Spring Security Principle

Pionpill [1]

本文档为作者学习 SpringSecurity 理论时的笔记。

2023 年 3 月 12 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础，本文前置技术栈:

- Java: Java 基础
- Spring: SpringBoot 基础

本文重点是 Spring Secruity 原理，入门及实战可以参考网上的一些视频教程。主要参考资料:

- SpringSecurity 框架教程 (视频): `https://www.bilibili.com/video/BV1mm4y1X7Hc/`
- « 深入浅出 Spring Security»: 王松清华大学出版社 2021-3

本人的编写及开发环境如下：

- Java: Java17
- SpringBoot: 3.0.2
- SpringSecurity: 6.0.1
- OS: Windows11

2023 年 3 月 12 日

目录

I Spring Security 基础

1 概览

安全框架 (包括 Spring Security) 的两个主要功能: 认证，授权。认证否则身份验证，授权负责访问控制。

Spring Security 的优点有很多，比如:

- Spring 家族成员。
- 良好的微服务配置。
- 自动防止网络攻击。

同时也有一个主要缺点: 臃肿。和其他 Spring 项目类似，很快入门，极难精通。

1.1 认证与授权

认证

在 Spring Security 架构设计中，认证和授权是分开的，两者互不影响。

用户认证信息主要由 `Authentication` 实现类来保存:

```
public interface Authentication extends Principal, Serializable {
    // 获取用户权限
    Collection<? extends GrantedAuthority> getAuthorities();
    // 获取用户凭证，一般是密码
    Object getCredentials();
    // 获取用户携带的详细信息，可能是当前请求
    Object getDetails();
    // 获取当前用户，用户名或用户对象
    Object getPrincipal();
    // 当前用户是否认证成功
    boolean isAuthenticated();
    void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException;
}
```

认证工作主要由 `AuthenticationManager` 接口来负责:

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

`authenticate` 方法用于做认证，有三个不同的返回值:

- 返回 `Authentication`，表认证成功。

- 抛出 AuthenticationException，表示输入了无效的凭证。
- 返回 null，表示不能断定。

AuthenticationManager 最主要的实现类是 ProviderManager，它管理了众多 AuthenticationProvider 实例，AuthenticationProvider 类似于 AuthenticationManager，但多了一个 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 AuthenticationProvider 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(例如账号密码登录,短信验证登录),多个 ProviderManager 统一由 ProviderManager 管理。同时 ProviderManager 具有一个可选的 parent，如果所有认证都失败，就调用 parent 进行认证 (备胎)。

授权

授权体系中，有两个关键接口:

- AccessDecisionManager: 决策器，判断此次访问是否被允许。
- AccessDecisionVoter: 投票器，检查用户是否具有应有的角色。

AccessDecisionManager 会挨个遍历 AccessDecisionVoter 进而决定是否允许用户访问。

在 Spring Security 中，用户请求一个资源所需要的角色会被封装成一个 ConfigAttribute 对象，ConfigAttribute 只有一个 getAttribute 方法，但会角色的名称。

1.2 过滤器

SpringSecurity 中认证授权等功能都是基于过滤器来完成的。过滤器按照既定的优先级排列，最终形成一个过滤器链，开发者可以自定义过滤器链，并通过 @Order 注解去调整过滤器位置。

SpringSecurity 中的过滤器链通过 FilterChainProxy 嵌入到 Web 项目的原生过滤器中，过滤器链可以不止一个，多个过滤器链之间要指定优先级:

FilterChainProxy 作为一个顶层管理者，将统一管理 Security Filter。FilterChainProxy 本身通过 Spring 框架提供的 DelegatingFilterProxy 整合到原生过滤器链中:

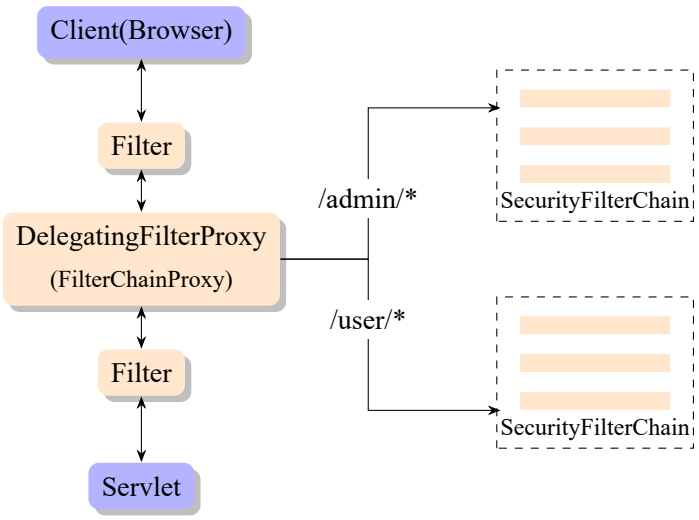

图 1.1 过滤器链

1.3 登录数据

Spring Security 将数据保存在 Session 中，但做了一些调整。

用户成功登陆后，数据会被保存在 `SecurityContextHolder` 中，`SecurityContextHolder` 中的数据保存默认通过 `ThreadLocal` 实现。只允许当前线程获取。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。

2 Spring Security 认证

2.1 默认认证

2.1.1 流程分析

首先我们需要引入 SpringSecurity 依赖:

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

此后我们请求任意接口，都会被强制跳转到登录界面 (/login)，Spring Security 提供了一个简单的账号密码登录验证。默认用户名为 user，密码会在控制台中显示。在这个过程中发生了下图所示请求 (假设访问 /hello)。

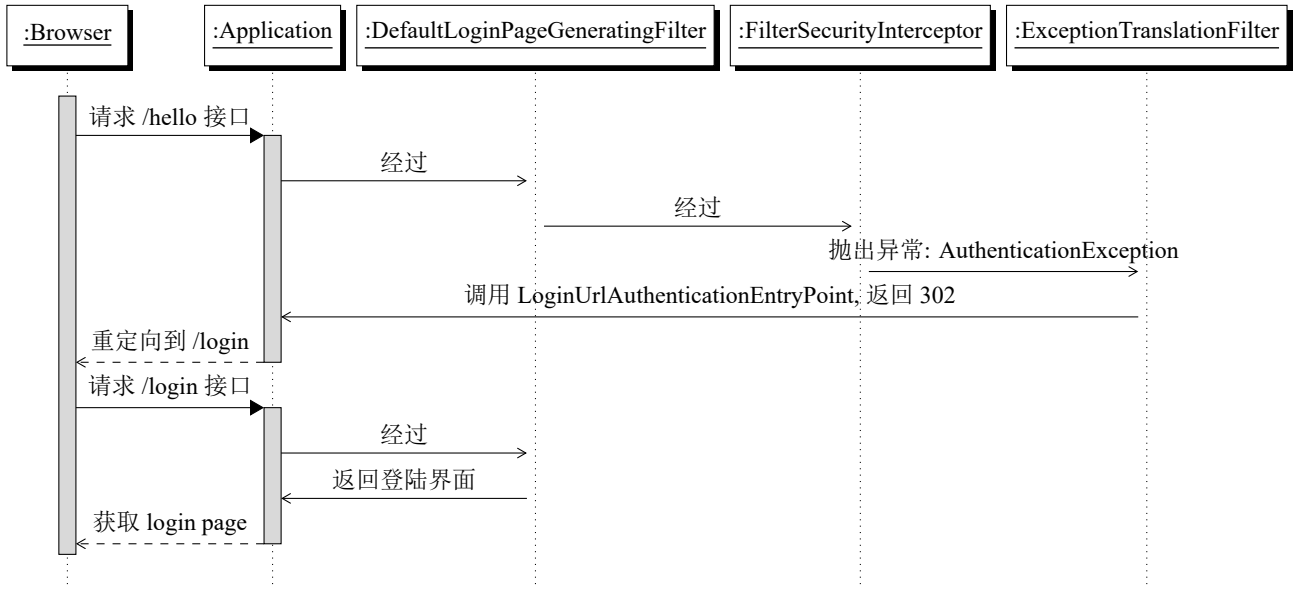

图 2.1 认证请求流程

整个流程如下:

1. **客户端访问 /hello 接口**
2. **请求走过滤器链**, 在 `FilterSecurityInterceptor` 过滤器中被拦截，发现用户未认证，抛出 `AuthenticationException` 异常。
3. **异常被捕获**: `ExceptionTranslationFilter` 过滤器捕获异常,调用 `LoginUrlAuthenticationEntryPoint` 方法返回 302 状态码，重定向到 /login 界面。
4. **客户端发送 /login 请求**
5. **/login 请求被拦截**: /login 请求被 `DefaultLoginPageGeneratingFilter` 拦截，并在过滤器中返回该界面。

我们只引入了一个依赖，Spring Security 就做了这么多事情，主要包括:

- 开启 Spring Security 自动化配置，创建 `springSecurityFilterChain` 并注入到 Spring 容器中。
- 创建一个 `UserDetailService` 实例，负责提供用户数据，默认用户数据是基于内容的。
- 生成一个默认的登陆界面。
- 开启各项攻击防御。

2.1.2 原理分析

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，UserDetails 接口定义如下：

```
public interface UserDetails extends Serializable {
    // 获取账户具备的权限
    Collection<? extends GrantedAuthority> getAuthorities();
    String getPassword();
    String getUsername();
    // 账户是否未过期
    boolean isAccountNonExpired();
    // 账户是否未被锁
    boolean isAccountNonLocked();
    // 账户凭证(如密码)是否未过期
    boolean isCredentialsNonExpired();
    boolean isEnabled();
}
```

负责提供用户数据源的接口是: `UserDetailService`:

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

`loadUserByUsername` 有一个参数是 `username`, 这是瀛湖认证时传入的用户名，最常见的就是用户在登陆时输入的用户名，实际上在单点登录等其他情况下，可能是别的用户名参数。开发者在这里拿到 `username` 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，需要开发者自己实现 `UserDetailsService` 接口，如果没有实现，Spring Security 也提供了默认实现:

- **UserDetailManager**: 在 `UserDetailsService` 基础上，增加了添加用户，更新用户，删除用户，修改密码，判断用户是否存在 5 个方法。
- **JdbcDaoImpl**: `UserDetailsService` 基础上，通过 spring-jdbc 实现了从数据库中查询用户的方法。
- **InMemoryUserDetailsManager**: 实现了 `UserDetailsService` 中关于用户增删改查的方法，不过都是基于内存的操作，没有持久化。

- ……

如果我们只引入一个 Spring Security 依赖，则默认使用 `InMemoryUserDetailsManager`。

默认情况下 Spring Security 是怎么处理的呢？可以看一下 `UserDetailsServiceAutoConfiguration` 源代码 (比较长，不贴了)，这里只讲关键的地方:

- 默认明文密码需要前缀: noop
- 加载一个 `InMemoryUserDetailsManager` 注入到容器中。

同时启用改配置还有一些前置要求:

- 当前 classPath 下存在 `AuthenticationManager` 类。
- 项目中，系统没有提供 `AuthenticationManager, AuthenticationProvider...` 实例。

默认页面生成

在过滤器链中有两个和页面相关的过滤器: `DefaultLoginPageGeneratingFilter` 和 `DefaultLogoutPageGeneratingFilter`。分别负责登录，登出。

这两个类实现的主要功能 (以 Login 为例) 是: 判断当前请求是否为登陆出错，注销成功，登录请求。如果是这三者中的任一个就会在 `DefaultLoginPageGeneratingFilter` 过滤器中生成登录页并返回。否则请求继续往下走，执行下一个过滤器。

2.2 自定义登陆表

Spring Security5.4 之后，通过创建 `SecurityFilterChain` bean 来配置 `HttpSecurity`。[1]，两篇推荐的文献:

- 注入 `SecurityFilterChain` Bean 完成配置: `https://blog.csdn.net/lazy_LYF/article/details/127284459`
- 基于数据库的配置: `https://blog.csdn.net/lazy_LYF/article/details/127284982`

简单地进行自定义配置如下:

```
@Configuration
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .anyRequest()    // 针对任意请求
                .authenticated() // 进行认证
                .and()           // 回到 HttpSecurity 实例
                .formLogin()     // 开启表单登录
                .loginPage("/login") // 配置登录页面
                .loginProcessingUrl("/doLogin") // 配置登录接口地址
```

[1]后续均以 Spring Security 6 为准，均使用注入 bean 的方法实现

```
                .defaultSuccessUrl("/index") // 默认成功登陆后跳转的地址
                .failureUrl("/login") // 失败后跳转的地址
                .usernameParameter("username") // 用户名参数名
                .passwordParameter("pwd")    // 密码参数名
                .permitAll();    // 允许所有请求进入该界面
        return http.build();
    }
}
```

这里我们通过注入自定义过滤器链替代了 Spring Security 默认的过滤连，并做了一些基础配置。大致流程如下:

- 确定请求类型。
- 配置登录界面及成功与失败处理。
- 返回 `http.build()` 注入到容器中替代默认配置。

上述代码有几个注意的地方:

- `and()`: 表示回到 HttpSecurity 实例,官方提供的链式编程方法,与重新使用 `http.formLogin()` 等效。
- `loginPage()` 表示登录界面，`loginProcessingUrl()` 表示后端处理的接口地址。

2.2.1 登陆成功

在配置方法中，与成功登录相关的方法是 `defaultUrl()`: 表示成功登录之后，会自动重定向到登陆前的地址上。例如访问的是 /hello 接口，302 重定向到 /login 接口认证之后会回到之前的 /hello 界面。

这存在一个问题，如果之前访问的是 /login 接口，那么认证成功后会停留在当前界面，这对用户不是很友好。有两种解决方案:

- 使用 `successForwardUrl`，只要成功登录，强制跳转到指定的界面。
- `defaultSuccess` 有一个重载方法，第二个参数传入 `true`，则不考虑用户之前的访问地址，进入 `defaultSuccess` 页面。

这两者有一个不同之处，`successForwardUrl` 通过服务器跳转实现，`defaultSuccess` 则通过让客户端重定向实现。

登陆成功后会由 `AuthenticationSuccessHandler` 接口处理:

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

    void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
        Authentication authentication) throws IOException, ServletException;
```

```
}
```

AuthenticationSuccessHandler 有两个直接实现类:

- SimpleUrlAuthenticationSuccessHandler: 继承自 AbstractAuthenticationTargetUrlRequestHandler，通过它的 handle 方法实现请求重定向。
- ForwardAuthenticationSuccessHandler: 服务端跳转。

比较重要的是 SaveRequestAwareAuthenticationSuccessHandler，继承自 SimpleUrlAuthenticationSuccessHandler，在其基础上添加了请求缓存功能，可以记录之前的地址，进而在登陆成功之后重定向到一开始访问的地址。defaultSuccessUrl 对应的实现类就是它。它的处理逻辑如下:

- 获取缓存请求，如果没有，说明在访问登录页之前没有访问其他页面，交给父类处理，重定向到 defaultSuccessUrl 地址。
- 如果存在缓存，且存在一个 targetUrlParameter 或者 defaultSuccessUrl 第二个参数为 true 则直接重定向到默认地址。
- 否则，获取重定向地址进行重定向。

我们可以自定义对应的逻辑:

```
@Bean
SavedRequestAwareAuthenticationSuccessHandler savedRequestAwareAuthenticationSuccessHandler() {
    SavedRequestAwareAuthenticationSuccessHandler handler = new SavedRequestAwareAuthenticationSuccessHandler();
    handler.setDefaultTargetUrl("/index");
    handler.setTargetUrlParameter("target");
    return handler;
}
```

successForwardUrl 对应 ForwardAuthenticationSuccessHandler，其功能特别简单，就是一个服务器转发。

现在流行的前后端分离架构中，一般不需要后端处理这些逻辑，应该返回 json 数据由前端进行判断:

```
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","成功登录!");
        ObjectMapper om = new ObjectMapper();
        String s = om.writeValueAsString(resp);
        response.getWriter().write(s);
    }
}
```

然后将其写入带 Config 文件中即可:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests()
            ...
            .successHandler(new MyAuthenticationSuccessHandler())
            ...
    return http.build();
}
```

2.2.2 登录失败

登录失败和成功逻辑类似，相关方法是 `failureUrl`，表示登录失败后重定向的路由。如果希望展示请求失败的异常信息，可以使用这种方式:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests()
        ...
        .failureForwardUrl("mylogin")
        ...
    return http.build();
}
```

`failureForwardUrl` 是一种服务器跳转，好处是可以携带登陆异常信息。

这两种方式所配置的都是 `AuthenticationFailureHandler` 接口:

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

它对应的实现类有很多，这里列出重要的几个:

- **`SimpleUrlAuthenticationFailureHandler`**: 默认处理逻辑，重定向到登陆界面，是 `failureUrl()` 方法的底层实现。
- **`ExceptionMappingAuthenticationFailureHandler`**: 根据异常类型映射到不同路径。
- **`ForwardAuthenticationFailureHandler`**: 通过服务端跳转到登陆界面，是 `failureForwardUrl` 方法的底层实现。

自定义登陆失败的方法和登陆成功类似，不再赘述。

2.2.3 注销登录

Spring Security 提供了默认的注销界面:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
```

```
    http.authorizeHttpRequests()
        ...
        .and()
        .logout()  // 开启注销登录配置
        .logoutUrl("logout") // 指定请求地址
        .invalidateHttpSession(true) // 是否使 session 失效
        .clearAuthentication(true) // 是否清除认证信息
        .logoutSuccessUrl("/myLogout") // 注销后跳转的地址
        ...
    return http.build();
}
```

自定义配置方法类似，不再赘述。

2.3 登录用户数据获取

登录成功之后，用户信息将默认保存在 HttpSession 中，Spring Security 对齐进行了封装，开发者获取用户数据有两种主流的思路:

- 从 `SecurityContextHolder` 中获取。
- 从当前请求对象中获取。

无论使用哪种方案都离不开 `Authentication` 对象:

```
public interface Authentication extends Principal, Serializable {
    // 获取用户权限
    Collection<? extends GrantedAuthority> getAuthorities();
    // 获取用户凭证，一般是密码
    Object getCredentials();
    // 获取用户携带的详细信息，可能是当前请求
    Object getDetails();
    // 获取当前用户，用户名或用户对象
    Object getPrincipal();
    // 当前用户是否认证成功
    boolean isAuthenticated();
    void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException;
}
```

`Authentication` 对象主要有两方面的功能:

- 作为 `AuthenticationManager` 输入参数，提供用户身份认证的凭证。
- 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` 对象主要包括三方面信息:

- **`principal`**: 定义认证的用户，通常是 `UserDetail` 对象。
- **`credentials`**: 登录凭证，比如密码。登陆成功后，登录凭证会被自动擦除。
- **`authorities`**: 用户被赋予的权限信息。

Java 本身提供了 `Principal` 接口来描述认证主体，`Authentication` 则继承自 `Principal`。`Authentication` 有多个实现类，常用的有两个:

- **`UsernamePasswordAuthenticationToken`**: 表单登录时封装的用户对象。
- **`RememberMeAuthenticationToken`**: 使用 RememberMe 方式时封装的用户对象。

2.3.1 从 `SecurityContextHolder` 中获取

一般的，获取用户信息的代码如下:

```
@GetMapping("/user")
public void userInfo() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    String username = authentication.getName();
    ......
}
```

其中 `SecurityContextHolder.getContext()` 是一个静态方法，用于返回一个 `SecurityContext` 对象。

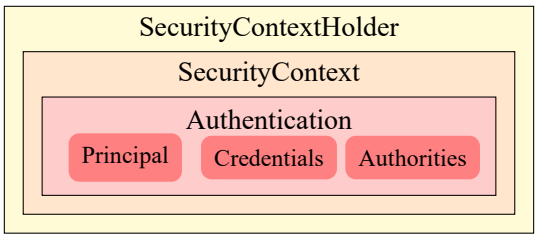

图 2.2 SecurityContextHolder 关系

`SecurityContextHolder` 中存放的是 `SecurityContext`，共定义了三种不同的数据存储策略:

- **MODE_THREADLOCAL**: 将 `SecurityContext` 存放到 `ThreadLocal` 中，是默认的存储策略。如果开启了子线程，则无法获取用户信息。
- **MODE_INHERITABLETHREADLOCAL**: 适用于多线程模式。
- **MODE_GLOBAL**: 实际上将数据保存在静态变量中，Web 开发中很少使用.

`SecurityContextHolderStrategy` 用于规范存储策略中的方法:

```
public interface SecurityContextHolderStrategy {
    void clearContext();
    SecurityContext getContext();
    default Supplier<SecurityContext> getDeferredContext() {
        return () -> {
            return this.getContext();
        };
    }
    void setContext(SecurityContext context);
    default void setDeferredContext(Supplier<SecurityContext> deferredContext) {
        this.setContext((SecurityContext)deferredContext.get());
```

```
    }
    SecurityContext createEmptyContext();
}
```

对应三种存储策略, 有三个 SecurityContextHolderStrategy 实现类. 都比较简单, 读者有兴趣可以自行查源代码.

SecurityContextHolderFilter

SecurityContextHolderFilter 是专为存储 Security Context 而设计的. 主要做了两件事:

- 请求到来, 从 HttpSession 中获取 SecurityContext 并存入 SecurityContextHolder 中.
- 请求完成, 从 SecurityContextHolder 中获取 SecurityContext 并存入 HttpSession 中.

在此之前, 先要了解 SecurityContextRepository 接口:

```
public interface SecurityContextRepository {
    /** @deprecated */
    @Deprecated
    SecurityContext loadContext(HttpRequestResponseHolder requestResponseHolder);
    default DeferredSecurityContext loadDeferredContext(HttpServletRequest request) {
        Supplier<SecurityContext> supplier = () -> {
            return this.loadContext(new HttpRequestResponseHolder(request,
                (HttpServletResponse)null));
        };
        return new SupplierDeferredSecurityContext(SingletonSupplier.of(supplier),
            SecurityContextHolder.getContextHolderStrategy());
    }
    void saveContext(SecurityContext context, HttpServletRequest request, HttpServletResponse
        response);
    boolean containsContext(HttpServletRequest request);
}
```

它的几个方法很好理解, 主要实现类是 HttpSessionSecurityContextRepository. 实现了将 SecurityContext 存储到 HttpSession 以及从 HttpSession 中加载 SecurityContext 出来. 具体的实现方案比较复杂, 有兴趣请自行查源代码.

SecurityContextHolderFilter 主要通过 SecurityContextRepository 获取用户信息, 主要方法如下:

```
public class SecurityContextHolderFilter extends GenericFilterBean {
    private final SecurityContextRepository securityContextRepository;
    private SecurityContextHolderStrategy securityContextHolderStrategy =
        SecurityContextHolder.getContextHolderStrategy();

    // 获取 SecurityContextRepository 实例
    public SecurityContextHolderFilter(SecurityContextRepository securityContextRepository) {
```

```
        Assert.notNull(securityContextRepository, "securityContextRepository cannot be null");
        this.securityContextRepository = securityContextRepository;
    }

    private void doFilter(HttpServletRequest request, HttpServletResponse response,
        FilterChain chain) throws ServletException, IOException {
        if (request.getAttribute(FILTER_APPLIED) != null) {
            chain.doFilter(request, response);
        } else {
          request.setAttribute(FILTER_APPLIED, Boolean.TRUE);
          // 获取信息
          Supplier<SecurityContext> deferredContext =
              this.securityContextRepository.loadDeferredContext(request);
          try {
              // 写入信息
              this.securityContextHolderStrategy.setDeferredContext(deferredContext);
              chain.doFilter(request, response);
          } finally {
              this.securityContextHolderStrategy.clearContext();
              request.removeAttribute(FILTER_APPLIED);
          }

        }
    }
}
```

一言以蔽之，请求在到达过滤器之后，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。

2.3.2 从当前请求对象中获取

开发者可以直接在 Controller 请求参数中放入 `Authentication` 对象来获取登录用户信息.

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

Controller 中的方法是当前请求 `HttpServletRequest` 带来的. 那么这些数据 (`Authentication` 参数) 是如何放入的呢?

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。

此外, 还有一个 SecurityContextHolderAwareRequestFilter 过滤器, 对 HttpServletRequest 请求进行再包装.

对请求的 HttpServletRequest 包装之后，接下来在过滤器链中传递的 HttpServletRequest 对象,它的多个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) 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.

2.4 用户定义

自定义用户其实是使用 UserDetailService 的不同实现类来提供用户数据, 同时将配置好的 UserDetailService 配置给 AuthenticationManagerBuilder, 系统再将 UserDetailService 提供给 AuthenticationProvider 使用.

一种配置方案如下 (部分代码省略):

```
@Bean
AuthenticationManager authenticationManager(HttpSecurity httpSecurity) throws Exception {
    UserDetailsService userServiceImpl;
    AuthenticationManager authenticationManager =
        httpSecurity.getSharedObject(AuthenticationManagerBuilder.class)
           .userDetailsService(userServiceImpl)
           .passwordEncoder(passwordEncoder())
           .and()
           .build();
    return authenticationManager;
}
```

3 认证流程

3.1 登录流程分析

与登录流程相关的三个基本组件: AuthenticationManager, ProviderManager 和 AuthenticationProvider. 相关过滤器: AbstractAuthenticationProcessingFilter.

3.1.1 AuthenticationManager

AuthenticationManager 是一个认证管理器, 定义了 Spring Security 要如何执行认证操作. 认证成功后会返回一个 Authentication 对象, 这个对象会被设置到 SecurityContextHolder 中.

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

AuthenticationManager 对传入的 Authentication 进行身份认证, 此时传入的 Authentication 只有用户名, 密码等简单的数学, 如果认证成功, 会得到补充.

AuthenticationManager 常用的的实现类是 ProviderManager 也是 Spring Security 默认的实现类.

3.1.2 AuthenticationProvider

AuthenticationProvider 针对不同的身份类型执行具体的身份认证. 常见的认证方案及对应实现类如下:

- DaoAuthenticationProvider: 支持用户名/密码登录认证.
- RememberMeAuthenticationProvider: 支持 “记住我” 认证.

```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication) throws AuthenticationException;
    // 判断是否支持对应的身份类型
    boolean supports(Class<?> authentication);
}
```

大部分实现类都继承自 AbstractUserDetailsAuthenticationProvider, 它的几个主要属性和方法如下:

- **userCache**: 声明一个用户缓存对象, 默认情况下不启用.
- **hideUserNotFoundExceptions**: 隐藏失败异常, 抛出一个模糊的 BadCredentialsException 异常代替查不到用户, 验证错误等异常, 默认开启.
- **forcePrincipalAsString**: 默认关闭, 返回 UserDetails 对象, 开启后仅返回用户名.
- **preAuthenticationChecks**: 状态检查, 校验前
- **postAuthenticationChecks**: 状态检测, 校验后

- **`additionalAuthentication()`**: 校验密码
- **`authenticate()`**: 核心校验方法

`authenticate()` 方法检验用户名密码登录的逻辑如下:

- 从登陆数据中获取用户名;
- 查用户获取用户对象;
  - 根据用户名去缓存中查询用户对象;
  - 如果查不到, 仅数据库加载用户;
  - 如果还是查不到, 抛出异常 (用户不存在);
- 获取到用户对象后, 调用 `preAuthenticationChecks.check()` 进行用户状态检查.
- 调用 `additionalAuthenticationChecks()` 进行密码校验;
- 调用 `postAuthenticationChecks.check()` 检查密码是否过期,
- 调用 `createSuccessAuthentication` 方法创建一个认证后的 `UsernamePasswordAuthenticationToken` 对象并返回.

3.1.3 `ProviderManager`

`ProviderManager` 是 `AuthenticationManager` 的一个重要实现类:

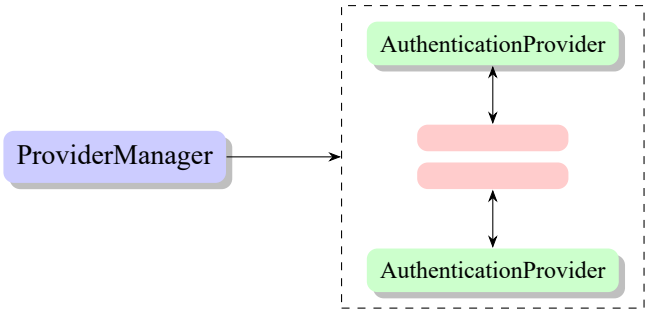

图 3.1 Provider Manager

多个 `AuthenticationProvider` 将组成一个列表, 这个列表将由 `ProviderManager` 代理, 在 `ProviderManager` 中遍历每一个 `AuthenticationProvider` 去执行身份认证, 最终得到认证结果.

理论上, 多个 `ProviderManager` 本身也可以再配置一个 `AuthenticationManager` 作为 parent, 一直套娃下去.

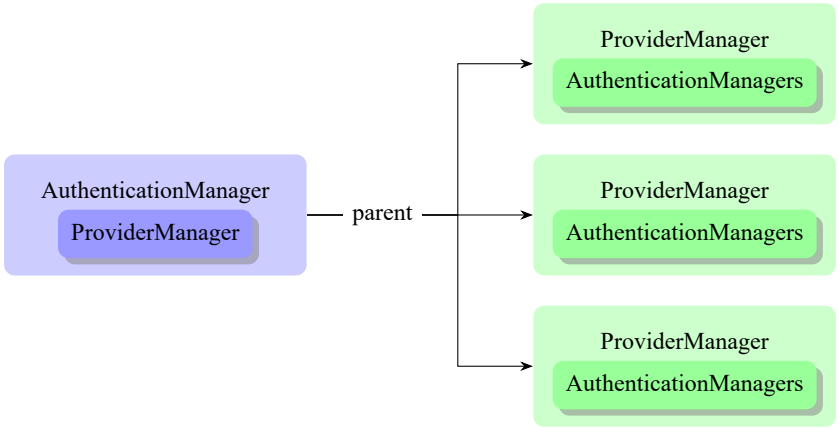

图 3.2 ProviderManager 包含多个 AuthenticationProvider

ProviderManager 的方法 authenticate() 主要执行逻辑如下:

- 遍历 ProviderManager 所代理的所有 AuthenticationProvider 对象进行身份验证.
- 判断当前 AuthenticationProvider 是否支持当前 Authentication 对象, 不支持则使用下一个 AuthenticationProvider 对象.
- 调用 provider.authenticate 方法进行身份认证, 如果认证成功, 返回 Authentication 对象, 同时做一些后处理.
- 如果遍历完后, 所有认证均失败, 此时如果 parent 不为空, 则继续调用 parent 的 authenticate 进行认证.
- 如果认证成功, 则擦除凭证, 将 result 返回, 停止执行后续代码.
- 如果认证失败, 则派出异常.

3.1.4 AbstractAuthenticationProcessingFilter

AbstractAuthenticationProcessingFilter 负责将前面的几个类串联起来, 用于处理任何提交给它的身份认证:

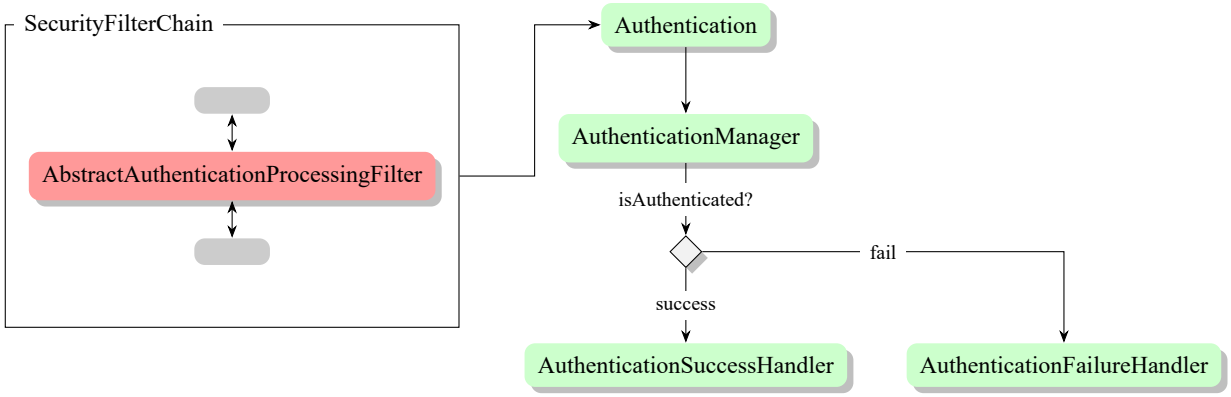

图 3.3 AbstractAuthenticationProcessingFilter 处理流程

如果是使用账户密码登录, 对应的实际类如下:

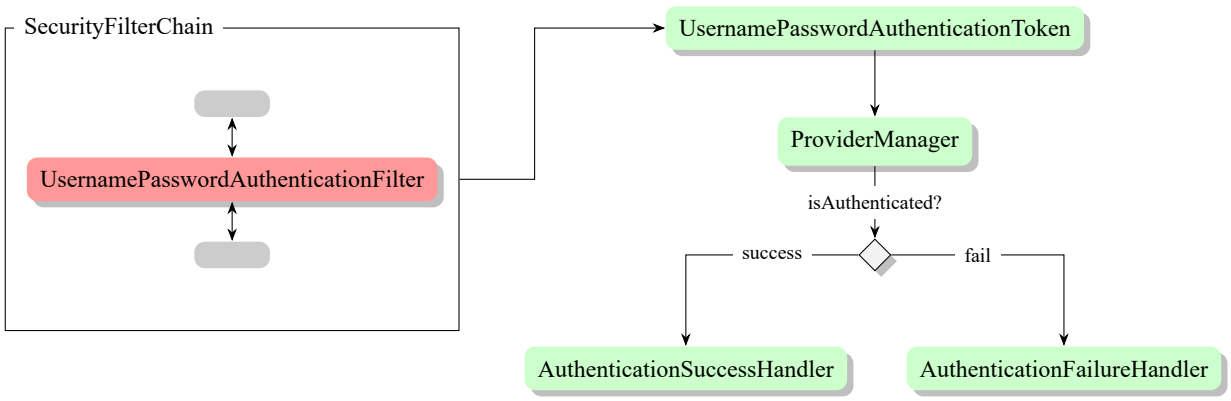

图 3.4 用户名密码登录处理流程

上图的大致处理流程如下:

- 用户提交登录请求,`UsernamePasswordAuthenticationFilter` 从 `HttpServletRequest` 中获取用户名/密码, 然后创建一个 `UsernamePasswordAuthenticationToken` 对象.
- `UsernamePasswordAuthenticationToken` 对象被传入 `ProviderManager` 进行认证.
- 认证失败, 则 `SecurityContextHolder` 中相关信息被清除, 并进行失败回调.
- 认证成功, 则进行登录信息存储, Session 并发处理, 登陆成功时间发布以及登陆方法回调等操作.

`AbstractAuthenticationProcessingFilter` 的具体实现逻辑还是比较简单的, 有兴趣可以自行查阅源码.

3.2 配置多个数据源

配置多个数据源, 即认证时, 如果第一张表没有查找到用户, 就去第二张表中查, 以此类推.

实现这个很简单, 如前文所述, 一个 `AuthenticationProvider` 配置了一个 `UserDetailService` 不同的 `UserDetailService` 代表不同的数据源, 所以只需要配置多个 `AuthenticationProvider` 并提供不同的 `UserDetailService` 即可.

```
@Bean
@Primary
UserDetailsService us1() {
    return new InMemoryUserDetailsManager(User.builder()
      .username("pionpill").password("{noop}123456").roles("admin").build());
}
@Bean
UserDetailsService us2() {
    return new InMemoryUserDetailsManager(User.builder()
      .username("beian").password("{noop}123456").roles("user").build());
}

@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;
}
```

3.3 添加登录验证码

Spring Security 没有给出自动配置登录验证码的方案, 有两种实现登录验证码的思路:

- 自定义过滤器.
- 自定义认证逻辑.

原书这里使用自定义认证逻辑来实现. 逻辑上就是重写 `AuthenticationProvider` 的 `authenticate()` 方法, 先进行验证码判断, 再调用 `super.authenticate()`. 由于不涉及新的技能点, 这里不再赘述, 可以参考这篇文章 `https://blog.csdn.net/zhouzhiwengang/article/details/96155447`.

4 过滤器链

4.1 初始化流程分析

Spring Security 初始化流程设计很多零碎的知识点, 这里先介绍一些常见的关键组件.

ObjectPostProcessor

ObjectPostProcessor 听名字就知道, 是用于做后处理的.

```
public interface ObjectPostProcessor<T> {
    <O extends T> O postProcess(O object);
}
```

它有两个默认的继承类:

- AutowireBeanFactoryObjectPostProcessor: 用于注入容器的. 和框架相关, 无需过多了解.
- CompositeObjectPostProcessor: 是一个组合的对象后处理器, 了吗维护一个 List 集合, 集合中存放了某个对象的所有后置处理器, 遍历调用. 默认情况下只有一个后处理器 AutowireBeanFactoryObjectPostProcessor.

在 Spring Security 中，开发者可以灵活地配置项目中需要哪些 Spring Security 过滤器，一旦选定过滤器之后，每一个过滤器都会有一个对应的配置器，叫作 xxxConfigurer 过滤器都是在 xxxConfigurer 中 new 出来的，然后在 postProcess 方法中处理一遍，就将这些过滤器注入到 Spring 容器中了。

这是对象后置处理器 ObjectPostProcessor 的主要作用。

SecurityFilterChain

SecurityFilterChain 即过滤器链对象:

```
public interface SecurityFilterChain {
    // 判断当前 request 请求是否被当前过滤器链处理
    boolean matches(HttpServletRequest request);
    // 存放过滤器,如果 matches 判断为 true,则进行处理
    List<Filter> getFilters();
}
```

SecurityFilterChain 只有一个默认实现类 DefaultSecurityFilterChain.

SecurityBuilder

Spring Security 中所有需要构建的对象都可以通过 SecurityBuilder 实现:

```
public interface SecurityBuilder<O> {
```

```
    O build() throws Exception;
}
```

`SecurityBuilder` 的子类较多, 我们只需要知道它最终会返回过滤器链 (`FilterChainProxy`) 给我们就行.

`FilterChainProxy`

`FilterChainProxy` 通过 `DelegatingFilterProxy` 代理过滤器被集成到 Web Filter 中. Spring Security 中的过滤器链最终执行就是在 `FilterChainProxy` 中.

`SecurityConfigurer`

`SecurityConfigurer` 有两个核心方法, 一个是 `init` 方法, 用来完成配置类的初始化操作, 另一个是 `configure` 方法, 进行配置类的配置:

```
public interface SecurityConfigurer<O, B extends SecurityBuilder<O>> {
    void init(B builder) throws Exception;
    void configure(B builder) throws Exception;
}
```

初始化流程

Spring Security 的默认自动化配置类是 `SecurityAutoConfiguration`:

```
@AutoConfiguration
@ConditionalOnClass({DefaultAuthenticationEventPublisher.class})
@EnableConfigurationProperties({SecurityProperties.class})
@Import({SpringBootWebSecurityConfiguration.class, SecurityDataConfiguration.class})
public class SecurityAutoConfiguration {
    public SecurityAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean({AuthenticationEventPublisher.class})
    public DefaultAuthenticationEventPublisher
        authenticationEventPublisher(ApplicationEventPublisher publisher) {
        return new DefaultAuthenticationEventPublisher(publisher);
    }
}
```

主要作用是导入了两个配置类, 并定义了一个默认的事件发布器, 配置类的作用如下:

- `SpringBootWebSecurityConfiguration`: 如果开发者没有提供 `WebSecurityConfigurerAdapter` 则提供一个默认的实例.
- `SecurityDataConfiguration`: 提供一个 `SecurityEvaluationContextExtension` 实例, 以便通过 SpEL 为经过身份验证的用户提供数据查询.

4.2 多种用户定义方式

前面我共用到了两种用户定义方式:

- **重写 `configure` 方法**: 对应全局 `AuthenticationManager`.
- **直接注入 `UserDetailsService`**: 对应局部 `AuthenticationManager`.

当用户进行身份验证时, 会首先通过局部身份验证, 如果不通过再进行全局身份验证. 两者互不影响.