# **Oauth2.0**

OAuth2.0是一种开放的协议，为桌面程序或基于BS的web应用提供了一种简单的，标准的方式去访问需要用户授权的API 服务。

## ****一 应用场景****

为了更好的理解OAuth2的适用场合，举一个假设的例子；

假设有两家互联网企业 A 和 B，其中 B 是一家提供相片云存储的公司。即 B 的用户可以把相片上传到 B 网站上长期保存，然后可以在不同的设备上查看。某一天，A 和 B 谈成了一项合作：希望 B 用户在使用 A 的客户端时，也可以观看他在 B 的相片。按照普通的做法是：

要不让 B 提供一个接口：

http://xxx.xxx.com/getPhoto?account=

参数：account ： B 账号；

返回：指定账号下的所有相片；

有了这个接口，A 的客户端只需在界面上显示一个输入框，让用户输入他的 B 账号，然后调用这个接口来获取相片就可以了。

这样可行吗？

NO

因为实现并开放这样一个接口，相当于直接把 B 公司的相片资源全部暴露在互联网中，虽然并没有公开，但是对于有点意识的技术人员来说，利用这个接口获取用户图片非常容易。

那还可以加个密码参数：

http://xxx.xxx.com/getPhoto?account=&pwd=

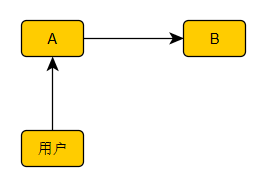
除了要求用户输入账号，还要输入密码。只有当账号密码验证通过，才返回该账号下的所有相片。这样，即使发现了这个接口，他不知道用户的密码，所以没办法窃取用户的相片了。

这样做对于获取图片的安全性已经没什么问题了，但实际上，这种方法更加不可取，因为这样做的话，用户必须在 A 的软件里输入他在 B 的账号和密码，从用户的角度就已经感受到一种不安全感，凭什么让我信任你 A，就比如说用户会把淘宝的账号密码告诉腾讯么？

同样，站在B公司（云存储图片）的角度，也不可能完全相信A公司，如果A公司在接收到账号密码之后马上把请求发过来，这是OK的，但是A公司如果中间把B公司的账号存储起来呢？

那怎么办呢？

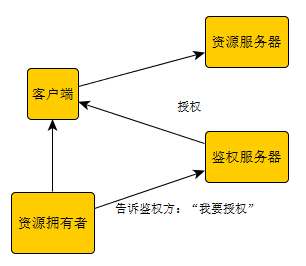
我们分析一下这个问题产生的原因，主要是在于 A、B、用户 三方的交互模型有问题。请看：



在这个场景下，用户需要访问他在 B 的相片资源，但是他不能直接和 B 打交道，而是必须通过 A。在这个前提下去考虑问题，无论如何无法想出一个既能实现功能，又能让用户和 B 都感到放心的实现方案。

Oauth2.0 就是为了解决这个“不信任”的问题而提出来的交互模型。它告诉人们，在这种场景下，三方要怎么打交道，才能做到安全、合理。

具体来说，Oauth2.0 的交互模型的核心是这个样子的：



**资源拥有者**，即用户；

**客户端**，泛指所有需要访问资源的软件实体。在我们上面举的例子中，资源即 B 用户的相片。A 要访问 B 用户的相片，那么 A 就是一个客户端；

**资源服务器**，即资源的存放地点，或者说资源的访问入口。在例子中，资源服务器即 getPhoto 接口所部署的服务器。客户端必须经由这里去访问资源；

**鉴权服务器**，这是一个对用户的身份进行认证、并对客户端进行授权的地方。这也是在我们的例子中所没有的一个节点。通常情况下，鉴权服务器也是属于 B 公司的。

好，接下来看看整个交互过程是怎样的。

　　首先，同样是 B 的用户在使用 A 的软件，然后，A 需要访问 B 用户的相片。这个时候，A 并不是展示一个输入框给用户，而是打开一个页面。这个页面就是 B 部署在鉴权服务器上面的一个鉴权页面，通常情况下，它长得类似下面这个样子：

上面这个是腾讯给有道云笔记进行授权的页面。

　　这个页面有两个要素：

　　1，有认证机制。在腾讯这个例子中，你需要输入QQ账号密码，证明你是一个合法的QQ用户；

　　2，展示了授权信息。看页面右方“有道云笔记将获得以下权限”部分。这是在告诉用户，如果你授权给客户端，那么，客户端将获得访问你这些资源的权限；

　　注意，这个页面是部署在 B 的鉴权服务器上，所有用户输入的账号密码是直接提交给 B，A 是没有任何机会拿到的。

　　如果用户同意授权并且认证通过，那么，接下来鉴权服务器会通知 A，并给 A 发送一个访问令牌（access token）。有了这个访问令牌，A 就可以拿着它去找 资源服务器要资源了。

　　所以，获取相片的接口会是这个形式：

　　http://xxx.xxx.com/getPhoto?accesstoken=

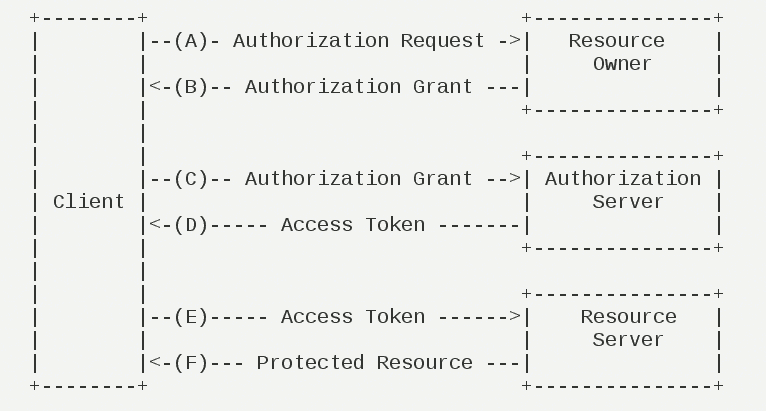
　　资源服务器在接收到这个请求之后，会拿着 access token，再去找鉴权服务器，检查这个 access token 的合法性和权限，如果通过的话，才返回资源给客户端。

## ****二 Oauth2思路****

OAuth在"客户端"与"服务提供商"之间，设置了一个授权层（authorization layer）。"客户端"不能直接登录"服务提供商"，只能登录授权层，以此将用户与客户端区分开来。"客户端"登录授权层所用的令牌（token），与用户的密码不同。用户可以在登录的时候，指定授权层令牌的权限范围和有效期。

"客户端"登录授权层以后，"服务提供商"根据令牌的权限范围和有效期，向"客户端"开放用户储存的资料。

## ****三 运行流程****



（A）用户打开客户端以后，客户端要求用户给予授权。

（B）用户同意给予客户端授权。

（C）客户端使用上一步获得的授权，向认证服务器申请令牌。

（D）认证服务器对客户端进行认证以后，确认无误，同意发放令牌。

（E）客户端使用令牌，向资源服务器申请获取资源。

（F）资源服务器确认令牌无误，同意向客户端开放资源。

上面六个步骤之中，B是关键，即用户怎样才能给于客户端授权。有了这个授权以后，客户端就可以获取令牌，进而凭令牌获取资源。

## ****四 授权模式****

OAuth 2.0定义了四种授权方式：

* 授权码模式（authorization code）
* 简化模式（implicit）
* 密码模式（resource owner password credentials）
* 客户端模式（client credentials）

## ****五 授权码模式****

（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"（redirection 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的服务器上完成的，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，向客户端发送访问令牌（access token）和更新令牌（refresh token）。

下面是上面这些步骤所需要的参数。

A步骤中，客户端申请认证的URI，包含以下参数：

response\_type：表示授权类型，必选项，此处的值固定为"code"

client\_id：表示客户端的ID，必选项

redirect\_uri：表示重定向URI，可选项

scope：表示申请的权限范围，可选项

state：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

例子：



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：

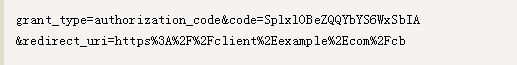
grant\_type：表示使用的授权模式，必选项，此处的值固定为"authorization\_code"。

code：表示上一步获得的授权码，必选项。

redirect\_uri：表示重定向URI，必选项，且必须与A步骤中的该参数值保持一致。

client\_id：表示客户端ID，必选项。

下面是一个例子：



E步骤中，认证服务器发送的HTTP回复，包含以下参数：

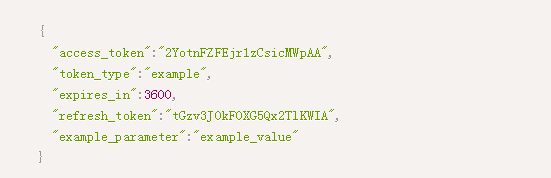
access\_token：表示访问令牌，必选项。

token\_type：表示令牌类型，该值大小写不敏感，必选项，可以是bearer类型或mac类型。

expires\_in：表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。

refresh\_token：表示更新令牌，用来获取下一次的访问令牌，可选项。

scope：表示权限范围，如果与客户端申请的范围一致，此项可省略。



其他三种模式：

* 简化模式（implicit grant type）：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，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且客户端不需要认证。
* 密码模式（resource owner password credentials）: 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"服务商提供商"索要授权。在这种模式中，用户必须把自己的密码给客户端，但是客户端不得储存密码。这通常用在用户对客户端高度信任的情况下，比如客户端是操作系统的一部分，或者由一个著名公司出品。而认证服务器只有在其他授权模式无法执行的情况下，才能考虑使用这种模式。(这中模式适用于微服务)。
* 客户端模式（Client Credentials Grant）：客户端模式指客户端以自己的名义，而不是以用户的名义，向"服务提供商"进行认证。严格地说，客户端模式并不属于OAuth框架所要解决的问题。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求"服务提供商"提供服务，其实不存在授权问题。

## ****六 更新令牌****

如果用户访问的时候，客户端的"访问令牌"已经过期，则需要使用"更新令牌"申请一个新的访问令牌。

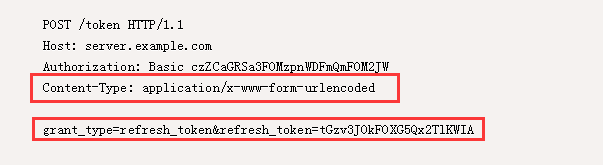
客户端发出更新令牌的HTTP请求，包含以下参数：

granttype：表示使用的授权模式，此处的值固定为"refreshtoken"，必选项。

refresh\_token：表示早前收到的更新令牌，必选项。

scope：表示申请的授权范围，不可以超出上一次申请的范围，如果省略该参数，则表示与上一次一致。

下面是一个例子：



## ****七 搭建授权服务****

### 添加依赖

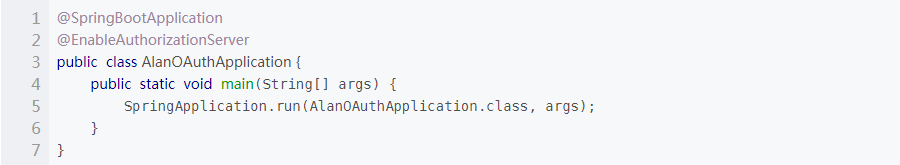
授权服务是基于Spring Security的，因此需要在项目中引入两个依赖：



前者为 Security，后者为Security的OAuth2扩展。

### 添加注解和配置

在启动类中添加@EnableAuthorizationServer注解：



完成这些我们的授权服务最基本的骨架就已经搭建完成了。但是要想跑通整个流程，我们必须分配 client\_id, client\_secret才行。Spring Security OAuth2的配置方法是编写@Configuration类继承AuthorizationServerConfigurerAdapter，然后重写void configure(ClientDetailsServiceConfigurer clients)方法，如：

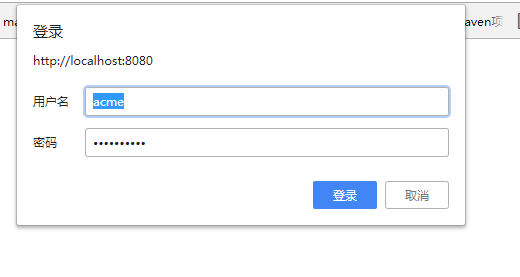


### 授权流程

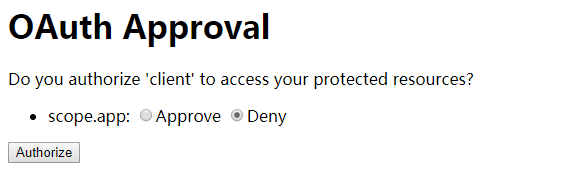
访问授权页面：

http://localhost:8080/oauth/authorize?client\_id=client&response\_type=code&redirect\_uri=http://www.baidu.com

此时浏览器会让你输入用户名密码，这是因为 Spring Security 在默认情况下会对所有URL添加Basic Auth认证。默认的用户名为user, 密码是随机生成的，在控制台日志中可以看到。



点击登录后跳转该页面：



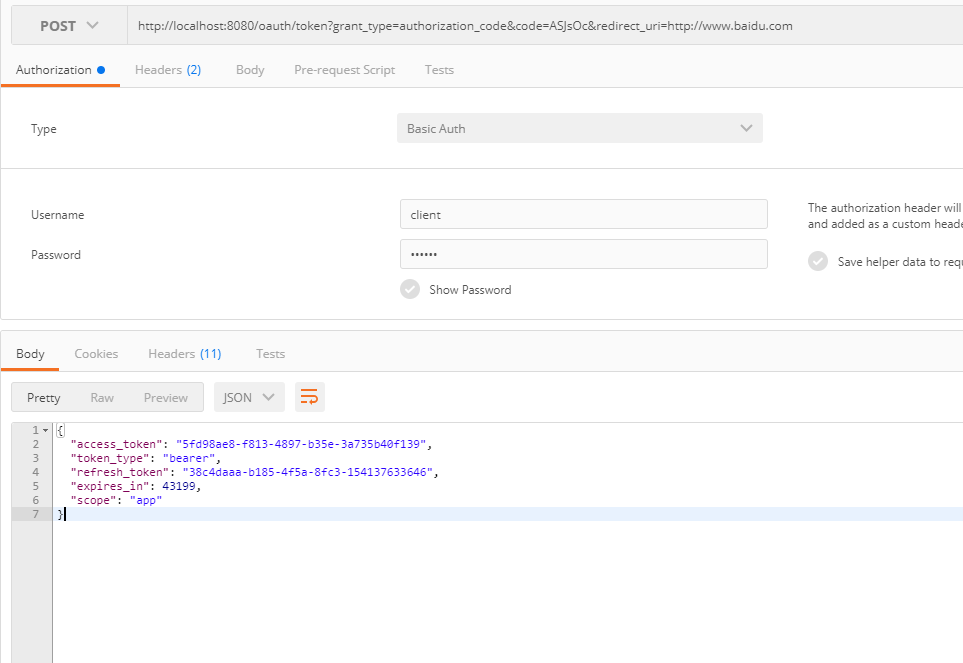
点击Authorize后，浏览器就会重定向到[百度](https://www.baidu.com/s?wd=%E7%99%BE%E5%BA%A6&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，并带上code参数：



拿到code以后，就可以调用

http://localhost:8080/oauth/token?grant\_type=authorization\_code&code=ASJsOc&redirect\_uri=http://www.baidu.com

来换取access\_token了



到此我们最最基本的授权服务就搭建完成了，但是完全无法运用到工作中。

认证服务需要重写AuthorizationServerConfigurerAdapter 中的三个方法：

• 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)。

认证服务器一些重要的方法：

/oauth/authorize：验证

/oauth/token：获取token

/oauth/confirm\_access：用户授权

/oauth/error：认证失败

/oauth/check\_token：资源服务器用来校验token

/oauth/token\_key：如果jwt模式则可以用此来从认证服务器获取公钥

### 管理令牌

对于令牌的管理有三种方式：

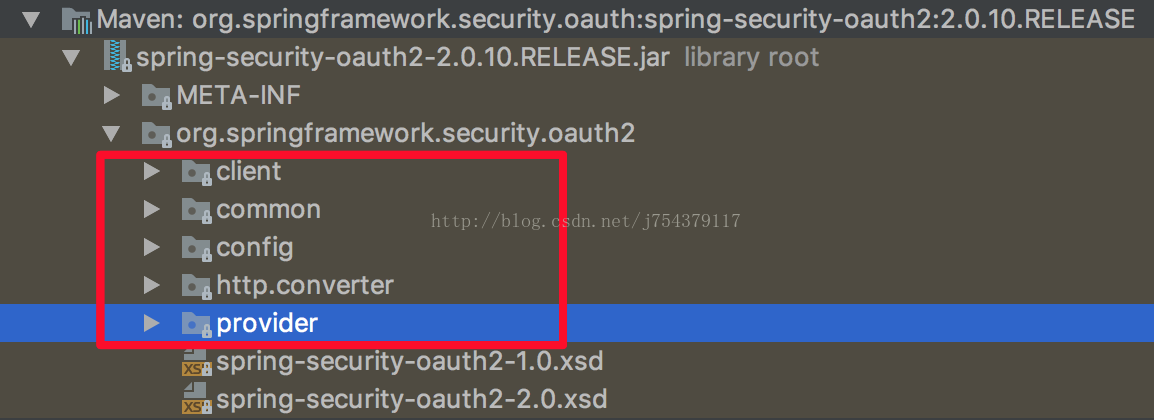
InMemoryTokenStore：这个版本的实现是被默认采用的，它可以完美的工作在单服务器上（即访问并发量压力不大的情况下，并且它在失败的时候不会进行备份），大多数的项目都可以使用这个版本的实现来进行尝试，你可以在开发的时候使用它来进行管理，因为不会被保存到磁盘中，所以更易于调试。

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）。

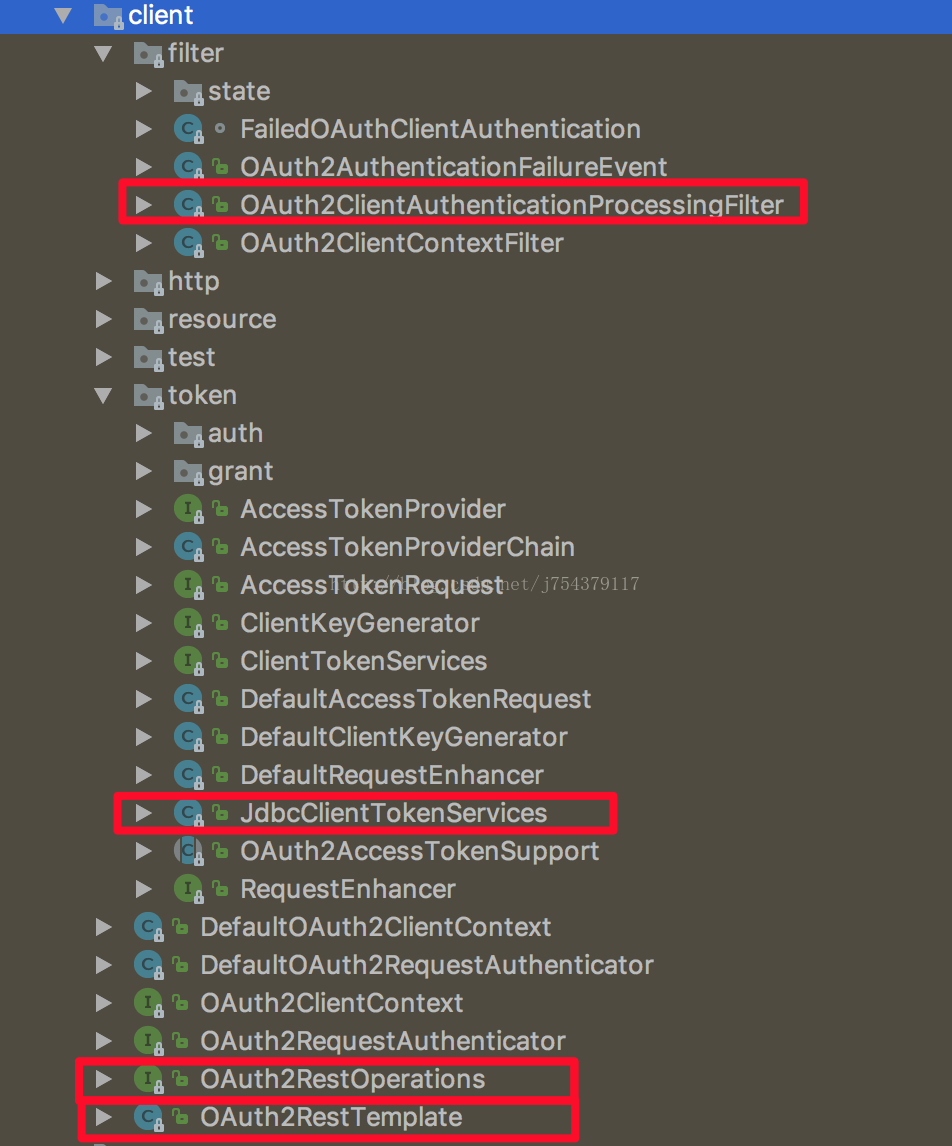
## ****八**** 源码简介

Spring cloud oauth2.0的代码结构图如下：



可以看到Spring oauth2.0的代码结构分为了五层，client层负责应用客户端的管理。common层为公共类的所在地。config层为配置层。http.converter层为转换层。provider层是最重要的一层，它负责管理认证[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)和资源服务器。

### client层：如图



几个比较典型的类：

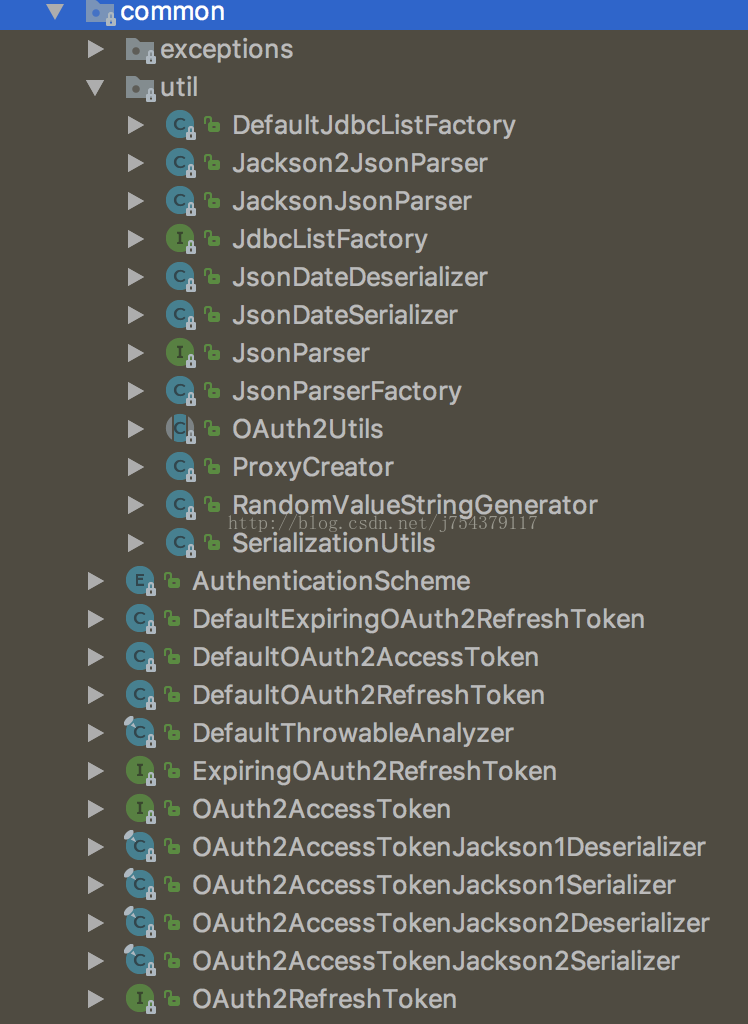
（1）OAuth2ClientAuthenticationProcessingFilter：负责对客户端的访问进行过滤，验证token处理等操作。

（2）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。

（3）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。

当然还有其它的一些类，比如DefaultOAuth2ClientContext实现了OAuth2ClientContext接口，里面存储了client的上下文信息。

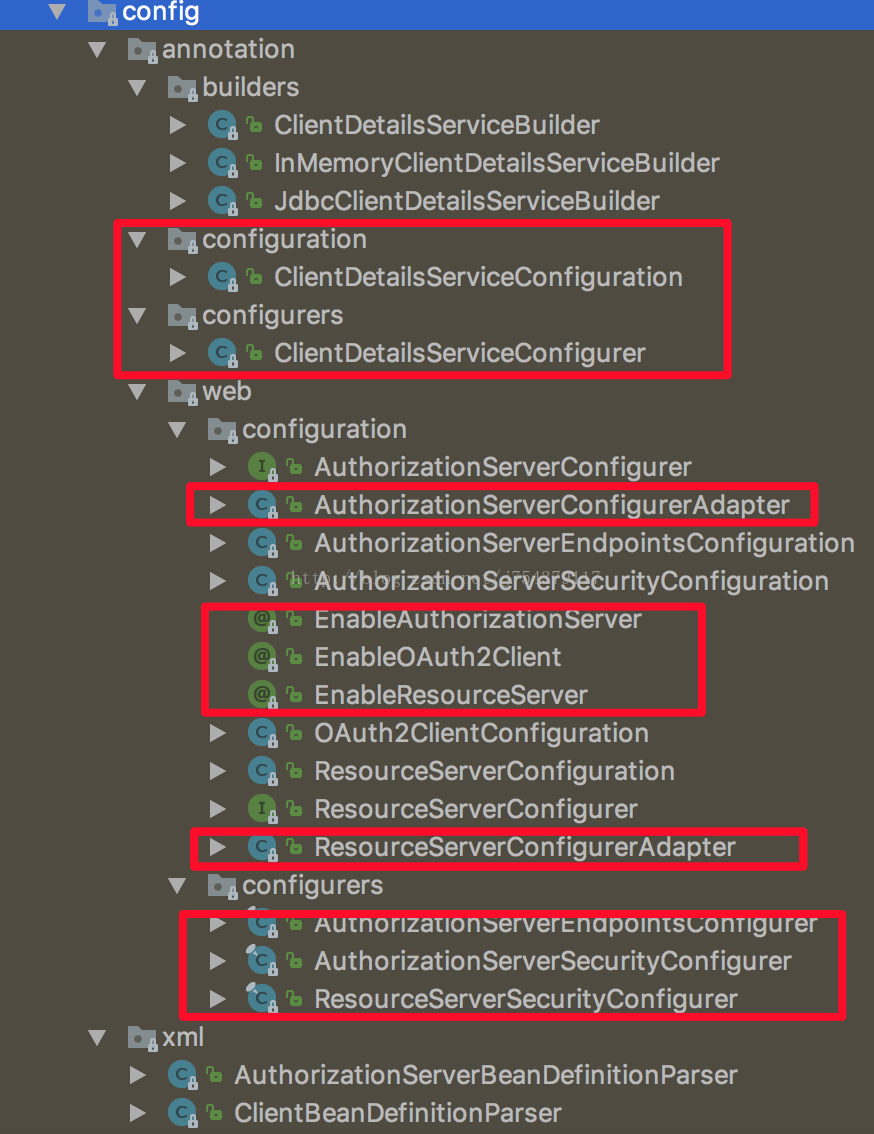
### common公共层：如图



这一层主要是对json字符串处理、解析和实例化等操作

### config层：

注解的定义，以及认证服务器和资源服务器的各个配置类的定义都在这个包，是贯穿整个代码非常核心的一层。如图：



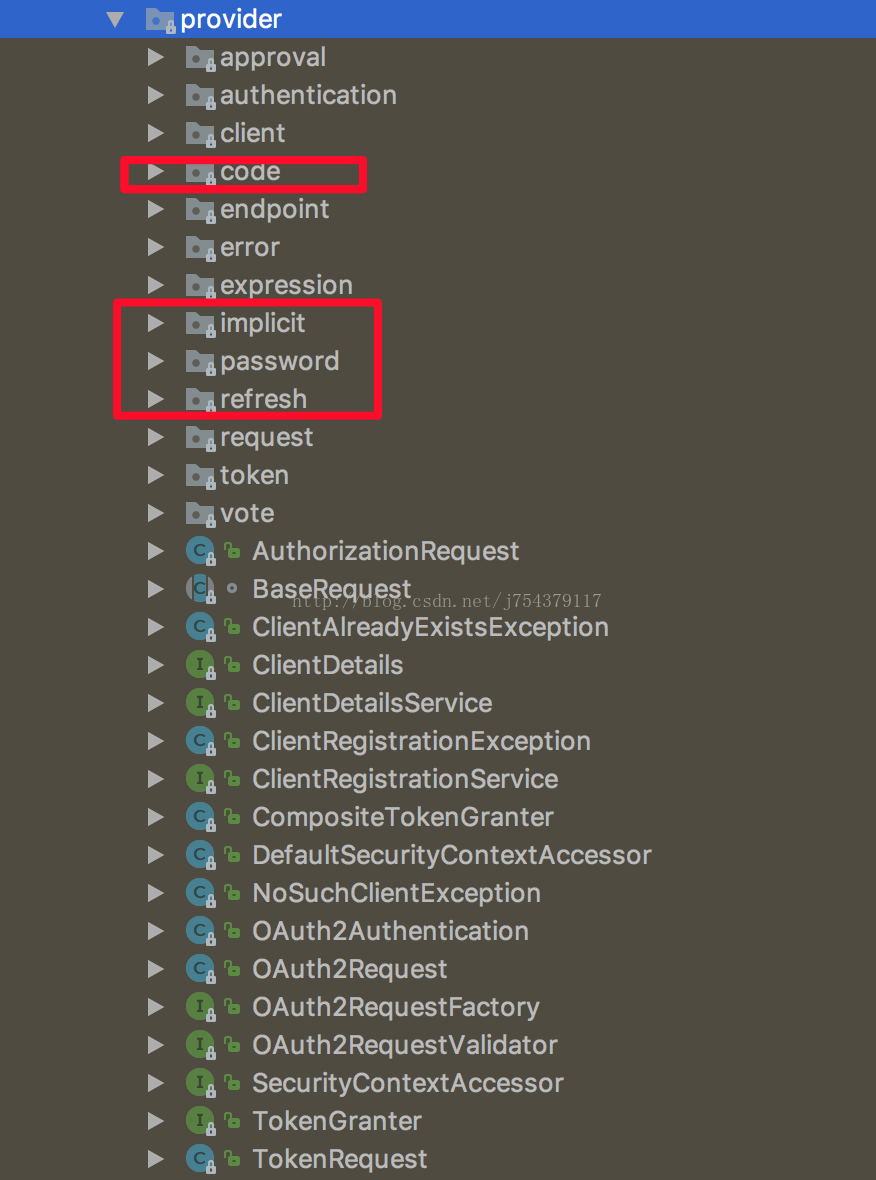
可见，我们在代码里所用的三个基本注解EnableAuthorizationServer、EnableOAuth2Client、EnableResourceServer都在这里。还有

AuthorizationServerConfigurerAdapter、ResourceServerConfigurerAdapter这两个最核心的配置类也在这里，只要实现这两个核心配置类，复写他们的函数便可以实现

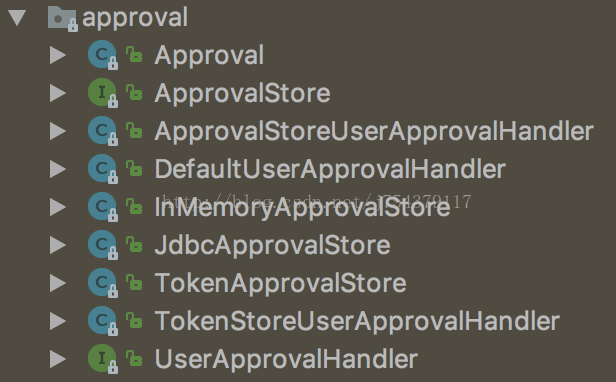
个性化的配置。

### provider层

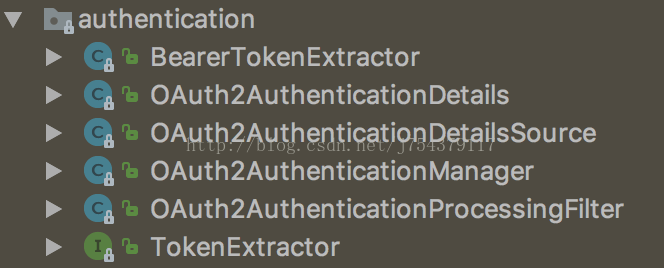
最核心的一层，里面包含了endpoint、认证、以及四种授权方式的管理。如图：



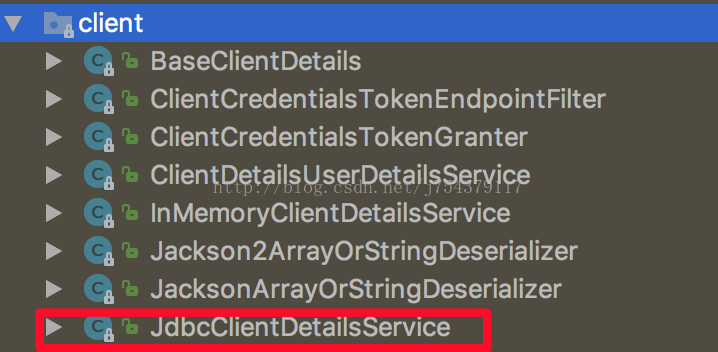
**关于approval包：**里面包含了授权的信息，如果想要自定义展示页面上的元素，需要研读此包里面的代码进行改写，如图：



authentication包：这一层只讲一个类，就是OAuth2AuthenticationProcessingFilter类，他主要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。

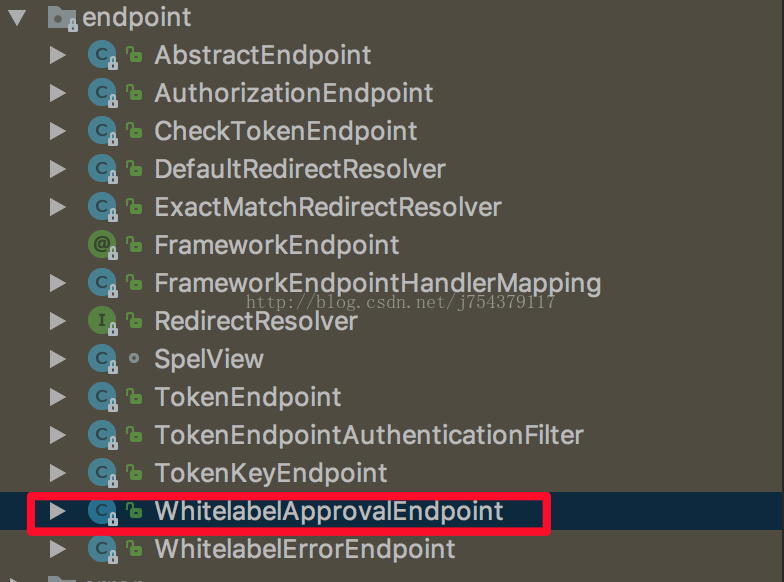


client包：这里只讲一个JdbcClientDetailsService一个类，打开此类的源码可以看到它里面声明了很多的sql语句。此类主要是关系着client信息的管理。以jdbc的方式进行读取。



code、implicit、password、refresh层，就是对应着四种授权方式的管理。

endpoint层：此层很重要，代码里面定义了基本的接口地址，如/oauth/authorize、/oauth/token、/oauth/confirm\_access等等。如下图，这里只说一下WhitelabelApprovalEndpoint类，此类里面声明了@RequestMapping({"/oauth/confirm\_access"})这个endpoint，这个endpoint就是生成我们的授权页面，就是那个有很多选项，让你选择允许还是拒绝的那个页面，这里此类并没有对应的web页面，而是在java代码里创建了一个html的ModelAndView视图，然后展示出来。所以如果你想要自定义授权页，最简单的方法就是把这个类copy到你的controller包里面，然后修改成你自己想要的。



## ****八****resource-server

资源服务器提供被OAuth2 令牌保护的资源，Spring OAuth提供一个Spring Security授权过滤器，它实现保护资源的功能。在@Configuration（全局配置）类中，你可以使用@EnableResourceServer来开启/关闭过滤器，通过继承 ResourceServerConfigurerAdapter 类来配置资源服务器。 下面是可配置的属性：

* tokenServices：定义令牌服务的实体（ResourceServerTokenServices类的实例）。
* resourceId：资源ID（可选的，如果不为空，授权服务器会对它进行验证）。
* 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。





## ****八 JWT****

JSON Web Token（JWT）是目前最流行的跨域身份验证解决方案

### JWT的原则

JWT的原则是在服务器身份验证之后，将生成一个JSON对象并将其发送回用户，如下所示。

{

"UserName": "name",

"Role": "Admin"

}

之后，当用户与服务器通信时，客户在请求中发回JSON对象。服务器仅依赖于这个JSON对象来标识用户。为了防止用户篡改数据，服务器将在生成对象时添加签名，服务器不保存任何会话数据，即服务器变为无状态，使其更容易扩展。

### JWT的数据结构

典型的一个JWT看起来如下图。

该对象为一个很长的字符串，字符之间通过"."分隔符分为三个子串。注意JWT对象为一个长字串，各字串之间也没有换行符，此处为了演示需要，我们特意分行并用不同颜色表示了。每一个子串表示了一个功能块，总共有以下三个部分：

JWT的三个部分如下。JWT头、有效载荷和签名，将它们写成一行如下。



### JWT头

JWT头部分是一个描述JWT元数据的JSON对象，通常如下所示。

{

"alg": "HS256",

"typ": "JWT"

}

在上面的代码中，alg属性表示签名使用的算法，默认为HMAC SHA256（写为HS256）；typ属性表示令牌的类型，JWT令牌统一写为JWT。

最后，使用Base64 URL算法将上述JSON对象转换为字符串保存。

### 有效载荷

有效载荷部分，是JWT的主体内容部分，也是一个JSON对象，包含需要传递的数据。 JWT指定七个默认字段供选择。

iss：发行人

exp：到期时间

sub：主题

aud：用户

nbf：在此之前不可用

iat：发布时间

jti：JWT ID用于标识该JWT

除以上默认字段外，我们还可以自定义私有字段，如下例：

{

"sub": "1234567890",

"name": "name",

"admin": true

}

请注意，默认情况下JWT是未加密的，任何人都可以解读其内容，因此不要构建隐私信息字段，存放保密信息，以防止信息泄露。

JSON对象也使用Base64 URL算法转换为字符串保存。

### 签名哈希

签名哈希部分是对上面两部分数据签名，通过指定的算法生成哈希，以确保数据不会被篡改。

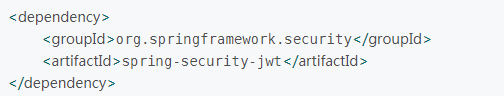
首先，需要指定一个密码（SigningKey）。该密码仅仅为保存在服务器中，并且不能向用户公开。然后，使用标头中指定的签名算法（默认情况下为HMAC SHA256）根据以下公式生成签名。

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),secret)

在计算出签名哈希后，JWT头，有效载荷和签名哈希的三个部分组合成一个字符串，每个部分用"."分隔，就构成整个JWT对象。

## **九 Oauth2.0+JWT**

首先，我们需要在我们的pom.xml中添加spring-security-jwt依赖项。



### 授权服务器

接下来，我们将配置我们的授权服务器使用JwtTokenStore - 如下所示



在JwtAccessTokenConverter中使用了一个对称密钥来签署我们的令牌 - 这意味着我们需要为资源服务器使用同样的密钥。

### 资源服务器

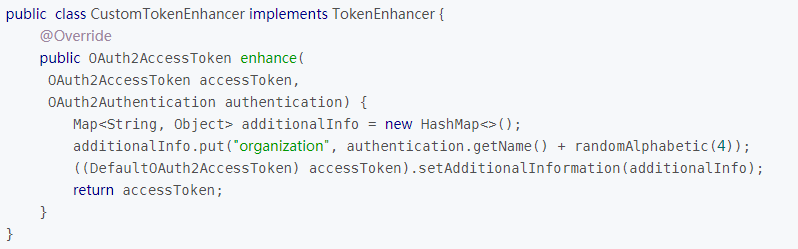
资源服务器配置 - 这与授权服务器的配置非常相似：



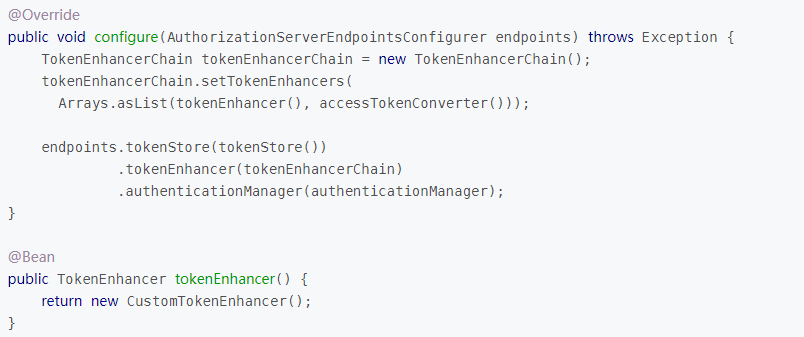
我们将这两个服务器定义为完全独立且可独立部署的服务器。这就是我们需要在新配置中再次声明一些相同的bean的原因。

### 令牌中的自定义声明

现在设置一些基础设施，以便能够在访问令牌中添加一些自定义声明。框架提供的标准声明都很好，但大多数情况下我们需要在令牌中使用一些额外的信息来在客户端使用。 我们将定义一个TokenEnhancer来定制我们的Access Token与这些额外的声明。 在下面的例子中，我们将添加一个额外的字段“organization”到我们的访问令牌 （CustomTokenEnhancer）：



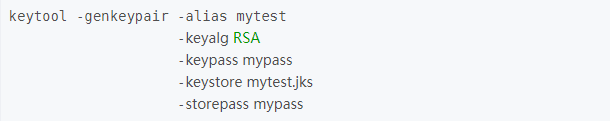
然后，我们将把它连接到我们的授权服务器配置 - 如下所示：



### 非对称加密

#### 1 生成JKS Java KeyStore文件

我们首先使用命令行工具keytool生成密钥 .jks文件：



该命令将生成一个名为mytest.jks的文件，其中包含我们的密钥 - 公钥和私钥。 还要确保keypass和storepass是一样的。

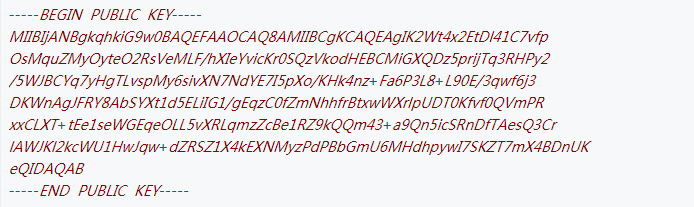
#### 导出公钥

我们需要从生成的JKS中导出我们的公钥，我们可以使用下面的命令来实现：



如果机器未安装OpenSSL，则需要代码完成抽取，如需要私钥，需将jks文件转换成pem格式，在解压，在线转换地址：https://myssl.com/cert\_convert.html

抽取后公钥：



将其复制到我们的资源服务器src / main / resources / public.txt中

接下来，我们不希望JMS文件被maven过滤进程拾取 - 所以我们将确保将其排除在pom.xml中：



我们需要确保我们的JKS文件通过Spring Boot Maven插件添加到应用程序classpath - addResources：



### 授权服务器

最后，我们需要配置我们的资源服务器使用公钥 - 如下所示：



## **十 释义**

TokenStore：oken令牌持久化

JwtAccessTokenConverter：根据签名生成JwtToken，告知资源服务器如何解析JWTTOKEN

authenticationManager: 直接注入一个AuthenticationManager，自动开启密码授权类型

TokenEnhancer : 令牌增强器

TokenEnhancerChain：增强链，将令牌增强其加入其中，完成自定义令牌部分；

DefaultTokenService：oauth2中操作token的默认实现，它使用随机值创建令牌，并处理除永久令牌以外的所有令牌，对于永久令牌，它委托TokenStore类进行处理。 令牌默认采用[基于内存实现的存储方式](http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/token/store/InMemoryTokenStore.html)，但也有一些其它的存储方式。 下面是其中一些方式的简介：

1.默认的InMemoryTokenStore（缓存） 处理类对于单服务器场景非常适用

2. JdbcTokenStore：是实现存储令牌的[JDBC 版本](https://github.com/spring-projects/spring-security-oauth/blob/master/docs/JdbcTokenStore)，它将令牌信息保存到数据库中；

3.jwt令牌