

尚筹网

[09-后台管理系统-分配]

1 SpringSecurity 回顾

1.1 准备工作

准备 SpringMVC 的环境。发送请求访问资源时完全没有限制。

1.2 加入 SpringSecurity 环境

1.2.1 加入 SpringSecurity 依赖

```
<!-- SpringSecurity 对 Web 应用进行权限管理 -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>4.2.10.RELEASE</version>
</dependency>

<!-- SpringSecurity 配置 -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>4.2.10.RELEASE</version>
</dependency>

<!-- SpringSecurity 标签库 -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>4.2.10.RELEASE</version>
</dependency>
```

1.2.2 在 web.xml 中配置 DelegatingFilterProxy

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
```

```
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

注意：SpringSecurity 会根据 DelegatingFilterProxy 的 filter-name 到 IOC 容器中查找所需要的 bean。所以 filter-name 必须是 springSecurityFilterChain 名字。

1.2.3 创建基于注解的配置类

```
// 注意！这个类一定要放在自动扫描的包下，否则所有配置都不会生效！
// 将当前类标记为配置类
@Configuration

// 启用 Web 环境下权限控制功能
@EnableWebSecurity

public class WebAppSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder builder) throws Exception {
        // 与 SpringSecurity 环境下用户登录相关
    }

    @Override
    protected void configure(HttpSecurity security) throws Exception {
        // 与 SpringSecurity 环境下请求授权相关
    }

}
```

1.3 放行首页和静态资源

```
security
    .authorizeRequests()           // 对请求进行授权
    .antMatchers("/index.jsp")    // 针对 /index.jsp 路径进行授权
    .permitAll()                  // 可以无条件访问
    .antMatchers("/layui/**")     // 针对 /layui 目录下所有资源进行授权
    .permitAll()                  // 可以无条件访问
    .and()
    .authorizeRequests()           // 对请求进行授权
    .anyRequest()                  // 任意请求
    .authenticated()              // 需要登录以后才可以访问
```

设置授权信息时需要注意，范围小的放在前面、范围大的放在后面。不然的话，小范围的设置会被大范围设置覆盖。

效果：未登录请求访问需要登录的请求时会看到 403 页面。

1.4 未认证请求跳转到登录页

```
security
.....
.and()
.formLogin()                // 使用表单形式登录

// 关于 loginPage()方法的特殊说明
// 指定登录页的同时会影响到：“提交登录表单的地址”、“退出登录地址”、“登录失败地址”
// /index.jsp GET - the login form 去登录页面
// /index.jsp POST - process the credentials and if valid authenticate the user 提交登录表单
// /index.jsp?error GET - redirect here for failed authentication attempts 登录失败
// /index.jsp?logout GET - redirect here after successfully logging out 退出登录
.loginPage("/index.jsp")    // 指定登录页面（如果没有指定会访问 SpringSecurity 自带的登录页）

// loginProcessingUrl()方法指定了登录地址，就会覆盖 loginPage()方法中设置的默认值/index.jsp POST
.loginProcessingUrl("/do/login.html") // 指定提交登录表单的地址
.usernameParameter("loginAcct")      // 定制登录账号的请求参数名
.passwordParameter("userPswd")       // 定制登录密码的请求参数名
.defaultSuccessUrl("/main.html")     // 登录成功后前往的地址
```

formLogin()开启表单登录功能

loginPage()指定登录页地址

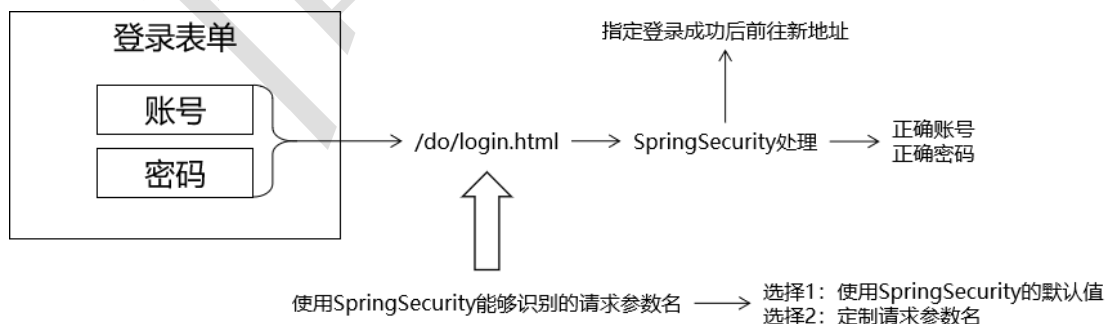
loginProcessingUrl()处理登录请求的 URL 地址

usernameParameter()设置用户名的请求参数名

passwordParameter()设置密码的请求参数名

defaultSuccessUrl()登录成功后前往的地址

1.5 实现完整的登录流程



1.5.1 设置表单

给 index.jsp 设置表单

```
<p>${SPRING_SECURITY_LAST_EXCEPTION.message}</p>
<form action="${pageContext.request.contextPath }/do/login.html" method="post">
```

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

注意：要取消页面的“假”提交。不用管 layui 的语法。

```
/* form.on('submit(LAY-user-login-submit)', function(obj) {
    obj.elem.classList.add("layui-btn-disabled");//样式上的禁用效果
    obj.elem.disabled = true;//真正的禁用效果
    layer.msg("登陆成功，即将跳转");
    setTimeout(function(){
        location.href="main.html";
    }, 2000);
}); */
```

账号、密码的请求参数名

SpringSecurity 默认账号的请求参数名：username
SpringSecurity 默认密码的请求参数名：password

要么修改页面上的表单项的 name 属性值，要么修改配置。如果修改配置可以调用 usernameParameter() 和 passwordParameter() 方法。

1.5.2 设置正确的账号、密码

重写 configure(AuthenticationManagerBuilder builder) 方法

```
builder
    .inMemoryAuthentication() // 在内存中完成账号、密码的检查
    .withUser("tom")          // 指定账号
    .password("123123")        // 指定密码
    .roles("ADMIN","学徒")     // 指定当前用户的角色
    .and()
    .withUser("jerry")         // 指定账号
    .password("123123")        // 指定密码
    .authorities("UPDATE","内门弟子") // 指定当前用户的权限
    ;
```

如果没有提供角色或权限，那么会抛出异常，异常消息是 Cannot pass a null GrantedAuthority collection。

SpringSecurity 的用意是：仅仅账号、密码正确还不够，还必须具备访问特定资源的角色或权限才能够完成认证。

1.6 退出登录

```
security
    .and()
    .csrf()
```

```
.disable()           // 禁用 CSRF 功能
.logout()           // 开启退出功能
.logoutUrl("/do/logout.html") // 指定处理退出请求的 URL 地址
.logoutSuccessUrl("/index.jsp") // 退出成功后前往的地址
```

如果没有禁用 CSRF:

请求必须携带 CSRF 的 token 值。

如果已经禁用 CSRF:

没有限制。

1.7 基于角色或权限实现访问控制

```
security
.....

.antMatchers("/level1/**") // 针对/level1/**路径设置访问要求
.hasRole("学徒")           // 要求用户具备“学徒”角色才可以访问

.antMatchers("/level2/**") // 针对/level2/**路径设置访问要求
.hasAuthority("内门弟子")   // 要求用户具备“内门弟子”权限才可以访问
```

注意: SpringSecurity 会在底层用 “ROLE_” 区分角色和权限。角色信息会被附加 “ROLE_” 前缀。

```
private static String hasRole(String role) {
    Assert.notNull(role, "role cannot be null");
    if (role.startsWith("ROLE_")) {
        throw new IllegalArgumentException(
            "role should not start with 'ROLE_' since"
            + role + "");
    }
    return "hasRole('ROLE_" + role + "')";
}
```

效果: 访问被拒绝后返回 403 页面。

1.8 指定 403 页面

1.8.1 简易方案

```
security
.....

.exceptionHandling() // 指定异常处理器
.accessDeniedPage("/to/no/auth/page.html") // 访问被拒绝时前往的页面

/to/no/auth/page.html 地址需要能够访问到指定的页面。
```

1.8.2 定制方案

```
security
.....

.exceptionHandling()                // 指定异常处理器
.accessDeniedHandler(new AccessDeniedHandler() {

    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
        AccessDeniedException accessDeniedException) throws IOException, ServletException {
        request.setAttribute("message", "抱歉！您无法访问这个资源！☆☆☆");
        request.getRequestDispatcher("/WEB-INF/views/no_auth.jsp").forward(request, response);
    }
})
```

1.9 记住我-内存版（不重要）

调用 security.rememberMe()方法开启记住我功能。

表单中提供名为 remember-me 的请求参数。为了用户便于操作，通常会使用多选框。

```
<input type="checkbox" name="remember-me" lay-skin="primary" title="记住我"/>
```

1.10 记住我-数据库版（不重要）

1.10.1 加入依赖

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.12</version>
</dependency>

<!-- mysql 驱动 -->
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
```

```
<version>4.3.20.RELEASE</version>
</dependency>
```

1.10.2 创建数据库

```
CREATE DATABASE `security` CHARACTER SET utf8;
```

1.10.3 配置数据源

```
<!-- 配置数据源 -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="username" value="root"></property>
    <property name="password" value="root"></property>
    <property
        value="jdbc:mysql://localhost:3306/security?useSSL=false"></property>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    </bean>

<!-- jdbcTemplate-->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

1.10.4 在配置类中装配数据源

```
@Autowired
private DataSource dataSource;
```

1.10.5 创建数据库表

- 办法一：自己拿建表的 SQL 语句到数据库执行。
- 办法二：修改 JdbcTokenRepositoryImpl 源码，把 initDao() 方法改成 public 权限。

PS：修改框架的源码的操作

- 1、创建和原类完全相同的包。
- 2、创建和原类同名的类。
- 3、把原类的代码全部复制到我们自己创建的类中。
- 4、根据需要修改。

```
// 准备 JdbcTokenRepositoryImpl 对象
JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();
tokenRepository.setDataSource(dataSource);
```

```
// 创建数据库表
tokenRepository.setCreateTableOnStartup(true);
tokenRepository.initDao();
```

```
security.tokenRepository(tokenRepository)
```

1.11 查数据库完成认证

1.11.1 实现 UserDetailsService 接口

```
@Component
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    // 总目标：根据表单提交的用户名查询 User 对象，并装配角色、权限等信息
    @Override
    public UserDetails loadUserByUsername(

        // 表单提交的用户名
        String username

    ) throws UsernameNotFoundException {

        // 1.从数据库查询 Admin 对象
        String sql = "SELECT id,loginacct,userpswd,username,email FROM t_admin WHERE loginacct=?";

        List<Admin> list = jdbcTemplate.query(sql, new
        BeanPropertyRowMapper<>(Admin.class), username);

        Admin admin = list.get(0);

        // 2.给 Admin 设置角色权限信息
        List<GrantedAuthority> authorities = new ArrayList<>();

        authorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
        authorities.add(new SimpleGrantedAuthority("UPDATE"));

        // 3.把 admin 对象和 authorities 封装到 UserDetails 中
```



```
String userpswd = admin.getUserpswd();

return new User(username, userpswd, authorities);
}

}
```

1.11.2 把 UserDetailsService 装配到配置类中

```
@Autowired
private MyUserDetailsService userDetailsService;
```

1.11.3 使用 UserDetailsService 对象

```
// builder
//     .inMemoryAuthentication() // 在内存中完成账号、密码的检查
//     .withUser("tom")          // 指定账号
//     .password("123123")        // 指定密码
//     .roles("ADMIN","学徒")     // 指定当前用户的角色
//     .and()
//     .withUser("jerry")         // 指定账号
//     .password("123123")        // 指定密码
//     .authorities("UPDATE","内门弟子") // 指定当前用户的权限
//
//
// 装配 userDetailsService 对象
builder
    .userDetailsService(userDetailsService);
```

1.12 密码加密

1.12.1 认识 SpringSecurity 提供的加密接口

```
public interface PasswordEncoder {

    /**
     * 加密
     * Encode the raw password. Generally, a good encoding algorithm applies a SHA-1 or
     * greater hash combined with an 8-byte or greater randomly generated salt.
     */
    String encode(CharSequence rawPassword);
}
```

```
/**
 * 校验：检查一个明文密码是否和一个密文密码一致
 * Verify the encoded password obtained from storage matches the submitted raw
 * password after it too is encoded. Returns true if the passwords match, false if
 * they do not. The stored password itself is never decoded.
 *
 * @param rawPassword the raw password to encode and match
 * @param encodedPassword the encoded password from storage to compare with
 * @return true if the raw password, after encoding, matches the encoded password from
 * storage
 */
boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

1.12.2 创建 PasswordEncoder 实现类

```
@Component
public class MyPasswordEncoder implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {

        return privateEncode(rawPassword);

    }

    @Override
    public boolean matches(CharSequence rawPassword, String encodedPassword) {

        // 1.对明文密码进行加密
        String formPassword = privateEncode(rawPassword);

        // 2.声明数据库密码
        String databasePassword = encodedPassword;

        // 3.比较
        return Objects.equals(formPassword, databasePassword);

    }

    private String privateEncode(CharSequence rawPassword) {
        try {
```

```
// 1.创建 MessageDigest 对象
String algorithm = "MD5";
MessageDigest messageDigest = MessageDigest.getInstance(algorithm);

// 2.获取 rawPassword 的字节数组
byte[] input = ((String)rawPassword).getBytes();

// 3.加密
byte[] output = messageDigest.digest(input);

// 4.转换为 16 进制数对应的字符
String encoded = new BigInteger(1, output).toString(16);

return encoded;

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
    return null;
}
}
```

1.12.3 在 SpringSecurity 的配置类中装配 MyPasswordEncoder

```
@Autowired
private MyPasswordEncoder passwordEncoder;
```

1.12.4 使用 MyPasswordEncoder 对象

```
builder.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder);
```

1.12.5 潜在问题

固定的明文对应固定的密文，虽然很难从密文通过算法破解反推回明文。但是可以借助已有的明文和密文的对应关系猜解出来。

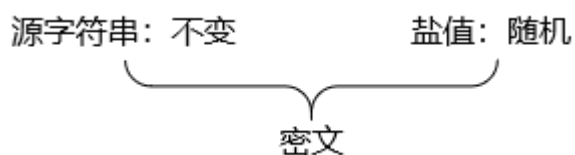
123123→4297F44B13955235245B2497399D7A93

1.13 带盐值的加密

1.13.1 概念

借用生活中烹饪时加盐值不同，菜肴的味道不同这个现象，在加密时每次使用

一个随机生成的盐值，让加密结果不固定。



1.13.2 用法

创建 BCryptPasswordEncoder 对象，传给 passwordEncoder()方法。

```
// 每次调用这个方法时会检查 IOC 容器中是否有了对应的 bean，如果有就不会真正执行这个函数，因为 bean 默认是单例的
// 可以使用@Scope(value="")注解控制是否单例
@Bean
public BCryptPasswordEncoder getBCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

```
builder
    .userDetailsService(userDetailsService).passwordEncoder(getBCryptPasswordEncoder());
```

2 项目中加入 SpringSecurity

2.1 加入 SpringSecurity 环境

2.1.1 依赖

在原有的 SSM 整合环境基础上加入 SpringSecurity 的依赖。

2.1.2 在 web.xml 中配置 DelegatingFilterProxy

参照前面的笔记。

2.1.3 创建基于注解的配置类

参照前面的笔记。

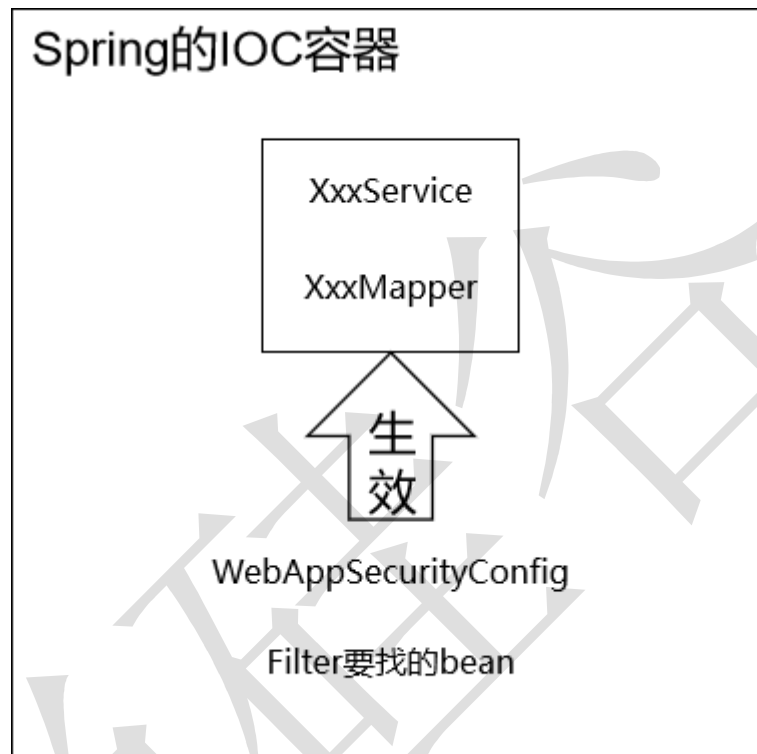
```
// 表示当前类是一个配置类
@Configuration

// 启用 Web 环境下权限控制功能
@EnableWebSecurity
public class WebAppSecurityConfig extends WebSecurityConfigurerAdapter {
```

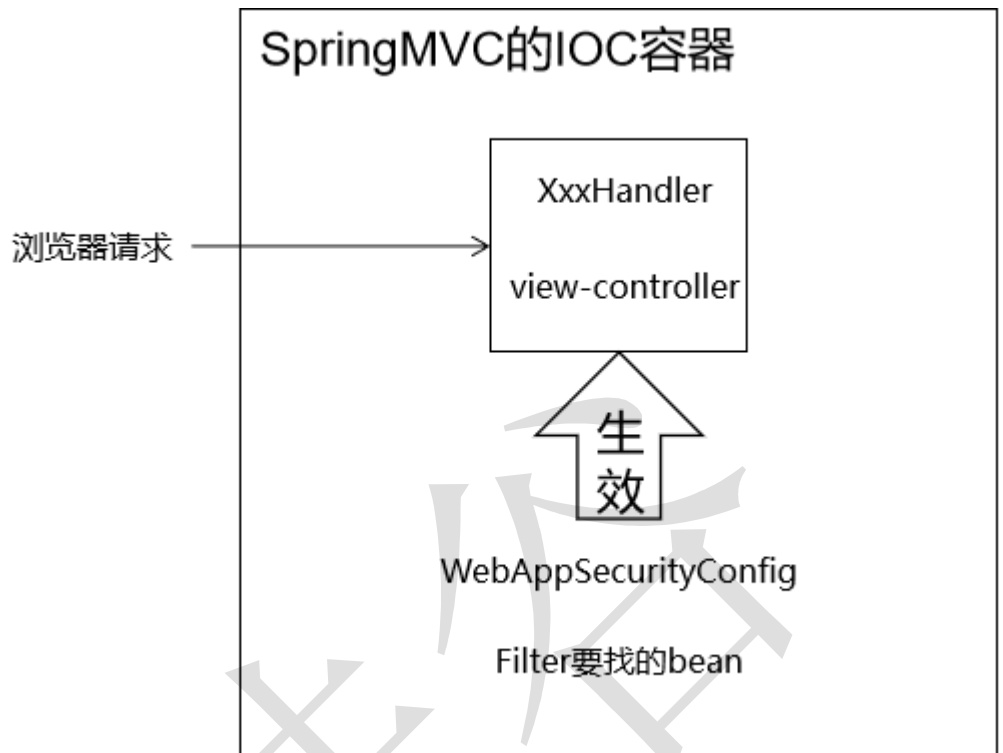
```
}
```

2.1.4 谁来把 WebAppSecurityConfig 扫描到 IOC 里？

如果是 Spring 的 IOC 容器扫描：



如果是 SpringMVC 的 IOC 容器扫描：



结论：为了让 SpringSecurity 能够针对浏览器请求进行权限控制，需要让 SpringMVC 来扫描 WebAppSecurityConfig 类。

衍生问题：DelegatingFilterProxy 初始化时需要到 IOC 容器查找一个 bean，这个 bean 所在的 IOC 容器要看是谁扫描了 WebAppSecurityConfig。

如果是 Spring 扫描了 WebAppSecurityConfig，那么 Filter 需要的 bean 就在 Spring 的 IOC 容器。

如果是 SpringMVC 扫描了 WebAppSecurityConfig，那么 Filter 需要的 bean 就在 SpringMVC 的 IOC 容器。

2.2 提出找不到 bean 的问题

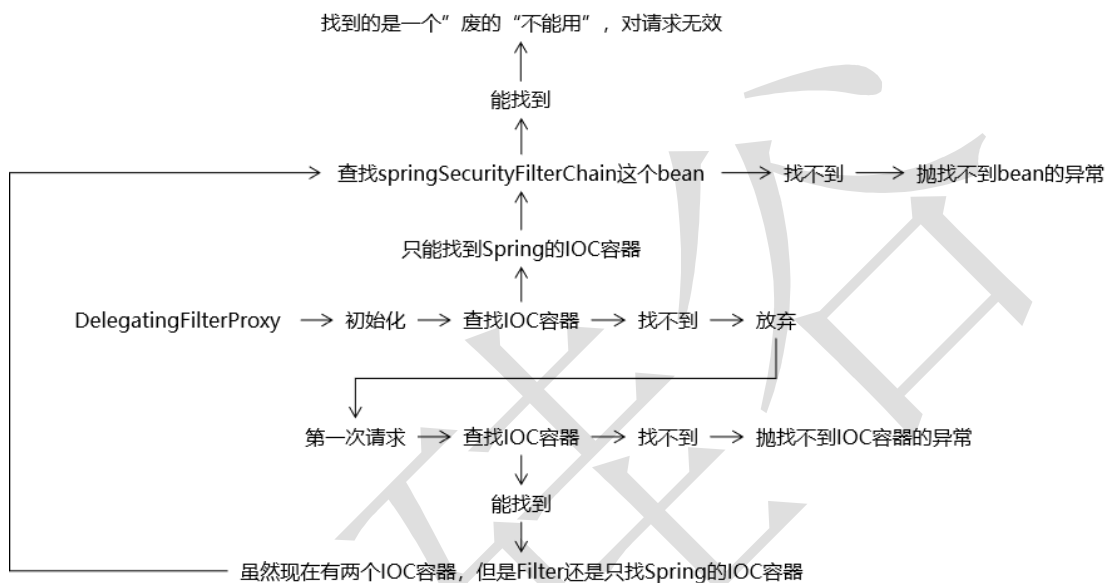
```
org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'springSecurityFilterChain' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:747)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:1344)
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:284)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:202)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1086)
    at org.springframework.web.filter.DelegatingFilterProxy.initDelegate(DelegatingFilterProxy.java:327)
    at org.springframework.web.filter.DelegatingFilterProxy.initFilterBean(DelegatingFilterProxy.java:235)
    at org.springframework.web.filter.GenericFilterBean.init(GenericFilterBean.java:236)
    at org.apache.catalina.core.ApplicationFilterConfig.initFilter(ApplicationFilterConfig.java:279)
    at org.apache.catalina.core.ApplicationFilterConfig.getFilter(ApplicationFilterConfig.java:260)
    at org.apache.catalina.core.ApplicationFilterConfig.<init>(ApplicationFilterConfig.java:105)
    at org.apache.catalina.core.StandardContext.filterStart(StandardContext.java:4830)
    at org.apache.catalina.core.StandardContext.startInternal(StandardContext.java:5510)
    at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:150)
    at org.apache.catalina.core.ContainerBase$StartChild.call(ContainerBase.java:1575)
    at org.apache.catalina.core.ContainerBase$StartChild.call(ContainerBase.java:1565)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
```

2.3 分析问题

2.3.1 明确三大组件启动顺序

首先：ContextLoaderListener 初始化，创建 Spring 的 IOC 容器
 其次：DelegatingFilterProxy 初始化，查找 IOC 容器、查找 bean
 最后：DispatcherServlet 初始化，创建 SpringMVC 的 IOC 容器

2.3.2 DelegatingFilterProxy 查找 IOC 容器然后查找 bean 的工作机制



2.4 解决方案一：把两个 IOC 容器合二为一

不使用 ContextLoaderListener，让 DispatcherServlet 加载所有 Spring 配置文件。

- DelegatingFilterProxy 在初始化时查找 IOC 容器，找不到，放弃。
- 第一次请求时再次查找。
- 找到 SpringMVC 的 IOC 容器。
- 从这个 IOC 容器中找到所需要的 bean。

遗憾：会破坏现有程序的结构。原本是 ContextLoaderListener 和 DispatcherServlet 两个组件创建两个 IOC 容器，现在改成只有一个。

2.5 解决方案二：改源码



修改 DelegatingFilterProxy 的源码，修改两处：

2.5.1 初始化时直接跳过查找 IOC 容器的环节

```
@Override
protected void initFilterBean() throws ServletException {
    synchronized (this.delegateMonitor) {
        if (this.delegate == null) {
            // If no target bean name specified, use filter name.
            if (this.targetBeanName == null) {
                this.targetBeanName = getFilterName();
            }
            // Fetch Spring root application context and initialize the delegate early,
            // if possible. If the root application context will be started after this
            // filter proxy, we'll have to resort to lazy initialization.
            /*WebApplicationContext wac = findWebApplicationContext();
            if (wac != null) {
                this.delegate = initDelegate(wac);
            }*/
        }
    }
}
```

2.5.2 第一次请求的时候直接找 SpringMVC 的 IOC 容器

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)
    throws ServletException, IOException {

    // Lazily initialize the delegate if necessary.
```



```
Filter delegateToUse = this.delegate;
if (delegateToUse == null) {
    synchronized (this.delegateMonitor) {
        delegateToUse = this.delegate;
        if (delegateToUse == null) {

            // 把原来的查找 IOC 容器的代码注释掉
            // WebApplicationContext wac = findWebApplicationContext();

            // 按我们自己的需要重新编写
            // 1.获取 ServletContext 对象
            ServletContext sc = this.getServletContext();

            // 2.拼接 SpringMVC 将 IOC 容器存入 ServletContext 域的时候使用的属性
            String servletName = "springDispatcherServlet";

            String attrName = FrameworkServlet.SERVLET_CONTEXT_PREFIX +
servletName;

            // 3.根据 attrName 从 ServletContext 域中获取 IOC 容器对象
            WebApplicationContext wac = (WebApplicationContext)
sc.getAttribute(attrName);

            if (wac == null) {
                throw new IllegalStateException("No WebApplicationContext found: " +
                    "no ContextLoaderListener or DispatcherServlet registered?");
            }
            delegateToUse = initDelegate(wac);
        }
        this.delegate = delegateToUse;
    }
}

// Let the delegate perform the actual doFilter operation.
invokeDelegate(delegateToUse, request, response, filterChain);
}
```

2.6 意外收获

发现了 SpringSecurity 的工作原理：在初始化时或第一次请求时准备好过滤器链。具体任务由具体过滤器来完成。

```
org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter
org.springframework.security.web.context.SecurityContextPersistenceFilter
org.springframework.security.web.header.HeaderWriterFilter
org.springframework.security.web.csrf.CsrfFilter
org.springframework.security.web.authentication.logout.LogoutFilter
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter
org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter
org.springframework.security.web.authentication.www.BasicAuthenticationFilter
org.springframework.security.web.savedrequest.RequestCacheAwareFilter
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter
org.springframework.security.web.authentication.AnonymousAuthenticationFilter
org.springframework.security.web.session.SessionManagementFilter
org.springframework.security.web.access.ExceptionTranslationFilter
org.springframework.security.web.access.intercept.FilterSecurityInterceptor
```

2.7 目标 1：放行登录页和静态资源

2.7.1 思路

在 SpringSecurity 的配置类 WebAppSecurityConfig 中重写 configure(HttpSecurity security)方法并设置。

2.7.2 代码

```
@Override
protected void configure(HttpSecurity security) throws Exception {

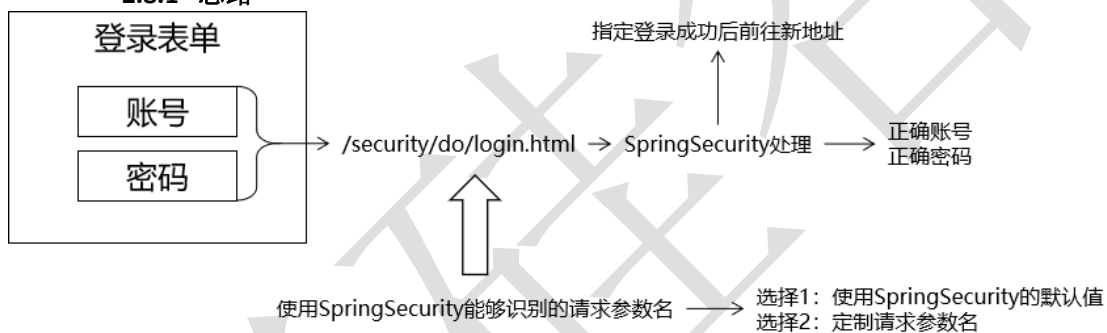
    security
        .authorizeRequests() // 对请求进行授权
        .antMatchers("/admin/to/login/page.html") // 针对登录页进行设置
        .permitAll() // 无条件访问
        .antMatchers("/bootstrap/**") // 针对静态资源进行设置，无条件访问
        .permitAll() // 针对静态资源进行设置，无条件访问
        .antMatchers("/crowd/**") // 针对静态资源进行设置，无条件访问
        .permitAll() // 针对静态资源进行设置，无条件访问
        .antMatchers("/css/**") // 针对静态资源进行设置，无条件访问
        .permitAll() // 针对静态资源进行设置，无条件访问
        .antMatchers("/fonts/**") // 针对静态资源进行设置，无条件访问
        .permitAll() // 针对静态资源进行设置，无条件访问
        .antMatchers("/img/**") // 针对静态资源进行设置，无条件访问
        .permitAll() // 针对静态资源进行设置，无条件访问
        .antMatchers("/jquery/**") // 针对静态资源进行设置，无条件访问
        .permitAll() // 针对静态资源进行设置，无条件访问
```

```
.antMatchers("/layer/**") // 针对静态资源进行设置，无条件访问
.permitAll() // 针对静态资源进行设置，无条件访问
.antMatchers("/script/**") // 针对静态资源进行设置，无条件访问
.permitAll() // 针对静态资源进行设置，无条件访问
.antMatchers("/ztree/**") // 针对静态资源进行设置，无条件访问
.permitAll()
.anyRequest()
.authenticated();

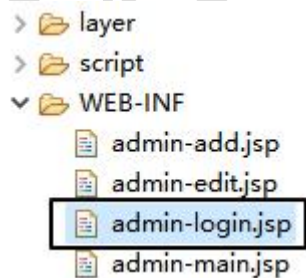
}
```

2.8 目标 2：提交登录表单做内存认证

2.8.1 思路



2.8.2 操作 1：设置表单



```
<form action="/security/do/login.html" method="post" class="form-signin" role="form">
  <h2 class="form-signin-heading">
    <i class="glyphicon glyphicon-log-in"></i> 管理员登录
  </h2>
  <p>${requestScope.exception.message }</p>
  <p>${SPRING_SECURITY_LAST_EXCEPTION.message }</p>
  <div class="form-group has-success has-feedback">
    <input type="text" name="loginAcct" value="tom" class="form-control" id="inputSuccess4"
      placeholder="请输入登录账号" autofocus> <span
      class="glyphicon glyphicon-user form-control-feedback"></span>
```

```
</div>
<div class="form-group has-success has-feedback">
    <input type="text" name="userPswd" value="123123" class="form-control" id="inputSuccess4"
        placeholder="请输入登录密码" style="margin-top: 10px;"> <span
        class="glyphicon glyphicon-lock form-control-feedback"></span>
</div>
<button type="submit" class="btn btn-lg btn-success btn-block">登录</button>
</form>
```

2.8.3 操作 2: SpringSecurity 配置

```
security
.....

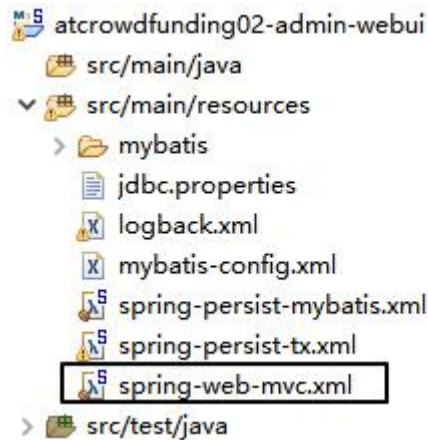
.anyRequest()                // 其他任意请求
.authenticated()             // 认证后访问
.and()
.csrf()                      // 防跨站请求伪造功能
.disable()                   // 禁用
.formLogin()                 // 开启表单登录的功能
.loginPage("/admin/to/login/page.html") // 指定登录页面
.loginProcessingUrl("/security/do/login.html") // 指定处理登录请求的地址
.defaultSuccessUrl("/admin/to/main/page.html") // 指定登录成功后前往的地址
.usernameParameter("loginAcct") // 账号的请求参数名称
.passwordParameter("userPswd") // 密码的请求参数名称
;
```

```
@Override
protected void configure(AuthenticationManagerBuilder builder) throws Exception {

    // 临时使用内存版登录的模式测试代码
    builder.inMemoryAuthentication().withUser("tom").password("123123").roles("ADMIN");

}
```

2.8.4 操作 3：取消以前的自定义登录拦截器



<!-- 注册拦截器：使用 SpringSecurity 后当前自定义的登录拦截器不再使用了

<mvc:interceptors>

<mvc:interceptor>

 mvc:mapping 配置要拦截的资源

 /*对应一层路径，比如：/aaa

 /**对应多层路径，比如：/aaa/bbb 或/aaa/bbb/ccc 或/aaa/bbb/ccc/ddd

 <mvc:mapping path="/**"/>

 mvc:exclude-mapping 配置不拦截的资源

 <mvc:exclude-mapping path="/admin/to/login/page.html"/>

 <mvc:exclude-mapping path="/admin/do/login.html"/>

 <mvc:exclude-mapping path="/admin/do/logout.html"/>

 配置拦截器类

 <bean class="com.atguigu.crowd.mvc.interceptor.LoginInterceptor"/>

 </mvc:interceptor>

</mvc:interceptors> -->

2.9 目标 3：退出登录

security

.....

.and()

.logout() // 开启退出登录功能

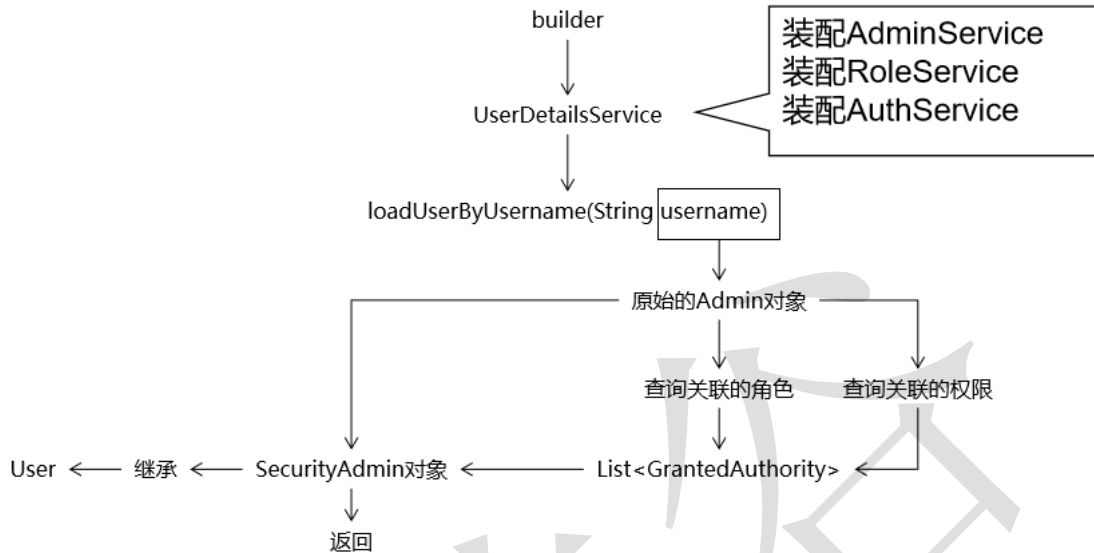
.logoutUrl("/seucrity/do/logout.html") // 指定退出登录地址

.logoutSuccessUrl("/admin/to/login/page.html") // 指定退出成功以后前往的地址

;

2.10 目标 4：把内存登录改成数据库登录

2.10.1 思路



2.10.2 操作 1：根据 adminId 查询已分配的角色

这个操作以前已经写好啦！

```

public interface RoleService {

    PageInfo<Role> getPageInfo(Integer pageNum, Integer pageSize, String keyword);

    void saveRole(Role role);

    void updateRole(Role role);

    void removeRole(List<Integer> roleIdList);

    List<Role> getAssignedRole(Integer adminId);

    List<Role> getUnAssignedRole(Integer adminId);

}
    
```

2.10.3 操作 2：根据 adminId 查询已分配权限

```

public interface AuthService {

    List<Auth> getAll();

}
    
```

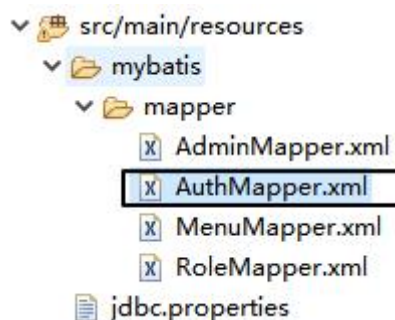
```
List<Integer> getAssignedAuthIdByRoleId(Integer roleId);

void saveRoleAuthRelationship(Map<String, List<Integer>> map);

List<String> getAssignedAuthNameByAdminId(Integer adminId);

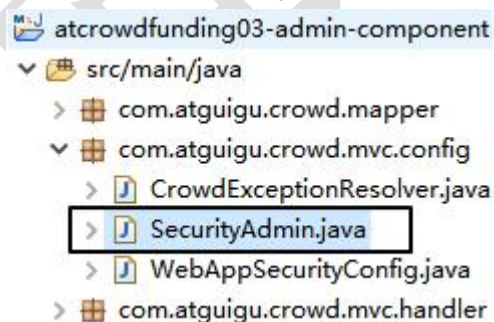
}
```

SQL 语句如下:



```
<select id="selectAssignedAuthNameByAdminId" resultType="string">
    SELECT DISTINCT t_auth.name
    FROM t_auth
    LEFT JOIN inner_role_auth ON t_auth.id=inner_role_auth.auth_id
    LEFT JOIN inner_admin_role ON inner_admin_role.role_id=inner_role_auth.role_id
    WHERE inner_admin_role.admin_id=#{adminId} and t_auth.name != "" and t_auth.name is
not null
</select>
```

2.10.4 操作 3: 创建 SecurityAdmin 类



```
/**
 * 考虑到 User 对象中仅仅包含账号和密码, 为了能够获取到原始的 Admin 对象, 专门创建
 * 这个类对 User 类进行扩展
 * @author Lenovo
 *
 */
```

```
public class SecurityAdmin extends User {

    private static final long serialVersionUID = 1L;

    // 原始的 Admin 对象，包含 Admin 对象的全部属性
    private Admin originalAdmin;

    public SecurityAdmin(
        // 传入原始的 Admin 对象
        Admin originalAdmin,

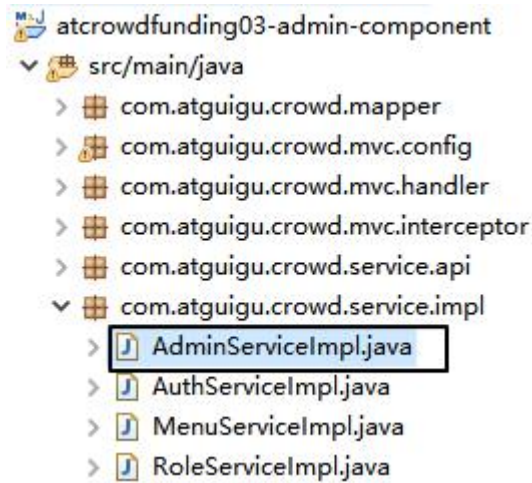
        // 创建角色、权限信息的集合
        List<GrantedAuthority> authorities) {

        // 调用父类构造器
        super(originalAdmin.getLoginAcct(), originalAdmin.getUserPswd(), authorities);

        // 给本类的 this.originalAdmin 赋值
        this.originalAdmin = originalAdmin;
    }

    // 对外提供的获取原始 Admin 对象的 getXxx()方法
    public Admin getOriginalAdmin() {
        return originalAdmin;
    }
}
```


2.10.5 操作 4：根据账号查询 Admin



```
@Override
public Admin getAdminByLoginAcct(String username) {

    AdminExample example = new AdminExample();

    Criteria criteria = example.createCriteria();

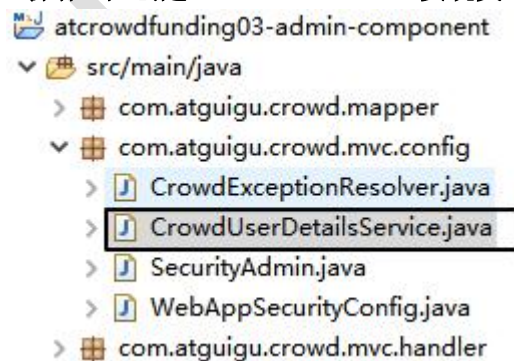
    criteria.andLoginAcctEqualTo(username);

    List<Admin> list = adminMapper.selectByExample(example);

    Admin admin = list.get(0);

    return admin;
}
```

2.10.6 操作 5：创建 UserDetailsService 实现类



```
@Component
public class CrowdUserDetailsService implements UserDetailsService {
```

```
@Autowired
private AdminService adminService;

@Autowired
private RoleService roleService;

@Autowired
private AuthService authService;

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

    // 1.根据账号名称查询 Admin 对象
    Admin admin = adminService.getAdminByLoginAcct(username);

    // 2.获取 adminId
    Integer adminId = admin.getId();

    // 3.根据 adminId 查询角色信息
    List<Role> assignedRoleList = roleService.getAssignedRole(adminId);

    // 4.根据 adminId 查询权限信息
    List<String> authNameList = authService.getAssignedAuthNameByAdminId(adminId);

    // 5.创建集合对象用来存储 GrantedAuthority
    List<GrantedAuthority> authorities = new ArrayList<>();

    // 6.遍历 assignedRoleList 存入角色信息
    for (Role role : assignedRoleList) {

        // 注意：不要忘了加前缀！
        String roleName = "ROLE_" + role.getName();

        SimpleGrantedAuthority simpleGrantedAuthority = new
SimpleGrantedAuthority(roleName);

        authorities.add(simpleGrantedAuthority);
    }

    // 7.遍历 authNameList 存入权限信息
```

```
for (String authName : authNameList) {  
  
    SimpleGrantedAuthority simpleGrantedAuthority = new  
SimpleGrantedAuthority(authName);  
  
    authorities.add(simpleGrantedAuthority);  
}  
  
// 8.封装 SecurityAdmin 对象  
SecurityAdmin securityAdmin = new SecurityAdmin(admin, authorities);  
  
return securityAdmin;  
}  
}
```

2.10.7 操作 6：在配置类中使用 UserDetailsService

```
@Override  
protected void configure(AuthenticationManagerBuilder builder) throws Exception {  
  
    // 临时使用内存版登录的模式测试代码  
    // builder.inMemoryAuthentication().withUser("tom").password("123123").roles("ADMIN");  
  
    // 正式功能中使用基于数据库的认证  
    builder.userDetailsService(userDetailsService);  
}
```

2.11 目标 5：密码加密

2.11.1 修改 t_admin 表结构

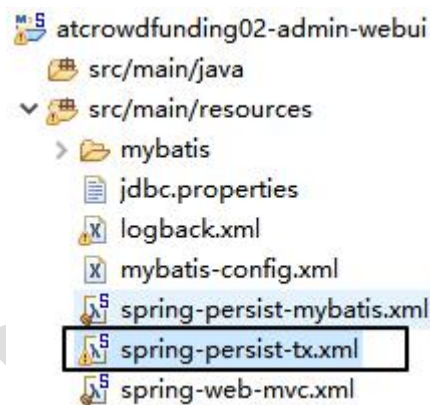
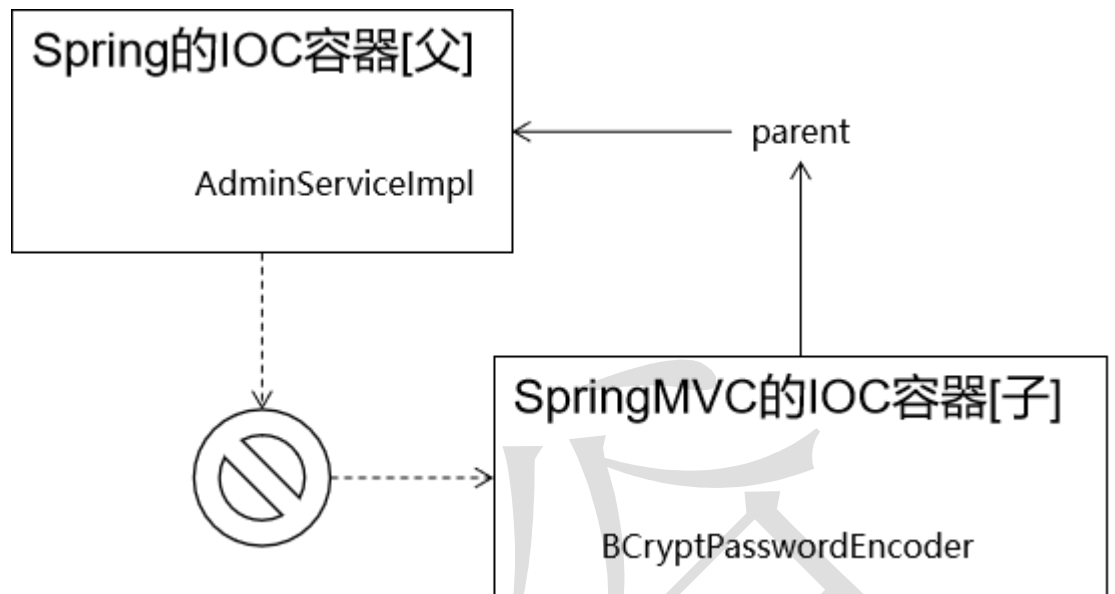
修改的原因：以前使用 JDK 自带的 MessageDigest 进行加密操作，生成的密文长度为 32。现在使用带盐值的加密方式，生成的密文长度超过这个数值，所以要修改。

```
ALTER TABLE `project_crowd`.`t_admin` CHANGE `user_pswd` `user_pswd` CHAR(100) CHARSET  
utf8 COLLATE utf8_general_ci NOT NULL;
```

2.11.2 准备 BCryptPasswordEncoder 对象

注意：如果在 SpringSecurity 的配置类中用 @Bean 注解将

BCryptPasswordEncoder 对象存入 IOC 容器，那么 Service 组件将获取不到。



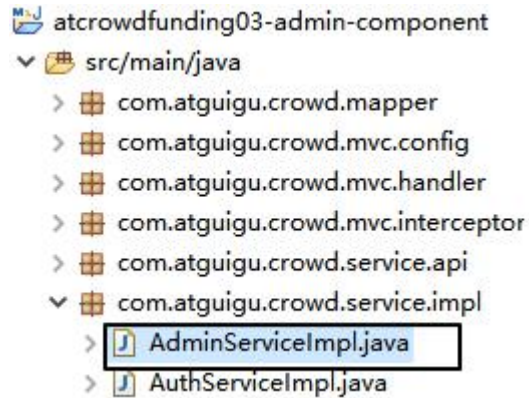
```
<!-- 配置 BCryptPasswordEncoder 的 bean -->
<bean id="passwordEncoder"
class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>
```

然后在有需要的地方@Autowired 注解装配即可。

2.11.3 使用 BCryptPasswordEncoder 对象

```
// 正式功能中使用基于数据库的认证
builder
    .userService(userDetailsService)
    .passwordEncoder(passwordEncoder);
```

2.11.4 使用 BCryptPasswordEncoder 在保存 Admin 时加密

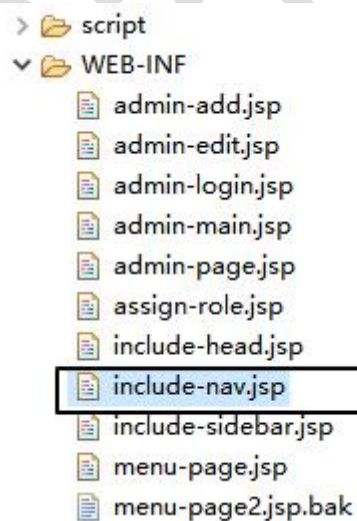


```
@Autowired
private BCryptPasswordEncoder passwordEncoder;

@Override
public void saveAdmin(Admin admin) {

    // 1. 密码加密
    String userPswd = admin.getUserPswd();
    // userPswd = CrowdUtil.md5(userPswd);
    userPswd = passwordEncoder.encode(userPswd);
    admin.setUserPswd(userPswd);
}
```

2.12 目标 6：在页面上显示用户昵称



2.12.1 导入标签库

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security" %>
```

2.12.2 通过标签获取已登录用户信息

```
<security:authentication property="principal.originalAdmin.userName"/>
```

分析过程:

<p>显示出来才发现, principal 原来是我们自己封装的 SecurityAdmin 对象</p>
 <p>SpringSecurity 处理完登录操作之后把登录成功的 User 对象以 principal 属性名存入了 UsernamePasswordAuthenticationToken 对象</p>
 Principal: <security:authentication property="principal.class.name"/>

 访问 SecurityAdmin 对象的属性 : <security:authentication property="principal.originalAdmin.loginAcct"/>

 访问 SecurityAdmin 对象的属性 : <security:authentication property="principal.originalAdmin.userPswd"/>

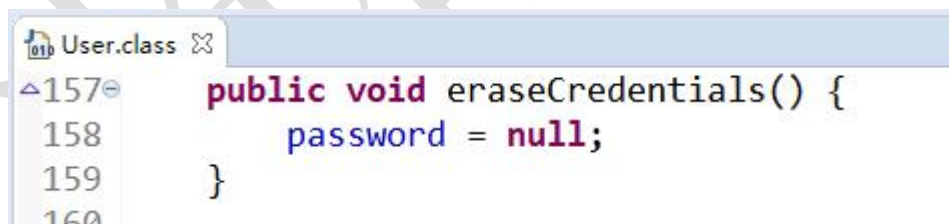
 访问 SecurityAdmin 对象的属性 : <security:authentication property="principal.originalAdmin.userName"/>

 访问 SecurityAdmin 对象的属性 : <security:authentication property="principal.originalAdmin.email"/>

 访问 SecurityAdmin 对象的属性 : <security:authentication property="principal.originalAdmin.createTime"/>

2.13 目标 7: 密码的擦除

本身 SpringSecurity 是会自动把 User 对象中的密码部分擦除。



```
public void eraseCredentials() {  
    password = null;  
}
```

但是我们创建 SecurityAdmin 对象扩展了 User 对象, User 对象中的密码被擦除了, 但是原始 Admin 对象中的密码没有擦除。

如果要把原始的 Admin 对象中的密码也擦除需要修改 SecurityAdmin 类代码:

```
public SecurityAdmin(  
    // 传入原始的 Admin 对象  
    Admin originalAdmin,  
  
    // 创建角色、权限信息的集合  
    List<GrantedAuthority> authorities) {  
  
    // 调用父类构造器  
    super(originalAdmin.getLoginAcct(), originalAdmin.getUserPswd(), authorities);
```

```
// 给本类的 this.originalAdmin 赋值
this.originalAdmin = originalAdmin;

// 将原始 Admin 对象中的密码擦除
this.originalAdmin.setUserPswd(null);

}
```

擦除密码是在不影响登录认证的情况下，避免密码泄露，增加系统安全性。

2.14 目标 8：权限控制

2.14.1 设置测试数据

运行时计算权限需要的数据：

用户：adminOperator

角色：经理

权限：无

角色：经理操作者

权限：user:save

最终组装后：ROLE_经理，ROLE_经理操作者，user:save

用户：roleOperator

角色：部长

权限：无

角色：部长操作者

权限：role:delete

最终组装后：ROLE_部长，ROLE_部长操作者，role:delete，user:get

测试时进行操作的数据：

admin01

admin02

.....

role01

role02

.....

2.14.2 测试 1

要求：访问 Admin 分页功能时具备“经理”角色

“加锁”的代码：

```
security

.antMatchers("/admin/get/page.html")// 针对分页显示 Admin 数据设定访问控制
.hasRole("经理") // 要求具备经理角色
```

效果：

adminOperator 能够访问

roleOperator 不能访问

HTTP Status 403 - Access is denied

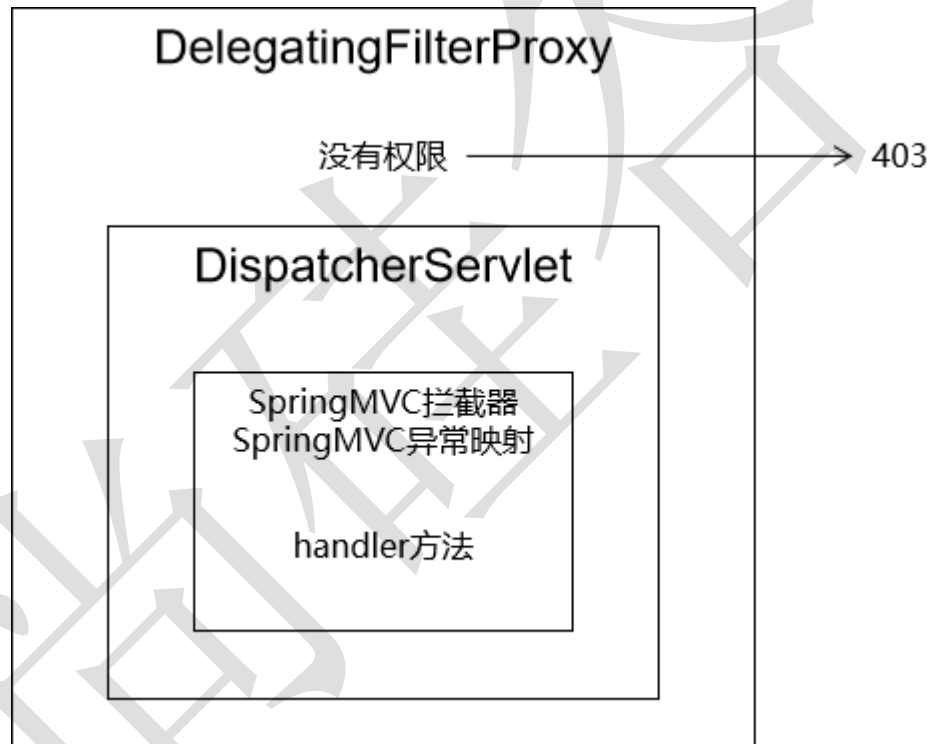
type Status report

message Access is denied

description Access to the specified resource has been forbidden.

Apache Tomcat/7.0.57

这个结果为什么没有经过异常映射机制？



所以要在 SpringSecurity 的配置类中进行配置

```

security
    .and()
    .exceptionHandling()
    .accessDeniedHandler(new AccessDeniedHandler() {

        @Override
        public void handle(HttpServletRequest request, HttpServletResponse response,
            AccessDeniedException accessDeniedException) throws IOException,
            ServletException {
            request.setAttribute("exception", new
                Exception(CrowdConstant.MESSAGE_ACCESS_DENIED));
        }
    });
  
```



```
request.getRequestDispatcher("/WEB-INF/system-error.jsp").forward(request,
response);
    }
}
```

2.14.3 测试 2

要求：访问 Role 的分页功能时具备“部长”角色

“加锁”的代码：

```
@PreAuthorize("hasRole('部长')")
// @ResponseBody
@RequestMapping("/role/get/page/info.json")
public ResultEntity<PageInfo<Role>> getPageInfo(
    @RequestParam(value="pageNum", defaultValue="1") Integer pageNum,
    @RequestParam(value="pageSize", defaultValue="5") Integer pageSize,
    @RequestParam(value="keyword", defaultValue="") String keyword
){

    // 调用 Service 方法获取分页数据
    PageInfo<Role> pageInfo = roleService.getPageInfo(pageNum, pageSize, keyword);

    // 封装到 ResultEntity 对象中返回（如果上面的操作抛出异常，交给异常映射机制处理）
    return ResultEntity.successWithData(pageInfo);
}
```

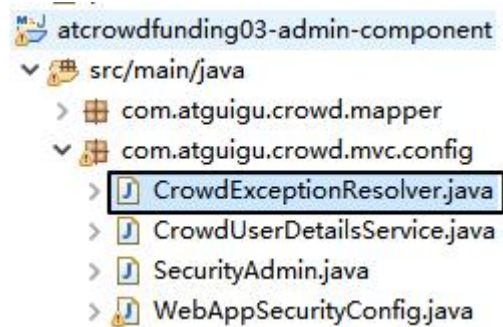
要“加锁”的代码生效：需要在配置类上加注解

```
// 启用全局方法权限控制功能，并且设置 prePostEnabled = true。保证 @PreAuthority、
// @PostAuthority、@PreFilter、@PostFilter 生效
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebAppSecurityConfig extends WebSecurityConfigurerAdapter {
```

效果：

adminOperator 不能访问
roleOperator 能够访问

补充：完善基于注解的异常映射



```
@ExceptionHandler(value = Exception.class)
public ModelAndView resolveException(
    Exception exception,
    HttpServletRequest request,
    HttpServletResponse response
) throws IOException {

    String viewName = "system-error";

    return commonResolve(viewName, exception, request, response);
}
```

发现：基于注解的异常映射和基于 XML 的异常映射如果映射同一个异常类型，那么基于注解的方案优先。

2.14.4 测试 3

要求：访问 Admin 保存功能时具备 user:save 权限

“加锁”的代码：

```
@PreAuthorize("hasAuthority('user:save')")
@RequestMapping("/admin/save.html")
public String save(Admin admin) {

    adminService.saveAdmin(admin);

    return "redirect:/admin/get/page.html?pageNum="+Integer.MAX_VALUE;
}
```

效果：

adminOperator 能够访问
roleOperator 不能访问

2.14.5 测试 4

要求：访问 Admin 分页功能时具备“经理”角色或“user:get”权限二者之一

“加锁”的代码：

```
security
```

```
.antMatchers("/admin/get/page.html")// 针对分页显示 Admin 数据设定访问控制
.access("hasRole('经理') OR hasAuthority('user:get')") // 要求具备“经理”角色和
“user:get”权限二者之一
```

效果：

adminOperator 能够访问
roleOperator 能够访问

PS：附带看一下其他注解（了解）

@PostAuthorize：先执行方法然后根据方法返回值判断是否具备权限。

例如：查询一个 Admin 对象，在 **@PostAuthorize** 注解中和当前登录的 Admin 对象进行比较，如果不一致，则判断为不能访问。实现“只能查自己”效果。

```
@PostAuthorize("returnObject.data.loginAcct == principal.username")
```

使用 returnObject 获取到方法返回值，使用 principal 获取到当前登录用户的主体对象

通过故意写错表达式，然后从异常信息中发现表达式访问的是下面这个类的属性：

```
org.springframework.security.access.expression.method.MethodSecurityExpressionRoot
```

@PreFilter：在方法执行前对传入的参数进行过滤。只能对集合类型的数据进行过滤。

```
@PreFilter(value="filterObject%2==0")
@ResponseBody
@RequestMapping("/admin/test/pre/filter")
public ResultEntity<List<Integer>> saveList(@RequestBody List<Integer> valueList) {
    return ResultEntity.successWithData(valueList);
}
```

@PostFilter：在方法执行后对方法返回值进行过滤。只能对集合类型的数据进行过滤。

2.15 目标 9：页面元素的权限控制

2.15.1 要求

页面上的局部元素，根据访问控制规则进行控制。

2.15.2 标签库

```
<security:authorize access="hasRole('经理')">
    <!-- 开始和结束标签之间是要进行权限控制的部分。检测当前用户是否有权限，有权限
    就显示这里的内容，没有权限就不显示。 -->
    .....
</security:authorize>
```

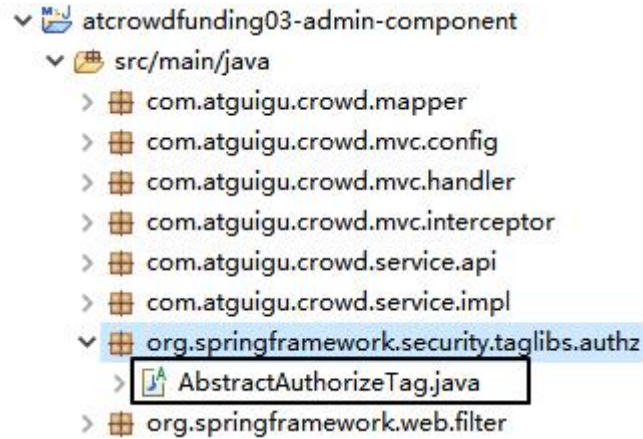
access 属性可以传入权限控制相关的表达式。

2.15.3 异常

No visible WebSecurityExpressionHandler instance could be found

原因：AbstractAuthorizeTag 类默认是查找“根级别”的 IOC 容器。而根级别的 IOC 容器中没有扫描 SpringSecurity 的配置类，所以没有相关的 bean。

解决办法：修改 AbstractAuthorizeTag 类的源码。



```
@SuppressWarnings({ "unchecked", "rawtypes" })
private SecurityExpressionHandler<FilterInvocation> getExpressionHandler()
    throws IOException {
    //
    // Application Context appContext =
    SecurityWebApplicationContextUtils.findRequiredWebApplicationContext(getServletContext())
    ;

    // 1.获取 ServletContext 对象
    ServletContext servletContext = getServletContext();

    // 2.拼接 SpringMVC 在 ServletContext 域中的属性名
    String attrName = FrameworkServlet.SERVLET_CONTEXT_PREFIX +
    "springDispatcherServlet";

    // 3.从 ServletContext 域中获取 IOC 容器对象
    ApplicationContext appContext = (ApplicationContext)
    servletContext.getAttribute(attrName);

    Map<String, SecurityExpressionHandler> handlers = appContext
        .getBeansOfType(SecurityExpressionHandler.class);

    for (SecurityExpressionHandler h : handlers.values()) {
        if (FilterInvocation.class.equals(GenericTypeResolver.resolveTypeArgument(
            h.getClass(), SecurityExpressionHandler.class))) {
            return h;
        }
    }
}
```

```
throw new IOException(  
    "No visible WebSecurityExpressionHandler instance could be found in the  
application "  
    + "context. There must be at least one in order to support expressions in  
JSP 'authorize' tags.");  
}
```