

1.本文介绍的认证流程范围

本文主要对从用户发起获取token的请求（/oauth/token），到请求结束返回token中间经过的几个关键点进行说明。

2.认证会用到的相关请求

注：所有请求均为post请求。

- **获取access_token请求（/oauth/token）**

请求所需参数：client_id、client_secret、grant_type、username、password

```
http://localhost/oauth/token?  
client_id=demoClientId&client_secret=demoClientSecret&grant_type=password  
&username=demoUser&password=50575tyL86xp290380t1
```

- **检查token是否有效请求（/oauth/check_token）**

请求所需参数：token

```
http://localhost/oauth/check_token?token=f57ce129-2d4d-4bd7-1111-  
f31ccc69d4d1
```

- **刷新token请求（/oauth/token）**

请求所需参数：grant_type、refresh_token、client_id、client_secret

其中grant_type为固定值：grant_type=refresh_token

```
http://localhost/oauth/token?  
grant_type=refresh_token&refresh_token=fbde81ee-f419-42b1-1234-  
9191f1f95be9&client_id=demoClientId&client_secret=demoClientSecret
```

2.认证核心流程

注：文中介绍的认证服务器端token存储在Reids，用户信息存储使用数据库，文中会包含相关的部分代码。

2.1.获取token的主要流程：

加粗内容为每一步的重点，不想细看的可以只看加粗内容：

1. 用户发起获取token的请求。
2. 过滤器会**验证path**是否是认证的请求/oauth/token，如果为false，则直接返回没有后续操作。
3. 过滤器通过clientId查询**生成一个Authentication对象**。
4. 然后通过username和生成的Authentication对象**生成一个UserDetails对象**，并检查用户是否存在。
5. **以上全部通过会进入地址/oauth/token**，即TokenEndpoint的postAccessToken方法中。
6. postAccessToken方法中会**验证Scope**，然后**验证是否是refreshToken请求**等。
7. 之后调用AbstractTokenGranter中的grant方法。
8. grant方法中调用AbstractUserDetailsAuthenticationProvider的authenticate方法，**通过username和Authentication对象来检索用户是否存在**。
9. 然后通过DefaultTokenServices类**从tokenStore中获取OAuth2AccessToken对象**。
10. 然后将**OAuth2AccessToken对象**包装进响应流返回。

2.2.刷新token（refresh token）的流程

刷新token（refresh token）的流程与获取token的流程只有⑨有所区别：

- 获取token调用的是AbstractTokenGranter中的getAccessToken方法，然后调用tokenStore中的getAccessToken方法获取token。
- 刷新token调用的是RefreshTokenGranter中的getAccessToken方法，然后使用tokenStore中的refreshAccessToken方法获取token。

2.3.tokenStore的特点

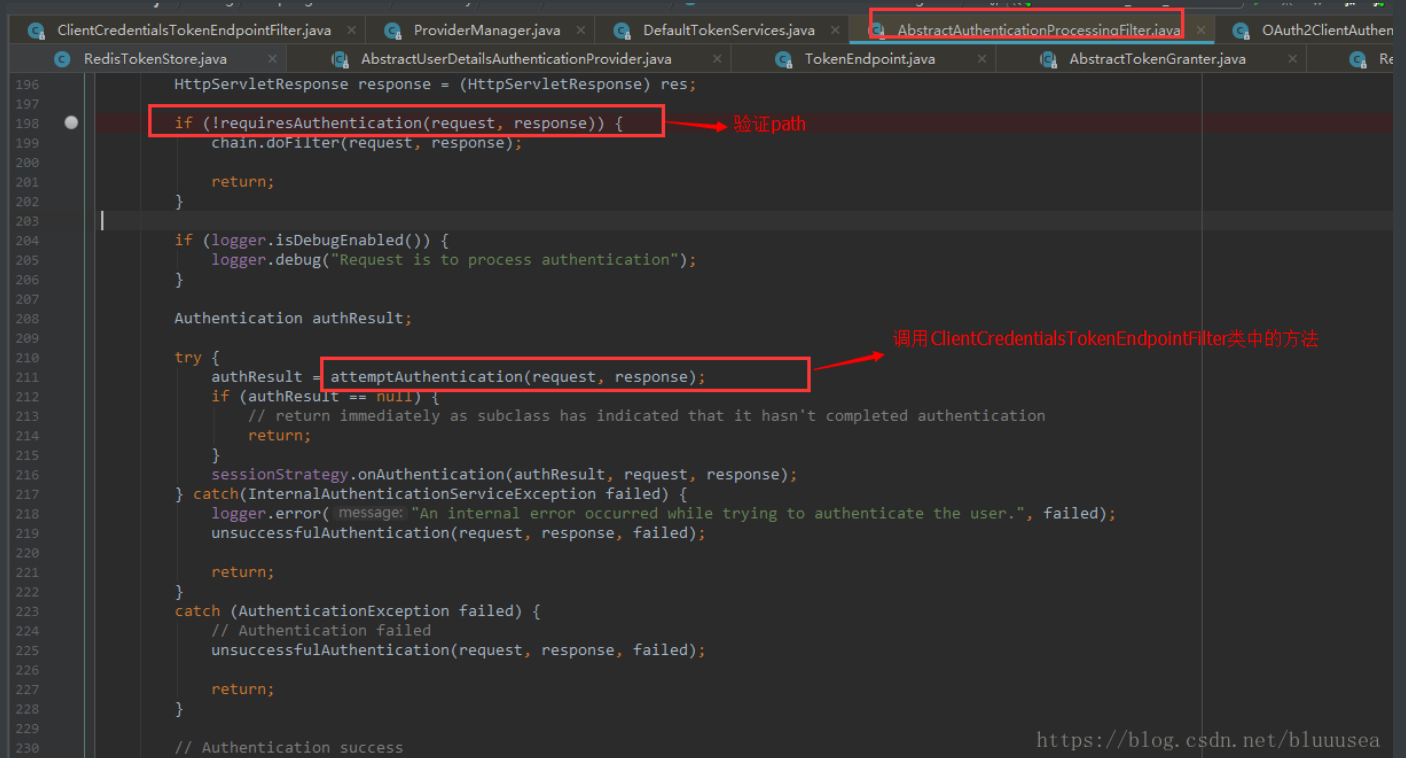
tokenStore通常为自定义实现，一般放置在缓存或者数据库中。此处可以利用自定义tokenStore来实现多种需求，如：

- 同一用户每次获取token，获取到的都是同一个token，只有token失效后才会获取新token。
- 同一用户每次获取token都生成一个完成周期的token并且保证每次生成的token都能够使用（多点登录）。
- 同一用户每次获取token都保证只有最后一个token能够使用，之前的token都设为无效（单点token）。

3.获取token的详细流程（代码截图）

3.1.代码截图梳理流程

1. 一个比较重要的过滤器



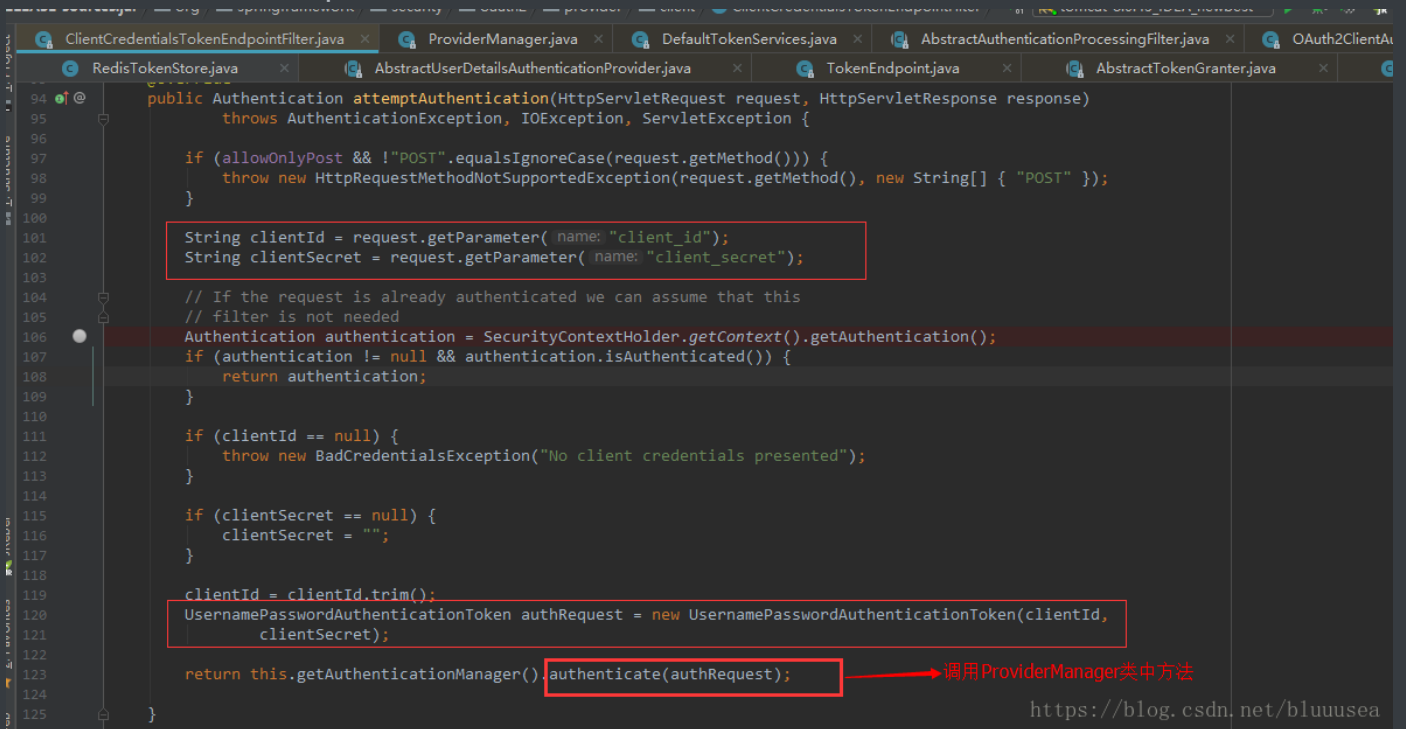
This screenshot shows the `AbstractAuthenticationProcessingFilter` class. Two red boxes highlight key logic points:

- Line 198:** `if (!requiresAuthentication(request, response)) { chain.doFilter(request, response); return; }`. An annotation "验证path" (Verify path) points to this block.
- Line 211:** `authResult = attemptAuthentication(request, response);`. An annotation "调用ClientCredentialsTokenEndpointFilter类中的方法" (Call the method in the ClientCredentialsTokenEndpointFilter class) points to this line.

The code continues with session strategy handling, logging, and exception catching for authentication failures.

<https://blog.csdn.net/bluusea>

2. 此处是①中的attemptAuthentication方法



This screenshot shows the `attemptAuthentication` method. Two red boxes highlight key logic points:

- Lines 101-102:** `String clientId = request.getParameter("client_id"); String clientSecret = request.getParameter("client_secret");`. This block extracts client credentials from the request.
- Line 122:** `return this.getAuthenticationManager().authenticate(authRequest);`. An annotation "调用ProviderManager类中方法" (Call the method in the ProviderManager class) points to this line.

The method also includes checks for POST requests, existing authentication, and handling of missing or invalid client credentials.

<https://blog.csdn.net/bluusea>

3. 此处是②中调用的authenticate方法

```
ClientCredentialsTokenEndpointFilter.java x ProviderManager.java x DefaultTokenServices.java x AbstractAuthenticationProcessingFilter.java x OAuth2ClientAuthen
RedisTokenStore.java x AbstractUserDetailsAuthenticationProvider.java x TokenEndpoint.java x AbstractTokenGranter.java x Re
38 * @throws AuthenticationException if authentication fails.
39 */
40 public Authentication authenticate(Authentication authentication) throws AuthenticationException {
41     Class<? extends Authentication> toTest = authentication.getClass();
42     AuthenticationException lastException = null;
43     Authentication result = null;
44     boolean debug = Logger.isDebugEnabled();
45
46     for (AuthenticationProvider provider : getProviders()) {
47         if (!provider.supports(toTest)) {
48             continue;
49         }
50
51         if (debug) {
52             Logger.debug("Authentication attempt using " + provider.getClass().getName());
53         }
54
55         try {
56             result = provider.authenticate(authentication);
57
58             if (result != null) {
59                 copyDetails(authentication, result);
60                 break;
61             }
62         } catch (AccountStatusException e) {
63             prepareException(e, authentication);
64             // SEC-546: Avoid polling additional providers if auth failure is due to invalid account status
65             throw e;
66         } catch (InternalAuthenticationServiceException e) {
67             prepareException(e, authentication);
68             throw e;
69         } catch (AuthenticationException e) {
70             lastException = e;
71         }
72     }
73 }
```

遍历所有的Provider，只要有一个返回结果就返回，
此处是AbstractUserDetailsAuthenticationProvider类返回结果

<https://blog.csdn.net/bluusea>

4.此处是③中调用的AbstractUserDetailsAuthenticationProvider类的authenticate方法

```
ClientCredentialsTokenEndpointFilter.java x ProviderManager.java x DefaultTokenServices.java x AbstractAuthenticationProcessingFilter.java x OAuth2ClientAuthen
AbstractUserDetailsAuthenticationProvider.java x DaoAuthenticationProvider.java x TokenEndpoint.java x AbstractTokenGranter.java x Refresh
113 Assert.notNull(this.messages, message: "A message source must be set");
114 doAfterPropertiesSet();
115 }
116
117 public Authentication authenticate(Authentication authentication) throws AuthenticationException {
118     Assert.isInstanceOf(UsernamePasswordAuthenticationToken.class, authentication,
119         messages.getMessage(code: "AbstractUserDetailsAuthenticationProvider.onlySupports",
120             defaultMessage: "Only UsernamePasswordAuthenticationToken is supported"));
121
122     // Determine username
123     String username = (authentication.getPrincipal() == null) ? "NONE_PROVIDED" : authentication.getName();
124
125     boolean cacheWasUsed = true;
126     UserDetails user = this.userCache.getUserFromCache(username);
127
128     if (user == null) {
129         cacheWasUsed = false;
130
131         try {
132             user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);
133         } catch (UsernameNotFoundException notFound) {
134             logger.debug("User '" + username + "' not found");
135
136             if (hideUserNotFoundExceptions) {
137                 throw new BadCredentialsException(messages.getMessage(
138                     code: "AbstractUserDetailsAuthenticationProvider.badCredentials", defaultMessage: "Bad credentials"));
139             } else {
140                 throw notFound;
141             }
142         }
143
144         Assert.notNull(user, message: "retrieveUser returned null - a violation of the interface contract");
145     }
146
147     try {
148         preAuthenticationChecks.check(user);
149         additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authentication);
150     } catch (AuthenticationException exception) {
151         if (cacheWasUsed) {
152             // ...
153         }
154     }
155 }
```

检索用户是否存在，
此处调用的是DaoAuthenticationProvider类

验证ClientId,ClientSecret是否正确
也是调用的
DaoAuthenticationProvider类

<https://blog.csdn.net/bluusea>

5.此处是④中调用的DaoAuthenticationProvider类的retrieveUser方法

```
ClientCredentialsTokenEndpointFilter.java x ProviderManager.java x DefaultTokenServices.java x AbstractAuthenticationProcessingFilter.java x OAuth2ClientAuthenticationProcess
AbstractUserDetailsAuthenticationProvider.java x DaoAuthenticationProvider.java x ClientDetailsUserDetailsService.java x TokenEndpoint.java x AbstractTokenG
95 }
96
97 protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication)
98     throws AuthenticationException {
99     UserDetails loadedUser;
100
101     try {
102         loadedUser = this.getUserDetailsService().loadUserByUsername(username);
103     } catch (UsernameNotFoundException notFound) {
104         if (authentication.getCredentials() != null) {
105             String presentedPassword = authentication.getCredentials().toString();
106             passwordEncoder.isPasswordValid(userNotFoundEncodedPassword, presentedPassword, (salt: null));
107         }
108         throw notFound;
109     } catch (Exception repositoryProblem) {
110         throw new InternalAuthenticationServiceException(repositoryProblem.getMessage(), repositoryProblem);
111     }
112
113     if (loadedUser == null) {
114         throw new InternalAuthenticationServiceException(
115             "UserDetailsService returned null, which is an interface contract violation");
116     }
117     return loadedUser;
118 }
119
```

此处调用 ClientDetailsUserDetailsService 类

<https://blog.csdn.net/bluusea>

6. 此处为⑤中调用的 ClientDetailsUserDetailsService 类的 loadUserByUsername 方法，执行完后接着返回执行④之后的方法

```
ClientCredentialsTokenEndpointFilter.java x ProviderManager.java x DefaultTokenServices.java x AbstractAuthenticationProcessingFilter.java x OAuth2ClientAuthent
AbstractUserDetailsAuthenticationProvider.java x DaoAuthenticationProvider.java x ClientDetailsUserDetailsService.java x TokenEndpoint.java x AbstractTokenG
private final ClientDetailsService clientDetailsService;
private String emptyPassword = "";

public ClientDetailsUserDetailsService(ClientDetailsService clientDetailsService) {
    this.clientDetailsService = clientDetailsService;
}

/**
 * @param passwordEncoder the password encoder to set
 */
public void setPasswordEncoder>PasswordEncoder passwordEncoder) { this.emptyPassword = passwordEncoder.encode( rawPassword: ""); }

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    ClientDetails clientDetails;
    try {
        clientDetails = clientDetailsService.loadClientByClientId(username);
    } catch (NoSuchClientException e) {
        throw new UsernameNotFoundException(e.getMessage(), e);
    }
    String clientSecret = clientDetails.getClientSecret();
    if (clientSecret == null || clientSecret.trim().length() == 0) {
        clientSecret = emptyPassword;
    }
    return new User(username, clientSecret, clientDetails.getAuthorities());
}
}
```

此处中的 clientDetailsService 一般注入自己的 service

<https://blog.csdn.net/bluusea>

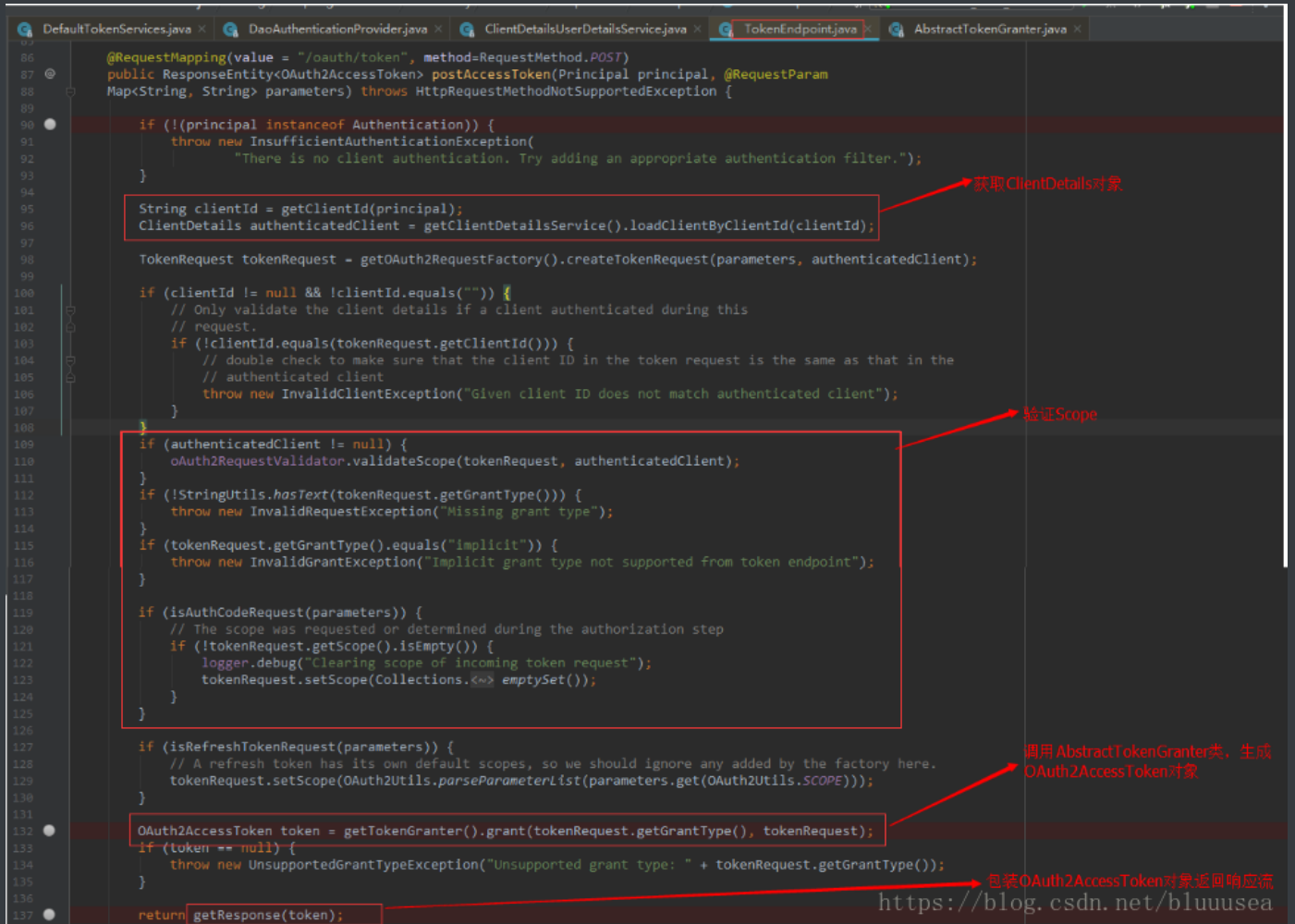
7. 此处为④中调用的 DaoAuthenticationProvider 类的 additionalAuthenticationChecks 方法，此处执行完则主要过滤器执行完毕，后续会进入 /oauth/token 映射的方法。

```
ClientCredentialsTokenEndpointFilter.java x ProviderManager.java x DefaultTokenServices.java x AbstractAuthenticationProcessingFilter.java x OAuth2ClientAuthent
AbstractUserDetailsAuthenticationProvider.java x DaoAuthenticationProvider.java x ClientDetailsUserDetailsService.java x TokenEndpoint.java x AbstractTokenG
77 // deprecated
78
79 @deprecation
80 protected void additionalAuthenticationChecks(UserDetails userDetails,
81     UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {
82     Object salt = null;
83
84     if (this.saltSource != null) {
85         salt = this.saltSource.getSalt(userDetails);
86     }
87
88     if (authentication.getCredentials() == null) {
89         logger.debug("Authentication failed: no credentials provided");
90
91         throw new BadCredentialsException(messages.getMessage(
92             code: "AbstractUserDetailsAuthenticationProvider.badCredentials", (defaultMessage: "Bad credentials"), userDetails));
93     }
94
95     String presentedPassword = authentication.getCredentials().toString();
96
97     if (!passwordEncoder.isPasswordValid(userDetails.getPassword(), presentedPassword, salt)) {
98         logger.debug("Authentication failed: password does not match stored value");
99
100         throw new BadCredentialsException(messages.getMessage(
101             code: "AbstractUserDetailsAuthenticationProvider.badCredentials", (defaultMessage: "Bad credentials"), userDetails));
102     }
103 }
104
```

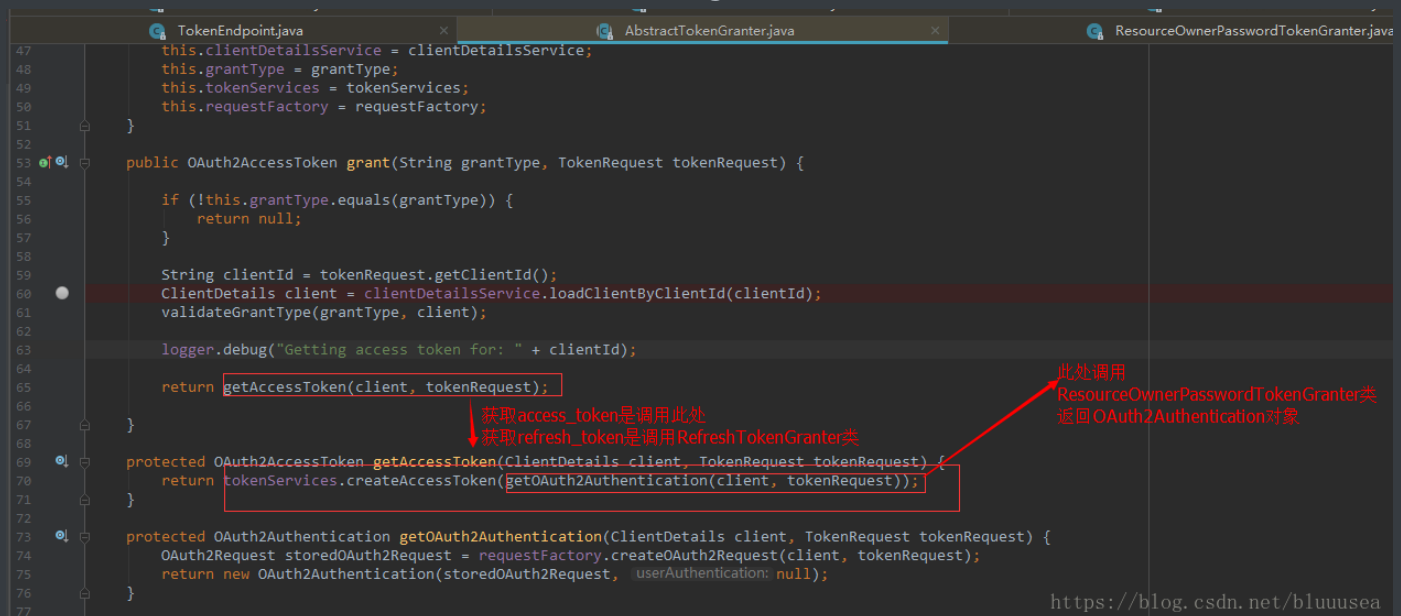
此处 password 为 ClientSecret

<https://blog.csdn.net/bluusea>

8. 此处进入 /oauth/token 映射的 TokenEndpoint 类的 postAccessToken 方法

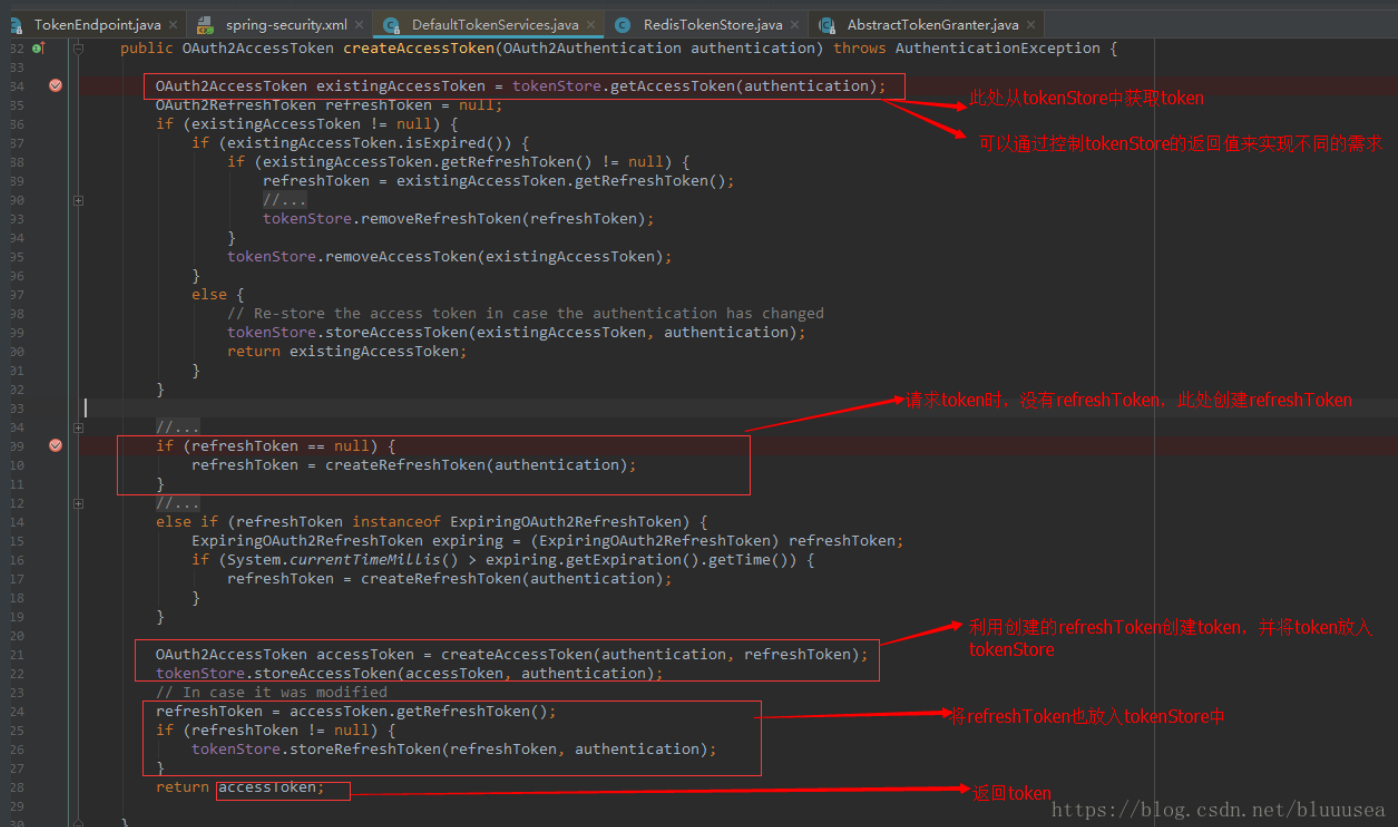


9.此处为⑧中调用的AbstractTokenGranter类的grant方法

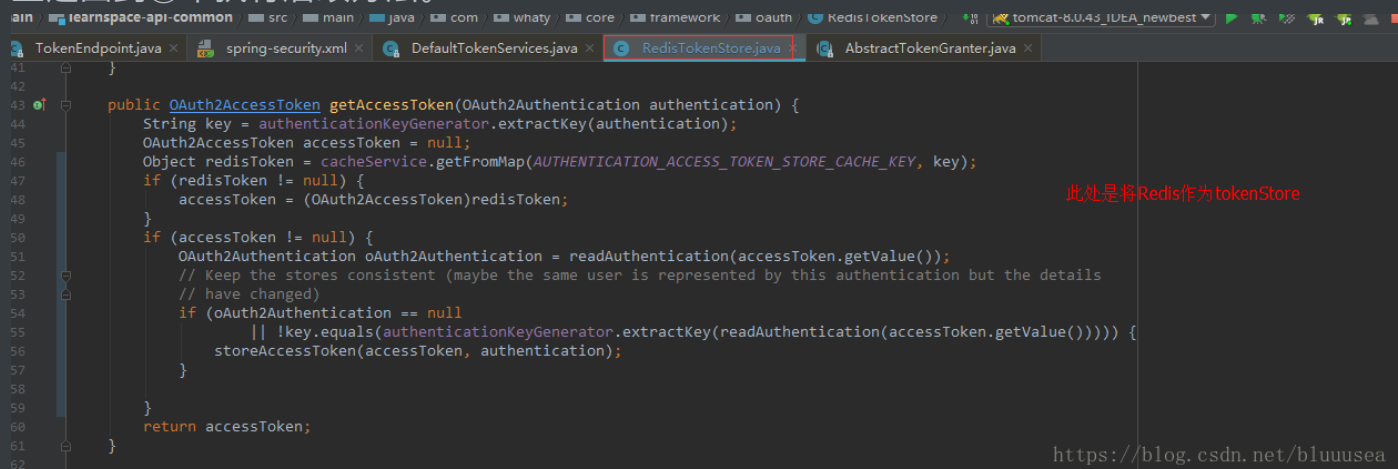


10.此处为⑨中调用的ResourceOwnerPasswordTokenGranter类中的getOAuth2Authentication方法

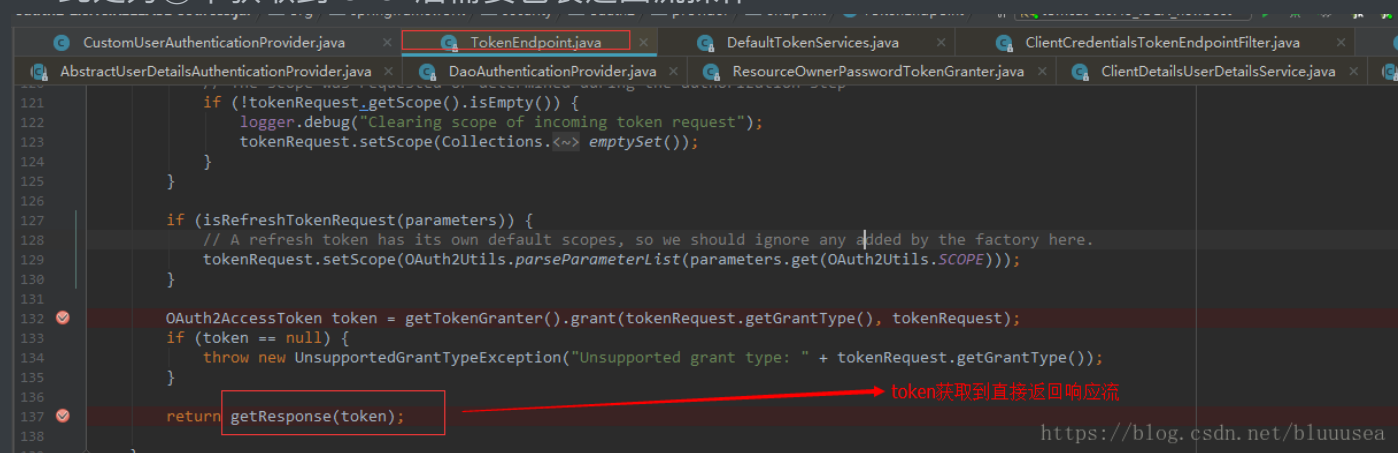

```
CustomUserAuthenticationProvider.java x TokenEndpoint.java x spring-security.xml x DefaultTokenServices.java x RedisTokenStore.java
ClientDetailsUserService.java x AbstractTokenGranter.java x ResourceOwnerPasswordTokenGranter.java
55 super(tokenServices, clientDetailsService, requestFactory, grantType);
56 this.authenticationManager = authenticationManager;
57
58
59 @Override
60 protected OAuth2Authentication getOAuth2Authentication(ClientDetails client, TokenRequest tokenRequest) {
61     Map<String, String> parameters = new LinkedHashMap<>().putAll(tokenRequest.getRequestParameters());
62     String username = parameters.get("username");
63     String password = parameters.get("password");
64     // Protect from downstream leaks of password
65     parameters.remove("password");
66
67     Authentication userAuth = new UsernamePasswordAuthenticationToken(username, password);
68     ((AbstractAuthenticationToken) userAuth).setDetails(parameters);
69     try {
70         userAuth = authenticationManager.authenticate(userAuth);
71     } catch (AccountStatusException ase) {
72         //covers expired, locked, disabled cases (mentioned in section 5.2, draft 31)
73         throw new InvalidGrantException(ase.getMessage());
74     } catch (BadCredentialsException e) {
75         // If the username/password are wrong the spec says we should send 400/invalid grant
76         throw new InvalidGrantException(e.getMessage());
77     }
78     if (userAuth == null || !userAuth.isAuthenticated()) {
79         throw new InvalidGrantException("Could not authenticate user: " + username);
80     }
81
82     OAuth2Request storedOAuth2Request = getRequestFactory().createOAuth2Request(client, tokenRequest);
83     return new OAuth2Authentication(storedOAuth2Request, userAuth);
84 }
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
26
```

13. 此处为12中调用的RedisTokenStore中的getAccessToken方法等，此处执行完，则一直向上返回到⑧中执行后续方法。



14. 此处为⑧中获取到token后需要包装返回流操作



3.2.示例中spring-security.xml的部分配置

```
<!-- 认证地址 -->
<sec:http pattern="/oauth/token" create-session="stateless"
           authentication-manager-ref="authenticationManager" >
    <sec:intercept-url pattern="/oauth/token"
access="IS_AUTHENTICATED_FULLY" />
    <sec:anonymous enabled="false" />
    <sec:http-basic entry-point-ref="clientAuthenticationEntryPoint" />
    <sec:custom-filter ref="clientCredentialsTokenEndpointFilter"
before="BASIC_AUTH_FILTER" />
    <sec:access-denied-handler ref="oauthAccessDeniedHandler" />
</sec:http>

<bean id="clientAuthenticationEntryPoint"

    class="org.springframework.security.oauth2.provider.error.OAuth2Authent
icationEntryPoint">
    <property name="realmName" value="springsec/client" />
    <property name="typeName" value="Basic" />
</bean>

<bean id="clientCredentialsTokenEndpointFilter"

    class="org.springframework.security.oauth2.provider.client.ClientCreden
tialsTokenEndpointFilter">
    <property name="authenticationManager" ref="authenticationManager"
/>
</bean>

<bean id="oauthAccessDeniedHandler"

    class="org.springframework.security.oauth2.provider.error.OAuth2AccessD
eniedHandler">
</bean>
```

```
<!-- 认证管理器-->
<sec:authentication-manager alias="authenticationManager">
    <sec:authentication-provider user-service-
ref="clientDetailsUserService" />
</sec:authentication-manager>

<!-- 注入自定义clientDetails-->
<bean id="clientDetailsUserService"

    class="org.springframework.security.oauth2.provider.client.ClientDetail
sUserDetailsService">
    <constructor-arg ref="clientDetails" />
</bean>

<!-- 自定义clientDetails-->
<bean id="clientDetails"
class="com.xxx.core.framework.oauth.CustomClientDetailsServiceImpl">
</bean>
<!-- 注入自定义provider-->
<sec:authentication-manager id="userAuthenticationManager">
    <sec:authentication-provider ref="customUserAuthenticationProvider"
/>
</sec:authentication-manager>
<!--自定义用户认证provider-->
<bean id="customUserAuthenticationProvider"

    class="com.xxx.core.framework.oauth.CustomUserAuthenticationProvider">
</bean>

<oauth:authorization-server
    client-details-service-ref="clientDetails" token-services-
ref="tokenServices" check-token-enabled="true" >
    <oauth:authorization-code />
    <oauth:implicit/>
    <oauth:refresh-token/>
```

```

        <oauth:client-credentials />
        <oauth:password authentication-manager-
ref="userAuthenticationManager"/>
    </oauth:authorization-server>

    <!-- 自定义tokenStore-->
    <bean id="tokenStore"
        class="com.xxx.core.framework.oauth.RedisTokenStore" />

    <!-- 设置access_token有效期，设置支持refresh_token，refresh_token有效期默认为
    30天-->
    <bean id="tokenServices"

        class="org.springframework.security.oauth2.provider.token.DefaultTokenS
ervices">
        <property name="tokenStore" ref="tokenStore" />
        <property name="supportRefreshToken" value="true" />
        <property name="accessTokenValiditySeconds" value="43200">
    </property>
        <property name="clientDetailsService" ref="clientDetails" />
    </bean>

```

4.总结

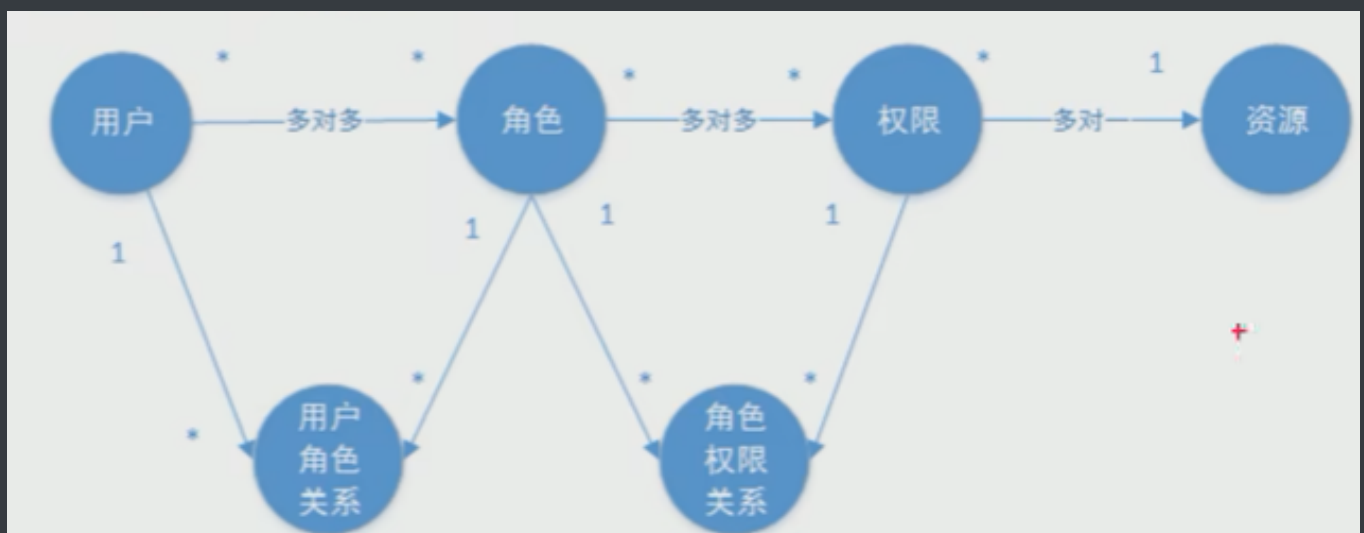
本文中的流程能够结果的问题：

- 需要自定义修改获取到的token
- token单点问题
- 使用refresh_token的情况

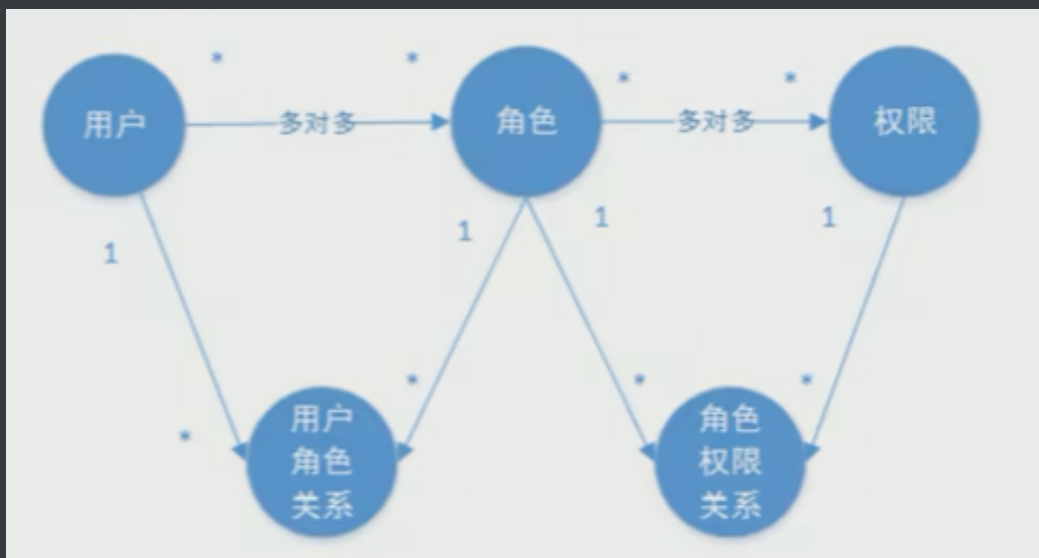
PS：梳理这个认证流程也是因为最近工作需要设置token超时机制，刷新token，检查token等，需要对认证这块了解的特别清楚才行，后面的代码截图只是详细的介绍了获取access_token的流程，其实refresh_token与access_token的流程基本是一样的，如果您在使用refresh_token过程中有什么问题，也可以详细看下上面的截图，或许会有一些收获。

1. 基本概念

- 认证: 用户认证就是判断一个用户的身份是否合法的过程，用户去访问系统资源时系统要求验证用户的身份信息，身份合法方可继续访问，不合法则拒绝访问。常见的用户身份认证方式有:用户名密码登录，二维码登录，手机短信登录，指纹认证等方式。
- 会话:用户认证通过后,为了避免用户的每次操作都进行认证可将用户的信息保证在会话中。会话就是系统为了保持当前用户的登录状态所提供的机制,常见的有基于session方式、基于token方式等。
- 授权:授权是用户认证通过后根据用户的权限来控制用户访问资源的过程,拥有资源的访问权限则正常访问，没有权限则拒绝访问。授权可简单理解为Who对What(which)进行How操作，
- Who，即主体(Subject)， 主体一般是指用户，也可以是程序，需要访问系统中的资源。
- What，即资源(Resource)， 如系统菜单、页面、按钮、代码方法、系统商品信息、系统订单信息等。系统菜单、页面、按钮、代码方法都属于 系统功能资源,对于web系统每个功能资源通常对应一个URL ;系统商品信息系统订单信息都属于 实体资源(数据资源)， 实体资源由资源类型和资源实例组成,比如商品信息为资源类型，商品编号为001的商品为资源实例。
- How ,权限/许可(Permission)， 规定了用户对资源的操作许可,权限离开资源没有意义，如用户查询权限、用户添加权限、某个代码方法的调用权限、编号为001的用户的修改权限等，通过权限可知用户对哪些资源都有哪些操作许可。
- 授权的数据模型



资源和权限可以合并一起



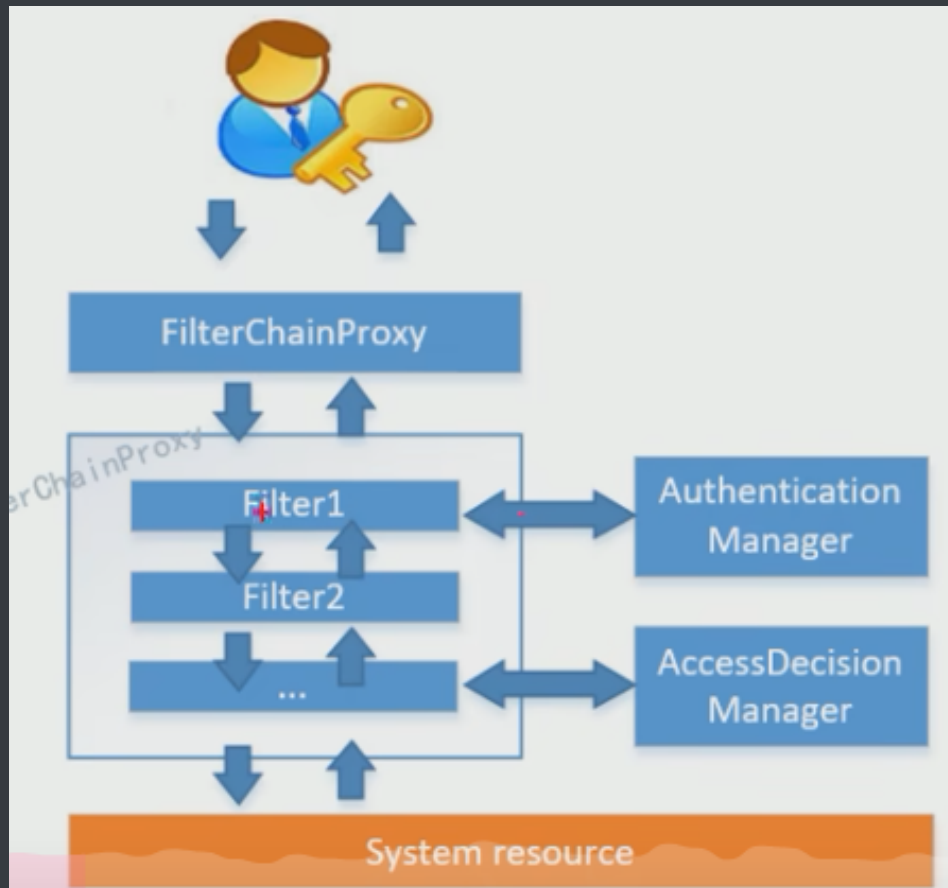
■ RBAC

基于角色的访问控制 `if (主体.hasRole()){}`

基于资源的访问控制 `if(主体.hasPermission()){}`

2. Spring Boot Security

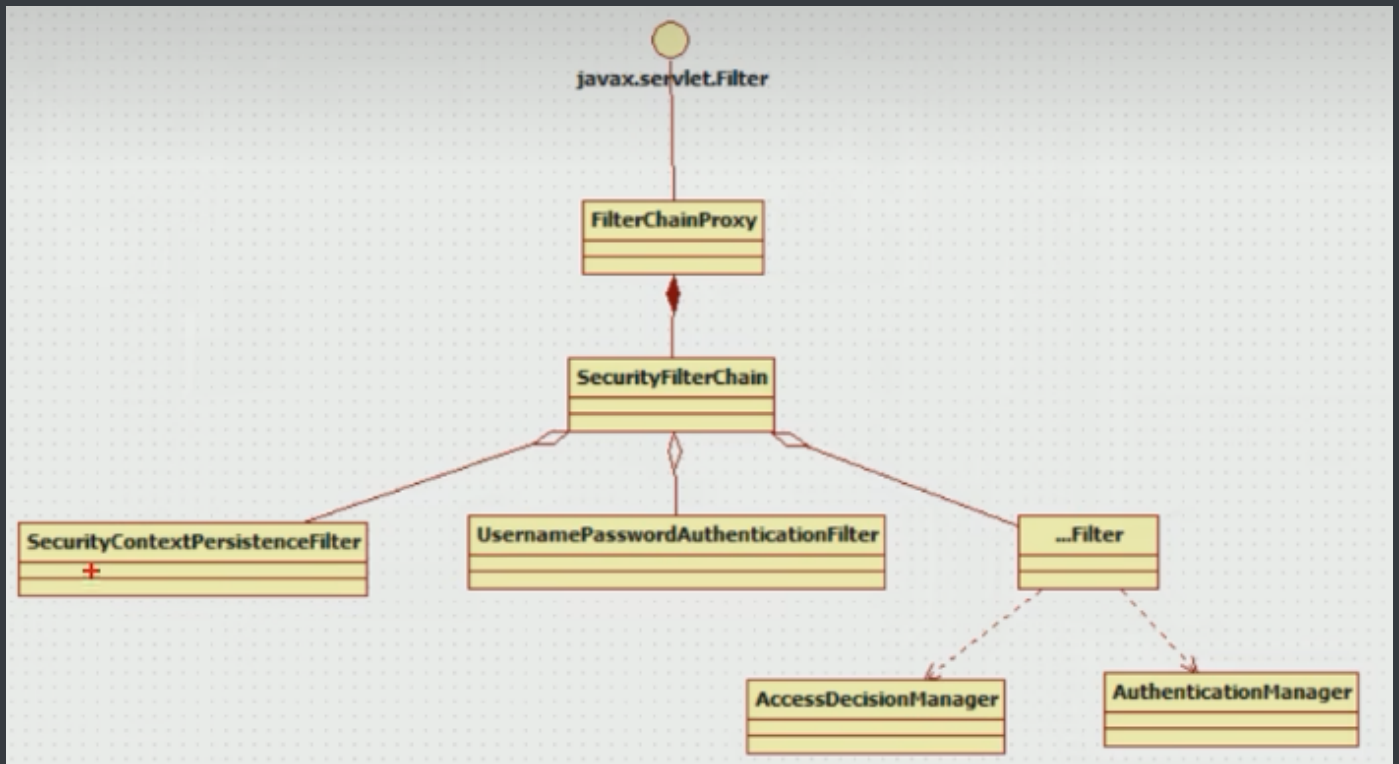
对所有进入系统的请求进行拦截，校验每个请求是否能够访问它所期望的资源，通过Filter实现



2.1 结构原理

初始化Spring Security时，初始化一个类型为
`org.springframework.security.web.FilterChainProxy`的过滤器，

外部的请求会经过此类。FilterChainProxy是一个代理，真正起作用的是FilterChainProxy中
SecurityFilterChain所包含的各个Filter，同时这些Filter作为Bean被Spring管理，它们是
Spring Security核心，并不直接处理用户的认证，也不直接处理用户的授权，而是把它们交给了
认证管理器（AuthenticationManager）和 决策管理器（AccessDecisionManager）进行处
理



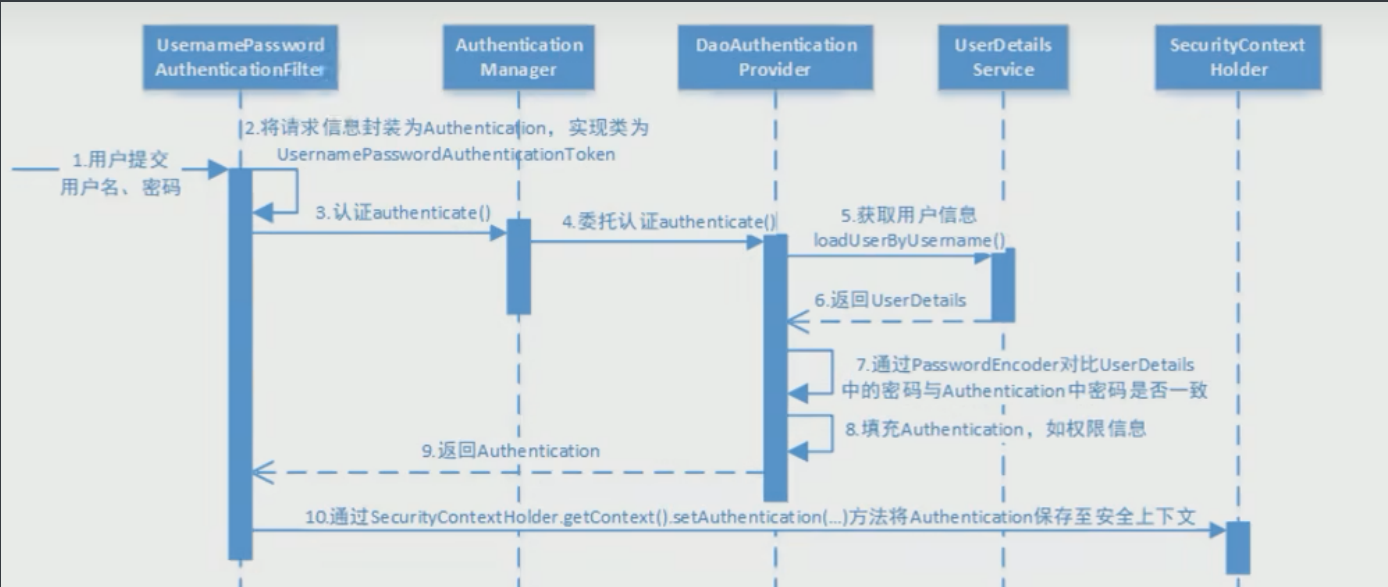
spring Security功能的实现主要是由一系列过滤器链相互配合完成。



2.2 认证过程

1. 用户提交用户名、密码被SecurityFilterChain中的 UsernamePasswordAuthenticationFilter 过滤器获取到，封装为请求Authentication，通常情况下是 UsernamePasswordAuthenticationToken这个实现类。
2. 然后过滤器将Authentication提交至认证管理器（AuthenticationManager）进行认证
3. 认证成功后， AuthenticationManager 身份管理器返回一个被填充满了信息的（包括上面提到的权限信息，身份信息，细节信息，但密码通常会被移除） Authentication 实例。
4. SecurityContextHolder 安全上下文容器将第3步填充了信息的 Authentication ，通过

SecurityContextHolder.getContext().setAuthentication(...)方法，设置到其中。



- AuthenticationManager接口（认证管理器）是认证相关的核心接口，也是发起认证的出发点，

它的实现类为ProviderManager。

- 而Spring Security支持多种认证方式，因此ProviderManager维护着一个List 列表，存放多种认证方式，最终实际的认证工作是由AuthenticationProvider完成。不同的认证方式使用不同的AuthenticationProvider。如使用用户名密码登录时，使用DaoAuthenticationProvide
- DaoAuthenticationProvider，它的内部又维护着一个UserDetailsService负责UserDetails的获取。最终AuthenticationProvider将UserDetails填充至Authentication。

2.3 密码处理

Spring Security提供了很多内置的PasswordEncoder，能够开箱即用，使用某种PasswordEncoder只需要进行如

下声明即可

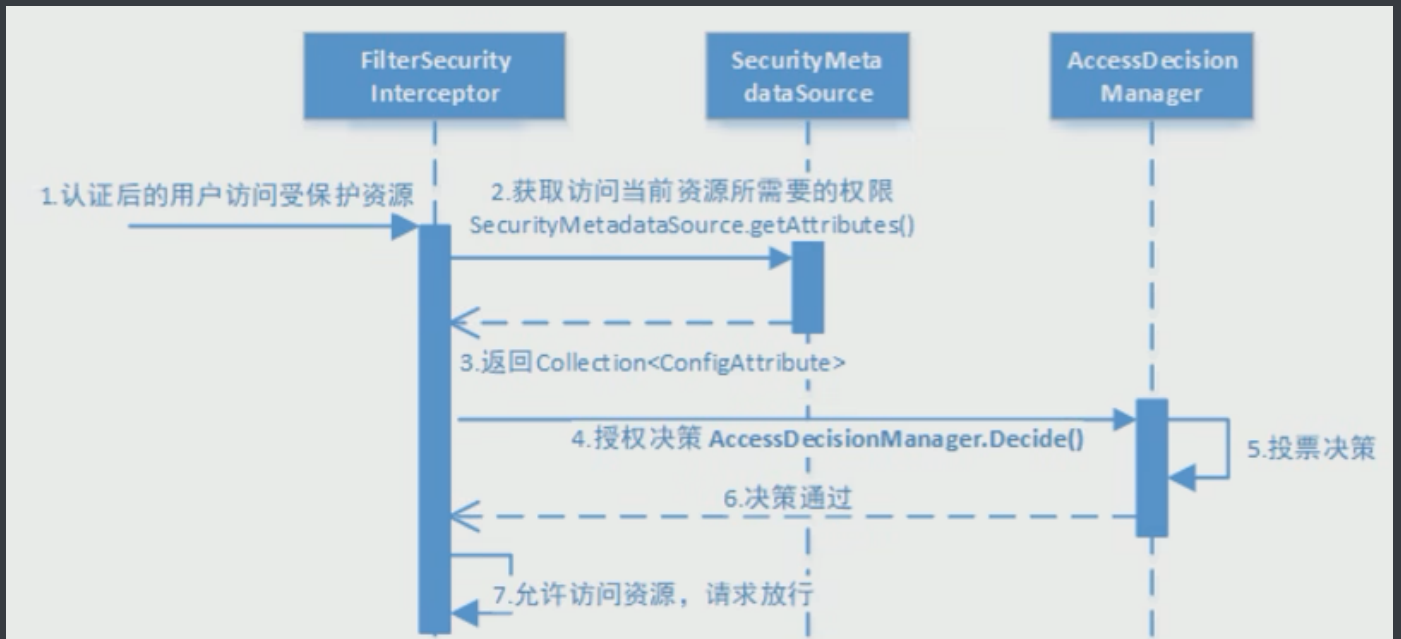
```
//密码编码器
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

常见的密码编码器有

NoOpPasswordEncoder、BCryptPasswordEncoder、Pbkdf2PasswordEncoder、SCryptPasswordEncoder

2.4 授权过程

- 已认证用户访问受保护的web资源将被SecurityFilterChain中的 FilterSecurityInterceptor 的子类拦截。
- FilterSecurityInterceptor会从 SecurityMetadataSource 的子类 DefaultFilterInvocationSecurityMetadataSource 获取要访问当前资源所需要的权限 Collection
- SecurityMetadataSource其实就是读取“安全拦截机制”
- FilterSecurityInterceptor会调用 AccessDecisionManager 进行授权决策，若决策通过，则允许访问资源，否则将禁止访问。此处会从安全上下文中获取用户：Authentication authentication = SecurityContextHolder.getContext().getAuthentication(); 再进行比对和判断。



- decide接口就是用来鉴定当前用户是否有访问对应受保护资源的权限。
- authentication：要访问资源的访问者的身份
- object：要访问的受保护资源，web请求对应FilterInvocation
- configAttributes：是受保护资源的访问策略，通过SecurityMetadataSource获取。

```

4 个实现
public interface AccessDecisionManager {
    3 个实现
    void decide(Authentication var1, Object var2, Collection<ConfigAttribute> var3) throws AccessDeniedException

    1 个实现
    boolean supports(ConfigAttribute var1);

    1 个实现
    boolean supports(Class<?> var1);
}
  
```

Spring Security内置了三个基于投票的AccessDecisionManager实现类

AffirmativeBased 只要有一个赞成票，则表示同意用户访问

ConsensusBased：赞成票多余反对票，则表示同意用户访问

UnanimousBased：只要有一个反对票，则表示反对用户访问

spring security为防止CSRF（Cross-site request forgery跨站请求伪造）的发生，限制了除了get以外的大多数方法，

解决方法1:

屏蔽CSRF控制，即spring security不再限制CSRF。httpSecurity.csrf().disable()

解决方法2:

表单添加隐藏域: <input type="hidden" name="csrf.parameterName" value="{_csrf.token}"/>

2.5 实际应用

用户认证通过后，为了避免用户的每次操作都进行认证可将用户的信息保存在会话中。spring security提供会话管理，认证通过后将身份信息放入SecurityContextHolder上下文，SecurityContextHolder与当前线程进行绑定，方便获取用户身份。

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
```

2.5.1 会话控制

- Spring Security会为每个登录成功的用户会新建一个Session

```
httpSecurity.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
```

- session超时、session失效后，通过Spring Security 设置跳转的路径。

```
httpSecurity.sessionManagement()
```

```
.expiredUrl("/login-view?error=EXPIRED_SESSION")
```

```
.invalidSessionUrl("/login-view?error=INVALID_SESSION");
```

- session退出

```
httpSecurity.logout()
```

```
.logoutUrl("/logout")
```

```
.logoutSuccessUrl("/login-view?logout")
```

```
.logoutSuccessHandler(logoutSuccessHandler)
```

```
.addLogoutHandler(logoutHandler)
```

```
.invalidateHttpSession(true)
```

定制的 LogoutSuccessHandler，实现用户退出成功时的处理。如果指定了这个选项那么 logoutSuccessUrl() 的设置被忽略；

添加一个 LogoutHandler，用于实现用户退出时的清理工作。默认 SecurityContextLogoutHandler 会被添加为最后一个 LogoutHandler；

指定是否在退出时让 HttpSession 无效。默认设置为 true。

2.5.2 web授权

使用 http.authorizeRequests() 对web资源进行授权保护

```
httpSecurity
```

```
.authorizeRequests()
```

```
.antMatchers("/r/r1").hasAuthority("p1") (
```

```
.antMatchers("/r/r2").hasAuthority("p2")
```

```
.antMatchers("/r/r3").access("hasAuthority('p1') and hasAuthority('p2')")
```

```
.antMatchers("/r/**").authenticated()
```

```
.anyRequest().permitAll()
```

```
.and()
```

```
.formLogin()
```

- 保护URL常用的方法有：

`authenticated()` 保护URL，需要用户登录

`permitAll()` 指定URL无需保护，一般应用与静态资源文件

`hasRole(String role)` 限制单个角色访问，角色将被增加 “ROLE_” .所以”ADMIN” 将和 “ROLE_ADMIN”进行比较.

`hasAuthority(String authority)` 限制单个权限访问

`hasAnyRole(String... roles)`允许多个角色访问.

`hasAnyAuthority(String... authorities)` 允许多个权限访问.

`access(String attribute)` 该方法使用 SpEL表达式, 所以可以创建复杂的限制.

`hasIpAddress(String ipAddressExpression)` 限制IP地址或子网

2.5.3.方法授权

可以在任何 `@Configuration` 实例上使用 `@EnableGlobalMethodSecurity` 注释来启用基于注解的安全性。

`@PreAuthorize`,`@PostAuthorize`, `@Secured`三类注解作用域服务层方法上，限制访问的访问

例如：

- 匿名访问

`@Secured("IS_AUTHENTICATED_ANONYMOUSLY")`

`@PreAuthorize("isAnonymous()")`

方法可匿名访问，底层使用`WebExpressionVoter`投票器

- `@PreAuthorize`

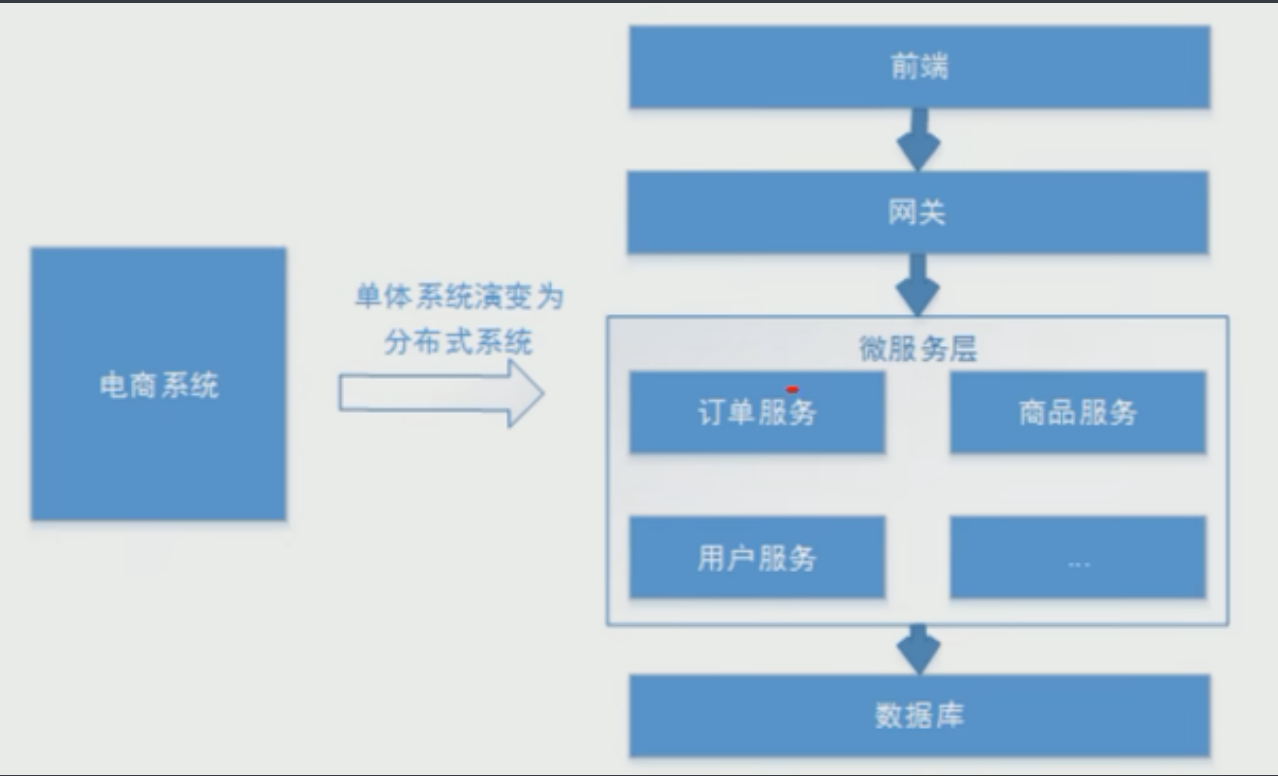

```
@PreAuthorize("hasAuthority('p_transfer') and hasAuthority('p_read_account')")
```

同时拥有p_transfer和p_read_account权限才能访问，底层使用WebExpressionVoter投票器

另外，还有注解@PostAuthorize

3. 分布式系统认证方案

软件的架构由单体结构演变为分布式架构，具有分布式架构的系统叫分布式系统，分布式系统的运行通常依赖网络，它将单体结构的系统分为若干服务，服务之间通过网络交互来完成用户的业务处理，当前流行的微服务架构就是分布式系统架构。



3.1 分布式认证需求

分布式系统的每个服务都会有认证、授权的需求，考虑分布式系统共享性的特点，需要由独立的认证服务处理系统认证授权的请求；考虑分布式系统开放性的特点，不仅对系统内部服务提供认证，对第三方系统也要提供认证。分布式认证的需求总结如下：

统一认证授权

提供独立的认证服务，统一处理认证授权。无论是不同类型的用户，还是不同种类的客户端 (web端，H5、APP)，均采用一致的认证、权限、会话机制，实现统一认证授权。要实现统一认证方式必须可扩展，支持各种认证需求，比如：用户名密码认证、短信验证码、二维码、人脸识别等认证方式，并可以非常灵活的切换。

应用接入认证

应提供扩展和开放能力，提供安全的系统对接机制，并可开放部分API给接入第三方使用，一方应用（内部系统服务）和三方应用（第三方应用）均采用统一机制接入。

3.2 选型分析

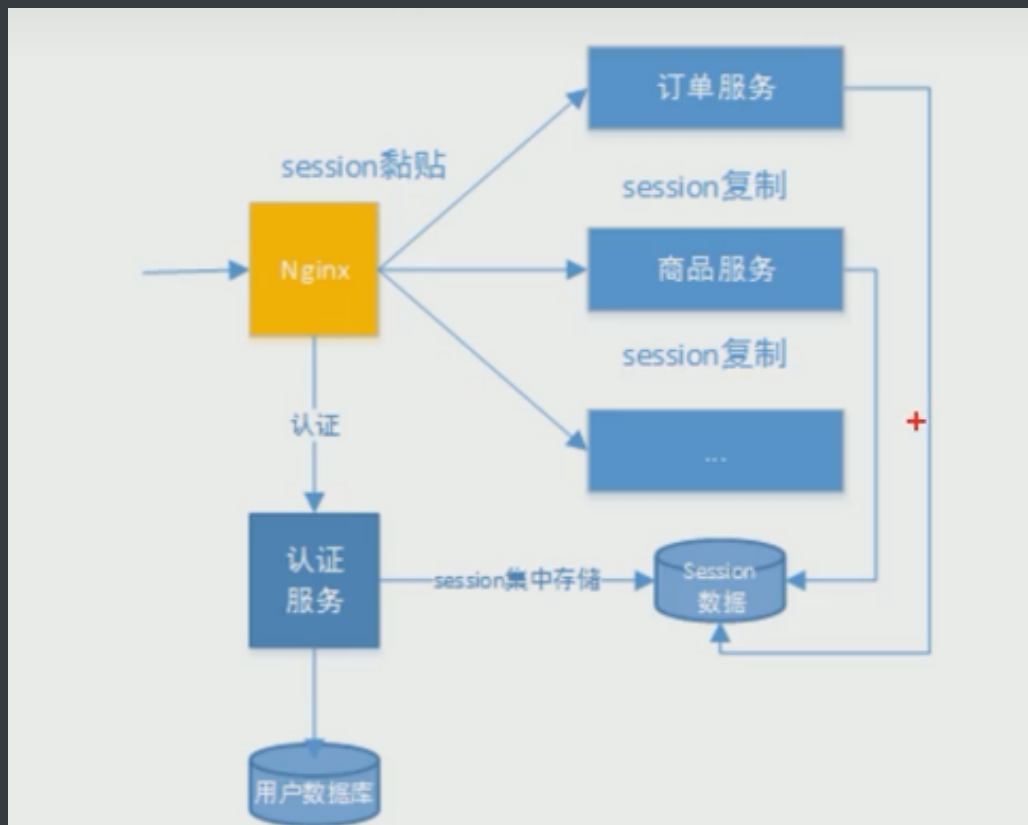
3.2.1 基于Session认证

每个应用服务都需要在session中存储用户身份信息，通常的做法有下面几种：

Session复制：多台应用服务器之间同步session，使session保持一致，对外透明。

Session黏贴：当用户访问集群中某台服务器后，强制指定后续所有请求均落到此机器上。

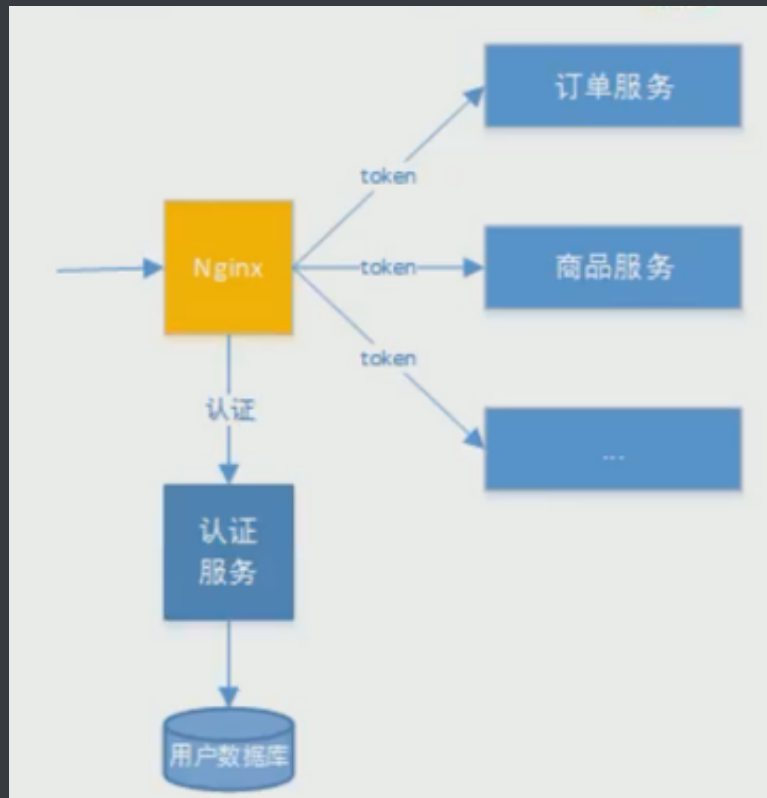
Session集中存储：将Session存入分布式缓存中，所有服务器应用实例统一从分布式缓存中存取Session。



基于session认证的认证方式，可以更好的在服务端对会话进行控制，且安全性较高。但是，session机制方式基于cookie，在复杂多样的移动客户端上不能有效的使用，并且无法跨域，另外随着系统的扩展需提高session的复制、黏贴及存储的容错性。

3.2.2 基于Token认证

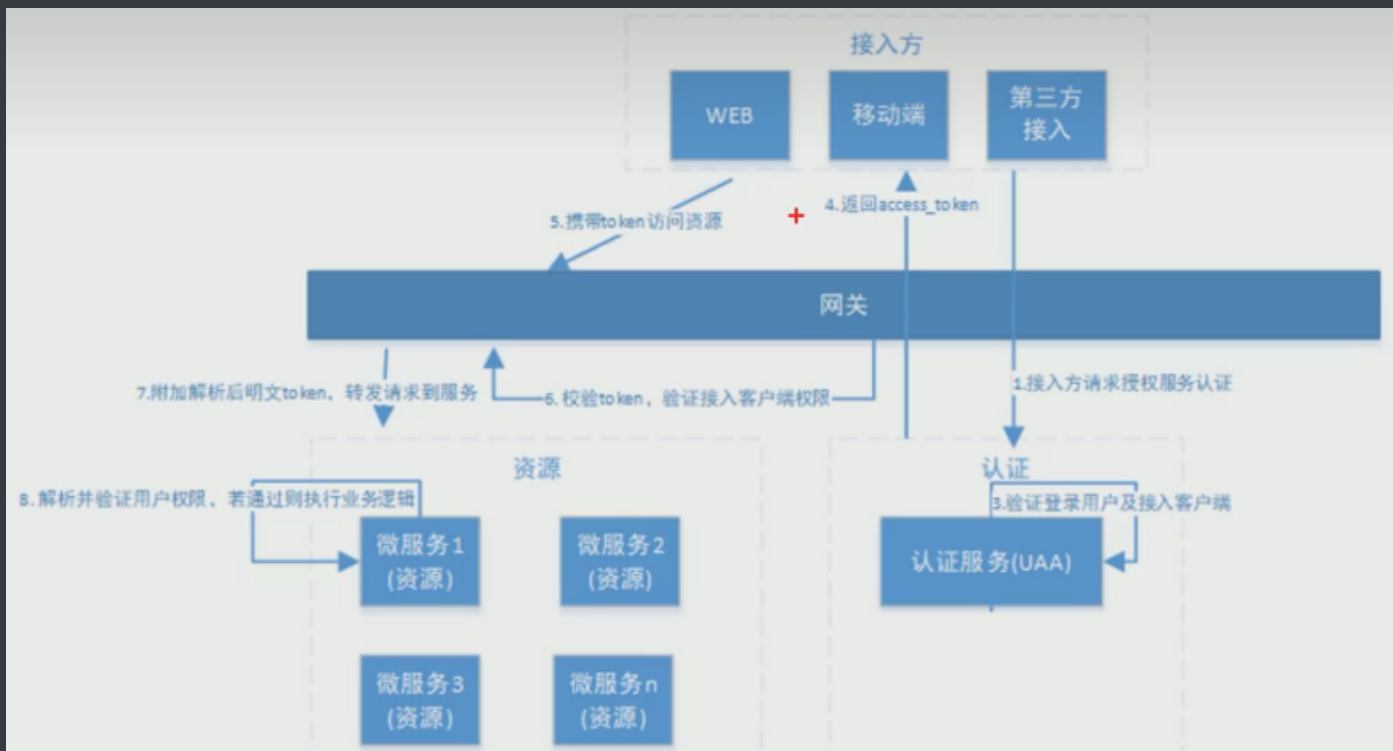
基于token的认证方式，服务端不用存储认证数据，易维护扩展性强，客户端可以把token 存在任意地方，并且可以实现web和app统一认证机制。其缺点也很明显，token由于自包含信息，因此一般数据量较大，而且每次请求都需要传递，因此比较占带宽。另外，token的签名验签操作也会给cpu带来额外的处理负担。



3.2.3 技术方案

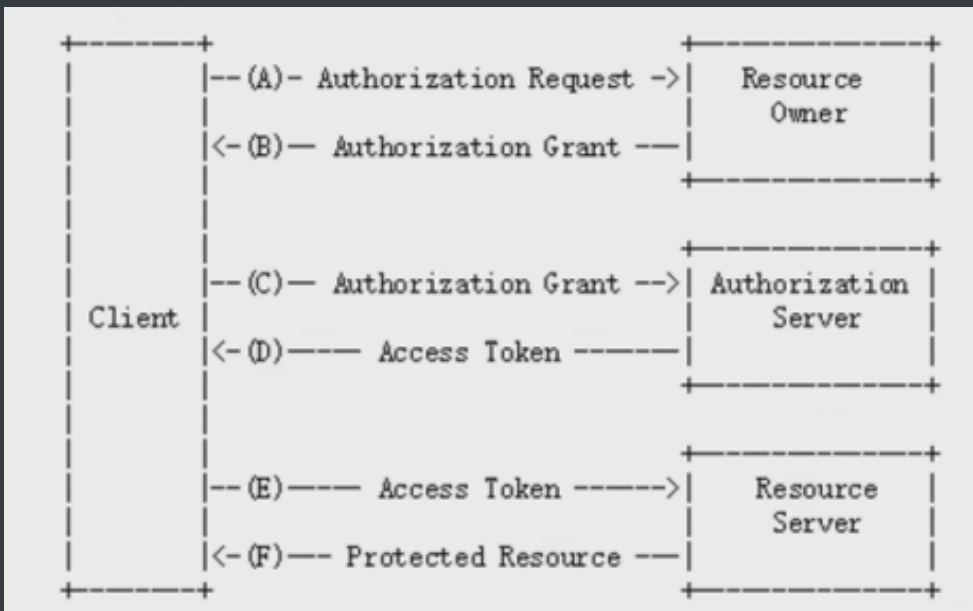
根据 选型分析，采用token认证，它的优点是：

1. 适合统一认证的机制，客户端、一方应用、三方应用都遵循一致的认证机制。
2. token认证方式对第三方应用接入更适合，因为它更开放，可使用当前有流行的开放协议 OAuth2.0、JWT等。
3. 一般情况服务端无需存储会话信息，减轻了服务端的压力。



3.3 Oauth2.0

OAuth（开放授权）是一个开放标准，允许用户授权第三方应用访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方应用或分享他们数据的所有内容。例如，第三方登录，用户 授权 电商网站 访问 用户存储在微信平台的用户信息



OAuth2.0包括以下角色：

1. 客户端

本身不存储资源，需要通过资源拥有者的授权去请求资源服务器的资源，比如：Android客户端、Web客户端（浏览器端）、微信客户端等。

2. 资源拥有者

通常为用户，也可以是应用程序，即该资源的拥有者。

3. 资源服务器

存储资源的服务器，例如微信平台用户信息资源。

服务提供商会给准入的接入方一个身份，用于接入时的凭据：

client_id：客户端标识

client_secret：客户端密钥

4. 授权服务器（也称认证服务器）

用于服务提供商对资源拥有的身份进行认证、对访问资源进行授权，认证成功后会给客户端发放令牌，作为客户端访问资源服务器的凭据。例如：微信认证服务器。

授权服务器对OAuth2.0中的两个角色进行认证授权，分别是资源拥有者、客户端

4. Spring Cloud Security OAuth2

Spring-Security-OAuth2是对OAuth2的一种实现，并且跟Spring Security相辅相成，与SpringCloud体系的集成也非常便利。其服务实现包括两个服务：

- 认证授权服务 (Authorization Server)

应包含对接入端以及登入用户的合法性进行验证并颁发token等功能，对令牌的请求端点由Spring MVC 控制器进行实现，下面是配置一个认证服务必须要实现的endpoints：

AuthorizationEndpoint 服务于认证请求。默认 URL：

/oauth/authorize 。

TokenEndpoint 服务于访问令牌请求。默认 URL:

/oauth/token。

- 资源服务 (Resource Server),

应包含对资源的保护功能, 对非法请求进行拦截, 对请求中token进行解析鉴权等, 下面的过滤器用于实现 OAuth 2.0 资源服务:

OAuth2AuthenticationProcessingFilter用来对请求给出的身份令牌解析鉴权。

4.1 授权服务器配置

可以使用

@Configuration

@EnableAuthorizationServer

注解并继承AuthorizationServerConfigurerAdapter来配置OAuth2.0 授权服务器。

AuthorizationServerConfigurerAdapter要求配置

ClientDetailsServiceConfigurer: 用来配置客户端详情服务 (ClientDetailsService), 客户端详情信息在这里进行初始化

AuthorizationServerEndpointsConfigurer: 用来配置令牌 (token) 的访问端点和令牌服务 (tokenservices)。

AuthorizationServerSecurityConfigurer: 用来配置令牌访问端点的安全约束

4.1.1 客户端详情

ClientDetailsServiceConfigurer能够使用内存或者JDBC来实现客户端详情, ClientDetailsService负责查找ClientDetails, 而ClientDetails有几个重要的属性


```

//客户端详情服务
@Override
public void configure(ClientDetailsServiceConfigurer clients)
    throws Exception {
    clients.withClientDetails(clientDetailsService);
    clients.inMemory()// 使用in-memory存储，也可以使用数据库存储
        .withClient( clientId: "c1")// client_id (必须的) 用来标识客户端的Id
        .secret(new BCryptPasswordEncoder().encode( rawPassword: "secret"))//客户端密钥
        .resourceIds("res1")//资源列表
        .authorizedGrantTypes("authorization_code", "password", "client_credentials", "implicit", "refresh_token")// 该client允许的授权类型
        .scopes("all")// 用来限制客户端的访问范围，如果为空（默认）的话，那么客户端拥有全部的访问范围，一般比resourceIds的属性值范围小
        .autoApprove(false)//false跳转到授权页面
        .redirectUri( ...registeredRedirectUri: "http://www.baidu.com"); //验证回调地址
}

```

客户端详情（Client Details）能够在应用程序运行的时候进行更新，可以通过访问底层的存储服务（例如将客户端详情存储在一个关系数据库的表中，就可以使用 JdbcClientDetailsService）或者通过实现

ClientRegistrationService接口（同时你也可以实现 ClientDetailsService 接口）来管理。

此次客户端详情读取数据库的配置：

```

//密码编码器
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

//将客户端信息存储到数据库
@Bean
public ClientDetailsService clientDetailsService(DataSource dataSource) {
    ClientDetailsService clientDetailsService = new JdbcClientDetailsService(dataSource);
    ((JdbcClientDetailsService) clientDetailsService).setPasswordEncoder(passwordEncoder);
    return clientDetailsService;
}

//客户端详情服务
@Override
public void configure(ClientDetailsServiceConfigurer clients)
    throws Exception {
    clients.withClientDetails(clientDetailsService);
}

```

4.1.2 令牌管理服务

AuthorizationServerTokenServices 接口定义了一些操作可以对令牌进行一些必要的管理,这个接口的实现,则需要继承 DefaultTokenServices,里面包含了一些有用实现,你可以使用它来修改令牌的格式和令牌的存储,其持久化令牌委托一个 TokenStore 接口来实现,TokenStore 这个接口有一个默认的实现,它就是 InMemoryTokenStore。

- InMemoryTokenStore:

这个版本的实现是被默认采用的,它可以完美的工作在单服务器上(即访问并发量压力不大的情况下,并且它在失败的时候不会进行备份),大多数的项目都可以使用这个版本的实现来进行尝试,可以在开发的时候使用它来进行管理,因为不会被保存到磁盘中,更易于调试。

- JdbcTokenStore:

这是一个基于JDBC的实现版本,令牌会被保存进关系型数据库。使用这个版本的实现时,你可以在不同的服务器之间共享令牌信息,使用这个版本的时候需要把"spring-jdbc"这个依赖加入到classpath。

- JwtTokenStore

这个版本的全称是 JSON Web Token (JWT),它可以把令牌相关的数据进行编码(因此对于后端服务来说,它不需要进行存储,这将是一个重大优势),但是它有一个缺点,那就是撤销一个已经授权令牌将会非常困难,所以它通常用来处理一个生命周期较短的令牌以及撤销刷新令牌(refresh_token)。另外一个缺点就是这个令牌占用的空间会比较大,如果你加入了比较多用户凭证信息。JwtTokenStore 不会保存任何数据,但是它在转换令牌值以及授权信息方面与 DefaultTokenServices 所扮演的角色是一样的。

```
@Bean
public TokenStore tokenStore() {
    //JWT令牌存储方案
    return new JwtTokenStore(accessTokenConverter());
}
```

1 个用法

```
@Bean
public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey(SIGNING_KEY); //对称密钥，资源服务器使用该密钥来验证
    return converter;
}
```

```
@Bean
public AuthorizationServerTokenServices tokenService() {
    DefaultTokenServices service = new DefaultTokenServices();
    service.setClientDetailsService(clientDetailsService); //客户端详情服务
    service.setSupportRefreshToken(true); //支持刷新令牌
    service.setTokenStore(tokenStore); //令牌存储策略
    //令牌增强
    TokenEnhancerChain tokenEnhancerChain = new TokenEnhancerChain();
    tokenEnhancerChain.setTokenEnhancers(Arrays.asList(accessTokenConverter));
    service.setTokenEnhancer(tokenEnhancerChain);

    service.setAccessTokenValiditySeconds(7200); // 令牌默认有效期2小时
    service.setRefreshTokenValiditySeconds(259200); // 刷新令牌默认有效期3天
    return service;
}
```

4.1.3 令牌访问端点

4.1.3.1 认证管理器

AuthorizationServerEndpointsConfigurer通过设定以下属性决定支持的授权类型（Grant Types）：

- authenticationManager

认证管理器，选择了资源所有者密码（password）授权类型的时候，请设置这个属性注入一个 AuthenticationManager 对象。

- userDetailsService

设置了这个属性，需要有一个 UserDetailsService 接口的实现

- authorizationCodeServices:

用来设置授权码服务（即 AuthorizationCodeServices 的实例对象），主要用于 "authorization_code" 授权码类型模式。

- implicitGrantService

这个属性用于设置隐式授权模式，用来管理隐式授权模式的状态。

- tokenGranter:

授权将会交由自己来完全掌控，并且会忽略掉上面的这几个属性，这个属性一般是用作拓展用途

```
@Bean
public AuthorizationCodeServices authorizationCodeServices(DataSource dataSource) {
    return new JdbcAuthorizationCodeServices(dataSource); //设置授权码模式的授权码如何存取
}
```

4.1.3.2 令牌访问点

框架的默认URL访问链接如下列表

/oauth/authorize: 授权端点。

/oauth/token: 令牌端点。

/oauth/confirm_access: 用户确认授权提交端点。

/oauth/error: 授权服务错误信息端点。

/oauth/check_token: 用于资源服务访问的令牌解析端点。

/oauth/token_key: 提供公有密钥的端点，如果你使用JWT令牌的话。

AuthorizationServerEndpointsConfigururer 这个配置对象有一个叫做 pathMapping() 的方法用来配置端点URL链接，它有两个参数：

第一个参数：String 类型的，这个端点URL的默认链接。

第二个参数：String 类型的，你要进行替代的URL链接。

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints
        .authenticationManager(authenticationManager)//认证管理器
        .authorizationCodeServices(authorizationCodeServices)//授权码服务
        .tokenServices(tokenService())//令牌管理服务
        .allowedTokenEndpointRequestMethods(HttpMethod.POST);
}
```

4.1.4 令牌访问端点安全约束

AuthorizationServerSecurityConfigurer: 用来配置令牌端点(Token Endpoint)的安全约束

```
@Override
public void configure(AuthorizationServerSecurityConfigurer security) {
    security
        .tokenKeyAccess("permitAll()") //oauth/token_key是公开
        .checkTokenAccess("permitAll()") //oauth/check_token公开
        .allowFormAuthenticationForClients() //允许表单认证（申请令牌）
    ;
}
```

4.1.5 WEB安全配置

```
//安全拦截机制（最重要）
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable() HttpSecurity
        .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
        .antMatchers( ...antPatterns: "/r/r1").hasAnyAuthority( ...authorities: "p1")
        .antMatchers( ...antPatterns: "/login*").permitAll()
        .anyRequest().authenticated()
        .and() HttpSecurity
        .formLogin() ;
}
```

4.1.6 授权类型

4.1.6.1 授权码模式

- 资源拥有者打开客户端，客户端要求资源拥有者给予授权，它将浏览器被重定向到授权服务器，重定向时会附加客户端的身份信息

/uaa/oauth/authorize?

client_id=c1&response_type=code&scope=all&redirect_uri=<http://www.baidu.com>

- 浏览器出现向授权服务器授权页面，之后用户同意授权
- 授权服务器将授权码（AuthorizationCode）经浏览器发送给client
- 客户端拿着授权码向授权服务器索要访问access_token

/uaa/oauth/token?

client_id=c1&client_secret=secret&grant_type=authorization_code&code=5PgfcD&redirect_uri=<http://www.baidu.com>

- 授权服务器返回令牌(access_token)

参数列表如下

client_id：客户端准入标识。

response_type：授权码模式固定为code。

scope：客户端权限。

client_secret：客户端密钥。

grant_type：授权类型，填写authorization_code，表示授权码模式

code：授权码，就是刚刚获取的授权码，注意：授权码只使用一次就无效了，需要重新申请。

redirect_uri：申请授权码时的跳转url，一定和申请授权码时用的redirect_uri一致。

4.1.6.2 简化模式

- 资源拥有者打开客户端，客户端要求资源拥有者给予授权，它将浏览器被重定向到授权服务器，重定向时会附加客户端的身份信息

/uaa/oauth/authorize?

client_id=c1&response_type=token&scope=all&redirect_uri=<http://www.baidu.com>

- 浏览器出现向授权服务器授权页面，之后用户同意授权
- 授权服务器将授权码将令牌（access_token）以Hash的形式存放在重定向uri的fargment中发送给浏览器。fragment 主要是用来标识 URI 所标识资源里的某个资源，在 URI 的末尾通过（#）作为 fragment 的开头，其中 # 不属于 fragment 的值。

4.1.6.3 密码模式

- 资源拥有者将用户名、密码发送给客户端
- 客户端拿着资源拥有者的用户名、密码向授权服务器请求令牌（access_token）

/uaa/oauth/token?

client_id=c1&client_secret=secret&grant_type=password&username=shangsan&password=123

- 授权服务器将令牌（access_token）发送给client

4.1.6.4 客户端模式

- 客户端向授权服务器发送自己的身份信息，并请求令牌（access_token）
- 确认客户端身份无误后，将令牌（access_token）发送给client

/uaa/oauth/token?client_id=c1&client_secret=secret&grant_type=client_credentials

4.2资源服务器配置

@Configuration

@EnableResourceServer


```
public class ResourceServerConfig extends ResourceServerConfigurerAdapter
```

@EnableResourceServer 注解自动增加了一个类型为 OAuth2AuthenticationProcessingFilter 的过滤器链

- ResourceServerSecurityConfigurer中主要包括：

tokenServices: ResourceServerTokenServices 类的实例，用来实现令牌服务。

tokenStore: TokenStore类的实例，指定令牌如何访问，与tokenServices配置可选

resourceId: 这个资源服务的ID，这个属性是可选的，但是推荐设置并在授权服务中进行验证。

其他的拓展属性例如 tokenExtractor 令牌提取器用来提取请求中的令牌。

- HttpSecurity配置这个与Spring Security类似：

请求匹配器，用来设置需要进行保护的资源路径，默认的情况下是保护资源服务的全部路径。

通过http.authorizeRequests()来设置受保护资源的访问规则

其他的自定义权限保护规则通过 HttpSecurity 来进行配置。

```
//资源服务令牌解析服务
1 个用法
@Bean
public ResourceServerTokenServices tokenService() {
    //使用远程服务请求授权服务器校验token,必须指定校验token 的url、client_id, client_secret
    RemoteTokenServices service = new RemoteTokenServices();
    service.setCheckTokenEndpointUrl("http://localhost:53020/uaa/oauth/check_token");
    service.setClientId("c1");
    service.setClientSecret("secret");
    return service;
}
```

RemoteTokenServices 资源服务器通过 HTTP 请求来解码令牌，每次都请求授权服务器端点 /oauth/check_token

```

@Override
public void configure(ResourceServerSecurityConfigurer resources) {
    resources.resourceId(RESOURCE_ID)//资源 id
//    .tokenStore(tokenStore)
    .tokenServices(tokenService())//验证令牌的服务
    .stateless(true);
}

@Override
public void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
        .antMatchers( ...antPatterns: "/*").access( attribute: "#oauth2.hasScope('ROLE_ADMIN')")
        .and().csrf().disable() HttpSecurity
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}

```

此处使用JWT校验令牌

```

@Autowired
TokenStore tokenStore;

@Override
public void configure(ResourceServerSecurityConfigurer resources) {
    resources.resourceId(RESOURCE_ID)//资源 id
    .tokenStore(tokenStore)
//    .tokenServices(tokenService())//验证令牌的服务
    .stateless(true);
}

```

配置安全拦截机制

```

@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    //安全拦截机制（最重要）
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
                .antMatchers( ...antPatterns: "/r/**").authenticated()//所有/r/**的请求必须认证通过
                .anyRequest().permitAll()//除了/r/**，其它的请求可以访问
            ;
    }
}

```

@EnableResourceServer 注解自动增加了一个类型为 OAuth2AuthenticationProcessingFilter 的过滤器链,此过滤器作用，主要是用来解析网关传过来的用户信息字符串，并存入安全上下文中

```

@Component
public class TokenAuthenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse httpServletResponse, FilterChain filterChain) throws ServletException {
        //解析出头中的token
        String token = request.getHeader("json-token");
        if(token!=null){
            String json = EncryptUtil.decodeUTF8StringBase64(token);
            //将token转成json对象
            JSONObject jsonObject = JSON.parseObject(json);
            //用户身份信息
            UserDTO userDTO = JSON.parseObject(jsonObject.getString("principal"), UserDTO.class);
            //用户权限
            JSONArray authoritiesArray = jsonObject.getJSONArray("authorities");
            String[] authorities = authoritiesArray.toArray(new String[authoritiesArray.size()]);
            //将用户信息和权限填充到用户身份token对象中
            UsernamePasswordAuthenticationToken authenticationToken
                = new UsernamePasswordAuthenticationToken(userDTO, null, AuthorityUtils.createAuthorityList(authorities));
            authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            //将authenticationToken填充到安全上下文
            SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        }
        filterChain.doFilter(request, httpServletResponse);
    }
}

```

4.3 网关

网关整合 OAuth2.0作为OAuth2.0的资源服务器角色，实现接入客户端权限拦截、令牌解析并转发当前登录用户信息(jsonToken)给微服务

2 个用法

```
public static final String RESOURCE_ID = "res1";
```

//uaa资源服务配置

@Configuration

@EnableResourceServer

```
public class UAAServerConfig extends ResourceServerConfigurerAdapter {
```

1 个用法

@Autowired

```
private TokenStore tokenStore;
```

@Override

```
public void configure(ResourceServerSecurityConfigurer resources){  
    resources.tokenStore(tokenStore).resourceId(RESOURCE_ID)  
        .stateless(true);  
}
```

@Override

```
public void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers( ...antPatterns: "/uaa/**").permitAll();  
}
```

```
}
```

//order资源

//uaa资源服务配置

@Configuration

@EnableResourceServer

```
public class OrderServerConfig extends ResourceServerConfigurerAdapter {
```

1 个用法

@Autowired

```
private TokenStore tokenStore;
```

@Override

```
public void configure(ResourceServerSecurityConfigurer resources){  
    resources.tokenStore(tokenStore).resourceId(RESOURCE_ID)  
        .stateless(true);  
}
```

@Override

```
public void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
        .antMatchers( ...antPatterns: "/order/**").access( attribute: "#oauth2.hasScope('ROLE_API')");  
}
```

```
}
```

```
public class AuthFilter extends ZuulFilter
```

校验客户端token，验证接入客户端权限，并解析后明文token，转发请求到微服务

一、OAuth 2.0协议 关于Scope的说明

1.概念

Scope是 **OAuth 2.0** 中的一种机制，用于限制应用程序对用户帐户的访问。应用程序可以请求一个或多个范围，然后该信息会在同意屏幕中呈现给用户，并且颁发给应用程序的访问令牌将仅限于授予的范围。

OAuth 规范允许授权服务器或用户根据请求修改授予应用程序的范围，尽管在实践中这样做的服务示例并不多。

OAuth 没有为范围定义任何特定值，因为它高度依赖于服务的内部架构和需求。

GitHub 文档描述

通过作用域，您可以准确指定所需的访问权限类型。作用域限制 **OAuth** 令牌的访问权限。它们不会授予超出用户权限范围的任何额外权限。

在 **GitHub** 上设置 **OAuth** 应用程序时，请求的作用域会在授权表单上显示给用户。

例如通过授权码获取访问令牌后，可以通过以下方式查询当前作用域：

```
$ curl -H "Authorization: token OAUTH-TOKEN"
https://api.github.com/users/codertocat -I
HTTP/2 200
X-OAuth-Scopes: repo, user \ \ 列出令牌已授权的作用域。
X-Accepted-OAuth-Scopes: user \ \ 列出操作检查的作用域。
```

GitHub定义了一些可用作用域

名称	描述
(无作用域)	授予对公共信息的只读访问权限（包括用户个人资料信息、公共仓库信息和 gist）
repo	授予对仓库（包括私有仓库）的完全访问权限。这包括对仓库和组织的代码、提交状态、仓库和组织项目、邀请、协作者、添加团队成员身份、部署状态以及仓库 web 挂钩的读取/写入权限。还授予管理用户项目的权限。
repo:status	授予对公共和私有仓库提交状态的读/写权限。仅在授予其他用户或服务对私有仓库提交状态的访问权限而不授予对代码的访问权限时，才需要此作用域。
repo_deployment	授予对公共和私有仓库的部署状态的访问权限。仅在授予其他用户或服务对部署状态的访问权限而不授予对代码的访问权限时，才需要此作用域。
public_repo	将访问权限限制为公共仓库。这包括对公共仓库和组织的代码、提交状态、仓库项目、协作者以及部署状态的读取/写入权限。标星公共仓库也需要此权限。

CSDN @云烟疏雨

1. 添加作用域

在数据库或者内存中，给当前客户端添加了三个作用域。

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
    // 配置客户端
    clients
        // 使用内存设置
        .inMemory()
        // client_id
        .withClient("client")
        // client_secret
        .secret(passwordEncoder.encode("secret"))
        // 授权类型：授权码、刷新令牌、密码、客户端、简化模式、短信验证码
        "refresh_token"
        .authorizedGrantTypes("authorization_code", "password",
"client_credentials", "implicit", "sms_code")
        // 授权范围，也可根据这个范围标识，进行鉴权
        .scopes("admin:org","write:org","read:org")
    }
```

```
.accessTokenValiditySeconds(300)
.refreshTokenValiditySeconds(3000)
// 授权码模式 授权页面是否自动授权
//.autoApprove(false)
// 拥有的权限
.authorities("add:user")
// 允许访问的资源服务 ID
//.resourceIds("oauth2-resource-server001-demo")
// 注册回调地址
.redirectUri("http://localhost:20000/code");
}
```

2. 授权页面选择scope

授权使用不带scope参数访问授权码端点：

```
http://localhost:20000/oauth/authorize?
client_id=client&client_secret=secret&response_type=code
```

可以看到授权页面会弹出所有的scope范围

OAuth Approval

Do you authorize "client" to access your protected resources?

- scope.admin:org: ☐ Approve ☒ Deny
- scope.write:org: ☐ Approve ☒ Deny
- scope.read:org: ☐ Approve ☒ Deny

Authorize

CSDN @云烟成雨

如果携带了scope参数，则表明只需要对当前作用域进行授权：

```
http://localhost:20000/oauth/authorize?
client_id=client&client_secret=secret&response_type=code&scope=admin:org
```

OAuth Approval

Do you authorize "client" to access your protected resources?

- scope.admin:org: ☒ Approve ☐ Deny

Authorize

CSDN @云烟成雨

如果当前客户端没有配置scope作用域，申请的时候也没有传递scope参数，则会报错：

```
Handling OAuth2 error: error="invalid_scope", error_description="Empty
scope (either the client or the user is not allowed the requested
scopes)"
```

3.资源服务器对于scope的访问控制

授权获取到授权码以后，申请访问令牌，通过访问令牌访问资源服务器。先看下这时认证信息都有些啥

```
authentication = (OAuth2Authentication@7831) *org.springframework.security.oauth2.provider.OAuth2Authentication@90f902ef: Principal: org.springframework.security.core.userdetails.User@36ebcb: Username: use
  storedRequest = (OAuth2Request@7869)
    resourceIds = (HashSet@7897) size = 0
    authorities = (HashSet@7898) size = 1
    approved = true
    refresh = null
    redirectUri = "https://www.baidu.com"
    responseTypes = (HashSet@7900) size = 1
    extensions = (HashMap@7901) size = 0
    clientId = "client"
    scope = (Collections$UnmodifiableSet@7903) size = 3
    requestParameters = (Collections$UnmodifiableMap@7904) size = 6
  userAuthentication = (UsernamePasswordAuthenticationToken@7870) *org.springframework.security.authentication.UsernamePasswordAuthenticationToken@567159b2: Principal: org.springframework.security.co
    principal = (User@7875) *org.springframework.security.core.userdetails.User@36ebcb: Username: user; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true; credentialsNonExpired: true; AccountNo
      password = "$2a$10$5Twm4rWbfzmX3sbFsQDSbeT8awhvXt23FAQsqN54mq4DEuVrwT94C"
      username = "user"
      authorities = (Collections$UnmodifiableSet@7883) size = 1
        accountNonExpired = true
        accountNonLocked = true
        credentialsNonExpired = true
        enabled = true
```

OAuth请求时候的信息

用户信息

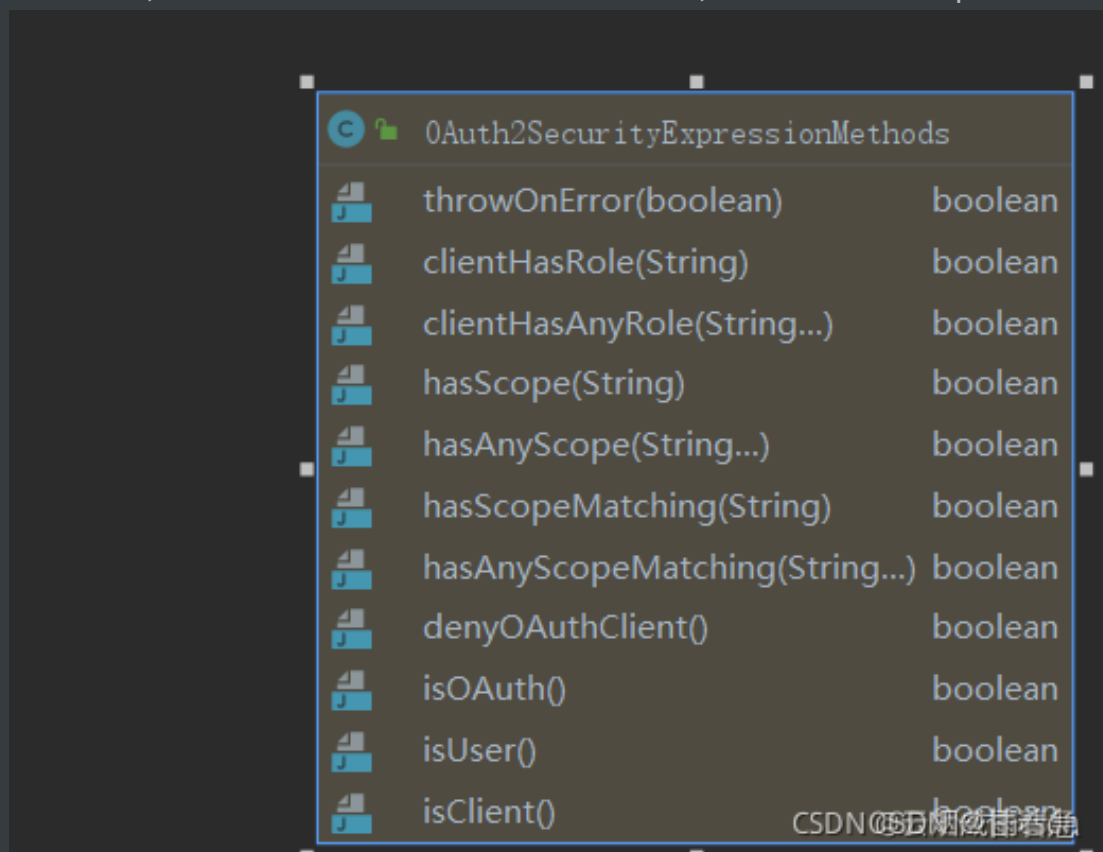
CSDN @云烟成雨

可以看到当前令牌对应的认证信息，包含授权码模式中的用户及Oauth信息，OAuth2Request对象，就保存了授权时的scope信息，如果没有授权的scope则不会出现在这里。

这样资源服务器也就获取到了scope数据，那么具体应该怎么使用scope进行访问控制呢？

之前有分析过Security 基于注解的权限控制@PreAuthorize等注解的使用方法，在spring-security-oauth也提供了相应的表达式，我们只需要在注解中使用oauth2相关的表达式就可以了。

在源码中，可以看到OAuth2相关的表达式写法，其中就有对scope作用域的访问控制。



以下例子，可以使用`hasScope`，表示当前OAuth应用，毕竟具有`admin:user`的作用域，否则会拒绝访问.

```
@GetMapping("/resource")
@PreAuthorize("#oauth2.hasScope('admin:user')") //
public String resource(){
    Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
    System.out.println(authentication.toString());
    return "访问到了resource 资源 ";
}
```

可以看到当前没有admin:user，所以去访问资源的服务器时会报错。

```
1 {
2   "access_token": "21d8d063-3b6b-4525-8dec-3f615e44089e_client",
3   "token_type": "bearer",
4   "expires_in": 41669,
5   "scope": "admin:org write:org read:org"
6 }
```

CSDN @甘着急

Body Cookies (1) Headers (10) Test Results

Pretty

Raw

Preview

Visualize

JSON



```
1 {
2   "error": "access_denied",
3   "error_description": "不允许访问"
4 }
```

CSDN @云烟成雨

二、总结

通过以上案例分析，Spring Security OAuth2除了使用resourceId对服务级别进行控制，也能基于scope 添加更小粒度的控制。

https://blog.csdn.net/jiangjun_dao519/article/details/125242434

[Spring Security OAuth2和RBAC权限授权_rbac oauth2-CSDN博客](#)

1、什么是RBAC?

RBAC是指基于角色的权限访问控制，通过权限与角色相关联，用户可以通过成为适当角色的成员而得到这些角色的权限。简单来说，RBAC认为权限授权的过程可以抽象为：

Who能否对What进行How的操作？并对这个逻辑表达式进行判断是否为True的求解过程，所以如果项目上需要设计实现权限管理模块，那必定要考虑RBAC的思想。

RBAC分3个组成部分：用户、角色和权限。

User（用户）：每个用户都有唯一的UID识别，并被授予不同的角色

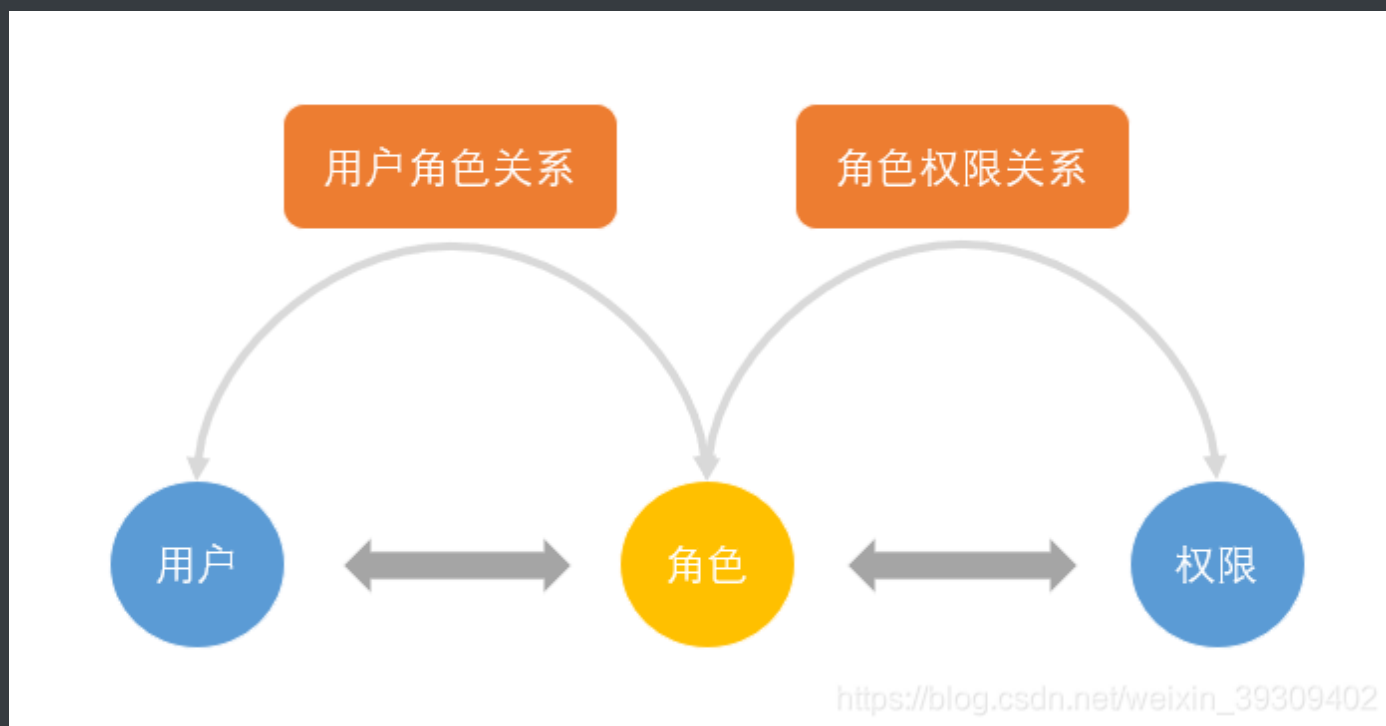
Role（角色）：不同角色具有不同的权限

Permission（权限）：访问权限

用户-角色映射：用户和角色之间的映射关系

角色-权限映射：角色和权限之间的映射

它们之间的关系如下图所示：



2、为什么开发权限管理都喜欢使用OAuth?

OAuth是关于授权的开放网络标准，在全世界得到广泛应用，目前的版本是2.0版。oauth2根据使用场景分4种实现模式：

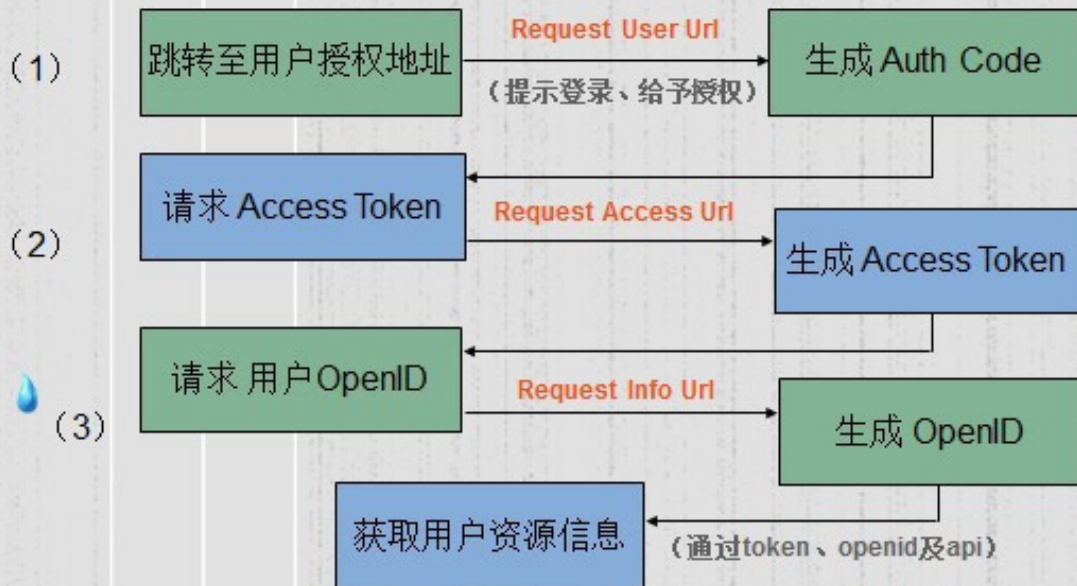
- 1、授权码模式
- 2、简化模式
- 3、密码模式
- 4、客户端模式

在实际项目中，我们通常使用授权码模式(微信登陆)

OAuth 2.0 认证流程

OAuth2.0的处理流程：

- 1、得到授权码code
- 2、获取access token
- 3、通过access token,获取OpenID
- 4、通过access token及OpenID调用API,获取用户授权信息



https://blog.csdn.net/weixin_39309402

OAuth2授权主要分两部分：

- 1、认证服务
- 2、资源服务

在实际项目中以上两个服务板块可以部署在同一台服务器，也可以分开部署，而且实际开发中，推荐使用Spring Security OAuth2的实现方式，因为配置灵活，代码开发方便，如果结合路由组件，能更好的实现微服务权限控制扩展。

本文转自 <https://www.cnblogs.com/Chary/p/18006353>，如有侵权，请联系删除。