关于权限控制，你需要知道的一切

基本概念

权限控制主要分为认证和鉴权两个部分。

认证（authentication）：

> Authentication is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity.

即：认证是证明用户是他声明的身份的过程

鉴权（authorization）：

> Authorization is the function of specifying access rights/privileges to resources, which is related to general information security and computer security, and to access control in particular.

即：鉴权是验证系统中账号是否有权访问某种特定资源的过程

这两个概念涉及大量的名词，下表整理了最基础的概念：

关键词	**中文**	**释义**
user	用户	屏幕前的你我他，一个个活生生的人
identity	身份	在身份提供商中存储的数据，描述一个全局唯一的用户的信息
account	账号	一份系统内部保存的数据，是该系统的**使用者**
identity provider	身份提供商	负责身份管理的组件，提供认证功能
authentication	认证	证明用户是他声明的身份的过程
authorization	鉴权	系统中验证使用者是否有权进行某种操作的过程
token	令牌	一种认证/授权的实体，通常是登录后系统为用户签发的，以免后续请求需要重复登录
credential	凭证	用户登录时输入的证明自己身份的东西，例如用户名、密码

上面的释义可能比较晦涩，举个现实中的例子，大家就能轻松理解这些概念。小明拿着身份证去坐高铁的过程，就涉及了完整的认证和鉴权。

> 假设小明拿着身份证去做高铁。

小明是用户（user），后端系统是高铁，小明要进行的操作是在某个时间搭乘去往某个站点的某趟列车。

> 小明在安检口将身份证放在闸机上，并通过人脸识别通过了闸机，进入到高铁站内。

小明的身份由身份证定义，身份证通过身份证号（identifier）保持身份的唯一性。国家的身份信息网站是 identity provider，向高铁提供了身份信息。

小明要过闸机，他首先持有身份证，声明我的身份是小明对应的身份，这张身份证可以被视为一个凭证。最简单的，系统可以只认小明的身份证，认为**持有这张身份证的人就是小明**。这是最简单的单因子认证。

那你可能会问了：如果小明身份证丢了，有不法分子拿着小明身份证怎么办呢？所以一般高铁还要做人脸识别。这就是**多因子认证（MFA：multi factor authentication）**。

> 那身份证和人脸的区别又在哪儿呢？

身份证国家为其颁发的身份凭证，并非小明自主、主动掌握的。小明的脸是一种生物密码，是小明天生就有，其主动权掌握在小明手中。

身份证和人脸识别可以类比到 token 和密码。token 是后端系统签发，是通过了认证了之后的产物。而密码是用户主动设置的，用作认证的凭证。它们的相同点在于：一旦它们泄露或丢失，伪造者即可伪装成该用户，拥有用户的所有操作权限。它们的区别在于：基础 token 是一种授权码，严格来说，使用不包含用户身份信息的 token 通过系统验证，是一种授权行为，而非认证行为。

这里先讲一个超标的概念：标准的 OAuth 就是一种授权机制，而 OIDC 由于在 token 中加入了不可伪造的用户信息，就成为了一种授权协议。

现在，小明证明了自己是“小明”，即用户通过了认证，验证了自己的身份。在系统看来，小明对应的“账号”就是已认证的状态了。

> 小明上车前将身份证放在闸机，通过后走上了月台，顺利的乘上了车。

抛开闸机的认证环节不谈，闸机主要做了鉴权。闸机记录在这个闸机刷卡的人是要搭乘 XXX 号列车。由于小明在 12306 买过票了，闸机知晓有权搭乘这趟列车，于是放行。

在这个过程中，小明在 12306 买票时，并不是由人直接和计算机系统打交道，而是通过“账号”和系统打交道。闸机已知代表小明的账号在高铁系统中的账号买过票了，现在有个认证过自己是“小明”的人来刷卡，闸机就打开了闸门，小明就搭乘上列车啦。

这里一些归纳出四个非常重要的点，大家一定要牢记在心：

1. 用户与身份是一一对应的

2. 每个账号都对应一个身份

3. 认证面向账号，需用户证明其身份与账号的关联身份一致

4. 鉴权面向账号，验证账号是否有权做什么操作；而认证确认了用户 - 身份 - 账号的正确关联，从而关联到了用户有权做什么操作

有用的基础知识

从 cookie/session 到 json web token

标准的 HTTP 请求是无状态的，在古早的时代，为了让某个域名下的所有网页能够共享某些数据，session 和 cookie出现了。客户端访问服务器的流程如下

- 首先，客户端会发送一个 http 请求到服务器端。
- 服务器端接受客户端请求后，建立一个 session，并发送一个 http 响应到客户端，这个响应头，其中就包含 Set-Cookie 头部。该头部包含了 `sessionId`。Set-Cookie 格式为 `Set-Cookie: value[; expires=date][; domain=domain][; path=path][; secure]`，具体请看 Cookie详解
- 在客户端发起的第二次请求，假如服务器给了 set-Cookie，浏览器会自动在请求头中添加 cookie
- 服务器接收请求，分解 cookie，验证信息，核对成功后返回 response 给客户端

Cookie 和 session 作为古早的技术方案，在流量小的单机情况下，除了占用一些客户端或者服务器的存储空间外，其实没有太大的问题。但在分布式场景下，session 的一致性问题比较难解决；访问量大时，server 端将耗费大量空间存储会话；此外，它们不能防止 CSRF 攻击。

在技术的变革中，json web token(jwt) 应运而生。它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。

Jwt 由三部分组成，它们之间用圆点(.)连接。这三部分分别是：

- Header
- Payload
- Signature

因此，一个典型的JWT看起来是这个样子的：

> xxxxx.yyyyy.zzzzz

接下来，具体看一下每一部分：

- Header header典型的由两部分组成：token的类型（“JWT”）和算法名称（比如：HMAC SHA256或者RSA等等）。然后，用Base64对这个JSON编码就得到JWT的第一部分。
- Payload JWT的第二部分是payload，它包含声明（要求）。声明是关于实体(通常是用户)和其他数据的声明。对payload进行Base64编码就得到JWT的第二部分。
- 为了得到签名部分，你必须有编码过的header、编码过的payload、一个秘钥，签名算法是header中指定的那个，然对它们签名即可。签名是用于验证消息在传递过程中有没有被更改，并且，对于使用私钥签名的token，它还可以验证JWT的发送方是否为它所称的发送方。

Jwt 的工作流程：在认证的时候，当用户用他们的凭证成功登录以后，一个JSON Web Token将会被返回。此后，token就是用户凭证了，你必须非常小心以防止出现安全问题。一般而言，你保存令牌的时候不应该超过你所需要它的时间。无论何时用户想要访问受保护的路由或者资源的时候，用户代理（通常是浏览器）都应该带上JWT。服务器上的受保护的路由将会检查Authorization header中的JWT是否有效，如果有效，则用户可以访问受保护的资源。

jwt 的优点有：

1. 无状态和可扩展性：存储在客户端，完全无状态，可扩展。我们的负载均衡器可以将用户传递到任意服务器，因为在任何地方都没有状态或会话信息。
2. 安全性：在签发 token 的秘钥不泄露时，没有人可以伪造 token。
3. 时效性：jwt 设计了过期机制，减少 token 意外泄露造成的风险。

user-agent 带来的问题：从 jwt 到 OAuth

回忆一下账号的意义：人无法和计算机后台系统直接打交道，所以需要**账号**作为直接操作后台系统的实体。那人是怎么操作账号的呢？人同样无法直接操作账号，它需要**用户代理**，代替人执行指令。这就是 user-agent 的概念。

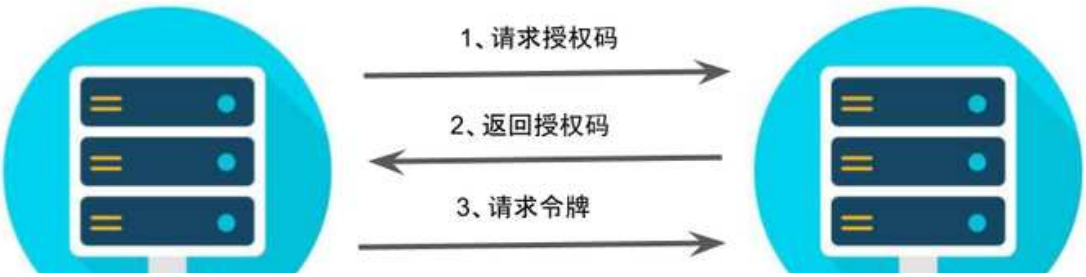
比如，当你在电脑上打开网页时，user-agent 就是 chrome；当你用 kubectl 操作 k8s 集群时，user-agent 就是 kubectl。

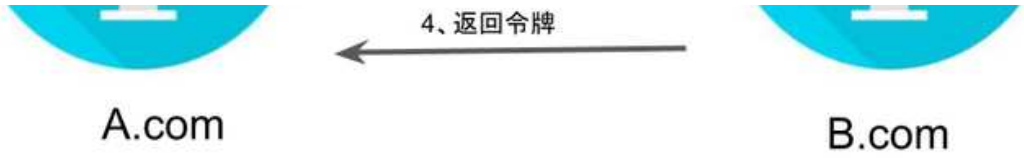
在简单的业务场景时，用户很少感知到 user-agent 的存在。那它存在的价值是什么呢？我们考虑这样一个涉及三方的权限控制问题：我住在一个大型的居民小区，小区有门禁系统，进入的时候需要输入密码。我经常网购和外卖，每天都有快递员来送货。我必须找到一个办法，让快递员通过门禁系统，进入小区。如果我把自己的密码，告诉快递员，他就拥有了与我同样的权限，这样好像不太合适。万一我想取消他进入小区的权力，也很麻烦，我自己的密码也得跟着改了，还得通知其他的快递员。有没有一种办法，让快递员能够自由进入小区，又不必知道小区居民的密码，而且他的唯一权限就是送货，其他需要密码的场合，他都没有权限？

这个问题的解法如下：门禁系统的密码输入器下面，增加一个按钮，叫做"获取授权"。快递员需要首先按这个按钮，去申请授权。他按下按钮以后，屋主（也就是我）的手机就会跳出对话框：有人正在要求授权。系统还会显示该快递员的姓名、工号和所属的快递公司。我确认请求属实，就点击按钮，告诉门禁系统，我同意给予他进入小区的授权。第三步，门禁系统得到我的确认以后，向快递员显示一个进入小区的令牌（access token）。令牌就是类似密码的一串数字，只在短期内（比如七天）有效。

上面一节说到了，jwt 解决了授权问题，可以使得用户不必重复认证，即可访问资源。基于 jwt 的这个优秀特性，我们在互联网场景里解决上面的问题，而这就是 OAuth 的设计精华。**OAuth 就是一种授权机制。数据的所有者告诉系统，同意授权第三方应用进入系统，获取这些数据。系统从而产生一个短期的进入令牌（token），用来代替密码，供第三方应用使用。**

OAuth 的工作流程大致如下：



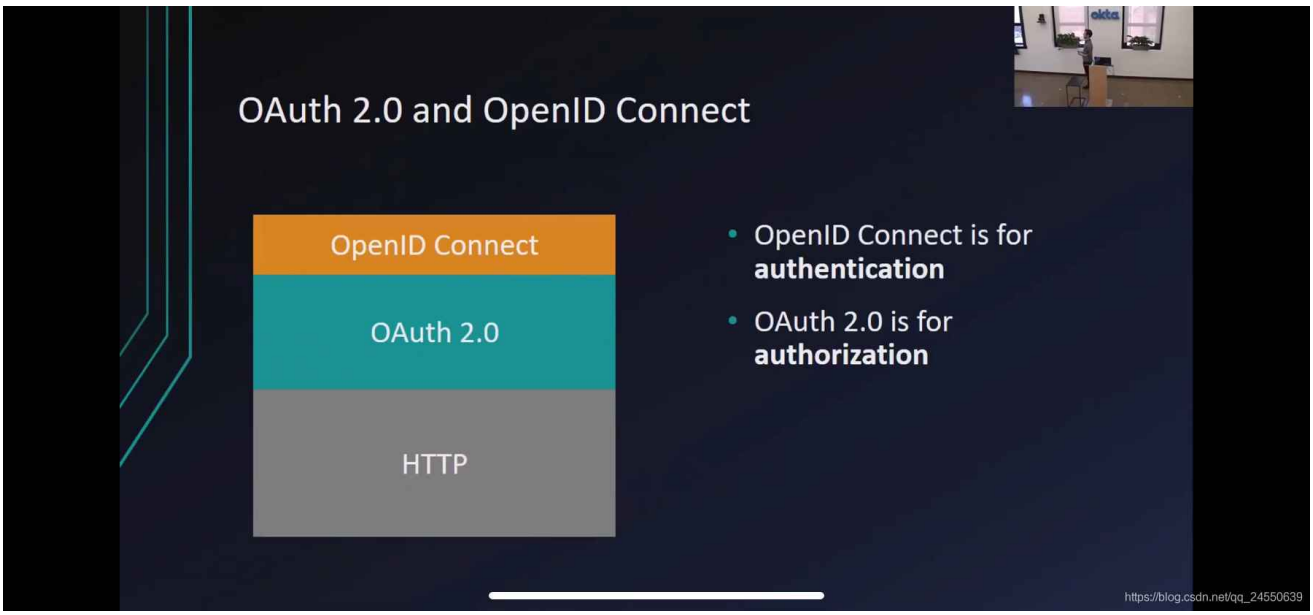


注意授权和认证的区别。一般来说，B.com 这个网站会集成认证的能力。比如，github OAuth，就会让你输入 github 账号的用户名密码，但这不代表 OAuth 拥有认证能力。OAuth 只拥有授权能力，而认证是大家额外拓展的标准外能力。

授权还是认证？从 OAuth 到 OIDC

OAuth 解决了授权的问题，但没有解决认证。OAuth 没有一个标准的方法来搜集 user 用户本身的资料，它只关心 user-agent，而非 user。假设在一次授权中，user-agent 拿到了 OAuth 签发的 token，服务器可以知道该 token 是有效的，但无法知道该 user-agent 代表了哪个用户。对流量的染色、统计，以及在此之上的诸多数据分析，都会受到一定影响。

于是，业界用 OIDC（Open ID Connect）基于 OAuth 实现了认证能力。



OIDC 相比 OAuth 最核心的改动为：

- 使用 ID token （附带user的信息）代替 access token
- 使用 userinfo endpoint，供获取详细的用户信息

ID token 也是一个 jwt，但用标准化的格式附着了用户的信息。一个例子见 ID Token 完整字段含义。

有个很有趣的点：我直接用 OAuth2，然后在 access token 中附着用户信息不行吗？答案是可以的，只是 OIDC 协议把它标准化了。

那 OIDC 的真正价值会体现在哪里？和授权问题类似，我们考虑这样一个问题：我们基于后端的系统 A、B、C 等，在上层封装了管理平台 Z。现在，我们要实现用户请求从管理平台向后端系统的打通，在此过程中，我们需要身份信息在不同的系统中传递。

如果我们建设了 OIDC 系统，我们可以这么玩：

- 用户在管理平台登录时，从 OIDC 系统获取到一个 ID token
- 管理平台 Z 访问后端各个系统时，均携带该 ID token
- 后端各个系统均验证 token 的合法性，并从中读取身份信息
- 后端各个系统要主动访问管理平台 Z，或访问其他系统时，也从 OIDC 获取 ID token 来访问

这样一来，我们的各个子系统的账号体系就打通了，通过标准化的 ID token 实现了不同系统间身份信息的传递，解决了整个大系统的身份认证问题。

如果我们不建设 OIDC 系统，当然也是可以玩的，后面会更详细的聊一下到底会有哪些玩法。

Impersonation 的价值

虽然 OAuth 和 OIDC 已经成为了业界标准，但从头开发一个完整的 OIDC 系统，是一件极其耗费人力物力的事情，需要一步步实现协议族的每个细节。当然，现在基于 golang 的开源库也有不少，比如

- https://github.com/coreos/go-oidc
- https://github.com/caos/oidc

但我仍然要指出的是：这些开源实现要么**功能不完整**，要么**社区不繁荣**。建设完整的 OIDC 仍然是一件 ROI 比较低的工作项。

那么上一节提到的大系统中的身份认证问题，没有 OIDC 该怎么玩呢？一个可行的方案是身份伪装（impersonation）。

Impersonation 通常发生在系统 A 访问系统 B 时，在绕过认证的同时传递用户信息。为此，系统 A 需要与系统 B 事先协商一个 admin 账号，系统 A 访问系统 B 时，携带该 admin 账号的认证凭证。同时，系统 A 在请求中携带原始用户信息。系统 B 通过 admin 账号的认证凭证进行认证，在鉴权时面向原始用户信息所代表的的账号。

例如在 k8s 中，就有 impersonation 的机制：

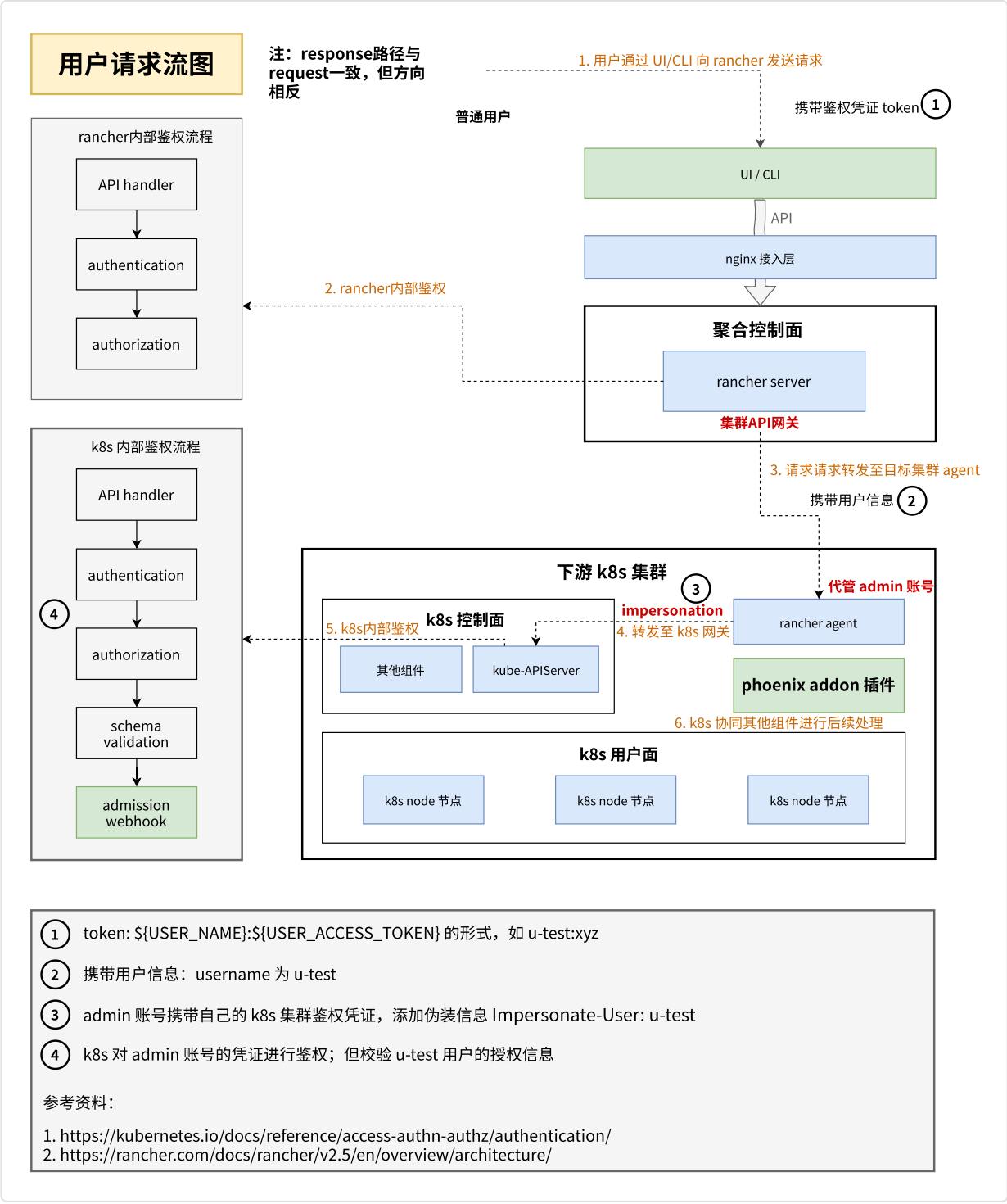
> A user can act as another user through impersonation headers. These let requests manually override the user info a request authenticates as.

通常原始用户信息放在 `Impersonate-User` 这一标准 header 中。

下面讲一个 impersonation 的实例案例。rancher 的架构中，下游的集群会部署一个 cluster agent。当请求从 rancher 的管理面到达 k8s 控制面的过程，需要由 cluster agent 进行转发。rancher 的认证和 k8s 的认证是两套机制，那么，请求是如何通过 k8s 的认证的呢？答案就是 impersonation。

rancher 将请求转发到 agent 时，携带 `Impersonate-User` header，声明原始用户的信息；agent 由于是部署在下游集群中的组件，通过配置 k8s 的 service account 与 RBAC，使用了超级管理员账号，agent 将 token 附着到请求中，转发到 k8s APIServer。这样，请求就通过了 k8s 的认证。

具体的流程如下图所示：



对认证的总结

认证：认证是证明用户是他声明的身份的过程。

为了解决无状态的 http 协议的状态保存问题，在业界方案从 cookie/session 演进 jwt 的过程中，我们看到业界越来越注重分布式系统下的认证。从 jwt 衍生而出的 OAuth、OIDC 等协议，进一步解决了授权问题。由于 OIDC 的复杂性，在简单的业务场景中，使用 impersonation 是一种性价比更高的方案。

ACL：DAC 和 MAC 的区别

为了控制系统中不同账号的权限，业界的鉴权方案也在不断演进中。比较古老的鉴权模型是 access control list（ACL），它通过一张表记录不同账号对不同资源的访问权限，且是针对每个账号的。

基于 ACL 有两种拓展方式：

- Discretionary Access Control (DAC)：规定资源可以被哪些主体进行哪些操作。同时，主体可以将资源、操作的权限，授予其他主体。DAC 的优势是强调灵活性。
- Mandatory Access Control (MAC)：a. 规定资源可以被哪些类别的主体进行哪些操作 b. 规定主体可以对哪些等级的资源进行哪些操作 当一个操作，同时满足a与b时，允许操作。MAC 的优势就是实现资源与主体的双重验证，确保资源的交叉隔离，提高安全性。

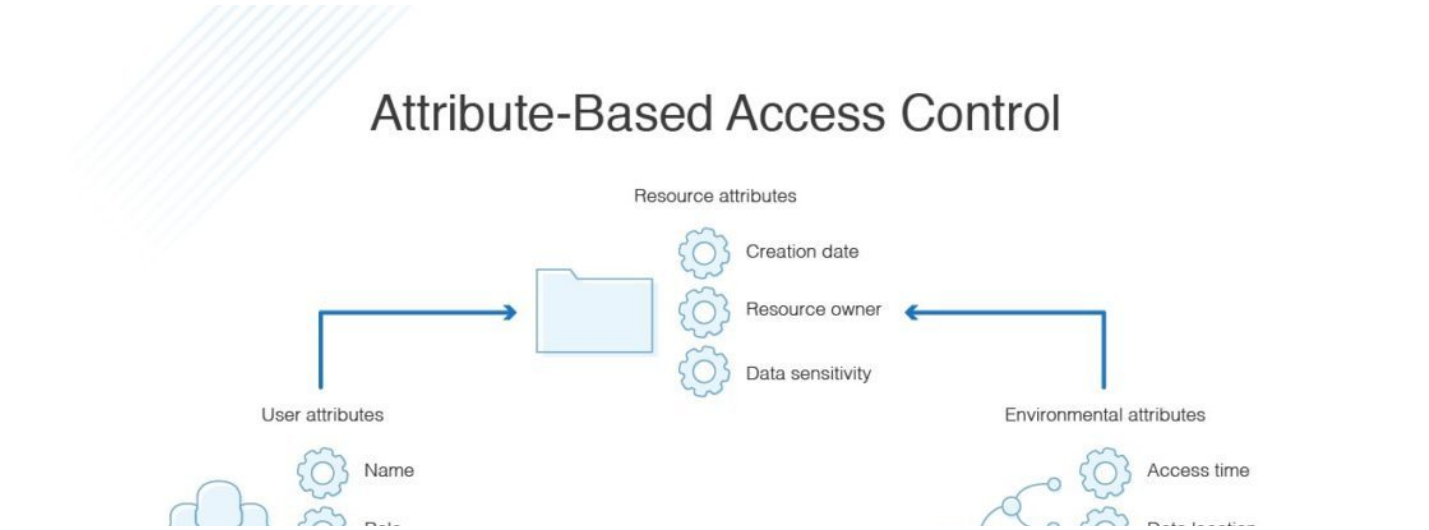
现代鉴权系统：RBAC/ABAC

更为流行的鉴权方式是 RBAC，即 Role-Based Access Control。或者是 ABAC，即 Attribute-Based Access Control。

RBAC 通过定义角色与用户的绑定关系，来定义鉴权规则。比如定义角色 reader，拥有查看资源的权限，只要定一个 role-binding，将角色 reader 授予用户 A，用户 A 就拥有了查看资源的权限。RBAC 通常支持对用户组的授权。比如将角色 reader 授予用户组 G，那么用户组 G 下的所有的用户都拥有查看资源的权限了。

RBAC 适合场景简单、同种用户权限趋于一致的场景，维护复杂度低、易于实现。但缺点是假设需要让某个用户有特定的权限，就需要额外增加一个角色。故在权限频繁变化的场景下，RBAC 并不是很好的选择。

ABAC 的模型更加复杂，它通过对**属性**的控制，定义更细致的鉴权规则。用户与资源均可以有属性，例如只允许在特定的时间或与相关员工相关的某些分支机构进行访问，而不是让人力资源部门的人员总是能够访问员工和工资信息。



ABAC 适合更精细、更复杂的场景，但它的实现复杂度更高、难度更大，维护的成本也更高。

未来趋势：context-aware 鉴权

考虑这样一个问题：学校的门禁系统希望控制普通学生晚上 10 点之后不能离开宿舍。类似这样的场景不仅仅涉及资源与用户，还涉及**鉴权上下文**。

鉴权上下文指鉴权发生时的上下文，比如时间、地点等。随着越来越多的公司关注企业安全，简单的基于资源与用户的模型，将难以满足日益复杂的鉴权需求。未来，context-aware authorization 会变的更加流行。

控制面鉴权与数据面鉴权

不管是 ABAC 还是 RBAC，都属于控制面鉴权的范畴，本质上是对用户对于控制面资源的权限控制。系统在控制面定义了资源类型后，通过一个资源**与真实的数据建立映射关系**，这种资源就是控制面资源。举个例子，用户的一个训练任务，在 linux 上就是一些进程；在 slurm 的控制面上是一个 training job 的记录，这条记录与实际的进程（数据）有某种映射关系，从而使得我们可以通过操作这条记录，操作系统上的对应进程。

有时候控制面鉴权并不能解决所有的问题。考虑这样的场景：我们有一些训练任务，训练任务可能跑的是 parrots 任务，也可能跑 pytorch 任务，我们希望某些用户只能看 parrots 任务（当然这样的场景比较扯淡，只是举个例子）。这种场景下，我们需要关心**数据面鉴权**。

数据面鉴权是每个子系统需要自行处理的，与具体的数据格式、业务逻辑息息相关。这里不作额外展开。

各个系统的权限控制体系介绍

k8s 的权限管理体系

认证

读者可以参考官方的用户认证部分。整个文档可以概括为这么几个关键点：

1. k8s 不做用户管理，只管理 service account
2. k8s 支持用公私钥、token 进行认证，也支持 webhook authentication
3. k8s 对 OIDC 的支持相当有限
4. k8s 支持身份伪装（impersonation）

一、service account 的管理

我们可以用 kubectl 查看 k8s 中的 service account，比如

Groovy

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2022-01-28T03:10:56Z"
  name: default
  namespace: default
  resourceVersion: "348"
  uid: a9c4602e-fbf2-49f3-89d5-f4b8220770d0
secrets:
- name: default-token-wcjmt
```

其中的 secret 字段包含了该 service account 的 token，解码之后，是一个标准的 jwt，通过在线的解码，可看到其 payload 为

JSON

```
{
    "iss": "kubernetes/serviceaccount",
    "kubernetes.io/serviceaccount/namespace": "default",
    "kubernetes.io/serviceaccount/secret.name": "default-token-wcjmt",
    "kubernetes.io/serviceaccount/service-account.name": "default",
    "kubernetes.io/serviceaccount/service-account.uid": "a9c4602e-fbf2-49f3-89d5-f4b8220770d0",
    "sub": "system:serviceaccount:default:default"
}
```

二、webhook authentication

k8s 向来以可拓展性闻名，支持 webhook authentication。

三、对 OIDC 的支持

k8s 对 OIDC 采用信任而非集成的方式，见 openid-connect-tokens。用户需要首先从 OIDC 系统中拿到 ID token，k8s 会校验 token 的合法性，决定是否通过鉴权。

为什么说这种支持方式非常有限呢？因为 k8s 完全不处理后续的鉴权。它依赖上层的管理平台集成 OIDC，并打通鉴权。比如管理平台定义的鉴权体系，由一个 agent 翻译成 k8s 的 role & role-binding。

四、impersonation

k8s impersonation 是一个相当好的设计，为其他系统的接入提供了便利。

鉴权

读者可参考鉴权概述。关键知识点：

1. 通过 PSP 和 RBAC 提供基础的鉴权能力
2. ABAC 的支持相当有限

一、RBAC 的细节

k8s 的 RBAC 用 namespace 进行了最小的租户定义，角色分为 role/clusterRole，分别是 namespaced 和 cluster-scoped，对应的有 roleBinding/clusterRoleBinding。一个 role 可定义如下的 spec

YAML

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: configmap-updater
rules:
- apiGroups: [""]
  # 在 HTTP 层面，用来访问 ConfigMap 的资源的名称为 "configmaps"
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

YAML

```yaml
rules:
  - nonResourceURLs: ["/healthz", "/healthz/*"] # nonResourceURL 中的 '*' 是一个全局通配符
    verbs: ["get", "post"]
```

其中要注意的是 `nonResourceURLs`，它是一个简化版的**数据面鉴权**。但能力相当有限。

role binding 可以给单个用户授权，也可以给用户组授权。

YAML

```yaml
subjects:
- kind: User
  name: "alice@example.com"
  apiGroup: rbac.authorization.k8s.io
```

YAML

```yaml
subjects:
- kind: Group
  name: "frontend-admins"
  apiGroup: rbac.authorization.k8s.io
```

该值是一个用户持有有效的 token 或者 kubeconfig 发起请求并通过认证后，由 k8s 解析出的用户信息。k8s 定义了一个 user + group 的身份识别规范，它没有管理用户账号，但对身份的字段进行了规范，在鉴权中将身份当做一个账号使用。

二、ABAC 的支持

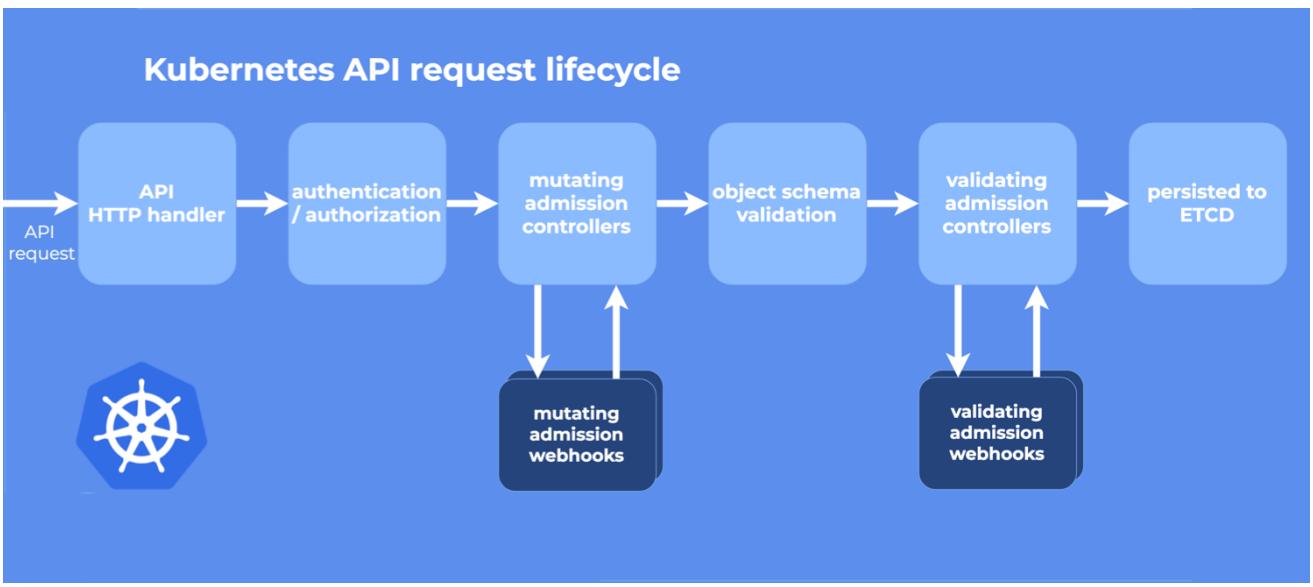
参考使用 ABAC 鉴权，支持的 spec 和 RBAC 一样。ABAC 由于其复杂性，k8s 对其支持的优先级一直不高，k8s 更倾向于使用更简单直接的 RBAC，如果需要复杂的策略，需要开发人员在 k8s 之上构建自己的鉴权体系。

dynamic admission

k8s 支持更灵活、动态的准入机制，参考 Webhook 模式。分为 mutate webhook 与 validate webhook。我们在 phx-k8s 上实现的额外的权限控制系统 admission-webhook 就是基于 k8s 的这个机制。

总结

一个 API 请求在 k8s 系统中均要通过认证与鉴权，在写入 etcd 库之前，其生命周期为



即：优先通过认证与鉴权，之后走 mutate 插件，再走 validate 插件。

slurm 的权限管理体系

账号管理

slurm 的账号管理复用 linux 的账号管理系统。通常用 NIS 在不同节点间进行账号的同步。

> NIS, (Network Information Services), enables account logins and other services (host name resolution, xinetd network services configuration, ...), to be centralized to a single NIS server.

认证

使用 munge 进行用户认证。

> munge是认证服务，用于生成和验证证书。应用于大规模的HPC集群中。它允许进程在【具有公用的用户和组的】主机组中，对另外一个【本地或者远程的】进程的UID和GID进行身份验证。这些主机构成由共享密钥定义的安全领域。在此领域中的客户端能够在不使用root权限，不保留端口，或其他特定平台下进行创建凭据和验证

鉴权

控制面鉴权：slurm 通过 sacctmgr CLI 配置用户对分区的使用权限，使用 mysql 存储 ACL 信息。

数据面鉴权：slurm 限制了普通用户只能删除自己创建的任务，这一限制在 slurm 中 hard code 了。

auth plugin

slurm 支持 auth plugin 实现认证和鉴权策略的拓展。并支持用动态链接的方式加载插件。开发人员不需要用 C 语言在 slurm 仓库中修改代码，用其他语言编译成一个动态链接库即可。

rancher 的权限管理体系

rancher 为什么要做用户管理

首先明确一下，任何直接服务于用户的系统都必须有账号。账号才是系统的直接使用者。所以 rancher 作为一个企业级 k8s 管理平台，必然会建设自己的账号体系。如果没有账号的话，用户压根就没法登陆，总不能像 k8s 一样，管理员手动去创建 x509 证书吧……

rancher 的认证方式

通过官方文档可以发现，rancher 可以可对接许多公有的 IDentity Provider，比如 google OAuth、github、Azure AD 等，也可以对接私有的 OpenLDAP 服务器。在使用过程中，管理员可以在**安全-认证**中配置 IDP，之后登陆时，用户就可以选择除了 local authentication 之外的认证方式。

不管使用哪种认证方式，rancher 在 IDP 进行了认证行为之后，会在本地保存这个用户的账号，并把 IDP 中的身份信息和本地账号进行一次 **mapping**。在 k8s 集群上部署 rancher 之后，k8s 集群上会出现若干 rancher 的 CRD，其中一个叫做 `users.management.cattle.io`，rancher 就是这样使用 CRD 进行用户管理。如果配置了认证方式，比如配置了 openldap 认证方式，那么当一个用户通过 openldap 方式登录到系统中时，管理员可以看到一个新建的 user CR。

那这个 mapping 是怎么做的呢？user 中记录了一个 `principleIDs` 的字段，分别标志了不同的认证域内的用户主体信息。比如

YAML

```yaml
principalIds": [
  "local://u-xxxxxxx",
  "openldap_user://CN=xxx,OU=xxx,DC=xxx"
],
```

即这个用户的本地账号的 ID 是 `u-xxxxxx`，在 openldap 服务器上的 distinguish name 是 `CN=xxx,OU=xxx,DC=xxx`（DN 是 openldap 协议中唯一标志一个用户的属性）。这样就完成了身份与账号的映射。

要注意，除了 local authentication 外，rancher 最多只能同时对接一种外部认证方式。这个也非常好理解，正如之前强调过多次，身份是唯一的，账号不是。如果对接了多个 openldap 服务器，那么假设用户在 IDP1 中的 DN 和在 IDP2 中的 DN 可能是不同的，那么在 rancher 看来，这就是**两个用户**。故只能对接一个 IDP。

如果要有同时对接多个 IDP 的需求，那么必须在多个 IDP 之间就做好用户的同步、合并、账号映射，但是这样一来，又只需要对接一个合并后的 IDP 就可以了。比如 Azure AD 可以进行其他域的 AD 的托管，将信息同步到 Azure AD 服务中，保障系统只对接一个 IDP。

rancher 发起的请求在 k8s 上的认证

先说结论：rancher 巧妙地利用**代管账号+身份伪装**（impersonation）进行。

rancher 会在每个纳管的 k8s 集群上部署一个 cluster-agent，该 agent 使用的 service account 被授予了 `cluster-admin` 的权限。在一开始将隧道的时候我们已经讲解过了，从 rancher 发往 k8s 的请求都通过这个 agent 进行，且转发给 agent 的时候，携带了用户在 rancher 上的账号信息，也就是 user CR 的 metadata name。

agent在 k8s 上认证时，实际使用的是 service account 的身份完成的认证，却伪装自己是 rancher 普通用户。之后在鉴权环节，k8s 看到的**身份**就不再是 service account 的身份，而是 rancher 账号的身份了。

rancher 鉴权模式

rancher 大致上沿用了 k8s 的 RBAC 机制，但拓展出了更丰富的角色层级。rancher 将角色分为三个层级

- global role：全局角色，适用于所有的集群和项目，往往可以对管理员授予这样的权限
- cluster role：集群级别的权限，该角色授权给用户时，必须指定一个集群
- project role：项目级别的权限，该角色授权给用户时，必须指定一个项目

注：rancher 用项目（project）表示一组用户的工作空间的隔离。一个 project 必须在一个集群内，包含若干个命名空间，而一个命名空间最多属于一个项目。project 是对 namespace 语义的简单拓

展。

rancher 的鉴权模式是服务于这样的用户隔离模型的。将角色拆解的更细，更易用。比如：

- 全局管理员不需要被单独授予每个集群的权限
- 管理员不需要为每个用户新建的 namespace 给用户手动授权

rancher 鉴权体系与 k8s 的对接

但注意，rancher 只是定义了 role 和 roleBinding，但并不在 rancher 做真正的**鉴权**。真正的鉴权发生在 k8s 侧。

rancher 通过 k8s operator 的方式将 role template + role template binding 映射为 k8s role + role-binding，保证了用户授权信息的最终一致性。

在真正转发请求时，rancher 会做一些必要的 validation，通过之后，再把请求转到 agent 侧。上面认证中已经说过，通过 impersonation，agent 伪装自己是普通用户，之后 k8s 的 RBAC 机制完成了真正的鉴权。

one-api 的权限管理体系

账号管理

one-api 不直接管理身份，只管理账号。它依赖外部的 IDP 或 IAM 服务。以目前 Y 计划中的架构为例，one-api 提供 user 的 create 接口，中台的账号中心负责将 ldap 中的用户信息同步过来，one-api 接收到创建的请求，在本地落库。账号用 k8s CRD 进行存储，其通用属性为

Go

```go
    IdentityProvider IdentityProvider `json:"identityProvider" protobuf:"bytes,1,name=identityProvider"`

    // type of user, either service account or user
    UserType UserType `json:"type" protobuf:"bytes,2,name=type"`

    // login name of user, unique within the realm of an identity provider
    LoginName string `json:"loginName" protobuf:"bytes,3,name=loginName"`

    // records principle of user, such as openldap DN, unique user id locally
    // +optional
    PrincipalIDs []string `json:"principleIds,omitempty" protobuf:"bytes,7,opt,name=principleIds"`
```

分别代表了

- 账号的身份来源，由于我们允许本地账号登录平台，idp 分为 local 与 external 两种
- 账号类型，分为 user account 和 service account
- 登录名称，用户的登录名称

- principle：an Internet-style login name for a user based on the Internet standard RFC 822.

认证

我们支持多种认证方式，比如 basic auth, token auth 等，只要通过任何一种认证，都认为该请求通过认证。所有的认证通过之后，我们会向请求方发放一个 bearer token，作为授权凭证。后续请求方携带该凭证访问 one-api 即可。

鉴权

one-api 采用 role 定义角色，采用 vp role binding, cluster role binding, global role binding 定义授权信息，这个设计是参考 rancher 的。这里可以展开讲一下鉴权系统的方案演进。

第一版设计中，我们用一个统一的 role binding CRD 实现，里面填写

YAML

```yaml
spec:
  role: admin
  user: user1
  kind: cluster # means cluster role binding
  scope: cluster1 # namespace in cluster role binding
```

但逐渐我们发现这样的方式非常难用，首先造成大量的数据堆积在 role binding 中，导致一次 LIST 的开销变大，且授权信息糅杂在一起难以区分。于是，我们将 role binding 分割为不同的种类，这是板上钉钉的事情。

既然 role binding 分割了，role 是否也要分割呢？我们在资源建模后确实发现资源也是分 scope 的，我们确实想过把 role 也分成 global role, cluster role，每种 role 只包含自己 scope 的资源，但发现这样的设计是难以落地的。

在实际业务场景中，可能有这样的需求：每个用户申请时可以看到所有的 cluster 列表和 vp 列表。这意味着一个 global member 需要有 cluster 和 vp 的权限，并不具备拆分的可能性。

Y 计划里我们是怎么把各个系统的账号打通的

通过身份完成账号映射

Y 计划有三个系统：中台、存储、调度。在账号层面由中台负责与各个其他系统拉通。中台做的事情：

1. 与 IDP 对接，将 IDP 的不同用户在中台系统中分别初始化为不同的账号
2. 调度与中台预先沟通好 service account，作为互相访问的基础
3. 调度、存储提供用户创建的接口，中台负责将用户信息同步到其他系统中
4. 提供登录基础设施，采用 OIDC 的协议族实现。用户从登陆服务中获取 ID token，持有 token 访问调度系统。调度系统验签进行认证，之后根据 token 中身份信息进行账号映射，后续的鉴权由调度自行管理

还有哪些不足

一、身份信息的同步不到位

调度可能需要用户的其他身份信息，比如 DN、uid number、组织架构……同步账号时目前只同步了 login name，并没有其他的身份信息。因为没有类似 SCIM 一样的标准协议定义格式，身份信息的传递一直没有提上日程去落地。

二、不支持非 user account 的登录

中台的定位在于**用户管理**。所谓的用户管理即：依赖 IDP，同步用户信息，至于 service account 与 local user account 不考虑。各个系统在做自举时，需要一个 local user account 作为管理员。管理员的认证不走中台，前端需建设额外的登录界面。

三、不管理授权

可能造成权限的混乱。比如用户 A 在中台是普通用户，在调度是管理员，在存储是组员。该用户的请求在三系统中流转时，可能遇到权限问题。

四、没有对用户注册的 service account 的管理能力

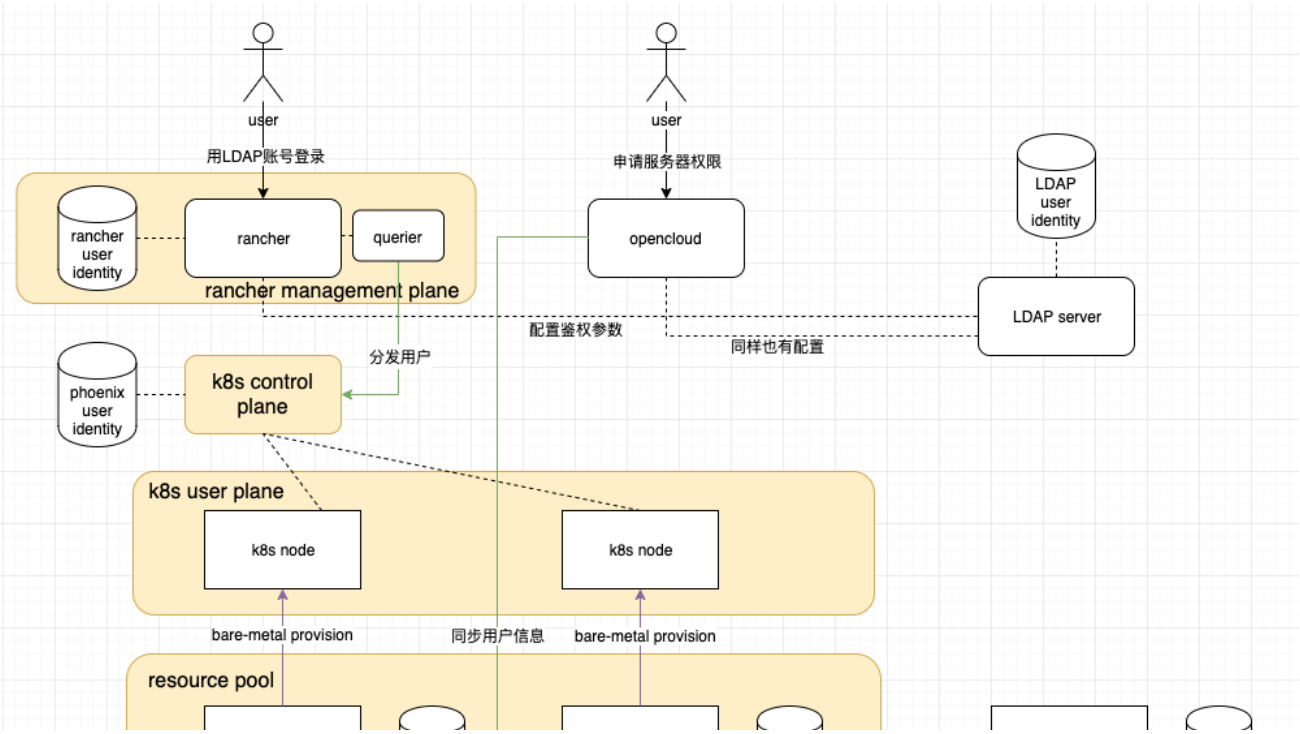
用户可能除了自己使用平台之外，还可能生成一些 service account，并给 service account 进行授权。而我们是不支持这样的进阶用法的。

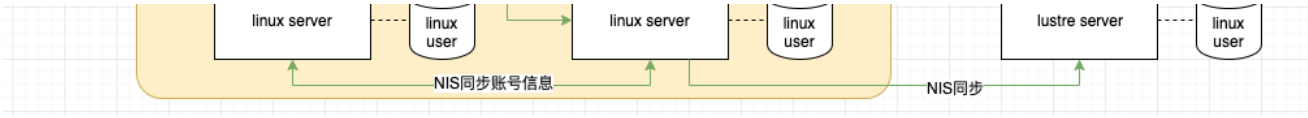
带你看懂 IAM

IAM 即身份接入管理，它管理系统中的身份，并对身份进行有效的授权。IAM 是企业 IT 建设中极其重要、极其复杂的部分。下面通过我们 phoenix-k8s 的例子，带你理解 IAM。

phoenix-k8s 场景下的账号模型

phoenix-k8s 是一个典型的多 service provider 的系统，其不同子系统（rancher、k8s、linux）上的账号的关系图如下：





用户账号的身份均来源与 ldap 服务器。通过 OC 上的工单系统，账号在物理机上被创建，并通过 NIS 在集群中得以同步。rancher 与 k8s 则完全与 OC 无关，rancher 对接 ldap，在用户登录 rancher 后完成账号的创建，再由 phoenix 的自研组件将身份信息在 k8s 上同步。

我们遇到这么几个问题或糟糕体验：

1. 用户必须登录 rancher，才能在 rancher 上有账号
2. 用户离职之后，开发机往往得不到删除，管理员也不知道用户已离职
3. 用户有调度的权限不等于拥有存储的权限
4. 用户没有统一的 SSO，登录各个系统要重复输入用户名
5. 用户的权限需要在各个系统上分别开通

所以我在去年提出过这样的问题，但是由于研发压力与 IAM 的复杂度，只是梳理了现状，没有提出解决办法。

重要性与复杂度

Identity and Access Management 是信息安全计划的重要组成部分。他们确保只有经过授权和身份验证的用户才能访问您的资源。例如，你需要定义一些身份主体（在您的账户中进行操作的用户、组、服务和角色），根据这些主体创建策略，并实施严格的凭证管理。这些权限管理元素构成了身份验证和授权的核心概念。

如果执行得当，IAM 有助于确保业务生产力和数字系统的顺畅运作。 员工可以随处无缝工作，而集中式管理则确保他们只访问自己的工作所需的特定资源。 此外，向客户、承包商和供应商开放系统可以提高效率并降低成本。

举个例子，IAM 的复杂性与挑战主要体现在这么几种场景：

一、日益分散的员工队伍

由于员工分散在全国各地甚至全世界，企业 IT 团队面临着更为严峻的挑战：在不牺牲安全性的前提下，保持员工连接企业资源的一致体验。

二、分布式应用

随着基于云和软件即服务 (SaaS) 应用的增长，用户现在可以随时随地使用任何设备登录关键业务应用，如 Salesforce、Office 365、Concur 等。然而，随着分布式应用的增加，管理这些应用的用户身份的复杂性也在上升。用户可能在密码管理上费心费力，而 IT 部门则面临着支持成本因用户不满而不断上升。

三、高效处理访问权限

如果没有一个集中的 IAM 系统，IT 人员就必须手动授予访问权限。用户在获取关键业务应用的访问权限上耗时越长，工作效率就越低。反过来说，如果不能撤销已经离开组织或调入不同部门的员工的访问权限，会造成严重的安全后果。为了关闭此类漏洞和风险的窗口，IT 人员必须尽快撤销此类员工对企业数据的访问权限。然而问题在于，在许多组织中，这意味着 IT 人员必须通过每个用户的帐户来了解他们可以访问哪些资源，然后手动撤销该访问权限。手动授予和撤销访问权限是重体力劳动，而且容易出现人为错误或疏忽。

四、自携设备

员工、承包商、合作伙伴和其他人员出于工作和个人原因带来个人设备并连接到企业网络。BYOD 的挑战不在于是否将外部设备接入企业网络，而在于 IT 部门是否能够迅速做出反应，以保护组织的业务资产，同时不影响员工的生产力，并提供选择的自由。

身份管理通用协议：SCIM

设想这样的场景：有一天一些员工离职了，他可能使用着大大小小几十个 app，不仅如此，各个 app 中还有它们申请的 service account。离职员工的账号需要一一注销，这个事情谁来做？

为了解决身份管理问题，业界提出了标准化的 SCIM 协议：

> **System for Cross-domain Identity Management (SCIM)** is a standard for automating the exchange of user identity information between identity domains, or IT systems.

SCIM 帮助在各个系统中管理身份信息。

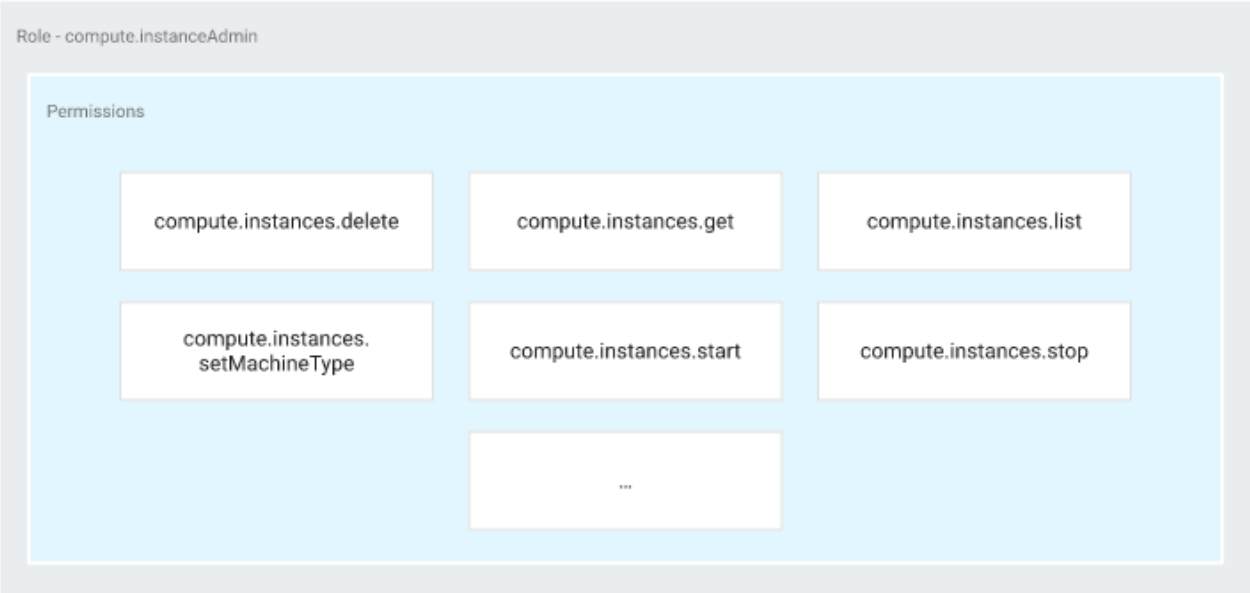
- 核心模式提供了用于表示用户和组的抽象模式和扩展模型。 该模式的标准序列化采用JSON格式提供。
- SCIM协议规范定义了 REST API，用于通过JSON交换身份资源。

各个 app 需要有一个 SCIM client，与 IAM 通过 SCIM 协议进行身份同步。该同步可不是我们在 Y 计划中做的那么简单，除了 principle 之外，SCIM 还会同步 attribute；SCIM 还对 service provider，也就是后端各个系统有一定的管理能力。完整的协议在这里。

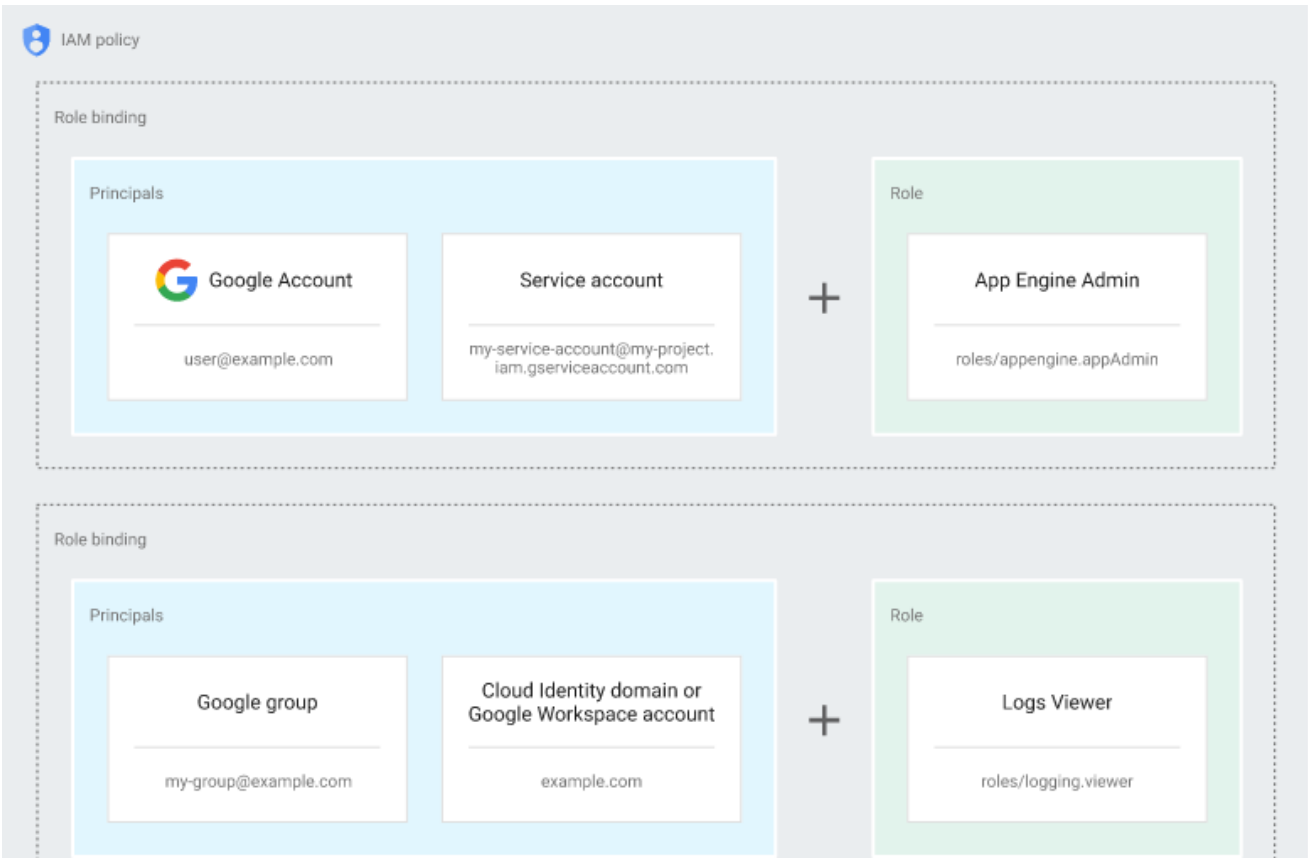
授权

IAM 中的授权与各个系统中的授权不同。之前说过，各个系统中的鉴权是针对账号的，因为账号是服务的直接使用者。但IAM 不直接对**账号**进行授权，由于账号分布在后端各个系统中，IAM 必须对**身份**进行授权。

在角色上，也不是 RBAC 那么简单的对**资源**的规定。在 IAM 中，一个后端服务叫做一个 service provider，角色通常也是细分到 service provider 中的。比如：



以 google cloud 的 IAM 为例，其授权模型如下：一组账号通过一个 principle 进行关联，其中有一个**主账号**，也就是用户账号，以及一堆附属 service account。一次授权针对一个身份（一组账号）进行。此外，还可以对用户组进行授权、继承用户组安全策略等……



总结一下

之前在基础知识里将 token、OAuth、OIDC，看上去似乎通过这些东西就能构建一个**拉通账号**的服务了。但其实在 IAM 的场景中，这些都只是开胃小菜。IAM 总结下来有四种能力：

- 认证
- 鉴权
- 身份生命周期管理
- 身份信息管理

再进一步：Identity as a Service

什么是 IDaaS

IDaaS 的全称是Identify as a Service，字面意思即**身份即服务**，是一种通过利用云基础设施，构架在云上的身份服务。

那 IDaaS 和 IAM 是啥关系呢？IDaaS = IAM + SaaS

业界案例

OKTA: The World’ s #1 Identity Platform

Figure 1: Magic Quadrant for Access Management

Source: Gartner (November 2021)

Okta是一家toB的SaaS公司，**其产品方向是在企业和应用之间打造安全、可靠、便捷的入口和用户身份管理平台。**

Okta 的产品有：

Products

Securely connect the right people to the right technologies at the right time

Single Sign-On
Secure cloud single sign-on that IT, security, and users will love

Multi-factor Authentication
Secure, intelligent access to delight your workforce and customers

Lifecycle Management
Manage provisioning like a pro with easy-to-implement automation

Universal Directory
One directory for all your users, groups, and devices

Authentication
Create secure, seamless customer experiences with strong user auth

Access Gateway
Extend modern identity to on-prem apps and protect your hybrid cloud

Advanced Server Access

包括：

- 单点登录：提供对所有Web应用程序的无缝访问
- Universal Directory：Multiple identity sources. One view + Centralize user management
- Advanced Server Access: Centralized access at scale
- API Access Management: extend authentication & authorization policies to APIs
- 多因子认证
- 认证：We want one login, one password, and one digital identity [for our customers] that unlocks the door to many of the services that the Digital division provides（包括联邦认证）
- 用户管理：中心化的用户管理系统
- 生命周期管理：管理身份声明周期，自动添加、接触授权
- B2B 集成：Easily connect customer, vendor, and partner identities

Okta 的复杂度非常高，是行业霸主之一。同时也揭示了 IDaaS 的复杂度。

注：UCP 的建设中，IAM 系统也是采购的 Okta 的方案。

IDaaS non-goals: it's a service, not a replacement

有一个非常重要的理念：有了身份云，并不代表 app 不需要认证和鉴权。

考虑这样一种情况：one-api 接入了一个身份云，one-api 希望有一个需求：普通用户只能删除自己的任务。

身份云进行的是控制面的权限控制。它可以感知到任务这一资源，并设置任务资源的访问控制策略。但它无法感知到具体的任务数据，这些信息只有在 one-api 的控制面上才有。故 one-api 依然需要有鉴权。在之前有说过，各个系统中，鉴权面向账号，故 one-api 也需要有账号（除非它像 k8s 一样不做 user account 管理，那样会导致账号与权限的分割）。

故一种最理想（但最复杂）的形态是：

- 后端各系统可有自己的账号，可建设自己的认证与鉴权体系
- 身份云向多个系统进行身份的同步
- 身份云提供 SSO 能力
- 身份云提供接入控制能力，并通过某种机制向后端系统同步

总结与建议

总结：

1. 认证和鉴权的基础概念需要理清
2. 从 session/cookie 到 jwt 到 OAuth 到 OIDC，响应了分布式系统、移动互联网演进的必然需求
3. 面向越来越复杂的安全与身份管理场景，IAM 的重要性越加凸显
4. 身份云就是云化的 IAM

建议：

1. 每个系统可以把能力委托给网关或身份云，但都应当具备账号管理、认证、鉴权的潜在能力
2. IAM 在复杂场景下凸显其作用，但其复杂度也超乎想象，对 IAM 的建设应当谨慎