节需要和读者分享一下。

在前面的配置中，我们用 `defaultSuccessUrl` 表示用户登录成功后的跳转地址，用 `failureUrl` 表示用户登录失败后的跳转地址。关于登录成功和登录失败，除了这两个方法可以配置之外，还有另外两个方法也可以配置。

### 2.2.2.1 登录成功

当用户登录成功之后，除了 `defaultSuccessUrl` 方法可以实现登录成功后的跳转之外，`successForwardUrl` 也可以实现登录成功后的跳转，代码如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login.html")
            .loginProcessingUrl("/doLogin")
            .successForwardUrl("/index ")
            .failureUrl("/login.html")
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .permitAll()
    }
}
```

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

`defaultSuccessUrl` 和 `successForwardUrl` 的区别如下：

(1) `defaultSuccessUrl` 表示当用户登录成功之后，会自动重定向到登录之前的地址上，如果用户本身就是直接访问的登录页面，则登录成功后就会重定向到 `defaultSuccessUrl` 指定的页面中。例如，用户在未认证的情况下，访问了/hello 页面，此时会自动重定向到登录页面，当用户登录成功后，就会自动重定向到/hello 页面；而用户如果一开始就访问登录页面，则登录成功后就会自动重定向到 `defaultSuccessUrl` 所指定的页面中。

(2) `successForwardUrl` 则不会考虑用户之前的访问地址，只要用户登录成功，就会通过服务器端跳转到 `successForwardUrl` 所指定的页面。

(3) `defaultSuccessUrl` 有一个重载方法，如果重载方法的第二个参数传入 `true`，则 `defaultSuccessUrl` 的效果与 `successForwardUrl` 类似，即不考虑用户之前的访问地址，只要登录成功，就重定向到 `defaultSuccessUrl` 所指定的页面。不同之处在于，`defaultSuccessUrl` 是通过重定向实现的跳转（客户端跳转），而 `successForwardUrl` 则是通过服务器端跳转实现的。

无论是 `defaultSuccessUrl` 还是 `successForwardUrl`，最终所配置的都是 `AuthenticationSuccessHandler` 接口的实例。

Spring Security 中专门提供了 `AuthenticationSuccessHandler` 接口用来处理登录成功事项：

```
public interface AuthenticationSuccessHandler {  
    default void onAuthenticationSuccess(HttpServletRequest request,  
                                         HttpServletResponse response,  
                                         FilterChain chain,  
                                         Authentication authentication)  
                                         throws IOException, ServletException{  
        onAuthenticationSuccess(request, response, authentication);  
        chain.doFilter(request, response);  
    }  
    void onAuthenticationSuccess(HttpServletRequest request,  
                                HttpServletResponse response,  
                                Authentication authentication)  
                                throws IOException, ServletException;  
}
```

由上述代码可以看到，`AuthenticationSuccessHandler` 接口中一共定义了两个方法，其中一个 `default` 方法，此方法是 Spring Security 5.2 开始加入进来的，在处理特定的认证请求 `Authentication Filter` 中会用到；另外一个非 `default` 方法，则用来处理登录成功的具体事项，其中 `request` 和 `response` 参数好理解，`authentication` 参数保存了登录成功的用户信息。我们将在后面的章节中详细介绍 `authentication` 参数。

`AuthenticationSuccessHandler` 接口共有三个实现类，如图 2-7 所示。

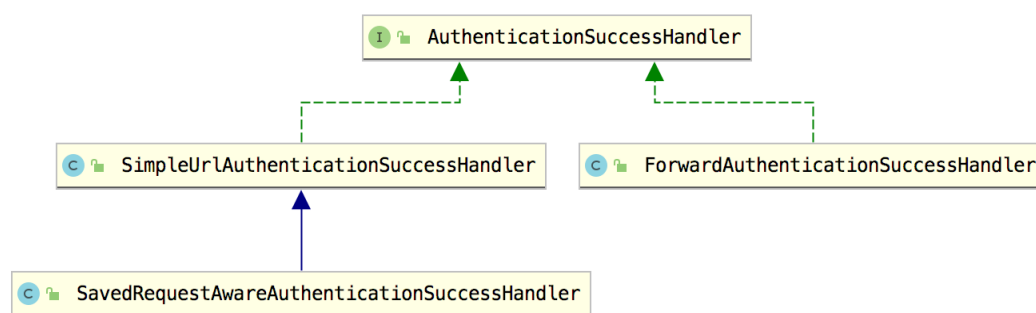


图 2-7 AuthenticationSuccessHandler 的三个实现类

(1) SimpleUrlAuthenticationSuccessHandler 继承自 AbstractAuthenticationTargetUrlRequestHandler, 通过 AbstractAuthenticationTargetUrlRequestHandler 中的 handle 方法实现请求重定向。

(2) SavedRequestAwareAuthenticationSuccessHandler 在 SimpleUrlAuthenticationSuccessHandler 的基础上增加了请求缓存的功能, 可以记录之前请求的地址, 进而在登录成功后重定向到一开始访问的地址。

(3) ForwardAuthenticationSuccessHandler 的实现则比较容易, 就是一个服务端跳转。

我们来重点分析 SavedRequestAwareAuthenticationSuccessHandler 和 ForwardAuthenticationSuccessHandler 的实现。

当通过 defaultSuccessUrl 来设置登录成功后重定向的地址时, 实际上对应的实现类就是 SavedRequestAwareAuthenticationSuccessHandler, 由于该类的源码比较长, 这里列出来一部分核心代码:

```

public class SavedRequestAwareAuthenticationSuccessHandler extends
    SimpleUrlAuthenticationSuccessHandler {
    private RequestCache requestCache = new HttpSessionRequestCache();
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
                                       HttpServletResponse response,
                                       Authentication authentication)
        throws ServletException, IOException {
        SavedRequest savedRequest = requestCache.getRequest(request, response);
        if (savedRequest == null) {
            super.onAuthenticationSuccess(request, response, authentication);
            return;
        }
        String targetUrlParameter = getTargetUrlParameter();
        if (isAlwaysUseDefaultTargetUrl()
            || (targetUrlParameter != null && StringUtils.hasText(request
                .getParameter(targetUrlParameter)))) {
            requestCache.removeRequest(request, response);
            super.onAuthenticationSuccess(request, response, authentication);
        }
    }
}
  
```

```

        return;
    }
    clearAuthenticationAttributes(request);
    String targetUrl = savedRequest.getRedirectUrl();
    getRedirectStrategy().sendRedirect(request, response, targetUrl);
}
public void setRequestCache(RequestCache requestCache) {
    this.requestCache = requestCache;
}
}

```

这里的核心方法就是 `onAuthenticationSuccess`:

(1) 首先从 `requestCache` 中获取缓存下来的请求, 如果没有获取到缓存请求, 就说明用户在访问登录页面之前并没有访问其他页面, 此时直接调用父类的 `onAuthenticationSuccess` 方法来处理, 最终会重定向到 `defaultSuccessUrl` 指定的地址。

(2) 接下来会获取一个 `targetUrlParameter`, 这个是用用户显式指定的、希望登录成功后重定向的地址, 例如用户发送的登录请求是 `http://localhost:8080/doLogin?target=/hello`, 这就表示当用户登录成功之后, 希望自动重定向到 `/hello` 这个接口。 `getTargetUrlParameter` 就是要获取重定向地址参数的 `key`, 也就是上面的 `target`, 拿到 `target` 之后, 就可以获取到重定向地址了。

(3) 如果 `targetUrlParameter` 存在, 或者用户设置了 `alwaysUseDefaultTargetUrl` 为 `true`, 这个时候缓存下来的请求就没有意义了。此时会直接调用父类的 `onAuthenticationSuccess` 方法完成重定向。 `targetUrlParameter` 存在, 则直接重定向到 `targetUrlParameter` 指定的地址; `alwaysUseDefaultTargetUrl` 为 `true`, 则直接重定向到 `defaultSuccessUrl` 指定的地址; 如果 `targetUrlParameter` 存在并且 `alwaysUseDefaultTargetUrl` 为 `true`, 则重定向到 `defaultSuccessUrl` 指定的地址。

(4) 如果前面的条件都不满足, 那么最终会从缓存请求 `savedRequest` 中获取重定向地址, 然后进行重定向操作。

这就是 `SavedRequestAwareAuthenticationSuccessHandler` 的实现逻辑, 开发者也可以配置自己的 `SavedRequestAwareAuthenticationSuccessHandler`, 代码如下:

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login.html")
            .loginProcessingUrl("/doLogin")
            .successHandler(successHandler())
            .failureUrl("/login.html")
            .usernameParameter("uname")
    }
}

```



```

        .passwordParameter("passwd")
        .permitAll()
        .and()
        .csrf().disable();
    }

    SavedRequestAwareAuthenticationSuccessHandler successHandler() {
        SavedRequestAwareAuthenticationSuccessHandler handler =
            new SavedRequestAwareAuthenticationSuccessHandler();
        handler.setDefaultTargetUrl("/index");
        handler.setTargetUrlParameter("target");
        return handler;
    }
}

```

注意在配置时指定了 `targetUrlParameter` 为 `target`，这样用户就可以在登录请求中，通过 `target` 来指定跳转地址了，然后我们修改一下前面 `login.html` 中的 `form` 表单：

```
<form id="login-form" class="form"
      action="/doLogin?target=/hello" method="post">
  <h3 class="text-center text-info">登录</h3>
  <div class="form-group">
    <label for="username" class="text-info">用户名:</label><br>
    <input type="text" name="uname" id="username" class="form-control">
  </div>
  <div class="form-group">
    <label for="password" class="text-info">密码:</label><br>
    <input type="text" name="passwd" id="password" class="form-control">
  </div>
  <div class="form-group">
    <input type="submit" name="submit" class="btn btn-info btn-md"
      value="登录">
  </div>
</form>
```

在 form 表单中，action 修改为/doLogin?target=/hello，这样当用户登录成功之后，就始终跳转到/hello 接口了。

当我们通过 `successForwardUrl` 来设置登录成功后重定向的地址时，实际上对应的实现类就是 `ForwardAuthenticationSuccessHandler`，`ForwardAuthenticationSuccessHandler` 的源码特别简单，就是一个服务端转发，代码如下：

[illegible]

```

        Authentication authentication)
        throws IOException, ServletException {
    request.getRequestDispatcher(forwardUrl).forward(request, response);
}
}

```

由上述代码可以看到，主要功能就是调用 `getRequestDispatcher` 方法进行服务端转发。

`AuthenticationSuccessHandler` 默认的三个实现类，无论是哪一个，都是用来处理页面跳转的。有时候页面跳转并不能满足我们的需求，特别是现在流行的前后端分离开发中，用户登录成功后，就不再需要页面跳转了，只需要给前端返回一个 JSON 数据即可，告诉前端登录成功还是登录失败，前端收到消息之后自行处理。像这样的需求，我们可以通过自定义 `AuthenticationSuccessHandler` 的实现类来完成：

```

public class MyAuthenticationSuccessHandler implements
    AuthenticationSuccessHandler{
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
                                       HttpServletResponse response,
                                       Authentication authentication)
        throws IOException, ServletException {
        response.setContentType("application/json;charset=utf-8");
        Map<String, Object> resp = new HashMap<>();
        resp.put("status", 200);
        resp.put("msg", "登录成功!");
        ObjectMapper om = new ObjectMapper();
        String s = om.writeValueAsString(resp);
        response.getWriter().write(s);
    }
}

```

在自定义的 `MyAuthenticationSuccessHandler` 中，重写 `onAuthenticationSuccess` 方法，在该方法中，通过 `HttpServletResponse` 对象返回一段登录成功的 JSON 字符串给前端即可。最后，在 `SecurityConfig` 中配置自定义的 `MyAuthenticationSuccessHandler`，代码如下：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login.html")
            .loginProcessingUrl("/doLogin")
            .successHandler(new MyAuthenticationSuccessHandler())
            .failureUrl("/login.html")
            .usernameParameter("uname")
    }
}

```



```

        <h3 class="text-center text-info">登录</h3>
        <div th:text="${SPRING_SECURITY_LAST_EXCEPTION}">
            </div>

        <div class="form-group">
            <label for="username"
                class="text-info">用户名:</label><br>
            <input type="text" name="uname"
                id="username" class="form-control">
        </div>
        <div class="form-group">
            <label for="password"
                class="text-info">密码:</label><br>
            <input type="text" name="passwd"
                id="password" class="form-control">
        </div>
        <div class="form-group">
            <input type="submit" name="submit"
                class="btn btn-info btn-md" value="登录">
        </div>
    </form>
</div>
</div>
</div>
</div>
</div>
</body>

```

mylogin.html 和前面的 login.html 基本类似，前面的 login.html 是静态页面，这里的 mylogin.html 是 thymeleaf 模板页面，mylogin.html 页面在 form 中多了一个 div，用来展示登录失败时候的异常信息，登录失败的异常信息会放在 request 中返回到前端，开发者可以将其直接提取出来展示。

既然 mylogin.html 是动态页面，就不能像静态页面那样直接访问了，需要我们给 mylogin.html 页面提供一个访问控制器：

```

@Controller
public class MyLoginController {
    @RequestMapping("/mylogin.html")
    public String mylogin() {
        return "mylogin";
    }
}

```

最后再在 SecurityConfig 中配置登录页面，代码如下：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {

```

```

        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/mylogin.html")
            .loginProcessingUrl("/doLogin")
            .defaultSuccessUrl("/index.html")
            .failureUrl("/mylogin.html")
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .permitAll()
            .and()
            .csrf().disable();
    }
}

```

`failureUrl` 表示登录失败后重定向到 `mylogin.html` 页面。重定向是一种客户端跳转，重定向不方便携带请求失败的异常信息（只能放在 URL 中）。

如果希望能够在前端展示请求失败的异常信息，可以使用下面这种方式：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/mylogin.html")
            .loginProcessingUrl("/doLogin")
            .defaultSuccessUrl("/index.html")
            .failureForwardUrl("/mylogin.html")
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .permitAll()
            .and()
            .csrf().disable();
    }
}

```

`failureForwardUrl` 方法从名字上就可以看出，这种跳转是一种服务器端跳转，服务器端跳转的好处是可以携带登录异常信息。如果登录失败，自动跳转回登录页面后，就可以将错误信息展示出来，如图 2-8 所示。

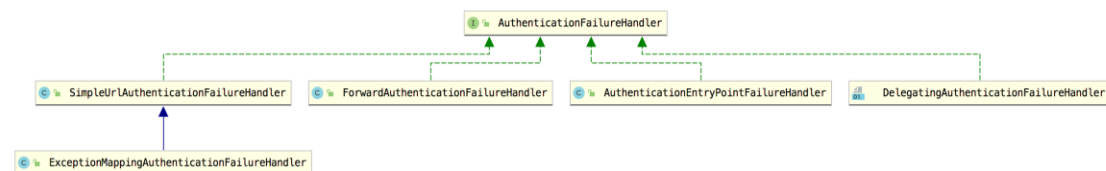


图 2-8 登录失败后展示异常信息

无论是 `failureUrl` 还是 `failureForwardUrl`，最终所配置的都是 `AuthenticationFailureHandler` 接口的实现。`Spring Security` 中提供了 `AuthenticationFailureHandler` 接口，用来规范登录失败的实现：

```
public interface AuthenticationFailureHandler {  
    void onAuthenticationFailure(HttpServletRequest request,  
        HttpServletResponse response, AuthenticationException exception)  
        throws IOException, ServletException;  
}
```

`AuthenticationFailureHandler` 接口中只有一个 `onAuthenticationFailure` 方法，用来处理登录失败请求，`request` 和 `response` 参数很好理解，最后的 `exception` 则表示登录失败的异常信息。`Spring Security` 中为 `AuthenticationFailureHandler` 一共提供了五个实现类，如图 2-9 所示。

图 2-9 `AuthenticationFailureHandler` 的实现类

(1) `SimpleUrlAuthenticationFailureHandler` 默认的处理逻辑就是通过重定向跳转到登录页面，当然也可以通过配置 `forwardToDestination` 属性将重定向改为服务器端跳转，`failureUrl` 方法的底层实现逻辑就是 `SimpleUrlAuthenticationFailureHandler`。

(2) `ExceptionMappingAuthenticationFailureHandler` 可以实现根据不同的异常类型，映射到不同的路径。

(3) `ForwardAuthenticationFailureHandler` 表示通过服务器端跳转来重新回到登录页面，`failureForwardUrl` 方法的底层实现逻辑就是 `ForwardAuthenticationFailureHandler`。

(4) `AuthenticationEntryPointFailureHandler` 是 `Spring Security 5.2` 新引进的处理类，可以通过 `AuthenticationEntryPoint` 来处理登录异常。

(5) `DelegatingAuthenticationFailureHandler` 可以实现为不同的异常类型配置不同的登录失败处理回调。

这里举一个简单的例子。假如不使用 `failureForwardUrl` 方法，同时又想在登录失败后通过服务器端跳转回到登录页面，那么可以自定义 `SimpleUrlAuthenticationFailureHandler` 配置，并将 `forwardToDestination` 属性设置为 `true`，代码如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/mylogin.html")
            .loginProcessingUrl("/doLogin")
            .defaultSuccessUrl("/index.html")
            .failureHandler(failureHandler())
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .permitAll()
            .and()
            .csrf().disable();
    }

    SimpleUrlAuthenticationFailureHandler failureHandler() {
        SimpleUrlAuthenticationFailureHandler handler =
            new SimpleUrlAuthenticationFailureHandler("/mylogin.html");
        handler.setUseForward(true);
        return handler;
    }
}
```

这样配置之后，如果用户再次登录失败，就会通过服务端跳转重新回到登录页面，登录页面也会展示相应的错误信息，效果和 `failureForwardUrl` 一致。

`SimpleUrlAuthenticationFailureHandler` 的源码也很简单，我们一起来看一下实现逻辑（源码比较长，这里列出来核心部分）：

```
public class SimpleUrlAuthenticationFailureHandler implements
    AuthenticationFailureHandler {
    private String defaultFailureUrl;
    private boolean forwardToDestination = false;
    private boolean allowSessionCreation = true;
    private RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();
    public SimpleUrlAuthenticationFailureHandler() {
    }
    public SimpleUrlAuthenticationFailureHandler(String defaultFailureUrl) {
        setDefaultFailureUrl(defaultFailureUrl);
    }
    public void onAuthenticationFailure(HttpServletRequest request,
        HttpServletResponse response, AuthenticationException exception)
```

```

        throws IOException, ServletException {
        if (defaultFailureUrl == null) {
            response.sendError(HttpStatus.UNAUTHORIZED.value(),
                HttpStatus.UNAUTHORIZED.getReasonPhrase());
        }
        else {
            saveException(request, exception);
            if (forwardToDestination) {
                request.getRequestDispatcher(defaultFailureUrl)
                    .forward(request, response);
            }
            else {
                redirectStrategy
                    .sendRedirect(request, response, defaultFailureUrl);
            }
        }
    }

    protected final void saveException(HttpServletRequest request,
        AuthenticationException exception) {
        if (forwardToDestination) {
            request
                .setAttribute(WebAttributes.AUTHENTICATION_EXCEPTION, exception);
        }
        else {
            HttpSession session = request.getSession(false);
            if (session != null || allowSessionCreation) {
                request.getSession()
                    .setAttribute(WebAttributes.AUTHENTICATION_EXCEPTION,
                        exception);
            }
        }
    }

    public void setUseForward(boolean forwardToDestination) {
        this.forwardToDestination = forwardToDestination;
    }
}

```

从这段源码中可以看到，当用户构造 `SimpleUrlAuthenticationFailureHandler` 对象的时候，就传入了 `defaultFailureUrl`，也就是登录失败时要跳转的地址。在 `onAuthenticationFailure` 方法中，如果发现 `defaultFailureUrl` 为 `null`，则直接通过 `response` 返回异常信息，否则调用 `saveException` 方法。在 `saveException` 方法中，如果 `forwardToDestination` 属性设置为 `true`，表示通过服务器端跳转回到登录页面，此时就把异常信息放到 `request` 中。再回到 `onAuthenticationFailure` 方法中，如果用户设置了 `forwardToDestination` 为 `true`，就通过服务器端跳转回到登录页面，否则通过重定向回到登录页面。

如果是前后端分离开发，登录失败时就不需要页面跳转了，只需要返回 JSON 字符串给前端即可，此时可以通过自定义 `AuthenticationFailureHandler` 的实现类来完成，代码如下：



```

public class MyAuthenticationFailureHandler implements
    AuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request,
                                         HttpServletResponse response,
                                         AuthenticationException exception)
        throws IOException, ServletException {
        response.setContentType("application/json;charset=utf-8");
        Map<String, Object> resp = new HashMap<>();
        resp.put("status", 500);
        resp.put("msg", "登录失败!" + exception.getMessage());
        ObjectMapper om = new ObjectMapper();
        String s = om.writeValueAsString(resp);
        response.getWriter().write(s);
    }
}

```

然后在 SecurityConfig 中进行配置即可：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/mylogin.html")
            .loginProcessingUrl("/doLogin")
            .defaultSuccessUrl("/index.html")
            .failureHandler(new MyAuthenticationFailureHandler())
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .permitAll()
            .and()
            .csrf().disable();
    }
}

```

配置完成后，当用户再次登录失败，就不会进行页面跳转了，而是直接返回 JSON 字符串，如图 2-10 所示。

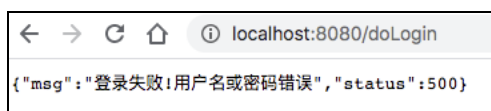


图 2-10 用户登录失败后直接返回 JSON 字符串

### 2.2.2.3 注销登录

Spring Security 中提供了默认的注销页面，当然开发者也可以根据自己的需求对注销登录进行定制。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            //省略其他配置
            .and()
            .logout()
            .logoutUrl("/logout")
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .logoutSuccessUrl("/mylogin.html")
            .and()
            .csrf().disable();
    }
}
```

- (1) 通过.logout()方法开启注销登录配置。
- (2) logoutUrl 指定了注销登录请求地址，默认是 GET 请求，路径为/logout。
- (3) invalidateHttpSession 表示是否使 session 失效，默认为 true。
- (4) clearAuthentication 表示是否清除认证信息，默认为 true。
- (5) logoutSuccessUrl 表示注销登录后的跳转地址。

配置完成后，再次启动项目，登录成功后，在浏览器中输入 <http://localhost:8080/logout> 就可以发起注销登录请求了。注销成功后，会自动跳转到 mylogin.html 页面。

如果项目有需要，开发者也可以配置多个注销登录的请求，同时还可以指定请求的方法：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            //省略其他配置
            .and()
            .logout()
            .logoutRequestMatcher(new OrRequestMatcher(
                new AntPathRequestMatcher("/logout1", "GET"),
```

```

        new AntPathRequestMatcher("/logout2", "POST")))
        .invalidateHttpSession(true)
        .clearAuthentication(true)
        .logoutSuccessUrl("/mylogin.html")
        .and()
        .csrf().disable();
    }
}

```

上面这个配置表示注销请求路径有两个：

- 第一个是/logout1，请求方法是 GET。
- 第二个是/logout2，请求方法是 POST。

使用任意一个请求都可以完成登录注销。

如果项目是前后端分离的架构，注销成功后就不需要页面跳转了，只需将注销成功的信息返回给前端即可，此时我们可以自定义返回内容：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            //省略其他配置
            .and()
            .logout()
            .logoutRequestMatcher(new OrRequestMatcher(
                new AntPathRequestMatcher("/logout1", "GET"),
                new AntPathRequestMatcher("/logout2", "POST")))
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .logoutSuccessHandler((req, resp, auth) -> {
                resp.setContentType("application/json; charset=utf-8");
                Map<String, Object> result = new HashMap<>();
                result.put("status", 200);
                result.put("msg", "注销成功!");
                ObjectMapper om = new ObjectMapper();
                String s = om.writeValueAsString(result);
                resp.getWriter().write(s);
            })
            .and()
            .csrf().disable();
    }
}

```

配置 `logoutSuccessHandler` 和 `logoutSuccessUrl` 类似于前面所介绍的 `successHandler` 和

`defaultSuccessUrl` 之间的关系，只是类不同而已，因此这里不再赘述，读者可以按照我们前面的分析思路自行分析。

配置完成后，重启项目，登录成功后再去注销登录，无论是使用 `/logout1` 还是 `/logout2` 进行注销，只要注销成功后，就会返回一段 JSON 字符串。

如果开发者希望为不同的注销地址返回不同的结果，也是可以的，配置如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            //省略其他配置
            .and()
            .logout()
            .logoutRequestMatcher(new OrRequestMatcher(
                new AntPathRequestMatcher("/logout1", "GET"),
                new AntPathRequestMatcher("/logout2", "POST")))
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .defaultLogoutSuccessHandlerFor((req, resp, auth) -> {
                resp.setContentType("application/json;charset=utf-8");
                Map<String, Object> result = new HashMap<>();
                result.put("status", 200);
                result.put("msg", "使用 logout1 注销成功!");
                ObjectMapper om = new ObjectMapper();
                String s = om.writeValueAsString(result);
                resp.getWriter().write(s);
            }, new AntPathRequestMatcher("/logout1", "GET"))
            .defaultLogoutSuccessHandlerFor((req, resp, auth) -> {
                resp.setContentType("application/json;charset=utf-8");
                Map<String, Object> result = new HashMap<>();
                result.put("status", 200);
                result.put("msg", "使用 logout2 注销成功!");
                ObjectMapper om = new ObjectMapper();
                String s = om.writeValueAsString(result);
                resp.getWriter().write(s);
            }, new AntPathRequestMatcher("/logout2", "POST"))
            .and()
            .csrf().disable();
    }
}
```

通过 `defaultLogoutSuccessHandlerFor` 方法可以注册多个不同的注销成功回调函数，该方法第一个参数是注销成功回调，第二个参数则是具体的注销请求。当用户注销成功后，使用了哪

个注销请求，就给出对应的响应信息。

## 2.3 登录用户数据获取

登录成功之后，在后续的业务逻辑中，开发者可能还需要获取登录成功的用户对象，如果不使用任何安全管理框架，那么可以将用户信息保存在 `HttpSession` 中，以后需要的时候直接从 `HttpSession` 中获取数据。在 `Spring Security` 中，用户登录信息本质上还是保存在 `HttpSession` 中，但是为了方便使用，`Spring Security` 对 `HttpSession` 中的用户信息进行了封装，封装之后，开发者若再想获取用户登录数据就会有两种不同的思路：

- (1) 从 `SecurityContextHolder` 中获取。
- (2) 从当前请求对象中获取。

这里列出来的两种方式是主流的做法，开发者也可以使用一些非主流的方式获取登录成功后的用户信息，例如直接从 `HttpSession` 中获取用户登录数据。

无论是哪种获取方式，都离不开一个重要的对象：`Authentication`。在 `Spring Security` 中，`Authentication` 对象主要有两方面的功能：

- (1) 作为 `AuthenticationManager` 的输入参数，提供用户身份认证的凭证，当它作为一个输入参数时，它的 `isAuthenticated` 方法返回 `false`，表示用户还未认证。
- (2) 代表已经经过身份认证的用户，此时的 `Authentication` 可以从 `SecurityContext` 中获取。

一个 `Authentication` 对象主要包含三个方面的信息：

- (1) **principal**：定义认证的用户。如果用户使用用户名/密码的方式登录，**principal** 通常就是一个 `UserDetails` 对象。
- (2) **credentials**：登录凭证，一般就是指密码。当用户登录成功之后，登录凭证会被自动擦除，以防止泄漏。
- (3) **authorities**：用户被授予的权限信息。

Java 中本身提供了 `Principal` 接口用来描述认证主体，`Principal` 可以代表一个公司、个人或者登录 ID。`Spring Security` 中定义了 `Authentication` 接口用来规范登录用户信息，`Authentication` 继承自 `Principal`：

```
public interface Authentication extends Principal, Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();
    Object getCredentials();
    Object getDetails();
    Object getPrincipal();
    boolean isAuthenticated();
    void setAuthenticated(boolean isAuthenticated);
}
```

```
}
```

这里接口中定义的方法都很好理解：

- `getAuthorities` 方法：用来获取用户权限。
- `getCredentials` 方法：用来获取用户凭证，一般来说就是密码。
- `getDetails` 方法：用来获取用户的详细信息，可能是当前的请求之类。
- `getPrincipal` 方法：用来获取当前用户信息，可能是一个用户名，也可能是一个用户对象。
- `isAuthenticated` 方法：当前用户是否认证成功。

可以看到，在 Spring Security 中，只要获取到 `Authentication` 对象，就可以获取到登录用户的详细信息。

不同的认证方式对应不同的 `Authentication` 实例，Spring Security 中的 `Authentication` 实现类如图 2-11 所示。

这些实现类现看起来可能会觉得陌生，不过没关系，在后续的章节中，这些实现类基本上都会涉及。现在我们先对每个类的功能做一个大概介绍：

(1) `AbstractAuthenticationToken`：该类实现了 `Authentication` 和 `CredentialsContainer` 两个接口，在 `AbstractAuthenticationToken` 中对 `Authentication` 接口定义的各个数据获取方法进行了实现，`CredentialsContainer` 则提供了登录凭证擦除方法。一般在登录成功后，为了防止用户信息泄漏，可以将登录凭证（例如密码）擦除。

(2) `RememberMeAuthenticationToken`：如果用户使用 `RememberMe` 的方式登录，登录信息将封装在 `RememberMeAuthenticationToken` 中。

(3) `TestingAuthenticationToken`：单元测试时封装的用户对象。

(4) `AnonymousAuthenticationToken`：匿名登录时封装的用户对象。

(5) `RunAsUserToken`：替换验证身份时封装的用户对象。

(6) `UsernamePasswordAuthenticationToken`：表单登录时封装的用户对象。

(7) `JaasAuthenticationToken`：JAAS 认证时封装的用户对象。

(8) `PreAuthenticatedAuthenticationToken`：Pre-Authentication 场景下封装的用户对象。

在这些 `Authentication` 的实例中，最常用的有两个：`UsernamePasswordAuthenticationToken` 和 `RememberMeAuthenticationToken`。在 2.1 节中的案例对应的用户认证对象就是 `UsernamePasswordAuthenticationToken`。

了解了 `Authentication` 对象之后，接下来我们来看一下如何在登录成功后获取用户登录信息，即 `Authentication` 对象。

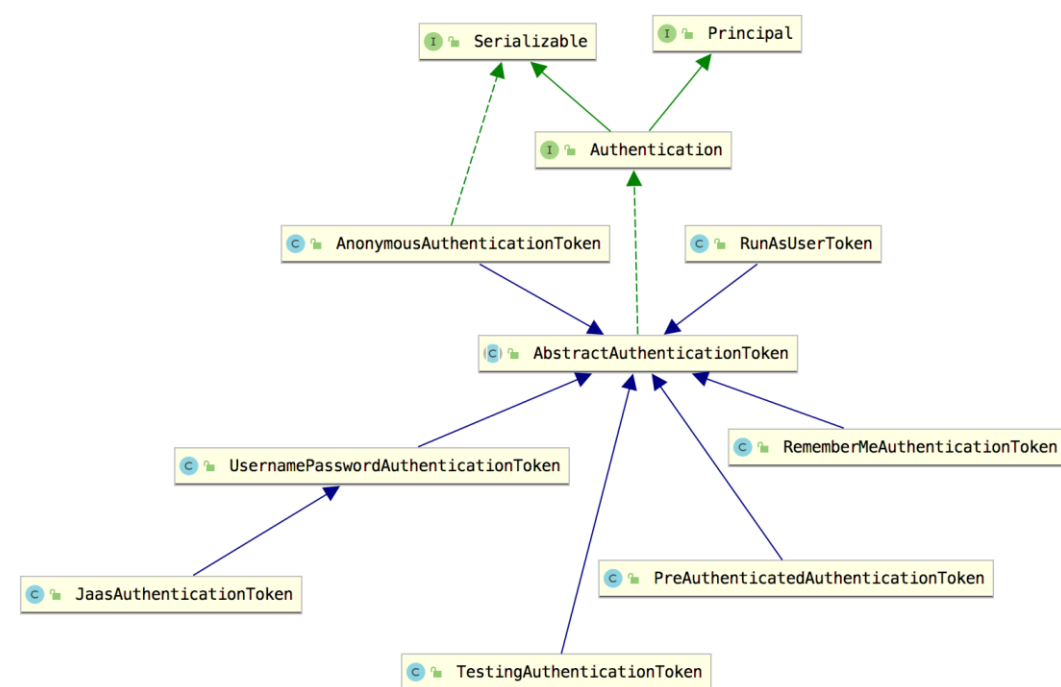


图 2-11 Authentication 的实现类

### 2.3.1 从 SecurityContextHolder 中获取

我们在 2.2 节案例的基础上，再添加一个 UserController，内容如下：

```

@RestController
public class UserController {
    @GetMapping("/user")
    public void userInfo() {
        Authentication authentication =
            SecurityContextHolder.getContext().getAuthentication();
        String name = authentication.getName();
        Collection<? extends GrantedAuthority> authorities =
            authentication.getAuthorities();

        System.out.println("name = " + name);
        System.out.println("authorities = " + authorities);
    }
}
  
```

配置完成后，启动项目，登录成功后，访问/user 接口，控制台就会打印出登录用户信息，当然，由于我们目前没有给用户配置角色，所以默认的用户角色为空数组，如图 2-12 所示。

```

name = javaboy
authorities = []
  
```

图 2-12 登录成功后打印出来的用户名和用户角色

这里为了演示方便，我们在 Controller 中获取登录用户信息，可以发现，`SecurityContextHolder.getContext()` 是一个静态方法，也就意味着我们随时随地都可以获取到登录用户信息，在 service 层也可以获取到登录用户信息（在实际项目中，大部分情况下也都是在 service 层获取登录用户信息）。

获取登录用户信息的代码很简单，那么 `SecurityContextHolder` 到底是什么？它里边的数据又是从何而来的？接下来我们将进行一一解析。

### 2.3.1.1 SecurityContextHolder

`SecurityContextHolder` 中存储的是 `SecurityContext`，`SecurityContext` 中存储的则是 `Authentication`，三者的关系如图 2-13 所示。

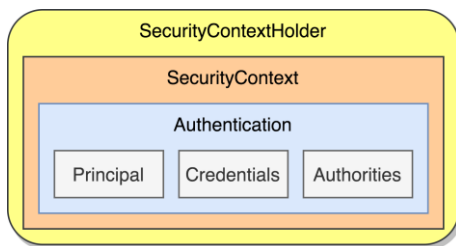


图 2-13 `SecurityContextHolder`、`SecurityContext` 以及 `Authentication` 之间的关系

这幅图清晰地描述了 `SecurityContextHolder`、`SecurityContext` 以及 `Authentication` 三者之间的关系。

首先在 `SecurityContextHolder` 中存放的是 `SecurityContext`，`SecurityContextHolder` 中定义了三种不同的数据存储策略，这实际上是一种典型的策略模式：

(1) `MODE_THREADLOCAL`：这种存放策略是将 `SecurityContext` 存放在 `ThreadLocal` 中，大家知道 `ThreadLocal` 的特点是在哪个线程中存储就要在哪个线程中读取，这其实非常适合 Web 应用，因为在默认情况下，一个请求无论经过多少 Filter 到达 Servlet，都是由一个线程来处理的。这也是 `SecurityContextHolder` 的默认存储策略，这种存储策略意味着如果在具体的业务处理代码中，开启了子线程，在子线程中去获取登录用户数据，就会获取不到。

(2) `MODE_INHERITABLETHREADLOCAL`：这种存储模式适用于多线程环境，如果希望在子线程中也能够获取到登录用户数据，那么可以使用这种存储模式。

(3) `MODE_GLOBAL`：这种存储模式实际上是将数据保存在一个静态变量中，在 Java Web 开发中，这种模式很少使用到。

Spring Security 中定义了 `SecurityContextHolderStrategy` 接口用来规范存储策略中的方法，我们来看一下：

```
public interface SecurityContextHolderStrategy {
    void clearContext();
    SecurityContext getContext();
    void setContext(SecurityContext context);
    SecurityContext createEmptyContext();
}
```



```
}

```

接口中一共定义了四个方法：

- (1) **clearContext**：该方法用来清除存储的 `SecurityContext` 对象。
- (2) **getContext**：该方法用来获取存储的 `SecurityContext` 对象。
- (3) **setContext**：该方法用来设置存储的 `SecurityContext` 对象。
- (4) **createEmptyContext**：该方法则用来创建一个空的 `SecurityContext` 对象。

在 Spring Security 中，`SecurityContextHolderStrategy` 接口一共有三个实现类，对应了三种不同的存储策略，如图 2-14 所示。

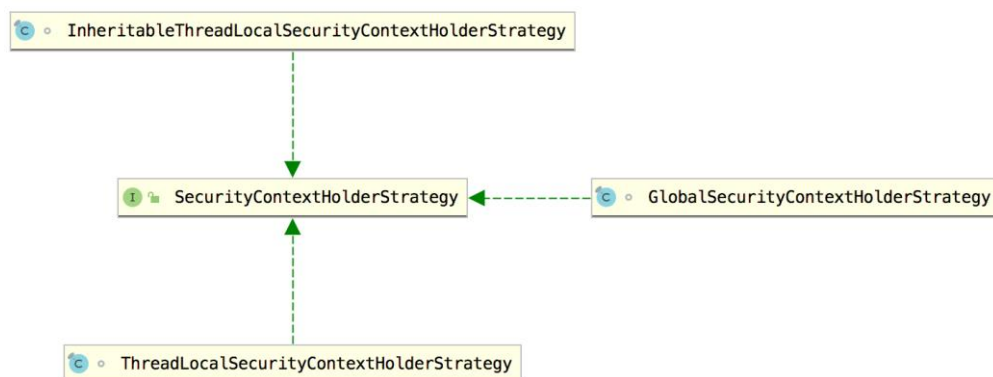


图 2-14 `SecurityContextHolderStrategy` 的三个实现类

每一个实现类都对应了不同的实现策略，我们先来看一下 `ThreadLocalSecurityContextHolderStrategy`：

```

final class ThreadLocalSecurityContextHolderStrategy implements
    SecurityContextHolderStrategy {
    private static final ThreadLocal<SecurityContext> contextHolder =
        new ThreadLocal<>();

    public void clearContext() {
        contextHolder.remove();
    }

    public SecurityContext getContext() {
        SecurityContext ctx = contextHolder.get();
        if (ctx == null) {
            ctx = createEmptyContext();
            contextHolder.set(ctx);
        }
        return ctx;
    }

    public void setContext(SecurityContext context) {
        contextHolder.set(context);
    }

    public SecurityContext createEmptyContext() {

```

```

        return new SecurityContextImpl();
    }
}

```

`ThreadLocalSecurityContextHolderStrategy` 实现了 `SecurityContextHolderStrategy` 接口，并实现了接口中的方法，存储数据的载体就是一个 `ThreadLocal`，所以针对 `SecurityContext` 的清空、获取以及存储，都是在 `ThreadLocal` 中进行操作，例如清空就是调用 `ThreadLocal` 的 `remove` 方法。`SecurityContext` 是一个接口，它只有一个实现类 `SecurityContextImpl`，所以创建就直接新建一个 `SecurityContextImpl` 对象即可。

再来看 `InheritableThreadLocalSecurityContextHolderStrategy`：

```

final class InheritableThreadLocalSecurityContextHolderStrategy
    implements SecurityContextHolderStrategy {
    private static final ThreadLocal<SecurityContext> contextHolder =
        new InheritableThreadLocal<>();

    public void clearContext() {
        contextHolder.remove();
    }

    public SecurityContext getContext() {
        SecurityContext ctx = contextHolder.get();
        if (ctx == null) {
            ctx = createEmptyContext();
            contextHolder.set(ctx);
        }
        return ctx;
    }

    public void setContext(SecurityContext context) {
        contextHolder.set(context);
    }

    public SecurityContext createEmptyContext() {
        return new SecurityContextImpl();
    }
}

```

`InheritableThreadLocalSecurityContextHolderStrategy` 和 `ThreadLocalSecurityContextHolderStrategy` 的实现策略基本一致，不同的是存储数据的载体变了，在 `InheritableThreadLocalSecurityContextHolderStrategy` 中存储数据的载体变成了 `InheritableThreadLocal`。`InheritableThreadLocal` 继承自 `ThreadLocal`，但是多了一个特性，就是在子线程创建的一瞬间，会自动将父线程中的数据复制到子线程中。该存储策略正是利用了这一特性，实现了在子线程中获取登录用户信息的功能。

最后再来看一下 `GlobalSecurityContextHolderStrategy`：

```

final class GlobalSecurityContextHolderStrategy implements
    SecurityContextHolderStrategy {
    private static SecurityContext contextHolder;

    public void clearContext() {

```

```

        contextHolder = null;
    }
    public SecurityContext getContext() {
        if (contextHolder == null) {
            contextHolder = new SecurityContextImpl();
        }
        return contextHolder;
    }
    public void setContext(SecurityContext context) {
        contextHolder = context;
    }
    public SecurityContext createEmptyContext() {
        return new SecurityContextImpl();
    }
}

```

`GlobalSecurityContextHolderStrategy` 的实现就更简单了，用一个静态变量来保存 `SecurityContext`，所以它也可以在多线程环境下使用。但是一般在 Web 开发中，这种存储策略使用得较少。

最后我们再来看一下 `SecurityContextHolder` 的源码：

```

public class SecurityContextHolder {
    public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";
    public static final String MODE_INHERITABLETHREADLOCAL =
        "MODE_INHERITABLETHREADLOCAL";
    public static final String MODE_GLOBAL = "MODE_GLOBAL";
    public static final String SYSTEM_PROPERTY = "spring.security.strategy";
    private static String strategyName = System.getProperty(SYSTEM_PROPERTY);
    private static SecurityContextHolderStrategy strategy;
    private static int initializeCount = 0;
    static {
        initialize();
    }
    public static void clearContext() {
        strategy.clearContext();
    }
    public static SecurityContext getContext() {
        return strategy.getContext();
    }
    public static int getInitializeCount() {
        return initializeCount;
    }
    private static void initialize() {
        if (!StringUtils.hasText(strategyName)) {
            strategyName = MODE_THREADLOCAL;
        }
        if (strategyName.equals(MODE_THREADLOCAL)) {
            strategy = new ThreadLocalSecurityContextHolderStrategy();
        }
    }
}

```

```

    }
    else if (strategyName.equals(MODE_INHERITABLETHREADLOCAL)) {
        strategy = new InheritableThreadLocalSecurityContextHolderStrategy();
    }
    else if (strategyName.equals(MODE_GLOBAL)) {
        strategy = new GlobalSecurityContextHolderStrategy();
    }
    else {
        try {
            Class<?> clazz = Class.forName(strategyName);
            Constructor<?> customStrategy = clazz.getConstructor();
            strategy =
                (SecurityContextHolderStrategy) customStrategy.newInstance();
        }
        catch (Exception ex) {
            ReflectionUtils.handleReflectionException(ex);
        }
    }
    initializeCount++;
}

public static void setContext(SecurityContext context) {
    strategy.setContext(context);
}

public static void setStrategyName(String strategyName) {
    SecurityContextHolder.strategyName = strategyName;
    initialize();
}

public static SecurityContextHolderStrategy getContextHolderStrategy() {
    return strategy;
}

public static SecurityContext createEmptyContext() {
    return strategy.createEmptyContext();
}
}

```

从这段源码中可以看到，`SecurityContextHolder` 定义了三个静态常量用来描述三种不同的存储策略；存储策略 `strategy` 会在静态代码块中进行初始化，根据不同的 `strategyName` 初始化不同的存储策略；`strategyName` 变量表示目前正在使用的存储策略，开发者可以通过配置系统变量或者调用 `setStrategyName` 来修改 `SecurityContextHolder` 中的存储策略，调用 `setStrategyName` 后会重新初始化 `strategy`。

默认情况下，如果开发者试图从子线程中获取当前登录用户数据，就会获取失败，代码如下：

```

@RestController
public class UserController {
    @GetMapping("/user")
    public void userInfo() {

```

```

Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
String name = authentication.getName();
Collection<? extends GrantedAuthority> authorities =
    authentication.getAuthorities();
System.out.println("name = " + name);
System.out.println("authorities = " + authorities);
new Thread(new Runnable() {
    @Override
    public void run() {
        Authentication authentication =
            SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null) {
            System.out.println("获取用户信息失败");
            return;
        }
        String name = authentication.getName();
        Collection<? extends GrantedAuthority> authorities =
            authentication.getAuthorities();
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + ":name = " + name);
        System.out.println(threadName + ":authorities = " + authorities);
    }
}).start();
}
}

```

在子线程中尝试获取登录用户数据时，获取到的数据为 `null`，如图 2-15 所示。

```

name = javaboy
authorities = []
获取用户信息失败

```

图 2-15 子线程获取登录用户信息为 `null`

子线程之所以获取不到登录用户信息，就是因为数据存储在 `ThreadLocal` 中，存储和读取不是同一个线程，所以获取不到。如果希望子线程中也能够获取到登录用户信息，可以将 `SecurityContextHolder` 中的存储策略改为 `MODE_INHERITABLETHREADLOCAL`，这样就支持多线程环境下获取登录用户信息了。

默认的存储策略是通过 `System.getProperty` 加载的，因此我们可以通过配置系统变量来修改默认的存储策略，以 IntelliJ IDEA 为例，首先单击启动按钮，选择 `Edit Configurations` 按钮，如图 2-16 所示，然后在打开的选项中，配置 `VM options` 参数，添加如下一行，配置界面如图 2-17 所示。

```
-Dspring.security.strategy=MODE_INHERITABLETHREADLOCAL
```

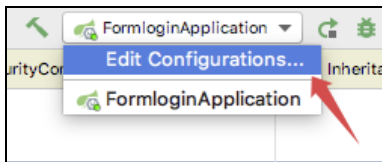


图 2-16 编辑启动配置

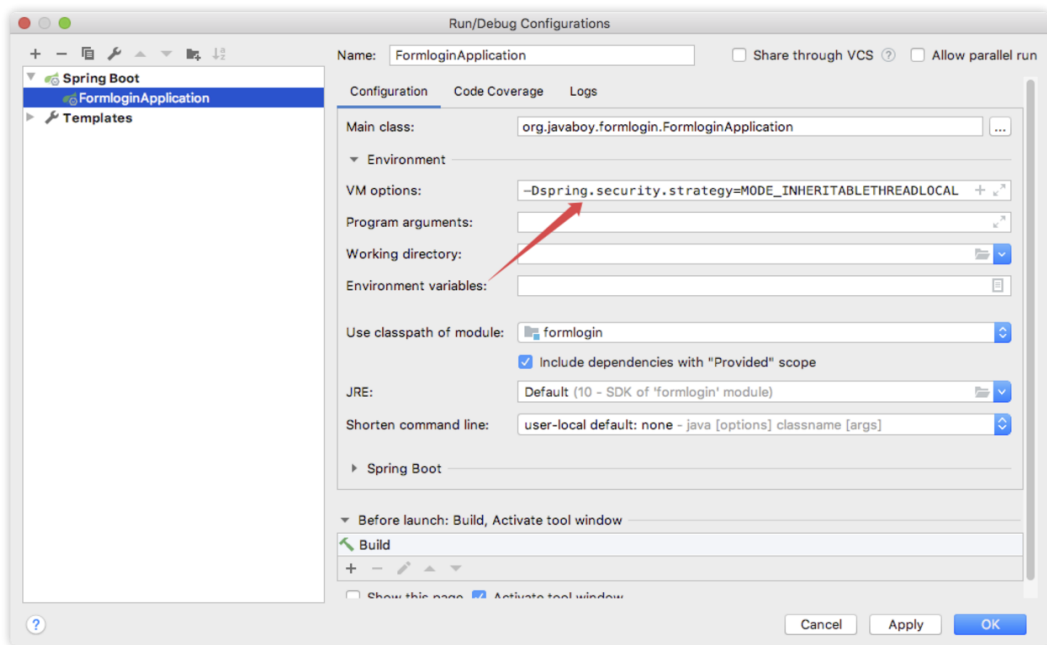


图 2-17 配置 SecurityContextHolder 中的存储策略

这样，在 SecurityContextHolder 中通过 System.getProperty 加载到的默认存储策略就支持多线程环境了。

配置完成之后，再次启动项目，此时访问/user 接口，即使在子线程中，也可以获取到登录用户信息了，如图 2-18 所示。

```
name = javaboy
authorities = []
Thread-132:name = javaboy
Thread-132:authorities = []
```

图 2-18 子线程中也可以获取到登录用户信息

看到这里读者不禁要问了，既然 SecurityContextHolder 默认是将用户信息存储在 ThreadLocal 中，在 Spring Boot 中不同的请求都是由不同的线程处理的，那为什么每一次请求都能从 SecurityContextHolder 中获取到登录用户信息呢？这就不得不提到 Spring Security 过滤器链中重要的一环——SecurityContextPersistenceFilter。

### 2.3.1.2 SecurityContextPersistenceFilter

前面介绍了 Spring Security 中的常见过滤器，在这些过滤器中，存在一个非常重要的过滤

器就是 `SecurityContextPersistenceFilter`。

默认情况下，在 Spring Security 过滤器链中，`SecurityContextPersistenceFilter` 是第二道防线，位于 `WebAsyncManagerIntegrationFilter` 之后。从 `SecurityContextPersistenceFilter` 这个过滤器的名字上就可以推断出来，它的作用是为了存储 `SecurityContext` 而设计的。

整体上来说，`SecurityContextPersistenceFilter` 主要做两件事情：

(1) 当一个请求到来时，从 `HttpSession` 中获取 `SecurityContext` 并存入 `SecurityContextHolder` 中，这样在同一个请求的后续处理过程中，开发者始终可以通过 `SecurityContextHolder` 获取到当前登录用户信息。

(2) 当一个请求处理完毕时，从 `SecurityContextHolder` 中获取 `SecurityContext` 并存入 `HttpSession` 中（主要针对异步 Servlet），方便下一个请求到来时，再从 `HttpSession` 中拿出来使用，同时擦除 `SecurityContextHolder` 中的登录用户信息。

### 注 意

在 `SecurityContextPersistenceFilter` 过滤器中，当一个请求处理完毕时，从 `SecurityContextHolder` 中获取 `SecurityContext` 存入 `HttpSession` 中，这一步的操作主要是针对异步 Servlet。如果不是异步 Servlet，在响应提交时，就会将 `SecurityContext` 保存到 `HttpSession` 中了，而不会等到在 `SecurityContextPersistenceFilter` 过滤器中再去存储。

这就是 `SecurityContextPersistenceFilter` 大致上做的事情，在正式开始介绍 `SecurityContextPersistenceFilter` 之前，需要先介绍另外一个接口，这就是 `SecurityContextRepository` 接口。

将 `SecurityContext` 存入 `HttpSession`，或者从 `HttpSession` 中加载数据并转为 `SecurityContext` 对象，这些事情都是由 `SecurityContextRepository` 接口的实现类完成的，因此这里我们就先从 `SecurityContextRepository` 接口开始看起。

首先我们来看一下 `SecurityContextRepository` 接口的定义：

```
public interface SecurityContextRepository {
    SecurityContext loadContext(HttpRequestResponseHolder holder);
    void saveContext(SecurityContext context, HttpServletRequest request,
        HttpServletResponse response);
    boolean containsContext(HttpServletRequest request);
}
```

`SecurityContextRepository` 接口中一共定义了三个方法：

(1) `loadContext`：这个方法用来加载 `SecurityContext` 对象出来，对于没有登录的用户，这里会返回一个空的 `SecurityContext` 对象，注意空的 `SecurityContext` 对象是指 `SecurityContext` 中不存在 `Authentication` 对象，而不是该方法返回 `null`。

(2) `saveContext`：该方法用来保存一个 `SecurityContext` 对象。

(3) `containsContext`：该方法可以判断 `SecurityContext` 对象是否存在。

在 Spring Security 框架中，为 `SecurityContextRepository` 接口一共提供了三个实现类，如

图 2-19 所示。

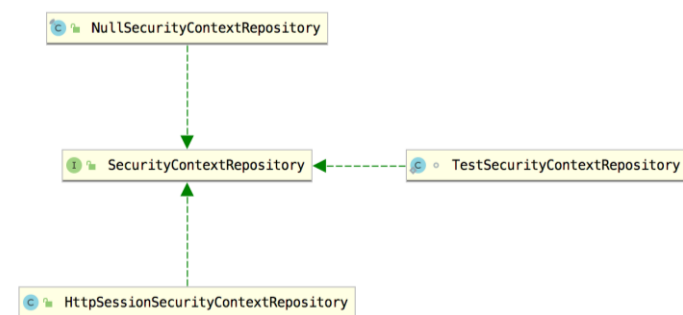


图 2-19 SecurityContextRepository 接口的三个实现类

在这三个实现类中，TestSecurityContextRepository 为单元测试提供支持；NullSecurityContextRepository 实现类中，loadContext 方法总是返回一个空的 SecurityContext 对象，saveContext 方法未做任何实现，containsContext 方法总是返回 false，所以 NullSecurityContextRepository 实现类实际上未做 SecurityContext 的存储工作。

在 Spring Security 中默认使用的实现类是 HttpSessionSecurityContextRepository，通过 HttpSessionSecurityContextRepository 实现了将 SecurityContext 存储到 HttpSession 以及从 HttpSession 中加载 SecurityContext 出来。这里我们来重点看一下 HttpSessionSecurityContextRepository 类。

在正式开始介绍 HttpSessionSecurityContextRepository 之前，首先来看一下 HttpSessionSecurityContextRepository 中定义的关于请求和封装的两个内部类。

首先是 HttpSessionSecurityContextRepository 中定义的对于响应的封装类 SaveToSessionResponseWrapper，我们先来看一下 SaveToSessionResponseWrapper 的继承关系图，如图 2-20 所示。

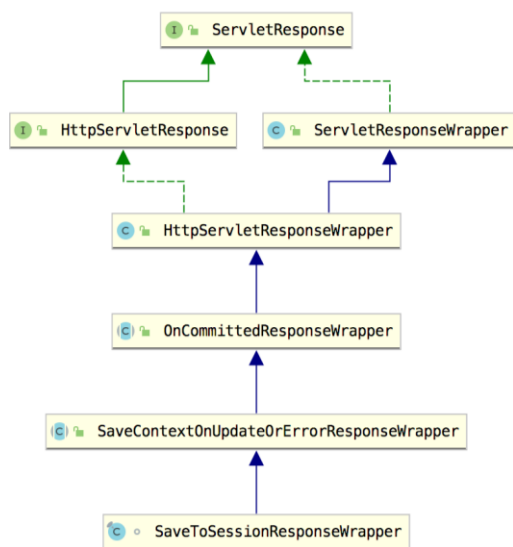


图 2-20 SaveToSessionResponseWrapper 继承关系图



从这幅继承关系图中可以看到, `SaveToSessionResponseWrapper` 实际上就是我们所熟知的 `HttpServletResponse` 功能的扩展。这里有三个关键的实现类:

(1) `HttpServletResponseWrapper`: `HttpServletResponseWrapper` 实现了 `HttpServletResponse` 接口, 它是 `HttpServletResponse` 的装饰类, 利用 `HttpServletResponseWrapper` 可以方便地操作参数和输出流等。

(2) `OnCommittedResponseWrapper`: `OnCommittedResponseWrapper` 继承自 `HttpServletResponseWrapper`, 对其功能进行了增强, 最重要的增强在于可以获取 `HttpServletResponse` 的提交行为。当 `HttpServletResponse` 的 `sendError`、`sendRedirect`、`flushBuffer`、`flush` 以及 `close` 等方法被调用时, `onResponseCommitted` 方法会被触发, 开发者可以在 `onResponseCommitted` 方法中做一些数据保存操作, 例如保存 `SecurityContext`。不过 `OnCommittedResponseWrapper` 中的 `onResponseCommitted` 方法只是一个抽象方法, 并没有具体的实现, 具体的实现则在它的实现类 `SaveContextOnUpdateOrErrorResponseWrapper` 中。

(3) `SaveContextOnUpdateOrErrorResponseWrapper`: 该类继承自 `OnCommittedResponseWrapper` 并对 `onResponseCommitted` 方法做了实现。在 `SaveContextOnUpdateOrErrorResponseWrapper` 类中声明了一个 `contextSaved` 变量, 表示 `SecurityContext` 是否已经存储成功。当 `HttpServletResponse` 提交时, 会调用 `onResponseCommitted` 方法, 在 `onResponseCommitted` 方法中调用 `saveContext` 方法, 将 `SecurityContext` 保存到 `HttpSession` 中, 同时将 `contextSaved` 变量标记为 `true`。`saveContext` 方法在这里也是一个抽象方法, 具体的实现则在 `SaveToSessionResponseWrapper` 类中。

接下来看一下 `HttpSessionSecurityContextRepository` 中 `SaveToSessionResponseWrapper` 的定义:

```
final class SaveToSessionResponseWrapper extends
    SaveContextOnUpdateOrErrorResponseWrapper {
    private final HttpServletRequest request;
    private final boolean httpSessionExistedAtStartOfRequest;
    private final SecurityContext contextBeforeExecution;
    private final Authentication authBeforeExecution;
    SaveToSessionResponseWrapper(HttpServletRequest response,
        HttpServletRequest request,
        boolean httpSessionExistedAtStartOfRequest,
        SecurityContext context) {
        super(response, disableUrlRewriting);
        this.request = request;
        this.httpSessionExistedAtStartOfRequest =
            httpSessionExistedAtStartOfRequest;
        this.contextBeforeExecution = context;
        this.authBeforeExecution = context.getAuthentication();
    }
    @Override
    protected void saveContext(SecurityContext context) {
```

```

        final Authentication authentication = context.getAuthentication();
        HttpSession httpSession = request.getSession(false);
        if (authentication == null ||
            trustResolver.isAnonymous(authentication)) {
            if (httpSession != null && authBeforeExecution != null) {
                httpSession.removeAttribute(springSecurityContextKey);
            }
            return;
        }
        if (httpSession == null) {
            httpSession = createNewSessionIfAllowed(context);
        }
        if (httpSession != null) {
            if (contextChanged(context)
                ||
                httpSession.getAttribute(springSecurityContextKey) == null) {
                httpSession.setAttribute(springSecurityContextKey, context);
            }
        }
    }

    private boolean contextChanged(SecurityContext context) {
        return context != contextBeforeExecution
            || context.getAuthentication() != authBeforeExecution;
    }

    private HttpSession createNewSessionIfAllowed(SecurityContext context) {
        if (isTransientAuthentication(context.getAuthentication())) {
            return null;
        }
        if (httpSessionExistedAtStartOfRequest) {
            return null;
        }
        if (!allowSessionCreation) {
            return null;
        }
        if (contextObject.equals(context)) {
            return null;
        }
        try {
            return request.getSession(true);
        }
        catch (IllegalStateException e) {
        }
        return null;
    }
}

```

在 `SaveToSessionResponseWrapper` 中其实主要定义了三个方法：`saveContext`、`context Changed` 以及 `createNewSessionIfAllowed`。

(1) `saveContext`: 该方法主要是用来保存 `SecurityContext`, 如果 `authentication` 对象为 `null` 或者它是一个匿名对象, 则不需要保存 `SecurityContext` (参见 SEC-776: <https://github.com/spring-projects/spring-security/issues/1036>); 同时, 如果 `httpSession` 不为 `null` 并且 `authBeforeExecution` 也不为 `null`, 就从 `httpSession` 中将保存的登录用户数据移除, 这个主要是为了防止开发者在注销成功的回调中继续调用 `chain.doFilter` 方法, 进而导致原始的登录信息无法清除的问题 (参见 SEC-1587: <https://github.com/spring-projects/spring-security/issues/1826>); 如果 `httpSession` 为 `null`, 则去创建一个 `HttpSession` 对象; 最后, 如果 `SecurityContext` 发生了变化, 或者 `httpSession` 中没有保存 `SecurityContext`, 则调用 `httpSession` 中的 `setAttribute` 方法将 `SecurityContext` 保存起来。

(2) `contextChanged`: 该方法主要用来判断 `SecurityContext` 是否发生变化, 因为在程序运行过程中, 开发者可能修改了 `SecurityContext` 中的 `Authentication` 对象。

(3) `createNewSessionIfAllowed`: 该方法用来创建一个 `HttpSession` 对象。

这就是 `HttpSessionSecurityContextRepository` 中封装的 `SaveToSessionResponseWrapper` 对象, 一个核心功能就是在 `HttpServletResponse` 提交的时候, 将 `SecurityContext` 保存到 `HttpSession` 中。

接下来看一下 `HttpSessionSecurityContextRepository` 中关于 `SaveToSessionRequestWrapper` 的定义, `SaveToSessionRequestWrapper` 相对而言就要简单很多了:

```
private static class SaveToSessionRequestWrapper extends
    HttpServletRequestWrapper {
    private final SaveContextOnUpdateOrElseErrorResponseWrapper response;
    SaveToSessionRequestWrapper(HttpServletRequest request,
        SaveContextOnUpdateOrElseErrorResponseWrapper response) {
        super(request);
        this.response = response;
    }
    @Override
    public AsyncContext startAsync() {
        response.disableSaveOnResponseCommitted();
        return super.startAsync();
    }
    @Override
    public AsyncContext startAsync(ServletRequest servletRequest,
        ServletResponse servletResponse) throws IllegalStateException {
        response.disableSaveOnResponseCommitted();
        return super.startAsync(servletRequest, servletResponse);
    }
}
```

`SaveToSessionRequestWrapper` 类实际上是在 Spring Security 3.2 之后出现的封装类, 在 Spring Security 3.2 之前并不存在 `SaveToSessionRequestWrapper` 类。封装的 `SaveToSessionRequestWrapper` 类主要作用是禁止在异步 Servlet 提交时, 自动保存 `SecurityContext`。

为什么要禁止呢？我们来看如下一段简单的代码：

```
@GetMapping("/user2")
public void userInfo(HttpServletRequest req, HttpServletResponse resp) {
    AsyncContext asyncContext = req.startAsync();
    CompletableFuture.runAsync(() -> {
        try {
            PrintWriter out = asyncContext.getResponse().getWriter();
            out.write("hello javaboy!");
            asyncContext.complete();
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
}
```

可以看到，在异步 Servlet 中，当任务执行完毕之后，`HttpServletResponse` 也会自动提交，在提交的过程中会自动保存 `SecurityContext` 到 `HttpSession` 中，但是由于是在子线程中，因此无法获取到 `SecurityContext` 对象（`SecurityContextHolder` 默认将数据存储在 `ThreadLocal` 中），所以会保存失败。如果开发者使用了异步 Servlet，则默认情况下会禁用 `HttpServletResponse` 提交时自动保存 `SecurityContext` 这一功能，改为在 `SecurityContextPersistenceFilter` 过滤器中完成 `SecurityContext` 保存操作。

看完了 `HttpSessionSecurityContextRepository` 中封装的两个请求/响应对象之后，接下来我们再来整体上看一下 `HttpSessionSecurityContextRepository` 类的功能：

```
public class HttpSessionSecurityContextRepository implements
    SecurityContextRepository {
    public static final String SPRING_SECURITY_CONTEXT_KEY =
        "SPRING_SECURITY_CONTEXT";

    private final Object contextObject =
        SecurityContextHolder.createEmptyContext();

    private boolean allowSessionCreation = true;
    private boolean disableUrlRewriting = false;
    private String springSecurityContextKey = SPRING_SECURITY_CONTEXT_KEY;
    private AuthenticationTrustResolver trustResolver =
        new AuthenticationTrustResolverImpl();

    public SecurityContext loadContext(
        HttpRequestResponseHolder requestResponseHolder) {
        HttpServletRequest request = requestResponseHolder.getRequest();
        HttpServletResponse response = requestResponseHolder.getResponse();
        HttpSession httpSession = request.getSession(false);
        SecurityContext context = readSecurityContextFromSession(httpSession);
        if (context == null) {
            context = generateNewContext();
        }
        SaveToSessionResponseWrapper wrappedResponse =
            new SaveToSessionResponseWrapper(response,
```

```

        request,
        httpSession != null,
        context);

requestResponseHolder.setResponse(wrappedResponse);
requestResponseHolder.setRequest(new SaveToSessionRequestWrapper(
    request, wrappedResponse));
return context;
}

public void saveContext(SecurityContext context,
    HttpServletRequest request,
    HttpServletResponse response) {
    SaveContextOnUpdateOrElseResponseWrapper responseWrapper = WebUtils
        .getNativeResponse(response,
            SaveContextOnUpdateOrElseResponseWrapper.class);
    if (responseWrapper == null) {
        throw new IllegalStateException("");
    }
    if (!responseWrapper.isContextSaved()) {
        responseWrapper.saveContext(context);
    }
}

public boolean containsContext(HttpServletRequest request) {
    HttpSession session = request.getSession(false);
    if (session == null) {
        return false;
    }
    return session.getAttribute(springSecurityContextKey) != null;
}

private SecurityContext readSecurityContextFromSession(
    HttpSession httpSession) {

    if (httpSession == null) {
        return null;
    }
    Object contextFromSession =
        httpSession.getAttribute(springSecurityContextKey);
    if (contextFromSession == null) {
        return null;
    }
    if (!(contextFromSession instanceof SecurityContext)) {
        return null;
    }
    return (SecurityContext) contextFromSession;
}

protected SecurityContext generateNewContext() {
    return SecurityContextHolder.createEmptyContext();
}

public void setAllowSessionCreation(boolean allowSessionCreation) {
    this.allowSessionCreation = allowSessionCreation;
}

```

```

    }
    public void setDisableUrlRewriting(boolean disableUrlRewriting) {
        this.disableUrlRewriting = disableUrlRewriting;
    }
    public void setSpringSecurityContextKey(String springSecurityContextKey) {
        this.springSecurityContextKey = springSecurityContextKey;
    }
    private static class SaveToSessionRequestWrapper extends
        HttpServletRequestWrapper {
        //省略
    }
    final class SaveToSessionResponseWrapper extends
        SaveContextOnUpdateOrElseResponseWrapper {
        //省略
    }
    private boolean isTransientAuthentication(Authentication authentication) {
        return AnnotationUtils.getAnnotation(authentication.getClass(),
            Transient.class) != null;
    }
    public void setTrustResolver(AuthenticationTrustResolver trustResolver) {
        this.trustResolver = trustResolver;
    }
}

```

(1) 首先通过 `SPRING_SECURITY_CONTEXT_KEY` 变量定义了 `SecurityContext` 在 `HttpSession` 中存储的 key，如果开发者需要手动操作 `HttpSession` 中存储的 `SecurityContext`，可以通过该 key 来操作。

(2) `trustResolver` 是一个用户身份评估器，用来判断当前用户是匿名用户还是通过 `RememberMe` 登录的用户。

(3) 在 `loadContext` 方法中，通过调用 `readSecurityContextFromSession` 方法来获取 `SecurityContext` 对象。如果获取到的对象为 `null`，则调用 `generateNewContext` 方法去生成一个空的 `SecurityContext` 对象，最后构造请求和响应的装饰类并存入 `requestResponseHolder` 对象中。

(4) `saveContext` 方法用来保存 `SecurityContext`，在保存之前，会先调用 `isContextSaved` 方法判断是否已经保存了，如果已经保存了，则不再保存。正常情况下，在 `HttpServletResponse` 提交时 `SecurityContext` 就已经保存到 `HttpSession` 中了；如果是异步 `Servlet`，则提交时不会自动将 `SecurityContext` 保存到 `HttpSession`，此时会在这里进行保存操作。

(5) `containsContext` 方法用来判断请求中是否存在 `SecurityContext` 对象。

(6) `readSecurityContextFromSession` 方法执行具体的 `SecurityContext` 读取逻辑，从 `HttpSession` 中获取 `SecurityContext` 并返回。

(7) `generateNewContext` 方法用来生成一个不包含 `Authentication` 的空的 `SecurityContext` 对象。

(8) `setAllowSessionCreation` 方法用来设置是否允许创建 `HttpSession`，默认是 `true`。

(9) `setDisableUrlRewriting` 方法表示是否禁用 URL 重写，默认是 `false`。

(10) `setSpringSecurityContextKey` 方法可以用来配置 `HttpSession` 中存储 `SecurityContext` 的 key。

(11) `isTransientAuthentication` 方法用来判断 `Authentication` 是否免于存储。

(12) `setTrustResolver` 方法用来配置身份评估器。

这就是 `HttpSessionSecurityContextRepository` 所提供的所有功能，这些功能都将在 `SecurityContextPersistenceFilter` 过滤器中进行调用，那么接下来我们就来看一下 `SecurityContextPersistenceFilter` 中的调用逻辑：

```
public class SecurityContextPersistenceFilter extends GenericFilterBean {
    private SecurityContextRepository repo;
    private boolean forceEagerSessionCreation = false;
    public SecurityContextPersistenceFilter() {
        this(new HttpSessionSecurityContextRepository());
    }
    public SecurityContextPersistenceFilter(SecurityContextRepository repo) {
        this.repo = repo;
    }
    public void doFilter(ServletRequest req,
                        ServletResponse res, FilterChain chain)
                        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        if (request.getAttribute(FILTER_APPLIED) != null) {
            chain.doFilter(request, response);
            return;
        }
        request.setAttribute(FILTER_APPLIED, Boolean.TRUE);
        if (forceEagerSessionCreation) {
            HttpSession session = request.getSession();
        }
        HttpRequestResponseHolder holder =
            new HttpRequestResponseHolder(request, response);
        SecurityContext contextBeforeChainExecution =
            repo.loadContext(holder);
        try {
            SecurityContextHolder.setContext(contextBeforeChainExecution);
            chain.doFilter(holder.getRequest(), holder.getResponse());
        }
        finally {
            SecurityContext contextAfterChainExecution =
                SecurityContextHolder.getContext();
            SecurityContextHolder.clearContext();
            repo.saveContext(contextAfterChainExecution,
                            holder.getRequest(),
                            holder.getResponse());
            request.removeAttribute(FILTER_APPLIED);
        }
    }
}
```

```
    }  
}  
public void setForceEagerSessionCreation(  
    boolean forceEagerSessionCreation) {  
    this.forceEagerSessionCreation = forceEagerSessionCreation;  
}  
}
```

过滤器的核心方法当然是 `doFilter`，我们就从 `doFilter` 方法开始介绍：

(1) 首先从 `request` 中获取 `FILTER_APPLIED` 属性，如果该属性值不为 `null`，则直接执行 `chain.doFilter` 方法，当前过滤器到此为止，这个判断主要是确保该请求只执行一次该过滤器。如果确实是该 `request` 第一次经过该过滤器，则给其设置上 `FILTER_APPLIED` 属性。

(2) `forceEagerSessionCreation` 变量表示是否要在过滤器链执行之前确保会话有效，由于这是一个比较耗费资源的操作，因此默认为 `false`。

(3) 构造 `HttpRequestResponseHolder` 对象，将 `HttpServletRequest` 和 `HttpServletResponse` 都存储进去。

(4) 调用 `repo.loadContext` 方法去加载 `SecurityContext`，`repo` 实际上就是我们前面所说 `HttpSessionSecurityContextRepository` 的实例，所以 `loadContext` 方法这里就不再赘述了。

(5) 将读取到的 `SecurityContext` 存入 `SecurityContextHolder` 之中，这样，在接下来的处理逻辑中，开发者就可以直接通过 `SecurityContextHolder` 获取当前登录用户对象了。

(6) 调用 `chain.doFilter` 方法使请求继续向下走，但是要注意，此时传递的 `request` 和 `response` 对象是在 `HttpSessionSecurityContextRepository` 中封装后的对象，即 `SaveToSessionResponseWrapper` 和 `SaveToSessionRequestWrapper` 的实例。

(7) 当请求处理完毕后，在 `finally` 模块中，获取最新的 `SecurityContext` 对象（开发者可能在后续处理中修改了 `SecurityContext` 中的 `Authentication` 对象），然后清空 `SecurityContextHolder` 中的数据；再调用 `repo.saveContext` 方法保存 `SecurityContext`，具体的保存逻辑前面已经说过，这里就不再赘述了。

(8) 最后，从 `request` 中移除 `FILTER_APPLIED` 属性。

这就是整个 `SecurityContextPersistenceFilter` 过滤器的工作逻辑。一言以蔽之，请求在到达 `SecurityContextPersistenceFilter` 过滤器之后，先从 `HttpSession` 中读取 `SecurityContext` 出来，并存入 `SecurityContextHolder` 之中以备后续使用；当请求离开 `SecurityContextPersistenceFilter` 过滤器的时候，获取最新的 `SecurityContext` 并存入 `HttpSession` 中，同时清空 `SecurityContextHolder` 中的登录用户信息。

这就是第一种登录数据的获取方式，即从 `SecurityContextHolder` 中获取。



## 2.3.2 从当前请求对象中获取

接下来我们来看一下第二种登录数据获取方式——从当前请求中获取。获取代码如下：

```
@RequestMapping("/authentication")
public void authentication(Authentication authentication) {
    System.out.println("authentication = " + authentication);
}
@RequestMapping("/principal")
public void principal(Principal principal) {
    System.out.println("principal = " + principal);
}
```

开发者可以直接在 **Controller** 的请求参数中放入 **Authentication** 对象来获取登录用户信息。通过前面的讲解，大家已经知道 **Authentication** 是 **Principal** 的子类，所以也可以直接在请求参数中放入 **Principal** 来接收当前登录用户信息。需要注意的是，即使参数是 **Principal**，真正的实例依然是 **Authentication** 的实例。

用过 Spring MVC 的读者都知道，**Controller** 中方法的参数都是当前请求 **HttpServletRequest** 带来的。毫无疑问，前面的 **Authentication** 和 **Principal** 参数也都是 **HttpServletRequest** 带来的，那么这些数据到底是何时放入 **HttpServletRequest** 的呢？又是以何种形式存在的呢？接下来我们一起分析一下。

在 **Servlet** 规范中，最早有三个和安全管理相关的方法：

```
public String getRemoteUser();
public boolean isUserInRole(String role);
public java.security.Principal getUserPrincipal();
```

- (1) **getRemoteUser** 方法用来获取登录用户名。
- (2) **isUserInRole** 方法用来判断当前登录用户是否具备某一个指定的角色。
- (3) **getUserPrincipal** 方法用来获取当前认证主体。

从 **Servlet 3.0** 开始，在这三个方法的基础之上，又增加了三个和安全管理相关的方法：

```
public boolean authenticate(HttpServletRequest response)
    throws IOException, ServletException;
public void login(String username, String password) throws ServletException;
public void logout() throws ServletException;
```

- (1) **authenticate** 方法可以判断当前请求是否认证成功。
- (2) **login** 方法可以执行登录操作。
- (3) **logout** 方法可以执行注销操作。

不过 **HttpServletRequest** 只是一个接口，这些安全认证相关的方法，在不同环境下会有不同的实现。

如果是一个普通的 Web 项目,不使用任何框架,HttpServletRequest 的默认实现类是 Tomcat 中的 RequestFacade,从这个类的名字上就可以看出来,这是一个使用了 Facade 模式(外观模式)的类,真正提供底层服务的是 Tomcat 中的 Request 对象,只不过这个 Request 对象在实现 Servlet 规范的同时,还定义了很多 Tomcat 内部的方法,为了避免开发者直接调用到这些内部方法,这里使用了外观模式。

在 Tomcat 的 Request 类中,对上面这些方法都做了实现,基本上都是基于 Tomcat 提供的 Realm 来实现的,这种认证方式非常冷门,项目中很少使用,因此这里不做过多介绍,感兴趣的读者可以查看 <https://github.com/lenve/javaboy-code-samples> 仓库中的 basiclogin 案例来了解其用法。

如果使用了 Spring Security 框架,那么我们在 Controller 参数中拿到的 HttpServletRequest 实例将是 Servlet3SecurityContextHolderAwareRequestWrapper,很明显,这是被 Spring Security 封装过的请求。

我们来看一下 Servlet3SecurityContextHolderAwareRequestWrapper 的继承关系,如图 2-21 所示。

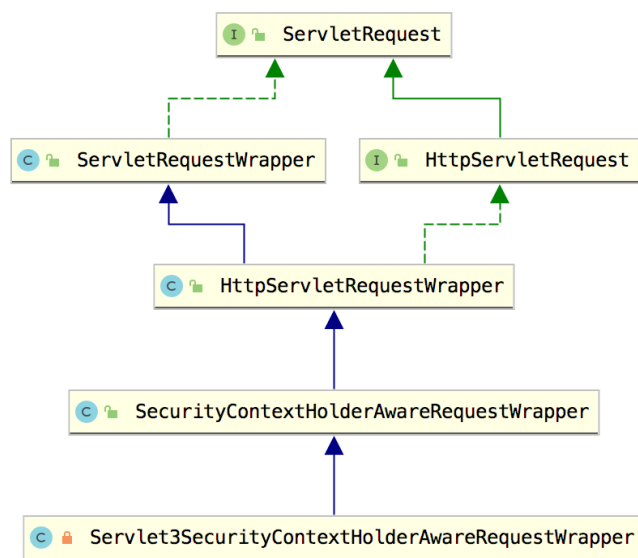


图 2-21 Servlet3SecurityContextHolderAwareRequestWrapper 继承关系图

HttpServletRequestWrapper 就不用过多介绍了, SecurityContextHolderAwareRequestWrapper 类主要实现了 Servlet 3.0 之前和安全管理相关的三个方法,也就是 getRemoteUser()、isUserInRole(String) 以及 getUserPrincipal()。Servlet 3.0 中新增的三个安全管理相关的方法,则在 Servlet3SecurityContextHolderAwareRequestWrapper 类中实现。获取用户登录信息主要和前面三个方法有关,因此这里我们主要来看一下 SecurityContextHolderAwareRequestWrapper 类中相关方法的实现。

```
public class SecurityContextHolderAwareRequestWrapper
    extends HttpServletRequestWrapper {
```

```

private final AuthenticationTrustResolver trustResolver;
private final String rolePrefix;
public SecurityContextHolderAwareRequestWrapper(
    HttpServletRequest request,
    String rolePrefix) {
    this(request, new AuthenticationTrustResolverImpl(), rolePrefix);
}
public SecurityContextHolderAwareRequestWrapper(
    HttpServletRequest request,
    AuthenticationTrustResolver trustResolver, String rolePrefix) {
    super(request);
    this.rolePrefix = rolePrefix;
    this.trustResolver = trustResolver;
}
private Authentication getAuthentication() {
    Authentication auth =
        SecurityContextHolder.getContext().getAuthentication();
    if (!trustResolver.isAnonymous(auth)) {
        return auth;
    }
    return null;
}
@Override
public String getRemoteUser() {
    Authentication auth = getAuthentication();
    if ((auth == null) || (auth.getPrincipal() == null)) {
        return null;
    }
    if (auth.getPrincipal() instanceof UserDetails) {
        return ((UserDetails) auth.getPrincipal()).getUsername();
    }
    return auth.getPrincipal().toString();
}
@Override
public Principal getUserPrincipal() {
    Authentication auth = getAuthentication();
    if ((auth == null) || (auth.getPrincipal() == null)) {
        return null;
    }
    return auth;
}
private boolean isGranted(String role) {
    Authentication auth = getAuthentication();
    if (rolePrefix != null && role != null
        && !role.startsWith(rolePrefix)) {
        role = rolePrefix + role;
    }
    if ((auth == null) || (auth.getPrincipal() == null)) {

```

```

        return false;
    }
    Collection<? extends GrantedAuthority> authorities =
        auth.getAuthorities();

    if (authorities == null) {
        return false;
    }
    for (GrantedAuthority grantedAuthority : authorities) {
        if (role.equals(grantedAuthority.getAuthority())) {
            return true;
        }
    }
    return false;
}
@Override
public boolean isUserInRole(String role) {
    return isGranted(role);
}
}

```

**SecurityContextHolderAwareRequestWrapper** 类其实非常好理解：

(1) **getAuthentication**：该方法用来获取当前登录对象 **Authentication**，获取方式就是我们前面所讲的从 **SecurityContextHolder** 中获取。如果不是匿名对象就返回，否则就返回 **null**。

(2) **getRemoteUser**：该方法返回了当前登录用户的用户名，如果 **Authentication** 对象中存储的 **Principal** 是当前登录用户对象，则返回用户名；如果 **Authentication** 对象中存储的 **Principal** 是当前登录用户名（字符串），则直接返回即可。

(3) **getUserPrincipal**：该方法返回当前登录用户对象，其实就是 **Authentication** 的实例。

(4) **isGranted**：该方法是一个私有方法，作用是判断当前登录用户是否具备某一个指定的角色。判断逻辑也很简单，先对传入进来的角色进行预处理，有的情况下可能需要添加 **ROLE\_**前缀，角色前缀的问题在本书后面的章节中会做详细介绍，这里先不做过多的展开。然后调用 **Authentication#getAuthorities** 方法，获取当前登录用户所具备的所有角色，最后再和传入进来的参数进行比较。

(5) **isUserInRole**：该方法调用 **isGranted** 方法，进而实现判断当前用户是否具备某一个指定角色的功能。

看到这里，相信读者已经明白了，在使用了 **Spring Security** 之后，我们通过 **HttpServletRequest** 就可以获取到很多当前登录用户信息了，代码如下：

```

@RequestMapping("/info")
public void info(HttpServletRequest req) {
    String remoteUser = req.getRemoteUser();
    Authentication auth = ((Authentication) req.getUserPrincipal());
    boolean admin = req.isUserInRole("admin");
    System.out.println("remoteUser = " + remoteUser);
}

```

```

        System.out.println("auth.getName() = " + auth.getName());
        System.out.println("admin = " + admin);
    }

```

执行该方法，打印结果如下：

```

remoteUser = javaboy
auth.getName() = javaboy
admin = false

```

前面我们直接将 `Authentication` 或者 `Principal` 写到 `Controller` 参数中，实际上就是 Spring MVC 框架从 `Servlet3SecurityContextHolderAwareRequestWrapper` 中提取的用户信息。

那么 Spring Security 是如何将默认的请求对象转化为 `Servlet3SecurityContextHolderAwareRequestWrapper` 的呢？这就涉及 Spring Security 过滤器链中另外一个重要的过滤器——`SecurityContextHolderAwareRequestFilter`。

前面我们提到 Spring Security 过滤器中，有一个 `SecurityContextHolderAwareRequestFilter` 过滤器，该过滤器的主要作用就是对 `HttpServletRequest` 请求进行再包装，重写 `HttpServletRequest` 中和安全管理相关的方法。`HttpServletRequest` 在整个请求过程中会被包装多次，每一次的包装都会给它增添新的功能，例如在经过 `SecurityContextPersistenceFilter` 请求时就会对它进行包装。

我们来看一下 `SecurityContextHolderAwareRequestFilter` 过滤器的源码（部分）：

```

public class SecurityContextHolderAwareRequestFilter
    extends GenericFilterBean {
    public void doFilter(ServletRequest req,
        ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
        chain.doFilter(this.requestFactory.create((HttpServletRequest) req,
            (HttpServletRequestResponse) res), res);
    }
    private HttpServletRequestFactory createServlet3Factory(String rolePrefix) {
        HttpServletRequestFactory factory =
            new HttpServletRequest3RequestFactory(rolePrefix);
        factory.setTrustResolver(this.trustResolver);
        factory.setAuthenticationEntryPoint(this.authenticationEntryPoint);
        factory.setAuthenticationManager(this.authenticationManager);
        factory.setLogoutHandlers(this.logoutHandlers);
        return factory;
    }
}
final class HttpServletRequest3RequestFactory implements HttpServletRequestFactory {
    @Override
    public HttpServletRequest create(HttpServletRequest request,
        HttpServletResponse response) {
        return new Servlet3SecurityContextHolderAwareRequestWrapper(request,
            this.rolePrefix, response);
    }
}

```

```
private class Servlet3SecurityContextHolderAwareRequestWrapper
    extends SecurityContextHolderAwareRequestWrapper {
    //.....
}
}
```

从这段源码中可以看到，在 `SecurityContextHolderAwareRequestFilter#doFilter` 方法中，会调用 `requestFactory.create` 方法对请求重新进行包装。`requestFactory` 就是 `HttpServletRequestFactory` 类的实例，它的 `create` 方法里边就直接创建了一个 `Servlet3SecurityContextHolderAwareRequestWrapper` 实例。

对请求的 `HttpServletRequest` 包装之后，接下来在过滤器链中传递的 `HttpServletRequest` 对象，它的 `getRemoteUser()`、`isUserInRole(String)` 以及 `getUserPrincipal()` 方法就可以直接使用了。

`HttpServletRequest` 中 `getUserPrincipal()` 方法有了返回值之后，最终在 Spring MVC 的 `ServletRequestMethodArgumentResolver#resolveArgument(Class<?>, HttpServletRequest)` 方法进行默认参数解析，自动解析出 `Principal` 对象。开发者在 `Controller` 中既可以通过 `Principal` 来接收参数，也可以通过 `Authentication` 对象来接收。

经过前面的介绍，相信读者对于 Spring Security 中两种获取登录用户信息的方式，以及这两种获取方式的原理，都有一定的了解了。

## 2.4 用户定义

在前面的案例中，我们的登录用户是基于配置文件来配置的（本质是基于内存），但是在实际开发中，这种方式肯定是不可取的，在实际项目中，用户信息肯定要存入数据库之中。

Spring Security 支持多种用户定义方式，接下来我们就逐个来看一下这些定义方式。通过前面的介绍（参见 2.1.3 小节），大家对于 `UserDetailsService` 以及它的子类都有了一定的了解，自定义用户其实就是使用 `UserDetailsService` 的不同实现类来提供用户数据，同时将配置好的 `UserDetailsService` 配置给 `AuthenticationManagerBuilder`，系统再将 `UserDetailsService` 提供给 `AuthenticationProvider` 使用。

### 2.4.1 基于内存

前面案例中用户的定义本质上还是基于内存，只是我们没有将 `InMemoryUserDetailsManager` 类明确抽来自定义，现在我们通过自定义 `InMemoryUserDetailsManager` 来看一下基于内存的用户是如何自定义的。

重写 `WebSecurityConfigurerAdapter` 类的 `configure(AuthenticationManagerBuilder)` 方法，内容如下：

```
@Override
```

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
    manager.createUser(User.withUsername("javaboy")
        .password("{noop}123").roles("admin").build());
    manager.createUser(User.withUsername("sang")
        .password("{noop}123").roles("user").build());
    auth.userDetailsService(manager);
}
```

首先构造了一个 `InMemoryUserDetailsManager` 实例，调用该实例的 `createUser` 方法来创建用户对象，我们在这里分别设置了用户名、密码以及用户角色。需要注意的是，用户密码加了一个 `{noop}` 前缀，表示密码不加密，明文存储（关于密码加密问题，会在后面的章节中专门介绍）。

配置完成后，启动项目，此时就可以使用这里配置的两个用户登录了。

`InMemoryUserDetailsManager` 的实现原理很简单，它间接实现了 `UserDetailsService` 接口并重写了它里边的 `loadUserByUsername` 方法，同时它里边维护了一个 `HashMap` 变量，`Map` 的 `key` 就是用户名，`value` 则是用户对象，`createUser` 就是往这个 `Map` 中存储数据，`loadUserByUsername` 方法则是从该 `Map` 中读取数据，这里的源码比较简单，就不贴出来了，读者可以自行查看。

## 2.4.2 基于 JdbcUserDetailsManager

`JdbcUserDetailsManager` 支持将用户数据持久化到数据库，同时它封装了一系列操作用户的方法，例如用户的添加、更新、查找等。

Spring Security 中为 `JdbcUserDetailsManager` 提供了数据库脚本，位置在 `org.springframework.security.core.userdetails.jdbc/users.ddl`，内容如下：

```
create table users(username varchar_ignorecase(50) not null
primary key,
password varchar_ignorecase(500) not null,
enabled boolean not null);

create table authorities (username varchar_ignorecase(50) not null,
authority varchar_ignorecase(50) not null,
constraint fk_authorities_users
foreign key(username) references users(username));

create unique index ix_auth_username on authorities (username,authority);
```

可以看到这里一共创建了两张表，`users` 表就是存放用户信息的表，`authorities` 则是存放用户角色的表。但是大家注意 SQL 的数据类型中有一个 `varchar_ignorecase`，这个其实是针对 `HSQldb` 的数据类型，我们这里使用的是 `MySQL` 数据库，所以这里手动将 `varchar_ignorecase` 类型修改为 `varchar` 类型，然后去数据库中执行修改后的脚本。

另一方面，由于要将数据存入数据库中，所以我们的项目也要提供数据库支持，JdbcUserDetailsManager 底层实际上是使用 JdbcTemplate 来完成的，所以这里主要添加两个依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

然后在 resources/application.properties 中配置数据库连接信息：

```
spring.datasource.username=root
spring.datasource.password=123
spring.datasource.url=jdbc:mysql:///security?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
```

配置完成后，我们重写 WebSecurityConfigurerAdapter 类的 configure(AuthenticationManagerBuilder) 方法，内容如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    DataSource dataSource;
    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        JdbcUserDetailsManager manager =
            new JdbcUserDetailsManager(dataSource);
        if (!manager.userExists("javaboy")) {
            manager.createUser(User.withUsername("javaboy")
                .password("{noop}123").roles("admin").build());
        }
        if (!manager.userExists("sang")) {
            manager.createUser(User.withUsername("sang")
                .password("{noop}123").roles("user").build());
        }
        auth.userDetailsService(manager);
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        //省略
    }
}
```

(1) 当引入 spring-boot-starter-jdbc 并配置了数据库连接信息后，一个 DataSource 实例就



有了，这里首先引入 `DataSource` 实例。

(2) 在 `configure` 方法中，创建一个 `JdbcUserDetailsManager` 实例，在创建时传入 `DataSource` 实例。通过 `userExists` 方法可以判断一个用户是否存在，该方法本质上就是去数据库中查询对应的用户；如果用户不存在，则通过 `createUser` 方法可以创建一个用户，该方法本质上就是向数据库中添加一个用户。

(3) 最后将 `manager` 实例设置到 `auth` 对象中。

配置完成后，重启项目，如果项目启动成功，数据库中就会自动添加进来两条数据，如图 2-22、图 2-23 所示。

authorities	username	password	enabled
users	javaboy	{noop}123	1
	sang	{noop}123	1

图 2-22 数据库中自动存入两条 users 数据

authorities	username	authority
users	javaboy	ROLE_admin
	sang	ROLE_user

图 2-23 数据库中自动存入两条角色数据

此时，我们就可以使用 `javaboy/123`、`sang/123` 进行登录测试了。

在 `JdbcUserDetailsManager` 的继承体系中，首先是 `JdbcDaoImpl` 实现了 `UserDetailsService` 接口，并实现了基本的 `loadUserByUsername` 方法。`JdbcUserDetailsManager` 则继承自 `JdbcDaoImpl`，同时完善了数据库操作，又封装了用户的增删改查方法。这里，我们以 `loadUserByUsername` 为例，看一下源码，其余的增删改操作相对来说都比较容易，这里就不再赘述了。

`JdbcDaoImpl#loadUserByUsername`:

```
@Override
public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException {

    List<UserDetails> users = loadUsersByUsername(username);
    if (users.size() == 0) {
        throw new UsernameNotFoundException(
            this.messages.getMessage("JdbcDaoImpl.notFound",
                new Object[] { username }, "Username {0} not found"));
    }
    UserDetails user = users.get(0);
    Set<GrantedAuthority> dbAuthsSet = new HashSet<>();
    if (this.enableAuthorities) {
        dbAuthsSet.addAll(loadUserAuthorities(user.getUsername()));
    }
    if (this.enableGroups) {
```

```

        dbAuthsSet.addAll(loadGroupAuthorities(user.getUsername()));
    }
    List<GrantedAuthority> dbAuths = new ArrayList<>(dbAuthsSet);
    addCustomAuthorities(user.getUsername(), dbAuths);
    if (dbAuths.size() == 0) {
        throw new UsernameNotFoundException(this.messages.getMessage(
            "JdbcDaoImpl.noAuthority", new Object[] { username },
            "User {0} has no GrantedAuthority"));
    }
    return createUserDetails(username, user, dbAuths);
}
protected List<UserDetails> loadUsersByUsername(String username) {
    return getJdbcTemplate().query(this.usersByUsernameQuery,
        new String[] { username }, (rs, rowNum) -> {
            String username1 = rs.getString(1);
            String password = rs.getString(2);
            boolean enabled = rs.getBoolean(3);
            return new User(username1, password, enabled, true, true, true,
                AuthorityUtils.NO_AUTHORITIES);
        });
}
}

```

(1) 首先根据用户名，调用 `loadUsersByUsername` 方法去数据库中查询用户，查询出来的是一个 `List` 集合，集合中如果没有数据，说明用户不存在，则直接抛出异常。

(2) 如果集合中存在数据，则将集合中的第一条数据拿出来，然后再去查询用户角色，最后根据这些信息创建一个新的 `UserDetails` 出来。

(3) 需要注意的是，这里还引入了分组的概念，不过考虑到 `JdbcUserDetailsManager` 并非我们实际项目中的主流方案，因此这里不做过多介绍。

这就是使用 `JdbcUserDetailsManager` 做数据持久化。这种方式看起来简单，都不用开发者自己写 SQL，但是局限性比较大，无法灵活地定义用户表、角色表等，而在实际开发中，我们还是希望能够灵活地掌控数据表结构，因此 `JdbcUserDetailsManager` 使用场景非常有限。

### 2.4.3 基于 MyBatis

使用 `MyBatis` 做数据持久化是目前大多数企业应用采取的方案，`Spring Security` 中结合 `MyBatis` 可以灵活地定制用户表以及角色表，我们对此进行详细介绍。

首先需要设计三张表，分别是用户表、角色表以及用户角色关联表，三张表的关系如图 2-24 所示。

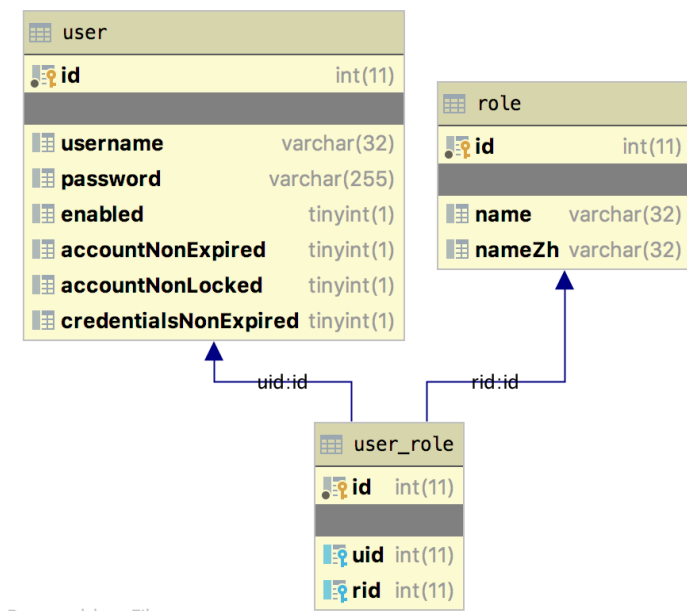


图 2-24 用户表、角色表以及用户角色关联表

用户和角色是多对多的关系，我们使用 `user_role` 来将两者关联起来。  
数据库脚本如下：

```

CREATE TABLE `role` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL,
  `nameZh` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `enabled` tinyint(1) DEFAULT NULL,
  `accountNonExpired` tinyint(1) DEFAULT NULL,
  `accountNonLocked` tinyint(1) DEFAULT NULL,
  `credentialsNonExpired` tinyint(1) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `user_role` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `uid` int(11) DEFAULT NULL,
  `rid` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `uid` (`uid`),
  KEY `rid` (`rid`)
)
  
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

对于角色表，三个字段从上往下含义分别为角色 id、角色英文名称以及角色中文名称。

对于用户表，七个字段从上往下含义依次为：用户 id、用户名、用户密码、账户是否可用、账户是否没有过期、账户是否没有锁定以及凭证（密码）是否没有过期。

数据库创建完成后，可以向数据库中添加几条模拟数据，代码如下：

```
INSERT INTO `role` (`id`, `name`, `nameZh`)
VALUES
  (1, 'ROLE_dba', '数据库管理员'),
  (2, 'ROLE_admin', '系统管理员'),
  (3, 'ROLE_user', '用户');

INSERT INTO `user` (`id`, `username`, `password`, `enabled`,
  `accountNonExpired`, `accountNonLocked`, `credentialsNonExpired`)
VALUES
  (1, 'root', '{noop}123', 1, 1, 1, 1),
  (2, 'admin', '{noop}123', 1, 1, 1, 1),
  (3, 'sang', '{noop}123', 1, 1, 1, 1);

INSERT INTO `user_role` (`id`, `uid`, `rid`)
VALUES
  (1, 1, 1),
  (2, 1, 2),
  (3, 2, 2),
  (4, 3, 3);
```

这样，数据库的准备工作就算完成了。

在 Spring Security 项目中，我们需要引入 MyBatis 和 MySQL 依赖，代码如下：

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.1.3</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

同时在 resources/application.properties 中配置数据库基本连接信息：

```
spring.datasource.username=root
spring.datasource.password=123
spring.datasource.url=jdbc:mysql:///security02?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
```

接下来创建用户类和角色类：

```
public class User implements UserDetails {
```

```
private Integer id;
private String username;
private String password;
private Boolean enabled;
private Boolean accountNonExpired;
private Boolean accountNonLocked;
private Boolean credentialsNonExpired;
private List<Role> roles = new ArrayList<>();
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    List<SimpleGrantedAuthority> authorities = new ArrayList<>();
    for (Role role : roles) {
        authorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return authorities;
}
@Override
public String getPassword() {
    return password;
}
@Override
public String getUsername() {
    return username;
}
@Override
public boolean isAccountNonExpired() {
    return accountNonExpired;
}
@Override
public boolean isAccountNonLocked() {
    return accountNonLocked;
}
@Override
public boolean isCredentialsNonExpired() {
    return credentialsNonExpired;
}
@Override
public boolean isEnabled() {
    return enabled;
}
//省略其他 getter/setter
}

public class Role {
    private Integer id;
    private String name;
    private String nameZh;
    //省略 getter/setter
}
```

自定义用户类需要实现 `UserDetails` 接口，并实现接口中的方法，这些方法的含义我们在 2.1.3 小节中已经介绍过了，这里不再赘述。其中 `roles` 属性用来保存用户所具备的角色信息，由于系统获取用户角色调用的方法是 `getAuthorities`，所以我们在 `getAuthorities` 方法中，将 `roles` 中的角色转为系统可识别的对象并返回。

### 注 意

User 类中的 `isXXX` 方法可以当成 `get` 方法对待，不需要再给这些属性生成 `get` 方法。

接下来我们自定义 `UserDetailsService` 以及对应的数据库查询方法：

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    UserMapper userMapper;
    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        User user = userMapper.loadUserByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("用户不存在");
        }
        user.setRoles(userMapper.getRolesByUid(user.getId()));
        return user;
    }
}

@Mapper
public interface UserMapper {
    List<Role> getRolesByUid(Integer id);
    User loadUserByUsername(String username);
}
```

自定义 `MyUserDetailsService` 实现 `UserDetailsService` 接口，并实现该接口中的方法。`loadUserByUsername` 方法经过前面章节的讲解，相信大家已经很熟悉了，该方法就是根据用户名去数据库中加载用户，如果从数据库中没查到用户，则抛出 `UsernameNotFoundException` 异常；如果查询到用户了，则给用户设置 `roles` 属性。

`UserMapper` 中定义两个方法用于支持 `MyUserDetailsService` 中的查询操作。

最后，在 `UserMapper.xml` 中定义查询 SQL，代码如下：

```
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.javaboy.formlogin.mapper.UserMapper">
    <select id="loadUserByUsername"
            resultType="org.javaboy.formlogin.model.User">
        select * from user where username=#{username};
    </select>
```

```

        <select id="getRolesByUid" resultType="org.javaboy.formlogin.model.Role">
            select r.* from role r,user_role ur where r.`id`=ur.`rid`
        </select>
    </mapper>

```

为了方便，我们将 UserMapper.xml 文件和 UserMapper 接口放在了相同的包下。为了防止 Maven 打包时自动忽略了 XML 文件，还需要在 pom.xml 中添加如下配置：

```

<build>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.xml</include>
            </includes>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
        </resource>
    </resources>
</build>

```

最后一步，就是在 SecurityConfig 中注入 UserDetailsService：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    MyUserDetailsService myUserDetailsService;
    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.userDetailsService(myUserDetailsService);
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            //省略
    }
}

```

配置 UserDetailsService 的方式和前面配置 JdbcUserDetailsManager 的方式基本一致，只不过配置对象变成了 myUserDetailsService 而已。

至此，整个配置工作就完成了。

接下来启动项目，利用数据库中添加的模拟用户进行登录测试，就可以成功登录了，测试方式和前面章节一致，这里不再赘述。

## 2.4.4 基于 Spring Data JPA

考虑到在 Spring Boot 技术栈中也有不少人使用 Spring Data JPA，因此这里针对 Spring Security+Spring Data JPA 也做一个简单介绍，具体思路和基于 MyBatis 的整合类似。

首先引入 Spring Data JPA 的依赖和 MySQL 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>
</dependency>
```

然后在 resources/application.properties 中配置数据库和 JPA，代码如下：

```
spring.datasource.username=root
spring.datasource.password=123
spring.datasource.url=jdbc:mysql:///security03?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai

spring.jpa.database=mysql
spring.jpa.database-platform=mysql
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

数据库的配置还是和以前一样，JPA 的配置则主要配置了数据库平台，数据表更新方式、是否打印 SQL 以及对应的数据库方言。

使用 Spring Data JPA 的好处是我们不用提前准备 SQL 脚本，所以接下来配置两个数据库实体类即可：

```
@Entity(name = "t_user")
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private boolean accountNonExpired;
    private boolean accountNonLocked;
    private boolean credentialsNonExpired;
    private boolean enabled;
    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    private List<Role> roles;
```



```

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    List<SimpleGrantedAuthority> authorities = new ArrayList<>();
    for (Role role : getRoles()) {
        authorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return authorities;
}
@Override
public String getPassword() {
    return password;
}
@Override
public String getUsername() {
    return username;
}
@Override
public boolean isAccountNonExpired() {
    return accountNonExpired;
}
@Override
public boolean isAccountNonLocked() {
    return accountNonLocked;
}
@Override
public boolean isCredentialsNonExpired() {
    return credentialsNonExpired;
}
@Override
public boolean isEnabled() {
    return enabled;
}
//省略 getter/setter
}
@Entity(name = "t_role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String nameZh;
    //省略 getter/setter
}

```

这两个实体类和前面 MyBatis 中实体类的配置类似，需要注意的是 roles 属性上多了一个多对多配置。

接下来配置 UserDetailsService，并提供数据查询方法：

```

@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    UserDao userDao;
    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        User user = userDao.findUserByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("用户不存在");
        }
        return user;
    }
}

public interface UserDao extends JpaRepository<User,Integer> {
    User findUserByUsername(String username);
}

```

`MyUserDetailsService` 的定义也和前面的类似，不同之处在于数据查询方法的变化。定义 `UserDao` 继承自 `JpaRepository`，并定义一个 `findUserByUsername` 方法，剩下的事情 Spring Data JPA 框架会帮我们完成。

最后，再在 `SecurityConfig` 中配置 `MyUserDetailsService`，配置方式和 `MyBatis` 一模一样，这里就不再把代码贴出来了。

使用了 Spring Data JPA 之后，当项目启动时，会自动在数据库中创建相关的表，而不用我们自己去写脚本，这也是使用 Spring Data JPA 的方便之处。

为了测试方便，我们可以在单元测试中执行如下代码，向数据库中添加测试数据：

```

@Autowired
UserDao userDao;
@Test
void contextLoads() {
    User u1 = new User();
    u1.setUsername("javaboy");
    u1.setPassword("{noop}123");
    u1.setAccountNonExpired(true);
    u1.setAccountNonLocked(true);
    u1.setCredentialsNonExpired(true);
    u1.setEnabled(true);
    List<Role> rs1 = new ArrayList<>();
    Role r1 = new Role();
    r1.setName("ROLE_admin");
    r1.setNameZh("管理员");
    rs1.add(r1);
    u1.setRoles(rs1);
    userDao.save(u1);
}

```

测试数据添加成功之后，接下来启动项目，使用测试数据进行登录测试，具体测试过程就不再赘述了。

至此，四种不同的用户定义方式就介绍完了。这四种方式，异曲同工，只是数据存储的方式不一样而已，其他的执行流程都是一样的。

## 2.5 小 结

本章主要介绍了 Spring Security 认证的一些基本操作和原理，对于认证流程做了简单的梳理，登录表单进行了详细配置；同时还研究了获取当前登录用户信息的两种方式以及四种不同的用户定义方式。本章算是一个引子，让大家先来感受一下 Spring Security 的基本用法，接下来的章节中，我们将对这里的诸多登录细节做进一步的深入讲解。

# 第 3 章

---

## 认证流程分析

Spring Security 中默认的一套登录流程是非常完善并且严谨的。但是项目需求非常多样化，很多时候，我们可能还需要对 Spring Security 登录流程进行定制，定制的前提是开发者先深刻理解 Spring Security 登录流程，然后在此基础上，完成对登录流程的定制。本章将从头梳理 Spring Security 登录流程，并通过几个常见的登录定制案例，让读者更加深刻地理解 Spring Security 登录流程。

本章涉及的主要知识点有：

- 登录流程分析。
- 配置多个数据源。
- 添加登录验证码。

### 3.1 登录流程分析

要搞清楚 Spring Security 认证流程，我们得先认识与之相关的三个基本组件（Authentication 对象在第 2 章已经做过介绍，这里不再赘述）：AuthenticationManager、ProviderManager 以及 AuthenticationProvider，同时还要去了解接入认证功能的过滤器 AbstractAuthenticationProcessingFilter，这四个类搞明白了，基本上认证流程也就清楚了，下面我们逐个分析一下。

#### 3.1.1 AuthenticationManager

从名称上可以看出，AuthenticationManager 是一个认证管理器，它定义了 Spring Security

过滤器要如何执行认证操作。`AuthenticationManager` 在认证成功后，会返回一个 `Authentication` 对象，这个 `Authentication` 对象会被设置到 `SecurityContextHolder` 中。如果开发者不想用 Spring Security 提供的一套认证机制，那么也可以自定义认证流程，认证成功后，手动将 `Authentication` 存入 `SecurityContextHolder` 中。

```
public interface AuthenticationManager {
    Authentication authenticate(Authentication authentication)
        throws AuthenticationException;
}
```

从 `AuthenticationManager` 的源码中可以看到，`AuthenticationManager` 对传入的 `Authentication` 对象进行身份认证，此时传入的 `Authentication` 参数只有用户名/密码等简单的属性，如果认证成功，返回的 `Authentication` 的属性会得到完全填充，包括用户所具备的角色信息。

`AuthenticationManager` 是一个接口，它有着诸多的实现类，开发者也可以自定义 `AuthenticationManager` 的实现类，不过在实际应用中，我们使用最多的是 `ProviderManager`。在 Spring Security 框架中，默认也是使用 `ProviderManager`。

### 3.1.2 AuthenticationProvider

2.3 节介绍了 Spring Security 支持多种不同的认证方式，不同的认证方式对应不同的身份类型，`AuthenticationProvider` 就是针对不同的身份类型执行具体的身份认证。例如，常见的 `DaoAuthenticationProvider` 用来支持用户名/密码登录认证，`RememberMeAuthenticationProvider` 用来支持“记住我”的认证。

`AuthenticationProvider` 的源码如下：

```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication)
        throws AuthenticationException;
    boolean supports(Class<?> authentication);
}
```

- (1) `authenticate` 方法用来执行具体的身份认证。
- (2) `supports` 方法用来判断当前 `AuthenticationProvider` 是否支持对应的身份类型。

当使用用户名/密码的方式登录时，对应的 `AuthenticationProvider` 实现类是 `DaoAuthenticationProvider`，而 `DaoAuthenticationProvider` 继承自 `AbstractUserDetailsAuthenticationProvider` 并且没有重写 `authenticate` 方法，所以具体的认证逻辑在 `AbstractUserDetailsAuthenticationProvider` 的 `authenticate` 方法中。我们就从 `AbstractUserDetailsAuthenticationProvider` 开始看起：

```
public abstract class AbstractUserDetailsAuthenticationProvider implements
    AuthenticationProvider, InitializingBean, MessageSourceAware {
```

```

private UserCache userCache = new NullUserCache();
private boolean forcePrincipalAsString = false;
protected boolean hideUserNotFoundExceptions = true;
private UserDetailsChecker preAuthenticationChecks =
    new DefaultPreAuthenticationChecks();
private UserDetailsChecker postAuthenticationChecks =
    new DefaultPostAuthenticationChecks();
protected abstract void additionalAuthenticationChecks(
    UserDetails userDetails,
    UsernamePasswordAuthenticationToken authentication)
    throws AuthenticationException;
public Authentication authenticate(Authentication authentication)
    throws AuthenticationException {
    String username =
        (authentication.getPrincipal() == null) ? "NONE_PROVIDED"
        : authentication.getName();

    boolean cacheWasUsed = true;
    UserDetails user = this.userCache.getUserFromCache(username);
    if (user == null) {
        cacheWasUsed = false;
        try {
            user = retrieveUser(username,
                (UsernamePasswordAuthenticationToken) authentication);
        }
        catch (UsernameNotFoundException notFound) {
            if (hideUserNotFoundExceptions) {
                throw new BadCredentialsException(messages.getMessage(
                    "AbstractUserDetailsAuthenticationProvider.badCredentials",
                    "Bad credentials"));
            }
            else {
                throw notFound;
            }
        }
    }
    try {
        preAuthenticationChecks.check(user);
        additionalAuthenticationChecks(user,
            (UsernamePasswordAuthenticationToken) authentication);
    }
    catch (AuthenticationException exception) {
        if (cacheWasUsed) {
            cacheWasUsed = false;
            user = retrieveUser(username,
                (UsernamePasswordAuthenticationToken)
                    authentication);
            preAuthenticationChecks.check(user);
            additionalAuthenticationChecks(user,

```

```

        (UsernamePasswordAuthenticationToken) authentication);
    }
    else {
        throw exception;
    }
}
postAuthenticationChecks.check(user);
if (!cacheWasUsed) {
    this.userCache.putUserInCache(user);
}
Object principalToReturn = user;
if (forcePrincipalAsString) {
    principalToReturn = user.getUsername();
}
return createSuccessAuthentication(principalToReturn,
                                   authentication, user);
}
protected Authentication createSuccessAuthentication(Object principal,
    Authentication authentication, UserDetails user) {
    UsernamePasswordAuthenticationToken result =
        new UsernamePasswordAuthenticationToken(
            principal, authentication.getCredentials(),
            authoritiesMapper.mapAuthorities(user.getAuthorities()));
    result.setDetails(authentication.getDetails());
    return result;
}
protected abstract UserDetails retrieveUser(String username,
    UsernamePasswordAuthenticationToken authentication)
    throws AuthenticationException;
public boolean supports(Class<?> authentication) {
    return (UsernamePasswordAuthenticationToken.class
        .isAssignableFrom(authentication));
}
private class DefaultPreAuthenticationChecks implements UserDetailsChecker {
    public void check(UserDetails user) {
        if (!user.isAccountNonLocked()) {
            throw new LockedException(messages.getMessage(
                "AbstractUserDetailsAuthenticationProvider.locked",
                "User account is locked"));
        }
        if (!user.isEnabled()) {
            throw new DisabledException(messages.getMessage(
                "AbstractUserDetailsAuthenticationProvider.disabled",
                "User is disabled"));
        }
        if (!user.isAccountNonExpired()) {
            throw new AccountExpiredException(messages.getMessage(
                "AbstractUserDetailsAuthenticationProvider.expired",

```

```

        "User account has expired"));
    }
}
private class DefaultPostAuthenticationChecks implements
    UserDetailsChecker {
    public void check(UserDetails user) {
        if (!user.isCredentialsNonExpired()) {
            throw new CredentialsExpiredException(messages.getMessage(
                "AbstractUserDetailsAuthenticationProvider.credentialsExpired",
                "User credentials have expired"));
        }
    }
}
}
}

```

`AbstractUserDetailsAuthenticationProvider` 是一个抽象类，抽象方法在它的实现类 `DaoAuthenticationProvider` 中完成。`AbstractUserDetailsAuthenticationProvider` 本身逻辑很简单，我们一起来看一下：

(1) 一开始先声明一个用户缓存对象 `userCache`，默认情况下没有启用缓存对象。

(2) `hideUserNotFoundExceptions` 表示是否隐藏用户名查找失败的异常，默认为 `true`。为了确保系统安全，用户在登录失败时只会给出一个模糊提示，例如“用户名或密码输入错误”。在 `Spring Security` 内部，如果用户名查找失败，则会抛出 `UsernameNotFoundException` 异常，但是该异常会被自动隐藏，转而通过一个 `BadCredentialsException` 异常来代替它，这样，开发者在处理登录失败异常时，无论是用户名输入错误还是密码输入错误，收到的总是 `BadCredentialsException`，这样做的一个好处是可以避免新手程序员将用户名输入错误和密码输入错误两个异常分开提示。

(3) `forcePrincipalAsString` 表示是否强制将 `Principal` 对象当成字符串来处理，默认是 `false`。`Authentication` 中的 `principal` 属性类型是一个 `Object`，正常来说，通过 `principal` 属性可以获取到当前登录用户对象（即 `UserDetails`），但是如果 `forcePrincipalAsString` 设置为 `true`，则 `Authentication` 中的 `principal` 属性返回就是当前登录用户名，而不是用户对象。

(4) `preAuthenticationChecks` 对象则是用于做用户状态检查，在用户认证过程中，需要检验用户状态是否正常，例如账户是否被锁定、账户是否可用、账户是否过期等。

(5) `postAuthenticationChecks` 对象主要负责在密码校验成功后，检查密码是否过期。

(6) `additionalAuthenticationChecks` 是一个抽象方法，主要就是校验密码，具体的实现在 `DaoAuthenticationProvider` 中。

(7) `authenticate` 方法就是核心的校验方法了。在方法中，首先从登录数据中获取用户名，然后根据用户名去缓存中查询用户对象，如果查询不到，则根据用户名调用 `retrieveUser` 方法从数据库中加载用户；如果没有加载到用户，则抛出异常（用户不存在异常会被隐藏）。拿到用户对象之后，首先调用 `preAuthenticationChecks.check` 方法进行用户状态检查，然后调用



`additionalAuthenticationChecks` 方法进行密码的校验操作，最后调用 `postAuthenticationChecks.check` 方法检查密码是否过期，当所有步骤都顺利完成后，调用 `createSuccessAuthentication` 方法创建一个认证后的 `UsernamePasswordAuthenticationToken` 对象并返回，认证后的对象中包含了认证主体、凭证以及角色等信息。

这就是 `AbstractUserDetailsAuthenticationProvider` 类的工作流程，有几个抽象方法是在 `DaoAuthenticationProvider` 中实现的，我们再来看一下 `DaoAuthenticationProvider` 中的定义：

```
public class DaoAuthenticationProvider extends
    AbstractUserDetailsAuthenticationProvider {
    private static final String USER_NOT_FOUND_PASSWORD =
        "userNotFoundPassword";

    private PasswordEncoder passwordEncoder;
    private volatile String userNotFoundEncodedPassword;
    private UserDetailsService userDetailsService;
    private UserDetailsPasswordService userDetailsPasswordService;
    public DaoAuthenticationProvider() {
        setPasswordEncoder(PasswordEncoderFactories.
            createDelegatingPasswordEncoder());
    }
    protected void additionalAuthenticationChecks(UserDetails userDetails,
        UsernamePasswordAuthenticationToken authentication)
        throws AuthenticationException {
        if (authentication.getCredentials() == null) {
            throw new BadCredentialsException(messages.getMessage(
                "AbstractUserDetailsAuthenticationProvider.badCredentials",
                "Bad credentials"));
        }
        String presentedPassword = authentication.getCredentials().toString();
        if (!passwordEncoder
            .matches(presentedPassword, userDetails.getPassword())) {
            throw new BadCredentialsException(messages.getMessage(
                "AbstractUserDetailsAuthenticationProvider.badCredentials",
                "Bad credentials"));
        }
    }
    protected final UserDetails retrieveUser(String username,
        UsernamePasswordAuthenticationToken authentication)
        throws AuthenticationException {
        prepareTimingAttackProtection();
        try {
            UserDetails loadedUser =
                this.getUserDetailsService().loadUserByUsername(username);
            if (loadedUser == null) {
                throw new InternalAuthenticationServiceException(
                    "UserDetailsService returned null, which is an interface
                    contract violation");
            }
        }
    }
}
```

```

        }
        return loadedUser;
    }
    catch (UsernameNotFoundException ex) {
        mitigateAgainstTimingAttack(authentication);
        throw ex;
    }
    catch (InternalAuthenticationServiceException ex) {
        throw ex;
    }
    catch (Exception ex) {
        throw new InternalAuthenticationServiceException(ex.getMessage(),
                                                         ex);
    }
}
@Override
protected Authentication createSuccessAuthentication(Object principal,
                                                     Authentication authentication, UserDetails user) {
    boolean upgradeEncoding = this.userDetailsPasswordService != null
        && this.passwordEncoder.upgradeEncoding(user.getPassword());
    if (upgradeEncoding) {
        String presentedPassword =
            authentication.getCredentials().toString();
        String newPassword =
            this.passwordEncoder.encode(presentedPassword);
        user = this.userDetailsPasswordService
            .updatePassword(user, newPassword);
    }
    return super
        .createSuccessAuthentication(principal, authentication, user);
}
private void prepareTimingAttackProtection() {
    if (this.userNotFoundEncodedPassword == null) {
        this.userNotFoundEncodedPassword =
            this.passwordEncoder.encode(USER_NOT_FOUND_PASSWORD);
    }
}
private void mitigateAgainstTimingAttack(
    UsernamePasswordAuthenticationToken authentication) {
    if (authentication.getCredentials() != null) {
        String presentedPassword =
            authentication.getCredentials().toString();
        this.passwordEncoder.matches(presentedPassword,
                                     this.userNotFoundEncodedPassword);
    }
}
}
}

```

在 DaoAuthenticationProvider 中：

(1) 首先定义了 USER\_NOT\_FOUND\_PASSWORD 常量，这个是当用户查找失败时的默认密码；passwordEncoder 是一个密码加密和比对工具，这个在第5章会有专门的介绍，这里先不做过多解释；userNotFoundEncodedPassword 变量则用来保存默认密码加密后的值；userDetailsService 是一个用户查找工具，userDetailsService 在第2章已经讲过，这里不再赘述；userDetailsPasswordService 则用来提供密码修改服务。

(2) 在 DaoAuthenticationProvider 的构造方法中，默认就会指定 PasswordEncoder，当然开发者也可以通过 set 方法自定义 PasswordEncoder。

(3) additionalAuthenticationChecks 方法主要进行密码校验，该方法第一个参数 userDetails 是从数据库中查询出来的用户对象，第二个参数 authentication 则是登录用户输入的参数。从这两个参数中分别提取出来用户密码，然后调用 passwordEncoder.matches 方法进行密码比对。

(4) retrieveUser 方法则是获取用户对象的方法，具体做法就是调用 UserDetailsServiceImpl#loadUserByUsername 方法去数据库中查询。

(5) 在 retrieveUser 方法中，有一个值得关注的地方。在该方法一开始，首先会调用 prepareTimingAttackProtection 方法，该方法的作用是使用 PasswordEncoder 对常量 USER\_NOT\_FOUND\_PASSWORD 进行加密，将加密结果保存在 userNotFoundEncodedPassword 变量中。当根据用户名查找用户时，如果抛出了 UsernameNotFoundException 异常，则调用 mitigateAgainstTimingAttack 方法进行密码比对。有读者会说，用户都没查找到，怎么比对密码？需要注意，在调用 mitigateAgainstTimingAttack 方法进行密码比对时，使用了 userNotFoundEncodedPassword 变量作为默认密码和登录请求传来的用户密码进行比对。这是一个一开始就注定要失败的密码比对，那么为什么还要进行比对呢？这主要是为了避免旁道攻击 (Side-channel attack)。如果根据用户名查找用户失败，就直接抛出异常而不进行密码比对，那么黑客经过大量的测试，就会发现有的请求耗费时间明显小于其他请求，那么进而可以得出该请求的用户名是一个不存在的用户名 (因为用户名不存在，所以不需要密码比对，进而节省时间)，这样就可以获取到系统信息。为了避免这一问题，所以当用户查找失败时，也会调用 mitigateAgainstTimingAttack 方法进行密码比对，这样就可以迷惑黑客。

(6) createSuccessAuthentication 方法则是在登录成功后，创建一个全新的 UsernamePasswordAuthenticationToken 对象，同时会判断是否需要进行密码升级，如果需要进行密码升级，就会在该方法中进行加密方案升级。

通过对 AbstractUserDetailsAuthenticationProvider 和 DaoAuthenticationProvider 的讲解，相信读者已经很明白 AuthenticationProvider 中的认证逻辑了。

### 提示

在密码学中，旁道攻击 (Side-channel attack) 又称侧信道攻击、边信道攻击。这种攻击方式不是暴力破解或者是研究加密算法的弱点。它是基于从密码系统的物理实现中获取信息，比如时间、功率消耗、电磁泄漏等，这些信息可被利用于对系统的进一步破解。

### 3.1.3 ProviderManager

ProviderManager 是 AuthenticationManager 的一个重要实现类。在开始学习之前，我们先通过一幅图来了解一下 ProviderManager 和 AuthenticationProvider 之间的关系，如图 3-1 所示。

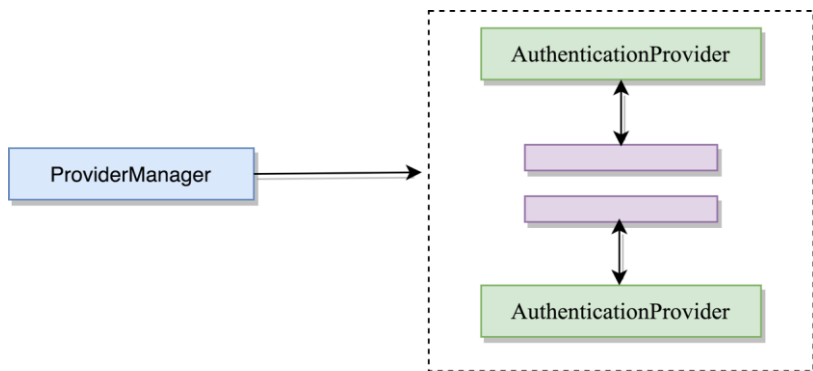


图 3-1 ProviderManager 和 AuthenticationProvider 的关系

在 Spring Security 中，由于系统可能同时支持多种不同的认证方式，例如同时支持用户名/密码认证、RememberMe 认证、手机号码动态认证等，而不同的认证方式对应了不同的 AuthenticationProvider，所以一个完整的认证流程可能由多个 AuthenticationProvider 来提供。

多个 AuthenticationProvider 将组成一个列表，这个列表将由 ProviderManager 代理。换句话说，在 ProviderManager 中存在一个 AuthenticationProvider 列表，在 ProviderManager 中遍历列表中的每一个 AuthenticationProvider 去执行身份认证，最终得到认证结果。

ProviderManager 本身也可以再配置一个 AuthenticationManager 作为 parent，这样当 ProviderManager 认证失败之后，就可以进入到 parent 中再次进行认证。理论上来说，ProviderManager 的 parent 可以是任意类型的 AuthenticationManager，但是通常都是由 ProviderManager 来扮演 parent 的角色，也就是 ProviderManager 是 ProviderManager 的 parent。

ProviderManager 本身也可以有多个，多个 ProviderManager 共用同一个 parent，当存在多个过滤器链的时候非常有用。当存在多个过滤器链时，不同的路径可能对应不同的认证方式，但是不同路径可能又会同时存在一些共有的认证方式，这些共有的认证方式可以在 parent 中统一处理。

根据上面的介绍，我们绘出新的 ProviderManager 和 AuthenticationProvider 关系图，如图 3-2 所示。

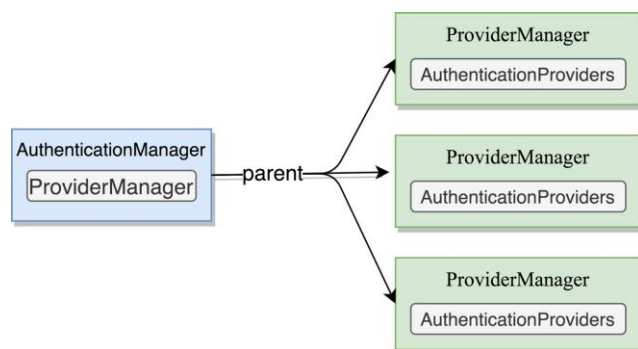


图 3-2 ProviderManager 中包含多个 AuthenticationProvider，并共享同一个 parent

我们重点看一下 ProviderManager 中的 authenticate 方法：

```

public Authentication authenticate(Authentication authentication)
    throws AuthenticationException {
    Class<? extends Authentication> toTest = authentication.getClass();
    AuthenticationException lastException = null;
    AuthenticationException parentException = null;
    Authentication result = null;
    Authentication parentResult = null;
    for (AuthenticationProvider provider : getProviders()) {
        if (!provider.supports(toTest)) {
            continue;
        }
        try {
            result = provider.authenticate(authentication);
            if (result != null) {
                copyDetails(authentication, result);
                break;
            }
        }
        catch (AccountStatusException |
                InternalAuthenticationServiceException e) {
            prepareException(e, authentication);
            throw e;
        }
        catch (AuthenticationException e) {
            lastException = e;
        }
    }
    if (result == null && parent != null) {
        try {
            result = parentResult = parent.authenticate(authentication);
        }
        catch (ProviderNotFoundException e) {
        }
        catch (AuthenticationException e) {
            lastException = parentException = e;
        }
    }
}

```

```
    }  
    }  
    if (result != null) {  
        if (eraseCredentialsAfterAuthentication  
            && (result instanceof CredentialsContainer)) {  
            ((CredentialsContainer) result).eraseCredentials();  
        }  
        if (parentResult == null) {  
            eventPublisher.publishAuthenticationSuccess(result);  
        }  
        return result;  
    }  
    if (lastException == null) {  
        lastException = new ProviderNotFoundException(messages.getMessage(  
            "ProviderManager.providerNotFound",  
            new Object[] { toTest.getName() },  
            "No AuthenticationProvider found for {0}"));  
    }  
    if (parentException == null) {  
        prepareException(lastException, authentication);  
    }  
    throw lastException;  
}
```

这段源码的逻辑还是非常清晰的，我们分析一下：

- (1) 首先获取 `authentication` 对象的类型。
- (2) 分别定义当前认证过程抛出的异常、`parent` 中认证时抛出的异常、当前认证结果以及 `parent` 中认证结果对应的变量。
- (3) `getProviders` 方法用来获取当前 `ProviderManager` 所代理的所有 `AuthenticationProvider` 对象，遍历这些 `AuthenticationProvider` 对象进行身份认证。
- (4) 判断当前 `AuthenticationProvider` 是否支持当前 `Authentication` 对象，要是不支持，则继续处理列表中的下一个 `AuthenticationProvider` 对象。
- (5) 调用 `provider.authenticate` 方法进行身份认证，如果认证成功，返回认证后的 `Authentication` 对象，同时调用 `copyDetails` 方法给 `Authentication` 对象的 `details` 属性赋值。由于可能是多个 `AuthenticationProvider` 执行认证操作，所以如果抛出异常，则通过 `lastException` 变量来记录。
- (6) 在 `for` 循环执行完成后，如果 `result` 还是没有值，说明所有的 `AuthenticationProvider` 都认证失败，此时如果 `parent` 不为空，则调用 `parent` 的 `authenticate` 方法进行认证。
- (7) 接下来，如果 `result` 不为空，就将 `result` 中的凭证擦除，防止泄漏。如果使用了用户名/密码的方式登录，那么所谓的擦除实际上就是将密码字段设置为 `null`，同时将登录成功的事件发布出去（发布登录成功事件需要 `parentResult` 为 `null`。如果 `parentResult` 不为 `null`，表示在 `parent` 中已经认证成功，认证成功的事件也已经在 `parent` 中发布出去了，这样会导致认证

成功的事件重复发布)。如果用户认证成功,此时就将 `result` 返回,后面的代码也就不再执行了。

(8) 如果前面没能返回 `result`,说明认证失败。如果 `lastException` 为 `null`,说明 `parent` 为 `null` 或者没有认证亦或者认证失败了但是没有抛出异常,此时构造 `ProviderNotFoundException` 异常赋值给 `lastException`。

(9) 如果 `parentException` 为 `null`,发布认证失败事件(如果 `parentException` 不为 `null`,则说明认证失败事件已经发布过了)。

(10) 最后抛出 `lastException` 异常。

这就是 `ProviderManager` 中 `authenticate` 方法的身份认证逻辑,其他方法的源码要相对简单很多,在这里就不一一解释了。

现在,大家已经熟悉了 `Authentication`、`AuthenticationManager`、`AuthenticationProvider` 以及 `ProviderManager` 的工作原理了,接下来的问题就是这些组件如何跟登录关联起来?这就涉及一个重要的过滤器——`AbstractAuthenticationProcessingFilter`。

### 3.1.4 AbstractAuthenticationProcessingFilter

作为 Spring Security 过滤器链中的一环,`AbstractAuthenticationProcessingFilter` 可以用来处理任何提交给它的身份认证,图 3-3 描述了 `AbstractAuthenticationProcessingFilter` 的工作流程。

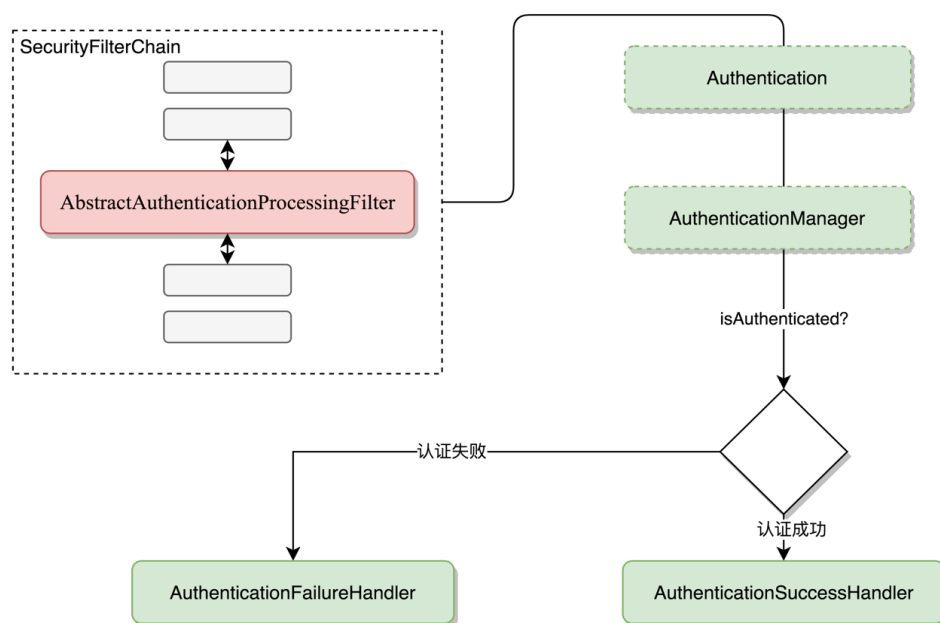


图 3-3 `AbstractAuthenticationProcessingFilter` 所处的位置

图中显示的流程是一个通用的架构。

`AbstractAuthenticationProcessingFilter` 作为一个抽象类,如果使用用户名/密码的方式登录,那么它对应的实现类是 `UsernamePasswordAuthenticationFilter`,构造出来的 `Authentication` 对象

则是 `UsernamePasswordAuthenticationToken`。至于 `AuthenticationManager`，前面已经说过，一般情况下它的实现类就是 `ProviderManager`，这里在 `ProviderManager` 中进行认证，认证成功就会进入认证成功的回调，否则进入认证失败的回调。因此，我们可以对上面的流程图再做进一步细化，如图 3-4 所示。

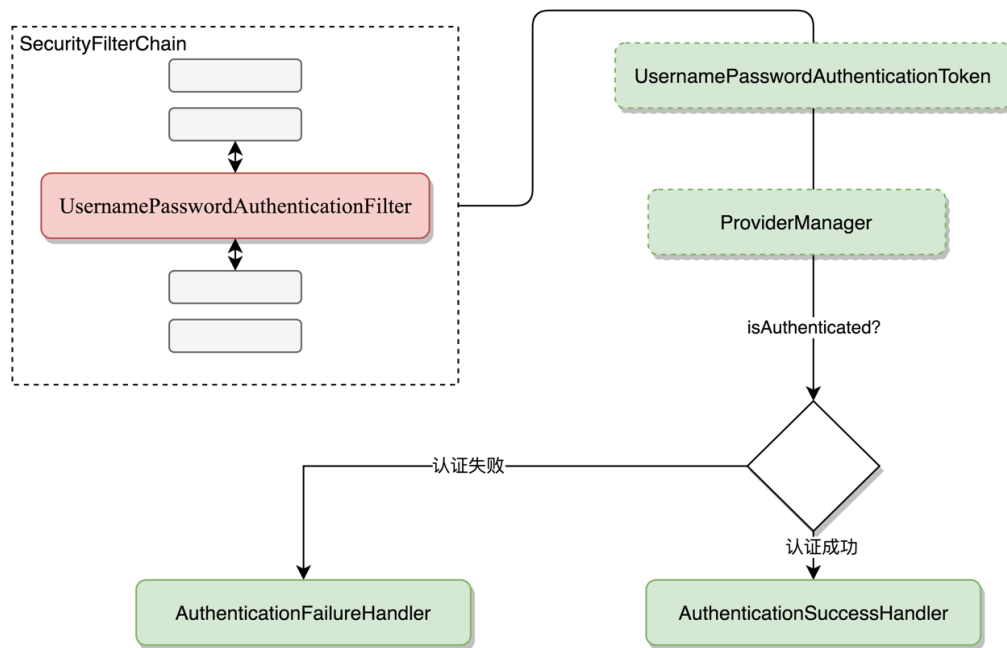


图 3-4 使用用户名/密码进行身份认证流程图

前面第 2 章中所涉及的认证流程基本上就是这样，我们来大致梳理一下：

(1) 当用户提交登录请求时，`UsernamePasswordAuthenticationFilter` 会从当前请求 `HttpServletRequest` 中提取出登录用户名/密码，然后创建出一个 `UsernamePasswordAuthenticationToken` 对象。

(2) `UsernamePasswordAuthenticationToken` 对象将被传入 `ProviderManager` 中进行具体的认证操作。

(3) 如果认证失败，则 `SecurityContextHolder` 中相关信息将被清除，登录失败回调也会被调用。

(4) 如果认证成功，则会进行登录信息存储、Session 并发处理、登录成功事件发布以及登录成功方法回调等操作。

这是认证的一个大致流程。接下来我们结合 `AbstractAuthenticationProcessingFilter` 和 `UsernamePasswordAuthenticationFilter` 的源码来看一下。

先来看 `AbstractAuthenticationProcessingFilter` 源码（部分核心代码）：

```
public abstract class AbstractAuthenticationProcessingFilter
    extends GenericFilterBean
```



```

implements ApplicationEventPublisherAware, MessageSourceAware {
public void doFilter(ServletRequest req,
                    ServletResponse res, FilterChain chain)
                    throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;
    if (!requiresAuthentication(request, response)) {
        chain.doFilter(request, response);
        return;
    }
    Authentication authResult;
    try {
        authResult = attemptAuthentication(request, response);
        if (authResult == null) {
            return;
        }
        sessionStrategy.onAuthentication(authResult, request, response);
    }
    catch (InternalAuthenticationServiceException failed) {
        unsuccessfulAuthentication(request, response, failed);
        return;
    }
    catch (AuthenticationException failed) {
        unsuccessfulAuthentication(request, response, failed);
        return;
    }
    if (continueChainBeforeSuccessfulAuthentication) {
        chain.doFilter(request, response);
    }
    successfulAuthentication(request, response, chain, authResult);
}
protected boolean requiresAuthentication(HttpServletRequest request,
                                         HttpServletResponse response) {
    return requiresAuthenticationRequestMatcher.matches(request);
}
public abstract Authentication attemptAuthentication(
                                         HttpServletRequest request,
                                         HttpServletResponse response)
                                         throws AuthenticationException, IOException, ServletException;
protected void successfulAuthentication(HttpServletRequest request,
                                         HttpServletResponse response, FilterChain chain,
                                         Authentication authResult)
                                         throws IOException, ServletException {
    SecurityContextHolder.getContext().setAuthentication(authResult);
    rememberMeServices.loginSuccess(request, response, authResult);
    if (this.eventPublisher != null) {
        eventPublisher
            .publishEvent(new InteractiveAuthenticationSuccessEvent(

```

```

        authResult, this.getClass()));
    }
    successHandler.onAuthenticationSuccess(request, response, authResult);
}
protected void unsuccessfulAuthentication(HttpServletRequest request,
    HttpServletResponse response, AuthenticationException failed)
    throws IOException, ServletException {
    SecurityContextHolder.clearContext();
    rememberMeServices.loginFail(request, response);
    failureHandler.onAuthenticationFailure(request, response, failed);
}
}
}

```

(1) 首先通过 `requiresAuthentication` 方法来判断当前请求是不是登录认证请求，如果是认证请求，就执行接下来的认证代码；如果不是认证请求，则直接继续走剩余的过滤器即可。

(2) 调用 `attemptAuthentication` 方法来获取一个经过认证后的 `Authentication` 对象，`attemptAuthentication` 方法是一个抽象方法，具体实现在它的子类 `UsernamePasswordAuthenticationFilter` 中。

(3) 认证成功后，通过 `sessionStrategy.onAuthentication` 方法来处理 session 并发问题。

(4) `continueChainBeforeSuccessfulAuthentication` 变量用来判断请求是否还需要继续向下走。默认情况下该参数的值为 `false`，即认证成功后，后续的过滤器将不再执行了。

(5) `unsuccessfulAuthentication` 方法用来处理认证失败事宜，主要做了三件事：①从 `SecurityContextHolder` 中清除数据；②清除 Cookie 等信息；③调用认证失败的回调方法。

(6) `successfulAuthentication` 方法主要用来处理认证成功事宜，主要做了四件事：①向 `SecurityContextHolder` 中存入用户信息；②处理 Cookie；③发布认证成功事件，这个事件类型是 `InteractiveAuthenticationSuccessEvent`，表示通过一些自动交互的方式认证成功，例如通过 `RememberMe` 的方式登录；④调用认证成功的回调方法。

这就是 `AbstractAuthenticationProcessingFilter` 大致上所做的事情，还有一个抽象方法 `attemptAuthentication` 是在它的继承类 `UsernamePasswordAuthenticationFilter` 中实现的，接下来我们来看一下 `UsernamePasswordAuthenticationFilter` 类：

```

public class UsernamePasswordAuthenticationFilter extends
    AbstractAuthenticationProcessingFilter {
    public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "username";
    public static final String SPRING_SECURITY_FORM_PASSWORD_KEY = "password";
    private String usernameParameter = SPRING_SECURITY_FORM_USERNAME_KEY;
    private String passwordParameter = SPRING_SECURITY_FORM_PASSWORD_KEY;
    private boolean postOnly = true;
    public UsernamePasswordAuthenticationFilter() {
        super(new AntPathRequestMatcher("/login", "POST"));
    }
    public Authentication attemptAuthentication(HttpServletRequest request,
        HttpServletResponse response) throws AuthenticationException {

```

```

        if (postOnly && !request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException(
                "Authentication method not supported: "
                    + request.getMethod());
        }
        String username = obtainUsername(request);
        String password = obtainPassword(request);
        if (username == null) {
            username = "";
        }
        if (password == null) {
            password = "";
        }
        username = username.trim();
        UsernamePasswordAuthenticationToken authRequest =
            new UsernamePasswordAuthenticationToken(username, password);
        setDetails(request, authRequest);
        return this.getAuthenticationManager().authenticate(authRequest);
    }

    @Nullable
    protected String obtainPassword(HttpServletRequest request) {
        return request.getParameter(passwordParameter);
    }

    @Nullable
    protected String obtainUsername(HttpServletRequest request) {
        return request.getParameter(usernameParameter);
    }

    protected void setDetails(HttpServletRequest request,
        UsernamePasswordAuthenticationToken authRequest) {
        authRequest.setDetails(authenticationDetailsSource.buildDetails(request));
    }

    public void setUsernameParameter(String usernameParameter) {
        this.usernameParameter = usernameParameter;
    }

    public void setPasswordParameter(String passwordParameter) {
        this.passwordParameter = passwordParameter;
    }

    public void setPostOnly(boolean postOnly) {
        this.postOnly = postOnly;
    }

    public final String getUsernameParameter() {
        return usernameParameter;
    }

    public final String getPasswordParameter() {
        return passwordParameter;
    }
}

```

(1) 首先声明了默认情况下登录表单的用户名字段和密码字段，用户名字段的 key 默认是 username，密码字段的 key 默认是 password。当然，这两个字段也可以自定义，自定义的方式就是我们在 SecurityConfig 中配置的 .usernameParameter("uname") 和 .passwordParameter("passwd")（参考 2.2 节）。

(2) 在 UsernamePasswordAuthenticationFilter 过滤器构建的时候，指定了当前过滤器只用来处理登录请求，默认的登录请求是/login，当然开发者也可以自行配置。

(3) 接下来就是最重要的 attemptAuthentication 方法了，在该方法中，首先确认请求是 post 类型；然后通过 obtainUsername 和 obtainPassword 方法分别从请求中提取出用户名和密码，具体的提取过程就是调用 request.getParameter 方法；拿到登录请求传来的用户名/密码之后，构造出一个 authRequest，然后调用 getAuthenticationManager().authenticate 方法进行认证，这就进入到我们前面所说的 ProviderManager 的流程中了，具体认证过程就不再赘述了。

以上就是整个认证流程。

搞懂了认证流程，那么接下来如果想要自定义一些认证方式，就会非常容易了，比如定义多个数据源、添加登录校验码等。下面，我们将通过两个案例，来活学活用上面所讲的认证流程。

## 3.2 配置多个数据源

多个数据源是指在同一个系统中，用户数据来自不同的表，在认证时，如果第一张表没有查找到用户，那就去第二张表中查询，依次类推。

看了前面的分析，要实现这个需求就很容易了。认证要经过 AuthenticationProvider，每一个 AuthenticationProvider 中都配置了一个 UserDetailsService，而不同的 UserDetailsService 则可以代表不同的数据源。所以我们只需要手动配置多个 AuthenticationProvider，并为不同的 AuthenticationProvider 提供不同的 UserDetailsService 即可。

为了方便起见，这里通过 InMemoryUserDetailsManager 来提供 UserDetailsService 实例，在实际开发中，只需要将 UserDetailsService 换成自定义的即可，具体配置如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    @Primary
    UserDetailsService us1() {
        return new InMemoryUserDetailsManager(User.builder()
            .username("javaboy").password("{noop}123").roles("admin").build());
    }
    @Bean
    UserDetailsService us2() {
        return new InMemoryUserDetailsManager(User.builder()
```

```

        .username("sang").password("{noop}123").roles("user").build());
    }
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean()
        throws Exception {
        DaoAuthenticationProvider dao1 = new DaoAuthenticationProvider();
        dao1.setUserDetailsService(us1());
        DaoAuthenticationProvider dao2 = new DaoAuthenticationProvider();
        dao2.setUserDetailsService(us2());
        ProviderManager manager = new ProviderManager(dao1, dao2);
        return manager;
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            //省略
    }
}

```

首先定义了两个 `UserDetailsService` 实例，不同实例中存储了不同的用户；然后重写 `authenticationManagerBean` 方法，在该方法中，定义了两个 `DaoAuthenticationProvider` 实例并分别设置了不同的 `UserDetailsService`；最后构建 `ProviderManager` 实例并传入两个 `DaoAuthenticationProvider`。当系统进行身份认证操作时，就会遍历 `ProviderManager` 中不同的 `DaoAuthenticationProvider`，进而调用到不同的数据源。

#### 提 示

在本书的配套案例中，笔者提供了一个基于 MyBatis 配置多数据源的案例，读者可以参考。

## 3.3 添加登录验证码

登录验证码也是项目中一个常见的需求，但是 `Spring Security` 对此并未提供自动化配置方案，需要开发者自行定义。一般来说，有两种实现登录验证码的思路：

- (1) 自定义过滤器。
- (2) 自定义认证逻辑。

通过自定义过滤器来实现登录验证码，这种方案我们会在本书后面的过滤器章节中介绍，这里先来看如何通过自定义认证逻辑实现添加登录验证码功能。

生成验证码，可以自定义一个生成工具类，也可以使用一些现成的开源库来实现。这里采用开源库 `kaptcha`，首先引入 `kaptcha` 依赖，代码如下：

```
<dependency>
  <groupId>com.github.penggle</groupId>
  <artifactId>kaptcha</artifactId>
  <version>2.3.2</version>
</dependency>
```

然后对 **kaptcha** 进行配置：

```
@Configuration
public class KaptchaConfig {
    @Bean
    Producer kaptcha() {
        Properties properties = new Properties();
        properties.setProperty("kaptcha.image.width", "150");
        properties.setProperty("kaptcha.image.height", "50");
        properties.setProperty("kaptcha.textproducer.char.string",
                                "0123456789");
        properties.setProperty("kaptcha.textproducer.char.length", "4");
        Config config = new Config(properties);
        DefaultKaptcha defaultKaptcha = new DefaultKaptcha();
        defaultKaptcha.setConfig(config);
        return defaultKaptcha;
    }
}
```

配置一个 **Producer** 实例，主要配置一下生成的图片验证码的宽度、长度、生成字符、验证码的长度等信息。配置完成后，我们就可以在 **Controller** 中定义一个验证码接口了：

```
@Autowired
Producer producer;
@GetMapping("/vc.jpg")
public void getVerifyCode(HttpServletResponse resp, HttpSession session)
    throws IOException {
    resp.setContentType("image/jpeg");
    String text = producer.createText();
    session.setAttribute("kaptcha", text);
    BufferedImage image = producer.createImage(text);
    try(ServletOutputStream out = resp.getOutputStream()) {
        ImageIO.write(image, "jpg", out);
    }
}
```

在这个验证码接口中，我们主要做了两件事：

- (1) 生成验证码文本，并将文本存入 **HttpSession** 中。
- (2) 根据验证码文本生成验证码图片，并通过 **IO** 流写出到前端。

接下来修改登录表单，加入验证码，代码如下：

```
<form id="login-form" class="form" action="/doLogin" method="post">
```

```

<h3 class="text-center text-info">登录</h3>
<div th:text="{${SPRING_SECURITY_LAST_EXCEPTION}}"></div>
<div class="form-group">
    <label for="username" class="text-info">用户名:</label><br>
    <input type="text" name="uname" id="username" class="form-control">
</div>
<div class="form-group">
    <label for="password" class="text-info">密码:</label><br>
    <input type="text" name="passwd" id="password" class="form-control">
</div>
<div class="form-group">
    <label for="kaptcha" class="text-info">验证码:</label><br>
    <input type="text" name="kaptcha" id="kaptcha" class="form-control">
    
</div>
<div class="form-group">
    <input type="submit" name="submit" class="btn btn-info btn-md"
        value="登录">
</div>
</form>

```

在登录表单中增加一个验证码输入框，验证码的图片地址就是我们在 **Controller** 中定义的验证码接口地址。

接下来就是验证码的校验了。经过前面的介绍，读者已经了解到，身份认证实际上就是在 **AuthenticationProvider#authenticate** 方法中完成的。所以，验证码的校验，我们可以在该方法执行之前进行，需要配置如下类：

```

public class KaptchaAuthenticationProvider extends DaoAuthenticationProvider {
    @Override
    public Authentication authenticate(Authentication authentication)
        throws AuthenticationException {
        HttpServletRequest req = ((ServletRequestAttributes) RequestContextHolder
            .getRequestAttributes()).getRequest();
        String kaptcha = req.getParameter("kaptcha");
        String sessionKaptcha =
            (String) req.getSession().getAttribute("kaptcha");
        if (kaptcha != null && sessionKaptcha != null
            && kaptcha.equalsIgnoreCase(sessionKaptcha)) {
            return super.authenticate(authentication);
        }
        throw new AuthenticationServiceException("验证码输入错误");
    }
}

```

这里重写 **authenticate** 方法，在 **authenticate** 方法中，从 **RequestContextHolder** 中获取当前请求，进而获取到验证码参数和存储在 **HttpSession** 中的验证码文本进行比较，比较通过则继续执行父类的 **authenticate** 方法，比较不通过，就抛出异常。

**注 意**

有的读者可能会想到通过重写 `DaoAuthenticationProvider` 类的 `additionalAuthenticationChecks` 方法来完成验证码的校验，这个从技术上来说是没有问题的，但是这会让验证码失去存在的意义，因为当 `additionalAuthenticationChecks` 方法被调用时，数据库查询已经做了，仅仅剩下密码没有校验，此时，通过验证码来拦截恶意登录的功能就已经失效了。

最后，在 `SecurityConfig` 中配置 `AuthenticationManager`，代码如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    UserDetailsService usl() {
        return new InMemoryUserDetailsManager(User.builder()
            .username("javaboy").password("{noop}123").roles("admin").build());
    }
    @Bean
    AuthenticationProvider kaptchaAuthenticationProvider() {
        KaptchaAuthenticationProvider provider =
            new KaptchaAuthenticationProvider();
        provider.setUserDetailsService(usl());
        return provider;
    }
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        ProviderManager manager =
            new ProviderManager(kaptchaAuthenticationProvider());
        return manager;
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/vc.jpg").permitAll()
            .anyRequest().authenticated()
            //省略
    }
}
```

这里配置分三步：首先配置 `UserDetailsService` 提供数据源；然后提供一个 `AuthenticationProvider` 实例并配置 `UserDetailsService`；最后重写 `authenticationManagerBean` 方法，提供一个自己的 `ProviderManager` 并使用自定义的 `AuthenticationProvider` 实例。

另外需要注意，在 `configure(HttpSecurity)` 方法中给验证码接口配置放行，`permitAll` 表示这个接口不需要登录就可以访问。

配置完成后，启动项目，浏览器中输入任意地址都会跳转到登录页面，如图 3-5 所示。





图 3-5 包含有验证码的登录页面

此时，输入用户名、密码以及验证码就可以成功登录。如果验证码输入错误，则登录页面会自动展示错误信息。

## 3.4 小 结

本章主要分析了 Spring Security 认证流程中涉及的几个重要的类，`AuthenticationManager`、`AuthenticationProvider`、`ProviderManager` 以及 `AbstractAuthenticationProcessingFilter`，通过这几个类让大家理解 Spring Security 的认证流程到底是什么样子的，同时我们通过两个案例来深化读者对这几个类的理解，并提示目前流行的短信验证码登录的实现思路，也可以通过自定义 `AuthenticationProvider` 来实现，感兴趣的读者可以自行尝试。