目录

[1. 云计算 1](#_Toc522137448)

[1.1 定义 1](#_Toc522137449)

[1.2 三层架构 1](#_Toc522137450)

[2. 什么是kvm? 2](#_Toc522137451)

[2.1 KVM的来源 2](#_Toc522137452)

[2.2 KVM的架构 2](#_Toc522137453)

[2.3 用户态和内核态 3](#_Toc522137454)

[2.3.1 用户态和内核态的概念 3](#_Toc522137455)

[2.3.2 用户态和内核态之间切换 3](#_Toc522137456)

[2.4 qemu、kvm、qemu-kvm、openstack、libvert之间的关系 5](#_Toc522137457)

[2.4.1 QEMU 5](#_Toc522137458)

[2.4.2 KVM 5](#_Toc522137459)

[2.4.3 KVM-QEMU 6](#_Toc522137460)

[2.4.4 LIBVIRT 6](#_Toc522137461)

[2.4.5 openstack, kvm, qemu-kvm及libvirt之间的关系 9](#_Toc522137462)

[3. Openstack 10](#_Toc522137463)

[3.1 定义 10](#_Toc522137464)

[3.2 openstack的架构 10](#_Toc522137465)

[3.3 扩展一：块存储、对象存储、文件存储 12](#_Toc522137466)

[4. 虚拟化工具VMware WorkStations的四种网路模型 13](#_Toc522137467)

[4.1 NAT模式(network address translation) 13](#_Toc522137468)

[4.2 Bridged Adapter模式 13](#_Toc522137469)

[4.3 Internal模式 14](#_Toc522137470)

[4.4 Host-only Adapter模式 14](#_Toc522137471)

[4.5 总结 14](#_Toc522137472)

[5. 构建网络拓扑环境 15](#_Toc522137473)

[5.1 物理硬件配置 15](#_Toc522137474)

[6.2 节点服务 16](#_Toc522137475)

[6.3 网络架构图 16](#_Toc522137476)

[6. 构建基础环境 17](#_Toc522137477)

[2.1关闭firewalld 17](#_Toc522137478)

[2.2 修改selinux 17](#_Toc522137479)

[2.3 关闭NetworkManager 17](#_Toc522137480)

[2.4 安装ntp服务 17](#_Toc522137481)

[2.4 配置openstack源 18](#_Toc522137482)

[2.5 安装SQL database 19](#_Toc522137483)

[2.6 安装消息队列 19](#_Toc522137484)

[2.7 安装memcached 19](#_Toc522137485)

[2.6 安装etcd服务 20](#_Toc522137486)

[7. Keystone组件 20](#_Toc522137487)

[7.1 Keystone的架构 20](#_Toc522137488)

[7.2 Keystone的基本概念 21](#_Toc522137489)

[7.3 上述概念之间的关系 22](#_Toc522137490)

[7.4 keystone在Openstack中的工作流程实例 23](#_Toc522137491)

[7.5 实战：Keystone的手动搭建 24](#_Toc522137492)

[7.5.1 控制节点的安装配置 24](#_Toc522137493)

[7.5.2 创建域、项目、用户和角色 25](#_Toc522137494)

[7.5.3 验证 26](#_Toc522137495)

[8.Glance组件 27](#_Toc522137496)

[8.1 定义 27](#_Toc522137497)

[8.2 glance架构 28](#_Toc522137498)

[8.3 Glance支持的Image格式 29](#_Toc522137499)

[8.4 glance的访问权限 29](#_Toc522137500)

[8.5 状态类型及转化 29](#_Toc522137501)

[8.6 两种API的区别 30](#_Toc522137502)

[8.6.1 glance API V1 30](#_Toc522137503)

[8.6.2 glance API V2 31](#_Toc522137504)

[8.7 实战： glance的手动搭建 31](#_Toc522137505)

[8.7.1 控制节点的安装配置 31](#_Toc522137506)

[8.7.2 验证 33](#_Toc522137507)

[9.Nova组件 33](#_Toc522137508)

[9.1 nova的定义 33](#_Toc522137509)

[9.2 Nova的架构 34](#_Toc522137510)

[9.3 Nova 的逻辑模块 34](#_Toc522137511)

[9.4 Nova启动一个虚拟机的过程 35](#_Toc522137512)

[9.5 Nova Scheduler Filter的类型 37](#_Toc522137513)

[9.6 实战：nova的手动搭建 39](#_Toc522137514)

[9.6.1 Controller节点 39](#_Toc522137515)

[9.6.2 Compute节点 42](#_Toc522137516)

[9.6.3 验证 44](#_Toc522137517)

[10.Neutron 44](#_Toc522137518)

[10.1 概述 44](#_Toc522137519)

[10.2 网络虚拟化的原因 44](#_Toc522137520)

[10.2.1 现有物理网络不能满足云计算的需求 44](#_Toc522137521)

[10.2.2 现有物理网络不能满足云计算SDN的需求 46](#_Toc522137522)

[10.3 Neutron的架构 49](#_Toc522137523)

[10.4 Neutron支持的网络模型 50](#_Toc522137524)

[10.5 Neutron中的各种概念 53](#_Toc522137525)

[10.6 Neutron中的Plugin和agent 56](#_Toc522137526)

[10.7 实战：neutron的手动搭建 56](#_Toc522137527)

[10.7.1 linuxbridge+vxlan模式下的手动部署 56](#_Toc522137528)

[10.7.2 Ovs+vxlan模式的手动部署 62](#_Toc522137529)

[11.实战：Horizon的手动搭建 62](#_Toc522137530)

[12. 创建一个实例 63](#_Toc522137531)

[12.1 创建虚拟网络 63](#_Toc522137532)

[12.1.1 创建provider网络 63](#_Toc522137533)

[12.1.2 创建私有网络 self-services 63](#_Toc522137534)

[12.1.3 验证 64](#_Toc522137535)

[12.2 创建flavor 64](#_Toc522137536)

[12.3 生成密钥对 64](#_Toc522137537)

[12.4 添加安全组规则 64](#_Toc522137538)

[12.5启动一个实例 65](#_Toc522137539)

[13. Cinder组件 68](#_Toc522137540)

[13.1 cinder的定义 68](#_Toc522137541)

[13.2 cinder的架构 69](#_Toc522137542)

[13.3 实战：cinder的手动搭建 70](#_Toc522137543)

[13.3.1 控制节点 70](#_Toc522137544)

[13.3.2 存储节点 72](#_Toc522137545)

[13.3 验证 73](#_Toc522137546)

[13.4实战：给虚拟机分配一个虚拟磁盘 73](#_Toc522137547)

[14. Swift组件 73](#_Toc522137548)

[14.1 swift概述 73](#_Toc522137549)

[14.2 swift的基本原理一：一致性散列 74](#_Toc522137550)

[14.3 swift的基本原理二：数据一致性模型 75](#_Toc522137551)

[14.4 swift的基本原理三：环的数据结构 76](#_Toc522137552)

[13.5 swift的基本原理四：数据模型 77](#_Toc522137553)

[14.6 swift的基本原理四：系统架构 78](#_Toc522137554)

[14.7 swift的特性 80](#_Toc522137555)

[14.8 实战：swift的手动搭建 81](#_Toc522137556)

[14.8.1 控制节点 81](#_Toc522137557)

[14.8.2 存储节点 84](#_Toc522137558)

[14.8.3 Create and distribute initial rings 87](#_Toc522137559)

[14.8.4 验证 89](#_Toc522137560)

[附录1：制作openstack的Linux镜像 90](#_Toc522137561)

[附录2：制作OpenStack的window镜像 90](#_Toc522137562)

[附录3：容器制作openstack本地yum源 90](#_Toc522137563)

# 云计算

## 1.1 定义

维基百科定义：

**Cloud computing** is a type of [Internet](https://en.wikipedia.org/wiki/Internet" \o "Internet)**-based** computing that provides **shared computer processing resources and data** to computers and other devices **on demand**.

百度百科定义：

**Cloud computing** is a style of computing in which **dynamically scalable** and often virtualized resources are provided as a service over the Internet.

翻译：[云计算](http://baike.baidu.com/view/1316082.htm" \t "_blank)（cloud computing）是**基于互联网**的相关服务的增加、使用和交付模式，通常涉及通过互联网来提供**动态易扩展**且经常是**虚拟化**的资源。

## 1.2 三层架构

三层网络架构如图1-1所示。



图1-1 云计算的三层体系结构

**IaaS**：基础设施即服务，物理机的管理，虚拟机的管理，存储资源的管理。

**PaaS**：平台及服务，虚拟机搭建的开发平台，比如配置好的Apache、MySql或者PHP环境

**SaaS**：软件即服务，比如购物网站、博客网站、微博网站等。

扩展：openstack位于那一层？

# 什么是kvm?

## 2.1 KVM的来源

KVM—基于内核的虚拟机（Kernel-Based Virtual Machine），将Linux内核变成Hypervisor的一种虚拟化技术，属于硬件级虚拟化类型的全虚拟化的解决方案。是Linux的内核特性，KVM需要两个条件：硬件支持全虚拟化，操作系统是Linux。

## 2.2 KVM的架构

KVM的架构如图2-1所示。



图2-1 KVM的虚拟化系统架构

KVM作为Hypervisor运行在宿主机内核（Host OS Kernel），支持对CPU、内存、I/O的模拟，对虚拟机的监控，并为Qemu提供实体支撑。

Qemu作为进程运行在宿主机的用户态，它基于KVM及内核的特性，为Guest OS模拟出CPU、内存、I/O等硬件，支撑Guest OS在进程中运行。

虚拟化管理服务有多种，大部分都基于libvirt实现，libvirt 库是一种实现 Linux 虚拟化功能的 Linux API，它支持各种虚拟机监控程序。

## 2.3 用户态和内核态

### 2.3.1 用户态和内核态的概念

当一个任务（进程）执行**系统调用**而陷入内核代码中执行时，我们就称进程处于**内核运行态（或简称为内核态）**。此时处理器处于**特权级**最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。这与处于内核态的进程的状态有些类似。

核态与用户态是操作系统的两种运行级别,跟intel cpu没有必然的联系, intel cpu提供Ring0-Ring3三种级别的运行模式，Ring0级别最高，Ring3最低。Linux使用了Ring3级别运行用户态，Ring0作为内核态，没有使用Ring1和Ring2。Ring3状态不能访问Ring0的地址空间，包括代码和数据。Linux进程的4GB地址空间，3G-4G部分大家是共享的，是内核态的地址空间，这里存放在整个内核的代码和所有的内核模块，以及内核所维护的数据。用户运行一个程序，该程序所创建的进程开始是运行在用户态的，如果要执行文件操作，网络数据发送等操作，必须通过write，send等系统调用，这些系统调用会调用内核中的代码来完成操作，这时，必须切换到Ring0，然后进入3GB-4GB中的内核地址空间去执行这些代码完成操作，完成后，切换回Ring3，回到用户态。这样，用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用。

### 2.3.2 用户态和内核态之间切换

a. 系统调用

这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

b. 异常

当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

c. 外围设备的中断

当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

2）具体的切换操作

从触发方式上看，可以认为存在前述3种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一致的，没有任何区别，都相当于执行了一个中断响应的过程，因为**系统调用实际上最终是中断机制实现**的，而异常和中断的处理机制基本上也是一致的，关于它们的具体区别这里不再赘述。关于中断处理机制的细节和步骤这里也不做过多分析，涉及到由用户态切换到内核态的步骤主要包括：

[1] 从当前进程的描述符中提取其内核栈的ss0及esp0信息。

[2] 使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp信息保存起来，这个

过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一

条指令。

[3] 将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始

执行中断处理程序，这时就转到了内核态的程序执行了。

## 2.4 qemu、kvm、qemu-kvm、openstack、libvert之间的关系

### 2.4.1 QEMU

**Qemu是一个独立的虚拟化解决方案**，通过intel-VT 或AMD SVM实现全虚拟化，安装qemu的系统，可以直接模拟出另一个完全不同的系统环境，虚拟机的创建通过qemu-image既可完成。**QEMU本身可以不依赖于KVM**，但是如果有KVM的存在并且硬件(处理器)支持比如Intel VT功能，那么QEMU在对处理器虚拟化这一块可以利用KVM提供的功能来提升性能。

### KVM

**KVM是集成到Linux内核的Hypervisor是X86架构且硬件支持虚拟化技术（IntelVT或AMD-V）的Linux的全虚拟化解决方案**。它是Linux的一个很小的模块，利用Linux做大量的事，如任务调度、内存管理与硬件设备交互等。准确来说，KVM是Linux kernel的一个模块。可以用命令modprobe去加载KVM模块。加载了模块后，才能进一步通过其他工具创建虚拟机。但仅有KVM模块是远远不够的，因为用户无法直接控制内核模块去作事情,你还必须有一个运行在用户空间的工具才行。这个用户空间的工具，kvm开发者选择了已经成型的开源虚拟化软件QEMU。说起来QEMU也是一个虚拟化软件。它的特点是可虚拟不同的CPU。比如说在x86的CPU上可虚拟一个Power的CPU，并可利用它编译出可运行在Power上的程序。KVM使用了QEMU的一部分，并稍加改造，就成了可控制KVM的用户空间工具了。所以你会看到，官方提供的KVM下载有两大部分(qemu和kvm)三个文件(KVM模块、QEMU工具以及二者的合集)。也就是说，你可以只升级KVM模块，也可以只升级QEMU工具。这就是KVM和QEMU的关系。

 KVM内核模块本身只能提供CPU和内存的虚拟化，所以它必须结合QEMU才能构成一个完成的虚拟化技术，这就是下面要说的qemu-kvm。

### KVM-QEMU

Qemu将KVM整合进来，通过ioctl调用/dev/kvm接口，将有关CPU指令的部分交由内核模块来做。**kvm负责cpu虚拟化+内存虚拟化，实现了cpu和内存的虚拟化**，但kvm不能模拟其他设备。**qemu模拟IO设备（网卡，磁盘等），kvm加上qemu之后就能实现真正意义上服务器虚拟化**。因为用到了上面两个东西，所以称之为qemu-kvm。

Qemu模拟其他的硬件，如Network, Disk，同样会影响这些设备的性能，于是又产生了pass through半虚拟化设备virtio\_blk, virtio\_net，提高设备性能。

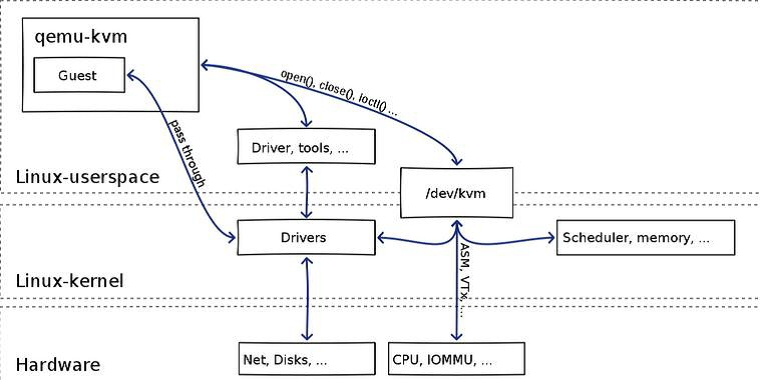


图2-2 qemu-kvm的使用方式

### 2.4.4 LIBVIRT

a)概念

虚拟云实现的三部曲：虚拟化技术实现-->虚拟机管理-->集群资源管理(云管理)。各种不同的虚拟化技术都提供了基本的管理工具。比如，启动，停用，配置，连接控制台等。这样在构建云管理的时候就存在两个问题：

1) 如果采用混合虚拟技术，上层就需要对不同的虚拟化技术调用不同管理工具，很是麻烦。

2) 虚拟化技术发展很迅速，系统虚拟化和容器虚拟化均在发展和演化中。可能有新的虚拟化技术更加符合现在的应用场景，需要迁移过去。这样管理平台就需要大幅改动。

为了适应变化，我们惯用的手段是分层，使之相互透明，在虚拟机和云管理中设置一个抽象管理层。libvirt就是扮演的这个角色。libvirt提供各种API，供上层来管理不同的虚拟机。

**Libvirt是管理虚拟机和其他虚拟化功能，比如存储管理，网络管理的软件集合。它包括一个API库，一个守护程序（libvirtd）和一个命令行工具（virsh）；libvirt本身构建于一种抽象的概念之上。它为受支持的虚拟机监控程序实现的常用功能提供通用的AP**I。

 libvirt的主要目标是为各种虚拟化工具提供一套方便、可靠的编程接口，用一种单一的方式管理多种不同的虚拟化提供方式。

b)功能

虚拟机管理：包括不同的领域生命周期操作，比如：启动、停止、暂停、保存、恢复和迁移。支持多种设备类型的热插拔操作，包括：磁盘、网卡、内存和CPU。

  远程机器支持：只要机器上运行了libvirt daemon，包括远程机器，所有的libvirt功能就都可以访问和使用。支持多种网络远程传输，使用最简单的SSH，不需要额外配置工作。

    存储管理：任何运行了libvirt daemon的主机都可以用来管理不同类型的存储：创建不同格式的文件镜像（qcow2、vmdk、raw等）、挂接NFS共享、列出现有的LVM卷组、创建新的LVM卷组和逻辑卷、对未处理过的磁盘设备分区、挂接iSCSI共享，等等等等。因为libvirt可以远程工作，所有这些都可以通过远程主机使用。

    网络接口管理：任何运行了libvirt daemon的主机都可以用来管理物理和逻辑的网络接口。

    虚拟NAT和基于路由的网络：任何运行了libvirt daemon的主机都可以用来管理和创建虚拟网络。

c) 体系架构

没有使用libvirt的虚拟机管理方式如图2-3所示：

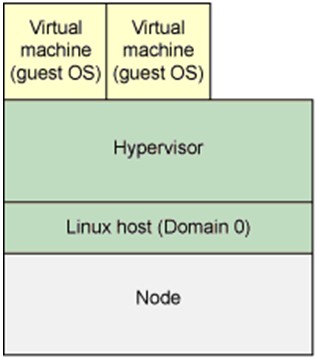


图2-3 没有使用libvirt的虚拟机管理方式

    为支持各种虚拟机监控程序的可扩展性，libvirt实施一种基于驱动程序的架构，该架构允许一种通用的API以通用方式为大量潜在的虚拟机监控程序提供服务。下图展示了libvirt API与相关驱动程序的层次结构。这里也需要注意，libvirtd提供从远程应用程序访问本地域的方式。

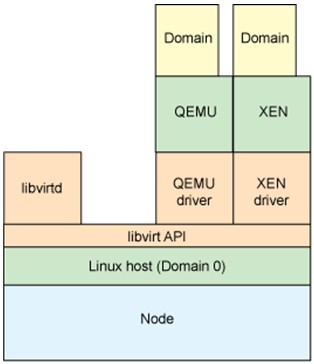


图2-4 使用libvirt的虚拟机管理方式

    libvirt的控制方式有两种：

    1）管理应用程序和域位于同一节点上。管理应用程序通过libvirt工作，以控制本地域。

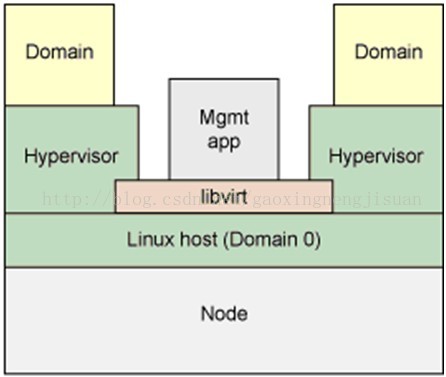


图2-5 libvirt的控制方式1

    2）管理应用程序和域位于不同节点上。该模式使用一种运行于远程节点上、名为libvirtd的特殊守护进程。当在新节点上安装libvirt时该程序会自动启动，且可自动确定本地虚拟机监控程序并为其安装驱动程序。该管理应用程序通过一种通用协议从本地libvirt连接到远程libvirtd。

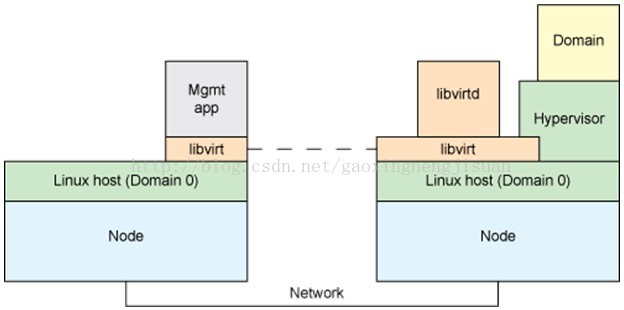


图2-6 libvirt的控制方式2

### openstack, kvm, qemu-kvm及libvirt之间的关系

**KVM是最底层的hypervisor，它是用来模拟CPU的运行，它缺少了对network和周边I/O的支持，所以我们是没法直接用它的。QEMU-KVM就是一个完整的模拟器，它是构建基于KVM上面的，它提供了完整的网络和I/O支持。Openstack不会直接控制qemu-kvm，它会用一个叫libvirt的库去间接控制qemu-kvm。libvirt提供了跨VM平台的功能，它可以控制除了QEMU之外的模拟器，包括vmware, virtualbox， xen等等。所以为了openstack的跨VM性，所以openstack只会用libvirt而不直接用qemu-kvm。libvirt还提供了一些高级的功能，例如pool/vol管理。**

# Openstack

## 3.1 定义

OpenStack是由美国国家宇航局（NASA）和RackSpace合作开发的旨在为共有云和私有云提供软件的开源项目，其目的在于提供可靠的云部署方案及良好的可扩展性。简单的说OpenStack就是云操作系统，或者说是云管理平平台，自身并不提供云服务，只是提供部署和管理平平台。

## 3.2 openstack的架构

OpenStacK是开源的云管理平台，由各个相互独立的模块构成。OpenStack的架构如图3-1所示。

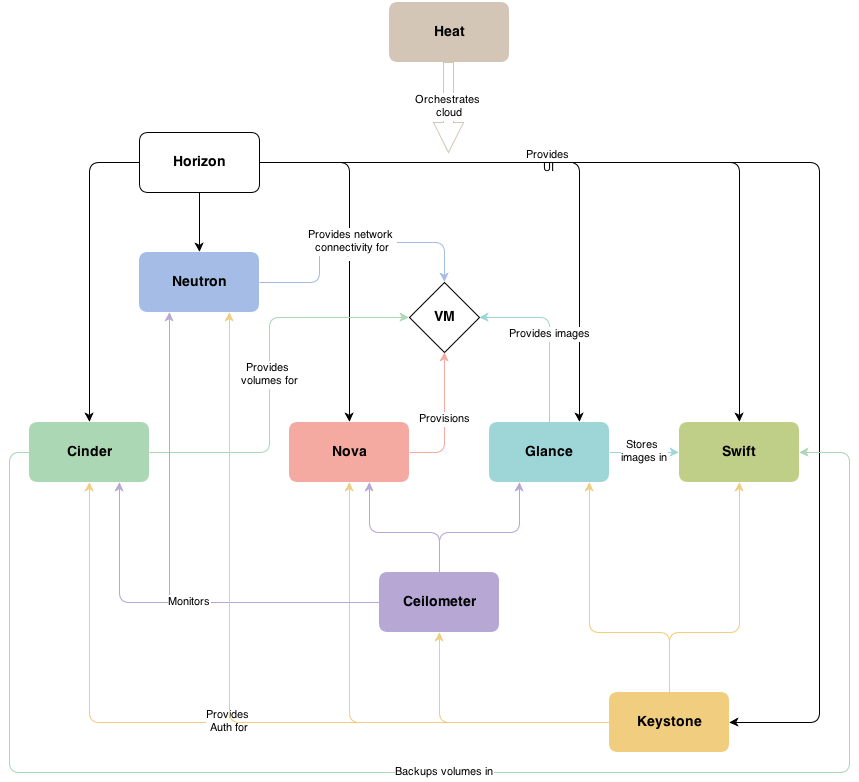


图3-1 OpenStack的架构

OpenStack组件及功能介绍如表3-1所示。

表3-1 OpenStack组件及功能介绍

| **服务** | **项目名称** | **描述** |
| --- | --- | --- |
| [Dashboard](http://www.openstack.org/software/releases/liberty/components/horizon) | [Horizon](http://docs.openstack.org/developer/horizon/) | **提供了一个基于web的自服务门户，与OpenStack底层服务交互，诸如启动一个实例，分配IP地址以及配置访问控制。** |
| [Compute](http://www.openstack.org/software/releases/liberty/components/nova) | [Nova](http://docs.openstack.org/developer/nova/) | **在OpenStack环境中计算实例的生命周期管理。按需响应包括生成、调度、回收虚拟机等操作。** |
| [Networking](http://www.openstack.org/software/releases/liberty/components/neutron) | [Neutron](http://docs.openstack.org/developer/neutron/) | **确保为其它OpenStack服务提供网络连接即服务，比如OpenStack计算。为用户提供API定义网络和使用。基于插件的架构其支持众多的网络提供商和技术。** |
| 存储 |  |  |
| [Object Storage](http://www.openstack.org/software/releases/liberty/components/swift) | [Swift](http://docs.openstack.org/developer/swift/) | **通过一个 [RESTful](http://docs.openstack.org/mitaka/zh_CN/install-guide-rdo/common/glossary.html" \l "term-restful),基于HTTP的应用程序接口存储和任意检索的非结构化数据对象。它拥有高容错机制，基于数据复制和可扩展架构。它的实现并像是一个文件服务器需要挂载目录。在此种方式下，它写入对象和文件到多个硬盘中，以确保数据是在集群内跨服务器的多份复制。** |
| [Block Storage](http://www.openstack.org/software/releases/liberty/components/cinder) | [Cinder](http://docs.openstack.org/developer/cinder/) | **为运行实例而提供的持久性块存储。它的可插拔驱动架构的功能有助于创建和管理块存储设备。** |
| 共享服务 |  |  |
| [Identity service](http://www.openstack.org/software/releases/liberty/components/keystone) | [Keystone](http://docs.openstack.org/developer/keystone/) | **为其他OpenStack服务提供认证和授权服务，为所有的OpenStack服务提供一个端点目录。** |
| [Image service](http://www.openstack.org/software/releases/liberty/components/glance) | Glance | **存储和检索虚拟机磁盘镜像，OpenStack计算会在实例部署时使用此服务。** |
| Telemetry | Ceilometer | **为OpenStack云的计费、基准、扩展性以及统计等目的提供监测和计量。** |
| 高层次服务 |  |  |
| Orchestration | Heat | **Orchestration服务支持多样化的综合的云应用，通过调用OpenStack-native REST API和CloudFormation-compatible Query API，支持:term:[`](http://docs.openstack.org/mitaka/zh_CN/install-guide-rdo/overview.html" \l "id1)HOT <Heat Orchestration Template (HOT)>`格式模板或者AWS CloudFormation格式模板** |

## 3.3 扩展一：块存储、对象存储、文件存储

块存储，典型代表--SAN。对于用户来说，SAN好比是一块大磁盘，用户可以根据需要随意将SAN格式化成想要的文件系统来使用。SAN在网络中通过iSCSI（IPSAN）协议连接，属block及存储，但可扩展性较差。

文件存储，典型代表--NAS。对于用户来说，NAS好比是一个[共享文件夹](https://www.baidu.com/s?wd=%E5%85%B1%E4%BA%AB%E6%96%87%E4%BB%B6%E5%A4%B9&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-PHKhmyD1Pj0vm161mHP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW6kP1ckPHDznHf1PHDkrHTz" \t "_blank)，文件系统已经存在，用户可以直接将自己的数据存放在NAS上。NAS以文件为传输协议，开销很大，不利于在高性能集群中使用。  
 对象存储，就是每个数据对应着一个唯一的id，在[面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-PHKhmyD1Pj0vm161mHP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW6kP1ckPHDznHf1PHDkrHTz" \t "_blank)存储中，不再有类似文件系统的目录层级结构，完全扁平化存储，即可以根据对象的id直接定位到数据的位置，这一点类似SAN，而每个数据对象即包含元数据又包括存储数据，含有文件的概念，这一点类似NAS。除此之外，用户不必关系数据对象的安全性，数据恢复，自动负载平衡等等问题，这些均由[对象存储系统](https://www.baidu.com/s?wd=%E5%AF%B9%E8%B1%A1%E5%AD%98%E5%82%A8%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-PHKhmyD1Pj0vm161mHP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW6kP1ckPHDznHf1PHDkrHTz" \t "_blank)自身完成。而且，[面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-PHKhmyD1Pj0vm161mHP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW6kP1ckPHDznHf1PHDkrHTz" \t "_blank)存储还解决了SAN面临的有限扩充和NAS传输性能开销大问题，能够实现海量[数据存储](https://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E5%AD%98%E5%82%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-PHKhmyD1Pj0vm161mHP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW6kP1ckPHDznHf1PHDkrHTz" \t "_blank)。

# 虚拟化工具VMware WorkStations的四种网路模型

## 4.1 NAT模式(network address translation)

Vhost访问网络的所有数据都是由主机提供的，vhost并不真实存在于网络中，主机与网络中的任何机器都不能查看和访问到Vhost的存在。

**虚拟机与主机：**虚拟机可以通过网络访问到主机，主机无法通过网络访问到虚拟机。

**虚拟机与其他主机：**虚拟机可以访问到网络中其他主机，其他主机不能通过网络访问到虚拟机。

**虚拟机之间：**相互不能访问，虚拟机与虚拟机各自完全独立，相互间无法通过网络访问彼此。

## 4.2 Bridged Adapter模式

通过主机网卡，架设了一条桥，直接连入到网络中了。因此，它使得虚拟机能被分配到一个网络中独立的IP，所有网络功能完全和在网络中的真实机器一样。

**虚拟机与主机：**可以相互访问，因为虚拟机在真实网络段中有独立IP，主机与虚拟机处于同一网络段中，彼此可以通过各自IP相互访问。

**虚拟机与其他主机：** 可以相互访问，同样因为虚拟机在真实网络段中有独立IP，虚拟机与所有网络其他主机处于同一网络段中，彼此可以通过各自IP相互访问。

**虚拟机之间：**可以相互访问。

## 4.3 Internal模式

内网模式，虚拟机与外网完全断开，只实现虚拟机于虚拟机之间的内部网络模式。

**虚拟机与主机：**不能相互访问，彼此不属于同一个网络，无法相互访问。

**虚拟机与其他主机：**不能相互访问。

**虚拟机之间：**可以相互访问，前提是在设置网络时，两台虚拟机设置同一网络名称。

## 4.4 Host-only Adapter模式

可以理解为Vbox在主机中模拟出一张专供虚拟机使用的网卡，所有虚拟机都是连接到该网卡上的，我们可以通过设置这张网卡来实现上网及其他很多功能，比如网卡共享、网卡桥接等。

**虚拟机与主机：**默认不能相互访问，双方不属于同一IP段，host-only网卡默认IP段为192.168.56.X 子网掩码为255.255.255.0，后面的虚拟机被分配到的也都是这个网段。通过网卡共享、网卡桥接等，可以实现虚拟机于主机相互访问。

**虚拟机其他主机：**默认不能相互访问，通过设置，可以实现相互访问。

**虚拟机之间：**默认可以相互访问，都是同处于一个网段。

## 4.5 总结

关于以上四种网络模式通信做如下总结 如表4-1所示。

表4-1 四种网络模式通信

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | NAT | Bridged Adapter | Internal | Host-Only Adapter |
| 虚拟机到主机 | 1 | 1 | 0 | 1 |
| 主机到虚拟机 | 0 | 1 | 0 | 1 |
| 虚拟机到其他主机 | 1 | 1 | 0 | 默认不可以 |
| 其他主机到虚拟机 | 0 | 1 | 0 | 默认不可以 |
| 虚拟机之间 | 0 | 1 | 同一个网段可以 | 1 |

# 构建网络拓扑环境

## 5.1 物理硬件配置

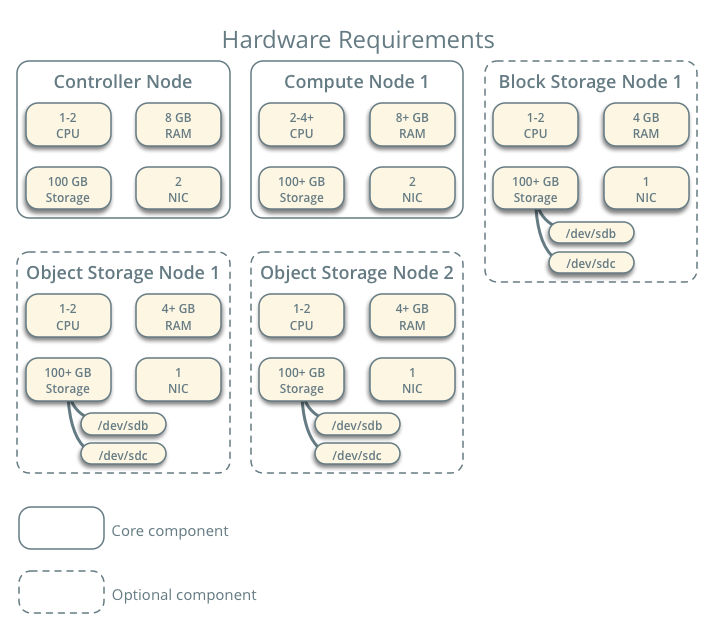


图5-1 物理硬件配置

## 5.2 节点服务

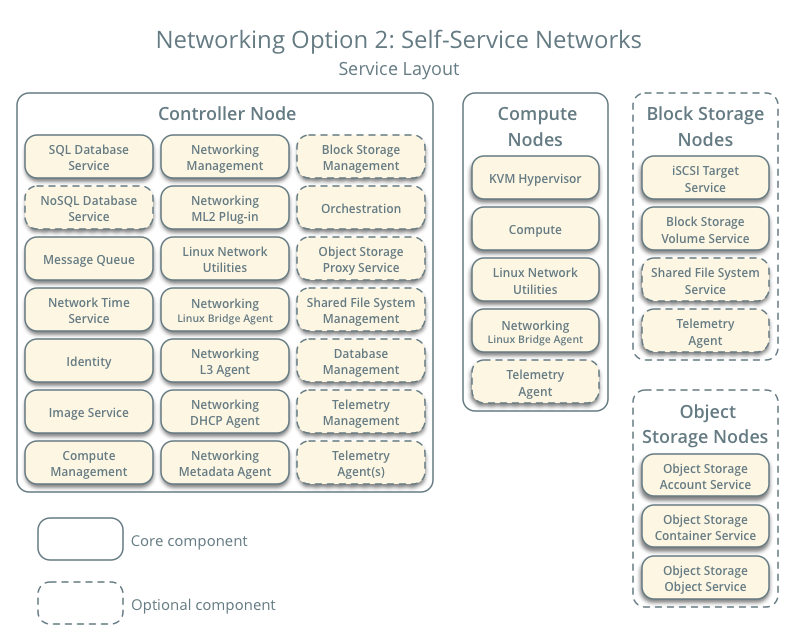


图5-2 节点服务

## 5.3 网络架构图

图5-3 网络拓扑图

管理/存储/互联网络为：192.168.237.0/24

数据网络：172.16.0/24 # vxlan隧道网络

本地网络：br-exteral

# 构建基础环境

## 6.1关闭firewalld

systemctl stop firewalld

systemctl disable firewalld

## 6.2 修改selinux

setenforce 0 # 临时关闭

vim /etc/sysconfig/selinux

selinux=enforcing 改为：selinux=disabled

## 6.3 关闭NetworkManager

systemctl stop NetworkManager

systemctl disable NetworkManager

**6.4 改/etc/hosts**

10.1.1.11 controller

10.1.1.21 compute01

10.1.1.22 compute02

10.1.1.31 block01

192.168.237.161 object01

192.168.237.162 object02

## 6.5 安装ntp服务

控制节点：

yum -y install chrony

修改配置文件：vim /etc/chrony.conf

allow 192.168.237.0/24

systemctl enable chronyd.service

systemctl start chronyd.service

another node:

yum install -y chrony

修改配置文件： vim /etc/chrony.conf

server controller iburst

systemctl enable chronyd.service && systemctl start chronyd.service

验证：

控制节点：

chronyc sources

[root@controller01 ~]# chronyc sources

210 Number of sources = 4

MS Name/IP address Stratum Poll Reach LastRx Last sample

===============================================================================

^+ xk-6-95-a8.bta.net.cn 2 6 17 3 -397us[ -623us] +/- 46ms

^\* 120.25.115.20 2 6 17 3 -1036us[-1262us] +/- 24ms

^- 203.107.6.88 2 6 17 3 -300us[ -300us] +/- 10ms

^? static-5-103-139-163.ip.> 0 7 0 - +0ns[ +0ns] +/- 0ns

计算节点：

[root@compute01 ~]# chronyc sources

210 Number of sources = 1

MS Name/IP address Stratum Poll Reach LastRx Last sample

===============================================================================

^\* controller01 3 6 17 5 +36us[ +80us] +/- 23ms

## 6.6 配置openstack源

参考配置文件 /etc/yum.repos.d/文件

或者安装

yum install epel-release

yum install centos-release-openstack-queens

yum upgrade –y

yum install openstack-selinux

yum install python-openstackclient

## 6.7 安装SQL database

控制节点：

yum install –y mariadb mariadb-server python2-PyMySQL

修改配置文件：vim /etc/my.cnf.d/openstack.cnf

[mysqld]

bind-address = 10.1.1.11 # 控制节点的管理IP

default-storage-engine = innodb

innodb\_file\_per\_table = on

max\_connections = 4096

collation-server = utf8\_general\_ci

character-set-server = utf8

服务启动并加入开启启动：

systemctl enable mariadb.service && systemctl start mariadb.service

初始化mysql的设置

mysql\_secure\_installation

## 6.9 安装消息队列

控制节点：

yum install rabbitmq-server

服务启动并加入开启启动：

systemctl enable rabbitmq-server.service && systemctl start rabbitmq-server.service

添加openstack用户：

rabbitmqctl add\_user openstack openstack

为openstack用户添加读写权限：

rabbitmqctl set\_permissions openstack ".\*" ".\*" ".\*"

## 6.10 安装memcached

控制节点：

yum install memcached python-memcached

编辑配置文件：vim /etc/sysconfig/memcached

OPTIONS="-l 127.0.0.1,::1,controller1"

服务启动并设置开启启动

systemctl enable memcached.service && systemctl start memcached.service

## 6.11 安装etcd服务

控制节点：

yum -y install etcd

修改配置文件：vim /etc/etcd/etcd.conf

#[Member]

ETCD\_DATA\_DIR="/var/lib/etcd/default.etcd"

ETCD\_LISTEN\_PEER\_URLS="http://controller:2380" ETCD\_LISTEN\_CLIENT\_URLS="http://controller:2379"

ETCD\_NAME="controller"

#[Clustering]

ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS="http://controller:2380" ETCD\_ADVERTISE\_CLIENT\_URLS="http://controller:2379" ETCD\_INITIAL\_CLUSTER="controller1=http://controller:2380" ETCD\_INITIAL\_CLUSTER\_TOKEN="etcd-cluster-01" ETCD\_INITIAL\_CLUSTER\_STATE="new"

服务启动并设置开机启动

systemctl enable etcd && systemctl start etcd

# 7. Keystone组件

## Keystone的架构

Keystone作为Openstack的核心模块，为Nova(计算),Glance(镜像),Swift(对象存储),Cinder(块存储),Neutron(网络)以及Horizon(Dashboard)提供认证服务。Keystone架构如图8-1所示。



图7-1 keystone的架构

如图4-2所示，除了涉及到Keystone模块之外，还有KeystoneMiddleware子项目，KeystoneMiddleware是对Keystone提供的对令牌合法性进行验证的中间件，缓存了相关的证书和令牌信息。

Keystone项目本身，除了后端的数据库，主要包含了一个处理RestFful请求的API服务进程，这些API涵盖了Identity、Token、Catalogy和Plolicy提供的各种各种服务,不同的服务都是有相应的后端Driver（BackendDriver）来实现。

## 7.2 Keystone的基本概念

* **User**

　　User即用户，他们代表可以通过keystone进行访问的人或程序。Users通过认证信息（credentials，如密码、API Keys等）进行验证。

* **Tenant**

　　Tenant即租户，它是各个服务中的一些可以访问的资源集合。例如，在Nova中一个tenant可以是一些机器，在Swift和Glance中一个tenant可以是一些镜像存储，在Neutron中一个tenant可以是一些网络资源。Users默认的总是绑定到某些tenant上。

* **Role**

　　Role即角色，Roles代表一组用户可以访问的资源权限，例如Nova中的虚拟机、Glance中的镜像。Users可以被添加到任意一个全局的或租户的角色中。在全局的role中，用户的role权限作用于所有的租户，即可以对所有的租户执行role规定的权限；在租户内的role中，用户仅能在当前租户内执行role规定的权限。

* **Service**

Service即服务，如Nova、Glance、Swift。根据前三个概念（User，Tenant和Role）一个服务可以确认当前用户是否具有访问其资源的权限。但是当一个user尝试着访问其租户内的service时，他必须知道这个service是否存在以及如何访问这个service，这里通常使用一些不同的名称表示不同的服务。

* **Endpoint**

Endpoint，翻译为“端点”，我们可以理解它是一个服务暴露出来的访问点，如果需要访问一个服务，则必须知道他的endpoint。因此，在keystone中包含一个endpoint模板，这个模板提供了所有存在的服务endpoints信息。一个endpoint template包含一个URLs列表，列表中的每个URL都对应一个服务实例的访问地址，并且具有public、private和admin这三种权限。public url可以被全局访问（如http://compute.example.com），private url只能被局域网访问（如http://compute.example.local），admin url被从常规的访问中分离。

* **Token**

Token是访问资源的钥匙。它是通过Keystone验证后的返回值，在之后的与其他服务交互中只需要携带Token值即可。每个Token都有一个有效期，Token只在有效期内是有效的。

## 7.3 上述概念之间的关系

各种概念之间的关系如图7-2所示。

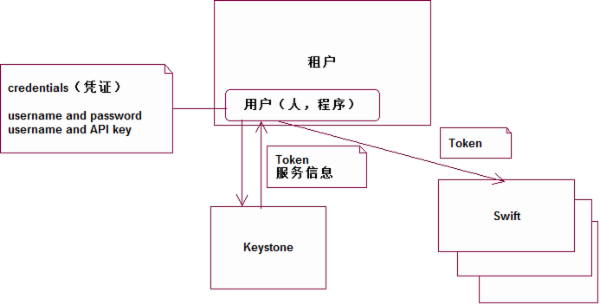


图7-2 各个概念之间的关系

1、租户下，管理着一堆用户（人，或程序）。

2、每个用户都有自己的credentials（凭证）用户名+密码或者用户名+API key，或其他凭证。

3、用户在访问其他资源（计算、存储）之前，需要用自己的credential去请求keystone服务，获得验证信息（主要是Token信息）和服务信息（服务目录和它们的endpoint）。

4、用户拿着Token信息，就可以去访问特点的资源了。

## 7.4 keystone在Openstack中的工作流程实例

Keystone在OpenStack中的工作流程如图7-3所示。



图7-3 keystone在OpenStack中的工作流程

## 实战：Keystone的手动搭建

### 7.5.1 控制节点的安装配置

数据库相关操作：

mysql -uroot -popenstack <<EOF

create database keystone;

GRANT ALL PRIVILEGES ON keystone.\* TO 'keystone'@'localhost' IDENTIFIED BY 'openstack'; GRANT ALL PRIVILEGES ON keystone.\* TO 'keystone'@'%' IDENTIFIED BY 'openstack';

EOF

安装相关包并配置

yum install openstack-keystone httpd mod\_wsgi -y

vim /etc/keystone/keystone.conf

[database]

connection = mysql+pymysql://keystone:openstack@controller/keystone

[token]

provider = fernet

初始化数据库，创建数据结构：

su -s /bin/sh -c "keystone-manage db\_sync" keystone

初始化fernet密钥存储库

keystone-manage fernet\_setup --keystone-user keystone --keystone-group keystone

keystone-manage credential\_setup --keystone-user keystone --keystone-group keystone

创建keystone的服务端口：

keystone-manage bootstrap --bootstrap-password openstack --bootstrap-admin-url [http://controller1:35357/v3/](http://controller:35357/v3/) --bootstrap-internal-url [http://controller1:5000/v3/](http://controller:5000/v3/) --bootstrap-public-url [http://controller1:5000/v3/](http://controller:5000/v3/) --bootstrap-region-id RegionOne

配置http 服务

vim /etc/httpd/conf/httpd.conf

ServerName controller

给/usr/share/keystone/wsgi-keystone.conf创建一个软连接

ln -s /usr/share/keystone/wsgi-keystone.conf /etc/httpd/conf.d/

服务启动并设置开机自启

systemctl enable httpd.service && systemctl start httpd.service

创建管理员账号

vim admin-openrc

export OS\_USERNAME=admin

export OS\_PASSWORD=f4mtdycd

export OS\_PROJECT\_NAME=admin

export OS\_USER\_DOMAIN\_NAME=Default

export OS\_PROJECT\_DOMAIN\_NAME=Default

export OS\_AUTH\_URL=http://controller: 35357/v3

export OS\_IDENTITY\_API\_VERSION=3

### 7.5.2 创建域、项目、用户和角色

openstack project create --domain default --description "Service Project" service

openstack project create --domain default --description "Demo Project" demo

下面这句要分开执行，要输入密码：

openstack user create --domain default --password-prompt demo

openstack role create user

openstack role add --project demo --user demo user

### 7.5.3 验证

unset OS\_AUTH\_URL OS\_PASSWORD

下面这句要分开执行，要输入admin用户的密码：

openstack --os-auth-url http://controller1:35357/v3 \

--os-project-domain-name Default \

--os-user-domain-name Default \

--os-project-name admin \

--os-username admin token issue

下面这句要分开执行，要输入demo用户的密码：

openstack --os-auth-url http://controller1:5000/v3 \

--os-project-domain-name Default \

--os-user-domain-name Default \

--os-project-name demo --os-username demo token issue

3.4 创建openstack客户端的脚本环境

创建管理员的账号

vim admin-openrc

export OS\_PROJECT\_DOMAIN\_NAME=Default

export OS\_USER\_DOMAIN\_NAME=Default

export OS\_PROJECT\_NAME=admin

export OS\_USERNAME=admin

export OS\_PASSWORD=openstack

export OS\_AUTH\_URL=http://controller:35357/v3

export OS\_IDENTITY\_API\_VERSION=3

export OS\_IMAGE\_API\_VERSION=2

创建demo用户的账号

export OS\_PROJECT\_DOMAIN\_NAME=Default

export OS\_USER\_DOMAIN\_NAME=Default

export OS\_PROJECT\_NAME=demo

export OS\_USERNAME=demo

export OS\_PASSWORD=openstack

export OS\_AUTH\_URL=http://controller:5000/v3

export OS\_IDENTITY\_API\_VERSION=3

export OS\_IMAGE\_API\_VERSION=2

使用脚本验证

source admin-openrc

openstack token issue

可获取token admin用户的token

source demo-openrc

openstack token issue

可以获取token demo用户的token

# 8.Glance组件

## 8.1 定义

Glance image services include **discovering**, **registering**, and retrieving virtual machine images. Glance has a **RESTful API** that allows querying of VM image **metadata** as well as **retrieval of the actual image**.

Image service项目代号Glance，是[OpenStack](http://lib.csdn.net/base/openstack" \t "_blank" \o "OpenStack知识库)的镜像服务组件。Glance主要提供了一个虚拟机镜像文件的存储、查询和检索服务，通过提供一个虚拟磁盘映像目录和存储库，为Nova的虚拟机提供镜像服务，现在Glance具有V1和V2两个版本。

## 8.2 glance架构

glance的架构如图8-1所示。

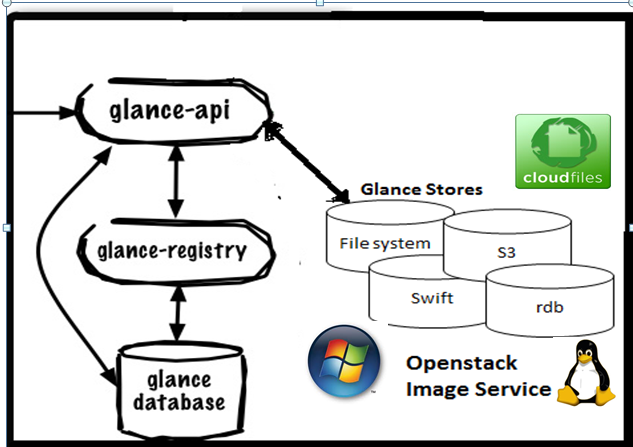


图8-1 glance的架构

**镜像服务组件：**

* **Glance-api**：是一个对外的API接口，能够接受外部的API镜像请求。主要用于分析、分发、响应各种镜像管理的REST Request，然后通过其他模块(EG. glance-registry、Store Backend后端存储接口)完成镜像的发现、获取、存储等操作。默认绑定端口是**9292。**
* glance-registry：用于存储、处理、获取Image Metadata。通过响应从glance-api发送过来的Image Metadata REST Request，然后与[MySQL](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)进行交互，实现Image Metadate的存储、处理、获取。默认绑定的端口是9191。
* glance-db：在Openstack中使用MySQL来支撑，用于存放Image Metadata。   
  Image Metadate(镜像元数据)：指通过glance-registry来保存在MySQL Database。
* Image Store：用于存储镜像文件。通过Store Backend后端存储接口来与glance-api联系。通过这个接口，glance可以从Image Store获取镜像文件再交由Nova用于创建虚拟机。
* Glance 通过Store Adapter(存储适配器)支持多种Imange Store方案

支持swift、file system、s3、sheepdog、rbd、cinder等。

## 8.3 Glance支持的Image格式

* raw – 非结构化的镜像格式
* vhd – 一种通用的虚拟机磁盘格式， 可用于Vmware、Xen、Microsoft Virtual PC/Virtual Server/Hyper-V、VirtualBox等
* vmdk – Vmware的虚拟机磁盘格式， 同样也支持多种Hypervisor
* vdi – VirtualBox、QEMU等支持的虚拟机磁盘格式
* qcow2 – 一种支持QEMU并且可以动态扩展的磁盘格式
* aki – Amazon Kernel 镜像
* ari – Amazon Ramdisk 镜像
* ami – Amazon 虚拟机镜像

## 8.4 glance的访问权限

* Public 公共的：可以被所有的Tenant使用。
* Private 私有的/项目的：只能被Image Owner所在的Tenant使用。
* Shared 共享的：一个非公共的Image可以共享给指定的Tenant，通过member-\*操作来实现。
* Protected 受保护的：Protected Image不能被删除。

## 8.5 状态类型及转化

* Queued：没有上传Image数据，只SQL Database中存有该镜像的元数据。
* Saving：正在上传Image。
* Active：正常状态。
* Deleted/pending\_delete： 已删除/等待删除的Image。
* Killed：Image元数据不正确，等待被删除。

状态的转化

* ‘queued’ => (‘saving’, ‘active’, ‘deleted’)
* ‘saving’ => (‘active’, ‘killed’, ‘deleted’, ‘queued’)
* ‘active’ => (‘queued’, ‘pending\_delete’, ‘deleted’)
* ‘killed’ => (‘deleted’)
* ‘pending\_delete’ => (‘deleted’)
* ‘deleted’ => ()

## 8.6 两种API的区别

### 8.6.1 glance API V1

Glance V1架构如图 8-2 所示。



图8-2 glance v1的架构

**V1的功能**：提供了基本的Image和Member操作   
1. 镜像文件的创建、删除、查询、更改   
2. 镜像Tenant成员的创建、删除和查询

**V1**包含有glance-api和glance-registry两个WSGI service，都提供了REST API接口来接收虚拟机镜像管理的请求.

**需要注意的是：glance-api 不会真正去处理REST Request，可以将glance-api再分为两部分：**

* **一部分是中间件，主要用于对REST Request的分析、分发工作。**
* **另一部分来提供实际的服务Store Backend后端存储接口交互，实现镜像上传、下载)**

所以若glance-api接收到涉及SQL Database的操作请求时，会调用registry-client并生成HTTP指令，然后转发给glance-registry API进行处理。

### 8.6.2 glance API V2

Glance V1架构如图 8-3所示。



图8-3 所示glance v2的架构

**V2的功能**：除了拥有V1的功能之外，还能够：   
1. 镜像Location的添加、删除和修改   
2. Metadata、Namespace、Image tag操作

**V2**在实现上，把glance-registry和glance-api合并到了一起，减少了一个中间环节。

## 实战： glance的手动搭建

### 8.7.1 控制节点的安装配置

数据库相关操作：

mysql -uroot -pf4mtdycd <<EOF

create database glance;

GRANT ALL PRIVILEGES ON glance.\* TO 'glance'@'localhost' IDENTIFIED BY 'openstack'; GRANT ALL PRIVILEGES ON glance.\* TO 'glance'@'%' IDENTIFIED BY 'openstack';

EOF

创建glance用户，并在service项目中添加管理员角色

source admin\_openrc

下面这句要分开执行，要输入glance用户的密码：

openstack user create --domain default --password-prompt glance

openstack role add --project service --user glance admin

创建glance服务及端口

openstack service create --name glance --description "OpenStack Image" image

openstack endpoint create --region RegionOne image public http://controller1:9292

openstack endpoint create --region RegionOne image internal http://controller1:9292

openstack endpoint create --region RegionOne image admin [http://controller1:9292](http://controller:9292)

安装相关包并配置

yum install openstack-glance

vim /etc/glance/glance-api.conf

[database]

connection = mysql+pymysql://glance:openstack@controller1/glance

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = glance

password = openstack

[paste\_deploy]

flavor = keystone

[glance\_store]

stores = file,http

default\_store = file

filesystem\_store\_datadir = /var/lib/glance/images/

vim /etc/glance/glance-registry.conf

[database]

connection = mysql+pymysql://glance:openstack@controller1/glance

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = glance

password = openstack

[paste\_deploy]

flavor = keystone

初始化数据库表结构

su -s /bin/sh -c "glance-manage db\_sync" glance

服务启动并设置开启自启

systemctl enable openstack-glance-api.service openstack-glance-registry.service && systemctl start openstack-glance-api.service openstack-glance-registry.service

### 8.7.2 验证

source admin-openrc

下载实验镜像cirros

wget http://download.cirros-cloud.net/0.3.5/cirros-0.3.5-x86\_64-disk.img

创建镜像：

openstack image create "cirros" --file cirros-0.3.5-x86\_64-disk.img --disk-format qcow2 --container-format bare --public

openstack image list

# 9.Nova组件

## 9.1 nova的定义

Use OpenStack Compute to **host** and **manage** cloud computing systems. OpenStack Compute is **a major part of** an Infrastructure-as-a-Service ([IaaS](http://docs.openstack.org/mitaka/install-guide-rdo/common/glossary.html" \l "term-iaas)) system. The main modules are implemented in Python.

OpenStack Compute interacts with OpenStack Identity for authentication; OpenStack Image service for disk and server images; and OpenStack dashboard for the user and administrative interface. Image access is limited by projects, and by users; quotas are limited per project (the number of instances, for example). OpenStack Compute can **scale horizontally** on standard hardware, and download images to launch instances.

Nova是openstack中最核心的组件。openstack的其他组件归根结底是为Nova组件服务的，基于用户需求为VM提供计算资源管理。

## **9.2 Nova的架构**

Openstack中Nova的架构如图9-1所示。



图9-1 Nova架构

目前的Nova主要由API、Compute、Conductor、Scheduler四个核心服务组成，他们之间通过AMQP通信。 API是进入Nova的HTTP接口。Compute和VMM（虚拟机管理器）交互来运行虚拟机并管理虚拟机的生命周期（通常是一个主机一个compute服务）。Scheduler从可用池中选择最合适的节点来创建虚拟机实例。

## 9.3 Nova 的逻辑模块

Nova服务是由多个子服务构成，子服务是通过RPC实现通信。服务之间有很松的耦合性。各个逻辑模块如下：

1.Nova API ：HTTP服务，用于接收和处理客户端发送的HTTP请求

2.Nova Cell ：Nova Cell子服务的目的便于实现横向扩展和大规模的部署，同时不增加[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)和RPC消息中间件的复杂度。在Nova Scheduler服务的主机调度的基础上实现了区域调度。

3.Nova Cert ：用于管理证书，为了兼容AWS。AWS提供一整套的基础设施和应用程序服务，使得几乎所有的应用程序在云上运行。

4.Nova Compute ：Nova组件中最核心的服务，**实现虚拟机管理的功能**。实现了在计算节点上创建、启动、暂停、关闭和删除虚拟机、虚拟机在不同的计算节点间迁移、虚拟机安全控制、管理虚拟机磁盘镜像以及快照等功能。

5.Nova Conductor ：RPC服务，**主要提供数据库查询功能**。以前的openstack版本中，Nova Compute子服务中定义了许多的数据库查询方法。但是，由于Nova Compute子服务需要在每个计算节点上启动，一旦某个计算节点被攻击，就将完全获得数据库的访问权限。有了Nova Compute子服务之后，便可在其中实现**数据库访问权限的控制**。

6.Nova Scheduler ：Nova调度子服务。当客户端向Nova 服务器发起创建虚拟机请求时，决定将虚拟机创建在哪个节点上。

7.Nov Console、Nova Consoleauth、Nova VNCProxy ：Nova控制台子服务。功能是实现客户端通过代理服务器远程访问虚拟机实例的控制界面。

## 9.4 Nova启动一个虚拟机的过程

nova 启动虚拟机的过程如图9-2所示。



图9-2 nova启动虚拟机过程

1. 界面或者命令行通过RESTful API向keystone发送认证信息。
2. Keystone通过用户认证消息，并生成auth\_token返回给对应的认证请求。
3. 界面或者命令行通过RESTful API向nova API发送一个boot instance的请求（携带auth-token）
4. nova-api接受请求后，向keystone发送认证请求，查看token是否为有效用户的token。
5. keystone验证token是否有效，如有效，则返回有效的认证和对应的角色（备注：有些操作需要有角色权限才能操作）
6. 通过认证后，nova-api和数据库通信。
7. 初始化新建虚拟机的数据库记录。
8. nova-api通过rpc.call向nova-scheduler请求是否有创建虚拟机的资源（Host ID）。
9. nova-schduler进程监听消息队列，获取nova-api的请求。
10. nova-scheduler通过nova数据库中计算资源的的情况，并通过调度算法计算符合虚拟机创建需要的的主机。
11. 对于有符合虚拟机创建的主机，nova-scheduler更新数据库虚拟机对应的物理主机信息。
12. nova-scheduler通过rpc.cast向nova-compute发送对应的创建虚拟机请求的消息。、
13. nova-compute会从对应的消息队列中获取创建虚拟机请求的消息。
14. nova-compute通过rpc.call向nova-conductor请求获取虚拟机消息（flavor）。
15. nova-conductor从消息队列中拿到nova-compute请求信息。
16. nova-conductor根据消息查询虚拟机对应的消息。
17. nova-conductor 从数据库中获取虚拟机对应信息。
18. nova-conductor把虚拟机信息通过消息的方式发送的消息队列中。
19. nova-compute从对应的消息队列中获取虚拟节信息消息。
20. nova-compute通过keystone的RESTfull API拿到的认证的token，并通过HTTP请求glance-api获取创建虚拟机所需要的镜像。
21. glance-api向keystone认证token是否有效，并返回验证结果。
22. token验证通过，nova-compute获取虚拟机镜像信息（URL）。
23. nova-compute通过keystone的RESTfull API拿到的认证的token，并通过HTTP请求neutron-server获取创建虚拟机所需要的网络信息。
24. neutron-server向keystone认证token是否有效，并返回验证结果。
25. token验证通过，nova-compute获取网络信息。
26. nova-compute通过keystone的RESTfull API拿到的认证的token，并通过HTTP请求cinder-api获取创建虚拟机所需要的持久化存储信息。
27. cinder-api向keystone认证token是否有效，并返回验证结果。
28. token验证通过，nova-compute获取虚拟机持久化存储信息。
29. nova-compute根据instance的信息调用配置的虚拟化驱动来创建虚拟机。

## 9.5 Nova Scheduler Filter的类型

选择一个虚拟机在哪个主机运行的方式有多种，nova支持的方式主要有以下三种：

1. ChanceScheduler(随机调度器)：从所有nova-compute服务正常运行的节点中随机选择。
2. FilterScheduler(过滤调度器)：根据指定的过滤条件以及权重挑选最佳节点。
3. CachingScheduler：FilterScheduler一种，在FileterScheduler的基础上，将主机资源的信息存到本地的内存中，然后通过后台的定时任务定时的从数据库中获取最新的主机资源信息。

Nova Scheduler的工作流程如图9-3 所示：



图9-3 nova-scheduler的工作流程

FilterScheduler首先使用指定的Fileters(过滤器)得到符合条件的主机，比如内存小于50%，然后对得到的主机重新计算权重并且排列，获取最佳的一个。具体的Filter有如下几种：

1. RetryFilter：重试过滤，假设Host1，Host2，Host3过滤筛选出来了，Host1权重最高，被选中，由于其它原因VM在Host1上落地失败，nova-scheduler会重新筛选新的host;Host1因为失败不会再入选。可通过scheduler\_max\_attempts=3 设置重试的次数。
2. AvalilabilityZoneFilter 可选域过滤，可以提供容灾性和隔离服务，计算节点可以纳入一个创建好的AZ中，创建VM的时候可以指定AZ，这样虚拟机会落到指定的host中。
3. RamFilter 内存过滤 创建VM时会选择flavor，不满足flavor中内存要求的host会过滤掉。可以通过linux里面的free来查看，超量使用的设置：ram\_allocation\_ratio=3(如果计算节点有16G内存，那个openstack会认为有48G内存)。
4. CoreFilter CPU core过滤 创建VM时会选择flavor，不满足flavor中core要求的host会过滤掉。CPU的超量设置 cpu\_allocaion\_ratio=16.0(若计算节点为24core，那么openstack会认为348core)
5. DiskFilter 磁盘容量过滤 创建VM时会选择flavor，不满足flavor中磁盘要求的host会过滤掉。Disk超量使用设置：disk\_allocation\_ratio=1.0(硬盘容量不建议调大)
6. ComputeFilter nova-compute服务的正常过滤 创建VM时，若host的nova-compute的服务不正常，那么就会别筛掉。
7. ComputeCababilitiesFilter 根据计算节点的特性来筛选。例如x86\_64
8. ImagePropertiesFilter 根据所选的image的属性来匹配计算节点 例如希望某个image只能运行在KVM的hypervisor上，可以通过“Hypervisor Type”属性来指定。
9. ServerGroupAntiAffinityFilter 尽量将Instance部署到不同的节点上。例如vm1，vm2，vm3，计算节点有Host1，host2，host3。

创建一个anti-affinity策略 server group “group-1”

nova server-group-create –policy anti-affinity group-1

nova boot-image IMAGE\_ID –flavor 1 –hint group-group1 vm1

nova boot-image IMAGE\_ID –flavor 1 –hint group-group1 vm2

nova boot-image IMAGE\_ID –flavor 1 –hint group-group1 vm3

1. ServerGroupAffinityFilter 尽量将Instance部署到同一节点上。例如vm1，vm2，vm3，计算节点有Host1，host2，host3。

创建一个group-affinity策略 server group “group-2”

nova server-group-create –policy anti-affinity group-2

nova boot-image IMAGE\_ID –flavor 1 –hint group-group2 vm1

nova boot-image IMAGE\_ID –flavor 1 –hint group-group2 vm2

nova boot-image IMAGE\_ID –flavor 1 –hint group-group2 vm3

## 9.6 实战：nova的手动搭建

### Controller节点

数据库相关操作：

mysql -uroot -popenstack <<EOF

create database nova\_api;

create database nova;

create database nova\_cell0;

GRANT ALL PRIVILEGES ON nova\_api.\* TO 'nova'@'localhost' IDENTIFIED BY 'openstack'; GRANT ALL PRIVILEGES ON nova\_api.\* TO 'nova'@'%' IDENTIFIED BY 'openstack';

GRANT ALL PRIVILEGES ON nova.\* TO 'nova'@'localhost' IDENTIFIED BY 'openstack'; GRANT ALL PRIVILEGES ON nova.\* TO 'nova'@'%' IDENTIFIED BY 'openstack';

GRANT ALL PRIVILEGES ON nova\_cell0.\* TO 'nova'@'localhost' IDENTIFIED BY 'openstack'; GRANT ALL PRIVILEGES ON nova\_cell0.\* TO 'nova'@'%' IDENTIFIED BY 'openstack';

EOF

创建nova用户，并在service项目中添加管理员角色

source admin\_openrc

下面这句要分开执行，要输入nova用户的密码：

openstack user create --domain default --password-prompt nova

openstack role add --project service --user nova admin

创建nova服务及端口

openstack service create --name nova --description "OpenStack Compute" compute

openstack endpoint create --region RegionOne compute public http://controller1:8774/v2.1

openstack endpoint create --region RegionOne compute internal http://controller1:8774/v2.1

openstack endpoint create --region RegionOne compute admin http://controller1:8774/v2.1

创建placement用户，并在service项目中添加管理员角色

source admin\_openrc

下面这句要分开执行，要输入nova用户的密码：

openstack user create --domain default --password-prompt placement

openstack role add --project service --user placement admin

创建placement服务及端口

openstack service create --name placement --description "Placement API" placement

openstack endpoint create --region RegionOne placement public http://controller1:8778

openstack endpoint create --region RegionOne placement internal http://controller1:8778

openstack endpoint create --region RegionOne placement admin http://controller1:8778

安装相关包并配置

yum install openstack-nova-api openstack-nova-conductor openstack-nova-console openstack-nova-novncproxy openstack-nova-scheduler openstack-nova-placement-api

vim /etc/nova/nova.conf

[DEFAULT]

enabled\_apis = osapi\_compute,metadata

[api\_database]

connection = mysql+pymysql://nova:openstack@controller1/nova\_api

[database]

connection = mysql+pymysql://nova:openstack@controller1/nova

[DEFAULT]

transport\_url = rabbit://openstack:openstack@controller1

[api]

auth\_strategy = keystone

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = nova

password = openstack

[DEFAULT]

my\_ip = 192.168.237.131

[DEFAULT]

use\_neutron = True

firewall\_driver = nova.virt.firewall.NoopFirewallDriver

[vnc]

enabled = true

vncserver\_listen =192.168.237.131

vncserver\_proxyclient\_address = 192.168.237.131

[glance]

api\_servers = http://controller1:9292

[oslo\_concurrency]

lock\_path = /var/lib/nova/tmp

[placement]

os\_region\_name = RegionOne

project\_domain\_name = Default

project\_name = service

auth\_type = password

user\_domain\_name = Default

auth\_url = [http://controller1:35357/v3](http://controller:35357/v3)

username = placement

password = openstack

编辑配置文件 vim /etc/httpd/conf.d/00-nova-placement-api.conf

<Directory /usr/bin>

<IfVersion >= 2.4>

Require all granted

</IfVersion>

<IfVersion < 2.4>

Order allow,deny

Allow from all

</IfVersion>

</Directory>

systemctl restart httpd

初始化nova\_api数据库表结构：

su -s /bin/sh -c "nova-manage api\_db sync" nova

返回如下错误：[root@controller ~]# su -s /bin/sh -c "nova-manage api\_db sync" nova

/usr/lib/python2.7/site-packages/oslo\_db/sqlalchemy/enginefacade.py:332: NotSupportedWarning: Configuration option(s) ['use\_tpool'] not supported

exception.NotSupportedWarning

修改文件：vim /usr/lib/python2.7/site-packages/oslo\_db/sqlalchemy/enginefacade.py

175 行，'db\_max\_retry\_interval', 'backend', 加入 'use\_tpool'即可。

加上：

注册cell0数据库

su -s /bin/sh -c "nova-manage cell\_v2 map\_cell0" nova

创建cell1：

su -s /bin/sh -c "nova-manage cell\_v2 create\_cell --name=cell1 --verbose" nova

初始化nova数据库的表结构：

su -s /bin/sh -c "nova-manage db sync" nova

验证cell0和cell1是否注册

nova-manage cell\_v2 list\_cells

服务启动并加入开机自启

systemctl enable openstack-nova-api.service openstack-nova-consoleauth.service openstack-nova-scheduler.service openstack-nova-conductor.service openstack-nova-novncproxy.service

systemctl start openstack-nova-api.service openstack-nova-consoleauth.service openstack-nova-scheduler.service openstack-nova-conductor.service openstack-nova-novncproxy.service

### 9.6.2 Compute节点

安装相关包并进行配置

yum install openstack-nova-compute

vim /etc/nova/nova.conf

[DEFAULT]

enabled\_apis = osapi\_compute,metadata

[DEFAULT]

transport\_url = rabbit://openstack:openstack@controller1

[api]

auth\_strategy = keystone

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = nova

password = openstack

[DEFAULT]

my\_ip = 192.168.237.141

[DEFAULT]

use\_neutron = True

firewall\_driver = nova.virt.firewall.NoopFirewallDriver

[vnc]

enabled = True

vncserver\_listen = 0.0.0.0

vncserver\_proxyclient\_address =192.168.237.141

novncproxy\_base\_url = http://192.168.237.131:6080/vnc\_auto.html

[glance]

api\_servers = http://controller1:9292

[oslo\_concurrency]

lock\_path = /var/lib/nova/tmp

[placement]

os\_region\_name = RegionOne

project\_domain\_name = Default

project\_name = service

auth\_type = password

user\_domain\_name = Default

auth\_url = [http://controller1:35357/v3](http://controller:35357/v3)

username = placement

password = openstack

egrep -c '(vmx|svm)' /proc/cpuinfo

若返回0

vim /etc/nova/nova.conf

[libvirt]

virt\_type = qemu

服务启动，并加入开启自启

systemctl enable libvirtd.service openstack-nova-compute.service && systemctl start libvirtd.service openstack-nova-compute.service

将compute节点添加到cell数据库

source admin-openrc

openstack compute service list --service nova-compute

su -s /bin/sh -c "nova-manage cell\_v2 discover\_hosts --verbose" nova

When you add new compute nodes, you must run nova-manage cell\_v2 discover\_hosts on the controller node to register those new compute nodes. Alternatively, you can set an appropriate interval in /etc/nova/nova.conf:

[scheduler]

discover\_hosts\_in\_cells\_interval = 300

### 9.6.3 验证

source admin-openrc

openstack compute service list

openstack catalog list

openstack image list

nova-status upgrade check

# 10.Neutron

## 10.1 概述

Quantum(Neutron的前身)在Openstack的Folsom（2012年9月发布）版本正式发布，其实它已经作为试用组件包含在之前的Essex版本中。在Grizzly里功能得到了增强。因为商标侵权的原因，Openstack在Havana版本上将Quantum更名为Neutron，所以Neutron并不是什么新的东西。在Havana版里，Neutron也只增加和增强了少数功能。

 Neutron是OpenStack中的一个项目，在各接口设备之间提供网络即服务(networking as a service),而且受其他OpenStack服务管理,如Nova。 具体来说，Neutron为OpenStack云更灵活地划分物理网络，在多租户环境下提供给每个租户独立的网络环境。另外，Neutron提供API来实现这种目标。Neutron中“网络”是一个可以被用户创建的对象，如果要和物理环境下的概念映射的话，这个对象相当于一个巨大的交换机，可以拥有无限多个动态可创建和销毁的虚拟端口。

## 10.2 网络虚拟化的原因

为了满足云计算的需求：1）数据中心的现有网络不能满足云计算的物理要求2）数据中心的现有网络不能满足云计算软件化即SDN（Software Define Network）的需求。

### 10.2.1 现有物理网络不能满足云计算的需求

  互联网行业数据中心的基本特征就是服务器的规模偏大。进入云计算时代后，其业务特征变得更加复杂，包括：虚拟化支持、多业务承载、资源灵活调度等（如图10-1所示）。与此同时，互联网云计算的规模不但没有缩减，反而更加庞大。这就给云计算的网络带来了巨大的压力。



图10-1 互联网云计算的业务特点

（1）**大容量的MAC表项和ARP表项**

虚拟化会导致更大的MAC表项。假设一个互联网云计算中心的服务器有5000台，按照1:20的比例进行虚拟化，则有10万个虚拟机。通常每个虚拟机会配置两个业务网口，这样这个云计算中心就有20万个虚拟网口，对应的就是需要20万个MAC地址和IP地址。云计算要求资源灵活调度，业务资源任意迁移。也就是说任意一个虚拟机可以在整个云计算网络中任意迁移。这就要求全网在一个统一的二层网络中。全网任意交换机都有可能学习到全网所有的MAC表项。与此对应的则是，目前业界主流的接入交换机的MAC表项只有32K，基本无法满足互联网云计算的需求。另外，网关需要记录全网所有主机、所有网口的ARP信息。这就需要网关设备的有效ARP表项超过20万。大部分的网关设备芯片都不具备这种能力。

（2）**4K VLAN Trunk问题**

传统的大二层网络支持任意VLAN的虚拟机迁移到网络的任意位置，一般有两种方式。方式一：虚拟机迁移后，通过自动化网络管理平台动态的在虚拟机对应的所有端口上下发VLAN配置；同时，还需要动态删除迁移前虚拟机对应所有端口上的VLAN配置。这种方式的缺点是实现非常复杂，同时自动化管理平台对多厂商设备还面临兼容性的问题，所以很难实现。方式二：在云计算网络上静态配置VLAN，在所有端口上配置VLAN trunk all。这种方式的优点是非常简单，是目前主流的应用方式。但这也带来了巨大的问题**：任一VLAN内如果出现广播风暴，则全网所有VLAN内的虚拟机都会受到风暴影响，出现业务中断**。

（3）**4K VLAN上限问题**

云计算网络中有可能出现多租户需求。如果租户及业务的数量规模超出**VLAN的上限（4K**），则无法支撑客户的需求。

（4）**虚拟机迁移网络依赖问题**

VM迁移需要在同一个二层域内，基于IP子网的区域划分限制了二层网络连通性的规模。

### 10.2.2 现有物理网络不能满足云计算SDN的需求

  数据中心（Data Center）中的物理网络是固定的、需要手工配置的、单一的、没有多租户隔离的网络。这是一个Csico给出的一个物理网络局部实例：

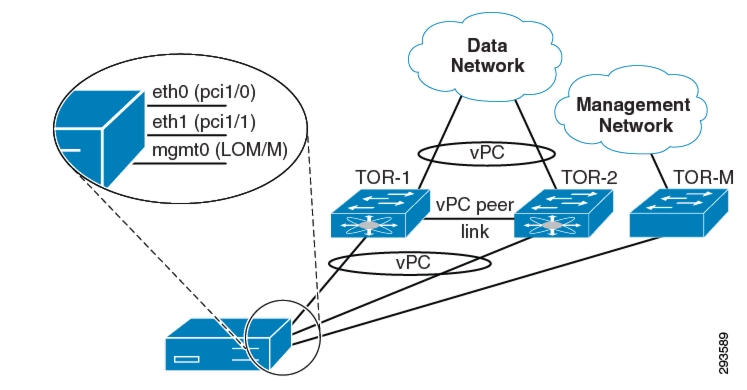


图10-2 物理网络局部实例

（图中的一个服务器有三块网卡，分别连到连接数据网络和管理网络的交换机。）

而云架构往往是多租户架构，这意味着多个客户会共享单一的物理网络。因此，除了提供基本的网络连接能力以外，云还需要提供网络在租户之间的隔离能力；同时云是自服务的，这意味着租户可以通过云提供的 API 来使用虚拟出的网络组建来设计，构建和部署各种他们需要的网络。

而云架构往往是多租户架构，这意味着多个客户会共享单一的物理网络。因此，除了提供基本的网络连接能力以外，云还需要提供网络在租户之间的隔离能力；同时云是自服务的，这意味着租户可以通过云提供的 API 来使用虚拟出的网络组件来设计，构建和部署各种他们需要的网络。MidoNet的网络架构如图10-3 所示。

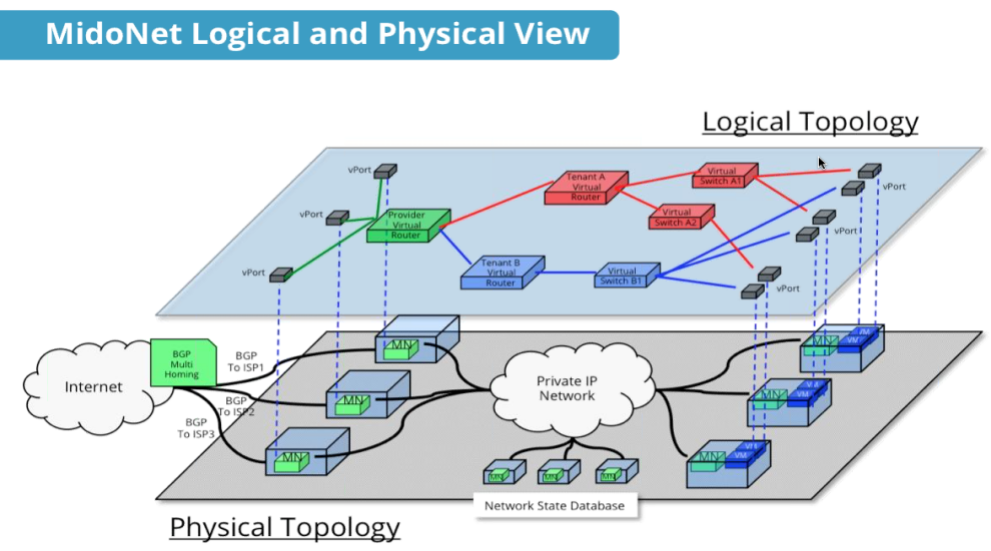


图10-3 MidoNet提供的逻辑网络和物理网络的映射图

MidoNet是类似于OpenContrail, Neutron DVR, DragonFlow, OVN的SDN产品，其主要特性如下：  
1, L2交换、L3路由、L4负载均衡  
2, 有状态和无状态NAT  
3，逻辑和分布式防火墙  
4, BGP与ECMP支持  
5, GRE、VxLAN  
6, VTEP(VXLAN Tunneling End Point) support with OVSDB protocol  
7, partial docker integration  
8, Re-written with Java/scala。

架构如图10-4 所示。



图10-4 MidoNet物理架构

在计算节点上要运行Midolman也就是MidoAgent来同时具备L2-L4层干活功能。采用分布式数据库NSDB来作集中的控制平面存储port, node, mac等映射关系。另外，它还支持BGP，如下图，它并不支持在同一个数据中心的多个跨三层的openstack之间运行iBGP（走GRE/VxLAN隧道），只支持在跨广域网的多个数据中心之间运行eBGP。如图10-5 所示。

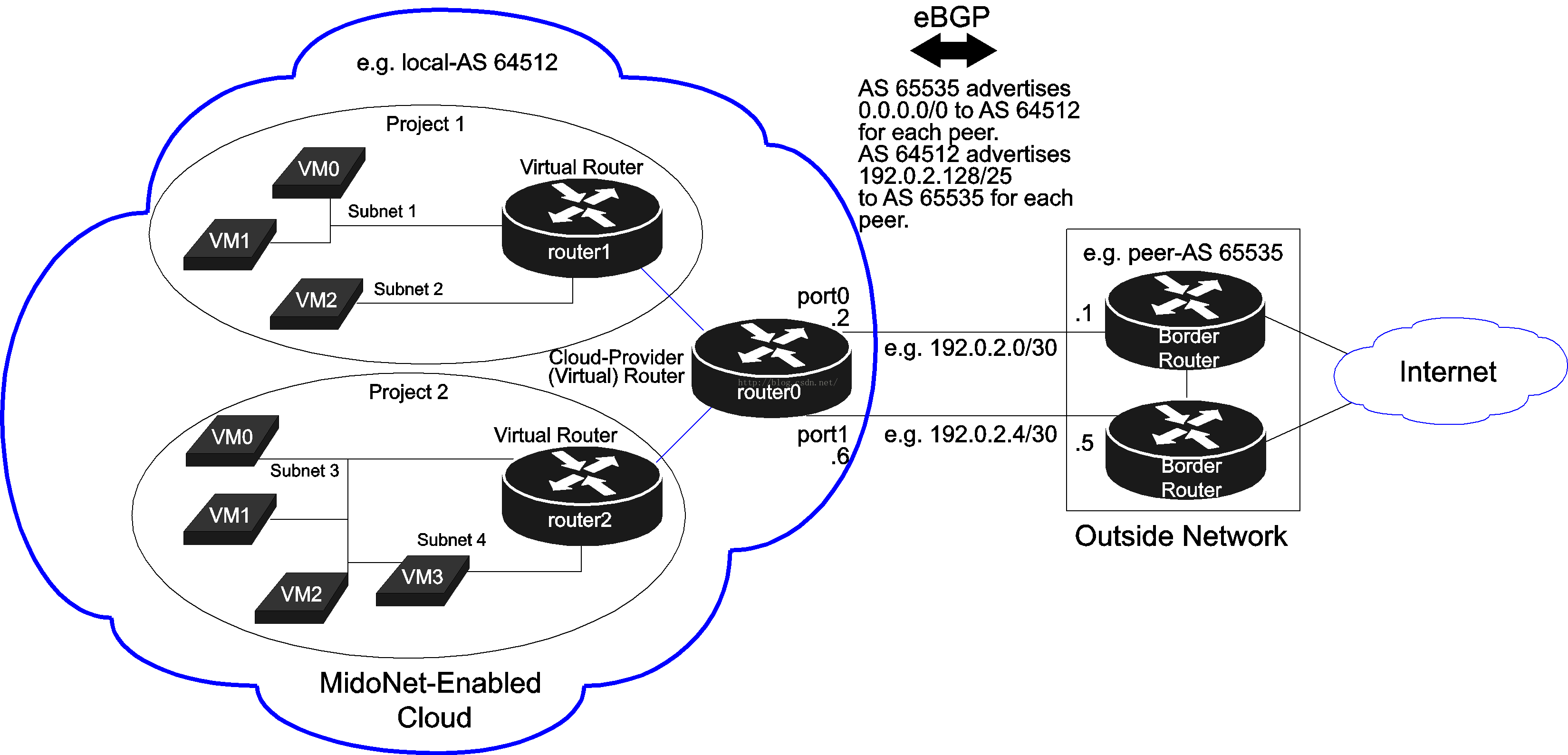


图10-5 MidoAgent支持的BGP协议

openStack 云也不例外。 OpenStack 通过 Neutron 项目在物理网络环境之上提供满足多租户要求的虚拟网络和服务。Neutron 提供的网络虚拟化能力包括：

（1）二层到七层网络的虚拟化：L2（virtual switch）、L3（virtual Router 和 LB）、L4-7（virtual Firewall ）等

（2）网络连通性：二层网络和三层网络

（3）租户隔离性

（4）网络安全性

（4）网络扩展性

（5）REST API

（6）更高级的服务，包括 LBaaS。

## 10.3 Neutron的架构

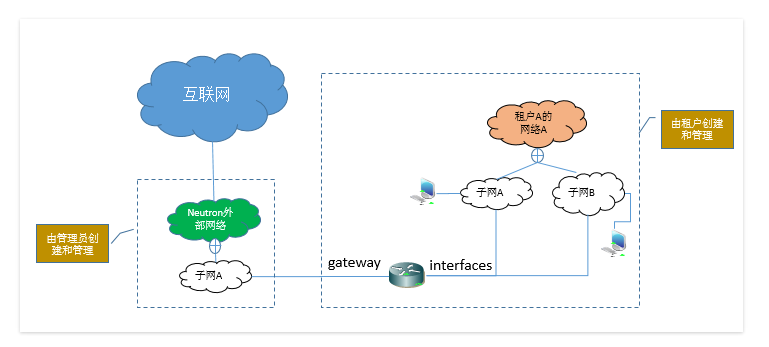


图10-6 OpenStack的架构

管理员创建和管理Neutron外部网络，是租户虚拟机与互联网信息交互的桥梁。更具体的，外部网络会分出一个子网，它是一组在互联网上可寻址的IP地址。一般情况下，外部网络只有一个（neutron是支持多个外部网络的），且由管理员创建。租户虚拟机创建和管理租户网络，每个网络可以根据需要划分成多个子网。诸多子网通过路由器与Neutron外部网络(图中具体是子网A)连接。路由器的gateway网关端连接外部网络的子网，interfaces接口端有多个，连接租户网络的子网。路由器及interface接口端连接的网络都是由租户根据需要自助创建，管理者只创建和管理Neutron外部网络部分。

总结来看，创建一个Neutron网络的过程如下：

       1 、首先管理员拿到一组可以在互联网上寻址的IP地址，并且创建一个外部网络和子网

       2、 租户创建一个网络和子网

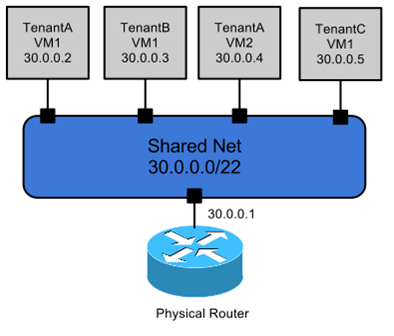
       3、 租户创建一个路由器并且连接租户子网和外部网络

       4、 租户创建虚拟机

## 10.4 Neutron支持的网络模型

在 OpenStack 网络组件没有独立出来之前，OpenStack 最初的 nova-network 网络模型，如图 10-7。

**图 10-7. 单一平面网络**



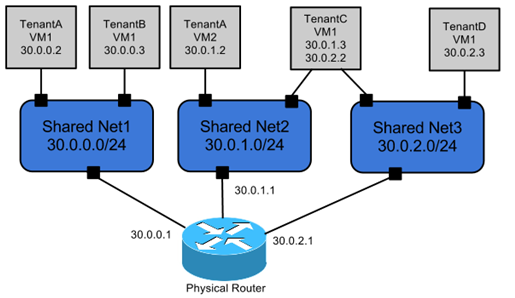
单一平面网络的缺点：

存在单一网络瓶颈，缺乏可伸缩性。

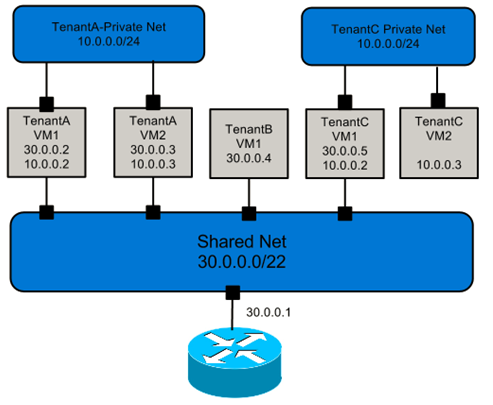
缺乏合适的多租户隔离。

OpenStack nova-network 独立成为单独的组件 Neutron 后，形象的网络模型的多平面网络、混合平面私有网络。如图 10-8，图10-9，图 10-10，图 10-11。

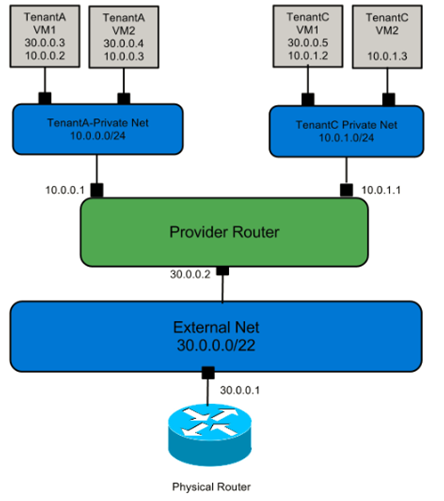
**图 10-8. 多平面网络**



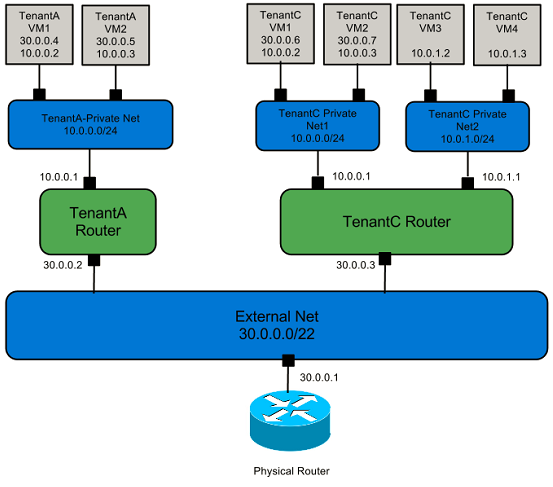
**图 10-9. 混合平面私有网络**



**图 10-10. 通过私有网络实现运营商路由功能**



**图 10-11. 通过私有网络实现每个租户创建自己专属的网络区段**



## 10.5 Neutron中的各种概念

Neutron 管理的网络资源包括 Network，subnet 和 port，下面依次介绍。

#### 10.5.1 network

network 是一个隔离的二层广播域。Neutron 支持多种类型的 network，包括 local, flat, VLAN, VxLAN 和 GRE。

Local：local 网络与其他网络和节点隔离。local 网络中的 instance 只能与位于同一节点上同一网络的 instance 通信，local 网络主要用于单机测试。

Flat：flat 网络是无 vlan tagging 的网络。flat 网络中的 instance 能与位于同一网络的 instance 通信，并且可以跨多个节点。

Vlan：vlan 网络是具有 802.1q tagging 的网络。vlan 是一个二层的广播域，同一 vlan 中的 instance 可以通信，不同 vlan 只能通过 router 通信。vlan 网络可以跨节点，是应用最广泛的网络类型。

Vxlan：vxlan 是基于隧道技术的 overlay 网络。vxlan 网络通过唯一的 segmentation ID（也叫 VNI）与其他 vxlan 网络区分。vxlan 中数据包会通过 VNI 封装成 UPD 包进行传输。因为二层的包通过封装在三层传输，能够克服 vlan 和物理网络基础设施的限制。

gre  
gre 是与 vxlan 类似的一种 overlay 网络。主要区别在于使用 IP 包而非 UDP 进行封装。

不同 network 之间在二层上是隔离的。  
以 vlan 网络为例，network A 和 network B 会分配不同的 VLAN ID，这样就保证了 network A 中的广播包不会跑到 network B 中。当然，这里的隔离是指二层上的隔离，借助路由器不同 network 是可能在三层上通信的。

network 必须属于某个 Project（ Tenant 租户），Project 中可以创建多个 network。network 与 Project 之间是 1对多关系。

#### 10.5.2 subnet

subnet 是一个 IPv4 或者 IPv6 地址段。instance 的 IP 从 subnet 中分配。每个 subnet 需要定义 IP 地址的范围和掩码。

subnet 与 network 是 多对1关系。一个 subnet 只能属于某个 network；一个 network 可以有多个 subnet，这些 subnet 可以是不同的 IP 段，但不能重叠。下面的配置是有效的：

network A

 subnet A-a: 10.10.1.0/24  {"start": "10.10.1.1", "end": "10.10.1.50"}

 subnet A-b: 10.10.2.0/24  {"start": "10.10.2.1", "end": "10.10.2.50"}

但下面的配置则无效，因为 subnet 有重叠

networkA

 subnet A-a: 10.10.1.0/24  {"start": "10.10.1.1", "end": "10.10.1.50"}

 subnet A-b: 10.10.1.0/24  {"start": "10.10.1.51", "end": "10.10.1.100"}

这里不是判断 IP 是否有重叠，而是 subnet 的 CIDR 重叠（都是 10.10.1.0/24）

但是，如果 subnet 在不同的 network 中，CIDR 和 IP 都是可以重叠的，比如

network A       subnet A-a: 10.10.1.0/24  {"start": "10.10.1.1", "end": "10.10.1.50"}

networkB        subnet B-a: 10.10.1.0/24  {"start": "10.10.1.1", "end": "10.10.1.50"}

这里大家不免会疑惑： 如果上面的IP地址是可以重叠的，那么就可能存在具有相同 IP 的两个 instance，这样会不会冲突？ 简单的回答是：不会！

具体原因： 因为 Neutron 的 router 是通过 Linux network namespace 实现的。network namespace 是一种网络的隔离机制。通过它，每个 router 有自己独立的路由表。

上面的配置有两种结果：

如果两个 subnet 是通过同一个 router 路由，根据 router 的配置，只有指定的一个 subnet 可被路由。

如果上面的两个 subnet 是通过不同 router 路由，因为 router 的路由表是独立的，所以两个 subnet 都可以被路由。

这里只是先简单做个说明，我们会在后面三层路由的章节详细分析这种场景。

#### 10.5.3 port

port 可以看做虚拟交换机上的一个端口。port 上定义了 MAC 地址和 IP 地址，当 instance 的虚拟网卡 VIF（Virtual Interface） 绑定到 port 时，port 会将 MAC 和 IP 分配给 VIF。

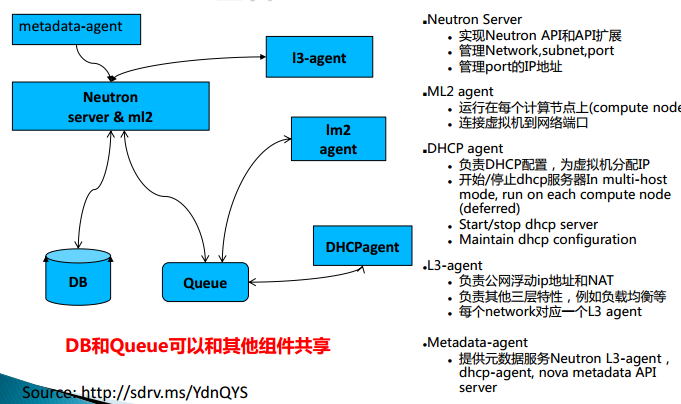
port 与 subnet 是 1对多关系。一个 port必须属于某个subnet；一个subnet 可以有多个 port。

#### 10.5.4 小节

下面总结了 Project，Network，Subnet，Port 和 VIF 之间关系。

Project1 : mNetwork1 : mSubnet 1 : m Port 1 : 1 VIF m : 1 Instance

## 10.6 Neutron中的Plugin和agent



## 10.7 实战：neutron的手动搭建

### 10.7.1 linuxbridge+vxlan模式下的手动部署

#### 10.7.1.1 控制节点

mysql -uroot -popenstack <<EOF

create database neutron;

GRANT ALL PRIVILEGES ON neutron.\* TO 'neutron'@'localhost' IDENTIFIED BY 'openstack'; GRANT ALL PRIVILEGES ON neutron.\* TO 'neutron'@'%' IDENTIFIED BY 'openstack';

EOF

创建neutron用户，并在service项目中添加管理员角色

source admin\_openrc

下面这句要分开执行，要输入neutron用户的密码：

openstack user create --domain default --password-prompt neutron

openstack role add --project service --user neutron admin

创建网络服务及端口

openstack service create --name neutron --description "OpenStack Networking" network

openstack endpoint create --region RegionOne network public [http://controller1:9696](http://controller:9696)

openstack endpoint create --region RegionOne network admin [http://controller1:9696](http://controller:9696)

openstack endpoint create --region RegionOne network internal [http://controller1:9696](http://controller:9696)

安装相关包并进行配置

yum install openstack-neutron openstack-neutron-ml2 openstack-neutron-linuxbridge ebtables

编辑配置文件：

vim /etc/neutron/neutron.conf

[database]

connection = mysql+pymysql://neutron:openstack@controller1/neutron

[DEFAULT]

core\_plugin = ml2

service\_plugins = router

allow\_overlapping\_ips = true

[DEFAULT]

transport\_url = rabbit://openstack:openstack@controller1

[DEFAULT]

auth\_strategy = keystone

[keystone\_authtoken]

auth\_uri = http://controller1:5000

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = cpontroller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = neutron

password = openstack

[DEFAULT]

notify\_nova\_on\_port\_status\_changes = true

notify\_nova\_on\_port\_data\_changes = true

[nova]

auth\_url = [http://controller1:35357](http://controller:35357)

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

region\_name = RegionOne

project\_name = service

username = nova

password = openstack

[oslo\_concurrency]

lock\_path = /var/lib/neutron/tmp

vim /etc/neutron/plugins/ml2/ml2\_conf.ini

[ml2]

type\_drivers = flat,vlan,vxlan

[ml2]

tenant\_network\_types = vxlan

[ml2]

mechanism\_drivers = linuxbridge,l2population

[ml2]

extension\_drivers = port\_security

[ml2\_type\_flat]

flat\_networks = provider

[ml2\_type\_vxlan]

vni\_ranges = 1:1000

[securitygroup]

enable\_ipset = true

vim /etc/neutron/plugins/ml2/linuxbridge\_agent.ini

[linux\_bridge]

physical\_interface\_mappings = provider:eth2

[vxlan]

enable\_vxlan = true

local\_ip = 10.2.2.11

l2\_population = true

[securitygroup]

enable\_security\_group = true

firewall\_driver = neutron.agent.linux.iptables\_firewall.IptablesFirewallDriver

确认操作系统内核支持桥接：

echo "net.bridge.bridge-nf-call-iptables = 1" >> /etc/sysctl.conf

echo "net.bridge.bridge-nf-call-ip6tables = 1" >> /etc/sysctl.conf

sysctl -p

如果“sysctl -p”加载不成功，报” No such file or directory”错误，需要加载内核模块“br\_netfilter”；

命令“modinfo br\_netfilter”查看内核模块信息；

命令“modprobe br\_netfilter”加载内核模块

再次执行成功

vim /etc/neutron/l3\_agent.ini

[DEFAULT]

interface\_driver = linuxbridge

vim /etc/neutron/dhcp\_agent.ini

[DEFAULT]

interface\_driver = linuxbridge

dhcp\_driver = neutron.agent.linux.dhcp.Dnsmasq

enable\_isolated\_metadata = true

vim /etc/neutron/metadata\_agent.ini

[DEFAULT]

nova\_metadata\_host = 192.168.237.131

metadata\_proxy\_shared\_secret = openstack

vim /etc/nova/nova.conf

[neutron]

url = [http://controller1:9696](http://controller:9696)

auth\_url = [http://controller1:35357](http://controller:35357)

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

region\_name = RegionOne

project\_name = service

username = neutron

password = openstack

service\_metadata\_proxy = true

metadata\_proxy\_shared\_secret = openstack

ln -s /etc/neutron/plugins/ml2/ml2\_conf.ini /etc/neutron/plugin.ini

初始化neutron数据库及表结构：

su -s /bin/sh -c "neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/neutron/plugins/ml2/ml2\_conf.ini upgrade head" neutron

重启nova服务

systemctl restart openstack-nova-api.service

服务启动并设置开机自启

systemctl enable neutron-server.service neutron-linuxbridge-agent.service neutron-dhcp-agent.service neutron-metadata-agent.service && systemctl start neutron-server.service neutron-linuxbridge-agent.service neutron-dhcp-agent.service neutron-metadata-agent.service

systemctl enable neutron-l3-agent.service && systemctl start neutron-l3-agent.service

#### 计算节点

yum install openstack-neutron-linuxbridge ebtables ipset

vim /etc/neutron/neutron.conf

[DEFAULT]

transport\_url = rabbit://openstack:[openstack@controller](mailto:f4mtdycd@192.168.10.11)1

[DEFAULT]

auth\_strategy = keystone

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = neutron

password = openstack

[oslo\_concurrency]

lock\_path = /var/lib/neutron/tmp

vim /etc/neutron/plugins/ml2/linuxbridge\_agent.ini

[linux\_bridge]

physical\_interface\_mappings = provider: ens34

[vxlan]

enable\_vxlan = true

local\_ip = 172.17.2.141

l2\_population = true

[securitygroup]

enable\_security\_group = true

firewall\_driver = neutron.agent.linux.iptables\_firewall.IptablesFirewallDriver

vim /etc/nova/nova.conf

[neutron]

url = [http://controller1:9696](http://controller:9696)

auth\_url = [http://controller1:35357](http://controller:35357)

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

region\_name = RegionOne

project\_name = service

username = neutron

password = openstack

重启nova-compute服务

systemctl restart openstack-nova-compute.service

启动服务并设置开机自启

systemctl enable neutron-linuxbridge-agent.service && systemctl start neutron-linuxbridge-agent.service

#### 10.7.1.3 验证

source admin-openrc

openstack extension list --network

openstack network agent list

### 10.7.2 Ovs+vxlan模式的手动部署

#### 10.7.2.1 控制节点

#### 10.7.2.2 计算节点

# 11.实战：Horizon的手动搭建

yum install openstack-dashboard

vim /etc/openstack-dashboard/local\_settings

OPENSTACK\_HOST = "controller1"

ALLOWED\_HOSTS = ['\*']

SESSION\_ENGINE = 'django.contrib.sessions.backends.cache'

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',

'LOCATION': 'controller1:11211',

}

}

注释掉其他cache

OPENSTACK\_KEYSTONE\_URL = "http://%s:5000/v3" % OPENSTACK\_HOST

OPENSTACK\_KEYSTONE\_MULTIDOMAIN\_SUPPORT = True

OPENSTACK\_API\_VERSIONS = {

"identity": 3,

"image": 2,

"volume": 2,

}

OPENSTACK\_KEYSTONE\_DEFAULT\_DOMAIN = "Default"

OPENSTACK\_KEYSTONE\_DEFAULT\_ROLE = "user"

OPENSTACK\_NEUTRON\_NETWORK = {

...

'enable\_router': True,

'enable\_quotas': True,

'enable\_distributed\_router': True,

'enable\_ha\_router': True,

'enable\_lb': True,

'enable\_firewall': True,

'enable\_vpn': False,

'enable\_fip\_topology\_check': True,

}

TIME\_ZONE = "Asia/Chongqing"

vim /etc/httpd/conf.d/openstack-dashboard.conf

WSGIApplicationGroup %{GLOBAL}

systemctl restart httpd.service memcached.service

# 12. 创建一个实例

## 12.1 创建虚拟网络

### 12.1.1 创建provider网络

. admin-openrc

openstack network create --share --external \

--provider-physical-network provider \

--provider-network-type flat provider

openstack subnet create --network provider \

--allocation-pool start=192.168.1.151,end=192.168.1.200 \

--dns-nameserver 8.8.4.4 --gateway 192.168.1.1 \

--subnet-range 192.168.1.1/24 provider

### 12.1.2 创建私有网络 self-services

. demo-openrc

openstack network create selfservice

openstack subnet create --network selfservice \

--dns-nameserver 8.8.4.4 --gateway 172.16.1.1 \

--subnet-range 172.16.1.0/24 selfservice

openstack router create router

openstack router add subnet router selfservice

openstack router set router --external-gateway provider

### 12.1.3 验证

. admin-openrc

ip netns

openstack port list --router router

ping -c 192.168.1.158

## 12.2 创建flavor

openstack flavor create --id 0 --vcpus 1 --ram 64 --disk 1 m1.nano

## 生成密钥对

. demo-openrc

ssh-keygen -q -N ""

openstack keypair create --public-key ~/.ssh/id\_rsa.pub mykey

openstack keypair list

## 添加安全组规则

openstack security group rule create --proto icmp default

openstack security group rule create --proto tcp --dst-port 22 default

## 12.5启动一个实例

. demo-openrc

openstack flavor list

openstack image list

openstack network list

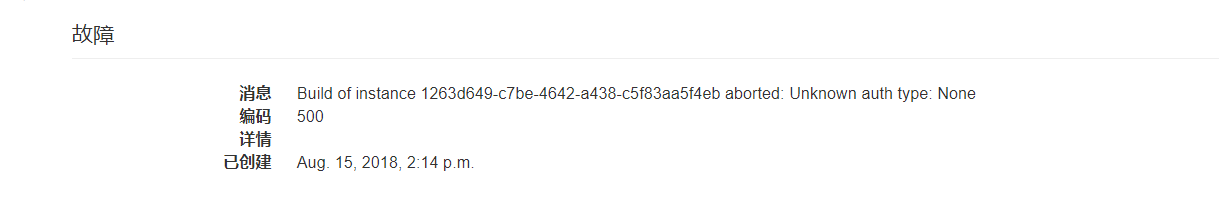
openstack security group list

openstack server create --flavor m1.nano --image cirros \

--nic net-id=SELFSERVICE\_NET\_ID --security-group default \

--key-name mykey selfservice-instance

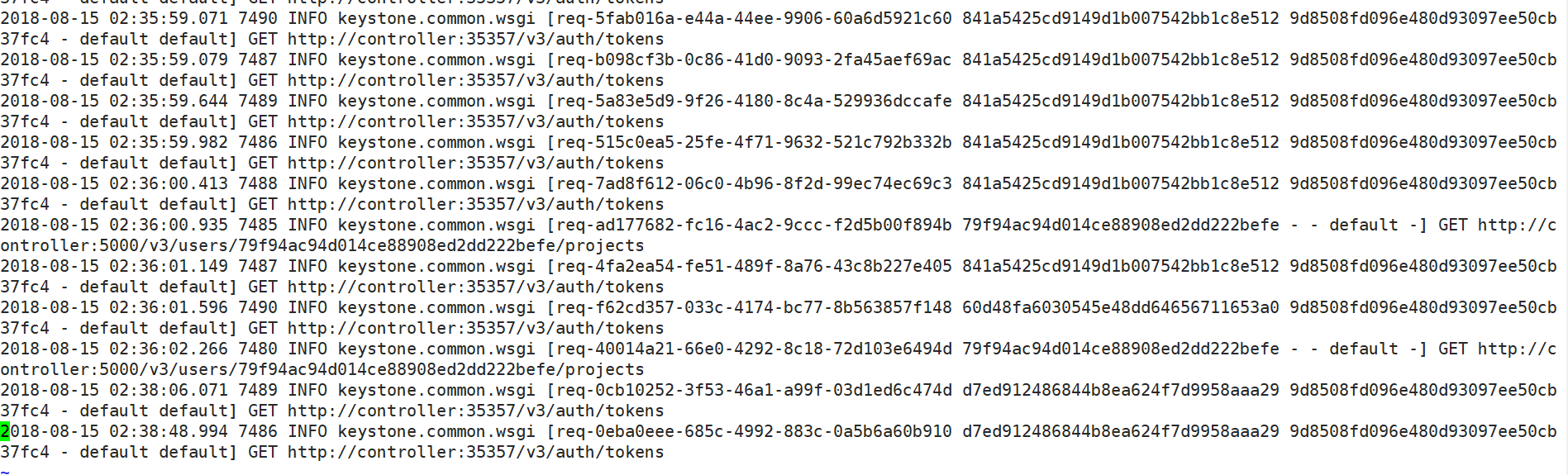
通过界面查看，报如下错误：



错误给出的提示是：没有认证类型，

首先检查了各个服务的状态，全部正常。

依次查看了 vim /var/log/keystone/keystone.log日志如下：

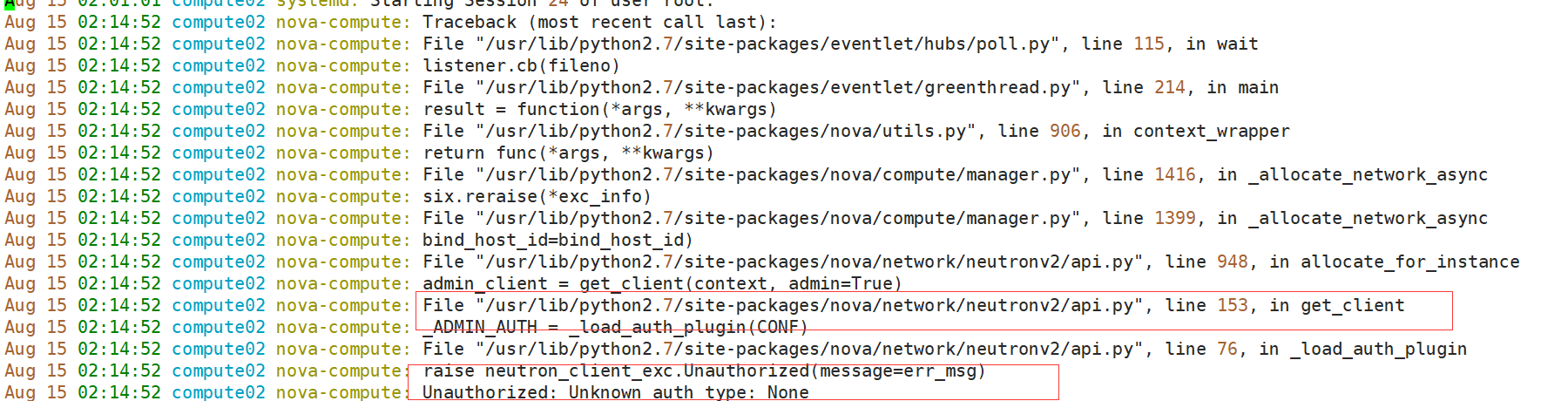


日志在打印过程中，没有任何错误，之后指出了token的获取过程

然后查看了/var/log/nova/下面的日志，nova-api,nova-scheduler,nova-conductor的日志，都没有报错，不再一一展示。

最后查看了两个计算节点 的compute日志，/var/log/nova/nova-compute.log 卧槽来，依然没错，怎么办？

抱着试试看的态度，挨个查看了controller、compute节点的/var/log/messages中的日志，找到了错误信息，在compute2节点上报了如下错误：



最后断定在nova在通知neutron创建网络的时候出现问题，然后参看了compute2中的/etc/nova/nova.conf 配置文件，如下：

[root@compute02 log]# egrep -v "^$|^#" /etc/nova/nova.conf

[DEFAULT]

my\_ip = 10.1.1.22

use\_neutron=True

firewall\_driver = nova.virt.firewall.NoopFirewallDriver

enabled\_apis=osapi\_compute,metadata

transport\_url = rabbit://openstack:openstack@controller

[api]

auth\_strategy=keystone

[api\_database]

[barbican]

[cache]

[cells]

[cinder]

[compute]

[conductor]

[console]

[consoleauth]

[cors]

[crypto]

[database]

[devices]

[ephemeral\_storage\_encryption]

[filter\_scheduler]

[glance]

api\_servers = http://controller:9292

[guestfs]

[healthcheck]

[hyperv]

[ironic]

[key\_manager]

[keystone]

[keystone\_authtoken]

auth\_uri = http://controller:5000

auth\_url = http://controller:35357

memcached\_servers = controller:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = nova

password = openstack

[libvirt]

virt\_type=qemu

[matchmaker\_redis]

[metrics]

[mks]

[neutron]

[notifications]

[osapi\_v21]

[oslo\_concurrency]

lock\_path = /var/lib/nova/tmp

[oslo\_messaging\_amqp]

[oslo\_messaging\_kafka]

[oslo\_messaging\_notifications]

[oslo\_messaging\_rabbit]

[oslo\_messaging\_zmq]

[oslo\_middleware]

[oslo\_policy]

[pci]

[placement]

os\_region\_name = RegionOne

project\_domain\_name = Default

project\_name = service

auth\_type = password

user\_domain\_name = Default

auth\_url = http://controller:35357/v3

username = placement

password = openstack

[quota]

[rdp]

[remote\_debug]

[scheduler]

[serial\_console]

[service\_user]

[spice]

[upgrade\_levels]

[vault]

[vendordata\_dynamic\_auth]

[vmware]

[vnc]

enabled = True

server\_listen = 0.0.0.0

server\_proxyclient\_address = $my\_ip

novncproxy\_base\_url = http://10.1.1.11:6080/vnc\_auto.html

[workarounds]

[wsgi]

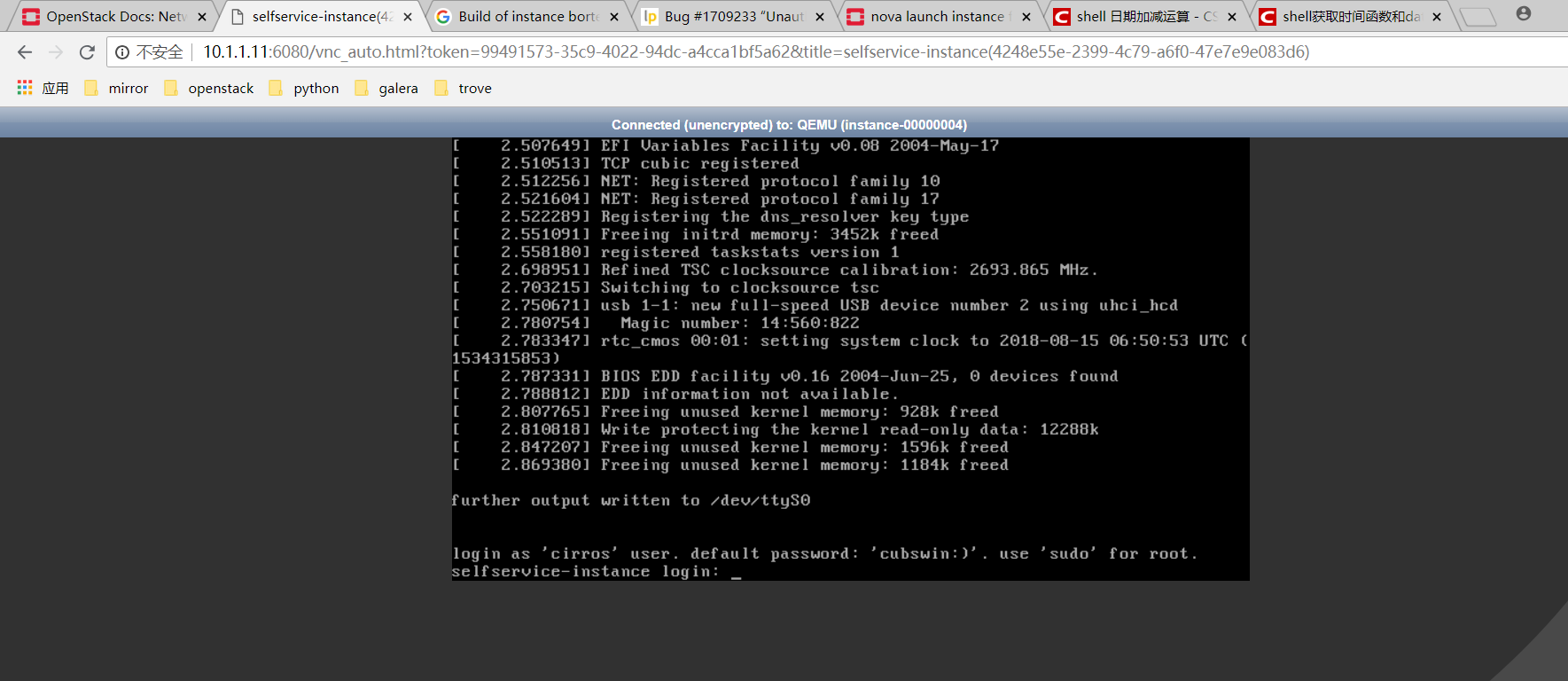
[xenserver]

[xvp]

关于neutron的配置为空，加入neutron的调用信息，再次创建实例成功。

验证：ping 172.168.1.1可以ping 通。

查看控制台，实例创建成功。



# 13. Cinder组件

## 13.1 cinder的定义

操作系统获得存储空间的方式一般有两种：

1. 通过某种协议（SAS,SCSI,SAN,iSCSI 等）挂接裸硬盘，然后分区、格式化、创建文件系统；或者直接使用裸硬盘存储数据（数据库）
2. 通过 NFS、CIFS 等 协议，mount 远程的文件系统

第一种裸硬盘的方式叫做 Block Storage（块存储），每个裸硬盘通常也称作 Volume（卷） 第二种叫做文件系统存储。NAS 和 NFS 服务器，以及各种分布式文件系统提供的都是这种存储。

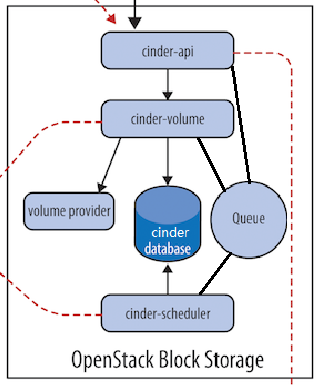
Block Storage Service提供对 volume 从创建到删除整个生命周期的管理。

从 instance 的角度看，挂载的每一个 Volume 都是一块硬盘。

OpenStack 提供 Block Storage Service 的是 Cinder，其具体功能是：

1. 提供 REST API 使用户能够查询和管理 volume、volume snapshot 以及 volume type
2. 提供 scheduler 调度 volume 创建请求，合理优化存储资源的分配
3. 通过 driver 架构支持多种 back-end（后端）存储方式，包括 LVM，NFS，Ceph 和其他诸如 EMC、IBM 等商业存储产品和方案

## cinder的架构



Cinder 包含如下几个组件：

**cinder-api**  
接收 API 请求，调用 cinder-volume 执行操作。

**cinder-volume**  
管理 volume 的服务，与 volume provider 协调工作，管理 volume 的生命周期。运行 cinder-volume 服务的节点被称作为存储节点。

**cinder-scheduler**  
scheduler 通过调度算法选择最合适的存储节点创建 volume。

**volume provider**  
数据的存储设备，为 volume 提供物理存储空间。 cinder-volume 支持多种 volume provider，每种 volume provider 通过自己的 driver 与cinder-volume 协调工作。

**Message Queue**  
Cinder 各个子服务通过消息队列实现进程间通信和相互协作。因为有了消息队列，子服务之间实现了解耦，这种松散的结构也是分布式系统的重要特征。

**Database** Cinder 有一些数据需要存放到数据库中，一般使用 MySQL。数据库是安装在控制节点上的，比如在我们的实验环境中，可以访问名称为“cinder”的数据库。

## 实战：cinder的手动搭建

### 控制节点

数据库相关操作：

mysql -uroot -popenstack <<EOF

create database cinder;

GRANT ALL PRIVILEGES ON cinder.\* TO 'cinder'@'localhost' IDENTIFIED BY 'openstack';

GRANT ALL PRIVILEGES ON cinder.\* TO 'cinder'@'%' IDENTIFIED BY 'openstack';

EOF

创建cinder用户，并在service项目中添加管理员角色

. admin\_openrc

下面这句要分开执行，要输入cinder用户的密码：

openstack user create --domain default --password-prompt cinder

openstack role add --project service --user cinder admin

创建cinder服务及端口

openstack service create --name cinderv2 --description "OpenStack Block Storage" volumev2

openstack service create --name cinderv3 --description "OpenStack Block Storage" volumev3

openstack endpoint create --region RegionOne volumev2 public http://controller1:8776/v2/%\(project\_id\)s

openstack endpoint create --region RegionOne volumev2 internal http://controller1:8776/v2/%\(project\_id\)s

openstack endpoint create --region RegionOne volumev2 admin http://controller1:8776/v2/%\(project\_id\)s

openstack endpoint create --region RegionOne volumev3 public http://controller1:8776/v3/%\(project\_id\)s

openstack endpoint create --region RegionOne volumev3 internal http://controller1:8776/v3/%\(project\_id\)s

openstack endpoint create --region RegionOne volumev3 admin http://controller1:8776/v3/%\(project\_id\)s

安装包并修改配置

yum install openstack-cinder

vim /etc/cinder/cinder.conf

[database]

connection = mysql+pymysql://cinder:openstack@controller1/cinder

[DEFAULT]

transport\_url = rabbit://openstack:openstack@controller1

[DEFAULT]

auth\_strategy = keystone

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = cinder

password = openstack

[DEFAULT]

my\_ip = 192.168.237.131

[oslo\_concurrency]

lock\_path = /var/lib/cinder/tmp

su -s /bin/sh -c "cinder-manage db sync" cinder

配置计算服务使用cinder

vim /etc/nova/nova.conf

[cinder]

os\_region\_name = RegionOne

计算服务重启

systemctl restart openstack-nova-api.service

服务启动并设置开机自启

systemctl enable openstack-cinder-api.service openstack-cinder-scheduler.service && systemctl start openstack-cinder-api.service openstack-cinder-scheduler.service

### 存储节点

添加一个新磁盘 /dev/sdb

安装相关包，并进行配置

yum install lvm2 device-mapper-persistent-data

systemctl enable lvm2-lvmetad.service && systemctl start lvm2-lvmetad.service

pvcreate /dev/sdb

vgcreate cinder-volumes /dev/sdb

vim /etc/lvm/lvm.conf

devices { ... filter = [ "a/sdb/", "r/.\*/"]

若操作系统安装时候了lvm，必须也要accept，配置如下，

devices { ... filter = ["a/sda/", "a/sdb/", "r/.\*/"]

yum install openstack-cinder targetcli python-keystone

vim /etc/cinder/cinder.conf

[database]

connection = mysql+pymysql://cinder:openstack@controller1/cinder

[DEFAULT]

transport\_url = rabbit://openstack: openstack@controller1

[DEFAULT]

auth\_strategy = keystone

[keystone\_authtoken]

auth\_uri = [http://controller1:5000](http://controller:5000)

auth\_url = [http://controller1:35357](http://controller:35357)

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_name = default

user\_domain\_name = default

project\_name = service

username = cinder

password = openstack

[DEFAULT]

my\_ip = 192.168.237.151

[lvm]

volume\_driver = cinder.volume.drivers.lvm.LVMVolumeDriver

volume\_group = cinder-volumes

iscsi\_protocol = iscsi

iscsi\_helper = lioadm

[DEFAULT]

enabled\_backends = lvm

[DEFAULT]

glance\_api\_servers = [http://controller1:9292](http://controller:9292)

[oslo\_concurrency]

lock\_path = /var/lib/cinder/tmp

服务启动并加入到开机自启

systemctl enable openstack-cinder-volume.service target.service && systemctl start openstack-cinder-volume.service target.service

### 13.3 验证

source .admin-openrc

openstack volume service list

## 13.4实战：给虚拟机分配一个虚拟磁盘

. demo-openrc

openstack volume create --size 1 volume1

openstack volume list

openstack server add volume provider-instance volume1

openstack volume list

# 14. Swift组件

## 14.1 swift概述

OpenStack Object Storage(Swift)是OpenStack开源云计算项目的子项目之一，被称为对象存储，提供了强大的扩展性、冗余和持久性。对象存储，用于永久类型的静态数据的长期存储。  
 Swift 最初是由 Rackspace 公司开发的高可用分布式对象存储服务，并于 2010 年贡献给 OpenStack 开源社区作为其最初的核心子项目之一，为其 Nova 子项目提供虚机镜像存储服务。Swift 构筑在比较便宜的标准硬件存储基础设施之上，无需采用 RAID（磁盘冗余阵列），通过在软件层面引入一致性散列技术和数据冗余性，牺牲一定程度的数据一致性来达到高可用性和可伸缩性，支持多租户模式、容器和对象读写操作，适合解决互联网的应用场景下**非结构化数据存储**问题。

## 14.2 swift的基本原理一：一致性散列

一致性Hash是由麻省理工学院在1997年提出的一种DHT算法，目的是为了解决分布式网络中的热点问题。其基本思想是将物理节点和key按照相同的hash算法映射到的虚拟的散列空间环上。当接收到一个读/写请求时，计算key值K的哈希值Hash(k),如果该值与某个物理节点的hash值相同，就直接对该物理节点进行读/写操作；如果不同，就按照顺时针方向查找到第一个物理节点，对该节点进行读写操作；如果超过了从0开始从新查找。



一致性Hash的过程如下几个步骤组成：

1. 计算每个对象的hash值，并且把它们均匀的分布到一个圆形散列的地址空间上；
2. 假设有个存储节点，将存储空间均匀分成等份，每一份的虚拟空间大小为；
3. 假设一个对象的hash值为n,那么其对应的存储节点为。相当于将hash值向右位移位。

在Swift对象存储系统中，为了减少节点的加入或者是删除导致的大规模的数据迁移，引入了虚拟节点的概念。以前的存储方式由的一层映射关系演变成的两层映射关系。



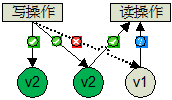
在整个Swift对象存储系统的生命周期中，虚拟节点的总数是不变的，在构建Swift对象存储系统的时，考虑整个物理存储集群的最大值，虚拟节点总数是物理节点总数的100倍。

## 14.3 swift的基本原理二：数据一致性模型

按照 Eric Brewer 的 CAP（Consistency，Availability，Partition Tolerance）理论，无法同时满足 3 个方面，Swift 放弃严格一致性（满足 ACID 事务级别），而采用最终一致性模型（Eventual Consistency），来达到高可用性和无限水平扩展能力。为了实现这一目标，Swift 采用 Quorum 仲裁协议(Quorum 有法定投票人数的含义)：

（1）定义：N：数据的副本总数；W：写操作被确认接受的副本数量；R：读操作的副本数量  
 （2）强一致性：R+W>N，以保证对副本的读写操作会产生交集，从而保证可以读取到最新版本；如果 W=N，R=1，则需要全部更新，适合大量读少量写操作场景下的强一致性；如果 R=N，W=1，则只更新一个副本，通过读取全部副本来得到最新版本，适合大量写少量读场景下的强一致性。  
 （3）弱一致性：R+W<=N，如果读写操作的副本集合不产生交集，就可能会读到脏数据；适合对一致性要求比较低的场景。

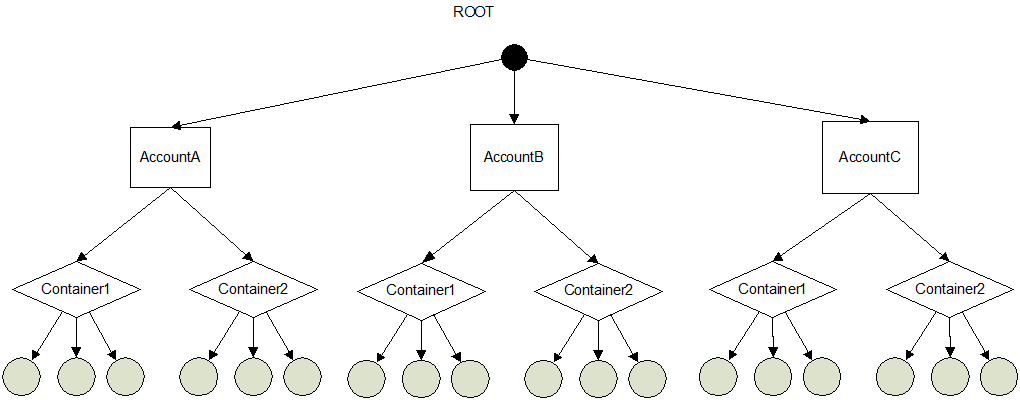
Swift 针对的是读写都比较频繁的场景，所以采用了比较折中的策略，即写操作需要满足至少一半以上成功 W >N/2，再保证读操作与写操作的副本集合至少产生一个交集，即 R+W>N。Swift 默认配置是 N=3，W=2>N/2，R=1 或 2，即每个对象会存在 3 个副本，这些副本会尽量被存储在不同区域的节点上；W=2 表示至少需要更新 2 个副本才算写成功；当 R=1 时意味着某一个读操作成功便立刻返回，此种情况下可能会读取到旧版本（弱一致性模型）；当 R=2 时，需要通过在读操作请求头中增加 x-newest=true 参数来同时读取 2 个副本的元数据信息，然后比较时间戳来确定哪个是最新版本（强一致性模型）；如果数据出现了不一致，后台服务进程会在一定时间窗口内通过检测和复制协议来完成数据同步，从而保证达到最终一致性。



## 14.4 swift的基本原理三：环的数据结构

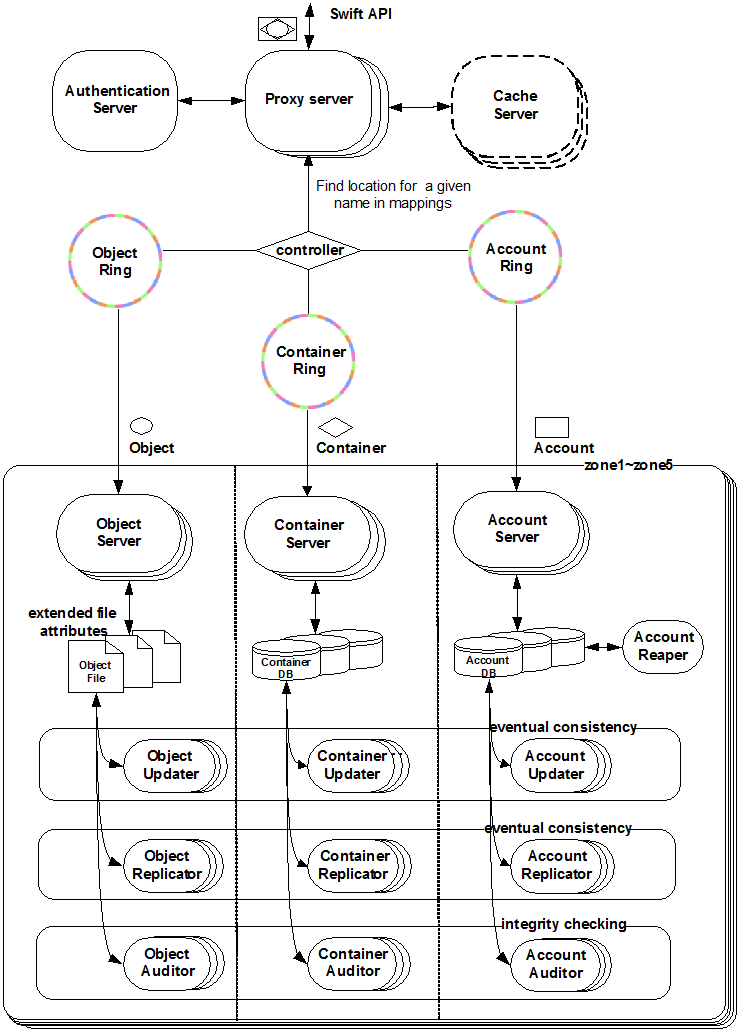
环是为了将虚拟节点（分区）映射到一组物理存储设备上，并提供一定的冗余度而设计的，其数据结构由以下信息组成：存储设备列表、设备信息包括唯一标识号（id）、区域号（zone）、权重（weight）、IP 地址（ip）、端口（port）、设备名称（device）、元数据（meta）。分区到设备映射关系（replica2part2dev\_id 数组)。计算分区号的位移(part\_shift 整数，以查找一个对象的计算过程为例：  
  
 使用对象的层次结构 account/container/object 作为键，使用 MD5 散列算法得到一个散列值，对该散列值的前 4 个字节进行右移操作得到分区索引号，移动位数由上面的 part\_shift 设置指定；按照分区索引号在分区到设备映射表（replica2part2dev\_id）里查找该对象所在分区的对应的所有设备编号，这些设备会被尽量选择部署在不同区域（Zone）内，区域只是个抽象概念，它可以是某台机器，某个机架，甚至某个建筑内的机群，以提供最高级别的冗余性，建议至少部署 5 个区域；权重参数是个相对值，可以来根据磁盘的大小来调节，权重越大表示可分配的空间越多，可部署更多的分区。Swift为账户，容器和对象分别定义了的环，查找账户和容器的是同样的过程。

## swift的基本原理四：数据模型

Swift 采用层次数据模型，共设三层逻辑结构：Account/Container/Object（即账户/容器/对象)，每层节点数均没有限制，可以任意扩展。这里的账户和个人账户不是一个概念，可理解为租户，用来做顶层的隔离机制，可以被多个个人账户所共同使用；容器代表封装一组对象，类似文件夹或目录；叶子节点代表对象，由元数据和内容两部分组成.  


## swift的基本原理五：系统架构

Swift 采用完全对称、面向资源的分布式系统架构设计，所有组件都可扩展，避免因单点失效而扩散并影响整个系统运转；通信方式采用非阻塞式 I/O 模式，提高了系统吞吐和响应能力。



代理服务（Proxy Server）：对外提供对象服务 API，会根据环的信息来查找服务地址并转发用户请求至相应的账户、容器或者对象服务；由于采用无状态的 REST 请求协议，可以进行横向扩展来均衡负载。  
 认证服务（Authentication Server）：验证访问用户的身份信息，并获得一个对象访问令牌（Token），在一定的时间内会一直有效；验证访问令牌的有效性并缓存下来直至过期时间。  
 缓存服务（Cache Server）：缓存的内容包括对象服务令牌，账户和容器的存在信息，但不会缓存对象本身的数据；缓存服务可采用 Memcached 集群，Swift 会使用一致性散列算法来分配缓存地址。   
 账户服务（Account Server）：提供账户元数据和统计信息，并维护所含容器列表的服务，每个账户的信息被存储在一个 SQLite 数据库中。   
 容器服务（Container Server）：提供容器元数据和统计信息，并维护所含对象列表的服务，每个容器的信息也存储在一个 SQLite 数据库中。  
 对象服务（Object Server）：提供对象元数据和内容服务，每个对象的内容会以文件的形式存储在文件系统中，元数据会作为文件属性来存储，建议采用支持扩展属性的 XFS 文件系统。  
 复制服务（Replicator）：会检测本地分区副本和远程副本是否一致，具体是通过对比散列文件和高级水印来完成，发现不一致时会采用推式（Push）更新远程副本，例如对象复制服务会使用远程文件拷贝工具 rsync 来同步；另外一个任务是确保被标记删除的对象从文件系统中移除。  
 更新服务（Updater）：当对象由于高负载的原因而无法立即更新时，任务将会被序列化到在本地文件系统中进行排队，以便服务恢复后进行异步更新；例如成功创建对象后容器服务器没有及时更新对象列表，这个时候容器的更新操作就会进入排队中，更新服务会在系统恢复正常后扫描队列并进行相应的更新处理。  
审计服务（Auditor）：检查对象，容器和账户的完整性，如果发现比特级的错误，文件将被隔离，并复制其他的副本以覆盖本地损坏的副本；其他类型的错误会被记录到日志中。  
账户清理服务（Account Reaper）：移除被标记为删除的账户，删除其所包含的所有容器和对象。

Ring是Swift最重要的组件，用于记录存储对象与物理位置间的映射关系。在涉及查询Account、Container、Object信息时，就需要查询集群的Ring信息。 Ring使用Zone、Device、Partition和Replica来维护这些映射信息。Ring中每个Partition在集群中都（默认）有3个Replica。每个Partition的位置由Ring来维护，并存储在映射中。Ring文件在系统初始化时创建，之后每次增减存储节点时，需要重新平衡一下Ring文件中的项目，以保证增减节点时，系统因此而发生迁移的文件数量最少。

## swift的特性

1. 极高的数据持久性

数据持久性和系统可用性不同，指的是数据的可靠性，数据存储到系统后，到某一天丢失的可能性。AS3的数据持久性是11个9，即如果存储1万个（4个0）文件到S3中，1千万（7个0）年之后，可能会丢失1个文件。  
我们从理论上测算过，Swift在5个Zone、5×10个存储节点的环境下，数据复制份是为3，数据持久性的SLA能达到10个9。

1. 完全对称的系统架构

“对称”意味着Swift中各节点可以完全对等，能极大地降低系统维护成本。

1. 无限的可扩展性

（1）数据存储容量无限可扩展；（2）Swift性能(如QPS、吞吐量等)可线性提升。  
Swift是完全对称的架构，扩容只需简单地新增机器，系统会自动完成数据迁移等工作，使各存储节点重新达到平衡状态。

1. 无单点故障

元数据问题，Swift的元数据存储是完全均匀随机分布的，并且与对象文件存储一样，元数据也会存储多份。

1. 简单、可依赖

设计简单

## 实战：swift的手动搭建

### 控制节点

openstack user create --domain default --password-prompt swift

openstack role add --project service --user swift admin

openstack service create --name swift --description "OpenStack Object Storage" object-store

openstack endpoint create --region RegionOne object-store public [http://controller1:8080/v1/AUTH\_%\(project\_id\)s](http://controller:8080/v1/AUTH_%25\\(project_id\\)s)

openstack endpoint create --region RegionOne object-store internal [http://controller1:8080/v1/AUTH\_%\(project\_id\)s](http://controller:8080/v1/AUTH_%25\\(project_id\\)s)

openstack endpoint create --region RegionOne object-store admin [http://controller1:8080/v1](http://controller:8080/v1)

yum install openstack-swift-proxy python-swiftclient python-keystoneclient python-keystonemiddleware memcached

下载swift-proxy.conf的配置文件，并修改配置：

curl -o /etc/swift/proxy-server.conf <https://git.openstack.org/cgit/openstack/swift/plain/etc/proxy-server.conf-sample?h=stable/quee>ns

vim /etc/swift/proxy-server.conf

[DEFAULT]

bind\_port = 8080

swift\_dir = /etc/swift

user = swift

[pipeline:main]

pipeline = catch\_errors gatekeeper healthcheck proxy-logging cache container\_sync bulk ratelimit authtoken keystoneauth container-quotas account-quotas slo dlo versioned\_writes proxy-logging proxy-server

[app:proxy-server]

use = egg:swift#proxy

account\_autocreate = True

[filter:tempauth]

use = egg:swift#tempauth

user\_admin\_admin = admin .admin .reseller\_admin

user\_test\_tester = testing .admin

user\_test2\_tester2 = testing2 .admin

user\_test\_tester3 = testing3

user\_test5\_tester5 = testing5 service

[filter:authtoken]

paste.filter\_factory = keystonemiddleware.auth\_token:filter\_factory

www\_authenticate\_uri = http://controller1:5000

auth\_url = http://controller1:35357

memcached\_servers = controller1:11211

auth\_type = password

project\_domain\_id = default

user\_domain\_id = default

project\_name = service

username = swift

password = openstack

delay\_auth\_decision = True

[filter:keystoneauth]

use = egg:swift#keystoneauth

operator\_roles = admin,user

[filter:healthcheck]

use = egg:swift#healthcheck

[filter:cache]

use = egg:swift#memcache

memcache\_servers = controller1:11211

[filter:ratelimit]

use = egg:swift#ratelimit

[filter:domain\_remap]

use = egg:swift#domain\_remap

[filter:catch\_errors]

use = egg:swift#catch\_errors

[filter:cname\_lookup]

use = egg:swift#cname\_lookup

[filter:staticweb]

use = egg:swift#staticweb

[filter:tempurl]

use = egg:swift#tempurl

[filter:formpost]

use = egg:swift#formpost

[filter:name\_check]

use = egg:swift#name\_check

[filter:list-endpoints]

use = egg:swift#list\_endpoints

[filter:proxy-logging]

use = egg:swift#proxy\_logging

[filter:bulk]

use = egg:swift#bulk

[filter:slo]

use = egg:swift#slo

[filter:dlo]

use = egg:swift#dlo

[filter:container-quotas]

use = egg:swift#container\_quotas

[filter:account-quotas]

use = egg:swift#account\_quotas

[filter:gatekeeper]

use = egg:swift#gatekeeper

[filter:container\_sync]

use = egg:swift#container\_sync

[filter:xprofile]

use = egg:swift#xprofile

[filter:versioned\_writes]

use = egg:swift#versioned\_writes

[filter:copy]

use = egg:swift#copy

[filter:keymaster]

use = egg:swift#keymaster

encryption\_root\_secret = changeme

[filter:kms\_keymaster]

use = egg:swift#kms\_keymaster

[filter:encryption]

use = egg:swift#encryption

[filter:listing\_formats]

use = egg:swift#listing\_formats

[filter:symlink]

use = egg:swift#symlink

### 14.8.2 存储节点

yum install xfsprogs rsync

mkfs.xfs /dev/sdb

mkfs.xfs /dev/sdc

mkdir -p /srv/node/sdb

mkdir -p /srv/node/sdc

vim /etc/fstab

/dev/sdb /srv/node/sdb xfs noatime,nodiratime,nobarrier,logbufs=8 0 2

/dev/sdc /srv/node/sdc xfs noatime,nodiratime,nobarrier,logbufs=8 0 2

mount /srv/node/sdb

mount /srv/node/sdc

vim /etc/rsyncd.conf

uid = swift

gid = swift

log file = /var/log/rsyncd.log

pid file = /var/run/rsyncd.pid

address = 192.168.237.161

[account]

max connections = 2

path = /srv/node/

read only = False

lock file = /var/lock/account.lock

[container]

max connections = 2

path = /srv/node/

read only = False

lock file = /var/lock/container.lock

[object]

max connections = 2

path = /srv/node/

read only = False

lock file = /var/lock/object.lock

systemctl enable rsyncd.service && systemctl start rsyncd.service

yum install openstack-swift-account openstack-swift-container openstack-swift-object

curl -o /etc/swift/account-server.conf https://git.openstack.org/cgit/openstack/swift/plain/etc/account-server.conf-sample?h=stable/queens

curl -o /etc/swift/container-server.conf https://git.openstack.org/cgit/openstack/swift/plain/etc/container-server.conf-sample?h=stable/queens

curl -o /etc/swift/object-server.conf https://git.openstack.org/cgit/openstack/swift/plain/etc/object-server.conf-sample?h=stable/queens

vim /etc/swift/account-server.conf

[DEFAULT]

...

bind\_ip = 192.168.237.161

bind\_port = 6202

user = swift

swift\_dir = /etc/swift

devices = /srv/node

mount\_check = True

[pipeline:main]

pipeline = healthcheck recon account-server

[filter:recon]

use = egg:swift#recon

...

recon\_cache\_path = /var/cache/swift

vim  /etc/swift/container-server.conf

[DEFAULT]

...

bind\_ip = 192.168.237.161

bind\_port = 6201

user = swift

swift\_dir = /etc/swift

devices = /srv/node

mount\_check = True

[filter:recon]

use = egg:swift#recon

...

recon\_cache\_path = /var/cache/swift

vim /etc/swift/object-server.conf

[DEFAULT]

...

bind\_ip = 192.168.237.161

bind\_port = 6200

user = swift

swift\_dir = /etc/swift

devices = /srv/node

mount\_check = True

[pipeline:main]

pipeline = healthcheck recon object-server

[filter:recon]

use = egg:swift#recon

recon\_cache\_path = /var/cache/swift

recon\_lock\_path = /var/lock

Chow -R swift:swift /srv/node

mkdir -p /var/cache/swift

chown -R root:swift /var/cache/swift

chmod -R 775 /var/cache/swift

### 14.8.3 Create and distribute initial rings

cd /etc/swift

swift-ring-builder account.builder create 10 3 1

swift-ring-builder account.builder add \

--region 1 --zone 1 --ip 192.168.237.161 --port 6202 --device sdb --weight 100

swift-ring-builder account.builder add \

--region 1 --zone 1 --ip 192.168.237.161 --port 6202 --device sdc --weight 100

swift-ring-builder account.builder add \

--region 1 --zone 2 --ip 192.168.237.162 --port 6202 --device sdb --weight 100

swift-ring-builder account.builder add \

--region 1 --zone 2 --ip 192.168.237.162 --port 6202 --device sdc --weight 100

swift-ring-builder account.builder

swift-ring-builder account.builder rebalance

swift-ring-builder container.builder create 10 3 1

swift-ring-builder container.builder add \

--region 1 --zone 1 --ip 192.168.237.161 --port 6201 --device sdb - -weight 100

swift-ring-builder container.builder add \

--region 1 --zone 1 --ip 192.168.237.161 --port 6201 --device sdc --weight 100

swift-ring-builder container.builder add \

--region 1 --zone 2 --ip 192.168.237.162 --port 6201 --device sdb --weight 100

swift-ring-builder container.builder add \

--region 1 --zone 2 --ip 192.168.237.162 --port 6201 --device sdc --weight 100

swift-ring-builder container.builder

swift-ring-builder container.builder rebalance

swift-ring-builder object.builder create 10 3 1

swift-ring-builder object.builder add \

--region 1 --zone 1 --ip 192.168.237.161 --port 6200 --device sdb --weight 100

swift-ring-builder object.builder add \

--region 1 --zone 1 --ip 192.168.237.161 --port 6200 --device sdc --weight 100

swift-ring-builder object.builder add \

--region 1 --zone 2 --ip 192.168.237.162 --port 6200 --device sdb --weight 100

swift-ring-builder object.builder add \

--region 1 --zone 2 --ip 192.168.237.162 --port 6200 --device sdc --weight 100

swift-ring-builder object.builder

swift-ring-builder object.builder rebalance

scp account.ring.gz container.ring.gz object.ring.gz object01:/etc/swift/

scp account.ring.gz container.ring.gz object.ring.gz object02:/etc/swift/

获取swift.conf配置文件

curl -o /etc/swift/swift.conf \

<https://git.openstack.org/cgit/openstack/swift/plain/etc/swift.conf-sample?h=stable/queens>

vim /etc/swift/swift.conf

[swift-hash]

...

swift\_hash\_path\_suffix = HASH\_PATH\_SUFFIX

swift\_hash\_path\_prefix = HASH\_PATH\_PREFIX

[storage-policy:0]

...

name = Policy-0

default = yes

scp /etc/swift/swift.conf object01:/etc/swift

scp /etc/swift/swift.conf object02:/etc/swift

所有节点执行：

chown -R root:swift /etc/swift

控制节点：

systemctl enable openstack-swift-proxy.service memcached.service && systemctl restart openstack-swift-proxy.service memcached.service

存储节点：

systemctl enable openstack-swift-account.service openstack-swift-account-auditor.service openstack-swift-account-reaper.service openstack-swift-account-replicator.service

systemctl restart openstack-swift-account.service openstack-swift-account-auditor.service openstack-swift-account-reaper.service openstack-swift-account-replicator.service

systemctl enable openstack-swift-container.service openstack-swift-container-auditor.service openstack-swift-container-replicator.service openstack-swift-container-updater.service

systemctl restart openstack-swift-container.service openstack-swift-container-auditor.service openstack-swift-container-replicator.service openstack-swift-container-updater.service

systemctl enable openstack-swift-object.service openstack-swift-object-auditor.service openstack-swift-object-replicator.service openstack-swift-object-updater.service

Systemctl restart openstack-swift-object.service openstack-swift-object-auditor.service openstack-swift-object-replicator.service openstack-swift-object-updater.service

### 验证

备注：首先检查 /var/log/audit/audit.log,若存在selinux的信息，使得swift 进程无法访问，做如下修改：

chcon -R system\_u:object\_r:swift\_data\_t:s0 /srv/node

. demo-openrc

swift stat

openstack container create container1

openstack object create contain**er1 FILE**

openstack object list container1

openstack object save container1 FILE

# 附录1：制作openstack的Linux镜像

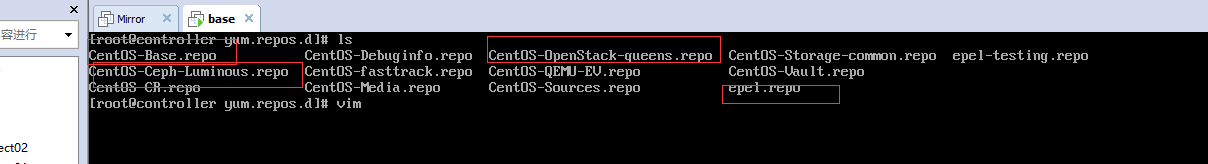
**<http://note.youdao.com/noteshare?id=ddca18336995001dbc249385f5bbdd23&sub=E0DEDD10722C48E880E4D0B49D98F6B5>**

# 附录2：制作OpenStack的window镜像

<http://note.youdao.com/noteshare?id=19bedded6fc3aca9329b8c7dfa5c56a2&sub=B4DFBC3FB285456A97EE3E48090A8BBE>

# 附录3：容器制作openstack本地yum源

1. 使用reposync同步openstack源到本地。



/etc/yum.repos.d/Centos-Base.repo

/etc/yum.repos.d/epel.repo

/etc/yum.repos.d/openstack-queens.repo

/etc/yum.repos.d/ceph-luminous.repo

/etc/yum.repos.d/qemu-ev.repo

reposync –p /opt

2．使用createrepo创建rpm包之间的依赖关系

Createrepo –p /opt/xxx

3. 使用docker容器搭建nginx服务器。

可参考博客：<https://www.cnblogs.com/yufeng218/p/9398809.html>

将我们的镜像源执行nginx的根目录

4. 配置客户端使用本地yum 源。