一 概括

1 spring security

Spring Security 是一个安全框架，前身是 Acegi Security，能够为 Spring 企业应用系统提供声明式的安全访问控制。Spring Security 基于 Servlet 过滤器、IoC 和 AOP，为 Web 请求和方法调用提供身份确认和授权处理，避免了代码耦合，减少了大量重复代码工作。

2 oauth2

oAuth 协议为用户资源的授权提供了一个安全的、开放而又简易的标准，协议。与以往的授权方式不同之处是 oAuth 的授权不会使第三方触及到用户的帐号信息（如用户名与密码），即第三方无需使用用户的用户名与密码就可以申请获得该用户资源的授权，而且 oAuth 是安全的。

3 为什么使用 oauth2

演示一个一个虎牙登录管理.

与此类似的场景还有很多，比如访问直播平台，访问视频网站，小说网站等等，都允许使用微信，微博，yy，qq 登录.这种情况就属于第三方授权. 这种场景即方便了用户使用客户端软件，因为不需要重新注册，也有利于授权方，因为用户体验提高了.

以直播为例.有一个直播平台，叫兔唇直播，要不就游客身份访问，要不就使用微信，qq 登录.登陆后可以显示 qq，微信用户的基本信息.

? 这种软件，平台，凭什么获取用户信息?

传统做法(给第三方用户名密码)就是实现登录逻辑.登录成功后就可以获取资源了。但这样的做法会有以下几个问题:

作为第三方，能不能使用用户用户名密码?显然不安全.

用户如何限制第三方应用程序获取用户资源的权限范围，有效期？难道把所有资料都永久暴露给它吗？显然也不能

如果一个第三方能够登录使用用户资源，那么也有有第二，第三个第三方登录使用用户资源，一旦用户修改了用户名密码，那么这些第三方将都会失效.

只要有一个接入的第三方应用程序遭到破解，那么用户的密码就会泄露那就更不安全了.

所以，出现了 oauth2 这种授权的协议.这种授权的协议可以让有用户资源的服务器(比如微信，qq)在不需要用户名密码的授权方式下，给第三方(比如直播，视频，读书软件)使用用户资源.并且保证安全

二　oauth 协议交互

1　名词解释

在学习 spring security oauth2 的过程中，会出现一些名词，统一解释一下.

1.A　第三方应用程序（Third-party application）：

又称之为客户端（client），比如访问直播平台，可以使用 qq 用户信息登录访问，这个直播平台就叫做第三方应用程序，对于 qq 用户资源服务器直播平台就是客户端.同时客户端软件不一定是同一个服务提供商的产品，也可以是同一个服务提供商的产品.

1.B　HTTP 服务提供商（HTTP service）：

比如 qq，微信这种产品所属的供应商，腾讯，就属于服务提供商.基本就是一个公司，或集团.

1.C　资源所有者（Resource Owner）：

又称之为用户（user），访问直播平台，允许直播平台使用用户信息的这个人，就是资源所有者。

1.D　认证授权服务器（Authorization server）：

即服务提供商专门用来处理认证授权的服务器，比如，授权时处理扫码，填写用户名密码的服务器就是认证授权的过程.认证和授权其实是 2 个不同的过程，可以同时在一台服务器处理，也可以分开处理.

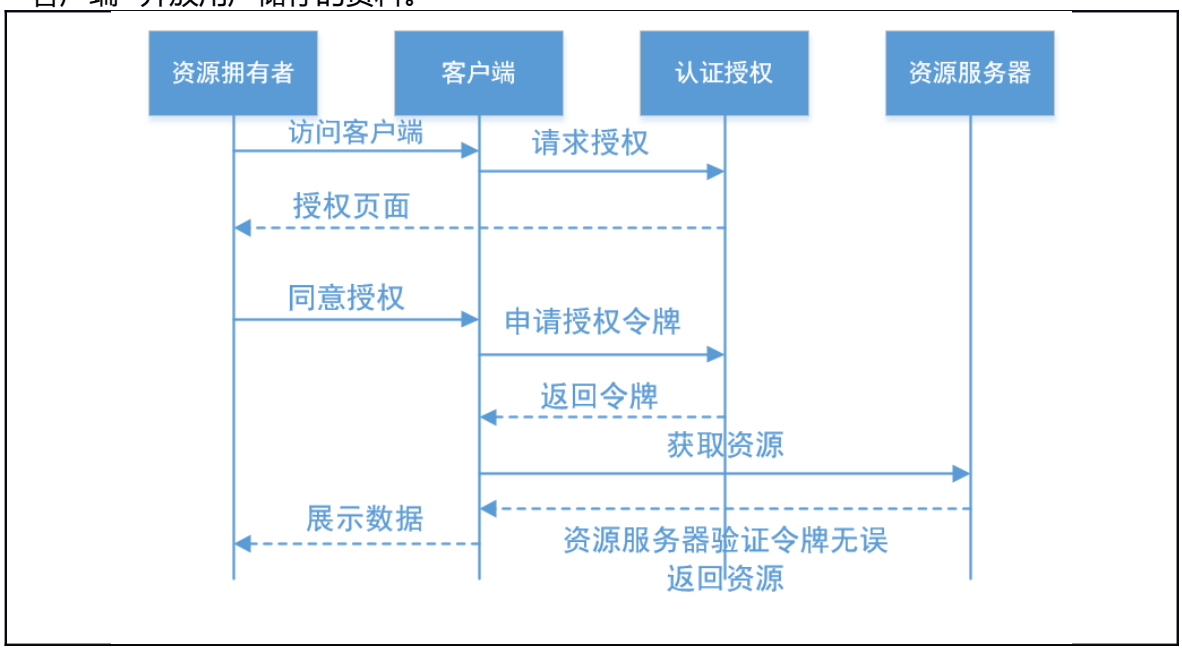
1.E　资源服务器（Resource server）：

即服务提供商存放用户生成的资源的服务器。比如，登录 qq 后可以添加好友，删除好友这些操作就是资源服务器功能，还比如，第三方登录后拿到的用户信息也是资源.都是服务提供商的东西，所以可以和认证授权服务器在同一个服务器中，也可以分开.

2　oauth2 授权流程

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。



这是 oauth2 一般授权流程,区别在于客户端不同,授权的方式也不同.基本流程如下

- 用户访问客户端功能
- 客户端第三方授权
- 服务提供商的认证授权返回授权页面(比如登录,比如扫码)
- 用户查看信息授权(比如用户点击授权)
- 客户端在授权通过后到认证授权服务器获取令牌 token
- 客户端拿着令牌 token 访问资源(比如第三方想要获取的用户基本信息)
- 最终展示数据(比如第三方软件显示 qq 用户的名称头像)

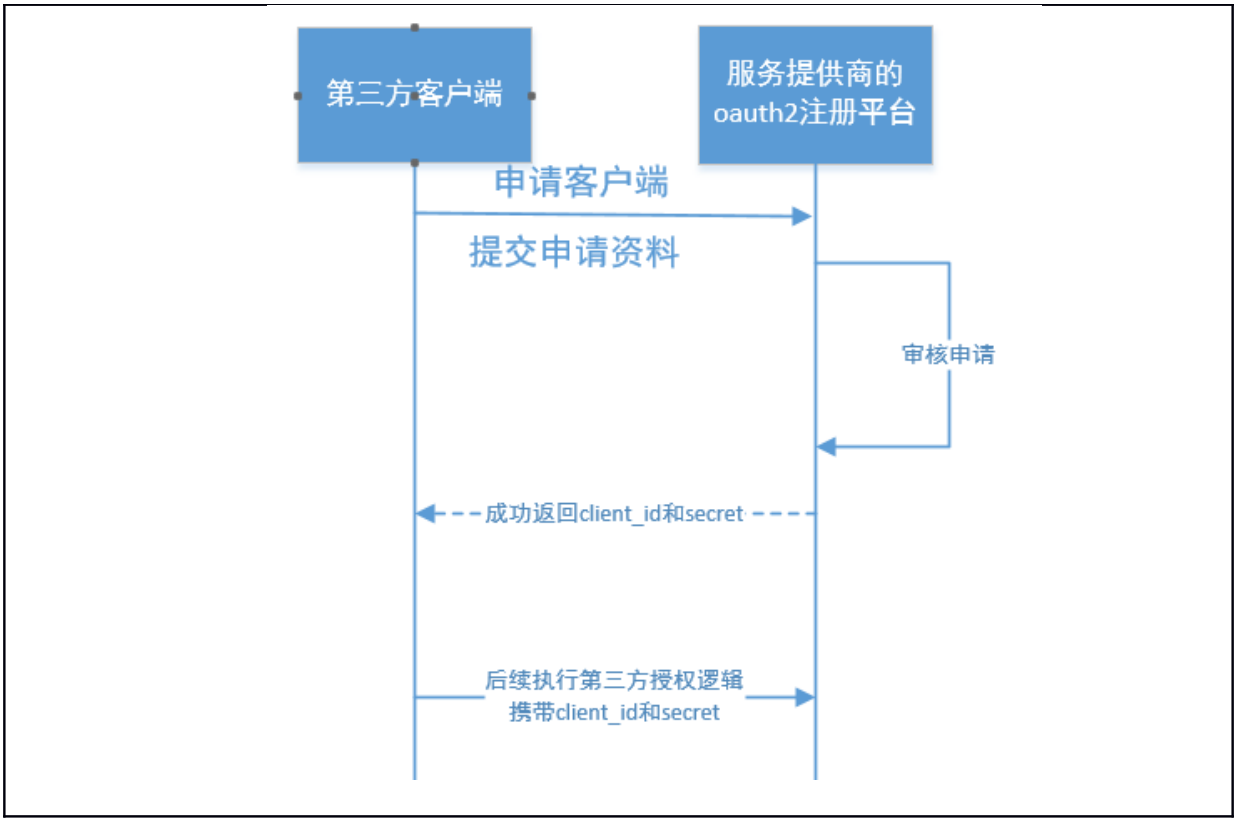
3 客户端身份

通过上述介绍,至少我们了解在这个交互过程中涉及到的三方都有:

- 资源拥有者:用户
- 第三方客户端:app
- 服务提供商:有 2 个角色功能
  - 认证
  - 授权

? 是不是所有的第三方客户端都可以到某一个服务提供商获取授权呢?

显然是不可以的.这里 oauth2 提出开放客户端的协议.当客户端想要通过某一个服务提供商的认证授权实现第三方登录,必须经过 oauth2 开放的一个可以注册,审核的平台,实现客户端身份的核实工作.比如虎牙想要开启微信登录功能,作为虎牙平台的注册公司要使用公司资质证件到微信服务商开放的注册平台注册 app 信息.平台会生成一个客户端的 **client_id** 同时生成对应这个 client_id 的 **secret** 密码.第三方客户端妥善保管这个信息,在授权认证时出示使用.服务提供商的授权认证才会允许执行,否则直接拒绝.



三令牌安全机制

1 ACCESS_TOKEN

Access Token 是客户端访问资源服务器的令牌。拥有这个令牌代表着得到用户的授权。然而，这个授权应该是 临时 的，有一定有效期。这是因为，Access Token 在使用的过程中 可能会泄露。给 Access Token 限定一个 较短的有效期 可以降低因 Access Token 泄露而带来的风险。

然而引入了有效期之后，客户端使用起来就不那么方便了。每当 Access Token 过期，客户端就必须重新向用户索要授权。这样用户可能每隔几天，甚至每天都需要进行授权操作。这是一件非常影响用户体验的事情。希望有一种方法，可以避免这种情况。

于是 oAuth2.0 引入了 Refresh Token 机制

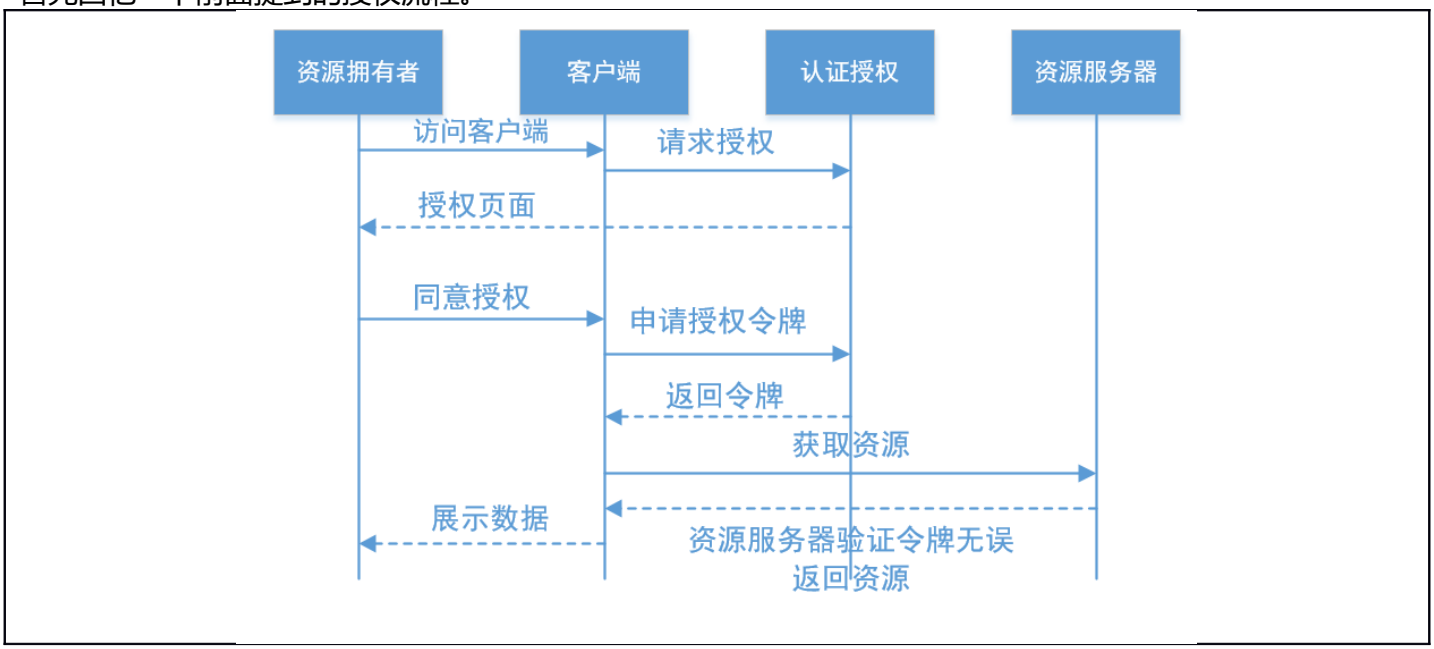
2 REFRESH_TOKEN

Refresh Token 的作用是用来刷新 Access Token。刷新令牌也是有有效期的,但是相对于 ACCESS_TOKEN 有效期非常长(可以是 1 个月甚至更长).使用刷新令牌可以跳过授权过程,也就是用户不在场的情况下,依然可以获取有效的 ACCESS_TOKEN.刷新令牌的使用必须配合 client_id 和 client_secret 同时使用,否则无效.并且刷新令牌要保存在客户端的服务器上,不能对外暴露显示.而且刷新令牌的作用是获取

ACCESS_TOKEN 即使泄露在没有 client_id 和 client_secret 的情况下什么都做不了(所以你会问,都泄露了呢?万一都泄露了,那客户端角色就是被攻破了.)

四授权模型

首先回忆一下前面提到的授权流程。



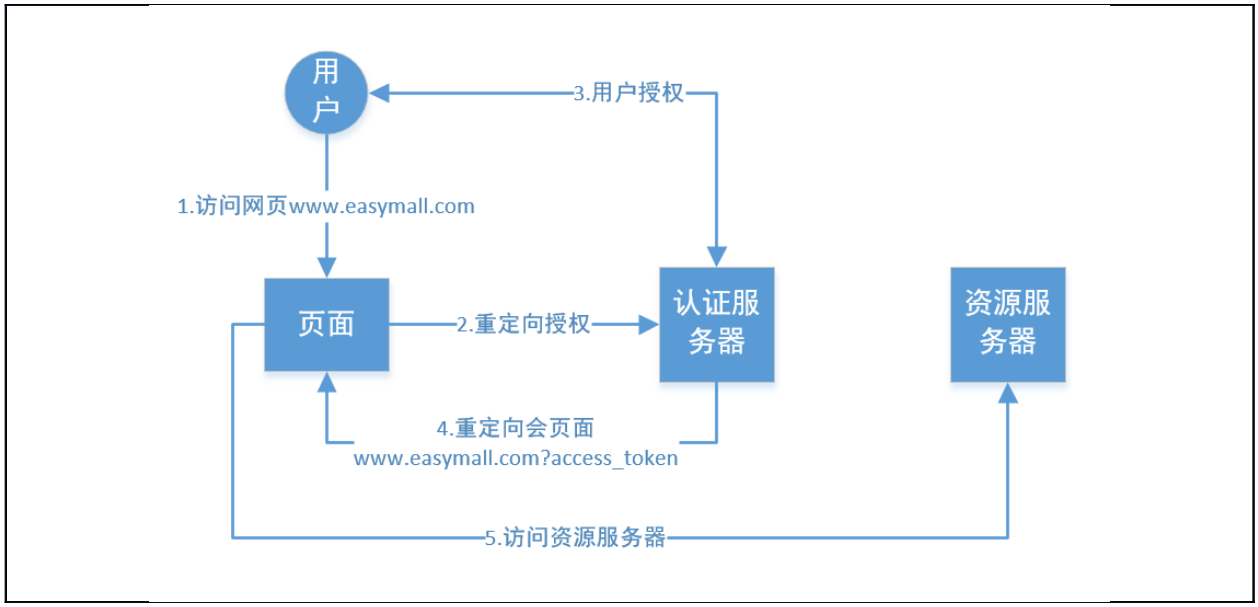
在这个过程中，当客户端请求授权时，可以根据应用场景不同选择不同的授权模型。oauth2 协议在这里提供了 4 中授权模型。

- implicit：简化模式，不推荐使用
- **authorization code**：授权码模式(oatuh2 最推荐使用一种授权模型)
- resource owner password credentials：密码模式(使用用户密码授权)
- client credentials：客户端模式

1 简化模式

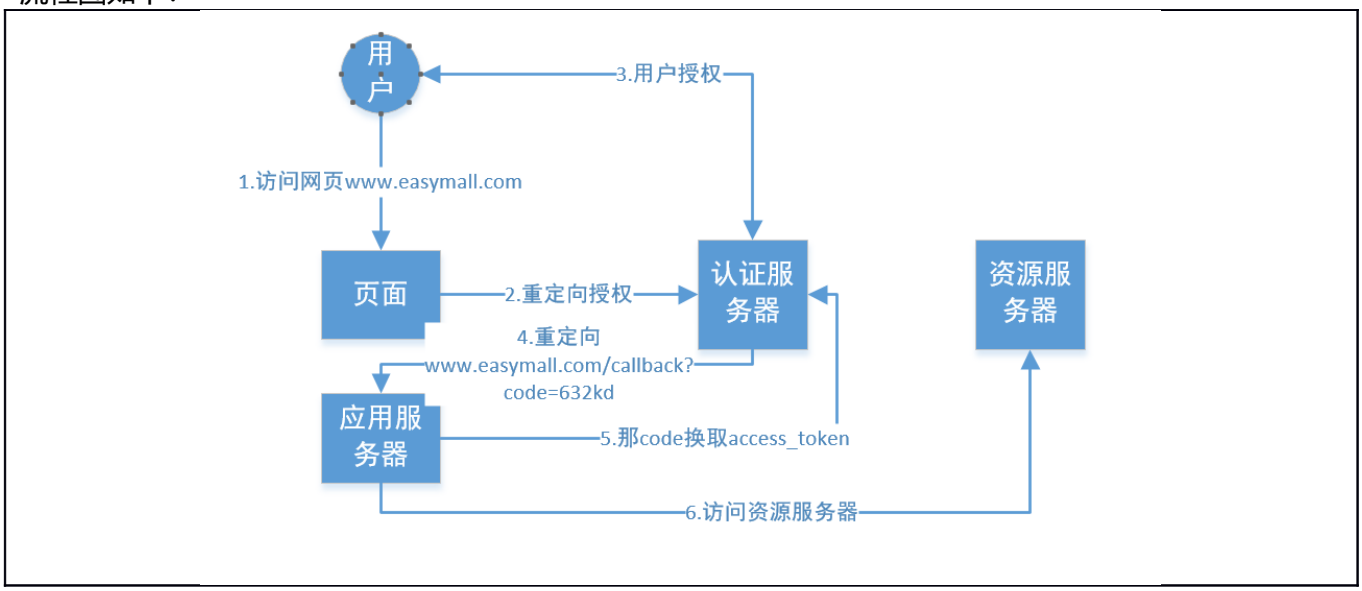
简化模式适用于纯静态页面应用。所谓纯静态页面应用，也就是应用没有在服务器上执行代码的权限（通常是把代码托管在别人的服务器上），只有前端 JS 代码的控制权。

这种场景下，应用是没有持久化存储的能力的。因此，按照 oAuth2.0 的规定，这种应用是拿不到 Refresh Token 的。



2 授权码模式

授权码模式适用于有自己的服务器的应用，它是一个一次性的临时凭证，用来换取 access_token 和 refresh_token。认证服务器提供了一个类似这样的接口：
http://shuxs.nat300.topwww.easymall.com/oauth/authorize?code=&client_id=&client_secret=
需要传入 code、client_id 以及 client_secret。验证通过后，返回 access_token 和 refresh_token。
一旦换取成功，code 立即作废，不能再使用第二次。
Code 为临时生成，client_id 和 client_secret 是注册时生成。
流程图如下：

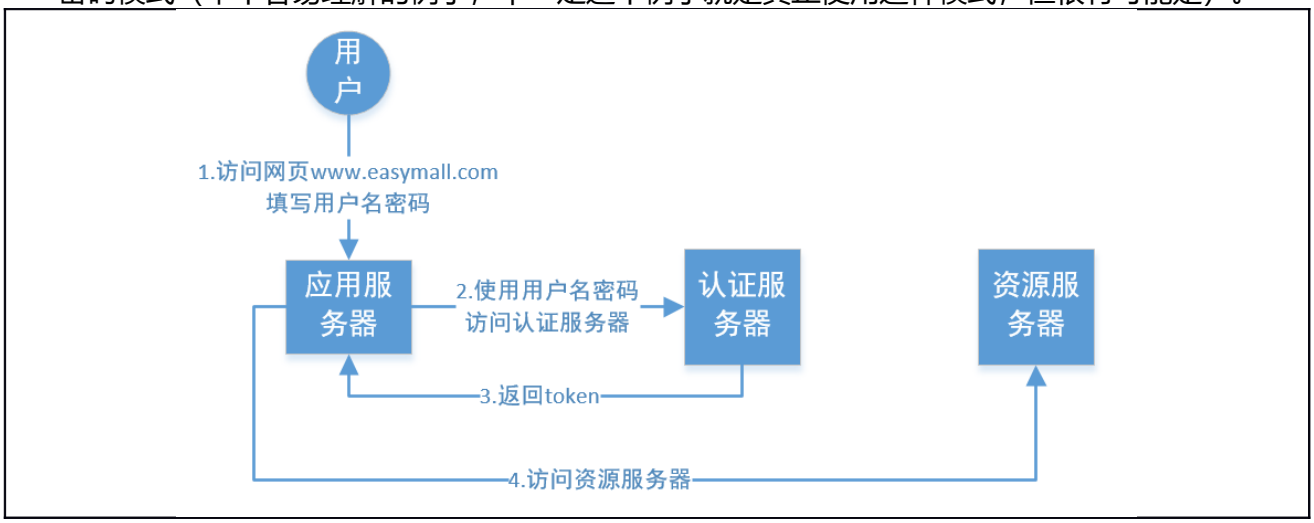


简单模式中直接将 access_token 返回页面直接就在页面看到 token 很不安全，这里使用一次性的 code，第一窃取已经被使用的 code，无效，第二窃取没使用的 code，同时要窃取 client_secret 才能使用 code 大大降低了可能性。

有了这个 code，token 的安全性大大提高。因此，oAuth2.0 鼓励使用这种方式进行授权，而简单模式则是在不得已情况下才会使用。

3 密码模式

oauth2 协议中，授权的流程是一样的，只是根据不同场景使用的授权模型不同，换句话说就是给用户看到的授权界面，特点，使用授权数据不同。所以也能支持用户输入用户名密码的形式授权。一般这种情况都是使用同个企业的产品。比如 qq 资源服务器的资源被 qq 游戏等产品访问就可以使用密码模式（举个容易理解的例子，不一定这个例子就是真正使用这种模式，但很有可能是）。



4 客户端授权

如果信任关系再进一步，或者调用者是一个后端的模块，没有用户界面的时候，可以使用客户端模式。认证授权服务器直接对客户端进行身份验证(仅仅使用 client_id client_secret)，验证通过后，返回 token。这里就没有用户操作的事了.

一般情况，用户密码，客户端授权都是 app，最广泛推荐的授权方式是 code 授权码模式。

五　授权服务

1　创建授权服务器

1.A　pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.shu</groupId>
    <artifactId>oauth2-test</artifactId>
    <version>1.0-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.13.RELEASE</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!--基于内存管理的-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-oauth2</artifactId>
            <exclusions>
                <exclusion>

<groupId>org.springframework.security.oauth.boot</groupId>
                    <artifactId>spring-security-oauth2-autoconfigure</artifactId>
                </exclusion>
            </exclusions>
            <version>2.1.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security.oauth.boot</groupId>
            <artifactId>spring-security-oauth2-autoconfigure</artifactId>
            <version>2.1.0.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

1.B 启动类

```
@SpringBootApplication
public class StarterOauth2 {
    public static void main(String[] args) {
        SpringApplication.run(StarterOauth2.class,args);
    }
}
```

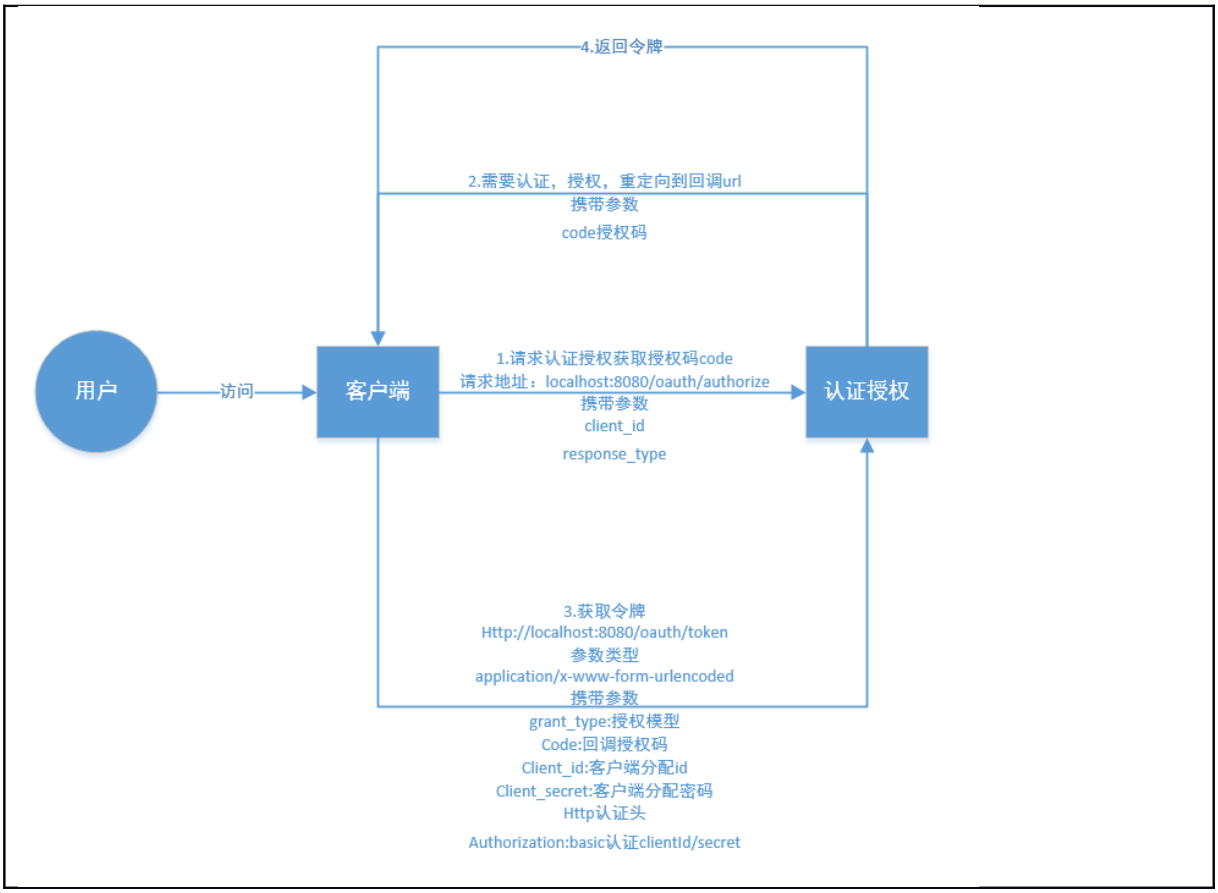
1.C Application.properties

```
server.port=9090
```

1.D 基于内存授权配置

在当前系统中，由于 security 的存在，所有请求都会先进行认证，才能访问，对于这部分我们暂时不做处理。由于依赖中存在 oauth2-autoconfigure，也会进行 oauth2 授权配置。这里包括管理的客户端信息，包括客户端 id 密码，授权模式等等。并且配合 security 实现拦截授权。可以通过实现配置类自定义配置一些逻辑。从案例中要了解-认证 授权 令牌的概念

1.D.a 授权模式的详细流程



上述流程基本上分为 4 步完成

- 第一步：客户端（本来应该是一个应用程序，案例中使用浏览器），携带参数表示客户端身份的 client_id 和 reponse_type 表示授权码 code 参数访问授权服务器
- 第二步：授权服务器要求客户端准许授权（浏览器直接显示授权页面给用户）。用户查看授权范围（授权服务器通知该授权的授权范围，读，写，删等）

- 第三步：用户授权后，授权服务器按照记录的回调地址，重定向到 url 携带一个参数 code。
- 重定向地址一般都是客户端可以接收处理的请求 url，所以客户端拿着 code，继续访问授权服务器获取 token 数据。

1.D.b MyWebSecurityConfig

security 配置

spring security 封装的 oauth2 授权逻辑必须基于认证之后实现.所以在本项目中以最简单的形式配置安全框架

```
@EnableWebSecurity
public class MyWebSecurityConfig extends WebSecurityConfigurerAdapter {

    /**
     * 用户密码加密方式
     * 不加密
     * @return
     */
    @Bean
    public PasswordEncoder myPasswordEncoder(){
        return NoOpPasswordEncoder.getInstance();
    }

    /**
     * 创建一个内存的 security 用户
     * 认证时使用
     * @param auth
     * @throws Exception
     */
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
                .withUser("admin")
                .password("123456")
                .authorities("all");
    }
}
```

1.D.c MyAuthorizationConfig

oauth 配置

```
@Configuration
@EnableAuthorizationServer
public class MyAuthorizationConfig extends
AuthorizationServerConfigurerAdapter {
    /**
     * 手动定义二一个内存的客户端对象
     * 实际中是客户端到系统申请
     * @param clients
     * @throws Exception
     * F71ON7
     */
```

```
    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.inMemory()
                //客户端 id
                .withClient("client")
                //客户端 secret
                .secret("secret")
                //认证类型
                .
authorizedGrantTypes("authorization_code","refresh_token")
                .scopes("app")
                //认证完成后重定向到某个网页并携带 code 参数
                .redirectUris("http://shuxs.nat300.top");
    }
}
```

在这里我们以内存的存储形式来记录当前授权端可以授权通过的客户端信息（实际项目中是通过申请的方式，例如微信公众号）.其中包括

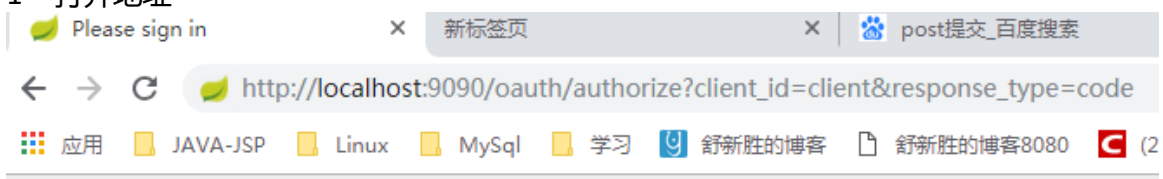
- clients 对象：控制授权服务器的客户端信息的对象.
- inMemory()：该方法表示授权端记录的客户端数据都在内存.
- withClient()：表示分配给客户端的 client_id,客户端访问时必须通过参数 client_id 携带该值
- secret()：表示分配给客户端的认证密码 client_secret,客户端获取 access_token 是必须携带该值,这里需要注意的是,如果我们使用加密的 PasswordEncoder 需要对其值加密处理,这里我们使用明文,所以没有处理
- authorizedGrantTypes()：表示该客户端可以使用的授权模式,可以一次给多个值（参数为可变参数）.
- redirectUris()：表示客户端认证授权成功后回调访问的 uri 地址.重定向到这个地址时,携带 code 授权码
- scopes()：自定义的值？表示该客户端的授权范围,资源服务器可以在验证 token 时查验该值,这样就可以控制客户端能访问那些资源

1.E 测试

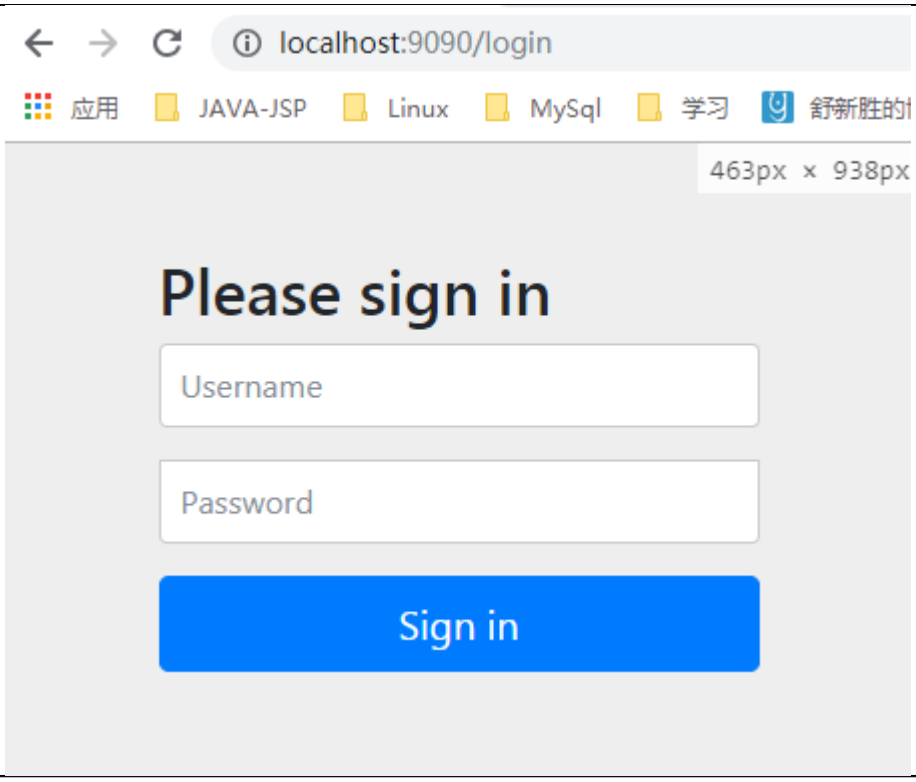
1.E.a 获取 code 授权码

参数：客户端 id，响应类型
http://localhost:9090/oauth/authorize?client_id=client&response_type=code
因为配置了 security，所以首次访问时需要登录

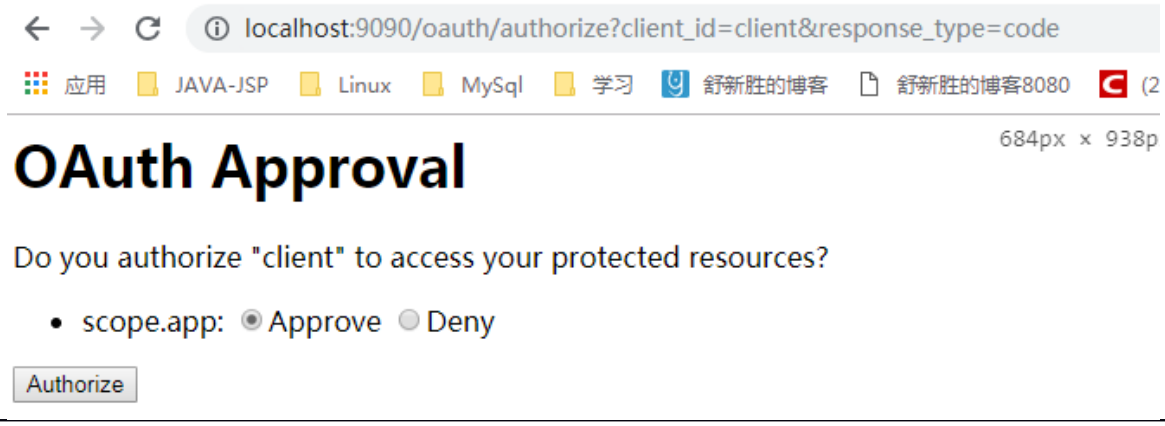
1 打开地址



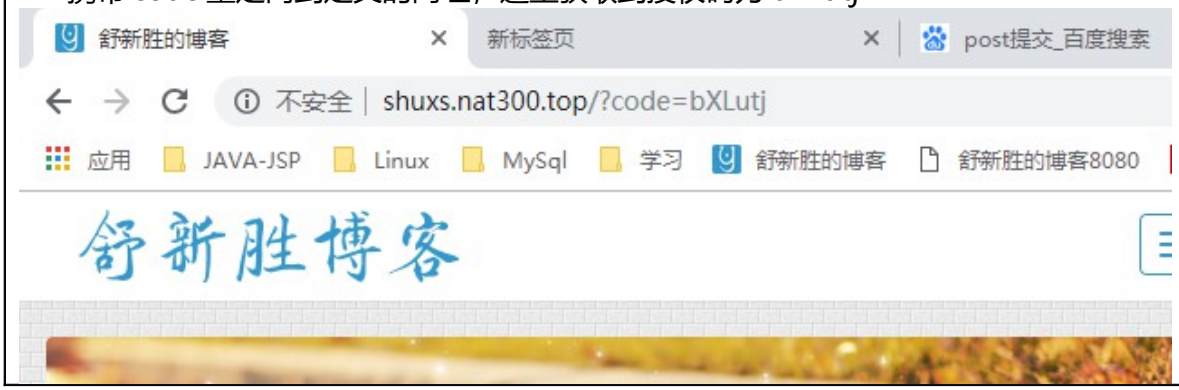
2 登录：admin，123456



3 认证成功后,跳出授权页面,点击 approve 授权允许,deny 拒绝授权



4 携带 code 重定向到定义的网址，这里获取到授权码为 bXLutj



1.E.b 获取 token

这里使用 postman 测试工具模拟 post 请求

拿到授权码之后,我们就可以访问另外一个地址实现 token 的获取,这里我们使用 postman 来提交 post 访问.

http://localhost:9090/oauth/token

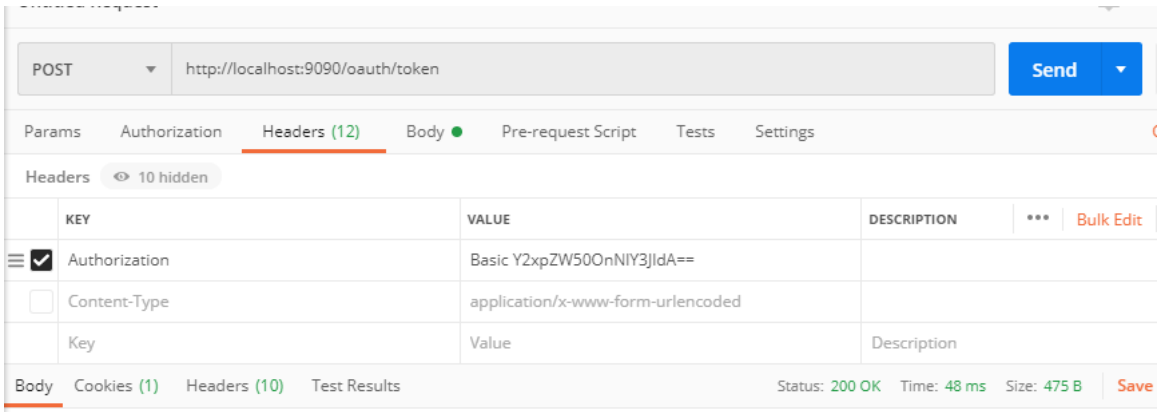
- 默认情况下,oauth2 支持 http 的 basic 认证,需要使用 base64 加密"客户端:授权密码"字符串

放到头 Authorization 中.

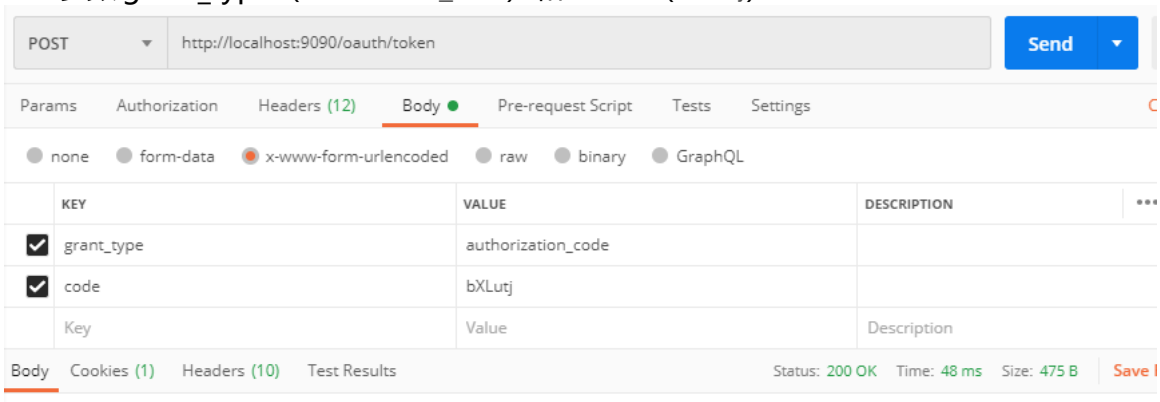
这里如果不知道 base64 加密结果,可以调用代码计算也可以到网络找在线加密器.

client:secret 明文加密后 添加 Basic 为:

Basic Y2xpZW50OnNlY3JldA==



- 填写地址和携带 body 参数,选择 body 传参,选择 x-www-form-urlencoded,传递 body 两个参数 grant_type（authorization_code）和 code （bXLutj）



- 点击发送获取结果