认证 Authentication

一、架构组件 Architecture Components

- 组件列表-简述

  - SecurityContextHolder – SecurityContextHolder 是 Spring Security 存储身份验证者详细信息的地方
  - SecurityContext – 从 SecurityContextHolder 中获取，包含当前认证用户的身份验证
  - Authentication – 可以是 AuthenticationManager 的输入，以提供用户提供的用于身份验证的凭据 或来自 SecurityContext 的当前用户
  - GrantedAuthority – 在身份验证上授予主体的权限（即角色、范围等）
  - AuthenticationManager – 定义 Spring Security 的过滤器如何执行身份验证的 API
  - ProviderManager – 最常见的 AuthenticationManager 的实现
  - AuthenticationProvider – ProviderManager 使用它来执行特定类型的身份验证
  - Request Credentials with AuthenticationEntryPoint – 用于从客户端请求凭据（即重定向到登录页面、发送 WWW-Authenticate 响应等
  - AbstractAuthenticationProcessingFilter – 用于身份验证的基本过滤器。这也让我们很好地了解了认证的高层流程，以及各部分如何协同工作。

1.1 认证上下文持有人 SecurityContextHolder

Spring Security 认证模型的核心是 SecurityContextHolder （权限上下文持有者）。它包含 SecurityContext（权限上下文）。

SecurityContextHolder 是 Spring Security 存储身份验证者详细信息的地方。 Spring Security 不关心 SecurityContextHolder 是如何填充的。如果它包含一个值，则将其用作当前经过身份验证的用户。

指示用户已通过身份验证的最简单方法是直接设置 SecurityContextHolder。

Java

```
/**
 * 直接设置用户上下文
 */

// 我们首先创建一个空的 SecurityContext。
// 重要的是创建一个新的 SecurityContext 实例而不是使用
//  - SecurityContextHolder.getContext().setAuthentication(authentication)
//  - 以避免跨多个线程的竞争条件
SecurityContext context = SecurityContextHolder.createEmptyContext();

// 接下来我们创建一个新的 Authentication 对象。
// Spring Security 不关心 SecurityContext 上设置了哪种类型的身份验证实现。
// 这里我们使用 TestingAuthenticationToken 因为它非常简单。
// 更常见的生产场景是 UsernamePasswordAuthenticationToken(userDetails, passwor
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER");
// 最后，我们在 SecurityContextHolder 上设置 SecurityContext。
// Spring Security 将使用此信息进行授权。
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);


// ========= 无注释的 =======
/**
 * 直接设置用户上下文
 */
SecurityContext context = SecurityContextHolder.createEmptyContext();
```

```
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER");
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);
```

如果想要获取有关通过身份验证的主体的信息，可以通过访问 SecurityContextHolder 来实现

Java

```
/**
 * 访问当前经过身份验证的用户
 */
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities = authentication.getAuth
```

默认情况下，SecurityContextHolder 使用 ThreadLocal 来存储用户认证主体详细信息，所以 SecurityContext 始终可用于同一执行线程中的方法，即使 SecurityContext 没有明确地作为参数传递给这些方法。

如果在处理当前主体的请求后注意清除线程，那么以这种方式使用 ThreadLocal 是非常安全的。 Spring Security 的 FilterChainProxy 确保 SecurityContext 总是被清除

1.2 安全上下文 SecurityContext

SecurityContext 是从 SecurityContextHolder 中获去的包含了 Authentication 对象的一个对象

1.3 身份认证 Authentication

Authentication 在 Spring Security 中主要有两个作用

- AuthenticationManager 的输入，用于提供用户提供的身份验证凭据。在这种情况下使用时，调用 Authentication 的 isAuthenticated() 会返回 false
- 代表当前认证的用户。当前的 Authentication 可以从 SecurityContext 中获得。该 Authentication 包含：
    - principal – 标识用户。当使用用户名/密码进行身份验证时，这通常是 UserDetails 的一个实例
    - credentials – 通常是密码。在许多情况下，这将在用户通过身份验证后被清除，以确保它不被泄露
    - authorities – GrantedAuthoritys 是用户被授予的高层次权限，一般是用户的角色或者权限范围

1.4 授予的权限 GrantedAuthority

GrantedAuthoritys 是用户被授予的高层次权限，一般是用户的角色或者权限范围

GrantedAuthoritys 可以从 Authentication.getAuthorities() 方法中获得。此方法提供 GrantedAuthority 对象的 `Collection` 。GrantedAuthority 是授予委托人的权限。此类权限通常是"角色"，例如 `ROLE_ADMINISTRATOR` 或 `ROLE_HR_SUPERVISOR` 。这些角色稍后会针对 Web 授权、方法授权和域对象授权进行配置。 Spring Security 的其他部分能够解释这些权限，并期望它们存在。当使用基于用户名/密码的身份验证时，GrantedAuthoritys 通常由 UserDetailsService 加载。

通常，GrantedAuthority 对象是应用范围的权限。它们并不特定于某个特定的域对象。因此，你不可能有一个 GrantedAuthority 来代表第 54 号 Employee 对象的权限，因为如果有成千上万个这样的授权，你会很快耗尽内存（至少，导致应用程序需要很长的时间来验证用户）。您应该为此目的使用项目的域对象安全功能。

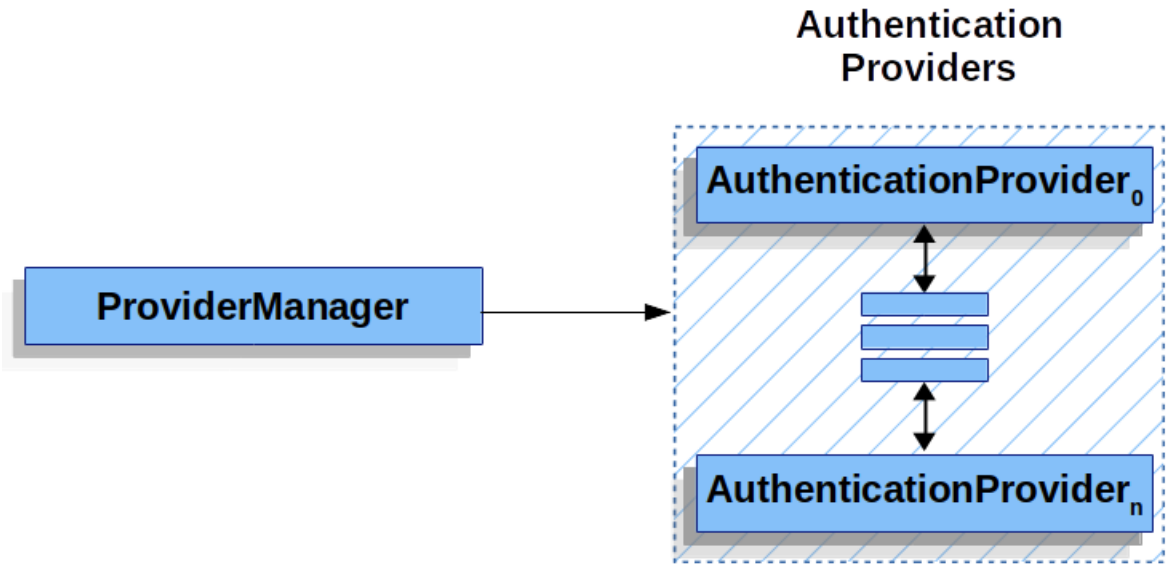
1.5 身份认证管理器 AuthenticationManager

AuthenticationManager 是定义 Spring Security 的过滤器如何执行身份验证的 API。然后由调用 AuthenticationManager 的控制器（即 Spring Security 的 Filterss）在 SecurityContextHolder 上设置返回的 Authentication。如果你没有与 Spring Security 的 Filterss 集成，你可以直接设置 SecurityContextHolder 并且不需要使用 AuthenticationManager

虽然 AuthenticationManager 的实现可以是任何东西，但最常见的实现是 ProviderManage

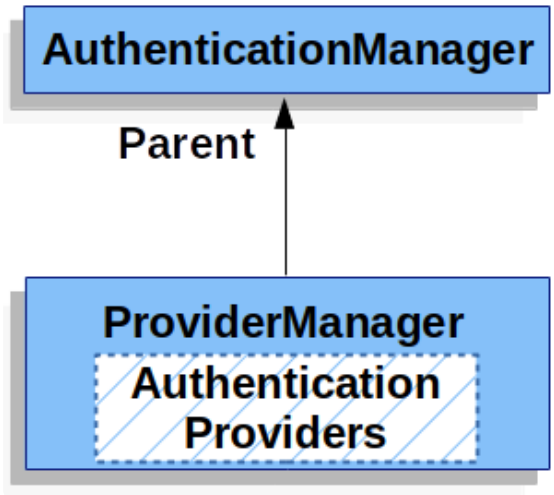
1.6 认证提供管理者 ProviderManager

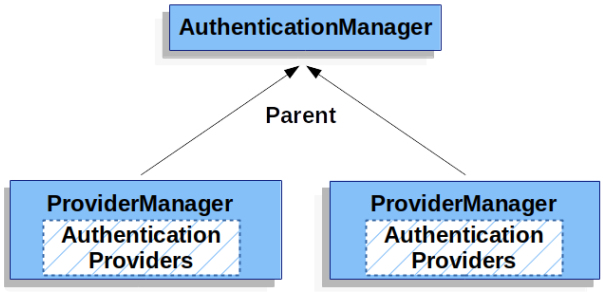
ProviderManager 是 AuthenticationManager 最常用的实现。 ProviderManager 委托给一个 AuthenticationProviders 列表。每个 AuthenticationProvider 都有机会指示身份验证应该成功、失败或指示它无法做出决定并允许下游 AuthenticationProvider 做出决定。如果没有配置的 AuthenticationProviders 可以进行身份验证，则身份验证将失败并出现 ProviderNotFoundException，这是一个特殊的 AuthenticationException，表明 ProviderManager 未配置支持传递给它的身份验证类型



实际上，每个 AuthenticationProvider 都知道如何执行特定类型的身份验证。例如，一个 AuthenticationProvider 可能能够验证用户名/密码，而另一个可能能够验证 SAML 断言。这允许每个 AuthenticationProvider 执行非常特定类型的身份验证，同时支持多种类型的身份验证并且只公开单个 AuthenticationManager bean

ProviderManager 还允许配置一个可选的父 AuthenticationManager，如果没有 AuthenticationProvider 可以执行身份验证，则可以参考该父 AuthenticationManager。父级可以是任何类型的 AuthenticationManager，但它通常是 ProviderManager 的一个实例（住过找不到匹配的认证提供者，就可以参考父级的认证提供管理器）





多个 ProviderManager 实例可能共享同一个父 AuthenticationManager。这在有多个 SecurityFilterChain 实例具有一些共同的身份验证（共享父 AuthenticationManager）但也有不同的身份验证机制（不同的 ProviderManager 实例）的情况下有些常见。

当您使用用户对象的缓存时，这可能会导致问题，例如，为了提高无状态应用程序的性能。如果 Authentication 包含对缓存中对象的引用（例如 UserDetails 实例）并且已删除其凭据，则将不再可能针对缓存的值进行身份验证。如果您使用缓存，则需要考虑到这一点。一个明显的解决方案是首先在缓存实现中或在创建返回的 Authentication 对象的 AuthenticationProvider 中制作对象的副本。或者，您可以禁用 ProviderManager 上的 eraseCredentialsAfterAuthentication 属性。

1.7 身份认证提供者 AuthenticationProvider

可以将多个 AuthenticationProviders 注入 ProviderManager。每个 AuthenticationProvider 执行特定类型的身份验证。例如：

- DaoAuthenticationProvider 支持基于用户名/密码的身份验证
- JwtAuthenticationProvider 支持对 JWT 令牌进行身份验证

1.8 使用 AuthenticationEntryPoint 请求凭据 Request Credentials with `AuthenticationEntryPoint`

`AuthenticationEntryPoint` 用于发送从客户端请求凭据的 HTTP 响应。

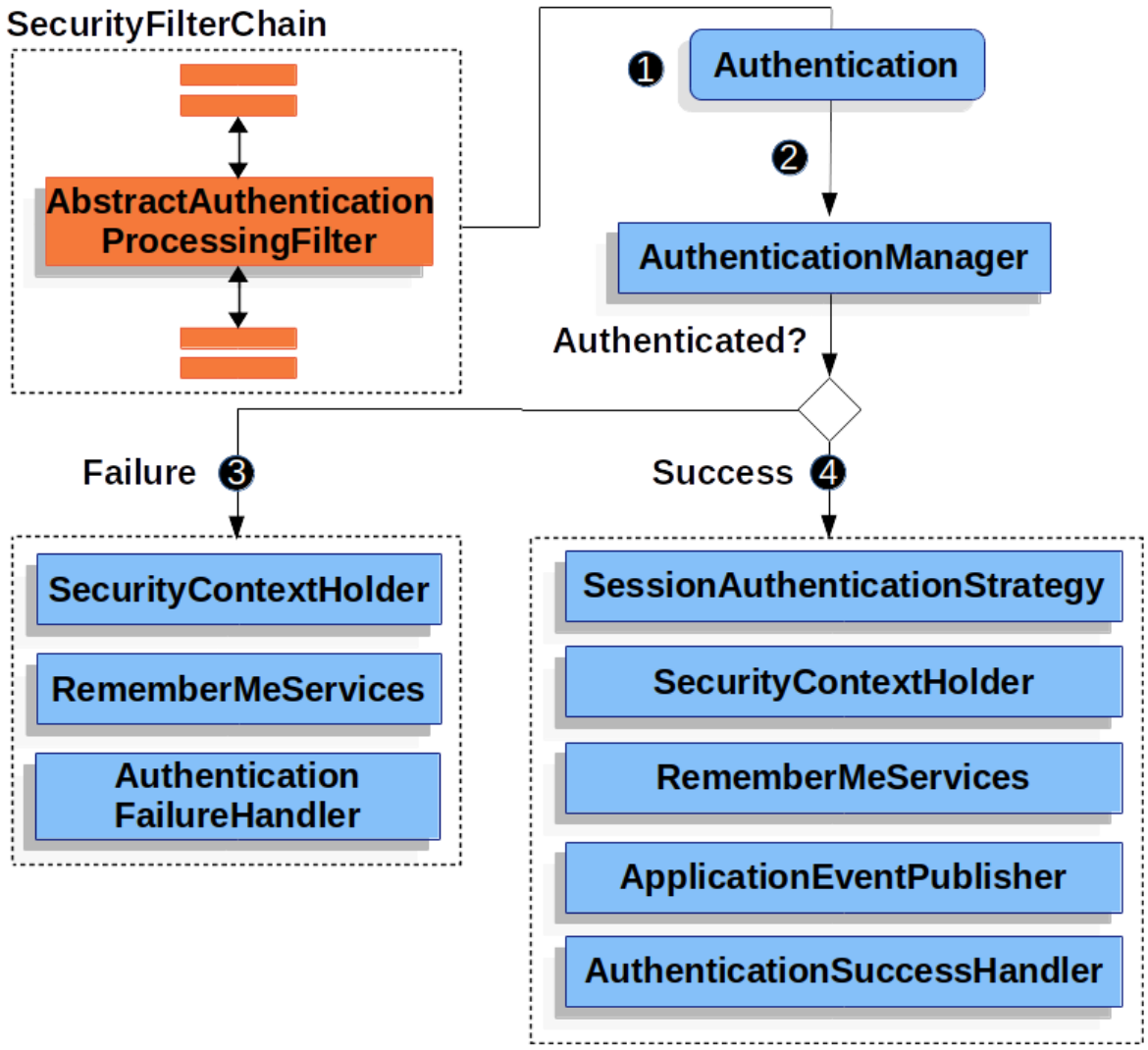
有时，客户端会主动包含凭据（例如用户名/密码）来请求资源。在这些情况下，Spring Security 不需要提供从客户端请求凭据的 HTTP 响应，因为它们已经包含在内。

在其他情况下，客户端将对他们无权访问的资源发出未经身份验证的请求。在这种情况下，AuthenticationEntryPoint 的实现用于从客户端请求凭据。AuthenticationEntryPoint 实现可能会重定向到登录页面，使用 WWW-Authenticate 标头等进行响应。

1.9 基本认证处理过滤器 AbstractAuthenticationProcessingFilter

AbstractAuthenticationProcessingFilter 用作验证用户凭据的基本过滤器。在认证凭证之前，Spring Security 通常使用 AuthenticationEntryPoint 请求凭据。

AbstractAuthenticationProcessingFilter 可以对提交给它的任何身份验证请求进行身份验证



1. 当用户提交他们的凭据时， AbstractAuthenticationProcessingFilter 从 HttpServletRequest 创建一个 Authentication 以进行身份验证。创建的 Authentication 类型取决于 AbstractAuthenticationProcessingFilter 的子类。例如，UsernamePasswordAuthenticationFilter 根据在 HttpServletRequest 中提交的用户名和密码创建 UsernamePasswordAuthenticationToken。
2. 接下来，将 Authentication 传递给 AuthenticationManager 进行身份验证
3. 如果身份验证失败
    - SecurityContextHolder 被清除
    - 调用 RememberMeServices.loginFail()。如果 `记住我` 没有配置，这是一个空操作。
    - AuthenticationFailureHandler 被调用

4. 如果认证成功

- SessionAuthenticationStrategy 收到新登录通知
- Authentication 在 SecurityContextHolder 上设置。稍后 SecurityContextPersistenceFilter 将 SecurityContext 保存到 HttpSession
- RememberMeServices.loginSuccess()。如果记住我没有配置，这是一个空操
- ApplicationEventPublisher 发布一个 InteractiveAuthenticationSuccessEvent(交互式认证成功事件)

二、认证机制 Authentication Mechanisms

- 认证机制 – 简述
  - Username and Password – 如何使用用户名/密码进行身份验证
  - OAuth 2.0 Login – OAuth 2.0 使用 OpenID Connect 登录和非标准 OAuth 2.0 登录（即 GitHub）
  - SAML 2.0 Login – SAML 2.0 登录
  - Central Authentication Server (CAS) – 中央身份验证服务器 (CAS) 支持
  - Remember Me – 如何记住一个过了会话有效期的用户
  - JAAS Authentication – 使用 JAAS 进行身份验证
  - OpenID – OpenID 身份验证（不要与 OpenID Connect 混淆）
  - Pre-Authentication Scenarios – 使用 SiteMinder 或 Java EE 安全等外部机制进行身份验证，但仍使用 Spring Security 进行授权和防止常见漏洞利用
  - X509 Authentication – X509 认证

2.1 用户名/密码验证 Username/Password Authentication

用户名/密码验证 Username/Password Authentication

2.2 会话管理 Session Management

会话管理 Session Management

2.3 Remember-Me

Remember-Me

2.4 OpenID 支持

命名空间支持 OpenID 登录，只需要在基于表单的登录之外，只需进行简单的更改

```xml
<http>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <openid-login />
</http>
```

然后，您应该向 OpenID 提供者（例如 myopenid.com）注册自己，并将用户信息添加到内存中的 <user-service>

```xml
<user name="https://jimi.hendrix.myopenid.com/" authorities="ROLE_USER" />
```

您应该能够使用 myopenid.com 站点登录以进行身份验证。也可以通过在 openid-login 元素上设置 user-service-ref 属性来选择一个特定的 UserDetailsService bean 来使用 OpenID。请注意，我们在上述用户配置中省略了密码属性，因为这组用户数据仅用于加载用户的权限。将在内部生成一个随机密码，以防止您在配置中的其他地方意外将此用户数据用作身份验证源