Spring Security

spring security可以实现认证和授权

spring security的验证流程

1. 从前端的表单得到用户密码，包装成一个**Authentication**类的对象
2. 将Authentication对象传给 “验证管理器” **ProviderManager**进行验证
3. ProviderManager在**一条链上**依次调用**AuthenticationProvider**进行验证
4. 验证成功则返回一个**封装了权限信息**的Authentication对象
5. 将此对象放入安全上下文SecurityContext中
6. 需要时，可以将Authentication对象从SecurityContextHolder上下文中取出

涉及的重要类

```
Authentication                                  # 验证信息类
    - Principal                                 # 身份信息类，通常时UserDetail
    - Credential                                # 凭据，通常是密码
    - Collection<? extends GrantedAuthority>    # 一组权限信息
AuthenticationManager                           # 验证管理类
    - ProviderManager                           # 是AuthenticationManager的实现类，
                                                # ProviderManager管理了一串AuthenticationProvider
    - AuthenticationProvider                    # 进行真正的验证功能
SecurityContextHolder                           # 用来保存Authentication对象
```

Authentication

```
public interface Authentication extends Principal, Serializable {

    // 代表一组已经分发的权限，即本次验证的角色（role）集合
    Collection<? extends GrantedAuthority> getAuthorities();

    // 代表验证凭据，即密码（在核心配置类的config方法中配置对应的表单字段）
    Object getCredentials();

    Object getDetails();

    // 代表身份信息，即用户名等（在核心配置类的config方法中配置对应的表单字段）
    Object getPrincipal();

    boolean isAuthenticated();

    void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException;
}

```

AuthenticationManager

> 他是验证管理类的总接口
>
> 而具体的验证管理需要ProviderManager类，他有一个List authenticationProviders属性
>
> 这个属性实际上就是一个个AuthenticationProvider实例构成的验证链，由这些AuthenticationProvider进行具体的验证工作
>
> 在ProviderManager管理的验证链上，任何一个AuthenticationProvider通过了验证，则验证成功

使用Spring Security的数据库需求

> 一般需要5张表，分别是
>
> menu　　菜单表，也可以设计为权限表permission
>
> role　　角色表
>
> role_menu　　角色菜单对应表
>
> user　　用户表
>
> user_role　　用户角色对应表

使用spring security建议从核心配置类开始

```
// 核心配置类需要继承WebSecurityConfigurerAdapter
```

spring security的核心配置类中核心就是其中的三个config方法

```
// 这三个方法全都来自WebSecurityConfigurerAdapter类
// 这个方法中的参数auth是AuthenticationManagerBuilder类型，它用来处理认证，也就是登录
// 可以通过AuthenticationManagerBuilder将AuthenticationProvider放入AuthenticationProvider链中
// 在进行认证的时候会依次调用链上的AuthenticationProvider
protected void configure(AuthenticationManagerBuilder auth) throws Exception {

// 这个方法可以配置哪些请求不进行拦截（也可以在下边的方法中配置）
public void configure(WebSecurity web) throws Exception {

// 这个是最核心的配置
// 可以配置自定义登录成功、失败，鉴权成功、失败等处理器
// 可以添加过滤器filter
protected void configure(HttpSecurity http) throws Exception {
```

认证配置configure(AuthenticationManagerBuilder auth)

```
@Override
protected void configure(AuthenticationManagerBuilder auth){
    //这里可启用我们自己的登陆验证逻辑
    auth.authenticationProvider(userAuthenticationProvider);
}
```

主要配置configure(HttpSecurity http)

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            // 不进行权限验证的请求或资源(从配置文件中读取)
            .antMatchers(JWTConfig.antMatchers.split(",")).permitAll()
            // 其他的需要登陆后才能访问  其他url都需要验证
            .anyRequest().authenticated()
            .and()
            // 配置未登录自定义处理类

.httpBasic().authenticationEntryPoint(userAuthenticationEntryPointHandler)
            .and()
            // 配置登录地址
            .formLogin()
            //配置security表单登录页面地址 默认是login，这里因为是前后端分离的项目，可以不配置，直接通过response返回消息即可
            .loginPage("/login")
            //配置security提交form表单请求的接口地址 默认是/login/userLogin
            .loginProcessingUrl("/user/login")
            //设置security提交的用户名属性值是那个 默认是username
            .usernameParameter("idNumber")
            //设置security提交的密码属性名是那个 默认是password
            .passwordParameter("password")
            // 配置登录成功自定义处理类
            .successHandler(userLoginSuccessHandler)
            // 配置登录失败自定义处理类
            .failureHandler(userLoginFailureHandler)
            .and()
            // 配置登出地址
            .logout()
            .logoutUrl("/user/logout")
            // 配置用户登出自定义处理类
            .logoutSuccessHandler(userLogoutSuccessHandler)
            .and()
            // 配置没有权限自定义处理类

.exceptionHandling().accessDeniedHandler(userAuthAccessDeniedHandler)
            .and()
            // 开启跨域
            .cors()
            .and()
            // 取消跨站请求伪造防护
            .csrf().disable();
    // 基于Token不需要session
```

```
 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    // 禁用缓存
    http.headers().cacheControl();
    // 添加JWT过滤器
    http.addFilter(new JWTAuthenticationTokenFilter(authenticationManager()));
}
```

配置过滤器filter

> 过滤器的主要作用就是对token进行拦截，如果token有效则封装一个携带了权限信息的Authentication对象，即不需要再次登录
>
> filter需要再核心配置类的configure(HttpSecurity http)方法中进行配置

```
@Slf4j
public class JWTAuthenticationTokenFilter extends BasicAuthenticationFilter {

    public JWTAuthenticationTokenFilter(AuthenticationManager authenticationManager) {
        super(authenticationManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        // 获取请求头中JWT的Token
        String tokenHeader = request.getHeader(JWTConfig.tokenHeader);
        if (null!=tokenHeader && tokenHeader.startsWith(JWTConfig.tokenPrefix)) {
            try {
                // 截取JWT前缀
                String token = tokenHeader.replace(JWTConfig.tokenPrefix, "");
                String id = JwtUtil.getMemberIdByJwtToken(token);

                SecurityUser securityUser = new SecurityUser();
                SysUser sysUser = new SysUser();
                sysUser.setId(id);
                securityUser.setSysUser(sysUser);
                List<GrantedAuthority> authorities = new ArrayList<>();

                /*String authority = claims.get("authorities").toString();
                if(!StringUtils.isEmpty(authority)){
                    List<Map<String,String>> authorityMap = JSONObject.parseObject(authority, List.class);
                    for(Map<String,String> role : authorityMap){
                        if(!StringUtils.isEmpty(role)) {
                            authorities.add(new SimpleGrantedAuthority(role.get("authority")));
                        }
```

```
                    }
                }*/

                UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(securityUser, id, authorities);

 SecurityContextHolder.getContext().setAuthentication(authentication);

            } catch (ExpiredJwtException e){
                log.info("Token过期");
            } catch (Exception e) {
                log.info("Token无效");
            }
        }
        filterChain.doFilter(request, response);
        return;
    }
}
```

自定义自己的处理类

> 这里主要的做法是通过HttpServletResponse直接将json写出

以下的处理器需要在核心配置类的configure(HttpSecurity http)方法中进行配置

登录成功处理类

```
@Slf4j
@Component
public class UserLoginSuccessHandler implements AuthenticationSuccessHandler
{

    /**
     * 登录成功返回结果
     * @param request
     * @param response
     * @param authentication
     * @author 20170
     * @date 2021/2/3 21:23
     * @return void
     */
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
HttpServletResponse response, Authentication authentication) throws
IOException, ServletException {
        // 组装JWT
        SecurityUser securityUser = (SecurityUser)
authentication.getPrincipal();
        String token =
JwtUtil.getJwtToken(securityUser.getSysUser().getId(),
securityUser.getUsername());
        token = JWTConfig.tokenPrefix + token;
```

```
        ResponseUtil.out(response, R.ok().message("登录成功").data("token", token));
    }
}
```

登录失败处理类

```
@Slf4j
@Component
public class UserLoginFailureHandler implements AuthenticationFailureHandler {

    /**
     * 登录失败返回结果
     * @param request
     * @param response
     * @param exception
     * @author 20170
     * @date 2021/2/3 21:21
     * @return void
     */
    @Override
    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response, AuthenticationException exception) throws IOException {
        // 这些对于操作的处理类可以根据不同异常进行不同处理
        if (exception instanceof UsernameNotFoundException){
            log.info("【登录失败】"+exception.getMessage());
            ResponseUtil.out(response, R.error().message("用户名不存在"));
        }
        if (exception instanceof BadCredentialsException){
            log.info("【登录失败】"+exception.getMessage());
            ResponseUtil.out(response, R.error().message("用户名密码不正确"));
        }
        ResponseUtil.out(response, R.error().message("登录失败"));
    }
}
```

用户未登录处理类

```
@Component
public class UserAuthenticationEntryPointHandler implements AuthenticationEntryPoint {
    /**
     * 用户未登录返回结果
     * @param request
     * @param response
     * @param exception
     * @author 20170
     * @date 2021/2/3 21:20
     * @return void
     */
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException exception){
        ResponseUtil.out(response, R.error().message("未登录"));
```

```
    }
}
```

暂无权限处理类

> 这个类需要配合权限使用

```
@Component
public class UserAuthAccessDeniedHandler implements AccessDeniedHandler {
    /**
     * 暂无权限返回结果
     * @param request
     * @param response
     * @param exception
     * @author 20170
     * @date 2021/2/3 21:16
     * @return void
     */
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response, AccessDeniedException exception){
        ResponseUtil.out(response, R.error().message("未授权"));
    }
}
```

登出处理类

```
@Component
public class UserLogoutSuccessHandler implements LogoutSuccessHandler {

    /**
     * 用户登出返回结果
     * @param request
     * @param response
     * @param authentication
     * @author 20170
     * @date 2021/2/5 9:05
     * @return void
     */
    @Override
    public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication){

        SecurityContextHolder.clearContext();
        ResponseUtil.out(response, R.ok().message("登出成功"));

    }
}
```

致敬大佬的文章

https://zhuanlan.zhihu.com/p/47224331

一、入门

1.前言

对于绝大部分这篇文章的读者来说，并不关心Spring Security里面那些设计绝妙的n个类，对于那些一上来就给你展示n个类的文章更是深恶痛绝。老板让我明天就交活搞定网站登录验证功能，你还让我看设计模式入门？有鉴于此，我写了这个系列文章，希望没有任何安全、验证、授权管理基础的Coder可以快速入门，在掌握有限知识的前提下，完成网站登录验证等系列功能。

阅读本文需要的基础知识：

- 熟练掌握Java
- 掌握了Spring Boot基础知识
- 一点点前端知识包括html/css/javascript
- 了解一点后端框架thymeleaf

2.需求以及示例代码

即使没有任何网站登录验证功能编码基础的人，也能想出下面的这个功能需求：

1. 网站分为首页、登录页、用户页面、管理员页面和报错页面；
2. 使用用户名加密码登录，登录错误要报错；
3. 不同的用户拥有不同的权限，不同的权限可以访问不同的网页；
4. 首页和登录页不需要任何权限；
5. 用户页面需要USER权限；
6. 管理员页面需要ADMIN权限；
7. 如果用户没有登录，则访问需要权限的页面时自动跳转登录页面。

好，我现在告诉你，实现以上功能，除了前端页面和SpringBoot代码外，和Security相关的代码只需要一个类SecurityConfiguration（继承了WebSecurityConfigurerAdapter），类中只有不到20行代码。

代码在我的github中：

apkkids/spring_security_examgithub.com

该项目一共包含四个可独立运行的子项目：本文对应的是**simple_security子项目。**从Github下载代码后，进入simple_security项目直接运行即可观察效果。

能力强的同学可以直接去看代码了，如果有些地方看不懂就继续看下面的文章。

3.代码解释

首先把需求对应的实现一一解释如下：

1. 网站分为首页（index.html）、登录页(login.html)、用户页面(user/user.html)、管理员页面(admin/admin.html)和报错页面(error/403.html)；
2. 使用用户名加密码登录，登录错误要报错(SpringSecurity 自带，不用写代码)；
3. 不同的用户拥有不同的权限，不同的权限可以访问不同的网页（SecurityConfiguration中定义）；
4. 首页和登录页不需要任何权限（SecurityConfiguration中定义）；
5. 用户页面需要USER权限（SecurityConfiguration中定义）；
6. 管理员页面需要ADMIN权限（SecurityConfiguration中定义）；
7. 如果用户没有登录，则访问需要权限的页面时自动跳转登录页面（SecurityConfiguration中定义）。

解释一下SecurityConfiguration类的第一个方法：

```
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        /**
         * 在内存中创建一个名为 "user" 的用户，密码为 "pwd"，拥有 "USER" 权限，密码使
用BCryptPasswordEncoder加密
         */
        auth.inMemoryAuthentication().passwordEncoder(new
BCryptPasswordEncoder())
                .withUser("user").password(new
BCryptPasswordEncoder().encode("pwd")).roles("USER");
        /**
         * 在内存中创建一个名为 "admin" 的用户，密码为 "pwd"，拥有 "USER"
和"ADMIN"权限
         */
        auth.inMemoryAuthentication().passwordEncoder(new
BCryptPasswordEncoder())
                .withUser("admin").password(new
BCryptPasswordEncoder().encode("pwd")).roles("USER","ADMIN");
    }
```

该方法在内存中建立了两个用于验证的User对象，每当用户从网页登录时，与内存中的对象进行比较，如果用户名密码都匹配，则验证通过，否则不通过。值得注意的是，从SpringSecurity5.0以后，这种方式的密码都要使用passwordEncoder进行加密，否则就会报错。

SecurityConfiguration类的第二个方法在代码的注释中已经清楚解释了。

4.Spring Security原理初探

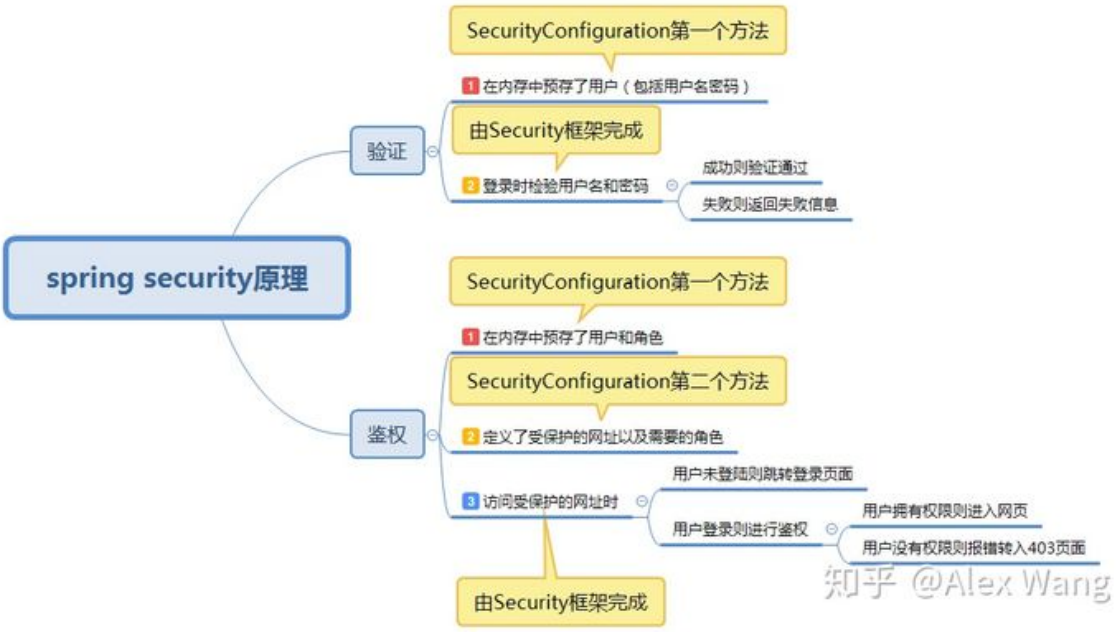
代码跑起来了，心情舒畅了，再来看看原理，念头就通达了。网络上很多关于Spring Security原理和框架的文章，我只能说他们水平太高，我看不懂，我只好自己写一版。

任何一个权限管理系统，主要都分为两个功能：验证和鉴权。验证就是确认用户的身份，一般采用用户名和密码的形式；鉴权就是确认用户拥有的身份（角色、权限）能否访问受保护的资源。这里面其实就涉及到了三个东西，用户、角色、受保护的资源。在上面的例子中，它们三者如下所示：

用户：user（角色为USER）、admin（角色为USER、ADMIN）

受保护的资源：user/**（需要USER角色）**，admin/（需要ADMIN角色）

那么Spring Security的功能原理如下图所示：



5.小结

看到这里，你应该有点概念了。但是，这玩意怎么能用啊？我还有一肚子问题呢：

- 用户名、密码和角色不是应该存在数据库里面吗？
- 验证成功后，验证信息保存在哪里？在程序中如何获取这些信息？
- 怎么管理cookie？验证信息多久过期？
- 真正的系统中角色和用户应该能够动态调整，鉴权规则也很复杂，这怎么定制？
- 验证、鉴权的流程是怎么实现的？说好的filter链呢？说好的依赖注入、AOP呢？

这些东西就留待下一篇再解释了，我会给出一个稍微复杂一点的示例来进一步解释Spring Security的原理。

好了，可以点赞了，或者到github上star我。

二、验证

1.前言

这是本系列的第二篇文章，上一篇文章主要讲了一个入门例子

Spring Security主要包括两大功能：**验证和鉴权。**验证就是确认用户的身份，一般采用用户名和密码的形式；鉴权就是确认用户拥有的身份（角色、权限）能否访问受保护的资源。本文主要讲述如何在Spring Security中进行验证功能的开发，阅读本文需要的基础知识：

- 熟练掌握Java
- 掌握Spring Boot基础知识
- 一点点前端知识包括html/css/javascript
- 了解一点后端框架thymeleaf

2.需求以及示例代码

本文是根据第一篇文章进行的改进，在第一篇文章中，实现了以下需求：

1. 网站分为首页、登录页、用户页面、管理员页面和报错页面；
2. 使用用户名加密码登录，登录错误要报错；
3. 不同的用户拥有不同的权限，不同的权限可以访问不同的网页；
4. 首页和登录页不需要任何权限；
5. 用户页面需要USER权限；
6. 管理员页面需要ADMIN权限；
7. 如果用户没有登录，则访问需要权限的页面时自动跳转登录页面。

为了让读者掌握具体的验证原理、流程和代码，本文加入了以下三个功能：

1. 修改用户名密码的参数名称
2. 通过自定义一个AuthenticationProvider在系统中加入一个后门
3. 将验证身份信息展示到前端

代码在我的github中：

apkkids/spring_security_examgithub.com

该项目一共包含四个可独立运行的子项目：本文对应的是**normal_security子项目。**从Github下载代码后，进入**normal_security**项目直接运行即可观察效果。

能力强的同学可以直接去看代码了，但还是建议先看下面的原理解释，我将在文章的最后给出代码解释和执行界面展示。

3.原理图

由于Spring Security本身关于验证的架构较为复杂，设计的原理、概念、流程和实现代码很多，因此先画了一个思维导图帮助理解，如下图所示：

图1 Spring Security原理图

各部分内容将在后面一一解释。

4.参与验证的要素

Spring Security不仅仅支持网页登录这一种验证模式，它还支持多种其他验证模式。但是其参与验证的要素基本上还是用户、密码、角色、受保护的资源这四种。

用户名称一般在前端由访问者填入，而系统用户在后端一般存储在内存中或数据库中。

密码一般在前端由访问者填入，用于验证用户身份，在Security 5.0后密码必须使用PasswordEncoder加密，一般来说使用BCryptPasswordEncoder即可。

角色，又可以被看做权限，在Spring Security中，有时用Role表示，有时用Authority表示。前端一般看不到系统的角色。后端角色由管理者赋予给用户（可以事先赋予，也可以动态赋予），角色信息一般存储在内存或数据库中。

受保护的资源，一般来说就是指网址，有时也可以将某些函数方法定义为资源，但本文不涉及这类情况。

参与验证的要素（用户名、密码）在前端由表单提交，由网络传入后端后，会形成一个Authentication类的实例。（尽管我很讨厌直接介绍各种Class和Interface，但是本文不可避免的要涉及到比较多的类，本文会尽量只介绍最重要的）该实例在进行验证前，携带了用户名、密码等信息；在验证成功后，则携带了身份信息、角色等信息。Authentication接口代码节选如下：

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

其中getCredentials()返回一个Object credentials，它代表验证凭据，即密码；getPrincipal()返回一个Object principal，它代表身份信息，即用户名等；getAuthorities()返回一个Collection<? extends GrantedAuthority>，它代表一组已经分发的权限，即本次验证的角色（本文中权限和角色可以通用）集合。

有了Authentication实例，则验证流程主要围绕这个实例来完成。它会依次穿过整个验证链，并存储在SecurityContextHolder中。也可以像本文中的代码一样，在验证途中伪造一个Authentication实例，骗过验证流程，获得所有权限。

5.验证的规则

验证规则定义了以下几个东西：

1. 受保护的资源即网址，它们一般按访问所需权限分为几类
2. 哪一类资源可以由哪些角色访问
3. 规则定义在WebSecurityConfigurerAdapter的子类中

具体规则的定义方法可以在代码中观察。

6.验证流程

前文已经介绍了Authentication类，它代表了验证信息。

再介绍一个类AuthenticationManager，它是验证管理类的总接口；而具体的验证管理需要ProviderManager类，它具有一个List providers属性，这实际上是一个AuthenticationProvider实例构成的验证链。链上都是各种AuthenticationProvider实例，这些实例进行具体的验证工作，它们之间的关系如下图（图来自互联网）所示：

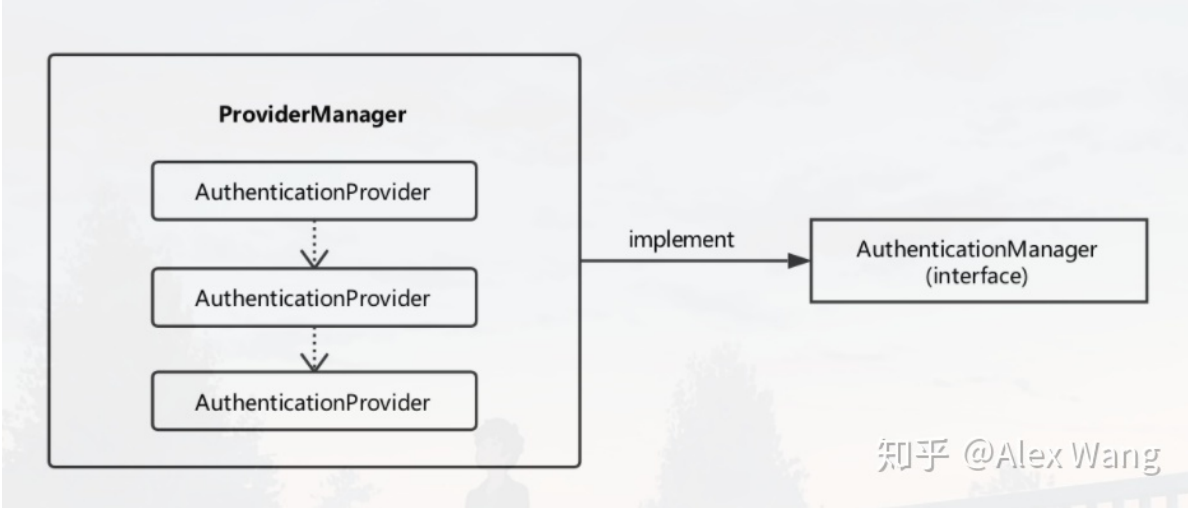


图2 验证管理类关系图

验证成功后，验证实例Authentication会被存入SecurityContextHolder中，而它则利用线程本地存储TLS功能。在验证成功且验证未过期的时间段内，验证会一直有效。而且，可以在需要的地方，从SecurityContextHolder中取出验证信息，并进行操作。例如将验证信息展示在前端。

具体的验证流程如下：

1. 后端从前端的表单得到用户密码，包装成一个Authentication类的对象；
2. 将Authentication对象传给"验证管理器"ProviderManager进行验证；
3. ProviderManager在一条链上依次调用AuthenticationProvider进行验证；
4. 验证成功则返回一个封装了权限信息的Authentication对象（即对象的Collection<? extends GrantedAuthority>属性被赋值）；
5. 将此对象放入安全上下文SecurityContext中；
6. 需要时，可以将Authentication对象从SecurityContextHolder上下文中取出。

注意，在ProviderManager管理的验证链上，任何一个AuthenticationProvider通过了验证，则验证成功。所以，要在系统中留一个后门，只需要在代码中添加一个AuthenticationProvider的子类BackdoorAuthenticationProvider，并在输入特定的用户名（alex）时，直接伪造一个验证成功的Authentication，即可通过验证，代码如下：

```
@Component
public class BackdoorAuthenticationProvider implements AuthenticationProvider {
    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String name = authentication.getName();
        String password = authentication.getCredentials().toString();

        //利用alex用户名登录，不管密码是什么都可以，伪装成admin用户
        if (name.equals("alex")) {
            Collection<GrantedAuthority> authorityCollection = new ArrayList<>();
            authorityCollection.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
            authorityCollection.add(new SimpleGrantedAuthority("ROLE_USER"));
            return new UsernamePasswordAuthenticationToken(
                    "admin", password, authorityCollection);
        } else {
            return null;
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(
                UsernamePasswordAuthenticationToken.class);
    }
}
```

然后在SecurityConfiguration类中，将BackdoorAuthenticationProvider的实例加入到验证链中即可，代码如下：

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Autowired
    BackdoorAuthenticationProvider backdoorAuthenticationProvider;
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        ...省略
         //将自定义验证类注册进去
         auth.authenticationProvider(backdoorAuthenticationProvider);
    }
  ...省略
}
```

7.代码解释

1)修改用户名密码的参数名称

在前端login.html：

```
        <!-- 1.自定义参数名：myusername和mypassword-->
        <div>
            用户名： <input type="text" name="myusername"/>
        </div>
        <div>
            密码： <input type="password" name="mypassword"/>
        </div>
```

在后端SecurityConfiguration.java：

```
  @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .antMatchers("/","/index","/error").permitAll()
                .antMatchers("/user/**").hasRole("USER")
                .antMatchers("/admin/**").hasRole("ADMIN")
                .and()
                .formLogin().loginPage("/login").defaultSuccessUrl("/user")
                //1.自定义参数名称，与login.html中的参数对应

.usernameParameter("myusername").passwordParameter("mypassword")
                .and()
                .logout().logoutUrl("/logout").logoutSuccessUrl("/login");
    }
```

2)通过自定义一个AuthenticationProvider加入一个后门

上一节中已经详细介绍了这个功能的实现。留下后门后，使用用户名alex，配合任何密码，都可以成功登陆并获得管理员权限，而且登陆后的页面上显示的也是admin用户。

3)将验证身份信息展示到前端

前面已经提到，验证成功后，验证信息存入SecurityContextHolder中。因此可以在需要的地方，将其提取出来，然后在前端展示出来。

后端代码：

```java
@Controller
public class UserController {

    @RequestMapping("/user")
    public String user(@AuthenticationPrincipal Principal principal, Model model) {
        model.addAttribute("username", principal.getName());

        //从SecurityContextHolder中得到Authentication对象，进而获取权限列表，传到前端
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        Collection<GrantedAuthority> authorityCollection = (Collection<GrantedAuthority>) auth.getAuthorities();
        model.addAttribute("authorities", authorityCollection.toString());
        return "user/user";
    }

    @RequestMapping("/admin")
    public String admin(@AuthenticationPrincipal Principal principal, Model model) {
        model.addAttribute("username", principal.getName());

        //从SecurityContextHolder中得到Authentication对象，进而获取权限列表，传到前端
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        Collection<GrantedAuthority> authorityCollection = (Collection<GrantedAuthority>) auth.getAuthorities();
        model.addAttribute("authorities", authorityCollection.toString());
        return "admin/admin";
    }
}
```

前端代码：

```html
<p>你的当前用户名是：</p>
<p th:text="${username}" style="margin-top: 25px; color: crimson">wxb</p>
<p>你的权限是：</p>
<p th:text="${authorities}" style="margin-top: 25px; color: crimson">authorities</p>
```

8.小结

关于验证的内容实在是比较多，因此文本中依然没有介绍如何利用数据库中存储的信息进行验证，那将会在下一篇文章中介绍。

三、使用数据库验证

1.前言

这是本系列的第三篇文章，前两篇主要讲了使用Spring Security进行验证的原理、流程和一个比较简单的例子。

这一篇的内容主要是介绍如何利用数据库中存储的信息进行验证。Spring Security内置了一些类，利用这些类可以很方便的将数据库中存储的用户、密码、角色信息导入系统中进行验证。由于涉及到数据库了，因此阅读本文需要的基础知识比前两章要多一些：

- 熟练掌握Java
- 掌握Spring Boot基础知识
- 了解MyBatis的基本使用方法
- 了解MySQL的一些基本使用方法
- 一点点前端知识包括html/css/javascript
- 了解一点后端框架thymeleaf

2.需求以及示例代码

第一篇文章中，实现了以下需求：

1. 网站分为首页、登录页、用户页面、管理员页面和报错页面；
2. 使用用户名加密码登录，登录错误要报错；
3. 不同的用户拥有不同的权限，不同的权限可以访问不同的网页；
4. 首页和登录页不需要任何权限；
5. 用户页面需要USER权限；
6. 管理员页面需要ADMIN权限；
7. 如果用户没有登录，则访问需要权限的页面时自动跳转登录页面。

第二篇中加入了以下三个功能：

1. 修改用户名密码的参数名称
2. 通过自定义一个AuthenticationProvider在系统中加入一个后门
3. 将验证身份信息展示到前端

本文将加入以下功能：

1. 从数据库中读取用户名、密码，与前端输入的信息进行对比验证；
2. 验证通过后，登陆用户会得到数据库中存储的角色信息。

代码在我的github中（本项目为security_withdb项目）：

apkkids/spring_security_examgithub.com

3.原理介绍

在第二章中介绍了验证的流程，因此可知，要加入想自定义的验证功能，就是向ProviderManager中加入一个自定义的AuthenticationProvider实例。为了加入使用数据库进行验证的DaoAuthenticationProvider类（这个类在我们的代码中是透明的）实例，可以使用AuthenticationManagerBuilder类的userDetailsService(UserDetailsService)方法。代码如下：

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        ...省略
        //加入数据库验证类，下面的语句实际上在验证链中加入了一个DaoAuthenticationProvider
        auth.userDetailsService(myUserDetailsService).passwordEncoder(new BCryptPasswordEncoder());
    }
```

而我们需要掌握的，就是由Security框架提供的两个接口UserDetails和UserDetailsService。其中UserDetails接口中定义了用于验证的“用户详细信息”所需的方法。而UserDetailsService接口仅定义了一个方法loadUserByUsername(String username)。这个方法由接口的实现类来具体实现，它的作用就是通过用户名username从数据库中查询，并将结果赋值给一个UserDetails的实现类实例。验证流程如下：

1. 由于在上面的configure方法中调用了userDetailsService(myUserDetailsService)方法，因此在ProviderManager的验证链中加入了一个DaoAuthenticationProvider类的实例；
2. 验证流程进行到DaoAuthenticationProvider时，它调用用户自定义的myUserDetailsService服务的loadUserByUsername方法，这个方法会从数据库中查询用户名是否存在；
3. 若存在，则从数据库中返回的信息会组成一个UserDetails接口的实现类的实例，并将此实例返回给DaoAuthenticationProvider进行密码比对，比对成功则通过验证。

仍然要注意，在ProviderManager管理的验证链上，任何一个AuthenticationProvider通过了验证，则验证成功。第二章加入的验证方法依然存在，所以系统user、admin、alex等用户依然能通过验证。

4.代码解释

1.要实现数据库验证，第一步是设计mysql数据库的库表结构，为了清晰起见，本文只使用了一个表，相关sql语句在项目的user.sql文件中：

```
CREATE DATABASE `mysecurity` DEFAULT CHARACTER SET utf8;

USE `mysecurity`;
SET FOREIGN_KEY_CHECKS=0;
-- ----------------------------
--  Table structure for `user`
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL COMMENT '姓名',
  `address` varchar(64) DEFAULT NULL COMMENT '联系地址',
  `username` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL COMMENT '账号',
  `password` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL COMMENT '密码',
  `roles` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL COMMENT '身份',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
-- ----------------------------
--  Records of `user`
-- ----------------------------
BEGIN;
INSERT INTO `user` VALUES ('1', 'Adam', 'beijing', 'adam','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER');
INSERT INTO `user` VALUES ('2', 'SuperMan', 'shanghang', 'super','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER,ROLE_ADMIN');
COMMIT;
```

运行此sql文件，就会建立一个user表，并插入两条记录，注意记录中的密码字段已经用BCryptPasswordEncoder加过密了，就是pwd加密后的字符串。

2.接下来就是实现UserDetails接口，MyUserBean类中唯一值得注意的就是getAuthorities方法的实现，它将数据库中的roles字段取出来分解为多个SimpleGrantedAuthority对象加入List中。

3.接下来用MyBatis实现MyUserBean的mapper接口，接口中仅定义了selectByUsername方法。

4.用MyUserDetailsService实现UserDetailsService接口，使用MyUserMapper来进行数据查询，并实现UserDetailsService的loadUserByUsername方法即可。

5.在SecurityConfiguration类中调用auth.userDetailsService(myUserDetailsService)方法，在验证链中加入一个DaoAuthenticationProvider。

如此就实现了使用数据库进行Spring Security的验证，你可以试试在数据库中加入新的记录，并使用新加入的用户登录。

5.小结

使用数据库进行验证其实只需要掌握两个接口即可，即UserDetailsService和UserDetails。除此外还要设计好自己的数据库表格。本文中为了简单，把用户名、密码和角色存储在一张表中，而实际上应该将用户名-密码和角色分开存储，便于实现动态的权限管理。这部分内容我们将会在下一篇文章中详细介绍。

另，Spring Security实际上提供了一套默认的数据库表格和具体的实现类，如果觉得自己的系统在验证功能上没有特殊性，也可以直接使用它的库表结构和实现类。

到此为止，验证功能就讲完了，下一章开始介绍鉴权，那也将是非常有意思的内容。

四、鉴权

1.前言

这是本系列的第四篇文章，前三篇主要讲了使用Spring Security进行验证的原理、流程和例子

本篇的内容主要介绍Spring Security两大功能之二：鉴权。意思是利用数据库中存储的用户角色信息、资源角色信息，对用户访问受限资源的过程进行权限的判断。阅读本文需要的基础知识如下：

- 熟练掌握Java
- 掌握Spring Boot基础知识
- 了解MyBatis的基本使用方法
- 了解MySQL的一些基本使用方法
- 一点点前端知识包括html/css/javascript
- 了解一点后端框架thymeleaf

2.需求以及示例代码

前三章的需求不赘述了，直接把本文完成的需求列举如下：

1. 网站分为首页、用户页面、部门1页面、部门2页面、管理员页面和登录页面；
2. 使用用户名加密码登录，登录错误要报错；
3. 根据前三章的功能，用户来自三个方面：1）内存用户user、admin；2）使用后门filter的黑客alex；3）以及数据库中的用户5个（详见数据库数据）
4. 不同的用户拥有不同的权限，不同的权限可以访问不同的网页；
5. 登录页不需要任何权限；
6. 用户页面需要USER权限；管理员页面需要ADMIN权限；部门1页面需要USER1或者MANAGER权限；部门2页面需要USER2或者MANAGER权限；
7. 如果用户没有登录，则访问需要权限的页面时自动跳转登录页面；
8. 如果用户已经登录，访问无权页面时服务器会返回一个json，告诉用户访问了没有权限的页面。

代码在我的github中（本项目为authorization_withdb项目）：

https://github.com/apkkids/spring_security_examgithub.com

运行项目时，apkkids/spring_security_exam运行项目时，

1.找到spring_security_exam\authorization_withdb\src\main\resources\application.properties，修改mysql的配置；

2.然后在mysql中运行user_source.sql脚本，创建相应的数据库表。

3.最后运行springboot应用程序。

3.鉴权原理

前三章已经介绍过，Spring Securtiy的两大功能是验证和鉴权。验证是验明用户的身份，允许用户登录系统。这个验证过程由一系列AuthenticationProvider构成的链来完成。一旦验证成功，用户身份得到确认，它也同时拥有了一些角色，例如ROLE_USER或者ROLE_ADMIN。

而鉴权则是一系列判断用户是否有权限访问资源的过程。以本文为例，我们使用 数据库中的myauthorization.resource表来存储了资源以及所需的角色。例如/depart2/**资源，需要ROLE_ADMIN,ROLE_MANAGER,ROLE_DEPART2这三个角色中的任意一个。而如果登录的用户拥有其中某个资源，则可以顺利访问，否则将会抛出AccessDeniedException异常，进入异常处理程序。

下面按照鉴权处理流程来说一遍鉴权所需编写的代码：

1.当用户未登录时，访问任何需要权限的资源都会转向登录页面，尝试进行登录；

2.当用户登录成功时，他会获得一系列角色。

3.用户访问某资源/xxx时，FilterInvocationSecurityMetadataSource这个类的实现类（本文是MySecurityMetadataSource）会调用getAttributes方法来进行资源匹配。它会读取数据库resource表中的所有记录，对/xxx进行匹配。若匹配成功，则将/xxx对应所需的角色组成一个 Collection返回；匹配不成功则说明/xxx不需要什么额外的访问权限；

4.流程来到鉴权的决策类AccessDecisionManager的实现类（MyAccessDecisionManager）中，它的decide方法可以决定当前用户是否能够访问资源。decide方法的参数中可以获得当前用户的验证信息、第3步中获得的资源所需角色信息，对这些角色信息进行匹配即可决定鉴权是否通过。当然，你也可以加入自己独特的判断方法，例如只要用户具有ROLE_ADMIN角色就一律放行；

5.若鉴权成功则用户顺利访问页面，否则在decide方法中抛出AccessDeniedException异常，这个异常会被AccessDeniedHandler的实现类（MyAccessDeniedHandler）处理。它仅仅是生成了一个json对象，转换为字符串返回给客户端了。

4.代码解释

1.首先是关于数据库的库表设计问题，本文中依然使用了单表结构，角色这个字段在两个表中是以逗号分隔，然后在程序中分解开来的，如下所示：

```
-- ----------------------------
--  Table structure for `user`
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL COMMENT '姓名',
  `address` varchar(64) DEFAULT NULL COMMENT '联系地址',
  `username` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL COMMENT '账号',
  `password` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL COMMENT '密码',
  `roles` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL COMMENT '角色',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
-- ----------------------------
--  Records of `user`
-- ----------------------------
BEGIN;
INSERT INTO `user` VALUES ('1', 'Adam', 'beijing', 'adam','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER');
```

```
INSERT INTO `user` VALUES ('2', 'SuperMan', 'shanghang', 'super','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER,ROLE_ADMIN');
INSERT INTO `user` VALUES ('3', 'Manager', 'beijing', 'manager','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER,ROLE_MANAGER');
INSERT INTO `user` VALUES ('4', 'User1', 'shanghang', 'user1','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER,ROLE_DEPART1');
INSERT INTO `user` VALUES ('5', 'User2', 'shanghang', 'user2','$2a$10$9SIFu8l8asZUKxtwqrJM5ujhWarz/PMnTX44wXNsBHfpJMakWw3M6', 'ROLE_USER,ROLE_DEPART2');
COMMIT;

-- ----------------------------
--  Table structure for `resource`
-- ----------------------------
DROP TABLE IF EXISTS `resource`;
CREATE TABLE `resource` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `url` varchar(255) DEFAULT NULL COMMENT '资源',
  `roles` varchar(255) DEFAULT NULL COMMENT '所需角色',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
-- ----------------------------
--  Records of `resource`
-- ----------------------------
BEGIN;
INSERT INTO `resource` VALUES ('1', '/depart1/**', 'ROLE_ADMIN,ROLE_MANAGER,ROLE_DEPART1');
INSERT INTO `resource` VALUES ('2', '/depart2/**', 'ROLE_ADMIN,ROLE_MANAGER,ROLE_DEPART2');
INSERT INTO `resource` VALUES ('3', '/user/**', 'ROLE_ADMIN,ROLE_USER');
INSERT INTO `resource` VALUES ('4', '/admin/**', 'ROLE_ADMIN');
COMMIT;
```

而在实际项目中，应该使用多表关联更加灵活。

2.关于FilterInvocationSecurityMetadataSource类（安全元数据源类）如何设置到WebSecurityConfigurerAdapter中的问题，综合网上目前的文章，使用JavaConfig只有一种方法，即使用withObjectPostProcessor方法，代码如下：

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Autowired
    BackdoorAuthenticationProvider backdoorAuthenticationProvider;
    @Autowired
    MyUserDetailsService myUserDetailsService;
    @Autowired
    MyAccessDecisionManager myAccessDecisionManager;
    @Autowired
    MySecurityMetadataSource mySecurityMetadataSource;
    @Autowired
    MyAccessDeniedHandler myAccessDeniedHandler;

...省略
    @Override
```

```
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .withObjectPostProcessor(new
ObjectPostProcessor<FilterSecurityInterceptor>() {
                    @Override
                    public <O extends FilterSecurityInterceptor> O
postProcess(O object) {

 object.setSecurityMetadataSource(mySecurityMetadataSource);

 object.setAccessDecisionManager(myAccessDecisionManager);
                        return object;
                    }
                })
   ...省略
```

3.鉴权决策类AccessDecisionManager设置到WebSecurityConfigurerAdapter中有两种方法，其一是如上的方法，其二是

```
        http

.authorizeRequests().accessDecisionManager(myAccessDecisionManager)
```

4.本文中MyAccessDeniedHandler类只是返回一个json字符串。如果不处理AccessDeniedException异常，则会显示403错误页面。此处返回的字符串可以和前端结合展示更加丰富的页面。

5.小结

在网上其他介绍鉴权的文章中，经常提到Spring Security内置了三个基于投票的AccessDecisionManager实现类，它们分别是AffirmativeBased、ConsensusBased和UnanimousBased。然后如何实现一个投票类。但是我觉得直接继承AccessDecisionManager，然后重写decide方法更加简便。

至此，本系列完结了，我可以专心去看前端知识了：）