5 客户端认证 OAuth2ClientAuthenticationFilter

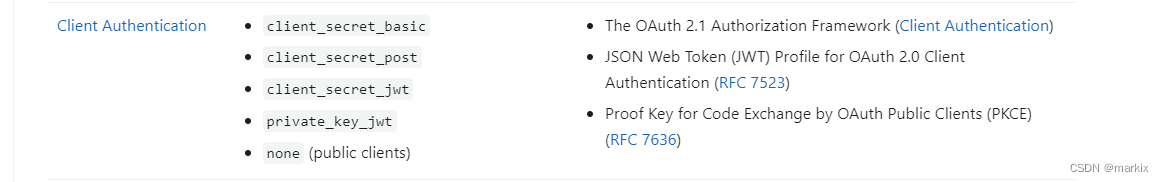
客户端认证

https://openid.net/specs/openid-connect-core-1\_0.html#ClientAuthentication

https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#token-endpoint-auth-method

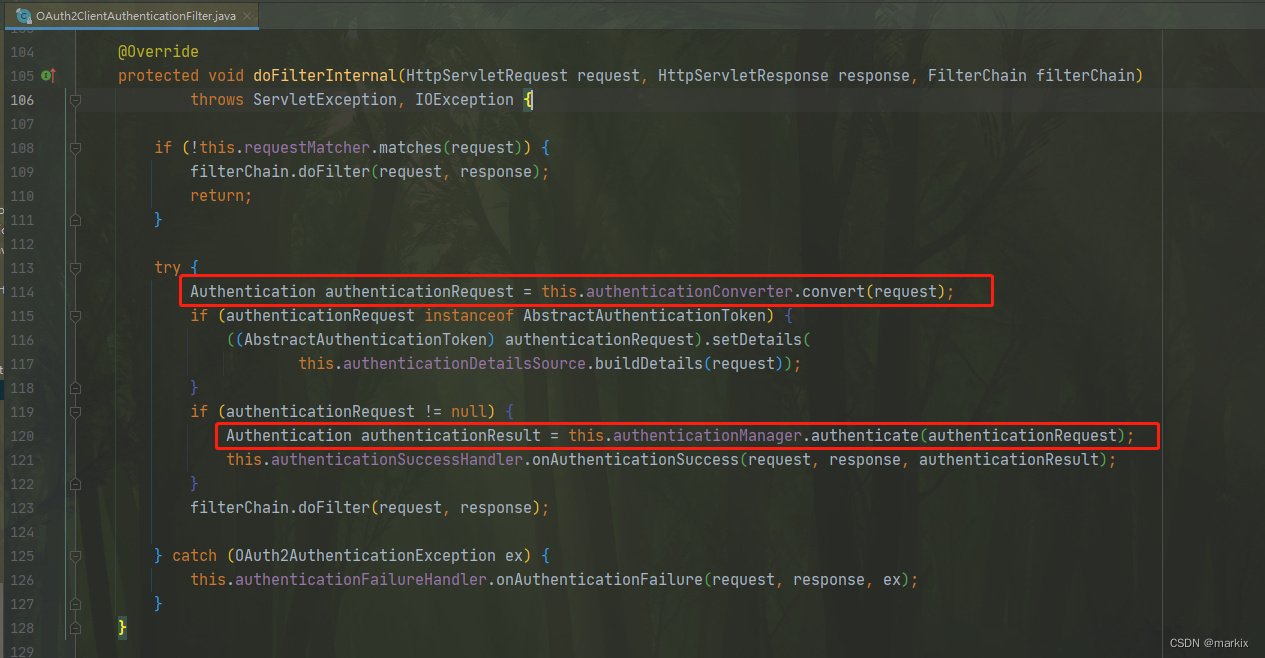
当访问 OAuth2 相关接口时（/oauth2/token、/oauth2/introspect、/oauth2/revoke），授权服务器需要进行客户端认证。

Spring Authorization Server 截至目前支持如下五种客户端认证方式：client\_secret\_basic、client\_secret\_post、client\_secret\_jwt、private\_key\_jwt、none (针对公共客户端)



OAuth2ClientAuthenticationFilter

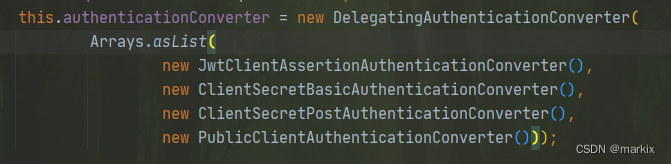
实现客户端认证的拦截器就是 OAuth2ClientAuthenticationFilter。 其核心代码如下：



其核心逻辑就是通过 authenticationConverter 从 request 中解析出客户端认证信息，构建成 Authentication，再通过 authenticationManager 对 Authentication 进行认证。

不同的解析方式实际上就代表不同的认证方式（传参不同）。

DelegatingAuthenticationConverter



authenticationConverter 的类型实际上是 DelegatingAuthenticationConverter，它持有一个 AuthenticationConverter 列表（不同的认证请求，其参数不同，所以会有不同的AuthenticationConverter实现类）。

DelegatingAuthenticationConverter 在解析请求时会遍历 AuthenticationConverter 列表，当某个 AuthenticationConverter 解析成功时，立即返回，这也能确定此请求是什么认证方式，后续再执行对应的认证逻辑。

ProviderManager

authenticationManager 的类型实际上是 ProviderManager，它持有一个 AuthenticationProvider 列表（不同的认证方式，其认证逻辑不同，所以会有不同的AuthenticationProvider实现类）。

ProviderManager 在执行认证时会遍历 AuthenticationProvider 列表，当某个 AuthenticationProvider 认证成功时，立马返回。

往 ProviderManager 中添加 AuthenticationProvider 的核心代码如下：（被框架封装了一层，显得不是很直观）

OAuth2ClientAuthenticationConfigurer#createDefaultAuthenticationProviders

小结

每种客户端认证方式实际就是一个 AuthenticationConverter 搭配一个 AuthenticationProvider 组合而成，一个负责解析请求参数，一个负责处理请求参数，执行认证逻辑！

题外话：上面说的这个实现模式，实际上是参照了 Spring Security 中用户认证的实现方式。毕竟 SAS 是基于 Spring Security 实现的。

在用户认证中，UsernamePasswordAuthenticationFilter 是直接将 request 解析成 UsernamePasswordAuthenticationToken，再交给 authenticationManager 进行认证（真正作用的是 DaoAuthenticationProvider）。

而 OAuth2ClientAuthenticationFilter 相当于做了更灵活的处理，支持多种解析方式和认证策略。

这也给我们一个启发，当我们需要做用户认证方式扩展时（比如增加个手机号登录），实际上也可以参照这种方式。

客户端认证方式

client\_secret\_basic

将 clientId 和 clientSecret 通过 ‘:’ 号拼接，并使用 Base64 进行编码得到一个字符串。将此编码字符串放到请求头(Authorization)去请求授权服务器接口。授权服务器会通过 client\_secret 进行验证。

传参：（请求头）

Authorization: Basic {Base64.encode(client\_id:client\_secret)}

核心类：

ClientSecretBasicAuthenticationConverter

ClientSecretAuthenticationProvider

client\_secret\_post

将 clientId 和 clientSecret 放到请求体（表单）中去请求授权服务器接口。授权服务器会通过 client\_secret 进行验证。

传参：（表单）

client\_id

client\_secret

核心类：

ClientSecretPostAuthenticationConverter

ClientSecretAuthenticationProvider

client\_secret\_jwt

客户端使用 client\_secret 通过 HMAC 算法生成 jwt，调用 授权服务器接口。授权服务器会通过 HMAC 算法验证 jwt。

传参：

client\_id

client\_assertion\_type：固定值 urn:ietf:params:oauth:client-assertion-type:jwt-bearer

client\_assertion：client生成的jwt，详见 authorization-server-clientauth.ClientJwtTest

核心类：

JwtClientAssertionAuthenticationConverter

JwtClientAssertionAuthenticationProvider

private\_key\_jwt

客户端自己维护密钥对，使用私钥生成 jwt，并将公钥暴露给 授权服务器。授权服务器通过客户端的公钥验证 jwt。

传参：

client\_assertion\_type 固定值 urn:ietf:params:oauth:client-assertion-type:jwt-bearer

client\_assertion client生成的jwt，详见 private\_key\_jwt\_client.ClientJwtTest

核心类：

JwtClientAssertionAuthenticationConverter

JwtClientAssertionAuthenticationProvider

none

PKCE(Proof Key for Code Exchange) 流程

公共客户端没有后端，所以授权成功后会携带code到前端，利用code就能获取token（即使有secret也是存在前端），这样code一旦泄露就能获取到token，很容易被盗用。

所以，公共客户端在每次获取授权时随机生成一个密文一起发给授权服务器，授权服务器缓存起来，

当公共客户端利用code获取token时需要携带密文对应的明文，授权服务器会计算明文，比对是否和原先的密文相同。相同则发放token。这样就避免了code被盗用。

访问授权页传参（授权码流程）：

code\_challenge client生成的密文，详见 authorization-server-clientauth.ClientPkceTest

code\_challenge\_method 固定值 S256

获取Token时传参：

code\_verifier 明文

client\_id

核心类：

PublicClientAuthenticationConverter

PublicClientAuthenticationProvider

小结

上文只是从大方向上介绍了客户端认证的不同方式，不同认证方式的实现以及细节各不相同，后续我打算结合一些具体示例来讲解，进而深入到源码进行分析。敬请期待。