

OpenStack 源码分析之 Neutron

yeasy

Published
with GitBook



目錄

1. [前言](#)
2. [整体结构](#)
3. [bin](#)
4. [doc](#)
5. [etc](#)
 - i. [init.d/](#)
 - ii. [neutron/](#)
 - iii. [api-paste.ini](#)
 - iv. [dhcp_agent.ini](#)
 - v. [fwaas_driver.ini](#)
 - vi. [l3_agent.ini](#)
 - vii. [lbaas_agent.ini](#)
 - viii. [metadata_agent.ini](#)
 - ix. [metering_agent.ini](#)
 - x. [vpn_agent.ini](#)
 - xi. [neutron.conf](#)
 - xii. [policy.json](#)
 - xiii. [rootwrap.conf](#)
 - xiv. [services.conf](#)
6. [neutron](#)
 - i. [agent](#)
 - i. [common/](#)
 - ii. [linux/](#)
 - iii. [metadata/](#)
 - iv. [dhcp_agent.py](#)
 - v. [firewall.py](#)
 - vi. [l2population_rpc.py](#)
 - vii. [l3_agent.py](#)
 - viii. [l3_ha_agent.py](#)
 - ix. [netns_cleanup_util.py](#)
 - x. [ovs_cleanup_util.py](#)
 - xi. [rpc.py](#)
 - xii. [securitygroups_rpc.py](#)
 - ii. [api](#)
 - i. [rpc](#)
 - ii. [v2](#)
 - iii. [views](#)
 - iv. [api_common.py](#)
 - v. [extensions.py](#)
 - vi. [versions.py](#)
 - iii. [cmd](#)
 - iv. [common](#)

- i. [config.py](#)
- ii. [constants.py](#)
- iii. [exceptions.py](#)
- iv. [ipv6_utils.py](#)
- v. [log.py](#)
- vi. [rpc.py](#)
- vii. [test_lib.py](#)
- viii. [topics.py](#)
- ix. [utils.py](#)
- v. **db**
 - i. [agents_db.py](#)
 - ii. [agentschedulers_db.py](#)
 - iii. [api.py](#)
 - iv. [common_db_mixin.py](#)
 - v. [db_base_plugin_v2.py](#)
 - vi. [migration](#)
 - vii. [model_base.py](#)
 - viii. [models_v2.py](#)
 - ix. [securitygroups_rpc_base.py](#)
 - x. [sqlalchemyutils.py](#)
 - xi. **扩展资源和操作类**
 - i. [allowedaddresspairs_db.py](#)
 - ii. [dvr_mac_db.py](#)
 - iii. [external_net_db.py](#)
 - iv. [extradhcpopt_db.py](#)
 - v. [extraroute_db.py](#)
 - vi. [firewall](#)
 - vii. [l3_agentschedulers_db.py](#)
 - viii. [l3_attrs_db.py](#)
 - ix. [l3_db.py](#)
 - x. [l3_dvr_db.py](#)
 - xi. [l3_dvrscheduler_db.py](#)
 - xii. [l3_gwmode_db.py](#)
 - xiii. [l3_hamode_db.py](#)
 - xiv. [l3_hascheduler_db.py](#)
 - xv. [loadbalancer](#)
 - xvi. [metering](#)
 - xvii. [portbindings_base.py](#)
 - xviii. [portbindings_db.py](#)
 - xix. [portsecurity_db.py](#)
 - xx. [quota_db.py](#)
 - xxi. [routedserviceinsertion_db.py](#)
 - xxii. [routerservicetype_db.py](#)
 - xxiii. [securitygroups_db.py](#)
 - xxiv. [servicetype_db.py](#)
 - xxv. [vpn](#)

- vi. [debug](#)
 - i. [commands.py](#)
 - ii. [debug_agent.py](#)
 - iii. [shell.py](#)
- vii. [extensions](#)
 - i. [agent.py](#)
 - ii. [allowedaddresspairs.py](#)
 - iii. [dhcpagentscheduler.py](#)
 - iv. [dvr.py](#)
 - v. [external_net.py](#)
 - vi. [extraroute.py](#)
 - vii. [extra_dhcp_opt.py](#)
 - viii. [firewall.py](#)
 - ix. [flavor.py](#)
 - x. [l3.py](#)
 - xi. [l3agentscheduler.py](#)
 - xii. [l3_ext_gw_mode.py](#)
 - xiii. [l3_ext_ha_mode.py](#)
 - xiv. [lbaas_agentscheduler.py](#)
 - xv. [loadbalancer.py](#)
 - xvi. [metering.py](#)
 - xvii. [multiprovidernet.py](#)
 - xviii. [portbindings.py](#)
 - xix. [portsecurity.py](#)
 - xx. [providernet.py](#)
 - xxi. [quotasv2.py](#)
 - xxii. [routedserviceinsertion.py](#)
 - xxiii. [routerservicetype.py](#)
 - xxiv. [securitygroup.py](#)
 - xxv. [servicetype.py](#)
 - xxvi. [vpnaas.py](#)
 - xxvii. [__init__.py](#)
- viii. [hacking](#)
 - i. [checks.py](#)
 - ii. [__init__.py](#)
- ix. [locale](#)
- x. [notifiers](#)
 - i. [nova.py](#)
 - ii. [__init__.py](#)
- xi. [openstack](#)
 - i. [common](#)
 - i. [cache](#)
 - ii. [context.py](#)
 - iii. [eventlet_backdoor.py](#)
 - iv. [fileutils.py](#)
 - v. [fixture](#)

- vi. [local.py](#)
- vii. [lockutils.py](#)
- viii. [log.py](#)
- ix. [loopingcall.py](#)
- x. [middleware](#)
- xi. [periodic_task.py](#)
- xii. [policy.py](#)
- xiii. [processutils.py](#)
- xiv. [service.py](#)
- xv. [systemd.py](#)
- xvi. [threadgroup.py](#)
- xvii. [uuidutils.py](#)
- xviii. [versionutils.py](#)
- xix. [_i18n.py](#)
- xii. [plugins](#)
 - i. [bigswitch](#)
 - i. [agent](#)
 - ii. [config.py](#)
 - iii. [db](#)
 - iv. [extensions](#)
 - v. [l3_router_plugin.py](#)
 - vi. [plugin.py](#)
 - vii. [routerrule_db.py](#)
 - viii. [servermanager.py](#)
 - ix. [tests](#)
 - x. [vcsversion.py](#)
 - xi. [version.py](#)
 - xii. [__init__.py](#)
 - ii. [brocade](#)
 - i. [db](#)
 - ii. [NeutronPlugin.py](#)
 - iii. [nos](#)
 - iv. [tests](#)
 - v. [vlanbm.py](#)
 - vi. [__init__.py](#)
 - iii. [cisco](#)
 - i. [cfg_agent](#)
 - ii. [common](#)
 - iii. [db](#)
 - iv. [extensions](#)
 - v. [l2device_plugin_base.py](#)
 - vi. [l3](#)
 - vii. [models](#)
 - viii. [n1kv](#)
 - ix. [network_plugin.py](#)
 - x. [service_plugins](#)

- xi. `__init__.py`
- iv. `common`
 - i. `constants.py`
 - ii. `utils.py`
 - iii. `__init__.py`
- v. `embrane`
 - i. `agent`
 - ii. `base_plugin.py`
 - iii. `common`
 - iv. `l2base`
 - v. `plugins`
 - vi. `__init__.py`
- vi. `hyperv`
 - i. `agent`
 - ii. `agent_notifier_api.py`
 - iii. `common`
 - iv. `db.py`
 - v. `hyperv_neutron_plugin.py`
 - vi. `model.py`
 - vii. `rpc_callbacks.py`
 - viii. `__init__.py`
- vii. `ibm`
 - i. `agent`
 - ii. `common`
 - iii. `sdnve_api.py`
 - iv. `sdnve_api_fake.py`
 - v. `sdnve_neutron_plugin.py`
- viii. `linuxbridge`
 - i. `agent`
 - ii. `common`
 - iii. `db`
 - iv. `__init__.py`
- ix. `metaplugin`
 - i. `common`
 - ii. `meta_db_v2.py`
 - iii. `meta_models_v2.py`
 - iv. `meta_neutron_plugin.py`
 - v. `proxy_neutron_plugin.py`
 - vi. `__init__.py`
- x. `midonet`
 - i. `agent`
 - ii. `common`
 - iii. `midonet_lib.py`
 - iv. `plugin.py`
 - v. `__init__.py`
- xi. `ml2`

- i. [common](#)
 - ii. [config.py](#)
 - iii. [db.py](#)
 - iv. [drivers](#)
 - v. [driver_api.py](#)
 - vi. [driver_context.py](#)
 - vii. [managers.py](#)
 - viii. [models.py](#)
 - ix. [plugin.py](#)
 - x. [rpc.py](#)
- xii. [mlnx](#)
 - i. [agent](#)
 - ii. [agent_notify_api.py](#)
 - iii. [common](#)
 - iv. [db](#)
 - v. [mlnx_plugin.py](#)
 - vi. [rpc_callbacks.py](#)
 - vii. [__init__.py](#)
- xiii. [nec](#)
 - i. [agent](#)
 - ii. [common](#)
 - iii. [db](#)
 - iv. [drivers](#)
 - v. [extensions](#)
 - vi. [nec_plugin.py](#)
 - vii. [nec_router.py](#)
 - viii. [ofc_driver_base.py](#)
 - ix. [ofc_manager.py](#)
 - x. [packet_filter.py](#)
 - xi. [router_drivers.py](#)
 - xii. [__init__.py](#)
- xiv. [nuage](#)
 - i. [common](#)
 - ii. [extensions](#)
 - iii. [nuagedb.py](#)
 - iv. [nuage_models.py](#)
 - v. [plugin.py](#)
 - vi. [syncmanager.py](#)
 - vii. [__init__.py](#)
- xv. [ofagent](#)
 - i. [agent](#)
 - ii. [common](#)
- xvi. [oneconvergence](#)
 - i. [agent](#)
 - ii. [lib](#)
 - iii. [plugin.py](#)

- iv. `__init__.py`
- xvii. `opencontrail`
 - i. `common`
 - ii. `contrail_plugin.py`
 - iii. `__init__.py`
- xviii. `openvswitch`
 - i. `agent`
 - ii. `common`
 - iii. `ovs_models_v2.py`
 - iv. `__init__.py`
- xix. `plumgrid`
 - i. `common`
 - ii. `drivers`
 - iii. `plumgrid_plugin`
 - iv. `__init__.py`
- xx. `sriovnicagent`
 - i. `common`
 - ii. `eswitch_manager.py`
 - iii. `pci_lib.py`
 - iv. `sriov_nic_agent.py`
 - v. `__init__.py`
- xxi. `vmware`
 - i. `api_client`
 - ii. `check_nsx_config.py`
 - iii. `common`
 - iv. `dbexts`
 - v. `dhcpmeta_modes.py`
 - vi. `dhcp_meta`
 - vii. `extensions`
 - viii. `nsxlib`
 - ix. `nsx_cluster.py`
 - x. `plugin.py`
 - xi. `plugins`
 - xii. `shell`
 - xiii. `vshield`
 - xiv. `__init__.py`
- xiii. `scheduler`
 - i. `dhcp_agent_scheduler.py`
 - ii. `l3_agent_scheduler.py`
- xiv. `server`
- xv. `service.py`
- xvi. `services`
 - i. `firewall`
 - i. `agents`
 - ii. `drivers`
 - iii. `fwaas_plugin.py`

- ii. [l3_router](#)
 - i. [brocade](#)
 - ii. [l3_apic.py](#)
 - iii. [l3_arista.py](#)
 - iv. [l3_router_plugin.py](#)
 - v. [__init__.py](#)
- iii. [loadbalancer](#)
 - i. [agent](#)
 - ii. [agent_scheduler.py](#)
 - iii. [constants.py](#)
 - iv. [drivers](#)
 - v. [plugin.py](#)
- iv. [metering](#)
 - i. [agents](#)
 - ii. [drivers](#)
 - iii. [metering_plugin.py](#)
- v. [provider_configuration.py](#)
- vi. [service_base.py](#)
- vii. [vpn](#)
 - i. [agent.py](#)
 - ii. [common](#)
 - iii. [device_drivers](#)
 - iv. [plugin.py](#)
 - v. [service_drivers](#)
 - vi. [__init__.py](#)
- viii. [__init__.py](#)
- xvii. [tests](#)
 - i. [base.py](#)
 - ii. [common](#)
 - i. [agents](#)
 - ii. [__init__.py](#)
 - iii. [etc](#)
 - i. [rootwrap.d](#)
 - iv. [fake_notifier.py](#)
 - v. [functional](#)
 - i. [agent](#)
 - ii. [base.py](#)
 - iii. [contrib](#)
 - iv. [db](#)
 - v. [sanity](#)
 - vi. [__init__.py](#)
 - vi. [post_mortem_debug.py](#)
 - vii. [tools.py](#)
 - viii. [unit](#)
 - ix. [var](#)
 - x. [__init__.py](#)

- xviii. [auth.py](#)
- xix. [context.py](#)
- xx. [hooks.py](#)
- xxi. [i18n.py](#)
- xxii. [manager.py](#)
- xxiii. [neutron_plugin_base_v2.py](#)
- xxiv. [policy.py](#)
- xxv. [quota.py](#)
- xxvi. [service.py](#)
- xxvii. [version.py](#)
- xxviii. [wsgi.py](#)
- 7. [rally-jobs](#)
 - i. [extra](#)
 - i. [README.rst](#)
 - ii. [plugins](#)
 - i. [README.rst](#)
 - ii. [__init__.py](#)
 - iii. [neutron-neutron.yaml](#)
 - iv. [README.rst](#)
- 8. [tools](#)
 - i. [check_bash.sh](#)
 - ii. [check_i18n.py](#)
 - iii. [check_i18n_test_case.txt](#)
 - iv. [clean.sh](#)
 - v. [i18n_cfg.py](#)
 - vi. [install_venv.py](#)
 - vii. [install_venv_common.py](#)
 - viii. [pretty_tox.sh](#)
 - ix. [with_venv.sh](#)
- 9. [理解代码](#)
 - i. [调用逻辑](#)
 - ii. [REST API 专题](#)
 - iii. [RPC 专题](#)
 - i. [agent RPC](#)
 - ii. [plugin RPC](#)
 - iii. [neutron-server RPC](#)
 - iv. [Plugin 专题](#)
 - v. [Extension 专题](#)
 - vi. [Agent 专题](#)
 - vii. [Driver 专题](#)

OpenStack Neutron 源码分析

Neutron 是 OpenStack 项目中负责提供网络服务的组件，它基于软件定义网络的思想，实现了网络虚拟化下的资源管理。

本书将剖析 Neutron 组件的代码。

最新版本在线阅读：[GitBook](#)。

本书源码在 Github 上维护，欢迎参与：https://github.com/yeasy/openstack_code_Neutron。

感谢所有的 [贡献者](#)。

更新历史:

- V0.8: 2015-02-03
 - 按照最新版本进行更新
 - 添加更多服务的实现分析。
- 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。

参加步骤

- 在 GitHub 上 `fork` 到自己的仓库，如 `user/openstack_code_Neutron`，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:user/openstack_code_Neutron.git
$ cd openstack_code_Neutron
$ git config user.name "User"
```

```
$ git config user.email user@email.com
```

- 修改代码后提交，并推送到自己的仓库。

```
$ #do some change on the content  
$ git commit -am "Fix issue #1: change helo to hello"  
$ git push
```

- 在 GitHub 网站上提交 pull request。
- 定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/openstack_code_Neutron  
$ git fetch upstream  
$ git checkout master  
$ git rebase upstream/master  
$ git push -f origin master
```

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。

bin

主要包括 neutron-rootwrap、neutron-rootwrap-xen-dom0 两个文件。提供一些可执行命令。

doc

包括文档生成的相关源码。

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/：核心的库代码。
- 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。

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。

tools

一些相关的代码格式化检测、环境安装的脚本。

其它文档

- README.rst：介绍了项目的情况和连接。
- TESTING.rst：介绍如何进行开发后的测试。官方配置的 jenkins 当 gerrit 上有代码提交 review 的时候会触发 tox 测试。实际上，OpenStack 中的项目使用 [tox](#) 来管理测试的虚拟环境，使用 [testr](#) 来管理运

行测试案例的顺序。

- `run_tests.sh`：自动创建一个虚拟环境并进行测试。命令为 `./run_tests -v`。

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。

doc

可以利用 sphinx 工具来生成文档。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。

- source 子目录：文档相关的代码。
- Makefile：用户执行 make 命令的模板文件。
- pom.xml：maven 项目管理文件。

etc

存放 Neutron 服务运行相关的配置文件。

init.d/

neutron-server 是 Upstart 支持的系统服务脚本，核心部分为

```
case "$1" in
  start)
    test "$ENABLED" = "true" || exit 0
    log_daemon_msg "Starting neutron server" "neutron-server"
    start-stop-daemon -SbmV --pidfile $PIDFILE --chdir $DAEMON_DIR --exec $DAEMON -- $DAEMON
    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
    ;;
esac
```

neutron/

plugins/

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

rootwrap.d

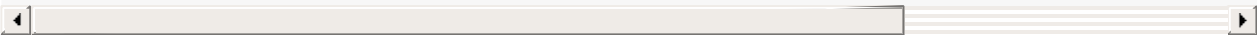
包括一系列的filter文件。包括debug.filters rootwrap是实现让非特权用户以root权限去运行某些命令。这些命令就在filter中指定。以neutron用户为例，在/etc/sudoers.d/neutron文件中有

```
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/rootwrap.conf中定义了

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



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。

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 = oslo.middleware:RequestId.factory

[filter:catch_errors]
paste.filter_factory = oslo.middleware:CatchErrors.factory

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

[filter:authtoken]
paste.filter_factory = keystonemiddleware.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

```
class APIRouter(wsgi.Router):

    @classmethod
    def factory(cls, global_config, **local_config):
        return cls(**local_config)

    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"])
```

neutron server 启动后，根据配置文件动态加载对应的 core plugin 和 service plugin。neutron 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。

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。

需要注意的是，这些 ini 文件内容多为注释掉的默认值的情况，是自动从代码中提取的。

```
hcp.Dnsmasq

# Allow overlapping IP (Must have kernel build with CONFIG_NET_NS=y and
# iproute2 package that supports namespaces).
# use_namespaces = True

# The DHCP server can assist with providing metadata support on isolated
# networks. Setting this value to True will cause the DHCP server to append
# specific host routes to the DHCP request. The metadata service will only
# be activated when the subnet does not contain any router port. The guest
# instance must be configured to request host routes via DHCP (Option 121).
# enable_isolated_metadata = False

# Allows for serving metadata requests coming from a dedicated metadata
# access network whose cidr is 169.254.169.254/16 (or larger prefix), and
# is connected to a Neutron router from which the VMs send metadata
# request. In this case DHCP Option 121 will not be injected in VMs, as
# they will be able to reach 169.254.169.254 through a router.
# This option requires enable_isolated_metadata = True
# enable_metadata_network = False

# Number of threads to use during sync process. Should not exceed connection
# pool size configured on server.
# num_sync_threads = 4

# Location to store DHCP server config files
# dhcp_confs = $state_path/dhcp

# Domain to use for building the hostnames
# dhcp_domain = openstacklocal

# Override the default dnsmasq settings with this file
# dnsmasq_config_file =

# Comma-separated list of DNS servers which will be used by dnsmasq
# as forwarders.
# dnsmasq_dns_servers =

# Limit number of leases to prevent a denial-of-service.
# dnsmasq_lease_max = 16777216

# Location to DHCP lease relay UNIX domain socket
# dhcp_lease_relay_socket = $state_path/dhcp/lease_relay

# Use broadcast in DHCP replies
# dhcp_broadcast_reply = False

# Location of Metadata Proxy UNIX domain socket
# metadata_proxy_socket = $state_path/metadata_proxy
```

```
# dhcp_delete_namespaces, which is false by default, can be set to True if
# namespaces can be deleted cleanly on the host running the dhcp agent.
# Do not enable this until you understand the problem with the Linux iproute
# utility mentioned in https://bugs.launchpad.net/neutron/+bug/1052535 and
# you are sure that your version of iproute does not suffer from the problem.
# If True, namespaces will be deleted when a dhcp server is disabled.
# dhcp_delete_namespaces = False

# Timeout for ovs-vsctl commands.
# If the timeout expires, ovs commands will fail with ALARMCLOCK error.
# ovs_vsctl_timeout = 10
```


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
```

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。

```
[DEFAULT]
# Show debugging output in log (sets DEBUG log level output)
# debug = False

# L3 requires that an interface driver be set. Choose the one that best
# matches your plugin.
# interface_driver =

# Example of interface_driver option for OVS based plugins (OVS, Ryu, NEC)
# that supports L3 agent
# interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver

# Use veth for an OVS interface or not.
# Support kernels with limited namespace support
# (e.g. RHEL 6.5) so long as ovs_use_veth is set to True.
# ovs_use_veth = False

# Example of interface_driver option for LinuxBridge
# interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver

# Allow overlapping IP (Must have kernel build with CONFIG_NET_NS=y and
# iproute2 package that supports namespaces).
# use_namespaces = True

# If use_namespaces is set as False then the agent can only configure one router.

# This is done by setting the specific router_id.
# router_id =

# When external_network_bridge is set, each L3 agent can be associated
# with no more than one external network. This value should be set to the UUID
# of that external network. To allow L3 agent support multiple external
# networks, both the external_network_bridge and gateway_external_network_id
# must be left empty.
# gateway_external_network_id =

# Indicates that this L3 agent should also handle routers that do not have
# an external network gateway configured. This option should be True only
# for a single agent in a Neutron deployment, and may be False for all agents
# if all routers must have an external network gateway
# handle_internal_only_routers = True

# Name of bridge used for external network traffic. This should be set to
# empty value for the linux bridge. when this parameter is set, each L3 agent
# can be associated with no more than one external network.
# external_network_bridge = br-ex

# TCP Port used by Neutron metadata server
# metadata_port = 9697

# Send this many gratuitous ARPs for HA setup. Set it below or equal to 0
# to disable this feature.
```

```

# send_arp_for_ha = 3

# seconds between re-sync routers' data if needed
# periodic_interval = 40

# seconds to start to sync routers' data after
# starting agent
# periodic_fuzzy_delay = 5

# enable_metadata_proxy, which is true by default, can be set to False
# if the Nova metadata server is not available
# enable_metadata_proxy = True

# Location of Metadata Proxy UNIX domain socket
# metadata_proxy_socket = $state_path/metadata_proxy

# router_delete_namespaces, which is false by default, can be set to True if
# namespaces can be deleted cleanly on the host running the L3 agent.
# Do not enable this until you understand the problem with the Linux iproute
# utility mentioned in https://bugs.launchpad.net/neutron/+bug/1052535 and
# you are sure that your version of iproute does not suffer from the problem.
# If True, namespaces will be deleted when a router is destroyed.
# router_delete_namespaces = False

# Timeout for ovs-vsctl commands.
# If the timeout expires, ovs commands will fail with ALARMCLOCK error.
# ovs_vsctl_timeout = 10

# The working mode for the agent. Allowed values are:
# - legacy: this preserves the existing behavior where the L3 agent is
#   deployed on a centralized networking node to provide L3 services
#   like DNAT, and SNAT. Use this mode if you do not want to adopt DVR.
# - dvr: this mode enables DVR functionality, and must be used for an L3
#   agent that runs on a compute host.
# - dvr_snat: this enables centralized SNAT support in conjunction with
#   DVR. This mode must be used for an L3 agent running on a centralized
#   node (or in single-host deployments, e.g. devstack).
# agent_mode = legacy

# Location to store keepalived and all HA configurations
# ha_confs_path = $state_path/ha_confs

# VRRP authentication type AH/PASS
# ha_vrrp_auth_type = PASS

# VRRP authentication password
# ha_vrrp_auth_password =

# The advertisement interval in seconds
# ha_vrrp_advert_int = 2

```

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。

```
[DEFAULT]
# Show debugging output in log (sets DEBUG log level output).
# debug = False

# The LBaaS agent will resync its state with Neutron to recover from any
# transient notification or rpc errors. The interval is number of
# seconds between attempts.
# periodic_interval = 10

# LBaaS requires an interface driver be set. Choose the one that best
# matches your plugin.
# interface_driver =

# Example of interface_driver option for OVS based plugins (OVS, Ryu, NEC, NVP,
# BigSwitch/Floodlight)
# interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver

# Use veth for an OVS interface or not.
# Support kernels with limited namespace support
# (e.g. RHEL 6.5) so long as ovs_use_veth is set to True.
# ovs_use_veth = False

# Example of interface_driver option for LinuxBridge
# interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver

# The agent requires drivers to manage the loadbalancer. HAProxy is the opensource versi
# Multiple device drivers reflecting different service providers could be specified:
# device_driver = path.to.provider1.driver.Driver
# device_driver = path.to.provider2.driver.Driver
# Default is:
# device_driver = neutron.services.loadbalancer.drivers.haproxy.namespace_driver.HaproxyN

[haproxy]
# Location to store config and state files
# loadbalancer_state_path = $state_path/lbaas

# The user group
# user_group = nogroup

# When delete and re-add the same vip, send this many gratuitous ARPs to flush
# the ARP cache in the Router. Set it below or equal to 0 to disable this feature.
# send_gratuitous_arp = 3
```

metadata_agent.ini

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

```
[DEFAULT]
# Show debugging output in log (sets DEBUG log level output)
# debug = True

# The Neutron user information for accessing the Neutron API.
auth_url = http://localhost:5000/v2.0
auth_region = RegionOne
# Turn off verification of the certificate for ssl
# auth_insecure = False
# Certificate Authority public key (CA cert) file for ssl
# auth_ca_cert =
admin_tenant_name = %SERVICE_TENANT_NAME%
admin_user = %SERVICE_USER%
admin_password = %SERVICE_PASSWORD%

# Network service endpoint type to pull from the keystone catalog
# endpoint_type = adminURL

# IP address used by Nova metadata server
# nova_metadata_ip = 127.0.0.1

# TCP Port used by Nova metadata server
# nova_metadata_port = 8775

# Which protocol to use for requests to Nova metadata server, http or https
# nova_metadata_protocol = http

# Whether insecure SSL connection should be accepted for Nova metadata server
# requests
# nova_metadata_insecure = False

# Client certificate for nova api, needed when nova api requires client
# certificates
# nova_client_cert =

# Private key for nova client certificate
# nova_client_priv_key =

# When proxying metadata requests, Neutron signs the Instance-ID header with a
# shared secret to prevent spoofing. You may select any string for a secret,
# but it must match here and in the configuration used by the Nova Metadata
# Server. NOTE: Nova uses a different key: neutron_metadata_proxy_shared_secret
# metadata_proxy_shared_secret =

# Location of Metadata Proxy UNIX domain socket
# metadata_proxy_socket = $state_path/metadata_proxy

# Number of separate worker processes for metadata server. Defaults to
# half the number of CPU cores
# metadata_workers =

# Number of backlog requests to configure the metadata server socket with
```

```
# metadata_backlog = 4096

# URL to connect to the cache backend.
# default_ttl=0 parameter will cause cache entries to never expire.
# Otherwise default_ttl specifies time in seconds a cache entry is valid for.
# No cache is used in case no value is passed.
# cache_url = memory://?default_ttl=5
```

metering_agent.ini

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

```
[DEFAULT]
# Show debugging output in log (sets DEBUG log level output)
# debug = True

# Default driver:
# driver = neutron.services.metering.drivers.noop.noop_driver.NoopMeteringDriver
# Example of non-default driver
# driver = neutron.services.metering.drivers.iptables.iptables_driver.IptablesMeteringDriver

# Interval between two metering measures
# measure_interval = 30

# Interval between two metering reports
# report_interval = 300

# interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver

# use_namespaces = True
```

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。

```
[DEFAULT]
# VPN-Agent configuration file
# Note vpn-agent inherits l3-agent, so you can use configs on l3-agent also

[vpnagent]
# vpn device drivers which vpn agent will use
# If we want to use multiple drivers, we need to define this option multiple times.
# vpn_device_driver=neutron.services.vpn.device_drivers.ipsec.OpenSwanDriver
# vpn_device_driver=neutron.services.vpn.device_drivers.cisco_ipsec.CiscoCsrIPsecDriver
# vpn_device_driver=another_driver

[ipsec]
# Status check interval
# ipsec_status_check_interval=60
```


neutron.conf

neutron-server 启动后读取的配置信息。

```
[DEFAULT]
# Print more verbose output (set logging level to INFO instead of default WARNING level).
# verbose = False

# =====Start Global Config Option for Distributed L3 Router=====
# Setting the "router_distributed" flag to "True" will default to the creation
# of distributed tenant routers. The admin can override this flag by specifying
# the type of the router on the create request (admin-only attribute). Default
# value is "False" to support legacy mode (centralized) routers.
#
# router_distributed = False
#
# =====End Global Config Option for Distributed L3 Router=====

# Print debugging output (set logging level to DEBUG instead of default WARNING level).
# debug = False

# Where to store Neutron state files. This directory must be writable by the
# user executing the agent.
# state_path = /var/lib/neutron

# Where to store lock files
lock_path = $state_path/lock

# log_format = %(asctime)s %(levelname)s [%s] %(message)s
# log_date_format = %Y-%m-%d %H:%M:%S

# use_syslog                                -> syslog
# log_file and log_dir                      -> log_dir/log_file
# (not log_file) and log_dir                -> log_dir/{binary_name}.log
# use_stderr                              -> stderr
# (not user_stderr) and (not log_file)      -> stdout
# publish_errors                          -> notification system

# use_syslog = False
# syslog_log_facility = LOG_USER

# use_stderr = True
# log_file =
# log_dir =

# publish_errors = False

# Address to bind the API server to
# bind_host = 0.0.0.0

# Port to bind the API server to
# bind_port = 9696

# Path to the extensions. Note that this can be a colon-separated list of
# paths. For example:
# api_extensions_path = extensions:/path/to/more/extensions:/even/more/extensions
```

```

# The __path__ of neutron.extensions is appended to this, so if your
# extensions are in there you don't need to specify them here
# api_extensions_path =

# (StrOpt) Neutron core plugin entrypoint to be loaded from the
# neutron.core_plugins namespace. See setup.cfg for the entrypoint names of the
# plugins included in the neutron source distribution. For compatibility with
# previous versions, the class name of a plugin can be specified instead of its
# entrypoint name.
#
# core_plugin =
# Example: core_plugin = ml2

# (ListOpt) List of service plugin entrypoints to be loaded from the
# neutron.service_plugins namespace. See setup.cfg for the entrypoint names of
# the plugins included in the neutron source distribution. For compatibility
# with previous versions, the class name of a plugin can be specified instead
# of its entrypoint name.
#
# service_plugins =
# Example: service_plugins = router,firewall,lbaas,vpnaas,metering

# Paste configuration file
# api_paste_config = api-paste.ini

# The strategy to be used for auth.
# Supported values are 'keystone'(default), 'noauth'.
# auth_strategy = keystone

# Base MAC address. The first 3 octets will remain unchanged. If the
# 4th octet is not 00, it will also be used. The others will be
# randomly generated.
# 3 octet
# base_mac = fa:16:3e:00:00:00
# 4 octet
# base_mac = fa:16:3e:4f:00:00

# DVR Base MAC address. The first 3 octets will remain unchanged. If the
# 4th octet is not 00, it will also be used. The others will be randomly
# generated. The 'dvr_base_mac' *must* be different from 'base_mac' to
# avoid mixing them up with MAC's allocated for tenant ports.
# A 4 octet example would be dvr_base_mac = fa:16:3f:4f:00:00
# The default is 3 octet
# dvr_base_mac = fa:16:3f:00:00:00

# Maximum amount of retries to generate a unique MAC address
# mac_generation_retries = 16

# DHCP Lease duration (in seconds). Use -1 to
# tell dnsmasq to use infinite lease times.
# dhcp_lease_duration = 86400

# Allow sending resource operation notification to DHCP agent
# dhcp_agent_notification = True

# Enable or disable bulk create/update/delete operations
# allow_bulk = True
# Enable or disable pagination
# allow_pagination = False

```

```

# Enable or disable sorting
# allow_sorting = False
# Enable or disable overlapping IPs for subnets
# Attention: the following parameter MUST be set to False if Neutron is
# being used in conjunction with nova security groups
# allow_overlapping_ips = False
# Ensure that configured gateway is on subnet. For IPv6, validate only if
# gateway is not a link local address. Deprecated, to be removed during the
# K release, at which point the check will be mandatory.
# force_gateway_on_subnet = True

# Default maximum number of items returned in a single response,
# value == infinite and value < 0 means no max limit, and value must
# be greater than 0. If the number of items requested is greater than
# pagination_max_limit, server will just return pagination_max_limit
# of number of items.
# pagination_max_limit = -1

# Maximum number of DNS nameservers per subnet
# max_dns_nameservers = 5

# Maximum number of host routes per subnet
# max_subnet_host_routes = 20

# Maximum number of fixed ips per port
# max_fixed_ips_per_port = 5

# Maximum number of routes per router
# max_routes = 30

# ===== items for agent management extension =====
# Seconds to regard the agent as down; should be at least twice
# report_interval, to be sure the agent is down for good
# agent_down_time = 75
# ===== end of items for agent management extension =====

# ===== items for agent scheduler extension =====
# Driver to use for scheduling network to DHCP agent
# network_scheduler_driver = neutron.scheduler.dhcp_agent_scheduler.ChanceScheduler
# Driver to use for scheduling router to a default L3 agent
# router_scheduler_driver = neutron.scheduler.l3_agent_scheduler.ChanceScheduler
# Driver to use for scheduling a loadbalancer pool to an lbaas agent
# loadbalancer_pool_scheduler_driver = neutron.services.loadbalancer.agent_scheduler.ChanceScheduler

# Allow auto scheduling networks to DHCP agent. It will schedule non-hosted
# networks to first DHCP agent which sends get_active_networks message to
# neutron server
# network_auto_schedule = True

# Allow auto scheduling routers to L3 agent. It will schedule non-hosted
# routers to first L3 agent which sends sync_routers message to neutron server
# router_auto_schedule = True

# Allow automatic rescheduling of routers from dead L3 agents with
# admin_state_up set to True to alive agents.
# allow_automatic_l3agent_failover = False

# Number of DHCP agents scheduled to host a network. This enables redundant
# DHCP agents for configured networks.

```

```

# dhcp_agents_per_network = 1

# ===== end of items for agent scheduler extension =====

# ===== items for l3 extension =====
# Enable high availability for virtual routers.
# l3_ha = False
#
# Maximum number of l3 agents which a HA router will be scheduled on. If it
# is set to 0 the router will be scheduled on every agent.
# max_l3_agents_per_router = 3
#
# Minimum number of l3 agents which a HA router will be scheduled on. The
# default value is 2.
# min_l3_agents_per_router = 2
#
# CIDR of the administrative network if HA mode is enabled
# l3_ha_net_cidr = 169.254.192.0/18
# ===== end of items for l3 extension =====

# ===== WSGI parameters related to the API server =====
# Number of separate worker processes to spawn. The default, 0, runs the
# worker thread in the current process. Greater than 0 launches that number of
# child processes as workers. The parent process manages them.
# api_workers = 0

# Number of separate RPC worker processes to spawn. The default, 0, runs the
# worker thread in the current process. Greater than 0 launches that number of
# child processes as RPC workers. The parent process manages them.
# This feature is experimental until issues are addressed and testing has been
# enabled for various plugins for compatibility.
# rpc_workers = 0

# Sets the value of TCP_KEEPIDLE in seconds to use for each server socket when
# starting API server. Not supported on OS X.
# tcp_keepidle = 600

# Number of seconds to keep retrying to listen
# retry_until_window = 30

# Number of backlog requests to configure the socket with.
# backlog = 4096

# Max header line to accommodate large tokens
# max_header_line = 16384

# Enable SSL on the API server
# use_ssl = False

# Certificate file to use when starting API server securely
# ssl_cert_file = /path/to/certfile

# Private key file to use when starting API server securely
# ssl_key_file = /path/to/keyfile

# CA certificate file to use when starting API server securely to
# verify connecting clients. This is an optional parameter only required if
# API clients need to authenticate to the API server using SSL certificates
# signed by a trusted CA

```

```

# ssl_ca_file = /path/to/cafile
# ===== end of WSGI parameters related to the API server =====

# ===== neutron nova interactions =====
# Send notification to nova when port status is active.
# notify_nova_on_port_status_changes = True

# Send notifications to nova when port data (fixed_ips/floatingips) change
# so nova can update it's cache.
# notify_nova_on_port_data_changes = True

# URL for connection to nova (Only supports one nova region currently).
# nova_url = http://127.0.0.1:8774/v2

# Name of nova region to use. Useful if keystone manages more than one region
# nova_region_name =

# Username for connection to nova in admin context
# nova_admin_username =

# The uuid of the admin nova tenant
# nova_admin_tenant_id =

# The name of the admin nova tenant. If the uuid of the admin nova tenant
# is set, this is optional. Useful for cases where the uuid of the admin
# nova tenant is not available when configuration is being done.
# nova_admin_tenant_name =

# Password for connection to nova in admin context.
# nova_admin_password =

# Authorization URL for connection to nova in admin context.
# nova_admin_auth_url =

# CA file for novaclient to verify server certificates
# nova_ca_certificates_file =

# Boolean to control ignoring SSL errors on the nova url
# nova_api_insecure = False

# Number of seconds between sending events to nova if there are any events to send
# send_events_interval = 2

# ===== end of neutron nova interactions =====

#
# Options defined in oslo.messaging
#

# Use durable queues in amqp. (boolean value)
# Deprecated group/name - [DEFAULT]/rabbit_durable_queues
#amqp_durable_queues=false

# Auto-delete queues in amqp. (boolean value)
#amqp_auto_delete=false

# Size of RPC connection pool. (integer value)
#rpc_conn_pool_size=30

```

```

# Qpid broker hostname. (string value)
#qpid_hostname=localhost

# Qpid broker port. (integer value)
#qpid_port=5672

# Qpid HA cluster host:port pairs. (list value)
#qpid_hosts=$qpid_hostname:$qpid_port

# Username for Qpid connection. (string value)
#qpid_username=

# Password for Qpid connection. (string value)
#qpid_password=

# Space separated list of SASL mechanisms to use for auth.
# (string value)
#qpid_sasl_mechanisms=

# Seconds between connection keepalive heartbeats. (integer
# value)
#qpid_heartbeat=60

# Transport to use, either 'tcp' or 'ssl'. (string value)
#qpid_protocol=tcp

# Whether to disable the Nagle algorithm. (boolean value)
#qpid_tcp_nodelay=true

# The qpid topology version to use. Version 1 is what was
# originally used by impl_qpid. Version 2 includes some
# backwards-incompatible changes that allow broker federation
# to work. Users should update to version 2 when they are
# able to take everything down, as it requires a clean break.
# (integer value)
#qpid_topology_version=1

# SSL version to use (valid only if SSL enabled). valid values
# are TLSv1, SSLv23 and SSLv3. SSLv2 may be available on some
# distributions. (string value)
#kombu_ssl_version=

# SSL key file (valid only if SSL enabled). (string value)
#kombu_ssl_keyfile=

# SSL cert file (valid only if SSL enabled). (string value)
#kombu_ssl_certfile=

# SSL certification authority file (valid only if SSL
# enabled). (string value)
#kombu_ssl_ca_certs=

# How long to wait before reconnecting in response to an AMQP
# consumer cancel notification. (floating point value)
#kombu_reconnect_delay=1.0

# The RabbitMQ broker address where a single node is used.
# (string value)

```

```

#rabbit_host=localhost

# The RabbitMQ broker port where a single node is used.
# (integer value)
#rabbit_port=5672

# RabbitMQ HA cluster host:port pairs. (list value)
#rabbit_hosts=$rabbit_host:$rabbit_port

# Connect over SSL for RabbitMQ. (boolean value)
#rabbit_use_ssl=false

# The RabbitMQ userid. (string value)
#rabbit_userid=guest

# The RabbitMQ password. (string value)
#rabbit_password=guest

# the RabbitMQ login method (string value)
#rabbit_login_method=AMQPPLAIN

# The RabbitMQ virtual host. (string value)
#rabbit_virtual_host=/

# How frequently to retry connecting with RabbitMQ. (integer
# value)
#rabbit_retry_interval=1

# How long to backoff for between retries when connecting to
# RabbitMQ. (integer value)
#rabbit_retry_backoff=2

# Maximum number of RabbitMQ connection retries. Default is 0
# (infinite retry count). (integer value)
#rabbit_max_retries=0

# Use HA queues in RabbitMQ (x-ha-policy: all). If you change
# this option, you must wipe the RabbitMQ database. (boolean
# value)
#rabbit_ha_queues=false

# If passed, use a fake RabbitMQ provider. (boolean value)
#fake_rabbit=false

# ZeroMQ bind address. Should be a wildcard (*), an ethernet
# interface, or IP. The "host" option should point or resolve
# to this address. (string value)
#rpc_zmq_bind_address=*

# MatchMaker driver. (string value)
#rpc_zmq_matchmaker=oslo.messaging._drivers.matchmaker.MatchMakerLocalhost

# ZeroMQ receiver listening port. (integer value)
#rpc_zmq_port=9501

# Number of ZeroMQ contexts, defaults to 1. (integer value)
#rpc_zmq_contexts=1

# Maximum number of ingress messages to locally buffer per

```

```

# topic. Default is unlimited. (integer value)
#rpc_zmq_topic_backlog=<None>

# Directory for holding IPC sockets. (string value)
#rpc_zmq_ipc_dir=/var/run/openstack

# Name of this node. Must be a valid hostname, FQDN, or IP
# address. Must match "host" option, if running Nova. (string
# value)
#rpc_zmq_host=oslo

# Seconds to wait before a cast expires (TTL). Only supported
# by impl_zmq. (integer value)
#rpc_cast_timeout=30

# Heartbeat frequency. (integer value)
#matchmaker_heartbeat_freq=300

# Heartbeat time-to-live. (integer value)
#matchmaker_heartbeat_ttl=600

# Size of RPC greenthread pool. (integer value)
#rpc_thread_pool_size=64

# Driver or drivers to handle sending notifications. (multi
# valued)
#notification_driver=

# AMQP topic used for OpenStack notifications. (list value)
# Deprecated group/name - [rpc_notifier2]/topics
#notification_topics=notifications

# Seconds to wait for a response from a call. (integer value)
#rpc_response_timeout=60

# A URL representing the messaging driver to use and its full
# configuration. If not set, we fall back to the rpc_backend
# option and driver specific configuration. (string value)
#transport_url=<None>

# The messaging driver to use, defaults to rabbit. Other
# drivers include qpid and zmq. (string value)
#rpc_backend=rabbit

# The default exchange under which topics are scoped. May be
# overridden by an exchange name specified in the
# transport_url option. (string value)
#control_exchange=openstack

[matchmaker_redis]

#
# Options defined in oslo.messaging
#

# Host to locate redis. (string value)
#host=127.0.0.1

```



```

# Use this port to connect to redis host. (integer value)
#port=6379

# Password for Redis server (optional). (string value)
#password=<None>

[matchmaker_ring]

#
# Options defined in oslo.messaging
#

# Matchmaker ring file (JSON). (string value)
# Deprecated group/name - [DEFAULT]/matchmaker_ringfile
#ringfile=/etc/oslo/matchmaker_ring.json

[quotas]
# Default driver to use for quota checks
# quota_driver = neutron.db.quota_db.DbQuotaDriver

# Resource name(s) that are supported in quota features
# quota_items = network,subnet,port

# Default number of resource allowed per tenant. A negative value means
# unlimited.
# default_quota = -1

# Number of networks allowed per tenant. A negative value means unlimited.
# quota_network = 10

# Number of subnets allowed per tenant. A negative value means unlimited.
# quota_subnet = 10

# Number of ports allowed per tenant. A negative value means unlimited.
# quota_port = 50

# Number of security groups allowed per tenant. A negative value means
# unlimited.
# quota_security_group = 10

# Number of security group rules allowed per tenant. A negative value means
# unlimited.
# quota_security_group_rule = 100

# Number of vips allowed per tenant. A negative value means unlimited.
# quota_vip = 10

# Number of pools allowed per tenant. A negative value means unlimited.
# quota_pool = 10

# Number of pool members allowed per tenant. A negative value means unlimited.
# The default is unlimited because a member is not a real resource consumer
# on Openstack. However, on back-end, a member is a resource consumer
# and that is the reason why quota is possible.
# quota_member = -1

# Number of health monitors allowed per tenant. A negative value means
# unlimited.

```

```

# The default is unlimited because a health monitor is not a real resource
# consumer on Openstack. However, on back-end, a member is a resource consumer
# and that is the reason why quota is possible.
# quota_health_monitor = -1

# Number of routers allowed per tenant. A negative value means unlimited.
# quota_router = 10

# Number of floating IPs allowed per tenant. A negative value means unlimited.
# quota_floatingip = 50

# Number of firewalls allowed per tenant. A negative value means unlimited.
# quota_firewall = 1

# Number of firewall policies allowed per tenant. A negative value means
# unlimited.
# quota_firewall_policy = 1

# Number of firewall rules allowed per tenant. A negative value means
# unlimited.
# quota_firewall_rule = 100

[agent]
# Use "sudo neutron-rootwrap /etc/neutron/rootwrap.conf" to use the real
# root filter facility.
# Change to "sudo" to skip the filtering and just run the comand directly
# root_helper = sudo

# Set to true to add comments to generated iptables rules that describe
# each rule's purpose. (System must support the iptables comments module.)
# comment_iptables_rules = True

# ===== items for agent management extension =====
# seconds between nodes reporting state to server; should be less than
# agent_down_time, best if it is half or less than agent_down_time
# report_interval = 30

# ===== end of items for agent management extension =====

[keystone_auth_token]
auth_host = 127.0.0.1
auth_port = 35357
auth_protocol = http
admin_tenant_name = %SERVICE_TENANT_NAME%
admin_user = %SERVICE_USER%
admin_password = %SERVICE_PASSWORD%

[database]
# This line MUST be changed to actually run the plugin.
# Example:
# connection = mysql://root:pass@127.0.0.1:3306/neutron
# Replace 127.0.0.1 above with the IP address of the database used by the
# main neutron server. (Leave it as is if the database runs on this host.)
# connection = sqlite://
# NOTE: In deployment the [database] section and its connection attribute may
# be set in the corresponding core plugin '.ini' file. However, it is suggested
# to put the [database] section and its connection attribute in this
# configuration file.

```

```

# Database engine for which script will be generated when using offline
# migration
# engine =

# The SQLAlchemy connection string used to connect to the slave database
# slave_connection =

# Database reconnection retry times - in event connectivity is lost
# set to -1 implies an infinite retry count
# max_retries = 10

# Database reconnection interval in seconds - if the initial connection to the
# database fails
# retry_interval = 10

# Minimum number of SQL connections to keep open in a pool
# min_pool_size = 1

# Maximum number of SQL connections to keep open in a pool
# max_pool_size = 10

# Timeout in seconds before idle sql connections are reaped
# idle_timeout = 3600

# If set, use this value for max_overflow with sqlalchemy
# max_overflow = 20

# Verbosity of SQL debugging information. 0=None, 100=Everything
# connection_debug = 0

# Add python stack traces to SQL as comment strings
# connection_trace = False

# If set, use this value for pool_timeout with sqlalchemy
# pool_timeout = 10

[service_providers]
# Specify service providers (drivers) for advanced services like loadbalancer, VPN, Firew
# Must be in form:
# service_provider=<service_type>:<name>:<driver>[:default]
# List of allowed service types includes LOADBALANCER, FIREWALL, VPN
# Combination of <service type> and <name> must be unique; <driver> must also be unique
# This is multiline option, example for default provider:
# service_provider=LOADBALANCER:name:lbaas_plugin_driver_path:default
# example of non-default provider:
# service_provider=FIREWALL:name2:firewall_driver_path
# --- Reference implementations ---
service_provider=LOADBALANCER:Haproxy:neutron.services.loadbalancer.drivers.haproxy.plugins
service_provider=VPN:openswan:neutron.services.vpn.service_drivers.ipsec.IPsecVPNDriver:default
# In order to activate Radware's lbaas driver you need to uncomment the next line.
# If you want to keep the HA Proxy as the default lbaas driver, remove the attribute default
# Otherwise comment the HA Proxy line
# service_provider = LOADBALANCER:Radware:neutron.services.loadbalancer.drivers.radware.drivers
# uncomment the following line to make the 'netscaler' LBaaS provider available.
# service_provider=LOADBALANCER:NetScaler:neutron.services.loadbalancer.drivers.netscaler
# Uncomment the following line (and comment out the OpenSwan VPN line) to enable Cisco's
# service_provider=VPN:cisco:neutron.services.vpn.service_drivers.cisco_ipsec.CiscoCsrIPsec
# Uncomment the line below to use Embrane heleos as Load Balancer service provider.
# service_provider=LOADBALANCER:Embrane:neutron.services.loadbalancer.drivers.embrane.drivers

```

```
# Uncomment the line below to use the A10 Networks LBaaS driver. Requires 'pip install a
#service_provider = LOADBALANCER:A10Networks:neutron.services.loadbalancer.drivers.a10net
# Uncomment the following line to test the LBaaS v2 API _WITHOUT_ a real backend
# service_provider = LOADBALANCER:LoggingNoop:neutron.services.loadbalancer.drivers.loggi
```

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。

```
{
  "context_is_admin": "role:admin",
  "admin_or_owner": "rule:context_is_admin or tenant_id:%(tenant_id)s",
  "context_is_advsvc": "role:advsvc",
  "admin_or_network_owner": "rule:context_is_admin or tenant_id:%(network:tenant_id)s",
  "admin_only": "rule:context_is_admin",
  "regular_user": "",
  "shared": "field:networks:shared=True",
  "shared_firewalls": "field:firewalls:shared=True",
  "external": "field:networks:router:external=True",
  "default": "rule:admin_or_owner",

  "create_subnet": "rule:admin_or_network_owner",
  "get_subnet": "rule:admin_or_owner or rule:shared",
  "update_subnet": "rule:admin_or_network_owner",
  "delete_subnet": "rule:admin_or_network_owner",

  "create_network": "",
  "get_network": "rule:admin_or_owner or rule:shared or rule:external or rule:context_i",
  "get_network:router:external": "rule:regular_user",
  "get_network:segments": "rule:admin_only",
  "get_network:provider:network_type": "rule:admin_only",
  "get_network:provider:physical_network": "rule:admin_only",
  "get_network:provider:segmentation_id": "rule:admin_only",
  "get_network:queue_id": "rule:admin_only",
  "create_network:shared": "rule:admin_only",
  "create_network:router:external": "rule:admin_only",
  "create_network:segments": "rule:admin_only",
  "create_network:provider:network_type": "rule:admin_only",
  "create_network:provider:physical_network": "rule:admin_only",
  "create_network:provider:segmentation_id": "rule:admin_only",
  "update_network": "rule:admin_or_owner",
  "update_network:segments": "rule:admin_only",
  "update_network:shared": "rule:admin_only",
  "update_network:provider:network_type": "rule:admin_only",
  "update_network:provider:physical_network": "rule:admin_only",
  "update_network:provider:segmentation_id": "rule:admin_only",
  "update_network:router:external": "rule:admin_only",
  "delete_network": "rule:admin_or_owner",
```

```

"create_port": "",
"create_port:mac_address": "rule:admin_or_network_owner or rule:context_is_advsvc",
"create_port:fixed_ips": "rule:admin_or_network_owner or rule:context_is_advsvc",
"create_port:port_security_enabled": "rule:admin_or_network_owner or rule:context_is_a",
"create_port:binding:host_id": "rule:admin_only",
"create_port:binding:profile": "rule:admin_only",
"create_port:mac_learning_enabled": "rule:admin_or_network_owner or rule:context_is_a",
"get_port": "rule:admin_or_owner or rule:context_is_advsvc",
"get_port:queue_id": "rule:admin_only",
"get_port:binding:vif_type": "rule:admin_only",
"get_port:binding:vif_details": "rule:admin_only",
"get_port:binding:host_id": "rule:admin_only",
"get_port:binding:profile": "rule:admin_only",
"update_port": "rule:admin_or_owner or rule:context_is_advsvc",
"update_port:fixed_ips": "rule:admin_or_network_owner or rule:context_is_advsvc",
"update_port:port_security_enabled": "rule:admin_or_network_owner or rule:context_is_a",
"update_port:binding:host_id": "rule:admin_only",
"update_port:binding:profile": "rule:admin_only",
"update_port:mac_learning_enabled": "rule:admin_or_network_owner or rule:context_is_a",
"delete_port": "rule:admin_or_owner or rule:context_is_advsvc",

"get_router:ha": "rule:admin_only",
"create_router": "rule:regular_user",
"create_router:external_gateway_info:enable_snat": "rule:admin_only",
"create_router:distributed": "rule:admin_only",
"create_router:ha": "rule:admin_only",
"get_router": "rule:admin_or_owner",
"get_router:distributed": "rule:admin_only",
"update_router:external_gateway_info:enable_snat": "rule:admin_only",
"update_router:distributed": "rule:admin_only",
"update_router:ha": "rule:admin_only",
"delete_router": "rule:admin_or_owner",

"add_router_interface": "rule:admin_or_owner",
"remove_router_interface": "rule:admin_or_owner",

"create_firewall": "",
"get_firewall": "rule:admin_or_owner",
"create_firewall:shared": "rule:admin_only",
"get_firewall:shared": "rule:admin_only",
"update_firewall": "rule:admin_or_owner",
"update_firewall:shared": "rule:admin_only",
"delete_firewall": "rule:admin_or_owner",

"create_firewall_policy": "",
"get_firewall_policy": "rule:admin_or_owner or rule:shared_firewalls",
"create_firewall_policy:shared": "rule:admin_or_owner",
"update_firewall_policy": "rule:admin_or_owner",
"delete_firewall_policy": "rule:admin_or_owner",

"create_firewall_rule": "",
"get_firewall_rule": "rule:admin_or_owner or rule:shared_firewalls",
"update_firewall_rule": "rule:admin_or_owner",
"delete_firewall_rule": "rule:admin_or_owner",

"create_qos_queue": "rule:admin_only",
"get_qos_queue": "rule:admin_only",

"update_agent": "rule:admin_only",

```

```
"delete_agent": "rule:admin_only",
"get_agent": "rule:admin_only",

"create_dhcp-network": "rule:admin_only",
"delete_dhcp-network": "rule:admin_only",
"get_dhcp-networks": "rule:admin_only",
"create_l3-router": "rule:admin_only",
"delete_l3-router": "rule:admin_only",
"get_l3-routers": "rule:admin_only",
"get_dhcp-agents": "rule:admin_only",
"get_l3-agents": "rule:admin_only",
"get_loadbalancer-agent": "rule:admin_only",
"get_loadbalancer-pools": "rule:admin_only",

"create_floatingip": "rule:regular_user",
"update_floatingip": "rule:admin_or_owner",
"delete_floatingip": "rule:admin_or_owner",
"get_floatingip": "rule:admin_or_owner",

"create_network_profile": "rule:admin_only",
"update_network_profile": "rule:admin_only",
"delete_network_profile": "rule:admin_only",
"get_network_profiles": "",
"get_network_profile": "",
"update_policy_profiles": "rule:admin_only",
"get_policy_profiles": "",
"get_policy_profile": "",

"create_metering_label": "rule:admin_only",
"delete_metering_label": "rule:admin_only",
"get_metering_label": "rule:admin_only",

"create_metering_label_rule": "rule:admin_only",
"delete_metering_label_rule": "rule:admin_only",
"get_metering_label_rule": "rule:admin_only",

"get_service_provider": "rule:regular_user",
"get_lsn": "rule:admin_only",
"create_lsn": "rule:admin_only"
}
```

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。

```
# Configuration for neutron-rootwrap
# This file should be owned by (and only-writeable by) the root user

[DEFAULT]
# List of directories to load filter definitions from (separated by ',').
# These directories MUST all be only writeable by root !
filters_path=/etc/neutron/rootwrap.d,/usr/share/neutron/rootwrap

# List of directories to search executables in, in case filters do not
# explicitly specify a full path (separated by ',')
# If not specified, defaults to system PATH environment variable.
# These directories MUST all be only writeable by root !
exec_dirs=/sbin,/usr/sbin,/bin,/usr/bin

# Enable logging to syslog
# Default value is False
use_syslog=False

# Which syslog facility to use.
# Valid values include auth, authpriv, syslog, local0, local1...
# Default value is 'syslog'
syslog_log_facility=syslog

# Which messages to log.
# INFO means log all usage
# ERROR means only log unsuccessful attempts
syslog_log_level=ERROR

[xenapi]
# XenAPI configuration is only required by the L2 agent if it is to
# target a XenServer/XCP compute host's dom0.
xenapi_connection_url=<None>
xenapi_connection_username=root
xenapi_connection_password=<None>
```


services.conf

配置一些特殊的 service 信息。

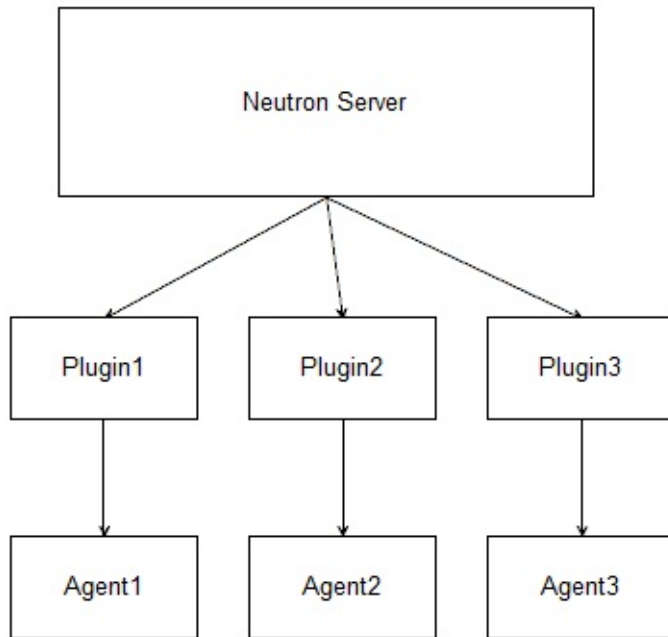
```
[radware]
#vdirect_address = 0.0.0.0
#ha_secondary_address=
#vdirect_user = vDirect
#vdirect_password = radware
#service_ha_pair = False
#service_throughput = 1000
#service_ssl_throughput = 200
#service_compression_throughput = 100
#service_cache = 20
#service_adc_type = VA
#service_adc_version=
#service_session_mirroring_enabled = False
#service_isl_vlan = -1
#service_resource_pool_ids = []
#actions_to_skip = 'setup_l2_l3'
#l4_action_name = 'BaseCreate'
#l2_l3_workflow_name = openstack_l2_l3
#l4_workflow_name = openstack_l4
#l2_l3_ctor_params = service: _REPLACE_, ha_network_name: HA-Network, ha_ip_pool_name: de
#l2_l3_setup_params = data_port: 1, data_ip_address: 192.168.200.99, data_ip_mask: 255.25

[netscaler_driver]
#netscaler_ncc_uri = https://ncc_server.acme.org/ncc/v1/api
#netscaler_ncc_username = admin
#netscaler_ncc_password = secret

[heleoslb]
#esm_mgmt =
#admin_username =
#admin_password =
#lb_image =
#inband_id =
#oob_id =
#mgmt_id =
#dummy_utif_id =
#resource_pool_id =
#async_requests =
#lb_flavor = small
#sync_interval = 60
```

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。



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。

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。

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。

一些配置常量

包括：

```
ROOT_HELPER_OPTS = [
    cfg.StrOpt('root_helper', default='sudo',
               help=_('Root helper application.')),
]

AGENT_STATE_OPTS = [
    cfg.FloatOpt('report_interval', default=30,
                 help=_('Seconds between nodes reporting state to server; '
                        'should be less than agent_down_time, best if it '
                        'is half or less than agent_down_time.')),
]

INTERFACE_DRIVER_OPTS = [
    cfg.StrOpt('interface_driver',
               help=_("The driver used to manage the virtual interface.")),
]

USE_NAMESPACES_OPTS = [
    cfg.BoolOpt('use_namespaces', default=True,
                help=_("Allow overlapping IP.")),
]
```

注册函数

往全局的 conf 中注册各个常量

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。

async_process.py

实现了AsyncProcess类，对异步进程进行管理。

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。

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。

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）。

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OVSIInterfaceDriver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。

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。

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()`。

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。

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。

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）。

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。

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。

metadata/

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如下：

```
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。

namespace_proxy.py

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

```
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()

```


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.agentnotifiers.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']
```

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()。

DhcpPluginApi 类

提供从agent往plugin一侧进行rpc调用的api。

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)
```

firewall.py

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

l2population_rpc.py

主要定义了 L2populationRpcCallBackMixin 元类和 L2populationRpcCallBackTunnelMixin 类。

前者作为通用类，处理跟 L2populationRpc 相关的回调。

后者为隧道处理 L2population 的 Rpc 回调。

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'):
    _register_opts(cfg.CONF)
    common_config.init(sys.argv[1:])
    config.setup_logging()
    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。

从 conf.CONF 中注册各个系统常量，包括 L3 相关 agent 的参数、接口驱动、命名空间等等。

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，管理类为 L3NATAgentWithStateReport。

L3NATAgent 类

继承自 firewall_l3_agent.FWaaS_L3AgentRpcCallback、l3_ha_agent.AgentMixin 和 manager.Manager。

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。

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。

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。

主要添加了两个方法。

- `_report_state()`：定期的汇报自己的状态，以 `topic.PLUGIN` 作为主题向 rpc 队列中写入 agent 的状态信息消息。这些消息会被各个 plugin 收到。

- `agent_updated()` : 收到 `agent_updated` 消息的处理。

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。

目前定义了下面几个方法：

- `get_routers()` 通过 rpc 调用 L3 Plugin 的 `sync_routers()` 方法来获取外部网络的id。
- `get_external_network_id()` 通过 rpc 调用 `external_network_id()` 来获取外部网络的id。
- `update_floatingip_statuses()` 通过 rpc 调用 `update_floatingip_statuses()` 来更新 floating ip 的状态。
- `get_ports_by_subnet()` 通过 rpc 调用 `get_ports_by_subnet()` 来获取对应 subnet 中的端口信息。
- `get_service_plugin_list()` 通过 rpc 调用 `get_service_plugin_list()` 来获取 L3 Plugin 中激活服务的列表。

l3_ha_agent.py

L3 的 HA 主要是通过 VRRP 来实现的，这里定义了两个类。

- AgentMixin
- RouterMixin

这两个类中都实现了跟 HA 相关的一些方法，供 L3 Agent 使用。

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。

```
def main():
    """Main method for cleaning up network namespaces.

    This method will make two passes checking for namespaces to delete. The
    process will identify candidates, sleep, and call garbage collect. The
    garbage collection will re-verify that the namespace meets the criteria for
    deletion (ie it is empty). The period of sleep and the 2nd pass allow
    time for the namespace state to settle, so that the check prior deletion
    will re-confirm the namespace is empty.

    The utility is designed to clean-up after the forced or unexpected
    termination of Neutron agents.

    The --force flag should only be used as part of the cleanup of a devstack
    installation as it will blindly purge namespaces and their devices. This
    option also kills any lingering DHCP instances.
    """
    conf = setup_conf()
    conf()
    config.setup_logging()

    root_helper = agent_config.get_root_helper(conf)
    # Identify namespaces that are candidates for deletion.
    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)
```


ovs_cleanup_util.py

清理无用的ovs网桥和端口。

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。

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。

api

提供RestAPI访问。

rpc

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。

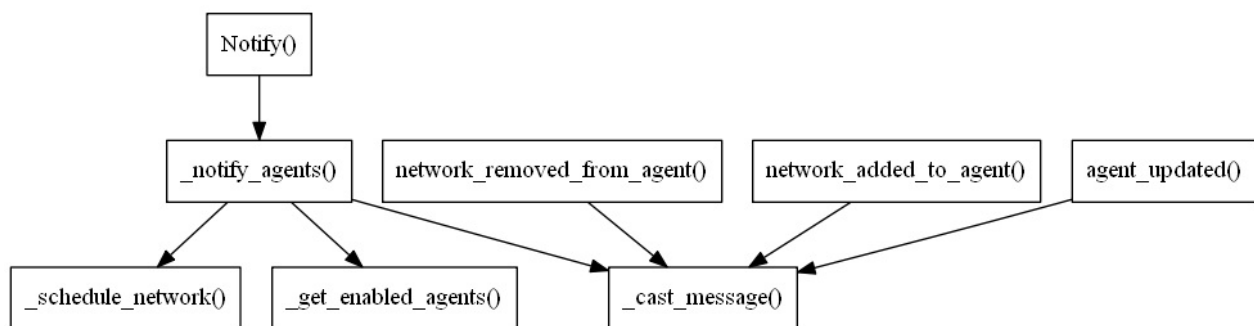
```
def notify(self, context, data, method_name):
    # data is {'key' : 'value'} with only one key
    if method_name not in self.VALID_METHOD_NAMES:
        return
```

```

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API的agent 通知调用的主要调用结构如下图所示。



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。

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。

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。

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。

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。

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。

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)

```


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。

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。

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。

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
```

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。

common

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

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。

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。

exceptions.py

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

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。

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。

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，最后一个是所有广播请求。

Connection类代表了rpc请求的相关连接。

test_lib.py

定义了test_config={}, 用于各个plugin进行测试。

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-notifier'
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'
```

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。

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。

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crud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。

agentschedulers_db.py

跟agent scheduler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。

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。

common_db_mixin.py

定义了核心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。

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。

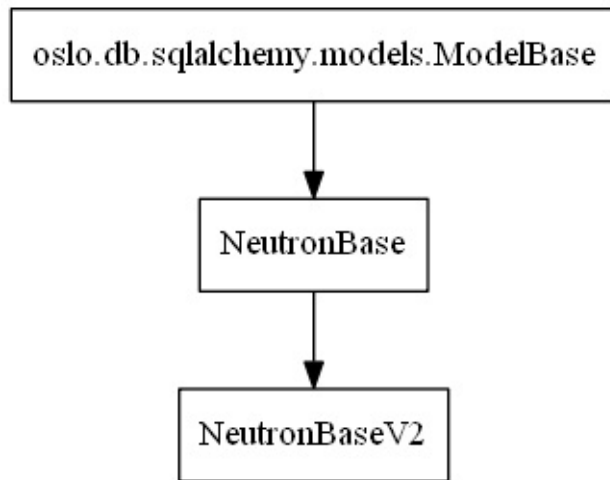
该类利用SQLAlchemy的模型实现了neutron plugin 的接口，主要包括对网络、子网、端口等资源的 CRUD 操作。

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。

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)
```


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。

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。

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。

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。

allowedaddresspairs_db.py

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

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。

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。

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。

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。

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；



l3_agentschedulers_db.py

定义类 L3AgentSchedulerDbMixin 和 RouterL3AgentBinding。

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。

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。

l3_dvr_db.py

定义了类 `L3_NAT_with_dvr_db_mixin`，对DVR进行支持。

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。

I3_hamode_db.py

I3_hascheduler_db.py

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。



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。

portbindings_base.py

portbindings_db.py

定义了端口绑定的基础类 PortBindingBaseMixin。

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。

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。

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。

router servicetype_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。

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。

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。

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。

debug

提供简单的辅助debug功能。

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。

debug_agent.py

通过调用client来执行各项查询操作和debug命令。

shell.py

提供一个shell环境来完成debug。

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。

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。

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。

allowedaddresspairs.py

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

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。

dvr.py

定义了Dvr类，代表分布式虚拟路由器的扩展类。

external_net.py

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

extraroute.py

Extraroute 类，提供附加的路由支持。

extra_dhcp_opt.py

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

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。

flavor.py

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

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。

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。

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。

l3_ext_ha_mode.py

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。

loadbalancer.py

Loadbalancer扩展类，继承自extensions.ExtensionDescriptor，支持Ibaas 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。

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。

multiprovidernet.py

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

portbindings.py

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

portsecurity.py

Portsecurity扩展类，提供端口安全支持。

providernet.py

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

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。

routedserviceinsertion.py

Routedserviceinsertion扩展类，提供路由服务类型支持。

routerservicetype.py

Routerservicetype扩展类，提供路由服务类型支持。

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。

servicetype.py

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

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。

hacking

定义一些修改代码中需要的辅助函数，主要包括 `check.py` 模块。

checks.py

定义了一些辅助函数。包括检查作者标签，日志格式等。

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。

notifiers

nova.py

定义 `Notifier` 类，发送nova可能关心的一些事件消息。

openstack

common

公共模块。

cache

- _backends
- backends.py
- cache.py

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()。

eventlet_backdoor.py

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。

fixture

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

local.py

管理对线程局部的变量。

lockutils.py

跟锁相关的方法，包括锁和同步等。

log.py

日志相关的类和方法。

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。

middleware

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

periodic_task.py

定义了类 PeriodicTasks，表示定期的任务。

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。

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`

service.py

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

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

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。

uuidutils.py

对uuid进行检查和生成。

versionutils.py

检查版本号的兼容性。

_i18n.py

plugins

包括实现网络功能的各个插件。

bigswitch

agent

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。

ibm

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。

common

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。

sdnve_api_fake.py

伪造响应 API 调用请求，可用于测试。

sdnve_neutron_plugin.py

主要定义了 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。

ml2

common

config.py

db.py

drivers

driver_api.py

driver_context.py

managers.py

两个manager类：MechanismManager和TypeManager。

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。

plugin.py

主要实现类 ML2Plugin，是 ML2 的 Plugin 端的主类。继承自多个父类。

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。

ofagent

openflow agent机制的驱动应用。

agent

ofa_neutron_agent.py

ports.py

定义了一个Port类，表示一个OF端口。

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。

openvswitch

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。

setup_rpc()

setup_rpc()首先创建了两个rpc句柄，分别是

```
self.plugin_rpc = OVSPPluginApi(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。



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。

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。

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。

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。

scheduler

实现调度、负载均衡的算法。

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。

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。

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的绿色池来运行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。

下面的图表总结了neutron-server的核心启动过程。

neutron.server	neutron.service#serve_wsgi()	neutron.service.NeutronApiService
neutron_api = service.serve_wsgi(service.NeutronApiService)	service = cls.create(); service.start()	service = cls(app_name='neutron')

services

firewall

防火墙服务。

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。

drivers

包括对linux防火墙和varmour防火墙进行操作的驱动类。

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之间的调用关系如下面图表所示。



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。

l3_router_plugin.py

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

loadbalancer

与其他服务实现类似，多了 agent 的 scheduler。

agent

包括 agent.py、agent_api.py、agent_device_driver.py agent_manager.py 等模块，实现 LBaaS 服务的 agent 部分。

agent.py

按照标准流程，启动了一个 LbaasAgentService 服务，主代码如下。

```
def main():
    cfg.CONF.register_opts(OPTS)
    cfg.CONF.register_opts(manager.OPTS)
    # import interface options just in case the driver uses namespaces
    cfg.CONF.register_opts(interface.OPTS)
    config.register_interface_driver_opts_helper(cfg.CONF)
    config.register_agent_state_opts_helper(cfg.CONF)
    config.register_root_helper(cfg.CONF)

    common_config.init(sys.argv[1:])
    config.setup_logging()

    mgr = manager.LbaasAgentManager(cfg.CONF)
    svc = LbaasAgentService(
        host=cfg.CONF.host,
        topic=topics.LOADBALANCER_AGENT,
        manager=mgr
    )
    service.launch(svc).wait()
```

其中，管理服务类为 agent_manager 中的 LbaasAgentManager 类。

LbaasAgentService 类是一个 rpc 服务类，定期执行管理服务类中的周期性任务。

```
class LbaasAgentService(n_rpc.Service):
    def start(self):
        super(LbaasAgentService, self).start()
        self.tg.add_timer(
            cfg.CONF.periodic_interval,
            self.manager.run_periodic_tasks,
            None,
            None
        )
```

agent_manager.py

主要就是定义了 LbaasAgentManager 类，继承自 n_rpc.RpcCallback 和 periodic_task.PeriodicTasks。

作为 agent 中的管理类。它实现了 LBaaS 中定义的各种操作 API，包括：

- agent_updated
- collect_stats
- create_member

- create_pool
- create_pool_health_monitor
- create_vip
- delete_member
- delete_pool
- delete_pool_health_monitor
- delete_vip
- initialize_service_hook
- periodic_resync
- remove_orphans
- sync_state
- update_member
- update_pool
- update_pool_health_monitor
- update_vip 这些操作又是通过调用更下一层的 driver 来实现。

同时，它定义了一个 rpc api 调用成员，向 topics.LOADBALANCER_PLUGIN 主题发送消息。

```
self.plugin_rpc = agent_api.LbaasAgentApi(
    topics.LOADBALANCER_PLUGIN,
    self.context,
    self.conf.host
)
```

agent_api.py

主要定义了 LbaasAgentApi 类，继承自 n_rpc.RpcProxy，也是一个典型的 RPC 代理类。

这个类在 LbaasAgentManager 类中使用，用来负责替 agent 向指定的主题发送消息，调用监听者的对应方法。主要包括：

- get_logical_device
- get_ready_devices
- plug_vip_port
- pool_deployed
- pool_destroyed
- unplug_vip_port
- update_pool_stats
- update_status

agent_device_driver.py

主要定义了 AgentDeviceDriver 类，是一个抽象类，抽象出来支持 LBaaS agent 所需要 API 的一个驱动类。这些 API 包括对 pool、member 等资源的 CRUD 等。

agent_scheduler.py

loadbalancer池和agent之间的绑定和调度维护。

constants.py

常量定义。

drivers

包括支持 LBaaS agent 中需要的 API 的一些驱动，包括 a10networks、embrane、haproxy、logging_noop、netscaler、radware 等等。

driver_base.py

定义了一系列的实现 driver 所需要的基础类，包括 BaseHealthMonitorManager、BaseListenerManager、BaseLoadBalancerManager BaseMemberManager、BasePoolManager、LoadBalancerBaseDriver。

driver_mixins.py

plugin.py

定义了该服务plugin的实现主类LoadBalancerPlugin。

metering

agents

drivers

metering_plugin.py

定义了metering服务plugin的实现主类MeteringPlugin。

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。

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。

tests

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。

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。

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。

i18n.py

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。

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	Read	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	

policy.py

quota.py

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。

Service 类

Service是很重要的一個概念，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。

version.py

从prb中获取version信息。

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

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 类，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。

rally-jobs

extra

README.rst

plugins

README.rst

init.py

neutron-neutron.yaml

README.rst

tools

包括跟安装和格式检查相关的一些工具。

check_bash.sh

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

clean.sh

清除代码编译中间结果。

i18n_cfg.py

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。

pretty_tox.sh

with_venv.sh

启用虚环境。

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。

调用逻辑

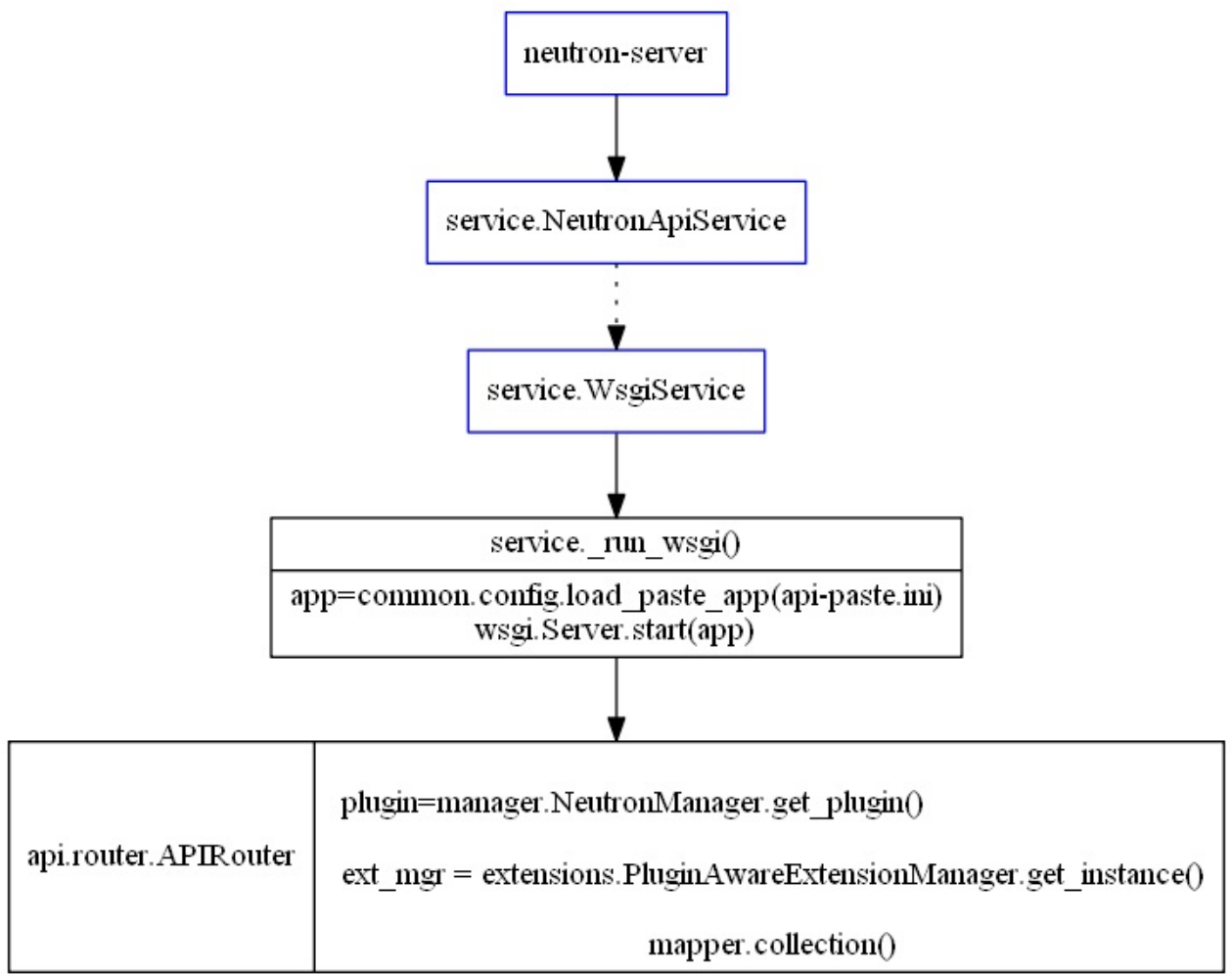
REST API 专题

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。



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。

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。

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的客户端，通过代码

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

该客户端实际上定义了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，实际上就是调用客户端的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。

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。

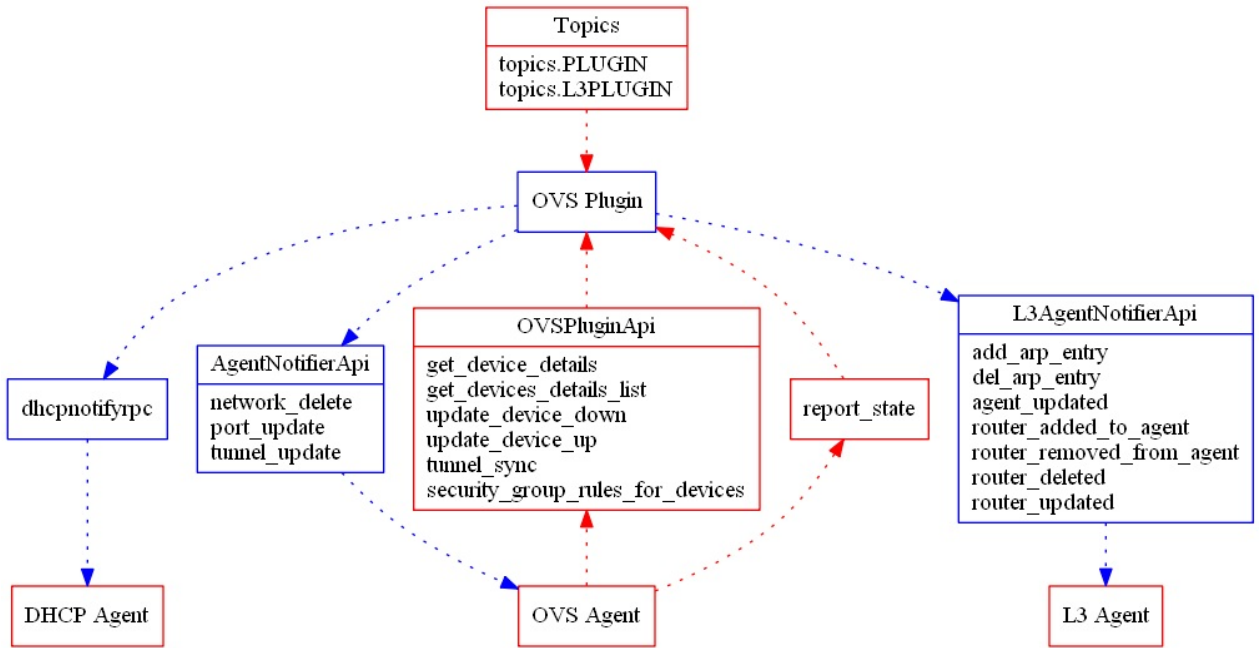
其代码为

```
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。



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()
```


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）。

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。

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。

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，并同时反馈信息给远端的 plugin。

agent 可以分为四种：核心 agent、dhcp agent、l3 agent、其它 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，还支持配置的调度策略。

其它 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。