OpenStack Neutron 代码分析

最新版:
https://github.com/yeasy/tech_writing/tree/master/OpenStack/OpenStack Neutron 代码分析.pdf

更新历史:
V0.6: 2014-07-11
　　完成对 api 部分的分析；
　　增加目录；
　　增加新的一章，集中从专题角度剖析代码。
V0.5: 2014-07-07
　　完成对 agent 部分的补充修订。
V0.4: 2014-05-19
　　完成对 OpenvSwitch plugin 的分析。
V0.3: 2014-05-12
　　完成对 IBM 的 SDN-VE plugin 的分析。
V0.2: 2014-05-06
　　完成配置文件（etc/）相关分析。
V0.1: 2014-04-14
　　完成代码基本结构。

目录

第 1 章　整体结构

源代码主要分为 5 个目录：

bin，doc，etc，neutron 和 tools。

1.1 bin

neutron-rootwrap

neutron-rootwrap-xen-dom0

提供利用 root 权限执行命令时候的操作接口，通过检查，可以配置不同用户利用管理员身份执行命令的权限。其主要实现是利用了 oslo.rootwrap 包中的 cmd 模块。

1.2 doc

可以利用 sphinx 来生成文档。

source 子目录：文档相关的代码。

Makefile：用户执行 make 命令。

pom.xml：

1.3 etc

跟服务和配置相关的文件，基本上该目录中内容在安装时会被复制到系统的/etc/目录下。

init.d/neutron-server：neutron-server 系统服务脚本，支持 start、stop、restart 和 status 操作。

neutron/:

plugins/: 各种厂商的 plugin 相关的配置文件（*.ini），其中被注释掉的行表明了（不指明情况下的）默认值。

rootwrap.d/: 一些 filters 文件，用来限定各个模块执行命令的权限。

各种 ini 和 conf 文件，包括 api-paste.ini、dhcp_agent.ini、l3_agent.ini、fwaas_driver.ini、lbaas_agent.ini、metadata_agent.ini、metering_agent.ini，以及 neutron.conf、policy.json、rootwrap.conf、services.conf 等。

基本上 neutron 相关的各个组件的配置信息都在这里了。

1.4 neutron

核心的代码实现都在这个目录下。

可以通过下面的命令来统计主要实现的核心代码量。

```
find neutron  -name "*.py" | xargs cat | wc -l
```

目前版本，约为 215k 行。

1.5 tools

一些相关的代码格式化检测、环境安装的脚本。

第 2 章　etc 目录

2.1 init.d/

neutron-server 是系统服务脚本，核心部分为

```
start)
    test "$ENABLED" = "true" || exit 0
    log_daemon_msg "Starting neutron server" "neutron-server"
    start-stop-daemon -Sbmv --pidfile $PIDFILE --chdir $DAEMON_DIR --exec $DAEMON --
$DAEMON_ARGS
    log_end_msg $?
    ;;
  stop)
    test "$ENABLED" = "true" || exit 0
    log_daemon_msg "Stopping neutron server" "neutron-server"
    start-stop-daemon --stop --oknodo --pidfile ${PIDFILE}
    log_end_msg $?
    ;;
  restart|force-reload)
    test "$ENABLED" = "true" || exit 1
    $0 stop
    sleep 1
    $0 start
    ;;
  status)
    test "$ENABLED" = "true" || exit 0
    status_of_proc -p $PIDFILE $DAEMON neutron-server && exit 0 || exit $?
    ;;
  *)
    log_action_msg "Usage: /etc/init.d/neutron-server {start|stop|restart|force-reload|status}"
    exit 1
    ;;
```

2.2 neutron/

2.2.1 plugins/

包括 bigswitch、brocade、cisco、……等多种插件的配置文件（ini 文件）。

2.2.2 rootwrap.d

包括一系列的 filter 文件。包括 debug.filters

rootwrap 是实现让非特权用户以 root 权限去运行某些命令。这些命令就在 filter 中指定。

以 neutron 用户为例，在/etc/sudoers.d/neutron 文件中有

```
neutron ALL = (root) NOPASSWD: SETENV: /usr/bin/neutron-rootwrap
```

使得 neutron 可以以 root 权限运行 neutron-rootwrap。

而在/etc/neutron/rootwarp.conf 中定义了

```
filters_path=/etc/neutron/rootwrap.d,/usr/share/neutron/rootwrap,/etc/quantum/rootwrap.d,/usr/share/quantum/rootwrap
```

这些目录中定义了命令的 filter，也就是说匹配这些 filter 中定义的命令就可以用 root 权限执行了。这些 filter 中命令的典型格式为

```
cmd-name: filter-name, raw-command, user, args
```

例如/usr/share/neutron/rootwrap/dhcp.filters 文件中的如下命令允许以 root 身份执行 ovs-vsctl 命令。

```
ovs-vsctl: CommandFilter, ovs-vsctl, root
```

需要注意/etc/neutron/rootwrap.d，/usr/neutron/nova/rootwrap 必须是 root 权限才能修改。

2.3 api-paste.ini

定义了 WSGI 应用和路由信息。利用 Paste 来实例化 Neutron 的 APIRouter 类，将资源（端口、网络、子网）映射到 URL 上，以及各个资源的控制器。

在 neutron-server 启动的时候，一般会指定参数--config-file neutron.conf --config-file xxx.ini。看 neutron/server/__init__.py 的代码: main()主程序中会调用 config.parse(sys.argv[1:])来读取这些配置文件中的信息。而 api-paste.ini 信息中定义了 neutron、neutronapi_v2_0、若干 filter 和两个 app。

```
[composite:neutron]
use = egg:Paste#urlmap
/: neutronversions
/v2.0: neutronapi_v2_0

[composite:neutronapi_v2_0]
use = call:neutron.auth:pipeline_factory
noauth = request_id catch_errors extensions neutronapiapp_v2_0
keystone = request_id catch_errors authtoken keystonecontext extensions neutronapiapp_v2_0

[filter:request_id]
paste.filter_factory =
neutron.openstack.common.middleware.request_id:RequestIdMiddleware.factory

[filter:catch_errors]
paste.filter_factory =
neutron.openstack.common.middleware.catch_errors:CatchErrorsMiddleware.factory

[filter:keystonecontext]
paste.filter_factory = neutron.auth:NeutronKeystoneContext.factory

[filter:authtoken]
paste.filter_factory = keystoneclient.middleware.auth_token:filter_factory

[filter:extensions]
paste.filter_factory = neutron.api.extensions:plugin_aware_extension_middleware_factory

[app:neutronversions]
paste.app_factory = neutron.api.versions:Versions.factory

[app:neutronapiapp_v2_0]
paste.app_factory = neutron.api.v2.router:APIRouter.factory
```

neutron-server 在读取完配置信息后，会执行 neutron/common/config.py:load_paste_app("neutron")，即将 neutron 应用 load 进来。从 api-paste.ini 中可以看到，neutron 实际上是一个 composite，分别将 URL "/" 和 "/v2.0" 映射到 neutronversions 应用和 neutronapi_v2_0（也是一个 composite）。

前者实际上调用了 neutron.api.versions 模块中的 Versions.factory 来处理传入的请求。

后者则要复杂一些，首先调用 neutron.auth 模块中的 pipeline_factory 处理。如果是 noauth，则传入参数为 request_id，catch_errors，extensions 这些 filter 和 neutronapiapp_v2_0 应用；如果是 keystone，则多传入一个 authtoken filter，最后一个参数仍然是 neutronapiapp_v2_0 应用。来看 neutron.auth 模块中的 pipeline_factory 处理代码。

```
def pipeline_factory(loader, global_conf, **local_conf):
    """Create a paste pipeline based on the 'auth_strategy' config option."""
    pipeline = local_conf[cfg.CONF.auth_strategy]
    pipeline = pipeline.split()
    filters = [loader.get_filter(n) for n in pipeline[:-1]]
    app = loader.get_app(pipeline[-1])
    filters.reverse()
    for filter in filters:
        app = filter(app)
    return app
```

最终的代码入口是 neutron.api.v2.router:APIRouter.factory。该方法主要代码为

```
per = routes_mapper.Mapper()
        plugin = manager.NeutronManager.get_plugin()
        ext_mgr = extensions.PluginAwareExtensionManager.get_instance()
        ext_mgr.extend_resources("2.0", attributes.RESOURCE_ATTRIBUTE_MAP)

        col_kwargs = dict(collection_actions=COLLECTION_ACTIONS,
                          member_actions=MEMBER_ACTIONS)

        def _map_resource(collection, resource, params, parent=None):
            allow_bulk = cfg.CONF.allow_bulk
            allow_pagination = cfg.CONF.allow_pagination
            allow_sorting = cfg.CONF.allow_sorting
            controller = base.create_resource(
                collection, resource, plugin, params, allow_bulk=allow_bulk,
                parent=parent, allow_pagination=allow_pagination,
                allow_sorting=allow_sorting)
            path_prefix = None
            if parent:
                path_prefix = "/%s/{%s_id}/%s" % (parent['collection_name'],
                                                  parent['member_name'],
                                                  collection)
            mapper_kwargs = dict(controller=controller,
                                 requirements=REQUIREMENTS,
                                 path_prefix=path_prefix,
                                 **col_kwargs)
            return mapper.collection(collection, resource,
                                     **mapper_kwargs)

        mapper.connect('index', '/', controller=Index(RESOURCES))
        for resource in RESOURCES:
            _map_resource(RESOURCES[resource], resource,
                          attributes.RESOURCE_ATTRIBUTE_MAP.get(
                              RESOURCES[resource], dict()))

        for resource in SUB_RESOURCES:
            _map_resource(SUB_RESOURCES[resource]['collection_name'], resource,
                          attributes.RESOURCE_ATTRIBUTE_MAP.get(
                              SUB_RESOURCES[resource]['collection_name'],
                              dict()),
                          SUB_RESOURCES[resource]['parent'])

        super(APIRouter, self).__init__(mapper)
```

neutron server 启动后，根据配置文件动态加载对应的 core plugin 和 service plugin。neturon server 中会对收到的 rest api 请求进行解析，并最终转换成对该 plugin(core or service)中相应方法的调用。

2.4 dhcp_agent.ini

dhcp agent 相关的配置信息。包括与 neutron 的同步状态的频率、超时、驱动信息等。

2.5 fwaas_driver.ini

配置 fwaas 的 driver 信息，默认为

```
[fwaas]
#driver = neutron.services.firewall.drivers.linux.iptables_fwaas.IptablesFwaasDriver
#enabled = True
```

2.6 l3_agent.ini

L3 agent 相关的配置信息。

当存在外部网桥的时候，每个 agent 最多只能关联到一个外部网络。

2.7 lbaas_agent.ini

配置 LBaaS agent 的相关信息，包括跟 Neutron 定期同步状态的频率等。

2.8 metadata_agent.ini

metadata agent 的配置信息，包括访问 Neutron API 的用户信息等。

2.9 metering_agent.ini

metering agent 的配置信息，包括 metering 的频率、driver 等。

2.10 vpn_agent.ini

配置 vpn agent 的参数，vpn agent 是从 L3 agent 继承来的，也可以在 L3 agent 中对相应参数进行配置。

2.11 neutron.conf

neutron-server 启动后读取的配置信息。

2.12 policy.json

配置策略。

每次进行 API 调用时，会采取对应的检查，policy.json 文件发生更新后会立即生效。

目前支持的策略有三种：rule、role 或者 generic。

其中 rule 后面会跟一个文件名，例如

```
"get_floatingip": "rule:admin_or_owner",
```

其策略为 rule:admin_or_owner，表明要从文件中读取具体策略内容。

role 策略后面会跟一个 role 名称，表明只有指定 role 才可以执行。

generic 策略则根据参数来进行比较。

2.13 rootwrap.conf

neutron-rootwrap 的配置文件。

给定了一系列的 filter 文件路径和可执行文件路径，以及 log 信息。

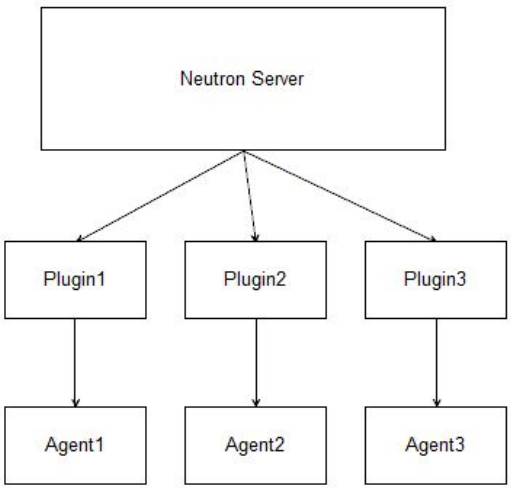
2.14 services.conf

配置一些特殊的 service 信息。

第 3 章　neutron 目录

neutron 从设计理念上来看，可以分为 neutron-server 相关（含各种 plugin）和 neutron-agent 相关两大部分。

其中 neutron-server 维护 high-level 的抽象网络管理，并通过不同产品的 plugin（这些 plugin 需要实现 neutron 定义的一系列操作网络的 API）转化为各自 agent 能理解的指令，agent 具体执行指令。简单的说，neutron-server 是做决策的，各种 neutron-agent 是实际干活的，plugin 是上下沟通的。如图表 1 所示。在这种结构中，同一时间只能有一套 plugin--agent 机制发生作用。



图表 1 neutron 中组件的逻辑关系

目前，ML2 子项目希望统一 plugin 对上接口，通过提供不同的驱动，来沟通不同产品的实现机制。

3.1 agent

在 neutron 的架构中，各种 agent 运行在计算节点和网络节点上，接收来自 neutron-server 的 plugin 的指令，对所管理的网桥进行实际的操作，属于“直接干活”的部分。plugin 和 agent 之间进行双向交互，一般的，每个 plugin 会创建一个 RPC server 来监听 agent 的请求。

agent 可以大致分为 core agent、dhcp、l3 和其它（metadata 等）。

本部分代码实现各种 agent 所需要的操作接口和库函数。

3.1.1 common/

主要包括 config.py，其中定义了 agent 的一些配置的关键字和默认值，和一些注册配置的函数。

3.1.2 linux/

主要包括跟 linux 环境相关的一些函数实现，为各种 agent 调用系统命令进行包装，例如对 iptables 操作，ovs 操作等等。

3.1.2.1 async_process.py

实现了 AsyncProcess 类，对异步进程进行管理。

3.1.2.2 daemon.py

实现一个通用的 Daemon 基类。一个 daemon 意味着一个后台进程，可以通过对应的 pid 文件对其进行跟踪。

3.1.2.3 dhcp.py

实现了 Dnsmasq 类、DhcpBase 类、DhcpLocalProcess 类、DeviceManager 类、DicModel 类、NetModel 类。对 linux 环境下 dhcp 相关的分配和维护实现进行管理。

通过调用 dnsmasq 工具来管理 dhcp 的分配。

3.1.2.4 external_process.py

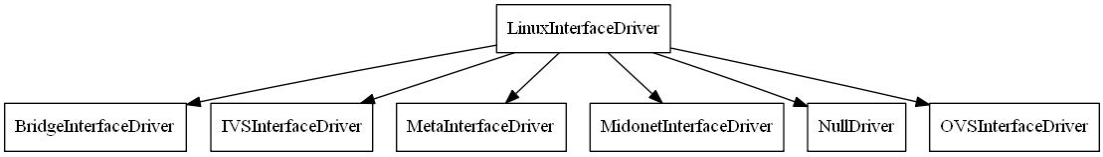
定义了 ProcessManager 类，对 neutron 孵化出的进程进行管理（可以通过跟踪 pid 文件进行激活和禁用等）。

3.1.2.5 interface.py

提供对网桥上的接口进行管理的一系列驱动。

定义了常见的配置信息，包括网桥名称，用户和密码等。

定义了几个不同类型网桥的接口驱动类，包括 LinuxInterfaceDriver 元类和由它派生出来的 MetaInterfaceDriver、BridgeInterfaceDriver、IVSInterfaceDriver、MidonetInterfaceDriver、NullDriver 和 OVSInterfaceDriver 等。



其中 LinuxInterfaceDriver 元类定义了 plug()和 unplug()两个抽象方法，需要继承类自己来实现。init_l3()方法则提供对接口进行 IP 地址的配置。

3.1.2.6 ip_lib.py

对 ip 相关的命令进行封装，包括一些操作类。例如 IpAddrCommand、IpLinkCommand、IpNetnsCommand、IpNeighCommand、IpRouteCommand、IpRule 等。基本上需要对 linux 上 ip 相关的命令进行操作都可以通过这个库提供的接口进行。

3.1.2.7 iptables_firewall.py

利用 iptables 的规则实现的防火墙驱动，主要包括两个防火墙驱动类。

IptablesFirewallDriver，继承自 firewall.FirewallDriver，默认通过 iptables 规则启用了 security group 功能，包括添加一条 sg-chain 链，为每个端口添加两条链（physdev-out 和 physdev-in）。

OVSHybridIptablesFirewallDriver，继承自 IptablesFirewallDriver，基本代码没动，修改了两个获取名字函数的实现：_port_chain_name()和_get_deice_name()。

3.1.2.8 iptables_manager.py

对 iptables 规则、表资源进行封装，提供操作接口。

定义了 IptablesManager 类、IptablesRule 类、IptablesTable 类。

其中 IptablesManager 对 iptables 工具进行包装。首先，创建 neutron-filter-top 链，加载到 FORWARD 和 OUTPUT 两条链开头。默认的 INPUT、OUTPUT、FORWARD 链会被包装起来，即通过原始的链跳转到一个包装后的链。此外，neutron-filter-top 链中有一条规则可以跳转到一条包装后的 local 链。

3.1.2.9 ovs_lib.py

提供对 OVS 网桥的操作支持，包括一个 VifPort，BaseOVS 类和继承自它的 OVSBridge 类。提供对网桥、端口等资源的添加、删除，执行 ovs-vsctl 命令等。

3.1.2.10 ovsdb_monitor.py

提供对 ovsdb 的监视器。包括一个 OvsdbMonitor 类（继承自 neutron.agent.linux.async_process.AsyncProcess）和 SimpleInterfaceMonitor 类（继承自前者）。

3.1.2.11 polling.py

监视 ovsdb 来决定何时进行 polling。包括一个 BasePollingManager 和继承自它的 InterfacePollingMinimizer 类等。

3.1.2.12 utils.py

一些辅助函数，包括 create_process 通过创建一个进程来执行命令、get_interface_mac、replace_file 等。

3.1.3 metadata/

3.1.3.1 agent.py

主要包括 MetadataProxyHandler、UnixDomainHttpProtocol、WorkerService、UnixDomainWSGIServer、UnixDomainMetadataProxy 几个类和一个 main 函数。

该文件的主逻辑代码为：

```
cfg.CONF.register_opts(UnixDomainMetadataProxy.OPTS)
    cfg.CONF.register_opts(MetadataProxyHandler.OPTS)
    cache.register_oslo_configs(cfg.CONF)
    cfg.CONF.set_default(name='cache_url', default='memory://?default_ttl=5')
    agent_conf.register_agent_state_opts_helper(cfg.CONF)
    config.init(sys.argv[1:])
    config.setup_logging(cfg.CONF)
    utils.log_opt_values(LOG)
    proxy = UnixDomainMetadataProxy(cfg.CONF)
    proxy.run()
```

在读取相关配置完成后，则实例化一个 UnixDomainMetadataProxy，并调用其 run 函数。run 函数则进一步创建一个 server = UnixDomainWSGIServer('neutron-metadata-agent')对象，并调用其 start()和 wait()函数。

run 函数会将应用绑定到 MetadataProxyHandler()类，该类包括一个__call__函数，调用_proxy_request()对传入的 HTTP 请求进行处理。

3.1.3.2 namespace_proxy.py

定义了 UnixDomainHTTPConnection、NetworkMetadataProxyHandler、ProxyDaemon 三个类和主函数。主函数代码为

```
    eventlet.monkey_patch()
    opts = [
        cfg.StrOpt('network_id',
                   help=_('Network that will have instance metadata '
                          'proxied.')),
        cfg.StrOpt('router_id',
                   help=_('Router that will have connected instances\' '
                          'metadata proxied.')),
        cfg.StrOpt('pid_file',
                   help=_('Location of pid file of this process.')),
        cfg.BoolOpt('daemonize',
                    default=True,
                    help=_('Run as daemon.')),
        cfg.IntOpt('metadata_port',
                   default=9697,
                   help=_("TCP Port to listen for metadata server "
                          "requests.")),
        cfg.StrOpt('metadata_proxy_socket',
                   default='$state_path/metadata_proxy',
                   help=_('Location of Metadata Proxy UNIX domain '
                          'socket'))
    ]

    cfg.CONF.register_cli_opts(opts)
    # Don't get the default configuration file
    cfg.CONF(project='neutron', default_config_files=[])
    config.setup_logging(cfg.CONF)
    utils.log_opt_values(LOG)
    proxy = ProxyDaemon(cfg.CONF.pid_file,
                        cfg.CONF.metadata_port,
                        network_id=cfg.CONF.network_id,
                        router_id=cfg.CONF.router_id)

    if cfg.CONF.daemonize:
        proxy.start()
    else:
        proxy.run()
```

其基本过程也是读取完成相关的配置信息，然后启动一个 ProxyDaemon 实例，以 daemon 或 run 方法来运行。run 方法则创建一个 wsgi 服务器，然后运行。最终绑定的应用为 Network MetadataProxyHandler。

```
proxy = wsgi.Server('neutron-network-metadata-proxy')
proxy.start(handler, self.port)
proxy.wait()
```

3.1.4 dhcp_agent.py

dhcp 服务的 agent 端，负责实现 dhcp 的分配等。

主要包括 DhcpAgent()类、继承自它的 DhcpAgentWithStateReport 类和继承自 RpcProxy 的 DhcpPluginApi 类。

主函数为

```
def main():
    register_options()
    common_config.init(sys.argv[1:])
    config.setup_logging(cfg.CONF)
    server = neutron_service.Service.create(
        binary='neutron-dhcp-agent',
        topic=topics.DHCP_AGENT,
        report_interval=cfg.CONF.AGENT.report_interval,
        manager='neutron.agent.dhcp_agent.DhcpAgentWithStateReport')
    service.launch(server).wait()
```

读取和注册相关配置（包括 dhcpagent、interface_driver、use_namespace 等）。

然后创建一个 neutron_service。绑定的主题是 DHCP_AGENT，默认驱动是 Dnsmasq，默认的管理器是 DhcpAgentWithStateReport 类

然后启动这个 service。

dhcp agent 的任务包括：汇报状态、处理来自 plugin 的 RPC 调用 API、管理 dhcp 信息。

plugin 端的 rpc 调用方法（一般由 neutron.api.v2.base.py 发出通知）在 neutron.api.rpc.agentnoftifiers.DhcpAgentNotifyAPI()类中实现，其中发出 notification 消息，会调用 agent 中对应的方法，包括（其中点符号替换为下划线符号）

```
VALID_RESOURCES = ['network', 'subnet', 'port']
    VALID_METHOD_NAMES = ['network.create.end',
                          'network.update.end',
                          'network.delete.end',
                          'subnet.create.end',
                          'subnet.update.end',
                          'subnet.delete.end',
                          'port.create.end',
                          'port.update.end',
                          'port.delete.end']
```

3.1.4.1 DhcpAgent 类

继承自 manager.Manager 类。

manager.Manager 类继承自 n_rpc.RpcCallback 类和 periodic_task.PeriodicTasks 类，提供周期性运行任务的方法。

初始化方法会首先从配置中导入 driver 类信息，然后获取 admin 的上下文。之后创建一个 DhcpPluginApi 类作为向 plugin 发出 rpc 消息的 handler。

after_start()方法会调用 run()方法，执行将 neutron 中状态同步到本地和孵化一个新的协程来周期性同步状态。

```
def after_start(self):
        self.run()
        LOG.info(_("DHCP agent started"))

def run(self):
        """Activate the DHCP agent."""
        self.sync_state()
        self.periodic_resync()
```

其中 sync_state()会发出 rpc 消息给 plugin，获取最新的网络状态，然后更新本地信息，调用 dnsmasq 进程使之生效。该方法在启动后运行一次。

periodic_resync()方法则孵化一个协程来运行_periodic_resync_helper()方法，该函数是一个无限循环，它周期性的调用 sync_state()。

3.1.4.2 DhcpPluginApi 类

提供从 agent 往 plugin 一侧进行 rpc 调用的 api。

3.1.4.3 DhcpAgentWithStateReport 类

该类继承自 DhcpAgent，主要添加了状态汇报。

汇报状态主要是 DhcpAgentWithStateReport 初始化中指定了一个 agent_rpc.PluginReportStateAPI(topics.PLUGIN)类作为状态汇报 rpc 消息的处理 handler。

```
if report_interval:
            self.heartbeat = loopingcall.FixedIntervalLoopingCall(self._report_state)
            self.heartbeat.start(interval=report_interval)
```

这些代码会让_report_state()定期执行来汇报自身状态。

其中_report_state()方法主要代码为：

```
self.agent_state.get('configurations').update(
                self.cache.get_state())
            ctx = context.get_admin_context_without_session()
            self.state_rpc.report_state(ctx, self.agent_state, self.use_call)
```

3.1.5 firewall.py

提供 FirewallDriver 元类和继承自它的简单的防火墙驱动类 NoopFirewallDriver。

3.1.6 l2population_rpc.py

主要定义了 L2populationRpcCallBackMixin 元类。

3.1.7 l3_agent.py

提供 L3 层服务的 agent，包括 L3NATAgent 类、继承自它的 L3NATAgentWithStateReport 类（作为 manager）、继承自 n_rpc.RpcProxy 类的 L3PluginApi 类（作为 agent 调用 plugin 一侧的 api）和 RouterInfo 类。

主过程为

```
def main(manager='neutron.agent.l3_agent.L3NATAgentWithStateReport'):
    conf = cfg.CONF
    conf.register_opts(L3NATAgent.OPTS)
    config.register_interface_driver_opts_helper(conf)
    config.register_use_namespaces_opts_helper(conf)
    config.register_agent_state_opts_helper(conf)
    config.register_root_helper(conf)
    conf.register_opts(interface.OPTS)
    conf.register_opts(external_process.OPTS)
    common_config.init(sys.argv[1:])
    config.setup_logging(conf)
    server = neutron_service.Service.create(
        binary='neutron-l3-agent',
        topic=topics.L3_AGENT,
        report_interval=cfg.CONF.AGENT.report_interval,
        manager=manager)
    service.launch(server).wait()
```

也是标准的 service 流程，启动一个管理 neutron-l3-agent 执行程序的服务，该服务将监听 topic 为 topics.L3_AGENT 的 rpc 消息队列。

3.1.7.1 L3NATAgent 类

继承自 firewall_l3_agent.FWaaSL3AgentRpcCallback 和 manager.Manager 两个类。

前者是由于现在的 FWaaS 设计都是挂载到 router 上的，因此，在创建 router 的时候，需要把对应的 firewall 添加上。不得不说这是个十分不合理的临时方案。

而 Manager 作为一个进行 rpc 调用管理和执行周期性任务的基础类。

初始化中根据配置信息，导入 driver，获取 admin 的上下文，获取 L3PluginApi，然后定期执行 self._rpc_loop()方法。该方法根据数据库中的信息来同步本地的 router。

调用 self._process_routers()方法和 self._process_router_delete()方法，这两个方法会进一步对本地的 iptables 进行操作，完成 router 的添加或删除。

3.1.7.2 L3PluginApi 类

继承自 neutron.common.rpc.RpcProxy 类，是一个进行 rpc 调用的代理。

被 L3NATAgent 类来调用，负责向 L3 的 Plugin 发出 rpc 消息（主题为 topics.L3PLUGIN），这些消息到达 plugin，最终被 plugin 的父类 neutron.db.l3_rpc_base.L3RpcCallbackMixin 类中的对应方法来处理，这些方法进一步调用父类 neutron.db.l3_db.L3_NAT_db_mixin 类中的对应方法跟数据库进行交互。

目前定义了三个方法：get_external_network_id()通过 rpc 调用 external_network_id()来获取外部网络的 id；get_routers()通过 rpc 调用 sync_routers()来获取所有的 router 的信息；update_floatingip_statuses()通过 rpc 调用 update_floatingip_statuses()来更新 flaoting ip 的状态。

3.1.7.3 L3NATAgentWithStateReport 类

该类是 L3 agent 资源 service 的 manager，其继承自 L3NATAgent，并添加了 rpc.PluginReportStateAPI 类来进行周期性状态汇报。该类会以 topic.PLUGIN 向 rpc 队列中写入 report_state()方法，并携带 agent 的状态信息作为参数。这些消息会被各个 plugin 收到。

3.1.8 netns_cleanup_util.py

清理无用的网络名字空间。当 neutron 的 agent 非正常退出时可以通过该工具来清理环境。

主过程十分简单，第一步是获取可能的无用名字空间，第二步是 sleep 后清除这些名字空间。

```
candidates = [ns for ns in
              ip_lib.IPWrapper.get_namespaces(root_helper)
              if eligible_for_deletion(conf, ns, conf.force)]

  if candidates:
     eventlet.sleep(2)

     for namespace in candidates:
        destroy_namespace(conf, namespace, conf.force)
```

3.1.9 ovs_cleanup_util.py

清理无用的 ovs 网桥和端口。

3.1.10 rpc.py

定义了 create_consumer()方法，设置 agent 进行 RPC 时候的消费者。

定义了 PluginApi 类和 PluginReportStateAPI 类。两者都是继承自 rpc.RpcProxy 类。

前者代表 rpc API 在 agent 一侧部分。后者是 agent 汇报自身状态。

PluginApi 类包括四个方法：get_device_details()、tunnel_sync()、update_device_down()和 update_device_up()。

PluginReportStateAPI 类只提供一个方法：report_state，将 agent 获取的本地的状态信息发出去。

3.1.11 securitygroups_rpc.py

定义了 SecurityGroupAgentRpcApiMixin 类、SecurityGroupAgentRpcCallbackMixin 类、SecurityGroupAgentRpcMixin 和 SecurityGroupServerRpcApiMixin。

其中*RpcApi 类提供了在 agent 端的对 RPC 的支持。

3.2 api

提供 RestAPI 访问。

3.2.1 rpc

3.2.1.1 agentnotifiers

主要负责发出一些 rpc 的通知给 agent，包括三个文件：dhcp_rpc_agent_api.py、l3_rpc_agent_api.py、metering_rpc_agent_api.py。

分别实现向 dhcp、l3 或者 metering 的 agent 发出通知消息。

以 dhcp_rpc_agent_api.py 为例，定义了 DhcpAgentNotifyAPI 类，该类继承自 neutron.common.rpc.RpcProxy。

首先定义允许对 agent 操作的资源和方法。

```
VALID_RESOURCES = ['network', 'subnet', 'port']
   VALID_METHOD_NAMES = ['network.create.end',
                         'network.update.end',
                         'network.delete.end',
                         'subnet.create.end',
                         'subnet.update.end',
                         'subnet.delete.end',
                         'port.create.end',
                         'port.update.end',
                         'port.delete.end']
```

实现的方法包括 agent_updated()、network_added_to_agent()、network_removed_from_agent()，分别 cast 一条 rpc 消息到 dhcp agent，调用对应方法。

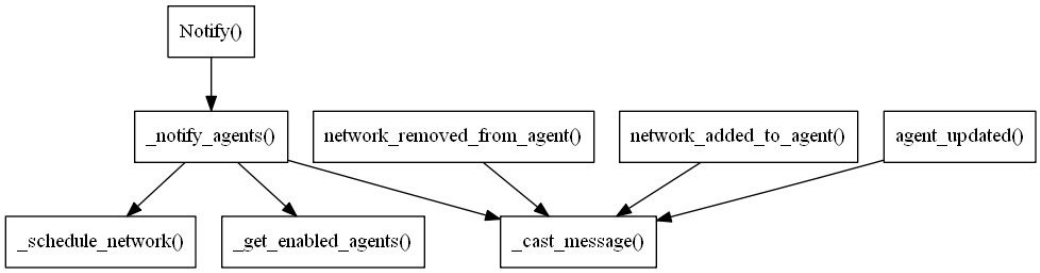
```
 def _cast_message(self, context, method, payload, host,
                   topic=topics.DHCP_AGENT):
       """Cast the payload to the dhcp agent running on the host."""
       self.cast(
          context, self.make_msg(method, payload=payload),topic='%s.%s' % (topic, host))

 def network_removed_from_agent(self, context, network_id, host):
       self._cast_message(context, 'network_delete_end',
                          {'network_id': network_id}, host)

 def network_added_to_agent(self, context, network_id, host):
       self._cast_message(context, 'network_create_end',
                          {'network': {'id': network_id}}, host)

 def agent_updated(self, context, admin_state_up, host):
       self._cast_message(context, 'agent_updated',
                          {'admin_state_up': admin_state_up}, host)
```

另外，实现 notify()方法，可以调用所允许的方法。

neutron 的 api 中会直接调用 notify()方法。

```
def notify(self, context, data, method_name):
    # data is {'key' : 'value'} with only one key
    if method_name not in self.VALID_METHOD_NAMES:
        return
    obj_type = data.keys()[0]
    if obj_type not in self.VALID_RESOURCES:
        return
    obj_value = data[obj_type]
    network_id = None
    if obj_type == 'network' and 'id' in obj_value:
        network_id = obj_value['id']
    elif obj_type in ['port', 'subnet'] and 'network_id' in obj_value:
        network_id = obj_value['network_id']
    if not network_id:
        return
    method_name = method_name.replace(".", "_")
    if method_name.endswith("_delete_end"):
        if 'id' in obj_value:
            self._notify_agents(context, method_name,
                                {obj_type + '_id': obj_value['id']},
                                network_id)
    else:
        self._notify_agents(context, method_name, data, network_id)
```

整体的主要方法调用结构如下图所示。



3.2.2 v2

实现 neutron api 第 2 个版本的定义。

主要方法包括 index、update、create、show 和 delete。

3.2.2.1 attributes.py

这里面定义了一系列的_validate_xxx 方法，包括_validate_mac_address、_validate_ip_address、_validate_boolean 等，对传入的参数进行格式检查。

3.2.2.2 base.py

定义了 Controller 类和 create_resource 方法。
后者根据传入参数声明一个 Controller 并用它初始化一个 wsgi 的资源。

```
def create_resource(collection, resource, plugin, params, allow_bulk=False,
                    member_actions=None, parent=None, allow_pagination=False,
                    allow_sorting=False):
    controller = Controller(plugin, collection, resource, params, allow_bulk,
                            member_actions=member_actions, parent=parent,
                            allow_pagination=allow_pagination,
                            allow_sorting=allow_sorting)

    return wsgi_resource.Resource(controller, FAULT_MAP)
```

Controller 类负责对 rest API 调用的资源进行处理。

成员包括一个对 dhcp agent 的 notifier。

3.2.2.3 resource.py

主要定义了 Request 类和 Resource 方法。

Request 类继承自 wsgi.Request，代表一个资源请求。

Resource 方法会根据传入的 Controller 构造一个 resource 对象。

3.2.2.4 resource_helper.py

包括 build_plural_mappings 和 build_resource_info 两个方法。

前者对所有的资源创建从其复数到单数形式的映射；后者为 advanced services 扩展创建 API 资源对象。

3.2.2.5 router.py

此处的 router 意味着是在 wsgi 框架下对请求的 rest api 进行调度的 router，并非网络中的 router。从外面 api-paste.ini 文件中，可以看到最终 app 指向的是

```
[app:neutronapiapp_v2_0]
paste.app_factory = neutron.api.v2.router:APIRouter.factory
```

该文件主要包括了 APIRouter 类，继承自 wsgi.Router 类，定义了 factory 方法。

其中 factory()方法返回一个该类的实体。

分析其初始化方法：

```
def __init__(self, **local_config):
    mapper = routes_mapper.Mapper()
    plugin = manager.NeutronManager.get_plugin()
    ext_mgr = extensions.PluginAwareExtensionManager.get_instance()
    ext_mgr.extend_resources("2.0", attributes.RESOURCE_ATTRIBUTE_MAP)

    col_kwargs = dict(collection_actions=COLLECTION_ACTIONS,
                      member_actions=MEMBER_ACTIONS)

    def _map_resource(collection, resource, params, parent=None):
        allow_bulk = cfg.CONF.allow_bulk
        allow_pagination = cfg.CONF.allow_pagination
        allow_sorting = cfg.CONF.allow_sorting
        controller = base.create_resource(
            collection, resource, plugin, params, allow_bulk=allow_bulk,
            parent=parent, allow_pagination=allow_pagination,
            allow_sorting=allow_sorting)
        path_prefix = None
        if parent:
            path_prefix = "/%s/{%s_id}/%s" % (parent['collection_name'],
                                              parent['member_name'],
                                              collection)
        mapper_kwargs = dict(controller=controller,
                             requirements=REQUIREMENTS,
                             path_prefix=path_prefix,
                             **col_kwargs)
        return mapper.collection(collection, resource,
                                 **mapper_kwargs)

    mapper.connect('index', '/', controller=Index(RESOURCES))
    for resource in RESOURCES:
        _map_resource(RESOURCES[resource], resource,
                      attributes.RESOURCE_ATTRIBUTE_MAP.get(
                          RESOURCES[resource], dict()))

    for resource in SUB_RESOURCES:
        _map_resource(SUB_RESOURCES[resource]['collection_name'], resource,
                      attributes.RESOURCE_ATTRIBUTE_MAP.get(
                          SUB_RESOURCES[resource]['collection_name'],
                          dict()),
                      SUB_RESOURCES[resource]['parent'])

    super(APIRouter, self).__init__(mapper)
```

首先初始化一个 router 的 mapper；之后获取 plugin，通过调用 NeutronManger 类（负责解析配置文件并读取其中的 plugin 信息）；然后获取支持的扩展的资源管理者，并把默认的对网络、子网和端口的资源的操作添加到扩展的资源管理者类中。

接下来，绑定资源的请求到各个资源上。

3.2.3 views

主要包括 versions.py，其中定义了 ViewBuilder 类和全局的 get_view_builder()方法，该方法返回一个 ViewBuilder 类的实例。

3.2.4 api_common.py

一些实现 api 通用的类和方法，包括。

类：PaginationHelper、PaginationEmulatedHelper、PaginationNativeHelper、NoPaginationHelper、SortingHelper、SortingEmulatedHelper、SortingNativeHelper、NoSortingHelper、NeutronController。

方法：get_filters、get_previous_link、get_next_link、get_limit_and_marker、list_args、get_sorts、get_page_reverse、get_pagination_links。

3.2.5 extensions.py

定义了实现 extension 的几个类。

ExtensionDescriptor 类定义了作为 extension 描述的基础类。

ActionExtensionController 类继承自 wsgi.Controller，负责对扩展行动的管理。

RequestExtensionController 类继承自 wsgi.Controller，负责对扩展 request 的管理。

ExtensionController 类继承自 wsgi.Controller，定义了 index、show、delete、create 等方法，对扩展进行管理。

ExtensionMiddleware 继承自 wsgi.Middleware，负责处理扩展的中间件。

ExtensionManager 类负责从配置文件中加载扩展。PluginAwareExtensionManager 类继承自 ExtensionManager，增加对 plugin 对 extension 支持情况的检查。

3.2.6 versions.py

当 rest api 请求是版本号时候，调用该模块中的类 Versions 进行处理。

绑定也是在 api-paste.ini 文件中。

```
/: neutronversions
[app:neutronversions]
paste.app_factory = neutron.api.versions:Versions.factory
```

调用的是 Versions 类中的 factory()方法，该方法返回一个类的实体。该类是一个 callable 对象，主要函数就是__call__()方法。

```
@webob.dec.wsgify(RequestClass=wsgi.Request)
    def __call__(self, req):
        """Respond to a request for all Neutron API versions."""
        version_objs = [
            {
                "id": "v2.0",
                "status": "CURRENT",
            },
        ]

        if req.path != '/':
            language = req.best_match_language()
            msg = _('Unknown API version specified')
            msg = gettextutils.translate(msg, language)
            return webob.exc.HTTPNotFound(explanation=msg)

        builder = versions_view.get_view_builder(req)
        versions = [builder.build(version) for version in version_objs]
        response = dict(versions=versions)
        metadata = {
            "application/xml": {
                "attributes": {
                    "version": ["status", "id"],
                    "link": ["rel", "href"],
                }
            }
        }

        content_type = req.best_match_content_type()
        body = (wsgi.Serializer(metadata=metadata).
                serialize(response, content_type))

        response = webob.Response()
        response.content_type = content_type
        response.body = body

        return response
```

3.3 cmd

usage_audit.py，目前还十分简单，只是检测存在哪些网络资源（包括网络、子网、端口、路由器和浮动 IP）。

3.4 common

3.4.1 config.py

对配置进行管理。

定义了默认的 core_opts，包括绑定的主机地址、端口、配置文件默认位置、策略文件位置、VIF 的起始 Mac 地址、DNS 数量、子网的主机路由限制、DHCP 释放时间、nova 的配置信息等。以及 core_cli_opts，包括状态文件的路径。

主要包括

load_paste_app(app_name)方法，通过默认的 paste config 文件来读取配置，生成并返回 WSGI 应用。

parse(args)方法，在启动 neutron-server 的时候解析所有的命令行参数，并检查通过命令行传入的 base_mac 参数是否合法。

setup_logging(conf)方法，配置 logging 模块的名称。

3.4.2 constants.py

定义一些常量，例如各种资源的 ACTIVE、BUILD、DOWN、ERROR 状态，DHCP 等网络协议端口号，VLAN TAG 范围等

3.4.3 exceptions.py

定义了各种情况下的异常类，包括 NetworkInUse、PolicyFileNotFound 等等。

3.4.4 ipv6_utils.py

目前主要定义了 get_ipv6_addr_by_EUI64(prefix, mac)方法，通过给定的 v4 地址和 mac 来获取 v6 地址。

3.4.5 log.py

基于 neutron.openstack.common 中的 log 模块。

主要是定义了 log 修饰，在执行方法时会添加类名，方法名，参数等信息进入 debug 日志。

3.4.6 rpc.py

定义了类 class PluginRpcDispatcher(dispatcher.RpcDispatcher)，重载了 dispatch()方法，将 RPC 的通用上下文转换为 Neutron 的上下文。

3.4.7 test_lib.py

定义了 test_config={}，用于各个 plugin 进行测试。

3.4.8 topics.py

管理消息队列传递过程中的 topic 信息。

```
NETWORK = 'network'
SUBNET = 'subnet'
PORT = 'port'
SECURITY_GROUP = 'security_group'
L2POPULATION = 'l2population'

CREATE = 'create'
DELETE = 'delete'
UPDATE = 'update'

AGENT = 'q-agent-notifier'
PLUGIN = 'q-plugin'
L3PLUGIN = 'q-l3-plugin'
DHCP = 'q-dhcp-notifer'
FIREWALL_PLUGIN = 'q-firewall-plugin'
METERING_PLUGIN = 'q-metering-plugin'
LOADBALANCER_PLUGIN = 'n-lbaas-plugin'

L3_AGENT = 'l3_agent'
DHCP_AGENT = 'dhcp_agent'
METERING_AGENT = 'metering_agent'
LOADBALANCER_AGENT = 'n-lbaas_agent'
```

3.4.9 utils.py

一些辅助函数，包括查找配置文件，封装的 subprocess_open，解析映射关系、获取主机名等等。

3.5 db

数据库相关操作的实现。

3.6 debug

测试功能。

commands.py

debug_agent.py

shell.py

3.7 extensions

对现有 neutron API 的扩展。某些 plugin 可能还支持额外的资源或操作，可以先以 extension 的方式使用。包括 vpnaas，l3，lbaas 等

3.8 locale

多语言支持。

3.9 notifiers

3.10 openstack

公共模块。

3.11 plugins

包括实现网络功能的各个插件。

3.11.1 bigswitch

3.11.2 brocade

3.11.3 cisco

3.11.4 common

3.11.5 embrane

3.11.6 hyperv

3.11.7 ibm

3.11.7.1 agent/

sdnve_neutron_agent.py，该文件主要实现一个在计算节点和网络节点上的 daemon，对本地的网桥进行实际操作。其主要过程代码为

```
def main():
    eventlet.monkey_patch()
    cfg.CONF.register_opts(ip_lib.OPTS)
    cfg.CONF(project='neutron')
    logging_config.setup_logging(cfg.CONF)

    try:
        agent_config = create_agent_config_map(cfg.CONF)
    except ValueError as e:
        LOG.exception(_("%s Agent terminated!"), e)
        raise SystemExit(1)

    plugin = SdnveNeutronAgent(**agent_config)

    # Start everything.
    LOG.info(_("Agent initialized successfully, now running... "))
    plugin.daemon_loop()
```

其中，eventlet.monkey_patch()是使用 eventlet 的 patch，将本地的一些 python 库进行绿化，使之支持协程。

```
cfg.CONF.register_opts(ip_lib.OPTS)
cfg.CONF(project='neutron')
logging_config.setup_logging(cfg.CONF)
```

这三行则初始化配置信息。register_opts 是注册感兴趣的关键字并设置它们的默认值，只有感兴趣的关键字才会被后面的步骤进行配置更新。

最关键的 cfg.CONF(project='neutron')，这其实是个函数调用，实际上调用了 cfg.ConfigOpts 类的__call__方法，来解析 project 参数所指定的相关配置文件，并从中读取配置信息。需要注意的是，外部的 sys.argv 参数会传递给所 import 的 cfg 模块进行解析。因此，如果在启动 agent 的时候通过命令行给出了参数，则 cfg.ConfigOpts 类会解析这些命令行参数。否则，将默认去~/.${project}/、~/、/etc/${project}/、/etc/等地方搜索配置文件（默认为 os.path.basename(sys.argv[0])）。如果不进行这一步，那么所有的关键字只带有默认的信息，配置文件中信息就不起作用了。

```
try:
        agent_config = create_agent_config_map(cfg.CONF)
    except ValueError as e:
        LOG.exception(_("%s Agent terminated!"), e)
        raise SystemExit(1)
```

这部分则试图从全局配置库中读取 agent 相关的一些配置项。包括网桥、接口 mapping、控制器 IP 等等。

后面部分是实例化一个 SdnveNeutronAgent 类，并调用它的 daemon_loop()方法。

3.11.7.2 common/

这里面的文件主要是定义一些常量。

config.py 定义了配置选项（关键词）和默认值等，包括 sdnve_opts 和 sdnve_agent_opts 两个配置组，并且将这些配置项导入到全局的 cfg.CONF 中。只要导入该模块，相应的配置组和配置选项就会被认可合法，从而可以通过解析配置文件中这些关键词，而为这些配置选项赋值；

constants.py 则分别定义了一些固定的常量；

exceptions.py 中定义了一些异常类型。

3.11.7.3 sdnve_api.py

封装 sdnve 控制器所支持的操作为一些 API。

RequestHandler 类，处理与 sdnve 控制器的请求和响应消息的基本类。提供 get、post、put、delete 等请求。对 HTTP 消息处理的实现通过其内部的 httplib2.Http 成员来进行。

Client 类，继承自 RequestHandler 类。提供对 sdnve 中各种网络资源（网络，子网，端口，租户，路由器，浮动 IP）的 CRUD 操作的 API 和对应实现。

KeystongClient 类，主要是获取系统中的租户信息。

3.11.7.4 sdnve_neutron_plugin.py

SdnvePluginV2 类，继承自如下几个基础类：

db_base_plugin_v2.NeutronDbPluginV2：提供在数据库中对网络、子网、端口的 CRUD 操作 API；

external_net_db.External_net_db_mixin：为 db_base_plugin_v2 添加对外部网络的操作方法；

portbindings_db.PortBindingMixin：端口绑定相关的操作；

l3_gwmode_db.L3_NAT_db_mixin：添加可配置的网关模式，为端口和网络提供字典风格的扩展函数。

agents_db.AgentDbMixin：为 db_base_plugin_v2 添加 agent 扩展，对 agent 的创建、删除、获取等。

SdnvePluginV2 类实现了 neutron 中定义的 API，实现基于 SDN-VE 对上提供网络抽象的支持。包括对网络、子网、端口、路由器等资源的 CRUD 操作。

3.11.8 linuxbridge

3.11.9 metaplugin

3.11.10 midonet

3.11.11 ml2

3.11.12 mlnx

3.11.13 nec

3.11.14 nicira

3.11.15 nuage

3.11.16 ofagent

3.11.17 oneconvergence

3.11.18 openvswitch

3.11.18.1 agent/

主要包括 xenapi 目录（xen 相关）和 ovs_neutron_agent.py 文件（运行在各个节点上的对网桥进行操作的代理）。

其 main 函数主要过程如下：

```
def main():
    eventlet.monkey_patch()
    cfg.CONF.register_opts(ip_lib.OPTS)
    cfg.CONF(project='neutron')
    logging_config.setup_logging(cfg.CONF)
    q_utils.log_opt_values(LOG)

    try:
        agent_config = create_agent_config_map(cfg.CONF)
    except ValueError as e:
        LOG.error(_('%s Agent terminated!'), e)
        sys.exit(1)

    is_xen_compute_host = 'rootwrap-xen-dom0' in agent_config['root_helper']
    if is_xen_compute_host:
        # Force ip_lib to always use the root helper to ensure that ip
        # commands target xen dom0 rather than domU.
        cfg.CONF.set_default('ip_lib_force_root', True)

    agent = OVSNeutronAgent(**agent_config)
    signal.signal(signal.SIGTERM, handle_sigterm)

    # Start everything.
    LOG.info(_("Agent initialized successfully, now running... "))
    agent.daemon_loop()
    sys.exit(0)
```

首先是读取各种配置信息，然后提取 agent 相关的属性。

然后生成一个 agent 实例，并调用其 daemon_loop()函数，该函数进一步执行 rpc_loop()。

agent 实例初始化的时候，会依次调用 setup_rpc()、setup_integration_br()和 setup_physical_bridges()。

3.11.18.1.1 setup_rpc()

setup_rpc()创建了两个 rpc，分别是

```
self.plugin_rpc = OVSPluginApi(topics.PLUGIN)
self.state_rpc = agent_rpc.PluginReportStateAPI(topics.PLUGIN)
```

其中，前者是与 neutron-server（准确的说是 ovs plugin）进行通信，后者是 agent 将自身的状态上报给 neutron-server。

之后，创建 dispatcher 和所关注的消息主题：

```
self.dispatcher = self.create_rpc_dispatcher()
        # Define the listening consumers for the agent
        consumers = [[topics.PORT, topics.UPDATE],
                     [topics.NETWORK, topics.DELETE],
                     [constants.TUNNEL, topics.UPDATE],
                     [topics.SECURITY_GROUP, topics.UPDATE]]
```

这样，neutron-server 发到这四个主题的消息，会被 agent 接收到。agent 会检查端口是否在本地，如果在本地则进行对应动作。

创建 rpc 连接：

```
self.connection = agent_rpc.create_consumers(self.dispatcher,
                                             self.topic,
                                             consumers)
```

最后，创建 heartbeat，定期的调用 self._report_state()，通过 state_rpc 来汇报本地状态。

3.11.18.1.2 setup_integration_br()

清除 integration 网桥上的 int_peer_patch_port 端口和流表，添加一条 normal 流。

3.11.18.1.3 setup_physical_bridges()

创建准备挂载物理网卡的网桥，添加一条 normal 流，然后创建 veth 对，连接到 integration 网桥，添加 drop 流规则，禁止未经转换的流量经过 veth 对。

3.11.18.2 common/

包括 config.py 和 constants.py 两个文件。

其中 config.py 文件中定义了所关注的配置项和默认值，并注册了 OVS 和 AGENT 两个配置组到全局的配置项中。

而 constants.py 中则定义了一些常量，包括 ovs 版本号等。

3.11.18.3 ovs_db_v2.py

跟 ovsdb 打交道的一些函数，包括获取端口和网络绑定信息等。

3.11.18.4 ovs_models_v2.py

定义了继承自 model_base.BASEV2 的四个类。

NetworkBinding 代表虚拟网和物理网的绑定。

TunnelAllocation 代表隧道 id 的分配状态。

TunnelEndpoint 代表隧道的一个端点。

VlanAllocation 代表物理网上的 vlan id 的分配状态。

3.11.18.5 ovs_neutron_plugin.py

plugin 的主要实现。

包括三个类：AgentNotifierApi、OVSNeutronPluginV2 和 OVSRpcCallbacks。

AgentNotifierApi 代表了 openvswitch 进行 rpc api 时往 agent 端发出的操作。包括三个函数：network_delete、port_update 和 tunnel_update，分别发出消息到指定主题上，该消息会被 agent 所监听到。

OVSRpcCallbacks 负责对 agent 发来的 rpc 消息（包括获取设备，获取端口、同步 tunnel、更新设备状态）进行的处理。例如收到一个设备起来的消息，则调用 update_device_up()来将 ovsdb 中的设备状态置为 ACTIVE。

OVSNeutronPluginV2 是 plugin 的主类。其初始化过程读取和检查配置参数。然后调用 setup_rpc()创建相关的 rpc。注册监听 topics.PLUGIN、和 topics.L3PLUGIN 两个主题的消息。并创建了对 plugin agent、dhcp agent 和 l3 agent 的通知 api。

3.11.19 plumgrid

3.11.20 ryu

3.11.21 vmware

3.12 scheduler

调度、负载均衡等。

3.12.1 dhcp-agent_scheduler.py

3.12.2 l3-agent_scheduler.py

3.13 server

实现 neutron-server 的主进程。包括一个 main()函数，是 WSGI 服务器开始的模块，并且通过调用 serve_wsgi 来创建一个 NeutronApiService 的实例。然后通过 eventlet 的 greenpool 来运行 WSGI 的应用程序，响应来自客户端的请求。

主要过程为：

```
eventlet.monkey_patch()
```

绿化各个模块为支持协程（通过打补丁的方式让本地导入的库都支持协程）。

```
config.parse(sys.argv[1:])
if not cfg.CONF.config_file:
    sys.exit(_("ERROR: Unable to find configuration file via the default"
               " search paths (~/.neutron/, ~/, /etc/neutron/, /etc/) and"
               " the '--config-file' option!"))
```

通过解析命令行传入的参数，获取配置文件所在。

```
pool = eventlet.GreenPool()
```

创建基于协程的线程池。

```
neutron_api = service.serve_wsgi(service.NeutronApiService)
api_thread = pool.spawn(neutron_api.wait)
```

创建 NeutronApiService 实例（作为一个 WsgiService），并调用 start()来启动 socket 服务器端，还会通过调用 load_paste_app()方法从配置文件读取相关的配置信息来生成一个 WSGI 的应用。通过新的协程进行处理。

```
try:
        neutron_rpc = service.serve_rpc()
    except NotImplementedError:
        LOG.info(_("RPC was already started in parent process by plugin."))
    else:
        rpc_thread = pool.spawn(neutron_rpc.wait)
        rpc_thread.link(lambda gt: api_thread.kill())
        api_thread.link(lambda gt: rpc_thread.kill())
```

创建 rpc 请求服务，并将 api 和 rpc 的生存绑定到一起，一个死掉，则另外一个也死掉。

```
pool.waitall()
```

最后是后台不断等待。

3.14 services

包括 WsgiService 类和继承自它的 NeutronApiService 类、RpcWorker 类、继承自 n_rpc.Service 的 Service 类。

一个 service 用来管理一组应用程序的运行，并且有一个 manager，通过监听队列（注册到某个 topic）来使用 rpc，并且可以周期性的运行计划任务。支持 start、stop、create、kill、periodic_tasks()、report_state()和 wait()等方法。

3.15 tests

3.16 其他文件

3.16.1 auth.py

3.16.2 context.py

3.16.3 hooks.py

3.16.4 manager.py

3.16.5 neutron_plugin_base_v2.py

该文件作为 plugin 的基础类，是实现 plugin 的参考和基础，它其中声明了实现一个 neutron plugin 所需的基本方法。

包括下面的方法：

属性	create	delete	get	update
port	Y	Y	Y	Y
ports			Y	

ports_count			Y	
network	Y	Y	Y	Y
networks			Y	
networks_count			Y	
subnet	Y	Y	Y	Y
subnets			Y	
subnet_count			Y	

3.16.6 policy.py

3.16.7 quota.py

3.16.8 service.py

定义了相关的配置信息，包括 periodic_interval，api_workers，rcp_workers，periodic_fuzzy_delay。

实现 neutron 中跟服务相关的类。

包括 NeutronApiService，RpcWorker，继承自 n_rpc.Service 的 Service 和 WsgiService。

Service 是各个服务的基类。

WsgiService 是实现基于 WSGI 的服务的基础类。

NeutronApiService 继承自 WsgiService，添加了 create()方法，配置 log 相关的选项，并返回类实体。

3.16.8.1 Service 类

Service 是很重要的一个概念，各个服务的组件都以 Service 类的方式来进行交互。此处 Service 类继承自 rpc 中的 Service，整体的继承关系为

neutron.openstack.common.service.Service 类-->neutron.common.rpc.Service 类-->neutron.service.Service 类。

其中 neutron.openstack.common.service.Service 类定义了简单的 reset()、start()、stop()和 wait()方法。该类初始化后会维护一个线程组。

neutron.common.rpc.Service 类中进一步丰富了 start()和 stop()方法，并在初始化中引入了 host、topic、manager 和 serializer 参数。

start()增加创建了 Connection 对象，之后创建了三个 consumer，分别监听主题为参数传入的 topic（fanout 分别为 True 和 False），以及主题为 topic.host。然后调用 manager 的初始化。最后作为 server 启动所有的 consumer。

neutron.service.Service 类的初始化中更进一步的增加了 binary、report_interval、periodic_interval、periodic_fuzzy_delay 等参数。除丰富了 start()、stop()和 wait()方法外，还增加了 create()类方法、kill()、periodic_tasks()和 report_state()。

start()增加了周期性执行 report_state()和 periodic_tasks()，并且调用 manager 的 init_host()和 after_start()方法。

create()方法是类方法，它根据传入的参数 binary 参数获取真实的程序名，并在未给定参数的情况下尝试从配置文件中解析 manager 和 report_interval、periodic_interval、periodic_fuzzy_delay 等参数。最后是返回生成的 Service 类对象。

report_state()方法仅定义了接口。

periodic_tasks()则首先获取 admin 的上下文，然后调用 manager 的 periodic_tasks()方法执行。

总结一下，neutron.service.Service 类，初始化会处理传入参数，并解析配置文件。start()方法则创建并启动三个 consumer，监听传入的 topic 和 topic.host。初始化 manager 并周期性运行它的 periodic_tasks()和 report_state()方法。

3.16.9 version.py

3.16.10 wsgi.py

第 4 章 理解 Neutron 代码

本部分试图从专题和业务流程的角度来剖析 neutron 代码，以便理解如此设计的内涵。

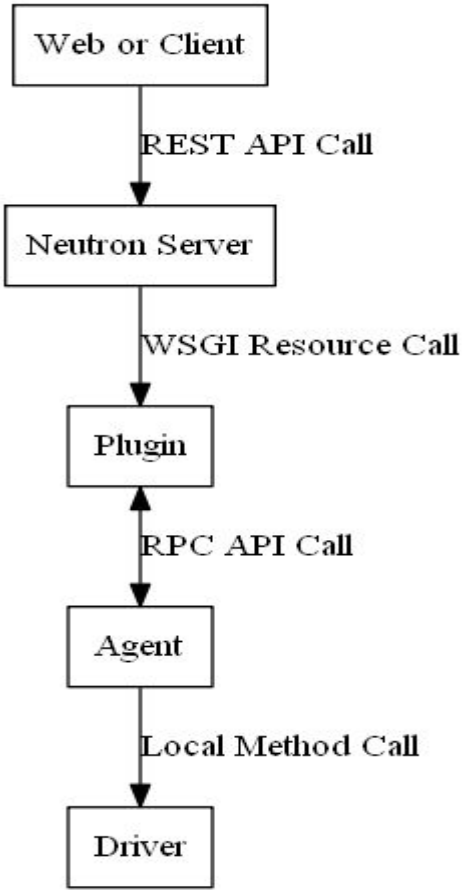
4.1 代码逻辑

OpenStack 在设计上，保持了较好的可扩展风格，包括所有的组件之间尽量松耦合，代码逻辑严格分层，以及对服务 cluster 的支持等。

在 OpenStack 中，大部分的组件都可以看做是服务或资源。如果需要获取其他组件的信息，就利用某种方式进行 api 访问。实际上，按照计算（nova）、网络（neutron）、鉴

权（keystone）、存储（cinder）、对象（swift）这几个服务来看。服务之间通过 Rest API 相互访问，服务内的不同组件通过 RPC API 来访问。

以 neutron 为例，整体的自上而下代码逻辑如下图所示。



可见，各层调用使用的方法不同，这一方面是从执行效率上考虑，也是从逻辑耦合关系上进行考虑。

4.2 Rest API 专题

4.3 RPC 专题

4.4 Plugin 专题

4.5 Extension 专题

4.6 Agent 专题