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
1.3.2 FlatDHCP
1.3.3 VLAN
1.3.4 Floating Ip
1.4 Quantum 的软件架构和 Plugin 的作用

2. Quantum 的基本工作过程

2.1 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。

2.1.1 创建 network

2.1.1.1 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：

2.1.1.2 LinuxBridge 插件的初始化

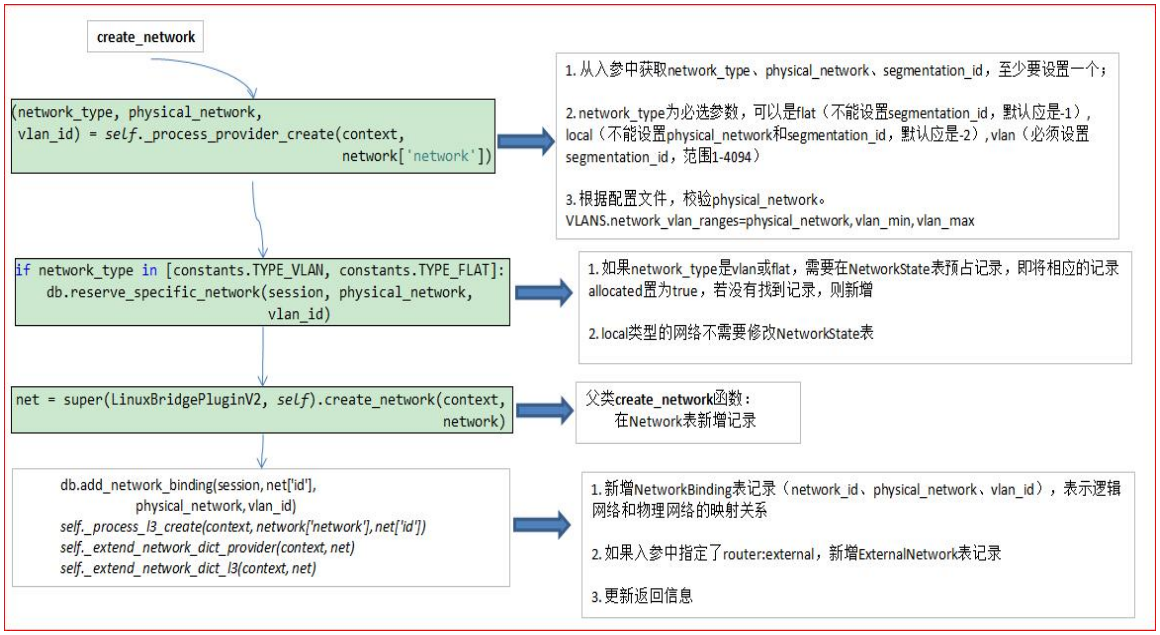
```
def __init__(self):
    # 连接数据库相关
    db.initialize()
    # 初始化network_vlan_ranges配置项，存到network_vlan_ranges变量
    self._parse_network_vlan_ranges()
    # 根据配置中physical_network和vlan信息更新networkstate表
    db.sync_network_states(self.network_vlan_ranges)
    self.tenant_network_type = cfg.CONF.VLANS.tenant_network_type
    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

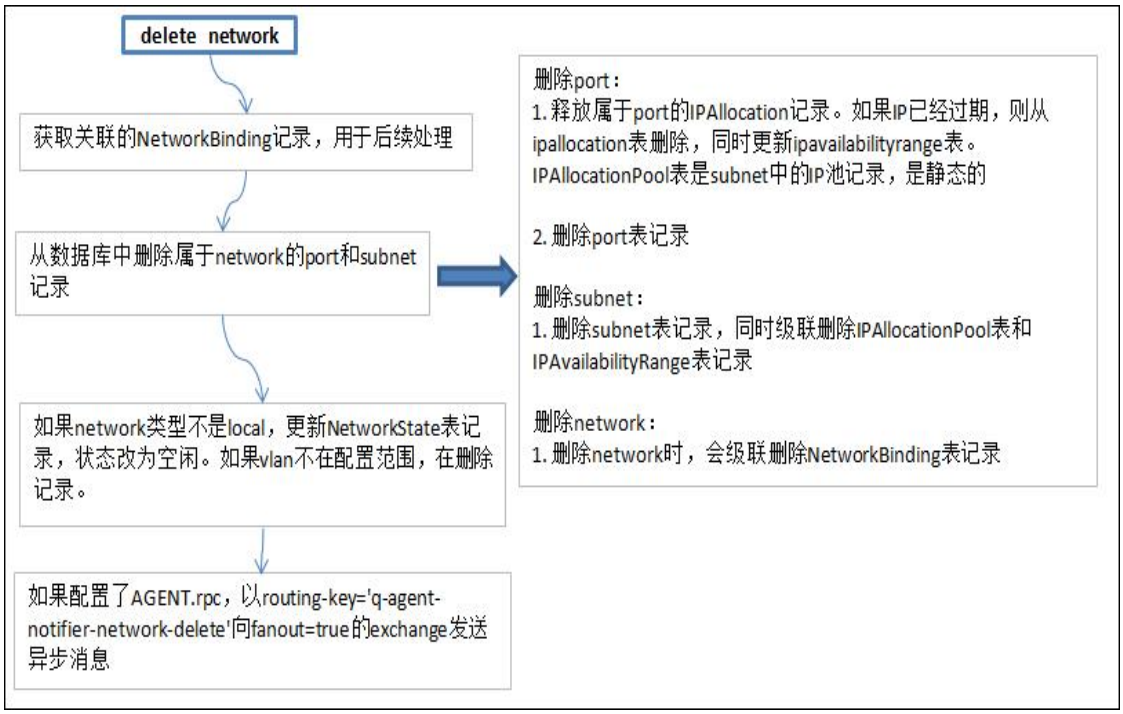
向agent发送消息，目前有两个消息：network_delete和port_update

2.1.1.3 流程

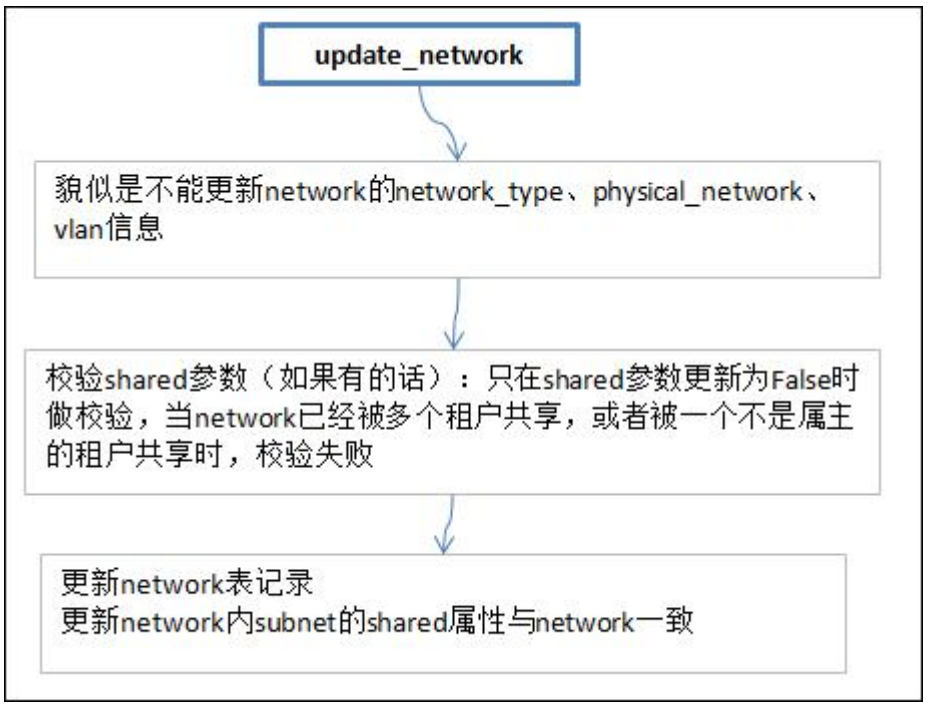


2.1.2 删除 Network

2.1.2.1 流程

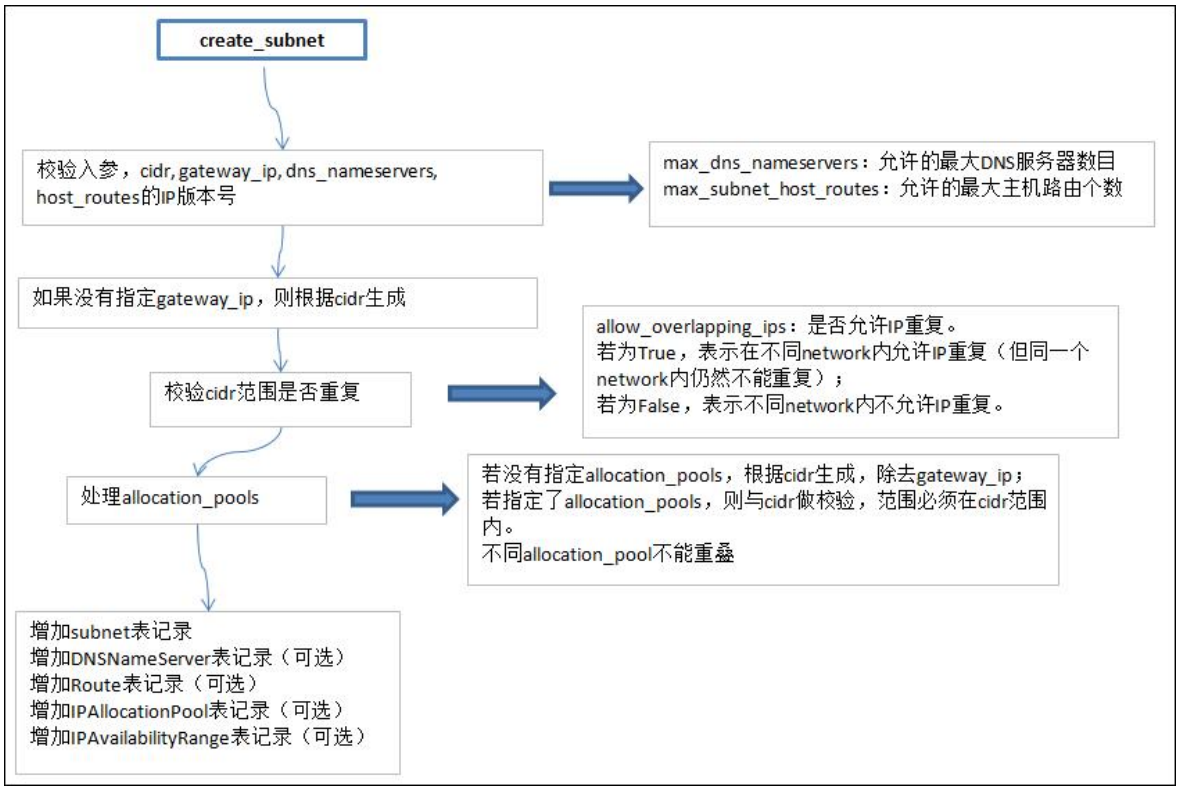


2.1.3 更新 Network



2.2 关于 Subnet 的操作（使用 Linux Bridge）

2.2.1 创建 Subnet



2.2.2 删除 Subnet

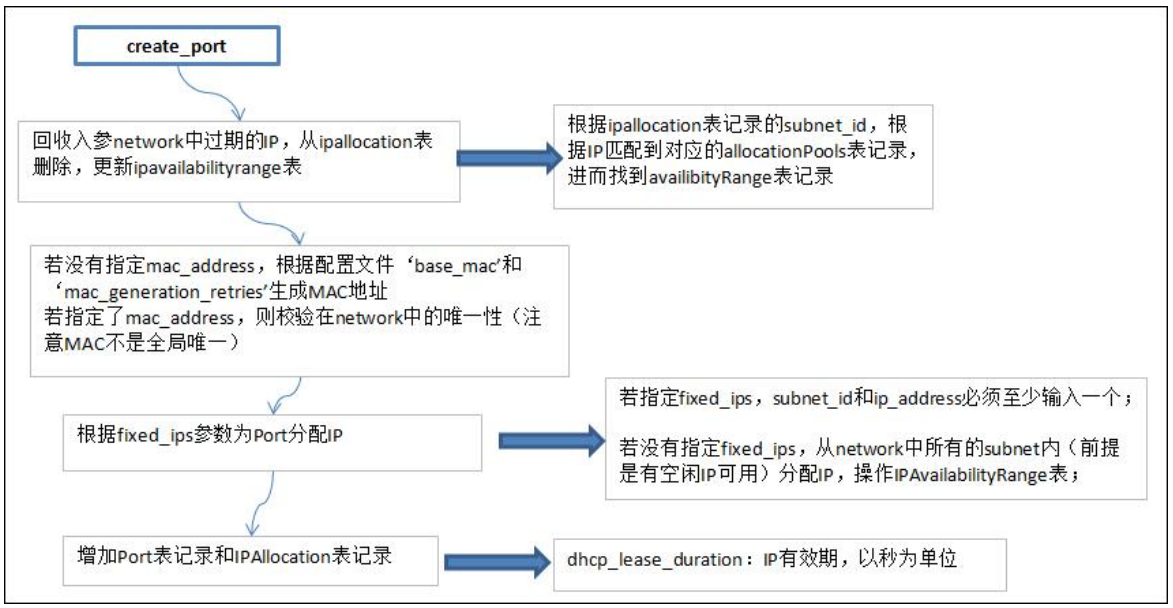
操作数据库，删除相关表记录。但删除的前提是，subnet 内 ipallocation 对应的 port 的 device_owner 属性是'network:dhcp'或者 'network:router_interface'，或者 ipallocation 没有对应的 port。

2.2.3 更新 Subnet

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

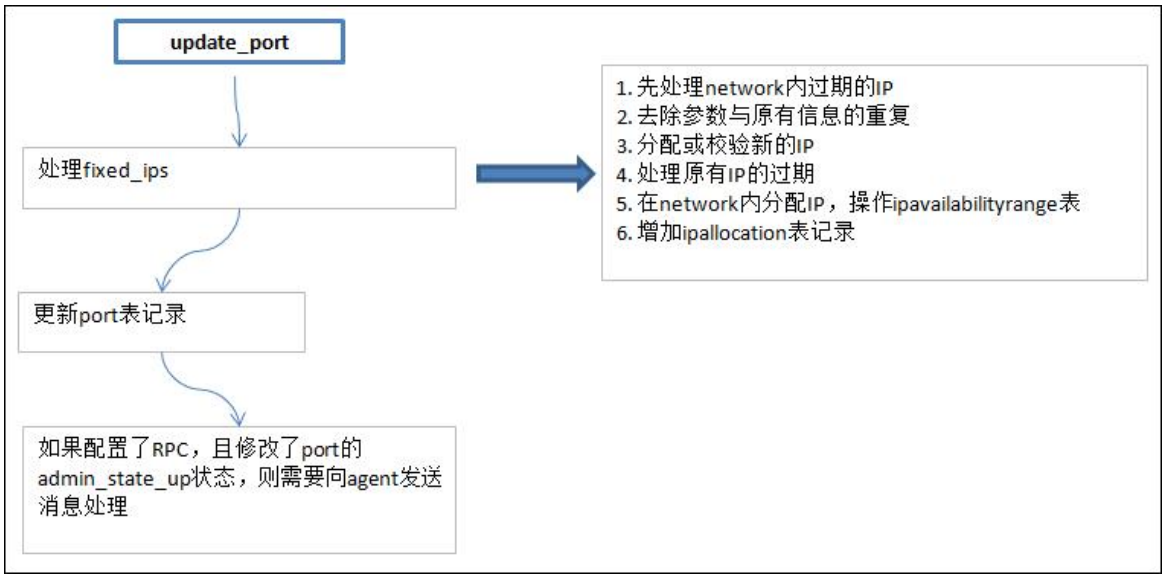
2.3 关于端口的操作（使用 Linux Bridge）

2.3.1 创建 Port



2.3.2 删除 Port

2.3.3 更新 Port



2.4 Quantum 消息处理流程

2.4.1 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
```

2.4.2 authtoken

调用 keystone 进行权限检查

2.4.3 keystonecontext

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
```

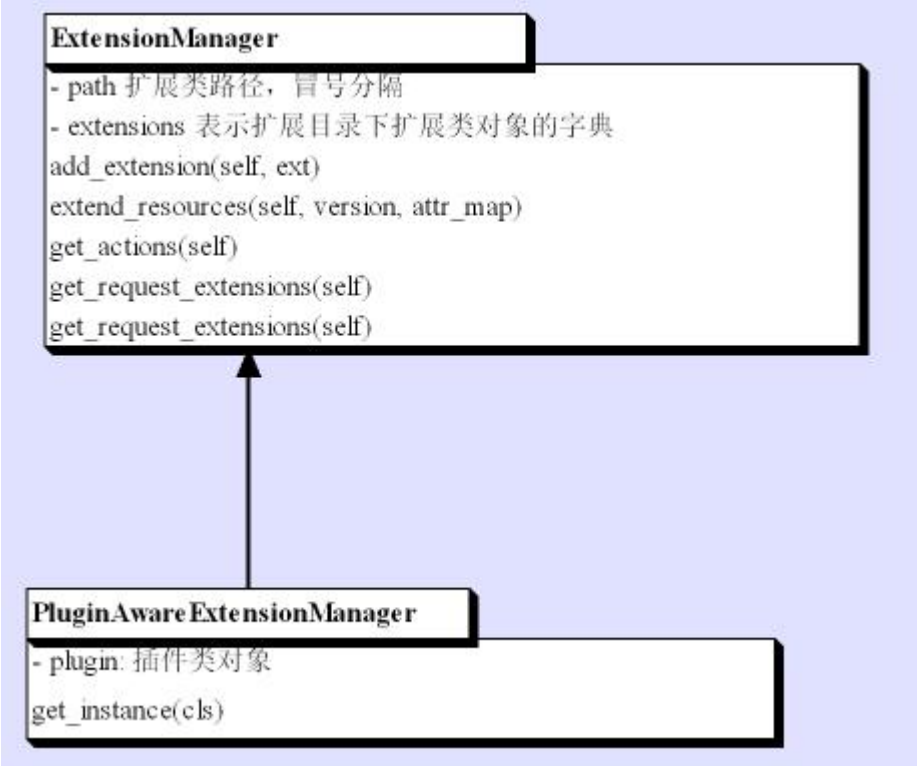
在环境上下文中增加quantum.context项

2.4.4 extensions

1) 获取 quantum.conf 中 core_plugin 配置的插件类
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
3) 加载 core_plugin 插件类
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：

```
def _load_all_extensions_from_path(self, path):
    for f in os.listdir(path):
        try:
            LOG.info(_('Loading extension file: %s'), f)
            mod_name, file_ext = os.path.splitext(os.path.split(f)[-1])
            ext_path = os.path.join(path, f)
            if file_ext.lower() == '.py' and not mod_name.startswith('_'):
                mod = imp.load_source(mod_name, ext_path)
                ext_name = mod_name[0].upper() + mod_name[1:]
                new_ext_class = getattr(mod, ext_name, None)
                if not new_ext_class:
                    LOG.warn(_('Did not find expected name '
                               '"%(ext_name)s" in %(file)s'),
                             {'ext_name': ext_name,
                              'file': ext_path})
                    continue
                new_ext = new_ext_class()
                self.add_extension(new_ext)
        except Exception as exception:
            LOG.warn("extension file %s wasnt loaded due to %s",
                     f, exception)
```

获取extensions目录下的所有文件并递归

先将文件名全路径分割为路径和文件全名，然后将文件全名分割为文件名和扩展名，如 quantum/extensions/l3.py，执行后，mod_name='l3', file_ext='.py'

过滤目录中文件名不以'_'开头且后缀为.py的文件加载文件

获取文件中的类，如l3.py中的L3类

将扩展对象加入extensions字典，字典项的键是扩展对象get_alias()方法返回的字符串，值是扩展对象

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。

Python 代码

```
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

2.4.5 quantumapiapp_v2_0
处理类：quantum/api/v2/router.py::APIRouter 类
初始化：
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) 同样使用 Routes，定义 URL 和应用接口的映射关系。
3) 在父类的初始化中,同样调用:
self._router = routes.middleware.RoutesMiddleware(self._dispatch, self.map)

3. 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。