第 1 章

什么是身份认证

提　　示
毫无疑问，对于一个安全的应用来说，身份认证是第一道门槛，它为后续所有的安全措施提供“身份”这样一个关键信息。

在数字化时代，身份认证成为保护用户隐私和确保数据安全的重要环节。无论是登录社交媒体账户、进行在线银行交易还是访问个人健康记录，身份认证都扮演着关键的角色。然而，什么是身份认证？为了深入理解身份认证的概念和意义，让我们给予身份认证一个明确的定义。

身份认证是一种确认用户或实体声明的过程，通过验证其所提供的凭证和属性，确定其所宣称的身份的真实性和合法性。简而言之，身份认证旨在确认一个主体是他所声称的那个主体。这个过程涉及验证用户所提供的标识信息，如用户名、密码、指纹、虹膜等，以确保被认证的主体是合法的用户，并有权访问特定资源或执行特定操作。

如果要再细化一点，可以将身份与认证再拆开成身份识别与认证两部分。举个例子，当你使用用户名、密码登录一个系统时，用户名用来识别你的身份，而密码用来认证。如果你使用指纹登录某个系统，那么指纹既被用作身份识别，也被用作认证。

身份认证的核心目标是确保只有经过授权的用户才能够获得访问权限，并阻止未经授权的个体侵入系统或获取敏感信息。通过使用各种认证方法和技术，身份认证可以提供一种可靠的方式来验证用户的身份，从而建立起可信任的数字交互环境。

身份认证在现代计算机系统和网络中具有广泛的应用。它不仅仅用于个人用户的登录和访问控制，还用于机器对机器的通信和交互，以及各种企业和组织间的身份验证。无论是通过传统的用户名和密码认证，还是使用更先进的生物特征识别和多因素认证方法，身份认证都是构建安全可靠的数字身份基础设施的关键组成部分。

本章将深入探讨身份认证的概念和原理，以及与授权之间的联系和区别。我们将研究不同的认证目标对象，如机器和人类用户，并探讨各种常见的认证方式和场景。通过全面理解身份认证的本质，读者将能够更好地应用和理解后续章节中介绍的具体身份认证技术和解决方案。

让我们深入探索身份认证的世界，揭示其中的奥秘和挑战，为构建安全可靠的数字化未来铺平道路。

1.1　身份认证简介以及和授权的联系与区别

本节将介绍身份认证的概念，并探讨身份认证与授权之间的联系与区别。虽然这两个概念经常被一起提及，但它们在数字身份管理中扮演着不同的角色。

1.1.1　身份认证简介

身份认证是确认用户或实体声明的过程，通过验证其所提供的凭证和属性，确定其所宣称的身份的真实性和合法性。身份认证的目标是验证一个主体是他所声称的那个主体，并验证其对特定资源或操作的访问权限。身份认证通常涉及以下要素。

- 标识信息：用户提供的用于识别自己身份的信息，如用户名、邮箱、手机号等。
- 凭证：用户提供的用于证明自己身份的凭证，如密码、数字证书、生物特征等。
- 认证过程：用于验证用户所提供的标识信息和凭证的过程，以确保其真实性和合法性。

1.1.2　身份认证与授权的联系与区别

身份认证和授权是数字身份管理中的两个核心概念，它们密切相关，但又有明显的区别。

身份认证用于确认用户的身份，确保其是合法用户。它验证用户所提供的标识信息和凭证，并通过比对和验证来确定用户的真实身份。认证是一个“是谁”的问题。

授权是指在身份认证的基础上，为用户分配适当的权限和访问权限，以决定用户能够访问哪些资源或执行哪些操作。授权是一个“能做什么”的问题。

换句话说，身份认证用于确认用户的身份，而授权决定用户在系统或应用中的权限范围。身份认证是用户身份验证的过程，而授权是根据用户的身份和权限进行访问控制的过程，如图 1-1 所示。

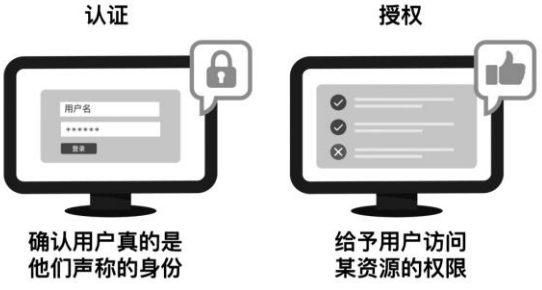

图 1-1　认证与授权的区别

需要注意的是，身份认证和授权是互相补充且相互依赖的过程。在访问控制的流程中，首先要进行身份认证，验证用户的身份。只有在成功认证后，用户才能接受授权并获得相应的权限。而在开放身份互联的过程中，身份认证协议又会在授权过程完成之后为客户端颁发身份令牌，即身份认

证和授权是相互依赖、紧密联系的过程。

总结起来，身份认证是确认用户身份的过程，授权是为用户分配权限和访问权限的过程。通过清晰地区分身份认证和授权的概念，我们能够更好地理解和应用身份管理领域的相关技术和实践。

在接下来的章节中，我们将更深入地探讨不同的身份认证方式、场景和技术，帮助读者建立起更全面的身份认证知识体系。

1.2 认证目标对象有哪些

本节将探讨身份认证的目标对象，即身份认证所针对的主体类型。身份认证并不局限于人类用户，它可以应用于不同的目标对象，涵盖广泛的应用场景。

1.2.1 机器认证

在现代计算机网络和通信领域，机器之间的通信也需要进行身份认证。机器之间的身份认证被称为 Peer Authentication，主要用于确保通信的可靠性和安全性。例如，在分布式系统中，不同的节点需要通过身份认证来建立安全的通信通道，以确保数据的机密性和完整性。机器身份认证的实现方式与人类用户的身份认证有所不同，通常涉及使用密钥、证书或其他身份凭据。

机器是秘密的非人类消费者，如机器人、移动终端、服务器、虚拟机、容器、应用程序、微服务、Kubernetes 服务账户、Ansible 节点和其他自动化进程，为机器提供可靠、安全的识别和授权是身份进化很重要的组成部分。一旦可以对各类机器进行有效识别和授权，就能够使用策略来控制机器可以访问哪些秘密内容，以及还有哪些用户（机器和人员）可以访问指定机器，从而更好地完成操作管理、SSH 访问、流量控制等工作[6]。

提　示

在说到身份认证时，尽管人们首先会想到人类用户，但有趣的是，最先应用数字身份认证技术的却是机器之间的通信。在 1970 年，美国国防部的研究人员就开始研究如何在分布式系统中实现机器之间的身份认证，以确保通信的安全性和可靠性。在 1980 年，随着公钥密码学的发展，数字身份认证技术才开始应用于人类用户的身份认证，成为现代身份认证技术的基础。

在 Open API 概念盛行的今天，API 都要为开放而设计。但是开放后如何保证安全呢？为开放而设计的认证方案就非常适合用于 Open API。为此，我们需要理解一些安全概念以及安全共建责任模型。

1. 安全共建责任模型

详情可以参考 AWS 的安全共建责任模型。简单来说，没有任何单一实体可以保证系统安全；安全是一个共建的过程，需要所有的参与者共同努力。在这个模型中，AWS 为客户提供了一个安全的基础设施，但是客户也需要确保自己的应用程序和数据是安全的。图 1-2 展示了一个概念图。

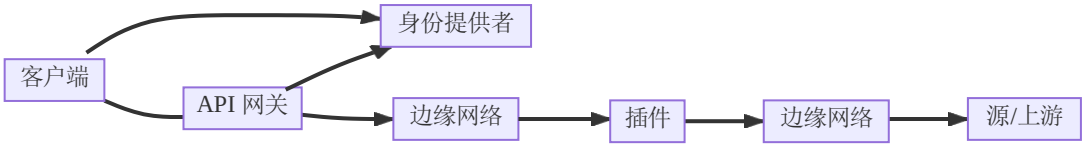

图 1-2　概念图

图 1-2 涉及一些术语，它们的解释如表 1-1 所示。

表 1-1　相关术语

术　　语	解　　释
身份	用户或者应用的身份，在面向机器的认证中，特指 API 的身份，也称安全主体
认证	即验证身份的处理过程
授权	指定并且实施对资源的访问，确保调用者能且只能做有权限做的事情，也称访问控制
身份提供者	身份提供者为安全主体创建、维护和管理身份信息，同时也为依赖它的应用通过联邦的方式或者在分布式网络中提供认证服务
客户端	访问 API 的一个用户代理、程序或者设备，也称 API 消费者
边缘网络	API 网关的一部分，从客户端接收 IP 流量。在使用 Kong 的组织里，边缘网络通常是一个或者多个 Kong 节点，又称数据平面
API 网关	API 网关组成部分的所有集合，比如边缘网络、控制平面、数据平面、插件等
源	API 网关在处理请求时的内部调用源头，也称上游
传输层	网络中的节点所使用的网络协议，比如和 TLS 结合使用的 TCP/IP、UDP/IP
Json Web Token	简称 JWT，用于向 API 提供认证信息。这种令牌包含由身份提供者签名的声明。如果令牌中包含范围，那么授权也可以被实施。一般情况下，JWT 以 Bearer token 的形式通过 Authorization 标头发送给 API

除 AWS 外，几乎所有的云服务商都会使用安全责任共建模型。值得注意的是，账号与身份都是客户的责任，而不是云服务商的责任。这意味着，客户需要确保自己的账号和身份是安全的，以防止被攻击者利用。

2. 实践模式

在实践中，幻影令牌和零信任 API 事件是两种常见的模式，可用于安全共建责任模型。

1）幻影令牌

这个模式用于在身份验证和授权过程中，通过使用临时的、单次性的令牌来增强安全性。在该模式下，客户端向服务器请求一个幻影令牌，该令牌包含一些用于验证客户端身份和访问权限的信息。服务器验证该幻影令牌并生成一个授权令牌，该授权令牌用于后续的请求。这种模式可以提供额外的安全性，因为幻影令牌的生命周期很短，且只能用于一次特定的请求，降低了令牌被滥用或被劫持的风险。

幻影令牌结合了不透明令牌和结构化令牌的优点，可以在不暴露用户信息的情况下，提供更好的安全性和可扩展性。

- 不透明令牌更安全，用在边缘网络中。在身份验证和授权系统中，不透明令牌是一种令牌类型，它的内容对客户端来说是不可解读的。客户端无法直接理解或访问令牌的详细信息，而是将令牌

发送给授权服务器进行验证。授权服务器可以解读和验证令牌，并根据需要提供相应的访问权限。

- 它仅仅是一个随机字符串，没有任何结构化信息，更没有 PII 信息。
- 体积更小，对缓存友好。
- 更易于刷新或者撤销。

- 结构化令牌是内部使用的最佳实践。结构化令牌是一种令牌类型，其内容被组织为特定的结构或格式。这种令牌通常包含有关用户、权限、有效期等信息，并使用某种标准或协议进行编码和解码。客户端可以直接读取和解析结构化令牌的内容，以获得与身份验证和授权相关的信息。结构化令牌可以提供更多的灵活性和可扩展性，允许在令牌中携带更多自定义的数据。
  - 在上游源中性能可以进一步优化。服务通过使用结构化令牌，从而得到了所有需要的信息，而不需要再次查询身份提供者。
  - 内部使用结构化令牌，外部对其不可知，从而在调整内部结构化令牌的结构时，不会对外部造成不稳定的影响。

然而，要使用这种模式，需要结合 API 网关架构，并且要在 API 网关和身份提供者之间增加一个中间层（反向代理层），用于处理幻影令牌和授权令牌之间的转换。图 1-3 展示了该架构。

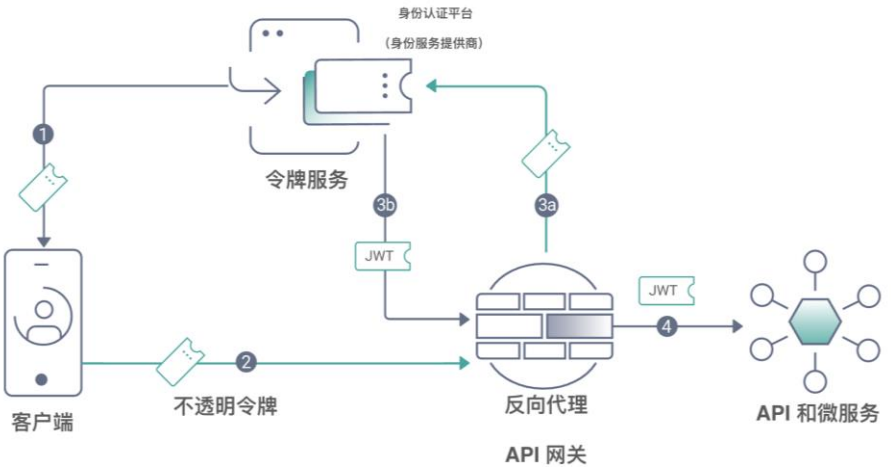

图 1-3　幻影令牌架构

2）零信任 API 事件

在处理事件消息时，很容易失去身份控制，这可能导致网络内部的安全威胁。在零信任架构中，Zero Trust API Events Pattern 可以帮助确保以下事项：

（1）在异步工作流程恢复时，事件消息中的数据可以进行数字验证。
（2）在消费事件消息时，可以验证访问权限。
（3）当处理的事件包含非关键数据且不需要额外的访问控制时，这种模式可能是多余的。

1.2.2　人类认证

在众多的应用场景中，人类用户是身份认证的主要目标对象。无论是登录一个网站、访问一个应用程序，还是进行在线交易，人类用户都需要通过身份认证来证明自己的身份。这种类型的身份

认证通常涉及用户提供的标识信息和凭证的验证，以确保只有合法用户可以访问特定的资源或执行特定的操作。

身份认证的目标对象可以是不同的个体，包括但不限于个人用户、企业合作伙伴、开发者、内部员工等，如图 1-4 所示。根据不同的场景和应用需求，身份认证可以针对特定类型的目标对象进行定制化的实现。

通过对认证目标对象的深入理解，我们可以更好地把握身份认证的适用范围和应用场景，为不同类型的用户提供安全、可靠的身份认证解决方案。

接下来，我们将详细介绍不同的身份认证方式、技术和实践，以帮助读者全面理解身份认证的概念和应用。

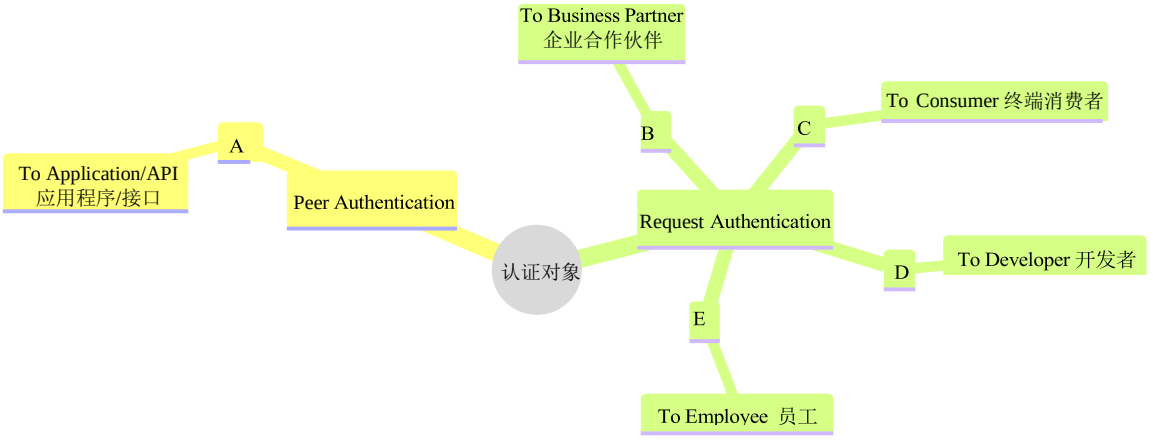

图 1-4　认证对象和场景

1.3　认证场景有哪些

身份认证在不同的场景下有不同的需求和应用方式。简单地分类，认证的主要场景有对外认证和对内认证[1]，对外认证面对的主要是终端客户，而对内认证面对的主要是员工和合作方。出乎意料的是，对外认证的场景更单一，而对内认证的场景则非常复杂。笔者曾以为企业的生存更依赖外部客户，因此对外认证“当然”应该更为重要，也更加复杂。但是，随着自己经验的积累，才发现事实正好相反，一般来说，对外认证场景要比对内认证的场景简单得多。

为了更加深入地了解和解决实际问题，笔者更愿意将场景再细分一下，并且在这个细分的场景下，会发现有一些场景是同时存在对外和对内认证的。笔者将这些场景按照首字母列举出来，发现正好可以用 ABCDE 来概括，如图 1-5 所示。其中，面向开发者的认证场景，即 2D，既存在于对内又存在于对外的情况，以下一一详述。

[1] https://time.geekbang.org/column/article/178520

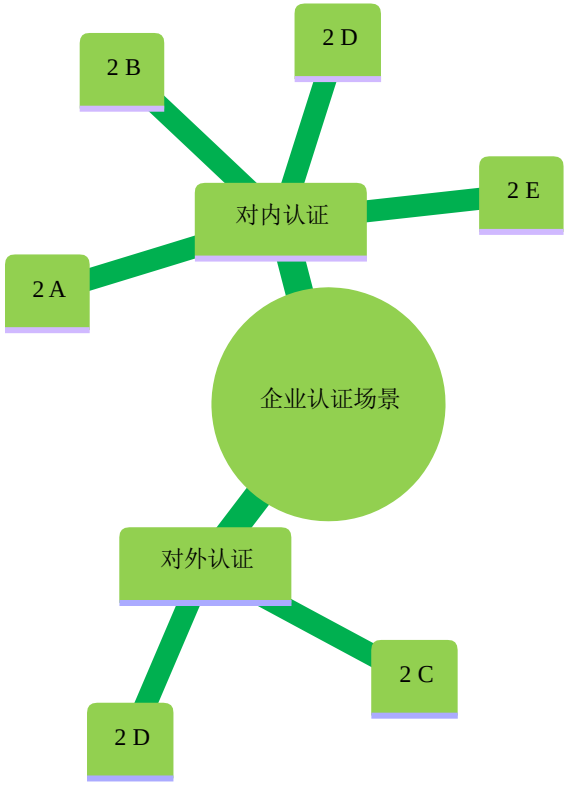

图 1-5　企业认证场景

1.3.1　2A，面向 API 的身份认证

在许多应用程序中，API（Application Programming Interface，应用程序编程接口）扮演着关键的角色，用于数据交换和服务访问。2A（To API）身份认证场景是指在 API 访问中使用身份认证来验证请求的合法性。在这种场景下，身份认证通常使用 API 密钥、令牌或其他形式的凭证。

这种场景其实也包括对内和对外。一般建议有两个 API 网关，一个网关面向内部，另一个网关面向外部。但是网关的作用都是类似的，都会对 API 的调用者鉴权。一个企业内部会存在多个领域和子领域，能力的复用也可以通过 API 的形式来提供。

API 可以分为同步 API 和异步 API，同步 API 多是我们熟悉的 Restful API，或者是 GraphQL。而异步 API 一般应用了事件驱动架构。建议领域内部通过同步 API 来调用，而跨领域使用异步 API 以解耦。

1.3.2　2B，面向企业合作伙伴的身份认证

企业合作伙伴之间的信息交换和资源共享需要进行身份认证以确保安全性和可信度。2B（To Business）身份认证场景是指企业间的身份认证，允许合作伙伴之间进行受控的访问和交互。

1.3.3　2C，面向客户的身份认证

面向客户的身份认证是指为终端用户提供访问应用程序或在线服务的身份验证机制。2C（To Consumer）身份认证场景在电子商务、社交媒体和在线银行等领域非常常见。在这种场景下，常用的认证方式包括用户名密码登录、社交登录和多因素身份验证等。

关于该领域的详细探讨，可以参考 Simon Moffatt 编写的 *Consumer Identity & Access Management Design Fundamentals* 一书。

1.3.4　2D，面向开发者的身份认证

2D（To Developer）身份认证场景是指开发者之间进行身份认证，以便在开发过程中访问受保护的资源和服务。在这种场景下，通常使用开发者密钥、API 密钥或令牌进行认证。

在企业内部，有很多供开发者使用的系统，如常见的日志系统、监控系统等。在这种情况下，开发者是企业的内部员工的一个子集。当然，还有面向外部开发者的系统，比如大的平台，都会提供开发者门户网站，或者开放平台等，这时面向的主要就是外部开发者了。

谈到一门生意，一般我们会想到 2B 或者 2C，而实际上，2D 也可以做成很大的生意，比如微信，其 2D 业务如图 1-6 所示。

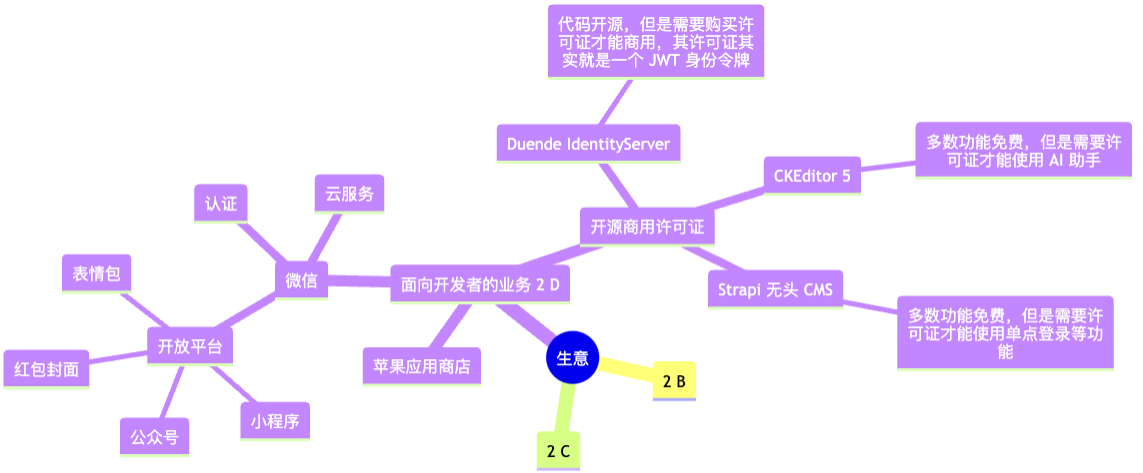

图 1-6　常见的 2D 业务

这时 2D 的身份认证建设至关重要。

1.3.5　2E，面向内部员工的身份认证

企业内部系统和资源的访问需要对内部员工进行身份认证。2E（To Employee）身份认证场景是指面向内部员工的身份验证，确保他们只能访问其所需的受保护资源和权限。

1.4 常用术语有哪些

在身份认证领域，有一些常用的术语和概念，在相关的文章中以及本书后面的内容中，将常常提到这些术语，如图 1-7 所示。

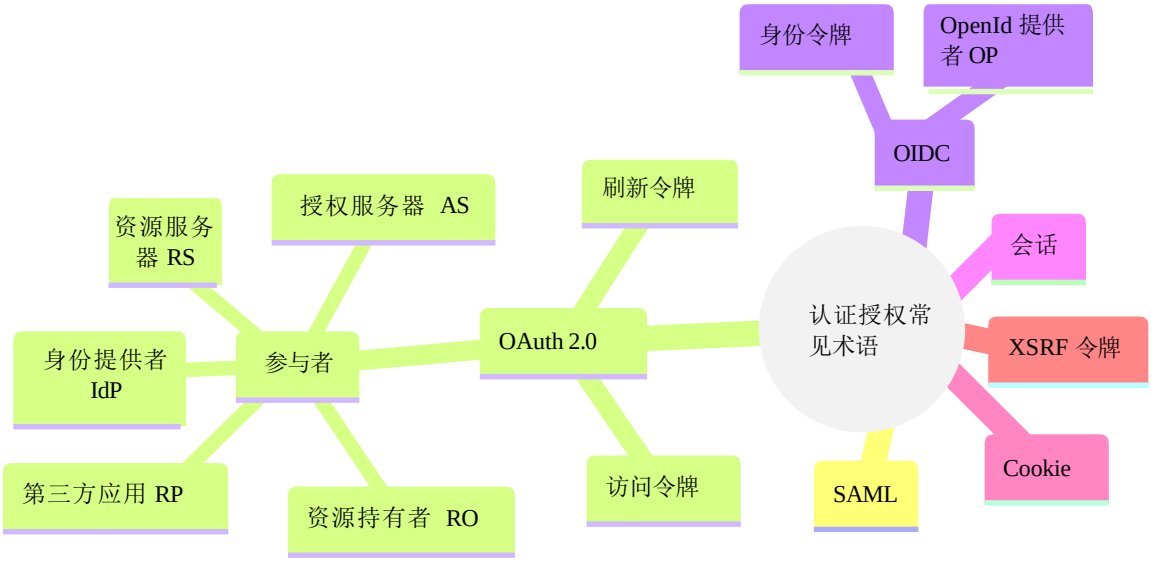

图 1-7 常见术语

1.4.1 令牌与会话如何选择

在身份认证中，令牌（Token）和会话（Session）是两个常用的概念。它们用于表示用户的身份信息和认证状态，但在实际应用中使用的方式有所不同。

- 令牌用于在身份认证和授权过程中传递身份信息和访问权限的数据结构。令牌是一种包含身份信息的数据结构，通常是一个字符串。它可以被用于验证用户的身份和权限，以及在不同的请求之间传递认证信息。令牌可以存储在客户端的本地存储或 Cookie 中，并在每个请求中发送给服务器进行验证。在身份认证领域中，常见的令牌有三种，分别是访问令牌、刷新令牌与身份令牌。
- 会话是一种服务器端的状态保持机制，用于跟踪用户的认证状态。在会话中，服务器会为每个用户分配一个唯一的标识符（通常是一个会话 ID），并将用户的认证信息保存在服务器上。用户在认证后，服务器会将会话 ID 发送给客户端，并在后续的请求中使用该会话 ID 来验证用户的身份。一般来说，这种会话 ID 都是通过 Cookie 发送给客户端的。

选择使用令牌还是会话取决于具体的应用需求和安全策略。令牌适用于分布式系统和无状态的 API 认证，而会话适用于传统的 Web 应用和需要服务器端状态管理的场景。

在上面介绍令牌与会话时，都提到了 Cookie，这是因为 Cookie 是实现会话的一种常用方式。Cookie 是一种存储在客户端的小型文本文件，用于存储用户的认证状态和其他信息。在身份认证中，Cookie 通常用于存储会话 ID，以及其他一些用户信息。总之，Cookie 在身份认证过程中扮演了非常

重要的角色。下面来看两个例子。

1. 如何推测一个网站使用了令牌还是会话

前面我们讲到，令牌是一种无状态的认证方式，而会话则是一种有状态的认证方式。那么，如何推测一个网站使用了令牌还是会话呢？

虽然有些网站是两者同时使用，但是我们还是可以通过一些方法来推测网站使用的是令牌还是会话。我们可以从以下几个方面来推测该网站主要使用了令牌还是会话。

（1）首先，在该网站上登录，然后通过浏览器的开发者工具查看请求头中是否有 Cookie 字段，以及 Cookie 字段中是否有名称中包含 Session 或 JSESSIONID 的字段，如果有，那么该网站使用的是会话，否则使用的是令牌。

（2）通过浏览器的开发者工具查看请求头中是否有 Authorization 字段，如果有，那么该网站使用的是令牌，否则使用的是会话。

令牌和会话各有优缺点，我们可以根据自己的需求来选择使用哪一种认证方式。

1）令牌的优缺点

- 令牌的优点：不需要服务器端存储，节省服务器端的资源。
- 令牌的缺点：令牌的有效期一般比较长。如果令牌被盗用，攻击者可以在较长时间内使用该令牌，从而造成更大的损失。

除非使用复杂的技术，一般来说，令牌在有效期内是无法被撤销的。所以很多站点对颁发的访问令牌给予了较短的有效期，比如 1 个小时，或者 5 分钟。这样，即便令牌被暴露，攻击者也只能在有效期内使用该令牌，有效期过后，攻击者就无法再使用该令牌，从而减少了损失。但为了用户体验更好，不要求用户频繁地输入登录凭据，很多网站会在令牌过期前自动刷新令牌，这就是很多网站会同时颁发访问令牌和刷新令牌，并且将刷新令牌的有效期设置得比访问令牌长的原因。

2）会话的优缺点

- 会话的优点：会话的有效期一般比较短，如果会话被盗用，那么攻击者只能在会话有效期内使用该会话，有效期过后，攻击者就无法再使用该会话，从而减少了损失。
- 会话的缺点：会话需要服务器端存储，因而要占用服务器端的资源。

如今的网站会话普遍使用了 Cookie 来保存会话编号，以便让服务器通过会话编号查找用户信息。这样，会话的有效期就取决于 Cookie 的有效期。如果 Cookie 的有效期设置得比较长，那么会话的有效期也就比较长，反之，如果 Cookie 的有效期设置得比较短，那么会话的有效期也就比较短。其中有一种特殊的 Cookie，叫作 Session Cookie。Session Cookie 的有效期设置为浏览器会话结束时失效，也就是说，只要关闭浏览器，会话就结束了。重新打开浏览器时，需要重新登录才能进入登录状态。很多对安全性要求比较高的网站，都设置了这种 Cookie。

了解了这个特性之后，其实我们又多了一种从观察网站的登录状态的表现来推测网站使用的是令牌还是会话的方法。如果我们在网站上登录之后关闭浏览器，再立即重新打开该网站，如果还处于登录状态，那么该网站使用的是令牌（因为令牌的有效期不会立刻过期），否则使用的是会话。

关于会话与令牌的选择，还有一个很有意思的现象。一般传统的 MVC 网站会使用基于 Cookie

的会话；而新兴的单页应用则偏爱基于 Local Storage 的令牌。尽管完全可以使用 Local Storage 来存储会话信息，也完全可以使用 Cookie 来保存令牌，但实际上这样做的网站并不多。

在本书后面会详细讲解单点登录的实现。在单点登录中，会使用多个网站之间的会话共享，这时会话就需要存储在服务器端，而令牌则不需要。所以，单点登录一般都是基于会话的。

而且，单点登录会有一个集中的身份认证中心，这个身份认证中心会有一个单独的域名，这个域名一般是不会被其他网站使用的，所以单点登录中的身份认证中心一般都是基于 Cookie 的会话。尽管身份认证中心自己一般使用基于 Cookie 的会话，但是它也会提供基于令牌的身份认证服务，以便其他网站或者应用使用。很多网站应用只是纯前端应用，没有后端（或者有后端，但却并不想再维护一个会话状态，而将所有会话都委托给身份认证中心），这时就只能使用基于令牌的认证方式了。于是形成了如图 1-8 所示的情况。

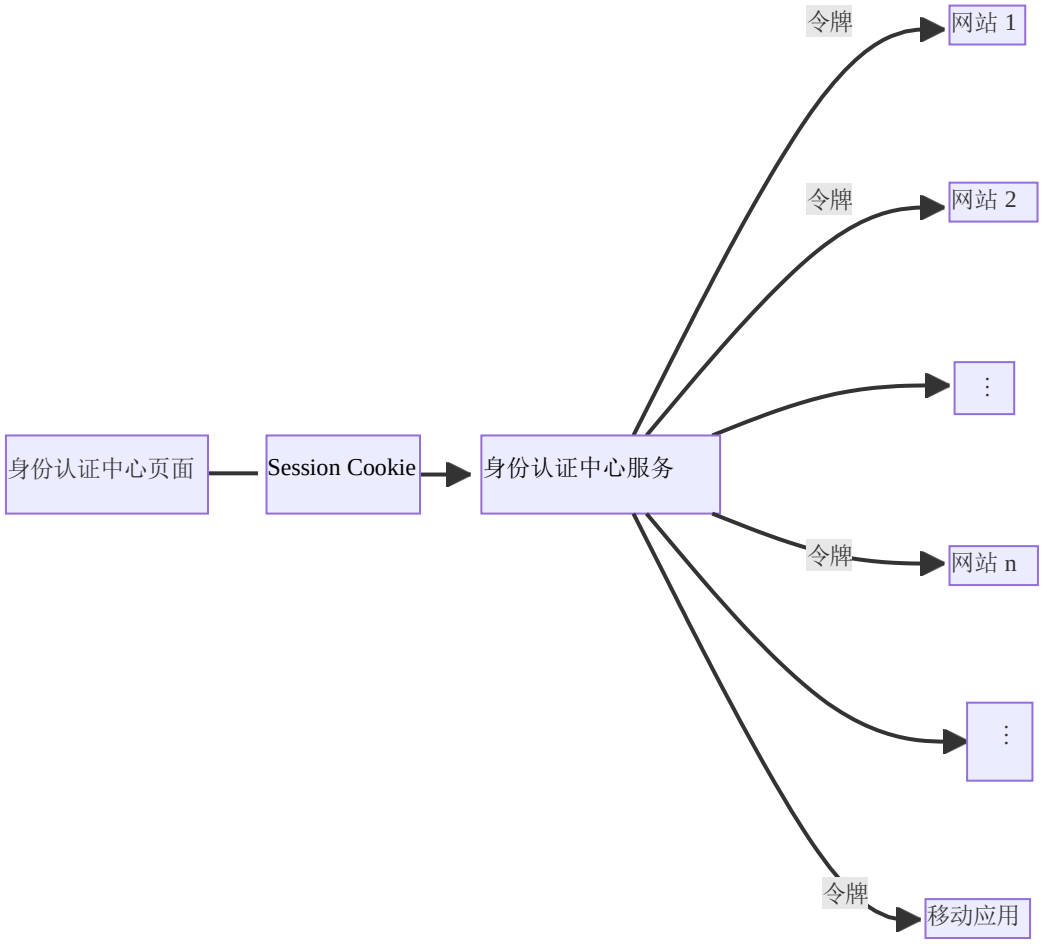

图 1-8　Cookie 和 Token 同时使用

2. 如何利用 Cookie 绕过登录进入登录状态

在前面的章节中，我们讲解了 Cookie 的基础知识，现在来灵活运用一下这些知识，解决一些自动化测试的问题。

有的团队使用 Cypress 对站点进行自动化测试，但是很多场景是用户在登录状态下进行的，需要

首先解决登录问题。尽管预先注册账号，并将密码保存在一个安全的地方，在运行测试时动态获取，然后操作页面模拟用户登录。但是一般登录页面都会带上防机器人的脚本，直接阻止这样的自动化登录。比如有的网站使用了 WAF 服务商提供的 JS 脚本，其工作原理可以参考笔者的一篇博文[1]。

这时，我们可以考虑使用 Cookie 绕过登录，直接进入登录状态。这里有一个前提，就是我们需要有一个登录状态的 Cookie。该方案要求正常人类用户先手动登录站点，获得 Cookie，然后在自动化脚本中直接使用该 Cookie 访问目标站点，从而直接进入登录状态。这个方案听上去有点像黑客行为，因此，在最终使用该方案前，我们也可以考虑一下其他的解决方案，在由于各种原因导致其他方案实在不可行时，再来考虑使用 Cookie 绕过登录。

1）可能的解决方案

（1）使用特殊的 HTTP 标头。在目标站点的代码中做一个小改动。当检测到请求头中有一个特殊的字段，并且其值为允许的值时（比如 X-Internal=a-jwt-token），直接放行，不要应用机器人检测。这个方案改动量很小，比如：

```
    // 如果头部 "X-Internal" 存在并且包含有效的访问令牌，则绕过机器人检测分析
    if (request.headers["X-Internal"]) {
        if (await validateInternalAuthToken(request, request.headers["X-Internal"]
[0].value, lambdaContext, requestId))
            return request;
        else
          return buildBlockResponse(request, requestId);
    }
```

但是它要求允许的测试客户端小心地保存这个秘密值。不过，即使客户端保存得很好，但终究需要明文传输，总有可能会被暴露。

（2）IP 白名单。这种方法可以轻松解决问题，但是在服务器端维护这个白名单会增加工作量，尤其当 IP 可能发生变化时。

（3）给目标站点添加多个入口域名，在内部域名访问时，去除机器人检测。可以给站点设定一个公开域名和一个内部域名，如图 1-9 所示。内部域名只允许通过 VPN 访问，相当于内网环境，不用检测机器人。对于公开域名，可以通过某些 DNS 服务来动态添加机器人检测脚本（比如 Cloudflare 这样的提供商）。自动化测试时使用内部域名。

Cloudflare 提供这样的服务，详见其官网文档：https://developers.cloudflare.com/bots/reference/javascript-detections。然而，如果公司没有使用这样的云，很可能就不能这样做了。

另外，如果公司的自动化测试运行在云上的 Elastic Runner 上，也不一定能够使用 VPN。

如果以上几种方案都不可行，那么可以尝试使用 Cookie 绕过登录，前提是目标网站没有使用 Session Cookie，而是使用了一个有效期较长的 Cookie。比如知乎网站、Bilibili 网站都采用了有效期较长的 Cookie，因为通过有效期较长的 Cookie 来减少用户登录的次数可以提升用户体验，同时开发量又很小。

要实现这个方案，需要先手动登录目标站点，然后获取 Cookie，在自动化脚本中使用该 Cookie 来访问目标站点，从而直接进入登录状态。说起来简单，实现起来的开发量并不小，但是该方案一旦实现之后，也可以用在其他的场景，这样边际成本就很小了。

[1]可以在知乎上搜索：网络应用防火墙（WAF）防机器刷流量的工作原理，以登录举例。

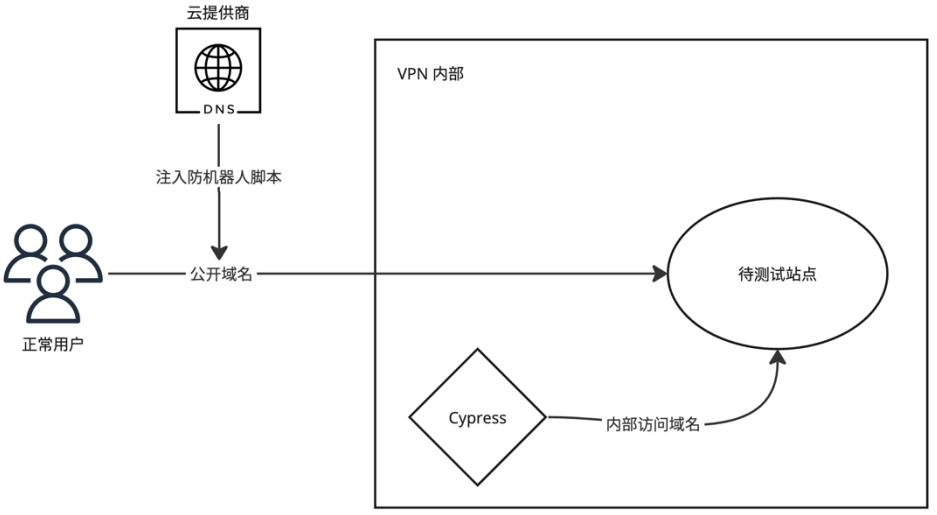

图 1-9　给站点设定一个外部域名和一个内部域名

其实，在笔者的两篇博文[1]中，曾经提到的“叽歪同步工具”用于自动化发布知乎专栏，就是采用了这个方案。让我们来仔细查看它的架构和核心代码。

如何获取用户 Cookie？

如果是浏览器登录，一般可以通过 F12 键打开开发者工具栏看到 Cookie，如图 1-10 所示。

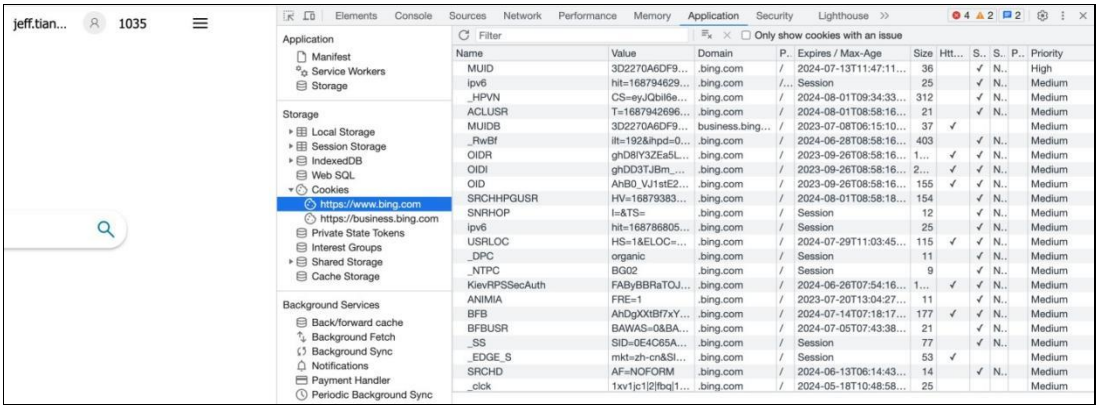

图 1-10　查看 Cookie

对于自动化测试来说，正常登录后，将 Cookie 手动保存下来即可。对于其他场景，可能需要自己做一个站点，来收集用户的 Cookie。笔者曾经录过的一个视频[2]中有提到我开发的一个收集知乎 Cookie 的站点。

2）架构图

整体架构图如图 1-11 所示。

[1]可以在知乎上搜索：如何优雅地将 Markdown 文档同步到知乎专栏（叽歪同步工具介绍），以及如何优雅地将 Markdown 文档同步到知乎专栏（叽歪同步工具介绍之有头模式）。

[2]可以在知乎上搜索：“扫二维码登录一定安全吗？”查看该视频。

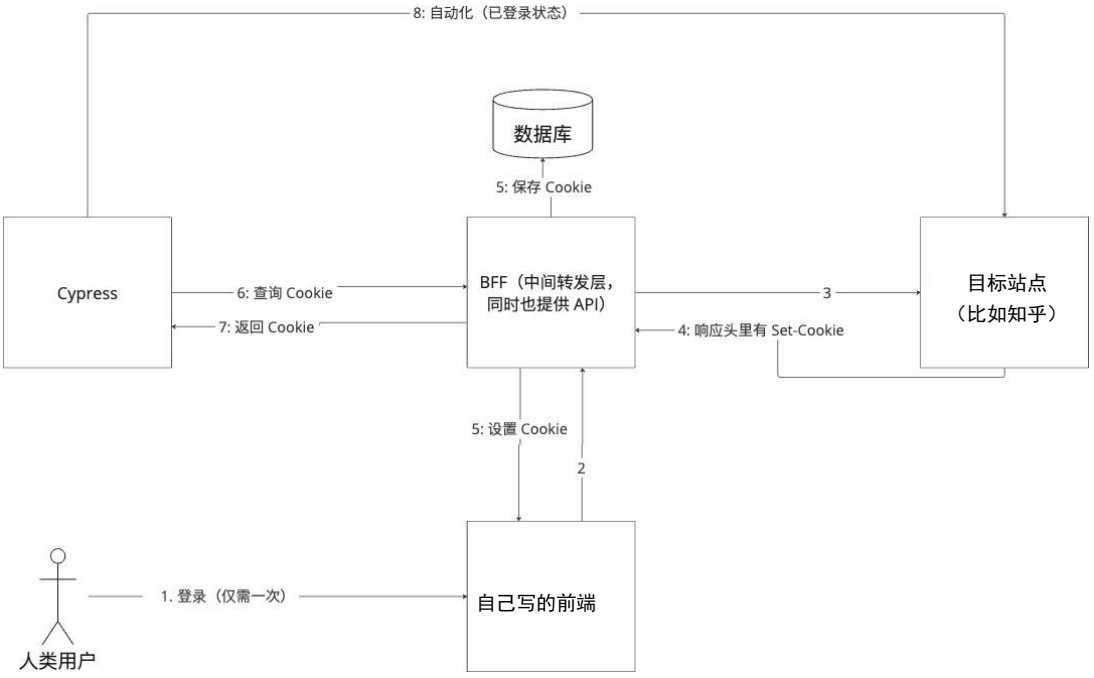

图 1-11　收集 Cookie 的架构

3）核心代码

Cypress 启动时，禁用浏览器安全：

```
const {defineConfig} = require("cypress");

module.exports = defineConfig({ chromeWebSecurity: false, /* 其他设置省略 */ })
```

运行测试用例前，获取 Cookie：

```
describe("feature", ()  =>  {
    let cookie
    before(()  =>  {
        cy.request(
            'POST', THE_PROXY_API_FOR_GET_COOKIE,
            {
                "query": "a graphql request body",
                "variables": {"key": "user-cookie-60808105033728"}
            }
        ).then((response)  =>  {
           cookie = JSON.parse(response.body.data.cookie);
        })
     })

    it('should login automatically and do stuff', ()  =>  {
       login(cookie);
       //...
    })
})
```

使用 Cookie 登录：

```
    function login(cookie) {
       // 一开始未登录
       cy.visit('https:  www.the.site')
       cy.clearCookies();
       Cypress.Cookies.debug(true)     // Cypress 替换 Cookie 时，会打印日志显示出来

       JSON.parse(cookie).data.map(item  =>  {
          const cookie = Cookie.parse(item);
          cy.setCookie(cookie.key, cookie.value, {
            expiry: (!!cookie.expires  && cookie.expires !== 'Infinity') ? new
Date(cookie.expires).getTime() / 1000 : undefined,
            path: cookie.path,
            domain: cookie.domain ? cookie.domain : undefined,
            secure: cookie.secure,
          })
       });
       // 现在是登录状态了
       cy.visit('https:  www.the.site')
    }
```

1.4.2 什么是 SAML、OAuth 2.0 和 OIDC

讲到身份认证，总能碰到一些常见的术语。首先，从大的概念上讲，会碰到几种常见协议的名称，它们分别是 SAML（Security Assertion Markup Language、OAuth（Open Authorization）2.0 和 OIDC（OpenID Connect）。

- SAML：一种基于 XML 的开放标准，用于在身份提供者和服务提供商之间交换认证和授权信息。它主要用于企业间的单点登录（Single Sign On，SSO）和身份提供者（Identity Provider，IdP）与服务提供商（Service Provider，SP）之间信任关系的建立。SAML 解决了企业间身份集成和信任关系建立的问题。
- OAuth 2.0：一种开放标准，用于授权第三方应用访问受保护资源的框架。OAuth 提供了安全且可扩展的身份认证和授权机制。它允许用户授权第三方应用访问它们在其他应用中存储的资源，而无须共享它们的凭据。OAuth 2.0 广泛应用于 Web 和移动应用的身份验证和授权场景。
- OIDC：建立在 OAuth 2.0 之上的身份认证协议，提供了基于令牌的身份验证和用户信息交换的能力，由 OpenID 基金会开发。

在身份认证领域，以上几个标准和协议被广泛应用于认证和授权的流程。其中 SAML 主要使用在单点登录场景，我们留到后面的 1.4.6 节和其他单点登录协议一并介绍，下面分别详细介绍 OAuth 2.0 和 OIDC。

1. OAuth 2.0

对于 OAuth 2.0 来说，“第三方”是一个关键字，也就是说，如果你的应用既不准备使用任何第三方能力，也不准备为任何第三方赋能，不会被任何第三方来调用，那么这样的孤立应用程序（Siloed Application）是完全不需要使用 OAuth 开放标准的。

提　示

如果你确定需要使用 OAuth 协议，那么在动手前还要想一想，你的应用是第三方还是授权服务器。如果是第三方，那么你需要利用 OAuth 客户端来和 OAuth 服务器进行交互；如果是授权服务器，那么你很可能需要实现一个 OAuth 服务器。

OAuth 允许用户不共享凭据的情况下授权第三方应用访问它们在其他应用中的资源，这是通过用共享访问令牌替代共享凭据（比如密码）来实现的。产生这个访问令牌的方式涉及不同的许可类型（或者称为授权模式）。在后文中，我们将举例逐一介绍不同的许可类型，但是现在让我们总体了解一下 OAuth 2.0 的授权流程，如图 1-12 所示。

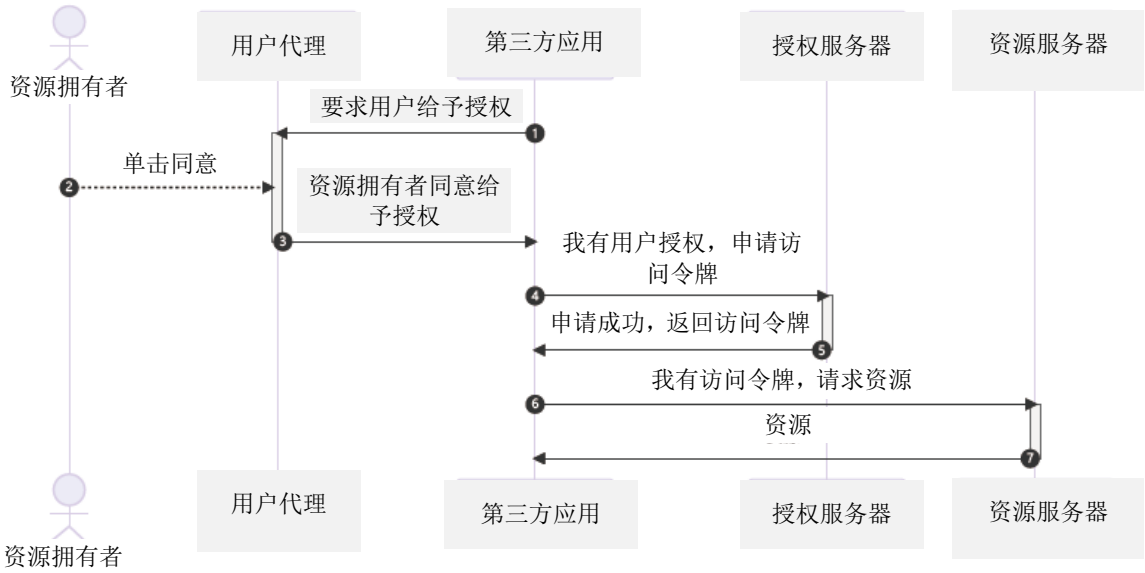

图 1-12　OAuth 2.0 的授权流程

在图 1-12 中，提到了如下几个参与方，它们也是常见术语的一部分。

- 第三方应用（Third-Party Application）：需要得到资源持有者授权访问其资源的那个应用。也经常被称为客户端或者请求方，它从授权服务器请求访问令牌，并且可以携带访问令牌从资源服务器访问用户的资源。
- 资源持有者（Resource Owner）：拥有授权权限的人，是第三方应用的用户，拥有资源并可以授权他人访问其拥有的资源。
- 授权服务器（Authorization Server）：能够根据资源持有者的意愿提供授权（授权前进行了必要的认证过程）的服务器。
- 资源服务器（Resource Server）：能够提供第三方应用所需资源的服务器，它与认证服务器可以是相同的服务器，也可以是不同的服务器。它接收令牌，并且在令牌验证有效时返回资源。
- 用户代理（User Agent）：资源持有者用来访问服务器的工具。对于人类用户来说，这通常是浏览器，或者原生应用。对于机器用户，即在微服务的互相调用的认证场景中，一个服务经常会作为另一个服务的用户，此时的用户代理就是 HttpClient、RPCClient 等。

如果你现在觉得上面的名称有些抽象，没关系，在后面的具体示例中会有案例说明，到时候就一清二楚了。

2. OIDC

OIDC 是一个基于 OAuth 2.0 的身份验证协议，它提供了在 Web 应用中验证和获取用户身份信息的机制，是 OAuth 2.0 的超集，即 OIDC=授权协议+身份认证。OIDC 在 OAuth 2.0 的基础上添加了标准化的身份验证流程和用户信息交换。它被广泛应用于 Web 应用程序中的用户身份验证和授权，并且可以提供单点登录的能力，使用和 OAuth 2.0 相同的授权流程。

OIDC 是 OAuth 2.0 的一个扩展，为 OAuth 2.0 协议提供了一个标准的身份验证流程。通过 OIDC，客户端可以通过 OAuth 2.0 的授权流程获取用户的身份令牌，并使用该令牌验证用户的身份和获取用户的身份信息。它利用了 OAuth 2.0 提供的基础，所以它和 OAuth 2.0 一样，是一种基于 REST、JSON 和 API 的系统。

如果说 OAuth 2.0 是一层非常厚的基于令牌的授权协议，那么 OIDC 是一层非常薄的基于令牌的身份认证协议，它们之间的关系如图 1-13 所示。

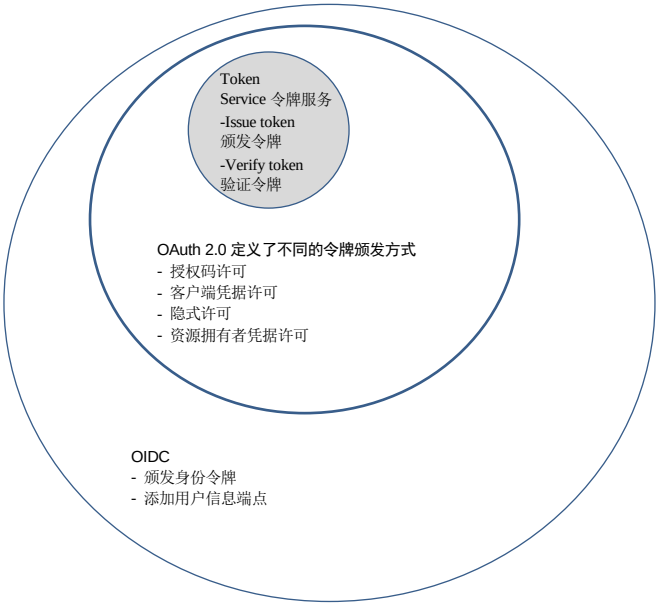

图 1-13　OIDC 和 OAuth 2.0 的关系

OIDC 定义了三种主要角色，分别说明如下：

- EU（End User），即最终用户，也就是使用我们的应用的用户。
- RP（Relying Party），即依赖认证服务的第三方，也就是我们的应用。
- OP（OpenID Provider），即 OIDC 服务提供者，也就是我们的认证服务器。

在前言中讲过，笔者认为身份认证不仅仅局限于登录注册。OIDC 基于授权服务器执行的验证用户的过程简化了确认用户身份的方式，并提供了类 RESTful 接口的互操作方式，用于获取用户资料。这种开放互联的设计使得不同的系统能够轻松集成，使数字化系统的搭建变得像搭建积木一样简单高效。OIDC 为应用和网站开发者赋能，使他们能够启用登录流程并在基于网页、移动端和

JavaScript 的客户端之间验证用户身份，此外，OIDC 支持诸如身份数据加密、身份提供者发现以及会话注销等扩展功能。对于开发者来说，OIDC 安全可靠地回答了一个问题：“正在使用当前浏览器或者移动应用的个体的数字身份是什么？”。OIDC 的优势在于，它使开发者不再需要负责设置、存储和管理密码，而这些任务非常困难，凭据泄露事件频繁发生就是最好的证明。因此，将这些烦琐的任务交给 OIDC，开发者可以更加专注于业务逻辑。图 1-14 展示了 OIDC 连接协议套件的示意图[1]，其中的最小核心组件已足以实现系统之间的开放互联。而将整个套件全部包含进来，则形成了一个非常理想的身份认证平台。

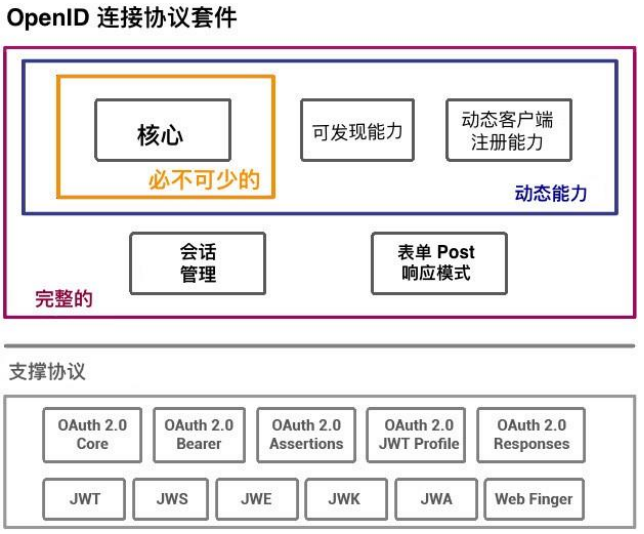

图 1-14　OIDC 连接协议套件示意图

前面反复提到 OIDC 是基于 OAuth 2.0 的，那么在日常开发实践中如何快速区分它们呢？这里有一个不十分严谨但非常高效的做法：你可以简单地认为，OAuth 2.0 提供访问令牌（Access Token），而 OIDC 的最小核心组件额外提供身份令牌（ID Token）。

- 访问令牌：表示用户身份和访问权限的令牌，用于向资源服务器请求受保护资源的访问。
- 身份令牌：在 OIDC 中使用的令牌类型，包含有关用户身份的信息，用于进行身份验证和用户信息交换。

OIDC 致力于为授权过程中的主体（或者说用户）提供更多的保证。身份令牌，正如名称所暗示的，它聚焦于提供身份相关的信息，它也是 JWT 格式的。

除客户端凭据许可类型外，其他的许可类型会同时提供访问令牌和身份令牌。我们后面会详细介绍在 OIDC 中的许可类型，不同许可类型对访问令牌和身份令牌的支持情况表如表 1-2 所示。

表 1-2　不同许可类型对访问令牌和身份令牌的支持情况表

许可类型	访问令牌	身份令牌
授权码许可	支持	支持
隐式许可	支持	支持

[1] https://openid.net/developers/how-connect-works/

（续表）

许可类型	访问令牌	身份令牌
密码许可	支持	支持
客户端凭据许可	支持	不支持
设备许可	支持	支持

访问令牌不需要被使用方解析，可以使用不透明令牌，而身份令牌是一个 JSON Web Token（JWT），属于结构化令牌，需要使用方对令牌进行解析，从中获取用户的身份信息，处理用户的登录状态等逻辑。尽管一个结构化令牌的载荷可以自定义，但是 OIDC 规范中定义了一些标准的字段，用于表示用户的身份信息，以及身份令牌的有效期等信息。如果要颁发一个身份令牌，则以下 5 个 JWT 声明参考是必需的：

（1）iss，全称 issuer，是令牌的颁发者，其值就是 OpenID 提供者（OpenID Provider，OP）的 URL，这个身份认证服务，在 OpenID 规范里，也被称为 OpenID 身份提供者。

（2）sub，全称 subject，是令牌的主题，其值就是终端用户（End User，EU）的唯一标识，比如用户的 ID。

（3）aud，全称 audience，是令牌的受众，其值就是依赖认证服务的第三方应用（Relying Party，RP）的 app_id。

（4）exp，全称 expiration time，是令牌的过期时间，其值就是一个时间戳，表示令牌的过期时间。

（5）iat，全称 issued at，是令牌的颁发时间，其值就是一个时间戳，表示令牌的颁发时间。

一个生成身份令牌的 Java 代码示例如下（后面的示例中还会展示如何用 Node.js 来实现同样的逻辑）：

```
private String generateIdToken(String appId, String user) {
//密钥
   String sharedTokenSecret="hello-auth"; Key key = new SecretKeySpec(
     sharedTokenSecret.getBytes(),
   //采用 HS256 算法
   SignatureAlgorithm.HS256.getJcaName()
);

//ID 令牌的头部信息
Map<String, Object> headerMap = new HashMap  (); headerMap.put("typ", "JWT");
headerMap.put("alg", "HS256");

Map<String, Object> payloadMap = new HashMap  ();
//ID 令牌的主体信息
payloadMap.put("iss", "http:  localhost:8081/");
payloadMap.put("sub", user);
payloadMap.put("aud", appId);
payloadMap.put("exp", 1584105790703L);
payloadMap.put("iat", 1584105948372L);
```

```
    return Jwts.builder()
              .setHeaderParams(headerMap)
              .setClaims(payloadMap)
              .signWith(key, SignatureAlgorithm.HS256)
              .compact();
    }
```

第三方解析身份令牌的示例代码如下（在后面讲解 JWT 的过程中也有 Node.js 实现，而身份令牌也是一个 JWT）：

```
    private Map<String, String> parseJwt(String jwt) {
    // 密钥
    String  sharedTokenSecret="hello-auth";
    // HS256 算法
    SignatureAlgorithm.HS256.getJcaName()
    );
    Map<String, String> map = new HashMap<String, String>(); Jws<Claims> claimsJws =
Jwts.parserBuilder()
    .setSigningKey(key)
    .build()
    .parseClaimsJws(jwt);
    // 解析 ID 令牌主体信息
    Claims body = claimsJws.getBody(); map.put("sub", body.getSubject()); map.put("aud",
body.getAudience()); map.put("iss", body.getIssuer());
    map.put("exp", String.valueOf(body.getExpiration().getTime())); map.put("iat",
String.valueOf(body.getIssuedAt().getTime())); return map;
    }
```

这里再特别强调一下，第三方只需要验证身份令牌的签名是否正确就可以确定身份令牌的真实性。这是唯一需要的合法性校验，随后的解析只需要保留 JWT 中的载荷部分即可。

OIDC 在很多方面都可以看成是现代化的 SAML。SAML 是一种基于 XML 的旧协议，在 2000 年开始流行起来。它是跨组织使用断言形式进行信息联邦的事实标准。从用户角度来看，SAML 的使用比起以前更少了。

以上介绍的 SAML、OAuth 2.0 和 OIDC 是身份认证领域常用的标准和协议，了解它们的特点和用途对于设计和实现身份认证系统至关重要。

1.4.3　许可类型

在身份认证和授权过程中，许可（Grant）类型用于确定用户授权的方式和权限范围。许可类型定义了用户如何获取访问令牌（Access Token）以及使用令牌所拥有的权限，它们之间最大的不同在于获取访问令牌的方式不一样，但是整体的流程是相似的。图 1-15 是 OAuth 2.0 的概览流程，不同的授权许可类型只是在其中的个别步骤上有一些差异。

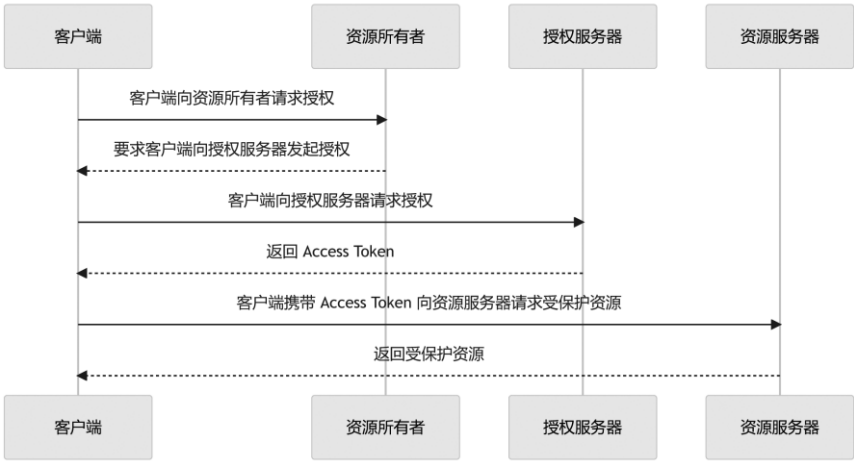

图 1-15　OAuth 2.0 的概览流程

OAuth 2.0 中常见的许可类型如表 1-3 所示。后面将逐个详细介绍。

表 1-3　OAuth 2.0 中常见的许可类型

授权许可类型	获取访问令牌的方式
授权码许可	通过授权码（code）换取用户的访问令牌（access token）
客户端凭据许可	通过第三方调用客户端的密钥对（client_id 和 client_secret）来换取客户端级别的访问令牌
隐式许可	通过嵌入浏览器中的第三方调用客户端的标识（client_id）来换取客户端级别的访问令牌
资源拥有者凭据许可	通过资源拥有者的用户名和密码换取用户的访问令牌，因此也叫密码许可类型

1. 授权码许可

对于普通用户来说，这是最常见的授权方式，可能每天都会和这个授权许可打交道。比如微信扫码登录第三方网站，就是这种授权许可模式。

作为普通用户，不用关心其中的细节，而对于技术人员，可以观察一下：当你扫码后，手机上会询问是否允许第三方网站获取你的信息。一旦你点击“允许”，浏览器上的二维码页面会重定向至第三方网站，并携带一个微信颁发的授权码。你可以在浏览器的地址栏中看到这个授权码，它以查询字符串的明文方式出现。

这个授权码是和你的身份绑定的，也就是说，微信可以通过这个授权码定位到你的身份。当然，你基本上不用担心这个授权码被泄露，因为它的有效时间很短，而且只能使用一次，使用一次后立即失效。

这个授权码在浏览器的地址栏一闪而过，立即就被第三方网站消费掉了。第三方网站拿到你的授权码，就会使用这个授权码去向微信换取你的访问令牌以及身份令牌。再次说明一下，不用太担心授权码的泄露，因为即使攻击者拿到了授权码，并能够抢在你授权的第三方网站去微信端换取你的身份信息，也不能成功。因为在使用授权码换取访问令牌和身份令牌时，还需要提供第三方网站的客户端级别的访问令牌，而这个令牌是需要第三方提前使用其客户端密钥对（client_id 和 client_secret）来获得的。

这个流程可以总结为如图 1-16 所示的时序图。

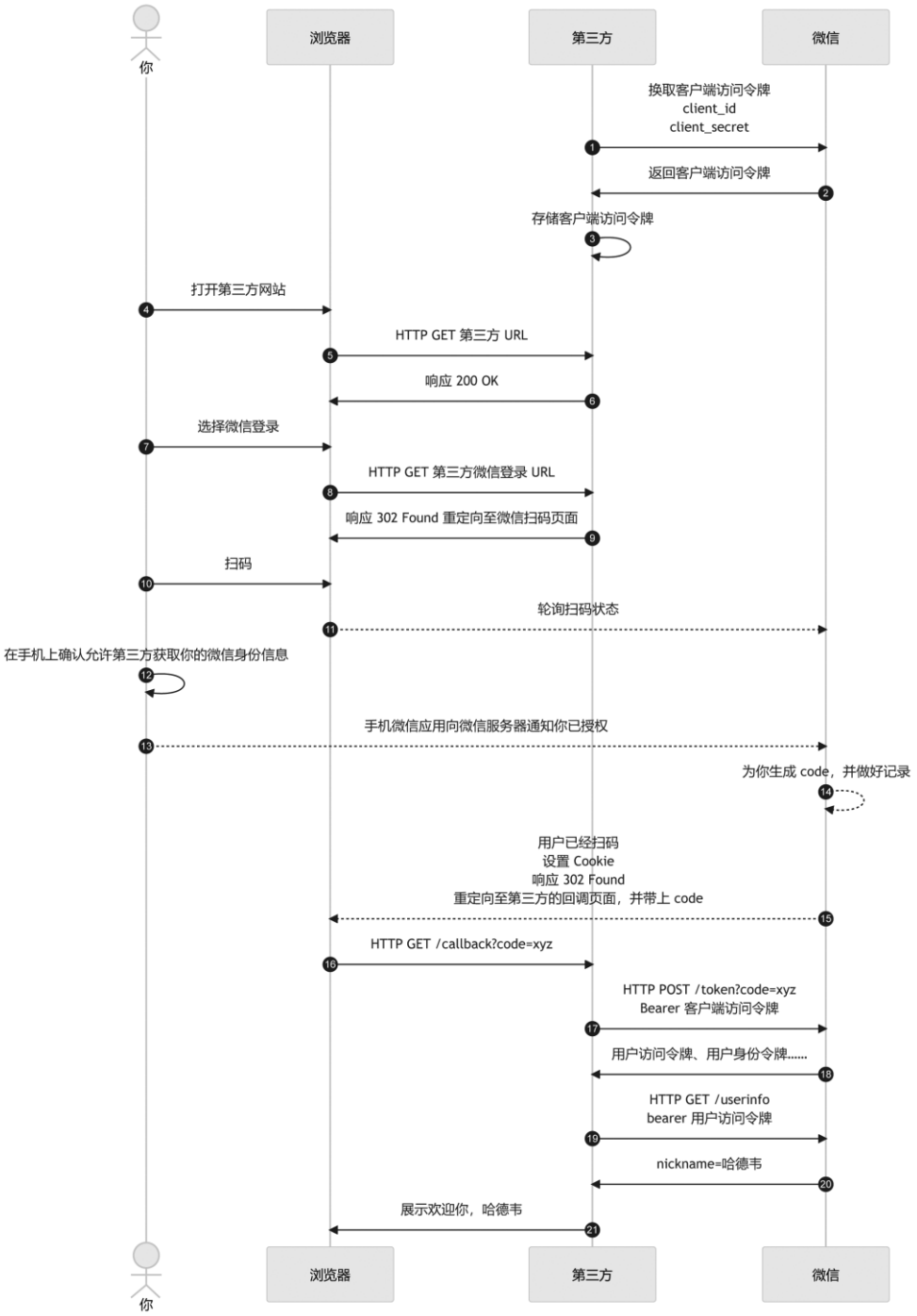

图 1-16　授权码许可时序图

在 OAuth 2.0 的标准协议中，有一些更抽象的名称，比如资源拥有者，在图 1-16 中就是你，第三方网站需要你的个人信息，你是你个人信息的拥有者；而授权服务器就是微信，由它来颁发令牌；

资源服务器也是微信，因为第三方网站想要使用你在微信端的头像、昵称等，这些就是资源，保存在微信中，但拥有者是你；第三方应用是允许你使用微信登录的网站；而网页浏览器和手机上的微信应用都是用户代理。

尽管在前面的 OAuth 2.0 授权流程中分别列出了资源拥有者、用户代理、第三方应用、授权服务器以及资源服务器，但是在这个场景（网站的微信扫码登录）中，用户代理有两个具体的实例，分别是网页浏览器和手机上的微信应用；而授权服务器和资源服务器是同一个，即微信。在后面更多的具体实例中，我们会碰到更多的场景，在有些场景中，授权服务器和资源服务器是分开的，而且在很多场景中，用户代理都是网页浏览器的一个实例。

综上所述，授权码许可类型（Authorization Code Grant Type）是一种常见的许可类型，它用于 Web 应用的身份认证和授权流程。在授权码许可类型中，用户首先通过身份提供者（Identity Provider，IdP）登录（在上述微信扫码登录网页的例子中，IdP 就是微信），并授权给客户端应用访问其资源。一旦用户授权成功，身份提供者将生成一个授权码并将其发送给客户端应用。然后，客户端应用使用授权码向身份提供者请求访问令牌，进而获取用户资源的访问权限。

授权码许可类型的优势在于客户端应用不直接获取访问令牌，而是通过授权码进行交互，从而增加了安全性。此外，授权码许可类型还支持客户端的机密性，并且授权码只能用于特定的客户端应用。

除了用于网页浏览器外，授权码许可也可用于手机应用。网页的回调通过 HTTP 302 重定向实现，而在手机上的回调通过 Scheme URL 来实现。不过，在手机上，恶意软件可以通过 Scheme URL 冒充真实的应用而截获授权码，这是一个安全隐患。为了解决这个问题，OAuth 2.0 引入了 Proof Key for Code Exchange（PKCE）协议，这个协议的目的是防止授权码被截获。后面可以通过详细的例子来说明这个协议，在详细了解 PKCE 之前，让我们先来看一看隐式许可类型，并且详述一下它的安全性问题，以及为什么可以通过启用 PKCE 的授权码许可类型来替代隐式许可类型（Implicit Grant Type）。

2. 隐式许可

隐式许可类型是另一种常见的许可类型，它通常用于单页面应用（Single Page Application，SPA）或移动应用的身份认证和授权流程。在隐式许可类型中，用户直接在浏览器中进行身份认证并授权，身份提供者会直接向客户端应用返回访问令牌。因此，隐式许可类型不涉及授权码的交换过程。

1）为什么会有隐式许可类型

既然不推荐该许可类型，那么它为什么会出现呢？这是因为在 OAuth 2.0 的初期，隐式许可类型是唯一一种支持单页面应用的许可类型。在 OAuth 2.0 应用的初期，单页面应用在浏览器中受到的限制比较大，比如 JavaScript 不能访问浏览器的浏览历史，也不能访问浏览器的本地存储。并且，那时候很多服务器也不接受跨站 POST 请求，导致单页面应用没法通过/token 端点获取到令牌，从而使得整个授权码流程不能在单页面应用上正常运行。

我们现在假设要开发一个名为“哈德韦”的单页面应用，它使用隐式许可类型接入了微信登录，那么流程如图 1-17 所示。

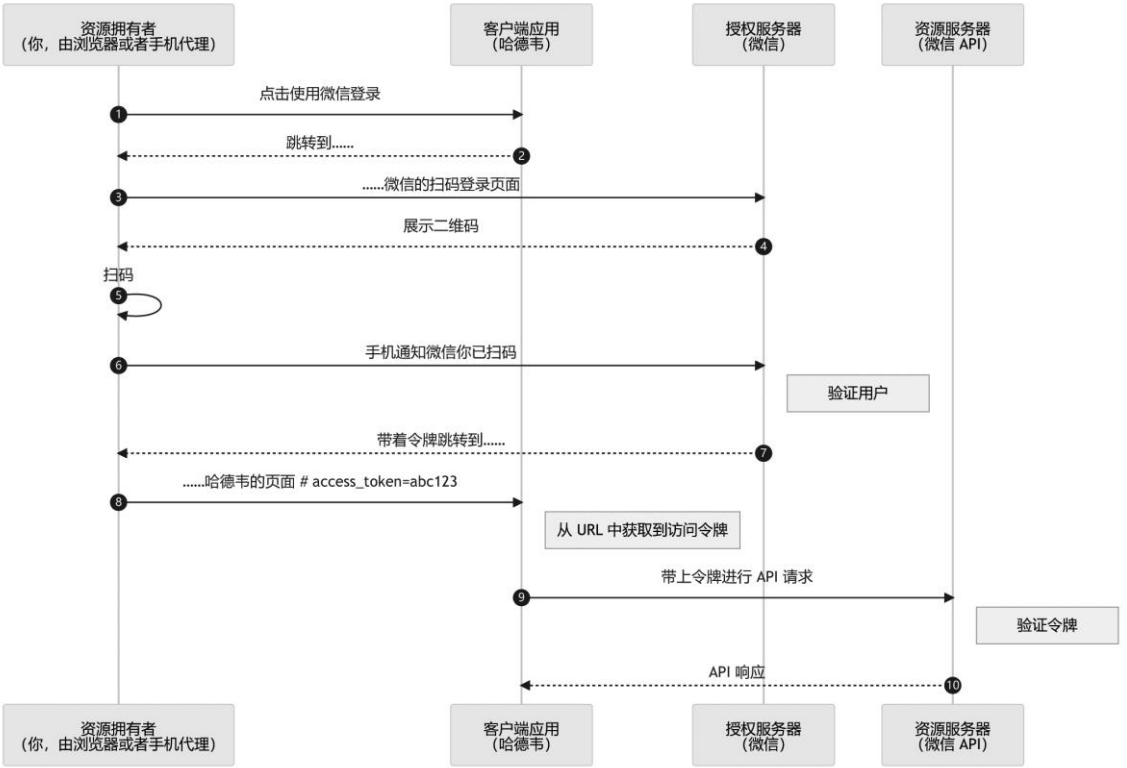

图 1-17　隐式许可时序图

注意，当你认证完成，授权服务器（微信）在响应回客户端应用（哈德韦）时，会在 URL 上直接带上令牌。这个时候，微信并不能确认是否真的是浏览器（期待的接收者）接收到了令牌，更糟的是，多数浏览器或者插件支持同步浏览历史，这会导致令牌泄露。为了缓解这个问题，一般支持隐式流程的授权服务器，都会通过#token=xxx 的方式将令牌返回到客户端应用（在授权链接中通过 response_mode=fragment 指定），因为#后面的部分不会被发送到服务器，仅存在于浏览器端，但是这仍然不够安全。

2）单页面应用的路由模式该如何选择

单页面应用是一种特殊的 Web 应用，它将所有的活动都在一个页面中完成，通过动态加载 HTML、CSS、JavaScript 等资源来实现页面的切换和更新。一般单页面应用框架都设置了两种路由模式供开发者选择，分别是 browser 和 hash。browser 模式使用 HTML5 的 history API 来实现路由，而 hash 模式则是在 URL 中使用#来作为路由标记。笔者认为，browser 模式大家更加熟悉，而且对服务器渲染更加友好，而 hash 模式仅能够在客户端进行路由，并且#后面的内容不会被发送到服务器，因此在很多情况下，笔者都会推荐使用 browser 模式。

不过，如果单页面应用要对接隐式许可，那么推荐使用 hash 模式。或者至少在接收访问令牌时，访问令牌一定要放在#后面，这样可以避免将访问令牌发送到服务器，从而减少暴露面。如果不将访问令牌放在#后面，那么服务器开发者或者运维人员可以从应用的访问日志中查看到访问令牌，在有效期内可以轻易冒充最终用户的身份。

如图 1-17 中的第 8 步所显示的，在隐式许可类型中，令牌是通过 URL 中的#后面的查询字符串来传递的。一个典型的隐式许可的授权链接如下：

```
    https: dev-micah.okta.com/oauth2/default/v1/authorize? client_id=0oapu4btsL2xI0y8y356
    &redirect_uri=http: localhost:8080/callback &response_type=id_token token
&response_mode=fragment
    &state=SU8nskju26XowSCg3bx2LeZq7MwKcwnQ7h6vQY8twd9QJECHRKs14OwXPdpNBI58
&nonce=Ypo4cVlv0spQN2KTFo3W4cgMIDn6sLcZpInyC40U5ff3iqwUGLpee7D4XcVGCVco
    &scope=openid+profile+email
```

而对应的典型的隐式许可的授权响应的跳转链接如下（见图 1-18）：

```
    http: localhost:8080/callback# id_token=eyJraWQiOiI3bFV0aGJyR2hWVmx
&access_token=eyJraWQiOiI3bFV0aGJyR2 &token_type=Bearer
    &expires_in=3600 &scope=profile+openid+email
    &state=SU8nskju26XowSCg3bx2LeZq7MwKcwnQ7h6vQY8twd9QJECHRKs14OwXPdpNBI58
```

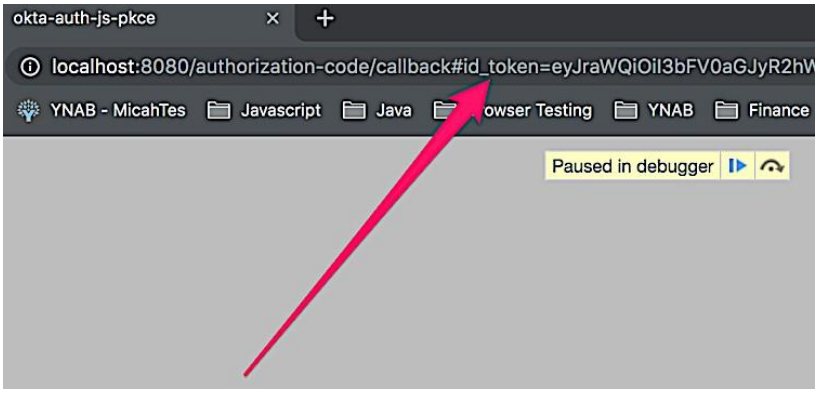

图 1-18　隐式许可跳转链接 URL

如果不用隐式流程，对于单页面应用，那该怎么办呢？其实可以使用 PKCE，它的全称是 Proof Key for Code Exchange，作为授权码流程的扩展，已经被广泛用于手机原生应用。

3）使用 PKCE 让应用更安全

PKCE 有单独的规格说明，它让使用公开客户端的授权码流程更加安全。在理解这一点前，我们先来看看不使用 PKCE 时为什么不够安全。这里涉及公开客户端，也就是在授权服务器的应用列表里，该客户端只有 client id 而没有 client secret。在原生应用场景中，当授权服务器将授权码返回给应用时，它会调用应用的 Scheme URL 来唤起该应用。然而，如果有恶意应用通过注册与合法应用相同的 Scheme URL 来冒充，那么授权码可能会被该恶意应用截获。从而抢在真正的应用之前向授权服务器换取令牌（因为没有 client secret，而 client id 又可以通过网络抓包获取到，所以只需通过 client id 加上授权码即可换取令牌）。

PKCE 流程会在发起整个授权码流程之前，通过动态生成的秘密值进行一些准备工作，并且在授权码流程结束附近处增加验证环节，让确保接收授权码的应用能够证明自己是发起整个流程的合法应用。

具体来说，应用在发起流程前生成一个随机值，称为 code verifier。应用将该值 hash 之后，即成为 code challenge。然后，应用像普通的授权码流程一样发起整个流程，只是在查询字符串中带上

了 code challenge，一并发往授权服务器。授权服务器会存储 hash 后的 code challenge，在后面校验时会用到。当用户验证成功后，授权服务器把授权码发回应用。

应用收到授权码会再次发起请求以换取令牌，但是要带上 code verifier（虽然不需要传递固定的 client secret 了，但是需要传递一个动态的 secret，code verifier 就相当于动态的 secret）。现在授权服务器可以 hash 这个 code verifier，并且和它之前存储起来的 hash 过后的值进行比较，如果一致，验证就通过，正常返回令牌。这是一个非常有效的使用动态密钥代替静态密钥的机制。

这个流程最初只被运用在原生应用场景，因为那时候的浏览器和多数的授权服务器都不支持 PKCE。但是现在这个情况已经改变了，浏览器和授权服务器都可以支持 PKCE。

在前面的隐式许可的例子中，如果改为 PKCE，新的流程图如图 1-19 所示。

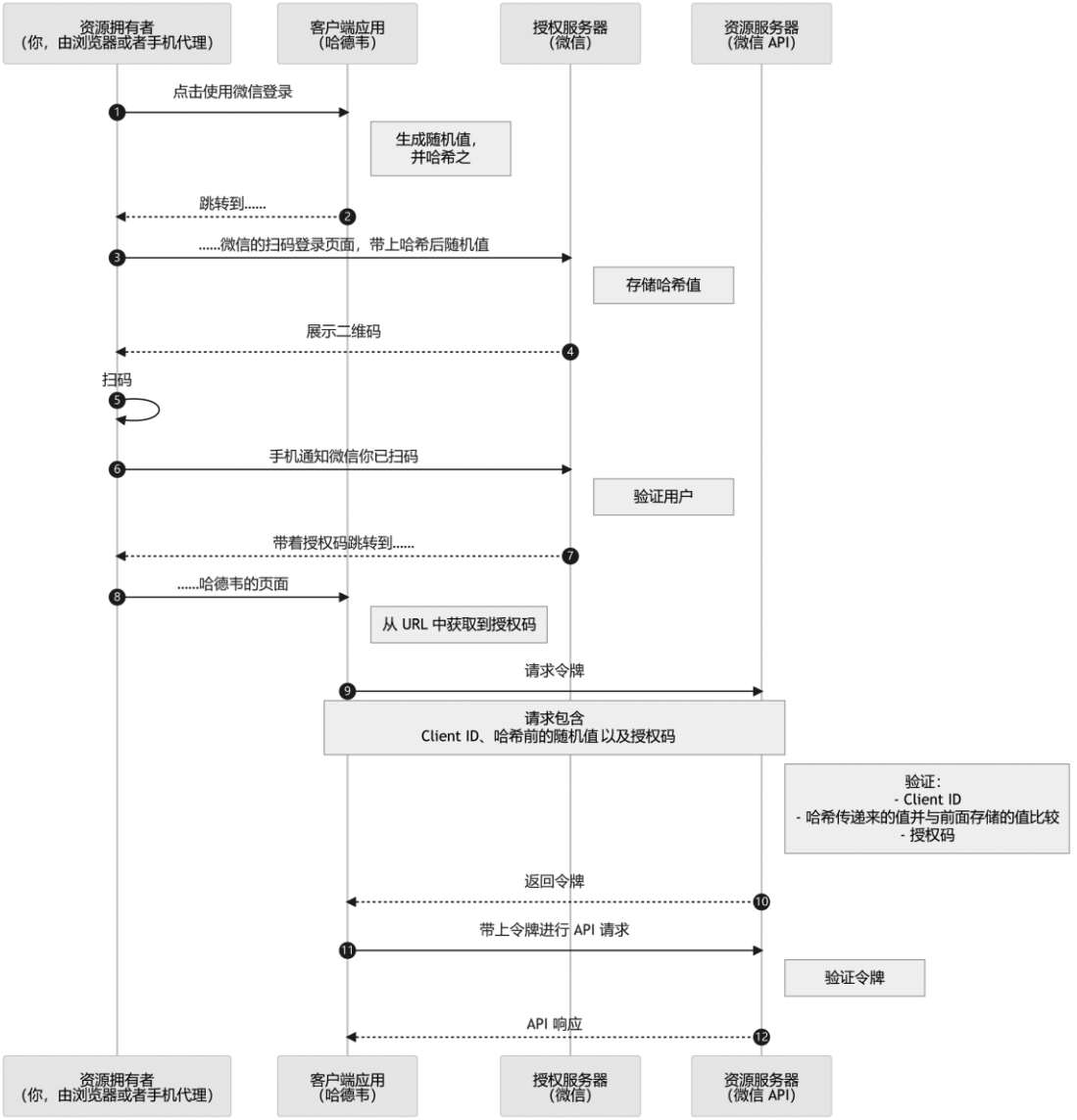

图 1-19　PKCE 流程时序图

3. 资源拥有者凭据许可/密码许可

密码许可（Password Grant，也称为密码授权）类型是一种直接使用用户凭据进行认证和授权的许可类型。用户名和密码有时也统称为用户凭据。用户在 OAuth 2.0 的流程中扮演了资源拥有者的角色，因此，密码许可类型有时也被称为资源拥有者凭据类型。总之，在密码许可类型中，用户向客户端应用提供其用户名和密码，客户端应用将这些凭据直接发送给身份提供者进行验证。一旦验证通过，身份提供者将直接向客户端应用返回访问令牌，以供后续的资源访问。

密码许可类型的优势在于简单直接，适用于用户对客户端应用的信任度较高的场景。然而，由于涉及用户凭据的传输和存储，密码许可类型的安全性需要特别关注，如采取适当的加密和保护措施。

4. 客户端凭据许可

以上几种许可类型都对应着面向人类用户的场景，但是 OAuth 2.0 也支持面向机器的场景。

在客户端凭证许可（Client Credentials Grant）中，客户端使用其自身的凭证（例如，客户端 ID 和密钥）直接向身份提供者请求访问令牌。这种许可类型适用于没有用户参与的场景，例如后台任务或服务到服务的通信。

这些是常见的许可类型，但实际应用中可能还存在其他类型的许可，我们可以根据具体的身份认证和授权需求来选择合适的许可类型。

注意，许可类型的选择应根据应用程序的特性和安全需求进行评估，以确保使用合适的授权机制和权限管理。

1.4.4 访问令牌和身份令牌的区别

在身份认证和授权过程中，访问令牌和身份令牌是两个关键的概念，它们具有不同的作用和特点。

- 访问令牌是一个用于访问受保护资源的令牌。它是身份认证后获取的，并且包含关于用户和授权信息的一组凭据。访问令牌通常用于向资源服务器发出请求，并验证用户对资源的访问权限。访问令牌通常具有较短的有效期限，以增加安全性。客户端在发送请求时需要将访问令牌包含在请求头或参数中。
- 身份令牌是在身份认证完成后返回给客户端的令牌。它包含有关用户身份和认证信息的声明。身份令牌通常用于在客户端应用程序中表示用户身份，以便进行个性化显示、用户信息展示或其他业务逻辑的处理。身份令牌通常具有较长的有效期限，以确保在用户会话期间持续有效。客户端可以在前端应用程序中存储身份令牌，以便在需要时使用。

访问令牌和身份令牌在功能和用途上有所区别。访问令牌主要用于授权和资源访问，而身份令牌主要用于表示用户身份和提供用户相关信息。

在一些身份认证和授权协议中，如 OAuth 2.0 和 OpenID Connect，访问令牌和身份令牌通常同时返回给客户端。客户端可以根据自己的需求和业务逻辑使用这两个令牌，以实现安全的资源访问和用户身份管理。

1.4.5　刷新令牌是什么

在前面介绍令牌与会话时，提到了令牌机制的一个缺陷，就是覆水难收。一旦令牌泄露，攻击者就可以使用该令牌访问受保护的资源。为了缓解这一安全隐患，在授权服务器颁发访问令牌时，需要将令牌的有效期设置得很短（比如 5 分钟）。但是，这样做会导致用户每隔 5 分钟就需要重新登录一次，如此一来，用户的体验是很差的。为了解决这个问题，OAuth 2.0 引入了刷新令牌（Refresh Token）的概念。

在身份认证和授权过程中，刷新令牌是一种用于获取新访问令牌的令牌。它通常用于在访问令牌过期后，向身份提供者（IdP）请求新的访问令牌。刷新令牌通常具有较长的有效期限，以确保在访问令牌过期后仍然有效。

一般来说，当 OIDC 客户端在获取用户信息时收到 401 错误，表明访问令牌已经过期。此时，可以使用刷新令牌来重新获取一个新的访问令牌，然后使用新的访问令牌获取用户信息，从而完成一次刷新令牌的过程。在这个过程中，终端用户通常无感知，感觉登录状态一直有效。

在 OAuth 2.0 规范中，如果要在令牌接口中获取刷新令牌，需要保证在构建授权链接时，scope 参数中包含 offline_access，这样才能在令牌接口中获取到刷新令牌。如果不传入这个参数，那么在令牌接口中就不会返回刷新令牌。

在使用刷新令牌换取新的访问令牌时，使用的是同一个令牌接口，但是传入的参数不同，需要传入 grant_type=refresh_token，而不是 grant_type=authorization_code 或者其他，同时还需要传入 refresh_token 参数，这个参数就是之前获取到的刷新令牌。除这几个参数外，还需要传入 client_id 和 client_secret，这两个参数是客户端的标识和密钥，用于在授权服务器校验客户端的身份，否则将得到一个不正确的客户端的响应。这个客户端凭据和参数不一样，是通过 HTTP Basic Authorization 头部传入的，格式详见前面关于 Basic 认证的讨论。

1.4.6　什么是单点登录

在自序中，笔者提到自己在一家外企将几个系统对接到公司的域账号后，既省去了用户的认知负担，又简化了管理员的工作。后来笔者才知道，让用户登录一个系统后，在另一个系统中可以重用之前登录的信息，省去了用不同的账号登录的过程，这就是单点登录（Single Sign On，SSO）。

单点登录是一种身份认证机制，它允许用户在多个应用系统中使用相同的凭据进行登录，从而实现一次登录，访问多个应用的目的。它的核心思想是让用户只需要提供一次身份验证，就可以在多个系统中无须重复输入凭据。

在传统的身份认证方式中，用户需要为每个应用系统单独进行身份验证，每次登录都需要输入用户名和密码。这不仅烦琐，还容易导致密码管理问题和用户体验不佳。而单点登录解决了这个问题，使用户只需登录一次，即可访问多个应用系统。

单点登录的工作原理通常涉及一个称为身份提供商的中心认证系统。用户向 IdP 提供凭据进行身份验证，一旦验证成功，IdP 会向用户颁发一个令牌（如访问令牌或身份令牌）。用户在访问其他应用系统时，只需提供这个令牌给应用系统，而无须再次进行身份验证。

通过单点登录，用户可以享受到以下好处。

- 便捷性：用户只需登录一次，即可访问多个应用系统，简化了登录过程，提高了用户体验。

- 安全性：由于用户只需提供一次凭据，减少了密码输入的次数，因此降低了密码泄露和重放攻击的风险。
- 统一管理：单点登录集中了用户身份验证和授权管理，方便对用户账户和权限进行集中管理。

单点登录已经成为现代应用开发中常用的身份认证方式之一，广泛应用于企业内部系统、云服务、社交媒体平台等各种应用场景。常见的单点登录标准协议包括 OIDC、OAuth 2.0、SAML2、CAS 3.0 和 LDAP。前面我们已经介绍了 OIDC 和 OAuth 2.0，接下来介绍 LDAP 和 SAML，并将 CAS 留到后面的 3.2 节进行介绍。

1. LDAP

轻量目录访问协议（Lightweight Directory Access Protocol，LDAP）是基于 X.500 的行业标准协议，用于管理账户和作为身份验证平台。但 LDAP 不会定义登录这些系统的方式，而是在身份验证和访问控制过程中扮演一部分角色。例如，在用户能够访问某些资源之前，LDAP 被用来查询用户及其所属的组织，以决定用户是否具备权限。

用户（有时是设备）数据在目录存储中以一种层次结构进行组织，并且有一系列的标准协议可以用来进行访问、存取和复制。LDAP 在互联网中的历史很悠久，新版本是 v3，在 RFC4511[1]中定义。不过 LDAP 的最初设计是在 1990 年出现的，主要用在电信领域和 x500 标准中。x500 是一个聚焦于描述 DAP（Directory Access Protocol）、DSP（Directory System Protocol）、DISP（Directory Information Shadowing Protocol）和 DOP（Directory Object Protocol）的组件集合。

LDAP 是 x500 DAP 中描述的一些概念的简要版本，尽管不是高精尖的科技，但是在很多面向内部或者外部的身份平台中扮演了非常重要的角色。为什么呢？因为它为存储和扩展用户数据提供了坚实的基础。比如微软的 Active Directory 以及 Azure Active Directory（后被改成 Entra ID），就是基于 LDAP 的。

大规模的用户存储有一些基本的属性要求。首先，它们需要能够存储大量的用户数据，而且需要能够快速地进行查询。其次，需要有能力将数据高速地复制到不同的办公室、站点或者地理位置以实现容错和高可用。

LDAP 对实体的模式描述基于它们在 DIT（Directory Information Tree，即目录信息树，将数据的层次结构放到树中的不同分支）中的位置。比如一个名叫 John Doe 的用户实体，可能会被描述为：

```
cn=John Doe,ou=People,dc=example,dc=com
```

在这个描述中，cn 是 common name 的缩写，ou 是 organizational unit 的缩写，dc 是 domain component 的缩写。这个描述中的每个部分都是一个 RDN（Relative Distinguished Name，即相对唯一名称），它们都是相对于 DIT 的根节点而言的。

基于 LDAP 的系统可以类比为作为用户数据存储的关系数据库。尽管很多数据库都可以存储这样的数据，但基于 LDAP 的服务仅仅聚焦于一点，那就是存储实体并提供高吞吐量和高扩展性的身份认证——无论是用户身份还是设备身份。

2. SAML

SAML（Security Assertion Markup Language，安全声明标记语言）是一种基于 XML 的开放数

[1] https://tools.ietf.org/html/rfc4511

据格式，允许计算机在网络间共享安全令牌。SAML 2.0 可以基于跨域网络实现单点登录，从而减少向单个用户发放多个身份令牌的开销。

它包含两个实体：

- 服务提供商。
- 身份提供者。

授权过程大致如下：

（1）服务提供商向身份提供者发送 SAML 请求，请求用户身份验证。

（2）身份提供者要求用户输入用户名和密码，并对其进行验证。

（3）如果验证成功，身份提供者将 SAML 响应返回给服务提供商，其中包含额外的信息以确保响应未被篡改。

其具体时序图如图 1-20 所示。

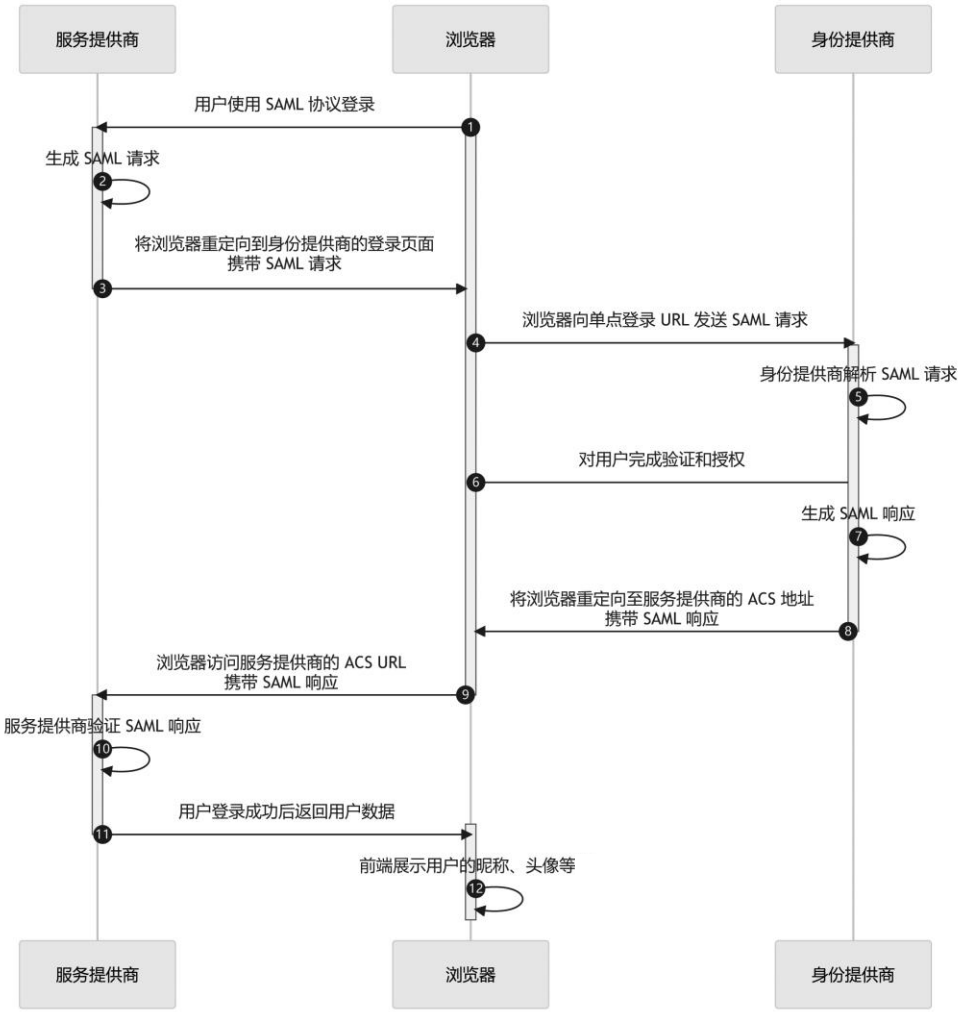

图 1-20　SAML 认证流程

图 1-20 中提及的 ACS 是 Access Control System（访问控制系统）的缩写。

3. 总结

单点登录有多个标准协议可选，如果想要实现单点登录，建议使用的协议顺序是：OIDC > SAML2 > CAS 3.0 > LDAP > OAuth 2.0。表 1-4 是一个更加详细的单点登录协议对比表格。

表 1-4 单点登录协议对比表格

协议名称	OIDC	SAML2	CAS 3.0	OAuth 2.0
验证	是	是	是	否
授权	是	是	否	是
易于使用的程度	😀 😀 😀 😀 😀	😀 😀	😀 😀 😀	😀 😀 😀 😀
安全性	都是安全的			
使用场景	多应用之间的单点登录 API 授权	企业内部单点登录	企业内部单点登录	多应用之间的单点登录 API 授权

可以看到，单点登录协议有多个，而 OIDC 协议是首选。因为 OIDC 在 OAuth 2.0 的基础上添加了身份验证能力，也可以说 OIDC 是在 CAS 协议的基础上添加了授权能力，但同时它比 SAML 协议更加简单。它的便利性、隐私保护和安全特性让它成为组织实现单点登录最好的选择。

1.5 小 结

本章介绍了身份认证的基本概念，包括身份认证、授权、许可类型、访问令牌、身份令牌、刷新令牌和单点登录等，强调了身份认证和授权的联系与区别。这些概念是理解身份认证和授权的基础，对于设计和实现身份认证系统至关重要。同时，按照目标对象进行了分类，还介绍了一些常见的身份认证协议和标准，包括 SAML、OAuth 2.0 和 OIDC 等。另外，对认证场景进行了细分，并重点介绍了 OAuth 2.0 的几种许可类型，包括授权码许可、隐式许可、资源拥有者凭据许可和客户端凭据许可。最后，介绍了刷新令牌和单点登录的概念，以及它们的作用和特点。

下一章将介绍身份认证所依赖的安全协议，并详细介绍一些在身份认证场合常用的加密和哈希算法。希望通过这些介绍消除读者在实际使用时的迷惑，让读者能够更加自信地使用这些算法。