Table of Contents

OpenStack Heat 源码分析

Heat 是 OpenStack 项目中负责部署、协调资源的平台工具，它允许用户提前写好一个模板文件，然后通过 Heat 引擎来启动、停止所定义的资源。

本书将剖析 Heat 组件（服务端）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_Heat。

感谢所有的 贡献者。

更新历史:

- 0.2: 2014-08-18
  - 完成对整体代码的分析。
- 0.1: 2014-08-09
  - 完成代码基本结构分析。

参加步骤

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

```
$ git clone git@github.com:user/openstack_code_Heat.git
$ cd openstack_code_Heat
$ 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_Heat
$ 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 6 个目录和若干文件： bin，contrib，doc，etc，neutron和tools。

除了这 6 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 heat-api、heat-api-cfn、heat-api-cloudwatch、heat-engine、heat-manage。 这些是相应服务的主文件。

contrib

包括若干第三方的插件，包括 docker、extraroute、keystoneclient_v2、marconi和rackspace等。

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。

- environment.d/default.yaml 文件中可以预定义一些环境参数。
- templates 目录下有两个 yaml 格式的模板文件。
- api-paste.ini，启动 heat-api 服务时候的 paste-deploy 配置文件。

heat

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

目前版本，约为 107k 行。

tools

一些相关的代码格式化检测、环境安装的脚本。

heat-api

启动 heat-api 服务的主文件。

主要代码为

```
cfg.CONF(project='heat', prog='heat-api')
cfg.CONF.default_log_levels = ['amqplib=WARN',
                               'qpid.messaging=INFO',
                               'keystone=INFO',
                               'eventlet.wsgi.server=WARN',
                               ]
logging.setup('heat')

app = config.load_paste_app()

port = cfg.CONF.heat_api.bind_port
host = cfg.CONF.heat_api.bind_host
LOG.info('Starting Heat ReST API on %s:%s' % (host, port))
server = wsgi.Server()
server.start(app, cfg.CONF.heat_api, default_port=port)
notify.startup_notify(cfg.CONF.onready)
server.wait()
```

可见，其主要过程为，从 api-paste.ini 文件中读取应用并加载，之后启动一个 wsgi 服务，上面运行配置好的应用，并根据配置文件指定，发出启动完成的通知。

heat-api-cfn

类似的，该文件启动 heat-api-cfn 服务，提供 AWS 风格的 rest 请求，并处理请求并通过 RPC 调用发送给 heat-engine。

heat-api-cloudwatch

类似的，该文件启动 heat-api-cloudwatch 服务，为 heat cloudwatch 提供 API 服务。

heat-engine

启动 heat-engine 服务，其过程如下

```
cfg.CONF(project='heat', prog='heat-engine')
logging.setup('heat')
messaging.setup()

from heat.engine import service as engine

srv = engine.EngineService(cfg.CONF.host, rpc_api.ENGINE_TOPIC)
launcher = service.launch(srv, workers=cfg.CONF.num_engine_workers)
# We create the periodic tasks here, which mean they are created
# only in the parent process when num_engine_workers>1 is specified
srv.create_periodic_tasks()
notify.startup_notify(cfg.CONF.onready)
launcher.wait()
```

其主要通过调用 engine.EngineService 类来实现应用，用 service 包来管理服务。 heat-engine 作为 heat-api 的后端，是 heat 中最核心的组件。

heat-keystone-setup-domain

配置 heat domain。

heat-manage

调用 heat.cmd.manage 包中的 main 方法。响应 heat-manage 命令。

cinder-keystone-setup

Cinder 相关认证信息。

heat-db-setup

shell 脚本，安装和启动一个 mysql 服务器来作为 Heat 的数据库后端。

heat-keystone-setup

shell 脚本，配置 heat 在 keystone 中的用户信息。

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。
- docbkx目录：api 参考和 Heat 管理文档。

etc

environment.d

主要包括 default.yaml，其中可以预定义一些环境参数。Heat 也可以通过传入参数和加载资源中定义来载入参数映射。

templates

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。

```
# heat-api pipeline
[pipeline:heat-api]
pipeline = request_id faultwrap ssl versionnegotiation osprofiler authurl authtoken co
ntext apiv1app

# heat-api pipeline for standalone heat
# ie. uses alternative auth backend that authenticates users against keystone
# using username and password instead of validating token (which requires
# an admin/service token).
# To enable, in heat.conf:
#   [paste_deploy]
#   flavor = standalone
#
[pipeline:heat-api-standalone]
pipeline = request_id faultwrap ssl versionnegotiation authurl authpassword context ap
iv1app

# heat-api pipeline for custom cloud backends
# i.e. in heat.conf:
#   [paste_deploy]
#   flavor = custombackend
#
[pipeline:heat-api-custombackend]
pipeline = request_id faultwrap versionnegotiation context custombackendauth apiv1app

# heat-api-cfn pipeline
[pipeline:heat-api-cfn]
pipeline = cfnversionnegotiation osprofiler ec2authtoken authtoken context apicfnv1app

# heat-api-cfn pipeline for standalone heat
# relies exclusively on authenticating with ec2 signed requests
[pipeline:heat-api-cfn-standalone]
pipeline = cfnversionnegotiation ec2authtoken context apicfnv1app

# heat-api-cloudwatch pipeline
[pipeline:heat-api-cloudwatch]
pipeline = versionnegotiation osprofiler ec2authtoken authtoken context apicwapp

# heat-api-cloudwatch pipeline for standalone heat
# relies exclusively on authenticating with ec2 signed requests
[pipeline:heat-api-cloudwatch-standalone]
pipeline = versionnegotiation ec2authtoken context apicwapp

[app:apiv1app]
```

```
paste.app_factory = heat.common.wsgi:app_factory
heat.app_factory = heat.api.openstack.v1:API

[app:apicfnv1app]
paste.app_factory = heat.common.wsgi:app_factory
heat.app_factory = heat.api.cfn.v1:API

[app:apicwapp]
paste.app_factory = heat.common.wsgi:app_factory
heat.app_factory = heat.api.cloudwatch:API

[filter:versionnegotiation]
paste.filter_factory = heat.common.wsgi:filter_factory
heat.filter_factory = heat.api.openstack:version_negotiation_filter

[filter:faultwrap]
paste.filter_factory = heat.common.wsgi:filter_factory
heat.filter_factory = heat.api.openstack:faultwrap_filter

[filter:cfnversionnegotiation]
paste.filter_factory = heat.common.wsgi:filter_factory
heat.filter_factory = heat.api.cfn:version_negotiation_filter

[filter:cwversionnegotiation]
paste.filter_factory = heat.common.wsgi:filter_factory
heat.filter_factory = heat.api.cloudwatch:version_negotiation_filter

[filter:context]
paste.filter_factory = heat.common.context:ContextMiddleware_filter_factory

[filter:ec2authtoken]
paste.filter_factory = heat.api.aws.ec2token:EC2Token_filter_factory

[filter:ssl]
paste.filter_factory = heat.common.wsgi:filter_factory
heat.filter_factory = heat.api.openstack:sslmiddleware_filter

# Middleware to set auth_url header appropriately
[filter:authurl]
paste.filter_factory = heat.common.auth_url:filter_factory

# Auth middleware that validates token against keystone
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_factory

# Auth middleware that validates username/password against keystone
[filter:authpassword]
paste.filter_factory = heat.common.auth_password:filter_factory

# Auth middleware that validates against custom backend
[filter:custombackendauth]
paste.filter_factory = heat.common.custom_backend_auth:filter_factory
```

```
# Middleware to set x-openstack-request-id in http response header
[filter:request_id]
paste.filter_factory = oslo.middleware.request_id:RequestId.factory

[filter:osprofiler]
paste.filter_factory = osprofiler.web:WsgiMiddleware.factory
hmac_keys = SECRET_KEY
enabled = yes
```

以heat-api服务为例，会最终找到：

```
[app:apiv1app]
paste.app_factory = heat.common.wsgi:app_factory
heat.app_factory = heat.api.openstack.v1:API
```

其中，heat.common.wsgi:app_factory 实际上根据传入参数进一步加载应用。

```
def __call__(self, global_conf, **local_conf):
        """The actual paste.app_factory protocol method."""
        factory = self._import_factory(local_conf)
        return factory(self.conf, **local_conf)
```

运行时，

```
global={'__file__': '/usr/share/heat/api-paste-dist.ini', 'here': '/usr/share/heat'}
local={'heat.app_factory': 'heat.api.openstack.v1:API'}
```

所以实际上加载的还是 heat.api.openstack.v1.API。 而 heat.api.openstack.v1:API 中定义了对各种资源（包括 stack、resource、event、action、info、software config、software deployment 等）进行操作的 URL 所对应的处理方法。经过简单处理，后会发出对应的 rpc 消息给 heat-engine。

heat.conf.sample

样例配置程序，是 Heat 服务的默认主配置文件。

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。 该文件在开头定义

```
"context_is_admin":  "role:admin",
"deny_stack_user": "not role:heat_stack_user",
"deny_everybody": "!",
```

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

```
"cloudformation:ListStacks": "rule:deny_stack_user",
```

其策略为 rule:deny_stack_user，表明仅限 heat_stack_user 角色使用。 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。

heat

该目录下包含了 Heat 实现的主要代码。

api

因为 Heat 面向的操作对象是可扩展的，包括 openstack 资源、aws 等等，所以 api 也分为不同的类型，以备后面仅限扩展。 heat-api 以服务形式存在，接受 heatclient 传递过来的 rest 请求，从中读取模板信息，重新整理请求后以 rpc 的方式发送给 heat-engine。

aws

ec2token.py

定义EC2Token类，利用keystone来验证一个EC2请求，并将它转化为token。

exception.py

定义处理api的异常类。

utils.py

一些辅助方法，包括提取参数等。

cfn

v1

signal.py

定义SignalController类。

stacks.py

定义StackController类，作为stack资源的WSGI控制器。

versions.py

返回版本信息的控制器类。

cloudwatch

init.py

定义了继承自wsgi.Router的API类。

versions.py

定义返回版本信息的控制器类。

watch.py

定义WatchController类，包含有一个EngineClient句柄，发送请求给heat-engine。

middleware

fault.py

定义了Fault类和FaultWrapper类。

ssl.py

定义了SSLMiddleware类。

version_negotiation.py

定义了VersionNegotiationFilter类。

openstack

v1

views

定义了格式化输出的一些方法。

init.py

里面定义了一个很重要的类：API，这个类直接接收 restful 的 api 调用，它里面带有一个StackController类句柄，将不同的 url 请求映射为对应资源的方法。例如

```
stacks_resource = stacks.create_resource(conf)
with mapper.submapper(controller=stacks_resource,
                              path_prefix="/{tenant_id}") as stack_mapper:
stack_mapper.connect("template_validate",
                                 "/validate",
                                 action="validate_template",
                                 conditions={'method': 'POST'})
```

会将url到/tenant_id/validate的POST请求交给stacks_resource对象的validate_template方法处理，即调用stacks.StackController类的validate_template方法。 该类中分别定义了对Stack资源、Resource资源、Event资源、Action资源、Info资源、Software config资源、Software deployment资源的相关http 请求的处理，分别映射到该包中对应模块下的控制器上。 这些控制器根据收到的请求调用自身的对应方法，这些方法调用自身的engineclient（调用 messaging 中的 rpc client）来发出 rpc 请求调用 engine 中的方法，主题为 api.ENGINE_TOPIC。

actions.py

定义ActionController类，响应对Action资源的请求，调用自身的engineclient进行stack的暂停和继续操作。

build_info.py

定义 BuildInfoController 类，响应对 BuildInfo 资源的请求，调用自身的 engineclient 进行获取 buildinfo 操作。

events.py

定义 EventController类，响应对 Event 资源的请求，调用自身的 engineclient 进行列出和获取操作。

resources.py

software_configs.py

stacks.py

util.py

versions.py

响应对 api 版本请求，返回 http response。

cloudinit

loguserdata.py

将用户数据写到日志。

part_handler.py

boothook.sh

config

cmd

manager.py

支持一些用户命令，包括 db_version、db_sync、purge_deleted 等。

common

auth_password.py

定义 KeystonePasswordAuthProtocol 类，提供通过用户名密码来进行认证的方法。

auth_url.py

定义 AuthUrlFilter类。

config.py

定义配置文件中使用的配置项关键字和默认值。以及 load_paste_app() 等方法。

context.py

定义 RequestContext 类，请求的上下文信息。

crypt.py

加密和解密的相关方法。

custom_backend_auth.py

定义 AuthProtocol 类。

environment_format.py

对环境字符串进行解析。

exception.py

各种异常类。

heat_keystoneclient.py

定义 KeystoneClient 类和 KeystoneClientV3 类。封装 keystoneclient，以供资源使用。

identifier.py

定义 HeatIdentifier 类。

messaging.py

定义一些消息的序列化处理。

param_utils.py

解析参数辅助函数。

plugin_loader.py

定义动态从指定包加载所有模块的方法 load_modules。

policy.py

定义 Enforcer 类，加载和实施规则。

serializers.py

json 和 xml 回复的序列化类。

short_id.py

生成 id 方法。

template_format.py

一些模板格式化的方法。

timeutils.py

对时间进行处理的方法，包括转化为ISo 8601格式等。

urlfetch.py

从 URL 中获取数据的方法。

wsgi.py

wsgi 相关的封装类，包括请求、路由器、服务器等。

db

sqlalchemy

api.py

对数据库进行操作的API。

sync.py

同步数据库。

utils.py

数据库辅助方法。

doc

engine

cfn

functions.py

一些继承自 function.Function 的类，代表模板中的各种方法。

template.py

定义 CfnTemplate 类，代表一个 cfn 的 stack 模板。

hot

functions.py

定义模板中函数相关的解析方法。

parameters.py

定义 HOTParameters 类和 HOTParamSchema 类，解析模板中的参数。

template.py

定义 HOTemplate，代表一个模板。

clients

os

包括对 OpenStack 中各种服务 client 的插件，例如 NeutronClientPlugin、NovaClientPlugin 等。

init.py

定义了 OpenStackClients，来创建和使用各种 client。

client_plugin.py

定义 ClientPlugin 类，抽象基础元类，包含各种异常类。

notification

init.py

autoscaling.py

定义 send 方法。

stack.py

定义 send 方法。

resources

这里面定义了对应 template 中各种资源的具体类，一般都是继承自 resource.Resource 类。这些类在最后会定义一个 resource_mapping 方法，返回 template 的某个关键词和对应类的映射。 例如 instance 模块中定义：

```
def resource_mapping():
    return {
        'AWS::EC2::Instance': Instance,
        'OS::Heat::HARestarter': Restarter,
    }
```

同时，每个资源都定义了一套标准的 Action（例如创建、更新、删除等），均以 handle_ 为前缀。

init.py

在导入包的时候，会创建 plugin_manager 并从所有资源中导入可用的映射信息。 其中，定义的 initialise 方法在 heat-engine 服务启动的时候会被调用。 initialise 方法同时调用了 _load_global_environment 方法，该方法一个是调用 _load_global_resources() 来加载和注册资源，一个是调用 read_global_environment() 来从指定的环境配置目录下查找环境配置文件来载入环境配置信息。

```
def _load_global_environment(env):
    _load_global_resources(env)
    environment.read_global_environment(env)
```

_load_global_resources() 方法则负责加载和注册资源以及限制信息，这些信息都被添加到全局的 _environment 环境字典中。

```
def _load_global_resources(env):
    _register_constraints(env, _get_mapping('heat.constraints'))

    manager = plugin_manager.PluginManager(__name__)
    # Sometimes resources should not be available for registration in Heat due
    # to unsatisfied dependencies. We look first for the function
    # 'available_resource_mapping', which should return the filtered resources.
    # If it is not found, we look for the legacy 'resource_mapping'.
    resource_mapping = plugin_manager.PluginMapping(['available_resource',
                                                     'resource'])
    constraint_mapping = plugin_manager.PluginMapping('constraint')

    _register_resources(env, resource_mapping.load_all(manager))

    _register_constraints(env, constraint_mapping.load_all(manager))
```

首先创建一个 pluginmanager，负责管理所有的资源插件，注意参数为当前包名（即 heat.engine.resources），会在配置文件中指定的目录（默认为 /usr/lib/heat 和 /usr/lib64/heat）和当前包去查找所有模块。之后定义了两个 mapping，这两个 mapping 分别在给定的模块中查找 available_resource_mapping() 方法、resource_mapping() 方法和 constraint_mapping() 方法，之后通过 load_all() 方法来在所有的 pluginmanager 管理的模块中调用查找到的方法，并将结果添加到环境字典中。这些方法会返回一个字典，包括资源名到对应类的映射信息，例如在 route_table.py 资源文件中代码为

```
def resource_mapping():
    return {
        'AWS::EC2::RouteTable': RouteTable,
        'AWS::EC2::SubnetRouteTableAssociation': SubnetRouteTableAssociation,
    }
```

ceilometer

定义对 Alarm 和 CombinationAlarm 两种资源的处理。

neutron

定义对防火墙、floatingip、负载均衡、metering、net、gateway、port、router、subnet、vpn等网络相关的资源的处理。以 Port 为例，定义了 Port 类，通过resource_mapping指定 OS::Neutron::Port 资源映射到 Port 类上。Port 类中定义 properties_schema 和 attributes_schema。并定义对 CUD 操作的对应方法 handle_create()、handle_delete()、handle_update()。

software_config

autoscaling.py

处理 autoscaling 相关关键字。

cloud_watch.py

处理 OS::Heat::CWLiteAlarm 关键字。

eip.py

处理 AWS 中 EIP 相关关键字。

glance_image.py

处理 glance image 关键字。

instance.py

处理 AWS::EC2::Instance 和 OS::Heat::HARestarter 关键字。

internet_gateway.py

处理 AWS::EC2::InternetGateway和AWS::EC2::VPCGatewayAttachment 关键字。

loadbalancer.py

处理 AWS::ElasticLoadBalancing::LoadBalancer 关键字。

network_interface.py

处理 AWS::EC2::NetworkInterface 关键字。

nova_floatingip.py

处理 floatingIP 相关关键字。

nova_keypair.py

处理 OS::Nova::KeyPair 关键字。

nova_servergroup.py

处理 OS::Nova::ServerGroup 关键字。

os_database.py

处理 OS::Trove::Instance 关键字。

random_string.py

处理 OS::Heat::RandomString 关键字。

resource_group.py

处理 OS::Heat::ResourceGroup 关键字。

route_table.py

处理路由表相关关键字。

s3.py

处理 AWS::S3::Bucket 关键字。

security_group.py

处理 AWS::EC2::SecurityGroup 关键字。

server.py

处理 OS::Nova::Server 关键字。

stack.py

处理 AWS::CloudFormation::Stack 关键字。

subnet.py

处理 AWS::EC2::Subnet 关键字。

swift.py

处理 OS::Swift::Container 关键字。

user.py

处理 AWS::IAM::User、AWS::IAM::AccessKey 和 OS::Heat::AccessPolicy 关键字。

volume.py

处理 AWS::EC2::Volume、AWS::EC2::VolumeAttachment、OS::Cinder::Volume和 OS::Cinder::VolumeAttachment 关键字。

vpc.py

处理 AWS::EC2::VPC 关键字。

wait_condition.py

iso_8601.py

定义 ISO8601Constraint 类，解析 iso8601 标准的时间为普通时间对象。

nova_utils.py

一些辅助处理函数，包括更新 meta、刷新等。

template_resource.py

定义 TemplateResource 类，代表嵌套 stack 资源。

api.py

定义一些格式化的方法，包括对 stack、资源属性等进行格式化。让它们符合 API 输出格式。

attributes.py

代表资源的 attribute。定义类 Attribute、Attributes和Schema。

constraints.py

一些限制类和 schema，例如处理允许的 pattern 等。

dependencies.py

用图模型来处理资源的依赖关系。包括 Node、Graph、Dependencies 等类，描述不同资源之间的关系。

environment.py

定义资源信息类 ResourceInfo 和环境类 Environment 等。 所有的资源注册信息和映射关系等都被 Environment 类来统一管理。

event.py

定义 Event 类，代表资源的状态发生改变。

function.py

定义 Function 类，是模板中函数的抽象基础类。

parameter_groups.py

定义 ParameterGroups 类，处理模板中的参数组。

parameters.py

定义 Parameter、NumberParam、BooleanParam、StringParam 等类，处理模板中的参数。

parser.py

指定 Template为template.Template，Stack为stack.Stack 类。

plugin_manager.py

对 plugin 进行管理。 定义 PluginManager 类和 PluginMapping 类。 前者会导入所有的插件。在初始化的时候根据传入参数以及配置文件 heat.conf 中 plugin_dirs 指定的目录（默认是/usr/lib64/heat,/usr/lib/heat）去进行扫描所有模块并导入。 后者定义方法去扫描 PluginManager 中加载的模块，并根据指定感兴趣的mapping方法名称，例如 resource_mapping、available_resource_mapping、constraint_resource_mapping 等来导入相关的方法。

```
def load_from_module(self, module):
        '''Return the mapping specified in the given module.

        If no such mapping is specified, an empty dictionary is returned.
        '''
        for mapping_name in self.names:
            mapping_func = getattr(module, mapping_name, None)
            if callable(mapping_func):
                fmt_data = {'mapping_name': mapping_name, 'module': module}
                try:
                    mapping_dict = mapping_func(*self.args, **self.kwargs)
                except Exception:
                    LOG.error(_('Failed to load %(mapping_name)s '
                                'from %(module)s') % fmt_data)
                    raise
                else:
                    if isinstance(mapping_dict, collections.Mapping):
                        return mapping_dict
                    elif mapping_dict is not None:
                        LOG.error(_('Invalid type for %(mapping_name)s '
                                    'from %(module)s') % fmt_data)
        return {}
```

properties.py

定义 Property 类，处理模板中的 property。

resource.py

定义 Resource 基础类，是其他资源类的基类。 资源的属性包括行动、状态等，另外定义 default_client_name，用来向 openstack 中的服务发起操作指令。

rsrc_defn.py

定义 ResourceDefinition 类和 ResourceDefinitionCore 类。

scheduler.py

实现资源调度。对资源的操作（例如创建）是作为一个调度任务来管理的。 定义了 TaskRunner 等来实现调度任务管理。

service.py

定义了若干很重要的服务类，首先是 EngineListener，继承自 service.Service，监听 AMQP 队列。 EngineService 服务类，继承自 service.Service，bin/heat-engine 脚本启动的服务就是这个类。heatclient 发出的命令经过 heat-api，最终会以 rpc 请求消息的方式调用本类中的方法。包括对 stack、software_config 等资源的 create、delete 等。该服务启动后会创建一个 EngineListener 来监听消息。 EngineService 服务类在初始化的时候通过调用 resources.initialise() 来完成资源的注册，包括各种关键字对应的映射放到全局环境字典中。以 stack create 操作为例，对应方法为 EngineService 服务类中的 create_stack() 方法。该方法首先通过 _parse_template_and_validate_stack() 方法简单解析 stack 对应的模板类是哪种。

signal_responder.py

定义 SignalResponder 类，继承自 stack_user.StackUser，响应 create、delete 方法，清除数据等。

stack.py

定义了 Stack 类，继承自 collections.Mapping，代表 Heat template 中的一个 stack 对象，支持创建、删除、更新、回滚、暂停、继续、快照等操作。

stack_lock.py

定义StackLock类，用于维护一个stack资源上的锁。

stack_resource.py

定义了 StackResource 类，继承自 resource.Resource。抽象类，允许父级 stack 像管理一个资源一样来管理一个子 Stack。

stack_user.py

定义了 StackUser 类，继承自 resource.Resource。 创建一个关联 stack 的 user。

support.py

定义 SupportStatus，表示支持状态。

template.py

定义 Template 类和 TemplatePluginManager类，管理模板。

timestamp.py

定义 Timestamp 类。管理时间戳。

update.py

定义 StackUpdate 类，升级一个存在的 stack 到一个新的模板。

watchrule.py

定义 WatchRule 类。

locale

多国语言支持相关。

openstack

6.9.1 Common 含有 openstack 提供的基础服务类。openstack-common 项目负责维护这些代码，为所有的 openstack 子项目提供方法支持。

Config

还有 generator.py 文件，可以根据指定的代码文件来提取配置项，生成样例配置文件。

Crypto

跟加密相关的辅助方法。

Middleware

实现 WSGI 的 middleware 层。

Context.py

定义处理 http 请求上下文的 ReuqestContext 类。

eventlet_backdoor.py

实现 eventlet 的后门，监听。

excutils.py

定义处理异常类 save_and_reraise_exception，可以保存当前异常，运行一些代码，然后再次抛出异常。

fileutils.py

定义一些对文件进行操作的辅助方法。

gettextutils.py

语言国际化。

importutils.py

根据输入参数来动态导入对象、类等等。

jsonutils.py

对 json 内容进行处理。

local.py

弱本地引用类 WeakLocal。

lockutils.py

处理锁相关操作。

log.py

log 相关操作。

loopingcall.py

周期性调用类，包括 DynamicLoopingCall、FixedIntervalLoopingCall、LoopingCallBase。

network_utils.py

一些网络配置相关的辅助方法，包括从字符串解析主机名和端口、解析url等。

policy.py

对策略的处理。

processutils.py

管理进程。

service.py

实现服务基础类 Service、Services 等。

sslutils.py

定义使用 ssl 的辅助方法。

strutils.py

对字符串进行解码、编码等操作。

systemd.py

对 systemd 进行管理的方法。

threadgroup.py

定义 thread 类和 threadgroup 类，对线程进行管理。

timeutils.py

对时间相关的字符串进行处理等。

uuidutils.py

生成和检查 uuid。

versionutils.py

检查版本兼容信息。

rpc

api.py

定义了一些常量，包括 ENGINE_TOPIC、STACK_KEYTS 等。

client.py

定义了 EngineClient 类，该类可以利用 rpc 跟 heat-engine 进行通信完成各种资源操作。对资源进行操作的各个控制器实际上就是调用本类中的方法。 支持利用 call 和 cast 方法发出各种消息来调用方法，包括对 stack 的 show、count 等。

scaling

template.py

定义方法，支持利用已有的资源创建新的模板等。

tests

测试相关。

init.py

安装 _() 方法，支持国际化。

version.py

获取版本信息。

rally-scenarios

包括OpenStack CI进行rally benchmark测试时候的一些场景。

tools

包括跟安装和开发相关的一些工具。

config

检查配置文件和生成样例配置文件。

其它

- cfn-json2yaml 转化 cfn-json 格式为 yaml 格式。
- install_venv.py，Python 开发的时候，可能需要一套虚环境，该脚本可以安装一个 Python 虚环境。
- install_venv_common.py，为 install_venv.py 提供必要的方法。
- with_venv.sh，启用虚环境。
- heat-db-drop，清除 Heat 的数据库。
- requirements_style_check.sh，检查 requirement 的格式。

清理

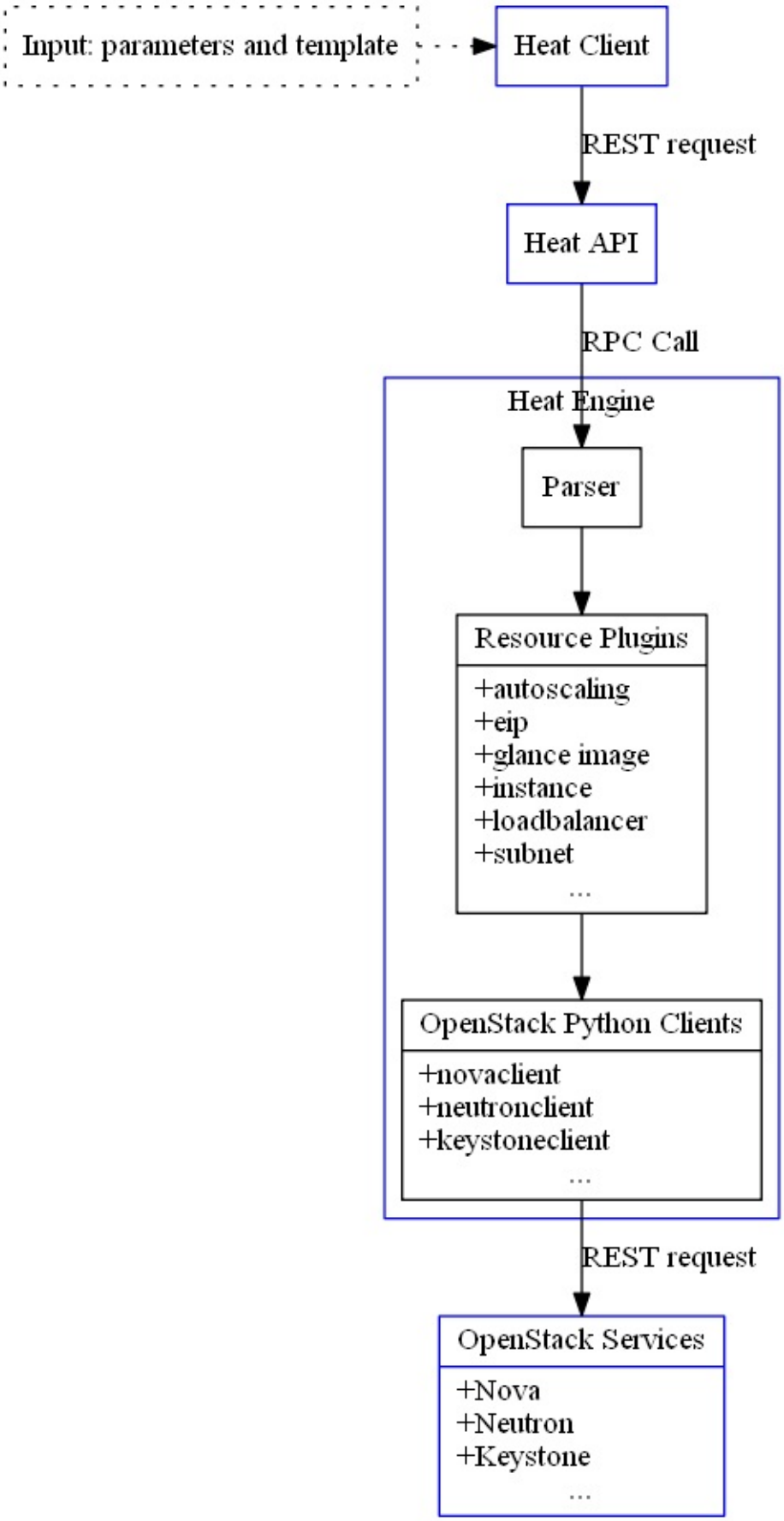
clean.sh，清除代码编译中间结果。

理解代码

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

调用逻辑

整体的调用逻辑如下图所示。



- heat-client，接受输入命令、参数和模板（URL、文件路径或数据），处理信息后转为 REST API 请求发送到 heat-api 服务。

- heat-api，接受请求，读入模板信息，处理后利用 rpc 请求发送给 heat-engine。
- heat-engine，解析模板数据，调用各种资源插件。
- resource-plugins，各种资源插件通过 OpenStack 的 clients 发送指令到 OpenStack 服务。