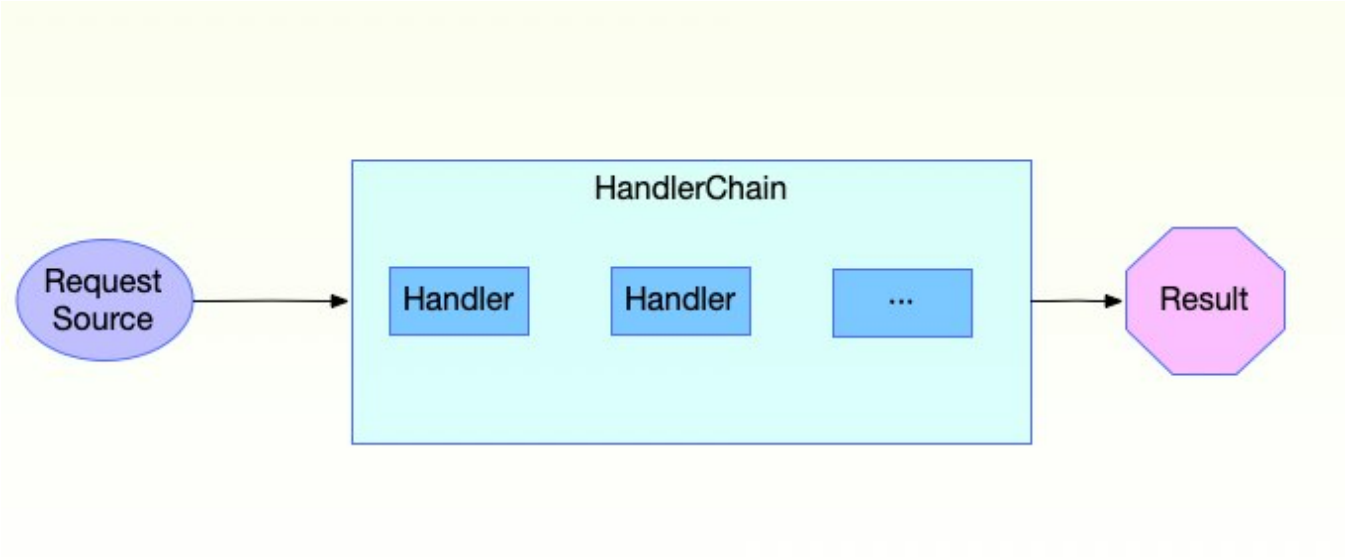
Spring Security 实战干货：内置 Filter 全解析

spring security (/categories/spring-security/)

java (/tags/java/) spring security (/tags/spring-security/) 2019/10/23 486

1. 前言

上一文我们使用 **Spring Security** 实现了各种登录聚合的场面。其中我们是通过在 `UsernamePasswordAuthenticationFilter` 之前一个自定义的过滤器实现的。我怎么知道自定义过滤器要加在 `UsernamePasswordAuthenticationFilter` 之前。我在这个系列开篇说了 **Spring Security** 权限控制的一个核心关键就是 **过滤器链**，这些过滤器如下图进行过滤传递，甚至比这个更复杂！这只是一个最小单元。



Spring Security 内置了一些过滤器，他们各有各的本事。如果你掌握了这些过滤器，很多实际开发中的需求和问题都很容易解决。今天我们来见识一下这些内置的过滤器。

2. 内置过滤器初始化

在 **Spring Security** 初始化核心过滤器时 `HttpSecurity` 会通过将 **Spring Security** 内置的一些过滤器以 `FilterComparator` 提供的规则进行比较按照比较结果进行排序注册。

2.1 排序规则

`FilterComparator` 维护了一个顺序的注册表 `filterToOrder` 。

```
    FilterComparator() {
        Step order = new Step(INITIAL_ORDER, ORDER_STEP);
        put(ChannelProcessingFilter.class, order.next());
        put(ConcurrentSessionFilter.class, order.next());
        put(WebAsyncManagerIntegrationFilter.class, order.next
        put(SecurityContextPersistenceFilter.class, order.next
        put(HeaderWriterFilter.class, order.next());
        put(CorsFilter.class, order.next());
        put(CsrfFilter.class, order.next());
        put(LogoutFilter.class, order.next());
        filterToOrder.put(
            "org.springframework.security.oauth2.client.web.OA
                order.next());
        filterToOrder.put(
                "org.springframework.security.saml2.provider.s
                order.next());
        put(X509AuthenticationFilter.class, order.next());
        put(AbstractPreAuthenticatedProcessingFilter.class, or
        filterToOrder.put("org.springframework.security.cas.we
                order.next());
        filterToOrder.put(
            "org.springframework.security.oauth2.client.web.OA
                order.next());
        filterToOrder.put(
                "org.springframework.security.saml2.provider.s
                order.next());
        put(UsernamePasswordAuthenticationFilter.class, order
        put(ConcurrentSessionFilter.class, order.next());
        filterToOrder.put(
                "org.springframework.security.openid.OpenIDAut
        put(DefaultLoginPageGeneratingFilter.class, order.next
        put(DefaultLogoutPageGeneratingFilter.class, order.nex
        put(ConcurrentSessionFilter.class, order.next());
        put(DigestAuthenticationFilter.class, order.next());
        filterToOrder.put(
                "org.springframework.security.oauth2.server.re
        put(BasicAuthenticationFilter.class, order.next());
        put(RequestCacheAwareFilter.class, order.next());
        put(SecurityContextHolderAwareRequestFilter.class, or
        put(JaasApiIntegrationFilter.class, order.next());
```

```
            put(RememberMeAuthenticationFilter.class, order.next()
            put(AnonymousAuthenticationFilter.class, order.next()
            filterToOrder.put(
                "org.springframework.security.oauth2.client.web.OA
                    order.next());
            put(SessionManagementFilter.class, order.next());
            put(ExceptionTranslationFilter.class, order.next());
            put(FilterSecurityInterceptor.class, order.next());
            put(SwitchUserFilter.class, order.next());
        }
```

这些就是所有内置的过滤器。 他们是通过下面的方法获取自己的序号:

```
        private Integer getOrder(Class<?> clazz) {
            while (clazz != null) {
                Integer result = filterToOrder.get(clazz.getName()
                if (result != null) {
                    return result;
                }
                clazz = clazz.getSuperclass();
            }
            return null;
        }
```

通过过滤器的类全限定名从注册表 `filterToOrder` 中获取自己的序号，如果没有直接获取到序号通过递归获取父类在注册表中的序号作为自己的序号，序号越小优先级越高。上面的过滤器并非全部会被初始化。有的需要额外引入一些功能包，有的看 `HttpSecurity` 的配置情况。
在上一篇文章 (https://www.felord.cn/spring-security-login.html)中。我们禁用了 `CSRF` 功能，就意味着 `CsrfFilter` 不会被注册。

3. 内置过滤器讲解

接下来我们就对这些内置过滤器进行一个系统的认识。**我们将按照默认顺序进行讲解。**

3.1 ChannelProcessingFilter

`ChannelProcessingFilter` 通常是用来过滤哪些请求必须用 `https` 协议， 哪些请求必须用 `http` 协议， 哪些请求随便用哪个协议都行。它主要有两个属性:

- `ChannelDecisionManager` 用来判断请求是否符合既定的协议规则。它维护了一个 `ChannelProcessor` 列表 这些`ChannelProcessor` 是具体用来执行 `ANY_CHANNEL` 策略 （任何通道都可以），`REQUIRES_SECURE_CHANNEL` 策略 （只能通过`https` 通道），`REQUIRES_INSECURE_CHANNEL` 策略 （只能通过 `http` 通道）。
- `FilterInvocationSecurityMetadataSource` 用来存储 url 与 对应的`ANY_CHANNEL` 、`REQUIRES_SECURE_CHANNEL`、`REQUIRES_INSECURE_CHANNEL` 的映射关系。

`ChannelProcessingFilter` 通过 `HttpScurity#requiresChannel()` 等相关方法引入其配置对象 `ChannelSecurityConfigurer` 来进行配置。

3.2 ConcurrentSessionFilter

`ConcurrentSessionFilter` 主要用来判断`session`是否过期以及更新最新的访问时间。其流程为：

1. `session` 检测，如果不存在直接放行去执行下一个过滤器。存在则进行下一步。
2. 根据`sessionid`从`SessionRegistry`中获取`SessionInformation`，
   从`SessionInformation`中获取`session`是否过期；没有过期则更新`SessionInformation`中的访问日期；
   如果过期，则执行`doLogout()`方法，这个方法会将`session`无效，并将 `SecurityContext` 中的`Authentication`中的权限置空，同时在`SecurityContenxtHoloder`中清除`SecurityContext`然后查看是否有跳转的 `expiredUrl`，如果有就跳转，没有就输出提示信息。

`ConcurrentSessionFilter` 通过`SessionManagementConfigurer` 来进行配置。

3.3 WebAsyncManagerIntegrationFilter

`WebAsyncManagerIntegrationFilter`用于集成SecurityContext到Spring异步执行机制中的WebAsyncManager。用来处理异步请求的安全上下文。具体逻辑为：

1. 从请求属性上获取所绑定的`WebAsyncManager`，如果尚未绑定，先做绑定。
2. 从`asyncManager` 中获取 `key` 为 `CALLABLE_INTERCEPTOR_KEY` 的安全上下文多线程处理器 `SecurityContextCallableProcessingInterceptor`，如果获取到的为 `null`，新建一个 `SecurityContextCallableProcessingInterceptor` 并绑定 `CALLABLE_INTERCEPTOR_KEY` 注册到 `asyncManager` 中。

这里简单说一下 `SecurityContextCallableProcessingInterceptor` 。它实现了接口 `CallableProcessingInterceptor`，
当它被应用于一次异步执行时， `beforeConcurrentHandling()` 方法会在调用者线程执行，

该方法会相应地从当前线程获取`SecurityContext`,然后被调用者线程中执行逻辑时，会使用这个 `SecurityContext`，从而实现安全上下文从调用者线程到被调用者线程的传输。

`WebAsyncManagerIntegrationFilter` 通过 `WebSecurityConfigurerAdapter#getHttp()`方法添加到 `HttpSecurity` 中成为 `DefaultSecurityFilterChain` 的一个链节。

3.4 SecurityContextPersistenceFilter

`SecurityContextPersistenceFilter` 主要控制 `SecurityContext` 的在一次请求中的生命周期 。 请求来临时，创建`SecurityContext` 安全上下文信息，请求结束时清空 `SecurityContextHolder`。

`SecurityContextPersistenceFilter` 通过 `HttpScurity#securityContext()` 及相关方法引入其配置对象 `SecurityContextConfigurer` 来进行配置。

3.5 HeaderWriterFilter

`HeaderWriterFilter` 用来给 `http` 响应添加一些 `Header`,比如 `X-Frame-Options`, `X-XSS-Protection` ， `X-Content-Type-Options`。

你可以通过 `HttpScurity#headers()` 来定制请求`Header` 。

3.6 CorsFilter

跨域相关的过滤器。这是`Spring MVC Java`配置和`XML` 命名空间 `CORS` 配置的替代方法，仅对依赖于`spring-web`的应用程序有用（不适用于`spring-webmvc`）或 要求在`javax.servlet.Filter` 级别进行CORS检查的安全约束链接。这个是目前官方的一些解读，但是我还是不太清楚实际机制。

你可以通过 `HttpSecurity#cors()` 来定制。

3.7 CsrfFilter

`CsrfFilter` 用于防止`csrf`攻击，前后端使用json交互需要注意的一个问题。

你可以通过 `HttpSecurity.csrf()` 来开启或者关闭它。在你使用 `jwt` 等 `token` 技术时，是不需要这个的。

3.8 LogoutFilter

`LogoutFilter` 很明显这是处理注销的过滤器。

你可以通过 `HttpSecurity.logout()` 来定制注销逻辑，非常有用。

3.9 OAuth2AuthorizationRequestRedirectFilter

和上面的有所不同，这个需要依赖 `spring-scurity-oauth2` 相关的模块。该过滤器是处理 `OAuth2` 请求首选重定向相关逻辑的。以后会我会带你们认识它，请多多关注公众号：`Felordcn` 。

3.10 Saml2WebSsoAuthenticationRequestFilter

这个需要用到 `Spring Security SAML` 模块，这是一个基于 `SMAL` 的 `SSO` 单点登录请求认证过滤器。

关于SAML

`SAML` 即安全断言标记语言，英文全称是 `Security Assertion Markup Language`。它是一个基于 `XML` 的标准，用于在不同的安全域（`security domain`）之间交换认证和授权数据。在 `SAML` 标准定义了身份提供者 (`identity provider`) 和服务提供者 (`service provider`)，这两者构成了前面所说的不同的安全域。 `SAML` 是 `OASIS` 组织安全服务技术委员会(**Security Services Technical Committee**) 的产品。

`SAML`（**Security Assertion Markup Language**）是一个 `XML` 框架，也就是一组协议，可以用来传输安全声明。比如，两台远程机器之间要通讯，为了保证安全，我们可以采用加密等措施，也可以采用 `SAML` 来传输，传输的数据以 `XML` 形式，符合 `SAML` 规范，这样我们就可以不要求两台机器采用什么样的系统，只要求能理解 `SAML` 规范即可，显然比传统的方式更好。`SAML` 规范是一组 `Schema` 定义。

可以这么说，在`Web Service` 领域， `schema` 就是规范，在 `Java` 领域， `API` 就是规范

3.11 X509AuthenticationFilter

`X509` 认证过滤器。你可以通过 `HttpSecurity#X509()` 来启用和配置相关功能。

3.12 AbstractPreAuthenticatedProcessingFilter

`AbstractPreAuthenticatedProcessingFilter` 处理处理经过预先认证的身份验证请求的过滤器的基类，其中认证主体已经由外部系统进行了身份验证。 目的只是从传入请求中提取主体上的必要信息，而不是对它们进行身份验证。

你可以继承该类进行具体实现并通过 `HttpSecurity#addFilter` 方法来添加个性化的`AbstractPreAuthenticatedProcessingFilter` 。

3.13 CasAuthenticationFilter

`CAS` 单点登录认证过滤器 。依赖 Spring Security CAS 模块

3.14 OAuth2LoginAuthenticationFilter

这个需要依赖 `spring-scurity-oauth2` 相关的模块。`OAuth2` 登录认证过滤器。处理通过 `OAuth2` 进行认证登录的逻辑。

3.15 Saml2WebSsoAuthenticationFilter

这个需要用到 `Spring Security SAML` 模块，这是一个基于 `SMAL` 的 `SSO` 单点登录认证过滤器。关于SAML

3.16 UsernamePasswordAuthenticationFilter

这个看过我相关文章的应该不陌生了。处理用户以及密码认证的核心过滤器。认证请求提交的`username`和 `password`，被封装成`token`进行一系列的认证，便是主要通过这个过滤器完成的，在表单认证的方法中，这是最最关键的过滤器。

你可以通过 `HttpSecurity#formLogin()` 及相关方法引入其配置对象 `FormLoginConfigurer` 来进行配置。 我们在 Spring Security 实战干货： 玩转自定义登录(https://www.felord.cn/spring-security-login.html) 已经对其进行过个性化的配置和魔改。

3.17 ConcurrentSessionFilter

参见 3.2 ConcurrentSessionFilter。 该过滤器可能会被多次执行。

3.18 OpenIDAuthenticationFilter

基于`OpenID` 认证协议的认证过滤器。 你需要在依赖中依赖额外的相关模块才能启用它。

3.19 DefaultLoginPageGeneratingFilter

生成默认的登录页。默认 `/login` 。

3.20 DefaultLogoutPageGeneratingFilter

生成默认的退出页。 默认 `/logout` 。

3.21 ConcurrentSessionFilter

参见 3.2 ConcurrentSessionFilter 。 该过滤器可能会被多次执行。

3.23 DigestAuthenticationFilter

`Digest`身份验证是 `Web` 应用程序中流行的可选的身份验证机制 。`DigestAuthenticationFilter` 能够处理 `HTTP` 头中显示的摘要式身份验证凭据。你可以通过 `HttpSecurity#addFilter()` 来启用和配置相关功能。

3.24 BasicAuthenticationFilter

和`Digest`身份验证一样都是`Web` 应用程序中流行的可选的身份验证机制 。`BasicAuthenticationFilter` 负责处理 `HTTP` 头中显示的基本身份验证凭据。这个 **Spring Security** 的 **Spring Boot** 自动配置默认是启用的 。

`BasicAuthenticationFilter` 通过 `HttpSecurity#httpBasic()` 及相关方法引入其配置对象 `HttpBasicConfigurer` 来进行配置。

3.25 RequestCacheAwareFilter

用于用户认证成功后，重新恢复因为登录被打断的请求。当匿名访问一个需要授权的资源时。会跳转到认证处理逻辑，此时请求被缓存。在认证逻辑处理完毕后，从缓存中获取最开始的资源请求进行再次请求。

`RequestCacheAwareFilter` 通过 `HttpScurity#requestCache()` 及相关方法引入其配置对象 `RequestCacheConfigurer` 来进行配置。

3.26 SecurityContextHolderAwareRequestFilter

用来 实现`j2ee`中 `Servlet Api` 一些接口方法, 比如 `getRemoteUser` 方法、`isUserInRole` 方法，在使用 **Spring Security** 时其实就是通过这个过滤器来实现的。

`SecurityContextHolderAwareRequestFilter` 通过 `HttpSecurity.servletApi()` 及相关方法引入其配置对象 `ServletApiConfigurer` 来进行配置。

3.27 JaasApiIntegrationFilter

适用于`JAAS` （`Java` 认证授权服务）。 如果 `SecurityContextHolder` 中拥有的 `Authentication` 是一个 `JaasAuthenticationToken`，那么该 `JaasApiIntegrationFilter` 将使用包含在 `JaasAuthenticationToken` 中的 `Subject` 继续执行 `FilterChain`。

3.28 RememberMeAuthenticationFilter

处理 **`记住我`** 功能的过滤器。

`RememberMeAuthenticationFilter` 通过 `HttpSecurity.rememberMe()` 及相关方法引入其配置对象 `RememberMeConfigurer` 来进行配置。

3.29 AnonymousAuthenticationFilter

匿名认证过滤器。**对于 `Spring Security` 来说，所有对资源的访问都是有 `Authentication` 的。对于无需登录（`UsernamePasswordAuthenticationFilter`）直接可以访问的资源，会授予其匿名用户身份。**

`AnonymousAuthenticationFilter` 通过 `HttpSecurity.anonymous()` 及相关方法引入其配置对象 `AnonymousConfigurer` 来进行配置。

3.30 SessionManagementFilter

`Session` 管理器过滤器，内部维护了一个 `SessionAuthenticationStrategy` 用于管理 `Session` 。

`SessionManagementFilter` 通过 `HttpScurity#sessionManagement()` 及相关方法引入其配置对象 `SessionManagementConfigurer` 来进行配置。

3.31 ExceptionTranslationFilter

主要来传输异常事件，还记得之前我们见过的 `DefaultAuthenticationEventPublisher` 吗?

3.32 FilterSecurityInterceptor

这个过滤器决定了访问特定路径应该具备的权限，访问的用户的角色，权限是什么？访问的路径需要什么样的角色和权限？这些判断和处理都是由该类进行的。**如果你要实现动态权限控制就必须研究该类** 。

3.33 SwitchUserFilter

`SwitchUserFilter` 是用来做账户切换的。默认的切换账号的`url`为`/login/impersonate`，默认注销切换账号的`url`为`/logout/impersonate`，默认的账号参数为`username` 。

你可以通过此类实现自定义的账户切换。

4. 总结

所有内置的 31个过滤器作用都讲解完了，**有一些默认已经启用。有一些需要引入特定的包并且对 `HttpSecurity` 进行配置才会生效 。而且它们的顺序是既定的。** 只有你了解这些过滤器你才能基于业务深度定制 **Spring Security** 。

‹ 上一篇 (/spring-security-logout.html)

下一篇 › (/spring-boot-condition.html)