第 3 章
边缘计算软件架构

在“云－边－端”的系统架构中，针对业务类型和所处边缘位置的不同，边缘计算硬件选型设计往往也会不同。例如，边缘用户端节点设备采用低成本、低功耗的 ARM 或者英特尔的 Atom 处理器，并搭载诸如 Movidius 或者 FPGA 异构计算硬件进行特定计算加速；以 SDWAN 为代表的边缘网络设备衍生于传统的路由器网关形态，采用 ARM 或者英特尔 Intel Xeon-D 处理器；边缘基站服务器采用英特尔至强系列处理器。相对硬件架构设计，系统软件架构却大同小异，其特点为与设备无关的微服务、容器及虚拟化技术、云端无服务化套件等。

以上技术应用统一了云端和边缘的服务运行环境，减少了因硬件基础设施的差异而带来的部署及运维问题。而在这些技术背后是云原生软件架构在边缘侧的演化。

典型的边缘系统软件架构如图 3-1 所示。本章节就边缘软件系统的各个重要架构和组件进行介绍。

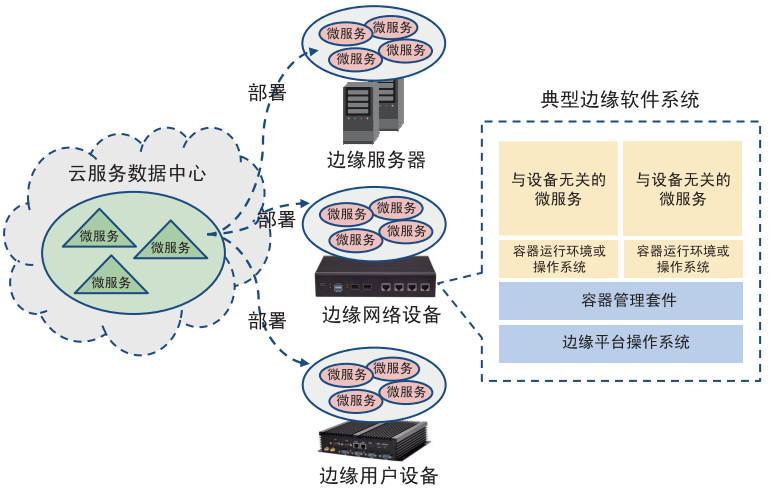

图 3-1　典型的边缘系统软件架构

3.1　云原生

3.1.1　随云计算诞生

早期的应用程序运行在单独的一台计算机上完成有限的功能。它的开发是由规模较小的团队以面向需求、过程或者功能为主要的设计方法构建的；随着计算能力的提高和软件需求的复杂化，大型软件的开发需要由不同的软件团队协作完成。为了保证能够满足需求并提高软件的复用性，面向对象的设计模式和统一建模语言（Unified Modeling Language）成了软件架构设计的主流。云计算的诞生给软件开发的架构和方法带来了新的挑战，例如如何使得软件设计符合负载的弹性需求，如何快速使用集群扩展能力解决系统性能的瓶颈以实现水平扩容，如何使用敏捷开发模式快速迭代、开发、部署应用程序以达到高效的交付，如何使得基础设施服务化并按量支付，如何使得故障得到及时的隔离并自动恢复。

于是，Matt Stine 提出了云原生概念。它是一套设计思想、管理方法的集合，包括 DevOps、持续交付、微服务、敏捷基础设施、康威定律等，以及根据商业能力对公司进行重组，如图 3-2 所示。

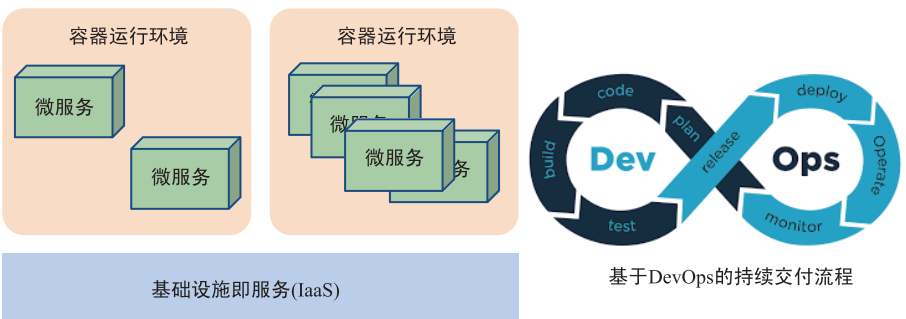

图 3-2　云原生的概念

1. 康威定律

Melvin Conway 在 1968 年发表的论文 *How Do Committees Invent* 指出，系统设计的结构必定复制设计该系统的组织的沟通结构。简单来说，系统设计（产品结构）等同于组织形式。每个设计系统的组织，其产生的设计等同于组织之间的沟通结构。例如，当团队里的所有员工在同一地点工作时，沟通成本较低，开发出来的软件耦合度比较高；若员工分散在不同地点甚至时区，协调沟通成本较高，开发出来的软件则倾向于更加模块化，耦合度低。

2. 持续交付

在 DevOps 的方法中，开发人员提交新的代码改动后，立刻触发自动构建和（单元）测试，并及时地将测试结果反馈给开发团队，这是持续集成。持续交付是在持续集成的基础上，将测试通过的代码自动集成并部署到“类生产环境”中进行下一步严格的自动测试，以确保业务应用和服务符合预期。更进一步，在持续交付的基础上，把部署到生产环境的过程自动化，实现最终的持续部署。

3. 微服务

微服务是一种架构模式，是将传统的单体架构模式的程序拆分为一组小的服务。它们可以被独立部署，运行于虚拟化或容器环境中。各个服务之间采用轻量级的通信机制相互沟通，是松耦合的。微服务通常完成单一的业务模块，对底层的物理硬件架构依赖较小，与设备无关。

4. 敏捷基础设施

提供弹性、按需计算、存储、网络资源能力。可以通过 OpenStack、KVM、Ceph、OVS 等技术手段实现。2015 年，Linux 基金会还专门成立了云本地化计算基金（Cloud Native Computing Foundation）。

3.1.2 单体架构和基于微服务的云原生架构

传统的单体应用模式是把所有展示、业务、持久化的代码都放在一起，而在微服务模式下，应用则是将子业务分布在不同的进程或容器节点中，如图 3-3 所示。

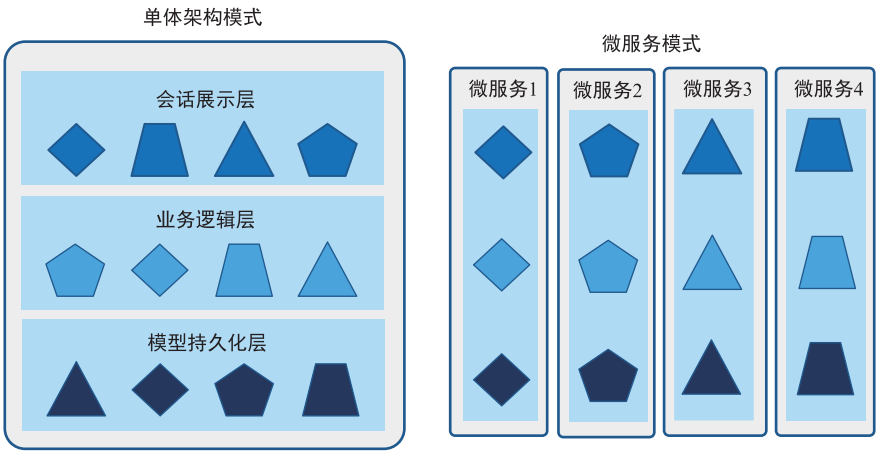

图 3-3　单体架构和微服务架构比较

传统单体架构和云原生架构的比较如表 3-1 所示。

表 3-1　传统单体架构和云原生架构的比较

项目	传统单体架构	基于微服务的云原生架构
系统架构弹性和稳定性	易变，很难预测。传统的应用程序在其体系结构或开发方式上受到需求变化和定制的影响。这些老系统中的应用许多都是单一独立的，需要更长的时间来构建、升级。基于瀑布式批量发布的方法，应用程序的扩展只能逐渐完成。这些系统中的大多数不会在开发环境或测试环境中部署，需要高度的人工干预，容易出现单点故障	一致性和可预测性。云原生应用程序开发框架旨在通过可预测的行为，最大限度地提高系统的弹性。例如，可以使用高度自动化、容器驱动的基础设施将传统的应用程序从本地部署移动到云中，利用公共云的基础设施重新对这些较旧的代码进行平台化改造，使用自动化的开发测试和生产环境重新部署到一致性的基础结构上
操作系统耦合度	操作系统的依赖。传统应用程序的体系结构将应用程序、底层操作系统、硬件、存储及后台服务紧密地耦合在一起。这些依赖使应用程序很难在不同的平台或新的基础设施上进行移植和扩展	操作系统的抽象。云原生应用程序体系结构允许开发人员使用抽象的运行平台，从而摆脱对底层基础设施的各种依赖关系。团队关注的不是配置、修补和维护操作系统，而是软件本身
系统设计和资源利用的灵活性	过度设计。传统单体应用程序是基于定制化的基础设施进行设计的，从而拖延了应用程序的部署周期。通常基于最坏情况下的容量估计，规模过大	高效的资源利用率。云原生的运行时管理工具优化了应用程序的生命周期，包括基于需求的扩展、提高资源利用率、最小化了故障恢复的停机时间
团队协作程度	组织化和流程化的隔阂。传统的 IT 操作是将完成的应用程序代码从开发人员移交给操作人员，然后在生产环境中运行。组织优先权优先于客户价值，导致内部冲突、交付缓慢，以及员工士气低落等问题	促进协同。云原生模式是人、过程和工具的结合，促进了软件开发和管理流程之间的紧密协作，加快了应用程序代码从开发到生产环境的交付和部署速度
软件开发交付方式	瀑布式开发。开发团队定期发布软件，通常间隔几周或几个月。客户想要或需要的功能被延迟，企业将错过竞争、赢得客户的机会	持续交付。开发团队发布独立的微服务软件的更新，获得更紧密的反馈回路，能更有效地响应客户的需求
软件子系统耦合度	紧耦合。单一应用架构将许多不同的服务捆绑到一个部署包中，服务之间有很多不必要的依赖性，导致开发和部署丧失灵活性	松耦合。微服务体系结构将应用程序分解为小的、松散耦合的独立服务。这些服务映射到更小的、独立的开发团队，进行频繁、独立的更新，从而使得扩展、故障转移或重启不会影响其他服务，降低停机成本
运维扩展的复杂度	人工扩展。传统基础设施包括服务器、网络和存储配置。在基础设施扩展中，操作人员很难快速诊断和解决复杂的问题	自动化的扩展性。基础设施自动化的扩展能力消除了人为错误造成的停机时间。在任何规模的部署中始终应用一致性的规则
备份和恢复机制	糟糕的备份和恢复机制。大多数传统体系结构都缺乏自动化备份能力和灾难恢复能力	自动备份和恢复。业务流程被部署到跨虚拟机集群的容器中实现动态管理，以便在应用程序或基础架构发生故障时提供弹性扩展、恢复和重启服务

3.2 微服务

3.2.1 微服务的架构组成

微服务架构如图 3-4 所示，主要由以下几个部分组成。

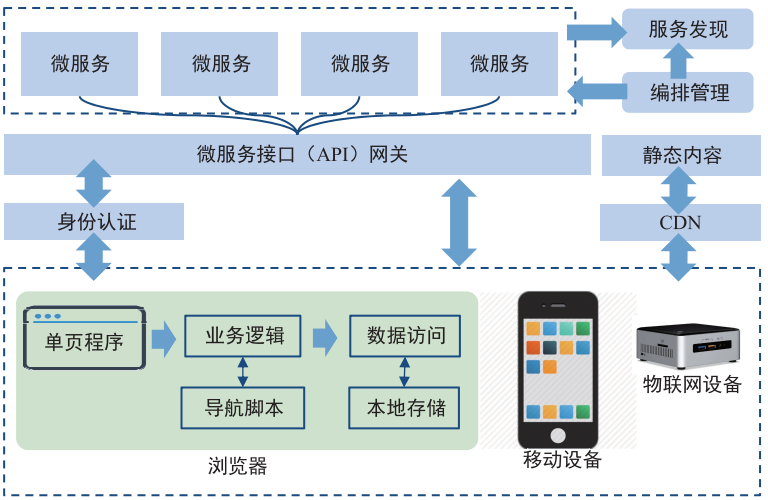

图 3-4 微服务架构

（1）客户端。支持不同类型设备的接入，例如运行在浏览器里面的单页程序、移动设备和物联网设备等。

（2）身份认证。为客户端的请求提供统一的身份认证，然后请求再转发到内部的微服务。

（3）微服务接口（API）网关。作为微服务的入口，提供同步调用和异步消息两种访问方式。同步消息使用 REST（Representational State Transfer），依赖于无状态 HTTP 协议。异步消息使用 AMQP、STOMP、MQTT 等应用。

（4）编排管理。注册、管理、监控所有的微服务，发现和自动恢复故障。

（5）服务发现。维护所有微服务节点列表，提供通信路由查找。

此外，每个微服务都由一个私有数据库来保存数据。微服务的业务功能的生命周期应尽量精简、无状态。

3.2.2 边缘计算中的微服务

云端数据中心根据实时性、安全性和边缘侧异构计算的需求，将微服务灵活地部署到边缘

的用户设备、网关设备或小型数据中心。这体现了分布式边缘计算比传统集中式云计算拥有的巨大优势，而微服务即是算力和 IT 功能部署的载体和最小单位。在边缘设备注册到云服务器提供商以后，这种微服务的部署对于终端用户是非常容易甚至无感的。

亚马逊、微软 Azure 云服务提供商都给出了使用边缘计算加速机器学习中神经网络推理的案例，如图 3-5 所示。机器学习根据现有数据所学习（该过程称为训练）的统计算法，对新数据做出决策（该过程称为推理）。在训练期间，将识别数据中的模式和关系以建立模型。该模型让系统能够对之前从未遇到过的数据做出明智的决策。在优化模型过程中会压缩模型大小，以便快速运行。训练和优化机器学习模型需要大量计算资源，因此与云是天然良配。但是，推理需要的计算资源要少得多，并且往往在有新数据可用时实时完成。要想确保物联网应用程序能够快速响应本地事件，必须能够以非常低的时延获得推理结果。

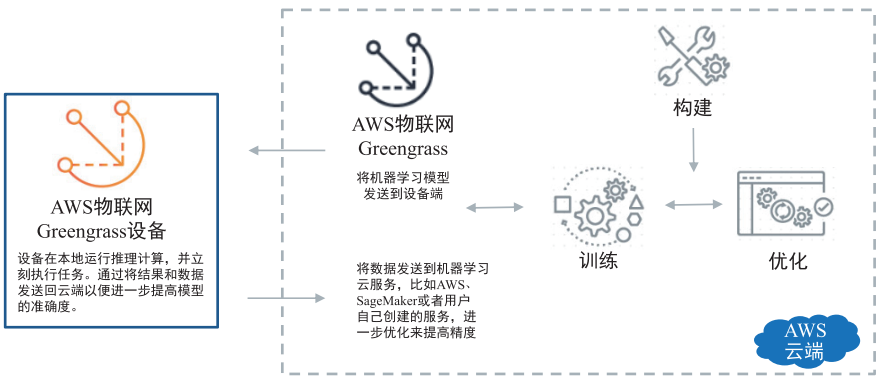

图 3-5　亚马逊将机器学习的推理微服务部署到边缘侧

微软 Azure 的视频流分析系统如图 3-6 所示，提供了运行于物联网设备的容器中的跨平台方案。在云端只需简单配置就可以将机器学习的实时媒体流分析服务部署到靠近用户侧的物联

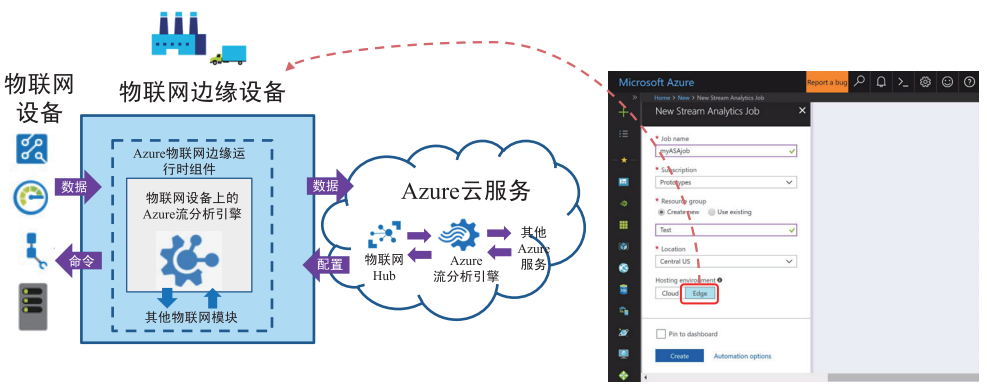

图 3-6　微软 Azure 的视频流分析系统

网设备之上。

在云计算领域，传统 IT 软件的微服务化已经得到了充分的演化，趋于成熟。如前所述，边缘计算是传统的工业领域的 OT（Operational Technology）、通信领域的 CT（Communications Technology）和 IT 的融合，而大部分的 CT 和 OT 的软件是基于整体式架构根据定制的需求开发的。传统 CT 和 OT 软件的微服务化是目前边缘计算产品落地的重要方面之一。

3.3 边缘计算的软件系统

传统云计算是将微服务部署于虚拟机之中，OpenStack 提供了云平台的基础设施。边缘计算是云平台的延伸，但缺少云数据中心的高性能服务器物理设施来部署和运行完整的虚拟化环境。于是，轻量级的容器取代了虚拟机成了边缘计算平台的标准技术之一。2017 年，谷歌开发的 Kubernetes 成了边缘计算平台标准的容器管理编排平台。

从软件架构角度上看，一个简化的边缘系统由边缘硬件、边缘平台软件系统和边缘容器系统组成，如图 3-7 所示。

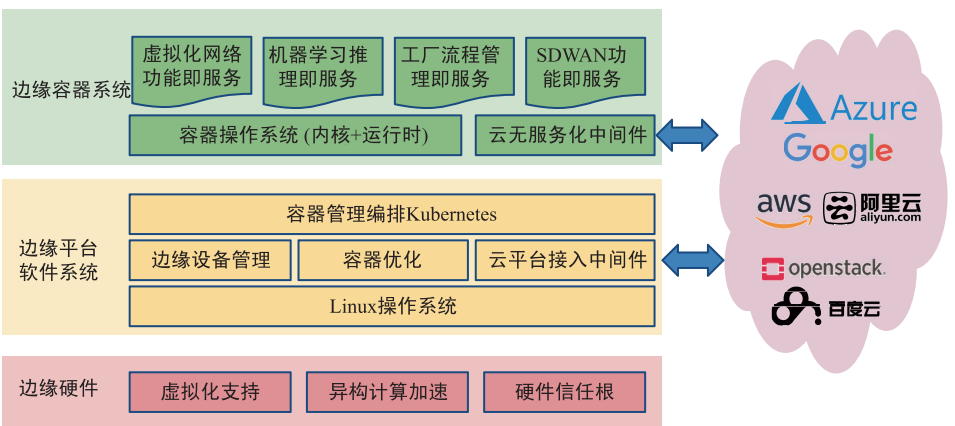

图 3-7　简化的边缘系统

3.3.1 边缘的硬件基础设施

边缘硬件包括边缘节点设备、网络设备和小型数据中心，比较多样化。和传统物联网设备采集数据和简单数据处理不同，边缘硬件需要运行从云端部署的 IT 微服务。所以，边缘硬件一般具有一定的计算能力，使用微处理芯片（MPU）而不是物联网设备使用的微控制器（MCU）。因为边缘计算可以应用到各种领域，如工业、运输、零售、通信、能源等，所以硬件系统也是多种多样的，从低功耗树莓派（Raspberry）系统，到英特尔的酷睿，甚至至强系统。

微软的 Azure 物联网云就支持多达 1000 种设备的认证。

云端的基础设施被抽象成计算节点、网络节点和存储节点，以屏蔽底层基础设施的差异化。而边缘硬件往往使用异构的计算引擎进行加速以满足低功耗、实时性和定制化计算的需要，最常见的是使用 FPGA、Movidius、NPU 对机器学习神经网络推理的加速。而这些异构计算加速引擎很难在云端大规模部署和运维。

边缘设备硬件更加靠近数据源和用户侧，设备和系统的安全相比云数据中心更具有挑战性。和传统物联网设备一样，集成基于硬件的信任根可以极大地保障边缘系统的安全，同时也可以极大地降低网络通信的开销，提高微服务的实时性。如果每次微服务调用都需要通过云做认证，就会带来极高的时延。在芯片上，ARM 的 Trustzone、英特尔的 TPM/PTT 都提供了底层的基于硬件的信任根的支持，同时英特尔的 SGX 也提供了运行态的安全隔离。

以 Docker 为主的容器技术是边缘设备上微服务的运行环境，并不需要特殊的虚拟化支持。然而，硬件虚拟化可以为 Docker 容器提供更加安全的隔离。2017 年年底，OpenStack 基金会正式发布基于 Apache 2.0 协议的容器技术 KataContainers 项目，主要目标是使用户能同时拥有虚拟机的安全及容器技术的迅速和易管理性。

3.3.2　容器技术

云服务提供商使用虚拟化或者容器来构建平台及服务。如图 3-8 所示，应用程序和依赖的二进制库被打包运行在独立的容器中。在每个容器中，网络、内存和文件系统是隔离的。容器引擎管理所有的容器。所有的容器共享物理主机上的操作系统内核。

图 3-8　典型的容器架构

容器和虚拟机的比较如表 3-2 所示。

表 3-2　容器和虚拟机的比较

特性	容器	虚拟机
启动时间	秒级	分钟级
硬盘使用单位	一般为 MB	一般为 GB
性能	接近原生	弱于原生
系统支持量	单机支持上千个容器	一般支持几十个容器

如上所述，以 Docker 为主的容器技术逐渐成为边缘计算的技术标准，各大云计算厂商都选择容器技术构建边缘计算平台的底层技术栈。

边缘计算的应用场景非常复杂。从前面的分析可以清晰地看到边缘计算平台并不是传统意义上的只负责数据收集转发的网关。更重要的是，边缘计算平台需要提供智能化运算能力，而且能产生可操作的决策反馈，用来反向控制设备端。过去，这些运算只能在云端完成。现在需要将 Spark、

TensorFlow 等云端的计算框架通过裁剪、合并等简化手段，迁移至边缘计算平台，使得能在边缘计算平台上运行云端训练后的智能分析算法。因此，边缘计算平台需要一种技术在单台计算机或者少数几台计算机组成的小规模集群环境中隔离主机资源，实现分布式计算框架的资源调度。

边缘计算所需的开发工具和编程语言具有多样性。目前，计算机编程技术呈百花齐放的趋势，开发人员运用不同的编程语言处理不同场景的问题已经成为常态，所以在边缘计算平台也需要支持多种开发工具和多种编程语言的运行时环境。因此，在边缘计算平台使用一种运行时环境的隔离技术便成为自然的需求。

容器技术和容器编排技术的逐渐成熟。容器技术是在主机虚拟化技术后最具颠覆性的计算机资源隔离技术。通过容器技术进行资源的隔离，不仅对 CPU、内存和存储的额外开销非常小，而且容器的生命周期管理非常快捷，可以在毫秒级开启和关闭容器。

3.3.3 容器虚拟化

与云计算中使用的主机虚拟机不同，容器技术的初衷是轻量级基于 Linux 操作系统的内核命名空间的资源隔离，用于简化 DevOps 的流程。容器的应用程序共享主机操作系统的内核，并不像虚拟机系统那样完全使用虚拟化隔离。容器虚拟化如图 3-9 所示。

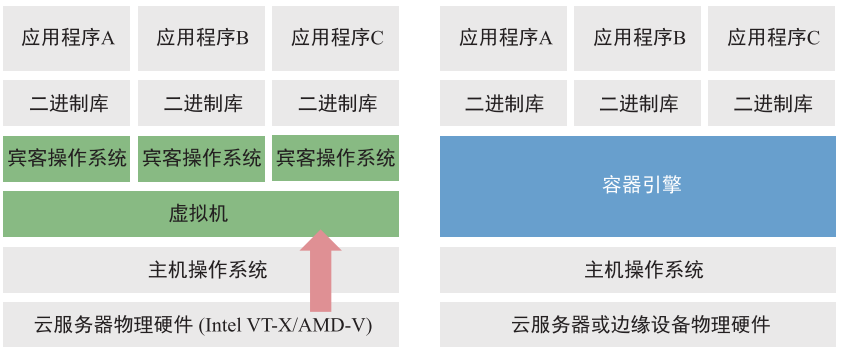

图 3-9 容器虚拟化

随着容器技术的成熟，并逐渐被运用于云端和边缘设备的生产环境中，纯软件的基于内核命名空间的隔离显得不够安全。另外，最初 Docker 技术只是适用于 Linux 系统，而不适用于 Windows，微软也在积极推动 Windows Server 的容器化以及跨操作系统的容器化。于是，结合使用英特尔的 VT-X 和 AMD 的 AMD-V 虚拟化技术和容器管理技术，容器虚拟化技术诞生了。

2015 年，微软和 Docker 公司联合发布了 Windows Server 的 Docker 支持，包括 Hyper-V 容器、NanoServer、最小化的 Windows Server 的 footprint 安装包，针对云环境高度优化，是容器运行的理想环境。

微软全新的容器解决方案实现了资源的隔离。这个领域之前被物理机方案或者虚拟机方案

垄断，同时通过跨平台的 Docker 集成来提供持续的敏捷性和高效性。微软的 Window Server 包括两种容器方式，如图 3-10 所示。

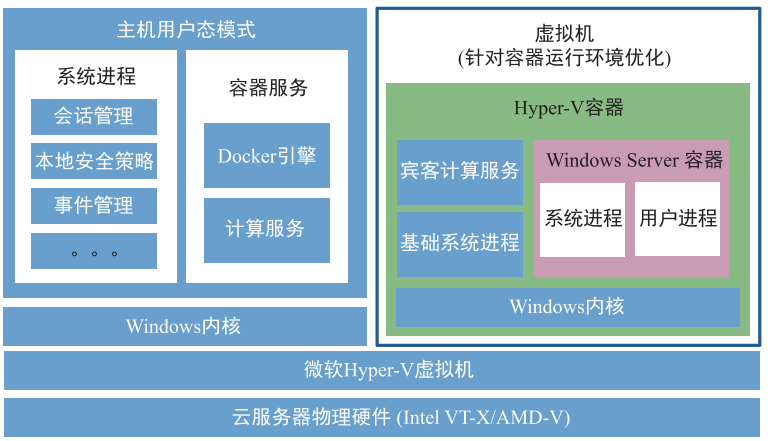

图 3-10　Window Server 的两种容器方式

共用系统核心资源的 Window Server 容器，更像 Linux 上的 Docker 容器。拥有独立系统核心资源的 Hyper-V 容器，更像包装了虚拟机能力的容器。

如图 3-11 所示，OpenStack 基金会发布了开源项目 Kata Containers。该项目立足于英特尔

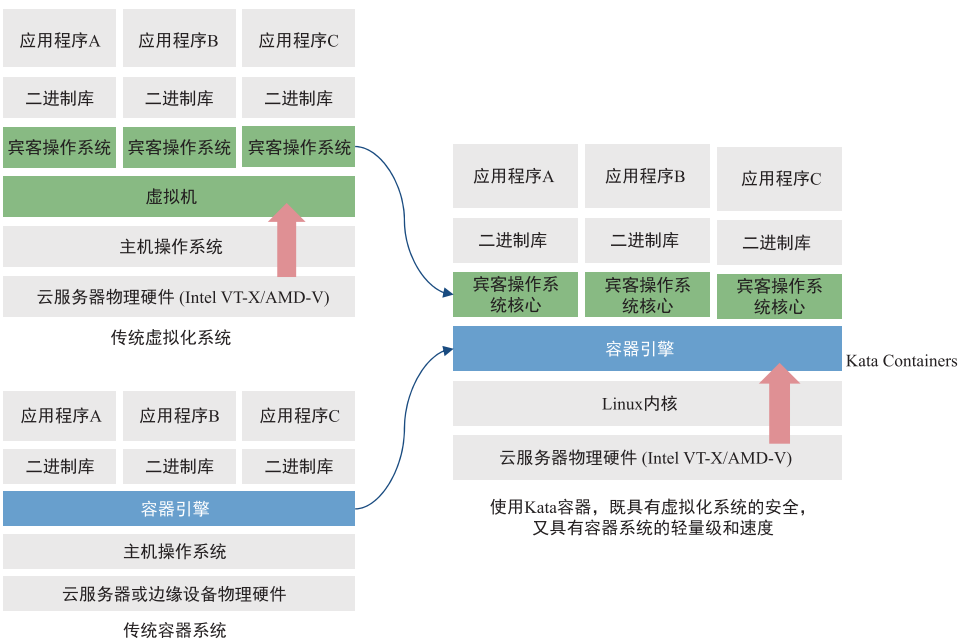

图 3-11　Kata Container 解决方案

贡献的 Intel Clear Containers 技术以及 Hyper 提供的 runV 技术，其目标是将虚拟机（VM）的安全优势与容器的高速及可管理性结合起来，为用户带来最出色的容器解决方案，同时提供强大的虚拟机机制。

Kata Containers 项目最初包括 Agent、Runtime、Shim、Proxy、内核和 QEMU 2.9 六个组件，能够运行在多个虚拟机管理程序上。其优势包括：

- 强大的安全性。Kata Containers 运行在一个优化过的内核之上，基于 Intel VT 技术能够提供针对网络、I/O 和内存等资源硬件级别的安全隔离。
- 良好的兼容性。Kata Containers 兼容主流的容器接口规范，如 Open Container Initiative（OCI）和 Kubernetes Container Runtime Interface（CRI）；同时，也兼容不同架构的硬件平台和不同的虚拟化环境。
- 高效的性能。Kata Containers 优化过的内核可以提供与传统容器技术一样的速度。

3.3.4 容器管理编排和 Kubernetes

1. 容器编排工具的功能

运行一个容器，就像一个乐器单独播放它的交响乐乐谱。容器编排允许指挥家通过管理和塑造整个乐团的声音来统一管弦乐队，提供了有用且功能强大的解决方案，用于跨多个主机协调创建、管理和更新多个容器，具体包括：

（1）部署。这些工具在容器集群中提供或者调度容器，还可以启动容器。在理想情况下，它们会根据用户的需求，例如资源和部署位置，在最佳 VM 中启动容器。

（2）配置脚本。脚本保证把指定的配置加载到容器中，和 Juju Charms、Puppet Manifests 或 Chef recipes 的配置方式一样。通常这些配置用 YAML 或 JSON 编写。

（3）监控。容器管理工具跟踪和监控容器的健康，将容器维持在集群中。在正常工作情况下，监视工具会在容器崩溃时启动一个新实例。如果服务器出现故障，工具会在另一台服务器上重启容器。这些工具还会运行系统健康检查，报告容器不规律行为以及 VM 或服务器的不正常情况。

（4）滚动升级和回滚。当需要部署新版本的容器或者升级容器中的应用时，容器管理工具会自动在集群中更新容器或应用。如果出现问题，它们允许回滚到正确配置的版本。

（5）服务发现。在旧式应用程序中，需要明确指出软件运行所需的每项服务的位置。容器使用服务发现来找到它们的资源。

（6）策略管理。指定容器的运行资源，如 CPU 个数、内存大小等。

（7）互操作。容器管理编排工具需要和容器以及容器运行时相兼容。