

封面

Spring Security

实战干货

码农小胖哥 著



2022 加强版

felord.cn 出品

关于作者

我是 **码农小胖哥**，一名软件工程师，同时也是一名热衷于技术分享、拥抱开源的技术博主。

我的技术栈主要是 **Java**、**Python**、**Golang** 和 **Kotlin**。欢迎各位同行进行交流分享。

✓我的个人技术分享都在我的个人博客: <https://felord.cn>

✓业余运营有自己的公众号:



✓也在微信视频号上进行一些创作:



✓我在其它平台的专栏:



1-概要

公众号: 码农小胖哥 回复 **2021开工福利** 可获取旧版初学者教程。

Spring Security曾是我的阴影

2016年在开发一个项目的时候初次接触到**Spring Security**，那时候我们刚用上**Spring+SpringMVC+MyBatis**。权限控制相关的开发由我负责，当时**Spring Security**给我留下很深刻的印象：太复杂了！完全无从下手！差点给我劝退！这个是当年写的DEMO配置：

```
<!--app 端接口 -->
<http pattern="/content/**" security="none"/>
<http pattern="/webjs/**" security="none"/>
<http pattern="/login**.jsp" security="none"/>
<http pattern="/shareCoupon/*" security="none"/>
<http pattern="/api/**">
    <intercept-url pattern="/api/v2/**" access="hasRole('MOBILE')"/>
</http>
<http auto-config="false">
    <access-denied-handler ref="accessDeniedHandler"/>
<!--
<form-login login-page="/login.jsp" authentication-failure-url="/login.jsp?login_error=1" default-target-url="/" always-use-default-target="true" />
-->
<http-basic/>
<custom-filter position="CONCURRENT_SESSION_FILTER" ref="concurrencyFilter"/>
<custom-filter ref="loginFilter" before="FORM_LOGIN_FILTER"/>
<!-- 替换默认的LogoutFilter <logout logout-success-url="/login.jsp" /> -->
<custom-filter ref="logoutFilter" position="LOGOUT_FILTER"/>
<!-- 增加一个自定义的customSecurityInterceptor，放在FILTER_SECURITY_INTERCEPTOR之前，实现用户、角色、权限、资源的数据库管理。 -->
<custom-filter ref="customSecurityInterceptor" before="FILTER_SECURITY_INTERCEPTOR"/>
<remember-me/>
<!-- 会话管理配置 -->
<session-management session-authentication-strategy-ref="sessionAuthenticationStrategy">
    invalid-session-url="/logon/commonSessionExpired.htm"/>

<http>
<!--<beans:bean id="concurrencyFilter" class="org.springframework.security.web.session.ConcurrentSessionFilter"> -->
<beans:bean id="concurrencyFilter" class="cn.felord.bussiness.aop.RouterFilter">
    <beans:property name="sessionRegistry" ref="sessionRegistry"/>
    <!-- <beans:property name="expiredUrl" value="/logon/commonSessionExpired.htm" />-->
<beans:bean>
<beans:bean id="sessionAuthenticationStrategy"
    class="org.springframework.security.web.authentication.session.ConcurrentSessionControlStrategy">
```

Spring Security给我留下了阴影，之后的两年我再也没有碰这个框架。

Spring Security并不可怕

随着Spring Boot的流行，Spring Security的配置难度下降了不少。刚好有一阵时间很闲，就花了点精力学习了Spring Security。随着学习的深入，我发现Spring Security并没有想象中的那么重、那么复杂，反而感觉它的灵活度非常高，能够适应很多涉及到认证、授权、权限控制的场景。它的复杂只是表面的，你需要的是一个合适突破口来对Spring Security进行抽丝剥茧，这也是我写这个教程的目的。

Spring Security的本质

Spring Security本质上就是一条流水线。一次请求（Request）到达服务器，都会经过一系列的过滤器链（FilterChain）。在这个链条里，有的负责检查请求头合法，有的负责管理Session会话，有的负责认证，有的负责授权……，这一套组合技最终决定响应的结果。明白这个，就好学多了。

本教程面向的人群

期望掌握Spring Security定制能力的开发者；期望对Spring Security的一些核心概念进一步了解的开发者。同时我也会在教程中分享一些我自己的学习经验，来帮助一些初学者掌握学习的方法。

如果你是初学者，请通过公众号：**码农小胖哥** 回复 2021开工福利 获取循序渐进的学习教程。

环境准备

在编写本教程的时候，刚好Spring Boot 2.6发布了，它集成了Spring Security 5.6，正好借此机会和大家一起学习最新的版本。涉及的主要技术栈以及版本如下：

- Spring Boot 2.6及以上
- Spring Security 5.6及以上
- Spring MVC
- Maven

- Git
- JDBC
- Mybatis

一些学习心得

其实编程是一个手艺活，经常有人问我是如何学习的，无他唯手熟尔。在这里分享几点学习心得：

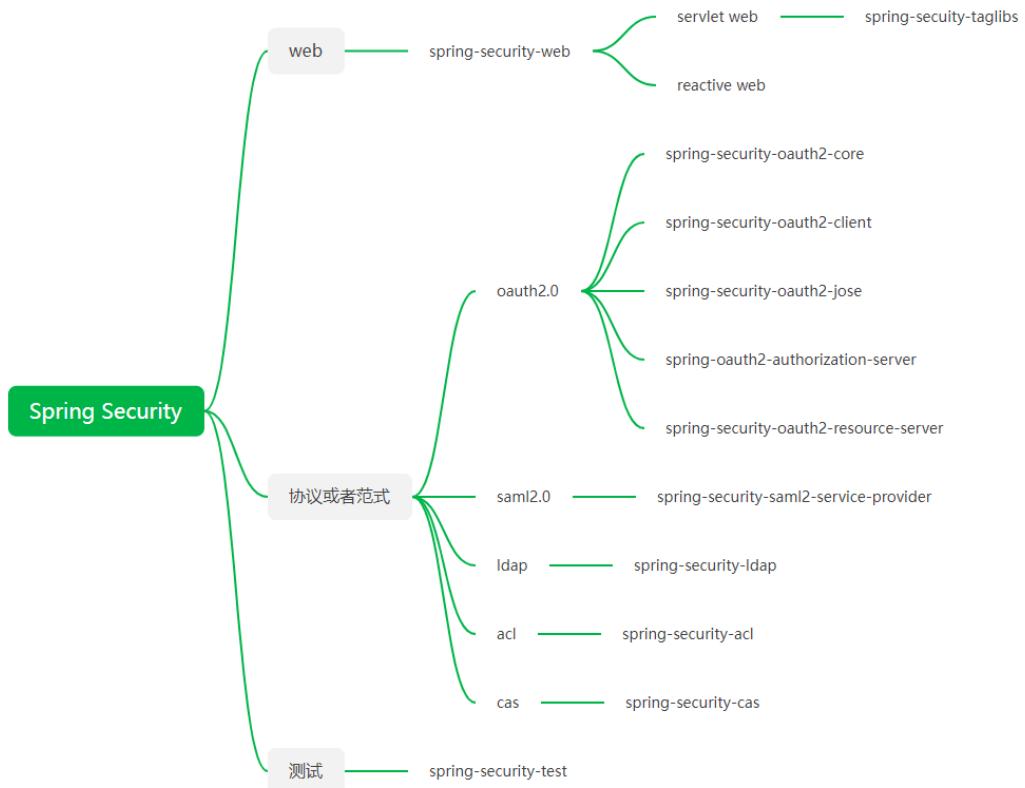
- 看再多教程都不如亲自写一些DEMO，所以期望你掌握每一个知识点后都能够自己去实践一下，理解设计意图。
- 尝试举一反三，自己跟着引导的教程去看一些没有提及的源码，个人能力的突破往往在于养成一些良好的习惯。
- 不要过分追求实现原理的细节，过分追求细节会让你越陷越深。
- 尝试形成独立思考和解决问题的能力。

接下来，让我们开始吧！

2-Spring Security 模块

Spring Security给很多学习者留下了“太重了”的印象，劝退了不少学习者，本篇将对Spring Security的模块体系进行拆解以便各位同学能够分清主次，分而治之。

Spring Security划分了很多模块。下面是根据我个人的见解对Spring Security模块进行的分类。



接下来我就根据这个分类来说明一下哪些模块比较重要，哪些模块一般重要，哪些模块可能不需要去学。

web

目前Spring的Web体系分为**Servlet Web(Spring MVC)**和**Reactive Web(Spring Webflux)**两个体系。**Spring Webflux**目前发展很快，但是**Spring MVC**依然是主流。而且你学会了**Spring Security**涉及**Servlet Web**体系的用法之后，上手另一个体系的难度也会大大降低。这个是学习**Spring Security**的重中之重后面的教程会围绕**Spring Security**在**Servlet Web**体系的运用展开。

spring-security-taglibs 是用来做JSP按钮级别标签控制用的。如果你使用了前后端不分离的开发模式，可以借鉴这个实现特定模板 (freemarker、thymeleaf等) 的标签控制。

协议范式

Spring Security对常见的认证、授权、访问控制协议都做了针对性的实现（模块名称参考上图）。这些协议范式包括：

- **oauth2.0(oidc)** 当下比较火的授权协议。**oidc**在**oauth2.0**的流程上增加了用户认证。这个是必学的协议之一。
- **saml2.0** 安全断言标记语言协议。比较复杂，主要做用户认证、单点登录。目前使用它的也不少，这个学习成本高些，很容易劝退你，因此学习的优先级不高。
- **ldap** 轻量目录访问协议。主要场景是企业级门户，在目前的Web开发中用的并不是很多。
- **cas** 单点登录服务器，目前还有不少老项目使用。和**ldap**一样，用到了再学。
- **acl** 访问控制列表，我不确定它是不是一个协议，只知道它是基于领域模型的资源访问控制技术，可以对资源做到细粒度的访问控制。有这方面需求的可以去了解一下，初级入门不需要学习。

学习这些模块的时候，你要确保以下两点：

- 搞清楚这些协议的应用场景。
- 熟悉协议的流程。



微信搜一搜



码农小胖哥

3-Spring Security用户管理

集成Spring Security

为Spring Boot项目集成**Spring Security**只需要引入其Starter即可：

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-security</artifactId>
4 </dependency>
```

确保你的项目已经集成Spring MVC或者Spring Webflux，本教程以Spring MVC为例。

用户管理

随意编写个GET请求的接口并调用，如果能成功跳转到 `/login` 登录页面，就说明集成成功了。

Please sign in

Username

Password

Sign in

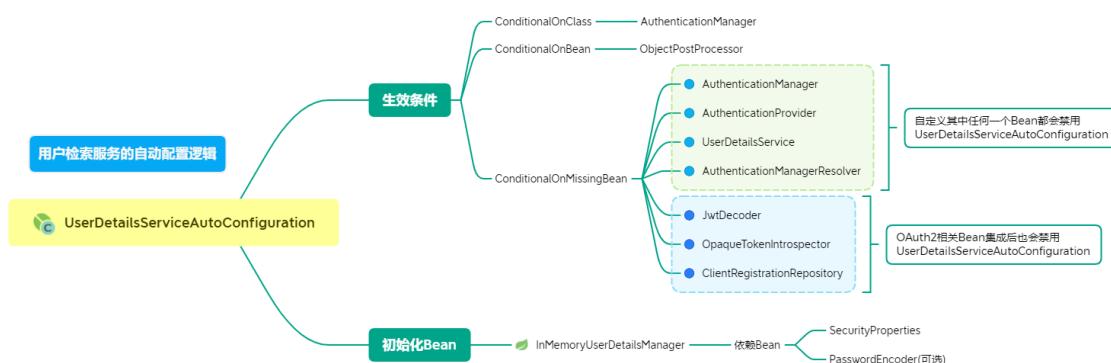
那账号密码是啥呢？

默认用户配置

用户名默认是 `user`。Spring Boot启动后，控制台会打印出的：

```
1 Using generated security password: e1f163be-ad18-4be1-977c-88a6bcee0d37
```

后面的长串就是密码。IDEA快捷键 `CTRL SHIFT R` 全局搜索 `Using generated security password`，最后定位到 `UserDetailsServiceAutoConfiguration`。它只初始化了一个叫 `InMemoryUserDetailsService` 的 Spring Bean。



这么说来只要我们自己定义一个 `UserDetailsService` 的话就可以取而代之。

UserDetailsService

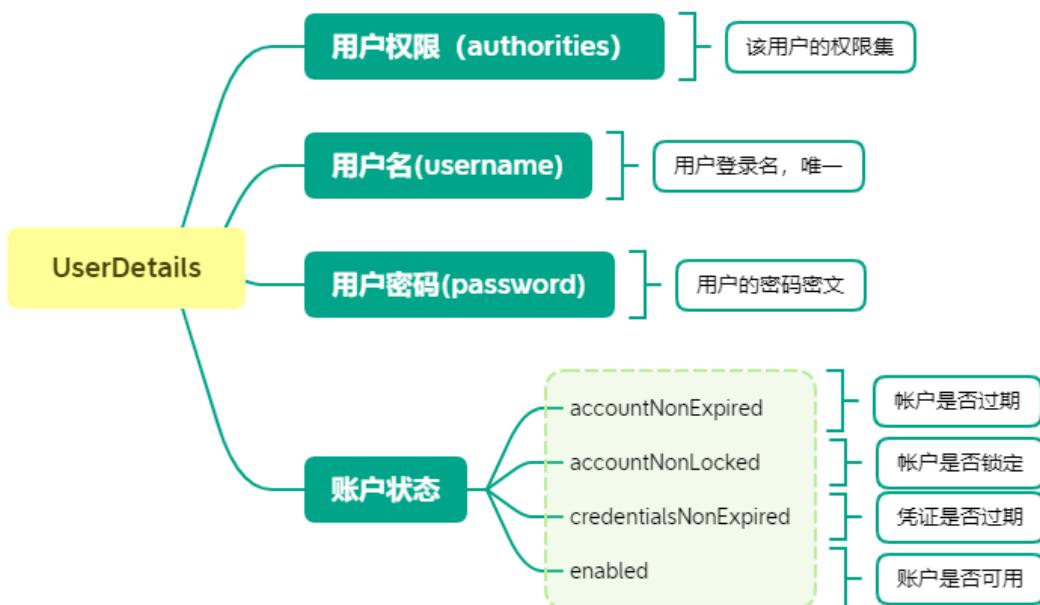
UserDetailsService 是Spring Security中非常重要的一个接口。

```
1 public interface UserDetailsService {  
2  
3     UserDetails loadUserByUsername(String username) throws  
4         UsernameNotFoundException;  
5 }
```

这接口就一个方法，根据用户名加载用户 UserDetails，可能抛出用户不存在的异常。不管什么登陆方式都推荐用 UserDetailsService 去检索个人信息。

UserDetails

这个是对用户信息的抽象描述。



我们照着 InMemoryUserDetailsService 高仿一个试试：

```
1 @Configuration(proxyBeanMethods = false)  
2 public class UserDetailsServiceConfiguration {  
3  
4     @Bean  
5     UserDetailsService userDetailsService() {  
6  
7         return username ->  
8             // 用户名  
9             User.withUsername(username)  
10                // 密码  
11                .password("password")  
12                // 权限集  
13                .authorities("ROLE_USER", "ROLE_ADMIN")  
14                .build();  
15     }  
16  
17 }
```

如果这个成功了切换到数据库还不易如反掌，根据用户名 `username` 查询用户然后封装成 `UserDetails` 就可以了。但是上面的写法居然报错了：

```
1  java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id
2      "null"
3      at
4          org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdP
5              passwordEncoder.matches(DelegatingPasswordEncoder.java:254) ~[spring-security-
6                  crypto-5.6.0.jar:5.6.0]
7      at
8          org.springframework.security.crypto.password.DelegatingPasswordEncoder.matches(Del
9              egatingPasswordEncoder.java:202) ~[spring-security-crypto-5.6.0.jar:5.6.0]
10     at
11         org.springframework.security.authentication.dao.DaoAuthenticationProvider.addition
12             alAuthenticationChecks(DaoAuthenticationProvider.java:76) ~[spring-security-core-
13                 5.6.0.jar:5.6.0]
```

别慌，有异常堆栈怕啥！我们下一章专门搞明白这个问题。



4-Spring Security密码处理

上一章我们自定义了一个 `UserDetailsService` 的 Spring Bean 想代替默认配置，想不到居然翻车了。抛出了一个异常，关键信息如下：

```
1  There is no PasswordEncoder mapped for the id "null"
```

大意就是一个为 `null` 的 `id` 没有映射到对应的 `PasswordEncoder`。`PasswordEncoder` 又是啥？跟密码有关系？

PasswordEncoder

`PasswordEncoder` 确实跟密码有关系，它提供了三个方法：

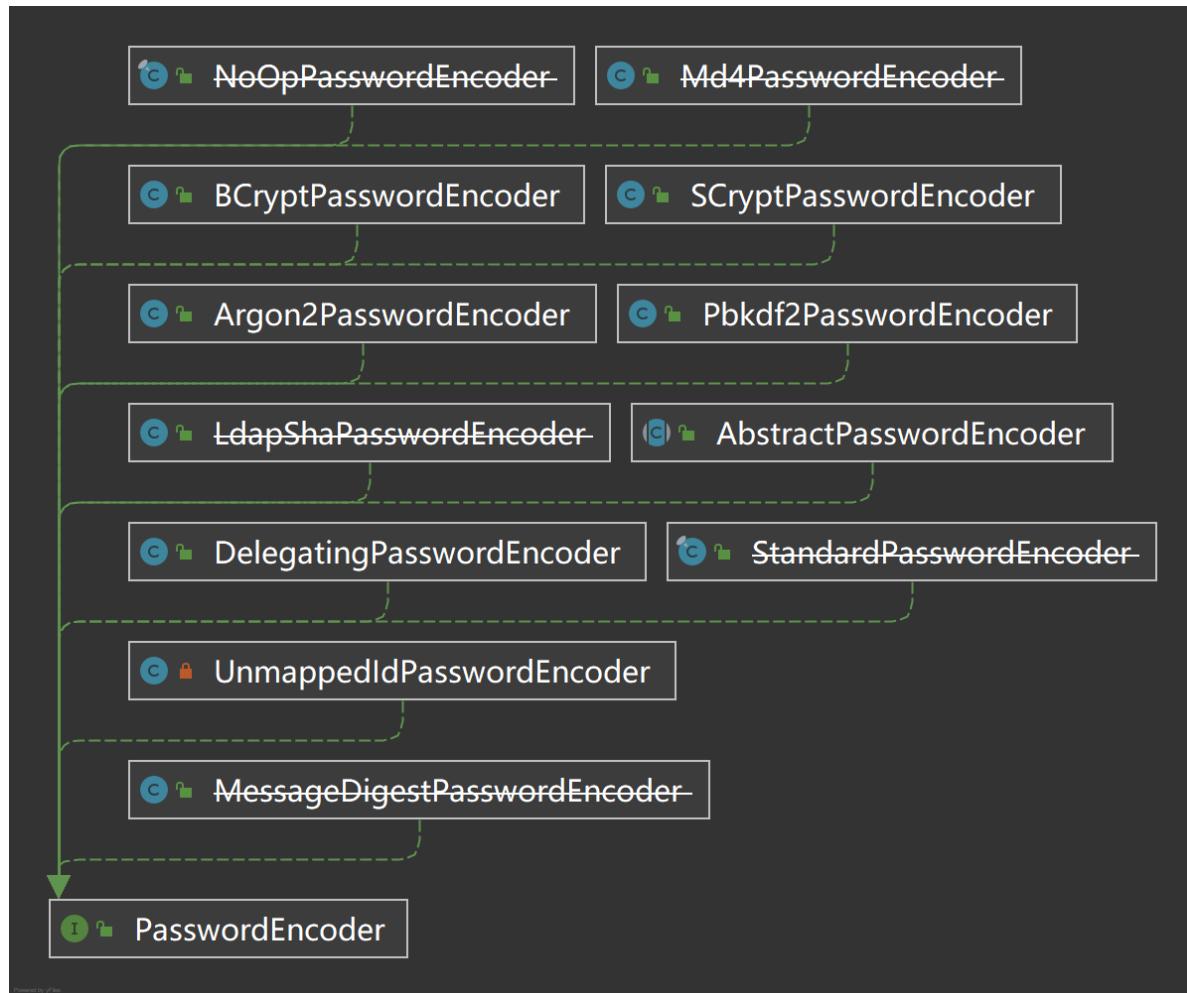
```
1  public interface PasswordEncoder {
2
3      /**
4       * 将原始明文密码编码为密文密码
5       */
6      String encode(CharSequence rawPassword);
7
8      /**
9       * 这个肯定是校验用的明文和持久化的密文进行对比
10      */
```

```

11     boolean matches(CharSequence rawPassword, String encodedPassword);
12
13     /**
14      * 是否需要为密文密码再次编码，编码了两次？
15      */
16     default boolean upgradeEncoding(String encodedPassword) {
17         return false;
18     }
19
20 }

```

我们搞一个 `PasswordEncoder` 的实现试试看，`PasswordEncoder` 有不少算法实现



第一排好像都过时了，我们就挑第二排第一个 `BCryptPasswordEncoder` 吧，至于为什么挑它，你可以认为是一种缘分。

```

1     PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
2     String rawPassword = "felord";
3     String encoded = passwordEncoder.encode(rawPassword);
4     boolean matches = passwordEncoder.matches(rawPassword, encoded);
5     // 断言为false
6     Assertions.assertFalse(matches, encoded);

```

实际的校验结果是 `true`，编码后的密码为

`$2a$10$n0sqByGq6m/9AgN0cXRfkUQiFA/JSDntSm0NCDP/iH1JXztDo.Kzu`。

解决问题

这里知道密码的编码器出了问题，并且借着这个机会熟悉了密码编码器的工作模式。如何解决异常？这里有三种方式。我觉得有必要列举一下：

- 第一种就是直觉，如果你认真观察了上面 `PasswordEncoder` 的实现，会发现一个叫 `UnmappedIdPasswordEncoder` 的，没有映射到 `id` 的 `PasswordEncoder`，答案都直接写脸上了。
- 第二种Debug大法，只要代码可以Debug几乎没有定位不了的问题。直接Debug `PasswordEncoder` 接口，你同样也能找到是 `UnmappedIdPasswordEncoder` 的问题。
- 第三种比较野路子，全局搜关键字 `There is no PasswordEncoder mapped for the id`，这种最省脑细胞，而且大部分情况下还见效快。

既然找到病源了，有病治病。`UnmappedIdPasswordEncoder` 居然是一个叫 `DelegatingPasswordEncoder` 内部类。如果你发现有类或者接口以 `Delegating` 开头的就要注意了，这种接口就像有些中间商，他只接活，但是他又没能力干活，然后又把活转包出去，他赚差价。如果找不到人干就让亲信 `UnmappedIdPasswordEncoder` 来处理。看样子这活是没人接了。先看看 `DelegatingPasswordEncoder` 怎么找“人”代工的。它有个提取 `id` 的方法：

```
1 private static final String PREFIX = "{";
2
3 private static final String SUFFIX = "}";
4
5 private String extractId(String prefixEncodedPassword) {
6     if (prefixEncodedPassword == null) {
7         return null;
8     }
9     int start = prefixEncodedPassword.indexOf(PREFIX);
10    if (start != 0) {
11        return null;
12    }
13    int end = prefixEncodedPassword.indexOf(SUFFIX, start);
14    if (end < 0) {
15        return null;
16    }
17    return prefixEncodedPassword.substring(start + 1, end);
18 }
```

从上面可以得出 `prefixEncodedPassword` 的格式是 `{id}encodedPassword`，原来它是根据这个格式来找“工具人”的。

我们再找找它是怎么初始化的，查到下面这个：

```
1 public final class PasswordEncoderFactories {
2
3     private PasswordEncoderFactories() {
4     }
5
6     @SuppressWarnings("deprecation")
7     public static PasswordEncoder createDelegatingPasswordEncoder() {
8         String encodingId = "bcrypt";
9         Map<String, PasswordEncoder> encoders = new HashMap<>();
10        encoders.put(encodingId, new BCryptPasswordEncoder());
11        encoders.put("ldap", new
12            org.springframework.security.crypto.password.LdapShaPasswordEncoder());
13        encoders.put("MD4", new
14            org.springframework.security.crypto.password.Md4PasswordEncoder());
```

```

13         encoders.put("MD5", new
14             org.springframework.security.crypto.password.MessageDigestPasswordEncoder("MD5"))
15         ;
16         encoders.put("noop",
17             org.springframework.security.crypto.password.NoOpPasswordEncoder.getInstance());
18         encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
19         encoders.put("scrypt", new SCryptPasswordEncoder());
20         encoders.put("SHA-1", new
21             org.springframework.security.crypto.password.MessageDigestPasswordEncoder("SHA-
22             1"));
23         encoders.put("SHA-256",
24             new
25             org.springframework.security.crypto.password.MessageDigestPasswordEncoder("SHA-
26             256"));
26         encoders.put("sha256", new
27             org.springframework.security.crypto.password.StandardPasswordEncoder());
28         encoders.put("argon2", new Argon2PasswordEncoder());
29         return new DelegatingPasswordEncoder(encodingId, encoders);
30     }
31
32 }

```

这下都了然了吧。`DelegatingPasswordEncoder` 原来是从编码后的密码前缀中提取一个 `PasswordEncoder` 的标识 `id`，进而调用该 `PasswordEncoder`。那我们配置的 `PasswordEncoder` 是哪个？

把上一篇配置中我们硬写入了一个用户，我们测一下：

```

1     UserDetails userDetails = User.withUsername("felord")
2         // 密码
3         .password("password")
4         // 权限集
5         .authorities("ROLE_USER", "ROLE_ADMIN")
6         .build();
7
8     String password = userDetails.getPassword();
9     Assertions.assertEquals("password", password);

```

上面的测试结果证明密码非但没有编码，而且不符合格式。`User` 的构建类 `UserBuilder` 提供了指定 `PasswordEncoder` 的功能，只需要把上一篇中的配置改成下面就可以了：

```

1     @Bean
2     UserDetailsService userDetailsService() {
3
4         return username ->
5             // 用户名
6             User.withUsername(username)
7                 // 密码
8                 .password("password")
9                 // 委托密码机
10
11                .passwordEncoder(PasswordEncoderFactories.createDelegatingPasswordEncoder()::enco
12                de)
13
14                // 权限集
15                .authorities("ROLE_USER", "ROLE_ADMIN")
16                .build();
17
18 }

```

这回异常就没了。再得寸进尺一下，如何改成数据库持久化用户呢？这个问题我们下一篇再细说。



5-持久化用户到数据库

实际生产中很少把 `UserDetails` 放在内存中，不然一重启用户全没了，老板不跟你急才怪。今天我们
将实现把用户信息 `UserDetails` 持久化到数据库。

引入持久化依赖

为了方便起见，我们使用Mybatis和H2(一种内置数据库)组合，H2很容易切换为其它关系型数据库。在前
面的教程中增量添加相关依赖：

```
1 <dependency>
2   <groupId>org.mybatis.spring.boot</groupId>
3   <artifactId>mybatis-spring-boot-starter</artifactId>
4   <version>2.2.0</version>
5 </dependency>
6 <dependency>
7   <groupId>com.h2database</groupId>
8   <artifactId>h2</artifactId>
9   <scope>runtime</scope>
10 </dependency>
```

配置

这里涉及到Mybatis的配置、数据源Datasource的配置、H2数据库的一些配置、数据库DDL、DML脚本的
的初始化。

Mybatis配置

这里只需要配置 `mapper.xml` 的路径、数据库下划线转驼峰。

```
1 # mybatis 配置
2 mybatis:
3   mapper-locations: classpath:mapper/*.xml
4   configuration:
5     map-underscore-to-camel-case: true
```

Datasource配置

嵌入式数据库H2的配置如下，如果想进一步了解H2数据库可参考其它文章。

```
1 spring:
2   datasource:
3     # h2驱动
4     driver-class-name: org.h2.Driver
5     # h2 内存数据库 内存模式连接配置 库名: spring_security mysql模式
6     url: jdbc:h2:mem:spring_security;MODE=MySQL;DATABASE_TO_LOWER=TRUE
```

H2数据库配置

Spring Boot 提供H2的支持。下面的目的是开启控制台，可以直观地查询H2数据库。

```
1 spring:
2   h2:
3     # 开启console 访问 默认false
4     console:
5       enabled: true
6       settings:
7         # 开启h2 console 跟踪 方便调试 默认 false
8         trace: true
9         # 允许console 远程访问 默认false
10        web-allow-others: true
11        # h2 数据库开启页面访问      http://localhost:8085/h2-console
12        # jdbc url 需要改为 jdbc:h2:mem:spring_security
13        # 该路径目前会被Spring Security禁止访问
14        path: /h2-console
```

如果想放开H2控制台访问，可以引入下面的配置：

```
1 @Bean
2 WebSecurityCustomizer webSecurityCustomizer(){
3   return web->web.ignoring().antMatchers("/h2-console");
4 }
```

脚本初始化

Spring Boot 2.6的脚本初始化配置做了一些改变，现在使用 `spring.sql.init` 来进行初始化SQL脚本：

```
1 spring
2   sql:
3     init:
4       # 初始化数据表 DDL
5       schema-locations: classpath:sql/dml.sql
6       # 初始化数据 DML
7       data-locations: classpath:sql/dml.sql
8       mode: embedded
```

默认情况下内置数据库才生效，而且这种方式一般建议演示的时候使用，生产不建议使用这种初始化方式。

数据库DDL脚本如下：

```
1 drop schema if exists `spring_security`;
2 create schema `spring_security`;
```

```

3  use
4      `spring_security`;
5
6  DROP TABLE IF EXISTS `sys_user`;
7  CREATE TABLE `sys_user`
8  (
9      `user_id`          varchar(64)  NOT NULL COMMENT '主键ID',
10     `username`         varchar(128) NOT NULL unique COMMENT '用户名',
11     `encode_password` varchar(1024) NOT NULL COMMENT '编码后的密码',
12     `expired`          tinyint(1)   NOT NULL DEFAULT 1 COMMENT '账户是否过期',
13     `locked`           tinyint(1)   NOT NULL DEFAULT 1 COMMENT '账户是否锁定',
14     `enabled`          tinyint(1)   NOT NULL DEFAULT 1 COMMENT '账户是否可用',
15     PRIMARY KEY (`user_id`)
16 ) ENGINE = InnoDB COMMENT '用户账户信息'
17     DEFAULT CHARSET = utf8mb4;
18
19  DROP TABLE IF EXISTS `role`;
20  CREATE TABLE `role`
21  (
22      `role_id`          varchar(64)  NOT NULL COMMENT '主键ID',
23      `role_name`        varchar(32)  NOT NULL UNIQUE COMMENT '角色标识',
24      `role_comment`    varchar(500)  DEFAULT NULL COMMENT '说明',
25      `enabled`          tinyint(1)  DEFAULT '1' COMMENT '是否可用',
26      PRIMARY KEY (`role_id`)
27 ) ENGINE = InnoDB
28     DEFAULT CHARSET = utf8mb4
29     COMMENT ='角色表';
30
31  DROP TABLE IF EXISTS `user_role`;
32  CREATE TABLE `user_role`
33  (
34      `user_role_id`    varchar(64)  NOT NULL COMMENT '主键ID',
35      `username`        varchar(64)  NOT NULL COMMENT '用户名',
36      `role_id`          varchar(64)  NOT NULL COMMENT '角色ID',
37      `role_name`        varchar(32)  NOT NULL COMMENT '角色标识',
38      `enabled`          tinyint(1)  DEFAULT '1' COMMENT '是否可用',
39      PRIMARY KEY (`user_role_id`)
40 ) ENGINE = InnoDB
41     DEFAULT CHARSET = utf8mb4
42     COMMENT ='用户角色表';

```

脚本仅为学习之用，生产需要根据业务自行设计。

DML脚本：

```

1 # 初始化一个用户 用户名felord 密码 123456 密码一定要hash一下可别学某站放明文密码
2 INSERT INTO spring_security.sys_user (user_id, username, encode_password,
3 expired, locked, enabled)
4 VALUES ('1', 'felord',
5 '{bcrypt}$2a$10$iNfPvjiZitM.DzuiPgpjYuLGI16pz0MTK8RIOgOLuPpsXZxcc0wu2', 1, 1, 1);
6 # 初始化两个角色
7 INSERT INTO spring_security.role (role_id, role_name, role_comment, enabled)
8 VALUES ('1', 'ROLE_USER', '用户角色', 1);
9 INSERT INTO spring_security.role (role_id, role_name, role_comment, enabled)
10 VALUES ('2', 'ROLE_ADMIN', '管理员角色', 1);
11 # 给用户赋予角色
12 INSERT INTO spring_security.user_role (user_role_id, username, role_id,
13 role_name, enabled)
14 VALUES ('1', 'felord', '1', 'ROLE_USER', 1);
15 INSERT INTO spring_security.user_role (user_role_id, username, role_id,
16 role_name, enabled)
17 VALUES ('2', 'felord', '2', 'ROLE_ADMIN', 1);

```

UserDetailsService配置

基于JDBC的 `UserDetailsService` 配置：

```

1 @Configuration(proxyBeanMethods = false)public class
2 UserDetailsServiceConfiguration {    private final MessageSourceAccessor messages
3 = SpringSecurityMessageSource.getAccessor();    /**     * 基于JDBC (Mybatis) 的用户角
4 色管理, 输入密码为{@code 123456}, 参考脚本{@code src/main/resources/sql/dml.sql}     *
5     * @param sysUserService 用户服务      * @param userRoleService 用户角色服务     *
6     * @return the userDetailsService     */    @Bean    UserDetailsService
7 jdbcUserDetailsService(SysUserService sysUserService, UserRoleService
8 userRoleService) {        return username -> {            // ① 从数据库查询用户信息
9             SysUser sysUser = sysUserService.findByUsername(username);            if
10            (sysUser == null) {                throw new
11 UsernameNotFoundException(this.messages.getMessage("JdbcDaoImpl.notFound",
12                         new Object[]{username}, "Username {0} not found"));            }
13            // ② 从数据库查询用户的角色信息            List<String> roles =
14            userRoleService.findByUsername(username);            if
15            (CollectionUtils.isEmpty(roles)) {                throw new
16 UsernameNotFoundException(this.messages.getMessage("JdbcDaoImpl.noAuthority",
17                         new Object[]{username}, "User {0} has no GrantedAuthority"));            }
18            String[] authorities = roles.toArray(new String[0]);
19            // ③ 组装UserDetails            return User.withUsername(username)
20            // 这里不是原始密码, 格式 {算法id}密文            // bcrypt算法: 123456
21            -> {bcrypt}$2a$10$iNfPvjiZitM.DzuiPgpjYuLGI16pz0MTK8RIOgOLuPpsXZxcc0wu2
22                .password(sysUser.getEncodePassword())
23                .authorities(authorities)            .build();        };    }

```

这里不再需要为 `User` 指派 `PasswordEncoder` ;另外注意写入数据库的密码的格式要符合 `{算法id}密文` 。添加注册用户时密码时，明文原始密码到密文的转换可以这么实现：

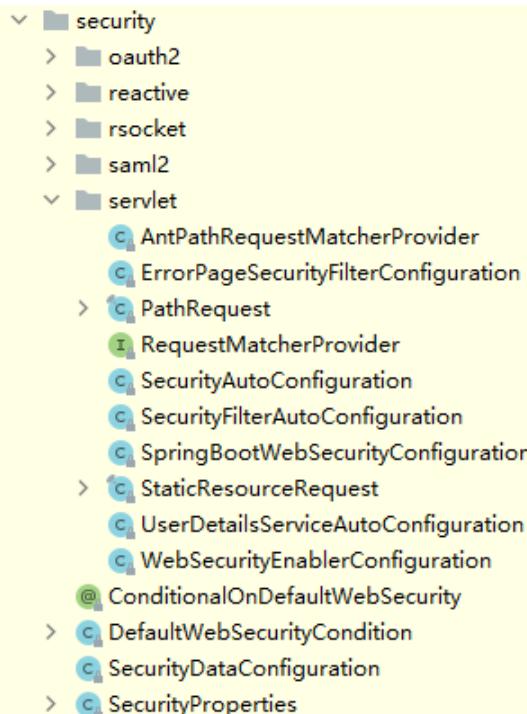
```
1 private final PasswordEncoder  
delegatingPasswordEncoder=PasswordEncoderFactories.createDelegatingPasswordEncoder()  
();@Testpublic void encodePassword(){ String rawPassword="this is a  
password"; String  
encodedPassword=delegatingPasswordEncoder.encode(rawPassword); // 摘要后为  
" {bcrypt} $2a$10$1mrdWJt.yRXAfE20j6vJTemALsMs.p3YWzBdnnnyF68N5Gfgf7xsv6";  
boolean matches=delegatingPasswordEncoder.matches(rawPassword,  
" {bcrypt} $2a$10$1mrdWJt.yRXAfE20j6vJTemALsMs.p3YWzBdnnnyF68N5Gfgf7xsv6");  
Assertions.assertTrue(matches); }
```

你也可以根据业务的需要定制自己的 `UserDetails`。



6-Spring Security自动配置

前面我们花了不少功夫来处理用户登录、密码机、用户储存的事。线索来自于 `UserDetailsServiceAutoConfiguration` 这个类。我有个小窍门告诉你，你要格外注意以 `AutoConfiguration` 结尾的类。这些类大都是自动配置类，Spring Boot通过类似SPI机制让这些配置自动生效。Spring Boot为Spring Security提供了一套默认的配置，供入门学习。相关的配置都在 `spring-boot-autoconfigure` 模块下的 `org.springframework.boot.autoconfigure.security` 包下：



建议你现在从IDE打开源码并找到上面的位置。

和 `UserDetailsServiceAutoConfiguration` 在同一包下自动配置类还有两个：

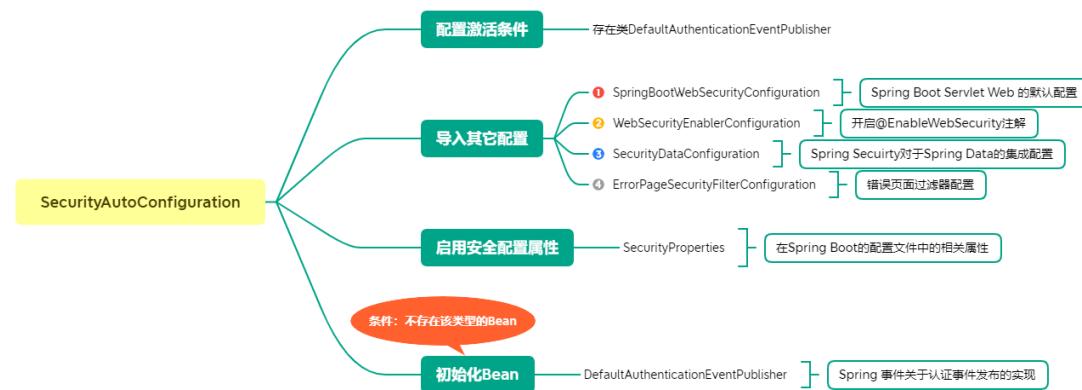
- `SecurityAutoConfiguration`
- `SecurityFilterAutoConfiguration`

这两个 Spring Security 的自动配置就没那么简单了，我们一个一个来看。

这里建议你打开源码亲自看一看下面这两个配置类。

SecurityAutoConfiguration

表面上 `SecurityAutoConfiguration` 的东西不多，但是实际上涉及的东西可真的不少。包含了认证事件发布、Spring Boot Servlet Web 安全配置、开启 WebSecurity 能力、Spring Data 支持以及错误界面的处理，通过下面这张图你可能会更清晰一些。



@EnableWebSecurity

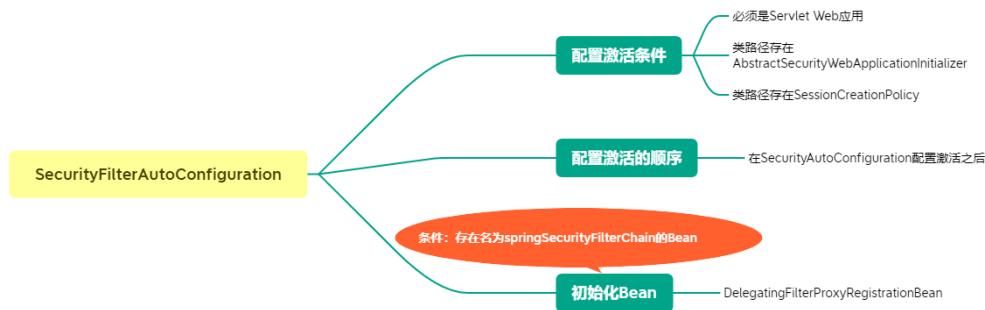
`@EnableWebSecurity` 这个注解非常重要，主要导入并开启了 Spring Security 的一些默认配置，详细的脉络如下图：



里面的几个概念都挺重要的，后面会结合场景进行讲解。这里说一个该注解非常有用的功能，当设置 `debug=true` 时，你的调用链路会打印到控制台中，非常方便初学者进行调用追踪和问题排查，**不过该功能不要在生产环境中开启**。

SecurityFilterAutoConfiguration

`SecurityFilterAutoConfiguration` 相较于 `SecurityAutoConfiguration` 要简单一些，它的作用是初始化一个 `DelegatingFilterProxyRegistrationBean` 的 Spring Bean。



初始化的这个Bean十分重要，我会在后面合适的时候来讲解它。

为什么要讲这两个配置类

如果你要集成 Spring Security，必然会从配置开始。默认提供的配置往往不能满足项目的需要，但是这些配置类声明了很多的激活条件，也就是 `Conditional` 系列注解，来帮助你打破常规默认，定制属于你自己的配置。这里引入这两个配置类，就是让你从宏观上对这两个类有个整体的认识。在接下来的几个篇幅中我会对这两个类进行“庖丁解牛”式的解析。



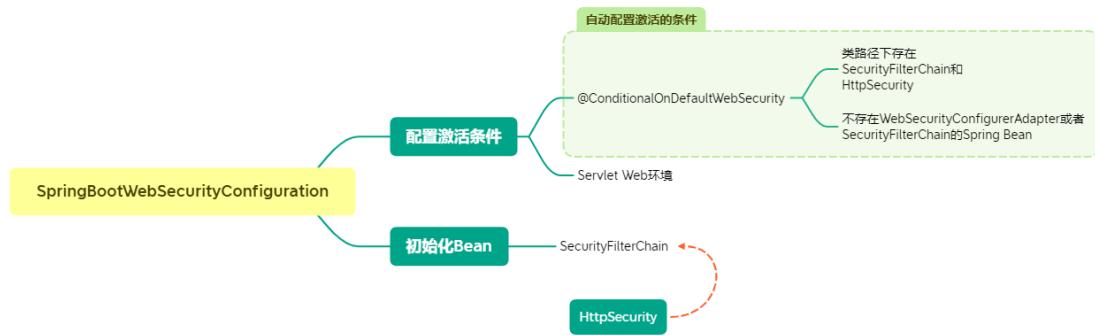
7-Spring Security过滤器链架构

在上一章中我对 Spring Security 在 Spring Boot 中的自动配置进行了宏观层面的分析。从现在起我会带着大家深入一些细节层面。在前几章学习 Spring Security 的时候你有没有下面这两个疑问：

- Spring Security 的登录是怎么配置的？
- Spring Security 的访问控制是怎么配置的？

SpringBootWebSecurityConfiguration

上面两个疑问的答案就在 `SpringBootWebSecurityConfiguration` 中。你可以按照下面个思维导图去理解这个自动配置：



`SpringBootWebSecurityConfiguration` 为 Spring Boot 应用提供了一套默认的 Spring Security 配置。

```
1 @Bean
2     @Order(SecurityProperties.BASIC_AUTH_ORDER)
3     SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws
4     Exception {
5
6     http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().http
7     Basic();
8         return http.build();
9     }
```

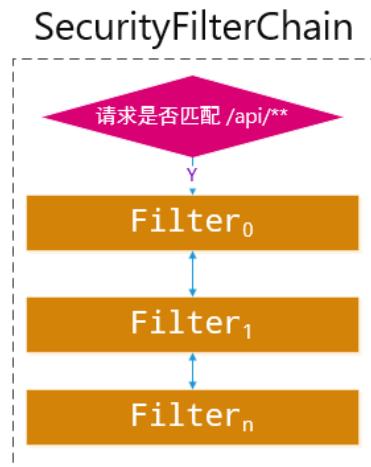
这里的配置为：所有的请求都必须是认证用户发起的，同时开启表单登录功能以及 `Http Basic Authentication` 认证功能。我们访问 `/foo/bar` 时需要登录认证并且能够进行表单登录就是这个配置起作用了。这个是我们日常开发需要自定义的，后面会专门去细讲。

SecurityFilterChain

从上面看得出 `HttpSecurity` 就是一个构建类，它的使命就是构建出一个 `SecurityFilterChain`：

```
1 public interface SecurityFilterChain {
2     // 当前请求是否匹配
3     boolean matches(HttpServletRequest request);
4     // 一揽子过滤器组成的有序过滤器链
5     List<Filter> getFilters();
6 }
```

当一个请求 `HttpServletRequest` 进入 `SecurityFilterChain` 时，会通过 `matches` 方法来确定是否满足条件进入过滤器链。就好比你是VIP走的是VIP通道，享受的是VIP的一系列待遇；你是普通用户，就走普通用户的通道并享受普通用户的待遇。



不管用户是哪种角色，都走的是一个过滤器链，一个应用中存在 `1-n` 个 `SecurityFilterChain`。那谁来管理多个 `SecurityFilterChain` 呢？

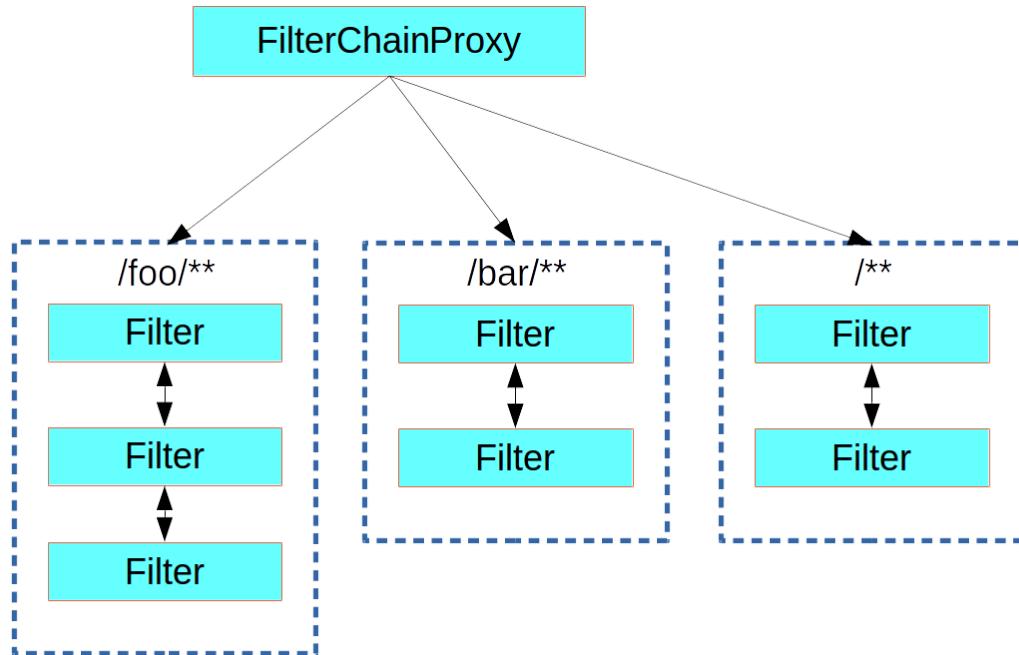
记住这个公式 `HttpSecurity ->SecurityFilterChain`。

FilterChainProxy

`FilterChainProxy` 是一个 `GenericFilterBean`（即使Servlet Filter又是Spring Bean），它管理了所有注入 Spring IoC 容器的 `SecurityFilterChain`。在我刚接触 `Spring Security` 的时候是这样配置 `FilterChainProxy` 的：

```
1  <bean id="myfilterChainProxy"
2      class="org.springframework.security.web.FilterChainProxy">
3          <constructor-arg>
4              <util:list>
5                  <security:filter-chain pattern="/do/not/filter*" filters="none"/>
6                  <security:filter-chain pattern="/**"
7                      filters="filter1,filter2,filter3"/>
8          </util:list>
9      </constructor-arg>
10 
```

根据不同的请求路径匹配走不同的 `SecurityFilterChain`。下面是示意图：



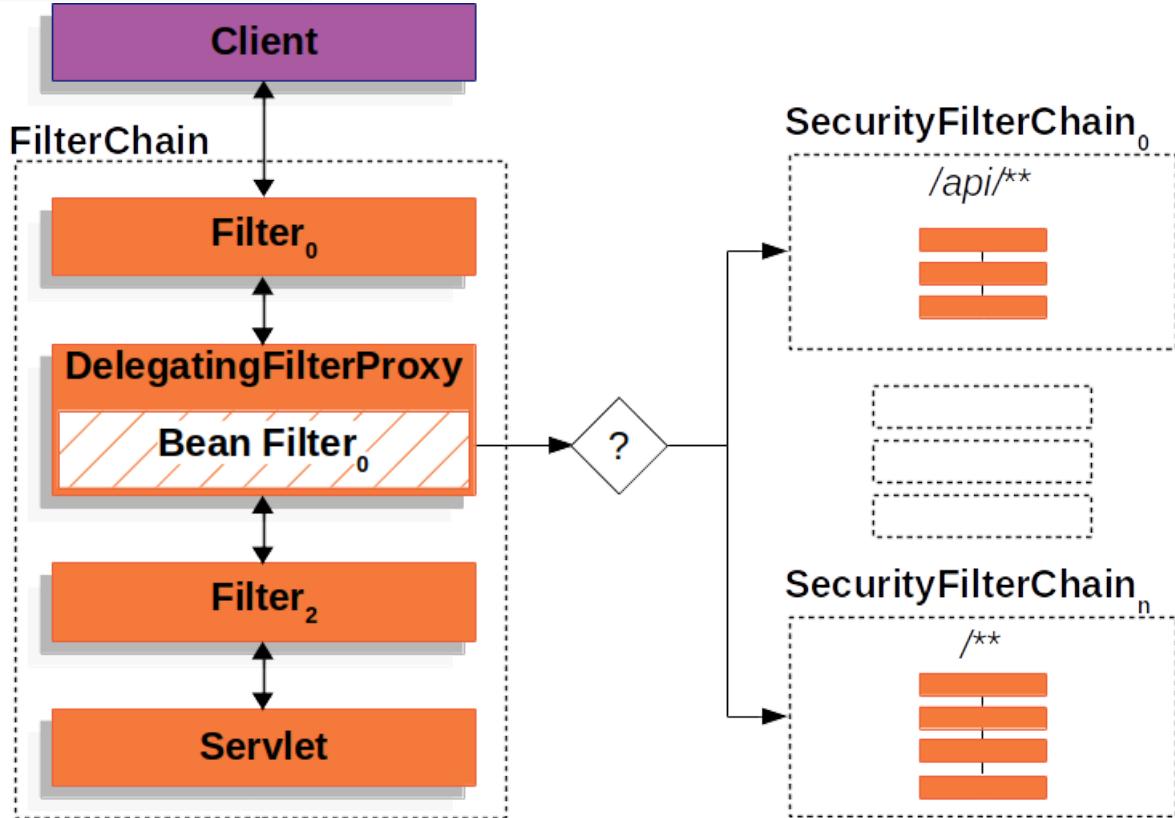
后面还会对接触这个类，现在你只需要明白上面这个图就行了。

请注意：在同一过滤器链中不建议有多个 `FilterChainProxy` 实例，而且不应将其作为单纯的过滤器使用，它只应该承担管理 `SecurityFilterChain` 的功能。

DelegatingFilterProxy

Servlet 容器和 Spring IoC 容器之间的 Filter 生命周期并不匹配。为了让 Spring IoC 容器管理 Filter 的生命周期，`FilterChainProxy` 便交由 Spring Web 下的 `DelegatingFilterProxy` 来代理。而且 `FilterChainProxy` 不会在添加到应用程序上下文的任何过滤器 Bean 上调用标准 Servlet 过滤器生命周期方法，`FilterChainProxy` 的生命周期方法会委托给 `DelegatingFilterProxy` 来执行。而

`DelegatingFilterProxy` 作为 Spring IoC 和 Servlet 的连接器存在。



简单总结

上面的三个概念非常重要，涉及到 Spring Security 的整个过滤器链体系。但是作为初学者来说，能看懂多少就看懂多少，不要纠结哪些没有理解，因为目前学习阶段的层次达不到是非常正常的。但是等你学完了 Spring Security 之后，这几个概念一定要搞明白。



8-Spring Security拦截特定请求

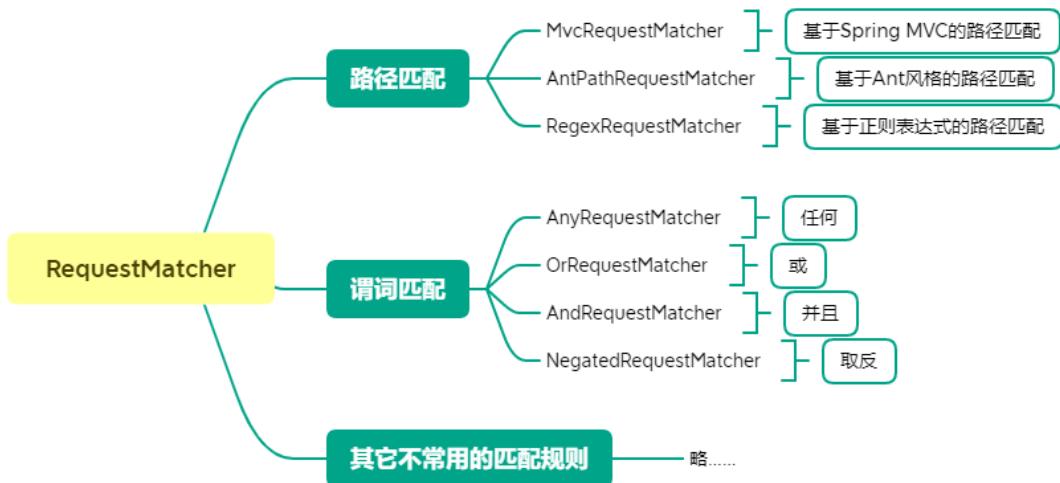
我们已经知道 `SecurityFilterChain` 决定了特定请求经过的过滤器链，那么 `SecurityFilterChain` 是如何匹配到特定请求的呢？

如何拦截特定的请求

只有满足了 `SecurityFilterChain` 的 `match` 方法的请求才能被该 `SecurityFilterChain` 处理，那如何配置才能让一个 `SecurityFilterChain` 处理特定的路径呢？

RequestMatcher

`HttpSecurity` 内置了 `RequestMatcher` 属性来处理路径匹配问题。`RequestMatcher` 可总结为以下几大类：



使用Ant路径：

```
1 httpSecurity.antMatcher("/foo/**");
```

如果你配置了全局的 `Servlet Path` 的话，例如 `/v1`，配置ant路径的话就要 `/v1/foo/**`，使用MVC风格可以保持一致：

```
1 httpSecurity.mvcMatcher("/foo/**");
```

另外MVC风格可以自动匹配后缀，例如 `/foo/hello` 可以匹配 `/foo/hello.do`、`/foo/hello.action` 等等。

另外你也可以使用正则表达式来进行路径匹配：

```
1 httpSecurity.regexMatcher("/foo/.+");
```

如果上面的都满足不了需要的话，你可以通过 `HttpSecurity.requestMatcher` 方法自定义匹配规则；如果你想匹配多个规则的话可以借助于 `HttpSecurity.requestMatchers` 方法来自由组合匹配规则，就像这样：

```
1 httpSecurity.requestMatchers(requestMatchers ->
2     requestMatchers.mvcMatchers("/foo/**")
3     .antMatchers("/admin/*get"));
```

一旦你配置了路径匹配规则的话，你会发现默认的表单登录404了，因为默认是 `/login`，你加了前缀后当然访问不到了。

使用场景

比如你后台管理系统和前端应用各自走不通的过滤器链，你可以根据访问路径来配置各自的过滤器链。例如：

```
1  /**
2   * Admin 过滤器链.
3   *
4   * @param http the http
5   * @return the security filter chain
6   * @throws Exception the exception
7   */
8  @Bean
9  SecurityFilterChain adminSecurityFilterChain(HttpSecurity http) throws
Exception {
10    http.requestMatchers(requestMatchers ->
requestMatchers.mvcMatchers("/admin/**"))
11      //todo 其它配置
12      return http.build();
13  }
14
15 /**
16  * App 过滤器链.
17  *
18  * @param http the http
19  * @return the security filter chain
20  * @throws Exception the exception
21  */
22  @Bean
23  SecurityFilterChain appSecurityFilterChain(HttpSecurity http) throws
Exception {
24    http.requestMatchers(requestMatchers ->
requestMatchers.mvcMatchers("/app/**"));
25      //todo 其它配置
26      return http.build();
27  }
```

思考一下 `HttpSecurity` 这个 Spring Bean 为什么能够公用。



微信搜一搜

码农小胖哥

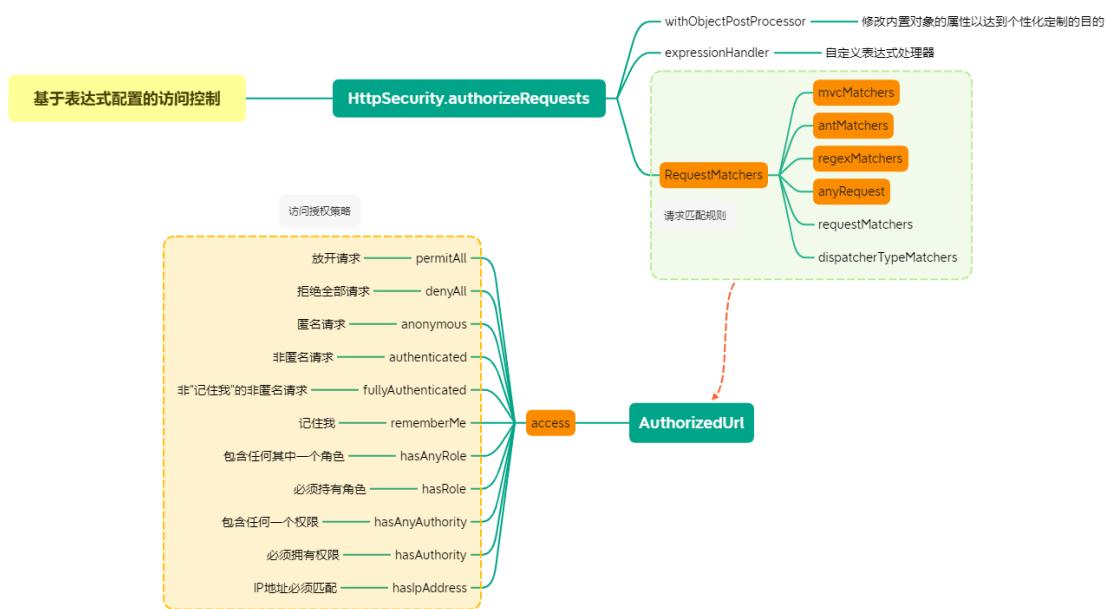
通过上一章我们已经能够对特定的请求进行拦截处理了。今天更进一步，来实现对特定请求的访问控制。后端的接口都是受权限控制的，A 用户的角色是会计，那么他就可以访问财务相关的资源。B 用户是人事，那么他只能访问人事相关的资源。在 Spring Security 中如何实现这种功能呢？

基于SpEL表达式的访问控制

`HttpSecurity` 提供了基于 SpEL 表达式的访问控制配置。在学习这个知识之前我们需要理清楚访问控制的要素有哪些，这个问题将贯穿整个访问控制的设计。我个人觉得可以抽象出来以下几点：

- 当前访问的主体。可以是终端用户、客户端的各种形式。
- 需要访问的资源标识。在 Web 应用中通常为 URI。
- 挑战（Challenge）。我觉得用这个词来描述相当合适，**挑战**是一个动作。访问主体发起对资源的访问规则进行挑战，挑战通过就可以访问该资源，挑战失败就无法访问该资源。在 Spring Security 中体现为用户持有的角色和资源所属角色是否有交集。

当前访问用户主体暂且不管，我们把资源的访问控制规则定义清楚后再来处理当前访问为用户主体的问题。为了方便学习，我用思维导图描述了基于表达式访问控制的架构，结合下面的图需要你对一些概念进行深入的了解。



资源配置

其实和 `SecurityFilterChain` 的匹配方式一样，都是通过定义一些 `RequestMatcher` 来进行资源配置，这里不再赘述相关的细节。比如 `/foo/**` 会匹配到 `/foo/hello`。

```
1 ExpressionUrlAuthorizationConfigurer<HttpSecurity>.MvcMatchersAuthorizedUrl  
2     mvcMatchersAuthorizedUrl=http  
3         .authorizeRequests().mvcMatchers("/foo/**");
```

访问控制

`MvcMatchersAuthorizedUrl` 提供了一系列的方法来对上面匹配到的资源进行访问控制。例如 `hasRole(String role)` 方法，等同于 `access("hasRole('" + "ROLE_" + role + "')")` 方法。其它的方法底层也都是 `access` 方法（参见上图），该方法将 URL 匹配和访问控制表达式封装成 `UrlMapping` 对象：

```

1     static final class UrlMapping {
2
3         private final RequestMatcher requestMatcher;
4
5         private final Collection<ConfigAttribute> configAttrs;
6
7         // 省略
8     }

```

`ConfigAttribute` 是一个资源的访问配置的抽象，有很多不同的实现。很多情况下你不会直接接触到这个抽象概念，但是如果你需要去定制一些规范实现的时候，可能需要去了解它。

`access` 可不单单局限于上面的一些表达式，还可以实现动态权限控制。`Spring Security` 扩展了对表达式进行了扩展，支持引用任何公开的 Spring Bean，假如我们有一个实现下列接口的 Spring Bean：

```

1  /**
2  * 角色检查器接口.
3  *
4  * @author n1
5  * @since 2021 /4/6 16:28
6  */
7  public interface RoleChecker extends InitializingBean {
8
9      /**
10      * Check boolean.
11      *
12      * @param authentication the authentication
13      * @param request         the request
14      * @return the boolean
15      */
16      boolean check(Authentication authentication, HttpServletRequest request);
17  }
18
19  @Bean
20  public RoleChecker roleChecker() {
21      return new RoleChecker() {
22          @Override
23          public boolean check(Authentication authentication,
24                  HttpServletRequest request) {
25              // todo 自行实现
26              return false;
27          }
28
29          @Override
30          public RoleChecker roleChecker() {
31              // todo 自行实现
32              return null;
33          }
34      }
35  }

```

这个 Bean 必须符合 `(authentication, request)-> something` 范式，也就是通过当前认证信息 `Authentication` 和当前请求信息 `HttpServletRequest` 进行一些逻辑处理。然后再结合 SpEL 表达式就能实现动态权限控制了，伪代码如下：

```
1 httpSecurity.authorizeRequests()  
2     .anyRequest()  
3         .access("@roleChecker.check(authentication, request)")
```

甚至连路径参数都支持：

```
1 httpSecurity.authorizeRequests()  
2     .antMatchers("/foo/{id}/**")  
3         .access("@roleChecker.check(authentication, #id)")
```

简单实践

所有的接口都要登录才能访问，并且 `/foo/**` 需要 `ROLE_USER` 角色才能访问；`/bar/**` 需要 `ROLE_USER` 或者 `ROLE_ADMIN` 角色；

上面的需求这样配置：

```
1 httpSecurity.authorizeRequests()  
2     .mvcMatchers("/foo/**").access("hasRole('USER')")  
3     .mvcMatchers("/bar/**").hasAnyAuthority("ROLE_USER", "ROLE_ADMIN")  
4     .anyRequest().authenticated()
```

访问用户 `UserDetails` 的 `authorities` 中需要包含配置中需要的角色才能访问 `/foo/**` 下的资源。

最好自己写个DEMO试一试。



10-HttpSecurity

看了前面几章后我们已经知道 `SecurityFilterChain` 由 `HttpSecurity` 构建，所以在特定调用链路中的一些访问策略都会在 `HttpSecurity` 中进行配置。以前会在 `SecurityConfigurerAdapter` 中配置 `HttpSecurity`，从 Spring Security 5.4 版本开始有了更加灵活的手段，不再需要去继承 `SecurityConfigurerAdapter`。有了初始化 `HttpSecurity` 的新方式。

```
1 @Bean(HTTPSECURITY_BEAN_NAME)  
2 @Scope("prototype")  
3 HttpSecurity httpSecurity() throws Exception {  
4     WebSecurityConfigurerAdapter.LazyPasswordEncoder passwordEncoder = new  
5         WebSecurityConfigurerAdapter.LazyPasswordEncoder(  
6             this.context);  
7     AuthenticationManagerBuilder authenticationBuilder = new  
8         WebSecurityConfigurerAdapter.DefaultPasswordEncoderAuthenticationManagerBuilder()
```

```
7             this.objectPostProcessor, passwordEncoder);
8
9     authenticationBuilder.parentAuthenticationManager(authenticationManager());
10    HttpSecurity http = new HttpSecurity(this.objectPostProcessor,
11        authenticationBuilder, createSharedObjects());
12        // @formatter:off
13        http
14            .csrf(withDefaults())
15            .addFilter(new WebAsyncManagerIntegrationFilter())
16            .exceptionHandling(withDefaults())
17            .headers(withDefaults())
18            .sessionManagement(withDefaults())
19            .securityContext(withDefaults())
20            .requestCache(withDefaults())
21            .anonymous(withDefaults())
22            .servletApi(withDefaults())
23            .apply(new DefaultLoginPageConfigurer<>());
24    http.logout(withDefaults());
25    // @formatter:on
26    return http;
27 }
```

这里会构建基于原型的 `HttpSecurity` Bean，并且初始化了一些默认配置供我们来使用。涉及 Spring Security 的日常开发都是围绕这个类进行的，所以这个类是学习 Spring Security 的重中之重。

基于原型（`prototype`）的 Spring Bean 的一个典型应用场景，

基本配置

日常我们使用的一些配置项如下：

方法	说明
requestMatchers()	为 <code>SecurityFilterChain</code> 提供 URL 拦截策略，具体还提供了 <code>antMatcher</code> 和 <code>mvcMatcher</code>
openidLogin()	用于基于 <code>OpenId</code> 的验证
headers()	将安全标头添加到响应，比如说简单的 XSS 保护
cors()	配置跨域资源共享（CORS）
sessionManagement()	配置会话管理
portMapper()	配置一个 <code>PortMapper(HttpSecurity#(getSharedObject(class)))</code> ，其他提供 <code>SecurityConfigurer</code> 的对象使用 <code>PortMapper</code> 从 HTTP 重定向到 HTTPS 或者从 HTTPS 重定向到 HTTP。默认情况下，Spring Security 使用一个 <code>PortMapperImpl</code> 映射 HTTP 端口 8080 到 HTTPS 端口 8443，HTTP 端口 80 到 HTTPS 端口 443
jee()	配置基于容器的预认证。在这种情况下，认证由 Servlet 容器管理
x509()	配置基于 x509 的预认证
rememberMe	配置“记住我”的验证
authorizeRequests()	基于使用 <code>HttpServletRequest</code> 限制访问
requestCache()	配置请求缓存
exceptionHandling()	配置错误处理
securityContext()	在 <code>HttpServletRequests</code> 之间的 <code>SecurityContextHolder</code> 上设置 <code>SecurityContext</code> 的管理。当使用 <code>WebSecurityConfigurerAdapter</code> 时，这将自动应用
servletApi()	将 <code>HttpServletRequest</code> 方法与其上找到的值集成到 <code>SecurityContext</code> 中。当使用 <code>WebSecurityConfigurerAdapter</code> 时，这将自动应用
csrf()	添加 CSRF 支持，使用 <code>WebSecurityConfigurerAdapter</code> 时，默认启用
logout()	添加退出登录支持。当使用 <code>WebSecurityConfigurerAdapter</code> 时，这将自动应用。默认情况是，访问 URL “/ logout”，使 HTTP Session 无效来清除用户，清除已配置的任何 <code>#rememberMe()</code> 身份验证，清除 <code>SecurityContextHolder</code> ，然后重定向到 <code>/login?success</code>
anonymous()	配置匿名用户的表示方法。当与 <code>WebSecurityConfigurerAdapter</code> 结合使用时，这将自动应用。默认情况下，匿名用户将使用 <code>org.springframework.security.authentication.AnonymousAuthenticationToken</code> 表示，并包含角色 <code>ROLE_ANONYMOUS</code>
authenticationManager()	配置 <code>AuthenticationManager</code>
authenticationProvider()	添加 <code>AuthenticationProvider</code>
formLogin()	指定支持基于表单的身份验证。如果未指定 <code>FormLoginConfigurer#LoginPage(String)</code> ，则将生成默认登录页面
oauth2Login()	根据外部 OAuth 2.0 或 OpenID Connect 1.0 提供程序配置身份验证
oauth2Client()	OAuth 2.0 客户端相关的配置
oauth2ResourceServer()	OAuth 2.0 资源服务器相关的配置
requiresChannel()	配置通道安全。为了使该配置有用，必须提供至少一个到所需信道的映射
httpBasic()	配置 Http Basic 验证
addFilter()	添加一个已经在内置过滤器注册表注册过的过滤器实例或者子类
addFilterBefore()	在指定的 Filter 类之前添加过滤器

方法	说明
addFilterAt()	在指定的Filter类的位置添加过滤器
addFilterAfter()	在指定的Filter类的之后添加过滤器
and()	连接以上策略的连接器，用来组合安全策略。实际上就是“而且”的意思

高级玩法

新手建议先把上面的基本玩法有选择的弄明白，然后有精力的话去研究下 `HttpSecurity` 的高级玩法。

apply

这个方法用来把其它的一些配置合并到当前的配置中去，形成插件化，支持 `SecurityConfigurerAdapter` 或者 `SecurityConfigurer` 的实现。其实内置的一些配置都是以这种形式集成到 `HttpSecurity` 中去的。例如文章开头的配置中有默认登录页面相关的配置：

```
1     httpSecurity.apply(new DefaultLoginPageConfigurer<>());
```

objectPostProcessor

配置一个自定义 `ObjectPostProcessor`。`ObjectPostProcessor` 可以改变某些配置内部的机制，这些配置往往不直接对外提供操作接口。

获取、移除配置类

`getConfigurer` 用来获取已经 `apply` 的配置类；`getConfigurers` 用来获取已经 `apply` 某个类型的所有的配置类。

配置、获取SharedObject

`SharedObject` 是在配置中进行共享的一些对象，`HttpSecurity` 共享了一些非常有用的对象可以供外部使用，比如 `AuthenticationManager`。相关的方法有 `setSharedObject`、`getSharedObject`、`getSharedObjects`。

获取SecurityFilterChain

`HttpSecurity` 也提供了构建目标对象 `SecurityFilterChain` 的实例的方法。你可以通过 `build()` 来对配置进行初次构建；也可以通过 `getObject()` 来获取已经构建的实例；甚至你可以使用 `getOrBuild()` 来进行直接获取实例或者构建实例。

这一篇非常重要

这一篇的东西非常重要，不是马上就能掌握的，需要有些耐心。



微信搜一搜

码农小胖哥

11-过滤器的注册

`HttpSecurity` 中的过滤器顺序是怎么维护的？我想很多开发者都对这个问题感兴趣。本篇我和大家一起探讨下这个问题。

`HttpSecurity` 包含了一个成员变量 `FilterOrderRegistration`，这个类是一个内置过滤器注册表。至于这些过滤器的作用，不是本文介绍的重点，有兴趣可以去看看 `FilterOrderRegistration` 的源码。

内置过滤器的顺序

`FilterOrderRegistration` 维护了一个变量 `filterToOrder`，它记录了类之间的顺序和上下之间的间隔步长。我们复制了一个 `FilterOrderRegistration` 来直观感受一下过滤器的顺序：

```
1      CopyFilterOrderRegistration filterOrderRegistration = new
2      CopyFilterOrderRegistration();
3          // 获取内置过滤器 此方法并未提供
4          Map<String, Integer> filterToOrder =
5              filterOrderRegistration.getFilterToOrder();
6          TreeMap<Integer, String> orderToFilter = new TreeMap<>();
7          filterToOrder.forEach((name, order) -> orderToFilter.put(order, name));
8          orderToFilter.forEach((order, name) -> System.out.println(" 顺序: " +
9              order+" 类名: " + name ));
```

打印结果：

```
顺序: 100 类名: org.springframework.security.web.access.channel.ChannelProcessingFilter
顺序: 300 类名: org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter
顺序: 400 类名: org.springframework.security.web.context.SecurityContextPersistenceFilter
顺序: 500 类名: org.springframework.security.web.header.HeaderWriterFilter
顺序: 600 类名: org.springframework.web.filter.CorsFilter
顺序: 700 类名: org.springframework.security.web.csrf.CsrfFilter
顺序: 800 类名: org.springframework.security.web.authentication.logout.LogoutFilter
顺序: 900 类名: org.springframework.security.oauth2.client.web.OAuth2AuthorizationRequestRedirectFilter
顺序: 1000 类名: org.springframework.security.saml2.provider.service.servlet.filter.Saml2WebSSoAuthenticationRequestFilter
顺序: 1100 类名: org.springframework.security.web.authentication.preauth.X509X509AuthenticationFilter
顺序: 1200 类名: org.springframework.security.web.authentication.preauth.AbstractPreAuthenticatedProcessingFilter
顺序: 1300 类名: org.springframework.security.cas.web.CasAuthenticationFilter
顺序: 1400 类名: org.springframework.security.oauth2.client.web.OAuth2LoginAuthenticationFilter
顺序: 1500 类名: org.springframework.security.saml2.provider.service.servlet.filter.Saml2WebSSoAuthenticationFilter
顺序: 1600 类名: org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter
顺序: 1800 类名: org.springframework.security.openid.OpenIDAuthenticationFilter
顺序: 1900 类名: org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter
顺序: 2000 类名: org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter
顺序: 2100 类名: org.springframework.security.web.session.ConcurrentSessionFilter
顺序: 2200 类名: org.springframework.security.web.authentication.www.DigestAuthenticationFilter
顺序: 2300 类名: org.springframework.security.oauth2.server.resource.web.BearerTokenAuthenticationFilter
顺序: 2400 类名: org.springframework.security.web.authentication.www.BasicAuthenticationFilter
顺序: 2500 类名: org.springframework.security.web.savedrequest.RequestCacheAwareFilter
顺序: 2600 类名: org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter
顺序: 2700 类名: org.springframework.security.web.jaasapi.JaasApiIntegrationFilter
顺序: 2800 类名: org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFilter
顺序: 2900 类名: org.springframework.security.web.authentication.AnonymousAuthenticationFilter
顺序: 3000 类名: org.springframework.security.oauth2.client.web.OAuth2AuthorizationCodeGrantFilter
顺序: 3100 类名: org.springframework.security.web.session.SessionManagementFilter
顺序: 3200 类名: org.springframework.security.web.access.ExceptionTranslationFilter
顺序: 3300 类名: org.springframework.security.web.access.intercept.FilterSecurityInterceptor
顺序: 3400 类名: org.springframework.security.web.access.intercept.AuthorizationFilter
顺序: 3500 类名: org.springframework.security.web.authentication.switchuser.SwitchUserFilter
```

我们可以看得出内置过滤器之间的位置是相对固定的，除了第一个跟第二个步长为 200 外，其它步长为 100。

内置过滤器并非一定会生效，仅仅是预置了它们的排位，需要通过 `HttpSecurity` 的 `addFilterXXXX` 系列方法显式添加才行。

注册过滤器的逻辑

`FilterOrderRegistration` 提供了一个 `put` 方法：

```
1 void put(Class<? extends Filter> filter, int position) {  
2     String className = filter.getName();  
3     // 如果这个类已经注册就忽略  
4     if (this.filterToOrder.containsKey(className)) {  
5         return;  
6     }  
7     // 如果没有注册就注册顺序。  
8     this.filterToOrder.put(className, position);  
9 }
```

从这个方法我们可以得到几个结论：

- 内置的 34 个过滤器是有固定序号的，不可被改变。
- 新加入的过滤器的类全限定名是不能和内置过滤器重复的。
- 新加入的过滤器的顺序是可以和内置过滤器的顺序重复的。

获取已注册过滤器的顺序值

`FilterOrderRegistration` 还提供了一个 `getOrder` 方法：

```
1 Integer getOrder(Class<?> clazz) {  
2     // 如果类Class 或者 父类Class 名为空就返回null  
3     while (clazz != null) {  
4         Integer result = this.filterToOrder.get(clazz.getName());  
5         // 如果获取到顺序值就返回  
6         if (result != null) {  
7             return result;  
8         }  
9         // 否则尝试去获取父类的顺序值  
10        clazz = clazz.getSuperclass();  
11    }  
12    return null;  
13 }
```

HttpSecurity维护过滤器的方法

接下来我们分析一下 `HttpSecurity` 维护过滤器的几个方法。

`addFilterAtOffsetOf`

`addFilterAtOffsetOf` 是一个 `HttpSecurity` 的内置私有方法。 `Filter` 是想要注册到 `DefaultSecurityFilterChain` 中的过滤器， `offset` 是向右的偏移值， `registeredFilter` 是已经注册到 `FilterOrderRegistration` 的过滤器，而且 `registeredFilter` 没有注册的话会空指针。

```
1     private HttpSecurity addFilterAtOffsetOf(Filter filter, int offset, Class<?>
2         extends Filter> registeredFilter) {
3         // 首先会根据registeredFilter的顺序和偏移值来计算filter的
4         int order = this.filterOrders.getOrder(registeredFilter) + offset;
5         // filter添加到集合中待排序
6         this.filters.add(new OrderedFilter(filter, order));
7         // filter注册到 FilterOrderRegistration
8         this.filterOrders.put(filter.getClass(), order);
9         return this;
10    }
```

务必记着 `registeredFilter` 一定是已注册入 `FilterOrderRegistration` 的 `Filter`。

addFilter系列方法

这里以 `addFilterAfter` 为例。

```
1     @Override
2     public HttpSecurity addFilterAfter(Filter filter, Class<? extends Filter>
3         afterFilter) {
4         return addFilterAtOffsetOf(filter, 1, afterFilter);
5     }
```

`addFilterAfter` 是将 `filter` 的位置置于 `afterFilter` 后一位，假如 `afterFilter` 顺序值为 400，则 `filter` 顺序值为 401。`addFilterBefore` 和 `addFilterAt` 逻辑和 `addFilterAfter` 仅仅是偏移值的区别，这里不再赘述。

`addFilter` 的方法比较特殊：

```
1     @Override
2     public HttpSecurity addFilter(Filter filter) {
3         Integer order = this.filterOrders.getOrder(filter.getClass());
4         if (order == null) {
5             throw new IllegalArgumentException("The Filter class " +
filter.getClass().getName()
+ " does not have a registered order and cannot be added
without a specified order. Consider using addFilterBefore or addFilterAfter
instead.");
6         }
7         this.filters.add(new OrderedFilter(filter, order));
8         return this;
9     }
10    }
```

`filter` 必须是已经注册到 `FilterOrderRegistration` 的 `Filter`，这意味着它可能是内置的 `Filter`，也可能是先前通过 `addFilterBefore`、`addFilterAt` 或者 `addFilterAfter` 注册的非内置 `Filter`。

问题来了

之前看到一个问题，如果 `HttpSecurity` 注册两个重复序号的 `Filter` 会是怎么样的顺序？我们先来看下排序的机制：

```
1     // filters
2     private List<OrderedFilter> filters = new ArrayList<>();
3     //排序
4     this.filters.sort(OrderComparator.INSTANCE);
```

看了下 `OrderComparator` 源码，其实还是通过数字的自然排序，数字越小越靠前。如果数字相同，索引越小越靠前。也就是同样的序号，谁先 `add` 到 `filters` 谁就越靠前。



12-表单登录的秘密

为什么Spring Security的表单登录配置这么简单，仅仅添加了 `formLogin()` 方法就可以了呢？

FormLoginConfigurer

我们来看看 `formLogin()` 方法：

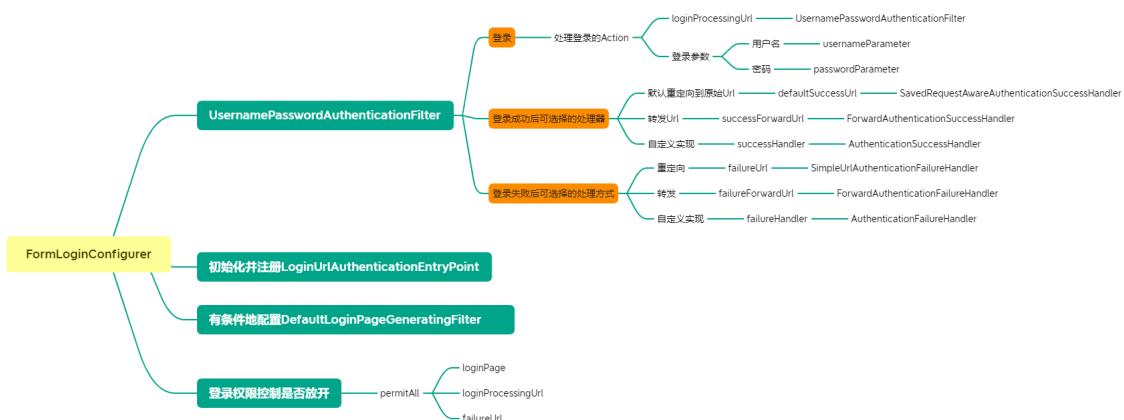
```
1  public FormLoginConfigurer<HttpSecurity> formLogin() throws Exception{
2      return getOrApply(new FormLoginConfigurer<>());
3  }
```

这个方法是说如果 `FormLoginConfigurer` 已经存在与配置就使用已有的，如果没有就初始化一个，看来表单登录的秘密就在 `FormLoginConfigurer` 中。

`FormLoginConfigurer` 用来配置过滤器 `UsernamePasswordAuthenticationFilter`。该过滤器用来处理表单提交的身份认证流程。`FormLoginConfigurer` 提供的配置项方法有：

方法	方法说明
<code>loginPage</code>	配置登录页面URL，默认 <code>/login</code> ，由Spring Security提供。区别于接口。
<code>loginProcessingUrl</code>	实际表单向后台提交用户信息的 <code>Action</code> ，再由过滤器 <code>UsernamePasswordAuthenticationFilter</code> 拦截处理，该 <code>Action</code> 其实不会处理任何逻辑。默认使用 <code>/login</code> 。
<code>usernameParameter</code>	用来自定义用户名参数名，默认 <code>username</code> 。
<code>passwordParameter</code>	用来自定义用户密码名，默认 <code>password</code> 。
<code>failureUrl</code>	登录失败后会重定向到此URL。
<code>failureForwardUrl</code>	登录失败会转发到此URL，可定义一个 Controller（控制器）来处理返回值。
<code>failureHandler</code>	实现自定义认证失败处理器 <code>AuthenticationFailureHandler</code> 处理认证失败逻辑。
<code>defaultSuccessUrl</code>	默认登陆成功后重定向到此URL。
<code>successForwardUrl</code>	登录成功后转发到此URL。
<code>successHandler</code>	实现自定义认证成功处理器 <code>AuthenticationSuccessHandler</code> 处理认证成功逻辑。
<code>permitAll</code>	form 表单登录是否放开

你可以顺着下面的思维导图进行梳理 `FormLoginConfigurer`：



UsernamePasswordAuthenticationFilter

这个过滤器是处理用户密码认证的默认过滤器，十分重要，`FormLoginConfigurer` 就是用来配置它的。它继承于 `AbstractAuthenticationProcessingFilter`。它的作用是拦截登录请求并获取账号和密码，然后把账号密码封装到认证凭据 `UsernamePasswordAuthenticationToken` 中，然后把凭据交给特定配置的认证管理器 `AuthenticationManager` 去作认证。

```

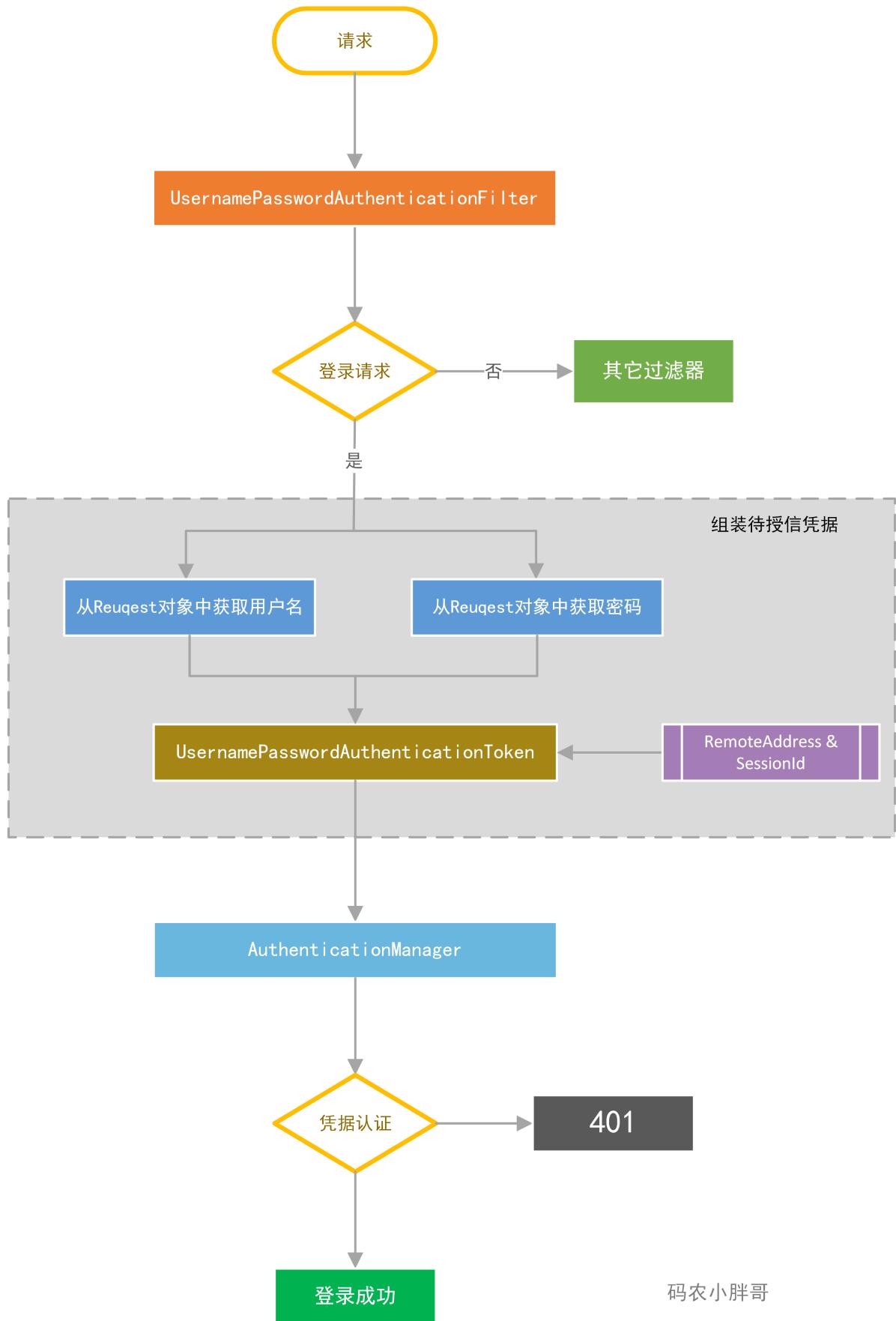
1 public class UsernamePasswordAuthenticationFilter extends
2     AbstractAuthenticationProcessingFilter {
3     // 默认取账户名、密码的key
  
```

```
4     public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "username";
5     public static final String SPRING_SECURITY_FORM_PASSWORD_KEY = "password";
6     // 可以通过对应的set方法修改
7     private String usernameParameter = SPRING_SECURITY_FORM_USERNAME_KEY;
8     private String passwordParameter = SPRING_SECURITY_FORM_PASSWORD_KEY;
9     // 默认只支持 POST 请求
10    private boolean postOnly = true;
11
12    // 初始化一个用户密码 认证过滤器 默认的登录uri 是 /login 请求方式是POST
13    public UsernamePasswordAuthenticationFilter() {
14        super(new AntPathRequestMatcher("/login", "POST"));
15    }
16
17    // 实现其父类 AbstractAuthenticationProcessingFilter 提供的钩子方法 用去尝试认证
18    public Authentication attemptAuthentication(HttpServletRequest request,
19                                                 HttpServletResponse response)
20                                         throws AuthenticationException {
21        // 判断请求方式是否是POST
22        if (postOnly && !request.getMethod().equals("POST")) {
23            throw new AuthenticationServiceException(
24                "Authentication method not supported: " +
25                request.getMethod());
26
27        // 先去 HttpServletRequest 对象中获取账号名、密码
28        String username = obtainUsername(request);
29        String password = obtainPassword(request);
30
31        if (username == null) {
32            username = "";
33        }
34
35        if (password == null) {
36            password = "";
37        }
38
39        username = username.trim();
40
41        // 然后把账号名、密码封装到 一个认证Token对象中，这是就是一个通行证，但是这时的状态时
42        // 不可信的，一旦通过认证就变为可信的
43        UsernamePasswordAuthenticationToken authRequest = new
44        UsernamePasswordAuthenticationToken(
45            username, password);
46
47        // 会将 HttpServletRequest 中的一些细节 request.getRemoteAddr()
48        // request.getSession 存入的到Token中
49        setDetails(request, authRequest);
50
51        // 然后 使用 父类中的 AuthenticationManager 对Token 进行认证
52        return this.getAuthenticationManager().authenticate(authRequest);
53    }
54
55    // 获取密码 很重要 如果你想改变获取密码的方式要么在此处重写，要么通过自定义一个前置的过滤器
56    // 保证能此处能get到
57    @Nullable
58    protected String obtainPassword(HttpServletRequest request) {
59        return request.getParameter(passwordParameter);
60    }
61
```

```
56
57     // 获取账户很重要 如果你想改变获取密码的方式要么在此处重写，要么通过自定义一个前置的过滤器保
58     // 证能此处能get到
59     @Nullable
60     protected String obtainUsername(HttpServletRequest request) {
61         return request.getParameter(usernameParameter);
62     }
63
64     // 参见上面对应的说明为凭据设置一些请求细节
65     protected void setDetails(HttpServletRequest request,
66                               UsernamePasswordAuthenticationToken authRequest) {
67
68         authRequest.setDetails(authenticationDetailsSource.buildDetails(request));
69     }
70
71     // 设置账户参数的key
72     public void setUsernameParameter(String usernameParameter) {
73         Assert.hasText(usernameParameter, "Username parameter must not be empty
74         or null");
75         this.usernameParameter = usernameParameter;
76     }
77
78     // 设置密码参数的key
79     public void setPasswordParameter(String passwordParameter) {
80         Assert.hasText(passwordParameter, "Password parameter must not be empty
81         or null");
82         this.passwordParameter = passwordParameter;
83     }
84
85
86     public final String getUsernameParameter() {
87         return usernameParameter;
88     }
89
90     public final String getPasswordParameter() {
91         return passwordParameter;
92     }
93 }
```

为了加强对这个流程的理解，我特意画了一个流程图来说明

`UsernamePasswordAuthenticationFilter` 执行认证的大致流程：



具体 `AuthenticationManager` 是如何认证的先不要纠结，你现在还把握不住，你只需要知道认证这件事管就行了。结果要么成功，要么失败。相关逻辑由下面两个抽象接口来处理。

AuthenticationSuccessHandler

用户认证成功后执行逻辑的策略的抽象。通俗说就是，我认证成功了，我接下来该干嘛，一般通过重定向或转发来处理。典型的场景就是用户通过提交登录表单登录后，应用程序需要决定他们之后应该重定向到哪里，是首页，还是返回登录成功的标识等等。

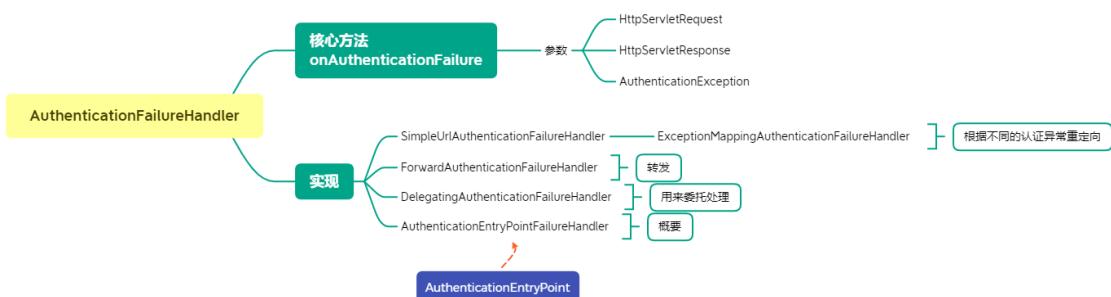


AuthenticationFailureHandler

有认证成功就有认证失败。和 `AuthenticationSuccessHandler` 相反，

AuthenticationFailureHandler

用来处理认证失败相关的逻辑，一般也通过重定向或转发来处理。当用户认证失败的时候会抛出一个认证失败的异常 `AuthenticationException`，你可以借助于这个接口进行相关的逻辑处理。



实践

Form登录成功后返回用户信息、失败后返回401可以通过上面两个Handler接口来实现，过于简单这里就不再演示了。



13-认证管理器

认证管理器接口 `AuthenticationManager` 在前面的文章中已经多次遇到它了。今天时机已经成熟，该把它搞明白了。用户认证就是由它来处理的。

AuthenticationManager的配置

AuthenticationManager 由 spring-security-config 模块下的配置类

AuthenticationConfiguration 负责配置的。初始化 AuthenticationManager 的方法为：

```
1 public AuthenticationManager getAuthenticationManager() throws Exception {
2     // 先判断 AuthenticationManager 是否初始化
3     if (this.authenticationInitialized) {
4         // 如果已经初始化 那么直接返回初始化的
5         return this.authenticationManager;
6     }
7     // 否则就去 Spring IoC 中获取其构建类
8     AuthenticationManagerBuilder authBuilder =
9         this.applicationContext.getBean(AuthenticationManagerBuilder.class);
10    // 如果不是第一次构建 好像是每次总要通过Builder来进行构建
11    if (this.buildingAuthenticationManager.getAndSet(true)) {
12        // 返回一个委托的AuthenticationManager
13        return new AuthenticationManagerDelegator(authBuilder);
14    }
15    // 如果是第一次通过Builder构建 将全局的认证配置整合到Builder中 那么以后就不用再整合全局的
16    // 配置了
17    for (GlobalAuthenticationConfigurerAdapter config : globalAuthConfigurers) {
18        authBuilder.apply(config);
19    }
20    // 构建AuthenticationManager
21    authenticationManager = authBuilder.build();
22    // 如果构建结果为null
23    if (authenticationManager == null) {
24        // 再次尝试去Spring IoC 获取懒加载的 AuthenticationManager Bean
25        authenticationManager = getAuthenticationManagerBean();
26    }
27    // 更新初始化状态避免多次初始化
28    this.authenticationInitialized = true;
29    return authenticationManager;
30 }
```

你可以借助于思维导图去理解 AuthenticationManager 的初始化流程：



自定义

从上面可以归纳出以下几种自定义的方法：

- 自定义 GlobalAuthenticationConfigurerAdapter 并注入 Spring IoC 来修改 AuthenticationManagerBuilder，不限制数量，但是要注意有排序问题。
- 自定义 UserDetailsService 到 Spring IoC 来定制用户信息提取服务（已经详细介绍过），限量一个。
- 自定义 PasswordEncoder Spring IoC 来定制密码编码方式，一般没有必要，除非强制要求用什么 SM3 之类的算法，限量一个。

- 自定义 `UserDetailsService` 到 Spring IoC 来定制用户信息密码更新服务，限量一个。
- 自定义 `AuthenticationProvider` 到 Spring IoC 来定制认证策略，限量一个。

这五个知识点非常重要，是个性化开发必会知识点。

ProviderManager

根据 `AuthenticationManagerBuilder` 的 `performBuild` 方法可以知道构建出来的 `AuthenticationManager` 的实际类型为 `ProviderManager`。 `ProviderManager` 我这么讲你可能更好理解，有一个叫 `ProviderManager` 的项目经理整天摸鱼不干活，上面派下来的需求他就按照类型去匹配，是前端的活就去找前端，是后端就去找后端，活干完了他去汇报，老板夸他活干得漂亮，个人能力强，你说气人不气人。从这里我们知道 `ProviderManager` 有两个功能：

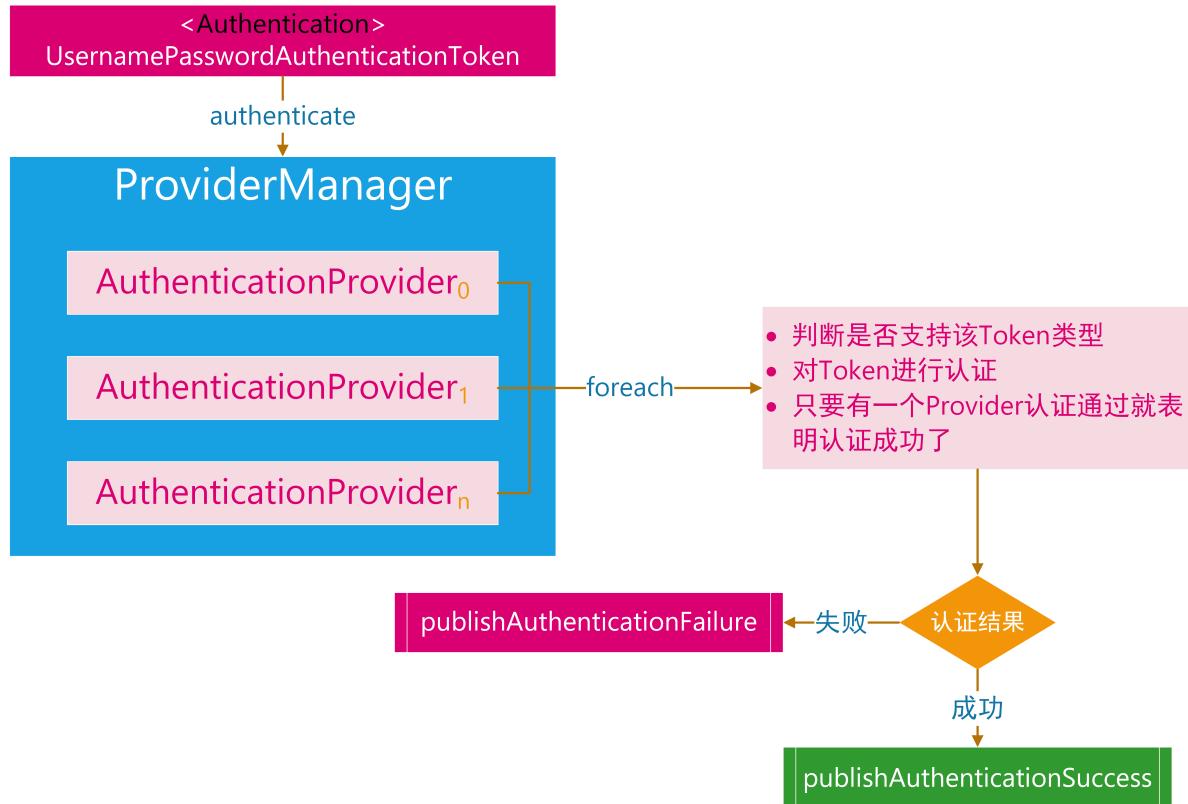
- 管理手下的工具人 `AuthenticationProvider`。这个比较重要，稍后弄清楚它。
- 汇报认证工作 `AuthenticationEventPublisher`。认证成功了就发个认证成功的 Spring 事件；认证失败了就发个认证失败的 Spring 事件。

AuthenticationProvider

这个接口承担的作用上面已经提了，我们来通过源码看看细节：

```
1  public interface AuthenticationProvider {  
2  
3      /**  
4          * 具体处理认证的方法，传入待授信认证信息。  
5          * 如果认证成功就返回认证成功的信息；失败将抛出异常  
6          */  
7      Authentication authenticate(Authentication authentication) throws  
8          AuthenticationException;  
9  
10     /**  
11         * 这个用来匹配限制认证的类型，以做到专用。  
12         */  
13     boolean supports(Class<?> authentication);  
14 }
```

每一个 `AuthenticationProvider` 都只支持特定类型的 `Authentication`，`ProviderManager` 会遍历所有的 `AuthenticationProvider` 来匹配 `Authentication` 并认证，只要有一个 `AuthenticationProvider` 被匹配到并认证成功，那么就认为认证成功，所有的都没有通过才认为是认证失败。认证成功后的 `Authentication` 就变成授信凭据，并触发认证成功的事件，认证失败的就抛出异常触发认证失败的事件。



从这里我们可以看出认证管理器 `AuthenticationManager` 针对特定的 `Authentication` 提供了特定的认证功能，我们可以利用这个机制实现多种认证并存。



14-认证是什么

认证是什么？简单说就是证明“你就是你”。你说你是你，你总得拿出凭证吧。证明了你确实是您，你才能享受你应该享受的权利。

Authentication

所以在 `Spring Security` 中抽象了一个认证的概念接口 `Authentication`。认证主体的认证信息、状态都由 `Authentication` 来负责维护。

Authentication

Principal

Credentials

Authorities

进一步来说 **Authentication** 是一个认证信息的容器，包含了用户的身份（principal）、认证凭据（credentials）和赋予的权限（authorities）。

- principal 对于一个实体个体来说在不同的系统和场景中充当着不同的身份。在公司你是一名工程师，在家的时候你可能是父亲或者儿子（女儿）。
- credentials 那么对于不同身份的用户自然有对应的认证凭据。在公司能直接证明你是工程师的就是你的工牌之类的，在家如果证明你和家人的血缘关系的就是户口本之类的了。
- authorities 只有用户身份得到认证之后，才能明确在这个身份下具有的权利。

Authentication 的生命周期

拿上一章中的表单登录来说可能更好理解一些，用户名 `username` 和密码 `password` 被封装成 `UsernamePasswordAuthenticationToken` 交给了 `AuthenticationManager` 去认证，认证成功后会返回一个 `UsernamePasswordAuthenticationToken.isAuthenticated() == true` 的 `UsernamePasswordAuthenticationToken`。然后 `UsernamePasswordAuthenticationToken` 可以被放入 `SecurityContext`，并通过 `SecurityContextHolder` 与当前线程相关联。当用户认证后再次去访问资源时，可以通过 `Session`（`SecurityContextPersistenceFilter`）或者 `Token` 等具体策略关联当前线程和 `Authentication`，当调用链完成后 `FilterChainProxy` 会从当前线程中清除 `Authentication`。伪代码如下：

```
1      // ①关联当前线程
2      SecurityContext context = SecurityContextHolder.createEmptyContext();
3      context.setAuthentication(anAuthentication);
4      SecurityContextHolder.setContext(context);
5
6      // ② 从当前线程获取当前Authentication信息
7      SecurityContext currentContext = SecurityContextHolder.getContext();
8
9      // ③从当前线程中清除
10     SecurityContextHolder.clearContext();
```

在实际场景中，你可以定义一个 `Controller` 的超类以方便在业务中注入当前用户信息，不过请注意认证用户可能是匿名用户。

```
1  public abstract class BaseController {
2      public BaseController() {
3      }
4
5      /**
6       * 当前用户认证信息
7       * @return currentUser
8       */
9      public Object currentUser() {
10         Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
```

```

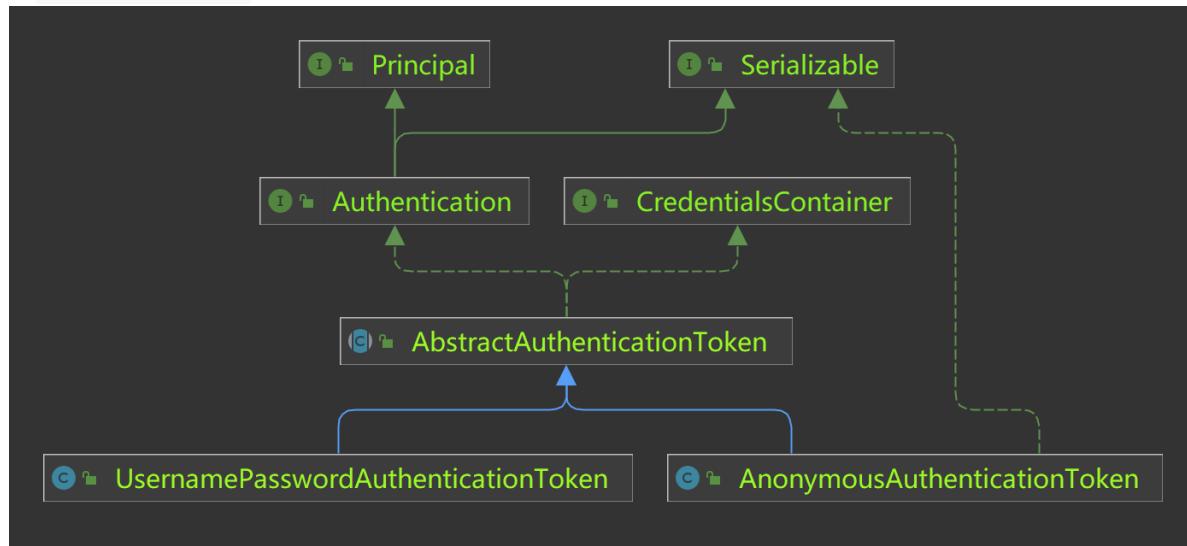
11         return authentication instanceof AnonymousAuthenticationToken ? new
12             User("-1", authentication.getName(), "[PROTECTED]", true, true, true,
13                 authentication.getAuthorities()) : authentication.getPrincipal();
14     }
15
16     /**
17      * 当前用户id
18      * @return userid
19      */
20     public String currentUserId() {
21         return ((User)this.currentUser()).getUserId();
22     }
23
24     /**
25      * 当前用户名
26      * @return username
27      */
28     public String currentUserName() {
29         return ((User)this.currentUser()).getUsername();
30     }

```

还有其它获取当前用户认证信息的方法你知道吗?

Authentication 的体系

以 `UsernamePasswordAuthenticationToken` 和 `AnonymousAuthenticationToken` 为参考, 你会发现 `Authentication` 的体系是这样的:



在不同的场景下的 `Authentication` 都可以通过实现 `AbstractAuthenticationToken` 来注入一些个性化的逻辑。现在手机验证码越来越流行了, 如果要集成手机验证码登录是不是也能照着上面的继承体系模仿一个呢? 来试一试。

```

1  /**
2  * 验证码认证凭据.
3  * @author felord.cn
4  */
5  public class CaptchaAuthenticationToken extends AbstractAuthenticationToken {
6
7      private static final long serialVersionUID =
8          SpringSecurityCoreVersion.SERIAL_VERSION_UID;
9
10 }

```

```
9     private final Object principal;
10    private String captcha;
11
12    /**
13     * 此构造函数用来初始化未授信凭据.
14     *
15     * @param principal the principal
16     * @param captcha the captcha
17     */
18    public CaptchaAuthenticationToken(Object principal, String captcha) {
19        super(null);
20        this.principal = principal;
21        this.captcha = captcha;
22        setAuthenticated(false);
23    }
24
25    /**
26     * 此构造函数用来初始化授信凭据.
27     *
28     * @param principal      the principal
29     * @param captcha       the captcha
30     * @param authorities the authorities
31     */
32    public CaptchaAuthenticationToken(Object principal, String captcha,
33                                     Collection<? extends GrantedAuthority>
34                                     authorities) {
35        super(authorities);
36        this.principal = principal;
37        this.captcha = captcha;
38        super.setAuthenticated(true); // must use super, as we override
39    }
40    @Override
41    public Object getCredentials() {
42        return this.captcha;
43    }
44    @Override
45    public Object getPrincipal() {
46        return this.principal;
47    }
48    @Override
49    public void setAuthenticated(boolean isAuthenticated) throws
50        IllegalArgumentException {
51        if (isAuthenticated) {
52            throw new IllegalArgumentException(
53                "Cannot set this token to trusted - use constructor which
54                takes a GrantedAuthority list instead");
55        }
56
57    }
58    @Override
59    public void eraseCredentials() {
60        super.eraseCredentials();
61        captcha = null;
62    }
}
```

上面这个实现也要对应一个 `CaptchaAuthenticationFilter`。实现这个可能还需要学习更多的相关知识，放在后面去解决。



15-自定义验证码认证

经过前几章的铺垫，Spring Security中认证的机制逐渐清晰了起来，今天就利用前面所学到的知识来开发一个自定义认证。我选择目前比较流行的验证码认证。期望可以通过以下HTTP调用成功认证：

```
1 POST /login/captcha HTTP/1.1
2 Host: localhost:8085
3 Content-Type: application/x-www-form-urlencoded
4
5 phone=11111111111&captcha=123456
```

认证流程的几个要素

从前面表单登录的研究中，可以分析出认证需要下面几个要素。

Authentication

`Authentication` 充当认证信息的载体，细分的话可分为 **认证前的信息载体** 和 **认证后的信息载体**。可参考 `UsernamePasswordAuthenticationToken` 进行实现，由于在第13章 **认证是什么** 中我们已经实现了验证码的认证Token `CaptchaAuthenticationToken`，这里就不再赘述了。

AuthenticationProvider

`Authentication` 需要一个 `AuthenticationProvider` 来进行认证的具体操作。这里也要实现一个专门认证 `CaptchaAuthenticationToken` 的 `Provider` `CaptchaAuthenticationProvider`。它的认证逻辑包含两个步骤。

验证码校验

从 `CaptchaAuthenticationToken` 中提取传入手机号和收到的验证码，和缓存中的验证码进行比对，根据比对的布尔值结果来决定是否进行下一步认证操作。这里抽象了这个流程：

```

1  public interface CaptchaService {
2
3      /**
4      * verify captcha
5      *
6      * @param phone 手机号 用来获取缓存验证码
7      * @param rawCode 收到的验证码 用来和缓存验证码作比对
8      * @return isVerified 比对的结果
9      */
10     boolean verifyCaptcha(String phone, String rawCode);
11 }

```

UserDetails加载

`UserDetailsService` 是通过用户名去加载 `UserDetails`，而这里我们需要一个通过手机号去加载。本来打算复用 `UserDetailsService`，因为手机号作为用户名也说得过去，但是 Spring IoC 中不允许存在多个 `UserDetailsService`，复用的话就不能同时为 Spring Bean 了。这里重新定义一个：

```

1  /**
2  * @author felord
3  */
4  public interface CaptchaUserDetailsService {
5      /**
6      * load user by phone
7      *
8      * @param phone phone
9      * @return userDetails
10     * @throws UsernameNotFoundException not found user
11    */
12     UserDetails loadUserByPhone(String phone) throws UsernameNotFoundException;
13 }

```

CaptchaAuthenticationProvider

接下来实现验证码认证的流程。首先 `supports` 方法定义为专门支持 `CaptchaAuthenticationToken` 的认证；然后无非就是从 `CaptchaAuthenticationToken` 对象中提取手机号和验证码并把它们依次交给 `CaptchaService` 和 `CaptchaUserDetailsService` 进行处理。认证成功后把 `UserDetails` 封装为认证过的 `CaptchaAuthenticationToken` 返回。代码如下：

```

1  /**
2  * 验证码认证器
3  *
4  * @author n1
5  */
6  public class CaptchaAuthenticationProvider implements AuthenticationProvider,
7  InitializingBean, MessageSourceAware {
8
9      private final GrantedAuthoritiesMapper authoritiesMapper = new
10     NullAuthoritiesMapper();
11     private final CaptchaUserDetailsService captchaUserDetailsService;
12     private final CaptchaService captchaService;
13     private MessageSourceAccessor messages =
14     SpringSecurityMessageSource.getAccessor();
15
16     public CaptchaAuthenticationProvider(CaptchaUserDetailsService
17     captchaUserDetailsService, CaptchaService captchaService) {
18         this.captchaUserDetailsService = captchaUserDetailsService;
19     }
20
21     @Override
22     public Authentication authenticate(Authentication authentication)
23             throws AuthenticationException {
24
25         if (authentication instanceof
26             CaptchaAuthenticationToken) {
27
28             CaptchaAuthenticationToken token =
29                 (CaptchaAuthenticationToken) authentication;
30
31             String phone = token.getPhone();
32             String code = token.getCode();
33
34             boolean verified = captchaService.verifyCaptcha(phone, code);
35
36             if (!verified) {
37                 throw new UsernameNotFoundException(
38                     "验证码错误");
39             }
40
41             UserDetails userDetails =
42                 captchaUserDetailsService.loadUserByPhone(phone);
43
44             CaptchaAuthenticationToken result =
45                 new CaptchaAuthenticationToken(userDetails, token);
46
47             result.setAuthenticated(true);
48
49             return result;
50         }
51
52         throw new AuthenticationServiceException("不支持的凭证类型");
53     }
54
55     @Override
56     public void destroy() {
57
58     }
59
60     @Override
61     public void afterPropertiesSet() {
62
63     }
64
65     @Override
66     public void init() {
67
68     }
69
70     @Override
71     public void destroy() {
72
73     }
74
75     @Override
76     public void afterPropertiesSet() {
77
78     }
79
80     @Override
81     public void init() {
82
83     }
84
85     @Override
86     public void destroy() {
87
88     }
89
90     @Override
91     public void afterPropertiesSet() {
92
93     }
94
95     @Override
96     public void init() {
97
98     }
99
100    @Override
101    public void destroy() {
102
103    }
104
105    @Override
106    public void afterPropertiesSet() {
107
108    }
109
110    @Override
111    public void init() {
112
113    }
114
115    @Override
116    public void destroy() {
117
118    }
119
120    @Override
121    public void afterPropertiesSet() {
122
123    }
124
125    @Override
126    public void init() {
127
128    }
129
130    @Override
131    public void destroy() {
132
133    }
134
135    @Override
136    public void afterPropertiesSet() {
137
138    }
139
140    @Override
141    public void init() {
142
143    }
144
145    @Override
146    public void destroy() {
147
148    }
149
150    @Override
151    public void afterPropertiesSet() {
152
153    }
154
155    @Override
156    public void init() {
157
158    }
159
160    @Override
161    public void destroy() {
162
163    }
164
165    @Override
166    public void afterPropertiesSet() {
167
168    }
169
170    @Override
171    public void init() {
172
173    }
174
175    @Override
176    public void destroy() {
177
178    }
179
180    @Override
181    public void afterPropertiesSet() {
182
183    }
184
185    @Override
186    public void init() {
187
188    }
189
190    @Override
191    public void destroy() {
192
193    }
194
195    @Override
196    public void afterPropertiesSet() {
197
198    }
199
200    @Override
201    public void init() {
202
203    }
204
205    @Override
206    public void destroy() {
207
208    }
209
210    @Override
211    public void afterPropertiesSet() {
212
213    }
214
215    @Override
216    public void init() {
217
218    }
219
220    @Override
221    public void destroy() {
222
223    }
224
225    @Override
226    public void afterPropertiesSet() {
227
228    }
229
230    @Override
231    public void init() {
232
233    }
234
235    @Override
236    public void destroy() {
237
238    }
239
240    @Override
241    public void afterPropertiesSet() {
242
243    }
244
245    @Override
246    public void init() {
247
248    }
249
250    @Override
251    public void destroy() {
252
253    }
254
255    @Override
256    public void afterPropertiesSet() {
257
258    }
259
260    @Override
261    public void init() {
262
263    }
264
265    @Override
266    public void destroy() {
267
268    }
269
270    @Override
271    public void afterPropertiesSet() {
272
273    }
274
275    @Override
276    public void init() {
277
278    }
279
280    @Override
281    public void destroy() {
282
283    }
284
285    @Override
286    public void afterPropertiesSet() {
287
288    }
289
290    @Override
291    public void init() {
292
293    }
294
295    @Override
296    public void destroy() {
297
298    }
299
300    @Override
301    public void afterPropertiesSet() {
302
303    }
304
305    @Override
306    public void init() {
307
308    }
309
310    @Override
311    public void destroy() {
312
313    }
314
315    @Override
316    public void afterPropertiesSet() {
317
318    }
319
320    @Override
321    public void init() {
322
323    }
324
325    @Override
326    public void destroy() {
327
328    }
329
330    @Override
331    public void afterPropertiesSet() {
332
333    }
334
335    @Override
336    public void init() {
337
338    }
339
340    @Override
341    public void destroy() {
342
343    }
344
345    @Override
346    public void afterPropertiesSet() {
347
348    }
349
350    @Override
351    public void init() {
352
353    }
354
355    @Override
356    public void destroy() {
357
358    }
359
360    @Override
361    public void afterPropertiesSet() {
362
363    }
364
365    @Override
366    public void init() {
367
368    }
369
370    @Override
371    public void destroy() {
372
373    }
374
375    @Override
376    public void afterPropertiesSet() {
377
378    }
379
380    @Override
381    public void init() {
382
383    }
384
385    @Override
386    public void destroy() {
387
388    }
389
390    @Override
391    public void afterPropertiesSet() {
392
393    }
394
395    @Override
396    public void init() {
397
398    }
399
400    @Override
401    public void destroy() {
402
403    }
404
405    @Override
406    public void afterPropertiesSet() {
407
408    }
409
410    @Override
411    public void init() {
412
413    }
414
415    @Override
416    public void destroy() {
417
418    }
419
420    @Override
421    public void afterPropertiesSet() {
422
423    }
424
425    @Override
426    public void init() {
427
428    }
429
430    @Override
431    public void destroy() {
432
433    }
434
435    @Override
436    public void afterPropertiesSet() {
437
438    }
439
440    @Override
441    public void init() {
442
443    }
444
445    @Override
446    public void destroy() {
447
448    }
449
450    @Override
451    public void afterPropertiesSet() {
452
453    }
454
455    @Override
456    public void init() {
457
458    }
459
460    @Override
461    public void destroy() {
462
463    }
464
465    @Override
466    public void afterPropertiesSet() {
467
468    }
469
470    @Override
471    public void init() {
472
473    }
474
475    @Override
476    public void destroy() {
477
478    }
479
480    @Override
481    public void afterPropertiesSet() {
482
483    }
484
485    @Override
486    public void init() {
487
488    }
489
490    @Override
491    public void destroy() {
492
493    }
494
495    @Override
496    public void afterPropertiesSet() {
497
498    }
499
500    @Override
501    public void init() {
502
503    }
504
505    @Override
506    public void destroy() {
507
508    }
509
510    @Override
511    public void afterPropertiesSet() {
512
513    }
514
515    @Override
516    public void init() {
517
518    }
519
520    @Override
521    public void destroy() {
522
523    }
524
525    @Override
526    public void afterPropertiesSet() {
527
528    }
529
530    @Override
531    public void init() {
532
533    }
534
535    @Override
536    public void destroy() {
537
538    }
539
540    @Override
541    public void afterPropertiesSet() {
542
543    }
544
545    @Override
546    public void init() {
547
548    }
549
550    @Override
551    public void destroy() {
552
553    }
554
555    @Override
556    public void afterPropertiesSet() {
557
558    }
559
560    @Override
561    public void init() {
562
563    }
564
565    @Override
566    public void destroy() {
567
568    }
569
570    @Override
571    public void afterPropertiesSet() {
572
573    }
574
575    @Override
576    public void init() {
577
578    }
579
580    @Override
581    public void destroy() {
582
583    }
584
585    @Override
586    public void afterPropertiesSet() {
587
588    }
589
590    @Override
591    public void init() {
592
593    }
594
595    @Override
596    public void destroy() {
597
598    }
599
600    @Override
601    public void afterPropertiesSet() {
602
603    }
604
605    @Override
606    public void init() {
607
608    }
609
610    @Override
611    public void destroy() {
612
613    }
614
615    @Override
616    public void afterPropertiesSet() {
617
618    }
619
620    @Override
621    public void init() {
622
623    }
624
625    @Override
626    public void destroy() {
627
628    }
629
630    @Override
631    public void afterPropertiesSet() {
632
633    }
634
635    @Override
636    public void init() {
637
638    }
639
640    @Override
641    public void destroy() {
642
643    }
644
645    @Override
646    public void afterPropertiesSet() {
647
648    }
649
650    @Override
651    public void init() {
652
653    }
654
655    @Override
656    public void destroy() {
657
658    }
659
660    @Override
661    public void afterPropertiesSet() {
662
663    }
664
665    @Override
666    public void init() {
667
668    }
669
670    @Override
671    public void destroy() {
672
673    }
674
675    @Override
676    public void afterPropertiesSet() {
677
678    }
679
680    @Override
681    public void init() {
682
683    }
684
685    @Override
686    public void destroy() {
687
688    }
689
690    @Override
691    public void afterPropertiesSet() {
692
693    }
694
695    @Override
696    public void init() {
697
698    }
699
700    @Override
701    public void destroy() {
702
703    }
704
705    @Override
706    public void afterPropertiesSet() {
707
708    }
709
710    @Override
711    public void init() {
712
713    }
714
715    @Override
716    public void destroy() {
717
718    }
719
720    @Override
721    public void afterPropertiesSet() {
722
723    }
724
725    @Override
726    public void init() {
727
728    }
729
730    @Override
731    public void destroy() {
732
733    }
734
735    @Override
736    public void afterPropertiesSet() {
737
738    }
739
740    @Override
741    public void init() {
742
743    }
744
745    @Override
746    public void destroy() {
747
748    }
749
750    @Override
751    public void afterPropertiesSet() {
752
753    }
754
755    @Override
756    public void init() {
757
758    }
759
760    @Override
761    public void destroy() {
762
763    }
764
765    @Override
766    public void afterPropertiesSet() {
767
768    }
769
770    @Override
771    public void init() {
772
773    }
774
775    @Override
776    public void destroy() {
777
778    }
779
780    @Override
781    public void afterPropertiesSet() {
782
783    }
784
785    @Override
786    public void init() {
787
788    }
789
790    @Override
791    public void destroy() {
792
793    }
794
795    @Override
796    public void afterPropertiesSet() {
797
798    }
799
800    @Override
801    public void init() {
802
803    }
804
805    @Override
806    public void destroy() {
807
808    }
809
810    @Override
811    public void afterPropertiesSet() {
812
813    }
814
815    @Override
816    public void init() {
817
818    }
819
820    @Override
821    public void destroy() {
822
823    }
824
825    @Override
826    public void afterPropertiesSet() {
827
828    }
829
830    @Override
831    public void init() {
832
833    }
834
835    @Override
836    public void destroy() {
837
838    }
839
840    @Override
841    public void afterPropertiesSet() {
842
843    }
844
845    @Override
846    public void init() {
847
848    }
849
850    @Override
851    public void destroy() {
852
853    }
854
855    @Override
856    public void afterPropertiesSet() {
857
858    }
859
860    @Override
861    public void init() {
862
863    }
864
865    @Override
866    public void destroy() {
867
868    }
869
870    @Override
871    public void afterPropertiesSet() {
872
873    }
874
875    @Override
876    public void init() {
877
878    }
879
880    @Override
881    public void destroy() {
882
883    }
884
885    @Override
886    public void afterPropertiesSet() {
887
888    }
889
890    @Override
891    public void init() {
892
893    }
894
895    @Override
896    public void destroy() {
897
898    }
899
900    @Override
901    public void afterPropertiesSet() {
902
903    }
904
905    @Override
906    public void init() {
907
908    }
909
910    @Override
911    public void destroy() {
912
913    }
914
915    @Override
916    public void afterPropertiesSet() {
917
918    }
919
920    @Override
921    public void init() {
922
923    }
924
925    @Override
926    public void destroy() {
927
928    }
929
930    @Override
931    public void afterPropertiesSet() {
932
933    }
934
935    @Override
936    public void init() {
937
938    }
939
940    @Override
941    public void destroy() {
942
943    }
944
945    @Override
946    public void afterPropertiesSet() {
947
948    }
949
950    @Override
951    public void init() {
952
953    }
954
955    @Override
956    public void destroy() {
957
958    }
959
960    @Override
961    public void afterPropertiesSet() {
962
963    }
964
965    @Override
966    public void init() {
967
968    }
969
970    @Override
971    public void destroy() {
972
973    }
974
975    @Override
976    public void afterPropertiesSet() {
977
978    }
979
980    @Override
981    public void init() {
982
983    }
984
985    @Override
986    public void destroy() {
987
988    }
989
990    @Override
991    public void afterPropertiesSet() {
992
993    }
994
995    @Override
996    public void init() {
997
998    }
999
1000   @Override
1001  public void destroy() {
1002
1003  }
1004
1005  @Override
1006  public void afterPropertiesSet() {
1007
1008  }
1009
1010  @Override
1011  public void init() {
1012
1013  }
1014
1015  @Override
1016  public void destroy() {
1017
1018  }
1019
1020  @Override
1021  public void afterPropertiesSet() {
1022
1023  }
1024
1025  @Override
1026  public void init() {
1027
1028  }
1029
1030  @Override
1031  public void destroy() {
1032
1033  }
1034
1035  @Override
1036  public void afterPropertiesSet() {
1037
1038  }
1039
1040  @Override
1041  public void init() {
1042
1043  }
1044
1045  @Override
1046  public void destroy() {
1047
1048  }
1049
1050  @Override
1051  public void afterPropertiesSet() {
1052
1053  }
1054
1055  @Override
1056  public void init() {
1057
1058  }
1059
1060  @Override
1061  public void destroy() {
1062
1063  }
1064
1065  @Override
1066  public void afterPropertiesSet() {
1067
1068  }
1069
1070  @Override
1071  public void init() {
1072
1073  }
1074
1075  @Override
1076  public void destroy() {
1077
1078  }
1079
1080  @Override
1081  public void afterPropertiesSet() {
1082
1083  }
1084
1085  @Override
1086  public void init() {
1087
1088  }
1089
1090  @Override
1091  public void destroy() {
1092
1093  }
1094
1095  @Override
1096  public void afterPropertiesSet() {
1097
1098  }
1099
1100  @Override
1101  public void init() {
1102
1103  }
1104
1105  @Override
1106  public void destroy() {
1107
1108  }
1109
1110  @Override
1111  public void afterPropertiesSet() {
1112
1113  }
1114
1115  @Override
1116  public void init() {
1117
1118  }
1119
1120  @Override
1121  public void destroy() {
1122
1123  }
1124
1125  @Override
1126  public void afterPropertiesSet() {
1127
1128  }
1129
1130  @Override
1131  public void init() {
1132
1133  }
1134
1135  @Override
1136  public void destroy() {
1137
1138  }
1139
1140  @Override
1141  public void afterPropertiesSet() {
1142
1143  }
1144
1145  @Override
1146  public void init() {
1147
1148  }
1149
1150  @Override
1151  public void destroy() {
1152
1153  }
1154
1155  @Override
1156  public void afterPropertiesSet() {
1157
1158  }
1159
1160  @Override
1161  public void init() {
1162
1163  }
1164
1165  @Override
1166  public void destroy() {
1167
1168  }
1169
1170  @Override
1171  public void afterPropertiesSet() {
1172
1173  }
1174
1175  @Override
1176  public void init() {
1177
1178  }
1179
1180  @Override
1181  public void destroy() {
1182
1183  }
1184
1185  @Override
1186  public void afterPropertiesSet() {
1187
1188  }
1189
1190  @Override
1191  public void init() {
1192
1193  }
1194
1195  @Override
1196  public void destroy() {
1197
1198  }
1199
1200  @Override
1201  public void afterPropertiesSet() {
1202
1203  }
1204
1205  @Override
1206  public void init() {
1207
1208  }
1209
1210  @Override
1211  public void destroy() {
1212
1213  }
1214
1215  @Override
1216  public void afterPropertiesSet() {
1217
1218  }
1219
1220  @Override
1221  public void init() {
1222
1223  }
1224
1225  @Override
1226  public void destroy() {
1227
1228  }
1229
1230  @Override
1231  public void afterPropertiesSet() {
1232
1233  }
1234
1235  @Override
1236  public void init() {
1237
1238  }
1239
1240  @Override
1241  public void destroy() {
1242
1243  }
1244
1245  @Override
1246  public void afterPropertiesSet() {
1247
1248  }
1249
1250  @Override
1251  public void init() {
1252
1253  }
1254
1255  @Override
1256  public void destroy() {
1257
1258  }
1259
1260  @Override
1261  public void afterPropertiesSet() {
1262
1263  }
1264
1265  @Override
1266  public void init() {
1267
1268  }
1269
1270  @Override
1271  public void destroy() {
1272
1273  }
1274
1275  @Override
1276  public void afterPropertiesSet() {
1277
1278  }
1279
1280  @Override
1281  public void init() {
1282
1283  }
1284
1285  @Override
1286  public void destroy() {
1287
1288  }
1289
1290  @Override
1291  public void afterPropertiesSet() {
1292
1293  }
1294
1295  @Override
1296  public void init() {
1297
1298  }
1299
1300  @Override
1301  public void destroy() {
1302
1303  }
1304
1305  @Override
1306  public void afterPropertiesSet() {
1307
1308  }
1309
1310  @Override
1311  public void init() {
1312
1313  }
1314
1315  @Override
1316  public void destroy() {
1317
1318  }
1319
1320  @Override
1321  public void afterPropertiesSet() {
1322
1323  }
1324
1325  @Override
1326  public void init() {
1327
1328  }
1329
1330  @Override
1331  public void destroy() {
1332
1333  }
1334
1335  @Override
1336  public void afterPropertiesSet() {
1337
1338  }
1339
1340  @Override
1341  public void init() {
1342
1343  }
1344
1345  @Override
1346  public void destroy() {
1347
1348  }
1349
1350  @Override
1351  public void afterPropertiesSet() {
1352
1353  }
1354
1355  @Override
1356  public void init() {
1357
1358  }
1359
1360  @Override
1361  public void destroy() {
1362
1363  }
1364
1365  @Override
1366  public void afterPropertiesSet() {
1367
1368  }
1369
1370  @Override
1371  public void init() {
1372
1373  }
1374
1375  @Override
1376  public void destroy() {
1377
1378  }
1379
1380  @Override
1381  public void afterPropertiesSet() {
1382
1383  }
1384
1385  @Override
1386  public void init() {
1387
1388  }
1389
1390  @Override
1391  public void destroy() {
1392
1393  }
1394
1395  @Override
1396  public void afterPropertiesSet() {
1397
1398  }
1399
1400  @Override
1401  public void init() {
1402
1403  }
1404
1405  @Override
1406  public void destroy() {
1407
1408  }
1409
1410  @Override
1411  public void afterPropertiesSet() {
1412
1413  }
1414
1415  @Override
1416  public void init() {
1417
1418  }
1419
1420  @Override
1421  public void destroy() {
1422
1423  }
1424
1425  @Override
1426  public void afterPropertiesSet() {
1427
1428  }
1429
1430  @Override
1431  public void init() {
1432
1433  }
1434
1435  @Override
1436  public void destroy() {
1437
1438  }
1439
1440  @Override
1441  public void afterPropertiesSet() {
1442
1443  }
1444
1445  @Override
1446  public void init() {
1447
1448  }
1449
1450  @Override
1451  public void destroy() {
1452
1453  }
1454
1455  @Override
1456  public void afterPropertiesSet() {
1457
1458  }
1459
1460  @Override
1461  public void init() {
1462
1463  }
1464
1465  @Override
1466  public void destroy() {
1467
1468  }
1469
1470  @Override
1471  public void afterPropertiesSet() {
1472
1473  }
1474
1475  @Override
1476  public void init() {
1477
1478  }
1479
1480  @Override
1481  public void destroy() {
1482
1483  }
1484
1485  @Override
1486  public void afterPropertiesSet() {
1487
1488  }
1489
1490  @Override
1491  public void init() {
1492
1493  }
1494
1495  @Override
1496  public void destroy() {
1497
1498  }
1499
1500  @Override
1501  public void afterPropertiesSet() {
1502
1503  }
1504
1505  @Override
1506  public void init() {
1507
1508  }
1509
1510  @Override
1511  public void destroy() {
1512
1513  }
1514
1515  @Override
1516  public void afterPropertiesSet() {
1517
1518  }
1519
1520  @Override
1521  public void init() {
1522
1523  }
1524
1525  @Override
1526  public void destroy() {
1527
1528  }
1529
1530  @Override
1531  public void afterPropertiesSet() {
1532
1533  }
1534
1535  @Override
1536  public void init() {
1537
1538  }
1539
1540  @Override
1541  public void destroy() {
1542
1543  }
1544
1545  @Override
1546  public void afterPropertiesSet() {
1547
1548  }
1549
1550  @Override
1551  public void init() {
1552
1553  }
1554
1555  @Override
1556  public void destroy() {
1557
1558  }
1559
1560  @Override
1561  public void afterPropertiesSet() {
1562
1563  }
1564
1565  @Override
1566  public void init() {
1567
1568  }
1569
1570  @Override
1571  public void destroy() {
1572
1573  }
1574
1575  @Override
1576  public void afterPropertiesSet() {
1577
1578  }
1579
1580  @Override
1581  public void init() {
1582
1583  }
1584
1585  @Override
1586  public void destroy() {
1587
1588  }
1589
1590  @Override
1591  public void afterPropertiesSet() {
1592
1593  }
1594
1595  @Override
1596  public void init() {
1597
1598  }
1599
1600  @Override
1601  public void destroy() {
1602
1603  }
1604
1605  @Override
1606  public void afterPropertiesSet() {
1607
1608  }
1609
1610  @Override
1611  public void init() {
1612
1613  }
1614
1615  @Override
1616  public void destroy() {
1617
1618  }
1619
1620  @Override
1621  public void afterPropertiesSet() {
1622
1623  }
1624
1625  @Override
1626  public void init() {
1627
1628  }
1629
1630  @Override
1631  public void destroy() {
1632
1633  }
1634
1635  @Override
1636  public void afterPropertiesSet() {
1637
1638  }
1639
1640  @Override
1641  public void init() {
1642
1643  }
1644
1645  @Override
1646  public void destroy() {
1647
1648  }
1649
1650  @Override
1651  public void afterPropertiesSet() {
1652
1653  }
1654
1655  @Override
1656  public void init() {
1657
1658  }
1659
1660  @Override
1661  public void destroy() {
1662
1663  }
1664
1665
```

```
15         this.captchaService = captchaService;
16     }
17
18     @Override
19     public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
20         Assert.isInstanceOf(CaptchaAuthenticationToken.class, authentication,
21             () -> messages.getMessage(
22                 "CaptchaAuthenticationProvider.onlySupports",
23                 "Only CaptchaAuthenticationToken is supported"));
24
25         CaptchaAuthenticationToken unAuthenticationToken =
26         (CaptchaAuthenticationToken) authentication;
27
28         String phone = unAuthenticationToken.getName();
29         String rawCode = (String) unAuthenticationToken.getCredentials();
30         // 验证码校验
31         if (captchaService.verifyCaptcha(phone, rawCode)) {
32             UserDetails userDetailsService =
33             captchaUserDetailsService.loadUserByPhone(phone);
34             //TODO 此处省略对UserDetails 的可用性 是否过期 是否锁定 是否失效的检验 建议
根据实际情况添加 或者在 UserDetailsService 的实现中处理
35             return createSuccessAuthentication(authentication, userDetailsService);
36         } else {
37             throw new BadCredentialsException("captcha is not matched");
38         }
39
40     @Override
41     public boolean supports(Class<?> authentication) {
42         return CaptchaAuthenticationToken.class.isAssignableFrom(authentication);
43     }
44
45     @Override
46     public void afterPropertiesSet() throws Exception {
47         Assert.notNull(captchaUserDetailsService, "captchaUserDetailsService must
not be null");
48         Assert.notNull(captchaService, "captchaService must not be null");
49     }
50
51     @Override
52     public void setMessageSource(MessageSource messageSource) {
53         this.messages = new MessageSourceAccessor(messageSource);
54     }
55
56     /**
57      * 认证成功将非授信凭据转为授信凭据 .
58      * 封装用户信息 角色信息。
59      *
60      * @param authentication the authentication
61      * @param user           the user
62      * @return the authentication
63      */
64     protected Authentication createSuccessAuthentication(Authentication
65     authentication, UserDetails user) {
66
67         Collection<? extends GrantedAuthority> authorities =
68         authoritiesMapper.mapAuthorities(user.getAuthorities());
69     }
70 }
```

```
66     CaptchaAuthenticationToken authenticationToken = new
67     CaptchaAuthenticationToken(user, null, authorities);
68     authenticationToken.setDetails(authentication.getDetails());
69     return authenticationToken;
70 }
71
72 }
```

注意：认证成功的Token封装一定要使用内置执行 `super.setAuthenticated(true)` 的构造函数。

放入AuthenticationManager

所有 `AuthenticationProvider` 都要被注入 `AuthenticationManager`，
`CaptchaAuthenticationProvider` 也不例外。我们需要找一个合适的切入点把它放进去。[第12章](#) 提到了一个方法可以实现这个功能，你可以回看一下；更加合理的方法是[第9章](#) 提到的 `HttpSecurity.authenticationProvider` 方法，稍后我会讲如何做。

CaptchaAuthenticationFilter

从 `UsernamePasswordAuthenticationFilter` 的源码中可以看出，认证过滤器的作用很纯粹。就是拦截到特定的认证URL，提取认证参数，封装为 `Authentication`，最后交给 `AuthenticationManager` 认证处理。这里模仿了 `UsernamePasswordAuthenticationFilter`：

```
1  public class CaptchaAuthenticationFilter extends
2      AbstractAuthenticationProcessingFilter {
3
4      public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "phone";
5
6      public static final String SPRING_SECURITY_FORM_CAPTCHA_KEY = "captcha";
7
8      private static final AntPathRequestMatcher DEFAULT_ANT_PATH_REQUEST_MATCHER =
9          new AntPathRequestMatcher("/login/captcha",
10             "POST");
11
12      private String usernameParameter = SPRING_SECURITY_FORM_USERNAME_KEY;
13
14      private String captchaParameter = SPRING_SECURITY_FORM_CAPTCHA_KEY;
15
16      private Converter<HttpServletRequest, CaptchaAuthenticationToken>
17          captchaAuthenticationTokenConverter;
18
19      private boolean postOnly = true;
20
21
22      public CaptchaAuthenticationFilter() {
23          super(DEFAULT_ANT_PATH_REQUEST_MATCHER);
24          this.captchaAuthenticationTokenConverter = defaultConverter();
25      }
26
27      public CaptchaAuthenticationFilter(AuthenticationManager
28          authenticationManager) {
29          super(DEFAULT_ANT_PATH_REQUEST_MATCHER, authenticationManager);
30          this.captchaAuthenticationTokenConverter = defaultConverter();
31      }
32
33      @Override
```

```
29     public Authentication attemptAuthentication(HttpServletRequest request,
30             HttpServletResponse response) throws AuthenticationException {
31         // 拦截认证URL已经由父类AbstractAuthenticationProcessingFilter实现
32         // ①是否只支持Post请求
33         if (this.postOnly && !HttpMethod.POST.matches(request.getMethod())) {
34             throw new AuthenticationServiceException("Authentication method not
35                     supported: " + request.getMethod());
36         }
37         // ②提取认证信息封装为CaptchaAuthenticationToken
38         CaptchaAuthenticationToken authRequest =
39             captchaAuthenticationTokenConverter.convert(request);
40         // Allow subclasses to set the "details" property
41         setDetails(request, authRequest);
42         // ③交给AuthenticationManager认证处理
43         return this.getAuthenticationManager().authenticate(authRequest);
44     }
45
46     // 从request中提取参数封装为Token
47     private Converter<HttpServletRequest, CaptchaAuthenticationToken>
48     defaultConverter() {
49         return request -> {
50             String username = request.getParameter(this.usernameParameter);
51             username = (username != null) ? username : "";
52             String captcha = request.getParameter(this.captchaParameter);
53             captcha = (captcha != null) ? captcha : "";
54             return new CaptchaAuthenticationToken(username, captcha);
55         };
56     }
57
58
59     protected void setDetails(HttpServletRequest request,
60             CaptchaAuthenticationToken authRequest) {
61
62         authRequest.setDetails(this.authenticationDetailsSource.buildDetails(request));
63     }
64
65     public void setUsernameParameter(String usernameParameter) {
66         Assert.hasText(usernameParameter, "Username parameter must not be empty
67             or null");
68         this.usernameParameter = usernameParameter;
69     }
70
71     public void setCaptchaParameter(String captchaParameter) {
72         Assert.hasText(captchaParameter, "Password parameter must not be empty or
73             null");
74         this.captchaParameter = captchaParameter;
75     }
76
77     public void setConverter(Converter<HttpServletRequest,
78             CaptchaAuthenticationToken> converter) {
79         Assert.notNull(converter, "Converter must not be null");
80         this.captchaAuthenticationTokenConverter = converter;
81     }
82
83     public void setPostOnly(boolean postOnly) {
84         this.postOnly = postOnly;
85     }
86
87     public void setAuthenticationDetailsSource(AuthenticationDetailsSource<HttpServletRequest> authenticationDetailsSource) {
88         this.authenticationDetailsSource = authenticationDetailsSource;
89     }
90 }
```

```
78     public final String getUsernameParameter() {
79         return this.usernameParameter;
80     }
81
82     public final String getCaptchaParameter() {
83         return this.captchaParameter;
84     }
85 }
```

上面的代码中默认情况下会去拦截 `/login/captcha` 请求并提取手机号 `phone` 和验证码 `captcha`（当然这些默认设定都是可自定义配置的），然后封装成 `CaptchaAuthenticationToken` 交给 `AuthenticationManager` 认证处理。另外，细心的同学会发现 `CaptchaAuthenticationFilter` 有一个特别成员变量：

```
1     private Converter<HttpServletRequest, CaptchaAuthenticationToken>
2         captchaAuthenticationTokenConverter;
```

这里是我增加了一个扩展点。`UsernamePasswordAuthenticationFilter` 的请求参数是Form提交，现在很多时候需要JSON提交。为了适应这种情况就引入了一个转换器函数接口，传入一个 `HttpServletRequest` 返回 `CaptchaAuthenticationToken`，这有点类似Java内置的 `Function<T, R>`，你可以根据需要实现这个函数来解析请求参数，这里我提供了一个和`一致的默认实现：

```
1     private Converter<HttpServletRequest, CaptchaAuthenticationToken>
2         defaultConverter() {
3             return request -> {
4                 String username = request.getParameter(this.usernameParameter);
5                 username = (username != null) ? username.trim() : "";
6                 String captcha = request.getParameter(this.captchaParameter);
7                 captcha = (captcha != null) ? captcha.trim() : "";
8                 return new CaptchaAuthenticationToken(username, captcha);
9             };
10        }
```

配置

到这里其实功能已经实现了，我们用最简单的办法来配置验证码登录认证功能：

```
1     httpSecurity.authenticationProvider(new CaptchaAuthenticationProvider(phone ->
2         userDetailsService.loadUserByUsername("felord"),(phone, rawCode) -> true))
3         .addFilterBefore(new
4             CaptchaAuthenticationFilter(http.getSharedObject(AuthenticationManager.class)),
5             UsernamePasswordAuthenticationFilter.class);
```

就连我自己认为这样就可以了，但是事实上运行中会发现 `http.getSharedObject(AuthenticationManager.class)==null`，导致认证出错。最直观的解决办法是我们自己搞一个包含 `CaptchaAuthenticationProvider` 的 `AuthenticationManager` 放到 `CaptchaAuthenticationFilter` 就可以了。

```
1     ProviderManager providerManager = new
2         ProviderManager(Collections.singletonList(captchaAuthenticationProvider));
3         captchaAuthenticationFilter.setAuthenticationManager(providerManager);
```

上面的方案确实可行，但是却没有充分利用Spring Security提供的机制，我会在下一篇进行讲解更好的办法。



微信搜一搜

码农小胖哥

16-配置的高级玩法

在上一章我带大家实现了验证码认证，今天我们来改良一下验证码认证的配置方式。

AbstractAuthenticationFilterConfigurer

`CaptchaAuthenticationFilter` 是通过模仿 `UsernamePasswordAuthenticationFilter` 实现的。同样的道理，由于 `UsernamePasswordAuthenticationFilter` 的配置是由 `FormLoginConfigurer` 来完成的，应该也能模仿一下 `FormLoginConfigurer`，实现一个配置类去配置 `CaptchaAuthenticationFilter`。

```
1 public final class FormLoginConfigurer<H extends HttpSecurityBuilder<H>> extends
2     AbstractAuthenticationFilterConfigurer<H, FormLoginConfigurer<H>,
3     UsernamePasswordAuthenticationFilter> {
4
5     // 省略
6 }
```

`FormLoginConfigurer` 看起来有点复杂，不过继承关系并不复杂，只继承了 `AbstractAuthenticationFilterConfigurer`。

```
1 public abstract class AbstractAuthenticationFilterConfigurer<B extends
2     HttpSecurityBuilder<B>, T extends AbstractAuthenticationFilterConfigurer<B, T, F>,
3     F extends AbstractAuthenticationProcessingFilter>
4     extends AbstractHttpConfigurer<T, B> {
```

理论上我们模仿一下，也继承一下这个类，但是你会发现这种方式行不通。因为它的 `javadoc` 是这样描述的：

Base class for configuring {@link AbstractAuthenticationFilterConfigurer}. This is intended for internal use only.

大意就是 `AbstractAuthenticationFilterConfigurer` 只能 `Spring Security` 内部使用，不建议用来做自定义。根本原因在于它最终向 `HttpSecurity` 添加过滤器使用的是 `HttpSecurity.addFilter(Filter)` 方法，这个方法只有内置过滤器（参见 `FilterOrderRegistration`）才能使用。了解了这个机制之后，我们对 `AbstractAuthenticationFilterConfigurer` 进行改造就能满足需要了。

改造

`AbstractAuthenticationFilterConfigurer<B, T, F>` 中的 `B` 是实际指的 `HttpSecurity`，因此这个要保留；

`T` 指的是它本身的实现，我们配置 `CaptchaAuthenticationFilter` 不需要下沉一层到 `FormLoginConfigurer` 这个继承级别，直接在 `AbstractAuthenticationFilterConfigurer` 这个继承级别实现即可，因此 `T` 这里指的就是需要配置类本身，也不需要再抽象化，因此是不需要的；同样的原因 `F` 也不需要，很明确是 `CaptchaAuthenticationFilter`，不需要再泛化。这样 `CaptchaAuthenticationFilter` 的配置类结构可以这样定义：

```
1  public class CaptchaAuthenticationFilterConfigurer<H extends
2      HttpSecurityBuilder<H>> extends
3      AbstractHttpConfigurer<CaptchaAuthenticationFilterConfigurer<H>, H> {
4          // 不再泛化 具体化
5          private final CaptchaAuthenticationFilter authFilter;
6          // 特定的验证码用户服务
7          private CaptchaUserDetailsService captchaUserDetailsService;
8          // 验证码处理服务
9          private CaptchaService captchaService;
10         // 保存认证请求细节的策略
11         private AuthenticationDetailsSource<HttpServletRequest, ?>
12             authenticationDetailsSource;
13         // 默认使用保存请求认证成功处理器
14         private SavedRequestAwareAuthenticationSuccessHandler defaultSuccessHandler =
15             new SavedRequestAwareAuthenticationSuccessHandler();
16         // 认证成功处理器
17         private AuthenticationSuccessHandler successHandler =
18             this.defaultSuccessHandler;
19         // 登录认证端点
20         private LoginUrlAuthenticationEntryPoint authenticationEntryPoint;
21         // 是否 自定义页面
22         private boolean customLoginPage;
23         // 登录页面
24         private String loginPage;
25         // 登录成功url
26         private String loginProcessingUrl;
27         // 认证失败处理器
28         private AuthenticationFailureHandler failureHandler;
29         // 认证路径是否放开
30         private boolean permitAll;
31         // 认证失败的url
32         private String failureUrl;
33
34         /**
35          * Creates a new instance with minimal defaults
36          */
37         public CaptchaAuthenticationFilterConfigurer() {
38             setLoginPage("/login/captcha");
39             this.authFilter = new CaptchaAuthenticationFilter();
40         }
41
42         public CaptchaAuthenticationFilterConfigurer<H> formLoginDisabled() {
43             this.formLoginEnabled = false;
44             return this;
45         }
46
47         public CaptchaAuthenticationFilterConfigurer<H> formLoginEnabled() {
48             this.formLoginEnabled = true;
49             return this;
50         }
51
52         public CaptchaAuthenticationFilterConfigurer<H> loginPage(String loginPage) {
53             this.loginPage = loginPage;
54             return this;
55         }
56
57         public CaptchaAuthenticationFilterConfigurer<H> loginProcessingUrl(String loginProcessingUrl) {
58             this.loginProcessingUrl = loginProcessingUrl;
59             return this;
60         }
61
62         public CaptchaAuthenticationFilterConfigurer<H> failureUrl(String failureUrl) {
63             this.failureUrl = failureUrl;
64             return this;
65         }
66
67         public CaptchaAuthenticationFilterConfigurer<H> authFilter(CaptchaAuthenticationFilter authFilter) {
68             this.authFilter = authFilter;
69             return this;
70         }
71
72         public CaptchaAuthenticationFilterConfigurer<H> authenticationDetailsSource(
73             AuthenticationDetailsSource<HttpServletRequest, ?> authenticationDetailsSource) {
74             this.authenticationDetailsSource = authenticationDetailsSource;
75             return this;
76         }
77
78         public CaptchaAuthenticationFilterConfigurer<H> defaultSuccessHandler(
79             AuthenticationSuccessHandler defaultSuccessHandler) {
80             this.defaultSuccessHandler = defaultSuccessHandler;
81             return this;
82         }
83
84         public CaptchaAuthenticationFilterConfigurer<H> savedRequestAwareAuthenticationSuccessHandler(
85             SavedRequestAwareAuthenticationSuccessHandler savedRequestAwareAuthenticationSuccessHandler) {
86             this.savedRequestAwareAuthenticationSuccessHandler =
87                 savedRequestAwareAuthenticationSuccessHandler;
88             return this;
89         }
90
91         public CaptchaAuthenticationFilterConfigurer<H> authenticationEntryPoint(
92             LoginUrlAuthenticationEntryPoint authenticationEntryPoint) {
93             this.authenticationEntryPoint = authenticationEntryPoint;
94             return this;
95         }
96
97         public CaptchaAuthenticationFilterConfigurer<H> permitAll(boolean permitAll) {
98             this.permitAll = permitAll;
99             return this;
100        }
101    }
```

```

42     public CaptchaAuthenticationFilterConfigurer<H>
43         captchaUserDetailsService(CaptchaUserDetailsService captchaUserDetailsService) {
44             this.captchaUserDetailsService = captchaUserDetailsService;
45             return this;
46         }
47
48     public CaptchaAuthenticationFilterConfigurer<H> captchaService(CaptchaService
49         captchaService) {
50         this.captchaService = captchaService;
51         return this;
52     }
53
54     public CaptchaAuthenticationFilterConfigurer<H> usernameParameter(String
55         usernameParameter) {
56         authFilter.setUsernameParameter(usernameParameter);
57         return this;
58     }
59
60     public CaptchaAuthenticationFilterConfigurer<H> captchaParameter(String
61         captchaParameter) {
62         authFilter.setCaptchaParameter(captchaParameter);
63         return this;
64     }
65
66     @Override
67     public void init(H http) throws Exception {
68         updateAuthenticationDefaults();
69         updateAccessDefaults(http);
70         registerDefaultAuthenticationEntryPoint(http);
71         // 这里禁用默认页面过滤器 如果你想自定义登录页面 可以自行实现 可能和FormLogin冲突
72         // initDefaultLoginFilter(http);
73         // 把对应的Provider也在init时写入HttpSecurity
74         initProvider(http);
75     }
76
77     @Override
78     public void configure(H http) throws Exception {
79
80         //这里改为使用前插过滤器方法
81         http.addFilterBefore(filter, LogoutFilter.class);
82     }
83
84     // 其它方法 同AbstractAuthenticationFilterConfigurer

```

其实就是在模仿 `AbstractAuthenticationFilterConfigurer` 及其实现类的风格把用的配置项实现一遍。这里值得一提的是 `CaptchaAuthenticationProvider` 的配置，参考其它配置的做法，用到的一些组件可以通过方法配置，也可以从 Spring IoC 中查找（参考 `getBeanOrNull` 方法，这个方法在 Spring Security 中随处可见，建议借鉴），这样更加灵活。

```

1     private void initProvider(H http) {
2

```

```

3         ApplicationContext applicationContext =
4             http.getSharedObject(ApplicationContext.class);
5             // 没有配置CaptchaUserDetailsService就去Spring IoC获取
6             if (captchaUserDetailsService == null) {
7                 captchaUserDetailsService = getBeanOrNull(applicationContext,
8                     CaptchaUserDetailsService.class);
8             }
9             // 没有配置CaptchaService就去Spring IoC获取
10            if (captchaService == null) {
11                captchaService = getBeanOrNull(applicationContext,
12                    CaptchaService.class);
12            }
13            // 初始化 Provider
14            CaptchaAuthenticationProvider captchaAuthenticationProvider =
15             this.postProcess(new CaptchaAuthenticationProvider(captchaUserDetailsService,
16                 captchaService));
16            // 会增加到ProviderManager的注册列表中
17            http.authenticationProvider(captchaAuthenticationProvider);
18        }

```

效果

我们来看看 `CaptchaAuthenticationFilterConfigurer` 的配置效果：

```

1     @Bean
2     SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http,
3         UserDetailsService userDetailsService) throws Exception {
4
5         http.csrf().disable()
6             .authorizeRequests()
7                 .mvcMatchers("/foo/**").access("hasAuthority('ROLE_USER')")
8                 .anyRequest().authenticated()
9                 .and()
10                // 所有的 AbstractHttpConfigurer 都可以通过apply方法加入HttpSecurity
11                .apply(new CaptchaAuthenticationFilterConfigurer<>())
12                // 配置验证码处理服务 这里直接true 方便测试
13                .captchaService((phone, rawCode) -> true)
14                // 通过手机号去拿验证码，这里为了方便直接写死了，实际phone和username做个映
15                射
15                .captchaUserDetailsService(phone ->
16                    userDetailsService.loadUserByUsername("felord"))
16                    // 默认认证成功跳转到/路径 这里改造成把认证信息直接返回json
17                    .successHandler((request, response, authentication) -> {
18                        // 这里把认证信息以JSON形式返回
19                        ServletServerHttpResponse servletServerHttpResponse = new
20                        ServletServerHttpResponse(response);
21                        MappingJackson2HttpMessageConverter
22                        mappingJackson2HttpMessageConverter = new MappingJackson2HttpMessageConverter();
23
24                        mappingJackson2HttpMessageConverter.write(authentication,
25                            MediaType.APPLICATION_JSON, servletServerHttpResponse);
25                    });
26
27
28         return http.build();
29     }

```

是不是要优雅很多。学习一定要模仿，先模仿成功，然后再分析思考为什么会模仿成功，最后形成自己的创造力。千万不要被一些陌生概念唬住，有些改造是不需要去深入了解细节的。下一章我们再对上面提到一些陌生概念进行一些解读。



17-深入理解Spring Security配置构建机制

在上一章中我们实现了验证码认证，但是配置没有充分利用Spring Security提供的机制。导致我们需要在Filter内部去定义一个 `AuthenticationManager` 来处理 `AuthenticationProvider` 注册的问题。

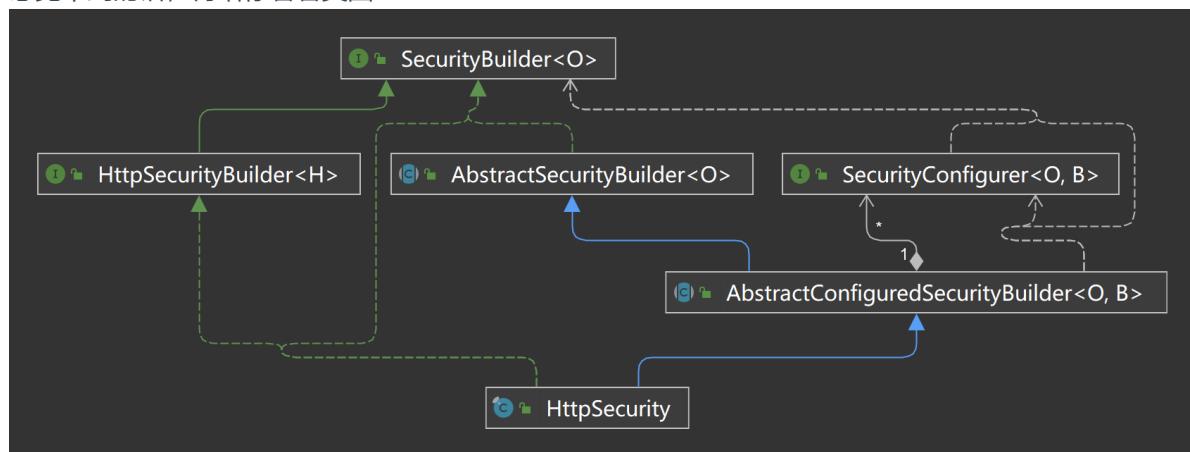
Spring Security配置机制

我们已经知道Spring Security的配置都通过 `HttpSecurity` 来做，并且我们在第9章也对 `HttpSecurity` 做了一个入门级别的介绍。为了优化自定义配置，就必须更深入去了解 `HttpSecurity`。

`HttpSecurity` 说实话不太简单：

```
1 public final class HttpSecurity extends  
2     AbstractConfiguredSecurityBuilder<DefaultSecurityFilterChain, HttpSecurity>  
3     implements SecurityBuilder<DefaultSecurityFilterChain>,  
4     HttpSecurityBuilder<HttpSecurity> {  
5         // 省略  
6     }
```

感觉不到的话，再给你看看类图：



为什么要这么复杂？我第一次看到 `HttpSecurity` 的结构时我怀疑我自己是不是Java开发。多年以后，当我深入学习了之后才理解了这种设计。作为一个框架，尤其是安全框架，配置必须足够灵活才能适用于更多的业务场景。Spring Security采取了配置与构建分离的架构设计来保证这一点。

配置与构建分离

配置只需要去收集配置项，构建只需要把所有的配置构建成目标对象。各干各的，分离职责，这种做法能够提高代码的可维护性和可读写性。Spring Security利用接口隔离把配置和构建进行高度抽象，提高灵活度，降低复杂度。不过这个体系依然非常庞大。为了降低学习难度需要把大问题拆解成小问题，各个击破，这种学习方法在学习一些复杂的抽象理论时很凑效。

SecurityBuilder

`SecurityBuilder` 就是对构建的抽象。你看上面的类图过于复杂，而看 `SecurityBuilder` 就非常的简单了。

```
1  public interface SecurityBuilder<O> {
2      // 构建
3      O build() throws Exception;
4  }
```

就一个动作，构建泛化的目标对象 `O`。通过下面这一组抽象和具体的定义我想你应该明白 `SecurityBuilder` 了吧。

```
1  // 抽象
2  SecurityBuilder -> O
3  // 具体
4  HttpSecurity->DefaultSecurityFilterChain
```

一句话，构建的活都是我来干。

AbstractSecurityBuilder

`AbstractSecurityBuilder` 是对 `SecurityBuilder` 的实现。源码如下：

```
1  public abstract class AbstractSecurityBuilder<O> implements SecurityBuilder<O> {
2
3      private AtomicBoolean building = new AtomicBoolean();
4
5      private O object;
6
7      @Override
8      public final O build() throws Exception {
9          if (this.building.compareAndSet(false, true)) {
10              //构建的核心逻辑由钩子方法提供
11              this.object = doBuild();
12              return this.object;
13          }
14          throw new AlreadyBuiltException("This object has already been built");
15      }
16      // 获取构建目标对象
17      public final O getObject() {
18          if (!this.building.get()) {
19              throw new IllegalStateException("This object has not been built");
20          }
21          return this.object;
22      }
23
24      /**
25      * 钩子方法
26      */
```

```
27     protected abstract O doBuild() throws Exception;
28
29 }
```

它通过原子类 `AtomicBoolean` 对构建方法 `build()` 进行了调用限制：每个目标对象只能被构建一次，避免安全策略发生不一致的情况。构建方法还加了 `final` 关键字，不可覆写！构建的核心逻辑通过预留的钩子方法 `doBuild()` 来扩展，钩子方法是很常见的一种继承策略。另外 `AbstractSecurityBuilder` 还提供了获取已构建目标对象的方法 `getObject`。

一句话，构建的活我只干一次。

HttpSecurityBuilder

```
1  public interface HttpSecurityBuilder<H extends HttpSecurityBuilder<H>>
2      extends SecurityBuilder<DefaultSecurityFilterChain> {
3
4      // 根据类名获取配置
5      <C extends SecurityConfigurer<DefaultSecurityFilterChain, H>> C
6      getConfigurer(Class<C> clazz);
7      // 根据类名移除配置
8      <C extends SecurityConfigurer<DefaultSecurityFilterChain, H>> C
9      removeConfigurer(Class<C> clazz);
10     // 把某个对象设置为共享，以便于在多个SecurityConfigurer中使用
11     <C> void setSharedObject(Class<C> sharedType, C object);
12     // 获取某个共享对象
13     <C> C getSharedObject(Class<C> sharedType);
14     // 添加额外的 AuthenticationProvider
15     H authenticationProvider(AuthenticationProvider authenticationProvider);
16     // 添加额外的 UserDetailsService
17     H userDetailsService(UserDetailsService userDetailsService) throws Exception;
18     // 在过滤器链已有的afterFilter类后面注册一个过滤器
19     H addFilterAfter(Filter filter, Class<? extends Filter> afterFilter);
20     // 在过滤器链已有的beforeFilter类前面注册一个过滤器
21     H addFilterBefore(Filter filter, Class<? extends Filter> beforeFilter);
22     // 在过滤器链注册一个过滤器，该过滤器必须在内置注册表 FilterOrderRegistration 中
23     H addFilter(Filter filter);
```

`HttpSecurityBuilder` 对 `DefaultSecurityFilterChain` 的构建进行了增强，为其构建器增加了一些额外的获取配置或管理配置的入口，参见上面的注释。补充一点这个接口最大的功能就是打通了构建和配置的关系，可以操作下面要讲的 `SecurityConfigurer`。

一句话，我只构建 `DefaultSecurityFilterChain`。

SecurityConfigurer

`SecurityConfigurer` 是对配置的抽象。配置只是手段，构建才是目的，因此配置是对构建的配置。

```
1  public interface SecurityConfigurer<O, B extends SecurityBuilder<O>> {
2      // 构建器初始化需要注入的配置，用来后续的信息共享
3      void init(B builder) throws Exception;
4      // 其它的一些必要配置
5      void configure(B builder) throws Exception;
6  }
```

`SecurityConfigurer` 有两个方法，都非常重要：

- 一个是 `init` 方法，这个方法你可以认为是 `SecurityBuilder` 构造函数的逻辑。如果你想在 `SecurityBuilder` 初始化的时候执行一些逻辑或者在后续配置中共享一些变量的话就可以在 `init` 方法中去实现。
- 另一个方法是 `configure`，为 `SecurityBuilder` 配置一些必要的属性。

到这里还没完？这两个方法有着明确的先后执行顺序。在一次构建内可能有多个 `SecurityConfigurer`，只有全部的 `init` 逐个执行完毕后才会逐个执行 `configure` 方法。相关的源码在 `AbstractConfiguredSecurityBuilder` 中的标记部分：

```
1     @Override
2     protected final O doBuild() throws Exception {
3         synchronized (this.configurers) {
4             this.buildState = BuildState.INITIALIZING;
5             beforeInit();
6             // ① 执行所有的初始化方法
7             init();
8             this.buildState = BuildState.CONFIGURING;
9             beforeConfigure();
10            // ② 执行所有的configure方法
11            configure();
12            this.buildState = BuildState.BUILDING;
13            O result = performBuild();
14            this.buildState = BuildState.BUILT;
15            return result;
16        }
17    }
```

一句话，配置 `SecurityBuilder` 的事都是我来干。

SecurityConfigurerAdapter

`SecurityConfigurer` 在某些场景下是有局限性的，它不能获取正在配置的 `SecurityBuilder`，因此你无法进一步操作 `SecurityBuilder`，配置的扩展性将大打折扣。因此引入了 `SecurityConfigurerAdapter` 来扩展 `SecurityConfigurer`。

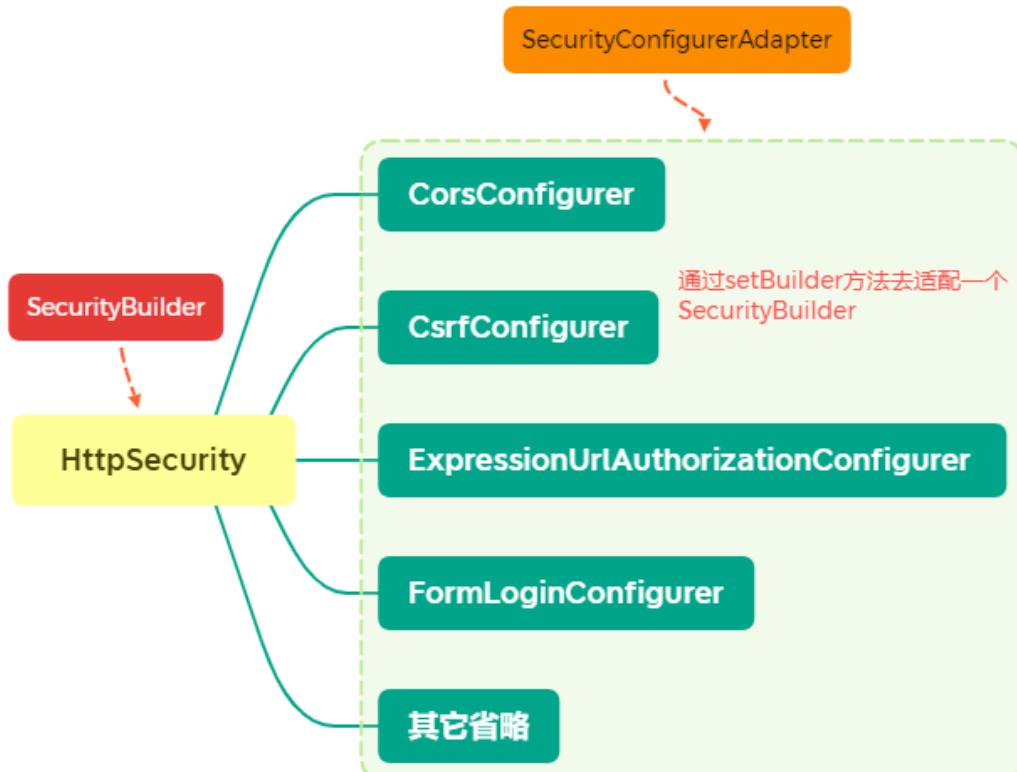
```
1
2     public abstract class SecurityConfigurerAdapter<O, B extends SecurityBuilder<O>>
3         implements SecurityConfigurer<O, B> {
4
5         private B securityBuilder;
6
7         private CompositeObjectPostProcessor objectPostProcessor = new
8             CompositeObjectPostProcessor();
9
10        @Override
11        public void init(B builder) throws Exception {
12        }
13
14        @Override
15        public void configure(B builder) throws Exception {
16        }
17        // 获取正在配置的构建器，以暴露构建器的api
18        public B and() {
19            return getBuilder();
```

```

18     }
19
20     protected final B getBuilder() {
21         Assert.state(this.securityBuilder != null, "securityBuilder cannot be
22             null");
23         return this.securityBuilder;
24     }
25
26     // 用复合对象后置处理器去处理对象，以改变一些对象的特性
27     @SuppressWarnings("unchecked")
28     protected <T> T postProcess(T object) {
29         return (T) this.objectPostProcessor.postProcess(object);
30     }
31     // 添加一个ObjectPostProcessor到符合构建器
32     public void addObjectPostProcessor(ObjectPostProcessor<?>
33         objectPostProcessor) {
34         this.objectPostProcessor.addObjectPostProcessor(objectPostProcessor);
35     }
36     // 设置 需要配置的构建器，这样可以让多个SecurityConfigurerAdapter去配置一个
37     SecurityBuilder
38     public void setBuilder(B builder) {
39         this.securityBuilder = builder;
40     }
41     // 其它省略
42 }

```

这样可以指定 `SecurityBuilder`，而且可以把 `SecurityBuilder` 暴露出来，随时随地去调整 `SecurityBuilder`，灵活性大大提高。



具体说的话，你可以通过 `and()` 方法获取 `SecurityBuilder` 并对 `SecurityBuilder` 的其它配置项进行操作，比如上图中 `SecurityConfigurerAdapter` 之间的切换。

除此之外还引入了 `ObjectPostProcessor` 来后置操作一些并不开放的内置对象。关于 `ObjectPostProcessor` 会找个合适的场景去讲解它。

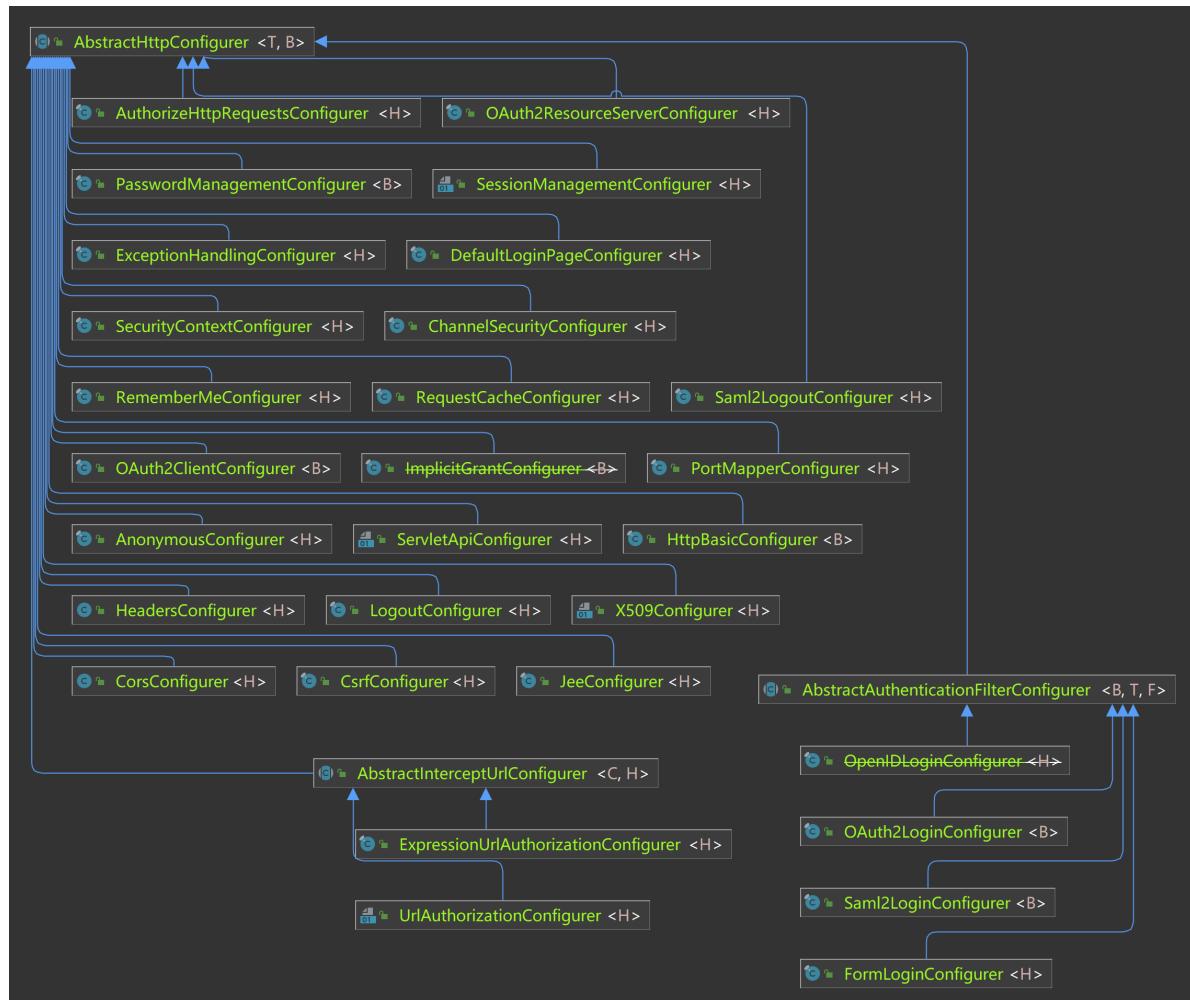
一句话，配置 `SecurityBuilder` 不算什么，灵活适配才是花活。

AbstractHttpConfigurer

不是所有的配置都是有用的，有些配置我们希望有个关闭的入口功能。比如 `csrf` 功能，控制着 `csrf` 的配置的是 `CsrfConfigurer`，如果 `CsrfConfigurer` 有一个关闭功能就好了。因此从 `SecurityConfigurerAdapter` 衍生出 `AbstractHttpConfigurer` 来满足这个需求。

```
1  public abstract class AbstractHttpConfigurer<T extends AbstractHttpConfigurer<T,  
B>, B extends HttpSecurityBuilder<B>> extends  
SecurityConfigurerAdapter<DefaultSecurityFilterChain, B> { // 关闭当前配置  
    @SuppressWarnings("unchecked") public B disable() {  
        getBuilder().removeConfigurer(getClass()); return getBuilder(); } //  
        增强了父类的新增ObjectPostProcessor方法  
        @SuppressWarnings("unchecked") public T  
        withObjectPostProcessor(ObjectPostProcessor<?> objectPostProcessor) {  
            addObjectPostProcessor(objectPostProcessor); return (T) this; } }
```

`AbstractHttpConfigurer` 的实现类非常多，日常的配置项大都由 `AbstractHttpConfigurer` 的实现类来控制。



这个类是做定制化配置的一个重要入口之一，如果你想精通 Spring Security，这个类一定要掌握。

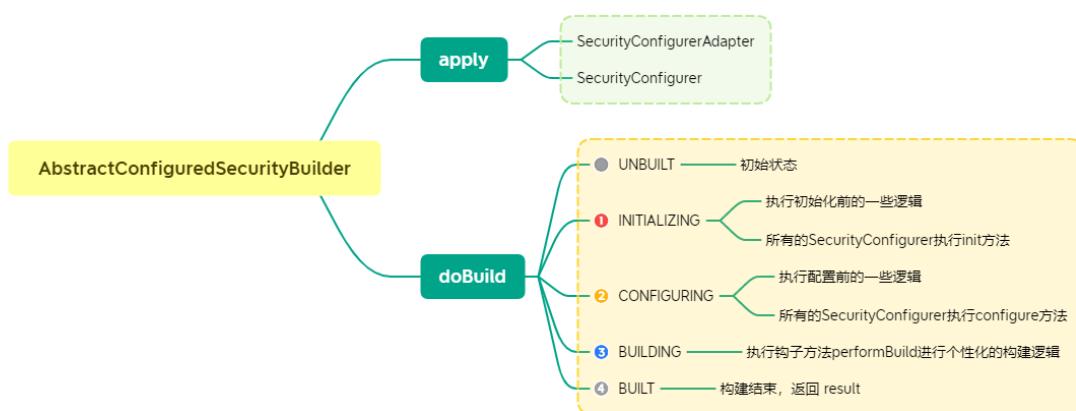
一句话，我能“杀”我自己。

AbstractConfiguredSecurityBuilder

我们希望有多个 `SecurityConfigurer` 配置 `SecurityBuilder`，表单登录的、会话管理、`csrf` 等等。用到什么配置什么，让配置基于策略。因此引入了 `AbstractConfiguredSecurityBuilder`。

```
1     public <C extends SecurityConfigurerAdapter<O, B>> C apply(C configurer)
2         throws Exception {           // 把 objectPostProcessor注入到configurer
3             configurer.addObjectPostProcessor(this.objectPostProcessor);           // 为
4             SecurityConfigurerAdapter 设置Builder 以便于能够get到           // 注意区别于其它
5             SecurityConfigurer configurer.setBuilder((B) this);           add(configurer);
6             return configurer; }   public <C extends SecurityConfigurer<O, B>> C apply(C
7             configurer) throws Exception {           add(configurer);           return configurer;
8         }
```

通过上面两个 `apply` 方法就可以把所有的 `SecurityConfigurer` 适配进来，然后通过 `doBuilder` 进行分阶段、精细化构建生命周期。你可以在各个生命周期阶段进行一些必要的操作。



一句话，最终的构建都由我来进行精细化处理。



微信搜一搜

码农小胖哥

18-匿名用户的作用

匿名用户是很多新手不理解的一个概念，这一章我尝试来解读一下匿名用户，希望能帮你更好的理解这个概念。

流程一致性

通常情况下所有的资源访问都应该是有条件的。用来验证这些条件的流程也应该是一致的。我们来看实际生活中一个例子，老王是一家公司的老板，他的车进地下车库是免费的，其他人进地下车库是计费的。我们来思考如何实现这个需求。

如果流程不一致的话，需要两条道，一条是VIP通道，一条是大众通道，这两个通道的维护成本会很高，还有人会经常走错道，甚至招致不满，凭啥他要搞特权；如果流程一致，这事就好办多了，不管是谁都得按同一个通道流程进入停车场，只需要维护一个VIP标签就行了，成本大大降低，流程也简化了。这个VIP标签就是所谓的“匿名用户”。

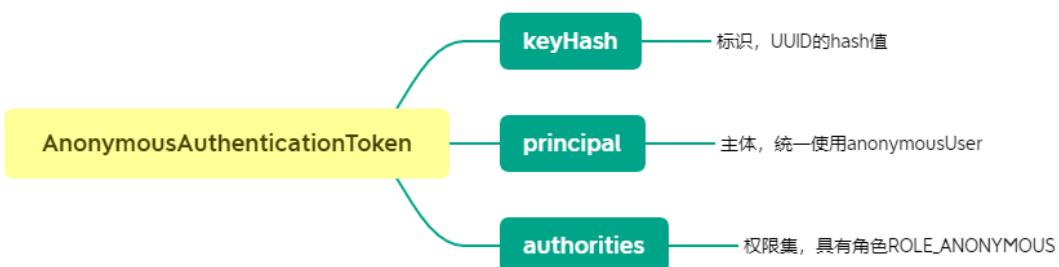
匿名用户

Spring Security 中专门设计了匿名用户，它的作用其实也是为了在保证流程一致的前提下执行一些特殊的认证逻辑，比如程序的登录、主页的数据接口，这些未认证的用户场景需要绕过访问控制检查，通过引入一个特殊的“匿名身份”可以做到这一点，匿名用户可以做什么、不可以做什么都可以轻松定义，这就是我们所说的匿名身份验证。

请注意：“经过匿名身份验证”的用户和未经身份验证的用户之间没有真正的差异，你可以认为匿名用户就是未认证用户，你也可以认为匿名用户是执行了匿名认证流程后的认证用户。

匿名用户的配置

匿名用户是认证体系最后的一道认证流程，负责匿名认证的过滤器是 `AnonymousAuthenticationFilter`，当发现请求不具备任何其它认证条件后，会生成一个 `AnonymousAuthenticationToken`，它包含三个属性：



`keyHash` 仅仅在 `AnonymousAuthenticationFilter` 和 `AnonymousAuthenticationProvider` 之间共享，以避免一些恶意客户端去伪造 `AnonymousAuthenticationToken`。

权限控制只需要针对 `ROLE_ANONYMOUS` 进行限制即可，也可以通过认证投票器 `AuthenticatedVoter` 的 `IS_AUTHENTICATED_ANONYMOUSLY` 属性来限制。

下面这几种配置都可以用来控制匿名用户的访问权限：

```
1      http
2          .authorizeRequests()
3              .mvcMatchers("/anonymous0").access("hasAuthority('ROLE_USER')")
4              .mvcMatchers("/anonymous1").hasAuthority("ROLE_ANONYMOUS")
5              .mvcMatchers("/anonymous2").hasRole("ANONYMOUS")
6              .mvcMatchers("/anonymous3").access("isAnonymous()")
7              .mvcMatchers("/anonymous4").access("IS_AUTHENTICATED_ANONYMOUSLY")
8              .mvcMatchers("/anonymous5").anonymous()
```

获取匿名用户

Spring MVC中使用它自己的参数解析器来获取当前Principal：

```
1  @GetMapping("/")
2  public String method(Authentication authentication) {
3      if (authentication instanceof AnonymousAuthenticationToken) {
4          return "anonymous";
5      } else {
6          return "not anonymous";
7      }
8  }
```

以上方式将永远返回 `not anonymous`，即使是匿名请求。如果您想获取匿名请求的 `Authentication`，请改用 `@CurrentSecurityContext`：

```
1  @GetMapping("/")
2  public String method(@CurrentSecurityContext SecurityContext context) {
3      Authentication authentication = context.getAuthentication()
4      if (authentication instanceof AnonymousAuthenticationToken) {
5          return "anonymous";
6      } else {
7          return "not anonymous";
8      }
9  }
```



19-基于注解的权限控制

基于JavaConfig的访问控制在前面已经讲过，Spring Security还提供基于注解的访问控制。

开启方法注解访问控制

Spring Security默认是关闭方法注解的，开启它只需要通过引入 `@EnableGlobalMethodSecurity` 注解即可：

```

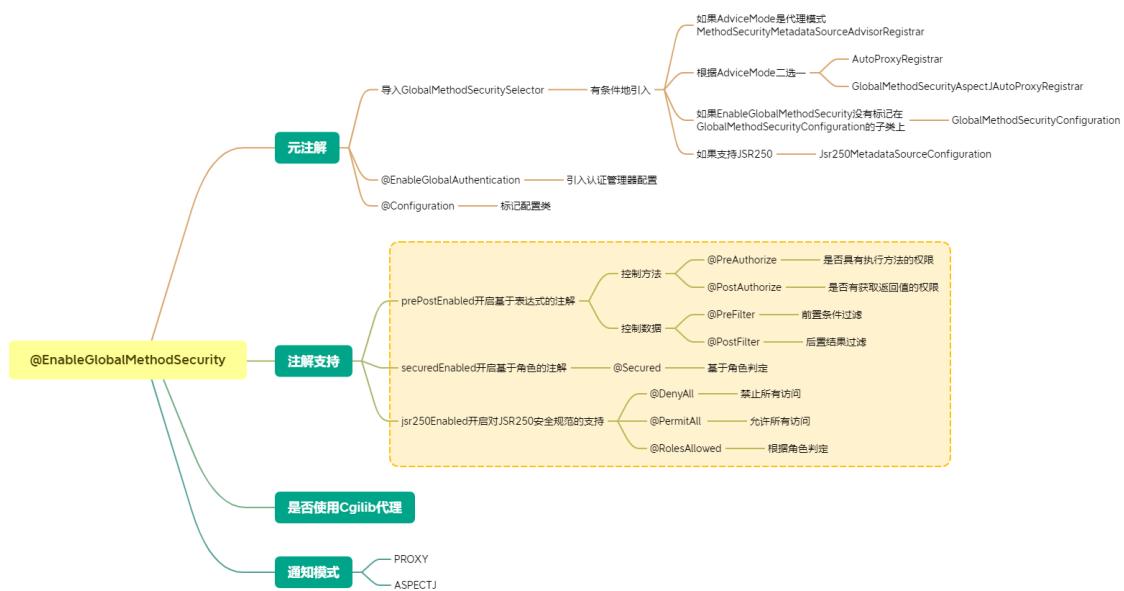
1  /**
2   * 开启方法安全注解
3   *
4   * @author felord.cn
5   */
6  @EnableGlobalMethodSecurity(prePostEnabled = true,
7      securedEnabled = true,
8      jsr250Enabled = true)
9  public class MethodSecurityConfig {
10 }

```

`@EnableGlobalMethodSecurity` 提供了 `prePostEnabled`、`securedEnabled` 和 `jsr250Enabled` 三种方式，可以根据需要选择使用这三种的一种或者其中几种。

@EnableGlobalMethodSecurity

`@EnableGlobalMethodSecurity` 的思维导图可以帮助你梳理相关的逻辑。



黄色区域是注解访问控制的基本知识点，需要重点掌握，我们先来看看基本的用法。

@PreAuthorize 和 @PostAuthorize

`prePostEnabled` 等于 `true` 时启用。

在标记的方法调用之前或者之后，通过 `SpEL` 表达式来计算是否可以调用方法或者调用后是否可以返回结果。总结了一些常用的表达式样例：

SpEL 表达式	说明
<code>principal.username ne 'felord'</code>	当前 <code>principal</code> 的 <code>username</code> 不等于 <code>felord</code>
<code>authentication.authorities.contains('ROLE_ADMIN')</code>	当前 <code>Authentication</code> 的 <code>authorities</code> 包含 <code>ROLE_ADMIN</code>
<code>hasRole('ADMIN')</code>	当前用户必须有角色 <code>ROLE_ADMIN</code> ，等同于上面
<code>hasAnyRole('ADMIN', 'USER')</code>	当前用户角色必须有 <code>ROLE_ADMIN</code> 或者 <code>USER</code>

SpEL表达式	ROLE_USER 说明
hasAuthority('ROLE_ADMIN')	同 hasRole
hasAnyAuthority('ROLE_ADMIN', 'ROLE_USER')	同 hasAnyRole
#requestParam eq 'felord'	当前请求参数 requestParam (可选, 这里是字符串示例)值等于 felord

其它表达式可参考 [SpEL官方文档](#)。

如果用户 felord 访问下面这个接口, 方法不但不执行还会 403。

```

1  /**
2   * 当前用户名不是felord 才能访问 否则403
3   * @param req req
4   * @return map
5   */
6  @GetMapping("/prepost")
7  @PreAuthorize("authentication.principal.username ne 'felord'")
8  public Map<String, String> prepost(String req){
9      Map<String, String> map = Collections.singletonMap("req", req);
10     System.out.println("map = " + map);
11     return map;
12 }
```

如果方法上的注解改为 `@PostAuthorize("authentication.principal.username ne 'felord'")`, 控制台会打印

```

2022-01-05 15:55:30.460 DEBUG 13780 --- [nio-8085-exec-4] o.s.security.web.FilterChainProxy      : Secured GET /foo/prepost?req=xxxxx
2022-01-05 15:55:30.461 TRACE 13780 --- [nio-8085-exec-4] o.s.web.servlet.DispatcherServlet       : GET "/foo/prepost?req=xxxxx", parameters={masked}, head
2022-01-05 15:55:30.461 TRACE 13780 --- [nio-8085-exec-4] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped to cn.felord.springsecurity.controller.FooController@...
2022-01-05 15:55:30.472 TRACE 13780 --- [nio-8085-exec-4] o.s.web.method.HandlerMethod           : Arguments: [xxxxx]
2022-01-05 15:55:30.475 DEBUG 13780 --- [nio-8085-exec-4] o.s.s.a.i.a.MethodSecurityInterceptor  : Authorized ReflectiveMethodInvocation: public java.util.Map<String, String> cn.felord.springsecurity.controller.FooController.prepost(java.lang.String)
map = {req=xxxxx}
2022-01-05 15:55:30.478 DEBUG 13780 --- [nio-8085-exec-4] e.m.ExpressionBasedPostInvocationAdvice : PostAuthorize expression rejected access
2022-01-05 15:55:30.483 TRACE 13780 --- [nio-8085-exec-4] o.s.web.servlet.DispatcherServlet       : Failed to complete request
org.springframework.security.access.AccessDeniedException: Access is denied
Create breakpoint : Access is denied
```

从日志上可以明显看出方法确实执行了, 但是还是给了 403。

@PreFilter和@PostFilter

`prePostEnabled` 等于 `true` 时启用。

这两个注解可以看做 `@PreAuthorize` 和 `@PostAuthorize` 的加强版。它们除了能实现 `@PreAuthorize` 和 `@PostAuthorize` 外还具有过滤请求响应数据的能力。限定处理的数据类型有 `Collection`、`数组`、`Map`、`Stream`。举个例子:

```

1  // ids = ["Felordcn", "felord", "jetty"]
2  @PostMapping("/prefilter")
3  @PreFilter(value = "hasRole('ADMIN') and filterObject.startsWith('f')",
4             filterTarget = "ids")
5  public Collection<String> preFilter(@RequestBody Collection<String> ids){
6      // ids = ["felord"]
7      System.out.println("ids = " + ids);
8      return ids;
9 }
```

上面的接口方法有两层含义:

- 当前用户必须持有角色 `ROLE_ADMIN`，否则方法不执行。**这一条件不存在的话方法一定会执行。**
- 如果方法执行的话，入参 `ids` 集合中不包含 `f` 开头的元素都会被移除，返回值为 `felord`。

过滤元素的底层是 `java.util.Collection#remove(Object)`；另外多个入参需要使用 `filterTarget` 指定参数名称。

`@PostFilter` 也很好理解，拿下面的方法来说：

```

1      @GetMapping("/postfilter")
2      @PostFilter("hasRole('ADMIN') and filterObject.startsWith('f')")
3      public Collection<String> postfilter(){
4          List<String> list = new ArrayList<>();
5          list.add("Felordcn");
6          list.add("felord");
7          list.add("jetty");
8          // list = ["felord"]
9          System.out.println("list = " + list);
10         return list;
11     }

```

无论是否满足 `@PostFilter` 表达式的条件，都会打印 `list = ["felord"]`，条件成立返回 `list`，不成立则返回 `403`。

这两个注解用来控制请求和相应中集合、流中的数据。

`@Secured`

`securedEnabled` 等于 `true` 时启用。

该注解功能要简单的多，默认情况下只能基于角色（默认需要带前缀 `ROLE_`）集合来进行访问控制决策。该注解的机制是只要其声明的角色集合（`value`）中包含当前用户持有的任一角色就可以访问，也就是用户的角色集合和 `@Secured` 注解的角色集合要存在非空的交集。不支持使用 `SpEL` 表达式进行决策。过于简单不再演示。

`@Secured` 等同于 `hasAnyAuthority`。

JSR-250

`jsr250Enabled` 等于 `true` 时启用。

启用 JSR-250 安全控制注解，这属于 JavaEE 的安全规范（现为 jakarta 项目）。Spring Security 中使用了 JavaEE 安全注解中的以下三个：

- `@DenyAll` 拒绝所有的访问
- `@PermitAll` 同意所有的访问
- `@RolesAllowed` 用法和上面的 `@Secured` 一样。

注解控制的优劣

使用注解的好处就是绑定了接口方法，控制粒度非常细，甚至能做一些数据层面的访问控制。劣势在于它是静态织入 Java 代码中的，灵活性难以把握。



微信搜一搜

码农小胖哥

20-基于SpEL表达式和注解进行灵活权限控制

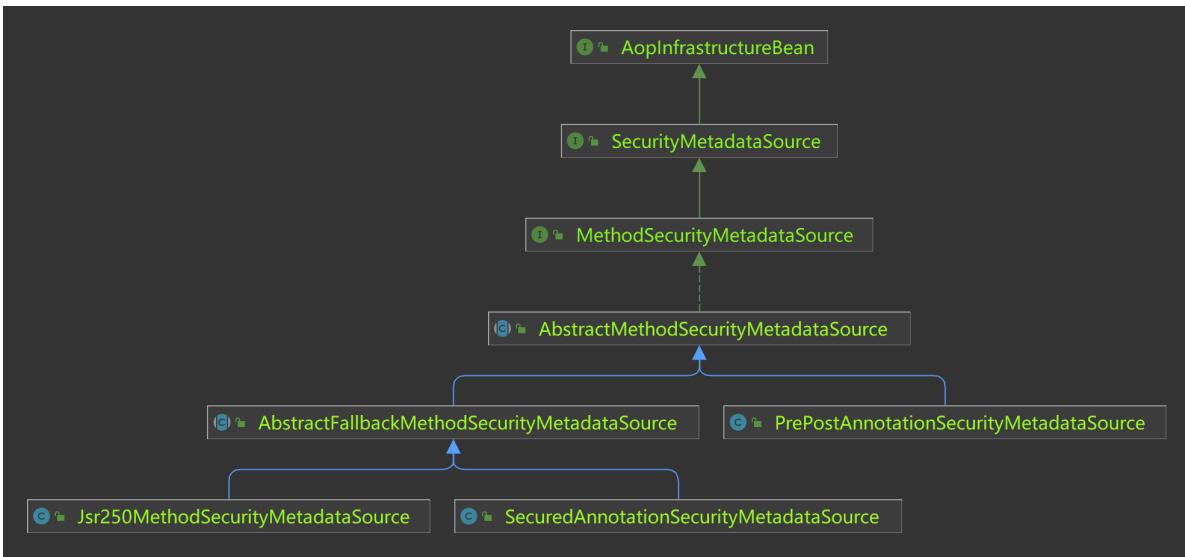
注解权限控制因为是静态的，所以天然不具备灵活性，你想改规则就要重新编译Java代码。这并不意味着不能实现动态权限控制。

GlobalMethodSecurityConfiguration

默认情况下 `GlobalMethodSecurityConfiguration` 是方法安全控制的核心配置。大部分情况下使用默认配置就好，不过它也暴露了两个方法供我们实现一些自定义逻辑。

```
1  @EnableGlobalMethodSecurity(prePostEnabled = true,
2      securedEnabled = true,
3      jsr250Enabled = true)
4  public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
5
6
7      @Override
8      protected MethodSecurityExpressionHandler createExpressionHandler() {
9          // todo 自定义一个方法表达式处理器 难度较高 而且容易破坏现有的表达式机制 不推荐
10         return null;
11     }
12
13     @Override
14     protected MethodSecurityMetadataSource customMethodSecurityMetadataSource() {
15         //todo 这个是用来作自定义的权限控制的 是用来实现额外的自定义。
16         return null;
17     }
18 }
```

自定义 `MethodSecurityExpressionHandler` 容易破坏现有的三种注解的机制，因此不推荐去改写。另一种途径，去自定义 `MethodSecurityMetadataSource` 还是比较简单的。为什么说简单呢？



上面是对应的三种注解的实现，参考三种注解实现机制去模仿一个就行了，并不需要知道太多的原理，只需要知道实现方法的关键要素即可：

```

1     @Override
2     public Collection<ConfigAttribute> getAttributes(Method method, Class<?>
3             targetClass) {
4         //todo 要实现的方法
5     }
  
```

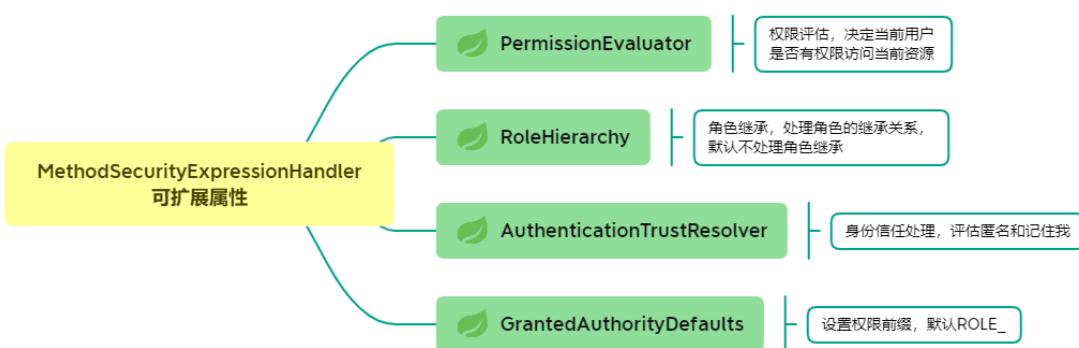
这个方法的含义就是通过方法的信息去获取该方法关联的安全配置信息（角色），你只需要知道如何通过参数 `method` 和 `targetClass` 去获取当前接口方法的一些信息，然后设计一下和角色的动态绑定关系就能实现了，**切记不要钻牛角尖**。这里只说一下思路就不进行演示了。

虽然简单，但是还是要有源码的解读能力的。

另一种简单的实现方式

之所以不写上面的例子，我也觉得这个实现起来太繁琐了，对新手不友好。但是我在 `GlobalMethodSecurityConfiguration` 中发现虽然 `MethodSecurityExpressionHandler` 的四种扩展属性。

这种扩展机制很有必要学习一下。



这四个属性中最有用的就是 `PermissionEvaluator` 和 `RoleHierarchy` 了。

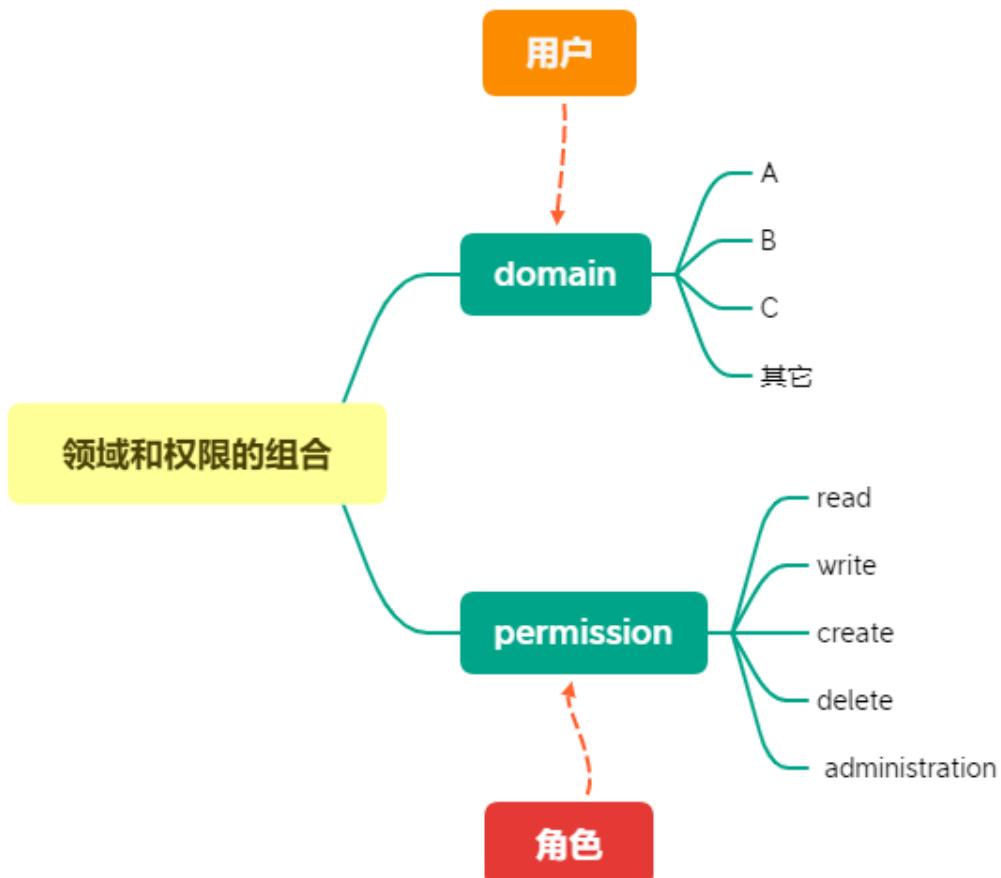
RoleHierarchy 是处理角色继承逻辑的，并不直接涉及到动态权限。

PermissionEvaluator

这个接口是用作权限评估的，用来决定用户是否有权限访问**目标域对象**。这里的**目标领域对象**是一个非常抽象的概念，不太好理解。先来看接口：

```
1  public interface PermissionEvaluator extends AopInfrastructureBean {  
2      // 传入了 当前认证用户信息    目标领域对象    许可信息  
3      boolean hasPermission(Authentication authentication, Object  
targetDomainObject, Object permission);  
4      // 传入了 当前认证用户信息    目标id    目标领域对象    许可信息  
5      boolean hasPermission(Authentication authentication, Serializable targetId,  
String targetType, Object permission);  
6  
7  }
```

看了接口后我大致明白了**目标领域对象**的作用。比如我是A部门的老大，A部门我说了算，但是我去B部门协调的时候，B部门的就不会随便鸟我。这个例子中**部门**就是**目标领域对象**。**领域对象加上权限标识符**就可以灵活定义一个具体的权限，比如 **A:administration** 就可以表示A部门的管理权限，将 **A** 关联到用户本身（**Authentication**），将 **administration** 关联到 **角色**，后面只需要实现 **A:administration** 关联到接口即可。



假设用户的 **domain** 和角色只有匹配接口的 **domain** 和 **permission** 才能访问，用 **PermissionEvaluator** 就可以这样实现：

```

1  /**
2  * 开启方法安全注解
3  *
4  * @author felord.cn
5  */
6 @EnableGlobalMethodSecurity(prePostEnabled = true,
7     securedEnabled = true,
8     jsr250Enabled = true)
9 public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
10 /**
11 * 设置为Bean
12 * @return PermissionEvaluator
13 */
14 @Bean
15 PermissionEvaluator resourcePermissionEvaluator(){
16     return new ResourcePermissionEvaluator();
17 }
18
19     public static class ResourcePermissionEvaluator implements
20 PermissionEvaluator {
21
22     @Override
23     public boolean hasPermission(Authentication authentication, Object
24 targetDomainObject, Object permission) {
25
26         Collection<? extends GrantedAuthority> authorities =
27         authentication.getAuthorities();
28         //todo 查询当前Authentication 是否包含targetDomainObject 包含就继续 不包
29         //含false
30         // A:administration 包含哪些角色
31         Set<GrantedAuthority> grantedAuthorities =
32         findRoleByKey(targetDomainObject,permission);
33         // 角色集合去交集 有交集 肯定 size 小于两个集合size之和 没有交集 等于或
34         //者大于两个set大小之和
35         int a = authorities.size();
36         int b = grantedAuthorities.size();
37         grantedAuthorities.addAll(authorities);
38         int c = grantedAuthorities.size();
39         return a + b > c;
40     }
41
42     @Override
43     public boolean hasPermission(Authentication authentication, Serializable
44 targetId, String targetType, Object permission) {
45
46         // 如果设置的id 当前用户不能访问 该条数据
47         return false;
48     }
49
50     private Set<GrantedAuthority> findRoleByKey(Object targetDomainObject,
51 Object permission){
52         String key = targetDomainObject + ":" + permission;
53         //TODO          查询 key 和 authority 的关联关系
54         //
55         //          模拟 permission 关联 角色 根据key 去查
56         grantedAuthorities
57         Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
58         grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
59         return "A:administration".equals(key)?grantedAuthorities:new
60         HashSet<>();

```

```
49         }
50     }
51 }
```

借助于 `@PreAuthorize` 注解写个接口：

```
1  /**
2   * 当前用户持有A领域和角色ROLE_ADMIN
3   *
4   * @return the collection
5   */
6  @GetMapping("/postfilter")
7  @PreAuthorize("hasPermission('A', 'administration')")
8  public Collection<String> postfilter(){
9      List<String> list = new ArrayList<>();
10     list.add("Felordcn");
11     list.add("felord");
12     list.add("jetty");
13     System.out.println("list = " + list);
14     return list;
15 }
```

那么就可以用 `domain` 和角色来灵活控制当前用户的权限了。另外一个方法似乎可以通过 `targetId` 用于数据控制，你可以试一试。



21-基于SpEL表达式的动态权限控制

网上关于Spring Security动态权限控制的教程，大都通过自定义 `FilterInvocationSecurityMetadataSource` 和 `AccessDecisionManager` 两个接口实现了动态权限控制。这里需要我们做的事情比较多，有一定的学习成本。今天来介绍一种更加简单和容易理解的方法实现动态权限控制。

基于表达式的访问控制

```
1  httpSecurity.authorizeRequests()
2      .anyRequest()
3      .access("hasRole('admin')")
```

这种方式不用多说了吧，我们配置了表达式 `hasRole('admin')` 后，Spring Security会调用 `SecurityExpressionRoot` 的 `hasRole(String role)` 方法来判断当前用户是否持有角色 `admin`，进而作出是否放行的决策。这种方式除了可以静态的权限控制之外还能够动态的权限控制。

基于Bean的访问控制表达式

Spring Security扩展了对表达式进行了扩展，支持引用任何公开的Spring Bean，假如我们有一个实现下列接口的Spring Bean：

```
1  /**
2   * 角色检查器接口.
3   *
4   * @author n1
5   * @since 2021 /4/6 16:28
6   */
7  public interface RoleChecker extends InitializingBean {
8
9      /**
10       * Check boolean.
11       *
12       * @param authentication the authentication
13       * @param request         the request
14       * @return the boolean
15       */
16      boolean check(Authentication authentication, HttpServletRequest request);
17 }
```

基于JDBC的角色检查，最好这里做个缓存：

```
1  /**
2   * 基于jdbc的角色检查 最好这里做个缓存
3   * @author n1
4   * @since 2021/4/6 16:43
5   */
6  public class JdbcRoleChecker implements RoleChecker {
7      // 系统集合的抽象实现，这里你可以采用更加合理更加效率的方式
8      private Supplier<Set<AntPathRequestMatcher>> supplier;
9
10
11     @Override
12     public boolean check(Authentication authentication, HttpServletRequest
request) {
13         Collection<? extends GrantedAuthority> authorities =
authentication.getAuthorities();
14
15         // 当前用户的角色集合
16         System.out.println("authorities = " + authorities);
17         //todo 这里自行实现比对逻辑
18         // supplier.get().stream().filter(matcher -> matcher.matches(request));
19         // true false 为是否放行
20         return true;
21     }
22
23     @Override
24     public void afterPropertiesSet() throws Exception {
25         Assert.notNull(supplier.get(), "function must not be null");
26     }
27 }
```

我们就可以这样配置 `HttpSecurity`：

```
1 httpSecurity.authorizeRequests()
2     .anyRequest()
3         .access("@roleChecker.check(authentication, request)")
```

通过 `RoleChecker` 中的 `Authentication` 我们可以获得当前用户的信息，尤其是权限集。通过 `HttpServletRequest` 我们可以获得当前请求的URI。该URI在系统中的权限集和用户的权限集进行交集判断就能作出正确的访问决策。

有些时候我们的访问URI中还包含了路径参数，例如 `/foo/{id}`。我们也可以通过基于Bean的访问控制表达式结合具体的 `id` 值来控制。这时应该这么写：

```
1 /**
2  * 角色检查器接口.
3  *
4  * @author n1
5  * @since 2021 /4/6 16:28
6 */
7 public interface RoleChecker extends InitializingBean {
8
9     /**
10      * Check boolean.
11      *
12      * @param authentication the authentication
13      * @param request        the request
14      * @return the boolean
15      */
16     boolean check(Authentication authentication, String id);
17 }
```

对应的配置为：

```
1 httpSecurity.authorizeRequests()
2     .antMatchers("/foo/{id}/**")
3         .access("@roleChecker.check(authentication,#id)")
```

这样当 `/foo/123` 请求被拦截后，`123` 就会赋值给 `check` 方法中的 `id` 处理。

和实现 `PermissionEvaluator` 接口比起来，这个完全可以不依赖注解，天然支持路径Ant风格。



22-新的基于URL权限控制

Spring Security 5.5 增加了一个新的授权管理器接口 `AuthorizationManager<T>`，它让动态权限的控制接口化了。

AuthorizationManager

它用来检查当前认证信息 `Authentication` 是否可以访问特定对象 `T`。`AuthorizationManager` 将访问决策抽象更加泛化。

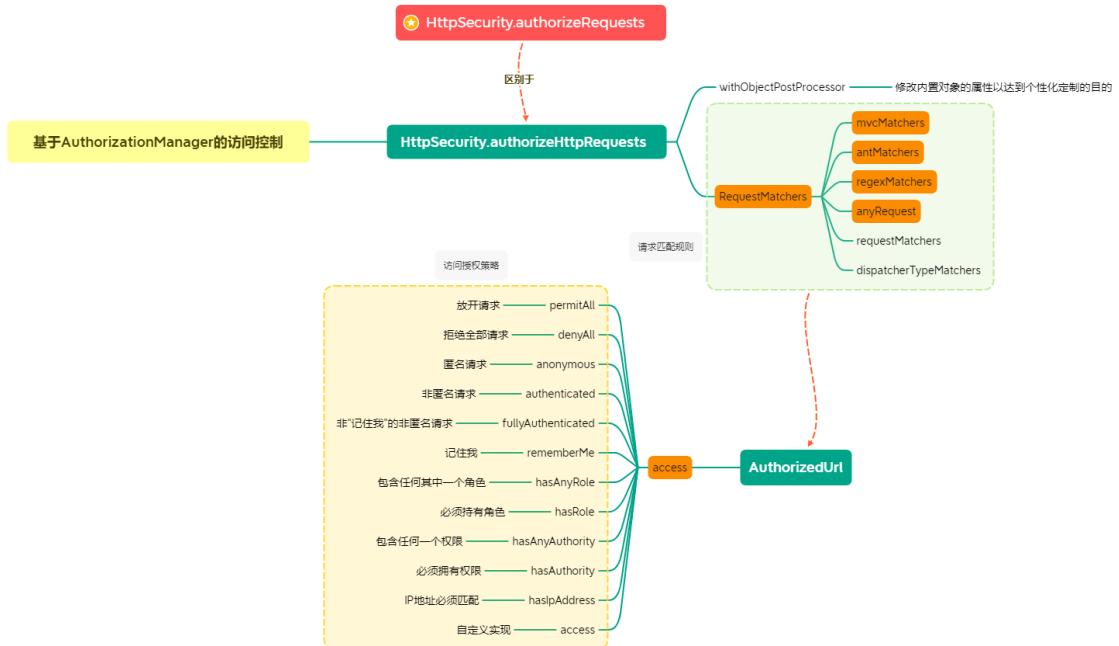
```
1  @FunctionalInterface
2  public interface AuthorizationManager<T> {
3
4      default void verify(Supplier<Authentication> authentication, T object) {
5          AuthorizationDecision decision = check(authentication, object);
6          // 授权决策没有经过允许就403
7          if (decision != null && !decision.isGranted()) {
8              throw new AccessDeniedException("Access Denied");
9          }
10         // todo 没有null 的情况
11     }
12
13     // 钩子方法。
14     @Nullable
15     AuthorizationDecision check(Supplier<Authentication> authentication, T
16     object);
17 }
```

我们只需要实现钩子方法 `check` 就可以了，它将当前提供的认证信息 `authentication` 和泛化对象 `T` 进行权限检查并返回授权的最终决定 `AuthorizationDecision`，

`AuthorizationDecision.isGranted` 将决定是否能够访问当前资源。`AuthorizationManager` 提供了两种使用方式。

基于配置

为了使用 `AuthorizationManager`，引入了相关配置是 `AuthorizeHttpRequestsConfigurer`，这个配置类非常类似于第九章中的基于表达式的访问控制。



在Spring Security 5.5中，我们就可以这样去实现了：

```
1  // 注意和 httpSecurity.authorizeRequests的区别
```

```

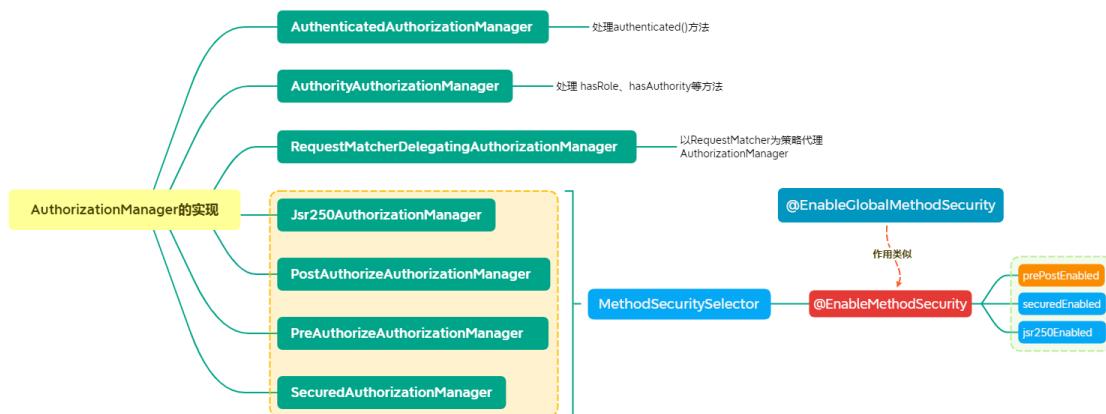
2         httpSecurity.authorizeHttpRequests()
3             .anyRequest()
4             .access((authenticationSupplier, requestAuthorizationContext) ->
5                 {
6                     // 当前用户的权限信息 比如角色
7                     Collection<? extends GrantedAuthority> authorities =
8                         authenticationSupplier.get().getAuthorities();
9                     // 当前请求上下文
10                    // 我们可以获取携带的参数
11                    Map<String, String> variables =
12                         requestAuthorizationContext.getVariables();
13                     // 我们可以获取原始request对象
14                     HttpServletRequest request =
15                         requestAuthorizationContext.getRequest();
16                     //todo 根据这些信息 和业务写逻辑即可 最终决定是否授权 isGranted
17                     boolean isGranted = true;
18                     return new AuthorizationDecision(isGranted);
19                 });

```

这样门槛是不是低多了呢？同样地，它也可以作用于注解。

基于注解

`AuthorizationManager` 还提供了基于注解的使用方式。但是在了解这种方式之前我们先来看看它的实现类关系：



胖哥发现这一点也是从 `AuthorizationManager` 的实现中倒推出来的，最终发现了 `@EnableMethodSecurity` 这个注解，它的用法和 `@EnableGlobalMethodSecurity` 类似，对同样的三种注解(参见 `EnableGlobalMethodSecurity`)进行了支持。用法也几乎一样，开启注解即可使用：

```

1  @EnableMethodSecurity(
2      securedEnabled = true,
3      jsr250Enabled = true)
4  public class MethodSecurityConfig {
5
6  }

```

例子就不在这里重复了。

这个是Spring Security 5.6版本的新玩法，默认支持 `prePostEnabled`。



微信搜一搜

Q 码农小胖哥

Spring Security OAuth2.0

重构中，内容涵盖**Spring Authorization Server**、**Spring Security OAuth2.0**、**Spring Cloud Security实践**，预计3月发布。请关注公众号：**码农小胖哥** 第一时间获取发布信息。