

SpringSecurity 实战

第一章 权限管理

- 权限管理
- SpringSecurity 简介
- 整体架构

权限管理

基本上涉及到用户参与的系统都要进行权限管理，权限管理属于系统安全的范畴，权限管理实现对用户访问系统的控制，按照安全规则或者安全策略控制用户可以访问而且只能访问自己被授权的资源。

权限管理包括用户身份认证和授权两部分，简称认证授权。对于需要访问控制的资源用户首先经过身份认证，认证通过后用户具有该资源的访问权限方可访问。

认证

身份认证，就是判断一个用户是否为合法用户的处理过程。最常用的简单身份认证方式是系统通过核对用户输入的用户名和口令，看其是否与系统中存储的该用户的用户名和口令一致，来判断用户身份是否正确。对于采用**指纹**等系统，则出示指纹；对于硬件Key等刷卡系统，则需要刷卡。

授权

授权，即访问控制，控制谁能访问哪些资源。主体进行身份认证后需要分配权限方可访问系统的资源，对于某些资源没有权限是无法访问的。

解决方案

和其他领域不同，在 Java 企业级开发中，安全管理框架非常少，目前比较常见的就是：

- Shiro
 - 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，在微服务面前和扩展方面，无法充分展示自己的优势。

- 开发者自定义
 - 也有很多公司选择自定义权限，即自己开发权限管理。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
 - 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。

简介

官方定义

- <https://spring.io/projects/spring-security>

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements.

Spring Security 是一个功能强大、可高度定制的身份验证和访问控制框架。它是保护基于 Spring 的应用程序的事实标准。

Spring Security 是一个面向 Java 应用程序提供身份验证和安全性的框架。与所有 Spring 项目一样，Spring Security 的真正威力在于它可以轻松地扩展以满足定制需求。

- 总结

Spring Security 是一个功能强大、可高度定制的 **身份验证** 和 **访问控制** 的框架。或者说用来实现系统中权限管理的框架。

历史

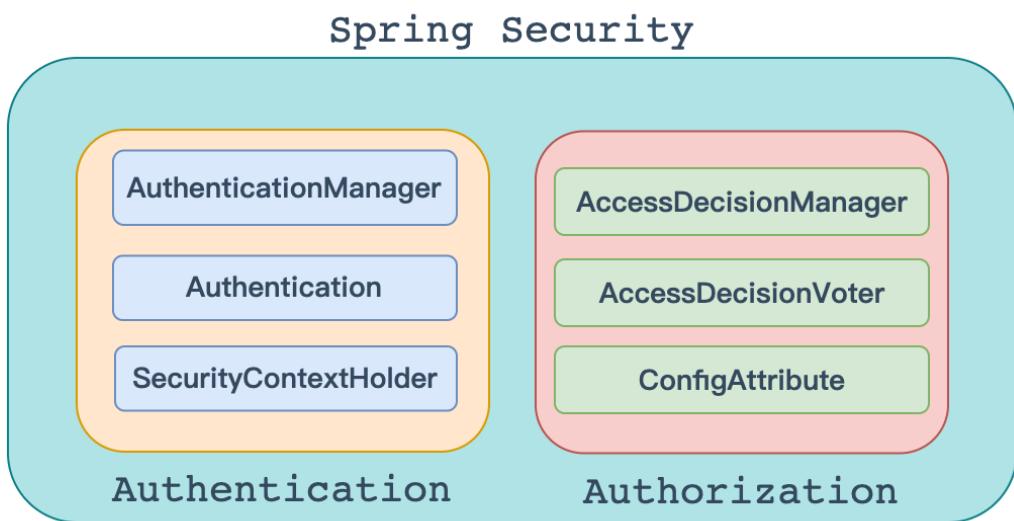
Spring Security 最早叫 Acegi Security，这个名称并不是说它和 Spring 就没有关系，它依然是为 Spring 框架提供安全支持的。Acegi Security 基于 Spring，可以帮助我们为项目建立丰富的角色与权限管理系统。Acegi security 虽然好用，但是最为人诟病的则是它臃肿烦琐的配置这一问题最终也遗传给了 Spring Security。

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，到目前为止，Spring Security 的最新版本已经到了 5.6.1。和 Shiro 相比，Spring Security 重量级并且配置烦琐，直至今天，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 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。

整体架构

在的架构设计中，**认证** 和 **授权** 是分开的，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，就是可以非常方便地整合一些外部的解决方案。



认证

AuthenticationManager

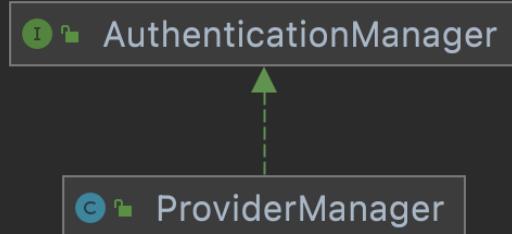
在Spring Security中认证是由 **AuthenticationManager** 接口来负责的，接口定义为：

```
public interface AuthenticationManager {  
  
    /** Attempts to authenticate the passed {@link Authentication} object, returning a ...*/  
    Authentication authenticate(Authentication authentication) throws AuthenticationException;  
  
}
```

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

- 返回 `Authentication` 表示认证成功
- 返回 `AuthenticationException` 异常，表示认证失败。

`AuthenticationManager` 主要实现类为 `ProviderManager`，在 `ProviderManager` 中管理了众多 `AuthenticationProvider` 实例。在一次完整的认证流程中，Spring Security 允许存在多个 `AuthenticationProvider`，用来实现多种认证方式，这些 `AuthenticationProvider` 都是由 `ProviderManager` 进行统一管理的。



Authentication

认证以及认证成功的信息主要是由 `Authentication` 的实现类进行保存的，其接口定义为：

Authentication

- getAuthorities(): Collection<? extends GrantedAuthority>
- getCredentials(): Object
- getDetails(): Object
- getPrincipal(): Object
- isAuthenticated(): boolean
- setAuthenticated(boolean): void

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

- `getAuthorities` 获取用户权限信息
- `getCredentials` 获取用户凭证信息，一般指密码
- `getDetails` 获取用户详细信息
- `getPrincipal` 获取用户身份信息，用户名、用户对象等
- `isAuthenticated` 用户是否认证成功

SecurityContextHolder

`SecurityContextHolder` 用来获取登录之后用户信息。Spring Security 会将登录用户数据保存在 `Session` 中。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

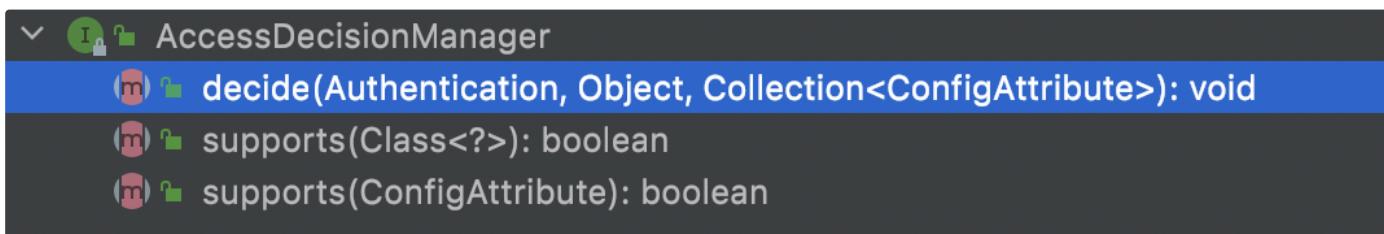
据拿出来保存到 Session 中，然后将 Security SecurityContextHolder 中的数据清空。这一策略非常方便用户在 Controller、Service 层以及任何代码中获取当前登录用户数据。

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，

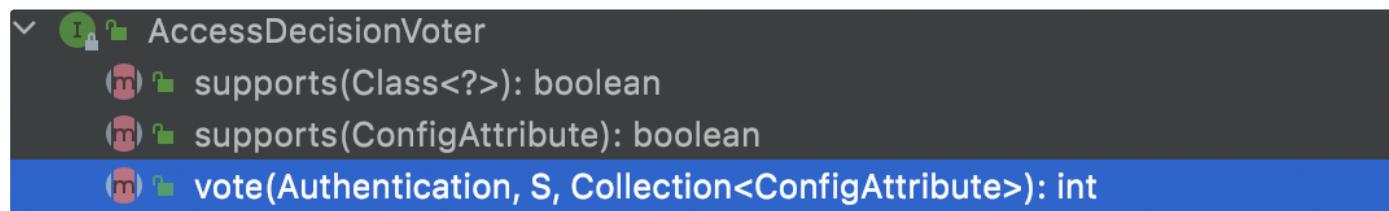
AccessDecisionManager

AccessDecisionManager（访问决策管理器），用来决定此次访问是否被允许。



AccessDecisionVoter

AccessDecisionVoter（访问决定投票器），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。



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。

ConfigAttribute

ConfigAttribute, 用来保存授权时的角色信息



在 Spring Security 中, 用户请求一个资源(通常是一个接口或者一个 Java 方法)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。

第二章 环境搭建

- 环境搭建
- 自动配置细节

环境搭建

- spring boot
- spring security
 - 认证: 判断用户是否是系统合法用户过程
 - 授权: 判断系统内用户可以访问或具有访问那些资源权限过程

创建项目

1. 创建 springboot 应用

```
✓ spring-security-01 ~/Desktop/SpringSecurity/codes/spring-security-01
> .mvn
└ src
  > main
  > test
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
  spring-security-01.iml
```

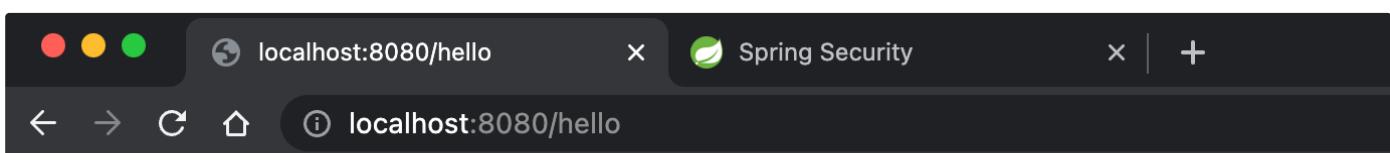
2. 创建 controller

```
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello() {
        System.out.println("hello security");
        return "hello security";
    }
}
```

```
✓ spring-security-01 ~/Desktop/SpringSecurity/codes/spring-security-01
> .mvn
└ src
  └ main
    └ java
      └ com.baizhi
        └ controller
          C HelloController
          S SpringSecurity01Application
```

3. 启动项目进行测试

- <http://localhost:8080/hello>



hello security

整合 Spring Security

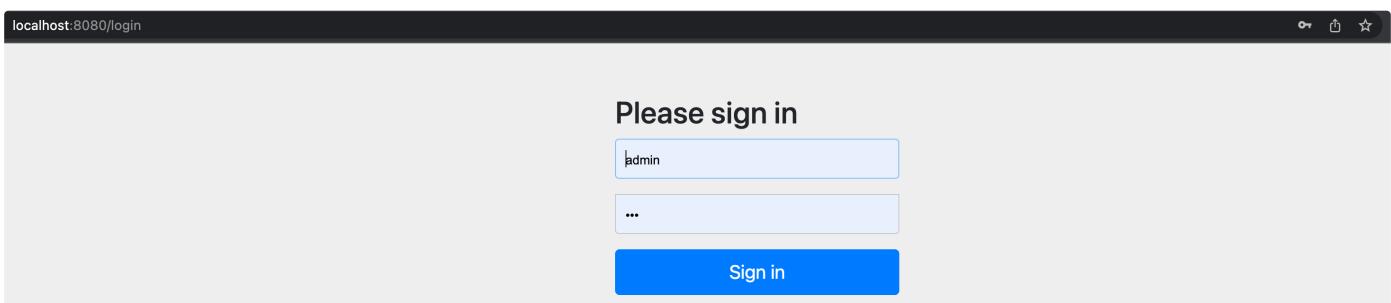
1.引入spring security相关依赖

```
←!—引入spring security依赖→  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

2.再次启动项目

- 1.启动完成后控制台生成一个密码
- 2.访问 hello 发现直接跳转到登录页面

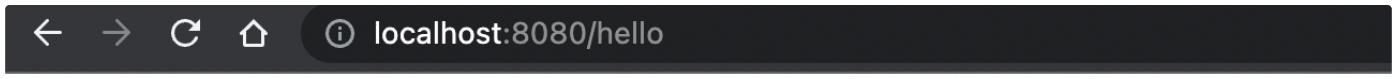
```
2022-01-10 11:40:15.929  INFO 5227 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]      :  
2022-01-10 11:40:15.929  INFO 5227 --- [           main] w.s.c.ServletWebServerApplicationContext :  
2022-01-10 11:40:16.033  INFO 5227 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :  
  
Using generated security password: 10d80a5a-d898-492c-9c5b-312f712d9691
```



3.登录系统

- 默认用户名为: user
- 默认密码为: 控制台打印的 uuid

Please sign in



hello security

这就是 Spring Security 的强大之处，只需要引入一个依赖，所有的接口就会自动保护起来！

思考🤔？

- 为什么引入 Spring Security 之后没有任何配置所有请求就要认证呢？
- 在项目中明明没有登录界面，**登录界面**怎么来的呢？
- 为什么使用 `user` 和 `控制台密码` 能登陆，登录时验证数据源存在哪里呢？

实现原理

<https://docs.spring.io/spring-security/site/docs/5.5.4/reference/html5/#servlet-architecture>

虽然开发者只需要引入一个依赖，就可以让 Spring Security 对应用进行保护。Spring Security 又是如何做到的呢？

在 Spring Security 中 **认证、授权** 等功能都是基于**过滤器**完成的。

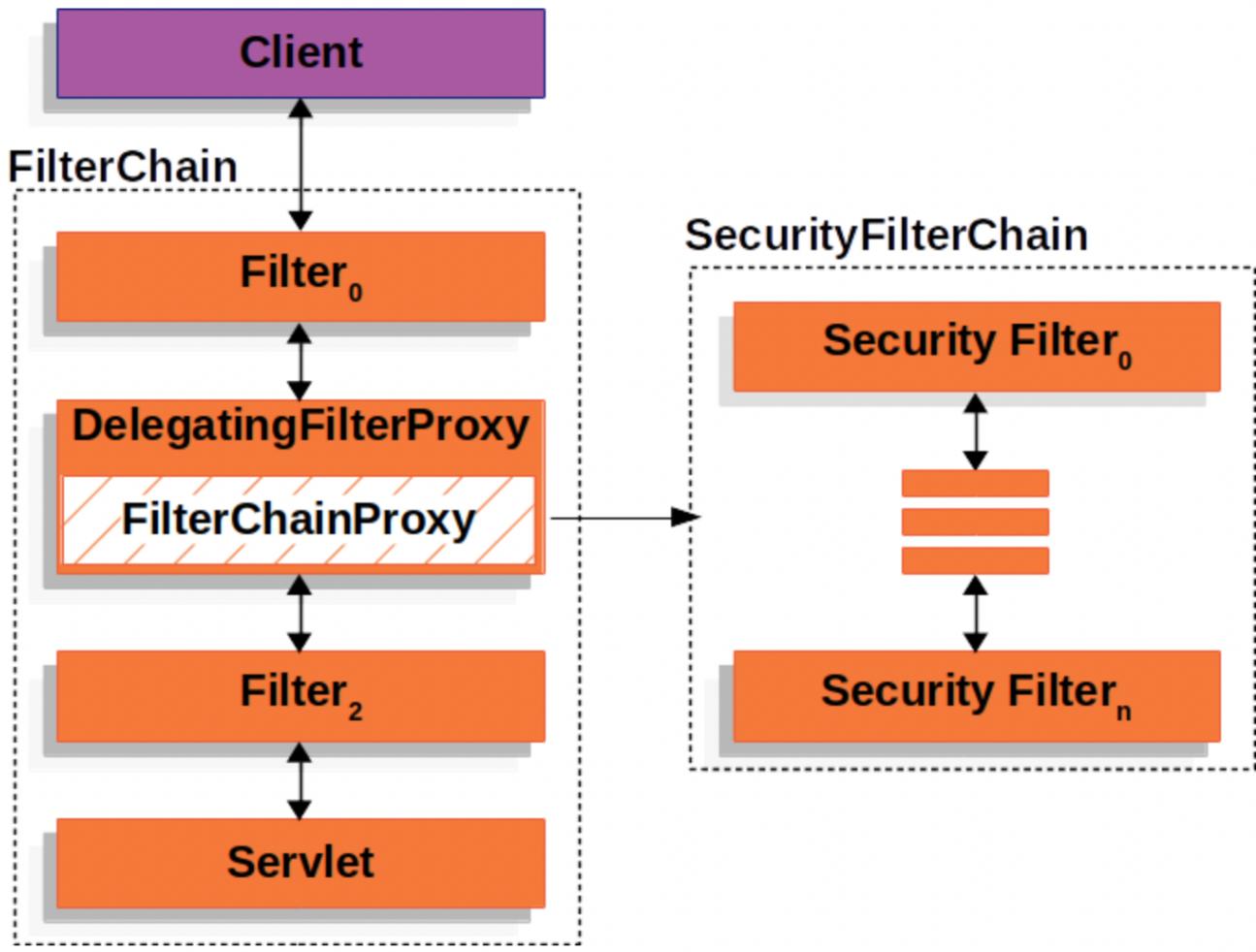
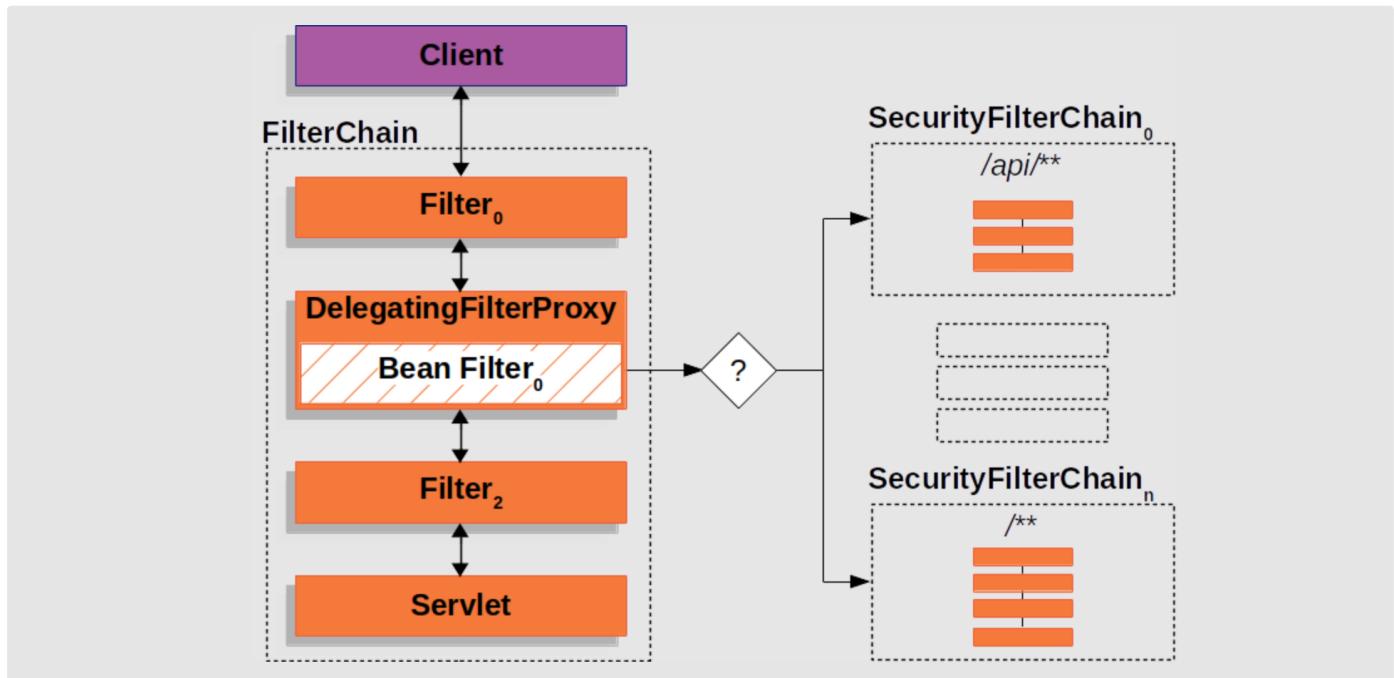


Figure 4. SecurityFilterChain



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。**FilterChainProxy** 作为一个顶层的管理者，将统一管理 **Security Filter**。**FilterChainProxy** 本身是通过

Spring 框架提供的 DelegatingFilterProxy 整合到原生的过滤器链中。

Security Filters

那么在 Spring Security 中给我们提供那些过滤器？默认情况下那些过滤器会被加载呢？

过滤器	过滤器作用	默认是否加载
ChannelProcessingFilter	过滤请求协议 HTTP 、 HTTPS	NO
WebAsyncManagerIntegrationFilter	将 WebAsyncManger 与 SpringSecurity 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	YES
HeaderWriterFilter	处理头信息加入响应中	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 认证的 Access Token	NO
BasicAuthenticationFilter	处理 HttpBasic 登录	YES
RequestCacheAwareFilter	处理请求缓存	YES
SecurityContextHolder AwareRequestFilter	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 认证中授权码	NO
SessionManagementFilter	处理 session 并发问题	YES
ExceptionTranslationFilter	处理认证/授权中的异常	YES
FilterSecurityInterceptor	处理授权相关	YES
SwitchUserFilter	处理账户切换	NO

可以看出，Spring Security 提供了 30 多个过滤器。默认情况下 Spring Boot 在对 Spring Security 进入自动化配置时，会创建一个名为 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，包括用户认证、授权、重定向到登录页面等。具体可以参考 WebSecurityConfiguration 的源码：

The screenshot shows the IntelliJ IDEA code editor with the following details:

- Class Hierarchy:** A tree view on the left shows the class hierarchy for `WebSecurityConfiguration`. The `springSecurityFilterChain(): Filter` method is highlighted with a red rectangle.
- Method Implementation:** The main pane displays the implementation of the `springSecurityFilterChain()` method.

```
for (SecurityFilterChain securityFilterChain : this.securityFilterChains) {    securityFilterChain: "DefaultSecurityFilterChain@5856"    this.webSecurity.addSecurityFilterChainBuilder(() -> securityFilterChain);    webSecurity: WebSecurity@5850      for (Filter filter : securityFilterChain.getFilters()) {        securityFilterChain.filters: [size = 15]          > 0 = {WebAsyncManagerIntegrationFilter@5920}          > 1 = {SecurityContextPersistenceFilter@5928}          > 2 = {HeaderWriterFilter@5935}          > 3 = {CsrfFilter@5942}          > 4 = {LogoutFilter@5949}          > 5 = {UsernamePasswordAuthenticationFilter@5956}          > 6 = {DefaultLoginPageGeneratingFilter@5965}          > 7 = {DefaultLogoutPageGeneratingFilter@5972}          > 8 = {BasicAuthenticationFilter@5986}          > 9 = {RequestCacheAwareFilter@5987}          > 10 = {SecurityContextHolderAwareRequestFilter@5988}          > 11 = {AnonymousAuthenticationFilter@5989}          > 12 = {SessionManagementFilter@5990}          > 13 = {ExceptionTranslationFilter@5991}          > 14 = {FilterSecurityInterceptor@5992}        }      }    }  for (WebSecurityCustomizer customizer : this.webSecurityCustomizers) {    customizer.customize(this.webSecurity);  } return this.webSecurity.build();}
```

SpringBootWebSecurityConfiguration

这个类是 spring boot 自动配置类，通过这个源码得知，默认情况下对所有请求进行权限控制：

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnDefaultWebSecurity
@ConditionalOnWebApplication(type = Type.SERVLET)
class SpringBootWebSecurityConfiguration {
    @Bean
    @Order(SecurityProperties.BASIC_AUTH_ORDER)
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity
http)
        throws Exception {
        http.authorizeRequests().anyRequest()
            .authenticated().and().formLogin().and().httpBasic();
        return http.build();
    }
}
```

```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
    return http.build();
}
```

这就是为什么在引入 `Spring Security` 中没有任何配置情况下，请求会被拦截的原因！

通过上面对自动配置分析，我们也能看出默认生效条件为：

```
class DefaultWebSecurityCondition extends AllNestedConditions {

    DefaultWebSecurityCondition() {
        super(ConfigurationPhase.REGISTER_BEAN);
    }

    @ConditionalOnClass({ SecurityFilterChain.class,
HttpSecurity.class })
    static class Classes {

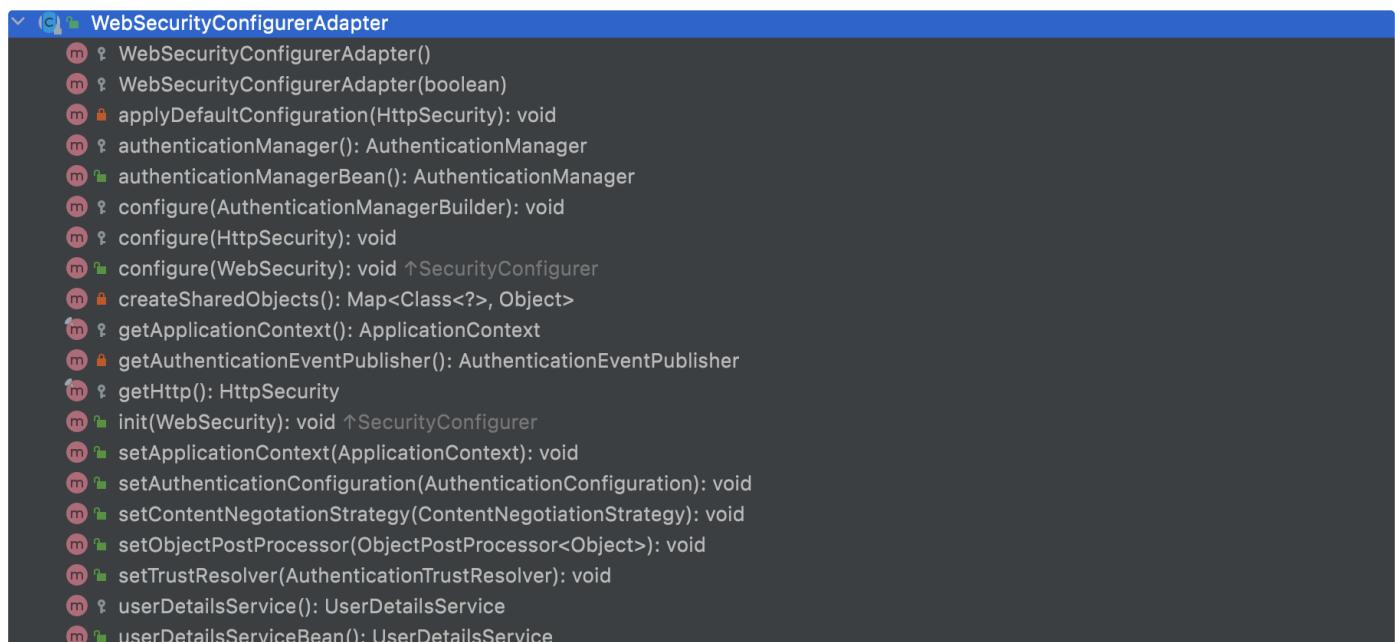
    }

    @ConditionalOnMissingBean({ WebSecurityConfigurerAdapter.class,
SecurityFilterChain.class })
```

```
static class Beans {  
}  
}
```

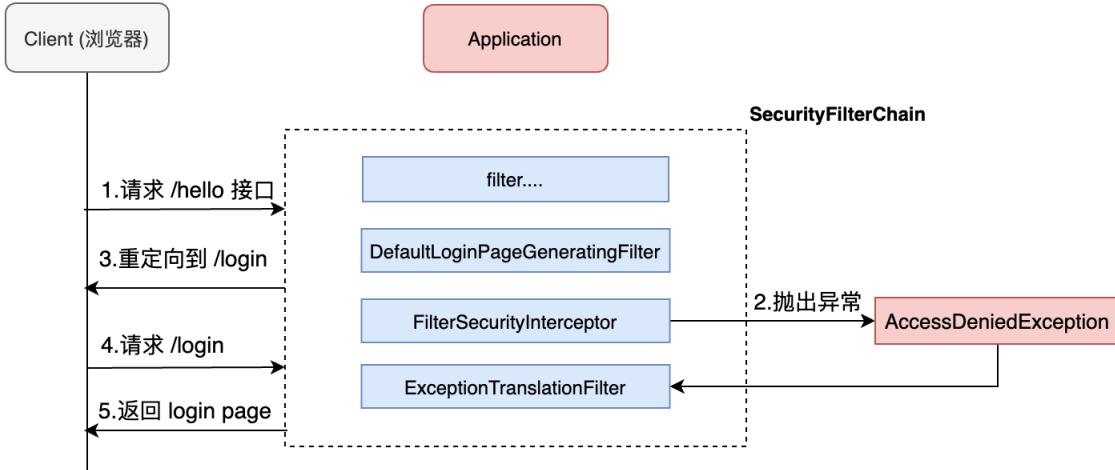
- 条件一 classpath中存在 SecurityFilterChain.class, HttpSecurity.class
- 条件二 没有自定义 WebSecurityConfigurerAdapter.class, SecurityFilterChain.class

默认情况下，条件都是满足的。WebSecurityConfigurerAdapter 这个类极其重要，Spring Security 核心配置都在这个类中：



如果要对 Spring Security 进行自定义配置，就要自定义这个类实例，通过覆盖类中方法达到修改默认配置的目的。

流程分析



1. 请求 /hello 接口，在引入 spring security 之后会先经过一些列过滤器
2. 在请求到达 FilterSecurityInterceptor 时，发现请求并未认证。请求拦截下来，并抛出 AccessDeniedException 异常。
3. 抛出 AccessDeniedException 的异常会被 ExceptionTranslationFilter 捕获，这个 Filter 中会调用 LoginUrlAuthenticationEntryPoint#commence 方法给客户端返回 302，要求客户端进行重定向到 /login 页面。
4. 客户端发送 /login 请求。
5. /login 请求会再次被拦截器中 DefaultLoginPageGeneratingFilter 拦截到，并在拦截器中返回生成登录页面。

就是通过这种方式，Spring Security 默认过滤器中生成了登录页面，并返回！

默认用户生成

1. 查看 SpringBootWebSecurityConfiguration#defaultSecurityFilterChain 方法表单登录

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnDefaultWebSecurity
@ConditionalOnWebApplication(type = Type.SERVLET)
class SpringBootWebSecurityConfiguration {

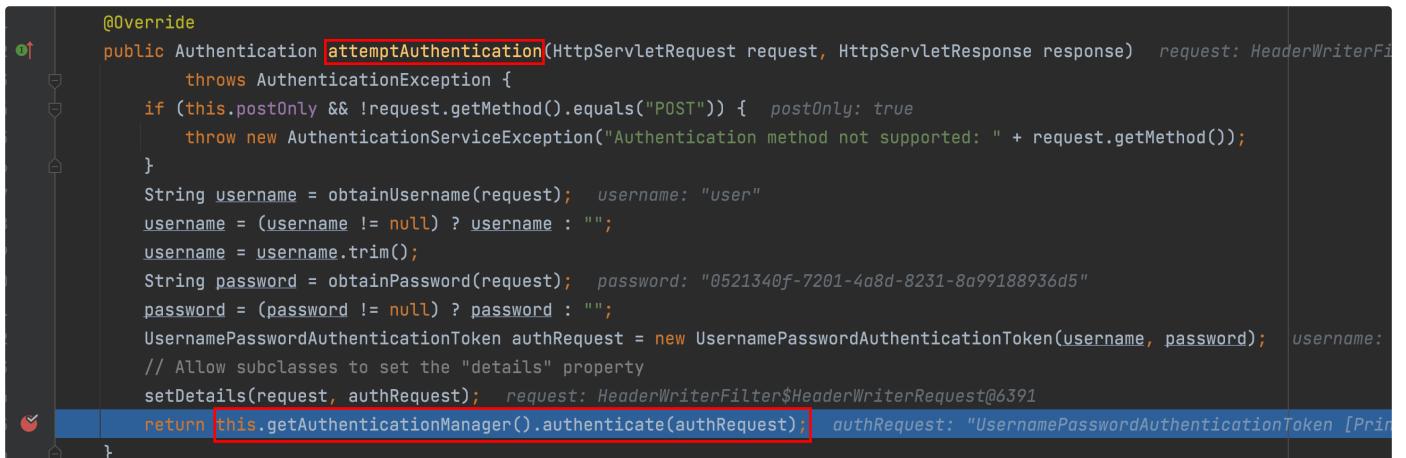
    @Bean
    @Order(SecurityProperties.BASIC_AUTH_ORDER)
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
        return http.build();
    }
}

```

2. 处理登录为 FormLoginConfigurer 类中 调用 UsernamePasswordAuthenticationFilter 这个类实例

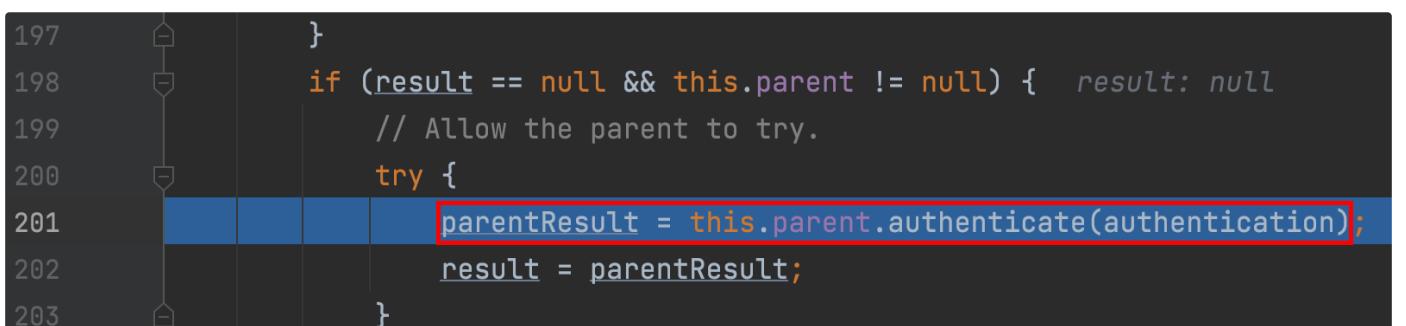
```
public FormLoginConfigurer() {
    super(new UsernamePasswordAuthenticationFilter(), defaultLoginProcessingUrl: null);
    usernameParameter("username");
    passwordParameter("password");
}
```

3. 查看类中 UsernamePasswordAuthenticationFilter#attemptAuthentication 方法得知实际调用 AuthenticationManager 中 authenticate 方法



```
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
    if (this.postOnly && !request.getMethod().equals("POST")) { postOnly: true
        throw new AuthenticationServiceException("Authentication method not supported: " + request.getMethod());
    }
    String username = obtainUsername(request); username: "user"
    username = (username != null) ? username : "";
    username = username.trim();
    String password = obtainPassword(request); password: "0521340f-7201-4a8d-8231-8a99188936d5"
    password = (password != null) ? password : "";
    UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(username, password); username:
    // Allow subclasses to set the "details" property
    setDetails(request, authRequest); request: HeaderWriterFilter$HeaderWriterRequest@6391
    return this.getAuthenticationManager().authenticate(authRequest); authRequest: "UsernamePasswordAuthenticationToken [Pri
}
```

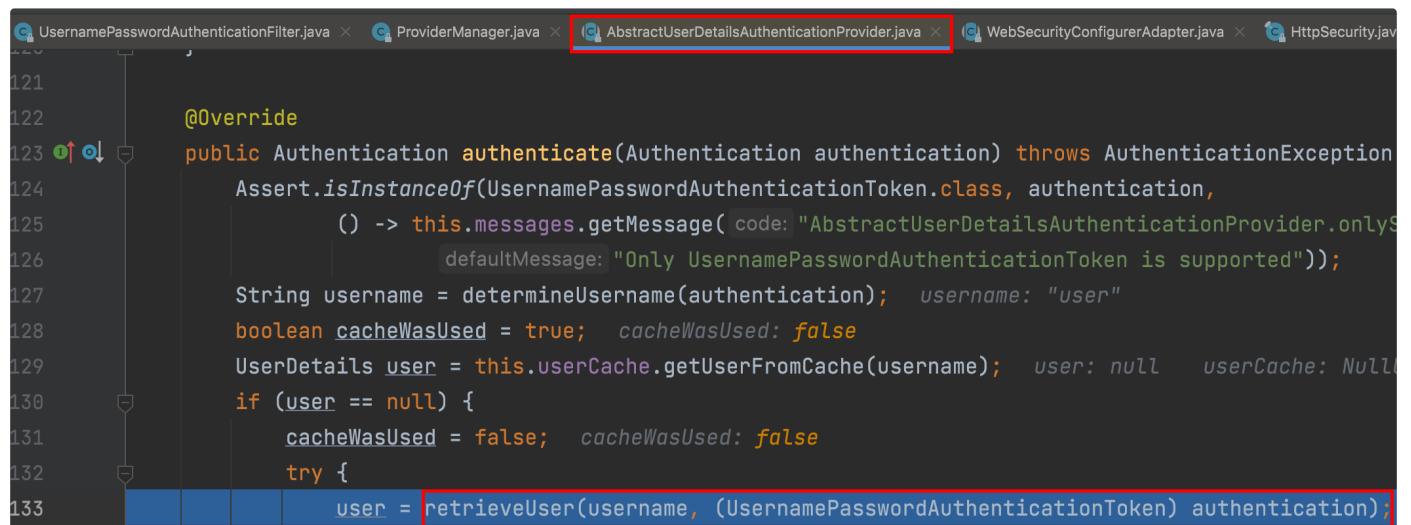
4. 调用 ProviderManager 类中方法 authenticate



```
197     }
198     if (result == null && this.parent != null) { result: null
199         // Allow the parent to try.
200         try {
201             parentResult = this.parent.authenticate(authentication); // This line is highlighted with a red box
202             result = parentResult;
203         }

```

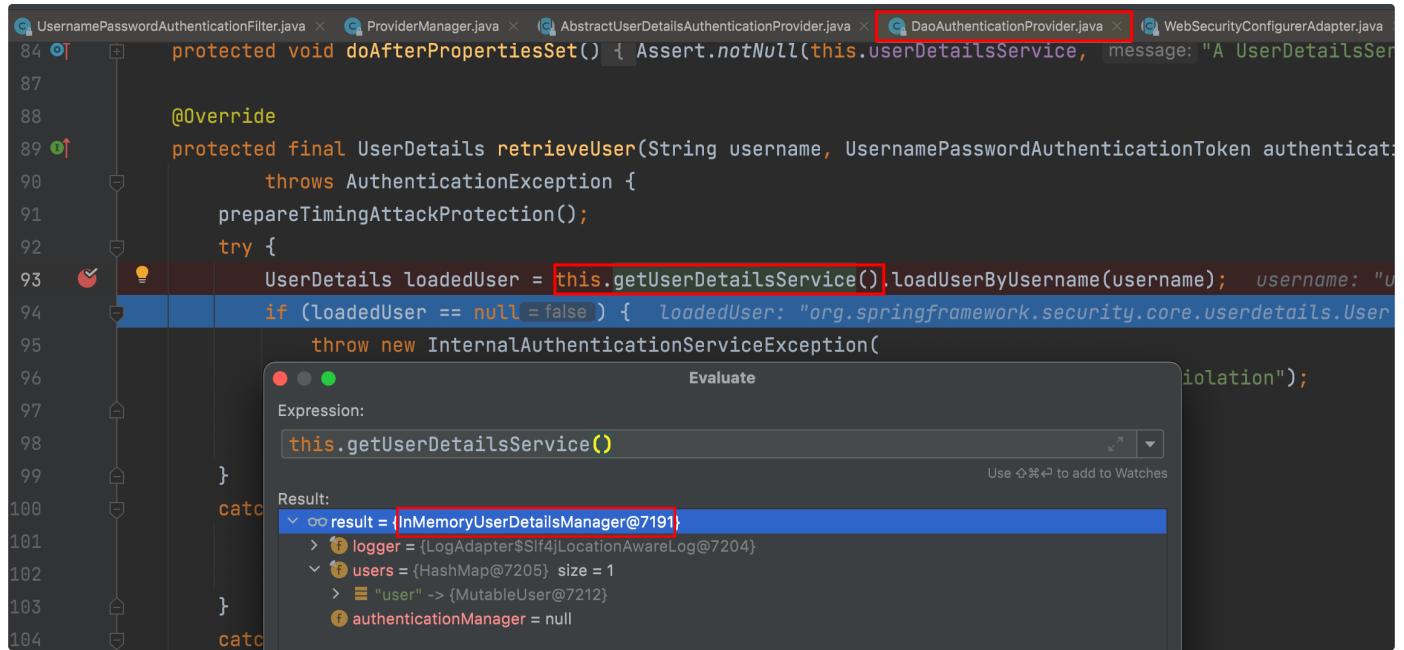
5. 调用了 ProviderManager 实现类中 AbstractUserDetailsAuthenticationProvider 类中方法



```
121
122     @Override
123     public Authentication authenticate(Authentication authentication) throws AuthenticationException
124         Assert.isInstanceOf(UsernamePasswordAuthenticationToken.class, authentication,
125             () -> this.messages.getMessage(code: "AbstractUserDetailsAuthenticationProvider.onlyS
126                                         defaultMessage: "Only UsernamePasswordAuthenticationToken is supported"));
127         String username = determineUsername(authentication); username: "user"
128         boolean cacheWasUsed = true; cacheWasUsed: false
129         UserDetails user = this.userCache.getUserFromCache(username); user: null userCache: Null
130         if (user == null) {
131             cacheWasUsed = false; cacheWasUsed: false
132             try {
133                 user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication), // This line is highlighted with a red box

```

6. 最终调用实现类 DaoAuthenticationProvider 类中方法比较



```
protected void doAfterPropertiesSet() { Assert.notNull(this.userDetailsService, message: "A UserDetailsService must be specified"); }

@Override
protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {
    prepareTimingAttackProtection();
    try {
        UserDetails loadedUser = this.getUserDetailsService().loadUserByUsername(username);
        if (loadedUser == null || !loadedUser.isAccountNonLocked()) {
            loadedUser: "org.springframework.security.core.userdetails.User@7191"
            throw new InternalAuthenticationServiceException("User account is locked");
        }
    } catch (Exception e) {
        logger.error("Error retrieving user from userDetailsService", e);
        throw new InternalAuthenticationServiceException("Error retrieving user from userDetailsService");
    }
}
```

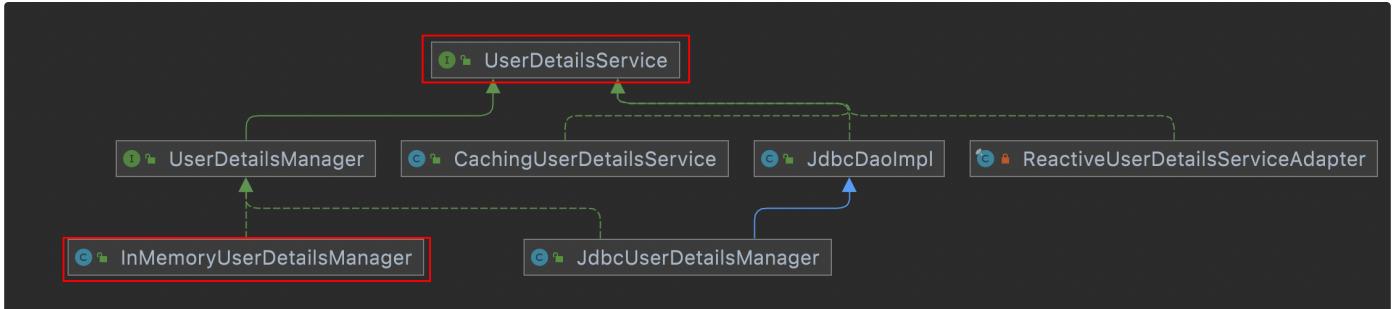
```
result = {InMemoryUserDetailsManager@7248}
> f logger = {LogAdapter$Slf4jLocationAwareLog@7266}
< f users = {HashMap@7267} size = 1
  <= "user" -> {MutableUser@7272}
    > = key = "user"
    <= value = {MutableUser@7272}
      > f password = "{noop}0b7a5c32-d3f3-4571-b394-087f423f0948"
      > f delegate = {User@7274} "org.springframework.security.core.userdetails.User [Username=us... View
```

看到这里就知道默认实现是基于 `InMemoryUserDetailsService` 这个类，也就是内存的实现！

UserDetailsService

通过刚才源码分析也能得知 `UserDetailsService` 是顶层父接口，接口中 `loadUserByUsername` 方法是用来在认证时进行用户名认证方法，默认实现使用是内存实现，如果想要修改数据库实现我们只需要自定义 `UserDetailsService` 实现，最终返回 `UserDetails` 实例即可。

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



UserDetailServiceAutoConfiguration

这个源码非常多，这里梳理了关键部分：

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(AuthenticationManager.class)
@ConditionalOnBean(ObjectPostProcessor.class)
@ConditionalOnMissingBean(
    value = { AuthenticationManager.class,
    AuthenticationProvider.class, UserDetailsService.class,
    AuthenticationManagerResolver.class },
    type = { "org.springframework.security.oauth2.jwt.JwtDecoder",
    "org.springframework.security.oauth2.server.resource.introspection.
    OpaqueTokenIntrospector" },
    "org.springframework.security.oauth2.client.registration.ClientReg
    istrationRepository" })
public class UserDetailsServiceAutoConfiguration {

    //.....
    @Bean
    @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());
    }
    // ...
}
```

结论

- 从自动配置源码中得知当 classpath 下存在 AuthenticationManager 类
- 当前项目中，系统没有提供 AuthenticationManager.class、AuthenticationProvider.class、UserDetailsService.class、AuthenticationManagerResolver.class、实例

默认情况下都会满足，此时 Spring Security 会提供一个 InMemoryUserDetailManager 实例

```
@Bean
@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());
}
```

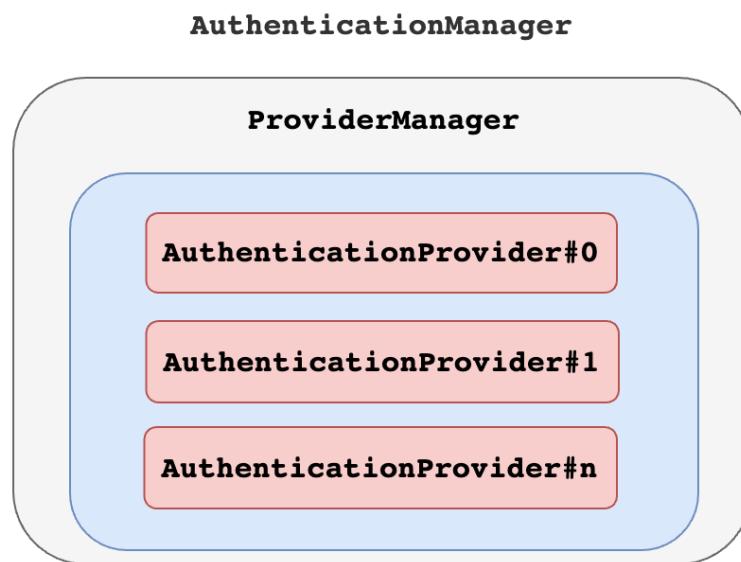
```
@ConfigurationProperties(prefix = "spring.security")
public class SecurityProperties {
    private final User user = new User();
    public User getUser() {
        return this.user;
    }
    //....
    public static class User {
        private String name = "user";
        private String password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

这就是默认生成 user 以及 uuid 密码过程！另外看明白源码之后，就知道只要在配置文件中加入如下配置可以对内存中用户和密码进行覆盖。

```
spring.security.user.name=root  
spring.security.user.password=root  
spring.security.user.roles=admin,users
```

总结

- AuthenticationManager、ProviderManger、以及 AuthenticationProvider 关系



- WebSecurityConfigurerAdapter 扩展 Spring Security 所有默认配置

```
(C) WebSecurityConfigurerAdapter
(m) ? WebSecurityConfigurerAdapter()
(m) ? WebSecurityConfigurerAdapter(boolean)
(m) 🔒 applyDefaultConfiguration(HttpSecurity): void
(m) ? authenticationManager(): AuthenticationManager
(m) 🟢 authenticationManagerBean(): AuthenticationManager
(m) ? configure(AuthenticationManagerBuilder): void
(m) ? configure(HttpSecurity): void
(m) 🟢 configure(WebSecurity): void ↑SecurityConfigurer
(m) 🔒 createSharedObjects(): Map<Class<?>, Object>
(m) ? getApplicationContext(): ApplicationContext
(m) 🔒 getAuthenticationEventPublisher(): AuthenticationEventPublisher
(m) ? getHttp(): HttpSecurity
(m) 🟢 init(WebSecurity): void ↑SecurityConfigurer
(m) 🟢 setApplicationContext(ApplicationContext): void
(m) 🟢 setAuthenticationConfiguration(AuthenticationConfiguration): void
(m) 🟢 setContentNegotiationStrategy(ContentNegotiationStrategy): void
(m) 🟢 setObjectPostProcessor(ObjectPostProcessor<Object>): void
(m) 🟢 setTrustResolver(AuthenticationTrustResolver): void
(m) ? userDetailsService(): UserDetailsService
(m) 🟢 userDetailsServiceBean(): UserDetailsService
```

- **UserDetailService** 用来修改默认认证的数据源信息

```
public interface UserDetailsService {

    /**
     * Locates the user based on the username. In the actual implementation, the search may possibly
     * be case sensitive, or case insensitive depending on how the implementation instance is
     * configured. In this case, the UserDetailsService object that comes back may have a username that is of
     * a different case than what was actually requested..
     *
     * Params: username – the username identifying the user whose data is required.
     * Returns: a fully populated user record (never null)
     * Throws: UsernameNotFoundException – if the user could not be found or the user has no
     * GrantedAuthority
     */
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}
```

第三章-第四章 认证原理&自定义认证

- 认证配置
- 表单认证
- 注销登录
- 前后端分离认证
- 添加验证码

自定义认证

自定义资源权限规则

- /index 公共资源
- /hello 受保护资源 权限管理

在项目中添加如下配置就可以实现对资源权限规则设定：

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .mvcMatchers("/index").permitAll()
            .anyRequest().authenticated()
            .and().formLogin();
    }
}
```

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .mvcMatchers( ...mvcPatterns: "/index").permitAll()
            .anyRequest().authenticated()
            .and().formLogin();
    }
}
```

说明

- `permitAll()` 代表放行该资源，该资源为公共资源 无需认证和授权可以直接访问
- `anyRequest().authenticated()` 代表所有请求，必须认证之后才能访问
- `formLogin()` 代表开启表单认证

注意：放行资源必须放在所有认证请求之前！

自定义登录界面

- 引入模板依赖

```
<!—thymeleaf—>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

- 定义登录页面 controller

```
@Controller
public class LoginController {

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

- 在 templates 中定义登录界面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>登录</title>
</head>
<body>
    <h1>用户登录</h1>
    <form method="post" th:action="@{/doLogin}">
        用户名:<input name="uname" type="text"/><br>
        密码:<input name="passwd" type="password"/><br>
        <input type="submit" value="登录"/>
    </form>
</body>
</html>
```

需要注意的是

- 登录表单 method 必须为 post , action 的请求路径为 /doLogin
 - 用户名的 name 属性为 uname
 - 密码的 name 属性为 passwd
- 配置 Spring Security 配置类

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
            .mvcMatchers("/login.html").permitAll()
            .mvcMatchers("/index").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login.html")
            .loginProcessingUrl("/doLogin")
            .usernameParameter("uname")
            .passwordParameter("passwd")
            .successForwardUrl("/index")      //forward 跳转
        注意:不会跳转到之前请求路径
        // .defaultSuccessUrl("/index")      //redirect 重定
        向    注意:如果之前请求路径,会有优先跳转之前请求路径
        .failureUrl("/login.html")
        .and()
        .csrf().disable(); //这里先关闭 CSRF
    }
}
```

- successForwardUrl 、 defaultSuccessUrl 这两个方法都可以实现成功之后跳转
 - successForwardUrl 默认使用 forward 跳转 注意:不会跳转到之前请求路径
 - defaultSuccessUrl 默认使用 redirect 跳转 注意:如果之前

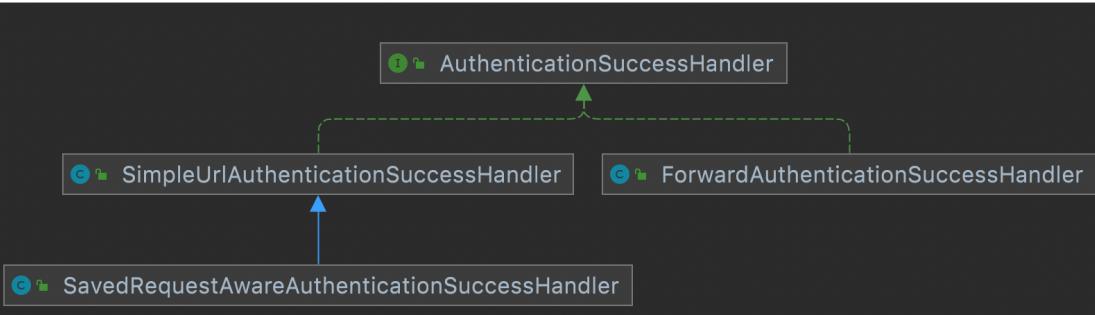
请求路径，会有优先跳转之前请求路径，可以传入第二个参数进行修改

自定义登录成功处理

有时候页面跳转并不能满足我们，特别是在前后端分离开发中就不需要成功之后跳转页面。只需要给前端返回一个 JSON 通知登录成功还是失败与否。这个时候可以通过自定义 `AuthenticationSuccessHandler` 实现

```
public interface AuthenticationSuccessHandler {  
  
    /**  
     * Called when a user has been successfully authenticated.  
     * @param request the request which caused the successful  
     * authentication  
     * @param response the response  
     * @param authentication the <tt>Authentication</tt> object  
     * which was created during  
     * the authentication process.  
     */  
    void onAuthenticationSuccess(HttpServletRequest request,  
        HttpServletResponse response,  
        Authentication authentication) throws IOException,  
        ServletException;  
}
```

根据接口的描述信息，也可以得知登录成功会自动回调这个方法，进一步查看它的默认实现，你会发现 `successForwardUrl`、`defaultSuccessUrl` 也是由它的子类实现的



- 自定义 `AuthenticationSuccessHandler` 实现

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

- 配置 AuthenticationSuccessHandler

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            // ...
            .and()
            .formLogin()
            //....
            .successHandler(new
MyAuthenticationSuccessHandler())
            .failureUrl("/login.html")
            .and()
            .csrf().disable(); //这里先关闭 CSRF
    }
}
```

```
localhost:8080/doLogin
1 // 20220113062635
2 // http://localhost:8080/doLogin
3
4 {
5     "msg": "登录成功",
6     "status": 200
7 }
```

显示登录失败信息

为了能更直观在登录页面看到异常错误信息，可以在登录页面中直接获取异常信息。Spring Security 在登录失败之后会将异常信息存储到 `request`、`session` 作用域中 key 为 `SPRING_SECURITY_LAST_EXCEPTION` 命名属性中，源码可以参考 `SimpleUrlAuthenticationFailureHandler`：

```
@Override
public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,
    AuthenticationException exception) throws IOException, ServletException {
    if (this.defaultFailureUrl == null) {
        if (this.logger.isTraceEnabled()) {
            this.logger.trace("Sending 401 Unauthorized error since no failure URL is set");
        }
        else {
            this.logger.debug("Sending 401 Unauthorized error");
        }
        response.sendError(HttpStatus.UNAUTHORIZED.value(), HttpStatus.UNAUTHORIZED.getReasonPhrase());
        return;
    }
    saveException(request, exception);
    if (this.forwardToDestination) {
        this.logger.debug("Forwarding to " + this.defaultFailureUrl);
        request.getRequestDispatcher(this.defaultFailureUrl).forward(request, response);
    }
    else {
        this.redirectStrategy.sendRedirect(request, response, this.defaultFailureUrl);
    }
}

Caches the AuthenticationException for use in view rendering.
If forwardToDestination is set to true, request scope will be used, otherwise the exception will be stored in the session. If there is no session and allowSessionCreation is true, a new session will be created. Otherwise the exception will not be stored.
protected final void saveException(HttpServletRequest request, AuthenticationException exception) {
    if (this.forwardToDestination) {
        request.setAttribute(WebAttributes.AUTHENTICATION_EXCEPTION, exception);
    }
}
```

org.springframework.security.web.WebAttributes
public static final String AUTHENTICATION_EXCEPTION = "SPRING_SECURITY_LAST_EXCEPTION";
Used to cache an authentication-failure exception in the session.
See Also: org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler
Maven: org.springframework.security:spring-security-web:5.6.1

- 显示异常信息

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>登录</title>
</head>
<body>
    ....
    <div th:text="${SPRING_SECURITY_LAST_EXCEPTION}"></div>
</body>
</html>
```

- 配置

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
            //..
            .and()
            .formLogin()
            //....
            //.failureUrl("/login.html")
            .failureForwardUrl("/login.html")
            .and()
            .csrf().disable(); //这里先关闭 CSRF
    }
}
```

- failureUrl、failureForwardUrl 关系类似于之前提到的 successForwardUrl 、 defaultSuccessUrl 方法
 - failureUrl 失败以后的重定向跳转

- `failureForwardUrl` 失败以后的 `forward` 跳转 注意：因此获取 `request` 中异常信息，这里只能使用`failureForwardUrl`

自定义登录失败处理

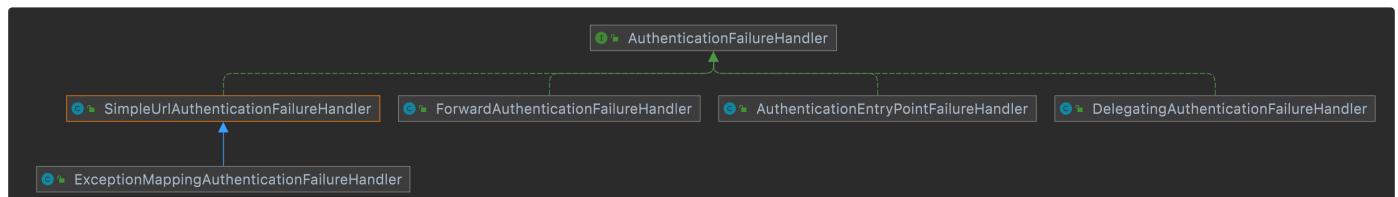
和自定义登录成功处理一样，Spring Security 同样为前后端分离开发提供了登录失败的处理，这个类就是 `AuthenticationFailureHandler`，源码为：

```
public interface AuthenticationFailureHandler {

    /**
     * Called when an authentication attempt fails.
     * @param request the request during which the authentication
     * attempt occurred.
     * @param response the response.
     * @param exception the exception which was thrown to reject the
     * authentication
     * @param request.
     */
    void onAuthenticationFailure(HttpServletRequest request,
        HttpServletResponse response,
        AuthenticationException exception) throws IOException,
        ServletException;

}
```

根据接口的描述信息，也可以得知登录失败会自动回调这个方法，进一步查看它的默认实现，你会发现`failureUrl`、`failureForwardUrl`也是由它的子类实现的。



- 自定义 `AuthenticationFailureHandler` 实现

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

- 配置 AuthenticationFailureHandler

```
@Configuration  
public class WebSecurityConfigurer extends  
WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http.authorizeHttpRequests()  
            // ...  
            .and()  
            .formLogin()  
            // ..  
            .failureHandler(new  
MyAuthenticationFailureHandler())  
            .and()  
            .csrf().disable(); // 这里先关闭 CSRF  
    }  
}
```

```
localhost:8080/doLogin
1 // 20220113062606
2 // http://localhost:8080/doLogin
3
4 {
5     "msg": "登录失败：用户名或密码错误",
6     "status": 500
7 }
```

注销登录

Spring Security 中也提供了默认的注销登录配置，在开发时也可以按照自己需求对注销进行个性化定制。

- 开启注销登录 **默认开启**

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws
Exception {
    http.authorizeHttpRequests()
        // ...
        .and()
        .formLogin()
        // ...
        .and()
        .logout()
        .logoutUrl("/logout")
        .invalidateHttpSession(true)
        .clearAuthentication(true)
        .logoutSuccessUrl("/login.html")
        .and()
        .csrf().disable(); // 这里先关闭 CSRF
    }
}
```

- 通过 `logout()` 方法开启注销配置
- `logoutUrl` 指定退出登录请求地址，默认是 GET 请求，路径为 `/logout`

- `invalidateHttpSession` 退出时是否是 session 失效，默认值为 true
 - `clearAuthentication` 退出时是否清除认证信息，默认值为 true
 - `logoutSuccessUrl` 退出登录时跳转地址
- 配置多个注销登录请求

如果项目中有需要，开发者还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 WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
            // ...
            .and()
            .formLogin()
            // ...
            .and()
            .logout()
            .logoutRequestMatcher(new OrRequestMatcher(
                new
                AntPathRequestMatcher("/logout1", "GET"),
                new
                AntPathRequestMatcher("/logout", "GET")
            ))
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .logoutSuccessUrl("/login.html")
            .and()
            .csrf().disable(); // 这里先关闭 CSRF
    }
}
```

- 前后端分离注销登录配置

如果是前后端分离开发，注销成功之后就不需要页面跳转了，只需要将注销成功的信息返回前端即可，此时我们可以通过自定义 `LogoutSuccessHandler` 实现来返回注销之后信息：

```
public class MyLogoutSuccessHandler implements  
LogoutSuccessHandler {  
    @Override  
    public void onLogoutSuccess(HttpServletRequest request,  
HttpServletResponse response, Authentication authentication)  
throws IOException, ServletException {  
    Map<String, Object> result = new HashMap<String,  
Object>();  
    result.put("msg", "注销成功");  
    result.put("status", 200);  
    response.setContentType("application/json;charset=UTF-  
8");  
    String s = new  
ObjectMapper().writeValueAsString(result);  
    response.getWriter().println(s);  
}  
}
```

```
@Configuration  
public class WebSecurityConfigurer extends  
WebSecurityConfigurerAdapter {  
    @Override  
    protected void configure(HttpSecurity http) throws  
Exception {  
        http.authorizeHttpRequests()  
            //....  
            .and()  
            .formLogin()  
            // ...  
            .and()  
            .logout()  
            // .logoutUrl("/logout")  
            .logoutRequestMatcher(new OrRequestMatcher(
```

```

        new
AntPathRequestMatcher("/logout1", "GET"),
        new
AntPathRequestMatcher("/logout", "GET")
))
.invalidateHttpSession(true)
.clearAuthentication(true)
//.logoutSuccessUrl("/login.html")
.logoutSuccessHandler(new
MyLogoutSuccessHandler())
.and()
.csrf().disable(); //这里先关闭 CSRF
}
}

```

```

← → C ⌂ ⓘ localhost:8080/logout
1 // 20220113114124
2 // http://localhost:8080/logout
3
4 {
5   "msg": "注销成功",
6   "status": 200
7 }

```

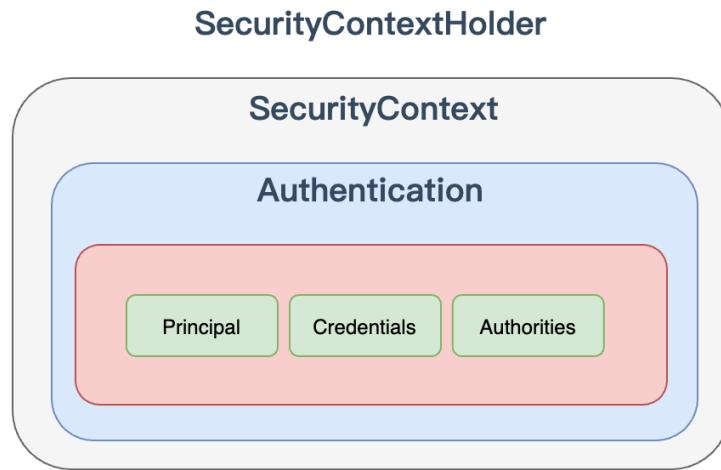
登录用户数据获取

SecurityContextHolder

Spring Security 会将登录用户数据保存在 Session 中。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。

实际上 `SecurityContextHolder` 中存储是 `SecurityContext`, 在 `SecurityContext` 中存储是 `Authentication`。



这种设计是典型的策略设计模式：

```
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";  
    private static final String MODE_PRE_INITIALIZED =  
"MODE_PRE_INITIALIZED";  
    private static SecurityContextHolderStrategy strategy;  
    //....  
    private static void initializeStrategy() {  
        if (MODE_PRE_INITIALIZED.equals(strategyName)) {  
            Assert.state(strategy != null, "When using " +  
MODE_PRE_INITIALIZED  
                + ", setContextHolderStrategy must be called with the  
fully constructed strategy");  
            return;  
        }  
        if (!StringUtils.hasText(strategyName)) {  
            // Set default  
            strategyName = MODE_THREADLOCAL;  
        }  
        if (strategyName.equals(MODE_THREADLOCAL)) {
```

```

        strategy = new ThreadLocalSecurityContextHolderStrategy();
        return;
    }
    if (strategyName.equals(MODE_INHERITABLETHREADLOCAL)) {
        strategy = new
InheritableThreadLocalSecurityContextHolderStrategy();
        return;
    }
    if (strategyName.equals(MODE_GLOBAL)) {
        strategy = new GlobalSecurityContextHolderStrategy();
        return;
    }
    //.....
}
}

```

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 JavaWeb 开发中，这种模式很少使用到。

SecurityContextHolderStrategy

通过 `SecurityContextHolder` 可以得知，`SecurityContextHolderStrategy` 接口用来定义存储策略方法

```

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：

- `clearContext`: 该方法用来清除存储的 `SecurityContext` 对象。
- `getContext`: 该方法用来获取存储的 `SecurityContext` 对象。
- `setContext`: 该方法用来设置存储的 `SecurityContext` 对象。
- `create Empty Context`: 该方法则用来创建一个空的 `SecurityContext` 对象。



从上面可以看出每一个实现类对应一种策略的实现。

代码中获取认证之后用户数据

```
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello() {
        Authentication authentication = SecurityContextHolder
            .getContext().getAuthentication();
        User principal = (User) authentication.getPrincipal();
        System.out.println("身份 :" + principal.getUsername());
        System.out.println("凭证 :" + authentication.getCredentials());
        System.out.println("权限 :" + authentication.getAuthorities());
        return "hello security";
    }
}
```

多线程情况下获取用户数据

```
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello() {
        new Thread(()→{
            Authentication authentication = SecurityContextHolder
                .getContext().getAuthentication();
```

```

        User principal = (User) authentication.getPrincipal();
        System.out.println("身份 :" + principal.getUsername());
        System.out.println("凭证
:" + authentication.getCredentials());
        System.out.println("权限
:" + authentication.getAuthorities());
    }).start();
    return "hello security";
}
}

```

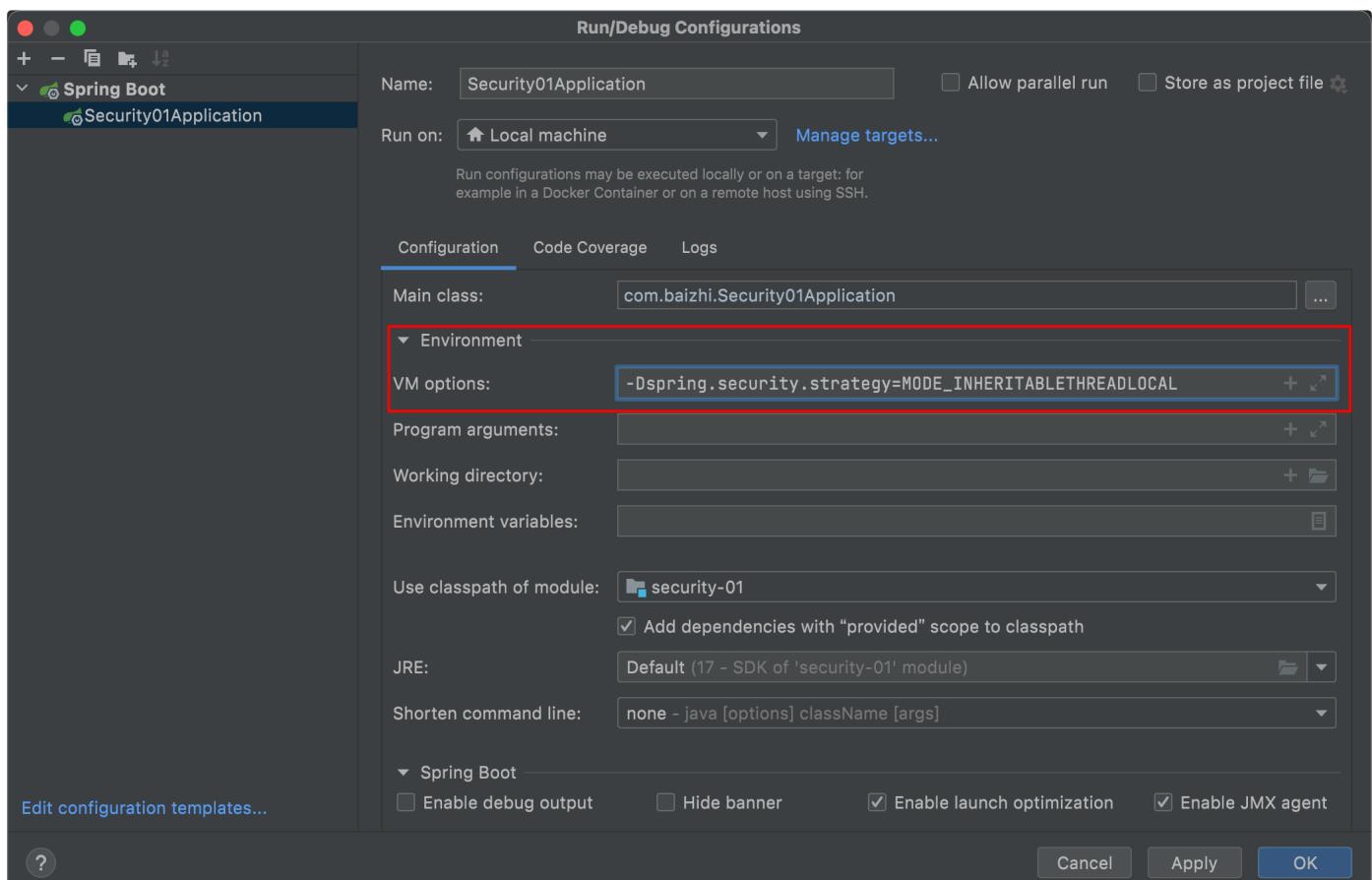
```

Exception in thread "Thread-1" java.lang.NullPointerException Create breakpoint : Cannot invoke
"org.springframework.security.core.Authentication.getPrincipal()" because "authentication" is null
at com.baizhi.controller.HelloController.lambda$hello$0(HelloController.java:17)
<1 internal line>

```

可以看到默认策略，是无法在子线程中获取用户信息，如果需要在子线程中获取必须使用第二种策略，默认策略是通过 `System.getProperty` 加载的，因此我们可以通过增加 `VM Options` 参数进行修改。

`-Dspring.security.strategy=MODE_INHERITABLETHREADLOCAL`



页面上获取用户信息

- 引入依赖

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

- 页面加入命名空间

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

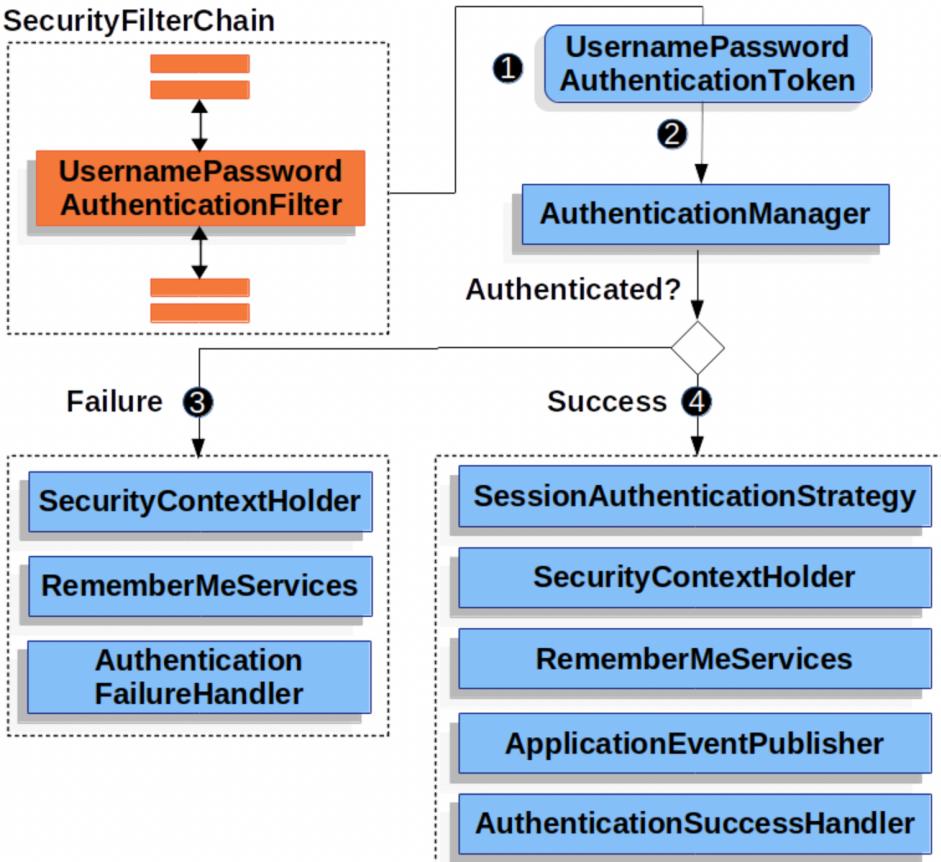
- 页面中使用

```
<!— 获取认证用户名 —>
<ul>
    <li sec:authentication="principal.username"></li>
    <li sec:authentication="principal.authorities"></li>
    <li sec:authentication="principal.accountNonExpired"></li>
    <li sec:authentication="principal.accountNonLocked"></li>
    <li sec:authentication="principal.credentialsNonExpired">
    </li>
</ul>
```

自定义认证数据源

认证流程分析

<https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html>



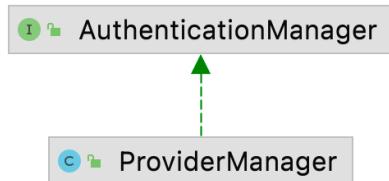
- 发起认证请求，请求中携带用户名、密码，该请求会被 `UsernamePasswordAuthenticationFilter` 拦截
- 在 `UsernamePasswordAuthenticationFilter` 的 `attemptAuthentication` 方法中将请求中用户名和密码，封装为 `Authentication` 对象，并交给 `AuthenticationManager` 进行认证
- 认证成功，将认证信息存储到 `SecurityContextHolder` 以及调用记住我等，并回调 `AuthenticationSuccessHandler` 处理
- 认证失败，清除 `SecurityContextHolder` 以及 记住我中信息，回调 `AuthenticationFailureHandler` 处理

三者关系

从上面分析中得知，`AuthenticationManager` 是认证的核心类，但实际上在底层真正认证时还离不开 `ProviderManager` 以及 `AuthenticationProvider`。他们三者关系是怎样的呢？

- `AuthenticationManager` 是一个认证管理器，它定义了 Spring Security 过滤器要执行认证操作。
- `ProviderManager` `AuthenticationManager` 接口的实现类。Spring Security 认证时默认使用就是 `ProviderManager`。
- `AuthenticationProvider` 就是针对不同的身份类型执行的具体的身份认证。

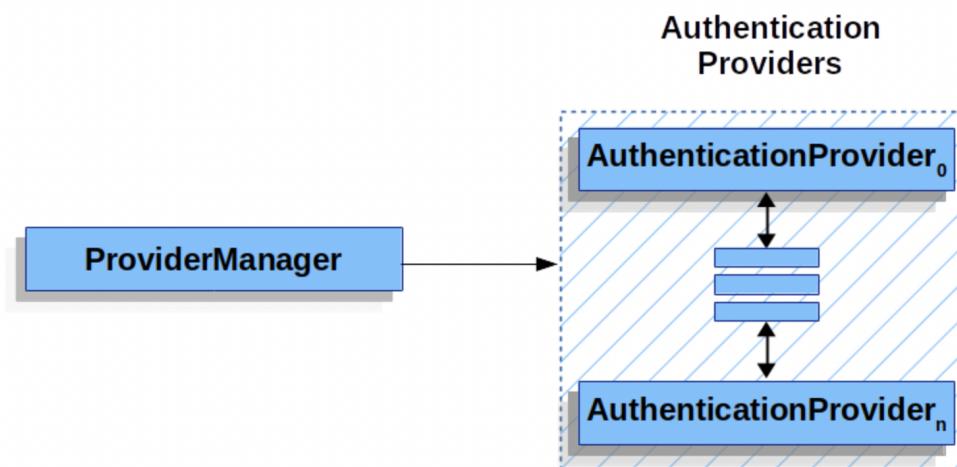
AuthenticationManager 与 ProviderManager



`ProviderManager` 是 `AuthenticationManager` 的唯一实现，也是 Spring Security 默认使用实现。从这里不难看出默认情况下 `AuthenticationManager` 就是一个 `ProviderManager`。

ProviderManager 与 AuthenticationProvider

摘自官方：<https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html>



在 Spring Security 中，允许系统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。

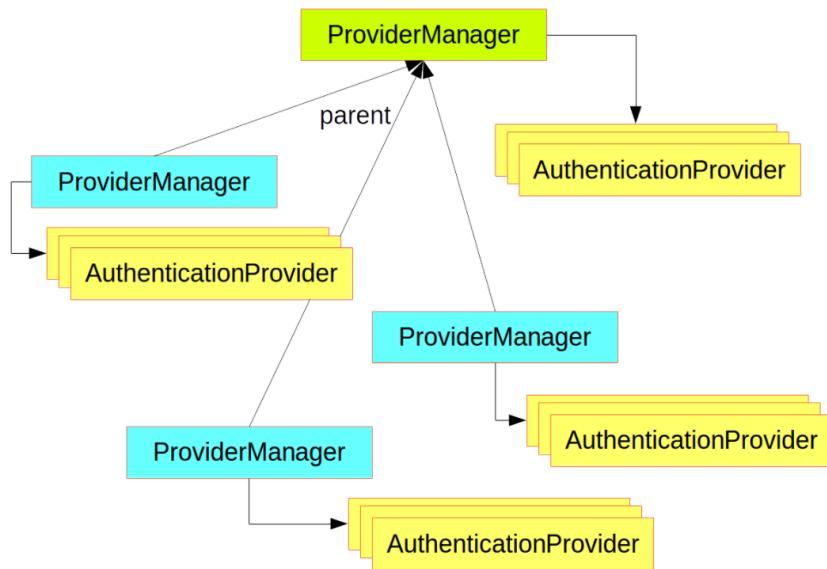
多个 `AuthenticationProvider` 将组成一个列表，这个列表将由 `ProviderManager` 代理。换句话说，在 `ProviderManager` 中存在一个 `AuthenticationProvider` 列表，在 `Provider Manager` 中遍历列表中的每一个 `AuthenticationProvider` 去执行身份认证，最终得到认证结果。

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。

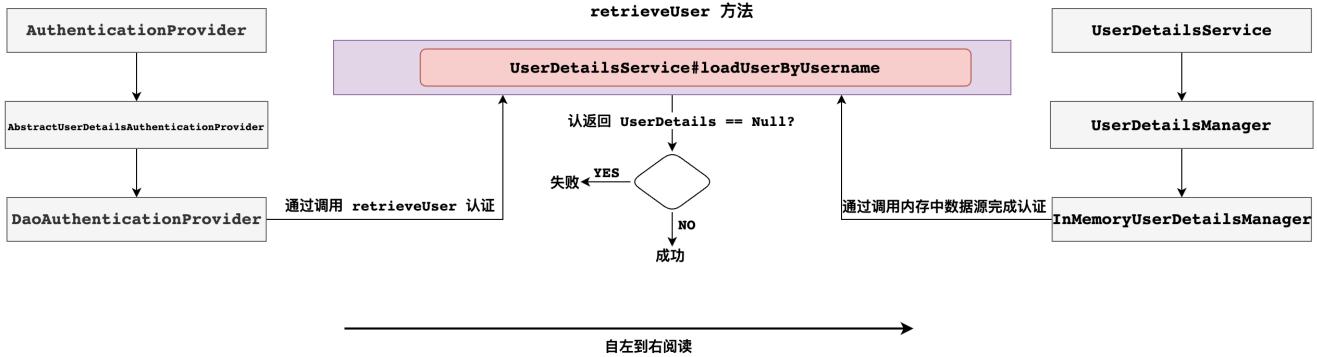
ProviderManager 本身也可以有多个，多个 ProviderManager 共用同一个 parent。有时，一个应用程序有受保护资源的逻辑组（例如，所有符合路径模式的网络资源，如 /api/**），每个组可以有自己的专用 AuthenticationManager。通常，每个组都是一个 ProviderManager，它们共享一个父级。然后，父级是一种 **全局** 资源，作为所有提供者的后备资源。

根据上面的介绍，我们绘出新的 AuthenticationManager、ProvideManager 和 AuthentictionProvider 关系

摘自官网：<https://spring.io/guides/topicals/spring-security-architecture>



弄清楚认证原理之后我们来看下具体认证时数据源的获取。默认情况下 AuthenticationProvider 是由 DaoAuthenticationProvider 类来实现认证的，在 DaoAuthenticationProvider 认证时又通过 UserDetailsService 完成数据源的校验。他们之间调用关系如下：



总结：AuthenticationManager 是认证管理器，在 Spring Security 中有全局 AuthenticationManager，也可以有局部AuthenticationManager。全局的 AuthenticationManager 用来对全局认证进行处理，局部的AuthenticationManager 用来对某些特殊资源认证处理。当然无论是全局认证管理器还是局部认证管理器都是由 ProviderManger 进行实现。每一个ProviderManger 中都代理一个 AuthenticationProvider 的列表，列表中每一个实现代表一种身份认证方式。认证时底层数据源需要调用 UserDetailsService 来实现。

配置全局 AuthenticationManager

<https://spring.io/guides/topicals/spring-security-architecture>

- 默认的全局 AuthenticationManager

```

@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Autowired
    public void initialize(AuthenticationManagerBuilder builder)
    {
        //builder..
    }
}

```

- springboot 对 security 进行自动配置时自动在工厂中创建一个全局 AuthenticationManager

总结

- 默认自动配置创建全局AuthenticationManager 默认找当前项目中是否存在自定义 UserDetails 实例 自动将当前项目 UserDetails 实例 设置为数据源
- 默认自动配置创建全局AuthenticationManager 在工厂中使用时直接在代码中

注入即可

- 自定义全局 AuthenticationManager

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Override
    public void configure(AuthenticationManagerBuilder builder) {
        //builder ....
    }
}
```

- 自定义全局 AuthenticationManager

总结

1. 一旦通过 configure 方法自定义 AuthenticationManager 实现 就会将工厂中自动配置 AuthenticationManager 进行覆盖
 2. 一旦通过 configure 方法自定义 AuthenticationManager 实现 需要在实现中指定认证数据源对象 UserDetailsService 实例
 3. 一旦通过 configure 方法自定义 AuthenticationManager 实现 这种方式创建 AuthenticationManager 对象工厂内部本地一个 AuthenticationManager 对象 不允许在其他自定义组件中进行注入
- 用来在工厂中暴露自定义 AuthenticationManager 实例

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {

    //1.自定义AuthenticationManager 推荐 并没有在工厂中暴露出来
    @Override
    public void configure(AuthenticationManagerBuilder builder)
throws Exception {
        System.out.println("自定义AuthenticationManager: " +
builder);
        builder.userDetailsService(userDetailsService());
    }
}
```

```
//作用：用来将自定义AuthenticationManager在工厂中进行暴露，可以在任何位置注入
@Override
@Bean
public AuthenticationManager authenticationManagerBean()
throws Exception {
    return super.authenticationManagerBean();
}
}
```

自定义内存数据源

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService(){
        InMemoryUserDetailsManager inMemoryUserDetailsManager
            = new InMemoryUserDetailsManager();
        UserDetails u1 = User.withUsername("zhangs")
            .password("{noop}111").roles("USER").build();
        inMemoryUserDetailsManager.createUser(u1);
        return inMemoryUserDetailsManager;
    }

    @Override
    protected void configure(AuthenticationBuilder auth)
        throws Exception {
        auth.userDetailsService(userDetailsService());
    }
}
```

自定义数据库数据源

- 设计表结构

```
-- 用户表
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 AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
-- 角色表
CREATE TABLE `role`(
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL,
  `name_zh` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
-- 用户角色关系表
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 AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;
```

- 插入测试数据

```
-- 插入用户数据
```

```

BEGIN;
    INSERT INTO `user`
VALUES (1, 'root', '{noop}123', 1, 1, 1, 1);
    INSERT INTO `user`
VALUES (2, 'admin', '{noop}123', 1, 1, 1, 1);
    INSERT INTO `user`
VALUES (3, 'blr', '{noop}123', 1, 1, 1, 1);
COMMIT;

-- 插入角色数据
BEGIN;
    INSERT INTO `role`
VALUES (1, 'ROLE_product', '商品管理员');
    INSERT INTO `role`
VALUES (2, 'ROLE_admin', '系统管理员');
    INSERT INTO `role`
VALUES (3, 'ROLE_user', '用户管理员');
COMMIT;

-- 插入用户角色数据
BEGIN;
    INSERT INTO `user_role`
VALUES (1, 1, 1);
    INSERT INTO `user_role`
VALUES (2, 1, 2);
    INSERT INTO `user_role`
VALUES (3, 2, 2);
    INSERT INTO `user_role`
VALUES (4, 3, 3);
COMMIT;

```

- 项目中引入依赖

```

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.0</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>

```

```
<artifactId>mysql-connector-java</artifactId>
<version>5.1.38</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.7</version>
</dependency>
```

- 配置 springboot 配置文件

```
# datasource
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/security?
characterEncoding=UTF-8&useSSL=false
spring.datasource.username=root
spring.datasource.password=root

# mybatis
mybatis.mapper-locations=classpath:com/baizhi/mapper/*.xml
mybatis.type-aliases-package=com.baizhi.entity

# log
logging.level.com.baizhi=debug
```

- 创建 entity

- 创建 user 对象

```
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<GrantedAuthority> grantedAuthorities = new
ArrayList<>();
    roles.forEach(role→grantedAuthorities.add(new
SimpleGrantedAuthority(role.getName())));
    return grantedAuthorities;
}

    @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;
    }
    //get/set....
}
```

- 创建 role 对象

```
public class Role {
    private Integer id;
    private String name;
    private String nameZh;
    //get set..
}
```

- 创建 UserDao 接口

```
@Mapper
public interface UserDao {
    //根据用户名查询用户
    User loadUserByUsername(String username);

    //根据用户id查询角色
    List<Role> getRolesByUid(Integer uid);
}
```

- 创建 UserMapper 实现

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.baizhi.dao.UserDao">
    ←!—查询单个→
    <select id="loadUserByUsername" resultType="User">
        select id,
               username,
               password,
               enabled,
               accountNonExpired,
```

```

        accountNonLocked,
        credentialsNonExpired
    from user
    where username = #{username}
</select>

<!--—查询指定行数据—
&lt;select id="getRolesByUid" resultType="Role"&gt;
    select r.id,
        r.name,
        r.name_zh nameZh
    from role r,
        user_role ur
    where r.id = ur.rid
        and ur.uid = #{uid}
&lt;/select&gt;
&lt;/mapper&gt;
</pre>

```

- 创建 UserDetailsService 实例

```

@Component
public class MyUserDetailsService implements UserDetailsService {
    private final UserDao userDao;

    @Autowired
    public MyUserDetailsService(UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        User user = userDao.loadUserByUsername(username);
        if( ObjectUtils.isEmpty(user))throw new
        RuntimeException("用户不存在");
        user.setRoles(userDao.getRolesByUid(user.getId()));
    }
}

```

```
        return user;
    }
}
```

- 配置 authenticationManager 使用自定义UserDetailsService

```
@Configuration
public class WebSecurityConfigurer extends
WebSecurityConfigurerAdapter {

    private final UserDetailsService userDetailsService;

    @Autowired
    public WebSecurityConfigurer(UserDetailsService
userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder
builder) throws Exception {
        builder.userDetailsService(userDetailsService);
    }

    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        //web security..
    }
}
```

- 启动测试即可
-

添加认证验证码

配置验证码

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

```
@Configuration
public class KaptchaConfig {
    @Bean
    public 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;
    }
}
```

传统 web 开发

- 生成验证码 controller

```
@Controller
public class KaptchaController {
    private final Producer producer;

    @Autowired
```

```

public KaptchaController(Producer producer) {
    this.producer = producer;
}

@GetMapping("/vc.jpg")
public void getVerifyCode(HttpServletRequest response,
HttpSession session) throws IOException {
    response.setContentType("image/png");
    String code = producer.createText();
    session.setAttribute("kaptcha", code); //可以更换成 redis
实现
    BufferedImage bi = producer.createImage(code);
    ServletOutputStream os = response.getOutputStream();
    ImageIO.write(bi, "jpg", os);
}
}

```

- 自定义验证码异常类

```

public class KaptchaNotMatchException extends
AuthenticationException {

    public KaptchaNotMatchException(String msg) {
        super(msg);
    }

    public KaptchaNotMatchException(String msg, Throwable
cause) {
        super(msg, cause);
    }
}

```

- 自定义filter验证验证码

```

public class KaptchaFilter extends
UsernamePasswordAuthenticationFilter {

    public static final String KAPTCHA_KEY = "kaptcha"; //默认值
}

```

```

private String kaptcha = KAPTCHA_KEY;

public String getKaptcha() {
    return kaptcha;
}

public void setKaptcha(String kaptcha) {
    this.kaptcha = kaptcha;
}

@Override
public Authentication
attemptAuthentication(HttpServletRequest request,
HttpServletResponse response) throws AuthenticationException {
    //1.判断是否是 post 方式
    if (request.getMethod().equals("POST")) {
        throw new
AuthenticationServiceException("Authentication method not
supported: " + request.getMethod());
    }
    //2.获取验证码
    String kaptcha = request.getParameter(getKaptcha());
    String sessionKaptcha = (String)
request.getSession().getAttribute("kaptcha");
    if (!ObjectUtils.isEmpty(kaptcha) &&
!ObjectUtils.isEmpty(sessionKaptcha) &&
        kaptcha.equalsIgnoreCase(sessionKaptcha)) {
        return super.attemptAuthentication(request,
response);
    }
    throw new KaptchaNotMatchException("验证码输入错误!");
}
}

```

- 放行以及配置验证码 filter

```
@Configuration
```

```
public class WebSecurityConfigurer extends  
WebSecurityConfigurerAdapter {  
  
    private final UserDetailsService userDetailsService;  
  
    @Autowired  
    public WebSecurityConfigurer(UserDetailsService  
userDetailsService) {  
        this.userDetailsService = userDetailsService;  
    }  
  
    @Override  
    protected void configure(AuthenticationManagerBuilder  
builder) throws Exception {  
        builder.userDetailsService(userDetailsService);  
    }  
  
    @Override  
    @Bean  
    public AuthenticationManager authenticationManagerBean()  
throws Exception {  
        return super.authenticationManagerBean();  
    }  
  
    @Bean  
    public KaptchaFilter kaptchaFilter() throws Exception {  
        KaptchaFilter kaptchaFilter = new KaptchaFilter();  
        //指定接收验证码请求参数名  
        kaptchaFilter.setKaptcha("kaptcha");  
        //指定认证管理器  
  
        kaptchaFilter.setAuthenticationManager(authenticationManagerBe  
an());  
        //指定认证成功和失败处理  
        kaptchaFilter.setAuthenticationSuccessHandler(new  
MyAuthenticationSuccessHandler());  
        kaptchaFilter.setAuthenticationFailureHandler(new  
MyAuthenticationFailureHandler());
```

```

    // 指定处理登录
    kaptchaFilter.setFilterProcessesUrl("/doLogin");
    kaptchaFilter.setUsernameParameter("uname");
    kaptchaFilter.setPasswordParameter("passwd");
    return kaptchaFilter;
}

@Override
protected void configure(HttpSecurity http) throws
Exception {
    http.authorizeHttpRequests()
        .mvcMatchers("/vc.jpg").permitAll()
        .mvcMatchers("/login.html").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .LoginPage("/login.html")
        ...
    http.addFilterAt(kaptchaFilter(),
UsernamePasswordAuthenticationFilter.class);
}
}

```

- 登录页面添加验证码

```

<form method="post" th:action="@{/doLogin}">
    用户名:<input name="uname" type="text"/><br>
    密码:<input name="passwd" type="password"/><br>
    验证码: <input name="kaptcha" type="text"/> <img alt="">
    th:src="@{/vc.jpg}"><br>
    <input type="submit" value="登录"/>
</form>

```

前后端分离开发

- 生成验证码 controller

```
@RestController
public class KaptchaController {
    private final Producer producer;

    @Autowired
    public KaptchaController(Producer producer) {
        this.producer = producer;
    }

    @GetMapping("/vc.png")
    public String getVerifyCode(HttpSession session) throws
IOException {
        //1.生成验证码
        String code = producer.createText();
        session.setAttribute("kaptcha", code); //可以更换成 redis
实现
        BufferedImage bi = producer.createImage(code);
        //2.写入内存
        FastByteArrayOutputStream fos = new
FastByteArrayOutputStream();
        ImageIO.write(bi, "png", fos);
        //3.生成 base64
        return Base64.encodeBase64String(fos.toByteArray());
    }
}
```

- 定义验证码异常类

```
public class KaptchaNotMatchException extends  
AuthenticationException {  
  
    public KaptchaNotMatchException(String msg) {  
        super(msg);  
    }  
  
    public KaptchaNotMatchException(String msg, Throwable  
cause) {  
        super(msg, cause);  
    }  
}
```

- 在自定义LoginKaptchaFilter中加入验证码验证

```
//自定义 filter  
public class LoginKaptchaFilter extends  
UsernamePasswordAuthenticationFilter {  
  
    public static final String FORM_KAPTCHA_KEY = "kaptcha";  
  
    private String kaptchaParameter = FORM_KAPTCHA_KEY;  
  
    public String getKaptchaParameter() {  
        return kaptchaParameter;  
    }  
  
    public void setKaptchaParameter(String kaptchaParameter) {  
        this.kaptchaParameter = kaptchaParameter;  
    }  
  
    @Override  
    public Authentication  
attemptAuthentication(HttpServletRequest request,  
HttpServletResponse response) throws AuthenticationException {  
    if (!request.getMethod().equals("POST")) {
```

```
        throw new  
AuthenticationServiceException("Authentication method not  
supported: " + request.getMethod());  
    }  
    try {  
        //1.获取请求数据  
        Map<String, String> userInfo = new  
ObjectMapper().readValue(request.getInputStream(), Map.class);  
        String captcha =  
userInfo.get(getKaptchaParameter());//用来获取数据中验证码  
        String username =  
userInfo.get(getUsernameParameter());//用来接收用户名  
        String password =  
userInfo.get(getPasswordParameter());//用来接收密码  
        //2.获取 session 中验证码  
        String sessionVerifyCode = (String)  
request.getSession().getAttribute("kaptcha");  
        if (!ObjectUtils.isEmpty(captcha) &&  
!ObjectUtils.isEmpty(sessionVerifyCode) &&  
  
captcha.equalsIgnoreCase(sessionVerifyCode)) {  
            //3.获取用户名 和密码认证  
            UsernamePasswordAuthenticationToken authRequest  
= new UsernamePasswordAuthenticationToken(username, password);  
            setDetails(request, authRequest);  
            return  
this.getAuthenticationManager().authenticate(authRequest);  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    throw new KaptchaNotMatchException("验证码不匹配!");  
}
```

- 配置

```
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {

    //自定义内存数据源
    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager =
new InMemoryUserDetailsManager();

        inMemoryUserDetailsManager.createUser(User.withUsername("root")
).password("{noop}123").roles("admin").build());
        return inMemoryUserDetailsManager;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.userDetailsService(userDetailsService());
    }

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean()
throws Exception {
        return super.authenticationManagerBean();
    }

    //配置
    @Bean
    public LoginKaptchaFilter loginKaptchaFilter() throws
Exception {
        LoginKaptchaFilter loginKaptchaFilter = new
LoginKaptchaFilter();
        //1.认证 url
        loginKaptchaFilter.setFilterProcessesUrl("/doLogin");
        //2.认证 接收参数
        loginKaptchaFilter.setUsernameParameter("uname");
    }
}
```

```
    loginKaptchaFilter.setPasswordParameter("passwd");
    loginKaptchaFilter.setKaptchaParameter("kaptcha");
    //3.指定认证管理器

    loginKaptchaFilter.setAuthenticationManager(authenticationMa
gerBean());
    //4.指定成功时处理

    loginKaptchaFilter.setAuthenticationSuccessHandler((req, resp,
authentication) → {
        Map<String, Object> result = new HashMap<String,
Object>();
        result.put("msg", "登录成功");
        result.put("用户信息",
authentication.getPrincipal());
        resp.setContentType("application/json;charset=UTF-
8");
        resp.setStatus(HttpStatus.OK.value());
        String s = new
ObjectMapper().writeValueAsString(result);
        resp.getWriter().println(s);
    });
    //5.认证失败处理

    loginKaptchaFilter.setAuthenticationFailureHandler((req, resp,
ex) → {
        Map<String, Object> result = new HashMap<String,
Object>();
        result.put("msg", "登录失败: " + ex.getMessage());

        resp.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
        resp.setContentType("application/json;charset=UTF-
8");
        String s = new
ObjectMapper().writeValueAsString(result);
        resp.getWriter().println(s);
    });
    return loginKaptchaFilter;
}
```

```
}

@Override
protected void configure(HttpSecurity http) throws
Exception {
    http.authorizeRequests()
        .mvcMatchers("/vc.jpg").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .and()
        .exceptionHandling()
        .authenticationEntryPoint((req, resp, ex) → {

    resp.setContentType("application/json;charset=UTF-8");

    resp.setStatus(HttpStatus.UNAUTHORIZED.value());
        resp.getWriter().println("必须认证之后才能访
问!");
    })
    .and()
    .logout()
    .and()
    .csrf().disable();

    http.addFilterAt(loginKaptchaFilter(),
UsernamePasswordAuthenticationFilter.class);
}
```

- 测试验证

第五章 密码加密

- 密码为什么要加密
- 常见加密的解决方案
- PasswordEncoder 详解
- 优雅使用加密

简介

加密意义

2011 年12月21 日，有人在网络上公开了一个包含600万个 CSDN 用户资料的数据库，数据全部为明文储存，包含用户名、密码以及注册邮箱。事件发生后 CSDN 在微博、官方网站等渠道发出了声明，解释说此数据库系2009 年备份所用，因不明原因泄漏，已经向警方报

案，后又在官网发出了公开道歉信。在接下来的十多天里，金山、网易、京东、当当、新浪等多家公司被卷入到这次事件中。整个事件中最触目惊心的莫过于 CSDN 把用户密码明文存储，由于很多用户是多个网站共用一个密码，因此一个网站密码泄漏就会造成很大的安全隐患。由于有了这么多前车之鉴，我们现在做系统时，密码都要加密处理。

在前面的案例中，凡是涉及密码的地方，我们都采用明文存储，在实际项目中这肯定是不可取的，因为这会带来极高的安全风险。在企业级应用中，密码不仅需要加密，还需要加[盐](#)，最大程度地保证密码安全。

常见方案

Hash 算法

最早我们使用类似 SHA-256 、SHA-512 、MD5等这样的单向 Hash 算法。用户注册成功后，保存在数据库中不再是用户的明文密码，而是经过 SHA-256 加密计算的一个字符串，当用户进行登录时，用户输入的明文密码用 SHA-256 进行加密，加密完成之后，再和存储在数据库中的密码进行比对，进而确定用户登录信息是否有效。如果系统遭遇攻击，最多也只是存储在数据库中的密文被泄漏。

这样就绝对安全了吗？由于彩虹表这种攻击方式的存在以及随着计算机硬件的发展，每秒执行数十亿次 HASH 计算已经变得轻轻松松，这意味着即使给密码加密加盐也不再安全。

参考：[彩虹表](#)

单向自适应函数

在Spring Security 中，我们现在是用一种自适应单向函数（Adaptive One-way Functions）来处理密码问题，这种自适应单向函数在进行密码匹配时，会有意占用大量系统资源（例如CPU、内存等），这样可以增加恶意用户攻击系统的难度。在Spring Security 中，开发者可以通过 bcrypt、PBKDF2、scrypt 以及 argon2 来体验这种自

适应单向函数加密。由于自适应单向函数有意占用大量系统资源，因此每个登录认证请求都会大大降低应用程序的性能，但是 Spring Security 不会采取任何措施来提高密码验证速度，因为它正是通过这种方式来增强系统的安全性。

参考 1: https://byronhe.gitbooks.io/libsodium/content/password_hashing/

参考 2: <https://github.com/xitu/gold-miner/blob/master/TOD01/password-hashing-pbkdf2-scrypt-bcrypt-and-argon2.md>

- BCryptPasswordEncoder

BCryptPasswordEncoder 使用 bcrypt 算法对密码进行加密，为了提高密码的安全性，bcrypt 算法故意降低运行速度，以增强密码破解的难度。同时 BCryptPasswordEncoder “为自己带盐”开发者不需要额外维护一个“盐”字段，使用 BCryptPasswordEncoder 加密后的字符串就已经“带盐”了，即使相同的明文每次生成的加密字符串都不相同。

- Argon2PasswordEncoder

Argon2PasswordEncoder 使用 Argon2 算法对密码进行加密，Argon2 曾在 Password Hashing Competition 竞赛中获胜。为了解决在定制硬件上密码容易被破解的问题，Argon2 也是故意降低运算速度，同时需要大量内存，以确保系统的安全性。

- PbKdf2PasswordEncoder

PbKdf2PasswordEncoder 使用 PBKDF2 算法对密码进行加密，和前面几种类似，PBKDF2 算法也是一种故意降低运算速度的算法，当需要 FIPS (Federal Information Processing Standard, 美国联邦信息处理标准) 认证时，PBKDF2 算法是一个很好的选择。

- SCryptPasswordEncoder

SCryptPasswordEncoder 使用 scrypt 算法对密码进行加密，和前面的几种类似，scrypt 也是一种故意降低运算速度的算法，而且需要大量内存。

PasswordEncoder

通过对认证流程源码分析得知，实际密码比较是由PasswordEncoder完成的，因此只需要使用PasswordEncoder 不同实现就可以实现不同方式加密。

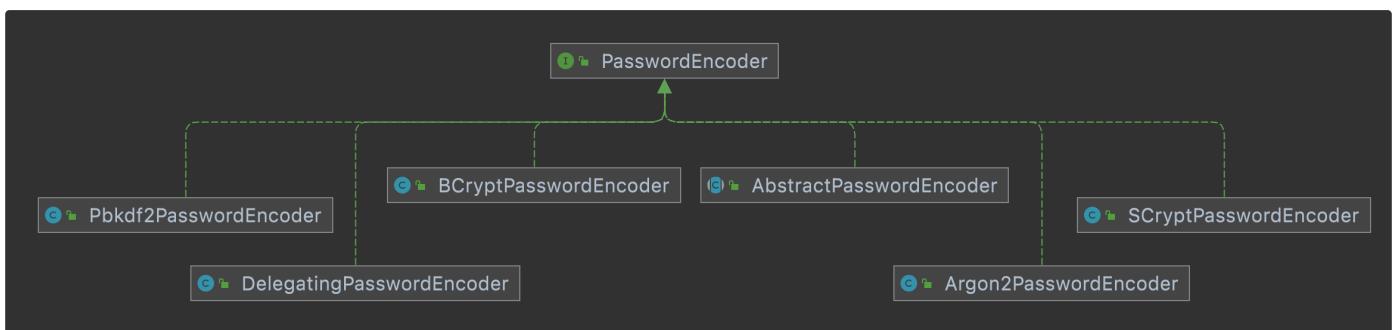
```

public interface PasswordEncoder {
    String encode(CharSequence rawPassword);
    boolean matches(CharSequence rawPassword, String encodedPassword);
    default boolean upgradeEncoding(String encodedPassword) {
        return false;
    }
}

```

- encode 用来进行明文加密的
- matches 用来比较密码的方法
- upgradeEncoding 用来给密码进行升级的方法

默认提供加密算法如下：



PasswordEncoder

Show Implementations

LazyPasswordEncoder	Maven: org.springframework.security:spring-security-config:5.6.1 (spring-security-config-5.6.1.jar)
LazyPasswordEncoder	Maven: org.springframework.security:spring-security-config:5.6.1 (spring-security-config-5.6.1.jar)
LdapShaPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
Md4PasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
MessageDigestPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
NoOpPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
StandardPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
UnmappedIdPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
AbstractPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
Argon2PasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
BCryptPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
DelegatingPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
Pbkdf2PasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)
SCryptPasswordEncoder	Maven: org.springframework.security:spring-security-crypto:5.6.1 (spring-security-crypto-5.6.1.jar)

DelegatingPasswordEncoder

根据上面 PasswordEncoder的介绍，可能会以为 Spring security 中默认的密码加密方案应该是四种自适应单向加密函数中的一种，其实不然，在 spring Security 5.0之后，默认的密码加密方案其实是 DelegatingPasswordEncoder。从名字上来看，DelegatingPasswordEncoder 是一个代理类，而并非一种全新的密码加密方案，DelegatingPasswordEncoder 主要用来代理上面介绍的不同的密码加密方案。为什么采用 DelegatingPasswordEncoder 而不是某一个具体加密方式作为默认的密码加密方案呢？主要考虑了如下两方面的因素：

- 兼容性：使用 DelegatingPasswordEncoder 可以帮助许多使用旧密码加密方式的系统顺利迁移到 Spring security 中，它允许在同一个系统中同时存在多种不同的密码加密方案。
- 便捷性：密码存储的最佳方案不可能一直不变，如果使用 DelegatingPasswordEncoder 作为默认的密码加密方案，当需要修改加密方案时，只需要修改很小一部分代码就可以实现。

DelegatingPasswordEncoder源码

```
public class DelegatingPasswordEncoder implements PasswordEncoder {  
    ....  
}
```

- encode 用来进行明文加密的
- matches 用来比较密码的方法
- upgradeEncoding 用来给密码进行升级的方法

PasswordEncoderFactories源码

```
public static PasswordEncoder createDelegatingPasswordEncoder() {  
    String encodingId = "bcrypt";  
    Map<String, PasswordEncoder> encoders = new HashMap<>();  
    encoders.put(encodingId, new BCryptPasswordEncoder());  
    encoders.put("ldap", new  
        org.springframework.security.crypto.password.LdapShaPasswordEncoder());  
    encoders.put("MD4", new  
        org.springframework.security.crypto.password.Md4PasswordEncoder());  
}
```

```

        encoders.put("MD5", new
org.springframework.security.crypto.password.MessageDigestPasswordEncoder("MD5"));
        encoders.put("noop",
org.springframework.security.crypto.password.NoOpPasswordEncoder.getInstance());
        encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
        encoders.put("scrypt", new SCryptPasswordEncoder());
        encoders.put("SHA-1", new
org.springframework.security.crypto.password.MessageDigestPasswordEncoder("SHA-1"));
        encoders.put("SHA-256",
new
org.springframework.security.crypto.password.MessageDigestPasswordEncoder("SHA-256"));
        encoders.put("sha256", new
org.springframework.security.crypto.password.StandardPasswordEncoder());
        encoders.put("argon2", new Argon2PasswordEncoder());
    return new DelegatingPasswordEncoder(encodingId, encoders);
}

```

如何使用 PasswordEncoder

- 查看WebSecurityConfigurerAdapter类中源码

```

static class LazyPasswordEncoder implements PasswordEncoder {
    private ApplicationContext applicationContext;
    private PasswordEncoder passwordEncoder;
    LazyPasswordEncoder(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }
    @Override
    public String encode(CharSequence rawPassword) {
        return getPasswordEncoder().encode(rawPassword);
    }
}

```

```
    @Override
    public boolean matches(CharSequence rawPassword, String
encodedPassword) {
        return getPasswordEncoder().matches(rawPassword,
encodedPassword);
    }

    @Override
    public boolean upgradeEncoding(String encodedPassword) {
        return
getPasswordEncoder().upgradeEncoding(encodedPassword);
    }

    private PasswordEncoder getPasswordEncoder() {
        if (this.passwordEncoder != null) {
            return this.passwordEncoder;
        }
        PasswordEncoder passwordEncoder =
getBeanOrNull(PasswordEncoder.class);
        if (passwordEncoder == null) {
            passwordEncoder =
PasswordEncoderFactories.createDelegatingPasswordEncoder();
        }
        this.passwordEncoder = passwordEncoder;
        return passwordEncoder;
    }

    private <T> T getBeanOrNull(Class<T> type) {
        try {
            return this.applicationContext.getBean(type);
        }
        catch (NoSuchBeanDefinitionException ex) {
            return null;
        }
    }

    @Override
    public String toString() {
```

```
    return getPasswordEncoder().toString();
}

}
```

通过源码分析得知如果在工厂中指定了PasswordEncoder，就会使用指定PasswordEncoder，否则就会使用默认DelegatingPasswordEncoder。

密码加密实战

- 测试生成的密码

```
@Test
public void test() {
    //1.BCryptPasswordEncoder
    BCryptPasswordEncoder bCryptPasswordEncoder = new
    BCryptPasswordEncoder();
    System.out.println(bCryptPasswordEncoder.encode("123"));

    //2.Pbkdf2PasswordEncoder
    Pbkdf2PasswordEncoder pbkdf2PasswordEncoder = new
    Pbkdf2PasswordEncoder();
    System.out.println(pbkdf2PasswordEncoder.encode("123"));

    //3.SCryptPasswordEncoder //需要额外引入依赖
    SCryptPasswordEncoder sCryptPasswordEncoder = new
    SCryptPasswordEncoder();
    System.out.println(sCryptPasswordEncoder.encode("123"));

    //4.Argon2PasswordEncoder //需要额外引入依赖
    Argon2PasswordEncoder argon2PasswordEncoder = new
    Argon2PasswordEncoder();
    System.out.println(argon2PasswordEncoder.encode("123"));
}
```

- 使用固定密码加密方案

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public PasswordEncoder BcryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager =
new InMemoryUserDetailsManager();

        inMemoryUserDetailsManager.createUser(User.withUsername("root").p
assword("$2a$10$WGFkRsZC0kzaftK0PcW0NeLvNvg2jqd3U09qd5gjJGSHE5b0yo
y6a").roles("xxx").build());
        return inMemoryUserDetailsManager;
    }
}
```

- 使用灵活密码加密方案 推荐

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager =
new InMemoryUserDetailsManager();

        inMemoryUserDetailsManager.createUser(User.withUsername("root").p
assword(
{bcrypt}$2a$10$WGFkRsZC0kzaftK0PcW0NeLvNvg2jqd3U09qd5gjJGSHE5b0yo
y6a").roles("xxx").build());
        return inMemoryUserDetailsManager;
    }
}
```

密码自动升级

推荐使用`DelegatingPasswordEncoder` 的另外一个好处就是自动进行密码加密方案的升级，这个功能在整合一些老的系统时非常有用。

- 准备库表

```
-- 用户表
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 AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
-- 角色表
CREATE TABLE `role`
(
    `id`          int(11) NOT NULL AUTO_INCREMENT,
    `name`        varchar(32) DEFAULT NULL,
    `name_zh`     varchar(32) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
-- 用户角色关系表
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 AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;
```

- 插入数据

```

-- 插入用户数据
BEGIN;
    INSERT INTO `user`
    VALUES (1, 'root', '{noop}123', 1, 1, 1, 1);
    INSERT INTO `user`
    VALUES (2, 'admin', '{noop}123', 1, 1, 1, 1);
    INSERT INTO `user`
    VALUES (3, 'blr', '{noop}123', 1, 1, 1, 1);
COMMIT;

-- 插入角色数据
BEGIN;
    INSERT INTO `role`
    VALUES (1, 'ROLE_product', '商品管理员');
    INSERT INTO `role`
    VALUES (2, 'ROLE_admin', '系统管理员');
    INSERT INTO `role`
    VALUES (3, 'ROLE_user', '用户管理员');
COMMIT;

-- 插入用户角色数据
BEGIN;
    INSERT INTO `user_role`
    VALUES (1, 1, 1);
    INSERT INTO `user_role`
    VALUES (2, 1, 2);
    INSERT INTO `user_role`
    VALUES (3, 2, 2);
    INSERT INTO `user_role`
    VALUES (4, 3, 3);
COMMIT;

```

- 整合 mybatis

```

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
</dependency>

```

```

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.0</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.8</version>
</dependency>

```

```

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/security?
characterEncoding=UTF-8&serverTimezone=UTC&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
mybatis.mapper-locations=classpath:/mapper/*.xml
mybatis.type-aliases-package=com.baizhi.entity
logging.level.com.baizhi.dao=debug

```

- 编写实体类

```

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;
}

public void setPassword(String password) {
    this.password = password;
}

@Override
public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

@Override
public boolean isAccountNonExpired() {
    return accountNonExpired;
}

public void setAccountNonExpired(Boolean accountNonExpired) {
    this.accountNonExpired = accountNonExpired;
}

@Override
public boolean isAccountNonLocked() {
    return accountNonLocked;
}
```

```
public void setAccountNonLocked(Boolean accountNonLocked) {  
    this.accountNonLocked = accountNonLocked;  
}  
  
@Override  
public boolean isCredentialsNonExpired() {  
    return credentialsNonExpired;  
}  
  
public void setCredentialsNonExpired(Boolean  
credentialsNonExpired) {  
    this.credentialsNonExpired = credentialsNonExpired;  
}  
  
@Override  
public boolean isEnabled() {  
    return enabled;  
}  
  
public void setEnabled(Boolean enabled) {  
    this.enabled = enabled;  
}  
  
public void setRoles(List<Role> roles) {  
    this.roles = roles;  
}  
  
public Integer getId() {  
    return id;  
}  
  
public void setId(Integer id) {  
    this.id = id;  
}  
}  
  
public class Role {
```

```
private Integer id;
private String name;
private String nameZh;

public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getNameZh() {
    return nameZh;
}

public void setNameZh(String nameZh) {
    this.nameZh = nameZh;
}
```

- 创建dao

```
@Mapper
public interface UserDao {
    List<Role> getRolesByUid(Integer uid);
    User loadUserByUsername(String username);
    Integer updatePassword(@Param("username") String
username,@Param("password") String password);
}
```

- 编写 mapper

```
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.baizhi.dao.UserDao">

    <select id="loadUserByUsername" resultType="User">
        select id,
            username,
            password,
            enabled,
            accountNonExpired,
            accountNonLocked,
            credentialsNonExpired
        from `user`
        where username = #{username}
    </select>

    <select id="getRolesByUid" resultType="Role">
        select r.id,
            r.name,
            r.name_zh nameZh
        from `role` r,
            `user_role` ur
        where r.id = ur.rid
            and ur.uid = #{uid}
    </select>

    <update id="updatePassword">
        update `user` set password=#{password}
        where username=#{username}
    </update>

</mapper>
```

- 编写service 实现

```
@Service
public class MyUserDetailsService implements
UserDetailsService,UserDetailsPasswordService {
    private final UserDao userDao;

    @Autowired
    public MyUserDetailsService(UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userDao.loadUserByUsername(username);
        if (ObjectUtils.isEmpty(user)) {
            throw new RuntimeException("用户不存在!");
        }
        user.setRoles(userDao.getRolesByUid(user.getId()));
        return user;
    }

    @Override
    public UserDetails updatePassword(UserDetails user, String
newPassword) {
        Integer result =
userDao.updatePassword(user.getUsername(), newPassword);
        if (result == 1) {
            ((User) user).setPassword(newPassword);
        }
        return user;
    }
}
```

- 配置securityconfig

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final MyUserDetailsService myUserDetailsService;

    @Autowired
    public SecurityConfig(MyUserDetailsService myUserDetailsService) {
        this.myUserDetailsService = myUserDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        //查询数据库
        auth.userDetailsService(myUserDetailsService);
    }

}
```

- 启动项目测试

第六章 RememberMe

- 简介
- 基本使用
- 原理分析
- 持久化令牌

简介

RememberMe 这个功能非常常见，下图就是QQ 邮箱登录时的“记住我”选项。提到 RememberMe，一些初学者往往有一些误解，认为 RememberMe 功能就是把用户名/密码用 Cookie 保存在浏览器中，下次登录时不用再次输入用户名/密码。这个理解显然是不对的。我们这里所说的 RememberMe 是一种服务器端的行为。传统的登录方式基于 Session 会话，一旦用户的会话超时过期，就要再次登录，这样太过于繁琐。如果能有一种机制，让用户会话过期之后，还能继续保持认证状态，就会方便很多，RememberMe 就是为

了解解决这一需求而生的。

QQ邮箱，常联系！

没有什么能够阻挡
你对自由的向往
天马行空的生涯
你的心了无牵挂

摘自《蓝莲花》
听更多许巍音乐



[微信登录](#)[QQ登录](#)

×

下次自动登录

授权登录

[忘了密码?](#) [扫码快捷登录](#) [注册新帐号](#)

具体的实现思路就是通过 `Cookie` 来记录当前用户身份。当用户登录成功之后，会通过一定算法，将用户信息、时间戳等进行加密，加密完成后，通过响应头带回前端存储在 `cookie` 中，当浏览器会话过期之后，如果再次访问该网站，会自动将 `Cookie` 中的信息发送给服务器，服务器对 `Cookie` 中的信息进行校验分析，进而确定出用户的身份，`Cookie` 中所保存的用户信息也是有时效的，例如三天、一周等。

基本使用

开启记住我

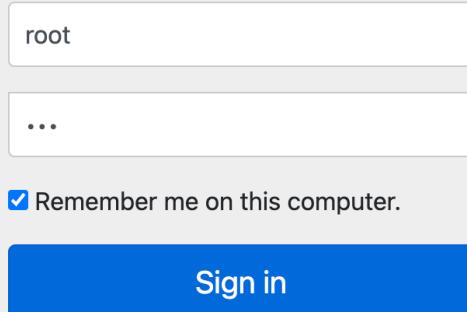
```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    //....
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .mvcMatchers("/login.html").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            // ...
            .and()
    }
}
```

```
.rememberMe() //开启记住我功能  
.and()  
.csrf().disable();  
}  
}
```

使用记住我

可以看到一旦打开了记住我功能，登录页面中会多出一个 RememberMe 选项。

Please sign in



root

...

Remember me on this computer.

Sign in

测试记住我

登录时勾选 RememberMe 选项，然后重启服务端之后，在测试接口是否能免登录访问。

原理分析

RememberMeAuthenticationFilter

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry  
        .anyRequest().authenticated()  
        .and() HttpSecurity  
        .formLogin() FormLoginConfigurer<HttpSecurity>  
        .and() HttpSecurity  
        .rememberMe() RememberMeConfigurer<HttpSecurity>  
        .and() HttpSecurity
```

从上图中，当在SecurityConfig配置中开启了"记住我"功能之后，在进行认证时如果勾选了"记住我"选项，此时打开浏览器控制台，分析整个登录过程。首先当我们登录时，在登录请求中多了一个 RememberMe 的参数。

The screenshot shows a NetworkMiner or similar packet capture tool interface. The 'Payload' tab is selected, showing a POST request with the following form data:

- username: root
- password: 123
- remember-me: on

很显然，这个参数就是告诉服务器应该开启 RememberMe 功能的。如果自定义登录页面开启 RememberMe 功能应该多加入一个一样的请求参数就可以啦。该请求会被 RememberMeAuthenticationFilter 进行拦截然后自动登录具体参见源码：

```
private void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    if (SecurityContextHolder.getContext().getAuthentication() != null) {
        this.logger.debug(LogMessage.of(() -> "SecurityContextHolder not populated with remember-me token, as it already contained: " +
            + SecurityContextHolder.getContext().getAuthentication() + """));
        chain.doFilter(request, response);
        return;
    }
    Authentication rememberMeAuth = this.rememberMeServices.autoLogin(request, response);
    if (rememberMeAuth != null) {
        // Attempt authentication via AuthenticationManager
        try {
            rememberMeAuth = this.authenticationManager.authenticate(rememberMeAuth);
            // Store to SecurityContextHolder
            SecurityContext context = SecurityContextHolder.createEmptyContext();
            context.setAuthentication(rememberMeAuth);
            SecurityContextHolder.setContext(context);
            onSuccessfulAuthentication(request, response, rememberMeAuth);
            this.logger.debug(LogMessage.of(() -> "SecurityContextHolder populated with remember-me token: " +
                + SecurityContextHolder.getContext().getAuthentication() + ""));
            if (this.eventPublisher != null) {
                this.eventPublisher.publishEvent(new InteractiveAuthenticationSuccessEvent(
                    SecurityContextHolder.getContext().getAuthentication(), this.getClass()));
            }
            if (this.successHandler != null) {
                this.successHandler.onAuthenticationSuccess(request, response, rememberMeAuth);
                return;
            }
        }
```

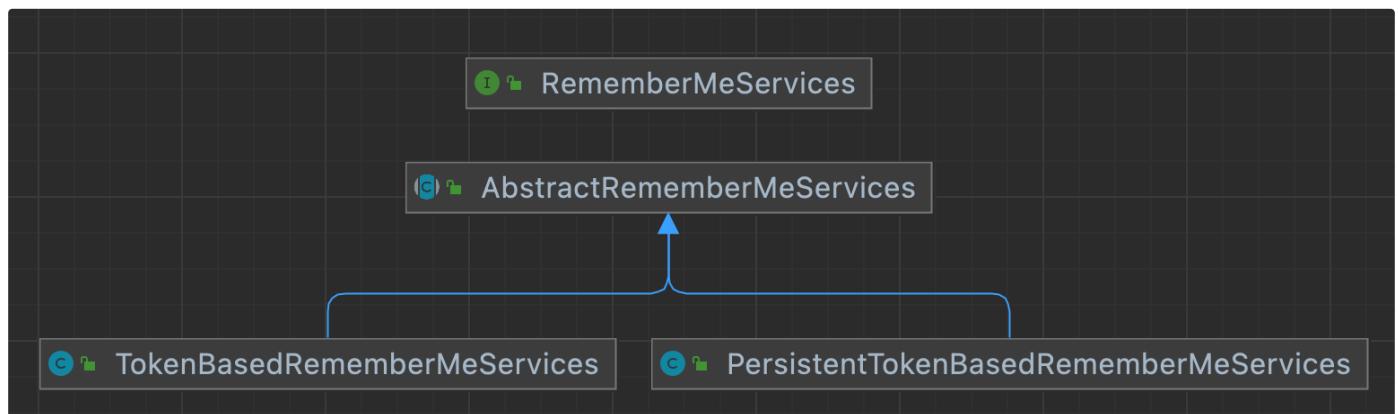
- (1) 请求到达过滤器之后，首先判断 SecurityContextHolder 中是否有值，没值的话表示用户尚未登录，此时调用 autoLogin 方法进行自动登录。

- (2) 当自动登录成功后返回的rememberMeAuth 不为null 时，表示自动登录成功，此时调用 authenticate 方法对 key 进行校验，并且将登录成功的用户信息保存到 SecurityContextHolder 对象中，然后调用登录成功回调，并发布登录成功事件。需要注意的是，登录成功的回调并不包含 RememberMeServices 中的 loginSuccess 方法。
- (3) 如果自动登录失败，则调用 rememberMeServices.loginFail方法处理登录失败回调。onUnsuccessfulAuthentication 和 onSuccessfulAuthentication 都是该过滤器中定义的空方法，并没有任何实现这就是 RememberMeAuthenticationFilter 过滤器做的事情，成功将 RememberMeServices 的服务集成进来。

RememberMeServices

这里一共定义了三个方法：

1. autoLogin 方法可以从请求中提取出需要的参数，完成自动登录功能。
2. loginFail 方法是自动登录失败的回调。
3. loginSuccess 方法是自动登录成功的回调。



TokenBasedRememberMeServices

在开启记住我后如果没有加入额外配置默认实现就是由 `TokenBasedRememberMeServices` 进行的实现。查看这个类源码中 `processAutoLoginCookie` 方法实现：

```

@Override
protected UserDetails processAutoLoginCookie(String[] cookieTokens, HttpServletRequest request,
                                             HttpServletResponse response) {
    if (cookieTokens.length != 3) {
        throw new InvalidCookieException(
            "Cookie token did not contain 3" + " tokens, but contained '" + Arrays.asList(cookieTokens) + "'");
    }
    long tokenExpiryTime = getTokenExpiryTime(cookieTokens);
    if (isTokenExpired(tokenExpiryTime)) {
        throw new InvalidCookieException("Cookie token[1] has expired (expired on '" + new Date(tokenExpiryTime)
                                         + "'; current time is '" + new Date() + "')");
    }
    // Check the user exists. Defer lookup until after expiry time checked, to
    // possibly avoid expensive database call.
    UserDetails userDetails = getUserDetailsService().loadUserByUsername(cookieTokens[0]);
    Assert.notNull(userDetails, () -> "UserDetailsService " + getUserDetailsService()
        + " returned null for username " + cookieTokens[0] + ". " + "This is an interface contract violation");
    // Check signature of token matches remaining details. Must do this after user
    // lookup, as we need the DAO-derived password. If efficiency was a major issue,
    // just add in a UserCache implementation, but recall that this method is usually
    // only called once per HttpSession - if the token is valid, it will cause
    // SecurityContextHolder population, whilst if invalid, will cause the cookie to
    // be cancelled.
    String expectedTokenSignature = makeTokenSignature(tokenExpiryTime, userDetails.getUsername(),
                                                       userDetails.getPassword());
    if (!equals(expectedTokenSignature, cookieTokens[2])) {
        throw new InvalidCookieException("Cookie token[2] contained signature '" + cookieTokens[2]
                                         + "' but expected '" + expectedTokenSignature + "'");
    }
    return userDetails;
}

```

processAutoLoginCookie 方法主要用来验证 Cookie 中的令牌信息是否合法：

1. 首先判断 cookieTokens 长度是否为了，不为了说明格式不对，则直接抛出异常。
2. 从 cookieTokens 数组中提取出第 1 项，也就是过期时间，判断令牌是否过期，如果已经过期，则抛出异常。
3. 根据用户名（cookieTokens 数组的第 1 项）查询出当前用户对象。
4. 调用 makeTokenSignature 方法生成一个签名，签名的生成过程如下：首先将用户名、令牌过期时间、用户密码以及 key 组成一个字符串，中间用“：“隔开，然后通过 MD5 消息摘要算法对该字符串进行加密，并将加密结果转为一个字符串返回。
5. 判断第 4 步生成的签名和通过 Cookie 传来的签名是否相等（即 cookieTokens 数组的第 2 项），如果相等，表示令牌合法，则直接返回用户对象，否则抛出异常。

```

@Override
public void onLoginSuccess(HttpServletRequest request, HttpServletResponse response,
    Authentication successfulAuthentication) {
    String username = retrieveUserName(successfulAuthentication);
    String password = retrievePassword(successfulAuthentication);

    // If unable to find a username and password, just abort as
    // TokenBasedRememberMeServices is
    // unable to construct a valid token in this case.
    if (!StringUtils.hasLength(username)) {
        this.logger.debug("Unable to retrieve username");
        return;
    }

    if (!StringUtils.hasLength(password)) {
        UserDetails user = getUserDetailsService().loadUserByUsername(username);
        password = user.getPassword();

        if (!StringUtils.hasLength(password)) {
            this.logger.debug("Unable to obtain password for user: " + username);
            return;
        }
    }

    int tokenLifetime = calculateLoginLifetime(request, successfulAuthentication);
    long expiryTime = System.currentTimeMillis();

    // SEC-949
    expiryTime += 1000L * ((tokenLifetime < 0) ? TWO_WEEKS_S : tokenLifetime);
    String signatureValue = makeTokenSignature(expiryTime, username, password);
    setCookie(new String[] { username, Long.toString(expiryTime), signatureValue }, tokenLifetime, request,
        response);
}

```

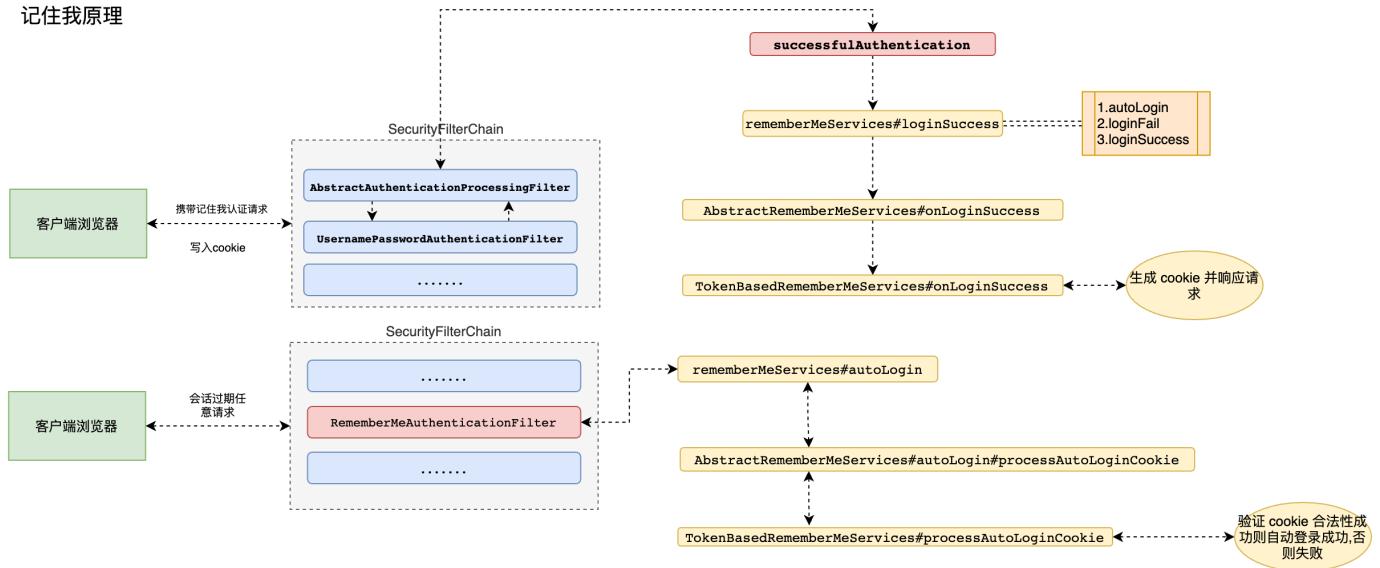
1. 在这个回调中，首先获取用户名和密码信息，如果用户密码在用户登录成功后从 `successfulAuthentication` 对象中擦除，则从数据库中重新加载出用户密码。
2. 计算出令牌的过期时间，令牌默认有效期是两周。
3. 根据令牌的过期时间、用户名以及用户密码，计算出一个签名。
4. 调用 `setCookie` 方法设置 Cookie，第一个参数是一个数组，数组中一共包含三项。用户名、过期时间以及签名，在 `setCookie` 方法中会将数组转为字符串，并进行 Base64 编码后响应给前端。

总结

当用户通过用户名/密码的形式登录成功后，系统会根据用户的用户名、密码以及令牌的过期时间计算出一个签名，这个签名使用 MD5 消息摘要算法生成，是不可逆的。然后再将用户名、令牌过期时间以及签名拼接成一个字符串，中间用“：“隔开，对拼接好的字符串进行 Base64 编码，然后将编码后的结果返回到前端，也就是我们在浏览器中看到的令牌。当会话过期之后，访问系统资源时会自动携带上 Cookie 中的令牌，服务端拿到 Cookie 中的令牌后，先进行 Base64 解码，解码后分别提取出令牌中的三项数据：接着根据令牌中的数据判断令牌是否已经过期，如果没有过期，则根据令牌中的用户名查询出用户信息：接着再计算出一个签名和令牌中的签名进行对比，如果一致，表示令牌是合法令牌，自动登录成功，否则自动登录失败。



记住我原理



内存令牌

PersistentTokenBasedRememberMeServices

```

94     @Override
95     protected UserDetails processAutoLoginCookie(String[] cookieTokens, HttpServletRequest request,
96                                                 HttpServletResponse response) {
97         if (cookieTokens.length != 2) {
98             throw new InvalidCookieException("Cookie token did not contain " + 2 + " tokens, but contained '" +
99                                         + Arrays.asList(cookieTokens) + "'");
100        }
101        String presentedSeries = cookieTokens[0];
102        String presentedToken = cookieTokens[1];
103        PersistentRememberMeToken token = this.tokenRepository.getTokenForSeries(presentedSeries);
104        if (token == null) {
105            // No series match, so we can't authenticate using this cookie
106            throw new RememberMeAuthenticationException("No persistent token found for series id: " + presentedSeries);
107        }
108        // We have a match for this user/series combination
109        if (!presentedToken.equals(token.getTokenValue())) {
110            // Token doesn't match series value. Delete all logins for this user and throw
111            // an exception to warn them.
112            this.tokenRepository.removeUserTokens(token.getUsername());
113            throw new CookieTheftException(this.messages.getMessage(
114                code: "PersistentTokenBasedRememberMeServices.cookieStolen",
115                defaultMessage: "Invalid remember-me token (Series/token) mismatch. Implies previous cookie theft attack."));
116        }
117        if (token.getDate().getTime() + getTokenValiditySeconds() * 1000L < System.currentTimeMillis()) {
118            throw new RememberMeAuthenticationException("Remember-me login has expired");
119        }
120        // Token also matches, so login is valid. Update the token value, keeping the
121        // same series number.
122        this.logger.debug(LogMessage.format( format: "Refreshing persistent login token for user '%s', series '%s'",
123                                              token.getUsername(), token.getSeries()));
124        PersistentRememberMeToken newToken = new PersistentRememberMeToken(token.getUsername(), token.getSeries(),
125                               generateTokenData(), new Date());
126        try {
127            this.tokenRepository.updateToken(newToken.getSeries(), newToken.getTokenValue(), newToken.getDate());

```

- 不同于 TokenBasedRememberMeServices 中的 processAutologinCookie 方法，这里 cookieTokens 数组的长度为 2，第一项是 series，第二项是 token。
- 从 cookieTokens 数组中分到提取出 series 和 token。然后根据 series 去内存中查询出一个 PersistentRememberMeToken 对象。如果查询出来的对象为 null，表示内存中并没有 series 对应的值，本次自动登录失败。如果查询出来的 token 和从 cookieTokens 中解析出来的 token 不相同，说明自动登录令牌已经泄漏（恶意用户利用令牌登录后，内存中的 token 变了），此时移除当前用户的所有自动登录记录并抛出异常。
- 根据数据库中查询出来的结果判断令牌是否过期，如果过期就抛出异常。
- 生成一个新的 PersistentRememberMeToken 对象，用户名和 series 不变， token 重新生成， date 也使用当前时间。newToken 生成后，根据 series 去修改内存中的 token 和 date（即每次自动登录后都会产生新的 token 和 date）
- 调用 addCookie 方法添加 Cookie，在 addCookie 方法中，会调用到我们前面所说的 setCookie 方法，但是要注意第一个数组参数中只有两项： series 和 token（即返回到前端的令牌是通过对 series 和 token 进行 Base64 编码得到的）
- 最后将根据用户名查询用户对象并返回。

使用内存中令牌实现

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .and()
        .rememberMe() //开启记住我功能
        .rememberMeServices(rememberMeServices())
        .and()
        .csrf().disable();
}

@Bean
public RememberMeServices rememberMeServices() {
    return new PersistentTokenBasedRememberMeServices(
        "key", //参数 1: 自定义一个生成令牌 key 默认 UUID
        userDetailsService(), //参数 2:认证数据源
        new InMemoryTokenRepositoryImpl()); //参数 3:令牌存储
    方式
}
}
```

持久化令牌

引入依赖

```
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>druid</artifactId>
<version>1.2.8</version>
</dependency>
```

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
</dependency>

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.0</version>
</dependency>
```

配置数据源

```
spring.thymeleaf.cache=false
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/security?
characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root
mybatis.mapper-locations=classpath:mapper/*.xml
mybatis.type-aliases-package=com.entity
```

配置持久化令牌

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // ...
    @Autowired
    private DataSource dataSource;
    @Bean
    public PersistentTokenRepository persistentTokenRepository(){
        JdbcTokenRepositoryImpl jdbcTokenRepository = new
        JdbcTokenRepositoryImpl();
```

```

        jdbcTokenRepository.setCreateTableOnStartup(false); //只需要
没有表时设置为 true
        jdbcTokenRepository.setDataSource(dataSource);
        return jdbcTokenRepository;
    }
    // ..
    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
            .mvcMatchers("/login.html").permitAll()
            .anyRequest().authenticated()
            // ...
            .logout()
            .and()
            .rememberMe() //开启记住我功能
            .tokenRepository(persistentTokenRepository())
            .and()
            .csrf().disable();
    }
}

```

启动项目并查看数据库

注意：启动项目会自动创建一个表，用来保存记住我的 token 信息

名	行	数据长度	引擎	创建日期	修改日期	排序规则	注释
persistent_logins	0	16.00 KB	InnoDB	2022-02-24 06:19:23		utf8mb4_general_ci	
role	2	16.00 KB	InnoDB	2022-01-21 23:19:39		utf8_general_ci	
user	3	16.00 KB	InnoDB	2022-01-21 23:19:39		utf8_general_ci	
user_role	4	16.00 KB	InnoDB	2022-01-21 23:19:39		utf8_general_ci	

再次测试记住我

在测试发现即使服务器重新启动，依然可以自动登录。

自定义记住我

查看记住我源码

AbstractUserDetailsAuthenticationProvider类中authenticate方法在最后认证成功之后实现了记住我功能，但是查看源码得知如果开启记住我，必须进行相关的设置

The screenshot shows the `AbstractAuthenticationProcessingFilter.java` file in a debugger. Several parts of the code are highlighted with red boxes and arrows pointing to specific annotations or variable values.

- A red box highlights the `successfulAuthentication` method call in the `doFilter` method, with a note: "认证成功之后的处理方法,进入这个方法内部查看记住我实现".
- A red box highlights the `successfulAuthentication` method signature, with a note: "进入内部查看,具体实现".
- A red box highlights the `rememberMeServices.loginSuccess` call, with a note: "rememberMeServices: TokenBasedR".
- A red box highlights the `rememberMeRequested` check in the `loginSuccess` method, with a note: "进入内部,查看实现".
- An annotation `@Override` is highlighted with a red arrow pointing to it.
- A red box highlights the `paramValue` variable in the `rememberMeRequested` method, with a note: "请求参数的值可以是 true、on、yes、1 中任何一个都会开启记住我功能".
- The bottom part of the screenshot shows the debugger's variable view with the `parameter` variable set to "remember-me".

```
235     // AUTHENTICATION SUCCESS
236     if (continueChainBeforeSuccessfulAuthentication) {
237         chain.doFilter(request, response);
238     }
239
240     successfulAuthentication(request, response, chain, authResult); // 认证成功之后的处理方法,进入这个方法内部查看记住我实现
241 }
242
243 /**
244 * Indicates whether this filter should attempt to process a login request for the
245 * current invocation.
246 */
247
248 protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult) throws IOException, ServletException {
249
250     if (logger.isDebugEnabled()) {
251         logger.debug("Authentication success. Updating SecurityContextHolder to contain: " + authResult); // 进入内部查看,具体实现
252     }
253
254     SecurityContextHolder.getContext().setAuthentication(authResult);
255
256     rememberMeServices.loginSuccess(request, response, authResult); // rememberMeServices: TokenBasedR
257 }
258
259
260 @Override
261 public final void loginSuccess(HttpServletRequest request, HttpServletResponse response, Authentication successfulAuthentication) { response: HeaderWriterFilter$HeaderWriterRequest@7740
262     if (!rememberMeRequested(request, parameter)) { request: HeaderWriterFilter$HeaderWriterRequest@7740
263         logger.debug("Remember-me login not requested."); // 进入内部,查看实现
264     }
265 }
266
267
268 protected boolean rememberMeRequested(HttpServletRequest request, String parameter) { request: HeaderWriterFilter$HeaderWriterRequest@7740
269     if (alwaysRemember) { alwaysRemember: false
270         return true;
271     } // 请求参数的值可以是 true、on、yes、1 中任何一个都会开启记住我功能
272
273     String paramValue = request.getParameter(parameter); // request: HeaderWriterFilter$HeaderWriterRequest@7740
274
275     if (paramValue != null) {
276         if (paramValue.equalsIgnoreCase("true") || paramValue.equalsIgnoreCase("yes") || paramValue.equals("1")) { // paramValue.equalsIgnoreCase("true") || paramValue.equalsIgnoreCase("yes") || paramValue.equals("1")
277             return true;
278         }
279     }
280 }
281
282
283 AbstractRememberMeServices > rememberMeRequested()
284 SpringbootsecurityApplication >
```

传统 web 开发记住我实现

通过源码分析得知必须在认证请求中加入参数remember-me值为"true,on,yes,1"其中任意一个才可以完成记住我功能,这个时候修改认证界面:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>登录</title>
</head>
<body>
<h1>用户登录</h1>
<form method="post" th:action="@{/doLogin}">
    用户名:<input name="uname" type="text"/><br>
    密码:<input name="passwd" type="password"/><br>
    记住我: <input type="checkbox" name="remember-me" value="on|yes|true|1"/><br>
    <input type="submit" value="登录" />
</form>
</body>
</html>
```

配置中开启记住我

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .....
            .and()
            .rememberMe() // 开启记住我
            // .alwaysRemember(true) 总是记住我
            .and()
            .csrf().disable();
    }
}
```

前后端分离开发记住我实现

自定义认证类 LoginFilter

```
/*
 * 自定义前后端分离认证 Filter
 */
public class LoginFilter extends UsernamePasswordAuthenticationFilter {

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
        System.out.println("=====");
        //1.判断是否是 post 方式请求
        if (!request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException("Authentication method not supported: " + request.getMethod());
        }
        //2.判断是否是 json 格式请求类型
        if (request.getContentType().equalsIgnoreCase(MediaType.APPLICATION_JSON_VALUE)) {
            //3.从 json 数据中获取用户输入用户名和密码进行认证
            {"uname":"xxx","password":"xxx","remember-me":true}
            try {
                Map<String, String> userInfo = new ObjectMapper().readValue(request.getInputStream(), Map.class);
                String username =
userInfo.getUsername();
                String password =
userInfo.getPassword();
                String rememberValue =
userInfo.get(AbstractRememberMeServices.DEFAULT_PARAMETER);
```

```

    if (!ObjectUtils.isEmpty(rememberValue)) {

        request.setAttribute(AbstractRememberMeServices.DEFAULT_PARAMETER
        , rememberValue);
        }

        System.out.println("用户名: " + username + " 密码: "
+ password + " 是否记住我: " + rememberValue);

        UsernamePasswordAuthenticationToken authRequest =
new UsernamePasswordAuthenticationToken(username, password);
        setDetails(request, authRequest);
        return
    this.getAuthenticationManager().authenticate(authRequest);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

return super.attemptAuthentication(request, response);
}
}

```

自定义 RememberMeService

```

/**
 * 自定义记住我 services 实现类
 */
public class MyPersistentTokenBasedRememberMeServices extends
PersistentTokenBasedRememberMeServices {
    public MyPersistentTokenBasedRememberMeServices(String key,
UserDetailsService userDetailsService, PersistentTokenRepository
tokenRepository) {
        super(key, userDetailsService, tokenRepository);
    }
    /**
     * 自定义前后端分离获取 remember-me 方式
     */
    @Override

```

```
protected boolean rememberMeRequested(HttpServletRequest request, String parameter) {
    String paramValue =
request.getAttribute(parameter).toString();
    if (paramValue != null) {
        if (paramValue.equalsIgnoreCase("true") ||
paramValue.equalsIgnoreCase("on")
            || paramValue.equalsIgnoreCase("yes") ||
paramValue.equals("1")) {
            return true;
        }
    }
    return false;
}
}
```

配置记住我

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsService userDetailsService() {
        //.....
        return inMemoryUserDetailsManager;
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.userDetailsService(userDetailsService());
    }

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean()
throws Exception {
        return super.authenticationManagerBean();
    }
}
```

```
//自定义 filter 交给工厂管理
@Bean
public LoginFilter loginFilter() throws Exception {
    LoginFilter loginFilter = new LoginFilter();
    loginFilter.setFilterProcessesUrl("/doLogin");//指定认证
url
    loginFilter.setUsernameParameter("uname");//指定接收json 用户名 key
    loginFilter.setPasswordParameter("passwd");//指定接收 json 密码 key

    loginFilter.setAuthenticationManager(authenticationManagerBean());
;
    loginFilter.setRememberMeServices(rememberMeServices());
//设置认证成功时使用自定义rememberMeService
    //认证成功处理
    loginFilter.setAuthenticationSuccessHandler((req, resp,
authentication) → {
        Map<String, Object> result = new HashMap<String,
Object>();
        result.put("msg", "登录成功");
        result.put("用户信息", authentication.getPrincipal());
        resp.setContentType("application/json;charset=UTF-8");
        resp.setStatus(HttpStatus.OK.value());
        String s = new
ObjectMapper().writeValueAsString(result);
        resp.getWriter().println(s);
    });
    //认证失败处理
    loginFilter.setAuthenticationFailureHandler((req, resp,
ex) → {
        Map<String, Object> result = new HashMap<String,
Object>();
        result.put("msg", "登录失败: " + ex.getMessage());

        resp.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
        resp.setContentType("application/json;charset=UTF-8");
    });
}
```

```
        String s = new
ObjectMapper().writeValueAsString(result);
        resp.getWriter().println(s);
    });
    return loginFilter;
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests()
        .anyRequest().authenticated() //所有请求必须认证
        .and()
        .formLogin()
        .and()
        .rememberMe() //开启记住我功能 cookie 进行实现 1.认
证成功保存记住我 cookie 到客户端 2.只有 cookie 写入客户端成功才能实现自
动登录功能
        .rememberMeServices(rememberMeServices()) //设置自
动登录使用哪个 rememberMeServices
        .and()
        .exceptionHandling()
        .authenticationEntryPoint((req, resp, ex) → {

resp.setContentType(MediaType.APPLICATION_JSON_UTF8_VALUE);

resp.setStatus(HttpStatus.UNAUTHORIZED.value());
        resp.getWriter().println("请认证之后再去处理!");
    })
    .and()
    .logout()
    .logoutRequestMatcher(new OrRequestMatcher(
        new AntPathRequestMatcher("/logout",
HttpMethod.DELETE.name()),
        new AntPathRequestMatcher("/logout",
HttpMethod.GET.name())
    ))
    .logoutSuccessHandler((req, resp, auth) → {
```

```

        Map<String, Object> result = new
HashMap<String, Object>();
        result.put("msg", "注销成功");
        result.put("用户信息", auth.getPrincipal());

resp.setContentType("application/json;charset=UTF-8");
        resp.setStatus(HttpStatus.OK.value());
        String s = new
ObjectMapper().writeValueAsString(result);
        resp.getWriter().println(s);
    }
    .and()
    .csrf().disable();

// at: 用来某个 filter 替换过滤器链中哪个 filter
// before: 放在过滤器链中哪个 filter 之前
// after: 放在过滤器链中那个 filter 之后
http.addFilterAt(loginFilter(),
UsernamePasswordAuthenticationFilter.class);
}


```

```

@Bean
public RememberMeServices rememberMeServices() {
    return new
MyPersistentTokenBasedRememberMeServices(UUID.randomUUID().toString(),
    userDetailsService(), new InMemoryTokenRepositoryImpl());
}

```

第七章 会话管理

- 简介
- 会话并发管理
- 会话共享实战

简介

当浏览器调用登录接口登录成功后，服务端会和浏览器之间建立一个会话（Session）浏览器在每次发送请求时都会携带一个 SessionId，服务端则根据这个 SessionId 来判断用户身份。当浏览器关闭后，服务端的 Session 并不会自动销毁，需要开发者手动在服务端调用 Session 销毁方法，或者等 Session 过期时间到了自动销毁。在 Spring Security 中，与 HttpSession 相关的功能由 SessionManagementFilter 和 SessionAuthenticationStrategy 接口来处理，SessionManagementFilter 过滤器将 Session 相关操作委托给 SessionAuthenticationStrategy 接口去完成。

会话并发管理

简介

会话并发管理就是指在当前系统中，同一个用户可以同时创建多少个会话，如果一个设备对应一个会话，那么也可以简单理解为同一个用户可以同时在多少台设备上进行登录。默认情况下，同一用户在多少台设备上登录并没有限制，不过开发者可以在 Spring Security 中对此进行配置。

开启会话管理

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    // ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .rememberMe()
            .and()
            .csrf().disable()
            .sessionManagement() // 开启会话管理
    }
}
```

```
        .maximumSessions(1); //设置会话并发数为 1
    }

    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }
}
```

1. `sessionManagement()` 用来开启会话管理、`maximumSessions` 指定会话的并发数为 1。
2. `HttpSessionEventPublisher` 提供一个`HttpSessionEventPublisher`-实例。Spring Security中通过一个 Map 集合来保护当前的 HttpSession 记录，进而实现会话的并发管理。当用户登录成功时，就向集合中添加一条 HttpSession 记录；当会话销毁时，就从集合中移除一条 HttpSession 记录。`HttpSessionEventPublisher` 实现了 `HttpSessionListener` 接口，可以监听到 HttpSession 的创建和销毁事件，并将 HttpSession 的创建/销毁事件发布出去，这样，当有 HttpSession 销毁时，Spring Security 就可以感知到该事件了。

测试会话管理

配置完成后，启动项目。这次测试我们需要两个浏览器，如果使用了 Chrome 浏览器，可以使用 Chrome 浏览器中的多用户方式（相当于两个浏览器）先在第一个浏览器中输入 `http://localhost:8080`，此时会自动跳转到登录页面，完成登录操作，就可以访问到数据了；接下来在第二个浏览器中也输入 `http://localhost:8080`，也需要登录，完成登录操作；当第二个浏览器登录成功后，再回到第一个浏览器，刷新页面。结果出现下图：



会话失效处理

传统 web 开发处理

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .anyRequest().authenticated()  
        .and()  
        ....  
        .sessionManagement() //开启会话管理  
        .maximumSessions(1) //允许同一个用户只允许创建一个会话  
        .expiredUrl("/login"); //会话过期处理  
}
```

前后端分离开发处理

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .anyRequest().authenticated()  
        ....  
        .sessionManagement() //开启会话管理  
        .maximumSessions(1) //允许同一个用户只允许创建一个会话  
        // .expiredUrl("/login") //会话过期处理 传统 web 开发  
        .expiredSessionStrategy(event → {  
            HttpServletResponse response = event.getResponse();  
            response.setContentType("application/json;charset=UTF-8");  
            Map<String, Object> result = new HashMap();  
            result.put("status", 500);  
            result.put("msg", "当前会话已经失效,请重新登录!");  
            String s = new ObjectMapper().writeValueAsString(result);  
            response.setContentType("application/json;charset=UTF-8");  
            response.getWriter().println(s);  
            response.flushBuffer();  
        }); // 前后端分离开发处理  
}
```

禁止再次登录

默认的效果是一种被“挤下线”的效果，后面登录的用户会把前面登录的用户“挤下线”。还有一种是禁止后来者登录，即一旦当前用户登录成功，后来者无法再次使用相同的用户名登录，直到当前用户主动注销登录，配置如下：

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .anyRequest().authenticated()  
        .and()  
        ....  
        .sessionManagement() //开启会话管理  
            .maximumSessions(1) //允许同一个用户只允许创建一个会话  
            //.expiredUrl("/login")//会话过期处理 传统 web 开发  
        .expiredSessionStrategy(event → {  
            HttpServletResponse response = event.getResponse();  
            response.setContentType("application/json;charset=UTF-8");  
            Map<String, Object> result = new HashMap();  
            result.put("status", 500);  
            result.put("msg", "当前会话已经失效,请重新登录!");  
            String s = new ObjectMapper().writeValueAsString(result);  
            response.getWriter().println(s);  
            response.flushBuffer();  
        }) //前后端分离开发处理  
        .maxSessionsPreventsLogin(true); //登录之后禁止再次登录  
}
```

会话共享

前面所讲的会话管理都是单机上的会话管理，如果当前是集群环境，前面所讲的会话管理方案就会失效。此时可以利用 `spring-session` 结合 `redis` 实现 `session` 共享。

实战

引入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

编写配置

```
spring.redis.host=localhost
spring.redis.port=6379
```

配置Security

```
package com.blr.config;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
```

```
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.session.FindByIndexNameSessionRepository;
import org.springframework.session.security.SpringSessionBackedSessionRegistry;

import javax.servlet.http.HttpServletResponse;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final FindByIndexNameSessionRepository
sessionRepository;

    @Autowired
    public SecurityConfig(FindByIndexNameSessionRepository
sessionRepository) {
        this.sessionRepository = sessionRepository;
    }

    @Bean
    public UserDetailsService userDetailsService() {
        ....
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.userDetailsService(userDetailsService());
```

```
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .and()
        .rememberMe()
        .and()
        .csrf().disable()
        .sessionManagement() //开启会话管理
        .maximumSessions(1) //允许同一个用户只允许创建一个会话
*/
    .expiredUrl("/login")//会话过期处理 传统 web 开发
    .expiredSessionStrategy(event → {
        HttpServletResponse response =
event.getResponse();

response.setContentType("application/json;charset=UTF-8");
        Map<String, Object> result = new HashMap();
        result.put("status", 500);
        result.put("msg", "当前会话已经失效,请重新登录!");
        String s = new
ObjectMapper().writeValueAsString(result);
        response.getWriter().println(s);
        response.flushBuffer();
    }).sessionRegistry(sessionRegistry()); //前后端分离开发处理
    // .maxSessionsPreventsLogin(true); //登录之后禁止再次登录*/
}

@Bean
public SpringSessionBackedSessionRegistry sessionRegistry() {
    return new
SpringSessionBackedSessionRegistry(sessionRepository);
}
```

}

测试

第八章 CSRF 漏洞保护

- CSRF 简介
- CSRF 防御&基本配置
- 实战

简介

CSRF (Cross-Site Request Forgery 跨站请求伪造), 也可称为一键式攻击 (one-click-attack) , 通常缩写为 **CSRF** 或者 **XSRF**。

CSRF 攻击是一种挟持用户在当前已登录的浏览器上发送恶意请求的攻击方法。相对于XSS 利用用户对指定网站的信任, CSRF则是利用网站对用户网页浏览器的信任。简单来说, CSRF 是致击者通过一些技术手段欺骗用户的浏览器, 去访问一个用户曾经认证过的网站并执行恶意请求, 例如发送邮件、发消息、甚至财产操作 (如转账和购买商品) 。由于客户端(浏览器)已经在该网站上认证过, 所以该网站会认为是真正用户在操作而执行请求 (实际上这个并非用户的本意) 。

举个简单的例子:

假设 blr 现在登录了某银行的网站准备完成一项转账操作, 转账的链接如下:

`https://bank.xxx.com/withdraw?account=blr&amount=1000&for=zhangsan`

可以看到, 这个链接是想从 blr 这个账户下转账 1000 元到 zhangsan 账户下, 假设 blr 没有注销登录该银行的网站, 就在同一个浏览器新的选项卡中打开了一个危险网站, 这个危险网站中有一幅图片, 代码如下:

一旦用户打开了这个网站, 这个图片链接中的请求就会自动发送出去。由于是同一个浏览器并且用户尚未注销登录, 所以该请求会自动携带上对应的有效的 Cookie 信息, 进而完成一次转账操作。这就是跨站请求伪造。

CSRF攻击演示

创建银行应用

- 引入依赖

```
<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>
```

- 修改配置

```
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager =
new InMemoryUserDetailsManager();

        inMemoryUserDetailsManager.createUser(User.withUsername("root")
.password("{noop}123").roles("admin").build());
        return inMemoryUserDetailsManager;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.userDetailsService(userDetailsService());
    }

    @Override
```

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated()
        .and().formLogin().and().csrf().disable();
}
```

- 创建 controller 并启动启动

```
@RestController
public class HelloController {

    @PostMapping("/withdraw")
    public String withdraw() {
        System.out.println("执行一次转账操作");
        return "执行一次转账操作";
    }
}
```

创建恶意应用

- 创建简单 springboot 应用
- 修改配置

```
server:
  port: 8081
```

- 准备攻击页面

```
<form action="http://127.0.0.1:8080/withdraw" method="post">
    <input name="name" type="hidden" value="blr"/>
    <input name="money" type="hidden" value="10000">
    <input type="submit" value="点我">
</form>
```

CSRF 防御

CSRF 攻击的根源在于浏览器默认的身份验证机制(自动携带当前网站的Cookie信息)，这种机制虽然可以保证请求是来自用户的某个浏览器，但是无法确保这请求是用户授权发送。攻击者和用户发送的请求一模一样，这意味着我们没有办法去直接拒绝这里的某一个请求。如果能在合法清求中额外携带一个攻击者无法获取的参数，就可以成功区分出两种不同的请求，进而直接拒绝掉恶意请求。在 SpringSecurity 中就提供了这种机制来防御 CSRF 攻击，这种机制我们称之为 **令牌同步模式**。

令牌同步模式

这是目前主流的 CSRF 攻击防御方案。具体的操作方式就是在每一个 HTTP 请求中，除了默认自动携带的 Cookie 参数之外，再提供一个安全的、随机生成的字符串，我们称之为 CSRF 令牌。这个 CSRF 令牌由服务端生成，生成后在 HttpSession 中保存一份。当前端请求到达后，将请求携带的 CSRF 令牌信息和服务端中保存的令牌进行对比，如果两者不相等，则拒绝掉该 HTTP 请求。

注意：考虑到会有一些外部站点链接到我们的网站，所以我们要求请求是幂等的，这样对子HEAD、OPTIONS、TRACE 等方法就没有必要使用 CSRF 令牌了，强行使用可能会导致令牌泄露！

开启 CSRF 防御

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.
            ...
            formLogin()
            .and()
            .csrf(); //开启 csrf
    }
}
```

查看登录页面源码

```
<body>
  <div class="container">
    <form class="form-signin" method="post" action="/login">
      <h2 class="form-signin-heading">Please sign in</h2>
      <p>
        <label for="username" class="sr-only">Username</label>
        <input type="text" id="username" name="username" class="form-control" placeholder="Username" required autofocus>
      </p>
      <p>
        <label for="password" class="sr-only">Password</label>
        <input type="password" id="password" name="password" class="form-control" placeholder="Password" required>
      </p>
      <input name="_csrf" type="hidden" value="b7dec1d8-8a49-4bfd-a185-6f67a711573f" />
      <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
    </form>
```

传统web开发使用CSRF

开启CSRF防御后会自动在提交的表单中加入如下代码，如果不能自动加入，需要在开启之后手动加入如下代码，并随着请求提交。获取服务端令牌方式如下：

```
<input type="hidden" th:name="${_csrf.parameterName}"
th:value="${_csrf.token}" />
```

开发测试 controller

```
@Controller
public class HelloController {
    @PostMapping("/hello")
    @ResponseBody
    public String hello() {
        System.out.println("hello success");
        return "hello success";
    }

    @GetMapping("/index.html")
    public String index() {
        return "index";
    }
}
```

创建 html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试 CSRF 防御</title>
</head>
<body>
<form th:action="@{/hello}" method="post">
    <input type="submit" value="hello"/>
</form>
</body>
</html>
```

测试查看index.html源码

```
▶ <div data-v-32478853 class="odm_extension image_downloader_wrapper">...</div>
▼ <form action="/hello" method="post">
  <input type="hidden" name="_csrf" value="60a0d98a-a247-4ace-a7c1-a649955c55d4">
  <input type='submit' value='hello'>
</form>
```

前后端分离使用 CSRF

前后端分离开发时，只需要将生成 csrf 放入到cookie 中，并在请求时获取 cookie 中令牌信息进行提交即可。

修改 CSRF 存入 Cookie

```

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

        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    }
}

```

访问登录界面查看 cookie

	Name	Value
Manifest	XSRF-TOKEN	8f06e94e-36fd-439b-ad49-16d8045dfa34
Service Workers	JSESSIONID	40CDE1B0B4DA9D7233A316FD51A08274
Storage	SESSION	YjdiNjNiODctYWM4OC00MzVkLTIIYjAtNTY2Y2FjZGM0MjI0
Storage		
▶ Local Storage		
▶ Session Storage		
IndexedDB		
Web SQL		
▼ Cookies		
http://127.0.0.1:8080		

发送请求携带令牌即可

- 请求参数中携带令牌

```

key: _csrf
value:"xxx"

```

- 请求头中携带令牌

```
X-XSRF-TOKEN:value
```

第九章 跨域

- Spring 处理方案。
- Spring Security 处理方案。

简介

跨域问题是实际应用开发中一个非常常见的需求，在 Spring 框架中对于跨域问题的处理方案有好几种，引了 Spring Security 之后，跨域问题的处理方案又增加了。

什么是 CORS

CORS (Cross-Origin Resource Sharing) 是由 W3C 制定的一种跨域资源共享技术标准，其目的就是为了解决前端的跨域请求。在 JavaEE 开发中，最常见的前端跨域请求解决方案是早期的 JSONP，但是 JSONP 只支持 GET 请求，这是一个很大的缺陷，而 CORS 则支持多种 HTTP 请求方法，也是目前主流的跨域解决方案。

CORS 中新增了一组 HTTP 请求头字段，通过这些字段，服务器告诉浏览器，那些网站通过浏览器有权限访问哪些资源。同时规定，对那些可能修改服务器数据的 HTTP 请求方法（如 GET 以外的 HTTP 请求等），浏览器必须首先使用 OPTIONS 方法发起一个预检请求（preflightst），预检请求的目的是查看服务端是否支持即将发起的跨域请求，如果服务端允许，才发送实际的 HTTP 请求。在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（如 Cookies、HTTP 认证信息等）。

CORS：同源/同域 = 协议+主机+端口

简单请求

GET 请求为例，如果需要发起一个跨域请求，则请求头如下：

```
Host: localhost:8080
Origin: http://localhost:8081
Referer:http://localhost:8081/index.html
```

如果服务端支持该跨域请求，那么返回的响应头中将包含如下字段：

Access-Control-Allow-Origin:http://localhost: 8081

Access-Control-Allow-Origin 字段用来告诉浏览器可以访问该资源的域，当浏览器收到这样的响应头信息之后，提取出 Access-Control-Allow-Origin 字段中的值，发现该值包含当前页面所在的域，就知道这个跨域是被允许的，因此就不再对前端的跨域请求进行限制。这属于简单请求，即不需要进行预检请求的跨域。

非简单请求

对于一些非简单请求，会首先发送一个预检请求。预检请求类似下面这样：

```
OPTIONS /put HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: */
Access-Control-Request-Method:PUT
Origin: http://localhost: 8081
Referer:http://localhost:8081/index.html
```

请求方法是 OPTIONS，请求头 Origin 就告诉服务端当前页面所在域，请求头 Access-Control-Request-Methods 告诉服务器端即将发起的跨域请求所使用的万法。服务端对此进行判断，如果允许即将发起的跨域请求，则会给出如下响应：

```
HTTP/1.1 200
Access-Control-Allow-Origin:http://localhost: 8081
Access-Control-Request-Methods: PUT
Access-Control-Max-Age: 3600
```

Access-Control-Allow-Methods 字段表示允许的跨域方法：Access-Control-Max-Age 字段表示预检请求的有效期，单位为秒，在有效期内如果发起该跨域请求，则不用再次发起预检请求。预检请求结束后，接下来就会发起一个真正的跨域请求，跨域请求和前面的简单请求跨域步骤类似。

Spring 跨域解决方案

@CrossOrigin

Spring 中第一种处理跨域的方式是通过`@CrossOrigin` 注解来标记支持跨域，该注解可以添加在方法上，也可以添加在 `Controller` 上。当添加在 `Controller` 上时，表示 `Controller` 中的所有

有接口都支持跨域，具体配置如下：

```
@RestController
public class HelloController{
    @CrossOrigin (origins ="http://localhost:8081")
    @PostMapping ("/post")
    public String post (){
        return "hello post";
    }
}
```

`@CrossOrigin` 注解各属性含义如下：

- `allowCredentials`: 浏览器是否应当发送凭证信息，如 `Cookie`。
- `allowedHeaders`: 请求被允许的请求头字段，`*` 表示所有字段。
- `exposedHeaders`: 哪些响应头可以作为响应的一部分暴露出来。
注意，这里只可以一一列举，通配符 `*` 在这里是无效的。
- `maxAge`: 预检请求的有效期，有效期内不必再次发送预检请求，默认是 **1800** 秒。
- `methods`: 允许的请求方法，`*` 表示允许所有方法。
- `origins`: 允许的域，`*` 表示允许所有域。

addCorsMapping

`@CrossOrigin` 注解需要添加在不同的 `Controller` 上。所以还有一种全局配置方法，就是通过重写 `WebMvcConfigurerComposite#addCorsMappings` 方法来实现，具体配置如下：

```
Configuration
public class WebMvcConfig implements WebMvcConfigurer{
    Override
    public void addCorsMappings (CorsRegistry registry){
        registry.addMapping("/**") // 处理的请求地址
            .allowedMethods ("*")
            .allowedOrigins("*")
            .allowedHeaders ("*")
            .allowCredentials (false)
            .exposedHeaders ("")
            .maxAge (3600) ;
    }
}
```

CrosFilter

Cosr Filter 是Spring Web 中提供的一个处理跨域的过滤器，开发者也可以通过该过滤器处理跨域。

```
@Configuration
public class WebMvcConfig {
    @Bean
    FilterRegistrationBean<CorsFilter> corsFilter() {
        FilterRegistrationBean<CorsFilter> registrationBean = new
        FilterRegistrationBean<>();
        CorsConfiguration corsConfiguration = new
        CorsConfiguration();
        corsConfiguration.setAllowedHeaders(Arrays.asList("*"));
        corsConfiguration.setAllowedMethods(Arrays.asList("*"));
        corsConfiguration.setAllowedOrigins(Arrays.asList("*"));
        corsConfiguration.setMaxAge(3600L);
        UrlBasedCorsConfigurationSource source = new
        UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**",
        corsConfiguration);
        registrationBean.setFilter(new CorsFilter(source));
    }
}
```

```
        registrationBean.setOrder(-1); //filter 0 1
    return registrationBean;
}
}
```

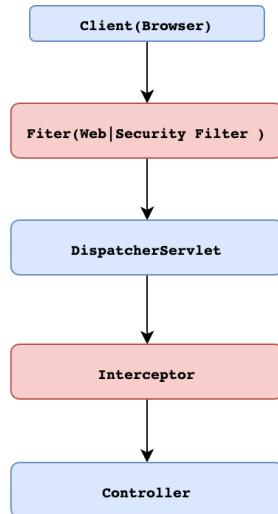
Spring Security 跨域解决方案

原理分析

当我们为项目添加了 Spring Security 依赖之后，发现上面三种跨域方式有的失效了，
有
则可以继续使用，这是怎么回事？

通过@CrossOrigin 注解或者重写 addCorsMappings 方法配置跨域，统统失效了，通过 CorsFilter 配置的跨域，有没有失效则要看过滤器的优先级，如果过滤器优先级高于 Spring Security 过滤器，即先于 Spring Security 过滤器执行，则 CorsFilter 所配置的跨域处理依然有效；如果过滤器优先级低于 Spring Security 过滤器，则 CorsFilter 所配置的跨域处理就会失效。

为了理清楚这个问题，我们先简略了解一下 Filter、DispatcherServlet 以及 Interceptor 执行顺序。



理清楚了执行顺序，我们再来看跨域请求过程。由于非简单请求都要首先发送一个预检请求（preflight request），而预检请求并不会携带认证信息，所以预检请求就有被 Spring Security 拦截的可能。因此通过@CrossOrigin 注解或者重写 addCorsMappings 方法配置跨域就会失效。如果使用 CorsFilter 配置的跨域，只要过滤器优先级高于 Spring Security

过滤器就不会有问题。反之同样会出现问题。

解决方案

Spring Security 中也提供了更专业的方式来解决预检请求所面临的问题。如：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest()
            .authenticated()
            .and()
            .formLogin()
            .and()
            .cors() //跨域处理方案
            .configurationSource(configurationSource())
            .and()
            .csrf().disable();
    }

    CorsConfigurationSource configurationSource() {
        CorsConfiguration corsConfiguration = new
CorsConfiguration();
        corsConfiguration.setAllowedHeaders(Arrays.asList("*"));
        corsConfiguration.setAllowedMethods(Arrays.asList("*"));
        corsConfiguration.setAllowedOrigins(Arrays.asList("*"));
        corsConfiguration.setMaxAge(3600L);
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**",
corsConfiguration);
        return source;
    }
}
```

第十章 异常处理

- Spring Security 异常体系
- 自定义异常配置

异常体系

Spring Security 中异常主要分为两大类：

- **AuthenticationException**: 认证异常
- **AccessDeniedException**: 授权异常

其中认证所涉及异常类型比较多， 默认提供的异常类型如下：

```
▽ * 🔒 AuthenticationException (org.springframework.security.core)
    🔒 InsufficientAuthenticationException (org.springframework.security.authentication)
    🔒 BadCredentialsException (org.springframework.security.authentication)
    🔒 UsernameNotFoundException (org.springframework.security.core.userdetails)
    🔒 SessionAuthenticationException (org.springframework.security.web.authentication.session)
    🔒 ProviderNotFoundException (org.springframework.security.authentication)
▽ 🔒 AuthenticationServiceException (org.springframework.security.authentication)
    🔒 InternalAuthenticationServiceException (org.springframework.security.authentication)
    🔒 PreAuthenticatedCredentialsNotFoundException (org.springframework.security.web.authentication.pr)
    🔒 AuthenticationCredentialsNotFoundException (org.springframework.security.authentication)
    🔒 NonceExpiredException (org.springframework.security.web.authentication.www)
▽ 🔒 RememberMeAuthenticationException (org.springframework.security.web.authentication.rememberm)
    🔒 CookieTheftException (org.springframework.security.web.authentication.rememberme)
    🔒 InvalidCookieException (org.springframework.security.web.authentication.rememberme)
▽ 🔒 AccountStatusException (org.springframework.security.authentication)
    🔒 LockedException (org.springframework.security.authentication)
    🔒 DisabledException (org.springframework.security.authentication)
    🔒 CredentialsExpiredException (org.springframework.security.authentication)
    🔒 AccountExpiredException (org.springframework.security.authentication)
```

相比于认证异常，权限异常类就要少了很多， 默认提供的权限异常如下：

```
▽ * 🔒 AccessDeniedException (org.springframework.security.access)
    🔒 CsrfException (org.springframework.security.web.server.csrf)
    🔒 AuthorizationServiceException (org.springframework.security.access)
▽ 🔒 CsrfException (org.springframework.security.web.csrf)
    🔒 MissingCsrfTokenException (org.springframework.security.web.csrf)
    🔒 InvalidCsrfTokenException (org.springframework.security.web.csrf)
```

在实际项目开发中，如果默认提供异常无法满足需求时，就需要根据实际需要来自定义异常类。

自定义异常处理配置

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest()
            .authenticated()
            // .....
            .and()
            .exceptionHandling() // 异常处理
            .authenticationEntryPoint((request, response, e) -
> {

        response.setContentType("application/json;charset=UTF-8");

        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.getWriter().write("尚未认证，请进行认证操作！");
    })
            .accessDeniedHandler((request, response, e) → {

        response.setContentType("application/json;charset=UTF-8");

        response.setStatus(HttpStatus.FORBIDDEN.value());
        response.getWriter().write("无权访问！");
    });
}
}
```

第十一章 授权

- 什么是权限管理
- 权限管理核心概念
- Spring Security 权限管理策略
- 基于 URL 地址的权限管理
- 基于方法的权限管理

- 实战

权限管理

认证

身份认证，就是判断一个用户是否为合法用户的处理过程。Spring Security 中支持多种不同方式的认证，但是无论开发者使用那种方式认证，都不会影响授权功能使用。因为 Spring Security 很好做到了认证和授权解耦。

授权

授权，即访问控制，控制谁能访问哪些资源。简单的理解授权就是根据系统提前设置好的规则，给用户分配可以访问某一个资源的权限，用户根据自己所具有权限，去执行相应操作。

授权核心概念

在前面学习认证过程中，我们得知认证成功之后会将当前登录用户信息保存到 Authentication 对象中，Authentication 对象中有一个 `getAuthorities()` 方法，用来返回当前登录用户具备的权限信息，也就是当前用户具有权限信息。该方法的返回值为 `Collection<? extends GrantedAuthority>`，当需要进行权限判断时，就根据集合返回权限信息调用相应方法进行判断。

```
public interface Authentication extends Principal, Serializable {  
  
    /** Set by an <code>AuthenticationManager</code> to indicate the authorities that the ...*/  
    Collection<? extends GrantedAuthority> getAuthorities();  
}
```

那么问题来了，针对于这个返回值 `GrantedAuthority` 应该如何理解呢？是角色还是权限？

我们针对于授权可以是**基于角色权限管理** 和 **基于资源权限管理**，从设计层面上来说，角色和权限是两个完全不同的东西：权限是一些具体操作，角色则是某些权限集合。如：`READ_BOOK` 和 `ROLE_ADMIN` 是完全不同的。因此至于返回值是什么取决于你的业务设计情况：

- 基于角色权限设计就是： 用户 \leftrightarrow 角色 \leftrightarrow 资源 三者关系 返回就是用户的 **角色**
- 基于资源权限设计就是： 用户 \leftrightarrow 权限 \leftrightarrow 资源 三者关系 返回就是用户的 **权限**
- 基于角色和资源权限设计就是： 用户 \leftrightarrow 角色 \leftrightarrow 权限 \leftrightarrow 资源 返回统称为用户的 **权限**

为什么可以统称为权限，因为从代码层面角色和权限没有太大不同都是权限，特别是在 Spring Security 中，角色和权限处理方式基本上都是相同的。唯一区别 SpringSecurity 在很多时候会自动给角色添加一个 **ROLE_** 前缀，而权限则不会自动添加。

权限管理策略

Spring Security 中提供的权限管理策略主要有两种类型：

- 基于过滤器(URL)的权限管理 (**FilterSecurityInterceptor**)
 - 基于过滤器的权限管理主要是用来拦截 HTTP 请求，拦截下来之后，根据 HTTP 请求地址进行权限校验。
- 基于 AOP (方法)的权限管理 (**MethodSecurityInterceptor**)
 - 基于 AOP 权限管理主要是用来处理方法级别的权限问题。当需要调用某一个方法时，通过 AOP 将操作拦截下来，然后判断用户是否具备相关的权限。

基于 URL 权限管理

- 开发 controller

```
@RestController
public class DemoController {

    @GetMapping("/admin")
    public String admin() {
        return "admin ok";
    }

    @GetMapping("/user")
    public String user() {
        return "user ok";
    }

    @GetMapping("/getInfo")
    public String getInfo() {
        return "info ok";
    }
}
```

- 配置授权

```
package com.blr.config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationBuilder;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    //创建内存数据源
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager =
new InMemoryUserDetailsManager();

        inMemoryUserDetailsManager.createUser(User.withUsername("root").p
assword("{noop}123").roles("ADMIN").build());

        inMemoryUserDetailsManager.createUser(User.withUsername("win7").p
assword("{noop}123").roles("USER").build());

        inMemoryUserDetailsManager.createUser(User.withUsername("lisi").p
assword("{noop}123").roles("READ_BOOK").build());
    }
}
```

```

        return inMemoryUserDetailsManager;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
    auth.userDetailsService(userDetailsService());
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasAnyRole("USER",
"ADMIN")
        .antMatchers("/getInfo").hasRole("READ_BOOK")
        .anyRequest().authenticated()
        .and().formLogin()
        .and().csrf().disable();
}
}

```

- 启动项目测试

权限表达式

The screenshot shows the JavaDoc browser in IntelliJ IDEA displaying the `SecurityExpressionOperations` interface. The interface is annotated with `@FunctionalInterface`. It contains the following methods:

- `getAuthentication(): Authentication`
- `hasAuthority(String): boolean`
- `hasAnyAuthority(String...): boolean`
- `hasRole(String): boolean`
- `hasAnyRole(String...): boolean`
- `permitAll(): boolean`
- `denyAll(): boolean`
- `isAnonymous(): boolean`
- `isAuthenticated(): boolean`
- `isRememberMe(): boolean`
- `isFullyAuthenticated(): boolean`
- `hasPermission(Object, Object): boolean`
- `hasPermission(Object, String, Object): boolean`

方法	说明
<code>hasAuthority(String authority)</code>	当前用户是否具备指定权限
<code>hasAnyAuthority(String ... authorities)</code>	当前用户是否具备指定权限中任意一个
<code>hasRole(String role)</code>	当前用户是否具备指定角色
<code>hasAnyRole(String ... roles);</code>	当前用户是否具备指定角色中任意一个
<code>permitAll();</code>	放行所有请求/调用
<code>denyAll();</code>	拒绝所有请求/调用
<code>isAnonymous();</code>	当前用户是否是一个匿名用户
<code>isAuthenticated();</code>	当前用户是否已经认证成功
<code>isRememberMe();</code>	当前用户是否通过 Remember-Me 自动登录
<code>isFullyAuthenticated();</code>	当前用户是否既不是匿名用户又不是通过 Remember-Me 自动登录的
<code>hasPermission(Object targetId, Object permission);</code>	当前用户是否具备指定目标的指定权限信息
<code>hasPermission(Object targetId, String targetType, Object permission);</code>	当前用户是否具备指定目标的指定权限信息

基于 方法 权限管理

基于方法的权限管理主要是通过 AOP 来实现的，Spring Security 中通过 `MethodSecurityInterceptor` 来提供相关的实现。不同在于 `FilterSecurityInterceptor` 只是在请求之前进行前置处理，`MethodSecurityInterceptor` 除了前置处理之外还可以进行后置处理。前置处理就是在请求之前判断是否具备相应的权限，后置处理则是对方法的执行结果进行二次过滤。前置处理和后置处理分别对应了不同的实现类。

@EnableGlobalMethodSecurity

EnableGlobalMethodSecurity 该注解是用来开启权限注解，用法如下：

```
@Configuration  
@EnableGlobalMethodSecurity(prePostEnabled=true, securedEnabled=true,  
    jsr250Enabled=true)  
public class SecurityConfig extends WebSecurityConfigurerAdapter{}
```

- **perPostEnabled**: 开启 Spring Security 提供的四个权限注解，`@PostAuthorize`、`@PostFilter`、`@PreAuthorize` 以及`@PreFilter`。
- **securedEnabled**: 开启 Spring Security 提供的 `@Secured` 注解支持，该注解不支持权限表达式
- **jsr250Enabled**: 开启 JSR-250 提供的注解，主要是`@DenyAll`、`@PermitAll`、`@RolesAll` 同样这些注解也不支持权限表达式

以上注解含义如下：

- `@PostAuthorize`: 在目前标方法执行之后进行权限校验。
- `@PostFilter`: 在目标方法执行之后对方法的返回结果进行过滤。
- `@PreAuthorize`: 在目标方法执行之前进行权限校验。
- `@PreFilter`: 在目前标方法执行之前对方法参数进行过滤。
- `@Secured`: 访问目标方法必须具备相应的角色。
- `@DenyAll`: 拒绝所有访问。
- `@PermitAll`: 允许所有访问。
- `@RolesAllowed`: 访问目标方法必须具备相应的角色。

这些基于方法的权限管理相关的注解，一般来说只要设置 `prePostEnabled=true` 就够用了。

基本用法

- 开启注解使用

```
@Configuration  
@EnableGlobalMethodSecurity(prePostEnabled=true, securedEnabled=true,  
    jsr250Enabled=true)  
public class SecurityConfig extends WebSecurityConfigurerAdapter{}
```

- 使用注解

```
@RestController
@RequestMapping("/hello")
public class AuthorizeMethodController {

    @PreAuthorize("hasRole('ADMIN') and
authentication.name='root'")
    @GetMapping
    public String hello() {
        return "hello";
    }

    @PreAuthorize("authentication.name=#name")
    @GetMapping("/name")
    public String hello(String name) {
        return "hello:" + name;
    }

    @PreFilter(value = "filterObject.id%2!=0",filterTarget =
"users")
    @PostMapping("/users") //filterTarget 必须是 数组 集合
    public void addUsers(@RequestBody List<User> users) {
        System.out.println("users = " + users);
    }

    @PostAuthorize("returnObject.id=1")
    @GetMapping("/userId")
    public User getUserById(Integer id) {
        return new User(id, "blr");
    }

    @PostFilter("filterObject.id%2==0")
    @GetMapping("/lists")
    public List<User> getAll() {
        List<User> users = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            users.add(new User(i, "blr:" + i));
        }
        return users;
    }
}
```

```
}

@Secured({"ROLE_USER"}) //只能判断角色
@GetMapping("/secured")
public User getUserByUsername() {
    return new User(99, "secured");
}

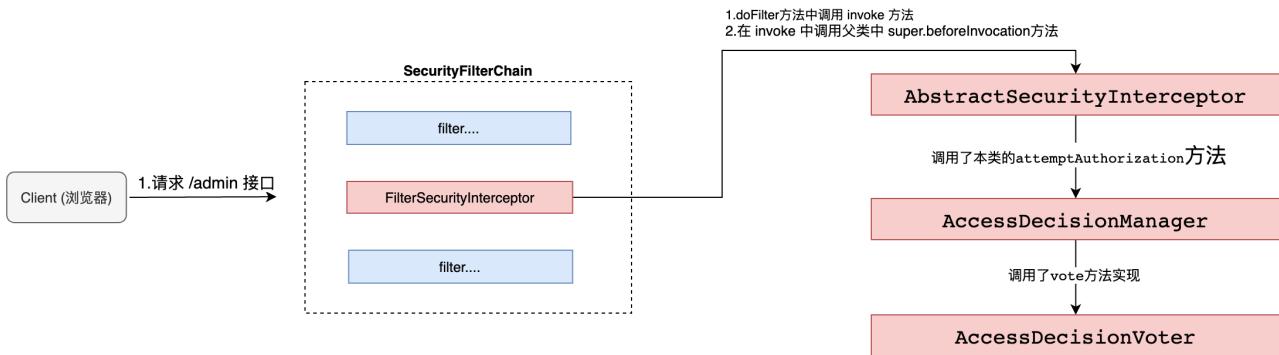
{@Secured({"ROLE_ADMIN", "ROLE_USER"}) //具有其中一个即可
@GetMapping("/username")
public User getUserByUsername2(String username) {
    return new User(99, username);
}

@PermitAll
@GetMapping("/permitAll")
public String permitAll() {
    return "PermitAll";
}

@DenyAll
@GetMapping("/denyAll")
public String denyAll() {
    return "DenyAll";
}

@RolesAllowed({"ROLE_ADMIN", "ROLE_USER"}) //具有其中一个角色即可
@GetMapping("/rolesAllowed")
public String rolesAllowed() {
    return "RolesAllowed";
}
```

原理分析



- **ConfigAttribute** 在 Spring Security 中，用户请求一个资源(通常是一个接口或者一个 Java 方法)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。
- **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。

实战

在前面的案例中，我们配置的 URL 拦截规则和请求 URL 所需要的权限都是通过代码来配置的，这样就比较死板，如果想要调整访问某一个 URL 所需要的权限，就需要修改代码。

动态管理权限规则就是我们将 URL 拦截规则和访问 URI 所需要的权限都保存在数据库中，这样，在不修改源代码的情况下，只需要修改数据库中的数据，就可以对权限进行调整。

用户←中间表→ 角色 ←中间表→ 菜单

库表设计

```
SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;
```

```
-- -----
-- Table structure for menu
-- -----  
  
DROP TABLE IF EXISTS `menu`;  
CREATE TABLE `menu` (  
    `id` int(11) NOT NULL AUTO_INCREMENT,  
    `pattern` varchar(128) DEFAULT NULL,  
    PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;  
  
-- -----  
-- Records of menu  
-- -----  
  
BEGIN;  
INSERT INTO `menu` VALUES (1, '/admin/**');  
INSERT INTO `menu` VALUES (2, '/user/**');  
INSERT INTO `menu` VALUES (3, '/guest/**');  
COMMIT;  
  
-- -----  
-- Table structure for menu_role
-- -----  
  
DROP TABLE IF EXISTS `menu_role`;  
CREATE TABLE `menu_role` (  
    `id` int(11) NOT NULL AUTO_INCREMENT,  
    `mid` int(11) DEFAULT NULL,  
    `rid` int(11) DEFAULT NULL,  
    PRIMARY KEY (`id`),  
    KEY `mid` (`mid`),  
    KEY `rid` (`rid`),  
    CONSTRAINT `menu_role_ibfk_1` FOREIGN KEY (`mid`) REFERENCES  
        `menu` (`id`),  
    CONSTRAINT `menu_role_ibfk_2` FOREIGN KEY (`rid`) REFERENCES  
        `role` (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;  
  
-- -----  
-- Records of menu_role
```

```
-- -----
BEGIN;
INSERT INTO `menu_role` VALUES (1, 1, 1);
INSERT INTO `menu_role` VALUES (2, 2, 2);
INSERT INTO `menu_role` VALUES (3, 3, 3);
INSERT INTO `menu_role` VALUES (4, 3, 2);
COMMIT;

-- -----
-- Table structure for role
-- -----

DROP TABLE IF EXISTS `role`;
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 AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- -----
-- Records of role
-- -----

BEGIN;
INSERT INTO `role` VALUES (1, 'ROLE_ADMIN', '系统管理员');
INSERT INTO `role` VALUES (2, 'ROLE_USER', '普通用户');
INSERT INTO `role` VALUES (3, 'ROLE_GUEST', '游客');
COMMIT;

-- -----
-- Table structure for user
-- -----

DROP TABLE IF EXISTS `user`;
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,
  `locked` tinyint(1) DEFAULT NULL,
```

```

PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- 
-- Records of user
-- 

BEGIN;
INSERT INTO `user` VALUES (1, 'admin', '{noop}123', 1, 0);
INSERT INTO `user` VALUES (2, 'user', '{noop}123', 1, 0);
INSERT INTO `user` VALUES (3, 'blr', '{noop}123', 1, 0);
COMMIT;

-- 
-- Table structure for user_role
-- 

DROP TABLE IF EXISTS `user_role`;
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`),
  CONSTRAINT `user_role_ibfk_1` FOREIGN KEY (`uid`) REFERENCES `user` (`id`),
  CONSTRAINT `user_role_ibfk_2` FOREIGN KEY (`rid`) REFERENCES `role` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

-- 
-- Records of user_role
-- 

BEGIN;
INSERT INTO `user_role` VALUES (1, 1, 1);
INSERT INTO `user_role` VALUES (2, 1, 2);
INSERT INTO `user_role` VALUES (3, 2, 2);
INSERT INTO `user_role` VALUES (4, 3, 3);
COMMIT;

```

```
SET FOREIGN_KEY_CHECKS = 1;
```

创建 springboot 应用

- 引入依赖

```
<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>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.8</version>
</dependency>
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.2</version>
</dependency>
```

- 配置配置文件

```
server.port=8080
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/security?
characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root
mybatis.mapper-locations=classpath:com/blr/mapper/*.xml
mybatis.type-aliases-package=com.blr.entity
```

- 创建实体类

```
public class User implements UserDetails {
    private Integer id;
    private String password;
    private String username;
    private boolean enabled;
    private boolean locked;
    private List<Role> roles;

    @Override
    public Collection<? extends GrantedAuthority>
getAuthorities() {
        return roles.stream().map(r → new
SimpleGrantedAuthority(r.getName())).collect(Collectors.toList());
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }
```

```
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return !locked;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return enabled;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    public void setLocked(boolean locked) {
        this.locked = locked;
    }
```

```
}

public void setRoles(List<Role> roles) {
    this.roles = roles;
}

public Integer getId() {
    return id;
}

public List<Role> getRoles() {
    return roles;
}

}
```

```
public class Role {
    private Integer id;
    private String name;
    private String nameZh;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNameZh() {
        return nameZh;
    }
}
```

```
}

public void setNameZh(String nameZh) {
    this.nameZh = nameZh;
}

}
```

```
public class Menu {
    private Integer id;
    private String pattern;
    private List<Role> roles;

    public List<Role> getRoles() {
        return roles;
    }

    public void setRoles(List<Role> roles) {
        this.roles = roles;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getPattern() {
        return pattern;
    }

    public void setPattern(String pattern) {
        this.pattern = pattern;
    }
}
```

- 创建 mapper 接口

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

```
@Mapper
public interface MenuMapper {
    List<Menu> getAllMenu();
}
```

- 创建 mapper 文件

```
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.blr.mapper.UserMapper">

    <select id="loadUserByUsername"
resultType="com.blr.entity.User">
        select *
        from user
        where username = #{username};
    </select>

    <select id="getUserRoleByUid"
resultType="com.blr.entity.Role">
        select r.*
        from role r,
             user_role ur
        where ur.uid = #{uid}
            and ur.rid = r.id
    </select>
</mapper>

<mapper namespace="com.blr.mapper.MenuMapper">
    <resultMap id="MenuResultMap" type="com.blr.entity.Menu">
```

```

<id property="id" column="id"/>
<result property="pattern" column="pattern"></result>
<collection property="roles"
ofType="com.blr.entity.Role">
    <id column="rid" property="id"/>
    <result column="rname" property="name"/>
    <result column="rnameZh" property="nameZh"/>
</collection>
</resultMap>

<select id="getAllMenu" resultMap="MenuResultMap">
    select m.*, r.id as rid, r.name as rname, r.nameZh as
rnameZh
    from menu m
        left join menu_role mr on m.`id` = mr.`mid`
        left join role r on r.`id` = mr.`rid`
</select>
</mapper>

```

- 创建 service 接口

```

@Service
public class UserService implements UserDetailsService {
    private final UserMapper userMapper;

    @Autowired
    public UserService(UserMapper userMapper) {
        this.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.getUserRoleByUid(user.getId()));
        return user;
    }
}
```

```
@Service
public class MenuService {
    private final MenuMapper menuMapper;

    @Autowired
    public MenuService(MenuMapper menuMapper) {
        this.menuMapper = menuMapper;
    }

    public List<Menu> getAllMenu() {
        return menuMapper.getAllMenu();
    }
}
```

- 创建测试 controller

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

    @GetMapping("/user/hello")
    public String user() {
        return "hello user";
    }

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

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

- 创建 CustomSecurityMetadataSource

```
@Component
public class CustomSecurityMetadataSource implements
FilterInvocationSecurityMetadataSource {
    private final MenuService menuService;

    @Autowired
    public CustomSecurityMetadataSource(MenuService
menuService) {
        this.menuService = menuService;
    }

    AntPathMatcher antPathMatcher = new AntPathMatcher();

    @Override
    public Collection<ConfigAttribute> getAttributes(Object
object) throws IllegalArgumentException {
        String requestURI = ((FilterInvocation)
object).getRequest().getRequestURI();
        List<Menu> allMenu = menuService.getAllMenu();
        for (Menu menu : allMenu) {
            if (antPathMatcher.match(menu.getPattern(),
requestURI)) {
                String[] roles = menu.getRoles().stream().map(r
→ r.getName()).toArray(String[]::new);
                return SecurityConfig.createList(roles);
            }
        }
        return null;
    }
}
```

```
    @Override
    public Collection<ConfigAttribute> getAllConfigAttributes()
    {
        return null;
    }

    @Override
    public boolean supports(Class<?> clazz) {
        return FilterInvocation.class.isAssignableFrom(clazz);
    }
}
```

- 配置 Security 配置

```
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {
    private final CustomSecurityMetadataSource
customSecurityMetadataSource;
    private final UserService userService;

    @Autowired
    public SecurityConfig(CustomSecurityMetadataSource
customSecurityMetadataSource, UserService userService) {
        this.customSecurityMetadataSource =
customSecurityMetadataSource;
        this.userService = userService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.userDetailsService(userService);
    }

    @Override
```

```
protected void configure(HttpSecurity http) throws
Exception {
    ApplicationContext applicationContext =
http.getSharedObject(ApplicationContext.class);
    http.apply(new UrlAuthorizationConfigurer<
(applicationContext))
        .withObjectPostProcessor(new
ObjectPostProcessor<FilterSecurityInterceptor>() {
            @Override
            public <O extends
FilterSecurityInterceptor> O postProcess(O object) {
                object.setSecurityMetadataSource(customSecurityMetadataSource)
;

                object.setRejectPublicInvocations(true);
                return object;
            }
        });
    http.formLogin()
        .and()
        .csrf().disable();
}
}
```

- 启动入口类进行测试

第十二章 OAuth2

如今在很多互联网应用中，OAuth2 是一个非常重要的认证协议，很多场景下都会用到它，Spring Security 对 OAuth2 协议提供了相应的支持，开发者可以非常方便的使用 OAuth2 协议。

- OAuth2 简介
- 四种授权模式
- Spring Security OAuth2
- GitHub 授权登录
- 授权服务器与资源服务器

- 使用 JWT

OAuth2 简介

OAuth 是一个开放标准，该标准允许用户让第三方应用访问该用户在某一网站上存储的私密资源（如头像、照片、视频等），并且在这个过程中无须将用户名和密码提供给第三方应用。通过令牌（token）可以实现这一功能，每一个令牌授权一个特定的网站在特定的时段内允许特定的资源。

OAuth 让用户可以授权第三方网站灵活访问它们存储在另外一些资源服务器上的特定信息，而非所有内容。对于用户而言，我们在互联网应用中最常见的 OAuth 应用就是各种第三方登录，例如QQ授权登录、微信授权登录、微博授权登录、GitHub 授权登录等。

例如用户想通过 QQ 登录今日头条，这时今日头条就是一个第三方应用，今日头条需要访问用户存储在 QQ 服务器上的一些基本信息，就需要得到用户的授权。如果用户把自己的 QQ 用户名/密码告诉今日头条，那么今日头条就能访问用户存储在 QQ 服务器上的所有数据，并且用户只有修改密码才能收回授权，这种授权方式安全隐患很大，如果使用 OAuth 协议就能很好地解决这一问题。

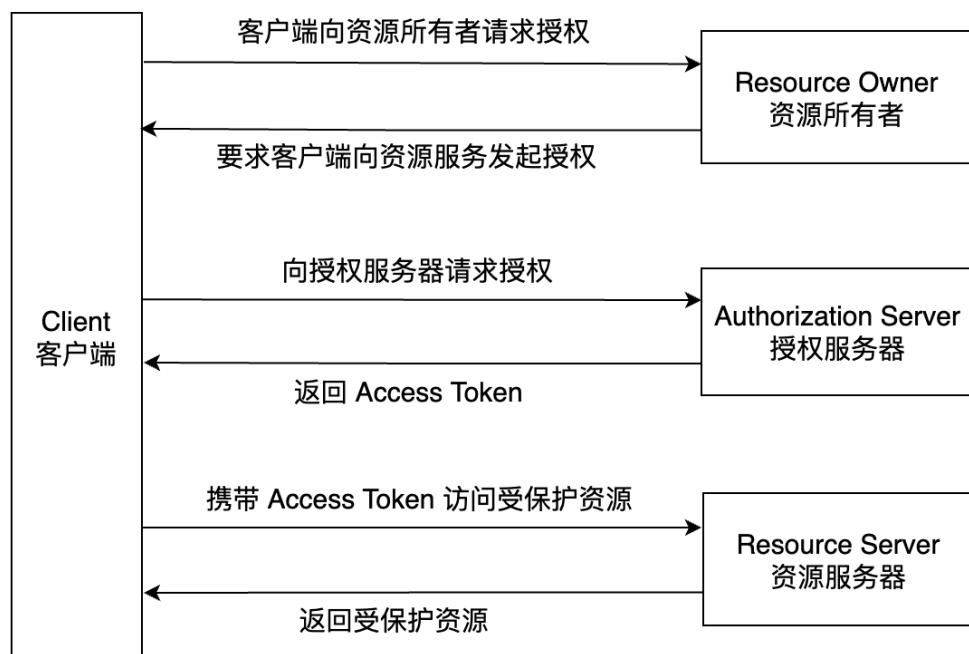
OAuth2 是 OAuth 协议的下一版本，但不兼容 OAuth 1.0。 OAuth2 关注客户端开发者的简易性，同时为 Web 应用、桌面应用、移动设备、IoT 设备提供专门的认证流程。

四种授权模式

OAuth2 协议一共支持四种不同的授权模式：

1. **授权码模式**：常见的第三方平台登录功能基本都是使用这种模式。
2. **简化模式**：简化模式是不需要第三方服务端参与，直接在浏览器中向授权服务器申请令牌（token），如果网站是纯静态页面，则可以采用这种方式。
3. **密码模式**：密码模式是用户把用户名/密码直接告诉客户端，客户端使用这些信息后授权服务器申请令牌（token）。这需要用户对客户端高度信任，例如客户端应用和服务提供商就是同一家公司。
4. **客户端模式**：客户端模式是指客户端使用自己的名义而不是用户的名义向服务提供者申请授权。严格来说，客户端模式并不能算作 OAuth 协议解决问题的一种解决方案，但是对于开发者而言，在一些为移动端提供的授权服务器上使用这种模式还是非常方便的。

无论哪种授权模式，其授权流程都是相似的，只不过在个别步骤上有差异而已。整体流程如下：



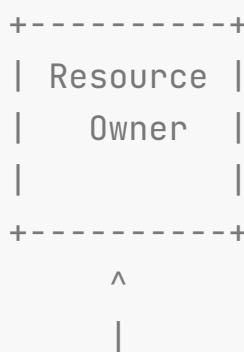
从上图中我们可以看出 OAuth2 中包含四种不同的角色：

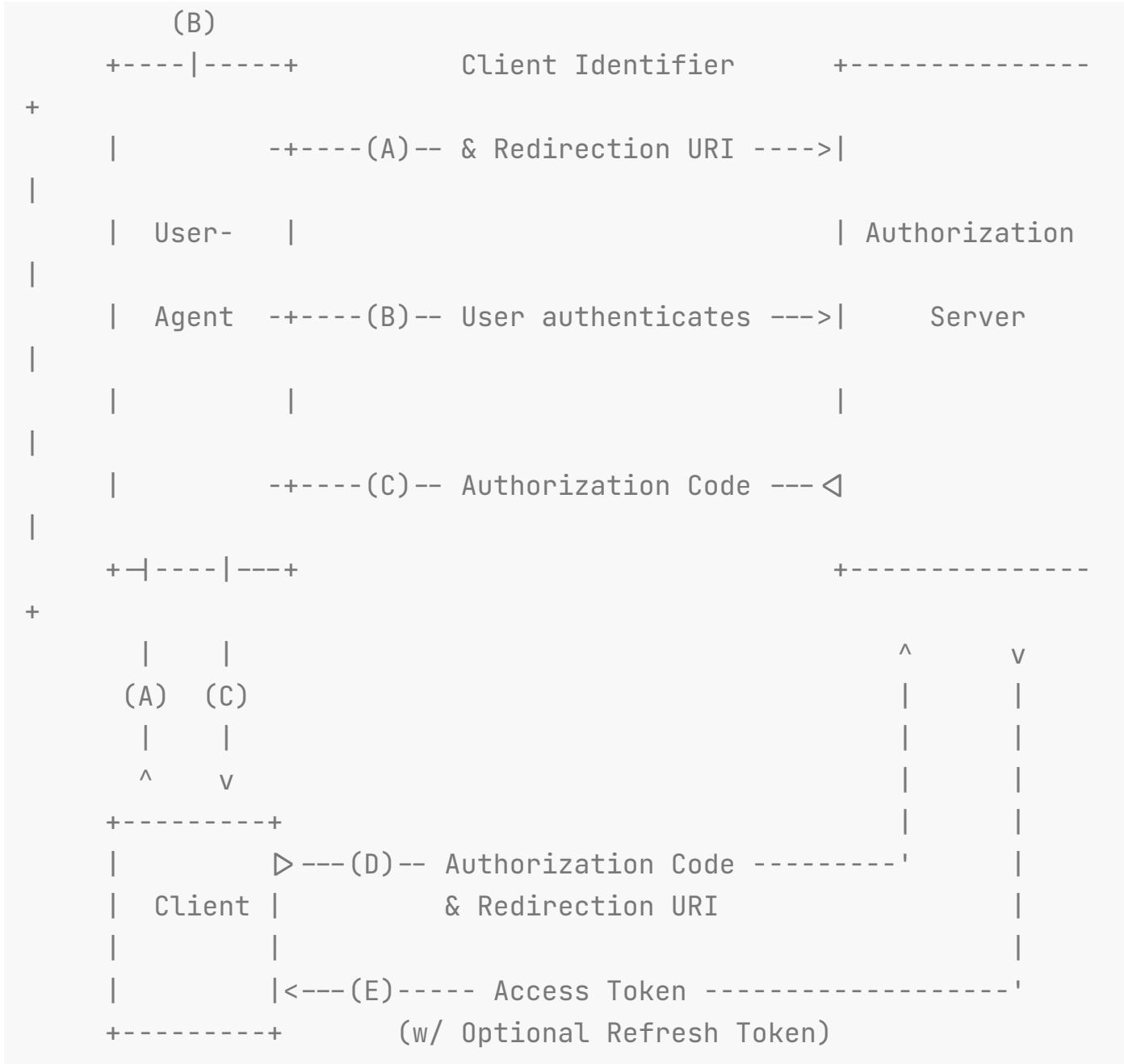
- **Client**: 第三方应用。
- **Resource Owner**: 资源所有者。
- **Authorization Server** : 授权服务器。
- **Resource Server**: 资源服务器。

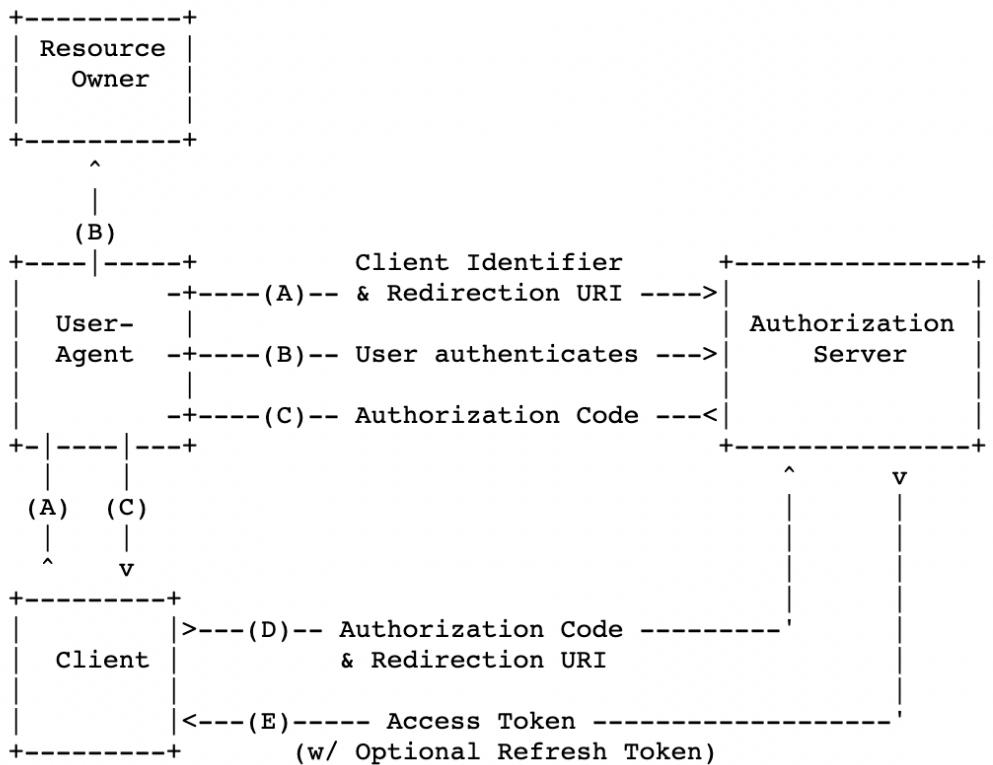
授权码模式

授权码模式 (Authorization Code) 是最安全并且使用最广泛的一种OAuth2授权模式。同时也是最复杂的一种授权模式，其具体的授权流程如图 15-2 所示（图片来自 RFC6749 文档

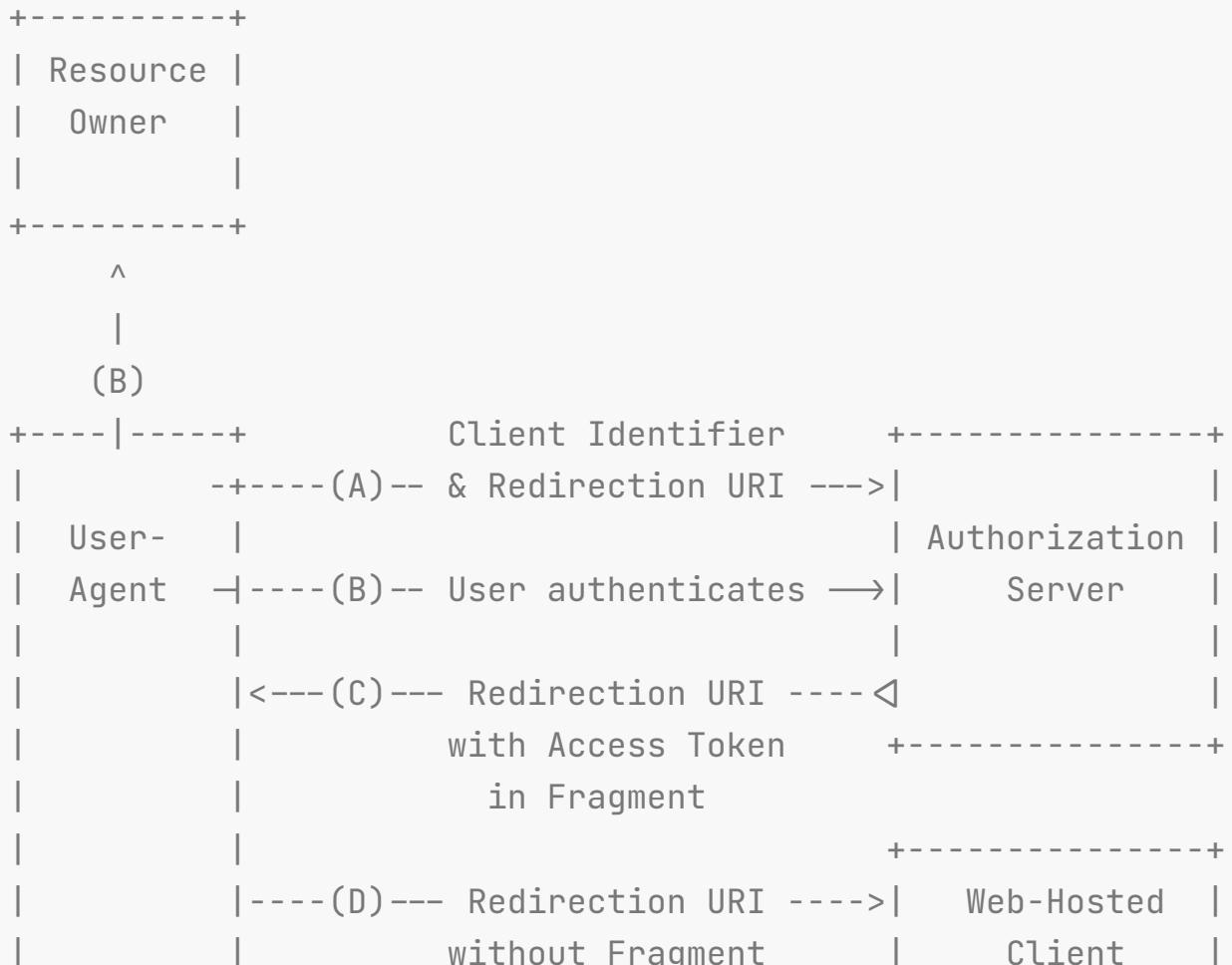
<https://tools.ietf.org/html/rfc6749>）

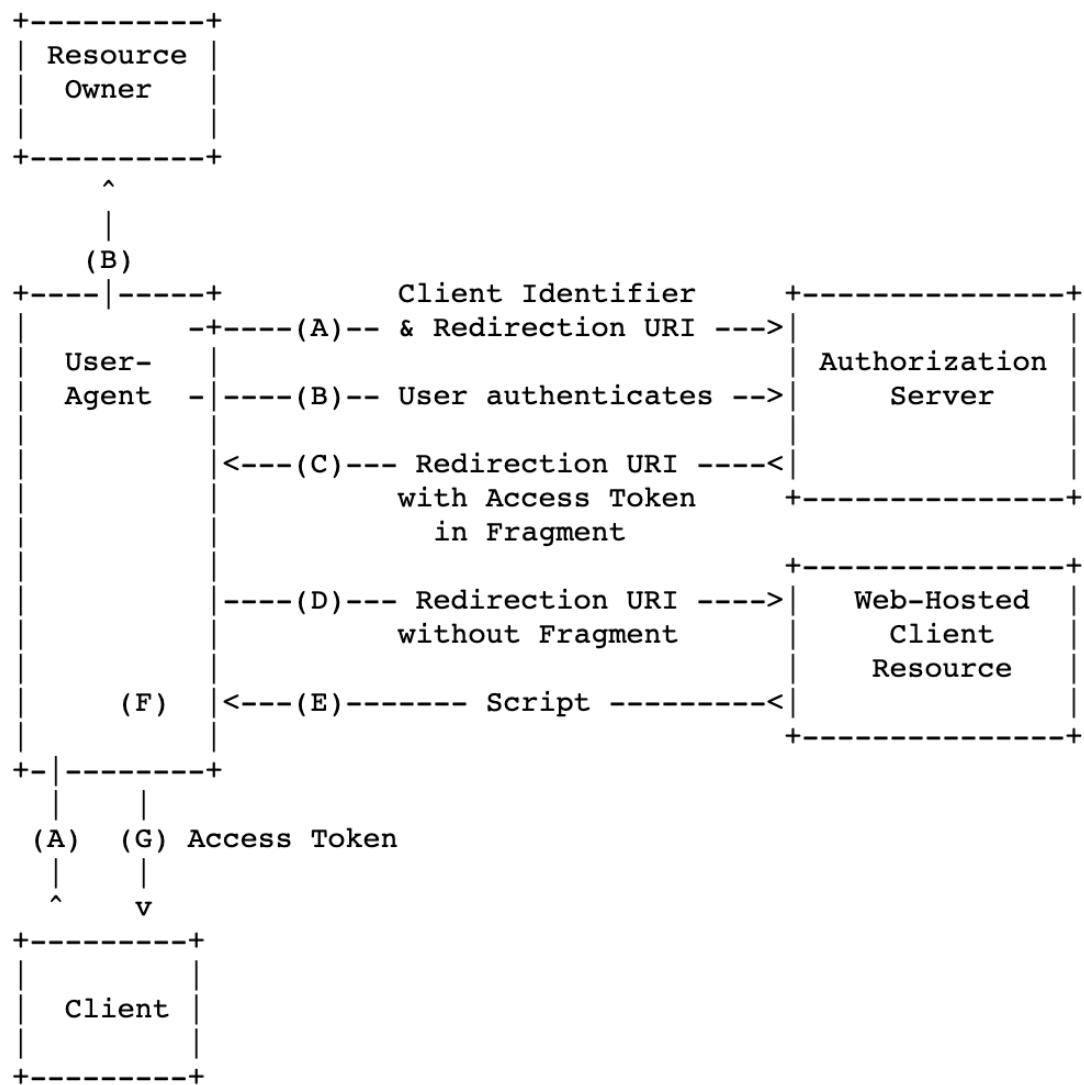




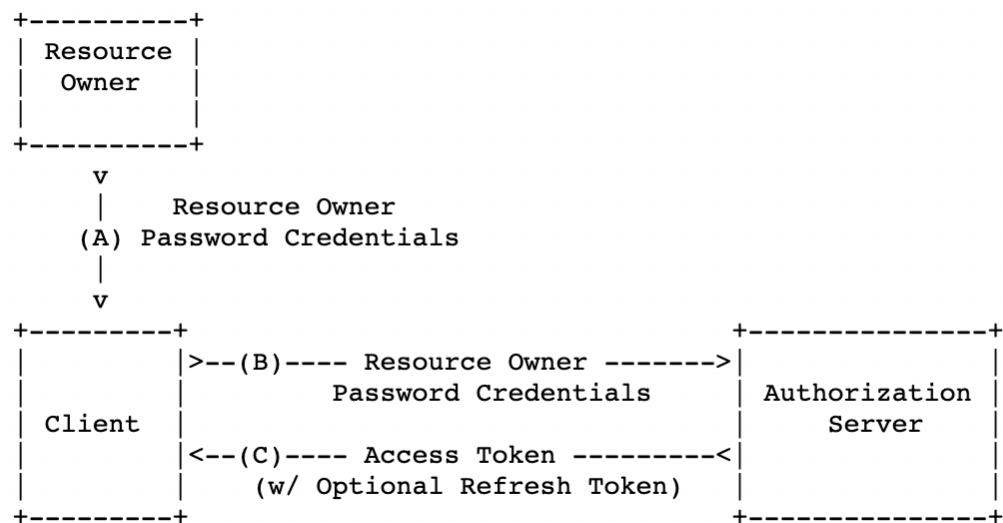
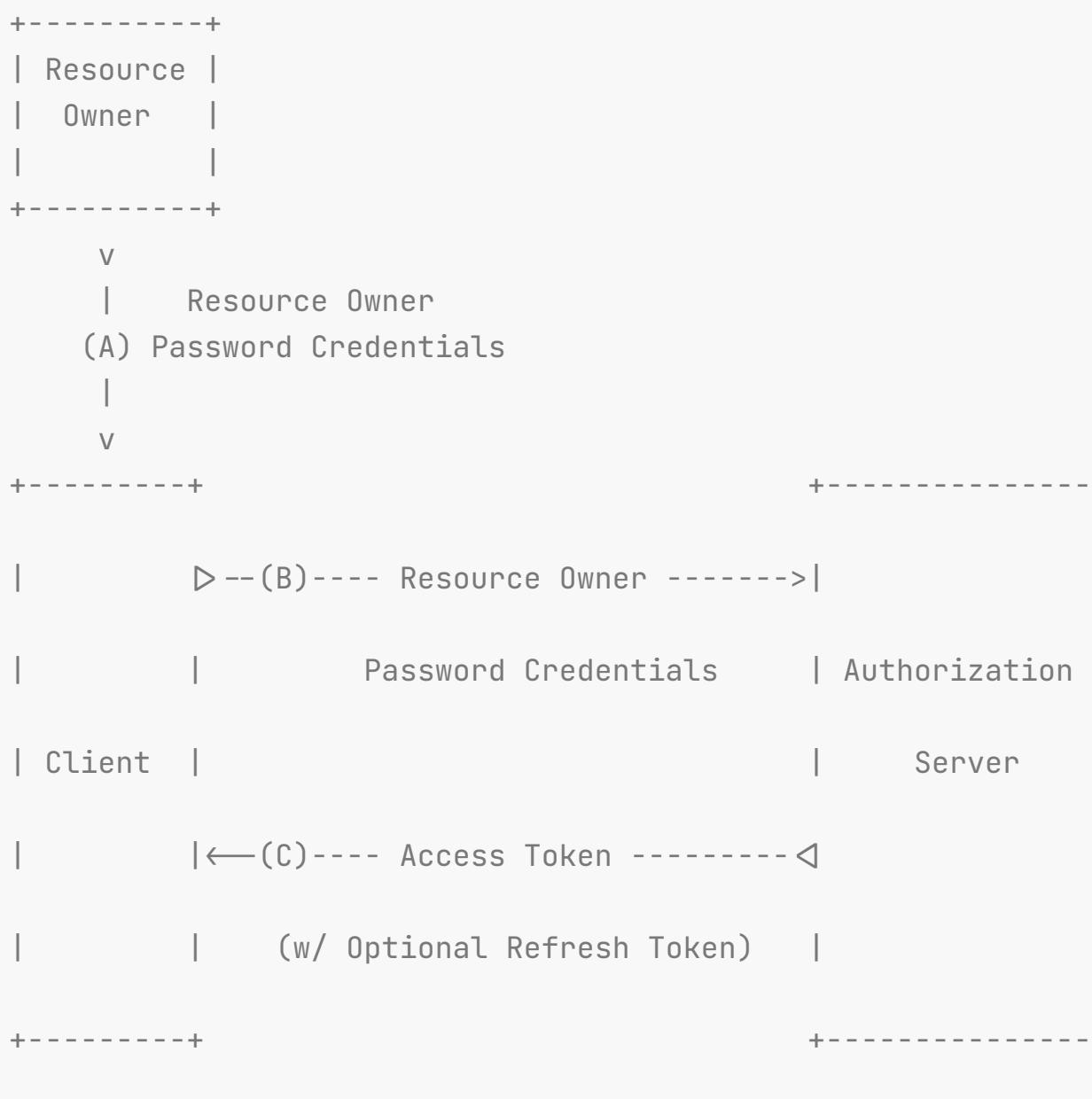


简化模式

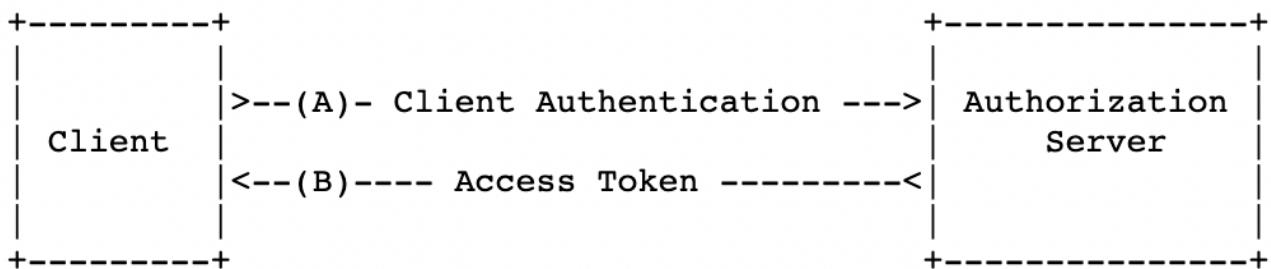




密码模式



客户端模式



Spring Security OAuth2

Spring Security 对 OAuth2 提供了很好的支持，这使得我们在 Spring Security 中使用 OAuth2 非常地方便。然而由于历史原因，Spring Security 对 OAuth2 的支持比较混乱，这里简单梳理一下。

大约十年前，Spring 引入了一个社区驱动的开源项目 Spring Security OAuth，并将其纳入 Spring 项目组合中。到今天为止，这个项目已经发展成为一个成熟的项目，可以支持大部分 OAuth 规范，包括资源服务器、客户端和授权服务器等。

然而早期的项目存在一些问题，例如：

- OAuth 是在早期完成的，开发者无法预料未来的变化以及这些代码到底要被怎么使用，

这导致很多 Spring 项目提供了自己的 OAuth 支持，也就带来了 OAuth 支持的碎片化。

- 最早的OAuth项目同时支持 OAuth1.0 和 OAuth2.0，而现在OAuth1.0 早已经不再使用，
可以放弃了。
- 现在我们有更多的库可以选择，可以在这些库的基础上去开发，以便更好地支持JWT等新技术。

基于以上这些原因，官方决定重写 Spring Security OAuth，以便更好地协调 Spring 和OAuth，并简化代码库，使Spring 的 OAuth 支持更加灵活。

然而，在重写的过程中，发生了不少波折。

2018年1月30日，Spring 官方发了一个通知，表示要逐渐停止现有的 OAuth2支持，然后在 Spring Security 5中构建下一代 OAuth2.0 支持。这么做的原因是当时 OAuth2 的落地方案比较混乱，在 Spring Security OAuth、Spring Cloud Security、Spring Boot 1.5.x 以及当时最新的Spring Security 5.x 中都提供了对 OAuth2 的实现。以至于当开发者需要使用 OAuth2 时，不得不问，到底选哪一个依赖合适呢？

所以Spring 官方决定有必要将 OAuth2.0 的支持统一到一个项目中，以便为用户提供明确的选择，并避免任何潜在的混乱，同时 OAuth2.0 的开发文档也要重新编写，以方便开发人员学习。所有的决定将在 Spring Security 5 中开始，构建下一代 OAuth2.0的支持。

从那个时候起，Spring Security OAuth 项目就正式处于维护模式。官方将提供至少一年的

错误/安全修复程序，并且会考虑添加次要功能，但不会添加主要功能。同时将 Spring Security OAuth中的所有功能重构到 Spring Security 5.x 中。

到了2019年11月14日，Spring 官方又发布一个通知，这次的通知首先表示 Spring Security

OAuth 在迁往 Spring Security 5.x 的过程非常顺利，大部分迁移工作已经完成了，剩下的将在5.3 版本中完成迁移，在迁移的过程中还添加了许多新功能。包括对 OpenID Connect1.0 的支持。同时还宣布将不再支持授权服务器，不支持的原因有两个：

1. 在2019年，已经有大量的商业和开源授权服务器可用。
2. 授权服务器是使用一个库来构建产品，而 Spring Security 作为框架，并不适合做这件事情。

一石激起千层浪，许多开发者表示对此难以接受。这件事也在Spring 社区引发了激烈的讨论，好在 Spring 官方愿意倾听来自社区的声音。

到了2020年4月15日，Spring 官方宣布启动 Spring Authorization server 项目。这是一个由 Spring Security 团队领导的社区驱动的项目，致力于向 Spring 社区提供 Authorization Server 支持，也就是说，Spring 又重新支持授权服务器了。

2020年8月21日，Spring Authorization Server 0.0.1 正式发布！

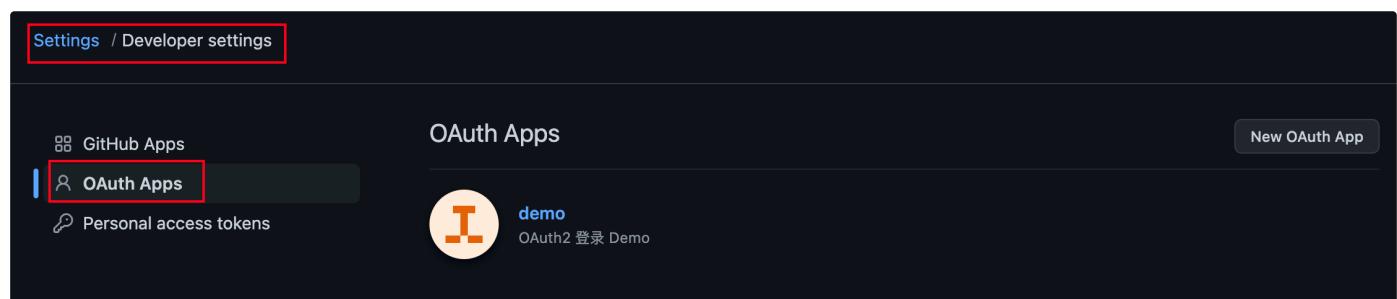
这就是 OAuth2 在Spring 家族中的发展历程了。在后面的学习中，客户端和资源服务器都将采用最新的方式来构建，授权服务器依然采用旧的方式来构建，因为目前的 Spring Authorization Server 0.0.1 功能较少且 BUG 较多。

一般来说，当我们在项目中使用 OAuth2 时，都是开发客户端，授权服务器和资源服务器都是由外部提供。例如我们想在自己搭建网站上集成 GitHub 第三方登录，只需要开发自己的客户端即可，认证服务器和授权服务器都是由 GitHub 提供的。

GitHub 授权登录

创建 OAuth 应用

访问 github 并登录，在`https://github.com/settings/profile`中找到 Developer Settings 选项



- 创建 OAuth App并输入一下基本信息：

Register a new OAuth application

Application name *

demo

Something users will recognize and trust.

Homepage URL *

http://localhost:8080

The full URL to your application homepage.

Application description

OAuth2.0 登录

This is displayed to all users of your application.

Authorization callback URL *

http://localhost:8080/login/oauth2/code/github

Your application's callback URL. Read our [OAuth documentation](#) for more information.

Enable Device Flow

Allow this OAuth App to authorize users via the Device Flow.

Read the [Device Flow documentation](#) for more information.

Register application

Cancel

- 注册成功后会获取到对应的 Client ID 和 Client Secret。

demo



blr-1993 owns this application.

[Transfer ownership](#)

You can list your application in the [GitHub Marketplace](#) so that other users can discover it.

[List this application in the Marketplace](#)

0 users

[Revoke all user tokens](#)

Client ID

f722bc8444ed6a95cdc7

Client secrets

[Generate a new client secret](#)

Make sure to copy your new client secret now. You won't be able to see it again.

✓ 34c53b93eb8a63984d75a9f4c3740e9542d6a6b0 [Copy](#)

Added now by blr-1993

Never used

[Delete](#)

You cannot delete the only client secret. Generate a new client secret first.

项目开发

- 创建 springboot 应用，并引入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<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>
```

- 创建测试 controller

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public DefaultOAuth2User hello(){
        System.out.println("hello ");
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        return (DefaultOAuth2User) authentication.getPrincipal();
    }
}
```

- 配置 security

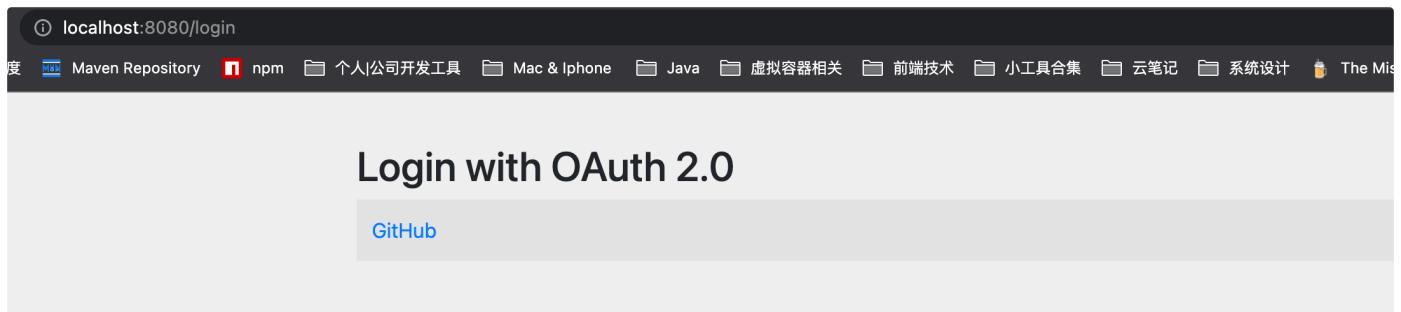
```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2Login();
    }
}
```

- 配置配置文件

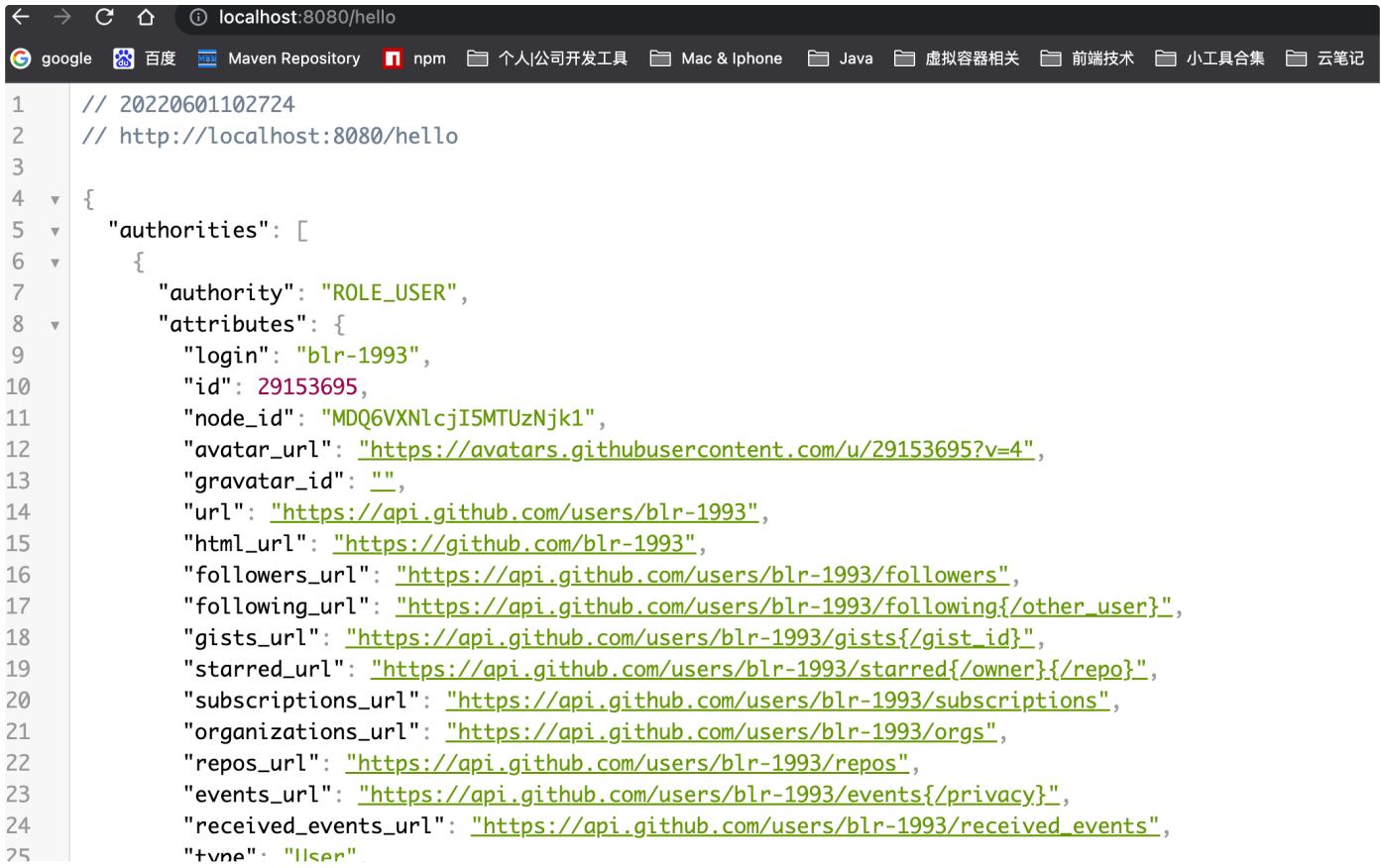
```
server.port=8080
```

```
spring.security.oauth2.client.registration.github.client-
id=d6ea299b9ade3cd3b97d
spring.security.oauth2.client.registration.github.client-
secret=aaa44b2675a7b636b1b43371e509e88ee9013816
# 一定要与重定向回调 URL 一致
spring.security.oauth2.client.registration.github.redirect-
uri=http://localhost:8080/login/oauth2/code/github
```

- 启动测试



- 点击 github 登录,点击授权 访问 hello 接口



```
// 20220601102724
// http://localhost:8080/hello

{
  "authorities": [
    {
      "authority": "ROLE_USER",
      "attributes": {
        "login": "blr-1993",
        "id": 29153695,
        "node_id": "MDQ6VXNlcjI5MTUzNjk1",
        "avatar_url": "https://avatars.githubusercontent.com/u/29153695?v=4",
        "gravatar_id": "",
        "url": "https://api.github.com/users/blr-1993",
        "html_url": "https://github.com/blr-1993",
        "followers_url": "https://api.github.com/users/blr-1993/followers",
        "following_url": "https://api.github.com/users/blr-1993/following{/other_user}",
        "gists_url": "https://api.github.com/users/blr-1993/gists{/gist_id}",
        "starred_url": "https://api.github.com/users/blr-1993/starred{/owner}{/repo}",
        "subscriptions_url": "https://api.github.com/users/blr-1993/subscriptions",
        "organizations_url": "https://api.github.com/users/blr-1993/orgs",
        "repos_url": "https://api.github.com/users/blr-1993/repos",
        "events_url": "https://api.github.com/users/blr-1993/events{/privacy}",
        "received_events_url": "https://api.github.com/users/blr-1993/received_events",
        "type": "User"
      }
    }
  ]
}
```

授权、资源服务器

前面的 GitHub 授权登录主要向大家展示了 OAuth2 中客户端的工作模式。对于大部分的开发者而言，日常接触到的 OAuth2 都是开发客户端，例如接入 QQ 登录、接入微信登录等。不过也有少量场景，可能需要开发者提供授权服务器与资源服务器，接下来我们就通过一个完整的案例演示如何搭建授权服务器与资源服务器。

搭建授权服务器，我们可以选择一些现成的开源项目，直接运行即可，例如：

- Keycloak: RedHat 公司提供的开源工具，提供了很多实用功能，例如单点登录、支持OpenID、可视化后台管理等。
- Apache Oltu: Apache 上的开源项目，最近几年没怎么维护了。

接下来我们将搭建一个包含授权服务器、资源服务器以及客户端在内的 OAuth2 案例。

项目规划首先把项目分为三部分：

- 授权服务器：采用较早的spring-security-oauth2 来搭建授权服务器。
- 资源服务器：采用最新的 Spring Security 5.x 搭建资源服务器，
- 客户端：采用最新的 Spring Security5.x 搭建客户端。

使用 JWT