# Spring security

# 一、Spring security框架简介

### 1、简介

一个能够为基于Spring的企业应用系统提供声明式的安全訪问控制解决方式的安全框架（简单说是对访问权限进行控制嘛），应用的安全性包括用户认证（Authentication）和用户授权（Authorization）两个部分。用户认证指的是验证某个用户是否为系统中的合法主体，也就是说用户能否访问该系统。用户认证一般要求用户提供用户名和密码。系统通过校验用户名和密码来完成认证过程。用户授权指的是验证某个用户是否有权限执行某个操作。在一个系统中，不同用户所具有的权限是不同的。比如对一个文件来说，有的用户只能进行读取，而有的用户可以进行修改。一般来说，系统会为不同的用户分配不同的角色，而每个角色则对应一系列的权限。   spring security的主要核心功能为 认证和授权，所有的架构也是基于这两个核心功能去实现的。

### 2、框架原理

众所周知 想要对对Web资源进行保护，最好的办法莫过于Filter，要想对方法调用进行保护，最好的办法莫过于AOP。所以springSecurity在我们进行用户认证以及授予权限的时候，通过各种各样的拦截器来控制权限的访问，从而实现安全。如下为其主要过滤器:

        WebAsyncManagerIntegrationFilter

        SecurityContextPersistenceFilter

        HeaderWriterFilter

        CorsFilter

        LogoutFilter

        RequestCacheAwareFilter

        SecurityContextHolderAwareRequestFilter

        AnonymousAuthenticationFilter

        SessionManagementFilter

        ExceptionTranslationFilter

        FilterSecurityInterceptor

        UsernamePasswordAuthenticationFilter

        BasicAuthenticationFilter

### 框架的核心组件

SecurityContextHolder：提供对SecurityContext的访问

      SecurityContext,：持有Authentication对象和其他可能需要的信息

      AuthenticationManager 其中可以包含多个AuthenticationProvider

      ProviderManager对象为AuthenticationManager接口的实现类

      AuthenticationProvider 主要用来进行认证操作的类 调用其中的authenticate()方法去进行认证操作

      Authentication：Spring Security方式的认证主体

      GrantedAuthority：对认证主题的应用层面的授权，含当前用户的权限信息，通常使用角色表示

     UserDetails：构建Authentication对象必须的信息，可以自定义，可能需要访问DB得到

      UserDetailsService：通过username构建UserDetails对象，通过loadUserByUsername根据userName获取UserDetail对象 （可以在这里基于自身业务进行自定义的实现  如通过数据库，xml,缓存获取等）

## 二、自定义安全配置的加载机制

### 1、前提 基于自身业务需要

有关springSecrity安全框架的理解参考：springSecurity安全框架介绍

自定义了一个springSecurity安全框架的配置类 继承WebSecurityConfigurerAdapter，重写其中的方法configure，但是并不清楚自定义的类是如何被加载并起到作用，这里一步步通过debug来了解其中的加载原理。

其实在我们实现该类后，在web容器启动的过程中该类实例对象会被WebSecurityConfiguration类处理

@Configuration

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private AccessDeniedHandler accessDeniedHandler;

@Autowired

private CustAuthenticationProvider custAuthenticationProvider;

// roles admin allow to access /admin/\*\*

// roles user allow to access /user/\*\*

// custom 403 access denied handler

//重写了其中的configure（）方法设置了不同url的不同访问权限

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests()

.antMatchers("/home", "/about","/img/\*").permitAll()

.antMatchers("/admin/\*\*","/upload/\*\*").hasAnyRole("ADMIN")

.antMatchers("/order/\*\*").hasAnyRole("USER","ADMIN")

.antMatchers("/room/\*\*").hasAnyRole("USER","ADMIN")

.anyRequest().authenticated()

.and()

.formLogin()

.loginPage("/login")

.permitAll()

.and()

.logout()

.permitAll()

.and()

.exceptionHandling().accessDeniedHandler(accessDeniedHandler);

}

// create two users, admin and user

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

// auth.inMemoryAuthentication()

// .withUser("user").password("user").roles("USER")

// .and()

// .withUser("admin").password("admin").roles("ADMIN");

// auth.jdbcAuthentication()

auth.authenticationProvider(custAuthenticationProvider);

}

### 2、WebSecurityConfiguration类

@Configuration

public class WebSecurityConfiguration implements ImportAware, BeanClassLoaderAware {

private WebSecurity webSecurity;

private Boolean debugEnabled;

private List<SecurityConfigurer<Filter, WebSecurity>> webSecurityConfigurers;

private ClassLoader beanClassLoader;

...省略部分代码

@Bean(

name = {"springSecurityFilterChain"}

)

public Filter springSecurityFilterChain() throws Exception {

boolean hasConfigurers = this.webSecurityConfigurers != null

&& !this.webSecurityConfigurers.isEmpty();

if(!hasConfigurers) {

WebSecurityConfigurerAdapter adapter = (WebSecurityConfigurerAdapter)

this.objectObjectPostProcessor

.postProcess(new WebSecurityConfigurerAdapter() {

});

this.webSecurity.apply(adapter);

}

return (Filter)this.webSecurity.build();

}

/\*1、先执行该方法将我们自定义springSecurity配置实例

（可能还有系统默认的有关安全的配置实例 ） 配置实例中含有我们自定义业务的权限控制配置信息

放入到该对象的list数组中webSecurityConfigurers中

使用@Value注解来将实例对象作为形参注入

\*/

@Autowired(

required = false

)

public void setFilterChainProxySecurityConfigurer(ObjectPostProcessor<Object>

objectPostProcessor,

@Value("#{@autowiredWebSecurityConfigurersIgnoreParents.getWebSecurityConfigurers()}")

List<SecurityConfigurer<Filter, WebSecurity>> webSecurityConfigurers)

throws Exception {

//创建一个webSecurity对象

this.webSecurity = (WebSecurity)objectPostProcessor.postProcess(new WebSecurity(objectPostProcessor));

if(this.debugEnabled != null) {

this.webSecurity.debug(this.debugEnabled.booleanValue());

}

//对所有配置类的实例进行排序

Collections.sort(webSecurityConfigurers, WebSecurityConfiguration.AnnotationAwareOrderComparator.INSTANCE);

Integer previousOrder = null;

Object previousConfig = null;

//迭代所有配置类的实例 判断其order必须唯一

Iterator var5;

SecurityConfigurer config;

for(var5 = webSecurityConfigurers.iterator(); var5.hasNext(); previousConfig = config) {

config = (SecurityConfigurer)var5.next();

Integer order = Integer.valueOf(WebSecurityConfiguration.AnnotationAwareOrderComparator.lookupOrder(config));

if(previousOrder != null && previousOrder.equals(order)) {

throw new IllegalStateException("@Order on WebSecurityConfigurers must be unique. Order of " + order + " was already used on " + previousConfig + ", so it cannot be used on " + config + " too.");

}

previousOrder = order;

}

//将所有的配置实例添加到创建的webSecutity对象中

var5 = webSecurityConfigurers.iterator();

while(var5.hasNext()) {

config = (SecurityConfigurer)var5.next();

this.webSecurity.apply(config);

}

//将webSercurityConfigures 实例放入该对象的webSecurityConfigurers属性中

this.webSecurityConfigurers = webSecurityConfigurers;

}

}

2.1、 setFilterChainProxySecurityConfigurer（）方法

@Value("#{@autowiredWebSecurityConfigurersIgnoreParents.getWebSecurityConfigurers()}") List<SecurityConfigurer<Filter, WebSecurity>> webSecurityConfigurers

   该参数webSecurityConfigurers会将所有的配置实例放入该形参中

## 该方法中 主要执行如下

## 1、创建webSecurity对象

## 2、主要检验了配置实例的order顺序（order唯一 否则会报错）

## 3、将所有的配置实例存放进入到webSecurity对象中，其中配置实例中含有我们自定义业务的权限控制配置信息

## 2.2、springSecurityFilterChain()方法

## 调用springSecurityFilterChain()方法，这个方法会判断我们上一个方法中有没有获取到webSecurityConfigurers，没有的话这边会创建一个WebSecurityConfigurerAdapter实例，并追加到websecurity中。接着调用websecurity的build方法。实际调用的是websecurity的父类AbstractSecurityBuilder的build方法 ，最终返回一个名称为springSecurityFilterChain的过滤器链。里面有众多Filter(springSecurity其实就是依靠很多的Filter来拦截url从而实现权限的控制的安全框架)

### 3、AbstractSecurityBuilder类

public abstract class AbstractSecurityBuilder<O> implements SecurityBuilder<O> {

private AtomicBoolean building = new AtomicBoolean();

private O object;

//调用build方法来返回过滤器链，还是调用SecurityBuilder的dobuild()方法

public final O build() throws Exception {

if(this.building.compareAndSet(false, true)) {

this.object = this.doBuild();

return this.object;

} else {

throw new AlreadyBuiltException("This object has already been built");

}

}

//...省略部分代码

}

3.1 调用子类的doBuild()方法

public abstract class AbstractConfiguredSecurityBuilder<O, B extends SecurityBuilder<O>> extends AbstractSecurityBuilder<O> {

private final Log logger;

private final LinkedHashMap<Class<? extends SecurityConfigurer<O, B>>, List<SecurityConfigurer<O, B>>> configurers;

private final List<SecurityConfigurer<O, B>> configurersAddedInInitializing;

private final Map<Class<? extends Object>, Object> sharedObjects;

private final boolean allowConfigurersOfSameType;

private AbstractConfiguredSecurityBuilder.BuildState buildState;

private ObjectPostProcessor<Object> objectPostProcessor;

//doBuild()核心方法 init(),configure(),perFormBuild()

protected final O doBuild() throws Exception {

LinkedHashMap var1 = this.configurers;

synchronized(this.configurers) {

this.buildState = AbstractConfiguredSecurityBuilder.BuildState.INITIALIZING;

this.beforeInit();

this.init();

this.buildState = AbstractConfiguredSecurityBuilder.BuildState.CONFIGURING;

this.beforeConfigure();

this.configure();

this.buildState = AbstractConfiguredSecurityBuilder.BuildState.BUILDING;

O result = this.performBuild();

this.buildState = AbstractConfiguredSecurityBuilder.BuildState.BUILT;

return result;

}

}

protected abstract O performBuild() throws Exception;

//调用init方法 调用配置类WebSecurityConfigurerAdapter的init()方法

private void init() throws Exception {

Collection<SecurityConfigurer<O, B>> configurers = this.getConfigurers();

Iterator var2 = configurers.iterator();

SecurityConfigurer configurer;

while(var2.hasNext()) {

configurer = (SecurityConfigurer)var2.next();

configurer.init(this);

}

var2 = this.configurersAddedInInitializing.iterator();

while(var2.hasNext()) {

configurer = (SecurityConfigurer)var2.next();

configurer.init(this);

}

}

private void configure() throws Exception {

Collection<SecurityConfigurer<O, B>> configurers = this.getConfigurers();

Iterator var2 = configurers.iterator();

while(var2.hasNext()) {

SecurityConfigurer<O, B> configurer = (SecurityConfigurer)var2.next();

configurer.configure(this);

}

}

private Collection<SecurityConfigurer<O, B>> getConfigurers() {

List<SecurityConfigurer<O, B>> result = new ArrayList();

Iterator var2 = this.configurers.values().iterator();

while(var2.hasNext()) {

List<SecurityConfigurer<O, B>> configs = (List)var2.next();

result.addAll(configs);

}

return result;

}

//...省略部分代码

}

3.2 先调用本类的init()方法

build过程主要分三步，init->configure->peformBuild

1  init方法做了两件事，一个就是调用getHttp()方法获取一个http实例，并通过web.addSecurityFilterChainBuilder方法把获取到的实例赋值给WebSecurity的securityFilterChainBuilders属性，这个属性在我们执行build的时候会用到，第二个就是为WebSecurity追加了一个postBuildAction，在build都完成后从http中拿出FilterSecurityInterceptor对象并赋值给WebSecurity。

2  getHttp()方法，这个方法在当我们使用默认配置时（大多数情况下）会为我们追加各种SecurityConfigurer的具体实现类到httpSecurity中，如exceptionHandling()方法会追加一个ExceptionHandlingConfigurer，sessionManagement()方法会追加一个SessionManagementConfigurer,securityContext()方法会追加一个SecurityContextConfigurer对象，这些SecurityConfigurer的具体实现类最终会为我们配置各种具体的filter。

3 另外getHttp()方法的最后会调用configure(http)，这个方法也是我们继承WebSecurityConfigurerAdapter类后最可能会重写的方法 。

4 configure(HttpSecurity http)方法，默认的configure(HttpSecurity http)方法继续向httpSecurity类中追加SecurityConfigurer的具体实现类，如authorizeRequests()方法追加一个ExpressionUrlAuthorizationConfigurer，formLogin()方法追加一个FormLoginConfigurer。 其中ExpressionUrlAuthorizationConfigurer这个实现类比较重要，因为他会给我们创建一个非常重要的对象FilterSecurityInterceptor对象，FormLoginConfigurer对象比较简单，但是也会为我们提供一个在安全认证过程中经常用到会用的一个Filter：UsernamePasswordAuthenticationFilter。

以上三个方法就是WebSecurityConfigurerAdapter类中init方法的主要逻辑，

public abstract class WebSecurityConfigurerAdapter implements

WebSecurityConfigurer<WebSecurity> {

public void init(final WebSecurity web) throws Exception {

final HttpSecurity http = this.getHttp();

web.addSecurityFilterChainBuilder(http).postBuildAction(new Runnable() {

public void run() {

FilterSecurityInterceptor securityInterceptor = (FilterSecurityInterceptor)http.getSharedObject(FilterSecurityInterceptor.class);

web.securityInterceptor(securityInterceptor);

}

});

}

protected final HttpSecurity getHttp() throws Exception {

if(this.http != null) {

return this.http;

} else {

DefaultAuthenticationEventPublisher eventPublisher = (DefaultAuthenticationEventPublisher)this.objectPostProcessor.postProcess(new DefaultAuthenticationEventPublisher());

//添加认证的事件的发布者

this.localConfigureAuthenticationBldr.authenticationEventPublisher(eventPublisher);

//获取AuthenticationManager对象其中一至多个进行认证处理的对象实例，后面会进行讲解

AuthenticationManager authenticationManager = this.authenticationManager();

this.authenticationBuilder.parentAuthenticationManager(authenticationManager);

Map<Class<? extends Object>, Object> sharedObjects = this.createSharedObjects();

this.http = new HttpSecurity(this.objectPostProcessor, this.authenticationBuilder, sharedObjects);

if(!this.disableDefaults) {

((HttpSecurity)((DefaultLoginPageConfigurer)((HttpSecurity)((HttpSecurity)((HttpSecurity)((HttpSecurity)((HttpSecurity)((HttpSecurity)((HttpSecurity)((HttpSecurity)this.http.csrf().and()).addFilter(new WebAsyncManagerIntegrationFilter()).exceptionHandling().and()).headers().and()).sessionManagement().and()).securityContext().and()).requestCache().and()).anonymous().and()).servletApi().and()).apply(new DefaultLoginPageConfigurer())).and()).logout();

ClassLoader classLoader = this.context.getClassLoader();

List<AbstractHttpConfigurer> defaultHttpConfigurers = SpringFactoriesLoader.loadFactories(AbstractHttpConfigurer.class, classLoader);

Iterator var6 = defaultHttpConfigurers.iterator();

while(var6.hasNext()) {

AbstractHttpConfigurer configurer = (AbstractHttpConfigurer)var6.next();

this.http.apply(configurer);

}

}

//最终调用我们的继承的WebSecurityConfigurerAdapter中重写的configure()

//将我们业务相关的权限配置规则信息进行初始化操作

this.configure(this.http);

return this.http;

}

}

protected AuthenticationManager authenticationManager() throws Exception {

if(!this.authenticationManagerInitialized) {

this.configure(this.localConfigureAuthenticationBldr);

if(this.disableLocalConfigureAuthenticationBldr) {

this.authenticationManager = this.authenticationConfiguration.getAuthenticationManager();

} else {

this.authenticationManager = (AuthenticationManager)this.localConfigureAuthenticationBldr.build();

}

this.authenticationManagerInitialized = true;

}

return this.authenticationManager;

}

}

3.3、第二步configure

configure方法最终也调用到了WebSecurityConfigurerAdapter的configure(WebSecurity web)方法，默认实现中这个是一个空方法，具体应用中也经常重写这个方法来实现特定需求。

3.4、第三步 peformBuild

具体的实现逻辑在WebSecurity类中

这个方法中最主要的任务就是遍历securityFilterChainBuilders属性中的SecurityBuilder对象，并调用他的build方法。

这个securityFilterChainBuilders属性我们前面也有提到过，就是在WebSecurityConfigurerAdapter类的init方法中获取http后赋值给了WebSecurity。因此这个地方就是调用httpSecurity的build方法。

 httpSecurity的build方式向其中追加一个个过滤器

public final class WebSecurity extends AbstractConfiguredSecurityBuilder<Filter, WebSecurity> implements SecurityBuilder<Filter>, ApplicationContextAware {

...省略部分代码

//调用该方法通过securityFilterChainBuilder.build()方法来创建securityFilter过滤器

//并添加到securityFilterChains对象中，包装成FilterChainProxy 返回

protected Filter performBuild() throws Exception {

Assert.state(!this.securityFilterChainBuilders.isEmpty(), "At least one SecurityBuilder<? extends SecurityFilterChain> needs to be specified. Typically this done by adding a @Configuration that extends WebSecurityConfigurerAdapter. More advanced users can invoke " + WebSecurity.class.getSimpleName() + ".addSecurityFilterChainBuilder directly");

int chainSize = this.ignoredRequests.size() + this.securityFilterChainBuilders.size();

List<SecurityFilterChain> securityFilterChains = new ArrayList(chainSize);

Iterator var3 = this.ignoredRequests.iterator();

while(var3.hasNext()) {

RequestMatcher ignoredRequest = (RequestMatcher)var3.next();

securityFilterChains.add(new DefaultSecurityFilterChain(ignoredRequest, new Filter[0]));

}

var3 = this.securityFilterChainBuilders.iterator();

while(var3.hasNext()) {

SecurityBuilder<? extends SecurityFilterChain> securityFilterChainBuilder = (SecurityBuilder)var3.next();

securityFilterChains.add(securityFilterChainBuilder.build());

}

FilterChainProxy filterChainProxy = new FilterChainProxy(securityFilterChains);

if(this.httpFirewall != null) {

filterChainProxy.setFirewall(this.httpFirewall);

}

filterChainProxy.afterPropertiesSet();

Filter result = filterChainProxy;

if(this.debugEnabled) {

this.logger.warn("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\*\*\*\*\*\*\*\*\*\* Security debugging is enabled. \*\*\*\*\*\*\*\*\*\*\*\*\*\n\*\*\*\*\*\*\*\*\*\* This may include sensitive information. \*\*\*\*\*\*\*\*\*\*\*\*\*\n\*\*\*\*\*\*\*\*\*\* Do not use in a production system! \*\*\*\*\*\*\*\*\*\*\*\*\*\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

result = new DebugFilter(filterChainProxy);

}

this.postBuildAction.run();

return (Filter)result;

}

}

### 4、举例说明如何将一个Configurer转换为filter

ExpressionUrlAuthorizationConfigurer的继承关系

ExpressionUrlAuthorizationConfigurer->AbstractInterceptUrlConfigurer->AbstractHttpConfigurer->SecurityConfigurerAdapter->SecurityConfigurer

对应的init方法在SecurityConfigurerAdapter类中，是个空实现，什么也没有做，configure方法在SecurityConfigurerAdapter类中也有一个空实现，在AbstractInterceptUrlConfigurer类中进行了重写

Abstractintercepturlconfigurer.java代码

@Override

    public void configure(H http) throws Exception {

        FilterInvocationSecurityMetadataSource metadataSource = createMetadataSource(http);

        if (metadataSource == null) {

            return;

        }

        FilterSecurityInterceptor securityInterceptor = createFilterSecurityInterceptor(

                http, metadataSource, http.getSharedObject(AuthenticationManager.class));

        if (filterSecurityInterceptorOncePerRequest != null) {

            securityInterceptor

                    .setObserveOncePerRequest(filterSecurityInterceptorOncePerRequest);

        }

        securityInterceptor = postProcess(securityInterceptor);

        http.addFilter(securityInterceptor);

        http.setSharedObject(FilterSecurityInterceptor.class, securityInterceptor);

    }

...

private AccessDecisionManager createDefaultAccessDecisionManager(H http) {

        AffirmativeBased result = new AffirmativeBased(getDecisionVoters(http));

        return postProcess(result);

    }

...

private FilterSecurityInterceptor createFilterSecurityInterceptor(H http,

            FilterInvocationSecurityMetadataSource metadataSource,

            AuthenticationManager authenticationManager) throws Exception {

        FilterSecurityInterceptor securityInterceptor = new FilterSecurityInterceptor();

        securityInterceptor.setSecurityMetadataSource(metadataSource);

        securityInterceptor.setAccessDecisionManager(getAccessDecisionManager(http));

        securityInterceptor.setAuthenticationManager(authenticationManager);

        securityInterceptor.afterPropertiesSet();

        return securityInterceptor;

    }

4.1、 在这个类的configure中创建了一个FilterSecurityInterceptor，并且也可以明确看到spring security默认给我们创建的AccessDecisionManager是AffirmativeBased。

4.2、.最后再看下HttpSecurity类执行build的最后一步 performBuild，这个方法就是在HttpSecurity中实现的

Httpsecurity.java代码

@Override

    protected DefaultSecurityFilterChain performBuild() throws Exception {

        Collections.sort(filters, comparator);

        return new DefaultSecurityFilterChain(requestMatcher, filters);

    }

可以看到，这个类只是把我们追加到HttpSecurity中的security进行了排序，用的排序类是FilterComparator，从而保证我们的filter按照正确的顺序执行。接着将filters构建成filterChian返回。在前面WebSecurity的performBuild方法中，这个返回值会被包装成FilterChainProxy，并作为WebSecurity的build方法的放回值。从而以springSecurityFilterChain这个名称注册到springContext中（在WebSecurityConfiguration中做的）

4.3.在WebSecurity的performBuild方法的最后一步还执行了一个postBuildAction.run，这个方法也是spring security给我们提供的一个hooks，可以在build完成后再做一些事情，比如我们在WebSecurityConfigurerAdapter类的init方法中我们利用这个hook在构建完成后将FilterSecurityInterceptor赋值给了webSecurity类的filterSecurityInterceptor属性

## 三、用户登录的验证和授权信息

1、用户一次完整的登录验证和授权，是一个请求经过 层层拦截器从而实现权限控制，整个web端配置为DelegatingFilterProxy（springSecurity的委托过滤其代理类 ），它并不实现真正的过滤，而是所有过滤器链的代理类，真正执行拦截处理的是由spring 容器管理的个个filter bean组成的filterChain.

调用实际的FilterChainProxy 的doFilterInternal()方法 去获取所有的拦截器并进行过滤处理如下是DelegatingFilterProxy的doFilter（）方法

public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) throws ServletException, IOException {

Filter delegateToUse = this.delegate;

if(delegateToUse == null) {

Object var5 = this.delegateMonitor;

synchronized(this.delegateMonitor) {

delegateToUse = this.delegate;

if(delegateToUse == null) {

WebApplicationContext wac = this.findWebApplicationContext();

if(wac == null) {

throw new IllegalStateException("No WebApplicationContext found: no ContextLoaderListener or DispatcherServlet registered?");

}

delegateToUse = this.initDelegate(wac);

}

this.delegate = delegateToUse;

}

}

//调用实际的FilterChainProxy 的doFilterInternal()方法 去获取所有的拦截器并进行过滤处理

this.invokeDelegate(delegateToUse, request, response, filterChain);

}

调用实际的FilterChainProxy 的doFilter()方法 去获取所有的拦截器并进行过滤处理。

2、FilterChainProxy类

    最终调用FilterChainProxy 的doFilterInternal()方法，获取所有的过滤器实例

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

boolean clearContext = request.getAttribute(FILTER\_APPLIED) == null;

if(clearContext) {

try {

request.setAttribute(FILTER\_APPLIED, Boolean.TRUE);

//doFilter 调用doFilterInternal方法

this.doFilterInternal(request, response, chain);

} finally {

SecurityContextHolder.clearContext();

request.removeAttribute(FILTER\_APPLIED);

}

} else {

this.doFilterInternal(request, response, chain);

}

}

private void doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

FirewalledRequest fwRequest = this.firewall.getFirewalledRequest((HttpServletRequest)request);

HttpServletResponse fwResponse = this.firewall.getFirewalledResponse((HttpServletResponse)response);

//过去所有的过滤器

List<Filter> filters = this.getFilters((HttpServletRequest)fwRequest);

if(filters != null && filters.size() != 0) {

FilterChainProxy.VirtualFilterChain vfc = new FilterChainProxy.VirtualFilterChain(fwRequest, chain, filters);

vfc.doFilter(fwRequest, fwResponse);

} else {

if(logger.isDebugEnabled()) {

logger.debug(UrlUtils.buildRequestUrl(fwRequest) + (filters == null?" has no matching filters":" has an empty filter list"));

}

fwRequest.reset();

chain.doFilter(fwRequest, fwResponse);

}

}

private List<Filter> getFilters(HttpServletRequest request) {

//遍历所有的matcher类 如果支持就继续获取

Iterator var2 = this.filterChains.iterator();

SecurityFilterChain chain;

do {

if(!var2.hasNext()) {

return null;

}

chain = (SecurityFilterChain)var2.next();

} while(!chain.matches(request));

//后去匹配中的所有过滤器

return chain.getFilters();

}