电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

专业学位硕士学位论文

MASTER THESIS FOR PROFESSIONAL DEGREE

论文题目 **Kubernetes 强多租户模型的设计与实现**

专业学位类别	电子信息
学　　号	**202022081208**
作者姓名	韩杰明
指导教师	向艳萍　教授
学　　院	计算机科学与工程学院

分类号 TP319 密级 公开

UDC 注1 004.77

学 位 论 文

Kubernetes 强多租户模型的设计与实现

（题名和副题名）

韩杰明

（作者姓名）

指导教师 向艳萍 教授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 专业学位类别 电子信息

专业学位领域 计算机技术

提交论文日期 2023 年 3 月 28 日 论文答辩日期 2023 年 5 月 22 日

学位授予单位和日期 电子科技大学 2023 年 6 月

答辩委员会主席

评阅人

注 1：注明《国际十进分类法 UDC》的类号。

Design and Implementation of Kubernetes' Strong Multi-tenancy Model

A Master Thesis Submitted to
University of Electronic Science and Technology of China

Discipline **Electronic Information**

Student ID **202022081208**

Author **Jieming Han**

Supervisor **Prof.Yanping Xiang**

School **School of Computer Science and Engineering**

摘 要

最近几年云计算技术发展迅速，出现了许多革命性技术，其中以 Docker 为代表的的容器技术正在取代传统的 Hypervisor 虚拟化技术，逐渐成为了云计算行业的首选，而容器编排技术 Kubernetes 因其具有开源性、多功能性、可拓展性、活跃的社区生态环境以及大量云计算产商支持等多方面的优势成为了容器编排领域的事实标准。但是，Kubernetes 的多租户能力仍然存在一些不足，在租户资源配额管理、访问控制、数据隔离、租户管理以及租户网络隔离等多方面都需要进一步地完善。如今，随着云计算技术地发展以及对多租户共享资源池地深入理解，如何保证多租户环境下云平台的运行安全和数据安全，是云计算发展过程中必须要跨越的障碍。

为了解决上述问题，本文在 Kubernetes 现有多租户能力基础之上，从访问控制、计算资源隔离以及多租户网络三个维度进行设计，提出了一种强多租户模型解决方案。在多租户访问控制模块设计了租户 API，明确了强多租户模型中的租户定义，然后通过引入 Keystone 身份服务组件实现了用户认证与管理，接着设计了 RBAC 控制器实现了用户自动授权与鉴权功能。在计算资源隔离模块提出了可信容器和非可信容器使用不同容器运行时的方案，通过引入 Kata 安全容器引擎，保证容器内应用的内核隔离，解决了内核共享带来的问题。在多租户网络管理模块使用 Contiv-VPP 网络插件构建租户网络，实现了多租户网络隔离，并且基于 Kubernetes 插件化思想设计了 Contiv Plugin、Network-management 以及 Network-cli 三个插件，实现了多租户网络管理以及 Pod 网络自动配置的功能。

最后对本文的 Kubernetes 强多租户模型解决方案进行了功能测试和性能测试，测试结果表明本文设计方案能够满足 Kubernetes 强多租户模型的设计要求，符合预期结果，能够提供真正的强多租户能力，满足大多数生产环境的要求。

关键词：Kubernetes，强多租户模型，资源隔离，访问控制

ABSTRACT

In recent years, cloud computing technology has developed rapidly and many revolutionary technologies have emerged. Container technology, represented by Docker, is replacing traditional Hypervisor virtualization technology and gradually becoming the preferred technology in the cloud computing industry. Kubernetes which is a container orchestration technology has become the de facto standard in the container orchestration field due to its open source nature, versatility, active community ecosystem, scalability, and strong support from many manufacturers. However, Kubernetes' multi-tenancy capabilities still have some shortcomings, and further improvements are needed in many aspects such as tenant resource quota management, access control, data isolation, tenant management, and tenant network isolation. Today, with the development of cloud computing technology and a deeper understanding of multi-tenant shared resource pools, ensuring the security of cloud platform operation and data security in a multi-tenant environment is a barrier that must be overcome in the development of cloud computing.

In order to solve the above problems, this thesis focuses on access control, compute resource isolation, and multi-tenant network and proposes a strong multi-tenancy model solution based on the existing multi-tenancy capabilities of Kubernetes. In the multi-tenant access control module, the thesis designs the tenant API which defines tenants in the strong multi-tenancy model. User authentication and management are implemented by introducing the Keystone identity service component. An RBAC controller is also designed to provide automatic authorization and authentication functions for users. In the compute resource isolation module, a solution is proposed for trusted and untrusted containers to use different container runtimes. By introducing the Kata secure container engine, kernel isolation of applications inside containers is ensured, and the problem caused by kernel sharing is solved. In the multi-tenant network management module, the Contiv-VPP network plugin is used to build a tenant network, achieving multi-tenant network isolation. Based on the Kubernetes plug-in thinking, the Contiv Plugin, Network-management, and Network-cli plugins are designed to achieve multi-tenant network management and automatic configuration of Pod networks.

Finally, functional and performance tests are conducted on the Kubernetes strong multi-tenancy model solution proposed in this thesis. The test results show that the

solution proposed in this thesis can meet the design goals of the Kubernetes strong multi-tenancy model, achieve the expected results, provide real strong multi-tenancy capabilities, and meet the requirements of most production environments.

Keywords: Kubernetes, Strong multi-tenancy model, Resource isolation, Access control

目 录

第一章 绪论

1.1 研究背景与意义

云计算是最近几年发展最迅速的互联网技术之一，其主要特点是以互联网为基础，提供可弹性伸缩的计算、存储、网络等服务，用户无需关心基础设施地维护和管理，只需按需使用云服务即可满足其业务需求。随着信息技术的不断进步，云计算已经逐步成为企业信息化转型和升级的必经之路。

云计算的本质就是将巨大的共享资源拆分成无数个小资源，然后通过网络对这些资源进行按需分配，只需少量的操作就能快速配置和发布应用。如图 1-1 所示，云计算主要分为私有云、公有云、混合云、以及多云[1]，主要通过三种模式来提供服务：

1）基础设施即服务（IaaS）：利用 Hypervisor 等虚拟化技术将计算机资源抽象成资源共享池，用户通过云平台来获取需要的资源。

2）平台即服务(PaaS)：提供了应用设计、开发、部署、运维全流程解决方案，用户只需关心核心业务逻辑的实现。

3）软件即服务(SaaS)：云计算厂商在云上部署各种软件服务，例如数据库服务，用户可以根据实际需求选购需要的软件。

图 1-1 云计算服务类型以及特点

最近几年，我国的云计算行业发展迅猛，市场规模增长率高达百分之三十[2]，呈现出高速增长趋势，市场规模已经由最初的十几亿增长至几千亿。云计算行业巨

大的市场规模让云厂商在云计算技术上不断创新，云计算技术得到了快速发展，同样的，云计算领域不断涌现出的革命性技术也让云计算行业有了更好的发展前景。云计算技术将彻底改变传统的工作模式和商业模式[3]，在现代社会中将发挥出无限的商业价值，未来必将涌现出更多云计算领域的革命性技术。

过去，互联网公司的IT基础设施都是以虚拟机为中心，而如今，许多企业已经将IT基础设施迁移到了容器云平台[4]，Kubernetes因其出色的容器编排能力以及完善的开源社区生态环境逐渐超越其他云平台技术[5]，成为了行业首选。Kubernetes提供了十分强大的容器编排功能，并且在成本控制、资源高效利用等方面也表现突出。但是，随着云计算行业的快速发展以及对资源池概念地深入理解[6]，怎样保证多租户环境下云平台的安全，是云计算发展过程中必须要跨越的障碍。

1.2 国内外研究现状

1.2.1 容器云多租户模型研究现状

容器云多租户模型是指在一个容器云平台上多个租户共享集群资源[6]，同时又能够保证租户之间的隔离与安全性[7]。目前，国内外都有许多关于容器云多租户模型的研究，也出现了许多研究成果。

国内研究者从不同角度出发，提出了不同的容器云多租户模型。例如，基于OpenShift的多租户模型[8]，可以为不同租户分配资源，隔离应用程序和网络；基于Docker Swarm的多租户模型[9]，通过创建不同的Swarm集群为不同租户提供隔离和安全性；基于Kubernetes的多租户模型，通过命名空间和RBAC等机制，实现了租户间的隔离和资源管理。

国外研究者也对容器云多租户模型做了大量的研究。例如，Google提出了基于Kubernetes的开源多租户容器云平台——GKE[10]，它通过基于Google Cloud Identity的认证和授权机制，实现了租户隔离和权限管理。Rancher Labs提供了Rancher容器管理平台，通过为每个租户创建一个独立的Kubernetes集群实现租户隔离。

总的来说，容器云多租户模型在企业中的应用前景广阔，但也需要解决多方面的问题。随着容器技术的不断发展和成熟，相信这些问题会逐渐得到解决，容器云多租户模型也会得到更广泛的应用。

1.2.2 Kubernetes 多租户能力现状分析

在多租户模型中，租户会共享Kubernetes集群中的资源，为了保证租户之间的资源隔离，Kubernetes容器云平台需要提供相应的隔离功能。Kubernetes现有的

Namespace 隔离技术无法有效应对复杂生产环境中多租户的隔离要求，在资源隔离以及多租户访问控制等方面都存在一些不足。本小节将从以下两个方面对 Kubernetes 多租户能力进行分析：

（1）访问控制

访问控制对于任何云平台来说都是需要重点解决的问题，Kubernetes 原生的用户认证、授权、准入控制等功能在一定程度上能够满足要求，但是在强多租户模型中还有值得完善的地方。首先，Kubernetes 并不具有租户概念，它的 Namespace 命名空间并不能提供实际的多租户能力。因此在设计 Kubernetes 强多租户模型时首先就要确定租户概念，这是实现访问控制、资源隔离的前提。此外，在管理集群外部账户时 Kubernetes 的支持能力也略显不足。Kubernetes 对集群外的 API 请求有着严格的访问控制，但是在集群内部，租户与租户之间可以任意进行资源访问，Kubernetes 对于这类访问请求没有提供隔离限制，所以在强多租户模型中还需要解决租户之间 API 访问的问题。

（2）资源隔离

租户之间的资源隔离主要涉及两个方面：计算资源和网络资源。Kubernetes 默认使用 Docker 容器引擎，容器在运行时会共享内核，容器内的应用可能会逃逸到内核从而影响到其它容器[11]，所以考虑使用安全容器方案来实现计算资源的隔离。另外，Kubernetes 默认采用 Flannel 网络插件，能够保证集群中容器互通，实现了一个扁平全通的网络[12]，任意 Pod 之间都可以进行通信，但是在强多租户模型中还需要保证租户间的网络隔离。同时，Kubernetes 原生的 Network Policy 通过标签选择器模拟分段网络，由于网络如何划分还是需要人为配置，所以为了减少配置的难度需要借助网络插件实现网络的自动配置与管理。

1.2.3 安全容器方案研究现状

Kubernetes 默认使用的是 Docker 容器，Docker 作为传统容器是基于 Namespace 和 cgroups 来实现隔离的[13,14]，虽然实现简单便捷，但是存在安全隐患。Docker 容器只提供了多个应用相互隔离的运行环境，这些应用仍然是在同一个节点上面运行的，它们共享宿主机的内核，一旦某个应用逃逸到内核，那么其他应用也会受其影响。因此，在多租户环境下出现了许多安全容器方案。

为了解决共享内核带来的问题，一些学者提出了为容器提供完整操作系统级别的执行环境的方案[15]，容器运行应用程序时会独立运行，与容器所在宿主机隔离，为每个容器提供虚拟的独立运行环境。这种方案在一定程度上降低了内核共享开销，但是没有从根本上解决内核逃逸问题。另外，Google 开源的 gVisor 容器沙

箱技术通过阻断容器内应用对物理机内核的依赖[16]，隔离了容器应用与内核之间的访问，只提供对 Linux 内核的系统调用，将容器应用的系统调用转换为对 gVisor 的访问。此外，由 runV 发展而来的 Kata container 利用 Hypervisor 技术创建安全的容器运行时，为每个容器提供单独的虚机，通过硬件虚拟化技术实现了强隔离[17]，在兼容性上也比前面一种方案更好，同时还提供了强大的拓展能力。这种方案由于借助了传统的虚拟化技术，而虚拟化技术本身会消耗计算机资源，所以启动会比传统容器慢。

1.3 本文主要内容

通过前面的介绍可以发现 Kubernetes 目前在强多租户支持上还存在缺陷，Kubernetes 在设计之初并没有将多租户作为主要的关注点，其设计目标主要是实现容器编排和自动化部署。虽然 Kubernetes 具有命名空间的概念，但这并不是真正意义上的多租户解决方案，而是一种轻量级的逻辑分区，这也是它在多租户能力上存在不足的原因之一。为了让 Kubernetes 在复杂的强多租户环境下能够满足租户资源隔离以及访问控制的要求，本文将提出一种 Kubernetes 强多租户解决方案，本文主要包括以下内容：

（1）分析 Kubernetes 在强多租户能力的现状和不足，提出复杂多租户环境下强多租户模型的实现思路，从访问控制、资源隔离以及多租户网络等方面进行系统设计。

（2）为了明确租户的定义本文对 Kubernetes 原生 API 进行拓展，设计了租户 API。租户 API 清晰地定义了租户的概念，这是 Kubernetes 实现强多租户模型的基础，也是实现高效管理租户的关键。

（3）分析并比较目前行业内流行的容器运行时，在 Kata 安全容器基础之上设计多租户计算资源隔离策略，保证底层容器运行时的隔离，解决了内核逃逸的问题；同时在 Kubernetes 现有网络架构基础之上实现了的多租户网络方案，保证了租户网络的隔离，提高了数据平面的隔离性。

（4）完善了 Kubernetes 访问控制功能，设计了租户 API 并根据实际场景需要将用户分为了系统管理员、租户管理员以及普通用户，另外还设计了一个代理插件来实现 Keystone 与 Kubernetes 的无缝对接。实现了 Kubernetes 对外部用户的间接管理，而且还在原来的用户认证能力之上提供了更丰富的认证方式。

（5）使用 Contiv-VPP 构建容器网络环境，通过设计 Contiv Plugin、Network-management 以及 Network-cli 三个模块实现 Contiv-VPP 对 Kubernetes 容器网络的

接管，利用 VPP 引擎来优化容器间网络流量的处理速度，并通过自定义网络策略实现流量控制、负载均衡等功能。

（6）部署 Kubernetes 集群，按照文中的方案实现强多租户模型并对它的功能和性能分别进行测试，通过测试结果分析本文提出的解决方案是否符合预期。

1.4 本文结构

本文共分为六大章节，每章的内容如下：

第一章是绪论。首先介绍了云计算相关背景与意义，接着对容器云平台强多租户模型以及安全容器方案的研究现状进行概述，并对 Kubernetes 强多租户支持能力存在的不足进行了分析，最后提出了本文的主要内容和论文结构。

第二章是 Kubernetes 及相关技术概述。首先根据实现方式的不同介绍了容器技术的两大分类，然后分析了行业目前应用最广泛的 Docker 容器技术，介绍了 Docker 的组织架构以及核心组件，然后介绍了 Kubernetes 容器编排技术，分析了 Kubernetes 的组织架构、核心组件、核心资源对象、访问控制以及网络模型等。

第三章是系统需求分析与总体设计。首先分析了 Kubernetes 强多租户模型的设计需求，接着介绍了本文设计方案的总体架构以及使用的技术路线，然后分别介绍了访问控制、计算资源隔离以及多租户网络的设计。在访问控制模块主要介绍了租户 API 的设计、租户网络模型的设计以及访问控制代理组件的设计。在计算资源隔离模块主要介绍了 Kata 安全容器的原理以及如何通过 Kata 实现租户的计算资源隔离。在多租户网络模块设计了 Contiv Plugin、Network-management 以及 Network-cli 模块，实现 Contiv-VPP 和 Kubernetes 的无缝集成。

第四章是 Kubernetes 强多租户模型的实现。主要对第三章中的设计方案进行具体的实现，在访问控制模块中通过 k8s-keystone-auth 代理组件接入 Keystone 认证服务，实现用户认证、用户授权与鉴权等功能。在计算资源隔离模块对可信容器和非可信容器使用不同的容器运行时，引入 Kata 安全容器，实现 Pod 在独立虚机上运行。在多租户网络模块通过 gRPC 实现了 Contiv Plugin、 Network-mangement 以及 Network-cli 三个模块，并且利用 Contiv-VPP 实现了多租户网络隔离、负载均衡以及流量控制等功能。

第五章是系统测试。对本文为实现 Kubernetes 强多租户模型而设计的各个模块的功能和性能都进行测试，并分析测试结果是否和设计预期一致。

第六章是总结与展望。首先对本文所做的工作与成果进行总结，然后对本文方案存在的不足做出说明，并为方案的后续改进提出一些思路。

第二章 Kubernetes 及相关技术概述

2.1 容器技术

2.1.1 容器技术分类

容器技术核心就是对计算机资源进行隔离和限制，把进程运行在一个独立的沙盒环境中，并且能够把这个沙盒打包成镜像移植到其它机器上运行。和传统虚拟机相比，容器技术占用的服务器资源更少，应用启动时间也更短，弹性伸缩能力更强，项目部署、运维的成本更低。应用程序使用传统方式进行部署时，部署效率低，系统迁移能力差，后期维护困难，容器技术的出现解决了这些问题[18]。在微服务应用盛行的今天，一个服务的背后可能需要几十上百个服务的支持，服务间存在复杂的调用关系，开发人员不得不消耗大量时间才能正常部署服务。容器技术可以将应用和它依赖的环境打包成镜像，在安装好容器运行时的服务器中就能一键拉取镜像并启动容器，减少了服务部署的难度。容器技术根据其使用的虚拟化技术可以划分为两类：

（1）基于操作系统虚拟化的容器技术

内核是操作系统中最重要的组成部分之一，通过对操作系统的内核进行虚拟化，让软件实例在虚拟化的内核单元中调度执行[19]，从用户视角来看，自己的应用好像就是在独立的专用服务器上运行，每个用户都觉得自己是在独占服务器资源。操作系统虚拟化本质是对内核的共享使用，它的性能开销小，但是隔离能力不足。

（2）基于 Hypervisor 虚拟化的容器技术

Hypervisor 虚拟化就是传统的服务器虚拟化，对计算机物理资源进行抽象并向用户提供模拟的计算机环境[20]。通过 Hypervisor 技术可以在一台物理主机上创建出多个虚拟机。基于操作系统虚拟化的容器技术对内核进行虚拟化，容器中的应用会共享内核，可能发生内核逃逸，而基于 Hypervisor 虚拟化的容器技术可以解决内核逃逸的问题，它通过 Hypervisor 技术实现虚拟化，而镜像文件的格式则沿用了 Docker 镜像文件格式。基于 Hypervisor 虚拟化的容器技术发展至今以及出现了许多优秀的解决方案，比如：Intel 公司研发的 Clear 容器和 OpenStack 基金会托管的 Kata 容器[21]。它们底层都采用了 Hypervisor 技术，对 Linux 内核进行裁剪并生成 Docker 镜像格式的镜像文件，在 Kubernetes 中运行时会为每个 Pod 创建独立的虚拟机来运行多个容器。

2.1.2 Docker 容器引擎

Docker 是将 Linux 已有的 cgroups 以及 namespace 等关键技术进行整合而形成的一个革命性成果[22,23]。Docker 通过 namespace 实现容器间的隔离，容器只能访问所在命名空间下的资源，对于其他资源它是无法感知的，一个容器的操作不会对其他容器产生影响[24,25]，Docker 通过 namespace 技术让每个容器都像是在单独的虚拟机中运行[26]，同时，Docker 还利用 cgroups 技术来限制容器对资源的使用[27]，避免单个容器占用太多资源。

Docker 采用了客户端服务器架构，其中 Docker 客户端与 Docker Daemon 使用一组 RESTful API 实现交互，Docker 客户端使用命令行或者调用 API 向 Docker Daemon 发送指令，Docker 处理完成后又由 Docker Daemon 将结果响应给客户端。Docker Daemon 负责维护和管理容器，包括容器的创建、运行与删除等。Docker 核心组件及架构如图 2-1 所示。

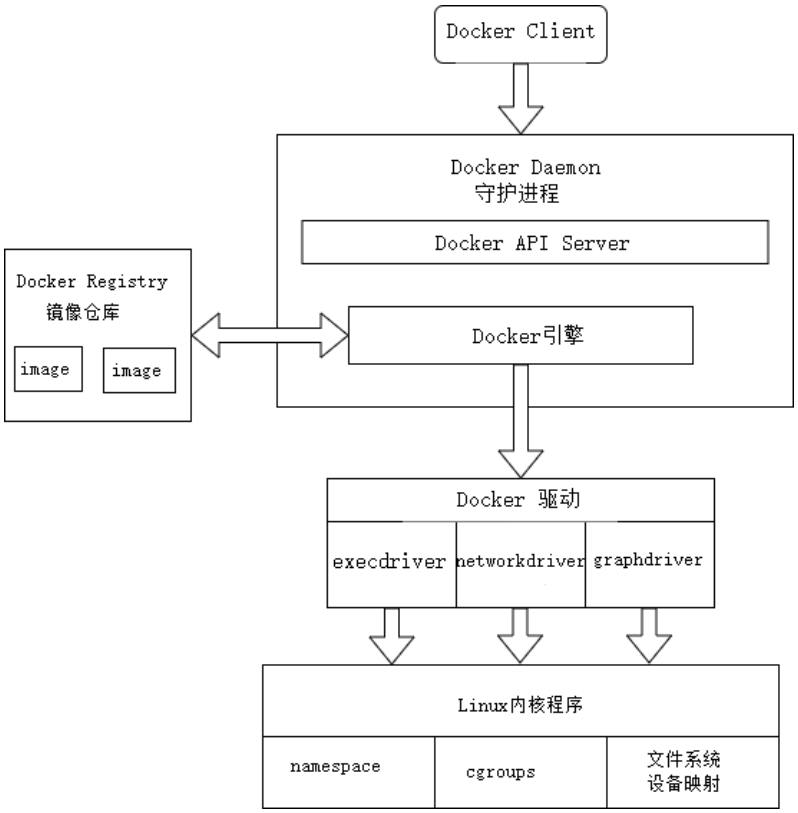

图 2-1 Docker 架构图

（1）Docker Daemon：它是运行在 Docker 主机上的后台进程，负责管理 Docker 容器和镜像，并响应来自 Docker 客户端的请求。Docker Daemon 接收来自客户端的命令，并根据这些命令创建、启动、停止或删除容器，或者构建和推送镜像。Docker Daemon 还负责在容器运行时，监视容器的状态并记录容器的日志。

（2）Docker Client：它是一个命令行工具，允许用户与 DockerDaemon 进行交互，管理 Docker 容器和镜像。Docker Client 可以通过在终端中运行命令来控制 Docker 守护进程，例如启动和停止容器、构建和推送镜像、创建和管理网络等等。通过 Docker Client，用户可以快速、轻松地部署和管理 Docker 容器化应用程序，提高开发效率和应用程序的可移植性。

（3）Docker Registry：它是一个开源的 Docker 镜像仓库，允许用户存储和分享 Docker 镜像。通过使用 Docker Registry，用户可以轻松地共享 Docker 镜像，并在不同的 Docker 主机之间移动它们。 Docker Registry 提供了一个中央存储库，用户可以将其 Docker 镜像推送到该存储库，并从其他 Docker 主机拉取这些镜像。

（4）Image：Docker Image 是 Docker 应用程序打包的一个可运行的软件包，包含了运行应用所需的所有文件，包括应用程序代码、运行环境、系统工具、库文件和依赖项等。它可以看作是一个容器的模板，用于创建和启动一个 Docker 容器。Docker 镜像移植能力非常强，任何主机只要安装了 Docker 引擎都可以启动 Docker 镜像，这也是 Docker 如此流行的的一个重要原因。

（5）Driver：从前文可知，Docker Daemon 负责处理用户的请求，实际上底层是通过各种 Driver 来实现的。Docker Driver 是 Docker 中用于管理各种数据存储插件的组件。它允许 Docker 运行时使用不同的后端存储系统，包括本地文件系统、网络存储、云存储、对象存储等等。Docker 驱动程序是一个插件式的系统，可以根据需要进行安装和配置。使用不同的驱动程序可以更好地适应不同的应用场景，提高 Docker 容器的可用性和性能。

（6）Network：Network 是 Docker 提供的一种可编程的网络模型，它允许用户创建和管理 Docker 容器之间的虚拟网络，从而实现容器之间的通信和资源共享。Docker Network 提供了一系列的网络驱动程序，包括 bridge、host、overlay 等，可以满足不同场景下的网络需求。用户可以使用 Docker 命令行或 Docker API 创建、删除、连接和断开 Docker 网络。Docker Network 还支持自定义网络和多宿主机容器通信。

2.2 Kubernetes 容器编排技术

Kubernetes 是一款开源的容器编排软件，它的设计灵感来源于 Google 内部的 Borg 系统[28]，可以让用户轻松实现高效的编排功能，是 Google 在容器领域经过多年实践和探索而形成的重要成果。Kubernetes 的出现让容器化部署变得简单高效，开发人员不必再担心应用程序复杂繁琐的运维工作，可以将精力放到代码开发上，在实现敏捷开发的同时还降低了开发成本。在容器编排技术经历了几年的快速发

展之后，Kubernetes 因其具有开源性、多功能性、可拓展性、活跃的社区生态环境以及大量云计算产商支持等多方面的优势逐渐成为了容器编排领域的事实标准[29]。

2.2.1 Kubernetes 架构及组件介绍

Kubernetes 是一个基于容器技术的的主从分布式集群管理系统[30],作为分布式系统它拥有主节点和从节点两类节点，并且通过多主模式提供了集群的可用性。Kubernetes 集群的正常运转由 master 节点负责，同时 master 节点还要管理其它工作节点。Kubernetes 架构如图 2-2 所示。

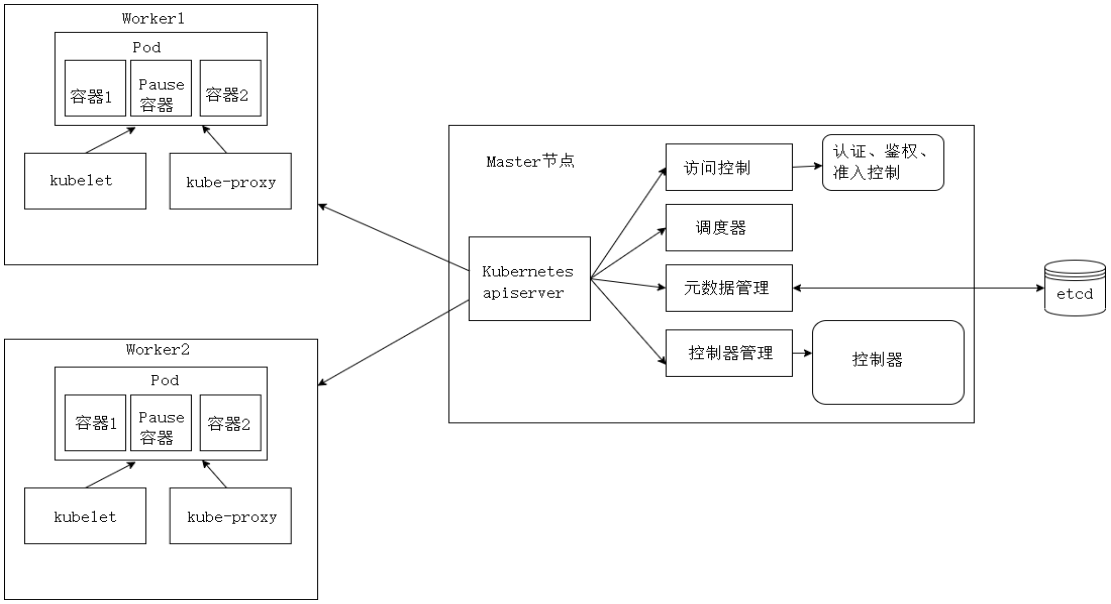

图 2-2 Kubernetes 组织架构

架构图中核心组件的功能介绍如下所示：

（1）APIServer：它是一种用于管理 Kubernetes 核心资源对象的组件，能够实现客户与集群之间的交互，包括访问控制、授权、注册、元数据存储等功能，为客户提供了一个便捷的服务。Kubernetes 的中的各个组件不会直接调用其它组件的服务，都是和 APIServer 这个中间人进行交互让其调用其它组件来完成服务调用的。另外，在 Kubernetes 中只有 APIServer 能够直接访问 etcd，通过 etcd 来维护各种资源对象的元数据。APIServer 提供了访问控制功能，并且还支持 HTTPS，保证集群能够安全运行。

（2）Scheduler：Scheduler 调度器负责集群的调度任务，运行在 Master 节点之上。当用户通过 kubectl 向 APIServer 发起一个新建 Pod 的请求时，APIServer 就会让 Scheduler 调度器将 Pod 绑定到最适当的节点上面。此外，还可以通过 Label 标签和 nodeSelector 属性将 Pod 分配到到指定节点。

（3）Kubelet：它是 Kubernetes 的一个核心组件，运行在每个节点上，负责管理该节点上的容器。kubelet 会定期从 kube-apiserver 上获取 PodSpec，以便根据其中的信息，创建和删除相应的容器，并确保它们处于最佳状态。同时，kubelet 还会检测节点的健康状况，并在节点出现故障时通知其他组件，如 kube-controller-manager 和 kube-scheduler。

（4）ETCD：从上面分析可知，当 Scheduler 将 Pod 分配到某个节点时，需要一个存储中心来记录这个 Pod 分配到了哪个节点，以便 APIServer 后续能够找到这个 Pod。在 Kubernetes 中，这个工作是由 ETCD 数据库完成的，它用来维护集群中各种资源对象的元数据信息，能够保证数据的强一致性，并且通过搭建 ETCD 集群实现高可用。

（5）Controller Manager：在 Kubernetes 中存在许多控制器，总共有八个，比如：Service Controller、Replication Controller 和 Node Controller。APIServer 作为集群的管理中心如果没有一个统一的方式来和这些控制器交互，那将会让集群的管理工作变得十分复杂。Controller Manager 组件负责 Controller 的管理，每个 Controller 通过 List-Watch 机制去监听和监控 APIServer 的事件以及集群中的资源并做出响应的资源状态调整。

2.2.2 Kubernetes 核心资源对象

Kubernetes 将所有内容都视为资源[31]，它拥有多种资源对象，例如 Pod、Service、Namespace 等。Kubernetes API 屏蔽了对这些资源对象底层实现操作细节，用户通过 Kubernetes API 可以简单方便地对这些资源对象进行增删改查操作。接下来介绍 Kubernetes 中的一些核心资源对象。

（1）Pod：Kubernetes 中的 Pod 是最小的可部署单元[32]，它是一个或多个紧密相关的容器的组合。Pod 中的容器共享网络和存储，并在同一节点上运行。每个 Pod 都有唯一的 IP 地址和主机名，因此它们可以直接通信。

（2）Volume：它为 Pod 中的容器提供了一个可访问的文件系统。Volume 可以将容器中的数据持久化到磁盘上，也可以让容器之间共享数据。Volume 的使用不受容器的限制，即使容器被毁坏，其中的数据也不会被自动清除。

（3）Namespace：它是一种资源隔离的机制，允许在同一个 Kubernetes 集群中创建多个逻辑隔离的虚拟集群。Namespace 可以帮助用户将资源进行逻辑分组，实现不同用户或应用程序之间的多租户支持。

（4）Service：上文已经讲到 Pod 会被 Scheduler 绑定到某个工作节点上面，但是外部用户如何去访问这个 Pod 呢，这便是 Service 要完成的工作。与 Docker 的

桥接模式不同，Kubernetes 不会通过端口映射来实现外部访问。Kubernetes 使用了 Service 组件，它将多个相同的 Pod 封装成一个整体向外提供服务，APIServer 会将 Service 的信息写入 ETCD，同时启动 kube-proxy 进程来实现 Pod 地址的代理以负载均衡工作。

（5）ConfigMap：用于存储配置信息，如环境变量、配置文件等，使应用程序的配置可以在容器中被轻松管理。

（6）ReplicaSet：Kubernetes 通过 replicaset 来定义某个服务的 Pod 副本数量，保证集群中 Service 对应的 Pod 的数量总是设置的数目，这些都是通过 RS 指定的模板来创建的。

（7）Deployment：用于管理 Pod 和 ReplicaSet。Deployment 定义了一个期望的状态，Kubernetes 控制器可以确保实际状态与期望状态相同。Deployment 可以用来升级应用程序版本、扩展或缩小应用程序副本数量以及回滚到先前的应用程序版本。Deployment 还支持滚动升级和滚动回滚，这使得应用程序可以平滑地升级和降级，而不会导致停机时间。

2.2.3 Kubernetes 网络模型

Kubernetes 有多种不同的基于 CNI（Container Networking Interface）的网络实现[33]，它的原生网络模型是在 Docker 网络模型基础上演变而来的， Kubernetes 为每个 Pod 分配唯一的 IP 地址，然后通过 Flannel 网络插件实现扁平全通的网络系统。在用户眼中，Pod 就是 Kubernetes 集群中的一个独立物理主机，它和宿主机处于同一个网络环境中。和 Docker 网络相比，这种扁平全通的网络降低了集群网络的复杂度，让 Kubernetes 集群在网络的部署和维护工作上节省了大量的人力成本。

在 Kubernetes 中，每个 Pod 都有一个 Pause 容器。Pause 容器是一个非常简单的容器，它的唯一功能是在容器启动时占用 Pod 的网络命名空间和 IPC 命名空间，然后进入睡眠状态。这样，其他容器就可以与 Pause 容器共享这些命名空间，从而使它们能够相互通信。Pause 容器还负责管理 Pod 的生命周期。当 Pod 中的所有容器都被终止时，Pause 容器也将被终止。这可以确保 Pod 中所有容器的行为是同步的，并且它们都可以一起被创建和销毁。尽管 Pause 容器本身非常简单，但它在 Kubernetes 中扮演了重要角色，因为它允许多个容器在同一个 Pod 中运行，而且它们之间可以共享网络和 IPC 命名空间，这为 Kubernetes 的网络模型提供了基础。

Kubernetes 容器网络发展至今已经出现了许多成熟的解决方案，按照不同的 IaaS 配置、设备以及性能要求，可以构建不同的容器网络，下面分别介绍几种实现方案：

（1）Flannel：它是 Kubernetes 默认使用的网络方案[34]，构建扁平全通的网络，覆盖多种场景，Flannel 还提供了多种后端网络实现，包括 VXLAN 和 Host-GW。

（2）Contiv-VPP：它提供了高性能的数据平面，基于 VPP 实现，支持 Linux 网络命名空间，提供了网络隔离、网络策略、负载均衡等功能。

（3）Calico[35]：提供了基于标准 Linux 网络协议的纯三层网络，通过 BGP 协议路由容器的 IP 流量，实现了高效、可靠的容器间通信和跨主机的网络互连。

（4）WeaveNet：使用 UDP 封装 overlay，并且能够对数据进行加密，如果对数据安全性有很高要求可以采用此方案。

2.2.4 Kubernetes 控制器模式

Kubernetes 控制器模式是一种在 Kubernetes 中用于实现自动化的机制，它允许开发者定义一些期望状态和一个调谐器。该模式的主要作用是监视资源对象的变化，根据期望状态和实际状态之间的差异，触发调谐器执行适当的操作来使它们保持一致[36]。Kubernetes 控制器架构如图 2-3 所示。

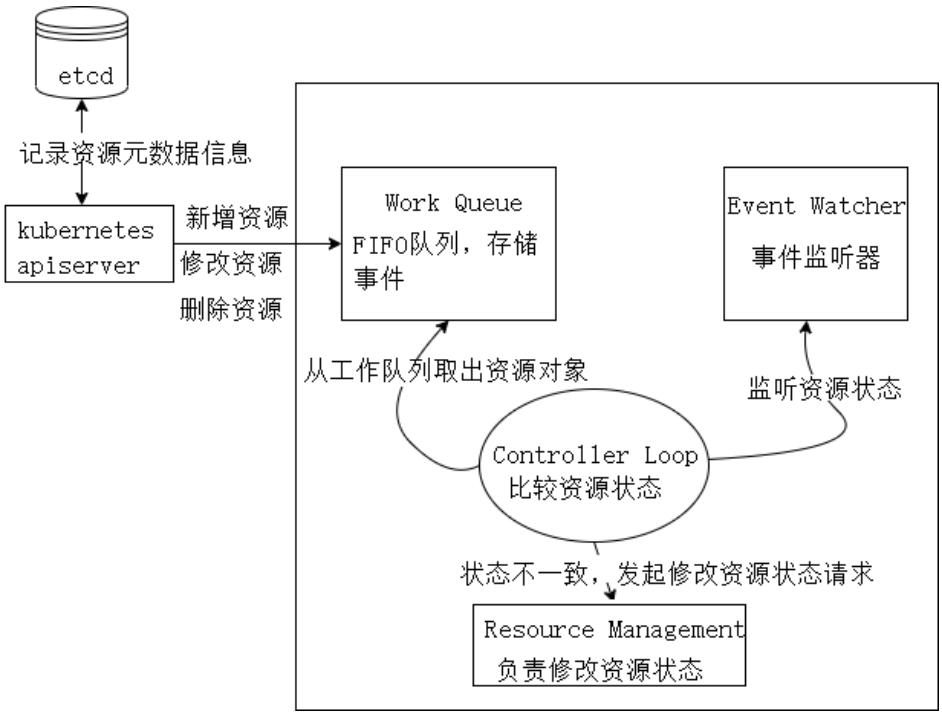

图 2-3 Kubernetes 控制器架构

Kubernetes 控制器设计模式的实现主要基于以下两个核心组件：

（1）Informer：Informer 是 Kubernetes 事件通知系统的一个核心组件，负责监视 Kubernetes 集群中的资源对象，并将资源对象的变化通知给 Kubernetes 控制器。

（2）控制器：Kubernetes 控制器是 Kubernetes 系统中的一个重要组件，它根据预设的规则或策略对集群资源进行控制或管理。控制器的主要功能是确保当前集群中的资源对象与所需的状态一致，以及自动修复任何异常或不一致的状态。控制器还可以基于定义的规则和策略来自动化处理任务，例如自动化扩展和收缩应用程序副本、自动化滚动升级、自动化弹性伸缩等等。

Kubernetes 控制器设计模式中，当 Kubernetes 集群中的资源对象发生变化时，Informer 将会发送通知给对应的控制器，控制器会根据期望状态和实际状态之间的差异，触发调谐器执行适当的操作来使它们保持一致。通常情况下，调谐器的主要功能是创建、更新或删除资源对象，以确保 Kubernetes 集群中的所有资源对象都处于所期望的状态。

使用控制器设计模式，开发者可以基于 Kubernetes 提供的 API 对象，构建各种自动化的解决方案，如创建自定义资源、部署应用程序、实现自动缩放、负载均衡等等。控制器设计模式也是 Kubernetes 内部很多核心组件的实现方式，例如 Deployment Controller、ReplicaSet Controller 以及 StatefulSet Controller 等等。

2.3 本章小结

本章对容器云相关技术进行介绍，包括容器技术以及 Kubernetes 容器编排技术。首先根据实现方式的不同介绍了两类容器技术，接着介绍了目前行业使用最广泛的 Docker 容器技术，分析了 Docker 容器的组织架构以及核心组件，然后介绍了 Kubernetes 容器编排技术，介绍了它的架构以及各个核心组件，并分析了 Kubernetes 的网络模型以及控制器模式。本章对 Kubernetes 相关技术的介绍为接下来的 Kubernetes 强多租户模型的设计与实现奠定了基础。

第三章 系统需求分析与概要设计

在前两章中已经介绍了云计算和 Kubernetes 相关的一些背景和技术，在这一章中将基于这些相关技术对 Kubernetes 强多租户模型的设计需求进行分析，并且给出强多租户模型的整体设计方案与技术路线。本章主要从用户访问控制、计算资源隔离以及多租户网络三个方面入手，设计一个满足 Kubernetes 强多租户模型设计目标的方案。

3.1 强多租户模型需求分析

本文将在 Kubernetes 现有多租户能力基础之上，设计一个强多租户模型解决方案。强多租户模型在设计时要满足两点功能需求：访问控制和资源隔离。访问控制就是在用户进入系统时对其进行身份认证，并为每位用户设置不同的资源访问权限，保证每个用户只能存取其权限范围内的各种资源，从而有效防止非法用户恶意存取租户资源。资源隔离要求租户的计算资源以及网络资源相互隔离，租户只能操作属于自己的资源，不能操作其他租户的资源。本文在设计强多租户模型解决方案时主要考虑了以下几个功能性需求：

（1）明确的租户定义

Kubernetes 强多租户模型大多用在组织关系复杂的企业或组织中，因此强多租户模型需要给出明确的租户定义，但是在 Kubernetes 中并没有租户的概念，Kubernetes 的 Namespace 对象提供了类似于租户的概念，但是在具有复杂层次关系的现实世界中，并不能提供与现实世界有强逻辑关系的租户概念。因此，在设计 Kubernetes 强多租户模型时首先要解决的就是租户定义的问题，多租户的认证授权以及资源隔离都是要以明确的租户定义为基础来实现。

（2）完善的访问控制系统

在 Kubernetes 容器云平台的强多租户模型中，访问控制的设计目标主要涉及用户认证和用户授权两个方面。首先，用户认证就是要保证容器云平台不会被匿名用户非法访问集群中的资源，避免匿名用户盗取或破坏合法租户的数据与资源，保障 Kubernetes 容器云平台的安全运行。Kubernetes 没有提供直接实现普通用户认证的功能，它只提供了一些认证策略，对于 Kubernetes 强多租户模型来说这些认证策略是不够的，所以需要在 Kubernetes 自带的认证策略基础上利用外部用户管理系统来实现多租户的身份认证。另外，Kubernetes 强多租户模型还需要实现用户的自动授权与鉴权功能。用户认证成功后就能够访问资源，但是在强多租户模型中，

普通用户只能访问部分资源，而系统管理员拥有 admin 权限，对集群所有资源都有读写权限。Kubernetes 自身就提供了一些授权功能，但是在复杂的强多租户模型中这些授权功能没办法实现层次化的自动授权与鉴权。通常情况下，每个用户都会绑定一个角色，授权时是直接对角色授权，当用户和角色绑定后就拥有了这个角色所对应的权限。因此，必须设计一种层次化的租户模型来解决强多租户模型自动授权困难的问题。

（3）租户资源的隔离

在 Kubernetes 强多租户模型中，资源隔离是一项重要的任务，它要求租户之间的计算资源和网络资源保持完全隔离，以确保一个租户的操作不会影响其他租户的资源，从而实现资源的合理利用。在 Kubernetes 容器云平台中，多个租户会共享同一个节点上的资源池，计算资源隔离就是要让租户只能操作自己分配到的那部分资源，不能发生资源侵占的问题。如今 Docker 容器是行业内使用最广泛的容器技术，但是金融、银行等对系统安全性有极高要求的领域中，Docker 容器远远不能达到安全容器的要求。除了 Docker 容器以外，Kubernetes 容器云平台还可以选择其它很多满足 CRI 标准的安全容器引擎，同时，还需要在不同的情况下采用不同的满足 CRI 标准的容器引擎，实现可信容器和非可信容器的隔离。另外，强多租户模型还必须要满足租户网络隔离。Kubernetes 利用扁平化全局虚拟网络和 Flannel 插件实现了 Pod 之间以及节点内和节点间的网络互通。但是在强多租户模型中，这种扁平化的全通网络会导致租户对网络资源的竞争，因此需要设计一种方案来隔离租户网络，保证租户的网络隔离。

Kubernetes 强多租户模型除了要考虑上述的功能性需求外还需要考虑非功能性需求。首先，强多租户模型应该具有强大的安全性，以确保每个租户的资源和数据得到保护，并且能够预防可能的攻击或数据泄露。其次，多租户模型应该具有易用性，以便管理员可以轻松地管理多个租户。另外，多租户模型应该具有高性能，以确保每个租户的资源都能得到快速的响应和处理，并且能够支持高负载和高并发的使用情况。最后，多租户模型还应该具有可扩展性，以支持系统的维护升级，便于后续开发的模块能够无缝集成到当前系统中。

3.2 整体架构与技术方案

按照上文提到的设计目标与解决方案，Kubernetes 强多租户模型访问控制和资源隔离方案如表 3-1 所示。

表 3-1 技术路线

设计目标		解决方案
租户对象的定义		拓展的租户 API
访问控制	用户模型	分层用户模型，包括系统管理员、租户管理员、普通用户
	用户管理	采用 KeyStone 进行用户管理
	租户认证	租户认证使用 KeyStone
	租户授权	拓展的 RBAC 控制器
资源隔离	计算资源隔离	Kata 安全容器引擎
	网络资源隔离	使用 Contiv-VPP 网络插件和自定义网络策略
租户网络管理以及 Pod 网络配置		设计 Contiv Plugin、 Network-management 以及 Network-cli 插件

本文设计方案的整体架构如图 3-1 所示，在管理平面新增了租户 API，使得 Kubernetes 能够更加有效地定义租户的概念，同时能够高效地管理租户下的资源。此外，新增的 Contiv-VPP 插件可以将租户网络的初始化、配置等操作进行封装，从而大大简化了复杂地网络配置操作流程。

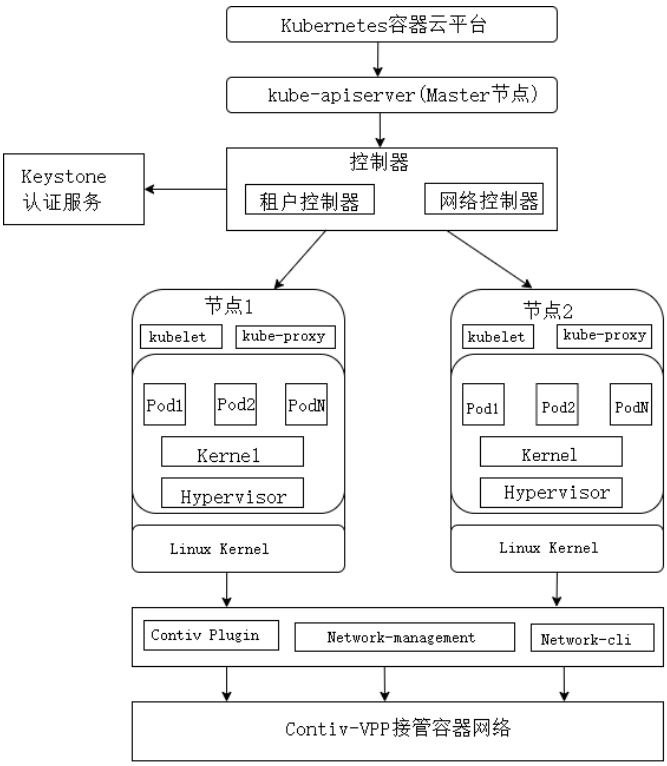

图 3-1 方案整体架构

首先要在 Kubernetes 原生的 API 上拓展出租户 API，实现强多租户模型明确的租户定义目标；其次，构建一个分层的用户模型，方便对用户进行访问控制；接

着，利用 OpenStack 身份服务组件 Keystone 实现用户管理和认证；另外，Kubernetes 提供了非常强大的 RBAC 授权策略，将在此基础之上设计租户授权控制器器，实现用户的自动授权与鉴权。

资源隔离也要设计相应的解决方案，首先，计算资源隔离需要为 Kubernetes 提供新的 CRI 实现，本文将使用 Kata 安全容器引擎。Kata 容器提供了虚机级别的安全性和隔离性，每个容器都拥有独立的内核，解决了多租户环境中应用逃逸到内核后相互影响的问题；另外，利用 Contiv-VPP 网络插件和 Network Policy 网络策略结合的方式来实现网络隔离，同时使用插件化的思想设计 Contiv Plugin、network-management 以及 network-cli 模块完成租户网络的管理以及 Pod 网络的自动配置。

3.3 租户访问控制

3.3.1 Kubernetes 访问控制策略

在 Kubernetes 中，各个组件都是和 APIServer 进行通信的，为了保证集群的安全，APIServer 会对 API 请求进行验证，只有通过验证的请求才能访问集群资源。Kubernetes 的访问控制流程主要由用户认证、鉴权以及准入控制三个步骤组成，完整的访问控制过程如图 3-2 所示。

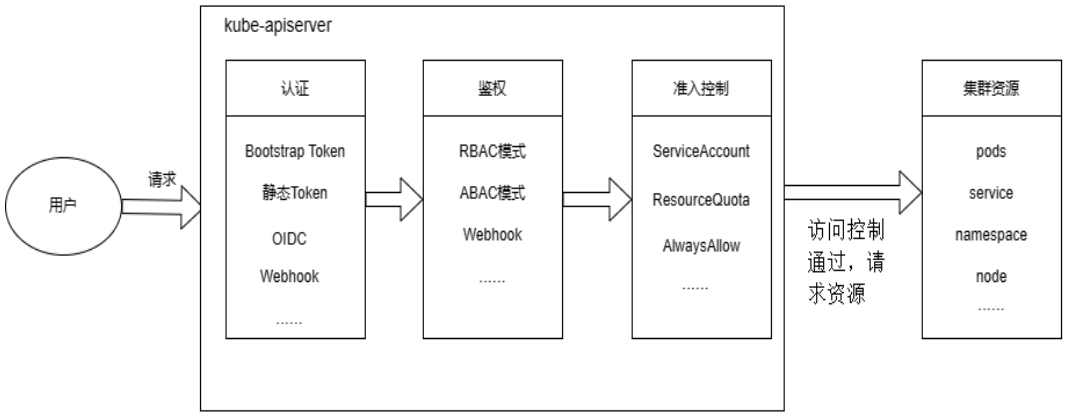

图 3-2 Kubernetes 访问控制过程

在 Kubernetes 中，Service Account 提供了一种 Pod 内部访问集群资源以及控制 Pod 访问权限的方式，但是对于外部用户，Kubernetes 并没有提供代表普通用户的资源对象。考虑到拓展性，Kubernetes 在设计用户认证功能时使用了插件化的思想，通过设计可插拔的认证模块来支持不同的认证方法。请求认证通过后会将用户信息封装到这个 API 请求中，后续会从 API 请求中获取到封装的信息完成用户鉴权。Kubernetes 支持多种认证策略，主要包含以下几种：

（1）基于 TLS 证书的认证

Kubernetes 默认情况下使用 TLS 认证。在 Kubernetes 集群中各个组件和 APIServer 之间通信都是使用 TLS 双向通信[37]，客户端和服务端都需校验对方身份，和单向认证相比，双向认证虽然过程更加复杂，但只有通信双方都认证成功后才能建立安全的通信通道，因此双向认证具备更强的安全性。双向认证具体流程如图 3-3 所示。认证通过后，APIServer 会将用户名、组名、ID 等信息交给授权模块进行下一步的处理。

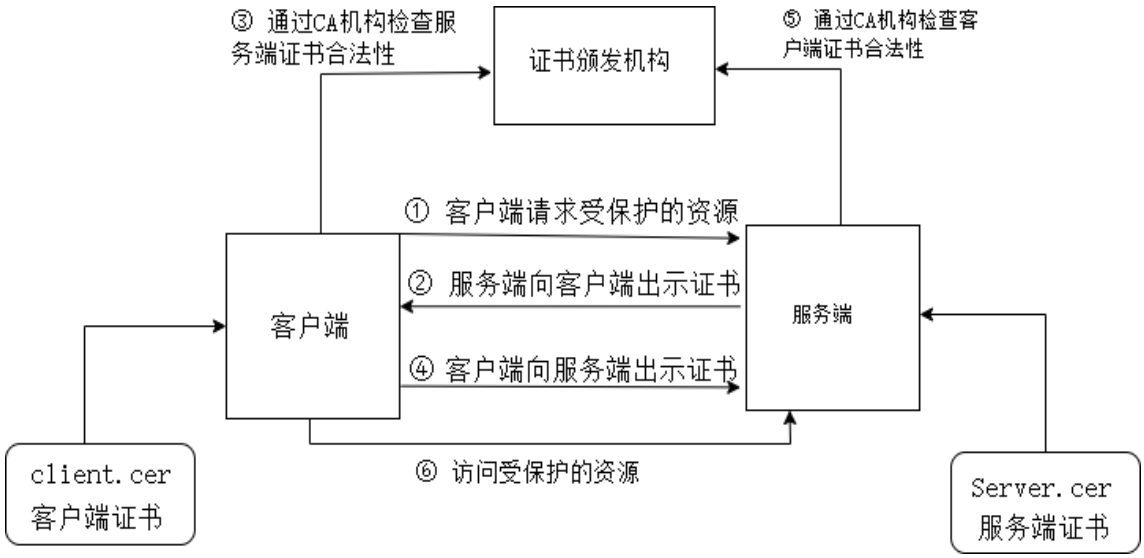

图 3-3 Kubernetes TLS 证书双向认证流程

（2）静态 Token 文件认证

用户向 Kubernetes 发起 API 请求时，认证模块会验证请求中的 Token 信息和静态 Token 文件中的 Token 是否匹配。该方法比较简单，但缺点是 Token 必须定期轮换，且需要手动管理 Token 文件。

（3）Bootstrap Token 认证

该认证方式和静态 Token 文件认证使用的验证方法相同，但是 Kubernetes 集群会自动创建 Token，无需手动管理 Token 文件[38]，更加简单易用。

（4）X.509 证书认证

该认证方法使用基于 X.509 标准的客户端证书作为凭据来验证用户身份。在此方法中，每个用户都有自己的证书，集群管理员需要负责证书的颁发和撤销。由于证书管理难度较大，所以很少使用到这种认证方法。

（5）OpenID Connect 认证

该认证策略允许用户使用基于 OIDC 协议的标准身份验证流程进行身份验证和授权[39]。当用户通过 OIDC 认证策略进行登录时，他们将被重定向到 Identity

Provider 进行身份验证。验证通过后用户将被重定向回 Kubernetes API Server，API Server 将使用令牌来验证并授权用户访问特定的资源。

（6）Webhook 模式认证

Webhook 模式认证允许 Kubernetes API Server 将认证请求发送给外部 HTTP 服务来验证身份和授权信息。Webhook 模式认证可以用于使用自定义认证后端的 Kubernetes 群集，以及与外部认证系统集成。使用 Webhook 模式认证可以很好地扩展 Kubernetes 的认证功能，通过与外部系统集成，可以轻松地实现各种自定义的身份验证和授权方案。

当用户的 API 请求通过认证后就会进行鉴权操作，判断用户是否对当前资源具有访问权限，用户鉴权的目的是确保 Kubernetes 集群的安全性，避免未经授权的用户访问和操作 Kubernetes 资源。Kubernetes 支持多种鉴权策略，主要包括以下几种：

（1）ABAC 策略

ABAC 模式是基于属性的访问控制策略，允许管理员通过定义基于属性的规则来限制 Kubernetes API 对象的访问[40]。Kubernetes 在接收到 API 请求后，会将请求中的用户信息和请求操作与 ABAC 规则进行匹配，从而判断是否接受该请求。当存在该匹配规则时，请求将被允许访问；否则，请求将被拒绝。

（2）RBAC 策略

RBAC 是基于角色的访问控制策略，可以通过定义角色和角色绑定来实现对用户地授权[41]。这种授权方式是基于用户、角色和资源之间的关系进行控制的，能够实现更精细化的授权管理。

（3）Webhook 策略

该模式需要创建一个鉴权服务器，kube-apiserver 会将用户的 API 请求发送给 Webhook 鉴权服务器进行权限鉴定，鉴权完成后服务器会将允许或者拒绝访问的结果发送给 kube-apiserver。这种鉴权方式的优势在于可以使用外部服务来进行更灵活、更细粒度的鉴权控制，例如可以通过集成 LDAP、OAuth 等标准协议来实现更丰富的认证和授权策略。同时，Webhook 鉴权方式还支持异步请求处理，可以减轻 APIServer 的负担，提高集群的鉴权能力。

在 Kubernete 中默认使用 RBAC 模式，RBAC 授权模式由 Role 和 RoleBinding 以及 ClusterRole 和 ClusterRoleBinding 两对资源对象来实现。其中 Role 和 RoleBinding 是针对 Namespace 级别的角色，只能在同一个 Namespace 下起作用；而 ClusterRole 和 ClusterRoleBinding 是集群级别的，作用域涵盖整个集群。Role 可

以看做是一组权限的集合，通过 RoleBinding 将用户和 Role 进行绑定，绑定后该用户就拥有了 Role 包含的所有权限，具体关系如图 3-4 所示。

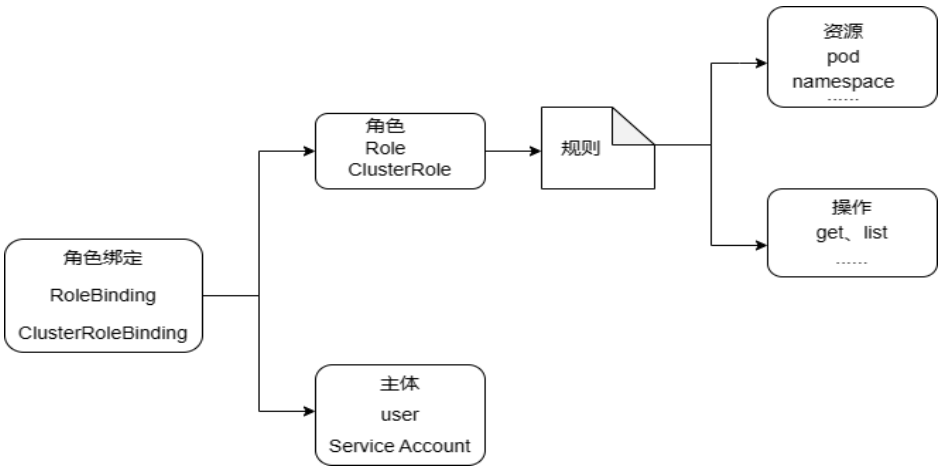

图 3-4 用户、角色以及权限的关系

Kubernetes 准入控制（Admission Control）是一种安全措施，它可以在请求被 Kubernetes API Server 接受之前，对请求进行拦截、审核和转换。准入控制的核心是一个或多个准入控制器（Admission Controller），它们是 Kubernetes API Server 中的插件，用于处理所有的准入请求。准入控制器可以拒绝请求、修改请求或增加新的对象，以便满足特定的需求。Kubernetes 提供了一些内置的准入控制器，如 NamespaceLifecycle、LimitRanger、ResourceQuota 等，同时也支持用户自定义准入控制器。管理员可以利用准入控制对 Kubernetes 集群中的资源和操作进行精细的控制和管理，从而确保集群的安全和稳定。

3.3.2 Keystone 认证组件

Keystone 在 OpenStack 开源软件中负责身份验证、访问控制和服务发现等工作。用户访问资源之前需要验证身份是否合法，执行操作之前需要检查是否具备权限，这些工作都是 Keystone 来完成的。另外，Keystone 作为 OpenStack 的服务总线还能提供服务注册功能，其它服务可以将自己的调用地址注册到 Keystone 中，服务之间的调用都要经过 Keystone 的验证，验证完成后再通过 Keystone 来获取服务调用地址，从而完成调用。用户向 OpenStack 请求资源时都会由 Keystone 完成身份认证工作，为用户生成一个访问令牌，Keystone 负责用户令牌的管理工作。用户向 Glance 请求镜像资源时 Keystone 的工作时序图如图 3-5 所示。请求其它资源工作流程类似，不再赘述。

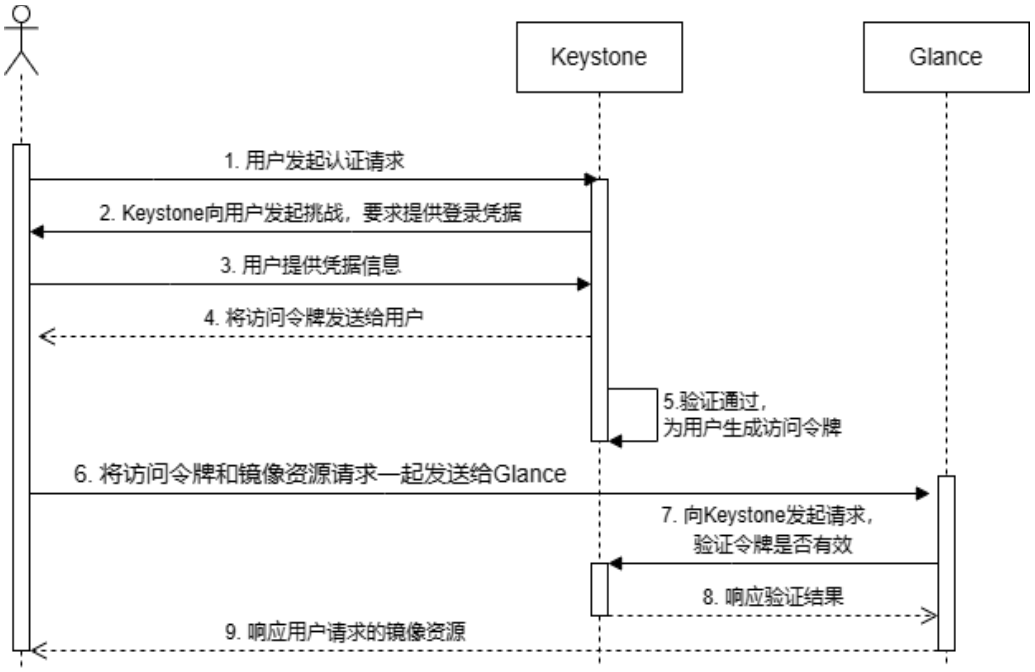

图 3-5 用户请求镜像资源时 Keystone 的工作时序图

3.3.3 租户 API 设计

在 Kubernetes 强多租户模型实现方案中首先要解决的就是租户定义问题，因此需要设计一个租户 API，通过租户 API 来让 Kubernetes 拥有 Tenant（租户）的概念，用户认证、用户鉴权以及多租户网络等功能的实现都要依托于租户对象，本文将借助 Kubernetes 提供的拓展机制来设计租户 API。Kubernetes 的强大拓展能力使其在众多容器编排技术中脱颖而出，它提供了一系列灵活的拓展机制，使得用户可以轻松实现多种功能，例如网络、存储、API 拓展等。这些拓展机制能够方便用户无侵入地自定义 Kubernetes 容器云平台的功能。Kubernetes 提供了两种 API 的拓展方式：CRD（自定义资源定义 API）和聚合 API。它们之间的比较如表 3-2 所示。

表 3-2 CRD 与聚合 API 的比较

CRD	聚合 API
不用编程	大量编程且需要编译出可执行文件
无需额外服务，由 API 服务器处理。	需要创建额外服务且服务可能失效
支持默认值，需通过 Webhook 来实现。	支持默认值设置
不支持定制存储。	支持定制存储
不支持多版本管理	支持多版本管理
通过 Webhook 来实现合法性检查	支持任何合法性检查
主控节点升级时自动进行缺陷修复	需要周期性的进行缺陷检查并修复
通过 Webhook 实现业务逻辑定制	支持自定义业务逻辑

通过上述分析发现，如果要拓展的 API 包含大量不同种类的字段，在操作资源对象时要进行任何类型的合法性检查并且还要支持多版本管理和业务逻辑自定义，那就通过 API 聚合进行拓展；相反，如果对象包含的字段比较少，涉及的业务逻辑比较简单，则通过 CRD 拓展更为方便。

定义租户资源对象时需要考虑以下几点：

（1）租户资源管理简单便捷：在强多租户模型中，租户与租户之间不能相互影响，要互相隔离资源，租户 API 与资源管理 API 相结合，使得租户资源管理简单便捷，并且对 Kubernetes 原生 API 无侵入。

（2）用户管理 API 简单易用：在 Kubernetes 容器云平台中，企业购买云平台的资源成为集群中的一个租户，该企业下拥有许多员工，这些员工是租户下的用户，在大型企业中员工众多，导致租户下用户数量繁多，租户负责用户的管理，因此设计租户 API 时需要考虑到用户管理的难度，尽可能设计出方便快捷的 API。

（3）租户状态快速查询：租户管理员负责管理租户资源，需要经常查询租户当前状态，检查租户资源是否处于可用状态，当租户资源不可用时会做出相应的故障恢复操作。因此为了方便租户管理员查询租户的状态，在设计租户 API 时应该设置相应的状态字段来展示租户状态。

Kubernetes 的 Namespace 提供了租户的逻辑隔离能力，将 Namespace 和租户资源进行映射，当系统管理员创建租户时，Kubernetes 会创建该租户对应的 Namespace 资源对象，通过对 Namespace 资源对象进行操作间接实现对租户对象的管理。租户对象包含的字段如表 3.3 所示，租户状态字段取值如表 3-4 所示。

表 3-3 租户对象各字段含义

字段名	描述
username	新建租户的用户名
password	新建租户的密码
uid	唯一标识，通过 uid 复用 Keystone 已有租户
state	租户资源的状态
message	租户资源处于当前状态的原因

表 3-4 租户对象状态字段取值

取值	描述
new	租户资源通过合法性校验刚被系统接收
handling	租户资源正在被租户控制器处理
available	租户资源可用
unavailable	租户资源不可用
Terminated	租户资源已经删除

当发起新建租户对象请求时，首先 kube-apiserver 会对该请求进行合法性检测，在检测过程中租户对象处于 new 状态；检测通过以后就会由租户控制器进行租户、Namespace 等资源创建工作，在创建成功之前租户资源一直处于 handling 状态；创建成功之后就可以在集群中使用该租户资源了，此时租户可以对外提供服务，处于 available 状态；如果创建失败，则会利用 msg 字段来反映该租户不可用的原因并且租户对象会进入 unavailable 状态，表示不能对外提供服务；当系统管理员要删除租户时就会向租户控制器发起删除请求，Kubernetes 集群会删除该租户对应的 Namespace 等资源对象，删除成功以后就会进入 terminated 状态。租户资源的状态流转如图 3-6 所示。

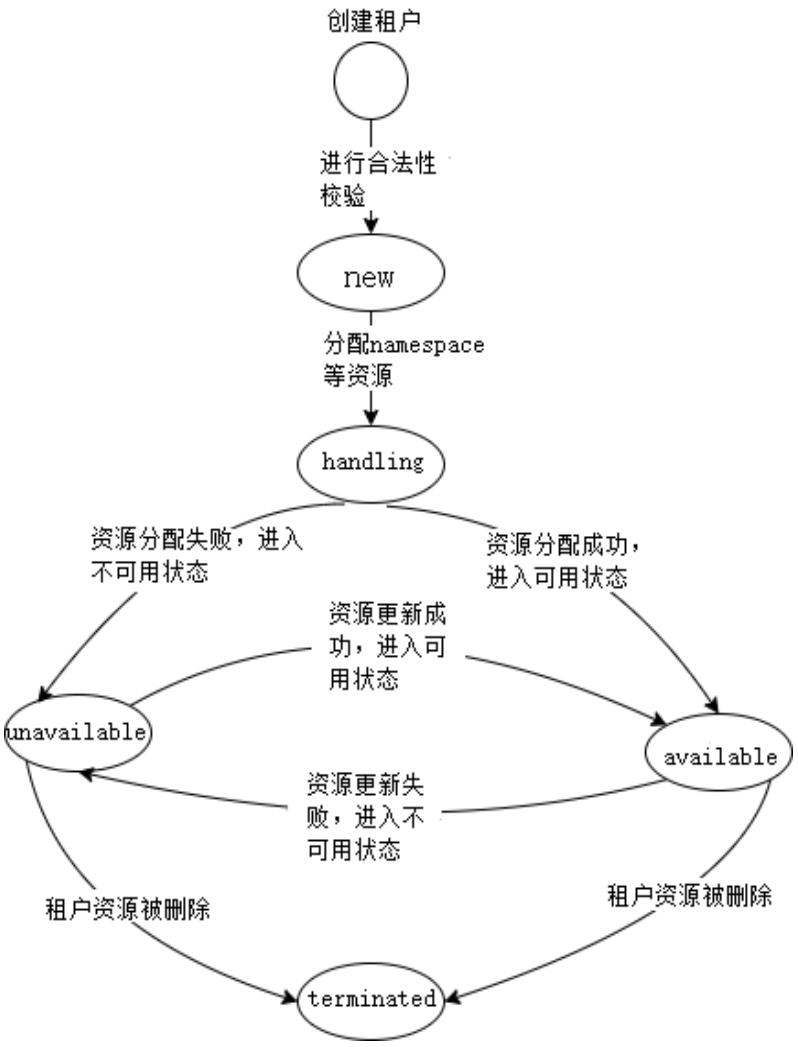

图 3-6 租户资源状态流转过程

3.3.4 租户管理模型设计

为了达到前文介绍的设计要求，需要设计一个合理的租户管理模型。在 Namespace 对象提供的隔离能力基础之上，可以设计如下的租户管理模型：每个租户在集群中拥有一个或多个 Namespace 资源对象，租户负责管理自己 Namespace 下的所有资源，在租户内会存在多个用户，他们能够共享该租户的资源；通过租户控制器来对租户下用户进行授权，限制用户的资源使用范围；当用户发起 API 请求时会进行鉴权，保证集群不会被恶意请求攻击。具体模型如图 3-7 所示：

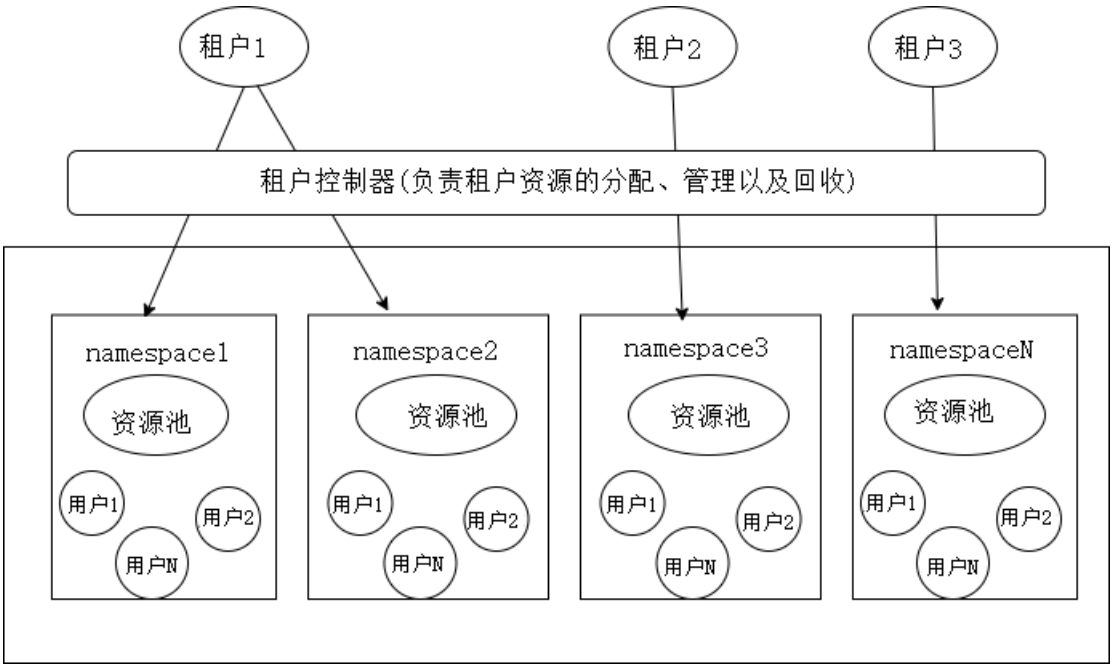

图 3-7 租户管理模型

在 Kubernetes 强多租户模型中将用户根据不同的权限划分成不同的类别，在方案中使用了三层用户模型：

（1）系统管理员：在集群中拥有 admin 权限，负责集群中所有资源的管理和控制，能够创建、删除租户以及租户下的用户。

（2）租户管理员：由系统管理员创建出租户管理员并和某个租户进行绑定，使其具备对该租户内的 Role、RoleBinding 等资源对象进行增删改查的能力，租户管理员还要负责租户内用户的授权工作。

（3）普通用户：能够和角色进行绑定，从而获取相应权限。

用户模型关系如图 3-8 所示，系统管理员具有最高权限，负责集群中所有用户的管理，它能够在租户下添加一个普通用户，同时也能将一个用户设置为租户管理员。系统管理员会为每个租户创建一个租户管理员，租户管理员拥有租户内的

admin 权限，负责租户内各种资源的管理工作，同时租户管理员还通过给普通用户绑定 role 实现权限分配，而普通用户默认不分配任何权限，只有租户管理员为其分配权限后他才能访问集群中的资源。

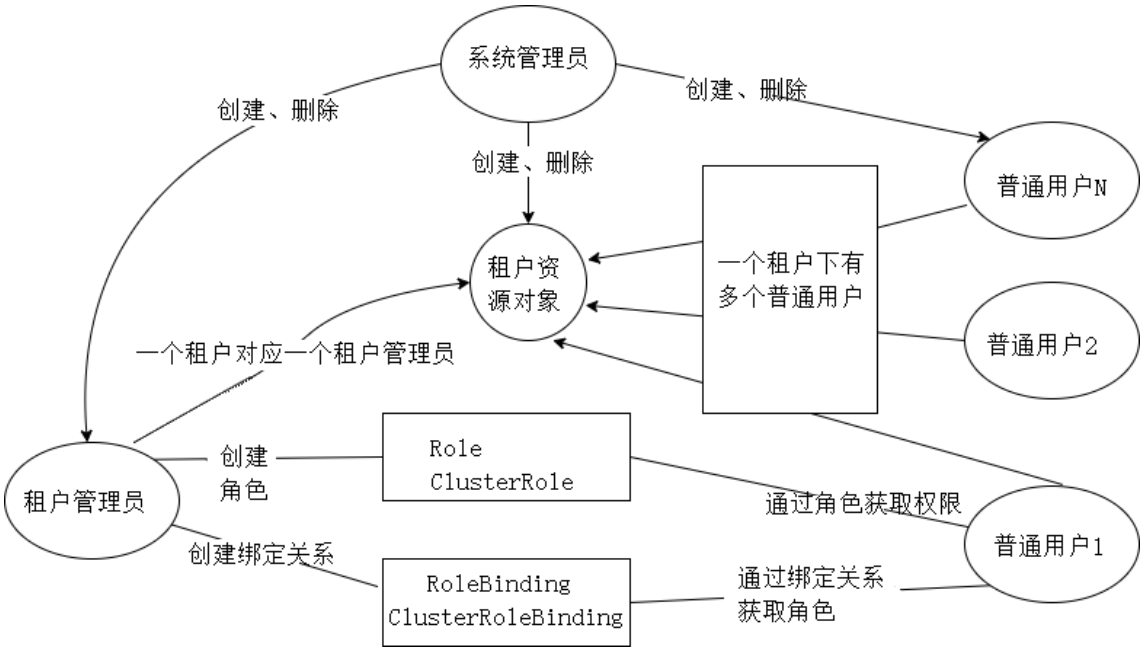

图 3-8 三层用户模型

3.3.5 访问控制设计

在 Kubernetes 容器云平台中，访问控制是至关重要的，特别是在多租户模式下，为了防止用户对集群资源的攻击和破坏，必须提供强大的访问控制功能，以保证系统的安全与稳定。首先，强多租户模型需要提供透明的用户管理，在用户无感知的情况下拓展用户接口来完成用户的管理。Keystone 是 OpenStack 中负责用户认证、授权的组件，在 Kubernetes 强多租户模型中作为外部用户管理系统，能够达到透明、无侵入的要求[42]。其次，Kubernetes 提供了许多原生的授权策略，为了在强多租户模型中实现授权透明化，需尽量采用原生的授权方式。在本文的方案中使用了 RBAC 模式，能够提供基于角色的授权能力，同时对 Kubernetes 原生的授权系统没有任何侵入性。最后，Kubernetes 提供了十分出色的准入控制功能，在本文方案中继承 Kubernetes 原生的准入控制方案，最大限度减少对 Kubernetes 地侵入。强多租户模型的访问控制模块应该达到以下目标：

（1）层次化用户管理：需要对用户进行多层次的管理，将用户进行分类，包括系统管理员、租户管理员和普通用户，以便实现不同级别地权限划分。

（2）多租户认证：需要支持多租户的认证，避免匿名用户进入系统对合法租户的资源进行恶意破坏，阻止所有非法用户访问集群资源。

（3）精细化权限控制：需要对 Kubernetes 集群中的各个资源进行精细化的权限控制，保证用户只能访问权限内的资源。

（4）灵活性：能够支持自定义访问控制策略，以满足各种应用和业务需求。

（5）可拓展性：需要具备较强的可扩展性，以支持更多的访问控制策略，并且能够适应集群规模的变化。

本文基于 Keystone 组件提出了强多租户模型的访问控制方案，具体流程如图 3-9 所示。Kubernetes 可以通过 Webhook 的方式与 Keystone 集成，集成成功后 Keystone 将成为 Kubernetes 集群的认证服务器，Kubernetes API 请求会被转发到 Keystone 进行认证。作为认证服务器，Keystone 将全面托管 Kubernetes 集群的身份认证服务，并作为外部用户管理组件进行访问控制。用户发起 API 请求前需要先向 Auth Server 发起登录请求，登陆成功以后 Auth Server 将会返回给用户一个 Token 信息，接着用户将携带 Token 信息发起 API 请求，Auth Server 会对用户进行 Webhook 认证并返回响应信息，如果用户认证成功那么就会由 Auth Server 对用户请求进行鉴权，检查其是否具备访问集群资源的权限，如果鉴权成功，那么用户的请求就会通过，顺利访问到集群资源，否者鉴权失败，用户请求就会被拒绝。鉴权通过以后，Api-server 按照配置的准入控制策略对用户请求进行处理，对集群资源进行相应的操作，访问控制过程结束。

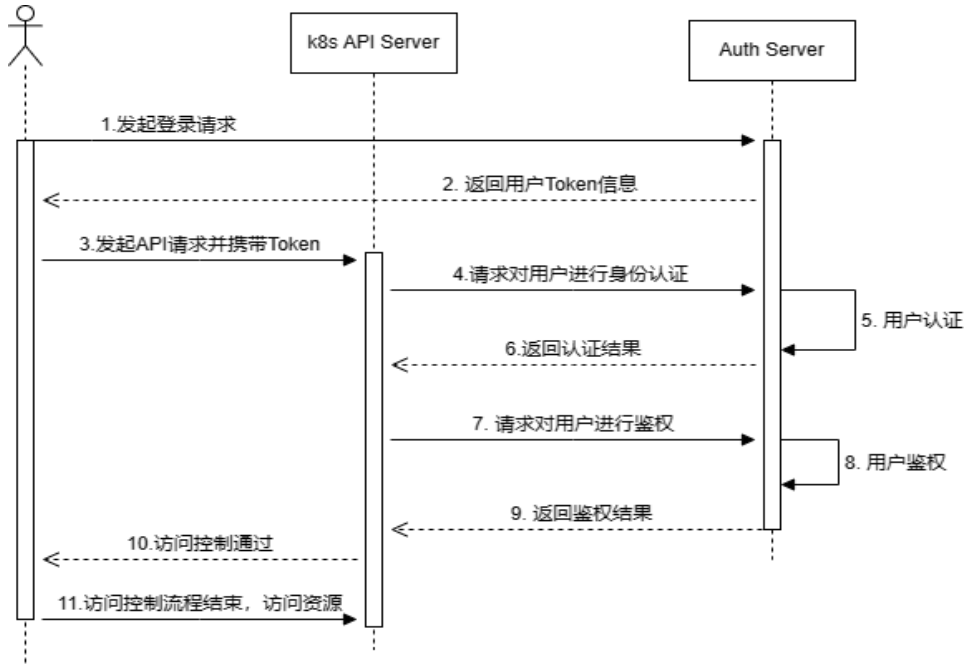

图 3-9 访问控制时序图

和 Kubernetes 不同，Keystone 默认就有租户和用户的概念，并且它还提供了非常完善的 API 来管理租户以及用户。通过将 Keystone 组件集成到 Kubernetes 中，

使得 Kubernetes 具备了强多租户访问控制能力。另外，考虑到应尽量减少对 Kubernetes 的侵入性，所以在方案中并没有使用 Keystone 的自动授权机制，而是对 Kubernetes 原生的 RBAC 授权模式进行封装，设计出了 RBAC 控制器来实现用户鉴权功能。

3.4 租户计算资源隔离

3.4.1 Kata 安全容器引擎

伴随着云计算行业的快速发展，容器技术在不受信任的多租户环境中如何保证安全工作遇到了挑战。 Kata Containers 利用开源管理程序作为每个容器（或每个 Pod）的隔离边界，解决了现有裸机容器方案共享内核所带来的困境。Kata 容器非常适合按需、基于事件的部署，例如：CI/CD 领域、7*24 小时运行的 Web 服务器应用程序[43]。和传统容器相比，Kata 容器为用户的应用提供了更强的安全性、更好的拓展性以及更高的资源利用率。

Kata Containers 使用轻量级虚拟机构建安全的容器运行时，这些虚拟机在感觉和性能上都像容器，但使用硬件虚拟化技术提供了更强大的工作负载隔离。Kata 容器是在 Clear 容器和 Hyper 容器的基础之上研发出来的，吸纳了它们的优势，同时包含二者的优点，并扩展了对主要架构的支持，包括 AMD64、ARM 和 x86_64 等。Kata Containers 还支持多种管理程序，包括 QEMU、Cloud-Hypervisor 和 Firecracker，并与 containerd 项目集成。

从图 3-10 中可以看出，Kata 容器和 Docker 等传统容器最主要的区别在于每个容器或 Pod 都是独立的管理单元，拥有独立的内核，不存在内核共享，和传统虚拟机一样安全，同时，Kata 与容器生态系统无缝集成，实现了强隔离。

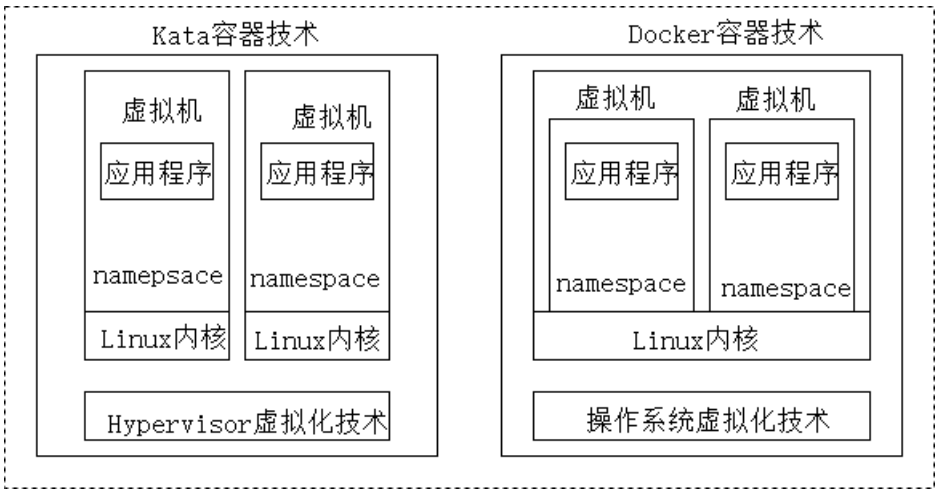

图 3-10 Kata 容器与 Docker 容器的主要区别

3.4.2 计算资源隔离设计

Kubernetes 容器云平台中计算资源的隔离主要包含 CPU 和内存等资源的隔离，集群中的各个节点共同组成了资源池，底层则是依托于容器引擎来实现的，因此计算资源的隔离实际上就是在容器层面上做到多租户之间的隔离。在容器隔离技术中有三个核心技术：Namespace、cgroups 和 rootfs。Namespace 是一种 Linux 内核特性，用于提供一种资源隔离的机制，将全局的系统资源抽象为多个独立的命名空间，使得在同一台机器上运行的不同进程能够共享系统资源，同时相互隔离。cgroups 也是一种 Linux 的内核功能，用于将一组进程限制在一定的系统资源使用范围内。它可以控制进程的 CPU、内存等资源的使用量，从而实现系统资源的限制、管理和隔离。Docker 容器便是通过 cgroups 实现资源控制的。容器文件系统通常会基于 rootfs 进行修改和定制，创建一个独立的、隔离的文件系统环境，使得容器可以运行在一个隔离的、独立的文件系统环境中，不受宿主机文件系统的影响。在 Kubernetes 强多租户模型中，除了上述容器隔离技术，还需要从容器运行时进行考虑。在本文的方案中，对于非可信容器的运行时环境采用 Kata container；而可信容器则直接使用 Docker container 的方式运行。

Kubernetes 默认使用 Docker 容器，但是不同租户创建的 Docker 容器可能会发生资源侵占的情况，导致租户间计算资源相互影响，甚至导致容器云平台运行崩溃，这加快了 OCI 标准下安全容器的发展，Kata 安全容器就是在这样的背景下得到快速发展的。Kata 会为每个容器单独创建一个虚机，如果是工作在 Kubernetes 下为每个 Pod 创建一个虚机，每个容器拥有独立的内核，解决了容器内应用逃逸到内核后影响其他容器的问题，实现了虚机级别的隔离性和安全性。计算资源隔离的架构如图 3-11 所示。

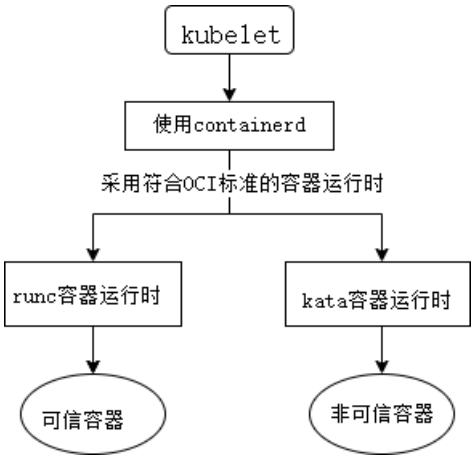

图 3-11 计算资源隔离架构

3.5 租户网络

在本文的设计方案中，一个租户对应一个 Contiv 中的 Network 资源，租户下可以有多个 namespace 命名空间，可以在租户下为每一个 namespace 划分一个子网，每个 namespace 对应一个子网，如果不在租户下配置通信路由，那么同一租户下的 namespace 是隔离的。本文将在 Contiv-VPP 的基础之上，使用插件化思想设计了 Contiv Plugin、network-management 以及 Network-cli 模块，随插随拔，对 Kubernetes 原生系统没有侵入性，并且网络管理模块升级维护也更加简单高效。

3.5.1 Contiv-VPP 技术分析

Contiv-VPP 是 Kubernetes 的一款网络插件，它使用了 FD.io/VPP 在 Kubernetes 集群的 Pod 之间实现了网络互通功能[44]。Contiv-VPP 包含多个组件，它会将自己部署在 kubernetes 的 kube-system 命名空间，其中 contiv-skr、contiv-crd 和 contiv-etcd 会部署在 master 节点上面，而 contiv-cni、contiv-vswitch 和 contiv-stn 组件则是部署在集群中每个节点上面。Contiv-VPP 的各个组件能够和 Kubernetes 完美对接，并且当集群状态发生改变时能够通过 kubernetes 的 API 进行重新编译部署。VPP 平台是 contiv-VPP 中最重要的组件，它能够提供简单易用的交换机和路由器功能。在 Kubernetes 集群的每个节点上都会运行一个 VPP 引擎，通过它来实现节点内 Pod 之间互通、主机与 Pod 互通以及节点外与 Pod 互通。Contiv-VPP 使用 DPDK（数据平面开发套件）快速访问网络 IO 层，从而提升对网络数据的处理能力。VPP 完全支持 Kubernetes 的服务和各种策略，不需要将数据包转发到 Linux 的网络堆栈中，因此 Contiv-VPP 非常高效并且具有很强的拓展性。Contiv-VPP 的架构如图 3-12 所示。

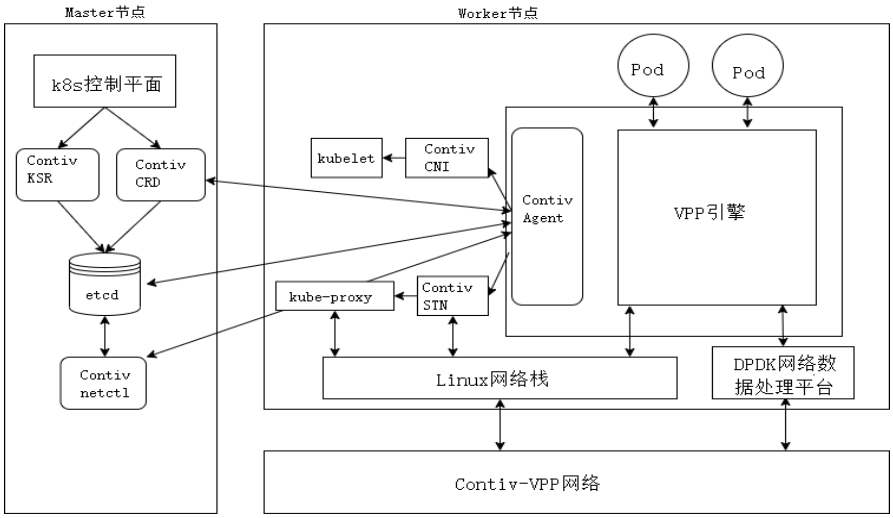

图 3-12 Contiv-VPP 架构图

（1）Contiv KSR：它是一个 Kubernetes 的代理组件，用于将集群状态同步到 ETCD 中，并实现高级网络服务。Contiv KSR 主要有两个功能：状态同步和网络资源映射。状态同步就是当 Kubernetes 中资源对象（如 Pod、Service 等）状态发生变化时，Contiv KSR 会讲这些变化自动同步到 Contiv 网络中，这样就能根据 Kubernetes 资源对象的变化来自动调整网络服务的配置，从而确保集群网络的高可用性和可靠性。网络服务映射就是将 Kubernetes 中的对象映射为 Contiv Networking 中的网络对象。例如，Service 对象被映射为 Contiv Networking 中的虚拟服务，Pod 对象被映射为 Contiv Networking 中的网络端点。通过网络服务映射，Contiv Netwoking 就能利用 Kubernetes 对象的信息提供负载均衡、流量控制、服务发现等高级功能。

（2）Contiv CRD：它可以处理 Kubernetes API 中的自定义资源，涵盖了节点的各个配置，比如：IP 地址和默认网关等。Contiv CRD 可以帮助 Contiv Networking 自定义资源和行为，并集成到 Kubernetes 中，以便 Kubernetes 可以更好的管理和操作 Contiv 网络。

（3）Contiv ETCD：Contiv-VPP 插件使用 ETCD 分布式键值数据库来存储 KSR 反馈的 Kubernetes 集群网络配置和状态信息，这些数据会被运行在各个节点上的 Contiv vSwitch 组件访问。它作为数据存储和同步服务可以帮助 Contiv Networking 实现多节点的数据同步和高可用性，并且保证数据的一致性。

（4）Contiv vSwitch：它是一款基于 VPP（Vector Packet Processing）的虚拟交换机，能够实现 Pod 间网络互通，被部署在 Kubernetes 集群的每个节点之上。在 Contiv 网络方案中提供了可编程的虚拟交换机功能，通过 Network Namespace 和 Label 实现强多租户隔离，同时它还支持 VLAN、VXLAN 和 GRE 等多组网络虚拟化技术。

（5）Contiv CNI：它是一个简单的二进制文件，实现了容器网络接口 API，并在 POD 创建和删除时由 Kubelet 执行。CNI 二进制文件将请求封装到 gRPC 请求中，并将其转发到运行在同一节点上的 Contiv Agent 组件，然后由其进行处理，并回复响应，最后再将其转发回 Kubelet。

（6）Contiv STN：它是 Contiv 网络解决方案的一部分，允许 Pod 访问物理网络，它提供一种实现单个 Pod 和物理网络通信的方法，同时还提供了实现了单个 Pod 与外部网络的路由的方法。STN 使用 DPDK（Data Plane Development Kit）驱动程序控制物理网络适配器，以在主机和 Pod 之间实现零拷贝数据传输。STN 是 Contiv 网络解决方案的关键组件之一，可以让 Pod 直接访问数据中心网络，从而在大规模 Kubernetes 部署中提供更高的网络性能和更好的可靠性。

3.5.2 多租户网络设计

Kubernetes 容器云平台中默认使用 Flannel 网络插件构建一个扁平化的全通虚拟网络，在同一个节点以及不同节点上都能够实现 Pod 间的互相通信。然而在 Kubernetes 强多租户模型中，每个租户应该拥有自己独立的网络，租户之间网络互不影响，互相隔离。

在强多租户模型的容器网络方案中，有很多种实现方案，比如：Flannel 插件、Calico 插件以及 OVS 插件。这些网络插件底层要么是路由方案，要么就是 Overlay 覆盖网络方案，它们在网络多租户的支持方面都存在缺陷。在本文的设计方案中使用了 Contiv-VPP 插件为每个租户构建独立的网络。Contiv-VPP 是一个开源的容器网络解决方案，它基于 Contiv 网络模型和 Vector Packet Processing（VPP）技术，提供了高性能、灵活的容器网络方案。Contiv-VPP 的主要作用是为 Kubernetes 等容器编排平台提供一个可插拔的、高性能的网络插件，用于实现容器之间的通信和网络隔离。同时，Contiv-VPP 支持 Kubernetes 原生的网络策略，利用 Contiv-VPP 构建的容器网络能够满足强多租户模型对租户网络隔离的要求。

在本文的设计方案中，Contiv-VPP 网络插件会接管 Kubernetes 的网络环境，它将租户划分到不同的虚拟局域网（VLAN）中，每个 VLAN 之间是隔离的，从而实现租户间网络隔离。Contiv-VPP 网络插件在运行时会接替宿主机网卡的工作，使用 VPP 引擎进行数据转发实现网络增速，该方案在 Kubernetes 网络接口下使用了插件化设计模式，兼容程度高，IP 地址分发效率高，Pod 间网络数据转发快，能够保证租户创建 Pod 时能够加入到网络中。本文的网络资源隔离模块能够独立升级，具有很强的拓展性。具体的多租户网络拓扑关系如图 3-13 所示。

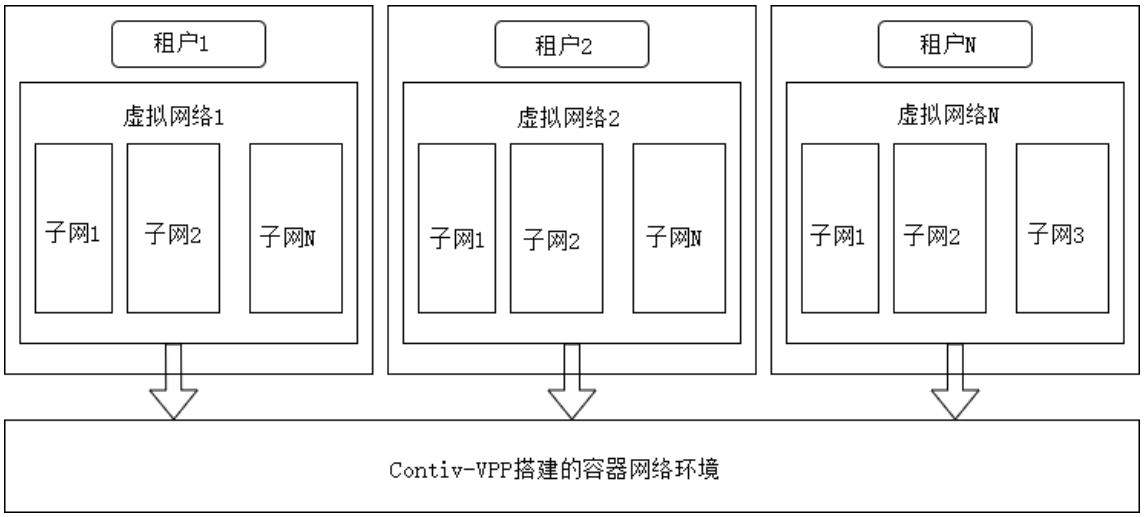

图 3-13 租户网络关系图

在创建租户时就要分配相应的网络资源，Contiv-VPP 插件还可以对租户网络进行更精细地划分，它可以在租户下创建不同的子网网段，一个子网网段可以让多个用户来使用，从而实现租户下各个项目之间的网络隔离。通过 Network Policy 网络策略和 Contiv-VPP 插件集成的方式构建 Kubernetes 集群的容器网络，能够满足强多租户模型对网络隔离的要求，具体的网络架构如图 3-13 所示。

从图 3-14 可以发现，租户路由下会挂载一个 Namespace 级的 vSwitch（虚拟交换机），它会为租户创建不同的子网，同时 Namepsace 下的 Pod 间能够实现互通，并且可以利用自定义网络策略来对 Pod 间流量进行控制。网络数据自顶向下进行传输，Contiv-KSR 接收来自 Kubernetes 控制平面的数据后将其存入 ETCD 中，然后会进行数据格式转换，Contiv Agent 将 ETCD 中存储的数据按照指定的格式策略转换成 VPP 引擎能够识别的内容，最终这些数据都会交给 VPP 引擎来处理。

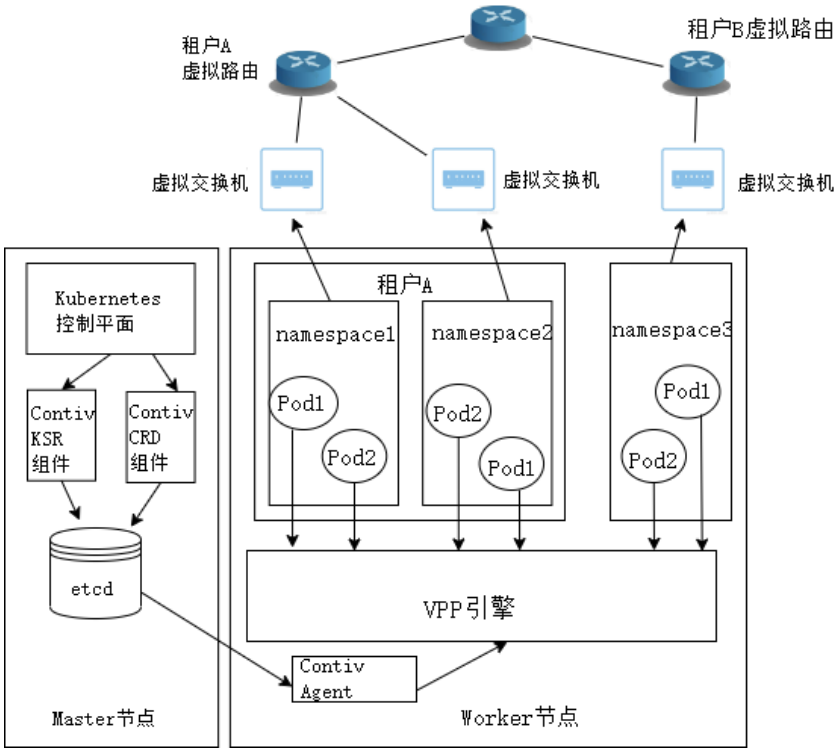

图 3-14 多租户网络架构

3.5.3 Contiv Plugin 模块设计

上一小节介绍了如何使用 Contiv-VPP 实现多租户的网络隔离，但是没有给出 Contiv-VPP 自动化管理 Kubernetes 集群网络的方案。为了让网络管理简单高效，本小节将基于 Kubernetes 插件化的设计思想设计一个可插拔的 Contiv-VPP Plugin 模块。

在 Kubernetes 集群中，每个节点都有一个 kubelet，它负责 Pod 和容器的管理，在集群启动时配置 kubelet，让 kubelet 使用 Contiv-VPP Plugin 来管理 Pod 以及容器的网络，从而实现 Pod 网络的自动配置。Contiv-VPP Plugin 的启动流程如图 3-15 所示，主要包括以下过程：

（1）Kubernetes 集群初始化完成开始启动每个节点上的 kubelet 组件。

（2）检查节点上是否配置了网络插件，如果没有配置则不使用网络插件，否者读取所有配置的网络插件。

（3）检查读取到的网络插件中是否包含 Contiv Pugin，如果没有则对其它网络插件进行初始化。

（4）读取到 Contiv Plugin 插件，对其进行配置并完成初始化。

完成上述步骤后，Contiv-VPP 就已经启动成功，可以在 Kubernetes 集群中对网络进行管理。

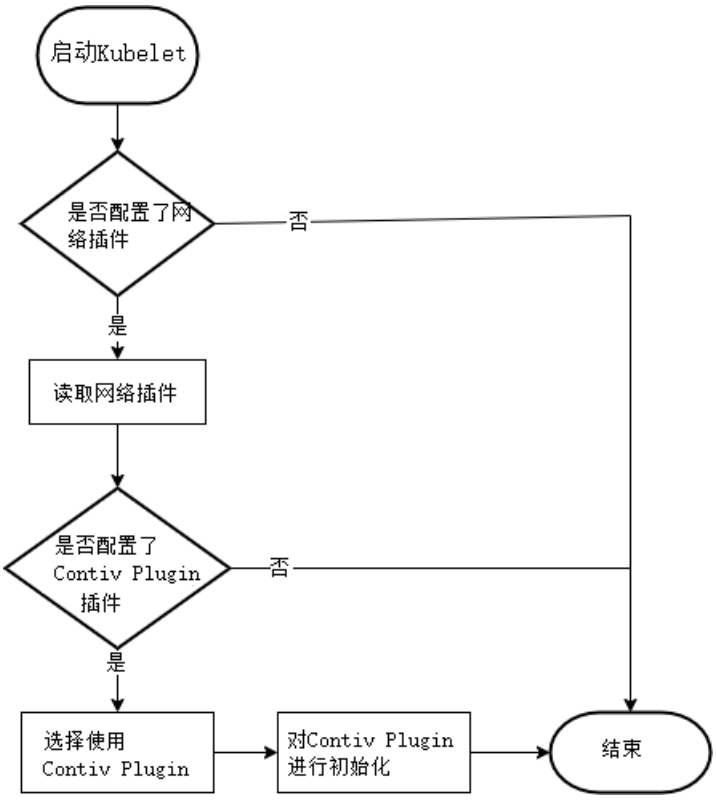

图 3-15 Contiv Plugin 启动流程

在本文的设计方案中，Contiv-VPP Plugin 将作为 Client 端使用 gPRC 框架来发起请求。gRPC 是一种高效的远程调用框架，它支持跨语言和多种语言，由 Google 开发[45]，可以实现高效的远程调用功能。gRPC 使用 Protocol Buffers 作为数据交换

格式，支持多种编程语言，并且具有自动生成代码、流式处理、双向流式处理、TLS/SSL 加密等特性。gRPC 的工作过程如下：

（1）定义服务和消息：首先，需要使用 Protocol Buffers 语言定义 gRPC 服务和消息格式。定义完之后，可以使用 gRPC 插件生成相应语言的代码，服务和消息定义格式如表 3-5 所示。

表 3-5 gRPC 服务和消息定义的格式

定义一个用于传输用户信息的服务，包括用户 ID、姓名和年龄，服务名为 UserService。

syntax = "proto3";
message UserRequest {
  int32 id = 1;
}
message UserResponse {
  int32 id = 1;
  string name = 2;
  int32 age = 3;
}
service UserService {
  rpc getUser (UserRequest) returns (UserResponse) {}
}

（2）实现服务接口：使用生成的代码实现服务接口。

（3）启动 gRPC Server：在服务端，需要启动 gRPC Server，并将服务实现类注册到 Server 中。

（4）创建 gRPC Client：在客户端需要创建 gRPC Client 并连接到 gRPC Server。

（5）调用远程方法：在客户端，可以使用 gRPC Client 调用远程方法，传递请求消息并等待响应。

（6）接收响应：当 Server 接收到请求后，会根据定义的方法，调用相应的服务，并将响应消息返回给 Client。

（7）关闭连接：当调用完成后，Client 和 Server 都可以关闭连接，释放资源。

总之，gRPC 使用 Protocol Buffers 定义服务和消息格式，利用底层协议 HTTP/2 进行数据传输和流控制，并使用 TLS/SSL 加密保证通信安全。在调用过程中，客户端和服务端都可以使用同步和异步方式进行通信，同时支持流式处理和双向流式处理，能够满足不同场景下的需求。

从前面的介绍可知，Contiv Plugin 作为 gRPC 的客户端会向 network-management 发起网络配置请求，请求主要包含租户网络（Network）、子网（Subnet）以及 Pod 网络三个方面，因此 Contiv Plugin 需要提供接口方法来实现各个网络的配置。Contiv Plugin 需要提供的接口服务如表 3-6 所示。

表 3-6 Contiv Plugin 提供的部分 gRPC 接口

接口方法	描述
initContivPluginClient	对 ContivPlugin 的 gRPC 客户端进行初始化
createTenantNetwork	新建一个租户网络
deleteTenantNetwork	删除租户网络
addSubnet	在租户网络下新建一个子网
deleteSubnet	删除租户网络下的一个子网
getPodNetwork	获取 Pod 所在的租户网络信息
configPodNetwork	配置 Pod 的网络信息
clearPodNetwork	删除 Pod 网络并释放对应资源
getPodStatus	获取 Pod 的网络状态信息

通过上面提供的接口可以向 network-management 模块发起 gRPC 请求完成租户网络、子网以及 Pod 网络的管理工作。gRPC 协议比传统的 HTTP 和 webservice 协议性能更高，因此 Contiv Plugin 提供的这些网络管理接口在请求并发量大的情况下也能提供稳定的网络管理能力。

3.5.4 Network-Management 模块设计

在前文已经介绍 ContivPlugin 会作为 gRPC 的 client 端发起网络配置请求，而处理请求的 server 端正是本节要介绍的 network-management 模块。该模块的设计也使用了插件化的思想，它部署在 Kubernetes 集群中各个节点之上，负责处理 Contiv Plugin 发起的请求，主要完成租户网络地申请与释放、子网地创建与删除、Pod 网络地配置等工作。如图 3-16 所示，network-management 启动流程主要包含以下几个步骤：

（1）启动时先读取 Contiv-VPP 的配置文件信息，获取 Contiv Network 节点的地址、认证信息等。

（2）对上一步读取的认证信息进行验证，验证失败直接结束，验证通过后对 Contiv-VPP 网络环境进行初始化。

（3）注册 network-management 作为 gRPC 服务端要提供的服务，这些服务将在网络管理过程中被 Contiv Plugin 模块调用。

（4）服务注册成功以后启动服务并监听端口，等待 gRPC 客户端的调用。

启动成功以后，network-management 模块就可以对租户网络和容器网络进行管理，作为 gRPC 的服务端，该模块中主要包含了 TenantNetworkServer 和 PodNetworkServer 两部分，其中 TenantNetworkServer 负责提供租户网络以及子网相关的服务，PodNetworkServer 负责提供 Pod 以及容器网络相关的服务。

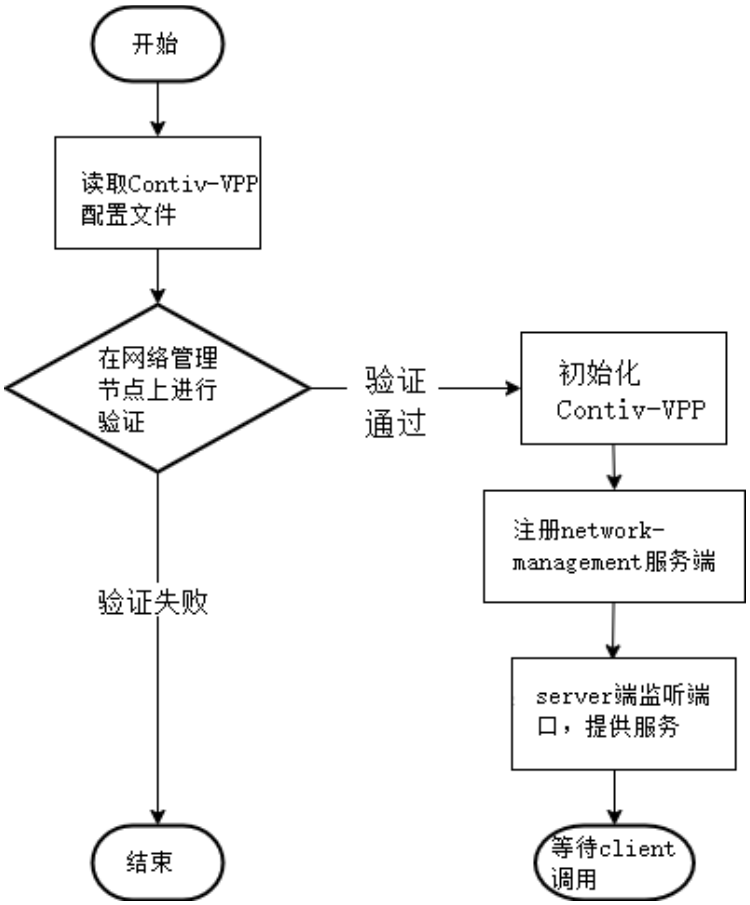

图 3-16 network-management 启动流程

TenantNetworkServer 提供对租户网络管理的能力，其提供的服务调用接口如表 3-7 所示，从表中可以看出它提供的服务主要涉及租户网络和租户子网两个方面。PodNetworkServer 提供了对 Pod 以及容器网络的管理能力，其提供的服务接口如表 3-8 所示。

表 3-7 TenantNetworkServer 提供的服务接口

接口方法	接口描述
createNetwork	新建租户时为该租户创建一个 Network 资源
deleteNetwork	删除租户的 Network 资源
getNetworkInfo	获取网络信息，如 IP 地址、网关等
getNetworkStatus	获取网络状态，以便检查网络是否可用
addSubnet	在租户网络下新建一个子网
deleteSubnet	删除租户网络下的一个子网
showAllSubnet	获取租户下所有的子网
connectSubnet	连接租户下的两个不同子网

表 3-8 PodNetworkServer 提供的服务接口

接口方法	接口描述
setPodNetwork	为租户下的 Pod 配置网络
getPodNetworkInfo	获取 Pod 的网络状态信息，如 IP、网关等
cleanPodNetwork	删除 Pod 网络并释放对应的网络资源

3.5.5 Network-Cli 模块设计

Network-Cli 是一个命令行工具，通过自定义命令独立调用 Network-management 模块提供的网络管理服务，完成 Contiv-VPP 中的 Network 资源以及 Subnet 资源的管理工作。本模块作为 gRPC 框架的客户端，Network-management 作为服务端，通过 RPC 调用实现网络管理，提供的命令如表 3-9 所示。

表 3-9 Network-Cli 模块设计的命令

命令	功能
network-cli create-tenant-network	创建租户的 Network 资源
network -cli delete-tenant-network	删除租户的 Network 资源
network -cli get-tenant-network-info	获取租户 Network 信息
network -cli update-tenant-network	修改租户的 Network
network -cli create-subnet	在租户下创建子网
network -cli delete-subnet	删除租户下的子网
network -cli get-subnet-info	获取子网信息
network -cli update-subnet	修改子网

3.6 本章小结

本章介绍了 Kubernetes 强多租户模型的设计需求以及各模块的总体设计。在访问控制模块介绍了 Keystone 认证服务组件，并设计出了租户 API、租户管理模型以及访问控制方案。在计算资源隔离模块分析了 Kata 安全容器引擎，提出了可信容器和非可信容器使用不同容器运行时的方案。在多租户网络模块设计了 Contiv Plugin、Network-management 插件以及 Network-cli 命令行工具实现了 Contiv-VPP 对 Kubernetes 网络的管理以及多租户之间的网络隔离。本章对各个模块的设计为第四章强多租户模型的实现打下了坚实的基础。

第四章 Kubernetes 强多租户模型的实现

4.1 访问控制

4.1.1 访问控制流程

用户请求资源时访问控制的流程如图 4-1 所示，用户在客户端请求访问 Kubernetes 集群中的某个资源。Kubernetes API Server 收到请求并检查请求的头部信息，包括认证令牌和请求者的身份信息。如果请求头部包含了 Keystone 认证令牌，API Server 将令牌发送给 Keystone 服务进行验证。Keystone 服务接收到 API Server 发送的令牌后，会验证令牌的有效性和访问权限。如果令牌验证通过，Keystone 服务将请求者的身份信息返回给 API Server。API Server 通过比较请求者的身份信息和 Kubernetes 集群访问控制策略来识别哪些资源允许访问以及哪些资源不能访问。如果请求者具有访问权限，则 API Server 将请求转发到相应的 Kubernetes 组件。Kubernetes 组件处理请求并将响应返回给 API Server。API Server 将响应返回给客户端。

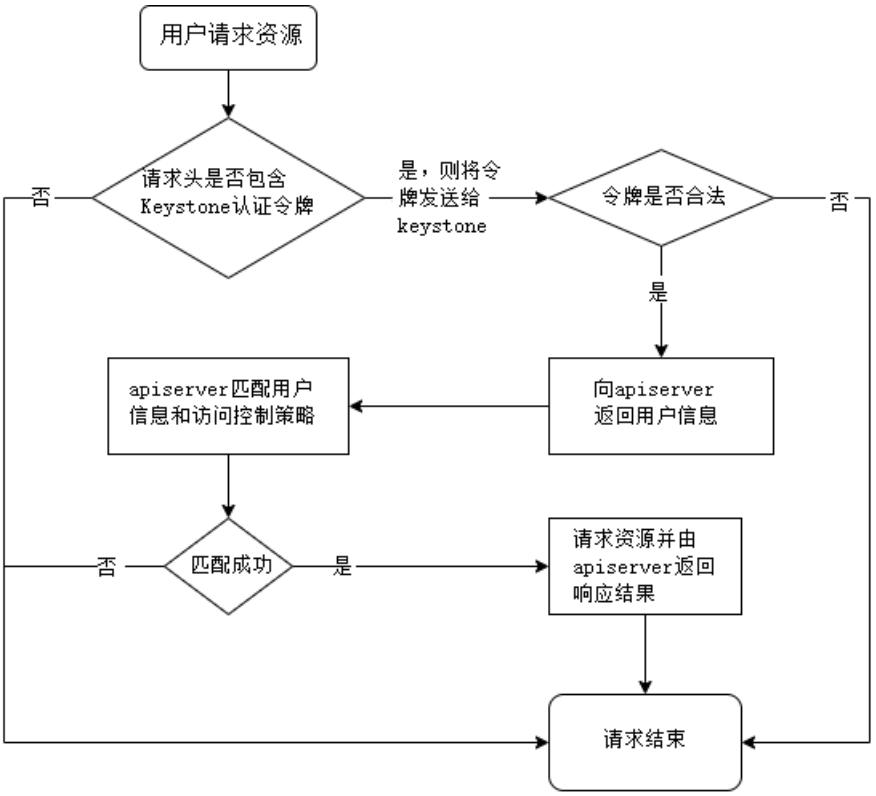

图 4-1 访问控制流程

4.1.2 认证代理组件实现

k8s-keystone-auth 是一个 Kubernetes 插件，主要用于在 Kubernetes 中集成 KeyStone，为 Kubernetes 集群提供统一的认证和授权服务。它的作用是将 Keystone 作为 Kubernetes 集群的用户认证源，同时允许 Keystone 通过 Webhook 模式来调用 Kubernetes API Server 的认证模块，实现 Kubernetes API 的安全认证和访问控制。具体来说，k8s-keystone-auth 会在 Kubernetes API Server 的请求处理流程中拦截 API 请求，然后调用 Keystone 的认证服务进行用户认证，并使用返回的认证令牌进行鉴权，以确保请求方具有所请求的资源的访问权限。

通过使用 k8s-keystone-auth 组件，Kubernetes 可以方便地将已有的 Keystone 用户集成到 Kubernetes 中，为 Kubernetes 强多租户环境下的用户管理提供了更多的灵活性。集成 Keystone 的步骤分为如下几步：

（1）新建 keystone-auth.yaml 文件，用来创建 ServiceAcccount、ClusterRole 和 ClusterRoleBinding，其中 ServiceAccount 用来对 Keystone 进行授权，以便 Keystone 可以访问 Kubernetes API，而 Cluster 和 ClusterRoleBinding 用来定义权限和管理 ServiceAccount。

（2）新建 keystone-config.yaml 文件，创建 ConfigMap 对象，用来指定 Keystone 的配置信息。该文件主要配置 Keystone 的数据库连接，数据库驱动、用户名和密码等信息。另外还指定了 Token 的存储和驱动方式。

（3）创建 keystone-deployment.yaml 文件，用来部署 Keystone 应用，文件关键内容如表 4-1 所示，该 Deployment 使用了 openstackhelm/keystone:4.0.4 镜像，并且定义了一个容器，容器中定义了两个端口，分别是 5000 和 35357。

表 4-1 keystone-deployment.yaml 文件关键内容

```
spec:
    containers:
    - name: keystone
      image: openstackhelm/keystone:4.0.4
      ports:
        - containerPort: 5000
        - containerPort: 35357
```

（4）创建 keystone-service.yaml 文件，文件关键内容如表 4-2 所示，该文件用来暴露 Keystone 容器的服务，该文件定义了一个 Service 对象，它将端口 5000 映射到容器的 5000 端口，并将端口 35357 映射到容器的 35357 端口。

表 4-2 keystone-service.yaml 文件关键内容

```
spec:
  ports:
  - name: http
    port: 5000
    targetPort: 5000
  - name: admin
    port: 35357
    targetPort: 35357
```

以上步骤完成后，Keystone 就已经集成到 Kubernetes 中了，可以通过 Service 的 IP 和端口访问 Keystone 服务。同时，RBAC-Controller 也会通过 keystone-auth ServiceAccount 获取 Keystone 的认证信息，以便在 Kubernetes 中进行授权。

4.1.3 用户认证实现

在 Kubernetes 中使用 Keystone 的认证服务需要借助 Kubernetes 提供的 Webhook 认证模式。Webhook 是一种 HTTP 回调，在某些条件下会触发 HTTP 的 POST 请求从而发送事件通知。具体来说，Kubernetes 在进行用户认证时会使用 Webhook 模式来调用 Keystone 提供的身份验证服务，从而实现对非法用户的检测与拦截。在 Kubernetes 集群中使用 Keystone 来进行身份验证需要满足两点要求：

（1）Kube-apiserver 能够向 Keystone 发起用户认证的请求。

（2）Kube-apiserver 和 Keystone 之间能够相互解析对方发送的用户信息。

要满足上述两点要求，实现 Kubernetes 和 Keystone 认证组件的集成，需要配置 kubectl 证书以及用户环境变量，配置成功以后 Keystone 将会全权负责 Kubernetes 集群的用户认证和管理工作，在认证用户时 Kubectl 就会先向 Keystone 发起请求，获取用户的 Token 信息，然后再将 Token 和用户信息一起发送给 kube-apiserver 完成请求。另外，kube-apiserver 和 Keystone 是两种不同的组件，使用的数据格式不同，在使用 WebHook 模式验证 Token 信息时，需要将 Token 信息转换成它们能够识别的格式，k8s-keystone-auth 插件就是 kube-apiserver 和 Keystone 之间的桥梁。K8s-keystone-auth 组件负责将 Kubernetes 的 API 请求转换为 Keystone 格式的请求并转发给 Keystone，同时还需要将 Keystone 格式的认证结果转换成 kube-apiserver 能够识别的格式并传给 kube-apiserver。具体的认证过程如图 4-2 所示：

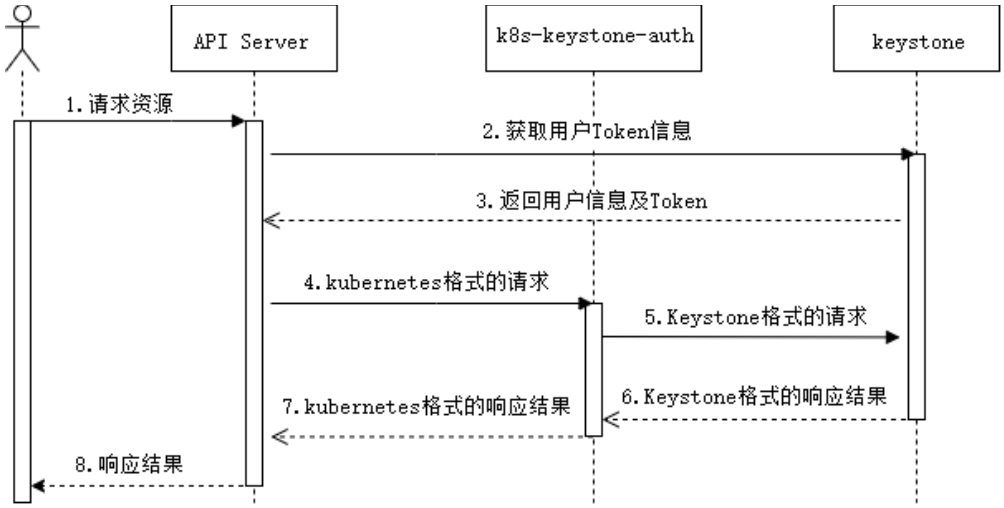

图 4-2 用户认证流程时序图

4.1.4 用户授权实现

本小节将在使用 Kubernetes 的控制器设计模式在原生的 RBAC 授权模式的基础上设计一个 RBAC-Controller，以便完成用户到角色地绑定。该模块的类图如图 4-3 所示，其中 Role 类和 ClusterRole 类表示 Kubernetes 的角色，只不过它们的作用域不同，Role 只能作用于某一个 Namespace 中，而 ClusterRole 作用于整个集群，它们都包含具有名称和权限等属性，可以为角色添加或删除权限规则；RoleBinding 类和 ClusterRoleBinding 类都是角色绑定类，分别用于 Role 和 ClusterRole 的绑定工作中。RBACController 类是该模块的核心，用于管理 Kubernetes 角色、角色绑定、集群级别的角色和角色绑定等，方法包括创建、修改和删除角色、角色绑定、集群级别的角色和角色绑定等，以及查询用户是否具有特定的权限。

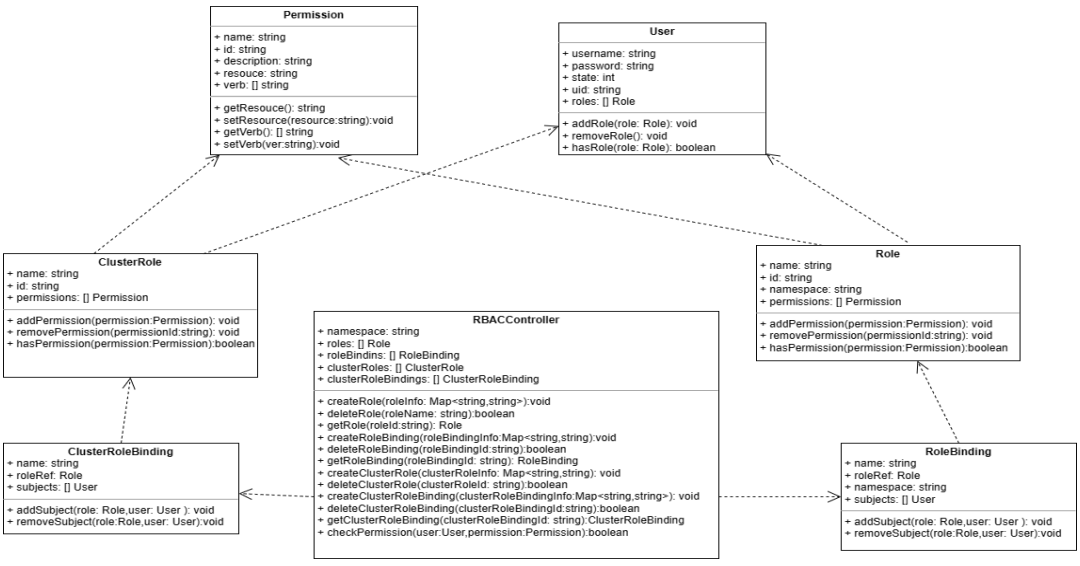

图 4-3 用户授权模块类图

通过 3.3.4 小节的设计可知，不同的用户能够访问的资源是不同的，它们只能访问权限内的资源。租户管理员在租户内拥有 admin 权限，租户下的各个资源都可以操作；而普通用户不具备这样的权限，只有租户管理员为其分配权限后它才能对该租户下的部分资源进行访问。系统管理员创建租户资源时会创建一个租户管理员，让其对租户下的资源进行管理。租户创建完成之后就要在 Kubernetes 集群中为该租户创建 Namespace 资源对象，以便实现租户之间的隔离。因此需要一种监听机制，一旦发现有租户创建成功就立刻为该租户创建相应的 Namespace 资源。Kubernetes 提供了 ListWatch API 来监听指定的资源和事件。ListWatch API 会返回一个实时更新的数据流，可以根据指定的资源和事件过滤数据流中的事件并执行相应的操作。在 RBAC-Controller 的配置文件中，可以通过指定 resources 字段和 verbs 字段来设置监听的资源和事件。

如表 4-3 所示的配置中，RBAC-Controller 会监听 pods、services、deployments 和 replicasets 资源的 create、update 和 delete 事件。当监听到事件发生时，RBAC-Controller 会根据配置的 RBAC 规则对请求进行鉴权，并根据鉴权结果决定是否允许请求的执行。

表 4-3 鉴权规则配置示例

```
rules:
-   apiGroups: ["","extensions","apps"]
    resources: ["pods","services","deployments","replicasets"]
    verbs: ["create","update","delete"]
```

RBAC-Controller 会在监听到 Namespace 的新增事件时创建一对 Role 和 RoleBinding 对象，用来赋予租户管理员在该租户下的 admin 权限。租户管理员角色绑定成功后就可以为普通用户分配相应的权限，实现用户的自动授权与鉴权。RBAC-Controller 的处理流程如图 4-4 所示。

（1）RBAC 控制器通过 Kubernetes 提供的 ListWatch 机制监听 Keystone 中创建角色的请求。

（2）RBAC 控制器监听到创建角色事件后会解析事件的内容，并对事件内容进行转换，转换成 RBAC 控制器能够识别的格式。

（3）RBAC 控制器向 kube-apiserver 发起创建 Role 或者 ClusterRole 资源对象的 API 请求。

（4）RBAC 控制器通过 ListWatch 机制监听 Keystone 中的用户授权操作。

（5）RBAC 控制器向 kube-apiserver 发起创建 RoleBinding 和 ClusterRoleBinding 资源对象的请求，完成用户和角色的绑定，从而实现用户的授权。

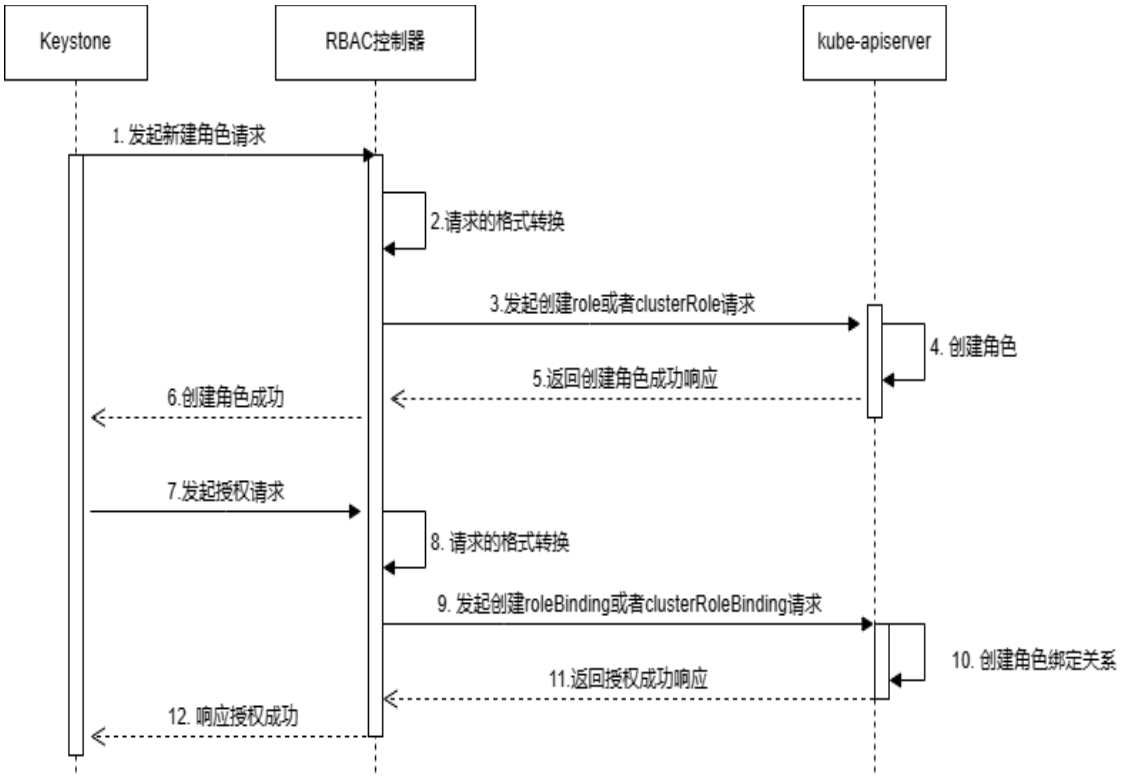

图 4-4 用户授权时序图

上诉流程是 RBAC-Controller 在监听到新建角色后的工作流程。另外，RBAC-Controller 还要负责监听集群中 RBAC 规则的更新，当发现规则发生变化时就需要创建出新的 role 和 rolebinding 对象，然后还要在 Keystone 中更新相应的角色和绑定关系。本文设计的 RBAC-Controller 能够将 Kubernetes 原生 RBAC 策略和多租户权限控制模块无缝衔接，共同构成了 Kubernetes 集群统一的权限控制系统。

4.2 计算资源隔离的实现

为了实现 Kubernetes 强多租户模型的计算资源隔离，需要保证容器运行应用程序时互不影响，在本文的解决方案中将使用 Kata 安全容器来实现 Pod 在独立内核上运行，保证租户的计算资源隔离。Kata 容器通过 Hypervisor 虚拟化技术实现容器安全，作为 Kubernetes 的容器运行时，Kata 会为每个 Pod 都创建一个独立的轻量级运行环境，每个运行环境中都有独立的内核，从而解决了容器内应用逃逸到内核后互相影响的问题。从表 4-4 可以看出，Kata 容器在多租户环境以及云原生应用程序中能够提供比 Docker 更高的安全性和隔离性，使用 Kata 容器运行时能够解决了内核共享造成的问题。Kata 容器经过多年的发展已经可以和现有的容器生态完美对接，能够在 Kubernetes 中安全运行并提供虚拟机级别的安全性和隔离性。

表 4-4 Kata 和 Docker 的比较

	Docker	Kata
设计目标	轻量级、可移植、容易部署	安全性、隔离性
实现方式	基于操作系统的虚拟化技术	Hypervisor 虚拟化技术
使用场景	微服务、容器化应用程序、DevOps 等	敏感数据的应用程序、多租户环境、云原生应用程序等

Kubernetes 接入 Kata 容器的方式如图 4-5 所示，Kata 容器包含 runtime、proxy、shim 和 agent 四部分，其中 runtime 只有一个，shim、proxy 和 agent 则是每个容器独立拥有的。Kubernetes 不为 Kata 提供容器运行时环境，不和 Kata 直接通信，它只提供 CRI（容器运行时接口），Kata 只跟实现了 CRI 的容器管理进程通信。在 Kubernetes 容器云平台上创建容器时会先向 Kube-apiserver 发送创建请求，收到请求后通过容器运行接口向 CRI-containerd 发出创建容器的指令。Kata-proxy 提供了一个强大的 gRPC 服务，可以让 Kata-shim-v2 调用，并且通过 Hypervisor 技术创建一个虚拟机，每个虚拟机都包含一个最小镜像，最后由 guest kernel 来启动虚拟机。Kata-proxy 在虚拟机启动后会向 Kata-agent 发出指令创建一个沙盒环境并且根据 OCI 协议配置 config.json 文件，以确保虚拟机能够在一个拥有自主内核的轻量级环境中运行。

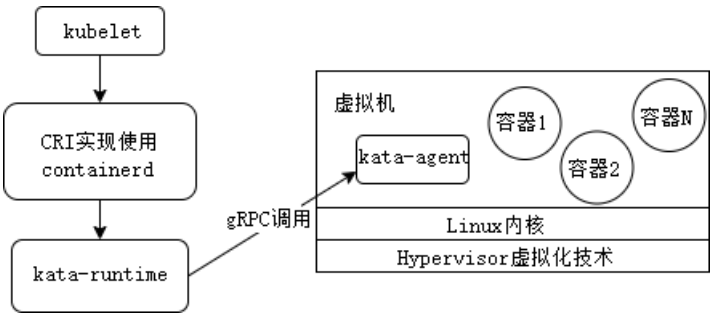

图 4-5 kubernetes 接入 Kata 安全容器

Kubernetes 底层 CRI 默认实现是 dockershim，在本文介绍的方案中要使用 Kata 容器，因此首先要在 Kubernetes 集群初始化时将 dockershim 替换成 containerd，然后集成 kata 和 contained，具体过程包含如下步骤：

（1）部署 Containerd：Containerd 是符合 CRI 标准的容器运行时，用于管理容器的生命周期。它用于提供更轻量化的容器运行时，以支持 Kubernetes 等容器编排工具。

（2）部署 Kata：部署 Kata 前需要开启 Hypervisor 功能以及嵌套虚拟化， Kata 会为每个 Pod 创建独立的虚拟机，创建虚拟机需要使用到 Hypervisor 技术，所以集群中每个节点都要开启虚拟化功能，另外还需要开启嵌套虚拟化功能。

（3）开启之后就可以安装 Kata，Kata 安装完成以后需要设置 containerd 使用的 Kata 容器运行时，默认使用的是 runc，所以需要在 config.toml 配置文件中进行修改。主要修改内容如表 4-5 所示。

表 4-5 配置 containerd

Config.toml 文件主要修改内容
version=2 root = "/var/lib/containerd" state = "/run/containerd" plugin_dir = "" disabled_plugins = [] required_plugins = [] oom_score = 0

（4）部署 kubernetes 集群：上述步骤执行完毕后就可以通过 kubeadm 工具部署以 contained 为 CRI 实现的集群了。

当上述步骤执行完以后就可以创建 Pod，具体的创建流程如图 4-6 所示。Kubernetes 集群在创建 Pod 等资源时如果是非可信容器则在 Kata 运行，利用 Kata-runtime 创建轻量级虚拟机，保证容器运行时安全，避免容器内应用逃逸到内核影响其他容器。

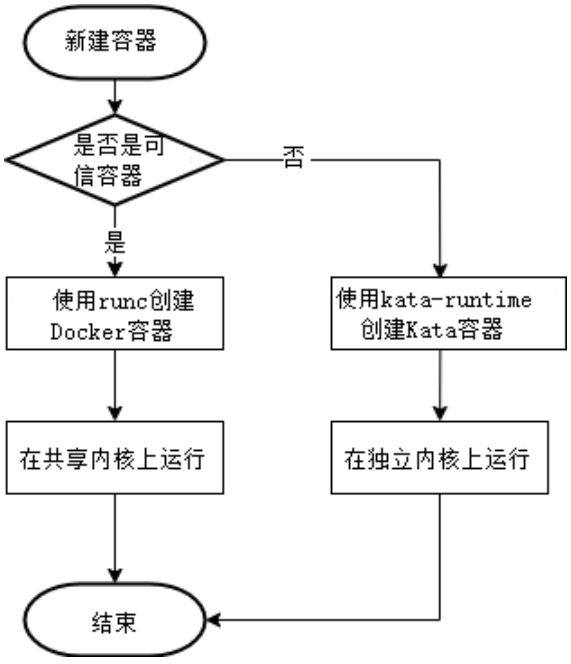

图 4-6 容器隔离流程

4.3 租户网络

4.3.1 多租户网络的实现

通过前文介绍可以知道，Contiv-VPP 可以提供高度可定制的网络功能，包括多租户网络隔离、可编程网络策略、网络服务链、弹性负载均衡等，适用于云计算和容器化环境。在本文的网络资源隔离方案中，通过 Contiv-VPP 构建租户网络，实现了网络隔离，并且通过自定义网络策略实现了流量控制等功能。

因此，首先要在集群中部署 Contiv-VPP 插件，在 Kubernetes 集群中部署 Contiv-VPP 网络插件可以按照如下步骤进行：

（1）准备环境，确保每个节点都满足以下要求：Linux 内核版本 3.16 或更高；Kubernetes 版本 1.15 或更高。

（2）部署 ETCD 集群，Contiv-VPP 使用 ETCD 作为配置数据库，配置三个节点的 ETCD 集群，确保每个 ETCD 节点之间都能互相通信。

（3）部署 Contiv-VPP 插件：通过 wget 指令下载 Contiv-VPP 的编排文件 contiv-vpp.yaml，然后使用 kubectl apply -f contiv-vpp.yaml 指令部署 Contiv-VPP。

（4）配置 Contiv-VPP：配置 Contiv-VPP 的运行参数，例如：CPU 核数、内存大小等，可以通过修改 Contiv-VPP DaemonSet 的配置文件进行修改。还可以通过修改 Contiv Agent 的配置文件修改使用的网络策略。

多租户的网络拓扑关系如图 4-7 所示，在本文方案中将为租户构建一个虚拟网络，一个租户对应一个网络资源，租户和网络是一对一的关系。

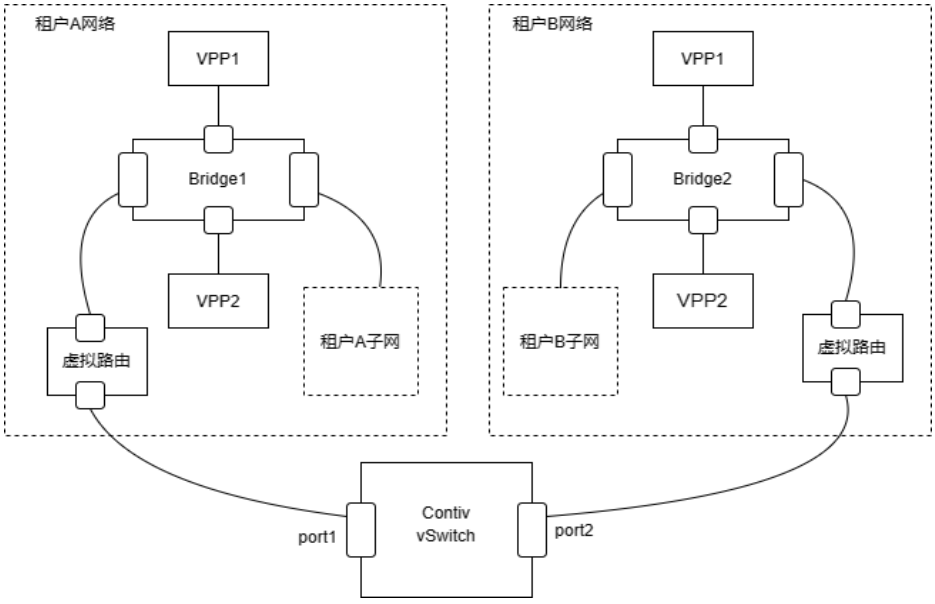

图 4-7 多租户网络拓扑关系

构建租户网络时会通过 Contiv-VPP 来为该租户对应的 Namespace 对象创建一个 VPP 实例，每个 VPP 实例都有独立的虚拟路由、交换机和防火墙等网络设备。VPP 实例可以为 Namespace 内的子网提供与外部通信的能力，并且可以通过虚拟交换机将该 Pod 配置到租户网络中，同时还可以通过配置各个 VPP 实例之间的路由规则和防火墙规则确保每个 Namespace 中的容器之间的网络隔离，而不会受到其它 Namespace 中容器的影响。

网络控制器会监听网络的新建、删除以及更新事件，其中新建租户网络的时序图如图 4-8 所示，具体有以下几个步骤：

（1）创建 NetworkAttachmentDefinition 对象：使用 kubectl 或 yaml 文件创建 NetworkAttachmentDefinition 对象，它指定了容器需要使用的网络配置信息，例如网络类型、IP 地址、网关等

（2）创建 VPP 实例：为每个 NetworkAttachmentDefinition 对象创建一个 VPP 实例，VPP 实例将提供网络隔离和转发功能。可以使用 Contiv-VPP 提供的 CLI 或 API 创建 VPP 实例。

（3）配置 VPP 实例：包括添加接口、路由、ACL 规则，确保网络的正确配置。

（4）将 VPP 实例绑定到租户对应的 Namespace 对象上：使用 Contiv-VPP 提供的 CLI 或 API 将 VPP 实例绑定到指定的 Namespace 上，确保该 Namespace 中的容器可以访问该 VPP 实例提供的网络资源。

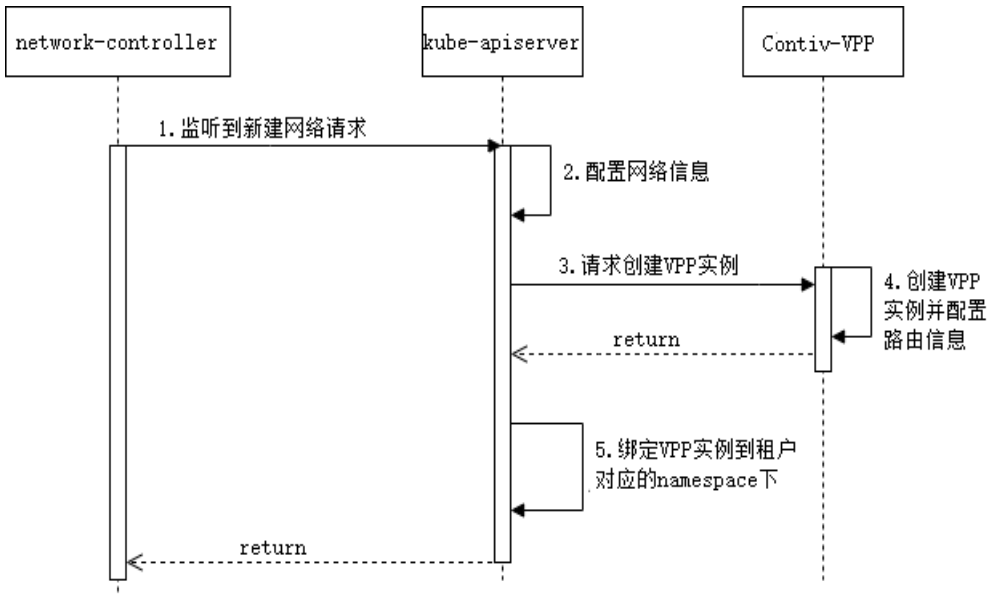

图 4-8 新建租户网络时序图

租户网络创建成功以后，就可以将租户下 Pod 加入到自己的网络并为其分配响应的网络资源，删除 Pod 时则需要清理对应的资源，使用 CNI 插件将 Pod 加入

到对应的租户网络并在删除 Pod 时释放相应的资源，Pod 加入租户网络以及删除 Pod 时释放资源的过程如图 4-9 所示。

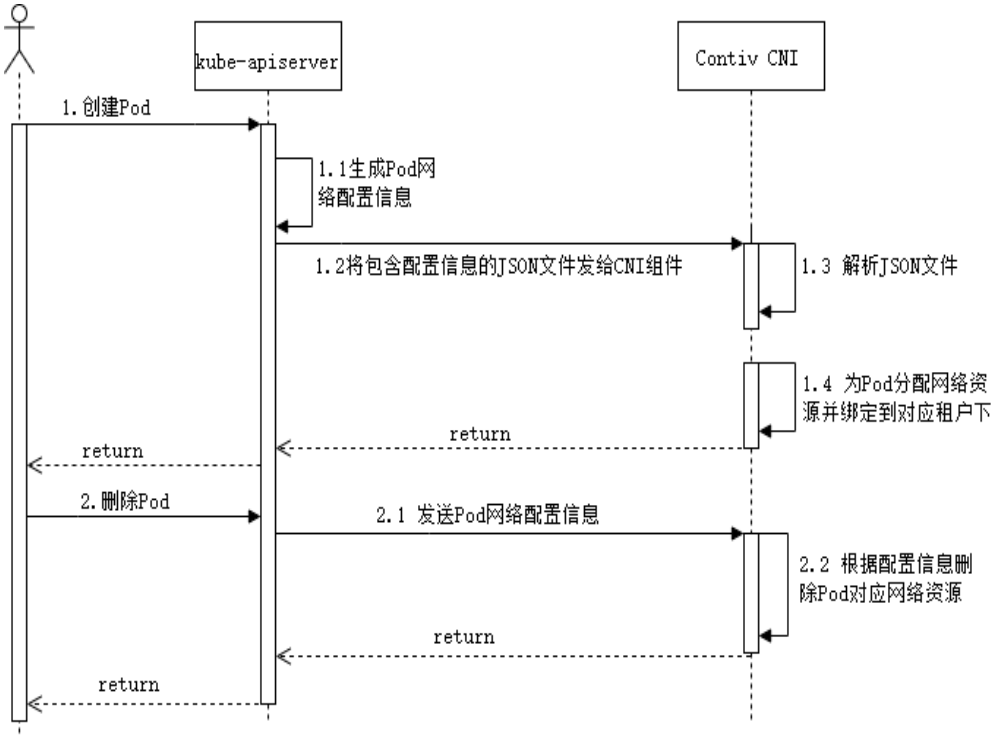

图 4-9 Pod 创建以及删除 Pod 的时序图

（1）当 Pod 创建时，Kubernetes 会调用 Contiv-VPP CNI 插件，传递一个包含 Pod 网络配置信息的 JSON 文件。

（2）Contiv-VPP CNI 插件会解析该 JSON 文件，然后创建一个 Linux Bridge，并将该 Linux Bridge 与 Pod 所在的网络 Namespace 绑定。

（3）Contiv-VPP CNI 插件还会为 Pod 配置一个 IP 地址、子网掩码等网络参数。

（4）当 Pod 删除时，Kubernetes 会再次调用 Contiv-VPP CNI 插件，传递一个包含 Pod 网络配置信息的 JSON 文件。

（5）Contiv-VPP CNI 插件会根据该 JSON 文件中的信息，删除之前创建的 Linux Bridge 和相关配置信息。

当 Pod 加入到租户网络后还需要对外暴露服务，Kubernetes 提供了多种对外暴露服务的方式，比如以下几种方式：

ClusterIP：默认的服务类型，只在集群内部可用，为 Service 分配一个虚拟 IP 地址，内部 Pod 可以通过该地址访问 Service。

NodePort：通过将节点上的端口映射到 Service 端口，可以实现外部访问者通过节点的 IP 地址和端口号来访问服务。

LoadBalancer：在云服务提供商中使用，自动创建一个外部负载均衡器，并将服务公开到外部 IP 地址和端口上。

Ingress：在 Service 之上引入一层代理，它可以根据 Ingress 资源的规则配置负载均衡器，从而将外部的流量路由到不同的服务上。

ClusterIP 和 NodePort 这两种方式都是通过 IP 和端口号来实现服务暴露的，但是在强多租户模型中一个节点上会存在多个租户资源，租户下可能存在多个相同的虚拟 IP 地址，所以通过 IP 加上端口号的方式来暴露服务在多租户模型中不可行。Contiv-VPP 作为一款优秀的网络代理插件，通过 LoadBalancer 负载均衡服务提供了对外暴露服务的能力。Contiv-VPP 对外暴露服务的步骤如图 4-10 所示，具体可以分为以下几个步骤：

（1）创建一个 Service 资源，指定 service type 为 LoadBalancer，该服务将被分配一个 Cluster IP 和一个外部 IP。

（2）当 Service 资源被创建时，Contiv-VPP 将创建一个 loadbalancer VPP 插件实例，并将其绑定到 Service 的 Cluster IP 上。

（3）Service 资源中的每个 Pod 的 IP 地址都将被添加到 Contiv-VPP 的 ACL 中，以允许服务的入站和出站流量。

（4）当外部客户端请求服务时，请求将被路由到 Service 的外部 IP 地址，然后通过 Contiv-VPP 的 loadbalancer 插件实例路由到相应的 Pod。

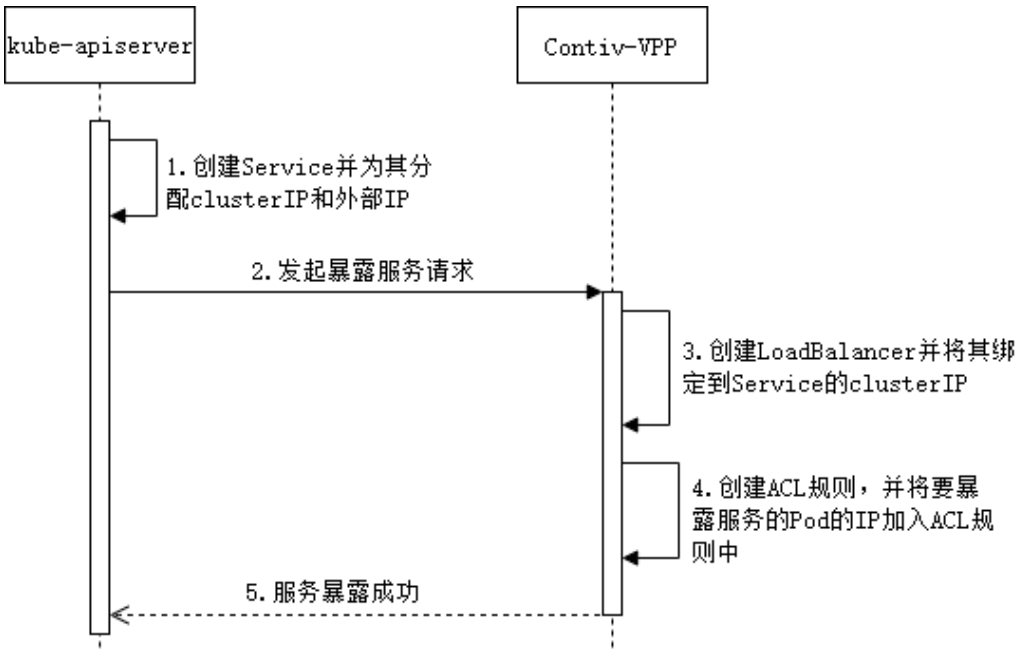

图 4-10 Contiv-VPP 对外暴露服务时序图

Contiv-VPP 通过 LoadBalancer 插件对外暴露服务并提供了负载均衡功能，减轻了 Pod 的单点压力，但是当用户请求量过大超过集群的负载能力时仍然可能造成节点宕机，因此可以对 Pod 进行流量控制，Contiv-VPP 提供了流量控制功能。

Contiv-VPP 的流量控制是基于 NetQos 的，可以通过 NetQos 来实现限制容器的带宽和容器的最大速率。NetQos 是 Linux 内核的一个子系统，用于控制网络流量的速率和带宽，可以实现流量限制、带宽控制、流量分类等功能。

要实现流量控制，首先需要创建一个包含限制规则的网络策略文件。网络策略文件定义了入口和出口的流量限制规则，以及服务的负载均衡规则。在网络策略文件中，可以通过指定标签选择要应用策略的 Pod 或 Service，然后在 Pod 或 Service 的标签中添加与策略文件匹配的标签，以便将策略文件应用到相应的 Pod 或 Service 上。这样，Contiv-VPP 就可以通过网络策略文件来限制 Pod 的流量。流量控制主要包含以下几个步骤：

（1）创建一个网络策略，限制入口和出口流量，如表 4-6 所示的内容定义了一个名为“testpolicy”的网络策略，作用于标签为“app=testapp”的 Pod。该策略包含了入口和出口的流量限制规则，允许来自 192.168.1.0/24 CIDR 块的 IP 地址的流量进入 Pod。

表 4-6 网络策略配置

```
kind: NetworkPolicy
metadata:
  name: testpolicy
spec:
  podSelector:
    matchLabels:
      app: testapp
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.1.0/24
```

（2）创建一个 Contiv-VPP policy profile，指定需要限制的网络流量规则，关键内容如表 4-7 所示，该文件创建了两个规则，一个是限制从 192.168.1.0/24 子网的 IP 地址发送到 Pod 的 80-81 端口的 TCP 流量，最大速率是 10KB/s，突发流量大小是 1KB；另一个规则是限制从 Pod 发送到 10.10.0.0/16 子网的 IP 地址的 80-81 端口的 TCP 流量，最大速率是 5KB/s，突发流量大小是 500 字节。

（3）将 policy profile 应用到需要限流的 Pod 上，通过命令“kubectl label pods testpod contiv-profile=test_policy_profile”命令将名字是“testpod”的 Pod 和 test_policy_profile 绑定到一起。

表 4-7 网络流量规则创建方式

```
"source_net": "192.168.1.100/24",
"port_range": "80-81",
"rate_limiter": {
        "avg_bps": 10000,
        "burst_bytes": 1000
  }
"dest_net": "10.10.0.0/16",
"port_range": "80-81",
"rate_limiter": {
        "avg_bps": 5000,
        "burst_bytes": 500
  }
```

完成上述步骤后，Contiv-VPP 就会自动将网络策略应用到相关的 Pod 上面，从而实现流量控制功能。

4.3.2 Contiv Plugin 模块的实现

从前文介绍可知，本文设计的 Contiv Plugin 使用了 RPC 远程调用协议，它作为 client 端向作为 server 端的 network-management 发起网络配置请求。本小节将介绍 Contiv Plugin 的具体实现细节，Contiv Plugin 模块的类图如图 4-11 所示。

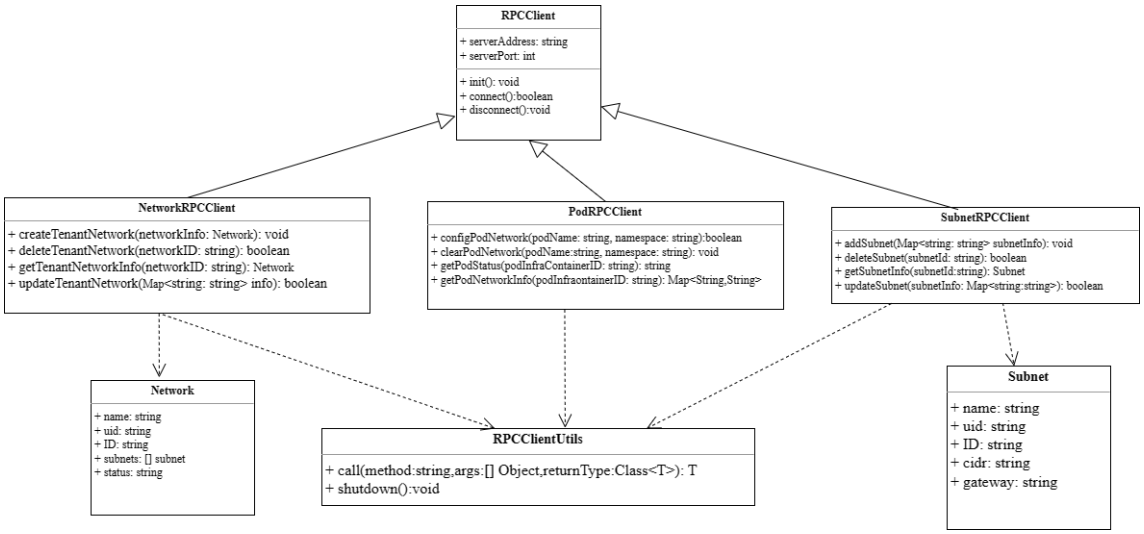

图 4-11 Contiv Plugin 模块类图

其中 Network 类是对网络资源的封装，它包括网络 ID、子网信息等；Subnet 类是对子网的封装，它包含子网 ID，所属 Network 的 ID、CIDR 以及网关等信息。

NetworkRPCClient 是负责租户网络管理的入口，它主要包含租户网络的创建、删除、修改、获取网络信息等接口。Network 类是对网络资源的封装，它包含网络 ID、子网信息等。PodRPCClient 是 Pod 和容器网络配置的入口，通过 configPodNetwork 接口完成 Pod 网络的配置，另外它还提供了清理 Pod 网络资源、查询 Pod 内容器网络信息的接口。SubnetRPCClient 负责在租户网络下子网的管理，通过 addSubnet 接口可以为租户创建新的子网，另外它还提供了移除子网、获取子网信息等接口。

Contiv Plugin 作为 Client 端要调用 Server 端的服务，因此 Kubelet 在启动时就要为 Contiv Plugin 配置 Server 端的 IP 地址，然后还要改进 Kubernetes 插件加载流程，加载插件时通过启动参数判断是否使用 Contiv-VPP 网络管理插件，如果要使用那么要对 Contiv Plugin 进行初始化并加入到 Kubernetes 插件数组中。Contiv Plugin 启动参数配置完成以后就要开始初始化工程，Contiv Plugin 初始化过程如图 4-12 所示，从图中可以看出初始化过程主要完成两个工作：和 Server 端建立网络连接以及初始化 gRPC Client 程序。通过 Kubelet 启动时配置的 Server 端地址来发起建立连接请求，建立成功以后对 Client 端程序进行初始化，初始化完成后等待用户的调用，用户发起请求后调用 Server 端服务完成请求处理。

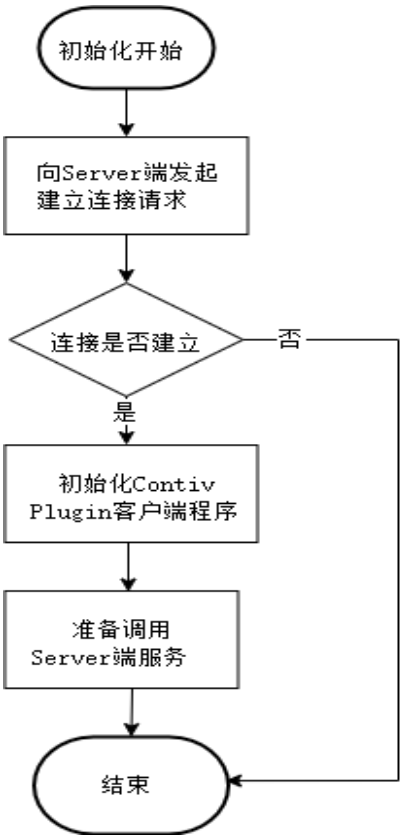

图 4-12 Contiv Plugin 初始化过程

Contiv Plugin 初始化完成后就可以对用户提供 Pod 网络网络管理接口，用户通过 Contiv Plugin 提供的接口实现对 Pod 网络的设置、删除等操作。

配置 Pod 网络功能是通过 configPodNetwork 接口来实现的，它的具体工作流程如图 4-13 所示，首先找到 Pod 所在的 Namespace，然后根据 Namespace 获取到对应的 Subnet，接着将 Subnet 的网络配置信息作为参数向 RPC Server 端发起 Pod 网络配置请求，最后由 network-management 服务端完成具体的配置工作。

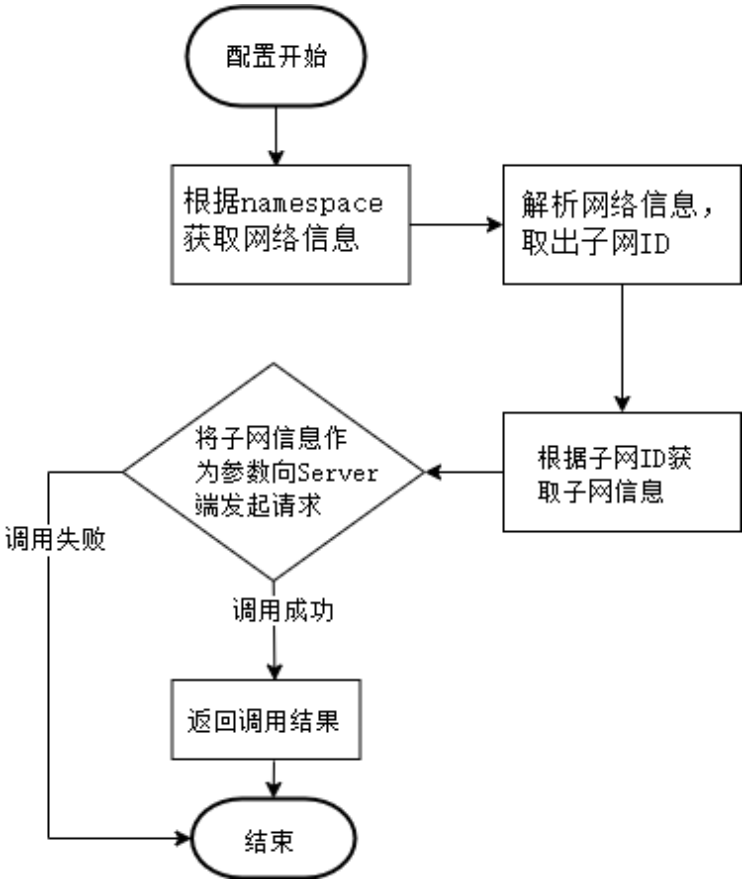

图 4-13 Contiv Plugin 配置 Pod 网络的流程

删除 Pod 网络并释放网络资源的功能是通过 clearPodNetwork 接口来完成的，工作流程如图 4-14 所示。删除 Pod 时首先根据 Namespace 找到 Pod 所在的租户网络信息，然后解析出租户网络中的 networkID 等信息，通过 networkID 可以找到 Contiv-VPP 中为该 Pod 分配的网络资源，最后调用 RPC Server 端的服务删除 Pod 并释放 Pod 绑定的资源。

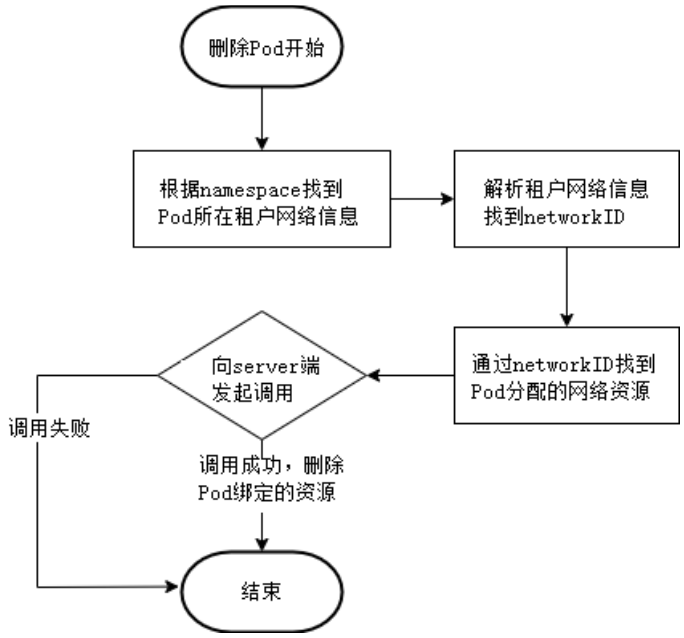

图 4-14 Contiv Plugin 删除 Pod 网络流程图

4.3.3 Network-management 模块的实现

上一小节中的 Contiv Plugin 模块作为 RPC 的客户端发起调用，本小节将介绍 gRPC 服务端即 Network-management 模块的具体实现。Network-management 模块的功能主要包含两个部分：租户网络资源的管理和 Pod 网络配置，该模块的类图如图 4-15 所示。

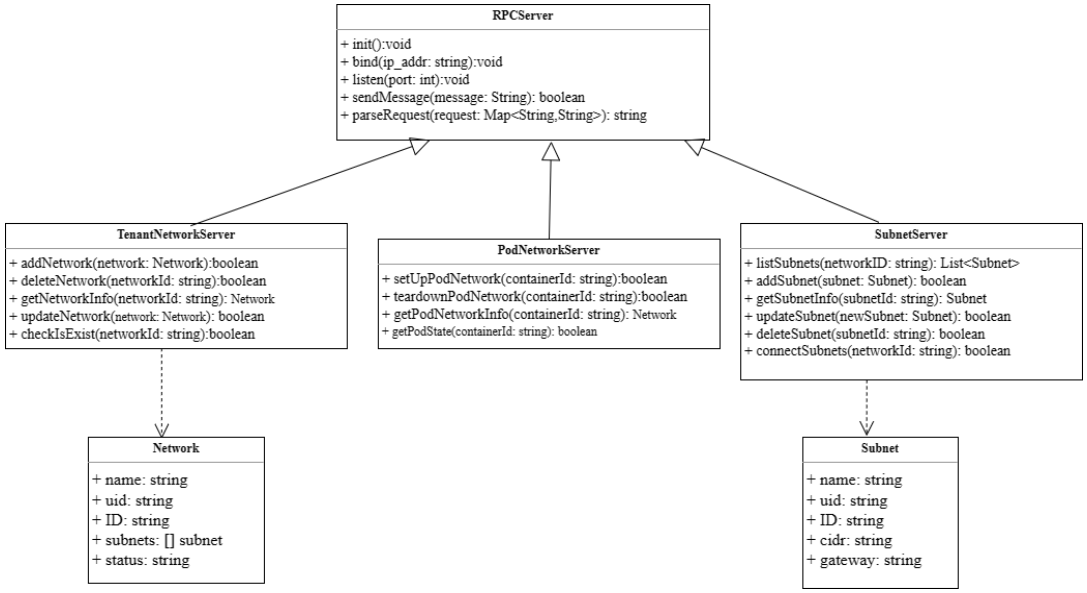

图 4-15 Network-management 模块的类图

其中租户网络资源管理由 TenantNetworkServer 来实现，它提供了 addNetwork 接口来为租户创建一个新的 Network 资源，当删除租户时利用 deleteNetwork 接口释放掉该租户对应的 Network 资源；子网管理由 SubnetServer 负责，主要包含创建子网、删除子网、查询租户网络下子网列表等接口；而 Pod 网络由 PodNetworkServer 管理，利用它提供的 setUpPodNetwork 接口可以将 Pod 配置到租户网络下面，另外还可以通过 teardownPodNetwork 接口删除 Pod 网络。

TenantNetworkServer 提供了对租户网络以及子网进行管理的接口，在 Kubernetes 中创建 Namespace 时为其绑定 Subnet 资源，删除时释放对应的资源。具体实现时，TeanantNetworkServer 会进一步调用 Contiv-VPP 的接口来完成网络的配置和管理。从图 4-16 可以看出，TenantNetworkServer 创建租户网络的过程主要分为以下几个步骤：

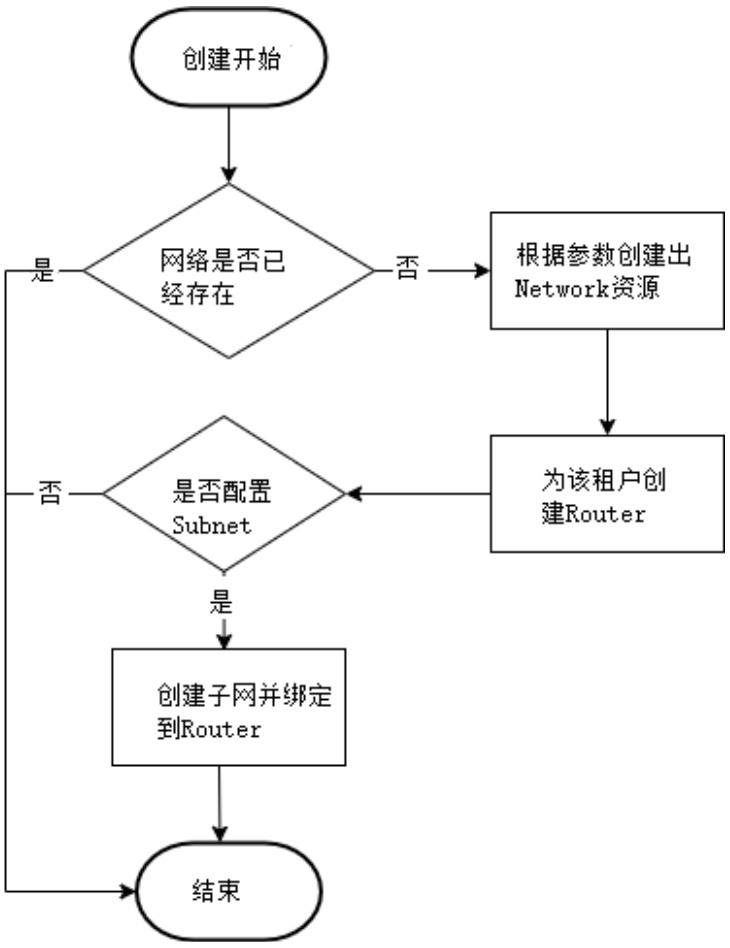

图 4-16 TenantNetworkServer 创建 Network 的流程

（1）判断该网络是否已经存在。根据参数中的 networkID 去调用 Contiv-VPP 的 API 接口，判断要创建的网络是否已经存在，如果已经存在则流程直接结束。

（2）创建网络。该网络不存在，则在 Contiv-VPP 中创建出该网络资源对象。

（3）创建 Contiv-VPP 虚拟路由。在本文的设计方案中，租户下面有多个 Namespace，它们对应着不同的子网，为了实现子网间的通信，需要在租户下创建一个 Router，它负责租户网络和子网之间的网络互通。

（4）创建子网。虚拟路由创建成功后需要根据调用参数判断是否需要创建子网，如果需要创建则调用 Contiv-VPP 的 API 接口完成子网的创建工作，否则流程直接结束。

完成上述步骤后，租户的网络资源就已经创建成功，接下来就可以在租户网络下创建子网，Subnet 创建过程如图 4-17 所示，可以分为以下几个步骤：

（1）判断子网是否已经存在。根据 subnetID 调用 Contiv-VPP 接口，判断要创建的子网是否已经存在，如果存在则调用结束。

（2）创建子网。根据调用参数中包含的信息在 Contiv-VPP 中创建 Subnet 资源对象。

（3）绑定到虚拟路由。租户下不同子网通信需要租户路由来实现，因此创建出 Subnet 资源后需要将其绑定到该租户的虚拟路由下，通过虚拟路由转发子网之间的网络数据。

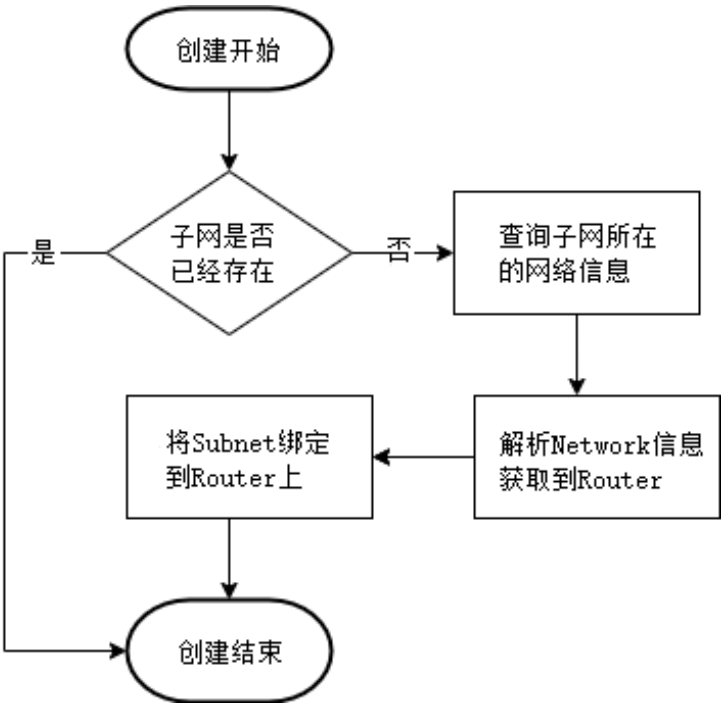

图 4-17 TenantNetworkServer 创建 Subnet 的流程

完成前面租户网络以及子网的创建后，Pod 网络配置的前置工作就已经全部完成了，接下来就可以对 Pod 进行网络配置，配置工作由 Network-management 模块

中的 PodNetworkServer 负责，配置过程如图 4-18 所示。从图中可以看出，Contiv Plugin 调用 PodNetworkServer 的接口请求配置 Pod 网络，PodNetworkServer 首先会为该 Pod 分配一个所在子网的一个 Port，这个 Port 用于连接容器网络和 OVS 网桥，实现容器与外界的通信，接着 PodNetworkServer 会调用 Contiv-VPP 的 API 接口，具体的网络配置由 Contiv-VPP 完成。为了将容器网络和 OVS 网桥连接到一起，Contiv-VPP 会创建一个 Veth-Pair 虚拟接口，Veth-Pair 有两个端口，可以用来连接两个虚拟设备实现网络互通，创建好以后就可以将 Pod 分配到的 Port 和 OVS 网桥通过 Veth-Pair 连接起来，Pod 网络配置完成。

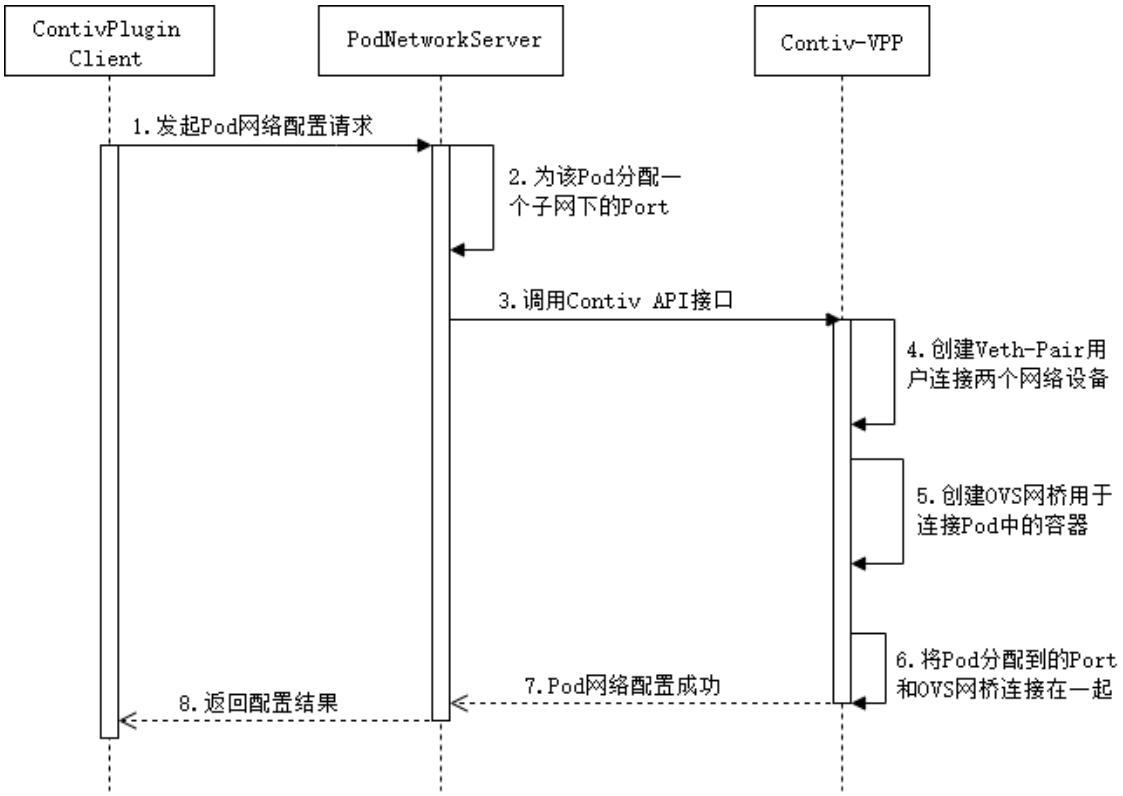

图 4-18 PodNetworkServer 配置 Pod 网络时序图

4.3.4 Network-Cli 模块的实现

该模块作为命令行工具向 Network-management 发起管理请求，为了实现自定义命令行，本文将使用 Go 语言，利用 Cobra 开源框架实现一套网络管理命令。Cobra 是一个用于创建命令行应用程序的开源 Go 语言框架，它提供了一组简单易用的 API，能够快速构建命令行应用程序，支持命令、子命令、参数和选项等常用命令行特性，Docker、Kubectl、Git 等工具的 CLI 工具都是使用 Cobra 框架构建的。

使用 Cobra 来实现本模块的自定义命令时，首先通过“cobra init network-cli”创建一个名字为“network-cli”的 Cobra 项目，然后在该项目下使用“cobra add”

命令创建出要自定义的命令，比如“cobra add network-cli create-tenant-network”，执行完该命令后会生成一个 go 文件，在 go 文件中就可以根据业务来编写具体的命令逻辑，接着可以注册命令并编译运行，最终会生成一个可执行文件。

通过上述方法就能够创建出自定义的命令，比如 network-cli create-tenant-network 命令会读取租户网络配置文件，根据文件中的内容来为租户创建出 Network 资源以及 Subnet 资源。如表 4-8 所示的配置文件，将会为租户 tenantA 创建一个 Network 资源，同时在该 Network 下创建一个子网 subnet1，并且还配置了 subnet1 使用的 CIDR 地址块以及网关。

表 4-8 租户网络配置文件

```
{
    "tenant_name":"tenantA",
    "subnet":[{
        "subnet_name":"tenantA_subnet1",
        "cidr":"192.168.20.0/24",
        "gateway":"192.168.20.1"
    }]
}
```

4.4 本章小结

本章在第三章各模块设计的基础之上，详细介绍了强多租户模型实现细节。首先通过设计 k8s-keystone-auth 组件实现了 Keystone 和 Kubernetes 的无缝集成，利用 Keystone 身份服务组件实现用户的管理、认证以及授权。接着介绍了计算资源隔离的实现，通过引入 Kata 安全容器实现 Pod 独立虚机运行，解决了内核共享带来的问题。最后介绍了如何利用 Contiv-VPP 构建容器网络环境并实现租户网络隔离，利用 Contiv-VPP 接管 Kubernetes 集群的网络，并实现了服务的负载均衡以及流量控制，同时还详细介绍了 Contiv Plugin、Network-management 以及 Network-cli 插件的实现细节。

第五章 系统测试与结果分析

Kubernetes 强多租户模型多用于公有云服务商以及公司组织架构比较复杂的场景，因此在测试本文的强多租户模型时需要尽可能模拟出真实的生产环境，本章将在实验室服务器上搭建 Kubernetes 集群，模拟在拥有复杂组织架构的公司中使用 Kubernetes 集群的应用场景，并按照本文的设计方案实现一个强多租户模型。环境搭建完成之后将进行功能测试和性能测试。功能测试主要包含租户访问控制、资源隔离以及租户网络管理三个方面。性能测试主要包括对租户创建时延以及集群网络性能的测试。

5.1 测试环境

本文所有测试均在实验室服务器提供的虚拟机上进行，使用了三台虚拟机搭建 Kubernetes 集群，其中一台作为 Master 节点管理集群，另外两台作为 Worker 节点，各个节点的配置情况如表 5-1 所示。操作系统使用 Ubuntu，内核版本 4.15，Kubernetes 使用的是 1.24 版本，其它相关软件版本如表 5-2 所示。

表 5-1 Kubernetes 集群配置

IP 地址	角色	内存	CPU	硬盘
10.20.100.34	Master	8G	4C	50G
10.20.100.35	Worker	4G	2C	50G
10.20.100.36	Worker	4G	2C	50G

表 5-2 软件配置

软件	版本	描述
Ubuntu	18.04.5	虚拟机操作系统
Kubernetes	1.24	容器编排
Kata	1.13	安全容器引擎
Keystone	Stein	OpenStack 认证系统
Contiv-VPP	v3.4.2	网络插件

5.2 租户访问控制测试及结果分析

在本文设计方案中有三类角色，分别是系统管理员、租户管理员以及普通用户，本小节将分别对这些角色进行访问控制测试，检测它们访问资源时是否符合强多租户模型的访问控制要求。

（1）系统管理员

由于要使用 OpenStack 的 Keystone 组件来进行用户管理，所以首先要在 kubernetes 中配置使用 Keystone 凭证，让 Keystone 来负责用户管理，然后通过修改系统变量来登录系统管理员账号，具体过程如图 5-1 所示。

```
root@master:~# kubectl config set-credentials keystoneuser -auth-provider=keystone
root@master:~# kubectl config set-context -Cluster=kubernetes --user=keystoneuser keystoneuser@k8s
root@master:~# kubectl config use-context keystoneuser@k8s
root@master:~#
root@master:~# export K8S_USERNAME="admin"
root@master:~# export K8S_PASSWORD="admin”
root@master:~# export K8S_TENANT="admin"
root@master:~# export K8S_DOMAINID="default"
root@master:~#
```

图 5-1 配置系统管理员账号

从图 5-2 可以看出，登录成功后系统管理员能够访问所有命名空间下的资源，例如节点信息、租户信息、网络信息、子网信息等。

```
root@master:~# kubectl get node
NAME          STATUS       ROLES            AGE     VERSION
master        Ready        master           1d      v1.24.5
worker1       Ready        slave            1d      v1.24.5
worker2       Ready        slave            1d      v1.24.5
root@master:~#
root@master:~# kubectl get tenant
NAME            AGE
defaut-tenant   1d
root@master:~# kubectl get namespace
NAME            STATUS          AGE
default         Active          1d
kube-public     Active          1d
kube-system     Active          1d
root@master:~# kubectl get pod -n default
NAMESPACE       NAME                          READY          STATUS         RESTARTS       AGE
default         k8s-keystone-auth-xb24z       1/1            Running        0              1d
default         k8s-tenant-controller         1/1            Running        0              1d
default         k8s-rbac-controller           1/1            Running        0              1d
default         k8s-contiv-plugin             1/1            Running        0              1d
default         k8s-network-management        1/1            Running        0              1d
root@master:~# contiv netctl network list
ID                                          NAME                          SUBNETS
n3xc3j7x-9xjs-12n-n08s-d03c12p721d          contiv-default-network        4d2fspb4-1xcl-430c-8d52-52bBsa19124d
x1a9ajs1-7n1l-01f-a11s-a042fc9f729          public                        cb1d4c25-dba3-4a6c-4ded-eff4650a21db
root@master:~#
root@master:~# contiv netctl subnet list
ID                                          NAME                          NETWORK                                 SUBNET
4d2fspb4-1xcl-430c-8d52-52bBsa19124d        contiv-default-subnet         n3xc3j7x-9xjs-12n-n08s-d03c12p721d      10.38.0.0/16
cb1d4c25-dba3-4a6c-4ded-eff4650a21db        public-subnet                 x1a9ajs1-7n1l-01f-a11s-a042fc9f729      192.168.1.0/24
root@master:~#
root@master:~# contiv netctl  router list
ID                                  NAME                  STATUS              STATE
2x1425aq-px1x-1a2d-8v12-exx1475a12vc   contiv-defaut-router    Active              UP
root@master:~#
```

图 5-2 系统管理员访问集群资源

系统管理员拥有 admin 权限，除了能够操作集群中的资源还能创建租户，从图 5-3 可以看到，系统管理员创建了一个租户 tenant-manager，并配置了用户名和密码，tenant-manager 创建成功以后，租户控制器和网络控制器为该租户创建了相应的 Namespace 资源和 Network 资源，在 Keystone 和 Contiv 中也都创建了对应的资源，满足租户创建预期。

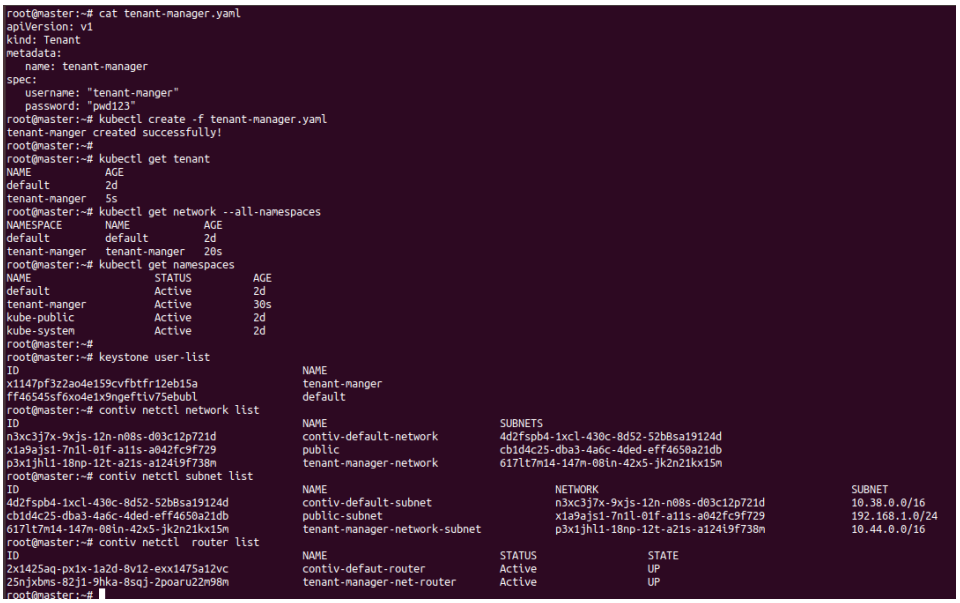

图 5-3 系统管理员创建租户管理员

系统管理员还能管理普通用户，如图 5-4 所示，系统管理员创建了一个普通用户 demo-tenant，该用户将在后续测试中检测普通用户的访问控制功能。

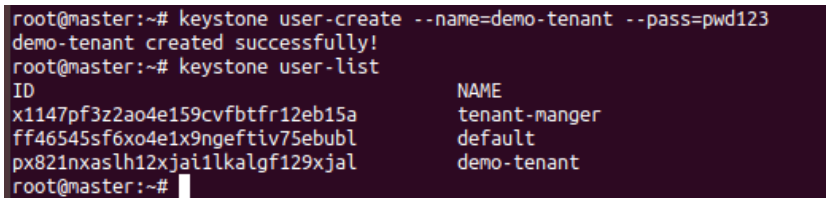

图 5-4 创建普通用户

（2）租户管理员

按照前文步骤创建两个租户管理员 tenant1 和 tenant2，然后分别在 tenant1 和 tenant2 下创建一个 Pod，登录 tenant1 后获取 Pod 信息时只能够获取到 tenant1 下面的 Pod 信息，访问 tenant2 下的 Pod 时会出现拒绝访问的错误，符合预期，具体测试过程如图 5-5 所示。

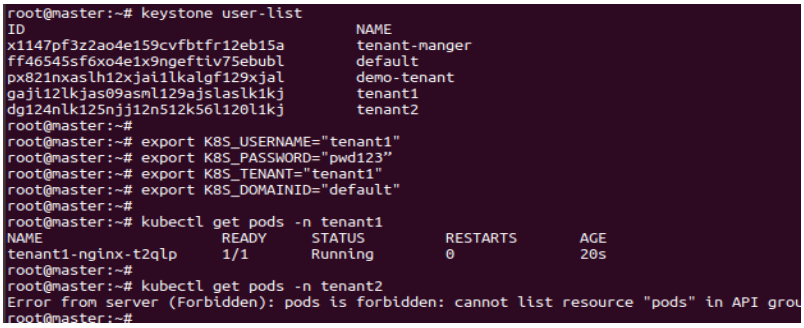

图 5-5 测试租户管理员权限

租户管理员具有当前租户下资源的 CRUD 权限，如图 5-6 所示，tenant1 在当前命名空间下成功部署了一个 nginx 服务，说明 tenant1 在当前命名空间下具有创建服务的权限，创建成功以后删除刚才创建的服务，发现能够删除成功。当 tenant1 尝试在 tenant2 命名空间下创建 nginx 服务时出现禁止访问的错误，满足租户管理员访问控制要求，具体测试过程如图 5-6 所示。

```
root@master:~# kubectl apply -f tenant1-nginx.yaml
deployment.apps/tenant1-nginx-test created
root@master:~#
root@master:~#  kubectl get pods-n tenant1
NAME                          READY     STATUS       RESTARTS       AGE
tenant1-nginx-t2qlp           1/1       Running      0              3m20s
tenant1-nginx-test-sn21l      1/1       Running      0              10s
root@master:~#
root@master:~# kubectl delete pod tenant1-nginx-test-sn21l -n tenant1
pod "tenant1-nginx-test-sn21l" deleted
root@master:~#  kubectl get pods-n tenant1
NAME                  READY     STATUS       RESTARTS       AGE
tenant1-nginx-t2qlp   1/1       Running      0              3m28s
root@master:~#
root@master:~# kubectl create namespace tenant3
Error from server (Forbidden): namespace is forbidden: cannot create resource "namespaces" in API group at the cluster scope.
root@master:~#
root@master:~#  kubectl apply -f tenant2-nginx.yaml
Error from server (Forbidden): error when retrieving current configuration of:Resource: "apps/v1, Resource=deployments" GroupersionKind: "apps/v1,Ki
Name: "tenant2-nginx",Namespace:"tenant2"from server for: "tenant2-nginx.yaml": deployments.apps "tenant2-nginx" is forbidden: cannot get resource "
root@master:~#
```

图 5-6 tenant1 访问租户空间外资源的操作结果

（3）普通用户

在本文的设计方案中普通用户创建后没有任何资源的访问权限，需要它的租户管理员为其绑定权限后才能访问租户下的资源，对于其他租户的资源同样没有访问权限，具体测试过程如图 5-7 所示。

```
root@master:~# export K8S_USERNAME="demo-tenant"
root@master:~# export K8S_PASSWORD="pwd123"
root@master:~# export K8S_TENANT="demo-tenant"
root@master:~# export K8S_DOMAINID="default"
root@master:~# kubectl create -f test-pod.yaml
Error from server (Forbidden): error when creating "test-pod.yaml": pods is forbidden: User
root@master:~#
root@master:~# export K8S_USERNAME="tenant-manager"
root@master:~# export K8S_PASSWORD="pwd123"
root@master:~# cat demo_rbac.yaml
apiVersion: rbac.authorization.k8s.o/v1
kind: RoleBinding
metadata:
  name: create-pods
subjects:
- kind: User
  name: demo-tenant
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization,k8s.io/v1
kind: Role
metadata:
  name: pod-creater
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get","watch","list","create"]
root@master:~# kubectl apply -f demo_rabc.yaml -n=tenant-manager
rolebinding "create-pods" created
role "pod-creater" created
root@master:~# export K8S_USERNAME="demo-tenant"
root@master:~# export K8S_PASSWORD="pwd123"
root@master:~# kubectl create -f test-pod.yaml
pod "demo-pod" created
root@master:~#
```

图 5-7 普通用户权限测试

目前为止，三类用户角色的访问控制测试已经全部完成，从测试结果可以看出，本文的访问控制方案能够满足 Kubernetes 强多租户模型的要求，符合实验预期。

5.3 计算资源隔离测试

集群搭建完成之后，从图 5-8 中可以看出，集群中所有节点的 container-runtime 都已经修改为 containerd，并且所有节点都处于 Ready 状态。

```
root@master:~# kubectl get nodes -o wide
NAME      STATUS       ROLES    AGE    VERSION        INTERNAL-IP          EXTERNAL-IP          OS-IMAGE              CONTAINER-RUNTIME
master    Ready        master   10h    v1.24.5        192.168.10.120       <none>               Ubuntu 18.04.5 LTS    containerd://1.6.14
worker1   Ready        <none>   9h     v1.24.5        192.168.10.130       <none>               Ubuntu 18.04.5 LTS    containerd://1.6.14
worker2   Ready        <none>   8h     v1.24.5        192.168.10.140       <none>               Ubuntu 18.04.5 LTS    containerd://1.6.14
root@master:~#
```

图 5-8 集群节点信息

计算资源隔离主要通过测试集群在创建非可信容器时由 Kata 来执行，为了完成测试，创建 untrusted-redis.yaml 文件，部署一个非可信的 Pod，使用非可信容器创建，该文件内容如表 5-3 所示。

表 5-3 untrusted-redis.yaml 文件内容

```
apiVersion: v1
kind: Pod
metadata:
   name:untrusted-redis
   annotations:
      io.kubernetes.cri.untrusted-workload: "true"
spec:
   containers:
   - name: untrusted-redis
      image: redis
```

从图 5-9 中可以看出，将该 yaml 文件应用到集群之后 Pod 能够顺利创建，此时集群中新增了一个 Kata 容器，说明刚才的应用被识别为非可信应用，并且是通过 Kata 来完成创建的，符合预期结果并且能够满足多租户计算资源隔离要求。

```
root@master:# kubectl apply -f untrusted-redis.yaml
pod/untrusted-redis created
root@master:~# kubectl get pods
NAME                  READY          STATUS                RESTARTS        AGE
untrusted-redis       0/1            ContainerCreating     0               10s
root@master:~# kata-runtime list
ID                                                                 PID             STATUS          BUNDLE
12cxz11e798376Vfxc26v3x31a2x990cx1997x2c35v32g45x2x36b0921n5Cv0cz   29983           running         /runc/contained/io.cc
```

图 5-9 非可信容器创建结果

5.4 多租户网络测试及结果分析

本小节主要测试租户网络的创建、Pod 网络的配置、租户下 Pod 网络连通性、不同租户下 Pod 网络隔离性。在本节中会创建两个租户 tenantA 和 tenantB，并且为它们创建两个 Network 资源对象。在 tenantA 和 tenantB 下面都会划分两个子网，其中 tenantA 下的子网是 tenantA_subnet1 和 tenantA_subnet2，tenantB 下的子网是 tenantB_subnet1 和 tenantB_subnet2。同时会为每个子网创建一个 Namespace 对象，通过测试不同 Namespace 下 Pod 的网络连通性来检测本文设计的多租户网络是否满足强多租户模型的网络要求。接下来将详细介绍具体的测试过程。

5.4.1 租户网络创建测试

从第四章多租户网络的实现可以知道，每个租户都会有一个对应的 Contiv-VPP 中的 Network 资源。租户创建成功后通过 Network-cli 模块自定义的命令行调用 Network-management 提供的服务完成租户网络的创建。如图 5-10 所示的 tenantA-network.json 文件，为租户 tenantA 创建 Network 的同时还创建了子网 tenantA_subnet1 和 tenantA_subnet2，并配置了子网的网关以及 CIDR 地址块。

```
root@master:~# vi tenantA-network.json
root@master:~# cat tenantA-network.json
{
    "tenant_name":"tenantA",
    "subnets":[
        {
                "subnet_name":"tenantA_subnet1",
                "cidr":"192.168.20.0/24",
                "gateway":"192.168.20.1"
        },
        {
                "subnet_name":"tenantA_subnet2",
                "cidr":"192.168.30.0/24",
                "gateway":"192.168.30.25"
        }
    ]
}
root@master:~#
```

图 5-10 tenantA-network.json 文件内容

通过 network-cli create-tenant-network 命令创建租户 tenantA 的 Network 资源，创建结果如图 5-11 所示，根据 network-cli get-tenant-network-info 命令查询到了 tenantA 的网络信息，并且创建了两个子网。其中，子网 tenantA_subnet1 的 CIDR 地址块为 192.168.20.0/24，网关是 192.168.20.1；子网 tenantA_subnet2 的 CIDR 和网关分别是 192.168.30.0/24 和 192.168.30.1。

```
root@master:~# network-cli create-tenant-network --file tenantA-network.json
Create Network successfully!
root@master:~# network-cli get-tenant-network-info tenantA
{
    "tenant_name:"tenantA",
    "uid":13551xcs-2dxv-as2d-124a-4cv441789m8k",
    "subnets":[
        {
            "subnet_name":"tenantA_subnet1",
            "uid":"bv45asf3-1vv5-2vb3-12n5-1917j87g90nv",
            "cidr":"192.168.20.0/24",
            "gateway":"192.168.20.1"
        },
        {
            "subnet_name":"tenantA_subnet2",
            "uid":"xv21cbs1-3xv1-0nx3-33n5-1277k12j87bm",
            "cidr":"192.168.30.0/24",
            "gateway":"192.168.30.1"
        }
    ]
}
root@master:~#
```

图 5-11 tenantA 网络资源创建情况

按照上述步骤创建另外一个租户 tenantB 的网络资源，同样也为它划分两个子网 tenantB_subnet1 和 tenantB_subnet2，子网 tenantB_subnet1 的 CIDR 地址块是 192.168.40.0/24，tenantB_subnet2 的 CIDR 地址块是 192.168.50.0/24，后续测试将会用到这两个子网。

5.4.2 Pod 网络配置测试

在上一小节中已经成功为 tenantA 创建了 Network 资源并且还为其划分了两个子网，本节将介绍如何将 Pod 配置到租户网络中。如图 5-12 所示，创建一个 Namespace 资源对象，然后将该 Namespace 绑定到 tenantA 租户的子网下。

```
root@master:~# cat tenantA-namespace1.yaml
apiVersion:v1
kind: Namespace
metadata:
    name: tenantA-namespace1
    annotations:
        subnetID: bv45asf3-1vv5-2vb3-12n5-1917j87g90nv
root@master:~#
```

图 5-12 创建 Namespace 并绑定指定子网

总共创建四个 Namespace 对象，它们分别是 tenantA_namespace1、tenantA_namespace2、tenantB_namespace1 以及 tenantB_namespace2，分别将它们绑定到子网 tenantA_subnet1、tenantA_subnet2、tenantB_subnet1 以及 tenantB_subnet2。创建结果如图 5-13 所示。

```
root@master:~# kubectl get namespace
NAME                  STATUS           AGE
default               Active           12d
tenantA_namespace1    Active           1d
tenantA_namespace2    Active           1d
tenantB_namespace1    Active           1d
tenantB_namespace2    Active           1d
root@master:~#
```

图 5-13 Namespace 创建结果

Namespace 绑定到子网后就可以测试 Pod 网络的配置了，如图 5-14 所示的 nginx-deloyment.yaml 文件，使用 Deployment 创建了副本数量为 3 的 Pod，Pod 中部署了 Nginx 服务，并且将 Pod 指定到 10.20.100.35 宿主机的 tenantA_subnet1 命名空间下面。

```
root@master:~# cat nginx-deloyment.yaml
apiVersion: v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: tenantA_namespace1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        kubernetes.io/hostname: 10.20.100.35
      containers:
        - name: nginx-container
          image: nginx
          ports:
            - containerPort: 80
root@master:~#
```

图 5-14 nginx-deloyment.yaml 文件内容

通过 kubectl apply 命令将文件应用到 Kubernetes 集群中，创建过程如图 5-15 所示，从图中可知，3 个 Pod 都成功创建并运行正常，它们都被分配到了 tenantA_subnet1 子网下的 IP 地址。

```
root@master:~# kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
root@master:~# kubectl get deployments -n tenantA_namespace1
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           1m
root@master:~# kubectl get pods -n tenantA_namespace1 -o wide
NAME                               READY   STATUS    RESTARTS   AGE     IP              NODE
nginx-deployment-xc3vs      1/1     Running   0          1m14s   192.168.20.10   10.20.100.35
nginx-deployment-2fxza      1/1     Running   0          1m17s   192.168.20.11   10.20.100.35
nginx-deployment-jgn2x      1/1     Running   0          1m20s   192.168.20.12   10.20.100.35
root@master:~#
```

图 5-15 Pod 创建情况

5.4.3 网络连通性测试

本小节主要针对同一租户下 Pod 之间的网络连通性测试，由于租户资源对应的 Namespace 可能位于同一个宿主机上，也可能在不同宿主机上，子网可以是同一子网，也可以是不同子网，所以可以分为以下四种情况进行测试。

（1）同一宿主机相同子网下 Pod 网络连通性测试。

在上一小节中创建了 3 个 Pod，登录其中两个 Pod，通过 Ping 命令测试这两个 Pod 之间的网络互通性，测试结果如图 5-16 所示，从图中可以看到，两个 Pod 能够 Ping 通，由于都在子网 tenantA_subnet1 下，所以网络流量没有经过其他设备。

```
root@nginx-deployment-xc3vs:/# ping 192.168.20.11
PING 192.168.20.11 (192.168.20.11) 56 bytes of data.
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=1 ttl=64 time=0.440 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=2 ttl=64 time=0.140 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=3 ttl=64 time=0.230 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=4 ttl=64 time=0.141 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=5 ttl=64 time=0.246 ms
^C
--- 192.168.20.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 1.197ms
rtt min/avg/max/mdev = 0.140/0.240/0.440/0.103 ms
root@master:~#
```

图 5-16 同节点同租户同子网下 Pod 网络连通性测试

（2）同一宿主机不同子网下 Pod 网络连通性测试。

按照前文方法在 tenantA_namespace2 下创建一个 Pod，使用子网 tenantA_subnet2 分配的 IP 地址，创建结果如图 5-17 所示。

```
root@master:~# kubectl get pods -n tenantA_namespace2 -o wide
NAME         READY   STATUS    RESTARTS   AGE     IP              NODE
pod-zxb2z    1/1     Running   0          1m14s   192.168.30.10   10.20.100.35
root@master:~#
```

图 5-17 同一宿主机不同子网下 Pod 的创建

登录该 Pod 并和 tenantA_subnet1 下的 Pod 进行测试，从图 5-18 可以看出两个 Pod 能够 Ping 通，但是由于在不同子网下流量需要经过路由设备，所以网络延迟会变大。

```
root@master:~# kubectl exec -it pod-zxb2z /bin/bash
root@pod-zxb2z:/# ping 192.168.20.11
PING 192.168.20.11 (192.168.20.11) 56 bytes of data.
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=1 ttl=64 time=3.52 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=2 ttl=64 time=2.61 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=3 ttl=64 time=1.98 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=4 ttl=64 time=1.56 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=5 ttl=64 time=1.82 ms
^C
--- 192.168.20.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 12.94ms
rtt min/avg/max/mdev = 1.56/2.31/3.52/0.986 ms
root@pod-zxb2z:/#
```

图 5-18 同一宿主机不同子网下 Pod 网络连通性测试

（3）不同宿主机相同子网下 Pod 网络连通性测试。

由于要测试不同宿主机上 Pod 的网络连通性，所以首先在节点 10.20.100.36 上创建一个 Pod，部署在 tanantA_namespace1 命名空间中，使用子网 tenantA_subnet1 分配的 IP 地址，该 Pod 的 IP 地址是 192.168.20.15，创建结果如图 5-19 所示。

```
root@master:~# kubectl get pods -n tenantA_namespace1 -o wide
NAME         READY   STATUS    RESTARTS   AGE   IP              NODE
pod-23xcf    1/1     Running   0          35s   192.168.20.13   10.20.100.36
root@master:~#
```

图 5-19 同一宿主机不同子网下 Pod 的创建

从图 5-20 可以看出，进入到 pod23xcf 后，Ping 命令发送的数据包可以被另外一个节点中相同子网下的 Pod 接收，说明同一租户下不同子网中的 Pod 也能够互相通信。

```
root@master:~# kubectl exec -it pod-23xcf /bin/bash
root@pod-23xcf:/# ping 192.168.20.11
PING 192.168.20.11 (192.168.20.11) 56 bytes of data.
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=1 ttl=64 time=0.341 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=2 ttl=64 time=0.242 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=3 ttl=64 time=0.232 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=4 ttl=64 time=0.181 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=5 ttl=64 time=0.245 ms
^C
--- 192.168.20.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 1.241ms
rtt min/avg/max/mdev = 0.181/0.148/0.340/0.113 ms
root@pod-23xcf:/#
```

图 5-20 不同宿主机相同子网下 Pod 网络连通性测试

（4）不同宿主机不同子网下 Pod 网络连通性测试。

和上一个测试类似，在节点 10.20.100.36 上创建一个 Pod，但是它使用另外一个子网 tenantA_subnet2 分配的 IP 地址，该 Pod 分配的 IP 地址是 192.168.30.11，从图 5-21 可以发现，该 Pod 创建成功后可以通过 Ping 通 10.20.100.35 节点下不同子网内的 Pod，但是网络延迟有所增加，因为跨子网传输数据需要经过路由设备。

```
root@master:~# kubectl get pods -n tenantA_namespace4 -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP              NODE
pod-34nv1     1/1     Running   0          30s     192.168.30.11   10.20.100.36
root@master:~# kubectl exec -it pod-34nv1 /bin/bash
root@pod-34nv1:/# ping 192.168.20.11
PING 192.168.20.11 (192.168.20.11) 56 bytes of data.
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=1 ttl=64 time=3.36 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=2 ttl=64 time=4.08 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=3 ttl=64 time=3.17 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=4 ttl=64 time=3.68 ms
64 bytes from 192.168.20.11 (192.168.20.11): icmp_seq=5 ttl=64 time=3.75 ms
^C
--- 192.168.20.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 18.04ms
rtt min/avg/max/mdev = 3.17/3.608/4.08/1.115 ms
root@pod-34nv1:/#
```

图 5-21 不同宿主机不同子网下 Pod 网络连通性测试

5.4.4 网络隔离性测试

网络隔离是指不同租户下 Pod 之间网络要隔离，不同互相访问。本小节将对同一宿主机以及不同宿主机下的租户网络隔离性进行测试，可以分为以下两种情况。

（1）相同宿主机下不同租户网络隔离性测试。

首先在节点 10.20.100.35 创建两个 Pod，让它们部署在不同租户所对应的命名空间中，如图 5-22 所示，在 tenantA_namespace1 下创建了 pod-8jxn3，在

tenantB_namespace1 下创建了 pod-9kna1。从图中可以看出，两个 Pod 之间的数据包无法到达，说明即便是在同一宿主机下，不同租户的网络也是互相隔离的。

```
root@master:~# kubectl get pods -n tenantA_namespace1 -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP              NODE
pod-8jxn3     1/1     Running   0          28s     192.168.20.18   10.20.100.35
root@master:~# kubectl get pods -n tenantB_namespace1 -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP              NODE
pod-9kna1     1/1     Running   0          30s     192.168.40.10   10.20.100.35
root@master:~# kubectl exec -it pod-8jxn3 /bin/bash
root@pod-8jxn3:/# ping 192.168.40.10
PING 192.168.40.10 (192.168.40.10) 56(84) bytes of data.
^C
--- 192.168.40.10 ping statistics ---
9 packets transmitted, 0 received, 100% packet loss, time 8170ms
root@pod-8jxn3:/#
```

图 5-22 相同宿主机下租户网络隔离性测试

（2）不同宿主机下不同租户网络隔离性测试。

在节点 10.20.100.36 下创建一个 Pod，将该 Pod 部署在 tenantB_namespace2 命名空间下，从图 5-23 中可以看出，该 Pod 和 tenantA_namespace1 下的 Pod 进行通行时网络数据包无法到达，说明它们之间的网络是隔离的。

```
root@master:~# kubectl get pods -n tenantB_namespace2 -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP              NODE
pod-xp2x1     1/1     Running   0          28s     192.168.50.10   10.20.100.36
root@master:~# kubectl exec -it pod-xp2x1 /bin/bash
root@pod-xp2x1:/# ping 192.168.20.10
PING 192.168.20.10 (192.168.20.10) 56(84) bytes of data.
^C
--- 192.168.20.10 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7148ms
root@pod-xp2x1:/#
```

图 5-23 不同宿主机下租户网络隔离性测试

5.5 性能测试及结果分析

相比于 Kubernetes 原生系统，本文的强多租户模型设计方案对租户资源和租户网络改动较大，为了检测本文方案的性能表现故需对租户创建时延和租户网络性能进行比较。

5.5.1 租户资源创建时延测试

系统管理员发起创建租户的请求，租户控制器完成租户创建的所有过程，为了测试系统管理员发起请求后经过多长时间该租户才能变成 available 状态，需要租户控制器的代码中添加带有时间戳格式的输出信息。模拟系统管理员发起大量创建租户的请求，从表 5-4 可以看出，单个租户的创建大约需要 4 秒的时间，当同时创建 100 个租户时，一共用了 9.3 分多钟，平均一个租户大约需要 5.5 秒；当同时创建 500 个租户时用了接近 49 分钟，每个租户用了接近 6 秒的时间；同时创建 1000 个租户用时 105 分钟，平均每个租户用了 6.3 秒。

表 5-4 租户创建时延测试结果

租户创建请求数量	总耗时（ms）	平均耗时（ms）
1	3968	3968
100	558900	5589
500	2958012	5916
1000	6312986	6313

通过 jmeter 对租户创建接口进行压力测试，接口响应时间的测试结果如图 5-24 所示，可以发现大多数请求的响应时间都是 4000 毫秒到 6000 毫秒之间，少部分请求响应时间大于 7000 毫秒，说明本文设计的租户创建流程满足预期，能够满足大多数场景的要求。

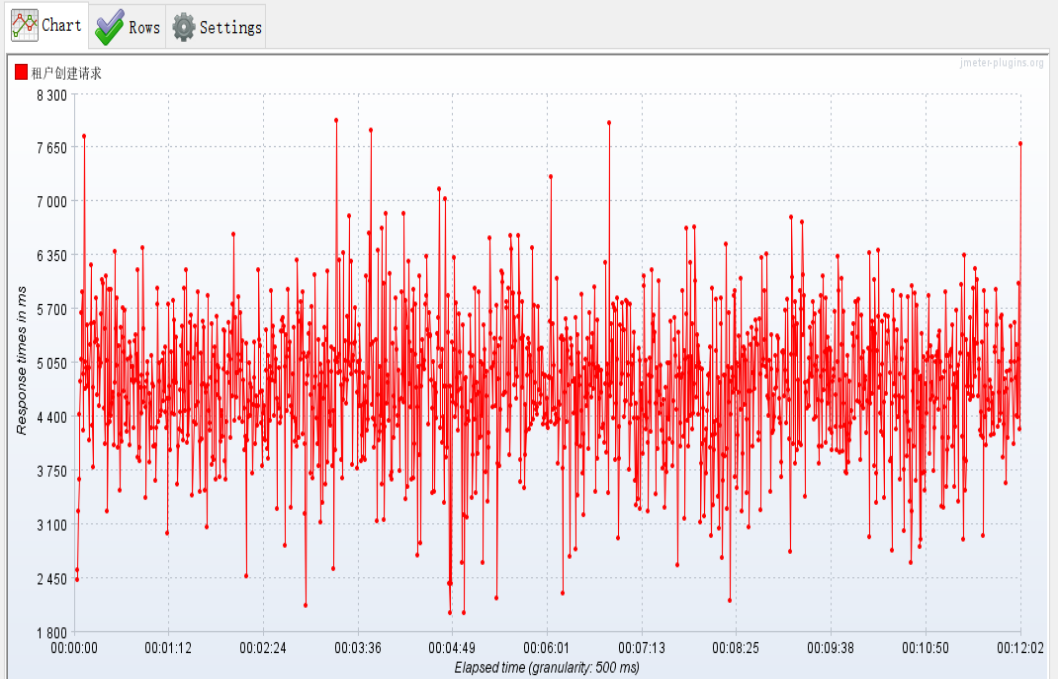

图 5-24 租户创建接口压测结果

5.5.2 租户网络性能测试

Kubernetes 使用了 Flannel 网络插件实现了扁平全通的网络，不满足强多租户模型网络隔离的要求，本文方案对多租户网络进行了改造，因此需要对改造后的网络性能进行测试，并与 Kubernetes 原生网络系统进行比较。使用 qperf 网络性能测试工具，它可以同时测试网络时延和带宽。不同租户之间的网络是隔离的，所以在测试网络性能时都是在同一租户下进行的，主要测试以下几种情况：

（1）同一节点相同子网

（2）同一节点不同子网

（3）不同节点相同子网

（4）不同节点不同子网

使用 qperf 时需要将通信双方一个设置为服务端，另一个设置为客户端，然后使用 tcp_lat 和 tcp_bw 获取网络时延和带宽，多次测试结果的平均值如表 5-5 所示。测试结果表明，本文设计的多租户网络和 Kubernetes 原生 Flannel 网络相比，在各种测试条件下都拥有更低的网络延迟以及更高的网络带宽，符合预期结果。

表 5-5 网络性能测试结果

测试场景	Kubernetes 原生 Flannel 网络		本文多租户网络设计方案	
	时延	带宽	时延	带宽
不同节点	0.086 ms	101 MB/sec	0.086 ms	101 MB/sec
同节点同子网	0.266 ms	78.4 MB/sec	0.205 ms	82.4 MB/sec
同节点不同子网	1.712 ms	70.6 MB/sec	1.316 ms	77.9 MB/sec
不同节点同子网	0.404 ms	69.5 MB/sec	0.311 ms	75.8 MB/sec
不同节点不同子网	1.725 ms	53.8 MB/sec	1.327 ms	60.2 MB/sec

5.6 本章小结

本章是对本文强多租户模型解决方案的测试，从功能测试和性能测试两个方面出发，对访问控制模块、计算资源隔离模块以及多租户网络进行了系统全面地测试，从测试结果可知，本文实现的 Kubernetes 强多租户模型解决方案满足预期，能够达到绝大多数生产场景的要求。

第六章 总结与展望

6.1 总结

最近几年，云计算技术快速发展，其中以 Kubernetes 为代表的的容器云平台逐渐成为了行业首选，随着业务规模以及复杂度的增加， Kubernetes 原生的多租户模型在金融、通信以及银行等要求强多租户隔离的领域正面临着巨大的挑战。如今，如何在 Kubernetes 基础上构建一个强多租户模型是众多互联网企业以及云计算厂商共同面临的问题。

本文在现有研究成果基础之上，充分调研 Kubernetes 强多租户模型市场需求以及研究现状之后，从租户资源隔离出发，对多租户访问控制、计算资源隔离以及多租户网络等几个方面进行了系统全面地设计。本文的主要工作与成果包含以下几个方面：

（1）在 Kubernetes 原生系统中引入第三方用户认证组件 Keystone，通过设计租户 API 来明确强多租户模型的租户定义，用户只需简单的 API 调用就能实现用户管理等功能，并为计算资源隔离以及多租户网络管理打下了基础。

（2）本文在用户管理模块设计了三层用户模型：系统管理员、租户管理员以及普通用户，通过 RBAC 控制器实现对用户的自动授权与鉴权。

（3）使用 Kata 容器实现底层容器隔离，对可信容器和非可信容器使用不同的 CRI 实现，保证容器内应用运行时的内核隔离，从而实现租户计算资源隔离。

（4）在多租户网络管理中使用插件化的思想，设计了 Contiv Plugin、Network-management 以及 Network-cli 模块。通过修改 kubelet 插件加载流程将 Contiv Plugin 绑定到 kubelet 启动时的插件列表中，实现 Pod 网络的自动配置。另外，还利用 gRPC 框架实现网络服务之间的调用，Network-cli 模块通过调用 Network-management 提供的接口实现对租户网络以及子网的管理。

（5）在 Kubernetes 中使用 Contiv-VPP 构建容器网络，利用 VPP 引擎来优化容器间网络流量的处理，提升数据的处理速度，并通过自定义网络策略实现流量控制、负载均衡等功能，增强了集群中服务的可用性。

6.2 展望

本文设计方案使用了插件化的设计思想，在设计访问控制、用户管理、网络管理方案时对 Kubernetes 原生系统侵入性小，本文解决方案具有良好的拓展与升级

能力，并且在拓展 API 时尽量考虑到了 API 的易用性，用户只需要简单的操作就能实现对 Kuberntes 集群的管理。但是，本文方案仍然有一些不足，需要进行后续地优化：

（1）从第五章系统测试结果可以看出，本文的设计方案在小规模生产环境中能够满足用户需求，但是在大规模生产环境中的表现还需进一步的完善，需要对访问控制、网络等多方面进行改进和优化。

（2）在设计 Kubernetes 强多租户模型时仅考虑如何实现各个模块，对于 Kubernetes 集群的安全问题没有进行详细的设计，在实际生产环境中可能存在各种各样的攻击，所以在后续的研究中应该考虑集群安全防护的问题。

（3）本文的设计虽然尽可能的减少对 Kubernetes 的侵入性，但是对 Kubernetes 的原生系统做了修改，因此无法通过 Kubernetes 官方的一致性测试，在后续研究中应该考虑设计一套标准实现，遵循 Kubernetes 的设计思路和规范，用于指导完善 Kubernetes 强多租户模型。

参考文献

[1] Mohammed C M, Zeebaree S R M. Sufficient comparison among cloud computing services: IaaS, PaaS, and SaaS: A review[J]. International Journal of Science and Business, 2021, 5(2): 17-30.

[2] 房秉毅, 张云勇, 程莹, 等.云计算国内外发展现状分析[J].电信科学, 2010, 26(8): 1-6.

[3] Fox A, Griffith R, Joseph A, et al. Above the clouds: A berkeley view of cloud computing[J]. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 2009, 28(13): 2009.

[4] 武志学.云计算虚拟化技术的发展与趋势[J].计算机应用, 2017, 37(4): 915-923.

[5] Malviya A, Dwivedi R K. A Comparative Analysis of Container Orchestration Tools in Cloud Computing[C]. 2022 9th International Conference on Computing for Sustainable Global Development . IEEE, 2022: 698-703.

[6] Ali M, Khan S U, Vasilakos A V. Security in cloud computing: Opportunities and challenges[J]. Information sciences, 2015, 305: 357-383.

[7] Truyen E, Van Landuyt D, Reniers V, et al. Towards a container-based architecture for multi-tenant SaaS applications[C]. 2016 15th international workshop on adaptive and reflective middleware, Austin, 2016: 1-6.

[8] Pousty S, Miller K. Getting Started with OpenShift: A Guide for Impatient Beginners[M]. California O'Reilly Media, Inc, 2014: 2-18.

[9] Marathe N, Gandhi A, Shah J M. Docker swarm and kubernetes in cloud computing environment[C]. 2019 3rd International Conference on Trends in Electronics and Informatics. IEEE, 2019: 179-184.

[10] Sabharwal N, Pandey P. Pro Google Kubernetes Engine[M]. Berkeley: Apress, 2020, 3-32.

[11] Sultan S, Ahmad I, Dimitriou T. Container security: Issues, challenges, and the road ahead[J]. IEEE access, 2019, 7: 52976-52996.

[12] 黄丹池, 何震苇, 严丽云, 等. Kubernetes 容器云平台多租户方案研究与设计[J]. 电信科学, 2020, 36(9): 102-111.

[13] Rosen R. Namespaces and cgroups, the basis of Linux containers[J]. Seville, Spain, Feb, 2016.

[14] Gao X, Gu Z, Li Z, et al. Houdini's escape: Breaking the resource rein of linux control groups[C]. 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1073-1086.

[15] Rudolph J W, Raemer D B, Simon R. Establishing a safe container for learning in simulation: the role of the presimulation briefing[J]. Simulation in Healthcare, 2014, 9(6): 339-349.

[16] Caraza-Harter T, Swift M M. Blending containers and virtual machines: a study of firecracker and gVisor[C]. 2020 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2020: 101-113.

[17] Randazzo A, Tinnirello I. Kata containers: An emerging architecture for enabling mec services in fast and secure way[C]. 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, 2019: 209-214.

[18] Sultan S, Ahmad I, Dimitriou T. Container security: Issues, challenges, and the road ahead[J]. IEEE access, 2019, 7: 52976-52996.

[19] Soltesz S, Pötzl H, Fiuczynski M E, et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors[C]. 2007 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007. 2007: 275-287.

[20] Bhardwaj A, Krishna C R. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey[J]. Arabian Journal for Science and Engineering, 2021, 46(9): 8585-8601.

[21] Kumar R, Thangaraju B. Performance analysis between runc and kata container runtime[C]. 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). IEEE, 2020: 1-4.

[22] Rad B B, Bhatti H J, Ahmadi M. An introduction to docker and analysis of its performance[J]. International Journal of Computer Science and Network Security (IJCSNS), 2017, 17(3): 228.

[23] Anderson C. Docker [software engineering][J]. Ieee Software, 2015, 32(3): 102-c3.

[24] Rosen R. Resource management: Linux kernel namespaces and cgroups[J]. Haifux,May, 2013, 186:70.

[25] Biederman E W, Networx L. Multiple instances of the global linux namespaces[C]. 2006 Linux Symposium. Citeseer, 2006, 1(1): 101-112.

[26] Miell I, Sayers A. Docker in practice[M]. Simon and Schuster, 2019.

[27] Seyfried S. Resource management in Linux with control groups[C]. 2010 Linux-Kongress. 2010

[28] Burns B, Grant B, Oppenheimer D, et al. Borg, omega, and kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-57.

[29] Truyen E, Kratzke N, Van Landuyt D, et al. Managing feature compatibility in Kubernetes: Vendor comparison and analysis[J]. Ieee Access, 2020, 8: 228420-228439.

[30] 张磊. 深入剖析 kubernetes[M]. 北京: 人民邮电出版社, 2021: 47-51.

[31] Kubernetes-handbook[EB/OL].(2023-01)[2023-03]. https://github.com/feiskyer/Kubernetes-handbook.

[32] Medel V, Rana O, Bañares J Á, et al. Modelling performance & resource management in kubernetes[C]. 2016 9th International Conference on Utility and Cloud Computing. 2016: 257-262.

[33] Kapočius N. Performance studies of kubernetes network solutions[C]. 2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream). IEEE, 2020: 1-6.

[34] Kumar R, Trivedi M C. Networking analysis and performance comparison of Kubernetes CNI plugins[C]. Advances in Computer, Communication and Computational Sciences: Proceedings of IC4S 2019. Springer Singapore, 2021: 99-109.

[35] Calico. Networking[EB/OL]. (2021-8) [2023-3]. https://docs.tigera.io/calico/3.25/about/about-networking.

[36] 控制器模式[EB/OL].(2022-7)[2023-3].https://cloudnative.to/kubebuilder/cronjob-tutorial/controller-overview.html.

[37] 龚正. Kubernetes 权威指南: 从 Docker 到 Kubernetes 实践全接触 (纪念版)[M]. BEIJING BOOK CO. INC., 2017: 180-200.

[38] 罗利民. 从 Docker 到 Kubernetes 入门与实战[M]. 北京: BEIJING BOOK CO. INC., 2019: 124-132.

[39] OpenID. OpenID Connect[EB/OL]. (2022-9) [2023-3]. https://openid.net/connect/.

[40] Dan N, Hua-Ji S, Yuan C, et al. Attribute based access control (ABAC)-based cross-domain access control in service-oriented architecture (SOA)[C]. 2012 International Conference on Computer Science and Service System. IEEE, 2012: 1405-1408.

[41] Coyne E, Weil T R. ABAC and RBAC: scalable, flexible, and auditable access management[J]. IT professional, 2013, 15(03): 14-16.

[42] 李伟东. 基于 Kubernetes 的容器云平台强多租户模型关键技术研究[D]. 杭州: 浙江大学, 2019, 43-44.

[43] Wang X, Du J, Liu H. Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes[J]. Cluster Computing, 2022, 25(2): 1497-1513.

[44] Contiv/VPP Kubernetes Network Plugin[EB/OL]. https://github.com/contiv/vpp.

[45] Indrasiri K, Kuruppu D. gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes[M]. O'Reilly Media, 2020: 3-10