nova读取配置文件流程

在我们安装nova的过程中，设置它的配置文件/etc/nova/nova.conf是必不可少的一步。配置好nova.conf文件，nova-compute、nova-network等服务才有可能正常启动。当然在修改nova.conf文件后，必须重启nova的所有服务，新的配置才能生效。

 其实我们的配置文件不一定非要存放为/etc/nova/nova.conf，也可以设置成任意目录下的任意文件。nova获取配置文件的方式有两种：

1、通过命令行选项--config-file指定任意文件为配置文件（当然前提是有读写权限）。

例如： /usr/bin/python /usr/bin/nova-compute --config-file=/etc/nova/nova.conf --config-file=/etc/nova/nova-compute.conf

通过--config-file选项，可以指定任意多个配置文件，上例给nova-compute服务指定了两个配置文件。此时，在这两个文件设置的配置项，都是有效的。但如果这两个文件有相同的配置项，而且这个配置项是不允许有多个值的，那会以哪个为有效指呢，还是会出现错误？

2、如果在服务启动时，没有通过--config-file指定配置文件，那么nova会在几个特定的目录按照先后顺序下寻找配置文件。在每个服务启动时，会按照以下顺序寻找配置文件：

[~/.nova/, ~/, /etc/nova/, /etc/]

例如，启动nova-compute时，它会在这些目录下分别寻找nova.conf和nova-compute.conf，如果～/.nova/nova.conf存在，那么就不会再寻找其它目录下的nova.conf了。对于nova-compute.conf文件也是一样。

nova配置文件内容格式如下：

[DEFAULT]

dhcpbridge\_flagfile=/etc/nova/nova.conf

dhcpbridge=/usr/bin/nova-dhcpbridge

libvirt\_use\_virtio\_for\_bridges=True

connection\_type=libvirt

...…

分析

 下文用于分析nova组件是如何读取命令行和文件配置，从分析流程中我们可以知道：

1）nova是如何寻找配置文件，配置文件存放在哪个目录下才能被它读取

2）配置文件的编辑应该遵守什么样的格式，才能被nova正常读取

3）当我们需要在配置文件中添加一个我们自己的选项时，我们应该在哪些地方进行如何修改，才能被nova获取到我们写入配置文件的值

nova有多个服务：nova-cert、nova-consoleauth、nova-manage、nova-novncproxy、nova-scheduler、nova-api、nova-compute、nova-dhcpbridge、nova-network、nova-rootwrap。各个服务在启动时，读取配置文件的流程都是一样的。下面针对nova-compute的配置读取过程进行分析，其他的服务相同。

compute启动脚本

/usr/bin/nova-compute文件：

 42 if \_\_name\_\_ == '\_\_main\_\_':

 43 flags.parse\_args(sys.argv)

 44 logging.setup('nova')

 45 utils.monkey\_patch()

 46 server = service.Service.create(binary='nova-compute')

 47 service.serve(server)

 48 service.wait()

第42行调用flags.parse\_args(sys.argv)函数，该函数定义于nova/flags.py。它调用CommonConfigOpts实例进行命令行参数和配置文件的读取：

nova/flags.py:

 36 FLAGS = cfg.CONF

 37

 38

 39 def parse\_args(argv, default\_config\_files=None):

 40 FLAGS.disable\_interspersed\_args()

 41 return argv[:1] + FLAGS(argv[1:],

 42 project='nova',

 43 default\_config\_files=default\_config\_files)

继续查看cfg.CONF

/nova/openstack/common/cfg.py

1654 CONF = CommonConfigOpts()

由上面代码可知，nova-compute后面的命令行参数直接调用CommonConfigOpts对象处理，并设project=‘nova’，默认配置文件为空。CommonConfigOpts乃何许类也，它都干些什么呢？

CommonConfigOpts是ConfigOpts的子类，而在ConfigOpts中定义\_\_call\_\_函数，所以CommonConfigOpts对象可以像函数一样，被直接调用。为了能够理解这两个类是干什么的，我们需要先了解另外几个类：

optparse.OptionParser

optparser库中的OptionParser，用来解析通过命令行指定的参数，该库目前不再更新，将会被argparser代替。但openstack使用该库来解析，所以做个简单介绍：

from optparse import OptionParser

parser = OptionParser()

parser.add\_option(-f, --file, dest=filename, #dest用来明确指定解析参数后对应的属性名，

 #如果没有指定则为file

help=write report to FILE, metavar=FILE)

parser.add\_option(-q, --quiet,

action=store\_false, dest=verbose, default=True,

help=don't print status messages to stdout)

(options, args) = parser.parse\_args(['-f', 'test.txt', 'ddd', 'ffff'])

该脚本执行过后，options.filename=test.txt、options.verbose=True。args=['ddd', 'ffff']，它是用于返回不能被OptionParser识别的参数。

optparse.OptionGroup

当参数较多时，可利用OptionGroup对其进行分类，这样在输出help信息时，能够分类整齐些，所以每一个OptionGroup都必须与一个OptionParser关联（多对一的关系）。在使用上，与OptionParser差不多。如下：

group = OptionGroup(parser, Dangerous Options, #其中parser为上面定义的类

Caution: use these options at your own risk.

It is believed that some of them bite.)

group.add\_option(-g, action=store\_true, help=Group option.)

parser.add\_option\_group(group)

(options, args) = parser.parse\_args(['-f', 'test.txt', 'ddd', 'ffff'])

该option和args的值与上例中结果一样。

openstack.common.cfg.Opt:

self.name = name #name对应命令行参数名和配置文件中的配置项名

self.dest = dest#解析命令行参数时生成属性的名，通过该名可获取命令行参数值，如上例filename

self.short = short#命令行参数的简写，如上例中f

self.default = default#该选项的默认值

self.metavar = metavar#用于在显示帮助信息时用的，如FILE

self.help = help

self.secret = secret #bool类型，指定该选项是否为保密信息，这样在打印日志时会用特殊字符代替

self.required = required #是否为必填项，在读取配置文件后，会依据该值进行检查

self.deprecated\_name = None#该选项的备用名，可用作命令行参数名和配置项名

ConfigParser(iniparser.BaseParser):

self.filename = filename

self.sections = sections

def parse(self):

with open(self.filename) as f:

return super(ConfigParser, self).parse(f)

对于每一个配置文件，都会生成一个ConfigParser对象。该类调用BaseParser.parse()函数，将文件内容解析成key value对，定义如下：

 def parse(self, lineiter):

key = None

value = []

for line in lineiter:

self.lineno += 1

line = line.rstrip()

if not line:

# Blank line, ends multi-line values

if key:

key, value = self.\_assignment(key, value)

continue

elif line[0] in (' ', '\t'):

# Continuation of previous assignment，这里可以知道，为什么配置项不能以空格开始了

if key is None:

self.error\_unexpected\_continuation(line)

else:

value.append(line.lstrip())

continue

if key:

# Flush previous assignment, if any

key, value = self.\_assignment(key, value)

if line[0] == '[':

# Section start

section = self.\_get\_section(line)

if section:

self.new\_section(section)

elif line[0] in '#;':#这里可以看出配置项的key和value可以通过#或;进行分隔

self.comment(line[1:].lstrip())

else:

key, value = self.\_split\_key\_value(line)

if not key:

return self.error\_empty\_key(line)

if key:#用作处理最后一个key value对

# Flush previous assignment, if any

self.\_assignment(key, value)

从上面的代码可知，ConfigParser读取配置文件，将每个配置项以{key：value}储存在sections中{‘section1’: {‘key11’: value}}。在配件文件中，每个配置项必须处于一个section之中，即在一个通过[section]定义的行之下。否则会报parse\_exc异常。这样，一个配置文件中所有配置项都被读到了ConfigParser的sections中。

class MultiConfigParser(object):

def \_\_init\_\_(self):

self.parsed = []

def read(self, config\_files):

read\_ok = []

for filename in config\_files:

sections = {}

parser = ConfigParser(filename, sections)

try:

parser.parse()

except IOError:

continue

self.parsed.insert(0, sections)

read\_ok.append(filename)

return read\_ok

def get(self, section, names, multi=False):

rvalue = []

for sections in self.parsed:

if section not in sections:

continue

for name in names:

if name in sections[section]:

if multi:

rvalue = sections[section][name] + rvalue

else:

return sections[section][name]

if multi and rvalue != []:

return rvalue

raise KeyError

该类的read函数，为每个配置文件初始化一个ConfigPaser对象，并将该对象解析过后的sections存放到parsed只中，这样每个配置文件就对应了parsed 列表中的一项，所有配置文件的配置项都存放在parsed成员变量中。

这里需要注意一点，在往parsed中添加sections时，使用parsed.insert(0, sections)。排在后面的配置文件，解析后会放在parsed的最前面。当调用get()获取配置项的时，如果该配置项不为多选项（multi），那么只会去parsed前面sections中的值。也就是说如果一个配置项在多个配置文件中被重复定义，那么只会读取最后一个配置文件中的值作为使用值。

好了，在了解了上面几个类之后，我们可以查看ConfigOpts和CommonConfigOpts类了。ConfigOpts的成员变量如下：

ConfigOpts(collections.Mapping):

/nova/openstack/common/cfg.py

def \_\_init\_\_(self):

Construct a ConfigOpts object.

self.\_opts = {}# dict of dicts of (opt:, override:, default:)

self.\_groups = {}

self.\_args = None

self.\_oparser = None#命令行参数解释器，即一个OptionParser对象

self.\_cparser = None#配置文件解释器，即一个MultiConfigParser对象

self.\_cli\_values = {} #程序启动时，通过\_oparser解释的命令行参数值

self.\_\_cache = {} #缓存

self.\_config\_opts = []

self.\_disable\_interspersed\_args = False

def \_\_call\_\_(self,

 args=None,

 project=None,

 prog=None,

 version=None,

 usage=None,

 default\_config\_files=None):

self.clear()

self.\_setup(project, prog, version, usage, default\_config\_files)

self.\_cli\_values, leftovers = self.\_parse\_cli\_opts(args)

self.\_parse\_config\_files()

self.\_check\_required\_opts()

return leftovers

clear()函数：

清空opts、groups、cache等成员变量，对于刚初始化的ConfigOpts对象，基本什么也没干。

\_setup函数：

def \_setup(self, project, prog, version, usage, default\_config\_files):

Initialize a ConfigOpts object for option parsing.

if prog is None:

prog = os.path.basename(sys.argv[0])

if default\_config\_files is None:

#在['~/.nova/', '~/', '/etc/nova/','/etc/'] 寻找配置文件nova.conf和nova-compute.conf

default\_config\_files = find\_config\_files(project, prog)

self.\_oparser = optparse.OptionParser(prog=prog,

version=version,

usage=usage)

if self.\_disable\_interspersed\_args:

self.\_oparser.disable\_interspersed\_args()

self.\_config\_opts = [

MultiStrOpt('config-file',

default=default\_config\_files,

metavar='PATH',

help='Path to a config file to use. Multiple config '

 'files can be specified, with values in later '

 'files taking precedence. The default files '

 ' used are: %s' % (default\_config\_files, )),

StrOpt('config-dir',

 metavar='DIR',

 help='Path to a config directory to pull \*.conf '

'files from. This file set is sorted, so as to '

'provide a predictable parse order if individual '

'options are over-ridden. The set is parsed after '

'the file(s), if any, specified via --config-file, '

'hence over-ridden options in the directory take '

'precedence.'),

]

#注册命令行参数config-file和config-dir，这样就可以通过命令行指定该配置项了。

self.register\_cli\_opts(self.\_config\_opts)

self.project = project

self.prog = prog

self.version = version

self.usage = usage

self.default\_config\_files = default\_config\_files

1、寻找配置文件，如果\_\_call\_\_的参数default\_config\_files=None。那么它将按一下顺序~/.nova/, ~/, /etc/nova/, /etc/搜索配置文件nova.conf和nova-compute.conf，对于每个配置文件只要在一个目录下搜索到，剩余的目录就不会被搜索。并将default\_config\_files赋值为搜索的结果，

2、用命令行参数config-file、config-dir初始化命令行解释器：

self.\_oparser = optparse.OptionParser(prog=prog, version=version,usage=usage)

这样就能解析通过命令行传递参数config-file、config-dir的值。

\_parse\_cli\_opts函数：

用于解析通过命令行传过来的参数：它先调用Opt.\_add\_to\_cli将所有的opts全都注册到self.\_oparser中。再调用self.\_oparser.parse\_args(args)解析所有的命令函数。然后将解析结果存放到self.\_cli\_values中。

此时命令行参数已经解析完成，然后解析配置文件中的选项。

\_parse\_config\_files函数：

Parse the config files from --config-file and --config-dir.

:raises: ConfigFilesNotFoundError, ConfigFileParseError

config\_files = list(self.config\_file)

if self.config\_dir:#如果指定了config\_dir，那么该目录下所有匹配\*.conf的文件，都会被当作配置文件

config\_dir\_glob = os.path.join(self.config\_dir, '\*.conf')

config\_files += sorted(glob.glob(config\_dir\_glob))

config\_files = [\_fixpath(p) for p in config\_files]

self.\_cparser = MultiConfigParser()

try:

read\_ok = self.\_cparser.read(config\_files)

except iniparser.ParseError as pe:

raise ConfigFileParseError(pe.filename, str(pe))

if read\_ok != config\_files:

not\_read\_ok = filter(lambda f: f not in read\_ok, config\_files)

raise ConfigFilesNotFoundError(not\_read\_ok)

在这个函数中，self.\_cparser被初始化：

self.\_cparser = MultiConfigParser()

如果通过命令行参数--config-file指定配置文件，和--config-dir指定配置目录。那这些文件全都会通过

read\_ok = self.\_cparser.read(config\_files)

解析后的选项和值都被存放到列表self.\_cparser.parsed中，其中每一元素对应一个配置文件，元素结构是：

sections[section][key]=value

\_check\_required\_opts函数：

根据每个opt的requred属性，判断是否所有必设置选项是否被设置，如果没有，则报异常。

class CommonConfigOpts(ConfigOpts):

CommonConfigOpts只是重新定义了构造函数，在初始化对象时，注册了几个Opt对象：-d\-v\--log-config\--log-format\--log-date-format\--log-file\--log-dir\--use-syslog\--syslog-log-facility。这样就可以通过命令行参数对这些配置项进行设置。

到此，命令行参数和配置文件参数都已解析完成，所有的命令行参数和配置文件内容都保存到FLAGS中。

获取配置项的值

 def \_do\_get(self, name, group=None):

if group is None and name in self.\_groups:

return self.GroupAttr(self, self.\_get\_group(name))

#首先要获取该对象的注册信息，所以使用前必须先注册

info = self.\_get\_opt\_info(name, group)

opt = info['opt']

if 'override' in info:

return info['override']

values = []

if self.\_cparser is not None:

section = group.name if group is not None else 'DEFAULT'

try:

value = opt.\_get\_from\_config\_parser(self.\_cparser, section)

except KeyError:

pass

except ValueError as ve:

raise ConfigFileValueError(str(ve))

else:

if not opt.multi:

# No need to continue since the last value wins

return value[-1]

values.extend(value)

name = name if group is None else group.name + '\_' + name

value = self.\_cli\_values.get(name)

if value is not None:

if not opt.multi:

return value

return value + values

if values:

return values

if 'default' in info:

return info['default']

return opt.default

从该流程中，可以看出。获取一个配置项的值，首先得注册该配置项。获取该配置项的值的优先级如下：

info['override']，配置文件，命令行参数、info['default']、Opt['default']

注册一个配置项流程如下：

compute\_opts = [

cfg.StrOpt('instances\_path',

 default='$state\_path/instances',

 help='where instances are stored on disk'),

cfg.IntOpt('live\_migration\_retry\_count',

 default=30,

 help=Number of 1 second retries needed in live\_migration),

]

FLAGS = flags.FLAGS

FLAGS.register\_opts(compute\_opts)

然后就可以如下进行访问了

retry = FLAGS.live\_migration\_retry\_count

StrOpt、IntOpt、BoolOpt、FloatOpt、ListOpt是Opt的子类，它们主要是重定义了获取值和注册OptionParser的函数，以IntOpt为例：

class IntOpt(Opt):

Int opt values are converted to integers using the int() builtin.

def \_get\_from\_config\_parser(self, cparser, section):

Retrieve the opt value as a integer from ConfigParser.

return [int(v) for v in self.\_cparser\_get\_with\_deprecated(cparser,

section)]

def \_get\_optparse\_kwargs(self, group, \*\*kwargs):

Extends the base optparse keyword dict for integer options.

return super(IntOpt,

 self).\_get\_optparse\_kwargs(group, type='int', \*\*kwargs)

这样，特定类型的Opt在获取配置文件中的值和通过命令行参数获得的值，都能够自动转换成相应的类型，方便使用。