**恒生电子股份有限公司**

**<（OAuth2.0）>**

技术方案说明

<终端发展部>

2013年5 月

**文档修改记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 修订人 | 修订说明 | 批准人 | 发布日期 |
| 1.0.1.0 | 蔡建龙 | 初稿 |  | 2013-05-07 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**说 明**

**本文档中所包含的信息属于商业机密信息，如无恒生电子股份有限公司的书面许可，任何人都无权复制或利用**

**模版版本信息**

**编辑部门：EPG**

**批准日期：2008/1/22**

目录

[1. 概述 1](#_Toc528569436)

[1.1. 术语和缩略语 1](#_Toc528569437)

[2. 环境准备 1](#_Toc528569438)

[2.1. 软件环境 1](#_Toc528569439)

[2.2. 添加Maven关键依赖包 1](#_Toc528569440)

[3. OAUTH2.0 2](#_Toc528569441)

[3.1. 表说明 2](#_Toc528569442)

[3.2. 关键业务类说明 5](#_Toc528569443)

[3.3. OAuth2.0源码目录说明 6](#_Toc528569444)

[4. DEMO目录结构说明 7](#_Toc528569445)

[5. OAuth底层实现机制与原理 8](#_Toc528569446)

[5.1. 认证流程 8](#_Toc528569447)

[5.2. 提供者实现原理 8](#_Toc528569448)

[1. 授权服务配置 9](#_Toc528569449)

[2. 资源服务配置 13](#_Toc528569450)

[6. OAuth DEMO具体代码实例 14](#_Toc528569451)

[6.1. 工程搭建说明 14](#_Toc528569452)

[3. 系统配置文件说明 14](#_Toc528569453)

[4. 日志配置文件说明 16](#_Toc528569454)

[6.2. DEMO核心配置类说明 23](#_Toc528569455)

[1. DB数据连接池配置 23](#_Toc528569456)

[2. 认证服务器校验配置 25](#_Toc528569457)

[3. 资源服务器认证配置 28](#_Toc528569458)

[4. 应用安全策略配置 28](#_Toc528569459)

[5. 服务验证策略配置 29](#_Toc528569460)

[6. 异常信息自定义配置 33](#_Toc528569461)

[7. 附录 36](#_Toc528569462)

# 概述

OAuth2是基于Spring Security 而建立的一套安全认证规范，并非独立于Spring之外的框架。

## 术语和缩略语

|  |  |
| --- | --- |
| **缩写、术语及符号** | **解 释** |
| THIRD-PARTY APPLICATION | 三方应用程序 |
| HTTP SERVICE | HTTP 服务提供商 |
| RESOURCE OWNER | 资源所有者 |
| USER AGENT | 用户代理 |
| AUTHORIZATION SERVER | 认证服务器 |
| RESOURCE SERVER | 资源服务器 |

# 环境准备

## 软件环境

1. JDK Version:1.6+
2. 支持的数据库有：H2、MYSQL、ORACLE、DB2
3. Intellij IDEA

## 添加Maven关键依赖包

<!-- SpringBoot OAuth2.0 认证 -->  
<dependency>  
 <groupId>org.springframework.security.oauth</groupId>  
 <artifactId>spring-security-oauth2</artifactId>  
</dependency>  
  
<!-- Druid数据连接池 -->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>${druid.version}</version>  
</dependency>

# OAUTH2.0

## 表说明

|  |  |
| --- | --- |
| **OAUTH\_CLIENT\_DETAILS 表** | |
| **字段名** | **字段说明** |
| CLIENT\_ID | 主键,必须唯一,不能为空.  用于唯一标识每一个客户端(client); 在注册时必须填写(也可由服务端自动生成). 对于不同的grant\_type,该字段都是必须的. 在实际应用中的另一个名称叫appKey,与client\_id是同一个概念. |
| RESOURCES\_IDS | 客户端所能访问的资源id集合,多个资源时用逗号(,)分隔,如: "unity-resource,mobile-resource".  该字段的值必须来源于与security.xml中标签‹oauth2:resource-server的属性resource-id值一致. 在security.xml配置有几个‹oauth2:resource-server标签, 则该字段可以使用几个该值.  在实际应用中, 我们一般将资源进行分类,并分别配置对应的‹oauth2:resource-server,如订单资源配置一个‹oauth2:resource-server, 用户资源又配置一个‹oauth2:resource-server. 当注册客户端时,根据实际需要可选择资源id,也可根据不同的注册流程,赋予对应的资源id. |
| CLIENT\_SECRET | 用于指定客户端(client)的访问密匙; 在注册时必须填写(也可由服务端自动生成). 对于不同的grant\_type,该字段都是必须的. 在实际应用中的另一个名称叫appSecret,与client\_secret是同一个概念. |
| SCOPE | 指定客户端申请的权限范围,可选值包括read,write,trust;若有多个权限范围用逗号(,)分隔,如: "read,write".  scope的值与security.xml中配置的‹intercept-url的access属性有关系. 如‹intercept-url的配置为: <intercept-url pattern="/m/\*\*" access="ROLE\_MOBILE,SCOPE\_READ"/>  则说明访问该URL时的客户端必须有read权限范围. write的配置值为SCOPE\_WRITE, trust的配置值为SCOPE\_TRUST.  在实际应该中, 该值一般由服务端指定, 常用的值为read,write. |
| AUTHORIZED\_GRANT\_TYPE | 指定客户端支持的grant\_type,可选值包括: authorization\_code,password,refresh\_token,implicit,client\_credentials,  若支持多个grant\_type用逗号(,)分隔,如: "authorization\_code,password".  在实际应用中,当注册时,该字段是一般由服务器端指定的,而不是由申请者去选择的,最常用的grant\_type组合有: "authorization\_code,refresh\_token"(针对通过浏览器访问的客户端); "password,refresh\_token"(针对移动设备的客户端).  implicit与client\_credential*s*在实际中很少使用. |
| WEB\_SERVER\_REDIRECT\_URI | 客户端的重定向URI,可为空, 当grant\_type为authorization\_code或implicit时, 在Oauth的流程中会使用并检查与注册时填写的redirect\_uri是否一致. 下面分别说明: 当grant\_type=authorization\_code时, 第一步 从 spring-oauth-server获取 'code'时客户端发起请求时必须有redirect\_uri参数, 该参数的值必须与web\_server\_redirect\_uri的值一致. 第二步 用 'code' 换取 'access\_token' 时客户也必须传递相同的redirect\_uri. 在实际应用中, web\_server\_redirect\_uri在注册时是必须填写的, 一般用来处理服务器返回的code, 验证state是否合法与通过code去换取access\_token值. 在spring-oauth-client项目中, 可具体参考AuthorizationCodeController.java中的authorizationCodeCallback方法. 当grant\_type=implicit时通过redirect\_uri的hash值来传递access\_token值.如: <http://localhost:7777/spring-oauth-client/implicit>?  access\_token=dc891f4a-ac88-4ba6-8224-a2497e013865&token\_type=bearer&expires\_in=43199 然后客户端通过JS等从hash值中取到access\_token值. |
| AUTHORITIES | 指定客户端所拥有的Spring Security的权限值,可选, 若有多个权限值,用逗号(,)分隔, 如: "ROLE\_UNITY,ROLE\_USER".  对于是否要设置该字段的值,要根据不同的grant\_type来判断, 若客户端在Oauth流程中需要用户的用户名(username)与密码(password)的(authorization\_code,password),  则该字段可以不需要设置值,因为服务端将根据用户在服务端所拥有的权限来判断是否有权限访问对应的API.  但如果客户端在Oauth流程中不需要用户信息的(implicit,client\_credentials),  则该字段必须要设置对应的权限值, 因为服务端将根据该字段值的权限来判断是否有权限访问对应的API.  (请在[spring-oauth-client](http://git.oschina.net/mkk/spring-oauth-client)项目中来测试不同grant\_type时authorities的变化) |
| ACCESS\_TOKEN\_VALIDITY | 设定客户端的access\_token的有效时间值(单位:秒),可选, 若不设定值则使用默认的有效时间值(60 \* 60 \* 12, 12小时).  在服务端获取的access\_token JSON数据中的expires\_in字段的值即为当前access\_token的有效时间值.  在项目中, 可具体参考 DefaultTokenServices.java中属性accessTokenValiditySeconds.  在实际应用中, 该值一般是由服务端处理的, 不需要客户端自定义. |
| REFRESH\_TOKEN\_VALIDITY | 设定客户端的refresh\_token的有效时间值(单位:秒),可选, 若不设定值则使用默认的有效时间值(60 \* 60 \* 24 \* 30, 30天).  若客户端的grant\_type不包括refresh\_token,则不用关心该字段 在项目中, 可具体参考DefaultTokenServices.java中属性refreshTokenValiditySeconds.  在实际应用中, 该值一般是由服务端处理的, 不需要客户端自定义. |
| ADDITIONAL\_INFORMATION | 这是一个预留的字段,在Oauth的流程中没有实际的使用,可选,但若设置值,必须是JSON格式的数据,如: {"country":"CN","country\_code":"086"} 按照spring-security-oauth项目中对该字段的描述  Additional information for this client, not need by the vanilla OAuth protocol but might be useful, for example,for storing descriptive information.  (详见ClientDetails.java的getAdditionalInformation()方法的注释)在实际应用中, 可以用该字段来存储关于客户端的一些其他信息,如客户端的国家,地区,注册时的IP地址等等. |
| CREATE\_TIME | 数据的创建时间,精确到秒,由数据库在插入数据时取当前系统时间自动生成(扩展字段) |
| ARCHIVED | 用于标识客户端是否已存档(即实现逻辑删除),默认值为'0'(即未存档).  对该字段的具体使用请参考CustomJdbcClientDetailsService.java,在该类中,扩展了在查询client\_details的SQL加上archived *=* 0条件 (扩展字段) |
| TRUSTED | 设置客户端是否为受信任的,默认为'0'(即不受信任的,1为受信任的).  该字段只适用于grant\_type="authorization\_code"的情况,当用户登录成功后,若该值为0,则会跳转到让用户Approve的页面让用户同意授权,  若该字段为1,则在登录后不需要再让用户Approve同意授权(因为是受信任的).  对该字段的具体使用请参考OauthUserApprovalHandler.java. (扩展字段) |
| AUTOAPPROVE | 设置用户是否自动Approval操作, 默认值为 'false', 可选值包括 'true','false', 'read','write'.  该字段只适用于grant\_type="authorization\_code"的情况,当用户登录成功后,若该值为'true'或支持的scope值,则会跳过用户Approve的页面, 直接授权.  该字段与 trusted 有类似的功能, 是 spring-security-oauth2 的 2.0 版本后添加的新属性. |
| **提醒：**在项目中,主要操作oauth\_client\_details表的类是JdbcClientDetailsService.java, 更多的细节请参考该类. 也可以根据实际的需要,去扩展或修改该类的实现. | |

|  |  |
| --- | --- |
| **OAUTH\_ACCESS\_TOKEN 表** | |
| **字段名** | **字段说明** |
| CREATE\_TIME | 数据的创建时间，精确到秒，由数据库在插入数据时取当前系统时间自动生产（扩展字段） |
| TOKEN\_ID | 该字段的值是将ACCESS\_TOKEN的值通过MD5加密后存储的 |
| TOKEN | 存储将OAuth2AccessToken.java对象序列化后的二进制数据，是真实的AccessToken的值。 |
| AUTHENTICATION\_ID | 该字段具有唯一性，其值是根据当前的username（如果有），client\_id与scope通过MD5加密生成的. 具体实现请参考DefaultAuthenticationKeyGenerator.java类. |
| USER\_NAME | 登录时的用户名，若客户端没有用户名（如grant\_type=’client\_credentials’）,则该值等于client\_id |
| CLIENT\_ID |  |
| AUTHENTICATION | 存储将OAuth2Authentication.java对象序列化后的二进制数据 |
| REFRESH\_TOKEN | 该字段的值是将REFRESH\_TOKEN的值通过MD5加密后存储的 |
| 提醒：在项目中,主要操作oauth\_access\_token表的对象是JdbcTokenStore.java. 更多的细节请参考该类 | |

|  |  |
| --- | --- |
| **OAUTH\_CLIENT\_TOKEN 表** | |
| **字段名** | **字段说明** |
| CREATE\_TIME | 数据的创建时间，精确到秒，由数据库在插入数据时取当前系统时间自动生产（扩展字段） |
| TOKEN\_ID | 从服务器获取的ACCESS\_TOKEN的值 |
| TOKEN | 这是一个二进制的字段，存储的数据是OAuth2AccessToken.java对象序列化后的二进制数据。 |
| AUTHENTICATION\_ID | 该字段具有唯一性，其值是根据当前的username（如果有），client\_id与scope通过MD5加密生成的. 具体实现请参考DefaultAuthenticationKeyGenerator.java类. |
| USER\_NAME | 登录时的用户名 |
| CLIENT\_ID |  |
| **提醒**：该表用于在客户端系统中存储从服务端获取的token数据, 在[spring-oauth-server](http://git.oschina.net/shengzhao/spring-oauth-server)项目中未使用到.  对oauth\_client\_token表的主要操作在JdbcClientTokenServices.java类中, 更多的细节请参考该类. | |

|  |  |
| --- | --- |
| **OAUTH\_REFRESH\_TOKEN 表** | |
| **字段名** | **字段说明** |
| CREATE\_TIME | 数据的创建时间，精确到秒，由数据库在插入数据时取当前系统时间自动生产（扩展字段） |
| TOKEN\_ID | 从服务器获取的ACCESS\_TOKEN的值 |
| TOKEN | 这是一个二进制的字段，存储的数据是OAuth2AccessToken.java对象序列化后的二进制数据。 |
| AUTHENTICATION | 存储将OAuth2Authentication.java对象序列化后的二进制数据 |
| **提醒**：在项目中,主要操作oauth\_refresh\_token表的对象是JdbcTokenStore.java. (与操作oauth\_access\_token表的对象一样);更多的细节请参考该类. 如果客户端的grant\_type不支持refresh\_token,则不会使用该表. | |
| **OAUTH\_CODE 表** | |
| **字段名** | **字段说明** |
| CREATE\_TIME | 数据的创建时间，精确到秒，由数据库在插入数据时取当前系统时间自动生产（扩展字段） |
| CODE | 存储服务器端存储的code值（未加密） |
| AUTHENTICATION | 存储将OAuth2Authentication.java对象序列化后的二进制数据 |
| **提醒**： 在项目中,主要操作oauth\_code表的对象是JdbcAuthorizationCodeServices.java. 更多的细节请参考该类. 只有当grant\_type为"authorization\_code"时,该表中才会有数据产生; 其他的grant\_type没有使用该表. | |

## 关键业务类说明

|  |  |
| --- | --- |
| **类名** | **说明** |
| JdbcClientDetailsService | 主要是关系着client信息的管理。以jdbc的方式进行读取 |
| JdbcClientTokenServices | 里面封装了对oauth\_client\_token表的一系列操作（如增删改查），此表保存的是此client的token信息。 |
| OAuth2RestTemplate | 实现了OAuthRestOperations接口，主要是用来去调用受保护资源，会自己带上当前的token信息。 |
| TokenEndpoint | 定义了核心接口如：/oauth/authorize、/oauth/token、/oauth/confirm\_access |
| ResourceServerConfigurerAdapter |  |
| OAuth2AuthenticationProcessingFilter | 负责当资源服务器接收到请求，此类将进行过滤校验等操作。 |

## OAuth2.0源码目录说明

|  |  |
| --- | --- |
| **包名** | **说明** |
| Client | 主要处理客户端认证，客户端TOKEN操作等。 |
| Common | 工具类层，主要对JSON字符串处理、解析和实例化操作。 |
| Config | 核心层，包括认证服务配置、资源服务配置等。 |
| http.converter |  |
| Provider | 最核心层，包含Endpoint、认证、以及四种授权方式的管理。 |
|  |  |

**注：**本表格指Spring-Cloud-Oauth2.0

# DEMO目录结构说明

**OAUTH2.0 DEMO：**

|  |  |
| --- | --- |
| SPRINGBOOT\_OAUTH | 应用工程根（可更名） |
| |-target | 编译集成后的输出目录 |
| |-src | 代码主目录 |
| |-main | 源代码根目录 |
| |-java | JAVA源码目录 |
| |-\*.base | BASE基础类 |
| |-\*.bean | 核心实体类 |
| |-\*.config | 配置类 |
| |-\*.constants | 枚举类 |
| |-\*.controller | 控制层 |
| |-\*.core | 核心层 |
| |-\*.exception | 自定义异常 |
| |-\*.mapper | Mybatis操作数据层 |
| |-\*.service | 服务层 |
| |-\*.util | 工具层 |
| Application.java | SpringBoot工程启动类 |
| |-resources | 核心资源文件目录 |
| |-mapper.\* | Mybatis配置文件 |
| |-application.yml | 系统配置文件 |
| |-logback.xml | 日志管理配置文件 |
| |-pom.xml | MVN依赖配置 |
| |-test | 测试主目录 |
| |-java | 测试片段代码目录 |

# OAuth底层实现机制与原理

## 认证流程

+--------+ +---------------+  
| |--(A)- Authorization Request -> | Resource |  
| | | Owner |  
| |<-(B)-- Authorization Grant --| |  
| | +---------------+  
| | +---------------+  
| |--(C)-- Authorization Grant --> | Authorization |  
| Client | | Server |  
| |<-(D)----- Access Token - ----- | |  
| | +---------------+  
| | +---------------+  
| |--(E)----- Access Token ------> | Resource |  
| | | Server |  
| |<-(F)--- Protected Resource - -- | |  
+--------+ +---------------+

## 提供者实现原理

Spring OAuth2.0提供者实际上分为：

* 授权服务 Authorization Service.
* 资源服务 Resource Service.

虽然这两个提供者有时候可能存在同一个应用程序中，但在Spring Security OAuth中你可以把

它们各自放在不同的应用上，而且你可以有多个资源服务，它们共享同一个中央授权服

务。

所有获取令牌的请求都将会在Spring MVC Controller Endpoints中进行处理，并且访问受保护

的资源服务的处理流程将会放在标准的Spring Security请求过滤器中(Filters)。

下面是配置一个授权服务必须要实现的Endpoints：

* AuthorizationEndpoint：用来作为请求者获得授权的服务，默认的URL是/oauth/authorize.
* TokenEndpoint：用来作为请求者获得令牌（Token）的服务，默认的URL是/oauth/token.

下面是配置一个资源服务必须要实现的过滤器：

* OAuth2AuthenticationProcessingFilter：用来作为认证令牌（Token）的一个处理流程过滤器。只有当过滤器通过之后，请求者才能获得受保护的资源。

配置提供者（授权、资源）都可以通过简单的Java注解@Configuration来进行适配，你也可以使用基于XML的声明式语法来进行配置，如果你打算这样做的话，那么请使用http://www.springframework.org/schema/security/spring-security-oauth2.xsd来作为XML的schema（即XML概要定义）以及使用http://www.springframework.org/schema/security/oauth2来作为命名空间。

### 授权服务配置

配置一个授权服务，你需要考虑几种授权类型（Grant Type），不同的授权类型为客户端（Client）提供了不同的获取令牌（Token）方式，为了实现并确定这几种授权，需要配置使用 ClientDetailsService 和 TokenService 来开启或者禁用这几种授权机制。到这里就请注意了，不管你使用什么样的授权类型（Grant Type），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。这也就是说，假如你提供了一个支持"client\_credentials"的授权方式，并不意味着客户端就需要使用这种方式来获得授权。下面是几种授权类型的列表，具体授权机制的含义可以参见RFC6749([中文版本](https://github.com/jeansfish/RFC6749.zh-cn))：

authorization\_code：授权码类型。

implicit：隐式授权类型。

password：资源所有者（即用户）密码类型。

client\_credentials：客户端凭据（客户端ID以及Key）类型。

refresh\_token：通过以上授权获得的刷新令牌来获取新的令牌。

可以用 @EnableAuthorizationServer 注解来配置OAuth2.0 授权服务机制，通过使用@Bean注解的几个方法一起来配置这个授权服务。下面咱们介绍几个配置类，这几个配置是由Spring创建的独立的配置对象，它们会被Spring传入AuthorizationServerConfigurer中：

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，你能够把客户端详情信息写死在这里或者是通过数据库来存储调取详情信息。

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

AuthorizationServerEndpointsConfigurer：用来配置授权（authorization）以及令牌（token）的访问端点和令牌服务(token services)。

（译者注：以上的配置可以选择继承AuthorizationServerConfigurerAdapter并且覆写其中的三个configure方法来进行配置。）

配置授权服务一个比较重要的方面就是提供一个授权码给一个OAuth客户端（通过 authorization\_code 授权类型），一个授权码的获取是OAuth客户端跳转到一个授权页面，然后通过验证授权之后服务器重定向到OAuth客户端，并且在重定向连接中附带返回一个授权码。

如果你是通过XML来进行配置的话，那么可以使用 <authorization-server/> 标签来进行配置。

（译者注：想想现在国内各大平台的社会化登陆服务，例如腾讯，用户要使用QQ登录到某个网站，这个网站是跳转到了腾讯的登陆授权页面，然后用户登录并且确定授权之后跳转回目标网站，这种授权方式规范在我上面提供的链接\*RFC6749\*的第4.1节有详细阐述。）

**配置客户端详情信息（Client Details)：**

ClientDetailsServiceConfigurer (AuthorizationServerConfigurer 的一个回调配置项，见上的概述) 能够使用内存或者JDBC来实现客户端详情服务（ClientDetailsService），有几个重要的属性如下列表：

clientId：（必须的）用来标识客户的Id。

secret：（需要值得信任的客户端）客户端安全码，如果有的话。

scope：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。

authorizedGrantTypes：此客户端可以使用的授权类型，默认为空。

authorities：此客户端可以使用的权限（基于Spring Security authorities）。

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 ClientDetailsManager 接口（同时你也可以实现 ClientDetailsService 接口）来进行管理。

（译者注：不过我并没有找到 ClientDetailsManager 这个接口文件，只找到了 ClientDetailsService）

**管理令牌（Managing Token）：**

AuthorizationServerTokenServices 接口定义了一些操作使得你可以对令牌进行一些必要的管理，在使用这些操作的时候请注意以下几点：

当一个令牌被创建了，你必须对其进行保存，这样当一个客户端使用这个令牌对资源服务进行请求的时候才能够引用这个令牌。

当一个令牌是有效的时候，它可以被用来加载身份信息，里面包含了这个令牌的相关权限。

当你自己创建 AuthorizationServerTokenServices 这个接口的实现时，你可能需要考虑一下使用 DefaultTokenServices 这个类，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。默认的，当它尝试创建一个令牌的时候，是使用随机值来进行填充的，除了持久化令牌是委托一个 TokenStore 接口来实现以外，这个类几乎帮你做了所有的事情。并且 TokenStore 这个接口有一个默认的实现，它就是 InMemoryTokenStore ，如其命名，所有的令牌是被保存在了内存中。除了使用这个类以外，你还可以使用一些其他的预定义实现，下面有几个版本，它们都实现了TokenStore接口：

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，你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，所以更易于调试。

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请注意把"spring-jdbc"这个依赖加入到你的classpath当中。

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。

**JWT令牌（JWT Tokens）：**

使用JWT令牌你需要在授权服务中使用 JwtTokenStore，资源服务器也需要一个解码的Token令牌的类 JwtAccessTokenConverter，JwtTokenStore依赖这个类来进行编码以及解码，因此你的授权服务以及资源服务都需要使用这个转换类。Token令牌默认是有签名的，并且资源服务需要验证这个签名，因此呢，你需要使用一个对称的Key值，用来参与签名计算，这个Key值存在于授权服务以及资源服务之中。或者你可以使用非对称加密算法来对Token进行签名，Public Key公布在/oauth/token\_key这个URL连接中，默认的访问安全规则是"denyAll()"，即在默认的情况下它是关闭的，你可以注入一个标准的 SpEL 表达式到 AuthorizationServerSecurityConfigurer 这个配置中来将它开启（例如使用"permitAll()"来开启可能比较合适，因为它是一个公共密钥）。

如果你要使用 JwtTokenStore，请务必把"spring-security-jwt"这个依赖加入到你的classpath中。

**配置授权类型（Grant Types）：**

授权是使用 AuthorizationEndpoint 这个端点来进行控制的，你能够使用 AuthorizationServerEndpointsConfigurer 这个对象的实例来进行配置(AuthorizationServerConfigurer 的一个回调配置项，见上的概述) ，如果你不进行设置的话，默认是除了资源所有者密码（password）授权类型以外，支持其余所有标准授权类型的（RFC6749），我们来看一下这个配置对象有哪些属性可以设置吧，如下列表：

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：如果啊，你设置了这个属性的话，那说明你有一个自己的 UserDetailsService 接口的实现，或者你可以把这个东西设置到全局域上面去（例如 GlobalAuthenticationManagerConfigurer 这个配置对象），当你设置了这个之后，那么 "refresh\_token" 即刷新令牌授权类型模式的流程中就会包含一个检查，用来确保这个账号是否仍然有效，假如说你禁用了这个账户的话。

authorizationCodeServices：这个属性是用来设置授权码服务的（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：这个属性就很牛B了，当你设置了这个东西（即 TokenGranter 接口实现），那么授权将会交由你来完全掌控，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的，即标准的四种授权模式已经满足不了你的需求的时候，才会考虑使用这个。

在XML配置中呢，你可以使用 "authorization-server" 这个标签元素来进行设置。

**配置授权端点的URL（Endpoint URLs）：**

AuthorizationServerEndpointsConfigurer 这个配置对象(AuthorizationServerConfigurer 的一个回调配置项，见上的概述) 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。

以上的参数都将以 "/" 字符为开始的字符串，框架的默认URL链接如下列表，可以作为这个 pathMapping() 方法的第一个参数：

/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。

需要注意的是授权端点这个URL应该被Spring Security保护起来只供授权用户访问，我们来看看在标准的Spring Security中 WebSecurityConfigurer 是怎么用的。

@Override

protected void configure(HttpSecurity http) throws Exception {

http .authorizeRequests().antMatchers("/login").permitAll().and()

// default protection for all resources (including /oauth/authorize)

.authorizeRequests() .anyRequest().hasRole("USER")

// ... more configuration, e.g. for form login

}

注意：如果你的应用程序中既包含授权服务又包含资源服务的话，那么这里实际上是另一个的低优先级的过滤器来控制资源接口的，这些接口是被保护在了一个访问令牌（access token）中，所以请挑选一个URL链接来确保你的资源接口中有一个不需要被保护的链接用来取得授权，就如上面示例中的 /login 链接，你需要在 WebSecurityConfigurer 配置对象中进行设置。

令牌端点默认也是受保护的，不过这里使用的是基于 HTTP Basic Authentication 标准的验证方式来验证客户端的，这在XML配置中是无法进行设置的（所以它应该被明确的保护）。

在XML配置中可以使用 <authorization-server/> 元素标签来改变默认的端点URLs，注意在配置 /check\_token 这个链接端点的时候，使用 check-token-enabled 属性标记启用。

**强制使用SSL（Enforcing SSL）：**

使用简单的HTTP请求来进行测试是可以的，但是如果你要部署到产品环境上的时候，你应该永远都使用SSL来保护授权服务器在与客户端进行通讯的时候进行加密。你可以把授权服务应用程序放到一个安全的运行容器中，或者你可以使用一个代理，如果你设置正确了的话它们应该工作的很好（这样的话你就不需要设置任何东西了）。

但是也许你可能希望使用 Spring Security 的 requiresChannel() 约束来保证安全，对于授权端点来说（还记得上面的列表吗，就是那个 /authorize 端点），它应该成为应用程序安全连接的一部分，而对于 /token 令牌端点来说的话，它应该有一个标记被配置在 AuthorizationServerEndpointsConfigurer 配置对象中，你可以使用 sslOnly() 方法来进行设置。当然了，这两个设置是可选的，不过在以上两种情况中，会导致Spring Security 会把不安全的请求通道重定向到一个安全通道中。（译者注：即将HTTP请求重定向到HTTPS请求上）。

**自定义错误处理（Error Handling）：**

端点实际上就是一个特殊的Controller，它用于返回一些对象数据。

授权服务的错误信息是使用标准的Spring MVC来进行处理的，也就是 @ExceptionHandler 注解的端点方法，你也可以提供一个 WebResponseExceptionTranslator 对象。最好的方式是改变响应的内容而不是直接进行渲染。

假如说在呈现令牌端点的时候发生了异常，那么异常委托了 HttpMessageConverters 对象（它能够被添加到MVC配置中）来进行输出。假如说在呈现授权端点的时候未通过验证，则会被重定向到 /oauth/error 即错误信息端点中。whitelabel error （即Spring框架提供的一个默认错误页面）错误端点提供了HTML的响应，但是你大概可能需要实现一个自定义错误页面（例如只是简单的增加一个 @Controller 映射到请求路径上 @RequestMapping("/oauth/error")）。

**映射用户角色到权限范围（Mapping User Roles to Scopes）：**

有时候限制令牌的权限范围是很有用的，这不仅仅是针对于客户端，你还可以根据用户的权限来进行限制。如果你使用 DefaultOAuth2RequestFactory 来配置 AuthorizationEndpoint 的话你可以设置一个flag即 checkUserScopes=true来限制权限范围，不过这只能匹配到用户的角色。你也可以注入一个 OAuth2RequestFactory 到 TokenEnpoint 中，不过这只能工作在 password 授权模式下。如果你安装一个 TokenEndpointAuthenticationFilter 的话，你只需要增加一个过滤器到 HTTP BasicAuthenticationFilter 后面即可。当然了，你也可以实现你自己的权限规则到 scopes 范围的映射和安装一个你自己版本的 OAuth2RequestFactory。AuthorizationServerEndpointConfigurer 配置对象允许你注入一个你自定义的 OAuth2RequestFactory，因此你可以使用这个特性来设置这个工厂对象，前提是你使用 @EnableAuthorizationServer 注解来进行配置（见上面介绍的授权服务配置）。

### 资源服务配置

一个资源服务（可以和授权服务在同一个应用中，当然也可以分离开成为两个不同的应用程序）提供一些受token令牌保护的资源，Spring OAuth提供者是通过Spring Security authentication filter 即验证过滤器来实现的保护，你可以通过 @EnableResourceServer 注解到一个 @Configuration 配置类上，并且必须使用 ResourceServerConfigurer 这个配置对象来进行配置（可以选择继承自 ResourceServerConfigurerAdapter 然后覆写其中的方法，参数就是这个对象的实例），下面是一些可以配置的属性：

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

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。

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。

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。

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

在XML配置中，使用 <resource-server />标签元素并指定id为一个servlet过滤器就能够手动增加一个标准的过滤器链。

ResourceServerTokenServices 是组成授权服务的另一半，如果你的授权服务和资源服务在同一个应用程序上的话，你可以使用 DefaultTokenServices ，这样的话，你就不用考虑关于实现所有必要的接口的一致性问题，这通常是很困难的。如果你的资源服务器是分离开的，那么你就必须要确保能够有匹配授权服务提供的 ResourceServerTokenServices，它知道如何对令牌进行解码。

在授权服务器上，你通常可以使用 DefaultTokenServices 并且选择一些主要的表达式通过 TokenStore（后端存储或者本地编码）。

RemoteTokenServices 可以作为一个替代，它将允许资源服务器通过HTTP请求来解码令牌（也就是授权服务的 /oauth/check\_token 端点）。如果你的资源服务没有太大的访问量的话，那么使用RemoteTokenServices 将会很方便（所有受保护的资源请求都将请求一次授权服务用以检验token值），或者你可以通过缓存来保存每一个token验证的结果。

使用授权服务的 /oauth/check\_token 端点你需要将这个端点暴露出去，以便资源服务可以进行访问，这在咱们授权服务配置中已经提到了，下面是一个例子：

@Override

public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {

oauthServer.tokenKeyAccess("isAnonymous() || hasAuthority('ROLE\_TRUSTED\_CLIENT')")  
 .checkTokenAccess("hasAuthority('ROLE\_TRUSTED\_CLIENT')");

}

在这个例子中，我们配置了 /oauth/check\_token 和 /oauth/token\_key 这两个端点（受信任的资源服务能够获取到公有密匙，这是为了验证JWT令牌）。这两个端点使用了HTTP Basic Authentication 即HTTP基本身份验证，使用 client\_credentials 授权模式可以做到这一点。

**配置OAuth-Aware表达式处理器（OAuth-Aware Expression Handler）：**

你也许希望使用 Spring Security's expression-based access control 来获得一些优势，一个表达式处理器会被注册到默认的 @EnableResourceServer 配置中，这个表达式包含了 #oauth2.clientHasRole，#oauth2.clientHasAnyRole 以及 #oauth2.denyClient 所提供的方法来帮助你使用权限角色相关的功能（在 OAuth2SecurityExpressionMethods 中有完整的列表）。

在XML配置中你可以注册一个 OAuth-Aware 表达式处理器即 <expression-handler />元素标签到 常规的 <http /> 安全配置上。

# OAuth DEMO具体代码实例

## 工程搭建说明

### 系统配置文件说明

#SPRING 系统配置  
spring:  
 application:  
 name: OAUTH DEMO  
 #数据源参数配置  
 datasource:  
 username: root  
 password: hsfund  
 driver-class-name: com.mysql.jdbc.Driver  
 url: jdbc:mysql://127.0.0.1:3306/activiti?characterEncoding=utf-8  
 initial-size: 5  
 max-active: 20  
 max-idle: 8  
 min-idle: 3  
 max-wait: 500  
 #是否自动回收超时连接  
 remove-abandoned: true  
 remove-abandoned-timeout: 180  
 #有些数据库连接的时候有超时限制（mysql连接在8小时后断开）  
 #或者由于网络中断等原因出现链接失效的情况  
 #设置一个testWhileIdle参数为true，可以保证连接池内部定时检测连接的可用性  
 #不可用的连接会被抛弃或者重建，最大情况的保证从连接池中得到的Connection对象是可用的。  
 test-while-idle: true  
 ##数据库连接有效性验证  
 validation-query: SELECT 1  
 validation-query-timeout: 1  
 ##每隔五分钟检测空闲超过1分钟的连接,并物理释放  
 min-evictable-idle-time-millis: 60000  
 ##链接在连接池中最小生存时间  
 time-between-eviction-runs-millis: 300000  
 ## Validate the connection before borrowing it from the pool.  
 test-on-borrow: true  
 num-tests-per-eviction-run: 20  
 #数据源JPA配置  
 jpa:  
 show-sql: true  
 hibernate:  
 ddl-auto: update  
 properties:  
 hibernate:  
 dialect: org.hibernate.dialect.MySQL5Dialect  
 format\_sql: true  
 use\_sql\_comments: true  
 activiti:  
 ##关闭自动部署 process下流程  
 check-process-definitions: false  
#OAUTH认证配置（如果代码中选择数据库认证之后，则此配置会被默认覆盖）  
security:  
 user:  
 name: admin  
 password: 123456  
 logging:  
 level: DEBUG  
#系统信息配置  
system:  
 #异常配置项  
 exception:  
 default-errorCode: 10000  
 default-error-info: 系统罢工了,请等会再来  
 default-status: 500  
 ######扫描包路径,出现异常时可提取此包下的堆栈信息,方便开发人员定位具体错误代码行  
 basePackages: com.hs.oauth.tools  
 ######异常错误号前缀配置,分为数据绑定,业务,系统 3类异常#######  
 bind-exception-prefix: BIND-  
 biz-exception-prefix: BIZ-  
 system-exception-prefix: SYS-  
  
#MYBATIS 配置文件路径扫描配置   
mybatis:  
 mapperLocations: classpath:mapper/\*.xml  
  
#应用服务配置项  
server:  
 port: 8080

### 日志配置文件说明

<?xml version="1.0" encoding="UTF-8"?>  
<configuration>  
<!--  
 说明：  
 1、日志级别及文件  
 日志记录采用分级记录，级别与日志文件名相对应，不同级别的日志信息记录到不同的日志文件中  
 例如：error级别记录到log\_error\_xxx.log或log\_error.log（该文件为当前记录的日志文件），而log\_error\_xxx.log为归档日志，  
 日志文件按日期记录，同一天内，若日志文件大小等于或大于2M，则按0、1、2...顺序分别命名  
 例如log-level-2013-12-21.0.log  
 其它级别的日志也是如此。  
 2、文件路径  
 若开发、测试用，在Eclipse中运行项目，则到Eclipse的安装路径查找logs文件夹，以相对路径../logs。  
 若部署到Tomcat下，则在Tomcat下的logs文件中  
 3、Appender  
 FILEERROR对应error级别，文件名以log-error-xxx.log形式命名  
 FILEWARN对应warn级别，文件名以log-warn-xxx.log形式命名  
 FILEINFO对应info级别，文件名以log-info-xxx.log形式命名  
 FILEDEBUG对应debug级别，文件名以log-debug-xxx.log形式命名  
 stdout将日志信息输出到控制上，为方便开发测试使用  
-->  
  
<!-- 彩色日志依赖的渲染类 -->  
<conversionRule conversionWord="clr" converterClass="org.springframework.boot.logging.logback.ColorConverter" />  
<conversionRule conversionWord="wex" converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />  
<conversionRule conversionWord="wEx" converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter" />  
<!-- 彩色日志格式 -->  
<property name="CONSOLE\_LOG\_PATTERN" value="${CONSOLE\_LOG\_PATTERN:-%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%logger){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}" />  
<!--包名输出缩进对齐-->  
<property name="CONSOLE\_LOG\_PATTERN" value="${CONSOLE\_LOG\_PATTERN:-%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}" />  
  
  
<contextName>xcloud-print</contextName>  
<property name="LOG\_PATH" value="d://xcloud-print-logs" />  
<!--设置系统日志目录-->  
<property name="APPDIR" value="logs" />  
  
<!-- 日志记录器，日期滚动记录 -->  
<appender name="FILEERROR" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- 正在记录的日志文件的路径及文件名 -->  
 <file>${LOG\_PATH}/${APPDIR}/xcloud-print\_log\_error.log</file>  
 <!-- 日志记录器的滚动策略，按日期，按大小记录 -->  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!-- 归档的日志文件的路径，例如今天是2013-12-21日志，当前写的日志文件路径为file节点指定，可以将此文件与file指定文件路径设置为不同路径，从而将当前日志文件或归档日志文件置不同的目录。  
 而2013-12-21的日志文件在由fileNamePattern指定。%d{yyyy-MM-dd}指定日期格式，%i指定索引 -->  
 <fileNamePattern>${LOG\_PATH}/${APPDIR}/error/xcloud-print\_log-error-%d{yyyy-MM-dd}.%i.log</fileNamePattern>  
 <!-- 除按日志记录之外，还配置了日志文件不能超过2M，若超过2M，日志文件会以索引0开始，  
 命名日志文件，例如log-error-2013-12-21.0.log -->  
 <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">  
 <maxFileSize>12MB</maxFileSize>  
 </timeBasedFileNamingAndTriggeringPolicy>  
 </rollingPolicy>  
 <!-- 追加方式记录日志 -->  
 <append>true</append>  
 <!-- 日志文件的格式 -->  
 <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level --- [%thread] %logger Line:%-3L - %msg%n</pattern>  
 <charset>utf-8</charset>  
 </encoder>  
 <!-- 此日志文件只记录error级别的 -->  
 <filter class="ch.qos.logback.classic.filter.LevelFilter">  
 <level>error</level>  
 <onMatch>ACCEPT</onMatch>  
 <onMismatch>DENY</onMismatch>  
 </filter>  
</appender>  
  
<!-- 日志记录器，日期滚动记录 -->  
<appender name="FILEWARN" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- 正在记录的日志文件的路径及文件名 -->  
 <file>${LOG\_PATH}/${APPDIR}/xcloud-print\_log\_warn.log</file>  
 <!-- 日志记录器的滚动策略，按日期，按大小记录 -->  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!-- 归档的日志文件的路径，例如今天是2013-12-21日志，当前写的日志文件路径为file节点指定，可以将此文件与file指定文件路径设置为不同路径，从而将当前日志文件或归档日志文件置不同的目录。  
 而2013-12-21的日志文件在由fileNamePattern指定。%d{yyyy-MM-dd}指定日期格式，%i指定索引 -->  
 <fileNamePattern>${LOG\_PATH}/${APPDIR}/warn/xcloud-print\_log-warn-%d{yyyy-MM-dd}.%i.log</fileNamePattern>  
 <!-- 除按日志记录之外，还配置了日志文件不能超过2M，若超过2M，日志文件会以索引0开始，  
 命名日志文件，例如log-error-2013-12-21.0.log -->  
 <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">  
 <maxFileSize>12MB</maxFileSize>  
 </timeBasedFileNamingAndTriggeringPolicy>  
 </rollingPolicy>  
 <!-- 追加方式记录日志 -->  
 <append>true</append>  
 <!-- 日志文件的格式 -->  
 <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level --- [%thread] %logger Line:%-3L - %msg%n</pattern>  
 <charset>utf-8</charset>  
 </encoder>  
 <!-- 此日志文件只记录warn级别的 -->  
 <filter class="ch.qos.logback.classic.filter.LevelFilter">  
 <level>warn</level>  
 <onMatch>ACCEPT</onMatch>  
 <onMismatch>DENY</onMismatch>  
 </filter>  
</appender>  
  
<!-- 日志记录器，日期滚动记录 -->  
<appender name="FILEINFO" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- 正在记录的日志文件的路径及文件名 -->  
 <file>${LOG\_PATH}/${APPDIR}/xcloud-print\_log\_info.log</file>  
 <!-- 日志记录器的滚动策略，按日期，按大小记录 -->  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!-- 归档的日志文件的路径，例如今天是2013-12-21日志，当前写的日志文件路径为file节点指定，可以将此文件与file指定文件路径设置为不同路径，从而将当前日志文件或归档日志文件置不同的目录。  
 而2013-12-21的日志文件在由fileNamePattern指定。%d{yyyy-MM-dd}指定日期格式，%i指定索引 -->  
 <fileNamePattern>${LOG\_PATH}/${APPDIR}/info/xcloud-print\_log-info-%d{yyyy-MM-dd}.%i.log</fileNamePattern>  
 <!-- 除按日志记录之外，还配置了日志文件不能超过2M，若超过2M，日志文件会以索引0开始，  
 命名日志文件，例如log-error-2013-12-21.0.log -->  
 <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">  
 <maxFileSize>12MB</maxFileSize>  
 </timeBasedFileNamingAndTriggeringPolicy>  
 </rollingPolicy>  
 <!-- 追加方式记录日志 -->  
 <append>true</append>  
 <!-- 日志文件的格式 -->  
 <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level --- [%thread] %logger Line:%-3L - %msg%n</pattern>  
 <charset>utf-8</charset>  
 </encoder>  
 <!-- 此日志文件只记录info级别的 -->  
 <filter class="ch.qos.logback.classic.filter.LevelFilter">  
 <level>info</level>  
 <onMatch>ACCEPT</onMatch>  
 <onMismatch>DENY</onMismatch>  
 </filter>  
</appender>  
  
<!-- 日志记录器，日期滚动记录 -->  
<appender name="FILEDEBUG" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- 正在记录的日志文件的路径及文件名 -->  
 <file>${LOG\_PATH}/${APPDIR}/xcloud-print\_log\_debug.log</file>  
 <!-- 日志记录器的滚动策略，按日期，按大小记录 -->  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!-- 归档的日志文件的路径，例如今天是2013-12-21日志，当前写的日志文件路径为file节点指定，可以将此文件与file指定文件路径设置为不同路径，从而将当前日志文件或归档日志文件置不同的目录。  
 而2013-12-21的日志文件在由fileNamePattern指定。%d{yyyy-MM-dd}指定日期格式，%i指定索引 -->  
 <fileNamePattern>${LOG\_PATH}/${APPDIR}/info/xcloud-print\_log-debug-%d{yyyy-MM-dd}.%i.log</fileNamePattern>  
 <!-- 除按日志记录之外，还配置了日志文件不能超过2M，若超过2M，日志文件会以索引0开始，  
 命名日志文件，例如log-error-2013-12-21.0.log -->  
 <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">  
 <maxFileSize>12MB</maxFileSize>  
 </timeBasedFileNamingAndTriggeringPolicy>  
 </rollingPolicy>  
 <!-- 追加方式记录日志 -->  
 <append>true</append>  
 <!-- 日志文件的格式 -->  
 <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level --- [%thread] %logger Line:%-3L - %msg%n</pattern>  
 <charset>utf-8</charset>  
 </encoder>  
 <!-- 此日志文件只记录debug级别的 -->  
 <filter class="ch.qos.logback.classic.filter.LevelFilter">  
 <level>debug</level>  
 <onMatch>ACCEPT</onMatch>  
 <onMismatch>DENY</onMismatch>  
 </filter>  
</appender>  
  
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <!--encoder 默认配置为PatternLayoutEncoder-->  
 <encoder>  
 <pattern>${CONSOLE\_LOG\_PATTERN}</pattern>  
 <!--<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level -&#45;&#45; [%thread] %logger Line:%-3L - %msg%n</pattern>-->  
 <charset>utf-8</charset>  
 </encoder>  
 <!--此日志appender是为开发使用，只配置最底级别，控制台输出的日志级别是大于或等于此级别的日志信息-->  
 <filter class="ch.qos.logback.classic.filter.ThresholdFilter">  
 <level>debug</level>  
 </filter>  
</appender>  
  
<logger name="org.springframework" level="WARN" />  
  
<!-- mybatis日志打印 -->  
<logger name="com.ibatis" level="DEBUG" />  
<logger name="com.ibatis.common.jdbc.SimpleDataSource" level="DEBUG" />  
<logger name="com.ibatis.common.jdbc.ScriptRunner" level="DEBUG" />  
<logger name="com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate" level="DEBUG" />  
  
<logger name="java.sql.Connection" level="DEBUG" />  
<logger name="java.sql.Statement" level="DEBUG" />  
<logger name="java.sql.PreparedStatement" level="DEBUG" />  
<logger name="net.hs.activiti.demo.domain" level="DEBUG"></logger>  
  
<!-- 生产环境下，将此级别配置为适合的级别，以免日志文件太多或影响程序性能 -->  
<root level="DEBUG">  
 <appender-ref ref="FILEERROR" />  
 <appender-ref ref="FILEWARN" />  
 <appender-ref ref="FILEINFO" />  
 <appender-ref ref="FILEDEBUG" />  
  
 <!-- 生产环境将请stdout,testfile去掉 -->  
 <appender-ref ref="STDOUT" />  
</root>  
  
<!--日志异步到数据库 -->  
<!--<appender name="DB" class="ch.qos.logback.classic.db.DBAppender">-->  
<!--&lt;!&ndash;日志异步到数据库 &ndash;&gt;-->  
<!--<connectionSource class="ch.qos.logback.core.db.DriverManagerConnectionSource">-->  
<!--&lt;!&ndash;连接池 &ndash;&gt;-->  
<!--<dataSource class="com.mchange.v2.c3p0.ComboPooledDataSource">-->  
<!--<driverClass>com.mysql.jdbc.Driver</driverClass>-->  
<!--<url>jdbc:mysql://localhost:3306/databaseName</url>-->  
<!--<user>root</user>-->  
<!--<password>root</password>-->  
<!--</dataSource>-->  
<!--</connectionSource>-->  
<!--</appender>-->  
  
  
</configuration>

## DEMO核心配置类说明

### DB数据连接池配置

注：此类主要将数据库连接的配置注入到数据库连接池中。

@Configuration  
public class DruidDataSourceConfig {  
  
 @Value("${spring.datasource.url:#{null}}")  
 private String dbUrl;  
 @Value("${spring.datasource.username: #{null}}")  
 private String username;  
 @Value("${spring.datasource.password:#{null}}")  
 private String password;  
 @Value("${spring.datasource.driver-class-name:#{null}}")  
 private String driverClassName;  
 @Value("${spring.datasource.initialSize:#{null}}")  
 private Integer initialSize;  
 @Value("${spring.datasource.minIdle:#{null}}")  
 private Integer minIdle;  
 @Value("${spring.datasource.maxActive:#{null}}")  
 private Integer maxActive;  
 @Value("${spring.datasource.maxWait:#{null}}")  
 private Integer maxWait;  
 @Value("${spring.datasource.removeAbandoned:#{null}}")  
 private Boolean removeAbandoned;  
 @Value("${spring.datasource.removeAbandonedTimeout:#{null}}")  
 private Integer removeAbandonedTimeout;  
 @Value("${spring.datasource.numTestsPerEvictionRun:#{null}}")  
 private Integer numTestsPerEvictionRun;  
 @Value("${spring.datasource.timeBetweenEvictionRunsMillis:#{null}}")  
 private Integer timeBetweenEvictionRunsMillis;  
 @Value("${spring.datasource.minEvictableIdleTimeMillis:#{null}}")  
 private Integer minEvictableIdleTimeMillis;  
 @Value("${spring.datasource.validationQuery:#{null}}")  
 private String validationQuery;  
 @Value("${spring.datasource.validationQueryTimeout:#{null}}")  
 private Integer validationQueryTimeout;  
 @Value("${spring.datasource.testWhileIdle:#{null}}")  
 private Boolean testWhileIdle;  
 @Value("${spring.datasource.testOnBorrow:#{null}}")  
 private Boolean testOnBorrow;  
 @Value("${spring.datasource.testOnReturn:#{null}}")  
 private Boolean testOnReturn;  
 @Value("${spring.datasource.poolPreparedStatements:#{null}}")  
 private Boolean poolPreparedStatements;  
 @Value("${spring.datasource.maxPoolPreparedStatementPerConnectionSize:#{null}}")  
 private Integer maxPoolPreparedStatementPerConnectionSize;  
  
 @Bean  
 @Primary  
 public DataSource dataSource(){  
 DruidDataSource datasource = new DruidDataSource();  
  
 datasource.setUrl(this.dbUrl);  
 datasource.setUsername(username);  
 datasource.setPassword(password);  
 datasource.setDriverClassName(driverClassName);  
  
 if(initialSize != null) {  
 datasource.setInitialSize(initialSize);  
 }  
 if(minIdle != null) {  
 datasource.setMinIdle(minIdle);  
 }  
 if(maxActive != null) {  
 datasource.setMaxActive(maxActive);  
 }  
 if(maxWait != null) {  
 datasource.setMaxWait(maxWait);  
 }  
 if(removeAbandoned != null){  
 datasource.setRemoveAbandoned(removeAbandoned);  
 }  
 if(removeAbandonedTimeout != null){  
 datasource.setRemoveAbandonedTimeout(removeAbandonedTimeout);  
 }  
 if(timeBetweenEvictionRunsMillis != null) {  
 datasource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);  
 }  
 if(minEvictableIdleTimeMillis != null) {  
 datasource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);  
 }  
 if(numTestsPerEvictionRun != null){  
 datasource.setNumTestsPerEvictionRun(numTestsPerEvictionRun);  
 }  
 if(validationQuery != null) {  
 datasource.setValidationQuery(validationQuery);  
 }  
 if(validationQueryTimeout != null){  
 datasource.setValidationQueryTimeout(validationQueryTimeout);  
 }  
 if(testWhileIdle != null) {  
 datasource.setTestWhileIdle(testWhileIdle);  
 }  
 if(testOnBorrow != null) {  
 datasource.setTestOnBorrow(testOnBorrow);  
 }  
 if(testOnReturn != null) {  
 datasource.setTestOnReturn(testOnReturn);  
 }  
 if(poolPreparedStatements != null) {  
 datasource.setPoolPreparedStatements(poolPreparedStatements);  
 }  
 if(maxPoolPreparedStatementPerConnectionSize != null) {  
 datasource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);  
 }  
 return datasource;  
 }  
  
}

### 认证服务器校验配置

CustomOAuth2Config类主要对Token的生成方式，校验规则依赖等进行了注入和声明。

@Configuration  
@EnableAuthorizationServer  
public class CustomOAuth2Config extends AuthorizationServerConfigurerAdapter {  
  
 @Autowired  
 private DataSource dataSource;  
  
 /\*\*  
 \* @info TokenStore 默认有三种实现方式  
 \* 1.InMemoryTokenStore Token存储在内存中  
 \* 2.JdbcTokenStore Token存储在数据库中  
 \* 3.JwtTokenStore 采用JWT认证机制  
 \* @return  
 \*/  
 @Bean  
 public TokenStore tokenStore() {  
 //使用数据库中的TOKEN  
 return new JdbcTokenStore(dataSource);  
 }  
  
 @Bean  
 public ClientDetailsService clientDetails(){  
 return new JdbcClientDetailsService(dataSource);  
 }  
  
 /\*\*  
 \* @info 注入认证管理类  
 \*/  
 @Autowired  
 private AuthenticationManager authenticationManager;  
  
 /\*\*  
 \* @info 注入UserDetailsService 的自定义实现服务  
 \* 用于校验客户信息并以此颁发TOKEN字串  
 \*/  
 @Autowired  
 private CCustomerService myUserDetailsService;  
  
 @Override  
 public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  
 clients.jdbc(dataSource);  
 }  
  
 /\*\*  
 \* @info 注入用户自定义异常翻译器，用来自定义异常信息输出  
 \*/  
 @Autowired  
 private WebResponseExceptionTranslator customerWebResponseExceptionTranslator;  
  
 /\*\*  
 \* @info 服务认证配置授权类型 授权主要由 AuthorizationEndpoint 负责。  
 \* @param endpoints  
 \* @throws Exception  
 \*/  
 @Override  
 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  
 endpoints.authenticationManager(authenticationManager);  
 endpoints.tokenStore(tokenStore());  
 endpoints.userDetailsService(myUserDetailsService);  
 endpoints.setClientDetailsService(clientDetails());  
 //配置TokenServices参数  
 DefaultTokenServices tokenServices = new DefaultTokenServices();  
 tokenServices.setTokenStore(endpoints.getTokenStore());  
 tokenServices.setSupportRefreshToken(true);  
 tokenServices.setClientDetailsService(endpoints.getClientDetailsService());  
 tokenServices.setTokenEnhancer(endpoints.getTokenEnhancer());  
 tokenServices.setAccessTokenValiditySeconds((int) TimeUnit.DAYS.toSeconds(1)); // 1 天  
 tokenServices.setRefreshTokenValiditySeconds((int)TimeUnit.DAYS.toSeconds(1));// 30 天  
 endpoints.tokenServices(tokenServices);  
 endpoints.exceptionTranslator(customerWebResponseExceptionTranslator);  
 }  
  
 @Override  
 public void configure(AuthorizationServerSecurityConfigurer security) throws Exception{  
 security.tokenKeyAccess("permitAll()");  
 security.checkTokenAccess("isAuthenticated()");  
 security.allowFormAuthenticationForClients();  
 }  
  
 @Bean  
 @Primary  
 public DefaultTokenServices tokenServices() {  
 DefaultTokenServices tokenServices = new DefaultTokenServices();  
 tokenServices.setSupportRefreshToken(true);  
 tokenServices.setTokenStore(tokenStore());  
 return tokenServices;  
 }  
  
}

### 资源服务器认证配置

/\*\*  
 \* 配置资源自定义拦截器  
 \*/  
@Configuration  
@EnableResourceServer  
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {  
  
  
 @Autowired  
 private MyFilterSecurityInterceptor myFilterSecurityInterceptor;  
  
 /\*\*  
 \* 用于配置对受保护的资源的访问规则  
 \* 默认情况下所有不在/oauth/\*\*下的资源都是受保护的资源  
 \* @link OAuth2WebSecurityExpressionHandler  
 \*/  
 @Override  
 public void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests().anyRequest().authenticated();  
 http.addFilterBefore(myFilterSecurityInterceptor,FilterSecurityInterceptor.class);  
 }  
  
 @Override  
 public void configure(ResourceServerSecurityConfigurer resourceServerSecurityConfigurer) throws Exception{  
 resourceServerSecurityConfigurer.authenticationEntryPoint(new AuthExceptionEntryPoint()).accessDeniedHandler(new CustomAccessDeniedHandler());  
 }  
  
}

### 应用安全策略配置

@Configuration  
public class WebSecurityConfiguration extends GlobalAuthenticationConfigurerAdapter {  
  
 private final CCustomerService userService;  
  
 @Autowired  
 public WebSecurityConfiguration(CCustomerService userService) {  
 this.userService = userService;  
 }  
  
 @Override  
 public void init(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userService);  
 }  
   
}

### 服务验证策略配置

注：自定义安全拦截器

/\*\*  
 \* 重写验证拦截器  
 \*/  
@Service  
public class MyFilterSecurityInterceptor extends AbstractSecurityInterceptor implements Filter{  
  
 @Autowired  
 private FilterInvocationSecurityMetadataSource securityMetadataSource;  
  
 @Autowired  
 public void setMyAccessDecisionManager(MyAccessDecisionManager myAccessDecisionManager) {  
 super.setAccessDecisionManager(myAccessDecisionManager);  
 }  
  
 @Override  
 public void init(FilterConfig filterConfig) throws ServletException {  
 }  
  
 @Override  
 public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  
 FilterInvocation fi = new FilterInvocation(request, response, chain);  
 invoke(fi);  
 }  
  
  
 public void invoke(FilterInvocation fi) throws IOException, ServletException {  
  
 /\*\*  
 \* Fi 里面有一个被拦截的url  
 \* 里面调用MyInvocationSecurityMetadataSource的getAttributes(Object object)这个方法获取fi对应的所有权限  
 \* 再调用MyAccessDecisionManager的decide方法来校验用户的权限是否足够  
 \*/  
 InterceptorStatusToken token = super.beforeInvocation(fi);  
  
 try {  
 //执行下一个拦截器  
 fi.getChain().doFilter(fi.getRequest(), fi.getResponse());  
 } finally {  
 super.afterInvocation(token, null);  
 }  
  
 }  
  
 @Override  
 public void destroy() {  
 }  
  
 @Override  
 public Class<?> getSecureObjectClass() {  
 return FilterInvocation.class;  
 }  
  
 @Override  
 public SecurityMetadataSource obtainSecurityMetadataSource() {  
 return this.securityMetadataSource;  
 }  
  
}

/\*\*  
 \* 校验请求路径是否具有权限  
 \*/  
@Service  
public class MyAccessDecisionManager implements AccessDecisionManager {  
  
 /\*\*  
 \* Decide 方法是判定是否拥有权限的决策方法，  
 \* Authentication 是释CustomUserService中循环添加到 GrantedAuthority 对象中的权限信息集合.  
 \* Object 包含客户端发起的请求的requset信息，可转换为 HttpServletRequest request = ((FilterInvocation) object).getHttpRequest();  
 \* ConfigAttributes 为MyInvocationSecurityMetadataSource的getAttributes(Object object)这个方法返回的结果，此方法是为了判定用户请求的url 是否在权限表中，如果在权限表中，则返回给 decide 方法，用来判定用户是否有此权限。如果不在权限表中则放行。  
 \*/  
 @Override  
 public void decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes) throws AccessDeniedException, InsufficientAuthenticationException {  
  
 if(null== configAttributes || configAttributes.size() <=0) {  
 return;  
 }  
 ConfigAttribute c;  
 String needRole;  
 for(Iterator<ConfigAttribute> iter = configAttributes.iterator(); iter.hasNext(); ) {  
 c = iter.next();  
 needRole = c.getAttribute();  
 for(GrantedAuthority ga : authentication.getAuthorities()) {  
 //authentication 为在注释1 中循环添加到 GrantedAuthority 对象中的权限信息集合  
 if(needRole.trim().equals(ga.getAuthority())) {  
 return;  
 }  
 }  
 }  
 throw new AccessDeniedException("no right");  
 }  
  
 @Override  
 public boolean supports(ConfigAttribute attribute) {  
 return true;  
 }  
  
 @Override  
 public boolean supports(Class<?> clazz) {  
 return true;  
 }  
  
}

@Service  
public class MyFilterInvocationSecurityMetadataSource implements FilterInvocationSecurityMetadataSource {  
  
 @Autowired  
 private CPermissionDAO permissionDao;  
  
 private HashMap<String, Collection<ConfigAttribute>> map =null;  
  
 /\*\*  
 \* 加载权限表中所有权限  
 \*/  
 public void loadResourceDefine(){  
 map = new HashMap<>();  
 Collection<ConfigAttribute> array;  
 ConfigAttribute cfg;  
 List<CPermission> permissionList = permissionDao.queryAllPermission();  
 for(CPermission permission : permissionList) {  
 array = new ArrayList<>();  
 cfg = new SecurityConfig(permission.getPermissionUrl());  
 //此处添加的信息将会作为MyAccessDecisionManager类的decide的第三个参数。  
 array.add(cfg);  
 //用权限的getUrl() 作为map的key，用ConfigAttribute的集合作为 value，  
 map.put(permission.getPermissionUrl(), array);  
 }  
  
 }  
  
 //此方法是为了判定用户请求的url 是否在权限表中，如果在权限表中，则返回给 decide 方法，用来判定用户是否有此权限。如果不在权限表中则放行。  
 @Override  
 public Collection<ConfigAttribute> getAttributes(Object object) throws IllegalArgumentException {  
 if(map == null) loadResourceDefine();  
 //Object 中包含用户请求的Request 信息  
 HttpServletRequest request = ((FilterInvocation) object).getHttpRequest();  
 AntPathRequestMatcher matcher;  
 String resUrl;  
 for(Iterator<String> iter = map.keySet().iterator(); iter.hasNext(); ) {  
 resUrl = iter.next();  
 matcher = new AntPathRequestMatcher(resUrl);  
 if(matcher.matches(request)) {  
 return map.get(resUrl);  
 }  
 }  
 return null;  
 }  
  
 @Override  
 public Collection<ConfigAttribute> getAllConfigAttributes() {  
 return null;  
 }  
 @Override  
 public boolean supports(Class<?> clazz) {  
 return true;  
 }  
}

### 异常信息自定义配置

@Component("customAccessDeniedHandler")  
public class CustomAccessDeniedHandler implements AccessDeniedHandler {  
  
 /\*\*  
 \* @info 权限不足时修改异常提示，原异常信息为：Access is denied  
 \* @param request  
 \* @param response  
 \* @param accessDeniedException  
 \* @throws IOException  
 \* @throws ServletException  
 \*/  
 @Override  
 public void handle(HttpServletRequest request, HttpServletResponse response, AccessDeniedException accessDeniedException) throws IOException, ServletException {  
 ExceptionErrorInfoEntity errorInfoEntity = ErrorBuilder.buildBizError(HttpServletResponse.SC\_UNAUTHORIZED, SystemErrorConstants.OAUTH\_TOKEN\_GET\_ERROR, "权限不足，拒绝访问");  
 String jsonString = JSON.toJSONString(new ResponseModel(errorInfoEntity));  
 response.setContentType("application/json;charset=UTF-8");  
 response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);  
 response.getWriter().write(jsonString);  
 }  
  
}

/\*\*  
 \* @info 自定义TOKEN 异常信息  
 \* @date 2018-10-19  
 \*/  
public class AuthExceptionEntryPoint implements AuthenticationEntryPoint {  
 @Override  
 public void commence(HttpServletRequest request, HttpServletResponse response,AuthenticationException authException) throws RuntimeException {  
 Throwable cause = authException.getCause();  
 if(cause instanceof InvalidTokenException) {  
 throw new BizException(ErrorBuilder.buildBizError(SystemErrorConstants.SYSTEM\_HTTP\_STATUS\_ERROR\_401, SystemErrorConstants.OAUTH\_TOKEN\_NO\_ALLOW,"无效的Token"));  
 }else{  
 throw new BizException(ErrorBuilder.buildBizError(SystemErrorConstants.SYSTEM\_HTTP\_STATUS\_ERROR\_401,SystemErrorConstants.OAUTH\_TOKEN\_NO\_ALLOW,"请传入认证信息"));  
 }  
 }  
}

/\*\*  
 \* @info 自定义异常翻译器  
 \* @date 2018-10-19  
 \*/  
@Component("customerWebResponseExceptionTranslator")  
public class CustomWebResponseExceptionTranslator implements WebResponseExceptionTranslator {  
  
 private static final Logger logger = LoggerFactory.getLogger(CustomWebResponseExceptionTranslator.class);  
  
 /\*\*  
 \* 抛出异常,跳出OAuth原有异常逻辑  
 \* @param ex  
 \* @return  
 \* @throws Exception  
 \*/  
 @Override  
 public ResponseEntity<OAuth2Exception> translate(Exception ex) throws Exception {  
 if(ex instanceof OAuth2Exception) {  
 OAuth2Exception oAuth2Exception = (OAuth2Exception) ex;  
 logger.error("OAuth认证异常： STATUS\_CODE = " + oAuth2Exception.getHttpErrorCode() + " | ERROR\_MESSAGE = " + oAuth2Exception.getMessage());  
 Integer statusCode = (Integer) oAuth2Exception.getHttpErrorCode();  
 ExceptionErrorInfoEntity errorInfoEntity = ErrorBuilder.buildBizError(statusCode, SystemErrorConstants.OAUTH\_TOKEN\_GET\_ERROR, "无效认证");  
 return new ResponseEntity(new ResponseModel(errorInfoEntity), HttpStatus.valueOf(statusCode));  
 }else{  
 logger.error("获取TOKEN异常：ERROR\_MESSAGE = " + ex.getMessage());  
 ExceptionErrorInfoEntity errorInfoEntity = ErrorBuilder.buildBizError(SystemErrorConstants.SYSTEM\_HTTP\_STATUS\_ERROR\_500, SystemErrorConstants.OAUTH\_TOKEN\_GET\_ERROR\_OTHER, "无效认证");  
 return new ResponseEntity(new ResponseModel(errorInfoEntity), HttpStatus.valueOf(SystemErrorConstants.SYSTEM\_HTTP\_STATUS\_ERROR\_500));  
 }  
 }  
}

public class CustomerOauthExceptionSerializer extends StdSerializer<CustomerOauthException> {  
  
 private static final Logger logger = LoggerFactory.getLogger(CustomerOauthExceptionSerializer.class);  
  
 public CustomerOauthExceptionSerializer() {  
 super(CustomerOauthException.class);  
 }  
  
 @Override  
 public void serialize(CustomerOauthException value, JsonGenerator gen, SerializerProvider provider) {  
 throw new BizException(ErrorBuilder.buildBizError(value.getHttpErrorCode(),"200006",value.getMessage()));  
 }   
}

# 附录