链滴

深入理解 Spring Cloud Security OAuth2 及 JWT

作者：alex18595752445

原文链接：https://ld246.com/article/1591706840856

来源网站：链滴

深入理解Spring Cloud Security OAuth2及JWT

因项目需要，需要和三方的oauth2服务器进行集成。网上关于spring cloud security oauth2的相关料，一般都是讲如何配置，而能把这块原理讲透彻的比较少，这边自己做一下总结和整理，顺带介绍下JWT的使用场景。

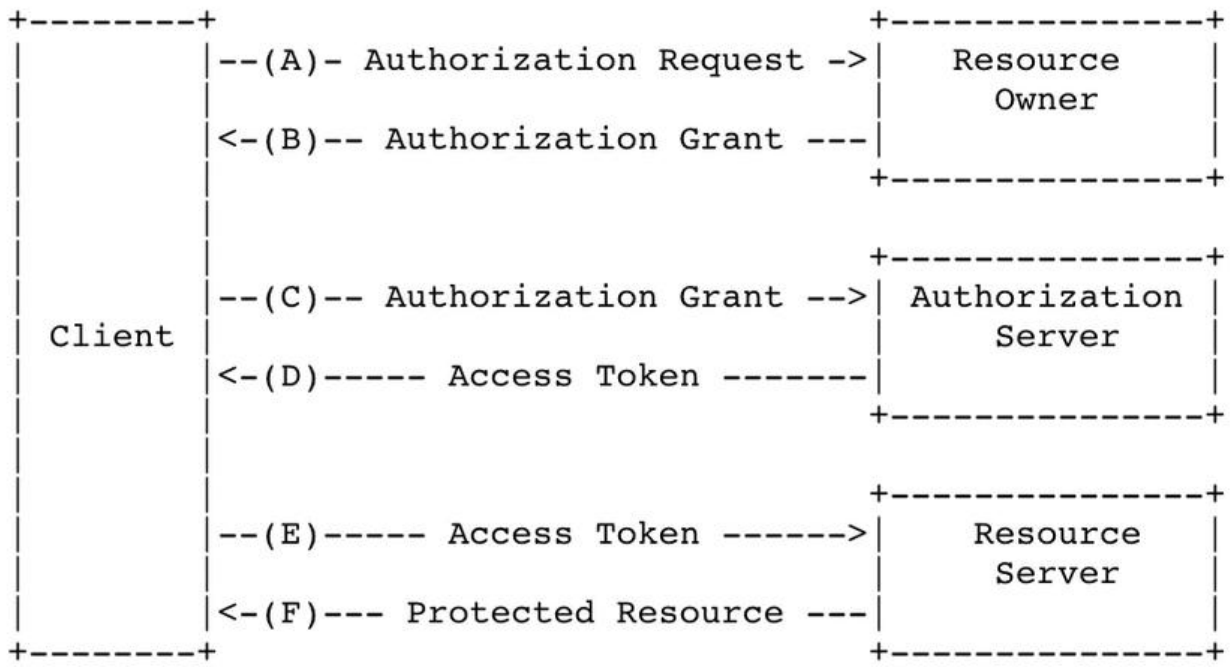
什么是OAuth2?

OAuth2是一个关于授权的开放标准，核心思路是通过各类认证手段（具体什么手段OAuth2不关心认证用户身份，并颁发token（令牌），使得第三方应用可以使用该令牌在**限定时间**、**限定范围**访问定资源。主要涉及的RFC规范有 RFC6749（整体授权框架）,RFC6750（令牌使用）,RFC6819（威胁型）这几个，一般我们需要了解的就是 RFC6749。获取令牌的方式主要有四种，分别是 授权码模式，单模式，密码模式和 客户端模式，如何获取token不在本篇文章的讨论范围，我们这里假定客户端已通过某种方式获取到了access_token，想了解具体的oauth2授权步骤可以移步阮一峰老师的理解OAuth 2.0，里面有非常详细的说明。

这里要先明确几个OAuth2中的几个重要概念：

- resource owner: 拥有被访问资源的用户
- user-agent: 一般来说就是浏览器
- client: 第三方应用
- Authorization server: 认证服务器，用来进行用户认证并颁发token
- Resource server：资源服务器，拥有被访问资源的服务器，需要通过token来确定是否有权限访问

明确概念后，就可以看OAuth2的协议握手流程，摘自RFC6749



Abstract Protocol Flow.png

什么是Spring Security?

Spring Security是一套安全框架，可以基于RBAC（基于角色的权限控制）对用户的访问权限进行控，核心思想是通过一系列的filter chain来进行拦截过滤，以下是ss中默认的内置过滤器列表，当然你可以通过 custom-filter来自定义扩展filter chain列表

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	ChannelProcessingFilter	http/intercept-url@requires-channel
SECURITY_CONTEXT_FILTER	SecurityContextPersistenceFilter	http
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	session-management/concurrency-control
HEADERS_FILTER	HeaderWriterFilter	http/headers
CSRF_FILTER	CsrfFilter	http/csrf
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509AuthenticationFilter	http/x509
PRE_AUTH_FILTER	AbstractPreAuthenticatedProcessingFilter	/A
CAS_FILTER	CasAuthenticationFilter	/A
FORM_LOGIN_FILTER	UsernamePasswordAuthenticationFilter	http/form-login
BASIC_AUTH_FILTER	BasicAuthenticationFilter	http/http-basic
SERVLET_API_SUPPORT_FILTER	SecurityContextHolderAwareRequestFilter	http/@servlet-api-provision
JAAS_API_SUPPORT_FILTER	JaasApiIntegrationFilter	http/@jaas-api-provision
REMEMBER_ME_FILTER	RememberMeAuthenticationFilter	http/remember-me
ANONYMOUS_FILTER	AnonymousAuthenticationFilter	http/anonymous
SESSION_MANAGEMENT_FILTER	SessionManagementFilter	session-management
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter	http
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserFilter	/A

这里面最核心的就是 FILTER_SECURITY_INTERCEPTOR，通过 FilterInvocationSecurityMetadataS

ur来进行资源权限的匹配，AccessDecisionManager来执行访问策略。

认证与授权（Authentication and Authorization）

一般意义来说的应用访问安全性，都是围绕认证（Authentication）和授权（Authorization）这两核心概念来展开的。即首先需要确定用户身份，在确定这个用户是否有访问指定资源的权限。认证这的解决方案很多，主流的有 CAS、SAML2、OAUTH2等（不巧这几个都用过-_-），我们常说的单点录方案（SSO）说的就是这块，授权的话主流的就是spring security和shiro。shiro我没用过，据说比较轻量级，相比较而言spring security确实架构比较复杂。

Spring Cloud Security Oauth2认证流程

将OAuth2和Spring Security集成，就可以得到一套完整的安全解决方案。

为了便于理解，现在假设有一个名叫“脸盆网”的社交网站，用户在首次登陆时会要求导入用户在facebook的好友列表，以便于快速建立社交关系。具体的授权流程如下：

1. 用户登陆脸盆网，脸盆网试图访问facebook上的好友列表

2. 脸盆网发现该资源是facebook的受保护资源，于是返回302将用户重定向至facebook登陆页面

3. 用户完成认证后，facebook提示用户是否将好友列表资源授权给脸盆网使用（如果本来就是已登陆acebook状态则直接显示是否授权的页面）

4. 用户确认后，脸盆网通过 授权码模式获取了facebook颁发的access_token

5. 脸盆网携带该token访问facebook的获取用户接口 https://api.facebook.com/user,facebook证token无误后返回了与**该token绑定**的用户信息

6. 脸盆网的spring security安全框架根据返回的用户信息构造出了principal对象并保存在session中

7. 脸盆网再次携带该token访问好友列表，facebook根据 **该token对应的用户**返回该用户的好友列信息

8. 该用户后续在脸盆网发起的访问facebook上的资源，只要在token有效期及权限范围内均可以正常取（比如想访问一下保存在facebook里的相册）

不难看出，这个假设的场景中，脸盆网就是第三方应用（client），而facebook既充当了认证服务器又充当了资源服务器。这个流程里面有几个比较重要的关键点，我需要重点说一下，而这也是其他的及spring security与OAuth2整合的文章中很少提及的，很容易云里雾里的地方。

细心的同学应该发现了，其实在标准的OAuth2授权过程中，5、6、8这几步都不是必须的，从上面的 RFC6749规范来看，只要有1、2、3、4、7这几步，就完成了被保护资源访问的整个过程。事实，RFC6749协议规范本身也并不关心用户身份的部分，它只关心token如何颁发，如何续签，如何用token访问被保护资源（facebook只要保证返回给脸盆网的就是当前用户的好友，至于当前用户是谁脸网不需要关心）。那为什么spring security还要做5、6这两步呢？这是因为spring security是一套完的安全框架，它必须关心用户身份！在实际的使用场景中，OAuth2一般不仅仅用来进行被保护资源访问，还会被用来做单点登陆（SSO）。在SSO的场景中，用户身份无疑就是核心，而token本身是携带用户信息的，这样client就没法知道认证服务器发的token到底对应的是哪个用户。设想一下这个景，脸盆网不想自建用户体系了，想直接用facebook的用户体系，facebook的用户和脸盆网的用户一对应（其实在很多中小网站现在都是这种模式，可以选择使用微信、QQ、微博等网站的用户直接陆），这种情况下，脸盆网在通过OAuth2的认证后，就希望拿到用户信息了。所以现在一般主流的Oauth2认证实现，都会预留一个用户信息获取接口，就是上面提到的 https://api.facebook.com/user(然这不是OAuth2授权流程中必须的)，这样client在拿到token后，就可以携带token通过这个接口获用户信息，完成SSO的整个过程。另外从用户体验的角度来说，如果获取不到用户信息，则意味者每要从脸盆网访问facebook的资源，都需要重定向一次进行认证，用户体验也不好。

OAuth2与SSO

首先要明确一点，**OAuth2并不是一个SSO框架，但可以实现SSO功能。**以下是一个使用github作为Auth2认证服务器的配置文件

```
server:
  port: 11001
security:
  user:
    password: user  # 直接登录时的密码
  ignored: /
  sessions: never   # session策略
  oauth2:
    sso:
      loginPath: /login   # 登录路径
    client:
      clientId: c40fb56cb4sdsdsdsd
      clientSecret: c910ec22981daa28e1b59c778sdfjh73j3
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

可以看到 accessTokenUri和 userAuthorizationUri都是为了完成OAuth2的授权流程所必须的配置而 userInfoUri则是spring security框架为了完成SSO所必须要的。所以总结一下就是：**通过将用户息这个资源设置为被保护资源，可以使用OAuth2技术实现单点登陆（SSO），而Spring Security Auth2就是这种OAuth2 SSO方案的一个实现。**

Spring Security在调用user接口成功后，会构造一个 OAuth2Authentication对象，这个对象是我们常使用的 UsernamePasswordAuthenticationToken对象的一个超集，里面封装了一个标准的 UsernamePasswordAuthenticationToken，同时在 detail中还携带了OAuth2认证中需要用到的一些关键息（比如 tokenValu，tokenTyp等），这时候就完成了SSO的登陆认证过程。后续用户如果再想访被保护资源，spring security只需要从principal中取出这个用户的token，再去访问资源服务器就行，而不需要每次进行用户授权。这里要注意的一点是**此时浏览器与client之间仍然是通过传统的cooki-session机制来保持会话，而非通过token。实际上在SSO的过程中，使用到token访问的只有clien与resource server之间获取user信息那一次，token的信息是保存在client的session中的，而不是用户本地。**这也是之前我没搞清楚的地方，以为浏览器和client之间也是使用token，绕了不少弯路，于Spring Security来说，**不管是用cas、saml2还是Oauth2来实现SSO，最后和用户建立会话保持方式都是一样的。**

OAuth2 SSO与CAS、SAML2的比较

根据前面所说，大家不难看出，OAuth2的SSO方案和CAS、SAML2这样的纯SSO框架是有本质区别。在CAS和SAML2中，没有资源服务器的概念，只有认证客户端（需要验证客户信息的应用）和认证务器（提供认证服务的应用）的概念。在CAS中这叫做 cas-client和 cas-server，SAML2中这叫做 Sevice Providers和 Identity Provider，可以看出CAS、SAML2规范天生就是为SSO设计的，在报文结上都考虑到了用户信息的问题（SAML2规范甚至还带了权限信息），而OAuth2本身不是专门为SSO计的，主要是为了解决资源第三方授权访问的问题，所以在用户信息方面，还需要额外提供一个接口。

Authorization Server与Resource Server分离

脸盆网的这个例子中，我们看到资源服务器和认证服务器是在一起的（都是facebook），在互联网

景下一般你很难找到一个独立的、权威的、第三方的认证中心（你很难想像腾讯的QQ空间通过支付的认证中心去授权，也很难想像使用谷歌服务要通过亚马逊去授权）。但是如果是在公司内部，这种景其实是很多的，尤其在微服务架构下，有大量服务会对外提供资源访问，他们都需要做权限控制。么最合理的当然就是建立一个统一的认证中心，而不是每个服务都做一个认证中心。我们前面也介绍，token本身是不携带用户信息的，在分离后resouce server在收到请求后，如何检验token的真实？又如何从token中获取对应的用户信息？这部分的介绍网上其实非常少，幸好我们可以直接从官方档获取相关的蛛丝马迹，官方文档对于resouce server的配置是这样描述的：

**

```
security:
  oauth2:
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

寥寥数语，但已经足够我们分析了。从这个配置可以看出，client在访问resource server的被保护资时，如果没有携带token，则资源服务器直接返回一个401未认证的错误

```
<oauth>
  <error_description>
    Full authentication is required to access this resource
  </error_description>
  <error>unauthorized</error>
</oauth>
```

如果携带了token，则资源服务器会使用这个token向认证服务器发起一个用户查询的请求，若token误或已经失效，则会返回

```
<oauth>
  <error_description>49e2c7d8720738cfb75f6b675d62e5ecd66</error_description>
  <error>invalid_token</error>
</oauth>
```

若token验证成功，则认证服务器向资源服务器返回对应的用户信息，此时resource server的spring security安全框架就可以按照标准的授权流程进行访问权限控制了。

认证与授权的解耦

从这个流程中我们可以看出，通过OAuth2进行SSO认证，有一个好处是做到了**认证与授权的解耦。**日常的使用场景来说，认证比较容易做到统一和抽象，毕竟你就是你，走到哪里都是你，但是你在不系统里面的角色，却可能千差万别（家里你是父亲，单位里你是员工，父母那里你是子女）。同时角的设计，又是和资源服务器的设计强相关的。从前面的配置中不难发现，如果希望获得为不同资源服器设计的角色，你只需要替换 https://api.facebook.com/user这个配置就行了，这为我们的权限控带来了更大的灵活性，而这是传统的比如SAML2这样的SSO框架做不到的。

JWT介绍

终于来到了著名的JWT部分了，JWT全称为Json Web Token，最近随着微服务架构的流行而越来越，号称新一代的认证技术。今天我们就来看一下，jwt的本质到底是什么。

我们先来看一下OAuth2的token技术有没有什么痛点，相信从之前的介绍中你也发现了，token技术大的问题是**不携带用户信息**，且资源服务器无法进行本地验证，每次对于资源的访问，资源服务器都

要向认证服务器发起请求，一是验证token的有效性，二是获取token对应的用户信息。如果有大量此类请求，无疑处理效率是很低的，且认证服务器会变成一个中心节点，对于SLA和处理性能等均有高的要求，这在分布式架构下是很要命的。

JWT就是在这样的背景下诞生的，从本质上来说，jwt就是一种**特殊格式**的token。普通的oauth2颁的就是一串随机hash字符串，本身无意义，而jwt格式的token是有特定含义的，分为三部分：

- 头部　Header
- 载荷　Payloa
- 签名　Signatur

这三部分均用base64进行编码，当中用 .进行分隔，一个典型的jwt格式的token类似 xxxxx.yyyyy.zzzz。关于jwt格式的更多具体说明，不是本文讨论的重点，大家可以直接去官网查看官方文档，这里不多赘述。

相信看到签名大家都很熟悉了，没错，jwt其实并不是什么高深莫测的技术，相反非常简单。认证服器通过对称或非对称的加密方式利用 payloa生成 signatur，并在 header中申明签名方式，仅此而已通过这种本质上极其传统的方式，jwt可以实现**分布式的token验证功能**，即资源服务器通过事先维护的对称或者非对称密钥（非对称的话就是认证服务器提供的公钥），直接在本地验证token，这种去心化的验证机制无疑很对现在分布式架构的胃口。jwt相对于传统的token来说，解决以下两个痛点：

- 通过验证签名，token的验证可以直接在本地完成，不需要连接认证服务器
- 在payload中可以定义用户相关信息，这样就轻松实现了token和用户信息的绑定

在上面的那个资源服务器和认证服务器分离的例子中，如果认证服务器颁发的是jwt格式的token，那资源服务器就可以直接自己验证token的有效性并绑定用户，这无疑大大提升了处理效率且减少了单隐患。

JWT适用场景与不适用场景

就像布鲁克斯在《人月神话》中所说的名言一样：“没有银弹”。JWT的使用上现在也有一种误区，为传统的认证方式都应该被jwt取代。事实上，jwt也不能解决一切问题，它也有适用场景和不适用场。

适用场景：

- 一次性的身份认证
- api的鉴权

这些场景能充分发挥jwt无状态以及分布式验证的优势

不适用的场景：

- 传统的基于session的用户会话保持

不要试图用jwt去代替session。这种模式下其实传统的session+cookie机制工作的更好，jwt因为其状态和分布式，事实上只要在有效期内，是无法作废的，用户的签退更多是一个客户端的签退，服务token仍然有效，你只要使用这个token，仍然可以登陆系统。另外一个问题是续签问题，使用token无疑令续签变得十分麻烦，当然你也可以通过redis去记录token状态，并在用户访问后更新这个状态但这就是硬生生把jwt的无状态搞成有状态了，而这些在传统的session+cookie机制中都是不需要去虑的。这种场景下，考虑高可用，我更加推荐采用分布式的session机制，现在已经有很多的成熟框可供选择了（比如spring session）。