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:
V0.71: 2014-08-07
　添加更多细节分析，添加对 ML2 的分析。
V0.7: 2014-07-18
　完成对 cmd、common 和 db 部分的分析；
　整体代码框架分析完毕。
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 个目录和若干文件：

bin，doc，etc，neutron 和 tools。

除了这 5 个目录外，还包括一些说明文档、安装需求说明文件等。

1.1 bin

主要包括 neutron-rootwrap 和 neutron-rootwrap-xen-dom0 两个文件。

提供一些可执行命令。。

1.2 doc

包括文档生成的相关源码。

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 226k 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　bin

neutron-rootwrap 文件，python 可执行文件

neutron-rootwrap-xen-dom0 文件，python 可执行文件。

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。

第 3 章　doc

可以利用 sphinx 工具来生成文档。
source 子目录：文档相关的代码。
Makefile：用户执行 make 命令。
pom.xml：maven 项目管理文件。
用户在该目录下通过执行 make html 可以生成 html 格式的说明文档。

第 4 章　etc

4.1 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
    ;;
```

4.2 neutron/

4.2.1 plugins/

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

4.2.2 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。

4.3 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。

4.4 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。

4.5 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
```

4.6 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。

4.7 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。

4.8 metadata_agent.ini

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

4.9 metering_agent.ini

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

4.10 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。

4.11 neutron.conf

neutron-server 启动后读取的配置信息。

4.12 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。

4.13 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。

4.14 services.conf

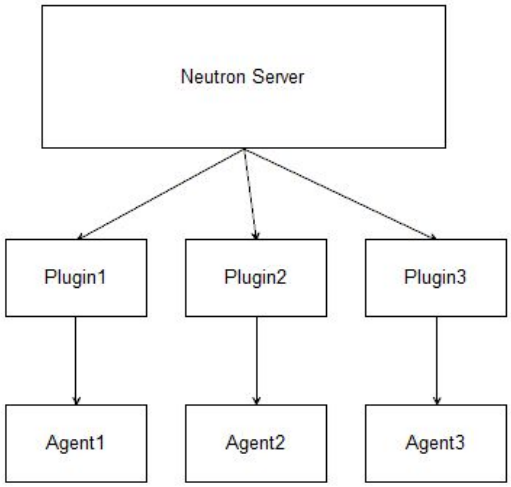
配置一些特殊的 service 信息。

第 5 章　neutron

该目录下包含了 neutron 实现的主要代码。

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。

5.1 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。

5.1.1 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。

5.1.2 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。

5.1.2.1 async_process.py

实现了 AsyncProcess 类，对异步进程进行管理。

5.1.2.2 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。

5.1.2.3 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。

5.1.2.4 external_process.py

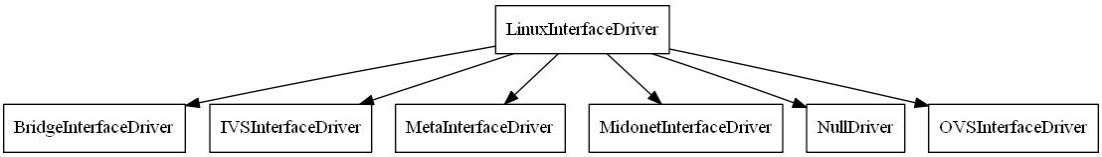
定义了 ProcessManager 类，对 neutron 孵化出的进程进行管理（可以通过跟踪 pid 文件进行激活和禁用等）。

5.1.2.5 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。

5.1.2.6 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。

5.1.2.7 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()。

5.1.2.8 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。

5.1.2.9 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。

5.1.2.10 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）。

5.1.2.11 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。

5.1.2.12 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。

5.1.3 metadata/

5.1.3.1 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。

5.1.3.2 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 NetworkMetadataProxyHandler。

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

5.1.4 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']
```

5.1.4.1 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()。

5.1.4.2 DhcpPluginApi 类

提供从 agent 往 plugin 一侧进行 rpc 调用的 api。

5.1.4.3 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)
```

5.1.5 firewall.py

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

5.1.6 l2population_rpc.py

主要定义了 L2populationRpcCallBackMixin 元类。

5.1.7 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。

5.1.7.1 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。

5.1.7.2 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。

5.1.7.3 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。

5.1.8 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)
```

5.1.9 ovs_cleanup_util.py

清理无用的 ovs 网桥和端口。

5.1.10 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，用于 agent 调用 plugin 的方法。后者是 agent 汇报自身状态，用于向 plugin 汇报状态信息。

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以 rpc 消息的方式发出去。

5.1.11 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。

5.2 api

提供 RestAPI 访问。

5.2.1 rpc

5.2.1.1 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."""
```

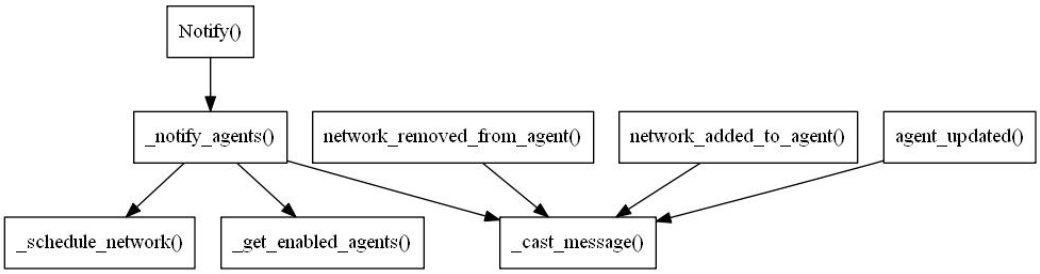
```
        self.cast(
            context, self.make_msg(method, payload=payload),topic='%s.%s' % (topic, host))

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图表 2 所示。



图表 2 API 的 agent 通知调用

5.2.1.2 handler

包括 dvr_rpc.py 文件。

其中定义了对 dvr 服务两端的 rpc 的 api 和 callback 类，包括 DVRServerRpcApiMixin、DVRServerRpcCallbackMixin、DVRAgentRpcApiMixin、DVRAgentRpcCallbackMixin。

5.2.2 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。

5.2.2.1 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。

5.2.2.2 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，将对资源的请求转化为对应的 plugin 中方法的调用。

成员包括一个对 dhcp agent 的 notifier。

5.2.2.3 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。

5.2.2.4 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。

5.2.2.5 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。_map_resource 方法中创建了对应的控制器和映射关系。控制器在 base.py 文件中，其中定义了 index、show、create、delete 和 update 方法。这些方法中会获取 plugin 的对应方法对请求进行处理。例如，在 create 方法中有

```
obj_creator = getattr(self._plugin, action)
...
obj = obj_creator(request.context, **kwargs)
```

5.2.3 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。

5.2.4 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。

5.2.5 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。

5.2.6 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
```

5.3 cmd

usage_audit.py，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），显示它们的信息。

sanity_check.py，进行一些简单的检查，包括是否支持 vxlan，是否支持 patch 端口，是否支持 nova 的 notify 等。

5.4 common

这个包里面定义的都是一些模块通用的功能，包括对配置的操作，日志管理、rpc 调用，以及一些常量等。

5.4.1 config.py

对配置进行管理。

定义了 core_opts 的属性和默认值，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。

定义了 core_cli_opts 的属性和默认值，包括状态文件的路径。

注册上面定义的配置项。

主要包括

load_paste_app(app_name)方法，从默认的 paste config 文件来读取配置，生成并返回 WSGI 应用。最关键的逻辑实现是

```
app = deploy.loadapp("config:%s" % config_path, name=app_name)
```

init(args)方法，读入配置文件，调用 rpc 的初始化函数，并检查 base_mac 参数是否合法。

setup_logging(conf)方法，配置 logging 模块，导入配置信息。

5.4.2 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

5.4.3 exceptions.py

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

5.4.4 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 地址前缀和 mac 来获取 v6 地址。

5.4.5 log.py

基于 neutron.openstack.common 中的 log 模块。

主要是定义了 log 修饰，在执行方法时会一条 debug 日志，包括类名，方法名，参数等信息。

5.4.6 rpc.py

定义了类 RequestContextSerializer，RpcProxy，RpcCallback，Service，Connection。

其中 RequestContextSerializer 类中定义了对实体和上下文的序列化/反序列化，将 RPC 的通用上下文转化到 Neutron 上下文。

RpcProxy 类提供 rpc 层的操作，基本上所有需要进行 rpc 调用的应用都会用到这个类。其中分别定义了 call，cast 和 fanout_cast 方法来发出 rpc 调用请求。

Service 类代表运行在主机上的应用程序所代表的的服务，继承自 service.Service，重载了 start 方法和 stop 方法。start 方法中会创建三个消费连接来监听 rpc 请求。第一个是监听发送到某个 topic 上的所有主机上，第二个是监听发送到某个 topic 的特定主机上，最后一个是所有的广播请求。

Connection 类代表了 rpc 请求的相关连接。

5.4.7 test_lib.py

定义了 test_config={}，用于各个 plugin 进行测试。

5.4.8 topics.py

管理 rpc 调用过程中的 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'
```

5.4.9 utils.py

一些辅助函数，包括查找配置文件，封装 subprocess.Popen 的 subprocess_open，解析映射关系、获取主机名、字典和字符串格式的转化、检查对扩展的支持等等。

5.5 db

跟数据库相关的操作。因为各项服务根本上的操作都需要跟数据库打交道，因此这部分定义了大量的数据库资源类和相关接口，可以被进一步继承实现。

包括对核心 plugin api 的实现基础类，其次是一些扩展的资源和方法的支持。

其中 model_base.py 和 models_v2.py 中定义了最基础的几个模型类。

5.5.1 api.py

定义了一些数据库的配置方法，包括

configure_db()启动一个数据库，创建一个引擎然后注册模型。

get_engine()方法获取引擎。

get_sessions()方法获取会话信息。

5.5.2 agents_db.py

包括三个类，继承自 model_base.BASEV2 和 models_v2.HasId 的 Agent 类，继承自 neutron.extensions.agent.AgentPluginBase 的 AgentDbMixin 类，继承自 rpc.RpcCallback 的 AgentExtRpcCallback。

Agent 类表示数据库中对一个 agent 的相关信息的记录。

AgentDbMixin 类用于添加对 agent 扩展的支持到 db_base_plugin_v2，提供了获取配置、对 agent 进行 curd 操作等方法。

AgentExtRpcCallback 类在 plugin 的实现中用于处理 rpc 汇报，其中定义了 report_state()方法用于向 plugin 汇报状态。

5.5.3 agentschedulers_db.py

跟 agent schedululer 相关的一些数据库资源和操作类。

继承自 model_base.BASEV2 的 NetworkDhcpAgentBinding 类表示数据库中网络和 dhcpagent 的一条绑定关系。

继承自 agents_db.AgentDbMixin 的 AgentSchedulerDbMixin 类。

继承自 AgentSchedulerDbMixin 类的 DhcpAgentSchedulerDbMixin 类用于添加 dhcp agent 扩展的支持到 db_base_plugin_v2。

5.5.4 common_db_mixin.py

定义了在核心 plugin 和服务 plugin 中的常见的数据库方法。

实现了扩展的插件类则需要通过 register_model_query_hook 来注册它的 hook。

5.5.5 db_base_plugin_v2.py

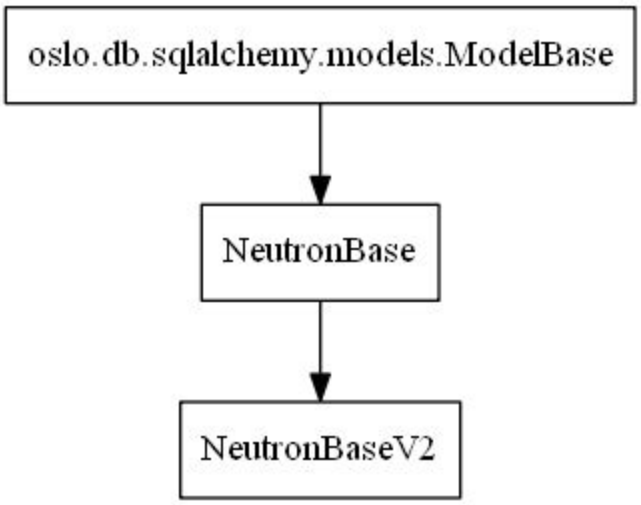
其中定义了 NeutronDbPluginV2，是各个 plugin 的基础类。

继承自 neutron.neutron_plugin_base_v2.NeutronPluginBaseV2 类和 neutron.common_db_mixin.CommonDbMixin 类。

该类利用 SQLAlchemy 的模型实现了 neutron plugin 的接口，主要包括对网络、子网、端口等资源的 CRUD 操作。

5.5.6 model_base.py

定义了继承自 oslo.db.sqlalchemy.models.ModelBase 的 NeutronBase 和继承自其的 NeutronBaseV2 类。



实际上，oslo.db.sqlalchemy.models.ModelBase 是 sqlalchemy 的模型基础类，因此继承自其的类自动完成映射，可以对数据库进行操作。

最后，还定义了 BASEV2 作为其他地方可以继承的模型类。

```
BASEV2 = declarative.declarative_base(cls=NeutronBaseV2)
```

5.5.7 models_v2.py

通过继承 model_base.BASE2，定义了几个数据模型，包括 IPAllocationPool，IPAllocation，Route，SubnetRoute，Port，DNSNameServer，Subnet，Network。

5.5.8 dhcp_rpc_base.py

定义了 DhcpRpcCallbackMixin 类，为 plugin 提供 dhcp agent 的 rpc 支持。

其中实现了对 dhcp 端口的获取、释放、创建、更新等方法。

5.5.9 l3_rpc_base.py

定义了类 L3RpcCallbackMixin，在 plugin 的实现中添加 l3 agent 的 rpc 支持。

5.5.10 securitygroups_rpc_base.py

实现对安全组的扩展 rpc 的基础支持。

继承自 neutron.db.securitygroups_rpc_base.SecurityGroupDbMixin 类的 SecurityGroupServerRpcMixin，以及 SecurityGroupServerRpcCallbackMixin 类。

后者为 plugin 提供了 securitygroup 的 agent 的支持。

5.5.11 migration

负责将原先的 ovs 或 linux bridge 的数据库迁移到 ml2 支持的数据库格式。

包括 cli.py，migrate_to_ml2.py 等。

5.5.12 sqlalchemyutils.py

定义了 sqlalchemyutils.py 方法，根据指定的排序和标记条件来返回一个查询。

5.5.13 扩展资源和操作类

包括一系列资源在数据库中对应的记录和操作，这些模块的格式都很相似，一般包括若干个静态资源类和一个操作的 mixin 实现类。这个 mixin 类一般都是扩展核心 plugin 的资源、方法等支持，即提供扩展资源操作，一般继承自 extension 包中对应的基础类。

5.5.13.1 firewall

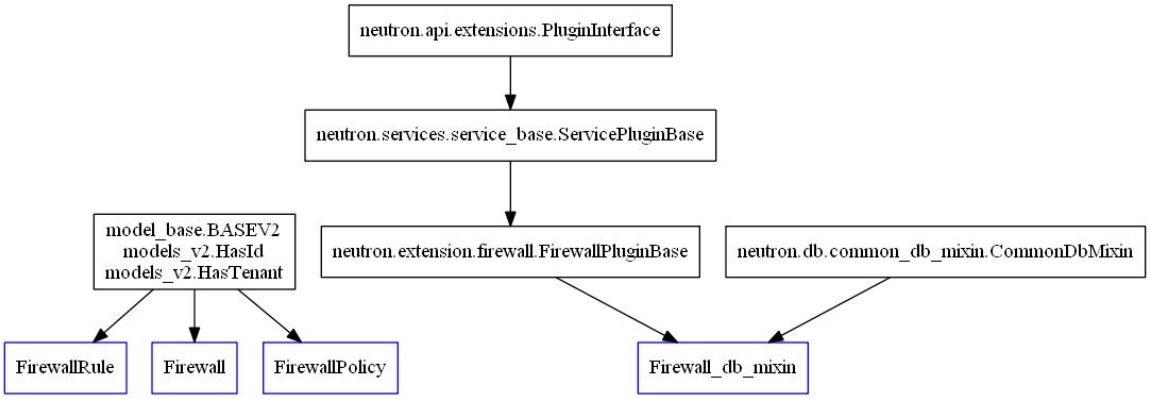
firewall_db.py 中定义了跟防火墙数据库相关的几个类，包括 FirewallRule、Firewall、FirewallPolicy 和 Firewall_db_mixin。

FirewallRule 表示数据库中一条防火墙规则记录；

Firewall 表示数据库中一个防火墙资源记录；

FirewallPolicy 表示数据库中一条防火墙策略记录；

Firewall_db_mixin 表示数据库中防火墙相关操作的实现类，包括创建、删除、更新和获取各种防火墙资源等操作；



5.5.13.2 loadbalancer

loadbalancer_db.py 中定义了跟负载均衡服务相关的几个数据库资源和操作类。

资源类都继承自 model_base.BASE2 类，此外还根据需求继承了其他几个类增添属性。

SessionPersistence 类表示数据库中一条 session 的持久化类型；

PoolStatistics 类表示数据库中一个 pool 的一些统计信息；

Vip 类表示数据库中一个 VIP 记录；

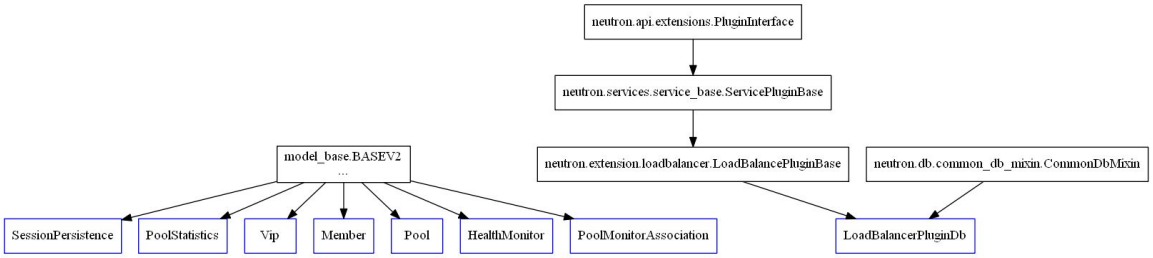
Member 类表示一个负载均衡的成员；

Pool 类表示一个负载均衡池资源；

HealthMonitor 类表示一个负载均衡的健康状态监视器资源；

PoolMonitorAssociation 类表示 Pool 到 HealthMonitor 的关联关系。

LoadBalancerPluginDb 则继承自 loadbalancer.LoadBalancerPluginBase 和 base_db.CommonDbMixin，代表对负载均衡相关的数据资源进行操作实现。



5.5.13.3 metering

包括 metering_db.py 和 metering_rpc.py。

前者跟 firewall_db.py 类似，定义了 metering 的资源和操作实现类。

后者主要定义了 MeteringRpcCallbacks 类。

5.5.13.4 vpn

vpn_db.py 文件跟 firewall_db.py 类似，定义了 VPN 的资源和操作实现类，以及一个 VPNPluginRpcDbMixin 类。

vpn_validator.py 文件定义了 VpnReferenceValidator 类，对 vpn 资源进行校验。

5.5.13.5 allowedaddresspairs_db.py

类似的，定义了数据库资源类 AllowedAddressPair，和操作类 AllowedAddressPairsMixin。

5.5.13.6 dvr_mac_db.py

数据库资源类 DistributedVirtualRouterMacAddress，和操作类 DVRDbMixin。后者继承自 neutron.extensions.dvr.DVRMacAddressPluginBase 类。

5.5.13.7 external_net_db.py

定义了继承自 model_base.BASEV2 的数据库资源类 ExternalNetwork，以及操作类 External_net_db_mixin。后者为 db_base_plugin_v2 添加了对外部网络方法的支持。

5.5.13.8 extradhcpopt_db.py

代表绑定到某个端口上的额外的属性。

定义了继承自 model_base.BASEV2 类和 models_v2.HasId 类的数据库资源类 ExtraDhcpOpt，以及操作类 ExtraDhcpOptMixin。

5.5.13.9 extraroute_db.py

定义了继承自 model_base.BASEV2 类、models_v2.Route 类的数据库资源类 RouterRoute 和资源操作类 ExtraRoute_db_mixin。

5.5.13.10 l3_attrs_db.py

定义了 RouterExtraAttributes 类（继承自 model_base.BASE 类），代表一个路由器的附加属性。

定义了 ExtraAttributesMixin 类，用来对这些虚拟属性进行支持。

5.5.13.11 l3_agentschedulers_db.py

5.5.13.12 l3_db.py

代表了路由器、浮动 IP 资源和对应操作实现。

定义了继承自 model_base.BASEV2 类、models_v2.HasId 类和 models_v2.HasTenant 类的数据库资源类 Router 和 FloatingIP，以及继承自 neutron.extensions.l3.RouterPluginBase 类的资源操作类 L3_NAT_db_mixin。L3_NAT_db_mixin 实际上为核心 plugin 添加了 L3/NAT 的扩展资源方法。

5.5.13.13 l3_dvr_db.py

定义了类 L3_NAT_with_dvr_db_mixin，对 DVR 进行支持。

5.5.13.14 l3_dvrscheduler_db.py

5.5.13.15 l3_gwmode_db.py

定义了继承自 l3_db.L3_NAT_db_mixin 类的 L3_NAT_db_mixin，添加对可配置网关路由器的支持。

5.5.13.16 portbindings_base.py

定义了端口绑定的基础类 PortBindingBaseMixin。

5.5.13.17 portbindings_db.py

定义了继承自 model_base.BASEV2 类的 PortBindingPort 资源类，以及继承自 portbindings_base.PortBindingBaseMixin 类的资源操作类 PortBindingMixin。

5.5.13.18 portsecurity_db.py

定义了继承自 model_base.BASEV2 类的 PortSecurityBinding 资源类和 NetworkSecurityBinding 资源类，以及资源操作类 PortSecurityDbMixin。

5.5.13.19 quota_db.py

定义了继承自 model_base.BASEV2 类、models_v2.HasId 类的数据库资源类 Quota 和资源操作类 DbQuotaDriver。

5.5.13.20 routedserviceinsertion_db.py

定义了继承自 model_base.BASEV2 类的数据库资源类 ServiceRouterBinding 和资源操作类 RoutedServiceInsertionDbMixin。

5.5.13.21 routerservicetype_db.py

实现对 router 服务类型的支持。

定义了继承自 model_base.BASEV2 类的数据库资源类 RouterServiceTypeBinding 和资源操作类 RouterServiceTypeDbMixin。

5.5.13.22 securitygroups_db.py

实现对安全组资源的支持。

包括三个资源类：SecurityGroup、SecurityGroupPortBinding、SecurityGroupRule，以及一个资源操作类 SecurityGroupDbMixin。

5.5.13.23 servicetype_db.py

定义了继承自 model_base.BASEV2 类的数据库资源类 ProviderResourceAssociation 和资源操作类 ServiceTypeManager。

5.6 debug

提供简单的辅助 debug 功能。

5.6.1 commands.py

包括若干命令的实现类：ProbeCommand 基类和继承自它的 CreateProbe，DeleteProbe，ListProbe，ClearProbe，ExecProbe，PingAll。

5.6.2 debug_agent.py

通过调用 client 来执行各项查询操作和 debug 命令。

5.6.3 shell.py

提供一个 shell 环境来完成 debug。

5.7 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，可以以 extension 的方式实现。包括 firewall、vpnaas、l3、lbaas 等。

5.7.1 agent.py

主要定义了两个类：Agent 和 AgentPluginBase。

前者提供对 agent 管理扩展；后者提供对 agent 进行操作的 rest API，包括对 agent 的 CRUD 操作，在 agent_db.py 中被 AgentDbMixin 类继承。

5.7.2 扩展资源类

这些扩展资源类，大部分都继承自 neutron.api.extensions.ExtensionDescriptor 类，一般都实现了如下的类方法。

get_name 、 get_alias 、 get_description 、 get_namespace 、 get_updated 、 get_extended_resources。

5.7.2.1 allowedaddresspairs.py

主要定义了 Allowedaddresspairs 类，表示支持允许地址对的扩展类。

5.7.2.2 dhcpagentscheduler.py

NetworkSchedulerController 类，继承自 wsgi.Controller，对网络调度器进行创建、删除和索引。

DhcpAgentsHostingNetworkController 类，继承自 wsgi.Controller，对 dhcp_agent_hosting_network 进行索引操作。

Dhcpagentscheduler 类，继承自 extensions.ExtensionDescriptor，对 dhcp agent 的调度进行支持。

DhcpAgentSchedulerPluginBase 类，提供对 dhcp agent 的调度器进行操作的 REST API，包括添加、删除和列出网络到 dhcp agent 等。

5.7.2.3 dvr.py

定义了 Dvr 类，代表分布式虚拟路由器的扩展类。

5.7.2.4 external_net.py

External_net 扩展类，继承自 extensions.ExtensionDescriptor，为 dhcp 增加配置选项。

5.7.2.5 extra_dhcp_opt.py

Extra_dhcp_opt 扩展类，继承自 extensions.ExtensionDescriptor，为网络增加 external network 属性。

5.7.2.6 extraroute.py

Extraroute 类，提供附加的路由支持。

5.7.2.7 firewall.py

Firewall 扩展类，继承自 extensions.ExtensionDescriptor，提供防火墙扩展支持。

FirewallPluginBase 抽象基础类，继承自 service_base.ServicePluginBase，定义了防火墙 plugin 的基础接口。

5.7.2.8 flavor.py

Flavor 扩展类，继承自 extensions.ExtensionDescriptor，为网络和路由器提供 flavor 支持。

5.7.2.9 l3.py

L3 扩展类，继承自 extensions.ExtensionDescriptor，提供路由器支持。

RouterPluginBase，抽象基础类，提供对路由器的操作，被 l3_db.py 中的 L3_NAT_db_mixin 类继承。

5.7.2.10 l3_ext_gw_mode.py

L3_ext_gw_mode 扩展类，继承自 extensions.ExtensionDescriptor，为路由器提供外部网关支持。

5.7.2.11 l3agentscheduler.py

RouterSchedulerController 类和 L3AgentsHostingRouterController 类，都继承自 wsgi.Controller。

L3agentscheduler 扩展类，继承自 extensions.ExtensionDescriptor，支持 l3 agent 的调度器。

L3AgentSchedulerPluginBase，为管理 l3 agent 调度器提供 REST API 支持。

5.7.2.12 lbaas_agentscheduler.py

PoolSchedulerController 类和 LbaasAgentHostingPoolController 类，都继承自 wsgi.Controller。

Lbaas_agentscheduler 扩展类，继承自 extensions.ExtensionDescriptor，支持 lbaas agent 调度器的支持。

LbaasAgentSchedulerPluginBase，为管理 l3 agent 调度器提供 REST API 支持。

5.7.2.13 loadbalancer.py

Loadbalancer 扩展类，继承自 extensions.ExtensionDescriptor，支持 lbaas agent 调度器的支持。

LoadBalancerPluginBase 抽象基础类，继承自 service_base.ServicePluginBase。被 loadbalancer_db.py 中的 LoadBalancerPluginDb 类继承。

5.7.2.14 metering.py

Metering 扩展类，继承自 extensions.ExtensionDescriptor，提供 metering 扩展支持。

MeteringPluginBase 抽象基础类，继承自 service_base.ServicePluginBase。被 metering_db.py 中的 MeteringDbMixin 类继承。

5.7.2.15 multiprovidernet.py

Multiprovidernet 扩展类，继承自 extensions.ExtensionDescriptor，提供多 provider 网络的扩展支持。

5.7.2.16 portbindings.py

Portbindings 扩展类，继承自 extensions.ExtensionDescriptor，提供端口绑定扩展支持。

5.7.2.17 portsecurity.py

Portsecurity 扩展类，提供端口安全支持。

5.7.2.18 providernet.py

Providernet 扩展类，继承自 extensions.ExtensionDescriptor，提供 provider 网络扩展支持。

5.7.2.19 quotasv2.py

QuotaSetsController 类，继承自 wsgi.Controller。

Quotasv2 扩展类，继承自 extensions.ExtensionDescriptor，提供 quotas 管理支持。

5.7.2.20 routedserviceinsertion.py

Routedserviceinsertion 扩展类，提供路由服务类型支持。

5.7.2.21 routerservicetype.py

Routerservicetype 扩展类，提供路由服务类型支持。

5.7.2.22 securitygroup.py

Securitygroup 扩展类，继承自 extensions.ExtensionDescriptor，提供安全组扩展支持。

SecurityGroupPluginBase 抽象基础类，定义对安全组进行管理操作的接口。

5.7.2.23 servicetype.py

Servicetype 扩展类，继承自 extensions.ExtensionDescriptor，提供服务类型的扩展支持。

5.7.2.24 vpnaas.py

Vpnaas 扩展类，继承自 extensions.ExtensionDescriptor，提供安全组扩展支持。

VPNPluginBase 抽象基础类，继承自 service_base.ServicePluginBase，定义对 vpn 服务进行管理操作的接口。这个类被 vpn_db.py 中的 VPNPluginDb 类继承。

5.8 hacking

checks.py 中定义了一些辅助函数。

5.9 locale

多语言支持。包括语言 de，en_AU，en_GB，en_US，es，fr，it，ja，ko_KR，pt_BR，sr，zh_CN，zh_TW。

在各子目录下包括各个语言对应的字符串。

5.10 notifiers

5.10.1 nova.py

Notifier 类，发送 nova 可能关心的一些事件消息。

5.11 openstack

5.11.1 common

公共模块。

5.11.1.1 cache

_backends
backends.py
cache.py

5.11.1.2 fixture

config.py
lockutils.py
mockpatch.py
moxstubout.py

5.11.1.3 middleware

base.py
catch_errors.py
correlation_id.py
debug.py
request_id.py
sizelimit.py

5.11.1.4 context.py

定义类 RequestContext，代表在 request 上下文中的有用信息，包括用户名、租户、认证信息等等。

还定义了两个全局方法 get_admin_context()和 get_context_from_function_and_args()。

5.11.1.5 eventlet_backdoor.py

5.11.1.6 excutils.py

定义了类 save_and_reraise_exception，可以保存现在的异常，运行一些代码然后重新触发。

5.11.1.7 fileutils.py

定义了跟文件操作相关的一些方法，比如创建目录，读取修改的文件和安全删除等。

5.11.1.8 gettextutils.py

翻译相关的类和方法。

5.11.1.9 importutils.py

跟导入相关的方法，包括根据字符串导入类和对象等。

5.11.1.10 jsonutils.py

json 处理，包括将对象转化为可序列化，dumps 和 load 等。

5.11.1.11 local.py

管理对线程局部的变量。

5.11.1.12 lockutils.py

跟锁相关的方法，包括锁和同步等。

5.11.1.13 log.py

日志相关的类和方法。

5.11.1.14 loopingcall.py

一些需要循环调用的类。

基础类 LoopingCallBase，和继承自其的 FixedIntervalLoopingCall 类和 DynamicLoopingCall 类。

5.11.1.15 network_utils.py

包括解析字符串为主机：端口格式的对的方法和解析 url 字符串方法。

5.11.1.16 periodic_task.py

定义了类 PeriodicTasks，表示定期的任务。

5.11.1.17 policy.py

对访问策略的管理，处理 policy.json 文件。

类 Rules，代表一条规则。

一系列的继承自 BaseCheck 的类，代表各种对规则格式进行的检查，例如是否为真，与或非逻辑等。

5.11.1.18 processutils.py

封装了一些进程操作，提供高层方法，包括：

_subprocess_setup()方法

execute()方法，在 shell 中通过 subprocess 来执行一条命令。

trycmd()方法，对 execute()方法的封装，处理警告和错误信息。

ssh_execute

5.11.1.19 service.py

定义了基础类 service 和 services，前者被 rpc.Service 类继承。

5.11.1.20 sslutils.py

定义了若干方法来处理 ssl 协议。

5.11.1.21 strutils.py

对字符串处理，包括解析字符串为 bool 类型或 bytes 等。

5.11.1.22 systemd.py

跟 systemd 打交道的若干方法，包括 notify()方法，发送通知给 systemd 系统服务等

5.11.1.23 threadgroup.py

定义了类 Thread 和 ThreadGroup。

前者是对 greenthread 的封装，拥有一个到 threadgroup 的引用，当线程结束时候通知 threadgroup 将自身移除。

后者对 greenthread 进行管理，可以为 greenthread 添加一个计时器。

5.11.1.24 timeutils.py

跟时间相关的一些方法，包括解析 iso 时间，转化为字符串格式，时间比较，获取时间等。

5.11.1.25 uuidutils.py

对 uuid 进行检查和生成。

5.11.1.26 versionutils.py

检查版本号的兼容性。

5.12 plugins

包括实现网络功能的各个插件。

5.12.1 bigswitch

5.12.2 brocade

5.12.3 cisco

5.12.4 common

5.12.5 embrane

5.12.6 hyperv

5.12.7 ibm

5.12.7.1 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。

5.12.7.2 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。

5.12.7.3 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。

5.12.7.4 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。

5.12.8 linuxbridge

5.12.9 metaplugin

5.12.10 midonet

5.12.11 ml2

5.12.11.1 common

5.12.11.2 drivers

5.12.11.3 __init__.py

5.12.11.4 config.py

5.12.11.5 db.py

5.12.11.6 driver_api.py

5.12.11.7 driver_context.py

5.12.11.8 managers.py

两个 manager 类：MechanismManager 和 TypeManager。

5.12.11.9 models.py

数据库模型，包括 DVRPortBinding、NetworkSegment、PortBinding 三种类型，都继承自 neutron.db.model_base.NeutronBaseV2。

5.12.11.10 plugin.py

主要实现类 Ml2Plugin，是 ML2 的 Plugin 端的主类。继承自多个父类。

5.12.11.11 rpc.py

包括两个类：RpcCallbacks 和 AgentNotifierApi。

前者负责当 agent 往 plugin 发出 rpc 请求时候，plugin 实现请求的相关动作，除了继承自父类（dhcp rpc、dvr rpc、sg_db rpc 和 tunnel rpc）中的方法，还包括 get_port_from_device、get_device_details 、 get_devices_details_list 、 update_device_down 、 update_device_up 、get_dvr_mac_address_by_host、get_compute_ports_on_host_by_subnet、get_subnet_for_dvr 等方法。

后者负责当 plugin 往 agent 发出 rpc 请求（plugin 通知 agent）的时候，plugin 端的方法。继承自 dvr、sg、tunnel 等父类。此外还实现了 network_delete、port_update 两个方法。

5.12.12 mlnx

5.12.13 nec

5.12.14 nicira

5.12.15 nuage

5.12.16 ofagent

openflow agent 机制的驱动应用。

5.12.16.1 agent

5.12.16.1.1 ofa_neutron_agent.py

5.12.16.1.2 ports.py

定义了一个 Port 类，表示一个 OF 端口。

5.12.16.2 common

主要包括 config.py，定义了 agent 的配置项，并注册 ovs 的相关配置和 agent 的配置项。

5.12.17 oneconvergence

5.12.18 openvswitch

5.12.18.1 agent

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

ovs_neutron_agent.py 文件 main 函数主要过程如下：

```
def main():
    cfg.CONF.register_opts(ip_lib.OPTS)
    common_config.init(sys.argv[1:])
    common_config.setup_logging(cfg.CONF)
    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（OVSNeutronAgent 类），并调用其 daemon_loop()函数，该函数进一步执行 rpc_loop()。

OVSNeutronAgent 类初始化的时候，会依次调用 setup_rpc()、setup_integration_br()和 setup_physical_bridges()。

ovs_dvr_neutron_agent.py 文件是 neutron 实现分布式路由器设计时的 agent。

5.12.18.1.1 setup_rpc()

setup_rpc()首先创建了两个 rpc 句柄，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，通过 rpc 消息调用 plugin 的方法。这些消息在 neutron.agent.rpc.PluginApi 类中定义，包括 get_device_details、get_devices_details_list、update_device_down、update_device_up、tunnel_sync、security_group_rules_for_devices 等。

后者是 agent 定期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]]
```

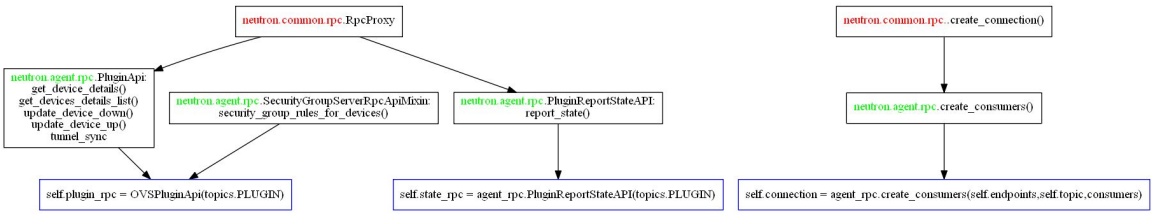
这样，neutron-server 发到这四个主题的消息，会被 agent 接收到。agent 会检查端口是否在本地，如果在本地则进行对应动作。

创建消费者 rpc 连接来接收 rpc 消息：

```
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。

图表 3 展示了这一过程，是一个很好的理解 agent 端的 rpc 实现的例子。



图表 3 Openvswitch 中 agent 端 rpc 的实现

5.12.18.1.2 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。

5.12.18.1.3 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。

5.12.18.2 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。

5.12.18.3 ovs_db_v2.py

跟 ovsdb 打交道的一些函数，包括添加、释放和获取端口、网络绑定信息等。

5.12.18.4 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。

这些模型类供 ovs_db_v2.py 进行使用。

5.12.18.5 ovs_neutron_plugin.py

plugin 的主要实现。

包括三个类：AgentNotifierApi、OVSNeutronPluginV2 和 OVSRpcCallbacks。

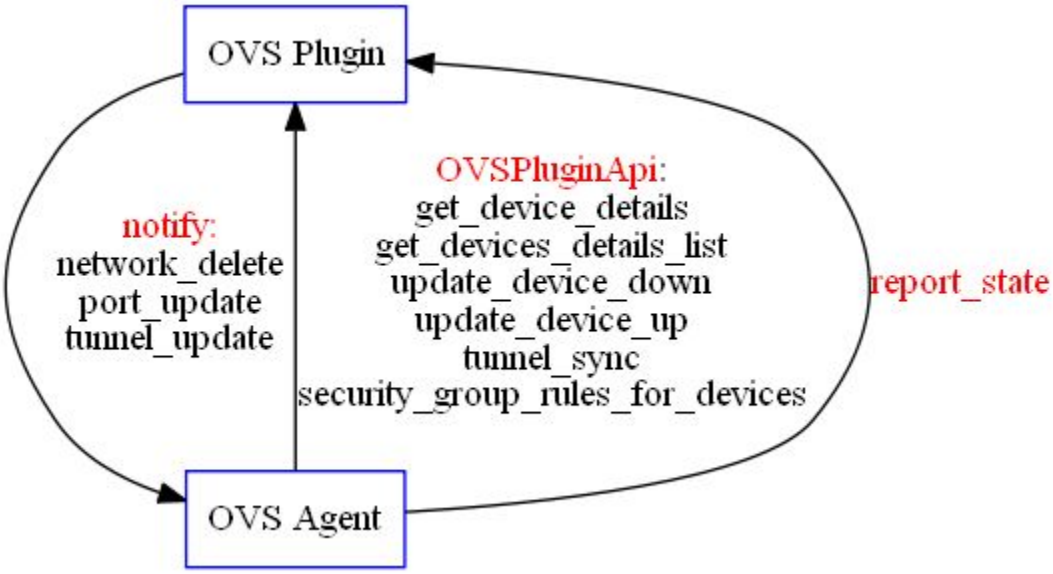
AgentNotifierApi 类，继承自 rpc.RpcProxy 类和 securitygroups_rpc.SecurityGroupAgentRpcApiMixin 类。代表了 openvswitch 往 agent 端发出的事件通知。包括三个函数：network_delete、port_update 和 tunnel_update，分别发出消息到指定主题上，该消息会被 agent 所监听到。

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。

```
def setup_rpc(self):
    # RPC support
    self.service_topics = {svc_constants.CORE: topics.PLUGIN,
                           svc_constants.L3_ROUTER_NAT: topics.L3PLUGIN}
    self.conn = n_rpc.create_connection(new=True)
    self.notifier = AgentNotifierApi(topics.AGENT)
    self.agent_notifiers[q_const.AGENT_TYPE_DHCP] = (
        dhcp_rpc_agent_api.DhcpAgentNotifyAPI()
    )
    self.agent_notifiers[q_const.AGENT_TYPE_L3] = (
        l3_rpc_agent_api.L3AgentNotifyAPI()
    )
    self.endpoints = [OVSRpcCallbacks(self.notifier, self.tunnel_type),
                      agents_db.AgentExtRpcCallback()]
    for svc_topic in self.service_topics.values():
        self.conn.create_consumer(svc_topic, self.endpoints, fanout=False)
    # Consume from all consumers in threads
    self.conn.consume_in_threads()
```

plugin 和 agent 之间的 rpc 关系如图表 4 所示。



图表 4 OpenvSwitch plugin 和 agent 之间的 rpc 消息

5.12.19 plumgrid

5.12.20 ryu

5.12.21 sriovnicagent

5.12.22 vmware

5.13 scheduler

实现调度、负载均衡的算法。

5.13.1 dhcp-agent_scheduler.py

定义了 ChanceScheduler 类，用于实现随机为一个网络分配一个 dhcp agent。

该类主要定义对外的两个方法，schedule()实现对一个网络返回调度给它的若干 agents，auto_schedule_networks()方法将还没有分配 agent 的所有网络安排到指定主机的 agent 上。

5.13.2 l3-agent_scheduler.py

定义了抽象基础类 L3Scheduler，和继承自它的 ChanceScheduler、LeastRoutersScheduler 两种分配机制。

5.14 server

实现 neutron-server 的主进程。

__init__.py 文件中包括一个 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)
```

serve_wsgi 方法创建 NeutronApiService 实例（作为一个 WsgiService），并调用其的 start()来启动 socket 服务器端。

```
#neutron.service
def serve_wsgi(cls):
    try:
        service = cls.create()
```

```
        service.start()
    except Exception:
        with excutils.save_and_reraise_exception():
            LOG.exception(_('Unrecoverable error: please check log '
                            'for details.'))
    return service
```

neutron.service.NeutronApiService 类继承自 neutron.service.WsgiService，其 create 方法返回一个 appname 默认为“neutron”的 WsgiService 对象；start 方法则调用_run_wsgi 方法。

```
def start(self):
    self.wsgi_app = _run_wsgi(self.app_name)
```

_run_wsgi 方法主要是从 api-paste.ini 文件中读取应用（最后是利用 neutron.api.v2.router:APIRouter.factory 来构造应用），然后为应用创建一个 wsgi 的服务端，并启动应用，主要代码为。

```
def _run_wsgi(app_name):
    app = config.load_paste_app(app_name)
    if not app:
        LOG.error(_('No known API applications configured.'))
        return
    server = wsgi.Server("Neutron")
    server.start(app, cfg.CONF.bind_port, cfg.CONF.bind_host,
                 workers=cfg.CONF.api_workers)
    # Dump all option values here after all options are parsed
    cfg.CONF.log_opt_values(LOG, std_logging.DEBUG)
    LOG.info(_("Neutron service started, listening on %(host)s:%(port)s"),
             {'host': cfg.CONF.bind_host,
              'port': cfg.CONF.bind_port})
    return server
```

至此，neutron server 启动完成，之后，需要创建 rpc 服务端。

```
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())
```

这些代码创建 plugin 的 rpc 服务端，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。

图表 5 总结了 neutron-server 的核心启动过程。

neutron.server	*neutron.service #serve_wsgi()*	*neutron.service.NeutronApiService*	*neutron.service.WsgiService*	*neutron.service#_run_wsgi()*
neutron_api = service.serve_wsgi(service.NeutronApiService)	service = cls.create()	service = cls(app_name='neutron')		
	service.start()		self.wsgi_app = _run_wsgi(self.ap	app = config.load_paste_app(a

			p_name)	pp_name)
				server = wsgi.Server("Neutron")
				server.start(app, cfg.CONF.bind_port, cfg.CONF.bind_host, workers=cfg.CONF.api_workers)

图表 5 neutron-server 核心启动过程

5.15 services

5.15.1 service_base.py

定义了抽象基础类 ServicePluginBase，继承自 extensions.PluginInterface 类。为所有的 service plugin 定义基础的接口，包括 get_plugin_type、get_plugin_name、get_plugin_description。

定义了方法 load_drivers，该方法为指定的 service 加载驱动。

5.15.2 provider_configuration.py

主要定义了类 ProviderConfiguration。代表一个 service 的实际提供者的配置信息。

5.15.3 firewall

5.15.3.1 agents

firewall_agent_api.py 中定义了 FWaaSPluginApiMixin 类和 FWaaSAgentRpcCallbackMixin 类。

前者继承自 rpc.RpcProxy，是 agent 往 plugin 发出 rpc 消息时候，为 agent 一端使用的方法。包括 set_firewall_status()方法和 firewall_deleted()方法。

后者为 agent 的实现提供 mixin，分别声明了 create_firewall、update_firewall、delete_firewall 三个接口，这三个接口用于处理 plugin 发出的对应 rpc 调用请求。

5.15.3.2 drivers

包括对 linux 防火墙和 varmour 防火墙进行操作的驱动类。

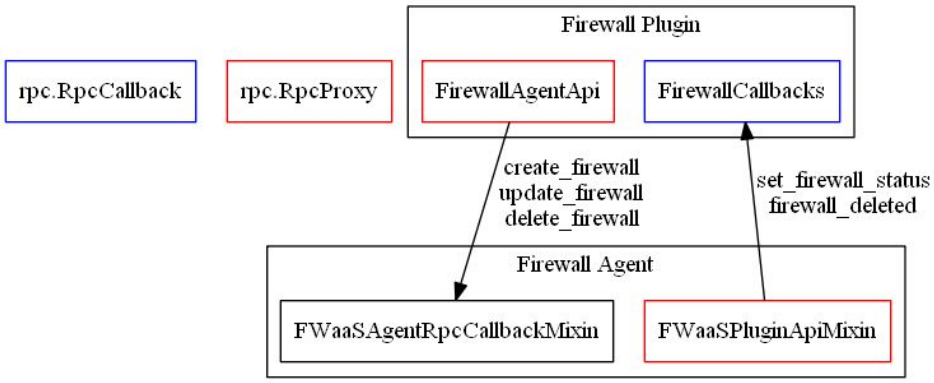
5.15.3.3 fwaas_plugin.py

FirewallCallbacks 类，继承自 rpc.RpcCallback，agent 利用 rpc 来调用该类的方法实现设置防火墙的状态、通知防火墙被删除、获取 tenant 所拥有的防火墙和规则、获取防火墙所属的租户们的信息。

FirewallAgentApi 类，继承自 rpc.RpcProxy 类，是 plugin 向 agent 端发送 rpc 调用时候，为 plugin 端使用的方法，包括对防火墙的创建、更新和删除。

FirewallPlugin 类，继承自 firewall_db.Firewall_db_mixin 类，是防火墙这个服务 plugin 的实现类，其中定义了一系列对防火墙进行操作的方法，包括 create_firewall、update_firewall、delete_db_firewall_object、delete_firewall、update_firewall_policy、update_firewall_rule、insert_rule、remove_rule 等。

plugin 和 agent 之间的调用关系如图表 6 所示。



图表 6 防火墙服务 plugin 和 agent 之间的 rpc 消息

5.15.4 l3_router

5.15.4.1 l3_apic.py

定义类 ApicL3ServicePlugin，是 Cisco APIC （Application Policy Infrastructure Controller）l3 路由器的服务插件主类。

5.15.4.2 l3_router_plugin.py

L3RouterPlugin 类，是 Neutron L3 路由器服务插件的实现主类。

5.15.5 loadbalancer

与其他服务实现类似，多了 agent 的 scheduler。

5.15.5.1 agent

5.15.5.2 drivers

5.15.5.3 agent_scheduler.py

loadbalancer 池和 agent 之间的绑定和调度维护。

5.15.5.4 constants.py

常量定义。

5.15.5.5 plugin.py

定义了该服务 plugin 的实现主类 LoadBalancerPlugin。

5.15.6 metering

5.15.6.1 agents

5.15.6.2 drivers

5.15.6.3 metering_plugin.py

定义了 metering 服务 plugin 的实现主类 MeteringPlugin。

5.15.7 vpn

5.15.7.1 common

5.15.7.2 device_drivers

5.15.7.3 service_drivers

5.15.7.4 agent.py

5.15.7.5 plugin.py

5.16 tests

5.17 其他文件

5.17.1 auth.py

定义了类 NeutronKeystoneContext，可以利用 keystone 的头部信息来生成一次请求的上下文。

定义了方法 pipeline_factory，是一个流水线处理，对给定的 auth_strategy（例如 filter1 filter2 filter3 ... app），逆序调用各个过滤器对 app 进行处理，并返回最终结果。

5.17.2 context.py

定义继承自 neutron.openstack.common.context.RequestContext 的基础类 ContextBase。以及继承自它的 Context。表示安全上下文和请求信息，用于代表执行给定操作的用户。

5.17.3 hooks.py

定义方法 setup_hook，对给定的配置进行处理，检查平台添加必要的信息。

5.17.4 manager.py

定义了类 Manager 和类 NeutronManager。

前者继承自 neutron.common.rpc.RpcCallback 类和 neutron.openstack.common.periodic_task.PeriodicTasks 类。该类会定义运行任务。

后者负责解析配置文件并初始化 neutron 的 plugin。

5.17.5 neutron_plugin_base_v2.py

Neutron plugin 的抽象基础类，是实现 plugin 的参考和基础，它定义了实现一个 neutron plugin 所需的基本接口。

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	

5.17.6 policy.py

5.17.7 quota.py

5.17.8 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。

5.17.8.1 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。

5.17.9 version.py

从 prb 中获取 version 信息。

```
version_info = pbr.version.VersionInfo('neutron')
```

5.17.10 wsgi.py

WorkerService 类，封装被 ProcessLauncher 处理的一个工作者。

Server 类，管理多个 WSGI socket 和应用。

Middleware 类，封装基本的 WSGI 中间件。

Request 类，继承自 webob.Request 类，代表请求信息。

ActionDispatcher 类，通过行动名将方法名映射到本地方法。

Application 类，是 WSGI 应用的封装基础类。

Debug 类，用于进行 debug 的中间件。

Router，WSGI 中间件，将到达的请求发送给应用。

Resource 类，继承自 Application 类，是 wsgi 应用，用于处理序列化和控制器分发。

Controller 类，是一个 WSGI 的应用，根据请求，调用响应的方法。

第 6 章　tools

包括跟安装和格式检查相关的一些工具。

6.1 国际化检查

check_i18n.py，检查国际化。

check_i18n_test_case.txt，是 check_i18n.py 进行国际化格式检查的用例

i18n_cfg.py

6.2 虚环境

install_venv.py，neutron 开发的时候，可能需要一套虚环境，该脚本可以安装一个 python 虚环境。

install_venv_common.py，为 install_venv.py 提供必要的方法。

with_venv.sh，启用虚环境。

6.3 清理

clean.sh，清除代码编译中间结果。

第 7 章　理解代码

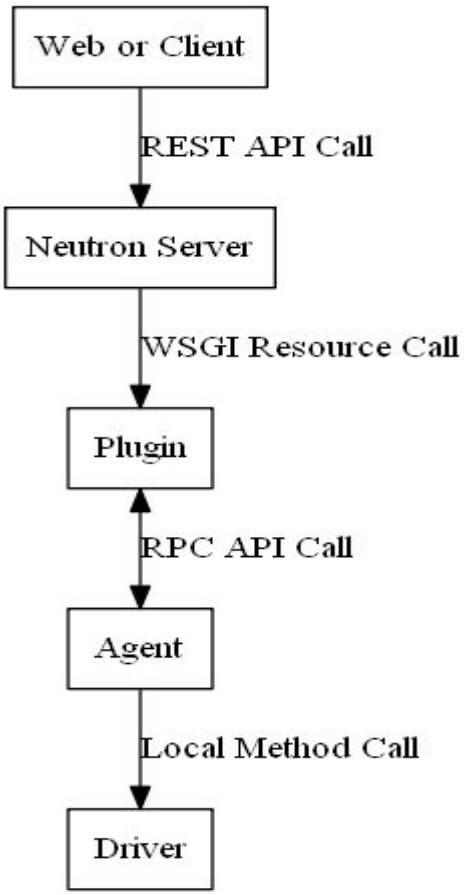
本部分试图从专题和业务流程的角度来剖析 neutron 代码，以便理解如此设计的内涵。

7.1 调用逻辑

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图表 7 所示。



图表 7 neutron 自顶向下的调用框架

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。

7.2 Rest API 专题

neutron-server（相当于 REST API Server）的核心作用之一就是要实现 REST API。

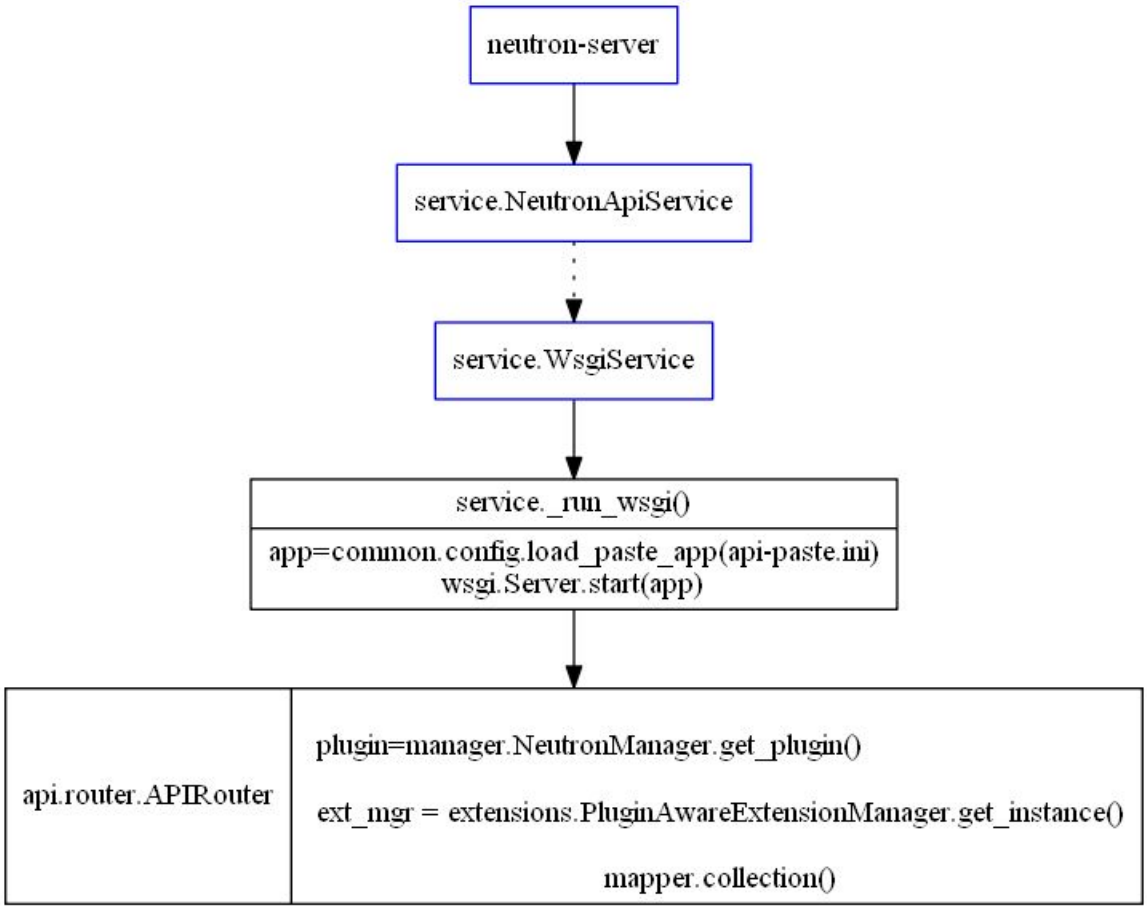
在 Neutron 中，neutron-server（相当于 REST API Server）负责将收到的 REST API 请求交由 Plugin 来进行相关处理。可以看出，这其实就是一个 web 服务器要完成的事情，将 http 请求转化为对资源的操作（通过 plugin 的方法调用），并返回响应。

REST API 可以分为两类：标准 API 和扩展 API。

前者主要是由原先的 nova-network 项目沿用而来，实现对网络二层的支持，核心资源只有网络、端口和子网；后者则通过进行扩展，提供更多的网络服务。目前已有的扩展有 L3（router）,L4(TPC/udp firewall)及 L7(http/https load balancer)。随着 neutron 项目的不断成熟，扩展 API 会演化为标准 API。

这个过程的主要涉及的模块包括

neutron.server、service 模块，以及定义了 neutron API 的 neutron.api 包。



图表 8 REST API 实现逻辑

以 neutron.api.v2 包为例，其中 base.py 中定义了 Controller 类，是实现 URL 到 plugin 的 api 进行 mapping 的核心。

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

顾名思义，create 用于创建资源，delete 用于删除资源，show 是获取资源、update 是更新资源，index 是返回请求资源的列表。

以 create 方法为例来看主要过程。

向 network 的 notifier 发送一个资源 create.start 的通知；然后进行 policy 的检查；之后调用 plugin 中相应的方法进行处理，最后向 network 的 notifier 发出一个资源 create.end 的通知。

7.3 RPC 专题

RPC 是 neutron 中跨模块进行方法调用的很重要的一种方式，主要包括 client 端和 server 端。client 端用于发出 rpc 消息，server 端用于监听消息并进行相应处理。

7.3.1 agent 端的 rpc

在 dhcp agent、l3 agent、firewall agent 以及 metering agent 的 main 函数中都能找到类似的创建一个 rpc 服务端的代码，以 dhcp agent 为例。

```
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()
```

最核心的，也是跟 rpc 相关的部分包括两部分，首先是创建 rpc 服务端。

```
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')
```

该代码实际上创建了一个 rpc 服务端，监听指定的 topic 并定期的运行 manager 上的定期任务。

create() 方法返回一个 neutron.service.Service 对象，neutron.service.Service 继承自 neutron.common.rpc.Service 类。

首先看 neutron.common.rpc.Service 类，该类定义了 start 方法，该方法主要完成两件事情：一件事情是将 manager 添加到 endpoints 中；一件是创建了三个 rpc 的 consumer，分别监听 topic、topic.host 和 fanout 的队列消息。

而在 neutron.service.Service 类中，初始化中生成了一个 manager 实例（即 neutron.agent.dhcp_agent.DhcpAgentWithStateReport）；并为 start 方法添加了周期性执行 report_state 方法和 periodic_tasks 方法。report_state 方法实际上什么都没做，periodic_tasks 方法则调用 manager 的 periodic_tasks 方法。

manager 实例（即 neutron.agent.dhcp_agent.DhcpAgentWithStateReport）在初始化的时候首先创建一个 rpc 的 client 端，通过代码

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

该 client 端实际上定义了 report_state 方法，可以状态以 rpc 消息的方式发送给 plugin。

manager 在初始化后，还会指定周期性运行_report_state 方法，实际上就是调用 client 端的 report_state 方法。

至此，对 rpc 服务端的创建算是完成了，之后执行代码。

```
service.launch(server).wait()
```

service.launch(server)方法首先会将 server 放到协程组中，并调用 server 的 start 方法来启动 server。

7.3.2 plugin 端的 rpc

以 openvswitch 的 plugin 为例进行分析。

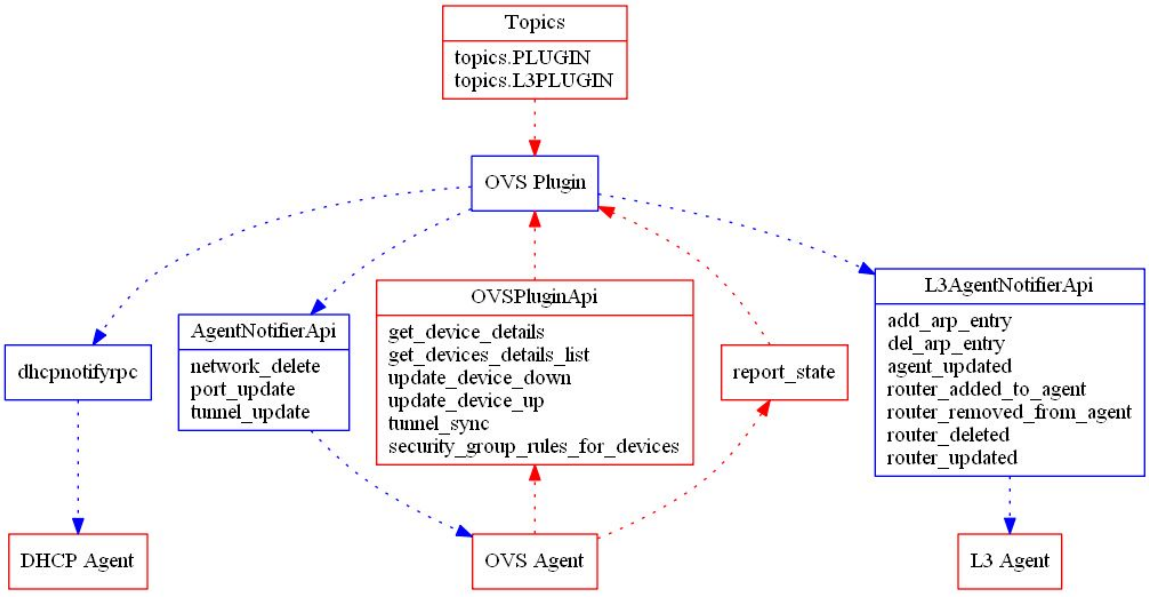
neutron.plugin.openvswitch.ovs_neutron_plugin.OVSNeutronPluginV2 类在初始化的时候调用了 self.setup_rpc 方法。其代码为

```
def setup_rpc(self):
    # RPC support
    self.service_topics = {svc_constants.CORE: topics.PLUGIN,
                           svc_constants.L3_ROUTER_NAT: topics.L3PLUGIN}
    self.conn = n_rpc.create_connection(new=True)
    self.notifier = AgentNotifierApi(topics.AGENT)
    self.agent_notifiers[q_const.AGENT_TYPE_DHCP] = (
        dhcp_rpc_agent_api.DhcpAgentNotifyAPI()
    )
    self.agent_notifiers[q_const.AGENT_TYPE_L3] = (
        l3_rpc_agent_api.L3AgentNotifyAPI()
    )
    self.endpoints = [OVSRpcCallbacks(self.notifier, self.tunnel_type),
                      agents_db.AgentExtRpcCallback()]
    for svc_topic in self.service_topics.values():
        self.conn.create_consumer(svc_topic, self.endpoints, fanout=False)
    # Consume from all consumers in threads
    self.conn.consume_in_threads()
```

创建一个通知 rpc 的客户端，用于向 l2 的 agent 发出通知。所有 plugin 都需要有这样一个发出通知消息的客户端。

分别创建了一个 dhcp agent 和 l3 agent 的通知 rpc 客户端。

之后，创建两个跟 service agent 相关的 consumer，分别监听 topics.PLUGIN 和 topics.L3PLUGIN 两个主题。



图表 9 plugin 端的 rpc

7.3.3 neutron-server 的 rpc

这个 rpc 服务端主要通过 neutron.server 中主函数中代码执行

```
neutron_rpc = service.serve_rpc()
```

方法的实现代码如下

```
def serve_rpc():
    plugin = manager.NeutronManager.get_plugin()

    # If 0 < rpc_workers then start_rpc_listeners would be called in a
    # subprocess and we cannot simply catch the NotImplementedError.  It is
    # simpler to check this up front by testing whether the plugin supports
    # multiple RPC workers.
    if not plugin.rpc_workers_supported():
        LOG.debug(_("Active plugin doesn't implement start_rpc_listeners"))
        if 0 < cfg.CONF.rpc_workers:
            msg = _("'rpc_workers = %d' ignored because start_rpc_listeners "
                    "is not implemented.")
            LOG.error(msg, cfg.CONF.rpc_workers)
        raise NotImplementedError

    try:
        rpc = RpcWorker(plugin)

        if cfg.CONF.rpc_workers < 1:
            rpc.start()
            return rpc
        else:
            launcher = common_service.ProcessLauncher(wait_interval=1.0)
            launcher.launch_service(rpc, workers=cfg.CONF.rpc_workers)
            return launcher
    except Exception:
        with excutils.save_and_reraise_exception():
            LOG.exception(_('Unrecoverable error: please check log '
                            'for details.'))
```

其中，RpcWorker(plugin)主要通过调用 plugin 的方法来创建 rpc 服务端。

```
self._servers = self._plugin.start_rpc_listeners()
```

该方法在大多数 plugin 中并未被实现，目前 ml2 支持该方法。

在 neutron.plugin.ml2.plugin.ML2Plugin 类中，该方法创建了一个 topic 为 topics.PLUGIN 的消费 rpc。

```
def start_rpc_listeners(self):
        self.endpoints = [rpc.RpcCallbacks(self.notifier, self.type_manager),
                          agents_db.AgentExtRpcCallback()]
        self.topic = topics.PLUGIN
        self.conn = n_rpc.create_connection(new=True)
        self.conn.create_consumer(self.topic, self.endpoints,
                                  fanout=False)
        return self.conn.consume_in_threads()
```

7.4 Plugin 专题

plugin 实现对 REST API 请求的后端支持。前端的 rest 框架会调用 plugin 的相应方法来实现 rest api 规定的语义。

plugin 包括两种类型：核心 plugin 和服务 plugin。

核心 Plugin 实现了对标准 API 的支持（对网络、端口和子网资源的相应操作），此外还可以选择性的支持一些扩展的 API（_supported_extension_aliases 属性中会指出）。

以 neutron 的 openvswitch plugin 为例，支持的扩展 API 包括

```
_supported_extension_aliases = ["provider", "external-net", "router",
                                "ext-gw-mode", "binding", "quotas",
                                "security-group", "agent", "extraroute",
                                "l3_agent_scheduler",
                                "dhcp_agent_scheduler",
                                "extra_dhcp_opt",
                                "allowed-address-pairs"]
```

服务 plugin 则专门用于实现扩展 API。目前，路由器、防火墙、vpn 等都有相应的 service plugin 来专门实现。

neturon server 启动时候，根据配置文件动态加载相应的核心 plugin 及服务 plugins。

目前，neutron-server 只能配置一个核心 Plugin，但可以配置多个服务 plugin（但每个 extension 的实现 plugin 不能超过一个）。但正在进行中的 ML2 项目会打破这个限制，ML2 core plugin 允许同时加载多个 mechanism driver，可以达到同时使用不同的 L2 实现技术的效果。

7.5 Extension 专题

Neutron 中的扩展，主要用于实现原先标准 API 中并不支持的扩展的网络服务，包括路由器、防火墙、vpn 等等。

扩展包括三种类型：资源扩展、行动扩展和请求扩展。

资源扩展意味着在网络中引入新的资源和相应的属性；行动扩展是为标准的 API 控制器添加行动支持；请求扩展是为 API 控制器提供额外的请求和响应支持。

7.6 Agent 专题

agent 一般作为 plugin 的后端，接收远端 plugin 的命令来具体实施对本地网络资源的操作，并同时汇报需要的信息给远端的 plugin。

agent 可以分为四种：核心 agent、dhcp agent、l3 agent、其他 agent。

核心 agent 用于服务核心 plugin，提供 L2 功能，所以又被称为 L2 agent。该 agent 在所有的计算和网络节点均存在。

dhcp agent 用于提供对虚机的 dhcp 管理和分配。核心 plugin 在实现上通过继承 DhcpAgentSchedulerDbMixin 来实现对 dhcp agent 的操作。dhcp agent 可以安装在任何节点上，并支持可以配置多个实例，还支持配置的的调度策略。

l3 agent 用于提供对路由器的管理（配置 iptables 规则）。l3 agent 也可以安装在任何节点上，并支持可以配置多个实例，还支持配置的的调度策略。

其他 agent，用于实现其它的网络服务等，包括 metadata、metering 等。

从 agent 的结构上来看，主要包括两部分：跟 plugin 之间的 rpc 消息处理，以及本地的操作。前者主要用于跟 plugin 进行交互，接收 plugin 命令、调用 plugin 方法、汇报状态等；后者用于对本地资源进行实际的操作，包括配置 iptables 表项，配置 dhcp 服务等。