

OpenStack Neutron 代码分析

最新版:

[https://github.com/yeasy/tech_writing/tree/master/OpenStack/OpenStack Neutron 代码分析.pdf](https://github.com/yeasy/tech_writing/tree/master/OpenStack/OpenStack%20Neutron%E4%BB%A3%E7%A0%B2%E6%9E%90%E5%8F%8C.pdf)

更新历史:

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。

1 源代码结构

源代码主要分为 5 个目录：

bin, doc, etc, neutron 和 tools。

1.1 bin

neutron-rootwrap

neutron-rootwrap-xen-dom0

quantum-rootwrap

quantum-rootwrap-xen-dom0

这四个文件都是提供利用 `root` 权限执行命令时候的操作接口，通过检查，可以仅允许用户在执行给定的命令。其主要实现是利用了 `oslo.rootwrap` 包中的 `cmd` 模块。

quantum-rpc-zmq-receiver。

1.2 doc

可以利用 `sphinx` 来生成文档。

source 子目录：文档相关的代码。

Makefile：用户执行 `make` 命令。

pom.xml：

1.3 etc

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

init.d/neutron-server：neutron-server 服务脚本。

neutron/：

plugins/：一些 plugin 相关的配置文件（*.ini），其中被注释掉的行表明了如果不指明情况下的是默认值。

rootwrap.d/：一些 filters 文件。

各种 ini 和 conf 文件。

1.4 neutron

核心的代码实现都在这个目录下。

可以通过下面的命令来统计主要实现代码量。

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

目前版本，约为 215k 行。

1.5 tools

一些相关的代码格式化检测、环境安装的脚本。

2 etc

2.1 init.d/

neutron-server 是系统服务脚本，核心部分为

```

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

```

2.2 neutron/

2.2.1 plugins

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

2.2.2 rootwrap.d

包括一系列的 filter 文件。包括 debug.filters

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

以 nova 用户为例，在/etc/sudoers.d/nova 文件中有

```
nova ALL = (root) NOPASSWD: /usr/bin/nova-rootwrap /etc/nova/rootwrap.conf *
```

使得 nova 可以以 root 权限运行 nova-rootwrap。而在 rootwrap.conf 中定义了 filters_path=/etc/nova/rootwrap.d, /usr/share/nova/rootwrap。这两个目录中定义的命令的 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。需要注意 /etc/nova/rootwrap.d, /usr/share/nova/rootwrap 必须是 root 权限才能修改。

2.2.3 api-paste.ini

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

在 neutron-server 启动的时候，一般会指定参数 `--config-file neutron.conf --config-file xxx.ini`。neutron/server/__init__.py: 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。比如下面的 api-paste.ini 信息中定义了 neutron、neutronapi_v2_0、若干 filter 和两个 app。

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.4 dhcp_agent.ini

`dhcp agent` 相关的配置信息。包括与 `neutron` 的同步状态的频率等。

2.2.5 fwaas_driver.ini

配置 `fwaas` 的 `driver` 信息，默认为

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

2.2.6 l3_agent.ini

`L3 agent` 相关的配置信息。

当存在外部网桥的时候，每个 `agent` 最多只能关联到一个外部网络。

2.2.7 lbaas_agent.ini

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

2.2.8 metadata_agent.ini

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

2.2.9 metering_agent.ini

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

2.2.10 vpn_agent.ini

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

2.2.11 neutron.conf

neutron-server 启动后读取的配置信息。

2.2.12 policy.json

配置策略。

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

目前支持的策略有三种：rule、role 或者 generic。

其中 rule 后面会跟一个文件名，例如

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

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

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

generic 策略则根据参数来进行比较。

2.2.13 rootwrap.conf

neutron-rootwrap 的配置文件。

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

2.2.14 services.conf

配置一些特殊的 service 信息。

3 neutron

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

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

3.1 agent

在 neutron 的架构中，各种 agent 运行在计算节点和网络节点上，接收来自 neutron-server 的指令，对所管理的网桥进行实际的操作，属于“直接干活”的部分。

本部分代码实现各种 agent 所需要的操作接口和库函数。

3.1.1 common/

主要包括 config.py，其中定义了 agent 的一些默认配置参数，和相应的配置函数。

3.1.2 linux/

主要包括跟 linux 环境相关的一些函数实现。

3.1.2.1 async_process.py

实现了 AsyncProcess 类，对异步进程进行管理。

3.1.2.2 daemon.py

实现一个通用的 Daemon 类。

3.1.2.3 dhcp.py

实现了 Dnsmasq 类、DhcpBase 类、DhcpLocalProcess 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。

3.1.2.4 external_process.py

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

3.1.2.5 interface.py

提供对网桥上的接口进行管理。

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、MetaInterfaceDriver、BridgeInterfaceDriver、IVSInterfaceDriver、MidonetInterfaceDriver、NullDriver 和 OVSIInterfaceDriver。

3.1.2.6 ip_lib.py

对 ip 相关的命令进行封装，包括一些操作类。例如 IpAddrCommand、IpLinkCommand、IpNetnsCommand 等。

3.1.2.7 iptables_firewall.py

利用 iptables 的规则实现防火墙，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。

OVSHybridIptablesFirewallDriver，继承自 IptablesFirewallDriver。

3.1.2.8 iptables_manager.py

对 iptables 规则进行管理，提供操作接口。

定义了 IptablesManager 类、IptablesRule 类、IptablesTable 类。

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

3.1.2.9 ovs_lib.py

提供对 OVS 网桥的操作支持，包括一个 OVSBridge 类、VifPort 类。

3.1.2.10 ovssdb_monitor.py

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

3.1.2.11 polling.py

监视 ovssdb 来决定何时进行 polling。包括一个 InterfacePollingMinimizer 类等。

3.1.2.12 utils.py

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

3.1.3 metadata/

agent.py

3.2 api

提供 RestAPI 访问。

3.3 cmd

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

3.4 common

3.4.1 config.py

对配置进行管理。

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

主要包括

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

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

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

3.4.2 constants.py

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

3.4.3 exceptions.py

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

3.4.4 ipv6_utils.py

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

3.4.5 log.py

基于 `neutron.openstack.common` 中的 `log` 模块。

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

3.4.6 rpc.py

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

3.4.7 test_lib.py

定义了 `test_config={}`，用于各个 plugin 进行测试。

3.4.8 topics.py

管理消息队列传递过程中的 topic 信息。

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

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

AGENT = 'q-agent-notifier'
PLUGIN = 'q-plugin'
L3PLUGIN = 'q-l3-plugin'
DHCP = 'q-dhcp-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'
```

3.4.9 utils.py

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

3.5 db

数据库相关操作的实现。

3.6 debug

测试功能。

`commands.py`

debug_agent.py

shell.py

3.7 extensions

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

3.8 locale

多语言支持。

3.9 notifiers

3.10 openstack

公共模块。

3.11 plugins

包括实现网络功能的各个插件。

3.11.1 bigswitch

3.11.2 brocade

3.11.3 cisco

3.11.4 common

3.11.5 embrane

3.11.6 hyperv

3.11.7 ibm

3.11.7.1 agent/

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

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

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

    plugin = SdnveNeutronAgent(**agent_config)

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

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

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

这三行则初始化配置信息。

其中最关键的 `cfg.CONF(project='neutron')`是个函数调用，实际上调用了 `cfg.ConfigOpts` 类的 `_call_` 方法。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])`）。

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

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

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

3.11.7.2 common/

这里面的文件主要是定义一些常量。

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

`constants.py` 则分别定义了一些固定的常量；
`exceptions.py` 中定义了一些异常类型。

3.11.7.3 sdnve_api.py

封装 `sdnve` 控制器所支持的操作为一些 API。

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

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

KeystongClient 类，主要是获取系统中的租户信息。

3.11.7.4 sdnve_neutron_plugin.py

SdnvePluginV2 类，继承自如下几个基础类：

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

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

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

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

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

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

- 3.11.8 linuxbridge**
- 3.11.9 metaplugin**
- 3.11.10 midonet**
- 3.11.11 ml2**
- 3.11.12 mlnx**
- 3.11.13 nec**
- 3.11.14 nicira**
- 3.11.15 nuage**
- 3.11.16 ofagent**
- 3.11.17 oneconvergence**
- 3.11.18 openvswitch**
- 3.11.19 plumgrid**
- 3.11.20 ryu**
- 3.11.21 vmware**

3.12 scheduler

调度、负载均衡等。

- 3.12.1 dhcp-agent_scheduler.py**
- 3.12.2 l3-agent_scheduler.py**

3.13 server

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

主要过程为：

```
eventlet.monkey_patch()
```

绿化各个模块为支持协程。

```
config.parse(sys.argv[1:])
```

通过解析命令行传入的参数，获取配置文件所在。

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

创建基于协程的线程池。

```
neutron_api = service.serve_wsgi(service.NeutronApiService)
```

```
api_thread = pool.spawn(neutron_api.wait)
```

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

```
try:
```

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

```
except NotImplementedError:
```

```
    LOG.info(_("RPC was already started in parent process by plugin."))
```

```
else:
```

```
    rpc_thread = pool.spawn(neutron_rpc.wait)
```

```
    rpc_thread.link(lambda gt: api_thread.kill())
```

```
    api_thread.link(lambda gt: rpc_thread.kill())
```

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

```
pool.waitall()
```

最后是后台不断等待。

3.14 services

3.15 tests

3.16 其他文件

3.16.1 auth.py

3.16.2 context.py

3.16.3 hooks.py

3.16.4 manager.py

3.16.5 neutron_plugin_base_v2.py

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

包括下面的方法：

属性	create	delete	get	update
port	Y	Y	Y	Y
ports			Y	
ports_count			Y	
network	Y	Y	Y	Y
networks			Y	
networks_count			Y	
subnet	Y	Y	Y	Y
subnets			Y	
subnet_count			Y	

3.16.6 policy.py

3.16.7 quota.py

3.16.8 service.py

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

实现 `neutron` 中跟服务相关的类。

包括 `NeutronApiService`，`RpcWorker`，`Service` 和 `WsgiService`。

`WsgiService` 是实现基于 `WSGI` 的服务的基础类。

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

3.16.9 version.py

3.16.10 wsgi.py

