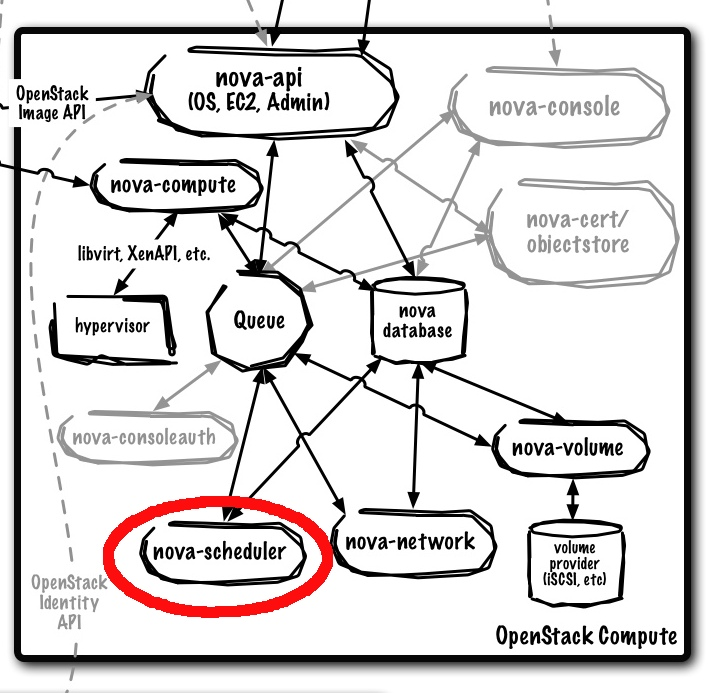
# Nova-scheduler浅析

#### 简介

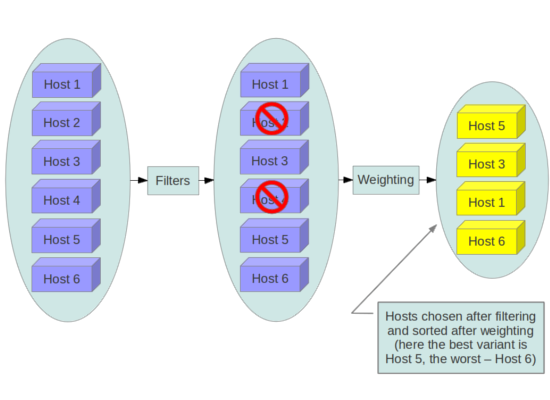
Nova-scheduler是openstack核心组件nova的核心组件之一，由此可见其在openstack中的地位。

顾名思义，Nova-scheduer是nova的调度器，目前是负责为创建/启动虚拟机实例寻找合适的计算节点，迁移虚拟机的时候也负责检查目的端的物理资源是否足够，其重要性随着计算节点数量的增加而增加。

下图是openstack架构图，从中我们可以看出nova-scheduer是通过与message queue和nova database交互来完成调度任务的。

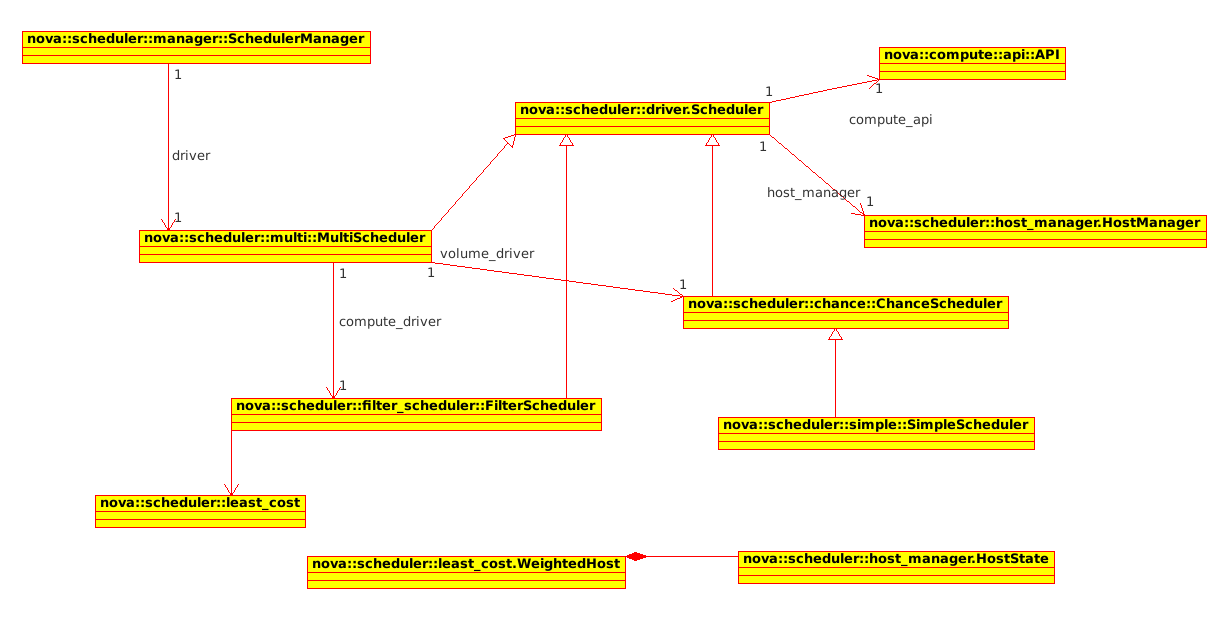


Nova-scheduer的简化工作流程可以用下图表示：



也即过滤不可用节点，并对可用节点进行权重排序，根据用户配置的策略选出最优节点。

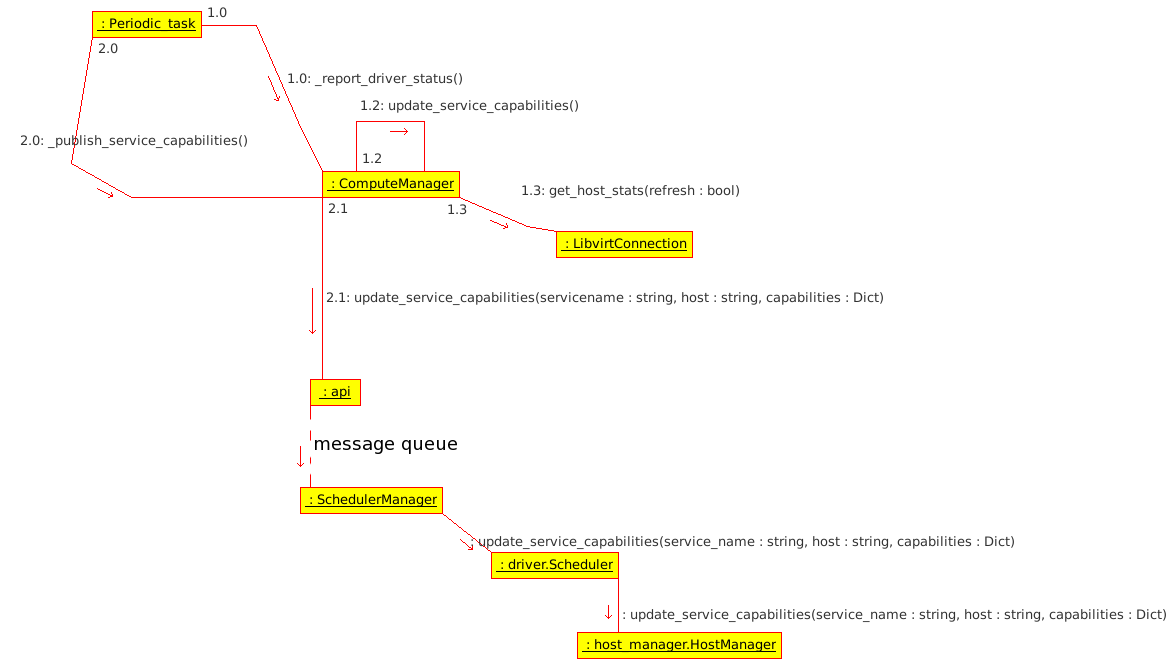
下面是nova-scheduler的类图：



#### 计算节点可用资源通知

要想选出策略最优的节点，就需要首先知道各个节点当前的可用资源状况以及请求需要的资源信息，而后者是用户通过API请求主动传递过来的，所以不是nova-scheduer关注的重点。

下图描述了nova-compute服务如何更新其节点资源信息并发布到message queue，以及nova-scheduer如何收集并保存这些信息以备后续使用：



上述图示可以分为三个相互关联的部分：信息收集、信息发布、信息存储。信息收集（上图1.X部分）由nova-compute服务完成，当然它要依赖hypervisor适配层提供的接口，比如libvirt接口或者XenAPI接口等（实际执行收集动作的函数为nova/virt/libvirt/connection.py:HostState.update\_status()），收集到的信息供信息发布函数使用；信息发布（上图2.X部分）是nova-compute服务通过RPC机制完成的，因为发布是跨服务的（nova-compute到nova-scheduer），所以使用message queue也是顺理成章的事情，如果信息收集函数没能把最新的信息传递过来，那么发布上次更新到的信息；信息存储（上图message queue下面的部分）是nova-scheduer服务完成的，它从message queue中获取相关信息并存储起来，就可以用来作为相关决策的依据。

*nova/scheduler/host\_manager.py:HostManager:*

……

def update\_service\_capabilities(self, service\_name, host, capabilities):

"""Update the per-service capabilities based on this notification."""

LOG.debug(\_("Received %(service\_name)s service update from "

"%(host)s.") % locals())

service\_caps = self.service\_states.get(host, {})

# Copy the capabilities, so we don't modify the original dict

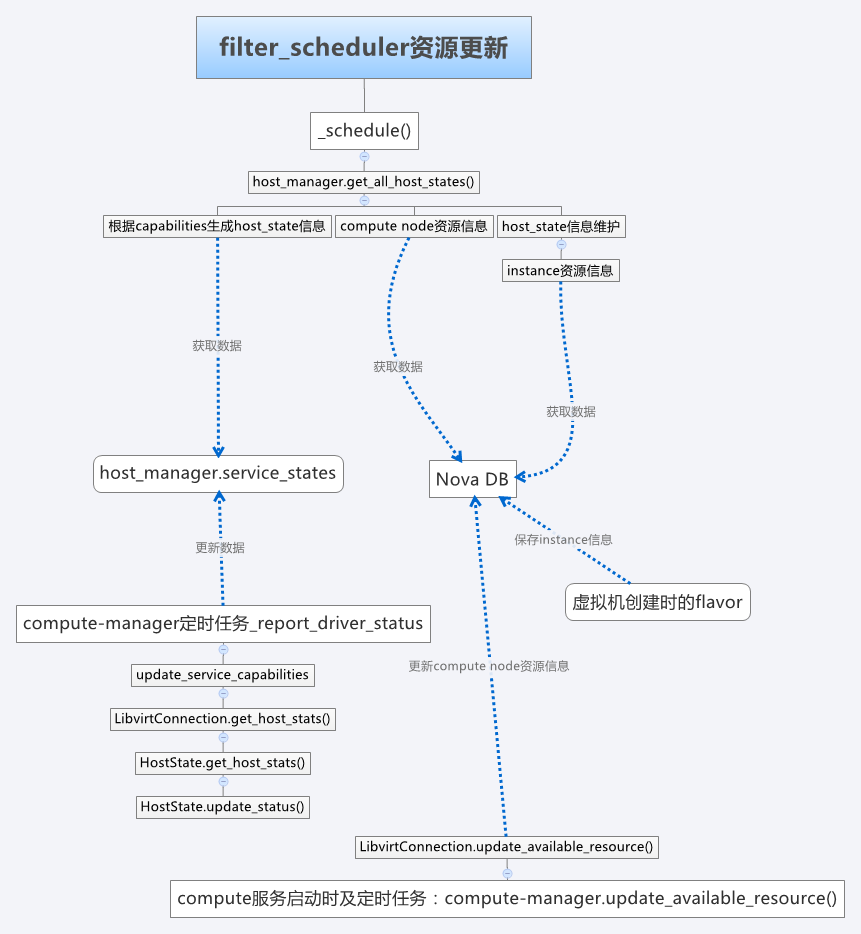
capab\_copy = dict(capabilities)

capab\_copy["timestamp"] = utils.utcnow() # Reported time

service\_caps[service\_name] = capab\_copy

self.service\_states[host] = service\_caps #把拿到的资源信息存在自己的类属性里面以备使用

这里有个问题就是当上一个资源占用请求还没有处理的时候，又来了一个新的请求，并且被调度到的计算节点的资源只能满足一个请求，那么第二个请求就会失败。



#### 从API到manager

这里的API指的是scheduler的API（当然除了nova-api之外的其他服务的API到manager的流程也是类似的，这里仅讨论nova-scheduer）。我们知道API到manager的消息传递是通过message queue完成的，nova里面的message queue采用的是AMQP，与AMQP建立连接的过程是在服务启动的时候完成的，所以我们首先要看下nova-scheduer服务的启动流程。

Nova-scheduler服务脚本：

创建服务实例和最后的阻塞等待就不用多说了，我们直接看启动服务流程，nova/service.py:serve->launch\_server->eventlet用greenthread启动主线程执行run\_server->Service.start():

*nova/service.py:Service.start():*

……

#关于AMQP和Nova的详细信息可参考：<http://jzhihui.iteye.com/blog/1497031>

self.conn = rpc.create\_connection(new=True) #建立到rabbitMQ的连接

LOG.debug(\_("Creating Consumer connection for Service %s") %

self.topic)

# Share this same connection for these Consumers

self.conn.create\_consumer(self.topic, self, fanout=False) #创建消费者（基于topic的）

node\_topic = '%s.%s' % (self.topic, self.host)

self.conn.create\_consumer(node\_topic, self, fanout=False) #创建消费者（基于topic&host，可用来接受定向消息）

self.conn.create\_consumer(self.topic, self, fanout=True) #创建消费者（fanout类型，可用于接收广播消息）

# Consume from all consumers in a thread

self.conn.consume\_in\_thread() #启动消费者线程

……

我们继续看创建消费者的过程：

*nova/rpc/impl\_kombu.py:Connection:*

……

# 为什么会走到这里？而不是amqp.py中的该方法？

# 因为create\_connection()返回的是这个文件里面的Connection类的实例。

def create\_consumer(self, topic, proxy, fanout=False):

"""Create a consumer that calls a method in a proxy object"""

if fanout:

self.declare\_fanout\_consumer(topic,

rpc\_amqp.ProxyCallback(proxy, Connection.pool)) #注册RPC调用的callback，所有的调用都是从这里分发出去的（这里的proxy就是Service类的self）

else:

self.declare\_topic\_consumer(topic,

rpc\_amqp.ProxyCallback(proxy, Connection.pool)) #同上

……

这样就走到了ProxyCallback类里面：

*nova/rpc/amqp.py:ProxyCallback:*

……

# 为什么会走到这里？

# 可以参考\_\_call\_\_的含义：http://docs.python.org/reference/datamodel.html#emulating-callable-objects。

def \_\_call\_\_(self, message\_data):

……

rpc\_common.\_safe\_log(LOG.debug, \_('received %s'), message\_data) #amqp receive XXX的日志就是这里打印的

ctxt = unpack\_context(message\_data)

method = message\_data.get('method') #获取要调用的方法

args = message\_data.get('args', {}) #获取调用的参数

if not method:

……

self.pool.spawn\_n(self.\_process\_data, ctxt, method, args) #启动线程执行\_process\_data()处理调用流程

……

继续

*nova/rpc/amqp.py:ProxyCallback:*

……

def \_process\_data(self, ctxt, method, args):

"""Thread that magically looks for a method on the proxy

object and calls it.

"""

ctxt.update\_store()

try:

node\_func = getattr(self.proxy, str(method)) #获取proxy中的method

node\_args = dict((str(k), v) for k, v in args.iteritems())

# NOTE(vish): magic is fun!

rval = node\_func(context=ctxt, \*\*node\_args) #调用method

#reply

……

return

……

我们看到获取method是用getattr()方法实现的，在Service类里面定义了\_\_getattr\_\_方法，

*nova/service.py:Service:*

……

def \_\_getattr\_\_(self, key):

manager = self.\_\_dict\_\_.get('manager', None)

return getattr(manager, key) #再次查询manager中的method并返回，这样就走到了nova-scheduer的manager类中

……

另外在nova-scheduer的manager.py的SchedulerManager也定义了\_\_getattr\_\_方法，用来处理manager中未实现的被调用到的方法，这些方法都被定向到了\_schedule方法。

*nova/scheduler/manager.py:*

……

def \_\_getattr\_\_(self, key):

"""Converts all method calls to use the schedule method"""

return functools.partial(self.\_schedule, key)

……

关于\_\_get\_attr\_\_方法：

object.\_\_getattr\_\_(self, name)

Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for self). name is the attribute name. This method should return the (computed) attribute value or raise an AttributeError exception.

至此API到manager的流程结束。

这部分要感谢田田高同学的鼎力基情支持！

#### 从manager到driver

这部分的流程比较简单，在SchedulerManager类初始化的时候可以看到driver的初始化过程：

*nova/scheduler/manager.py:*

……

scheduler\_driver\_opt = cfg.StrOpt('scheduler\_driver',

default='nova.scheduler.multi.MultiScheduler', #scheduler driver的默认配置

help='Default driver to use for the scheduler')

……

def \_\_init\_\_(self, scheduler\_driver=None, \*args, \*\*kwargs):

if not scheduler\_driver:

scheduler\_driver = FLAGS.scheduler\_driver #scheduler的driver是根据配置确定的，当然也有默认值

self.driver = utils.import\_object(scheduler\_driver)

……

比如我们在resize虚拟机的时候，nova的compute-api的resize方法会通过message queue调用scheduler的prep\_resize方法：self.\_cast\_scheduler\_message(context, {"method": "prep\_resize", "args": args})，这样就会调用到SchedulerManager里面的prep\_resize方法，进而调用到MultiScheduler里面的schedule\_prep\_resize方法，

*nova/scheduler/multi.py:MultiScheduler:*

……

def schedule\_prep\_resize(self, \*args, \*\*kwargs):

return self.drivers['compute'].schedule\_prep\_resize(\*args, \*\*kwargs)

……

这里我们又看到了一个dict类型的self.drivers，可以在MultiScheduler的\_\_init\_\_方法中找到它的初始化过程，

*nova/scheduler/multi.py:MultiScheduler:*

……

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

super(MultiScheduler, self).\_\_init\_\_()

compute\_driver = utils.import\_object(FLAGS.compute\_scheduler\_driver)

volume\_driver = utils.import\_object(FLAGS.volume\_scheduler\_driver)

self.drivers = {'compute': compute\_driver,

'volume': volume\_driver}

……

这里我们看到了两个子driver，compute\_driver和volume\_driver，也都是可以在配置文件里面配的，默认值如下：

*nova/scheduler/multi.py:MultiScheduler:*

……

multi\_scheduler\_opts = [

cfg.StrOpt('compute\_scheduler\_driver',

default='nova.scheduler.'

'filter\_scheduler.FilterScheduler',

help='Driver to use for scheduling compute calls'),

cfg.StrOpt('volume\_scheduler\_driver',

default='nova.scheduler.chance.ChanceScheduler',

help='Driver to use for scheduling volume calls'),

]

……

volume\_driver用到的地方不多，在nova/volume/api.py:API.create()方法里面有用到，目的是为了选择一个nova-volume节点创建卷。

compute\_driver用到的地方就比较多了，几乎所有涉及到启动虚拟机类的动作都跟它有关，比如创建虚拟机、start虚拟机、resize虚拟机、迁移虚拟机等等，因为启动虚拟机涉及到选择在哪个节点上启动的策略问题。我们继续以resize为例进行说明，从MultiScheduler的schedule\_prep\_resize方法调用到了FilterScheduler里的schedule\_prep\_resize方法：

*nova/scheduler/multi.py:MultiScheduler:*

……

def schedule\_prep\_resize(self, context, request\_spec, \*args, \*\*kwargs):

……

hosts = self.\_schedule(context, 'compute', request\_spec, \*args, \*\*kwargs) #过滤出可用的host列表

if not hosts:

raise exception.NoValidHost(reason="")

host = hosts.pop(0) #目前只选取列表中的第一个host

# NOTE(comstud): Make sure we do not pass this through. It

# contains an instance of RpcContext that cannot be serialized.

kwargs.pop('filter\_properties', None) #过滤掉RPC调用不支持的参数

# Forward off to the host

driver.cast\_to\_compute\_host(context, host.host\_state.host, 'prep\_resize', \*\*kwargs) #基于host的定向RPC调用

……

这样就把prep\_resize动作定向到了特定的nova-compute节点，以便让resize的目的端节点做好相关准备工作。

除了FilterScheduler调度器之外还有其他的调度器，比如ChanceScheduler、SimpleScheduler，ChanceScheduler是随机选择一个运行中的host，SimpleScheduler则是根据已经占用的核数来选择host，当前占用核数最少的host将被选中。这里我们重点讨论默认的FilterScheduler，这个调度器负责host的选择以及选中后host的资源预占，选择host的依据是使用过滤器过滤host和对host计算权重费用，成功通过所有的过滤器并且权重费用最低的host将被选中。

#### 从Driver到filter

FilterScheduler（目前只支持compute服务调度）有很多的filter用来选择host，可以用的filter通过配置项’scheduler\_available\_filters’来确定，默认值为’scheduler\_available\_filters’，根据注释可以知道默认是遍历所有可用的filter，也就是只要在nova/scheduler/filters目录下的\*\_filter.py都可以使用。

Nova-scheduler可用filter列表：

*'nova.scheduler.filters.isolated\_hosts\_filter.IsolatedHostsFilter'*

*'nova.scheduler.filters.compute\_filter.ComputeFilter'*

*'nova.scheduler.filters.availability\_zone\_filter.AvailabilityZoneFilter'*

*'nova.scheduler.filters.ram\_filter.RamFilter'*

*'nova.scheduler.filters.json\_filter.JsonFilter'*

*'nova.scheduler.filters.all\_hosts\_filter.AllHostsFilter'*

*'nova.scheduler.filters.core\_filter.CoreFilter'*

*'nova.scheduler.filters.affinity\_filter.AffinityFilter'*

*'nova.scheduler.filters.affinity\_filter.DifferentHostFilter'*

*'nova.scheduler.filters.affinity\_filter.SameHostFilter'*

*'nova.scheduler.filters.affinity\_filter.SimpleCIDRAffinityFilter’*

另外还有一个配置项是’scheduler\_default\_filters’，这个参数可以配置默认使用的filter，也就是说虽然可用的filter很多，但并未全部使用，其默认配置有三个filter：'AvailabilityZoneFilter', 'RamFilter', 'ComputeFilter'。

我们先看下\_scheduler这个方法，它负责筛选可用的Host列表，我们知道如果要做出决策，一定要有数据支持，否则就是瞎蒙了，所以下面我们先看下筛选Host所需要的数据：

*nova/scheduler/filter\_scheduler.py:FilterScheduler:*

……

def \_schedule(self, context, topic, request\_spec, \*args, \*\*kwargs):

……

elevated = context.elevated()

if topic != "compute":

msg = \_("Scheduler only understands Compute nodes (for now)") #目前仅支持compute节点

raise NotImplementedError(msg)

instance\_properties = request\_spec['instance\_properties']

instance\_type = request\_spec.get("instance\_type", None)

cost\_functions = self.get\_cost\_functions() #costfunction，下一节将会讲到

config\_options = self.\_get\_configuration\_options()

filter\_properties = kwargs.get('filter\_properties', {})

filter\_properties.update({'context': context, 'request\_spec': request\_spec, 'config\_options': config\_options,

'instance\_type': instance\_type}) #把用户指定的信息整合起来

self.populate\_filter\_properties(request\_spec, filter\_properties)

……

hosts = self.host\_manager.filter\_hosts(hosts, filter\_properties) #筛选Hosts，这里需要两个参数做决策依据

……

从代码里面可以看出需要两个参数作为决策依据：Host和filter\_properties，既然是筛选Hosts，那以Host作为输入也是理所当然的，Host信息的获取方式是：首先查询数据库获取到所有的compute 服务节点，然后查询自己之前保存的service\_states属性获取相关信息，最后封装成HostState对象返回。

剩下的就是filter\_properties了，这个参数保存了虚拟机的规格、以及特定的Host选择要求等信息，这些信息都是由用户在启动虚拟机的时候指定的，比如我们可以在创建虚拟机的API的data里面设置如下参数：

{

'server': {

'name': 'server-1',

'imageRef': 'cedef40a-ed67-4d10-800e-17455edce175',

'flavorRef': '1'

},

'os:scheduler\_hints': {

“different\_host”: [“a0cf03a5-d921-4877-bb5c-86d26cf818e1”,“8c19174f-4220-44f0-824a-cd1eeef10287”],

“same\_host”: [“a0cf03a5-d921-4877-bb5c-86d26cf818e1”,“8c19174f-4220-44f0-824a-cd1eeef10287”],

“build\_near\_host\_ip”: “192.168.1.1”,

“cidr”: “24”

}

}

(different\_host: Schedule the instance on a different host from a set of instances.

same\_host: Schedule the instance on the same host as another instance in a set of of instances.

IP and cidr: Schedule the instance based on host IP subnet range.)

我们还可以指定特定的Host来启动虚拟机或者指定不在特定的Host上启动：

'force\_hosts': ['testhost']，ignore\_hosts: ['testhost']

这两个参数都是在执行过滤动作之前生效的，看下代码实现：

*nova/scheduler/host\_manager.py:HostState:*

……

def passes\_filters(self, filter\_fns, filter\_properties):

"""Return whether or not this host passes filters."""

if self.host in filter\_properties.get('ignore\_hosts', []): #先检查这个Host是否为ignore\_host，是则直接过滤掉

LOG.debug(\_('Host filter fails for ignored host %(host)s'), {'host': self.host})

return False

force\_hosts = filter\_properties.get('force\_hosts', [])

if force\_hosts: #再检查是否为force\_host

if not self.host in force\_hosts:

LOG.debug(\_('Host filter fails for non-forced host %(host)s'), {'host': self.host})

return self.host in force\_hosts #只要用户指定了force\_host，那么就不再执行过滤动作，直接返回

for filter\_fn in filter\_fns: #最后才开始用过滤器过滤该Host

if not filter\_fn(self, filter\_properties):

LOG.debug(\_('Host filter function %(func)s failed for ' '%(host)s'), {'func': repr(filter\_fn), 'host': self.host})

return False

……

我们甚至还可以传入任意key-value对到scheduler里以备特定用途使用。

filter\_properties里面整合了request\_spec这个参数，这也是filter过滤Host的主要依据，通常request\_spec包含的内容如下，其中黄色高亮部分是对filter或者costfunction比较重要的项：

{

'num\_instances': 1,

'block\_device\_mapping': [],

'image': {

'status': 'active',

'name': 'cirros\_blank',

'deleted': False,

'container\_format': 'ami',

'created\_at': '2012-04-05 14:26:24',

'disk\_format': 'ami',

'updated\_at': '2012-04-05 14:26:25',

'properties': {

'kernel\_id': '46bf134e-2e6e-472a-a159-f4cd51f36d84',

'ramdisk\_id': '106dc550-783e-4de7-951d-f4f3d5427698' },

'min\_ram': '0',

'checksum': '2f81976cae15c16ef0010c51e3a6c163',

'min\_disk': '0',

'is\_public': True,

'deleted\_at': None,

'id': 'a3fb743d-42df-49ba-b9c4-8042ebbd344e',

'size': 25165824},

'instance\_type': {

'root\_gb': 0L,

'name': u 'm1.tiny',

'deleted': False,

'created\_at': None,

'ephemeral\_gb': 0L,

'updated\_at': None,

'memory\_mb': 512L,

'vcpus': 1L,

'flavorid': u '1',

'swap': 0L,

'rxtx\_factor': 1.0,

'extra\_specs': {},

'deleted\_at': None,

'vcpu\_weight': None,

'id': 2L },

'instance\_properties': { # used to comsume virtual resources

'vm\_state': 'building',

'ephemeral\_gb': 0L,

'access\_ip\_v6': None,

'access\_ip\_v4': None,

'kernel\_id': '46bf134e-2e6e-472a-a159-f4cd51f36d84',

'key\_name': None,

'ramdisk\_id': '106dc550-783e-4de7-951d-f4f3d5427698',

'instance\_type\_id': 2L,

'user\_data': '',

'vm\_mode': None,

'display\_name': u 'myserver',

'config\_drive\_id': '',

'reservation\_id': 'r-bdbnl7aa',

'key\_data': None,

'root\_gb': 0L,

'user\_id': u '81ced34d11954800906096555539c885',

'uuid': u '4ccc7c93-cbde-4233-a7cb-5db81f82489b',

'root\_device\_name': None,

'availability\_zone': u 'myzone', # default to FLAGS.default\_schedule\_zone

'launch\_time': '2012-04-11T15:08:55Z',

'metadata': {},

'display\_description': u 'myserver',

'memory\_mb': 512L,

'launch\_index': 0,

'vcpus': 1L,

'locked': False,

'image\_ref': u 'a3fb743d-42df-49ba-b9c4-8042ebbd344e',

'architecture': None,

'power\_state': 0,

'auto\_disk\_config': None,

'progress': 0,

'os\_type': None,

'project\_id': u '9d049e4b60b64716978ab415e6fbd5c0',

'config\_drive': '' },

'security\_group': ['default']

}

上面提到的passes\_filters方法，是\_schedule方法中执行过滤过程filter\_hosts用到的，它属于HostState类，也就是说每个需要过滤的Host都要通过这个方法进行过滤，成功通过的返回True，否则返回False。

*nova/scheduler/host\_manager.py:HostManager:*

……

def filter\_hosts(self, hosts, filter\_properties, filters=None):

"""Filter hosts and return only ones passing all filters"""

filtered\_hosts = []

filter\_fns = self.\_choose\_host\_filters(filters) #选择过滤器，默认为FLAGS.scheduler\_default\_filters配置的三个

for host in hosts:

if host.passes\_filters(filter\_fns, filter\_properties): #用选出的过滤器过滤每个Host

filtered\_hosts.append(host) #成功通过的加入到过滤出的Hosts列表里

return filtered\_hosts

……

我们再看\_choose\_host\_filters是如何选择过滤器的，以及返回的是什么东东：

*nova/scheduler/host\_manager.py:HostManager:*

……

def \_choose\_host\_filters(self, filters):

"""Since the caller may specify which filters to use we need

to have an authoritative list of what is permissible. This

function checks the filter names against a predefined set

of acceptable filters.

"""

if filters is None:

filters = FLAGS.scheduler\_default\_filters #使用默认过滤器

if not isinstance(filters, (list, tuple)):

filters = [filters]

good\_filters = []

bad\_filters = []

for filter\_name in filters: #遍历所有配置的过滤器（我们使用的是默认过滤器）

found\_class = False

for cls in self.filter\_classes: #遍历所有可用的过滤器

if cls.\_\_name\_\_ == filter\_name: #如果在可用过滤器中找到配置的过滤器，则认为它可以使用

found\_class = True

filter\_instance = cls()

# Get the filter function

filter\_func = getattr(filter\_instance, 'host\_passes', None) #获取过滤器类中的host\_passes方法

if filter\_func:

good\_filters.append(filter\_func) #把过滤器的host\_passes方法添加到好用过滤器的列表中

break

if not found\_class:

bad\_filters.append(filter\_name)

if bad\_filters:

msg = ", ".join(bad\_filters) #注意这里如果配置了不可用的过滤器，就直接raise了

raise exception.SchedulerHostFilterNotFound(filter\_name=msg)

return good\_filters #返回好用的过滤器列表

……

默认的过滤器肯定是可以用的，因为他们都是从可用过滤器里挑选出来的，\_choose\_host\_filters选出的好用的过滤器的host\_passes方法最后都作为参数传给了passes\_filters，之后被调用用来过滤Host，我们再来看下这三个过滤器的host\_passes方法：

*nova/scheduler/filters/availability\_zone\_filter.py:*

class AvailabilityZoneFilter(filters.BaseHostFilter):

"""Filters Hosts by availabilty zone."""

def host\_passes(self, host\_state, filter\_properties):

spec = filter\_properties.get('request\_spec', {})

props = spec.get('instance\_properties', {})

availability\_zone = props.get('availability\_zone')

if availability\_zone:

return availability\_zone == host\_state.service['availability\_zone']#只是判断下指定的az是否与当前host的az相同

return True #如果没有指定az，则该Host通过过滤

*nova/scheduler/filters/ram\_filter.py:*

ram\_allocation\_ratio\_opt = cfg.FloatOpt("ram\_allocation\_ratio",

default=1.5,

help="virtual ram to physical ram allocation ratio")

class RamFilter(filters.BaseHostFilter):

"""Ram Filter with over subscription flag"""

def host\_passes(self, host\_state, filter\_properties):

"""Only return hosts with sufficient available RAM."""

instance\_type = filter\_properties.get('instance\_type')

requested\_ram = instance\_type['memory\_mb']

free\_ram\_mb = host\_state.free\_ram\_mb

return free\_ram\_mb \* FLAGS.ram\_allocation\_ratio >= requested\_ram #注意这里的配置是启用了内存复用的，也就是说虽然Host实际可用内存达不到指定的内存，但还是可以通过过滤（内存复用比为1.5，也即可以超售50%）

*nova/scheduler/filters/compute\_filter.py:*

class ComputeFilter(filters.BaseHostFilter):

"""HostFilter hard-coded to work with InstanceType records."""

def \_satisfies\_extra\_specs(self, capabilities, instance\_type):

"""Check that the capabilities provided by the compute service

satisfy the extra specs associated with the instance type"""

if 'extra\_specs' not in instance\_type: #默认我们是不设置extra\_specs的，

return True

# NOTE(lorinh): For now, we are just checking exact matching on the

# values. Later on, we want to handle numerical

# values so we can represent things like number of GPU cards #看作者的注释可能是为了以后支持GPU虚拟化做准备

for key, value in instance\_type['extra\_specs'].iteritems():

if capabilities.get(key, None) != value:

return False

return True

def host\_passes(self, host\_state, filter\_properties):

"""Return a list of hosts that can create instance\_type."""

instance\_type = filter\_properties.get('instance\_type')

if host\_state.topic != 'compute' or not instance\_type: #仅对compute服务生效

return True

capabilities = host\_state.capabilities

service = host\_state.service

#service\_is\_up就是检查心跳是否正常

if not utils.service\_is\_up(service) or service['disabled']: #检测服务状态是否可用，如果不可用就不能通过过滤器

return False #也就是如果状态上报异常（比如超时）就会无法通过

if not capabilities.get("enabled", True): #没看到有设置这个字段，默认为True，所以这里应该不满足条件

return False

if not self.\_satisfies\_extra\_specs(capabilities, instance\_type): #检查instance\_type里的extra\_specs

return False

return True

心跳的报告是在服务启动的时候开始的，它是一个基于greenthread的定时任务：

*nova/service.py:service.start()*

……

if self.report\_interval:

pulse = utils.LoopingCall(self.report\_state) #调用service类的report\_state方法进行心跳报告

pulse.start(interval=self.report\_interval, now=False)

self.timers.append(pulse)

……

#### 从filter到cost function

用预定义的三个过滤器过滤完之后，会得到一个Hosts列表，保存了所有通过三个过滤器的Host，列表中可能不止一个Host，所以还要在这些Host当中选择一个最优的节点，以起到均衡资源分配的目的，选择最优节点的方式就是利用costfunction计算出各个Host启动虚拟机所需的“费用”，消耗“费用”最低的节点将被选中用来执行启动或者迁入虚拟机之类的操作。

上一节中提到过costfunction是在FilterScheduler类的\_scheduler方法中通过调用get\_cost\_functions获取的，代码如下：

*nova/scheduler/filter\_scheduler.py:FilterScheduler:*

……

def get\_cost\_functions(self, topic=None):

……

if topic in self.cost\_function\_cache: #先尝试从cache里面拿

return self.cost\_function\_cache[topic]

cost\_fns = []

for cost\_fn\_str in FLAGS.least\_cost\_functions: #默认值'nova.scheduler.least\_cost.compute\_fill\_first\_cost\_fn'，可配多个

……

try:

……

cost\_fn = utils.import\_class(cost\_fn\_str) #导入costfunction

……

try:

flag\_name = "%s\_weight" % cost\_fn.\_\_name\_\_

weight = getattr(FLAGS, flag\_name) #获取costfunction对应的权重因子 compute\_fill\_first\_cost\_fn\_weight，默认值为-1.0意为spread first（均分优先），如果为正值，则意为fill first（填充优先）

……

cost\_fns.append((weight, cost\_fn)) #每个costfunction对应一个权重因子，二者配合以计算“费用”

self.cost\_function\_cache[topic] = cost\_fns #放入cache

return cost\_fns

……

我们再看下nova.scheduler.least\_cost.compute\_fill\_first\_cost\_fn的定义：

*nova/scheduler/least\_cost.py:*

def compute\_fill\_first\_cost\_fn(host\_state, weighing\_properties):

"""More free ram = higher weight. So servers will less free

ram will be preferred.""" #内存越多权重越高，由于权重因子为负值，所以计算出的“费用”就越低

return host\_state.free\_ram\_mb

现在我们来看最后的“费用”计算过程（也是在FilterScheduler类的\_scheduler方法中调用的）：

*nova/scheduler/least\_cost.py:*

def weighted\_sum(weighted\_fns, host\_states, weighing\_properties):

……

scores = [] #初始化分数也就是“费用”

for weight, fn in weighted\_fns:#这里的weight和fn分别为上面提到的compute\_fill\_first\_cost\_fn\_weight和compute\_fill\_first\_cost\_fn

scores.append([fn(host\_state, weighing\_properties) for host\_state in host\_states]) #对每个Host用compute\_fill\_first\_cost\_fn计算“费用”并保存，其实就是看下Host的空闲内存信息，由于目前只有一个compute\_fill\_first\_cost\_fn，所以计算结果类似[host1.freeRam, host2.freeRam,…]

# Adjust the weights in the grid by the functions weight adjustment

# and sum them up to get a final list of weights.

adjusted\_scores = []

for (weight, fn), row in zip(weighted\_fns, scores):

adjusted\_scores.append([weight \* score for score in row]) #用compute\_fill\_first\_cost\_fn\_weight对计算结果进行加权，计算结果类似[-host1.freeRam, -host2.freeRam,…]

# Now, sum down the columns to get the final score. Column per host.

final\_scores = [0.0] \* len(host\_states)

for row in adjusted\_scores: #如果有多个costfunction则把每个function计算出的Host的“费用”累加起来得到“总费用”

for idx, col in enumerate(row):

final\_scores[idx] += col #因为只有一个costfunction，所以这里的final\_scores跟上面的adjusted\_scores相同

# Super-impose the host\_state into the scores so

# we don't lose it when we sort. #把“费用”和Host组成tuple放到list里

final\_scores = [(final\_scores[idx], host\_state) for idx, host\_state in enumerate(host\_states)]

final\_scores = sorted(final\_scores) #对“费用”排序

weight, host\_state = final\_scores[0] # Lowest score is the winner!

return WeightedHost(weight, host\_state=host\_state) #返回“费用”最低的Host

#### 资源预占

还有一个问题就是，如果一次启动多个instance的话，nova-scheduer是一个一个的选择Host的，而且中间不会更新节点的可用资源，这里就有一个可用资源数量不准确的问题，比如Host1被选中启动了一个instance，对nova-scheduer来说Host1的资源并没有变化，第二个instance还是可能被启动到Host1上，这是个非常明显的问题，所以社区肯定是考虑到了的，社区的解决方法就是资源预占，简单来说就是先在nova-scheduer的进程内部把被选中的Host的可用资源中把已经启动到这个Host上的instance的相关资源信息给减掉，这样就可以避免资源数量不准确的问题了。

*nova/scheduler/host\_manager.py:HostState:*

……

def consume\_from\_instance(self, instance): #在\_scheduler方法里面选中Host后被调用

"""Update information about a host from instance info."""

disk\_mb = (instance['root\_gb'] + instance['ephemeral\_gb']) \* 1024

ram\_mb = instance['memory\_mb']

vcpus = instance['vcpus']

self.free\_ram\_mb -= ram\_mb

self.free\_disk\_mb -= disk\_mb

self.vcpus\_used += vcpus

……

#### 其他问题

Nova-scheduer多节点布置？？--essex版本不支持，folsom正在开发。

问题参考：

<https://answers.launchpad.net/nova/+question/199303>

社区蓝图：

<https://blueprints.launchpad.net/nova/+spec/scheduler-resource-race>

Work items:

Added scheduling retries when build errors occur: DONE

Added resource tracking in the compute host to more gracefully control resource usage and provide up-to-date information to the scheduler: INPROGRESS

已经在F-3版本中实现的失败后重新调度代码：

<https://review.openstack.org/#/c/9540/>