**【Spring实战】----security4.1.3认证的过程以及原请求信息的缓存及恢复（RequestCache）**

原创 2016年12月15日 16:20:10

* 标签：
* [security4.1.3](http://so.csdn.net/so/search/s.do?q=security4.1.3&t=blog) /
* [认证过程](http://so.csdn.net/so/search/s.do?q=%E8%AE%A4%E8%AF%81%E8%BF%87%E7%A8%8B&t=blog)
* 3348

**一、先看下认证过程**

认证过程分为7步:  
1.用户访问网站，打开了一个链接(origin url)。

2.请求发送给服务器，服务器判断用户请求了受保护的资源。  
  
3.由于用户没有登录，服务器重定向到登录页面  
  
4.填写表单，点击登录  
  
5.浏览器将用户名密码以表单形式发送给服务器  
  
6.服务器验证用户名密码。成功，进入到下一步。否则要求用户重新认证（第三步）  
  
7.服务器对用户拥有的权限（角色）判定: 有权限，重定向到origin url; 权限不足，返回状态码403("forbidden").  
  
从第3步，我们可以知道，用户的请求被中断了。  
  
用户登录成功后（第7步），会被重定向到origin url，spring security通过使用缓存的request，使得被中断的请求能够继续执行。

**二、使用缓存**

用户登录成功后，页面重定向到origin url。浏览器发出的请求优先被拦截器RequestCacheAwareFilter拦截，RequestCacheAwareFilter通过其持有的RequestCache对象实现request的恢复。

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **public** **void** doFilter(ServletRequest request, ServletResponse response,
2. FilterChain chain) **throws** IOException, ServletException {
4. // request匹配，则取出，该操作同时会将缓存的request从session中删除
5. HttpServletRequest wrappedSavedRequest = requestCache.getMatchingRequest(
6. (HttpServletRequest) request, (HttpServletResponse) response);
8. // 优先使用缓存的request
9. chain.doFilter(wrappedSavedRequest == **null** ? request : wrappedSavedRequest,
10. response);
11. }

**三、何时缓存**

首先，我们需要了解下RequestCache以及ExceptionTranslationFilter。  
  
1）RequestCache  
  
RequestCache接口声明了缓存与恢复操作。默认实现类是HttpSessionRequestCache。接口的声明如下:

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **public** **interface** RequestCache {
3. // 将request缓存到session中
4. **void** saveRequest(HttpServletRequest request, HttpServletResponse response);
6. // 从session中取request
7. SavedRequest getRequest(HttpServletRequest request, HttpServletResponse response);
9. // 获得与当前request匹配的缓存，并将匹配的request从session中删除
10. HttpServletRequest getMatchingRequest(HttpServletRequest request,
11. HttpServletResponse response);
13. // 删除缓存的request
14. **void** removeRequest(HttpServletRequest request, HttpServletResponse response);
15. }

实现类为HttpSessionRequestCache

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **public** **class** HttpSessionRequestCache **implements** RequestCache {
2. **static** **final** String SAVED\_REQUEST = "SPRING\_SECURITY\_SAVED\_REQUEST";
3. **protected** **final** Log logger = LogFactory.getLog(**this**.getClass());
5. **private** PortResolver portResolver = **new** PortResolverImpl();
6. **private** **boolean** createSessionAllowed = **true**;
7. **private** RequestMatcher requestMatcher = AnyRequestMatcher.INSTANCE;
9. /\*\*
10. \* Stores the current request, provided the configuration properties allow it.
11. \*/
12. **public** **void** saveRequest(HttpServletRequest request, HttpServletResponse response) {
13. **if** (requestMatcher.matches(request)) {
14. DefaultSavedRequest savedRequest = **new** DefaultSavedRequest(request,
15. portResolver);
17. **if** (createSessionAllowed || request.getSession(**false**) != **null**) {
18. // Store the HTTP request itself. Used by
19. // AbstractAuthenticationProcessingFilter
20. // for redirection after successful authentication (SEC-29)
21. request.getSession().setAttribute(SAVED\_REQUEST, savedRequest);
22. logger.debug("DefaultSavedRequest added to Session: " + savedRequest);
23. }
24. }
25. **else** {
26. logger.debug("Request not saved as configured RequestMatcher did not match");
27. }
28. }
30. **public** SavedRequest getRequest(HttpServletRequest currentRequest,
31. HttpServletResponse response) {
32. HttpSession session = currentRequest.getSession(**false**);
34. **if** (session != **null**) {
35. **return** (SavedRequest) session.getAttribute(SAVED\_REQUEST);
36. }
38. **return** **null**;
39. }
41. **public** **void** removeRequest(HttpServletRequest currentRequest,
42. HttpServletResponse response) {
43. HttpSession session = currentRequest.getSession(**false**);
45. **if** (session != **null**) {
46. logger.debug("Removing DefaultSavedRequest from session if present");
47. session.removeAttribute(SAVED\_REQUEST);
48. }
49. }
51. **public** HttpServletRequest getMatchingRequest(HttpServletRequest request,
52. HttpServletResponse response) {
53. DefaultSavedRequest saved = (DefaultSavedRequest) getRequest(request, response);
55. **if** (saved == **null**) {
56. **return** **null**;
57. }
59. **if** (!saved.doesRequestMatch(request, portResolver)) {
60. logger.debug("saved request doesn't match");
61. **return** **null**;
62. }
64. removeRequest(request, response);
66. **return** **new** SavedRequestAwareWrapper(saved, request);
67. }
69. /\*\*
70. \* Allows selective use of saved requests for a subset of requests. By default any
71. \* request will be cached by the {@code saveRequest} method.
72. \* <p>
73. \* If set, only matching requests will be cached.
74. \*
75. \* @param requestMatcher a request matching strategy which defines which requests
76. \* should be cached.
77. \*/
78. **public** **void** setRequestMatcher(RequestMatcher requestMatcher) {
79. **this**.requestMatcher = requestMatcher;
80. }
82. /\*\*
83. \* If <code>true</code>, indicates that it is permitted to store the target URL and
84. \* exception information in a new <code>HttpSession</code> (the default). In
85. \* situations where you do not wish to unnecessarily create <code>HttpSession</code>s
86. \* - because the user agent will know the failed URL, such as with BASIC or Digest
87. \* authentication - you may wish to set this property to <code>false</code>.
88. \*/
89. **public** **void** setCreateSessionAllowed(**boolean** createSessionAllowed) {
90. **this**.createSessionAllowed = createSessionAllowed;
91. }
93. **public** **void** setPortResolver(PortResolver portResolver) {
94. **this**.portResolver = portResolver;
95. }
96. }

可以看出原请求信息时实质上是被缓存到session中了，缓存的是HttpSessionRequestCache实例，任意HttpSessionRequestCache实例均可获得缓存的原请求信息，只要请求的session没有变化。  
  
2）ExceptionTranslationFilter  
  
ExceptionTranslationFilter 是Spring Security的核心filter之一，用来处理AuthenticationException和AccessDeniedException两种异常（由FilterSecurityInterceptor认证请求返回的异常）。  
  
在我们的例子中，AuthenticationException指的是未登录状态下访问受保护资源，AccessDeniedException指的是登陆了但是由于权限不足(比如普通用户访问管理员界面）。  
  
ExceptionTranslationFilter 持有两个处理类，分别是AuthenticationEntryPoint和AccessDeniedHandler。  
  
ExceptionTranslationFilter 对异常的处理是通过这两个处理类实现的，处理规则很简单：

**[html]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. 规则1. 如果异常是 AuthenticationException，使用 AuthenticationEntryPoint 处理
2. 规则2. 如果异常是 AccessDeniedException 且用户是匿名用户，使用 AuthenticationEntryPoint 处理
3. 规则3. 如果异常是 AccessDeniedException 且用户不是匿名用户，如果否则交给 AccessDeniedHandler 处理。

对应以下代码

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **private** **void** handleSpringSecurityException(HttpServletRequest request,
2. HttpServletResponse response, FilterChain chain, RuntimeException exception)
3. **throws** IOException, ServletException {
4. **if** (exception **instanceof** AuthenticationException) {
5. logger.debug(
6. "Authentication exception occurred; redirecting to authentication entry point",
7. exception);
9. sendStartAuthentication(request, response, chain,
10. (AuthenticationException) exception);
11. }
12. **else** **if** (exception **instanceof** AccessDeniedException) {
13. **if** (authenticationTrustResolver.isAnonymous(SecurityContextHolder
14. .getContext().getAuthentication())) {
15. logger.debug(
16. "Access is denied (user is anonymous); redirecting to authentication entry point",
17. exception);
19. sendStartAuthentication(
20. request,
21. response,
22. chain,
23. **new** InsufficientAuthenticationException(
24. "Full authentication is required to access this resource"));
25. }
26. **else** {
27. logger.debug(
28. "Access is denied (user is not anonymous); delegating to AccessDeniedHandler",
29. exception);
31. accessDeniedHandler.handle(request, response,
32. (AccessDeniedException) exception);
33. }
34. }
35. }

AccessDeniedHandler 默认实现是 AccessDeniedHandlerImpl。该类对异常的处理是返回403错误码。

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **public** **void** handle(HttpServletRequest request, HttpServletResponse response,
2. AccessDeniedException accessDeniedException) **throws** IOException,
3. ServletException {
4. **if** (!response.isCommitted()) {
5. **if** (errorPage != **null**) { // 定义了errorPage
6. // errorPage中可以操作该异常
7. request.setAttribute(WebAttributes.ACCESS\_DENIED\_403,
8. accessDeniedException);
10. // 设置403状态码
11. response.setStatus(HttpServletResponse.SC\_FORBIDDEN);
13. // 转发到errorPage
14. RequestDispatcher dispatcher = request.getRequestDispatcher(errorPage);
15. dispatcher.forward(request, response);
16. }
17. **else** { // 没有定义errorPage，则返回403状态码(Forbidden)，以及错误信息
18. response.sendError(HttpServletResponse.SC\_FORBIDDEN,
19. accessDeniedException.getMessage());
20. }
21. }
22. }

AuthenticationEntryPoint 如果不配置<http>标签的entry-point-ref属性，则默认实现是 LoginUrlAuthenticationEntryPoint, 如果配置了entry-point-ref则用配置的。

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **protected** **void** sendStartAuthentication(HttpServletRequest request,
2. HttpServletResponse response, FilterChain chain,
3. AuthenticationException reason) **throws** ServletException, IOException {
4. // SEC-112: Clear the SecurityContextHolder's Authentication, as the
5. // existing Authentication is no longer considered valid
6. SecurityContextHolder.getContext().setAuthentication(**null**);
7. requestCache.saveRequest(request, response);                   //缓存原请求
8. logger.debug("Calling Authentication entry point.");
9. authenticationEntryPoint.commence(request, response, reason);
10. }

LoginUflAuthenticationEntryPoint该类的处理是转发或重定向到登录页面

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **public** **void** commence(HttpServletRequest request, HttpServletResponse response,
2. AuthenticationException authException) **throws** IOException, ServletException {
4. String redirectUrl = **null**;
6. **if** (useForward) {
8. **if** (forceHttps && "http".equals(request.getScheme())) {
9. // First redirect the current request to HTTPS.
10. // When that request is received, the forward to the login page will be
11. // used.
12. redirectUrl = buildHttpsRedirectUrlForRequest(request);
13. }
15. **if** (redirectUrl == **null**) {
16. String loginForm = determineUrlToUseForThisRequest(request, response,
17. authException);
19. **if** (logger.isDebugEnabled()) {
20. logger.debug("Server side forward to: " + loginForm);
21. }
23. RequestDispatcher dispatcher = request.getRequestDispatcher(loginForm);
25. // 转发
26. dispatcher.forward(request, response);
28. **return**;
29. }
30. }
31. **else** {
32. // redirect to login page. Use https if forceHttps true
34. redirectUrl = buildRedirectUrlToLoginPage(request, response, authException);
36. }
38. // 重定向
39. redirectStrategy.sendRedirect(request, response, redirectUrl);
40. }

了解完这些，回到我们的例子。  
  
第3步时，用户未登录的情况下访问受保护资源，ExceptionTranslationFilter会捕获到AuthenticationException异常(规则1)。页面需要跳转，ExceptionTranslationFilter在跳转前使用requestCache缓存request。

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **protected** **void** sendStartAuthentication(HttpServletRequest request,
2. HttpServletResponse response, FilterChain chain,
3. AuthenticationException reason) **throws** ServletException, IOException {
4. // SEC-112: Clear the SecurityContextHolder's Authentication, as the
5. // existing Authentication is no longer considered valid
6. SecurityContextHolder.getContext().setAuthentication(**null**);
7. // 缓存 request
8. requestCache.saveRequest(request, response);
9. logger.debug("Calling Authentication entry point.");
10. authenticationEntryPoint.commence(request, response, reason);
11. }

requestCache使用的是HttpSessionRequestCache

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. /\*\*
2. \* Stores the current request, provided the configuration properties allow it.
3. \*/
4. **public** **void** saveRequest(HttpServletRequest request, HttpServletResponse response) {
5. **if** (requestMatcher.matches(request)) {
6. DefaultSavedRequest savedRequest = **new** DefaultSavedRequest(request,
7. portResolver);
9. **if** (createSessionAllowed || request.getSession(**false**) != **null**) {
10. // Store the HTTP request itself. Used by
11. // AbstractAuthenticationProcessingFilter
12. // for redirection after successful authentication (SEC-29)
13. request.getSession().setAttribute(SAVED\_REQUEST, savedRequest);
14. logger.debug("DefaultSavedRequest added to Session: " + savedRequest);
15. }
16. }
17. **else** {
18. logger.debug("Request not saved as configured RequestMatcher did not match");
19. }
20. }

总结，在跳转前进行缓存，是缓存到session中。

**四、了解了以上原理以及forward和redirect的区别**[forward和redirect的区别](http://blog.csdn.net/honghailiang888/article/details/52679176)**，配置实现如下，基于springsecurity4.1.3版本**

配置文件：完整的

**[html]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
3. **<beans:beans** xmlns="http://www.springframework.org/schema/security"
4. xmlns:beans="http://www.springframework.org/schema/beans"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans.xsd
8. http://www.springframework.org/schema/security
9. http://www.springframework.org/schema/security/spring-security.xsd"**>**
11. **<http** auto-config="true" use-expressions="true" entry-point-ref="myLoginUrlAuthenticationEntryPoint"**>**
12. **<form-login**
13. login-page="/login"
14. authentication-failure-url="/login?error"
15. login-processing-url="/login"
16. authentication-success-handler-ref="myAuthenticationSuccessHandler" **/>**
17. <!-- 认证成功用自定义类myAuthenticationSuccessHandler处理 -->
19. **<logout** logout-url="/logout"
20. logout-success-url="/"
21. invalidate-session="true"
22. delete-cookies="JSESSIONID"**/>**
24. **<csrf** disabled="true" **/>**
25. **<intercept-url** pattern="/order/\*" access="hasRole('ROLE\_USER')"**/>**
26. **</http>**
28. <!-- 使用自定义类myUserDetailsService从数据库获取用户信息 -->
29. **<authentication-manager>**
30. **<authentication-provider** user-service-ref="myUserDetailsService"**>**
31. <!-- 加密 -->
32. **<password-encoder** hash="md5"**>**
33. **</password-encoder>**
34. **</authentication-provider>**
35. **</authentication-manager>**
37. <!-- 被认证请求向登录界面跳转采用forward方式 -->
38. **<beans:bean** id="myLoginUrlAuthenticationEntryPoint"
39. class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint"**>**
40. **<beans:constructor-arg** name="loginFormUrl" value="/login"**></beans:constructor-arg>**
41. **<beans:property** name="useForward" value="true"**/>**
42. **</beans:bean>**
44. **</beans:beans>**

1）向登录界面跳转：主要配置

**[html]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **<http** auto-config="true" use-expressions="true" entry-point-ref="myLoginUrlAuthenticationEntryPoint"**>**
3. <!-- 被认证请求向登录界面跳转采用forward方式 -->
4. **<beans:bean** id="myLoginUrlAuthenticationEntryPoint"
5. class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint"**>**
6. **<beans:constructor-arg** name="loginFormUrl" value="/login"**></beans:constructor-arg>**
7. **<beans:property** name="useForward" value="true"**/>**
8. **</beans:bean>**

从上面的分析可知，默认情况下采用的是redirect方式，这里通过配置从而实现了forward方式，这里还是直接利用的security自带的类LoginUrlAuthenticationEntryPoint（当然也可以用户自定义了类，下一篇说明如何根据自定义了类实现向不同登录页面的跳转），只不过进行了以上配置：

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. /\*\*
2. \* Performs the redirect (or forward) to the login form URL.
3. \*/
4. **public** **void** commence(HttpServletRequest request, HttpServletResponse response,
5. AuthenticationException authException) **throws** IOException, ServletException {
7. String redirectUrl = **null**;
9. **if** (useForward) {
11. **if** (forceHttps && "http".equals(request.getScheme())) {
12. // First redirect the current request to HTTPS.
13. // When that request is received, the forward to the login page will be
14. // used.
15. redirectUrl = buildHttpsRedirectUrlForRequest(request);
16. }
18. **if** (redirectUrl == **null**) {
19. String loginForm = determineUrlToUseForThisRequest(request, response,
20. authException);
22. **if** (logger.isDebugEnabled()) {
23. logger.debug("Server side forward to: " + loginForm);
24. }
26. RequestDispatcher dispatcher = request.getRequestDispatcher(loginForm);
28. dispatcher.forward(request, response);
30. **return**;
31. }
32. }
33. **else** {
34. // redirect to login page. Use https if forceHttps true
36. redirectUrl = buildRedirectUrlToLoginPage(request, response, authException);
38. }
40. redirectStrategy.sendRedirect(request, response, redirectUrl);
41. }

2）登录信息提交后认证流程

跳转到登录界面，提交登录信息后，经过过滤器UsernamePasswordAuthenticationFilter，该过滤器继承了AbstractAuthenticationProcessingFilter，

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **public** **void** doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
2. **throws** IOException, ServletException {
4. HttpServletRequest request = (HttpServletRequest) req;
5. HttpServletResponse response = (HttpServletResponse) res;
7. **if** (!requiresAuthentication(request, response)) {
8. chain.doFilter(request, response);
10. **return**;
11. }
13. **if** (logger.isDebugEnabled()) {
14. logger.debug("Request is to process authentication");
15. }
17. Authentication authResult;
19. **try** {
20. authResult = attemptAuthentication(request, response);  //获取认证结果，其中包括从数据源中获取用户数据与登录时填写的信息比较，包括是否有该用户，密码是否正确，否则抛异常
21. **if** (authResult == **null**) {
22. // return immediately as subclass has indicated that it hasn't completed
23. // authentication
24. **return**;
25. }
26. sessionStrategy.onAuthentication(authResult, request, response);
27. }
28. **catch** (InternalAuthenticationServiceException failed) {
29. logger.error(
30. "An internal error occurred while trying to authenticate the user.",
31. failed);
32. unsuccessfulAuthentication(request, response, failed);
34. **return**;
35. }
36. **catch** (AuthenticationException failed) {
37. // Authentication failed
38. unsuccessfulAuthentication(request, response, failed);
40. **return**;
41. }
43. // Authentication success
44. **if** (continueChainBeforeSuccessfulAuthentication) {
45. chain.doFilter(request, response);
46. }
48. successfulAuthentication(request, response, chain, authResult);   //认证成功后调用
49. }

在以上程序中包含了从数据源中获取用户信息并和用户填写的信息进行对比的过程，功能实现attemptAuthentication，这里不对其进行详细分析。

判断是否认证成功，认证成功后执行如下代码：

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. **protected** **void** successfulAuthentication(HttpServletRequest request,
2. HttpServletResponse response, FilterChain chain, Authentication authResult)
3. **throws** IOException, ServletException {
5. **if** (logger.isDebugEnabled()) {
6. logger.debug("Authentication success. Updating SecurityContextHolder to contain: "
7. + authResult);
8. }
10. SecurityContextHolder.getContext().setAuthentication(authResult);
12. rememberMeServices.loginSuccess(request, response, authResult);
14. // Fire event
15. **if** (**this**.eventPublisher != **null**) {
16. eventPublisher.publishEvent(**new** InteractiveAuthenticationSuccessEvent(
17. authResult, **this**.getClass()));
18. }
20. successHandler.onAuthenticationSuccess(request, response, authResult);
21. }

其中successHandler就是配置的MyAuthenticationSuccessHandler。  
登录成功后的类配置，存入登录user信息后交给认证成功后的处理类MyAuthenticationSuccessHandler，该类继承了SavedRequestAwareAuthenticationSuccessHandler，他会从缓存中提取请求，从而可以恢复之前请求的数据。初次之外还可以通过配置自定义类实现认证成功后根据权限跳转到不同的页面，例如用户中心和后台管理中心，下一篇会详细说明。

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. /\*\*
2. \* 登录后操作
3. \*
4. \* @author HHL
5. \* @date
6. \*
7. \*/
8. @Component
9. **public** **class** MyAuthenticationSuccessHandler **extends**
10. SavedRequestAwareAuthenticationSuccessHandler {
12. @Autowired
13. **private** IUserService userService;
15. @Override
16. **public** **void** onAuthenticationSuccess(HttpServletRequest request,
17. HttpServletResponse response, Authentication authentication)
18. **throws** IOException, ServletException {
20. // 认证成功后，获取用户信息并添加到session中
21. UserDetails userDetails = (UserDetails) authentication.getPrincipal();
22. MangoUser user = userService.getUserByName(userDetails.getUsername());
23. request.getSession().setAttribute("user", user);
25. **super**.onAuthenticationSuccess(request, response, authentication);
27. }

30. }

SavedRequestAwareAuthenticationSuccessHandler中的onAuthenticationSuccess方法;

**[java]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. @Override
2. **public** **void** onAuthenticationSuccess(HttpServletRequest request,
3. HttpServletResponse response, Authentication authentication)
4. **throws** ServletException, IOException {
5. SavedRequest savedRequest = requestCache.getRequest(request, response);
7. **if** (savedRequest == **null**) {
8. **super**.onAuthenticationSuccess(request, response, authentication);
10. **return**;
11. }
12. String targetUrlParameter = getTargetUrlParameter();
13. **if** (isAlwaysUseDefaultTargetUrl()
14. || (targetUrlParameter != **null** && StringUtils.hasText(request
15. .getParameter(targetUrlParameter)))) {
16. requestCache.removeRequest(request, response);
17. **super**.onAuthenticationSuccess(request, response, authentication);
19. **return**;
20. }
22. clearAuthenticationAttributes(request);
24. // Use the DefaultSavedRequest URL
25. String targetUrl = savedRequest.getRedirectUrl();
26. logger.debug("Redirecting to DefaultSavedRequest Url: " + targetUrl);
27. getRedirectStrategy().sendRedirect(request, response, targetUrl);
28. }

4.1.3中如果配置了**authentication-success-handler-ref，则首先使用该配置的，如果配置了authentication-success-forward-url，则使用该配置的，如果都没有配置则**采用的SavedRequestAwareAuthenticationSuccessHandler进行处理，详情可参见：[Spring实战篇系列----源码解析Spring Security中的过滤器Filter初始化](http://blog.csdn.net/honghailiang888/article/details/53541664)  
  
  
上述实现了跳转到登录界面采用forward方式，就是浏览器地址栏没有变化，当然也可采用redirect方式，地址栏变为登录界面地址栏，当登录完成后恢复到原先的请求页面，请求信息会从requestCache中还原回来。可参考[Spring实战篇系列----spring security4.1.3配置以及踩过的坑](http://blog.csdn.net/honghailiang888/article/details/53520557)﻿﻿

五、附上信息恢复的调用栈

**[html]** [view plain](http://blog.csdn.net/honghailiang888/article/details/53671108) [copy](http://blog.csdn.net/honghailiang888/article/details/53671108)

1. WebUtils.getParametersStartingWith(ServletRequest, String) line: 663
2. ServletRequestParameterPropertyValues.**<init>**(ServletRequest, String, String) line: 77
3. ServletRequestParameterPropertyValues.**<init>**(ServletRequest) line: 52
4. ExtendedServletRequestDataBinder(ServletRequestDataBinder).bind(ServletRequest) line: 100
5. ServletModelAttributeMethodProcessor.bindRequestParameters(WebDataBinder, NativeWebRequest) line: 150
6. ServletModelAttributeMethodProcessor(ModelAttributeMethodProcessor).resolveArgument(MethodParameter, ModelAndViewContainer, NativeWebRequest, WebDataBinderFactory) line: 114
7. HandlerMethodArgumentResolverComposite.resolveArgument(MethodParameter, ModelAndViewContainer, NativeWebRequest, WebDataBinderFactory) line: 121
8. ServletInvocableHandlerMethod(InvocableHandlerMethod).getMethodArgumentValues(NativeWebRequest, ModelAndViewContainer, Object...) line: 161
9. ServletInvocableHandlerMethod(InvocableHandlerMethod).invokeForRequest(NativeWebRequest, ModelAndViewContainer, Object...) line: 128
10. ServletInvocableHandlerMethod.invokeAndHandle(ServletWebRequest, ModelAndViewContainer, Object...) line: 114
11. RequestMappingHandlerAdapter.invokeHandlerMethod(HttpServletRequest, HttpServletResponse, HandlerMethod) line: 827
12. RequestMappingHandlerAdapter.handleInternal(HttpServletRequest, HttpServletResponse, HandlerMethod) line: 738
13. RequestMappingHandlerAdapter(AbstractHandlerMethodAdapter).handle(HttpServletRequest, HttpServletResponse, Object) line: 85
14. DispatcherServlet.doDispatch(HttpServletRequest, HttpServletResponse) line: 963
15. DispatcherServlet.doService(HttpServletRequest, HttpServletResponse) line: 897
16. DispatcherServlet(FrameworkServlet).processRequest(HttpServletRequest, HttpServletResponse) line: 970
17. DispatcherServlet(FrameworkServlet).doPost(HttpServletRequest, HttpServletResponse) line: 872
18. DispatcherServlet(HttpServlet).service(HttpServletRequest, HttpServletResponse) line: 647
19. DispatcherServlet(FrameworkServlet).service(HttpServletRequest, HttpServletResponse) line: 846
20. DispatcherServlet(HttpServlet).service(ServletRequest, ServletResponse) line: 728
21. ApplicationFilterChain.internalDoFilter(ServletRequest, ServletResponse) line: 303
22. ApplicationFilterChain.doFilter(ServletRequest, ServletResponse) line: 208
23. WsFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 91
24. ApplicationFilterChain.internalDoFilter(ServletRequest, ServletResponse) line: 241
25. ApplicationFilterChain.doFilter(ServletRequest, ServletResponse) line: 208
26. SiteMeshFilter(ContentBufferingFilter).bufferAndPostProcess(FilterChain, HttpServletRequest, HttpServletResponse, Selector) line: 169
27. SiteMeshFilter(ContentBufferingFilter).doFilter(ServletRequest, ServletResponse, FilterChain) line: 126
28. SiteMeshFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 120
29. ConfigurableSiteMeshFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 163
30. ApplicationFilterChain.internalDoFilter(ServletRequest, ServletResponse) line: 241
31. ApplicationFilterChain.doFilter(ServletRequest, ServletResponse) line: 208
32. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 317
33. FilterSecurityInterceptor.invoke(FilterInvocation) line: 127
34. FilterSecurityInterceptor.doFilter(ServletRequest, ServletResponse, FilterChain) line: 91
35. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
36. ExceptionTranslationFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 115
37. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
38. SessionManagementFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 137
39. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
40. AnonymousAuthenticationFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 111
41. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
42. SecurityContextHolderAwareRequestFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 169
43. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
44. RequestCacheAwareFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 63
45. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
46. BasicAuthenticationFilter.doFilterInternal(HttpServletRequest, HttpServletResponse, FilterChain) line: 158
47. BasicAuthenticationFilter(OncePerRequestFilter).doFilter(ServletRequest, ServletResponse, FilterChain) line: 107
48. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
49. UsernamePasswordAuthenticationFilter(AbstractAuthenticationProcessingFilter).doFilter(ServletRequest, ServletResponse, FilterChain) line: 200
50. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
51. LogoutFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 121
52. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
53. HeaderWriterFilter.doFilterInternal(HttpServletRequest, HttpServletResponse, FilterChain) line: 66
54. HeaderWriterFilter(OncePerRequestFilter).doFilter(ServletRequest, ServletResponse, FilterChain) line: 107
55. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
56. WebAsyncManagerIntegrationFilter.doFilterInternal(HttpServletRequest, HttpServletResponse, FilterChain) line: 56
57. WebAsyncManagerIntegrationFilter(OncePerRequestFilter).doFilter(ServletRequest, ServletResponse, FilterChain) line: 107
58. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
59. SecurityContextPersistenceFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 105
60. FilterChainProxy$VirtualFilterChain.doFilter(ServletRequest, ServletResponse) line: 331
61. FilterChainProxy.doFilterInternal(ServletRequest, ServletResponse, FilterChain) line: 214
62. FilterChainProxy.doFilter(ServletRequest, ServletResponse, FilterChain) line: 177
63. DelegatingFilterProxy.invokeDelegate(Filter, ServletRequest, ServletResponse, FilterChain) line: 346
64. DelegatingFilterProxy.doFilter(ServletRequest, ServletResponse, FilterChain) line: 262
65. ApplicationFilterChain.internalDoFilter(ServletRequest, ServletResponse) line: 241
66. ApplicationFilterChain.doFilter(ServletRequest, ServletResponse) line: 208
67. CharacterEncodingFilter.doFilterInternal(HttpServletRequest, HttpServletResponse, FilterChain) line: 197
68. CharacterEncodingFilter(OncePerRequestFilter).doFilter(ServletRequest, ServletResponse, FilterChain) line: 107
69. ApplicationFilterChain.internalDoFilter(ServletRequest, ServletResponse) line: 241
70. ApplicationFilterChain.doFilter(ServletRequest, ServletResponse) line: 208
71. StandardWrapperValve.invoke(Request, Response) line: 223
72. StandardContextValve.invoke(Request, Response) line: 107
73. NonLoginAuthenticator(AuthenticatorBase).invoke(Request, Response) line: 504
74. StandardHostValve.invoke(Request, Response) line: 155
75. ErrorReportValve.invoke(Request, Response) line: 75
76. AccessLogValve.invoke(Request, Response) line: 934
77. StandardEngineValve.invoke(Request, Response) line: 90
78. CoyoteAdapter.service(Request, Response) line: 494
79. Http11AprProcessor(AbstractHttp11Processor**<S>**).process(SocketWrapper**<S>**) line: 1009
80. Http11AprProtocol$Http11ConnectionHandler(AbstractProtocol$AbstractConnectionHandler**<S**,P**>**).process(SocketWrapper**<S>**, SocketStatus) line: 632
81. Http11AprProtocol$Http11ConnectionHandler.process(SocketWrapper**<Long>**, SocketStatus) line: 281
82. AprEndpoint$SocketProcessor.doRun() line: 2248
83. AprEndpoint$SocketProcessor.run() line: 2237
84. ThreadPoolExecutor(ThreadPoolExecutor).runWorker(ThreadPoolExecutor$Worker) line: not available
85. ThreadPoolExecutor$Worker.run() line: not available
86. TaskThread(Thread).run() line: not available

**﻿﻿  
总结：**

1）被认证请求被FilterSecurityInterceptor拦截看有没有对应权限，如果没有抛异常给ExceptionTranslationFilter

2）ExceptionTranslationFilter缓存原请求，利用LoginUrlAuthenticationEntryPoint入口跳转到登录界面

3）用户在登录界面填写登录信息后，提交，经过UsernamePasswordAuthenticationFilter对填写的信息和从数据源中获取的信息进行对比，成功则授权权限，并通过登录成功后入口SavedRequestAwareAuthenticationSuccessHandler跳转回原请求页面（跳转时有从缓存中对请求信息的恢复）

4）登录完成后返回原请求，由FilterSecurityInterceptor进行权限的验证（大部分工作有AbstractSecurityInterceptor来做），根据登录成功后生成的Authentication（Authentication authentication = SecurityContextHolder.getContext().getAuthentication();由SecurityContextHolder持有，而其中的SecurityContext由SecurityContextPersistentFilter保存到session中从而实现request共享）中的权限和请求所需的权限对比，如果一致则成功执行，如果权限不正确则返回403错误码﻿﻿

5）以上均是默认情况下，没有经过配置的执行过程，当然可以自定义LoginUrlAuthenticationEntryPoint和SavedRequestAwareAuthenticationSuccessHandler实现根据不同的请求所需权限跳转到不同登录页面及授权成功后根据权限跳转到不同页面，以及返回403错误码时跳转到对应的页面（AccessDeniedHandlerImpl）在下一篇中会对其进行实现﻿﻿

﻿﻿

﻿﻿

﻿﻿

﻿﻿

﻿﻿

版权声明：本文为博主原创文章，转载请注明出处http://blog.csdn.net/honghailiang888。 http://blog.csdn.net/honghailiang888/article/details/53671108

* 本文已收录于以下专栏：
* [Spring应用](http://blog.csdn.net/column/details/13067.html)