**Spring Security OAuth2.0认证授权**

## **1.基本概念**

### **1.1.什么是认证**

进入移动互联网时代，大家每天都在刷手机，常用的软件有微信、支付宝、头条等，下边拿微信来举例子说明认证相关的基本概念，在初次使用微信前需要注册成为微信用户，然后输入账号和密码即可登录微信，输入账号和密码登录微信的过程就是认证。

系统为什么要认证？

认证是为了保护系统的隐私数据与资源，用户的身份合法方可访问该系统的资源。

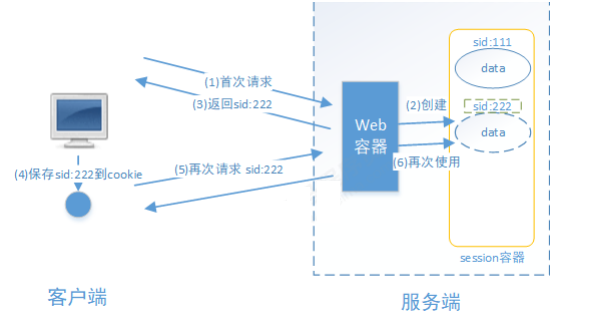
**认证** ：用户认证就是判断一个用户的身份是否合法的过程，用户去访问系统资源时系统要求验证用户的身份信息，身份合法方可继续访问，不合法则拒绝访问。常见的用户身份认证方式有：用户名密码登录，二维码登录，手机短信登录，指纹认证等方式。

### **1.2 什么是会话**

用户认证通过后，为了避免用户的每次操作都进行认证可将用户的信息保证在会话中。会话就是系统为了保持当前用户的登录状态所提供的机制，常见的有基于session方式、基于token方式等。

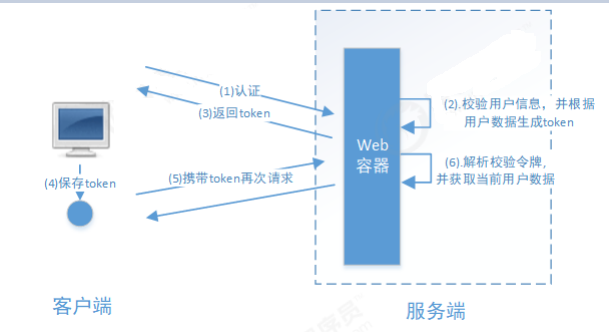
基于session的认证方式如下图：

它的交互流程是，用户认证成功后，在服务端生成用户相关的数据保存在session(当前会话)中，发给客户端的sesssion\_id 存放到 cookie 中，这样用户客户端请求时带上 session\_id 就可以验证服务器端是否存在 session 数据，以此完成用户的合法校验，当用户退出系统或session过期销毁时,客户端的session\_id也就无效了。



基于token方式如下图：

它的交互流程是，用户认证成功后，服务端生成一个token发给客户端，客户端可以放到 cookie 或 localStorage等存储中，每次请求时带上 token，服务端收到token通过验证后即可确认用户身份。



基于session的认证方式由Servlet规范定制，服务端要存储session信息需要占用内存资源，客户端需要支持cookie；基于token的方式则一般不需要服务端存储token，并且不限制客户端的存储方式。如今移动互联网时代更多类型的客户端需要接入系统，系统多是采用前后端分离的架构进行实现，所以基于token的方式更适合。

### **1.3 什么是授权**

还拿微信来举例子，微信登录成功后用户即可使用微信的功能，比如，发红包、发朋友圈、添加好友等，没有绑定银行卡的用户是无法发送红包的，绑定银行卡的用户才可以发红包，发红包功能、发朋友圈功能都是微信的资源即功能资源，用户拥有发红包功能的权限才可以正常使用发送红包功能，拥有发朋友圈功能的权限才可以使用发朋友圈功能，这个根据用户的权限来控制用户使用资源的过程就是授权。

为什么要授权？

认证是为了保证用户身份的合法性，授权则是为了更细粒度的对隐私数据进行划分，授权是在认证通过后发生的，控制不同的用户能够访问不同的资源。

授权： 授权是用户认证通过根据用户的权限来控制用户访问资源的过程，拥有资源的访问权限则正常访问，没有权限则拒绝访问。

### **1.4 授权的数据模型**

如何进行授权即如何对用户访问资源进行控制，首先需要学习授权相关的数据模型。

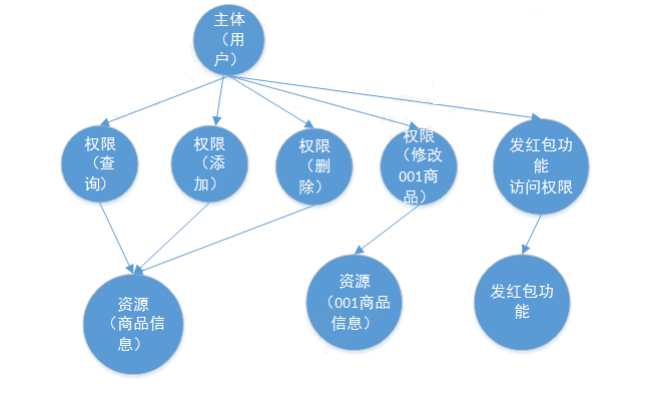
授权可简单理解为Who对What(which)进行How操作，包括如下：

Who，即主体（Subject），主体一般是指用户，也可以是程序，需要访问系统中的资源。

What，即资源（Resource），如系统菜单、页面、按钮、代码方法、系统商品信息、系统订单信息等。系统菜单、页面、按钮、代码方法都属于系统功能资源，对于web系统每个功能资源通常对应一个URL；系统商品信息、系统订单信息都属于实体资源（数据资源），实体资源由资源类型和资源实例组成，比如商品信息为资源类型，商品编号 为001的商品为资源实例。

How，权限/许可（Permission），规定了用户对资源的操作许可，权限离开资源没有意义，如用户查询权限、用户添加权限、某个代码方法的调用权限、编号为001的用户的修改权限等，通过权限可知用户对哪些资源都有哪些操作许可。

主体、资源、权限关系如下图：



主体、资源、权限相关的数据模型如下：

主体（用户id、账号、密码、...）

资源（资源id、资源名称、访问地址、...）

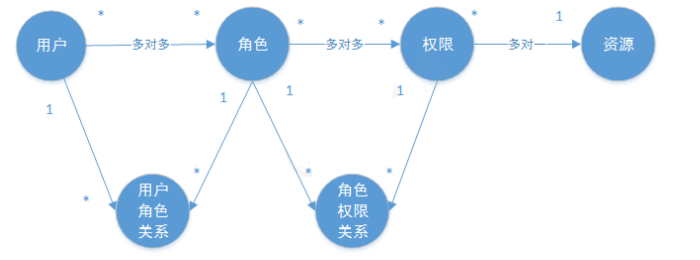
权限（权限id、权限标识、权限名称、资源id、...）

角色（角色id、角色名称、...）

角色和权限关系（角色id、权限id、...）

主体（用户）和角色关系（用户id、角色id、...）

主体（用户）、资源、权限关系如下图：



**通常企业开发中将资源和权限表合并为一张权限表，如下**：

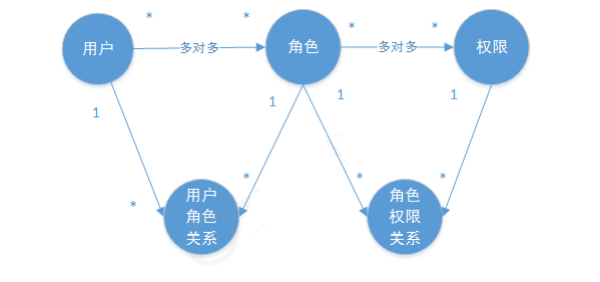
资源（资源id、资源名称、访问地址、...）

权限（权限id、权限标识、权限名称、资源id、...）

合并为：

权限（权限id、权限标识、权限名称、资源名称、资源访问地址、...）

修改后数据模型之间的关系如下图：

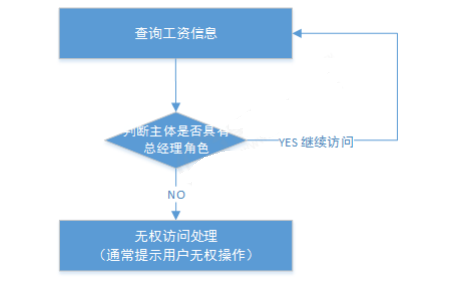


### **1.5 RBAC**

如何实现授权？业界通常基于RBAC实现授权。

#### **1.5.1 基于角色的访问控制**

RBAC基于角色的访问控制（Role-Based Access Control）是按角色进行授权，比如：主体的角色为总经理可以查询企业运营报表，查询员工工资信息等，访问控制流程如下：



根据上图中的判断逻辑，授权代码可表示如下：

if(主体.hasRole("总经理角色id")){  
 查询工资  
}

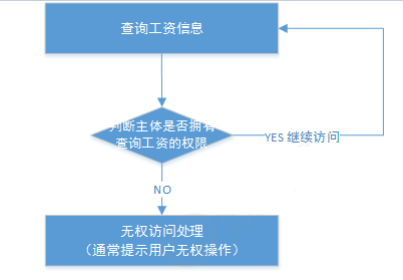
如果上图中查询工资所需要的角色变化为总经理和部门经理，此时就需要修改判断逻辑为“判断用户的角色是否是总经理或部门经理”，修改代码如下：

if(主体.hasRole("总经理角色id") || 主体.hasRole("部门经理角色id")){  
 查询工资  
}

根据上边的例子发现，当需要修改角色的权限时就需要修改授权的相关代码，系统可扩展性差。

#### **1.5.2 基于资源的访问控制**

RBAC基于资源的访问控制（Resource-Based Access Control）是按资源（或权限）进行授权，比如：用户必须具有查询工资权限才可以查询员工工资信息等，访问控制流程如下：



根据上图中的判断，授权代码可以表示为：

if(主体.hasPermission("查询工资权限标识")){   
 查询工资   
}

优点：系统设计时定义好查询工资的权限标识，即使查询工资所需要的角色变化为总经理和部门经理也不需要修改授权代码，系统可扩展性强。

## **2.Spring Security快速上手**

### **2.1 Spring Security介绍**

Spring Security是一个能够为基于Spring的企业应用系统提供声明式的安全访问控制解决方案的安全框架。由于它是Spring生态系统中的一员，因此它伴随着整个Spring生态系统不断修正、升级，在spring boot项目中加入springsecurity更是十分简单，使用Spring Security 减少了为企业系统安全控制编写大量重复代码的工作。

### **2.2 集成SpringBoot**

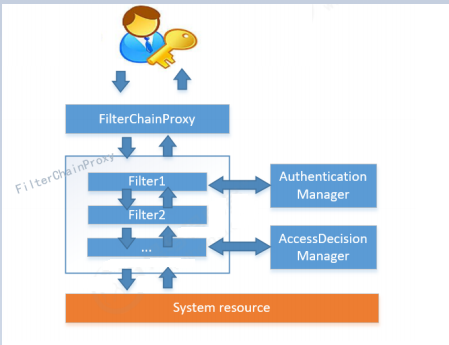
Spring Boot是一套Spring的快速开发框架，基于Spring 4.0设计，使用Spring Boot开发可以避免一些繁琐的工程搭建和配置，同时它集成了大量的常用框架，快速导入依赖包，避免依赖包的冲突。基本上常用的开发框架都支持Spring Boot开发，例如：MyBatis、Dubbo等，Spring 家族更是如此，例如：Spring cloud、Spring mvc、Spring security等，使用Spring Boot开发可以大大得高生产率，所以Spring Boot的使用率非常高。

### **2.3 工作原理**

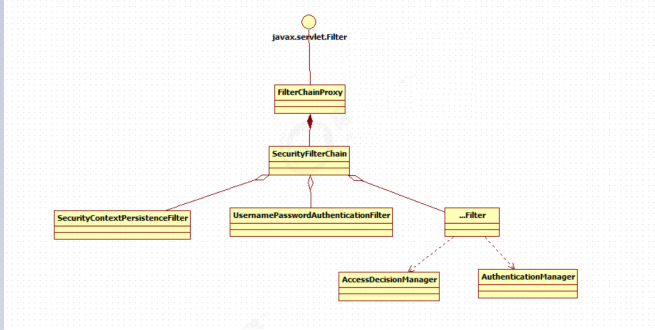
#### **2.3.1结构总览**

Spring Security所解决的问题就是安全访问控制，而安全访问控制功能其实就是对所有进入系统的请求进行拦截，校验每个请求是否能够访问它所期望的资源。根据前边知识的学习，可以通过Filter或AOP等技术来实现，SpringSecurity对Web资源的保护是靠Filter实现的，所以从这个Filter来入手，逐步深入Spring Security原理。

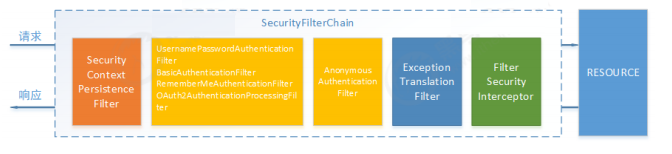
当初始化Spring Security时，会创建一个名为 SpringSecurityFilterChain 的Servlet过滤器，类型为org.springframework.security.web.FilterChainProxy，它实现了javax.servlet.Filter，因此外部的请求会经过此类，下图是Spring Security过虑器链结构图：



FilterChainProxy是一个代理，真正起作用的是FilterChainProxy中SecurityFilterChain所包含的各个Filter，同时这些Filter作为Bean被Spring管理，它们是Spring Security核心，各有各的职责，但他们并不直接处理用户的认证，也不直接处理用户的授权，而是把它们交给了认证管理器（AuthenticationManager）和决策管理器（AccessDecisionManager）进行处理，下图是FilterChainProxy相关类的UML图示。



spring Security功能的实现主要是由一系列过滤器链相互配合完成。



下面介绍过滤器链中主要的几个过滤器及其作用：

SecurityContextPersistenceFilter 这个Filter是整个拦截过程的入口和出口（也就是第一个和最后一个拦截器），会在请求开始时从配置好的 SecurityContextRepository 中获取 SecurityContext，然后把它设置给SecurityContextHolder。在请求完成后将 SecurityContextHolder 持有的 SecurityContext 再保存到配置好的 SecurityContextRepository，同时清除 securityContextHolder 所持有的 SecurityContext；

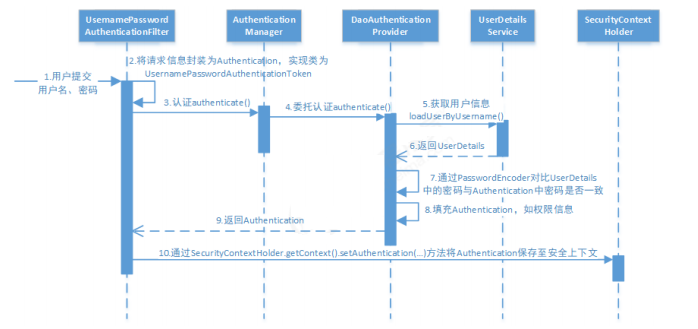
UsernamePasswordAuthenticationFilter 用于处理来自表单提交的认证。该表单必须提供对应的用户名和密码，其内部还有登录成功或失败后进行处理的 AuthenticationSuccessHandler 和

AuthenticationFailureHandler，这些都可以根据需求做相关改变；

FilterSecurityInterceptor 是用于保护web资源的，使用AccessDecisionManager对当前用户进行授权访问，前面已经详细介绍过了；

ExceptionTranslationFilter 能够捕获来自 FilterChain 所有的异常，并进行处理。但是它只会处理两类异常：AuthenticationException 和 AccessDeniedException，其它的异常它会继续抛出。

#### **2.3.2 认证流程**



让我们仔细分析认证过程：

1. 用户提交用户名、密码被SecurityFilterChain中的 UsernamePasswordAuthenticationFilter 过滤器获取到，封装为请求Authentication，通常情况下是UsernamePasswordAuthenticationToken这个实现类。

2. 然后过滤器将Authentication提交至认证管理器（AuthenticationManager）进行认证

3. 认证成功后， AuthenticationManager 身份管理器返回一个被填充满了信息的（包括上面提到的权限信息，身份信息，细节信息，但密码通常会被移除） Authentication 实例。

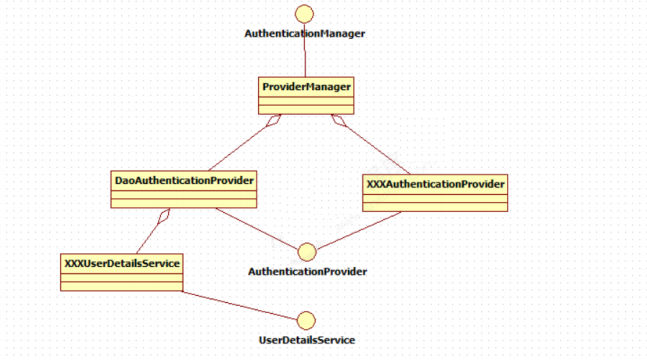
4. SecurityContextHolder 安全上下文容器将第3步填充了信息的 Authentication ，通过

SecurityContextHolder.getContext().setAuthentication(…)方法，设置到其中。 可以看出AuthenticationManager接口（认证管理器）是认证相关的核心接口，也是发起认证的出发点，它的实现类为ProviderManager。而Spring Security支持多种认证方式，因此ProviderManager维护着一个List<AuthenticationProvider> 列表，存放多种认证方式，最终实际的认证工作是由

AuthenticationProvider完成的。咱们知道web表单的对应的AuthenticationProvider实现类为

DaoAuthenticationProvider，它的内部又维护着一个UserDetailsService负责UserDetails的获取。最终AuthenticationProvider将UserDetails填充至Authentication。

认证核心组件的大体关系如下：



**2.3.2.1 AuthenticationProvider**

通过前面的Spring Security认证流程我们得知，认证管理器（AuthenticationManager）委托

AuthenticationProvider完成认证工作。

AuthenticationProvider是一个接口，定义如下：

public interface AuthenticationProvider {

Authentication authenticate(Authentication authentication) throws AuthenticationException;

boolean supports(Class<?> var1);

}

authenticate()方法定义了认证的实现过程，它的参数是一个Authentication，里面包含了登录用户所提交的用户、密码等。而返回值也是一个Authentication，这个Authentication则是在认证成功后，将用户的权限及其他信息重新组装后生成。

Spring Security中维护着一个 List<AuthenticationProvider> 列表，存放多种认证方式，不同的认证方式使用不同的AuthenticationProvider。如使用用户名密码登录时，使用AuthenticationProvider1，短信登录时使用AuthenticationProvider2等等这样的例子很多。

每个AuthenticationProvider需要实现supports（）方法来表明自己支持的认证方式，如我们使用表单方式认证，在提交请求时Spring Security会生成UsernamePasswordAuthenticationToken，它是一个Authentication，里面封装着用户提交的用户名、密码信息。而对应的，哪个AuthenticationProvider来处理它？

我们在DaoAuthenticationProvider的基类AbstractUserDetailsAuthenticationProvider发现以下代码：

public boolean supports(Class<?> authentication) {

return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);

}

**也就是说当web表单提交用户名密码时，Spring Security由DaoAuthenticationProvider处理。**

最后，我们来看一下Authentication(认证信息)的结构，它是一个接口，我们之前提到的

UsernamePasswordAuthenticationToken就是它的实现之一：

public interface Authentication extends Principal, Serializable { （1）

Collection<? extends GrantedAuthority> getAuthorities(); （2）

Object getCredentials(); （3）

Object getDetails(); （4）

Object getPrincipal(); （5）

boolean isAuthenticated();

void setAuthenticated(boolean var1) throws IllegalArgumentException;

}

（1）Authentication是spring security包中的接口，直接继承自Principal类，而Principal是位于 java.security包中的。它是表示着一个抽象主体身份，任何主体都有一个名称，因此包含一个getName()方法。

（2）getAuthorities()，权限信息列表，默认是GrantedAuthority接口的一些实现类，通常是代表权限信息的一系列字符串。

（3）getCredentials()，凭证信息，用户输入的密码字符串，在认证过后通常会被移除，用于保障安全。

（4）getDetails()，细节信息，web应用中的实现接口通常为 WebAuthenticationDetails，它记录了访问者的ip地址和sessionId的值。

（5）getPrincipal()，身份信息，大部分情况下返回的是UserDetails接口的实现类，UserDetails代表用户的详细信息，那从Authentication中取出来的UserDetails就是当前登录用户信息，它也是框架中的常用接口之一。

**2.3.2.2 UserDetailsService**

1）认识UserDetailsService

现在咱们现在知道DaoAuthenticationProvider处理了web表单的认证逻辑，认证成功后既得到一个Authentication(UsernamePasswordAuthenticationToken实现)，里面包含了身份信息（Principal）。这个身份信息就是一个 Object ，大多数情况下它可以被强转为UserDetails对象。

DaoAuthenticationProvider中包含了一个UserDetailsService实例，它负责根据用户名提取用户信息UserDetails(包含密码)，而后DaoAuthenticationProvider会去对比UserDetailsService提取的用户密码与用户提交的密码是否匹配作为认证成功的关键依据，因此可以通过将自定义的 UserDetailsService 公开为spring bean来定义自定义身份验证。

public interface UserDetailsService {

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}

很多人把DaoAuthenticationProvider和UserDetailsService的职责搞混淆，其实UserDetailsService只负责从特定的地方（通常是数据库）加载用户信息，仅此而已。而DaoAuthenticationProvider的职责更大，它完成完整的认证流程，同时会把UserDetails填充至Authentication。

上面一直提到UserDetails是用户信息，咱们看一下它的真面目：

public interface UserDetails extends Serializable {

Collection<? extends GrantedAuthority> getAuthorities();

String getPassword();

String getUsername();

boolean isAccountNonExpired();

boolean isAccountNonLocked();

boolean isCredentialsNonExpired();

boolean isEnabled();

}

它和Authentication接口很类似，比如它们都拥有username，authorities。Authentication的getCredentials()与UserDetails中的getPassword()需要被区分对待，前者是用户提交的密码凭证，后者是用户实际存储的密码，认证其实就是对这两者的比对。Authentication中的getAuthorities()实际是由UserDetails的getAuthorities()传递而形成的。还记得Authentication接口中的getDetails()方法吗？其中的UserDetails用户详细信息便是经过了AuthenticationProvider认证之后被填充的。

通过实现UserDetailsService和UserDetails，我们可以完成对用户信息获取方式以及用户信息字段的扩展。

Spring Security提供的InMemoryUserDetailsManager(内存认证)，JdbcUserDetailsManager(jdbc认证)就是UserDetailsService的实现类，主要区别无非就是从内存还是从数据库加载用户。

**2.3.2.3 PasswordEncoder**

1）认识PasswordEncoder

DaoAuthenticationProvider认证处理器通过UserDetailsService获取到UserDetails后，它是如何与请求Authentication中的密码做对比呢？

在这里Spring Security为了适应多种多样的加密类型，又做了抽象，DaoAuthenticationProvider通过PasswordEncoder接口的matches方法进行密码的对比，而具体的密码对比细节取决于实现：

public interface PasswordEncoder {

String encode(CharSequence var1);

boolean matches(CharSequence var1, String var2);

default boolean upgradeEncoding(String encodedPassword) {

return false;

}

}

而Spring Security提供很多内置的PasswordEncoder，能够开箱即用，使用某种PasswordEncoder只需要进行如下声明即可，如下：

@Bean

public PasswordEncoder passwordEncoder() {

return NoOpPasswordEncoder.getInstance();

}

NoOpPasswordEncoder采用字符串匹配方法，不对密码进行加密比较处理，密码比较流程如下：

1、用户输入密码（明文 ）

2、DaoAuthenticationProvider获取UserDetails（其中存储了用户的正确密码）

3、DaoAuthenticationProvider使用PasswordEncoder对输入的密码和正确的密码进行校验，密码一致则校验通过，否则校验失败。

NoOpPasswordEncoder的校验规则拿输入的密码和UserDetails中的正确密码进行字符串比较，字符串内容一致则校验通过，否则 校验失败。

实际项目中推荐使用BCryptPasswordEncoder, Pbkdf2PasswordEncoder, SCryptPasswordEncoder等，感兴趣的大家可以看看这些PasswordEncoder的具体实现。

2）使用BCryptPasswordEncoder

1、配置BCryptPasswordEncoder

在安全配置类中定义：

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

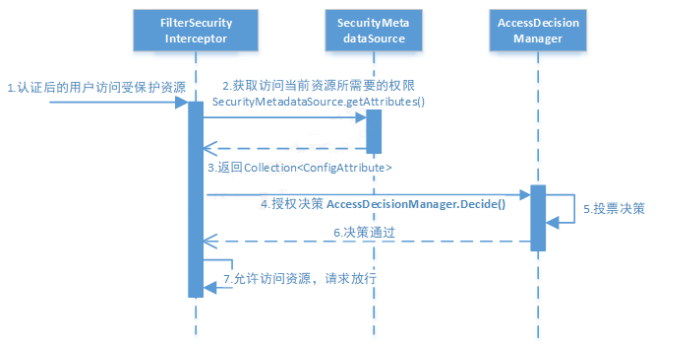
}

### **2.4 授权流程**

#### **2.4.1 授权流程**

通过快速上手我们知道，Spring Security可以通过 http.authorizeRequests() 对web请求进行授权保护。SpringSecurity使用标准Filter建立了对web请求的拦截，最终实现对资源的授权访问。

Spring Security的授权流程如下：



分析授权流程：

1. **拦截请求**，已认证用户访问受保护的web资源将被SecurityFilterChain中的 FilterSecurityInterceptor 的子类拦截。

2. **获取资源访问策略**，FilterSecurityInterceptor会从 SecurityMetadataSource 的子类

DefaultFilterInvocationSecurityMetadataSource 获取要访问当前资源所需要的权限

Collection<ConfigAttribute> 。

SecurityMetadataSource其实就是读取访问策略的抽象，而读取的内容，其实就是我们配置的访问规则，读取访问策略如：

http

.authorizeRequests()

.antMatchers("/r/r1").hasAuthority("p1")

.antMatchers("/r/r2").hasAuthority("p2")

...

3. 最后，FilterSecurityInterceptor会调用 AccessDecisionManager 进行授权决策，若决策通过，则允许访问资源，否则将禁止访问。

AccessDecisionManager（访问决策管理器）的核心接口如下:

public interface AccessDecisionManager {

/\*\*

\* 通过传递的参数来决定用户是否有访问对应受保护资源的权限

\*/

void decide(Authentication authentication , Object object, Collection<ConfigAttribute>

configAttributes ) throws AccessDeniedException, InsufficientAuthenticationException;

//略..

}

这里着重说明一下decide的参数：

authentication：要访问资源的访问者的身份

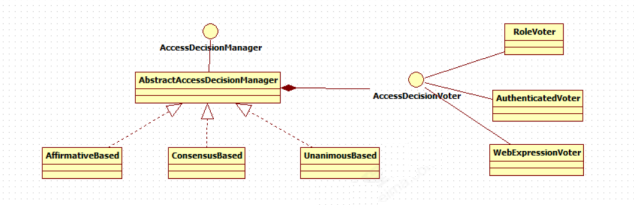
object：要访问的受保护资源，web请求对应FilterInvocation

confifigAttributes：是受保护资源的访问策略，通过SecurityMetadataSource获取。

**decide接口就是用来鉴定当前用户是否有访问对应受保护资源的权限**。

#### **2.4.2 授权决策**

AccessDecisionManager采用投票的方式来确定是否能够访问受保护资源。



通过上图可以看出，AccessDecisionManager中包含的一系列AccessDecisionVoter将会被用来对Authentication是否有权访问受保护对象进行投票，AccessDecisionManager根据投票结果，做出最终决策。

AccessDecisionVoter是一个接口，其中定义有三个方法，具体结构如下所示。

public interface AccessDecisionVoter<S> {

int ACCESS\_GRANTED = 1;

int ACCESS\_ABSTAIN = 0;

int ACCESS\_DENIED = ‐1;

boolean supports(ConfigAttribute var1);

boolean supports(Class<?> var1);

int vote(Authentication var1, S var2, Collection<ConfigAttribute> var3);

}

vote()方法的返回结果会是AccessDecisionVoter中定义的三个常量之一。ACCESS\_GRANTED表示同意，ACCESS\_DENIED表示拒绝，ACCESS\_ABSTAIN表示弃权。如果一个AccessDecisionVoter不能判定当前Authentication是否拥有访问对应受保护对象的权限，则其vote()方法的返回值应当为弃权ACCESS\_ABSTAIN。

Spring Security内置了三个基于投票的AccessDecisionManager实现类如下，它们分别是

**AffiffiffirmativeBased**、**ConsensusBased**和**UnanimousBased**。

**AffiffiffirmativeBased**的逻辑是：

（1）只要有AccessDecisionVoter的投票为ACCESS\_GRANTED则同意用户进行访问；

（2）如果全部弃权也表示通过；

（3）如果没有一个人投赞成票，但是有人投反对票，则将抛出AccessDeniedException。

Spring security默认使用的是AffiffiffirmativeBased。

**ConsensusBased**的逻辑是：

（1）如果赞成票多于反对票则表示通过。

（2）反过来，如果反对票多于赞成票则将抛出AccessDeniedException。

（3）如果赞成票与反对票相同且不等于0，并且属性allowIfEqualGrantedDeniedDecisions的值为true，则表示通过，否则将抛出异常AccessDeniedException。参数allowIfEqualGrantedDeniedDecisions的值默认为true。

（4）如果所有的AccessDecisionVoter都弃权了，则将视参数allowIfAllAbstainDecisions的值而定，如果该值为true则表示通过，否则将抛出异常AccessDeniedException。参数allowIfAllAbstainDecisions的值默认为false。

**UnanimousBased**的逻辑与另外两种实现有点不一样，另外两种会一次性把受保护对象的配置属性全部传递给AccessDecisionVoter进行投票，而UnanimousBased会一次只传递一个ConfifigAttribute给AccessDecisionVoter进行投票。这也就意味着如果我们的AccessDecisionVoter的逻辑是只要传递进来的ConfifigAttribute中有一个能够匹配则投赞成票，但是放到UnanimousBased中其投票结果就不一定是赞成了。

UnanimousBased的逻辑具体来说是这样的：

（1）如果受保护对象配置的某一个ConfifigAttribute被任意的AccessDecisionVoter反对了，则将抛出AccessDeniedException。

（2）如果没有反对票，但是有赞成票，则表示通过。

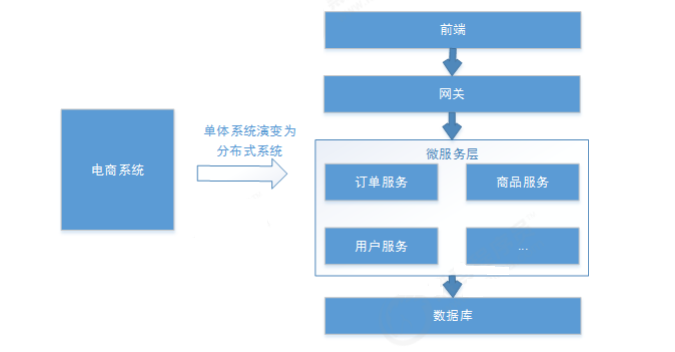
（3）如果全部弃权了，则将视参数allowIfAllAbstainDecisions的值而定，true则通过，false则抛出AccessDeniedException。

Spring Security也内置一些投票者实现类如RoleVoter、AuthenticatedVoter和WebExpressionVoter等，可以自行查阅资料进行学习。

## **3 分布式系统认证方案**

### **3.1 什么是分布式系统**

随着软件环境和需求的变化 ，软件的架构由单体结构演变为分布式架构，具有分布式架构的系统叫分布式系统，分布式系统的运行通常依赖网络，它将单体结构的系统分为若干服务，服务之间通过网络交互来完成用户的业务处理，当前流行的微服务架构就是分布式系统架构，如下图：



分布式系统具体如下基本特点：

1、分布性：每个部分都可以独立部署，服务之间交互通过网络进行通信，比如：订单服务、商品服务。

2、伸缩性：每个部分都可以集群方式部署，并可针对部分结点进行硬件及软件扩容，具有一定的伸缩能力。

3、共享性：每个部分都可以作为共享资源对外提供服务，多个部分可能有操作共享资源的情况。

4、开放性：每个部分根据需求都可以对外发布共享资源的访问接口，并可允许第三方系统访问。

### **3.2 分布式认证需求**

分布式系统的每个服务都会有认证、授权的需求，如果每个服务都实现一套认证授权逻辑会非常冗余，考虑分布式系统共享性的特点，需要由独立的认证服务处理系统认证授权的请求；考虑分布式系统开放性的特点，不仅对系统内部服务提供认证，对第三方系统也要提供认证。分布式认证的需求总结如下：

**统一认证授权**

提供独立的认证服务，统一处理认证授权。

无论是不同类型的用户，还是不同种类的客户端(web端，H5、APP)，均采用一致的认证、权限、会话机制，实现统一认证授权。

要实现统一则认证方式必须可扩展，支持各种认证需求，比如：用户名密码认证、短信验证码、二维码、人脸识别等认证方式，并可以非常灵活的切换。

**应用接入认证**

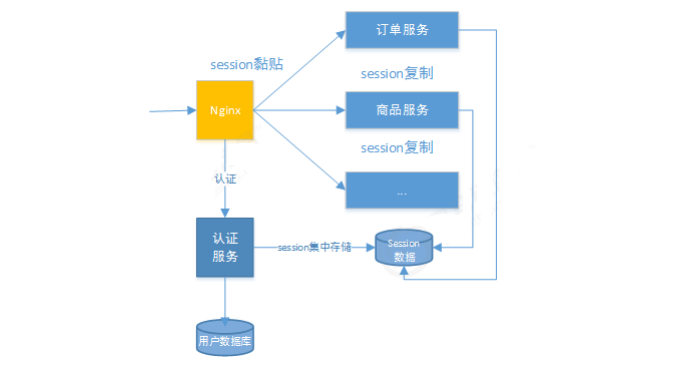
应提供扩展和开放能力，提供安全的系统对接机制，并可开放部分API给接入第三方使用，一方应用（内部 系统服务）和三方应用（第三方应用）均采用统一机制接入。

**3.3 分布式认证方案**

#### **3.3.1 选型分析**

1、基于session的认证方式

在分布式的环境下，基于session的认证会出现一个问题，每个应用服务都需要在session中存储用户身份信息，通过负载均衡将本地的请求分配到另一个应用服务需要将session信息带过去，否则会重新认证。



这个时候，通常的做法有下面几种：

**Session复制**：多台应用服务器之间同步session，使session保持一致，对外透明。

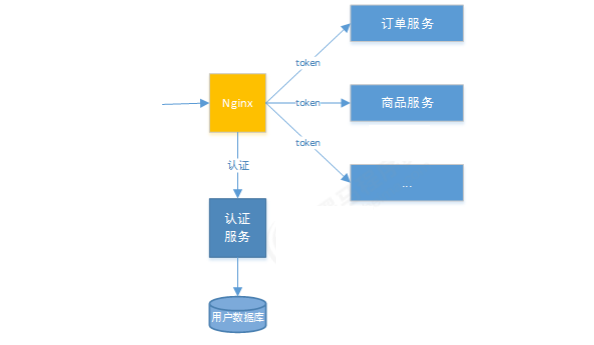
**Session黏贴**：当用户访问集群中某台服务器后，强制指定后续所有请求均落到此机器上。

**Session集中存储**：将Session存入分布式缓存中，所有服务器应用实例统一从分布式缓存中存取Session。

总体来讲，基于session认证的认证方式，可以更好的在服务端对会话进行控制，且安全性较高。但是，session机制方式基于cookie，在复杂多样的移动客户端上不能有效的使用，并且无法跨域，另外随着系统的扩展需提高session的复制、黏贴及存储的容错性。

2、基于token的认证方式

基于token的认证方式，服务端不用存储认证数据，易维护扩展性强， 客户端可以把token 存在任意地方，并且可以实现web和app统一认证机制。其缺点也很明显，token由于自包含信息，因此一般数据量较大，而且每次请求都需要传递，因此比较占带宽。另外，token的签名验签操作也会给cpu带来额外的处理负担。



#### **3.3.2 技术方案**

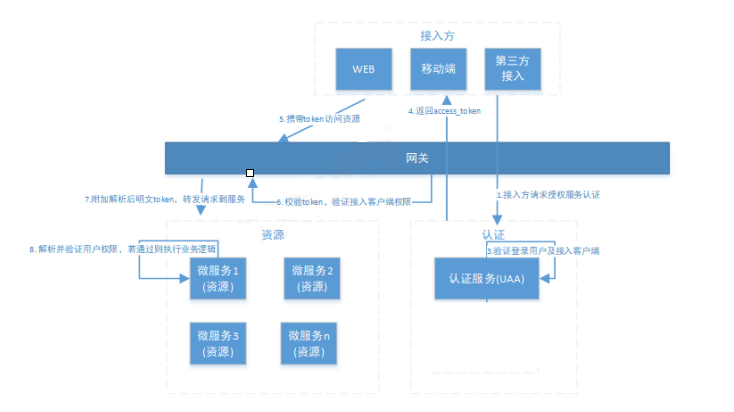
根据 选型的分析，**建议采用基于token的认证方式**，它的优点是：

1、适合统一认证的机制，客户端、一方应用、三方应用都遵循一致的认证机制。

2、token认证方式对第三方应用接入更适合，因为它更开放，可使用当前有流行的开放协议Oauth2.0、JWT等。

3、一般情况服务端无需存储会话信息，减轻了服务端的压力。

分布式系统认证技术方案见下图：



流程描述：

（1）用户通过接入方（应用）登录，接入方采取OAuth2.0方式在统一认证服务(UAA)中认证。

（2）认证服务(UAA)调用验证该用户的身份是否合法，并获取用户权限信息。

（3）认证服务(UAA)获取接入方权限信息，并验证接入方是否合法。

（4）若登录用户以及接入方都合法，认证服务生成jwt令牌返回给接入方，其中jwt中包含了用户权限及接入方权限。

（5）后续，接入方携带jwt令牌对API网关内的微服务资源进行访问。

（6）API网关对令牌解析、并验证接入方的权限是否能够访问本次请求的微服务。

（7）如果接入方的权限没问题，API网关将原请求header中附加解析后的明文Token，并将请求转发至微服务。

（8）微服务收到请求，明文token中包含登录用户的身份和权限信息。因此后续微服务自己可以干两件事：1，用户授权拦截（看当前用户是否有权访问该资源）2，将用户信息存储进当前线程上下文（有利于后续业务逻辑随时获取当前用户信息）

流程所涉及到UAA服务、API网关这三个组件职责如下：

**1）统一认证服务(UAA)**

它承载了OAuth2.0接入方认证、登入用户的认证、授权以及生成令牌的职责，完成实际的用户认证、授权功能。

**2）API网关**

作为系统的唯一入口，API网关为接入方提供定制的API集合，它可能还具有其它职责，如身份验证、监控、负载均衡、缓存等。API网关方式的核心要点是，所有的接入方和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。

## **4 OAuth2.0**

### **4.1 OAuth2.0介绍**

OAuth（开放授权）是一个开放标准，允许用户授权第三方应用访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方应用或分享他们数据的所有内容。OAuth2.0是OAuth协议的延续版本，但不向后兼容OAuth 1.0即完全废止了OAuth1.0。很多大公司如Google，Yahoo，Microsoft等都提供了OAUTH认证服务，这些都足以说明OAUTH标准逐渐成为开放资源授权的标准。

Oauth协议目前发展到2.0版本，1.0版本过于复杂，2.0版本已得到广泛应用。

参考：https://baike.baidu.com/item/oAuth/7153134?fr=aladdin

Oauth协议：https://tools.ietf.org/html/rfc6749

下边分析一个Oauth2认证的例子，通过例子去理解OAuth2.0协议的认证流程，本例子是知识星球网站使用微信认证的过程，这个过程的简要描述如下：

用户借助微信认证登录知识星球网站，用户就不用单独在知识星球网站注册用户，怎么样算认证成功吗？知识星球网站需要成功从微信获取用户的身份信息则认为用户认证成功，那如何从微信获取用户的身份信息？用户信息的拥有者是用户本人，微信需要经过用户的同意方可为知识星球网站生成令牌，知识星球网站拿此令牌方可从微信获取用户的信息。

1、客户端请求第三方授权

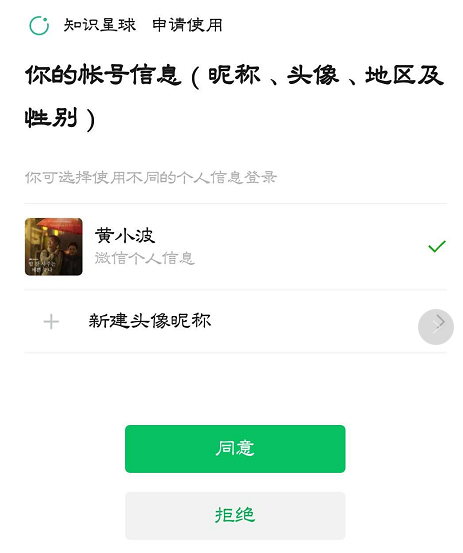
用户进入知识星球网站的登录页面，点击微信的图标以微信账号登录系统，用户是自己在微信里信息的资源拥有者。



此时用户扫描二维码，开始给知识星球网站授权。

2、资源拥有者同意给客户端授权

资源拥有者扫描二维码表示资源拥有者同意给客户端授权，微信会对资源拥有者的身份进行验证， 验证通过后，微信会询问用户是否给授权知识星球网站访问自己的微信数据，用户点击“确认登录”表示同意授权，微信认证服务器会颁发一个授权码，并重定向到知识星球的网站。



3、客户端获取到授权码，请求认证服务器申请令牌

此过程用户看不到，客户端应用程序请求认证服务器，请求携带授权码。

4、认证服务器向客户端响应令牌

微信认证服务器验证了客户端请求的授权码，如果合法则给客户端颁发令牌，令牌是客户端访问资源的通行证。

此交互过程用户看不到，当客户端拿到令牌后，用户在知识星球看到已经登录成功。

5、客户端请求资源服务器的资源

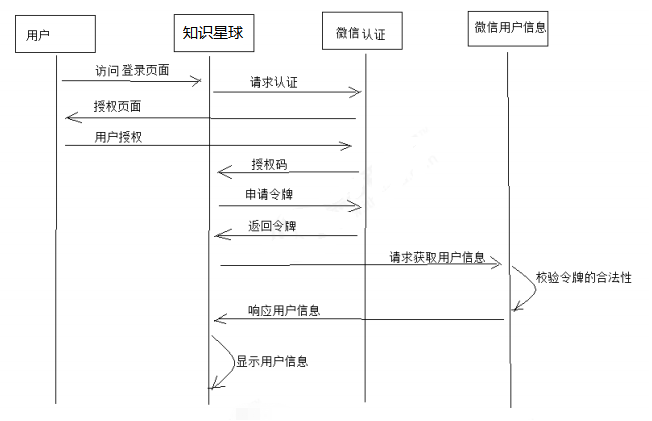
客户端携带令牌访问资源服务器的资源。

知识星球网站携带令牌请求访问微信服务器获取用户的基本信息。

6、资源服务器返回受保护资源

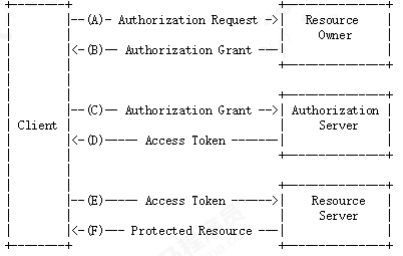
资源服务器校验令牌的合法性，如果合法则向用户响应资源信息内容。

以上认证授权详细的执行流程如下：



通过上边的例子我们大概了解了OAauth2.0的认证过程，下边我们看OAuth2.0认证流程：

引自OAauth2.0协议rfc6749 <https://tools.ietf.org/html/rfc6749>



OAauth2.0包括以下角色：

1、客户端

本身不存储资源，需要通过资源拥有者的授权去请求资源服务器的资源，比如：Android客户端、Web客户端（浏览器端）、微信客户端等。

2、资源拥有者

通常为用户，也可以是应用程序，即该资源的拥有者。

授权服务器（也称认证服务器）

用于服务提供商对资源拥有的身份进行认证、对访问资源进行授权，认证成功后会给客户端发放令牌

（access\_token），作为客户端访问资源服务器的凭据。本例为微信的认证服务器。

4、资源服务器

存储资源的服务器，本例子为微信存储的用户信息。

现在还有一个问题，服务提供商能允许随便一个客户端就接入到它的授权服务器吗？答案是否定的，服务提供商会给准入的接入方一个身份，用于接入时的凭据:

client\_id：客户端标识 client\_secret：客户端秘钥

因此，准确来说，**授权服务器**对两种OAuth2.0中的两个角色进行认证授权，分别是资源拥有者、客户端。

## **5 JWT令牌**

### **5.1 JWT介绍**

通过上边的测试我们发现，当资源服务和授权服务不在一起时资源服务使用RemoteTokenServices 远程请求授权服务验证token，如果访问量较大将会影响系统的性能 。

解决上边问题：

令牌采用JWT格式即可解决上边的问题，用户认证通过会得到一个JWT令牌，JWT令牌中已经包括了用户相关的信息，客户端只需要携带JWT访问资源服务，资源服务根据事先约定的算法自行完成令牌校验，无需每次都请求认证服务完成授权。

#### **5.1.1 什么是JWT？**

JSON Web Token（JWT）是一个开放的行业标准（RFC 7519），它定义了一种简介的、自包含的协议格式，用于在通信双方传递json对象，传递的信息经过数字签名可以被验证和信任。JWT可以使用HMAC算法或使用RSA的公钥/私钥对来签名，防止被篡改。

官网：https://jwt.io/

标准：https://tools.ietf.org/html/rfc7519

JWT令牌的优点：

1）jwt基于json，非常方便解析。

2）可以在令牌中自定义丰富的内容，易扩展。

3）通过非对称加密算法及数字签名技术，JWT防止篡改，安全性高。

4）资源服务使用JWT可不依赖认证服务即可完成授权。

缺点：

1. JWT令牌较长，占存储空间比较大。

#### **5.1.2 JWT令牌结构**

JWT令牌由三部分组成，每部分中间使用点（.）分隔，比如：xxxxx.yyyyy.zzzzz

**Header**

头部包括令牌的类型（即JWT）及使用的哈希算法（如HMAC SHA256或RSA）

一个例子如下：

下边是Header部分的内容

{

"alg": "HS256",

"typ": "JWT"

}

将上边的内容使用Base64Url编码，得到一个字符串就是JWT令牌的第一部分。

**Payload**

第二部分是负载，内容也是一个json对象，它是存放有效信息的地方，它可以存放jwt提供的现成字段，比如：iss（签发者）,exp（过期时间戳）, sub（面向的用户）等，也可自定义字段。

此部分不建议存放敏感信息，因为此部分可以解码还原原始内容。

最后将第二部分负载使用Base64Url编码，得到一个字符串就是JWT令牌的第二部分。

一个例子：

{

"sub": "1234567890",

"name": "456",

"admin": true

}

第三部分是签名，此部分用于防止jwt内容被篡改。

这个部分使用base64url将前两部分进行编码，编码后使用点（.）连接组成字符串，最后使用header中声明签名算法进行签名。

一个例子：

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

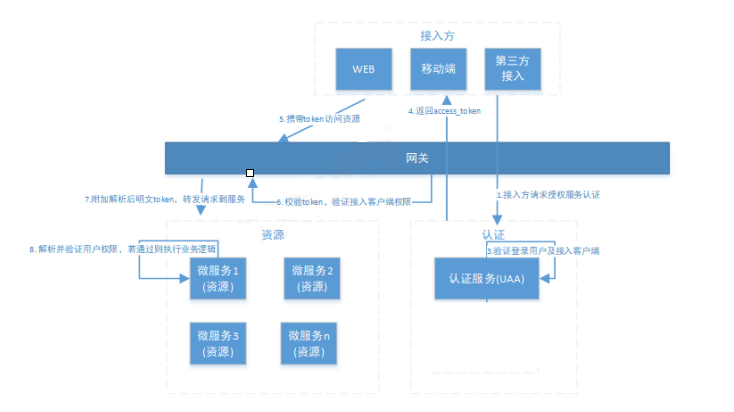
base64UrlEncode(header)：jwt令牌的第一部分。

base64UrlEncode(payload)：jwt令牌的第二部分。

secret：签名所使用的密钥。

## **6 一个完整的案例分析**

未完待续 可以见具体的代码 还未来得及贴代码。



1、UAA认证服务负责认证授权。

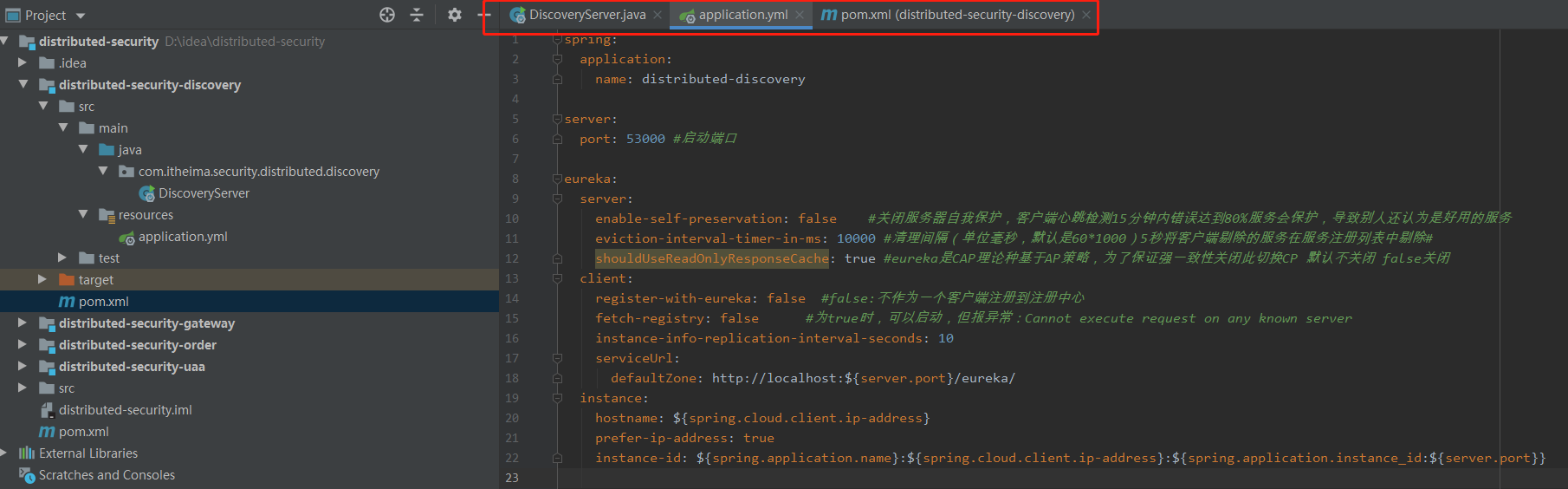
2、所有请求经过 网关到达微服务

3、网关负责鉴权客户端以及请求转发

4、网关将token解析后传给微服务，微服务进行授权。

### **6.1 注册中心**

所有微服务的请求都经过网关，网关从注册中心读取微服务的地址，将请求转发至微服务。



### **6.2 网关**

网关整合 OAuth2.0 有两种思路，一种是认证服务器生成jwt令牌, 所有请求统一在网关层验证，判断权限等操作；

另一种是由各资源服务处理，网关只做请求转发。

建议选用第一种。我们把API网关作为OAuth2.0的资源服务器角色，实现接入客户端权限拦截、令牌解析并转发当前登录用户信息(jsonToken)给微服务，这样下游微服务就不需要关心令牌格式解析以及OAuth2.0相关机制了。

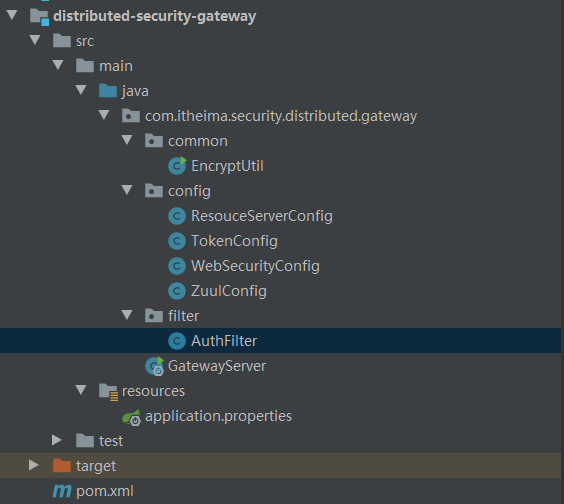
API网关在认证授权体系里主要负责两件事：

（1）作为OAuth2.0的资源服务器角色，实现接入方权限拦截。

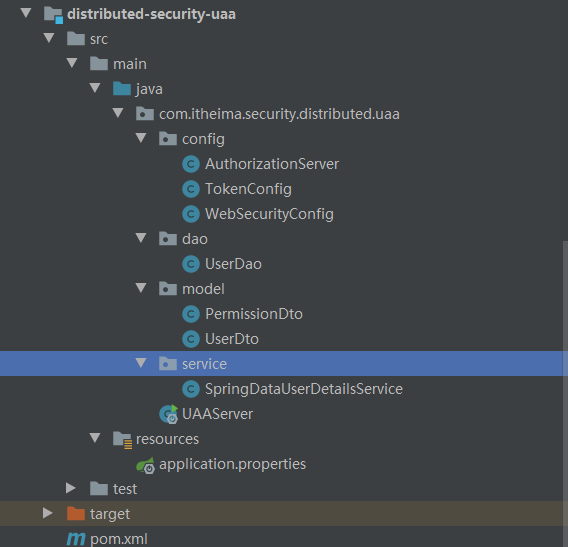
（2）令牌解析并转发当前登录用户信息（明文token）给微服务，微服务拿到明文token(明文token中包含登录用户的身份和权限信息)后也需要做两件事：

（1）用户授权拦截（看当前用户是否有权访问该资源）

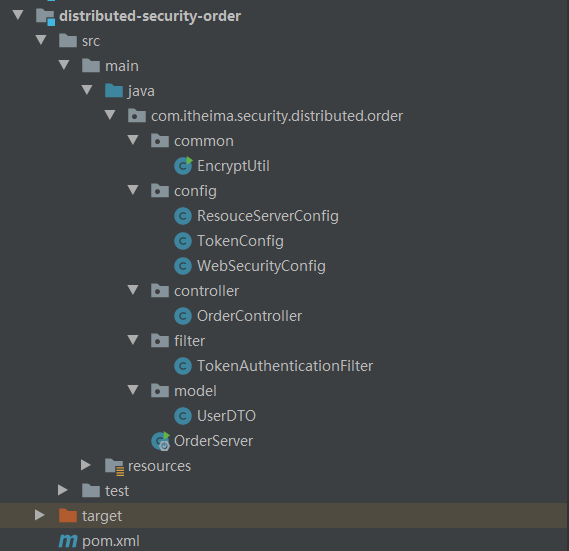
（2）将用户信息存储进当前线程上下文（有利于后续业务逻辑随时获取当前用户信息）



### **6.3 认证服务UUA**



### **6.4 资源微服务**



源代码地址：<https://github.com/hyb-bobo/distributed-security.git>