**Spring Security 与 OAuth2（介绍）**

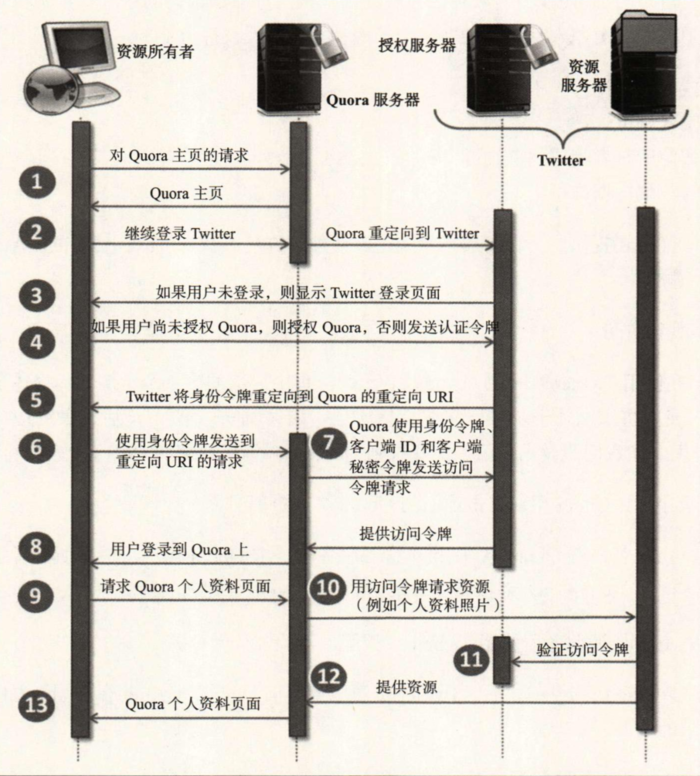
**摘要**：使用OAuth2 认证的好处就是你只需要一个账号密码，就能在各个网站进行访问，而面去了在每个网站都进行注册的繁琐过程，如：很多网站都可以使用微信登录，网站作为第三方服务、微信作为服务提供商

**OAuth2 角色**

* **resource owner**：资源所有者（指用户）
* **resource server**：资源服务器存放受保护资源，要访问这些资源，需要获得访问令牌（下面例子中的 Twitter 资源服务器）
* **client**：客户端代表请求资源服务器资源的第三方程序（下面例子中的 Quora）**客户端同时也可能是一个资源服务器**
* **authrization server**：授权服务器用于发放访问令牌给客户端（下面例子中的 Twitter 授权服务器）

**OAuth2 工作流程例子**

* **客户端 Quora 将自己注册到授权服务器上**
* 用户访问 Quora 主页，它显示了各种登陆选项
* 当用户点击使用 Twitter 登陆时，Quora 打开一个新窗口，将用户重定向到 Twitter 的登陆页面上
* 在这个新窗口中，用户使用他的账号密码登陆了 Twitter
* 如果用户之前未授权 Quora 应用程序使用他们的数据，则 Twitter 要求用户授权 Quora 来访问用户信息权限，如果用户已授权 Quora，此步骤则被跳过
* 经过正确的身份验证，Twitter **将用户和一个身份验证代码重定向到 Quora 的重定向 URI**
* **Quora 发送客户端 ID、客户端令牌和身份验证代码到 Twitter**
* Twitter 验证这些参数后，**将访问令牌发送到 Quora**
* 成功获取访问令牌后用户被登陆到 Quora 上，用户登录 Quora 后点击他们的个人资料页面
* Quora 从 Twitter 资源服务器请求用户的资源，并发送访问令牌
* **Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌**
* 成功验证访问令牌后，Twitter 资源服务器向 Quora 提供所需要的资源
* Quora 使用这些资源，并最终显示用户的个人资料页面

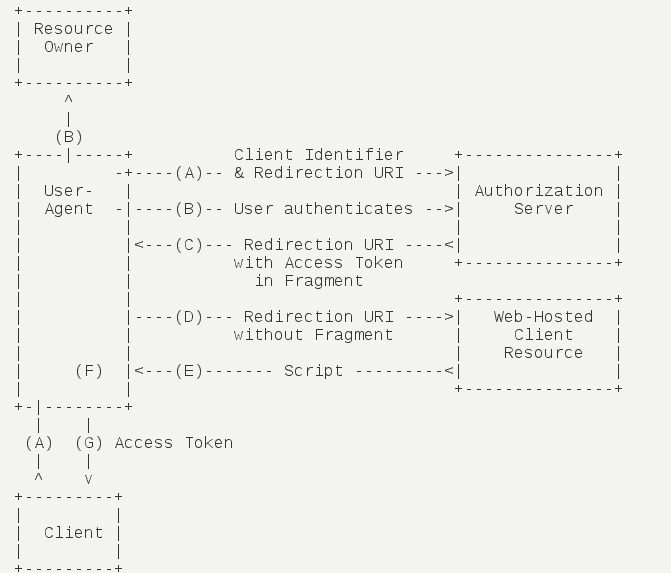


dtcoz.png

**OAuth2 授权模式（出自阮一峰OAuth2博客）**

**授权码模式**

* 授权码模式是功能最完整、流程最严密的授权模式，它的特点是通过客户端的后台服务器，与“服务器提供”的认证服务器进行互动

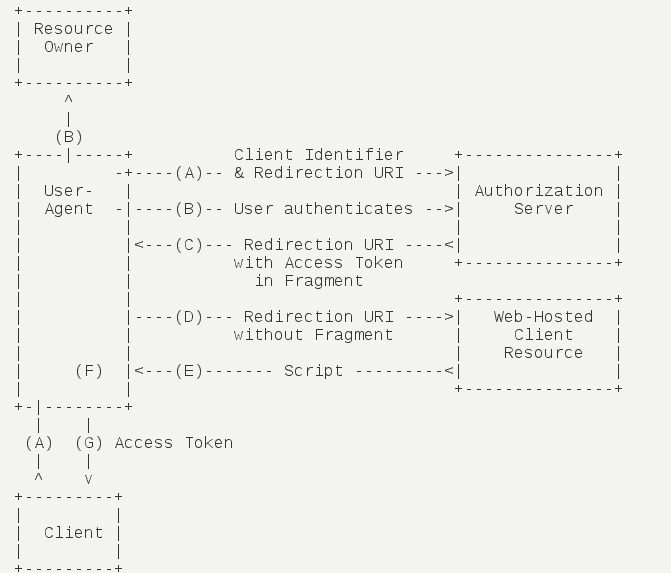


9434708-a464c0c64ca6ee9a.png

* 它的步骤如下：
  + （A）用户访问客户端，后者将前者导向认证服务器
  + （B）用户选择是否给予客户端授权
  + （C）假设用户给予授权，认证服务器将用户导向客户端事先指定的“重定向 URI”，同时附上一个授权码
  + （D）客户端收到授权码，附上早先的“重定向 URI”向认证服务器申请令牌，这一步是在客户端的后台服务器上完成的，对用户不可见
  + （E）认证服务器核对了授权码和重定向URI，确认无误后向客户端发送令牌和更新令牌
* 上述步骤中所需要的参数：
  + A步骤中，客户端申请认证的 URI，包含以下参数：
    - repsone\_type：授权类型，必选，此处固定值“code”
    - client\_id：客户端的ID，必选
    - client\_secret：客户端的密码，可选
    - redirect\_uri：重定向URI，可选
    - scope：申请的权限范围，可选
    - state：客户端当前的状态，可以指定任意值，认证服务器会原封不动的返回这个值
  + GET /authorize?response\_type=code&client\_id=s6BhdRkqt3&state=xyz
  + &redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
  + Host: server.example.com
  + C步骤中，服务器回应客户端的URI，包含以下参数：
    - code：表示授权码，必须按，该码有效期应该很短，通常10分钟，客户端只能使用一次，否则会被授权服务器拒绝，该码与客户端 ID 和 重定向 URI 是一一对应关系
    - state：如果客户端请求中包含着歌参数，认证服务器的回应也必须一模一样包含这个参数
  + HTTP/1.1 302 Found
  + Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
  + &state=xyz
  + D步骤中，客户端向认证服务器申请令牌的HTTP请求，包含以下参数：
    - grant\_type：表示使用的授权模式，必选，此处固定值为“authorization\_code”
    - code：表示上一步获得的授权吗，必选
    - redirect\_uri：重定向URI，必选，与步骤 A 中保持一致
    - client\_id：表示客户端ID，必选
  + POST /token HTTP/1.1
  + Host: server.example.com
  + Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  + Content-Type: application/x-www-form-urlencoded
  + grant\_type=authorization\_code&code=SplxlOBeZQQYbYS6WxSbIA
  + &redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
  + E步骤中，认证服务器发送的HTTP回复，包含以下参数：
    - access\_token：表示令牌，必选
    - token\_type：表示令牌类型，该值大小写不敏感，必选，可以是 bearer 类型或 mac 类型
    - expires\_in：表示过期时间，单位为秒，若省略该参数，必须设置其它过期时间
    - refresh\_token：表示更新令牌，用来获取下一次的访问令牌，可选
    - scope：表示权限范围
  + HTTP/1.1 200 OK
  + Content-Type: application/json;charset=UTF-8
  + Cache-Control: no-store
  + Pragma: no-cache
  + {
  + "access\_token":"2YotnFZFEjr1zCsicMWpAA",
  + "token\_type":"example",
  + "expires\_in":3600,
  + "refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  + "example\_parameter":"example\_value"
  + }
  + 从上面代码可以看到，参数使用 JSON 格式发送（Content-Type: application/json），才外，HTTP头信息中明确指定不得缓存

**简化模式**

* 简化模式不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了“授权码”这个步骤

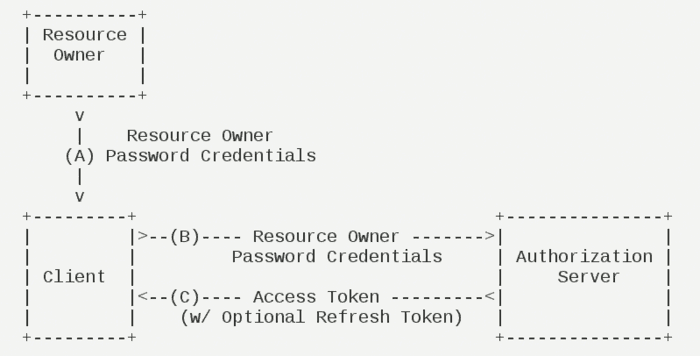


bg2014051205.png

* 它的步骤如下：
  + （A）客户端将用户导向认证服务器
  + （B）用户决定是否给予客户端授权
  + （C）假设用户给予授权，认证服务器将用户导向客户端指定的“重定向URI”，并在URI的Hash部分包含了访问令牌
  + （D）浏览器向资源服务器发出请求，其中不包括上一步收到的Hash值
  + （E）资源服务器返回一个网页，其中包含了代码可以获取Hash值中的令牌
  + （F）浏览器执行上一步获得的脚本，取出令牌
  + （G）浏览器将令牌发给客户端
* 上述步骤中所需要的参数：
  + A步骤中，客户端发出HTTP请求，包含以下参数：
    - response\_type：表示授权类型，此处固定值为"token"，必选
    - client\_id：表示客户端ID，必选
    - redirect\_uri：表示重定向URI，可选
    - scope：表示权限范围，可选
    - state：表示客户端当前状态，可指定任意值，认证服务器会原封不动返回这个值
  + GET /authorize?response\_type=token&client\_id=s6BhdRkqt3&state=xyz
  + &redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
  + Host: server.example.com
  + C步骤中，认证服务器回应客户端的URI，包含以下参数：
    - access\_token：表示访问令牌，必选
    - token\_type：表示令牌类型，该值大小写不敏感，必选
    - expires\_in：表示过期时间，单位为秒
    - scope：表示权限范围，如果与客户端申请的范围一致，可忽略
    - state：如果客户端请求中包含这个参数，认证服务器也要返回一模一样的参数
  + HTTP/1.1 302 Found
  + Location: http://example.com/cb#access\_token=2YotnFZFEjr1zCsicMWpAA
  + &state=xyz&token\_type=example&expires\_in=3600
* 上面例子中，认证服务器用HTTP头信息的Location栏，指定浏览器重定向的网址，注意，这个网址的Hash部分包含了令牌
* 根据D步骤，下一步浏览器会访问Location指定的网址，但是Hash部分不会被发送，接下来的E步骤，服务提供商的资源服务器发送过来的代码，提取出Hash令牌

**密码模式**

* 密码模式中，用户向客户端提供自己的用户名和密码，客户端使用这些信息向“服务提供商”索要授权
* 在这种模式中，用户必须把密码给客户端，但客户端不得存储密码，这通常在用户对客户端高端信任的情况下，比如客户端是操作系统的一部分，由一个著名的公司出品，而认证服务器只有在其它授权模式无法执行的情况下，才考虑该模式

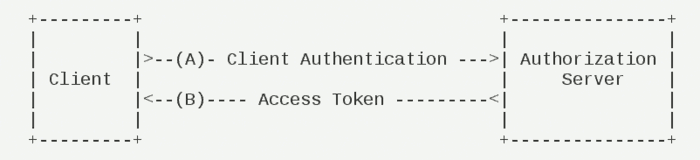


9434708-6165f69e2bfc8881.png

* 它的步骤如下：
  + （A）用户向客户端提供用户名和密码
  + （B）客户端将用户名密码发送认证给服务器，向后者请求令牌
  + （C）认证服务器确认无误后，向客户端提供访问令牌
* 上述步骤中所需要的参数：
  + B步骤中，客户端发出HTTP请求，包含以下参数：
    - grant\_type：授权类型，必选，此处固定值“password”
    - username：表示用户名，必选
    - password：表示用户密码，必选
    - scope：权限范围，可选
  + POST /token HTTP/1.1
  + Host: server.example.com
  + Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  + Content-Type: application/x-www-form-urlencoded
  + grant\_type=password&username=johndoe&password=A3ddj3w
  + C步骤中，认证服务器向客户端发送访问令牌，下面是一个例子：
  + HTTP/1.1 200 OK
  + Content-Type: application/json;charset=UTF-8
  + Cache-Control: no-store
  + Pragma: no-cache
  + {
  + "access\_token":"2YotnFZFEjr1zCsicMWpAA",
  + "token\_type":"example",
  + "expires\_in":3600,
  + "refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  + "example\_parameter":"example\_value"
  + }

**客户端模式**

* 客户端模式指客户端以自己名义，而不是用户名义，向“服务提供商”进行认证，严格地说，客户端模式不属于OAuth框架要解决的问题，在这种模式中，用户直接向客户端注册，客户端以自己名义要求“服务提供商”提供服务



bg2014051207.png

* 它的步骤如下：  
  -（A）:客户端向认证服务器进行身份认证，并要求一个访问令牌  
  -（B）:认证服务器确认无误后，向客户端提供访问令牌
* 上述步骤中所需要的参数：
  + A步骤中，客户端发出HTTP请求，包含以下参数：
    - granttype：表示授权类型，此处固定值为“clientcredentials”，必选
    - scope：表示权限范围，可选
  + POST /token HTTP/1.1
  + Host: server.example.com
  + Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  + Content-Type: application/x-www-form-urlencoded
  + grant\_type=client\_credentials
  + B步骤中，认证服务器向客户端发送访问令牌，下面是一个例子
  + HTTP/1.1 200 OK
  + Content-Type: application/json;charset=UTF-8
  + Cache-Control: no-store
  + Pragma: no-cache
  + {
  + "access\_token":"2YotnFZFEjr1zCsicMWpAA",
  + "token\_type":"example",
  + "expires\_in":3600,
  + "example\_parameter":"example\_value"
  + }

**更新令牌**

* 如果用户访问的时候，客户端“访问令牌”已经过期，则需要使用“更新令牌”申请一个新的令牌
* 客户端发出更新令牌请求，包含以下参数：
  + granttype：表示授权模式，此处固定值为“refreshtoken”，必选
  + refresh\_token：表示早前收到的更新令牌，必选
  + scope：表示申请权限范围，不得超出上一次申请的范围，若省略该参数，则表示与上一次一样
* POST /token HTTP/1.1
* Host: server.example.com
* Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
* Content-Type: application/x-www-form-urlencoded

grant\_type=refresh\_token&refresh\_token=tGzv3JOkF0XG5Qx2TlKWIA

# Spring Security 与 OAuth2（授权服务器）

### authrization-server(授权服务器)

#### 授权服务配置

* 配置一个授权服务，需要考虑 授权类型（GrantType）、不同授权类型为客户端（Client）提供了不同的获取令牌（Token）方式，每一个客户端（Client）都能够通过明确的配置以及权限来实现不同的授权访问机制，也就是说如果你提供了一个 “client\_credentials” 授权方式，并不意味着其它客户端就要采用这种方式来授权
* **使用 @EnableAuthorizationServer 来配置授权服务机制，并继承 AuthorizationServerConfigurerAdapter 该类重写 configure 方法定义授权服务器策略**

##### 配置客户端详情（Client Details）

* ClientDetailsServiceConfigurer 能够使用内存或 JDBC 方式实现获取已注册的客户端详情，有几个重要的属性：
  + clientId：客户端标识 ID
  + secret：客户端安全码
  + scope：客户端访问范围，默认为空则拥有全部范围
  + authorizedGrantTypes：客户端使用的授权类型，默认为空
  + authorities：客户端可使用的权限

##### 管理令牌（Managing Token）

* ResourceServerTokenServices 接口定义了令牌加载、读取方法
* AuthorizationServerTokenServices 接口定义了令牌的创建、获取、刷新方法
* ConsumerTokenServices 定义了令牌的撤销方法
* DefaultTokenServices 实现了上述三个接口,它包含了一些令牌业务的实现，如创建令牌、读取令牌、刷新令牌、获取客户端ID。默认的当尝试创建一个令牌时，是使用 UUID 随机值进行填充的，除了持久化令牌是委托一个 TokenStore 接口实现以外，这个类几乎帮你做了所有事情
* 而 TokenStore 接口也有一些实现：
  + InMemoryTokenStore：默认采用该实现，将令牌信息保存在内存中，易于调试
  + JdbcTokenStore：令牌会被保存近关系型数据库，可以在不同服务器之间共享令牌
  + JwtTokenStore：使用 JWT 方式保存令牌，它不需要进行存储，但是它撤销一个已经授权令牌会非常困难，所以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

##### JWT 令牌（JWT Tokens）

* 使用 JWT 令牌需要在授权服务中使用 JWTTokenStore，资源服务器也需要一个解码 Token 令牌的类 JwtAccessTokenConverter，JwtTokenStore 依赖这个类进行编码以及解码，因此授权服务以及资源服务都需要配置这个转换类
* Token 令牌默认是有签名的，并且资源服务器中需要验证这个签名，因此需要一个对称的 Key 值，用来参与签名计算
* 这个 Key 值存在于授权服务和资源服务之中，或者使用非对称加密算法加密 Token 进行签名，Public Key 公布在 /oauth/token\_key 这个 URL 中
* 默认 /oauth/token\_key 的访问安全规则是 "denyAll()" 即关闭的，可以注入一个标准的 SpingEL 表达式到 AuthorizationServerSecurityConfigurer 配置类中将它开启，例如 permitAll()
* **需要引入 spring-security-jwt 库**

##### 配置授权类型（Grant Types）

* 授权是使用 AuthorizationEndpoint 这个端点来进行控制的，使用 AuthorizationServerEndpointsConfigurer 这个对象实例来进行配置，默认是支持除了密码授权外所有标准授权类型，它可配置以下属性：
  + authenticationManager：认证管理器，当你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：可定义自己的 UserDetailsService 接口实现
  + authorizationCodeServices：用来设置收取码服务的（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：完全自定义授权服务实现（TokenGranter 接口实现），只有当标准的四种授权模式已无法满足需求时

##### 配置授权端点 URL（Endpoint URLs）

* AuthorizationServerEndpointsConfigurer 配置对象有一个 pathMapping() 方法用来配置端点的 URL，它有两个参数：
  + 参数一：端点 URL 默认链接
  + 参数二：替代的 URL 链接
* 下面是一些默认的端点 URL：
  + /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：提供公有密匙的端点，如果你使用JWT令牌的话
* 授权端点的 URL 应该被 Spring Security 保护起来只供授权用户访问

### 代码案例

引入依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-oauth2</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

#### 基于内存存储令牌

配置授权服务类，创建一个类继承 AuthorizationServerConfigurerAdapter 并添加 @EnableAuthorizationServer 注解，添加客户端信息

@Configuration

@EnableAuthorizationServer

public class OAuth2AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

//添加客户端信息

clients.inMemory() // 使用in-memory存储客户端信息

.withClient("client") // client\_id

.secret("secret") // client\_secret

.authorizedGrantTypes("authorization\_code") // 该client允许的授权类型

.scopes("app"); // 允许的授权范围

}

}

修改配置文件，设置 Security 密码为 password，用户名为 root，相当于一个资源拥有者（用户）的账号密码

security:

user:

name: root

password: 1234

server:

port: 8081

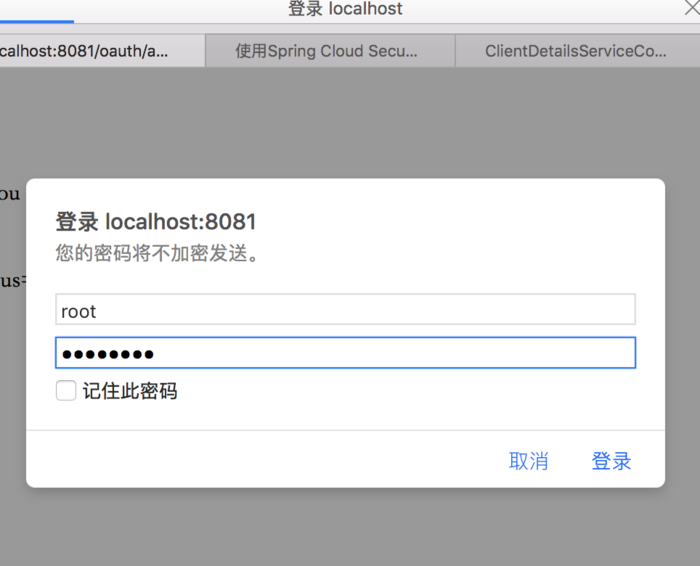
##### 测试

通过浏览器模拟客户端访问授权端点 /oauth/authorize

#（该步骤为\*\*授权码模式中的A\*\*），需要附上客户端申请认证的参数（\*\*A步骤中所包含的参数\*\*）

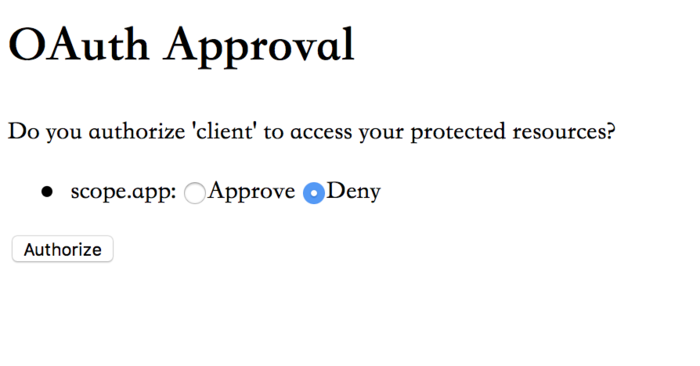
localhost:8081/oauth/authorize?client\_id=client&response\_type=code&redirect\_uri=http://www.baidu.com

进入用户登陆页面（该步骤为**授权码模式中的B**）



0gXCQ-1.png

输入 root 1234 登陆后会进入下面页面，询问用户是否授权客户端（该步骤为**授权码模式中的C**）



0g0kS.png

勾选授权后点击按钮会跳转到百度

#（\*\*A步骤中包含的参数定义了重定向URL\*\*），并在 URL 中包含一个授权码

https://www.baidu.com/?code=mhlA24

客户端拿到授权码后，附上先前设置的重定向 URL 向服务器申请令牌

# （该步骤为\*\*授权码模式中的D\*\*）,通过令牌端点 /oauth/token

# 使用 CURL 工具发送 POST 命令，授权码模式不需要 client\_sercet，因此该值可以为任意值

curl -X POST -H "Content-Type: application/x-www-form-urlencoded" -d 'grant\_type=authorization\_code&code=Li4NZo&redirect\_uri=http://www.baidu.com' "http://client:secret@localhost:8081/oauth/token"

返回令牌如下

{"access\_token":"d0e2f362-3bfd-43bb-a6ca-b6cb1b8ea9ee","token\_type":"bearer","expires\_in":43199,"scope":"app"}

#### 基于JDBC存储令牌

* Spring Cloud Security OAuth 已经为我们设计好了一套 Schema 和对应的 DAO 对象
* Spring Cloud Security OAuth2 通过 DefaultTokenServices 类来完成 token 生成、过期等 OAuth2 标准规定的业务逻辑，而 DefaultTokenServices 又是通过 TokenStore 接口完成对生成数据的持久化
* 在上面的 Demo 中，TokenStore 的默认实现为 InMemoryTokenStore 即内存存储，对于 Client 信息，ClientDetailsService 接口负责从存储仓库中读取数据，在上面的 Demo 中默认使用的也是 InMemoryClientDetailsService 实现类
* 要想使用数据库存储，只要提供这些接口的实现类即可，而框架已经为我们写好 JdbcTokenStore 和 JdbcClientDetailsService

##### 建表

* 框架已提前为我们设计好了数据库表，但对于 MYSQL 来说，默认建表语句中主键为 Varchar(256)，这超过了最大的主键长度，可改成 128，并用 BLOB 替换语句中的 LONGVARBINARY 类型

https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql

##### 代码

编写 @Configuration 类继承 AuthorizationServerConfigurerAdapter

@Configuration

@EnableAuthorizationServer

public class OAuth2AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private DataSource dataSource;

@Autowired

private TokenStore tokenStore;

private ClientDetailsService clientDetailsService;

@Bean // 声明TokenStore实现

public TokenStore tokenStore() {

return new JdbcTokenStore(dataSource);

}

@Bean // 声明 ClientDetails实现

public ClientDetailsService clientDetails() {

return new JdbcClientDetailsService(dataSource);

}

@Override // 配置框架应用上述实现

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

endpoints.authenticationManager(authenticationManager);

endpoints.tokenStore(tokenStore);

// 配置TokenServices参数

DefaultTokenServices tokenServices = new DefaultTokenServices();

tokenServices.setTokenStore(endpoints.getTokenStore());

tokenServices.setSupportRefreshToken(false);

tokenServices.setClientDetailsService(endpoints.getClientDetailsService());

tokenServices.setTokenEnhancer(endpoints.getTokenEnhancer());

tokenServices.setAccessTokenValiditySeconds( (int) TimeUnit.DAYS.toSeconds(30)); // 30天

endpoints.tokenServices(tokenServices);

}

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

clients.withClientDetails(clientDetailsService);

}

}

修改配置文件，并引入 MYSQL 和 JDBC 依赖库

spring:

datasource:

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://localhost:3306/client?useUnicode=yes&characterEncoding=UTF-8

username: root

password: 123456ly

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

往数据库 oauth\_client\_details 表添加客户端信息

https://upload-images.jianshu.io/upload_images/9434708-dbe8fe0a35c51f73.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/700

QfX0z.png

#### 基于JWT存储令牌

##### 对称加密，对称加密表示认证服务端和客户端的共用一个密钥

@Configuration

@EnableAuthorizationServer

public class OAuth2AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private TokenStore tokenStore;

//告诉Spring Security Token的生成方式

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

endpoints.tokenStore(tokenStore)

.accessTokenConverter(jwtAccessTokenConverter())

.authenticationManager(authenticationManager);

}

@Bean

public TokenStore tokenStore() {

return new JwtTokenStore(jwtAccessTokenConverter());

}

//使用同一个密钥来编码 JWT 中的 OAuth2 令牌

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter() {

JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

converter.setSigningKey("123");

return converter;

}

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

clients.inMemory() // 使用in-memory存储客户端信息

.withClient("client") // client\_id

.secret("secret") // client\_secret

.authorizedGrantTypes("authorization\_code") // 该client允许的授权类型

.scopes("app") // 允许的授权范围

.autoApprove(true); //登录后绕过批准询问(/oauth/confirm\_access)

}

}

##### 使用不对称的密钥来签署令牌

生成 JKS Java KeyStore 文件

keytool -genkeypair -alias mytest -keyalg RSA -keypass mypass -keystore mytest.jks -storepass mypass

导出公钥

keytool -list -rfc --keystore mytest.jks | openssl x509 -inform pem -pubkey

将公钥保存为 pubkey.txt，将 mytest.jks（）授权服务器） 和 pubkey.txt（资源服务器） 放到 resource 目录下

-----BEGIN PUBLIC KEY-----

MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhAF1qpL+8On3rF2M77lR

+l3WXKpGXIc2SwIXHwQvml/4SG7fJcupYVOkiaXj4f8g1e7qQCU4VJGvC/gGJ7sW

fn+L+QKVaRhs9HuLsTzHcTVl2h5BeawzZoOi+bzQncLclhoMYXQJJ5fULnadRbKN

HO7WyvrvYCANhCmdDKsDMDKxHTV9ViCIDpbyvdtjgT1fYLu66xZhubSHPowXXO15

LGDkROF0onqc8j4V29qy5iSnx8I9UIMEgrRpd6raJftlAeLXFa7BYlE2hf7cL+oG

hY+q4S8CjHRuiDfebKFC1FJA3v3G9p9K4slrHlovxoVfe6QdduD8repoH07jWULu

qQIDAQAB

-----END PUBLIC KEY-----

验证服务器配置

@Configuration

@EnableAuthorizationServer

public class OAuth2AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private TokenStore tokenStore;

//告诉Spring Security Token的生成方式

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

endpoints.tokenStore(tokenStore)

.accessTokenConverter(jwtAccessTokenConverter())

.authenticationManager(authenticationManager);

}

@Override

public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {

oauthServer

//允许所有资源服务器访问公钥端点（/oauth/token\_key）

//只允许验证用户访问令牌解析端点（/oauth/check\_token）

.tokenKeyAccess("permitAll()").checkTokenAccess("isAuthenticated()")

// 允许客户端发送表单来进行权限认证来获取令牌

.allowFormAuthenticationForClients();

}

@Bean

public TokenStore tokenStore() {

return new JwtTokenStore(jwtAccessTokenConverter());

}

@Bean

//使用私钥编码 JWT 中的 OAuth2 令牌

public JwtAccessTokenConverter jwtAccessTokenConverter() {

final JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(new ClassPathResource("mytest.jks"), "mypass".toCharArray());

converter.setKeyPair(keyStoreKeyFactory.getKeyPair("mytest"));

return converter;

}

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

clients.inMemory() // 测试用，将客户端信息存储在内存中

.withClient("client") // client\_id

.secret("secret") // client\_secret

.authorizedGrantTypes("authorization\_code") // 该client允许的授权类型

.scopes("app") // 允许的授权范围

.autoApprove(true); //登录后绕过批准询问(/oauth/confirm\_access)

}

}

#### 自定义令牌声明，添加额外的属性

添加一个额外的字段 "组织" 到令牌中

public class CustomTokenEnhancer implements TokenEnhancer {

@Override

public OAuth2AccessToken enhance(

OAuth2AccessToken accessToken,

OAuth2Authentication authentication) {

Map<String, Object> additionalInfo = new HashMap<>();

additionalInfo.put("organization", authentication.getName() + randomAlphabetic(4));

((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(additionalInfo);

return accessToken;

}

}

将把它连接到我们的授权服务器配置

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

TokenEnhancerChain tokenEnhancerChain = new TokenEnhancerChain();

tokenEnhancerChain.setTokenEnhancers(

Arrays.asList(tokenEnhancer(), accessTokenConverter()));

endpoints.tokenStore(tokenStore())

.tokenEnhancer(tokenEnhancerChain)

.authenticationManager(authenticationManager);

}

@Bean

public TokenEnhancer tokenEnhancer() {

return new CustomTokenEnhancer();

}

此时令牌如下

{

"user\_name": "john",

"scope": [

"foo",

"read",

"write"

],

"organization": "johnIiCh",

"exp": 1458126622,

"authorities": [

"ROLE\_USER"

],

"jti": "e0ad1ef3-a8a5-4eef-998d-00b26bc2c53f",

"client\_id": "fooClientIdPassword"

}

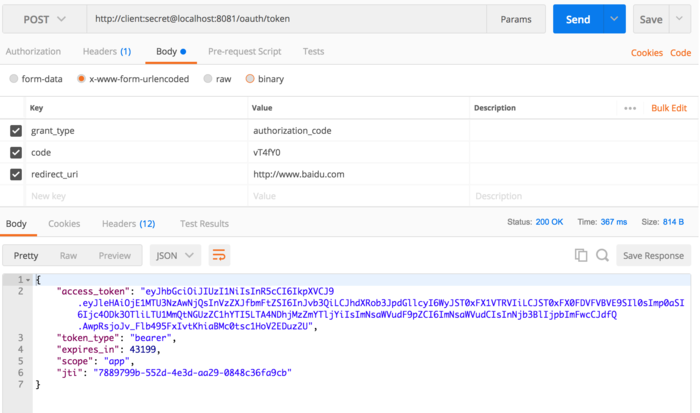
##### 测试

启动授权服务器、启动资源服务器

访问授权服务器 /oauth/authorize 端点获取授权码 code=vT4fY0

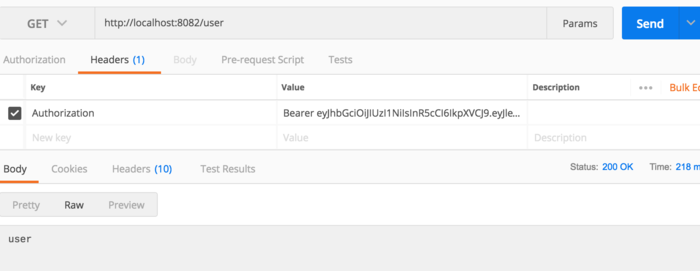
localhost:8081/oauth/authorize?client\_id=client&response\_type=code&redirect\_uri=http://www.baidu.com

访问授权服务器 /oauth/token 端点获取访问令牌



WX20180112-111436@2x.png

访问资源服务器受保护的资源，附上令牌在请求头，\*\*需加上 Bearer \*\*



WX20180112-111728@2x.png

# Spring Security 与 OAuth2（资源服务器）

### resource-server(资源服务器)

#### 资源服务器

* 要访问资源服务器受保护的资源需要携带令牌（从授权服务器获得）
* **客户端往往同时也是一个资源服务器，各个服务之间的通信（访问需要权限的资源）时需携带访问令牌**
* 资源服务器通过 @EnableResourceServer 注解来开启一个 **OAuth2AuthenticationProcessingFilter** 类型的过滤器
* 通过继承 ResourceServerConfigurerAdapter 类来配置资源服务器

#### ResourceServerProperties

* OAuth2 为资源服务器配置提供了 ResourceServerProperties 类，该类会读取配置文件中对资源服务器得配置信息（如授权服务器公钥访问地址）

#### ResourceServerSecurityConfigurer 可配置属性

* tokenServices：ResourceServerTokenServices 类的实例，用来实现令牌业务逻辑服务
* 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
* tokenExtractor 令牌提取器用来提取请求中的令牌
* 请求匹配器，用来设置需要进行保护的资源路径，默认的情况下是受保护资源服务的全部路径
* 受保护资源的访问规则，默认的规则是简单的身份验证（plain authenticated）
* 其他的自定义权限保护规则通过 HttpSecurity 来进行配置

#### 解析令牌方法：

* 使用 DefaultTokenServices 在资源服务器本地配置令牌存储、解码、解析方式
* 使用 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
* **若授权服务器是 JWT 非对称加密，则需要请求授权服务器的 /oauth/token\_key 来获取公钥 key 进行解码**

### 代码案例

#### 令牌解析（JWT 对称加密）

资源服务器和授权服务器不在同一个应用，则需告诉资源服务器令牌如何存储与解析，并与授权服务器使用相同的密钥进行解密

@Configuration

@EnableResourceServer

public class OAuth2ResourceConfig extends ResourceServerConfigurerAdapter{

@Bean

public TokenStore tokenStore() {

return new JwtTokenStore(jwtAccessTokenConverter());

}

//与授权服务器使用共同的密钥进行解析

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter() {

JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

converter.setSigningKey("123");

return converter;

}

}

#### 令牌解析（JWT 非对称加密）

* 非对称加密需要公钥，可以从本地获取，也可以从授权服务器提供的公钥端点获取
* 若本地获取不到公钥资源文件 pubkey.txt 则从授权服务器端点获取

@Configuration

@EnableResourceServer

public class OAuth2ResourceConfig extends ResourceServerConfigurerAdapter {

@Autowired

private ResourceServerProperties resourceServerProperties;

@Bean

public TokenStore tokenStore() {

return new JwtTokenStore(jwtAccessTokenConverter());

}

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter() {

JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

//设置用于解码的非对称加密的公钥

converter.setVerifierKey(getPubKey());

return converter;

}

private String getPubKey() {

Resource resource = new ClassPathResource("pubkey.txt");

try (BufferedReader br = new BufferedReader(new InputStreamReader(resource.getInputStream()))) {

System.out.println("本地公钥");

return br.lines().collect(Collectors.joining("\n"));

} catch (IOException ioe) {

return getKeyFromAuthorizationServer();

}

}

private String getKeyFromAuthorizationServer() {

ObjectMapper objectMapper = new ObjectMapper();

String pubKey = new RestTemplate().getForObject(resourceServerProperties.getJwt().getKeyUri(), String.class);

try {

Map map = objectMapper.readValue(pubKey, Map.class);

System.out.println("联网公钥");

return map.get("value").toString();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

}

#### 令牌解析（通过访问授权服务器解析令牌-适用 JDBC、内存存储）

* 资源服务器通过访问授权服务器 /oauth/check\_token 端点解析令牌需要使用 RemoteTokenServices
* 并且使用 DefaultAccessTokenConverter 来实现令牌数据的存储

@Autowired

private OAuth2ClientProperties oAuth2ClientProperties;

@Autowired

private AuthorizationServerProperties authorizationServerProperties;

@Bean

public ResourceServerTokenServices tokenServices() {

RemoteTokenServices remoteTokenServices = new RemoteTokenServices();

remoteTokenServices.setCheckTokenEndpointUrl(authorizationServerProperties.getCheckTokenAccess());

remoteTokenServices.setClientId(oAuth2ClientProperties.getClientId());

remoteTokenServices.setClientSecret(oAuth2ClientProperties.getClientSecret());

remoteTokenServices.setAccessTokenConverter(accessTokenConverter());

return remoteTokenServices;

}

@Bean

public AccessTokenConverter accessTokenConverter() {

return new DefaultAccessTokenConverter();

}

* 修改配置文件

security:

oauth2:

client:

clientId: resource1

clientSecret: secret

userAuthorizationUri: http://localhost:9005/oauth/authorize

grant-type: password

scope: read

access-token-uri: http://localhost:9005/oauth/token

resource:

userInfoUri: http://localhost:9005/user

authorization:

check-token-access: http://localhost:9005/oauth/check\_token

basic:

enabled: false

**Spring Security 与 OAuth2（客户端）**

**client(客户端) （改篇文章尚未修改，仅供参考）**

* OAuth2 客户端的实现方式没有太多任何规定，可自行编写登录逻辑
* 也可使用 OAuth2 提供的 **@EnableOAuth2Sso** 注解实现单点登录，该注解会添加身份验证过滤器替我们完成所有操作，只需在配置文件里添加授权服务器和资源服务器的配置即可

添加配置

server:

port: 8083

security:

oauth2:

sso:

loginPath: /login # 登录路径

client:

clientId: client

clientSecret: secret

userAuthorizationUri: http://localhost:8081/oauth/authorize

access-token-uri: http://localhost:8081/oauth/token

resource:

userInfoUri: http://localhost:8082/user

添加 Security 配置，并启动 @EnableOAuthSso

@Configuration

@EnableOAuth2Sso

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.

// 禁用 CSRF 跨站伪造请求，便于测试

csrf().disable()

// 验证所有请求

.authorizeRequests()

.anyRequest()

.authenticated()

//允许访问首页

.antMatchers("/","/login").permitAll()

.and()

// 设置登出URL为 /logout

.logout().logoutUrl("/logout").permitAll()

.logoutSuccessUrl("/")

.and()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

}

}

下面是测试用的控制类

@RestController

public class HelloController {

@GetMapping("/")

public String welcome() {

return "welcome";

}

}

* 测试

访问 localhost:9007/login

但此时会出现 Authentication Failed: Could not obtain access token

* 上面问题我查找了下，以下是某网友给出的答复

Centinul as you've figured out this happens due to a cookie conflict, unfortunately cookies don't respect the port numbers. And so both Apps interfere with each other since both are setting JSESSIONID. There are two easy workarounds:

1. use server.context-path to move each App to different paths, note that you need to do this for both

2. set the server.session.cookie.name for one App to something different, e.g., APPSESSIONID

I would suggest to put this workaround in a profile that you activate for localhost only.

* 修改配置文件，添加以下内容

# SESSION COOKIE 冲突

session:

cookie:

name: APPSESSIONID

**Spring Security 与 OAuth2（相关类参考）**

**AuthorizationServerConfigurerAdapter**

* 通过继承该类并重写 configure 方法来配置授权服务器：
  + configure(AuthorizationServerEndpointsConfigurer endpoints)：配置授权服务器端点，如令牌存储，令牌自定义，用户批准和授权类型，不包括端点安全配置
  + configure(AuthorizationServerSecurityConfigurer oauthServer)：配置授权服务器端点的安全
  + configure(ClientDetailsServiceConfigurer clients)：配置 ClientDetailsService 也就是客户端属性信息

**ResourceServerConfigurerAdapter**

* 通过继承该类并重写 configure 方法来配置资源服务器：
  + configure(org.springframework.security.config.annotation.web.builders.HttpSecurity http)：配置资源的访问规则
  + configure(ResourceServerSecurityConfigurer resources)：添加资源服务器特定的属性（如资源ID），默认值适用于很多情况，但至少需要修改下ID

**TokenStore**

* OAuth2 Token（令牌）持久化接口，用于定义 Token 如何存储，它有几个实现类：
  + InMemoryTokenStore：实现了在内存中存储令牌
  + JdbcTokenStore：通过 JDBC 方式存储令牌
  + JwtTokenStore：通过 JWT 方式存储令牌

**TokenEnhancer**

* 可用于自定义令牌策略，在令牌被 AuthorizationServerTokenServices 的实现存储之前增强令牌的策略，它有两个实现类：
  + JwtAccessTokenConverter：用于令牌 JWT 编码与解码
  + TokenEnhancerChain：一个令牌链，可以存放多个令牌，并循环的遍历令牌并将结果传递给下一个令牌

**TokenEndpoint**

* 该类根据 OAuth2 规范实现了令牌端点，客户端必须使用 Spring Security 身份验证来访问此端点，并从身份验证令牌客户端 ID，根据 OAuth2 规范，使用标准的 Spring Security 的 HTTP Basic 验证

**AuthorizationServerTokenServices**

* 该接口定义了一些操作可以对令牌进行一些管理，包含了三个方法声明：
  + createAccessToken(OAuth2Authentication authentication)：从 OAuth2Authentication 对象中创建令牌
  + getAccessToken(OAuth2Authentication authentication)：从 OAuth2Authentication 对象中获取令牌
  + refreshAccessToken(String refreshToken, TokenRequest tokenRequest)：刷新令牌

**OAuth2AuthenticationDetails**

* HTTP 请求中资源所有者（用户）相关的 OAuth2 Authentication（身份验证）信息，它可通过 SecurityContext 获得

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

Object details = authentication.getDetails();

* 它有几个方法：
  + getDecodedDetails()：
  + getRemoteAddress()：Authentication（身份验证）请求的 TCP/IP 地址
  + getSessionId()：Authentication（身份验证）请求的 SessionID
  + getTokenValue()：请求中包含的的令牌信息（通常在请求头中）
  + setDecodedDetails(Object decodedDetails)：设置令牌
  + getDecodedDetails()：解码请求中包含的令牌
  + getTokenType()：获取 Token 类型，例如 Bearer

**ResourceServerTokenServices**

* 令牌业务类接口，它定义了两个方法：
  + loadAuthentication(String accessToken)：加载 Authentication 验证信息
  + readAccessToken(String accessToken)：读取令牌
* 它有两个实现类：
  + DefaultTokenServices：详情看下面
  + RemoteTokenServices：通过访问授权服务器 /check\_token 端点来获取、解析令牌，如果端点返回 400 相应则表示该令牌无效

**DefaultTokenServices**

* 该类主要实现了 ResourceServerTokenServices、AuthorizationServerTokenServices、ConsumerTokenServices 三个接口
* 它包含了令牌业务的实现，如创建令牌、读取令牌、刷新令牌、撤销令牌、获取客户端ID。默认的当尝试创建一个令牌时，是使用 UUID 随机值进行填充的
* 除了持久化令牌是委托一个 TokenStore 接口实现以外，这个类几乎帮你做了所有事情

**AuthorizationEndpoint**

* 该类根据 OAuth2 规范实现了授权端点，如果授权类型是授权码模式，则处理用户批准，令牌是从令牌端点获得的，默认是支持除了密码授权外所有标准授权类型
* 它使用 AuthorizationServerEndpointsConfigurer 该类实例对象来进行配置，它可配置以下属性：
  + authenticationManager：认证管理器，当你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：可定义自己的 UserDetailsService 接口实现
  + authorizationCodeServices：用来设置收取码服务的（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：完全自定义授权服务实现（TokenGranter 接口实现），只有当标准的四种授权模式已无法满足需求时

**OAuth2AuthenticationProcessingFilter**

* OAuth2 受保护资源的预设认证过滤器，从请求中提取 OAuth2 令牌，并将令牌传入 OAuth2Authentication，（如果与OAuth2AuthenticationManager结合使用）传入Spring Security上下文

**ClientDetailsService**

* 提供有关 OAuth2 客户端的详细信息的服务，该接口仅声明了一个方法：
  + loadClientByClientId(String clientId)：通过客户端名字加载客户端详情
* 它有两个实现类：
  + InMemoryClientDetailsService：客户端详细信息服务的内存实现
  + JdbcClientDetailsService：客户端详细信息服务的 JDBC 实现，它会从数据库中读取客户端详情

**AccessTokenConverter**

* 令牌服务实现的转换器接口，用于将令牌数据存储在令牌中
* 它有两个实现类：
  + JwtAccessTokenConverter：用于令牌 JWT 编码与解码
  + DefaultAccessTokenConverter：AccessTokenConverter 的默认实现

**OAuth2AccessToken**

* OAuth2 Token（令牌）实体类，包含了令牌、类型（Bearer）、失效时间等

**OAuth2ClientProperties**

* OAuth2 Client（客户端）属性配置实体类

**OAuth2ProtectedResourceDetails**

* OAuth2 受保护资源（资源服务器）的配置实体类

**AuthorizationServerProperties**

* OAuth2 Authentication（授权服务器）属性配置实体类

**ResourceServerProperties**

* OAuth2 为资源服务器配置提供了 ResourceServerProperties 类，该类会读取配置文件中对资源服务器得配置信息（如授权服务器公钥访问地址）

**ClientDetails**

* 客户端详情接口，声明了获取客户端详情所需的一些方法，比如获取访问令牌失效时间、获取客户端权限、获取客户端 ID、获取客户端访问范围等方法

**AuthorizationServerProperties**

* OAuth2授权服务器的配置属性

**Spring Security 与 OAuth2（完整案例）**

**案例简述**

**简述：**

* 允许内存、数据库、JWT等方式存储令牌
* 允许 JWT 方式验证令牌
* 允许从内存、数据库中读取客户端详情
* 封装配置类，简化配置，通过注解方式定制使用何种令牌存储方式、客户端详情获取方式，可使用 JWT 方式存储令牌，从数据库中获取客户端详情
* 提供完整单元测试
* 较为详细的代码注释
* 允 **该代码尚未完善，仅供参考**
* 数据库 Schema : [https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql](https://link.jianshu.com/?t=https%3A%2F%2Fgithub.com%2Fspring-projects%2Fspring-security-oauth%2Fblob%2Fmaster%2Fspring-security-oauth2%2Fsrc%2Ftest%2Fresources%2Fschema.sql)
* Demo Git 地址：[https://gitee.com/LinYuanTongXue/OAuth2-Demo](https://link.jianshu.com/?t=https%3A%2F%2Fgitee.com%2FLinYuanTongXue%2FOAuth2-Demo)

**Demo 流程：**

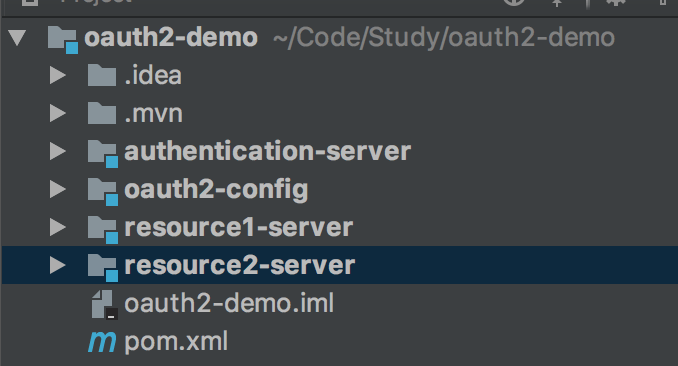
* 使用 OAuth2 密码授权方式提供令牌
* 资源服务器1（也为客户端）提供登录接口，资源所有者（用户）通过将个人账号密码提供给 资源服务器1，资源服务器1 通过该信息向授权服务器获取令牌
* 资源服务器1（也为客户端）通过令牌（其中包含了客户端、用户等信息）访问自身受保护的资源（需要权限才能查看的资源）
* 资源服务器2（也可资源服务器）不包含登录接口，但其提供了某些受保护的资源（需要资源服务器1带着访问令牌才能访问）
* 资源服务器1（也为客户端）通过令牌向 资源服务器2（资源服务器） 请求其受保护的资源

**使用**

* 授权服务器通过继承 **AuthServerConfig** 抽象类来配置授权服务器
* 资源服务器、客户端通过继承 **ResServerConfig** 抽象类来配置资源服务器
* Web 权限配置可继承 **AbstractSecurityConfig** 抽象类来简化配置
* 授权服务器通过在启动类添加以下注解来设置令牌存储、客户端信息获取方式
  + **@EnableAuthJWTTokenStore**：使用 JWT 存储令牌
  + **@EnableDBClientDetailsService**：通过数据库获取客户端详情
  + **@EnableDBTokenStore**：通过数据库存储令牌
* 资源服务器通过在启动类添加以下注解来设置令牌解析方式
  + **@EnableResJWTTokenStore**：使用 JWT 解析令牌
  + **@** **该代码尚未完善，仅供参考**

**项目结构**

* 下图是 Demo 项目结构，使用了 Maven 之间的继承关系，并添加了热部署，不了解的可以查看下 Git 上的 Demo 源码
  + oauth2-config：该包中定义了一些通用的类，例如授权服务器、资源服务器配置类，服务继承该类来简化配置
  + authentication-server：授权服务器
  + resource1-server：资源服务器1（也为客户端）
  + resource2-server：资源服务器2（也为资源服务器）



bd1je.png

**代码**

**oauth2-config（通用配置类库）**

权限枚举常量

/\*\*

\* @author: 林塬

\* @date: 2018/1/20

\* @description: 权限常量

\*/

public enum AuthoritiesEnum {

ADMIN("ROLE\_ADMIN"),

USER("ROLE\_USER"),

ANONYMOUS("ROLE\_ANONYMOUS");

private String role;

AuthoritiesEnum(String role) {

this.role = role;

}

public String getRole() {

return role;

}

}

授权服务器 JWT 方式存储令牌

/\*\*

\* @author: 林塬

\* @date: 2018/1/20

\* @description: 授权服务器 TokenStore 配置类，使用 JWT RSA 非对称加密

\*/

public class AuthJWTTokenStore {

@Bean("keyProp")

public KeyProperties keyProperties(){

return new KeyProperties();

}

@Resource(name = "keyProp")

private KeyProperties keyProperties;

@Bean

public TokenStore tokenStore(JwtAccessTokenConverter jwtAccessTokenConverter) {

return new JwtTokenStore(jwtAccessTokenConverter);

}

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter() {

JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

KeyPair keyPair = new KeyStoreKeyFactory

(keyProperties.getKeyStore().getLocation(), keyProperties.getKeyStore().getSecret().toCharArray())

.getKeyPair(keyProperties.getKeyStore().getAlias());

converter.setKeyPair(keyPair);

return converter;

}

}

资源服务器 JWT 方式解析令牌

/\*\*

\* @author: 林塬

\* @date: 2018/1/20

\* @description: 资源服务器 TokenStore 配置类，使用 JWT RSA 非对称加密

\*/

public class ResJWTTokenStore {

private static final String PUBLIC\_KEY = "pubkey.txt";

@Autowired

private ResourceServerProperties resourceServerProperties;

@Bean

public TokenStore tokenStore(JwtAccessTokenConverter jwtAccessTokenConverter) {

return new JwtTokenStore(jwtAccessTokenConverter);

}

@Bean

public JwtAccessTokenConverter jwtAccessTokenConverter() {

JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

converter.setVerifierKey(getPubKey());

return converter;

}

/\*\*

\* 获取非对称加密公钥 Key

\* @return 公钥 Key

\*/

private String getPubKey() {

Resource resource = new ClassPathResource(ResJWTTokenStore.PUBLIC\_KEY);

try (BufferedReader br = new BufferedReader(new InputStreamReader(resource.getInputStream()))) {

return br.lines().collect(Collectors.joining("\n"));

} catch (IOException ioe) {

return getKeyFromAuthorizationServer();

}

}

/\*\*

\* 通过访问授权服务器获取非对称加密公钥 Key

\* @return 公钥 Key

\*/

private String getKeyFromAuthorizationServer() {

ObjectMapper objectMapper = new ObjectMapper();

String pubKey = new RestTemplate().getForObject(resourceServerProperties.getJwt().getKeyUri(), String.class);

try {

Map map = objectMapper.readValue(pubKey, Map.class);

return map.get("value").toString();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

}

数据库方式存储令牌

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 使用数据库存取令牌

\*/

public class DBTokenStore {

@Autowired

private DataSource dataSource;

@Bean

public TokenStore tokenStore(){

return new JdbcTokenStore(dataSource);

}

}

数据库方式加载客户端详情

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 通过数据库加载客户端详情

\*/

public class DBClientDetailsService {

@Autowired

private DataSource dataSource;

@Bean

public ClientDetailsService clientDetailsService(){

return new JdbcClientDetailsService(dataSource);

}

}

授权服务器解析令牌

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 通过访问远程授权服务器 check\_token 端点验证令牌

\*/

public class RemoteTokenService {

@Autowired

private OAuth2ClientProperties oAuth2ClientProperties;

@Resource(name = "authServerProp")

private AuthorizationServerProperties authorizationServerProperties;

@Bean(name = "authServerProp")

public AuthorizationServerProperties authorizationServerProperties(){

return new AuthorizationServerProperties();

}

@Bean

public ResourceServerTokenServices tokenServices() {

RemoteTokenServices remoteTokenServices = new RemoteTokenServices();

remoteTokenServices.setCheckTokenEndpointUrl(authorizationServerProperties.getCheckTokenAccess());

remoteTokenServices.setClientId(oAuth2ClientProperties.getClientId());

remoteTokenServices.setClientSecret(oAuth2ClientProperties.getClientSecret());

remoteTokenServices.setAccessTokenConverter(accessTokenConverter());

return remoteTokenServices;

}

@Bean

public AccessTokenConverter accessTokenConverter() {

return new DefaultAccessTokenConverter();

}

}

WebSecurity 权限类

/\*\*

\* @author: 林塬

\* @date: 2018/1/20

\* @description: Web 权限配置类

\*/

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)

public abstract class AbstractSecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private AuthenticationManagerBuilder authenticationManagerBuilder;

@Autowired

private UserDetailsService userDetailsService;

@PostConstruct

public void init() {

try {

authenticationManagerBuilder

.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder());

} catch (Exception e) {

throw new BeanInitializationException("Security configuration failed", e);

}

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.exceptionHandling()

.and()

.csrf()

.disable()

.headers()

.frameOptions()

.disable()

.and()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.authorizeRequests()

.antMatchers("/swagger-ui/index.html").hasAuthority(AuthoritiesEnum.ADMIN.getRole());

}

授权服务器配置类

/\*\*

\* @author: 林塬

\* @date: 2018/1/20

\* @description: OAuth2 授权服务器配置类

\*/

@EnableAuthorizationServer

public abstract class AuthServerConfig extends AuthorizationServerConfigurerAdapter {

@Autowired

private ApplicationContext applicationContext;

@Autowired

private TokenStore tokenStore;

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private UserDetailsService userDetailsService;

@Autowired(required = false)

private JdbcClientDetailsService jdbcClientDetailsService;

//令牌失效时间

public int accessTokenValiditySeconds;

//刷新令牌失效时间

public int refreshTokenValiditySeconds;

//是否可以重用刷新令牌

public boolean isReuseRefreshToken;

//是否支持刷新令牌

public boolean isSupportRefreshToken;

public AuthServerConfig(int accessTokenValiditySeconds, int refreshTokenValiditySeconds, boolean isReuseRefreshToken, boolean isSupportRefreshToken) {

this.accessTokenValiditySeconds = accessTokenValiditySeconds;

this.refreshTokenValiditySeconds = refreshTokenValiditySeconds;

this.isReuseRefreshToken = isReuseRefreshToken;

this.isSupportRefreshToken = isSupportRefreshToken;

}

/\*\*

\* 配置授权服务器端点，如令牌存储，令牌自定义，用户批准和授权类型，不包括端点安全配置

\* @param endpoints

\* @throws Exception

\*/

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

Collection<TokenEnhancer> tokenEnhancers = applicationContext.getBeansOfType(TokenEnhancer.class).values();

TokenEnhancerChain tokenEnhancerChain=new TokenEnhancerChain();

tokenEnhancerChain.setTokenEnhancers(new ArrayList<>(tokenEnhancers));

DefaultTokenServices defaultTokenServices = new DefaultTokenServices();

defaultTokenServices.setReuseRefreshToken(isReuseRefreshToken);

defaultTokenServices.setSupportRefreshToken(isSupportRefreshToken);

defaultTokenServices.setTokenStore(tokenStore);

defaultTokenServices.setAccessTokenValiditySeconds(accessTokenValiditySeconds);

defaultTokenServices.setRefreshTokenValiditySeconds(refreshTokenValiditySeconds);

defaultTokenServices.setTokenEnhancer(tokenEnhancerChain);

//若通过 JDBC 存储令牌

if (Objects.nonNull(jdbcClientDetailsService)){

defaultTokenServices.setClientDetailsService(jdbcClientDetailsService);

}

endpoints

.authenticationManager(authenticationManager)

.userDetailsService(userDetailsService)

.tokenServices(defaultTokenServices);

}

/\*\*

\* 配置授权服务器端点的安全

\* @param oauthServer

\* @throws Exception

\*/

@Override

public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {

oauthServer

.tokenKeyAccess("permitAll()")

.checkTokenAccess("permitAll()")

.allowFormAuthenticationForClients();

}

}

资源服务器配置类

/\*\*

\* @author: 林塬

\* @date: 2018/1/20

\* @description: OAuth2 资源服务器配置类

\*/

@EnableResourceServer

@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)

public abstract class ResServerConfig extends ResourceServerConfigurerAdapter {

@Autowired(required = false)

private RemoteTokenServices remoteTokenServices;

@Autowired

private OAuth2ClientProperties oAuth2ClientProperties;

@Bean

@Qualifier("authorizationHeaderRequestMatcher")

public RequestMatcher authorizationHeaderRequestMatcher() {

return new RequestHeaderRequestMatcher("Authorization");

}

@Override

public void configure(HttpSecurity http) throws Exception {

http

.csrf()

.disable()

.exceptionHandling()

.and()

.headers()

.frameOptions()

.disable()

.and()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.requestMatcher(authorizationHeaderRequestMatcher());

}

@Override

public void configure(ResourceServerSecurityConfigurer resources) throws Exception {

super.configure(resources);

resources.resourceId(oAuth2ClientProperties.getClientId());

if (Objects.nonNull(remoteTokenServices)) {

resources.tokenServices(remoteTokenServices);

}

}

}

注解

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 在启动类上添加该注解来----开启 JWT 令牌存储（授权服务器-非对称加密）

\*/

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(AuthJWTTokenStore.class)

public @interface EnableAuthJWTTokenStore {

}

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 在启动类上添加该注解来----开启从数据库加载客户端详情

\*/

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(DBClientDetailsService.class)

public @interface EnableDBClientDetailsService {

}

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 在启动类上添加该注解来----开启通过数据库存储令牌

\* 数据库 schema ：https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql

\*/

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(DBTokenStore.class)

public @interface EnableDBTokenStore {

}

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 在启动类上添加该注解来----开启通过授权服务器验证访问令牌（适用于 JDBC、内存存储令牌）

\*/

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(RemoteTokenService.class)

public @interface EnableRemoteTokenService {

}

/\*\*

\* @author: 林塬

\* @date: 2018/1/22

\* @description: 在启动类上添加该注解来----开启 JWT 令牌存储（资源服务器-非对称加密）

\*/

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(ResJWTTokenStore.class)

public @interface EnableResJWTTokenStore {

}

**authentication-server（授权服务器）**

授权服务器配置

/\*\*

\* @author: 林塬

\* @date: 2018/1/10

\* @description: OAuth2 授权服务器配置

\*/

@Configuration

public class AuthorizationServerConfig extends AuthServerConfig {

/\*\*

\* 调用父类构造函数，设置令牌失效日期等信息

\*/

public AuthorizationServerConfig() {

super((int)TimeUnit.DAYS.toSeconds(1), 0, false, false);

}

/\*\*

\* 配置客户端详情

\* @param clients

\* @throws Exception

\*/

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

super.configure(clients);

clients.inMemory() // 使用内存存储客户端信息

.withClient("resource1") // client\_id

.secret("secret") // client\_secret

.authorizedGrantTypes("authorization\_code","password") // 该client允许的授权类型

.scopes("read") // 允许的授权范围

.autoApprove(true); //登录后绕过批准询问(/oauth/confirm\_access)

}

}

WebSecurity 权限配置

/\*\*

\* @author: 林塬

\* @date: 2018/1/19

\* @description: 权限配置

\*/

@Configuration

public class WebSecurityConfig extends AbstractSecurityConfig {

}

UserDetailsService 实现

/\*\*

\* @author: 林塬

\* @date: 2018/1/9

\* @description: 用户信息获取

\*/

@Service

public class UserDetailsService implements org.springframework.security.core.userdetails.UserDetailsService {

/\*\*

\* 通过 Username 加载用户详情

\* @param username 用户名

\* @return UserDetails

\* @throws UsernameNotFoundException

\*/

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

if (username.equals("linyuan")) {

PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

String password = passwordEncoder.encode("123456");

UserDetails userDetails = new User("linyuan",

password,

AuthorityUtils.commaSeparatedStringToAuthorityList(AuthoritiesEnum.USER.getRole()));

return userDetails;

}

return null;

}

}

启动类

@SpringBootApplication

@EnableAuthJWTTokenStore // 使用 JWT 存储令牌

//@EnableDBClientDetailsService //从 JDBC 加载客户端详情,需配置在启动类上，若在子类上会出现顺序问题，导致 Bean 创建失败

public class AuthenticationServerApplication {

public static void main(String[] args) {

SpringApplication.run(AuthenticationServerApplication.class, args);

}

}

微服务配置

server:

port: 9005

encrypt:

key-store:

location: mytest.jks

secret: mypass

alias: mytest

# 若从数据库中获取客户端信息则需配置数据库源

#spring:

# datasource:

# driver-class-name: org.h2.Driver

# url: jdbc:h2:mem:test

# username: sa

# h2:

# console:

# enabled: true

单元测试

/\*\*

\* @author: 林塬

\* @date: 2018/1/16

\* @description: 令牌单元测试

\*/

@Slf4j

@RunWith(SpringRunner.class)

@SpringBootTest(classes = AuthenticationServerApplication.class)

@AutoConfigureMockMvc

public class TokenControllerTest {

@Autowired

private MockMvc mockMvc;

/\*\*

\* 密码授权模式获取令牌

\*

\* @throws Exception

\*/

@Test

public void getToken() throws Exception {

MultiValueMap<String, String> map = new LinkedMultiValueMap<>();

map.put("username", Collections.singletonList("linyuan"));

map.put("password", Collections.singletonList("123456"));

map.put("grant\_type", Collections.singletonList("password"));

map.put("scope", Collections.singletonList("read"));

int status = this.mockMvc.perform(

post("/oauth/token")

.header("Authorization", "Basic " + Base64.getEncoder().encodeToString("resource1:secret".getBytes()))

.params(map)

.contentType(MediaType.MULTIPART\_FORM\_DATA)

.accept(MediaType.APPLICATION\_JSON)

).andDo(print()).andReturn().getResponse().getStatus();

switch (status) {

case HttpStatus.SC\_OK:

log.info("密码授权模式获取令牌---------------->成功（200）");

break;

case HttpStatus.SC\_UNAUTHORIZED:

log.info("密码授权模式获取令牌---------------->失败（401---没有权限，请检查验证信息，账号是否存在、客户端信息）");

break;

case HttpStatus.SC\_BAD\_REQUEST:

log.info("密码授权模式获取令牌---------------->失败（400---请求失败，请检查密码是否正确）");

break;

default:

log.info("密码授权模式获取令牌---------------->失败（{}---未知结果）",status);

break;

}

Assert.assertEquals(status,HttpStatus.SC\_OK);

}

}

**resource2-server（资源服务器2）**

资源服务器配置

/\*\*

\* @author: 林塬

\* @date: 2018/1/11

\* @description: 资源服务器访问权限配置

\*/

@Configuration

public class WebSecurityConfig extends ResServerConfig{

@Override

public void configure(HttpSecurity http) throws Exception {

super.configure(http);

http.authorizeRequests()

//访问受保护资源 /res 的要求：客户端 Scope 为 read，用户本身角色为 USER

.antMatchers("/res")

.access("#oauth2.hasScope('read') and hasRole('USER')");

}

}

受保护资源

/\*\*

\* @author: 林塬

\* @date: 2018/1/16

\* @description: 资源服务器2-资源接口

\*/

@RestController

public class ResController {

@GetMapping("/res")

public ResponseEntity<String> res(){

return ResponseEntity.ok("<h1>这是资源服务器2的受保护的资源</h1>");

}

}

启动类

@SpringBootApplication

@EnableResJWTTokenStore //OAuth2 使用 JWT 解析令牌

public class Resource2ServerApplication {

public static void main(String[] args) {

SpringApplication.run(Resource2ServerApplication.class, args);

}

}

微服务配置

server:

port: 9006

security:

oauth2:

resource:

jwt:

key-uri: http://localhost:9005/oauth/token\_key

**resource1-server（资源服务器1（客户端））**

资源服务器访问权限配置

/\*\*

\* @author: 林塬

\* @date: 2018/1/18

\* @description: 资源服务器访问权限配置

\*/

@Configuration

public class WebSecurityConfig extends ResServerConfig {

@Override

public void configure(HttpSecurity http) throws Exception {

super.configure(http);

http

.authorizeRequests()

.antMatchers("/login").permitAll()

.antMatchers("/res","/res2/res")

.access("#oauth2.hasScope('read') and hasRole('USER')");

}

}

用户登录数据传输对象

/\*\*

\* @author: 林塬

\* @date: 2018/1/15

\* @description: 用户登录数据传输对象

\*/

@Data

public class LoginDTO implements Serializable {

@NotBlank(message = "用户名不能为空")

private String username;

@NotBlank(message = "密码不能为空")

private String password;

}

资源服务器受保护资源

/\*\*

\* @author: 林塬

\* @date: 2018/1/16

\* @description: 资源服务器1-资源接口

\*/

@RestController

@AllArgsConstructor

public class ResController {

private RestTemplate restTemplate;

@GetMapping("/res")

public ResponseEntity<String> res(){

return ResponseEntity.ok("<h1>这是资源服务器1的受保护的资源</h1>");

}

/\*\*

\* 访问资源服务器2-资源接口

\* @param httpReq

\* @return

\*/

@GetMapping("/res2/res")

public ResponseEntity<String> remoteRes(HttpServletRequest httpReq){

//HttpEntity

HttpHeaders httpHeaders = new HttpHeaders();

httpHeaders.set("Authorization",httpReq.getHeader("Authorization"));

HttpEntity httpEntity = new HttpEntity(httpHeaders);

//请求资源服务器2的资源

return restTemplate.exchange("http://localhost:9006/res",HttpMethod.GET,httpEntity,String.class);

}

}

令牌管理接口

/\*\*

\* @author: 林塬

\* @date: 2018/1/16

\* @description: 令牌管理接口

\*/

@RestController

@AllArgsConstructor

public class TokenController {

private OAuth2ClientProperties oAuth2ClientProperties;

private OAuth2ProtectedResourceDetails oAuth2ProtectedResourceDetails;

private RestTemplate restTemplate;

/\*\*

\* 通过密码授权方式向授权服务器获取令牌

\* @param loginDTO

\* @param bindingResult

\* @return

\* @throws Exception

\*/

@PostMapping(value = "/login")

public ResponseEntity<OAuth2AccessToken> login(@RequestBody @Valid LoginDTO loginDTO, BindingResult bindingResult) throws Exception{

if (bindingResult.hasErrors()) {

throw new Exception("登录信息格式错误");

} else {

//Http Basic 验证

String clientAndSecret = oAuth2ClientProperties.getClientId()+":"+oAuth2ClientProperties.getClientSecret();

//这里需要注意为 Basic 而非 Bearer

clientAndSecret = "Basic "+Base64.getEncoder().encodeToString(clientAndSecret.getBytes());

HttpHeaders httpHeaders = new HttpHeaders();

httpHeaders.set("Authorization",clientAndSecret);

//授权请求信息

MultiValueMap<String, String> map = new LinkedMultiValueMap<>();

map.put("username", Collections.singletonList(loginDTO.getUsername()));

map.put("password", Collections.singletonList(loginDTO.getPassword()));

map.put("grant\_type", Collections.singletonList(oAuth2ProtectedResourceDetails.getGrantType()));

map.put("scope", oAuth2ProtectedResourceDetails.getScope());

//HttpEntity

HttpEntity httpEntity = new HttpEntity(map,httpHeaders);

//获取 Token

return restTemplate.exchange(oAuth2ProtectedResourceDetails.getAccessTokenUri(), HttpMethod.POST,httpEntity,OAuth2AccessToken.class);

}

}

}

启动类

@SpringBootApplication

@EnableResJWTTokenStore //OAuth2 使用 JWT 解析令牌

public class Resource1ServerApplication {

@Bean

public RestTemplate restTemplate(){

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(Resource1ServerApplication.class, args);

}

}

微服务配置

server:

port: 9007

security:

oauth2:

client:

clientId: resource1

clientSecret: secret

userAuthorizationUri: http://localhost:9005/oauth/authorize

grant-type: password

scope: read

access-token-uri: http://localhost:9005/oauth/token

resource:

jwt:

key-uri: http://localhost:9005/oauth/token\_key

basic:

enabled: false

单元测试

/\*\*

\* @author: 林塬

\* @date: 2018/1/19

\* @description: 资源获取单元测试

\*/

@Slf4j

@RunWith(SpringRunner.class)

@SpringBootTest(classes = Resource1ServerApplication.class)

@AutoConfigureMockMvc

public class ResControllerTest {

@Autowired

private MockMvc mockMvc;

private ObjectMapper objectMapper = new ObjectMapper();

private OAuth2AccessToken oAuth2AccessToken;

/\*\*

\* 获取令牌

\* @throws Exception

\*/

@Before

public void getToken() throws Exception {

LoginDTO loginDTO = new LoginDTO();

loginDTO.setUsername("linyuan");

loginDTO.setPassword("123456");

byte[] body = this.mockMvc.perform(

post("/login")

.content(objectMapper.writeValueAsBytes(loginDTO))

.contentType(MediaType.APPLICATION\_JSON) //请求数据的格式

.accept(MediaType.APPLICATION\_JSON) //接收返回数据的格式

).andExpect(status().isOk())

.andReturn().getResponse().getContentAsByteArray();

oAuth2AccessToken = objectMapper.readValue(body,OAuth2AccessToken.class);

}

/\*\*

\* 测试访问本地受保护资源

\* @throws Exception

\*/

@Test

public void testGetLocalRes() throws Exception{

int status = this.mockMvc.perform(

get("/res")

.header("Authorization",OAuth2AccessToken.BEARER\_TYPE+" "+oAuth2AccessToken.getValue())

.accept(MediaType.APPLICATION\_JSON)

).andDo(print()).andReturn().getResponse().getStatus();

printStatus(status);

Assert.assertEquals(status,HttpStatus.SC\_OK);

}

/\*\*

\* 测试访问资源服务器2受保护资源

\* @throws Exception

\*/

@Test

public void testGetRes2lRes() throws Exception{

int status = this.mockMvc.perform(

get("/res2/res")

.header("Authorization",OAuth2AccessToken.BEARER\_TYPE+" "+oAuth2AccessToken.getValue())

.accept(MediaType.APPLICATION\_JSON)

).andDo(print()).andReturn().getResponse().getStatus();

printStatus(status);

Assert.assertEquals(status,HttpStatus.SC\_OK);

}

private void printStatus(int status){

switch (status) {

case HttpStatus.SC\_OK:

log.info("测试访问受保护资源---------------->成功（200）");

break;

case HttpStatus.SC\_UNAUTHORIZED:

log.info("测试访问受保护资源---------------->失败（401---没有权限，请确认令牌无误，角色权限无误，请注意是否 Authorization 请求头部 Basic 打成了 Bearer）");

break;

case HttpStatus.SC\_BAD\_REQUEST:

log.info("测试访问受保护资源---------------->失败（400---请求失败，请检查用户密码是否正确）");

break;

default:

log.info("测试访问本地受保护资源---------------->失败（{}---未知结果）",status);

break;

}

}

}