OpenStack 原理分析

之 Nova 组件

OpenStack 简介

OpenStack 既是一个社区,也是一个项目和一个开源软件,它提供了一个部署云的操作平台或工具集。其宗旨在于,帮助组织运行为虚拟计算或存储服务的云,为公有云、私有云,也为大云、小云提供可扩展的、灵活的云计算。

OpenStack 旗下包含了一组由社区维护的开源项目,分别是:

（1）OpenStack Compute (Nova)

（2）OpenStack Object Storage (Swift)

（3）OpenStack Image Service (Glance)

（4）OpenStack Networking (Quantum)

（5）OpenStack Block Storage (Cinder)

（6）OpenStack Dashboard (Horizon)

（7）OpenStack Identity (Keystone)

（8）OpenStack Common Library (Oslo)

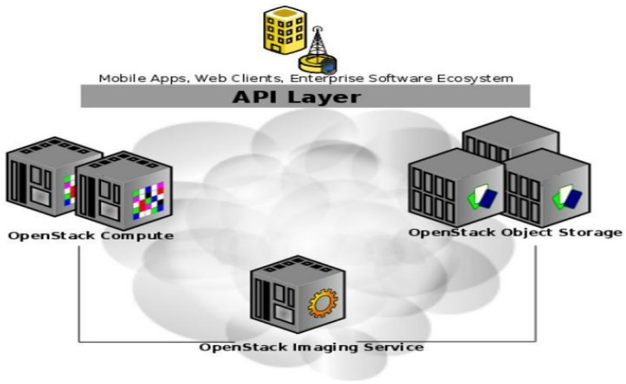
其中不可或却的至少有三个组件：Compute，Object Storage，Image Service。

OpenStack Compute,为云组织的控制器,它提供一个工具来部署云,包括运行实例、管理网络以及控制用户和其他项目对云的访问。它底层的开源项目代号是 Nova ,其提供的软件能控制 IaaS 云计算平台,类似于 AmazonEC2 和 RackSpace Cloud Servers。实际上它定义的是,与运行在主机操作系统上潜在的虚拟化机制交互的驱动,暴露基于 Web API 的功能。

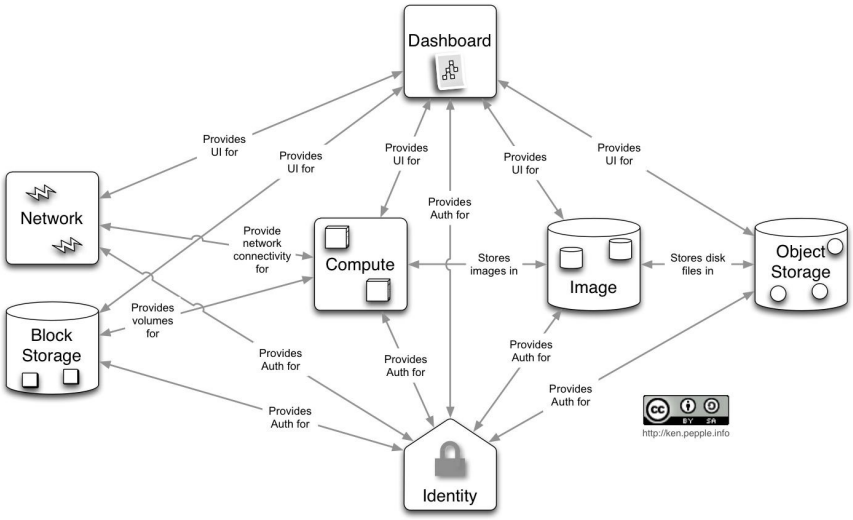
OpenStack Object Storage,是一个可扩展的对象存储系统。对象存储支持多种应用,比如复制和存档数据,图像或视频服务,存储次级静态数据,开发数据存储整合的新应用,存储容量难以估计的数据,为 Web 应用创建基于云的弹性存储。

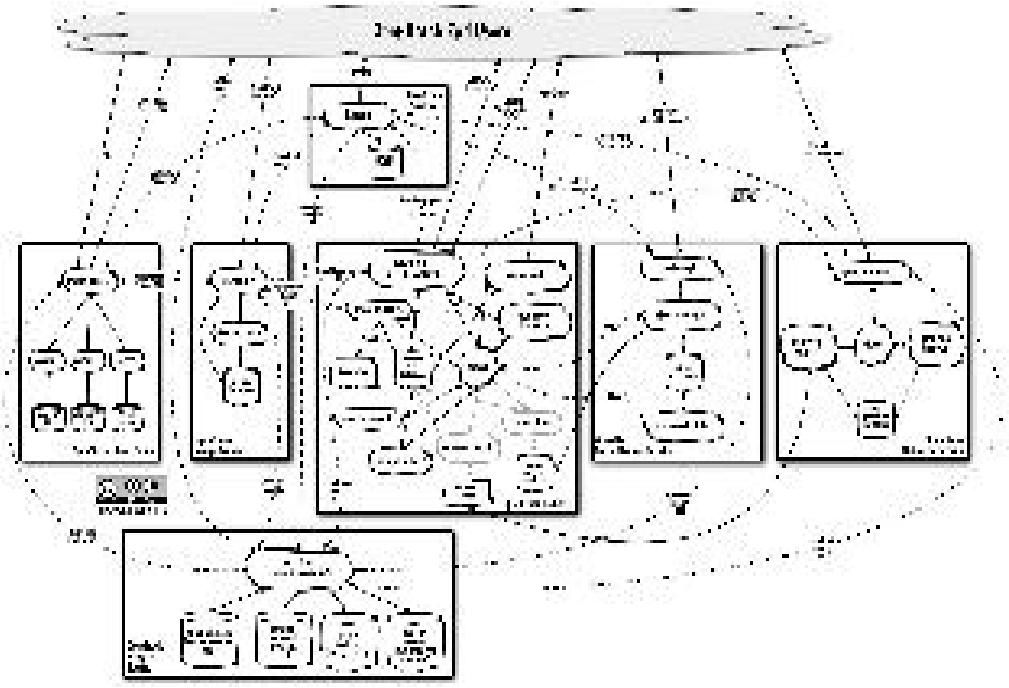
OpenStack Image Service,是一个虚拟机镜像的存储、查询和检索系统,服务包括的 RESTful API 允许用户通过 HTTP 请求查询 VM 镜像元数据,以及检索实际的镜像。VM 镜像有四种配置方式:简单的文件系统,类似 OpenStack Object Storage 的对 象存储系统,直接用 Amazon's Simple Storage Solution(S3)存储,用带有 Object Store 的 S3 间接访问 S3。

此三者关系如下:



各个组件的总体逻辑架构图如下：



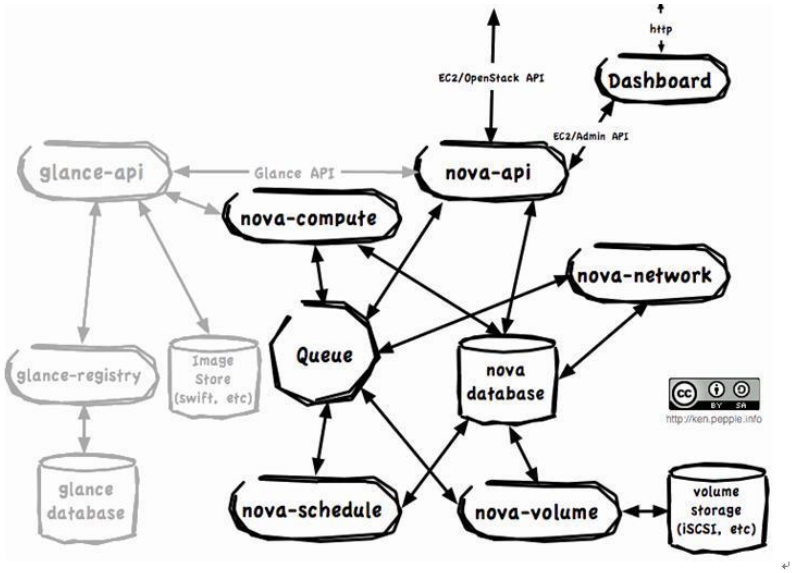


Nova 简介

Nova 是 OpenStack compute 组件的代号，是 OpenStack 的核心组件。主要负责云端计算资源的管理：如虚拟机的生命周期管理，硬件计算能力的分配，用户或 API 的请求响应，网络的配置,存储资源的管理。Nova 自身并没有提供任何虚拟化能力,相反它使用 libvirt API 来与被支持的 Hypervisors 交互。Nova 通过一个与 Amazon Web Services（AWS）EC2 API 兼容的 web services API 来对外提供服务。

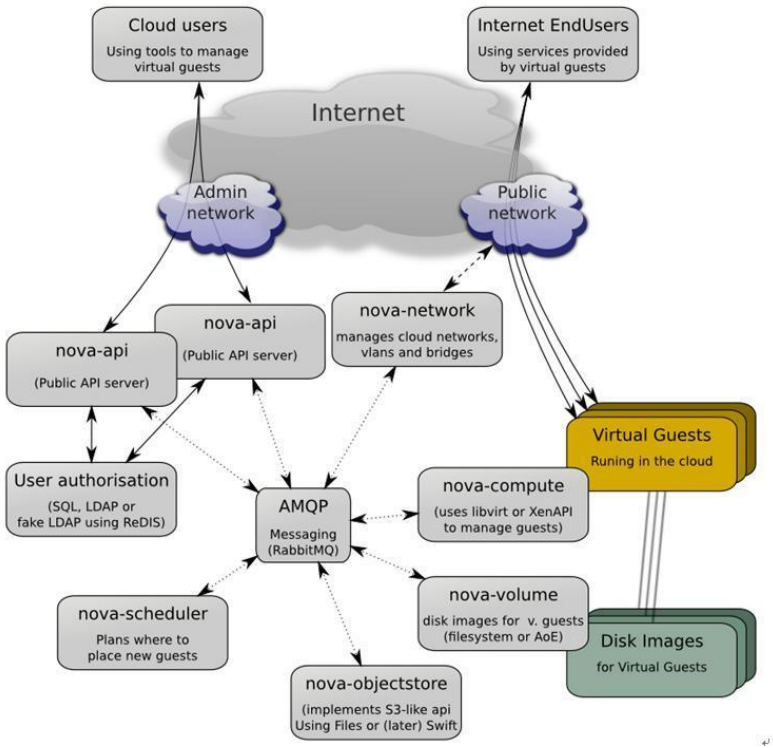
其开发宗旨如下：基于组件，高可用，容错，可恢复性好，开发标准，兼容已有 API。

Nova 主要由以下几个部件构成：Nova-API，Nova-Compute，Nova-Volume，Nova-Network，Nova-Scheduler 和 Message Queue。各个组件结构如图所示：



Nova 逻辑架构

OpenStack Compute 逻辑架构中的组件可分为两类：
a) 接收和协调 API 调用的 WSGI 进程(nova-api 等)
b) 执行部署任务的 Worker 守护进程(nova-compute, nova-network, nova-schedule 等)
另外还有两个不是基于 Python 的进程,它们是消息队列 Message queue 和数据库 database, 二者简化了复杂任务(通过消息传递和信息共享的任务)的异步部署。



1) nova-api WSGI 进程是 OpenStack Compute 的中心。它是 nova 的前台，负责接受和响应用户请求，为所有 API 查询(OpenStack API 或 EC2 API)提供端点。Nova-API 对外提供一个与云基础设施交互的接口，也是外部可用于管理基础设施的唯一组件。
2) nova-compute 守护进程主要是通过虚拟化 API 来创建,停止,迁移,挂起虚拟机实例 。其过程相当复杂,但是基本原理很简单: 从队列中接收请求,然后在更新数据库的状态时,执行一系列的系统命令执行他们。
3) nova-volume 守护进程负责存储功能,管理映射到虚拟机实例的卷的创建、挂载和卸载。现在被独立组件 cinder 替代。
4) Nova-network 守护进程从队列中接收网络任务,然后执行任务以操控网络,比如创建 bridging interfaces 或改变 iptables rules。现在被独立组件 quantum 替代。
5) Queue 提供中心 hub,为守护进程传递消息。当前用 RabbitMQ 实现。但是理论上能是 python ampqlib 支持的任何 AMPQ 消息队列。
6) SQL database 存储云基础架构中的绝大多数编译时和运行时状态。这包括了可用的实例类型,在用的实例,可用的网络和项目。理论上, OpenStack Compute 能支持 SQL-Alchemy 支持的任何数据库,但是当前广泛使用的数据库是 sqlite3, MySQL 和

PostgreSQL。

7) nova-consoleauth, nova-novncproxy, nova-console 允许终端用户通过 VNC 协议访问虚拟机实例。

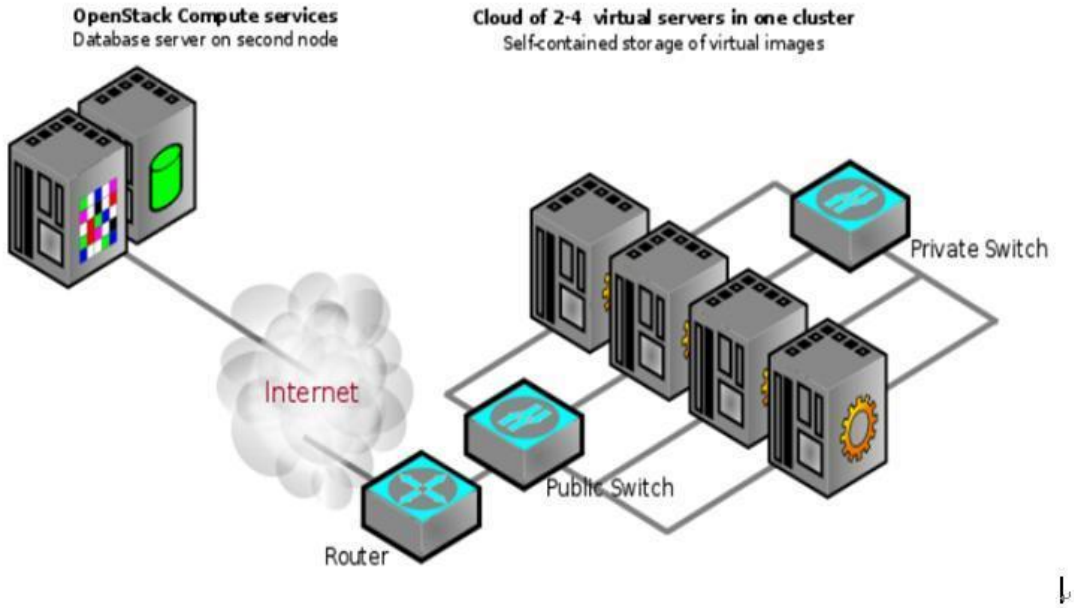
Nova 物理架构

OpenStack Compute 采用无共享、基于消息(shared nothing, message based)的架构,非常灵活,我们能安装每个 nova-service 在单独的服务器上,这意味着安装 OpenStack Compute 有多种可能的方法。可能多结点部署唯一的联合依赖性,是 Dashboard 必须被安装在 nova-api 服务器上。几种部署架构如下:

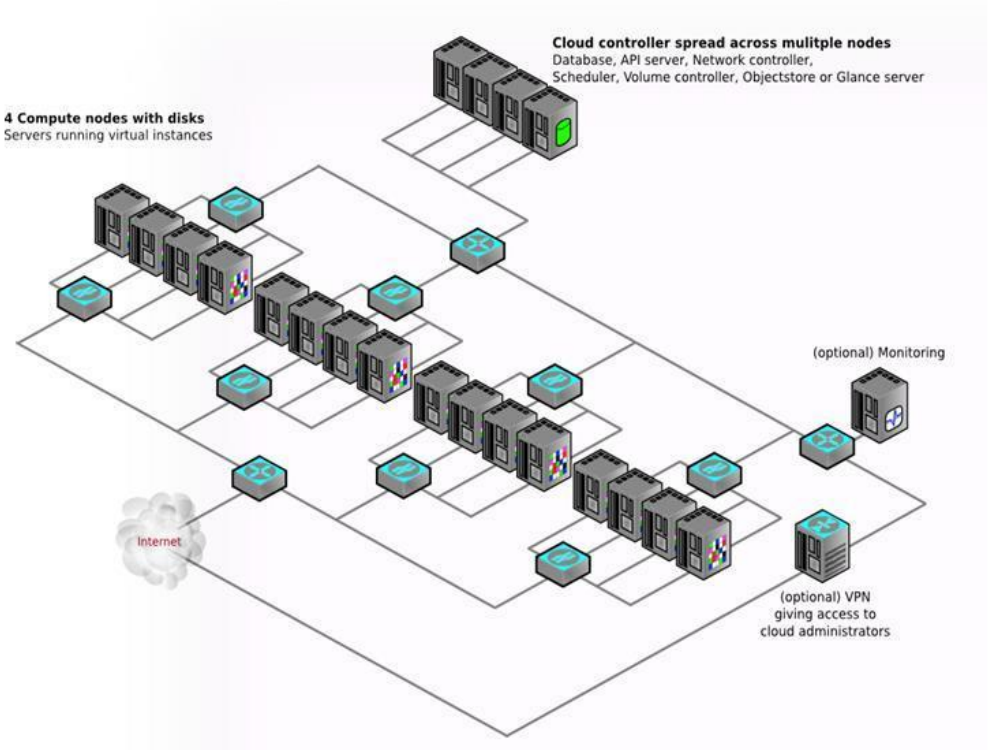
a) 单结点:一台服务器运行所有的 nova- services,同时也驱动虚拟实例。这种配置只为尝试 OpenStack Compute,或者为了开发目的;

b) 双结点:一个 cloud controller 结点运行除 nova-compute 外的所有 nova-services, compute 结点运行 nova-compute。一台客户计算机很可能需要打包镜像,以及和服务器进行交互,但是并不是必要的。这种配置主要用于概念和开发环境的证明。

c) 多结点:通过简单部署 nova-compute 在一台额外的服务器以及拷贝 nova.conf 文件到这个新增的结点,你能在两结点的基础上,添加更多的 compute 结点,形成多结点部署。在较为复杂的多结点部署中,还能增加一个 volume controller 和一个 network controller 作为额外的结点。对于运行多个需要大量处理能力的虚拟机实例,至少是 4 个结点是最好的。



如果你注意到消息队列中大量的复制引发了性能问题,一种可选的架构是增加更多的 Messaging 服务器。在这种情形下,除了可以扩展数据库服务器外,还可以增加一台额外的 RabbitMQ 服务器。部署中可以在任意服务器上运行任意 nova-service,只要 nova.conf 中配置为指向 RabbitMQ 服务器,并且这些服务器能发送消息到它


Cloud controller spread across mulitple nodes
Database, API server, Network controller,
Scheduler, Volume controller, Objectstore or Glance server
4 Compute nodes with disks
Servers running virtual instances
(optional) Monitoring
Internet
(optional) VPN
giving access to
cloud administrators

Nova 基本概念

（1）project/tenant

一个 project 就是一组被隔离的资源的组合,通常一个 project 都具有:

- 单独的 VLAN
- volumes
- instances
- images
- keys
- users

一个 project 一般都有自己的配额限制:

- 可创建的存储卷总数
- 存储卷总空间的大小(G)
- 可运行 VM 总数
- CPU 最大核数
- 公网 IP 最大数

为了和 EC2 保持一致,project 也叫 tenant,也就是说在 OpenStack 中,两者是一致的。

（2）user/role

角色	范围	职责
Cloud administrator	global	拥有整个云的所有权限
IT security	Global	仅限 IT 专业人员，可以隔离任何 project 的任何虚拟机
Project manager	Project	增加用户，操作镜像文件,运行/终止 VM
Network manager	Project	分配公网 ip,修改和创建 FW 规则
Developer	Project	普通用户

（3）区（zone）

由于性能较低以及硬件的限制(如一个子网中可用的交换机数),无法大规模部署。Zone 是一个资源管理的容器，一般是一组虚拟机实例的集合,可以映射到不同机房或者不同地区。每个 zone 都拥有独立的 Rabbitmq-server 、数据库、API Server.虚拟机可以进行不同 zone 之间的调度。

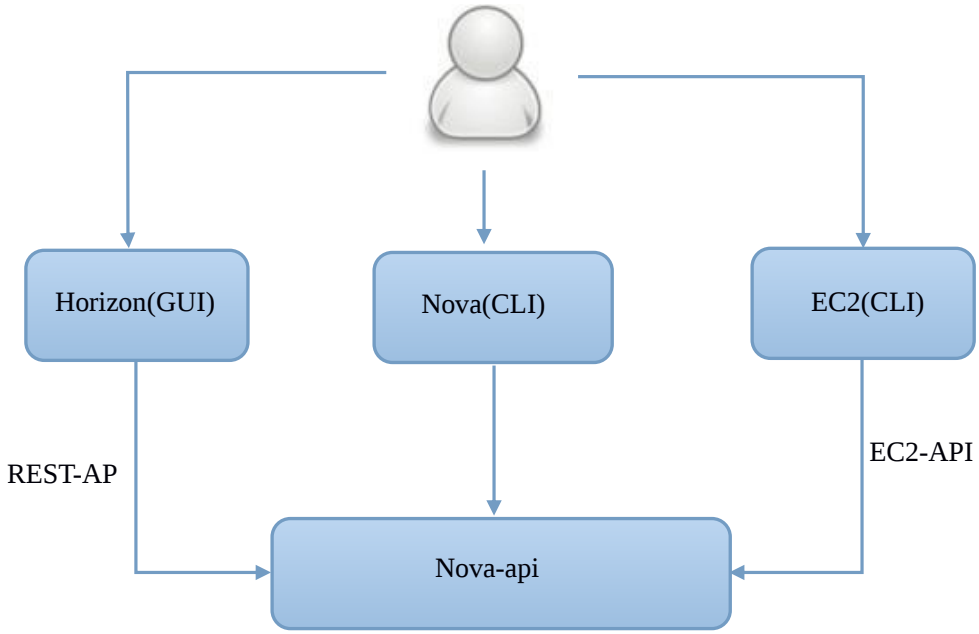
区可以嵌套，选择正确的 API 服务器来处理请求的方法如下：

1）请求从最顶级 Zone 进入,认证通过后,认证模块会递查询“管理该客户端的 Zone”,只要匹配成功即返回;

2）如果无响应,则由父区的 API 服务器处理
3）如果仅一个区响应,则返回该区的 API 服务器;
4）如果有多于 1 个区响应,则返回父区的 API 服务器。

Nova-api

nova-api 组件实现了 RESTful API 功能,是外部访问 Nova 的唯一途径。nova-api 接收外部的请求并通过 Message Queue 将请求发送给其他的服务组件。该组件也兼容 EC2 API,所以也可以用 EC2 的管理工具对 nova 进行日常管理。



（1）Nova-api 基本概念

Nova-api 基于 WSGI 技术来接受和处理用户请求，并且设计满足 REST 原则的 API，以下解释其基本概念。

（a) WSGI

WSGI 是 Python Web Server Gateway Interface 的缩写，即 python Web 服务器网关接口。它是 Python 应用程序和 Web 服务器之间的一种接口，已经被广泛接受。实现类似于 java servlet 的技术。WSGI 技术的出现，使得编写 network-based 应用程序更加简单。

WSGI 可以作为模块整合到 apache2 中使用，作用相当于 Servlet 的 Tomcat 容器。其处理流程如下：

1）客户端发送请求至 web 服务器

2）web 服务器启动并调用 WSGI，WSGI 根据客户端请求生成响应内容并将其传给 web 服务器

3）web 服务器将响应返回客户端

一个简单的 wsgi 程序示例：

```
"""Hello World using WSGI """
from paste import httpserver
```

```
def application(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/html')])
    return ['Hello World']
httpserver.serve(application, host='127.0.0.1', port=8080)
```

以上程序可以用 curl 工具测试，返回结果是“Hello World”。

(b) REST

REST，即 Representational State Transfer（表述性状态转移），是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。REST 是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。近年来已经成为最主要的 Web 服务设计模型，由于其使用相当方便，已经普遍地取代了基于 SOAP 和 WSDL 的接口设计。

REST 主要有以下原则：

1）为所有“事物”定义 ID

ID 采用 URI(Uniform Resource Identifier)来标识。不管是图片，Word 还是视频文件，甚至只是一种虚拟的服务，全部通过 URI 进行唯一的标识。

2）将所有事物链接在一起

任何可能的情况下，使用链接指引可以被标识的事物（资源）。通过 URI 以超链接形式将所有资源链接到一起，可以便捷访问。

3）使用 http 标准方法

应用程序知道如何处理 URI。原因在于所有的资源都支持同样的接口，一套同样的方法。在 HTTP 中叫做动词（verb），主要有 GET，POST，PUT、DELETE、HEAD 和 OPTIONS。这些方法的含义都定义在 HTTP 规范中，应用程序利用这些动词来解析操作。

4）资源多重表述

资源通过 xml 或者 json 格式描述，如果客户程序知道如何处理一种特定的数据格式，那就可以与所有提供这种表述格式的资源交互。

5）无状态通信

REST 要求服务器端不能保持单次请求之外的客户端状态。最直接的理由就是服务器保持客户端状态，大量的客户端交互会影响服务器的内存可用空间。同时，无状态约束使服务器的变化对客户端是不可见的，因为在两次连续的请求中，客户端并不依赖于同一台服务器。

以下提供遵从以上标准设计的 Compute api 的调用范例：

```
curl    -k
        -X ' POST '
        -v https://arm.trystack.org:5443/v2.0/tokens
        -d '{"auth":{"passwordCredentials":
                        {
                            "username": "joecool", "password":"coolword"
                        },
                        "tenantId":"5"
                    }
            }'
        -H 'Content-type: application/json'
```

其中 curl 命令将 -d, -H 的信息组合成 http 协议数据包，通过-X 指定的方法，发送到-v 指定的 https 地址，然后请求相应资源，获得返回。

（2）nova-api 原理

Nova-api 服务是一台基于 WSGI 的 API server，负责接收和处理用户的 API 调用，处于 apache 等 web 服务器的后端。

其调用过程如下：

a) 用户以 URI 形式发起 API 调用，请求资源，本质上是发起 http 请求，

b) web 服务器收到 http 请求，

c) 传递给请求给 nova-api 服务（也即 WSGI 服务器），

d) nova-api 解析得到需要执行的指令操作，

e) Nova-api 发送相应的消息至 AMQP 服务器（一般为 rabbitmq 服务器），

f) 相应组件检查自己的消息队列，收到消息完成指令操作，

g) 操作完成后将结果返回给 nova-api，

h) nova-api 将结果返回给服务器，

i) 再由服务器客户端。

（3）nova-api 调用方式

目前调用 nova-api 服务的方法主要有三种：

a)直接通过 http 构造相关数据包发生到指定的服务断点（endpoint）

如 curl -k -X 'POST' -v https://arm.trystack.org:5443/v2.0/tokens

b)安装 nova-pythonclient，该 client 可以自动封装命令和参数为 http 数据包，实现命令简洁的直接调用。

如 nova list

c)兼容 amazon 的 EC2 操作命令。

NOVA-Conductor

在 Grizzly 版的 Nova 中，取消了 nova-compute 的直接数据库访问。大概两个原因：

1. 安全考虑。

Benefit：因为 compute 节点通常会运行不可信的用户负载，一旦服务被攻击或用户虚拟机的流量溢出，则数据库会面临直接暴露的风险，增加 nova-conductor 更加安全。

Limitation：虽然 nova-conductor 限制了直接 DB 访问，但 compute 节点仍然可以通过 nova-conductor 获取所有节点的虚拟机数据，删除虚拟机等操作。特别是如果使用 E 版的 multihost，nova-api-metadata 和 nova-network 不能使用 nova-conductor。这样计算节点仍然是不安全的。

2. 方便升级。

Benefit：将 nova-compute 与数据库解耦的同时，也会与模式（schema）解耦，因此就不会担心旧的代码访问新的数据库模式。

Limitation：目前 rolling upgrade 在 G 版并不 ready

3. 性能

Benefit：当使用 green thread 时，所有的 DB 访问是阻塞的。而如果使用 nova-conductor，可以多线程使用 RPC 访问。

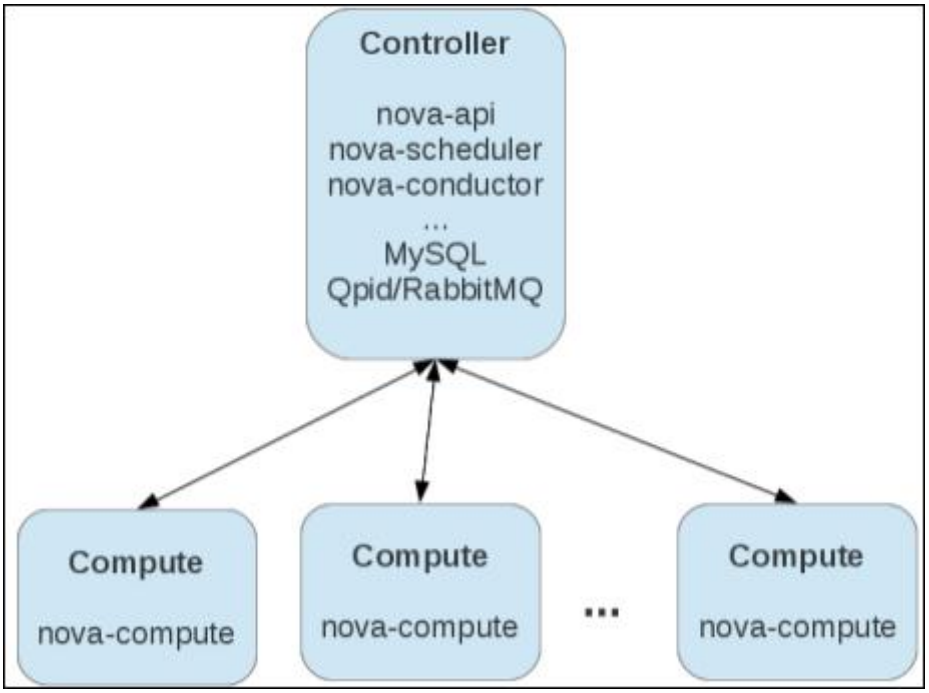
Limitation：通过 RPC 访问 nova-conductor 会受网络时延的影响。同时，nova-conductor 的 DB 访问也是阻塞的，如果 nova-conductor 实例较少，效果反而会更差

目前，nova-conductor 暴露的方法主要是数据库访问，但后续从 nova-compute 移植更多的功能，让 nova-compute 看起来更像一个没有大脑的干活者，而 nova-conductor 则会实现一些复杂而耗时的任务，比如 migration（迁移）或者 resize（修改虚拟机规格）

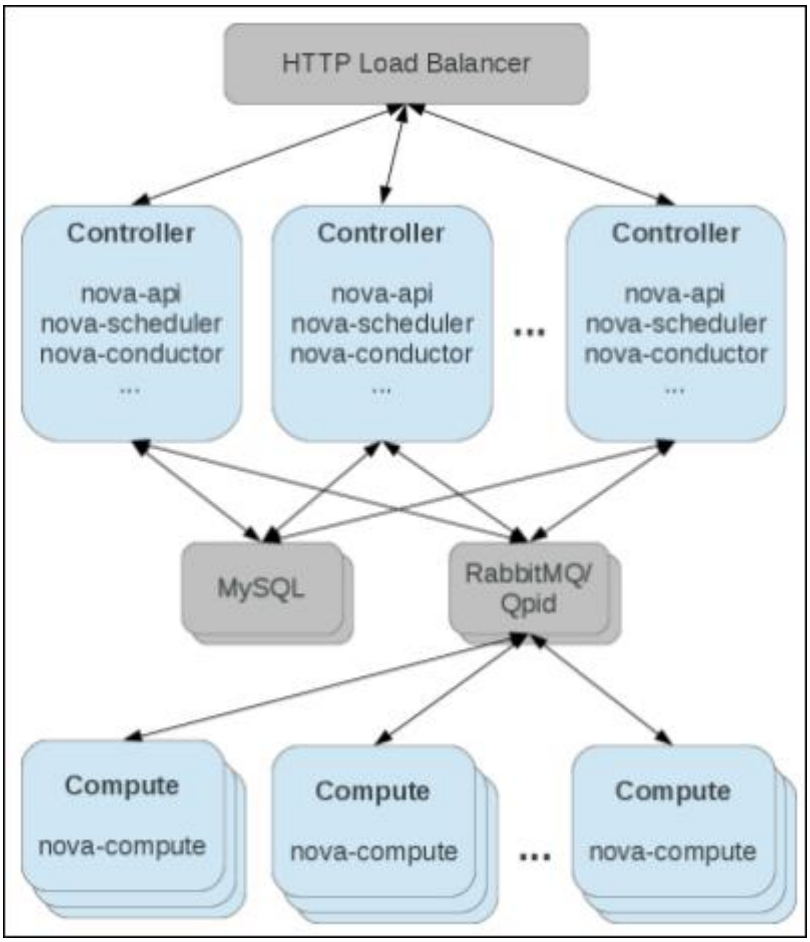
部署

nova-conductor 是在 nova-compute 之上的新的服务层。应该避免 nova-conductor 与 nova-compute 部署在同一个计算节点，否则移除直接数据库访问就没有任何意义了。同其他 nova 服务（nova-api, nova-scheduler）一样，nova-conductor 也可水平扩展，即可以在不同的物理机上运行多个 nova-conductor 实例。

经典的部署方式中（一个 controller 节点，多个 compute 节点），可以将 nova-conductor 运行在 Controller 节点



这也就意味着 Controller 节点将比以前运行更多的任务，负载变重，因此有必要对负载做监控，必要时需要扩展 contorller 服务。比如当在多个节点运行 nova-api 时，就需要在前端做负载均衡；多个节点运行 nova-scheduler 或 nova-conductor 时，负载均衡的任务就由消息队列服务器完成；

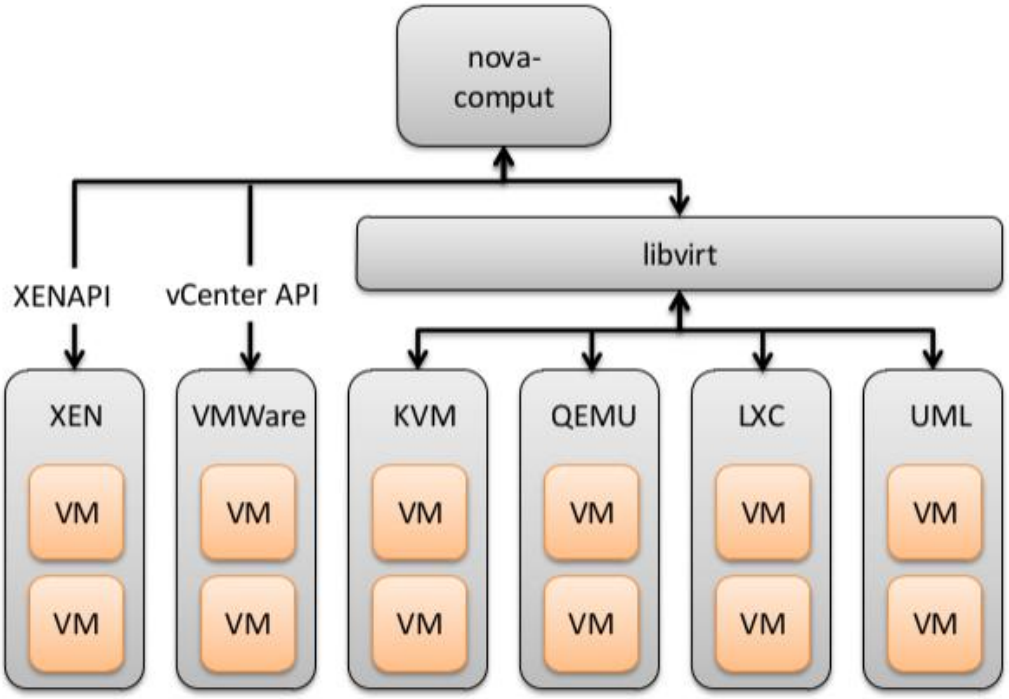


需要对 nova-conductor 进行性能监控以便决定是否对服务进行扩展。首先可以监控所在节点的 CPU 负载；其次可以监控消息队列中的消息个数及大小（对于 Qpid： qpid-stat，对于 RabbitMQ: rabbitmqctl list_queues）

Nova-compute

nova-compute 一般运行在计算节点上，通过 Message Queue 接收 VM 生命周期管理指令并实施具体的管理工作,如 VM 的创建、终止、迁移或 Resize 等操作。

Nova 本身并不提供虚拟化功能,但通过 libvirt API 的 python 绑定可对各种虚拟化平台进行管理，如图所示：



（1）libvirt 简介

Libvirt 是一套免费、开源的支持 Linux 下主流虚拟化平台的 C 函数库,其旨在为包括 Xen 在内的各种虚拟化平台提供一套方便、可靠的编程接口,支持与 C#, Java, Perl,Ruby,Python 等多种主流开发语言的绑定。当前主流 Linux 平台上默认的虚拟化管理工具 virt-manager(图形化),virt-install(命令行模式)等均基于 libvirt 开发而成。

Libvrit 支持多种平台的操作:

- The KVM/QEMU Linux hypervisor
- The Xen hypervisor on Linux and Solaris hosts.
- The LXC Linux container system
- The OpenVZ Linux container system
- The User Mode Linux paravirtualized kernel
- The VirtualBox hypervisor
- The VMware ESX and GSX hypervisors
- The VMware Workstation and Player hypervisors
- The Microsoft Hyper-V hypervisor

- Virtual networks using bridging, NAT, VEPA and VN-LINK.
- Storage on IDE/SCSI/USB disks, FibreChannel, LVM, iSCSI, NFS and filesystems

（2）python 对于 libvirt 库的绑定

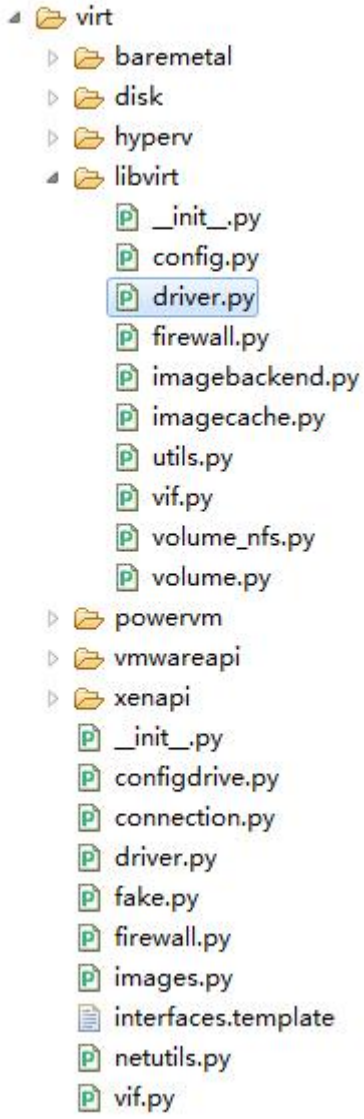
Openstack 中对于虚拟机的操作，均是通过调用 libvirt 的库函数来实现的。虽然 libvirt 实现为 c 语言函数库，但是提供了 python 语言绑定，可以直接调用。

在 python 中引入模块 libvirt 后就可以直接调用相关功能函数，部分函数示例如下：

C 函数	Python 函数	描述
Int virDomainDestroy (virDomainPtr domain)	virDomain::destroy(self)	销毁实例
Int virDomainGetAutostart (virDomainPtr domain, int * autostart)	virDomain::getAutoStart (self,autostart)	实例自动启动
Int virDomainHasCurrentSnapshot(virDomainPtr domain, unsigned int flags)	virDomain::hasCurrentSnapshot(self,flags)	获取快照
virDomainPtr virDomainMigrate (virDomainPtr domain)	virDomain::migrate	实例迁移

（3）Libvirt python-binding 类库分析

1, libvirt 的结构



2. Driver 类型

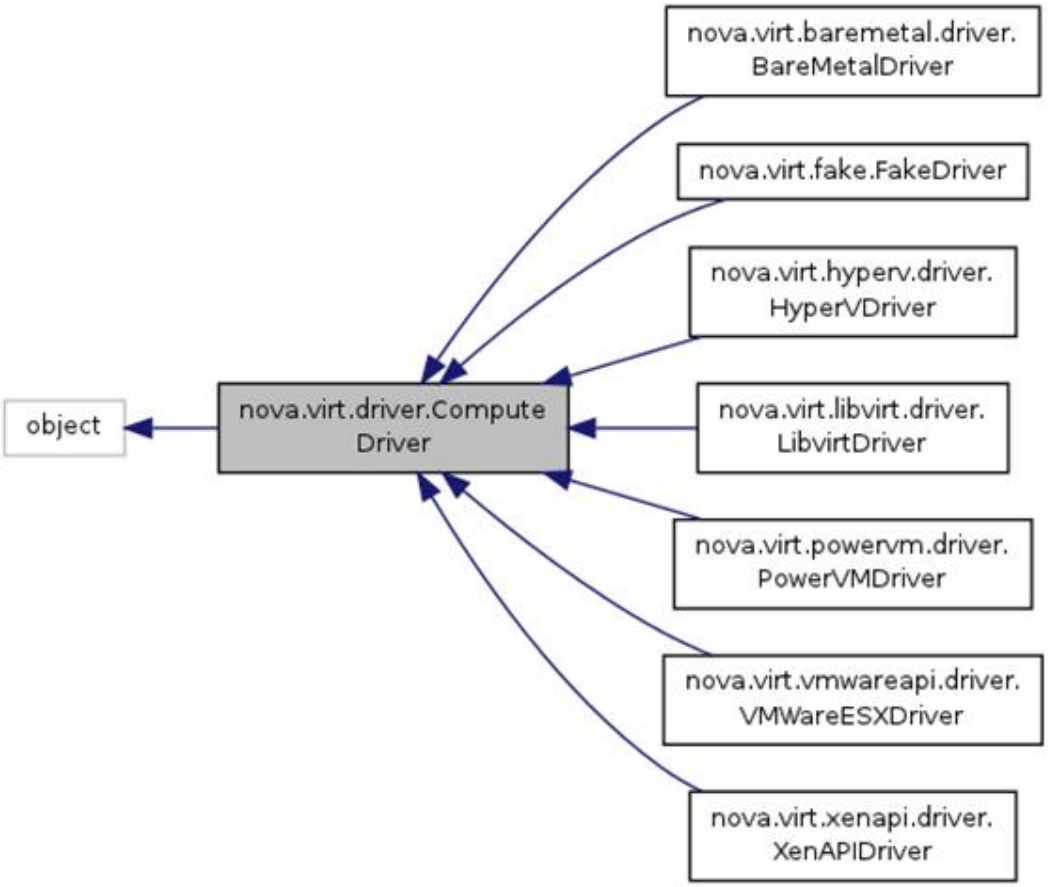
当前支持的虚拟化平台，主要有以下五种：

```
* fake
* libvirt
* xenapi
* vmwareapi
* baremetal
```

3. virt 继承关系

在 Folsom 版本中，对应不同的 hypervisor 实现，所有插件类全部继承于一个基类/nova/virt/driver.py，文件中定义到了所有需要 driver 支持的接口（当然有少部分接口是选配的），文件中仅有接口定义。

各个 driver 的继承关系如下：



4. libvirt Driver

4.1 有关 libvirt

目前 libvirt 下，一共支持 kvm、lxc、qemu、uml、xen 五种具体的 libvirt_type，默认选用 kvm。

虚拟机的状态映射如下：

VIR 内部状态	对外呈现状态	说明
VIR_DOMAIN_NOSTATE	NOSTATE	
VIR_DOMAIN_RUNNING	RUNNING	
VIR_DOMAIN_BLOCKED	RUNNING	仅在 Xen 中存在 Blocked 状态
VIR_DOMAIN_PAUSED	PAUSED	
VIR_DOMAIN_SHUTDOWN	SHUTDOWN	libvirt API 文档中已指出，此时 vm 可能仍旧处于 Running 状态
VIR_DOMAIN_SHUTOFF	SHUTDOWN	
VIR_DOMAIN_CRASHED	CRASHED	
VIR_DOMAIN_PMSUSPENDED	SUSPENDED	

4.2 libvirt 连接 hypervisor

整个 libvrit 的 driver.py 文件 非常长。初始化部分，首先调用基类 ComputerDriver 类中的__init__()，之后主要由配置文件读取了一些基本的配置项。

libvrit 接口由_conn = property(_get_connection) 来连接底层的 hypervisor。而实际的连接工作，由内部的一个静态方法 _connect()进行调用，一种是只读方式，另一种是拥有读写权限的，后一种需要用 root 身份登录。代码如下：

```
@staticmethod
    def _connect(uri, read_only):
        def _connect_auth_cb(creds, opaque):
            if len(creds) == 0:
                return 0
            LOG.warning(
                _("Can not handle authentication request for %d credentials")
                % len(creds))
            raise exception.NovaException(
                _("Can not handle authentication request for %d credentials")
                % len(creds))

        auth = [[libvirt.VIR_CRED_AUTHNAME,
                 libvirt.VIR_CRED_ECHOPROMPT,
                 libvirt.VIR_CRED_REALM,
                 libvirt.VIR_CRED_PASSPHRASE,
                 libvirt.VIR_CRED_NOECHOPROMPT,
                 libvirt.VIR_CRED_EXTERNAL],
                _connect_auth_cb,
                None]

        if read_only:
            return libvirt.openReadOnly(uri)
        else:
            return libvirt.openAuth(uri, auth, 0)
```

4.3 libvirt 创建 vm 实例

我们接着来看下 spawn()方法，这个是 libvirt 对外的创建 vm 接口。传入参数，都是由 Nova 处理过的虚拟机、镜像、网络、安全等方面的参数。

```
@exception.wrap_exception()
    def spawn(self, context, instance, image_meta, injected_files,
              admin_password, network_info=None, block_device_info=None):

        xml = self.to_xml(instance, network_info, image_meta,
                          block_device_info=block_device_info)
        self._create_image(context, instance, xml, network_info=network_info,
                           block_device_info=block_device_info,
                           files=injected_files,
```

```
                                        admin_pass=admin_password)
            self._create_domain_and_network(xml, instance, network_info,
                                                    block_device_info)
            LOG.debug(_("Instance is running"), instance=instance)

            def _wait_for_boot():
                """Called at an interval until the VM is running."""
                state = self.get_info(instance)['state']

                if state == power_state.RUNNING:
                    LOG.info(_("Instance spawned successfully."),
                             instance=instance)
                    raise utils.LoopingCallDone()

            timer = utils.LoopingCall(_wait_for_boot)
            timer.start(interval=0.5).wait()
```

方法入口处的 self.to_xml()方法，把之前传入的虚拟机、网络、镜像等信息，转化为创建 vm 时使用的 xml 描述文件。这个方法中，针对不同 libvrit_type（xen/kvm/qemu/…）做了大量的分别处理。方法最终走到 config.py 中，调用基类 LibvirtConfigObject 的 to_xml()方法， 生成描述文件，代码如下：

```
    def to_xml(self, pretty_print=True):
        root = self.format_dom()
        xml_str = etree.tostring(root, pretty_print=pretty_print)
        LOG.debug("Generated XML %s " % (xml_str,))
        return xml_str
```

回到 spawn()中，_create_image() 方法略过。接下来看下_create_domain_and_network()，用来设置卷映射、构建网络参数，最后创建 vm 实例。以下是方法的简明分析：

其中，在底层创建 vm 的动作，是在_create_domain()中完成的。它根据上一步生成好的 xml 的内容，定义一个虚拟机，相当于创建了一个 virDomain 对象，注意这时还没有启动这个虚拟机。

另外，其他需要 vm 的动作都是通过组装+调用该方法实现的。

```
    def _create_domain(self, xml=None, domain=None, launch_flags=0):
        """Create a domain.

        Either domain or xml must be passed in. If both are passed, then
        the domain definition is overwritten from the xml.
        """
        if xml:
            domain = self._conn.defineXML(xml)
        domain.createWithFlags(launch_flags)
        self._enable_hairpin(domain.XMLDesc(0))
        return domain
```

Nova-scheduler

Nova-scheduler 服务负责从集群中选择物理机来响应用户 API 调用。

主要有两个调度功能：

1）计算资源调度（compute）：根据当前计算资源使用状况及调度规则安排虚拟机实例在哪台物理机上运行；

2）存储资源调度（volume）：根据当前存储容量，数据流量负载，调度规则安排存储卷在那台物理机上创建。

（1）默认配置解析

默认做如下配置（nova.conf）：

```
scheduler_driver=nova.scheduler.multi.MultiScheduler
volume_scheduler_driver=nova.scheduler.chance.ChanceScheduler
compute_scheduler_driver=nova.scheduler.filter_scheduler.FilterScheduler
scheduler_available_filters=nova.scheduler.filters.all_filters
scheduler_default_filters=AvailabilityZoneFilter,RamFilter,ComputeFilter
least_cost_functions=nova.scheduler.least_cost.compute_fill_first_cost_fn
compute_fill_first_cost_fn_weight=-1.0
```

其中 scheduler 被配置为 multi-schduler，使得管理员可以根据具体需求为 compute 和 volume 配置不同的调度规则（过滤器）。

volume 默认配置为 chance scheduler，也即随机选取一台运行 cinder-volume 服务的机器处理请求。

Compute 调度默认配置为 Filter Scheduler。根据这个调度规则，调度器仅仅在用户指定的可用 zone(AvailabilityZoneFilter)内过滤出内存足够(RamFilter)的主机；从得到主机列表里，调度器根据各台主机的空闲内存来分别为其赋予代价权值，并乘以-1，然后选取代价最小的主机响应请求。也就是选择空闲内存最大的主机。

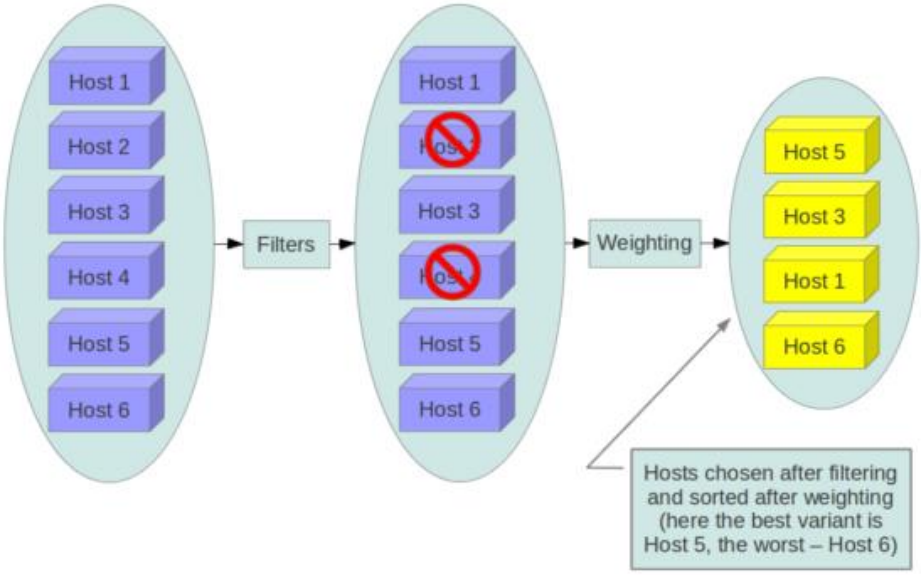
（2）调度原理

Nova-scheduler 作为一个后台进程运行,它会根据一定的算法从计算资源池中选择一个计算节点用于启动新的 VM 实例,其步骤主要有两步:

(1) 根据配置文件中定义的过滤器(filter)获得候选节点；

(2) 使用不同的算法加权计算获得代价，并选择代价最小的节点作为最终节点。

算法原理图示如下：

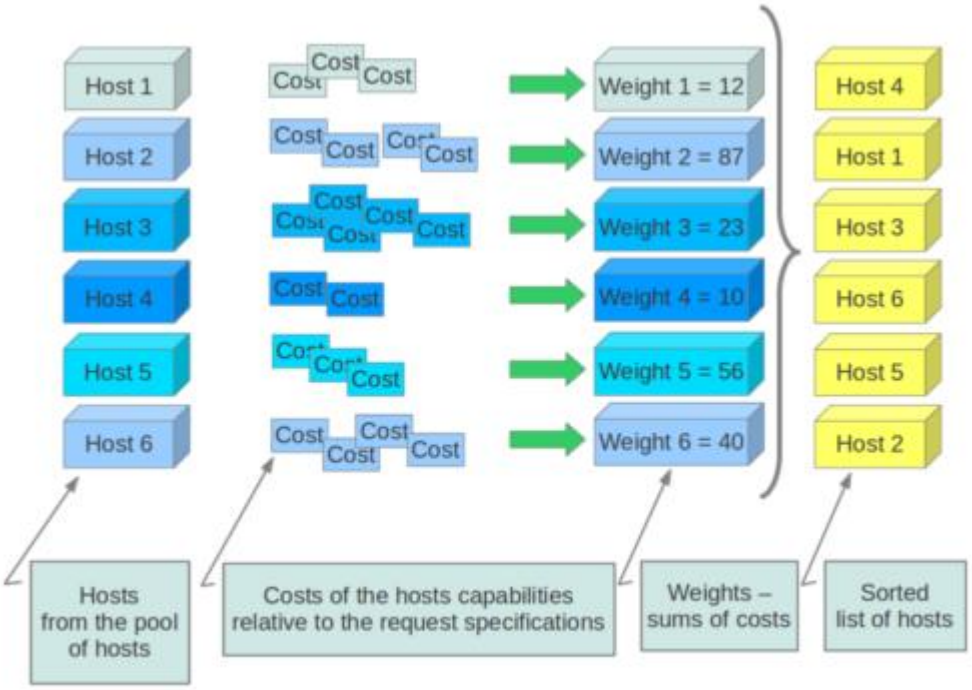


（1）过滤器

首先通过用户文件和命令参数里指定的过滤器得到可用的主机列表。可用的过滤器列表如下：

过滤器	描述
AllHostsFilter	不过滤,即全部节点均被命中
AvailabilityZoneFilter	指定区的节点
ComputeFilter	满足模板参数的节点，如体系结构，计算能力，支持的虚拟化类别
CoreFilter	cpu 核数满足条件的节点
RamFilter	筛选出空闲内存足够的节点
DifferentHostFilter	限定一组实例不能存在于同一节点
SameHostFilter	和同组中的其他实例存在于同一节点
IsolatedHostsFilter	某部分节点仅保留给某部分镜像使用
SimpleCIDRAffinityFilter	根据 IP 地址范围选择节点
AggregateInstanceExtraSpecsFilter	
AggregateMultiTenancyIsolation	限定某些主机只能运行特定 tenant 的虚拟机
ComputeCapabilitiesFilter	根据用户请求对主机的计算能力进行筛选，只有满足用户要求的主机才能够响应该请求
GroupAntiAffinityFilter	限定同一组的所有实例都必须位于不同物理机上
ImagePropertiesFilter	筛选出满足特定 image 特性的主机，比如虚拟化类型，cpu 类别。
JsonFilter	使得用户可以按照 json 格式自定义过滤器
RetryFilter	如果主机响应失败，则限定最多尝试次数。

（2）Weighting 算法



通过过滤器获得主机列表之后，nova 还需要通过评价算法在这些主机中选出最适合的主机来响应请求。

在 nova.conf 中可以设置不同的评价函数，以及相应的权值。示例如下：

```
least_cost_functions=nova.scheduler.least_cost.compute_fill_first_cost_fn
least_cost_functions=nova.scheduler.least_cost.noop_cost_fn
compute_fill_first_cost_fn_weight=-1.0
noop_cost_fn_weight=1.0
```

如果多个评价函数被赋予 least_cost_functions 选项，则 nova 将一台主机应用不同评价函数得到的权值进行加和，最后选择权值最小的主机响应请求。

Nova 提供的评价函数有三种：

函数名	描述
nova.scheduler.least_cost.compute_fill_first_cost_fn	该函数计算空闲内存的大小。
nova.scheduler.least_cost.retry_host_cost_fn	如果一台主机屡次被调用，则该函数增加其权值
nova.scheduler.least_cost.noop_cost_fn	该函数为所有主机返回 1

评价函数分析：

1) nova.scheduler.least_cost.compute_fill_first_cost_fn

该函数计算空闲内存大小。如果在 nova.conf 中设置 compute_fill_first_cost_fn_weight 为负数，则由于每次都会优先选取最大空闲内存的主机，从而造成 spread-first 现象。如果设

置 compute_fill_first_cost_fn_weight 为正数，则每次都会选择内存最小的那台主机，直到填满为止，造成 fill-up 现象。

2) nova.scheduler.least_cost.retry_host_cost_fn

为屡次尝试但响应失败的主机增加权值。一般设置其权重 retry_host_cost_fn_weight 为正值，从而使得被多次尝试并且出错的主机更少机会被选中。

3) nova.scheduler.least_cost.noop_cost_fn

为所有主机返回 1,因此起不到区分作用，实际中并不使用。

Nova VNC 访问原理

Nova 提供了访问虚拟机的方法，主要分为三类：

1) 直接通过命令行 ssh host-ip 命令访问。
2) 通过在虚拟机端启动 vncserver，在客户端启动 vnc-client 来访问。
3) SPICE HTML5，在后期版本中才加入，主要用于弥补 vnc 协议的限制和不足。

以下主要介绍 vnc 访问方式原理。

vnc 是 nova 提供的用来访问虚拟机的一项重要功能，用户可以通过 websocket 来访问，也可以通过 java 客户端来访问。通过 websket 访问虚拟机的功能已经集成到 horizon 中，而通过 java 客户端则需要先安装相应的软件。

Nova vnc 访问需要三个组件的支持：

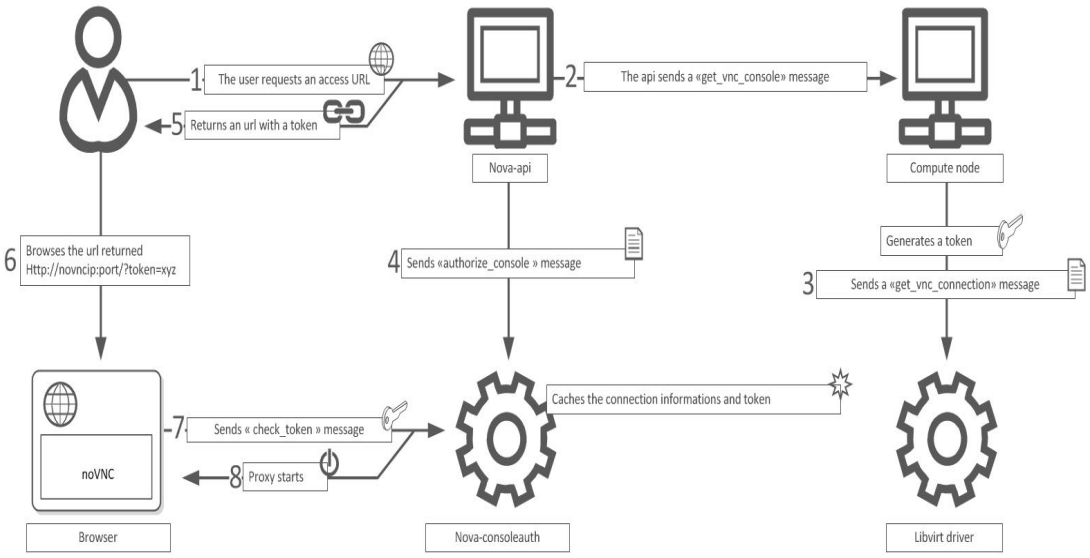
1) nova-consoleauth：验证用户传递的 token 是否合法，确认用户身份，维护 token 与 ip 地址、端口号的映射。
2) va-novncproxy：在虚拟机的 vncsever 和 vnc-client 之间充当代理，负责协议转换。Novncproxy 支持基于浏览器(websocket)的 vnc 客户端。通常与 nova-api 部署在一起。
3) nova-xvpvncproxy: 在虚拟机的 vncsever 和 vnc-client 之间充当代理，负责协议转换。xvpvncproxy 支持基于 java 的 vnc 客户端。通常与 nova-api 部署在一起。

（1）访问过程

通过基于浏览器的 novnc 访问虚拟机的实现方法如下，通过基于 java 的 xvpvnc 来访问虚拟机的方法与之类似:

1) 首先在启动一个虚拟机时启用 vnc，给 kvm 加上 vnc 参数即可。这样，kvm 就会启动一个 vncserver 监听虚拟机。
2) 用户调用 API 获取访问 url，url 的格式是：http://ip:port/?token=xxx，该地址实际上就是 vnc proxy 的地址。
3) 浏览器连接到 vnc proxy
4) vnc proxy 连接到 nova-consoleauth 来验证 token，并将 token 映射到虚拟机所在的宿主机的 ip 地址和某个端口，该端口就是虚拟机启动时所监听的端口。这个映射主要通过配置文件的 vncserver_proxyclient_address 选项完成，它指明 vnc proxy 应该通过哪个 IP 地址来连接 vncserver。
5) vnc proxy 与虚拟机所在的宿主机的 vncserver 建立连接，并开始代理，直到浏览器 session 结束。

该过程图示如下：



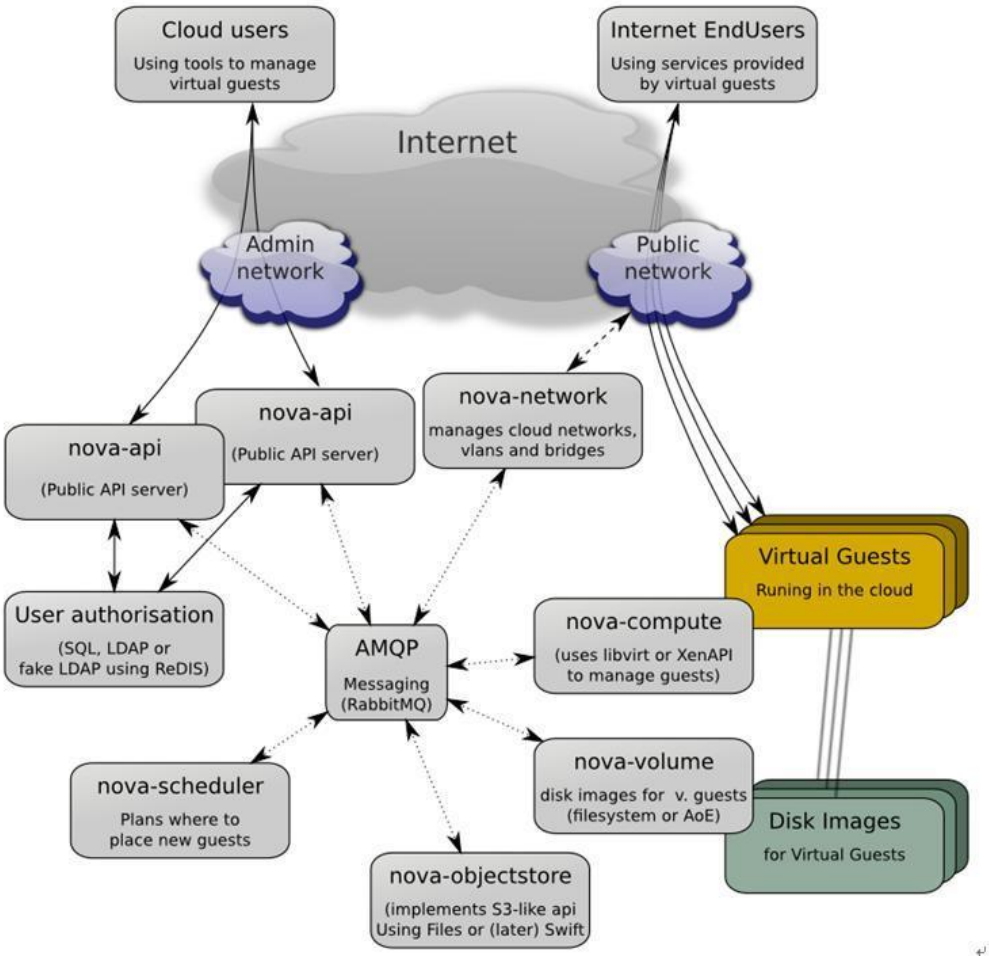
（2）vnc 配置方法

此外还需要对计算节点进行设当的配置。具体如下（nova.conf）:

选项	描述
vnc_enabled=True	启用虚拟机的 vnc 功能
vncserver_listen=0.0.0.0	默认是 127.0.0.1，即只可以从本机进行访问，通常情况下是配置为管理网的 IP 地址。设置为 0.0.0.0 主要是考虑到动态迁移时，目的宿主机没有相应的 IP 地址，动态迁移会失败。
vncserver_proxyclient_address	该地址指明 vnc proxy 应该通过那个 IP 地址来连接 vncserver，通常是管理网 IP 地址。
novncproxy_base_url=http://$SERVICE_HOST:6080/vnc_auto.html	指定浏览器 client 应该连接的地址。$SERVICE_HOST 通常是一个公网 IP 地址
xvpvncproxy_base_url=http://$SERVICE_HOST:6081/console	指定 java client 应该连接的地址。$SERVICE_HOST 通常是一个公网 IP 地址。
novncproxy_host=$SERVICE_HOS	支持基于浏览器的 noVnc proxy 监听主机地址
novncproxy_host=6080	支持基于浏览器的 noVnc proxy 监听主机端口
xvpvncproxy_host=$SERVICE_HOST	支持基于 java 的 xvpVnc proxy 监听主机地址
xvpvncproxy_port=6081	支持基于 java 的 xvpVnc proxy 监听主机端口

AMQP

Openstack 由多个组件构成，组件之间采用了“无共享，基于消息”(shared nothing, message based)的通信机制。Messaging Queue 提供中心 hub,为守护进程传递消息。当前用 RabbitMQ 实现。但是理论上能是 python ampqlib 支持的任何 AMPQ 消息队列。



（1）MQ 简介

MQ 全称为 Message Queue，消息队列。它是一种应用程序对应用程序的通信方法。应用程序通过写和检索出入队列的针对应用程序的数据（消息）来通信，而无需专用连接来链接它们。消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过 RPC 等远程调用技术直接调用彼此来通信。

MQ 的消费-生产者模型的一个典型的代表，一端往消息队列中不断的写入消息，而另一端则可以读取或者订阅队列中的消息。这种通信机制可以将一些无需即时返回且耗时的操作提取出来，进行异步处理，大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

（2）AMQP 协议简介

AMQP（advanced messaging queue protocol）即高级消息队列协议。使得遵从该规范的客户端应用和消息中间件服务器的全功能互操作成为可能

AMQP 模型描述了一套模块化的组件以及这些组件之间进行连接的标准规则。

在服务器中，三个主要功能模块连接成一个处理链完成预期的功能：

1) “exchange”接收发布应用程序发送的消息，并根据一定的规则将这些消息路由到“消息队列”。

2) “message queue”存储消息，直到这些消息被消费者安全处理完为止。

3) “binding”定义了 exchange 和 message queue 之间的关联，提供路由规则。

一个 AMQP 服务器类似于邮件服务器，exchage 类似于消息传输代理（email 里的概念），message queue 类似于邮箱。Binding 定义了每一个传输代理中的消息路由表，发布者将消息发给特定的传输代理，然后传输代理将这些消息路由到邮箱中，消费者从这些邮箱中取出消息。使用这个模型我们可以很容易的模拟出存储转发队列和主题订阅这些典型的消息中间件概念。

（3）典型消息传递流程

消息传递通信机制打破了原有的 comput, netwroking, scheduler 以及其他各个组件之间的直接调用，取而代之以统一的 http URI 形式的 API Endpoints。

一个典型的消息传递过程如下：

1) 首先 nova-api 收到来自用户的请求；

2) Nova-api 验证用户身份是否合法，解析出待执行的命令；

3) Nova-api 通过配置文件得到消息服务器（通常为 RabbitMq）的地址和端口，产生消息并通过 AMQP 协议发送到 RabbitMq 中对应 worker 守护进程的消息队列里；

各个 worker 持续监听自己的消息队列，得到消息后开始执行相关命令；在此期间，数据库 database 可能被访问并修改。

4) 完成后，worker 进程返回响应消息到 RabbitMq 中 nova-api 对应的消息队列里，由 nova-api 将响应封装成 http 数据包返回给用户。

应用实例分析

（1）实例启动过程

实例启动过程如下图所示。下面以虚拟机启动过程为例描述 nova 各个部件之间的工作：

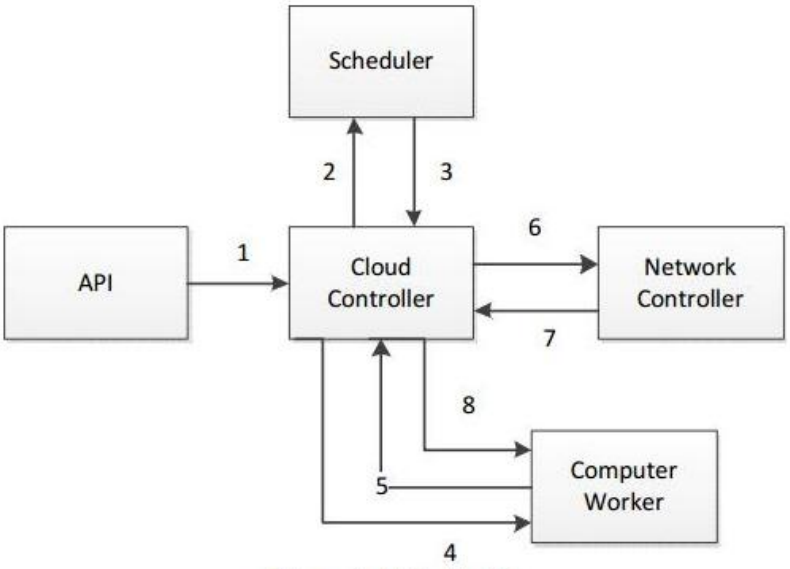


图 2 实例启动过程

1) 用户向 API Server 发送请求,有两种请求方式：

a) 通过 Opensta2 api(nova/apick api(nova/api/servers.py 类 class Controller(object)) create 方法；

b) 通过 ec2/cloud.py 中类 CloudController)调用

例如 def run_instances(self,context,**kwargs).

2) API Server 将请求消息发送到 Cloud Controller

在这个过程中会 Auth Manager 会进行鉴权以确保输入命令的用户拥有相应的权限。

3) Cloud Controller 将消息发送给 Scheduler

在 Openstack 中消息的分发全是通过 RabbitMQ 来进行的。

4) Scheduler 根据配置文件中写入的虚拟机部署算法选择对应于该消息的合适的物理机

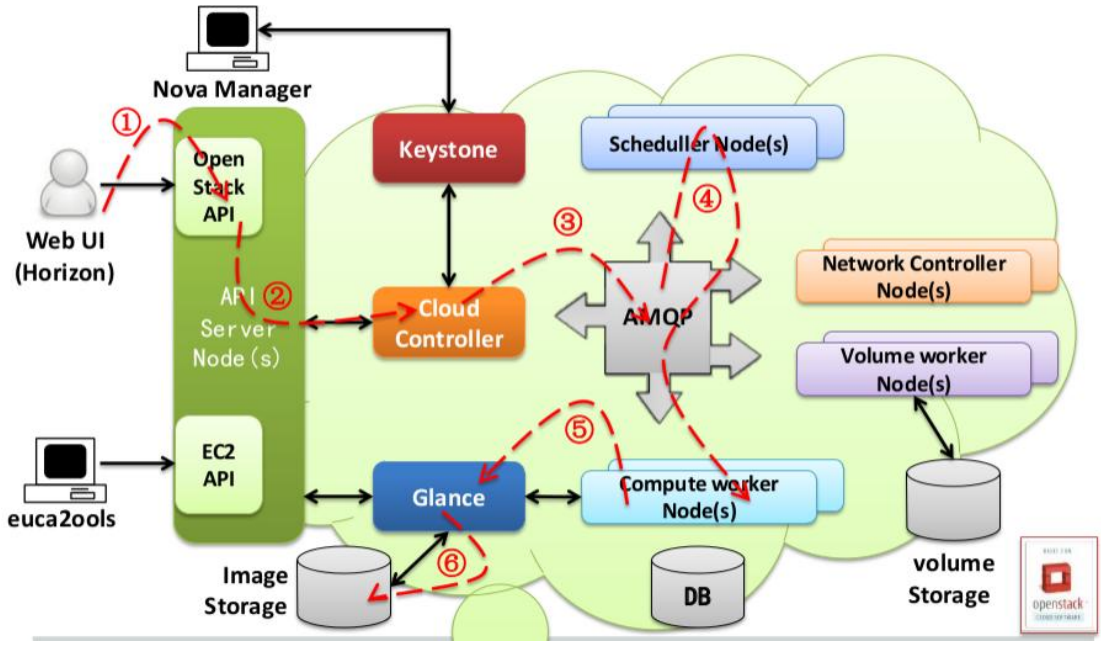
5) Cloud Controller 将这条消息发送到该物理机上。

6) 物理主机上的 Computer Worker 获取到这条消息,准备启动虚拟机时需要一个固定 IP；

7) Computer Worker 通过 Cloud Controller 向 Network Controller 请求分配一个固定 IP；

8) Computer Worker 获得 Network Controller 分配的 IP 从而继续虚拟机启动过程。

整体的下传递过程如下图所示：


Nova Manager
Web UI
(Horizon)
Open
Stack
API
API
Server
Node (s)
EC2
API
euca2ools
Keystone
Cloud
Controller
Glance
Image
Storage
Scheduller Node(s)
AMQP
Network Controller
Node(s)
Volume worker
Node(s)
Compute worker
Node(s)
DB
volume
Storage
①
②
③
④
⑤
⑥

源代码解析

（1）nova 服务的启动流程

Nova-api 中的服务分为两种：Service 和 WSGIService。
先看以下 nova-all 的源码：
建立进程启动器，绿化线程和相关的库，如图所示：

在 bin/nova-all 中 WSGIService 是由 Process Launcher 启动的。ProcessLauncher 启动一个总的进程，当 launcher_server 的时候，ProcesssLauncher 将会为所有的 Service 启动一个子进程，不管是 Service 还是 WSGIService。
接下来说明 WSGIService 是如何启动的，如图所示。

```
launcher = service.ProcessLauncher()
for api in flags.FLAGS.enabled_apis:
    server = service.WSGIService(api)
    launcher.launch_server(server, workers=server.workers or 1)
launcher.wait()
```

1 第一行调用了service中的ProcessLauncher生成了一个进程启动器

2 第二行使用了一个for循环从flags配置文件中读取需要启动的api

3 第三行利用这些api初始化WSGIService

```
self.name = name
self.manager = self._get_manager()
self.loader = loader or wsgi.Loader()
self.app = self.loader.load_app(name)
self.host = getattr(FLAGS, '%s_listen' % name, "0.0.0.0")
self.port = getattr(FLAGS, '%s_listen_port' % name, 0)
self.workers = getattr(FLAGS, '%s_workers' % name, None)
self.server = wsgi.Server(name,
self.app,
host=self.host,
port=self.port)
# Pull back actual port used
self.port = self.server.port
```

图 1.3

这些初始化主要是为了最后生成 self.server = wsgi.Server(name, self.app, host=self.host, port=self.port)。这个 server 在 ProcessLauncher 中将会被启动为一个子进程。其中 app 是 wsgi.Loader()从配置文件中读取过来的而 host，port，是从 FLAGS 中读出来的。

接下来我们看 wsgi.Server 的代码，如图所示。

```
【init】
self.name = name
self.app = app
self._server = None
self._protocol = protocol
#创建一个绿色线程池
self._pool = eventlet.GreenPool(pool_size or self.default_pool_size)
self._logger = logging.getLogger("nova.%s.wsgi.server" % self.name)
self._wsgi_logger = logging.WritableLogger(self._logger)
#用绿色线程去监听端口，生成一个socket
self._socket = eventlet.listen((host, port), backlog=backlog)
(self.host, self.port) = self._socket.getsockname()

【start】
self._server = eventlet.spawn(eventlet.wsgi.server,
self._socket,
self.app,
protocol=self._protocol,
custom_pool=self._pool,
log=self._wsgi_logger)
```

（2）Messaging Queue 源代码解析

Nova 中的每个组件都会连接消息服务器，一个组件可能是一个消息发送者（API、Scheduler），也可能是一个消息接收者（Compute、Volume、Network）。发送消息有两种方

式：同步调用(rpc.call)和异步调用(rpc.cast)。OpenStack 中默认使用 kombu（实现 AMQP 协议的 Python 函数库）连接 RabbitMQ 服务器。消息的收/发者都需要一个 Connetion 对象连接 RabbitMQ 服务器。

OpenStack 定义了以下几个类：

1.TopicConsumer。该对象在内部服务初始化时创建，在服务过程中一直存在。用于从队列中接收消息，调用消息属性中指定的函数。该对象通过一个共享队列或一个私有队列连接一个 topic 类型的交换器。每一个内部服务都有两个 topic consumer，一个用于 rpc.cast 调用（此时连接的是 binding-key 为“topic”的共享队列）；另一个用于 rpc.call 调用（此时连接的是 binding-key 为“topic.host”的私有队列）。类 TopicConsumer 源码如图 1.5 所示。

```
159 class TopicConsumer(ConsumerBase):
160     """Consumer class for 'topic'"""
161
162     def __init__(self, channel, topic, callback, tag, **kwargs):
163         """Init a 'topic' queue.
164
165         'channel' is the amqp channel to use
166         'topic' is the topic to listen on
167         'callback' is the callback to call when messages are received
168         'tag' is a unique ID for the consumer on the channel
169
170         Other kombu options may be passed
171         """
172         # Default options
173         options = {'durable': FLAGS.rabbit_durable_queues,
174                    'auto_delete': False,
175                    'exclusive': False}
176         options.update(kwargs)
177         exchange = kombu.entity.Exchange(
178                 name=FLAGS.control_exchange,
179                 type='topic',
180                 durable=options['durable'],
181                 auto_delete=options['auto_delete'])
182         super(TopicConsumer, self).__init__(
183                 channel,
184                 callback,
185                 tag,
186                 name=topic,
187                 exchange=exchange,
188                 routing_key=topic,
189                 **options)
```

图 1.5 类 TopicConsumer 源码

2.Direct Consumer

该对象在进行 rpc.call 调用时创建，用于接收响应消息。每一个对象都会通过一个的队列连接一个 direct 类型的交换器（队列和交换器以 UUID 命名）。类 DirectConsumer 源码如图 1.6 所示。

```
126 class DirectConsumer(ConsumerBase):
127     """Queue/consumer class for 'direct'"""
                                                20121114153517
128
129     def __init__(self, channel, msg_id, callback, tag, **kwargs):
130         """Init a 'direct' queue.
131
132         'channel' is the amqp channel to use
133         'msg_id' is the msg_id to listen on
134         'callback' is the callback to call when messages are received
135         'tag' is a unique ID for the consumer on the channel
136
137         Other kombu options may be passed
138         """
139         # Default options
140         options = {'durable': False,
141                 'auto_delete': True,
142                 'exclusive': True}
143         options.update(kwargs)
144         exchange = kombu.entity.Exchange(
145                 name=msg_id,
146                 type='direct',
147                 durable=options['durable'],
148                 auto_delete=options['auto_delete'])
149         super(DirectConsumer, self).__init__(
150                 channel,
151                 callback,
152                 tag,
153                 name=msg_id,
154                 exchange=exchange,
155                 routing_key=msg_id,
156                 **options)
```

图 1.6 类 DirectConsumer 源码

4.Topic Publisher

该对象在进行 rpc.call 或 rpc.cast 时创建，每个对象都会连接同一个 topic 类型的交换器，消息发送完毕后对象被回收。类 TopicPublisher 源码如图 1.7 所示。

```
272 class TopicPublisher(Publisher):
273     """Publisher class for 'topic'"""
274     def __init__(self, channel, topic, **kwargs):
275         """init a 'topic' publisher.
276
277         Kombu options may be passed as keyword args to override defaults
278         """
279         options = {'durable': FLAGS.rabbit_durable_queues,
280                 'auto_delete': False,
281                 'exclusive': False}
282         options.update(kwargs)
283         super(TopicPublisher, self).__init__(channel,
284                 FLAGS.control_exchange,
285                 topic,
286                 type='topic',
287                 **options)
```

图 1.7 类 TopicPublisher 源码

5.Direct Publisher

该对象在进行 rpc.call 调用时创建，用于向消息发送者返回响应。该对象会根据接收到的消息属性连接一个 direct 类型的交换器。类 DirectPublisher 源码如图 1.8 所示。

```
253 class DirectPublisher(Publisher):
254     """Publisher class for 'direct'"""
255     def __init__(self, channel, msg_id, **kwargs):
256         """init a 'direct' publisher.
257 
258         Kombu options may be passed as keyword args to override defaults
259         """
260 
261         options = {'durable': False,
262                 'auto_delete': True,
263                 'exclusive': True}
264         options.update(kwargs)
265         super(DirectPublisher, self).__init__(channel,
266                 msg_id,
267                 msg_id,
268                 type='direct',
269                 **options)
```

图 1.8 类 DirectPublisher 源码

6.Topic Exchange

topic 类型交换器，每一个消息代理节点只有一个 topic 类型的交换器。

7.Direct Exchange

direct 类型的交换器，存在于 rpc.call 调用过程中，对于每一个 rpc.call 的调用，都会产生该对象的一个实例。

消息发送时，消息发送方（Compute API）在 maxCount 为 1 的时候调用 RPC.cast 方法向 topic exchange(scheduler topic)发送启动实例的消息，消息消费者（nova-sheduler）从队列中获得消息，cast 调用不需要返回值。如图 1.9 所示，首先初始化一个 TopicPublisher,并将消息发送到消息队列；消息被交换器分发到 NODE-TYPE 消息队列（如图 1.9 中的 topic），对应服务节点的 TopicConsumer 从该队列中取出该消息，然后根据消息内容调用相应服务完成调用过程。

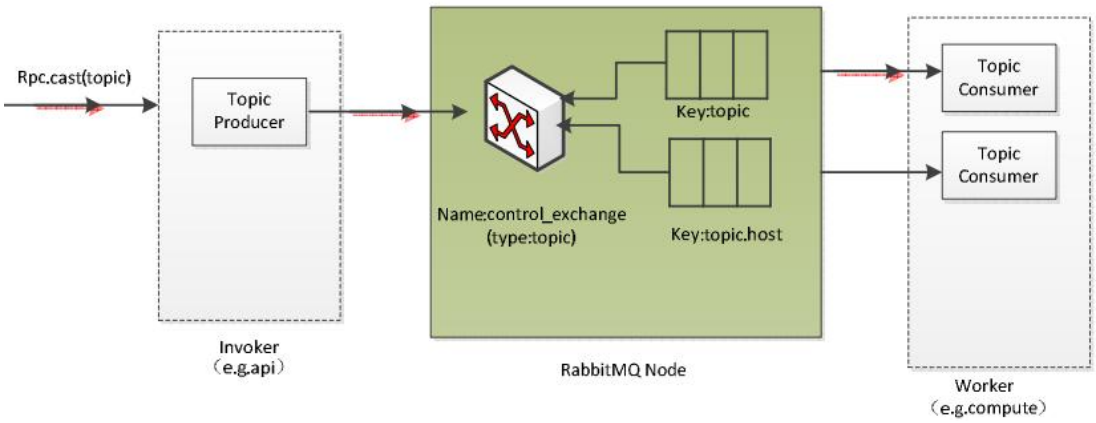


图 1.9 rpc.call 标准调用过程

（3）nova-compute 部分源代码解析

Nova-compute 负责对虚拟机实例进行创建，终止，迁移，资源重新分配的操作。工作原理可以简单描述为：从队列中接收请求，通过相关的系统命令执行他们，再更新数据库的

状态。

Nova-compute 启动分析可以按以下六步：

第一步：

```
utils.default_flagfile()
```

第二步：

```
flags.FLAGS(sys.argv)
```

flags.py 中有一句

FLAGS = FlagValues()，那么直接查看 FlagValues()这个类，这个类是继承于 gflags.FlagValues.

第三步：

```
logging.setup()
```

这个是对这个服务开启日志功能。

第四步：

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

这个函数位于/nova/service.py: 类 Service 中.

接下来对函数 Service.__init__()操作。

1、指定 host、binary、topic。

```
            self.host = host
            self.binary = binary
            self.topic = topic
```

2、动态指定 manager 类，并动态生成实例。

```
            self.manager_class_name = manager       #在 create 函数中指定。
            manager_class = utils.import_class(self.manager_class_name)   #动态地 import 此类。
            self.manager = manager_class(host=self.host, *args, **kwargs) #动态地生成这个类的实例 。
```

3、设置参数：应该是服务间隔时间之类的。

```
            self.report_interval = report_interval
            self.periodic_interval = periodic_interval
```

4、 设置多出来的一些参数。

```
            super(Service, self).__init__(*args, **kwargs)
            self.saved_args, self.saved_kwargs = args, kwargs
            self.timers = []
```

第五步：service.serve(server)开启服务

类 Launcher 主要作用是为每个服务开启一个线程。其中

```
server.start() #启动服务。
```

下面对 start 方法进行分析：

1、设置版本

```
            vcs_string = version.version_string_with_vcs()
            logging.audit(_('Starting %(topic)s node (version %(vcs_string)s)'),
```

{'topic': self.topic, 'vcs_string': vcs_string})

2、初始化 init_host(self): nova.compute.ComputeManager

3、接着从 service.start()接着 init_host()之后，得到 context.然后再更新当前机器上可用的资源。

4、更新 RPC 链接

第六步. Wait() 等待

Nova-Compute 启动的主要工作是把环境配置好。让 RqbbitMQ 的消息队列建立起来。最后服务的启动则主要是让 rabbitmq 的 consumer 运行。并且进入 wait 状态。消息的响应则主要是找到相应的函数地址，并执行之。

（4）nova-scheduler 部分源代码解析

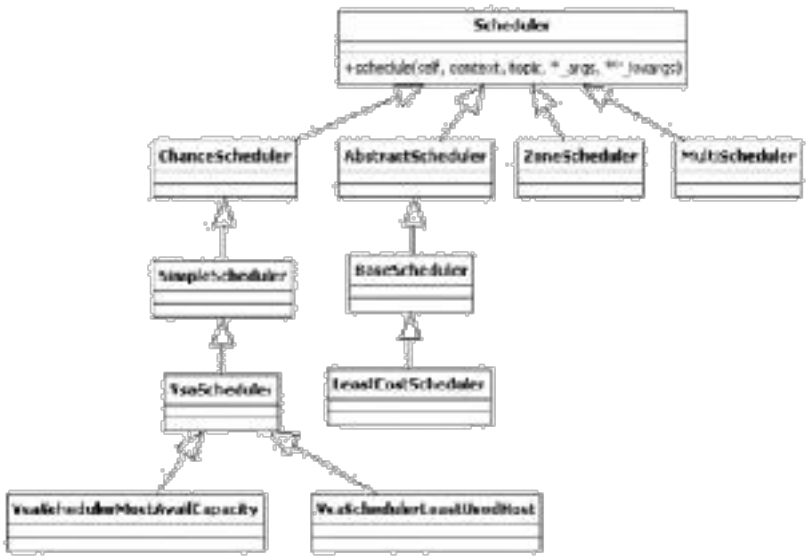
Nova-Scheduler 决定虚拟机实例应该运行在哪个计算结点（compute server host）上。但是，nova-scheduler 组件会渐渐变得复杂，因为它需要考虑当前的云基础平台的运行状态，并使用某些复杂的算法来达到资源的有效利用。为了这个目的，nova-scheduler 实现了可插拔的架构，让我们可以选择已有的算法或写出自己的算法。

通过源码分析，可以得到 nova-Scheduler 的工作流程如下：

1.nova-api 接到某个请求，例如“run_instance”，nova-api 将请求打包后发到 Topic 为“schedule”的队列。

2.队列接收到消息后，通知消费者（Consumer）。其实就是通过回调函数处理消息，这里的细节比较复杂，回调过程层层传递，我们只需要知道，最终会调用到 nova.scheduler.manager.SchedulerManager 的_schedule 函数。_schedule 函数有几个主要的参数：method、context、topic。

3.在 nova.scheduler.manager.SchedulerManager 的_schedule 函数中，SchedulerManager 首先尝试调用默认调度器的形如“schedule_*”（将*号替换成参数 method）的函数，如果没有找到，则调用调度器的 schedule 函数，选择一台主机。其中，Nova-Scheduler 的类图如图 1.10 所示。由图可知，scheduler 的所有调度器都是继承自 Scheduler 函数。



Scheduler 的调度算法简介

1) ChanceScheduler

ChanceScheduler 实现的是随机算法。

2) SimpleScheduler

(1)schedule_run_instance、schedule_start_instance

内部都调用了_schedule_instance 函数，选择已启动的并且运行实例数目最少的主机。

(2)schedule_create_volume

选择已启动的并且具有最小容量的主机。

(3)schedule_set_network_host

选择已启动的并且具有最少网络负载的主机。

(4)VsaSchedulerLeastUsedHost

继承 VSA scheduler，选择具有最少使用能力的主机。

3) VsaSchedulerMostAvailCapacity

继承 VSA scheduler，选择具有某种类型的最多可用容量的主机。

4) AbstractScheduler

(1) filter_hosts

过滤从 ZoneManager 返回的主机列表，只实现的简单的过滤，即满足请求所需内存的主机都被返回。

(2)weigh_hosts

对主机列表赋予权值。本类中只是将每个主机的权值都赋 1，更复杂的实现应由子类完成。

5) BaseScheduler

BaseScheduler 是创建跨域实例的调度器的基类。

BaseScheduler 覆写了父类 AbstractScheduler 中的 filter_hosts 和 weigh_hosts。

6) MultiScheduler

nova-scheduler 默认的 scheduler。MultiScheduler 内部包含了多个子调度器。所以原理上它可以根据不同的请求调用不同的调度器。但目前不同的请求调用的都是同一个调度器，即 ChanceScheduler。

（5）NOVA-api 分析

Nova-api 为所有 API 查询（OPENSTACK API 或 EC2 API）提供端点，初始化绝大多数的部署活动（如运行实例）

NOVA-API 脚本如下：

```
import eventlet
eventlet.monkey_patch()

import os
import sys


possible_topdir = os.path.normpath(os.path.join(os.path.abspath(
        sys.argv[0]), os.pardir, os.pardir))
if os.path.exists(os.path.join(possible_topdir, "nova", "__init__.py")):
    sys.path.insert(0, possible_topdir)


from nova import flags
from nova import log as logging
from nova import service
from nova import utils

if __name__ == '__main__':
    utils.default_flagfile()  #设置flag文件路径.
    flags.FLAGS(sys.argv)    #把flag文件中的参数放到args中。
    logging.setup()           #设置日志

    utils.monkey_patch()
    servers = []
    for api in flags.FLAGS.enabled_apis:
        servers.append(service.WSGIService(api)) #对外提供api服务
    service.serve(*servers)#创建并开始服务
    service.wait() #等待请求
```

（6）NOVA-compute 分析

Nova-compute 主要对内提供 rcp 调用。

```
"""Starter script for Nova Compute."""

import eventlet
eventlet.monkey_patch()

import os
import sys

# If ../nova/__init__.py exists, add ../ to Python search path, so that
# it will override what happens to be installed in /usr/(local/)lib/python...
POSSIBLE_TOPDIR = os.path.normpath(os.path.join(os.path.abspath(sys.argv[0]),
                                   os.pardir,
                                   os.pardir))
if os.path.exists(os.path.join(POSSIBLE_TOPDIR, 'nova', '__init__.py')):
    sys.path.insert(0, POSSIBLE_TOPDIR)


from nova import flags
from nova import log as logging
from nova import service
from nova import utils
```

```
if __name__ == '__main__':
    utils.default_flagfile()  #设置flag文件路径.
    flags.FLAGS(sys.argv)    #把flag文件中的参数放到args中。
    logging.setup()           #设置日志
    utils.monkey_patch()
    server = service.Service.create(binary='nova-compute')#对内提供rpc的调用
    #设置binary名字。
    #binary = os.path.basename(inspect.stack()[-1][1])
    #这是因为python可以查看动栈的内容。所以可以得到压入栈中的脚本的名字。
    #这时，binary="nova-compute"


    service.serve(server)
    service.wait()
```

这里调用的 service 是一个类，其定义在 service.py 中。而其中的 create 就是这个类的 一个类方法。我们可以打开 nova.service.py 查看 create()函数。其源码如下：

```
def create(cls, host=None, binary=None, topic=None, manager=None,
           report_interval=None, periodic_interval=None):
    """Instantiates class and passes back application object.

    :param host: defaults to FLAGS.host
    :param binary: defaults to basename of executable
    :param topic: defaults to bin_name - 'nova-' part
    :param manager: defaults to FLAGS.<topic>_manager
    :param report_interval: defaults to FLAGS.report_interval
    :param periodic_interval: defaults to FLAGS.periodic_interval

    """
    if not host:
        host = FLAGS.host
    if not binary:
        binary = os.path.basename(inspect.stack()[-1][1])
    if not topic:
        topic = binary.rpartition('nova-')[2]
    if not manager:
        manager = FLAGS.get('%s_manager' % topic, None)
    if not report_interval:
        report_interval = FLAGS.report_interval
    if not periodic_interval:
        periodic_interval = FLAGS.periodic_interval
    service_obj = cls(host, binary, topic, manager,
                      report_interval, periodic_interval)

    return service_obj
```

create 方法的流程如下：

1、设置 host 名字。

host = FLAGS.host

2、设置 binary 名字。

binary = os.path.basename(inspect.stack()[-1][1])

这是因为 python 可以查看动栈的内容。所以可以得到压入栈中的脚本的名字。

这时，binary="nova-compute"

3、设置 topic 的名字:也就是把 binary 的 nova-去掉。

topic = binary.rpartition('nova-')[2]

topic="compute"

4、设置 manager

manager = FLAGS.get('%s_manager' % topic, None)

很明显这里得到的是 compute_manager.

由于 FLAGS 是在 flag.py 中定义。所以可以在 flag.py 中得到如下定义;

DEFINE_string('compute_manager', 'nova.compute.manager.ComputeManager',

'Manager for compute')

由此可知 compute_manager 指的就是 nova.compute.manager.ComputeManager.

5、利用构造函数生成一个对象。

service_obj = cls(host, binary, topic, manager,

report_interval, periodic_interval)

Nova-volume/objectstore

Nova-network

Openstack 安装方案总结

Openstack 厂商支持

Openstack 安装脚本解析

Nova 基本概念
Nova-api
Nova-compute
Nova-scheduler
Libvirt-kvm
Nova-novnc
nova-consoleauth, nova-novncproxy, nova-console
Nova-volume/objectstore
Nova-network
源代码解析
Amqp
应用实例分析

Openstack 安装方案总结
Openstack 厂商支持
Openstack 安装脚本解析