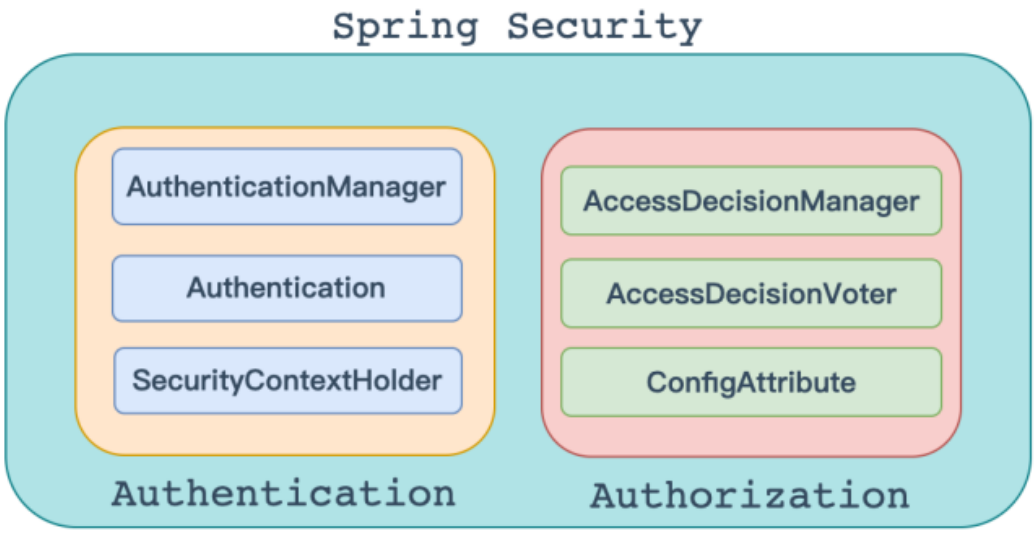
SpringSecurity

整体架构

在SpringSecurity的架构中，认证跟授权是分开的，无论采用什么样的认证方式，都不会影响授权，这是两个独立的存在。



认证

https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#servlet-authentication-authenticationmanager

AuthenticationManager

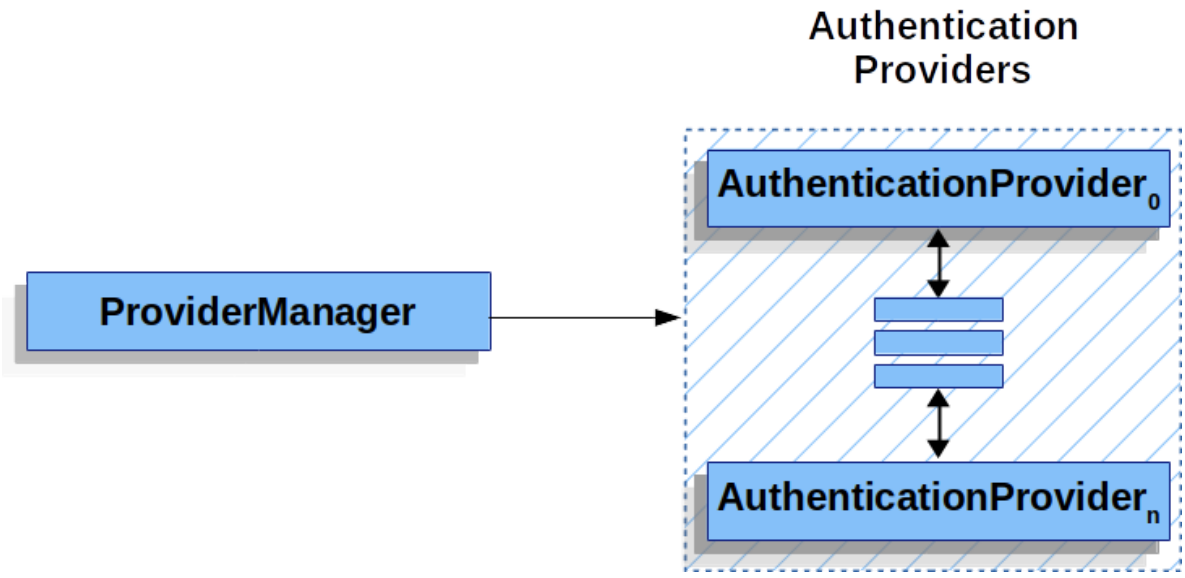
在 Spring Security中，认证是由该接口来负责的

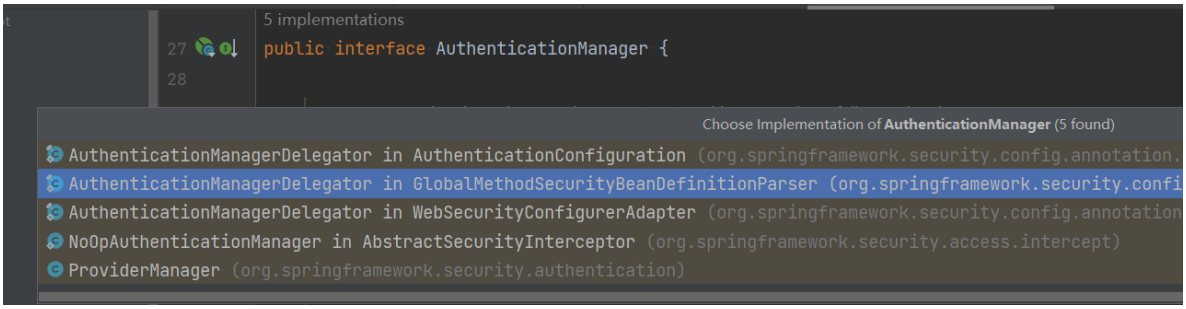
> `AuthenticationManager` is the API that defines how Spring Security's Filters perform authentication. The `Authentication` that is returned is then set on the SecurityContextHolder by the controller (that is, by Spring Security's `Filters` instances) that invoked the `AuthenticationManager`. If you are not integrating with Spring Security's `Filters` instances, you can set the `SecurityContextHolder` directly and are not required to use an `AuthenticationManager`.
>
> While the implementation of `AuthenticationManager` could be anything, the most common implementation is `ProviderManager`.

```java
public interface AuthenticationManager {

    /**
     * 返回 Authentication 表示认证成功
     * 抛出 AuthenticationException 表示认证失败
     */
   Authentication authenticate(Authentication authentication) throws AuthenticationException;

}
```

ProviderManager

> `ProviderManager` is the most commonly used implementation of `AuthenticationManager`. `ProviderManager` delegates to a `List` of `AuthenticationProvider` instances. Each `AuthenticationProvider` has an opportunity to indicate that authentication should be successful, fail, or indicate it cannot make a decision and allow a downstream `AuthenticationProvider` to decide. If none of the configured `AuthenticationProvider` instances can authenticate, authentication fails with a `ProviderNotFoundException`, which is a special `AuthenticationException` that indicates that the `ProviderManager` was not configured to support the type of `Authentication` that was passed into it.

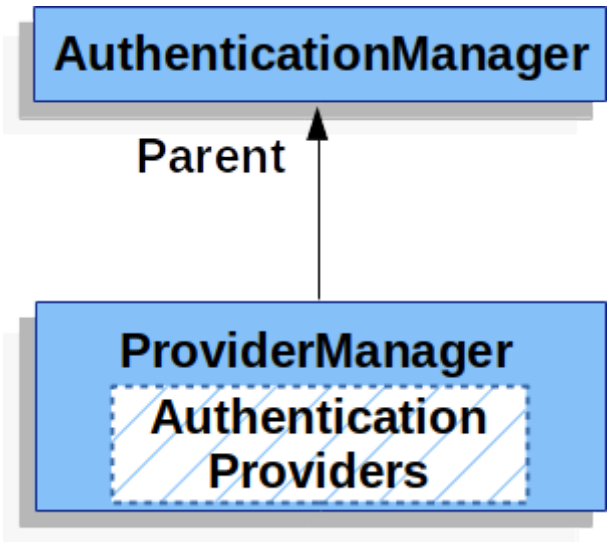




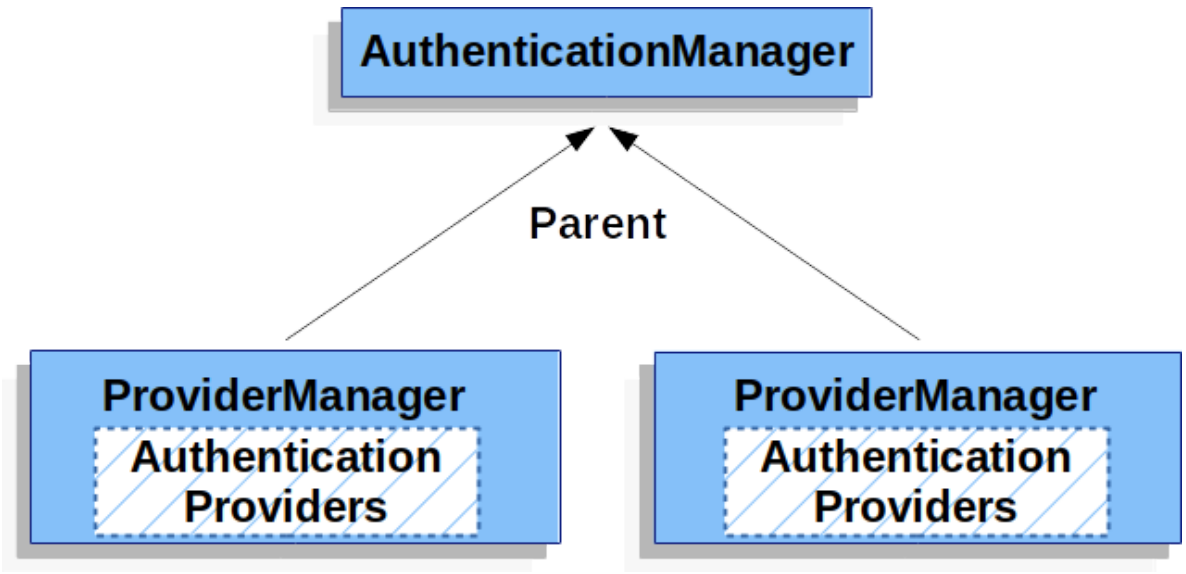
> In practice each `AuthenticationProvider` knows how to perform a specific type of authentication. For example, one `AuthenticationProvider` might be able to validate a username/password, while another might be able to authenticate a SAML assertion. This lets each `AuthenticationProvider` do a very specific type of authentication while

> supporting multiple types of authentication and expose only a single `AuthenticationManager` bean.

> ProviderManager `also allows configuring an optional parent` AuthenticationManager `, which is consulted in the event that no` AuthenticationProvider `can perform authentication. The parent can be any type of` AuthenticationManager `, but it is often an instance of` ProviderManager



> In fact, multiple `ProviderManager` instances might share the same parent `AuthenticationManager`. This is somewhat common in scenarios where there are multiple `SecurityFilterChain` instances that have some authentication in common (the shared parent `AuthenticationManager`), but also different authentication mechanisms (the different `ProviderManager` instances).



AuthenticationProvider

> You can inject multiple `AuthenticationProvider`s instances into `ProviderManager`. Each `AuthenticationProvider` performs a specific type of authentication. For example, `DaoAuthenticationProvider` supports username/password-based authentication, while `JwtAuthenticationProvider` supports authenticating a JWT token.

AuthenticationManager 主要实现类为 ProviderManager，在 ProviderManager中管理了众多 AuthenticationProvider 实例。在一次完整的认证流程中， Spring Security 允许存在多个 AuthenticationProvider ，用来实现多种认证方式，这些 AuthenticationProvider 都是由 ProviderManager 进行统一管理的。

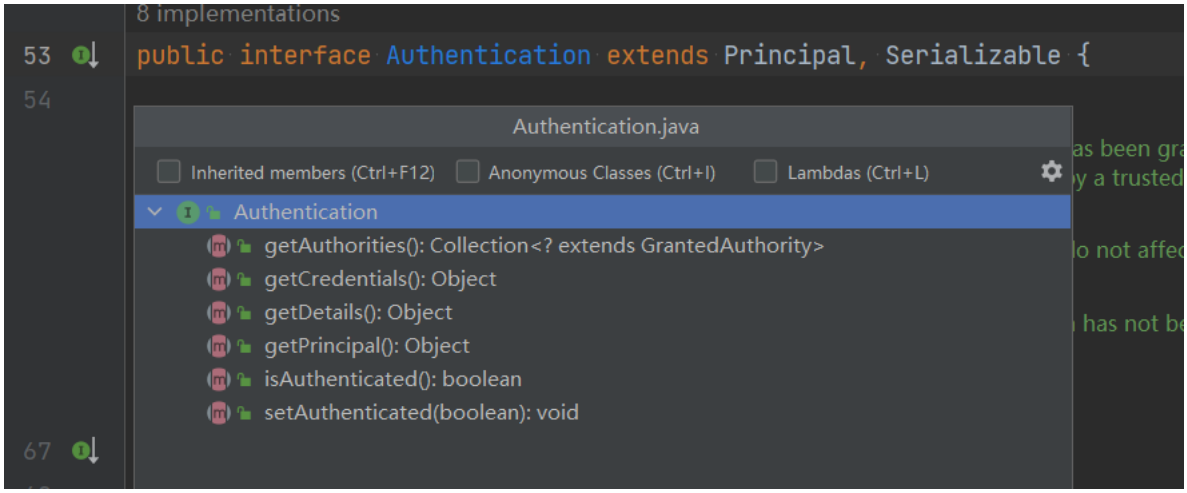
Authentication

The `Authentication` interface serves two main purposes within Spring Security:

- An input to `AuthenticationManager` to provide the credentials a user has provided to authenticate. When used in this scenario, `isAuthenticated()` returns `false`.
- Represent the currently authenticated user. You can obtain the current `Authentication` from the SecurityContext.

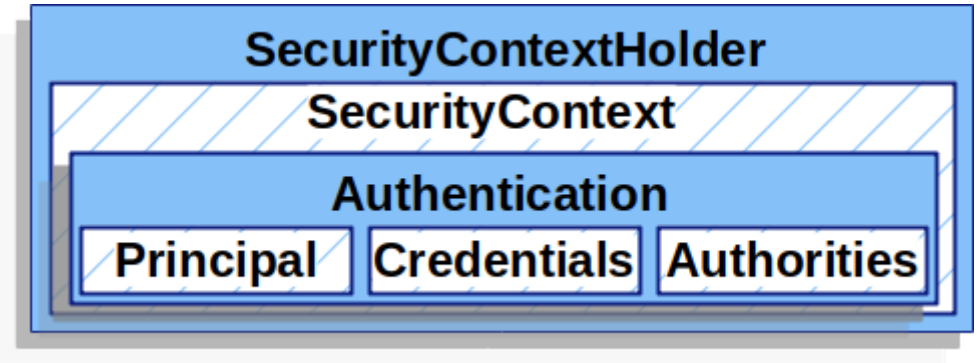
The `Authentication` contains:

- `principal`: Identifies the user. When authenticating with a username/password this is often an instance of `UserDetails`.
- `credentials`: Often a password. In many cases, this is cleared after the user is authenticated, to ensure that it is not leaked.
- `authorities`: The `GrantedAuthority` instances are high-level permissions the user is granted. Two examples are roles and scopes.



SecurityContextHolder

> The `SecurityContext` is obtained from the SecurityContextHolder. The `SecurityContext` contains an Authentication object.
>
> The `SecurityContextHolder` is where Spring Security stores the details of who is authenticated. Spring Security does not care how the `SecurityContextHolder` is populated. If it contains a value, it is used as the currently authenticated user.



> By default, `SecurityContextHolder` uses a `ThreadLocal` to store these details, which means that the `SecurityContext` is always available to methods in the same thread, even if the `SecurityContext` is not explicitly passed around as an argument to those methods. Using a `ThreadLocal` in this way is quite safe if you take care to clear the thread after the present principal's request is processed. Spring Security's FilterChainProxy ensures that the `SecurityContext` is always cleared.

SecurityContextHolder 用来获取登录之后用户信息。 Spring Security 会将登录用户数据保存在 Session 中。但是，为了使用方便,Spring Security在此基础上还做了一些改进，其中最主要的一个变化就是线程绑定。当用户登录成功后,Spring Security 会将登录成功的用户信息保存到 SecurityContextHolder 中。

SecurityContextHolder 中的数据保存默认是通过ThreadLocal 来实现的，使用ThreadLocal 创建的变量只能被当前线程访问，不能被其他线程访问和修改，也就是用户数据和请求线程绑定在一起。当登录请求处理完毕后， Spring Security 会将SecurityContextHolder 中的数据拿出来保存到 Session 中，同时将SecurityContexHolder 中的数据清空。以后每当有请求到来时， Spring Security就会先从 Session 中取出用户登录数据，保存到 SecurityContextHolder 中，方便在该请求的后续处理过程中使用，同时在请求结束时将SecurityContextHolder 中的数 据拿出来保存到 Session 中，然后将 Security SecurityContextHolder 中的数据清空。这一策略非常方便用户在 Controller、 Service 层以及任何代码中获取当前登录用户数据。

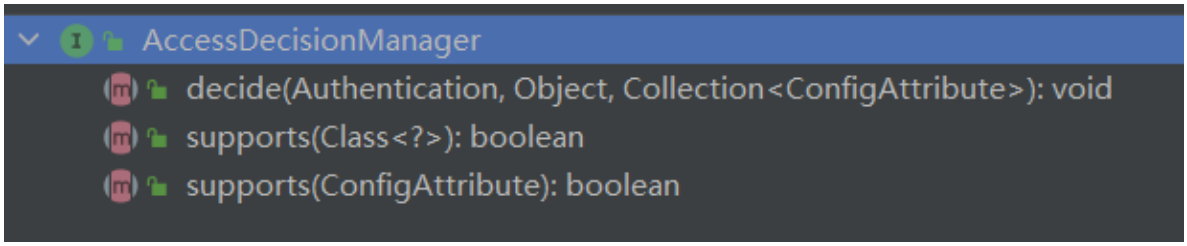
授权

https://docs.spring.io/spring-security/reference/servlet/authorization/architecture.html

AccessDecisionManager

AccessDecisionManager (访问决策管理器)，用来决定此次访问是否被允许

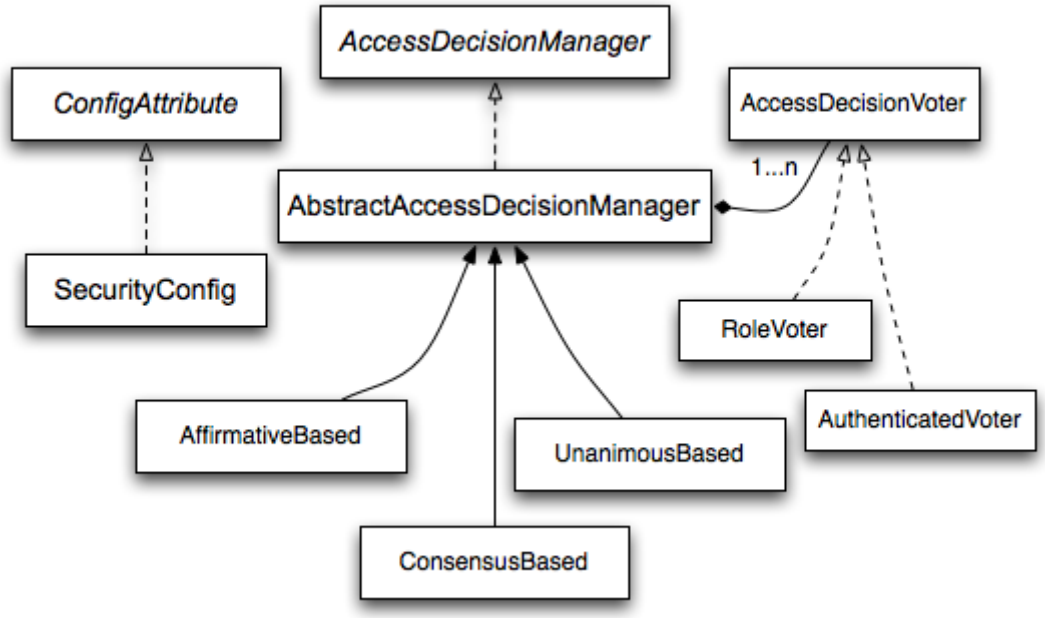
> The `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` and is responsible for making final access control decisions.



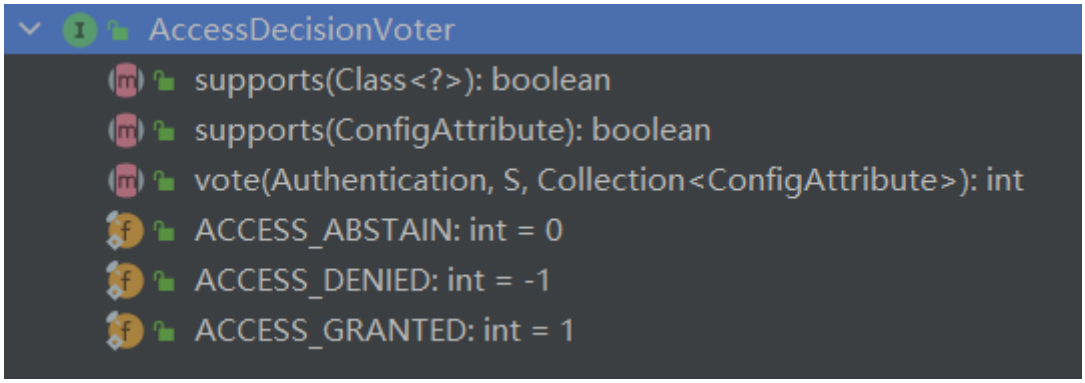
> The `decide` method of the `AccessDecisionManager` is passed all the relevant information it needs to make an authorization decision. In particular, passing the secure `Object` lets those arguments contained in the actual secure object invocation be inspected. For example, assume the secure object is a `MethodInvocation`. You can query the `MethodInvocation` for any `Customer` argument and then implement some sort of security logic in the `AccessDecisionManager` to ensure the principal is permitted to operate on that customer. Implementations are expected to throw an `AccessDeniedException` if access is denied.
>
> The `supports(ConfigAttribute)` method is called by the `AbstractSecurityInterceptor` at startup time to determine if the `AccessDecisionManager` can process the passed `ConfigAttribute`. The `supports(Class)` method is called by a security interceptor implementation to ensure the configured `AccessDecisionManager` supports the type of secure object that the security interceptor presents.

AccessDecisionVoter

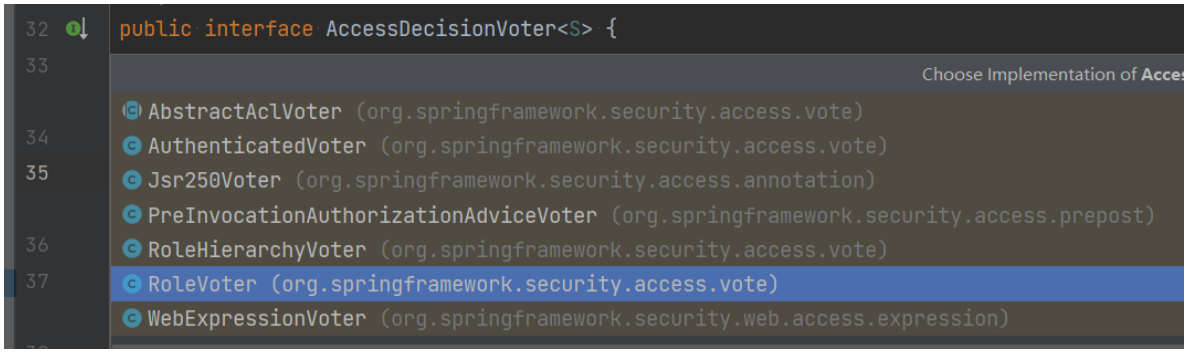
AccessDecisionVoter (访问决定投票器)，投票器会检查用户是否具备应有的角色，进而投出赞成、反对或者弃权票



> While users can implement their own `AccessDecisionManager` to control all aspects of authorization, Spring Security includes several `AccessDecisionManager` implementations that are based on voting. Voting Decision Manager describes the relevant classes.
>
> By using this approach, a series of `AccessDecisionVoter` implementations are polled on an authorization decision. The `AccessDecisionManager` then decides whether or not to throw an `AccessDeniedException` based on its assessment of the votes.



> Concrete implementations return an `int`, with possible values being reflected in the `AccessDecisionVoter` static fields named `ACCESS_ABSTAIN`, `ACCESS_DENIED` and `ACCESS_GRANTED`. A voting implementation returns `ACCESS_ABSTAIN` if it has no opinion on an authorization decision. If it does have an opinion, it must return either `ACCESS_DENIED` or `ACCESS_GRANTED`.
>
> There are three concrete `AccessDecisionManager` implementations provided with Spring Security to tally the votes. The `ConsensusBased` implementation grants or denies access based on the consensus of non-abstain votes. Properties are provided to control behavior in the event of an equality of votes or if all votes are abstain. The `AffirmativeBased` implementation grants access if one or more `ACCESS_GRANTED` votes were received (in other words, a deny vote will be ignored, provided there was at least one grant vote). Like the `ConsensusBased` implementation, there is a parameter that controls the behavior if all voters abstain. The `UnanimousBased` provider expects unanimous `ACCESS_GRANTED` votes in order to grant access, ignoring abstains. It denies access if there is any `ACCESS_DENIED` vote. Like the other implementations, there is a parameter that controls the behavior if all voters abstain.
>
> You can implement a custom `AccessDecisionManager` that tallies votes differently. For example, votes from a particular `AccessDecisionVoter` might receive additional weighting, while a deny vote from a particular voter may have a veto effect.



AccesDecisionVoter 和 AccessDecisionManager 都有众多的实现类，在AccessDecisionManager 中会换个遍历 AccessDecisionVoter，进而决定是否允许用户访问，因而 AaccesDecisionVoter 和 AccessDecisionManager 两者的关系类似于 AuthenticationProvider 和 ProviderManager 的关系。

RoleVoter

> The most commonly used `AccessDecisionVoter` provided with Spring Security is the `RoleVoter`, which treats configuration attributes as role names and votes to grant access if the user has been assigned that role.

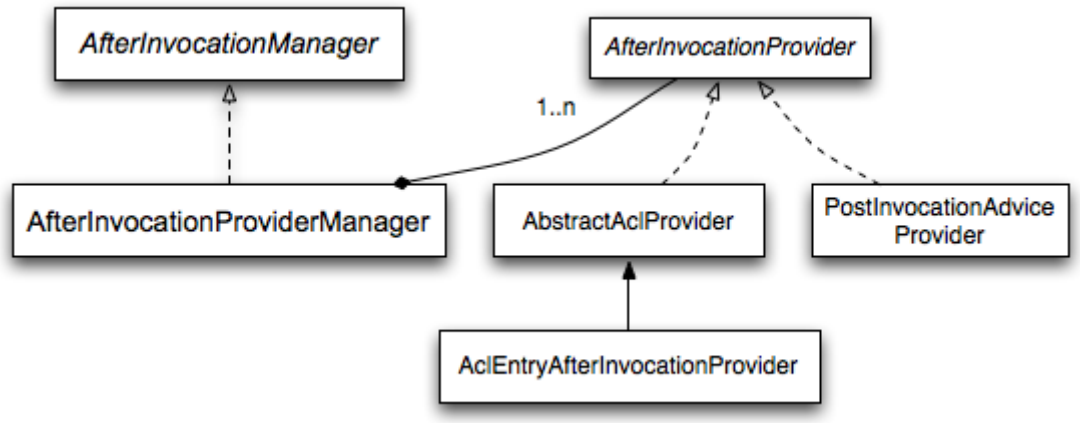
> It votes if any `ConfigAttribute` begins with the `ROLE_` prefix. It votes to grant access if there is a `GrantedAuthority` that returns a `String` representation (from the `getAuthority()` method) exactly equal to one or more `ConfigAttributes` that start with the `ROLE_` prefix. If there is no exact match of any `ConfigAttribute` starting with `ROLE_`, `RoleVoter` votes to deny access. If no `ConfigAttribute` begins with `ROLE_`, the voter abstains.

AuthenticatedVoter

> Another voter which we have implicitly seen is the `AuthenticatedVoter`, which can be used to differentiate between anonymous, fully-authenticated, and remember-me authenticated users. Many sites allow certain limited access under remember-me authentication but require a user to confirm their identity by logging in for full access.
>
> When we have used the `IS_AUTHENTICATED_ANONYMOUSLY` attribute to grant anonymous access, this attribute was being processed by the `AuthenticatedVoter`. For more information, see `AuthenticatedVoter`.

Custom Voters

> You can also implement a custom `AccessDecisionVoter` and put just about any access-control logic you want in it. It might be specific to your application (business-logic related) or it might implement some security administration logic. For example, on the Spring web site, you can find a blog article that describes how to use a voter to deny access in real-time to users whose accounts have been suspended.

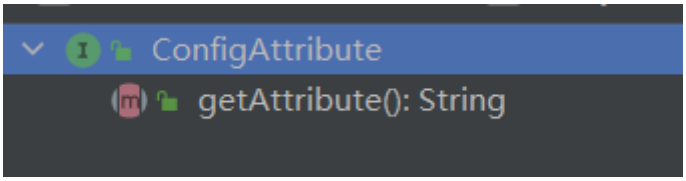


> Like many other parts of Spring Security, `AfterInvocationManager` has a single concrete implementation, `AfterInvocationProviderManager`, which polls a list of `AfterInvocationProvider`s. Each `AfterInvocationProvider` is allowed to modify the return object or throw an `AccessDeniedException`. Indeed multiple providers can modify the object, as the result of the previous provider is passed to the next in the list.
>
> Please be aware that if you're using `AfterInvocationManager`, you will still need configuration attributes that allow the `MethodSecurityInterceptor`'s `AccessDecisionManager` to allow an operation. If you're using the typical Spring Security included `AccessDecisionManager` implementations, having no configuration attributes defined for a particular secure method invocation will cause each `AccessDecisionVoter` to abstain from voting. In turn, if the `AccessDecisionManager` property

> “allowIfAllAbstainDecisions” is `false`, an `AccessDeniedException` will be thrown. You may avoid this potential issue by either (i) setting “allowIfAllAbstainDecisions” to `true` (although this is generally not recommended) or (ii) simply ensure that there is at least one configuration attribute that an `AccessDecisionVoter` will vote to grant access for. This latter (recommended) approach is usually achieved through a `ROLE_USER` or `ROLE_AUTHENTICATED` configuration attribute.

ConfigAttribute

ConfigAttribute，用来保存授权时的角色信息



在 Spring Security 中，用户请求一个资源(通常是一个接口或者一个 Java 方法)需要的角色会被封装成一个 ConfigAttribute 对象，在 ConfigAttribute 中只有一个getAttribute方法，该方法返回一个 String 字符串，就是角色的名称。一般来说，角色名称都带有一个 ROLE_ 前缀，投票器 AccessDecisionVoter 所做的事情，其实就是比较用户所具各的角色和请求某个资源所需的 ConfigAtuibute 之间的关系。

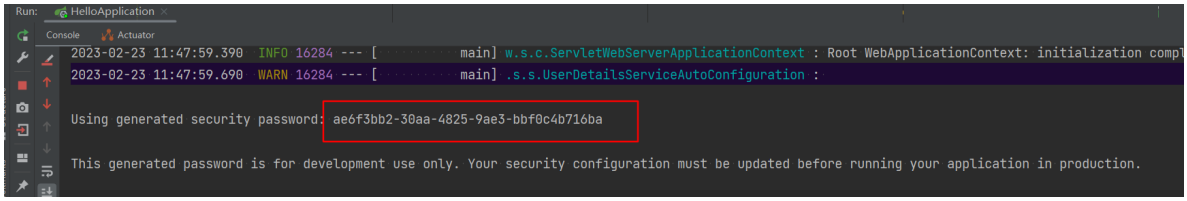
搭建环境

导入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

编写 Controller

```java
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() throws JsonProcessingException {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        Object principal = authentication.getPrincipal();
        return new ObjectMapper().writeValueAsString(principal);
    }

}
```



访问：http://localhost:8080/hello，页面会跳转到 http://localhost:8080/login 进行登入


```
- 默认用户名为: user
- 默认密码为: 控制台打印的 uuid
```

这就是 Spring Security 的强大之处，只需要引入一个依赖，所有的接口就会自动保护起来！
思考？

- 为什么引入 Spring Security 之后没有任何配置所有请求就要认证呢?
- 在项目中明明没有登录界面， 登录界面 怎么来的呢?
- 为什么使用 user 和 控制台密码 能登陆，登录时验证数据源存在哪里呢?

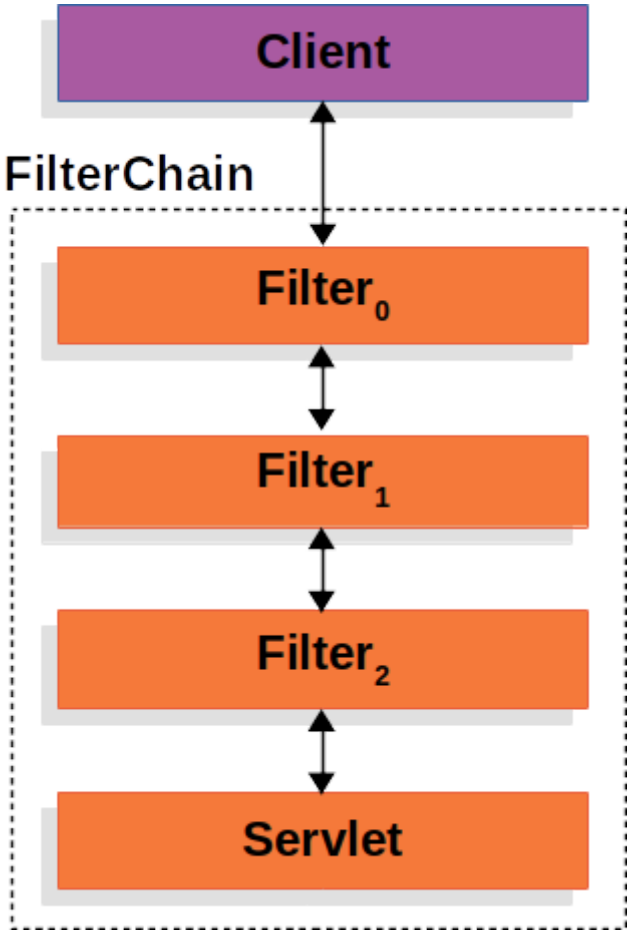
实现原理

https://docs.spring.io/spring-security/reference/servlet/architecture.html

总结一下:

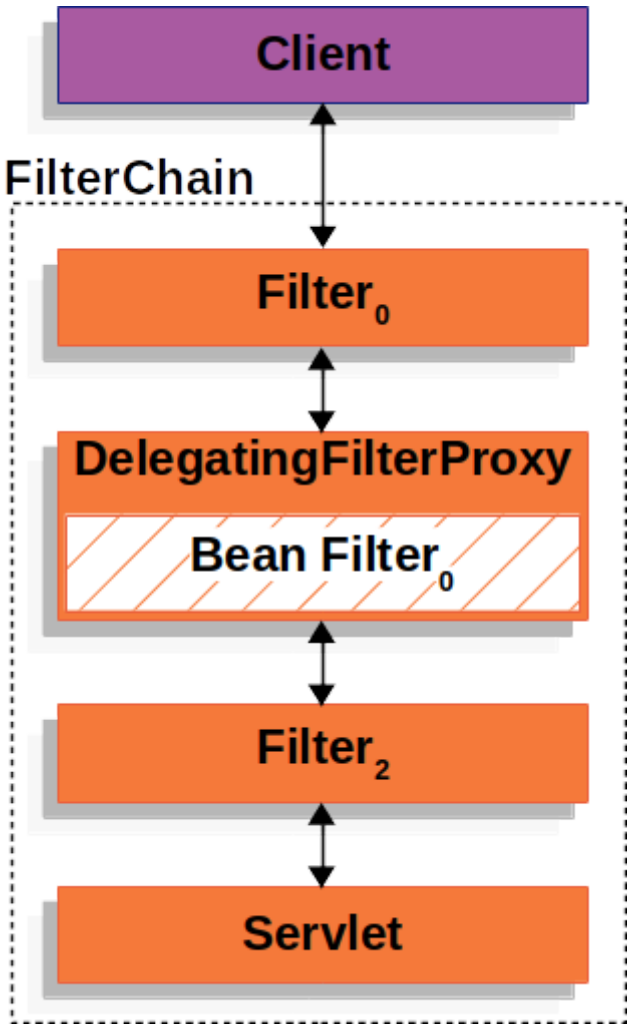
> Spring Security的Servlet支持是基于Servlet过滤器的

> 客户端向应用程序发送一个请求，容器创建一个FilterChain，其中包含Filter实例和Servlet，应该根据请求URI的路径来处理HttpServletRequest。在Spring MVC应用程序中，Servlet 是DispatcherServlet 的一个实例。

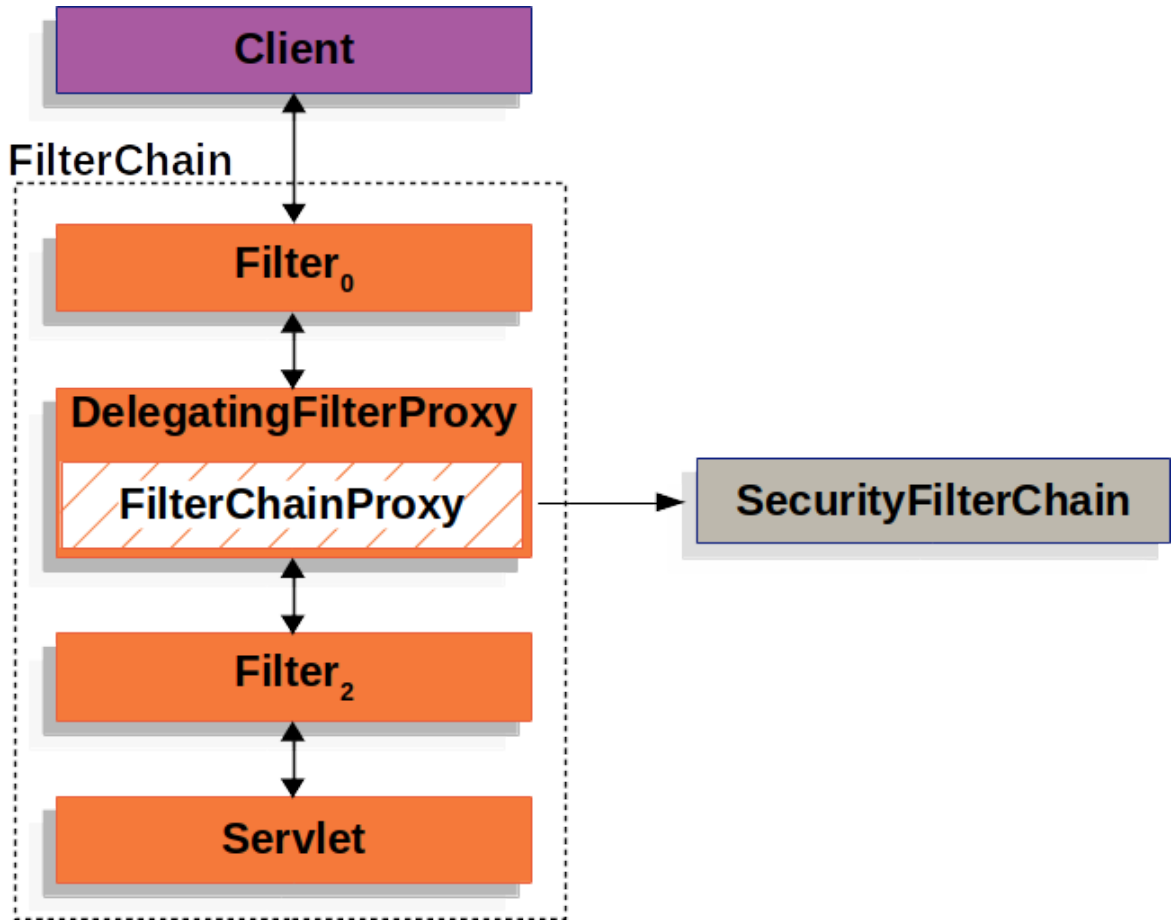


> 由于一个Filter只影响下游的Filter实例和Servlet，所以每个Filter的调用顺序是非常重要的。

> Spring 提供了一个名为 Delegating FilterProxy 的过滤器实现，允许在Servlet容器的生命周期和Spring 的ApplicationContext 之间建立桥梁。Servlet容器允许通过使用自己的标准来注册 Filter 实例，但它不知道 Spring 定义的 Bean。你可以通过标准的 Servlet 容器机制来注册DelegatingFilterProxy，但将所有工作委托给实现 Filter的 Spring Bean。
>
> DelegatingFilterProxy 的另一个好处是，它允许延迟查找 Filter Bean实例。这一点很重要，因为在容器启动之前，容器需要注册Filter实例。然而，Spring 通常使用 ContextLoaderListener 来加载 Spring Bean，这在需要注册Filter 实例之后才会完成。



> Servlet 容器允许使用自己的标准来注册Filter实例，自定义过滤器并不是直接放在 Web 项目的原生过滤器链中，而是通过一个 FlterChainProxy 来统一管理。 Spring Security 中的过滤器链通过 FilterChainProxy 嵌入到 Web 项目的原生过滤器链中。 FilterChainProxy 作为一个顶层的管理者，将统一管理 Security Filter。
>
> FilterChainProxy 本身是通过 Spring 框架提供的 DelegatingFilterProxy 整合到原生的过滤器链中。



servlet 与 spring 之间的联系：https://www.cnblogs.com/shawshawwan/p/9002126.html

为什么不直接注册到 Servlet 容器 或者 DelegatingFilterProxy ?

SecurityFilterChain 中注册的是 Bean，这些 Bean 是注册在 FilterChainProxy 中的，相对于直接注册到 Servelt 容器 或者 DelegatingFilterProxy，FilterChainProxy提供了许多优势：

- 它为 Spring Security 的所有 Servlet 支持提供了一个起点，方便代码调试
- 由于 FilterChainProxy 是 Spring Security 使用的核心，它可以执行一些不被视为可有可无的任务
- 它在确定何时应该调用 SecurityFilterChain 方面提供了更大的灵活性。在 Servlet 容器中，Filter 实例仅基于 URL 被调用。然而，FilterChainProxy 可以通过使用 RequestMatcher 接口，根据 HttpServletRequest 中的任何内容确定调用

源码解析

SpringBootWebSecurityConfiguration

这个类是 spring boot 自动配置类，通过这个源码得知，默认情况下对所有请求进行权限控制:

```
/*
 * Copyright 2012-2022 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
```

```
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.boot.autoconfigure.security.servlet;

import javax.servlet.DispatcherType;

import org.springframework.boot.autoconfigure.condition.ConditionalOnBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication
;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication
.Type;
import
org.springframework.boot.autoconfigure.security.ConditionalOnDefaultWebSecur
ity;
import org.springframework.boot.autoconfigure.security.SecurityProperties;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.boot.web.servlet.filter.ErrorPageSecurityFilter;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import org.springframework.security.config.BeanIds;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSe
curity;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.access.WebInvocationPrivilegeEvaluator;

/**
 * {@link Configuration @Configuration} class securing servlet applications.
 *
 * @author Madhura Bhave
 */
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
class SpringBootWebSecurityConfiguration {

    /**
     * The default configuration for web security. It relies on Spring
Security's
     * content-negotiation strategy to determine what sort of authentication
to use. If
```

```
     * the user specifies their own {@code WebSecurityConfigurerAdapter} or
     * {@link SecurityFilterChain} bean, this will back-off completely and the users
     * should specify all the bits that they want to configure as part of the custom
     * security configuration.
     */
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnDefaultWebSecurity
    static class SecurityFilterChainConfiguration {

        @Bean
        @Order(SecurityProperties.BASIC_AUTH_ORDER)
        SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated();
            http.formLogin();
            http.httpBasic();
            return http.build();
        }
    }

    /**
     * Adds the {@link EnableWebSecurity @EnableWebSecurity} annotation if Spring Security
     * is on the classpath. This will make sure that the annotation is present with
     * default security auto-configuration and also if the user adds custom security and
     * forgets to add the annotation. If {@link EnableWebSecurity @EnableWebSecurity} has
     * already been added or if a bean with name
     * {@value BeanIds#SPRING_SECURITY_FILTER_CHAIN} has been configured by the user, this
     * will back-off.
     */
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnMissingBean(name = BeanIds.SPRING_SECURITY_FILTER_CHAIN)
    @ConditionalOnClass(EnableWebSecurity.class)
    @EnableWebSecurity
    static class WebSecurityEnablerConfiguration {

    }

}
```

这就是为什么在引入 Spring Security 中没有任何配置情况下，请求会被拦截的原因！

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(DefaultWebSecurityCondition.class)
public @interface ConditionalOnDefaultWebSecurity {

}
```

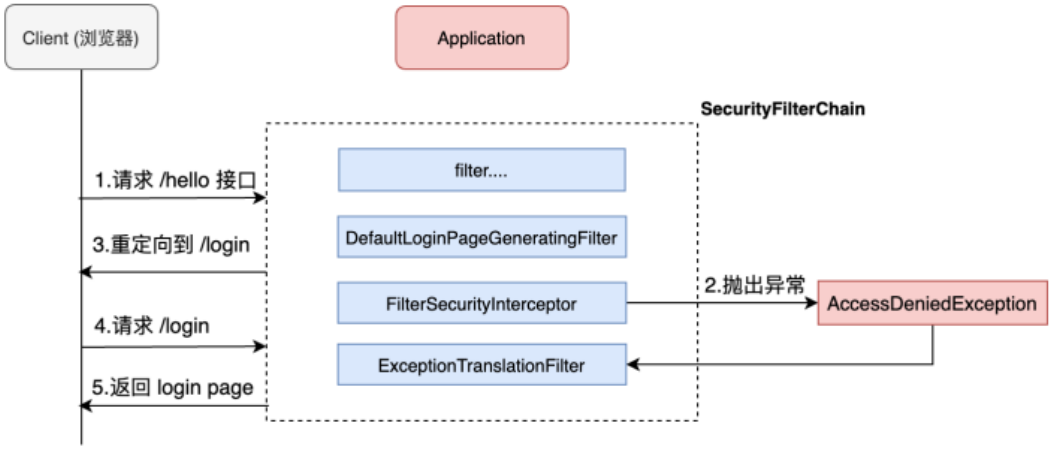
```
class DefaultWebSecurityCondition extends AllNestedConditions {

    DefaultWebSecurityCondition() {
        super(ConfigurationPhase.REGISTER_BEAN);
    }

    @ConditionalOnClass({ SecurityFilterChain.class, HttpSecurity.class })
    static class Classes {

    }

    @ConditionalOnMissingBean({
org.springframework.security.config.annotation.web.configuration.WebSecurity
ConfigurerAdapter.class,
            SecurityFilterChain.class })
    @SuppressWarnings("deprecation")
    static class Beans {

    }

}
```

通过上面对自动配置分析，我们也能看出默认生效条件为:

- 条件一 classpath中存在 SecurityFilterChain.class，httpSecurity.class
- 条件二 没有自定义 WebSecurityConfigurerAdapter.class，SecurityFilterChain.class

> 补充说明:
>
> @ConditionalOnClass：当项目中存在他条件中的某个类时才会使标有该注解的类或方法生效;
>
> @ConditionalOnMissingBean：判断 Spring 容器中该 bean 实例是否存在，存在则不注入，没有就注入

流程分析



1. 请求 /hello 接口，在引入 spring security 之后会先经过一些列过滤器

2. 在请求到达 FilterSecurityInterceptor时，发现请求并未认证。请求拦截下来，并抛出 AccessDeniedException 异常
3. 抛出 AccessDeniedException 的异常会被 ExceptionTranslationFilter 捕获，这个 Filter 中会调用 LoginUrlAuthenticationEntryPoint#commence方法给客户端返回 302，要求客户端进行重定向到 /login 页面。
4. 客户端发送 /login 请求。
5. /login 请求会再次被拦截器中 DefaultLoginPageGeneratingFilter 拦截到，并在拦截器中返回生成登录页面。

就是通过这种方式， Spring Security 默认过滤器中生成了登录页面，并返回!

默认用户生成

1.查看 SecurityFilterChainConfiguration.defaultSecurityFilterChain() 方法表单登录

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnDefaultWebSecurity
static class SecurityFilterChainConfiguration {

    @Bean
    @Order(SecurityProperties.BASIC_AUTH_ORDER)
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated();
        http.formLogin();
        http.httpBasic();
        return http.build();
    }

}
```

2.处理登录为 FormLoginConfigurer 类中 调用 UsernamePasswordAuthenticationFilter 这个类实例

```
public final class FormLoginConfigurer<H extends HttpSecurityBuilder<H>> extends AbstractAuthenticationFilterConfigurer
    public FormLoginConfigurer() {
        super(new UsernamePasswordAuthenticationFilter(), (String)null);
        this.usernameParameter("username");
        this.passwordParameter("password");
    }
```

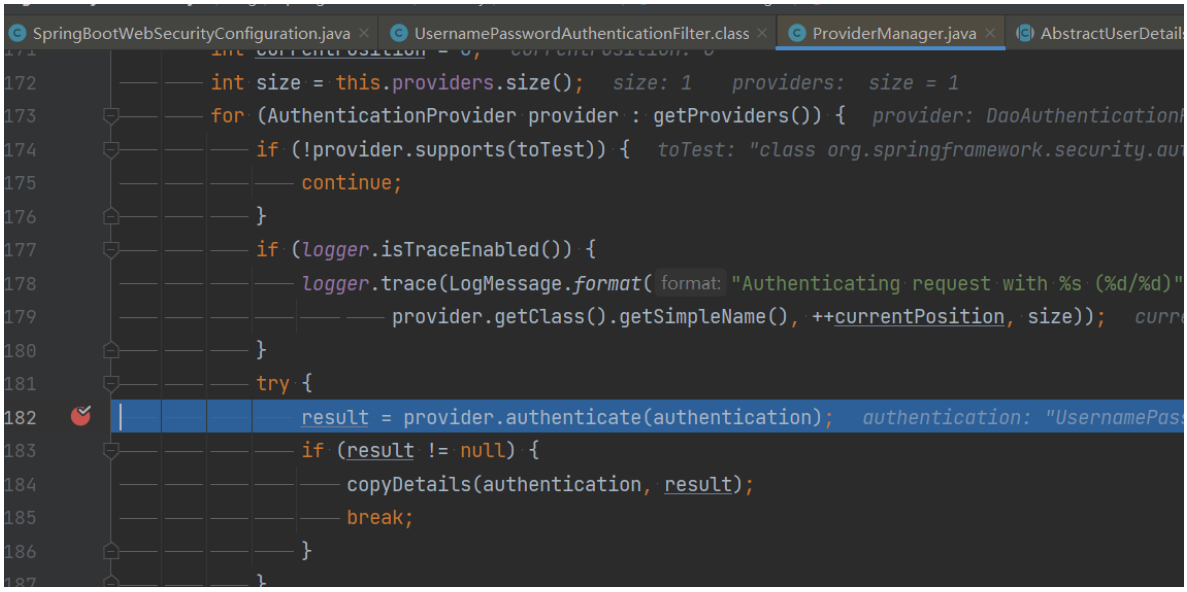
3.查看类中 UsernamePasswordAuthenticationFilter.attempAuthentication() 方法得知实际调用 AuthenticationManager 中 authenticate 方法

SpringBootWebSecurityConfiguration.java | UsernamePasswordAuthenticationFilter.class | ProviderManager.java | AbstractUserDetailsAuthenticationProvider.java | DaoAuthenticationProvide

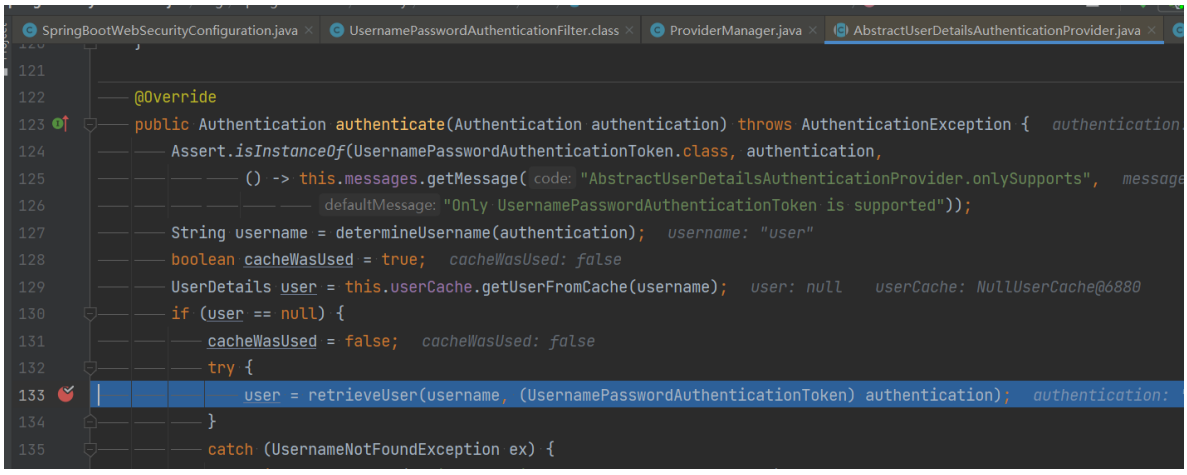
Decompiled .class file, bytecode version: 52.0 (Java 8)

```
31  public UsernamePasswordAuthenticationFilter(AuthenticationManager authenticationManager) {
32      super(DEFAULT_ANT_PATH_REQUEST_MATCHER, authenticationManager);
33  }
34
35  public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
36      if (this.postOnly && !request.getMethod().equals("POST")) {
37          throw new AuthenticationServiceException("Authentication method not supported: " + request.getMethod());
38      } else {
39          String username = this.obtainUsername(request);
40          username = username != null ? username.trim() : "";
41          String password = this.obtainPassword(request);
42          password = password != null ? password : "";
43          UsernamePasswordAuthenticationToken authRequest = UsernamePasswordAuthenticationToken.unauthenticated(username, password);
44          this.setDetails(request, authRequest);
45          return this.getAuthenticationManager().authenticate(authRequest);
46      }
```

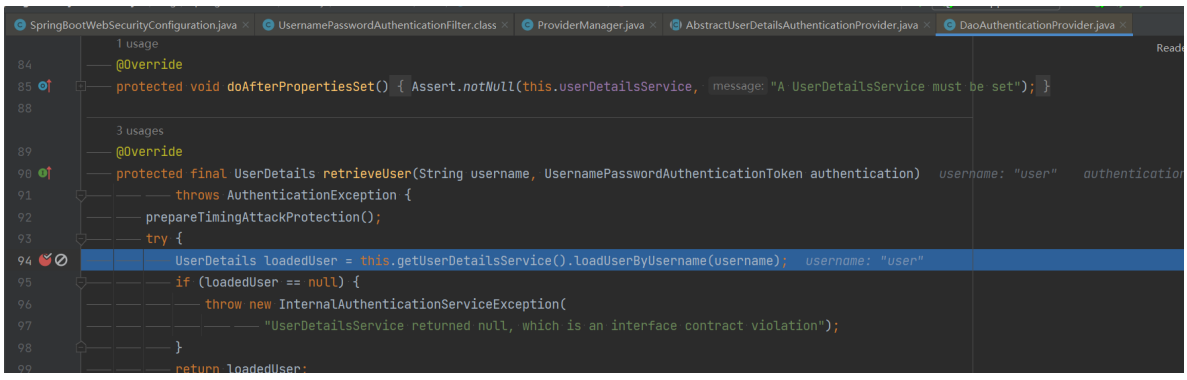
4.调用 ProviderManager 类中方法 authenticate

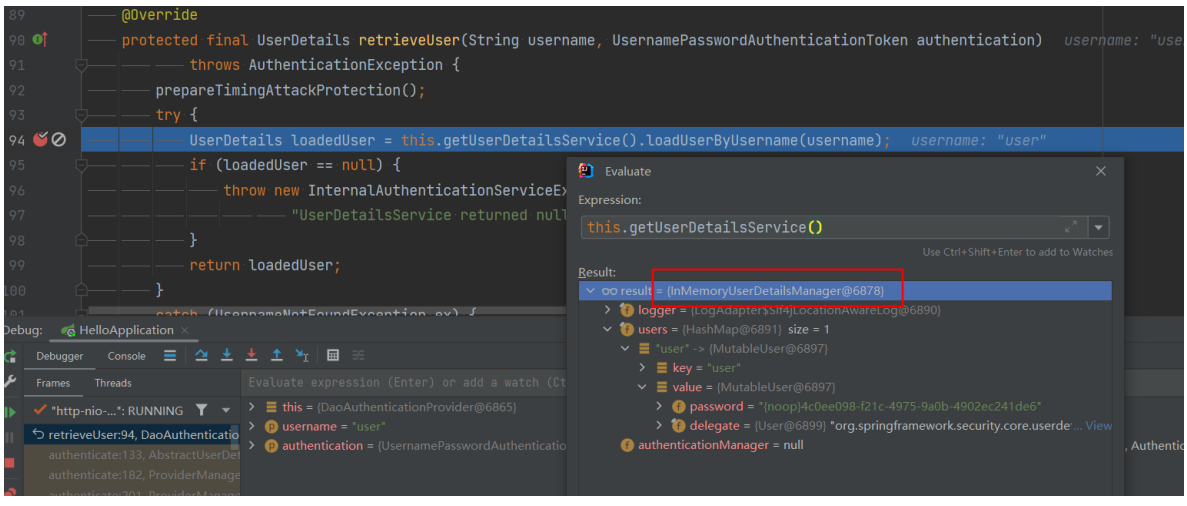


5.调用了 ProviderManager 实现类中 AbstractUserDetailsAuthenticationProvider 类中方法



6.最终调用实现类 DaoAuthenticationProvider 类中方法比较



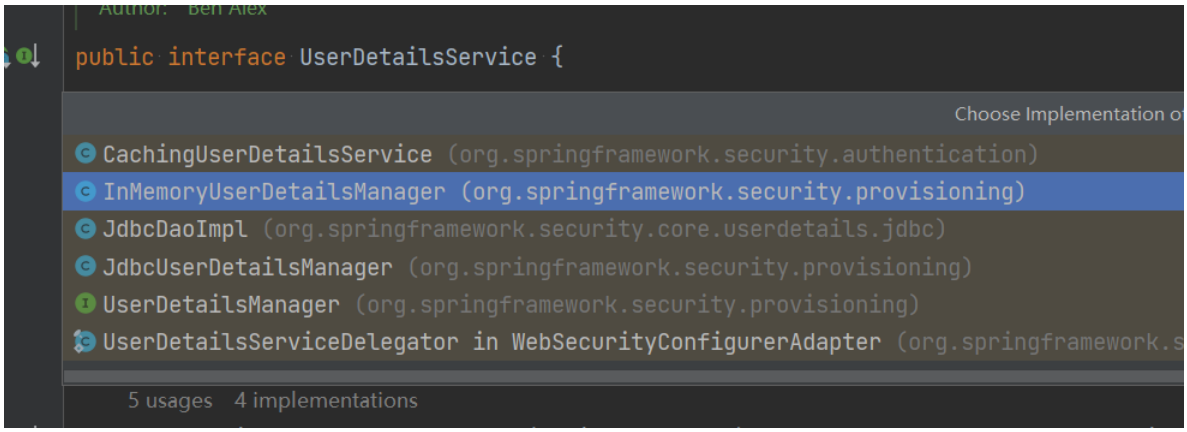


看到这里就知道默认实现是基于 InMemoryUserDetailsManager 这个类,也就是内存的实现!

UserDetailService

UserDetailService 是顶层父接口，接口中 loadUserByUserName 方法是用来在认证时进行用户名认证方法，默认实现使用是内存实现，如果想要修改数据库实现我们只需要自定义 UserDetailService 实现，最终返回 UserDetails 实例即可。

```
public interface UserDetailsService {

    /**
     * Locates the user based on the username. In the actual implementation, the search
     * may possibly be case sensitive, or case insensitive depending on how the
     * implementation instance is configured. In this case, the <code>UserDetails</code>
     * object that comes back may have a username that is of a different case than what
     * was actually requested..
     * @param username the username identifying the user whose data is required.
     * @return a fully populated user record (never <code>null</code>)
     * @throws UsernameNotFoundException if the user could not be found or the user has no
     * GrantedAuthority
     */
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```



UserDetailServiceAutoConfigutation

```
@AutoConfiguration
@ConditionalOnClass(AuthenticationManager.class)
@ConditionalOnBean(ObjectPostProcessor.class)
@ConditionalOnMissingBean(
        value = { AuthenticationManager.class, AuthenticationProvider.class, UserDetailsService.class,
```

```
                AuthenticationManagerResolver.class },
        type = { "org.springframework.security.oauth2.jwt.JwtDecoder",
"org.springframework.security.oauth2.server.resource.introspection.OpaqueTokenIntrospector",
"org.springframework.security.oauth2.client.registration.ClientRegistrationRepository",
"org.springframework.security.saml2.provider.service.registration.RelyingPartyRegistrationRepository" })
public class UserDetailsServiceAutoConfiguration {

    private static final String NOOP_PASSWORD_PREFIX = "{noop}";

    private static final Pattern PASSWORD_ALGORITHM_PATTERN = Pattern.compile("^\\{.+}.*$");

    private static final Log logger = LogFactory.getLog(UserDetailsServiceAutoConfiguration.class);

    @Bean
    @Lazy
    public InMemoryUserDetailsManager inMemoryUserDetailsManager(SecurityProperties properties,
            ObjectProvider<PasswordEncoder> passwordEncoder) {
        SecurityProperties.User user = properties.getUser();
        List<String> roles = user.getRoles();
        return new InMemoryUserDetailsManager(
User.withUsername(user.getName()).password(getOrDeducePassword(user, passwordEncoder.getIfAvailable()))
                        .roles(StringUtils.toStringArray(roles)).build());
    }

    private String getOrDeducePassword(SecurityProperties.User user, PasswordEncoder encoder) {
        String password = user.getPassword();
        if (user.isPasswordGenerated()) {
            logger.warn(String.format(
                    "%n%nUsing generated security password: %s%n%nThis generated password is for development use only. "
                            + "Your security configuration must be updated before running your application in "
                            + "production.%n",
                    user.getPassword()));
        }
        if (encoder != null || PASSWORD_ALGORITHM_PATTERN.matcher(password).matches()) {
            return password;
        }
        return NOOP_PASSWORD_PREFIX + password;
    }

}
```

结论

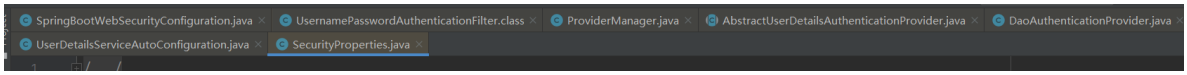
- 从自动配置源码中得知当 classpath 下存在 AuthenticationManager 类
- 当前项目中，系统没有提供 AuthenticationManager.class、AuthenticationProvider.class、UserDetailsService.class、AuthenticationManagerResolver.class实例

默认情况下都会满足，此时Spring Security会提供一个 InMemoryUserDetailManager 实例


```java
@ConfigurationProperties(prefix = "spring.security")
public class SecurityProperties {

    private final User user = new User();

    public User getUser() {
        return this.user;
    }

    public static class User {

        /**
         * Default user name.
         */
        private String name = "user";

        /**
         * Password for the default user name.
         */
        private String password = UUID.randomUUID().toString();

        /**
         * Granted roles for the default user name.
         */
        private List<String> roles = new ArrayList<>();

        // ...
    }
}
```

这就是默认生成 user 以及 uuid 密码过程! 另外看明白源码之后，就知道只要在配置文件中加入如下配置可以对内存中用户和密码进行覆盖。

```properties
spring.security.user.name=root
spring.security.user.password=root
spring.security.user.roles=admin,users
```



认证原理

自定义资源规则权限

> Spring Security 5.7.0 弃用了 WebSecurityConfigurerAdapter

官网博客链接地址：https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter

5.7.0 之前的配置

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .mvcMatchers("/index").permitAll()
            .anyRequest().authenticated()
            .and().formLogin();
    }
}
```

5.7.0 之后的配置

```
@Configuration
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index")  // 注意: 放行资源必须放在所有认证请求之前!
                .permitAll() // 代表放行该资源,该资源为公共资源 无需认证和授权可以直接访问
                .anyRequest().authenticated() // 代表所有请求,必须认证之后才能访问
                .and().formLogin(); // 代表开启表单认证
        return http.build();
    }
}
```

自定义登入成功 / 失败处理

由于现在项目都为前后端分离，所以这里展示页面跳转情况，如有需求，B站搜索：**编程不良人**

在前后端分离开发中就不需要成功之后跳转页面。只需要给前端返回一个 apiKey。

```
public interface AuthenticationSuccessHandler {

    /**
     * Called when a user has been successfully authenticated.
     * @param request the request which caused the successful authentication
     * @param response the response
     * @param chain the {@link FilterChain} which can be used to proceed other filters in
     * the chain
     * @param authentication the <tt>Authentication</tt> object which was created during
     * the authentication process.
     * @since 5.2.0
     */
    default void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, FilterChain chain,
            Authentication authentication) throws IOException, ServletException {
        onAuthenticationSuccess(request, response, authentication);
        chain.doFilter(request, response);
    }

    /**
     * Called when a user has been successfully authenticated.
     * @param request the request which caused the successful authentication
     * @param response the response
     * @param authentication the <tt>Authentication</tt> object which was created during
     * the authentication process.
     */
    void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
            Authentication authentication) throws IOException, ServletException;

}
```

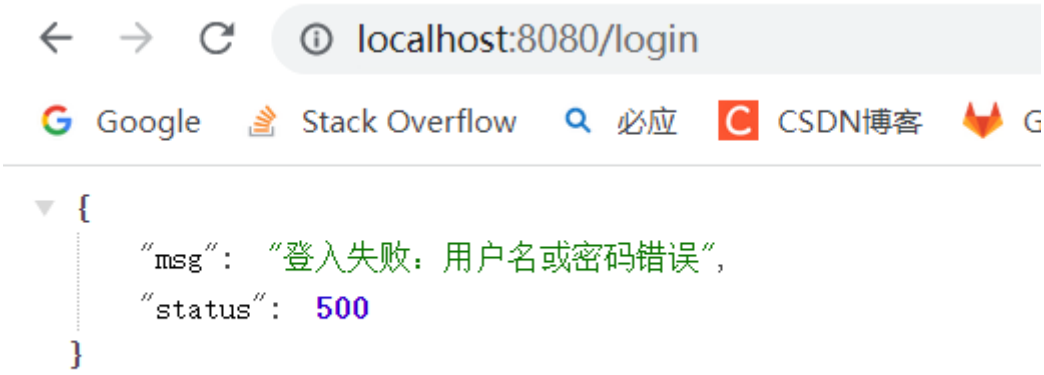
根据接口的描述信息,也可以得知登录成功会自动回调这个方法，进一步查看它的默认实现，你会发现 successForwardUrl、 defaultSuccessUrl也是由它的子类实现的

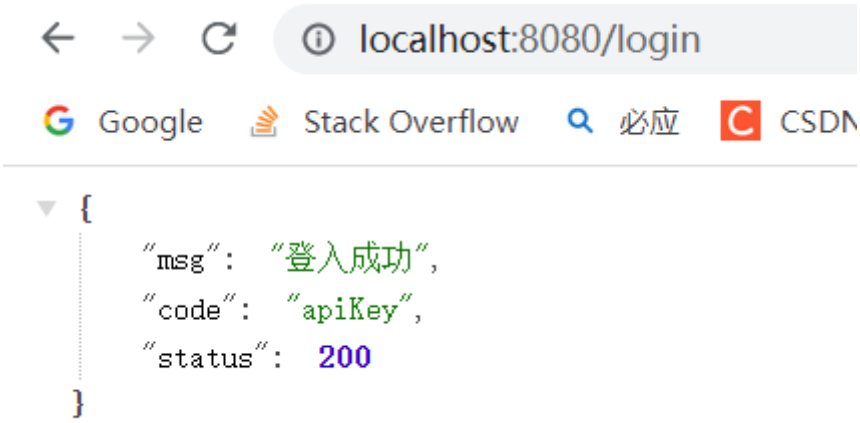
```
public class LoginSuccessHandler implements AuthenticationSuccessHandler {
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {
        HashMap<String, Object> map = new HashMap<>();
        map.put("msg", "登入成功");
        map.put("status", 200);
        map.put("code", "apiKey");
        response.setContentType("application/json;charset=UTF-8");
        String json = new ObjectMapper().writeValueAsString(map);
        response.getWriter().println(json);
    }
}
```

```java
public interface AuthenticationFailureHandler {

    /**
     * Called when an authentication attempt fails.
     * @param request the request during which the authentication attempt occurred.
     * @param response the response.
     * @param exception the exception which was thrown to reject the authentication
     * request.
     */
    void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,
            AuthenticationException exception) throws IOException, ServletException;

}
```

```java
public class LoginFailureHandler implements AuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response, AuthenticationException exception) throws IOException, ServletException {
        HashMap<String, Object> map = new HashMap<>();
        map.put("msg", "登入失败：" + exception.getMessage());
        map.put("status", 500);
        response.setContentType("application/json;charset=UTF-8");
        String json = new ObjectMapper().writeValueAsString(map);
        response.getWriter().println(json);
    }
}
```

```java
@Configuration
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index")
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .successHandler(new LoginSuccessHandler())
                .failureHandler(new LoginFailureHandler())
                .and().csrf().disable();
        return http.build();
    }
}
```





注销登入

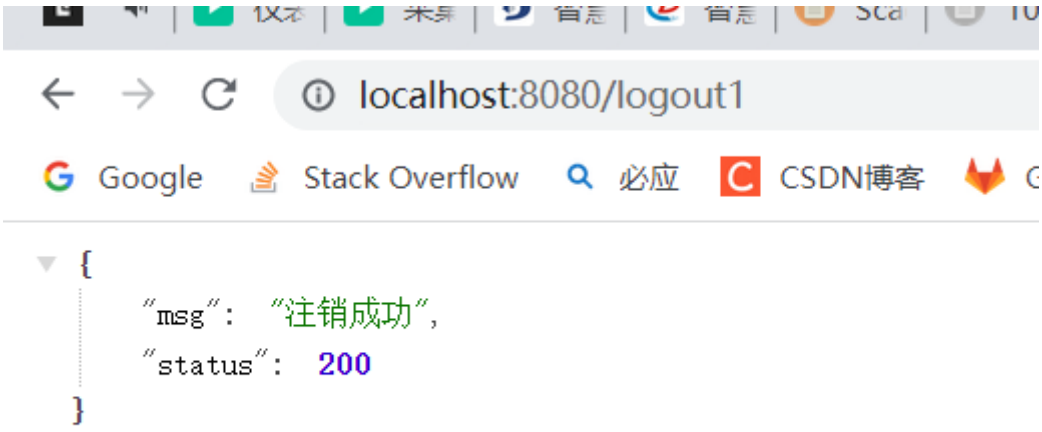
Spring Security 中也提供了默认的注销登录配置，在开发时也可以按照自己需求对注销进行个性化定制。

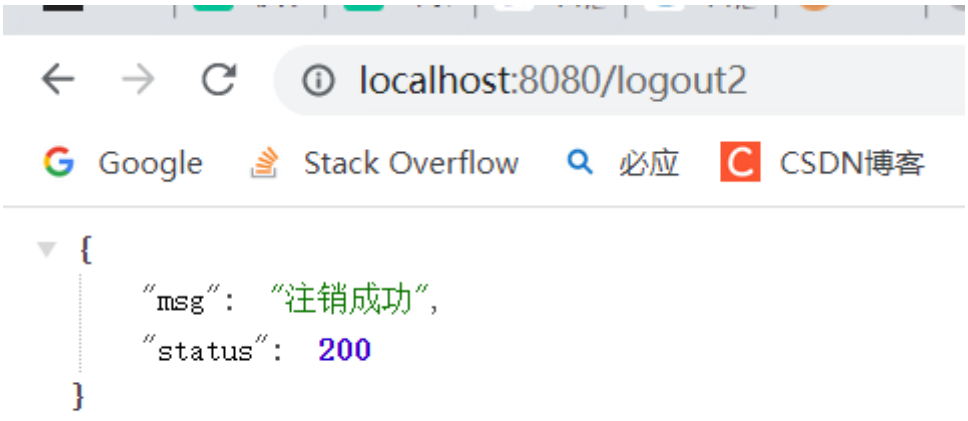
前后端分离开发，注销成功之后就不需要页面跳转了，只需要将注销成功的信息返回前端即可，此时我们可以通过自定义 LogoutSuccessHandler 实现来返回注销之后信息：

```
public class LogoutHandler implements LogoutSuccessHandler {
    @Override
    public void onLogoutSuccess(HttpServletRequest request,
HttpServletResponse response, Authentication authentication) throws
IOException, ServletException {
        HashMap<String, Object> map = new HashMap<>();
        map.put("msg", "注销成功" );
        map.put("status", 200);
        response.setContentType("application/json;charset=UTF-8");
        String json = new ObjectMapper().writeValueAsString(map);
        response.getWriter().println(json);
    }
}
```

```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
```

```
                .mvcMatchers("/index")
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .successHandler(new LoginSuccessHandler())
                .failureHandler(new LoginFailureHandler())
                .and()
                .logout()
                .logoutRequestMatcher(
                        new OrRequestMatcher(
                                new AntPathRequestMatcher("/logout1",
"GET"),
                                new AntPathRequestMatcher("/logout2",
"GET")))
                .logoutSuccessHandler(new LogoutHandler());
        return http.csrf().disable().build();
    }
```





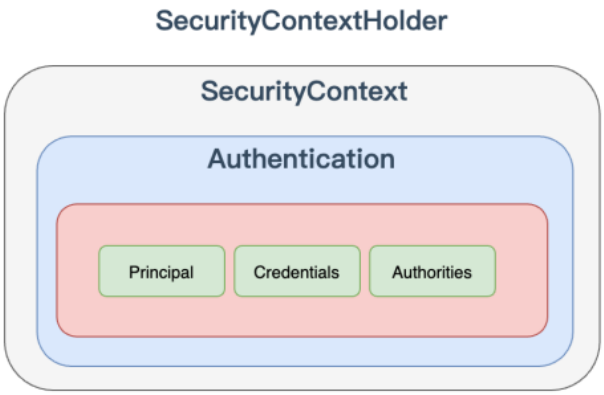
登录用户数据获取

SecurityContextHolder

Spring Security 会将登录用户数据保存在 Session 中。但是，为了使用方便, Spring Security 在此基础上还做了一些改进，其中最主要的一个变化就是线程绑定。当用户登录成功后,Spring Security 会将登录成功的用户信息保存到 SecurityContextHolder 中。

SecurityContextHolder 中的数据保存默认是通过 ThreadLocal 来实现的，使用 ThreadLocal 创建的变量只能被当前线程访问，不能被其他线程访问和修改，也就是用户数据和请求线程绑定在一起。当登录请求处理完毕后， Spring Security 会将SecurityContextHolder 中的数据拿出来保存到 Session 中，同时将 SecurityContexHolder 中的数据清空。以后每当有请求到来时， Spring Security 就会先从 Session 中取出用户登录数据，保存到SecurityContextHolder 中，方便在该请求的后续处理过程中使用，同时在请求结束时将 SecurityContextHolder 中的数据拿出来保存到 Session 中，然后将 SecurityContextHolder 中的数据清空。

实际上 SecurityContextHolder 中存储是 SecurityContext，在SecurityContext 中存储是 Authentication。



这种设计是典型的策略设计模式:

```
1  public class SecurityContextHolder {
2
3      public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";
4
5      public static final String MODE_INHERITABLETHREADLOCAL =
   "MODE_INHERITABLETHREADLOCAL";
6
7      public static final String MODE_GLOBAL = "MODE_GLOBAL";
8
9      private static final String MODE_PRE_INITIALIZED =
   "MODE_PRE_INITIALIZED";
10
11     public static final String SYSTEM_PROPERTY =
   "spring.security.strategy";
12
13     private static String strategyName =
   System.getProperty(SYSTEM_PROPERTY);
14
15     private static SecurityContextHolderStrategy strategy;
16
```

```
17      private static int initializeCount = 0;
18
19      static {
20          initialize();
21      }
22
23      private static void initialize() {
24          initializeStrategy();
25          initializeCount++;
26      }
27
28      private static void initializeStrategy() {
29          if (MODE_PRE_INITIALIZED.equals(strategyName)) {
30              Assert.state(strategy != null, "When using " +
MODE_PRE_INITIALIZED
31                      + ", setContextHolderStrategy must be called with the
fully constructed strategy");
32              return;
33          }
34          if (!StringUtils.hasText(strategyName)) {
35              // Set default
36              strategyName = MODE_THREADLOCAL;
37          }
38          if (strategyName.equals(MODE_THREADLOCAL)) {
39              strategy = new ThreadLocalSecurityContextHolderStrategy();
40              return;
41          }
42          if (strategyName.equals(MODE_INHERITABLETHREADLOCAL)) {
43              strategy = new
InheritableThreadLocalSecurityContextHolderStrategy();
44              return;
45          }
46          if (strategyName.equals(MODE_GLOBAL)) {
47              strategy = new GlobalSecurityContextHolderStrategy();
48              return;
49          }
50          // Try to load a custom strategy
51          try {
52              Class<?> clazz = Class.forName(strategyName);
53              Constructor<?> customStrategy = clazz.getConstructor();
54              strategy = (SecurityContextHolderStrategy)
customStrategy.newInstance();
55          }
56          catch (Exception ex) {
57              ReflectionUtils.handleReflectionException(ex);
58          }
59      }
60
61      /**
62       * Explicitly clears the context value from the current thread.
63       */
64      public static void clearContext() {
65          strategy.clearContext();
66      }
67
68      /**
69       * Obtain the current <code>SecurityContext</code>.
70       * @return the security context (never <code>null</code>)
```

```
71       */
72      public static SecurityContext getContext() {
73          return strategy.getContext();
74      }
75
76      /**
77       * Primarily for troubleshooting purposes, this method shows how many
    times the class
78       * has re-initialized its <code>SecurityContextHolderStrategy</code>.
79       * @return the count (should be one unless you've called
80       * {@link #setStrategyName(String)} or
81       * {@link #setContextHolderStrategy(SecurityContextHolderStrategy)} to
    switch to an
82       * alternate strategy).
83       */
84      public static int getInitializeCount() {
85          return initializeCount;
86      }
87
88      /**
89       * Associates a new <code>SecurityContext</code> with the current
    thread of execution.
90       * @param context the new <code>SecurityContext</code> (may not be
    <code>null</code>)
91       */
92      public static void setContext(SecurityContext context) {
93          strategy.setContext(context);
94      }
95
96      /**
97       * Changes the preferred strategy. Do <em>NOT</em> call this method
    more than once for
98       * a given JVM, as it will re-initialize the strategy and adversely
    affect any
99       * existing threads using the old strategy.
100      * @param strategyName the fully qualified class name of the strategy
    that should be
101      * used.
102      */
103     public static void setStrategyName(String strategyName) {
104         SecurityContextHolder.strategyName = strategyName;
105         initialize();
106     }
107
108     /**
109      * Use this {@link SecurityContextHolderStrategy}.
110      *
111      * Call either {@link #setStrategyName(String)} or this method, but
    not both.
112      *
113      * This method is not thread safe. Changing the strategy while
    requests are in-flight
114      * may cause race conditions.
115      *
116      * {@link SecurityContextHolder} maintains a static reference to the
    provided
117      * {@link SecurityContextHolderStrategy}. This means that the strategy
    and its members
```

```
     * will not be garbage collected until you remove your strategy.
     *
     * To ensure garbage collection, remember the original strategy like so:
     *
     * <pre>
     *     SecurityContextHolderStrategy original = SecurityContextHolder.getContextHolderStrategy();
     *     SecurityContextHolder.setContextHolderStrategy(myStrategy);
     * </pre>
     *
     * And then when you are ready for {@code myStrategy} to be garbage collected you can
     * do:
     *
     * <pre>
     *     SecurityContextHolder.setContextHolderStrategy(original);
     * </pre>
     * @param strategy the {@link SecurityContextHolderStrategy} to use
     * @since 5.6
     */
    public static void setContextHolderStrategy(SecurityContextHolderStrategy strategy) {
        Assert.notNull(strategy, "securityContextHolderStrategy cannot be null");
        SecurityContextHolder.strategyName = MODE_PRE_INITIALIZED;
        SecurityContextHolder.strategy = strategy;
        initialize();
    }

    /**
     * Allows retrieval of the context strategy. See SEC-1188.
     * @return the configured strategy for storing the security context.
     */
    public static SecurityContextHolderStrategy getContextHolderStrategy() {
        return strategy;
    }

    /**
     * Delegates the creation of a new, empty context to the configured strategy.
     */
    public static SecurityContext createEmptyContext() {
        return strategy.createEmptyContext();
    }

}
```

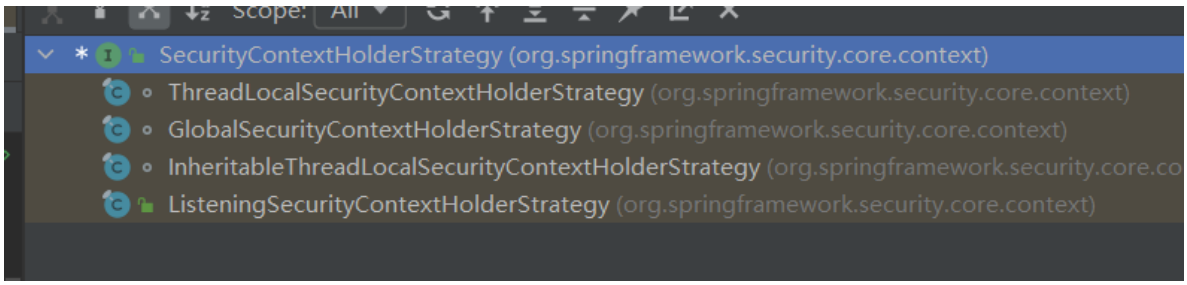
1. MODE THREADLOCAL：这种存放策略是将 SecurityContext 存放在 ThreadLocal 中，大家知道 Threadlocal 的特点是在哪个线程中存储就要在哪个线程中读取，这其实非常适合 web 应用，因为在默认情况下，一个请求无论经过多少 Filter 到达 Servlet，都是由一个线程来处理的。这也是 SecurityContextHolder 的默认存储策略，这种存储策略意味着如果在具体的业务处理代码中，开启了子线程，在子线程中去获取登录用户数据，就会获取不到。
2. MODE INHERITABLETHREADLOCAL：这种存储模式适用于多线程环境，如果希望在子线程中也能够获取到登录用户数据，那么可以使用这种存储模式。

3. MODE GLOBAL：这种存储模式实际上是将数据保存在一个静态变量中，在 JavaWeb 开发中，这种模式很少使用到。

SecurityContextHolderStrategy

通过 SecurityContextHolder 可以得知， SecurityContextHolderStrategy 接口用来定义存储策略方法

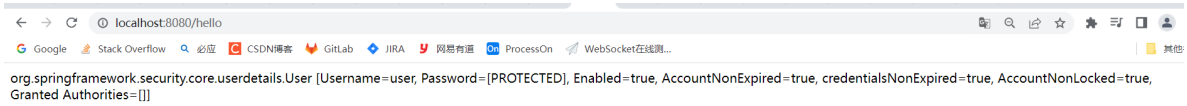
```
public interface SecurityContextHolderStrategy {

    /**
     * Clears the current context.
     */
    void clearContext();

    /**
     * Obtains the current context.
     */
    SecurityContext getContext();

    /**
     * Sets the current context.
     */
    void setContext(SecurityContext context);

    /**
     * Creates a new, empty context implementation, for use by
     */
    SecurityContext createEmptyContext();

}
```



从上面可以看出每一个实现类对应一种策略的实现。

获取用户数据

```
   @GetMapping("/hello")
    public String hello() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        User user = (User) authentication.getPrincipal();
        return user.toString();
    }
```



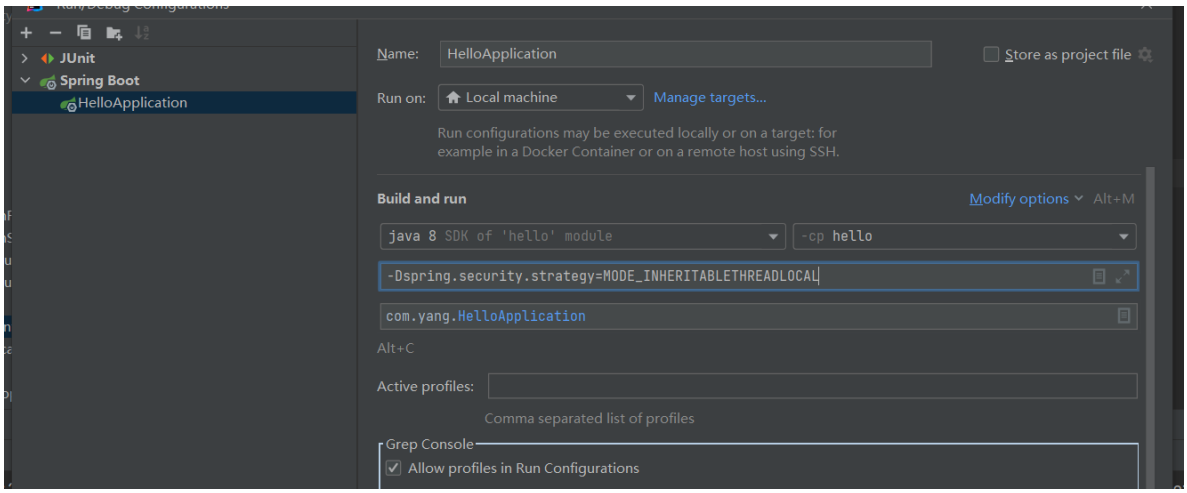
多线程下获取用户数据

```
    @GetMapping("/hello")
    public String hello() {
        new Thread(() -> {
            Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
            User user = (User) authentication.getPrincipal();
            System.out.println(user.toString());
        }).start();
        return "hello page success";
    }
```

```
Exception in thread "Thread-2" java.lang.NullPointerException Create breakpoint
    at com.yang.controller.HelloController.lambda$hello$0(HelloController.java:21)
    at java.lang.Thread.run(Thread.java:750)
Exception in thread "Thread-3" java.lang.NullPointerException Create breakpoint
    at com.yang.controller.HelloController.lambda$hello$0(HelloController.java:21)
    at java.lang.Thread.run(Thread.java:750)
```

可以看到默认策略，是无法在子线程中获取用户信息，如果需要在子线程中获取必须使用第二种策略，默认策略是通过 System.getProperty 加载的，因此我们可以通过增加 VM Options 参数进行修改。

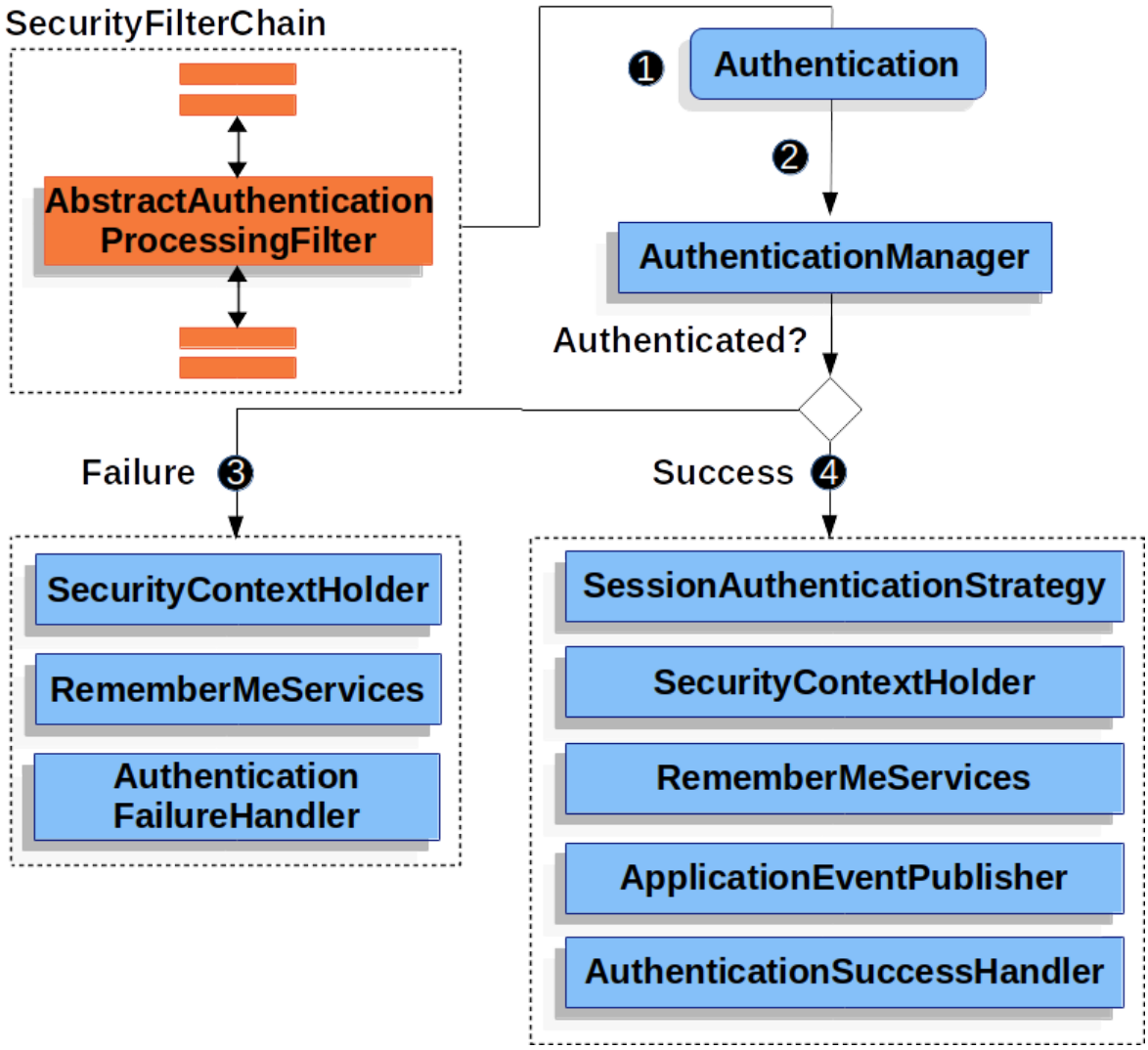
```
-Dspring.security.strategy=MODE_INHERITABLETHREADLOCAL
```



自定义认证数据源

认证流程分析

https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#page-title



- 发起认证请求，请求中携带用户名、密码，该请求会被 UsernamePasswordAuthenticationFilter 拦截
- 在 UsernamePasswordAuthenticationFilter 的 attemptAuthentication 方法中将请求中用户名和密码，封装为Authentication 对象，并交给 AuthenticationManager 进行认证
- 认证成功，将认证信息存储到 SecurityContextHodler 以及调用记住我等，并回调 AuthenticationSuccessHandler 处理
- 认证失败，清除 SecurityContextHodler 以及 记住我中信息，回调 AuthenticationFailureHandler 处理

三者关系

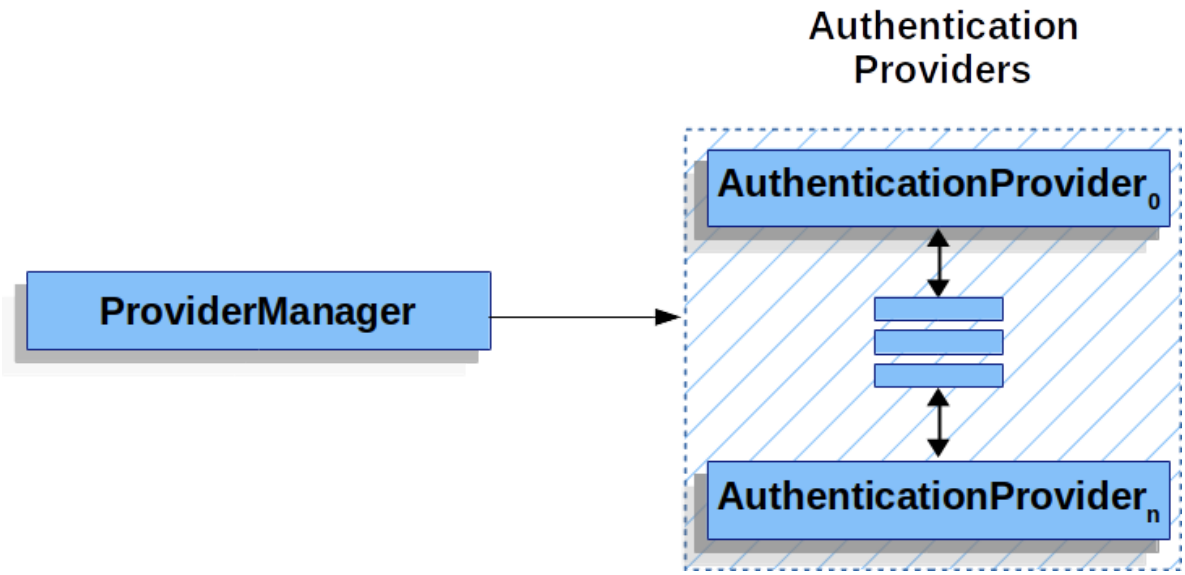
从上面分析中得知， AuthenticationManager 是认证的核心类，但实际上在底层真正认证时还离不开 ProviderManager 以及 AuthenticationProvider 。他们三者关系是样的呢?

- AuthenticationManager 是一个认证管理器，它定义了 Spring Security 过滤器要执行认证操作。
- ProviderManager AuthenticationManager接口的实现类。 Spring Security认证时默认使用就是 ProviderManager
- AuthenticationProvider 就是针对不同的身份类型执行的具体的身份认证。

AuthenticationManager 与 ProviderManager

ProviderManager 是 AuthenticationManager 的唯一实现，也是 Spring Security 默认使用实现。从这里不难看出默认情况下AuthenticationManager 就是一个ProviderManager。

ProviderManager 与 AuthenticationProvider



在 Spring Seourity 中，允许系统同时支持多种不同的认证方式，例如同时支持用户名/密码认证、RememberMe 认证、手机号码动态认证等，而不同的认证方式对应了不同的AuthenticationProvider，所以一个完整的认证流程可能由多个AuthenticationProvider 来提供。

多个 AuthenticationProvider 将组成一个列表，这个列表将由 ProviderManager 代理。换句话说，在ProviderManager 中存在一个 AuthenticationProvider 列表，在Provider Manager 中遍历列表中的每一个 AuthenticationProvider 去执行身份认证，最终得到认证结果。

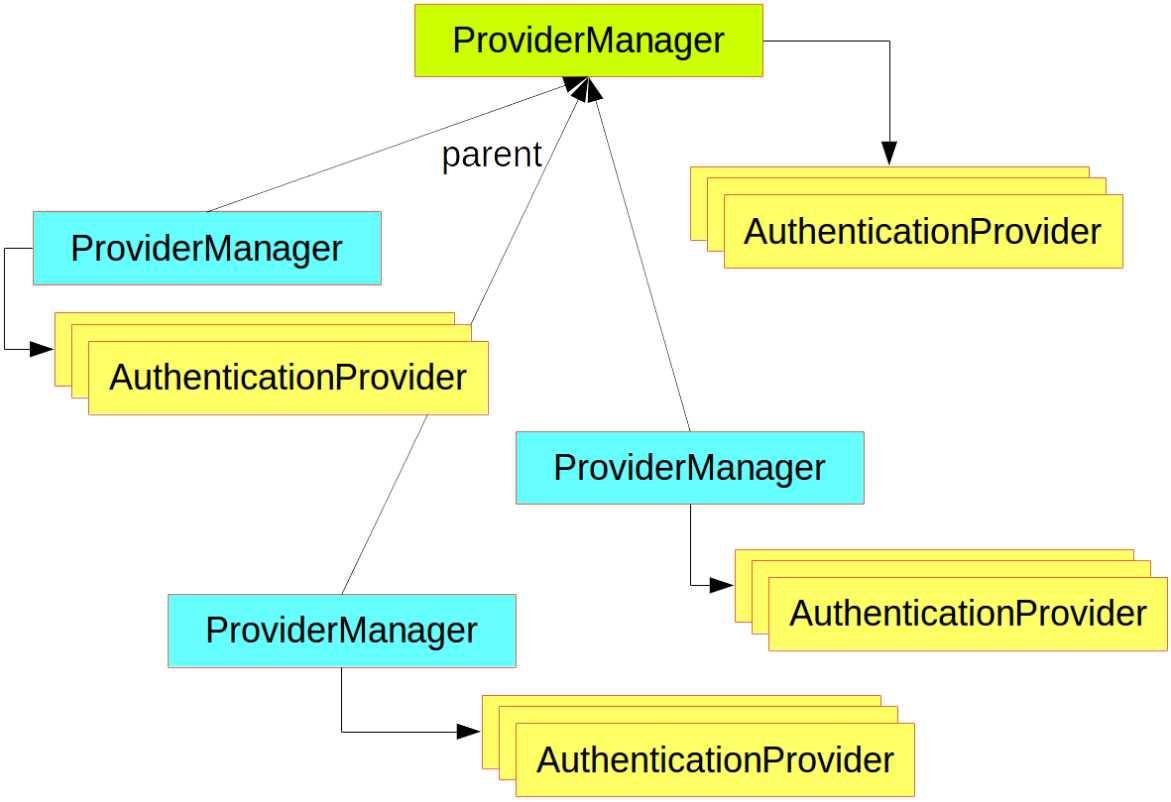
ProviderManager 本身也可以再配置一个 AuthenticationManager 作为 parent，这样当 ProviderManager 认证失败之后，就可以进入到 parent 中再次进行认证。理论上来说，ProviderManager 的 parent 可以是任意类型的

AuthenticationManager，但是通常都是由 ProviderManager 来扮演 parent 的角色，也就是 ProviderManager 是 ProviderManager 的 parent。

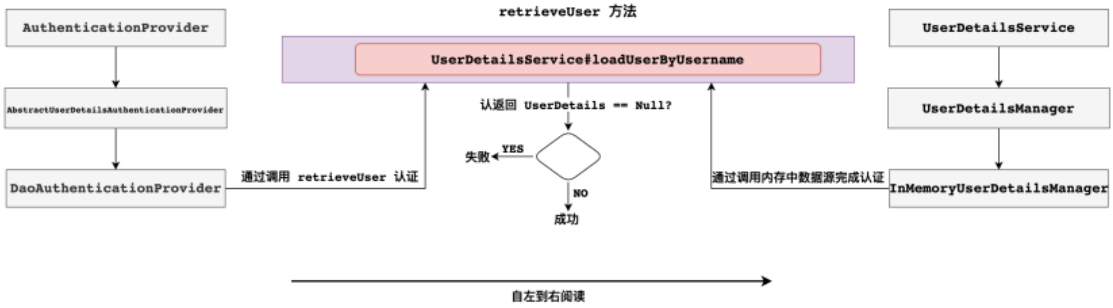
ProviderManager 本身也可以有多个，多个ProviderManager 共用同一个 parent。有时，一个应用程序有受保护资源的逻辑组（例如，所有符合路径模式的网络资源，如/api!!*），每个组可以有自己的专用 AuthenticationManager。通常，每个组都是一个ProviderManager，它们共享一个父级。然后，父级是一种 全局资源，作为所有提供者的后备资源。

https://spring.io/guides/topicals/spring-security-architecture/

根据上面的介绍，我们绘出新的 AuthenticationManager、 ProvideManager 和 AuthentictionProvider 关系



弄清楚认证原理之后我们来看下具体认证时数据源的获取。 默认情况下 AuthenticationProvider 是由 DaoAuthenticationProvider 类来实现认证的，在DaoAuthenticationProvider 认证时又通过 UserDetailsService 完成数据源的校验。 他们之间调用关系如下：



总结: AuthenticationManager 是认证管理器，在 Spring Security 中有全局 AuthenticationManager，也可以有局部AuthenticationManager。全局的 AuthenticationManager 用来对全局认证进行处理，局部的 AuthenticationManager 用
来对某些特殊资源认证处理。当然无论是全局认证管理器还是局部认证管理器都是由 ProviderManger 进行实现。 每一个ProviderManger 中都代理一个 AuthenticationProvider 的列表，列表中每一个实现代表一种身份认证方式。认证时底
层数据源需要调用 UserDetailService 来实现

配置全局 AuthenticationManager

https://spring.io/guides/topicals/spring-security-architecture

- 默认的全局 AuthenticationManager

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter
{
  @Autowired
  public void initialize(AuthenticationManagerBuilder builder) {
    //builder..
  }
}
```

- springboot 对 security 进行自动配置时自动在工厂中创建一个全局AuthenticationManager

总结

1. 默认自动配置创建全局AuthenticationManager 默认找当前项目中是否存在自定义 UserDetailService 实例 自动将当前项目 UserDetailService 实例设置为数据源
2. 默认自动配置创建全局AuthenticationManager 在工厂中使用时直接在代码中注入即可

- 自定义全局 AuthenticationManager

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter
{
  @Override
  public void configure(AuthenticationManagerBuilder builder) {
    //builder ....
  }
}
```

- 自定义全局 AuthenticationManager

总结

1. 一旦通过 configure 方法自定义 AuthenticationManager实现 就回将工厂中自动配置 AuthenticationManager 进行覆盖
2. 一旦通过 configure 方法自定义 AuthenticationManager实现 需要在实现中指定认证数据源对象 UserDetaiService 实例
3. 一旦通过 configure 方法自定义 AuthenticationManager实现 这种方式创建 AuthenticationManager对象工厂内部本地一个 AuthenticationManager 对象 不允许在其他自定义组件中进行注入

- 用来在工厂中暴露自定义AuthenticationManager 实例

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter
{

    //1.自定义AuthenticationManager  推荐  并没有在工厂中暴露出来
    @Override
    public void configure(AuthenticationManagerBuilder builder) throws
Exception {
        System.out.println("自定义AuthenticationManager: " + builder);
        builder.userDetailsService(userDetailsService());
    }

    //作用: 用来将自定义AuthenticationManager在工厂中进行暴露,可以在任何位置注入
    @Override
    @Bean
```

```
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}

```

自定义内存数据源

```
@Configuration
public class WebSecurityConfig {

    @Bean
    public UserDetailsService userDetailsService(){
        UserDetails user = User.withUsername("admin").password("{noop}123").roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index")
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .successHandler(new LoginSuccessHandler())
                .failureHandler(new LoginFailureHandler())
                .and().logout().logoutSuccessHandler(new LogoutHandler())
                .and().userDetailsService(userDetailsService());
        return http.csrf().disable().build();
    }
}
```

自定义数据库数据源

设计表结构

```
-- 用户表
CREATE TABLE `user`
(
    `id`                    int(11) NOT NULL AUTO_INCREMENT,
    `username`              varchar(32)  DEFAULT NULL,
    `password`              varchar(255) DEFAULT NULL,
    `enabled`               tinyint(1) DEFAULT NULL,
    `accountNonExpired`     tinyint(1) DEFAULT NULL,
    `accountNonLocked`      tinyint(1) DEFAULT NULL,
    `credentialsNonExpired` tinyint(1) DEFAULT NULL,
```

```
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- 角色表
CREATE TABLE `role`
(
    `id`      int(11) NOT NULL AUTO_INCREMENT,
    `name`    varchar(32) DEFAULT NULL,
    `name_zh` varchar(32) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- 用户角色关系表
CREATE TABLE `user_role`
(
    `id`  int(11) NOT NULL AUTO_INCREMENT,
    `uid` int(11) DEFAULT NULL,
    `rid` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY   `uid` (`uid`),
    KEY   `rid` (`rid`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;
```

插入测试数据

```
-- 插入用户数据
BEGIN;
  INSERT INTO `user`
  VALUES (1, 'root', '{noop}123', 1, 1, 1, 1);
  INSERT INTO `user`
  VALUES (2, 'admin', '{noop}123', 1, 1, 1, 1);
  INSERT INTO `user`
  VALUES (3, 'cheny', '{noop}123', 1, 1, 1, 1);
COMMIT;

-- 插入角色数据
BEGIN;
  INSERT INTO `role`
  VALUES (1, 'ROLE_product', '商品管理员');
  INSERT INTO `role`
  VALUES (2, 'ROLE_admin', '系统管理员');
  INSERT INTO `role`
  VALUES (3, 'ROLE_user', '用户管理员');
COMMIT;

-- 插入用户角色数据
BEGIN;
  INSERT INTO `user_role`
  VALUES (1, 1, 1);
  INSERT INTO `user_role`
  VALUES (2, 1, 2);
  INSERT INTO `user_role`
  VALUES (3, 2, 2);
  INSERT INTO `user_role`
  VALUES (4, 3, 3);
```

```
COMMIT;
```

项目中引入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.0</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.7</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

配置 springboot 配置文件

```
spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/security?useUnicode=true&useSSL=false&characterEncoding=utf8&serverTimezone=GMT%2B8&allowMultiQueries=true
    username: root
    password: root

mybatis:
  mapper-locations: mapper/*Mapper.xml
  type-aliases-package: com.yang.entity
```

创建 entity

```
@Data
public class User implements UserDetails {

    private Integer id;
    private String username;
    private String password;
    private Boolean enabled;
    private Boolean accountNonExpired;
    private Boolean accountNonLocked;
    private Boolean credentialsNonExpired;
    private List<Role> roles = new ArrayList<>();


    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
        roles.forEach(role->grantedAuthorities.add(new
SimpleGrantedAuthority(role.getName())));
        return grantedAuthorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return accountNonExpired;
    }

    @Override
    public boolean isAccountNonLocked() {
        return accountNonLocked;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return credentialsNonExpired;
    }

    @Override
    public boolean isEnabled() {
        return enabled;
    }
}
```

```java
@Data
public class Role {

    private Integer id;
    private String name;
    private String nameZh;
}
```

创建 UserMapper 接口，编写sql语句

```java
@Mapper
public interface UserMapper {

    //根据用户名查询用户
    User loadUserByUsername(String username);

    //根据用户id查询角色
    List<Role> getRolesByUid(Integer uid);
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.yang.mapper.UserMapper">
    <!--查询单个-->
    <select id="loadUserByUsername" resultType="com.yang.entity.User">
        select id,
               username,
               password,
               enabled,
               accountNonExpired,
               accountNonLocked,
               credentialsNonExpired
        from user
        where username = #{username}
    </select>

    <!--查询指定行数据-->
    <select id="getRolesByUid" resultType="com.yang.entity.Role">
        select r.id,
               r.name,
               r.name_zh nameZh
        from role r,
             user_role ur
        where r.id = ur.rid
          and ur.uid = #{uid}
    </select>
</mapper>
```

创建 service

```java
public interface UserService {
```

```
    UserDetails loadUserByUsername(String username);
}


@Service
public class UserServiceImpl implements UserService {

    private final UserMapper userMapper;

    @Autowired
    public UserServiceImpl(UserMapper userMapper) {
        this.userMapper = userMapper;
    }

    @Override
    public UserDetails loadUserByUsername(String username) {
        User user = userMapper.loadUserByUsername(username);
        if(ObjectUtils.isEmpty(user)){
            throw new RuntimeException("用户不存在");
        }
        user.setRoles(userMapper.getRolesByUid(user.getId()));
        return user;
    }
}
```

创建 UserDetailsService

```
@Component
public class UserDetailService implements UserDetailsService {

    private final UserService userService;

    public UserDetailService(UserService userService) {
        this.userService = userService;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return userService.loadUserByUsername(username);
    }
}
```

配置 authenticationManager 使用自定义UserDetailService

```
@Configuration
public class SecurityWebConfig {

    private final UserDetailService userDetailService;

    public SecurityWebConfig(UserDetailService userDetailService) {
        this.userDetailService = userDetailService;
```

```
    }

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }


    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index")
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .successHandler(new LoginSuccessHandler())
                .failureHandler(new LoginFailureHandler())
                .and().logout().logoutSuccessHandler(new LogoutHandler()) //
注销登入处理器
                .and().userDetailsService(userDetailService); // 自定义数据源
        return http.csrf().disable().build();
    }
}
```

添加验证码

```
<dependency>
   <groupId>com.github.penggle</groupId>
   <artifactId>kaptcha</artifactId>
   <version>2.3.2</version>
</dependency>
```

生成验证码

```
@Configuration
public class KaptchaConfig {

    @Bean
    public Producer kaptcha() {
        Properties properties = new Properties();
        properties.setProperty("kaptcha.image.width", "150");
        properties.setProperty("kaptcha.image.height", "50");
        properties.setProperty("kaptcha.textproducer.char.string",
"0123456789");
        properties.setProperty("kaptcha.textproducer.char.length", "4");
        Config config = new Config(properties);
        DefaultKaptcha defaultKaptcha = new DefaultKaptcha();
        defaultKaptcha.setConfig(config);
```

```
        return defaultKaptcha;
    }
}
```

```
@RestController
public class KaptchaController {
    private final Producer producer;

    public KaptchaController(Producer producer) {
        this.producer = producer;
    }

    @GetMapping("/vc.png")
    public String getVerifyCode(HttpSession session) throws IOException {
        //1.生成验证码
        String code = producer.createText();
        session.setAttribute("kaptcha", code);//可以更换成 redis 实现
        BufferedImage bi = producer.createImage(code);
        //2.写入内存
        FastByteArrayOutputStream fos = new FastByteArrayOutputStream();
        ImageIO.write(bi, "png", fos);
        //3.生成 base64
        return Base64.encodeBase64String(fos.toByteArray());
    }
}
```

定义验证码异常类

```
public class KaptchaNotMatchException extends AuthenticationException {

    public KaptchaNotMatchException(String msg) {
        super(msg);
    }

    public KaptchaNotMatchException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

在自定义LoginKaptchaFilter中加入验证码验证

```
/**
 * @Author: chenyang
 * @DateTime: 2023/2/27 10:14
 * @Description: 自定义过滤器
 */
public class LoginKaptchaFilter extends UsernamePasswordAuthenticationFilter
{

    public static final String FORM_CAPTCHA_KEY = "captcha";

    private String kaptchaParameter = FORM_CAPTCHA_KEY;
```

```
    public String getKaptchaParameter() {
        return kaptchaParameter;
    }

    public void setKaptchaParameter(String kaptchaParameter) {
        this.kaptchaParameter = kaptchaParameter;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
        if (!request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException("Authentication method not supported: " + request.getMethod());
        }
        try {
            //1.获取请求数据
            Map<String, String> userInfo = new ObjectMapper().readValue(request.getInputStream(), Map.class);
            String kaptcha = userInfo.get(getKaptchaParameter());//用来获取数据中验证码
            String username = userInfo.get(getUsernameParameter());//用来接收用户名
            String password = userInfo.get(getPasswordParameter());//用来接收密码
            //2.获取 session 中验证码
            String sessionVerifyCode = (String) request.getSession().getAttribute(FORM_CAPTCHA_KEY);
            if (!ObjectUtils.isEmpty(kaptcha) && !ObjectUtils.isEmpty(sessionVerifyCode) &&
                    kaptcha.equalsIgnoreCase(sessionVerifyCode)) {
                //3.获取用户名 和密码认证
                UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(username, password);
                setDetails(request, authRequest);
                return this.getAuthenticationManager().authenticate(authRequest);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        throw new KaptchaNotMatchException("验证码不匹配!");
    }
}
```

配置

```
@Configuration
public class WebSecurityConfig {

    private final UserDetailService userDetailService;

    public WebSecurityConfig(UserDetailService userDetailService) {
        this.userDetailService = userDetailService;
```

```
    }


    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public LoginKaptchaFilter loginKaptchaFilter(AuthenticationManager authenticationManager) {
        LoginKaptchaFilter filter = new LoginKaptchaFilter();
        //1.认证 url
        filter.setFilterProcessesUrl("/doLogin");

        //2.认证 接收参数
        filter.setUsernameParameter("username");
        filter.setPasswordParameter("pwd");
        filter.setKaptchaParameter("kaptcha");

        //3.指定认证管理器
        filter.setAuthenticationManager(authenticationManager);

        // 4.指定成功/失败时处理
        filter.setAuthenticationSuccessHandler(new LoginSuccessHandler());
        filter.setAuthenticationFailureHandler(new LoginFailureHandler());

        return filter;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index", "/vc.png")
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .and().logout().logoutSuccessHandler(new LogoutHandler()) // 注销登入处理器
                .and().exceptionHandling().authenticationEntryPoint(new UnAuthenticationHandler()) // 未认证处理器
                .and().userDetailsService(userDetailService) // 自定义数据源

.addFilterBefore(loginKaptchaFilter(http.getSharedObject(AuthenticationManager.class)), UsernamePasswordAuthenticationFilter.class); // 自定义过滤器
        return http.csrf().disable().build();
    }
}
```

自定义认证异常处理类

```
/**
```

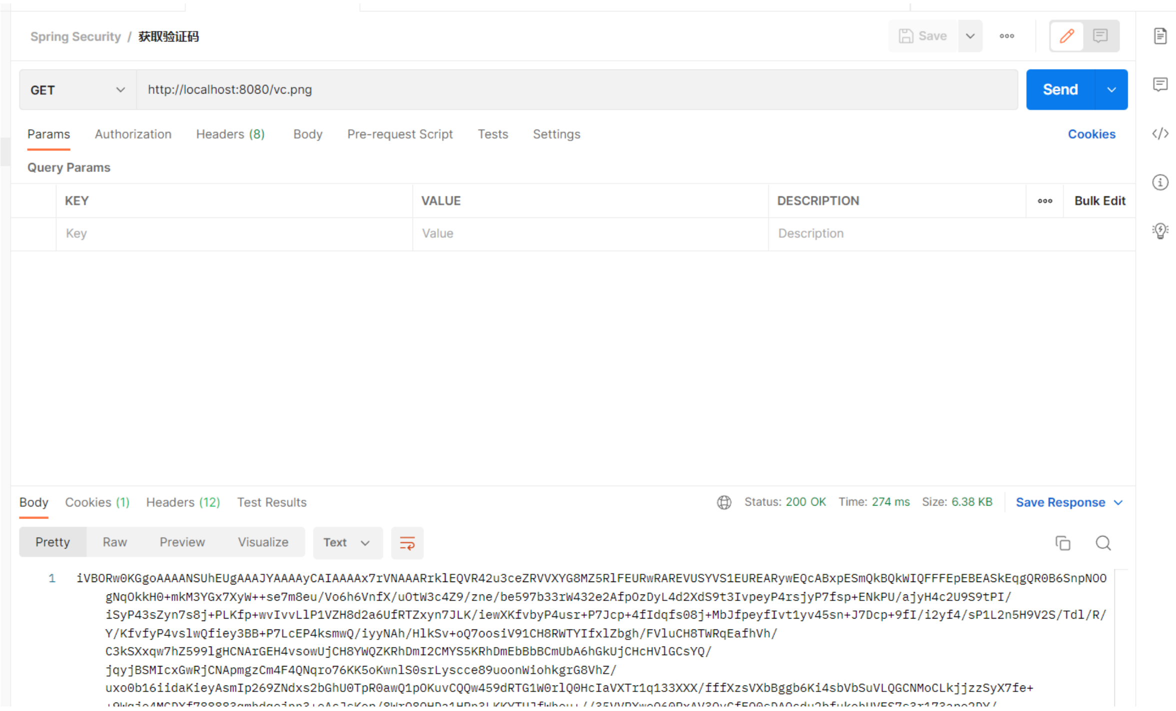
```java
 * @Author: chenyang
 * @DateTime: 2023/2/27 11:27
 * @Description: 未认证时请求处理器
 */
public class UnAuthenticationHandler implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) throws IOException {
        response.setContentType("application/json;charset=UTF-8");
        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.getWriter().println("必须认证之后才能访问!");
    }
}
```

https://juejin.cn/post/7050071382041821197

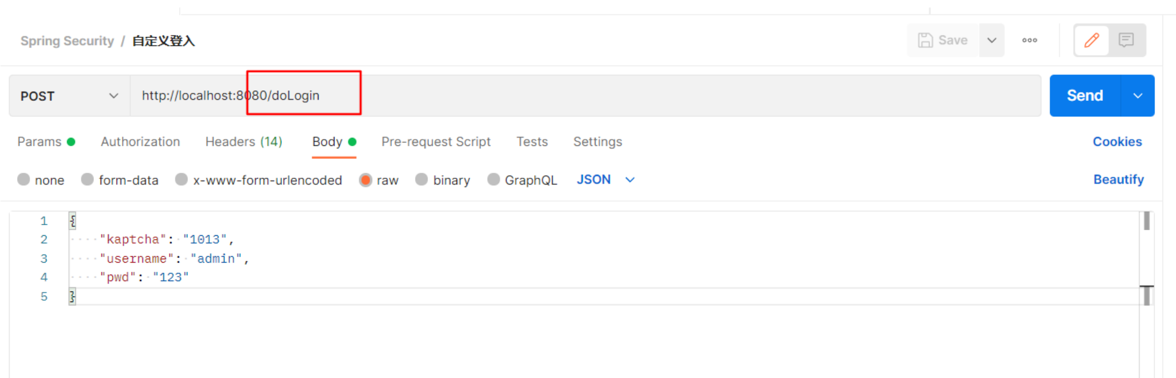
https://blog.csdn.net/hou_ge/article/details/120435303

测试验证

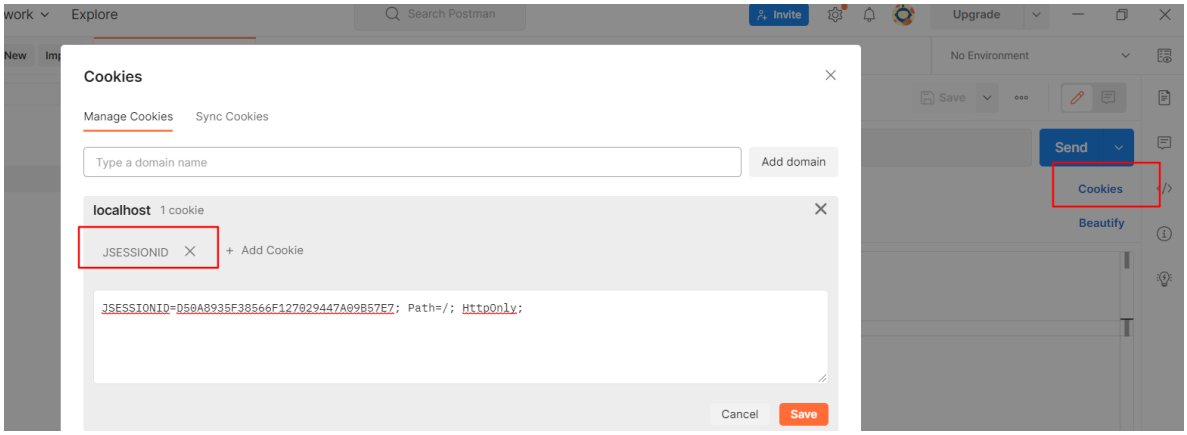
调用接口获取图片的Base64 编码，再将编码转换成图片

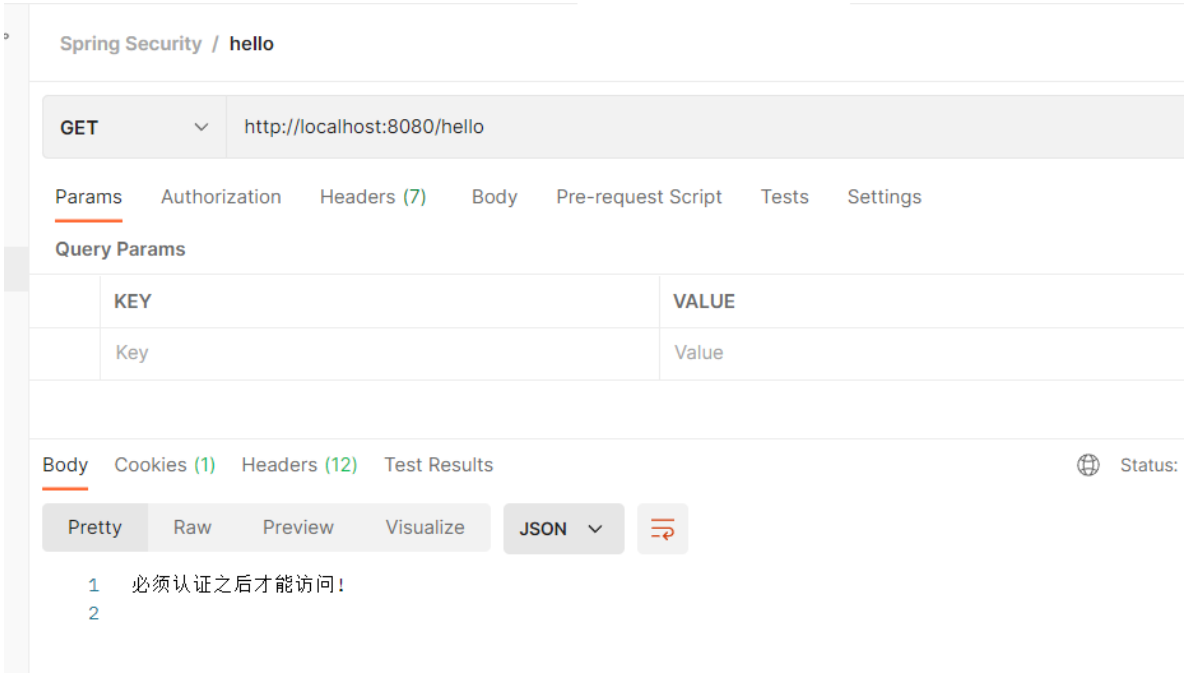


登入



调用获取验证码接口时会自动保存session

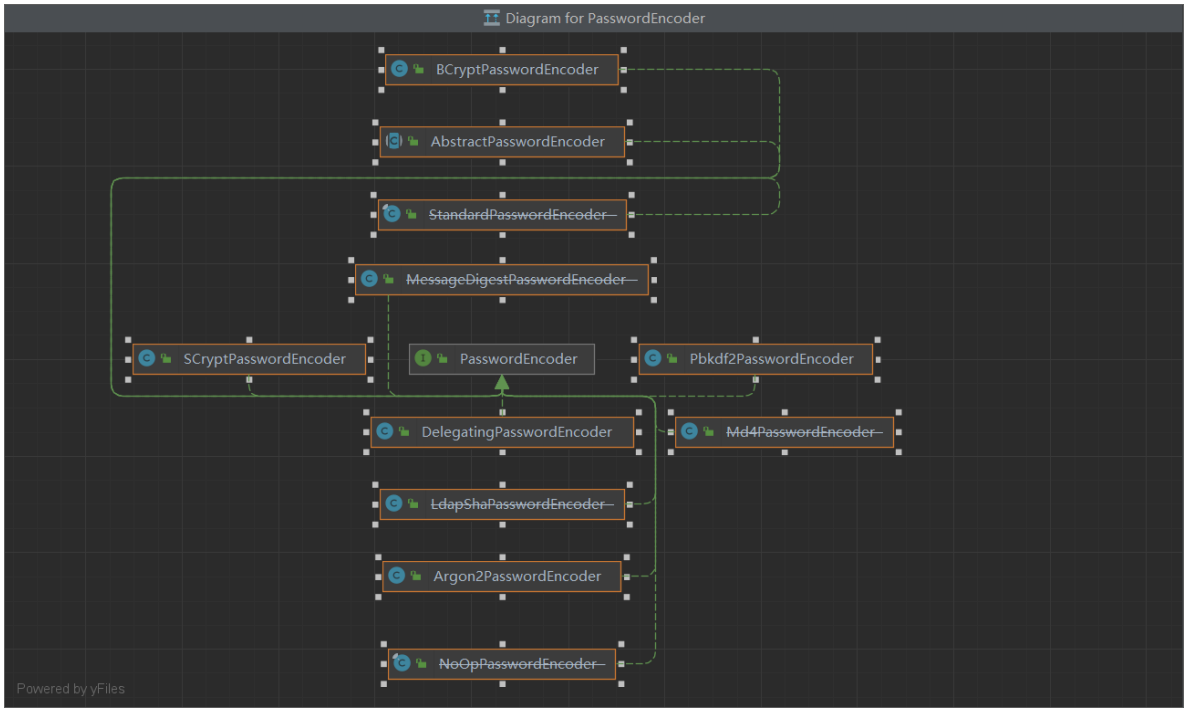


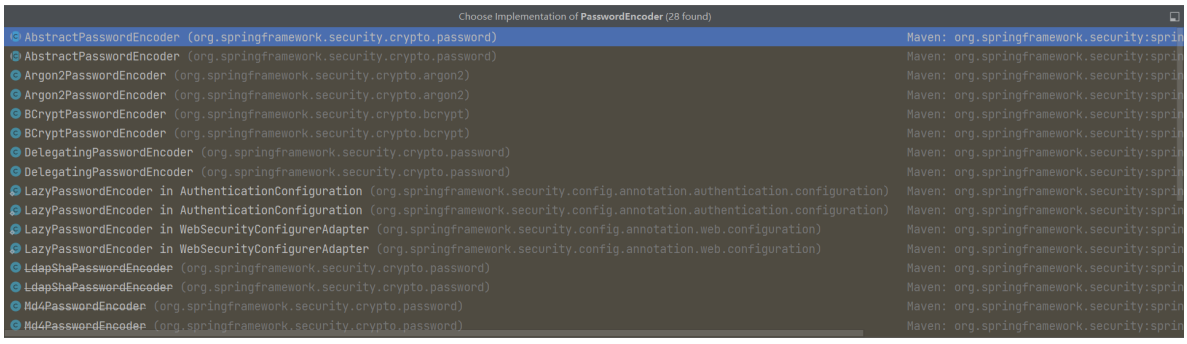


密码加密

实际密码比较是由PasswordEncoder完成的，因此只需要使用PasswordEncoder 不同实现就可以实现不同方式加密。

```java
public interface PasswordEncoder {
    // 进行明文加密
    String encode(CharSequence rawPassword);

    // 比较密码
    boolean matches(CharSequence rawPassword, String encodedPassword);

    // 密码升级
    default boolean upgradeEncoding(String encodedPassword) {
        return false;
    }
}
```





DelegatingPasswordEncoder

根据上面 PasswordEncoder的介绍，可能会以为 Spring security 中默认的密码加密方案应该是四种自适应单向加密函数中的一种，其实不然，在 spring Security 5.0之后，默认的密码加密方案其实是 DelegatingPasswordEncoder。从名字上来看，DelegatingPaswordEncoder 是一个代理类，而并非一种全新的密码加密方案，DeleggtinePasswordEncoder 主要用来代理上面介绍的不同的密码加密方案。为什么采DelegatingPasswordEncoder 而不是某一个具体加密方式作为默认的密码加密方案呢？主要考虑了如下两方面的因素：

- 兼容性：使用 DelegatingPasswrordEncoder 可以帮助许多使用旧密码加密方式的系统顺利迁移到 Spring security 中，它允许在同一个系统中同时存在多种不同的密码加密方案。

- 便捷性：密码存储的最佳方案不可能一直不变，如果使用 DelegatingPasswordEncoder作为默认的密码加密方案，当需要修改加密方案时，只需要修改很小一部分代码就可以实现。

DelegatingPasswordEncoder源码

```
public class DelegatingPasswordEncoder implements PasswordEncoder {
  ....
}
```

PasswordEncoderFactories源码

```
public final class PasswordEncoderFactories {

    private PasswordEncoderFactories() {
    }


    @SuppressWarnings("deprecation")
    public static PasswordEncoder createDelegatingPasswordEncoder() {
        String encodingId = "bcrypt";
        Map<String, PasswordEncoder> encoders = new HashMap<>();
        encoders.put(encodingId, new BCryptPasswordEncoder());
        encoders.put("ldap", new
org.springframework.security.crypto.password.LdapShaPasswordEncoder());
        encoders.put("MD4", new
org.springframework.security.crypto.password.Md4PasswordEncoder());
        encoders.put("MD5", new
org.springframework.security.crypto.password.MessageDigestPasswordEncoder("MD5"));
        encoders.put("noop",
org.springframework.security.crypto.password.NoOpPasswordEncoder.getInstance());
        encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
        encoders.put("scrypt", new SCryptPasswordEncoder());
        encoders.put("SHA-1", new
org.springframework.security.crypto.password.MessageDigestPasswordEncoder("SHA-1"));
        encoders.put("SHA-256",
                new
org.springframework.security.crypto.password.MessageDigestPasswordEncoder("SHA-256"));
        encoders.put("sha256", new
org.springframework.security.crypto.password.StandardPasswordEncoder());
        encoders.put("argon2", new Argon2PasswordEncoder());
        return new DelegatingPasswordEncoder(encodingId, encoders);
    }

}
```

使用 PasswordEncoder

查看WebSecurityConfigurerAdapter类中源码

```
static class LazyPasswordEncoder implements PasswordEncoder {
        private ApplicationContext applicationContext;
        private PasswordEncoder passwordEncoder;

        LazyPasswordEncoder(ApplicationContext applicationContext) {
            this.applicationContext = applicationContext;
        }

        public String encode(CharSequence rawPassword) {
            return this.getPasswordEncoder().encode(rawPassword);
        }

        public boolean matches(CharSequence rawPassword, String
encodedPassword) {
            return this.getPasswordEncoder().matches(rawPassword,
encodedPassword);
        }

        public boolean upgradeEncoding(String encodedPassword) {
            return
this.getPasswordEncoder().upgradeEncoding(encodedPassword);
        }

        private PasswordEncoder getPasswordEncoder() {
            if (this.passwordEncoder != null) {
                // 若指定的 passwordEncoder 不为空则使用指定的 passwordEncoder
                return this.passwordEncoder;
            } else {
                // 使用默认的 DelegatingPasswordEncoder
                PasswordEncoder passwordEncoder =
(PasswordEncoder)AuthenticationConfiguration.getBeanOrNull(this.applicationC
ontext, PasswordEncoder.class);
                if (passwordEncoder == null) {
                    passwordEncoder =
PasswordEncoderFactories.createDelegatingPasswordEncoder();
                }

                this.passwordEncoder = passwordEncoder;
                return passwordEncoder;
            }
        }

        public String toString() {
            return this.getPasswordEncoder().toString();
        }
    }
```

密码加密实战

使用固定密码加密方案

```
    @Bean
    public PasswordEncoder BcryptPasswordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailsService(){
        UserDetails user =
User.withUsername("admin").password("$2a$10$WGFkRsZC0kzafTKOPcWONeLvNvg2jqd3
U09qd5gjJGSHE5bOyoy6a").roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user);
    }
```

使用灵活密码加密方案 推荐

```
    @Bean
    public UserDetailsService userDetailsService(){
        UserDetails user =
User.withUsername("admin").password("$2a$10$WGFkRsZC0kzafTKOPcWONeLvNvg2jqd3U
09qd5gjJGSHE5bOyoy6a").roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user);
    }
```

密码自动升级

```
@Mapper
public interface UserMapper {

    //根据用户名查询用户
    User loadUserByUsername(String username);

    Integer updatePassword(@Param("username") String username,
@Param("password") String password);
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.yang.mapper.UserMapper">

    <update id="updatePassword">
        update `user` set password = #{password}
        where username= #{username}
    </update>

    <!--查询单个-->
    <select id="loadUserByUsername" resultType="com.yang.entity.User">
```

```
        select id,
               username,
               password,
               enabled,
               accountNonExpired,
               accountNonLocked,
               credentialsNonExpired
        from user
        where username = #{username}
    </select>

</mapper>
```

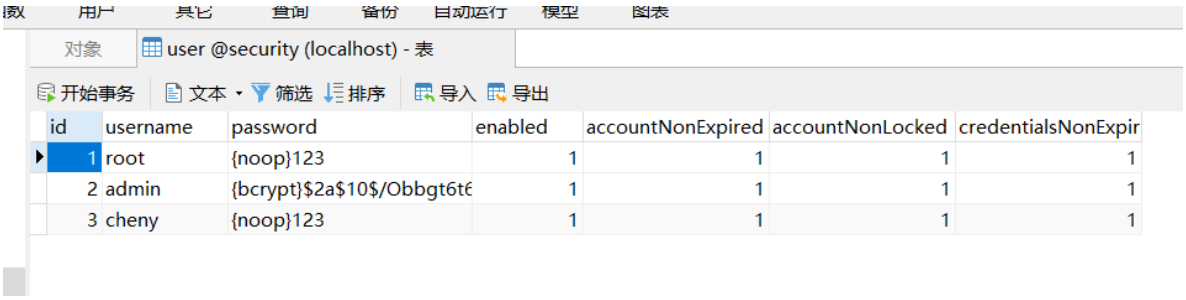
```
public interface UserService {

    UserDetails loadUserByUsername(String username);

    Integer updateUser(String username, String password);
}
```

```
@Service
public class UserServiceImpl implements UserService {

    private final UserMapper userMapper;

    @Autowired
    public UserServiceImpl(UserMapper userMapper) {
        this.userMapper = userMapper;
    }

    @Override
    public UserDetails loadUserByUsername(String username) {
        User user = userMapper.loadUserByUsername(username);
        if(ObjectUtils.isEmpty(user)){
            throw new RuntimeException("用户不存在");
        }
        user.setRoles(userMapper.getRolesByUid(user.getId()));
        return user;
    }

    @Override
    public Integer updateUser(String username, String password) {
        return userMapper.updatePassword(username, password);
    }
}
```

```
@Component
public class UserDetailService implements UserDetailsService,
UserDetailsPasswordService {

    private final UserService userService;

    public UserDetailService(UserService userService) {
        this.userService = userService;
    }
```

```
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return userService.loadUserByUsername(username);
    }

    @Override
    public UserDetails updatePassword(UserDetails user, String newPassword)
{
        Integer updateRow = userService.updateUser(user.getUsername(),
newPassword);
        if (updateRow == 1){
            ((User) user).setPassword(newPassword);
        }
        return user;
    }
}
```

```
@Configuration
public class WebSecurityConfig {

    private final UserDetailService userDetailService;

    public WebSecurityConfig(UserDetailService userDetailService) {
        this.userDetailService = userDetailService;
    }

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }


    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index")
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .and().userDetailsService(userDetailService); // 自定义数据源
        return http.csrf().disable().build();
    }
}
```



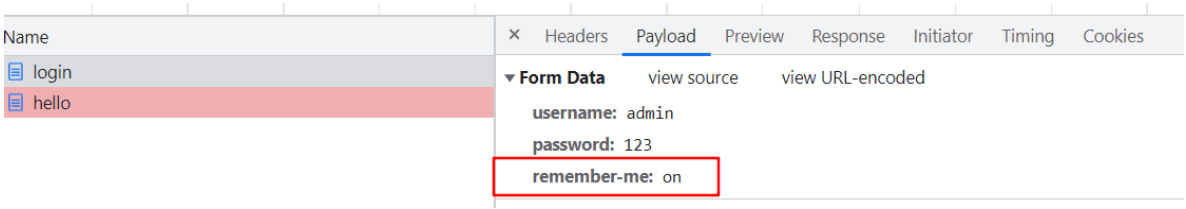
RemberMe

RememberMe 是一种服务器端的行为。传统的登录方式基于 Session会话，一旦用户的会话超时过期，就要再次登录，这样太过于烦琐。如果能有一种机制，让用户会话过期之后，还能继续保持认证状态，就会方便很多，RememberMe 就是为了解决这一需求而生的。

实现思路就是通过 Cookie 来记录当前用户身份。当用户登录成功之后，会通过一定算法，将用户信息、时间戳等进行加密，加密完成后，通过响应头带回前端存储在cookie中，当浏览器会话过期之后，如果再次访问该网站，会自动将 Cookie 中的信息发送给服务器，服务器对 Cookie中的信息进行校验分析，进而确定出用户的身份，Cookie中所保存的用户信息也是有时效的，例如三天、一周等。

基本使用

```
   @Bean
    public UserDetailsService userDetailsService(){
        UserDetails user = User.withUsername("admin").password("
{noop}123").roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index").permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .and().rememberMe() // 开启 记住我
                .userDetailsService(userDetailsService());
        return http.csrf().disable().build();
    }
```



原理分析

如果自定义登录页面开启 RememberMe 功能应该多加入一个一样的请求参数就可以啦。该请求会被`RememberMeAuthenticationFilter`进行拦截然后自动登录

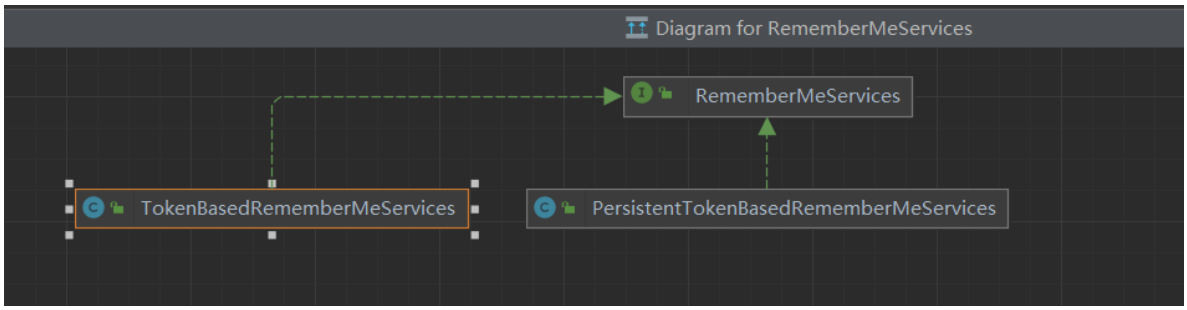
```
private void doFilter(HttpServletRequest request, HttpServletResponse
response, FilterChain chain)
            throws IOException, ServletException {
```

```
        if (SecurityContextHolder.getContext().getAuthentication() != null)
{
            this.logger.debug(LogMessage
                    .of(() -> "SecurityContextHolder not populated with
remember-me token, as it already contained: '"
                            +
SecurityContextHolder.getContext().getAuthentication() + "'"));
            chain.doFilter(request, response);
            return;
        }
        Authentication rememberMeAuth =
this.rememberMeServices.autoLogin(request, response);
        if (rememberMeAuth != null) {
            // Attempt authenticaton via AuthenticationManager
            try {
                rememberMeAuth =
this.authenticationManager.authenticate(rememberMeAuth);
                // Store to SecurityContextHolder
                SecurityContext context =
SecurityContextHolder.createEmptyContext();
                context.setAuthentication(rememberMeAuth);
                SecurityContextHolder.setContext(context);
                onSuccessfulAuthentication(request, response,
rememberMeAuth);
                this.logger.debug(LogMessage.of(() -> "SecurityContextHolder
populated with remember-me token: '"
                        +
SecurityContextHolder.getContext().getAuthentication() + "'"));
                this.securityContextRepository.saveContext(context, request,
response);
                if (this.eventPublisher != null) {
                    this.eventPublisher.publishEvent(new
InteractiveAuthenticationSuccessEvent(

SecurityContextHolder.getContext().getAuthentication(), this.getClass()));
                }
                if (this.successHandler != null) {
                    this.successHandler.onAuthenticationSuccess(request,
response, rememberMeAuth);
                    return;
                }
            }
            catch (AuthenticationException ex) {
                this.logger.debug(LogMessage
                        .format("SecurityContextHolder not populated with
remember-me token, as AuthenticationManager "
                                + "rejected Authentication returned by
RememberMeServices: '%s'; "
                                + "invalidating remember-me token",
rememberMeAuth),
                        ex);
                this.rememberMeServices.loginFail(request, response);
                onUnsuccessfulAuthentication(request, response, ex);
            }
        }
        chain.doFilter(request, response);
    }
```

- 请求到达过滤器之后，首先判断 SecurityContextHolder 中是否有值，没值的话表示用户尚未登录，此时调用 autoLogin 方法进行自动登录。
- 当自动登录成功后返回的rememberMeAuth 不为null 时，表示自动登录成功，此时调用 authenticate 方法对 key 进行校验，并且将登录成功的用户信息保存到 SecurityContextHolder 对象中，然后调用登录成功回调，并发布登录成功事件。需要注意的是，登录成功的回调并不包含 RememberMeServices 中的 1oginSuccess 方法。
- 如果自动登录失败，则调用 remenberMeServices.loginFail方法处理登录失败回调。onUnsuccessfulAuthentication 和 onSuccessfulAuthentication 都是该过滤器中定义的空方法，并没有任何实现这就是 RememberMeAuthenticationFilter 过滤器所做的事情，成功将 RememberMeServices的服务集成进来。

RememberMeServices

```
public interface RememberMeServices {


    // 从请求中提取出需要的参数，完成自动登录功能。
    Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);

    // 自动登录失败的回调
    void loginFail(HttpServletRequest request, HttpServletResponse response);

    // 自动登录成功的回调
    void loginSuccess(HttpServletRequest request, HttpServletResponse response,
            Authentication successfulAuthentication);

}
```



TokenBasedRememberMeServices

在开启记住我后如果没有加入额外配置默认实现就是由TokenBasedRememberMeServices进行的实现。查看这个类源码中 processAutoLoginCookie 方法实现:

```
@Override
    protected UserDetails processAutoLoginCookie(String[] cookieTokens, HttpServletRequest request,
            HttpServletResponse response) {
        if (cookieTokens.length != 3) {
```

```
            throw new InvalidCookieException(
                    "Cookie token did not contain 3" + " tokens, but
contained '" + Arrays.asList(cookieTokens) + "'");
        }
        long tokenExpiryTime = getTokenExpiryTime(cookieTokens);
        if (isTokenExpired(tokenExpiryTime)) {
            throw new InvalidCookieException("Cookie token[1] has expired
(expired on '" + new Date(tokenExpiryTime)
                    + "'; current time is '" + new Date() + "')");
        }
        // Check the user exists. Defer lookup until after expiry time
checked, to
        // possibly avoid expensive database call.
        UserDetails userDetails =
getUserDetailsService().loadUserByUsername(cookieTokens[0]);
        Assert.notNull(userDetails, () -> "UserDetailsService " +
getUserDetailsService()
                + " returned null for username " + cookieTokens[0] + ". " +
"This is an interface contract violation");
        // Check signature of token matches remaining details. Must do this
after user
        // lookup, as we need the DAO-derived password. If efficiency was a
major issue,
        // just add in a UserCache implementation, but recall that this
method is usually
        // only called once per HttpSession - if the token is valid, it will
cause
        // SecurityContextHolder population, whilst if invalid, will cause
the cookie to
        // be cancelled.
        String expectedTokenSignature = makeTokenSignature(tokenExpiryTime,
userDetails.getUsername(),
                userDetails.getPassword());
        if (!equals(expectedTokenSignature, cookieTokens[2])) {
            throw new InvalidCookieException("Cookie token[2] contained
signature '" + cookieTokens[2]
                    + "' but expected '" + expectedTokenSignature + "'");
        }
        return userDetails;
    }
```

processAutoLoginCookie 方法主要用来验证 Cookie 中的令牌信息是否合法：

1. 首先判断 cookieTokens 长度是否为了，不为了说明格式不对，则直接抛出异常。
2. 从cookieTokens 数组中提取出第 1项，也就是过期时间，判断令牌是否过期，如果已经过期，则抛出异常。
3. 根据用户名 （cookieTokens 数组的第。项）查询出当前用户对象。
4. 调用 makeTokenSignature 方法生成一个签名，签名的生成过程如下：首先将用户名、令牌过期时间、用户密码以及 key 组成一个字符串，中间用“：”隔开，然后通过 MD5 消息摘要算法对该字符串进行加密，并将加密结果转为一个字符串返回。
5. 判断第4 步生成的签名和通过 Cookie 传来的签名是否相等（即 cookieTokens 数组的第2项），如果相等，表示令牌合法，则直接返回用户对象，否则抛出异常。

```
@Override
```

```
    public void onLoginSuccess(HttpServletRequest request,
HttpServletResponse response,
            Authentication successfulAuthentication) {
        String username = retrieveUserName(successfulAuthentication);
        String password = retrievePassword(successfulAuthentication);
        // If unable to find a username and password, just abort as
        // TokenBasedRememberMeServices is
        // unable to construct a valid token in this case.
        if (!StringUtils.hasLength(username)) {
            this.logger.debug("Unable to retrieve username");
            return;
        }
        if (!StringUtils.hasLength(password)) {
            UserDetails user =
getUserDetailsService().loadUserByUsername(username);
            password = user.getPassword();
            if (!StringUtils.hasLength(password)) {
                this.logger.debug("Unable to obtain password for user: " +
username);
                return;
            }
        }
        int tokenLifetime = calculateLoginLifetime(request,
successfulAuthentication);
        long expiryTime = System.currentTimeMillis();
        // SEC-949
        expiryTime += 1000L * ((tokenLifetime < 0) ? TWO_WEEKS_S :
tokenLifetime);
        String signatureValue = makeTokenSignature(expiryTime, username,
password);
        setCookie(new String[] { username, Long.toString(expiryTime),
signatureValue }, tokenLifetime, request,
                response);
        if (this.logger.isDebugEnabled()) {
            this.logger.debug(
                    "Added remember-me cookie for user '" + username + "',
expiry: '" + new Date(expiryTime) + "'");
        }
    }
```

1. 在这个回调中，首先获取用户经和密码信息，如果用户密码在用户登录成功后从successfulAuthentication对象中擦除，则从数据库中重新加载出用户密码。
2. 计算出令牌的过期时间，令牌默认有效期是两周。
3. 根据令牌的过期时间、用户名以及用户密码，计算出一个签名。
4. 调用 setCookie 方法设置 Cookie， 第一个参数是一个数组，数组中一共包含三项。用户名、过期时间以及签名，在setCookie 方法中会将数组转为字符串，并进行 Base64编码后响应给前端。

总结

当用户通过用户名/密码的形式登录成功后，系统会根据用户的用户名、密码以及令牌的过期时间计算出一个签名，这个签名使用 MD5 消息摘要算法生成，是不可逆的。然后再将用户名、令牌过期时间以及签名拼接成一个字符串，中间用“:” 隔开，对拼接好的字符串进行Base64 编码，然后将编码后的结果返回到前端，也就是我们在浏览器中看到的令牌。当会话过期之后，访问系统资源时会自动携带上Cookie中的令牌，服务端拿到 Cookie中的令牌后，先进行 Bae64解码，解码后分别提取出令牌中的三项数据：接着根据令牌中的数据判断令牌是否已经过期，如果没有过期，则根据令牌中的用户名查询出

用户信息：接着再计算出一个签名和令牌中的签名进行对比，如果一致，表示会牌是合法令牌，自动登录成功，否则自动登录失败。

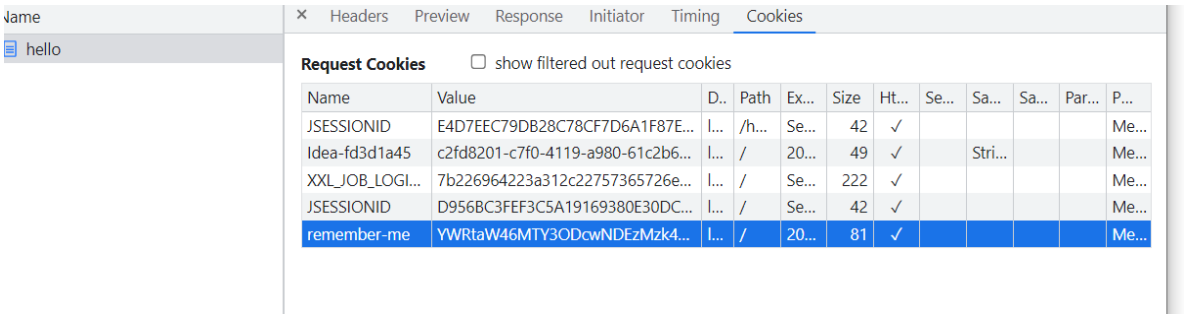


image-20220319124115432

内存令牌

PersistentTokenBasedRememberMeServices

```
protected UserDetails processAutoLoginCookie(String[] cookieTokens, HttpServletRequest request, HttpServletResponse response) {
        if (cookieTokens.length != 2) {
            throw new InvalidCookieException("Cookie token did not contain 2 tokens, but contained '" + Arrays.asList(cookieTokens) + "'");
        } else {
            String presentedSeries = cookieTokens[0];
            String presentedToken = cookieTokens[1];
            PersistentRememberMeToken token = this.tokenRepository.getTokenForSeries(presentedSeries);
            if (token == null) {
                throw new RememberMeAuthenticationException("No persistent token found for series id: " + presentedSeries);
            } else if (!presentedToken.equals(token.getTokenValue())) {
                this.tokenRepository.removeUserTokens(token.getUsername());
                throw new CookieTheftException(this.messages.getMessage("PersistentTokenBasedRememberMeServices.cookieStolen", "Invalid remember-me token (Series/token) mismatch. Implies previous cookie theft attack."));
            } else if (token.getDate().getTime() + (long)this.getTokenValiditySeconds() * 1000L < System.currentTimeMillis()) {
                throw new RememberMeAuthenticationException("Remember-me login has expired");
            } else {
                this.logger.debug(LogMessage.format("Refreshing persistent login token for user '%s', series '%s'", token.getUsername(), token.getSeries()));
                PersistentRememberMeToken newToken = new PersistentRememberMeToken(token.getUsername(), token.getSeries(), this.generateTokenData(), new Date());

                try {
                    this.tokenRepository.updateToken(newToken.getSeries(), newToken.getTokenValue(), newToken.getDate());
```

```
                    this.addCookie(newToken, request, response);
                } catch (Exception var9) {
                    this.logger.error("Failed to update token: ", var9);
                    throw new RememberMeAuthenticationException("Autologin failed due to data access problem");
                }

                return this.getUserDetailsService().loadUserByUsername(token.getUsername());
            }
        }
    }
```

1. 不同于 TokonBasedRemornberMeServices 中的 processAutologinCookie 方法，这里 cookieTokens 数组的长度为2，第一项是series，第二项是 token。
2. 从cookieTokens数组中分到提取出 series 和 token． 然后根据 series 去内存中查询出一个 PersistentRememberMeToken对象。如果查询出来的对象为null，表示内存中并没有series对应的值，本次自动登录失败。如果查询出来的 token 和从 cookieTokens 中解析出来的token不相同，说明自动登录会牌已经泄漏（恶意用户利用令牌登录后，内存中的token变了），此时移除当前用户的所有自动登录记录并抛出异常。
3. 根据数据库中查询出来的结果判断令牌是否过期，如果过期就抛出异常。
4. 生成一个新的 PersistentRememberMeToken 对象，用户名和series 不变，token 重新生成，date 也使用当前时间。newToken 生成后，根据 series 去修改内存中的 token 和 date(即每次自动登录后都会产生新的 token 和 date)
5. 调用 addCookie 方法添加 Cookie， 在addCookie 方法中，会调用到我们前面所说的 setCookie 方法，但是要注意第一个数组参数中只有两项：series 和 token（即返回到前端的令牌是通过对 series 和 token 进行 Base64 编码得到的）
6. 最后将根据用户名查询用户对象并返回。

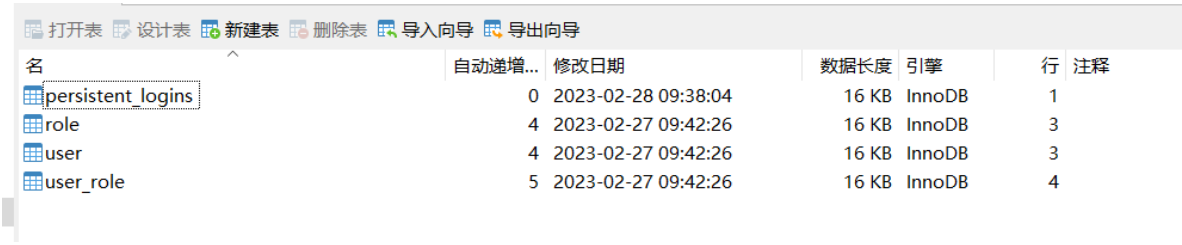
使用内存中令牌实现

```
   @Bean
    public UserDetailsService userDetailsService(){
        UserDetails user = User.withUsername("admin").password("{noop}123").roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public RememberMeServices rememberMeServices() {
        return new PersistentTokenBasedRememberMeServices(
                "key",//参数 1: 自定义一个生成令牌 key 默认 UUID
                userDetailsService(), //参数 2:认证数据源
                new InMemoryTokenRepositoryImpl());//参数 3:令牌存储方式
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index").permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .and().rememberMe()
```

```
                .userDetailsService(userDetailsService())
                .rememberMeServices(rememberMeServices());
        return http.csrf().disable().build();
    }
```

持久化令牌

```
@Configuration
public class WebSecurityConfig {


    private final DataSource dataSource;

    public WebSecurityConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public PersistentTokenRepository persistentTokenRepository(){
        JdbcTokenRepositoryImpl jdbcTokenRepository = new
JdbcTokenRepositoryImpl();
        // 项目启动时创建表。第一次启动后注释掉即可
        jdbcTokenRepository.setCreateTableOnStartup(true);
        jdbcTokenRepository.setDataSource(dataSource);
        return jdbcTokenRepository;
    }

    @Bean
    public UserDetailsService userDetailsService(){
        UserDetails user = User.withUsername("admin").password("
{noop}123").roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user);
    }


    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index").permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .and().rememberMe()
                .userDetailsService(userDetailsService())
                .tokenRepository(persistentTokenRepository());
        return http.csrf().disable().build();
    }

}
```



即使服务器重新启动，依然可以自动登录。

自定义记住我

自定义认证类 LoginFilter

```
1  public class LoginFilter extends UsernamePasswordAuthenticationFilter {
2
3      @Override
4      public Authentication attemptAuthentication(HttpServletRequest request,
   HttpServletResponse response) throws AuthenticationException {
5          System.out.println("========================================");
6          // 1. 判断请求方式
7          if (!request.getMethod().equals("POST")){
8              throw new AuthenticationServiceException("Authentication method
   not supported: " + request.getMethod());
9          }
10
11         // 2.判断是否是 json 格式请求类型
12         if
   (request.getContentType().equalsIgnoreCase(MediaType.APPLICATION_JSON_VALUE)
   ){
13             // 3.从 json 数据中获取用户输入用户名和密码进行认证
   {"uname":"xxx","password":"xxx","remember-me":true}
14             try {
15                 Map<String, String> userInfo = new
   ObjectMapper().readValue(request.getInputStream(), Map.class);
16                 String username = userInfo.get(getUsernameParameter());
17                 String password = userInfo.get(getPasswordParameter());
18                 String rememberValue =
   userInfo.get(AbstractRememberMeServices.DEFAULT_PARAMETER);
19                 if (!ObjectUtils.isEmpty(rememberValue)) {
20
    request.setAttribute(AbstractRememberMeServices.DEFAULT_PARAMETER,
   rememberValue);
21                 }
22                 System.out.println("用户名: " + username + " 密码: " +
   password + " 是否记住我: " + rememberValue);
23                 UsernamePasswordAuthenticationToken authRequest = new
   UsernamePasswordAuthenticationToken(username, password);
24                 setDetails(request, authRequest);
25                 return
   this.getAuthenticationManager().authenticate(authRequest);
26             } catch (IOException e) {
27                 e.printStackTrace();
```

```
            }
        }
        return super.attemptAuthentication(request, response);
    }
}
```

自定义 RememberMeService

```
public class RememberMeService extends
PersistentTokenBasedRememberMeServices {

    public RememberMeService(String key, UserDetailsService
userDetailsService, PersistentTokenRepository tokenRepository) {
        super(key, userDetailsService, tokenRepository);
    }

    @Override
    protected boolean rememberMeRequested(HttpServletRequest request, String
parameter) {
        String paramValue = request.getAttribute(parameter).toString();
        if (paramValue != null) {
            return paramValue.equalsIgnoreCase("true") ||
paramValue.equalsIgnoreCase("on")
                    || paramValue.equalsIgnoreCase("yes") ||
paramValue.equals("1");
        }
        return false;
    }
}
```

配置记住我

```
@Configuration
public class WebSecurityConfig {

    private final DataSource dataSource;

    public WebSecurityConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public PersistentTokenRepository persistentTokenRepository() {
        JdbcTokenRepositoryImpl jdbcTokenRepository = new
JdbcTokenRepositoryImpl();
        // 项目启动时创建表。第一次启动后注释掉即可
//        jdbcTokenRepository.setCreateTableOnStartup(true);
        jdbcTokenRepository.setDataSource(dataSource);
        return jdbcTokenRepository;
    }

```

```
19
20      @Bean
21      public UserDetailsService userDetailsService() {
22          UserDetails user = User.withUsername("admin").password("
        {noop}123").roles("ADMIN").build();
23          return new InMemoryUserDetailsManager(user);
24      }
25
26
27      @Bean
28      public RememberMeServices rememberMeServices() {
29          return new RememberMeService(UUID.randomUUID().toString(),
        userDetailsService(), persistentTokenRepository());
30      }
31
32      @Bean
33      public AuthenticationManager
        authenticationManager(AuthenticationConfiguration
        authenticationConfiguration) throws Exception {
34          return authenticationConfiguration.getAuthenticationManager();
35      }
36
37      @Bean
38      public LoginFilter loginFilter(AuthenticationManager
        authenticationManager) {
39          LoginFilter filter = new LoginFilter();
40          filter.setUsernameParameter("username");
41          filter.setPasswordParameter("password");
42          filter.setFilterProcessesUrl("/doLogin");
43
44          filter.setAuthenticationManager(authenticationManager);
45          filter.setRememberMeServices(rememberMeServices());
46
47          filter.setAuthenticationFailureHandler(new LoginFailureHandler());
48          filter.setAuthenticationSuccessHandler(new LoginSuccessHandler());
49
50          return filter;
51      }
52
53
54      @Bean
55      public SecurityFilterChain filterChain(HttpSecurity http) throws
        Exception {
56          http.authorizeHttpRequests()
57                  .mvcMatchers("/index").permitAll()
58                  .anyRequest().authenticated()
59                  .and().formLogin()
60                  .and().exceptionHandling().authenticationEntryPoint(new
        UnAuthenticationHandler())
61                  .and().logout().logoutSuccessHandler(new LogoutHandler())
62                  .and().rememberMe()
63                  .tokenRepository(persistentTokenRepository()) // 配置token持
        久化仓库
64                  .userDetailsService(userDetailsService())
65
        .and().addFilterBefore(loginFilter(http.getSharedObject(AuthenticationManage
        r.class)), UsernamePasswordAuthenticationFilter.class);
66          return http.csrf().disable().build();
```

```
    }

}
```

会话管理

当浏览器调用登录接口登录成功后，服务端会和浏览器之间建立一个会话 (Session) 浏览器在每次发送请求时都会携带一个 SessionId，服务端则根据这个 SessionId 来判断用户身份。当浏览器关闭后，服务端的 Session 并不会自动销毁，需要开发者手动在服务端调用 Session销毁方法，或者等 Session 过期时间到了自动销毁。在Spring Security 中，与HttpSession相关的功能由 SessionManagementFiter 和SessionAutheaticationStrateey 接口来处理，SessionManagomentFilter 过滤器将 Session 相关操作委托给 SessionAuthenticationStrateey 接口去完成。

会话并发管理就是指在当前系统中，同一个用户可以同时创建多少个会话，如果一个设备对应一个会话，那么也可以简单理解为同一个用户可以同时在多少台设备上进行登录。默认情况下，同一用户在多少台设备上登录并没有限制，不过开发者可以在 Spring Security 中对此进行配置。

```java
/**
 * @Author: chenyang
 * @DateTime: 2023/2/28 14:19
 * @Description: session 并发处理类
 *  前提：Session 并发处理的配置为 maxSessionsPreventsLogin(false)
 *  用户的并发 Session 会话数量达到上限，新会话登录后，最老会话会在下一次请求中失效，并执行此策略
 */
public class SessionExpiredHandler implements SessionInformationExpiredStrategy {

    @Override
    public void onExpiredSessionDetected(SessionInformationExpiredEvent event) throws IOException, ServletException {
        HttpServletResponse response = event.getResponse();
        response.setContentType("application/json;charset=UTF-8");
        Map<String, Object> result = new HashMap<>();
        result.put("status", 500);
        result.put("msg", "当前会话已经失效,请重新登录!");
        String s = new ObjectMapper().writeValueAsString(result);
        response.getWriter().println(s);
        response.flushBuffer();
    }
}
```

```java
    /**
     * 配置 Session 的监听器（注意：如果使用并发 Session 控制，一般都需要配置该监听器）
     * 解决 Session 失效后, SessionRegistry 中 SessionInformation 没有同步失效的问题
```

```
     *
     * @return
     */
    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }

    /**
     *
     * @param http
     * @return
     * @throws Exception
     */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index").permitAll()
                .mvcMatchers("/hello").authenticated()
                .and().formLogin();

        http.rememberMe();

        // session 管理
        http.sessionManagement() // 开启会话管理
                .maximumSessions(1) // 允许同一个用户只允许创建一个会话
                .expiredSessionStrategy(new SessionExpiredHandler()) //
session 失效处理类
                .maxSessionsPreventsLogin(false);// 登录之后禁止再次登录

        return http.build();
    }
```

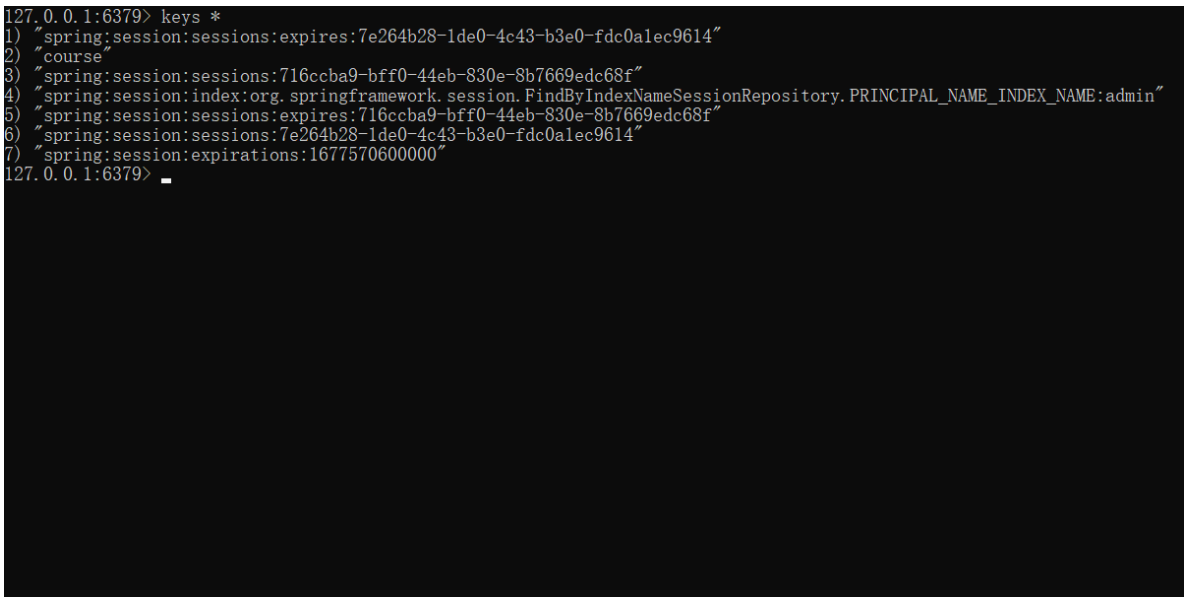
会话共享

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

```
    /**
     * 配置 Session 的监听器（注意：如果使用并发 Session 控制，一般都需要配置该监听
器）
     * 解决 Session 失效后，SessionRegistry 中 SessionInformation 没有同步失效的
问题
     *
     * @return
```

```
     */
    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }

    /**
     * 注册 SessionRegistry，该 Bean 用于管理 Session 会话并发控制
     * 默认为 SessionRegistryImpl 实现类
     *
     * @return
     */
    @Bean
    public SessionRegistry sessionRegistry() {
        return new SessionRegistryImpl();
    }

    /**
     * 当配置了.maximumSessions(1).maxSessionsPreventsLogin(false)要求只能一个用户 Session 登录时，
     * 我们在两个地方使用相同的账号，并且都勾选 remember-me 进行登录。
     * 最老会话的下一次请求不但会使老会话强制失效，还会使数据库中所有该用户的所有 remember-me 记录被删除
     *
     * @param http
     * @return
     * @throws Exception
     */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index").permitAll()
                .mvcMatchers("/hello").authenticated()
                .and().formLogin();

        http.rememberMe();

        // session 管理
        http.sessionManagement() // 开启会话管理
                .maximumSessions(1) // 允许同一个用户只允许创建一个会话
                .expiredSessionStrategy(new SessionExpiredHandler()) // session 失效处理类
                .sessionRegistry(sessionRegistry()) // session 存储策略
                .maxSessionsPreventsLogin(false);// 登录之后禁止再次登录

        return http.build();
    }
```

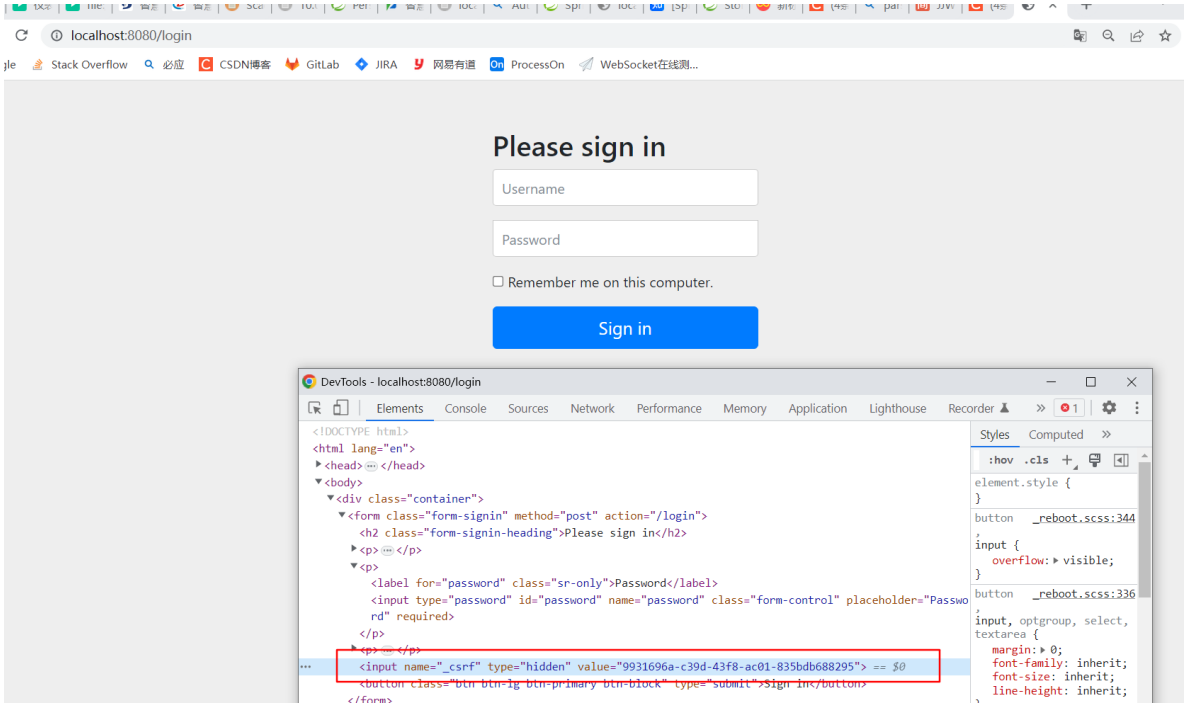
登入成功后查看 redis

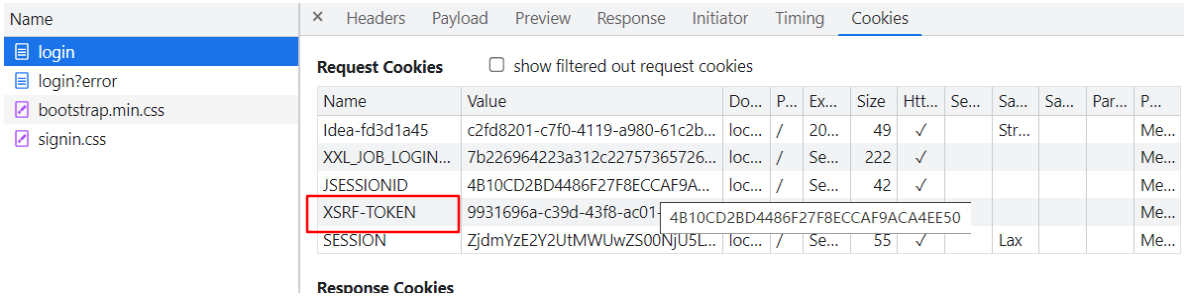


CSRF

```
@Configuration
public class SecurityWebConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(User.withUsername("admin").password("{noop}111").roles("SUPER_ADMIN").build());
        return manager;
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }


    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }


    @Bean
    public SessionRegistry sessionRegistry() {
        return new SessionRegistryImpl();
    }


    @Bean
```

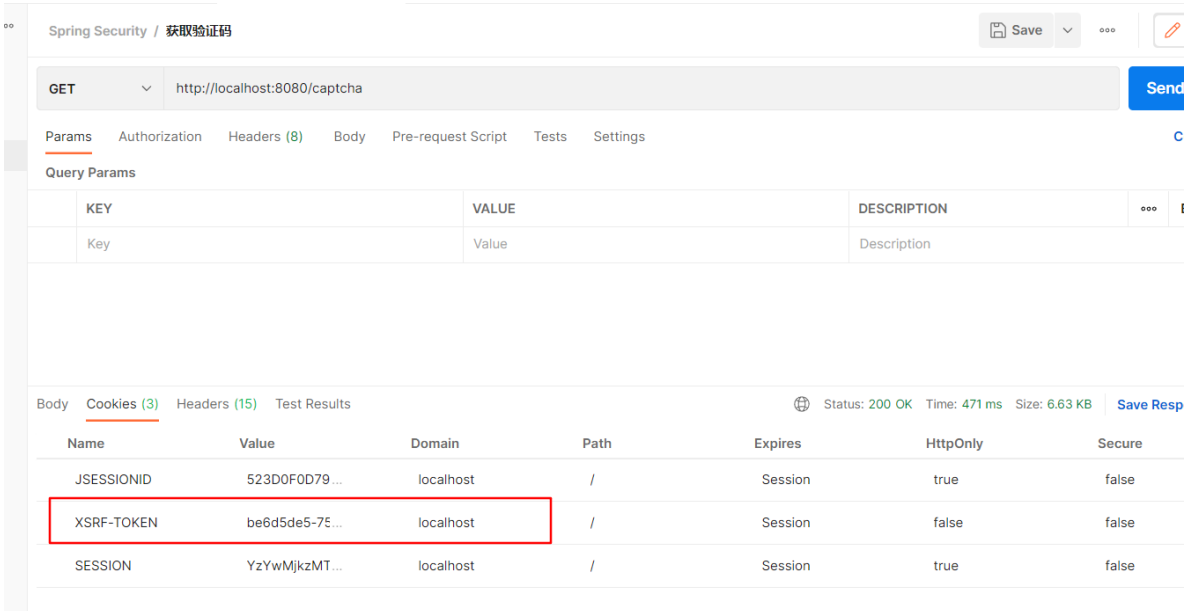
```
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index").permitAll()
                .mvcMatchers("/hello").authenticated()
                .and().formLogin();

        http.rememberMe();

        // session 管理
        http.sessionManagement() // 开启会话管理
                .maximumSessions(1) // 允许同一个用户只允许创建一个会话
                .expiredSessionStrategy(new SessionExpiredHandler()) // session 失效处理类
                .sessionRegistry(sessionRegistry()) // session 存储策略
                .maxSessionsPreventsLogin(false);// 登录之后禁止再次登录

        // 将生成 csrf 放入到cookie 中

 http.csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());

        // 跨域处理方案
        http.cors().configurationSource(configurationSource());


        // 异常处理，认证异常和授权异常
        return http.build();
    }

    /**
     * 跨域资源配置
     * @return
     */
    public CorsConfigurationSource configurationSource() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.setAllowedHeaders(Arrays.asList("*"));
        corsConfiguration.setAllowedMethods(Arrays.asList("*"));
        corsConfiguration.setAllowedOrigins(Arrays.asList("*"));
        corsConfiguration.setMaxAge(3600L);
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", corsConfiguration);
        return source;
    }
}
```





前后端分离

首先随便发起一次请求获取 XSRF-TOKEN



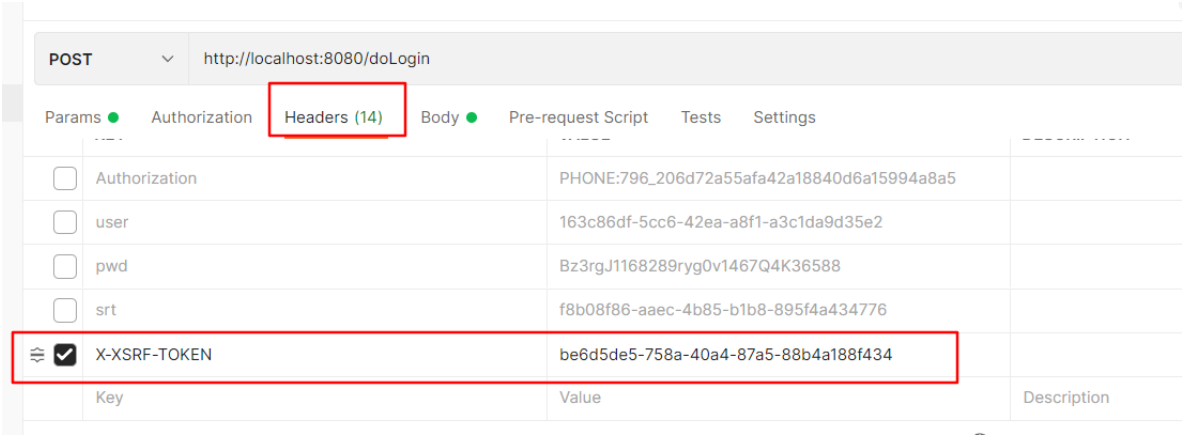
发送请求携带令牌即可

- 请求参数中携带令牌

```
key: _csrf
value:"xxx"
```

- 请求头中携带令牌

```
X-XSRF-TOKEN:value
```

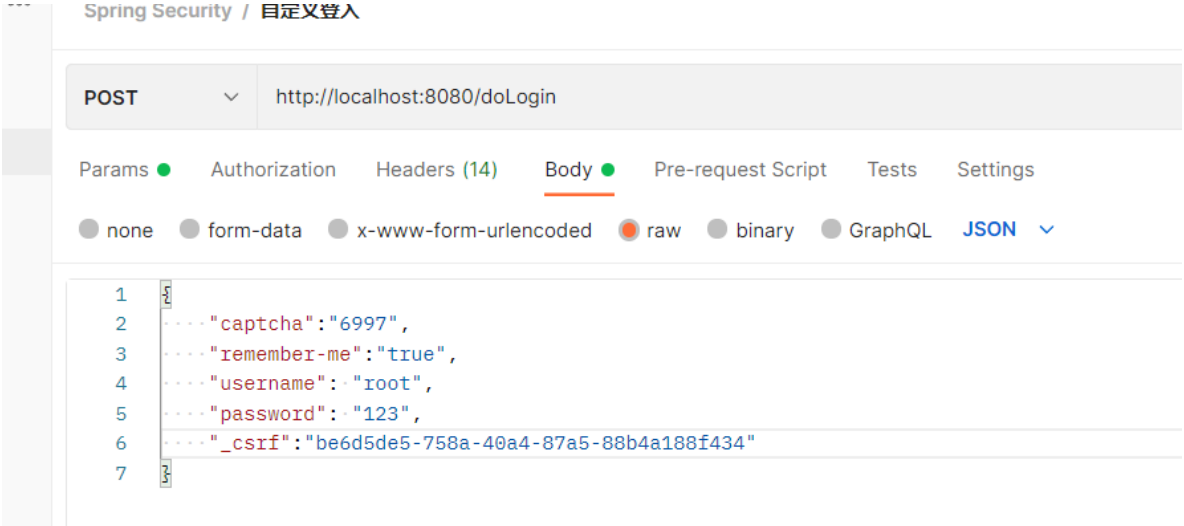


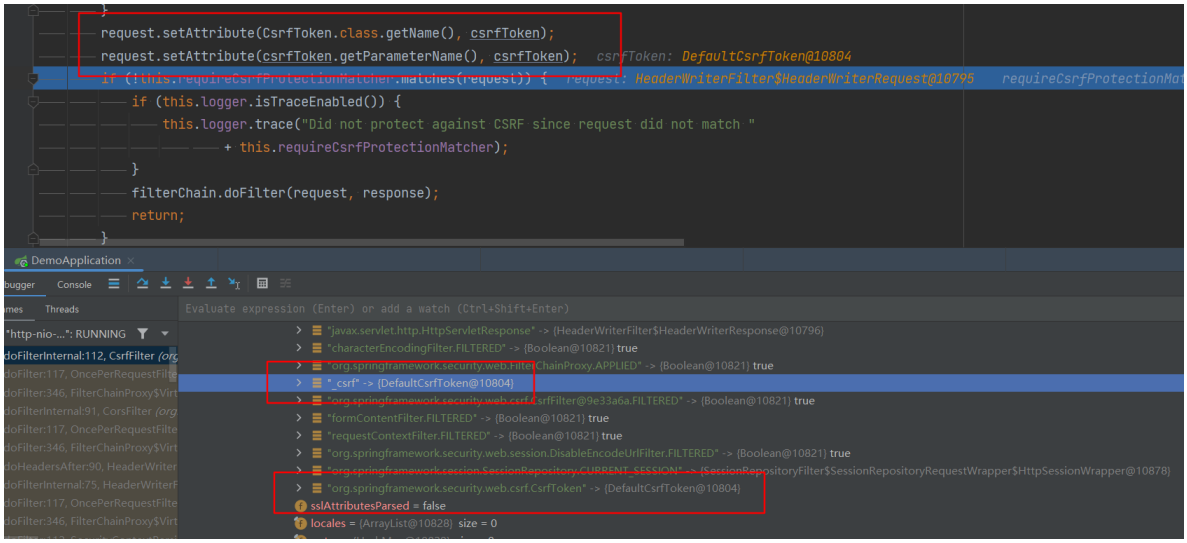
源码解析

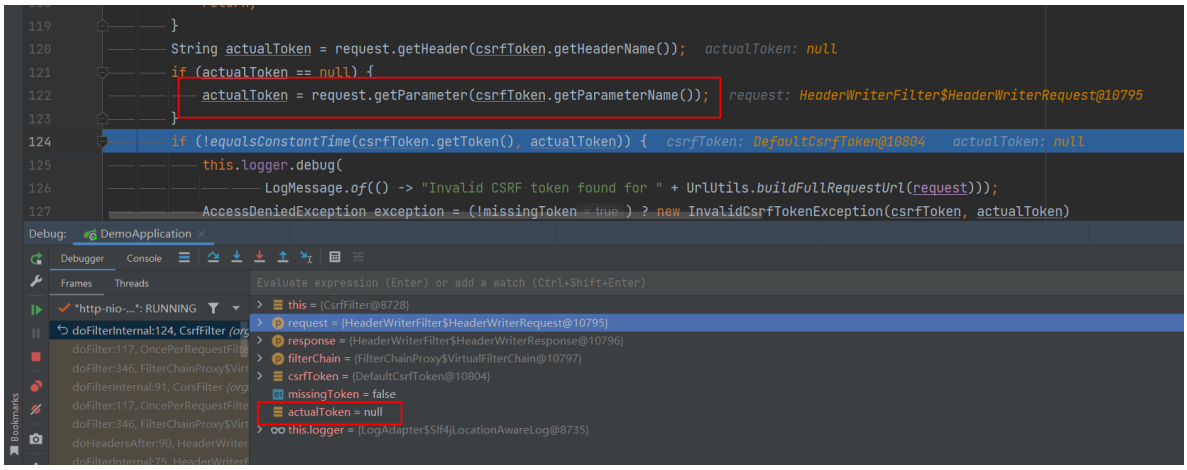

```
@Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        request.setAttribute(HttpServletResponse.class.getName(), response);
        CsrfToken csrfToken = this.tokenRepository.loadToken(request);
        boolean missingToken = (csrfToken == null);
        if (missingToken) {
```

```
            csrfToken = this.tokenRepository.generateToken(request);
            this.tokenRepository.saveToken(csrfToken, request, response);
        }
        request.setAttribute(CsrfToken.class.getName(), csrfToken);
        request.setAttribute(csrfToken.getParameterName(), csrfToken);
        if (!this.requireCsrfProtectionMatcher.matches(request)) {
            if (this.logger.isTraceEnabled()) {
                this.logger.trace("Did not protect against CSRF since
request did not match "
                        + this.requireCsrfProtectionMatcher);
            }
            filterChain.doFilter(request, response);
            return;
        }
        String actualToken = request.getHeader(csrfToken.getHeaderName());
        if (actualToken == null) {
            actualToken =
request.getParameter(csrfToken.getParameterName());
        }
        if (!equalsConstantTime(csrfToken.getToken(), actualToken)) {
            this.logger.debug(
                    LogMessage.of(() -> "Invalid CSRF token found for " +
UrlUtils.buildFullRequestUrl(request)));
            AccessDeniedException exception = (!missingToken) ? new
InvalidCsrfTokenException(csrfToken, actualToken)
                    : new MissingCsrfTokenException(actualToken);
            this.accessDeniedHandler.handle(request, response, exception);
            return;
        }
        filterChain.doFilter(request, response);
    }
```

请求参数中携带令牌







https://blog.csdn.net/zhanghuiyu01/article/details/68924818

由于请求参数为 JSON，所以 `request.getParameter(csrfToken.getParameterName())` 获取不到请求参数中的 `_csrf`,此次请求将会被拒绝。但是如果是 GET 请求就不会有问题。

总结：POST 请求必须将令牌写道 Header 中

GET 请求写在请求头或者请求参数中都是可以的

整合 JWT

导入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
```

```
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.0</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.7</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>

<dependency>
    <groupId>com.github.penggle</groupId>
    <artifactId>kaptcha</artifactId>
    <version>2.3.2</version>
</dependency>
```

JwtUtil

```
public class JwtUtil {

    public static final Long EXPIRE = 700L;

    public static final String HEAD = "Authentication";

    public static final String SECRET = "nice_try_secret";


    /**
     * 生成 Token
     *
     * @param username
     * @return
     */
    public static String createToken(String username) {

        Date nowDate = new Date();
```

```
        Date expireDate = new Date(nowDate.getTime() + EXPIRE * 1000);

        return Jwts.builder()
                .setHeaderParam("type", "JWT")
                .setSubject(username)
                .setIssuedAt(nowDate)
                .setExpiration(expireDate)
                .signWith(SignatureAlgorithm.HS512, SECRET)
                .compact();
    }


    /**
     * 获取token中注册信息
     *
     * @param token
     * @return
     */
    public static Claims getTokenClaim(String token) {
        try {
            return Jwts.parser()
                    .setSigningKey(SECRET)
                    .parseClaimsJws(token)
                    .getBody();
        } catch (Exception e) {
            return null;
        }
    }


    /**
     * 校验 Claims 是否 过期
     * @param claim
     * @return
     */
    public static Boolean checkClaimExpire(Claims claim) {
        if (Objects.isNull(claim)) {
            return false;
        }
        Date expiration = claim.getExpiration();
        return expiration.before(new Date());
    }
}
```

编写 filter

```
@Component
@Order(-1)
public class JwtFilter extends OncePerRequestFilter {

    @Autowired
    private UserService userService;

    @Override
```

```
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        String jwtToken = request.getHeader(JwtUtil.HEAD);
        if (Objects.isNull(jwtToken)){
            filterChain.doFilter(request, response);
            return;
        }
        Claims claim = JwtUtil.getTokenClaim(jwtToken);
        if (Objects.isNull(claim)){
            throw new RuntimeException("token 解析失败");
        }
        Boolean expireFlag = JwtUtil.checkClaimExpire(claim);
        if (expireFlag){
            throw new RuntimeException("token 已失效");
        }
        String username = claim.getSubject();
        User user = userService.loadUserByUsername(username);
        if (Objects.isNull(user)){
            throw new RuntimeException("用户信息失效");
        }
        UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
        SecurityContextHolder.getContext().setAuthentication(token);
        filterChain.doFilter(request, response);
    }
}
```

```
public class LoginFilter extends UsernamePasswordAuthenticationFilter {

    public static final String FORM_CAPTCHA_KEY = "captcha";

    private String captchaParameter = FORM_CAPTCHA_KEY;

    public String getCaptchaParameter() {
        return captchaParameter;
    }

    public void setCaptchaParameter(String captchaParameter) {
        this.captchaParameter = captchaParameter;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
        System.out.println("=========================================");

        if (!request.getMethod().equals("POST")){
            throw new AuthenticationServiceException("Authentication method not supported: " + request.getMethod());
        }

        if (request.getContentType().equalsIgnoreCase(MediaType.APPLICATION_JSON_VALUE)){
```

```
            Map<String, String> userInfo = null;
            try {
                userInfo = new ObjectMapper().readValue(request.getInputStream(), Map.class);
            } catch (IOException e) {
                e.printStackTrace();
            }

            if (Objects.isNull(userInfo)){
                throw new NullPointerException("登入参数为空！登入失败");
            }

            String username = userInfo.get(getUsernameParameter());
            String password = userInfo.get(getPasswordParameter());
            String captcha = userInfo.get(getCaptchaParameter());
            String sessionVerifyCode = (String) request.getSession().getAttribute(FORM_CAPTCHA_KEY);

            if (ObjectUtils.isEmpty(captcha) || ObjectUtils.isEmpty(sessionVerifyCode)){
                throw new CaptchaNotMatchException("验证码不能为空!");
            }

            if (!captcha.equalsIgnoreCase(sessionVerifyCode)){
                throw new CaptchaNotMatchException("验证码不匹配!");
            }

            UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(username, password);
            setDetails(request, authRequest);
            return this.getAuthenticationManager().authenticate(authRequest);
        }
        return super.attemptAuthentication(request, response);
    }
}
```

登入成功处理器

```
public class LoginSuccessHandler implements AuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException {

        String token = JwtUtil.createToken(authentication.getName());
        HashMap<String, Object> map = new HashMap<>();
        map.put("msg", "登入成功");
        map.put("status", 200);
        map.put("token", token);
        response.setContentType("application/json;charset=UTF-8");
        String json = new ObjectMapper().writeValueAsString(map);

        response.getWriter().println(json);
```

```
    }
}
```

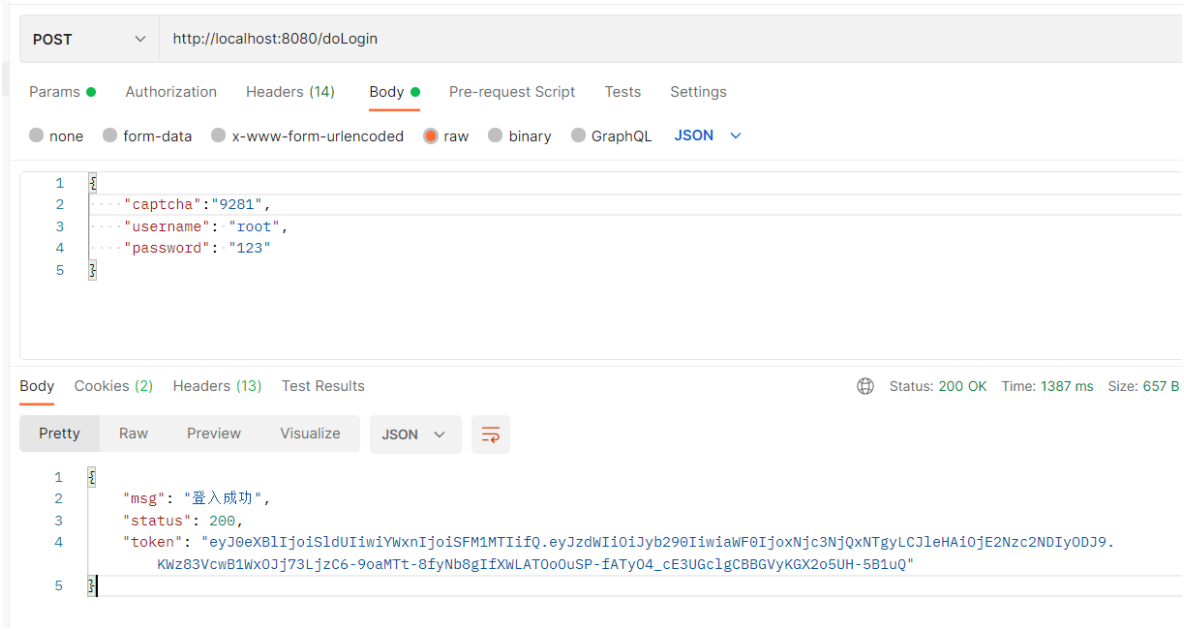
配置

```
@Configuration
public class WebSecurityConfig {

    @Autowired
    private JwtFilter jwtFilter;

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public LoginFilter loginFilter(AuthenticationManager authenticationManager) {
        LoginFilter filter = new LoginFilter();

        filter.setFilterProcessesUrl("/doLogin");

        filter.setUsernameParameter("username");
        filter.setPasswordParameter("password");
        filter.setCaptchaParameter("captcha");

        filter.setAuthenticationManager(authenticationManager);

        filter.setAuthenticationSuccessHandler(new LoginSuccessHandler());
        filter.setAuthenticationFailureHandler(new LoginFailureHandler());

        return filter;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .mvcMatchers("/index", "/captcha").permitAll()
                .anyRequest().authenticated()
                .and().formLogin();

        // 注销处理
        http.logout()
                .logoutSuccessHandler(new LogoutHandler());

        // session 管理 禁用 session

 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        // 授权、认证异常处理
```

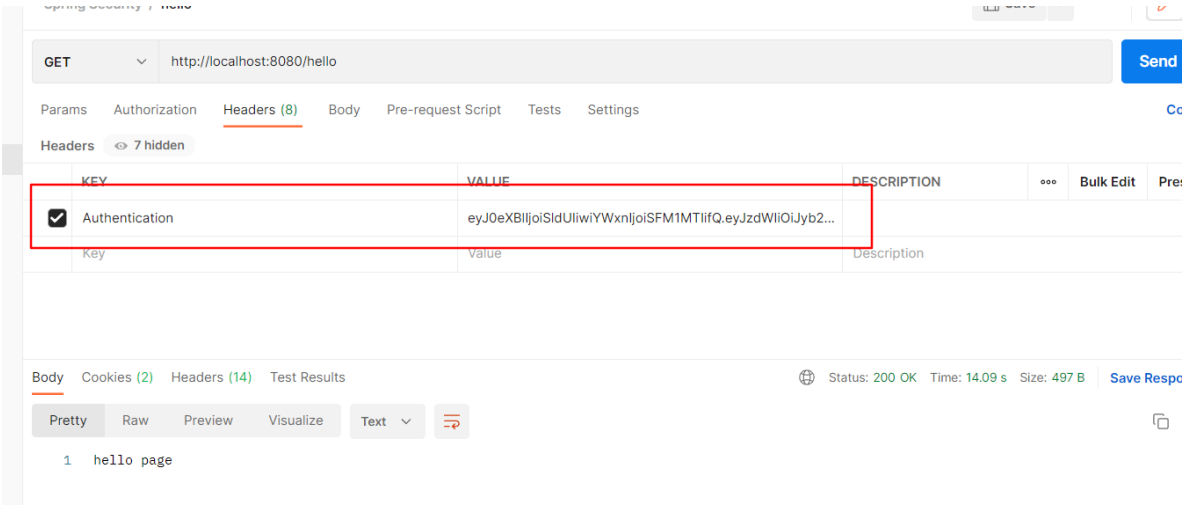
```
45          http.exceptionHandling()
46                  .authenticationEntryPoint(new UnAuthenticationHandler())
47                  .accessDeniedHandler(new UnAbleAccessHandler());
48
49          // 不使用 session， csrf 禁用，
50          http.csrf().disable();
51
52          http.headers().frameOptions().disable();
53
54          // 跨域处理方案
55          http.cors().configurationSource(configurationSource());
56
57          // 添加自定义过滤器
58          http.addFilterAt(jwtFilter, LoginFilter.class);
59
 http.addFilterBefore(loginFilter(http.getSharedObject(AuthenticationManager
.class)), UsernamePasswordAuthenticationFilter.class);
60          return http.build();
61      }
62
63
64      /**
65       * 跨域资源配置
66       *
67       * @return
68       */
69      public CorsConfigurationSource configurationSource() {
70          CorsConfiguration corsConfiguration = new CorsConfiguration();
71          corsConfiguration.setAllowedHeaders(Arrays.asList("*"));
72          corsConfiguration.setAllowedMethods(Arrays.asList("*"));
73          corsConfiguration.setAllowedOrigins(Arrays.asList("*"));
74          corsConfiguration.setMaxAge(3600L);
75          UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
76          source.registerCorsConfiguration("/**", corsConfiguration);
77          return source;
78      }
79  }
```

操作流程

登入获取 token



请求头携带 token



不带 token