第 1 章 Chapter 1

OpenStack 参考架构设计

云计算改变了企业 IT 服务的运行方式。基于不同的资源使用方式，云计算解决方案可分为私有云、公有云、混合云和社区云。很多组织机构已经迫切地感受到，不论使用哪种云解决方案，都需要在基础架构中引入编排引擎，以便更好地拥抱弹性和扩展性，以及获取极佳的用户体验。当下，OpenStack 这个私有云领域热门的编排解决方案，已让成千上万的企业进入了下一个数据中心时代。在撰写本书时，OpenStack 已经部署在许多大中型企业基础架构中，在各种生产环境中运行着不同类型的工作负载。由于多数 IT 巨头的支持以及遍布全球的庞大开发者社区的共同努力，OpenStack 的成熟度在不断提升。OpenStack 每个新版本都会带来很多出色的新功能。对很多企业而言，拥抱 OpenStack 无疑是最佳选择，因为 OpenStack 不仅可以更好地承载业务负载，还为企业带来了灵活的基础架构设施。

在本书第 2 版中，我们将继续介绍 OpenStack 最新发行版本中的各种新特性，并讨论 OpenStack 在提供极佳云体验方面存在的巨大潜力。

部署 OpenStack 仍然是一个极具挑战的事情。在部署之前，需要我们很好地理解与 OpenStack 有关的自动化、编排和灵活性等概念和知识点。如果对 OpenStack 有一个恰当而合适的期望，那么你会发现挑战将变成机会，值得你投入精力去努力。

在基础架构需求收集完成后，正式启动 OpenStack 项目之前，你需要一个完整的设计方案和基于不同基础架构设备的部署计划。

日本兵法家宫本武藏（Miyamoto Musashi）在 *Start Publishing LLC* 出版的《五轮书》（*The Book of Five Rings*）中写道：

> “在策略上，以近在咫尺的心态看待遥远的事物，而以宏观视角来考虑身旁事物，是非常重要的。”

我们的 OpenStack 之旅将从以下几方面的内容开始：

- ❑ 回顾 OpenStack 组件，熟悉 OpenStack 生态系统逻辑架构体系。
- ❑ 为恰当的环境选择适当的 OpenStack 核心服务，以便学习如何设计 OpenStack。
- ❑ 介绍为扩大 OpenStack 生态系统而在最新稳定版本中引入的新项目。
- ❑ 为大规模部署环境设计 OpenStack 初始架构。
- ❑ 通过梳理初次部署中的最佳实践，规划未来可能的增长所需的扩容。

1.1 OpenStack 引领新一代数据中心

云计算提供了各种类型的基础设施服务，例如**软件即服务**（SaaS）、**平台即服务**（PaaS）和**基础设施即服务**（IaaS）。公有云提供了敏捷、速度和自助服务。大多数公司都自建有昂贵的 IT 系统，多年来一直在持续开发和部署这些系统，然而它们是孤立的，且经常需要进行人为干预。在很多时候，企业 IT 系统一直在为匹配公有云服务的敏捷和速度而苦苦挣扎。在当今的敏捷服务交付环境中，传统数据中心及其孤立的基础架构并不具备可持续性。事实上，当今的企业数据中心必须专注于快速、灵活和自动化地提供服务，才可能成为高效的下一代数据中心。

软件定义基础架构的重大转变，使得运维管理人员在几分钟内即可提供完全自动化的基础架构。下一代数据中心将基础架构简化为单一、大型、灵活、可扩展和自动化的单元，最终结果是得到了可编程、可扩展和多租户感知的基础架构。这也正是 OpenStack 的发展方向：为下一代数据中心操作系统赋能。事实上，很多大型跨国企业（如 VMware、思科、瞻博网络、IBM、Red Hat、Rackspace、PayPal 和 eBay）都已感受到 OpenStack 无处不在的影响。如今，它们中的多家企业都在其生产环境中运行着基于 OpenStack 的大规模可扩展私有云。如果想要成为云计算行业里的创新企业，那么在 IT 基础架构中采用 OpenStack 将是转向下一代数据中心并获取宝贵云计算经验的最佳选择。

要了解更多有关许多公司成功案例的信息，请访问 https://www.openstack.org/user-stories。

1.2 OpenStack 逻辑架构介绍

在深入研究 OpenStack 架构之前，我们首先需要了解 OpenStack 的基础知识，并理解 OpenStack 每个核心组件的基本概念及其用法。

为了更好地理解 OpenStack 的工作原理，对 OpenStack 组件进行简单的剖析是非常有必要的。在下文中，我们将介绍各种 OpenStack 服务，这些服务协同工作以向终端用户提供各种云计算功能体验。虽然不同的 OpenStack 服务用来满足不同的需求，但是它们的设计都遵

循共同的思想，可归纳如下：

- 大多数 OpenStack 服务都采用 Python 语言进行开发，这极大地提升了开发速度。
- 所有 OpenStack 服务都提供 REST API。这些 API 是 OpenStack 服务的主要外部访问接口，由其他服务或最终用户使用。
- OpenStack 服务本身由不同组件实现。服务组件通过消息队列相互通信。消息队列具有很多优点，例如在多个工作守护进程之间实现请求排队、松散耦合和负载均衡。

基于这种通用设计思想，我们现在来对 OpenStack 各核心组件进行仔细分析。对每个组件，我们要问的第一个问题将是：这个组件是干什么的？

1.2.1　认证管理服务 Keystone

从架构角度来看，Keystone 提供了 OpenStack 体系中最简单的服务。它是 OpenStack 的核心组件之一，提供身份认证服务，包括 OpenStack 中租户的身份验证和授权。不同 OpenStack 服务之间的通信都必须经过 Keystone 认证，以确保授权的用户或服务能够访问所请求的 OpenStack 服务对象。Keystone 集成了许多身份验证机制，如基于用户名 / 密码和令牌 / 身份的验证系统。此外，可以将其与现有后端集成，例如**轻量级目录访问协议**（Lightweight Directory Access Protocol，LDAP）和**插拔式验证模块**（Pluggable Authentication Module，PAM）。

Keystone 还提供服务目录（service catalog）作为所有 OpenStack 服务的注册中心。

随着 Keystone 的发展，借助中心化的和联邦的身份解决方案，很多新功能在最新的 OpenStack 发行版本中得以实现。这使得用户可使用已有的、中心化的后端身份验证登录机制，同时可将身份验证机制与 Keystone 解耦。

联邦身份验证解决方案在 OpenStack Juno 版本中已变得更加稳定，这种方案使 Keystone 成为一个**服务供应商**（Service Provider，SP），并使用可信的**身份提供者**（Provider of Identity，IdP）、SAML 断言中的用户身份信息或 OpenID Connect 声明。IdP 可以由 LDAP、活动目录（Active Directory）或 SQL 支持。

1.2.2　对象存储服务 Swift

Swift 是 OpenStack 用户可以使用的一种存储服务。它通过 REST API 提供对象存储功能。与传统存储解决方案（如文件共享存储或基于块的存储）相比，对象存储采用对象方式来处理所存储的数据，并从对象存储中存储和检索对象。我们来抽象地概况一下对象存储。为了存储数据，对象存储将数据拆分为较小的块并将其存储在独立容器中。这些保存数据的容器分布在存储集群节点上，并拥有冗余副本，以提供高可用性、自动恢复能力和水平可伸缩性。

稍后我们会讨论 Swift 对象存储的架构。简单来说，对象存储有以下几个优点：

去中心化，无**单点故障**（Single Point of Failure，SPOF）。

具有自愈能力，这意味着发生故障时能自动恢复。

通过水平扩展可以将存储空间扩展到 PB 级以上。

具有高性能，通过在多个存储节点上分散负载实现。

支持廉价硬件，这些硬件可用于冗余存储集群。

1.2.3 块存储服务 Cinder

你可能会好奇，OpenStack 中是否还有另外的存储服务。事实上，通过 Cinder 服务，OpenStack 提供了持久性块存储服务。它的主要功能是为虚拟机提供块级存储。Cinder 为虚拟机提供了可用作存储磁盘的裸卷。

Cinder 提供的一些特性如下。

- **卷管理**：允许创建或删除卷。
- **快照管理**：允许创建或删除卷的快照。
- 将卷挂载到实例，或将卷从实例上分离。
- 克隆卷。
- 从快照创建卷。
- 从镜像创建卷，以及从卷创建镜像。

需要记住的是，像 Keystone 服务一样，Cinder 支持多个供应商（如 IBM、NetApp、Nexenta 和 VMware）的存储产品驱动插件，从而可以使用不同存储供应商的存储设备作为 Cinder 后端存储。

Cinder 已被证明是一种理想的解决方案，在架构层面上，Cinder 服务替代了 Folsom 版本之前的 nova-volume。更重要的是，Cinder 已经组织并创建了具有多种不同特征的基于块的存储设备的列表。但是，我们必须明确考虑商业存储的限制，例如冗余和自动伸缩。

在 OpenStack Grizzly 版本中，Cinder 实现了一个新功能，允许为 Cinder 卷创建备份（backup）。一个常见的用例是将 Swift 作为备份存储解决方案。在接下来的几个版本中，Cinder 增加了更多备份存储，例如 NFS、Ceph、GlusterFS、POSIX 文件系统和 IBM 备份解决方案 Tivoli Storage Manager。这种出色的备份可扩展功能由 Cinder 备份驱动插件所定义，在每个新版本中都会实现更加丰富的驱动插件。在 OpenStack Mitaka 版本中，Cinder 进一步丰富了其备份方案，引入了一种新的备份驱动插件，能够将卷备份保存到**谷歌云平台**（Google Cloud Platform，GCP）上，这样它就将两种不同类型的云环境连接起来了。这一解决方案使得 OpenStack 实现了混合云备份解决方案，为持久数据提供了灾难恢复策略。这么做安全吗？从 Kilo 版本开始，这个问题已得到圆满解决，因为在开始任何备份操作之前，Cinder 卷都会被加密。

1.2.4　文件共享存储服务 Manila

除了上一节中讨论的块和对象存储之外，从 Juno 版本开始，OpenStack 还提供了一个名为 Manila 的基于文件共享的存储服务。它实现了远程文件系统存储。在使用时，它类似于我们在 Linux 上使用的**网络文件系统**（Network File System，NFS）或 SAMBA 存储服务，而 Cinder 服务则类似于**存储区域网络**（Storage Area Network，SAN）服务。实际上，可以使用 NFS 和 SAMBA 或**通用网络文件系统**（Common Internet File System，CIFS）作为 Manila 服务的后端驱动插件。Manila 服务能在共享服务器上编排文件共享（file share）。

有关存储服务的更多细节将在第 5 章中讨论。

OpenStack 中的每个存储解决方案都是针对特定目的而设计的，并针对不同的目标进行实现。在做出任何架构设计决策之前，了解 OpenStack 中现有存储选项之间的差异至关重要，如表 1-1 所示。

表 1-1　OpenStack 不同存储服务功能对比

存储类型 / 规范指标	Swift	Cinder	Manila
访问方式	通过 REST API 访问对象	用作块存储设备	以文件方式访问
支持多访问（multi-access）	OK	不支持，只能由一个客户端使用	OK
持久性	OK	OK	OK
可访问性	任何位置	由单个虚拟机使用	被多个虚拟机同时使用
性能	OK	OK	OK

1.2.5　镜像注册服务 Glance

Glance 服务提供了镜像和元数据的注册服务，OpenStack 用户可通过镜像来启动虚拟机。Glance 支持各种镜像格式，用户可以根据虚拟化引擎选择使用，Glance 支持 KVM/Qemu、XEN、VMware、Docker 等镜像。

如果你是 OpenStack 新用户，你可能会问，Glance 和 Swift 有什么不同？两者都有存储功能。它们之间的区别是什么？为什么我需要集成这两个方案呢？

Swift 是一个存储系统，而 Glance 是镜像注册服务（image registry）。两者之间的区别在于，Glance 保持对虚拟机镜像和有关镜像元数据的跟踪。元数据可以是内核、磁盘镜像、磁盘格式等信息，Glance 通过 REST API 向 OpenStack 用户提供此信息。Glance 可以使用各种后端来存储镜像，默认使用文件目录，但在大规模生产环境中，可以使用其他方案进行镜像存储，如 NFS 或者 Swift。

相比之下，Swift 是一个纯粹的存储系统。它专门为对象存储而设计，你可以在其中保存虚拟磁盘、镜像、备份归档等数据。

Glance 的目的是提供镜像注册机制。从架构层次来看，Glance 的目标是专注于通过镜像服务 API 来存储和查询镜像信息。Glance 的典型用例是允许客户端（可以是用户或外部服务）注册新的虚拟磁盘镜像，而存储系统则专注于提供可高度扩展和具备冗余的数据存储。在这个层面上，作为技术运维人员，你所面临的挑战是提供合适的存储解决方案，以兼顾成本和性能要求，这将在本书的后续章节中讨论。

1.2.6 计算服务 Nova

或许你已经知道，Nova 是 OpenStack 中原始和核心的组件。从架构层面来看，它也被公认为是 OpenStack 最复杂的组件之一。Nova 在 OpenStack 中提供计算服务，并管理虚拟机以响应 OpenStack 用户提出的服务请求。

Nova 项目的作用是与大量其他 OpenStack 服务和内部组件进行交互，Nova 必须与各个服务组件相互协作，以响应用户运行虚拟机的请求。

接下来，我们将对 Nova 服务进行剖析。从架构层面来看，Nova 本质上是一个分布式应用程序，用于在不同组件之间进行调度编排以执行任务。

1. nova-api

nova-api 组件接受并响应终端用户的计算服务 API 调用请求。终端用户或其他组件与 OpenStack nova-api 接口通信，以通过 OpenStack API 或 EC2 API 创建实例。

nova-api 启动大多数编排活动，例如运行实例或执行某些特定策略。

2. nova-compute

nova-compute 组件本质上是一个守护进程，其通过虚拟化引擎的 API 接口（XenServer 的 XenAPI、Libvirt 的 KVM 和 VMware 的 VMware API）创建和终止虚拟机（Virtual Machine，VM）实例。

3. nova-network

nova-network 组件从队列中获取网络任务，然后执行这些任务来操作网络（例如设置桥接接口或更改 IP 表规则）。

Neutron 可替代 nova-network 服务。

4. nova-scheduler

nova-scheduler 组件从队列中获取 VM 实例的请求，并确定它应该在哪里运行（具体来说应该运行在哪台计算节点主机上）。在应用架构级别，术语调度（scheduling）或调度程序（scheduler）是指在既定基础架构中，通过系统性搜索算法来找到最佳放置点以提升其性能。

5. nova-conductor

nova-conductor 服务向计算节点提供数据库访问。其出发点是阻止从计算节点直接访问数据库，从而在某个计算节点受到威胁时可增强数据库安全性。

仔细观察 OpenStack 通用组件，我们会发现 Nova 会与几个服务进行交互，如用于身份验证的 Keystone、用于镜像注册的 Glance 和实现 Web 接口界面的 Horizon。其中一个关键的交互是与 Glance 服务的交互，Nova API 进程会将任何与镜像查询有关的请求转发至 Glance，而 nova-compute 会下载镜像以启动实例。

Nova 还提供控制台服务（console service），允许终端用户通过代理（如 nova-console、nova-novncproxy 和 nova-consoleauth）访问虚拟机实例控制台。

1.2.7　网络服务 Neutron

Neutron 为 OpenStack 服务（如 Nova）管理的接口设备提供真正的**网络即服务**（Network as a Service，NaaS）功能。Neutron 具备下述特性：

- 允许用户创建自己的网络，然后将虚拟机接口关联到网络上。
- 可插拔的后端架构设计使得用户可以利用各种通用商业设备或供应商特定的设备。
- 提供扩展功能，允许集成其他网络服务。

Neutron 还有很多新功能在不断成熟和发展的过程中。其中的一些新功能用于实现路由器、虚拟交换机和 SDN 网络控制器。Neutron 包含以下核心资源：

- **端口**（port）：Neutron 中的端口可看成是与虚拟交换机的一种连接方式。这些连接将实例和网络连接在一起。当实例连接到子网时，已定义了 MAC 和 IP 地址的接口将被插入子网中。
- **网络**（network）：Neutron 将网络定义为隔离的第 2 层网段。运维人员将网络视为由 Linux 桥接工具、Open vSwitch 或其他虚拟交换机软件实现的逻辑交换机。与物理网络不同，OpenStack 中的运维人员和用户都可以定义网络。
- **子网**（subnet）：Neutron 中的一个子网代表与某个网络相关联的一个 IP 地址段。该段的 IP 地址将被分配给端口。

Neutron 还提供了额外的扩展资源，以下是一些常用的扩展：

- **路由器**（router）：路由器提供网络之间的网关。
- **私有 IP**（private IP）：Neutron 定义了如下两种类型的网络。
  - **租户网络**（tenant network）：租户网络使用私有 IP 地址。私有 IP 地址在实例中可见，这允许租户在实例之间进行通信，同时保持与其他租户流量的隔离。私有 IP 地址对 Internet 不可见。
  - **外部网络**（external network）：外部网络是可见的，可从 Internet 上路由。它们必须使用可路由的子网段。

- **浮动 IP**（floating IP）：浮动 IP 是外部网络上分配的 IP 地址，Neutron 把它映射到实例的私有 IP。浮动 IP 地址分配给实例，以便它们能够连接到外部网络并访问 Internet。Neutron 通过使用**网络地址转换**（Network Address Translation，NAT）实现浮动 IP 到实例私有 IP 的映射。

Neutron 还提供了其他高级服务来实现 OpenStack 网络相关的功能，如下所示：

- **负载均衡即服务**（Load Balancing as a Service，LBaaS），用于在多个计算实例之间分配流量。
- **防火墙即服务**（Firewall as a Service，FWaaS），用于保护第 3 层和第 4 层网络边界访问。
- **虚拟专用网络即服务**（Virtual Private Network as a Service，VPNaaS），用于在实例或主机之间构建安全隧道。

可以参考最新的 Mitaka 版本文档，了解有关 OpenStack 网络的详细信息，请访问 http://docs.openstack.org/mitaka/networking-guide/。

Neutron 架构中的三个主要组件是：

- **Neutron 服务器**（Neutron server）：它接受 API 请求并将它们分配到适当的 Neutron 插件以便采取进一步响应。
- **Neutron 插件**（Neutron plugin）：它负责编排后端设备，例如插入或拔出端口、创建网络和子网或分配 IP 地址。
- **Neutron 代理**（Neutron agent）：Neutron 代理在计算节点和网络节点上运行。代理程序从 Neutron 服务器上的插件接收命令，并使更改在各个计算或网络节点上生效。不同类型的 Neutron 代理实现不同的功能。例如，Open vSwitch 代理通过在 Open vSwitch（OVS）网桥上插入和拔出端口来实现 L2 连接，它们在计算和网络节点上运行，而 L3 代理仅在网络节点上运行并提供路由和 NAT 服务。

代理和插件因特定云供应商所采用的技术而异，采用的网络技术可能是虚拟或物理 Cisco 交换机、NEC、OpenFlow、OpenSwitch 和 Linux 桥接等。

Neutron 是一种管理 OpenStack 实例之间网络连接的服务，它确保网络不会成为云部署中的瓶颈或限制因素，并为用户提供真正的网络自助服务，这类自助服务也包括网络配置。

Neutron 的另一个优势在于它能够实现不同供应商网络解决方案的集成，并提供一种灵活的网络扩展方式。Neutron 旨在提供插件和扩展机制，为网络管理员提供了通过 Neutron API 启用不同网络技术的选择。有关这一点的更多细节将在第 6 章和第 7 章中进行介绍。

请记住，Neutron 允许用户管理和创建网络，以及将服务器和节点连接到各种网络。

Neutron 的可扩展性优势将在后续**软件定义网络**（Software Defined Network，SDN）和**网络功能虚拟化**（Network Function Virtualization，NFV）主题中继续讨论，这些技术对于寻求高级网络和多租户的很多网络管理员而言是非常具有吸引力的。

1.2.8　计量服务 Telemetry

Telemetry 在 OpenStack 中提供计量服务。在 OpenStack 共享的多租户环境中，计量租户的资源使用率是非常重要的。

Ceilometer 是 Telemetry 的主要组件之一，它负责收集与资源相关的数据。资源可以是 OpenStack 云中的任何实体，例如虚拟机、磁盘、网络和路由器等。资源与计量指标关联在一起。与租户相关的资源使用数据通过计量指标定义的单位进行采样，并存储到相应的数据库中。Ceilometer 内置了资源使用数据的汇总功能。

Ceilometer 从各种数据源收集数据，例如消息总线、轮询资源和集中代理程序等。

Liberty 版本发行后，OpenStack 中 Telemetry 服务在设计上出现了变化，即**报警**（Alarming）服务从 Ceilometer 项目中被剥离出去，并重新孵化了一个名为 Aodh 的新项目。Telemetry 的报警服务专门用于管理报警，并根据收集到的计量数据和预定事件触发报警。

采用时序数据库 Gnoochi，Telemetry 服务得到了很多增强，它可应对大规模指标和事件存储带来的挑战，并可提高其性能。第 10 章中将会更详细地介绍 Telemetry 服务和系统监控。

1.2.9　编排服务 Heat

在 Havana 版本中，OpenStack 首次发行了编排项目 Heat。Heat 最初是针对部分 OpenStack 资源的编排进行开发的，包括计算、镜像、块存储和网络服务。Heat 通过编排不同的云资源，改善 OpenStack 中的资源管理方式。它是通过创建堆栈来编排资源的，最终用户只需点击执行按钮即可运行应用程序。使用简单的模板引擎文本文件——在 OpenStack 中称为 **HOT 模板**（Heat Orchestration Template），用户即可立即创建所需资源并运行应用程序。在最新的 OpenStack 版本中，Heat 成熟度和支持的资源目录在不断增加，因此 Heat 正在成为一个极具吸引力的 OpenStack 项目。基于 Heat 项目，社区已孵化了另外的 OpenStack 项目，如 Sahara（大数据即服务），Sahara 即是使用 Heat 引擎来编排底层资源堆栈创建的项目。Heat 正成长为 OpenStack 中的一个成熟组件，并可以与一些系统配置管理工具集成，如 Chef，从而实现全栈自动化和配置设置。

Heat 模板文件使用的是 YAML 或 JSON 格式，因此代码行的缩进很重要！

OpenStack 中的编排项目将在第 8 章中详细介绍。

1.2.10 仪表盘服务 Horizon

Horizon 是 OpenStack 中的 Web 控制面板项目，它将 OpenStack 生态系统中的全部组件整合在一起进行展示。

Horizon 为 OpenStack 服务提供了 Web 前端。目前，它实现了所有 OpenStack 服务以及一些孵化项目的可视化。Horizon 被设计为无状态和无数据的 Web 应用程序，它只不过是将用户请求转化为对 OpenStack 内部服务的 API 调用，并将 OpenStack 的调用结果返回给 Horizon 进行显示。除了会话信息之外，Horizon 不会保留任何额外数据。Horizon 是一种参考架构设计，云管理员可根据需求对其进行定制和扩展。部分公有云的 Web 界面正是基于 Horizon 构建的，其中比较著名的就是惠普公有云，核心思想就是通过 Horizon 的扩展模块来进行构建。

Horizon 由一系列**面板模块**（panel）构成，这些面板模块定义了每个服务的交互。根据特定需求，用户可以启用或禁用这些模块。除了具备灵活性的功能，Horizon 还具有**层叠样式表**（Cascading Style Sheet，CSS）风格。

1.2.11 消息队列

消息队列（Message Queue，MQ）提供了一个中心化的消息交换器，用于在不同服务组件之间传递消息。MQ 是不同守护进程之间共享信息的地方，这些进程以异步方式实现消息互通。队列系统的主要优势就是它可以缓冲请求，并向消息订阅者提供单播和基于组的通信服务。

1.2.12 数据库

数据库存储了云基础架构中大多数的构建时和运行时状态，包括可供使用的实例类型、正在使用的实例、可用网络和项目等。数据库提供了以上信息的持久存储，以便保留云基础架构的状态。数据库是所有 OpenStack 组件进行信息共享的重要组件。

1.3 资源准备与虚拟机创建

1.3.1 准备虚拟机资源

下面，我们通过一系列步骤将前面介绍的全部核心服务串联到一起，进而了解 OpenStack 的工作原理。

（1）第一步要执行的操作是身份验证。这是 Keystone 要做的事情。Keystone 根据用户名和密码等凭据对用户进行身份验证。

（2）Keystone 提供服务目录，其中包含有关 OpenStack 服务和 API 端点的信息。

（3）可以使用 Openstack CLI 获取服务目录：

```
$ openstack catalog list
```

服务目录是一种 JSON 结构，它列出了授权令牌请求上全部可用资源。

（4）通常，通过身份验证后，即可与 API 节点通信。OpenStack 生态系统中有不同的 API（OpenStack API 和 EC2 API）。图 1-1 显示了 OpenStack 工作原理的抽象视图。

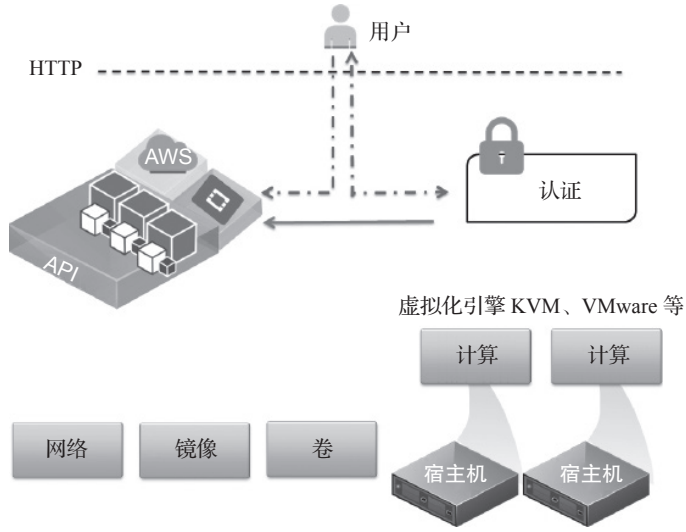

图 1-1　OpenStack 高级视图

（5）OpenStack 架构设计中的另一个关键点是调度器。调度器由基于工作守护进程构建的 OpenStack 服务实现。工作守护进程管理各个节点上的实例启动，并跟踪运行实例物理节点上的可用资源。OpenStack 服务中的调度器会查看物理节点上的资源状态（由工作守护程序提供），并从候选节点中选取一个最佳节点用于启动虚拟机。这种设计架构的一个实现就是 nova-scheduler。nova-scheduler 将选择合适的计算节点来运行虚拟机，而 Neutron L3 调度程序将决定在哪个 L3 网络节点上运行虚拟路由器。

OpenStack Nova 中的调度过程可以执行不同的算法，例如简单算法（simple）、机会算法（chance）和可用区算法（zone）。一种高级实现方法是利用权重计算（weighting）和过滤器（filter）对服务器进行排序。

1.3.2　虚拟机创建流程

理解 OpenStack 中的不同服务是如何协同工作以提供虚拟机服务是非常重要的。前文中，我们已经看到 OpenStack 中的请求是如何通过 API 进行处理的。接下来，通过图 1-2，我们来描述 OpenStack 虚拟机创建的整个流程。

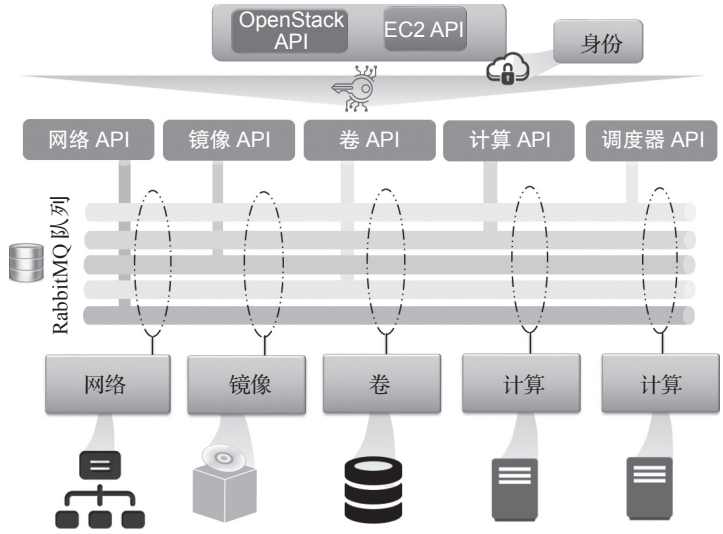

图 1-2　OpenStack 创建虚拟机 API 请求处理流程

虚拟机的启动过程涉及构建实例的多个 OpenStack 核心服务之间的交互，包括计算、网络、存储和基础镜像。如图 1-2 所示，OpenStack 服务通过在消息队列总线中提交和提取 RPC 调用来实现彼此之间的交互。虚拟机创建过程中每个步骤的信息都由 OpenStack 服务通过消息总线进行验证和传递。从架构设计角度来看，子系统调用是在 OpenStack API 端点（endpoint）包括 Nova、Glance、Cinder 和 Neutron 中定义和处理的。

另一方面，OpenStack 中 API 的相互通信需要一种可信的身份验证机制，这由 Keystone 实现。

从身份认证服务开始，以下步骤简要总结了基于 OpenStack 中 API 调用的虚拟机创建流程：

- ❑ 调用身份认证服务进行身份验证。
- ❑ 生成用于后续调用的令牌。
- ❑ 访问镜像服务以获取镜像列表，并获取目标基础镜像。
- ❑ 处理计算服务 API 请求。
- ❑ 处理计算服务对安全组和密钥的调用请求。
- ❑ 调用网络服务 API 确定可用网络。
- ❑ 通过计算节点调度程序选择 Hypervisor 节点。
- ❑ 调用块存储服务 API 为实例分配卷。
- ❑ 通过计算服务 API 调用在 Hypervisor 节点启动实例。
- ❑ 调用网络服务 API 为实例分配网络资源。

需要记住的是，在每个 API 调用和服务请求中，OpenStack 中的令牌是有时间限制的。OpenStack 中创建操作失败的主要原因之一是在后续 API 调用期间令牌失效了。此外，令牌

管理方式在不同的 OpenStack 版本中可能不同。在 Liberty 发布之前，OpenStack 中使用以下两种令牌管理方法：

- UUID（Universally Unique Identifier）：在 Keystone V2 中，将会生成 UUID 令牌，并在客户端 API 调用之间互传，并且需要返回给 Keystone 进行验证。事实证明，UUID 类型的令牌管理方式导致了 Keystone 的性能问题。
- PKI（Public Key Infurastructure）：在 Keystone V3 中，Keystone 在每次 API 调用时都不再验证令牌。各 API 端点可通过检查最初生成令牌时添加的 Keystone 签名来验证令牌。

从 Kilo 版本开始，Keystone 中的令牌引入了更复杂的加密身份验证令牌方法，如 Fernet，从而获得了进一步增强。基于 Fernet 的令牌有助于解决 UUID 和 PKI 令牌引起的性能问题。Mitaka 版本完全支持 Fernet，社区正在推动将其作为默认版本。此外，PKI 令牌在以后的 Kilo 版本中将被弃用，默认并且推荐使用 Fernet 令牌。

第 3 章简要介绍了 Keystone 中引入的有关新增功能的更多高级主题。

1.4　OpenStack 逻辑概念设计

OpenStack 的部署基于前面介绍的各个组件。部署是对你是否已完全理解 OpenStack 集群环境架构设计的一种确认。当然，考虑到这种云管理平台的多功能性和灵活性，OpenStack 提供了多种可能性，而这正是 OpenStack 的一种优势。但是，也正是由于这种灵活性，要实现一个满足自身需求的架构设计，反而不是一件轻松的事情。

最终的一切，都要归结到你所提供的云服务上去。

大多数企业通过三个阶段来成功地设计它们的 OpenStack 环境：从概念模型设计，到逻辑模型设计，再到物理设计。很明显，从概念设计到逻辑设计以及从逻辑设计到物理设计，复杂性都在增加。

1.4.1　概念模型设计

作为第一个概念设计阶段，我们对于自己所需要的 OpenStack 功能类型或服务项目要有一个较高层次的定位，表 1-2 是 OpenStack 主要功能项目的划分和角色定位。

表 1-2　OpenStack 功能项目的划分与角色定位

功能类别	角色
计算（Compute）	存储虚拟机镜像 提供用户接口
镜像（Image）	存储磁盘文件 提供用户接口
对象存储（Object Storage）	存储对象 提供用户接口

（续）

功能类别	角色
块存储（Block Storage）	提供卷 提供用户接口
网络（Network）	提供网络连接 提供用户接口
计量（Telemetry）	提供度量、指标和报警 提供用户接口
文件共享（File Share）	为 OpenStack 提供横向扩展的文件共享系统 提供用户接口
身份认证（Identity）	提供认证
仪表盘（Dashboard）	提供图形化用户界面
编排（Orchestration）	为堆栈创建提供编排引擎 提供用户接口

我们可以将表 1-2 中的功能组件映射到图 1-3 中。

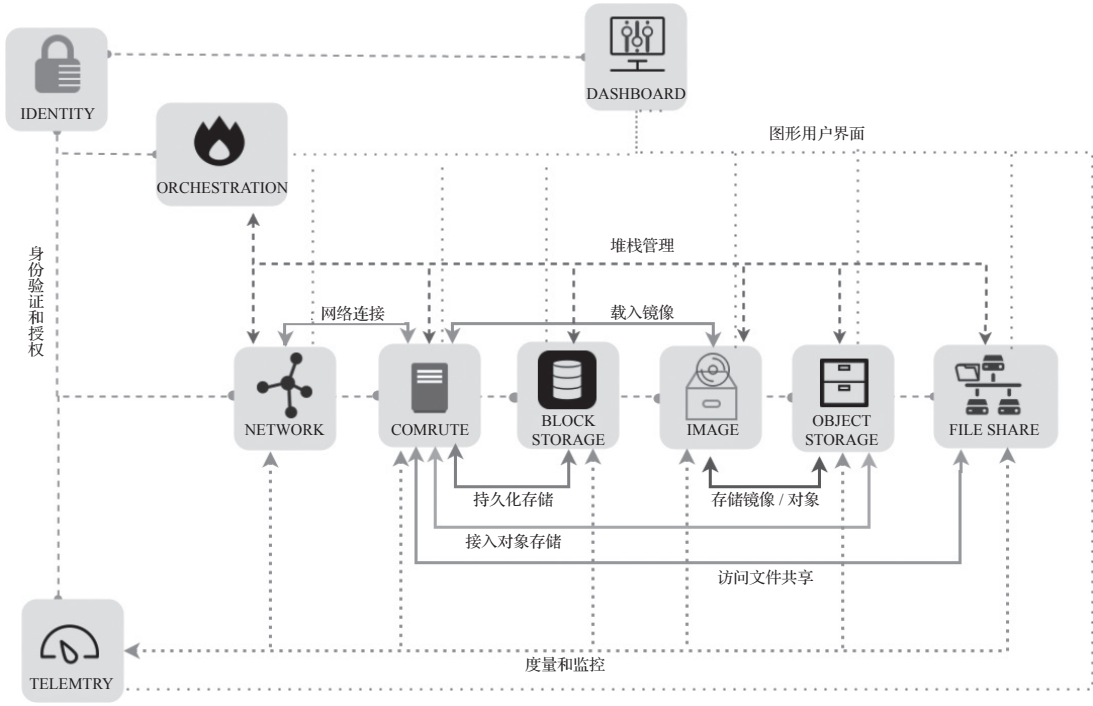

图 1-3　OpenStack 服务类别

请记住，图 1-3 可能会被反复定义和修改，因为我们的目标是在架构设计中集成更多的 OpenStack 服务。换句话说，我们遵循渐进式的设计方法，并且充分利用 OpenStack 架构设计的灵活性。

就这个层面而言，我们无须担心细节，只要找准主要目标的愿景和方向即可。

1.4.2　逻辑模型设计

基于对概念设计的反复思考，你很可能会对 OpenStack 的不同核心组件又有了新的想法，这将进一步推动逻辑设计。

首先，我们需要梳理清楚 OpenStack 核心服务之间的依赖和相关关系。在本节中，我们将探索 OpenStack 的部署架构。首先确定运行 OpenStack 服务的节点：云控制器、网络节点和计算节点。你可能会好奇为什么这里需要考虑物理设计上的分类。实际上，将云控制器和计算节点看成封装了大量 OpenStack 服务的简单软件包将有助于完善早期阶段的设计。此外，此方法还有助于提前一步规划高可用性和伸缩性需求，后面会对它们进行更详细的介绍。

第 3 章深入介绍如何在云控制器和计算节点之间分布各种 OpenStack 服务。

基于先前理论阶段的工作，通过为设计模型分配参数和值，将进一步细化物理模型设计。图 1-4 是逻辑设计模型的第一次迭代。

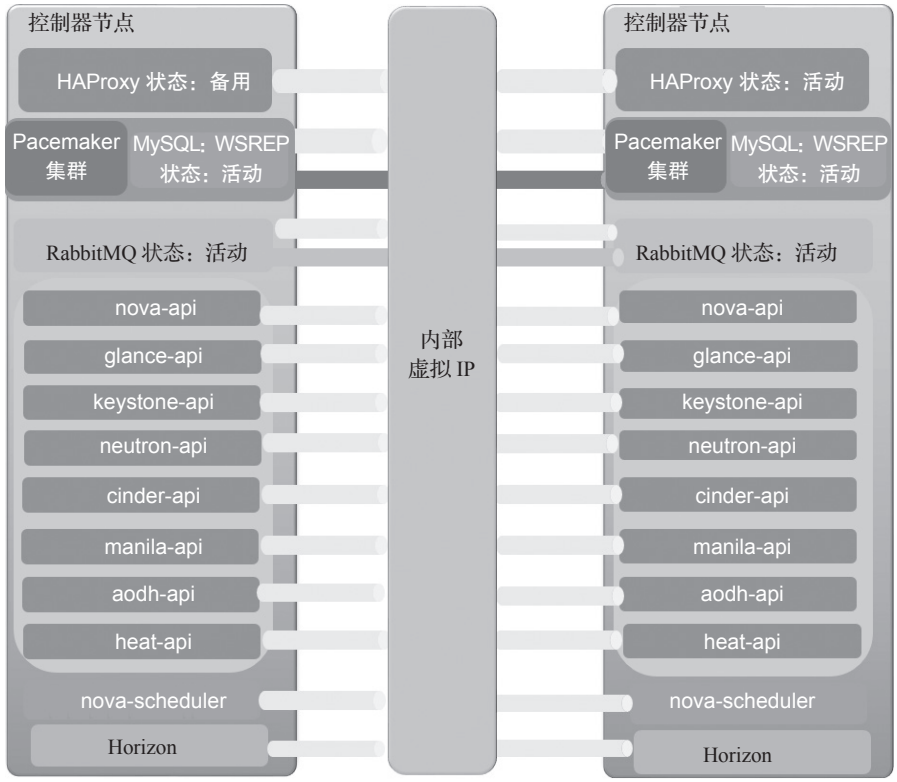

图 1-4　OpenStack 逻辑设计

很显然，在高可用集群配置中，我们应该为 OpenStack 中的每个服务实现一定程度的冗余。你可能会好奇本章上半部分介绍的 OpenStack 关键组件，即数据库和消息队列，为什么不在独立集群中部署或隔离封装起来？这是一个好问题。请记住，我们目前处在第二个逻辑设计阶段，我们是在没有更多细节信息的情况下来试图慢慢深入了解 OpenStack 基础架构。另外，我们将会从一个简单、通用的设计架构转向针对特定用例的设计。就现阶段而言，将 RabbitMQ 和 MySQL 等基础设施组件与 OpenStack 服务分离，就意味着直接跳过初级简单设计阶段了。

什么是高可用（High Availability）？

图 1-4 描述了实现 OpenStack 高扩展性和冗余性的几个基本解决方案，如**虚拟 IP**（VIP）、HAProxy 和 Pacemaker。这些高可用技术将在第 9 章中进行更详细的讨论。

计算节点相对简单，因为它们仅用于运行虚拟机。为了管理虚拟机，nova-compute 服务将运行在每个计算节点上。此外，计算节点并非孤立节点，Neutron 网络代理和可选的 Ceilometer 计算代理都可以运行在这些计算节点上。

网络节点将运行 Neutron 的 DHCP 和 L3 代理。

1.4.3 存储选型考虑

通过前面的介绍，现在你应该对 Swift、Cinder 和 Manila 这几种存储类型有了更深入的了解。

事实上，到目前为止，我们还未正式介绍基于第三方的软件定义存储、Swift 和 Cinder 相关的详细内容。

更多的存储相关细节，将在后续第 5 章中进行介绍。目前，我们将基于前期对 Cinder、Manila 和 Swift 的理解，来决定如何在我们的逻辑设计中应用这些存储模块。

在存储设计时，你需要自问一些问题，例如，我有多少数据需要存储？我未来的应用场景中是否会存在运行高负荷数据分析的应用程序？对于虚拟机快照备份的存储要求是什么？我真的需要在存储上管理文件系统吗？还是仅需文件共享就足够了？我是否需要实现虚拟机之间的存储共享？

很多人会问这样的问题：如果临时存储就可以满足要求，那为什么还要提供块和共享存储呢？要回答这个问题，你可以将临时存储视为与虚拟机关联的系统磁盘，这类存储在虚拟机终止后，终端用户就无法访问上面的数据了。当用户或应用程序不在虚拟机上存储数据时，可认为虚拟机状态并不是特别关键的，因此临时存储有时候也可用于生产环境中。然而，如果需要永久保存数据，则必须考虑使用 Cinder 或 Manila 等持久性存储服务。

需要指出的是，我们当前的逻辑设计是针对大中型 OpenStack 基础设施进行的。而临

时存储也可以作为某些用户的选择，例如，在构建测试环境时就可考虑直接使用临时存储。Swift 也是一种持久性存储，我们之前介绍过对象存储也可用于存储虚拟机镜像，但是我们在什么时候才需要使用这种解决方案呢？简单来说，就是当你认为你的云环境中有很多关键数据，并且开始有数据复制（replication）和冗余（redundancy）需求时，便需考虑这种方案。

1.4.4　逻辑网络设计

在所有 OpenStack 的服务中，最复杂的可能就是网络了。OpenStack 允许灵活多样的网络配置和隧道网络（tunneled network），如 GRE、VXLAN 等。在我们的设计中，在没有获得特定应用场景的情况下，是很难实现 Neutron 的，因为 Neutron 并不是一个容易理解的项目。这也意味着你的网络设计可能因不同的网络拓扑结构而不同，因为针对每个给定的网络用例，其工作模式和实现方式都是不同的。

OpenStack 已经从简单的网络功能转变为更复杂的网络服务，当然也带来了更大的网络灵活性！而这也正是我们使用 OpenStack 的原因，它具备了足够的灵活性！与网络相关的决策并不是随机决定的，下面我们来看看有哪些可用的网络模式。我们将对这些网络模式一一过滤，直到找到满足目标的网络拓扑为止，表 1-3 是对不同网络模式的描述。

表 1-3　不同网络模式对比

网络模式	网络特征	实现
nova-network	扁平网络设计，无租户流量隔离	nova-network Flat DHCP
	租户流量隔离和预定义的固定私有 IP 地址范围 有限的租户网络（VLAN 总数限制为 4K）	nova-network VLAN 管理程序
Neutron	租户流量隔离 有限的租户网络（VLAN 总数限制为 4K）	Neutron VLAN
	租户网络数量增加 数据包大小增加 性能较低	Neutron 隧道网络（GRE、VXLAN 等）

表 1-3 显示了 OpenStack 两种不同逻辑网络设计之间的简单区别。每种逻辑网络都给出了自己的需求，这一点是非常重要的，在部署之前应该对其进行充分考虑。

就我们的示例而言，因为我们的目标是部署一个非常灵活的大规模网络环境，因此我们将选择 Neutron 网络而不是传统的 nova-network。

请注意，你也可以继续使用 nova-network，但必须考虑基础架构中可能存在的任何**单点故障**（Single Point Of Failure，SPOF）。这里选择的是 Neutron，因为我们将从最基本的网络部署开始。在后续章节中将会介绍更多的高级功能。

与 nova-network 相比，Neutron 的主要优势就是 OSI 网络模型中第 2 层和第 3 层的虚拟化实现。

接下来，让我们一起来剖析逻辑网络设计。出于性能考虑，我们强烈建议实现这样一个网络拓扑：使用隔离的逻辑网络来处理不同类型的网络流量。通过这种方式，随着网络的增长，如果突然出现瓶颈或意外故障影响了网络，你的整个虚拟化网络仍然是可管理的。下面，我们来看看 OpenStack 环境中传输不同数据的网络。

1. 物理网络层

我们将从探讨云平台的物理网络需求开始分析物理网络。

2. 租户数据网络

数据网络的主要特征，就是它为 OpenStack 租户创建的虚拟网络提供了物理路径。租户网络将租户数据流与 OpenStack 服务组件之间相互通信的数据流进行了隔离。

3. 管理和 API 网络

在小规模部署中，OpenStack 组件之间的管理和通信流可以共享相同的物理网络。该物理网络提供了各种 OpenStack 组件（如 REST API 访问和数据库流量）之间的通信路径，以及用于管理 OpenStack 节点的路径。

对于生产环境，可以进一步细分网络以便提供更好的流量隔离，以将网络负载隔离到一个独立网络中去。

4. 存储网络

存储网络为虚拟机和存储节点之间的存储数据流提供了物理连接和隔离。由于存储网络的流量负载非常高，因此最好将存储网络负载与管理和租户流量隔离开来。

5. 虚拟网络类型

现在我们来看一下虚拟网络的类型及其功能。

- 外部网络

外部或公共网络的功能特征如下：

- 提供全局互联并使用可路由的 IP 寻址。
- 提供 SNAT 实现机制，以实现虚拟机对外部网络的访问。

SNAT 指的是**源网络地址转换**（Source Network Address Translation）。它允许来自私有网络的流量进入 Internet。OpenStack 通过 Neutron 路由器 API 接口实现 SNAT。更多信息请参考 `http://en.wikipedia.org/wiki/Network_address_translation`。

- 提供 DNAT 实现机制，以实现外部网络对虚拟机的访问。

在使用 VLAN 时，通过标记网络并将多个网络组合到一个**网卡**（NIC）中，可以选择丢弃 NIC 中没有打标记的网络流量，从而简化对 OpenStack 仪表板和公共 OpenStack API 端点的访问。

❑ 租户网络

租户网络的功能特征如下：

❑ 它在虚拟机之间提供专用网络。

❑ 使用私有 IP 空间。

❑ 提供租户流量隔离，允许网络服务的多租户需求。

图 1-5 是对我们上述网络逻辑设计的具体呈现。

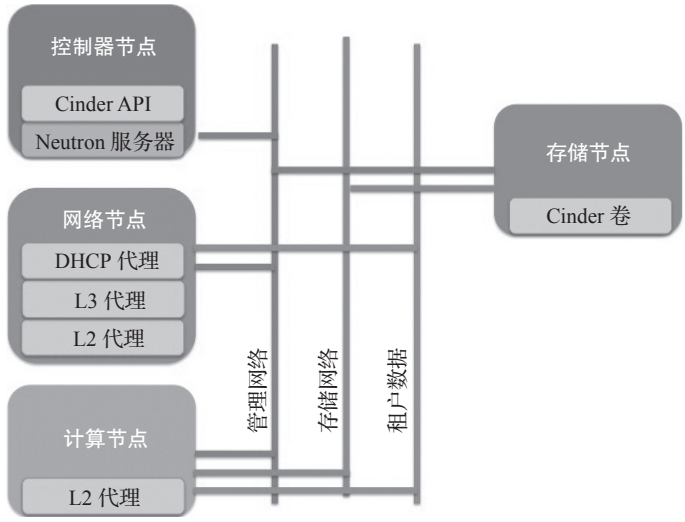

图 1-5　OpenStack 网络设计

1.5　OpenStack 物理模型设计

最后，我们将以物理设计的形式来实现逻辑设计。一开始，我们的目标只是为云环境实现初次部署，因此仅使用小规模数量的服务器集群。要实现**大规模集群架构**，你必须慎重考虑商业硬件设备的选择。

1.5.1　估算硬件容量

由于架构被设计为可水平扩展的，因此可以在集群中添加更多的服务器。我们一开始将使用低成本的商用服务器。

为了实现预期的经济成本，在首次部署时，对硬件成本进行评估是非常必要的。

在部署之前，你需要考虑到 CPU、RAM、网络和磁盘等资源存在争抢的可能性，你不能等到某些物理设备出现故障了才采取措施，这会使问题变得非常复杂。

我们来看下低估了容量规划进而造成影响的现实例子。某个云托管公司配置了两台中

型服务器，一台用于电子邮件服务器，另一台用于托管官方网站。该公司是我们的一个客户，在几个月内发展壮大，最终耗尽了磁盘空间。通常，解决此类问题预计将花费几个小时，而实际上却花费了几天的时间。这是由于云服务本质上能被按需获取，因此用户没能正确地使用云资源。这直接导致了**平均修复时间**（Mean Time To Repair，MTTR）呈指数增长。云供应商肯定不希望看到这种情况！

这件事充分说明恰当的容量规划对云基础架构的重要性。容量管理是日常职责，在软件和硬件升级后，也必须更新容量管理。通过对资源使用进行持续监控，可以降低 IT 风险并快速响应客户需求。在硬件部署之初，就应该通过反复调整、监视和分析来保证你的容量管理过程。

下一步就是考虑调整参数，并对硬件或软件进行适当的升级更新，这就涉及变更管理过程。下面，我们来根据特定需求进行第一次容量评估，例如，假设我们的目标是在 OpenStack 环境中运行 200 个虚拟机。

1.5.2 CPU 评估

以下是与评估相关的假设：

- 200 台虚拟机。
- 没有 CPU 超分。

CPU 超分（over subscription）定义为分配给所有已打开电源的虚拟机的 CPU 总数乘以硬件 CPU 核心（core/processor）。如果此数字大于购买的 GHz 数，则超分。

- 每个物理核心 GHz 数 = 2.6GHz。
- 物理核心超线程支持 = 使用因子 2。
- 每个虚拟机 GHz 数（平均计算单位）= 2GHz。
- 每个虚拟机 GHz 数（最大计算单位）= 16GHz。
- 每个 Intel Xeon E5-2648L v2 CPU 的核心数 =10。
- 每台服务器的 CPU 插槽数 = 2。

计算 CPU 核心总数的公式如下：

（虚拟机数量 × 每个虚拟机的 GHz 数）/ 每个 CPU 核心的 GHz 数

（200×2）/2.6 = 153.846

我们为 200 个虚拟机准备 153 个 CPU 核心。计算总 CPU 插槽数的公式如下：

CPU 核心 / 每个 CPU 的核心数

153/10 = 15.3

我们需要 15 个物理 CPU 插槽。计算物理服务器数量的公式如下：

CPU 插槽总数 / 每台服务器的 CPU 插槽数

15/2 = 7.5

你将需要大约 7 ～ 8 台双插槽服务器（通常称为双路服务器）。如果有 8 台双插槽服务器，则我们可以在每台服务器上部署 25 个虚拟机：

虚拟机数 / 服务器数

200/8 = 25

1.5.3　内存评估

根据前面的估算，每个计算节点可以部署 25 个虚拟机。内存大小调整对于避免进行不合理的资源分配也很重要。

我们先来假设一个资源清单（这取决于你的预算和需求）：

- ❑ 每个虚拟机配置 2GB 内存。
- ❑ 每个虚拟机最多动态分配 8GB 内存。
- ❑ 计算节点支持的内存条规格：2GB、4GB、8GB 和 16GB。
- ❑ 每个计算节点可以使用的内存：

8 × 25 = 200GB

考虑到服务器中内存支持的限制，你需要安装大约 256GB 内存。因此，安装的内存条总数可以通过以下方式计算：

总可用内存 / 最大可用内存条规格

256/16 = 16

1.5.4　网络评估

为了完成前面的规划架构，我们假设：

- ❑ 每个虚拟机带宽是 200Mbps。
- ❑ 最小网络延迟。

为此，可以通过为每个服务器使用 10GB 网卡来为我们的虚拟机提供服务，最终结果是：

10 000Mbps/25 = 400Mbps

这是一个非常令人满意的数值。我们还需要考虑另一个因素：高可用网络架构。因此，一个可采用的方案就是使用两个数据交换机，最少有 24 个数据端口。考虑到以后的扩容，我们将使用两个 48 端口交换机。

如何应对机架空间不够使用的情况？这种情况下，应该考虑在聚合中的交换机之间使用**多设备链路聚合**（Multi-Chassis Link Aggregation，MCLAG/MLAG）技术，此功能允许每个服务器机架在这对交换机之间划分其链路，以实现强大的活动 – 活动（Active-Active）转发，同时使用全带宽功能而无须启用生成树协议（spanning-tree protocol）。

MCLAG 是连接到交换机的服务器之间的第 2 层链路聚合协议，提供到核心网络冗余的、负载均衡的连接，并替换生成树协议。

网络配置还高度依赖于所选的网络拓扑。如图 1-5 所示，你应该要清楚，OpenStack 环境中的所有节点都必须相互通信。基于这个要求，管理员需要标准化每个单元并计算出所需的公共和浮动 IP 地址数。这个估算取决于 OpenStack 环境所配置的网络类型，包括 Neutron 或之前的 nova-network 服务。规划清楚哪些 OpenStack 单元需要使用公共 IP 或浮动 IP 是非常重要的。在我们的示例中，假设各个单元需要使用的公共 IP 地址如下：

- 云控制器节点：3 个。
- 计算节点：15 个。
- 存储节点：5 个。

在这种情况下，我们最初需要至少 18 个公共 IP 地址。此外，如果使用了负载均衡器高可用设置，则负载均衡器前端的虚拟 IP 地址将被视为附加的公共 IP 地址。

在 OpenStack 中使用 Neutron 网络，意味着需要准备用于与其他网络节点和私有云环境进行网络交互的虚拟设备和接口，我们假设：

- 20 个租户需要的虚拟路由器：20 个。
- 15 个计算节点可以容纳的最大虚拟机数量：375 个。

在这种情况下，我们最初需要至少 395 个浮动 IP 地址，因为每个虚拟路由器都要能够连接到公共网络。此外，应提前考虑增加可用带宽，因此，我们需要考虑使用 NIC 绑定，所以需要将网卡的数量乘以 2。网卡绑定将增强云网络的高可用性并实现网络带宽性能的提升。

1.5.5 存储评估

考虑到前面的示例，需要为每台服务器估算初始存储容量，每个服务器将为 25 个虚拟机提供服务。一个简单的计算是，假设每个虚拟机需要 100GB 的临时存储，则每个计算节点上需要 25 × 100GB = 2.5TB 的本地存储空间。可以为每个虚拟机分配 250GB 的持久存储，以使每个计算节点具有 25 × 250GB = 5TB 的持久存储。

在很多情况下，你可能想利用 OpenStack 中对象存储的副本机制，这意味着需要 3 倍的存储空间。换句话说，如果计划将 X TB 数据用于对象存储，则存储空间的需求为 $3X$ TB。

关于存储，还有一些其他的考虑因素，例如使用 SSD 将得到最佳的存储性能，实现更好的吞吐量，可以投入更多的 SSD 来获得更高的 IOPS。例如，在具有 8 个插槽驱动器的服务器中安装具有 20K IOPS 的 SSD，你所得到的读写 IOPS 为：

（20K × 8）/25 = 6.4K 读取 IOPS 和 3.2K 写入 IOPS

对于生产环境来说，这已经很不错了！

1.6　OpenStack 设计最佳实践

下面，通过仔细分析 OpenStack 设计架构，我们介绍一些最佳实践。在典型的 OpenStack 生产环境中，每个计算节点的磁盘空间最低要求为 300GB，最小 RAM 为 128GB，两路 8 核 CPU。

我们假设这样一个场景，由于预算限制，你的第一个计算节点拥有 600GB 磁盘空间、16 核 CPU 和 256GB RAM。假设 OpenStack 环境持续增长，你可能决定购买更多硬件。然后，第二个计算节点的扩容完成了。

之后不久，可能发现需求又在增加。这时候，你开始将请求拆分到不同的计算节点，但仍然进行硬件扩容。此时，可能会收到超出预算限额的提醒！有时候，最佳实践实际上对你的设计并非最佳。前面的示例仅是 OpenStack 部署中的一个通用需求。

如果严格遵循最佳实践推荐的最低硬件配置要求，则可能导致硬件费用呈指数上升，特别是对于刚入门的项目实施者，经常会碰到这种情况。因此，应该正确选择适合你自身环境的配置，并考虑到你自身环境中存在的约束。

请记住，最佳实践是一个指导原则，当你清楚需要部署的内容以及如何配置它们的时候，可以直接应用最佳实践。

另一方面，不要一味追求数字，而要追寻规则背后的精髓。我们再来看下前面的示例，与水平扩展（scaling out 或者 scaling horizontally）相比，纵向扩展（scaling up）存在更多的风险，并可能导致失败。这种设计背后的原因，就是以较低的计算成本和较小的系统代价，快速替换大规模集群中的功能模块。这也正是 OpenStack 的设计方式：功能已降级的单元可被丢弃，失效的工作负载可被替换。

计算节点中的事务和请求可能会在短时间内大幅增长，导致具有 16 核 CPU 的单个大型计算节点在性能方面出现问题，而多个仅有 4 核 CPU 的小型计算节点却可以继续成功完成作业。正如我们在上一节中论述的，容量规划是一项非常烦琐的工作，但是对于构建成功的 OpenStack 云平台却是至关重要的。

对于扩容的考虑和如何实现扩容，是在 OpenStack 原始设计中就必须思考的问题。我们所要考虑的是基于需求的增长，而 OpenStack 的中负载增长是弹性而非线性的。虽然之前的资源估算示例可能有助于我们规划 OpenStack 初期的资源需求，但是真正达到现实可接受的容量规划仍然需要我们进行更多的考虑和行动，例如，根据工作负载的增长对云性能进行详细分析。此外，通过使用更为复杂的监控工具，管理员应持续跟踪运行在 OpenStack 环境中的每个服务单元，例如，随着时间推移的整体资源消耗以及导致性能下降的某些单元的过度使用情况。由于我们对未来的硬件资源容量进行了粗略估算，因此可以在首次部署后利用每个计算主机节点上实例的规格来强化这个估算模型，并在具备了详细资源监控的情况下对其实现按需调整。

1.7 总结

本章回顾了 OpenStack 中最基本的组件，并介绍了一些新功能，比如计量服务、编排服务和文件共享项目等。

我们实现了初期的 OpenStack 设计架构，并为将来要做的部署针对逻辑设计模型做了持续优化。作为介绍性的章节，我们重新介绍了 OpenStack 的每个组件，并简要讨论了每个服务组件的用例及其在生态系统中的角色。我们还介绍了一些策略性提示，以规划和应对 OpenStack 生产环境中未来可能的扩容需求。

本章是其余章节的主要参考，后续我们将扩展本章所涉及的基本介绍来深入了解 OpenStack 中的每个组件和新功能。在接下来的章节中，我们将继续 OpenStack 之旅，并通过稳定、高效的 DevOps 方式来部署 OpenStack 集群。