API权限控制在微服务架构中的设计与体现

# 引言

Auth 系统有两方面的需求：

1. 认证与鉴权，对于请求的用户身份的授权以及合法性鉴权；
2. API级别的操作权限控制

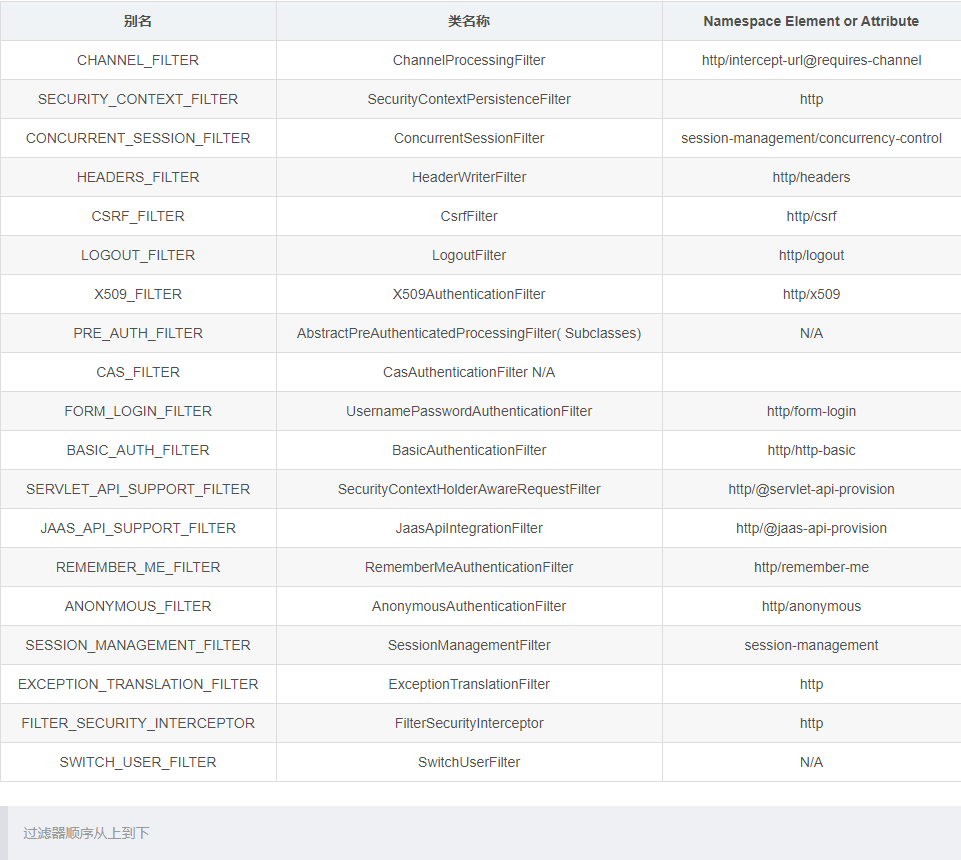
token的认证鉴权不是本文要探讨的内容。本文的重点是API级别的操作权限控制，这个在第一点之后，当鉴定完用户身份合法性之后，需要对该用户的某个具体的请求是否具有该操作权限进行校验。当然此处是设计token合法性校验在先，其次再对操作权限进行验证，如果前一个验证不通过则直接拒绝，通过则进入操作权限验证。

# Spring Security 过滤器链

如何进行 API级别权限校验？

添加一个用于API级别权限校验的过滤器。

Spring Security 的默认过滤器如下：



其中表格中倒数第二个过滤器 FilterSecurityInterceptor 是 filterchain 中比较复杂，也是比较核心的过滤器，主要负责web应用安全授权的工作。

# 自定义鉴权过滤器

这个自定义的过滤器需要依赖3个接口：

* AuthenticationManager:认证管理器，实现用户认证的入口;
* AccessDecisionManager:访问决策器，决定某个用户具有的角色，是否有足够的权限去访问某个资源;
* FilterInvocationSecurityMetadataSource:资源源数据定义，即定义某一资源可以被哪些角色访问.

1. **package** com.fengxuechao.examples.auth.authorization;
3. **import** lombok.extern.slf4j.Slf4j;
4. **import** org.springframework.security.access.SecurityMetadataSource;
5. **import** org.springframework.security.access.intercept.AbstractSecurityInterceptor;
6. **import** org.springframework.security.access.intercept.InterceptorStatusToken;
7. **import** org.springframework.security.authentication.AuthenticationManager;
8. **import** org.springframework.security.web.FilterInvocation;
10. **import** javax.servlet.\*;
11. **import** java.io.IOException;
13. /\*\*
14. \* 比较核心的过滤器: 主要负责web应用鉴权的工作。
15. \* 需要依赖:
16. \* - AuthenticationManager:认证管理器，实现用户认证的入口;
17. \* - AccessDecisionManager:访问决策器，决定某个用户具有的角色，是否有足够的权限去访问某个资源;
18. \* - FilterInvocationSecurityMetadataSource:资源源数据定义，即定义某一资源可以被哪些角色访问.
19. \*
20. \* @author fengxuechao
21. \* @version 0.1
22. \* @date 2019/6/17
23. \*/
24. @Slf4j
25. **public** **class** Oauth2FilterSecurityInterceptor **extends** AbstractSecurityInterceptor **implements** Filter {
27. **private** Oauth2FilterInvocationSecurityMetadataSource securityMetadataSource;
29. @Override
30. **public** **void** init(FilterConfig filterConfig) **throws** ServletException {
31. **if** (log.isInfoEnabled()) {
32. log.info("Oauth2FilterSecurityInterceptor init");
33. }
34. }
36. @Override
37. **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {
38. **if** (log.isInfoEnabled()) {
39. log.info("Oauth2FilterSecurityInterceptor doFilter");
40. }
41. FilterInvocation filterInvocation = **new** FilterInvocation(request, response, chain);
42. invoke(filterInvocation);
43. }
45. **public** **void** invoke(FilterInvocation filterInvocation) **throws** IOException, ServletException {
46. // filterInvocation里面有一个被拦截的url
47. // 里面调用 Oauth2AccessDecisionManager 的 getAttributes(Object object) 这个方法获取 filterInvocation 对应的所有权限
48. // 再调用 Oauth2AccessDecisionManager 的 decide方法来校验用户的权限是否足够
49. InterceptorStatusToken interceptorStatusToken = **super**.beforeInvocation(filterInvocation);
50. **try** {
51. // 执行下一个拦截器
52. filterInvocation.getChain().doFilter(filterInvocation.getRequest(), filterInvocation.getResponse());
53. } **finally** {
54. **super**.afterInvocation(interceptorStatusToken, **null**);
55. }
56. }
58. @Override
59. **public** **void** destroy() {
61. }
63. @Override
64. **public** Class<?> getSecureObjectClass() {
65. **return** FilterInvocation.**class**;
66. }
68. /\*\*
69. \* 资源源数据定义，设置为自定义的 SecureResourceFilterInvocationDefinitionSource
70. \*
71. \* @return
72. \*/
73. @Override
74. **public** SecurityMetadataSource obtainSecurityMetadataSource() {
75. **return** securityMetadataSource;
76. }
78. **public** **void** setOauth2AccessDecisionManager(Oauth2AccessDecisionManager accessDecisionManager) {
79. **super**.setAccessDecisionManager(accessDecisionManager);
80. }
82. @Override
83. **public** **void** setAuthenticationManager(AuthenticationManager authenticationManager) {
84. **super**.setAuthenticationManager(authenticationManager);
85. }
87. **public** **void** setSecurityMetadataSource(Oauth2FilterInvocationSecurityMetadataSource securityMetadataSource) {
88. **this**.securityMetadataSource = securityMetadataSource;
89. }
90. }

在第49行代码中调用了父类beforeInvocation方法，看下其中的部分源码：

1. **public** **abstract** **class** AbstractSecurityInterceptor **implements** InitializingBean, ApplicationEventPublisherAware, MessageSourceAware {
2. **protected** InterceptorStatusToken beforeInvocation(Object object) {
3. // 代码省略
5. // 根据 SecurityMetadataSource 获取配置的权限属性
6. Collection<ConfigAttribute> attributes = **this**.obtainSecurityMetadataSource()
7. .getAttributes(object);
9. // 代码省略
11. // 判断是否需要对认证实体重新认证，默认为否
12. Authentication authenticated = authenticateIfRequired();
14. // Attempt authorization
15. **try** {
16. // 决策管理器开始决定是否授权，如果授权失败，直接抛出 AccessDeniedException
17. **this**.accessDecisionManager.decide(authenticated, object, attributes);
18. }
19. **catch** (AccessDeniedException accessDeniedException) {
20. publishEvent(**new** AuthorizationFailureEvent(object, attributes, authenticated,
21. accessDeniedException));
23. **throw** accessDeniedException;
24. }
26. // 代码省略
27. }
28. }

# 自定义资源源数据定义

资源源数据定义，即定义某一资源可以被哪些角色访问 。

目前代码中为了方便直接写死资源所需权限，而在实际的生产环境中需要从数据源中取出。

1. **package** com.fengxuechao.examples.auth.authorization;
3. **import** com.fengxuechao.examples.auth.service.UserRolePermissionService;
4. **import** lombok.extern.slf4j.Slf4j;
5. **import** org.springframework.beans.factory.InitializingBean;
6. **import** org.springframework.security.access.ConfigAttribute;
7. **import** org.springframework.security.access.SecurityConfig;
8. **import** org.springframework.security.web.FilterInvocation;
9. **import** org.springframework.security.web.access.intercept.FilterInvocationSecurityMetadataSource;
10. **import** org.springframework.stereotype.Component;
12. **import** java.util.Collection;
14. /\*\*
15. \* 资源源数据定义，即定义某一资源可以被哪些角色访问
16. \*
17. \* @author fengxuechao
18. \* @version 0.1
19. \* @date 2019/6/14
20. \*/
21. @Slf4j
22. @Component
23. **public** **class** Oauth2FilterInvocationSecurityMetadataSource **implements** FilterInvocationSecurityMetadataSource, InitializingBean {
25. **private** UserRolePermissionService service;
27. **public** Oauth2FilterInvocationSecurityMetadataSource(UserRolePermissionService service) {
28. **this**.service = service;
29. }

32. @Override
33. **public** Collection<ConfigAttribute> getAttributes(Object object) **throws** IllegalArgumentException {
34. **if** ("/user/profile".equals(((FilterInvocation) object).getRequestUrl())) {
35. // [/user/profile] 不需要鉴权
36. **return** **null**;
37. }
38. /\*if (object instanceof FilterInvocation) {
39. FilterInvocation fi = (FilterInvocation) object;
40. String requestUrl = fi.getRequestUrl();
41. // 返回请求所需的权限
42. List<Role> roleList = service.findRoleListByPermissionUrl(requestUrl);
43. String[] roleArray = new String[roleList.size()];
44. roleArray = roleList.toArray(roleArray);
45. return SecurityConfig.createList(roleArray);
46. }
47. return Collections.EMPTY\_LIST;\*/
48. **return** SecurityConfig.createList("ROLE\_ADMIN");
49. }
51. @Override
52. **public** Collection<ConfigAttribute> getAllConfigAttributes() {
53. **return** **null**;
54. }
56. @Override
57. **public** **boolean** supports(Class<?> clazz) {
58. **return** **true**;
59. }
61. @Override
62. **public** **void** afterPropertiesSet() **throws** Exception {
64. }
65. }

# 自定义决策管理器

访问决策器，决定某个用户具有的角色，是否有足够的权限去访问某个资源。

其中：

* Authentication：用户凭证(已通过验证)
* Resource：资源 URL
* configAttributes：资源 URL 所需要的权限
* AccessDeniedException：资源拒绝访问
* InsufficientAuthenticationException：用户凭证不符

1. **package** com.fengxuechao.examples.auth.authorization;
3. **import** lombok.extern.slf4j.Slf4j;
4. **import** org.springframework.security.access.AccessDecisionManager;
5. **import** org.springframework.security.access.AccessDeniedException;
6. **import** org.springframework.security.access.ConfigAttribute;
7. **import** org.springframework.security.authentication.InsufficientAuthenticationException;
8. **import** org.springframework.security.core.Authentication;
9. **import** org.springframework.security.core.GrantedAuthority;
10. **import** org.springframework.security.web.FilterInvocation;
11. **import** org.springframework.stereotype.Component;
13. **import** java.util.Collection;
14. **import** java.util.Iterator;
16. /\*\*
17. \* 访问决策器，决定某个用户具有的角色，是否有足够的权限去访问某个资源
18. \*
19. \* @author fengxuechao
20. \* @version 0.1
21. \* @date 2019/6/14
22. \*/
23. @Slf4j
24. @Component
25. **public** **class** Oauth2AccessDecisionManager **implements** AccessDecisionManager {
27. /\*\*
28. \* @param authentication   用户凭证
29. \* @param resource         资源 URL
30. \* @param configAttributes 资源 URL 所需要的权限
31. \* @throws AccessDeniedException               资源拒绝访问
32. \* @throws InsufficientAuthenticationException 用户凭证不符
33. \*/
34. @Override
35. **public** **void** decide(Authentication authentication, Object resource, Collection<ConfigAttribute> configAttributes)
36. **throws** AccessDeniedException, InsufficientAuthenticationException {
37. log.info("[决策管理器]:开始判断请求 {} 需要的权限", ((FilterInvocation) resource).getRequestUrl());
38. **if** (configAttributes == **null** || configAttributes.isEmpty()) {
39. log.info("[决策管理器]:请求 {} 无需权限", ((FilterInvocation) resource).getRequestUrl());
40. **return**;
41. }
42. log.info("[决策管理器]:请求 {} 需要的权限 - {}", ((FilterInvocation) resource).getRequestUrl(), configAttributes);
43. // 判断用户所拥有的权限，是否符合对应的Url权限，用户权限是实现 UserDetailsService#loadUserByUsername 返回用户所对应的权限
44. Iterator<ConfigAttribute> ite = configAttributes.iterator();
45. log.info("[决策管理器]:用户 {} 拥有的权限 - {}", authentication.getName(), authentication.getAuthorities());
46. **while** (ite.hasNext()) {
47. ConfigAttribute neededAuthority = ite.next();
48. String neededAuthorityStr = neededAuthority.getAttribute();
49. **for** (GrantedAuthority existingAuthority : authentication.getAuthorities()) {
50. **if** (neededAuthorityStr.equals(existingAuthority.getAuthority())) {
51. **return**;
52. }
53. }
54. }
55. log.info("[决策管理器]:用户 {} 没有访问资源 {} 的权限!", authentication.getName(), ((FilterInvocation) resource).getRequestUrl());
56. **throw** **new** AccessDeniedException("权限不足!");
57. }
59. @Override
60. **public** **boolean** supports(ConfigAttribute attribute) {
61. **return** **true**;
62. }
64. /\*\*
65. \* 是否支持 FilterInvocationSecurityMetadataSource 需要将这里的false改为true
66. \*
67. \* @param clazz
68. \* @return
69. \*/
70. @Override
71. **public** **boolean** supports(Class<?> clazz) {
72. **return** **true**;
73. }
74. }

# 配置自定义鉴权过滤器在过滤器中的位置

最后还需要配置自定义过滤器的过滤器链中的位置。

说明：

HttpSecurity 有三个常用方法来配置：

* addFilterBefore(Filter filter, Class<? extends Filter> beforeFilter)：在 beforeFilter 之前添加 filter
* addFilterAfter(Filter filter, Class<? extends Filter> afterFilter)：在 afterFilter 之后添加 filter
* addFilterAt(Filter filter, Class<? extends Filter> atFilter)：在 atFilter 相同位置添加 filter， 此 filter 不覆盖 filter

通过在不同 Filter 的 doFilter() 方法中加断点调试，可以判断哪个 filter 先执行，从而判断 filter 的执行顺序 。

如果是其他的普通的自定义过滤器建议继承 GenericFilterBean

1. **package** com.fengxuechao.examples.auth.config;
3. **import** com.fengxuechao.examples.auth.authorization.Oauth2AccessDecisionManager;
4. **import** com.fengxuechao.examples.auth.authorization.Oauth2FilterInvocationSecurityMetadataSource;
5. **import** com.fengxuechao.examples.auth.authorization.Oauth2FilterSecurityInterceptor;
6. **import** lombok.extern.slf4j.Slf4j;
7. **import** org.springframework.beans.factory.annotation.Autowired;
8. **import** org.springframework.context.annotation.Configuration;
9. **import** org.springframework.security.authentication.AuthenticationManager;
10. **import** org.springframework.security.config.annotation.web.builders.HttpSecurity;
11. **import** org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
12. **import** org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;
13. **import** org.springframework.security.web.access.intercept.FilterSecurityInterceptor;
15. /\*\*
16. \* @author fengxuechao
17. \* @version 0.1
18. \* @date 2019/5/8
19. \*/
20. @Slf4j
21. @EnableResourceServer
22. @Configuration
23. **public** **class** ResourceServerConfig **extends** ResourceServerConfigurerAdapter {
25. @Autowired
26. AuthenticationManager manager;
28. @Autowired
29. Oauth2AccessDecisionManager accessDecisionManager;
31. @Autowired
32. Oauth2FilterInvocationSecurityMetadataSource securityMetadataSource;
34. @Override
35. **public** **void** configure(HttpSecurity http) **throws** Exception {
36. http.authorizeRequests().anyRequest().authenticated();
37. http.addFilterAfter(createApiAuthenticationFilter(), FilterSecurityInterceptor.**class**);
38. }
40. /\*\*
41. \* API权限控制
42. \* 过滤器优先度在 FilterSecurityInterceptor 之后
43. \* spring-security 的默认过滤器列表见 https://docs.spring.io/spring-security/site/docs/5.0.0.M1/reference/htmlsingle/#ns-custom-filters
44. \*
45. \* @return
46. \*/
47. **private** Oauth2FilterSecurityInterceptor createApiAuthenticationFilter() {
48. Oauth2FilterSecurityInterceptor interceptor = **new** Oauth2FilterSecurityInterceptor();
49. interceptor.setAuthenticationManager(manager);
50. interceptor.setAccessDecisionManager(accessDecisionManager);
51. interceptor.setSecurityMetadataSource(securityMetadataSource);
52. **return** interceptor;
53. }
54. }

# **配置用户权限**

为了方便调试，用户的权限写死为 ROLE\_USER

1. **package** com.fengxuechao.examples.auth.userdetails;
3. **import** org.springframework.security.core.authority.AuthorityUtils;
4. **import** org.springframework.security.core.userdetails.User;
5. **import** org.springframework.security.core.userdetails.UserDetails;
6. **import** org.springframework.security.core.userdetails.UserDetailsService;
7. **import** org.springframework.security.core.userdetails.UsernameNotFoundException;
8. **import** org.springframework.stereotype.Component;
10. /\*\*
11. \* @author fengxuechao
12. \* @version 0.1
13. \* @date 2019/5/15
14. \*/
15. @Component
16. **public** **class** UserDetailsServiceImpl **implements** UserDetailsService {
18. @Override
19. **public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {
20. **return** **new** User(username, "123456", AuthorityUtils.commaSeparatedStringToAuthorityList("ROLE\_USER"));
21. }
22. }

# 演示结果

## 测试用户拥有资源所需权限

将用户拥有的角色改成与资源所需角色一样

请求：

GET http://localhost:8080/order/1

HTTP/1.1 200

X-Application-Context: application:inMemory

X-Content-Type-Options: nosniff

X-XSS-Protection: 1; mode=block

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: 0

X-Frame-Options: DENY

Content-Type: text/plain;charset=UTF-8

Content-Length: 12

Date: Tue, 18 Jun 2019 01:50:48 GMT

order id : 1

Response code: 200; Time: 57ms; Content length: 12 bytes

日志：

2019-06-18 09:50:48.955  INFO 5288 --- [nio-8080-exec-3] .f.e.a.a.Oauth2FilterSecurityInterceptor : Oauth2FilterSecurityInterceptor doFilter

2019-06-18 09:50:48.955 DEBUG 5288 --- [nio-8080-exec-3] .f.e.a.a.Oauth2FilterSecurityInterceptor : Secure object: FilterInvocation: URL: /order/1; Attributes: [ROLE\_USER]

2019-06-18 09:50:48.956 DEBUG 5288 --- [nio-8080-exec-3] .f.e.a.a.Oauth2FilterSecurityInterceptor : Previously Authenticated: org.springframework.security.oauth2.provider.OAuth2Authentication@f5aeefea: Principal: org.springframework.security.core.userdetails.User@36ebcb: Username: user; Password: [PROTECTED]; Enabled: **true**; AccountNonExpired: **true**; credentialsNonExpired: **true**; AccountNonLocked: **true**; Granted Authorities: ROLE\_USER; Credentials: [PROTECTED]; Authenticated: **true**; Details: remoteAddress=127.0.0.1, tokenType=bearertokenValue=<TOKEN>; Granted Authorities: ROLE\_USER

2019-06-18 09:50:48.956  INFO 5288 --- [nio-8080-exec-3] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:开始判断请求 /order/1 需要的权限

2019-06-18 09:50:48.956  INFO 5288 --- [nio-8080-exec-3] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:请求 /order/1 需要的权限 - [ROLE\_USER]

2019-06-18 09:50:48.956  INFO 5288 --- [nio-8080-exec-3] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:用户 user 拥有的权限 - [ROLE\_USER]

2019-06-18 09:50:48.956 DEBUG 5288 --- [nio-8080-exec-3] .f.e.a.a.Oauth2FilterSecurityInterceptor : Authorization successful

2019-06-18 09:50:48.957 DEBUG 5288 --- [nio-8080-exec-3] .f.e.a.a.Oauth2FilterSecurityInterceptor : RunAsManager did not change Authentication object

结果服务期望效果

## 测试用户没有资源所需权限

将用户拥有的角色改成与资源所需角色不一样

请求：

GET http://localhost:8080/order/1

HTTP/1.1 403

Cache-Control: no-store

Pragma: no-cache

X-Content-Type-Options: nosniff

X-XSS-Protection: 1; mode=block

X-Frame-Options: DENY

Content-Type: application/json;charset=UTF-8

Transfer-Encoding: chunked

Date: Tue, 18 Jun 2019 01:44:49 GMT

{

  "error": "access\_denied",

  "error\_description": "权限不足!"

}

Response code: 403; Time: 35ms; Content length: 53 bytes

日志：

2019-06-18 09:44:44.684  INFO 10624 --- [nio-8080-exec-2] .f.e.a.a.Oauth2FilterSecurityInterceptor : Oauth2FilterSecurityInterceptor doFilter

2019-06-18 09:44:44.685 DEBUG 10624 --- [nio-8080-exec-2] .f.e.a.a.Oauth2FilterSecurityInterceptor : Public object - authentication not attempted

2019-06-18 09:44:49.448  INFO 10624 --- [nio-8080-exec-6] .f.e.a.a.Oauth2FilterSecurityInterceptor : Oauth2FilterSecurityInterceptor doFilter

2019-06-18 09:44:49.449 DEBUG 10624 --- [nio-8080-exec-6] .f.e.a.a.Oauth2FilterSecurityInterceptor : Secure object: FilterInvocation: URL: /order/1; Attributes: [ROLE\_ADMIN]

2019-06-18 09:44:49.449 DEBUG 10624 --- [nio-8080-exec-6] .f.e.a.a.Oauth2FilterSecurityInterceptor : Previously Authenticated: org.springframework.security.oauth2.provider.OAuth2Authentication@22d262ad: Principal: org.springframework.security.core.userdetails.User@36ebcb: Username: user; Password: [PROTECTED]; Enabled: **true**; AccountNonExpired: **true**; credentialsNonExpired: **true**; AccountNonLocked: **true**; Granted Authorities: ROLE\_USER; Credentials: [PROTECTED]; Authenticated: **true**; Details: remoteAddress=127.0.0.1, tokenType=bearertokenValue=<TOKEN>; Granted Authorities: ROLE\_USER

2019-06-18 09:44:49.450  INFO 10624 --- [nio-8080-exec-6] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:开始判断请求 /order/1 需要的权限

2019-06-18 09:44:49.450  INFO 10624 --- [nio-8080-exec-6] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:请求 /order/1 需要的权限 - [ROLE\_ADMIN]

2019-06-18 09:44:49.450  INFO 10624 --- [nio-8080-exec-6] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:用户 user 拥有的权限 - [ROLE\_USER]

2019-06-18 09:44:49.451  INFO 10624 --- [nio-8080-exec-6] c.f.e.a.a.Oauth2AccessDecisionManager    : [决策管理器]:用户 user 没有访问资源 /order/1 的权限!