

浅谈pigx对Spring Security的运用之pigx 的通行证——Access Token生成过程

Ver 1.0.1

PigX内部群资料

目录

浅谈pigx对Spring Security的运用之pigx的通行证——Access Token生成过程	1
写在前面	1
pigx的OAuth2.0认证流程详解	1
Spring Security OAuth核心类图解析	1
Spring Security OAuth的令牌生成过程——以pigx的登录过程为例	3
前期准备	3
开始学习	3
客户端选择	3
前端密码加解密讲解	3
构造请求参数	5
认证过程详解	7
更新日志	17

浅谈pigx对Spring Security的运用 之pigx的通行证——Access Token 生成过程 写在前面

pigx是什么?我想作者比我们有更多的发言权,下面的内容节选自[pig4cloud官网](#)介绍。

- 全网最新的Cloud 权限系统
- 基于Spring Boot 2.0.4.RELEASE
- 基于Finchley.SR1
- 网关基于 Spring Cloud Gateway
- 完整的OAuth 2.0 流程，资源服务器控制权限
- 基于 Spring Security OAuth 深度定制,学习OAuth2的不二选择

正如作者所言，pigx是一个脚手架，是一个Spring Cloud权限系统。因此我所理解的pigx就是对于Spring Security和Spring Cloud这两个框架的一种运用与实战。

既然pigx是对Spring Security OAuth的一种运用与实战，那么要想理解这种运用首先就需要知道Spring Security OAuth是怎么实现的。而群里的小伙伴有些由于不懂Spring Security OAuth总是不得其门而入（这个真的不是地图炮！），因此我写下这篇文章做一点微小的科普。虽然是以pigx为例，但是基础的知识部分，以及具体的实现对于Pig应该也是通用的。

这篇文章也会涉及到框架内部的一些知识，但是不会干巴巴地直接讲框架，本文的所有内容都将与pigx保持一致。

即：pigx实现的部分，本文会涉及，pigx没有实现的部分，本文将不会涉及。

其目的只有一个，帮助理解pigx的权限设计与实现。

让我们开始吧！

pigx的OAuth2.0认证流程详解 Spring Security OAuth核心类图解析

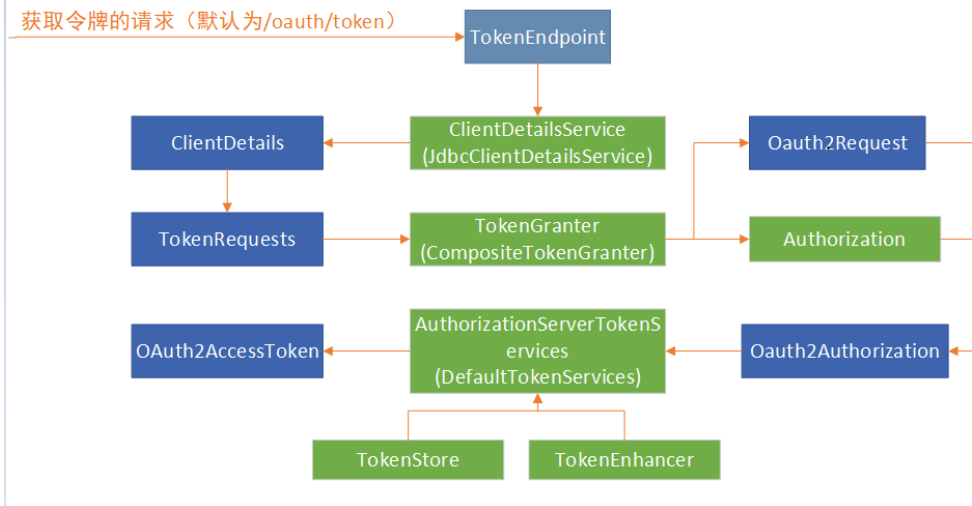
本文是一篇实用主义文章，只帮助理解pigx对于Spring Security OAuth2的运用，也就是说本文只会关注Spring Security对于OAuth2的实现，不会科普OAuth2。关于OAuth2是什么以及OAuth2的四种授权模式请移步[OAuth2官网](#)。

下面简单介绍一下关于Spring Security OAuth基本的原理。这也是理解pigx的第一步。

下面这张图涉及到了Spring OAuth的一些核心类和接口。

不多说，直接上图。

Spring Security Oauth核心源码



上图蓝色的方块代表执行过程中调用的具体的类，绿色的方块代表整个执行流程中调用的接口，绿色的括号中代表的是该接口调用的具体的实现类。

整个流程的入口点是在**TokenEndpoint**，由它来处理获取令牌请求，获取令牌请求默认是**/oauth/token**这个路径。

- 当TokenEndpoint收到请求时，它首先会调用ClientDetailsService,ClientDetailsService从名字上看就很可以知道是一个类似于UserDetailsService的接口，只不过UserDetailsService读取的是用户的信息，而ClientDetailsService读取的是第三方应用的信息。
- pigx会在登录请求头上带上Client的信息，而这个类就可以做到根据ClientId读取相应的配置信息。而ClientDetailsService读取到的信息都会封装到ClientDetails这个对象中。
- 同时，TokenEndpoint还会创建一个TokenRequests的对象，这个对象中封装了除了第三方应用以外的其他信息。比如说grant_type, scope, username, password(限密码模式)等等信息，而这些信息都是封装在TokenRequests里面的。同时，ClientDetails也会被放到TokenRequests中，因为第三方应用的信息也是令牌请求的一部分。
- 之后利用TokenRequests去调用一个叫做TokenGranter的令牌授权者的接口，这个接口其实是对四种不同的授权模式进行的一个封装。在这个接口里，它会根据请求传递过来的grant_type去挑一个具体的实现来执行令牌生成的逻辑。
- 不论采用哪种方式进行令牌的生成，在这个生成的过程中都会产生两个对象，一个是OAuth2Request,这个对象实际上是之前的ClientDetails和TokenRequests这两个对象的一个整合。另一个Authorization封装的实际上是当前授权用户的一些信息，也就是谁在进行授权行为，Authorization里封装的就是谁的信息。这里的用户信息是通过UserDetailsService进行读取的。
- OAuth2Request和Authorization这两个对象组合起来，会形成一个OAuth2Authorization对象，而这个最终产生的对象它的里面就包含了当前是哪个第三方应用在请求哪个用户以哪种授权模式（包括授权过程中的一些其他参数）进行授权，也就是这个对象会汇总之前的几个对象的信息都会封装到OAuth2Authorization这个对象中。
- 然后这个对象会传递到一个叫做AuthorizationServerTokenServices的接口的实现类，它拿到OAuth2Authorization中所有的信息之后最终会生成一个OAuth2的令牌OAuth2AccessToken。

Tips: 个性化token生成

AuthorizationServerTokenServices的接口的默认实现DefaultTokenServices中包含着其他两个接口的引用，TokenStore是用来定制token存储策略的，pigx用它实现了往redis里存放token,TokenEnhancer是token的增强器,pigx用它实现了返回的token的信息

的增强。

Spring Security OAuth的令牌生成过程——以pigx的登录过程为例

前期准备

光说不练假把式，下面我们结合pigx项目代码，来验证一下上面的过程。目前代码采用的是2018年9月1日最新的开发分支上的代码做演示，选择1.5.0的稳定版来复现以下的过程问题应该也不大，这部分逻辑最近几个版本也没啥变化。

首先启动核心的五个工程：注册中心，配置中心，认证中心，网关以及统一用户管理中心，同时启动前端工程以避免跨域问题。同时，为了避免影响测试结果请先在redis-cli上执行flushall或者flushdb清空redis。

个人建议萌新可以通过访问<http://127.0.0.1:9999/swagger-ui.html>通过swagger上面的Authorization按钮进行登录，我这里选择和作者视频里相同的curl的方式进行登录。

开始学习

客户端选择

我选择的应用是test,因为这个应用可以忽略验证码，关于这一块的配置可以参考pigx-gateway-dev.yml这个配置文件的ignore.clients属性，它可以接收一个想要忽略验证码的客户端的列表。至于为什么配置了这里的属性就可以忽略验证码也很简单。

网关工程有一个FilterIgnorePropertiesConfig类，这个类当配置文件里的ignore属性不为空时会生效。而验证码过滤器会对FilterIgnorePropertiesConfig中配置的客户端进行放行。如下所示：

```
// 终端设置不校验， 直接向下执行(1. 从请求参数中获取 2. 从header取)
String clientId = request.getQueryParams().getFirst("client_id");
if (StrUtil.isNotBlank(clientId)) {
    if (filterIgnorePropertiesConfig.getClients().contain(clientId)) {
        return chain.filter(exchange);
    }
}
```

能够获取到token的姿势有很多，我这里就选择一种类似pigx前端工程登录的方式。

前端密码加解密讲解

pigx大致的请求流程就是前端通过vue-router（模拟nginx）发送请求到后台网关，网关再根据配置的路由规则转发到各个微服务上。

根据pigx-gateway-dev.yml上的配置,可知经过认证中心的请求都需要经过两个过滤器，一个是验证码的处理，一个是将加密过的密码解密的过滤器。

```
routes:
# 认证中心
- id: pigx-auth
uri: lb://pigx-auth
predicates:
- Path=/auth/**
filters:
# 验证码处理
- ImageCodeGatewayFilter
# 前端密码解密
- PasswordDecoderFilter
- StripPrefix=1
```

验证码的过滤器已经被我们干掉了，密码解密的过滤器我这边不想处理，就直接网上搜个在线加

密的链接手动加密了,我用的是这个[在线AES加密解密、AES在线加密解密、AES encryption and decryption](#)。

AES作为对称加密的方式,前后端的加解密方式肯定是一致的,我们先看看后端的解密逻辑,之后再看看前端的加密逻辑做验证。后端的解密逻辑位于PasswordDecoderFilter这个类中,解密的代码不长,如下:

```

/*
 * AES/CBC/NoPadding 要求
 * 密钥必须是16字节长度的; Initialization vector (IV) 必须是16字节
 * 待加密内容的字节长度必须是16的倍数,如果不是16的倍数,就会出如下异常:
 * javax.crypto.IllegalBlockSizeException: Input length not multiple of 16 bytes
 *
 * 由于固定了位数,所以对于被加密数据有中文的,加、解密不完整
 *
 * 可以看到,在原始数据长度为16的整数n倍时,假如原始数据长度等于16*n,则使用NoPadding时加密后数据长度等于16*n,
 * 其它情况下加密数据长度等于16*(n+1)。在不足16的整数倍的情况下,假如原始数据长度等于16*n+m[其中m小于16],
 * 除了NoPadding填充之外的任何方式,加密数据长度都等于16*(n+1)。
 */

private static final String PASSWORD = "password";
private static final String KEY_ALGORITHM = "AES";
private static final String DEFAULT_CIPHER_ALGORITHM = "AES/CBC/NoPadding";
@Value("${security.encode.key:1234567812345678}")
private String encodeKey;

private static String decryptAES(String data, String pass) throws Exception {
    Cipher cipher = Cipher.getInstance(DEFAULT_CIPHER_ALGORITHM);
    SecretKeySpec keyspec = new SecretKeySpec(pass.getBytes(), KEY_ALGORITHM);
    IvParameterSpec ivspec = new IvParameterSpec(pass.getBytes());
    cipher.init(Cipher.DECRYPT_MODE, keyspec, ivspec);
    byte[] result = cipher.doFinal(Base64.decode(data.getBytes(CharsetUtil.UTF_8)));
    return new String(result, CharsetUtil.UTF_8);
}

```

Value注解是将配置文件中的security.encode.key的值作为参数注入,当这个参数不存在时,就会使用冒号后面的默认值1234567812345678,当然,在pigx的配置文件中这个参数值是肯定存在的,正是pigxpigxpigxpigx。这段解密逻辑说的就是采用AES的CBC加密方式,填充方式为零填充,密码和偏移量都是卸载配置文件中的pigxpigxpigxpigx。

接着我们去找一下前端的加密逻辑验证一下我们的判断,前端的加密逻辑位于ut il/ut il.js中,核心的加密代码如下:

```
/**
 * 加密处理
 */
export const encryption = (params) => {
  let {
    data,
    type,
    param,
    key
  } = params
  const result = JSON.parse(JSON.stringify(data))
  if (type === 'Base64') {
    param.forEach(ele => {
      result[ele] = btoa(result[ele])
    })
  } else {
    param.forEach(ele => {
      var data = result[ele]
      key = CryptoJS.enc.Latin1.parse(key)
      var iv = key
      // 加密
      var encrypted = CryptoJS.AES.encrypt(
        data,
        key, {
          iv: iv,
          mode: CryptoJS.mode.CBC,
          padding: CryptoJS.pad.ZeroPadding
        })
      result[ele] = encrypted.toString()
    })
  }
  return result
}
```

我们可以很显然地看到，加密采用的正式无填充的CBC模式，偏移量就是传入的加密密钥，零填充，这个和后端的代码是一致的，所以我们可以大胆地在上面地网站上填入参数了，填写的参数如下：

在线AES加密解密、AES在线加密解密、AES encryption and decryption

AES高级加密标准（英语：Advanced Encryption Standard，缩写：AES），在密码学中又称Rijndael加密法，是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的DES，已经被多方分析且广为全世界所使用。严格地说，AES和Rijndael加密法并不完全一样（虽然在应用中二者可以互换），因为Rijndael加密法可以支持更大范围的区块和密钥长度：AES的区块长度固定为128比特，密钥长度则可以是128、192或256比特；而Rijndael使用的密钥和区块长度可以是32位的整数倍，以128位为下限，256比特为上限。包括AES-ECB、AES-CBC、AES-CTR、AES-OFB、AES-CFB

AES加密模式: 填充: 数据块: 密钥: 偏移量: 输出: 字符集:

待加密、解密的文本:

AES加密、解密转换结果(base64了):

好了，密文：rKu1/348LvKp0rsVC06eCA==我们拿到了。

构造请求参数

接着我们开始吧。


```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -d "username=admin&password=rKu1/348LvKp0rsVC06eCA==&grant_type=password&scope=server" http://localhost:8000/auth/oauth/token
```

通过curl构造以上的链接，这个链接中包含着请求头的信息，请求头是一个"Basic"加一个空格加"clientId:clientSecret"base64化的一个**Authorization**字段，请求的参数里包含了grant_type和scope以及在password模式下必须的username和password字段。顺带一提，如果是windows中文语言的系统建议执行命令：

```
chcp 65001
```

将你的shell临时的更换为UTF-8的编码避免中文乱码的问题，虽然生成token的过程中不涉及中文文化的操作，但如果后期扩展了中文化可以避免问题。

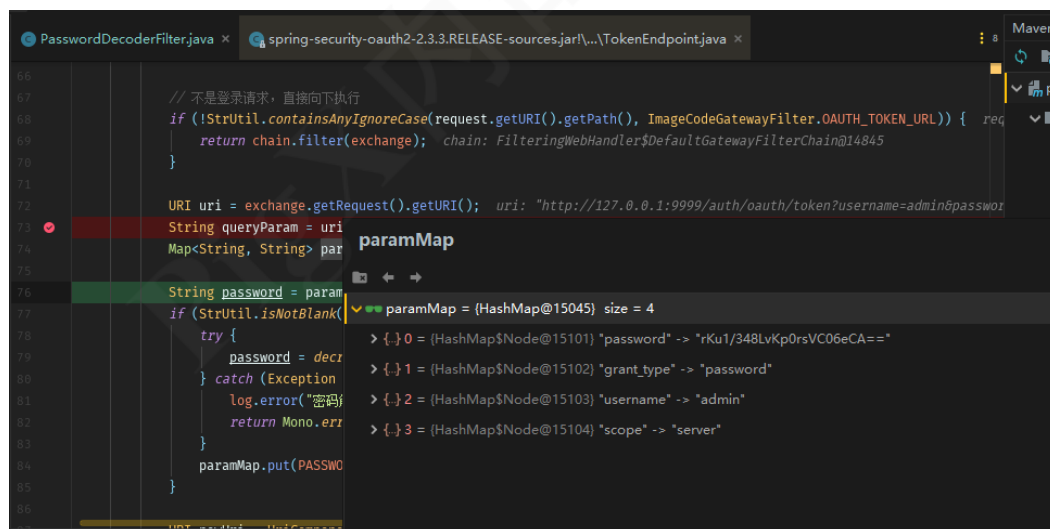
下面的是一个标准的POST请求并且在URL中携带参数的请求，但是这个请求不符合我们这边测试的要求，原因看下面的注意事项。

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -X POST http://localhost:8000/auth/oauth/token?username=admin&password=rKu1/348LvKp0rsVC06eCA==&grant_type=password&scope=server
```

所以我们把它改造一下。

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -X POST http://localhost:8000/auth/oauth/token?username=admin\&password=rKu1/348LvKp0rsVC06eCA%3D%3D\&grant_type=password\&scope=server
```

回车以后我们可以看到首先会经过网关的密码解密过滤器,并且参数经过我们的一通改造之后已经可以获得到正确的值了。

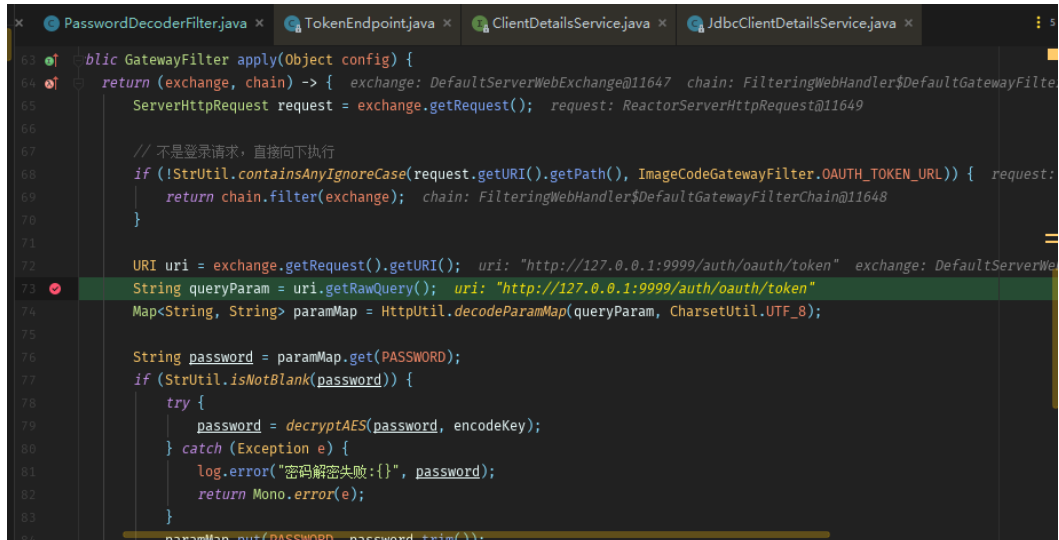


注意:

这个url.getRawQuery()方法会获取拼接到请求的URL后面的参数，这也是为什么我们之前构造的curl格式不是标准的把数据放入POST请求的请求体中的原因，标准的做法如下：

```
curl -H "Authorization:Basic dGVzdDp0ZXN0" -d "username=admin&password=rKu1/348LvKp0rsVC06eCA==&grant_type=password&scope=server" http://localhost:8000/auth/oauth/token
```

但是这种方式会有问题，在这套密码解密过滤器的机制下将会获取不到任何的参数。而会出现这个问题的原因也正是因为这73行的代码。



```

63 public GatewayFilter apply(Object config) {
64     return (exchange, chain) -> {
65         ServerHttpRequest request = exchange.getRequest();
66
67         // 不是登录请求，直接向下执行
68         if (!StrUtil.containsAnyIgnoreCase(request.getURI().getPath(), ImageCodeGatewayFilter.OAUTH_TOKEN_URL)) {
69             return chain.filter(exchange);
70         }
71
72         URI uri = exchange.getRequest().getURI();
73         String queryParam = uri.getRawQuery();
74         Map<String, String> paramMap = HttpUtil.decodeParamMap(queryParam, CharsetUtil.UTF_8);
75
76         String password = paramMap.get(PASSWORD);
77         if (StrUtil.isNotBlank(password)) {
78             try {
79                 password = decryptAES(password, encodeKey);
80             } catch (Exception e) {
81                 log.error("密码解密失败:{}", password);
82                 return Mono.error(e);
83             }
84         }
85         paramMap.put(PASSWORD, password.trim());
86     };
87 }

```

不过Spring的OAuth2.0本身是支持把数据放入POST请求体中的这种方式的。

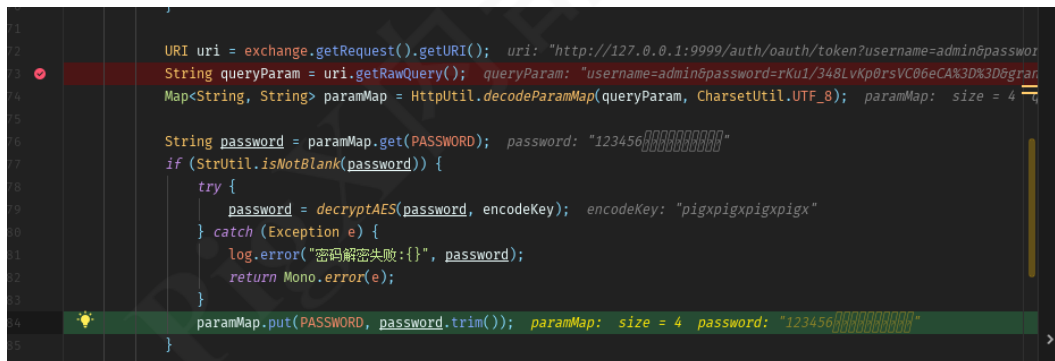
Tips:

URL中用于拼接多个参数的符号"&",在shell脚本中有特殊的意义（以daemon运行），所以要在"&"前加上反斜杠"\"转义一下。

Tips:

URL中"="这个符号具有特殊的含义,可能在服务器端无法获得正确的参数值，所以我们也用"%3D"转义一下。

继续我们刚才的过程，可以看到在密码解密过滤器接收到参数之后，就很简单了，整个密码解密过滤器的作用就是对登录请求中发送过来的加密密码进行解密的操作。我们直接看解密的结果。



```

71
72 URI uri = exchange.getRequest().getURI();
73 String queryParam = uri.getRawQuery();
74 Map<String, String> paramMap = HttpUtil.decodeParamMap(queryParam, CharsetUtil.UTF_8);
75
76 String password = paramMap.get(PASSWORD);
77 if (StrUtil.isNotBlank(password)) {
78     try {
79         password = decryptAES(password, encodeKey);
80     } catch (Exception e) {
81         log.error("密码解密失败:{}", password);
82         return Mono.error(e);
83     }
84     paramMap.put(PASSWORD, password.trim());
85 }

```

解密的结果对于不足16位的密码会填充空格到16位，trim之后我们就可以看到得到我们真正想要的密码"123456"了，这也侧面验证了我们之前生成的密码策略是正确的。

认证过程详解

经过上面的一通操作，我们已经拿到了获取token的一些必要的请求了。

clientId,clientSecret,grant_type,username,password,scope,终于可以带着我们的参数深入源码啦！

这里结合上文提到的核心类图来看效果更好

上文提过,OAuth2.0的认证的入口点位于TokenEndPoint。我们也可以看到，代码确实已经进来了。

```
86 @RequestMapping(value = "/oauth/token", method=RequestMethod.POST)
87 public ResponseEntity<OAuth2AccessToken> postAccessToken(Pincipal principal, @RequestParam principal: "org.springframework.security.authentication.UsernamePasswordAuthenticationToken" principal,
88 Map<String, String> parameters) throws HttpRequestMethodNotSupportedException { parameters: size = 4
89
90 if (!principal instanceof Authentication) { principal: "org.springframework.security.authentication.UsernamePasswordAuthenticationToken"
91 throw new InsufficientAuthenticationException(
92 "There is no client authentication. Try adding an appropriate authentication filter.");
93 }
94
95 String clientId = getClientId(principal);
96 ClientDetails authenticatedClient = getClientDetailsService().loadClientByClientId(clientId);
97
98 TokenRequest tokenRequest = getOAuth2RequestFactory().createTokenRequest(parameters, authenticatedClient);
99
100 if (clientId != null && !clientId.equals("")) {
101 // Only validate the client details if a client authenticated during this
102 // request.
103 if (!clientId.equals(tokenRequest.getClientId())) {
104 // double check to make sure that the client ID in the token request is the same as that in the
105 // request.
106 throw new InvalidClientException("Invalid client authentication.");
107 }
108 }
109
110 return new ResponseEntity<OAuth2AccessToken>(getOAuth2RequestFactory().createTokenRequest(parameters, authenticatedClient), HttpStatus.OK);
111 }
```

我们可以看到这个类上有一个@RequestMapping注解，它来处理/oauth/token的POST请求。

1. 进来之后的第一步，就是在代码的95行，获取请求头中的clientId。
2. 然后在96行调用getClientDetailsService().loadClientByClientId(clientId)方法获取整个第三方应用的详细配置。

扩展：

第三方应用有非常丰富的配置项,如图所示：

```
90 if (!principal instanceof Authentication) {
91 throw new InsufficientAuthenticationException(
92 "There is no client authentication. Try adding an appropriate authentication filter.");
93 }
94
95 String clientId = get
96 ClientDetails authent
97
98 TokenRequest tokenReq
99
100 if (clientId != null
101 // Only validate
102 // request.
103 if (!clientId.equ
104 // double che
105 // authentica
106 throw new Inv
107 }
```

authenticatedClient

- authenticatedClient = (BaseClientDetails@12414) "BaseClientDetails [clientId=test, clientSecret={noop}te ... View
- clientId = "test"
- clientSecret = "{noop}test"
- scope = (LinkedHashSet@12423) size = 1
- resourceIds = (Collections\$EmptySet@12424) size = 0
- authorizedGrantTypes = (LinkedHashSet@12425) size = 2
- registeredRedirectUris = null
- autoApproveScopes = (HashSet@12426) size = 1
- authorities = (Collections\$EmptyList@12427) size = 0
- accessTokenValiditySeconds = null
- refreshTokenValiditySeconds = null
- additionalInformation = (LinkedHashMap@12428) size = 0

Console | Frames | Variables

XNIO-2 task-1*@12,180 in

postAccessToken:98, TokenEr

具体的参数的意义可以看[spring-oauth-server 数据库表说明](#)

3. 在拿到客户端的信息之后在代码的98行通过传递进来的参数和查询出来的第三方应用信息构建TokenRequest。

创建TokenRequest的代码很简单，如下：

```

public TokenRequest createTokenRequest(Map<String, String> requestParameters, ClientDetails authenticatedClient) {

    String clientId = requestParameters.get(OAuth2Utils.CLIENT_ID);
    if (clientId == null) {
        // if the clientId wasn't passed in in the map, we add pull it from the authenticated client object
        clientId = authenticatedClient.getClientId();
    }
    else {
        // otherwise, make sure that they match
        if (!clientId.equals(authenticatedClient.getClientId())) {
            throw new InvalidClientException("Given client ID does not match authenticated client");
        }
    }
    String grantType = requestParameters.get(OAuth2Utils.GRANT_TYPE);

    Set<String> scopes = extractScopes(requestParameters, clientId);
    TokenRequest tokenRequest = new TokenRequest(requestParameters, clientId, scopes, grantType);

    return tokenRequest;
}

```

所以其实它就干了一件事，校验传递进来clientId和查询出来的clientId,如果匹配的话，就根据之前传递进来的clientId和和查询出来的第三方应用构建TokenRequest。

然后我们就拿到TokenRequest了，后面的代码很简单了：

```

if (clientId != null && !clientId.equals("")) {
    // Only validate the client details if a client authenticated during this
    // request.
    if (!clientId.equals(tokenRequest.getClientId())) {
        // double check to make sure that the client ID in the token request is the same as that in the
        // authenticated client
        throw new InvalidClientException("Given client ID does not match authenticated client");
    }
}
if (authenticatedClient != null) {
    oAuth2RequestValidator.validateScope(tokenRequest, authenticatedClient);
}
if (!StringUtils.hasText(tokenRequest.getGrantType())) {
    throw new InvalidRequestException("Missing grant type");
}
if (tokenRequest.getGrantType().equals("implicit")) {
    throw new InvalidGrantException("Implicit grant type not supported from token endpoint");
}

if (isAuthCodeRequest(parameters)) {
    // The scope was requested or determined during the authorization step
    if (!tokenRequest.getScope().isEmpty()) {
        logger.debug("Clearing scope of incoming token request");
        tokenRequest.setScope(Collections.<> emptySet());
    }
}

if (isRefreshTokenRequest(parameters)) {
    // A refresh token has its own default scopes, so we should ignore any added by the factory here.
    tokenRequest.setScope(OAuth2Utils.parseParameterList(parameters.get(OAuth2Utils.SCOPE)));
}

OAuth2AccessToken token = getTokenGranter().grant(tokenRequest.getGrantType(), tokenRequest);
if (token == null) {
    throw new UnsupportedGrantTypeException("Unsupported grant type: " + tokenRequest.getGrantType());
}

return getResponse(token);

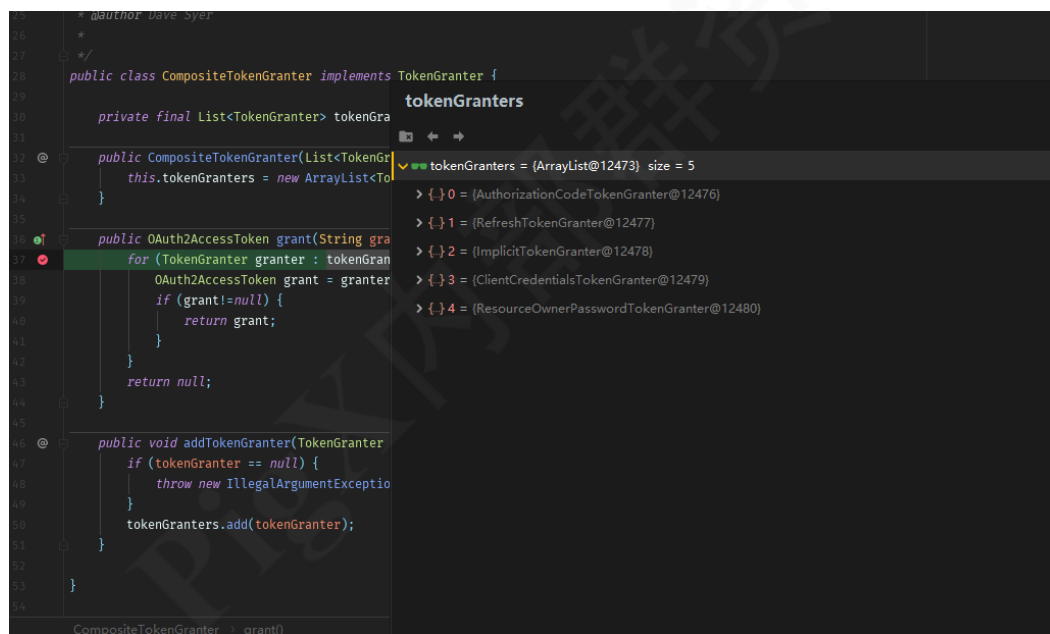
```

无非就是对下面这些参数的校验：

- clientId:是否有值，值是否和查询结果匹配
- scope:请求的一些授权内容，所请求的授权必须是第三方应用可以发送的授权集合的子集，否则无法通过校验)
- grant_type:必须显式指定按照哪种授权模式获取令牌
- 判断传递的授权模式是否是简化模式，如果是简化模式也会抛异常。因为简化模式其实是对授权码模式的一种简化:在用户的第一步的授权行为的时候就直接返回令牌,所以是不会有调用请求令牌服务的机会的
- 判断是不是授权码模式,因为授权码模式包含两个步骤，在授权码模式中发出的令牌中拥有的权限不是由发令牌的请求决定的，而是在发令牌之前的授权的请求里就已经决定好了。因此它会对请求过来的scope进行置空操作，然后根据之前发出去的授权码里的权限重新设置你的scope,因此它根本不会使用请求令牌的这个请求中携带的scope参数。
- 之后判断是不是刷新令牌的操作,应为刷新令牌的操作有自己的scope，所以也会进行重新设置scope的操作。

经过一系列的校验之后，最终TokenRequest会在132行传递给TokenGranter，然后由granter产生最终的accessToken。之后直接将accessToken写入响应里就可以了。

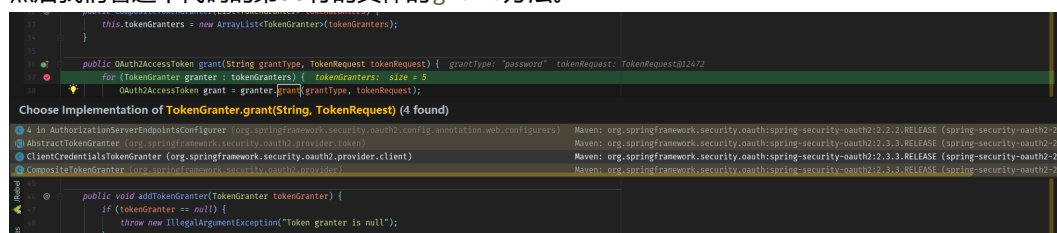
TokenGranter中总共封装了四种授权模式加一个刷新令牌的操作，我们看看其中的一些细节。



CompositeTokenGranter中有一个集合，这个集合里封装着的就是五个会产生令牌的操作。

它会对遍历这五种情况，并根据之前请求中携带的grant_type在五种情况中挑一种进行最终的accessToken的生成。

然后我们看这个代码的第38行的具体的grant方法。



```

53 public OAuth2AccessToken grant(String grantType, TokenRequest tokenRequest) { grantType: "password" tokenRequest: TokenRequest@12420
54
55     if (!this.grantType.equals(grantType)) { grantType: "authorization_code" grantType: "password"
56         return null;
57     }
58
59     String clientId = tokenRequest.getClientId();
60     ClientDetails client = clientDetailsService.loadClientByClientId(clientId);
61     validateGrantType(grantType, client);
62
63     if (logger.isDebugEnabled()) {
64         logger.debug("Getting access token for: " + clientId);
65     }
66
67     return getAccessToken(client, tokenRequest);
68 }

```

首先在org.springframework.security.oauth2.provider.token.AbstractTokenGranter中判断当前携带的授权类型和这个类所支持的授权类型是否匹配，如果不匹配就返回空值，如果匹配的话就进行令牌的生成操作。

59到第63行是重新获取一下clientId和客户端信息跟授权类型再做一个校验,67行的getAccessToken方法会产生最终的一个令牌。

这个方法也非常简单:

```

protected OAuth2AccessToken getAccessToken(ClientDetails client, TokenRequest tokenRequest) {
    return tokenServices.createAccessToken(getOAuth2Authentication(client, tokenRequest));
}

```

它实际上就是对tokenServices的一个调用，而tokenServices其实就是从37行我们可以看到其实就是AuthorizationServerTokenServices。这个类要想创建accessToken需要一个OAuth2Authentication对象，所以createAccessToken中包含了一个方法getOAuth2Authentication。

这个方法不同的授权模式会有不同的实现。

```

protected OAuth2AccessToken getAccessToken(ClientDetails client, TokenRequest tokenRequest) {
    return tokenServices.createAccessToken(getOAuth2Authentication(client, tokenRequest));
}

protected OAuth2Authentication getOAuth2Authentication(ClientDetails client, TokenRequest tokenRequest) {
    Choose Implementation of AbstractTokenGranter.getOAuth2Authentication(ClientDetails, TokenRequest) (3 found)
    AuthorizationCodeTokenGranter (org.springframework.security.oauth2.provider.code) Maven: org.springframework.security.oauth:spring-security-oauth2-2.3.3.RELEASE (spring-security-oauth2-2.3.3.RELEASE.jar)
    ImplicitTokenGranter (org.springframework.security.oauth2.provider.implicit) Maven: org.springframework.security.oauth:spring-security-oauth2-2.3.3.RELEASE (spring-security-oauth2-2.3.3.RELEASE.jar)
    ResourceOwnerPasswordTokenGranter (org.springframework.security.oauth2.provider.password) Maven: org.springframework.security.oauth:spring-security-oauth2-2.3.3.RELEASE (spring-security-oauth2-2.3.3.RELEASE.jar)

    Collection<String> authorizedGrantTypes = clientDetails.getAuthorizedGrantTypes();
    if (authorizedGrantTypes != null && !authorizedGrantTypes.isEmpty()) {
        if (!authorizedGrantTypes.contains(grantType)) {
            throw new InvalidClientException("Unauthorized grant type: " + grantType);
        }
    }
}

```

在Spring Security OAuth核心类图解析中我们已经知道最终产生的OAuth2Authentication包含两部分信息,一部分是请求中的一些信息，另一部分是根据请求获取的授权用户的信息。而在不同的授权模式下获取授权用户的信息的方式是不同的，比如说pigx所使用的密码模式就是使用请求中携带的用户名和密码来获取当前授权用户中的授权信息,而在授权码模式的两个步骤中是根据第一步发出授权码的同时会记录相关用户的信息，之后对第二步进行授权的时候根据第三方应用请求过来的授权码再读取该授权码对应的用户信息。所以getOAuth2Authentication对于不同的授权类型有不同的实现。

我们以pigx所使用的密码模式继续下面的流程。密码模式对应的是org.springframework.security.oauth2.provider.password.ResourceOwnerPasswordTokenGranter

```

@Override
protected OAuth2Authentication getOAuth2Authentication(ClientDetails client, TokenRequest tokenRequest) {
    Map<String, String> parameters = new LinkedHashMap<String, String>(tokenRequest.getRequestParameters());
    String username = parameters.get("username");
    String password = parameters.get("password");
    // Protect from downstream leaks of password
    parameters.remove("password");

    Authentication userAuth = new UsernamePasswordAuthenticationToken(username, password);
    ((AbstractAuthenticationToken) userAuth).setDetails(parameters);
    try {
        userAuth = authenticationManager.authenticate(userAuth);
    } catch (AccountStatusException ase) {
        // Covers expired, locked, disabled cases (mentioned in section 5.2, draft 31)
        throw new InvalidGrantException(ase.getMessage());
    } catch (BadCredentialsException e) {
        // If the username/password are wrong the spec says we should send 400/invalid grant
        throw new InvalidGrantException(e.getMessage());
    }
    if (userAuth == null || !userAuth.isAuthenticated()) {
        throw new InvalidGrantException("Could not authenticate user: " + username);
    }

    OAuth2Request storedOAuth2Request = getRequestFactory().createOAuth2Request(client, tokenRequest);
    return new OAuth2Authentication(storedOAuth2Request, userAuth);
}

```

而这个方法我们可以看到它其实就是根据所请求的用户名和密码去创建 UsernamePasswordAuthenticationToken,然后传递给authenticationManager做认证,在这个认证过程中它会去调用 com.pig4cloud.pigx.common.security.service.PigxUserDetailsServiceImpl 的 loadUserByUsername 方法,根据用户名和密码去读取用户的信息,之后我们其实就已经拿到 Authorization 的信息,而 OAuth2Request 根据第85行我们可以知道是根据传进来的第三方应用详情和 tokenRequest 产生出来的,而86行的 OAuth2Authentication 也是由 OAuth2Request 和 Authorization 这两个对象拼接起来的。而拼接的方式就是调用 org.springframework.security.oauth2.provider.request.DefaultOAuth2RequestFactory 的 createOAuth2Request 方法。

```

public OAuth2Request createOAuth2Request(ClientDetails client, TokenRequest tokenRequest) {
    return tokenRequest.createOAuth2Request(client);
}

```

这个方法最终会创建一个由 clientDetails 和 tokenRequest 组合而成的 OAuth2Request。

```

DefaultOAuth2RequestFactory.java x TokenRequest.java x OAuth2RequestFactory.java x PigxUserDetailsServiceImpl.java x TokenEndpoint.java

public void setScope(Collection<String> scope) { super.setScope(scope); }

/**
 * Set the Request Parameters on this authorization request, which represent the original request parameters and
 * should never be changed during processing. The map passed in is wrapped in an unmodifiable map instance.
 *
 * @see AuthorizationRequest#setRequestParameters
 *
 * @param requestParameters
 */
public void setRequestParameters(Map<String, String> requestParameters) {
    super.setRequestParameters(requestParameters);
}

public OAuth2Request createOAuth2Request(ClientDetails client) {
    Map<String, String> requestParameters = getRequestParameters();
    HashMap<String, String> modifiable = new HashMap<>(requestParameters);
    // Remove password if present to prevent leaks
    modifiable.remove("password");
    modifiable.remove("client_secret");
    // Add grant type so it can be retrieved from OAuth2Request
    modifiable.put("grant_type", grantType);
    return new OAuth2Request(modifiable, client.getClientId(), client.getAuthorities(), approved: true, this.getScope(),
        client.getResourceIds(), redirectUri: null, responseType: null, extensionProperties: null);
}

```

拿到 OAuth2Request 就可以去生成 OAuth2Authentication 了。

而 OAuth2Authentication 就是

org.springframework.security.oauth2.provider.token.AbstractTokenGranter 第71到73行最终传递进去生成 accessToken 的对象。

而 OAuth2Authentication 生成成功之后进行返回的话就可以执

行AuthorizationServerTokenServices的createAccessToken方法，而一旦这个access token生成成功并写入响应进行返回那么整个流程也就结束了，最终我们就拿到了想要的访问令牌。

```
protected OAuth2AccessToken getAccessToken(ClientDetails client, TokenRequest tokenRequest) {
    return tokenServices.createAccessToken(getOAuth2Authentication(client, tokenRequest));
}
```

具体创建accessToken的代码，我们需要仔细读一

读org.springframework.security.oauth2.provider.token.DefaultTokenServices的createAccessToken方法。

```
@Transactional
public OAuth2AccessToken createAccessToken(OAuth2Authentication authentication) throws AuthenticationException {
    OAuth2RefreshToken existingAccessToken = tokenStore.getAccessToken(authentication);
    if (existingAccessToken != null) {
        if (existingAccessToken.isExpired()) {
            if (existingAccessToken.getRefreshToken() != null) {
                refreshToken = existingAccessToken.getRefreshToken();
                // The token store could remove the refresh token when the
                // access token is removed, but we want to
                // be sure...
                tokenStore.removeRefreshToken(refreshToken);
            }
            tokenStore.removeAccessToken(existingAccessToken);
        }
        else {
            // Re-store the access token in case the authentication has changed
            tokenStore.storeAccessToken(existingAccessToken, authentication);
            return existingAccessToken;
        }
    }

    // Only create a new refresh token if there wasn't an existing one
    // associated with an expired access token.
    // Clients might be holding existing refresh tokens, so we re-use it in
    // the case that the old access token
    // expired.
    if (refreshToken == null) {
```

首先这个类一进来就会尝试在tokenStore中获取accessToken,因为同一个用户只要令牌没过期那么再次请求令牌的时候会把之前发送的令牌再次发还。因此一开始就会找当前用户已经存在的令牌。

如果已经发送的令牌不为空,那么会在87行判断当前的令牌是否已经过期,如果令牌过期了,那么就会在tokenStore里把accessToken和refreshToken一起删掉,如果令牌没过期,那么就把这个没过期的令牌重新再存一下。因为可能用户是使用另外的方式来访问令牌的,比如说一开始用授权码模式,后来用密码模式,而这两种模式需要存的信息是不一样的,所以这个令牌要重新store一次。之后直接返回这个不过期的令牌。

如果令牌已经过期了或者说这个是第一次请求,令牌压根没生成,就会走下面的逻辑。

```

104 // Only create a new refresh token if there wasn't an existing one
105 // associated with an expired access token.
106 // Clients might be holding existing refresh tokens, so we re-use it in
107 // the case that the old access token
108 // expired.
109 if (refreshToken == null) { refreshToken: null
110     refreshToken = createRefreshToken(authentication);
111 }
112 // But the refresh token itself might need to be re-issued if it has
113 // expired.
114 else if (refreshToken instanceof ExpiringOAuth2RefreshToken) {
115     ExpiringOAuth2RefreshToken expiring = (ExpiringOAuth2RefreshToken) refreshToken;
116     if (System.currentTimeMillis() > expiring.getExpiration().getTime()) {
117         refreshToken = createRefreshToken(authentication);
118     }
119 }
120
121 OAuth2AccessToken accessToken = createAccessToken(authentication, refreshToken);
122 tokenStore.storeAccessToken(accessToken, authentication);
123 // In case it was modified
124 refreshToken = accessToken.getRefreshToken();
125 if (refreshToken != null) {
126     tokenStore.storeRefreshToken(refreshToken, authentication);
127 }
128 return accessToken;
129
130 }
```

首先看看刷新的令牌有没有,如果刷新的令牌没有的话,那么创建一枚刷新的令牌。然后在121行根据authentication, refreshToken创建accessToken。而这个创建accessToken的方法也非常简单:


```
private OAuth2AccessToken createAccessToken(OAuth2Authentication authentication,
OAuth2RefreshToken refreshToken) {
    DefaultOAuth2AccessToken token = new DefaultOAuth2AccessToken(UUID.randomUUID()
().toString());
    int validitySeconds = getAccessTokenValiditySeconds(authentication.getOAuth2Request());
    if (validitySeconds > 0) {
        token.setExpiration(new Date(System.currentTimeMillis() + (validitySeconds * 1000L)));
    }
    token.setRefreshToken(refreshToken);
    token.setScope(authentication.getOAuth2Request().getScope());

    return accessTokenEnhancer != null ? accessTokenEnhancer.enhance(token, authentication) : token;
}
```

OAuth2AccessToken其实就是用UUID创建一个accessToken,然后把过期时间,刷新令牌和scope这些OAuth协议规定的必须要存在的参数设置上,设置完了以后它会判断是否存在tokenEnhancer,如果存在tokenEnhancer它就会按照定制的tokenEnhancer增强生成出来的token。

拿到返回的令牌之后,在122行tokenStore会把拿到的令牌存起来,然后拿refreshToken存起来,最后把生成的令牌返回回去。

于是我们就获取到了令牌。

```
c:\Windows\system32
curl -H "Authorization: Basic d9x2p02m03" -X POST http://localhost:8000/auth/token?username=admin&password=xKul/34RtVg0rsW006eC&30/30/30?grant_type=password&scope=server
{"access_token":"94aff8e9-1e33-42e4-adb-c9370e64db0c3","token_type":"bearer","refresh_token":"420971a9-ea0f-4d87-b750-764e955b6fe","expires_in":43199,"scope":"server","license":"made by pigx"}
```

扩展:

pigx对于查询JdbcClientDetailsService中的查询语句做了一些增强,为什么要做增强,下面来简单分析一下。

首先我们看用于处理认证的pigx-auth工程的WebSecurityConfigurer这个配置类中创建的是如下的一个PasswordEncoder:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

这个类是Spring Security5新出的一个类,列出了SpringSecurity5支持的所有的密码匹配器。

```
public static PasswordEncoder createDelegatingPasswordEncoder() {
    String encodingId = "bcrypt";
    Map<String, PasswordEncoder> encoders = new HashMap<>();
    encoders.put(encodingId, new BCryptPasswordEncoder());
    encoders.put("ldap", new LdapShaPasswordEncoder());
    encoders.put("MD4", new Md4PasswordEncoder());
    encoders.put("MD5", new MessageDigestPasswordEncoder("MD5"));
    encoders.put("noop", NoOpPasswordEncoder.getInstance());
    encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
    encoders.put("scrypt", new SCryptPasswordEncoder());
    encoders.put("SHA-1", new MessageDigestPasswordEncoder("SHA-1"));
    encoders.put("SHA-256", new MessageDigestPasswordEncoder("SHA-256"));
    encoders.put("sha256", new StandardPasswordEncoder());

    return new DelegatingPasswordEncoder(encodingId, encoders);
}
```

具体创建密码编码器的过程也展示了要求的新密码的格式：

```
public class DelegatingPasswordEncoder implements PasswordEncoder {
    // 密码匹配器id的前缀
    private static final String PREFIX = "{";
    // 密码匹配器id的后缀
    private static final String SUFFIX = "}";
    // 密码匹配器的类型
    private final String idForEncode;
    private final PasswordEncoder passwordEncoderForEncode;
    private final Map<String, PasswordEncoder> idToPasswordEncoder;

    /**
     * 密码的格式匹配不上就会报错，相信每个人港升级的时候都经历过There is no PasswordEncoder mapped for the id "null"的绝望吧！
     */
    private class UnmappedIdPasswordEncoder implements PasswordEncoder {

        @Override
        public String encode(CharSequence rawPassword) {
            throw new UnsupportedOperationException("encode is not supported");
        }

        @Override
        public boolean matches(CharSequence rawPassword,
            String prefixEncodedPassword) {
            String id = extractId(prefixEncodedPassword);
            throw new IllegalArgumentException("There is no PasswordEncoder mapped for the id \"" + id + "\"");
        }
    }
}
```

这个类就要求了密码必须符合带上{“具体的解密器id”，最后根据这个id去找密码匹配器匹配，clientSecret最终也是要参与解码的，所以它也需要带上{“id”，clientSecret我们并不需要做什么艰深的加密，所以使用原始密码就行，这个解密器就是NoOpPasswordEncoder，它的id从上文我们看到是“noop”，也就是说数据库里的clientSecret要想在Spring Security5下正常工作，clientId应该是testclientSecret应该是{noop}test，但是我们可以看到数据库里存储的都是test/test那为什么进行解密的时候没有抛出PasswordEncoder mapped for the id “null”的异常呢？

原因很简单。

在com.pig4cloud.pigx.common.core.constant.SecurityConstants查询客户端信息的语句中，我们可以看到{noop}这个字段在查询出来注入JdbcClientDetailsService之前，作者已经利用Mysql的连接函数帮我们拼接好了。

```
String CLIENT_FIELDS = "client_id, CONCAT('{noop}',client_secret) as client_secret, resource_ids, scope, "
    + "authorized_grant_types, web_server_redirect_uri, authorities, access_token_validity, "
    + "refresh_token_validity, additional_information, autoapprove";
```

同样巧妙的设定也体现在了用户密码的加密上。

在upms模块的UserController模块中，作者显式指定了密码解密器为BCryptPasswordEncoder。

```

@Slf4j
@Service
@AllArgsConstructor
public class SysUserServiceImpl extends ServiceImpl<SysUserMapper, SysUser> implements SysUserService {
    private static final PasswordEncoder ENCODER = new BCryptPasswordEncoder();
    private final SysMenuService sysMenuService;
    private final SysUserMapper sysUserMapper;
    private final SysRoleService sysRoleService;
    private final SysUserRoleService sysUserRoleService;
    private final SysDeptRelationService sysDeptRelationService;
    // 其他代码省略
}

```

Tips:

为什么都声明成final级别变量，要结合上面的lombok的@AllArgsConstructor注解来看，其实就是为了使用Lombok的黑科技进行构造器注入，这也是Spring 5 推荐的一种注入方式。

但是很显然，这种密码解密器直接参与进Spring Security5的执行流程又会报喜闻乐见的There is no PasswordEncoder mapped for the id “null” 错误，那么为什么没报呢？见代码的com.pig4cloud.pigx.common.security.service.PigxUserDetailsServiceImpl类的getUserDetails方法：

```

/**
 * 构建userdetails
 *
 * @param result 用户信息
 * @return
 */
private UserDetails getUserDetails(R<UserInfo> result) {
    if (result == null || result.getData() == null) {
        throw new UsernameNotFoundException("用户不存在");
    }

    UserInfo info = result.getData();
    Set<String> dbAuthsSet = new HashSet<>();
    if (ArrayUtil.isNotEmpty(info.getRoles())) {
        // 获取角色
        Arrays.stream(info.getRoles()).forEach(role -> dbAuthsSet.add(SecurityConstants.ROLE + role));
        // 获取资源
        dbAuthsSet.addAll(Arrays.asList(info.getPermissions()));
    }
    Collection<? extends GrantedAuthority> authorities
        = AuthorityUtils.createAuthorityList(dbAuthsSet.toArray(new String[0]));
    SysUser user = info.getSysUser();
    boolean enabled = StrUtil.equals(user.getDelFlag(), CommonConstant.STATUS_NORMAL);
    // 构造security用户

    return new PigxUser(user.getUserId(), user.getDeptId(), user.getUsername(), SecurityConstants.BCRYPT + user.getPassword(), enabled, true, true, true, authorities);
}

```

见构造security用户的部分，作者在构造Security的User对象进行认证之前，进行了和处理clientSecret类似的操作，手动拼接了“{bcrypt}”的字符。

作者的这两个操作，据我个人推测,应该是为了保证和Spring Security 4.x的密码格式的兼容性，

隐藏密码变更的细节。

更新日志

- 1.0.1

解决pdf渲染可能带来的bash命令异常的问题

解决在Acrobat或者Adobe Pdf Reader下可能存在的"读取本文档时出现问题(14)"错误
在不影响阅读体验的情况下添加内部群资料水印，请大家尽情地鄙视我，然后更新你们的文档

PigX内部群资料