

爆破专栏 | Spring Security系列教程之Spring Security认证授权流程

原创 一一哥 Java架构栈 1周前

收录于话题

#Spring Security爆破专栏

16个 >

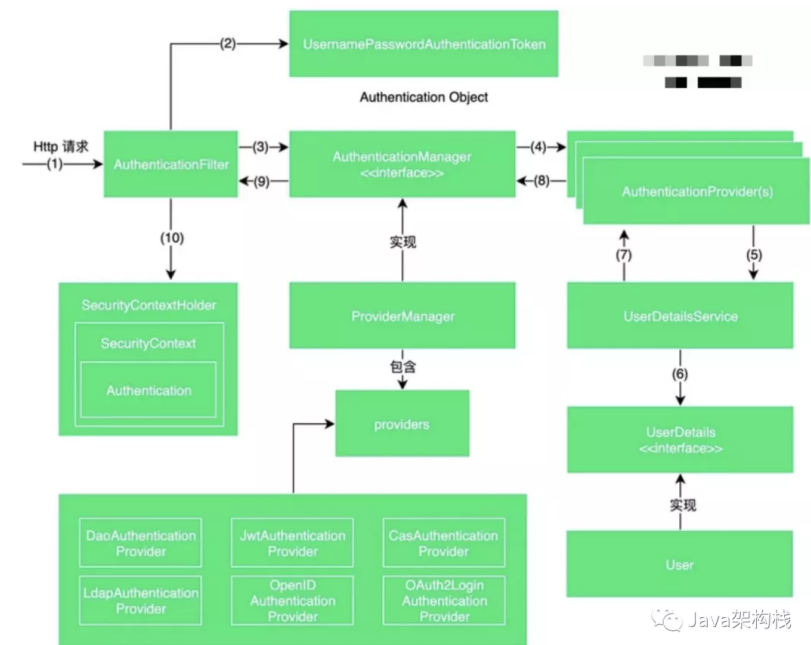
前言

在上一章节中，一一哥 带大家认识了Spring Security内部关于认证授权的几个核心API，以及这几个核心API之间的引用关系，掌握了这些之后，我们就能进一步研究分析认证授权的内部实现原理了。这样才真正的达到了 "知其所以然"！

本篇文章中，壹哥 带各位小伙伴进一步分析认证授权的源码实现，请各位再坚持一下吧.....

一. Spring Security认证授权流程图概述

在上一章节中，壹哥就给各位贴出过Spring Security的认证授权流程图，该图展示了认证授权时经历的核心API，并且展示了认证授权流程。接下来我们结合源码，一点点分析认证和授权的实现过程。



二. 简要剖析认证授权实现流程的代码逻辑

Spring Security的认证授权流程其实是非常复杂的，在我们对源码还不够了解的情况下，壹哥先给各位简要概括一下这个认证和授权流程，大致如下：

1. 用户登录前，默认生成的Authentication对象处于未认证状态，登录时会交由Authentication Manager负责进行认证。
2. AuthenticationManager会将Authentication中的用户名/密码与UserDetails中的用户名/密码对比，完成认证工作，认证成功后会生成一个已认证状态的Authentication对象；
3. 最后把认证通过的Authentication对象写入到SecurityContext中，在用户有后续请求时，可从Authentication中检查权限。

我们可以借鉴一下Spring Security官方文档中提供的一个最简化的认证授权流程代码，来认识一下认证授权的实现过程，该代码省略了UserDetails操作，只做了简单认证，可以对认证授权有个大概了解。

```
public class AuthenticationExample {

    private static AuthenticationManager am = new SampleAuthenticationManager();

    public static void main(String[] args) throws Exception {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        while (true) {
            //模拟输入用户名密码
            System.out.println("Please enter your username:");
            String name = in.readLine();

            System.out.println("Please enter your password:");
            String password = in.readLine();

            try {
                //根据用户名/密码，生成未认证Authentication
                Authentication request = new UsernamePasswordAuthenticationToken(name, password);
                //交给AuthenticationManager 认证
                Authentication result = am.authenticate(request);
                //将已认证的Authentication放入SecurityContext
                SecurityContextHolder.getContext().setAuthentication(result);
                break;
            } catch (AuthenticationException e) {
```

```

        System.out.println("Authentication failed: " + e.getMessage());
    }
}

System.out.println("Successfully authenticated. Security context contains: "
    + SecurityContextHolder.getContext().getAuthentication());
}
}

//认证类
class SampleAuthenticationManager implements AuthenticationManager {
    //配置一个简单的用户权限集合
    static final List<GrantedAuthority> AUTHORITIES = new ArrayList<GrantedAuthority>()

    static {
        AUTHORITIES.add(new SimpleGrantedAuthority("ROLE_USER"));
    }

    public Authentication authenticate(Authentication auth) throws AuthenticationException {
        //如果用户名和密码一致，则登录成功，这里只做了简单认证
        if (auth.getName().equals(auth.getCredentials())) {
            //认证成功，生成已认证Authentication，比未认证多了权限
            return new UsernamePasswordAuthenticationToken(auth.getName(), auth.getCredentials(),
                AUTHORITIES);
        }

        throw new BadCredentialsException("Bad Credentials");
    }
}

```

以上代码只是简单的模拟了认证的过程，那么真正的认证授权操作，是不是也这样呢？接下来请跟着 一一哥，咱们结合源码进行详细的剖析。

三. Spring Security认证流程源码详解

1. Spring Security过滤器链执行顺序

在 Spring Security 中，与认证、授权相关的校验其实都是利用一系列的过滤器来完成的，这些过滤器共同组成了一个过滤器链，如下图所示：



你可能会问，我们怎么知道这些过滤器在执行？其实我们只要开启Spring Security的debug调试模式，开发时就可以在控制台看到这些过滤器的执行顺序，如下：



所以在上图中可以看到，**Spring**

Security中默认执行的过滤器顺序如下：

- WebAsyncManagerIntegrationFilter
- SecurityContextPersistenceFilter
- HeaderWriterFilter
- CsrfFilter

- LogoutFilter
- UsernamePasswordAuthenticationFilter
- DefaultLoginPageGeneratingFilter
- DefaultLogoutPageGeneratingFilter
- RequestCacheAwareFilter
- SecurityContextHolderAwareRequestFilter
- AnonymousAuthenticationFilter: 如果之前的认证机制都没有更新 Security Context Holder 拥有的 Authentication, 那么一个 Anonymous Authentication Token 将会设给 Security Context Holder。
- SessionManagementFilter
- ExceptionTranslationFilter: 用于处理在 Filter Chain 范围内抛出的 Access Denied Exception 和 Authentication Exception, 并把它们转换为对应的 Http 错误码返回或者跳转到对应的页面。
- FilterSecurityInterceptor: 负责保护 Web URI, 并且在访问被拒绝时抛出异常。

2. SecurityContextPersistenceFilter

在上面的过滤器链中, 我们可以看到 Security Context PersistenceFilter这个过滤器。

Security Context Persistence Filter是**Security**中的一个拦截器, 它的执行时机非常早, 当请求来临时它会从**Security Context Repository**中把**SecurityContext**对象取出来, 然后放入**Security Context Holder**的**Thread Local**中。在所有拦截器都处理完成后, 再把**Security Context**存入**Security Context Repository**, 并清除**Security Context Holder** 内的 **Security Context** 引用。

3. AbstractAuthenticationProcessingFilter

在上面图所展示的一系列的过滤器中, 和认证授权直接相关的过滤器是 **Abstract Authentication Processing Filter** 和 **Username Password Authentication Filter**。但是你可能又会问, 怎么没看到 **Abstract Authentication Processing Filter** 这个过滤器呢? 这是因为它是一个抽象的父类, 其内部定义了认证处理的过程, **Username Password Authentication Filter** 就继承自 **Abstract Authentication Processing Filter**。如下图所示:



从上图中我们可以看出，

Abstract Authentication ProcessingFilter的父类是**Generic Filter Bean**，而 **GenericFilterBean** 是 *Spring* 框架中的过滤器类，最终的父接口是**Filter**，也就是我们熟悉的过滤器。那么既然是**Filter**的子类，肯定会执行其中最核心的do Filter()方法，我们来看看**Abstract Authentication Processing Filter**的do Filter()方法源码。



从上面的源码中我们可以看出，doFilter()方法的内部实现并不复杂，里面就只引用了5个方法，如下：

```
requiresAuthentication(request, response);
authResult = attemptAuthentication(request, response);
sessionStrategy.onAuthentication(authResult, request, response);
unsuccessfulAuthentication(request, response, failed);
successfulAuthentication(request, response, chain, authResult);
```

这几个方法具体的作用如下：

1. 首先执行 **requiresAuthentication(HttpServletRequest, HttpServletResponse)** 方法，来决定是否需要验证操作；
2. 如果需要验证，接着就会调用 **attempt Authentication(Http Servlet Request, Http Servlet Response)** 方法来封装用户信息再进行验证，可能会有三种结果产生：
 - a. 如果返回的Authentication对象为Null，则表示身份验证不完整，该方法将立即结束返回。
 - b. 如果返回的Authentication对象不为空，则调用配置的SessionAuthenticationStrategy 对象，执行onAuthentication()方法，然后调用successfulAuthentication(HttpServletRequest, HttpServletResponse, FilterChain, Authentication) 方法。
 - c. 验证时如果发生 AuthenticationException，则执行unsuccessful Authentication (HttpServletRequest, HttpServletResponse, Authentication Exception) 方法。

我们在上面的源码中得知有个**attempt Authentication()**方法，该方法是一个抽象方法，由子类来具体实现。



这个抽象方法由子类**Username Password Authentication Filter**来实现，如下图。



Abstract Authentication

Processing Filter是一个比较复杂的类，内部的处理流程比较多，我们做个简单梳理，如下图所示：



综上所述，我们可知AbstractAuthenticationProcessingFilter类，可以负责处理所有的HTTP Request和Response对象，并将其封装成AuthenticationMananger可以处理的Authentication对象。在身份验证成功或失败之后，将对应的行为转换为HTTP的Response对象。同时还能处理一些Web特有的资源，比如Session和Cookie等操作。到此为止，我们已经把Abstract Authentication Processing Filter这个类的核心功能给了解清楚了，接下来我们学习Abstract Authentication Processing Filter的子类Username Password Authentication Filter。

4. UsernamePasswordAuthenticationFilter

我在上一小节中说过，在 Spring Security 中，认证与授权的相关校验是在Abstract Authentication Processing Filter这个过滤器中完成的，但是该类是一个抽象类，它有个子类Username Password Authentication Filter，该类是和认证直接相关的过滤器实现子类。我们来看一下该类的核心源码：

```
public class UsernamePasswordAuthenticationFilter extends
    AbstractAuthenticationProcessingFilter {

    public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "username";
    public static final String SPRING_SECURITY_FORM_PASSWORD_KEY = "password";

    ...
}
```



```

...

// ~ Constructors
// =====

public UsernamePasswordAuthenticationFilter() {
    super(new AntPathRequestMatcher("/login", "POST"));
}

public Authentication attemptAuthentication(HttpServletRequest request,
    HttpServletResponse response) throws AuthenticationException {
    if (postOnly && !request.getMethod().equals("POST")) {
        throw new AuthenticationServiceException(
            "Authentication method not supported: " + request.getMethod());
    }

    String username = obtainUsername(request);
    String password = obtainPassword(request);

    if (username == null) {
        username = "";
    }

    if (password == null) {
        password = "";
    }

    username = username.trim();

    UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthentic
        username, password);

    // Allow subclasses to set the "details" property
    setDetails(request, authRequest);

    return this.getAuthenticationManager().authenticate(authRequest);
}

@Nullable
protected String obtainPassword(HttpServletRequest request) {
    return request.getParameter(passwordParameter);
}

```

```

@Nullable
protected String obtainUsername(HttpServletRequest request) {
    return request.getParameter(usernameParameter);
}

protected void setDetails(HttpServletRequest request,
    UsernamePasswordAuthenticationToken authRequest) {
    authRequest.setDetails(authenticationDetailsSource.buildDetails(request));
}

.....其他略.....
}

```

我来解释一下上面的源码：

1. 首先我们从构造方法中可以得知，该过滤器 只对post请求方式的"/login"接口有效；
2. 然后在该过滤器中，再利用 **obtainUsername** 和 **obtainPassword** 方法，提取出请求里边的用户名/密码，提取方式就是 `request.getParameter`，这也是为什么 Spring Security 中默认的表单登录要通过 key/value 的形式传递参数，而不能传递 JSON 参数。如果像传递 JSON 参数，我们可以通过修改这里的代码来进行实现。
3. 获取到请求里传递来的用户名/密码之后，接下来会 构造一个 **Username Password AuthenticationToken** 对象，传入 **username** 和 **password**。其中 **username** 对应了 `UsernamePasswordAuthenticationToken` 中的 **principal** 属性，而 **password** 则对应了它的 **credentials** 属性。
4. 接下来 再利用 **setDetails** 方法给 **details** 属性赋值，`UsernamePasswordAuthenticationToken` 本身是没有 **details** 属性的，这个属性是在它的父类 `AbstractAuthenticationToken` 中定义的。**details** 是一个对象，这个对象里边存放的是 `Web Authentication Details` 实例，该实例主要描述了 请求的 **remote Address** 以及请求的 **sessionId** 这两个信息。
5. 最后一步，就是利用 **AuthenticationManager** 对象来调用 **authenticate()** 方法去做认证校验。

5. AuthenticationManager与ProviderManager

咱们在上面 `Username Password Authentication Token` 类的 **attempt Authentication()** 方法中得知，该方法的最后一步会进行关于认证的校验，而要进行认证操作首先要获取到一个 **Authentication Manager** 对象，这里默认拿到的是 **Authentication Manager** 的子类 **Provider Manager**，如下图所示：



所以接下来我们要进入到

ProviderManager 的 `authenticate()`方法中，来看看认证到底是怎么实现的。因为这个方法的实现代码比较长，我这里仅摘列出几个重要的地方：





ProviderManager类中的

authenticate()方法代码很长，我通过截图，把该方法中的重点地方做了红色标记，方便大家重点记忆。

其实Spring Security中关于认证的重要逻辑几乎都是在这里完成的，所以接下来我分步骤，一点点带大家分析该认证的实现流程。

1. 首先利用反射，获取到要认证的 authentication 对象的 Class字节码，如下图：



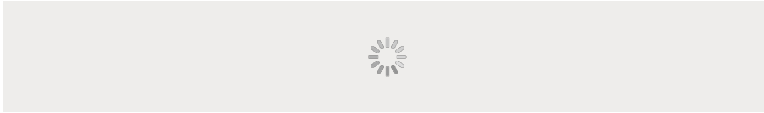
2. 判断当前 provider 是否支持该 authentication 对象，如下图：



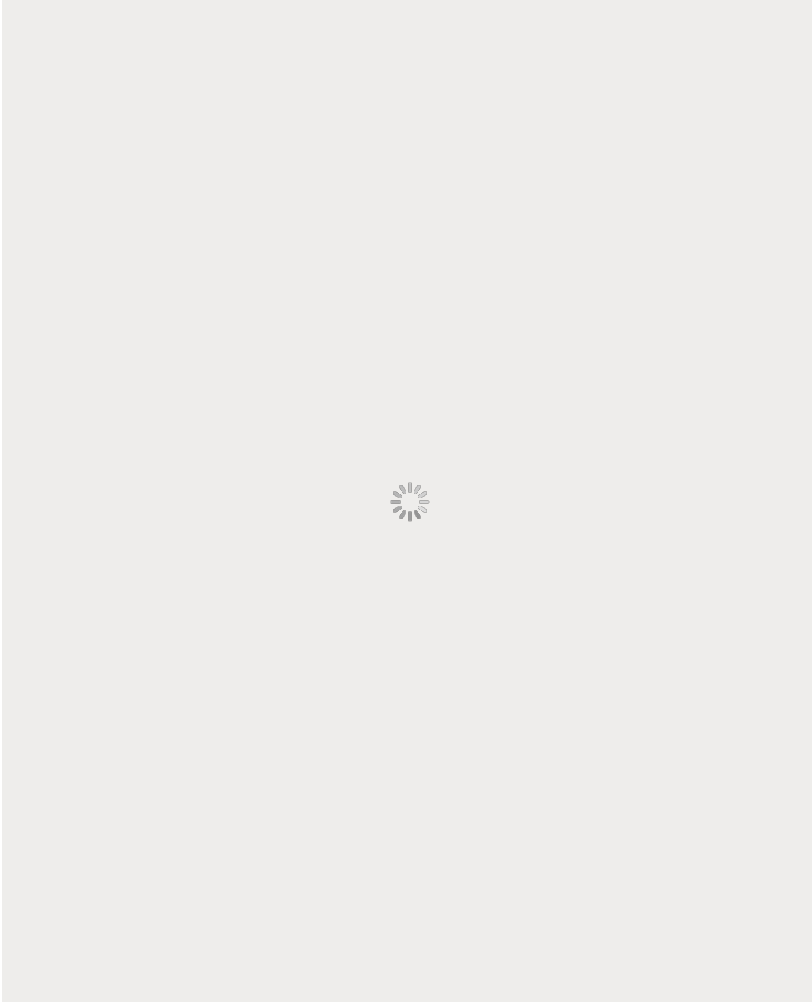
如果当前provider不支持该

authentication 对象，则退出当前判断，进行下一次判断。

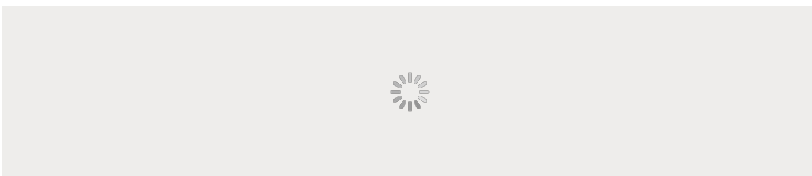
3. 如果支持，则调用 provider 的 authenticate 方法开始做校验，校验完成后，会返回一个新的 Authentication，如下图：



4. 这里的 provider 会有多个，我们在上一章节给大家介绍过，如下图：



这里如果 provider 的 authenticate 方法没能返回一个 Authentication 认证对象，则会调用 provider 的 parent 对象中的 authenticate 方法继续校验。



5. 而如果通过了校验，返回了一个 Authentication 认证对象，则调用 copyDetails() 方法把旧 Token 的 details 属性拷贝到新的 Token 中，如下图。



6. 接下来会调用 `eraseCredentials()` 方法来擦除凭证信息，也就是我们的密码，这个擦除方法比较简单，就是将 `Token` 中的 `credentials` 属性置空。



7. 最后通过 `publishAuthenticationSuccess()` 方法将认证成功的事件广播出去。



在以上代码的for循环中，第一次拿到的 `provider` 是一个 **Anonymous Authentication Provider**。这个 `provider` 是不支持 **Username Password Authentication Token** 的，所以会直接在 `provider.supports()` 方法中返回 `false`，结束当前for循环，并进入到下一个 `if` 判断中，最后直接调用 `parent` 的 `authenticate` 方法进行校验。

而 `parent` 就是 `ProviderManager` 对象，所以会再次回到这个 `authenticate()` 方法中。当再次回到 `authenticate()` 方法时，`provider` 会变成第二个 `Provider`，即 **Dao Authentication Provider**，这个 `provider` 是支持

UsernamePasswordAuthenticationToken 的，所以会顺利进入到该类的authenticate()方法去执行。

6. DaoAuthenticationProvider

DaoAuthenticationProvider继承自**Abstract UserDetails Authentication Provider**，DaoAuthenticationProvider类结构如下所示：



DaoAuthenticationProvider类

中并没有重写 **authenticate()** 方法，**authenticate()** 方法是在父类 **AbstractUserDetailsAuthenticationProvider**中实现的。所以我们看看Abstract User Details Authentication Provider # authenticate()方法的源码，这里我对该源码做了一些简化：

```
public Authentication authenticate(Authentication authentication)
    throws AuthenticationException {
    .....

    //获取authentication中存储的用户名
    String username = (authentication.getPrincipal() == null) ? "NONE_PROVIDED"
        : authentication.getName();

    //判断是否使用了缓存
    boolean cacheWasUsed = true;
    UserDetails user = this.userCache.getUserFromCache(username);

    if (user == null) {
        cacheWasUsed = false;

        try {
            //retrieveUser()是一个抽象方法，由子类DaoAuthenticationProvider来实现，
            user = retrieveUser(username,
                (UsernamePasswordAuthenticationToken) authentication);
        }
    }
}
```



```

        catch (UsernameNotFoundException notFound) {
            .....
        }

        try {
            //进行必要的认证前和额外认证的检查
            preAuthenticationChecks.check(user);

            //这是抽象方法，由子类DaoAuthenticationProvider来实现，用于进行密码对比
            additionalAuthenticationChecks(user,
                (UsernamePasswordAuthenticationToken) authentication);
        }
        catch (AuthenticationException exception) {
            if (cacheWasUsed) {
                //在发生异常时，尝试着从缓存中进行对象的加载
                cacheWasUsed = false;
                user = retrieveUser(username,
                    (UsernamePasswordAuthenticationToken) authentication);
                preAuthenticationChecks.check(user);
                additionalAuthenticationChecks(user,
                    (UsernamePasswordAuthenticationToken) authentication);
            }
            else {
                throw exception;
            }
        }

        //认证后的检查操作
        postAuthenticationChecks.check(user);

        if (!cacheWasUsed) {
            this.userCache.putUserInCache(user);
        }

        Object principalToReturn = user;

        if (forcePrincipalAsString) {
            principalToReturn = user.getUsername();
        }

        //认证成功后，封装认证对象
        return createSuccessAuthentication(principalToReturn, authentication, user);
    }

```

结合上面的源码，我们再做进一步的分析梳理：

1. 我在上面说过，**DaoAuthenticationProvider**这个子类并没有重写**authenticate()**方法，而是在父类**AbstractUserDetailsAuthenticationProvider**中实现的。**Abstract User Details Authentication Provider** 类中的 **authenticate()**方法执行时，首先会从 **Authentication** 提取出登录用户名，如下图所示：



2. 然后利用得到的 **username**，先去缓存中查询是否有该用户，如下所示：



3. 如果缓存中没有该用户，则去执行 **retrieveUser()** 方法获取当前用户对象。而这个 **retrieveUser()**方法是个抽象方法，在**Abstract User Details Authentication Provider**类中并没有实现，是由子类**DaoAuthenticationProvider**来实现的。



4. 在**DaoAuthenticationProvider**类的**retrieveUser()** 方法中，会调用**get User Details Service()**方法，得到**UserDetailsService**对象，执行我们自己在登录时候编写的 **loadUserByUsername()**方法，然后返回一个**UserDetails**对象，也就是我们的登录对象。如下图所示。



5. 接下来会继续往下执行**preAuthenticationChecks.check()**方法，检验 **user** 中各账户属性是否正常，例如账户是否被禁用、是否被锁定、是否过期等，如下所示。



6. 接着会继续往下执行**additionalAuthenticationChecks()**方法，进行密码比对。而该方法也是抽象方法，也是由子类**DaoAuthenticationProvider**进行实现。我们在注册用户时对密码加密之后，**Spring Security**就是在这里进行密码比对的。如下所示。



7. 然后在 **postAuthenticationChecks.check()**方法中检查密码是否过期，如下所示。



3. 然后判断是否进行了缓存，如果未进行缓存，则执行缓存操作，这个缓存是由 `SpringCacheBasedUserCache` 类来实现的。

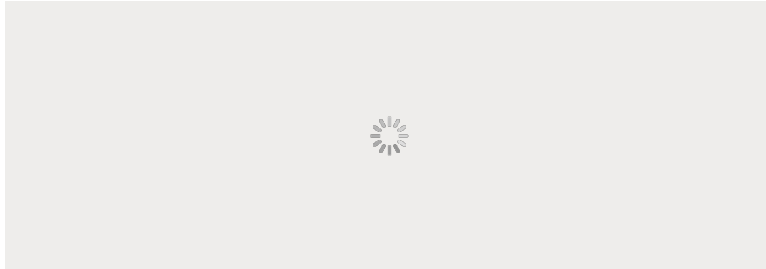


我们这里如果没有对缓存做配置，则会执行默认的缓存配置操作。如果我们对缓存进行了自定义的配置，比如配置了 `RedisCache`，就可以把对象缓存到redis中。

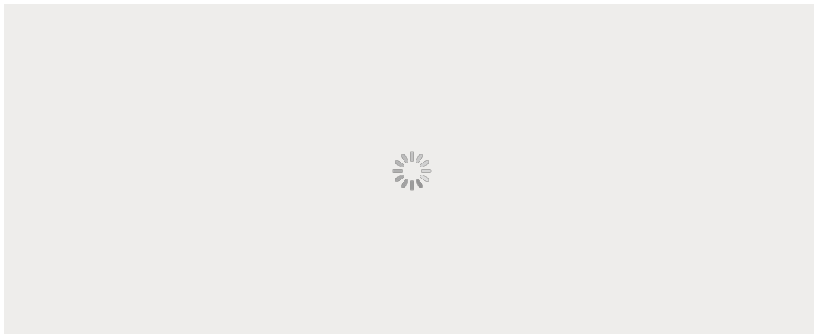


9. 接下来有一个 `forcePrincipalAsString` 属性，该属性表示 是否强制将

Authentication 中的 **principal** 属性设置为字符串，这个属性其实我们一开始就在 **UsernamePasswordAuthenticationFilter** 类中定义为了字符串(即username)。但是默认情况下，当用户登录成功之后，这个属性的值就变成当前用户这个对象了。之所以会这样，就是因为 **forcePrincipalAsString** 默认为 **false**，不过这块其实不用改，就用 **false**，这样在后期获取当前用户信息的时候反而方便很多。



0. 最后通过**createSuccessAuthentication()**方法构建出一个新的 **Username Password Authentication Token**对象。



1. 这样我们最终得到了认证通过的**Authentication**对象，并把该对象利用**publish Authentication Success()**方法，将该事件发布出去。



2. Spring Security会监听这个事件，接收到这个Authentication对象，进而调用 `SecurityContextHolder.getContext().setAuthentication(...)` 方法，将 Authentication Manager返回的 Authentication对象，存储在当前的 Security Context 对象中。

7. 保存Authentication认证信息

我们在上面说，**Authentication**认证信息最终是保存在**SecurityContext** 对象中的，但是具体的代码是在哪里实现的呢？

我们来到 `UsernamePasswordAuthenticationFilter` 的父类 `Abstract Authentication ProcessingFilter` 中，这个类我们经常会见到，因为很多时候当我们想要在 Spring Security 自定义一个登录验证码或者将登录参数改为 JSON 的时候，我们都需自定义过滤器继承自 `AbstractAuthenticationProcessingFilter`。

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
    throws IOException, ServletException {

    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;
```

```
if (!requiresAuthentication(request, response)) {
    chain.doFilter(request, response);

    return;
}

.....

Authentication authResult;

try {
    authResult = attemptAuthentication(request, response);
    if (authResult == null) {
        // return immediately as subclass has indicated that it hasn't completed
        // authentication
        return;
    }
    sessionStrategy.onAuthentication(authResult, request, response);
}

catch (InternalAuthenticationServiceException failed) {
    logger.error(
        "An internal error occurred while trying to authenticate the user.",
        failed);
    unsuccessfulAuthentication(request, response, failed);

    return;
}

catch (AuthenticationException failed) {
    // Authentication failed
    unsuccessfulAuthentication(request, response, failed);

    return;
}

// Authentication success
if (continueChainBeforeSuccessfulAuthentication) {
    chain.doFilter(request, response);
}
```

```
}

    //处理认证后的操作
    successfulAuthentication(request, response, chain, authResult);
}
```

在上面的源码中，有个successfulAuthentication()方法，如下图。



当

UsernamePasswordAuthenticationFilter#attemptAuthentication()方法被触发执行时，如果登录时抛出异常， unsuccessfulAuthentication()方法会被调用；而当登录成功时， **successfulAuthentication()**方法则会被调用，正是这个方法来保存的Authentication认证信息，我们来看看这个源码。



在这里有一段很重要的代码，就是

SecurityContextHolder.getContext().setAuthentication(authResult),登录成功的用户信息就被保存在这里。在认证成功后，我们就可以在任何地方，通过 SecurityContextHolder.getContext()获取到Authentication认证信息。

最后大家还看到还有一个 **successHandler.onAuthenticationSuccess()**方法，这是我们在 **SecurityConfig**中配置的登录成功时的回调处理方法，就是在这里被触发。

8. ExceptionTranslationFilter

Spring Security在进行认证授权的过程中，可能会产生各种认证授权异常。对于这些异常，都是由**ExceptionTranslationFilter**来捕获过滤器链中产生的所有异常并进行处理的。但是它只会处理两类异常：**AuthenticationException** 和 **Access Denied Exception**，其它的异常它会继续抛出。

如果捕获到的是 **AuthenticationException**异常，那么将会使用其对应的 **Authentication EntryPoint** 里的**commence()**方法来处理。在处理之前，**Exception Translation Filter**先使用 **RequestCache** 将当前的**Http Servlet Request**的信息保存起来，以至于用户认证登录成功后可以跳转到之前指定跳转到的界面。

如果捕获到的是 **AccessDeniedException**异常，那么将根据当前用户是否已经登录认证做出不同的处理。如果未登录，则会使用关联的 **AuthenticationEntryPoint** 的 **commence()**方法来进行处理；否则将会使用关联的 **AccessDeniedHandler** 的**handle()**方法来进行处理。

9. FilterSecurityInterceptor

当我们经历了前面一系列的认证授权处理后，最后还有一个**FilterSecurityInterceptor**用于保护**Http**资源，它内部引用了一个**AccessDecisionManager**和一个**Authentication Manager**对象。它会从 **SecurityContextHolder** 获取 **Authentication**对象，然后通过 **SecurityMetadataSource** 可以得知当前是否在请求受保护的资源。如果请求的是那些受保护的资源，如果**Authentication.isAuthenticated()** 返回**false**或者 **FilterSecurityInterceptor**的**alwaysReauthenticate** 属性为 **true**，那么将会使用其引用的 **AuthenticationManager** 再认证一次。认证之后再使用认证后的 **Authentication** 替换 **SecurityContextHolder** 中拥有的旧的那个**Authentication**对象，然后就是利用 **AccessDecisionManager** 进行权限的检查。

注：

- **AuthenticationEntryPoint** 是在用户未登录时，用于引导用户进行登录认证的；
- **AccessDeniedHandler** 是在用户已经登录后，但是访问了自身没有权限的资源时做出的对应处理。

10. 认证流程总结

Spring Security进行认证授权的源码执行流程，大致就是上面我婆媳剖析的那么，接下来我对认证过程做个简单总结：

1. 首先用户在登录表单中，填写用户名和密码，进行登录操作；

2. AbstractAuthenticationProcessingFilter结合

UsernamePasswordAuthenticationToken过滤器，将获取到的用户名和密码封装成一个实现了 Authentication 接口的实现子类 UsernamePasswordAuthenticationToken 对象。

3. 将上述产生的 token 对象传递给 AuthenticationManager的具体子类 ProviderManager 进行登录认证。

4. ProviderManager 认证成功后将会返回一个封装了用户权限等信息的 Authentication 对象。

5. 通过调用 SecurityContextHolder.getContext().setAuthentication(...) 将 AuthenticationManager 返回的 Authentication 对象赋予给当前的 Security Context。

我们可以结合下面两图，和上面的源码，深入理解掌握Spring Security的认证授权流程。





以后出去面试时，给面试官讲解这个简化流程就差不多了。

四. 相关面试题

另外可能我们有这么一个疑问：如何在 **request** 之间共享 **SecurityContext**？

既然 **SecurityContext** 是存放在 **ThreadLocal** 中的，而且在每次权限鉴定的时候，都是从 **ThreadLocal** 中获取 **SecurityContext** 中保存的 **Authentication**。那么既然不同的 **request** 属于不同的线程，为什么每次都可以从 **ThreadLocal** 中获取到当前用户对应的 **SecurityContext** 呢？

- 在 Web 应用中这是通过 **SecurityContextPersistentFilter** 实现的，默认情况下其在每次请求开始的时候，都会从 session 中获取 **SecurityContext**，然后把它设置给 **SecurityContextHolder**。
- 在请求结束后又会将 **SecurityContextHolder** 所持有的 **SecurityContext** 保存在 session 中，并且清除 **SecurityContextHolder** 所持有的 **SecurityContext**。
- 这样当我们第一次访问系统的时候，**SecurityContextHolder** 所持有的 **SecurityContext** 肯定是空的。待我们登录成功后，**SecurityContextHolder** 所持有的 **SecurityContext** 就不是空的了，且包含有认证成功的 **Authentication** 对象。
- 待请求结束后我们就会将 **SecurityContext** 存在 session 中，等到下次请求的时候就可以从 session 中获取到该 **SecurityContext** 并把它赋予给 **Security Context Holder** 了。
- 由于 **SecurityContextHolder** 已经持有认证过的 **Authentication** 对象了，所以下次访问的时候也就不再需要进行登录认证了。

到此为止，我就给大家剖析了 Spring Security 中最核心的认证授权源码，也就是底层执行原理，如果你可以把这些源码理解掌握了，出去面试时，就靠这一个知识点，就足以征服

面试官，让面试官臣服在你的“牛逼”之下。

你学会了吗？评论区留言666呗！

往期精彩

[Spring Security系列教程之创建项目\(1\)](#)

[Spring Security系列教程之实现HTTP基本认证\(2\)](#)

[Spring Security系列教程之实现Form表单认证\(3\)](#)

[Spring Security系列之实现HTTP摘要认证\(4\)](#)

[Spring Security系列之前后端分离时的安全处理方案\(5\)](#)

[Spring Security系列教程之基于内存模型实现授权\(6\)](#)

[Spring Security系列教程之基于默认数据库模型实现授权\(7\)](#)

[Spring Security系列教之基于自定义数据库模型实现授权\(8\)](#)

文末福利

留言评论，今天的内容你学会了吗？

点赞最多的同学获得水杯一个！



关注公众号，回复【**ss**】获取专栏大纲脑图

点击【**阅读原文**】，从零开始学Java那段文字

收录于话题 #Spring Security爆破专栏·16个 >

[< 上一篇](#)

爆破专栏 | Spring Security系列教程之
基于过滤器实现图形验证码

[下一篇 >](#)

爆破专栏 | 系列教程之Spring Security
核心API讲解

[阅读原文](#)

喜欢此内容的人还喜欢

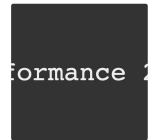
Android NativeCrash 捕获与解析

字节流动



写在 2021 的前端性能优化指南

前端桃园



微服务架构服务限流方案详解

架构之家

