从单体应用架构到分布式应用架构再到微服务架构，应用的安全访问在不断的经受考验。为了适应架构的变化、需求的变化，身份认证与鉴权方案也在不断的变革。面对数十个甚至上百个微服务之间的调用，如何保证高效安全的身份认证？面对外部的服务访问，该如何提供细粒度的鉴权方案？本文将会为大家阐述微服务架构下的安全认证与鉴权方案。

本文目录：

一、单体应用 VS 微服务

二、微服务常见安全认证方案

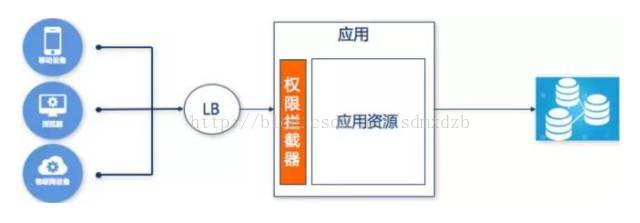
三、JWT介绍

四、OAuth 2.0 介绍

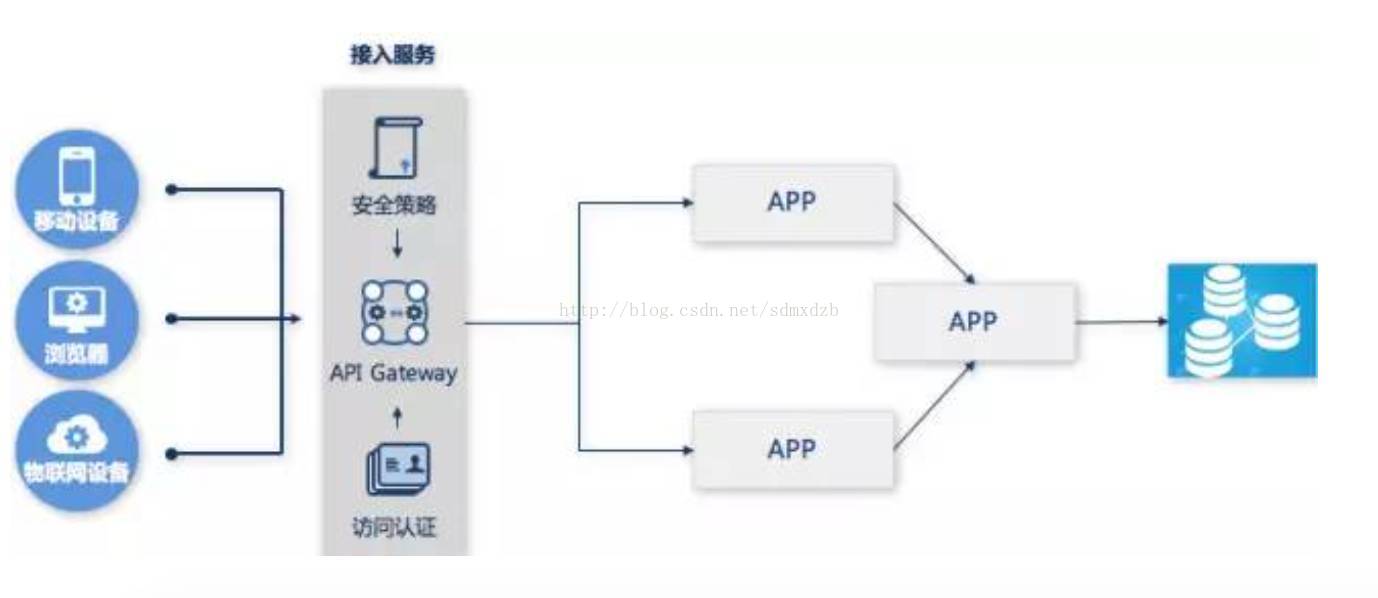
五、思考总结

一、单体应用 VS 微服务

随着微服务架构的兴起，传统的单体应用场景下的身份认证和鉴权面临的挑战越来越大。单体应用体系下，应用是一个整体，一般针对所有的请求都会进行权限校验。请求一般会通过一个权限的拦截器进行权限的校验，在登录时将用户信息缓存到 session 中，后续访问则从缓存中获取用户信息。



而微服务架构下，一个应用会被拆分成若干个微应用，每个微应用都需要对访问进行鉴权，每个微应用都需要明确当前访问用户以及其权限。尤其当访问来源不只是浏览器，还包括其他服务的调用时，单体应用架构下的鉴权方式就不是特别合适了。在为服务架构下，要考虑外部应用接入的场景、用户 - 服务的鉴权、服务 - 服务的鉴权等多种鉴权场景。



David Borsos 在伦敦的微服务大会上提出了四种方案：

1. 单点登录（SSO）

这种方案意味着每个面向用户的服务都必须与认证服务交互，这会产生大量非常琐碎的网络流量和重复的工作，当动辄数十个微应用时，这种方案的弊端会更加明显。

2. 分布式 Session 方案

分布式会话方案原理主要是将关于用户认证的信息存储在共享存储中，且通常由用户会话作为 key 来实现的简单分布式哈希映射。当用户访问微服务时，用户数据可以从共享存储中获取。在某些场景下，这种方案很不错，用户登录状态是不透明的。同时也是一个高可用且可扩展的解决方案。这种方案的缺点在于共享存储需要一定保护机制，因此需要通过安全链接来访问，这时解决方案的实现就通常具有相当高的复杂性了。

3. 客户端 Token 方案

令牌在客户端生成，由身份验证服务进行签名，并且必须包含足够的信息，以便可以在所有微服务中建立用户身份。令牌会附加到每个请求上，为微服务提供用户身份验证，这种解决方案的安全性相对较好，但身份验证注销是一个大问题，缓解这种情况的方法可以使用短期令牌和频繁检查认证服务等。对于客户端令牌的编码方案，Borsos 更喜欢使用 JSON Web Tokens（JWT），它足够简单且库支持程度也比较好。

4. 客户端 Token 与 API 网关结合

这个方案意味着所有请求都通过网关，从而有效地隐藏了微服务。 在请求时，网关将原始用户令牌转换为内部会话 ID 令牌。在这种情况下，注销就不是问题，因为网关可以在注销时撤销用户的令牌。

二、微服务常见安全认证方案

HTTP 基本认证

HTTP Basic Authentication（HTTP 基本认证）是 HTTP 1.0 提出的一种认证机制，这个想必大家都很熟悉了，我不再赘述。HTTP 基本认证的过程如下：

客户端发送 HTTP Request 给服务器。

因为 Request 中没有包含 Authorization header，服务器会返回一个 401 Unauthozied 给客户端，并且在 Response 的 Header "WWW-Authenticate" 中添加信息。

客户端把用户名和密码用 BASE64 加密后，放在 Authorization Header 中发送给服务器， 认证成功。

服务器将 Authorization Header 中的用户名密码取出，进行验证， 如果验证通

过，将根据请求，发送资源给客户端。

基于 Session 的认证

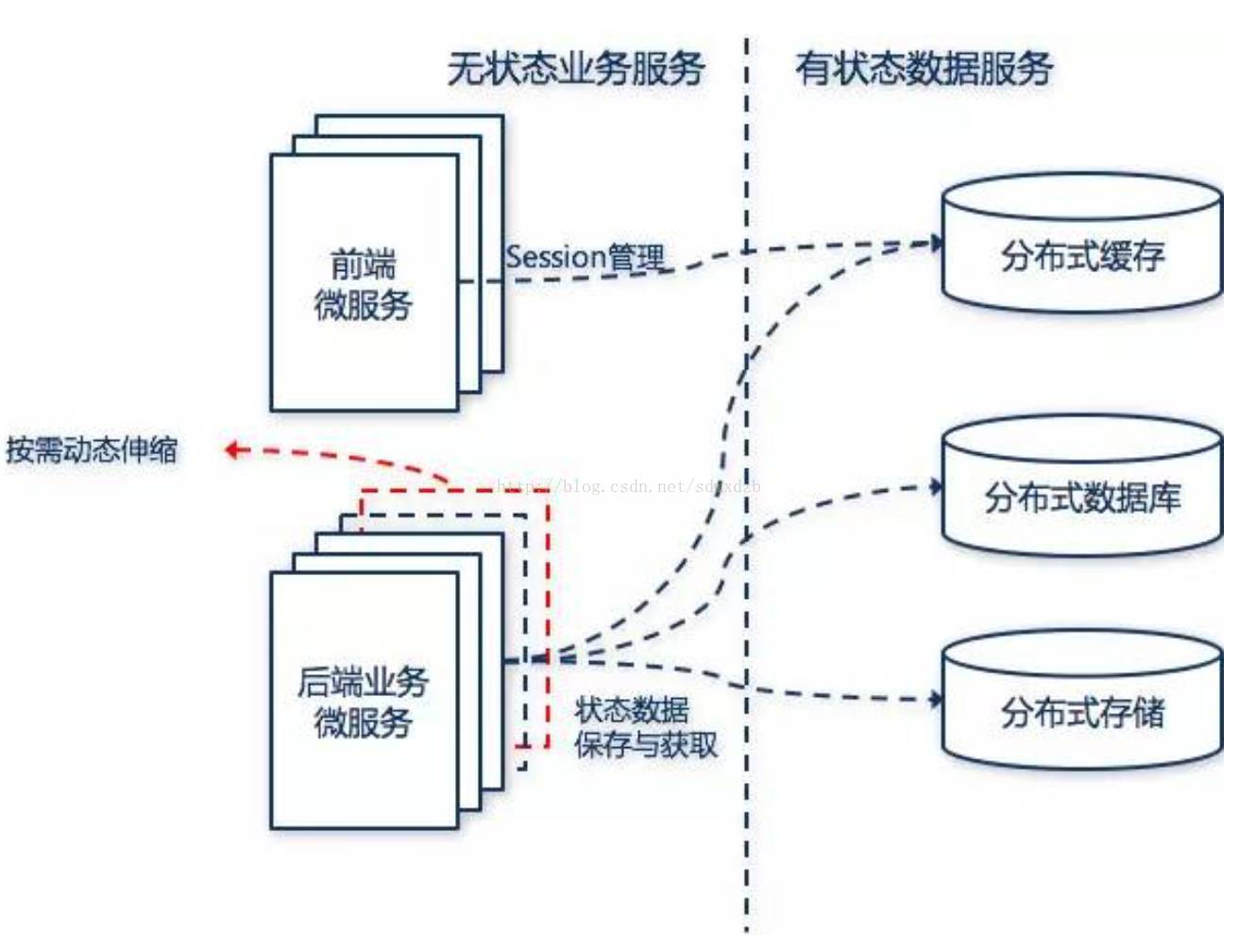
基于 Session 的认证应该是最常用的一种认证机制了。用户登录认证成功后，将用户相关数据存储到 Session 中，单体应用架构中，默认 Session 会存储在应用服务器中，并且将 Session ID 返回到客户端，存储在浏览器的 Cookie 中。

但是在分布式架构下，Session 存放于某个具体的应用服务器中自然就无法满足使用了，简单的可以通过 Session 复制或者 Session 粘制的方案来解决。

Session 复制依赖于应用服务器，需要应用服务器有 Session 复制能力，不过现在大部分应用服务器如 Tomcat、JBoss、WebSphere 等都已经提供了这个能力。

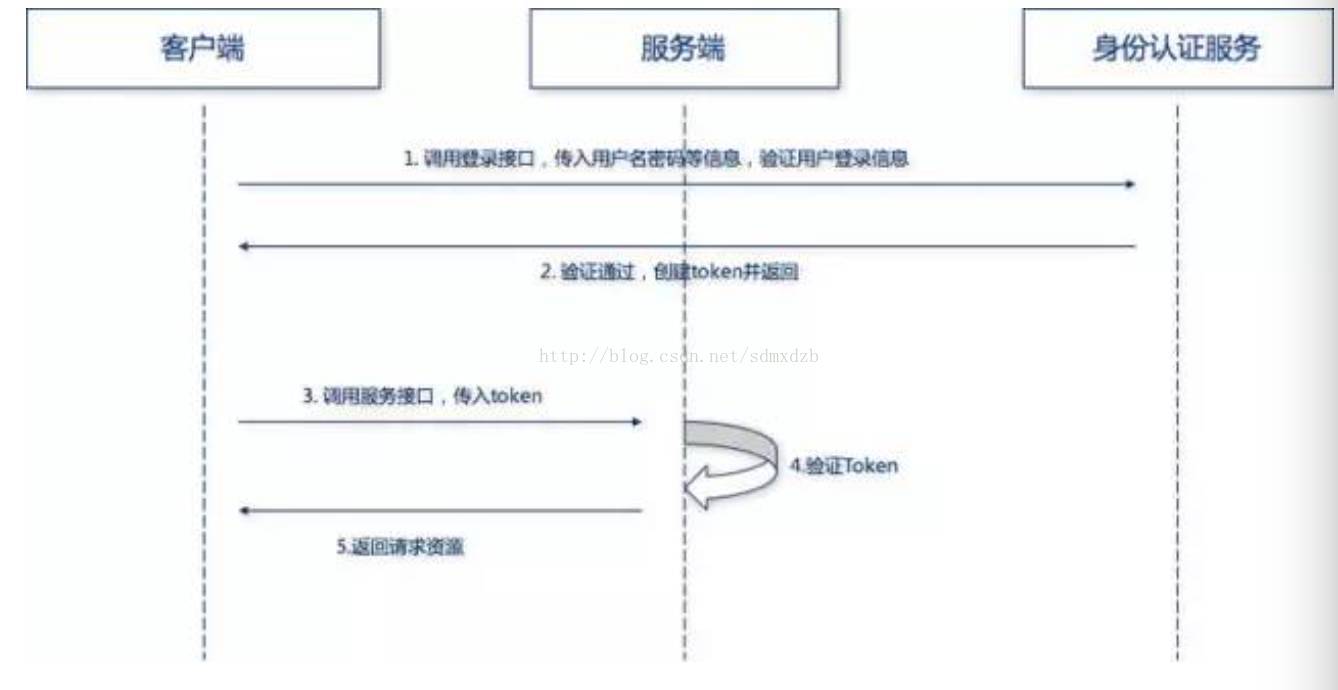
除此之外，Session 复制的一大缺陷在于当节点数比较多时，大量的 Session 数据复制会占用较多网络资源。Session 粘滞是通过负载均衡器，将统一用户的请求都分发到固定的服务器节点上，这样就保证了对某一用户而言，Session 数据始终是正确的。不过这种方案依赖于负载均衡器，并且只能满足水平扩展的集群场景，无法满足应用分割后的分布式场景。

在微服务架构下，每个微服务拆分的粒度会很细，并且不只有用户和微服务打交道，更多还有微服务间的调用。这个时候上述两个方案都无法满足，就要求必须要将 Session 从应用服务器中剥离出来，存放在外部进行集中管理。可以是数据库，也可以是分布式缓存，如 Memchached、Redis 等。这正是 David Borsos 建议的第二种方案，分布式 Session 方案。



基于 Token 的认证

随着 Restful API、微服务的兴起，基于 Token 的认证现在已经越来越普遍。Token 和 Session ID 不同，并非只是一个 key。Token 一般会包含用户的相关信息，通过验证 Token 就可以完成身份校验。像 Twitter、微信、QQ、GitHub 等公有服务的 API 都是基于这种方式进行认证的，一些开发框架如 OpenStack、Kubernetes 内部 API 调用也是基于 Token 的认证。基于 Token 认证的一个典型流程如下：



用户输入登录信息（或者调用 Token 接口，传入用户信息），发送到身份认证服务进行认证（身份认证服务可以和服务端在一起，也可以分离，看微服务拆分情况了）。

身份验证服务验证登录信息是否正确，返回接口（一般接口中会包含用户基础信息、权限范围、有效时间等信息），客户端存储接口，可以存储在 Session 或者数据库中。

用户将 Token 放在 HTTP 请求头中，发起相关 API 调用。

被调用的微服务，验证 Token 权限。

服务端返回相关资源和数据。

基于 Token 认证的好处如下：

服务端无状态：Token 机制在服务端不需要存储 session 信息，因为 Token 自身包含了所有用户的相关信息。

性能较好，因为在验证 Token 时不用再去访问数据库或者远程服务进行权限校验，自然可以提升不少性能。

支持移动设备。

支持跨程序调用，Cookie 是不允许垮域访问的，而 Token 则不存在这个问题。

下面会重点介绍两种基于 Token 的认证方案 JWT/Oauth2.0。

三、JWT介绍

JSON Web Token（JWT）是为了在网络应用环境间传递声明而执行的一种基于 JSON 的开放标准（RFC 7519）。来自 JWT RFC 7519 标准化的摘要说明：JSON Web Token 是一种紧凑的，URL 安全的方式，表示要在双方之间传输的声明。JWT 一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该 Token 也可直接被用于认证，也可被加密。

JWT 认证流程

客户端调用登录接口（或者获取 token 接口），传入用户名密码。

服务端请求身份认证中心，确认用户名密码正确。

服务端创建 JWT，返回给客户端。

客户端拿到 JWT，进行存储（可以存储在缓存中，也可以存储在数据库中，如果是浏览器，可以存储在 Cookie 中）在后续请求中，在 HTTP 请求头中加上 JWT。

服务端校验 JWT，校验通过后，返回相关资源和数据。

JWT 结构

JWT 是由三段信息构成的，第一段为头部（Header），第二段为载荷（Payload)，第三段为签名（Signature）。每一段内容都是一个 JSON 对象，将每一段 JSON 对象采用 BASE64 编码，将编码后的内容用. 链接一起就构成了 JWT 字符串。如下：

header.payload.signature

1. 头部（Header）

头部用于描述关于该 JWT 的最基本的信息，例如其类型以及签名所用的算法等。这也可以被表示成一个 JSON 对象。



在头部指明了签名算法是 HS256 算法。

2. 载荷（payload）

载荷就是存放有效信息的地方。有效信息包含三个部分：

标准中注册的声明

公共的声明

私有的声明

标准中注册的声明（建议但不强制使用）：

iss：JWT 签发者

sub：JWT 所面向的用户

aud：接收 JWT 的一方

exp：JWT 的过期时间，这个过期时间必须要大于签发时间

nbf：定义在什么时间之前，该 JWT 都是不可用的

iat：JWT 的签发时间

jti：JWT 的唯一身份标识，主要用来作为一次性 token, 从而回避重放攻击。

公共的声明 ：

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息. 但不建议添加敏感信息，因为该部分在客户端可解密。

私有的声明 ：

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为 base64 是对称解密的，意味着该部分信息可以归类为明文信息。

示例如下：



3. 签名（signature)

创建签名需要使用 Base64 编码后的 header 和 payload 以及一个秘钥。将 base64 加密后的 header 和 base64 加密后的 payload 使用. 连接组成的字符串，通过 header 中声明的加密方式进行加盐 secret 组合加密，然后就构成了 jwt 的第三部分。

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

JWT 的优点：

跨语言，JSON 的格式保证了跨语言的支撑

基于 Token，无状态

占用字节小，便于传输

关于 Token 注销：

Token 的注销，由于 Token 不存储在服务端，由客户端存储，当用户注销时，Token 的有效时间还没有到，还是有效的。所以如何在用户注销登录时让 Token 注销是一个要关注的点。一般有如下几种方式：

Token 存储在 Cookie 中，这样客户端注销时，自然可以清空掉

注销时，将 Token 存放到分布式缓存中，每次校验 Token 时区检查下该 Token 是否已注销。不过这样也就失去了快速校验 Token 的优点。

多采用短期令牌，比如令牌有效期是 20 分钟，这样可以一定程度上降低注销后 Token 可用性的风险。

四、OAuth 2.0 介绍

OAuth 的官网介绍：An open protocol to allow secure API authorization in a simple and standard method from desktop and web applications。OAuth 是一种开放的协议，为桌面程序或者基于 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。OAUTH 认证授权具有以下特点：

简单：不管是 OAuth 服务提供者还是应用开发者，都很容易于理解与使用；

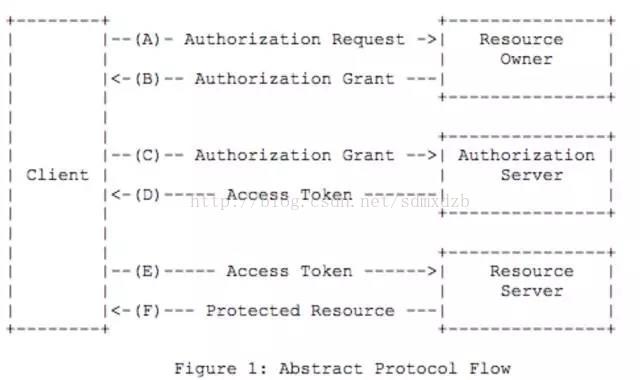
安全：没有涉及到用户密钥等信息，更安全更灵活；

开放：任何服务提供商都可以实现 OAuth，任何软件开发商都可以使用 OAuth；

OAuth 2.0 是 OAuth 协议的下一版本，但不向后兼容 OAuth 1.0，即完全废止了 OAuth 1.0。 OAuth 2.0 关注客户端开发者的简易性。要么通过组织在资源拥有者和 HTTP 服务商之间的被批准的交互动作代表用户，要么允许第三方应用代表用户获得访问的权限。同时为 Web 应用，桌面应用和手机，和起居室设备提供专门的认证流程。2012 年 10 月，OAuth 2.0 协议正式发布为 RFC 6749。

授权流程

OAuth 2.0 的流程如下：



（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。

四大角色

由授权流程图中可以看到 OAuth 2.0 有四个角色：客户端、资源拥有者、资源服务器、授权服务器。

客户端：客户端是代表资源所有者对资源服务器发出访问受保护资源请求的应用程序。

资源拥有者：资源拥有者是对资源具有授权能力的人。

资源服务器：资源所在的服务器。

授权服务器：为客户端应用程序提供不同的 Token，可以和资源服务器在统一服务器上，也可以独立出去。

客户端的授权模式

客户端必须得到用户的授权（Authorization Grant），才能获得令牌（access token）。OAuth 2.0 定义了四种授权方式：authorizationcode、implicit、resource owner password credentials、client credentials。

1. 授权码模式（authorization code）

授权码模式（authorization code）是功能最完整、流程最严密的授权模式。它的特点就是通过客户端的后台服务器，与"服务提供商"的认证服务器进行互动。流程如下：

用户访问客户端，后者将前者导向认证服务器。

用户选择是否给予客户端授权。

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。

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。

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）。

2. 简化模式（implicit）

简化模式（Implicit Grant Type）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。流程如下：

客户端将用户导向认证服务器。

用户决定是否给于客户端授权。

假设用户给予授权，认证服务器将用户导向客户端指定的"重定向 URI"，并在 URI 的 Hash 部分包含了访问令牌。

浏览器向资源服务器发出请求，其中不包括上一步收到的 Hash 值。

资源服务器返回一个网页，其中包含的代码可以获取 Hash 值中的令牌。

浏览器执行上一步获得的脚本，提取出令牌。

浏览器将令牌发给客户端。

3. 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。流程如下：

用户向客户端提供用户名和密码。

客户端将用户名和密码发给认证服务器，向后者请求令牌。

认证服务器确认无误后，向客户端提供访问令牌。

4. 客户端模式（Client Credentials）

客户端模式（Client Credentials Grant）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。流程如下：

客户端向认证服务器进行身份认证，并要求一个访问令牌。

认证服务器确认无误后，向客户端提供访问令牌。

五、思考总结

正如 David Borsos 所建议的一种方案，在微服务架构下，我们更倾向于将 Oauth 和 JWT 结合使用，Oauth 一般用于第三方接入的场景，管理对外的权限，所以比较适合和 API 网关结合，针对于外部的访问进行鉴权（当然，底层 Token 标准采用 JWT 也是可以的）。

JWT 更加轻巧，在微服务之间进行访问鉴权已然足够，并且可以避免在流转过程中和身份认证服务打交道。当然，从能力实现角度来说，类似于分布式 Session 在很多场景下也是完全能满足需求，具体怎么去选择鉴权方案，还是要结合实际的需求来。