# 基本概念

# spring security与spring security oauth2的关系

Spring Security 和 Spring Security OAuth2 是两个在 Spring 生态系统中常用的安全框架，它们可以一起使用来实现身份验证和授权的功能。

1、Spring Security 是一个用于在应用程序中处理身份验证（Authentication）和授权（Authorization）的框架。它提供了一套可扩展的认证和授权机制，可以用于保护 Spring Web 应用程序、REST API、方法级别的安全性等。Spring Security 基于 Servlet 过滤器，可以集成到任何支持 Servlet 规范的 Java Web 应用程序中。

# OAuth2.0

## 基本概念

[OAuth](http://en.wikipedia.org/wiki/OAuth)是一个关于授权（authorization）的开放网络标准，在全世界得到广泛应用，目前的版本是2.0版。

OAuth 2.0是一个**授权框架**，解决"如何让第三方应用安全访问用户资源"的问题。

## 发布时间

OAuth 2.0 标准是在**2012年10月**由**IETF（Internet Engineering Task Force）** 正式发布的，文档编号是 **RFC 6749**。

## 应用场景

为了理解OAuth的适用场合，让我举一个假设的例子。有一个"云冲印"的网站，可以将用户储存在Google的照片，冲印出来。用户为了使用该服务，必须让"云冲印"读取自己储存在Google上的照片。

问题是只有得到用户的授权，Google才会同意"云冲印"读取这些照片。那么，"云冲印"怎样获得用户的授权呢？传统方法是，用户将自己的Google用户名和密码告诉"云冲印"，后者就可以读取用户的照片了。这样的做法有以下几个严重的缺点：

（1）"云冲印"为了后续的服务，会保存用户的密码，这样很不安全。

（2）Google不得不部署密码登录，而我们知道，单纯的密码登录并不安全。

（3）"云冲印"拥有了获取用户储存在Google所有资料的权力，用户没法限制"云冲印"获得授权的范围和有效期。

（4）用户只有修改密码，才能收回赋予"云冲印"的权力。但是这样做会使得其他所有获得用户授权的第三方应用程序全部失效。

（5）只要有一个第三方应用程序被破解，就会导致用户密码泄漏，以及所有被密码保护的数据泄漏。

OAuth就是为了解决上面这些问题而诞生的。

## 名词定义

### 1、Third-party application

第三方应用程序（请求访问的应用），本文中又称为客户端，即上一节例子中的"云冲印"。

### 2、HTTP service

HTTP服务提供商，本文中简称"服务提供商"，即上一节例子中的Google。

### 3、Resource Owner

资源所有者，本文中又称"用户"（user）。

### 4、User Agent

用户代理，本文中就是指浏览器。

### 5、Authorization server

认证服务器，即服务提供商专门用来处理认证的服务器。

### 6、Resource server

资源服务器，即服务提供商存放用户生成的资源的服务器。它与认证服务器可以是同一台服务器，也可以是不同的服务器。

OAuth的作用就是让"客户端"安全可控地获取"用户"的授权，与"服务商提供商"进行互动。

## 认证服务器

### 基本概念

认证服务器是OAuth2协议中的核心组件之一，它是一个专门的服务器应用程序，负责：

1、验证用户身份

2、颁发访问令牌（Access Token）

3、管理客户端应用的注册信息。

4、处理授权请求。

### 认证服务器可以是哪些应用

#### 1. 第三方服务提供商

**Google OAuth服务器** - accounts.google.com

**微信开放平台** - open.weixin.qq.com

**GitHub OAuth** - github.com/login/oauth

**微博开放平台** - api.weibo.com

#### 2. 自建认证服务器

企业内部的统一认证中心

使用Spring Security OAuth2搭建的认证服务

使用Keycloak等开源认证服务器

自研的用户认证系统

### 核心功能

#### 1. 用户身份验证

用户 → 认证服务器 → 验证用户名密码/第三方登录

#### 2. 客户端验证

客户端应用 → 认证服务器 → 验证client\_id和client\_secret

#### 3. 授权码颁发

认证服务器 → 用户授权确认 → 颁发authorization\_code

#### 4. 访问令牌颁发

客户端+授权码 → 认证服务器 → 颁发access\_token

### 实际应用场景举例

#### 场景1：使用微信登录第三方应用

**认证服务器**：微信开放平台的OAuth服务器

**客户端**：第三方应用（如某个购物网站）

**资源服务器**：微信的用户信息API

**用户**：使用微信账号的用户

#### 场景2：企业内部系统

**认证服务器**：企业的统一认证中心（如基于Spring Security OAuth2）

**客户端**：各个业务系统（CRM、ERP等）

**资源服务器**：各个业务系统的API

**用户**：企业员工

### 总结

认证服务器是OAuth2体系中的"信任中心"，它可以是第三方服务（如Google、微信），也可以是企业自建的认证系统。它的核心职责是验证身份并颁发令牌，确保整个OAuth2流程的安全性。

## Oauth的思路

1、用户不给密码，给令牌。

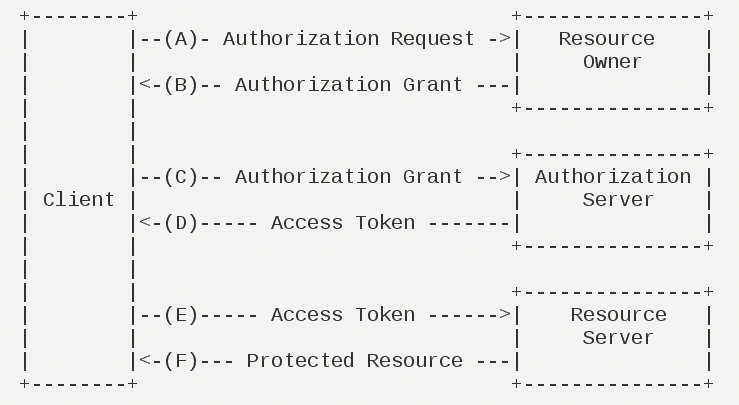
2、用户无需向第三方应用透露密码。

3、通过授权服务器颁发访问令牌。

4、第三方应用用令牌访问受保护资源。

## 运行流程

OAuth 2.0的运行流程如下图：



A：用户打开客户端之后，客户端要求用户给予授权。

B：用户同意给予客户端授权。

C：客户端使用上一步获得的授权，向认证服务器申请令牌。

D：认证服务器对客户端进行认证以后，确认无误，同意发放令牌。

E：客户端使用令牌，向资源服务器申请获取资源。

F：资源服务器确认令牌无误，同意向客户端开放资源。

不难看出来，上面六个步骤之中，B是关键，即用户怎样才能给于客户端授权，有了这个授权以后，客户端就可以获取令牌，进而凭令牌获取资源。

## 客户端的授权模式

客户端必须得到用户的授权（authorization grant）才能获得令牌（access token）。OAuth 2.0定义了四种授权方式。

（1）授权码模式（authorization code）

（2）简化模式（implicit）

（3）密码模式（resource owner password credentials）

（4）客户端模式（client credentials）

### 授权码模式（重点）

http://www.ruanyifeng.com/blog/2014/05/oauth\_2\_0.html

用户 → 授权页面 → 同意 → 返回code → 后端用code换token

## 实际示例

微信登录流程

1. 点击"微信登录" → 跳转微信授权页面

2. 用户确认授权 → 微信返回code

3. 应用后端用code换取access\_token

4. 用token获取用户信息

5. 完成登录

# 接口

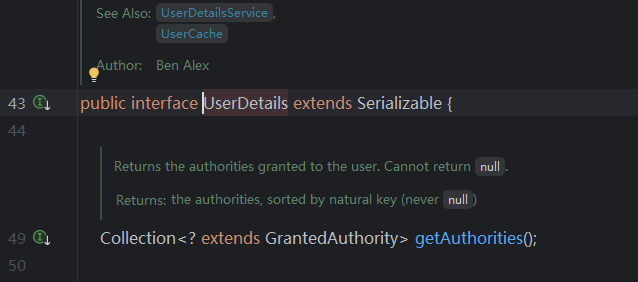
## GrantedAuthority

## UserDetails

### 基本概念

UserDetails 是 Spring Security 中用于描述用户信息的核心接口，它包含了认证和授权过程中所需的所有用户数据。

### 源码



### 方法

#### Collection<? extends GrantedAuthority> getAuthorities()

返回用户的权限集合，用于授权检查。



#### String getUsername()

获取用户名

#### String getPassword();

获取密码

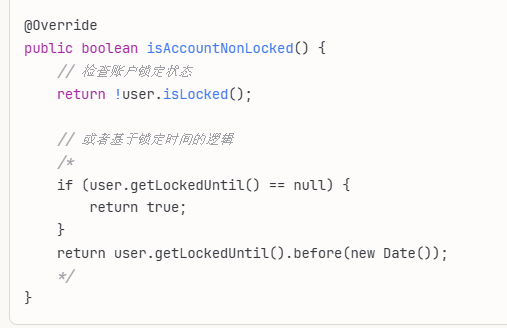
#### boolean isAccountNonExpired();

检查账户是否未过期



#### boolean isAccountNonLocked();

检查账户是否未被锁定



#### isCredentialsNonExpired()

检查凭证（密码）是否未过期



#### isEnabled()

检查账户是否已启用



### 实现方式

#### 1、简单实现





#### 2、基于实体类的实现









### Spring Security内置实现

#### 1、User类

Spring Security 提供了 User 类作为 UserDetails 的默认实现。



#### 2、使用UserBuilder



### 在认证过程中的使用

#### 1、登录成功后访问用户信息



#### 2、在服务中获取当前用户



### 总结

UserDetails 接口是 Spring Security 认证和授权的核心，它：

1、定义了用户的基本信息：用户名、密码、权限等

2、提供了账户状态检查：过期、锁定、启用状态等

3、支持灵活的权限模型：角色和权限的组合

4、可以扩展以满足特定需求：自定义字段和逻辑

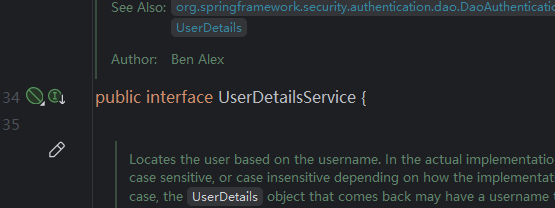
通过实现 UserDetails 接口，你可以将任何用户数据模型与 Spring Security 集成，实现完整的认证和授权功能。

## UserDetailsService

### 基本概念

UserDetailsService 是 Spring Security 中最重要的接口之一，用于从数据源（如数据库、LDAP、内存等）加载用户特定的数据。UserDetailsService 接口定义了一个方法，用于在认证过程中根据用户名检索用户信息。

### 源码



### 方法

#### UserDetails loadUserByUsername(String username)

根据用户名查询用户详情信息

### 作用

#### 1、用户认证的核心组件

（1）在用户登录时，Spring Security 会调用 loadUserByUsername 方法

（2）根据用户名从数据源获取用户详细信息

（3）返回 UserDetails 对象，包含用户名、密码、权限等信息。

#### 2、连接 Spring Security 和用户数据存储

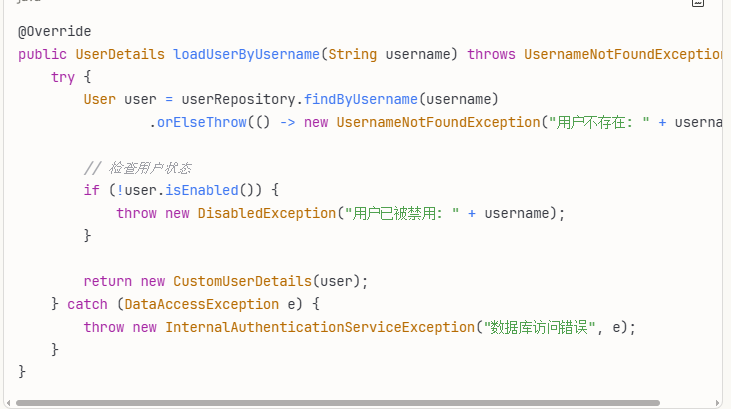
（1）作为 Spring Security 框架和实际用户数据存储之间的桥梁

（2）允许从任何数据源加载用户信息。

### 基本实现示例

### 最佳实践

#### 1、异常处理



#### 2、日志记录



### 总结

通过实现 UserDetailsService 接口，你可以将 Spring Security 与任何用户数据存储系统集成，为应用程序提供完整的认证和授权功能。

## PasswordEncoder

### 基本概念

PasswordEncoder 接口的作用是**处理用户密码的加密和验证**。它提供了一种抽象机制来对密码进行编码（加密）和匹配操作，增强了系统的安全性。

### 作用

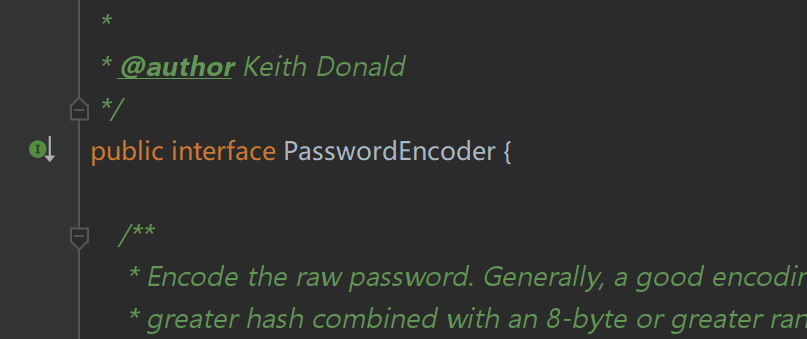
#### 密码加密

将明文密码转换为加密后的形式，通常是不可逆的（如哈希处理），以防止明文密码泄露。

#### 密码验证

将用户输入的密码与存储在数据库中的加密密码进行匹配，确保用户的身份。

### 源码



### 方法

#### String encode(CharSequence rawPassword)

对明文密码进行加密

#### boolean matches(CharSequence rawPassword, String encodedPassword)

验证密码是否匹配

### 为什么需要PasswordEncoder

#### 避免存储明文密码

如果直接存储用户的明文密码，一旦数据库被泄露，所有用户的密码都会暴露。通过使用 PasswordEncoder，即使数据库泄露，攻击者也无法轻易还原用户的密码。

#### 增强系统安全性

PasswordEncoder 可以采用安全的哈希算法（如 BCrypt、PBKDF2）加密密码，同时结合随机盐值和迭代哈希增强抗攻击能力。

#### 灵活适配多种编码方案

通过接口的实现，开发者可以根据具体需求选择不同的加密算法，满足不同的安全需求。

### 常见的PasswordEncoder 实现

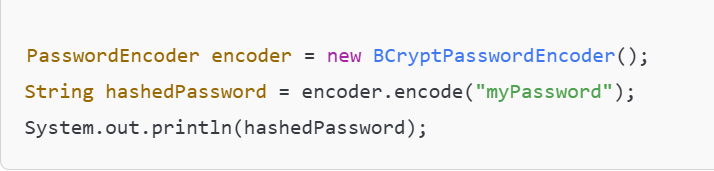
#### BCryptPasswordEncoder

1、基于 BCrypt 哈希算法，是 Spring Security 推荐的实现。

2、自动生成随机盐值，增加哈希结果的唯一性。

3、默认加密强度为 10，可通过构造函数调整。

示例：



特性：

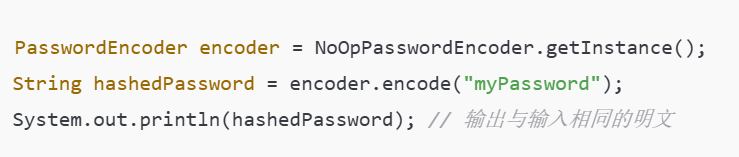
1、哈希结果每次都不同，因为会生成随机盐值。

2、强抗破解能力。

#### NoOpPasswordEncoder

1、不加密密码，直接返回明文。

2、已被标记为不推荐使用，仅用于开发或测试目的。

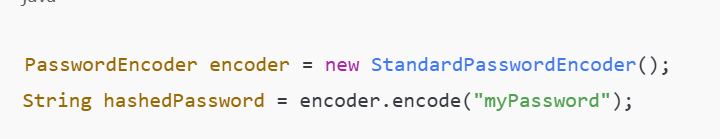


#### StandardPasswordEncoder

1、基于 SHA-256，默认使用固定盐值。

2、相比 BCryptPasswordEncoder，安全性略低。

示例：

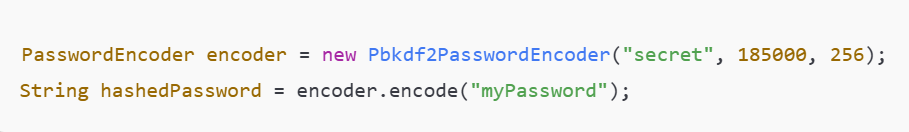


#### Pbkdf2PasswordEncoder

1、基于 PBKDF2 算法，支持迭代和盐值。

2、更灵活，但需手动配置参数。

示例：



#### DelegatingPasswordEncoder

1、支持多种编码算法，通过前缀标识选择使用的编码方式

2、适合在系统需要逐步迁移密码加密算法时使用。

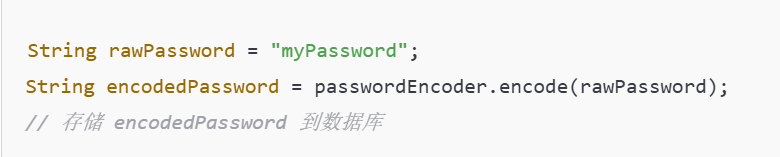
示例：



### 使用场景

#### 1、注册用户

调用 encode() 方法对用户输入的明文密码进行加密后存储。



#### 2、用户登录

使用 matches() 方法验证用户输入的密码是否与数据库中的加密密码匹配。



### 最佳实践

#### 1、始终使用强哈希算法

推荐使用 BCryptPasswordEncoder 或 Pbkdf2PasswordEncoder，避免使用简单的加密方式（如 MD5 或 SHA-1）。

#### 2、不要存储明文密码

数据库中存储的密码应该是加密后的形式，并且加密方式应该具备防止彩虹表攻击的能力。

#### 3、定期更新加密算法

如果系统使用的密码加密算法已经过时（如 MD5），可以使用 DelegatingPasswordEncoder 实现渐进式算法迁移。

## AuthenticationEntryPoint

### 基本概念

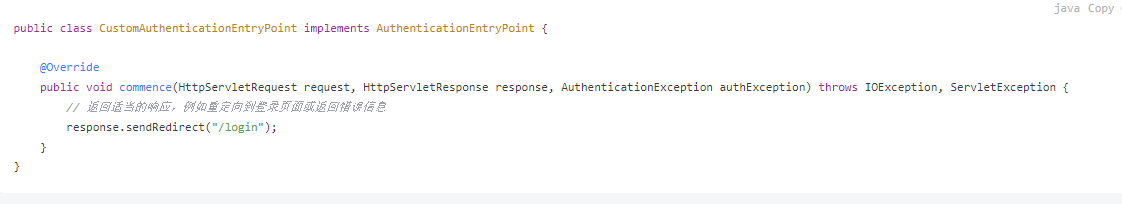
AuthenticationEntryPoint 接口是 Spring Security 中的一个重要接口，其作用是处理未经身份验证的请求，并负责返回适当的响应，提供用户进行身份验证的入口点。具体来说，AuthenticationEntryPoint 的作用如下：

1、处理未认证的请求：当用户尝试访问需要身份认证的资源，但用户尚未进行身份认证时，Spring Security 会调用 AuthenticationEntryPoint 的实现类来处理该请求。

2、返回适当的响应：AuthenticationEntryPoint 负责生成响应，以便提示用户进行身份验证。它可以返回包含登录页面的 HTML 页面、重定向到登录页面的 URL，或者其他适当的错误响应（如返回 HTTP 401 未授权错误）。

3、定制入口点行为：可以通过自定义实现 AuthenticationEntryPoint 接口来定制入口点的行为。例如，可以根据不同的业务需求，返回特定的错误信息、使用不同的身份验证方案（如 OAuth、JWT）等。

### 示例代码



上述代码中，自定义了一个 CustomAuthenticationEntryPoint 类实现了 AuthenticationEntryPoint 接口。在 commence() 方法中，可以根据业务需求自定义处理逻辑。在示例中，使用 response.sendRedirect() 方法将请求重定向到登录页面。

### 总结

AuthenticationEntryPoint 接口是 Spring Security 中处理未经身份验证的请求的入口点。它处理未认证的请求，并返回适当的响应，提示用户进行身份验证。通过自定义实现该接口，可以定制入口点的行为，以满足不同的业务需求。

### 说明

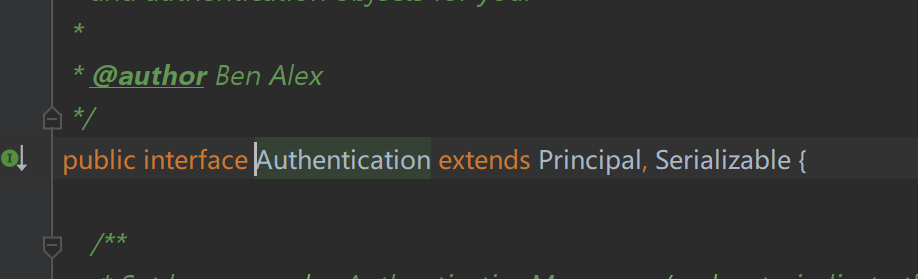
这个接口在spring-security-web-5.7.11.jar包中

## Authentication

### 基本概念

Spring Security 的核心是 Authentication 接口。它代表了当前用户的身份认证信息，包括用户名、密码、角色和其他相关的认证数据。

### 源码



### 作用

#### 1、身份认证

通过该接口，我们可以获取或设置当前用户的身份认证信息，以验证用户的身份是否合法。其中包括用户名、密码、角色和其他相关的认证数据。

#### 2、授权操作

基于身份认证信息，Authentication 接口提供了获取用户所具有的权限的方法，以便进行授权操作。通过 getAuthorities() 方法，可以获取用户所具有的权限列表，例如角色、权限标识等。

#### 3、上下文传递

Authentication 对象可以在系统的不同组件之间传递当前用户的身份认证信息。例如，在过滤器中进行认证后，将认证对象存储在 SecurityContextHolder 中，然后在控制器或服务中使用该对象进行权限检查或业务处理。

Authentication 接口的常用实现类是 UsernamePasswordAuthenticationToken，其中包含了用户的身份信息和凭据（密码）。除此之外，还有其他实现类如 RememberMeAuthenticationToken、JwtAuthenticationToken 等，用于支持不同的身份认证方式。

### 方法

#### Collection<? extends GrantedAuthority> getAuthorities()

返回用户权限列表

包含角色和权限信息（如 ROLE\_USER, ROLE\_ADMIN）

用于授权判断

#### Object getCredentials()

返回认证凭据

通常是密码、token等敏感信息

认证成功后通常会被清空

#### Object getDetails()

返回认证详细信息

额外的上下文信息（如IP地址、会话ID）

非核心认证数据

#### Object getPrincipal()

返回认证主体

通常是用户对象或用户名

代表"谁在访问系统"

#### boolean isAuthenticated();

返回认证状态

true：已认证

false：未认证

#### void setAuthenticated(boolean isAuthenticated)

设置认证状态

只能从 true 设为 false

从 false 设为 true 会抛异常（必须通过AuthenticationManager）

### 实现类

#### UsernamePasswordAuthenticationToken

#### RememberMeAuthenticationToken

#### JwtAuthenticationToken

### 使用步骤

#### 1、用户身份认证

在用户登录时，通过请求获取用户提供的用户名和密码，并将其封装成 Authentication 对象，通常是 UsernamePasswordAuthenticationToken 的实例。这个对象会被传递给 AuthenticationManager 进行认证。

#### 2、身份认证流程

AuthenticationManager 是 Spring Security 的核心身份认证管理器，负责对传入的 Authentication 对象进行身份验证。它会调用配置的 AuthenticationProvider 进行用户的身份认证。

#### 3、认证结果处理

认证提供者将根据用户提供的用户名和密码进行身份验证，并生成一个经过认证的 Authentication 对象。当认证成功时，Authentication 对象将包含用户的权限信息，并将其保存在 SecurityContextHolder 中供后续使用。

#### 4、授权操作

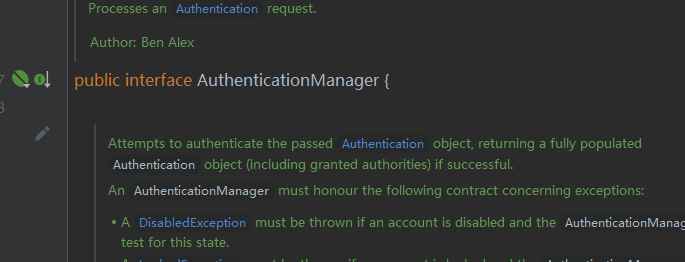
在应用程序的其他部分，可以使用 SecurityContextHolder 获取当前用户的 Authentication 对象，然后通过其中的方法获取用户的角色和权限信息，用于进行授权操作。

## AuthenticationManager

### 基本概念

用于处理身份认证操作。它负责对传入的 Authentication 对象进行身份验证，并返回经过认证的 Authentication 对象。

### 源码



### 作用

#### 1、身份认证

主要通过 authenticate() 方法，对传入的 Authentication 对象进行身份验证。它是 Spring Security 中的身份验证策略执行者。

#### 2、处理多种认证方式

AuthenticationManager 支持同时处理多种不同的身份认证方式，如用户名/密码、Remember Me、OAuth、JWT 等。根据不同的认证方式，会有相应的 AuthenticationProvider 类来实现具体的身份认证逻辑。

#### 3、链式调用

AuthenticationManager 可以配置多个 AuthenticationProvider，形成一个认证处理链。当一个 AuthenticationProvider 无法对请求进行身份认证时，会将认证请求传递给链中的下一个 AuthenticationProvider 进行尝试，直到有一个提供者成功认证或全部失败为止。

#### 4、异常处理

在进行身份认证时，AuthenticationManager 会抛出不同的异常来表示不同的认证结果。例如，BadCredentialsException 表示凭据（密码）不正确，LockedException 表示用户被锁定等。这些异常可由应用程序进行捕获并进行相应的处理。

### 实现类

AuthenticationManager 接口的常用实现类是 ProviderManager，它是基于 AuthenticationProvider 的实现。ProviderManager 可以配置多个 AuthenticationProvider 实例，并按顺序尝试使用它们进行身份认证。

### 使用步骤

#### 1、配置 AuthenticationManager

通常通过配置文件或编程方式配置 AuthenticationManager 的实例（例如，ProviderManager），并注入所需的 AuthenticationProvider 实现。

#### 2、调用 authenticate() 方法

当需要进行用户身份认证时，将 Authentication 对象传递给 AuthenticationManager 的 authenticate() 方法。AuthenticationManager 会根据配置的 AuthenticationProvider 进行身份认证操作。

#### 3、处理身份认证结果

authenticate() 方法返回一个经过认证的 Authentication 对象，其中包含了用户的身份信息和权限信息。可以根据返回的对象来判断认证成功与否，并根据需要进行后续的授权或其他处理。

### 总结

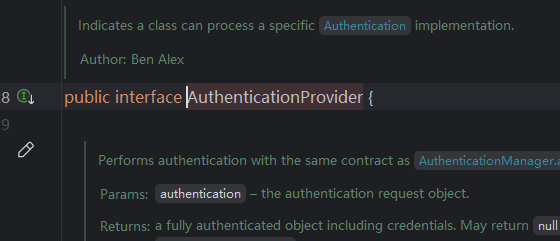
AuthenticationManager 接口是 Spring Security 中用于处理身份认证操作的核心接口。它实现了身份认证的策略和逻辑，并支持多种不同的认证方式。通过配置 AuthenticationManager 和 AuthenticationProvider，可以进行灵活的身份认证管理，并获取经过认证的 Authentication 对象供后续使用。

## AuthenticationProvider

### 基本概念

用于在身份认证过程中实现对用户身份的认证逻辑。它负责验证用户提供的身份凭据，并返回经过认证的 Authentication 对象。

### 源码



### 方法

#### Authentication authenticate(Authentication authentication)

#### boolean supports(Class<?> authentication)

### 作用

#### 1、身份认证

通过 authenticate() 方法，对用户提供的身份凭据进行验证。该方法接收一个 Authentication 对象作为参数，其中封装了用户提供的身份凭据（如用户名和密码），以及其他相关的认证数据。实现类需要根据具体的认证方式，如用户名/密码、Remember Me、OAuth、JWT 等，验证用户的身份凭据是否有效。

#### 2、认证结果处理

当身份凭据验证成功时，应该创建一个经过认证的 Authentication 对象，并将用户的身份信息和权限信息设置到该对象中。可以使用 Spring Security 提供的实现类，如 UsernamePasswordAuthenticationToken、RememberMeAuthenticationToken 等，也可以自定义实现类来满足特定的需求。

#### 3、异常处理

如果身份凭据验证失败，应该抛出相应的异常，如 BadCredentialsException 表示凭据不正确，LockedException 表示用户被锁定等。这些异常可由应用程序进行捕获并进行相应的处理，例如展示错误消息给用户。

### 实现类

AuthenticationProvider 接口的常见实现类是 DaoAuthenticationProvider，它基于数据访问对象 (UserDetailsService) 实现了身份验证逻辑。DaoAuthenticationProvider 通过查询数据库或其他存储介质，验证用户的身份凭据，并构建经过认证的 Authentication 对象。

### 使用步骤

#### 1、实现 AuthenticationProvider 接口

根据自己的需求，创建一个类实现 AuthenticationProvider 接口，实现其中的 authenticate() 方法。

#### 2、编写身份验证逻辑

在 authenticate() 方法中，根据具体的身份认证方式，如用户名/密码、Remember Me、OAuth、JWT 等，验证用户的身份凭据是否有效，并创建一个经过认证的 Authentication 对象返回。

#### 3、配置 AuthenticationProvider

将自定义的 AuthenticationProvider 实例配置到 AuthenticationManager 中，以便在进行身份认证时调用。

#### 4、处理身份认证结果

在 AuthenticationProvider 中，根据身份凭据的验证结果，可以抛出相应的异常或返回经过认证的 Authentication 对象。在应用程序中，可以根据返回的对象来判断认证成功与否，并根据需要进行后续的授权或其他处理。

### 总结

AuthenticationProvider 接口是 Spring Security 中用于处理用户身份认证逻辑的核心接口。它负责验证用户提供的身份凭据，并返回经过认证的 Authentication 对象。通过实现 AuthenticationProvider 接口，可以自定义身份认证逻辑，进行灵活的用户身份验证，并将认证结果返回给 AuthenticationManager 进行后续处理。

## TokenGranter

### 基本概念

在Spring Security中，TokenGranter接口是用于处理OAuth2认证流程中的授权请求的关键接口之一。它定义了一个方法 grant()，用于根据不同的授权类型进行授权操作，并返回授权结果。

TokenGranter接口的实现类通常会针对不同的授权类型（如授权码、密码、客户端凭证等）提供相应的处理逻辑。例如，授权码类型的TokenGranter会验证授权码是否有效、交换访问令牌等；密码类型的TokenGranter会验证用户名和密码是否正确，并分发访问令牌等。可以通过自定义并实现TokenGranter接口来扩展Spring Security的授权流程，以支持其他自定义的授权类型。实现TokenGranter接口的类需要将自定义的授权逻辑放在 grant() 方法中，并根据具体的业务逻辑进行处理。

需要注意的是，TokenGranter接口通常配合CompositeTokenGranter一起使用，后者用于组合多个TokenGranter并提供统一的入口。CompositeTokenGranter会根据授权请求的类型，将其分派给相应的TokenGranter进行处理。

总结来说，TokenGranter接口是Spring Security中用于处理OAuth2认证流程中授权请求的核心接口。它定义了一个方法 grant()，根据不同的授权类型执行相应的授权逻辑，并返回OAuth2AccessToken对象作为授权结果。通过自定义实现TokenGranter接口，可以扩展Spring Security的授权流程，支持其他自定义的授权类型。

### 方法

#### OAuth2AccessToken grant(String grantType, TokenRequest tokenRequest);

用于执行授权操作。

grantType：授权类型，表示当前授权请求的类型，如授权码、密码、客户端凭证等。

tokenRequest：TokenRequest对象，封装了当前授权请求的相关信息，如授权参数、客户端信息等。

grant() 方法会根据 grantType 和 tokenRequest 的值来执行相应的授权逻辑，并返回OAuth2AccessToken对象作为授权结果。

## AccessDecisionManager

### 基本概念

用于在访问控制过程中进行访问决策（Access Decision）的管理。它负责确定用户是否有访问特定资源（如 URL、方法、注解等）的权限。

### 作用

1、访问控制决策：通过 decide() 方法，根据用户的身份信息和权限信息，以及资源的安全属性，判断用户是否有权访问该资源。该方法接收一个 Authentication 对象作为参数，其中封装了用户的身份信息和权限信息；还接收一个 FilterInvocation 对象，该对象包含了当前请求的相关信息（如 URL、HTTP 方法）。实现类需要根据具体的访问控制策略，判断用户是否满足访问该资源的要求。

2、访问决策处理：当访问决策被确定后，AccessDecisionManager 可以返回一个 AccessDecision 对象，该对象指示用户是否有权访问该资源。实现类可以使用 Spring Security 提供的实现类，如 AffirmativeBased、ConsensusBased、UnanimousBased 等，也可以自定义实现类来满足特定的需求。

3、异常处理：如果访问决策无法确定，或者用户不具备访问权限，AccessDecisionManager 应该抛出相应的异常，如 AccessDeniedException 表示拒绝访问，InsufficientAuthenticationException 表示认证不足等。这些异常可由应用程序进行捕获并进行相应的处理，例如展示拒绝访问的错误页面给用户。

### 使用步骤

1、实现 AccessDecisionManager 接口：根据自己的需求，创建一个类实现 AccessDecisionManager 接口，实现其中的 decide() 方法。

2、编写访问控制决策逻辑：在 decide() 方法中，根据用户的身份信息和权限信息，以及资源的安全属性，判断用户是否有权访问该资源，并返回一个 AccessDecision 对象表示访问决策结果。

3、配置 AccessDecisionManager：将自定义的 AccessDecisionManager 实例配置到 AccessDecisionManager 中，以便在进行访问控制决策时调用。

4、处理访问控制决策结果：在 AccessDecisionManager 中，根据访问决策的结果，可以抛出相应的异常或返回一个 AccessDecision 对象。在应用程序中，可以根据返回的对象来判断用户是否有权访问资源，并根据需要进行后续的处理。

## ProviderManagerBuilder

## OAuth2AccessToken

### 基本概念

OAuth2AccessToken类用于表示OAuth 2.0授权访问令牌。它是一个具体的实现类，用于封装和处理OAuth 2.0协议中的访问令牌相关信息。

### 方法

#### public String getTokenValue()

### 作用

1、表示访问令牌：OAuth2AccessToken类用于表示从Authorization Server获取的访问令牌。访问令牌是OAuth 2.0协议中用于进行资源访问的凭证，它包含了访问资源所需的必要信息，例如令牌本身、过期时间、作用域、授权类型等。OAuth2AccessToken提供了相应的方法，使我们能够方便地获取和操作这些信息。

2、

## SecurityFilterChain

## SecurityContext

# 类

## WebSecurityConfiguration

## FilterChainProxy

## AuthorityUtils

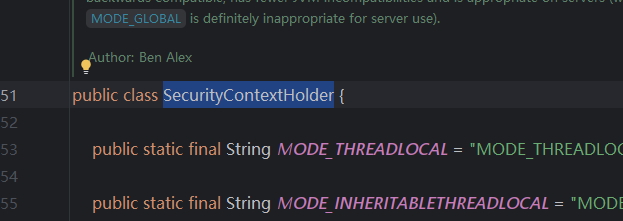
## SecurityContextPersistenceFilter

## SecurityContextHolder

### 基本概念

SecurityContextHolder 是 Spring Security 框架提供的一个中心化存储和访问安全上下文信息的工具类。它用于在应用程序中管理当前用户的认证和授权信息。SecurityContextHolder 的主要作用是提供了一种便捷的方式来获取和操作当前用户的安全上下文信息，无需显式传递这些信息。它基于 ThreadLocal 实现，将当前用户的安全上下文与当前线程绑定，确保在同一线程中的所有组件都可以方便地访问当前用户的安全信息。

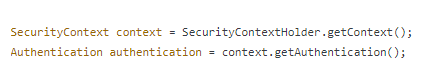
### 源码



### 方法

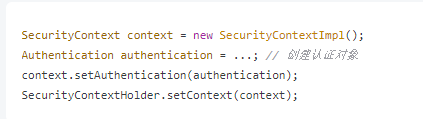
#### public static SecurityContext getContext()

用于获取当前线程中的安全上下文对象。返回值类型为 SecurityContext，可以通过调用 getAuthentication() 方法获取当前用户的认证对象。



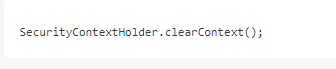
#### public static void setContext(SecurityContext context)

用于设置当前线程的安全上下文对象。



#### public static void clearContext()

用于清除当前线程中的安全上下文对象。通常在请求结束后使用，以确保不会意外地将上下文信息传递给下一个请求。



通过 SecurityContextHolder，我们可以在任何需要访问当前用户的安全信息的地方获取和操作安全上下文。例如，在控制器或服务中，可以使用 SecurityContextHolder 获取当前用户的认证对象，并根据认证对象进行权限检查或其他操作。需要注意的是，SecurityContextHolder 是基于 ThreadLocal 实现的。因此，在异步请求或线程池中使用时，可能需要额外的配置和注意事项来确保正确地传播安全上下文信息。

## WebAuthenticationDetails

## WebAuthenticationDetailsSource

## RequestContextHolder类与SecurityContextHolder类的区别

### RequestContextHolder适用场景

1、当处理基于 HTTP 协议的请求时，可以使用 RequestContextHolder 获取当前请求的上下文信息，如请求对象、会话对象等。

2、通常在 Web 应用程序中使用，例如控制器、服务、拦截器等组件中，用于获取当前请求的上下文信息，方便访问请求相关的数据。

### SecurityContextHolder 适用场景

1、当处理安全相关的操作时，可以使用 SecurityContextHolder 获取当前用户的安全上下文信息，如认证对象、权限信息等。

2、主要用于 Spring Security 框架中，用于管理当前用户的认证和授权信息。

### 区别

1、功能不同：RequestContextHolder 提供了访问当前请求的上下文信息的方法，而 SecurityContextHolder 提供了访问当前用户的安全上下文信息的方法。

2、使用场景不同：RequestContextHolder 主要在处理请求时使用，用于获取请求相关的信息。而 SecurityContextHolder 主要在处理安全操作时使用，用于获取当前用户的认证和授权信息。

### 总结

1、RequestContextHolder 用于获取当前请求的上下文信息，在处理基于 HTTP 协议的请求时使用，适用于 Web 应用程序中的控制器、服务、拦截器等组件。

2、SecurityContextHolder 用于获取当前用户的安全上下文信息，主要在处理安全相关操作时使用，适用于 Spring Security 框架中。它管理当前用户的认证和授权信息。

## WebSecurityConfigurerAdapter（重点）

### 基本概念

它是用于配置Spring Security的一个关键类。通过继承WebSecurityConfigurerAdapter类并覆盖其中的方法，我们可以定义自己的安全配置规则，以实现对Web应用程序的安全性进行定制。

### 方法

#### protected void configure(AuthenticationManagerBuilder auth)

#### protected void configure(HttpSecurity http)

#### public void configure(WebSecurity web)

### 作用

1、提供默认的安全配置：WebSecurityConfigurerAdapter类定义了一系列默认的安全配置规则，例如禁止跨站请求伪造（CSRF）保护、允许同源策略（默认情况下JavaScript无法从外部域加载）等。当我们需要在项目中使用Spring Security时，可以继承这个类，获取这些默认配置的基础上进行定制。

2、允许自定义安全配置：通过继承WebSecurityConfigurerAdapter类，我们可以覆盖其中的方法，以实现自定义的安全配置。其中最常用的方法是configure(HttpSecurity http)，它允许我们定义如何保护HTTP请求，哪些URL路径需要被保护，哪些URL路径可以被公开访问等。我们可以通过配置认证方式、权限控制、登录页面、注销处理、异常处理等来满足特定的安全需求。

3、集成其他安全功能：WebSecurityConfigurerAdapter类提供了一系列与其他安全功能集成的方法。例如，我们可以使用configure(AuthenticationManagerBuilder auth)方法配置用户认证功能，定义用户、密码和角色的来源；使用configure(WebSecurity web)方法配置不需要进行安全保护的静态资源路径等。这些方法提供了灵活的扩展性，以适应各种场景下的安全需求。

### 总结

WebSecurityConfigurerAdapter类是Spring Security中的一个关键类，通过继承该类并覆盖其中的方法，我们可以对Spring Security进行定制化的配置。它允许我们定义自己的安全规则，控制URL的访问权限、认证方式以及其他与安全相关的功能。通过合理使用WebSecurityConfigurerAdapter类，我们可以实现对Web应用程序的全面安全管理。

## HttpSecurity

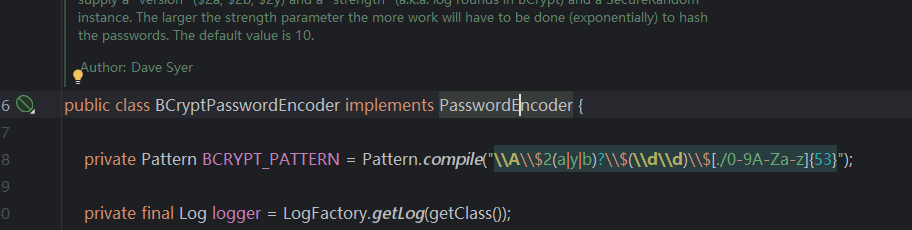
## DelegatingPasswordEncoder

## BCryptPasswordEncoder

### 基本概念

BCryptPasswordEncoder是Spring Security提供的密码加密工具，使用**BCrypt算法**进行不可逆哈希加密。

### 源码



### 核心特点

1、安全性高：内置salt，防止彩虹表攻击

2、慢哈希：计算成本高，抵御暴力破解

3、可调强度：支持4-31轮加密强度

### BCrypt算法特点

#### 核心机制

1、基于Blowfish：改进的Blowfish加密算法

2、自适应哈希：可调节计算成本，随硬件发展保持安全性

3、工作因子：2^rounds次迭代，默认2^10=1024次

#### 关键特性

##### 1、内置salt



##### 2、慢哈希设计

（1）故意消耗CPU时间

（2）每增加1轮，时间翻倍

（3）抵御暴力破解和硬件攻击

##### 3、输出格式



60字符固定长度

包含算法、轮数、salt、hash值

##### 4、单向加密

只能从明文生成哈希，无法从哈希反推明文

#### 安全优势

**1、防彩虹表**：每个密码有唯一salt

**2、时间成本**：破解单个密码需大量计算

3、**可扩展性**：随硬件升级调整轮数

4、**验证一致**：相同输入总是匹配成功。

#### 性能对比



#### 实现细节

1. 生成22字符随机salt

2. 将密码和salt组合

3. 进行2^rounds轮Blowfish加密

4. 输出标准格式字符串

BCrypt的核心就是**慢而安全**，通过时间成本换取密码安全。

### 构造函数

#### public BCryptPasswordEncoder()

默认强度是10轮

#### public BCryptPasswordEncoder(int strength)

#### public BCryptPasswordEncoder(BCryptVersion version)

### 方法

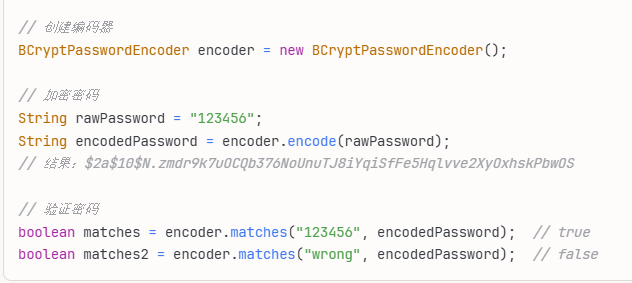
#### public String encode(CharSequence rawPassword)

给原始密码进行加密

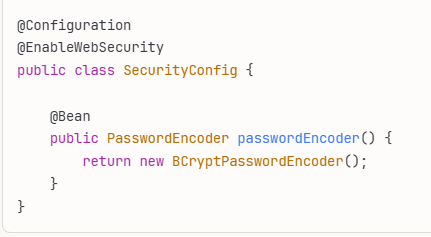
#### public boolean matches(CharSequence rawPassword, String encodedPassword)

密码进行匹配

### 基本使用



### Spring Boot集成



### 实际应用



### 重要特性



### 最佳实践

1、**统一使用**：整个应用使用同一个实例

2、**合理强度**：平衡安全性和性能

3、**异步处理**：注册/重置密码时考虑异步

4、**升级兼容**：老密码逐步升级到新强度

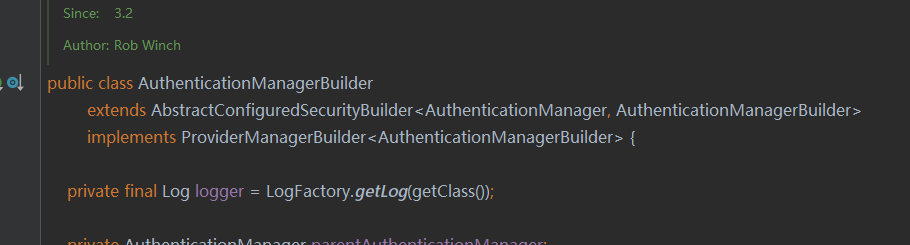
核心就是：encode()加密，matches()验证，简单安全。

## AuthenticationManagerBuilder

### 基本概念

在Spring Security中，AuthenticationManagerBuilder类用于构建和配置AuthenticationManager对象，它是处理用户身份认证的核心组件。AuthenticationManagerBuilder提供了一些方法，允许我们通过不同方式进行用户认证，例如内存验证、数据库验证、LDAP验证等。

### 源码



### 方法

#### public <T extends UserDetailsService> DaoAuthenticationConfigurer<AuthenticationManagerBuilder, T> userDetailsService(T userDetailsService)

### 作用

#### 1、简化用户认证配置

AuthenticationManagerBuilder提供了一系列方法，使得配置用户认证变得简单而灵活。我们可以使用inMemoryAuthentication()方法在内存中配置用户，也可以使用jdbcAuthentication()方法从数据库中进行用户认证。另外还有其他方法用于进行LDAP认证、自定义认证提供者等。通过这些方法，我们可以按照不同的场景和需求选择适合的认证方式进行配置。

2、配置用户信息存储：AuthenticationManagerBuilder类可以与UserDetailsService接口结合使用，用于从不同的数据源中加载用户信息。我们可以使用userDetailsService()方法配置自定义的UserDetailsService实现类，该实现类负责从数据库、LDAP或其他数据源中加载用户信息。这样，我们就可以根据实际情况灵活选择用户信息的存储方式。

3、配置密码加密和验证：AuthenticationManagerBuilder提供了一系列方法用于配置密码的加密和验证。通过调用passwordEncoder()方法，我们可以指定密码的加密算法，例如BCryptPasswordEncoder、PasswordEncoder等。此外，可以使用password("{加密后的密码}")方法设置已经加密的密码。AuthenticationManagerBuilder负责将传递的原始密码进行加密和验证，以确保密码的安全性。

4、自定义认证流程：通过AuthenticationManagerBuilder，我们可以实现自定义的认证流程。例如，我们可以使用authenticationProvider()方法添加自定义的AuthenticationProvider实现类来处理特定的认证需求。这样，我们可以扩展Spring Security的默认认证逻辑，并根据业务需求实现额外的认证方式或规则。

### 总结

AuthenticationManagerBuilder是Spring Security中用于构建和配置AuthenticationManager的重要组件。它简化了用户认证的配置过程，提供了多种认证方式的选择，包括内存验证、数据库验证、LDAP验证等。通过AuthenticationManagerBuilder，我们可以轻松地配置用户信息存储方式、密码加密和验证规则，还可以实现自定义的认证流程。这使得Spring Security能够满足不同场景下的用户认证需求。

## AbstractSecurityInterceptor

## ExceptionTranslationFilter

## JdbcClientDetailsService

### 基本概念

在Spring Security中，JdbcClientDetailsService是一个实现了ClientDetailsService接口的类，用于从数据库中获取客户端的详细信息。它的主要作用是提供客户端的验证和授权服务，以支持OAuth2认证流程中的客户端身份验证。

具体而言，JdbcClientDetailsService的作用如下：

1、获取客户端详情：通过查询数据库表中存储的客户端信息，JdbcClientDetailsService可以获取特定客户端的详细信息，包括客户端ID、客户端密钥（加密后）、授权模式、授权范围、重定向URI等。这些信息将在OAuth2流程中用于验证客户端的合法性和权限。

2、客户端验证：在OAuth2认证流程中，客户端需要提供有效的客户端ID和客户端密钥（加密后）来进行身份验证。JdbcClientDetailsService可以通过验证客户端提供的信息与数据库中存储的客户端信息进行匹配，确保客户端的合法性，防止未经授权的客户端访问受保护的资源。

3、授权流程支持：根据客户端的配置信息，JdbcClientDetailsService可以判断客户端是否具有特定的授权模式和授权范围。例如，客户端可能只被允许使用授权码模式（authorization\_code），或只能请求特定范围的访问令牌。JdbcClientDetailsService可以根据这些配置信息为每个客户端提供相应的授权服务。

4、动态客户端管理：通过数据库中存储的客户端信息，JdbcClientDetailsService支持动态地添加、修改和删除客户端。这意味着可以在运行时添加新的客户端，而无需修改代码。这对于需要管理大量客户端（如第三方应用程序）的情况非常有用。

需要注意的是，JdbcClientDetailsService需要与数据库交互来获取客户端信息，因此在使用之前需要正确配置数据源和相应的表结构。Spring Security提供了默认的表结构和SQL语句，但也可以自定义表结构和SQL语句以满足特定的需求。

总结来说，JdbcClientDetailsService在Spring Security中负责从数据库中获取并验证客户端的详细信息，以支持OAuth2认证流程中的客户端身份验证和授权。它是实现OAuth2认证服务器中客户端管理的关键组件之一。

## TokenEndpoint

## ProviderManager

### 基本概念

## AbstractTokenGranter

## CompositeTokenGranter

### 基本概念

在Spring Security中，CompositeTokenGranter类是一个实现了TokenGranter接口的类。它的主要作用是将多个TokenGranter组合在一起，形成一个可以处理多种授权类型的复合TokenGranter。

## InMemoryAuthorizationCodeServices

## OncePerRequestFilter

## DaoAuthenticationProvider

## SimpleGrantedAuthority

## UsernamePasswordAuthenticationFilter

## UsernamePasswordAuthenticationToken

### 基本概念

UsernamePasswordAuthenticationToken 是 Spring Security 中最常用的认证令牌类，用于封装用户名和密码的认证信息。它是 AbstractAuthenticationToken 的具体实现，在整个认证流程中扮演着核心角色。

# 注解

## @EnableWebSecurity

### 基本概念

主要用于启用 Spring Security 的功能。

### 作用

#### 1、启用 Spring Security 配置

@EnableWebSecurity 注解告诉 Spring 框架启用 Web 安全性，并加载 Spring Security 的配置。具体来说，它会引导 Spring Security 去寻找并应用 Web 安全相关的配置类。

如果你没有显式定义任何 WebSecurityConfigurerAdapter 或类似的安全配置，Spring Security 会提供一个默认的配置，这通常包括：

（1）对所有 URL 的访问都需要身份验证。

（2）默认提供一个内存中的用户（用户名为 user，密码在启动时打印在控制台中）。

（3）提供一个基本的登录页面。

#### 2、与 @Configuration 一起使用

@EnableWebSecurity 本质上是一个组合注解，包含了 @Configuration 注解，这意味着它也是一个配置类的标识。通常，带有 @EnableWebSecurity 的类会定义 Spring Security 的相关配置。

示例：



#### 3、引入 Spring Security 的核心组件

@EnableWebSecurity 注解会自动引入与 Spring Security 相关的一些核心组件，常见的包括：

WebSecurityConfigurer 接口及其实现类，用于自定义安全配置。

DelegatingFilterProxy，它是一个用于集成 Spring Security 的核心过滤器。

AuthenticationManager 和 SecurityContext，用于身份验证和上下文管理。

#### 4、替代 SpringSecurityFilterChain 的 XML 配置

在 Spring Security 早期版本中，安全配置通常通过 XML 文件完成，例如：



@EnableWebSecurity 的引入使得开发者能够通过 Java 配置替代 XML 配置，以实现更灵活、更强大的功能。

#### 5、与 @SpringBootApplication 的默认安全配置整合

在 Spring Boot 中，如果你不定义任何自定义的安全配置类，Spring Boot 会自动为你启用一个默认的安全配置。但一旦你添加了带有 @EnableWebSecurity 的配置类，这些默认配置会被覆盖。

### 底层实现原理

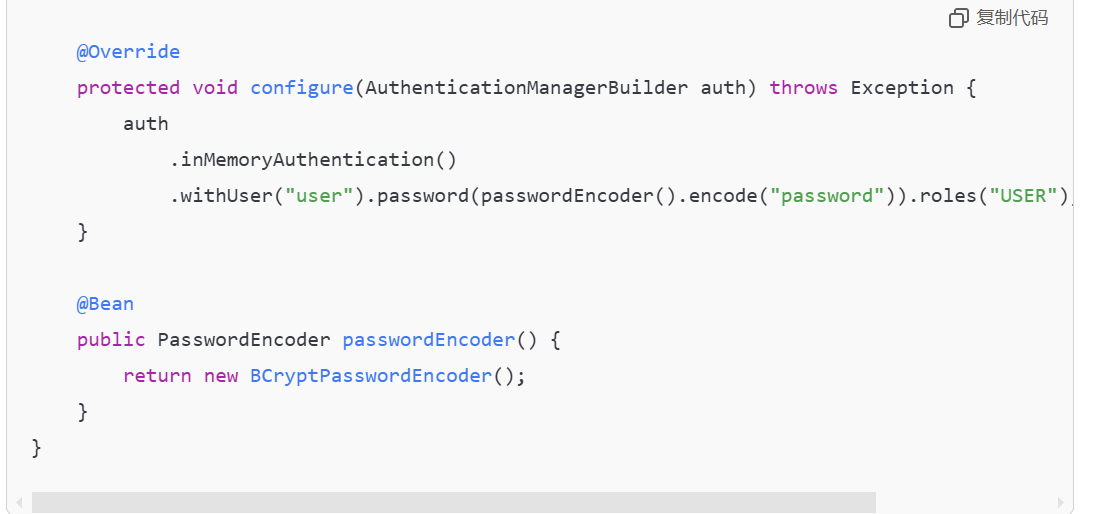
1、注解驱动 @EnableWebSecurity 会引入一个名为 SpringWebSecurityConfiguration 的配置类，这是 Spring Security 配置的入口。

2、自动配置 在 Spring 的启动过程中，@EnableWebSecurity 会触发 WebSecurityConfiguration 的加载，这个类会解析所有实现了 WebSecurityConfigurer 接口的 Bean（包括 WebSecurityConfigurerAdapter 的子类），并将它们合并到一个全局的安全过滤器链中。

3、生成 Filter Chain 它最终会生成一个 FilterChainProxy，这是 Spring Security 的核心过滤器链，负责拦截所有 HTTP 请求并执行相应的安全逻辑。

### 示例





### 注意事项

1、自定义配置优先 如果存在多个实现了 WebSecurityConfigurer 的类，Spring Security 会根据配置的顺序来决定优先级。

2、版本差异 在 Spring Security 5.7 开始，WebSecurityConfigurerAdapter 已被标记为过时，推荐使用 SecurityFilterChain 来定义安全配置。



通过 @EnableWebSecurity，开发者可以灵活配置 Spring Security 的行为，同时享受其默认的安全机制。这是 Spring Security 中启动安全功能的核心入口。

## @EnableAuthorizationServer

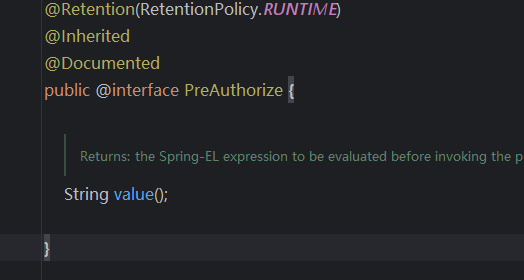
## @EnableResourceServer

## @PreAuthorize

### 基本概念

@PreAuthorize 注解允许你在方法执行前定义访问控制规则，如果权限检查失败，方法将不会被执行，并抛出 AccessDeniedException 异常。

### 源码

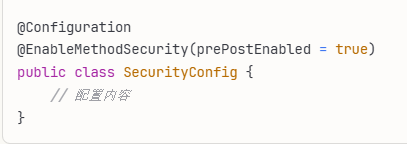


### 启用方法级安全

首先需要在配置类中启用方法级安全



或者在 Spring Boot 2.6+ 中



### 基本用法

#### 1、基于角色的权限控制



#### 2、基于权限的控制



#### 3、基于用户身份的控制



### 常用的SpEL 表达式

#### 权限检查方法

##### hasRole('ROLE\_NAME')

检查角色

##### hasAnyRole('ROLE1', 'ROLE2')

检查多个角色中的任一个

##### hasAuthority('PERMISSION')

检查权限

##### hasAnyAuthority('PERM1', 'PERM2')

检查多个权限中的任一个

#### 用户状态检查

##### isAuthenticated()

用户已认证

##### isAnonymous()

匿名用户

##### isRememberMe()

通过记住我功能登录

##### isFullyAuthenticated()

完全认证（非记住我）

#### 访问认证信息

##### authentication

当前认证对象

##### authentication.name

用户名

##### authentication.principal

用户主体对象

##### authentication.authorities

用户权限集合

### 实际应用示例



### 注意事项

1、性能考虑：每次方法调用都会执行权限检查，复杂的表达式可能影响性能

2、异常处理：权限检查失败会抛出 AccessDeniedException，需要适当处理

3、测试：需要在测试中模拟不同的用户角色和权限

4、表达式复杂度：避免过于复杂的表达式，可以考虑提取到自定义方法中

### 总结

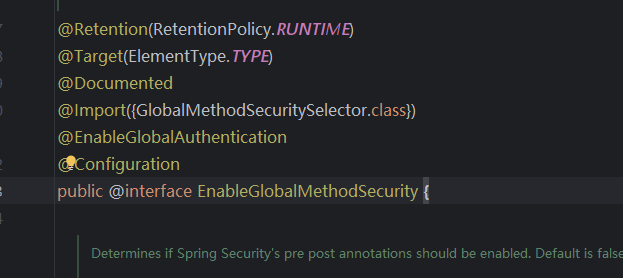
@PreAuthorize 提供了强大而灵活的方法级安全控制，是 Spring Security 中实现细粒度权限控制的重要工具。

## @EnableGlobalMethodSecurity

### 基本概念

@EnableGlobalMethodSecurity 注解用于在 Spring 应用程序中启用全局方法级别的安全控制，允许你在方法上使用各种安全注解来控制访问权限。

### 源码



### 属性

#### prePostEnabled

启用 @PreAuthorize 和 @PostAuthorize 注解支持，默认值是false

支持的注解：

@PreAuthorize - 方法执行前检查权限

@PostAuthorize - 方法执行后检查权限

@PreFilter - 方法执行前过滤集合参数

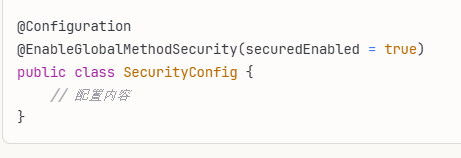
@PostFilter - 方法执行后过滤返回的集合

##### 使用示例



#### securedEnabled

启用 @Secured 注解支持



##### 使用示例



### 版本更新说明

在 Spring Security 5.6+ 版本中，@EnableGlobalMethodSecurity 已被标记为废弃，推荐使用 @EnableMethodSecurity。



### 注意事项

**1、性能影响**：方法级安全会增加方法调用的开销，需要权衡安全性和性能

**2、代理限制**：只对通过 Spring 代理调用的方法有效，内部方法调用不会触发安全检查

**3、异常处理**：需要适当处理 AccessDeniedException 异常

**4、测试复杂性**：需要在测试中模拟各种安全上下文

### 总结

@EnableGlobalMethodSecurity 是 Spring Security 方法级安全的基础，为应用程序提供了细粒度的权限控制能力。

## @EnableMethodSecurity

## @Secured