OpenStack Quantum 分析

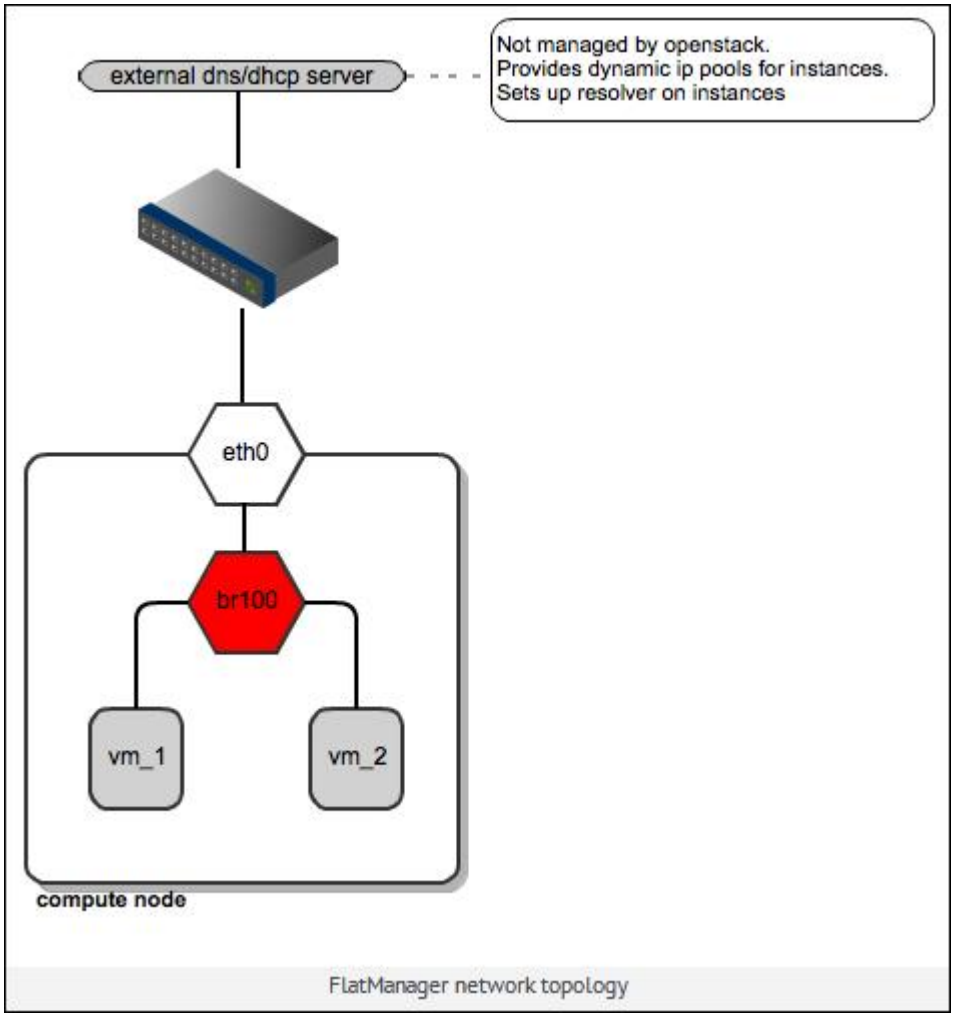
1. Quantum 的介绍和架构

1. 1 Network as a Service
云服务的一种，为云服务的用户提供虚拟化的网络连接，能够根据云平台的负载动态的调整其所提供的资源，如链接带宽，IP 地址等。

1.2 什么是 Quantum
The Quantum project was created to provide a rich and tenant-facing API for defining network connectivity and addressing in the cloud. The Quantum project gives operators the ability to leverage different networking technologies to power their cloud networking.

Quantum 主要提供以下资源的抽象：
网络：一个隔离的第二层（数据链路层）网段
子网：一组 V4 或者 V6 的 IP 地址，以及相关的
端口：某个虚拟设备在该虚拟网络上的连接点，包括其 MAC 和 IP 地址，以及其他配置

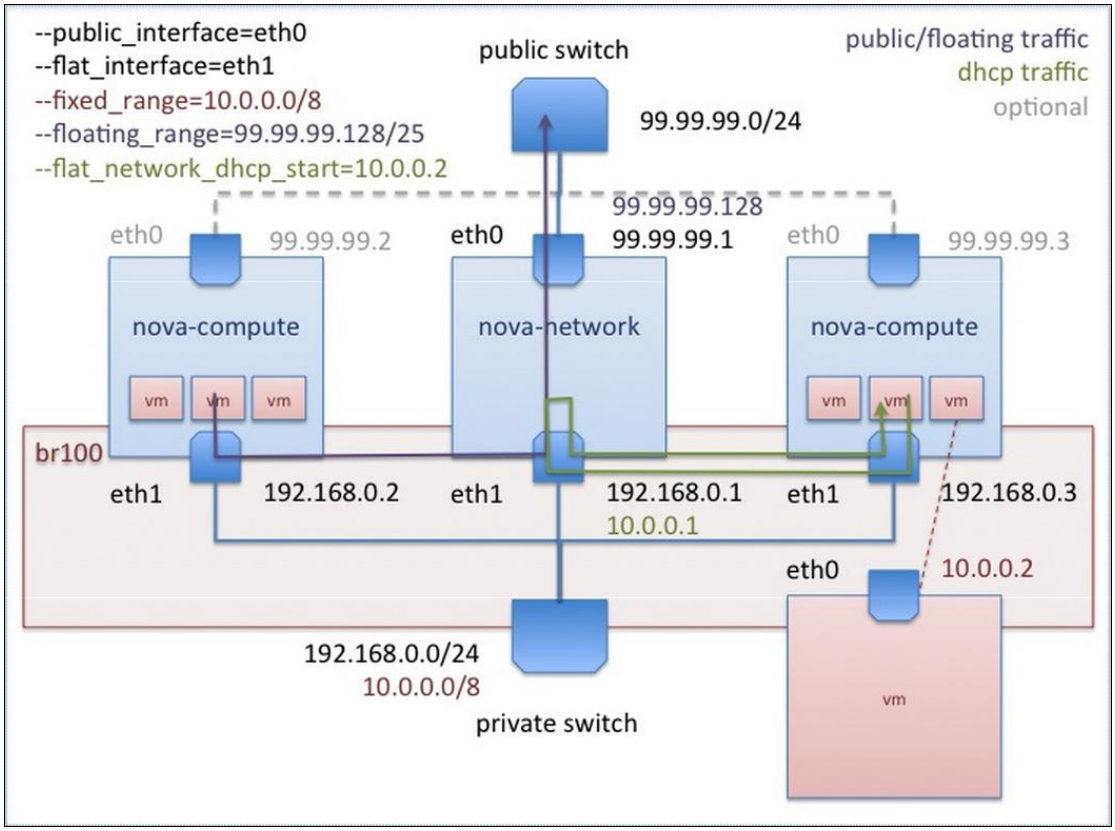
1.3 一些基本概念
1.3.1 Flat
Flat 模式和 FlatDHCP 模式其实区别不大，都是基于网桥网络，只是 FLat 模式需要管理员手动配置（包括配置网桥和外部的 DHCP 设备）。



1.3.2 FlatDHCP

这种模式下与 Flat 模式不同的地方在于有一个 DHCP 进程，每一个运行 nova-network 进程的节点（网络控制节点/nove-network 主机）就是一个单独的网络。Nova 会在 nove-network 主机建立网桥（默认名称 br100，配置项 flat_network_bridge=br100），并给该网桥指定该网络的网关 IP，同时 Nova 在网桥处起一个 DHCP 进程，最后，会建立 iptables 规则（SNAT/DNAT）使虚拟机能够与外界通信，同时与一个 metadata 服务器通信以取得 cloud 内的信息。

计算节点负责创建对应节点的网桥，此时的计算节点网卡可以不需要 IP 地址，因为网桥把虚拟机与 nove-network 主机连接在一个逻辑网络内。虚拟机启动时会发送 dhcpdiscover 以获取 IP 地址。如下图：



图中的虚线（两个计算节点的 eth0）表示节点之间的网络，可选。虚拟机通往外界的数据都要通过 nove-network 主机，DHCP 在网桥处监听，分配 fixed_range 指定的 IP 段。

这种部署方式的缺点之前说过----单节点故障、无二层隔离（即所有的虚拟机都在一个广播域）。

为了解决单节点故障问题，有如下几个方案可供选择：

Option 1: Failover

NTT 实验室的工作人员通过配置网络控制节点使用主备模式实现 HA，但宕机需要 4 秒的时间恢复，不能用于实时应用

Option 2: Multi-nic

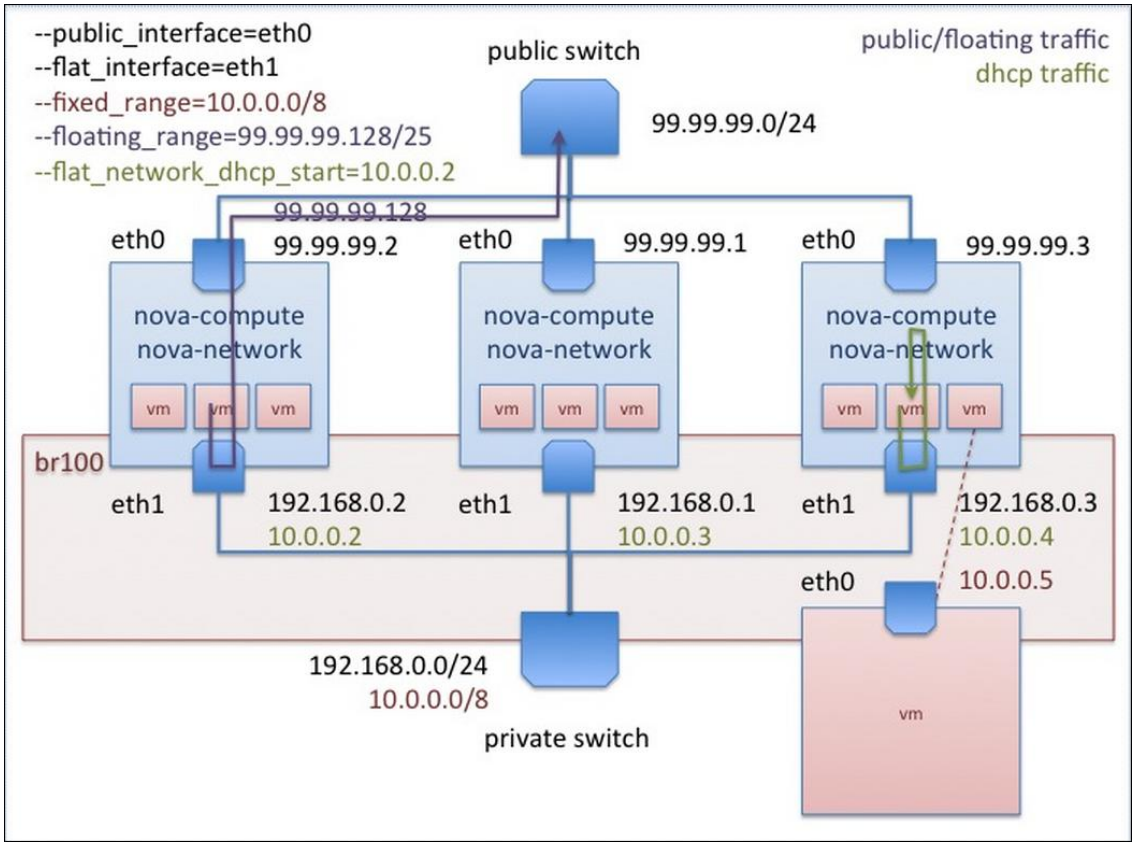
这种模式使虚拟机桥接到不同的网络，每个网络中为虚拟机创建网卡、指定 IP，每个网络可以有自己的 nove-network 主机作为网关。虚拟机可能会有多个路由，如果一个失败，可以选择使用另一个，该方式的缺点是对于用户来说，需要感知多个网络，同时指定切换策略。同时，必须为每一个网卡关联 floating_ip。

Option 3: HW Gateway

可以配置 dnsmasq 使用外部硬件网关作为虚拟机网关。可以配置 dhcpoption=3,<ip of gateway>。这需要人工干预，即需要在外部网关设备上设置 IP 的转发规则（而不是在 nove-network 主机）.该方式把 HA 的任务交给更可靠的硬件设备，但 nova-network 仍需要负责 floating_ip natting 和 DHCP，所以仍需要主备策略保证。

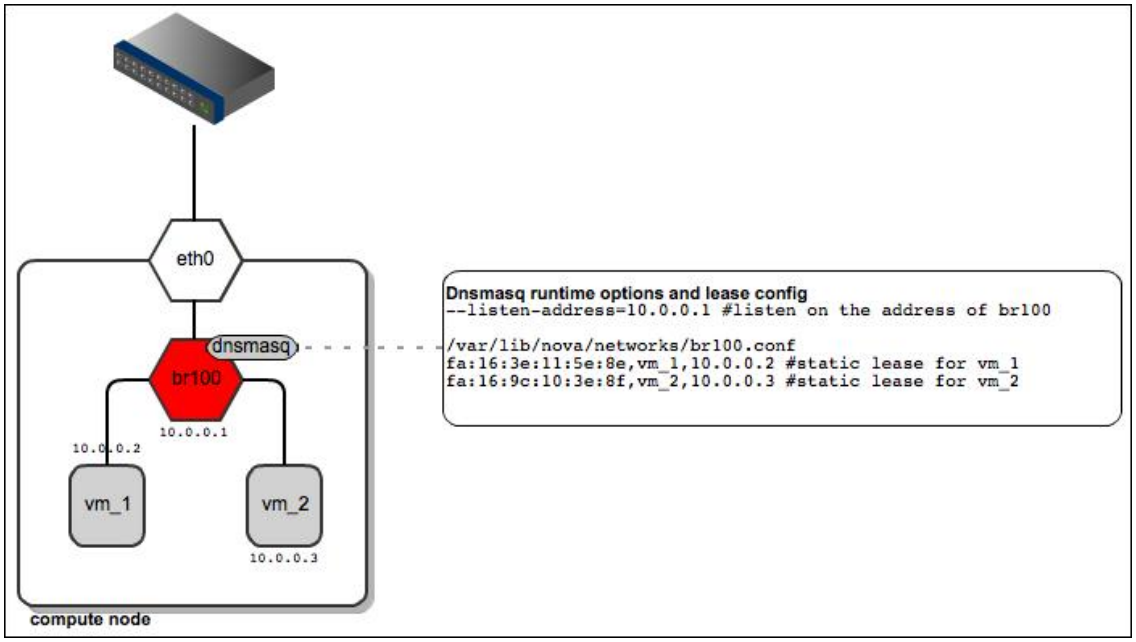
Option 4: Multi-host

该方式被认为是最佳的解决方案。即在每个计算节点上部署 nova-network，这样每个计算节点为自己主机上的虚拟机负责，为每个计算节点创建网桥并为网桥指定不同的 IP（从 fixed_ip 中取）作为虚拟机网关，在每个主机起 DHCP 服务，同时要求每个计算节点至少两个物理网卡，每个计算节点负责自己的虚拟机与外网的通信。如下图：



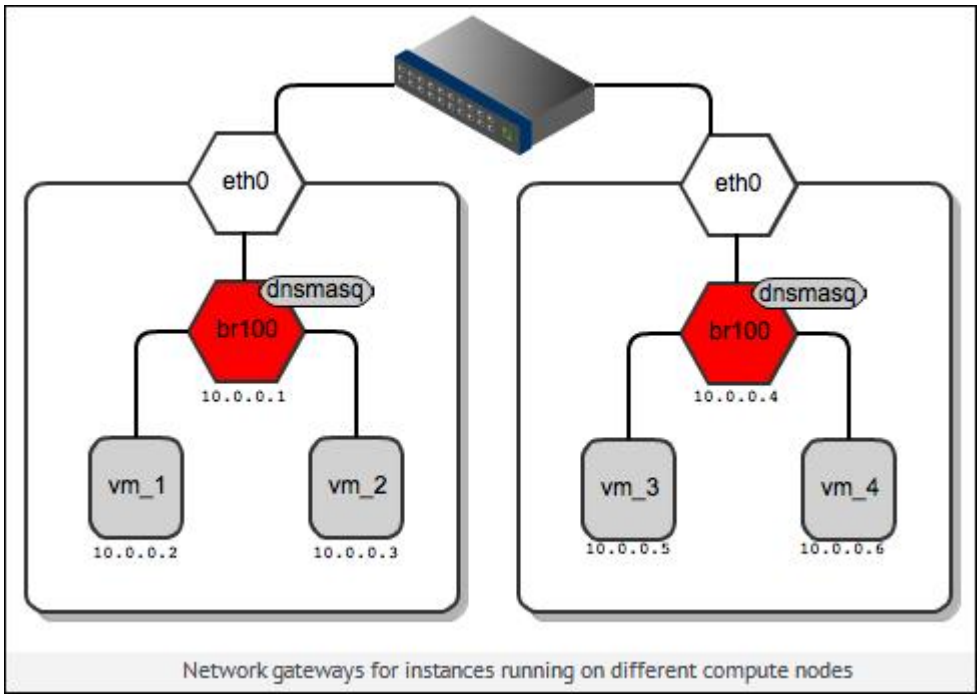
在每个计算节点：

从“flat”IP 池中取出一个给网桥，创建 dnsmasq DHCP 服务并在网桥 IP 监听，在该节点上创建的所有虚拟机的默认网关为网桥 IP。



FlatDHCPManager 在每个计算节点上创建一个静态的租约文件记录虚拟机的私有 IP，文件的数据是从 DB 中获取，包括 MAC、IP、虚拟机名。每个 dnsmasq 只负责为本节点的虚拟机发放 IP，所以从”instance”表的”host”字段过滤即可。如下图所示，每个计算节点的虚拟

机都有一个默认网关：



Network gateways for instances running on different compute nodes

登录虚拟机查看验证：

```
root@vm_1:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.0.1        0.0.0.0         UG    0      0        0 eth0
root@vm_3:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.0.4        0.0.0.0         UG    0      0        0 eth0
```

默认情况下，一个域中的所有虚拟机都可以看到彼此，而不管虚拟机属于哪个租户，但可以通过配置来强制虚拟机间的隔离：

```
allow_same_net_traffic=False
```

该配置通过 iptables 规则来阻止虚拟机间的通信（即便是同一个租户的虚拟机），除非使用安全组策略实现通信。

配置：

```
# 使用的网络模式
network_manager=nova.network.manager.FlatDHCPManager
```

```
# 连接虚拟机的网桥名称
flat_network_bridge=br100

# 网桥绑定的网卡
flat_interface=eth0

# 在 flat 模式，下面的配置允许在虚拟机启动前将 IP 地址注入到镜像的/etc/network/interfaces
flat_injected=True

# 私有 IP 池
fixed_range=YOUR_IP_RANGE_FOR_FIXED_IP_NETWORK
```

1.3.3 VLAN

1.3.3.1 与 Flat 模式的区别

在 Flat 模式下，管理员的工作流程应该是这样的：

1) 为所有租户创建一个 IP 池 nova-manage network create --fixed_range_v4=10.0.0.0/16 --label=public

2) 创建租户

3) 租户创建虚拟机，为虚拟机分配 IP 池中的可用 IP

数据库中的虚拟机信息可能是这样：

tenant_1:

ID	Name	Status	Networks
bf5c7bb3-0c4d-4e6d-ab23-07518fd047ea	tenant1_vm	ACTIVE	private=10.0.0.3

tenant_2:

ID	Name	Status	Networks
8c7f2f71-bc0d-4dc4-879c-d386f99b7b22	tenant2_vm	ACTIVE	private=10.0.0.4

我们看到，两个虚拟机在同一个网段。

而在 Vlan 模式下，流程变成如下：

1) 创建新的租户，并记下租户的标识

2) 为该租户创建独占的 fixed_ip 段 nova-manage network create --fixed_range_v4=10.0.1.0/24 --vlan=102 --project_id="tenantID"

3) 租户创建虚拟机，从租户的私有 IP 段内分配 IP 给虚拟机

所以，与 Flat 模式相比，Vlan 模式为网络增加了两个东西：将网络与租户关联，为网络分配一个 vlan 号。

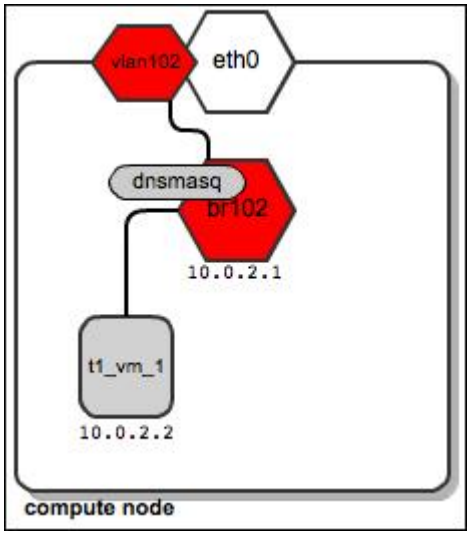
1.3.3.2 多 nova-network 主机部署的 VLAN 模式

VlanManager 做三件事：

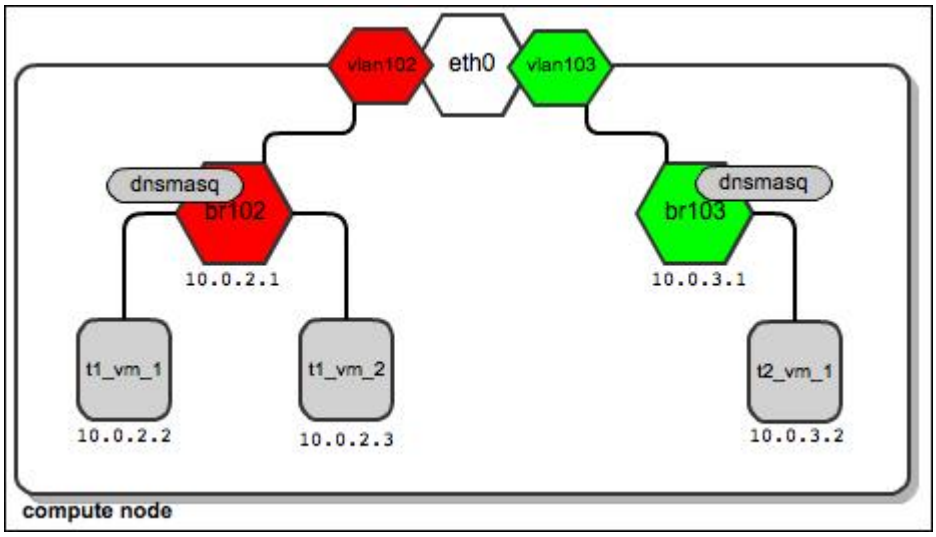
1）在计算节点为租户的网络创建独占的网桥

2）在计算节点的物理网口 eth0 之上创建 vlan 接口（虚拟接口）

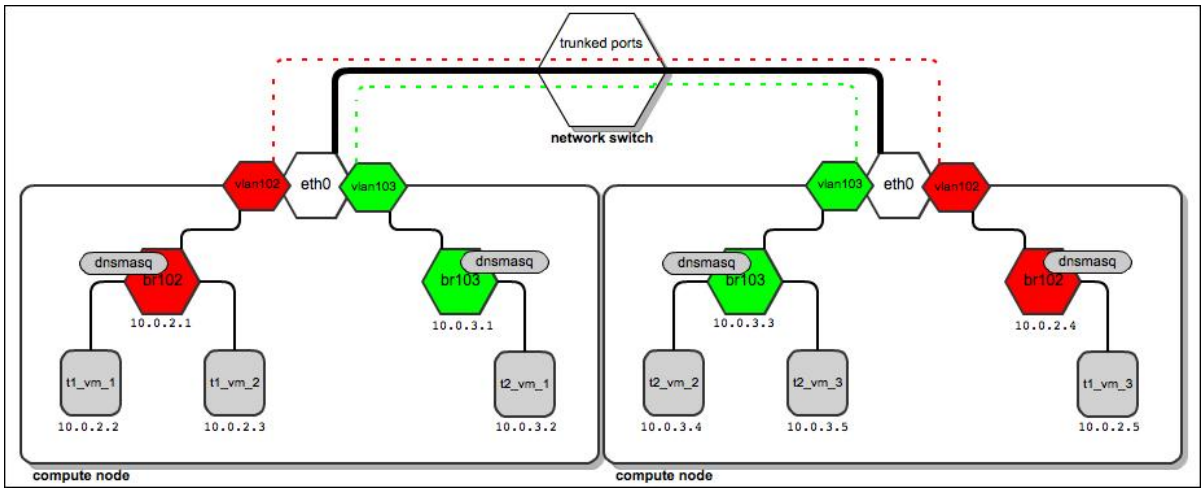
3）在网桥处关联一个 dnsmasq 进程，为虚拟机分配 IP



一个计算节点上有多个租户虚拟机的场景如下：



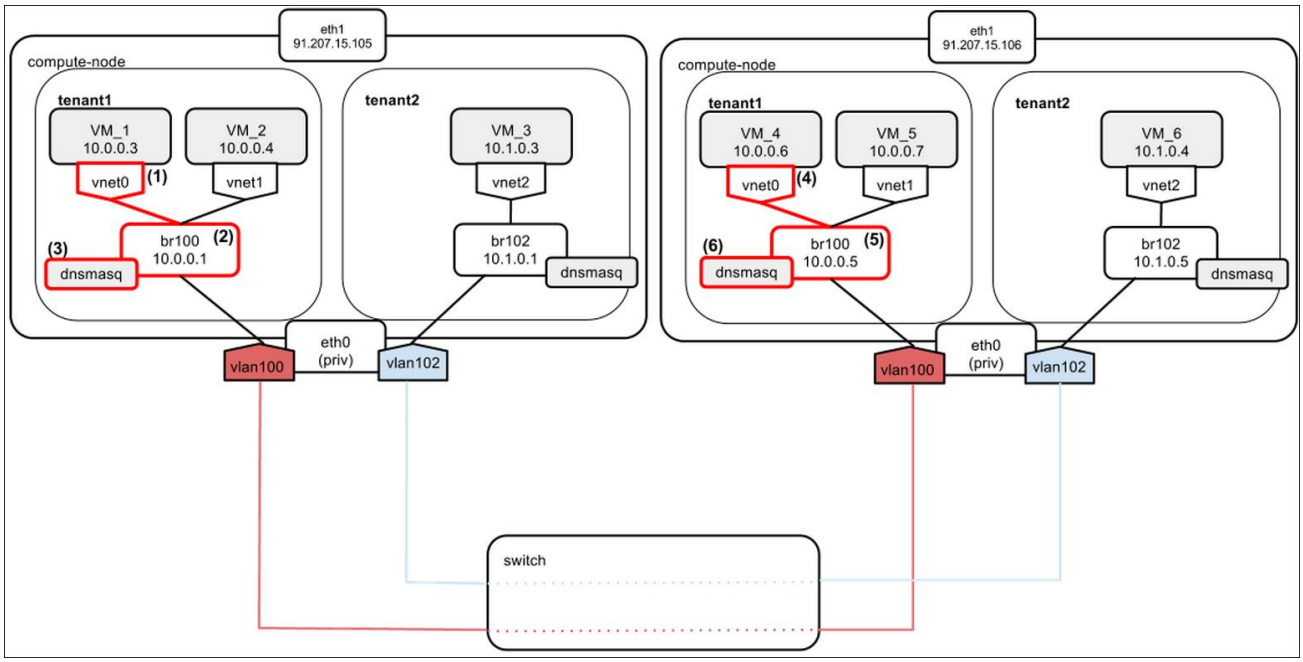
多个计算节点上场景：



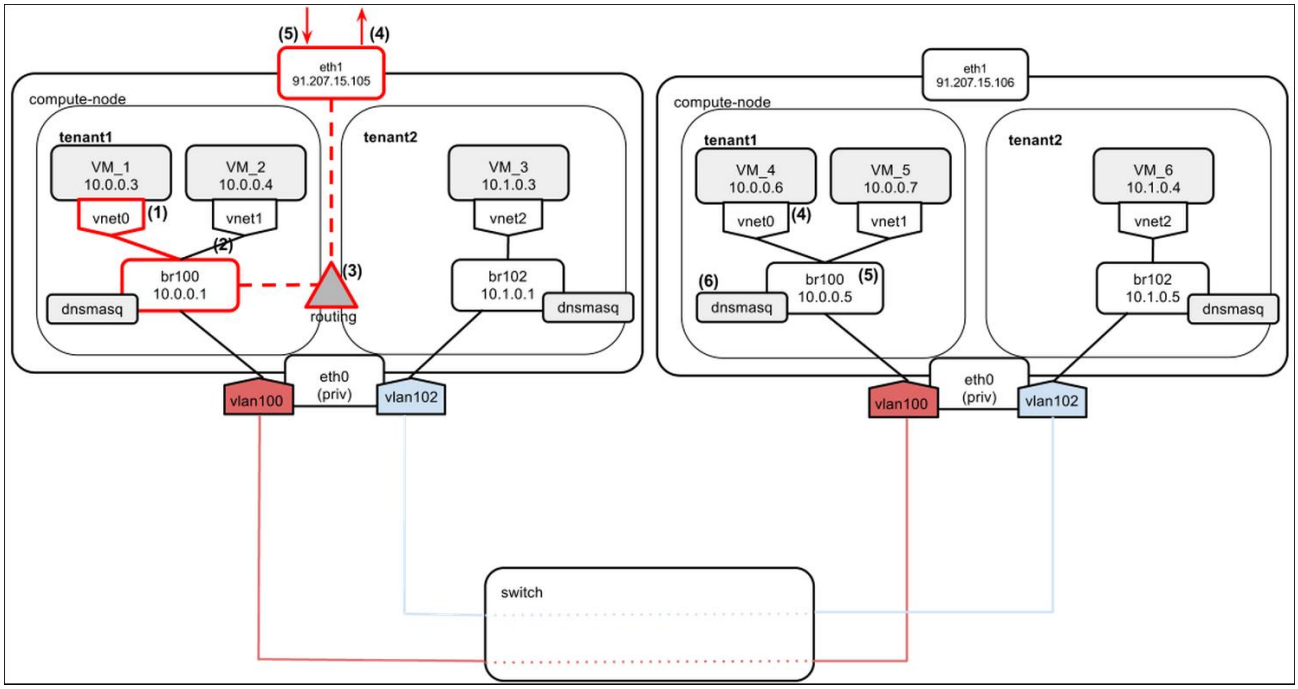
所以有以下结论：在一个计算节点上有几个租户，就有几个网桥，就会创建几个 dnsmasq 进程。而且不同租户的虚拟机之间不能通信。同时，依赖于系统管理员配置交换机。 上图

中，租户“t1”从 10.0.2.2 虚拟机 pingIP 地址 10.0.2.5 的通信过程：
1）首先数据包从 10.0.2.2 发往网桥 br102，在该数据包打上 vlan102 的标签发往交换机
2）交换机把数据包传递到第二个节点，此时会校验数据包的 vlan 标签
3）如果校验成功，节点把数据包发往 vlan102 接口
4）vlan102 接口把 vlan 标签从数据包中剥离，以便可以发往虚拟机
5）数据包通过 br102，最终到达 10.0.2.5

1.3.3.3　multi-host 部署模式下的通信流程
场景 1，tenant1 创建虚拟机：

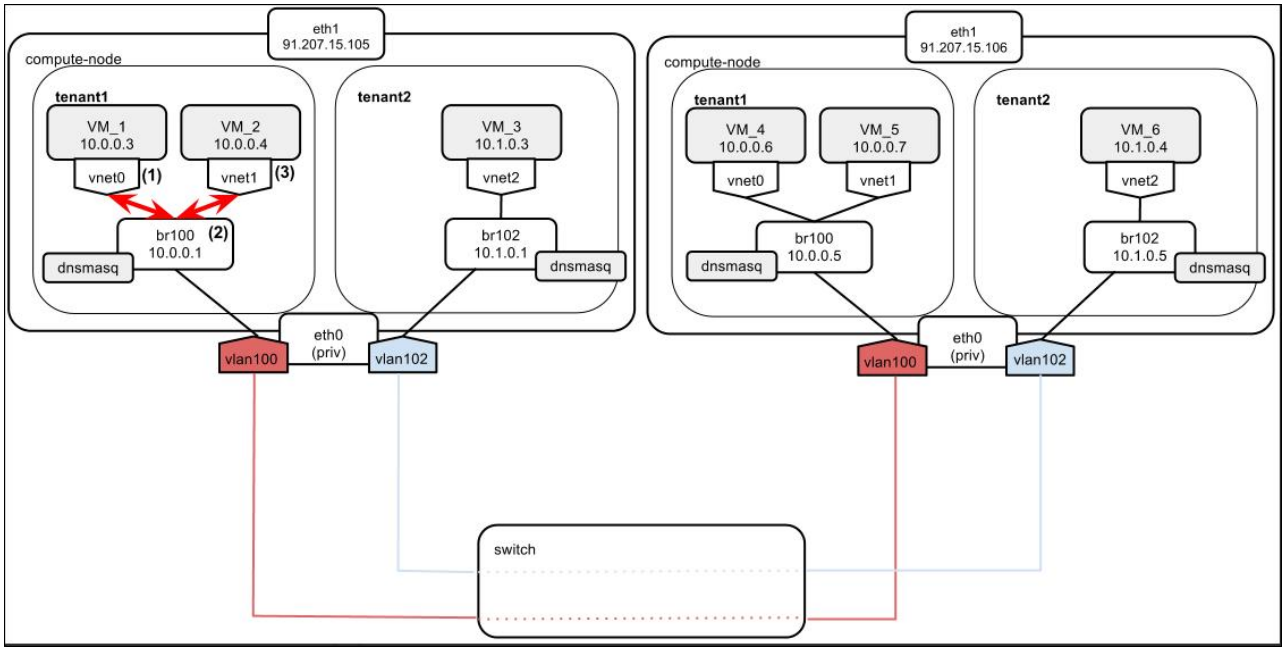


不同计算节点上的虚拟机有不同的网关。
场景 2，VM_1 访问外网（8.8.8.8），且该虚拟机只有 fixed_ip



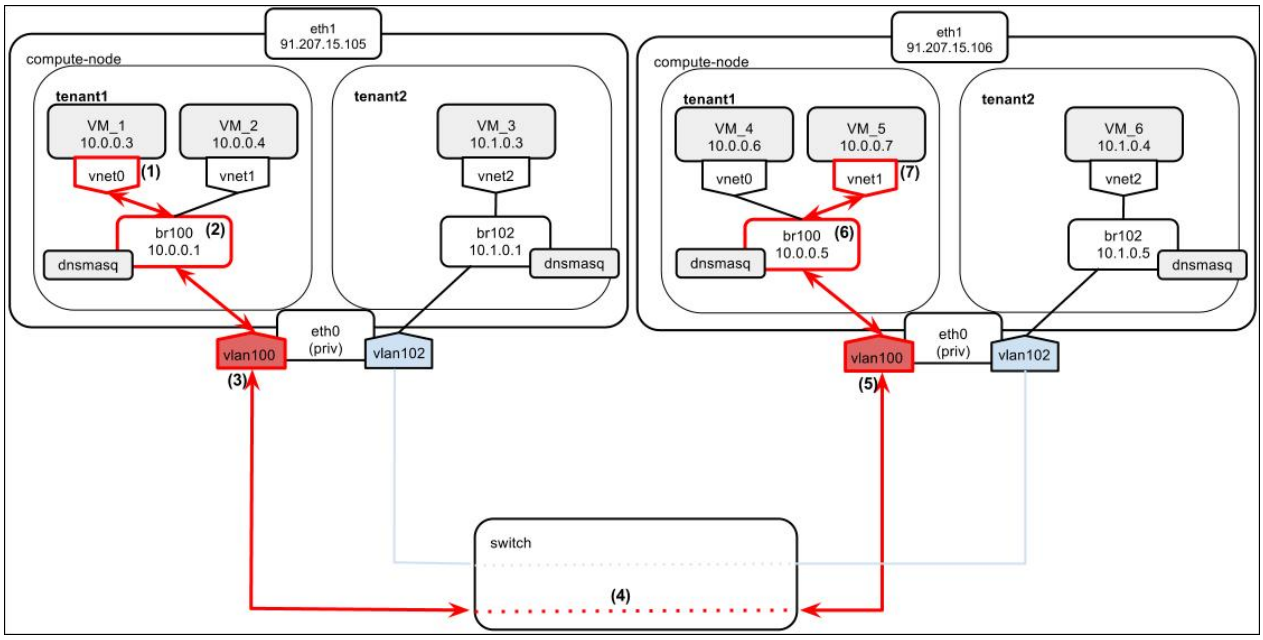
1）VM_1 发送 ping 数据包
2）目的地址不在本网段，于是数据包发往虚拟机的网关（10.0.0.1）
3）计算节点查看路由表，没有发现对应的路由规则，于是发送到计算节点默认网关（91.207.15.105）
4）iptables 的 SNAT 规则处理，nova-network-snat -s 10.0.0.0/24 -j SNAT --to-source 91.207.15.105 该规则是根据配置文件中的 routing_source_ip=91.207.15.105
5.）数据返回

场景 3，用户从 VM_1pingVM_2：



由于两个虚拟机在同一个计算节点，且属于同一个租户，所以是简单的二层通信。

场景 4，用户从 VM_1 ping VM_5：



1）两个虚拟机在同一网段，但在不同计算节点上。VM_1 发送 ARP 广播包查询 VM_5 的 MAC 地址
2）广播包到达 br100
3）数据包转发到 vlan100（带 vlan 标签）
4）数据包通过物理交换机（交换机配置为”trunk”模式）
5）数据包到达第二个计算节点，因为带有 vlan100 标签，所以只有 vlan100 接口能接收
6）通过 br100
7）VM_5 收到广播并返回响应

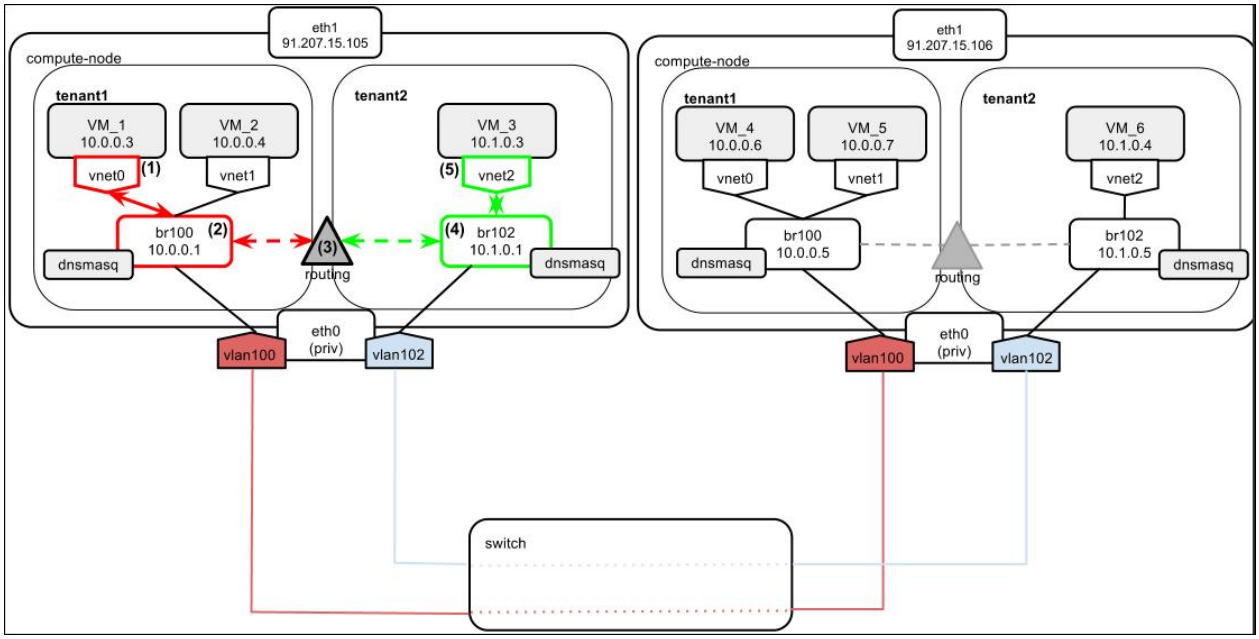
有这样的需求，可能不同的租户是来自同一个公司的不同开发者或部门，所以他们之间需要相互通信，此时就需要设置安全组规则：

```
tenant1: nova secgroup-add-rule default tcp 1 65535 10.1.0.0/24
```

```
tenant2: nova secgroup-add-rule default tcp 1 65535 10.0.0.0/24
```

这样，两个租户的虚拟机之间就可以相互通信。

场景 5，配置了安全组规则，VM_1 ping VM_3：



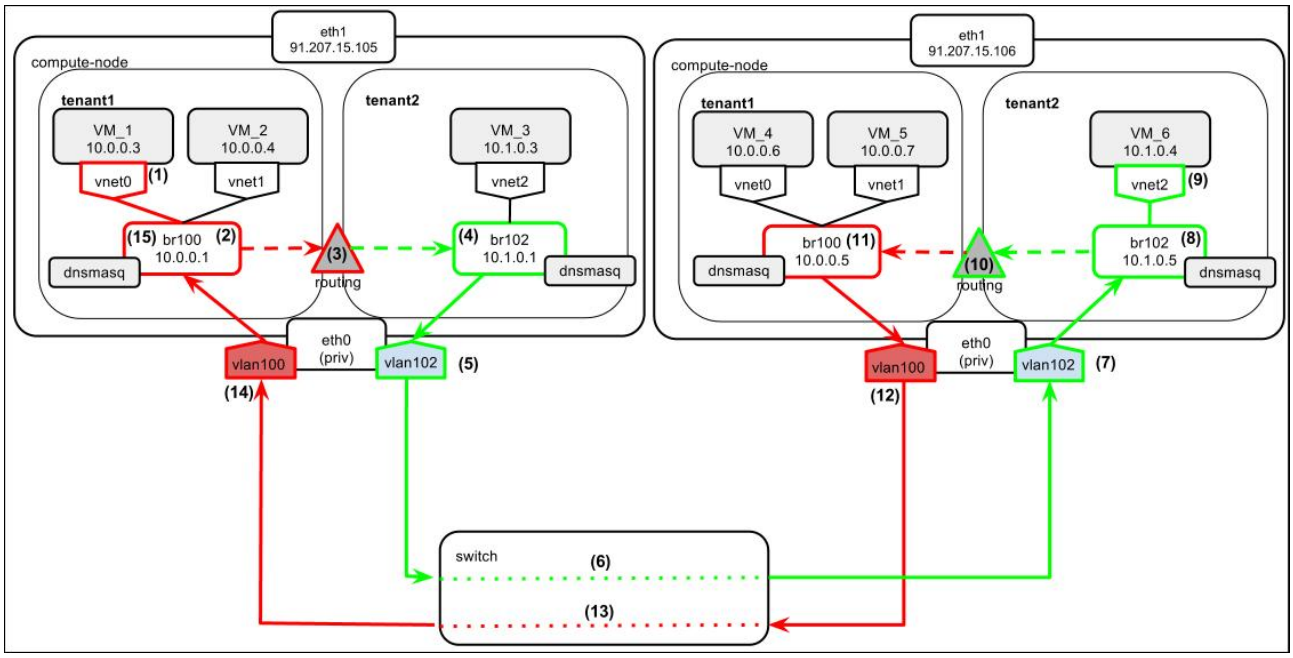
1）两个虚拟机属于不同的租户，不同的网段，但在同一个计算节点。于是来自 VM_1 的包发送到默认网关 10.0.0.1
2）包到达 br100
3）计算节点根据路由表将包路由到 br102
4）包到达 br102，根据 ARP 广播找到 VM_3 的 MAC 地址
5）VM_3 返回 MAC 地址。因为两个虚拟机不再同一个网段，VM_3 会把数据包发到默认网关 10.1.0.1，数据包随后会被路由到租户 1 的网络。

看一下两个计算节点的相关路由：
10.0.0.0/24 dev br100　proto kernel　scope link
10.1.0.0/24 dev br102　proto kernel　scope link

```
10.0.0.0/24 dev br100  proto kernel  scope link
```

10.1.0.0/24 dev br102 proto kernel scope link

场景 6，VM_1pingVM_6



1）两个虚拟机属于不同的租户，不同的网段，在不同的计算节点上。VM_1 的包发送到默认网关

2）包到达 br100

3）计算节点看到目的地址（10.1.0.0/24）的路由是 br102，于是包被路由到 br102

4）现在数据包在租户 2 的二层网络内

5）获取一个 vlan 标签

6）数据包通过交换机

7）包到达其他计算节点，因为带有 vlan102 的标签，于是通过 vlan102 接口，标签被剔除

8）包通过 br102 到达 VM_6

9）VM_6 向 VM_1 回响应（目的地址 10.0.0.3），包发送到默认网关 10.1.0.5

10）包被路由到 br100

11）现在数据包在租户 1 的二层网络内

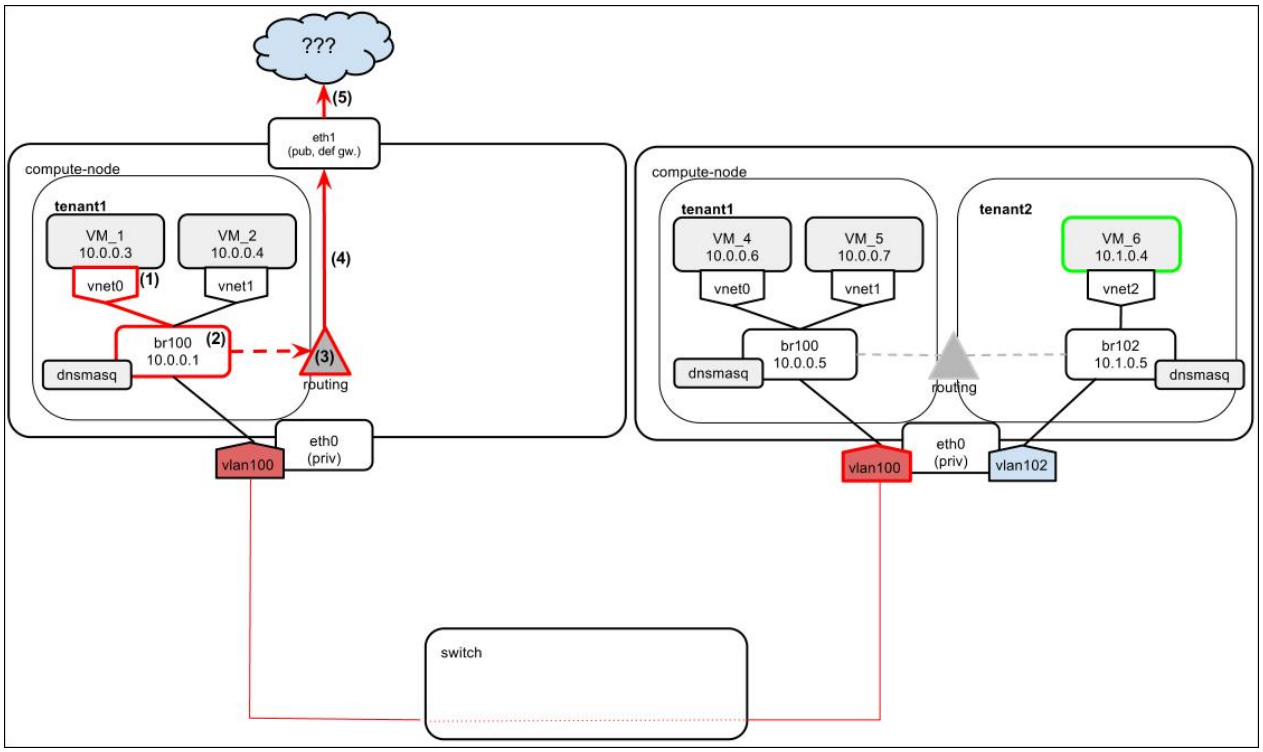
12）打 vlan100 标签

13）通过交换机

14）到达左边计算节点的物理网络接口，因为属于 vlan100，所以被转发到 vlan100 接口，剔除 vlan 标签

15）数据包通过 br100 到达 VM_1

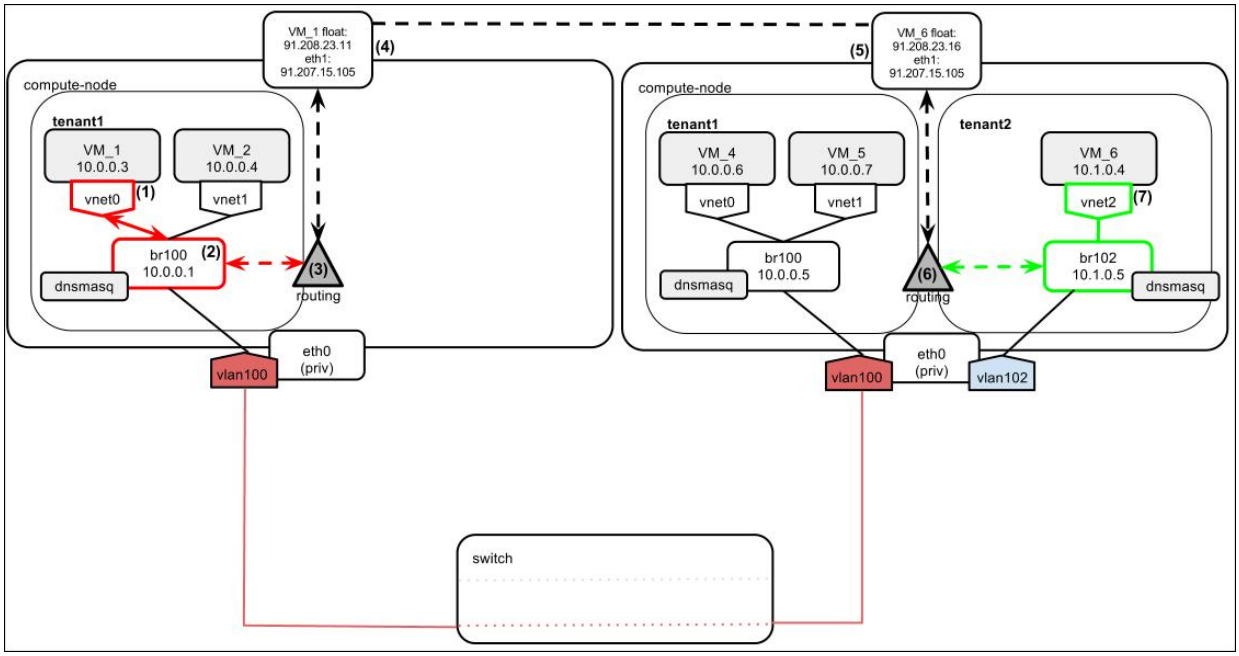
但是，如果是下面这个情况：



因为只有创建虚拟机时才会创建必要的网桥，所以在左边计算节点上没有租户 2 的网桥，此时从 VM_1 ping VM_6 时，数据包不会被路由到右边计算节点。

结论：vlan 模式下，不同租户的虚拟机之间通信不能依赖于 fixed_ip。

场景 7，VM_1 和 VM_6 都关联了 floating_ip，VM_1 ping VM_6：



假设虚拟机分配的 floating_ip 如下：

tenant1: VM_1: 91.208.23.11

tenant2: VM_6: 91.208.23.16

1）从 VM_1ping 91.208.23.16，数据包源地址 10.0.0.3，目的地址 91.208.23.16

2）数据包被发送到默认网关 10.0.0.1

3）计算节点的路由允许其被发送到 eth1
4）iptables 的 SNAT 规则处理，源地址改为 91.208.23.11
5）数据包到达另一个计算节点，由该计算节点上的 iptables 的 DNAT 规则处理，目的地址改为 10.1.0.4
6）根据目的地址，数据包被路由到 br102
7）到达目标虚拟机 VM_6
ICMP 的响应走的路径类似，但注意的是，ICMP 响应被认为与 ICMP 请求关联，所以在左边计算节点上没有显式的 DNAT 的处理，由系统内部处理。

1.3.3.4 配置

```
#指定 network manager
network_manager=nova.network.manager.VlanManager

#使用哪个接口创建 vlan
vlan_interface=eth0

#起始 vlan 号。这种情况下，小于106的 vlan 号可以用于内部通信
vlan_start=100
```

1.3.3.5 不足
1）vlan 模式下，不同租户使用的 IP 地址不能重复（在 AmazonVPC 中是可以重复的）
2）由于 vlan 标签的标志位是 12bit，所以 vlan 号的范围是 1-4096，也就是系统中最多只能有 4096 个租户，不适用于公有云（Nicira NVP 作为 Quantum 的插件解决了这个问题）

1.3.4 Floating IP
1.3.4.1 分配 floating_ip
首先，管理员先配置 IP 池：

```
nova-manage floating create --ip_range=PUBLICLY_ROUTABLE_IP_RANGE  --pool POOL_NAME
```

用户创建虚拟机：

```
+--------------------------------------+---------+--------+------------------------------+
|                  ID                  |   Name  | Status |           Networks           |
+--------------------------------------+---------+--------+------------------------------+
| 79935433-241a-4268-8aea-5570d74fcf42 |  inst1  | ACTIVE | private=10.0.0.4             |
+--------------------------------------+---------+--------+------------------------------+
```

查询可用的 floating_ip：

```
nova floating-ip-pool-list
```

从“pub/test”获取一个 IP：

```
nova floating-ip-create pub
+---------------+-------------+----------+-----+
|       Ip      | Instance Id | Fixed Ip | Pool |
+---------------+-------------+----------+-----+
| 172.24.4.225  |     None    |   None   | pub  |
+---------------+-------------+----------+-----+
```

为虚拟机绑定 IP：

```
nova add-floating-ip 79935433-241a-4268-8aea-5570d74fcf42 172.24.4.225
```

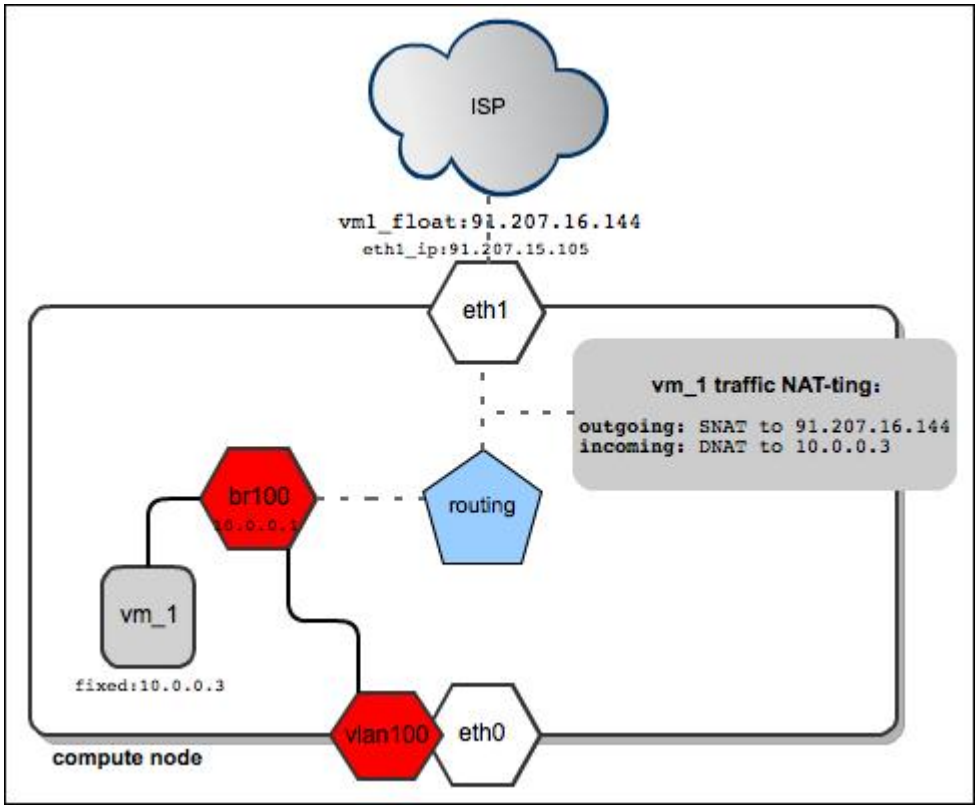
查看 floating_ip 的分配情况：

```
nova floating-ip-list
+---------------+--------------------------------------+----------+------+
|       Ip      |                Instance Id           | Fixed Ip | Pool |
+---------------+--------------------------------------+----------+------+
| 172.24.4.225 | 79935433-241a-4268-8aea-5570d74fcf42 | 10.0.0.4 | pub  |
+---------------+--------------------------------------+----------+------+
```

到此为止，虚拟机就能通过外网访问。

1.3.4.2 引起的变化

虚拟机在绑定 floating_ip 后，内部的网路配置并没有变化，所有的配置都由 nova-network 完成.先看下图示（vlanmanager 模式的 multi-host 部署）：



eth1 连接外网（IP：91.207.15.105，该 IP 也是该计算节点的默认网关），eth0 连接内网（没有指定 IP），当虚拟机关联 floating_ip 时，两个东西发生变化：

1）floating_ip 作为计算节点 eth1 的 secondary 地址，可以通过”ip addr show eth1”命令查看：

```
inet 91.207.15.105/24 scope global eth1      # primary eth1 ip
inet 91.207.16.144/32 scope global eth1      # floating ip of VM_1
```

2）增加计算节点的 iptables 中 NAT 表的规则：

```
# 这条规则保证了在计算节点上可以访问虚拟机私有 IP
-A nova-network-OUTPUT -d 91.207.16.144/32 -j DNAT --to-destination 10.0.0.3

# 这条规则保证从外网发向91.207.16.144的数据包能转发的10.0.0.3
-A nova-network-PREROUTING -d 91.207.16.144/32 -j DNAT --to-destination 10.0.0.3

# 这条规则保证从虚拟机发往外网的数据包的源 IP 为其 flaoting_IP
-A nova-network-float-snat -s 10.0.0.3/32 -j SNAT --to-source 91.207.16.144
```

相关的 nova-network 代码在 nova/network/linux_net.py 中：

```
def floating_forward_rules(floating_ip, fixed_ip):
    return [('PREROUTING', '-d %s -j DNAT --to %s' % (floating_ip, fixed_ip)),
            ('OUTPUT', '-d %s -j DNAT --to %s' % (floating_ip, fixed_ip)),
            ('float-snat',
             '-s %s -j SNAT --to %s' % (fixed_ip, floating_ip))]
```

1.3.4.3 通信流程

从外网访问虚拟机：

1）首先数据包到达计算节点的 eth1，DNAT 规则开始处理，数据包的目的 IP 变为私有 IP：91.207.16.144 -->10.0.0.3

2）计算节点通过查看路由表，将数据包发往 br100，通过 br100 发往目标虚拟机：

ip route show：

10.0.0.0/24 dev br100

从虚拟机访问外网（以 ping 8.8.8.8 为例）：

1）因为目的 IP 不在虚拟机网段，数据包会发送到虚拟机的默认网关，也就是 br100。

2）计算节点检查路由表，也没发现对应的路由规则，于是发往默认网关 91.207.15.105。

3）数据包由 iptables 的 SNAT 规则处理，源 IP 被修改为虚拟机的 floating_ip(91.207.16.144)。

1.3.4.4 需要注意的点

因为 openstack 有对网络设置的完全的控制权限，因此网络设置很容易被人为操作破坏。如果需要修改 iptables 的行为，最好的方式是修改代码（linux_net.py）。

这从另一个方面也说明，openstack 没有对 iptables 规则的监控策略，如果人为修改了规则，则需要 nova-network 重启才能恢复。举个例子，比如当前的计算节点有如下规则：

-A nova-network-PREROUTING -d 91.207.16.144/32 -j DNAT --to-destination 10.0.0.3

如果管理员不慎使用了：iptables －F －t nat

这样上述的 NAT 规则被清掉，但 eth1 仍然有 secondary 地址 91.207.16.144，当一个发往虚拟机的数据包到达计算节点时，因为没有了 DNAT 规则，所以数据包会直接到达计算节点。直到下次 nova-network 重启才能解决该问题。

1.3.4.5 相关配置

```
# floating_ip 绑定到哪个网络接口，作为该网络接口的 secondary IP
public_interface="eth1"

# 默认的 floating_ip 池
default_floating_pool="pub"

# 是否在创建虚拟机时自动分配 floating_ip
auto_assign_floating_ip=false
```

1.4 Quantum 的软件架构和 Plugin 的作用

2. Quantum 的基本工作过程

2.1 Quantum 的消息处理流程

2.1.1 Paste.deploy 配置

```
[composite:quantumapi_v2_0]
use = call:quantum.auth:pipeline_factory
noauth = extensions quantumapiapp_v2_0
keystone = authtoken keystonecontext extensions quantumapiapp_v2_0
```

2.1.2 authtoken

通过调用 keystone，进行权限的识别。

2.1.3 keystone context

根据鉴权信息（user_id，tenant_id，roles 等），更新请求中的环境上下文。

```
def __call__(self, req):
    # Determine the user ID
    user_id = req.headers.get('X_USER_ID', req.headers.get('X_USER'))
    if not user_id:
        LOG.debug("Neither X_USER_ID nor X_USER found in request")
        return webob.exc.HTTPUnauthorized()

    # Determine the tenant
    tenant_id = req.headers.get('X_TENANT_ID', req.headers.get('X_TENANT'))

    # Suck out the roles
    roles = [r.strip() for r in req.headers.get('X_ROLE', '').split(',')]

    # Create a context with the authentication data
    ctx = context.Context(user_id, tenant_id, roles=roles)

    # Inject the context...
    req.environ['quantum.context'] = ctx

    return self.application
```

在环境上下文中增加quantum.context项

2.1.4 extensions

2.1.4.1 获取 quantum.conf 中 core_plugin 配置的插件类。

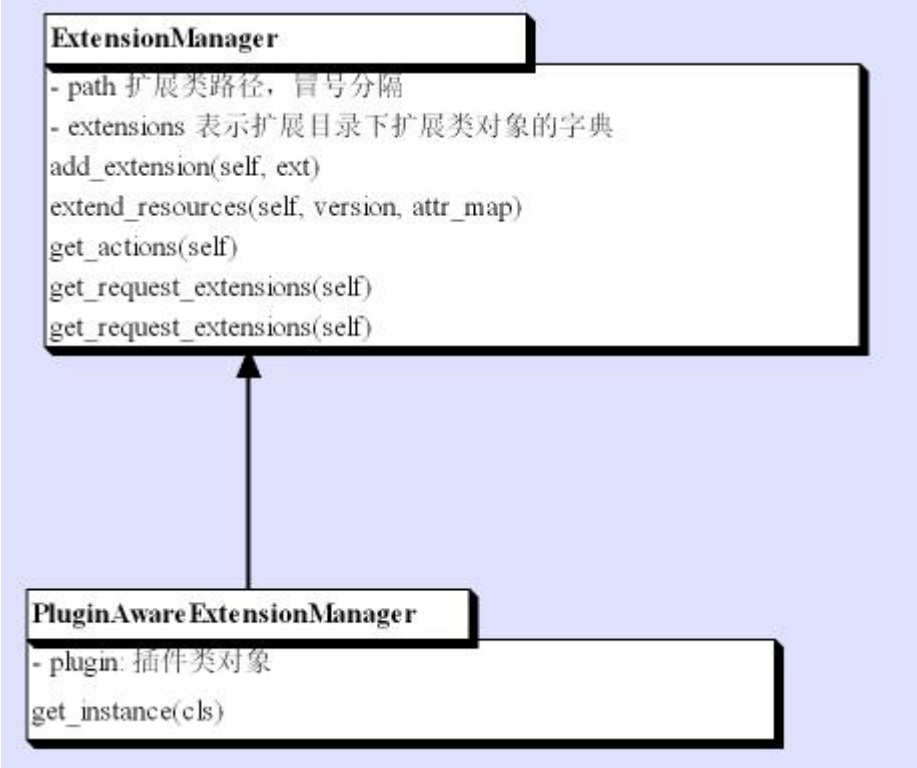
2.1.4.2 如果在 quantum/extensions/extensions.py 中 ENABLED_EXTS 中有该插件的配置信息，默认如下图：

```
ENABLED_EXTS = {
    'quantum.plugins.linuxbridge.lb_quantum_plugin.LinuxBridgePluginV2':
    {
        'ext_alias': ["quotas"],
        'ext_db_models': ['quantum.extensions._quotav2_model.Quota'],
    },
    'quantum.plugins.openvswitch.ovs_quantum_plugin.OVSQuantumPluginV2':
    {
        'ext_alias': ["quotas"],
        'ext_db_models': ['quantum.extensions._quotav2_model.Quota'],
    },
}
```

则加载 ext_db_models 表示的数据库建模类。

2.1.4.3 加载 core_plugin 插件类。

2.1.4.4 初始化 PluginAwareExtensionManager 类及其父类 ExtensionManager，两个类的类图如下：



类 ExtensionManager 初始化主要是加载扩展类目录中的扩展类，代码解析如下：

2.1.4.5 初始化 ExtensionMiddleware

整个初始化过程主要使用从 Ruby 移植到 Python 的 Routes 开发包，用来定义 URL 和应用程序接口之间的映射，这里不是很懂，网上关于 Routes 的资料除了官方文档外几乎没有。

```
    def __init__(self, application,
                 ext_mgr=None):

        self.ext_mgr = (ext_mgr
                        or ExtensionManager(
                        get_extensions_path()))
        # 定义 mapper
        mapper = routes.Mapper()

        # extended resources
        for resource in self.ext_mgr.get_resources():
            LOG.debug(_('Extended resource: %s'),
                      resource.collection)
            for action, method in resource.collection_actions.iteritems():
                path_prefix = ""
                parent = resource.parent
                conditions = dict(method=[method])
                path = "/%s/%s" % (resource.collection, action)
                if parent:
                    path_prefix = "/%s/{%s_id}" % (parent["collection_name"],
                                                   parent["member_name"])
                with mapper.submapper(controller=resource.controller,
                                      action=action,
```

```
                                                path_prefix=path_prefix,
                                                conditions=conditions) as submap:
                        submap.connect(path)
                        submap.connect("%s.:(format)" % path)
                mapper.resource(resource.collection, resource.collection,
                                controller=resource.controller,
                                member=resource.member_actions,
                                parent_resource=resource.parent)

        # extended actions
        action_controllers = self._action_ext_controllers(application,
                                                           self.ext_mgr, mapper)
        for action in self.ext_mgr.get_actions():
            LOG.debug(_('Extended action: %s'), action.action_name)
            controller = action_controllers[action.collection]
            controller.add_action(action.action_name, action.handler)

        # extended requests
        req_controllers = self._request_ext_controllers(application,
                                                         self.ext_mgr, mapper)
        for request_ext in self.ext_mgr.get_request_extensions():
            LOG.debug(_('Extended request: %s'), request_ext.key)
            controller = req_controllers[request_ext.key]
            controller.add_handler(request_ext.handler)

        # 个人理解，开始根据 Mapper 解析 URL，_dispatch()方法根据解析的结果进行处理
        self._router = routes.middleware.RoutesMiddleware(self._dispatch,
                                                           mapper)
        super(ExtensionMiddleware, self).__init__(application)
```

对于每一个资源的处理是在 quantum/api/v2/base.py/Controller 类中

2.1.5 quantumapiapp_v2_0，处理类：quantum/api/v2/router.py::APIRouter
2.1.5.1 初始化过程
2.1.5.1.1 根据扩展类的配置更新
quantum/api/v2/attributes.py::RESOURCE_ATTRIBUTE_MAP 定义的对象属性，因为有的扩展类扩展了标准对象（network, port, subnet）的属性。
2.1.5.1.2 同样使用 Routes，定义 URL 和应用接口的映射关系。
2.1.5.1.3 在父类的初始化中,同样调用:
self._router = routes.middleware.RoutesMiddleware(self._dispatch, self.map)

2.2 关于 Network 的操作
看待 Quantum 网络模型时可以从上层的逻辑概念和下层的物理概念区分。有一些概念或对象只在上层关注，下层的 agent 或物理网络只关心实际的操作和部署。需要上下层配合时，

就需要做映射，比如 networkbinding 表就是用来映射逻辑网络和实际物理网络的关系。

2.2.1 创建 network

2.2.1.1 入口

对资源的处理都在 Mapper 对应的 Controller 中：

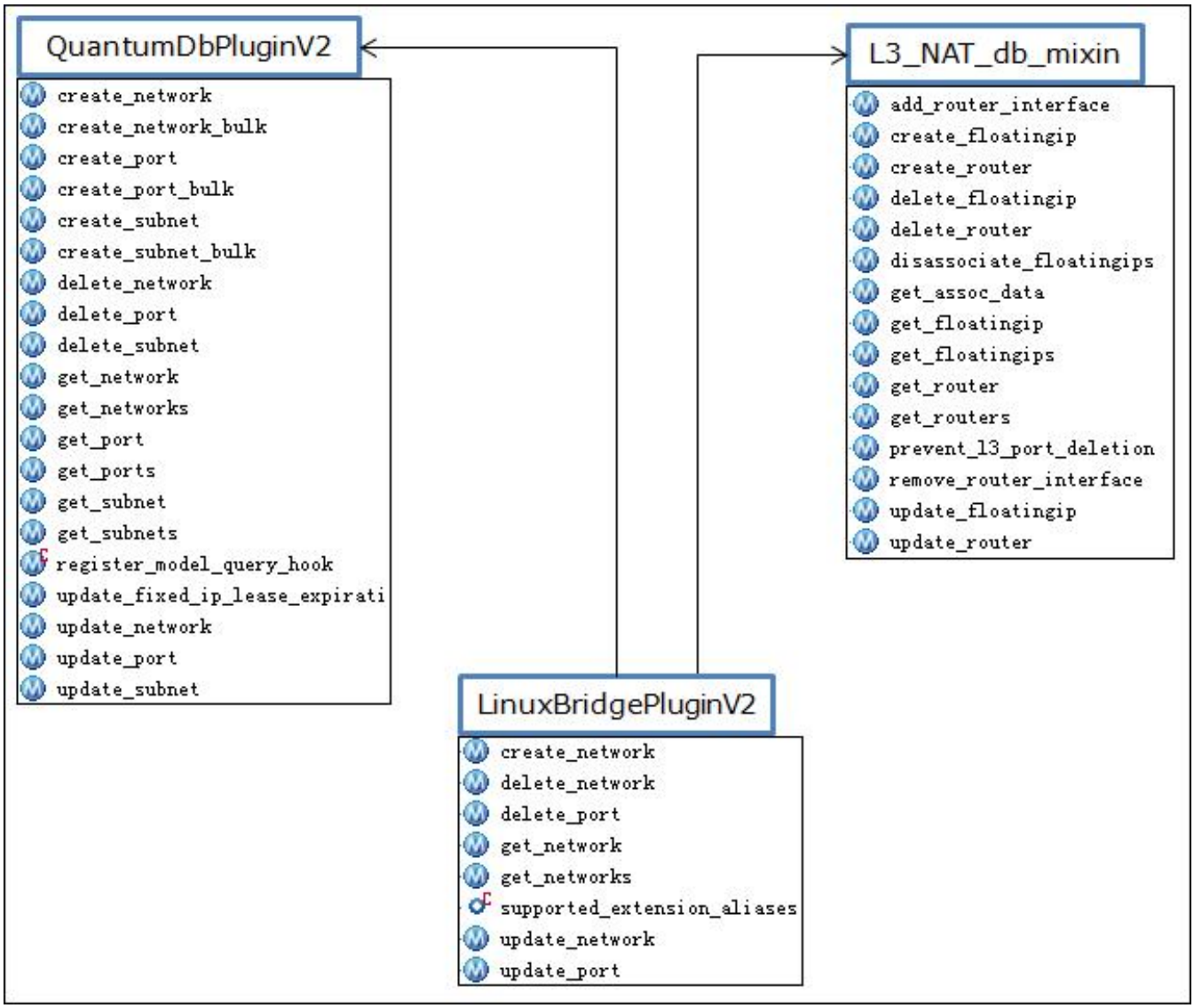
```
controller = base.create_resource(collection, resource,
                                  plugin, params,
                                  allow_bulk=allow_bulk)
mapper_kwargs = dict(controller=controller,
                     requirements=REQUIREMENTS,
                     **col_kwargs)
return mapper.collection(collection, resource,
                         **mapper_kwargs)
```

在 quantum/api/v2/base.py 的 Controller 类中的 create 函数中：

action = "create_%s" % self._resource

```
# 获取plugin中的方法
obj_creator = getattr(self._plugin, action)
if self._collection in body:
    # Emulate atomic bulk behavior
    objs = self._emulate_bulk_create(obj_creator, request, body)
    return notify({self._collection: objs})
else:
    kwargs = {self._resource: body}
    # 调用
    obj = obj_creator(request.context, **kwargs)
    return notify({self._resource: self._view(obj)})
```

本文以 linuxbridge 插件为例，所以代码中的 self._plugin=quantum/plugin/linuxbridge/lb_quantum_plugin.py/LinuxBridgePluginV2（需要注意的是，该类继承自 quantum/db/db_base_plugin_v2.py/QuantumDbPluginV2 和 quantum/db/l3_db.py/L3_NAT_db_mixin）。类的继承关系如下：



2.2.1.2 LinuxBridge 插件的初始化

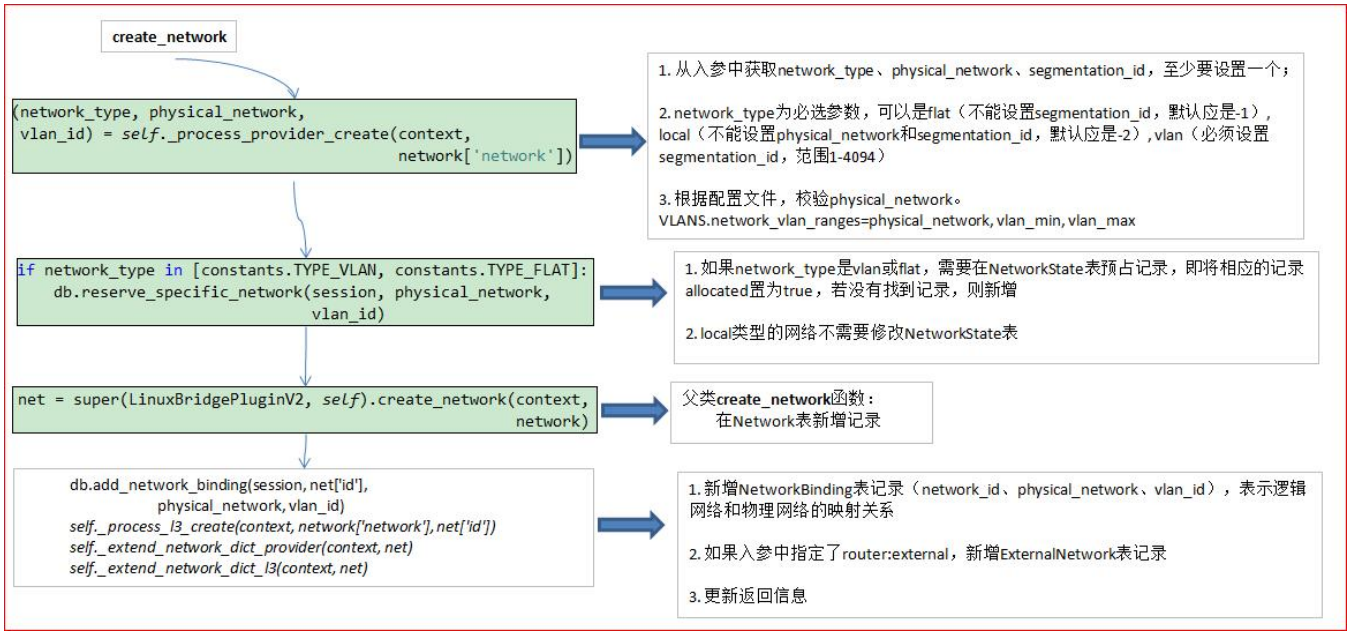
```
def __init__(self):
    # 连接数据库相关
    db.initialize()
    # 初始化network_vlan_ranges配置项，存到network_vlan_ranges变量
    self._parse_network_vlan_ranges()
    # 根据配置中physical_network和vlan信息更新networkstate表
    db.sync_network_states(self.network_vlan_ranges)
    self.tenant_network_type = cfg.CONF.VLANS.tenant_network_type
    if self.tenant_network_type not in [constants.TYPE_LOCAL,
                                        constants.TYPE_VLAN,
                                        constants.TYPE_NONE]:
        LOG.error("Invalid tenant_network_type: %s" %
                  self.tenant_network_type)
        sys.exit(1)
    self.agent_rpc = cfg.CONF.AGENT.rpc
    # 初始化RPC接收端和发送端
    self._setup_rpc()
    LOG.debug("Linux Bridge Plugin initialization complete")

def _setup_rpc(self):
    # RPC support
    self.topic = topics.PLUGIN
    self.rpc_context = context.RequestContext('quantum', 'quantum',
                                              is_admin=False)
    # 获取一个ConnectionContext对象
    self.conn = rpc.create_connection(new=True)
    self.callbacks = LinuxBridgeRpcCallbacks(self.rpc_context)
    # RpcDispatcher对象有一个属性callbacks
    self.dispatcher = self.callbacks.create_rpc_dispatcher()
    # 创建一个TopicConsumer，routing-key='q.plugin', exchange-name='quantum'
    self.conn.create_consumer(self.topic, self.dispatcher,
                              fanout=False)
    # Consume from all consumers in a thread
    self.conn.consume_in_thread()
    # 创建RPC发送端
    self.notifier = AgentNotifierApi(topics.AGENT)
```

networkstate表中三个字段：physical_network, vlan_id, allocated。
1. 如果某条记录的physical_network不在配置中，删除记录
2. 对于配置中有而表中没有的physical_network和vlan，在表中新增记录，状态为未分配
3. 如果某条记录的vlan不在配置中，且状态为空闲，删除记录

TopicConsumer用来接收agent的请求

向agent发送消息，目前有两个消息：network_delete和port_update

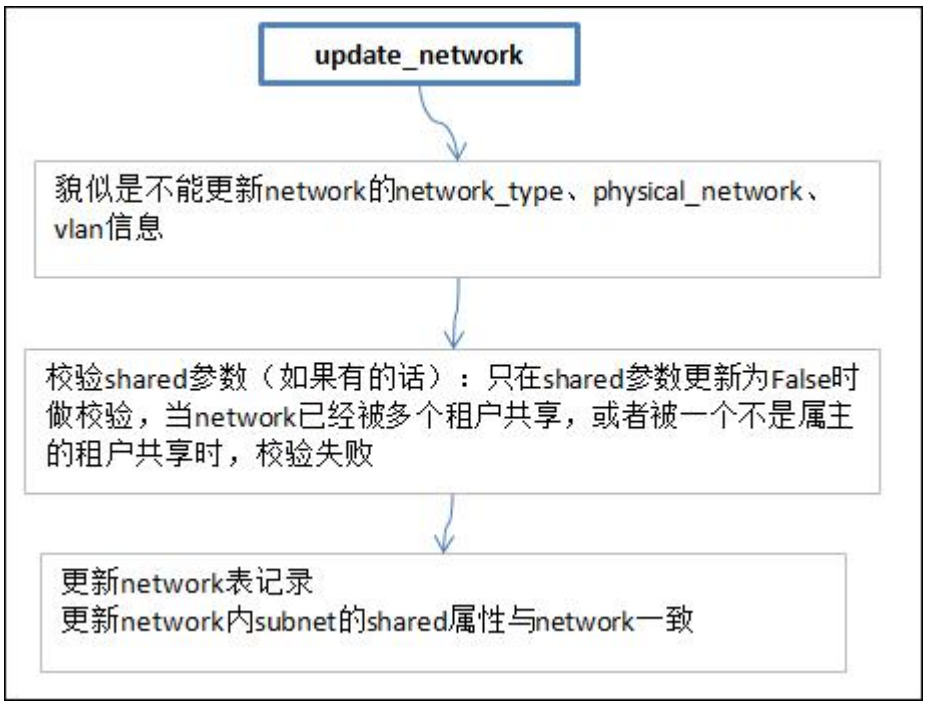
2.2.1.3 流程



2.2.2 删除 Network

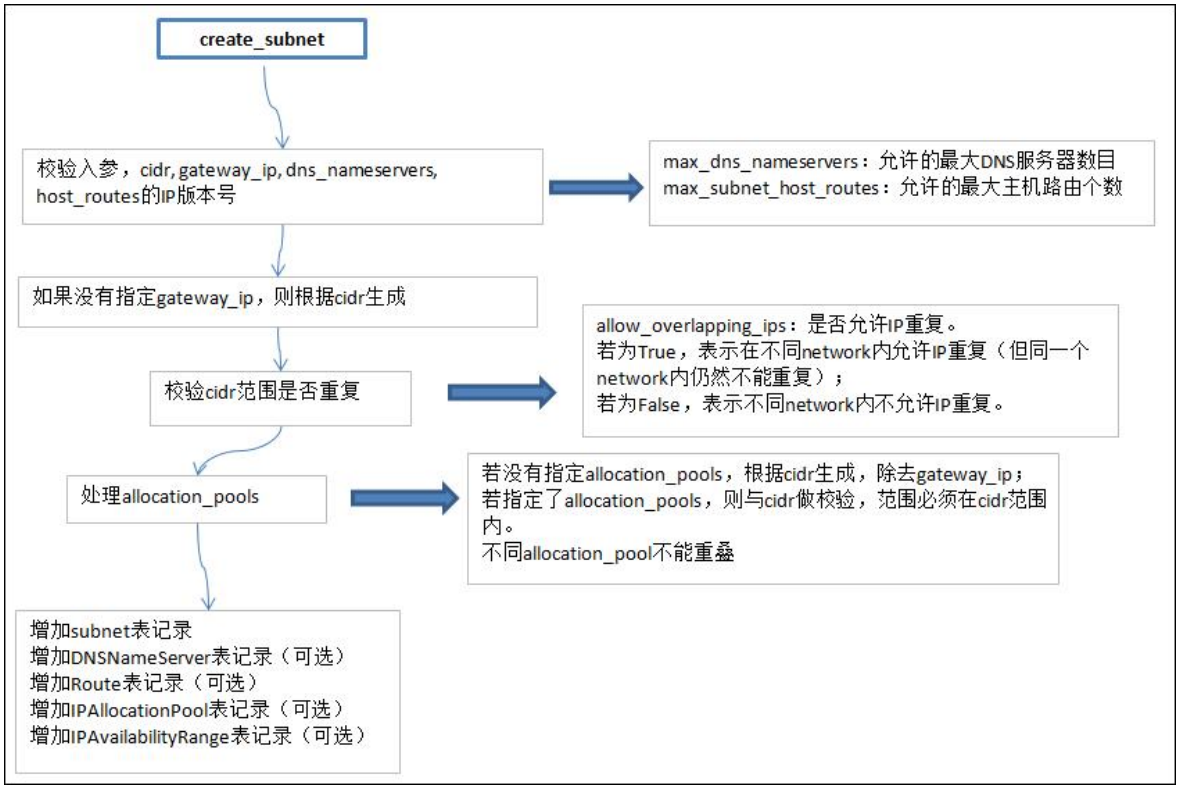
2.2.2.1 流程

2.2.3 更新 Network



2.3 关于 Subnet 的操作（使用 Linux Bridge）

2.3.1 创建 Subnet



2.3.2 删除 Subnet

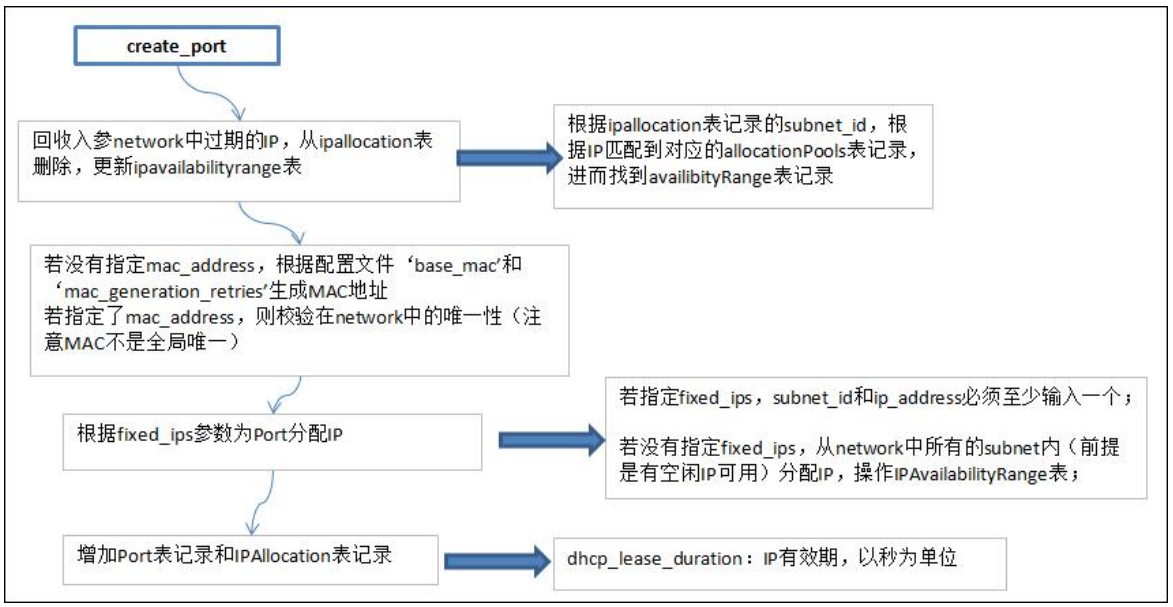
操作数据库，删除相关表记录。但删除的前提是，subnet 内 ipallocation 对应的 port 的 device_owner 属性是'network:dhcp'或者 'network:router_interface'，或者 ipallocation 没有对应的 port。

2.3.3 更新 Subnet

根据入参更新 DNSNameServer 表和 Route 表，然后更新 Subnet 表。

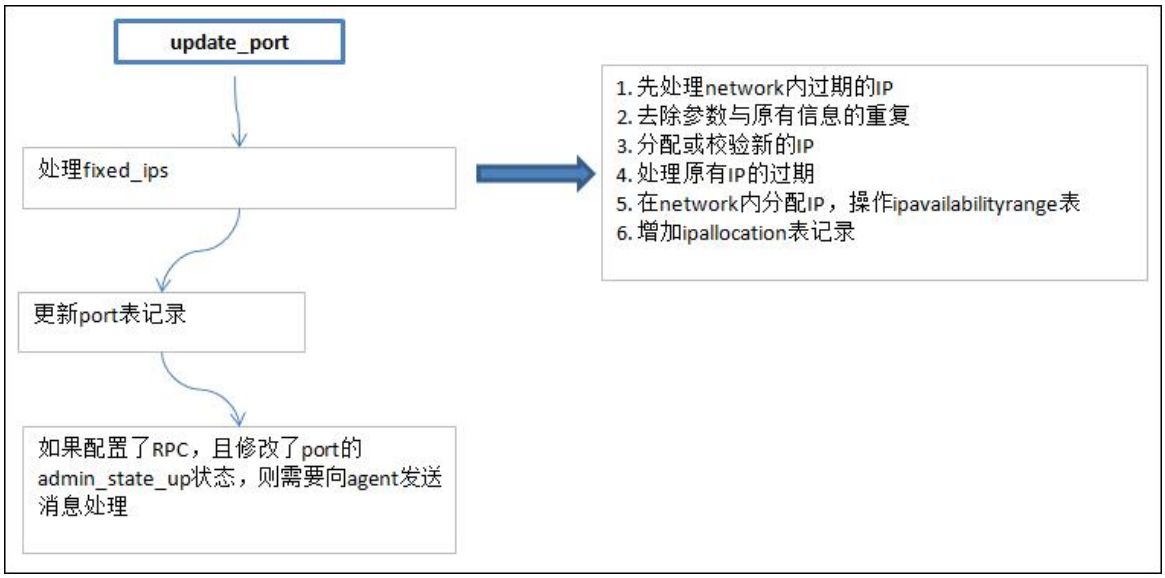
2.4 关于端口的操作（使用 Linux Bridge）

2.4.1 创建 Port

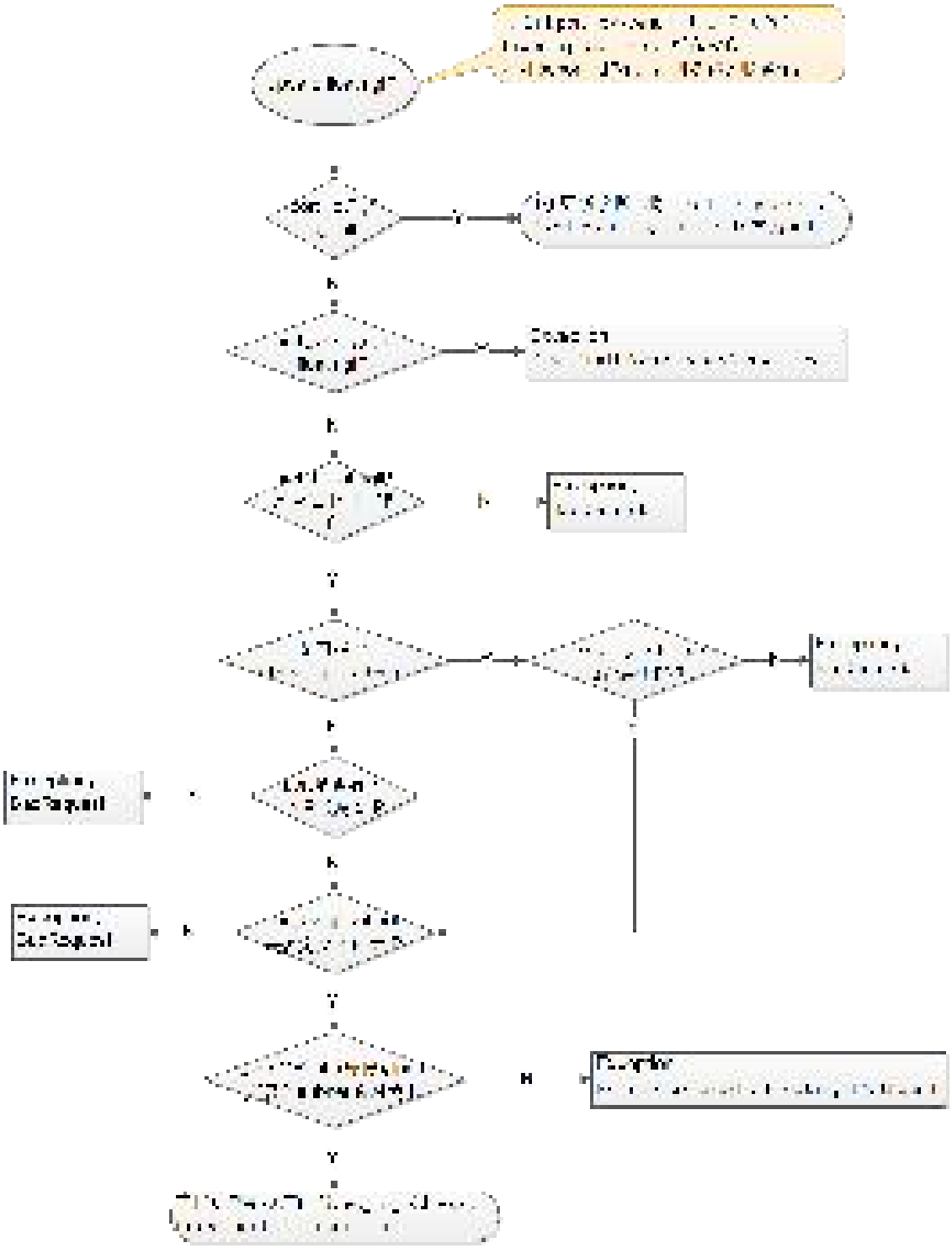


2.4.2 删除 Port

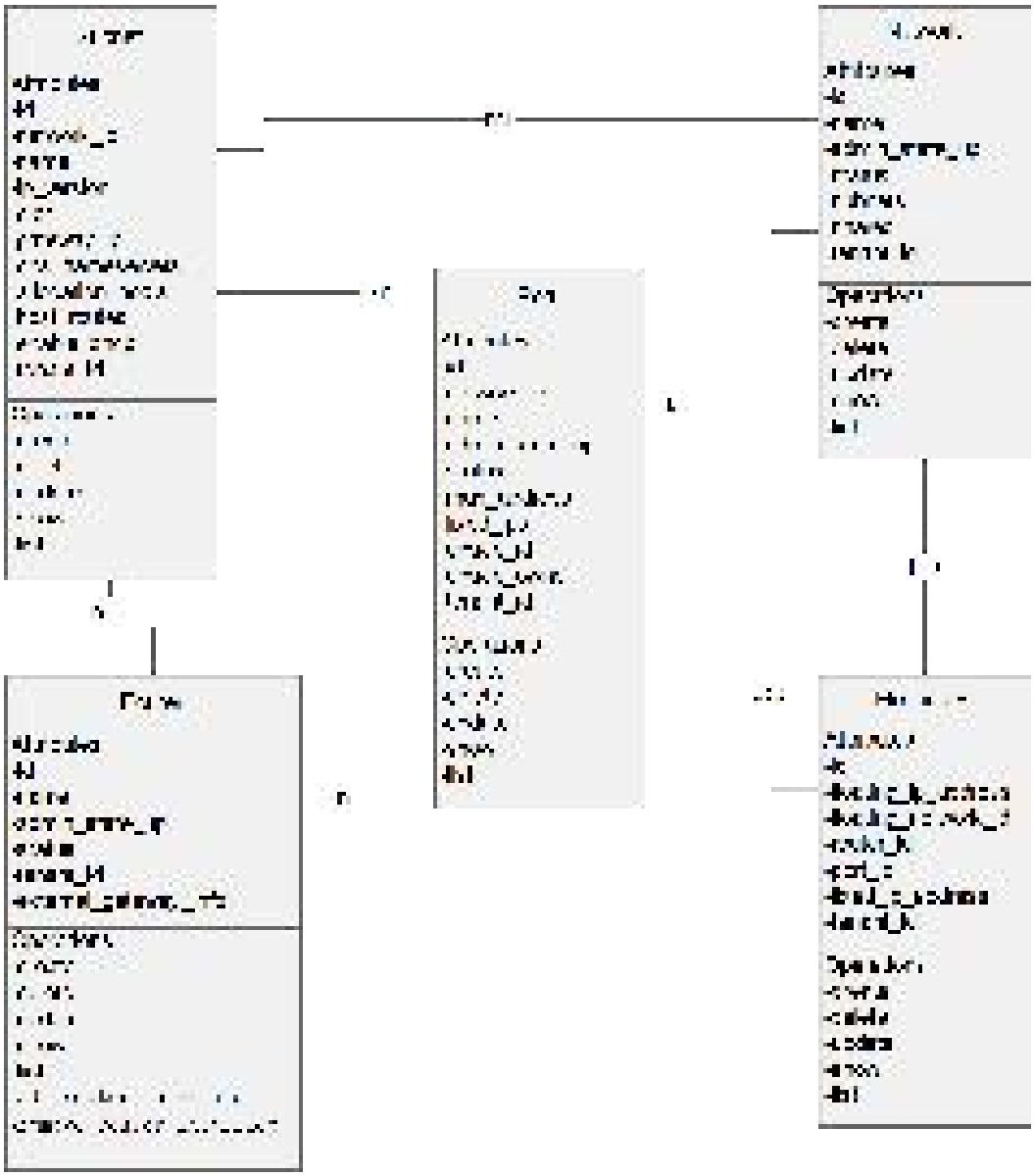
2.4.3 更新 Port



2.5 update_floatingIP 流程



2.6 对象模型



2.7

3. Quantum-LinuxBridge 插件-agent 的工作原理

3.1 初始化，参见文件
quantum/plugins/linuxbridge/agent/linuxbridge_quantum_agent.py

3.1.1 主函数

```
#============== ============主函数

def main():

eventlet.monkey_patch()

cfg.CONF(args=sys.argv, project='quantum')

# (TODO) gary - swap with common logging

logging_config.setup_logging(cfg.CONF)
```

```
interface_mappings = {}
# agent 使用的配置项
for mapping in cfg.CONF.LINUX_BRIDGE.physical_interface_mappings:
try:
physical_network, physical_interface = mapping.split(':')
# 更新字典信息，物理网络和物理接口的对应关系
interface_mappings[physical_network] = physical_interface
LOG.debug("physical network %s mapped to physical interface %s" %
(physical_network, physical_interface))
except ValueError as ex:
LOG.error("Invalid physical interface mapping: \'%s\' - %s" %
(mapping, ex))
sys.exit(1)
polling_interval = cfg.CONF.AGENT.polling_interval
reconnect_interval = cfg.CONF.DATABASE.reconnect_interval
root_helper = cfg.CONF.AGENT.root_helper
rpc = cfg.CONF.AGENT.rpc
# 如果在 AGENT 段配置了 rpc
if rpc:
plugin = LinuxBridgeQuantumAgentRPC(interface_mappings,
polling_interval,
root_helper)
# 如果没有配置
else:
db_connection_url = cfg.CONF.DATABASE.sql_connection
plugin = LinuxBridgeQuantumAgentDB(interface_mappings,
polling_interval,
reconnect_interval,
root_helper,
db_connection_url)
LOG.info("Agent initialized successfully, now running... ")
# 主函数，内部是个循环
plugin.daemon_loop()
sys.exit(0)
```

3.1.2 使用的 RPC 类（接收和发送消息）

#============= ============使用 RPC 的类

class **LinuxBridgeQuantumAgentRPC**:

def **__init__**(*self*, interface_mappings, polling_interval,

root_helper):

self.polling_interval = polling_interval

self.root_helper = root_helper

创建 LinuxBridge 对象

self.setup_linux_bridge(interface_mappings)

self.setup_rpc(interface_mappings.values())

def **setup_rpc**(*self*, physical_interfaces):

if physical_interfaces:

获取网络接口的 MAC 地址

mac = utils.get_interface_mac(physical_interfaces[0])

else:

devices = ip_lib.IPWrapper(*self*.root_helper).get_devices(True)

if devices:

mac = utils.get_interface_mac(devices[0].name)

else:

LOG.error("Unable to obtain MAC of any device for agent_id")

exit(1)

agent_id 在每一个计算节点上应该是不同的

self.agent_id = '%s%s' % ('lb', (mac.replace(":", "")))

LOG.info("RPC agent_id: %s" % *self*.agent_id)

topic='q-agent-notifier'

self.topic = topics.AGENT

创建一个 RPC 发送端，routing-key='q-plugin'

self.plugin_rpc = agent_rpc.PluginApi(topics.PLUGIN)

RPC network init

self.context = context.RequestContext('quantum', 'quantum',

is_admin=False)

Handle updates from service

self.callbacks = LinuxBridgeRpcCallbacks(*self*.context,

self.linux_br)

self.dispatcher = *self*.callbacks.create_rpc_dispatcher()

Define the listening consumers for the agent

创建两个队列监听者

routing-key 分别是'q-agent-notifier-network-delete'和'q-agent-notifier-port-update'

```
consumers = [[topics.PORT, topics.UPDATE],
[topics.NETWORK, topics.DELETE]]
self.connection = agent_rpc.create_consumers(self.dispatcher,
self.topic,
consumers)
# pyudev 库，驱动和硬件设备管理
self.udev = pyudev.Context()
# pyudev.Monitor 是一个同步的设备事件监听器, connecting to the kernel daemon through netlink
monitor = pyudev.Monitor.from_netlink(self.udev)
# 过滤事件
monitor.filter_by('net')
```

3.1.3 循环处理

```
#============== ============agent 的循环处理方法
def daemon_loop(self):
sync = True
devices = set()
LOG.info("LinuxBridge Agent RPC Daemon Started!")
while True:
start = time.time()
if sync:
LOG.info("Agent out of sync with plugin!")
devices.clear()
sync = False
# 使用 pyudev 库获取本机网络设备实时信息，如果与上次保存信息相同直接返回进行下一次循环
device_info = self.update_devices(devices)
# notify plugin about device deltas
if device_info:
LOG.debug("Agent loop has new devices!")
# If treat devices fails - indicates must resync with plugin
# 处理增加或删除设备信息的主函数 <=== 重要!
sync = self.process_network_devices(device_info)
devices = device_info['current']
# sleep till end of polling interval
elapsed = (time.time() - start)
if (elapsed < self.polling_interval):
```

```
time.sleep(self.polling_interval - elapsed)
else:
LOG.debug("Loop iteration exceeded interval (%s vs. %s)!",
self.polling_interval, elapsed)
```

以新增设备为例：

1. 首先向上层查询新增设备信息

2. 根据设备的 admin_state_up 状态调用 LinuxBridge 对象方法

```
physical_interface = self.interface_mappings.get(physical_network)
if not physical_interface:
LOG.error("No mapping for physical network %s" %
physical_network)
return False
# flat 模式
if int(vlan_id) == lconst.FLAT_VLAN_ID:
self.ensure_flat_bridge(network_id, physical_interface)
# vlan 模式
else:
# 为接口添加 vlan 并创建网桥（如果没有的话）
self.ensure_vlan_bridge(network_id, physical_interface,
vlan_id)
if utils.execute(['brctl', 'addif', bridge_name, tap_device_name],
root_helper=self.root_helper):
return False
LOG.debug("Done adding device %s to bridge %s" % (tap_device_name,
bridge_name))
return True
```

3. 如果是 vlan 类型网络

```
def ensure_vlan_bridge(self, network_id, physical_interface, vlan_id):
"""Create a vlan and bridge unless they already exist."""
interface = self.ensure_vlan(physical_interface, vlan_id)
bridge_name = self.get_bridge_name(network_id)
self.ensure_bridge(bridge_name, interface)
return interface
```

3.2 消息处理

在 LinuxBridge 插件的 agent 中只接收两种消息的处理，一种是删除 Network，一种是更新 Port。

3.2.1 network_delete

```
def network_delete(self, context, **kwargs):
LOG.debug("network_delete received")
network_id = kwargs.get('network_id')
# 获取逻辑网络对应的网桥名称，前缀"brq"
bridge_name = self.linux_br.get_bridge_name(network_id)
LOG.debug("Delete %s", bridge_name)
self.linux_br.delete_vlan_bridge(bridge_name)
def delete_vlan_bridge(self, bridge_name):
# 调用命令查询网桥是否存在
if self.device_exists(bridge_name):
# 获取网桥上的所有接口（列举目录/sys/devices/virtual/net/${bridge_name}/brif/）
interfaces_on_bridge = self.get_interfaces_on_bridge(bridge_name)
for interface in interfaces_on_bridge:
# 通过命令删除接口
self.remove_interface(bridge_name, interface)
for physical_interface in self.interface_mappings.itervalues():
# 若是 flat 类型的网络
if physical_interface == interface:
# This is a flat network => return IP's from bridge to
# interface
ips, gateway = self.get_interface_details(bridge_name)
self.update_interface_ip_details(interface,
bridge_name,
ips, gateway)
else:
if interface.startswith(physical_interface):
# 调用命令删除接口上创建的 vlan
self.delete_vlan(interface)
LOG.debug("Deleting bridge %s" % bridge_name)
if utils.execute(['ip', 'link', 'set', bridge_name, 'down'],
root_helper=self.root_helper):
return
# 删除网桥
if utils.execute(['brctl', 'delbr', bridge_name],
root_helper=self.root_helper):
```

```
return
LOG.debug("Done deleting bridge %s" % bridge_name)
else:
LOG.error("Cannot delete bridge %s, does not exist" % bridge_name)
```

3.2.2 port_update

```
def port_update(self, context, **kwargs):
LOG.debug("port_update received")
port = kwargs.get('port')
# 若操作状态为 True，增加设备
if port['admin_state_up']:
vlan_id = kwargs.get('vlan_id')
# create the networking for the port
self.linux_br.add_interface(port['network_id'],
vlan_id,
port['id'])
# 删除设备
else:
bridge_name = self.linux_br.get_bridge_name(port['network_id'])
tap_device_name = self.linux_br.get_tap_device_name(port['id'])
self.linux_br.remove_interface(bridge_name, tap_device_name)
```

4. Linux Bridge 的分析

4.1 Linux 网桥的原理

- 4.1.1 中继器，网桥，交换机，网桥和交换机的比较等
- 4.1.2 主机
- 4.1.3 用网桥合并不同的局域网
- 4.1.4 地址学习

等等。。。

4.2 Spanning Tree 协议

4.3 Linux 中网桥的实现

- 4.3.1 网桥设备的抽象
- 4.3.2 重要的数据结构
- 4.3.3 网桥代码的初始化
- 4.3.4 创建网桥设备和网桥端口
- 4.3.5 创建和删除网桥
- 4.3.6 为网桥添加端口
- 4.3.7 转发，等等。。。

5. Linux Bridge Plugin 如何利用 Linux 网桥进行虚拟局域网的创建和管理

对该内容目前了解不深，有待于进一步的学习和研究

6. 常用的 Quantum API 的实现分析

   在对以上内容的分析基础上，结合 Quantum 组件的源码，写出若干主要 API 的实现原理。

7. 针对本项目搭建的模拟系统中的一个具体应用实例，给出 Quantum 组件在其中的具体作用的流程分析。